Programming Data Acquisition Systems

A/Prof Jay Katupitiya
Mechatronics
School of Mech. & Manf. Eng.
J.Katupitiya@unsw.edu.au

September 8, 2009

Abstract

This presentation is on the programming of Data acquisition systems. It will explain the address map, memory mapping of a data acquisition card and developing routines for various data acquisition tasks. It will also describe the development of a data acquisition module.

What is a data acquisition system?

A data acquisition system is a packaged electronic hardware unit that facilitates the interfacing of signals between a peripheral device and a computer. A peripheral device could be any system that requires interaction with a computer. From the world of auto-motives the peripheral device could mean the ABS or an engine torque control system. The term acquisition is misleading as, sometimes, data acquisition systems are used to control the peripheral devices. The data acquisition systems come in various different forms. They could be stand alone systems with just one USB cable connecting it to the computer, or it can be in the form of a card that can be plugged into the computer. To get the jobs done the data acquisition systems (DAS) need to be programmed.

Information needed for programming a DAS?

- Specifications $\rightarrowtail$ i.e capabilities of the system
- Register structure $\rightarrowtail$ i.e. the addresses of the registers
- Register descriptions $\rightarrowtail$ i.e what can each register do.

All these can be obtained from the documentation that comes with the DAS. A number of example documentations can be found on WebCT Vista.

Specifications?

- Digital Inputs
- Digital Outputs
- Analog Inputs
- Analog Outputs
- Counter-Timers

Register Structure

- Where are the registers mapped to?
- How are the registeres arranged within the chosen area of the I/O memory?

Register Mapping

The registers of the DAS are mapped to a desired location by setting switches on the DAS. However, not all DAS have switches. Especially the PCI type cards do not have switches. They get mapped automatically to a free space.

An example switch set is shown in Fig 1.

Switch	1	2	3	4	5	6
Line	A9	A8	A7	A6	A5	A4

Figure 1: Switches for register mapping

How does the switches work?

- Address locations are determined by the bit patterns of the address bus.
- For ISA bus only ten address lines are used - from A9 - A0.
- Each switch corresponds to an address line.
- Suppose a switch that is ON is 1 and a OFF, 0.
- And consider the switch pattern below. Note that there are no switches for the address lines from A3 - A0.

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	0	0	1	X	X	X	X

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	0	0	1	X	X	X	X

First Example

- When all Xs are ZEROS, the address bit pattern evaluates to 0x390.

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0		
1	1	1	0	0	1	0	0	0	0	=	0x390

- When all Xs are ONES, the address bit pattern evaluates to 0x39F.

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0		
1	1	1	0	0	1	1	1	1	1	=	0x39F

- Hence the address range of the card is 0x390-0x39F.

Second Example

- When all Xs are ZEROS, the address bit pattern evaluates to 0x210.

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0		
1	0	0	0	0	1	0	0	0	0	=	0x210

- When all Xs are ONES, the address bit pattern evaluates to 0x21F.

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0		
1	0	0	0	0	1	1	1	1	1	=	0x21F

- Hence the address range of the card is 0x210-0x21F.

Arrangement of Registers

Address	Read	Write
BASE+0	A/D low byte & channel	Software A/D trigger
BASE+1	A/D high byte	A/D range control
BASE+2	MUX scan	MUX scan channel & range control pointer
BASE+3	DIO low byte (DIO 0-7)	DIO low byte (DIO 0-7)
BASE+4	N/A	D/A output data (PCM-3718HO only)
BASE+5	N/A	D/A output data (PCM-3718HO only)
BASE+6	N/A	FIFO interrupt control (PCM-3718HO only)
BASE+7	N/A	N/A
BASE+8	Status	Clear interrupt request
BASE+9	Control	Control
BASE+10	N/A	Counter enable
BASE+11	DIO high byte (DIO 8-15)	DIO high byte (DIO 8-15)
BASE+12	Counter 0	Counter 0
BASE+13	Counter 1	Counter 1
BASE+14	Counter 2	Counter 2
BASE+15	N/A	Counter control

Figure 2: Arrangement of DAS registers

Arrangement of Registers

- The arrangement of registers was shown in the last figure. In this table, which may be called the register table, each row has an address expressed with respect to the base. Then the purpose of each register is described under the "Read" column and the "write" column.

- As BASE we take the starting address of the address range described earlier. The BASE for the second example described earlier is 0x210.

Writing Data to Digital Outputs

The registers that can be used for this purpose is in the table entry against BASE+3 and BASE+11, in the "write" column. Out of 16 possible digital output lines, the 8 lines corresponding to the 8 bits at address BASE+3 is called the Low Byte and the byte at BASE+11 is called the High Byte.

Digital Output Functions

We can write two functions to write to the two ports as follows. Note that the parameters are BASE and the data to be written as one whole byte.

```
void DigitalOutLow(int base, unsigned char lowByte)
{
    outb(lowByte, base+0x03);
}

void DigitalOutHigh(int base, unsigned char highByte)
{
    outb(highByte, base+0x0B);
}
```

Reading Data from Digital Inputs

The registers that can be used for this purpose is in the table entry against BASE+3 and BASE+11, in the "read" column. Out of 16 possible digital input lines, the 8 lines corresponding to the 8 bits at address BASE+3 is called the Low Byte and the byte at BASE+11 is called the High Byte.

Digital Input Functions

We can write two functions to read from the two ports as follows. Note that the parameters are BASE and the data read is a whole byte.

```
unsigned char DigitalInLow(int base)
{
    return inb(base+0x03);
}

unsigned char DigitalInHigh(int base)
{
    return inb(base+0x0B);
}
```

Interfacing Analog Inputs

- Examples of analog signals are temperature, pressure, speed and position. Unlike the digital signals, which have only two states (hence two values 0 or 1), the analog signals will have a range of values.

- When and analog signal is digitized, the entire range of possible analog signal values will be represented by all possible combinations of the "bits" associated with the analog to digital conversion.

- For example, if the temperature changes from -40 to +40, and the corresponding voltages generated by the temperature sensor are from -5 to +5, then an "8-bit" analog to digital conversion system will slice that range into 256 equal segments.

- When the voltage is -5, the 8 bits will be set to all 0s and when the voltage is +5 the 8 bits will be set to all 1s and for all in between voltages, in between 8-bit patters will be delivered with the integer value represented by the bit pattern proportional to the voltage.

- Generally, in a DAS, there will be just one analog to digital convertor and many input channels. Only one channel at a time will be directed to the analog to digital convertor via a multiplexer (MUX).

Procedure to read analog inputs

1. Set input range (BASE+1 -write)
2. Select channel (BASE+2 -write)
3. Trigger a conversion (BASE+0 -write)
4. Wait for conversion to complete (BASE+8 -read)
5. Read the data (BASE+0/1 -read)

The text within brackets refer to the entries in the table in Slide No: 9.

Return Value Type and Parameters for Analog Input

- Given the description of the procedure, we can determine the parameters to our function that will read an analog input channel
- These are; `int BASE`, `unsigned char range` and `unsigned char channel`.
- The return value type must also be determined. Given that the data is 12 bits (refer to BASE+0/1) and if the sign is important we can set the return value type to `int`.

Skeletal `AnalogIn()` Function

```
int AnalogIn(int BASE, unsigned char range, unsigned char channel)
{
    // Set range
    // Select channel
    // Trigger Conversion
    // Wait for conversion to complete
    // Read data
    // return converted data
}
```

Setting the range

The format of the range control register is shown in the figure below:

BASE+1 (write only) - A/D range control code								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	N/A	N/A	N/A	N/A	G3	G2	G1	G0

Figure 3: Range control register - only lease significant 4 bits used

The way we can limit the user to input correct range values will be described later.

Skeletal `AnalogIn()` Function

```
int AnalogIn(int BASE, unsigned char range, unsigned char channel)
{
    // Set range
    outb(range, BASE+1);
    // Select channel
    // Trigger Conversion
    // Wait for conversion to complete
    // Read data
    // return converted data
}
```

Channel Selection

BASE+1 (write only) - A/D range control code								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	N/A	N/A	N/A	N/A	G3	G2	G1	G0

The register that associates with the channel selection is shown above. It has two groups of 4 bits with the least significant 4 bits selecting a channel number from which to start analog to digital conversion and the most significant 4 bits selecting the last channel to be converted. If we are interested in selecting just one channel, we will set both nibbles to the parameter `channel`.

Program Segment for Channel Selection

```
outb(BASE+2, channel | (channel << 4));
```

If need be channel numbers could be tested to see if they are legal channel numbers. This may be done at the beginning of the function.

Skeletal `AnalogIn()` Function

```
int AnalogIn(int BASE, unsigned char range, unsigned char channel)
{
    // Set range
    outb(range, BASE+1);
    // Select channel
    outb(BASE+2, channel | (channel << 4));
    // Trigger Conversion
    // Wait for conversion to complete
    // Read data
    // return converted data
}
```

Trigger Conversion

To trigger a conversion, all that needs to be done is to write "any data" to the register at BASE+0.

```
outb(0x00,BASE+0x0);
```

Skeletal `AnalogIn()` Function

```
int AnalogIn(int BASE, unsigned char range, unsigned char channel)
{
    // Set range
    outb(range, BASE+1);
    // Select channel
    outb(BASE+2, channel | (channel << 4));
    // Trigger Conversion
    outb(0x00,BASE+0x0);
    // Wait for conversion to complete
    // Read data
    // return converted data
}
```

Wait for Conversion to Complete

BASE+8 - A/D status								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	EOC	N/A	MUX	INT	CN3	CN2	CN1	CN0

The register associated with this is shown in the figure above. The documentation says that as long as the bit EOC is 1, the convertor is busy and when it is 0 the conversion is complete. This can be programmed as follows, using a `while` loop. We can read the port at BASE+0x08 and then filter out all bits except D7. If the result is non-zero (i.e. true), then the conversion is not complete yet and we must stay in the `while` loop

---

```
while(inb(BASE+0x08) & 0x80);
```

Skeletal AnalogIn() Function

```
int AnalogIn(int BASE, unsigned char range, unsigned char channel)
{
    // Set range
    outb(range, BASE+1);
    // Select channel
    outb(BASE+2, channel | (channel << 4));
    // Trigger Conversion
    outb(0x00,BASE+0x0);
    // Wait for conversion to complete
    while(inb(BASE+0x08) & 0x80);
    // Read data
    // return converted data
}
```

Reading the Converted Data

BASE+0 (read only) - A/D low byte & channel number								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	AD3	AD2	AD1	AD0	C3	C2	C1	C0

BASE+1 (read only) - A/D high byte								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	AD11	AD10	AD9	AD8	AD7	AD6	AD5	AD4

The data registers we need to read are shown in the above figure. However, after reading the bits AD11-AD0 must be properly positioned with AD0 in the least significant bit location. Also note that the data requires 12-bit space and hence we need at least an `int` data type.

Code Segment for Reading Data

```
int LoByte, HiByte;
LoByte = inb(BASE+0x00);
HiByte = inb(BASE+0x01);
return (LoByte >> 4) | (HiByte << 4);
```

AnalogIn() Function

```
int AnalogIn(int BASE,
             unsigned char range,
             unsigned char channel)
{
    int LoByte, HiByte;
    // Set range
    outb(range, BASE+1);
    // Select channel
    outb(BASE+2, channel | (channel << 4));
    // Trigger Conversion
    outb(0x00,BASE+0x0);
    // Wait for conversion to complete
    while(inb(BASE+0x08) & 0x80);
    // Read data
    int LoByte, HiByte;
    LoByte = inb(BASE+0x00);
    HiByte = inb(BASE+0x01);
    // return converted data
    return (LoByte >> 4) | (HiByte << 4);
}
```