ELE538 Microprocessor Systems
Lab 2: Programming the Liquid Crystal Display

Peter Hiscocks
Department of Electrical and Computer Engineering
Ryerson Polytechnic University
phiscock@ee.ryerson.ca
August 27, 2002

Contents

1 Overview **2**

2 Software Delay **2**
2.1 A Simple Loop . . . 2
2.2 Exercise: Simple Delay . . . 3
2.3 Tweaking the Delay Time . . . 3
2.4 Loops Within Loops . . . 3
2.5 A Generalized Delay . . . 3
2.6 Use With Caution . . . 4

3 Bit Mask Operations: Writing to Machine Registers **4**
3.1 Setting and Clearing a Specific Bit . . . 4
3.2 Reading a Specific Data Bit . . . 6

4 Programming the Liquid Crystal Display **6**
4.1 Display Control Instructions . . . 6
4.2 Displaying a Message: Overview . . . 7
4.3 Display Character . . . 8
4.4 Exercise: Display String . . . 9

5 Converting Hex to ASCII **9**
5.1 The ASCII Character Codes . . . 9
5.2 A Conversion Algorithm . . . 12

6 Assignment: Display Message and Hex Value **13**
6.1 Assignment Hints . . . 13

7 Appendix: The Shadow Register **14**

8 Appendix: Initializing the Liquid Crystal Display **14**

9 References **15**

1 Overview

The objectives of this exercise are to learn how to write text and numeric values to the LCD (Liquid Crystal Display). The LCD may then be used as a diagnostic and monitoring tool to print out information that indicates the state of the eebot robot.

To do this, we'll develop some necessary software tools and then explore the programming of the LCD.

- build a *software delay* routine.
- bit mask operations.
- convert a one-byte value to a displayable ASCII string.
- initialize the LCD.
- write messages to the LCD.
- writes a hexadecimal value in hexadecimal notation to the LCD.

2 Software Delay

The microprocessor requires a certain *execution time* to process each machine instruction. This execution time is measured in units of *machine cycles* (aka *the E clock cycle*). The machine cycle time is the basic unit of time in a microprocessor. On the 68HC11 it is 4 times the oscillator period[1]. The machines in the lab have a 4MHz crystal and consequently an oscillator frequency of 4MHz and an oscillator period of 250 nanoseconds. The E clock period is thus 1 microsecond.

When a brief delay is required in a computer program while waiting for something else to happen it is common practice to have the machine repeditively execute a loop of instructions. The loop contains a *loop counter* which is initialized to some value and then incremented or decremented each iteration of the loop. The loop also contains a *conditional branch instruction* that terminates the loop when the count reaches some predetermined value.

2.1 A Simple Loop

A very simple example of a software loop is shown below:

```
        LDAA #$FF        ; Initialize the loop counter ACCA to 255
LOOP    DECA             ; Decrement the loop counter
        BNE LOOP         ; If not done, continue to loop
        (program continues here)
```

The `LDAA#$FF` instruction initializes the accumulator A. Notice the *immediate* addressing mode.

Check the DEC instruction (which includes DECA) in the Pink Book, and you can see under *Condition Codes* that the Z (zero) flag is set when the result is zero.

Next, look at the *Description* of the BNE instruction: it causes a branch if Z is clear, ie, the result is not zero. So the mneumonic BNE should be interpreted as: *Branch if Not Equal* **(to zero)**

[1]On some machines, it is as much as 20 times the oscillator period, which is why computer clock speed is not necessarily indicative of the speed of a computer.

Consequently, the operation of the loop is to start at a loop count of 255 and decrement each trip around the loop until the loop count is zero. At that point, it does not branch and continues with the execution of the program.

Now we are in a position to calculate the delay. Turn back to the Pink Book instruction pages again. We can see that the DEC instruction requires 2 machine cycles, and the BNE instruction requires 3. So each trip around the loop requires 5 machine cycles, or 5 microseconds. For 255 trips, this represents 1275 microseconds, or about 1.2 milliseconds. Certainly not something that a human would ever detect, but a significant delay in microprocessor time.

2.2 Exercise: Simple Delay

Write a simple delay routine using the 16 bit X index register as the loop counter. How many loop counts can this delay execute? What is the maximum delay with a $1micro$second E clock?

2.3 Tweaking the Delay Time

The delay time may be tweaked (adjusted slightly) by adding instructions inside the loop. The `NOP` (No Operation) instruction is useful for this, since it requires 2 machine cycles to execute and has no effect on any machine registers.

2.4 Loops Within Loops

It is possible to *nest* one delay loop inside another, thereby creating a really long delay - in the order of N^2, where N is the loop iteration time. This can produce delays in the order of minutes, if required.

2.5 A Generalized Delay

A software delay routine with a specific delay time is of limited usefulness. It's much more useful to create a software delay where a parameter can specify the delay time. For example:

```
**************************************************************
*                Short Delay
*
* This subroutine generates delays approximately 5 microseconds per
*  count on a 1 microsecond E clock.
* Passed: The delay count in ACCA.
* Returns: (n/a)
* Side-effects: Clobbers ACCA

SHORTDELAY  DECA          ; Decrement the loop counter
            BNE LOOP      ; If not done, continue to loop
            RTS           ; Done, return
```

This can be use as a subroutine and called with with the invocation

```
COUNT       EQU 255
            LDAA #COUNT
            JSR SHORTDELAY
```

where `COUNT` is the delay count. It can be filed away in the 'library of useful routines' and then re-used in a program whenevery a (short) delay is needed.

2.6 Use With Caution

These delays are useful for simple applications but you should be aware of their limitations. For one thing, the machine is not doing useful work while it is executing a delay loop. In some applications, this is not acceptable. For example, the machine user interface should be available at all times. If the machine is off executing a delay loop instead of servicing the keyboard, the machine will appear to have crashed.

You should also be aware that the delay time depends of the computer clock speed. As the CPU clocks become faster, these delays shorten. We have hardware in the lab that will only operate on very old (slow) machines because the software contains such software delay loops. A better solution, where accuracy is important is to use the Real Time Clock (if the processor contains one) to time delays. The RTC should operate at the right speed regardless of the processor clock.

3 Bit Mask Operations: Writing to Machine Registers

It is frequently necessary to write to certain bits of some register, without affecting other bits. Or, when reading a register, it is necessary to examine certain bits while ignoring others.

For example, bit 0 of the MPP board general output register `GPOUT` is used to control the *eebot* starboard motor direction. (`GPOUT` is mapped into the MPP board address space at $1100). When this bit is 0, the motor rotates forward. When it is a 1, the motor rotates in reverse. Obviously, to control the motor direction this bit must be set and cleared.

The other bits in this same register control other machine functions. It is important when we change direction of the starboard motor bit that we not affect any of the other bits in the same register.

The logical `AND` instruction and the logical `OR` instruction can be used for these operations.

3.1 Setting and Clearing a Specific Bit

The correct way to **set** a bit in a register is to `OR` it with a logical 1 in that position (and zeros in the rest of the byte).

To see why this is so, consider the truth table for the Logical OR:

Data	Mask	Result
0	0	0
0	1	1
1	0	1
1	1	1

- ORing any data value with a logical 1 in the mask sets that data to 1.
- ORing any data value with a logical 0 in the mask has no effect on that data.

(For reasons that we'll explain later (see section 7 on page 14), we can't work directly with `GPOUT`. We'll work with something called the *shadow register*, which as an ordinary RAM read-write memory location that contains the identical contents to GPOUT. We'll call this location `GPOUT_SHADOW`.) So the sequence of instructions to set bit 0 in this register is:

```
LDAA GPOUT_SHADOW
ORAA #%0000001
STAA GPOUT_SHADOW
```

(The % symbol indicates to the assembler program that the argument is in binary notation.) Whatever contents are in the other bits of `GPOUT_SHADOW` are unaffected because ORing a 0 with 0 gives a zero and ORing a 0 with a 1 gives a 1.

Similarily, the sequence of instructions to clear a bit in a register is to `AND` it with a zero in that position (and 1's in the rest of the byte).

To see why this works, consider the truth table for the logical AND:

Data	Mask	Result
0	0	0
0	1	0
1	0	0
1	1	1

- ANDing any data value with a logical 0 in the mask clears that data to 0.
- ANDing any data value with a logical 1 in the mask has no effect on that data.

So the sequence of instructions to clear bit 0 of `GPOUT_SHADOW` to zero would be:

```
LDAA GPOUT_SHADOW
ANDA #%11111110
STAA GPOUT_SHADOW
```

Whatever contents are in the other bits of `GPOUT_SHADOW` are unaffected because ANDing a 0 with 1 gives a 0 and ANDing a 1 with a 1 gives a 1.

The arguments to the `OR` and `AND` instructions is often referred to as a *mask* byte because it hides certain bits. (Perhaps *filter byte* would be more descriptive.)

Test your understanding

1. What happens when a data bit is EORed with a mask bit of 0?
2. What happens when a data bit is EORed with a mask bit of 1?
3. What happens to the contents of GPOUT when they are EORed with the mask `%11111111`?

Recall the truth table for the Exclusive Or: *The EOR of two bits is a logical 1 if they are different*:

Data	Mask	Result
0	0	0
0	1	1
1	0	1
1	1	0

3.2 Reading a Specific Data Bit

A similar process can be used to isolate certain bits for testing. When we want to examine bit 7 of some byte called ADCTL, we can mask off all the other bits and then test for zero using a conditional branch:

```
        LDAA ADCTL
        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 the most significant bit is the one being tested it may be tested directly. Recall that an 8 bit 2's complement number has a zero in the most significant bit position when it is positive and a 1 when it is negative. Conseqently, the branch instructions `BPL` (*branch if positive*) and `BMI` (*branch if negative*) can be used directly. However, if the bit to be tested is in some other position in the byte, the data word then has to be rotated so that the bit to be tested is in the most significant position.)

4 Programming the Liquid Crystal Display

In this next section, you will learn how to write a message string to the liquid crystal display (LCD) that is attached to the MPP board. On the MPPV1 stations, the display is 16 characters by 2 lines. On the *eebot* the display is a more generous 20 characters by 2 lines. Programming is the same for both displays.

These LCD modules and others like it include their own microprocessor controller which takes care of refreshing the LCD and transferring data to and from the host microprocessor.

The LCD is mapped into the address space of the 68HC11 microprocessor. Two types of information may be written to the LCD: control bytes and data (display) bytes. Control bytes are written to address $1400. Display bytes are written to address $1401.

The status of the LCD may be determined by reading the control register at address $1400. If the LCD is busy, the MSBit of the data read from the control register will be set; if the LCD is ready, the MSBit will be cleared.

4.1 Display Control Instructions

Display control instructions are used to set up or change such display properties as

- display scroll as new characters are added (or not)
- cursor blinking or steady
- position of the cursor

- show or hide the cursor
- show or hide the display
- type of electrical interface, 4 or 8 bit
- address for the next character to be displayed

If you need to modify the display characteristics or change the position of a the message on the LCD, you will have to issue control instructions to the LCD. The control codes for these various functions are summarized in figure 1.

Code	Bits	Function
$01	0 0 0 0 0 0 0 1	Clear display, move cursor to home position (addr 0)
$02	0 0 0 0 0 0 1 0	Home cursor
$06	0 0 0 0 0 1 1 0	Entry Mode Set 0=No display shift on entry, 1=shift display 0=Decrement display address, 1=Increment display address
$0C	0 0 0 0 1 1 0 0	Display On-Off Control 0=No Blink, 1=Blink 0=No Cursor, 1=Cursor 0=Display hide, 1=Display visible
$38	0 0 1 1 1 0 0 0	Display Function Set 0=1 line display, 1=2 line display 0=4 bit interface, 1=8 bit interface
	1 _ _ _ _ _ _ _	Display Address (Character Position) Select Character address: 0 to 15 for upper line 64 to 79 for lower line

Figure 1: LCD Control Codes

4.2 Displaying a Message: Overview

Once the display is initialized, placing a message on the display is simply a matter of writing ASCII characters to the data register address. The display will write these characters in sequence.

Notice that the second line of the 16 character-per-line display does not begin at address **16**, as one might expect, but at decimal **64**. For the 20 character-per-line display on *eebot*, the second line also begins at address 64. This little idiosyncracy has caused more than one programmer to tear out wads of hair in frustration.

Writing a string to the display should be handled by a *display string* routine that is a subroutine and writings a null-terminated character string to the display. The *display string* routine is passed a pointer address to the start of the string and exits when it detects the terminating null character of the string.

The *display string* routine calls another subroutine, *display character*, which writes a single character to the display. This is passed in (say) accumulator A the character to be displayed and handles the hardware interaction with the actual LCD hardware[2].

Notice the philosophy here: the problem is decomposed into one of writing a null-terminated string, and then further decomposed into the simpler problem of writing one character to the display. This has several advantages:

- It keeps each routine short and simple, and makes each routine easier to maintain and debug.
- It protects against hardware changes. If the display hardware were to change, the *display character* would be the only one that would require modification. The *display string* routine would stay exactly the same.
- If we needed to add another display device, we could add a second *display character* routine and select the appropriate one with a software switch mechanism. Again, he *display string* routine would stay exactly the same.

A message string can be incorporated into the assembly language source code like this:

```
TESTMESSAGE FCC   'Hi There!'
            FCB 00
```

The `FCB 00` directive places a null byte at the end of the string, which is detected by the *display string* routine as the string terminator.

The sequence of instructions to write to the display will be something like

```
            LDX #TESTMESSAGE
            JSR DISPLAYSTRING
```

4.3 Display Character

The *display character* routine would be called with code like this:

```
        LDAA #'A'          ; Display A
        JSR DISPLAYCHAR
```

The *display character* routine would include the following features:

- instruction(s) to write the character to the display
- instructions to read the LCD control register and loop until the LCD is ready again, ie, the most significant bit (bit 7) becomes set. (You should mask off all bits except bit 7).
- the subroutine `RETURN` instruction

[2] Among programmers, the writing of *device drivers* is considered in to be a difficult and an advanced skill. This is your first device driver!

4.4 Exercise: Display String

The *display string* routine should include:

- a proper header, as usual
- an indirect load of the character to be printed, using the X index register as a pointer
- a test for the null character that causes the routine to exit when it is detected
- a subroutine call to the *write character* display routine
- an instruction to increment the string pointer (the X index register)

The code to write a string to the displayi would look something like this:

```
TESTMESSAGE FCC   'Hi There!'; The message
            FCB 00 ; The message terminator
            LDX #TESTMESSAGE ; Initializing the pointer into the message
            JSR DISPLAYSTRING   ; Write the string
            SWI ; Break to the monitor
```

followed by the code for the *display string* and *display character* subroutines.

5 Converting Hex to ASCII

The final building block in our quest to display A/D voltage readings on the LCD is a routine to convert 2 nibble hexadecimal value into a displayable character string. We begin with the concept of the *character code*.

5.1 The ASCII Character Codes

The standard for representing of textual information is known as *ASCII*, American Standard Code for Information Interchange. ASCII is a seven bit code, allowing the representation of 128 distinct characters: upper and lower case alphabetics, numerics, punctation, and some control characters. The complete ASCII code is shown in figure 2 on page 10 and figure 3 on page 11.

Dec	Hex	Character	Notes	Dec	Hex	Character	Notes
Control Codes							
0	00	Null		1	01	SOH	
2	02	STX	Start Transmission	3	03	ETX	End Transmission
4	04	EOT	End of Transmission	5	05	ENQ	
6	06	ACK	Acknowledge	7	07	BELL	Ring bell
8	08	BS	Backspace	9	09	HT	Tab
10	0A	LF	Line Feed	11	0B	VT	
12	0C	FF	Form Feed	13	0D	CR	Carriage Return
14	0E	SO	Shift Out	15	0F	SI	Shift In
16	10	DLE		17	11	DC1	XON, resume output
18	12	DC2		19	13	DC3	XOFF, suspend output
20	14	DC4		21	15	NAK	
22	16	SYN		23	17	ETB	
24	18	CAN		25	19	EM	
26	1A	SUB		27	1B	ESC	Escape sequence follows
28	1C	FS		29	1D	GS	
30	1E	RS		31	1F	US	
Printing Characters							
32	20	Space		33	21	!	
34	22	”	Double quote	35	23	#	
36	24	$		37	25	%	Percent
38	26	&		39	27	’	Apostrophe
40	28	(		41	29	)	
42	2A	*		43	2B	+	
44	2C	,		45	2D	-	
46	2E	.		47	2F	/	Forward slash
48	30	0		49	31	1	
50	32	2		51	33	3	
52	34	4		53	35	5	
54	36	6		55	37	7	
56	38	8		57	39	9	
58	3A	:		59	3B	;	
60	3C	<		61	3D	=	
62	3E	>		63	3F	?	

Figure 2: ASCII Character Codes

continued…

<table>
<tr><th>Dec</th><th>Hex</th><th>Character</th><th>Notes</th><th>Dec</th><th>Hex</th><th>Character</th><th>Notes</th></tr>
<tr><td>64</td><td>40</td><td>@</td><td></td><td>65</td><td>41</td><td>A</td><td></td></tr>
<tr><td>66</td><td>42</td><td>B</td><td></td><td>67</td><td>43</td><td>C</td><td></td></tr>
<tr><td>68</td><td>44</td><td>D</td><td></td><td>69</td><td>45</td><td>E</td><td></td></tr>
<tr><td>70</td><td>46</td><td>F</td><td></td><td>71</td><td>47</td><td>G</td><td></td></tr>
<tr><td>72</td><td>48</td><td>H</td><td></td><td>73</td><td>49</td><td>I</td><td></td></tr>
<tr><td>74</td><td>4A</td><td>J</td><td></td><td>75</td><td>4B</td><td>K</td><td></td></tr>
<tr><td>76</td><td>4C</td><td>L</td><td></td><td>77</td><td>4D</td><td>M</td><td></td></tr>
<tr><td>78</td><td>4E</td><td>N</td><td></td><td>79</td><td>4F</td><td>O</td><td></td></tr>
<tr><td>80</td><td>50</td><td>P</td><td></td><td>81</td><td>51</td><td>Q</td><td></td></tr>
<tr><td>82</td><td>52</td><td>R</td><td></td><td>83</td><td>53</td><td>S</td><td></td></tr>
<tr><td>84</td><td>54</td><td>T</td><td></td><td>85</td><td>55</td><td>U</td><td></td></tr>
<tr><td>86</td><td>56</td><td>V</td><td></td><td>87</td><td>57</td><td>W</td><td></td></tr>
<tr><td>88</td><td>58</td><td>X</td><td></td><td>89</td><td>59</td><td>Y</td><td></td></tr>
<tr><td>90</td><td>5A</td><td>Z</td><td></td><td>91</td><td>5B</td><td>[</td><td></td></tr>
<tr><td>92</td><td>5C</td><td>“</td><td></td><td>93</td><td>5D</td><td>]</td><td></td></tr>
<tr><td>94</td><td>5E</td><td>ˆ</td><td>Caret</td><td>95</td><td>5F</td><td>_</td><td>Underscore</td></tr>
<tr><td>96</td><td>60</td><td>‘</td><td>Grave accent</td><td>97</td><td>61</td><td>a</td><td></td></tr>
<tr><td>98</td><td>62</td><td>b</td><td></td><td>99</td><td>63</td><td>c</td><td></td></tr>
<tr><td>100</td><td>64</td><td>d</td><td></td><td>101</td><td>65</td><td>e</td><td></td></tr>
<tr><td>102</td><td>66</td><td>f</td><td></td><td>103</td><td>67</td><td>g</td><td></td></tr>
<tr><td>104</td><td>68</td><td>h</td><td></td><td>105</td><td>69</td><td>i</td><td></td></tr>
<tr><td>106</td><td>6A</td><td>j</td><td></td><td>107</td><td>6B</td><td>k</td><td></td></tr>
<tr><td>108</td><td>6C</td><td>l</td><td></td><td>109</td><td>6D</td><td>m</td><td></td></tr>
<tr><td>110</td><td>6E</td><td>n</td><td></td><td>111</td><td>6F</td><td>o</td><td></td></tr>
<tr><td>112</td><td>70</td><td>p</td><td></td><td>113</td><td>71</td><td>q</td><td></td></tr>
<tr><td>114</td><td>72</td><td>r</td><td></td><td>115</td><td>73</td><td>s</td><td></td></tr>
<tr><td>116</td><td>74</td><td>t</td><td></td><td>117</td><td>75</td><td>u</td><td></td></tr>
<tr><td>118</td><td>76</td><td>v</td><td></td><td>119</td><td>77</td><td>w</td><td></td></tr>
<tr><td>120</td><td>78</td><td>x</td><td></td><td>121</td><td>79</td><td>y</td><td></td></tr>
<tr><td>122</td><td>7A</td><td>z</td><td></td><td>123</td><td>7B</td><td>[</td><td></td></tr>
<tr><td>124</td><td>7C</td><td>|</td><td>Bar</td><td>125</td><td>7D</td><td>]</td><td></td></tr>
<tr><td>126</td><td>7E</td><td>˜</td><td>Tilde</td><td>127</td><td>7F</td><td>DEL</td><td>Delete</td></tr>
</table>

Figure 3: ASCII Character Codes

As the computer has developed, it has become customary to work with information in multiples of one byte, or 8 bits. The 8 bit byte is a good choice for storing ASCII characters: each ASCII character occupies 7 bits, so characters may be stored one per byte with one bit unused.

ASCII values from 0 to 31 are control codes, non-printing characters which are used to communicate control signals in a communication system. The important ones are annotated on the chart.

When writing assembly language for the 68HC11, it is not necessary to look up character codes: they can be represented within quote marks and the assembler will generate the corresponding character code. For example, the instruction `LDAA 'X'` will cause the ACCA to load with the value $58.

5.2 A Conversion Algorithm

A hexadecimal number such as `1A` cannot be printed directly: the digits must be converted into a sequence of two character codes. For example, the hexadecimal number `3F` is decimal `63` which represents the ASCII character **?**. So if you send `1A` to the LCD, it will display the question mark character '**?**'.

Consulting the ASCII table, we can see that the string to print the hexadecimal number `1A` would be `31 41`.

One possible algorithm for converting a hex digit to its ASCII code is the following:

- Consulting the ASCII table, we can see that the ASCII codes for numeric digits 0 through 9 may be obtained by adding $30 to the value of the digit. For example, $7 becomes $37, which is the display code for the character 7.
- For digits greater than 9, (letters between A and F) it is necessary to add a further value of $07. For example, $A becomes $41, which is the display code for the character **A**.

Computer code may often be re-used from another application. An example of HEX-ASCII conversion is found in the source code for the Buffalo Monitor and reproduced below. (Ask your lab supervisor if you're interested in reading the Buffalo Monitor assembly language listing). This example is lacking in documentation and doesn't do exactly what we want, but it's a starting point.

```
************************************************
*  OUTRHLF(), OUTLHLF(), OUTA()
*    Convert A from binary to ASCII and output.
*    Contents of A are destroyed..
************************************************
OUTLHLF  LSRA            shift data to right
         LSRA
         LSRA
         LSRA
OUTRHLF  ANDA #$0F       mask top half
         ADDA #$30       convert to ascii
         CMPA #$39
         BLE  OUTA       jump if 0-9
         ADDA #$07       convert to hex A-F
OUTA     JSR  OUTPUT     output character
         RTS
```

Points to notice:

- It's not clear from the documentation, but this routine is passed a two-digit hexadecimal (binary) number in ACCA and, depending on the entry point `OUTLHLF` or `OUTRHLF`, prints either the left or right digit as an ASCII character. This shows a common practice: dual entry points with a common exit point.
- Entering at `OUTLHLF` outputs the left digit; entering at `OUTRHLF` outputs the right digit. (Notice how the descriptive names are a big help in figuring out the routine function.)
- The `JSR OUTPUT` is definitely not required in our application: we have developed a separate routine for this. If we eliminate that line, we have a routine which is entered with the hex number in ACCA and (depending on the entry point) returns with either the left or right ASCII character in ACCA. A more subtle point is this: it is best if each routine is limited to one simple function. This routine not only does the conversion but sends the character to the output. This is not good practice, because we might want to use the routine in an application where we don't want to send the result to the output.

Now you should be able to reverse engineer the code and understand how it works.

6 Assignment: Display Message and Hex Value

Write a routine to print a text message followed by two hexadecimal values, to the LCD. The text message can be anything you like. It should be stored in the program text as a null-terminated string. The hexadecimal values should be taken from location $6000 and $6001. Your lab supervisor will specify the actual hex values at demo time.

The final program should loop endlessly: clear the display, write the information, wait for one second.

6.1 Assignment Hints

You'll need to build and test this program in a series of steps:

1. Write a routine, however crude, to write one ASCII character to the display. Test it.
2. Restructure this routine as a subroutine and test it.
3. Write a routine to write a string to the display. It's best if this routine uses one of the index registers to step through the string, but if you can't get that to work, use brute force: a series of calls to the *write character* subroutine.
4. Write a routine to convert a hex number into a 2 ascii characters. This routine shouldn't write to the display - it should accept a byte value in the accumulator and then convert it to 2 ascii characters and leave the result in one of the 68HC11 registers.
5. Put all this together to create one program that writes the string and numeric values to the display.
6. Add a delay routine and looping instructions so that the program endlessly loops, each time delaying for 1 second or so, clearing the display, and then writing the string and byte values.

When you have all this code working, you have some valuable intellectual property which should be properly stored and protected for re-use. Create a subdirectory `~/538/library`. Clean up each of the useful routines so that they can be incorporated in other programs, and store them in the library directory. Make a backup and keep it safe.

7 Appendix: The Shadow Register

In an ideal, simple, world it would be possible to read the general-purpose output register GPOUT (at location $1100 in memory), do the mask operations to set or clear the desired bit, and then write it back. So the code would look like this:

```
        LDAA GPOUT          ; Get the register data
ORAA #%00000001    ; set the least significant bit
        STAA GPOUT          ; and store it back.
```

Unfortunately, because of the hardware design of the MPP board, this won't work. It won't work because read and write operations to $1100 don't access the same register: they access different registers. A read operation at $1100 accesses a separate 8 bit input port.

This problem can be overcome by maintaining a *shadow register* in RAM somewhere, that can be read and written normally. The shadow register, which we'll call GPOUT_SHADOW, maintains a current copy of GPOUT. So the drill is to perform any mask operation on GPOUT_SHADOW and then write GPOUT_SHADOW to GPOUT. Then our code becomes:

```
        LDAA GPOUT_SHADOW  ; Get the register data
        ORAA #%00000001    ; set the least significant bit
        STAA GPOUT_SHADOW  ; and store it back.
STAA GPOUT          ; Update the hardware
```

Notice that the system must be initialized with the same information in GPOUT_SHADOW and GPOUT.

8 Appendix: Initializing the Liquid Crystal Display

The Buffalo monitor program has been modified to initialize the LCD and show a sign-on message. If you are downloading a program into the development system under control of the monitor and then running it in RAM you will not need to initialize the the display. However, if you build a stand-alone program that the 68HC11 executes from reset, you will need to initialize the LCD. For the sake of completeness, we include that information here.

The startup sequence specified for the LM016L display, from the Hitachi manual, is shown in figure 4 on page 15.

Our experience is that the displays are very tolerant of the startup sequence and almost anything will work. For example, we have found that it is not necessary to send the initial control code three times, and that once is sufficient. However, figure 4 shows the manufacturer approved procedure.

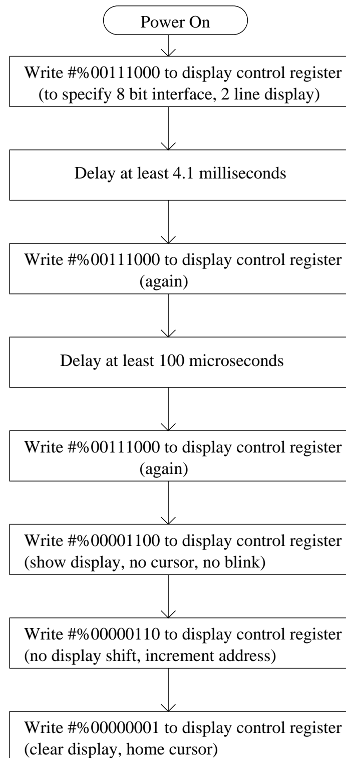

Figure 4: LCD Startup Sequence

9 References

eebot Technical Description
Peter Hiscocks, 2002
A complete technical description of the *eebot* mobile robot used in this course.

M68HC11 Reference Manual
Motorola Document M68HC11RM/AD REV 3, 1991
The authoratative source of information about the 68HC11 microprocessor.
Available from Motorola on request.

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 MPP Board used at Ryerson
and elsewhere.
Available from Active Electronics, at the Victoria Park-Gordon Baker store in Toronto.

M68HC11: An Introduction, Software and Hardware Interfacing
Han-Way Huang
Delmar Thompson, 2001
A basic text on the 68HC11 microprocessor.