Microprocessor Systems
Programming the I/O Devices

Contents

1 Overview

The objective of this exercise is to learn how to exchange data between a microprocessor and I/O (Input/Output) devices. In the first part of the exercise you will explore simple I/O devices, such as switches, a keypad, LEDs (Light Emitting Diodes) and a buzzer. In the second part of the lab, you will 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 hardware/software state of the *eebot* robot.

To do this, we'll develop some necessary software tools and then explore the programming of the I/O devices:

- build a *software delay* routine.
- bit mask operations.
- read from switches and write to LEDs
- 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. It is directly dependant on the oscillator frequency. The bench machines in the lab have a 16MHz oscillator crystal. To further increase the speed of operation, the Serial Monitor converts this frequency to 24MHz. It instructs a special Phase Lock Loop (PLL) circuit in the microcontroller to multiply the 16MHz clock by 3 and then divide it by 2. Thus, the clock period, hereinafter referred to as *E-clock*, is $1/(24\ 10^6) \approx 42$ nanoseconds.

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 repetitively 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 `DECA` in the HCS12 Reference, 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 mnemonic `BNE` should be interpreted as: *Branch if Not Equal* **(to zero)**

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 HCS12 instruction pages again. We can see that the `DEC` instruction requires 1 machine cycle[2]. And the `BNE` instruction requires 3 cycles, if the branch is

[2]Each letter in the **Access Detail** column of the Instruction Set represents a single machine cycle.

taken; otherwise, it requires 1 cycle. So each trip around the loop requires 4 machine cycles, or 4 × 42 = 168 ns. For 255 trips, this represents 42840 ns, or about 43 μs. 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 42 ns 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 1 machine cycle 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 168 nanoseconds
*  per count on a 42 nanosecond E clock.
*  Passed: The delay count in ACCA.
*  Returns: (n/a)
*  Side-effects: Clobbers ACCA

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

This can be used as a subroutine and called 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 whenever 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. Programs containing the software delay loops will normally operate on very slow machines. A better solution, where accuracy is important, is to use the hardware implemented Real Time Clock (if the processor contains one) to time delays. The real time clock 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 *eebot* microprocessor's `Port A` register is used to control the *eebot* Port Motor direction. 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 port 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.

So the sequence of instructions to set bit 0 in this register is:

```
LDAA  PORTA
ORAA  #%00000001
STAA  PORTA
```

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

Similarly, 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 `PORTA` to zero would be:

```
LDAA  PORTA
ANDA  #%11111110
STAA  PORTA
```

Whatever contents are in the other bits of `PORTA` are unaffected because ANDing a 0 with a 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

- What happens when a data bit is EORed with a mask bit of 0?
- What happens when a data bit is EORed with a mask bit of 1?
- What happens to the contents of `PORTA` 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 a register called ATDxSTAT0, we can mask off all the other bits and then test for zero using a conditional branch:

```
            LDAA    ATDxSTAT0
            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. Consequently, 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 The EvalH1 trainer board

In order to explore the HCS12 microprocessor features, we will use a special extension board, referred to as the EvalH1 interface trainer. A simplified schematic of this board is shown in figure 1. The HCS12 board is connected to the EvalH1 through the connector P1. The pins of this connector represent the identical microprocessor signals. We can observe a few *input* devices in figure 1:

- the single-in-line package (SIP) of eight switches (SW1).
- the keypad connected to P1 through the special scan chip (U2-74C922).
- the four push buttons (SW2, SW3, SW4 and SW5).

As well as a few *output* devices:

- the LED bar (LED1) comprising 10 LEDs.
- the color LED with the component signals RGB (LED2).

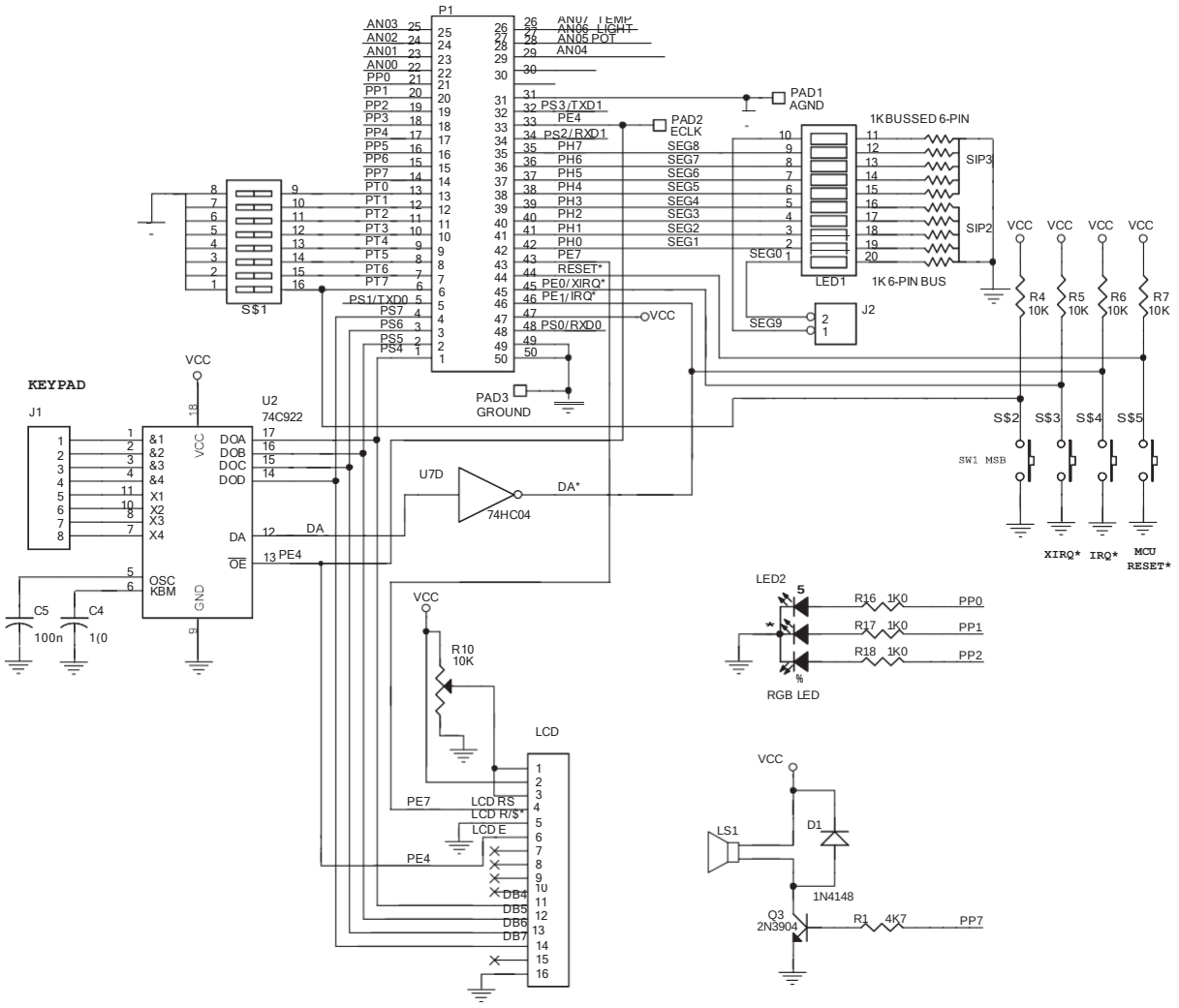

Figure 1: EvalH1 interface trainer

- the buzzer (LS1).
- the LCD module (LCD).

The operation of the switches, push buttons, LEDs and a buzzer is self-explanatory. The other units are explained in the next sections.

5 Reading the Keypad

The HEX keypad layout is shown in figure 2. A Data Available (`DA`) output goes high when a valid keypad entry has been made. At this time, a binary code (from `%0000` to `%1111`) of the key depressed appears at the data outputs `DOA`, `DOB`, `DOC` and `DOD`[3]. The `DA` output returns to a low level when the entered key is released. An internal register remembers the last key pressed even after the key is released. The active low Output Enable ($\overline{\texttt{OE}}$) signal enables the `DOA`, `DOB`, `DOC` and `DOD` outputs.

[3]Using a look up table, this binary code can be optionally converted to ASCII code that matches the character of the key depressed, e.g. if the key `A = %1010` was depressed the ASCII equivalent would be `$41` (see section 9.1).

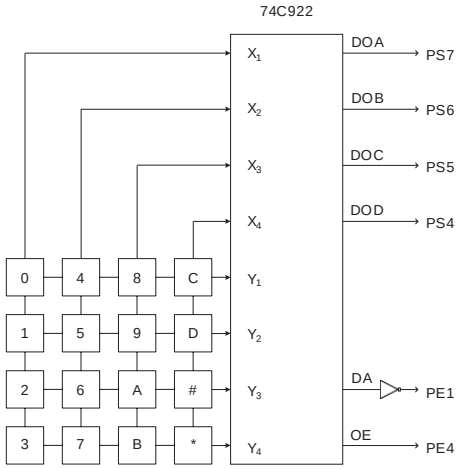

Figure 2: HEX keypad layout

A key code can be obtained by reading the DOA, DOB, DOC and DOD outputs. These outputs are connected to the HSC12 port S which is configured for input after reset. The same applies to port E. Therefore, the only pin that must be re-configured (for output) is the PE4.

The program below demonstrates the keypad read procedure (note the use of BSET and BCLR instructions).

```
BSET    DDRE,%00010000      ; Configure pin PE4 for output (enable bit)
BCLR    PORTE,%00010000     ; Enable keypad
LDAA    PTS                 ; Read a key code into AccA
```

All of the switches and push buttons represented in figure 1 can be read in a similar manner.

6 Interfacing with the LED Bar

The LED bar (LED1 in figure 1) is connected to port H. Since the LED1 is an output device, port H must be configured for output.

The program for driving the LED bar is demonstrated below.

```
BSET    DDRH,%11111111      ; Configure Port H for output
STAA    PTH                 ; Output the AccA content to LED1
```

We can output data to all of the LEDs and the buzzer represented in figure 1 in a similar manner.

7 Assignment 1: Read/Display Data and Generate a Tone

Demonstrate the three exercises given below to your lab instructor.

1. Run the following routine that reads the switches SW1 and immediately displays their states on the LED1.

```
            LDAA    #$FF        ; ACCA = $FF
            STAA    DDRH        ; Config. Port H for output
            STAA    PERT        ; Enab. pull-up res. of Port T

Loop:       LDAA    PTT         ; Read Port T
            STAA    PTH         ; Display SW1 on LED1 connected to Port H
            BRA     Loop        ; Loop
```

Note that switches SW1 in figure 1 are not pulled-up externally (which is required for getting a steady high logic signal). To satisfy this requirement, we can enable the pull-up resistors for input pins of Port T internally, using a special pull device enable register (PERT). The PERT must be written with 1's to enable the pull-ups.

2. Run the following routine to read the keypad. The program uses 3 bits of the acquired key code to control the color LED2.

```
            BSET    DDRP,%11111111 ; Configure Port P for output (LED2 cntrl)
            BSET    DDRE,%00010000  ; Configure pin PE4 for output (enable bit)
            BCLR    PORTE,%00010000 ; Enable keypad

Loop:       LDAA    PTS             ; Read a key code into AccA
            LSRA                    ; Shift right AccA
            LSRA                    ;       -"-
            LSRA                    ;       -"-
            LSRA                    ;       -"-
            STAA    PTP             ; Output AccA content to LED2
            BRA     Loop            ; Loop
```

Note that all configurations are done outside of the main loop.

3. Run the following program to generate a sound tone. A tone is made by creating a digital waveform of appropriate frequency and using it to drive the buzzer LS1. The frequency (of alternating 1's and 0's) is created by means of the software delay routine covered in section 2.

```
            BSET    DDRP,%11111111  ; Config. Port P for output
            LDAA    #%10000000      ; Prepare to drive PP7 high

MainLoop    STAA    PTP             ; Drive PP7
            LDX     #$1FFF          ; Initialize the loop counter
Delay       DEX                     ; Decrement the loop counter
            BNE     Delay           ; If not done, continue to loop
            EORA    #%10000000      ; Toggle the MSB of AccA
            BRA     MainLoop        ; Go to MainLoop
```

---

8 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 microcontroller board. On the stationary microcontroller stations, the display is 16 characters by 4 lines. On the *eebot* the display is a more generous, 20 characters by 2 lines. Programming is the same for both displays.

The LCD modules normally include their own microprocessor controller which takes care of refreshing the LCD and transferring data to and from the host microprocessor.

The LCD can be mapped into the address space of the microprocessor. The programming in this approach is generally easier and more straightforward. The LCD module can also be interfaced directly with an I/O port. In this configuration (that is actually implemented in the boards that we use), the designer will need to use I/O pins to manipulate the control signals. The programming is slightly more cumbersome.

Two types of information can be written to the LCD: control bytes and data (display) bytes. Some information can also be read from the LCD. The status of the LCD, for example, may be determined by reading the control register. 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. Before attempting to write any information, the microprocessor must check the LCD status. This method of communication between the host microprocessor and LCD module is the fastest, but it requires extra hardware and is more complicated. We will use a simpler (though slower) approach to write data to the LCD without reading its status back. Namely, we will provide a sufficiently long delay between the consecutive write operations. This will insure proper completion of the write cycle.

8.1 Display Control Instructions

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

- display scroll as new characters are added
- 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 *1-byte* control instructions to the LCD. The control codes for these various functions are summarized in figure 3. The values shown represent our choice for programming.

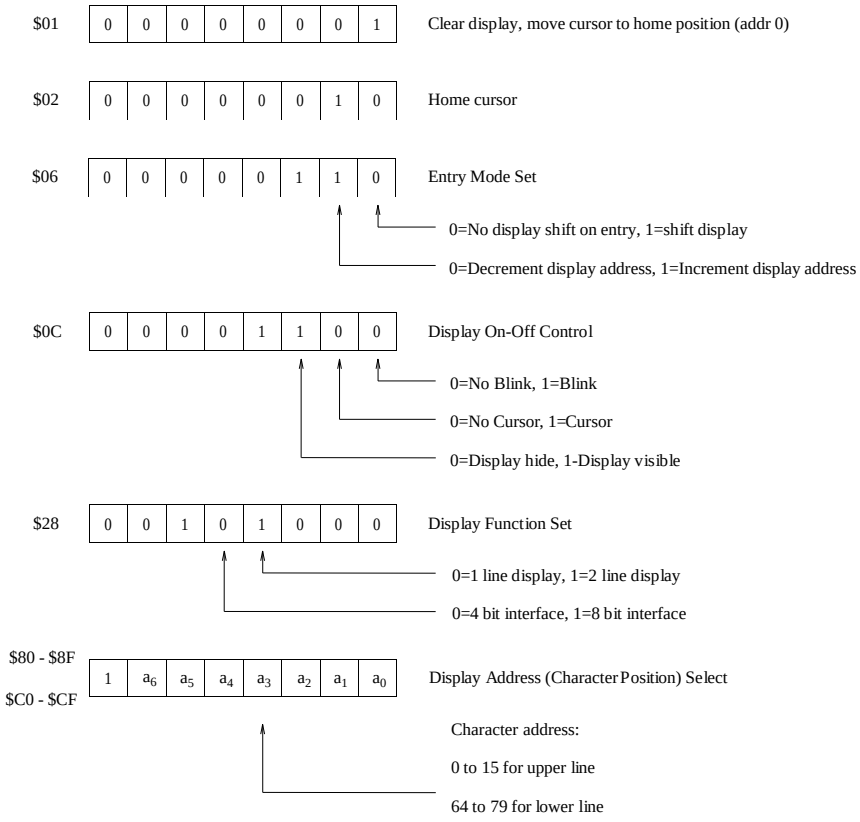

Figure 3: LCD Control Codes

8.2 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 Port. 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**. Similarly, the third and fourth lines begin at addresses **16** and **80** respectively[4]. 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 writes a null-terminated character string to the display. The *display string* routine[5] 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* (also referred to as the *putcLCD* subroutine), which writes a single character to the display. This subroutine passes the character to be displayed in (say) accumulator A and handles the hardware interaction with the actual LCD hardware[6].

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, the *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[7].

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

```
                LDX #TESTMESSAGE
                JSR  DISPLAYSTRING
```

8.3 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 wait (loop) for 50 microseconds until the character write operation is completed.
- the subroutine `RETURN` instruction

[4]For the **20** character-per-line display on *eebot*, the second line also begins at address **64**.

[5]We will also refer to this routine as the *putsLCD* routine.

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

[7]The same effect can be achieved using the following (single) instruction: `dc.b "Hi There!",0`

8.4 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 display 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.

9 Converting Hex to ASCII

In the next lab, we will try to display a 1 byte (or 2 nibble) A/D voltage readings on the LCD. So, the final building block in our quest here will be a routine to convert 2 nibble hexadecimal value into a displayable character string. We begin with the concept of the *character code*.

9.1 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 4 on page 12 and figure 5 on page 13.

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 HCS12, 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.

9.2 A Conversion Algorithm

A single byte can be considered as a two-digit hexadecimal number. This number cannot be printed directly: the two HEX 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 3F 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 3F would be 33 46.

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.

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 4: ASCII Character Codes

- For digits greater than 9, (letters between A and F), in addition to $30 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
```

continued...

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

Figure 5: ASCII Character Codes

```
            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.

10 Assignment 2: 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 $3000 and $3001. Your lab supervisor will specify the actual hex values at demo time (run the program *after* you have entered these values).

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

10.1 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 two-digit hex number into two ASCII characters. This routine shouldn't write to the display - it should accept a byte value in the accumulator and then convert it to two ASCII characters and leave the result in one of the HCS12 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.

Note that the first command to be written to the LCD must be the one that selects the 4-bit interface. Once the LCD accepts the most significant nibble (MSN) of this command, it will detect the 4-bit interface setting and will ignore the rest of the command (i.e. the least significant nibble - LSN). The LCD will then expect immediate transfer of the remaining LSN.

Also note that writing of a data to the LCD occurs at the HIGH-TO-LOW transition of the Enable signal. If the LCD RS-signal is low during this transition, the data (respectively, a byte or a nibble) will be written to the Instruction Register (IR), i.e. it will be interpreted as a command. Otherwise, it will be written to the Data Register (DR) and will be interpreted as a character to be displayed.

For more details about the LCD operation refer to chapter 7.7 (pp. 322-336) of the textbook [5].

Your program should look as follows.

```
*****************************************************************
* Writing to the LCD                                            *
*****************************************************************
; Definitions
LCD_DAT     EQU   PTS              ; LCD data port S, pins PS7,PS6,PS5,PS4
LCD_CNTR    EQU   PORTE            ; LCD control port E, pins PE7(RS),PE4(E)
LCD_E       EQU   $10              ; LCD enable signal, pin PE4
LCD_RS      EQU   $80              ; LCD reset signal, pin PE7

; code section
            ORG   $4000
Entry:
```

```
_Startup:
            LDS   #$4000            ; initialize stack pointer
            JSR   initLCD           ; initialize LCD

*****************************************************************
* Program starts here                                           *
*****************************************************************
MainLoop    JSR   clrLCD            ; clear LCD & home cursor

            LDX   ...               ; display msg1
            JSR   putsLCD           ;    -"-

            LDAA  ...               ; load contents at $3000 into A
            JSR   leftHLF           ; convert left half of A into ASCII
            STAA  ...               ; store the ASCII byte into mem1

            LDAA  ...               ; load contents at $3000 into A
            JSR   rightHLF          ; convert right half of A into ASCII
            STAA                    ; store the ASCII byte into mem2

            LDAA  ...               ; load contents at $3001 into A
            JSR   ...               ; convert left half of A into ASCII
            STAA  ...               ; store the ASCII byte into mem3

            LDAA  ...               ; load contents at $3001 into A
            JSR   ...               ; convert right half of A into ASCII
            STAA  ...               ; store the ASCII byte into mem4

            LDAA  ...               ; load 0 into A
            STAA  ...               ; store string termination character 00 into mem5

            LDX   #mem1             ; output the 4 ASCII characters
            JSR   putsLCD           ;    -"-

            LDY   #...              ; Delay = 1s
            JSR   del_50us
            BRA   MainLoop          ; Loop

msg1        dc.b  "Hi There! ",0

;subroutine section
*****************************************************************
* Initialization of the LCD: 4-bit data width, 2-line display,    *
* turn on display, cursor and blinking off. Shift cursor right.   *
*****************************************************************
initLCD     BSET  DDRS,%11110000    ; configure pins PS7,PS6,PS5,PS4 for output
            BSET  DDRE,%...         ; configure pins PE7,PE4 for output
            LDY   #2000             ; wait for LCD to be ready
            JSR   del_50us          ;      -"-
            LDAA  #$28              ; set 4-bit data, 2-line display
            JSR   cmd2LCD           ;      -"-
            LDAA  #$0C              ; display on, cursor off, blinking off
            JSR   cmd2LCD           ;      -"-
            LDAA  #$06              ; move cursor right after entering a character
            JSR   cmd2LCD           ;      -"-
            RTS

*****************************************************************
* Clear display and home cursor                                 *
*****************************************************************
clrLCD      LDAA  #$01              ; clear cursor and return to home position
            JSR   cmd2LCD           ;      -"-
            LDY   #40               ; wait until "clear cursor" command is complete
            JSR   del_50us          ;      -"-
            RTS
```

```
;*****************************************************************
;* ([Y] x 50us)-delay subroutine. E-clk=41,67ns.                 *
;*****************************************************************
del_50us:   PSHX                    ;2 E-clk
eloop:      LDX   #30               ;2 E-clk -
iloop:      PSHA                    ;2 E-clk  |
            PULA                    ;3 E-clk  |
            ...                               |
            PSHA                    ;2 E-clk  | 50us
            PULA                    ;3 E-clk  |
            NOP                     ;1 E-clk  |
            NOP                     ;1 E-clk  |
            DBNE  X,iloop           ;3 E-clk -
            DBNE  Y,eloop           ;3 E-clk
            PULX                    ;3 E-clk
            RTS                     ;5 E-clk

;*****************************************************************
;* This function sends a command in accumulator A to the LCD     *
;*****************************************************************
cmd2LCD:    BCLR  LCD_CNTR,LCD_RS   ; select the LCD Instruction Register (IR)
            JSR   dataMov           ; send data to IR
            RTS

;*****************************************************************
;* This function outputs a NULL-terminated string pointed to by X  *
;*****************************************************************
putsLCD     LDAA  1,X+              ; get one character from the string
            BEQ   donePS            ; reach NULL character?
            JSR   putcLCD
            BRA   putsLCD
donePS      RTS

;*****************************************************************
;* This function outputs the character in accumulator in A to LCD  *
;*****************************************************************
putcLCD     BSET  LCD_CNTR,LCD_RS   ; select the LCD Data register (DR)
            JSR   dataMov           ; send data to DR
            RTS

;*****************************************************************
;* This function sends data to the LCD IR or DR depening on RS     *
;*****************************************************************
dataMov     BSET  LCD_CNTR,LCD_E    ; pull the LCD E-sigal high
            STAA  LCD_DAT           ; send the upper 4 bits of data to LCD
            BCLR  LCD_CNTR,LCD_E    ; pull the LCD E-signal low to complete the write oper.

            LSLA                    ; match the lower 4 bits with the LCD data pins
            LSLA                    ;      -"-
            LSLA                    ;      -"-
            LSLA                    ;      -"-

            BSET  LCD_CNTR,LCD_E    ; pull the LCD E signal high
            STAA  LCD_DAT           ; send the lower 4 bits of data to LCD
            BCLR  LCD_CNTR,LCD_E    ; pull the LCD E-signal low to complete the write oper.

            LDY   #1                ; adding this delay will complete the internal
            JSR   del_50us          ; operation for most instructions
            RTS

;*****************************************************************
;* Binary to ASCII                                               *
;*****************************************************************
leftHLF     LSRA                    ; shift data to right
            LSRA
            LSRA
```

```
            LSRA
rightHLF    ANDA  #$0F              ; mask top half
            ADDA  #$30              ; convert to ascii
            CMPA  #$39
            BLE   out               ; jump if 0-9
            ADDA  #$07              ; convert to hex A-F
out         RTS
```

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.

References

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

[2] eebot Technical Description
Peter Hiscocks, 2002
A complete technical description of the *eebot* mobile robot.

[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.