xx

COE538 Microprocessor Systems
Lab 3: Battery and Bumper Displays[1]

Peter Hiscocks
Department of Electrical and Computer Engineering
Ryerson University
phiscock@ee.ryerson.ca

Contents

1 Overview **1**

2 Calculating and Displaying Battery Voltage **2**

3 The Analog to Digital Converter **2**
3.1 Using the A/D Converter 3
3.2 An A/D Conversion Routine 5
3.3 Testing A/D Conversion Routine 7

4 Some Math Required **8**
4.1 Scaling The Equation 8

5 Writing Battery Voltage to the Display **9**

6 Displaying Bumper Status **9**
6.1 Reading the Bumper Switches 9
6.2 Display Architecture 11
6.3 Display Mechanics 11

7 Assignment Summary **12**

8 Binary 16 to BCD Conversion Routine **13**

9 BCD to ASCII Conversion Routine: Version 1 **14**

10 BCD to ASCII Conversion Routine: Version 2 **15**

11 The General Structure of the Program **16**

1 Overview

The objective of this exercise is to develop software that can read an analog quantity (such as the power supply voltage) and display it on the LCD. This includes the following steps:

- Read the A/D Converter
- Do a fixed point calculation, including scaling
- Display a binary number on the LCD using binary-BCD and BCD-ASCII conversion routines

[1] This lab was adapted to be used with the HCS12 microcontroller by V. Geurkov.

2 Calculating and Displaying Battery Voltage

The *eebot* is fed by a 9.6 volt NiCad battery, when it is operating autonomously (on its own). It is important to monitor this voltage in order to ensure that the robot is operating correctly. The motors each draw a variable current of about 300mA, and this also tends to pull down the supply voltage. So the power supply voltage fluctuates according to load. When the supply voltage falls below about 7 volts, the 5 volt logic supplies will degrade and eventually the microprocessor will stop functioning.

The simplified circuit for the power supply monitor input is shown in figure 1.

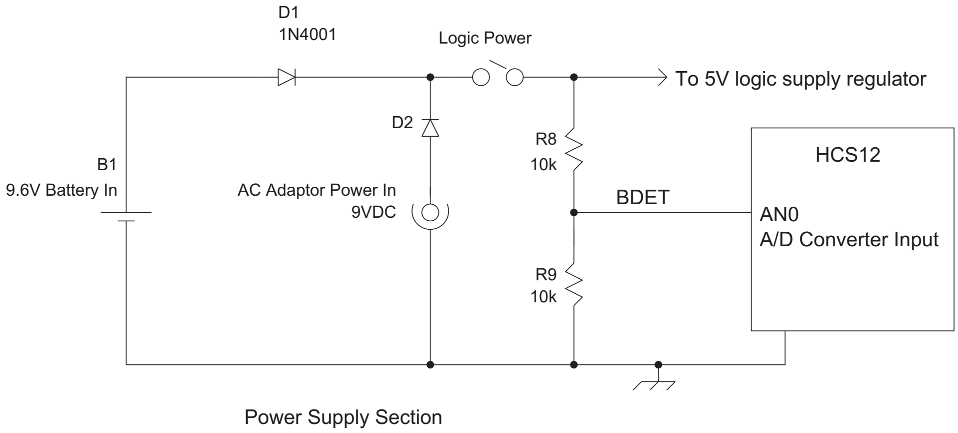

Figure 1: Voltage Monitor, Programming Model

In this exercise, we will construct the software to read and display the *eebot* battery voltage.

There are three stages to this:

1. Read the A/D converter channel 0 voltage (9.6 volt).
2. Process it through the equation relating battery voltage to A/D reading.
3. Convert this reading to an ASCII string and write it to the display.

At this point, we need to switch from the bench machine to the *eebot* [1]. The microcontroller employed in the *eebot*, mc9s12c32, has less resources compared to the one used in the bench board, mc9s12dg128. Namely, it can address only 32KB of memory space (vs 128KB). Also, it has only one A/D converter (vs two converters for the bench one). The mc9s12c32 RAM ranges from $3800 to $3FFF (vs $2000 ∼ $3FFF). And not all of the mc9s12dg128 ports are available on the mc9s12c32. Consequently, the *eebot's* LCD is connected to ports B and J, instead of ports S and E as on the bench board. Table 1 and figure 2 below summarize the relationship between the standard LCD signals and the corresponding pins of ports B and J of the mc9s12c32. As it can be seen from the figure, this interface allows 4-bit as well as 8-bit data transfers.

3 The Analog to Digital Converter

There are many applications where it is useful for a microcomputer to be able to read an analogue voltage. For example, many sensors output a continuously variable signal and this must be converted into a binary number for use by the microcomputer. In the case of the *eebot*, there is a manual potentiometer that generates a variable voltage, and 6 illumination sensors in the line follower circuitry that generate voltages in the range of 1 to 4 volts.

The A/D converter of the HCS12 reads voltages between zero and 5 volts, and generates a binary number proportional to the input voltage.

Table 1: LCD Signal Summary

LCD Signals			9S12C Port Pins		
Data 0	-	D0	Port B/0	-	PB0
Data 1	-	D1	Port B/1	-	PB1
Data 2	-	D2	Port B/2	-	PB2
Data 3	-	D3	Port B/3	-	PB3
Data 4	-	D4	Port B/4	-	PB4
Data 5	-	D5	Port B/5	-	PB5
Data 6	-	D6	Port B/6	-	PB6
Data 7	-	D7	Port B/7	-	PB7
Reset	-	RS	Port J/6	-	PJ6
Enable	-	E	Port J/7	-	PJ7

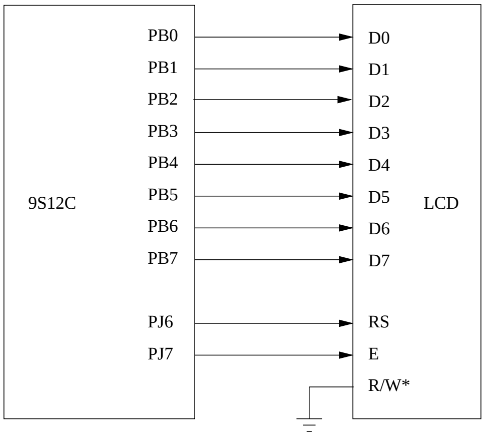

Figure 2: LCD Layout

In formula form,

$$N_{AD} = \frac{V_{in}}{V_{ref}} N_{max}$$

where N_{AD} is the value produced by the A/D converter, V_{in} is the input voltage, V_{ref} the reference voltage (in this case, the 5 volt logic supply), and N_{max} the maximum value contained in 8 bits (255).

For example, a 5 volt input will produce an output of 255 and a 2.5 volt input would produce 127. Notice as well that each *step* in the A/D converter output is equivalent to $5V/255 = 0.0196$ volts (approximately 20mV). It directly follows from the above formula: $1 = V_{in}/5V \times 255$.

The A/D of the HCS12 is preceded by an 8 channel analog multiplexer, so that any of 8 possible analog inputs (channels $0 \sim 7$) can be selected for conversion.

3.1 Using the A/D Converter

The HCS12 A/D converter is shown in figure 3 [5]. Its operation is controlled by certain registers that must be initialized before the use of the converter. A part of this initialization is done by the *reset sequence* which is initiated every time the reset button is depressed. The remaining initialization is taken place in the Serial Monitor. Note that if a (stand-alone) program runs on a microcontroller which does not have the Serial Monitor, this program would have to contain the missing initialization instructions.

The after-reset state of microprocessor control registers is indicated in data sheets by the acronym `res`. As sown in figure 4, the state of one of the control registers for the HCS12 A/D converter, the ATDCTL2 after reset is %00000000. It can later be changed by the programmer. For example, if you intend to use the A/D converter,

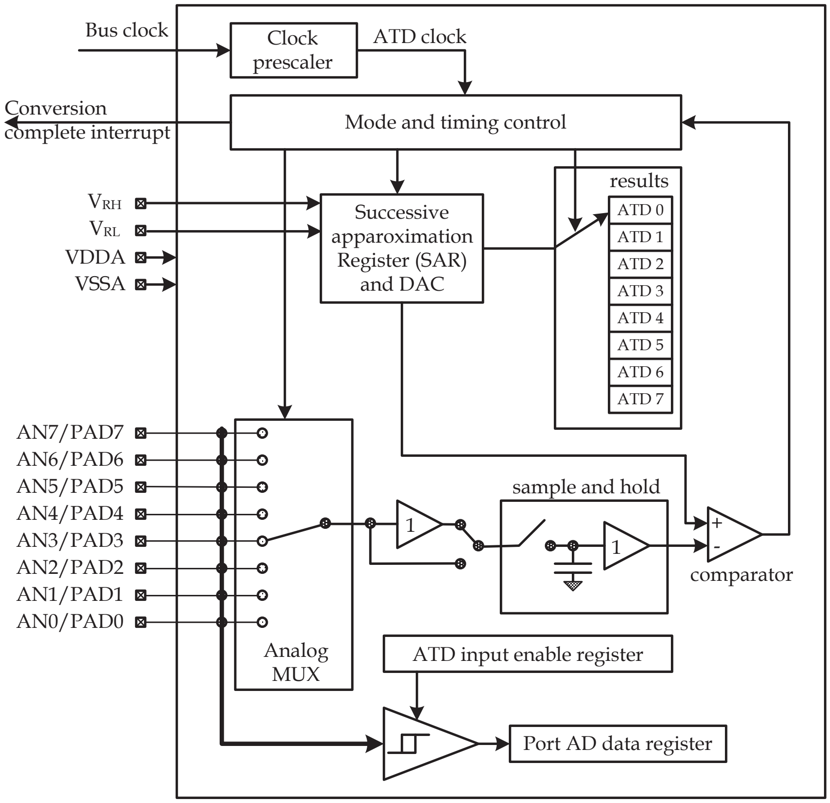

Figure 3: Analog-to-Digital Converter

the ADPU bit of the ATDCTL2 should be set to 1 to power up the A/D converter[2].

Few other important control registers for the A/D converter are the ATDCTL3, ATDCTL4 and ATDCTL5 shown in figures 5, 6 and 7 respectively (for the sake of convenience, these figures only describe the bits that are essential for our exercises; other bits can be left intact).

We will also use the Status Register 0 (ATDSTAT0) and the Conversion Result Register (ATDDRy, $y = 0 \sim 7$). The Sequence Complete Flag (SCF) is the most significant bit of the ATDSTAT0. It is set, when the conversion sequence is completed. The registers ATDDRy contain the results of the conversion sequence. Depending on the configuration of the control registers, we can perform from 1 to 8 conversions (see figure 5). (For more details, refer to [5], section 12.3).

These registers:

- select which one (or group of up to 8) of the 8 possible input channels will be converted
- selects whether the A/D will read one input or several successive inputs
- selects whether the A/D takes a reading on request or continuously, on its own
- indicates when a conversion sequence is complete

For our purposes, we will read the 8 input channels (AN0 to AN7) and take readings on request. Notice that a new request is initiated by writing (anything) to the ATDCTL5, so no explicit request is required.

When a group of 8 readings is requested, they are put in the A/D Result registers, ATDDR0 through ATDDR7, which are located at $0090 trough $009E.

[2] Note that the name of this as well as other control registers is different in the mc9s12c32 and mc9s12dg128 microcontrollers, ATDCTL2 vs ATD0CTL2 and ATD1CTL2 respectively. If you are in doubt of the correct name, check it with the appropriate .inc file in the CodeWarrior.

	7	6	5	4	3	2	1	0
ATD CTL2	ADPU	AFFC		ETRIGLE	ETRIGP	ETRIGE	ASCIE	ASCIF
reset:	0	0	0	0	0	0	0	0

ADPU: ATD power down bit
  0 = power down ATD; 1 = normal ATD operation
AFFC: ATD fast flag clear all bit
  0 = ATD flag is cleared normally, i.e., read the status reg-r before reading the result reg-r
  1 = any access to a result register will cause the associated CCF flag to clear automatically if it is set at the time

ETRIGLE \| ETRIGP

ETRIGLE	ETRIGP	
0	0	falling edge
0	1	rising edge
1	0	low level
1	1	high level

ETRIGE: External trigger mode enable
  0 or 1 = disable or enable external trigger on ATD channel 7
ASCIE: ATD sequence complete interrupt enable bit
  0 or 1 = disable or enable ATD interrupt on sequence complete (ASCIF = 1)
ASCIF: ATD sequence complete interrupt flag
  0 or 1 = no ATD interrupt occurred or ATD sequence complete interrupt pending

Figure 4: ATD Control Register 2

	7	6	5	4	3	2	1	0
ATD CTL3	0	S8C	S4C	S2C	S1C	FIFO		
reset:	0	0	0	0	0	0	0	0

S8C,S4C,S2C,S1C: Conversion sequence limit
  0000 = 8 conversions
  0001 = 1 conversion
  0010 = 2 conversions
  …
  0111 = 7 conversions
  1xxx = 8 conversions
FIFO: Result register FIFO mode
  0 = conversion results are placed in the corresponding result registers up to the selected sequence length
  1 = conversion results are placed in consecutive result registers (wrap around at end)

Figure 5: ATD Control Register 3

Another important registers that we will use in this lab are the ATD input enable register (ATDDIEN) and the Port AD data register (PORTAD0) (see figure 3). The ATDDIEN allows the user to enable Port AD pins as digital inputs (refer to figure 8). The states of these inputs are written to the PORTAD0 as the ATD conversion is completed.

3.2 An A/D Conversion Routine

Now we need to give some thought to the form of the A/D conversion software.

First, we need to decide how the A/D routine will pass its results (the 8 input channel readings) back to the calling routine. To keep things simple, as part of the subroutine we'll define 8 RAM registers to contain the results (we will be using an 8-bit conversion mode). The 8-byte block of RAM can be defined as:

```
ADDATA  RMB 8   ; Storage for A/D converter results
```

You can access the information in this block of memory with commands like

```
LDAA    ADDATA+7
```

	7	6	5	4	3	2	1	0
ATD CTL4	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0
reset:	0	0	0	0	0	1	0	1

SRES8: Select Resolution: 0 = 10-bits; 1 = 8-bits

SMP1 \| SMP0

SMP1	SMP0		
0	0	2	A/D conversion clock periods
0	1	4	-“-
1	0	8	-“-
1	1	16	-“-

PRS4--PRS0: ATD clock prescaler bits

00101	: 12
01011	: 24
10111	: 48

Figure 6: ATD Control Register 4

which would load the accumulator with the data from channel 7 of the A/D converter.

	7	6	5	4	3	2	1	0
ATD CTL5	DJM	DSGN	SCAN	MULT	0	CC	CB	CA
reset:	0	0	0	0	0	0	0	0

DJM: 0 or 1 - left or right justified data in the result registers
DSGN: 0 or 1 - unsigned or sign. data represent. in the result reg-s
SCAN: 0 or 1 - single or continuous convers. sequences (scan mode)
MULT: 0 or 1 - sample one or several channels
CC, CB, CA: Channel select code

CC	CB	CA	
0	0	0	AN0
0	0	1	AN1
…			
1	1	1	AN7

Figure 7: ATD Control Register 5

	7	6	5	4	3	2	1	0
ATD DIEN	IEN7	IEN6	IEN5	IEN4	IEN3	IEN2	IEN1	IEN0
reset:	0	0	0	0	0	0	0	0

IENx: ATD digital input enable on channel x (x = 0~7)
0 /1 = disable / enable digital input buffer to PTADx

Figure 8: ATD Input Enable Register

Now we can sketch out an algorithm for reading the A/D readings. First, read input channels 0 through 7:

- Initialize the ATDCTL3 bits 6 $\sim$ 3 to 0000 and set the ATDCTL5 bit 4 to 1, so that 8 conversion takes place on 8 successive channels.
- Initialize the ATDCTL5 bit 5 to 0 so that each conversions starts on a request.
- Read the ATDSTAT0 SCF bit 7. One way to to examine this bit is to mask off all the other bits and then test for zero using a conditional branch:

```
           LDAA    ATDSTAT0
           ANDA    #%10000000      ; Mask off all bits except 7
           BEQ     NOT_SET
ITS_SET    (continue here)         ; The flag is set
```

```
               (this code handles the flag set condition)
               BRA     CONTINUE
NOT_SET        (continue here)         ; The flag is not set
               (this code handles the flag not set condition)
CONTINUE       (program continues here)
```

The effect of the mask operation is to reduce all the bits except bit 7 to zero. If bit 7 is also zero, the whole byte is then zero. If bit is not zero, then the whole byte is not zero. Consequently, the condition of bit 7 may be tested by testing the whole byte for zero, using a BEQ (Branch if Equal to zero) conditional branch instruction.

Notice how the *Unconditional Branch* instruction BRA is used to branch around the NOT_SET code.

- When it is set, the conversion is complete. So read the ATD0DR0 through ATD0DR7 into the memory registers ADDATA through ADDATA+7. Now, you can also read/test the PORTAD0 bits.

At this point, all 8 analog input channels have been converted and stored in RAM locations ADDATA through ADDATA+7. If any port AD pins were enabled as digital inputs, their digital states are also available in the PORTAD0.

3.3 Testing A/D Conversion Routine

Hardware

A first step is to ensure that the A/D hardware is working correctly.

Use the WB command to modify location $0085 (ATDCTL5) to $B5. This puts the A/D in continuous scan mode and it starts scanning the eight A/D channels.

Now use the command DB $0090 to display one line of memory data starting at location $0090 (ATDDR0).

You will need to repeat the DB command few times.

Put the potentiometer that is hardwired to the A/D channel 4 (see figure 9) to its minimum setting and the A/D should read close to zero. Put the pot to its maximum setting and the A/D should read $FF (or close to it). Put the pot about mid-point, and the A/D should read something about halfway between 00 and $FF. (Ideally, $7F).

You can also observe contents of the memory locations directly at the memory window of the CodeWarrior.

Software

When the hardware is working as expected, you can test your A/D software as follows:

Using the algorithm of section 3.2 as a guide, write an A/D conversion routine that reads the 8 input channels into 8 RAM locations. The routine should be structured as a subroutine, ie, it should be callable from different locations with a JSR *(Jump To Subroutine)* instruction. The last instruction in the routine must be RTS *(Return from Subroutine)*.

Like all software developed for this course, the subroutine must contain descriptive header documentation.

To test the routine, assemble it with a *stub* test routine. This is a one line driver that calls the routine under test. It will be something like

```
START       JSR ADCCONVERT          ; Read 8 A/D channels into RAM
            SWI                     ; Break to Monitor
```

The subroutine code will follow the stub in the listing.

Assemble and test the routine. Using the potentiometer on A/D channel 5, vary its input voltage. Use the monitor DB instruction to see if the channel 5 reading changes as the pot is rotated.

When the routine works correctly, add it to your subroutine library directory, (`~/ele538/library`).

4 Some Math Required

On the *eebot*, the A/D channel `0` reads the voltage at the centre of an equal resistor voltage divider. The top end of this voltage divider is connected via one diode drop to the battery voltage. (When the bot is operated from a bench supply, the divider is connected to the bench supply voltage minus one diode drop.)

The voltage divider is required because the A/D converter can only cope with voltages up to 5 volts, and the battery supply is nearly double the maximum. The voltage divider effectively divides the `battery_voltage-0.6 volts` by two.

So the formula relating the A/D voltage to the battery voltage is (see figure 1):

$$V_{in} = (V_{batt} - 0.6)/2$$

Recapitulating, the A/D converter of the HCS12 reads voltages between zero and 5 volts, and generates a binary number proportional to the input voltage:

$$N_{AD} = \frac{V_{in}}{V_{ref}} N_{max}$$

where N_{AD} is the value produced by the A/D converter, V_{in} is the input voltage, V_{ref} the reference voltage (in this case, the 5 volt logic supply), and N_{max} the maximum value contained in 8 bits (255).

Combining the previous two equations to solve for battery voltage in terms of the A/D reading N_{AD}, we have[3]:

$$V_{batt} = 0.039 N_{AD} + 0.6$$

For example, an A/D reading of `$7F` (127_{10}) would correspond to a battery voltage of 5.6 volts[4].

4.1 Scaling The Equation

The next challenge is this: we need some way of representing the constants 0.039 and 0.6. We could, for example, include a *floating point math package* and then use the routines in that package to do the math. A floating point package can deal directly with such numbers as 0.039.

However, this is overkill for our application, so we'll use a different trick. The HCS12 can work quite easily with binary integers, so if we can convert the equation to integer numbers, we can use the math routines directly. Multiplying both sides of the previous equation by 1000 will do this for us:

$$\begin{aligned} 1000V_{batt} &= 1000(0.039N_{AD} + 0.6) \\ &= 39N_{AD} + 600 \end{aligned}$$

This process of *multiplying by 1000* is known as *scaling the equation*, and is often necessary to map a mathematical equation to some piece of analogue or digital hardware.

The nice thing about the constant 1000 is that it can be removed very simply by shifting the decimal place 3 places to the left, which we can do during the display routine by careful placement of the decimal point. For example, for N_{AD} equal to `$7F` (127_{10}), the value of $1000V_{batt}$ is 5553, or $V_{batt} = 5.553$ volts. We'd round this off to one decimal place, or 5.5 volts.

This approach to calculation is called *fixed point mathematics* in contrast to *floating point math*, which uses exponents to place the decimal point.

One potentially fatal problem with this approach is the problem of *overflow*. We must make sure that the maximum expected number does not exceed the capacity of the computer registers to represent it. The maximum value of an 8 bit register is 255, and of a 16 bit register 65535.

[3] Make sure you too can do this derivation.

[4] It has been suggested that this equation implies that if the A/D reading N_{AD} is zero, the battery voltage is then 0.6 volts. To be pedantically correct about it, a zero A/D reading implies that the voltage is 0.6 volts or less. In practice, even a fully discharged 9.6 volt ni-cad battery has a residual voltage of several volts (and a very high internal resistance).

The maximum possible number computed by this routine occurs when N_{AD} is 255, in which case $1000V_{batt}$ would be 10545 (10.545 volts). Since this is well below 65535, a dual byte register will be sufficient and overflow should never occur[5].

When you construct this routine, it should be passed the value of N_{AD} in one of the 8 bit accumulators and return the value of $1000V_{batt}$ in the D accumulator.

Before going any further, you should test this routine with various inputs and verify that it works. When it is complete, add it to your library.

5 Writing Battery Voltage to the Display

Now we need to convert the format of the previous answer into a form that can be written to the display. This is a three-step process:

- Convert the 16 bit '$1000 \times V_{batt}$' value, which is in binary, into binary coded decimal (BCD) digits
- Convert the BCD digits into ASCII
- Write the ASCII value to the display, formatting it in the way that the decimal point occurs in the correct location. You will need to use the control instruction of the LCD to position the cursor and to determine where each character is written. (The LCD control codes were given in Lab 2).

The two conversion routines are rather lengthy so they are appended to these lab directions. The *16 bit binary to BCD* routine is in section 7. There are two *BCD to ASCII* routines in section 8 and 10. They illustrate different styles of writing software, and you can choose according to your preference. These three routines are also available in the `~courses/coe538/lib` directory in electronic form so you don't need to type them in.

You must *attribute* the routines, ie, identify that they came from source other than your own work and identify that source.

While you are using these routines, reflect on how much more difficult it would be to use these routines if they were not documented properly. You are also welcome to identify any documentation shortcomings and suggest them to the author.

When the program is completed, demonstrate its operation using the potentiometer on the *eebot*. In this program, you will read the A/D channel `4` (see figure 9 below), so that the LCD readings will range from 0.6 to 10.5 volts. In the future, you can use the same program to read the A/D channel `0` (ie, the actual battery voltage).

6 Displaying Bumper Status

In this section, we'll develop code for displaying the status of the two bumper switches.

6.1 Reading the Bumper Switches

The bumpers may be used to signal the computer program for various purposes. For example, the operator may trigger the bow bumper switch when the motors are to start. Or, when the robot bumps into some object, it may be programmed to stop.

Two bumper switches, one at the bow and one at the stern, generate signals for this purpose. When a bumper switch is actuated, the corresponding LED on the back deck will illuminate.

The bumper signals also feed into 2 channels of the A/D converter so that the microprocessor can detect when a collision has occurred. The equivalent circuit for bumper switches and General Purpose Knob is shown in figure 9.

[5] A point about precision and accuracy: Our original measurement had an accuracy and precision of 8 bits, or one part in 256. Then we multiply this reading by another 8 bit value and get a 16 bit result. The precision of this answer is 16 bits or 1 part in 65535, but the lower 8 bits are meaningless, since the accuracy has not improved and is still 8 bit accuracy.

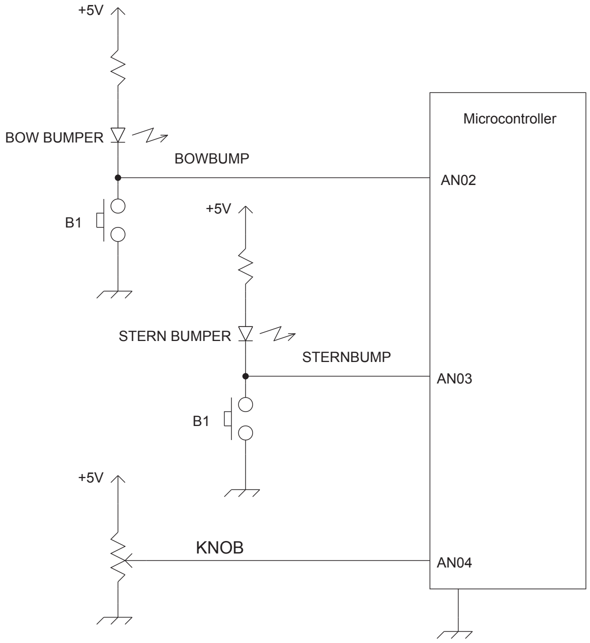

Figure 9: Bumper Switches and Frob Knob

- When a bumper switch is open (no bump), there will be no current through the resistor and LED. The LED has a fixed voltage drop of about 1.5 volts, so the input voltage to the A/D converter channel will be about 3.5 volts. If this channel is enabled as digital input, the 3.5 volts will be interpreted as logic 1.
- When the switch is closed (bump), the input voltage will be zero volts (or logic 0, respectively).
- The general purpose knob (aka *frob* knob) produces an input between zero and 5 volts, which will read between $00 and $FF at the A/D input channel AN04. (Please **note** that on the microcontroller bench system, the knob is connected to the input AN05 (see lab2, figure 1).

In designing the routine to signal a bow or stern bump, we should first decide how we want to record the result. We suggest that you create a two RAM variables named BUMPER_BOW and BUMPER_STERN. If a bumper is activated, its register LSbit is SET; otherwise its LSbit is CLEARED. (You could also use two bits in one RAM variable, but the activation and reading of the bits is a little more tricky.)

As one alternative for testing the bumper registers, you can AND a bumper register with the mask %00000001 to strip out any other bits than the one of interest and then use a BNE or BEQ conditional branch instruction to determine if the LSbit is set or cleared.

With this data structure in mind, our routine to test the bumpers includes the following steps:

- Read the 8 inputs of the A/D converter into RAM registers. (You developed a subroutine to do this earlier in the lab.)
- Check the digital state of the AN02 input. If it is logic 1, set the bow bumper bit. Otherwise, clear the bow bumper bit.
- Check the digital state of the AN03 input. If it is logic 1, set the stern bumper bit. Otherwise, clear the stern bumper bit.
- Return

When this routine is completed, you should also have a routine that can be called as a subroutine and will refresh the information in the two bumper registers.

Now we will give some thought to indicating the state of the bumpers on the display.

6.2 Display Architecture

In the final analysis, we will have a number of things to write to the LCD. This is a good time to give some thought to the overall architecture of the program and the methods it writes to the display.

The obvious method of writing to the LCD is very simple: any routine that needs to put something on the display simply writes it there. We'll refer to this as a *generator-centered* design, because the routine that generates the information puts it on the display. There are at least two problems with this approach:

- direct writing to the LCD by various routines requires that each one of these routines 'understand' where its information should go on the LCD. If the layout of the display changes, then all these routines must be modified, which is very cumbersome and prone to error.
- There is no easy way to control the update rate of the display, since each routine writes to the display whenever it's in the mood.
- There is no centralized control to determine what gets written to the display. If this requirement changes with time (the operator wishes to see different information, for example) there is no easy way to switch the display generators on and off.

A better approach is change the point of control to a *display-centered* architecture. In this arrangement, there is one *display driver* routine that is responsible for gathering the display information together and then putting it on the display. The display driver must know where to get the various pieces of information and how to format them before putting them on the display. But if the layout of the display changes, only the display driver routine needs to be modified. If the display rate is changed, then the display driver routine is simply called more or less frequently. The display driver can choose some data and skip other data according to various and changing requirements.

Like many issues in the architecture of software, when the program is small and simple it's not immediately obvious that you should take a particular approach. Almost anything will work. However, when you get a dozen different elements being written to the display and the display rate must be adjustable, the advantages of the display driver technique become clear.

6.3 Display Mechanics

When the display driver routine gets called, how exactly should it write to the display?

One simple strategy is *wipe and write:* clear the entire display and rewrite it. Unfortunately, this results in a very ugly display. At high update rates the display appears to have various artifacts traveling through it. At low update rates it flashes intermittently. Either way, it's very unpleasant to look at.

A better strategy is *selective replace.* Recall that the display cursor specifies where the next character will be written on the display. Happily, the LCD control codes include commands to position, hide or show the cursor. Thus, to rewrite some part of the display, ensure that the cursor is hidden, position it where the replacement is to occur, and write the new information over the old information. This requires that the new information completely overwrite the old or some garbage may be left behind. So it may be necessary for the display driver to pad the new information with leading or trailing blanks.

An even more intelligent display driver will check to see if information has changed. If it has, it will do a selective replace. Otherwise, it's left alone. This wastes less time and results in a better looking display. (The information generator sets a flag[6] every time the information has changed. The display driver tests the flag to see if it should rewrite the information and then clears the flag every time it updates the screen.)

[6] A flag is simply a memory location that contains a binary 1 or 0.

7 Assignment Summary

On the due date for lab3, you are required to demonstrate a program that reads the potentiometer and shows the equivalent battery voltage reading on the LCD. The same program should read the bumper switches and put symbols on the LCD to indicate whether the bumper switches are open or closed. For example, you might show a blank character if the switch is open and a letter (B for Bow, S for Stern for example) when the switch is closed.

The structure of the program should include a display driver routine that coordinates the writing of the battery voltage and bumper switch status displays. The display of battery voltage should include only one digit to the right of the decimal point.

The general structure of the program is given in section 11.

8 Binary 16 to BCD Conversion Routine

```
* file ref b16todec.asm
*
***********************************************************************
BCD_BUFFER      EQU *  The following registers are the BCD buffer area
TEN_THOUS       RMB 1  10,000 digit
THOUSANDS       RMB 1   1,000 digit
HUNDREDS        RMB 1     100 digit
TENS            RMB 1      10 digit
UNITS           RMB 1       1 digit
BCD_SPARE       RMB 2  Extra space for decimal point and string terminator
NO_BLANK        RMB 1  Used in 'leading zero' blanking by BCD2ASC
*****************************************************************
*             Integer to BCD Conversion Routine
* This routine converts a 16 bit binary number in .D into
*  BCD digits in BCD_BUFFER.
* Peter Hiscocks
* Algorithm:
*  Because the IDIV (Integer Division) instruction is available on
*   the HCS12, we can determine the decimal digits by repeatedly
*   dividing the binary number by ten: the remainder each time is
*   a decimal digit. Conceptually, what we are doing is shifting
*   the decimal number one place to the right past the decimal
*   point with each divide operation. The remainder must be
*   a decimal digit between 0 and 9, because we divided by 10.
*  The algorithm terminates when the quotient has become zero.
*  Bug note: XGDX does not set any condition codes, so test for
*   quotient zero must be done explicitly with CPX.
* Data structure:
* BCD_BUFFER      EQU *  The following registers are the BCD buffer area
* TEN_THOUS       RMB 1  10,000 digit, max size for 16 bit binary
* THOUSANDS       RMB 1   1,000 digit
* HUNDREDS        RMB 1     100 digit
* TENS            RMB 1      10 digit
* UNITS           RMB 1       1 digit
* BCD_SPARE       RMB 2  Extra space for decimal point and string terminator
INT2BCD  XGDX            Save the binary number into .X
         LDAA #0         Clear the BCD_BUFFER
         STAA TEN_THOUS
         STAA THOUSANDS
         STAA HUNDREDS
         STAA TENS
         STAA UNITS
         STAA BCD_SPARE
         STAA BCD_SPARE+1
*
         CPX #0         Check for a zero input
         BEQ CON_EXIT    and if so, exit
*
         XGDX           Not zero, get the binary number back to .D as dividend
         LDX #10        Setup 10 (Decimal!) as the divisor
         IDIV           Divide: Quotient is now in .X, remainder in .D
         STAB UNITS     Store remainder
         CPX #0         If quotient is zero,
         BEQ CON_EXIT    then exit
*
         XGDX            else swap first quotient back into .D
         LDX #10         and setup for another divide by 10
         IDIV
         STAB TENS
         CPX #0
         BEQ CON_EXIT
*
         XGDX           Swap quotient back into .D
         LDX #10        and setup for another divide by 10
         IDIV
         STAB HUNDREDS
         CPX #0
         BEQ CON_EXIT
*
         XGDX           Swap quotient back into .D
         LDX #10        and setup for another divide by 10
         IDIV
         STAB THOUSANDS
         CPX #0
         BEQ CON_EXIT
*
         XGDX           Swap quotient back into .D
         LDX #10        and setup for another divide by 10
         IDIV
         STAB TEN_THOUS
*
         CON_EXIT RTS            We’re done the conversion
```

9 BCD to ASCII Conversion Routine: Version 1

Note: If you use this version, you will also need the *strrev.asm* routine which should be in the *strings.asm* file in the course `\library` directory.

```
* file ref: bcdtoasc.asm

 ;; @name itoa_u16
 ;; Converts an unsigned 16-bit number to a decimal string.
 ;;
 ;; @param AccD 16-bit unsigned number to convert
 ;; @param IX Starting address of string.
 ;; @return Nothing
 ;; @side CC modified
 ;; @author K.Clowes

itoa_u16::
 psha
 pshb
 pshy
 pshx
 pshx
 puly

 ; zero is a special case
 cmpd #0
 bne u16_cont
 ldaa #’0
 staa 0,x
 clr 1,x
 pulx
 bra u16_ret ;We’re outa-here!

 ; it’s not zero
u16_cont:
 ldx #10
 idiv ; AccB is remainder, IX is quotient
 addb #’0 ; Convert remainder to ASCII
 stab 0,y
 iny
 cmpx #0
 beq u16_done
 xgdx
 bra u16_cont
```

```
u16_done:
 clr 0,y ; ensure generated string is null-terminated
 pulx
 jsr strrev ; string is in reverse order-->reverse it!
u16_ret:
 puly ; restore original registers
 pulb
 pula
 rts
```

10 BCD to ASCII Conversion Routine: Version 2

```
* file ref: bcdtoasc.asm
*
**********************************************************************
BCD_BUFFER      EQU *  The following registers are the BCD buffer area
TEN_THOUS       RMB 1  10,000 digit
THOUSANDS       RMB 1   1,000 digit
HUNDREDS        RMB 1     100 digit
TENS            RMB 1      10 digit
UNITS           RMB 1       1 digit
BCD_SPARE       RMB 10  Extra space for decimal point and string terminator
NO_BLANK        RMB 1  Used in 'leading zero' blanking by BCD2ASC

****************************************************************
*               BCD to ASCII Conversion Routine
* This routine converts the BCD number in the BCD_BUFFER
*  into ascii format, with leading zero suppression.
* Leading zeros are converted into space characters.
* The flag 'NO_BLANK' starts cleared and is set once a non-zero
*  digit has been detected.
* The 'units' digit is never blanked, even if it and all the
*  preceding digits are zero.
* Peter Hiscocks

BCD2ASC    LDAA #0           Initialize the blanking flag
           STAA NO_BLANK
*
C_TTHOU    LDAA TEN_THOUS    Check the 'ten_thousands' digit
           ORAA NO_BLANK
           BNE NOT_BLANK1
*
ISBLANK1   LDAA #' '         It's blank
           STAA TEN_THOUS     so store a space
           BRA C_THOU         and check the 'thousands' digit
*
NOT_BLANK1 LDAA TEN_THOUS    Get the 'ten_thousands' digit
           ORAA #$30         Convert to ascii
           STAA TEN_THOUS
           LDAA #$1          Signal that we have seen a 'non-blank' digit
           STAA NO_BLANK
*
C_THOU     LDAA THOUSANDS    Check the thousands digit for blankness
           ORAA NO_BLANK     If it's blank and 'no-blank' is still zero
           BNE NOT_BLANK2
*
ISBLANK2   LDAA #' '         Thousands digit is blank
           STAA THOUSANDS     so store a space
           BRA C_HUNS         and check the hundreds digit
*
NOT_BLANK2 LDAA THOUSANDS    (similar to 'ten_thousands' case)
           ORAA #$30
           STAA THOUSANDS
           LDAA #$1
           STAA NO_BLANK
*
C_HUNS     LDAA HUNDREDS     Check the hundreds digit for blankness
           ORAA NO_BLANK     If it’s blank and ’no-blank’ is still zero
           BNE NOT_BLANK3
*
ISBLANK3   LDAA #’ ’         Hundreds digit is blank
           STAA HUNDREDS      so store a space
           BRA C_TENS         and check the tens digit
*
NOT_BLANK3 LDAA HUNDREDS     (similar to ’ten_thousands’ case)
           ORAA #$30
           STAA HUNDREDS
           LDAA #$1
           STAA NO_BLANK
*
C_TENS     LDAA TENS         Check the tens digit for blankness
           ORAA NO_BLANK     If it’s blank and ’no-blank’ is still zero
           BNE NOT_BLANK4
*
ISBLANK4   LDAA #’ ’         Tens digit is blank
           STAA TENS          so store a space
           BRA C_UNITS        and check the units digit
*
NOT_BLANK4 LDAA TENS         (similar to ’ten_thousands’ case)
           ORAA #$30
           STAA TENS
*
C_UNITS    LDAA UNITS        No blank check necessary, convert to ascii.
           ORAA #$30
           STAA UNITS
*
           RTS               We’re done
```

11 The General Structure of the Program

```
*****************************************************************
* Displaying battery voltage and bumper states (s19c32)         *
*****************************************************************

; Definitions
LCD_DAT     EQU   PORTB                LCD data port, bits - PB7,...,PB0
LCD_CNTR    EQU   PTJ                  LCD control port, bits - PE7(RS),PE4(E)
LCD_E       EQU   $80                  LCD E-signal pin
LCD_RS      EQU   $40                  LCD RS-signal pin

; Variable/data section
            ORG   $3850

TEN_THOUS   RMB   1                    10,000 digit
THOUSANDS   RMB   1                     1,000 digit
HUNDREDS    RMB   1                       100 digit
TENS        RMB   1                        10 digit
UNITS       RMB   1                         1 digit
NO_BLANK    RMB   1                    Used in ’leading zero’ blanking by BCD2ASC

; Code section
            ORG   $4000
Entry:
_Startup:
            LDS   #$4000               initialize the stack pointer
            JSR   initAD               initialize ATD converter
            JSR   initLCD              initialize LCD
            JSR   clrLCD               clear LCD & home cursor
            LDX   #msg1                display msg1
            JSR   putsLCD                  "
            LDAA  #$C0                 move LCD cursor to the 2nd row
            JSR   cmd2LCD
            LDX   #msg2                display msg2
            JSR   putsLCD                  "

lbl         MOVB  #$90,ATDCTL5         r.just., unsign., sing.conv., mult., ch0, start conv.
            BRCLR ATDSTAT0,$80,*       wait until the conversion sequence is complete

            LDAA  ...                  load the ch4 result into AccA
            LDAB  ...                  AccB = 39
            MUL                        AccD = 1st result x 39
            ADDD  ...                  AccD = 1st result x 39 + 600

            JSR   int2BCD
            JSR   BCD2ASC

            LDAA  ...                  move LCD cursor to the 1st row, end of msg1
            JSR   cmd2LCD                  "

            LDAA  TEN_THOUS            output the TEN_THOUS ASCII character
            JSR   putcLCD                  "
            ...                        same for THOUSANDS, ’.’ and HUNDREDS

            LDAA  ...                  move LCD cursor to the 2nd row, end of msg2
            JSR   cmd2LCD                  "

            BRCLR PORTAD0,...,bowON
            LDAA  #$31                 output ’1’ if bow sw OFF
            BRA   bowOFF
bowON       LDAA  #$30                 output ’0’ if bow sw ON
bowOFF      JSR   putcLCD

            ...                        output a space character in ASCII

            BRCLR PORTAD0,...,sternON
            LDAA  #$31                 output ’1’ if stern sw OFF
            BRA   sternOFF
sternON     LDAA  #$30                 output ’0’ if stern sw ON
sternOFF    JSR   putcLCD

            JMP   lbl

msg1        dc.b  "Battery volt ",0
msg2        dc.b  "Sw status ",0

; Subroutine section
initLCD     ...
clrLCD      ...
del_50us    ...
cmd2LCD     ...
putsLCD     ...
putcLCD     ...
dataMov     ...
int2BCD     ...
BCD2ASC     ...
initAD      MOVB  #$C0,ATDCTL2         power up AD, select fast flag clear
            JSR   del_50us             wait for 50 us
            MOVB  #$00,ATDCTL3         8 conversions in a sequence
            MOVB  #$85,ATDCTL4         res=8, conv-clks=2, prescal=12
            BSET  ATDDIEN,$0C          configure pins AN03,AN02 as digital inputs
            RTS

; Interrupt vectors
            ...
```

References

[1] **eebot Technical Description**
Peter Hiscocks, 2002
A complete technical description of the *eebot* mobile robot.
Available on-line.

[2] **Schematics of the EvalH1 Interface Trainer Board**
Available on-line.

[3] **CPU12 Reference Manual**
Motorola Document CPU12RM/AD REV 3, 4/2002
The authoritative source of information about the 68HC12 & HCS12 microcontrollers.

[4] **68HC11 Microcontroller, Construction and Technical Manual**
Peter Hiscocks, 2001
Technical information on the MPP Board, 68HC11 Microprocessor Development System
Information on programming and interfacing the M68HC11 MPP Board.

[5] **HCS12/9S12: An Introduction to Software and Hardware Interfacing**
Han-Way Huang
Delmar Cengage Learning, 2010
A basic text on the HCS12 microcontroller.