MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 4

Lab Guide

Department of Mechanical and Industrial Engineering

University of Toronto

Introduction

We have now been writing simple assembly language programs for three labs. However, useful programs will require even more code than the longest program we have written thus far. In order to create large projects, we need the ability to organize and re-use sections of our code; we need subroutines. In this lab, you will learn how subroutines are handled at the machine level. Then, we will make the transition to high-level code, and learn the benefits and limitations of using a compiler for embedded code. In doing so, we will also begin build a **code library** of useful high-level functions that you can re-use in your later labs and design project.

Trainer Board and LCD Module

We are going to learn about basic subroutines by programming the LCD module on the trainer board. First we will write a simple assembly language subroutine to print a string of text. Then, we will convert this subroutine into a re-usable C function.

The LCD Module

The LCD module connects to Port K through the 68HC12 EVB. The process for using the LCD module involves a **protocol;** a pre-determined, specified language of communication over a particular medium. For the LCD screen, the original protocol follows a specification by Hitachi. Under this protocol, each instruction sent to the LCD screen will be either a single byte, or two 4-bit nibbles, depending on the physical connection. If commands are specified as 8-bits, we also need two bits for control under the specification.

This would mean that the total number of **input / output (I/O) pins** we would need to communicate with the LCD would become 10 (8 for the instructions, and 2 for control). This would use up Port K and part of another port. Thus, for this development board, the manufacturers chose 4-bit nibble mode for LCD communication. We will learn more about I/O in the coming lectures.

For this mode, we connect the LCD screen as follows:

Bit	Pin	Description
RS	PK0	Controls whether an **instruction** or **ASCII data** (text) is being sent to the LCD
RW	PK7	Controls whether data is being **written** or **read** from the LCD
EN	PK1	**Strobe Bit**: Tells the LCD that an instruction has been written or read
DB4	PK2	Least-Significant-Bit, LSB, (D0/D4) of the data nibble
DB5	PK3	Bit D1/D5 of the data nibble
DB6	PK4	Bit D2/D6 of the data nibble
DB7	PK5	Most-Significant Bit, MSB, (D3/D7) of the data nibble

There is more to the protocol than just the simple connections, however. For example, the very first thing that we must do before using the LCD in any way is to initialize it. This process is as follows:

1. Wait at least 15 ms from power on for the LCD to start up.
2. Write 0x03 to LCD and wait 160 ms.
3. Write 0x03 to LCD and wait 160 ms.
4. Write 0x03 to LCD and wait 160 ms.
5. Configure the LCD parameters:
    a. Send "Enable 4-bit mode" command (0x02)
    b. Send "Set Interface Length" command (0x02, 0x08)
    c. Send "Turn off Display" command (0x01, 0x00)
    d. Send "Clear Display" command (0x00, 0x01)
    e. Send "Set Cursor Move Direction" command (0x00, 0x06)
    f. Send "Enable Display/Cursor" command (0x00, 0x01)
    g. Send "Return Cursor to Home" command (0x00,0x02)

This process is best broken into multiple functions, as there is significant overlap in the code functionality. For instance, it would make sense to have a separate function for **sending data** and for **sending an instruction**. We will also need to make a flexible **wait subroutine**. We have already seen in the last lab the difficulty in producing a consistent waiting time. Later, it may also make sense to have a **library** of common commands, such as those to turn on / off the display, move the cursor, clear the display, etc.

The complete set of instructions available for our LCD is as follows:

R/S	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Instruction
0	0	0	0	0	0	0	0	0	1	Clear Display
0	0	0	0	0	0	0	0	1	X	Return Cursor and LCD to Home Position
0	0	0	0	0	0	0	1	ID	S	Set Cursor Move Direction
0	0	0	0	0	0	1	D	C	B	Enable Display / Cursor
0	0	0	0	0	1	SC	RL	X	X	Move Cursor / Shift Display
0	0	0	0	1	DL	N	F	X	X	Set Interface Length
0	0	0	1	A	A	A	A	A	A	Move Cursor into CGRAM
0	0	1	A	A	A	A	A	A	A	Move Cursor to Display
0	1	BF	X	X	X	X	X	X	X	Poll the "Busy Flag"
1	0	D	D	D	D	D	D	D	D	Write a Char. to Display at Current Pos.
1	1	D	D	D	D	D	D	D	D	Read Char. from Display at Current Pos.

Set Cursor Move Direction

ID – Set to '1' to increment the cursor after each data byte written to the display

S – Set to '1' to shift the display when a data byte is written

Enable Display / Cursor

D – Turn Display ON (1) or OFF (0)

C – Turn Cursor ON (1) or OFF (0)

B – Set Cursor Blink ON (1) or OFF (0)

Move Cursor / Shift Display

SC – Turn Shift Display ON (1) or OFF (0)

RL – Direction of shift – Right (1) or Left (0)

Set Interface Length

DL – Set Data Interface Length – Byte Mode (1) or Nibble Mode (0)

N – Number of Display Lines – One (0) or Two (1)

F – Character Font – 5x10 (1) or 5x7 (0)

Poll the 'Busy Flag'

BF – This bit is set while the LCD is processing

Move Cursor to CGRAM / Display

A – Address bits

Read / Write ASCII Data to Display

D – Data bits (in ASCII format)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Subroutines and the 68HC12

We will now try to create a set of reusable functions that will allow us to control the LCD screen. To do so, we need to examine how subroutine calls are handled for the 68HC12. To begin, we will start with the basic commands LCALL and RET.

For the 68HC12, the JSR instruction (**J**ump to **S**ub**R**outine) behaves like a **long jump**, meaning that it takes a full 16-bit jump address as its parameter. We can use it like a jump instruction by including a label, allowing the assembler to choose the appropriate addressing mode:

```
MAIN:   ; ... Prepare parameters to send ...
        JSR SUBR
        ; ... Get return values ...

SUBR:   ; Subroutine code goes here
        ; ... ... ...
        RTS
```

Parameter Passing Methods

From the lecture, we know that we have three basic options to pass parameters to a subroutine in assembly. The most basic option is to plan ahead and store the **parameters in registers directly**:

```
MAIN:   LDAA <Parameter 1>
        LDAB <Parameter 2>
        ; ... etc. ...
        JSR SUBR
        ; ... Get return values ...

SUBR:   ; Subroutine code goes here – use A and B directly
        ; ... ... ...
        RTS
```

The major drawback for this method, as we discussed in class, is that registers are the most limited resource. If the subroutine requires large operands, operands that do not fit in registers, or a large number of parameters, then this is an inefficient or impossible solution. For example, with the 68HC12, if we have more than **four** parameters we would insufficient room in registers only for them (A, B, X, and Y – no room for a fifth).

The next simplest option is to pass parameters by using **pre-defined memory locations**. This allows us to define a much larger number of parameters to pass. It also allows us to return multiple values easily, too. Notice that by using pre-defined memory locations to store subroutine parameters and return results we are essentially creating a **global variable** for every parameter/result of every subroutine. This directly trades memory space for simplicity, but is an acceptable solution **provided there is no recursion**:

```
MAIN:       LDAA #<Parameter 1 Value>
            STAA PARAM1
            LDAA #<Parameter 2 Value>
            STAA PARAM2
            ; ... etc. ...
            JSR SUBR
            ; ... Get return values ...

SUBR:       ; Subroutine code goes here
            LDAA PARAM1
            ; The above code makes a local copy of parameter 1 in A
            ; ... ... ...
            RTS

            ; The "DS (X)" directive allocates X bytes of blank space
PARAM1      DS 1
PARAM2      DS 1
```

In the above code, note that we are limited to one **copy** of the subroutine parameters. Any **recursion** (a second call to SUBR while inside SUBR already) would require us to **overwrite** the old parameter values in PARAM1/PARAM2.

The above two methods are quite useful for creating hand-written code. However, they are difficult to implement for a **compiler.** Recall that a compiler is a program that translates high-level code (like C/C++) into machine code. The reason it is difficult to implement is that the compiler has no concept of **code meaning.** It cannot easily determine when it is safe to use registers or pre-define memory locations because it does not know what you intend the code to actually do. In cases where a determination cannot be made, our last and most general option is to use the stack. This, in combination with registers for simpler methods, will be our preferred approach. Consider the code on the following page, which passes two parameters and returns one value.

When reading the code, recall that the 68HC12 has a stack which grows **downward** in memory. In other words, when we PUSH an item onto the stack, SP is decremented.

```
                LDS <Stack start>       ; Initialize the stack pointer so we can use it
MAIN:           LDAA #22                ; First parameter = 22
                PSHA                    ; Push accumulator A, saving parameter 1 to the stack
                LDAA #44                ; Second parameter = 44
                PSHA                    ; Push accumulator A, saving parameter 2 to the stack
                DES                     ; Decrement SP, leaving a blank space on the stack for return value
                JSR SUBR

                PULA                    ; POP accumulator A
                INS                     ; Remove the parameters by incrementing SP; note that the data
                INS                     ; is not actually overwritten until another PUSH instruction occurs
END:            JMP END                 ; End of program

SUBR:           PSHD                    ; Protect registers – PUSH Accumulators A and B (recall D = A:B)
                PSHX                    ; Save current value in X
                PSHY                    ; Save current value in Y..
                ;PSHC                   Optional – saves the current CCR; useful for interrupt routines
```

<<< Stack contents that follow are taken at this location in code >>>

```
                LDAA 10,SP              ; A = Parameter 1
                ADAA 9,SP               ; A = A + Parameter 2
                STAA 8,SP               ; return A (= Parameter 1 + Parameter 2)

                ;PULC                   Pull (Pop) saves register values
                PULY                    ; Notice that this must be done in exactly the reverse order that
                PULX                    ; registers were saved in, otherwise the register contents will be
                PULD                    ; swapped.

                RTS
```

Stack Contents at Highlighted Code Location:

Memory Address	Contents	Data Width	
<Stack Start>	Parameter 1 = 22	1 byte	SP+10 = "10, SP"
<Stack Start–1>	Parameter 2 = 44	1 byte	SP+9 = "9, SP"
<Stack Start–2>	<Blank Space>	1 byte	SP+8 = "8, SP"
<Stack Start–3>	<Return Address for JSR>	2 bytes	SP+7
<Stack Start–5>	<Backup of A, B>	2 bytes	SP+5
<Stack Start–7>	<Backup of X>	2 bytes	SP+3
<Stack Start–9>	<Backup of Y, **low byte**>	1 byte	SP+1
<Stack Start–10>	<Backup of Y, **high byte**>	1 byte	← SP

There are a few things to notice about the previous stack contents. For instance, notice that on the 68HC12, the SP points to the **last currently occupied space** on the stack. In some other microcontrollers (such as the 8051, for example), SP will point to the **next free space**. Also, notice that we have divided the stored value of Y (a **16-bit** or 2-byte register) into its upper and lower bytes. This is to show that SP specifically points to the last occupied **byte** on the stack. Lastly, we can notice the SP-relative addressing used to access earlier data on the stack. If SP is currently pointing to the upper byte of Y, then Parameter 1 is **always** stored +10 bytes relative to the current SP. This holds true even if we have multiple copies of parameters on the stack; SP+10 will always address the **copy** of Parameter 1 belonging to each call to SUBR.

Compared to the previous code, this is obviously quite long, and it makes extensive use of memory through the stack. There are ways to make the code more efficient – we will examine them in later labs through optimization. There are a few unique issues with the 68HC12 that you may notice, plus some new points:

1. The SP-relative addressing of the parameters does impose some limits on the range of memory (relative to SP) that we can address. Thus, it we still could not pass large arrays **by value** in this manner (it would not be efficient, anyway). These are best passed using pointers, which is the standard method to begin with.
2. We have to initialize the stack pointer to a location in internal ram to avoid over-writing registers and other data. The 68HC12 stack grows **downward** in memory, and must be controlled in size to avoid collision with stored code or variables. Uncontrolled recursion or large parameter sets can cause a **stack overflow** condition.
3. The above structure for a subroutine leaves the microcontroller in an identical state before and after its execution, at least in terms of register contents. When we later implement **interrupt service routines (ISR) –** specialized subroutines that are **automatically** called when an interrupt or asynchronous event occurs – we will keep this structure. The reason we do so is that an interrupt may be called asynchronously at **any time** during the normal execution of other code, meaning that we **cannot assume anything** about the microcontroller state **or modify anything**, either.
4. The above has implications on **calling other subroutines** from within an interrupt service routine / handler. If we use registers to pass values to another subroutine from within an ISR, we are safe provided we use the stack to restore A / B / X / Y / etc. after the ISR is complete. However, we are once again limited in number and size of parameters. Fixed memory locations **generally cannot be used** for subroutines called from inside an ISR; if the ISR is called at an inopportune time, the use of fixed-location variables within the ISR may unintentionally overwrite values already placed in these locations. Thus, we need to be aware of if our subroutines need to be **safe for ISR usage**.

We now have the ability to create truly independent subroutines. We will continue by creating and testing a library of subroutines to control the LCD panel.

Lab – Part 1: Basic Subroutines

Step 1. We will be using the LCD module in later sections of the lab. Whenever you use the LCD, ensure that no external connections are made to Port K, or else it will not function.

Step 2. To better understand the process of parameter passing, two of the parameter-passing methods shown in the introduction are implemented in projects Lab4_1.mcp (by fixed location) and Lab4_2.mcp (by stack). Load each project and step through code in the simulator to see their function. For Lab4_2.mcp in particular, pay close attention to the SP and the stack contents (use the 'view memory' function) – they should match the snapshot shown in the table below the code in the introduction. After stepping through each program, **answer Question1 and Question 2** in the deliverable.

Step 3. We will now try to use the LCD screen. Load the project Lab4_3.mcp, download to the 68HC12, and run. It should display 'A' on the top LCD line, and 'B' on the bottom LCD line. Read through the program and try to identify the LCD instruction codes mentioned in the introduction.

Step 4. Close the previous project and open Lab4_4.mcp. You will write code in this project to extend the previous code by calling the function used to print a single character. The new program will print a string of characters terminated by a '$'. Notice the following lines in the program:

```
STRING1:        FCC "First Name$"
STRING2:        FCC "Last Name$"
```

Replace 'first name' and 'last name' with one of your names, but keep the trailing '$'. Your task will now be to write a subroutine (named PRINT_STRING) that does the following tasks:

1. Takes one parameter as input, stored in X – the address of a string to write to the LCD screen. Your subroutine does not need to return any value.
2. Your subroutine should write the string found at the address parameter onto the LCD screen. You can do so by looping through each character in the string and using the existing subroutine 'DATWRT4.' This subroutine prints the character stored in accumulator A at the current cursor location. It also moves the cursor automatically to the next location. When you encounter the character '$,' your subroutine should stop printing characters and return.

You must also add code to the MAIN section to actually call your routine twice – once in to print your first name, and once to print your last name. Locations for each are clearly marked in the code. Write your completed code as **Question 3** on the deliverable, and also answer **Question 4.**

You now also have a reusable set of assembly routines that can write simple text to the LCD screen – you can use them for debug output in later labs, or as part of your term project. You may also want to write additional subroutines to do useful tasks, such as clearing the LCD, or setting the cursor to a particular position.

Lab – Part 2: C Code

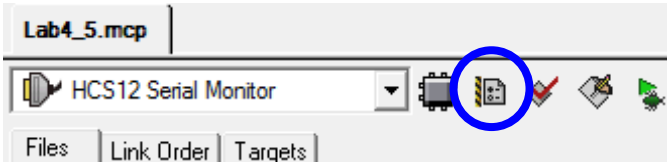
Step 5. We will now attempt our first high-level program using the compiler (which comes as a part of CodeWarrior). To begin, load the project 'Lab4_5.mcp.' This project contains a skeleton code listing, along with the files 'basicLCD.c' and 'basicLCD.h.' These files implement the same basic LCD functions we just saw in assembly. Take some time to look through the C code in "BasicLCD.c" and compare it to the assembly language routines we used before.

Step 6. Compile the code using the Project → Make.

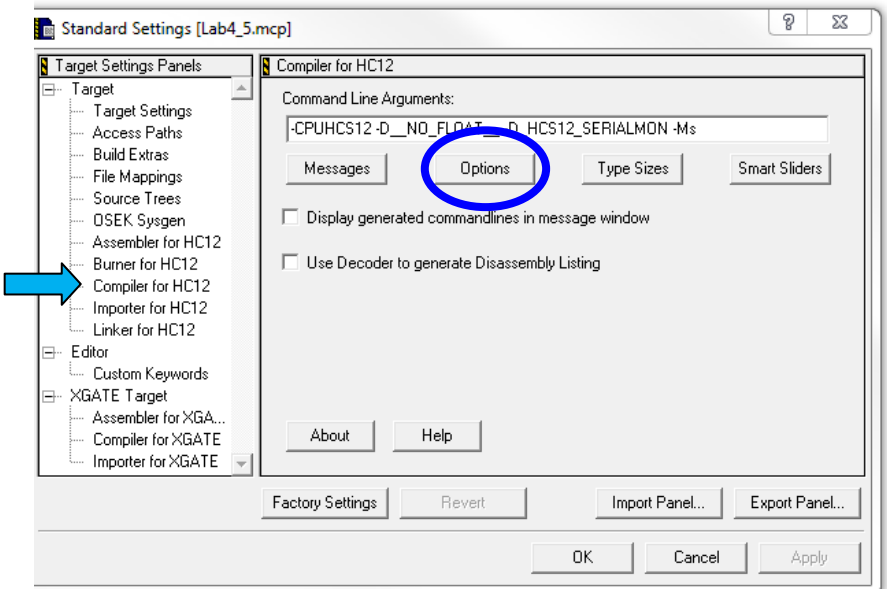
Step 7. Ensure that 'main.c' is open. From the Project menu, select the option 'Disassemble.' This will display the **compiled** version of the high-level 'main.c.'

In particular, you can see that this listing contains a disassembled copy of the machine code produced by the compiler for the routines 'void main( )' and 'void writeLCD(char *).' Search through the code to find the section for main( ) to find the two calls to writeLCD. Examine the assembly code to answer **Question 5.**

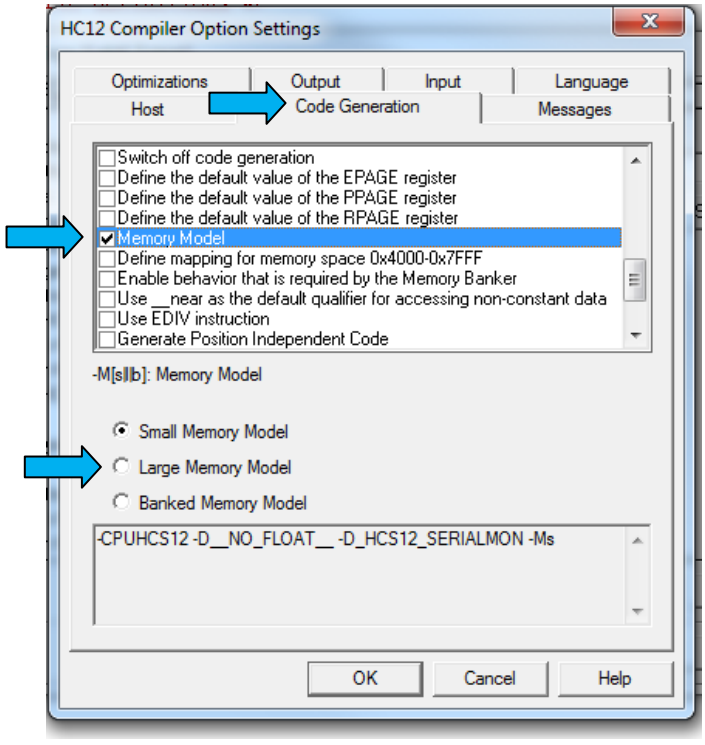
Step 8. Close the disassembly listing. The project is currently compiled using a 'small' memory model, which favours the use of registers and accumulators. This results in faster code in cases where register is **actually** minimal. When the compiler runs out of free registers it uses the stack, which quickly becomes inefficient. To change this mode, click the "Standard Settings…" button on the project tab:



In the list on the left, select “Compiler for HC12”. On the right, click ‘Options’:



In the window that opens, select the ‘Code Generation’ tab. Scroll down and select ‘Memory Model’ in the list. Select the option for ‘Large Memory Model.’



Press OK on this dialog, then OK again. A message may appear asking for “Target ‘Standard’ must be rebuilt.’ Press OK on this dialog as well. Re-open the file ‘main.c’ and select Project→Disassemble again.

The above process forces the compiler to use a different memory model (large). This changes the way the compiler stores local variables, global variables, etc., and the way in which it passes parameters. The large memory model assumes significant memory space is available. You may notice the 'PAGE(…)' notation which appears around some labels, and the use of CALL instead of JSR. This mode allows **paging**, a method used to access more memory than 16-bit direct addresses allow. You may ignore these notations.

For this mode, while it will still favour register-passing when the result is expected to be fast, it also allows the use of fixed-memory-location variables when useful. Note that the default memory model is actually not small or large – it is **banked.** You may recall the concept of register banking being mentioned briefly in the early lectures. Some microcontrollers have multiple 'copies' of the registers (called **register banks**). By switching back and forth between different sets, the compiler can more often make use of register-passing for parameters, which is fast and short. Otherwise, the banked model creates code **most like** the **large** model.

Answer **Question 6 and Question 7**.

Step 9. Use the previous procedure to change the memory model from **large** to **banked**. Change the strings in the code to one of your names, recompile the code, and then download into the 68HC12 (Debug/F5). Verify that the behaviour is the same as with the assembly language program.

***** End of Laboratory 4 *****

After the lab, it is strongly suggested to experiment more with writing C programs using the LCD. Some suggestions for things you can try:

- From the LCD specification, figure out the command to clear the screen and write a C function that does this.
- Write a function that moves the LCD cursor down one line.
- Modify the writeLCD function to accept newline ('\n') characters, and/or to write to a specific position on the screen.

MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 4

Deliverable

Department of Mechanical and Industrial Engineering

University of Toronto

Last Name, First Name	Student Number

1. Using Lab4_2.mcp, write out the numerical values of the stack contents at location marked in the given code (this is the same location shown in the introduction). Include addresses for each value as well. What is the value of SP?

2. Write and test code to call the subroutine a second time (immediately after the "PULA / INS / INS" code in MAIN). For parameter one, use the **returned value** from the first call to SUBR. For parameter two, use a constant value of your choice. Write your code here:

3. Record your calling code from MAIN and from your subroutine. You may attach a printout instead.

4. Would you consider your subroutine safe? Is there any input or use that might cause unexpected behavior?

5. How is the parameter for the string passed to the writeLCD function? How are the other local variables (such as 'int i' in writeLCD) stored?

6. Now, how is the parameter passed to the writeLCD function? How are the other local variables (such as 'int i') stored? Were there any changes? Speculate on why or why not.

7. Of the two compiled versions of the code, was one (significantly) shorter? Which do you think will be faster, and why?