Keithley KPCI-1801HC and KPCI-1802HC Register Level Description Document

By: T.L.Smith

December 17, 1999

Table of Contents

1. Scope

1.1 Introduction

This board has been designed for data acquisitions on the PCI bus. The board offers multi-channel analog input consisting of 64 Single ended or 32 Differential channels, two analog output channels, 4 bits of Digital IO, external triggering and gating on A/D output, a selection of 4 types of counter/timers (3 internal, 1 external).

1.2 1801/2HC PCI Configuration

This document does not go into great detail on the PCI bus protocol. An importance source or reference is the AMCC S5933 Matchmaker Data book. In order to understand some of the terminology in this document a brief description of how the board is configured will be provided. The board contains an NVRAM which holds the PCI configuration header space. On power up, the NVRAM initializes the board as a memory mapped device containing 4 pass through regions. The size of each region is contained in this header space along with the unique vendor and device ID. During power up, the active regions will be placed in the system's (PC) memory. The PCI BIOS will allow the proper amount of space specified in the header. These regions provide a way to assign a memory block for Add-On (FPGA) functionality. The table below shows the break down of the regions and the corresponding label from this document.

Table: *Base Address Breakdown*

Base Address	Content
0	AMCC Operation Registers
1	Analog to Digital
2	Digital to Analog
3	Counter/Timers, Digital IO, Interrupts

Instead of using the phrase "Base Address #", the truncated phrase of "BADDR#" will be used instead.

1.3 Convention

The representation of numbers will have a suffix to denote its base, "b": binary, "h": hexadecimal. For binary and hexadecimal numbers, the 'x' symbolizes that a bit or byte value is not relevant and the '?' symbolizes a valid but unknown bit or byte value. Bus representation will be: *bus name[msb:lsb]*. The bit # indexing always begins with 0.

2. Functionality

2.1 Analog to Digital

2.1.1 Features

The Analog to Digital design consists of Input Multiplexers, an Output FIFO, and a 12 bit A/D Converter (ADC) on board. Also internal logic is designed inside the Xilinx Spartan for A/D control. The board consists of 64 single ended / 32 differential input analog channels. These signals are provided through the 100 pin connector, J101. The transfer mode, sample rate, acquisition mode, data format, trigger setup, gain and channel control are all firmware/software selectable. The trigger setup for the A/D supports about, post and pre triggering.

2.1.2 Triggers

The trigger is an event that can be used to control the A/D acquisition. It also can be setup as an interrupt source (refer to section 2.4 for register configuration). The board supports digital and software triggering and a 24 bit counter. These triggers can be used together or separately to produce pre-, post-, and about triggering data. Each trigger will be described, followed by a section showing possible combinations for producing the pre-, post-, and about triggering.

2.1.2.1 24 bit Counter

The 24 bit Counter is started by the Digital Trigger. The count size is set by writing a single 24 bit value into BADDR1 + h14. There are two ways to reinitialize the counter. The first by disabling/reseting the A/D FIFO (BADDR1+h04 bit 1). The second by writing a new value into the counter. This 24 bit counter is used mainly with the Post Trigger setup. The table shows a configuration of the 24 bit Counter registers.

Address: BADDR1 +	Bit #	Function	Polarity
h10	12	Trigger Count Enable	1 – enabled / 0 - disabled
	Bits		Example / Description
h14	23:0	24 bit count value	h00 0064 : count value of 100

Ex. *Setting the trigger count to 10*

BADDR1 +h14	[31:0]	hxx00 000A
Enabling Trigger counter		
BADDR1 +h11	[7:0]	h1x

The counter can be set up to cause an interrupt (refer to section 2.4 on interrupts). The interrupt will occur once the counter reaches its final count.

2.1.2.2 Digital Trigger

The Digital Trigger/Gate signal (TGIN) is taken from digital input signal (DI1) from connector J101. This external signal can be used as a Digital Trigger/Gate or a General Purpose digital TTL input. The register located at the address below must be configured to properly select the function of DI1. The gating mode only allows points to be acquired by the A/D when the level matches the polarity selected. The trigger mode starts acquiring data with a rising edge when the polarity bit is set and with a falling edge when the bit is clear. The examples below show the bit patterns to select the Digital Trigger/Gate. Bits 11 and 8 MUST be set for the Digital Trigger to begin the A/D conversions.

Table: *Digital Triggering Bits*

Address: BADDR1 +	Bit #	Function	Settings
h10	11	Digital Trigger Enable	1 – on / 0 – off
	10	Digital Trigger Polarity	0 – ‘ – ‘ / 1 – ‘ + ‘
	9	Trigger/Gate Select	0 – trigger / 1 – gate
	8	Trigger Source	1 – digital / 0 – none

Ex1. *Enabling Digital Trigger to start A/D conversions with positive polarity and no counter*

```
BADDR1 +h11   [7:0]             hxD
```

Ex2. *Enabling Digital Trigger to start A/D conversions with negative polarity and counter*

```
BADDR1 +h11   [7:0]             h19
```

Ex3. *Enabling Digital Gate with positive polarity*

```
BADDR1 +h11   [7:0]             hx7
```

Ex4. *General Purpose / Digital TTL*

```
BADDR1 +h11   [7:0]             hx6 .. hx0
```

With Bit 11 and 8 cleared to zero, DI1 can be used as a General Purpose bit. Example 4 shows the range of bit patterns that would make DI1 General Purpose and disable the Digital Trigger.

2.1.2.3 Software Trigger

There is always some kind of trigger that needs to be performed in order for the A/D to start conversions. This section is about the Software Trigger. This is the default setting on power up. This will allow the ADC to sample and convert continuously upon receiving the Software Trigger. The Software Trigger is initiated by enabling the A/D FIFO bit shown below. When this bit is disabled, it clears the A/D FIFO and reinitializes the trigger count. This trigger is used to start and/or stop A/D conversions.

Table: *A/D FIFO Enable*

Address: BADDR1 +	Bit	Function	Polarity
h04	1	A/D FIFO Enable	0 – disabled / 1 - enabled

Ex1. *Software Trigger (start):*

```
BADDR1 +h11   [7:0]             hx6.. hx0
```

(refer to section 2.1.2.2 for bit descriptions)

```
BADDR1 +h04   [7:0]             hx2
```

Ex2. *Software Trigger (stop):*

```
BADDR1 +h11   [7:0]             hx6.. hx0
BADDR1 +h04   [7:0]             hx0
```

2.1.2.4 Pre-, Post-, and About Trigger Acquisitions

The are three modes to acquire data from a trigger. The Pre-Trigger applies to data taken prior to the trigger. The Post-Trigger consists of data occurring after the trigger. The About Trigger is data taken before and after the trigger. The sections below will explain configurations between the software and digital trigger for each acquisition mode.

Pre-Trigger:

There are two options for this setting. One is the start and stop trigger both being controlled by software. This is done by the software trigger above which toggles a bit controlling the A/D FIFO. The second is keeping the software trigger as the start trigger and having the stop trigger be the digital trigger with the 24 bit counter value set to 1.

Post-Trigger:

The two cases to acquire post trigger data are the Software Trigger and the Digital Trigger. The Digital Trigger, with the 24 bit counter enabled and count value set > 2, will enable the A/D to begin acquiring data and start the 24 bit counter counting. As for the Software Trigger, it will enable the A/D to start sampling. A counter in software must be setup to stop the A/D when its count is reached. The Software Trigger does not start the down counter. Example 4 under the Configuration Examples (section 2.1.9) shows this setup.

About-Trigger:

The About Trigger has only one configuration. The start trigger being the Software Trigger and the stop trigger as the Digital Trigger with the 24 bit counter enabled. The count value should be set to a value larger than 2. For the setup of the Digital Trigger, BADDR1+h10 bit 11 must be LOW. This will allow continuous A/D acquisition until a digital trigger has been detected. This will enable the counter to start counting and the A/D will stop sampling when the count has terminated. Just a note on the Digital Trigger, this signal can be unpredicable depending on the user's source.

2.1.3 A/D FIFO Status

These bits are only readable. They show the status of the FIFO. One can use these bits to determine if the FIFO has valid readings so to start reading data from the A/D.

Table: *FIFO Status bit*

Address: BADDR1 +	Bit #	Function	Polarity
h04	7	FIFO Full	0 – not full / 1 – full
	6	FIFO Half Full	0 – < half / 1 – > half
	5	FIFO Not Empty	0 – empty / 1 – not empty

2.1.4 Sample Rate

The sample clock is the rate the A/D samples data. The sample clock can be a counter/timer of the 8254, an external clock (XPCLK), or a software initiated write to the base address region offset zero. The register to select the sample clock source is located at address location BADDR1 + h5. For the 8254 counter/timers and software write, the A/D samples on the rising edge. As for the external clock, the polarity is selectable. Section 2.3 Counter/Timers will expand on the operation of the 8254 counter/timer. The 8254 register should be configured prior to the sample rate selection of any three of the 8254 counter/timer outputs. The example below sets the A/D sample clock source as an external clock having a positive polarity. The table below lists the available bit patterns for the sample clock source. Note: Bit patterns, b011 and b111, are invalid values. There are no sources available for those patterns.

Table: *Sample clock sources*

Address: BADDR1 +	Bits	Source Name	Bit pattern (b)
h04	[10:8]	8254 Counter 2 Output	000
		8254 Counter 1 Output	001
		8254 Counter 0 Output	010
		No source	*011*
		SW initiated write	100
		External clock, + polarity	101
		External clock, - polarity	110
		No source	*111*

Ex. *Sample clock as external clock (XPCLK) source*

BADDR1 +h05 [7:0] hx5 / bxxxx x101

2.1.5 Transfer Mode

There are two modes data can be transferred. The two modes are Pass Through (Target) and Bus Mastering (FIFO). The Pass Through mode is used to read and write to registers for board configuration and acquiring samples one at a time. It has modest transfer requirements and has lower demand on the PCI bus. A feature of Pass Through transfer that performs high performance applications is called bursting. In section 2.1.5, this term will be defined. Bus Mastering is used to acquire and process data as quickly as possible; like taking 1000 readings from the A/D. It is for high performance applications because it can attain peak transfer speeds.

The board is setup for 32 bit data transfers. In Bus Mastering mode, the 32 bit data will contain two 16 bit A/D samples. As for Pass Through mode, only the lower 16 bits will contain an A/D sample; the upper word would be masked with zeroes. So in Bus Mastering, an even number of counts would be needed to have the best performance. The table below specifies the address and bit polarity for setting the transfer mode.

Table: *Transfer Mode bit*

Address: BADDR1 +	Bit #	Mode	Polarity
h04	0	Target	0
		FIFO	1

2.1.6 Acquisition Mode

There are two acquisition modes to acquire data from the A/D. These two modes are Paced and Burst. Paced mode uses the sample rate, at each rising sample clock edge, to step through the Channel Queue RAM (section 2.1.7) acquiring a data value for each channel specified in the queue. As for the Burst mode, there is a 1 MHz clock source that gets divided by a burst count. In this mode, a value is taken from the Channel Queue on each rising edge of the burst frequency. The burst frequency starts its run at each rising sample clock edge. The burst count value which is used to divide down the 1 MHz clock needs to be written and observed with the queue's limit value (channel limit value) so there will be no data loss. The Channel Limit value (section 2.1.7) will determine the number of burst pulses (entry value conversions). If the Channel Limit value is equal to 0 (only 1 entry) while in Burst Mode, it will resemble Paced Mode. The table below specifies the address and bit polarity for setting the acquisition mode.

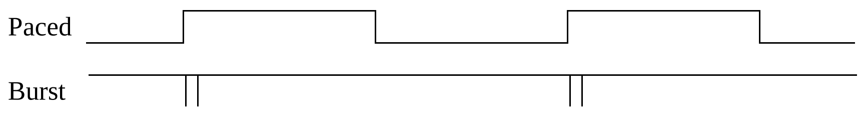
Table: *Acquisition Mode bit*

Address: BADDR1 +	Bit #	Mode	Polarity
h04	11	Paced	0
		Burst	1

Table: *Burst Count Value*

Address: BADDR1 +	Bits	Definition	Examples
h04	[23:16]	Burst count of 10, Produces a burst clock of 100 kHz.	h0A
		Burst count of 100, Produces a burst clock of 10 kHz.	h64

Diagram: *Paced and Burst clock behavior with a channel limit value = 1 (two entries):*



2.1.7 Data Format

The 12 bit A/D output returns a value that is represented in Offset Binary format. The FPGA design has the option to change the format from Offset Binary to 2's Complement output. This is done through software control. A bit has been declared for this purpose. The table specifies this bit at its address location where the software can control it.

Table: *Data format bit*

Address: BADDR1 +	Bit #	Mode	Polarity
h04	2	Stays Offset Binary	0
		To 2's complement	1

2.1.8 Channel Parameters

There are parameters a channel can be configured with. These parameters allow the user to select a channel with a specific input termination, polarity setting, and gain.
A channel can be set to either Unipolar or Bipolar range. The Unipolar range refers to positive voltage readings where the minimum value would be zero. As for Bipolar, it includes the entire voltage range. To allow this, the A/D input circuitry will offset the input signal to the range that is selected. For example, an input signal that has a range of +5 to –5 volts and is setup in Unipolar mode will result in a signal with a range of +10 to 0 volts. The table has the corresponding bit patterns for the channel selection along with its polarity.
There are 4 different gains that can be selected for a channel. The 1801HC board has gains of 1, 10, 50, and 250. The 1802HC board has gains of 1, 2, 4, and 8. The address location for these tables all begin at BADDR1+h4000.

Table: *1802HC Polarity and Gain bit patterns*

Address BADDR1 +	Bits	Description	Bit Pattern b... Polarity	Gain
h4000 to h40FF	[9:7]	Bipolar, Gain of 1	0	00
		Bipolar, Gain of 2		01
		Bipolar, Gain of 4		10
		Bipolar, Gain of 8		11
		Unipolar, Gain of 1	1	00
		Unipolar, Gain of 2		01
		Unipolar, Gain of 4		10
		Unipolar, Gain of 8		11

Table: *1801HC Polarity and Gain bit patterns*

Address BADDR1 +	Bits	Description	Bit Pattern b... Polarity	Gain
h4000 to h40FF	[9:7]	Bipolar, Gain of 1	0	00
		Bipolar, Gain of 10		01
		Bipolar, Gain of 50		10
		Bipolar, Gain of 250		11
		Unipolar, Gain of 1	1	00
		Unipolar, Gain of 10		01
		Unipolar, Gain of 50		10
		Unipolar, Gain of 250		11

The two modes of termination are Single Ended (SE) and Differential Ended (DE). Single Ended mode brings in the input channel as the positive feed while the negative feed is connected to analog ground. This mode gives the user 64 available input analog channels. Differential mode brings in a pair of input channel signals reflecting the positive and negative feeds mentioned above. This mode gives the user 32 available input analog channels.
The two muxs are fed with the same bit patterns shown below. The description contains the result from both muxs; the slash "/" will be used to separate each result.

Table: *Input Termination Selection*

Bits	Mode	Setting (b)
[6:5]	DE for CH0+/CH0- : CH31+/CH31-	00
	Short Ch32+/Ch32+ : Ch63+/Ch63+	01
	SE for CH0+:CH31+	10
	SE for CH32+:CH63+	11

Table: *Output Mux Bank Selection*

Mode	Bits [6:5] b..	Bits	Setting b..	Mux Bank Selection
DE	00	[4:3]	00	1
			01	2
			10	3
			11	4
SE Low Chs	10		00	1
			01	2
			10	3
			11	4
SE Hi Chs	11		00	5
			01	6
			10	7
			11	8

Table: *Channel bit patterns*

		Selection from each Mux Bank							
Bits	Bit Pattern b...	1	2	3	4	5 se,de	6 se,de	7 se,de	8 se,de
[2:0]	000	Ch0+	Ch8+	Ch16+	Ch24+	Ch32+, Ch0-	Ch40+, Ch8-	Ch48+, Ch16-	Ch56+, Ch24-
	001	Ch1+	Ch9+	Ch17+	Ch25+	Ch33+ , Ch1-	Ch41+ , Ch9-	Ch49+ , Ch17-	Ch57+ , Ch25-
	010	Ch2+	Ch10+	Ch18+	Ch26+	Ch34+ , Ch2-	Ch42+ , Ch10-	Ch50+ , Ch18-	Ch58+ , Ch26-
	011	Ch3+	Ch11+	Ch19+	Ch27+	Ch35+, Ch3-	Ch43+, Ch11-	Ch51+, Ch19-	Ch59+, Ch27-
	100	Ch4+	Ch12+	Ch20+	Ch28+	Ch36+, Ch4-	Ch44+, Ch12-	Ch52+, Ch20-	Ch60+, Ch28-
	101	Ch5+	Ch13+	Ch21+	Ch29+	Ch37+ , Ch5-	Ch45+ , Ch13-	Ch53+ , Ch21-	Ch61+ , Ch29-
	110	Ch6+	Ch14+	Ch22+	Ch30+	Ch38+ , Ch6-	Ch46+ , Ch14-	Ch54+ , Ch22-	Ch62+ , Ch30-
	111	Ch7+	Ch15+	Ch23+	Ch31+	Ch39+, Ch7-	Ch47+, Ch15-	Ch55+, Ch23-	Ch63+, Ch31-

Ex. 1802HC: *Select CH3, single ended, bipolar, with a gain of 4.*

```
BADDR1 +h4000     [31:0]      hx1 43
Break up of above result:
BADDR1 +h4000     [9:7]       b01 0    = bipolar, gain of 4
BADDR1 +h4000     [6:5]       b10      = single ended for channels 0 to 31
BADDR1 +h4000     [4:3]       b0 0     = mux bank 1
BADDR1 +h4000     [2:0]       b011     = channel 3
```

Ex. 1801HC: *Select CH53, single ended, bipolar, with a gain of 50.*

```
BADDR1 +h4000     [31:0]      hx1 75
Break up of above result:
BADDR1 +h4000     [9:7]       b01 0    = bipolar, gain of 50
BADDR1 +h4000     [6:5]       b11      = single ended for channels 32 to 63
BADDR1 +h4000     [4:3]       b1 0     = mux bank 7
BADDR1 +h4000     [2:0]       b101     = channel 53
```

A count value must be defined prior to the sample clock running through this queue. This count value is defined as the Channel limit value. The value for the limit can not exceed the maximum queue length of 64.

Table: *Channel Limit value*

Address: BADDR1 +	Bits Used	Definition	Setting
h08	[5:0]	1 Entry in Channel Queue	h00
		All 64 Entries in Channel Queue	h3F

2.1.9 Configuration Examples

The following examples give a variety of possible ways to configure the ADC to acquire samples. For all examples except the first, the 8254 counter/timer must be setup first. Refer to Section 2.3.1 for bit pattern description for the 8254 Counter/Timer.

Provided below with each example is C Code that were used toward a DOS32 executable. The complete files can be found in Appendix B. For the 8254 Setup, examples of code are located in Section 2.3.4.

Ex. *8254 Setup: Counter 0 source of 10MHz with counter value of 5, Counter 1 source of 5MHz with counter value of 10, Counter 2 source of 1MHz with counter value of 5.*

	Address			
Procedure	BADDR	Offset	Pattern	Description
PTWR32	3	h10	hx2x1x0	Selecting the source frequency for each counter.
PTWR8		h0C	h14	Format Counter 0
PTWR8		h00	h05	Counter 0 counter value
PTWR8		h0C	h54	Format Counter 1
PTWR8		h04	h0A	Counter 1 counter value
PTWR8		h0C	h94	Format Counter 2
PTWR8		h08	h05	Counter 2 counter value

Results of above action list would give on each counter's outputs the following frequencies:
Counter 0 Output = 2MHz; Counter 1 Output = 500KHz; Counter 2 Output = 200kHz.

Ex1. *One reading of Channel 0, differential mode, gain of 1, software initiated write as sample clock in Target mode.*

	Address			
Procedure	BADDR	Offset	Pattern	Description
PTWR8	1	h08	h00	Only 1 channel/configuration being used, channel limit value is zero based
PTWR32		h4000	h0000	Channel 0, Differential, Bipolar at a Gain of 1
PTWR8		h05	h04	Pacer clock source as SW initiated write to BADDR1+h0, paced acquisition selected
PTWR8		h04	h02	Offset Binary data, Target mode, A/D FIFO enabled
PTWR32		h00	hxx	SW initiated write – Pacer clock tick
PTRD32		h00	h????	Valid data from A/D FIFO only contained in lower word (Pass-Thru mode only)
PTWR8		h04	h00	Disables acquisition / Resets A/D FIFO

Taken from file: 180x_ad1.cpp

```
...
/*
**  File:   1801_ad1.cpp
**  By:     T.Smith
**  Date:   8/13/1999
*/

// The following header files are included with the Watcom Compiler.
#include <stdio.h>
#include <time.h>
#include "amcc.h"
#include "180xhc.h"
...

volatile dword *pAD_DATA[4];
volatile char *pAD_CHCNT[4];
volatile char *pTRIG_SOURCE[4];
volatile char *pDIG_TRIG[4];
volatile char *pTRIG_CNT[4];
volatile char *pAD_CONTROL[4];
volatile char *pAD_SCK_CONFIG[4];
volatile char *pAD_BURST_VAL[4];
volatile word *pAD_QRAM[4];
...

int get_board_info(void);
dword ad_virtual_read(int board_no);
void config_8254(int board_no);

//********************************** MAIN ***********************************

void  main(void)
{
    int board, Num_of_180xhcs;
    dword Data_Value;


    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("Using Board 0 for this test. \n");
        board = 0;

        printf("\nTHIS ROUTINE WILL SETUP & READ ONE READING FROM A/D CHANNEL0.\n");
        printf("A Virtual Write will be used as the Sample Clock.\n");

        // pointers to the A/D registers
        pAD_DATA[board] = (Baddr1[board].dp + 0x00);
        pAD_CHCNT[board] = (Baddr1[board].bp + 0x08);

        pTRIG_SOURCE[board] = (Baddr1[board].bp + 0x10);
        pDIG_TRIG[board] = (Baddr1[board].bp + 0x11);
        pTRIG_CNT[board] = (Baddr1[board].bp + 0x14);
        pAD_CONTROL[board] = (Baddr1[board].bp + 0x04);
        pAD_SCK_CONFIG[board] = (Baddr1[board].bp + 0x05);
        pAD_BURST_VAL[board] = (Baddr1[board].bp + 0x06);
        pAD_QRAM[board] = (Baddr1[board].wp + 0x2000);

        // pointers to the 8254 registers
        p8254_CNTRL0[board] = (Baddr3[board].bp + 0x10);
        p8254_CNTRL1[board] = (Baddr3[board].bp + 0x11);
        p8254_CNTRL2[board] = (Baddr3[board].bp + 0x12);
        p8254_FORMAT[board] = (Baddr3[board].bp + 0x0C);
        p8254_CNTR2_INITVAL[board] = (Baddr3[board].bp + 0x08);
        p8254_CNTR1_INITVAL[board] = (Baddr3[board].bp + 0x04);
        p8254_CNTR0_INITVAL[board] = (Baddr3[board].bp + 0x00);

//        printf("BADDR1 Address is 0x%lx. \n", Baddr1[board].dw);
//        printf("A/D CALRAM is located at 0x%lx. \n", pAD_CALRAM[board]);
//        printf("A/D QRAM is located at 0x%lx. \n", pAD_QRAM[board]);

        // Initializing trigger register for no triggers
        *pDIG_TRIG[board] = 0x00;

        // routine for configuring 8254
        config_8254(board);

        Data_Value = ad_virtual_read(board);

        printf("A/D reading = 0x%lx.\n", Data_Value);

    } // (else Num_of_180xhcs)

}
//********************************** END of MAIN ***********************************

// *************************** FUNCTIONS & ROUTINES *******************************

...
```

```
// *************** AD_VIRTUAL_READ *****************
// Setup A/D to take one reading.
// Using a SW Write for the pacer clock.
// The SW Write is also referred to as the Virtual Read.

dword ad_virtual_read(int board_no)
{
    #define DATA_MASK   0x0000FFFF
    #define FIFO_NEMPTY 0x20
    dword rtn_data;

    // Configuring for only one channel.
    *pAD_CHCNT[board_no] = 0x00;

    // Channel 0, Differential, Bipolar, Gain of 1
    *pAD_QRAM[board_no] = 0x0000;

    // Paced mode with SW Write/Virtual Write for pacer clock source
    *pAD_SCK_CONFIG[board_no] = 0x04;

    // Offset binary output, (Pass-Thru)Target mode, A/D Fifo enabled
    *pAD_CONTROL[board_no] = 0x02;

    // Triggering a conversion/sample clock tick
    *pAD_DATA[board_no] = 0x00;

    // wait for A/D fifo to be not empty
    // this will signify that there is data available in the FIFO
    while ( !(*pAD_CONTROL[board_no] & FIFO_NEMPTY) );

    // read data from A/D Data Register
    rtn_data = *pAD_DATA[board_no];
    rtn_data &= DATA_MASK;

    // clearing and disabling A/D FIFO
    *pAD_CONTROL[board_no] = 0x00;

    return rtn_data;
} // (int ad_virtual_read)
// ************************************************
```

Ex2. *Multiple readings of Channel 0, single ended mode, gain of 1, Counter Output 2 as sample clock in Target mode.*

Procedure	Address BADDR	Address Offset	Pattern	Description
PTWR8	1	h08	h00	Only 1 channel/configuration being used, channel limit value is zero based
PTWR32		h4000	h0240	Channel 0, Single Ended, Unipolar at a Gain of 1
PTWR8		h05	h00	Pacer clock source as 8254 Counter Output 2, paced acquisition selected
PTWR8		h04	h02	Offset Binary data, Target mode, A/D FIFO enabled
PTRD32		h00	h????	Valid data from A/D FIFO only contained in lower word (Pass-Thru mode only)
LOOP PTRD32		h00		
PTWR8		h04	h00	Disables acquisition / Resets A/D FIFO

Taken from file: 180x_ad2.cpp

```
...
/*
**  File:   1801_ad2.cpp
**  By:     T.Smith
**  Date:   8/13/1999
*/
...
```

```
volatile dword *pAD_DATA[4];
volatile char *pAD_CHCNT[4];
volatile char *pTRIG_SOURCE[4];
volatile char *pDIG_TRIG[4];
volatile char *pTRIG_CNT[4];
volatile char *pAD_CONTROL[4];
volatile char *pAD_SCK_CONFIG[4];
volatile char *pAD_BURST_VAL[4];
volatile word *pAD_QRAM[4];
...

volatile dword data_array[100];
...

int get_board_info(void);
void ad_multi_ptread(int board_no, int cnt);
void config_8254(int board_no);

//********************************** MAIN ************************************

void  main(void)
{
    int board, Num_of_180xhcs, no_reads;
    int count = 0;

    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("Using Board 0 for this test. \n");
        board = 0;
        printf("\nTHIS ROUTINE WILL SETUP & READ 50 READINGS FROM A/D CHANNEL0 IN PASS
THRU.\n");
        printf("Counter Output 2 used as Sample Clock with a frequency of 100kHz.\n");
        no_reads = 50;

        // pointers to the A/D registers
        pAD_DATA[board] = (Baddr1[board].dp + 0x00);
        pAD_CHCNT[board] = (Baddr1[board].bp + 0x08);

        pTRIG_SOURCE[board] = (Baddr1[board].bp + 0x10);
        pDIG_TRIG[board] = (Baddr1[board].bp + 0x11);
        pTRIG_CNT[board] = (Baddr1[board].bp + 0x14);
        pAD_CONTROL[board] = (Baddr1[board].bp + 0x04);
        pAD_SCK_CONFIG[board] = (Baddr1[board].bp + 0x05);
        pAD_BURST_VAL[board] = (Baddr1[board].bp + 0x06);
        pAD_QRAM[board] = (Baddr1[board].wp + 0x2000);

        // pointers to the 8254 registers

        ...

        // Initializing trigger register for no triggers
        *pDIG_TRIG[board] = 0x00;

        // routine for configuring 8254
        config_8254(board);

        // routine for taking multiple readings
        ad_multi_ptread(board, no_reads);

        printf("Storing data into file: PT50rdgs.txt !\n");

        // Storing data into file
        FILE *fp = fopen("PT50rdgs.txt", "w");
        if (fp != NULL)
        {
            fprintf(fp, "PT50rdgs.txt\n");
            fprintf(fp, "\nOutput from A/D's ptmulti.exe (a DOS32 executible). \nTaking
50 readings at 100kHz.\n\n");
```

```
            for (; count < no_reads; count++)
            {
                fprintf(fp, "A/D reading %i= 0x%lx.\n", count, data_array[count]);
//                printf("A/D reading %i= 0x%lx.\n", count, data_array[count]);
            } // for

            fprintf(fp, "-------------------------------------------------------- \n");
            fclose(fp);
        } // if (fp != NULL)

    } // (else Num_of_180xhcs)

}
//********************* END of MAIN **********************

...

// *************** AD_MULTI_PTREAD *****************
// Setup A/D to take 50 readings.
// Using the 8254 counter2's output for the pacer clock.
// In pass through mode.

void ad_multi_ptread(int board_no, int cnt)
{
    #define DATA_MASK   0x0000FFFF
    #define FIFO_NEMPTY 0x20
    #define FIFO_HALFFULL 0x40
    byte not_empty_status;
    int i = 0;

    // Configuring for only one channel.
    *pAD_CHCNT[board_no] = 0x00;

    // Channel 0, Differential, Bipolar, Gain of 1
    *pAD_QRAM[board_no] = 0x0000;

    // Paced mode with 8254 Counter2's Output for pacer clock source
    *pAD_SCK_CONFIG[board_no] = 0x00;

    // Offset binary output, (Pass-Thru)Target mode, A/D Fifo enabled
    *pAD_CONTROL[board_no] = 0x02;

    // wait for A/D fifo to be half full before reading FIFO
    while ( !(*pAD_CONTROL[board_no] & FIFO_HALFFULL) );

    for (; i< cnt; i++)
    {
        // read data from A/D Data Register
        data_array[i] = *pAD_DATA[board_no];
        data_array[i] &= DATA_MASK;
    }// for

    // clearing and disabling A/D FIFO
    *pAD_CONTROL[board_no] = 0x00;

} // (ad_multi_ptread(int board_no, int cnt)
// ************************************************
```

Ex3. *Multiple readings of Channel 0, single ended mode, gain of 100, Counter Output 1 as sample clock in Bus Mastering mode.*

Procedure	Address BADDR	Address Offset	Pattern	Description
PTWR8	1	h08	h00	Only 1 channel/configuration being used, channel limit value is zero based
PTWR8		h4000	h0240	Channel 0, Single Ended, Unipolar at Gain of 1
PTWR8		h04	h01	Bus Mastering mode, offset binary
PTWR8		h05	h00	Pacer clock source as 8254 Counter Output 2, paced acquisition selected
PTWR8		h04	h03	Bus Mastering mode, A/D FIFO enabled
The step to setup the Bus Mastering mode can be removed and one can enabled it when the A/D FIFO gets enabled (in the step above). Note: Always reset the A/D FIFO when procedure is completed.				
PTWR8	1	h04	h00	Disables acquisition / Resets A/D FIFO

Taken from file: 180x_ad3.cpp

```
...
/*
**  File:   1801_ad3.cpp
**  By:     T.Smith
**  Date:   8/13/1999
*/
...

volatile dword *pAD_DATA[4];
volatile char *pAD_CHCNT[4];
volatile char *pTRIG_SOURCE[4];
volatile char *pDIG_TRIG[4];
volatile char *pTRIG_CNT[4];
volatile char *pAD_CONTROL[4];
volatile char *pAD_SCK_CONFIG[4];
volatile char *pAD_BURST_VAL[4];
volatile word *pAD_QRAM[4];
...

// ******** Bus Master definitions ********

        // -- FIFO Flag Reset
        #define RESET_A2P_FLAGS     0x04000000L
        #define RESET_P2A_FLAGS     0x02000000L
        // -- FIFO priority
        #define A2P_HI_PRIORITY     0x00000100L
        #define P2A_HI_PRIORITY     0x00001000L
        // -- Enable Transfer Count
        #define EN_TCOUNT           0x10000000L
        // -- Enable Bus Mastering
        #define EN_A2P_TRANSFERS    0x00000400L
        #define EN_P2A_TRANSFERS    0x00004000L

// ********

dword data_array[100];
...

//********************************** MAIN ************************************

void  main(void)
{
    int board, Num_of_180xhcs, no_reads;
    int count = 0;
    word lo_word, hi_word;


    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("Using Board 0 for this test. \n");
        board = 0;

        printf("\nTHIS ROUTINE WILL SETUP & READ 50 READINGS FROM A/D CHANNEL0 IN BUS
MASTERING.\n");
        printf("Counter Output 2 used as Sample Clock with a frequency of 100kHz.\n");

        // for Bus Mastering mode there is a reading in the MSW and LSW
        // as for Pass Through where there the reading is only in the LSW
        // therefore the number of reads is half.
        no_reads = 25;

        // pointers to the A/D registers
        pAD_DATA[board] = (Baddr1[board].dp + 0x00);
        pAD_CHCNT[board] = (Baddr1[board].bp + 0x08);

        pTRIG_SOURCE[board] = (Baddr1[board].bp + 0x10);
        pDIG_TRIG[board] = (Baddr1[board].bp + 0x11);
        pTRIG_CNT[board] = (Baddr1[board].bp + 0x14);
        pAD_CONTROL[board] = (Baddr1[board].bp + 0x04);
        pAD_SCK_CONFIG[board] = (Baddr1[board].bp + 0x05);
        pAD_BURST_VAL[board] = (Baddr1[board].bp + 0x06);
        pAD_QRAM[board] = (Baddr1[board].wp + 0x2000);

        // pointers to the 8254 registers
        ...

        // Initializing trigger register for no triggers
        *pDIG_TRIG[board] = 0x00;

        // routine for configuring 8254
        config_8254(board);

        // routine for taking multiple readings
        ad_multi_bmread(board);

        // clearing and disabling A/D FIFO
        *pAD_CONTROL[board] = 0x00;

        printf("Storing data into file: 1801bm.txt.\n");

        // Storing data into file
        FILE *fp = fopen("1801bm.txt", "w");
        if (fp != NULL)
        {
            fprintf(fp, "1801bm.txt\n");
            fprintf(fp, "\nOutput from A/D's busmastr.exe (a DOS32 executible). \n");
            fprintf(fp, "Taking 50 readings at 100kHz (Bus Mastering mode).\n");
            fprintf(fp, "Each index contains two readings. One in the LO word and the
other in the HI word.\n\n");

            for (; count < no_reads; count++)
            {
                lo_word = (word)(data_array[count] & 0x0000FFFF);
                hi_word = (word)((data_array[count] & 0xFFFF0000) >> 16);
                fprintf(fp, "A/D reading index %i, hi = 0x%x, lo = 0x%x.\n", count,
hi_word, lo_word);
            } // for

            fprintf(fp, "------------------------------------ \n");
            fclose(fp);
        } // if (fp != NULL)

        printf("Finished.\n");
    } // (else Num_of_180xhcs)

}
//********************************** END of MAIN **********************************

// *************************** FUNCTIONS & ROUTINES *******************************
...
```

```
// *************** AD_MULTI_BMREAD *****************
// Setup A/D to take 50 readings.
// Using the 8254 counter2's output for the pacer clock.
// In bus master mode.

void ad_multi_bmread(int board_no)
{
    int i=0, j=0;
    dword temp = 0;
    dword temp2 = 0;
    volatile dword *pmwtc;
    volatile dword *pmwar;
    volatile dword *pmrar;
    volatile dword *pmrtc;
    volatile dword *pmcsr;

    // Configuring for only one channel.
    *pAD_CHCNT[board_no] = 0x00;

    // Channel 0, Differential, Bipolar, Gain of 1
    *pAD_QRAM[board_no] = 0x0000;

    // Paced mode with 8254 Counter2's Output for pacer clock source
    *pAD_SCK_CONFIG[board_no] = 0x00;

    // setup BusMastering registers

        pmcsr = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MCSR);
            // bm read and write addresses
        pmwar = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MWAR);
        pmrar = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MRAR);
            // bm read and write transfer count addresses
        pmwtc = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MWTC);
        pmrtc = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MRTC);

        *pmwar = (volatile dword)(&data_array[0]);
        *pmrar = (volatile dword)(&data_array[0]);

            // number of bytes to transfer 100 = 4x25
        *pmwtc = 100;
        *pmrtc = 100;

        // enable bus mastering and transfer count
        temp2 = *pmcsr;
        *pmcsr =
temp2|RESET_A2P_FLAGS|RESET_P2A_FLAGS|A2P_HI_PRIORITY|P2A_HI_PRIORITY|EN_A2P_TRANSFERS|EN
_P2A_TRANSFERS;

    // Offset binary output, (Bus Master)FIFO mode, A/D Fifo enabled
    *pAD_CONTROL[board_no] = 0x03;

//    printf("pmwtc = %lx\n", pmwtc);
//    printf("transfer count (*pmwtc) = %d\n", *pmwtc);

    // waiting for the transfer count to reach 0
    // when this is 0 all the data has been taken
    while(*pmwtc != 0)
    {
        if(temp == *pmwtc)
        {
            *pAD_CONTROL[board_no] = 0x00; // disable FIFO
            printf("ERROR: Transfer count not changing!\n");
            return;
        }
        printf("transfer count = %d\n", *pmwtc);
        temp = *pmwtc;
    } // for i

} // (ad_multi_bmread(int board_no)
// ************************************************
```

Example 4 does not have code source available. The above three examples along with the register and bit definitions should give the user enough information to create code for this example.

Ex4. *A positive edge Digital Trigger on Channel 0, single ended mode, Counter Output 2 as sample clock in Target mode with trigger count of 100.*

	Address			
Procedure	BADDR	Offset	Pattern	Description
PTWR8	1	h08	h00	Only 1 channel/configuration being used, channel limit value is zero based
PTWR32		h4000	h0240	Channel 0, Single Ended, Unipolar at a Gain of 1
PTWR8		h05	h00	Pacer clock source as 8254 Counter Output 2, paced acquisition selected
PTWR32		h14	h64	Trigger count limit value
PTWR8		h10	h00	Trigger on first entry in queue (h4000) – set up for CH0
PTWR8		h11	h1D	Positive edge trigger, TGIN selected, trigger counter selected
PTWR8		h04	h02	Offset Binary data, Target mode, A/D FIFO enabled
PTRD8		h04	bit5 -> 1	Read the status of the A/D FIFO Empty flag. If the bit is set then the FIFO has valid data. Once this is True then start reading.
PTRD32		h00	h????	Reading back the A/D data stored in the FIFO, Only the lower word is significant
PTWR8		h04	h00	Disables acquisition / Resets A/D FIFO

2.2 Digital to Analog

2.2.1 Features

The Digital to Analog design consists of Input FIFOs, two 12 bit D/A Converters (DAC), and internal logic in the Xilinx Spartan. The analog output signals are routed to connector J101. Both DAC outputs are located on this connector; channel 0 is on pin 86 and channel 1 is on pin 36. The transfer mode, update rate, acquisition mode, data format, range and channel control are all firmware/software selectable.

2.2.2 Power Up

On power up, both DAC output signals will be grounded until the DACs are properly configured by software. There is a bit that is software controlled that controls whether the output is active or grounded. Below is a table defining the location and polarity of the DAC output control bit. This is done so there will not be any damage of circuitry the user might have connected to the analog output signals at power up time.

Table: *DAC output control bit*

Address: BADDR2 +	Bit #	Mode	Polarity
h0C	7	Board's Analog Output Grounded	1
		Board's Analog Output Active	0

2.2.3 D/A FIFO Status

These bits are only readable. They show the status of the FIFO.

Table: *FIFO Status bit*

Address: BADDR2 +	Bit #	Function	Polarity
h04	7	FIFO Not Full	0 – full / 1 – not full
	6	FIFO Not Half Full	0 – > half / 1 – < half
	5	FIFO Empty	0 – not empty / 1 – empty

2.2.4 Update Rate

The update clock is the rate the D/A gets updated. This clock synchronizes the channel selection queue and DAC FIFO. The update clock can be a counter/timer of the 8254, an external clock (from DI0), or a software initiated write to the BADDR2 +h0. For the 8254 counter/timers and software write, the D/A gets updated on the rising edge of the pulse. As for the external clock, the polarity is selectable. The table below lists the available bit patterns for the sample clock source.
Note: Bit patterns, b011 and b111, are invalid values. There are no sources available for those patterns.

Table: *Update clock sources*

Address: BADDR2 +	Bits	Source Name	Bit pattern (b)
h04	[10:8]	8254 Counter 2 Output	000
		8254 Counter 1 Output	001
		8254 Counter 0 Output	010
		No source	*011*
		SW initiated write	100
		External clock, + polarity	101
		External clock, - polarity	110
		No source	*111*

The 8254 register should be configured prior to the sample rate selection of any of the three of the 8254 counter/timer outputs. Section 2.3 Counter/Timers will expand on the operation of the 8254 counter/timer.

2.2.5 Transfer Mode

There are two modes data can be transferred. The two modes are Pass Through (Target) and Bus Mastering (FIFO). The Pass Through mode has modest transfer requirements and has lower demand on the PCI bus. A feature of Pass Through transfer that performs high performance applications is called bursting. In section 2.2.5, this term will be defined. Bus Mastering is used to acquire and process data as quickly as possible. It is for high performance applications because it can attain peak transfer speeds.
The board is setup for 32 bit data transfers. In Bus Mastering mode, the 32 bit data will contain two 16 bit D/A update values. Internal in the FPGA, the low word on the PCI data bus (DQ [15:0]) will be latched out first onto the 16 bit DAC bus then DQ [31:16] will follow. As for the Pass Through mode, only the lower 16 bits will be accessed; the upper word would not be processed. For Bus Mastering, it is recommended to have an even number of counts to produce the best performance. The table below specifies the address and bit polarity for setting the transfer mode.

Table: *Transfer Mode bit*

Address: BADDR2 +	Bit #	Mode	Polarity
h04	0	Target	0
		FIFO	1

2.2.6 Acquisition Mode

There are two modes to configure and acquire analog output data. These two modes are Paced and Burst. Paced mode uses the update rate to step through the Channel Select Queue (section 2.2.7) that will select which DAC to update. As for the Burst mode, there is a 1 MHz clock source that gets divided by a burst count. In this mode, a value is taken from the Channel Queue on each rising edge of the burst frequency. The burst frequency starts its run at each rising sample clock edge. The Channel Limit value (section 2.2.7) will determine the number of burst pulses. If the Channel Limit value is equal to 0 (only 1 entry) while in Burst mode, it will resemble Paced mode. The table below specifies the address and bit polarity for setting the acquisition mode.

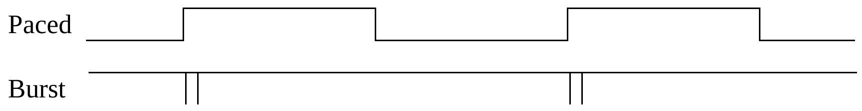
Table: *Acquisition Mode bit*

Address: BADDR2 +	Bit #	Mode	Polarity
h04	11	Paced	0
		Burst	1

Table: *Burst Count Value*

Address: BADDR2 +	Bits	Definition	Examples
h04	[23:16]	Burst count of 5, Produces a burst clock of 200 kHz.	h05
		Burst count of 250, Produces a burst clock of 4 kHz.	hFA

Diagram: *Paced and Burst clock behavior with a channel limit value = 1 (two entries):*



2.2.7 Data Format

The 12 bit D/A input MUST be in Offset Binary format. The actual D/A parts used on board are designed to accept Offset Binary codes. The FPGA was designed with the option to change the input format between Offset Binary and 2's Complement output. A bit has been declared to convert the incoming data. The table specifies this bit at its address location.

Table: *Data format bit*

Address: BADDR2 +	Bit #	Mode	Polarity
h04	2	No Conversion	0
		To Offset Binary	1

If the user prefers to use 2's Complement format for the DAC input values then the Data Format bit must be set in order for the output to reflect the correct result. If the input is not converted to its Offset Binary representation prior to the DAC then the DAC will interpret the 2's Complement value as an Offset Binary value and return the incorrect result.

2.2.8 Channel Select Queue

This queue is similar to the A/D parameter Queue except that the only parameter is the channel select. The values for the queue must be written prior to updating the DAC(s). Internal to the FPGA, there is a 1 bit 16 deep RAM block that is used to store the channel queue. The table gives the beginning and end address of the queue and the bit that needs setting. This bit selects between the two available DACs.
A count value must be defined prior to the update clock running through this queue. This count value is defined as the Channel limit value. The value for the limit can not exceed the maximum queue length, which is 16. A location and bit definition will be shown under the Channel limit value table. The examples are assuming a 32 bit write is taking place.
Note: For the use of this board the maximum Channel limit value will be 1. This limit index is zero based. As for the RAM mentioned above only 2 entries will be needed. Therefore the end of the queue will stop after address offset h84.

Table: *Channel Select Queue*

Address: BADDR2 +	Index	Bit #	Setting
h80	0	0	0 – DAC0 / 1 – DAC1
h84	1		

Table: *Channel Limit value*

Address: BADDR2 +	Bit	Definition	Setting
h08	0	1 Entry of Channel Queue	hxx xx x0

Ex1. *Taking only 1 entry which is configured for DAC0*

```
BADDR2 +h80   [31:0]          hxx xx xx x0
BADDR2 +h08   [31:0]          hxx xx xx x0
```

Ex2. *Taking 2 entries where the first is for DAC1 and the second is for DAC0*

```
BADDR2 +h80   [31:0]          hxx xx xx x1
BADDR2 +h84   [31:0]          hxx xx xx x0
BADDR2 +h08   [31:0]          hxx xx xx x1
```

2.2.9 Configuration Examples

The following examples give a variety of possible ways to configure the DAC to acquire samples. For all examples except the first, the 8254 counter/timer must be setup first. Refer to Section 2.3.1 for bit pattern description for the 8254 Counter/Timer.

Provided below with each example is C Code that was used toward a DOS32 executable. The complete files can be found in Appendix B. For the 8254 Setup, examples of code are located in Section 2.3.4.

Ex. *8254 Setup: Counter 0 source of 10MHz with counter value of 10, Counter 1 source of 5MHz with counter value of 10, Counter 2 source of Counter 1 Output with counter value of 5.*

Procedure	Address BADDR	Offset	Pattern	Description
PTWR32	3	h10	hx5x1x0	Selecting the source frequency for each counter.
PTWR8		h0C	h14	Format Counter 0
PTWR8		h00	h0A	Counter 0 counter value
PTWR8		h0C	h54	Format Counter 1
PTWR8		h04	h0A	Counter 1 counter value
PTWR8		h0C	h94	Format Counter 2
PTWR8		h08	h05	Counter 2 counter value

Results of above action list would give on each counter's outputs the following frequencies:
Counter 0 Output = 1MHz; Counter 1 Output = 500KHz; Counter 2 Output = 100kHz.

Ex1. *Configure DAC Channel 0 Output to +5V using a software initiated write as the update clock while in Target mode. Use the data format of Offset Binary.*

Procedure	Address BADDR	Offset	Pattern	Description
PTWR8	2	h08	h00	Only 1 channel being used, channel limit value is zero based
PTWR32		h80	h0	DAC Channel 0
PTWR8		h05	h04	Pacer clock source as SW initiated write to BADDR2+h0, paced acquisition selected
PTWR8		h04	h02	Offset Binary data, Target mode, DAC FIFO enabled
PTWR32		h00	hC000	SW initiated write – Update clock tick, Writes out voltage value to DAC
*PTWR8		h0C	h80	Supplies board's output of each DAC.
*Note: For this example DAC Channel 1 was not used or configured. So the above Pass-Thru Write to BADDR2+h0C will supply –FS or –10V onto DAC1's output. To force both DACs to 0V perform a PTWR8 at BADDR2+h0D with h00. This action could also be placed prior to writing data to the DAC. It would then supply –10V to both DAC outputs before the DAC(s) are configured. This is due to the register initializing to all zeroes. In offset binary, all zeroes represent –FS.				
Note: Always reset the A/D FIFO when action or procedure is completed.				
PTWR8	2	h04	h00	Disables acquisition / Resets D/A FIFO

Taken from file: 180x_da1.cpp

```
...
/*
**  File:   1801_da1.cpp
**  By:     T.Smith
**  Date:   8/13/1999
*/
...

volatile dword *pDA_DATA[4];
volatile char *pDA_CHCNT[4];
volatile char *pDA_CONTROL[4];
volatile char *pDA_UPCK_CONFIG[4];
volatile char *pDA_BURST_VAL[4];
volatile char *pDA_QRAM[4];
volatile char *pDA_ENABLE[4];
...

//************************************ MAIN ************************************

void  main(void)
{
    int board, Num_of_180xhcs;


    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("\nTHIS ROUTINE WILL PERFORM AN UPDATE TO A DAC CHANNEL.\n\n");
        printf("Using Board 0 for this test. \n");
        board = 0;

        // pointers to the A/D registers
        pDA_DATA[board] = (Baddr2[board].dp + 0x00);
        pDA_CHCNT[board] = (Baddr2[board].bp + 0x08);

        pDA_CONTROL[board] = (Baddr2[board].bp + 0x04);
        pDA_UPCK_CONFIG[board] = (Baddr2[board].bp + 0x05);
        pDA_BURST_VAL[board] = (Baddr2[board].bp + 0x06);
        pDA_QRAM[board] = (Baddr2[board].bp + 0x80);
        pDA_ENABLE[board] = (Baddr2[board].bp + 0x0C);

        // pointers to the 8254 registers
        ...

        // routine for configuring 8254
        config_8254(board);

        // routine for a D/A Update
        da_update(board);

        printf("Update Completed!\n");

        // clearing and disabling D/A FIFO
        *pDA_CONTROL[board] = 0x00;

    } // (else Num_of_180xhcs)

}
//************************************ END of MAIN ************************************

// **************************** FUNCTIONS & ROUTINES ********************************
...

// *************** DA_UPDATE *****************
// Setup D/A to write a value (from user) to either D/A channel.
// Using a SW Write for the update clock.
// The SW Write is also used to write the D/A data to the DAC channel.

void da_update(int board_no)
{
    #define DATA_MASK   0x0000FFFF
    #define CH_MASK     0x01
    byte dac_channel =0;
    byte dac_on;
    dword dac_data =0;

    printf("\nSelect DAC Channel '0' or '1': ");
    scanf("%x", &dac_channel);

    // Configuring for only one channel.
    *pDA_CHCNT[board_no] = 0x00;

    // DAC Channel 0 or 1 (from user input above)
    *pDA_QRAM[board_no] = (dac_channel & CH_MASK);

    // Paced mode with SW Write/Virtual Write for update clock source
    *pDA_UPCK_CONFIG[board_no] = 0x04;

    // Requesting Offset binary Data to Write
    printf("Offset Binary Value to Write: 0x ");
    scanf("%lx", &dac_data);

    // No conversion, (Pass-Thru)Target mode, D/A Fifo enabled
    *pDA_CONTROL[board_no] = 0x02;

    // Activating Board's D/A Output
    // To activate the board’s D/A outputs (0x00).
    // To ground board’s D/A outputs (0x01).
    dac_on = 0x00;
    *pDA_ENABLE[board_no] = dac_on;

    // Writing data to update DAC
    // For PassThru only the lower word is used. So masking upper word to zeroes.
    *pDA_DATA[board_no] = (dac_data & DATA_MASK);

    // Debugging printf's...
    //-- printf("Dac range: 0x%x\n", *pDA_RANGE[board_no]);

    //-- printf("\nReading DAC Data from register = 0x%lx\n",*pDA_DATA[board_no]);

} // (void da_update)
// ************************************************
```

Ex2. *Configure both DAC channels, CH0 to –5V and CH1 to +10V using 8254 Counter 2 Output as the update clock while in Target mode. Use the data format of Offset Binary.*

Procedure	Address BADDR	Address Offset	Pattern	Description
PTWR8	2	h08	h01	Both channels being used, channel limit value is zero based
PTWR32		h80	h0	DAC Channel 0
PTWR32		h84	h1	DAC Channel 1
PTWR8		h05	h00	Update clock source as 8254 Counter 2 Output, paced acquisition selected
*PTWR8		h0C	h01	Supplies board's output of each DAC.
PTWR8		h04	h02	Offset Binary data, Target mode, DAC FIFO enabled
PTWR32		h00	h4000	Writing to DAC FIFO, Ch0 data.
PTWR32		h00	hFFFF	Writing to DAC FIFO, Ch1 data.
*Note: This action, to open the relay to supply the DAC output, only needs to be performed at the first DAC write, unless this action was performed to close the relay (h00) which will ground both DAC outputs.				
Note: Always reset the D/A FIFO when action or procedure is completed.				
PTWR8	2	h04	h00	Disables acquisition / Resets D/A FIFO

Taken from file: 180x_da2.cpp

```
...
/*
**  File:   1801_da2.cpp
**  By:     T.Smith
**  Date:   8/16/1999
*/
...

volatile dword *pDA_DATA[4];
volatile char *pDA_CHCNT[4];
volatile char *pDA_CONTROL[4];
volatile char *pDA_UPCK_CONFIG[4];
volatile char *pDA_BURST_VAL[4];
volatile char *pDA_QRAM[4];
volatile char *pDA_ENABLE[4];
...

//********************************** MAIN ***********************************

void  main(void)
{
    int board, Num_of_180xhcs;


    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("\nTHIS ROUTINE WILL CREATE A PATTERN TO SEND TO A DAC CHANNEL IN PASS 
THRU.\n");
        printf("Counter Output 2 used as Update Clock with a frequency of 100Hz.\n");
        printf("Using Board 0 for this test. \n");
        board = 0;

        // pointers to the A/D registers
        pDA_DATA[board] = (Baddr2[board].dp + 0x00);
        pDA_CHCNT[board] = (Baddr2[board].bp + 0x08);

        pDA_CONTROL[board] = (Baddr2[board].bp + 0x04);
        pDA_UPCK_CONFIG[board] = (Baddr2[board].bp + 0x05);
        pDA_BURST_VAL[board] = (Baddr2[board].bp + 0x06);
        pDA_QRAM[board] = (Baddr2[board].bp + 0x80);
        pDA_ENABLE[board] = (Baddr2[board].bp + 0x0C);

        // pointers to the 8254 registers
        ...

        // routine for configuring 8254
        config_8254(board);

        // creates pattern
        gen_pattern();

        // routine for a D/A Update
        da_update(board);

        printf("Update Completed!\n");

        // clearing and disabling D/A FIFO
        *pDA_CONTROL[board] = 0x00;

    } // (else Num_of_180xhcs)

}
//********************************** END of MAIN ***********************************

// *************************** FUNCTIONS & ROUTINES ********************************
...
```

```
// **************** GEN_PATTERN ********************
```

```
/* generates the top half of a triangular sine wave using 50 points
**         ^
**       /   \       <-- like this
**     /       \
*/

void gen_pattern(void)
{
    int i = 0;
    #define INC_DEC_VAL     0x051E
    #define ISTART_VAL      0x8000
    #define DSTART_VAL      0xF000

    for(; i<MAX_COUNT; i++)
    {
        pattern[i] = ISTART_VAL + (INC_DEC_VAL * i);
    }
    printf("pattern[MAX_COUNT-1] = %x\n", pattern[MAX_COUNT-1]);
    for(i=0; i<MAX_COUNT; i++)
    {
        pattern[i+MAX_COUNT] = DSTART_VAL - (INC_DEC_VAL * i);
    }
    printf("pattern[(MAX_COUNT*2)-1] = %x\n", pattern[(MAX_COUNT*2)-1]);
}
// ************** END OF GEN_PATTERN ****************


// *************** DA_UPDATE *****************
// Setup D/A to write a triangle pattern to either D/A channel.
// Using 8254 Counter2 Output for the update clock.
// In Pass Through mode.

void da_update(int board_no)
{
    #define DATA_MASK   0x0000FFFF
    #define FIFO_EMPTY  0x20
    #define FIFO_FULL   0x80
    #define CH_MASK     0x01
    byte dac_channel =0;
    byte dac_on;
    dword dac_data =0;
    int i;


    printf("\nSelect DAC Channel '0' or '1': ");
    scanf("%x", &dac_channel);

    // Configuring for only one channel.
    *pDA_CHCNT[board_no] = 0x00;

    // DAC Channel 0 or 1 (from user input above)
    *pDA_QRAM[board_no] = (dac_channel & CH_MASK);

    // Paced mode with SW Write/Virtual Write for update clock source
    *pDA_UPCK_CONFIG[board_no] = 0x00;

    // No conversion, (Pass-Thru)Target mode, D/A Fifo enabled
    *pDA_CONTROL[board_no] = 0x02;

    // Activating Board's D/A Output
    // Must keep this set (0x01).
    // If cleared (0x00) then D/A outputs become grounded on board.
    dac_on = 0x00;
    *pDA_ENABLE[board_no] = dac_on;

    // Writing data to update DAC
    // For PassThru only the lower word is used. So masking upper word to zeroes.
    for(i=0; i<(MAX_COUNT*2); i++)
    {
            if( !(*pDA_CONTROL[board_no] & FIFO_FULL))
            {
                *pDA_DATA[board_no] = pattern[i] & DATA_MASK;
            }
            else
            {
                while( !(*pDA_CONTROL[board_no] & FIFO_EMPTY));
                printf("Count %d: FIFO status bits: full-half-empty %x.\n",
i,(*pDA_CONTROL[board_no] & 0xE0));
            }
    } // for i

    // Waiting till the D/A FIFO is empty (completed updating)
    // This may not work because the internal FIFO might not be deep enough.
    // 8/13/99: Only seeing 1/2 of data or just rising edge of waveform.
    while ( !(*pDA_CONTROL[board_no] & FIFO_EMPTY) );

}
// ************* END OF DA_UPDATE *******************
```

2.3 Counter/Timers

Supported on this board are 3 counter/timer sources. They are contained in the Harris 82C54 CMOS Programmable Timer chip. Each counter can be selected as the A/D sample clock or the D/A update clock.

2.3.1 Internal Sources

The board supports four internal frequency sources for the 3 counter/timers in the Harris 82C54 CMOS Programmable Timer. These sources are 10MHz, 5MHz, 1MHz and 100kHz. The 10 MHz signal is created from a 10 MHz crystal oscillator on board. It gets divided down into the other three inside the Xilinx.

2.3.2 Configuration

Both sections above share the same registers. So this section will outline the registers that are needed to configure an 8254 counter/timer with either an internal or external source. The first register is called the 8254 Timer Source control register. This register contains the list of possible sources available for each counter/timer on the 8254. The “Enabled” label in the table holds the gate signal HIGH allowing for a continuous pulse train.

Table: *8254 Timer Source Control*

Address: BADDR3 +	Bits Used	Definition	Setting	
h10	19	8254 Counter 2 Gate	0 – Enabled / 1 –Disabled	
	[18:16]	8254 Counter 2 Source	000 - +10 MHz 001 - +5 MHz 010 - +1 MHz 011 - +100 kHz	100 – no source 101 – Counter 1 Output 110 – Counter 0 Output 111 – no source
	11	8254 Counter 1 Gate	0 – Enabled / 1 –Disabled	
	[10:8]	8254 Counter 1 Source	000 - +10 MHz 001 - +5 MHz 010 - +1 MHz 011 - +100 kHz	100 – no source 101 – Counter 2 Output 110 – Counter 0 Output 111 – no source
	3	8254 Counter 0 Gate	0 – Enabled / 1 –Disabled	
	[2:0]	8254 Counter 0 Source	000 - +10 MHz 001 - +5 MHz 010 - +1 MHz 011 - +100 kHz	100 – no source 101 – Counter 2 Output 110 – Counter 1 Output 111 – no source

Ex1. *All gates enabled, Counter 0 source is 100 kHz, Counter 1 source is Counter 0’s output, and Counter 2 source is 5 MHz.*

```
BADDR3 +h10        [31:0]          hxxx1 x6x3
```

Ex2. *Counter1’s output stopped, Counter 0 source is 100 kHz, Counter 1 source is 1 MHz, and Counter 2 source is 5 MHz.*

```
BADDR3 +h10        [31:0]          hxxx1 xAx3
```

The 82C54 chip HAS TO BE CONFIGURED PRIOR to using any of the three counters’ outputs. There is a format byte that needs to be sent for each counter/timer. A count for each counter/timer must also be sent. The count is used to divide down the input source frequency selected from the above register. This allows a large range of frequencies in order to accommodate the user’s needs.

Table: *8254 Format Byte*

Address: BADDR3 +	Bits Used	Definition	Setting
h0C	[7:6]	Counter Select	00 – Counter 0 01 – Counter 1 10 – Counter 2 11 – Read Back Command
	[5:4]	Read/Write	00 – Counter Latch 01 – LSB Only 10 – MSB Only 11 – LSB then MSB
	[3:1]	Counter Mode	000 – Interrupt on Terminal count 001 – HW Retriggerable One Shot x10 – Rate Generator x11 – Square Wave Mode 100 – SW Triggered Mode 101 – HW Triggered Strobe
	0	Counter Format	0 – 16 bit Binary / 1 – 4 decade BCD

Table: *Counter Select count registers*

Address: BADDR3 +	Bits Used	Definition
h08	[7:0]	Counter 2 Count Value
h04	[7:0]	Counter 1 Count Value
h00	[7:0]	Counter 0 Count Value

2.3.3 Configuration Examples

Ex1. *8254 Setup: Counter 0 source of 10MHz with counter value of 5, Counter 1 source of 5MHz with counter value of 10, Counter 2 source of 1MHz with counter value of 5.*

Procedure	Address BADDR	Address Offset	Pattern	Description
PTWR32	3	h10	hx2x1x0	Selecting the source frequency for each counter.
PTWR8		h0C	h14	Format Counter 0
PTWR8		h00	h05	Counter 0 counter value
PTWR8		h0C	h54	Format Counter 1
PTWR8		h04	h0A	Counter 1 counter value
PTWR8		h0C	h94	Format Counter 2
PTWR8		h08	h05	Counter 2 counter value

Results of above action list would give on each counter’s outputs the following frequencies:
Counter 0 Output = 2MHz; Counter 1 Output = 500KHz; Counter 2 Output = 200kHz.

Ex2. *8254 Setup: Counter 0 source of 10MHz with counter value of 10, Counter 1 source of 5MHz with counter value of 10, Counter 2 source of Counter 1 Output with counter value of 5.*

Procedure	Address BADDR	Address Offset	Pattern	Description
PTWR32	3	h10	hx5x1x0	Selecting the source frequency for each counter.
PTWR8		h0C	h14	Format Counter 0
PTWR8		h00	h0A	Counter 0 counter value
PTWR8		h0C	h54	Format Counter 1
PTWR8		h04	h0A	Counter 1 counter value
PTWR8		h0C	h94	Format Counter 2
PTWR8		h08	h05	Counter 2 counter value

Results of above action list would give on each counter's outputs the following frequencies:
Counter 0 Output = 1MHz; Counter 1 Output = 500KHz; Counter 2 Output = 100kHz.

The following code examples are taken from (1) 180x_ad2.cpp and (2) 180x_da2.cpp. They might not match the above examples but will show the procedure.

(1) From file 180x_ad2.cpp:

. . .

```
/*
```

```
**  File:   1801_ad2.cpp
```

```
**  By:     T.Smith
```

```
**  Date:   8/13/1999
```

```
*/
...
        volatile char *p8254_CNTRL0[4];
        volatile char *p8254_CNTRL1[4];
        volatile char *p8254_CNTRL2[4];
        volatile char *p8254_FORMAT[4];
        volatile char *p8254_CNTR2_INITVAL[4];
        volatile char *p8254_CNTR1_INITVAL[4];
        volatile char *p8254_CNTR0_INITVAL[4];

...

int get_board_info(void);
void ad_multi_ptread(int board_no, int cnt);
void config_8254(int board_no);

//********************************** MAIN ***********************************

void  main(void)
{
    int board, Num_of_180xhcs, no_reads;
    int count = 0;

    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("Using Board 0 for this test. \n");
        board = 0;
        printf("\nTHIS ROUTINE WILL SETUP & READ 50 READINGS FROM A/D CHANNEL0 IN PASS THRU.\n");
        printf("Counter Output 2 used as Sample Clock with a frequency of 100kHz.\n");
        no_reads = 50;

        // pointers to the A/D registers

        ...

        // pointers to the 8254 registers
        p8254_CNTRL0[board] = (Baddr3[board].bp + 0x10);
        p8254_CNTRL1[board] = (Baddr3[board].bp + 0x11);
        p8254_CNTRL2[board] = (Baddr3[board].bp + 0x12);
        p8254_FORMAT[board] = (Baddr3[board].bp + 0x0C);
        p8254_CNTR2_INITVAL[board] = (Baddr3[board].bp + 0x08);
        p8254_CNTR1_INITVAL[board] = (Baddr3[board].bp + 0x04);
        p8254_CNTR0_INITVAL[board] = (Baddr3[board].bp + 0x00);


        // Initializing trigger register for no triggers
        *pDIG_TRIG[board] = 0x00;

        // routine for configuring 8254
        config_8254(board);

        // routine for taking multiple readings
        ad_multi_ptread(board, no_reads);

        printf("Storing data into file: PT50rdgs.txt !\n");

        // Storing data into file
        FILE *fp = fopen("PT50rdgs.txt", "w");
        if (fp != NULL)
        {
            ...

        } // if (fp != NULL)

    } // (else Num_of_180xhcs)
}
//********************************** END of MAIN ************************************

...

// *************** CONFIG_8254 *****************
// Configuring each counter of the 8254.
// -- All Counter Gates enabled; allowing continuous train.
// -- Counter0 -> (source) 10MHz % by 100 (count) = 100kHz
// -- Counter1 -> (source) 5MHz % by 10 (count) = 500kHz
// -- Counter2 -> (source) Counter 1 Output % by 5 (count) = 100kHz

void Config_8254(int board_no)
{
    // select sources and gating for 8254 counters
    *p8254_CNTRL0[board_no] = 0x00;
    *p8254_CNTRL1[board_no] = 0x01;
    *p8254_CNTRL2[board_no] = 0x05;


    // format 8254 counters
    *p8254_FORMAT[board_no] = 0x14;
    *p8254_CNTR0_INITVAL[board_no] = 0x64;
    *p8254_FORMAT[board_no] = 0x54;
    *p8254_CNTR1_INITVAL[board_no] = 0x0A;
    *p8254_FORMAT[board_no] = 0x94;
    *p8254_CNTR2_INITVAL[board_no] = 0x05;

//    printf("Finished configuring 8254's counters.\n");

} // (void 8254_config)
// ************************************************
```

(2) From file 180x_da2.cpp:

. . .

```
/*
```

```
**  File:   1801_da2.cpp
```

```
**  By:     T.Smith
**  Date:   8/16/1999
*/
...
        volatile char *p8254_CNTRL0[4];
        volatile char *p8254_CNTRL1[4];
        volatile char *p8254_CNTRL2[4];
        volatile char *p8254_FORMAT[4];
        volatile char *p8254_CNTR2_INITVAL[4];
        volatile char *p8254_CNTR1_INITVAL[4];
        volatile char *p8254_CNTR0_INITVAL[4];

...

void  main(void)
{
    int board, Num_of_180xhcs;


    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("\nTHIS ROUTINE WILL CREATE A PATTERN TO SEND TO A DAC CHANNEL IN PASS THRU.\n");
        printf("Counter Output 2 used as Update Clock with a frequency of 100Hz.\n");
        printf("Using Board 0 for this test. \n");
        board = 0;

        // pointers to the A/D registers

        ...

        // pointers to the 8254 registers
        p8254_CNTRL0[board] = (Baddr3[board].bp + 0x10);
        p8254_CNTRL1[board] = (Baddr3[board].bp + 0x11);
        p8254_CNTRL2[board] = (Baddr3[board].bp + 0x12);
        p8254_FORMAT[board] = (Baddr3[board].bp + 0x0C);
        p8254_CNTR2_INITVAL[board] = (Baddr3[board].bp + 0x08);
        p8254_CNTR1_INITVAL[board] = (Baddr3[board].bp + 0x04);
        p8254_CNTR0_INITVAL[board] = (Baddr3[board].bp + 0x00);


        // routine for configuring 8254
        config_8254(board);

        // creates pattern
        gen_pattern();

        // routine for a D/A Update
        da_update(board);

        printf("Update Completed!\n");

        // clearing and disabling D/A FIFO
        *pDA_CONTROL[board] = 0x00;

    } // (else Num_of_180xhcs)

}
//********************************** END of MAIN ************************************

...

// *************** CONFIG_8254 *****************
// Configuring each counter of the 8254.
// -- All Counter Gates enabled; allowing continuous train.
// -- Counter0 -> (source) 10MHz % by 100 (count) = 100kHz
// -- Counter1 -> (source) 100kHz % by 100 (count) = 1kHz
// -- Counter2 -> (source) Counter 1 Output % by 10 (count) = 100Hz

void Config_8254(int board_no)
{
    // select sources and gating for 8254 counters
    *p8254_CNTRL0[board_no] = 0x00;
    *p8254_CNTRL1[board_no] = 0x03;
    *p8254_CNTRL2[board_no] = 0x05;

    // format 8254 counters
    *p8254_FORMAT[board_no] = 0x14;
    *p8254_CNTR0_INITVAL[board_no] = 0x64;
    *p8254_FORMAT[board_no] = 0x54;
    *p8254_CNTR1_INITVAL[board_no] = 0x64;
    *p8254_FORMAT[board_no] = 0x94;
    *p8254_CNTR2_INITVAL[board_no] = 0x0A;

    // printf("Finished configuring 8254's counters.\n");

}
// ************* END OF CONFIG_8254 *******************
```

2.4 Interrupts

This board supports the ability to enable/disable the source of an interrupt and check who caused the interrupt. An ISR (interrupt service routine) created in software by the user or DriverLynx will execute to check what interrupt had fired and proceed accordingly. This is done on the PCI side.

On board, the Xilinx produces an interrupt depending if the user enabled a source. To enable a source, the software must configure a register that contains all the sources available to cause an interrupt. The sources include signals from the A/D FIFO, A/D Trigger, D/A FIFO, and the three 8254 Counter/Timer Outputs. This register is shown below in the Interrupt Enable/Disable register table.

Table: *Interrupt Enable/Disable register*

Address: BADDR3 +	Bits Used	Definition	Setting
h18	[18:16]	8254 Counter 2 Output	0 – Disable / 1 – Enable
		8254 Counter 1 Output	
		8254 Counter 0 Output	
	[10:8]	DAC FIFO Full	
		DAC FIFO Half Full	
		DAC FIFO Empty	
	[5:0]	ADS Trigger	
		ADS Trigger Count	
		ADS Frame Sync*	
		ADS FIFO Overflow	
		ADS FIFO Half Full	
		ADS FIFO Not Empty	

* The ADS Frame Sync is fired when the ADS Parameter Queue has completed a cycle. A cycle is the amount of entries defined by the Channel Limit value (section 2.1.7). If only one channel is the limit of the QRAM, this signal remains high for the sweep.

The interrupt signal from the Xilinx is connected to the mailbox interrupt control pin of the PCI controller chip in order for the PCI to recognize that an interrupt fired. Once the software recognizes that there was an interrupt, it needs to read the Interrupt Status register to find out which source caused the interrupt. Only those interrupt sources enabled in the Interrupt Enable register will affect this register. This register is shown in the Interrupt Status/Reset register table. Clearing the interrupt in this register will allow the interrupt to be recognized if it fires again. It will not disable the interrupt. In order to clear the interrupt, a ‘1’ MUST be sent to the bit.

Table: *Interrupt Status/Reset register*

Address: BADDR3 +	Bits Used	Definition	Setting
h18	[18:16]	8254 Counter 2 Output	0 – Reset / 1 – Fired
		8254 Counter 1 Output	
		8254 Counter 0 Output	
	[10:8]	DAC FIFO Full	
		DAC FIFO Half Full	
		DAC FIFO Empty	
	[5:0]	ADS Trigger	
		ADS Trigger Count	
		ADS Frame Sync*	
		ADS FIFO Overflow	
		ADS FIFO Half Full	
		ADS FIFO Not Empty	

2.5 I/O

2.5.1 Digital (TTL) IO

There are 8 bits of digital output and 4 bits of digital input supported on this board. The table below shows the register to control these bits of data.

Table: *Digital IO Data*

Address: BADDR3 +	Bits	Direction	Definition
h24	[7:0]	Read	Reads Input with upper nibble masked to zero.
		Write	Writes out the data value contained in this register.

Table: *Inputs*

Input bit	Use
3	General Purpose
2	General Purpose
1	General Purpose OR Digital Trigger / Gate (TGIN)
0	General Purpose OR External Pacer Clock

2.5.2 Other Outputs

There are three output lines that are available on connector J101. These signals are used in conjunction with certain accessory boards.

Table: *Outputs*

Outputs	Description
TGOUT	Trigger/Gate Output. The signal is the board's internal Trigger/Gate signal buffered.
SSHO	Simultaneous Sample and Hold Output.
DOSTRB	Data Output Strobe. The signal is createde when a write of the output digital data at BADDR3+h24 is performed.

3. Appendixes

These two appendixes include the complete C code sources. Appendix A has the AMCC C file and header file, Appendix B lists some general code, and Appendix C supplies the complete sources from the previous sections.

Appendix A: AMCC Code

This section provides C code provided by AMCC to find the device on the bus and read its configuration space. These source files are the complete code.

AMCC.H

```
/****************************************************************************/
/*                                                                          */
/* Module: AMCC.H                                                           */
/*                                                                          */
/* Purpose: Definitions for AMCC PCI Library                                */
/*                                                                          */
/****************************************************************************/

/****************************************************************************/
/*  General Constants                                                       */
/****************************************************************************/


typedef unsigned char byte;     /* 8-bit  */
typedef unsigned short word;    /* 16-bit */
typedef unsigned long dword;    /* 32-bit */

// Declare some global unions to simulate registers
union
{
unsigned long   dw[16];
word     w[32];
unsigned char     b[64];
}registers;

typedef union pntr
        {
        volatile char*  bp;
        volatile word*  wp;
        volatile dword* dp;
        byte    b;
        word    w;
        dword   dw;
        }pointer;


#define CARRY_FLAG 0x01         /* 80x86 Flags Register Carry Flag bit */

/****************************************************************************/
/*   PCI Functions                                                          */
/****************************************************************************/

#define PCI_FUNCTION_ID           0xb1
#define PCI_BIOS_PRESENT          0x01
#define FIND_PCI_DEVICE           0x02
#define FIND_PCI_CLASS_CODE       0x03
#define GENERATE_SPECIAL_CYCLE    0x06
#define READ_CONFIG_BYTE          0x08
#define READ_CONFIG_WORD          0x09
#define READ_CONFIG_DWORD         0x0a
#define WRITE_CONFIG_BYTE         0x0b
#define WRITE_CONFIG_WORD         0x0c
#define WRITE_CONFIG_DWORD        0x0d

/****************************************************************************/
```

```
/*   PCI Return Code List                                                 */
/****************************************************************************/

#define SUCCESSFUL                0x00
#define NOT_SUCCESSFUL            0x01
#define FUNC_NOT_SUPPORTED        0x81
#define BAD_VENDOR_ID             0x83
#define DEVICE_NOT_FOUND          0x86
#define BAD_REGISTER_NUMBER       0x87

/****************************************************************************/
/*   PCI Configuration Space Registers                                    */
/****************************************************************************/

#define PCI_CS_VENDOR_ID          0x00
#define PCI_CS_DEVICE_ID          0x02
#define PCI_CS_COMMAND            0x04
#define PCI_CS_STATUS             0x06
#define PCI_CS_REVISION_ID        0x08
#define PCI_CS_CLASS_CODE         0x09
#define PCI_CS_CACHE_LINE_SIZE    0x0c
#define PCI_CS_MASTER_LATENCY     0x0d
#define PCI_CS_HEADER_TYPE        0x0e
#define PCI_CS_BIST               0x0f
#define PCI_CS_BASE_ADDRESS_0     0x10
#define PCI_CS_BASE_ADDRESS_1     0x14
#define PCI_CS_BASE_ADDRESS_2     0x18
#define PCI_CS_BASE_ADDRESS_3     0x1c
#define PCI_CS_BASE_ADDRESS_4     0x20
#define PCI_CS_BASE_ADDRESS_5     0x24
#define PCI_CS_EXPANSION_ROM      0x30
#define PCI_CS_INTERRUPT_LINE     0x3c
#define PCI_CS_INTERRUPT_PIN      0x3d
#define PCI_CS_MIN_GNT            0x3e
#define PCI_CS_MAX_LAT            0x3f

/****************************************************************************/
/*   AMCC Operation Register Offsets                                      */
/****************************************************************************/

#define AMCC_OP_REG_OMB1          0x00
#define AMCC_OP_REG_OMB2          0x04
#define AMCC_OP_REG_OMB3          0x08
#define AMCC_OP_REG_OMB4          0x0c
#define AMCC_OP_REG_IMB1          0x11
#define AMCC_OP_REG_IMB2          0x14
#define AMCC_OP_REG_IMB3          0x18
#define AMCC_OP_REG_IMB4          0x1c
#define AMCC_OP_REG_FIFO          0x20
#define AMCC_OP_REG_MWAR          0x24
#define AMCC_OP_REG_MWTC          0x28
#define AMCC_OP_REG_MRAR          0x2c
#define AMCC_OP_REG_MRTC          0x30
#define AMCC_OP_REG_MBEF          0x34
#define AMCC_OP_REG_INTCSR        0x38
#define AMCC_OP_REG_MCSR          0x3c
#define AMCC_OP_REG_MCSR_NVDATA  (AMCC_OP_REG_MCSR + 2) /* Data in byte 2 */
#define AMCC_OP_REG_MCSR_NVCMD   (AMCC_OP_REG_MCSR + 3) /* Command in byte 3 */

/****************************************************************************/
/*   AMCC NV Bit Masks
*/
/****************************************************************************/
#define AMCC_NV_BUSY_MASK                              0x80
#define AMCC_NV_LOW_ADDRESS_COMMAND            0x80
#define AMCC_NV_HIGH_ADDRESS_COMMAND    0xA0
#define AMCC_NV_READ_COMMAND                   0xE0
#define AMCC_NV_WRITE_COMMAND                  0xC0
#define AMCC_NV_NO_COMMAND                             0x00

#define BADDR_0_CFIG_SPACE_OFFSET 0x10
```

```
#define BADDR_1_CFIG_SPACE_OFFSET 0x14
#define BADDR_2_CFIG_SPACE_OFFSET 0x18
#define BADDR_3_CFIG_SPACE_OFFSET 0x1C
#define BADDR_4_CFIG_SPACE_OFFSET 0x20

#define SI              registers.dw[4]
#define DI              registers.dw[5]
#define FLAGS           registers.dw[6]

#define EAX             registers.dw[0]
#define AX              registers.w[0]
#define AH              registers.b[1]
#define AL              registers.b[0]

#define EBX             registers.dw[1]
#define BX              registers.w[2]
#define BH              registers.b[5]
#define BL              registers.b[4]

#define ECX             registers.dw[2]
#define CX              registers.w[4]
#define CH              registers.b[9]
#define CL              registers.b[8]

#define EDX             registers.dw[3]
#define DX              registers.w[6]
#define DH              registers.b[13]
#define DL              registers.b[12]
```

AMCC.CPP

```
#include <stdio.h>
#include "amcc.h"

/****************************************************************************/
/*                                                                          */
/* Define macros to obtain individual bytes from a word register            */
/*                                                                          */
/****************************************************************************/

#define HIGH_BYTE(ax) (ax >> 8)
#define LOW_BYTE(ax) (ax & 0xff)

int read_configuration_area(byte function,
                                   byte bus_number,
                                   byte device_and_function,
                                   byte register_number,
                                   dword *data);

int write_configuration_area (byte function,
                                    byte bus_number,
                                    byte device_and_function,
                                    byte register_number,
                                    dword value);


void _int1a(dword *REGS);
#pragma aux _int1a=             \
        "push ebx"              \
        "pushad"                \
        "mov eax,   [ebx+0]"    \
        "mov ecx,   [ebx+8]"    \
        "mov edx,   [ebx+12]"   \
        "mov esi,   [ebx+16]"   \
        "mov edi,   [ebx+20]"   \
        "mov ebx,   [ebx+4]"    \
        "int 1ah"               \
        "push ebx"              \
        "mov ebx,   [esp+36]"   \
        "mov [ebx],eax"         \
        "pop eax"               \
        "mov [ebx+4],eax"       \
        "mov [ebx+8],ecx"       \
        "mov [ebx+12],edx"      \
        "mov [ebx+16],esi"      \
        "mov [ebx+20],edi"      \
        "pushf"                 \
        "pop ax"                \
        "mov [ebx+24],eax"      \
        "popad"                 \
        "pop ebx"               \
        parm [ebx];



/****************************************************************************/
/*                                                                          */
/*  PCI_BIOS_PRESENT                                                        */
/*                                                                          */
/* Purpose: Determine the presence of the PCI BIOS                          */
/*                                                                          */
/* Inputs: None                                                             */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    byte *hardware_mechanism                                              */
/*       Identifies the hardware characteristics used by the platform.      */
/*       Value not assigned if NULL pointer.                                */
/*          Bit 0 - Mechanism #1 supported                                  */
/*          Bit 1 - Mechanism #2 supported                                  */
/*                                                                          */
/*    word *interface_level_version                                         */
/*      PCI BIOS Version - Value not assigned if NULL pointer.              */
/*              High Byte - Major version number                            */
/*              Low Byte  - Minor version number                            */
/*                                                                          */
/*      byte *last_pci_bus_number                                           */
/*          Number of last PCI bus in the system. Value not assigned if NULL*/
/*       pointer                                                            */
/*                                                                          */
/*    Return Value - Indicates presence of PCI BIOS                         */
/*       SUCCESSFUL - PCI BIOS Present                                      */
/*       NOT_SUCCESSFUL - PCI BIOS Not Present                              */
/*                                                                          */
/****************************************************************************/

int pci_bios_present(byte *hardware_mechanism,word *interface_level_version,byte
*last_pci_bus_number)
{
   int ret_status;           /* Function Return Status. */
   byte bios_present_status; /* Indicates if PCI bios present */
   dword pci_signature;      /* PCI Signature ('P', 'C', 'I', ' ') */

   /* Load entry registers for PCI BIOS */
   AH=PCI_FUNCTION_ID;
   AL=PCI_BIOS_PRESENT;

   /* Call PCI BIOS Int 1Ah interface */
   _int1a(registers.dw);

   /* Save registers before overwritten by compiler usage of registers */
   pci_signature = EDX;

   //bios_present_status = HIGH_BYTE(AX);
   bios_present_status = AH;

   /* First check if CARRY FLAG Set, if so, BIOS not present */
   if ((FLAGS & CARRY_FLAG) == 0)
   {

      /* Next, must check that AH (BIOS Present Status) == 0 */
      if (bios_present_status == 0)
      {
         /* Check bytes in pci_signature for PCI Signature */
         if ((pci_signature & 0xff)         == 'P' &&
             ((pci_signature >> 8) & 0xff)  == 'C' &&
             ((pci_signature >> 16) & 0xff) == 'I' &&
             ((pci_signature >> 24) & 0xff) == ' ')
         {
            /* Indicate to caller that PCI bios present */
            ret_status = SUCCESSFUL;

            /* Extract calling parameters from saved registers */
            if (hardware_mechanism != NULL)
            {
                *hardware_mechanism = LOW_BYTE(AX);
            }
            if (hardware_mechanism != NULL)
            {
                *interface_level_version = BX;
            }
            if (hardware_mechanism != NULL)
            {
                *last_pci_bus_number = LOW_BYTE(CX);
            }
         }
         else
         {
                ret_status = NOT_SUCCESSFUL;
         }
      }
      else
      {
        ret_status = NOT_SUCCESSFUL;
      }
   }
   else
   {
        ret_status = NOT_SUCCESSFUL;
   }

   return (ret_status);
}

/****************************************************************************/
/*                                                                          */
/*  FIND_PCI_DEVICE                                                         */
/*                                                                          */
/* Purpose: Determine the location of PCI devices given a specific Vendor   */
/*          Device ID and Index number.  To find the first device, specify  */
/*          0 for index, 1 in index finds the second device, etc.           */
/*                                                                          */
/* Inputs:                                                                  */
/*                                                                          */
/*    word device_id                                                        */
/*       Device ID of PCI device desired                                    */
/*                                                                          */
/*    word vendor_id                                                        */
/*       Vendor ID of PCI device desired                                    */
/*                                                                          */
/*    word index                                                            */
/*       Device number to find (0 - (N-1))                                  */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    byte *bus_number                                                      */
/*       PCI Bus in which this device is found                              */
/*                                                                          */
/*    byte *device_and_function                                              */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits     */
/*                                                                           */
/*    Return Value - Indicates presence of device                            */
/*       SUCCESSFUL - Device found                                           */
/*       NOT_SUCCESSFUL - BIOS error                                         */
/*       DEVICE_NOT_FOUND - Device not found                                 */
/*       BAD_VENDOR_ID - Illegal Vendor ID (0xffff)                          */
/*                                                                           */
/***************************************************************************/

int find_pci_device(word device_id,
                    word vendor_id,
                    word index,
                    byte *bus_number,
                    byte *device_and_function)
{
   int ret_status;     /* Function Return Status */

   /* Load entry registers for PCI BIOS */
   CX=device_id;
   DX=vendor_id;
   SI=index;
   AH=PCI_FUNCTION_ID;
   AL=FIND_PCI_DEVICE;

   /* Call PCI BIOS Int 1Ah interface */
        _int1a(registers.dw);

   /* Save registers before overwritten by compiler usage of registers */

   /* First check if CARRY FLAG Set, if so, error has occurred */
   if ((FLAGS & CARRY_FLAG) == 0) {

      /* Get Return code from BIOS */
      ret_status = HIGH_BYTE(AX);

        if (ret_status == SUCCESSFUL) {

        /* Assign Bus Number, Device & Function if successful */
         if (bus_number != NULL) {
            *bus_number = HIGH_BYTE(BX);
         }
         if (device_and_function != NULL) {
            *device_and_function = LOW_BYTE(BX);
         }
      }
   }
   else {
        ret_status = NOT_SUCCESSFUL;
   }
   return (ret_status);
}

/***************************************************************************/
/*                                                                           */
/*  FIND_PCI_CLASS_CODE                                                      */
/*                                                                           */
/* Purpose: Returns the location of PCI devices that have a specific Class  */
/*          Code.                                                            */
/*                                                                           */
/* Inputs:                                                                   */
/*                                                                           */
/*    word class_code                                                        */
/*       Class Code of PCI device desired                                    */
/*                                                                           */
/*    word index                                                             */
/*       Device number to find (0 - (N-1))                                   */
/*                                                                           */
/* Outputs:                                                                  */
/*                                                                           */
/*    byte *bus_number                                                      */
/*       PCI Bus in which this device is found                              */
/*                                                                          */
/*    byte *device_and_function                                             */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits     */
/*                                                                          */
/*    Return Value - Indicates presence of device                           */
/*       SUCCESSFUL - Device found                                          */
/*       NOT_SUCCESSFUL - BIOS error                                        */
/*       DEVICE_NOT_FOUND - Device not found                                */
/*                                                                          */
/****************************************************************************/

int find_pci_class_code(dword class_code,
                word index,
                byte *bus_number,
                byte *device_and_function)
{
   int ret_status;     /* Function Return Status */

   /* Load entry registers for PCI BIOS */
   ECX=class_code;
   SI=index;
   AH=PCI_FUNCTION_ID;
   AL=FIND_PCI_CLASS_CODE;

   /* Call PCI BIOS Int 1Ah interface */
   //geninterrupt(0x1a);
   _int1a(registers.dw);

   /* Save registers before overwritten by compiler usage of registers */

   /* First check if CARRY FLAG Set, if so, error has occurred */
   if ((FLAGS & CARRY_FLAG) == 0) {

      /* Get Return code from BIOS */
      ret_status = HIGH_BYTE(AX);
      if (ret_status == SUCCESSFUL) {

         /* Assign Bus Number, Device & Function if successful */
         if (bus_number != NULL) {
            *bus_number = HIGH_BYTE(BX);
         }
         if (device_and_function != NULL) {
            *device_and_function = LOW_BYTE(BX);
         }
      }
   }
   else {
                ret_status = NOT_SUCCESSFUL;
   }

   return (ret_status);
}

/****************************************************************************/
/*                                                                          */
/*  GENERATE_SPECIAL_CYCLE                                                  */
/*                                                                          */
/* Purpose: Generates a PCI special cycle                                   */
/*                                                                          */
/* Inputs:                                                                  */
/*                                                                          */
/*    byte bus_number                                                       */
/*       PCI bus to generate cycle on                                       */
/*                                                                          */
/*    dword special_cycle_data                                              */
/*       Special Cycle Data to be generated                                 */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    Return Value - Indicates presence of device                             */
/*       SUCCESSFUL - Device found                                            */
/*       DEVICE_NOT_FOUND - Device not found                                  */
/*                                                                            */
/******************************************************************************/

int generate_special_cycle(byte bus_number,
                           dword special_cycle_data)
{
   int ret_status; /* Function Return Status */

   /* Load entry registers for PCI BIOS */
   BH=(bus_number);
   EDX=(special_cycle_data);
   AH=(PCI_FUNCTION_ID);
   AL=(FIND_PCI_CLASS_CODE);

   /* Call PCI BIOS Int 1Ah interface */
   //geninterrupt(0x1a);
   _int1a(registers.dw);

   /* Save registers before overwritten by compiler usage of registers */

   /* First check if CARRY FLAG Set, if so, error has occurred */
   if ((FLAGS & CARRY_FLAG) == 0) {

      /* Get Return code from BIOS */
      ret_status = HIGH_BYTE(AX);
   }
   else {
      ret_status = NOT_SUCCESSFUL;
   }

   return (ret_status);
}

/******************************************************************************/
/*                                                                            */
/*  READ_CONFIGURATION_BYTE                                                   */
/*                                                                            */
/* Purpose: Reads a byte from the configuration space of a specific device  */
/*                                                                            */
/* Inputs:                                                                    */
/*                                                                            */
/*    byte bus_number                                                         */
/*       PCI bus to read configuration data from                              */
/*                                                                            */
/*    byte device_and_function                                                */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits     */
/*                                                                            */
/*    byte register_number                                                    */
/*       Register Number of configuration space to read                       */
/*                                                                            */
/* Outputs:                                                                   */
/*                                                                            */
/*    byte *byte_read                                                         */
/*       Byte read from Configuration Space                                   */
/*                                                                            */
/*    Return Value - Indicates presence of device                             */
/*       SUCCESSFUL - Device found                                            */
/*       NOT_SUCCESSFUL - BIOS error                                          */
/*       BAD_REGISTER_NUMBER - Invalid Register Number                        */
/*                                                                            */
/******************************************************************************/

int read_configuration_byte(byte bus_number,
                            byte device_and_function,
                            byte register_number,
                            byte *byte_read)
{
   int ret_status; /* Function Return Status */
   dword data;

   /* Call read_configuration_area function with byte data */
   ret_status = read_configuration_area(READ_CONFIG_BYTE,
                                        bus_number,
                                        device_and_function,
                                        register_number,
                                        &data);
   if (ret_status == SUCCESSFUL) {

      /* Extract byte */
                *byte_read = (byte)(data & 0xff);
   }

   return (ret_status);
}


/****************************************************************************/
/*                                                                          */
/*  READ_CONFIGURATION_WORD                                                 */
/*                                                                          */
/* Purpose: Reads a word from the configuration space of a specific device  */
/*                                                                          */
/* Inputs:                                                                  */
/*                                                                          */
/*    byte bus_number                                                       */
/*       PCI bus to read configuration data from                            */
/*                                                                          */
/*    byte device_and_function                                              */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits     */
/*                                                                          */
/*    byte register_number                                                  */
/*       Register Number of configuration space to read                     */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    word *word_read                                                       */
/*       Word read from Configuration Space                                 */
/*                                                                          */
/*    Return Value - Indicates presence of device                           */
/*       SUCCESSFUL - Device found                                          */
/*       NOT_SUCCESSFUL - BIOS error                                        */
/*       BAD_REGISTER_NUMBER - Invalid Register Number                      */
/*                                                                          */
/****************************************************************************/

int read_configuration_word(byte bus_number,
                            byte device_and_function,
                            byte register_number,
                            word *word_read)
{
   int ret_status; /* Function Return Status */
   dword data;

   /* Call read_configuration_area function with word data */
   ret_status = read_configuration_area(READ_CONFIG_WORD,
                                        bus_number,
                                        device_and_function,
                                        register_number,
                                        &data);
   if (ret_status == SUCCESSFUL) {

      /* Extract word */
      *word_read = (word)(data & 0xffff);
   }

   return (ret_status);
}

/****************************************************************************/
```

```
/*                                                                          */
/*  READ_CONFIGURATION_DWORD                                                */
/*                                                                          */
/* Purpose: Reads a dword from the configuration space of a specific device */
/*                                                                          */
/* Inputs:                                                                  */
/*                                                                          */
/*    byte bus_number                                                       */
/*       PCI bus to read configuration data from                            */
/*                                                                          */
/*    byte device_and_function                                              */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits     */
/*                                                                          */
/*    byte register_number                                                  */
/*       Register Number of configuration space to read                     */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    dword *dword_read                                                     */
/*       Dword read from Configuration Space                                */
/*                                                                          */
/*    Return Value - Indicates presence of device                           */
/*       SUCCESSFUL - Device found                                          */
/*       NOT_SUCCESSFUL - BIOS error                                        */
/*       BAD_REGISTER_NUMBER - Invalid Register Number                      */
/*                                                                          */
/****************************************************************************/

word read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read)
{
   int ret_status; /* Function Return Status */
   dword data;

   /* Call read_configuration_area function with dword data */
   ret_status = read_configuration_area(READ_CONFIG_DWORD,
                                        bus_number,
                                        device_and_function,
                                        register_number,
                                        &data);
   if (ret_status == SUCCESSFUL) {

      /* Extract dword */
      *dword_read = data;
   }

   return (ret_status);
}

/****************************************************************************/
/*                                                                          */
/*  READ_CONFIGURATION_AREA                                                 */
/*                                                                          */
/* Purpose: Reads a byte/word/dword from the configuration space of a       */
/*          specific device                                                 */
/*                                                                          */
/* Inputs:                                                                  */
/*                                                                          */
/*    byte bus_number                                                       */
/*       PCI bus to read configuration data from                            */
/*                                                                          */
/*    byte device_and_function                                              */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits     */
/*                                                                          */
/*    byte register_number                                                  */
/*       Register Number of configuration space to read                     */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    dword *dword_read                                                   */
/*       Dword read from Configuration Space                              */
/*                                                                        */
/*    Return Value - Indicates presence of device                         */
/*       SUCCESSFUL - Device found                                        */
/*       NOT_SUCCESSFUL - BIOS error                                      */
/*       BAD_REGISTER_NUMBER - Invalid Register Number                    */
/*                                                                        */
/**************************************************************************/

//static?
 int read_configuration_area(byte function,
                                   byte bus_number,
                                   byte device_and_function,
                                   byte register_number,
                                   dword *data)
{
   int ret_status; /* Function Return Status */
   /* Load entry registers for PCI BIOS */
   BH=(bus_number);
   BL=(device_and_function);
   DI=(register_number);
   AH=(PCI_FUNCTION_ID);
   AL=(function);

   /* Call PCI BIOS Int 1Ah interface */
   _int1a(registers.dw);

   /* Save registers before overwritten by compiler usage of registers */

   /* First check if CARRY FLAG Set, if so, error has occurred */
   if ((FLAGS & CARRY_FLAG) == 0)
   {

      /* Get Return code from BIOS */
      ret_status = HIGH_BYTE(AX);

      /* If successful, return data */
      if (ret_status == SUCCESSFUL)
         *data = ECX;
      else
        printf("HIGH BYTE of AX is bad\n");
   }
   else
   {
        printf("CARRY FLAG is SET\n");
        ret_status = NOT_SUCCESSFUL;
   }


   return (ret_status);
}

/**************************************************************************/
/*                                                                        */
/*  WRITE_CONFIGURATION_BYTE                                              */
/*                                                                        */
/* Purpose: Writes a byte to the configuration space of a specific device */
/*                                                                        */
/* Inputs:                                                                */
/*                                                                        */
/*    byte bus_number                                                     */
/*       PCI bus to write configuration data to                           */
/*                                                                        */
/*    byte device_and_function                                            */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits   */
/*                                                                        */
/*    byte register_number                                                */
/*       Register Number of configuration space to write                  */
/*                                                                        */
/*    byte byte_to_write                                                  */
/*       Byte to write to Configuration Space                               */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    Return Value - Indicates presence of device                           */
/*       SUCCESSFUL - Device found                                          */
/*       NOT_SUCCESSFUL - BIOS error                                        */
/*       BAD_REGISTER_NUMBER - Invalid Register Number                      */
/*                                                                          */
/****************************************************************************/

int write_configuration_byte(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             byte byte_to_write)
{
   int ret_status; /* Function Return Status */

   /* Call write_configuration_area function with byte data */
   ret_status = write_configuration_area(WRITE_CONFIG_BYTE,
                                         bus_number,
                                         device_and_function,
                                         register_number,
                                         byte_to_write);
   return (ret_status);
}


/****************************************************************************/
/*                                                                          */
/*  WRITE_CONFIGURATION_WORD                                                */
/*                                                                          */
/* Purpose: Writes a word to the configuration space of a specific device   */
/*                                                                          */
/* Inputs:                                                                  */
/*                                                                          */
/*    byte bus_number                                                       */
/*       PCI bus to read configuration data from                            */
/*                                                                          */
/*    byte device_and_function                                              */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits     */
/*                                                                          */
/*    byte register_number                                                  */
/*       Register Number of configuration space to read                     */
/*                                                                          */
/*    word word_to_write                                                    */
/*       Word to write to Configuration Space                               */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    Return Value - Indicates presence of device                           */
/*       SUCCESSFUL - Device found                                          */
/*       NOT_SUCCESSFUL - BIOS error                                        */
/*       BAD_REGISTER_NUMBER - Invalid Register Number                      */
/*                                                                          */
/****************************************************************************/

int write_configuration_word(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             word word_to_write)
{
   int ret_status; /* Function Return Status */

   /* Call read_configuration_area function with word data */
   ret_status = write_configuration_area(WRITE_CONFIG_WORD,
                                         bus_number,
                                         device_and_function,
                                         register_number,
                                         word_to_write);
   return (ret_status);
}

/****************************************************************************/
/*                                                                          */
/*  WRITE_CONFIGURATION_DWORD                                               */
/*                                                                          */
/* Purpose: Reads a dword from the configuration space of a specific device */
/*                                                                          */
/* Inputs:                                                                  */
/*   .                                                                      */
/*    byte bus_number                                                       */
/*       PCI bus to read configuration data from                            */
/*                                                                          */
/*    byte device_and_function                                              */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits     */
/*                                                                          */
/*    byte register_number                                                  */
/*       Register Number of configuration space to read                     */
/*                                                                          */
/*    dword dword_to_write                                                  */
/*       Dword to write to Configuration Space                              */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    Return Value - Indicates presence of device                           */
/*       SUCCESSFUL - Device found                                          */
/*       NOT_SUCCESSFUL - BIOS error                                        */
/*       BAD_REGISTER_NUMBER - Invalid Register Number                      */
/*                                                                          */
/****************************************************************************/

word write_configuration_dword(byte bus_number,
                              byte device_and_function,
                              byte register_number,
                              dword dword_to_write)
{
   int ret_status; /* Function Return Status */

   /* Call write_configuration_area function with dword data */
   ret_status = write_configuration_area(WRITE_CONFIG_DWORD,
                                         bus_number,
                                         device_and_function,
                                         register_number,
                                         dword_to_write);
   return (ret_status);
}

/****************************************************************************/
/*                                                                          */
/*  WRITE_CONFIGURATION_AREA                                                */
/*                                                                          */
/* Purpose: Writes a byte/word/dword to the configuration space of a        */
/*          specific device                                                 */
/*                                                                          */
/* Inputs:                                                                  */
/*                                                                          */
/*    byte bus_number                                                       */
/*       PCI bus to read configuration data from                            */
/*                                                                          */
/*    byte device_and_function                                              */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits     */
/*                                                                          */
/*    byte register_number                                                  */
/*       Register Number of configuration space to read                     */
/*                                                                          */
/*    dword value                                                           */
/*       Value to write to Configuration Space                              */
/*                                                                          */
/* Outputs:                                                                 */
/*                                                                          */
/*    Return Value - Indicates presence of device                           */
/*       SUCCESSFUL - Device found                                           */
/*       NOT_SUCCESSFUL - BIOS error                                         */
/*       BAD_REGISTER_NUMBER - Invalid Register Number                       */
/*                                                                           */
/****************************************************************************/

int write_configuration_area (byte function,
                                    byte bus_number,
                                    byte device_and_function,
                                    byte register_number,
                                    dword value)
{
   int ret_status; /* Function Return Status */

   /* Load entry registers for PCI BIOS */
   BH=(bus_number);
   BL=(device_and_function);
   ECX=(value);
   DI=(register_number);
   AH=(PCI_FUNCTION_ID);
   AL=(function);

   /* Call PCI BIOS Int 1Ah interface */
   //geninterrupt(0x1a);
   _int1a(registers.dw);

   /* Save registers before overwritten by compiler usage of registers */

   /* First check if CARRY FLAG Set, if so, error has occurred */
   if ((FLAGS & CARRY_FLAG) == 0) {

      /* Get Return code from BIOS */
      ret_status = HIGH_BYTE(AX);
   }
   else {
      ret_status = NOT_SUCCESSFUL;
   }

   return (ret_status);
}
```

Appendix B: General Code

This appendix supplies the 1801/2hc header file that is included in the source code used in this document, along with general source code written to peak-n-poke the registers of the KPCI-1801/2hc boards. There are three source files included below under the ‘General Sources’ section.

180xhc.H

```
// KPCI-1801/2HC Series constants
```

```
#define VID_KEITHLEY 0x11f3
```

```
// KPCI-180xHC Series Boards Device ID #s
```

```
#define DID_1801  0x1801
#define DID_1802  0x1802

// *********** OFFSETS ************

// ** BADDR1 & BADDR2 register offsets...
#define CONTROL_REG     0x04
#define SCLK_CONFIG_REG 0x05
#define BURST_VAL_REG   0x06

// ** BADDR1: A/D register offsets...
#define AD_DATA         0x00
#define AD_CH_CNT_REG   0x08
#define TRIG_SOURCE_REG 0x10
#define DIG_TRIG_REG    0x11
#define TRIG_COUNT_REG  0x14
#define AD_QRAM_OFFSET      0x4000
// BADDR1 **

// ** BADDR2: D/A regiser offsets...
#define DA_DATA             0x00
#define DA_QRAM_OFFSET      0x80
// BADDR2 **

// ** BADDR3: Register offsets...
// -- 8254 Counter/Timer
#define CNTR_8254_CONTROL   0x10
#define CNTR_8254_FORMAT    0x0C
#define CNTR_8254_2_CNT     0x08
#define CNTR_8254_1_CNT     0x04
#define CNTR_8254_0_CNT     0x00
// -- Interrupt registers
#define INT_CONTROL     0x14
#define INT_ENABLE      0x18
// -- Digital IO
#define DIG_IO       0x24
// BADDR3 **


// *********** DEFINITIONS ************

// ** BADDR1 & BADDR2 definitions...
#define FIFO_EN     0x02
#define BUSMSTR     0x01
#define BURST       0x04

// sample clock selections
#define CNTR2OUT    0x00
#define CNTR1OUT    0x01
#define CNTR0OUT    0x02
#define SWWRITE     0x04
#define EXT_PPOL    0x05
#define EXT_NPOL    0x06

// ** BADDR1: A/D definitions...
// QRAM parameters
// -- polarity
#define UNIPOLAR    0x0200
#define BIPOLAR     0x0000
// -- gains for 1802HC
#define GAIN1       0x0000
#define GAIN2       0x0080
#define GAIN4       0x0100
#define GAIN8       0x0180
// -- gains for 1801HC
#define GAIN10      0x0080
#define GAIN50      0x0100
#define GAIN250     0x0180
// -- input terminations
#define DE_0to31    0x0000
#define CH_SHORT    0x0020
```

```
#define SE_LO32     0x0040
#define SE_HI32     0x0060
// -- channel selection
#define CH0_SEL     0x0000      // CH32_SEL for SE High channel mode
#define CH1_SEL     0x0001      // CH33_SEL for SE High channel mode
#define CH2_SEL     0x0002      // CH34_SEL for SE High channel mode
#define CH3_SEL     0x0003      // CH35_SEL for SE High channel mode
#define CH4_SEL     0x0004      // CH36_SEL for SE High channel mode
#define CH5_SEL     0x0005      // CH37_SEL for SE High channel mode
#define CH6_SEL     0x0006      // CH38_SEL for SE High channel mode
#define CH7_SEL     0x0007      // CH39_SEL for SE High channel mode
#define CH8_SEL     0x0008      // CH40_SEL for SE High channel mode
#define CH9_SEL     0x0009      // CH41_SEL for SE High channel mode
#define CH10_SEL    0x000A      // CH42_SEL for SE High channel mode
#define CH11_SEL    0x000B      // CH43_SEL for SE High channel mode
#define CH12_SEL    0x000C      // CH44_SEL for SE High channel mode
#define CH13_SEL    0x000D      // CH45_SEL for SE High channel mode
#define CH14_SEL    0x000E      // CH46_SEL for SE High channel mode
#define CH15_SEL    0x000F      // CH47_SEL for SE High channel mode
#define CH16_SEL    0x0010      // CH48_SEL for SE High channel mode
#define CH17_SEL    0x0011      // CH49_SEL for SE High channel mode
#define CH18_SEL    0x0012      // CH50_SEL for SE High channel mode
#define CH19_SEL    0x0013      // CH51_SEL for SE High channel mode
#define CH20_SEL    0x0014      // CH52_SEL for SE High channel mode
#define CH21_SEL    0x0015      // CH53_SEL for SE High channel mode
#define CH22_SEL    0x0016      // CH54_SEL for SE High channel mode
#define CH23_SEL    0x0017      // CH55_SEL for SE High channel mode
#define CH24_SEL    0x0018      // CH56_SEL for SE High channel mode
#define CH25_SEL    0x0019      // CH57_SEL for SE High channel mode
#define CH26_SEL    0x001A      // CH58_SEL for SE High channel mode
#define CH27_SEL    0x001B      // CH59_SEL for SE High channel mode
#define CH28_SEL    0x001C      // CH60_SEL for SE High channel mode
#define CH29_SEL    0x001D      // CH61_SEL for SE High channel mode
#define CH30_SEL    0x001E      // CH62_SEL for SE High channel mode
#define CH31_SEL    0x001F      // CH63_SEL for SE High channel mode

// TRIGGER parameters
// -- counter
#define TRIG_COUNTER_EN 0x10
// -- digital trigger (tgin)
#define DIG_TRIG_EN     0x01
#define TRIG_SEL        0x00
#define GATE_SEL        0x02
#define TGIN_NEGPOL     0x00
#define TGIN_POSPOL     0x04
#define TGIN_EN         0x08

// ** BADDR2: D/A definitions...
#define DAC_OUT_ACTIVE  0x80
#define DAC1_SEL        0x01
#define DAC0_SEL        0x00
// BADDR2 **

// ** BADDR3: definitions...
// BADDR3 **
```

General Sources

1. 180x_gen.cpp
   Locates a 1801hc board, reads its configuration, then shows base address values of each valid region found. Can change the Device ID to 1802hc by commenting out 1801hc's DID in STEP 2 of get_board_info() routine.

```
/*****************************************************************************/
/*                                                                           */
/*  COMPILER AND VERSION                                                     */
/*  --------------------                                                     */
/*  Watcom IDE for C/C++, Version 11.0                                       */
/*                                                                          */
/*  The compiler/IDE was installed on Windows 95 and the code was compiled   */
/*  for the 32-bit DOS environment to be used with the "DOS4GW.EXE" DOS      */
/*  extender that was included with the compiler.                            */
/*                                                                          */
/*  The source files used in compiling the test code are as follows:         */
/*                                                                          */
/*   [.cpp]                                                                 */
/*   amcc.cpp                                                               */
/*   180x_gen.cpp                                                            */
/*                                                                          */
/*   [.h]                                                                   */
/*   amcc.h                                                                 */
/*   180xhc.h                                                              */
/*   -other Watcom supplied header files not shown here.                     */
/*                                                                          */
/*   The compiled EXE file requires the file "dos4gw.exe" to be located in   */
/*   it's same directory.                                                   */
/*                                                                          */
/*   Note: Be careful to install the Watcom Compiler/IDE and code direct-    */
/*         ories using only 8 character folder names or else the IDE         */
/*         Editor won't find your files.                                    */
/*                                                                          */
/***************************************************************************/
/*
**  File:   180x_gen.cpp
**  By:     T. Smith
**  Date:   8/13/1999
*/

// The following header files are included with the Watcom Compiler.
#include <stdio.h>
#include <time.h>
#include "amcc.h"
#include "180xhc.h"


// A copy of the AMCC.H "pointer" typedef has been included below.

        pointer Baddr0[4];    // Memory mapped base address to access AMCC
                              // S5933 operation registers.  Up to a maximum
                              // of 4 KPCI-180xhcs.

        pointer Baddr1[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr2[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr3[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

// THE FOLLOWING PROTOTYPES ARE FOR FUNCTIONS HELD IN THE FILE "AMCC.CPP".
//  THESE FUNCTIONS ARE SUPPLIED BY AMCC.

extern int pci_bios_present(byte *hardware_mechanism,
                            word *interface_level_version,
                            byte *last_pci_bus_number);

extern int find_pci_device(word device_id, word vendor_id,
                            word index,
                            byte *bus_number,
                            byte *device_and_function);

extern word read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read);
```

```
int get_board_info(void);

//********************************** MAIN ***********************************

void  main(void)
{
    int board, Num_of_180xhcs;

    Num_of_180xhcs = get_board_info();
    if (Num_of_180xhcs == 0)
        return;
    else
    {
        printf("Using Board 0 (first KPCI-1801HC found).\n\n");
        board = 0;

            // Setup the memory maps  pointers to the A/D and D/A QRAM.
            volatile dword *pADS_DATA = (Baddr1[board].dp + 0x00);
            volatile word *pADS_QRAM = (Baddr1[board].wp + 0x2000);
            volatile dword *pDAC_DATA = (Baddr2[board].dp + 0x00);
            volatile char *pDAC_QRAM = (Baddr2[board].bp + 0x80);

            printf("BADDR0 of board %d starts at address %lx. \n\n", board,
Baddr0[board].dw);

            printf("BADDR1 of board %d starts at address %lx. \n", board, Baddr1[board].dw);
            printf("ADS_DATA of board %d = %lx address.\n", board, pADS_DATA);
            printf("ADS_QRAM of board %d = %lx address.\n\n", board, pADS_QRAM);

            printf("BADDR2 of board %d starts at address %lx. \n", board, Baddr2[board].dw);
            printf("DAC_DATA of board %d = %lx address.\n", board, pDAC_DATA);
            printf("DAC_QRAM of board %d = %lx address.\n\n", board, pDAC_QRAM);

            printf("BADDR3 of board %d starts at address %lx. \n", board, Baddr3[board].dw);

    } // (else Num_of_180xhcs)

}//  ***** end of Main


// ********* GET_BOARD_INFO **********

int get_board_info(void)
{
        // Here are the variables for Step 1

        byte hmech;
        word ilevel;
        byte lastbus;

        // Here are the variables for Step2

        word did;   // Device ID (did) => KPCI 1801
        word vid;   // Vendor ID (vid) => Keithley Instruments
        word index=0;
        int UUT_search;
        byte Number_of_UUTs;
        byte bus[4];
        byte devfunc[4];

        // Here are the variables for Step 3

        int board;
        int baddr_seek;

        printf ( "STARTING NOW \n");

        //******************  STEP  1 *****************
        //
        //  CHECK FOR THE PCI BIOS.
        //
        //***********************************************

        if(pci_bios_present(&hmech,&ilevel,&lastbus)==SUCCESSFUL)
        {
            printf("PCI Bios Found.\n");
        }
        else
        {
            printf(" PCI Bios Not Found.\n");
            return 0;
        }

        //*********************** STEP  2 ******************************
        //
        // VERIFY THE NUMBER OF KPCI-1801/2HCs WITH THE CORRECT VID / DID.
        //
        //************************************************************

        // Search the PCI bus for KPCI-1801/2HC boards.
        // Each pass through the loop will identify one UUT.

        did = DID_1801;         // Device ID is for the KPCI 1801HC
        // did = DID_1802;          // Device ID for the KPCI 1802HC
        vid = VID_KEITHLEY;     // Vendor ID is for Keithley Instruments

        // Look for the first occurrence of a KPCI-1801/2HC on the PCI bus.
        index = 0;
        UUT_search = find_pci_device(did,
                                    vid,
                                    index,
                                    &bus[index],
                                    &devfunc[index]);

        // If one was found then look for others.
        while ( UUT_search == SUCCESSFUL)
        {
            index++;
            UUT_search = find_pci_device(did,
                                        vid,
                                        index,
                                        &bus[index],
                                        &devfunc[index]);

        } // while (find_pci_device(did, vid, index, &bus[index] .....

        // If none were found then send a message and exit.
        if(index == 0)
        {
            printf("No UUTs found which match KPCI-%x's VID and/or DID! \n", did);
            return 0;
        } // if(index == 0)

        Number_of_UUTs = (unsigned char) index;

        //****************************** STEP  3 *************************************
        //
        //  READ THE PCI CONFIG SPACE OF EACH KPCI-1801/2hc AND SAVE THEIR UNIQUE "BADDR0",
        //  "BADDR1", "BADDR2", and BADDR3".  "BADDR" STANDS FOR BASE ADDRESS.
        //  FOUR BADDRs GET ASSIGNED FOR EACH KPCI-1801/2hc BY THE PLUG'N PLAY BIOS DURING
STARTUP.
        //
        //*******************************************************************************

        for (board=0; board < Number_of_UUTs ;board++)
        {

            // Find each boards base address (32 bit, BADDR0) for reading / writing
            //  to the AMCC S5933 operation registers.

            baddr_seek =  read_configuration_dword(bus[board],
                                    devfunc[board],
                                    BADDR_0_CFIG_SPACE_OFFSET,
                                    &(Baddr0[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr0 of board %d not found!",board);
            }

            // Now get (32 bit) BADDR1 for AMCC S5933 pass through reads and writes.  This
            //  is the base address used for reading and writing A/D register data from the
1801/2hc.

            baddr_seek =  read_configuration_dword(bus[board],
                                    devfunc[board],
                                    BADDR_1_CFIG_SPACE_OFFSET,
                                    &(Baddr1[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr1 of board %d not found!\n",board);
            }


            // Now get (32 bit) BADDR2 for AMCC S5933 pass through reads and writes.  This
            //  is the base address used for reading and writing D/A register data from the
1801/2hc.

            baddr_seek =  read_configuration_dword(bus[board],
                                    devfunc[board],
                                    BADDR_2_CFIG_SPACE_OFFSET,
                                    &(Baddr2[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr2 of board %d not found!\n",board);
            }

            //  Now get (32 bit) BADDR3 for AMCC S5933 pass through reads and writes.  This
            //  is the base address used for reading and writing timer/interrupt/digital io
register
            //  data from the 1801/2hc.

            baddr_seek =  read_configuration_dword(bus[board],
                                    devfunc[board],
                                    BADDR_3_CFIG_SPACE_OFFSET,
                                    &(Baddr3[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr3 of board %d not found!\n",board);
            }

        } // end of for (board=0; board<index ;board++)

        return index;
} // (void get_board_info)
```

2. ptrd.cpp
   Locates an 1801hc board, reads its configuration, asks the user for the transfer width, base address number, and offset then returns the data. Can change the Device ID to 1802hc by commenting out 1801hc's DID in STEP 2 of get_board_info() routine.

```
/******************************************************************************/
/*                                                                            */
/*  COMPILER AND VERSION                                                      */
/*  --------------------                                                      */
/*  Watcom IDE for C/C++, Version 11.0                                        */
/*                                                                            */
/*  The compiler/IDE was installed on Windows 95 and the code was compiled   */
/*  for the 32-bit DOS environment to be used with the "DOS4GW.EXE" DOS      */
/*  extender that was included with the compiler.                            */
/*                                                                           */
/*  The source files used in compiling the test code are as follows:         */
/*                                                                           */
/*   [.cpp]                                                                  */
/*   amcc.cpp                                                                */
/*   ptrd.cpp                                                             */
/*                                                                           */
/*   [.h]                                                                    */
/*   amcc.h                                                                  */
/*   180xhc.h                                                              */
/*   -other Watcom supplied header files not shown here.                     */
/*                                                                           */
/*   The compiled EXE file requires the file "dos4gw.exe" to be located in   */
/*   it's same directory.                                                    */
/*                                                                           */
/*   Note: Be careful to install the Watcom Compiler/IDE and code direct-    */
/*         ories using only 8 character folder names or else the IDE         */
/*         Editor won't find your files.                                     */
/*                                                                           */
/***************************************************************************/
/*
**  File:   ptrd.cpp
**  By:     T.Smith
**  Date:   8/13/1999
*/

// The following header files are included with the Watcom Compiler.
#include <stdio.h>
#include <time.h>
#include "amcc.h"
#include "180xhc.h"

// A copy of the AMCC.H "pointer" typedef has been included below.

        pointer Baddr0[4];    // Memory mapped base address to access AMCC
                              // S5933 operation registers.  Up to a maximum
                              // of 4 KPCI-180xhcs.

        pointer Baddr1[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr2[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr3[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

// THE FOLLOWING PROTOTYPES ARE FOR FUNCTIONS HELD IN THE FILE "AMCC.CPP".
//  THESE FUNCTIONS ARE SUPPLIED BY AMCC.

extern int pci_bios_present(byte *hardware_mechanism,
                            word *interface_level_version,
                            byte *last_pci_bus_number);

extern int find_pci_device(word device_id, word vendor_id,
                            word index,
                            byte *bus_number,
                            byte *device_and_function);

extern word read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read);

int get_board_info(void);
```

```
//********************************** MAIN ************************************

void  main(void)
{
    int board, Num_of_180xhcs;
    int ba_no, transfer_size;
    dword offset;
    volatile dword *pBAOFFdw;
    volatile word *pBAOFFw;
    volatile char *pBAOFFb;

    Num_of_180xhcs = get_board_info();
    if (Num_of_180xhcs == 0)
        return;
    else
    {
        printf("\nTHIS EXECUTABLE PERFORMS A PT READ ONLY!\n\n");
        board = 0;  // using first board found
        printf("Using Board 0 (first board found).\n\n");

        printf("\nSelect bit transfer size: (8, 16, or 32) ");
        scanf("%i", &transfer_size);

        printf("\nEnter a valid Base Address Number: (integer) ");
        scanf("%i", &ba_no);

        printf("\nEnter Offset: 0x ");
        scanf("%lx", &offset);

        switch (transfer_size)
        {
            case 16:
                offset = offset/2;
                // -- printf("offset/2 = 0x%lx\n", offset);

                switch (ba_no)
                {
                    case 0:
                        pBAOFFw = Baddr0[board].wp + offset;
                        break;
                    case 1:
                        pBAOFFw = Baddr1[board].wp + offset;
                        break;
                    case 2:
                        pBAOFFw = Baddr2[board].wp + offset;
                        break;
                    case 3:
                        pBAOFFw = Baddr3[board].wp + offset;
                        break;
                    default:
                        printf("ERROR: Invalid BADDR number!  TERMINATING program!\n");
                        return;
                } // switch (ba_no)

                //-- printf("\nBase Address %i with Offset 0x%lx was entered.\n", ba_no,
offset);
                printf("Data = 0x%x\n", *pBAOFFw);

                break;
            case 32:
                offset = offset/4;
                // -- printf("offset/4 = 0x%lx\n", offset);

                switch (ba_no)
                {
                    case 0:
                        pBAOFFdw = Baddr0[board].dp + offset;
                        break;
                    case 1:
                        pBAOFFdw = Baddr1[board].dp + offset;
                        break;
                    case 2:
                        pBAOFFdw = Baddr2[board].dp + offset;
                        break;
                    case 3:
                        pBAOFFdw = Baddr3[board].dp + offset;
                        break;
                    default:
                        printf("ERROR: Invalid BADDR number!  TERMINATING program!\n");
                        return;
                } // switch (ba_no)

                // -- printf("\nBase Address %i with Offset 0x%lx was entered.\n", ba_no,
offset);
                printf("Data = 0x%lx\n", *pBAOFFdw);

                break;
            default:
                if(transfer_size != 8)
                {
                    printf("\nInvalid bit transfer size selected. Defaulted to 8 bits.\n");
                    transfer_size = 8;
                } // if(transfer..)

                switch (ba_no)
                {
                    case 0:
                        pBAOFFb = Baddr0[board].bp + offset;
                        break;
                    case 1:
                        pBAOFFb = Baddr1[board].bp + offset;
                        break;
                    case 2:
                        pBAOFFb = Baddr2[board].bp + offset;
                        break;
                    case 3:
                        pBAOFFb = Baddr3[board].bp + offset;
                        break;
                    default:
                        printf("ERROR: Invalid BADDR number!  TERMINATING program!\n");
                        return;
                } // switch (ba_no)

                // -- printf("\nBase Address %i with Offset 0x%lx was entered.\n", ba_no,
offset);
                printf("Data = 0x%x\n", *pBAOFFb);

                break;
        } // switch (transfer_size)

    } // (else Num_of_180xhcs)

}//  ***** end of Main

// ********* GET_BOARD_INFO **********
int get_board_info(void)
{
    ...
} // (void get_board_info)
```

3. ptwr.cpp
   Locates an 1801hc board, reads its configuration, asks the user for the transfer width, base address number, offset and data to write. It does a read of the address given and displays the data. This is to check that the data was written correctly. Can change the Device ID to 1802hc by commenting out 1801hc’s DID in STEP 2 of get_board_info() routine.

```
/*************************************************************************/
```

```
/*                                                                                          */
```

```
/*  COMPILER AND VERSION                                                    */
/*  --------------------                                                    */
/*  Watcom IDE for C/C++, Version 11.0                                      */
/*                                                                          */
/*  The compiler/IDE was installed on Windows 95 and the code was compiled   */
/*  for the 32-bit DOS environment to be used with the "DOS4GW.EXE" DOS      */
/*  extender that was included with the compiler.                           */
/*                                                                          */
/*  The source files used in compiling the test code are as follows:        */
/*                                                                          */
/*   [.cpp]                                                                 */
/*   amcc.cpp                                                               */
/*   ptwr.cpp                                                            */
/*                                                                          */
/*   [.h]                                                                   */
/*   amcc.h                                                                 */
/*   180xhc.h                                                             */
/*   -other Watcom supplied header files not shown here.                    */
/*                                                                          */
/*   The compiled EXE file requires the file "dos4gw.exe" to be located in   */
/*   it's same directory.                                                   */
/*                                                                          */
/*   Note: Be careful to install the Watcom Compiler/IDE and code direct-    */
/*         ories using only 8 character folder names or else the IDE        */
/*         Editor won't find your files.                                    */
/*                                                                          */
/****************************************************************************/

// The following header files are included with the Watcom Compiler.
#include <stdio.h>
#include <time.h>
#include "amcc.h"
#include "180xhc.h"

// A copy of the AMCC.H "pointer" typedef has been included below.

        pointer Baddr0[4];    // Memory mapped base address to access AMCC
                              // S5933 operation registers.  Up to a maximum
                              // of 4 KPCI-180xhcs.

        pointer Baddr1[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr2[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr3[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.


// THE FOLLOWING PROTOTYPES ARE FOR FUNCTIONS HELD IN THE FILE "AMCC.CPP".
//  THESE FUNCTIONS ARE SUPPLIED BY AMCC.

extern int pci_bios_present(byte *hardware_mechanism,
                            word *interface_level_version,
                            byte *last_pci_bus_number);

extern int find_pci_device(word device_id, word vendor_id,
                            word index,
                            byte *bus_number,
                            byte *device_and_function);

extern word read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read);

int get_board_info(void);
```

```
//********************************** MAIN ************************************

void  main(void)
{
    int board, Num_of_180xhcs;
    int ba_no, transfer_size;
    dword offset;
    volatile dword *pBAOFFdw;
    volatile word *pBAOFFw;
    volatile char *pBAOFFb;
    word dataval_w;
    dword dataval_dw;
    byte dataval_b;


    Num_of_180xhcs = get_board_info();
    if (Num_of_180xhcs == 0)
        return;
    else
    {
        printf("\nTHIS EXECUTABLE PERFORMS A PT WRITE ONLY!\n\n");
        board = 0;  // using first board found
        printf("Using Board 0 (first board found).\n\n");

        printf("\nSelect bit transfer width: (8, 16, or 32) ");
        scanf("%i", &transfer_size);

        printf("\nEnter a valid Base Address Number: (integer) ");
        scanf("%i", &ba_no);

        printf("\nEnter Offset: 0x ");
        scanf("%lx", &offset);

        switch (transfer_size)
        {
            case 16:
                offset = offset/2;

                switch (ba_no)
                {
                    case 0:
                        pBAOFFw = Baddr0[board].wp + offset;
                        break;
                    case 1:
                        pBAOFFw = Baddr1[board].wp + offset;
                        break;
                    case 2:
                        pBAOFFw = Baddr2[board].wp + offset;
                        break;
                    case 3:
                        pBAOFFw = Baddr3[board].wp + offset;
                        break;
                    default:
                        printf("ERROR: Invalid BADDR number!  TERMINATING program!\n");
                        return;
                } // switch (ba_no)

                //-- printf("\nBase Address %i with Offset 0x%lx was entered.\n", ba_no,
offset);
                printf("\nEnter data (16 Bits) to write: 0x ");
                scanf("%x",&dataval_w);
                *pBAOFFw = dataval_w;
                printf("Data = 0x%x\n", *pBAOFFw);

                break;
            case 32:
                offset = offset/4;
                // -- printf("offset/4 = 0x%lx\n", offset);

                switch (ba_no)
                {
                    case 0:
                        pBAOFFdw = Baddr0[board].dp + offset;
                        break;
                    case 1:
                        pBAOFFdw = Baddr1[board].dp + offset;
                        break;
                    case 2:
                        pBAOFFdw = Baddr2[board].dp + offset;
                        break;
                    case 3:
                        pBAOFFdw = Baddr3[board].dp + offset;
                        break;
                    default:
                        printf("ERROR: Invalid BADDR number!  TERMINATING program!\n");
                        return;
                } // switch (ba_no)

                // -- printf("\nBase Address %i with Offset 0x%lx was entered.\n", ba_no,
offset);
                printf("\nEnter data (32 Bits) to write: 0x ");
                scanf("%lx",&dataval_dw);
                *pBAOFFdw = dataval_dw;
                printf("Data = 0x%lx\n", *pBAOFFdw);

                break;
            default:
                if(transfer_size != 8)
                {
                    printf("\nInvalid bit transfer size selected. Defaulted to 8 bits.\n");
                    transfer_size = 8;
                } // if(transfer..)

                switch (ba_no)
                {
                    case 0:
                        pBAOFFb = Baddr0[board].bp + offset;
                        break;
                    case 1:
                        pBAOFFb = Baddr1[board].bp + offset;
                        break;
                    case 2:
                        pBAOFFb = Baddr2[board].bp + offset;
                        break;
                    case 3:
                        pBAOFFb = Baddr3[board].bp + offset;
                        break;
                    default:
                        printf("ERROR: Invalid BADDR number!  TERMINATING program!\n");
                        return;
                } // switch (ba_no)

                // -- printf("\nBase Address %i with Offset 0x%lx was entered.\n", ba_no,
offset);
                printf("\nEnter data (8 Bits) to write: 0x ");
                scanf("%x",&dataval_b);
                *pBAOFFb = dataval_b;
                printf("Data = 0x%x\n", *pBAOFFb);

                break;
        } // switch (transfer_size)

    } // (else Num_of_180xhcs)

}//  ***** end of Main

// ********* GET_BOARD_INFO **********

int get_board_info(void)
{
...
```

```
} // (void get_board_info)
```

Appendix C: Complete Example Code

This appendix contains the complete code from the examples in sections 2.1.9 (A/D), 2.2.9 (D/A), 2.3.3 (Counter/Timer). There are two groups of sources, those for the A/D and D/A. There are three source code files for the A/D and two for the D/A. For some source listings, the routine "get_board_info(void)" contents are replaced with '...' to reduce redundancy in the document because each source contains the same unaltered routine.

A/D example sources

1. 1801_ad1.cpp
   Locates an 1801hc board, reads its configuration, configures the 1801hc to perform one reading on channel 0 using a virtual write as the sample clock. It completes by displaying the reading.

```
/*****************************************************************************/
/*                                                                           */
/*  COMPILER AND VERSION                                                     */
/*  --------------------                                                     */
/*  Watcom IDE for C/C++, Version 11.0                                       */
/*                                                                           */
/*  The compiler/IDE was installed on Windows 95 and the code was compiled   */
/*  for the 32-bit DOS environment to be used with the "DOS4GW.EXE" DOS      */
/*  extender that was included with the compiler.                            */
/*                                                                           */
/*  The source files used in compiling the test code are as follows:         */
/*                                                                           */
/*   [.cpp]                                                                  */
/*   amcc.cpp                                                                */
/*   1801_ad1.cpp                                                             */
/*                                                                           */
/*   [.h]                                                                    */
/*   amcc.h                                                                  */
/*   180xhc.h                                                               */
/*   -other Watcom supplied header files not shown here.                     */
/*                                                                           */
/*   The compiled EXE file requires the file "dos4gw.exe" to be located in   */
/*   it's same directory.                                                    */
/*                                                                           */
/*   Note: Be careful to install the Watcom Compiler/IDE and code direct-    */
/*         ories using only 8 character folder names or else the IDE         */
/*         Editor won't find your files.                                     */
/*                                                                           */
/*****************************************************************************/
/*
**  File:   1801_ad1.cpp
**  By:     T.Smith
**  Date:   8/13/1999
*/

// The following header files are included with the Watcom Compiler.
#include <stdio.h>
#include <time.h>
#include "amcc.h"
#include "180xhc.h"


// A copy of the AMCC.H "pointer" typedef has been included below.

        pointer Baddr0[4];    // Memory mapped base address to access AMCC
                              // S5933 operation registers.  Up to a maximum
                              // of 4 KPCI-180xhcs.

        pointer Baddr1[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr2[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr3[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        volatile dword *pAD_DATA[4];
        volatile char *pAD_CHCNT[4];
        volatile char *pTRIG_SOURCE[4];
        volatile char *pDIG_TRIG[4];
        volatile char *pTRIG_CNT[4];
        volatile char *pAD_CONTROL[4];
        volatile char *pAD_SCK_CONFIG[4];
        volatile char *pAD_BURST_VAL[4];
        volatile word *pAD_QRAM[4];
        volatile char *p8254_CNTRL0[4];
        volatile char *p8254_CNTRL1[4];
        volatile char *p8254_CNTRL2[4];
        volatile char *p8254_FORMAT[4];
        volatile char *p8254_CNTR2_INITVAL[4];
        volatile char *p8254_CNTR1_INITVAL[4];
        volatile char *p8254_CNTR0_INITVAL[4];

// THE FOLLOWING PROTOTYPES ARE FOR FUNCTIONS HELD IN THE FILE "AMCC.CPP".
//  THESE FUNCTIONS ARE SUPPLIED BY AMCC.

extern int pci_bios_present(byte *hardware_mechanism,
                            word *interface_level_version,
                            byte *last_pci_bus_number);

extern int find_pci_device(word device_id, word vendor_id,
                            word index,
                            byte *bus_number,
                            byte *device_and_function);

extern word read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read);

int get_board_info(void);
dword ad_virtual_read(int board_no);
void config_8254(int board_no);

//********************************** MAIN ************************************

void  main(void)
{
    int board, Num_of_180xhcs;
    dword Data_Value;


    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {

        printf("Using Board 0 for this test. \n");
        board = 0;

        printf("\nTHIS ROUTINE WILL SETUP & READ ONE READING FROM A/D CHANNEL0.\n");
        printf("A Virtual Write will be used as the Sample Clock.\n");

        // pointers to the A/D registers
        pAD_DATA[board] = (Baddr1[board].dp + 0x00);
        pAD_CHCNT[board] = (Baddr1[board].bp + 0x08);

        pTRIG_SOURCE[board] = (Baddr1[board].bp + 0x10);
        pDIG_TRIG[board] = (Baddr1[board].bp + 0x11);
        pTRIG_CNT[board] = (Baddr1[board].bp + 0x14);
        pAD_CONTROL[board] = (Baddr1[board].bp + 0x04);
        pAD_SCK_CONFIG[board] = (Baddr1[board].bp + 0x05);
        pAD_BURST_VAL[board] = (Baddr1[board].bp + 0x06);
        pAD_QRAM[board] = (Baddr1[board].wp + 0x2000);

        // pointers to the 8254 registers
        p8254_CNTRL0[board] = (Baddr3[board].bp + 0x10);
        p8254_CNTRL1[board] = (Baddr3[board].bp + 0x11);
        p8254_CNTRL2[board] = (Baddr3[board].bp + 0x12);
        p8254_FORMAT[board] = (Baddr3[board].bp + 0x0C);
        p8254_CNTR2_INITVAL[board] = (Baddr3[board].bp + 0x08);
        p8254_CNTR1_INITVAL[board] = (Baddr3[board].bp + 0x04);
        p8254_CNTR0_INITVAL[board] = (Baddr3[board].bp + 0x00);


//        printf("BADDR1 Address is 0x%lx. \n", Baddr1[board].dw);
//        printf("A/D CALRAM is located at 0x%lx. \n", pAD_CALRAM[board]);
//        printf("A/D QRAM is located at 0x%lx. \n", pAD_QRAM[board]);

        // Initializing trigger register for no triggers
        *pDIG_TRIG[board] = 0x00;

        // routine for configuring 8254
        config_8254(board);

        Data_Value = ad_virtual_read(board);

        printf("A/D reading = 0x%lx.\n", Data_Value);

    } // (else Num_of_180xhcs)

}
//********************************** END of MAIN ***********************************

// ****************************** FUNCTIONS & ROUTINES ********************************

// *************** GET_BOARD_INFO *****************
// checks for a PCI BIOS, any 1801/2hc boards on the bus
// if board(s) are found extracts the configuration of
// the board and stores the information in volatile memory.
// ************************************************

int get_board_info(void)
{
        // Here are the variables for Step 1

        byte hmech;
        word ilevel;
        byte lastbus;

        // Here are the variables for Step2

        word did;   // Device ID (did) => KPCI 1801/2HC
        word vid;   // Vendor ID (vid) => Keithley Instruments
        word index=0;
        int UUT_search;
        byte Number_of_UUTs;
        byte bus[4];
        byte devfunc[4];

        // Here are the variables for Step 3

        int board;
        int baddr_seek;


        printf ( "STARTING NOW \n");

        //*****************  STEP  1 *****************
        //
        //  CHECK FOR THE PCI BIOS.
        //
        //**********************************************

        if(pci_bios_present(&hmech,&ilevel,&lastbus)==SUCCESSFUL)
        {
            printf("PCI Bios Found.\n");
        }
        else
        {
            printf(" PCI Bios Not Found.\n");
            return 0;
        }

        //*********************** STEP  2 ******************************
        //
        // VERIFY THE NUMBER OF KPCI-180xhcs WITH THE CORRECT VID / DID.
        //
        //**************************************************************

        // Search the PCI bus for KPCI-1801/2hc boards.
        // Each pass through the loop will identify one UUT.

        did = DID_1801;        // Device ID is for the KPCI 1801HC
        //did = DID_1802;        // Device ID is for the KPCI 1802HC
        vid = VID_KEITHLEY;     // Vendor ID is for Keithley Instruments

        // Look for the first occurrence of a KPCI-1801/2hc on the PCI bus.
        index = 0;
        UUT_search = find_pci_device(did,
                                     vid,
                                     index,
                                     &bus[index],
                                     &devfunc[index]);
        // If one was found then look for others.
        while ( UUT_search == SUCCESSFUL)
        {
            index++;
            UUT_search = find_pci_device(did,
                                         vid,
                                         index,
                                         &bus[index],
                                         &devfunc[index]);

        } // while (find_pci_device(did, vid, index, &bus[index] .....


        // If none were found then send a message and exit.
        if(index == 0)
        {
            printf("No UUTs found which match KPCI-%x's VID and/or DID! \n, did");
            return 0;
        } // if(index == 0)

        Number_of_UUTs = (unsigned char) index;


        //****************************** STEP  3 ************************************
        //
        //  READ THE PCI CONFIG SPACE OF EACH KPCI-1801/2hc AND SAVE THEIR UNIQUE "BADDR0",
        //  "BADDR1", "BADDR2", and BADDR3".  "BADDR" STANDS FOR BASE ADDRESS.
        //  FOUR BADDRs GET ASSIGNED FOR EACH KPCI-1801/2hc BY THE PLUG'N PLAY BIOS DURING
STARTUP.
        //
        //***************************************************************************

        for (board=0; board < Number_of_UUTs ;board++)
        {

            // Find each boards base address (32 bit, BADDR0) for reading / writing
            //  to the AMCC S5933 operation registers.

            baddr_seek =  read_configuration_dword(bus[board],
                                                devfunc[board],
                                                BADDR_0_CFIG_SPACE_OFFSET,
                                                &(Baddr0[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr0 of board %d not found!",board);
            }

            // Now get (32 bit) BADDR1 for AMCC S5933 pass through reads and writes.  This
            //  is the base address used for reading and writing A/D register data from the
1801/2hc.

            baddr_seek =  read_configuration_dword(bus[board],
                                                devfunc[board],
                                                BADDR_1_CFIG_SPACE_OFFSET,
                                                &(Baddr1[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr1 of board %d not found!\n",board);
            }

            // Now get (32 bit) BADDR2 for AMCC S5933 pass through reads and writes.  This
            //  is the base address used for reading and writing D/A register data from the
1801/2hc.

            baddr_seek =  read_configuration_dword(bus[board],
                                                devfunc[board],
                                                BADDR_2_CFIG_SPACE_OFFSET,
                                                &(Baddr2[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr2 of board %d not found!\n",board);
            }

            //  Now get (32 bit) BADDR3 for AMCC S5933 pass through reads and writes.  This
            //  is the base address used for reading and writing timer/interrupt/digital io
register
            //  data from the 1801/2hc.

            baddr_seek =  read_configuration_dword(bus[board],
                                                devfunc[board],
                                                BADDR_3_CFIG_SPACE_OFFSET,
                                                &(Baddr3[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr3 of board %d not found!\n",board);
            }

        } // end of for (board=0; board<index ;board++)

        return index;
} // (int get_board_info)


// *************** CONFIG_8254 *****************
// Configuring each counter of the 8254.
// -- All Counter Gates enabled; allowing continuous train.
// -- Counter0 -> (source) 10MHz % by 100 (count) = 100kHz
// -- Counter1 -> (source) 5MHz % by 10 (count) = 500kHz
// -- Counter2 -> (source) Counter 1 Output % by 5 (count) = 100kHz

void Config_8254(int board_no)
{
    // select sources and gating for 8254 counters
    *p8254_CNTRL0[board_no] = 0x00;
    *p8254_CNTRL1[board_no] = 0x01;
    *p8254_CNTRL2[board_no] = 0x05;
    // format 8254 counters
    *p8254_FORMAT[board_no] = 0x14;
    *p8254_CNTR0_INITVAL[board_no] = 0x64;
    *p8254_FORMAT[board_no] = 0x54;
    *p8254_CNTR1_INITVAL[board_no] = 0x0A;
    *p8254_FORMAT[board_no] = 0x94;
    *p8254_CNTR2_INITVAL[board_no] = 0x05;

    // printf("Finished configuring 8254's counters.\n");

} // (void 8254_config)
// ************************************************


// *************** AD_VIRTUAL_READ *****************
// Setup A/D to take one reading.
// Using a SW Write for the pacer clock.
// The SW Write is also referred to as the Virtual Read.

dword ad_virtual_read(int board_no)
{
    #define DATA_MASK   0x0000FFFF
    #define FIFO_NEMPTY 0x20
    dword rtn_data;

    // Configuring for only one channel.
    *pAD_CHCNT[board_no] = 0x00;

    // Channel 0, Differential, Bipolar, Gain of 1
    *pAD_QRAM[board_no] = 0x0000;

    // Paced mode with SW Write/Virtual Write for pacer clock source
    *pAD_SCK_CONFIG[board_no] = 0x04;

    // Offset binary output, (Pass-Thru)Target mode, A/D Fifo enabled
    *pAD_CONTROL[board_no] = 0x02;

    // Triggering a conversion/sample clock tick
    *pAD_DATA[board_no] = 0x00;

    // wait for A/D fifo to be not empty
    // this will signify that there is data available in the FIFO
    while ( !(*pAD_CONTROL[board_no] & FIFO_NEMPTY) );

    // read data from A/D Data Register
    rtn_data = *pAD_DATA[board_no];
    rtn_data &= DATA_MASK;

    // clearing and disabling A/D FIFO
    *pAD_CONTROL[board_no] = 0x00;

    return rtn_data;
} // (int ad_virtual_read)
// ************************************************
```

2. 1801_ad2.cpp
   Locates an 1801hc board, reads its configuration, configures the 1801hc to perform 50 readings on channel 0 in Pass Thru mode using 8254's Counter2 Output as the sample clock at 100kHz. It stores all of the readings in the text file ‘1801rdgs.txt '.

```
/*****************************************************************************/
```

```
/*                                                                             */
/*  COMPILER AND VERSION                                                       */
/*  --------------------                                                       */
/*  Watcom IDE for C/C++, Version 11.0                                         */
/*                                                                             */
/*  The compiler/IDE was installed on Windows 95 and the code was compiled    */
/*  for the 32-bit DOS environment to be used with the "DOS4GW.EXE" DOS       */
/*  extender that was included with the compiler.                              */
/*                                                                             */
/*  The source files used in compiling the test code are as follows:          */
/*                                                                             */
/*   [.cpp]                                                                    */
/*   amcc.cpp                                                                  */
/*   1801_ad2.cpp                                                               */
/*                                                                             */
/*   [.h]                                                                      */
/*   amcc.h                                                                    */
/*   180xhc.h                                                                 */
/*   -other Watcom supplied header files not shown here.                      */
/*                                                                             */
/*   The compiled EXE file requires the file "dos4gw.exe" to be located in    */
/*   it's same directory.                                                      */
/*                                                                             */
/*   Note: Be careful to install the Watcom Compiler/IDE and code direct-     */
/*         ories using only 8 character folder names or else the IDE          */
/*         Editor won't find your files.                                       */
/*                                                                             */
/*****************************************************************************/
/*
**  File:   1801_ad2.cpp
**  By:     T.Smith
**  Date:   8/13/1999
*/

// The following header files are included with the Watcom Compiler.
#include <stdio.h>
#include <time.h>
#include "amcc.h"
#include "180xhc.h"


// A copy of the AMCC.H "pointer" typedef has been included below.

        pointer Baddr0[4];    // Memory mapped base address to access AMCC
                              // S5933 operation registers.  Up to a maximum
                              // of 4 KPCI-180xhcs.

        pointer Baddr1[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr2[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr3[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        volatile dword *pAD_DATA[4];
        volatile char *pAD_CHCNT[4];
        volatile char *pTRIG_SOURCE[4];
        volatile char *pDIG_TRIG[4];
        volatile char *pTRIG_CNT[4];
        volatile char *pAD_CONTROL[4];
        volatile char *pAD_SCK_CONFIG[4];
        volatile char *pAD_BURST_VAL[4];
        volatile word *pAD_QRAM[4];
        volatile char *p8254_CNTRL0[4];
        volatile char *p8254_CNTRL1[4];
        volatile char *p8254_CNTRL2[4];
        volatile char *p8254_FORMAT[4];
        volatile char *p8254_CNTR2_INITVAL[4];
        volatile char *p8254_CNTR1_INITVAL[4];
        volatile char *p8254_CNTR0_INITVAL[4];

        volatile dword data_array[100];

// THE FOLLOWING PROTOTYPES ARE FOR FUNCTIONS HELD IN THE FILE "AMCC.CPP".
//  THESE FUNCTIONS ARE SUPPLIED BY AMCC.

extern int pci_bios_present(byte *hardware_mechanism,
                            word *interface_level_version,
                            byte *last_pci_bus_number);

extern int find_pci_device(word device_id, word vendor_id,
                            word index,
                            byte *bus_number,
                            byte *device_and_function);

extern word read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read);

int get_board_info(void);
void ad_multi_ptread(int board_no, int cnt);
void config_8254(int board_no);

//********************************** MAIN ************************************

void  main(void)
{
    int board, Num_of_180xhcs, no_reads;
    int count = 0;

    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {

        printf("Using Board 0 for this test. \n");
        board = 0;
        printf("\nTHIS ROUTINE WILL SETUP & READ 50 READINGS FROM A/D CHANNEL0 IN PASS
THRU.\n");
        printf("Counter Output 2 used as Sample Clock with a frequency of 100kHz.\n");
        no_reads = 50;

        // pointers to the A/D registers
        pAD_DATA[board] = (Baddr1[board].dp + 0x00);
        pAD_CHCNT[board] = (Baddr1[board].bp + 0x08);

        pTRIG_SOURCE[board] = (Baddr1[board].bp + 0x10);
        pDIG_TRIG[board] = (Baddr1[board].bp + 0x11);
        pTRIG_CNT[board] = (Baddr1[board].bp + 0x14);
        pAD_CONTROL[board] = (Baddr1[board].bp + 0x04);
        pAD_SCK_CONFIG[board] = (Baddr1[board].bp + 0x05);
        pAD_BURST_VAL[board] = (Baddr1[board].bp + 0x06);
        pAD_QRAM[board] = (Baddr1[board].wp + 0x2000);

        // pointers to the 8254 registers
        p8254_CNTRL0[board] = (Baddr3[board].bp + 0x10);
        p8254_CNTRL1[board] = (Baddr3[board].bp + 0x11);
        p8254_CNTRL2[board] = (Baddr3[board].bp + 0x12);
        p8254_FORMAT[board] = (Baddr3[board].bp + 0x0C);
        p8254_CNTR2_INITVAL[board] = (Baddr3[board].bp + 0x08);
        p8254_CNTR1_INITVAL[board] = (Baddr3[board].bp + 0x04);
        p8254_CNTR0_INITVAL[board] = (Baddr3[board].bp + 0x00);
        // Initializing trigger register for no triggers
        *pDIG_TRIG[board] = 0x00;

        // routine for configuring 8254
        config_8254(board);

        // routine for taking multiple readings
        ad_multi_ptread(board, no_reads);

        printf("Storing data into file: 1801rdgs.txt !\n");

        // Storing data into file
        FILE *fp = fopen("1801rdgs.txt", "w");
        if (fp != NULL)
        {
            fprintf(fp, "1801rdgs.txt\n");
            fprintf(fp, "\nOutput from A/D's passthru.exe (a DOS32 executible). \n");
            fprintf(fp, "Taking 50 readings at 100kHz.\n\n");

            for (; count < no_reads; count++)
            {
                fprintf(fp, "A/D reading %i= 0x%lx.\n", count, data_array[count]);
//                printf("A/D reading %i= 0x%lx.\n", count, data_array[count]);
            } // for

            fprintf(fp, "------------------------------------------------------- \n");
            fclose(fp);
        } // if (fp != NULL)

    } // (else Num_of_180xhcs)

}
//********************** END of MAIN ************************

/* *************** FUNCTIONS & ROUTINES *******************
**
**  (1) int get_board_info(void)
**  (2) void Config_8254(int board_no)
**  (3) void ad_multi_ptread(int board_no, int cnt)
**
**************************************************************/

// ************** GET_BOARD_INFO ****************
// checks for a PCI BIOS, any 1801/2HC boards on the bus
// if board(s) are found extracts the configuration of
// the board and stores the information in volatile memory.
// ***********************************************

int get_board_info(void)
{
        ...
} // (int get_board_info)


// ************** CONFIG_8254 ****************
// Configuring each counter of the 8254.
// -- All Counter Gates enabled; allowing continuous train.
// -- Counter0 -> (source) 10MHz % by 100 (count) = 100kHz
// -- Counter1 -> (source) 5MHz % by 10 (count) = 500kHz
// -- Counter2 -> (source) Counter 1 Output % by 5 (count) = 100kHz

void Config_8254(int board_no)
{
    // select sources and gating for 8254 counters
    *p8254_CNTRL0[board_no] = 0x00;
    *p8254_CNTRL1[board_no] = 0x01;
    *p8254_CNTRL2[board_no] = 0x05;

    // format 8254 counters
    *p8254_FORMAT[board_no] = 0x14;
    *p8254_CNTR0_INITVAL[board_no] = 0x64;
    *p8254_FORMAT[board_no] = 0x54;
    *p8254_CNTR1_INITVAL[board_no] = 0x0A;
    *p8254_FORMAT[board_no] = 0x94;
    *p8254_CNTR2_INITVAL[board_no] = 0x05;

//    printf("Finished configuring 8254's counters.\n");

} // (void 8254_config)
// ************************************************


// *************** AD_MULTI_PTREAD *****************
// Setup A/D to take 50 readings.
// Using the 8254 counter2's output for the pacer clock.
// In pass through mode.

void ad_multi_ptread(int board_no, int cnt)
{
    #define DATA_MASK   0x0000FFFF
    #define FIFO_NEMPTY 0x20
    #define FIFO_HALFFULL 0x40
    int i = 0;

    // Configuring for only one channel.
    *pAD_CHCNT[board_no] = 0x00;

    // Channel 0, Differential, Bipolar, Gain of 1
    *pAD_QRAM[board_no] = 0x0000;

    // Paced mode with 8254 Counter2's Output for pacer clock source
    *pAD_SCK_CONFIG[board_no] = 0x00;

    // Offset binary output, (Pass-Thru)Target mode, A/D Fifo enabled
    *pAD_CONTROL[board_no] = 0x02;

    // wait for A/D fifo to be half full before reading FIFO
    while ( !(*pAD_CONTROL[board_no] & FIFO_HALFFULL) );

    // start reading data from A/D Data Register
    for (; i< cnt; i++)
    {
        data_array[i] = *pAD_DATA[board_no];
        data_array[i] &= DATA_MASK;
    }// for

    // clearing and disabling A/D FIFO
    *pAD_CONTROL[board_no] = 0x00;

} // (ad_multi_ptread(int board_no, int cnt)
// ************************************************
```

Output file: 1801rdgs.txt with a sine wave on Channel 0.

Output from A/D's passthru.exe (a DOS32 executible).

```
Taking 50 readings at 100kHz.

A/D reading 0= 0x87f0.
A/D reading 1= 0x7e60.
A/D reading 2= 0x74d0.
A/D reading 3= 0x6b40.
A/D reading 4= 0x61e0.
A/D reading 5= 0x58a0.

A/D reading 6= 0x4f90.
A/D reading 7= 0x46e0.
A/D reading 8= 0x3e60.
A/D reading 9= 0x3660.
A/D reading 10= 0x2ed0.
A/D reading 11= 0x27a0.
A/D reading 12= 0x2110.
A/D reading 13= 0x1b00.
A/D reading 14= 0x1590.
A/D reading 15= 0x10d0.
A/D reading 16= 0xcb0.
A/D reading 17= 0x8f0.
A/D reading 18= 0x610.
A/D reading 19= 0x3f0.
A/D reading 20= 0x270.
A/D reading 21= 0x1b0.
A/D reading 22= 0x1b0.
A/D reading 23= 0x270.
A/D reading 24= 0x3e0.
A/D reading 25= 0x600.
A/D reading 26= 0x8d0.
A/D reading 27= 0xc40.
A/D reading 28= 0x1080.
A/D reading 29= 0x1530.
A/D reading 30= 0x1a90.
A/D reading 31= 0x2090.
A/D reading 32= 0x2730.
A/D reading 33= 0x2e40.
A/D reading 34= 0x35a0.
A/D reading 35= 0x3da0.
A/D reading 36= 0x4620.
A/D reading 37= 0x4ed0.
A/D reading 38= 0x57d0.
A/D reading 39= 0x6110.
A/D reading 40= 0x6a80.
A/D reading 41= 0x7410.
A/D reading 42= 0x7d80.
A/D reading 43= 0x8720.
A/D reading 44= 0x90c0.
A/D reading 45= 0x9a20.
A/D reading 46= 0xa390.
A/D reading 47= 0xacc0.
A/D reading 48= 0xb590.
A/D reading 49= 0xbe40.
--------------------------------------------------------
```

3. 1801_ad3.cpp
   Locates an 1801hc board, reads its configuration, configures the 1801hc to perform 50 readings on channel 0 in Bus Mastering mode using 8254's Counter2 Output as the sample clock at 100kHz. It stores all of the readings in the text file '1801bm.txt'.

```
/*****************************************************************************/
```

```
/*                                                                    */
```

```
/*  COMPILER AND VERSION                                                    */
/*  --------------------                                                    */
/*  Watcom IDE for C/C++, Version 11.0                                      */
/*                                                                          */
/*  The compiler/IDE was installed on Windows 95 and the code was compiled  */
/*  for the 32-bit DOS environment to be used with the "DOS4GW.EXE" DOS     */
/*  extender that was included with the compiler.                           */
/*                                                                          */
/*  The source files used in compiling the test code are as follows:        */
/*                                                                          */
/*   [.cpp]                                                                 */
/*   amcc.cpp                                                               */
/*   1801_ad3.cpp                                                            */
/*                                                                          */
/*   [.h]                                                                   */
/*   amcc.h                                                                 */
/*   180xhc.h                                                              */
/*   -other Watcom supplied header files not shown here.                    */
/*                                                                          */
/*   The compiled EXE file requires the file "dos4gw.exe" to be located in  */
/*   it's same directory.                                                   */
/*                                                                          */
/*   Note: Be careful to install the Watcom Compiler/IDE and code direct-   */
/*         ories using only 8 character folder names or else the IDE        */
/*         Editor won't find your files.                                    */
/*                                                                          */
/****************************************************************************/
/*
**  File:   1801_ad3.cpp
**  By:     T.Smith
**  Date:   8/13/1999
*/

// The following header files are included with the Watcom Compiler.
#include <stdio.h>
#include <time.h>
#include "amcc.h"
#include "180xhc.h"


// A copy of the AMCC.H "pointer" typedef has been included below.

        pointer Baddr0[4];    // Memory mapped base address to access AMCC
                              // S5933 operation registers.  Up to a maximum
                              // of 4 KPCI-180xhcs.

        pointer Baddr1[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr2[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr3[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        volatile dword *pAD_DATA[4];
        volatile char *pAD_CHCNT[4];
        volatile char *pTRIG_SOURCE[4];
        volatile char *pDIG_TRIG[4];
        volatile char *pTRIG_CNT[4];
        volatile char *pAD_CONTROL[4];
        volatile char *pAD_SCK_CONFIG[4];
        volatile char *pAD_BURST_VAL[4];
        volatile word *pAD_QRAM[4];
        volatile char *p8254_CNTRL0[4];
        volatile char *p8254_CNTRL1[4];
        volatile char *p8254_CNTRL2[4];
        volatile char *p8254_FORMAT[4];
        volatile char *p8254_CNTR2_INITVAL[4];
        volatile char *p8254_CNTR1_INITVAL[4];
        volatile char *p8254_CNTR0_INITVAL[4];

    // ******** Bus Master definitions ********

        // -- FIFO Flag Reset
        #define RESET_A2P_FLAGS     0x04000000L
        #define RESET_P2A_FLAGS     0x02000000L
        // -- FIFO priority
        #define A2P_HI_PRIORITY     0x00000100L
        #define P2A_HI_PRIORITY     0x00001000L
        // -- Enable Transfer Count
        #define EN_TCOUNT           0x10000000L
        // -- Enable Bus Mastering
        #define EN_A2P_TRANSFERS    0x00000400L
        #define EN_P2A_TRANSFERS    0x00004000L

    // ********

        dword data_array[100];

// THE FOLLOWING PROTOTYPES ARE FOR FUNCTIONS HELD IN THE FILE "AMCC.CPP".
//  THESE FUNCTIONS ARE SUPPLIED BY AMCC.

extern int pci_bios_present(byte *hardware_mechanism,
                            word *interface_level_version,
                            byte *last_pci_bus_number);

extern int find_pci_device(word device_id, word vendor_id,
                            word index,
                            byte *bus_number,
                            byte *device_and_function);

extern word read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read);

int get_board_info(void);
void ad_multi_bmread(int board_no);
void config_8254(int board_no);

//********************************* MAIN ***********************************

void  main(void)
{
    int board, Num_of_180xhcs, no_reads;
    int count = 0;
    word lo_word, hi_word;


    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("Using Board 0 for this test. \n");
        board = 0;

        printf("\nTHIS ROUTINE WILL SETUP & READ 50 READINGS FROM A/D CHANNEL0 IN BUS 
MASTERING.\n");
        printf("Counter Output 2 used as Sample Clock with a frequency of 100kHz.\n");

        // for Bus Mastering mode there is a reading in the MSW and LSW
        // as for Pass Through where there the reading is only in the LSW
        // therefore the number of reads is half.
        no_reads = 25;
        // pointers to the A/D registers
        pAD_DATA[board] = (Baddr1[board].dp + 0x00);
        pAD_CHCNT[board] = (Baddr1[board].bp + 0x08);

        pTRIG_SOURCE[board] = (Baddr1[board].bp + 0x10);
        pDIG_TRIG[board] = (Baddr1[board].bp + 0x11);
        pTRIG_CNT[board] = (Baddr1[board].bp + 0x14);
        pAD_CONTROL[board] = (Baddr1[board].bp + 0x04);
        pAD_SCK_CONFIG[board] = (Baddr1[board].bp + 0x05);
        pAD_BURST_VAL[board] = (Baddr1[board].bp + 0x06);
        pAD_QRAM[board] = (Baddr1[board].wp + 0x2000);

        // pointers to the 8254 registers
        p8254_CNTRL0[board] = (Baddr3[board].bp + 0x10);
        p8254_CNTRL1[board] = (Baddr3[board].bp + 0x11);
        p8254_CNTRL2[board] = (Baddr3[board].bp + 0x12);
        p8254_FORMAT[board] = (Baddr3[board].bp + 0x0C);
        p8254_CNTR2_INITVAL[board] = (Baddr3[board].bp + 0x08);
        p8254_CNTR1_INITVAL[board] = (Baddr3[board].bp + 0x04);
        p8254_CNTR0_INITVAL[board] = (Baddr3[board].bp + 0x00);


        // Initializing trigger register for no triggers
        *pDIG_TRIG[board] = 0x00;

        // routine for configuring 8254
        config_8254(board);

        // routine for taking multiple readings
        ad_multi_bmread(board);

        // clearing and disabling A/D FIFO
        *pAD_CONTROL[board] = 0x00;


        printf("Storing data into file: 1801bm.txt.\n");

        // Storing data into file
        FILE *fp = fopen("1801bm.txt", "w");
        if (fp != NULL)
        {
            fprintf(fp, "1801bm.txt\n");
            fprintf(fp, "\nOutput from A/D's busmastr.exe (a DOS32 executible). \n");
            fprintf(fp, "Taking 50 readings at 100kHz (Bus Mastering mode).\n");
            fprintf(fp, "Each index contains two readings. One in the LO word and the other
in the HI word.\n\n");

            for (; count < no_reads; count++)
            {
                lo_word = (word)(data_array[count] & 0x0000FFFF);
                hi_word = (word)((data_array[count] & 0xFFFF0000) >> 16);
                fprintf(fp, "A/D reading index %i, hi = 0x%x, lo = 0x%x.\n", count, hi_word,
lo_word);
            } // for

            fprintf(fp, "------------------------------------- \n");
            fclose(fp);
        } // if (fp != NULL)

        printf("Finished.\n");
    } // (else Num_of_180xhcs)

}
//********************************** END of MAIN ***********************************

// *************************** FUNCTIONS & ROUTINES *******************************

// ************** GET_BOARD_INFO ****************
// checks for a PCI BIOS, any 1801/2HC boards on the bus
// if board(s) are found extracts the configuration of
// the board and stores the information in volatile memory.
// ***********************************************

int get_board_info(void)
{
        ...
} // (int get_board_info)


// *************** CONFIG_8254 *****************
// Configuring each counter of the 8254.
// -- All Counter Gates enabled; allowing continuous train.
// -- Counter0 -> (source) 10MHz % by 100 (count) = 100kHz
// -- Counter1 -> (source) 5MHz % by 10 (count) = 500kHz
// -- Counter2 -> (source) Counter 1 Output % by 5 (count) = 100kHz

void Config_8254(int board_no)
{
    // select sources and gating for 8254 counters
    *p8254_CNTRL0[board_no] = 0x00;
    *p8254_CNTRL1[board_no] = 0x01;
    *p8254_CNTRL2[board_no] = 0x05;

    // format 8254 counters
    *p8254_FORMAT[board_no] = 0x14;
    *p8254_CNTR0_INITVAL[board_no] = 0x64;
    *p8254_FORMAT[board_no] = 0x54;
    *p8254_CNTR1_INITVAL[board_no] = 0x0A;
    *p8254_FORMAT[board_no] = 0x94;
    *p8254_CNTR2_INITVAL[board_no] = 0x05;

//    printf("Finished configuring 8254's counters.\n");

} // (void 8254_config)
// ***********************************************


// *************** AD_MULTI_BMREAD *****************
// Setup A/D to take 50 readings.
// Using the 8254 counter2's output for the pacer clock.
// In bus master mode.

void ad_multi_bmread(int board_no)
{
    int i=0, j=0;
    dword temp = 0;
    dword temp2 = 0;
    volatile dword *pmwtc;
    volatile dword *pmwar;
    volatile dword *pmrar;
    volatile dword *pmrtc;
    volatile dword *pmcsr;

    // Configuring for only one channel.
    *pAD_CHCNT[board_no] = 0x00;

    // Channel 0, Differential, Bipolar, Gain of 1
    *pAD_QRAM[board_no] = 0x0000;

    // Paced mode with 8254 Counter2's Output for pacer clock source
    *pAD_SCK_CONFIG[board_no] = 0x00;

    // setup BusMastering registers

        pmcsr = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MCSR);
            // bm read and write addresses
        pmwar = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MWAR);
        pmrar = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MRAR);
            // bm read and write transfer count addresses
        pmwtc = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MWTC);
        pmrtc = (volatile dword *)(Baddr0[board_no].bp + AMCC_OP_REG_MRTC);
        *pmwar = (volatile dword)(&data_array[0]);
        *pmrar = (volatile dword)(&data_array[0]);

            // number of bytes to transfer 100 = 4x25
        *pmwtc = 100;
        *pmrtc = 100;

        // enable bus mastering and transfer count
        temp2 = *pmcsr;
        *pmcsr =
temp2|RESET_A2P_FLAGS|RESET_P2A_FLAGS|A2P_HI_PRIORITY|P2A_HI_PRIORITY|EN_A2P_TRANSFERS|EN_P2A
_TRANSFERS;

    // Offset binary output, (Bus Master)FIFO mode, A/D Fifo enabled
    *pAD_CONTROL[board_no] = 0x03;

//    printf("pmwtc = %lx\n", pmwtc);
//    printf("transfer count (*pmwtc) = %d\n", *pmwtc);

    // waiting for the transfer count to reach 0
    // when this is 0 all the data has been taken
    while(*pmwtc != 0)
    {
        if(temp == *pmwtc)
        {
            *pAD_CONTROL[board_no] = 0x00; // disable FIFO
            printf("ERROR: Transfer count not changing!\n");
            return;
        }
        printf("transfer count = %d\n", *pmwtc);
        temp = *pmwtc;
    } // for i


} // (ad_multi_bmread(int board_no)
// ************************************************
```

Output File: 1801bm.txt with a sine wave on Channel 0.

```
Output from A/D's busmastr.exe (a DOS32 executible).
Taking 50 readings at 100kHz (Bus Mastering mode).
Each index contains two readings. One in the LO word and the other in the HI word.

A/D reading index 0, hi = 0x6920, lo = 0x5fd0.
A/D reading index 1, hi = 0x7c30, lo = 0x72c0.
A/D reading index 2, hi = 0x8f70, lo = 0x85f0.
A/D reading index 3, hi = 0xa240, lo = 0x98f0.
A/D reading index 4, hi = 0xb480, lo = 0xab70.
A/D reading index 5, hi = 0xc560, lo = 0xbd20.
A/D reading index 6, hi = 0xd4d0, lo = 0xcd50.
A/D reading index 7, hi = 0xe260, lo = 0xdbd0.
A/D reading index 8, hi = 0xffe0, lo = 0xfff0.
A/D reading index 9, hi = 0xfcc0, lo = 0xfeb0.
A/D reading index 10, hi = 0xf6b0, lo = 0xfa10.
A/D reading index 11, hi = 0xee70, lo = 0xf2d0.
A/D reading index 12, hi = 0xe320, lo = 0xe900.
A/D reading index 13, hi = 0xd5e0, lo = 0xdcd0.
A/D reading index 14, hi = 0xc6a0, lo = 0xce80.
A/D reading index 15, hi = 0xb5b0, lo = 0xbe60.
A/D reading index 16, hi = 0xa3b0, lo = 0xacf0.
A/D reading index 17, hi = 0x90c0, lo = 0x9a40.
A/D reading index 18, hi = 0x7da0, lo = 0x8750.
A/D reading index 19, hi = 0x6a90, lo = 0x7420.
A/D reading index 20, hi = 0x5800, lo = 0x6150.
A/D reading index 21, hi = 0x4630, lo = 0x4ef0.
A/D reading index 22, hi = 0x35d0, lo = 0x3dc0.
A/D reading index 23, hi = 0x2740, lo = 0x2e50.
A/D reading index 24, hi = 0x1aa0, lo = 0x20a0.
-------------------------------------
```

D/A example sources

1. 1801_da1.cpp
   Locates an 1801hc board, reads its configuration, asks the user to select a DAC channel and range, and prompts the user to enter a 2's compliment number to write. A virtual write is used as the update clock.

```
/***************************************************************************/
```

```
/*                                                                            */
/*  COMPILER AND VERSION                                                      */
/*  --------------------                                                      */
/*  Watcom IDE for C/C++, Version 11.0                                        */
/*                                                                            */
/*  The compiler/IDE was installed on Windows 95 and the code was compiled   */
/*  for the 32-bit DOS environment to be used with the "DOS4GW.EXE" DOS      */
/*  extender that was included with the compiler.                             */
/*                                                                            */
/*  The source files used in compiling the test code are as follows:         */
/*                                                                            */
/*   [.cpp]                                                                   */
/*   amcc.cpp                                                                 */
/*   1801_da1.cpp                                                              */
/*                                                                            */
/*   [.h]                                                                     */
/*   amcc.h                                                                   */
/*   180xhc.h                                                                */
/*   -other Watcom supplied header files not shown here.                      */
/*                                                                            */
/*   The compiled EXE file requires the file "dos4gw.exe" to be located in   */
/*   it's same directory.                                                     */
/*                                                                            */
/*   Note: Be careful to install the Watcom Compiler/IDE and code direct-    */
/*         ories using only 8 character folder names or else the IDE         */
/*         Editor won't find your files.                                      */
/*                                                                            */
/****************************************************************************/
/*
**  File:   1801_da1.cpp
**  By:     T.Smith
**  Date:   8/13/1999
*/

// The following header files are included with the Watcom Compiler.
#include <stdio.h>
#include <time.h>
#include "amcc.h"
#include "180xhc.h"


// A copy of the AMCC.H "pointer" typedef has been included below.

        pointer Baddr0[4];    // Memory mapped base address to access AMCC
                              // S5933 operation registers.  Up to a maximum
                              // of 4 KPCI-1801/2hcs.

        pointer Baddr1[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-1801/2hcs.

        pointer Baddr2[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-1801/2hcs.

        pointer Baddr3[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-1801/2hcs.

        volatile dword *pDA_DATA[4];
        volatile char *pDA_CHCNT[4];
        volatile char *pDA_CONTROL[4];
        volatile char *pDA_UPCK_CONFIG[4];
        volatile char *pDA_BURST_VAL[4];
        volatile char *pDA_QRAM[4];
        volatile char *pDA_ENABLE[4];

        volatile char *p8254_CNTRL0[4];
        volatile char *p8254_CNTRL1[4];
        volatile char *p8254_CNTRL2[4];
        volatile char *p8254_FORMAT[4];
        volatile char *p8254_CNTR2_INITVAL[4];
        volatile char *p8254_CNTR1_INITVAL[4];
        volatile char *p8254_CNTR0_INITVAL[4];

// THE FOLLOWING PROTOTYPES ARE FOR FUNCTIONS HELD IN THE FILE "AMCC.CPP".
//  THESE FUNCTIONS ARE SUPPLIED BY AMCC.

extern int pci_bios_present(byte *hardware_mechanism,
                            word *interface_level_version,
                            byte *last_pci_bus_number);

extern int find_pci_device(word device_id, word vendor_id,
                            word index,
                            byte *bus_number,
                            byte *device_and_function);

extern word read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read);

int get_board_info(void);
void da_update(int board_no);
void config_8254(int board_no);

//********************************* MAIN ***********************************

void  main(void)
{
    int board, Num_of_180xhcs;


    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("\nTHIS ROUTINE WILL PERFORM AN UPDATE TO A DAC CHANNEL.\n\n");
        printf("Using Board 0 for this test. \n");
        board = 0;

        // pointers to the A/D registers
        pDA_DATA[board] = (Baddr2[board].dp + 0x00);
        pDA_CHCNT[board] = (Baddr2[board].bp + 0x08);

        pDA_CONTROL[board] = (Baddr2[board].bp + 0x04);
        pDA_UPCK_CONFIG[board] = (Baddr2[board].bp + 0x05);
        pDA_BURST_VAL[board] = (Baddr2[board].bp + 0x06);
        pDA_QRAM[board] = (Baddr2[board].bp + 0x80);
        pDA_ENABLE[board] = (Baddr2[board].bp + 0x0C);

        // pointers to the 8254 registers
        p8254_CNTRL0[board] = (Baddr3[board].bp + 0x10);
        p8254_CNTRL1[board] = (Baddr3[board].bp + 0x11);
        p8254_CNTRL2[board] = (Baddr3[board].bp + 0x12);
        p8254_FORMAT[board] = (Baddr3[board].bp + 0x0C);
        p8254_CNTR2_INITVAL[board] = (Baddr3[board].bp + 0x08);
        p8254_CNTR1_INITVAL[board] = (Baddr3[board].bp + 0x04);
        p8254_CNTR0_INITVAL[board] = (Baddr3[board].bp + 0x00);


        // routine for configuring 8254
        config_8254(board);

        // routine for a D/A Update
        da_update(board);

        printf("Update Completed!\n");
        // clearing and disabling D/A FIFO
        *pDA_CONTROL[board] = 0x00;

    } // (else Num_of_180xhcs)

}
//********************************** END of MAIN ***********************************

// *************************** FUNCTIONS & ROUTINES *******************************

// *************** GET_BOARD_INFO *****************
// checks for a PCI BIOS, any 1801/2hc boards on the bus
// if board(s) are found extracts the configuration of
// the board and stores the information in volatile memory.
// ***********************************************

int get_board_info(void)
{
        // Here are the variables for Step 1

        byte hmech;
        word ilevel;
        byte lastbus;

        // Here are the variables for Step2

        word did;   // Device ID (did) => KPCI 1801/2hc
        word vid;   // Vendor ID (vid) => Keithley Instruments
        word index=0;
        int UUT_search;
        byte Number_of_UUTs;
        byte bus[4];
        byte devfunc[4];

        // Here are the variables for Step 3

        int board;
        int baddr_seek;


        printf ( "STARTING NOW \n");

        //******************  STEP  1 *****************
        //
        //  CHECK FOR THE PCI BIOS.
        //
        //***********************************************

        if(pci_bios_present(&hmech,&ilevel,&lastbus)==SUCCESSFUL)
        {
            printf("PCI Bios Found.\n");
        }
        else
        {
            printf(" PCI Bios Not Found.\n");
            return 0;
        }

        //*********************** STEP  2 *****************************
        //
        // VERIFY THE NUMBER OF KPCI-1801/2hcs WITH THE CORRECT VID / DID.
        //
        //************************************************************

        // Search the PCI bus for KPCI-1801/2hc boards.
        // Each pass through the loop will identify one UUT.

        did = DID_1801;        // Device ID is for the KPCI 1801
        //did = DID_1802;        // Device ID is for the KPCI 1802
        vid = VID_KEITHLEY;     // Vendor ID is for Keithley Instruments
        // Look for the first occurrence of a KPCI-1801/2hc on the PCI bus.
        index = 0;
        UUT_search = find_pci_device(did,
                                     vid,
                                     index,
                                     &bus[index],
                                     &devfunc[index]);
        // If one was found then look for others.
        while ( UUT_search == SUCCESSFUL)
        {
            index++;
            UUT_search = find_pci_device(did,
                                         vid,
                                         index,
                                         &bus[index],
                                         &devfunc[index]);

        } // while (find_pci_device(did, vid, index, &bus[index] .....


        // If none were found then send a message and exit.
        if(index == 0)
        {
            printf("No UUTs found which match KPCI-%x's VID and/or DID! \n", did);
            return 0;
        } // if(index == 0)

        Number_of_UUTs = (unsigned char) index;


        //****************************** STEP  3 ************************************
        //
        //  READ THE PCI CONFIG SPACE OF EACH KPCI-1801/2hc AND SAVE THEIR UNIQUE "BADDR0",
        //  "BADDR1", "BADDR2", and BADDR3".  "BADDR" STANDS FOR BASE ADDRESS.
        //  FOUR BADDRs GET ASSIGNED FOR EACH KPCI-1801/2hc BY THE PLUG'N PLAY BIOS DURING
STARTUP.
        //
        //*****************************************************************************

        for (board=0; board < Number_of_UUTs ;board++)
        {

            // Find each boards base address (32 bit, BADDR0) for reading / writing
            //  to the AMCC S5933 operation registers.

            baddr_seek =  read_configuration_dword(bus[board],
                                                    devfunc[board],
                                                    BADDR_0_CFIG_SPACE_OFFSET,
                                                    &(Baddr0[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr0 of board %d not found!",board);
            }

            // Now get (32 bit) BADDR1 for AMCC S5933 pass through reads and writes.  This
            //  is the base address used for reading and writing A/D register data from the
1801/2hc.

            baddr_seek =  read_configuration_dword(bus[board],
                                                    devfunc[board],
                                                    BADDR_1_CFIG_SPACE_OFFSET,
                                                    &(Baddr1[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr1 of board %d not found!\n",board);
            }

            // Now get (32 bit) BADDR2 for AMCC S5933 pass through reads and writes.  This
            //  is the base address used for reading and writing D/A register data from the
1801/2hc.

            baddr_seek =  read_configuration_dword(bus[board],
                                                    devfunc[board],
                                                    BADDR_2_CFIG_SPACE_OFFSET,
                                                    &(Baddr2[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr2 of board %d not found!\n",board);
            }

            //  Now get (32 bit) BADDR3 for AMCC S5933 pass through reads and writes.  This
            //  is the base address used for reading and writing timer/interrupt/digital io
register
            //  data from the 1801/2hc.

            baddr_seek =  read_configuration_dword(bus[board],
                                                    devfunc[board],
                                                    BADDR_3_CFIG_SPACE_OFFSET,
                                                    &(Baddr3[board].dw));
            if (baddr_seek == SUCCESSFUL) ;
            else
            {
                printf("Baddr3 of board %d not found!\n",board);
            }

        } // end of for (board=0; board<index ;board++)

        return index;
} // (int get_board_info)


// *************** CONFIG_8254 *****************
// Configuring each counter of the 8254.
// -- All Counter Gates enabled; allowing continuous train.
// -- Counter0 -> (source) 10MHz % by 100 (count) = 100kHz
// -- Counter1 -> (source) 5MHz % by 10 (count) = 500kHz
// -- Counter2 -> (source) Counter 1 Output % by 5 (count) = 100kHz

void Config_8254(int board_no)
{
    // select sources and gating for 8254 counters
    *p8254_CNTRL0[board_no] = 0x00;
    *p8254_CNTRL1[board_no] = 0x01;
    *p8254_CNTRL2[board_no] = 0x05;

    // format 8254 counters
    *p8254_FORMAT[board_no] = 0x14;
    *p8254_CNTR0_INITVAL[board_no] = 0x64;
    *p8254_FORMAT[board_no] = 0x54;
    *p8254_CNTR1_INITVAL[board_no] = 0x0A;
    *p8254_FORMAT[board_no] = 0x94;
    *p8254_CNTR2_INITVAL[board_no] = 0x05;

    // printf("Finished configuring 8254's counters.\n");

} // (void 8254_config)
// ************************************************


// *************** DA_UPDATE *****************
// Setup D/A to write a value (from user) to either D/A channel.
// Using a SW Write for the update clock.
// The SW Write is also used to write the D/A data to the DAC channel.

void da_update(int board_no)
{
    #define DATA_MASK   0x0000FFFF
    #define CH_MASK     0x01
    byte dac_channel =0;
    byte dac_on;
    dword dac_data =0;

    printf("\nSelect DAC Channel '0' or '1': ");
    scanf("%x", &dac_channel);

    // Configuring for only one channel.
    *pDA_CHCNT[board_no] = 0x00;

    // DAC Channel 0 or 1 (from user input above)
    *pDA_QRAM[board_no] = (dac_channel & CH_MASK);

    // Paced mode with SW Write/Virtual Write for update clock source
    *pDA_UPCK_CONFIG[board_no] = 0x04;

    // Requesting Offset binary Data to Write
    printf("Offset Binary Value to Write: 0x ");
    scanf("%lx", &dac_data);

    // No conversion, (Pass-Thru)Target mode, D/A Fifo enabled
    *pDA_CONTROL[board_no] = 0x02;

    // Activating Board's D/A Output
    // Must keep this set (0x01).
    // If cleared (0x00) then D/A outputs become grounded on board.
    dac_on = 0x00;
    *pDA_ENABLE[board_no] = dac_on;

    // Writing data to update DAC
    // For PassThru only the lower word is used. So masking upper word to zeroes.
    *pDA_DATA[board_no] = (dac_data & DATA_MASK);

    // Debugging printf's...
    //-- printf("Dac range: 0x%x\n", *pDA_RANGE[board_no]);

    //-- printf("\nReading DAC Data from register = 0x%lx\n",*pDA_DATA[board_no]);


} // (void da_update)
// ************************************************
```

2. 1801_da2.cpp
   Locates an 1801hc board, reads its configuration, creates a triangle waveform, asks the user to select a DAC channel and range, and sends the waveform's pattern to the DAC in Pass Thru mode. The 8254's Counter2 Output is used as the update clock at 100Hz. An oscilloscope can be used to observe the waveform from the selected DAC output.

```
/**************************************************************************************/
```

```
/*                                                                          */
/*  COMPILER AND VERSION                                                    */
/*  --------------------                                                    */
/*  Watcom IDE for C/C++, Version 11.0                                      */
/*                                                                          */
/*  The compiler/IDE was installed on Windows 95 and the code was compiled   */
/*  for the 32-bit DOS environment to be used with the "DOS4GW.EXE" DOS      */
/*  extender that was included with the compiler.                           */
/*                                                                          */
/*  The source files used in compiling the test code are as follows:        */
/*                                                                          */
/*   [.cpp]                                                                 */
/*   amcc.cpp                                                               */
/*   1801_da2.cpp                                                            */
/*                                                                          */
/*   [.h]                                                                   */
/*   amcc.h                                                                 */
/*   180xhc.h                                                              */
/*   -other Watcom supplied header files not shown here.                    */
/*                                                                          */
/*   The compiled EXE file requires the file "dos4gw.exe" to be located in   */
/*   it's same directory.                                                   */
/*                                                                          */
/*   Note: Be careful to install the Watcom Compiler/IDE and code direct-    */
/*         ories using only 8 character folder names or else the IDE        */
/*         Editor won't find your files.                                    */
/*                                                                          */
/****************************************************************************/
/*
**  File:   1801_da2.cpp
**  By:     T.Smith
**  Date:   8/16/1999
*/

// The following header files are included with the Watcom Compiler.
#include <stdio.h>
#include <time.h>
#include "amcc.h"
#include "180xhc.h"


// A copy of the AMCC.H "pointer" typedef has been included below.

        pointer Baddr0[4];    // Memory mapped base address to access AMCC
                              // S5933 operation registers.  Up to a maximum
                              // of 4 KPCI-180xhcs.

        pointer Baddr1[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr2[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        pointer Baddr3[4];    // Memory mapped base address to access the
                              // pass thru data area / Keithley hardware.  Up
                              // to a maximum of 4 KPCI-180xhcs.

        volatile dword *pDA_DATA[4];
        volatile char *pDA_CHCNT[4];
        volatile char *pDA_CONTROL[4];
        volatile char *pDA_UPCK_CONFIG[4];
        volatile char *pDA_BURST_VAL[4];
        volatile char *pDA_QRAM[4];
        volatile char *pDA_ENABLE[4];

        volatile char *p8254_CNTRL0[4];
        volatile char *p8254_CNTRL1[4];
        volatile char *p8254_CNTRL2[4];
        volatile char *p8254_FORMAT[4];
        volatile char *p8254_CNTR2_INITVAL[4];
        volatile char *p8254_CNTR1_INITVAL[4];
        volatile char *p8254_CNTR0_INITVAL[4];

// THE FOLLOWING PROTOTYPES ARE FOR FUNCTIONS HELD IN THE FILE "AMCC.CPP".
//  THESE FUNCTIONS ARE SUPPLIED BY AMCC.

extern int pci_bios_present(byte *hardware_mechanism,
                            word *interface_level_version,
                            byte *last_pci_bus_number);

extern int find_pci_device(word device_id, word vendor_id,
                            word index,
                            byte *bus_number,
                            byte *device_and_function);

extern word read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read);

int get_board_info(void);
void da_update(int board_no);
void config_8254(int board_no);
void gen_pattern(void);

word pattern[100];
#define MAX_COUNT   25

//********************************** MAIN ************************************

void  main(void)
{
    int board, Num_of_180xhcs;


    Num_of_180xhcs = get_board_info();

    if (Num_of_180xhcs == 0)
    {
        return;
    }
    else
    {
        printf("\nTHIS ROUTINE WILL CREATE A PATTERN TO SEND TO A DAC CHANNEL IN PASS
THRU.\n");
        printf("Counter Output 2 used as Update Clock with a frequency of 100Hz.\n");
        printf("Using Board 0 for this test. \n");
        board = 0;

        // pointers to the A/D registers
        pDA_DATA[board] = (Baddr2[board].dp + 0x00);
        pDA_CHCNT[board] = (Baddr2[board].bp + 0x08);

        pDA_CONTROL[board] = (Baddr2[board].bp + 0x04);
        pDA_UPCK_CONFIG[board] = (Baddr2[board].bp + 0x05);
        pDA_BURST_VAL[board] = (Baddr2[board].bp + 0x06);
        pDA_QRAM[board] = (Baddr2[board].bp + 0x80);
        pDA_ENABLE[board] = (Baddr2[board].bp + 0x0C);

        // pointers to the 8254 registers
        p8254_CNTRL0[board] = (Baddr3[board].bp + 0x10);
        p8254_CNTRL1[board] = (Baddr3[board].bp + 0x11);
        p8254_CNTRL2[board] = (Baddr3[board].bp + 0x12);
        p8254_FORMAT[board] = (Baddr3[board].bp + 0x0C);
        p8254_CNTR2_INITVAL[board] = (Baddr3[board].bp + 0x08);
        p8254_CNTR1_INITVAL[board] = (Baddr3[board].bp + 0x04);
        p8254_CNTR0_INITVAL[board] = (Baddr3[board].bp + 0x00);


        // routine for configuring 8254
        config_8254(board);

        // creates pattern
        gen_pattern();

        // routine for a D/A Update
        da_update(board);

        printf("Update Completed!\n");

        // clearing and disabling D/A FIFO
        *pDA_CONTROL[board] = 0x00;

    } // (else Num_of_180xhcs)

}
//********************************** END of MAIN ************************************

// **************************** FUNCTIONS & ROUTINES ********************************

// *************** GET_BOARD_INFO *****************
// checks for a PCI BIOS, any 1801/2hc boards on the bus
// if board(s) are found extracts the configuration of
// the board and stores the information in volatile memory.
// ***********************************************

int get_board_info(void)
{
        ...
}
// ************* END OF GET_BOARD_INFO *******************


// *************** CONFIG_8254 *****************
// Configuring each counter of the 8254.
// -- All Counter Gates enabled; allowing continuous train.
// -- Counter0 -> (source) 10MHz % by 100 (count) = 100kHz
// -- Counter1 -> (source) 100kHz % by 100 (count) = 1kHz
// -- Counter2 -> (source) Counter 1 Output % by 10 (count) = 100Hz

void Config_8254(int board_no)
{
    // select sources and gating for 8254 counters
    *p8254_CNTRL0[board_no] = 0x00;
    *p8254_CNTRL1[board_no] = 0x03;
    *p8254_CNTRL2[board_no] = 0x05;

    // format 8254 counters
    *p8254_FORMAT[board_no] = 0x14;
    *p8254_CNTR0_INITVAL[board_no] = 0x64;
    *p8254_FORMAT[board_no] = 0x54;
    *p8254_CNTR1_INITVAL[board_no] = 0x64;
    *p8254_FORMAT[board_no] = 0x94;
    *p8254_CNTR2_INITVAL[board_no] = 0x0A;

    // printf("Finished configuring 8254's counters.\n");

}
// ************* END OF CONFIG_8254 *******************


// **************** GEN_PATTERN ********************
/* generates the top half of a triangular sine wave using 50 points
**       ^
**     /   \      <-- like this
**    /     \
*/

void gen_pattern(void)
{
    int i = 0;
    #define INC_DEC_VAL     0x051E
    #define ISTART_VAL      0x8000
    #define DSTART_VAL      0xF000

    for(; i<MAX_COUNT; i++)
    {
        pattern[i] = ISTART_VAL + (INC_DEC_VAL * i);
    }
    printf("pattern[MAX_COUNT-1] = %x\n", pattern[MAX_COUNT-1]);
    for(i=0; i<MAX_COUNT; i++)
    {
        pattern[i+MAX_COUNT] = DSTART_VAL - (INC_DEC_VAL * i);
    }
    printf("pattern[(MAX_COUNT*2)-1] = %x\n", pattern[(MAX_COUNT*2)-1]);
}
// ************** END OF GEN_PATTERN ****************


// *************** DA_UPDATE *****************
// Setup D/A to write a triangle pattern to either D/A channel.
// Using 8254 Counter2 Output for the update clock.
// In Pass Through mode.

void da_update(int board_no)
{
    #define DATA_MASK   0x0000FFFF
    #define FIFO_EMPTY  0x20
    #define FIFO_FULL   0x80
    #define CH_MASK     0x01
    byte dac_channel =0;
    byte dac_on;
    dword dac_data =0;
    int i;


    printf("\nSelect DAC Channel '0' or '1': ");
    scanf("%x", &dac_channel);

    // Configuring for only one channel.
    *pDA_CHCNT[board_no] = 0x00;

    // DAC Channel 0 or 1 (from user input above)
    *pDA_QRAM[board_no] = (dac_channel & CH_MASK);

    // Paced mode with SW Write/Virtual Write for update clock source
    *pDA_UPCK_CONFIG[board_no] = 0x00;

    // No conversion, (Pass-Thru)Target mode, D/A Fifo enabled
    *pDA_CONTROL[board_no] = 0x02;

    // Activating Board's D/A Output
    // Must keep this set (0x01).
    // If cleared (0x00) then D/A outputs become grounded on board.
    dac_on = 0x00;
    *pDA_ENABLE[board_no] = dac_on;

    // Writing data to update DAC
    // For PassThru only the lower word is used. So masking upper word to zeroes.
    for(i=0; i<(MAX_COUNT*2); i++)
    {
            if( !(*pDA_CONTROL[board_no] & FIFO_FULL))
            {
                *pDA_DATA[board_no] = pattern[i] & DATA_MASK;
            }
            else
            {
                while( !(*pDA_CONTROL[board_no] & FIFO_EMPTY));
                printf("Count %d: FIFO status bits: full-half-empty %x.\n",
i,(*pDA_CONTROL[board_no] & 0xE0));
            }
    } // for i
    // Waiting till the D/A FIFO is empty (completed updating)
    // This may not work because the internal FIFO might not be deep enough.
    // 8/13/99: Only seeing 1/2 of data or just rising edge of waveform.
    while ( !(*pDA_CONTROL[board_no] & FIFO_EMPTY) );

}
// ************* END OF DA_UPDATE *******************
```