Topic 3 – Interfacing With Internal and External Devices

Required supporting materials

- This Module and any supplementary material provided by the instructor
- Device documentation provided in the appendix of this CoursePack
- CNT MC9S12XDP512 Development Kit and 12 VDC Power Adapter
- BDM Pod and "A to B" USB Cable
- CodeWarrior

Rationale

An embedded controller contains a number of built-in peripherals, seamlessly interfaced through port management logic internal to the device. Some external peripheral devices connected to the microcontroller can be controlled and monitored using single Input/Output (GPIO) lines. Some devices are connected to the controller using serial interfaces. Some devices require parallel busses that operate using one-way communication – either to send data to the peripheral or receive data from that peripheral. Other devices require full two-way parallel communication. In order to operate Embedded Systems, the designer needs to be able to work with all of these possibilities.

Expected Outcomes

The following course outcomes will be partially addressed by this module:

Outcome #1: Develop and debug assembly language programs using an Integrated Development Environment (IDE).

Outcome #2: Create assembly language programs that manipulate data using operations and expressions.

Outcome #3: Interface with onboard, simple GPIO, and programmable devices.

As this course progresses, you will refine the basic skills and understanding of embedded systems and assembly language programming you learn as you complete this topic.

Connection Activity

Some devices are relatively easy to access or control. These devices require no internal programming, and often only require communication in one direction with no feedback. You've already read from a bank of switches and written to three LEDs. Another device on your board, the ICM7218 LED Display Driver, is similar in that it receives simple instructions through a GPIO port, and provides no feedback to the microcontroller.

Many microcontroller devices need, at some point, to send meaningful information to a computer or other communications-enabled device and/or receive meaningful information from such a device. Since the beginning of this course, you've been communicating with your microcontroller through a serial link connected to your computer. This kind of activity very often involves Asynchronous Serial Communication.

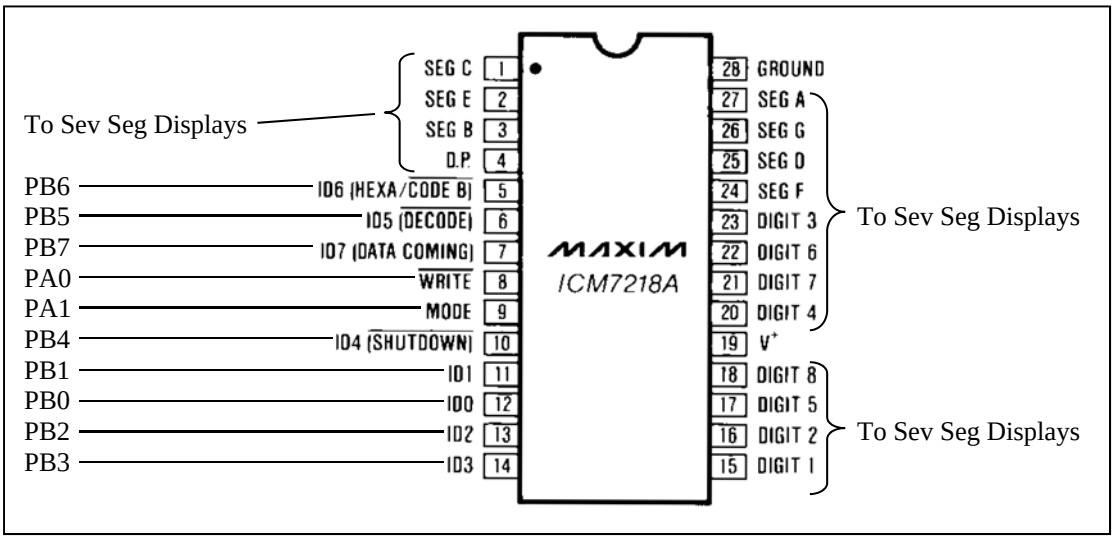
Microprocessors can manage a large number of devices and can transfer a large amount of data quickly because they use parallel communication arrangements. Consequently, many devices – such as memory ICs, banks of LEDs, and pixel array displays, have been designed to operate using parallel communication. However, the microprocessors embedded in most microcontrollers do not directly provide access to the address bus, the data bus, and the various control lines. Instead, designers must "recreate" the necessary interface lines using the general purpose I/O bus pins available on the controller.

Interfacing the ICM7218A 8-Digit LED Display Driver

At this point it is difficult to display program output in a very meaningful way. The 8-digit LED display driver would significantly add output capability to your programs. We will discuss this device now.

The ICM7218A is not a very complicated device, but it does require addressing and command codes to operate. This is more challenging than operating a simple indicator LED, but comes with the benefit of displaying far more useful information.

The first thing to note is how the device is connected to your 9S12. The ICM7218A device requires 8 instruction/data connections, and 2 control signal connections. On your development board, the instruction/data connections have been tied to Port B, and the control signals have been tied to PA0 and PA1 of Port A. All communication to the ICM7218A device will occur through GPIO on these two ports.



The ICM7218A is able to interpret input in a number of ways, and may be commanded to update single digits or update multiple digits. The full operation of the device is beyond the scope of this document, but is something you are encouraged to investigate.

The easiest way to get output on the display is to write a command byte to the device (which contains a digit address) and then write out the data for the digit. Because the device is connected to the 9S12 through GPIO, you must manually produce the correct signals to have this happen. Before any signals may be generated, Ports A and B must first be configured correctly.

You will only be writing to (not reading from) the ICM7218A, so all port pins used should be configured as outputs. It is also important that the active low "/write" line of the device stay high until you actually write to the device. To protect against this, you should set the outputs on the port to HIGH before your set the data direction registers for the ports.

Learning Exercise 1. Start a new text file, and call it library called "SevSeg_Lib.inc". The following is a template for the header, so you've got some idea of where we're headed. Look like a daunting task? Maybe: some of this will be provided, but some of it you will need to figure out on your own.

```
;*****************************************************************
;* Title:         Seven Segment Controller Library               *
;* Processor:     MC9S12XDP512                                   *
;* Xtal Speed:    16 MHz                                         *
;* Author:        This B. You                                    *
;* Date:          Now                                            *
;*                                                               *
;* Contains:      SevSeg_Init                                    *
;*                SevSeg_Char                                    *
;*                SevSeg_dChar                                   *
;*                SevSeg_BlChar                                  *
;*                SevSeg_BlAll                                   *
;*                SevSeg_Two                                     *
;*                SevSeg_Top4                                    *
;*                SevSeg_Low4                                    *
;*                SevSeg_Eight                                   *
;*                SevSeg_Cust                                    *
;*                                                               *
;*****************************************************************
```

Learning Exercise 2. This leads to the first attempt at an initialization subroutine:

```
;*****************************************************************
;*                SevSeg_Init                                    *
;*                                                               *
;*Regs affected:  none                                           *
;*                                                               *
;*Sets up Port A for the Seven Segment Controller as control,    *
;*Sets up Port B for the Seven Segment Controller as data        *
;*  *only b1 and b0 of Port A are used                           *
;*  *clears all eight digits                                     *
;*                                                               *
;*****************************************************************

SevSeg_Init:      BSET  PORTA,%00000011   ;preset A0:1 to HIGH
                  MOVB  #%11111111,PORTB  ;preset all PORTB HIGH

                  BSET  DDRA,%00000011    ;make A0:1 outputs
                  MOVB  #%11111111,DDRB   ;make all PORTB outputs
                  RTS
```

This routine will correctly initialize the ports for communication with the ICM7218A device. However, there's no guarantee as to what will appear on the display digits, so the last line in the header is a lie at this point – we'll revisit this later.

Getting the device to display information can be done in a variety of ways. You have the ability to turn on individual segments of any of the 8 digits, but the easiest thing to do is have the device decode the input as HEX. To get a digit on the display then, you must send a command byte to the device that describes the digit address and mode of operation. After this you must sent the byte value for the digit.

The device will require that the /write line to be lowered then raised to latch the data. This is referred to as "strobing" the /write line. The mode control signal will determine if the byte being written is an instruction or data. So the procedure is as follows:

- Present instruction on GPIO instruction/data lines (Port B)
- Set mode HIGH and write LOW (indicate that you are writing an instruction)
- Set write HIGH (device latches instruction)
- Present data on GPIO instruction/data lines (Port B)
- Set mode LOW and write LOW (indicate that you are writing data)
- Set write HIGH (device latches data)

The bits in the instruction byte control the behavior of the device, and, in the mode we're using, are also used to set the address of the digit being written to. (A note of caution: other manufacturers make versions of this controller that do not allow individual addressing of the digits – in these, all eight digits must be written in a single sequence each time the display is updated. The discussion in this course material is specific to the Maxim part.) Here's a link to the Maxim ICM7218A data sheet:
http://datasheets.maximintegrated.com/en/ds/ICM7218-ICM7228.pdf

Learning Exercise 3. In the data sheet for the Maxim ICM7218A, locate the table of Input Definitions (page 5 in the current version). For standard writing of a hex character, you are interested in Hex mode, Bank A, normal operation. What should the top five bits be?

7	6	5	4	3	2	1	0
					Digit Address		

Learning Exercise 4. Add the following routine to your library. Replace the "xxxxxxxx" with what you just determined as the mask for putting the device into hex, decode, no shutdown, Bank A memory.

```
;*****************************************************************
;*                 SevSeg_Char                                   *
;*                                                               *
;* Regs affected: none                                           *
;*                                                               *
;* Accepts a hex character in Accumulator A and                  *
;* a location (0 to 7) in Accumulator B and places the character  *
;*                                                               *
;* The routine expects the user to know the LED limits,          *
;* and assumes that SevSeg_Init has been run already.            *
;*                                                               *
;*****************************************************************
SevSeg_Char:
            PSHA
            PSHB

            ANDB  #%00000111        ;clean up address
            ORAB  #%xxxxxxxx        ;add control: hex, decode, no SD, Bnk A
            ANDA  #%00001111        ;clear upper nibble
            ORAA  #%10000000        ;no decimal point

            STAB  PORTB             ;instruction ready
            MOVB  #%00000010,PORTA  ;/mode high, write low
            MOVB  #%00000011,PORTA  ;/write back to high

            STAA  PORTB             ;data ready
            MOVB  #%00000000,PORTA  ;mode and /write low
            MOVB  #%00000011,PORTA  ;mode and /write back to high

            PULB
            PULA
            RTS
```

1. If Accumulator B comes with $05, what will it contain, in binary, after the line you just modified above, and why do we want that?

2. Why do we "ORAA #%10000000"?

Learning Exercise 5. Write a short "main" program to check your initialization routine and your SevSeg_Char routine. Try writing different hex codes to each of the eight possible locations (0 to 3 are top line, 4 to 7 are the lower line).

Learning Exercise 6. How could you use BSET and BCLR instead of MOVB to control the /write and mode lines? Try it, and see if it works. Are there any potentially hazardous assumptions made in this version? If so, add built-in protection.

Learning Exercise 7. Do the following to create two new subroutines and also to modify your "init" routine for proper initial blanking.

1. Research the data sheet for the ICM7218A to find out how to blank digits.

2. Write a subroutine called SevSeg_BlChar using the "no decode" mode to blank individual digits, using an address provided in Accumulator B.

```
;*****************************************************************
;*                 SevSeg_BlChar                                 *
;*                                                               *
;* Regs affected: none                                           *
;*                                                               *
;* Accepts a location (0 to 7) in Accumulator B and blanks       *
;* that location.                                                *
;*                                                               *
;*****************************************************************
```

3. Write a subroutine called SevSeg_BlAll using SevSeg_BlChar to blank all eight digits. Use a loop! (Alternatively, you could try the native "write all eight" mode.)

```
;*****************************************************************
;*                 SevSeg_BlAll                                  *
;*                                                               *
;* Regs affected: none                                           *
;*                                                               *
;* Blanks all eight digits.                                      *
;*                                                               *
;*****************************************************************
```

4. Now that you have "SevSeg_BlAll", use it to clean up your "Init" routine.

Learning Exercise 8. Read each of the following headers, and add the necessary code to your library to meet the specifications. After you create each item, test it using your "main" project.

```
;*****************************************************************
;*                 SevSeg_dChar                                  *
;*                                                               *
;* Regs affected: none                                           *
;*                                                               *
;* Accepts a hex character in Accumulator A and                  *
;* a location (0 to 7) in Accumulator B and places the character *
;* followed by a decimal point                                   *
;*                                                               *
;* The routine expects the user to know the LED limits,          *
;* and assumes that SevSeg_Init has been run already.            *
;*                                                               *
;*****************************************************************
```

```
;*******************************************************************
;*                 SevSeg_Two                                      *
;*                                                                 *
;* Regs affected: none                                             *
;*                                                                 *
;* Accepts two hex nibbles in Accumulator A and                    *
;* writes them to two locations starting as specified in Acc B     *
;* if Acc B points to digit 7, the second char should be digit 0   *
;*                                                                 *
;* Relies on SevSeg_Char                                           *
;*                                                                 *
;*******************************************************************

;*******************************************************************
;*                 SevSeg_Top4                                     *
;*                                                                 *
;* Regs affected: none                                             *
;*                                                                 *
;* Accepts four hex nibbles in Accumulator D and                   *
;* writes them to locations 0 to 3 (upper red LED bar)             *
;*                                                                 *
;* Relies on SevSeg_Char                                           *
;*                                                                 *
;*******************************************************************

;*******************************************************************
;*                 SevSeg_Low4                                     *
;*                                                                 *
;* Regs affected: none                                             *
;*                                                                 *
;* Accepts four hex nibbles in Accumulator D and                   *
;* writes them to locations 4 to 7 (lower yellow LED bar)          *
;* Relies on SevSeg_Char                                           *
;*                                                                 *
;*******************************************************************

;*******************************************************************
;*                 SevSeg_Eight                                    *
;*                                                                 *
;*Regs affected:  none                                             *
;*                                                                 *
;*Receives a pointer in the X-Register to an 8-byte memory array   *
;*containing the appropriate 8 hex nibbles as the lower 4 bits     *
;*of each byte, and displays them in order.                        *
;*                                                                 *
;* The memory location must be filled in the Main routine.         *
;*                                                                 *
;* Relies on SevSeg_Char                                           *
;*                                                                 *
;*******************************************************************
```

```
;*******************************************************************
;*                 SevSeg_Cust                                     *
;*                                                                 *
;* Regs affected: none                                             *
;*                                                                 *
;* Accepts custom character specs in Accumulator A and a           *
;* location (0 to 7) in Accumulator B and places the character.    *
;*                                                                 *
;* The routine expects the user to know the LED segments.          *
;*                                                                 *
;*******************************************************************
```

Binary-Coded Decimal Representation and Manipulation

At this point, you're probably aware of two conflicting realities: Your microcontroller only talks hexadecimal; and the bulk of humanity works with decimal numbers.

The cross-over between these two systems is something called Binary-Coded Decimal (BCD), a system that uses hexadecimal coding to represent decimal values. In BCD, the upper six hexadecimal values (ABCDEF) cannot be used. Instead, after 0123456789, the sequence must roll over to 10. Even though the microprocessor still considers this to be 16_{10}, it looks like 10_{10} to the rest of us, and, properly used, will be the BCD representation of 10_{10}. For clarity, we'll use the notation 10_{BCD}.

When it comes to using BCD, you need to be able to represent numbers in BCD, and you may occasionally need to do simple mathematical calculations (addition and subtraction) using BCD numbers. This author's recommendation is that you let your microcontroller do all its work in hexadecimal (true numeric values), then convert the results to BCD when needed for a human interface. Other instructors may feel differently.

Converting Hexadecimal Values to BCD

Remember that decimal values are made up of digits representing the multiples of 10. 3,275 is $3 \times 10^3 + 2 \times 10^2 + 7 \times 10^1 + 5 \times 10^0$. This is true whether 3,275 is represented as $3{,}275_{10}$ or CCB_{16}. So, to convert a hexadecimal number into BCD, we divide it by the largest required multiple of 10, store the integer part of the result as a BCD digit, and divide the remainder by the next multiple of 10, continuing until we reach 10^0.

Let's develop a Hex to BCD routine to handle the biggest BCD result that can be managed by a 16-bit register.

Learning Exercise 9. Do the following using CodeWarrior and your 9S12 kit.

1. What is the biggest BCD value that can exist in 16 bits?

2. What is the true value, in hexadecimal notation, that this represents?

3. Begin a new project, and call it something like "BCDMath".
4. Inside the project, start a "New Text File". Call this "Misc_Lib.inc", and store it under "Libraries" in the directory structure you set up previously.
5. Add the following header. As time goes by, you will add more items to this library – don't forget to update the header each time.

```
;*******************************************************************
;*HC12 Program:Generally useful library components                 *
;*Processor:      MC9S12XDP512                                     *
;*Xtal Speed:     16 MHz                                           *
;*Author:         This B. You                                      *
;*Date:           Whenever                                         *
;*                                                                 *
;*Details:  Subroutines for general consumption:                   *
;*          BCD routines, text management,                         *
;*          Timer initializations, delays, and interval management *
;*Contains:                                                        *
;*          Hex2BCD:    converts numeric values to Binary Coded Decimal *
;*                                                                 *
;*******************************************************************
```

6. Create the following subroutine header, and enter the maximum hex value you determined above.

```
;*******************************************************************
;*                  Hex2BCD                                        *
;*                                                                 *
;*Regs affected:  A and B                                          *
;*                                                                 *
;*A hexadecimal value arrives in Accumulator D and is converted    *
;*to a 16-bit BCD, returned in D                                   *
;*                                                                 *
;*Maximum hexadecimal value allowed is ____                        *
;*                                                                 *
;*******************************************************************

Hex2BCD:
```

7. Start by pushing the current value of X onto the stack, since we will be using X in our manipulation. (We use A and B too, but we will be returning our result in D, so we can't save and restore the current values.)

8. Using the Reference Guide, determine the result for the following, if the D Accumulator initially contains CCB_{16} as an example. (Add these and subsequent lines to your code.)

```
      LDX   #1000
      IDIV
```

9. Since your final result will end up in the D Accumulator, where should the BCD nibble from the previous calculation eventually end up?

10. Explain how the following partially achieves what you said in the previous question. Add comments as you add this to your code.

```
      XGDX
      LSLB
      LSLB
      LSLB
      LSLB
```

11. We're going to use the stack for temporary storage as we proceed. Here's a table for you to use as we do. SP (initial) and SP–1 would contain the return address for the subroutine, stacked low byte first (bottom). Following that, you would have the low byte and high byte of the X-register's original contents, due to the PSHX command above. Show what would be on the stack after the following, and comment this.

```
PSHB
```

SP–8	
SP–7	
SP–6	
SP–5	
SP–4	
SP–3	X register hi
SP–2	X register lo
SP–1	Return addr hi
SP (initial)	Return addr lo

12. What will be in the D Accumulator after the following?

```
XGDX
```

13. What will be on the stack after the following? Add comments.

```
LDX   #100
IDIV
XGDX
PSHB
```

14. What will be on the stack, in the D accumulator, and in the X register after the following?

```
XGDX
LDX   #10
IDIV              ;X contains nibble1, B contains nibble0
XGDX              ;ready to move result to upper nibble
LSLB
LSLB
LSLB
LSLB
PSHB
```

15. Explain how the following places the proper BCD value in the D accumulator:

```
XGDX
ORAB  0,SP        ;nibble0 with nibble1 in B
PULA              ;skip two eight-bit stackings
PULA
PULA              ;get nibble2
ORAA  -2,SP       ;combine with nibble3
```

16. Explain how the previous also returns the stack pointer to a place where we can grab the old value of X and return.

17. Finish off your subroutine with a PULX and an RTS.

18. Write the mini Main program shown below, and step through your subroutine to verify that it produces the correct results. (What are you expecting?)

```
Main:
                LDD         #$0CCB      ;(that's 3275 base 10)
                JSR         Hex2BCD
                bra         *
```

19. You should now have a very workable hexadecimal to BCD converter. As long as the number you send to this routine is within the range you specified, you will get back the proper 16-bit BCD representation, one digit per nibble of the D accumulator. If you want to provide more range to this function, you might want to directly manipulate the "C" (carry) and "V" (overflow) flags in the CCR. "C" could be used to increase the range to 19999_{BCD} (think about that for a while), and "V" could be used to flag the error condition in which the incoming number is too big even for that. This is not a required exercise, but you might find it to be an entertaining diversion.

Manipulating BCD Values using a Hexadecimal-Oriented Microcontroller

It is possible, with the 9S12 microcontroller and its various derivatives, to do simple BCD math. For this, you will need to know the capabilities and limitations of a special instruction called Decimal Adjust A (DAA). The full description of this command is found in the S12CPUV2 Reference Manual, a 450 page book that has been summarized into the Reference Guide you've been using. Here's a link:

http://cache.freescale.com/files/microcontrollers/doc/ref_manual/S12CPUV2.pdf?fpsp=1&Parent_nodeId=1209483269395726476236&Parent_pageType=product

Locate the DAA command (page 161 in the current version).

Learning Exercise 10. Refer to the DAA description in the Reference Manual to answer the following. Keep in mind that the A accumulator is only eight bits.

1. There are ONLY three instructions that can be followed by DAA. What are they?

2. Which accumulator must your result be in, in order to use DAA?

3. What significant limitation is placed on the values used in a calculation preceding the use of DAA?

4. Keeping in mind that the microcontroller only works in hexadecimal, show what the results would be for the following sets of instructions:

   a.
   ```
   LDAA   #$37
   LDAB   #$25
   ABA
   DAA
   ```

   b. (Watch this one – it may save you a lot of time troubleshooting later!)
   ```
   LDAA   #37
   LDAB   #25
   ABA
   DAA
   ```

   c.
   ```
   LDAA   #$37
   ADDA   #$1
   DAA
   ```

   d. Can you INCA before using DAA? What else could you do?

   e.
   ```
   LDAA   #$45
   ADDA   #$92
   DAA
   ```

   f. What ends up in the "Carry" bit for the previous question?

g.
```
LDAA   #$27
ADDA   #$39
DAA
```

h.
```
LDAA   #$53
ADDA   #$99
DAA
```

i. Can you DECA before using DAA? What else could you do?

Switch Management

Switches, as user interface devices, have two complicating features:

- Long Activation Times
- Bounce

Consider a computer keyboard: When you press a key, the mechanical action of that key results in a number of connects and disconnects while the spring mechanism settles down. Once the key is pressed, your finger remains on that key for a certain period of time, which seems quite short to you but could be thousands, even millions of cycles of the computer's clock. How does the keyboard controller "know" that you only intended one instance of that particular keystroke, even though it's aware of multiple quick changes of state followed by thousands of readings of the switch's new condition? Let's work through these problems.

Learning Exercise 11. On a computer loaded with CodeWarrior, write a short program (start from scratch) that goes through the LEDs in sequence (Red/Yellow/Green) when the left button is pressed. Don't add in any delays in your code other than what is necessary to read the switch and turn the LEDs on and off. What do you observe when you run this code?

Detecting Switch Change of State

In order to make the program you wrote in the previous exercise advance just once for each button press, you will need to detect a change of state. In this case, you need to compare the switch's most recent state to its previous condition. As long as the two are the same, you don't want to respond to the switch's condition; only changes in state matter.

For this, we could have a blocking sequence or a non-blocking sequence. A blocking sequence holds up the program until a set of conditions is met; a non-blocking sequence reports the current conditions and allows the program to proceed regardless of those conditions – decisions as to how to proceed are made in the program itself. When it comes to reading switches, you probably don't want your program to be held up as long as the operator holds his finger on a switch if there are other processes occurring. However, if there are no other processes occurring, a blocking sequence may be acceptable.

Learning Exercise 12. Blocking Switch Change Detection: Modify your code from the previous exercise so that it only advances one LED per button press. You will need to detect switch change of state. Use a variable in RAM at $2000 (call it SwState) to keep track of the state. We'll learn more about variables and constants soon.
Hints: Set up the variable in RAM using "SwState: ds.b 1"; initialize the variable by clearing it; load the condition of all the switches; mask off the LEDs; compare to the existing value in SwState; if it has changed, store the new set of conditions in SwState before you move on; otherwise, go back and wait for the switches to change (block).

Learning Exercise 13. Non-blocking Switch Change Detection: For this, you want to change your program so that the LED that's currently lit blinks approximately every 200 ms while the system waits for you to press the switch again. The blinking should occur whether the switch is held down or not.
Hints: Use a subroutine called "SwChange" to check and keep track of the switches; have your subroutine indicate a change in switch condition using the Carry bit in the CCR (Set = change, Clear = no change); the subroutine will return the cleaned-up switches in A; your main program's primary concern will be to keep track of time and blink the current LED.

Debouncing

In the previous two exercises, you probably noticed that the sequence of LEDs sometimes skips ahead by one or two when the switch is pressed. This is due to switch bounce, which we will now address.

The simplest form of debouncing involves detecting a change of switch state, then ignoring all subsequent changes for a period of time that's long enough to ensure that the switch has reached a steady state.

A slightly more reliable debounce sequence involves waiting for a short period of time, then checking to see if the switch is still in the new condition. If not, it must be bouncing – store the condition, wait for another short period of time, and check again. Once the state is consistent from one loop to the next, assume that the switch is stable and continue on.

These two types of debouncing both require a blocking loop – the program is held up in a timing loop while we wait for the switch to settle down. It is possible to design a non-blocking debounce routine which continues to run the main program while it waits for the switch to stabilize, but we won't need to get that complicated in this course. The amount of time we spend in the debounce routine is so small (on the order of 10 ms) that it shouldn't affect the routines we're creating.

Learning Exercise 14. Non-blocking Switch Change Detection with Debounce: Write a "SwCk" subroutine using the debounce routine that is "slightly more reliable" as described above. This routine is general enough that we will soon add it to a library of switch and LED routines, to be used whenever you need to respond to single switch presses.
Hints: Don't use any variables in your subroutine – use A and B as temporary storage locations instead; CBA compares B to A without changing anything; if the switch conditions have changed, wait 10 ms and check again; keep doing this until no further changes are observed before you return from the subroutine with the Carry flag set and the cleaned-up switch conditions in A; make sure B hasn't been changed when you leave the subroutine.

Please note that you don't always need to handle the current state of your switches, nor do you always need to manage switch bounce. If you have a situation in which you want the program to respond as long as a switch is pressed, you will simply want to read the raw condition of the switch repeatedly. Managing bounce or comparing to a previous condition will not only be unnecessary and time-consuming but will produce the wrong results.

Make sure you analyze the system requirements before you decide how you will handle the switches. Key phrases to look out for are

- "for each press of the switch" – requires state checking and debounce
- "while the switch is pressed" – does not require either state checking or debounce: just read the port or check a bit in that port

Learning Exercise 15. Create a library called "SW_LED_Lib.inc" to match the following header. (Hints to follow.)

```
;*******************************************************************
;* HC12 Library:  Switches and LEDs Library Components             *
;* Processor:     MC9S12XDP512                                     *
;* Xtal Speed:    16 MHz                                           *
;* Author:        This B. You                                      *
;* Date:          Now                                              *
;*                                                                 *
;* Details:       Subroutines to initialize PT1AD1 to match CNT    *
;*                development kit, to read and debounce the switches,  *
;*                and turn on LEDs                                 *
;*                                                                 *
;* Routines:                                                       *
;*                                                                 *
;* SW_LED_Init:   initializes ports and registers for switches and LEDs *
;* RedLEDon:      turns Red LED on                                 *
;* RedLEDoff:     turns Red LED off                                *
;* YelLEDon:      turns Yellow LED on                              *
;* YelLEDoff:     turns Yellow LED off                             *
;* GrnLEDon:      turns Green LED on                               *
;* GrnLEDoff:     turns Green Led off                              *
;* SwCk:          returns debounced condition of all Sw's in Acc A     *
;* ChkLeftSw:     returns debounced condition of Left Sw as Carry      *
;* ChkMidSw:      returns debounced condition of Mid Sw as Carry       *
;* ChkRtSw:       returns debounced condition of Right Sw as Carry     *
;* ChkUpSw:       returns debounced condition of Up Sw as Carry        *
;* ChkDnSw:       returns debounced condition of Down Sw as Carry      *
;*                                                                 *
;*******************************************************************
```

Hints:

1. In a new project, create a tiny "Main.asm" that you can use to check each of your subroutines as you develop them (one by one!)
2. Develop your library as a separate file, store it appropriately within your file structure, and include it in your Main.asm.
3. When you get to the switch routines, create the "SwCk" routine first, then use it for each of the individual switches.
4. Use approximately 10 ms as your delay for debouncing the switches.
5. You can move bits into the Carry flag by left– or right–shifting.
6. As much as possible, try to return the registers to their initial condition before exiting a subroutine.
7. Your instructor will be looking for proper documentation – headers, comments, etc.

The Serial Communications Interface

Your 9S12 chip contains SCI (Serial Communication Interface) modules for asynchronous serial communications. You will use one of the SCI modules to communicate with a PC running "terminal emulation" software over a standard RS-232 connection.

Using the PC as a terminal allows you to interact with a color display and a keyboard. This will bring improved I/O to your programs.

Because you will be reading bytes from and writing bytes to the serial port, the SCI module acts as a parallel-to-serial and serial-to-parallel converter. There is an external chip on your microcontroller board that level shifts the signals from the 9S12 (TTL levels) to RS-232 levels (typically around ±10V).

In asynchronous communications, the transmitter may begin a data send to the receiver at any time. Once started, a complete block of data (known as a data character) must be completely transmitted. The delay between data characters may be any length. Transmission of the individual bits in the data character is driven by a clock. The transmitter and receiver must use a clock rate that is approximately equal in order to correctly exchange data. The term "asynchronous" refers to the fact that the clocks in the two pieces of equipment are independent, and communication can be initiated at any time.

The RS-232C standard for serial communication allows for a wide range of signaling characteristics. Here are a few of them:

- Transmission rates vary from 75 baud to 115 200 baud (these must be at clearly-specified speeds only, like 9600, but not 10 000, for example)
- Data can be sent as 7-bit standard ASCII characters, 8-bit extended ASCII characters, or binary data
- Simple error checking, in the form of a Parity Bit, may or may not be activated
- The Parity Bit, if present, can be Even, Odd, always 1, or always 0
- The minimum rest time ("stop bits") between data characters can be adjusted
- "Handshaking" for setting up and maintaining sessions can be configured or ignored

The 9S12 SCI modules are able to send 8-bit or 9-bit data payloads. This provides a fair bit of flexibility:

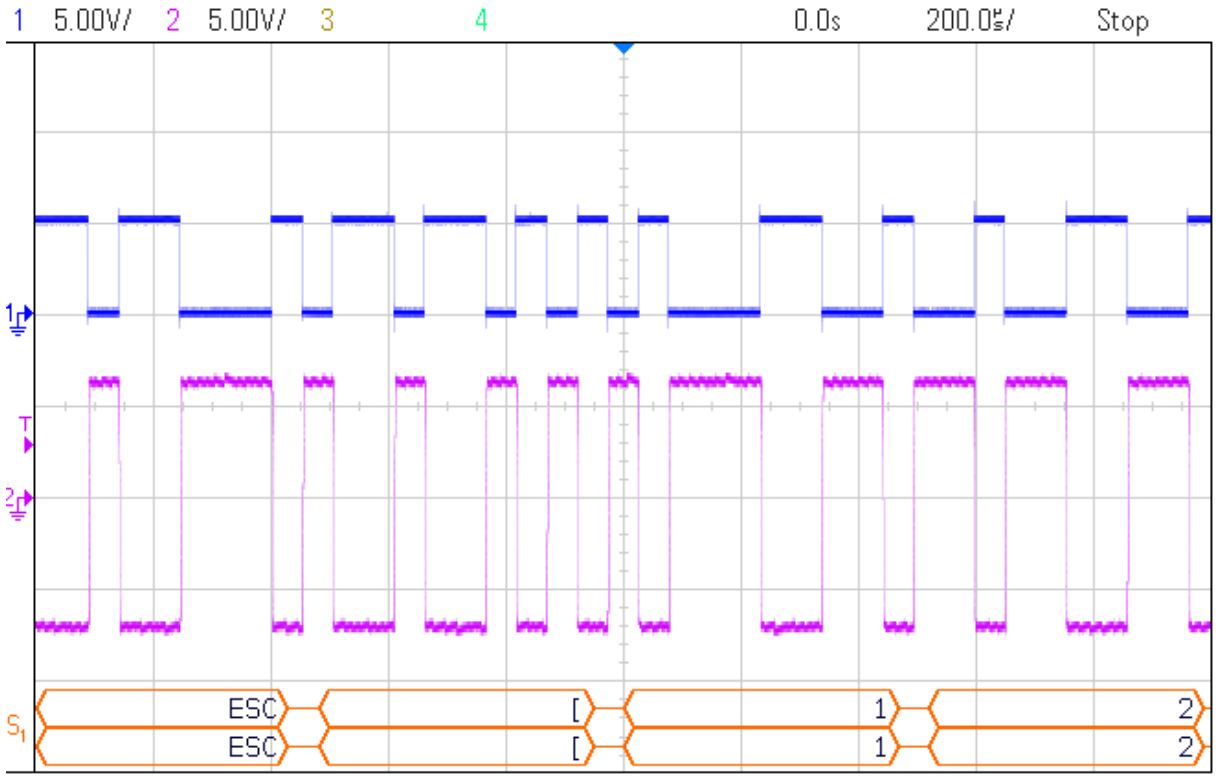
- 9-bit mode provides for 9 actual data bits (very rarely used) or 8 data bits and a parity bit for error checking
- 8-bit mode provides for 8 actual data bits or 7 data bits and a parity bit

Since the 9-bit configurations require us to check two data registers (eight bits in one and the ninth in another), we'll restrict our work to one of the 8-bit modes: 8 actual bits with no parity. The simple error checking made available by the parity bit isn't something we need to concern ourselves with, as, in the lab, we'll be within two metres of the computer we're using as a terminal. If you find yourself in a situation involving greater distance or an electrically-noisy environment, you might consider enabling and checking parity.

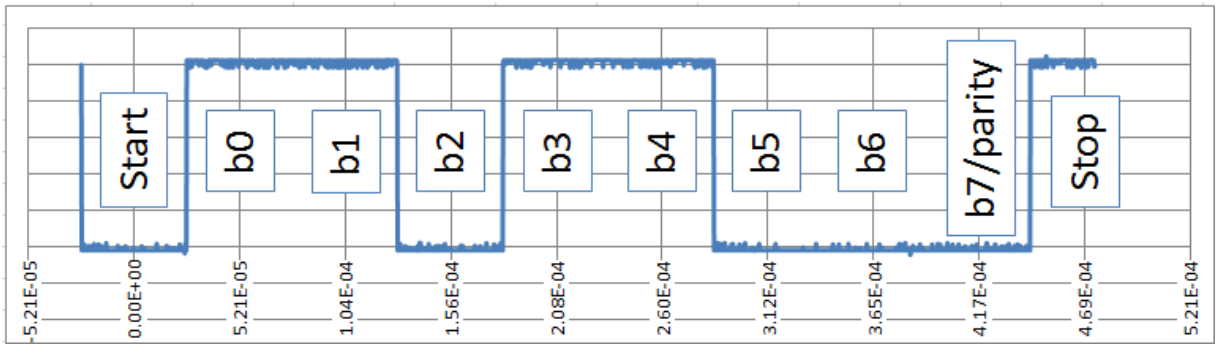
Also, we will be working with the simplest electrical connection possible between our board and the terminal – three wires only: Ground, Transmit Data, and Receive Data. This means that we will not be using the bundle of handshaking wires made available in the RS-232C standard.

When you set up your terminal, select "flow control: none".

The data character sent in this configuration will consist of 1 start bit, 8 data bits (sent LSB first), and 1 stop bit. In terminal language, this is referred to as "8N1" communication – 8 bits, no parity, one stop bit. The start bit signals the start of a data character. The data bits are the data payload. The stop bit signals the end of the data character and is the minimum delay required between data characters. With early communication equipment, the stop bit gave the receiver time to process the received data – this is typically a non-issue these days. In many pieces of equipment, this wait time can be set to 1 bit length, 1.5 bit lengths, or 2 bit lengths. The 9S12's SCI port only offers 1 bit length – that's another thing to remember when you're setting up your terminal. So, once your SCI port is set up to match the conditions above, you will see something like the top trace on the TX pin from the microcontroller, and the bottom trace on the TX pin of the Comm Port.



The following shows the formatting and order of bits, as seen at the microcontroller output.



Note: The TTL level for a mark (logic 1) is +5 V, and 0 V for a space (logic 0). However, these values are approximately –7 V (mark), and +7 V (space) when level-translated to non-return-to-zero RS-232 levels.

The transmitter clocks out serial data at the transmission rate. The receiver samples the line at intervals determined by that clock rate to receive the data. It is critical that the sending and receiving clocks are at the same rate, otherwise the receiver will be sampling the line at the wrong times. In actuality the receiver typically samples the line at a much higher rate and considers multiple samples per bit time to determine the state of each received bit. The 9S12 SCI modules have a sampling rate that is 16 times the bit rate.

The resting state between characters is called “mark idle”, and is a continuation of the stop bit. Therefore, the start bit is always a space, to let the equipment know data is coming.

The number of bits transmitted per second is known as the baud rate. The data rate is actually less, since the framing start and stop bits and the error-checking parity bit, if used, do not contribute to the data payload.

Baud rates for serial communications are relatively slow by today’s standards. The following is a fairly comprehensive list of available baud rates:

- 75
- 110
- 300
- 600
- 1 200
- 2 400
- 4 800
- 9 600
- 14 400
- 19 200
- 38 400
- 57 600
- 115 200

Learning Exercise 16. How long would it take to transfer a 2 GB file at 19 200 baud with 1 start bit, 8 data bits, 1 stop bit, and no parity bit per byte?

Learning Exercise 17. How long does it take to transmit one byte? How many bus cycles does this translate to?

Initializing the Serial Communications Interface

To activate an SCI module on the 9S12, you typically need to configure just three registers. A full description of the activities of these registers is found in chapter 11 of the 9S12 "Data Sheet. Look these up to complete the following exercises.

The first register, SCIBD, controls the baud rate for the module, and requires a 13-bit value as a clock divisor.

The three most significant bits of this 16-bit register can be 0, as you will not be using the infrared configuration for this course. The ultimate baud rate is the bus frequency (8 MHz on your board) divided by 16 divided by the 13-bit value provided in SCIBDH:SCIBDL.

Because integer division might make it impossible to hit the desired baud rate exactly, you will need to select a value that makes the baud rate as close as possible to the target rate. For example, if you wanted to generate a 19 200 baud rate, what value would you put in SCIBDH:SCIBDL?

8000000 / 16 / x = 19200

X = 26.0417

We can't place a value of 26.0417 into the baud register. A value of 26 will provide a baud rate of 19230.8 baud. The oversampling mechanism used by the receiver compensates to some extent for baud rate mismatch – but it has its limits.

As it turns out, the SCI module is somewhat tolerant of clock slippage. The Data Sheet indicates that slow data tolerance (characters arriving slower than expected) is 4.63% and fast data tolerance (characters arriving faster than expected) is 3.75%.

It is suggested that your baud rates not deviate by more than 2%, as the other side of the connection will likely have tolerances to deal with as well.

Learning Exercise 18. What is the percent error between the ideal speed and the closest approximation available on the 9S12 board for each of the following?

a. 19 200 baud

b. 38 400 baud

c. 57 600 baud

Learning Exercise 19. What are the theoretical maximum and minimum baud rates your device can produce? (In other words, don't try to make them fit into the preferred values indicated above.)

The next configuration register to consider is the SCI Control Register 1 – look it up. This register controls the main communications behaviours of the module. We don't want loopback mode, we want the SCI to be enabled in Wait Mode, we want 8 bit data, the device should wake up even on an idle line following a start bit, and we don't want parity checking.

Learning Exercise 20. Determine what should be written to SCICR1 to match the discussion above.

The final configuration register to consider is the SCI Control Register 2. This register configures power state and interrupts for the module. We aren't interested (yet) in interrupts, but do want the transmitter and receiver turned on.

Learning Exercise 21. Determine what should be written to SCICR2 to match the discussion above.

As there is more than one SCI module available on the 9S12XDP512, there are a set of configuration registers for each. The registers use a number in their name to indicate the module they service. SCI0 is the module that is connected to the 9-pin RS-232 connector on your 9S12 board. SCI1 is connected to the infrared hardware on your 9S12 board. The register sets for each module have the port number in their names to differentiate the modules.

Learning Exercise 22. Put together the previous discussion to come up with an initialization subroutine for SCI0, setting it to 19 200 baud, 8N1, with no interrupts enabled, but with TX and RX enabled. This will be the first routine in a library called *SCI_Lib.inc*. Hints: You have three registers to write values to, one of which is 16-bit. For SCI0, their labels in the *mc9s12xdp512.inc* file are *SCI0BD*, *SCI0CR1*, and *SCI0CR2.* Call your initialization routine *SCI0_Init19_2*.

Learning Exercise 23. While you're at it, create another initialization routine to setup 9600 baud 8N1 communication. This is a very common communication rate for RS-232C. Call this one *SCI0_Init9600*.

Communicating through the Serial Communications Interface

Characters are transmitted when written to a register called SCIDRL (SCI Data Register, Low byte). There are low and high SCIDR registers, but the high register is only used for 9-bit data formats. Since you will only be using 8-bit data transfers in this course, you will only need to write to the low data register.

Care must be taken to write data to the SCIDRL only when the module is ready. The SCI Status Register 1 (SCISR1) indicates readiness using the Transmit Data Register Empty Flag (TDRE) in bit 7. The TDRE flag should be checked before a write. If TDRE is a '1', then it is OK to write a new byte to the SCIDRL. NOTE: TDRE does not indicate transmission complete – it only indicates that the transmit data register is empty.

Learning Exercise 24. Add the following routine to your SCI library. Feel free to include more information in the header, including notes about registers modified, and add comments.

```
;*******************************************************************
;*    SCI0_TxByte                                                  *
;*    Sends a byte brought in accumulator A to SCI0, and waits until    *
;*    the transmit buffer is empty before returning (blocking routine)  *
;*    Registers affected:  none                                    *
;*******************************************************************
SCI0_TxByte:      TST   SCI0SR1
                  BPL   SCI0_TxByte
                  STAA  SCI0DRL
                  RTS
```

1. What does "TST" do?

2. Why has the branch command "BPL" been used in the above code?

To read data from the SCI module you need to read the SCIDRL register. This is a bidirectional register, as writing to it transmits data and reading from it fetches received data.

You can't read a valid received byte until one has actually been received. You can check to see if a byte has been received by looking at the Receive Data Register Full flag (RDRF), which is bit 5 in the SCISR1 register. This flag will be set when a byte of data has been received by the module. If this flag is a 1, you should read from SCIDRL to extract the received byte.

NOTE: The SCI module is only able to buffer one received byte. If you fail to extract a received byte before another is received, a buffer overrun condition occurs and the previous data character will be lost.

When you create a routine to receive a byte of data from the SCI, you don't necessarily want the subroutine to block, waiting for a byte to be received. Because the byte may never be sent, the subroutine could block forever. As a general rule, you want to avoid creating routines that could block indefinitely.

A better approach would be to check to see if a byte has been received and is waiting to be read. If so, return it; if not return from the subroutine, but indicate that a byte was not available. You could use accumulator A for the data and the Carry flag to signal validity of the data:

Learning Exercise 25. Write the following subroutine into your SCI library. Again, feel free to make the header more informative, and add appropriate comments.

```
;*********************************************************************
;*    SCI0_RxByte                                                    *
;*    Receives a byte in accumulator A from SCI0 if a valid byte is  *
;*    present.  The presence or absence of a valid byte is indicated *
;*    using the Carry bit as a flag.                                 *
;*    Registers affected:  A                                         *
;*********************************************************************
SCI0_RxByte:      BRCLR SCI0SR1,%00100000,SCI0RxNot
                  LDAA  SCI0DRL
                  SEC
                  BRA   SCI0_RxEnd
SCI0RxNot:        CLC
SCI0_RxEnd:       RTS
```

Learning Exercise 26. Explain the “leapfrogging” done with the branches in the above routine.

It’s up to the calling code to look at the carry flag and take action. This routine is definitely preferable to a blocking version if your calling code has other stuff to do while it is waiting for characters to be received.

Learning Exercise 27. Write a small section of “main” code for a program that uses your library components to wait for a character to arrive, then echoes it back out to the sender. It then loops back to look for another character. With this, you will be able to “receive” characters from the keyboard of your terminal emulator (computer running emulation software, such as Tera Term) and see those characters appear in the terminal window. Your instructor will be happy to instruct you in the setup of the terminal emulator. You will also need to connect the DB-9 Comm port on your board to a Comm port on your computer.

The VT100/VT52 Terminal

The terminal program on the PC side of the connection will emulate a VT100 or VT52 terminal. (If you're interested, do a web search to see what these looked like.) These terminals were able to display received characters, manage a cursor, and send typed characters from a keyboard through an RS-232 connection.

Characters are limited to those found in the standard 128-character ASCII table:

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
(NUL)	0	0x00	(SP)	32	0x20	@	64	0x40	`	96	0x60
(SOH)	1	0x01	!	33	0x21	A	65	0x41	a	97	0x61
(STX)	2	0x02	"	34	0x22	B	66	0x42	b	98	0x62
(ETX)	3	0x03	#	35	0x23	C	67	0x43	c	99	0x63
(EOT)	4	0x04	$	36	0x24	D	68	0x44	d	100	0x64
(ENQ)	5	0x05	%	37	0x25	E	69	0x45	e	101	0x65
(ACK)	6	0x06	&	38	0x26	F	70	0x46	f	102	0x66
(BEL)	7	0x07	'	39	0x27	G	71	0x47	g	103	0x67
(BS)	8	0x08	(	40	0x28	H	72	0x48	h	104	0x68
(HT)	9	0x09	)	41	0x29	I	73	0x49	i	105	0x69
(NL)	10	0x0a	*	42	0x2a	J	74	0x4a	j	106	0x6a
(VT)	11	0x0b	+	43	0x2b	K	75	0x4b	k	107	0x6b
(NP)	12	0x0c	,	44	0x2c	L	76	0x4c	l	108	0x6c
(CR)	13	0x0d	-	45	0x2d	M	77	0x4d	m	109	0x6d
(SO)	14	0x0e	.	46	0x2e	N	78	0x4e	n	110	0x6e
(SI)	15	0x0f	/	47	0x2f	O	79	0x4f	o	111	0x6f
(DLE)	16	0x10	0	48	0x30	P	80	0x50	p	112	0x70
(DC1)	17	0x11	1	49	0x31	Q	81	0x51	q	113	0x71
(DC2)	18	0x12	2	50	0x32	R	82	0x52	r	114	0x72
(DC3)	19	0x13	3	51	0x33	S	83	0x53	s	115	0x73
(DC4)	20	0x14	4	52	0x34	T	84	0x54	t	116	0x74
(NAK)	21	0x15	5	53	0x35	U	85	0x55	u	117	0x75
(SYN)	22	0x16	6	54	0x36	V	86	0x56	v	118	0x76
(ETB)	23	0x17	7	55	0x37	W	87	0x57	w	119	0x77
(CAN)	24	0x18	8	56	0x38	X	88	0x58	x	120	0x78
(EM)	25	0x19	9	57	0x39	Y	89	0x59	y	121	0x79
(SUB)	26	0x1a	:	58	0x3a	Z	90	0x5a	z	122	0x7a
(ESC)	27	0x1b	;	59	0x3b	[	91	0x5b	{	123	0x7b
(FS)	28	0x1c	<	60	0x3c	\	92	0x5c	|	124	0x7c
(GS)	29	0x1d	=	61	0x3d	]	93	0x5d	}	125	0x7d
(RS)	30	0x1e	>	62	0x3e	^	94	0x5e	~	126	0x7e
(US)	31	0x1f	?	63	0x3f	_	95	0x5f	(DEL)	127	0x7f

Some of these characters (everything less than $20) have special meaning – BD, LF, FF, CR, BEL, etc. The characters in this table are the standard 7-bit ASCII characters. The other 128 available characters ($80 to $FF) are called "extended ASCII" and are not standardized. Trying them out will produce different results on different terminals, so you're welcome to play around with them if you have a fairly high tolerance for frustration.

Upper and Lower Case ASCII Codes

Look at the table of ASCII characters to see what's different between uppercase and lowercase letters. (Hint: write them out as binary representations.)

Learning Exercise 28. Add two more routines to *Misc_Lib.inc* called *ToUpper* and *ToLower*. The names should be self-explanatory. Use the A accumulator to transfer characters back and forth. Make sure your routines only affect letters of the alphabet!

Escape Sequences

Some sequences of multiple characters are interpreted in a special way by the terminal, and are not displayed. Instead, these sequences trigger a change in the terminal. These sequences may alter the cursor position, character or background color, and other terminal settings.

The character sequences the VT100 terminal recognizes are typically 'escape sequences'. The name comes from the fact that these character sequences begin with an escape character. You will use several different escape sequences to make the terminal do your bidding.

The following small table shows a few of the sequences of interest. A more complete set is available in Moodle.

A word of caution: Not all terminal emulators produce the same results from the escape sequences, even if they claim to emulate the same terminals (e.g. VT100). You will probably soon find that a number of the sequences you try within Tera Term don't do what you want them to do. HyperTerminal will produce yet other results. Trial and error will, hopefully, bring you satisfactory performance.

Escape Sequence (<esc> means escape character)	**Function**
<esc>[2K	Erase Line
<esc>[y;xH or <esc>[y;xf	Set Cursor Position (y = row, x = column)
<esc>[31m	Set text red
<esc>[?25l (that's lowercase L for LOW)	Cursor off
<esc>[?25h (that's lowercase H for HIGH)	Cursor on

Learning Exercise 29. What hex byte do you need to use in place of <esc>?

Forming Constants

One of the things you will need to add to your programs is a collection of lookup tables and constant strings. As you should recall, constants are defined in the CodeWarrior assembler with the DC directive. Like the DS directive, DC supports a DC.B, DC.W, and DC.L form. Constants are defined in ROM and are inherently set to the values you specify. The syntax is the directive followed by just about anything the assembler can interpret as data. You will usually be using the DC.B form to create constant strings and blocks of values.

The constants block must be in ROM, and may be positioned anywhere after your code and before the vectors. The following screen capture shows some constants defined in ROM, and what actually appears in memory as a result of these definitions.

One of the main reasons for defining such constants is to include the "escape sequences" that format the display in the way you want it displayed. These sequences are somewhat awkward to set up, and you will use a number of them repeatedly. Pointing to them in a look-up table saves you entering the sequence multiple times.

```
;*****************************************************************
;*                    Constants                                  *
;*****************************************************************
Name:       dc.b 'Lance Fundbonker'
Data:       dc.b $5,2,@4,%101,'G'
RedName:    dc.b $1B, "[31mBootney Farnsworth", 0

;*****************************************************************
;*                    Includes                                   *
;*****************************************************************
```

```
Name:       dc.b 'Lance Fundbonker'
Data:       dc.b $5,2,@4,%101,'G'
RedName:    dc.b $1B, "[31mBootney Farnsworth", 0
```

Memory

Name 4015 - 4024

```
004010 04 31 FA 20 F3 4C 61 6E  .1. .Lan
004018 63 65 20 46 75 6E 64 62  ce Fundb
004020 6F 6E 6B 65 72 05 02 04  onker...
004028 05 47 1B 5B 33 31 6D 42  .G.[31mB
004030 6F 6F 74 6E 65 79 20 46  ootney F
004038 61 72 6E 73 77 6F 72 74  arnswort
004040 68 00 36 18 0B 80 00 E8  h.6.....
```

In order to use these sequences, you will want another routine in your SCI library – a routine that can print each character sequentially in a string, detecting the end of the string appropriately so that only the desired characters are transmitted. The most common way to do this is with what we call a "Null Terminated String" format. Since the $00 character is non-printable (NULL), we can use it as a marker for the end of the string we've created. The string labeled "RedName" in the figure above is a null terminated string.

Our string-transmitting program will walk through the string, and when it encounters $00, it will exit and return to the calling program.

Learning Exercise 30. Given what you now know, you are ready to create SCI0_TxStr – A routine to transmit a string. Point X at the start of the string and transmit characters until you encounter a NULL.

Hexadecimal to ASCII conversion

Look back at the table of ASCII characters. Notice that the ASCII character codes for numeric digits (0123456789) and the hexadecimal extensions (ABCDEF) do not match their actual value. In other words, if you want to display "7" on your terminal emulator, sending the ASCII code "7" will make the terminal beep instead. What you need to do is send "$37" in order to display "7" on the screen.

Converting regular digits (0123456789) to ASCII is easy – just add $30 to the digit or OR the digit with %00110000.

Converting the hexadecimal values ABCDEF to ASCII is similar, but with a different offset.

Learning Exercise 31. What would you need to add to the numerical value of "$B" to display it as the corresponding ASCII character?

Now, you've got two different things you need to do to hex digits in order to convert them to ASCII. This means that you will need to determine first which range the digit falls into, then massage it appropriately.

Learning Exercise 32. Create a new routine for your *Misc_Lib.inc* that will accept a single hexadecimal digit (i.e. nibble) in the lower half of accumulator A and convert it to the corresponding ASCII character, returned in accumulator A. Call your routine *Hex2ASC1.* Create a proper header for this routine, and indicate its presence in the library by modifying the header for the library.

Learning Exercise 33. Create another routine for your *Misc_Lib.inc* called *Hex2ASC* that uses *Hex2ASC1* to convert a two-nibble hexadecimal number into two ASCII bytes, the most significant of which is returned in accumulator A and the other in B.

Learning Exercise 34. Write simple programs to check all of your SCI0_Lib and Misc_Lib routines. Your instructor will want to see that everything is working.

Parallel Interfaces: Get On The Bus

For high-speed communication over short distances, designers prefer to use parallel interfaces. These interfaces provide one conductor per bit, and deliver all bits in a particular piece of information simultaneously.

Early microprocessors had 4-bit busses, and could communicate the bits in a nibble simultaneously. Later, 8-bit busses were introduced, and whole bytes could be transferred at once. Since then, microprocessors have graduated to 16-bit busses, then 32-bit busses, and now to 64-bit busses in an attempt to keep up with the growing speed requirements in the computer market.

The microprocessor at the heart of the 9S12 microcontroller uses a 16-bit bus. In fact, it uses two 16-bit busses: one for data, and one for addresses.

Data Bus

The data bus carries information between two devices, and is typically bidirectional. In other words, data can be sent to the device and data can be received from the device. Some specialized devices require only one of these directions. All bussed devices in a piece of equipment will share the same bus, but only one device can talk at a time. If more than one device tries to talk, the results will be, at best, totally unintelligible, and at worst, damaging to one or more of the devices on the bus. To prevent this, bus interfaces on idle devices are put into a state called “High-Z”, or high impedance, effectively disconnecting them from the bus so they won’t interfere with other devices.

Address Bus

In order for the microprocessor at the heart of a bussed communication system to talk to the right devices at the right time, each device (and, almost always, each memory location within a device) will be given a unique address. So, for example, when you want to see if the switches on your board are pressed, you need to look at address $0270 in the memory space of the 9S12 micro – the address assigned to PTADHi. As you run your code, the Program Counter steps its way through the addresses in ROM where the bytes that make up the opcodes and operands in your assembled machine code reside.

The number of unique addresses depends on the number of address lines in the address bus. If the address bus is 16 bits wide, as in the 9S12, we can access 2^{16} unique addresses, or 65,536. Obviously, your home computer’s address bus is a lot bigger than 16 bits in order to access all the RAM and all the peripherals it’s got.

Control Lines

So, in order to talk to a device at a particular address, we must put the correct address on the address bus. But there’s more: the device needs to be activated (placed on the data bus), and it needs to know if data is coming or is required from it. Some devices need to notify the micro that they need to be serviced, and initiate an Interrupt Request (IRQ). Sometimes, a device also needs something to synchronize its internal activities with the microprocessor’s bus clock. All these activities are managed by a separate set of control lines. The following are typical for Motorola-based microprocessor bus devices:

/EN – when LOW, this line takes the device out of High-Z mode and “places it on the bus”.

R/W – when HIGH, the microprocessor READS from the device; when LOW, the microprocessor WRITES to the device. (Some non-Motorola-based devices require separate /READ and /WRITE lines – watch out for these if you end up doing design work!)

PH2 or ECLK – this is a clock line that lags the bus clock by 90º. It is used by devices that require a bit of time to respond or that need to know when data on the bus is truly valid.

LCD Displays Using The Hitachi HD44780U Controller

A particular LCD controller IC is almost ubiquitous: almost any small character array LCD will have one of the variants of the Hitachi HD44780 as its brains. In fact, Wikipedia says "An **HD44780 Character LCD** is a de facto industry standard liquid crystal display (LCD) display device designed for interfacing with embedded systems." – lousy English, but true. The 4 row by 20 character LCD display on your development kit is driven by one of these.

The HD44780 is, itself, an embedded microcontroller. So, in effect, your development board an example of parallel processing – two microcontrollers running separate processes, but communicating with each other to produce coordinated results.

The HD44780 is designed to operate within a bussed, or parallel, interconnect system. It has eight data lines, requires a single address line to select between two internal registers, has an active HIGH enable line (that's unusual), and a R/W line.

This controller is quite flexible. The full details of its capabilities are listed in the data sheet, available in the Appendix. Here are some of its capabilities:

- Can be used with a variety of LCD displays, ranging from 1 line x 8 characters to 4 lines x 40 characters.
- Can be used on an eight-bit bus or, by multiplexing data lines, on a four-bit bus.
- Can print stationary characters from left to right or right to left, or can scroll characters to the left or right.
- Can produce characters in a 5 x 8 dot matrix or in a 5 x 10 dot matrix.
- Can display standard ASCII characters or use extended character sets of symbols from different languages.
- Can be used to display up to 8 user-defined special characters.
- Can control the cursor in a variety of ways.

Upon start-up, the HD44780 has no idea what it's connected to, on either side: It doesn't know whether it's on a 4-bit or 8-bit bus, and it doesn't know what LCD it's connected to, so it doesn't know whether to produce 5 x 8 or 5 x 10 characters, or how many rows and characters per row it should be producing. You are responsible to tell it everything it needs to know, and you can only do that by communicating with it through the 9S12.

The HD44780-controlled LCD on the 9S12 Development Kit

If you check out the schematic for your development kit, you'll discover the following set of interconnections between the 9S12 and the HD44780.

Port H is used to create an eight-bit data bus, using PH0 through PH7 to map to b0 through b7, respectively. Port K is used for the address line and the two control lines:

PK0 => Enable (active HIGH)
PK1 => R/W
PK2 => RS (internal address select)

Operation:

The LCD controller is able to read *instructions* and *data*. The device uses a separate address line (RS for Register Select) to differentiate between the two. Address 0 accesses the Instruction Register (IR) and provides control of the device. Address 1 accesses the Data Register (DR) and provides information to and from the device.

As previously mentioned, the HC44780 LCD Controller is a microcontroller designed to drive a number of different LCD displays. In our application, it needs to drive a 4-line x 20-character display, with characters built using a 5 x 8 matrix. The 9S12 Development Kit is also designed for operation using the 8-bit interface described above.

Inside the controller, the display is actually two lines of 40 characters per line, each in a unique memory location. On our 4-line display, the first "line" of 40 characters actually appears on lines 1 and 3, and the second "line" appears on lines 2 and 4.

The addresses for the various character locations are as follows:

Line on screen	Address (decimal)	Address (hexadecimal)
First	0 to 19	$00 to $13
Second	64 to 83	$40 to $53
Third	20 to 39	$14 to $27
Fourth	84 to 103	$54 to $67

Note that the display memory addresses for the lower "line" have bit 6 turned on, which may be useful if you want to switch between lines. The display memory addresses $28-$3F (40 to 63) are not to be used, and may be mirrors of other display address locations, resulting in unpredictable behaviour.

You may write instructions to the LCD to shift the display position. This means something different for different displays – on a two-line display, you can bring "hidden" characters in from the part of the internal line that are outside of the window. On our four-line display, the characters roll between lines 1 and 3, and between lines 2 and 4, which isn't usually desirable.

The LCD features a cursor. The cursor is configurable for appearance and behavior. The cursor is usually set to automatically advance to the next location after a display write (i.e. to the right of the previous character), but you may change this.

The LCD internally keeps track of the display data address (i.e. the character location in the display, also known as DDRAM). When you write display data to the device, it goes into the memory location specified by the current display data address. The controller may be configured to increase or decrease the display data address after a write (i.e. move right or move left). The display may also be set to shift after a write, providing a scrolling effect – again, either to the right or to the left. With the four-line display, this means switching to the alternate line when the DDRAM address gets to the end (or beginning) of the addresses for the current visible line – again, probably not what you were hoping for.

Instructions:

When writing instructions to the LCD, the first set bit (HIGH) in the instruction byte determines the instruction. These instructions are found in the data sheet for the HD44780, for which a link has been provided in Moodle, and here as well:

http://pdf1.alldatasheet.com/datasheet-pdf/view/63673/HITACHI/HD44780.html

Please access that for the following discussion.

The default condition of the controller is as follows: 8-bit mode, 1-line display, 5 x 8 matrix, display off, cursor off, blink mode off, increment cursor position (move to the right), with shifting turned off (display doesn't scroll). We need to initialize the controller to make it match our hardware.

Timing is critical in all communications with this controller. To begin with, the Busy flag is not active until a particular sequence of commands has been executed. In addition, data needs to be present 60 ns prior to an Enable pulse, and the Enable pulse must be HIGH for at least 500 ns followed by at least 500 ns LOW. Due to internal activity in the HC44780U, at least 40 ms must be allowed following power-up. After the first command is sent to the controller, at least 4.1 ms must be allowed before the second command is sent, then 100 μs must be allowed before the third command is sent. After the third command, the Busy flag becomes available, and can thereafter be used to monitor the controller's activity.

The bus clock frequency on your board is 8 MHz, so a cycle is 125 ns.

LCD Controller Initialization:

The 44780 LCD controller must be initialized carefully, as seen the the datasheet. Locate the table that shows how to initialize the device for an 8-bit interface.

The following code sections show how both the timing considerations outlined above and the requirements for initialization can be achieved. Note: The initialization routine continues for the next few pages, with explanatory comments in between section of code.

Learning Exercise 35. Begin by starting a new library called "LCD_Lib.inc". The following header tells you what we're going to put in this library:

```
;*****************************************************************
;* HC12 Library:  LCD_Lib.inc                                    *
;* Processor:     MC9S12XDP512                                   *
;* Xtal Speed:    16 MHz                                         *
;* Author:  You!                                                 *
;* Date:    Now!                                                 *
;*                                                               *
;* Contains:            LCD_Init                                 *
;*                      LCD_Ctrl                                 *
;*                      LCD_Busy                                 *
;*                      LCD_Char                                 *
;*                      LCD_Addr                                 *
;*                      LCD_Blink                                *
;*                      LCD_NoBlink                              *
;*                      LCD_CursOff                              *
;*                      LCD_Clear                                *
;*                      LCD_Home                                 *
;*                      LCD_String                               *
;*                      LCD_CharGen                              *
;*                      LCD_CharGen8                             *
;*                                                               *
;*****************************************************************
```

Learning Exercise 36. First on the list is the initialization routine "LCD_Init", but to make it work properly, we need to have "LCD_Ctrl" and "LCD_Busy" working properly. So, we'll have to build these three components before we can do any testing. First, we need to set up Port PTH as our interface. This involves controlling the Direction Register, DDRH. We'll be changing DDRH on the fly, as we sometimes want to send data to the LCD controller, and we sometimes want to receive data from the controller. We'll begin by setting it to outputs so we can send an instruction to the LCD controller.

```
LCD_Init:
            PSHD                    ;save accumulator

            CLR   PTH               ;ready to initialize
            MOVB  #%11111111,DDRH   ;setup Port H to write to LCD
```

We also need to set up Port K so that we can use it for our control lines:

```
            CLR   PORTK             ;/WRITE low, enable low, RS low
            MOVB  #%00000111,DDRK   ;now make them outputs
```

Timing is critical, so the next item needed is a long delay:

```
            LDD   #$FFFF            ;setup a 49.15 ms countdown
NotYet1:    LBRN  *                 ;[3]
            DBNE  D,NotYet1         ;[3] decrement, loop
```

Learning Exercise 37. Each loop of "NotYet1" is 6 cycles long, ([3] + [3]). Verify that this timer loop is long enough to meet the requirements on a previous clip from the 44780 data sheet.

The first instruction that must be sent to the LCD during initialization is preliminary "Function Set" to tell the LCD controller how many bits will be used on the data bus to communicate with the device (4 or 8).

Locate the line in the Command Summary that explains this. The following are instructive:

DL == 1 (8 bit interface)
DL == 0 (4 bit interface)

N == 1 (2 Line Display)
N == 0 (1 Line Display)

F == 1 (5 * 10 dot characters)
F == 0 (5 * 8 dot characters)

Learning Exercise 38. From the above, determine why we will start by sending a value of %00111000 for the Function Set command, given the way the board is wired.

To send the controller a command, we need to place this command on the 8-bit bus, make the Register Select LOW, make the R/W line LOW, and then "strobe" the EN line to activate the LCD Controller by driving it HIGH then LOW

Learning Exercise 39. Interpret the above discussion in terms of what needs to be done to write the command %00111000 to the controller, and add this to your initialization routine. Use BSET and BCLR, as they are 4 cycles long or 500 ns each, thereby satisfying the requirements for a 500 ns strobe followed by at least 500 ns.

Learning Exercise 40. Now for another delay: We need at least 4 ms before we can try to communicate again. Given that "DBNE D,*" is 3 clock cycles long, determine how many times you would need to go through this loop to get 4.125 ms. Add the appropriate code to your initialization routine.

Learning Exercise 41. What's next, according to the previous flow chart? Add this to your initialization routine.

Learning Exercise 42. Since, according to the data sheet, the Busy flag isn't working yet, we need to allow the controller time to complete processing – at least 100 μs. Calculate the number of cycles needed, and add another delay to your routine.

Learning Exercise 43. Close examination of the flowchart for initialization indicates that we need to send out our control byte twice more, with 100 μs between. Do so.

At this point, the controller has been crudely initialized and the Busy flag is now working. We now want to send it some finer controls to get it to match our hardware. Before we do, though, we want to set up a library routine that sends control signals properly, which means checking for the "Busy" flag from the controller rather than just waiting 100 μs between commands.

We'll create a subroutine called LCD_Ctrl, but because the "wait for busy" routine it needs is something we can use elsewhere, we'll create it as a sub-subroutine (or is that a sub-sub-subroutine?) LCD_Busy watches the Busy line to see when the HD44780 controller is available for more commands.

LCD Control:

The LCD_Ctrl routine expects an 8-bit instruction to arrive in the A accumulator, which it sends out as a command (RS HIGH, R/W LOW, Enable strobed) as soon as it is sure that the "Busy" flag is cleared.

Learning Exercise 44. The "LCD_Busy" routine is going to use the Carry bit of the CCR to tell us if the 44780 LCD controller is busy (set) or available (clear). Here's a start for the LCD_Ctrl routine. Finish it off from what you know at this point.

```
;*****************************************************************
;*LCD_Ctrl                                                       *
;*                                                               *
;*Regs affected:  None                                           *
;*                                                               *
;*sends command received in Accumulator A to LCD                *
;*                                                               *
;*This routine relies on LCD_Busy, which returns a Carry         *
;*                                                               *
;*****************************************************************

LCD_Ctrl:
            JSR   LCD_Busy          ;wait until the LCD controller is free
            BCS   LCD_Ctrl          ;-- as reported in Carry


      (finish this routine)
```

You won't be able to test this routine yet, because it relies on *LCD_Busy*, which checks to see if the controller is available for communication.

LCD Busy Check:

The LCD_Busy routine must query the LCD controller for the info in its status register which contains the Busy flag and seven bits representing the cursor's current location. (We don't care about the info about the cursor's location, so we'll ignore it and just look at the flag). Getting this information from the LCD controller can be done by executing a READ of the RS register.

This involves switching the direction of the data bits to inputs. The routine we're going to write is a non-blocking routine, allowing the user to write programs that continue on until the LCD controller is free. In the LCD_Ctrl routine above, we block until the controller is free anyway, as indicated by a cleared Carry bit. However, the LCD_Busy routine as written allows us the option of moving on if we want to.

Once we've read the status register, we need to switch the port back to outputs.

Learning Exercise 45. Here's a start for the LCD_Busy routine. Finish it off from what you know at this point.

```
;*****************************************************************
;* LCD_Busy                                                      *
;*                                                               *
;*Regs affected:  CCR   (Carry)                                  *
;*                                                               *
;*Reads busy flag at b7, returns Carry to reflect Busy           *
;*                                                               *
;*****************************************************************

LCD_Busy:
            CLR   DDRH                    ;make inputs to read from LCD

            BSET  PORTK,%00000011         ;Read high, enable high
            BCLR  PORTK,%00000111         ;Read low, enable low (and RS, too)
            BRCLR PTH,%100000000,NotBusy  ;check busy flag, found in b7
```

Learning Exercise 46. Write code that sets the Carry bit if the controller is busy or clears the Carry bit if the controller is available. Both cases need to take you to a final cleanup that returns Port H to outputs before you leave this routine.

Now we're finally ready to finish off our LCD_Init routine with commands that will put the LCD controller into the modes we're looking for, with the cursor ready to display data in the top right corner (Home).

Learning Exercise 47. In LCD_Init, add a command that will turn the display ON, turn the cursor ON, and turn blinking OFF. (Go back to the instruction table, and look for "Display on/off control" to figure out what you need to send.)

Learning Exercise 48. You can make a simple Main.asm program that checks your work to this point, which should give you an Underline cursor in the top left corner if all goes well.

Learning Exercise 49. In LCD_Init, add a command that will put the controller into "increment cursor mode" with "no shift". (This is the "Entry mode set" command.)

Learning Exercise 50. Add one last command to your initialization routine – move the cursor to HOME position (Figure this one out yourself.)

Now that we have a proper initialization routine and the ability to send commands to the LCD controller, we're almost ready to start using the LCD display! The LCD display expects us to send, as data, ASCII codes.

Learning Exercise 51. Add another routine, LCD_Char, to your library. This routine is similar to LCD_Ctrl, except that the byte sent using Accumulator A is supposed to be a valid ASCII code, and must go to the data register instead of to the instruction register. This involves setting RS HIGH.

Learning Exercise 52. Test your LCD_Char routine by calling it from a Main.asm.

Learning Exercise 53. Add the following routines to your library, matching the descriptions given. Most of these will be tiny routines, although LCD_String will be a bit more intense.

- `LCD_Addr – sets the DDRAM address to a value in A (this will allow you to place output characters where you want)`
- `LCD_Blink – sets the cursor to blink (and turns it on)`
- `LCD_NoBlink – turns cursor blinking off (and turns cursor on)`
- `LCD_CursOff – turns the cursor off (surprise)`
- `LCD_Clear – clears the display and returns home`
- `LCD_Home – moves the cursor home without clearing display`
- `LCD_String – uses LCD_Char to send each of the characters in a null-terminated string pointed to by X to the LCD.`

NOTE: The first six of these subroutines are almost identical, except for labelling and the actual instruction byte sent to the LCD controller. They all rely on LCD_Ctrl.

- Test your routines by writing code segments for each. For display purposes, you could write a small chunk of code with all of the subroutines from your library represented, and simply comment out different commands to see how the others work, or step through the code to see what each one does.

Your instructor will want to look at your library. Headers and comments are to be expected at this point.

Character Generation

Now that you know how to initialize your LCD, write to it, and control the appearance of display, it will be useful to know how to make the LCD produce characters you design instead of using the built-in tables of characters.

Your LCD is able to display 8 user-defined characters, which you design pixel by pixel within the 5 x 8 pixel matrix. The memory in the device that holds the pixel pattern is known as character generator RAM (CGRAM).

To see the user-defined characters, reference ASCII characters $00 to $07. By default, these characters will likely show junk until you define them. In the ASCII table, these are non-printable characters, so Hitachi engineers decided to re-define them as the spaces available for your custom characters.

To get your user defined character patterns into the device, you must program them, row by row, for each character. The device must also be instructed to accept CG data. The "Set CGRAM Address" instruction does this. Look it up in the data sheet's table of commands.

This instruction is written to the LCD Instruction Register and tells the LCD that all subsequent data written to the Data Register will be CG (Character Generation) data. The instruction includes the address of the CGRAM to start at. Only 6 bits are required, because only 64 bytes are required to represent the eight 5 x 8 characters – each row requires one byte, so each of the eight characters requires eight bytes.

The top of ASCII character $00 is at CG address $00, and extends to address $07. Note that the first 3 bits (most significant) are ignored, as the characters are only 5 pixels wide.

In the current version of the data sheet, Table 5 shows you how to build the bitmap for special characters in CGRAM.

CG data may be programmed at any time, including when characters for that type are currently being displayed. Some unusual animation effects may be generated by programming the characters that are currently being displayed.

Once the programming of CG data is complete, the device should be set back to display data (DDRAM). The "Set DDRAM Address" instruction does this. Go to DDRAM address 0, the home position on the display as a good place to start.

This instruction sets the LCD to accept DD information from the DR for all subsequent writes.

Remember that you can create up to eight custom characters.

Learning Exercise 54. Create the following library components:

- LCD_CharGen – a routine that builds a single custom character for the ASCII code indicated by accumulator A ($00 to $07), with the character definition pattern beginning at a location pointed to by the X register. Incidentally, since the eight bytes representing the eight lines in a custom character are contiguous, you don't need to point to each of them: just write them out one after the other.
- LCD_CharGen8 – a routine that builds eight custom characters, with their definition patterns beginning at a location pointed to by the X register. (If you don't need all eight characters, just fill the unused row bytes with nulls to produce blank characters.) This should look a lot like "loadCGData" in the code found in the Appendix, but for eight characters instead of five.

Accurate Timing

Up to this point, you've created simple but not necessarily very accurate timing delays using counters and loops. There is a better way!

The 9S12XDP512 chip contains a built-in timer module, with a great deal of functionality and flexibility. Locate the block diagram for the timer modules in the 9S12 data sheet. In the current version, this is on page 310. Here's the link to the datasheet again:

http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf

At the core of the timer module exists a 16-bit counter, the current count value being available at the 16-bit location TCNT (addresses $0044 and $0045). This counter simply counts up as long as the timer module is enabled. The points of interest here are that this is a 16-bit register (occupies two bytes), and a 16-bit read takes an instantaneous snapshot of the register's contents, while the counter itself runs on.

To enable Timer 0, you must set the TEN bit in the TSCR1 register. Looking at the datasheet for the timer module, we find:

Learning Exercise 55. Look at what the datasheet says about TSCR1. We want to enable the timer, but write 0s to all the other bits in this register. In your "Misc_Lib.inc", begin a timer initialization routine called "TimInit8us". This name will make more sense soon. Determine what should go into TSCR1, and make it so.

The rate that the timer counts at is determined by the bus clock and a prescaler. The bus frequency on your boards is 8 MHz (1/2 the crystal frequency). If the prescale value was left unmodified the counter would count at the bus frequency, or 1 count every 125 ns. In terms of generating useful delays, this is a VERY small amount of time. The prescale value may be 1, 2, 4, 8, 16, 32, 64, or 128. The bus frequency is divided by the prescale value to determine the counter frequency. The prescale value is set in the TSCR2 register. Look this one up in the datasheet. It also contains a couple of enables – one for a Timer Overflow interrupt, which we won't enable, and one to allow for resetting the timer to zero in a special operational mode of the timer, which we also don't want to do.

Learning Exercise 56. For the purposes of generating a reasonable count time, we will set the timer to 8 µs ticks (hence the name of the routine you're writing). Determine what should be written into the prescaler portion of TSCR2, and make it so.

Once the timer is enabled and the prescale is set, you are free to perform "output compare" functions. An output compare is done by selecting a 16-bit value to compare to the free-running counter. When the values are equal, the output comparison event occurs, and is indicated in a flag register.

There are eight different output compares that can be used simultaneously – OC0 through OC7, although OC7 has special features that allow it to control the timer in special ways. Pretend you never heard that, and back away from OC7 … (that is, unless you need its special features for your capstone project). We will limit our access to just one – OC0. Each channel has its own output comparison register, and in the case of OC0, the register is TC0. We need to activate OC0 using the TIM0_TIOS register. This register is used to determine if we want to do a comparison to the internal timer (output compare) or and input capture (grab the value in the timer when an external event occurs). We want an output compare.

Learning Exercise 57. Determine what is needed in TIM0_TIOS (found in the discussion of TIOS), and add that to your routine.

The condition of OC0 is controlled by a 16-bit register called TC0. TC0 is compared to TCNT each tick to determine if an output compare event has occurred on this channel. Since the TC0 register will be some unspecified value upon startup (most probably $0000), an output compare event will occur naturally each time TCNT wraps around to that value. This happens every 65536 ticks at 8 µs sec/tick, or every 0.524288 s. We will typically manipulate the amount of time between output compare events by changing the contents of TC0 for every desired time period.

Observing the Action of the Timer

There are two ways to observe the action of the timer. One is programmatic, the other is electrical. Programmatically, every time TCNT matches TC0, a flag is set in a register called TFLG1. If you want to know when an output compare event has occurred, poll for the event flag in the TFLG1 register. This register will have a 1 in the corresponding channel bit after the event has occurred. In the datasheet, locate TFLG1 for the following discussion.

The simplest way to handle an output compare is to repeatedly check for the flag of the corresponding channel in your code (a blocking delay). Once the event is detected, you must write a 1 to the corresponding bit to acknowledge it and reset the flag.

For more useful systems, you should check the flag once each time through a loop that allows you to carry out other functions, such as checking switches or controlling LEDs.

Learning Exercise 58. In your initialization routine, clear the flag for TC0 by setting it.

Electrically, we can connect the timer output compare event to an associated pin on the device. When the timer compare event occurs you may set the pin to toggle logic levels. (You can also make the timer force this pin into SET or CLEAR logic levels, or you can disconnect the pin from the timer). The timer can be connected to the pins of Port T (PT0 through PT7). Each of these pins can respond to the corresponding output compare event so that you can observe what's going on with an oscilloscope or drive some piece of external circuitry using the clock.

The action of these pins is established using the TCTL registers. Look these up in the data sheet for the following discussion.

To see some nice action on the scope, the pin at Port T0 should be set to toggle on the OC0 output compare event. This means that every time the free-running counter and the output compare value match, the pin will flip. You will need a pin soldered to your board at Pin 9 of the microcontroller.

Learning Exercise 59. In your initialization routine, set up Port T0 to toggle on output compares of OC0.

Learning Exercise 60. Run your initialization routine in a test program. Use your oscilloscope to monitor Pin 9 – it should be toggling about twice a second.

Learning Exercise 61. In "Misc_Lib.inc", add an initialization routine called "TimInit1us". This should be very similar to the one you've just finished. Make appropriate changes to match the label.

Learning Exercise 62. Add another routine called "TimInit125ns".

Delays *vs.* Intervals

We can use the timer in two slightly different ways: individual delays or regular intervals.

Here's a real-life analogy. You may want to sleep an extra ten minutes before getting up in the morning. To do this, you check the time, add ten minutes to it, and set your alarm for the new time. When the alarm goes off, you've experienced a ten-minute delay. After yawning and stretching, you may feel like another ten-minute delay, so you check the clock again, add ten minutes, and set the alarm to the new alarm time. As a result, your alarm will go off a bit more than twenty minutes after the original wake-up time. If, however, you've got kids playing in the back yard, you may want to check on them every ten minutes. In this case, you check the initial time on the clock and add ten minutes to it to set the alarm. When the alarm goes, you set the alarm to ten minutes past the old alarm setting (even if you get distracted in between), and as a result you end up checking on the kids exactly six times for every hour, since the alarm goes off in ten-minute intervals regardless of how long it takes you to get around to resetting the alarm (assuming you get to it within ten minutes, that is). We'll get back to the difference between these two ways of handling a timer, but we need to know a couple more things first.

Delays

Specific timing delays can be generated by looking at the current TCNT value and adding an offset that matches the desired delay time. If you write this value into the TC0 register, the event will occur at precisely the desired delay time. The following code, for example, generates a 10ms blocking delay. If you're really picky, you might want to adjust the value added to the timer to account for the time spent managing the instructions, but I doubt that it's that significant.

```
; assume the Timer has been initialized for OC0 @ 8 us per tick

Delay10ms:  PSHD                          ;store for later

            LDD   TCNT                    ;get current clock setting
            ADDD  #1250                   ;10 ms
            STD   TC0

            BRCLR TFLG1,%00000001,* ;block until delay is over

            BSET  TFLG1,%00000001         ;clear flag

            PULD                          ;restore old value of D
            RTS
```

Learning Exercise 63. Answer the following questions about this code:

1. Explain the line "ADDD #1250".

2. Explain the line "BRCLR TFLG1,%00000001,*"

3. Explain the line "BSET TFLG1,%00000001"

Learning Exercise 64. Add the following three routines to your Misc_Lib.inc library. Make sure you add the descriptions to the header, and write a suitable header for each. Make sure your header specifies that these routines must be preceded by "TimInit8us". The code part of "Delay10ms" is on the previous page.

- `Delay10ms  (must follow TimInit8us)`
- `Delay1ms  (must follow TimInit8us)`
- `DelayXms  (X ms is from the X accumulator, relies on Delay1ms)`

1. Check your 10 ms delay by calling it repetitively, and observe with your oscilloscope. Remember that the output will toggle after each delay, so the period (two toggles) will be twice the expected delay, plus whatever setup time is required.
2. Check your 1 ms delay by calling it repetitively, and observe with your oscilloscope. Remember that the output will toggle after each delay, so the period (two toggles) will be twice the expected delay, plus whatever setup time is required.
3. Run your X ms delay routine set for 1000 ms. What do you see on PT0, and why?

4. To properly check this routine, you will need to have some other indicator to show the end of the interval. Toggle the Red LED, and observe the results on the oscilloscope using the port pin that's associated with it. Hints: Consult your schematic; Exclusive-OR toggles an output; Don't forget to initialize.
5. What is the longest delay you can create with DelayXms?

Learning Exercise 65. Create another routine called "DelayAms". This one sends the number of milliseconds in the A Accumulator, and uses "MUL" to help come up with the value placed in TC0. It does not use Delay1ms, and it toggles PT0 correctly. Hints: Check to see what is multiplied in "MUL" and where the answer ends up; You can add the contents of a memory location (like TCNT) directly to the D Accumulator.

What is the biggest delay you can create with DelayAms?

Learning Exercise 66. Just to prove a couple of points, run your "Delay1ms" routine after incorrectly initializing with "TimInit125ns".

1. What is the delay time now?

2. Why isn't it 1 ms?

3. Why isn't it 1 ms / 64, or 15.625 µs?

Intervals

To manage intervals, we get the initial value in the counter to set a new target. Then, each time we detect an output compare, we add the value to the previous *target event* value instead of to the current *timer* value. Since we add the count interval to the previous event value and not to the immediate clock value, our timing interval will be accurate even if we don't service it immediately.

With the timer, we can perform other tasks of varying length while we wait for the output compare event to occur. This becomes especially useful when use Interrupt Service Routines, something you will be introduced to as a tool available for your Capstone project.

To use the timer to set up exact intervals, you will need two routines: one to set up the first interval, and one to manage the remaining intervals thereafter. Here's some pseudo-code for these two routines, assuming that the time interval is provided using the X register:

IntXInitB – Subroutine for setting up an interval timer using a value in X, prescaler in B for more flexibility

Initialize Timer using the prescaler supplied in B – enable Timer, set up OC0 (and probably connect to PT0)
Add the increment from the X register to Timer's TCNT
Store this value into the timer compare register
Clear the C0F interrupt flag

IntX – Subroutine for managing an interval timer using a value in X

Wait for the interval to elapse (blocking delay)
Add the increment from the X register to the previous timer compare value
Store the new value back into the timer compare register
Clear the C0F interrupt flag

Learning Exercise 67. Add the following timer routines to Misc_Lib.inc.

- `IntXInitB (sets up an interval with a value in X, prescaler in B)`
- `IntX (waits for and resets the interval using a value in X)`

1. Check your interval timer, set up to produce a 1.0 s interval. You'll have to decide on suitable values for both B and X.

2. Set B to 0 and X to 125, values to produce a 15.625 us interval. Check the time interval. Why is this different from what you observed when we tried to produce a 15.625 us delay using a delay timer a few pages back?

3. How small an interval can you create before this timer fails?

4. Why does it fail at attempts to make even smaller intervals?

Input Capture and Pulse Accumulation

The timer pins can be used to provide timing information from outside events to the microcontroller. There are two ways we might want the microcontroller to respond to outside events: We might want to know how much time has elapsed since the last event (Input Capture) or we might want to know how many events have occurred over a set period of time (Pulse Accumulation).

Input Capture

Input Capture is a very simple procedure. Once a channel is configured as an Input Capture pin, each time an event occurs on that pin, the current value of the internal clock is stored in the 16-bit Timer Compare (TCx) register associated with that pin. The usual initialization steps are required – setting up the clock speed and enabling the clock. In addition, we need to set up the channel we're using for input capture. This involves TIOS, which we previously used when we set up our timer channel for Output Compare. Look it up in the datasheet, and determine what would be needed to set up an Input Compare instead.

Another parameter that should be controlled is the Input Capture Edge – sometimes you want to count when the signal goes from LOW to HIGH (rising edge) or when the signal goes from HIGH to LOW (falling edge). Sometimes, you might want to detect all changes, rising or falling (incidentally, this would double the frequency of a square wave). This involves Timer Control Registers 3 and 4 (TCTL3 and TCTL4). Locate these in the data sheet (in the current version, on page 325). There are two bits associated with each channel, since there are four possible options.

The Input Capture channel indicates that an event has occurred by setting the corresponding bit in the Flag register, TFLG1. As usual, you will need to clear this flag before you can wait for it to appear again.

Learning Exercise 68. Write a program and test it as follows:

1. Set up the timer using 1 µs ticks (TimInit1us).
2. Set up Timer Channel 7 (PT7) for Input Capture on rising edges only.
3. Read the current timer value, then wait for an input capture event to occur.
4. Subtract the old timer value from the new timer value, and display the results, in hexadecimal, on the top four digits of the seven segment display.
5. Display "Period =" on the top line, the difference from part #4 in BCD on the second line, and " us" on the third line of the LCD display.
6. Connect a signal from the TTL sync of your signal generator to PT7, and observe the results. Your oscilloscope can help you with determining the period of the signal.

Pulse Accumulation

Pulse accumulation means to count the number of incoming events that occur over a time interval. As you may have deduced from the previous exercise, Input Capture provides information that directly relates to a signal's period. Pulse accumulation is the inverse: it tells us information that directly relates to a signal's frequency.

To do a pulse accumulation, you will need two timer channels: one to set up the time period over which you wish to count events, and another to count the events that occur in that time period.

We'll just use routines we developed previously to set up the required time period, so that means Timer Channel 0 will be used for that.

For the Pulse Accumulator, there are a number of options. The MC9S12XDP512 has four 8-bit Pulse Accumulators connected to Timer Channels 3 through 0, or, in a different mode, it has two 16-bit Pulse Accumulators connected to Channels 7 and 0. The easiest one of these to work with, and the one that doesn't interfere with our time period counter, is Pulse Accumulator A, connected to PT7.

First, we need to enable Pulse Accumulator A, which automatically selects 16-bit mode on Channel 7. We also need to tell it that we're going to use it as an Event Counter. While we're at it, we also determine whether it will respond to rising edges or falling edges. All of this is done in PACTL, the Pulse Accumulator A Control. Look it up in the datasheet (in the current version, on page 322 and following).

Learning Exercise 69. Determine what should be written into PACTL to enable Pulse Accumulator A, set it up as a rising edge sensing event counter, using the prescaler clock. Since we haven't covered interrupts, turn off the interrupt features.

Once the Pulse Accumulator is set up, we need to set up our regular clock, clear the contents of the Pulse Accumulator register, and whenever the clock indicates that the time period is up, we read the Pulse Accumulator Count Register (16-bit) and reset it to zero for the next count. The Pulse Accumulator Count register is the 16-bit combination of PACN3 and PACN2. In the mc9s12xdp512.inc file, they provide us with the option of doing a 16-bit read of PACN3 or, with the same functionality, PACN32, an alternate name that probably helps you remember it's a 16-bit register.

Learning Exercise 70. Write a program and test it as follows:

1. Set up Pulse Accumulator A as determined above.
2. Using a 1 s timer interval, display the frequency, in hexadecimal, on the top four digits of the seven segment display. Check to see that the results are correct.

9S12 – PWM Module

When the first personal computers hit the market, the best audio they could manage was a collection of annoying squeaks and beeps. Creative individuals figured out how to make these squeaks and beeps sound like badly recorded voices and music by varying the frequencies and duty cycles of the waveforms in a technique called pulse-width modulation. With the advent of sound cards, those days are now in the past. However, it might surprise you to know that many of our low-power audio devices (and expensive high-powered audio amps, as well) use pulse-width modulation with slightly more sophisticated integration and filtering circuitry to produce high-fidelity sound. This is called Class-D power amplification.

Also, in the early days of remote-controlled toys, motors would either be turned fully on or off, resulting in jack-rabbit starts and stops, and crazy all-or-nothing turns. Pulse-width modulation now allows for much smoother motor control, not only for remote-controlled toys but for industrial processes, automotive devices, etc. Many microcontrollers have sophisticated pulse-width modulators to allow for programmable control of such devices. The 9S12 has a highly-configurable eight-channel PWM module. We will only begin to scratch the surface of the capabilities of this module.

The PWM module is used to create waveforms with programmable period and duty. There are a number of uses for programmable waveforms, most residing well outside the scope of this course.

For fun, we connect a speaker to one of the PWM channels of your board, channel 6, with a jumper to enable you to disable the speaker when you see an angry hulk approaching. We also have three channels of the PWM (channels 0, 1, and 4) wired to an RGB LED to allow you to control the resulting colour and brightness of this LED, and we have channel 3 wired to the backlight of the LCD display to allow you to control that, as well. The other three channels are available at the general breakout headers on the board.

PWM Channel	Function
0	RGB Blue
1	RGB Green
4	RGB Red
3	Backlight
6	Speaker

Using the PWM channel connected to the speaker, you can create waveforms of the correct period and duty to generate amusing sounds on your speaker. You can use these sounds to add useful enhancements to your code (key clicks, alarms, start-up sounds, etc.), create cheap ‘80s style music, or generally drive your lab mates crazy.

The PWM subsystem is fairly easy to get along with, and relies heavily on a series of clocks. As with most modules on the 9S12, the PWM subsystem is configured through a series of registers. We’ll learn about these registers as you need them.

There are eight fairly independent PWM channels in the MC9S12XDP512 – “fairly” independent, because you can control a lot of characteristics independently; however, the clocks, although highly configurable, are shared by four channels each, which can be challenging if you want to control devices requiring radically different timing characteristics.

An additional feature available in the MC9S12XDP512’s PWM module is the ability to combine (concatenate) pairs of eight-bit PWM channels into sixteen-bit channels. If you ever need to do this (likely to make extremely long signal periods), you will need to use the PWM Control Register (***PWMCTL***). Since this isn’t a common application, you’re left on your own to research this option if you need it.

There are four clocks that are available to drive the PWM rates, and all are based on the bus clock. The clocks are called A, SA (scaled A), B, and SB (scaled B). Either the A/SA or B/SB clock pairs are available for the PWM you're working with, defined in the hardware of the device. For each PWM channel, you must decide whether you want to use the basic clock that's available (A or B) or whether you want to use the scaled clock (SA or SB). This is done through the ***PWMCLK*** register. Look it up in the data sheet (in the current version, page 371).

The next register to configure is ***PWMPRCLK***. This register determines how clock A and/or clock B is divided down from the bus clock. Locate it in the data sheet. Notice that both prescalers are contained in the same register, but at different bit locations. Be careful with this!

If you have chosen to use the scaled version of the clock, SA or SB, it is necessary to use the ***PWMSCLA*** or ***PWMSCLB*** register, as well. This provides fine control over the settings chosen for the A or B clock previously. Look up these Scale Registers in the data sheet (in the current version, starting on page 375).

Be aware that the clock is divided by TWICE the value you choose for the scaling value, not the scaling value itself.

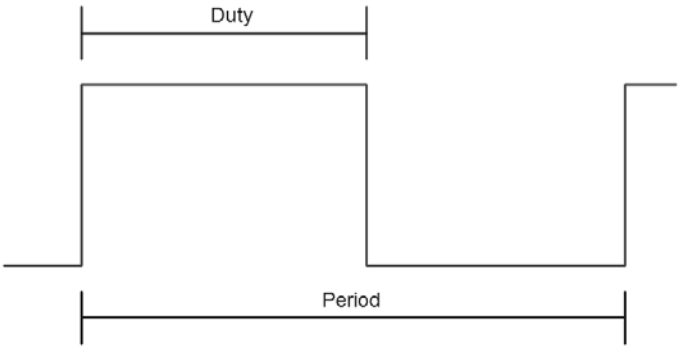
There's a "NOTE" on the data sheet indicating that 0x0000 means 256. This does not seem to be true. The biggest divisor available seems to be 2 x 255 = 510.

PWMSCLA divides down Clock A to generate Clock SA, and PWMSCLB divides down Clock B to generate Clock SB. In order to know what values to put into these registers, we need to know something about the desired clock rate.

Generating Waveforms

The waveforms you will be generating are managed by byte-sized period and duty values, controlled using two more registers: ***PWMPERx*** and ***PWMDTYx***, where "x" is the number of our selected channel. In addition, the ***PWMPOL*** register, which determines the polarity of the pulse produced, needs to be controlled.

The diagram below shows graphically the parts of a pulse, using standard terminology (positive polarity). For negative or inverted polarity, the "Duty" would be the time that the signal is LOW.



Locate ***PWMPOL*** in the data sheet (in the current version, page 370). Notice that the default condition of each PWM channel is negative or inverted polarity. If the duty cycle of your signal is something other than 50%, you will need to control PWMPOL.

The period and duty values are expressed as numbers of clock cycles. This means that the shortest period would be 2 cycles of the clock, (up for one cycle, down for one) and the longest would be 255 cycles of the clock. There are several strategies you may use to determine clock scaling values, the simplest being to pick a frequency and a fixed duty cycle, then work backwards to solve for the required clock pre-scalers, period value, and duty value. Enter the number of clock cycles for the period into PWMPERx, then enter the number of clock cycles (less than PWMPERx for obvious reasons) into PWMDTYx.

The period, and therefore the frequency, of the output signal are, therefore, controlled by two variables if the prescaled clocks are not used: The A or B clock pre-scaler (PWMPRCLK) and the number of PWM clock cycles per period in the PWMPERx register.

If the prescaled clocks are used, three variables control the resulting frequency: PWMPRCLK, the SA or SB clock prescaler (PWMSCLA or PWMSCLB), and the number of PWM clock cycles per period in the PWMPERx register.

Learning Exercise 71. From the information given, determine a formula for the period of a single cycle of PWM Channel 0 using the basic clock (A or B – indicate which one you will be using). You will need to include the original Bus Clock, PWMPRCLK, and PWMPERx (pick the right value for x!).

Learning Exercise 72. Determine the formula for the period of a single cycle of PWM Channel 6, this time using the additional prescaler (SA or SB – pick the right one!).

Learning Exercise 73. Use your previous answer to help you determine a formula for the frequency of Channel 6 using the parameters above.

For reasonably accurate frequencies, you should try to keep the value of PWMPERx large. For example, if you are off by a cycle, one cycle out of 255 is much less significant than one out of three!

Learning Exercise 74. Manipulate the formula you've arrived at above to produce suitable values for each register in order to form a 1000 Hz waveform. Keep in mind the ranges and possible values available for each.

The last thing to do is actually turn on the channel, which is done by setting the corresponding bit in the ***PWME*** register. Look up PWME in the datasheet (in the current version, on page 368 and following).

Manipulating this register allows you to turn your signals on and off under software control.

Learning Exercise 75. Use what you know to create a 1000 Hz signal with a duty cycle close to 50% on PWM channel 6 (the speaker channel). You've got three variables to work with, so there are lots of possible solutions to this. Observe the signal at TP1 using an oscilloscope, and, using a two-pin jumper cap on JP1, connect it to the speaker so you can hear it, too. The potentiometer (VR2) allows you to control the volume. Choose a volume level suitable to the prevention of physical violence in the lab.

Learning Exercise 76. Write a program that sets the frequency of the LCD backlight to as close to 100 Hz as possible when PWMPERx is set to 100, and make the duty cycle swing from 1% to 99%, repeating continuously, with 10 ms delay between changes in duty cycle. Display a word or two on the LCD so you can see your program in action.

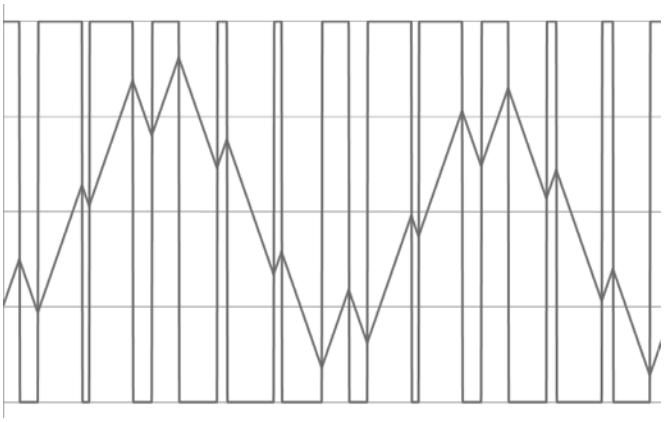
Learning Exercise 77. Why did it make sense for the designers of this board choose to use channels 0, 1, and 4 instead of 0, 1, and 2 to control the RGB LED?

Learning Exercise 78. Write a program that makes the RGB LED go RED for a second, GREEN for a second, BLUE for a second, then all three colours together for a second, repeating continuously. Set the frequency for each channel to 100 Hz, and use a duty cycle of about 10% to prevent retinal damage (that LED is bright!)

Depending on what your instructor wants you to do with the PWM module, you may be asked to write subroutines and put them in a library called PWM_Lib.inc. There are so many possibilities that a single initialization routine wouldn't make sense.

True Pulse-Width Modulation

For many applications, a pulse-width modulator is set to operate at a fairly high but constant frequency, and the pulse width, or duty cycle, is modified (modulated) to produce a varying average voltage. The result is fed into an integrator, which produces a signal averaged over time. If this average signal is constant, the result will be a DC control voltage; if the average signal follows a pattern set up in software, it could be any arbitrary waveform – sine wave, square wave, triangle wave, ramp wave, parabolas, you name it! The figure below shows this process.



The "jagged sinusoid" shown above would be the result of a simple integration of the PWM signal on which it is juxtaposed. Note that, when the duty cycle is greater than 50%, the integrated signal rises; when the duty cycle is less than 50%, the integrated signal falls.

In reality, the difference in frequency between the PWM signal and the integrated signal would be so great that the "jaggedness" would be greatly reduced.

Learning Exercise 79. Write a program that includes the following code. This sets the speaker channel, PWM6, to run at 62.5 kHz, then modulates the duty cycle at a constant rate to produce a ramp wave instead of a square wave after the filter built in to your board. With the speed of this device, we're somewhat limited. We end up with only 127 different possible duty cycles – not really good enough for audio, but ok for a demonstration.

```
Init:
            BCLR  PWMCLK,%01000000        ;set PWM6 to use B directly
            BCLR  PWMPRCLK,%01110000      ;Set B clock to Bus speed
            MOVB  #128,PWMPER6            ;128 clock cycles = 62.5 kHz
            MOVB  #64,PWMDTY6             ;initially 50% duty cycle
            BSET  PWME,%01000000          ;turn on PWM6
```

If you run this routine, you should hear nothing, as 62.5 kHz is too high a frequency for the human ear. However, you should be able to verify its action using an oscilloscope.

Learning Exercise 80. Using an oscilloscope, verify that your signal is correct.

Now, we'll add repetitive code to change the duty cycle "on the fly" to create the desired signal, without changing the PWM frequency. For more complicated signals, we would do this using a look-up table. For now, we'll simply use a bounded ramp – count up until we reach a maximum, and start again. We'll use our Interval Timer to keep the frequency of our modulating signal accurate.

```
            CLRB                  ;divide by 1 prescaler in interval timer
```

```
            LDX   #200              ;200 *125 ns = 25 us interval per sample
            JSR   IntXInitB         ;first interval
Ramp:       LDAA  #1                ;bottom limit
UpRamp:     STAA  PWMDTY6           ;set duty cycle
            JSR   IntX              ;next interval
            INCA
            BMI   Ramp              ;reached 128? Restart ramp
            BRA   UpRamp            ;otherwise, keep ramping
```

Make sure you understand what this code does, particularly the decision it makes.

Run your program (with the speaker disabled), and observe the PWM signal on the oscilloscope.

Enable the speaker, run your program, and be amazed. The sound produced should be somewhat more mellow than the square waves you've been producing up to this point. You can observe the integrated signal by probing TP1 with JP1 installed.

Try this with "LDX #5000" instead for a cleaner display (at 12.6 Hz, which is too low a frequency to hear properly) on the oscilloscope.

Using a Simple Lookup Table

In order to create a waveform with a different shape from the simple ramp you created above, you will probably need to read values from a lookup table. This can be done a couple of ways: one is to start by pointing the X or Y register to the beginning of the lookup table, then increment the register to advance from one character to the next; the other is to keep the X or Y register value constant, pointing to the beginning of the lookup table, and advance the index value – the number of addresses from the beginning of the table. Let's use the second method.

Learning Exercise 81. Determine appropriate values for the duty cycle in order to make an approximation of a sine wave. Using Excel or similar spreadsheet software, create a table based on the formula shown in the following cells, up to an angle of 358º:

Angle	Value
0	=INT(63*SIN(RADIANS(A2))+64)
2	=INT(63*SIN(RADIANS(A3))+64)
4	=INT(63*SIN(RADIANS(A4))+64)

Some explanations: Excel works in radians only, hence the conversion of angles to radians; the resulting values will range from 63+64 = 127 to –63+64 = 1, giving us all positive values in a sine wave with an offset of 64; we want the nearest integer, since the 9S12 microcontroller doesn't use floating-point decimals; we don't include 360º, as that's the same as 0º in our repeating sine wave.

Learning Exercise 82. Figure out a way to take the values you've calculated and put them into a table in Code Warrior. Call your table "Wave:". You will want to put probably eight or sixteen values per row, and you will need to start each row with DC.B.

Learning Exercise 83. Write code that accesses the values in this table using the following version of the load accumulator command: LDAB A,Y where Accumulator A is used as a counter to move through the table. Make sure you limit the count so only the 180 values in your table are accessed. Use a 25 us interval timer, and complete your code to generate the sine wave defined by the table. Check your results using the oscilloscope to probe TP1 with JP1 installed.

Interrupts

Up to this point, you've been using a technique called "polling" when designing your software for the 9S12. In this system, you check each peripheral on a regular basis to see if it needs servicing. So, when you use your switches, you check to see if any of them are pressed as part of your program; in your timer routines, you sit and wait until a flag is set to let you know that the time interval has elapsed; when you are connected to a dumb terminal through the SCI port, you check to see if a key has been pressed each time you go through that part of your program.

Interrupt programming unleashes a level of power you've experienced in your C# programming – the ability to have one process running and having another process temporarily take control as the result of some event that occurs, such as the click of a mouse button.

Here's an analogy to show you the difference between polling and interrupts. In a classroom, an instructor can constantly walk around the lab benches asking each student, one at a time, if they need help: That's polling, and it keeps the instructor busy all class period long. Or, the instructor can sit at his desk getting caught up on *marking*, while students put up their hands to call him over when they need help: That's using interrupts.

With polling, there's no problem figuring out where to go next in the program: everything is linear, and if the routine that requires your attention is in a subroutine, the program always leaves from a defined point to go to the subroutine, and always returns to where it left from to continue on.

However, an interrupt can happen at any time and can be completely unpredictable. The program control must be able to leave what it's doing, service the interrupt, then pick up where it was at as if nothing had happened in between. This is a good time to compare what's required for *branches*, *subroutines*, and *interrupts*.

Action	Going To	Returning From
Branch	Jump to branch address	N/A
Subroutine	Stack the return point address Jump to subroutine address	Retrieve return point from stack (pushes and pulls handled in code)
Interrupt	Stack everything – return address, Y, X, A, B, CCR	Retrieve everything – program continues as if nothing happened

.Note: Since everything is retrieved from the stack when returning from an interrupt, you can't use A, B, D, X, Y, or any of the condition code register bits to return information from an interrupt. You must place information in global variables instead.

It's also important to know how to get to and return from these types of routines:

Action	Going To	Returning From
Branch	JMP, BRA, LBRA, BRSET, BRCLR, BEQ, BNE, BCC, BCS, BVC, BVS, BGE, BGT, BLE, BPL, BMI, BLT, BHI, BLO, BHS, BLS, DBEQ, DBNE, IBEQ, IBNE	N/A
Subroutine	BSR, JSR	RTS
Interrupt	Vector table	RTI

So, how do interrupts work? Each item that can be used as an interrupt will have an interrupt enable, an interrupt flag, and an interrupt vector address associated with it. The interrupt vector is a hard-coded address that the microprocessor uses to determine where to transfer control to when a particular interrupt flag is set. The interrupt vector must be programmed with the address of the desired routine (called an interrupt service routine or ISR).

Here's how to set up a program to use an interrupt:

1. Write the interrupt service routine (ISR), preferably collected with other ISRs under a header that sets them apart from the rest of the code.
2. Start the ISR with an informative label that the Assembler will interpret as the entry address.
3. Inside the ISR, make sure you clear the interrupt flag that brought you here. That usually means writing a "1" to that flag in the associated flag register.
4. Make sure you exit the ISR with RTI – never try to exit any other way!
5. Associate the starting address of the ISR with the appropriate Interrupt Vector.
6. In the initialization of your program, enable the specific interrupt for this routine.
7. Enable the maskable interrupts with CLI. (Incidentally, SEI turns interrupts off.)

Learning Exercise 84. Start a new project, and follow the steps below to make an interrupt-driven timer.

1. Enter the following code below where you would normally put subroutines:

```
;*********************************************************************
;*    Interrupt Service Routines                                     *
;*********************************************************************

;*********************************************************************
;*          Timer_ISR                                                *
;*100 ms time interval using TC0.  Requires initial setup:          *
;*    Prescaler:  2^7                                                *
;*    Count interval:  6250                                          *
;*Also requires a 16-bit variable called TimCounter                  *
;*********************************************************************

Timer_ISR:
            BSET  TFLG1,%00000001   ;clear interrupt flag
            LDD   #6250             ;place counter delay value in D
            ADDD  TC0               ;add old event target
            STD   TC0               ;new event target
            LDD   TimCounter        ;increment the counter variable
            ADDD  #1
            STD   TimCounter
            JSR   SevSeg_Top4       ;display the new value
            RTI
```

You've just covered parts 1 to 4 above – wrote the ISR, gave it a label, cleared the interrupt flag, and ended with RTI.

2. In the datasheet for the 9S12XDP512, locate the Interrupt Vector Table (in the current version, page 73 and following). Find the vector address for "Enhanced capture timer channel 0". The default "Vector Base" is $FF00. At the end of your skeleton file, you'll find a section called "Interrupt Vectors". Set the Assembler to this vector address using an ORG statement, then tell it what to point to (Timer_ISR). You can follow, as an example, the interrupt vector that tells your micro where to start after a Reset.

3. The interrupt enable register for the timer is called TIE, and, as expected, the flag for Channel 0 is the least significant bit. At the top of your Main code, you'll be doing some initializations, the last of which should be the following in order to enable the Channel 0 interrupt and turn on access to all enabled maskable interrupts. (Vectors like Reset are non-maskable, so we don't need to use CLI to activate it.)

```
BSET  TIE,%00000001     ;enable OC0 interrupt
CLI                     ;enable interrupts
```

Your interrupt routine is now ready to run, except for the setups required in the Main program.

4. Create the required 16-bit variable TimCounter.
5. Make the appropriate initializations. To save you time, here they are. Put this above the commands to enable interrupts, so the last thing you do is "turn on the power".

```
;Initializations
            JSR   SevSeg_Init

            MOVB  #%10000000,TSCR1  ;enable timer module 0
            LDAB  #%00000111        ;2^7 prescaler
            STAB  TSCR2             ;prescaler set to Bus/(2^B)
            MOVB  #%00000001,TIOS   ;set IOS0 for output compare
            MOVB  #%00000001,TCTL2  ;toggle mode for PT0
            LDD   #6250             ;100 ms
            ADDD  TCNT              ;set new event timer value based on clock
            STD   TC0
            BSET  TFLG1,%00000001   ;clear flag
```

All we need now is a dummy "Main" program to run. Let's just start with

```
            BRA   *
```

Try it out! Notice that, even though your actual program is doing nothing, the counter is advancing because the interrupt service routine is responding to the interrupt generated every 100 ms. Let's play with this a bit.

6. In the Main program, replace the "BRA *" line with the following simple delay loop:

```
            LDD   #0
RegularLoop:
            LDX   #0
            DBNE  X,*
            ADDD  #1
            JSR   SevSeg_Low4
            BRA   RegularLoop
```

When you run your code, you should get two up-counters, running at different speeds – one from the "RegularLoop" program and the other one from the ISR.

7. Now, replace the "CLI" with "SEI" to disable the interrupts. Explain the results. Make sure you go back to "CLI" before moving on.

How about running two Interrupt Service Routines simultaneously? Let's have your micro respond to interrupts generated by the SCI port when you type on the keyboard.

8. Add the following ISR:

```
;*******************************************************************
;*          SCI_Echo_ISR                                           *
;*Receives a character from the keyboard and echoes it to terminal  *
;*Requires an initialization routine for the SCI port               *
;*******************************************************************

SCI_Echo_ISR:
            BSET  SCI0SR1,%00100000 ;clear interrupt flag
            LDAA  SCI0DRL           ;receive character
            JSR   SCI0TxByte        ;display character
            RTI
```

9. From the Interrupt Vector Table, find the vector address for SCI0. Under "Interrupt Vectors, set the Assembler to this vector address using an ORG statement, then tell it what to point to (SCI_Echo_ISR).
10. The interrupt enable register for the SCI0 port is called SCI0CR2. You want to enable only the Receive Interrupt Enable (RIE) bit in it. See if you can track this one down using all the documentation you've been provided, and enable it in your initializations prior to the CLI command.
11. Also, before you run the CLI command, you should make sure the flag is cleared.

Your interrupt routine is now ready to run, except for the setups required in the Main program. You will have to add your "SCI0_Lib.inc" to the mix, and you'll need to run an initialization routine. Also, don't forget to hook up the COMM port cable!

If all went well, you should be running the equivalent of three independent programs on your micro board! The only one that would show any slow-down from interference with the other two is the simple clock cycle-counting delay loop in the Main routine, because every jump to an ISR adds time to the loop. Since these jumps can be completely unpredictable, this drives home the need to use the timer module in interval mode if you want to have consistent, predictable timing.

12. Let's go for broke, and add another interrupt. This one requires a jumper (wire-wrap) from Pin 68 (the MID switch) to Pin 22 (PortJ0). That's because, in this version of the 9S12, PortAD doesn't have an interrupt controller associated with it, but PortJ does, so we'll run the MID switch into PortJ0 as well as into Port AD.
13. Add a "Switch_ISR" that just handles the appropriate flag and clears the 16-bit variable "TimCounter" used by the timer. Hint: It's 16-bit!
14. Port J requires some setup. In the datasheet, locate the related information in the chapter on the Port Integration Module (in the current version, page 952 and following). Make sure PTJ0 is an input using DDRJ; set the polarity of the interrupt for Rising Edge using PPSJ; handle the flag, found in PIFJ; and enable the interrupt in PIEJ.
15. You should be ready to run this program – it should clear the upper line of the seven segment display every time you press the MID switch.
16. Notice that you don't need a variable to keep track of the previous condition of the MID switch, and you don't need to debounce the switch (although, in some situations, debouncing would still be a good idea). See if you can explain why using an interrupt has made switch handling so much simpler than polling.

Real-Time Loop

One formalized use of interrupts is Real-Time Loop Multiplexing. Unless your instructor has extra time available, you probably won't do an exercise on this. However, you should know how this is intended to work.

The idea is to have just one interrupt – a regular timer that establishes the loop interval. During each interval, a set number of tasks are handled in order, then the microcontroller is put into a power-down WAIT condition until the interval timer's interrupt occurs. Here's an example.

```
RTL:
      JSR         SevSegTask
      JSR         SecTask
      JSR         ADCDACTask
      JSR         VoutTask

      WAI

      BRA         RTL

;*************************************************************
;*          Timer Interrupt Service Request                  *
;*************************************************************
TIM_ISR:
      BSET        TFLG1,%00000001   ;clear interrupt
      LDD         #1250             ;10 ms interval
      ADDD        TC0
      STD         TC0               ;new interval
      RTI
```

The four tasks are in carefully-designed subroutines, and are accessed during each interval. The "WAI" command puts the microcontroller into a low-power sleep mode, but it still responds to the timer compare interrupt, which wakes it up and sends it to the beginning of the real-time loop.

Of course, for this system to work, the total time taken by the task subroutines must be less than the time interval.

This can be handled two ways.

- One is to make the interval long enough to handle the maximum time required by all the tasks. Sometimes this is unrealistic, and results in jittery code management.
- The other is to find ways to break up tasks that occasionally have long bursts of activity into smaller pieces. For example, if one task occasionally sends a long string of text to a dumb terminal through the SCI port, consider sending the string one character at a time, or maybe no more than 10 characters at a time, until the entire string is sent. This would require putting the string into a buffer and keeping track of the current location in the buffer.

A well-planned real-time loop multiplexing system is the perfect application for a microcontroller acting as the brains for a repetitive system, such as the "computer controls" in an automotive fuel injection and ignition system.

Programming in C

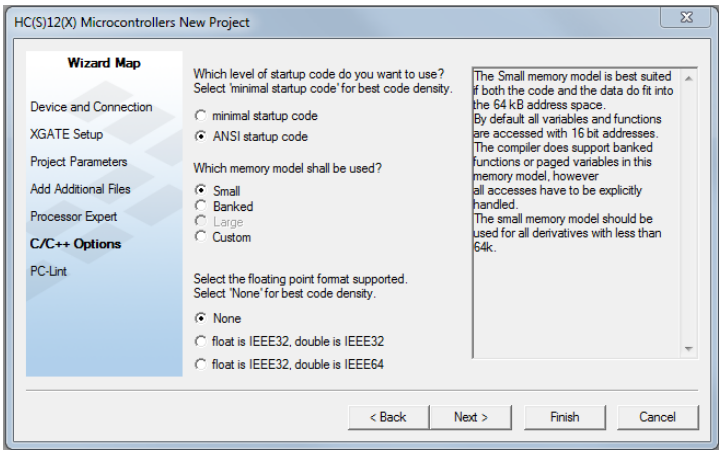
Every time you've started a new project, you've unchecked "C" and checked "Absolute Assembly" instead. You probably don't even think about it anymore; but what if ….

C is a much older language than the C# you've been working with. However, although Microsoft has tried to cloak the basics in Orwellian doublethink, you will probably find that there are some transferable concepts, fundamental operations, and syntax similarities that will make learning C for the 9S12 fairly easy. Unless your instructor has chosen to approach the order of this course differently, you should, at this point, have learned the fundamentals of programming in 9S12 Assembly Language, which is tightly linked to an understanding of the operation of the microprocessor and its peripherals. Programming in C also requires a clear understanding of the operation of the microcontroller, particularly its registers; however, the C cross-compiler used by Code Warrior is capable of handling some stuff so you don't have to worry about all the details.

In the next few exercises, we'll learn how to start a C project, how to write and run simple code, how to write and use "Functions" (these are what C# architects decided to refer to as "Methods"), how to write, include, and use uncompiled code libraries, and how to run an interrupt service routine. Sounds ambitious? Hopefully this will give you just enough of an introduction to be useful to you – perhaps now as well as in your fourth semester project.

Learning Exercise 85. Start a new project, following the steps below:

1. Follow the usual steps to start a new project, including selecting the appropriate derivative of the 9S12, the BDM interface you're using, and "Single Core" if you are using an XGate device.
2. Leave the "C" box checked, set the appropriate "Location" (your Projects folder), and give your project a name – something like "C_SW_LED". Don't hit "Finish", as there's one more screen we need to deal with.
3. When you get to the "C/C++ Options" page in the Wizard Map, select "ANSI startup code" and the "Small" memory model. Leave floating point format as "None" (on your own, you might want to try working with floating point values – the general ANSI C libraries support this, but it takes more memory). Now you can click "Finish" to create your project.



4. Open the "main.c" program. It will contain some basic code (a bit less annoying than the code they inserted in "main.asm", but still not entirely useful. Instead, create a "skel_C.txt" file like the one on the following page, replace the text in "main.c" with it, and tidy up the header information.

```
/*****************************************************************/
// HC12 Program:  YourProg - MiniExplanation
// Processor:           MC9S12XDP512
// Xtal Speed:          16 MHz
// Author:              This B. You
// Date:                LatestRevisionDate

// Details: A more detailed explanation of the program is entered here
/*****************************************************************/

#include <hidef.h>            /* common defines and macros */
#include "derivative.h"       /* derivative-specific definitions */
/*****************************************************************/
//          Library includes
/*****************************************************************/

//#include "Your_Lib.h"

/*****************************************************************/
//          Prototypes
/*****************************************************************/


/*****************************************************************/
//          Variables
/*****************************************************************/


/*****************************************************************/
//          Lookups
/*****************************************************************/


void main(void)
{
// main entry point
_DISABLE_COP();

/*****************************************************************/
// initializations
/*****************************************************************/


      for (;;)
      {
      //main program loop
      }
}

/*****************************************************************/
//          Functions
/*****************************************************************/


/*****************************************************************/
//          Interrupt Service Routines
/*****************************************************************/
```

Learning Exercise 86. Our sample project will count up on the LEDs at an observable rate controlled by an interval timer. Let's start by putting all the functionality into the main program loop. We'll use a couple of variables to show how they operate, as well. Under "Variables", create an *int* called "iTimeVal", and initialize it to 62500. We'll use this in our timer interval. Also create a *byte* called "bPrescaler", and initialize it to 5. (For this cross-compiler, binary coded values start with 0b, hex coded values start with 0x, and decimal values have no prefix. It might make sense to put the bPrescaler in binary so you remember it has to fit into the last three bits of TSCR2.)

Learning Exercise 87. Why did we use *int* and *byte* as we did in the previous exercise?

Notice that we assigned initial values to *variables* above. In Assembly Language, this would be a no-no, as these values would be written into RAM, and would disappear when we turn the power off. The C cross-compiler will create the variable space in RAM, but, in ROM, it will write its own initialization routine that copies the desired initialization values from ROM into RAM. This can come in handy, particularly if you've got a basic string in which you want to be able to manipulate particular characters under program control. In Assembly Language, you would have to write your own string copy routine that copies the ROM version into RAM character by character. The C cross-compiler does this automatically. Interesting point: If you define something as a *const*, the cross-compiler will likely just insert its value into the code where it's needed rather than providing memory space for it.

Learning Exercise 88. In the initializations section, make the LED bits of DDR1AD1 into outputs and the Switch bits inputs; enable the inputs in ATD1DIEN1; and make sure the LEDs are all off to begin with.

Learning Exercise 89. Also in the initialization section, set up the timer for half-second intervals as follows: Turn on the timer module using TSCR1; To ensure you don't mess up any bits in the prescaler other than the ones we want to change, 'AND' TSCR2 with a mask that clears only the prescaler bits, then 'OR' TSCR2 with the contents of the variable "bPrescaler" that we defined earlier; Set up TIOS for output compare on Channel 0 (don't mess with the other channels); Set up the first interval to be the current value of TCNT plus "iTimeVal"; and clear the appropriate bit in TFLG1 without messing up the other bits. If you want, you can also set up TCTL2 to toggle the PT0 pin.

Learning Exercise 90. Inside the endless loop "for (;;){}", do a binary up-count on the LEDs by adding 0b00100000 to the current value of PT1AD1. (Incidentally, there are various ways to create an endless loop in C, but the "for" loop shown is nice because it doesn't generate any warning statements.)

Learning Exercise 91. Also inside the endless loop, put in a routine that blocks until timer channel 0's flag comes true, at which point it adds "iTimeVal" to the previous value of TC0 and clears the flag. Run your code, and debug if necessary.

Wanna make some of the stuff you just wrote into ***functions***? Of course you do! Let's do that with the two timer routines. This might be a good time to save what you've done so far as a separate file, for your records.

Learning Exercise 92. First, we need a prototype for each. For this cross-compiler, the prototypes need to specify parameters passed to and from a function, even if they are all void. Remember that for later, as the functions we're going to create actually pass parameters. Add the following prototypes in the appropriate space at the top of your skeleton file:

```
void TimerSetup(int iTime, byte bPre);
void TimerInterval(int iTime);
```

Learning Exercise 93. Move the code for these two functions into the "Functions" area with the appropriate formatting and use of the variables indicated in the prototype.

Learning Exercise 94. Call these functions from the main program by passing the variables indicated. Run your program and debug if necessary.

Next step: Let's make a ***library*** of functions and link that into our program. Again, you might want to save your work to this point as a separate file, for your records.

This time, we'll create a couple of functions related to the LEDs and switches, and we'll expand our program to respond to a particular switch press. As with other flavours of C, to use an uncompiled library we will need two files: a header file (SW_LED_Lib.h) and a C code file (SW_LED_Lib.c).

Learning Exercise 95. Create a suitable header file with the following two headers (prototypes) and a generally-informative file header as follows:

```
//Switches and LEDs
//Processor:  MC9S12SCP512
//Crystal:  16 MHz
//by:  This B. You
//date:  Now

void SW_LED_Init(void);
byte SwCk(void);
```

Learning Exercise 96. Now, create the code file. It will have to start as follows:

```
//Switches and LEDs
//Processor:  MC9S12SCP512
//Crystal:  16 MHz
//by:  This B. You
//date:  Now

#include <hidef.h>
#include "derivative.h"
#include "SW_LED_Lib.h"
```

The last "include" adds the prototypes to the main file. Keep going with the required code for the initialization routine, which you will take out of your main code.

Learning Exercise 97. I'll give you the code for the SwCk routine. Make sure you understand what it's doing, particularly the way it handles assembly commands! You would also be wise to put in some informative comments ….

```
byte SwCk(void)
{
      byte bSample1=1;
      byte bSample2=0;

      while(bSample1!=bSample2)
      {
            bSample1=PT1AD1&0b00011111;
            asm   LDX   #26667;     /* 26667 x 3 cycles x 125 ns = 10 ms */
            asm   DBNE  X,*;
            bSample2=PT1AD1&0b00011111;
      }
      return bSample1;
}
```

Learning Exercise 98. Back in your main code, you will need to access the two library files you've created. Under "Library includes", put

#include "SW_LED_Lib.h".

Now, in the Project Inspector window under "Sources", add the .c file.

Also in the Project Inspector window under "Includes", add the .h file.

Learning Exercise 99. Modify your code so that it calls the switch and LED initialization routine from the library, and make sure everything works.

Learning Exercise 100. Now, let's add a quick routine to use the other library component you added – the debounced switch. Inside the endless loop, write an IF ELSE statement that clears the three LEDs if the MID switch is detected in the value returned by SwCk, otherwise it counts up in binary on the LEDs. Run and, if necessary, debug your program.

Ok, one last thing: Let's turn your interval timer into an Interrupt Service Routine. One more time, save your current work as a separate file, for your records.

This is a bit different from the way we handled things in Assembly Language, as we rely on something called pragma interrupt handling. Also, since we're not allowed to pass parameters to or from an interrupt (remember everything get stacked), we'll have to use global variables for anything we want to send to or receive from an interrupt routine.

Remember that there are seven things we need to do to handle an interrupt properly:

1. Write the interrupt service routine (ISR), preferably collected with other ISRs under a header that sets them apart from the rest of the code.
2. Start the ISR with an informative label that the compiler will interpret as the entry address.
3. Inside the ISR, make sure you clear the interrupt flag that brought you here. That usually means writing a "1" to that flag in the associated flag register.
4. Make sure you exit the ISR properly (not so hard in C).
5. Associate the ISR with the appropriate Interrupt Vector.
6. In the initialization of your program, enable the specific interrupt for this routine.
7. Enable the maskable interrupts.

Learning Exercise 101. Here are steps 1 to 5.

```
/*****************************************************************/
//          Interrupt Service Routines
/*****************************************************************/

interrupt VectorNumber_Vtimch0 void TimerInterval(void)
{
  TC0 =(int)(iTimeVal+TC0);   // next time
  TFLG1 |= 0b00000001;        // acknowledge interrupt
}
```

"VectorNumber_Vtimch0" is interpreted by the compiler using the MC9S12SDP512.c file to point to the correct interrupt vector for timer channel 0. "TimerInterval" is the name of our ISR. Notice it is "void (void)", since we can't pass parameters to or from it.

We tell the compiler to cast the result of "iTimeVal+TC0" to *int*, because the compiler knows the result could be bigger than two bytes, and will give us a warning otherwise.

After clearing the flag, we simply end with a curly bracket, and the compiler knows to use RTI to end the routine. So, that's five out of the seven requirements.

Back in the main program, we need to enable the interrupts (steps six and seven), and we also need to make sure the iTimeVal variable is global.

Learning Exercise 102. It probably makes sense to enable the Timer channel 0 interrupt in the TimerSetup function. Set up the appropriate value for TIE there.

Learning Exercise 103. In the initializations for the main program, enter the following:

```
EnableInterrupts;
```

Now for the iTimeVal variable: As it sits, it is a global variable, so we should be able to use it without any changes. However, in case there's a chance it could be changed by the program during operation, it might be wise to put "volatile" in front of "int" to prevent it from being mangled by an interrupt occurring while it is being changed.

Finally, let's change the endless loop in our main code so that it works in conjunction with the interrupt we've just designed. If you've been following this set of exercises, your main program should be pretty simple by now. Modify it so it looks like the following:

```
      for (;;)
      {
//main program loop
            if(SwCk()==0b00000001)
            {
                  PT1AD1&=0b00011111;
            }
            else
            {
                  PT1AD1+=0b00100000;
            }
            asm WAI;
      }
```

The "WAI" command puts the micro to sleep, waiting for the next interrupt from the timer. As long as the timer is initialized properly and interrupts are running, this program should work just the way your previous edition did.

That was a lot of work! See if you can persuade your instructor to give you marks for that!

A To D Conversion

The 9S12XDP512 has two fairly complex Analog to Digital Conversion blocks. Locate the block diagram in the datasheet (in the current version, page 126).

http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf

This peripheral is highly configurable, and can do a lot of things. Again, we'll just scratch the surface of its capabilities. Of the two available converters, we're going to be using ATD0.

Here are some of its features:

- It can run in single input or multiplexed input mode. In other words, it could be measuring up to 16 external voltages simultaneously.
- For a single input, it can take just one sample per reading or it can operate in continuous scan mode. This means you can either ask for a sample and wait for it, or you can have samples available all the time for faster reading. On top of this, you can choose to have the results placed in the matching ATD data register (DR), or the results can simply be placed in consecutive registers until they're all filled, then start at the top again.
- Sampling can either be clock driven or initiated by external trigger events.
- The results can be either 10-bit or 8-bit. 10-bit is better: just remember to read a two-byte word to get the result!
- The reference voltages, both top and bottom, can be set using external circuitry. In our case, V_{RL} is grounded, and V_{RH} is connected to the output of a REF02 that has a trimmer potentiometer connected to it. We'll set this to 5.120 V_{DC} to provide a nice step size.
- The sample rate is selectable. Fast sample rates allow for high-speed signals, but slow sample rates are more accurate.
- The input buffering of the signal is configurable.
- The data format is configurable. You can select signed or unsigned values (Single quadrant or 4-quadrant), and left or right justified values.

Learning Exercise 104. Connect a digital voltmeter between ground and VREF on your board, and adjust VR3 to set VREF to 5.120 V_{DC}.

Learning Exercise 105. We're going to input a signal to AN0. Using the schematic, determine which pin this is on your board.

Notice that we're using a port we've used before, which means you have to be careful not to mess up the functionality of the switches and LEDs when you're setting up the A to D converter. The switches and LEDs are on the upper byte of PORTAD, and our input channel is on the lower byte.

The full details for the registers we're using are found in Chapter 4 of the Data Sheet, available in Moodle.

Remember ATDDIEN0? We needed to enable the inputs in order to get a digital signal into them. However, to receive analog signals, we need to have ATDDIEN0 or, in our case, ATDDIEN1 set low for the input pins to enable analog inputs instead. Since the unit defaults to 0, we shouldn't have to worry about ATDDIEN1, unless somewhere else in software we've set these bits to "1".

In the datasheet, the registers are named “ATDCTLx”, etc. Since there are two A to D converters in this controller, we need to insert a “0” after “ATD” in each case to specify which one we’re using.

Let’s start our initialization. Locate the A to D control registers ATDCTL0 through ATCDTL5 in the datasheet. A summary of them appears, in the current version, on page 129.

The first two are for modes we’re not going to use. So, let’s start with ATD0CTL2.

Learning Exercise 106. We want to power up the A to D converter, run in fast flag mode (no need to write to the flag to clear it), halt in wait mode, operate without external triggering, and turn off interrupts. Determine what we should write to ATD0CTL2.

Learning Exercise 107. We want single-conversion mode, continuous writes to the next available data register, and we want the A to D converter to finish conversions before freezing on a break. Determine what we should write to ATD0CTL3.

Learning Exercise 108. We want to run this as a 10-bit converter, we’ll use four clocks per sample, and we want each sample to be at least 7 µs. Determine what we should write to ATDCTL4.

Learning Exercise 109. Write an initialization routine called “AtoD_Init” matching the above details. You can put it in a new library called AtoD_Lib.inc. You will need a delay of approximately 50 µs between writing to ATD0CTL2 and ATD0CTL3. Just use a simple countdown loop.

Learning Exercise 110. We’ve left writing to ATD0CTL5 out of the initialization routine because, in the single sample mode we’ve chosen, writing to ATD0CTL5 initiates a conversion sequence. So, we’ll do this each time we want to read the incoming value. Write a routine called “AtoD_AN0” as follows:

1. We want our data to be right-justified using unsigned numbers, we want single conversion using only one channel, and that channel is to be AN0. Determine what we should write to ATD0CTL5.
2. We need to wait until the SCF (sequence complete flag) is set in ATD0STAT0 before we can read Data Register
3. Once the flag is set, we can read the sixteen bit result (only the lower 10 are valid) from ATD0DR0 (for AN0).

Learning Exercise 111. Write a small program that initializes the A to D converter, samples AN0 once every 100 ms, and displays the raw data on the top line of the seven segment display. Very carefully set up a sine wave signal that varies from 0 to 5.12 V at a rate of about a tenth of a hertz. Connect this to AN0, and determine if your output is responding properly (what range are you expecting?). If so, display your results in millivolts on the bottom line. You’ll need to figure out the step size, then multiply your A to D reading appropriately before converting to BCD.

I²C Bus

There are a number of serial communication systems that can be used to communicate between devices on a printed circuit board or over short distances. One reason for doing this is to reduce the number of interconnections required between devices. We discussed this earlier, when we compared using more than 64 parallel wires (and, incidentally, more than 64 pins per device) with using RS-232 as a serial communication system that needed, as a minimum, three wires: transmit, receive, and ground. The 9S12XDP512 also provides other serial communication options: Serial Peripheral Interface, or SPI, Controller Area Network, or CAN Bus (typically used in automotive applications), and Inter-Integrated Circuit, or I²C Bus. Different peripherals are available for these different busses, depending on the desired application. What they share in common is the ability to put multiple devices on a single bus, which dramatically simplifies the hardware component, but complicates the software component somewhat.

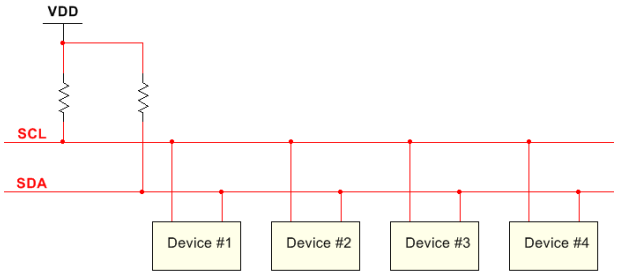
In previous versions of this course, we chose to work with the SPI bus, as it is an old workhorse that's not likely to go away anytime soon. However, when the current version of the microcontroller board was designed, we focused on the I²C bus. On your board, you will find the following I²C devices (they're tiny, so you might have to search for them):

- MPL3115A2 Precision Altimeter/Barometer/Thermometer
- 24AA512 Memory – 512kB of EEPROM
- M41T81 Real-time Clock (with battery backup – the battery is under the LCD display)

There's also a footprint available for a DS1624 Thermometer with 256 bytes of EEPROM memory that hasn't been installed – we decided the MPL3115A2 was more than enough.

Also, hopefully, there's a LTC2633HZ12 on an adapter board with wire-wrap connections available to it. This is a 12-bit Digital to Analog Converter (DAC) for you to use in generating high-quality analog signals. Links for all of these are available in Moodle.

The I²C bus requires, in addition to power and ground, two lines: SCL is the serial clock, and SDA is for bidirectional serial data transmission.



This bus is configured so any device can be "master" and take control of communication with any "slave" it chooses to talk to. Since only one device can talk at a time, this becomes a logistical issue: devices that aren't talking can't have any effect on the bus, or they will prevent other devices from communicating. If one device is holding the data LOW, no other device can drive it HIGH to send a different bit. This is achieved by making all of the connections to the bus ***open drain*** or ***open collector***. In your semiconductors course, you learned about at least one such device – the dedicated comparator. For these devices to work, an external pull-up resistor is needed. Internally, each device has a FET or BJT wired as a switch, but without an R_D or R_C, which we must provide. With the I²C bus, all the devices use the same pull-up resistor, which makes them act as *wired-OR* devices. When their transistors are turned OFF, the pull-up resistor pulls the line up to a logic HIGH. When any one of the devices turns on its transistor, the line pulls down to a logic LOW. So, as long as all devices rest with their transistors off, any single device can talk without interference from the rest.

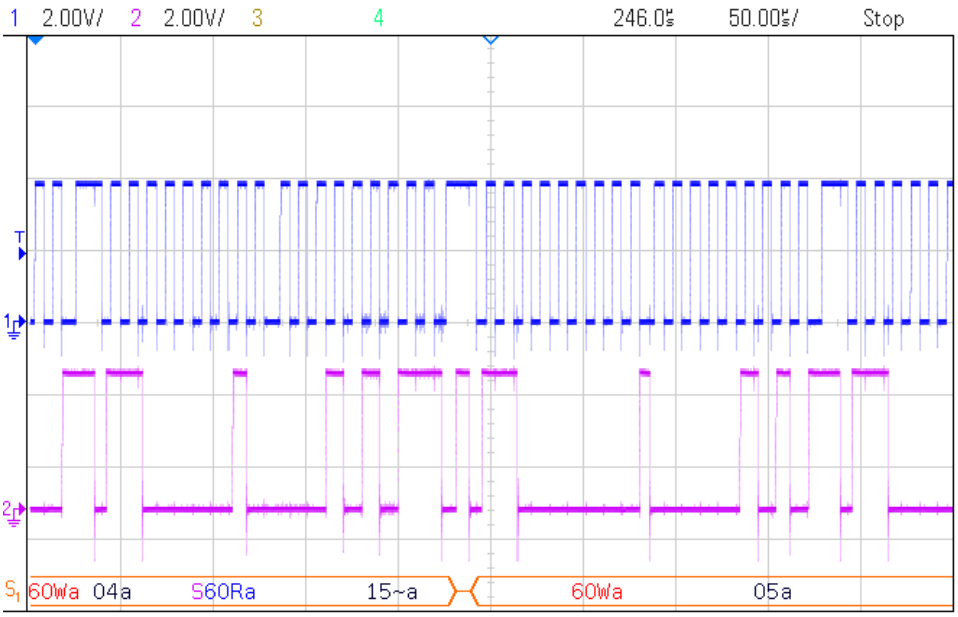
Another feature of this system is its ability to communicate at different speeds for different devices. The device talking at any point in time generates the clock, which synchronizes communication with the receiving device. The typical maximum speed for the I^2C bus is 100 kb/s, but some devices have been designed to handle up to 400 kb/s.

How do the devices know who's talking and who's listening? Each device has a unique 7-bit ***address***, partly built into the device and partly hard-wired when the board is built. For example, the LTC2633 DAC has, as the top bits of its address, 00100. There's one pin that makes possible three distinct addresses by controlling the bottom two bits. Here's how that works:

CA0 Condition	Lower two bits
Ground	00
Floating	01
V_{DD}	10

This means that you could have three LTC2633 DACs on the same bus with different addresses: 0010000, 0010001, and 0010010. Now for one of the things you need to know as a programmer: These seven bits are at the TOP of the address, and the least significant bit is used as a Read/Write line. So, in hexadecimal, these three addresses would be 0x20, 0x22, and 0x24. To confuse the issue a bit more, my oscilloscope refuses to conform to this addressing scheme, and would call them 0x10, 0x11, and 0x12!

Speaking of my oscilloscope, here's a screen shot of the 9S12 talking to a device with the seven-bit address 1100000 which we treat as 0xC0, but my scope calls 0x60.



The top trace is SCL. Notice that it starts and stops, depending on how the bus is being used. The lower trace is SCA, and contains communication both from the 9S12, acting as a "master", and the device at address 0xC0, acting as a slave. The line at the bottom shows that the master told the device ("0x60") it was going to Write something to it, sent 0x04 to it, restarted the clock, told the device it was going to Read from it, so the device put 0x15 on the bus. (The "a" and "~a" are ACK and NACK handshaking tools.) So, to use the bus, our micro has to do the following:

1. Check to see if the bus is available.
2. If it is, issue a "Start" command to indicate we're taking control.
3. Notify the desired slave by its address, indicating whether we're going to write to it or read from it.
4. Send a byte that contains the internal address of the register we want to put something in or take something out of.

Now, things go either of two ways, depending on whether we're writing or reading.

Writing

5. Send the data.
6. Issue a "Stop" to free up the bus for another device to take control (of course, this doesn't happen on our board, because there's just one device that has the brains to be "master", and that's our 9S12.

Reading

5. Send a "Restart", which allows us keep control of the bus, but lets us issue a new command to the slave of our choice.
6. Send the slave's address again, but this time indicating we're going to Read from it. (Usually, the contents of the register we indicated in step #4 will be waiting for us.)
7. Receive the data byte.
8. Issue a "Stop" to free up the bus.

You'll have to also check to see if data is actually available, and wait until communication is complete, etc., but for now that's the basic process.

Variants on the theme:

- If you need to write more than one byte to a device that knows what to do with that (e.g. auto-increment the internal address), you can just keep writing data bytes until you're done, then issue the "Stop" command. One example of this is for memory devices that require a 16-bit address. For these, we have to send two bytes to establish the internal memory location we're interested in. Then, we can keep sending sequential bytes until we've stored everything we intended to store.
- If you need to read more than one byte from a device that knows what to do with that, you can just keep reading data bytes until you're done before issuing the "Stop" command.

We won't investigate these variants, but be aware they are possible in case you need them.

So, now that you've got just a tiny bit of information on how the I^2C bus works, let's put it to work.

For this exercise, let's do our programming in C. This means you'll need to write a simplified C version of your seven segment display library, so we might as well do that first.

Learning Exercise 112. Write a header file and uncompiled library file for your seven segment display. You'll need the following:

```
void SevSeg_Init(void);
void SevSeg_Char(byte bChar, byte bDigit);
void SevSeg_Two(byte bChars, byte bDigit);
```

If you want, you could also include the following, and put their functionality in SevSeg_Init.

```
void SevSeg_BlChar(byte bDigit);
void SevSeg_BlAll(void);
```

Learning Exercise 113. Start a new library and library header called "IIC0_Lib". That's because there are two I^2C busses on our controller, and the devices on board are wired up to I^2C-0. These are the components you'll eventually have in it:

```
void IIC0_Init(void);
void IIC0_Write(byte bAddr, byte bReg, byte bData);
byte IIC0_Read(byte bAddr, byte bReg);    /*Optional*/
```

Learning Exercise 114. The Init routine sets up the I^2C-0 port in the 9S12XDP512. You'll need to find the I^2C chapter in the "Data Sheet". In the current version, it's Chapter 9. Here's the general link for your convenience:

http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf

You'll find there are five registers used by the I^2C controller: a Slave Address Register, a Frequency Divider Register, Control Register, a Status Register, and a Data Register.

IIC0_IBAD – We don't have to worry about this one, as we'll always be the Master, not the Slave.

IIC0_IBFD – This is one weird register. To set it up, you need to know what the requirements for the slowest device on the bus will be, then pick a value that matches. This register sets up the clock speed and how many clock cycles will be used for SDA Hold, SCL Hold for Start and SCL Hold for Stop. There's a divider and a multiplier and a complicated formula, all of which can be bypassed by using their lookup table (Table 9-5 in the current version of the Data Sheet). We want to operate at 100 kHz, with 20 cycles for SDA Hold, 32 cycles for SCL Hold for Start, and 42 cycles for SCL Hold for Stop. Figure out what we need to put in IIC0_IBFD, given that the bus clock is 8 MHz, and make it so.

IIC0_IBCR – We want to enable I^2C, turn off interrupts, and operate normally in WAIT mode. The rest of the bits can be 0 for now. One hitch: The Data Sheet says that I^2C must be enabled before changing any of the other bits in this register, so you'll have to turn that bit ON first by itself, then make sure the interrupts and WAIT mode bits are turned OFF after that.

The other two registers are used when we communicate on the bus, along with changes we'll make in the Control register on-the-fly.

Learning Exercise 115. As you can see from the header, the "Write" routine needs to be supplied with a device address, an internal address for the register we're interested in, and a byte of data to put in that register. Here's the procedure:

1. Watch the Status Register (IIC0_IBSR) to see when the bus is Not Busy. Use a While loop and some Boolean logic to check for just one bit.
2. Once the bus is free, change the micro to "Master" mode, set to "Transmit". These bits are in the Control Register.
3. Place the device address on the bus with the LSB set to "Write" mode.
4. Wait for the Byte Transfer Complete process is indicated on the IBIF flag of the Status Register.
5. Clear the IBIF flag by writing a "1" to it.
6. Repeat the last three steps, but this time with the internal address.
7. Repeat, but this time with the data byte.
8. "Stop" transmitting and exit "Master" mode, using the Control Register.

Since we're just going to use the DAC that's installed on your board for now, you don't need a "Read" routine. However, the following exercise will make it possible for you to read information from other devices on the board, like the Altimeter/Barometer/Thermometer and the Real-Time Clock.

Learning Exercise 116. (Optional) As you can see from the header, the "Read" routine needs to be supplied with a device address and an internal address for the register we're interested in. The contents of that register are returned to the main program. Here's the procedure, the first part of which you did in your "Write" routine:

1. Watch the Status Register (IIC0_IBSR) to see when the bus is Not Busy.
2. Once the bus is free, change the micro to "Master" mode, set to "Transmit.
3. Place the device address on the bus with the LSB in "Write" mode.
4. Wait for the IBIF flag of the Status Register.
5. Clear the IBIF flag.
6. Repeat the last three steps, but this time with the internal address.
7. Now, using the Control Register, issue a "Restart" command.
8. Place the device address on the bus with the LSB in "Read" mode.
9. Wait for the IBIF flag of the Status Register.
10. Clear the IBIF flag.
11. Using the Control Register, get ready to Receive a byte. Here's a new piece of information: the last byte received from a device is not supposed to have an ACK following it, so we need to indicate that no ACK is required. Since you need to SET one bit and CLEAR another bit, this will take two steps. Deal with ACK first.
12. Here's a curious fact: in order to initiate a Read, you need to read the Data Register (IIC0_IBDR) once, which will generate garbage, before you move on.
13. Next, you wait for the IBIF flag, but you don't clear it yet.
14. Instead, you "Stop" by exiting "Master" mode, using the Control Register.
15. Now, clear the flag.
16. Finally, you can read the real data out of the Data Register and return it to the main program.

In order to use the EEPROM on your board, you would need to make 16-bit address versions of these two routines. You might also want to make versions of these routines that could write or read a bunch of sequential bytes at once. For this, you would need to figure out how to create data arrays and how to pass pointers to these arrays – welcome to C!

I²C DAC

Learning Exercise 117. The DAC that's been added to your board is a dual 12-bit DAC, and it's been wired so that its internal reference is effectively 4.096 V. Determine the step size for this DAC.

Since it's a 12-bit device, it's obviously going to need more than one byte sent to it. In fact, it's going to need three bytes: A Command byte, a byte that contains the 8 upper bits, and a byte that contains the 4 lower bits followed by four zeros.

Learning Exercise 118. Write a new version of the I²C "Write" command that fits the following header:

```
void IIC0_WriteDACA(byte bAddr, byte bCommand, int iData);
```

We'll send the address, although it should always be 0x20. Assume that the user (you) will know what to put in the Command; however, the Data part of this will come to you right-justified: in other words, it will be in the lower 12 bits of iData rather than in the upper 12 bits, where this DAC needs it to be. Remember that you can only send 8 bits at a time on the I²C bus.

The Command byte is constructed as follows:

- The first four bits form the command.
- The last four bits determine which DAC channel or channels the command and data are destined for.

The data sheet for the LTC2633 tells you what you need to know. Here's a link:

http://cds.linear.com/docs/en/datasheet/2633fb.pdf

Learning Exercise 119. The device defaults to using the internal reference, so you don't have to send a command to specify the reference. However, you do want to "Write To and Update" DAC A. Figure out what the Command byte should be.

Learning Exercise 120. You should now be able to send data to your DAC, and the corresponding analog voltage will appear at VoutA, which is Pin 5 of the DAC. Monitor this with a DC-coupled oscilloscope, while you come up with creative ways of manipulating the binary data you're feeding it. You could even read an analog signal using your A to D converter, manipulate it to be the right number of bits, and then spit it out on the DAC to see if you get a matching voltage (keep in mind that the ranges of the two devices are different).

Parting Words

You have now touched on some of the capabilities of a very powerful microcontroller. You've learned, with varying levels of proficiency, how to use a wide range of peripherals, both internal to the microcontroller, and external, connected through a number of different interfaces. In addition, you've learned how to program the device in its native Assembly Language and in C. You know enough about electricity and electronics to be dangerous. With a bit of ingenuity, you could do some serious design work. Go forth and build things!