EE319K Laborartory Manual

Univ of Texas at Austin
Bard, Daniels, Welker
Spring 2008

Table of Contents

LAB 1. A DIGITAL LOCK 3

LAB 2 ANALYSIS OF MICROCONTROLLER EXECUTION 5

LAB 3. MINIMALLY INTRUSIVE DEBUGGING METHODS 7

LAB 4. TRAFFIC LIGHT CONTROLLER 15

LAB 5. LCD DEVICE DRIVER 19

LAB 6. REAL-TIME POSITION MEASUREMENT SYSTEM 25

LAB 7 DISTRIBUTED DATA ACQUISITION SYSTEM 29

LAB 8. MUSIC GENERATION USING A DIGITAL TO ANALOG CONVERTER 33

LABS 9 AND 10. TEXAS ROBOTS 1.7 (FOR THE LASTEST INFORMATION CHECK THE WEB SITE) 41

HOW TO DEVELOP ASSEMBLY PROGRAMS USING METROWERKS/TECH ARTS BOARD 51

HOW TO DEVELOP C PROGRAMS METROWERKS/TECH ARTS 9S12DP512 BOARD 55

68HC12 Op Code Map - Page 2

03-13-06 RGD

0	. 1	. 2	. 3	. 4 - F	
0	MOVW:IMM-IND	IDIV	LBRA		0
1	MOVW:EXT-IND	FDIV	LBRN		1
2	MOVW:IND-IND	EMACS	LBHI		2
3	MOVW:IMM-EXT	EMULS	LBLS		3
4	MOVW:EXT-EXT	EDIVS	LBCC		4
5	MOVW:IND-EXT	IDIVS	LBCS		5
6	ABA	SBA	LBNE		6
7	DAA	CBA	LBEQ		7
8	MOVB:IMM-IND	MAXA	LBVC		8
9	MOVB:EXT-IND	MINA	LBVS		9
A	MOVB:IND-IND	EMAXD	LBPL	REV	A
B	MOVB:IMM-EXT	EMIND	LBMI	REVW	B
C	MOVB:EXT-EXT	MAXM	LBGE	WAV	C
D	MOVB:IND-EXT	MINM	LBLT	TBL	D
E	TAB	EMAXM	LBGT	STOP	E
F	TBA	EMINM	LBLE	ETBL	F
	0	. 1	. 2	. 3 . 4 - F	

Trap

Hexadecimal to ASCII Conversion

Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII
$00	NUL	$20	SP *space*	$40	@	$60	*'grave*
$01	SOH	$21	!	$41	A	$61	a
$02	STX	$22	" *quote*	$42	B	$62	b
$03	ETX	$23	#	$43	C	$63	c
$04	EOT	$24	$	$44	D	$64	d
$05	ENQ	$25	%	$45	E	$65	e
$06	ACK	$26	&	$46	F	$66	f
$07	BEL *beep*	$27	' *apost.*	$47	G	$67	g
$08	BS *back sp*	$28	(	$48	H	$68	h
$09	HT *tab*	$29	)	$49	I	$69	i
$0A	LF *linefeed*	$2A	*	$4A	J	$6A	j
$0B	VT	$2B	+	$4B	K	$6B	k
$0C	FF	$2C	, *comma*	$4C	L	$6C	l
$0D	CR *return*	$2D	- *dash*	$4D	M	$6D	m
$0E	SO	$2E	. *period*	$4E	N	$6E	n
$0F	SI	$2F	/	$4F	O	$6F	o
$10	DLE	$30	0	$50	P	$70	p
$11	DC1	$31	1	$51	Q	$71	q
$12	DC2	$32	2	$52	R	$72	r
$13	DC3	$33	3	$53	S	$73	s
$14	DC4	$34	4	$54	T	$74	t
$15	NAK	$35	5	$55	U	$75	u
$16	SYN	$36	6	$56	V	$76	v
$17	ETB	$37	7	$57	W	$77	w
$18	CAN	$38	8	$58	X	$78	x
$19	EM	$39	9	$59	Y	$79	y
$1A	SUB	$3A	:	$5A	Z	$7A	z
$1B	ESCAPE	$3B	;	$5B	[	$7B	{
$1C	FS	$3C	<	$5C	\	$7C	|
$1D	GS	$3D	=	$5D	]	$7D	}
$1E	RS	$3E	>	$5E	^	$7E	~
$1F	US	$3F	?	$5F	_ *under*	$7F	DEL *delete*

Lab 1. A Digital Lock

Preparation

Read Chapter 1 of the book
Read Section 3.3.2 of the book
Install and run TExaS, execute **Help->GettingStarted**, read up to but not including "Developing C software…"
Download and run the first three lessons at
http://users.ece.utexas.edu/~valvano/Readme.htm

Purpose

The general purpose of this laboratory is to familiarize you with the software development steps using the **TExaS** simulator. The specific device you will create is a digital lock with two binary switch inputs and one LED output. The LED output represents the lock, and the operator will toggle the switches in order to unlock the door. Let **T** be the Boolean variable representing the lock (0 means LED is off and door is locked, 1 means LED is on and door is unlocked). Let **M** and **A** be Boolean variables representing the state of the two switches (0 means the switch is not pressed, and 1 means the switch is pressed). The specific function you will implement is

$$T = M \,\&\, \overline{A}$$

This means the LED will be on if and only if the **M** switch is pressed and the **A** switch is not pressed, as shown in Figure 1.1.

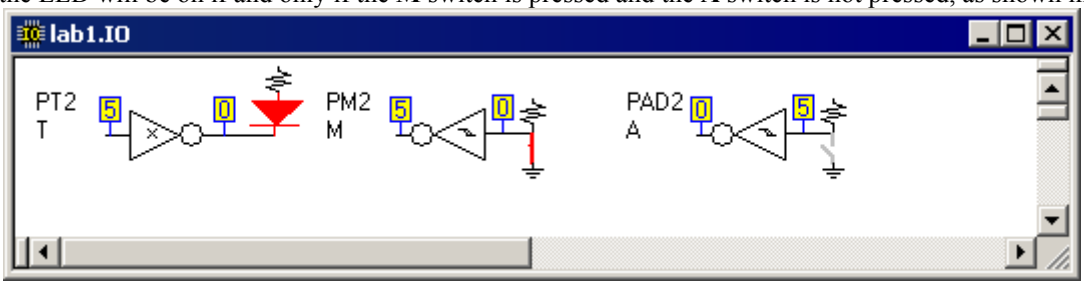


Figure 1.1. TExaS IO window showing the door is unlocked.

Description

Part a) Use the TExaS simulator to create three files. `Lab1.rtf` will contain the assembly source code. `Lab1.uc` will contain the microcomputer configuration. `Lab1.io` will define the external connections, which should be the two switches and one LED as shown in Figure 1.1. In this class we will use the 9S12DP512 microcomputer, which you can specify using the **Mode->Processor** command. You should connect switches to PAD2 (means Port AD0 bit 2) and to PM2 (means Port M bit 2). You should connect an LED to PT2 (means Port T bit 2). The switches should be labeled **M** and **A**, and the LED should be labeled **T**. When **M** switch is "off" or open position, the signal at PM2 will be 0V, which is a logic "0". For this situation, your software will consider **M** to be false. When the **M** switch is "on" or closed position, the signal at PM2 will be +5V, which is a logic "1". In this case, your software will consider **M** to be true. The **A** switch, which is connected to PAD2, will operate in a similar fashion. When your software writes a "1" to PT2, the LED will turn on. Figure 1.1 shows the condition where the LED is on because **A** is not pressed and **M** is pressed.

Part b) You will write assembly code that inputs from PM2 and PAD2, and outputs to PT2. Program 1.1 describes the software algorithm in C. Notice that this algorithm affects all bits in a port, although only one bit is used. In general, this will be unacceptable, and we will learn later how to write code that affects one bit at a time. You can copy and paste the address definitions for ports M, AD0, and T from the **`port12.rtf`** file. In particular, you will need to define **`DDRM DDRT ATD0DIEN PTM PORTAD0`** and **`PTT`**.

```
void main(void){
  ATD0DIEN = 0xFF;  // make Port AD0 digital input
  DDRM = 0x00;      // make Port M an input,  PM2 is M
  DDRT = 0xFF;      // make Port T an output, PT2 is T
  while(1){
    PTT = (~PORTAD0)&PTM; // LED on iff PAD2=0 and PM2=1
  }
}
```

Program 1.1. The first C program to illustrate Lab 1.

The structure of assembly programs in this class is shown as Program 1.2. The opening comments include: file name, overall objectives, hardware connections, specific functions, author name, and date. The **`equ`** pseudo-op is used to define port addresses. Global variables are declared in RAM, and the main program is placed in EEPROM. The 16-bit contents at $FFFE and $FFFF define where the computer will begin execution after a reset vector. This template, shown in Program 1.2, can be found in most example programs for the 9S12C32 or 9S12DP512.

```
;****************** Lab1.RTF ***************
; Program written by: Your Name
; Date Created: 1/22/2008
; Last Modified: 1/22/2008
; Section 1-2pm     TA: Nachiket Kharalkar
; Lab number: 1
; Brief description of the program
; The overall objective of this system is a digital lock
; Hardware connections
;  PM2 is switch input M
;  PAD2 is switch input A
;  PT2 is LED output T (on means unlocked)
; The specific operation of this system
;   unlock if A is not pressed and M is pressed
;I/O port definitions on the 9S12DP512
ATD0DIEN equ $008D  ; ATD Input Enable Mask Register
PTM      equ $0250  ; Port M I/O Register
PORTAD0  equ $008F  ; Port AD I/O Register
PTT      equ $0240  ; Port T I/O Register
DDRM     equ $0252  ; Port M Data Direction Register
DDRT     equ $0242  ; Port T Data Direction Register
      org  $0800   ; RAM
                   ; Global variables (none required for this lab)
      org  $4000   ; flash EEPROM
main
;Software performed once at the beginning
loop
;Software repeated over and over
      bra  loop
      org  $FFFE
      fdb  main      ;Starting address
```

Program 1.2. Assembly language template.

Part c) During the demonstration, you will be asked to run your program to verify proper operation. You should be able to single step your program and explain what your program is doing and why. You need to know how to set and clear breakpoints. You will be asked to look up the meaning of commands like **Mode->FollowPC** using the on-line help. Be prepared to make changes to **`Lab1.io`**, such as changing the names and colors of the switches and LEDs.

Lab 2 Analysis of Microcontroller Execution

Preparation

Read Chapter 2 as a review of EE306
Read Sections 3.1, 3.2, 3.3, and 4.6
Get the CPU12 data book from your instructor or download the electronic version
http://users.ece.utexas.edu/~valvano/EE319K/S12CPUV2.pdf

In preparation for this lab, you should look up these terms in the glossary: accumulator, address bus, arithmetic logic unit (ALU), bus, bus interface unit (BIU), control bus, control unit (CU), data bus, effective address register (EAR), memory-mapped I/O, opcode, operand, program counter (PC), and registers.

Purpose

The educational objectives of this lab are to

1) learn the difference between source code and object code
2) understand how the machine uses the IR and EAR while executing
3) know immediate, extended, indexed and PC-relative addressing
4) study how the computer executes software

In order to get a good grade in Lab 2, we suggest you use this example to practice determining bus cycles. You do not have to do this part, it will not be turned in, and it will not be graded. Start **TExaS**, and create a new Microcomputer file.

Step 1. Execute the **Mode->Processor…** command and select the MC9S12DP512. Save this document as **`Lab2.uc`**.

Step 2. Download the file **`Lab2.rtf`** from the class website, shown below. Assemble this program by executing the **Assemble->Assemble** command.

```
$3800                       org  $3800
$3800                 Sum   rmb  2          ;16-bit signed result
$0003                 SIZE  equ  3
$4000                       org  $4000
$4000 CE401A          main  ldx  #Array     ;pointer to array
$4003 CD0000                ldy  #0         ;Sum=0
$4006 7D3800                sty  Sum
$4009 A630            loop  ldaa 1,x+       ;get data from array
$400B B704                  sex  a,d        ;promote to 16-bits
$400D F33800                addd Sum
$4010 7C3800                std  Sum        ;Sum=Sum+data
$4013 8E401D                cpx  #Array+SIZE
$4016 26F1                  bne  loop       ;done?
$4018 183E            done  stop
$401A F414D8          Array fcb  -12,20,-40 ;array of data
$FFFE                       org  $FFFE      ;reset vector
$FFFE 4000                  fdb  main
```

Assembly program used in Lab 2 part a).

Step 3. For each op code, first determine the addressing mode it uses. The **`sex a,d`** is inherent mode, because it operates on the registers without accessing memory. **`org`** **`rmb`** **`equ`** **`fcb`** and **`fdb`** are pseudo ops and thus do not have addressing modes. The instructions that access **`Sum`** will use extended addressing, rather than direct addressing, because the address of **`Sum`** ($3800) is outside the $0000 to $00FF range of direct addressing.

Step 4. Execute this program by hand (using paper and pencil) up to but not including the **`stop`** instruction. For each instruction show the memory cycles generated and the values of any registers that change. Show the simplified cycles as described in the book. This program will execute 22 instructions, resulting in RegD being the 16-bit result -32. You do not need to show free cycles or changes to the CCR, but do include changes to the other registers including the **IR** and **EAR**. The **IR** is set after reading the op code, and the **EAR** is set before reading/writing memory with direct, extended, or indexed addressing modes. Some instructions, like **`leax leay leas`**, use indexed addressing mode and set the **EAR**, but do not access memory. Other than these exceptions, the **EAR** holds the address when reading data from memory or writing data to memory. E.g., the first instruction is

Instruction: `ldx #$401F`

R/W	Addr	Data	Changes
R	$4000	$CE	PC=$4001, IR=$CE
R	$4001	$40	PC=$4002
R	$4002	$1A	PC=$4003, X=$401A

The second instruction is

Instruction: `ldy #0`

R/W	Addr	Data	Changes
R	$4003	$CD	PC=$4004, IR=$CD
R	$4004	$00	PC=$4005
R	$4005	$00	PC=$4006, Y=$0000

The third instruction is

Instruction: `sty Sum`

R/W	Addr	Data	Changes
R	$4006	$7D	PC=$4007, IR=$7D
R	$4007	$38	PC=$4008
R	$4008	$00	PC=$4009, EAR=$3800
W	$3800	$00	
W	$3801	$00	

Step 5. Activate the **FollowPC CycleView InstructionView** and **LogRecord** modes using the commands in the Mode menu. Single step the program using **TExaS** up to an including the **`stop`** instruction. Verify the answers you gave for Step 4.

To do) Come to lab during your regularly scheduled lab time. At that time, you will be given an assembly listing and some blank pages containing boxes like the example at

http://users.ece.utexas.edu/~valvano/EE319K/Lab2InstructionSheet.pdf

You will be asked to determine the bus cycles for this program, like step 4 above. This program will be similar to the practice program above, but shorter in length. For each instruction you may or may not need all 5 entries. Your Lab2 grade will be based entirely on your performance on this Lab 2 quiz. For example, see

http://users.ece.utexas.edu/~valvano/EE319K/Lab2a.pdf

You will have access to some pages of the CPU12 manual (like the following), but not have access to the entire CPU12 manual, and you will not have access to **TExaS** itself.

SUBD Subtract Double Accumulator SUBD

Operation: (A : B) – (M : M + 1) ⇒ A : B

Description: Subtracts the content of memory location M : M + 1 from the content of double accumulator D and places the result in D.

Source Form	Address Mode	Object Code	HCS12 Access Detail
SUBD #opr16i	IMM	83 jj kk	PO
SUBD opr8a	DIR	93 dd	RPf
SUBD opr16a	EXT	B3 hh ll	RPO
SUBD oprx0_xysp	IDX	A3 xb	RPf
SUBD oprx9,xyssp	IDX1	A3 xb ff	RPO
SUBD oprx16,xysp	IDX2	A3 xb ee ff	fRPP

Lab 3. Minimally Intrusive Debugging Methods

Preparation

Read Chapter 4 as a review of EE306

Read Sections 3.4, 3.5, 4.3, 6.2, 6.3, 7.1, 7.3, 7.4, 7.5, and 7.6

This lab has these major objectives:

- Interfacing LEDs and switches to the microcontroller;
- Simple use of a for loop for creating time delays;
- Development of debugging tools appropriate for the real 9S12.

The basic approach to this lab will be to first develop and debug your system using the simulator. During this phase of the project you will run with a short time delay. After the software is debugged, you will build your hardware and run your software on the real 9S12. During this phase of the project you will run with time delays long enough so you will be able to see the LED flash (slower than 8 Hz).

There is a free design tool from ExpressPCB that we will be using in EE319K, EE345L and EE345M. To download this tool go to www.expresspcb.com. When installed, there will be two applications

ExpressSCH will be used to draw electrical circuits, e.g., Lab3.sch, Lab4.sch

ExpressPCB will be used to design PCB boards (not used until EE345L)

System Requirements

You will first design a system, and then add debugging instruments to prove the system is functioning properly. The system has one input switch and one output LED. The basic function of the system is to respond to the input switch, causing certain output patterns on the LED. Figure 3.1 shows that the switch is in positive logic. This means the **PT3** signal will be 0 (low, 0V) if the switch is not pressed, and the **PT3** signal will be 1 (high, +5V) if the switch is pressed. Overall functionality of this system is described in the following rules.

The system starts with the LED off (make **PT2** =0).

The system will return to the off state if the switch is not pressed (**PT3** is 0).

If the switch is pressed (**PT3** is 1), then the LED will flash on and off at about 4 Hz (any value from 1 to 8 Hz is ok).

One possible circuit diagram for the LED output and switch input is shown in Figure 3.1. You will attach this switch and LED to your protoboard (the white piece with all the holes), and interface them to your 9S12. During the first phase of this lab, you will simulate these hardware circuits in TExaS using positive logic mode for the switch and LED.

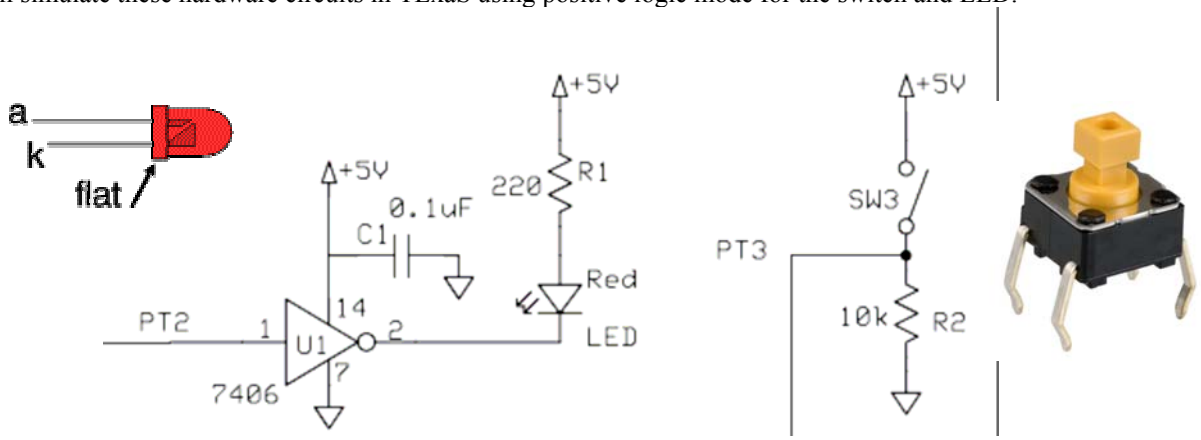


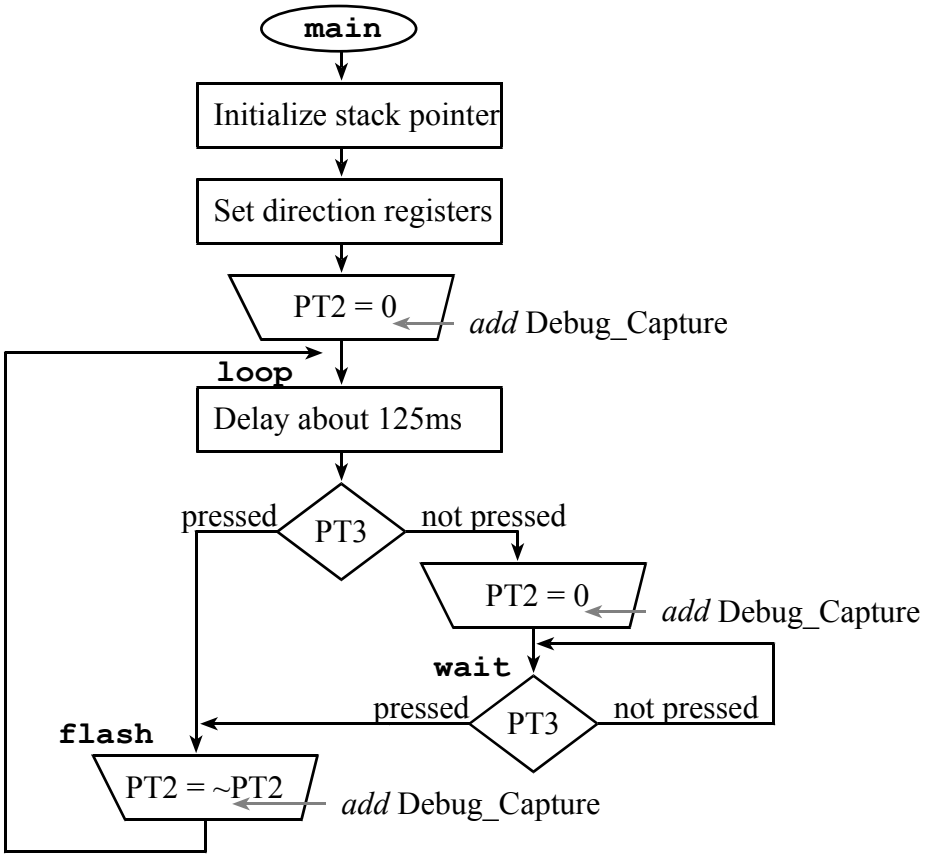
Figure 3.1. Hardware circuit diagram.

When visualizing software running in real-time on an actual microcomputer, it is important use minimally intrusive debugging tools. The objective of this lab is to develop debugging methods that do not depend on the simulator. During the first phase of this lab, you will develop and test your program and debugging instruments on the **TExaS** simulator. In particular, you will write debugging instruments to record input and output information as your system runs in real time. This software dump should store data into an array while it is running, and the information will be viewed at a later time. Software dumps are an effective technique when debugging software on an actual microcomputer. During the second phase of this lab, you will run your system on the real 9S12 with and without your debugging instruments.

Procedure
You will use the **TExaS** simulator to develop and test your debugging instructions, and then you will copy and paste your solutions into a Metrowerks program for running on the real 9S12.

Part a) Write a main program that implements the input/output system. The basic steps for the main program are as follows

Initialize the stack pointer
Enable interrupts for the Metrowerks debugger, **`cli`**
Set the direction register so **PT3** is an input and **PT2** is an output
Set **PT2** so the LED is off
`loop` delay about 125ms (any delay from 60 to 500 ms is OK)
read the switch and go to **`flash`** if the switch is pressed
Set **PT2** so the LED is off
`wait` read the switch and go to **`wait`** if the switch is not pressed
`flash` toggle the LED (if on turn it off, if off turn it on)
go to **`loop`**
	`DDRT &= ~0x08; // PT3 input`
	`DDRT |= 0x04; // PT2 output`
	`PTT &= ~0x04; // PT2 off`
	`while(1){`
	`  Delay(); // you write this`
	`  if((PTT&0x08)==0){`
	`  PTT &= ~0x04; // PT2 off`
	`  while((PTT&0x08)==0){};`
	`  }`
	`  PTT = PTT^0x04; // toggle`
	`}`



To implement the 125ms delay
Set a 16-bit register to a large number, then count it down to zero
Add multiple **`nop`** instructions so the loop takes at least 16 cycles to complete
E.g., 62500(loops)*16(cycles/loop)*125(ns/cycle)=125ms (in RUN mode)
The 9S12 executes 3 times faster in LOAD mode (24 MHz) than RUN mode (8 MHz). We do not care whether your program delays 125ms in RUN mode or in LOAD mode, as long as you understand that there is a difference in execution speed.

Part b) Write two debugging subroutines that implement a **dump** instrument. This is called *functional debugging* because you are capturing input/output data of the system, without information specifying when the input/output was collected. The first subroutine (**`Debug_Init`**) initializes your debugging system. The initialization should initialize a 100-byte array (start

it at $3880), initializing pointers and/or counters as needed. The second subroutine (**`Debug_Capture`**) that saves one data-point (**PT3** input data, and **PT2** output data) in the array. Since there are only two bits to save, pack the information into one 8-bit value for storage and ease of visualization. For example, if

Input (PT3)	Output (PT2)	save data
0	0	000**0**,000**0**$_2$, or $00
0	1	000**0**,000**1**$_2$, or $01
1	0	000**1**,000**0**$_2$, or $10
1	1	000**1**,000**1**$_2$, or $11

In this way, you will be able to visualize the entire array in an efficient manner. Place a call to **`Debug_Init`** at the beginning of the system, and a call to **`Debug_Capture`** just after each time you output to **`PTT`** (there will be 3 or 4 places where your software writes to **`PTT`**). Within **TExaS** you can observe the debugging array using a **`Stack`** window. After you have debugged your code make a printout of the instrumented software system.

The basic steps involved in designing the data structures for this debugging instrument are as follows

Allocate a 100-byte buffer starting at address $3880

Allocate a 16-bit pointer, which will point to the place to save the next measurement

The basic steps involved in designing **`Debug_Init`** are as follows

Set all entries of the 100-byte buffer to $FF (meaning no data yet saved)

Initialize the 16-bit pointer to the beginning of the buffer

The basic steps involved in designing **`Debug_Capture`** are as follows

Return immediately if the buffer is full (pointer past the end of the buffer)

Read **`PTT`** `data = PTT`

Mask capturing just bits 3,2 `data = ((data&$08)<<1)+((data&$04)>>2)`

Dump debugging information into buffer `(*pt) = data`

Increment pointer to next address `pt = pt+1`

Both routines should save and restore registers that it modifies (except CCR), so that the original program is not affected by the execution of the debugging instruments. The temporary variable **`data`** may be implemented in a register. However, the 100-byte buffer and the 16-bit pointer, **`pt`**, should be permanently allocated in global RAM.

Part c) By counting cycles in the listing file, estimate the execution time of the **`Debug_Capture`** subroutine. Assuming the actual E clock speed, convert the number of cycles to time. This time will be a quantitative measure of the intrusiveness of your debugging instrument. Either hand-write it on your printout or type it in as comments to your program.

Part d) Engineers must be able to read datasheets during the design, implementation and debugging phases of their projects. During the design phase, datasheets allow us to evaluate alternatives, selecting devices that balance cost, package size, power, and performance. For example, we could have used other IC chips like the 7405, 74LS05, or 74HC05 to interface the LED to the 9S12. In particular, we chose the 7406 because it has a large output current (I_{OL} = 40 mA), 6 drivers, and is very inexpensive (53¢). During the implementation phase, the datasheet helps us identify which pins are which. During the debugging phase, the datasheet specifies input/output parameters which we can test. Download the 7406, LED, and switch datasheets from the web

http://users.ece.utexas.edu/~valvano/EE319K/SN7406.pdf
http://users.ece.utexas.edu/~valvano/EE319K/LED_red.pdf
http://users.ece.utexas.edu/~valvano/EE319K/B3F-switch.pdf

and find in the datasheet for the 7406 the two pictures as shown in Figure 3.2. Next, hold your actual 7406 chip and identify the locations of pins 1-14. Find in the datasheet the specification that says the output low voltage (V_{OL}) will be 0.4V when the output low current (I_{OL}) is 16 mA (this is close to the operating point we will be using for the LED interface).

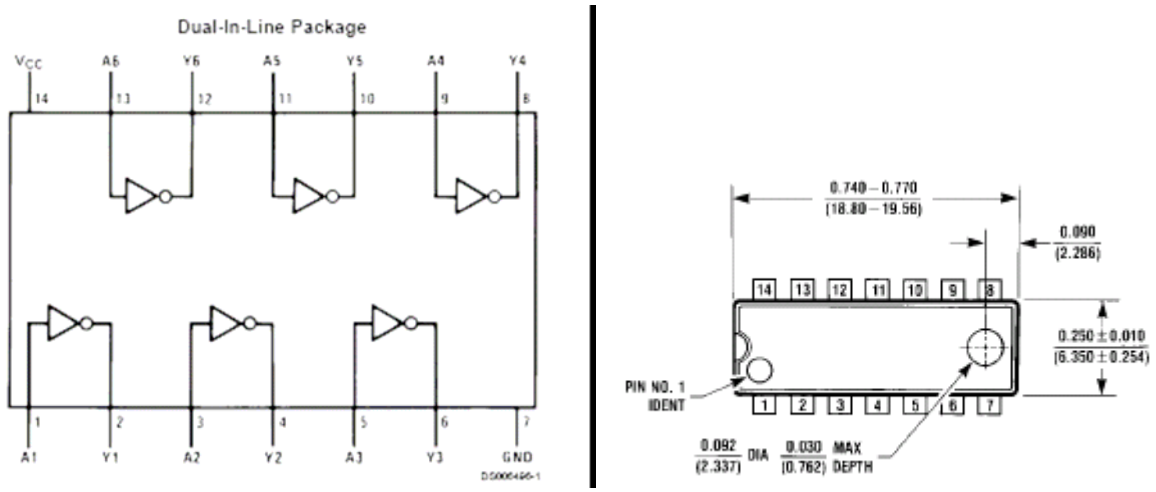


Figure 3.2. Connection diagram and physical package diagram for the 7406.

Similarly, hold an LED and identify which pin is the anode and which is the cathode. Find on the switch datasheet the two pictures of the B3F-1052 switch as shown in Figure 3.3. Next, hold an actual switch and identify the locations of pins 1-4.

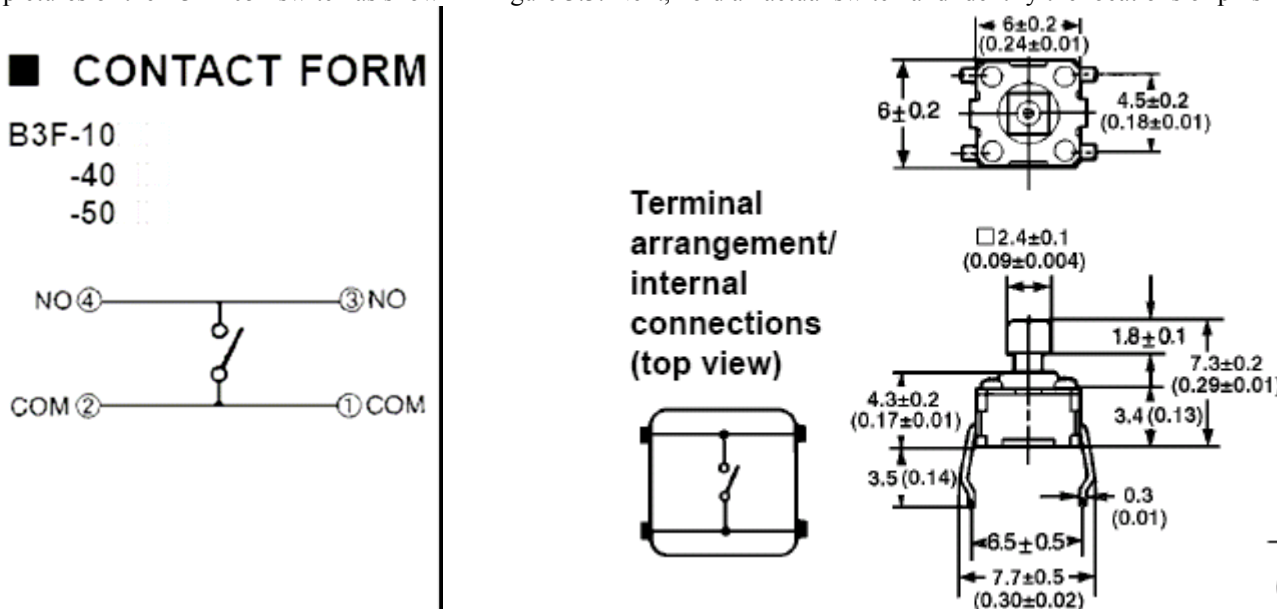


Figure 3.3. Connection diagram and physical package diagram for the B3F-1052 switch.

To build circuits, we'll use a solderless breadboard, also referred as a protoboard. The holes in the protoboard are internally connected in a systematic manner, as shown in Figure 3.4. The long columns of holes along the right and left of the protoboard are electrically connected. Some protoboards like the one in Figure 3.4 have four long columns (two on each side), while others have just two long columns (one on each side). We will connect one column to +5V and another column to ground, and we refer to them as power buses. In the middle of the protoboard, you'll find two groups of holes placed in a 0.1 inch grid. Each adjacent row of five pins is electrically connected. We usually insert components into these holes. IC chips are placed on the protoboard, such that the two rows of pins strattle the center valley. The 9S12 module also is inserted into the protoboard in such a way that the two rows of pins are across the center. The piece of paper from the kit or from Figure 3.7 should be trimmed and placed between the 9S12 and the protoboard, making it easy to identify the 9S12 pins. For example, assume we to connect 9S12 PT2 output to the 7406 input pin 1. First, cut a 24 guage solid wire about a 0.5 in longer than the distance between PT2 and pin 1 of the 7406. Next, strip about 0.25 in off each end. Place one end of the wire in one of the four remainding holes of the PT2 row and the other end in one of the four remainding of the 7406 pin 1.

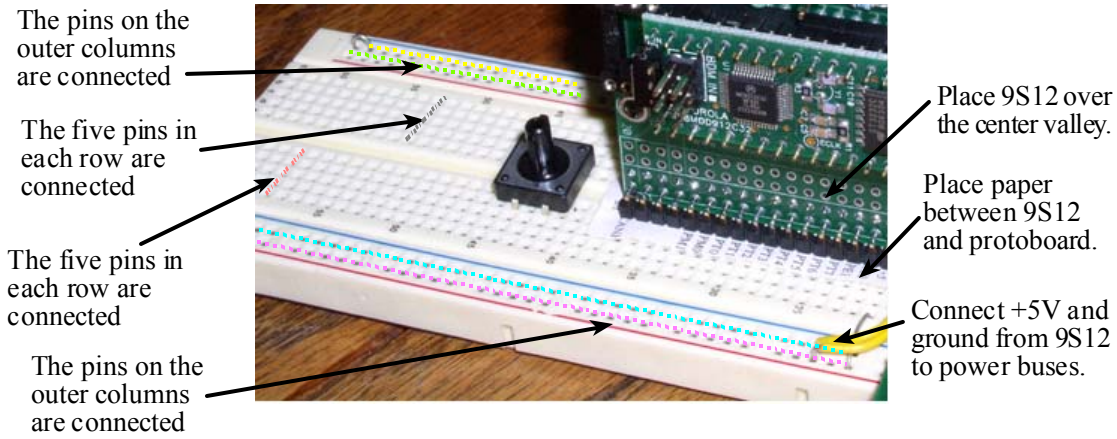


Figure 3.4. The dotted lines in this figure illustrate the pattern of which pins are internally connected.

Part e) After the software has been debugged on the simulator, you will implement it on the real board. **Lab3.sch** is a starter file you should use to draw your hardware circuit diagram (like Figure 3.1). The first step is to interface a push button switch. You can implement positive logic or negative logic switches, as shown in Figure 3.5. *Do not place or remove wires on the protoboard while the power is on.*

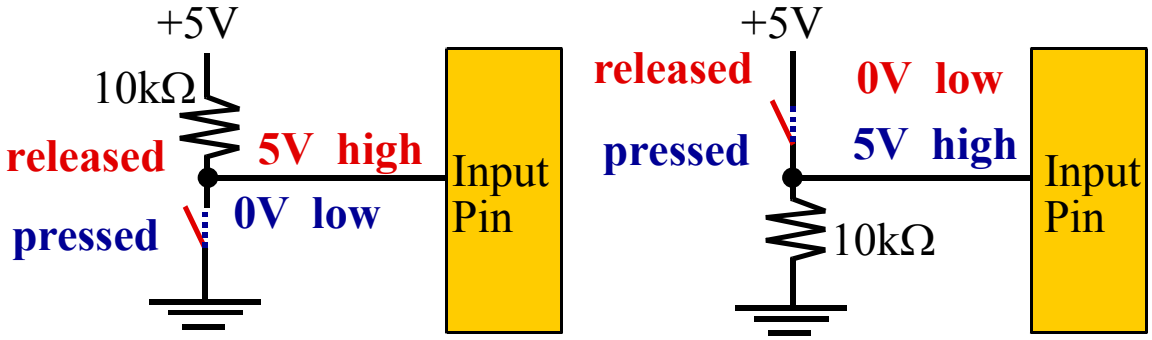


Figure 3.5. Switch interface.

Before connecting the switch to the microcomputer, please take these measurements using your digital multimeter. The input voltage (V_{PT3}) is the signal that will eventually be connected to **PT3**. If you implement a positive logic switch interface, the resistor current will be $V_{PT3}/R2$. If you implement a negative logic switch interface, the resistor current will be $(5-V_{PT3})/R2$. The voltages should be near +5 or near 0V and the currents should be less than 1 mA.

Parameter	Value	Units	Conditions
Resistance of the 10k resistor, R2		ohms	with power off and disconnected from circuit
Input Voltage, V_{PT3}		volts	Powered, but with switch not pressed
Resistor current		mA	Powered, but switch not pressed Negative logic: $(5- V_{PT3})/R2$ Positive logic: $V_{PT3}/R2$
Input Voltage, V_{PT3}		volts	Powered and with switch pressed
Resistor current		mA	Powered and switch pressed Negative logic: $(5- V_{PT3})/R2$ Positive logic: $V_{PT3}/R2$

Table 3.1. Switch measurements.

Next you can connect the input voltage to **PT3** and use the debugger to observe the input pin to verify the proper operation of the switch interface.

The next step is to build the LED output circuit. LEDs emit light when an electric current passes through them, as shown in Figure 3.6. LEDs have polarity, meaning current must pass from anode to cathode to activate. The anode is labeled **a** or + , and cathode is labeled **k** or **-**. The cathode is the short lead and there may be a slight flat spot on the body of round LEDs. Thus, the anode is the longer lead. LEDs are not usually damaged by heat when soldering. Figure 6.4 in the textbook shows an interface circuit that can be used in this lab. Look up the pin assignments in the 7406 data sheet. Be sure to connect +5V power to pin 14 and ground to pin 7. The capacitor from +5V to ground filters the power line.

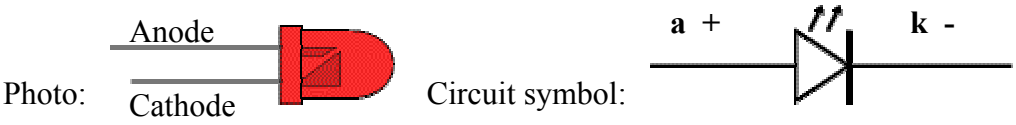


Figure 3.6. LEDs.

You can use the debugger to set the direction register for **PT2** to output. Then, you can set the **PT2** high and low, and measure the three voltages (input to 7406, output from 7406 which is the LED cathode voltage, and the LED anode voltage). When active, the LED voltage should be about 2 V, and the LED current should be about 10 mA.

Parameter	Value	Units	Conditions
Resistance of the 220Ω resistor, R1		ohms	with power off and disconnected from circuit
9S12 Output, V_{PT2} input to 7406		volts	with **PT2** =0
7406 Output, V_{k-} LED k-		volts	with **PT2** =0
LED a+, V_{a+} Bottom side of R1		volts	with **PT2** =0
LED voltage		volts	calculated as $V_{a+} - V_{k-}$
LED current		mA	calculated as $(5- V_{a+})/R1$
9S12 Output, V_{PT2} input to 7406		volts	with **PT2** =1
7406 Output, V_{k-} LED k-		volts	with **PT2** =1
LED a+, V_{a+} Bottom side of R1		volts	with **PT2** =1
LED voltage		volts	calculated as $V_{a+} - V_{k-}$
LED current		mA	calculated as $(5- V_{a+})/R1$

Table 3.2. LED measurements.

Part f) Debug your combined hardware/software system on the actual 9S12 board.

Part g) Run your debugging instrument capturing the sequence of input/outputs as you first touch, then release the switch.

Warning: NEVER INSERT/REMOVE WIRES/CHIPS WHEN THE POWER IS ON.

Deliverables

1) Circuit diagram (with your name and date)
2) Switch measurements (Table 3.1)
3) LED measurements (Table 3.2)
4) Assembly listing of your final program
5) Estimation of the execution time of your debugging instrument **`Debug_Capture`** (part c)
5) Results of the debugging instrument (part g)

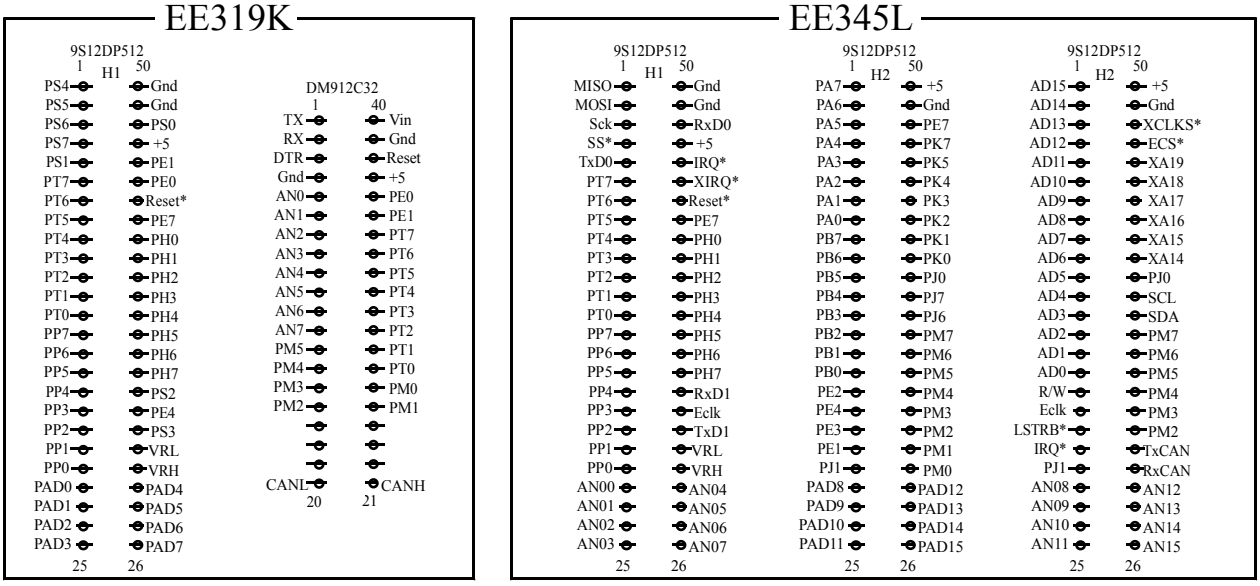


Figure 3.7. Cut and trim a label to place between the 9S12 and the protoboard.

Precautions to avoid damaging your system

1) Touch a grounded object before handling CMOS electronics. Try not to touch any exposed wires.

2) Do not plug or unplug the 9S12 board into a protoboard while the system is powered.

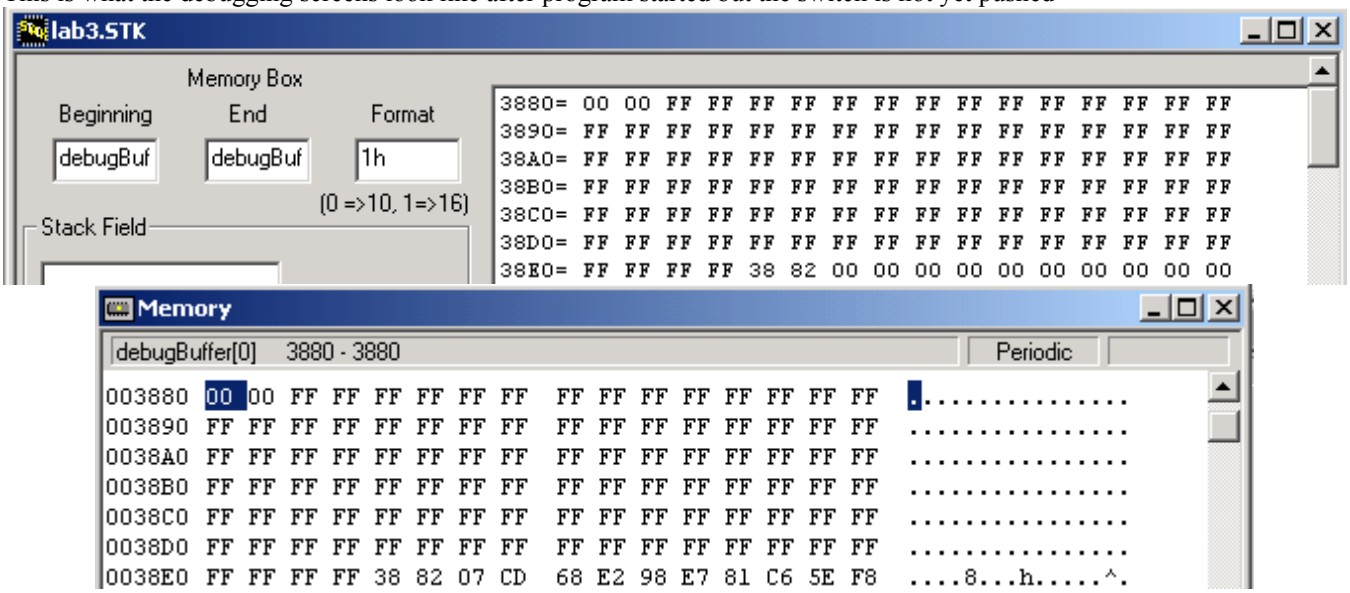
3) Never remove the 9S12C32 CPU module from the docking module. THE PINS ON THE CPU MODULE ARE VERY FRAGILE. On the other hand, the male pins on the docking module have been very robust as long as you limit the twisting forces. To remove the docking module from the protoboard pull straight up (or at least pull up a little at a time on each end.)

4) Use and store the system with the docking module plugged into a protoboard (this will reduce the chances of contacting the metal pins tied directly to the 6812 with either your fingers or stray electrical pulses).

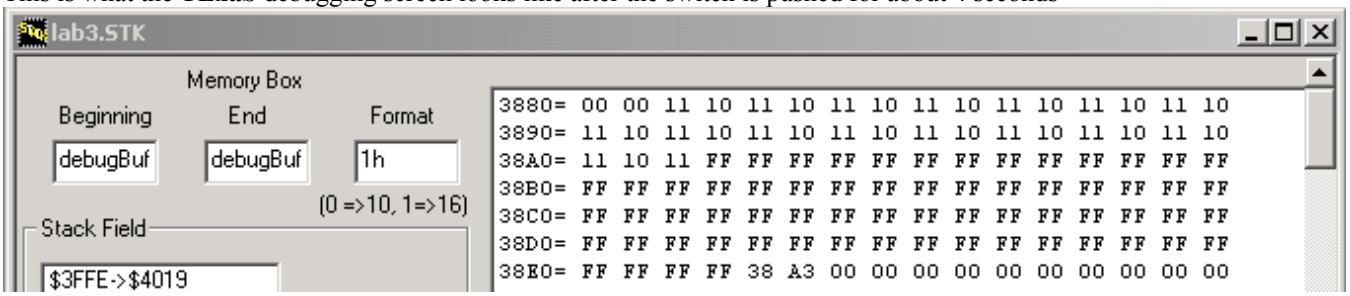
5) Do not use the 9S12 with any external power sources, other than the supplied wall-wart. In particular, avoid connecting signals to the 9S12 that are not within the 0 to +5V range. In particular, voltages less than 0V or greater than +5V will damage the ADC.

6) Do not connect any wires to the 9S12C32 pins labeled Vin, DTR, TX, or RX. These pins contain voltages outside the safe 0 to +5V range. Also do not connect to the Reset pin.

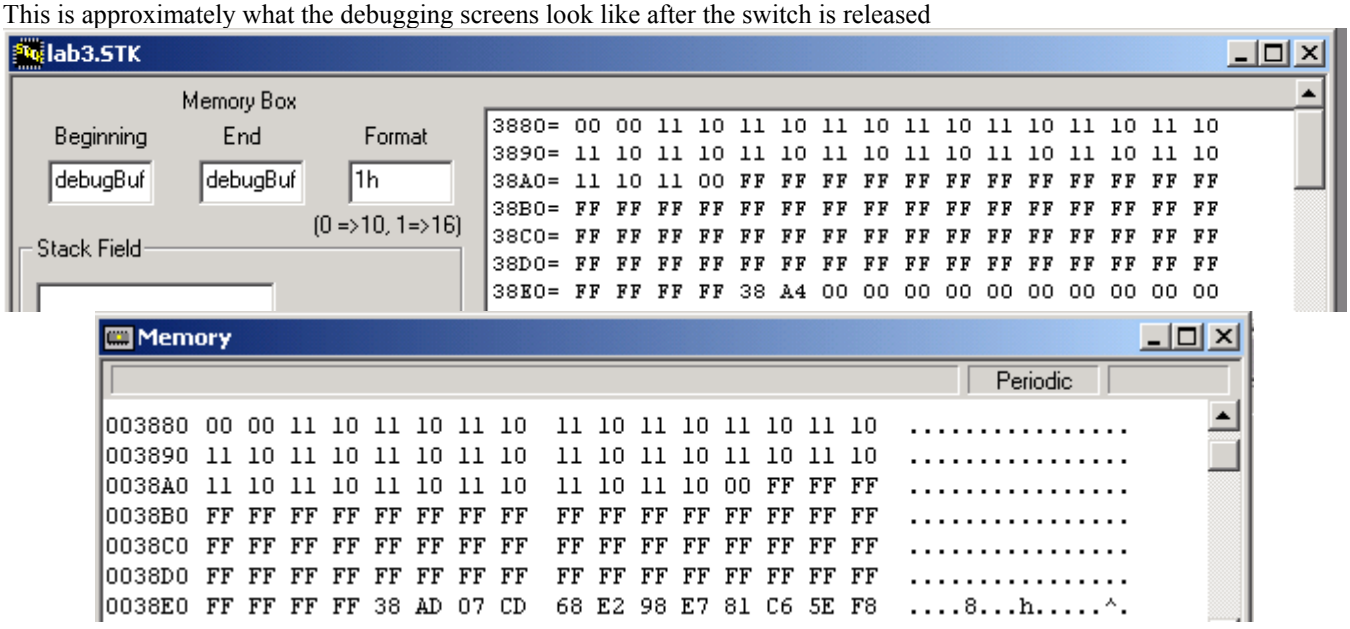
This is what the debugging screens look like after program started but the switch is not yet pushed



This is what the **TExaS** debugging screen looks like after the switch is pushed for about 4 seconds



This is approximately what the debugging screens look like after the switch is released



Lab 4. Traffic Light Controller

Preparation

Read Sections 3.5, 5.1, and 5.2 in the textbook
See http://users.ece.utexas.edu/~valvano/EE319K/LED_yellow.pdf
See http://users.ece.utexas.edu/~valvano/EE319K/LED_green.pdf

This lab has these major objectives:

- The usage of linked list data structures;
- Create a segmented software system;
- Real-time synchronization by designing an input-directed traffic light controller.

In preparation for this assignment, review finite state machines, linked lists, and memory allocation. You should also run and analyze the linked list controllers found in example files **moore.rtf** and **mealy.rtf**.

The basic approach to this lab will be to first develop and debug your system using the simulator. During this phase of the project you will run with a fast **TCNT** clock (**TSCR2**=0). After the software is debugged, you will interface actual lights and switches to the 9S12, and run your software on the real 9S12. During this phase of the project you will run with a slow **TCNT** clock (**TSCR2**=$07). As you have experienced, the simulator requires more actual time to simulate 1 cycle of the microcomputer. On the other hand, the correct simulation time is maintained in the **TCNT** register, which is incremented every cycle of simulation time. The simulator speed depends on the amount of information it needs to update into the windows. Unfortunately, even with the least amount of window updates, it would take a long for the simulator to process the typical 3 minutes it might take for a "real" car to pass through a "real" traffic intersection. Consequently, the cars in this traffic intersection travel much faster than "real" cars. In other words, you are encouraged to adjust the time delays so that the operation of your machine is convenient for you to debug and for the TA to observe during demonstration.

Description

You will create a segmented software system putting global variables into RAM, local variables into RAM, constants and fixed data structures into EEPROM, and program object code into EEPROM. Most microcontrollers have a rich set of timer functions. For this lab, you will the ability to wait a prescribed amount of time. This initialization function will enable the 16-bit **TCNT** timer. The value in **TSCR2** determines the rate at which **TCNT** will increment.

```
Timer_Init
     movb #$80,TSCR1   ; enable TCNT
     movb #$07,TSCR2   ; TCNT counts 128 times slower than the E clock
     rts
```

This function will wait a fixed number of cycles using the **TCNT** timer.

```
;************ Timer_Wait***************
; inputs: RegD is the number of cycles to wait
; outputs: none
; errors: RegD must be less or equal to 32767
Timer_Wait
     addd TCNT   ;TCNT value at the end of the wait
wait cpd  TCNT   ;RegD-TCNT<0 when RegD<Tcnt
     bpl  wait
     rts
```

This function will wait a fixed number of 0.01sec intervals using the previous function.

```
;************ Timer_Wait10ms***************
; time delay
; inputs: RegY is the number of 10ms to wait
; outputs: none
; errors: RegY=0 will wait 655.36 sec
Timer_Wait10ms
     ldd  #CYCLES10MS  ;this constant depends on speed of your microcontroller
     bsr  Timer_Wait
     dbne Y,Timer_Wait10ms
     rts
```

You are asked to modify **`Timer_Wait10ms`** so that it reads the input switches during the fixed wait, in such a way that it remembers if any of the three input switches becomes true during the wait. Essentially, you should read the 3-bit input port within the wait loop and or-together the data. The finite state machine will use this method to read the 3-bit input. This method allows your system to respond to a walk button that may go from false to true then to false during a wait.

In general, cycle-counting (simple for loops) has the problem of conditional branches and data-dependent execution times. If an interrupt were to occur during a cycle counting delay, then the delay would be inaccurate using the cycle-counting method. In the above method, however, the timing will be very accurate, even if an interrupt were to occur while the microcomputer was waiting. In more sophisticated systems, other timer modes provide even more flexible mechanisms for microcomputer synchronization. These techniques will be presented in Chapter 11. A linked list solution may not run the fastest, or occupy the fewest memory bytes, but it is a structured technique that is easy to understand, easy to implement, easy to debug, and easy to upgrade.

Consider a typical 4-corner intersection as shown in Figure 4.1. There are two one-way streets are labeled South (cars travel North) and West (cars travel East). There are three inputs to your 9S12, two are car sensors, and one is a walk button. The *South* sensor will be true (1) if one or more cars are near the South intersection. Similarly, the *West* sensor will be true (1) if one or more cars are near the West intersection. The *Walk* sensor will be true (1) if a pedestrian wishes to cross in any direction. There are 8 outputs from your microcomputer that control the two Red/Yellow/Green traffic lights, and the two walk/don't lights. The simulator allows you to attach binary switches to simulate the three inputs and LED lights to simulate the eight outputs.

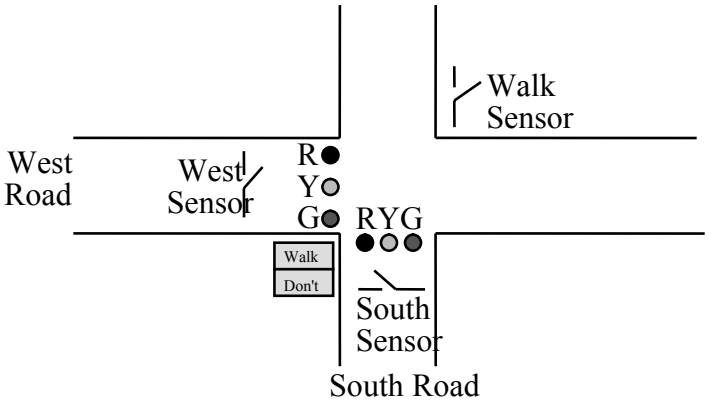


Figure 4.1. Traffic Light Intersection.

Traffic should not be allowed to crash. I.e., there should not be a green or yellow on South at the same time there is a green or yellow on West. You should exercise common sense when assigning the length of time that the traffic light will spend in each state, so that the simulated system changes at a speed convenient for the TA (stuff changes fast enough so the TA doesn't get bored, but not too fast that the TA can't see what is happening). Cars should not be allowed to hit the pedestrians. The walk sequence should be realistic (*walk*, flashing *don't*, continuous *don't*). Your system should consider both the average and worst case waiting time. You may assume the two car sensors remain active for as long as service is required. On the other hand, the walk button may be pushed and released, and the system must remember the walk has been requested.

Part a) Build an I/O system in **TExaS** with the appropriate names and colors on the lights and switches. Think about which ports you will be using in part d), so that you simulate the exact system you will eventually plan to build.

Part b) Design a finite state machine that implements a good traffic light system. Include a graphical picture of your finite state machine showing the various states, inputs, outputs, wait times and transitions. Remember the wait function will return input data collected while it is waiting.

Part c) Write the assembly code that implements the traffic light control system. There is no single, "best" way to implement your traffic light. However, your scheme must be segmented into RAM/EEPROM and you must use a linked-list data structure. There should be a 1-1 mapping from the FSM states and the linked list elements. A "good" solution has about 10 to 20 states in the finite state machine, and provides for input dependence. Try not to focus on the civil engineering issues. Rather, build a quality computer engineering solution that is easy to understand and easy to change. Do something reasonable, and have 10-20 states. A good solution has

1) 1-1 mapping between state graph and data structure
2) no conditional branches in program
3) the state graph defines exactly what it does in a clear and unambiguous fashion
4) the format of each state is the same
5) good names and labels.

Typically in real applications using an embedded system, we put the executable instructions and the finite state machine linked list data structure into the nonvolatile memory (flash EEPROM). A good implementation will allow minor changes to the finite machine (adding states, modifying times, removing states, moving transition arrows, changing the initial state) simply by changing the linked list controller, without changing the executable instructions. Making changes to executable code requires you to debug/verify the system again. If there is a 1-1 mapping from FSM to linked-list data structure, then if we just change the state graph and follow the 1-1 mapping, we can be confident our new system still operates properly. Obviously, if we add another input sensor or output light, it may be necessary to update the executable part of the software and re-assemble. During the debugging phase with the TExaS simulator, you can run with a fast **`TCNT`** clock (**`TSCR2`**=$00).

Part d) After the software has been debugged on the simulator, you will implement it on the real board. **EE319K.sch** is a starter file you should use to draw your hardware circuit diagram. The first step is to interface three push button switches for the sensors. You can implement positive logic or negative logic switches, as shown in Figure 3.2. *Do not place or remove wires on the protoboard while the power is on.* Build the switch circuits and test the voltages using a digital voltmeter. You can also use the debugger to observe the input pin to verify the proper operation of the interface.

The next step is to build six LED output circuits. You can use the two LEDs on the docking module (PT1, PT0) in addition to the 6 external LEDs you will build on your protoboard. Look up the pin assignments in the 7406 data sheet. Be sure to connect +5V power to pin 14 and ground to pin 7. You can use the debugger to set the direction register to output. Then, you can set the output high and low, and measure the three voltages (input to 7406, output from 7406 which is the LED cathode voltage, and the LED anode voltage).

Part e) Debug your combined hardware/software system on the actual 9S12 board. When using the real 9S12, you should run with a slow **`TCNT`** clock (**`TSCR2`**=$07).

During checkout, you will be asked to show both the simulated and actual 9S12 systems to the TA. An interesting question that may be asked during checkout is how you could experimentally prove your system works. In other words, what data should be collected and how would you collect it?

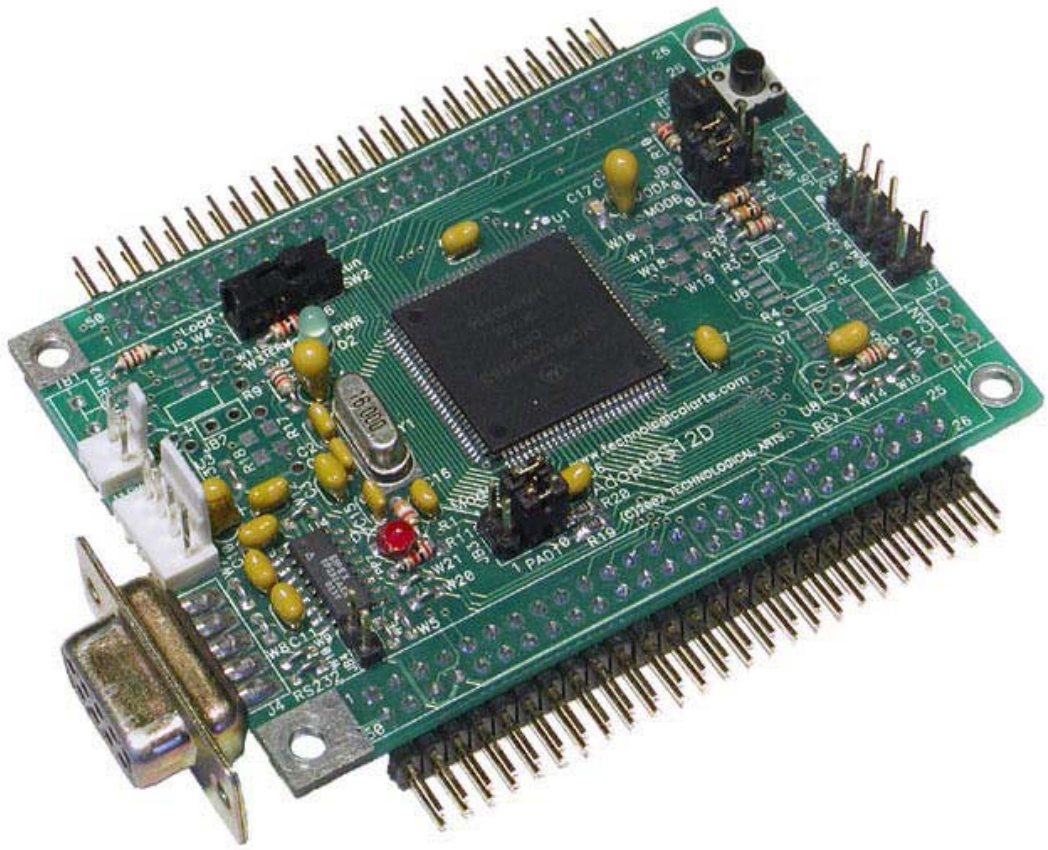
Deliverables

1) Circuit diagram (with your name and date)
2) Drawing of the finte state machine
3) Assembly listing of your final program

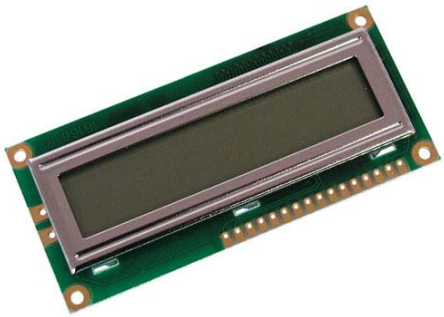
9S12DP512 I/O Pins

Chip	TechArts	TExaS V1.32 Simulation
PAD15-PAD8	H2	Not simulated
PAD7-PAD0	H1	Analog input or Digital input
PA7-PA0	H2	Digital I/O, no external data bus
PB7-PB0	H2	Digital I/O, no external data bus
PE7-PE0	H1/H2 no PE5,PE6	Digital I/O, no IRQ, no XIRQ, no external data bus
PH7-PH0	H1	Digital I/O, Key wakeup
PJ7-PJ6	H2	Digital I/O, Key wakeup, no I2C, no CAN
PJ1-PJ0	H2	Digital I/O, Key wakeup
PM7-PM0	H2	Digital I/O, no CAN, no SPI
PP7-PP0	H1	Digital I/O, Key wakeup, no SPI, no PWM
PS7-PS0	H1	Digital I/O, SCI0, no SCI1, no SPI
PT7-PT0	H1	Digital I/O, input capture, output compare

9S12DP512 Memory Map
$0000 to $03FF I/O ports
$0400 to $07FF 1k EEPROM
$0800 to $3FFF 14k RAM
$4000 to $FFFF 48K Flash EEPROM
Paged memory allows access to 512K EEPROM



Lab 5. LCD Device Driver



Preparation

Read Sections 2.11, 4.7, 6.5, and 8.1
Download the data sheets for the LCD display

http://users.ece.utexas.edu/~valvano/EE345L/DataSheets/OptrexLCD.pdf
http://users.ece.utexas.edu/~valvano/EE345L/DataSheets/LCD.pdf
http://users.ece.utexas.edu/~valvano/EE345L/DataSheets/hd44780.pdf

This lab has these major objectives:

- Interface an LCD interface used to display information on the embedded system;
- Development of a device driver;
- Allocation of local variables on the stack.

You should also observe the **`HD44780.RTF`** program included with the **TExaS** example files.

Purpose

The basic approach to this lab will be to first develop and debug your system using the simulator. During this phase of the project you will use the **TExaS** debugger to observe your software operation. After the software is debugged, you will run your software on the real 9S12. Figure 5.1 shows one possible connection between the 9S12DP512 and the LCD. This configuration employs 4-bit data mode, which requires fewer I/O pins but necessitates a more complex communication protocol. The examples in the book and in **`HD44780.RTF`** employ 8-bit mode. Many microcontrollers have a limited number of pins, therefore you will interface the LCD using 4-bit data mode, which requires only 6 output pins of the 9S12. This lab will use "blind cycle" synchronization, which means after the software issues an output command to the LCD, it will blindly wait a fixed amount of time for that command to complete. For 16-pin LCD devices pins 15 and 16 should be left not connected.

The LCD is physically configured as 16 characters in one row, but internally the device is configured as 2 rows of 8. The left-most 8 characters exist at addresses $00 $01 $02 $03 $04 $05 $06 and $07. The right-mode 8 characters exist at addresses $40 $41 $42 $43 $44 $45 $46 $47. In particular, the ASCII character at LCD address $07 is adjacent to the character at LCD address $40.

The objective of this lab is to develop a device driver for the LCD display. A device driver is a set of functions that facilitate the usage of an I/O port. In particular, there are three components of a device driver. First component is the description of the driver. If the software were being in C, then this description would have been the function prototypes for the public functions, which would have been placed in the header file of the driver, e.g., the LCD.H. Since this driver will be developed in assembly, your descriptions are placed in the comments before each subroutine. It is during the design phase of a project that this information is specified. In this lab, you are required to develop and test these seven public functions (notice that public functions include **`LCD_`** in their names)

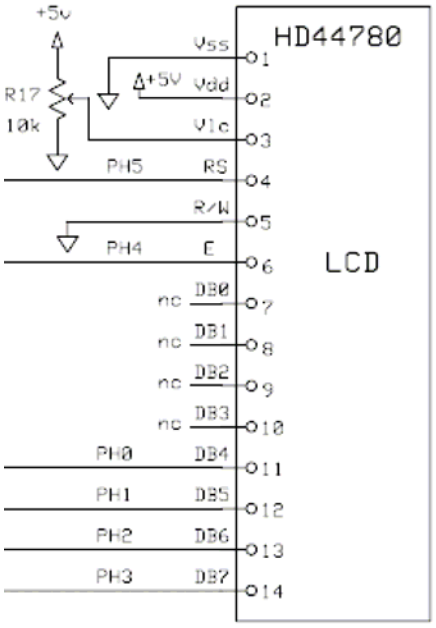


Figure 5.1. One possible circuit diagram that interfaces the LCD to the 9S12DP512.

```
;---------------------LCD_Open---------------------
; initialize the LCD display, called once at beginning
; Input: none
; Output: none
; Registers modified: CCR

;---------------------LCD_Clear--------------------
; clear the LCD display, send cursor to home
; Input: none
; Outputs: none
; Registers modified: CCR
```

```
;----------------------LCD_OutChar---------------------
; sends one ASCII to the LCD display
; Input: RegA (call by value) letter is ASCII code
; Outputs: none
; Registers modified: CCR

;-----------------------LCD_GoTo-----------------------
; Move cursor (set display address)
; Input: RegA is display address is 0 to 7, or $40 to $47
; Output: none
; errors: it will check for legal address

; ----------------------LCD_OutString-------------
; Output character string to LCD display, terminated by a NUL(0)
; Inputs:  RegX (call by reference) points to a string of ASCII characters
; Outputs: none
; Registers modified: CCR

;-----------------------LCD_OutDec----------------------
; Output a 16-bit number in unsigned decimal format
; Input: RegD (call by value) 16-bit unsigned number
; Output: none
; Registers modified: CCR

; ----------------------LCD_OutFix----------------------
; Output characters to LCD display in fixed-point format
; unsigned decimal, resolution 0.001, range 0.000 to 9.999
; Inputs:  RegD is an unsigned 16-bit number
; Outputs: none
; Registers modified: CCR
; E.g., RegD=0,    then output "0.000 "
;       RegD=3,    then output "0.003 "
;       RegD=89,   then output "0.089 "
;       RegD=123,  then output "0.123 "
;       RegD=9999, then output "9.999 "
;       RegD>9999, then output "*.*** "
```

The second component of a device driver is the implementation of the functions that perform the I/O. If the driver were being developed in C, then the implementations would have been placed in the corresponding code file, e.g., LCD.C. When developing a driver in assembly, the implementations are the instructions and comments placed inside the body of the subroutines. In addition to public functions, a device driver can also have private functions. This interface will require a private function that outputs to commands to the LCD (notice that private functions do not include **LCD_** in their names).

```
;----------------------outCsr---------------------
; sends one command code to the LCD control/status
; Input: RegA is 8-bit command to execute
; Output: none
  0) save any registers that will be destroyed by pushing on the stack
  1) E=0, RS=0
  2) 4-bit DB7,DB6,DB5,DB4 = most significant nibble of command
  3) E=1
  4) E=0       (latch 4-bits into LCD)
  5) 4-bit DB7,DB6,DB5,DB4 = least significant nibble of command
  6) E=1
  7) E=0       (latch 4-bits into LCD)
  8) blind cycle 90 us wait
  9) restore the registers by pulling off the stack
```

An important factor in device driver design is to separate the policies of the interface (how to use the programs, which is defined in the comments placed at the top of each subroutine) from the mechanisms (how the programs are implemented, which is described in the comments placed within the body of the subroutine.) Possible algorithms for the seven functions are as follows

```
LCD_Open
  0) save any registers that will be destroyed by pushing on the stack
  1) initialize timer Timer_Init()
  2) wait 100ms allowing the LCD to power up (can skip this step in TExaS)
  3) set DDRH so that PH5-0 are output signals to the LCD
  4) E=0, RS=0
  5) 4-bit DB7,DB6,DB5,DB4 = $02 (DL=0 4-bit mode)
  6) E=1
  7) E=0       (latch 4-bits into LCD)
  8) blind cycle 90 us wait
  9) outCsr($06)   // I/D=1 Increment, S=0 no displayshift
  10)outCsr($0C)   // D=1 displayon, C=0 cursoroff, B=0 blink off
  11)outCsr($14)   // S/C=0 cursormove, R/L=1 shiftright
  12)outCsr($28)   // DL=0 4bit, N=1 2 line, F=0 5by7 dots
  13)LCD_Clear()   // clear display
  14)restore the registers by pulling off the stack

LCD_OutChar
  0) save any registers that will be destroyed by pushing on the stack
  1) E=0, RS=1
  2) 4-bit DB7,DB6,DB5,DB4 = most significant nibble of data
  3) E=1
  4) E=0       (latch 4-bits into LCD)
  5) 4-bit DB7,DB6,DB5,DB4 = least significant nibble of data
  6) E=1
  7) E=0       (latch 4-bits into LCD)
  8) blind cycle 90 us wait
  9) restore the registers by pulling off the stack

LCD_Clear
  0) save any registers that will be destroyed by pushing on the stack
  1) outCsr($01)      // Clear Display
  2) blind cycle 1.64ms wait
  3) outCsr($02)      // Cursor to home
  4) blind cycle 1.64ms wait
  5) restore the registers by pulling off the stack

LCD_OutString
  0) save any registers that will be destroyed by pushing on the stack
  1) read one character from the string
  2) increment the sting pointer to the next character
  3) break out of loop (go to step 6) if the character is NUL(0)
  4) output the character to the LCD by calling LCD_OutChar
  5) loop back to step 1)
  6) restore the registers by pulling off the stack

LCD_GoTo
  0) save any registers that will be destroyed by pushing on the stack
  1) go to step 3 if DDaddr is $08 to $3F or $48 to $FF
  2) outCsr(DDaddr+$80)
  3) restore the registers by pulling off the stack
```

LCD_OutDec (recursive implementation)

```
  1) allocate local variable n on the stack
  2) set n with the input parameter passed in RegD
  3) if(n >= 10){
       LCD_OutDec(n/10);
       n = n%10;
     }
  4) LCD_OutChar(n+$30); /* n is between 0 and 9 */
  5) deallocate variable
```

LCD_OutFix

```
  0) save any registers that will be destroyed by pushing on the stack
  1) allocate local variables letter and num on the stack
  2) initialize num to input parameter, which is the integer part
  3) if number is less or equal to 9999, go the step 6
  4) output the string "*.*** " calling LCD_OutString
  5) go to step 19
  6) perform the division num/1000, putting the quotient in letter,
      and the remainder in num
  7) convert the ones digit to ASCII, letter = letter+$30
  8) output letter to the LCD by calling LCD_OutChar
  9) output '.' to the LCD by calling LCD_OutChar
  10)perform the division num/100, putting the quotient in letter,
      and the remainder in num
  11)convert the tenths digit to ASCII, letter = letter+$30
  12)output letter to the LCD by calling LCD_OutChar
  13)perform the division num/10, putting the quotient in letter,
      and the remainder in num
  14)convert the hundredths digit to ASCII, letter = letter+$30
  15)output letter to the LCD by calling LCD_OutChar
  16)convert the thousandths digit to ASCII, letter = num +$30
  17)output letter to the LCD by calling LCD_OutChar
  18)output ' ' to the LCD by calling LCD_OutChar
  19)deallocate variables
  20)restore the registers by pulling off the stack
```

The third component of a device driver is a main program that calls the driver functions. This software has two purposes. For the developer (you), it provides a means to test the driver functions. It should illustrate the full range of features available with the system. The second purpose of the main program is to give your client or customer (e.g., the TA) examples of how to use your driver. Here is a 9S12DP512 example test program, assuming a positive logic switch is connected to PORTAD0 bit 7 (PAD7).

```
      org  $4000
Entry lds  #$4000
      bset ATDDIEN,#$80   ;PAD7 digital input
      jsr  LCD_Open       ;***Your function that initializes the LCD***
start
      ldx  #Welcome
      jsr  LCD_OutString  ;***Your function that outputs a string***
      ldx  #TestData
loop  brset PTAD,#$80,*   ;wait for switch release
      brclr PTAD,#$80,*   ;wait for switch touch
      jsr  LCD_Clear      ;***Your function that clears the display***
      ldd  0,x
      jsr  LCD_OutDec     ;***Your function that outputs an integer***
      ldaa #$40           ;Cursor location of the 8th position
      jsr  LCD_GoTo       ;***Your function that moves the cursor***
```

```
      ldd  2,x+
      jsr  LCD_OutFix     ;***Your function that outputs a fixed-point***
      cpx  #TestDataEnd
      bne  loop
      jsr  LCD_Clear      ;***Your function that clears the display***
      bra  start
Welcome  fcc "Welcome "
         fcc "                                " ;32 spaces
         fcc "to 319K!"
         fcb 0
TestData fdb 0,5,16,123,5432,9876,9999,10000,23456,65535
TestDataEnd
```

Procedure
You will use the **TExaS** simulator to develop and test your device driver, and then you will copy and paste your solutions into the Metrowerks version for running on the real 9S12. There are many functions to write in this lab, so it is important to develop the device driver in small pieces. One technique you might find useful is **desk checking**. Basically, you hand-execute your functions with a specific input parameter. For example, using just a pencil and paper think about the sequential steps that will occur when **`LCD_OutDec`** or **`LCD_OutFix`** processes the input 9876. Later, while you are debugging the actual functions on the simulator, you can single step the program and compare the actual data with your expected data.

Part a) One by one each of the subroutines should be designed, implemented and tested. **Successive refinement** is a development approach that can be used to solve complex problems. If the problem is too complicated to envision a solution, you should redefine the problem and solve an easier problem. If it is still too complicated, redefine it again, simplifying it even more. You could simplify **`LCD_OutFix`**

1) implement the variables in global variables (rather than as local variables on the stack)
2) ignore special cases with illegal inputs
3) implement just one decimal digit

During the development phase, you implement and test the simpler problem then refine it, adding back the complexity required to solve the original problem. You could simplify **`LCD_OutDec`** in a similar fashion.

Part b) Using **ExpressSCH** (or another equivalent application), draw the hardware circuit diagram. You can find the ExpressSCH file used to create Figure 5.1 on **http://users.ece.utexas.edu/~valvano/Starterfiles/EE319K_DP512.sch**. You can checkout a LCD display from the second floor lab. If you choose to interface in a manner different than Figure 5.1, have a TA approve your design. Build the interface using the circuit diagram. Please double check your connections before applying power.

Part c) This lab is sufficiently complex that it should be first debugged on the **TExaS** simulator. Although this LCD physically looks like 16 characters by 1 row, internally it is configured as 8 characters by 2 rows. To simulate this LCD in **TExaS**, we will define a 16 by 2 LCD. The first 8 characters of the first row of the **TExaS** LCD will map into the first 8 characters (left most) of the real LCD. The first 8 characters of the second row of the **TExaS** LCD will map into the second (right most) 8 characters of the real LCD. Even though we will not actually be connecting R/W to PAD0, we will attach R/W=PAD0 in **TExaS**, so that line will be simulated as a 0. The "Busy cleared after 37us/1.54ms" option allows you to test the timing aspect of the LCD interface (the blind cycle waits).

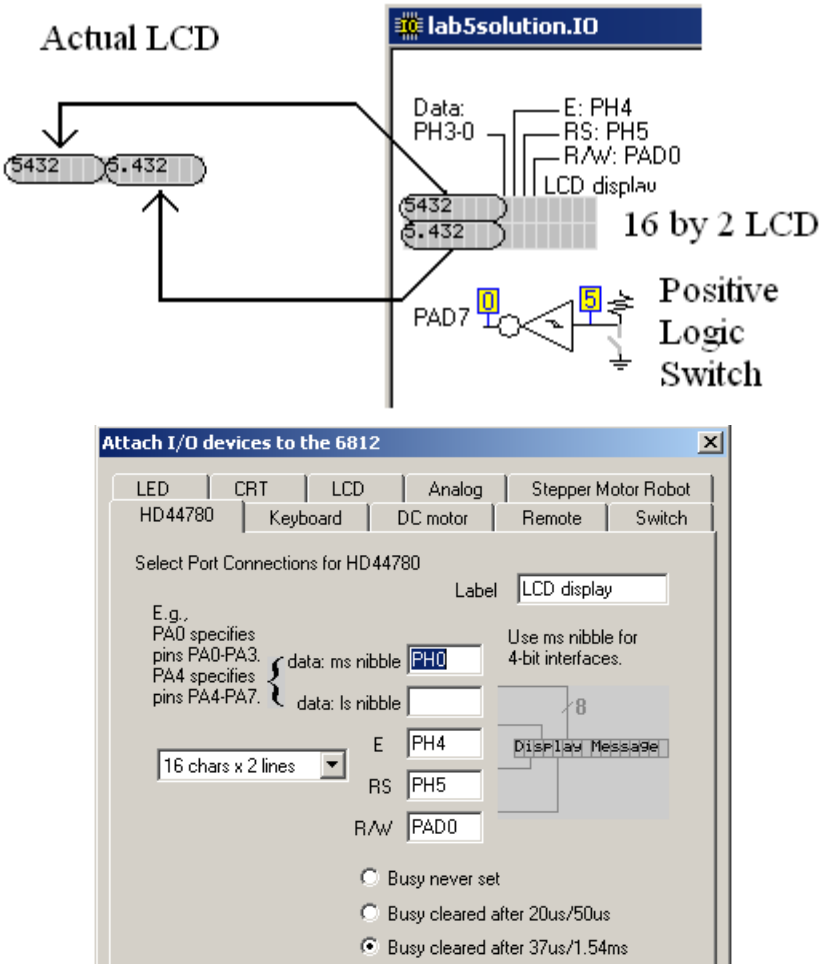


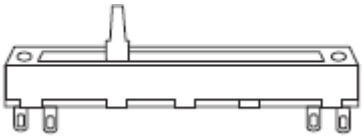
Figure 5.2. One possible way to interface the LCD in TExaS.

Once the system is debugged on the simulator, download and debug it on the real 9S12.

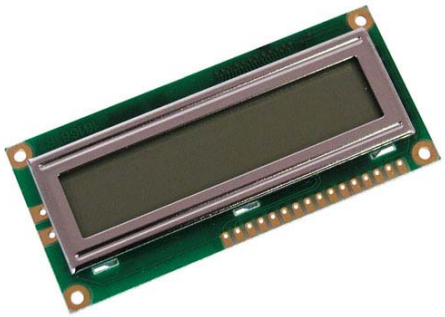
During demonstration to the TA, you will run your system on the simulator and show the allocation, access and deallocation of the local variables. You will also be required to demonstration the operation on the actual 9S12. Each time a function is called, an **activation record** is created on the stack, which includes parameters passed on the stack (none in this lab), the return address, and the local variables. You will be asked to create a stack window and identify the activation records created during the execution of **`LCD_OutDec`**.

Deliverables

1) Circuit diagram (with your name and date)
2) Assembly listing of your final program (device driver plus main program that tests the system)



Lab 6. Real-Time Position Measurement System



Preparation

Read Sections 5.1, 5.5, 6.6, 11.1, 11.5, and 11.9.2

http://users.ece.utexas.edu/~valvano/EE319K/312-9100F-SlidePot.pdf

This lab has these major objectives:

- An introduction to sampling analog signals using the ADC interface;
- Development of an ADC device driver;
- Data conversion and calibration techniques;
- Develop an interrupt-driven real-time sampling device drive.

Starter files

- `OC` example in **TExaS**;
- `TUT3` example in **TExaS**

The basic approach to this lab will be to first develop and debug your system using the simulator. During this phase of the project you will use the **TExaS** debugger to observe your software operation. After the software is debugged, you will run your software on the real 9S12.

Background

You will design a **position meter** with a range of about 3 cm. A linear slide potentiometer (Alpha RA300BF-10-20D1-B54) converts position into resistance (0<R<50 kΩ). You will use an electrical circuit to convert resistance into voltage (Vin). Since the potentiometer has three leads, one possible solution is shown in Figure 6.1. The 9S12 ADC will convert voltage into a 10-bit digital number (0 to 1023). Your software will calculate position from the ADC sample as a decimal fixed-point number. The position measurements will be displayed on the LCD using the LCD device driver developed in the last lab. A periodic interrupt will be used to establish the real-time sampling.

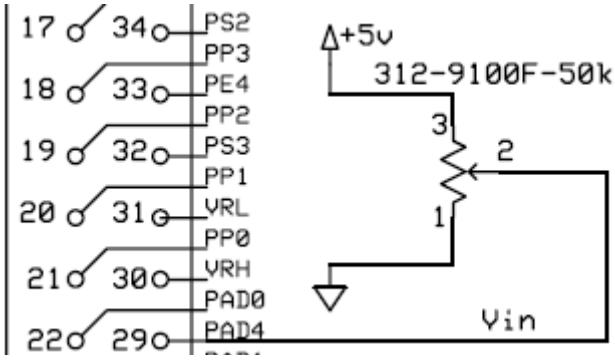


Figure 6.1. Possible circuit diagram of the sensor interface (look on the sensor for pin numbers 1,2,3).

The left of Figure 6.2 shows the data flow graph of this system. Dividing the system into modules allows for concurrent development and eases the reuse of code. The right of Figure 6.2 shows the call graph.

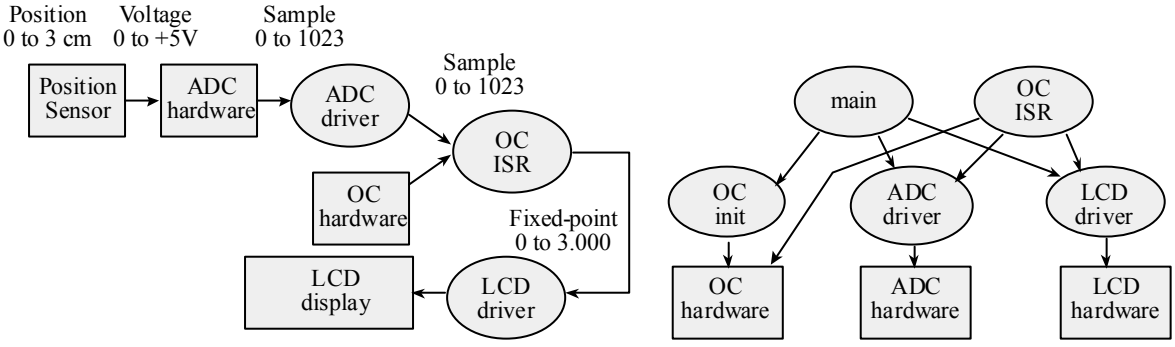


Figure 6.2. Data flow graph and call graph of the position meter system.

You should make the position resolution and accuracy as good as possible. The **position resolution** is the smallest change in position that your system can reliably detect. In other words, if the resolution were 0.01 cm and the position were to change from 1.00 to 1.01 cm, then your device would be able to recognize the change. Resolution will depend on the amount of electrical noise, the number of ADC bits, and the resolution of the output display software. Considering just the errors due to the 10-bit ADC, we expect the resolution to be 3cm/1024 or about 0.003cm. **Accuracy** is defined as the absolute difference between the true position and the value measured by your device. Accuracy is dependent on the same parameters as resolution, but in addition it is also dependent on the stability of the transducer and the quality of the calibration procedure.

In this lab, you will be measuring the position of the armature (the movable part) on the slide potentiometer. This signal has very few frequency components (0 to 2 Hz.) According to the Nyquist Theorem, we need a sampling rate greater than 4 Hz. Consequently, you will create a system with a sampling rate of 5 Hz. You will sample the ADC exactly every 0.2 sec and calculate position using decimal fixed-point with Δ of 0.001 cm. You should display the results on the LCD, including units. An output compare interrupt will be used to establish the real-time periodic sampling.

Nyquist Theorem: If f_{max} is the largest frequency component of the analog signal, then you must sample more than twice f_{max} in order to faithfully represent the signal in the digital samples. For example, if the analog signal is $A + B \sin(2\pi ft + \phi)$ and the sampling rate is greater than 2f, you will be able to determine A, B, f, and ϕ from the digital samples.

Valvano Postulate: If f_{max} is the largest frequency component of the analog signal, then you must sample more than ten times f_{max} in order for the reconstructed digital samples to look like the original signal when plotted on a voltage versus time graph.

When a transducer is not linear, you could use a piece-wise linear interpolation to convert the ADC sample to position (Δ of 0.001 cm.) The 9S12 assembly language **`etbl`** instruction is an efficient mechanism to perform the interpolation. The **`etbl.RTF`** assembly program included with **TExaS** is an example of a piece-wise linear interpolation using the **`etbl`** instruction. There are two small tables **`Xtable`** and **`Ytable`**. The **`Xtable`** contains the ADC results and the **`Ytable`** contains the corresponding positions. The ADC sample is passed into the lookup function. This function first searches the **`Xtable`** for two adjacent of points that surround the current ADC sample. Next, the function uses the **`etbl`** instruction to perform a linear interpolation to find the position that corresponds to the ADC sample. You are free to implement the conversion in any acceptable manner, with the exception that you are **not** allowed to use the **`etbl`** instruction.

The 10-bit ADC converters on the 9S12 are successive approximation devices with a short conversion time. You need to enable the ADC in **`ATD0CTL2`**. In particular, you will set **`ATD0CTL2=$80`**. You can define the number of ADC conversions (1 to 8) in a sequence using **`ATD0CTL3`**. For this lab, you will only need a single conversion, so you can set the control bits S8C S4C S2C S1C in **`ATD0CTL3`** equal to 0001 respectively. In particular, you will set **`ATD0CTL3=$08`**. Bit 7 of determines if the ADC operates with 8 bits or 10 bits. You will clear bit 7 to specify 10-bit precision. The remaining 7 bits of **`ATD0CTL4`** specify the ADC clock, which will determine the time to perform an ADC conversion. If the 9S12DP512 were running at 8 MHz, you should set **`ATD0CTL4=$03`**. At this setting, the ADC will be clocked at 1 MHz, and the ADC conversion time will be equal to 14 μs. However, in this lab we will be running the 9S12DP512 at 24 MHz, therefore you could set **`ATD0CTL4=$05`**, the ADC will be clocked at 2 MHz, and the ADC conversion time will be equal to 7 μs. In summary, the ADC initialization should set

```
ATD0CTL2=$80  turns on ADC
ATD0CTL3=$08  specifies ADC sequence will perform one conversion
ATD0CTL4=$05  specifics 10-bit mode, and 7us conversion time
```

Writing to the ADC Control register (**`ATD0CTL5`**) begins a conversion. The ADC chip clocks itself. To perform a right-justified ADC conversion of channel 4, you should write a $84 to **`ATD0CTL5`**. After the first sample is complete, **`CCF0`** is set and the result can be read out of the first result register, **`ATD0DR0`**. After the entire sequence has been converted, the **`SCF`** bit is set. In summary, the ADC conversion of channel 4 requires the following actions

```
1) ATD0CTL5=$84  starts the ADC
2) Read ATD0STAT1 and look at bit 0 (CCF0)
3) Loop back to step 2 over and over until CCF0 is set (7us)
4) Read 10-bit result in ATD0DR0
```

Procedure

The analog signal connected to the microcomputer comes from a position sensor, such that the analog voltage ranges from 0 to +5V as the position ranges from 0 to 3 cm. First, you will use output compare interrupts to establish 5 Hz sampling. In particular, the ADC should be started exactly every 0.2 s. Second, you will convert the ADC sample (0 to 1023) into a 16-bit unsigned decimal fixed-point number, with a Δ of 0.001 cm. Lastly, you will use your **`LCD_OutFix`** function from the previous lab to display the sampled signal on the LCD. Include units on your display.

Add this code to your project, so the system runs at 24 MHz in both Run and Load modes. You should double-check the wait times in the LCD routines to make sure the blind cycle waits are valid for the 24 MHz clock.

```
SYNR     equ $0034  ; CRG Synthesizer Register
REFDV    equ $0035  ; CRG Reference Divider Register
CRGFLG   equ $0037  ; CRG Flags Register
CLKSEL   equ $0039  ; CRG Clock Select Register
PLLCTL   equ $003A  ; CRG PLL Control Register
;********* PLL_Init ***************
; Active PLL so the 9S12 runs at 24 MHz
; Inputs: none
; Outputs: none
; Errors: will hang if PLL does not stabilize
PLL_Init
  movb #$02,SYNR       ; 9S12DP512 OSCCLK is 16 MHz
  movb #1,REFDV
  movb #0,CLKSEL       ; PLLCLK = 2 * OSCCLK * (SYNR + 1) / (REFDV + 1)
  movb #$D1,PLLCTL     ; Clock monitor, PLL On, high bandwidth filter
  brclr CRGFLG,#$08,*  ; wait for PLLCLK to stabilize.
  bset CLKSEL,#$80     ; Switch to PLL clock
  rts
```

Part a) You can create a scale by Xerox-copying a metric ruler. There are many ways to build the transducer. One method requires cutting, gluing, and soldering. Start with a piece of wood or plastic a little larger than the potentiometer. Glue the frame (the fixed part) of the potentiometer to this solid object. Tape or glue the metric ruler on the frame but near the armature (the movable part) of the sensor. Attach or draw a hair-line to the armature, which will define the position measurement. Solder three solid wires to the slide potentiometer. If you do not know how to solder, ask your TA for a lesson. Label these three wires +5 (Pin3), Vin (Pin2), and ground (Pin1), as shown in Figure 6.1.

Be careful when connecting the potentiometer to the computer, because if you mistakenly reverse two of the wires, you can cause a short from +5V to ground.

Part c) Write two subroutines: **`ADC_Init`** will initialize the ADC interface and **`ADC_In4`** will sample the ADC channel 4. Use the simulator to test these functions.

Part d) Write a simple simple version of the system, which you can use to collect calibration data. In particular, this system should first sample the ADC and then display the results as unsigned decimal numbers. You should use your **`LCD_OutDec`** developed in the previous lab. Collect five to ten calibration points and create a table showing the true position (as determined by reading the position of the hair-line on the ruler), the analog input measured with a digital voltmeter and the ADC sample (like the first three columns of Table 6.1).

Position	Analog input	ADC sample	Fixed-point Output
0.010 cm	0.000 V	0	10
0.741 cm	1.234 V	252	741
1.500 cm	2.500 V	512	1500
2.074 cm	3.456 V	707	2074
3.000 cm	5.000 V	1023	3000

Table 6.1. Calibration results of the conversion from ADC sample to fixed-point.

Part e) Use this calibration data to write a subroutine that converts a 10-bit binary ADC sample into a 16-bit unsigned fixed-point number. The input parameter (10-bit ADC sample) to the subroutine will be passed in using Register D, and your subroutine will return the result (integer portion of the fixed-point number) in Register D. Table 6.1 shows some example results. You are allowed to use a linear equation to convert the ADC sample into the fixed-point number.

Part f) Write a subroutine: **`OC_Init`** will initialize the output compare system to interrupt at exactly 5 Hz (every 0.2 second). Use the simulator to test these functions. When debugging your code in **TExaS** it will be more convenient to run with a shorter OC interrupt period, e.g., 10 to 50ms.

1) disable interrupts to make the initialization atomic (set I bit in CCR)
2) enable the timer and an output compare channel, make PT7 an output (interface a LED to this pin)
3) arm output compare
4) specify when the first output compare interrupt will be
5) enable interrupts (clear I bit in CCR)

Part g) Write an output compare interrupt handler that samples the ADC and outputs the data to the LCD. Use the simulator to test these functions. Using the interrupt synchronization, the ADC will be sampled at almost equal time intervals[1]. The interrupt service routine performs these tasks

1) acknowledge the output compare interrupt by clearing the flag that requested the interrupt
2) specify the time for the next interrupt
3) toggle PT7 (change from 0 to 1, or from 1 to 0)
4) sample the ADC
5) convert the sample into a fixed-point number (0 to 3000)
6) output the fixed-point number on the LCD
7) return from interrupt

Part h) Write a simple main program, which initializes the PLL, timer, LCD, ADC and output compare interrupts. After initialization, this main program (foreground) performs a do-nothing loop. The entire run-time operations occur in the output compare interrupt service routine (background).

Part i) Use the system to collect another five to ten data points, creating a table showing the true position (x_{ti} as determined by reading the position of the hair-line on the ruler), and measured position (x_{mi} using your device). Calculate average accuracy by calculating the average difference between truth and measurement,

$$\text{Average accuracy (with units in cm)} = \frac{1}{n}\sum_{i=1}^{n} \left| x_{ti} - x_{mi} \right|$$

True position x_{ti}	Measured Position x_{mi}	Error $x_{ti} - x_{mi}$

Table 6.2. Accuracy results of the position measurement system.

Deliverables

1) Circuit diagram showing the position sensor and LCD
2) Final version of the software
3) Calibration data, like Table 6.1
4) Accuracy data and accuracy calculation, Table 6.2

[1] More precisely, the output compare flag is set at exact time intervals. There is some variability in when the ISR runs depending on which instruction is being executed at the time when the flag is set.

Lab 7 Distributed Data Acquisition System

Goals • Develop a distributed data acquisition system
Review • Operation of the SCI in Section 5.3, 6.4, and 11.7,
• Fifo queues in Sections 10.8 and 11.2.
Starter files • **`TUT2 TUT4`** examples in **TExaS**

Procedure

You will extend the system from Lab 6 to implement a distributed system. In particular, one 9S12 will sample the data at 5 Hz and a second 9S12 will display the results on its LCD. Basically the hardware/software components from Lab 6 will be divided and distributed across two 9S12 microcontrollers. Figure 7.1 shows the data flow graph of the distributed data acquisition system. The sensor is attached to computer 1, and the ADC (**`ADC_In4`** function) in computer 1 generates a digital value from 0 to 1023. The output compare periodic interrupt in computer 1 establishes the real-time sampling at 5 Hz. You will send data from computer 1 to computer 2 using asynchronous serial communication (SCI1). Busy-wait synchronization on TDRE must be used in computer 1, and RDRF interrupt synchronization must be used on computer 2. You can choose any baud rate you wish, as long as both computers use the same rate.

One way to send the 10-bit sample is to break it into two parts and transmit it as two 8-bit bytes. For example, let $b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$ be a 10-bit sample. One possibility is to transmit two bytes $011b_9b_8b_7b_6b_5$ and $010b_4b_3b_2b_1b_0$. This way the receiver can combine the two bytes back into a 10-bit sample without mistakenly switching the most significant and least significant parts. Also notice that all transmissions are printable ASCII, so the two parts of the system can be separately debugged in **TExaS**. The time to transmit one bit is called the bit time, which is 1 divided by the baud rate. Every 200 ms, two bytes will be transferred (a total of 20 bits). Choose a baud rate so that the time to transmit 20 bits is short compared to 200 ms. This will guarantee that both the transmit data register and the transmit shift register will be empty at the time the OC ISR is executed. Therefore, calling **`SCI_OutChar`** twice (busy-wait synchronization) will not actually have to wait, because the first data will be moved immediately into the transmit shift register and the second data can be loaded into the transmit data register (to be transmitted after the first frame is done). With this protocol if you lose a transmission, then the receiver should discard the extra byte. In this scheme, it does not matter which computer starts first.

An alternate scheme to transmit 10-bit data on an 8-bit channel is to encode the data as a signed 8-bit difference from the previous data. The receiver starts with a 16-bit 0, each 8-bit signed data received is promoted to 16-bit signed and added to the previous value. One flaw in this protocol is if you lose a transmission, then an error will exist in all subsequent samples. However since each 10-bit sample is transmitted as only one 10-bit frame, this protocol will be twice as fast as the previous.

If you implement a scheme that requires 3 or more SCI transmissions per sample, then the busy-wait synchronization in the transmitter will actually have to wait. This system is simple enough that no data should be lost. Thus, you do not have to solve the lost data scenario. However, you might have to start computer 2 before starting computer 1.

An RDRF interrupt will occur in computer 2 for every SCI frame received. The FIFO queue is used to pass data from the RDRF interrupt service routine (background on computer 2) to the main program running in the foreground on computer 2. If the rate at which the ISR puts data into the FIFO is slower than the rate at which data can be sent to the LCD, then the FIFO will never become full. You are free to implement either a 16-bit FIFO (every other RDRF interrupt puts into the FIFO) or an 8-bit FIFO (every RDRF interrupt puts). The main program in computer 2 will output the position on its LCD.

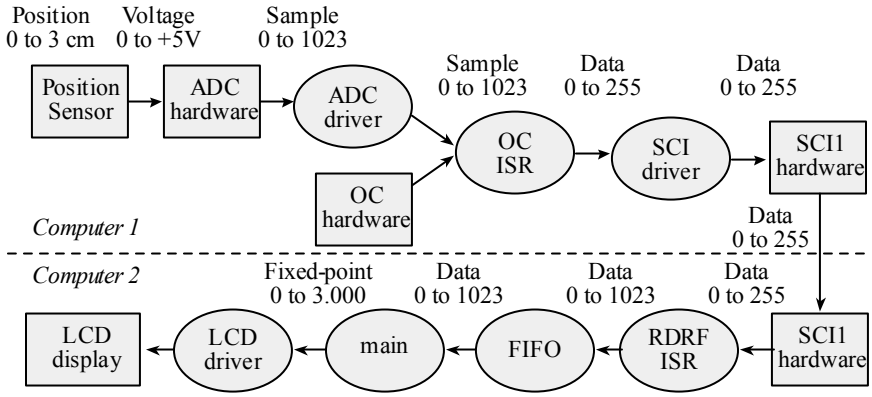


Figure 7.1. Data flows from the sensor through the two microcontrollers to the LCD. The output compare timer is used to trigger the real-time sampling. Use the special serial cable to connect the two SCI1 ports.

Figure 7.2 shows a possible call graph of the system. Dividing the system into modules allows for concurrent development and eases the reuse of code.

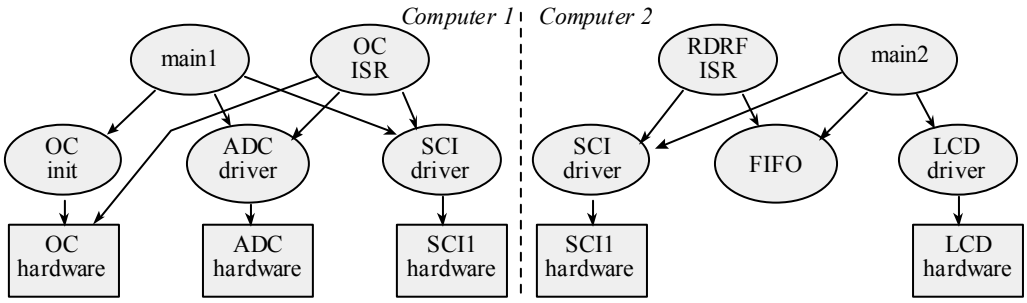


Figure 7.2. A call graph showing the modules used by the distributed data acquisition system.

Computer 1 software tasks

Part a) Write a subroutine: **`SCI_Init1`** that will initialize the SCI1 transmitter in computer 1.

1) enable SCI1 transmitter (no interrupts)
2) set the baud rate

Part b) Write a subroutine: **`SCI_OutChar`** for computer 1 that sends one byte using busy-wait synchronization on TDRE.

1) Wait for TDRE in `SCI1SR1` to be 1
2) Write a byte to `SCI1DRL`

Part c) Modify the output compare interrupt handler from Lab 6 so that it samples the ADC at 5 Hz and sends the data to the other computer using SCI. The interrupt service routine performs these tasks

1) acknowledge the output compare interrupt by clearing the flag that requested the interrupt
2) specify the time for the next interrupt
3) toggle PT7 (change from 0 to 1, or from 1 to 0)
4) sample the ADC
5) break the 10-bit sample into two parts and send two bytes to the other computer
(calls `SCI_OutChar` twice)
6) return from interrupt

Part d) Write the main program for computer 1, which initializes the PLL, timer, ADC, SCI1, and output compare interrupts. After initialization, this main program (foreground) performs a do-nothing loop. The entire run-time operations in computer 1 occur in the output compare interrupt service routine (background).

Computer 2 software tasks

Part e) Design, implement and test a FIFO software module for computer 2 that operates on either 8-bit or 16-bit values. This module should operate in a similar manner as the FIFOs in the example **`tut4`**. I.e., write three subroutines **`Fifo_Init`**, **`Fifo_Put`**, **`Fifo_Get`**. The size of the queue can be about 4 to 6 elements. Use the simulator to test these functions. The software design steps are

1) Define the names of the functions, input/output parameters, and calling sequence. Type these definitions in as comments that exist at the top of the subroutines.
2) Write pseudo-code for the operations. Type the sequences of operations as comments that exist within the bodies of the subroutines.
3) Write assembly code to handle the usual cases. I.e., at first, assume the FIFO is not full on a put, not empty on a get, and the pointers do not need to be wrapped.
4) Write a main program to test the FIFO operations. Debug in the **TExaS** simulator.
5) Iterate steps 3 and 4 adding code to handle the special cases.

Part f) Write a subroutine: **`SCI_Init2`** that will initialize the SCI1 receiver in computer 2.

1) clear a global error count
2) enable SCI1 receiver (arm interrupts for RDRF)
3) set the baud rate to match computer 1
4) enable interrupts

Part g) Write a RDRF interrupt handler that receives data from the other computer and puts them into a FIFO queue. The number of lost samples will be maintained in the global error count. The interrupt service routine performs these tasks

1) acknowledge the interrupt by clearing the flag which requested the interrupt
2) read the data received from SCI1DRL
3) toggle PT7 (change from 0 to 1, or from 1 to 0)
4) put the new data into the FIFO queue
5) increment a global error count if the FIFO fills up (but don't loop back)
6) return from interrupt

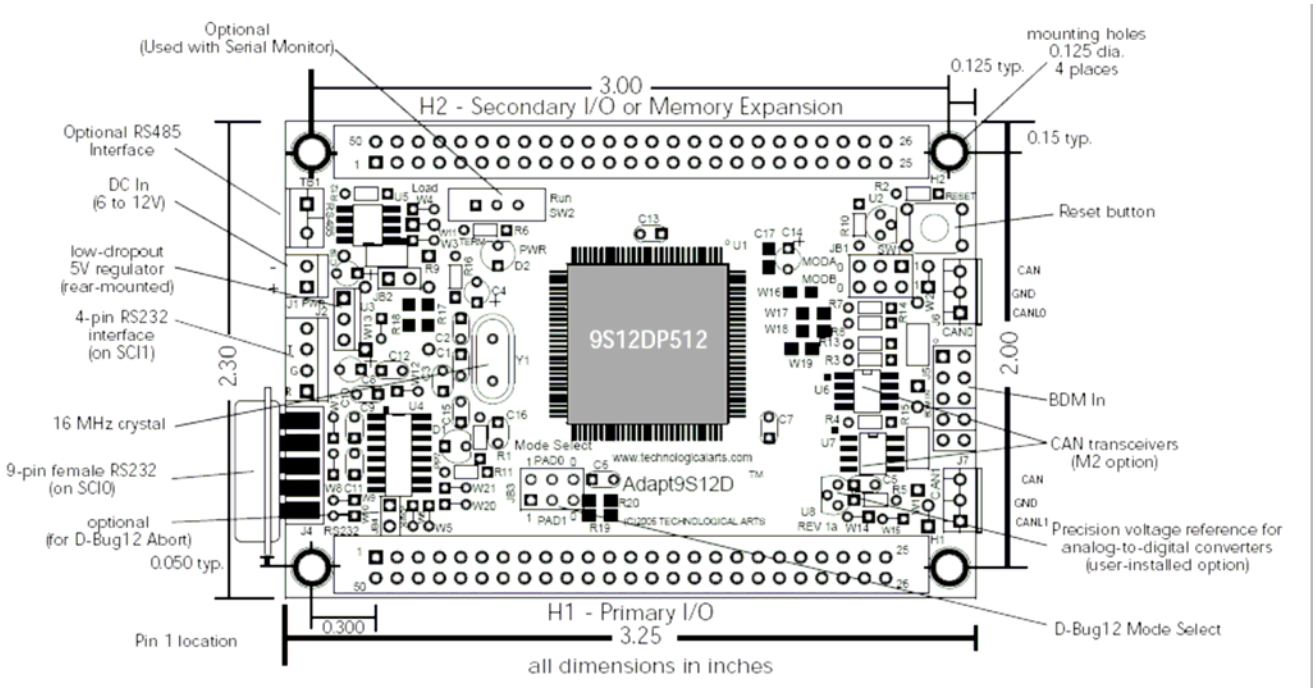
Part h) Design a main program for computer 2 that reads data from the FIFO, converts it to fixed-point, and displays the measurement using the same LCD routines developed in Lab 5 and used in Lab 6.
The main program in this data acquisition system performs these tasks

1) initialize PLL, FIFO, LCD, and SCI
2) try to remove a sample from the FIFO queue
3) go back to step 2 if the FIFO was empty and no data is available
4) convert sample to fixed-point (same as Lab 6)
5) output the result as a fixed-point number (same as Lab 6) with units
6) repeat steps 2,3,4,5 over and over

(5% extra credit) **Part i)** In this section you will estimate the maximum sampling rate. The limitation of computer 1 will be the time it takes to transmit 20 bits on the SCI. The limitation of computer 2 will be the time it takes to display one measurement on the LCD (e.g., if you move the LCD cursor rather than clearing the display, it will run faster). Change the output compare interrupt period in computer 1 so that it is close to but larger (slower) than the time for computer 1 to transmit one measurement and the time it takes computer 2 to display one measurement. Experimentally, verify the system can operate properly at this sampling rate (show the FIFO never gets full). Next, change the output compare sample period so that it is close to but smaller (faster) than these two times. Experimentally, determine want happens when you try to sample this fast (it doesn't work, explain what happens and why). Without actually doing it, describe two changes to the system you could do so that the sampling rate could be increased beyond this limit. Hint: changing the size of the FIFO will not affect the maximum sampling rate.

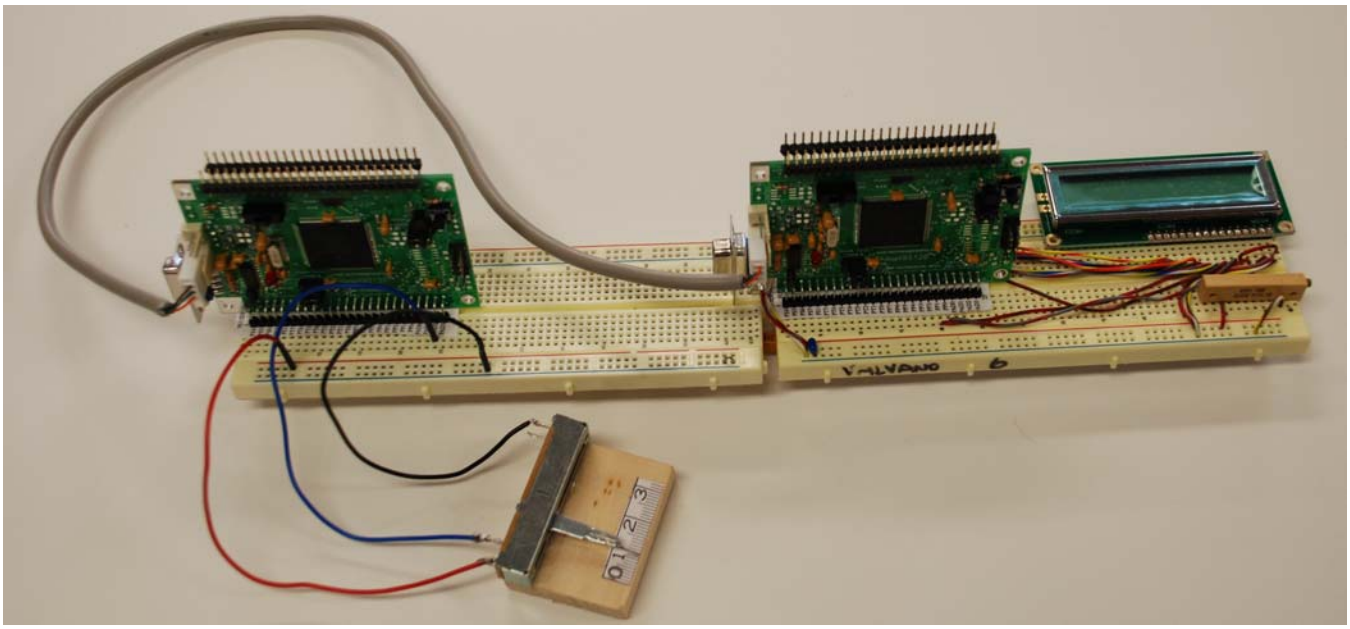
Deliverables

1) Circuit diagram showing the position sensor and LCD (should be the same as Lab 6)
2) Final versions of the software in the two computers



Special cable connects the 4-pin RS232 interface (on SCI1) between the two computers

Computer 1		Computer 2
TxD	----------	RxD
RxD	----------	TxD
Ground	----------	Ground



Lab 8. Music generation using a Digital to Analog Converter

Goals • DAC conversion,
• Design a data structure to represent music,
• Develop a system to play sounds.

Download http://users.ece.utexas.edu/~valvano/Starterfiles/dac.xls

Background

Most digital music devices rely on high-speed DAC converters to create the analog waveforms required to produce high-quality sound. In this lab you will create a very simple sound generation system that illustrates this application of the DAC. Your goal is to create an embedded system that plays three note (a digital piano with three keys). The first step is to design and test a 4-bit DAC, which converts 4 bits of digital output from the 9S12 to an analog signal. You are free to design your DAC with a precision more than 4 bits. You will convert the binary bits (digital) to an analog output using a simple resistor network. During the static testing phase, you will connect the DAC analog output to your voltmeter and measure resolution, range, precision and accuracy. During the dynamic testing phase you will connect the DAC output to headphones, and listen to sounds created by your software. It doesn't matter what range the DAC is, as long as there is an approximately linear relationship between the digital data and the speaker current. The performance score of this lab is not based on loudness, but sound quality. The quality of the music will depend on both hardware and software factors. The precision of the DAC, external noise and the dynamic range of the speaker are some of the hardware factors. Software factors include the DAC output rate and the complexity of the stored sound data. You can create a 3k resistor from two 1.5k resistors. You can create a 6k resistor from two 12k resistors,

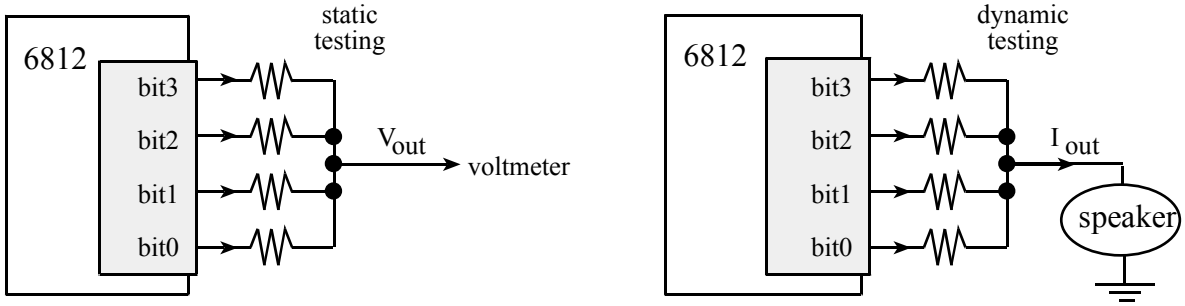


Figure 8.1. DAC allows the software to create music.

The second step is to design a low-level device driver for the DAC. Remember, the goal of a device driver is to separate what the device does (general descriptions of **`DAC_Init`** and **`DAC_Out`**) from how is does it (implementations of **`DAC_Init`** and **`DAC_Out`**). The third step is to design a data structure to store the sound waveform. You are free to design your own format, as long as it uses a formal data structure. Compressed data occupies less storage, but requires runtime calculation. The fourth step is to organize the digital piano software into a device driver. Although you will be playing only three notes, the design should allow additional notes to be added with minimal effort. For example, if your system plays C, D, E, then you will need public functions **`Piano_Stop Piano_C`**, **`Piano_D`** and **`Piano_E`**. The **`Stop`** function makes it silent and the other functions activate a sound. A background thread implemented with output compare will fetch data out of your music structure and send them to the DAC. The last step is to write a main program that inputs from binary switches and performs the four public functions.

If you output a sequence of numbers to the DAC that form a sine wave, then you will hear a continuous tone on the speaker, as shown in Figure 8.2. The measured data was collected using a 4-bit DAC with a range of 0 to +5 V. The plot on the left was measured with a digital scope (without the speaker attached). The plot on the left shows the frequency response of this data, plotting amplitude (in dB) versus frequency (in kHz). This measured waveform is approximately 2.7+2.3sin(2π440 t) volts. The two peaks in the spectrum are at DC and 440 Hz (e.g., 20*log(2.3)= 7.2 dB). The **loudness** of the tone is determined by the amplitude of the wave. The **pitch** is defined as the frequency of the wave. Table 8.1 contains frequency values for the notes in one octave.

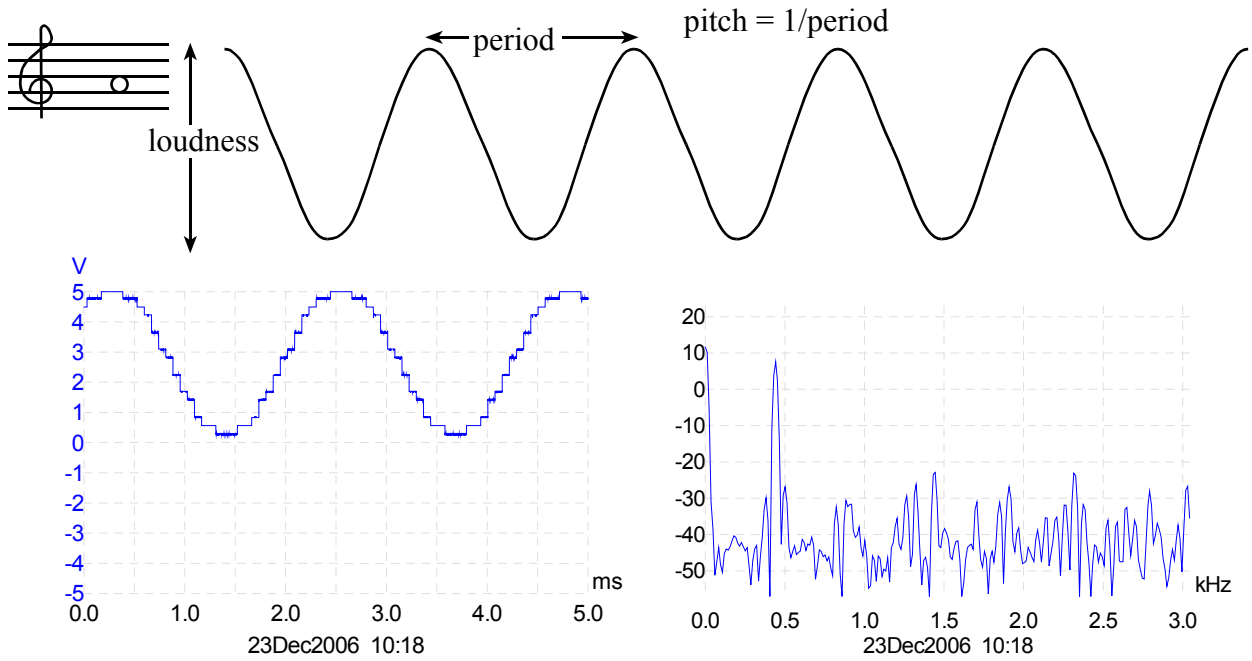


Figure 8.2. A 440Hz sine wave generates a pure tone with note A (theoretical and experimental). The plot on the right is the Fourier Transform(frequency spectrum dB versus kHz) of the data plotted on the left.

Note	frequency
C	523 Hz
B	494 Hz
B^b	466 Hz
A	*440 Hz*
A^b	415 Hz
G	392 Hz
G^b	370 Hz
F	349 Hz
E	330 Hz
E^b	311 Hz
D	294 Hz
D^b	277 Hz
C	262 Hz

Table 8.1. Fundamental frequencies of standard musical notes. The frequency for 'A' is exact.

The frequency of each note can be calculated by multiplying the previous frequency by $\sqrt[12]{2}$. You can use this method to determine the frequencies of additional notes above and below the ones in Table 8.1. There are twelve notes in an octave, therefore moving up one octave doubles the frequency. Figure 8.3 illustrates the concept of **instrument**. You can define the type of sound by the shape of the voltage versus time waveform. Brass instruments have a very large first harmonic frequency.

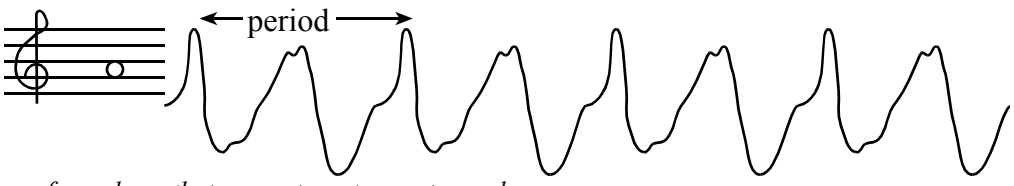


Figure 8.3. A waveform shape that generates a trumpet sound.

The **tempo** of the music defines the speed of the song. In 2/4 3/4 or 4/4 music, a **beat** is defined as a quarter note. A moderate tempo is 120 beats/min, which means a quarter note has a duration of ½ second. A sequence of notes can be separated by pauses (silences) so that each note is heard separately. The **envelope** of the note defines the amplitude versus time. A very simple envelope is illustrated in Figure 8.4. The 9S12DP512 has plenty of processing power to create these types of waves.

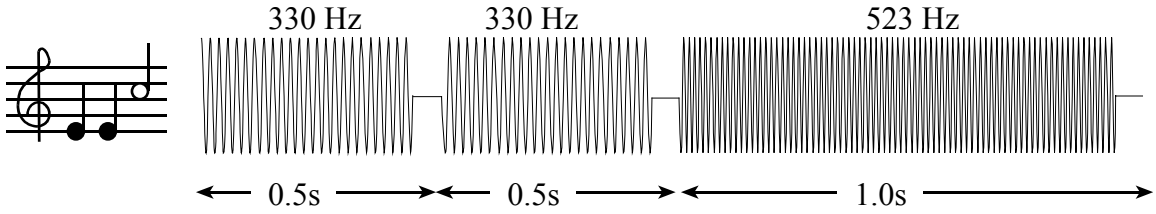


Figure 8.4. You can control the amplitude, frequency and duration of each note (not drawn to scale).

The smooth-shaped envelope, as illustrated in Figure 8.5, causes a less staccato and more melodic sound. This type of sound generation may be difficult to produce in real-time on the 9S12C32. You do not need to create envelopes in this lab.

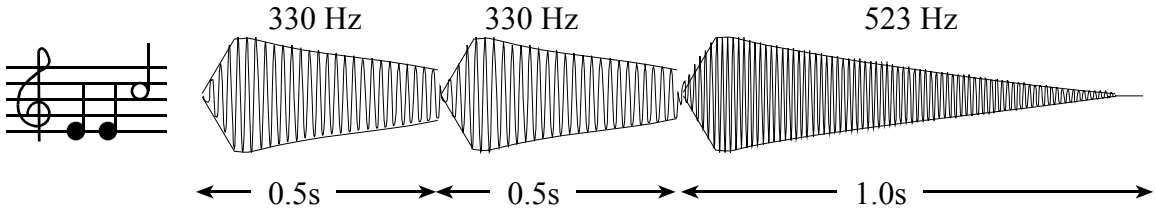


Figure 8.5. The amplitude of a plucked string drops exponentially in time.

A **chord** is created by playing multiple notes simultaneously. When two piano keys are struck simultaneously both notes are created, and the sounds are mixed arithmetically. You can create the same effect by adding two waves together in software, before sending the wave to the DAC. Figure 8.6 plots the mathematical addition of a 262 Hz (low C) and a 392 Hz sine wave (G), creating a simple chord. You do not need to create chords in this lab assignment.

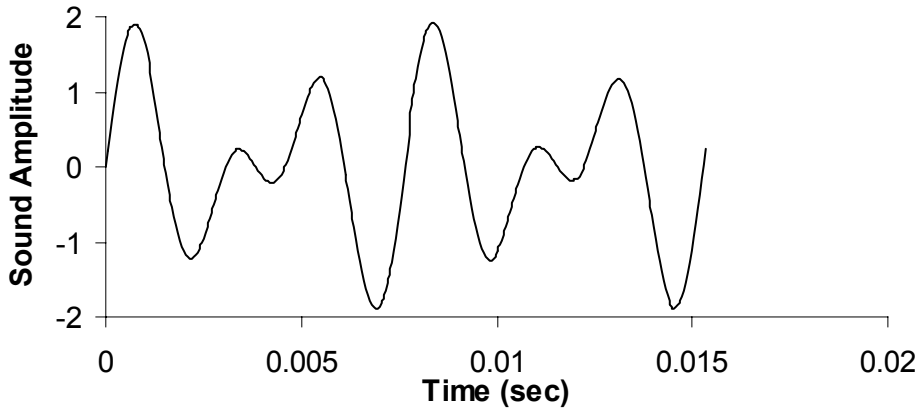


Figure 8.6. A simple chord mixing the notes C and G.

Procedure

Part a) Draw the circuit required to interface the DAC to the 9S12. Design the DAC converter using a simple resistor-adding technique. Use resistors in a 1/2/4/8 resistance ratio. Select values in the 1.5 kΩ to 12 kΩ range. For example, you could use 1.5 kΩ, 3 kΩ, 6 kΩ, and 12 kΩ. Notice that you could create double/half resistance values by placing identical resistors in series/parallel. It is a good idea to email your design to your TA and have him/her verify your design before you build it. You can solder 24 guage solid wires to the jack to simplify connecting your circuit to the headphones.

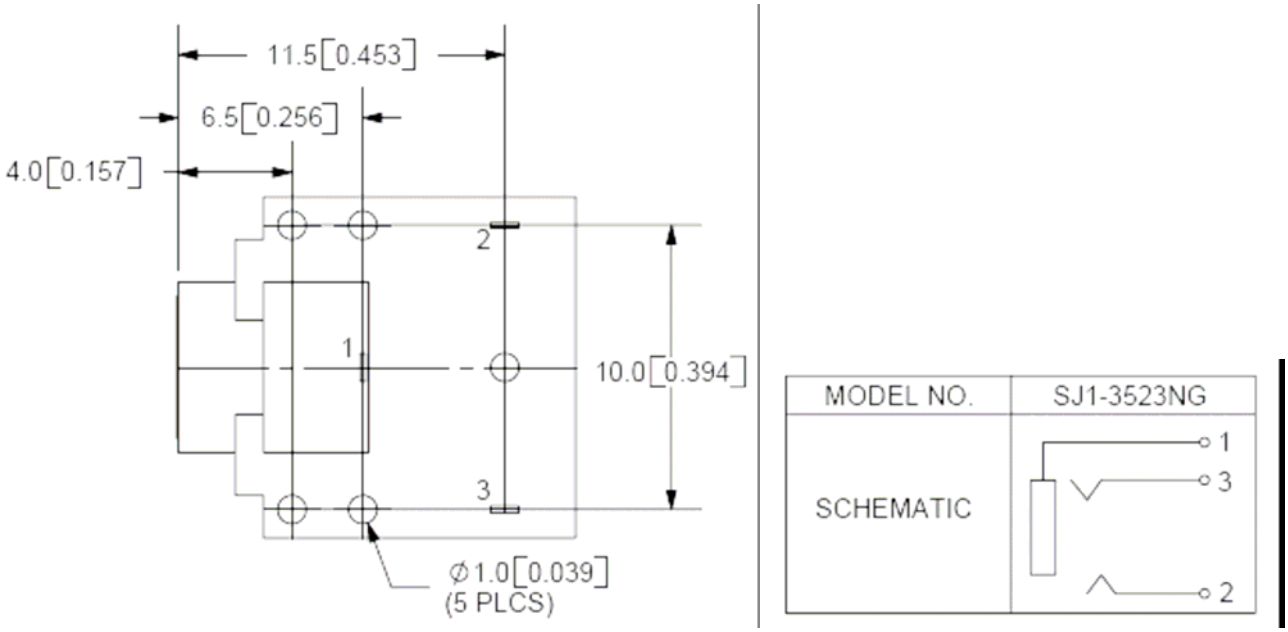


Figure 8.7. A bottom-view mechanical drawing of the stereo jack (connect pin 1 to ground and pin 3 to the DAC output).

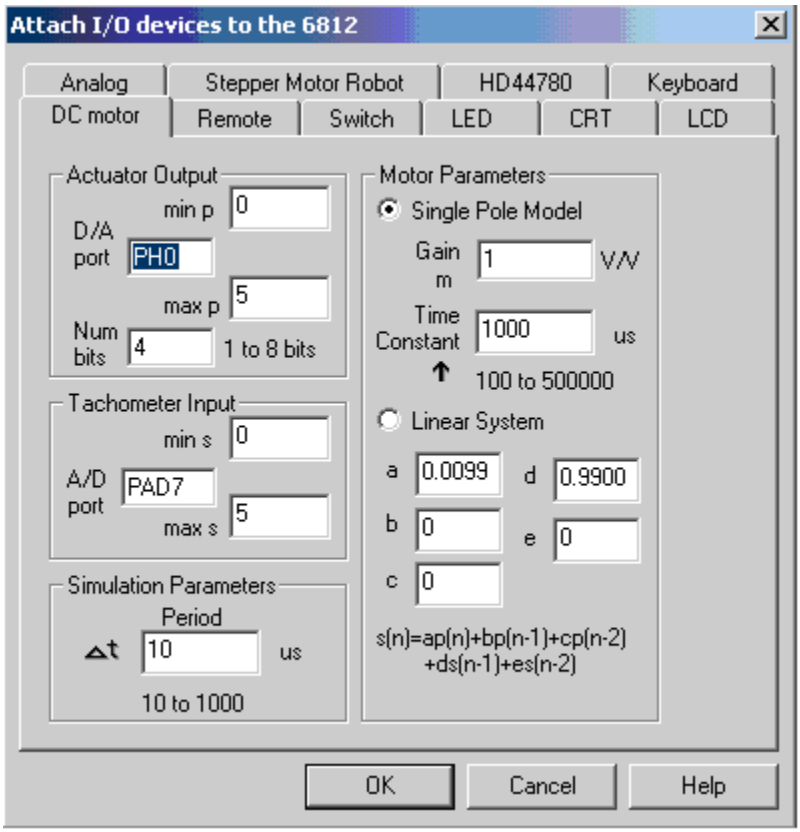


Figure 8.8. Use the DC motor feature to simulate a simple DAC (you must connect the tachometer input to an unused ADC pin, even though you are not using the tachometer).

Part b) Write a low-level device driver for the DAC interface. Include two functions that implement the DAC interface. The function **`DAC_Init()`** initializes the DAC, and the function **`DAC_Out`** sends a new data value to the DAC. You can debug your software in TExaS using the **DC motor** I/O device. This module allows you to connect a DAC to an output port. You can select the precision of the DAC (4 bits in this case). You can visualize the generated waveform on the scope by selecting

the D/A output (or DC motor power). Figure 8.8 shows the TExaS dialog to interface the DAC to PM3,2,1,0, and Figure 8.9 shows a sine wave generated by a 4-bit DAC simulated in TExaS.

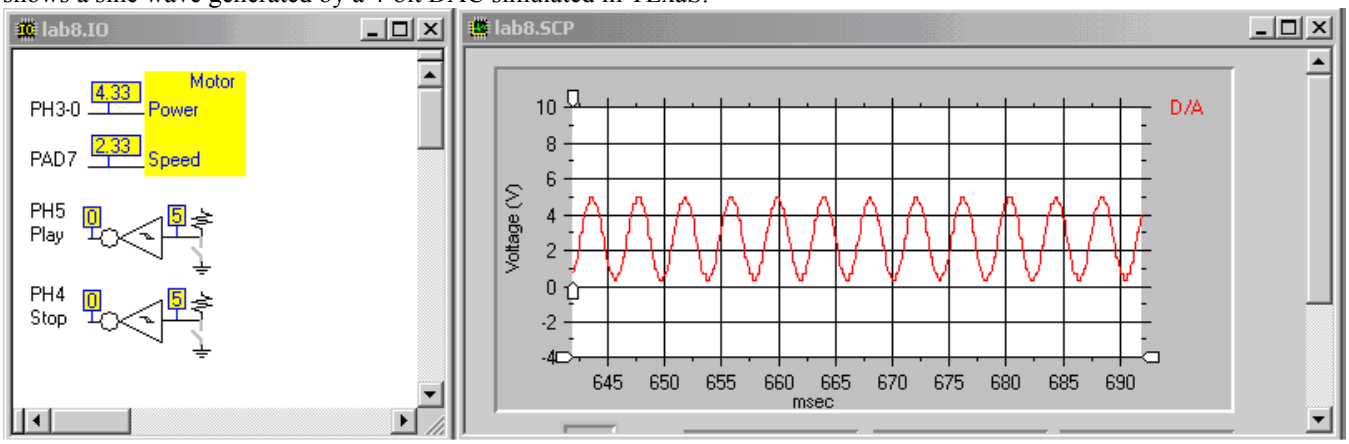


Figure 8.9. The 4-bit DAC is used to create a sin wave using TExaS.

Part c) Write a couple of simple main programs that test the DAC interface. This main program can be used for static testing. You can single step this program using the debugger to test the static function of the DAC (Table 8.2)

```
      org  $4000
Entry lds  #$4000
      jsr  DAC_Init
      clra
loop  jsr  DAC_Out
      inca
      anda #$0F
      bra  loop
```

Part d) Using Ohm's law and fact that the digital output voltages will be approximately 0 and 5 V, make a table of the theoretical DAC voltage and as a function of digital value (without the speaker attached). Calculate resolution, range, precision and accuracy. See Table 8.2.

This main program can be used for dynamic testing. It creates triangle waveform (adjust the **1000** to affect the frequency).

```
      org  $4000
Entry lds  #$4000
      jsr  Timer_Init
      jsr  DAC_Init
      clra
      psha
n     equ  0
loop  ldd  #1000
      jsr  Timer_Wait
      ldaa n,sp
      inca
      jsr  DAC_Out
      staa n,sp
      cmpa #15
      bne  loop
loop2 ldd  #1000
      jsr  Timer_Wait
      ldaa n,sp
      deca
      jsr  DAC_Out
      staa n,sp
      cmpa #0
```

```
bne  loop2
bra  loop
```

Bit3 bit2 bit1 bit0	Theoretical DAC voltage	Measured DAC voltage
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

Table 8.2. Static performance evaluation of the DAC.

Part e) Design and write the piano device driver software. Add minimally intrusive debugging instruments to allow you to visualize when interrupts are being processed.

Part f) Write a main program to run the entire system. Document clearly the operation of the routines. Figure 8.10 shows the data flow graph of the music player.

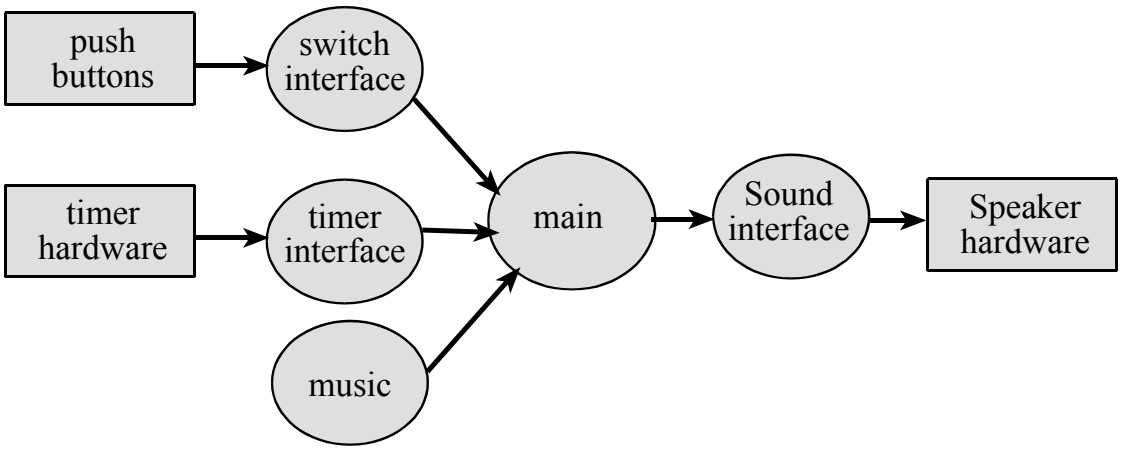


Figure 8.10. Data flows from the memory and the switches to the speaker.

Figure 8.11 shows a possible call graph of the system. Dividing the system into modules allows for concurrent development and eases the reuse of code.

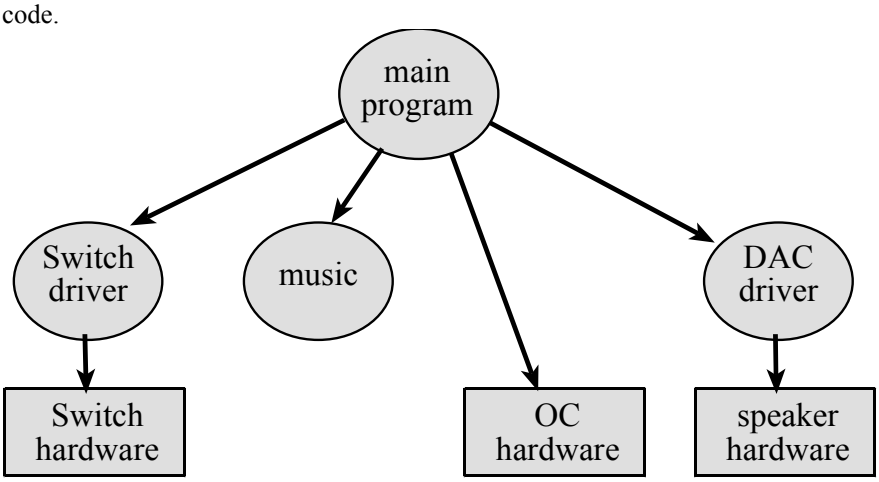


Figure 8.11. A call graph showing the three modules used by the music player.

Deliverables

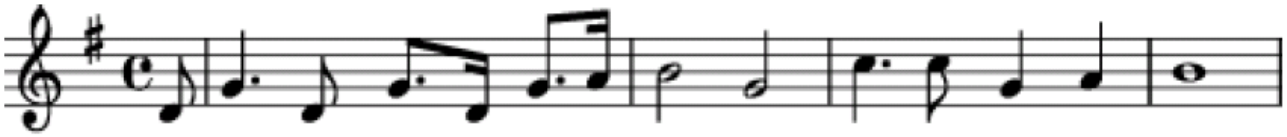
1) Circuit diagram showing the DAC and any other hardware used in this lab
2) Software Design
   - Draw pictures of the data structures used to store the sound data
   - If you organized the system different than Figure 8.7 and 8.8, then draw its data flow and call graphs
3) Measurement Data
   - Show the theoretical response of DAC voltage versus digital value (part d)
   - Show the experimental response of DAC voltage versus digital value (part d)
   - Calculate resolution, range, precision and accuracy
4) Final version of the music playing software (intermediate testing software is not required)

Checkout

You should be able to demonstrate the three notes. You should be prepared to discuss alternative approaches and be able to justify your solution.

Extra Credit. Extend the system so that is plays your favorite song (a sequence of notes, set at a specific tempo and includes an envelop like Figure 8.4). Your goal is to play your favorite song. One possible approach is to use two output compare interrupts. A fast output compare ISR outputs the sinewave to the DAC (Figure 8.2). The rate of this interrupt is set to specify the frequency (pitch) of the sound. A second slow output compare ISR occurs at the tempo of the music. For example, if the

song has just quarter notes at 120, then this interrupt occurs every 500ms. If the song has eight notes, quarter notes and half notes, then this interrupt occurs at 250, 500, 1000ms respectively. During this second ISR, the frequency of the first ISR is modified according to the note that is to be played next. Compressed data occupies less storage, but requires runtime calculation. On the other hand, a complete list of points will be simpler to process, but requires more storage than is available on the 9S12. The fourth step is to organize the music software into a device driver. Although you will be playing only one song, the song data itself will be stored in the main program, and the device driver will perform all the I/O and interrupts to make it happen. You will need public functions **`Play`** and **`Stop`**, which perform operations like a cassette tape player. The **`Play`** function has an input parameter that defines the song to play. If you complete the extra credit (with input switches that can be used to play and stop), then the piano functionality in parts e) and f) need not be completed. Either way, parts a) b) c) and d) are required.



Labs 9 and 10. TExaS Robots 1.7 (for the lastest information check the web site)

http://users.ece.utexas.edu/~valvano/EE319K

TRobots ("TExaS-Robots") is a 9S12 programming competition. Unlike arcade type games that require human inputs controlling some object, all strategy in **TRobots** must be complete before the actual game begins. Game strategy is condensed into a 9S12 assembly program that you design and write. Your software controls a robot tank, see Figure 1, whose mission is to seek out, track, and destroy other robots, each running different programs. Each robot is equally equipped, and from 2 to 50 robots may compete at one time. As a game is simulated, events are displayed graphically in real-time. Multiple simulations will be run in order to eliminate the random occurrences, and the best programmers will be crowned as the team with the largest total score.

Figure 1. Robots have two tracks and one cannon mounted on a turret.

TRobots consists of multiple 9S12-based virtual robots, and a battlefield display. The **TRobots** programs can be created by any 9S12 assembler or compiler, and the machine codes (S19 records) are run by the **TRobots** simulator. The robot programs are run in parallel, giving each robot the same number of 9S12 bus cycles. The virtual robot includes hardware features to scan for opponents, move, turn, fire cannons, position sensing, and direction sensing. After the 9S12 programs are assembled/compiled, the S19 records are loaded into separate robots. Robots moving, missiles flying and exploding, and status information are displayed on the screen in real-time during the battle.

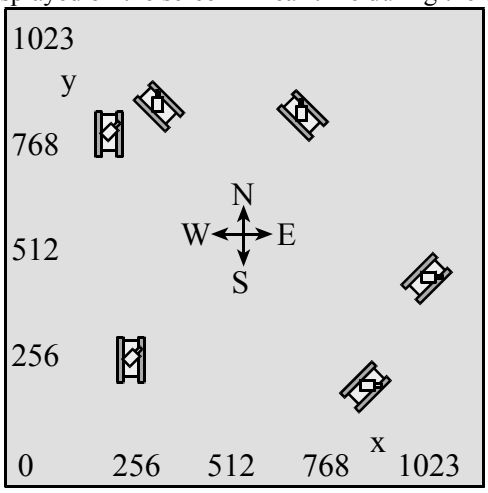


Figure 2. The battlefield is a 1024 by 1024 meter square. North is up.

The *battlefield*, as shown in Figure 2, has a wall surrounds the perimeter, so that a robot running into the wall will bounce off incurring damage. The lower left corner has the coordinates x = 0, y = 0; the upper right corner has the coordinates x = 1023m, y = 1023m. The four life-packs are at (980, 982), (41, 511) , (980, 39), and (511, 511) in meters. The life-packs are about 50 meters by 50 meters. The maximum health is 100%.

Robots are represented on the field by two characters of their S19 object code files, which must be one letter A to Z followed by one digit 0 to 3. The S19 files (extension **`.s19`** or **`.sx`**) must be loaded into the S19 directory, which must be a

subdirectory of the directory that hold the game engine, **`TRobots.exe`**. The number of robots in the competition is determined by the existence of object code files: **`A0.s19`**, **`B0.s19`**, **`C0.s19`**, **`D0.s19`**, … **`Z0.s19`**, **`A1.s19`**, **`B1.s19`**… **`Z3.s19`**, which are located in the S19 directory.

For collision purposes, each robot has a bounding-box that is about 4 meters wide, 5 meters long, and 3 meters high as shown in Figure 3. Robot positions are reported as the center of the box, which is the center of rotation if one track is moved forward, while the other track is moved backwards. The robot heading is the direction of the left and right tracks. The turret heading is relative to the robot. E.g., a turret heading of 0 means the turret is facing in the same direction as the tank.

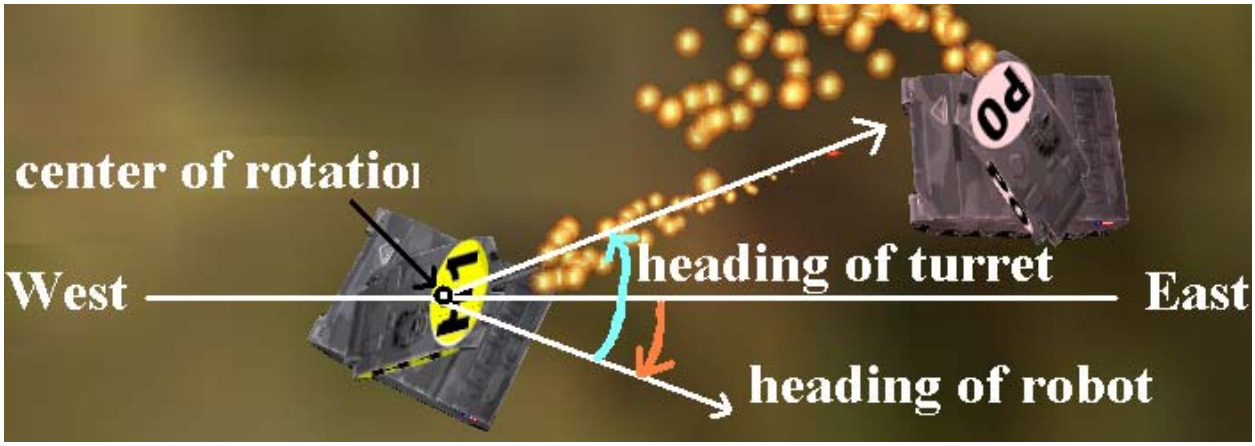


Figure 3. There are two headings of interest. In this picture, the robot heading is about 340°, and the turret heading is about 45°.

The offensive weapon is the *cannon*, which is mounted on a rotating turret. Missiles are fired in the direction of the turret. Transmitting a frame out the serial port fires a missile. The data value sent determines the launch angle of the missile, thus affecting the firing distance. There are an unlimited number of missiles that can be fired, but because of the serial baud rate, there is a maximum rate at which the cannon can be fired. Since the turret can rotate independently from the robot direction, it can fire any direction, regardless of robot heading.

The *scanner* is a sonic ranging device that scans in three sectors (front, left, and right). The scanner is located on the gun turret, therefore senses enemy robots in the same direction as missiles will be fired. The scanner has four resolutions, as shown in Table 1 and Figure 4, controlled by the two bits PTM5 and PTM4. The angles are relative to the gun turret. The values returned by the 10-bit ADC are binary fixed-point with a resolution of 0.25 m.

PTM5	PTM4	Resolution	Left Scanner	Front Scanner	Right Scanner
0	0	5°	+7.5° to +2.5°	+2.5° to -2.5°	-7.5° to -2.5°
0	1	10°	+15° to +5°	+5° to -5°	-5° to -15°
1	0	30°	+45° to +15°	+15° to -15°	-15° to -45°
1	1	120°	+180° to +60°	+60° to -60°	-60° to -180°

Table 1. You can set dynamically set the scanner resolution.

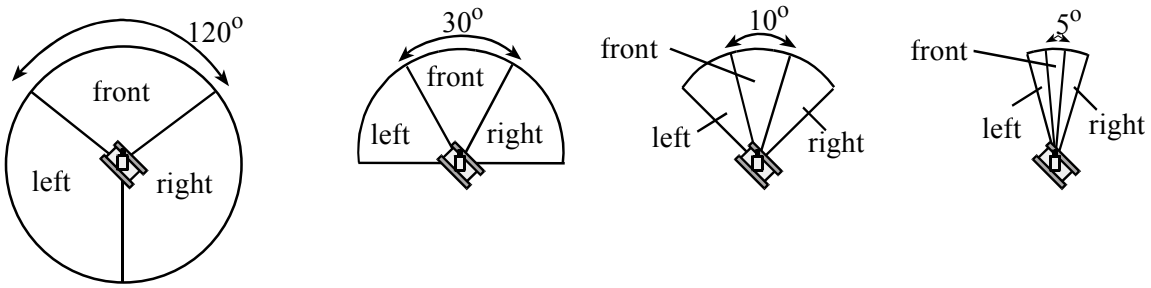


Figure 4. There are scanner resolutions. The front is aligned with the turret (not to scale).

There are three stepper motors that control the robot. One stepper motor controls the left track, and a second stepper motor controls the right track. These two motors cause the robot to turn or move. A third stepper motor rotates the gun turret.

Each stepper interface requires four output bits from the computer. The motors can be independently stepped forwards using a full-step sequence (5,6,10,9,...) or a half-step sequence (5,4,6,2,10,8,9,1,...). Forward stepping causes the robot to move forward. The motors can be also be stepped backwards using a full-step sequence (9,10,6,5,...) or a half-step sequence (1,9,8,10,2,6,4,5,...). Backward stepping causes the robot to move backward. The robot will turn about its center (without translation) if one track is stepped forward and the other is stepped backward. Each motor has 5 possible motions (full-step forward, half-step forward, none, half-step backward and full-step backward.) Since each motor is independent, there are 25 possible robot motions.

The smallest distance that the robot can be moved is 1 meter. The smallest angle that the robot can rotate is 1.5^{o}. In the following figures, the gray arrows represent the robot position and heading before the command, and the black arrow shows the net motion caused by the command. The gray circle is the center of the robot before the command. The command notation describes the left and right stepper actions, as listed in Table 2. For example, **(F,h)** means make the left motor move a full step forward at the same time as making the right motor move a half step backward.

Code	Stepper motor action
F	full step forward
H	half step forward
0	no change on this stepper
h	half step backward
f	full step backward

Table 2. There are five possible stepper motor actions.

The legal actions are shown in Figure 5. For example, if the four-bit stepper output is currently 10, and you change it to 6, then the motor will perform a full-step backwards, shown as bold in the figure. Be careful when mixing full-step and half-step commands. For example, changing the stepper output from 1 to 8 is illegal. Outputting illegal stepper commands will cause robot damage. For example, writing 0 (%0000) is an illegal operation.

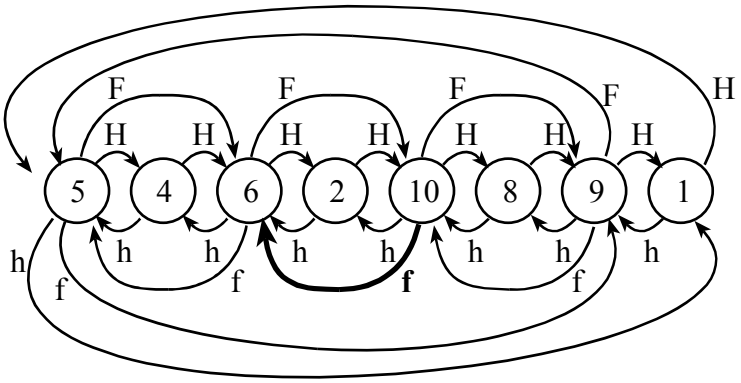


Figure 5. A state graph showing the legal stepper motor actions.

The first four commands are simple translations, without rotation, as shown in Figure 6.

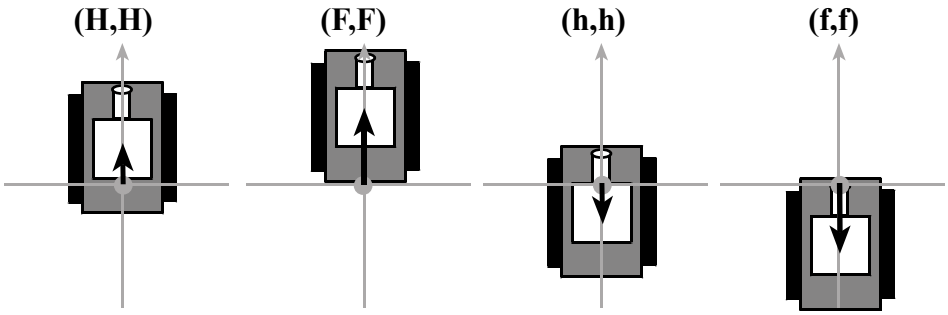


Figure 6. To move the robot in a straight line step both motors together.

The second four commands are simple rotations, without translation, as shown in Figure 7. A counter-clockwise (CCW) rotation adds to the robot compass heading, while a clockwise (CW) rotation decreases the value of the heading. A collision is possible after a pure rotation because the simulation checks for overlapping volumes. The four full-step motions are listed in Table 3.

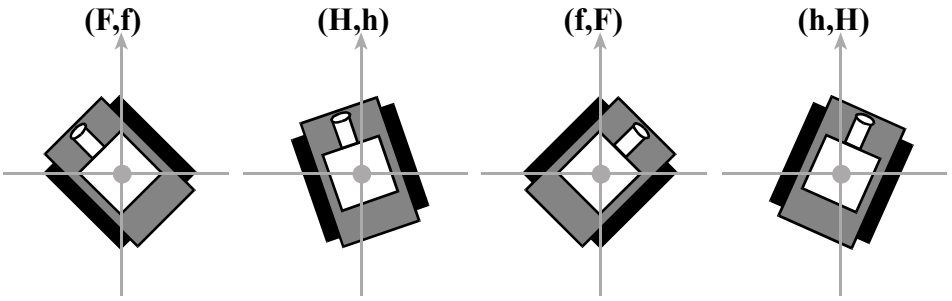


Figure 7. To rotate the robot step the track motors in opposite directions.

Command	Left track	Right track	Δθ	Δx	Robot motion
(F,F)	full-step forward	full-step forward	0	8	forward
(f,f)	full-step backward	full-step backward	0	-8	backward
(F,f)	full-step forward	full-step backward	-6	0	turn CW
(f,F)	full-step backward	full-step forward	6	0	CCW

Table 3. These four full-step commands are sufficient to move the robot.

It takes 15 **(F,f)** commands to turn the robot 90°. The four half-step motions are listed in Table 4. It takes 30 **(H,h)** commands to turn the robot 90°.

Command	Left track	Right track	Δθ	Δx	Robot motion
(H,H)	half-step forward	half-step forward	0	4	little forward
(h,h)	half-step backward	half-step backward	0	-4	little backward
(H,h)	half-step forward	half-step backward	-3	0	little CW
(h,H)	half-step backward	half-step forward	3	0	little CCW

Table 4. The half-step commands provide finer control of the robot.

There are a total of sixteen commands that cause a combined rotation and translation. Figure 8 shows four of these more complex movements. The white circle is the pivot point of a motion that involves both a translation and a rotation. The combined motions are listed in Table 5. The simulation first rotates the robot by the amount shown in Table 4, then it translates robot in the direction of the new heading.

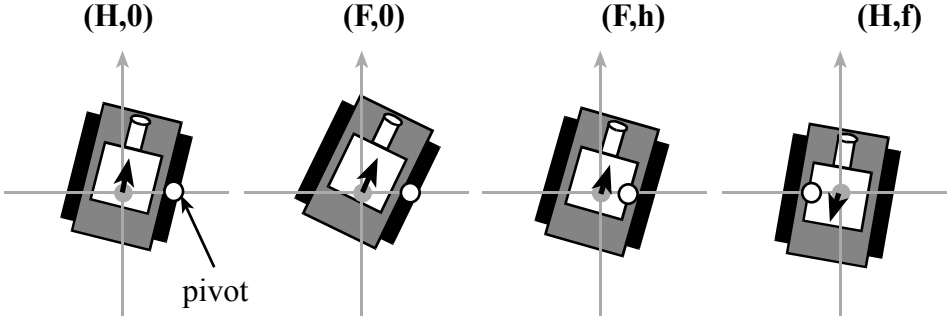


Figure 8. Four of the sixteen commands that result in both a rotation and a translation.

Command	Left track	Right track	Δθ	Δx	Robot motion
(F,h)	full-step forward	half-step backward	-4.5	1	tiny forward, CW
(H,f)	half-step forward	full-step backward	-4.5	-1	tiny back, CW
(h,F)	half-step backward	full-step forward	4.5	1	tiny forward, CCW
(f,H)	full-step backward	half-step forward	4.5	-1	tiny back, CCW
(F,H)	full-step forward	half-step forward	-1.5	3	little forward, little CW
(H,F)	half-step forward	full-step forward	1.5	3	little forward, little CCW
(f,h)	full-step backward	half-step backward	1.5	-3	little back, little CCW
(h,f)	half-step backward	full-step backward	-1.5	-3	little back, little CW
(F,0)	full-step forward	none	-3	2	turn CW about right
(0,F)	none	full-step forward	3	2	turn CCW about left
(f,0)	full-step backward	none	3	-2	turn CCW about right
(0,f)	none	full-step backward	-3	-2	turn CW about left
(H,0)	half-step forward	none	-1.5	1	little CW about right
(0,H)	none	half-step forward	1.5	1	little CCW about left
(h,0)	half-step backward	none	1.5	-1	little CCW about right
(0,h)	none	half-step backward	-1.5	-1	little CW about left

Table 5. There are sixteen commands that result in both a rotation and a translation.

A third stepper motor controls the angle of the gun turret. Forward steps rotate the turret CCW, while backward steps rotate the turret CW, as shown in Figure 9 and Table 6. The turret angle is relative the robot heading. The absolute turret angle is the sum of the robot heading and the turret angle.

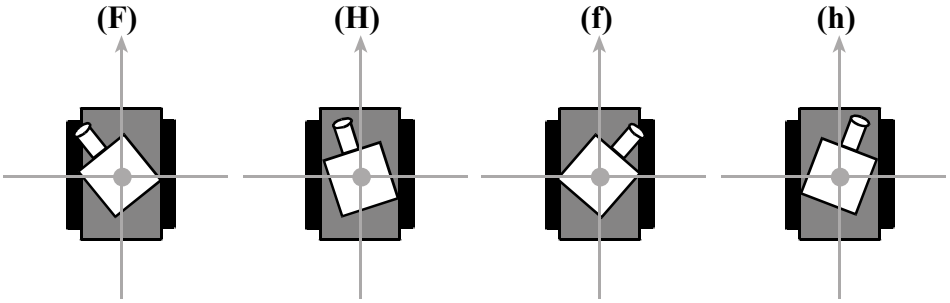


Figure 9. A third stepper motor independently controls the turret angle.

It takes 36 **(F)** commands to rotate the turret a complete 360 ^{o}

Command	Stepper motor action	Change in turret angle
(F)	full step forward	+10^{o}
(H)	half step forward	+5^{o}
(h)	half step backward	-5^{o}
(f)	full step backward	-10^{o}

Table 6. The third stepper controls the turret angle.

Analog status parameters provide feedback to your software, indicating results of the scanner. The status of your robot includes the percent damage, the location on the battlefield, the heading of the robot, and the heading of the gun turret. There are three ultrasonic range sensors to determine the distance to the closest enemy robot. These sensors are interfaced to the ADC.

A robot is considered dead when the health drops to 0%. A collision occurs when the bounding boxes of two objects overlap. Damage to health is inflicted as follows:

1% - collision caused by another robot running into you.
5% - collision into another robot or into a wall.
10% - a missile hitting your robot.
10% - a software bug or illegal stepper output.

A collision will cause the robot to bounce, changing directions after the impact. Damage is cumulative; however, a robot does not loose any mobility, or fire potential at high damage levels. In other words, a robot at 1% health performs equally well as a robot with no damage. There are four life-packs (circular targets) in the battlefield. Rolling over the pack grants you a 50%

increase in health, but each pack may be used only once per run. The maximum health is 100%. The life-pack triggers are 50m by 50m squares. For purposes of the competition, a score is maintained, separate from the health

20 point bonus when one of your missiles hits other robot
1 point penalty for launching a missile
5 point penalty when your robot is hit by a missile
5 point penalty when your robot has a collision.

A robot-robot collision causes both robots to loose points, but the robot initiating contact looses more health.

Analog sensors connected to 9S12C32 PAD0 to PAD7

The numbers in parentheses are 10-bit right-justified ADC values.

Channel 0: Current x-position of the center of robot (the pivot point). Near 0V (0) means the robot hit the West wall. Near 5V (1023) means the robot hit the East wall. See Figure 2.

Channel 1: Current y-position of the center of robot (the pivot point). Near 0V (0) means the robot hit the North wall. Near 5V (1023) means the robot hit the South wall. See Figure 2.

Channel 2: Current compass heading of the robot. The compass system is oriented so that due East (right) is 0V, 1.25V (256) is North, 2.5V (512) is West, 3.75V (768) is South. Just below due East is 5V (1023). See Figure 10.

Channel 3: Current heading of the turret/cannon relative to the robot. The heading is defined so that straight ahead (same direction as the robot is 0V, 1.25V (256) is to the left, 2.5V (512) is directly behind, 3.75V (768) is to the right. Just a tiny bit to the right is 5V (1023).

Channel 4: Results of the left scanner is the range to closest enemy robot within a cone-shaped volume to the left of the canon. 5V (1023) means no enemy robot within 256 meters. 2.5V (512) means 128 meters to the enemy robot.

Channel 5: Results of the front scanner is the range to closest enemy robot within a cone-shaped volume in the direction of cannon turret heading. 5V (1023) means no enemy robot within 256 meters. 2.5V (512) means 128 meters to the enemy robot.

Channel 6: Results of the right scanner is the range to closest enemy robot within a cone-shaped volume to the right of the canon. 5V (1023) means no enemy robot within 256 meters. 2.5V (512) means 128 meters to the enemy robot.

Channel 7: Current robot damage. 5V (1023) means full life and 0V (0) means dead.

It is possible to sample all 8 sensors with one 6812 command using 8-channel sequence length and multiple channel mode. However, continuous mode is not supported.

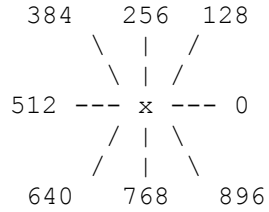


Figure 10. Compass directions are measured in degrees from East.

Two Stepper Motors Control Robot Motion

PT7-PT4 Right Track Stepper Motor
PT3-PT0 Left Track Stepper Motor

Sensor Resolution

PM5-PM4 00,01,10,11 is 5, 10, 30, 120 degrees respectively

Gun Control

PM3-PM0 Gun Turret Direction Stepper Motor
PS1 serial port, send a serial output frame to shoot a missile

9600 bits/sec baud rate, 1 start, 8-bit data, 1 stop frame protocol
8-bit data specifies the missile speed 0 to 255, range = about 0.7meters*data

An empirical experiment yielded the data in Table 7 at MissileSpeed=10.

Firing Strength	Range of targets that hit (very rough estimate)
100	80?? to 104 meters (?? Could be closer)
150	96 to 124 meters
200	112 to 160 meters

Table 7. The target robot is oriented sideways, lined up exactly at the correct firing angle.

Programming limitations
The 9S12C32 memory map and the following list of I/O ports are simulated. RTI and eight output compare interrupts are allowed; other interrupts are not allowed. The TCNT rate can not be changed.

```
DDRM:     equ $0252  ; Port M Data Direction Register 6 bits
DDRT:     equ $0242  ; Port T Data Direction Register
PTM:      equ $0250  ; Port M I/O Register 6 bits
PTT:      equ $0240  ; Port T I/O Register
TIOS:     equ $0040  ; Timer Input Capture/Output Compare Select
TCNT:     equ $0044  ; Timer Count Register
TSCR1:    equ $0046  ; set bit 7=1 to enable TCNT
TFLG1:    equ $004E  ; Main Timer Interrupt Flag 1
TIE:      equ $004C  ; Timer Interrupt Enable Register
TC0:      equ $0050  ; Timer Output Compare Register 0
TC1:      equ $0052  ; Timer Output Compare Register 1
TC2:      equ $0054  ; Timer Output Compare Register 2
TC3:      equ $0056  ; Timer Output Compare Register 3
TC4:      equ $0058  ; Timer Output Compare Register 4
TC5:      equ $005A  ; Timer Output Compare Register 5
TC6:      equ $005C  ; Timer Output Compare Register 6
TC7:      equ $005E  ; Timer Output Compare Register 7
SCIBD:    equ $00C8  ; 16-bit SCI Baud Rate Register
SCICR2:   equ $00CB  ; SCI Control Register 2
SCIDRL:   equ $00CF  ; SCI Data Register Low
SCISR1:   equ $00CC  ; SCI Status Register 1
ATDCTL0:  equ $0080  ; ATD Control Register 0
ATDCTL1:  equ $0081  ; ATD Control Register 1
ATDCTL2:  equ $0082  ; ATD Control Register 2 no interrupts, no fast clear
ATDCTL3:  equ $0083  ; ATD Control Register 3
ATDCTL4:  equ $0084  ; ATD Control Register 4
ATDCTL5:  equ $0085  ; ATD Control Register 5 no SCAN mode
ATDDR0:   equ $0090  ; A/D Conversion Result Register 0
ATDDR1:   equ $0092  ; A/D Conversion Result Register 1
ATDDR2:   equ $0094  ; A/D Conversion Result Register 2
ATDDR3:   equ $0096  ; A/D Conversion Result Register 3
ATDDR4:   equ $0098  ; A/D Conversion Result Register 4
ATDDR5:   equ $009A  ; A/D Conversion Result Register 5
ATDDR6:   equ $009C  ; A/D Conversion Result Register 6
ATDDR7:   equ $009E  ; A/D Conversion Result Register 7
ATDSTAT0: equ $0086  ; A/D Status Register 0
ATDSTAT1: equ $008B  ; A/D Status Register 1
```

Enhanced 9S12C32 Memory map

```
Internal RAM     $3800-$3FFF           ; 2K globals and stack
Shared RAM       $8000-$BFFF           ; 16K shared space with your team (not active)
Internal EEPROM  $4000-$7FFF,$C000-$FFFF ; 32K for constants and program
```

Stack overflow and stack underflow will cause memory errors. The PLL loop can not be altered to change the bus cycle time. Table 8 shows the complete list of 6812 simulation faults (bugs in your software) that can occur. Each fault will cause a 10% loss of health.

10 op code failure
11 Port E is read only
12 PORTAD is read only
13 ADR's are read only
14 ATDSTAT is input only
15 ATDCTL2 bits 6-1 are not implemented
16 ATDCTL3 FIFO mode is not implemented
17 ATDCTL5 scan mode not implemented
18 SCIBDH bits 7,6,5 not implemented
19 Only SCICR1=0 mode implemented
20 TCIE, ILIE, RWU, SBK not implemented
21 SCISR1 is read only
22 SCISR2 is read only
23 SCI 9-bit data mode not implemented
24 SCI receiver not implemented
25 Track stepper motor fault
26 Turret stepper motor fault
27 Lost data, write to TDR when TE=0
28 Lost data, write to TDR when TDRE=0
29 Read from unimplemented I/O port
30 Write to unimplemented I/O
31 Read from uninitialized RAM
32 Read from uninitialized External RAM
33 Read from unprogrammed EEPROM
34 Read from unprogrammed ROM
35 Read from undefined address
36 Write to EEPROM
37 Write to ROM
38 Write to undefined address
39 Executed unimplemented opcode
40 Executed unimplemented indexed mode
41 SCI baud rate mismatch
42 TCNT is read only
43 Tank is outputting to the track steppers too fast.

Table 8. Each time of these program errors occur there will be a 10% loss of health.

Esc	Menu
Space	Pause/Resume
Arrow keys	Camera Position
PageUp PageDown	Camera Angle
F1	Help
F5	Change Screen Resolution
F6	Screenshot
F7	Toggle camera mode
F8	Switch to next player
F12	Toggle sound/music
F11	Abort this run, and start another run
F10	Exit

Table 9. Hot keys available during TRobot simulation.

Schedule of Events, Spring 2008
Lab 9 is the initial design of tank motion
0) Download the lasted version of TRobot. Find a computer with DirectX 9.0c or later, unzip the package and test the system. You can determine the DirectX version by executing **DxDiag** from the command line. You should use **TExaS** to develop the software, creating S19 object files, and use the TRobot simulator to run the battle.
http://users.ece.utexas.edu/~valvano/EE319K

1) Form a group. You may work alone, or in a group of two. Email your TAs the names of all group members. You do not have to have the same TA. They will email you back a code name, which will consist of one letter (A-Z) followed by one number (0,1,2,3). For example, if you are team X1, you will create your solution as file X1.RTF.

Your tank will be run with 24 other tanks. The initial locations of all 25 tanks will random. Your goal is to move your tank to the center location (on or near location 512, 512) without hitting any walls or hitting other tanks. The other 24 tanks will not move or fire. In lab 9, your tank is not allowed to fire missles. You will demonstrate your Lab 9 solution to your TA running it 5 times. You will get full credit on the performance part of Lab 9 if your tank can move to the center and stop every time without hitting any walls or hitting any other tanks. No tank should be placed at the center location initially.

Lab 10 is the TRobot competition
2) Practice. TBA: At the start of each hour, the TA will collect S19 records from whoever is present and run them together in a practice competition. There are no formal lab demonstrations required for lab 10.

3) Program submission. Contact your TA for due dates: You must email your TA with your RTF (source code) file attached. For example, if you are team X1, then send your TA your file X1.RTF. Place the following phrase in the email title "EE319K TRobot submission". Make sure it assembles in **TExaS**. The TA will email you an acknowledgement. If you do not get the acknowledgement, put your RTF and S19 on a USB drive and bring it to your TA. Great confusion and sadness will occur if you send your TA two copies of your program.

4) Preliminary Contest. Contact your TA for due dates. There will be about 10 runs of 250ms duration. The top 50%, as scored by the total points of all the runs, will be invited back for the final competition.

5) Program submission. Contact your TA for due dates: If you want, you may email your TA with an updated RTF (source code) file attached. Place the following phrase in the email title "EE319K TRobot submission". The TA will email you an acknowledgement. If you do not get an acknowledgement, put your RTF and S19 on USB drive and bring it to the finals.

6) Final Contest Contact your TA for due dates: Note: there will be about the same number robots in the battlefield.

The grading scale for the TRobot Lab, which will count as two lab grades, is as follows

100	>50% scoring rank during preliminaries and software has good structure and style.
90	>50% scoring rank during preliminaries and software has poor style.
85	25-50% scoring rank during preliminaries and software has good structure and style.
75	25-50% scoring rank during preliminaries and software has poor style.
75	0-25% scoring rank during preliminaries and software has good structure and style.
50	0-25% scoring rank during preliminaries and software has poor style.

The **Trobot.ini** file allows you to change many parameters of the battle. The parameters are grouped into three categories. The first category simply affect on-screen displays and have no effect on the actual battle conditions. The student is free to adjust these parameters however they wish. The tanks are numbered A0,B0,C0… For example, **`Player=27;`** specifies the chase camera follows tank B1. The camera view can be adjusted during the game, but the **`Camera`** parameter specifies the initial view. The **`Show`** parameters determine whether or not that particular variable is visible on the screen during battle. There are two fonts available for the display. **`BigFont=1`** is appropriate for the in-class competition, however **`BigFont=0`** is appropriate for debugging on your personal computer. If you set your tank to the player, then the simulation will halt every time your software executes a **`stop`** instruction. You can restart simulation by pressing the space bar.

```
Player=0;      // Tank A0 is has cameras attached
Camera=5;      // 5=panoramic spin, 4=manual,  2=orbit, 0=1st
ShowX=0;
ShowY=0;
```

```
ShowDir=0;
ShowTurret=0;
ShowLeft=0;
ShowCenter=0;
ShowRight=0;
ShowHealth=1;
ShowPC=0;
ShowRegX=0;
ShowRegY=0;
ShowRAM=0;
ShowScore=1;
BigFont=1;            // 0 is little font, 1 for large green font
PauseBetweenGames=0; // execute all simulations without pausing
StopWillPause=1;     // executing stop will pause for the Player
TankCreateLogFile=1; // save results in logFile.txt
```

The second set of parameters will be determined by the instructor, but you can adjust these parameters during initial debugging for your convenience. However, the values of these parameters will be set by the instructor.

```
NumberOfBattles = 10;   // competition consists of 10 battles
BattleTime = 150;       // each battle is 150 ms simulation time
```

This third set of parameters is determined by the instructors of the course and should not be modified by the student. One parameter the instructor can modify is the **MissileSpeed**. The missile speed affects both the speed and range of a missile. A nonzero value stored into the parameter **MinStepTime** forces the tank to wait in between outputs to the track stepper motors. If the tank tries to move too fast, it will throw "43" errors and loose 10% health for each violation (motors burn up). A **MinStepTime** value of 0 means there is no speed limit. The **BaudRate** specifies the SCI baud rate, thus it determines the maximum rate at which the tank can fire. You should output to both track steppers simultaneously, because after writing to **PTT**, you have to wait another **MinStepTime** cycles before you can write to **PTT** again.

```
MissileSpeed=25;  // initial velocity of missiles (5 to 100)
MinStepTime=700;  // bus cycles between outputs to PTT
BaudRate=2400;    // SCI rate in bits/sec
```

How to develop assembly programs using Metrowerks/Tech Arts board

First, you need to install Metrowerks CodeWarrior for HC(S)12. You can go to
http://users.ece.utexas.edu/~valvano/S12C32.htm
for directions on how to download and install Metrowerks

A) To open an existing Metrowerks project
1) Start Metrowerks CW12 CodeWarrior for HC(S)12
2) Execute File->Open, navigate to an existing *.mcp file, and click "OK"

B) Creating a new 9S12DP512 assembly project

First, you will execute **`File->New`**, select **HC(S)12 NewProjectWizard**, and choose a name and place for the project, as shown in Figure 1. Next, choose the 9S12DP512 microcontroller, as shown in Figure 2.

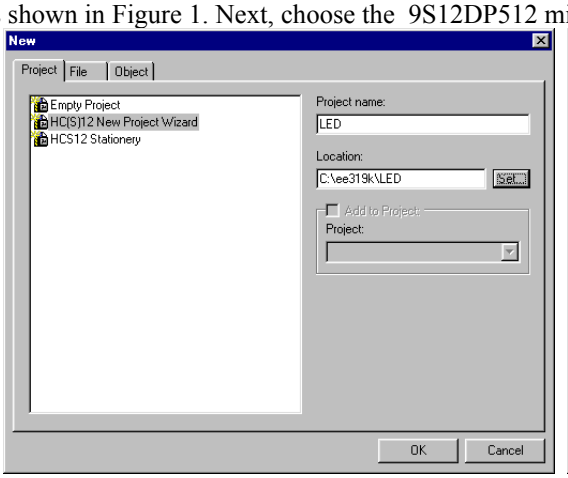


Figure 1. First dialog creating a new project.

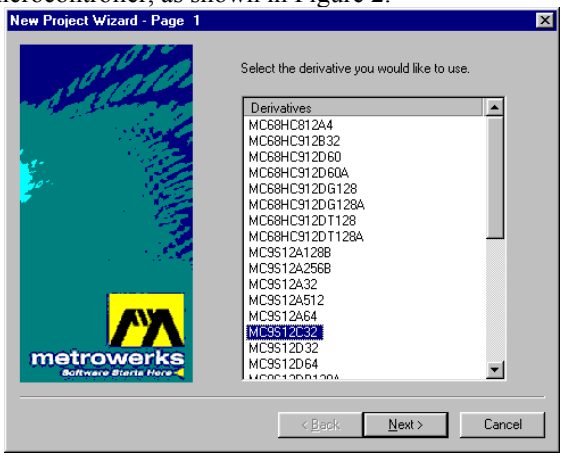


Figure 2. Second dialog creating a new project.

Next, you will select **Assembly** and deselect **C** and **C++.** For simple programs, you should select **Absolute Assembly**, as shown in Figure 4.

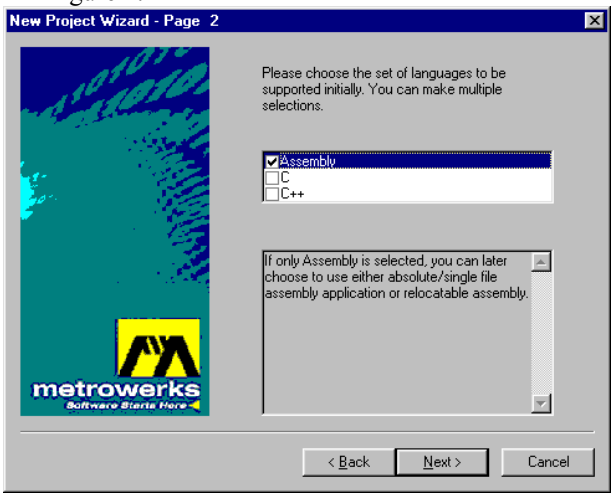


Figure 3. Third dialog creating a new project.

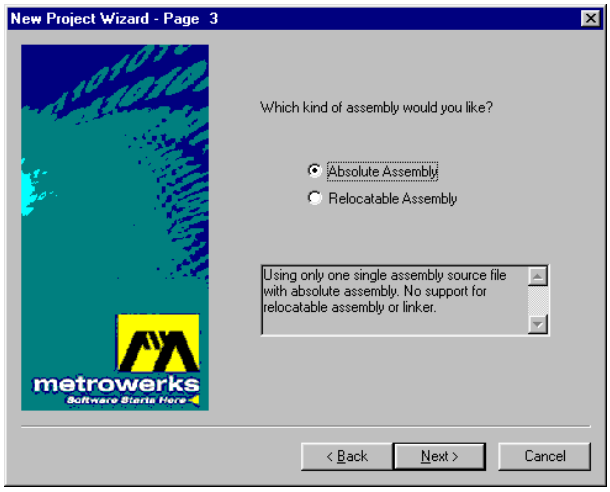


Figure 4. Fourth dialog creating a new project.

Because we have no BDM hardware, you should select **Motorola Serial Monitor Hardware Debugging**, as shown in Figure 5.

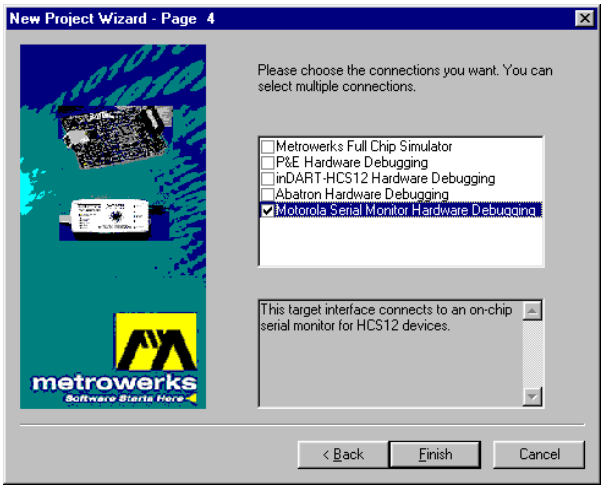


Figure 5. Fifth dialog creating a new project.

In the project window, double click the **main.asm** name to open the source file.

C) How to run Metrowerks on the Real 9S12DP512 board

Do this once

1) Connect PC-COM1 to the 9S12DP512 DB9 connector (any PC COM port will be ok),
2) Place the Run/Load switch on the 9S12DP512 board in Load mode
3) Connect power to 9S12DP512 board.
4) Touch the reset switch on the board

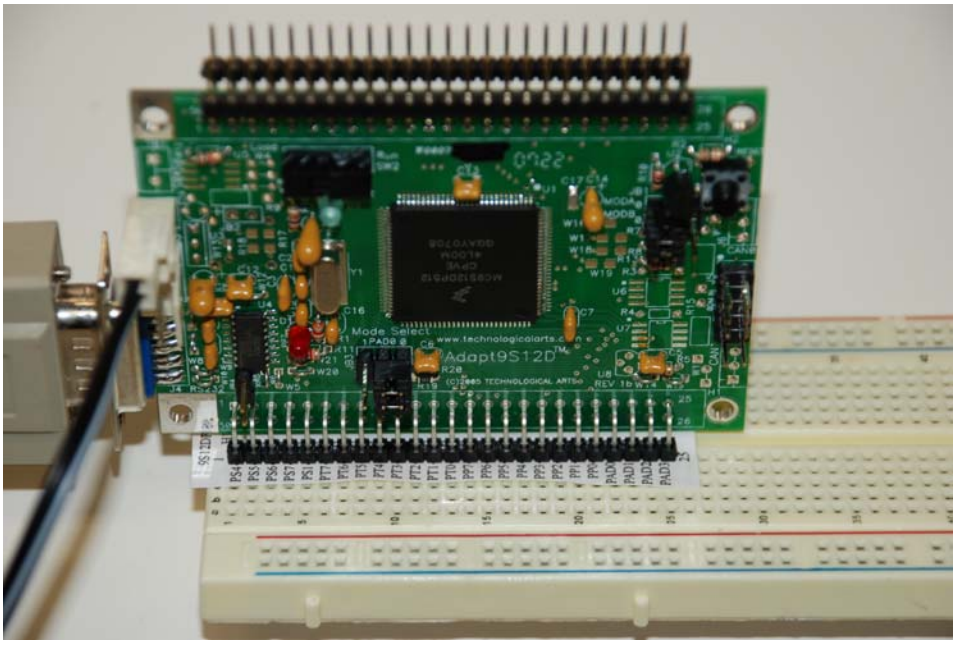


Figure 6. Photograph of the TechArts 9S12DP512 hardware setup.

For each edit/compile/run cycle for software that does not use the SCI

1) In Metrowerks, perform editing to source code
2) In Metrowerks, compile/Link/Load
Execute **`Project->Debug`**
3) Click the green arrow in the debugger to start. Runs at 24 MHz.

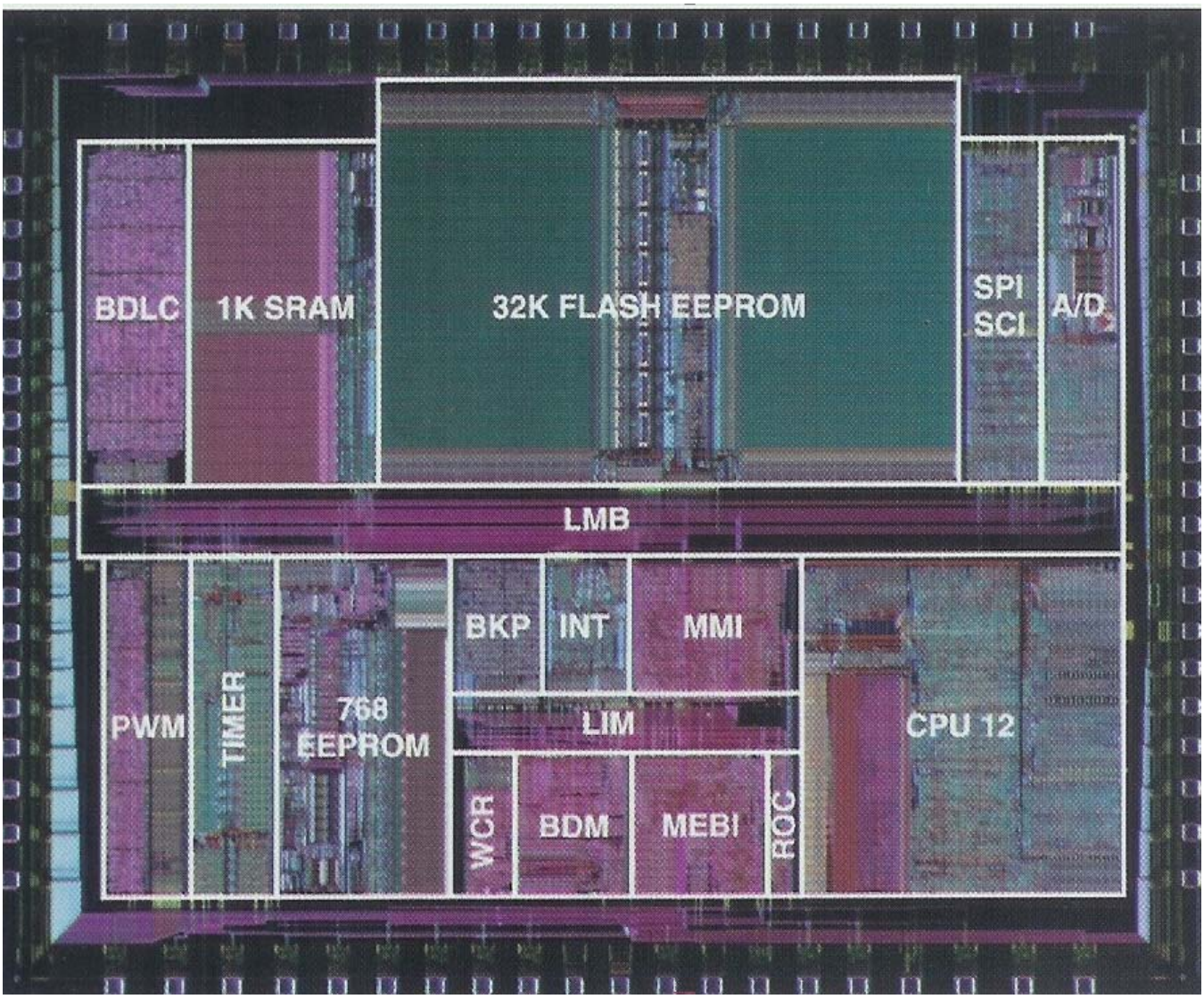
For each edit/compile/run cycle for software that does use the SCI0
1) set the Run/Load switch to Load mode, push the reset button on the 9S12DP512 board
2) execute Project->Debug (compiles and downloads code to 9S12DP512
3) quit MW debugger once programming complete. Quitting the debugger will release the COM port.
4) start a terminal program (like HyperTerminal)
specify proper COM port, 38400 bits/sec, no flow control that matches the 9S12 SCI initialization
5) set the Run/Load switch to Run mode and push the reset button on the 9S12DP512 board. The 9S12DP512 runs at 8 MHz.
6) when done, quit terminal program. Quitting the terminal program will release the COM port.

To run in embedded mode
1) disconnect the serial cable if not needed
2) set the Run/Load switch to Run mode,
3) apply power to the 9S12DP512 board
The 9S12DP512 runs at 8 MHz if you do not modify the PLL.
You can adjust the E clock rate by configuring the PLL

Add this code to your project, we you wish to run at 24 MHz in both Run and Load modes

```
SYNR     equ $0034  ; CRG Synthesizer Register
REFDV    equ $0035  ; CRG Reference Divider Register
CRGFLG   equ $0037  ; CRG Flags Register
CLKSEL   equ $0039  ; CRG Clock Select Register
PLLCTL   equ $003A  ; CRG PLL Control Register

;********* PLL_Init ****************
; Set PLL clock to 24 MHz, and switch 9S12 to run at this rate
; Inputs: none
; Outputs: none
; Errors: will hang if PLL does not stabilize
PLL_Init
  movb #$02,SYNR   ; OSCCLK is Crystal Clock Frequency
  movb #$01,REFDV
; PLLCLK = 2 * OSCCLK * (SYNR + 1) / (REFDV + 1)
  clr  CLKSEL      ; PLLCLK of 24 MHz
  movb #$D1,PLLCTL ; Clock monitor, PLL On, high bandwidth filter
  brclr CRGFLG,#$08,*  ; wait for PLLCLK to stabilize.
  bset CLKSEL,#$80     ; Switch to PLL clock
  rts
```



68HC12 Die Photo

How to develop C programs Metrowerks/Tech Arts 9S12DP512 board

Installing Metrowerks

CodeWarrior Version 3.1 or later will compile programs we need for EE319K/EE345L/EE345M. For version 3.1 you should use their 12K free educational license. There is an installer for Version 3.1 on the CD accompanying the second edition of the EE345L/EE345M textbook.

Follow these steps to install the Special edition of Metrowerks CodeWarrior Version 4.6 (32K free educational license)

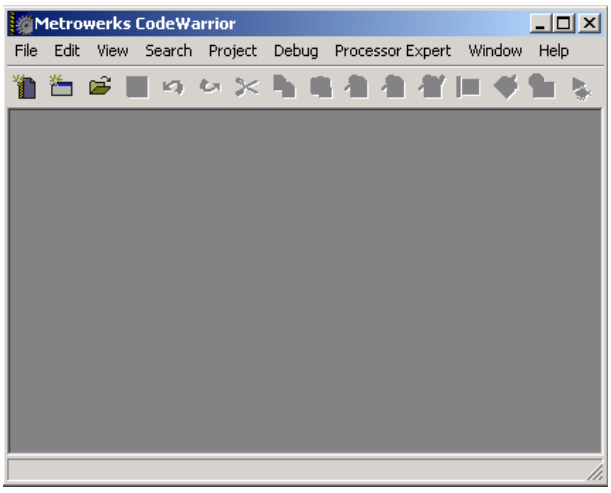
1) http://www.freescale.com/
2) click "**CodeWarrior Development Tools**" under Products
3) click "**HCS12(X)**" under CodeWarrior Products
4) scroll down and click "**Special edition**" for the Special Edition Evaluation for CodeWarrior Development Studio for HCS12X Microcontrollers V4.6 (or whichever version is latest)
5) Register as a new user (if you have registered before, just log in)
   email must be correct
   decide whether or not you want email from Metrowerks
6) Fill in the page with "project details" stating you are a student taking a class, fill in all required fields
7) Download CW12_V4_6.exe (335 MB) and install (the special edition does not require downloading a separate license)
8) Download instructions and starter projects from my web site at
http://users.ece.utexas.edu/~valvano/metrowerks/

A) To open an existing Metrowerks project

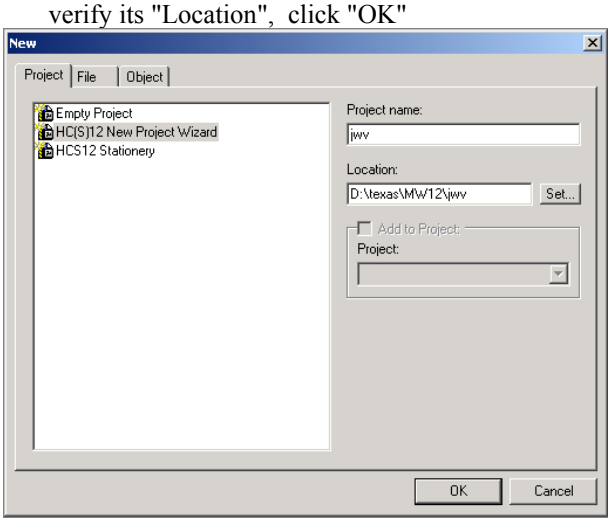
1) Start Metrowerks CW12
2) Execute File->Open, navigate to an existing *.mcp file, and click "OK"

B) How to configure create a new Metrowerks project

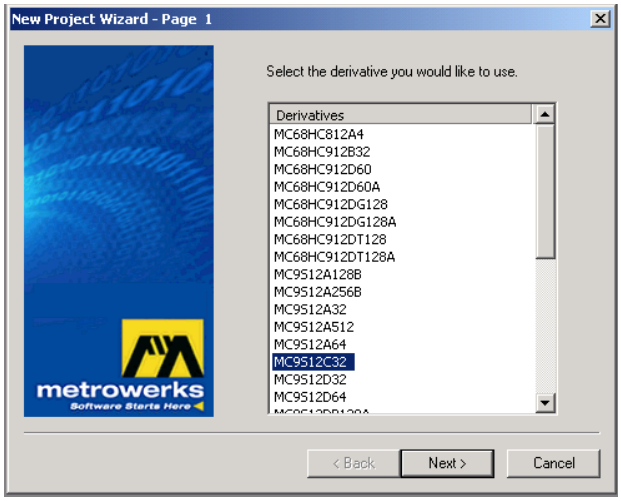
1) Start Metrowerks CW12



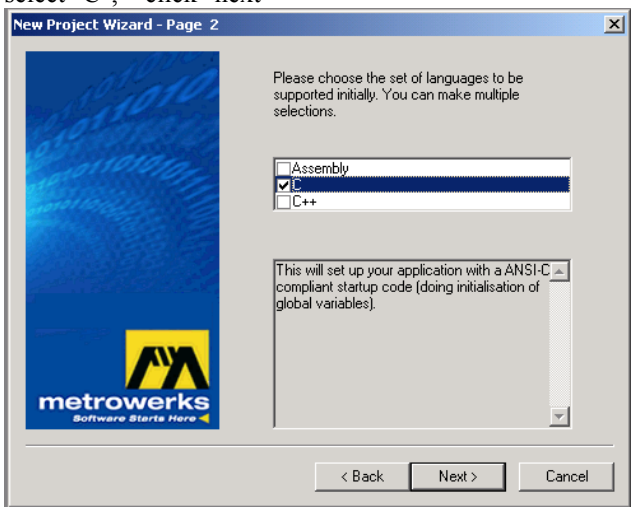
2) Execute File->New, click "Project" Tab
   select "HC(S)12 New Project Wizard"
   specify the "Project name"
   verify its "Location", click "OK"



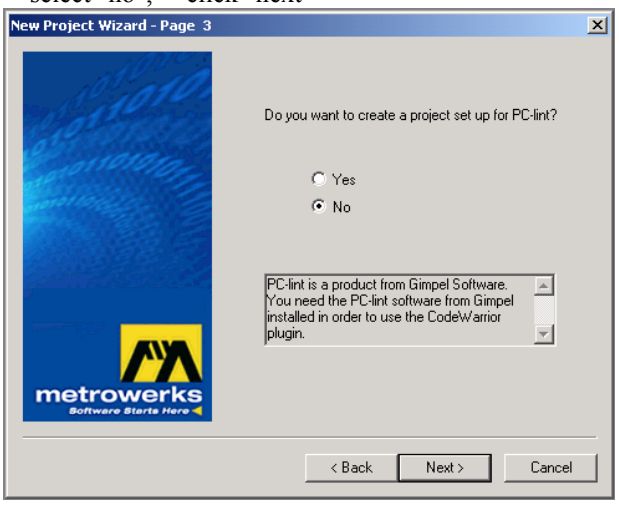
3) Select the derivative you want to use
"MC9S12DP512", click "next"



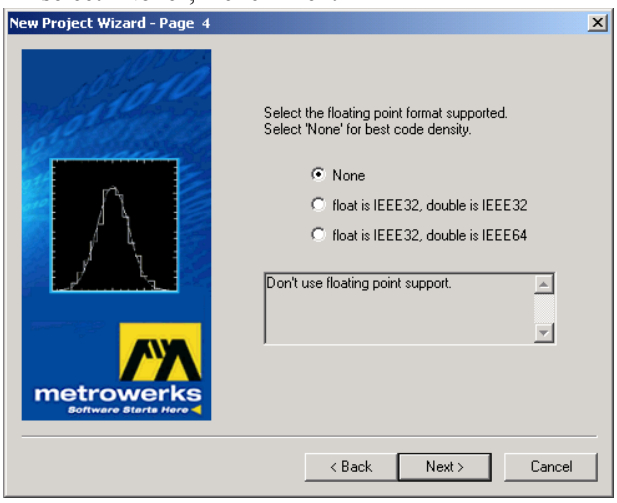
4) Choose the set of languages supported
select "C", click "next"



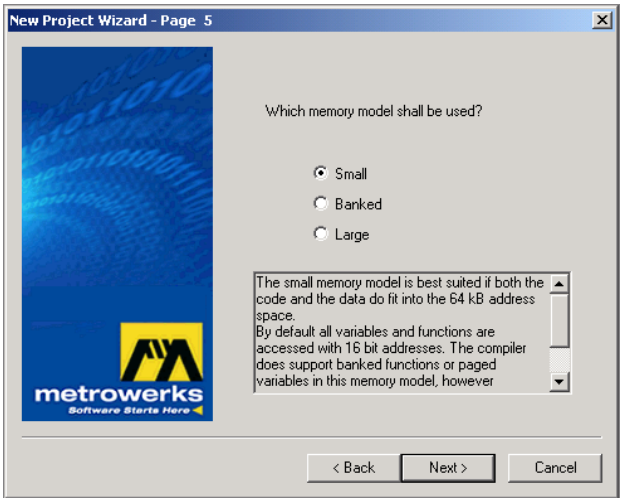
5) Do you want to create a setup for PC-lint
select "no", click "next"



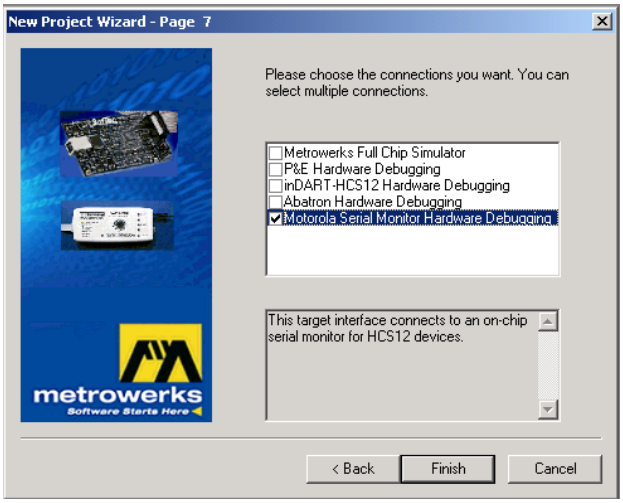
6) Select the floating point format supported
select "None", click "next"



7) Which memory model should be used?
select "Small"
click "next"



8) Please choose the connections you want
select "Serial Monitor Hardware Debugging"
click "Finish"

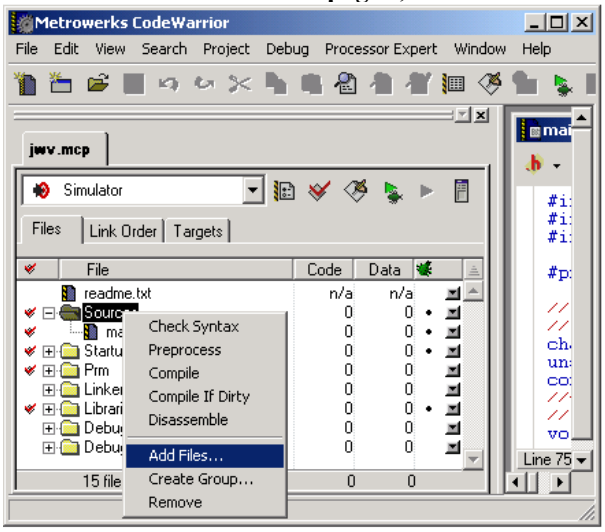


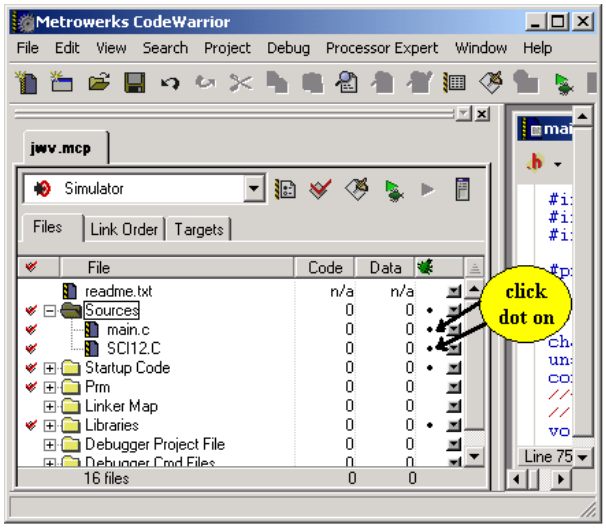
9) Create or copy program files **`*.c`** and **`*.h`**
place them into the "**`Sources`**" directory of your project

10) Add the necessary C files to project

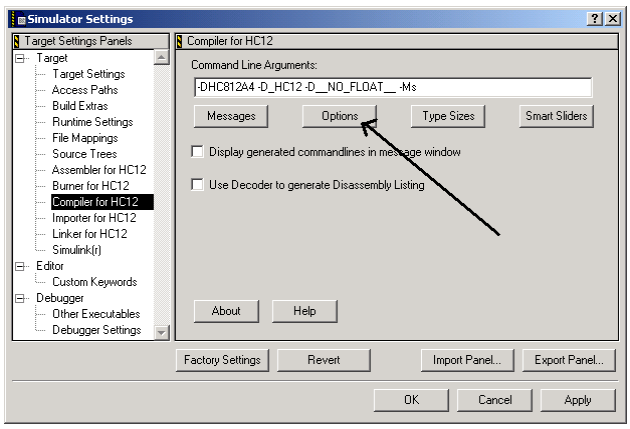
click on Sources in the "**`mcp`**" window
right click and execute "**`Add Files...`**"
"click dot on" in the field associated all C source files under the "bug" icon

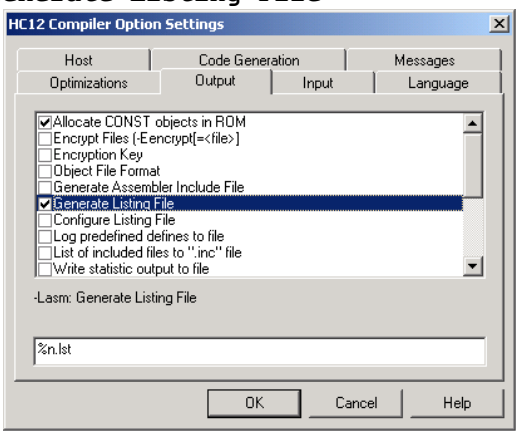
Metrowerks adds the file **datapage.c**, which is not needed. So, this file can be deleted from the project.



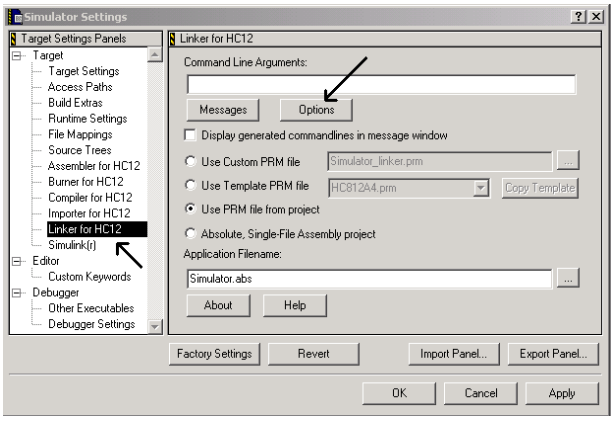


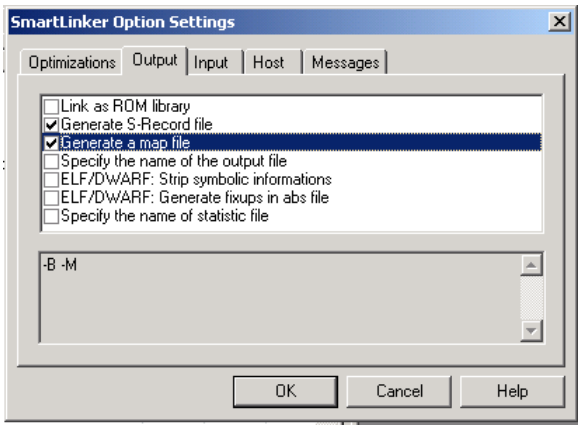
13) Change compiler/linker options
click the right-most toolbar ICON called "**`Simulator settings`**"
click "**`Compiler for HC12`**" choice
click "**`Options`**"
click "**`Output`**" tab
select "**`Allocate CONST objects in ROM`**" and "**`Generate Listing File`**"





click "**`Linker for HC12`**" choice
click "**`Options`**"
click "**`Output`**" tab
select "**`Generate S-Record`**" and "**`Generate a map file`**"

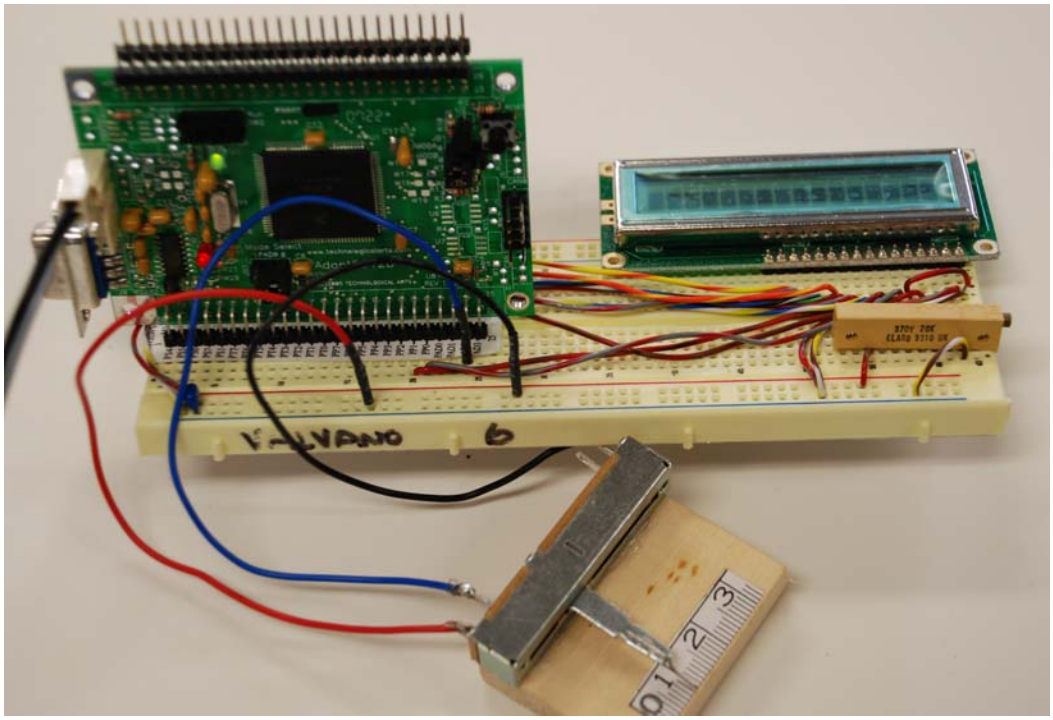




C) How to run Metrowerks C code on the Real 9S12DP512 board

Do this once

1) Connect PC-COM1 to the 9S12DP512 DB9 connector,
2) Place the Run/Load switch on the 9S12DP512 board in Load mode
3) Connect power to 9S12DP512 board using the wall wart that came with the kit.
4) Touch the reset switch on the 9S12DP512 board



For each edit/compile/run cycle for software that does not use SCI

1) In Metrowerks, perform editing to source code
2) In Metrowerks, compile/Link/Load
   Execute **`Project->Debug`**
3) Click the green arrow in the debugger to start. Runs at 24 MHz.

For each edit/compile/run cycle for software that does use SCI

1) set the Run/Load switch to Load mode, push the reset button on the 9S12DP512 board
2) execute Project->Debug (compiles and downloads code to 9S12DP512
3) quit MW debugger after programming is complete. Quitting the debugger will release the COM port.

4) start a terminal program (like HyperTerminal)
specify proper COM port, 38400 bits/sec, no flow control (match the baud rate of your 9S12 software)
5) set the Run/Load switch to Run mode and push the reset button on the 9S12DP512 board.
The 9S12DP512 runs at 8 MHz if you do not modify the PLL.
You can adjust the E clock rate by configuring the PLL
6) when done, quit terminal program. Quitting the terminal program will release the COM port.

To run in embedded mode
1) disconnect the serial cable if not needed
2) set the Run/Load switch to Run mode,
3) apply power to the 9S12DP512 board
The 9S12DP512 runs at 8 MHz if you do not modify the PLL.
You can adjust the E clock rate by configuring the PLL
Add this code to your project, we you wish to run at 24 MHz in both Run and Load modes

```
//********* PLL_Init ****************
// Set PLL clock to 24 MHz, and switch 9S12 to run at this rate
// Inputs: none
// Outputs: none
// Errors: will hang if PLL does not stabilize
void PLL_Init(void){
  SYNR = 0x02;   // OSCCLK is 8 MHz Crystal Clock Frequency
  REFDV = 0x01;
// PLLCLK = 2 * OSCCLK * (SYNR + 1) / (REFDV + 1)
  CLKSEL = 0x00; // PLLCLK of 24 MHz with 8 MHz crystal
// Meaning for CLKSEL:
//  Bit 7: PLLSEL = 0 Keep using OSCCLK until we are ready to switch to PLLCLK
//  Bit 6: PSTP   = 0 Do not need to go to Pseudo-Stop Mode
//  Bit 5: SYSWAI = 0 In wait mode system clocks stop.
//  But 4: ROAWAI = 0 Do not reduce oscillator amplitude in wait mode.
//  Bit 3: PLLWAI = 0 Do not turn off PLL in wait mode
//  Bit 2: CWAI     = 0 Do not stop the core during wait mode
//  Bit 1: RTIWAI = 0 Do not stop the RTI in wait mode
//  Bit 0: COPWAI = 0 Do not stop the COP in wait mode
  PLLCTL = 0xD1;
// Meaning for PLLCTL:
//  Bit 7: CME   = 1; Clock monitor enable - reset if bad clock when set
//  Bit 6: PLLON = 1; PLL On bit
//  Bit 5: AUTO  = 0; No automatic control of bandwidth, manual through ACQ
//  Bit 4: ACQ   = 1; 1 for high bandwidth filter (acquisition); 0 for low (tracking)
//  Bit 3:            (Not Used by 9s12c32)
//  Bit 2: PRE   = 0; RTI stops during Pseudo Stop Mode
//  Bit 1: PCE   = 0; COP diabled during Pseudo STOP mode
//  Bit 0: SCME  = 1; Crystal Clock Failure -> Self Clock mode NOT reset.
  while((CRGFLG&0x08) == 0){ }     // Wait for PLLCLK to stabilize.
  CLKSEL_PLLSEL = 1;  // Switch to PLL clock
}
```