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
- USBDM Pod or BDM Pod and "A to B" USB Cable
- CodeWarrior

Rationale

Well-structured and documented code results in dependable and maintainable systems.

Expected Outcomes

The following course outcome will be addressed by this module:
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 through programming the 9S12X using ANSI C.

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. This kind of activity often uses 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.

Disclaimer

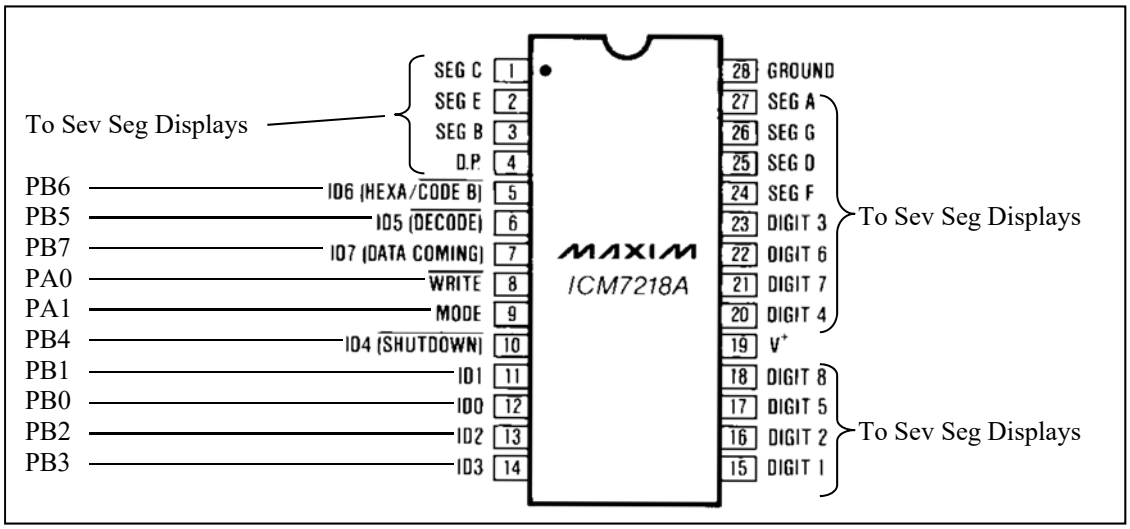
In earlier years in CNT, all of the programmatic control of the peripherals was done using variants of Assembly Language. Recently, the course has migrated to a heavier emphasis on ANSI C. Since the material developed in Assembly Language is still valid for the microcontroller you're currently using, the associated material has not been removed. You may find it useful, either if you choose to write some code in Assembly Language or if you choose to use it as reference material to help you gain a broader understanding of what is being done in the ANSI C adaptations.

Interfacing the ICM7218A 8-Digit LED Display Driver

At this point it is difficult for you to display program output in a very meaningful way. The 8-digit LED seven-segment 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 9S12X. 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 9S12X 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.

The code below is an S12XCPU Assembly Language initialization subroutine for the ICM7218A. To begin with, the two 9S12X ports attached to the device need to be set up as outputs. Good design procedure indicates that the data on the port should be set to a known condition that will have minimal effect on the connected device when the port is enabled as outputs, as shown in the following code.

```
;*******************************************************************
;*          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 using 8-digit sequential commands
;*
;*******************************************************************

SevSeg_Init:
            BSET    PORTA,%00000011     ;resting state:  mode and /write HIGH
            BSET    DDRA, %00000011     ;make A1 (mode) and A0 (/write) outputs

            MOVB    #%11111111,DDRB     ;make all PORTB outputs

            ;JSR        SevSeg_BlAll

            RTS
```

Note the use of "BSET" for DDRA and PORTA. By only affecting two bits in this register, the rest of PORTA is left untouched, which means it can be used for other purposes if desired.

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 when the device is first accessed, so the last line in the header is a lie at this point – this line could be included in the actual code once a subroutine called "SevSeg_BlAll" was written.

The following pages will provide you with a working knowledge of the operation of the ICM7218A seven-segment display driver, so that you can write code to display values on the seven segment display array.

The Maxim ICM7218A data sheet can be found here:
http://datasheets.maximintegrated.com/en/ds/ICM7218-ICM7228.pdf

Here are the most important programming-related tables from this data sheet.

ICM7218A Programming Tables

8 Digit LED Display Driver

ICM7218/ICM7228

Table 1. Input Definitions, ICM7218A and ICM7218B
Note: Pin Configurations for the ICM7218A/B are shown on last page.

INPUT	PIN	STATE	FUNCTION
WRITE	8	High Low	Input Not Loaded Into Memory Input Loaded Into Memory
MODE	9	High Low	Loads Control Word on WR Loads Input Data on WR
ID0–ID2, DIGIT ADDRESS	12, 11, 13	High Low	Loads "one" Loads "zero"
ID3, BANK SELECT	14	High Low	Select RAM Bank A (Hex or Code B Select RAM Bank B Data only)
ID4, SHUTDOWN (MODE High)	10	High Low	Normal Operation Shutdown
ID5, DECODE/NO DECODE (MODE High)	6	High Low	No Decode Decode
ID6, HEX/CODE B (MODE High)	5	High Low	Hexadecimal Decoding Code B Decoding
ID7, DATA COMING (MODE High)	7	High Low	Data Coming (control word) No Data Coming (control word)
ID0–ID7, INPUT DATA (MODE Low)	5-7, 10-14	High Low	Loads "one" (Note 1) Loads "zero" (Note 1)

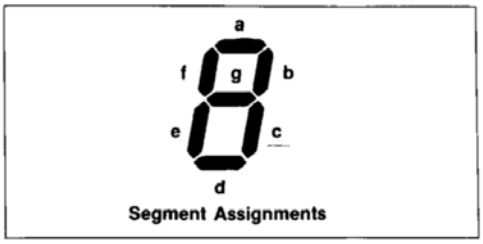
Note 1: A "zero" or low level on ID7 turns ON the decimal point. In the NO DECODE mode, a "one" or high input turns ON the corresponding segment, except for the decimal point which is turned OFF by a high level on ID7.

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 ("No Decode" – the digits map as shown below), but the easiest thing to do is have the device decode the input as Hex ("Decode" –> "Hexadecimal Decoding"). Another option is "Decode"–>"Code B Decoding", which produces a different set of characters including *H*, *E*, *L*, *P* and *Blank* but not the top six Hex numbers, *A* – *F*.

ICM7218/ICM7228

ID3	ID2	ID1	ID0	HEXADECIMAL	CODE B
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	A	-
1	0	1	1	b	E
1	1	0	0	C	H
1	1	0	1	d	L
1	1	1	0	E	P
1	1	1	1	F	(Blank)

Figure 7. Display Font



Segment Assignments

Microprocessor Interface, ICM7218C and ICM7218D

Data Input	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
Controlled Segment	Decimal Point	A	B	C	E	G	F	D

Sending Data to the ICM7218A

To get a digit on the display, you must send a control byte to the device that describes the digit address and mode of operation. After this you must send 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 a control byte or a data byte. Here are two timing diagrams from the datasheet that show how this works:

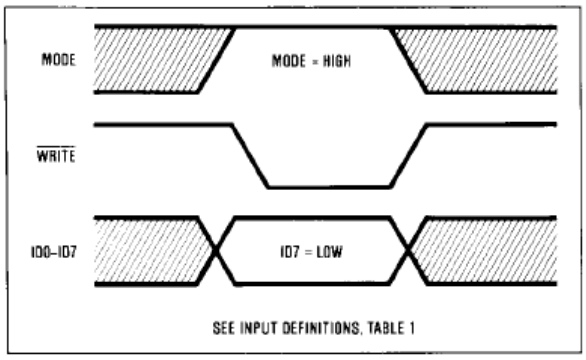


Figure 4. Control Word Update Timing—ICM7218A/B

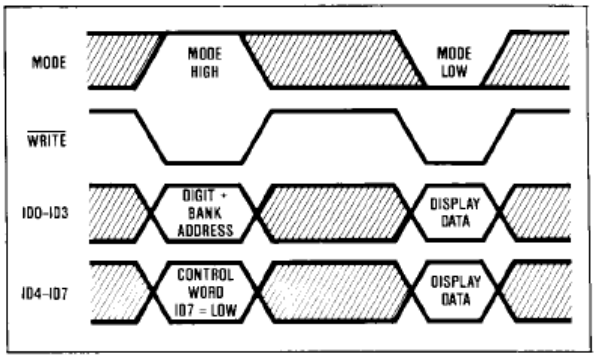


Figure 5. Single Digit Update Timing—ICM7218A/B

The procedure for sending a single control command (Figure 4) is as follows:

- Control:
  - Present control byte on GPIO data lines (Port B)
  - Set mode HIGH and write LOW (indicate that you are writing a control byte)
  - Set write HIGH (latches the control byte into the device)

The procedure for sending a single digit (Figure 5) requires both a control byte and a data byte, as follows:

- Control (containing the address and other control bits):
  - Present control byte on GPIO data lines (Port B)
  - Set mode HIGH and write LOW (indicate that you are writing a control byte)
  - Set write HIGH (latches the control byte into the device)
- Data:
  - Present data on GPIO data lines (Port B)
  - Set mode LOW and write LOW (indicate that you are writing a data byte)
  - Set write HIGH (latches the data byte into the device)

The bits in the control byte affect 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.)

Locate the table of Input Definitions (shown previously, from page 5 in the current data sheet). For standard writing of a hex character, you are interested in Hex mode, Bank A, normal operation. Using the table of Input Definitions, verify each of the bits in the control byte shown in the routine on the following page. Add an ANSI C version of this routine to your library.

Seven Segment Display Library Components

The following S12XCPU Assembly Language version of the initialization routine is included as a reference that can be used as a jumping point for all of the library components you will need to write for your ANSI C library. One thing you may find particularly useful from this version of the routine is the system used for bitwise commenting the command byte.

```
;*******************************************************************
;*      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 device limits,
;* and assumes that SevSeg_Init has been run already.
;*
;*******************************************************************
SevSeg_Char:
    PSHA
    PSHB

    ANDB  #%00000111    ;clean up address - only three bits valid
;            --------------"No data coming" means single digit mode
;           | -------------Hexadecimal Decoding
;           || ------------decode mode
;           ||| -----------not shutdown
;           |||| ----------memory bank A
;           ||||| ---------\
;           |||||| --------/ don't mess with the address
;           ||||||| ------/
;           ||||||||
    ORAB  #%01011000    ;add control: hex, decode, no SD, Bank A

    ANDA  #%00001111    ;prep the data digit:  clear upper nibble
    ORAA  #%10000000    ;... no decimal point

    STAB  PORTB     ;control byte placed on the data bus
    BCLR  PORTA,%00000001 ;mode still high (control), /write low
    BSET  PORTA,%00000011 ;resting state:  mode and /write HIGH

    STAA  PORTB     ;data byte placed on the data bus
    BCLR  PORTA,%00000011 ;mode low (data), /write low
    BSET  PORTA,%00000011 ;resting state:  mode and /write HIGH

    PULB
    PULA
    RTS
```

Note how "BSET" and "BCLR" were used so as to preserve the state of the upper six bits of PORTA, just in case this port is being used for some other purpose in your program. In ANSI C, you will need to use AND or OR operators to achieve the same functionality.

Seven-segment Display Control Using ANSI C

You are going to need three program files in a new project: a library header file (SevSeg_Lib.h), an uncompiled C library file (SevSeg_Lib.c), and the main program file within the project itself (its default name will be main.c).

SevSeg_Lib.h

The header file will probably be supplied to you by your instructor.

```
//Seven Segment Display Controller Library
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//June 2015

void SevSeg_Init(void);
void SevSeg_Char(unsigned char, unsigned char);     //character and digit address
void SevSeg_BlChar(unsigned char);                  //digit address
void SevSeg_BlAll(void);
void SevSeg_dChar(unsigned char, unsigned char);    //character and digit address
void SevSeg_Top4(unsigned int);                     //four chars compressed as four nibbles in an int
void SevSeg_Bot4(unsigned int);                     //four chars compressed as four nibbles in an int
void SevSeg_Cust(unsigned char, unsigned char);     //selected segments and digit address
```

SevSeg_Lib.c

Now comes the task of creating the actual library and the functions to match the prototypes in the header file. The library file needs to be linked to the files it will be using, which include the ones set up when a project is created: the ANSI C library hidef.h and the derivative.h file set up to provide the labels for all the ports in our particular 9S12X microcontroller. Also, if any of the functions call other functions in the library, it needs to be linked to itself. Notice again the different syntax for including a standard compiled ANSI C library and for including an uncompiled library. As previously discussed, you will need to add both the .h and .c files for the uncompiled libraries to the project, whereas the compiler will find the standard library without our help.

```
//Seven Segment Display Controller Library
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//June 2015

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

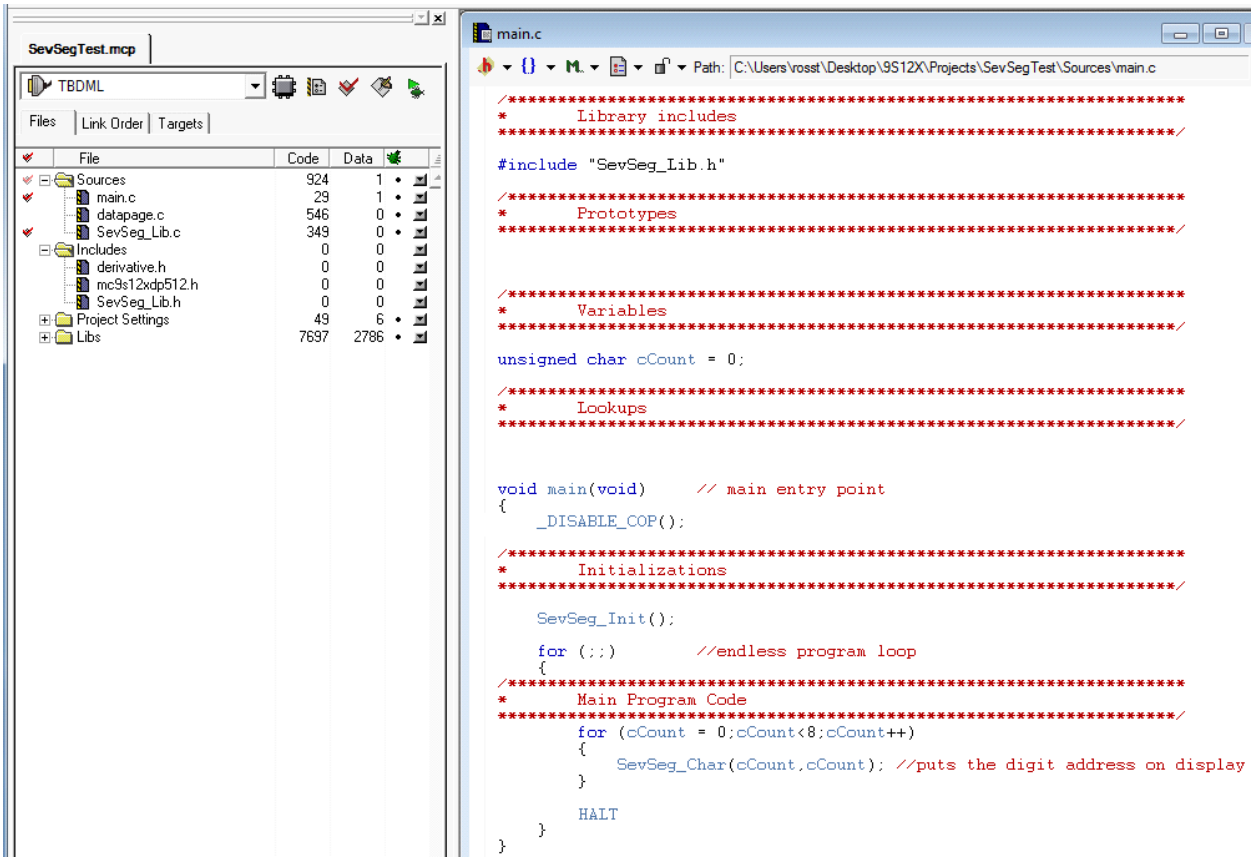
The first function you'll need is an initialization routine that does what the Assembly Language version did. In this case, we can do practically a line-by-line translation into C. This will not always be the case when moving from Assembly Language to C, as the thought processes involved in programming for the two languages is fundamentally different. Sometimes, Assembly Language provides the most succinct and efficient result, whereas other times the structured nature of C will allow the programmer to easily control program flow in ways that would be difficult to achieve in Assembly Language. Here's a suitable initialization function, but with the screen blanking function disabled until we build it.

```
void SevSeg_Init(void)
{
    PORTA |= 0b00000011;    //preset control lines high
    DDRA  |= 0b00000011;    //A0:1 outputs
    DDRB   = 0b11111111;    //all PORTB outputs

//  SevSeg_BlAll();
}
```

Notice the use of bitwise OR commands for working with PORTA. Again, the upper six bits of this register are not being used for the seven-segment display, so, if we are careful not to mess with them in our routines, they will be available for other applications if so needed. If

we had written an entire byte to PORTA and DDRA (i.e. PORTA = 0b00000011 and DDRA = 0b00000011), we would have messed up any other activity on the upper six bits.

This is a good time to create a project that you can use for testing the library you are working on. Follow the steps outlined previously, and create a project called something like "SevSegTest". The following screen capture shows a sample "main.c" file, but it also shows what needs to be done to include the library you're working on. (You won't be able to run this code yet, as you will need to create SevSeg_Char first. At least try your initialization routine, which, without the blanking function, will probably display garbage.)



To include the library you're building you need to do three things:

1. In the browser to the left of the screen, search for and add the library's ".c" file under "Sources".
2. In the browser, add the library's ".h" file under "Includes".
3. In the main file, put in the #include statement as shown.

In the main file, notice that a counter variable has been declared and initialized in the "Variables" section.

SevSeg_Init() is called in the "Initializations" section, outside of the main loop for the program. You only want to initialize the ports once – not each time through the loop.

Since this entire code was only intended to be run through once, there's a HALT command to stop execution at the end. This is not usually a useful command in a microcontroller program, but helps us with testing or troubleshooting functions as we program.

It's up to you now to build the rest of the functions in this library. For each item, make sure you adhere to the prototype in the .h file. Here's a sneak peek at a suitable function for blanking a single character, passed as an address between 0 and 7.

```
//requires a digit address (0 to 7) to blank
void SevSeg_BlChar(unsigned char cDigit)
{
  cDigit &= 0b00000111; //clean up in case of error
  cDigit |= 0b01111000; //single digit, (don't care), no decode, no SD,
  PORTB = cDigit;
  PORTA  &= 0b11111110; //Mode stays high, strobe /Write
  PORTA  |= 0b00000011; //back to resting state - Mode and /Write high

    RTB   = 0b10000000; //turn off all segments and dp
    TA  &= 0b11111100; //Mode low for data, strobe /Write
        |= 0b00000011; //back to resting state - Mode and /Write high
```

Once you've created the above function for blanking a single character, you can use it to blank all eight digits. A good way to do this, using the ANSI C way of thinking, is shown below.

```
//blanks the display using the SevSeg_BlChar routine in an 8-cycle loop
void SevSeg_BlAll(void)
{
    unsigned char cCount;
    for(cCount=0;cCount<8;cCount++)
    {
        SevSeg_BlChar(cCount);
    }
}
```

Go back to your initialization function and enable the line that calls SevSeg_BlAll(). You should now be able to display a completely blank set of eight digits on the seven segment display. (I know, I know, it's pretty exciting to you, but the average person on the street won't understand, so don't rush out looking for someone to show this to!)

Once you've created a function for displaying a single digit at a specific location, you should be able to run the code shown on the previous page. Then you should be able to finish off the rest of the items required for this library, as shown in the header file.

Hopefully, you can see how what you've learned in your C# courses transfers to writing code in ANSI C, which was the original language from which all the various C-family derivatives grew. With a basic knowledge of the operation of the 9S12XDP512, a chance to work at the machine level using S12XCPU Assembly Language, and a fair bit of experience programming in C#, you should soon be able to make your microcontroller development kit carry out some fairly sophisticated activities.

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 (actually binary) coding to represent decimal values. It's important to remember that BCD is a code, not a real number system. It's a way to use hexadecimal values to represent decimal numbers. Real math must be done by your microcontroller using hexadecimal ("real numbers"). BCD is only for display purposes.

In BCD, the upper six hexadecimal values (ABCDEF) cannot be used, since they're not a part of the decimal number system. Instead, after 0123456789, the sequence must roll over to 10. The microprocessor will consider this to be 10_{16} because it only does hexadecimal. But it looks like 10_{10} to the rest of us, and, properly used, would be the BCD representation of 10_{10}. For clarity, we'll use the notation 10_{BCD}.

BCD's only purpose is to display values in a form humans are comfortable with. Just because they could, the designers of microprocessors made by Freescale have included the DAA (Decimal Adjust Accumulator A) command that allows you to do simple addition of BCD values. 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 – humour them if necessary.

Converting Hexadecimal Values to BCD

In a previous course, you learned the "Division with Remainders" method of converting from numbers of any base to decimal. This was also reviewed at the end of Topic 2. Division with Remainders involves dividing the number repeatedly by 10 (i.e. A_{16}), each time concatenating the remainders together, starting at the right and moving left. Here's an example:

Convert $123F_{16}$ to decimal.

```
   _0, R:4
  A)4, R:6
 A)2E, R:7
A)1D3, R:1
A)123F
```

Final result: $123F_{16} = 4671_{10}$, or for human display, 4671_{BCD}

If we used a microcontroller to hold or display this value, it would hold it as 4671_{16}, which is certainly not 4671_{10}, nor is it intended to be thought of as 4671_{16}. That's why we use the notation 4671_{BCD}, and why we only use this to represent the value as a coded representation that humans are comfortable with.

Also in your previous course, you developed and used a Hexadecimal-to-BCD converter using S12XCPU Assembly Language. Here's one version of that routine:

```
;*****************************************************************
;*      Hex2BCD5
;*
;*  4-nibble hex to 5-digit BCD converter
;*
;*  Hexadecimal value arrives in D.  BCD returned in X and D
;*  with the most significant digit in X
;*
;*  Registers affected:  X and D only
;*
;*****************************************************************

Hex2BCD5:
        PSHY                    ;Save contents of Y for later
        LDY     #5              ;Downcounter for a 5-digit loop
Hex2BCD5Loop:
        LDX     #10             ;Division with remainders method:  /10
        IDIV                    ;Answer is in X, remainder in D (i.e. B)
        PSHB                    ;Store remainders from "right to left"
        TFR     X,D             ;Put answer in D for next division
        DBNE    Y,Hex2BCD5Loop  ;Decrement the counter
                                ;Not at the end?  Loop again

        CLRB                    ;now have 5 digits on the stack:  need 6...
        PSHB                    ;so put a 0 on the stack

        PULX                    ;pull first two digits (0 and 10,000's)

        PULA                    ;pull 1000's off stack...
        LSLA                    ;shift it into the high nibble...
        LSLA
        LSLA
        LSLA
        ADDA    0,SP            ;...and add the 100's, so 1000's and 100's are in A

        INS                     ;move up to beginning of 10's
        PULB                    ;pull 10's off the stack...
        LSLB                    ;shift it into the high nibble...
        LSLB
        LSLB
        LSLB
        ADDB    0,SP            ;...and add the 1's, so 10's and 1's are in B

        INS                     ;increment the stack so it's ready for a pull

        PULY                    ;restore original value of Y

        RTS                     ;and go back to the main program with answer in X and D
```


Notice that this routine does the entire hexadecimal to BCD routine in the six lines of the "Hex2BCD5Loop": the rest of the subroutine is concerned with compressing the resulting BCD representation into the D Accumulator, with the fifth digit in the X register.

Using ANSI C, all of the details of packing and returning the coded characters can be handled by the cross-compiler. Also, since we're no longer directly concerned with the sizes of the accumulators, we can change our parameters.

In the Assembly Language version, we chose to return a five-character value because the largest number we could send, in a sixteen-bit register, was $FFFF_{16}$, or $65{,}535_{BCD}$, which is five characters. Of course, that's difficult to display on the seven-segment displays we've got, at least in the way they're configured. On your own, you could build a board that has the displays side-by-side, in which case displaying five digits would make sense.

Now that we're no longer directly tied to the size of the accumulators, we've got some decisions to make:

- The way our board is configured, it probably doesn't make sense for us to work with numbers larger than 9999_{BCD}, or $207F_{16}$, if the target is the seven-segment display.
- If we want a generally-useful hexadecimal-to-BCD converter, we could choose to work with a "Long" data type, in which case we could return eight characters and raise the size of the number we're working with to $99{,}999{,}999_{BCD}$, or $5F5E0FF_{16}$. Handling all the digits returned by the function would then be up to the programmer.
- If the target is some device that works in ASCII, such as the LCD display, a computer acting as a "dumb terminal", or a Raspberry Pi, you might want to display your values as floating-point numbers converted to ASCII strings, formatted using the "sprintf" function available in the ANSI C "stdio.h" library.

For simplicity, let's work first with simple 4-digit converters suited to use with the seven-segment display, which is looking for actual numbers (i.e. 0 to F in hexadecimal or 0 to 9 in BCD), not ASCII (which would be 0x30 to 0x39 for both hexadecimal and BCD, and 0x41 to 0x46 for the rest of the hexadecimal numbers).

You are going to need a library for various miscellaneous functions. Here's a header file for this library, called "Misc_Lib.h", and it shows you the functions you will eventually add to the source-code library. It's best that you comment out the prototypes that you haven't developed code for, and enable them when you're ready to use them.

Misc_Lib.h

```
//Miscellaneous generally-useful routines
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//September 2016

//Binary-Coded Decimal conversion routines

unsigned int HexToBCD(unsigned int);  //integer math; result is BCD - not converted to ASCII; make it 4-digits for sev-seg display
unsigned int BCDToHex(unsigned int);  //integer math; requires BCD - not ASCII equivalent; make it 4-digits to complement HexToBCD


//ASCII-Code handling routines

unsigned char ToUpper(unsigned char);
unsigned char ToLower(unsigned char);
unsigned char HexToASCII(unsigned char);  //single character converter
unsigned char ASCIIToHex(unsigned char);  //single character converter


//9S12X simple timer routines

void TimInit125ns(void);
void TimInit8us(void);
void Sleep_ms(unsigned int);      //requires TimInit8us setup; blocking delay
```

Notice that the routines come in three groups: BCD, ASCII, and Timer. For now, we'll just do the simple BCD-related functions.

HexToBCD

In a source-code file named "Misc_Lib.c", you will want to create a function that does the Division with Remainders routine we used in the S12XCPU Assembly Language routine previously. The code snippet below shows the heart of this routine, done using division and modulus operations. Recall that the modulus operation (%) provides the remainder of an integer division, whereas the division operation provides the integer result with no rounding.

Notice that there are three local variables: *iBCDOut*, *cCount*, and *iPow*. Declare these and initialize them within the function, but ahead of any active code.

Prior to this code snippet, *iPow* was initialized to 1 and *iBCDOut* was initialized to 0. So, first time through, the remainder is multiplied by 1. The second time through, the remainder is multiplied by 16 and added to the previous remainder; in other words, it is put in the next most significant nibble of the result. The next time through, the remainder is multiplied by 256 and added to the result, and the last time through it is multiplied by 4096 and added to the result. Consequently, the BCD characters end up in the required four nibbles of the final result, starting with the least significant character and ending with the most significant character.

```
for (cCount=0;cCount<4;cCount++)     //Four digits in a loop
{
    iBCDOut+=(iHexIn%10)*iPow;       //Division-remainder conversion
    iHexIn/=10;
    iPow*=16;                        //16^cCount
}
```

In your function, you should also provide some sort of error trapping. If the number sent to the function is greater than $9{,}999_{10}$, it's probably best to return 0 rather than some gibbled half-BCD/half-hex result.

BCDToHex

As with HexToBCD, we'll create a simple routine to convert integers up to $9{,}999_{BCD}$ to their equivalent true number values in hexadecimal so that the microcontroller can use them for numeric calculations. In your previous course, you worked with an S12XCPU version of this routine, as shown below. This routine is pretty complex, given that we needed to manipulate the digits individually, and, to avoid variables, we used the stack for storage. One item we won't be able to duplicate easily in C is indicating a valid value using CARRY.

```
;*******************************************************************
;*      BCD2Hex4
;*
;*  4-digit BCD to hex converter with error checking: not BCD = 0000 and Carry
;*
;*  BCD value arrives in D,  hex returned in D
;*
;*  Registers affected:  D only
;*
;*******************************************************************

BCD2Hex4:
        PSHX
        PSHY

        TFR     D,X             ;for now, hold BCD in X

BCDCheck:
        ANDB    #%00001111      ;check for valid 1's digit
        CMPB    #$09
        BHI     ErrorBCDtoHex4

        ANDA    #%00001111      ;check for valid 100's digit
        CMPA    #$09
        BHI     ErrorBCDtoHex4

        TFR     X,D             ;get original BCD again

        ANDB    #%11110000      ;check for valid 10's digit
        CMPB    #$90
        BHI     ErrorBCDtoHex4

        ANDA    #%11110000      ;check for valid 1000's digit
        CMPA    #$90
        BHI     ErrorBCDtoHex4
        BRA     BCDConvert      ;if all are valid, continue on ...

ErrorBCDtoHex4:
        LDD     #0              ;if not, return zero ...
        SEC                     ;set Carry Flag as error indicator ...
        BRA     FinishBCDtoHex4 ;and leave

BCDConvert:
        TFR     X,D
        ANDB    #%00001111      ;select 1's digit
        PSHB                    ;put it on the stack ...
        CLRB                    ;and pad it with a zero for 16 bit add ...
        PSHB                    ;on the stack

        TFR     X,D             ;grab the original ...
        ISRB                    ;shift over the 10's digit ...
        ISRB
        ISRB
        ISRB
        PSHB                    ;put it on the stack ...
        CLRB                    ;and pad it with a zero for 16 bit add ...
        PSHB                    ;on the stack

        TFR     X,D             ;grab the original again ...
        ANDA    #%00001111      ;get the 100's digit ...
        PSHA                    ;put it on the stack ...
        CLRA                    ;and pad it with a zero for 16 bit add ...
        PSHA                    ;on the stack

        TFR     X,D             ;grab the original one last time ...
        ISRA                    ;shift over the 1000's digit ...
        ISRA
        ISRA
        ISRA
        TAB                     ;move it to the low byte of D ...
        CLRA                    ;blank the high byte of D to start with D = 1000's digit

        LDX     #3              ;FOR loop downcounter for 3 passes
                                ;algorithm:  hex=((((Ths*10)+Huns)*10)+Tens)*10+Ones
WeightedLoop:
        LDY     #10             ;going to multiply current value by 10 ...
        EMUL                    ;useful part of answer is in D
        ADDD    0,SP            ;add in the next digit
        INS                     ;move the stack pointer past the used digit
        INS
        DBNE    X,WeightedLoop  ;not the last digit?  Loop again

        CLC                     ;Carry Flag cleared for good answer

FinishBCDtoHex4:
        PULY
        PULX
        RTS
```

This routine doesn't need error trapping for the size of the number, because the hexadecimal value will always take the same or fewer nibbles. However, it does need a trap for invalid characters (A through F, since these are never valid in a BCD representation of a number). Since there's no particularly easy way, like checking the CARRY bit, to indicate an invalid BCD value, we'll simply return 0000 and leave it up to the programmer to decide what to do with that. (One method would be: every time a 0000 appears, check to see if what was sent was 0000; if it wasn't, there's been an error.)

Since the logic and mathematics behind this routine can be a bit complicated, a fully-functional version of the code is shown below. Before you simply copy this routine, make sure you understand how it works. Other instructors may ask you to use or develop slightly different code to perform the same task – being nice to them will be to your advantage!

```
//BCDToHex handles numbers up to 4 digits (9999), returns Hex as 2 bytes (int)
//does not require the math.h C library
unsigned int BCDToHex(unsigned int iBCDIn)
{
    unsigned int iHexOut=0;
    unsigned char cCount=0; //using an incremented counter instead of FOR to allow breakout
    unsigned char cDigit;
    unsigned int iPow=1;

    while (cCount<4)
    {
        cDigit=iBCDIn%16;   //locate right-most digit
        if (cDigit<10)      //not valid BCD (0-9)?  might as well quit!
        {
            iHexOut+=cDigit*iPow;   //multiply by correct power of 10 and add
            iBCDIn/=16;     //get ready for next digit
            cCount++;
            iPow*=10;       //10^cCount
        }else
        {
            iHexOut=0;      //error?  return zero
            cCount=4;       //... and break out of loop
        }
    }
    return iHexOut;
}
```

First of all, notice how much more compact the ANSI C version of this routine is than the S12XCPU version! To a great extent, that's because we can declare and use local variables, so we don't need to manipulate the stack. (Incidentally, the C compiler may ignore your declared variables and use the stack instead – you'll never know until you disassemble the code to see what it did or try to trace the variables, which won't be listed if the compiler chooses not to use them. Some of us old-guard programmers find that mildly disturbing.)

Notice that we use 16 in our division and modulus calculations. That's because the microprocessor only really works in binary (i.e. hexadecimal) values, so even if we picture a BCD value as a real number, the microprocessor thinks of it as hexadecimal. So, to correctly locate and identify the characters, we need to work with them in groups of four bits, or 16's, not 10's. For example, consider 1264_{BCD}: 0x1264 / 16 = 0x126 with a remainder of 4. By doing so, we identify the lowest digit, and preserve the upper three digits for the next stage of the calculation. If we tried 0x1264 / 10, we'd get 0x1D6 with a remainder of 8, which is no use to us at all.

Once we identify the characters, we multiply them by powers of ten to make them into real numbers, which we add together to get the final true number. In the above example, we'd get 0x4F0, which is – you guessed it – 1264_{10}.

Each incoming character is checked to see if it is valid for BCD. If any invalid character is encountered, we break out of the loop and return 0000.

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.

Detecting Switch Change of State

There are basically four ways to write a program to respond to switches.

- The first situation is one in which it doesn't matter if the switch condition is read thousands or millions of times – the output directly relates to the current condition of the switch at all times.

  For example, you could have a program that turns the RED LED on as long as the LEFT switch button is pressed. "while (LEFT()) RedOn;" is pseudo-code for this.

The other three are ways to make it so that your program will respond just once to each button press or change of switch condition.

- One way is to have the program branch away from the routine that's checking for the switch as soon as the switch change is detected, thereby ignoring the condition of the switch until it is needed again. This works in situations like a menu-driven application, where selecting an item from a menu sends the microprocessor off on a particular task that doesn't check the switch again.

```
if((PT1AD1&0b00010000)!=0)  //UP: redirects to another routine if pressed
{
    UpHandler();
}
```

- A second way is what's called a *blocking routine*, where the program is held up until the switch condition changes back to its original condition. For example, the switch may normally be open. When it is closed, the program executes the desired action, then enters a loop, waiting for the switch to be released before it continues on to other commands. This blocking action may or may not be an issue. If you have other things that the program should be doing, holding it up waiting for a switch to be released is a bad thing; but if your program has nothing better to do than wait for the switch to be released, blocking isn't a problem.

```
if(PT1AD1&0b00001000)          //LEFT switch pressed?
  {
    SevSeg_Top4(++iCount);
    while(PT1AD1&0b00001000); //wait for LEFT release: blocks program
  }
```

  A variant of this which is sometimes useful is to wait for the switch to be released before executing the code. For most applications, this feels odd, because the action doesn't happen when you press the button – only when you release it. You're familiar with one application of this: touch screen item selection. With a touch screen, you can put your finger on an icon or control on the screen, but it doesn't respond until you lift your finger. This allows you to change your mind – if you decide you don't want to do what you've just touched, you can move your finger away before lifting, and the originally-selected action doesn't happen. This would not be a good thing to do with an emergency shutoff switch, though – you want the

equipment to stop as soon as you press the switch, not when your unconscious body finally falls away, releasing the switch!

```
if(PT1AD1&0b00000010)         //RIGHT switch pressed?
  {
    while(PT1AD1&0b00000010); //wait for RIGHT release: blocks program
    SevSeg_Top4(++iCount);
  }
```

Remember that either of these will "block", or hold up the processor, which may not be acceptable. Choose wisely!

- The non-blocking way to handle this is to use memory (i.e. a variable) to keep track of the previous condition of the switch. This technique is the best for continuous loops that need to monitor a switch or a set of switches continuously, such as in a control system or in something like a keypad or keyboard entry system for a calculator or computer. Here's a typical sequence:
  - With the switch open, the variable is cleared to indicate that the switch has not been pressed in the recent past.
  - If at some point the switch is closed, the program compares the current condition to the previous condition, detects a difference, records the new condition by setting the variable, and provides an indication to the main program that the switch condition has changed.
  - The next time through, if the switch is still closed, the current condition will be the same as the previous condition, so the routine will report no change, and therefore the program can ignore the switch.
  - Once the switch is released, the difference will be detected, the new condition will be stored (i.e. the variable will be cleared), and the main program will be notified that the switch has been released. The program can be set up either to respond to this change or to ignore it (which is probably the most likely situation).
  - Next time through, the variable and the condition of the switch will be the same (both cleared), so no change will be reported to the main program.

```
/*****************************************************************
*       Variables
*****************************************************************/

char cSwNew;
char cSwState=0;

/*****************************************************************
*       Main Program Code
*****************************************************************/

        cSwNew=PT1AD1&0b00000100;       //DOWN:  read the current condition of just DOWN
        if (cSwNew!=cSwState)           //only enters if a change in DOWN happens
        {
            cSwState=cSwNew;            //store new DOWN switch condition
            if ((cSwState&0b00000100)!=0)   //means change is a PRESS -- ignore RELEASE
            {
                SevSeg_Top4(++iCount);
            }
        }
```

Note: If you need to keep track of the states of a number of switches, read them all at once and, if there's a change, store all of them in the switch state variable. Your code for the above situation, then, would look like this:

```
/*****************************************************************
*       Main Program Code
*****************************************************************/

        cSwNew=PT1AD1&0b00011111;       //Read the current condition of all switches
        if (cSwNew!=cSwState)           //only enters if a change in the switches happens
        {
            cSwState=cSwNew;            //store new switch conditions
            if ((cSwState&0b00000100)!=0)   //means change is DOWN PRESSED -- ignore RELEASE
            {
                SevSeg_Top4(++iCount);
            }
        }
```

Other switches would then be checked using their own "if" statements inside the main "if" statement, just like the DOWN switch was checked.

Debouncing

If you've entered and tried the previous code snippets, you've probably noticed that the displayed count 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 pretty much 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 shouldn'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 probably won't affect the routines we're creating.

This is a good time to make a library of switch and LED-related functions to link into our program. The following is the contents of a header file that your instructor will probably make available to you in one form or another.

```
//Switches and LEDs
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//June 2015

void SwLED_Init(void);  //LEDs as outputs, Switches as inputs, dig in enabled
void LED_On(char);  //accepts R, G, Y, A (for all)
void LED_Off(char); //accepts R, G, Y, A (for all)
void LED_Tog(char); //accepts R, G, Y, A (for all), and toggles the condition of the LED(s) indicated
char Sw_Ck(void);   //returns debounced condition of all switches in a byte, LED values = 0
```

All but the last item in this list should be relatively easy for you to create. (Your instructor will likely ask you to complete these items.) The ***SwCk()*** routine, which returns a debounced version of the present conditions of all five of the switches, is provided below. **Add comments!**

SwCk() Debounced Switch Routine

```
char SwCk(void)
{
    char cSample1=1;
    char cSample2=0;

    while(cSample1!=cSample2)
    {
        cSample1=PT1AD1&0b00011111;
        asm LDX     #26667;         /* 26667 x 3 cycles x 125 ns = 10 ms */
        asm DBNE    X,*;
        cSample2=PT1AD1&0b00011111;
    }
    return cSample1;
}
```

In the "main" program, you will need a variable to keep track of the previous condition of the switches, as in the example on the previous page. In that example, reading PT1AD1 into *cSwNew* would be replaced by a call to *SwCk()*.

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 four bits in a nibble simultaneously. Later, 8-bit busses were introduced, transmitting whole bytes. Since then, microprocessors have gone 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 9S12X 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 0x0270 in the memory space of the 9S12X 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 sixteen bits wide, as in the 9S12X, we can access 2^{16} unique addresses, or 65,536. Obviously, your home computer's address bus is a lot bigger than sixteen 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 to it 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 – "enable" is usually LOW), and a R/W line.

This controller is quite flexible. The full details of its capabilities are listed in the data sheet, available in Moodle, with some key parts appearing as needed in this topic. Here are some of its capabilities:

- Can be used with a variety of LCD displays, ranging from 1 line x 8 characters to 2 lines x 40 characters or 4 lines x 20 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 on the micro side, and it doesn't know what LCD it's connected to on the device side, 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 for telling it everything it needs to know, and you can only do that by communicating with it through the 9S12X.

The HD44780-controlled LCD on the 9S12X Development Kit

If you check out the schematic for your development kit, you'll discover the following set of interconnections between the 9S12X 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:

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

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 9S12X 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 from $28 to $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.

In this course, you're expected to have and use the functions shown in the following header file:

```
//Hitachi 44780 initialization and commands
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//May 2015

void LCD_Init(void);            //8-bit, 2-line, 5x8 chars, disp on, curs on, blink off, inc curs mode, no shift, clear, home
void LCD_Ctrl(unsigned char);
unsigned char LCD_Busy(void);
void LCD_Char(unsigned char);
void LCD_Addr(unsigned char);   //raw LCD DDRAM address -- requires knowledge of device
void LCD_Pos(unsigned char,unsigned char);  //Row and Column, zero based; out of range values go to home location
void LCD_String(char *);        //requires a NULL-terminated string of ASCII characters in main program
```

Unfortunately, the _Init routine requires _Ctrl and _Busy, which complicates things. Your instructor may also ask you to develop functions to generate special characters later.

HD44780 Instructions

In the LCD instructions, the first bit that's set (HIGH) in the instruction byte determines the group of instructions to choose from. These instructions are found in the data sheet for the HD44780, for which a link has been provided in Moodle, and are shown below:

HITACHI **HD44780U**

Table 6 Instructions

Instruction	Code RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 µs
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 µs
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*

I/D = 1: Increment
I/D = 0: Decrement
S = 1: Accompanies display shift
S/C = 1: Display shift
S/C = 0: Cursor move
R/L = 1: Shift to the right
R/L = 0: Shift to the left
DL = 1: 8 bits, DL = 0: 4 bits
N = 1: 2 lines, N = 0: 1 line
F = 1: 5 × 10 dots, F = 0: 5 × 8 dots
BF = 1: Internally operating
BF = 0: Instructions acceptable

DDRAM: Display data RAM
CGRAM: Character generator RAM
ACG: CGRAM address
ADD: DDRAM address (corresponds to cursor address)
AC: Address counter used for both DD and CGRAM addresses

Execution time changes when frequency changes
Example:
When f_{cp} or f_{OSC} is 250 kHz,
$37\ \mu s \times \frac{270}{250} = 40\ \mu s$

Note: — indicates no effect.

* After execution of the CGRAM/DDRAM data write or read instruction, the RAM address counter is incremented or decremented by 1. The RAM address counter is updated after the busy flag turns off. In Figure 10, t_{ADD} is the time elapsed after the busy flag turns off until the address counter is updated.

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.

LCD Controller Initialization

Timing is critical in all communications with this controller, and particularly so in the initialization of the device. 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.

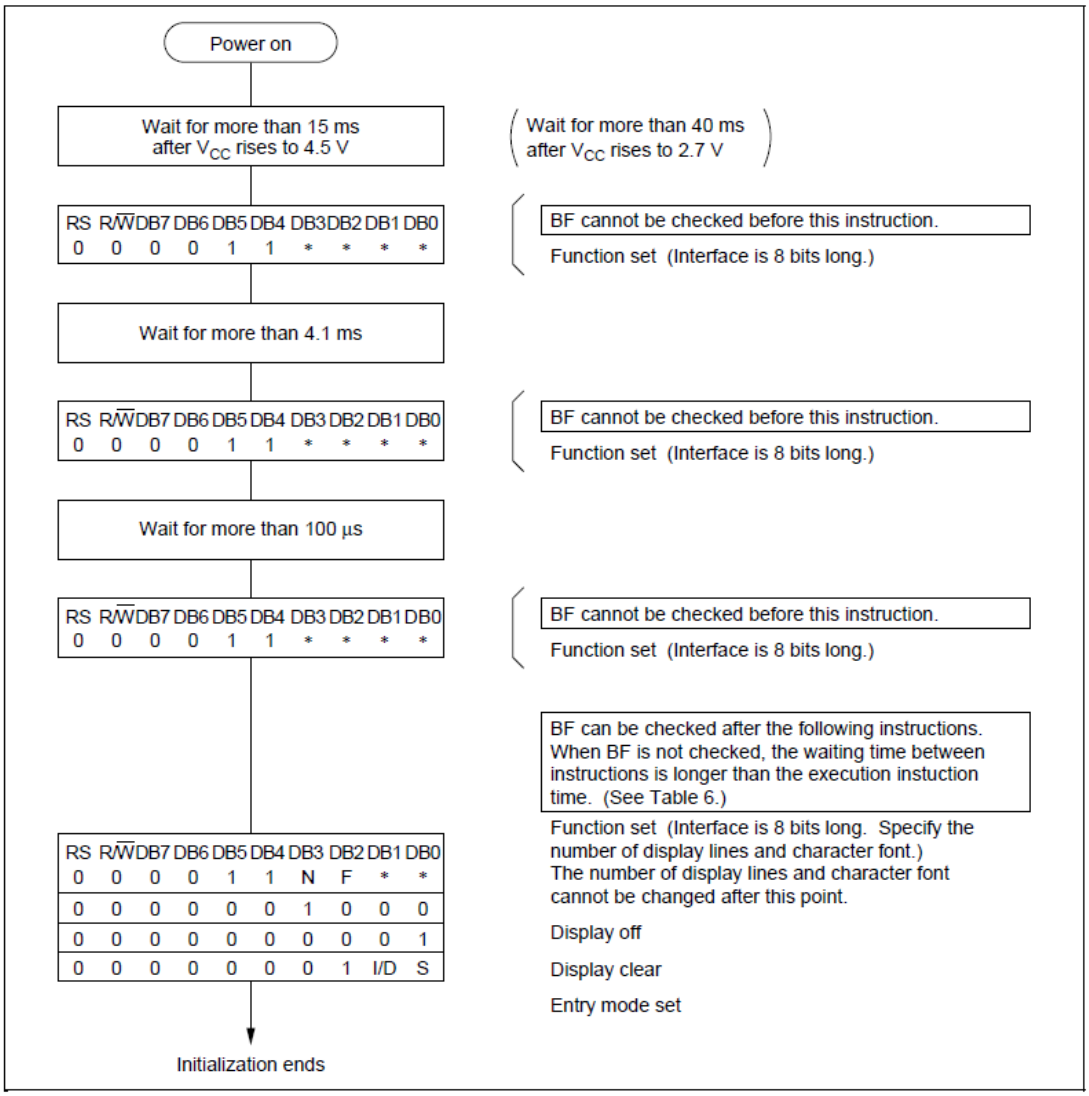


Figure 23 8-Bit Interface

HITACHI

45

LCD_Init

For this discussion, the initialization flowchart from the previous page will be used as a template for the code developed.

Before we can do anything at all, we need to set up our simple parallel bus interface between the microcontroller and the LCD controller. Checking back to the schematic for our board, we see that Port H (PTH) is being used as the eight-bit data bus and the lower three bits of Port K (PORTK) are being used as the control lines:

PORTK bits:
7 – x
6 – x
5 – x
4 – x
3 – x
2 – RS (register select: LOW for Control, HIGH for Data)
1 – R/W (HIGH for READ, LOW for WRITE)
0 – EN (chip enable: HIGH for Enable)

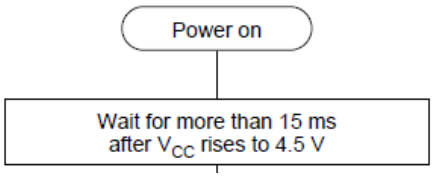
Most of the time, we will be writing to the LCD controller: we will write control bytes to it to tell it how we want it to look and respond; we will write data bytes to it, primarily providing it with the ASCII codes we want to display on the screen. So, it makes sense for us to set the default condition for the data bus, PTH, as outputs for all eight bits using the Port H Data Direction Register (DDRH).

Although it doesn't really matter what's on the bus when we enable it, good programming practice suggests we should write something innocuous to the bus, so clearing all eight bits before we change the pins to outputs is a good idea.

Since we're only using three of the eight bits in PORTK, we should leave the other five bits alone in case there's some other possible use for those bits. So, instead of writing an entire byte to the Port K Data Direction Register (DDRK), we'll OR the three bits we need with 1s to make them into outputs, while leaving the other five alone.

Before we do that, however, we should set the bits in PORTK to the condition we want them to be in when the port pins are enabled. The resting state that makes sense for us is to have all three controls lines LOW – RS set for control, R/W set to Write, and EN low so that the chip is not being addressed. The code below also shows the beginning of the LCD_Lib.c file that you will be building to match the header file shown previously.

```
//Hitachi 44780 initialization and commands
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//May 2015

#include <hidef.h>
#include "derivative.h"
#include "LCD_Lib.h"

void LCD_Init(void)
{
    PTH   =0b00000000;
    DDRH  =0b11111111;  //data bus as outputs for write
    PORTK&=0b11111000;  //preset RS low, R/W low, EN low
    DDRK |=0b00000111;  //activate three control lines
```



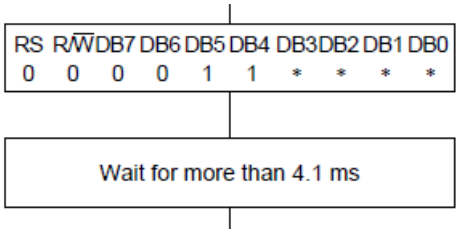
Since we don't know how long it might take for the power supply to reach 4.5 V, we'll make the wait time quite a bit longer than 15 ms. If we do two loops of a full 16-bit countdown using the three clock cycle Assembly instruction DBNE, we'll have 49.152 ms, which should be ample:

2 x 65,536 x 3 cycles x 125 ns/cycle = 49.152 ms

To ensure that we get full control of the microcontroller at this point, we need to do this part of the code in S12XCPU Assembly Language. This is done by putting the keyword "asm" in front of the Assembly code, which is now followed by a semicolon since it's in *C*.

```
asm LDD #0;          //need a 49.15 ms delay
asm DBNE D,*;        //24.576 ms delay
asm DBNE D,*;        //...twice
```

Next step in the flowchart:

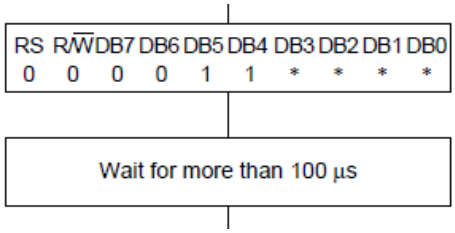


Note the position of the first HIGH in the byte they intend us to send: This is a "function" control byte. At this point, the controller will only respond to the contents of DB5, which is SET, meaning that we're going to be using an eight-bit bus. If you look at the flowchart, we'll be sending a "function" control byte four times in a row, with the other settings included the fourth time through. Since these are "don't care" for the first three times, it's ok for us to make them what we want them to be the last time through. So, we'll put the final version of the command on the bus, then just write it four times with the appropriate delays in between. Here's the first one (indicated in the flowchart snippet above). Note that we manipulate the control lines to do what we want, then return them to their resting state. This is sometimes called "strobing" the control lines, most particularly the Enable.

```
    PTH    =0b00111000;
/*                |||||_____(don't care)
                  ||||______(don't care)
                  |||_______font:  5x8 matrix (LOW)
                  ||________lines: 2 (HIGH)
                  |_________data:  8-bit (HIGH)
                            function set of commands
*/
    PORTK|=0b00000001;  //write a control byte
    PORTK&=0b11111000;  //resting state
/*                  |||_____EN  (Enable: HIGH to enable)
                    ||______R/W (LOW for WRITE)
                    |_______RS  (Register Select:  LOW for control)
*/
    asm LDD #11000;     //need 4.125 ms delay
    asm DBNE D,*;
```

Check the timing: 11,000 x 3 cycles x 125 ns/cycle = 4.125 ms

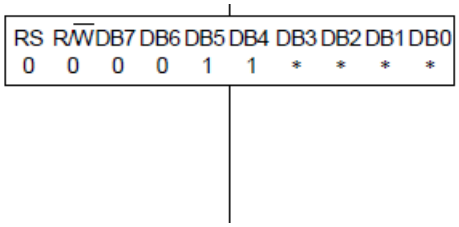
Now for the second attempt: Since the control byte is still sitting on the bus, all we need to do is strobe the control lines again, then wait the required time.




```
PORTK|=0b00000001;   //RS low, R/W low, EN high to write a control
PORTK&=0b11111000;   //resting state

asm LDD #267;        //need 100 us delay
asm DBNE D,*;
```

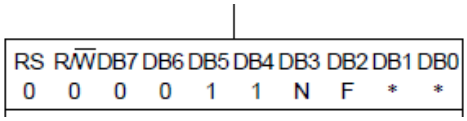
Check the timing: 267 x 3 cycles x 125 ns/cycle = 100 µs

Third time from the flowchart. Oddly, for this, no delay time is indicated. However, Our microcontroller is faster than the LCD controller, so we'll insert a delay, anyway. We've already got a 100 µs delay calculated, so we'll use it.




```
PORTK|=0b00000001;   //RS low, R/W low, EN high to write a control
PORTK&=0b11111000;   //resting state

asm LDD #267;        //need 100 us delay
asm DBNE D,*;
```

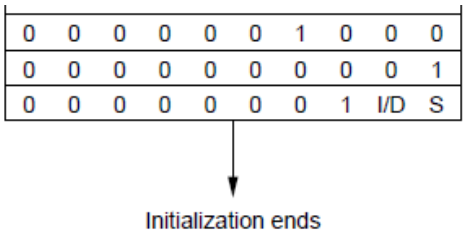
According to the notes, the HD44780 LCD controller should be working properly, and its "Busy Flag" should be available for further instructions. And, although it seems from experience that the HD44780 has been properly configured at this point, the flowchart says "do it again", so who are we to argue?



This time, we'll rely on the busy flag, and will use the *LCD_Ctrl()* command. Oh, wait a second, we haven't written that yet! We'll code in the function call, then come back to writing it later.

```
LCD_Ctrl(0b00111000);   //same as above, but using LCD_Ctrl (Busy is active)
```

Finally, we'll fine-tune the settings. The settings in the flowchart don't match the operation this author prefers, so you'll notice differences in the code that follows. For one, the flowchart turns the display off, which seems counterproductive after all that work!




```
    LCD_Ctrl(0b00001110);   //display controls
/*                  ||||_____Blink:   LOW for off
                    |||______Cursor:  HIGH for on
                    ||_______Display: HIGH for on
                    |________Display Control commands
*/
    LCD_Ctrl(0b00000001);   //clear display, home position

    LCD_Ctrl(0b00000110);   //mode controls
/*                  |||_____Shift:   LOW for no display shift
                    ||______Inc/Dec: HIGH for increment (to the left)
                    |_______Entry Mode commands
*/
}
```

LCD_Ctrl

The LCD_Ctrl routine passes an 8-bit control byte, which it sends out, once the "Busy" flag is cleared, using RS HIGH for control, R/W LOW for WRITE, and Enable strobed HIGH then LOW. That's a pretty simple routine, given what you've had to do in the LCD_Init routine. However, it relies on yet one more function, LCD_Busy, which we will also have to write.

```
void LCD_Ctrl(unsigned char cCommand)
{
    while (LCD_Busy()!=0);  //wait for the Busy Flag to be CLEARED
    PTH=cCommand;           //put control byte on the bus

    PORTK|=0b00000001;      //RS low, R/W low, EN high to write a control
    PORTK&=0b11111000;      //resting state
}
```

Make sure you understand how *LCD_Ctrl* works: you'll need to use a very similar technique for sending a character (data byte) to the LCD later. By the way, we could have used (while(!LCD_Busy()); to wait for the Busy Flag to be CLEARED.

We still need *LCD_Busy*, which checks to see if the controller is available for communication.

LCD_Busy

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, which is the most significant bit). Getting this information from the LCD controller can be done by executing a READ of the internal register, which becomes the Status Register when read from, instead of the Control Register we've been writing to.

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 non-zero return from *LCD_Busy*. 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.

```
unsigned char LCD_Busy(void)
{
    unsigned char cBusy;

    DDRH = 0b00000000;      //data bus as inputs for read

    PORTK|=0b00000011;
/*                  |||_________EN  (Enable:  HIGH to enable)
                    ||__________R/W (HIGH for READ)
                    |___________RS  (Register Select:  LOW for Status)
*/
    PORTK&=0b11111000;      //resting state

    cBusy=PTH&0b10000000;   //Busy Flag is the MSB of the Status Register

    DDRH = 0b11111111;      //data bus returned to outputs for next write

    return cBusy;
}
```

As always, make sure you understand how this routine works before you start using it.

LCD_Char

Up to this point, all we've done is get the LCD controller set up and ready to work for us. The LCD is now sitting there doing nothing until we send it ASCII characters to display. *LCD_Char()* is the name of the function we'll create to carry out this task.

The only differences between *LCD_Char()* and *LCD_Ctrl()* are the type of data sent and the internal register it's sent to.

- The data sent will be a single ASCII character. So, if you want to send a number, you'll have to do a Hex to ASCII conversion first.
- The target in the LCD controller is the Data Register, not the Control Register. That means that you will need to SET RS when you strobe the control lines.

That should be enough information for you to create and test *LCD_Char()*. If you need further help, your instructor can provide it.

LCD_String

Quite often, you'll want to send multiple characters to the LCD (or to other peripherals or equipment that's looking for ASCII characters). The best way to send longer strings of characters is by using a ***null-terminated string*** of ASCII characters. The *LCD_String* function you will be creating can handle strings of any length (up to the length of a row on the LCD display, or 20 characters), since it's not looking for a particular length, but is expecting the string to end with the *NULL* character (ASCII code 0). The routine transmits each character, then checks to see if the character was a *NULL*. If it is a *NULL*, program execution exits the function.

You will be using *LCD_Char()* as the working block of *LCD_String(*)*, a routine that sends a null-terminated string, from a memory location specified in your program, to the LCD.

In ANSI C, the starting point of a string or array of bytes is referenced using *pointers*. The following screen clip of the author's *LCD_String* function shows how to get the contents of an address which is being pointed to using an asterisk (*). *cString* is the label of the string you want to transmit, and is actually initially the address of the first character in the string.

```
void LCD_String(char * cString)
{
  while(*cString!=0)      //watch for NULL terminator
    LCD_Char(*cString++); //send next character
}
```

Take some time to make sense of how the pointer is being used in the LCD_String() function. In this course, there won't be much need to learn more about pointers than you see in this example. If you need to know more, the Internet is an endless source of wisdom and hilarity regarding pointers, along with the dereferencing "*" operator and the "&" address operator used for pointer management.

LCD_Addr

This is intended to be a simple way to locate a particular position on the LCD display. All it does is to take the address provided, convert it into a DDRAM Address control byte, and send that command to the LCD controller. The cursor will move to that location, ready for you to send a character to that spot. Here's the code for this function:

```
void LCD_Addr(unsigned char cAddr)
{                           //assumes programmer understands raw addresses
    cAddr|=0b10000000;  //add command bit for "Set DDRAM Address"
    LCD_Ctrl(cAddr);
}
```

Compare back to the table of instructions for the HD44780 controller to see why the MSB needed to be changed to 1 before sending the address to the control register.

LCD_Pos

In order to use LCD_Addr effectively, the programmer needs to know what the valid addresses are and how they are arranged on the display. The following table appeared earlier in this document, but has been duplicated here for convenience:

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

LCD_Pos() is intended to simplify the process of moving to a particular location on the display. The basic idea is to pass two numbers to the function: a ROW and a COLUMN, and let the function generate the correct address to send to the controller using *LCD_Addr()*.

For consistency between classes, the instructors for this course have settled on zero-based row and column addressing, so the available rows are 0 through 3 and the available columns are 0 through 19. Part of the necessary code for this routine is shown below; you are expected to complete the function so that it works satisfactorily. This code demonstrates an application for the *switch->case* operation in *C*.

```
void LCD_Pos(unsigned char cRow, unsigned char cCol)
{                           //zero-based for both row and column
    if (cRow>3 || cCol>19)  //display invalid inputs at home location
    {
        LCD_Addr(0);
    }
    else
    {
        switch (cRow)
        {
            case 0:     //first row is 0 to 19
            LCD_Addr(cCol);
            break;

            case 1:     //second row is 64 to 83
            LCD_Addr(cCol+64);
            break;

            case 2:     //third row
            LCD_Addr(cCol+20);
```

At this point, you will have at your disposal all of the LCD functions deemed necessary by all of your instructors. Your instructor, however, may want you to generate a collection of other functions to enhance your use of the LCD on your board. Some of these may simply be *LCD_Ctrl()* calls that do things like turn the cursor on or off, blink on or off, clear the display, etc. without requiring you to constantly look up the instructions on the table.

Character Generation

One set of optional functions you can try out allow you to access the LCD's capacity for generating characters other than the ones in the ASCII set contained in its memory. If your instructor deems this extraneous, you can skip the next few pages.

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).

These user-defined characters will take the place of the first eight ASCII characters. 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. So, to access the characters you generate, simply reference ASCII characters 0x00 to 0x07.

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. Here it is from the data sheet's table of instructions:

Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs

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 CGRAM to start at for a given character. Only 6 bits are required, since only 64 bytes are need to represent the eight 5 x 8 characters – each row of pixels in a character 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, as shown on the next page:

Table 5 Relationship between CGRAM Addresses, Character Codes (DDRAM) and Character Patterns (CGRAM Data)

For 5 × 8 dot character patterns

Character Codes (DDRAM data)	CGRAM Address	Character Patterns (CGRAM data)	
7 6 5 4 3 2 1 0 (High … Low)	5 4 3 2 1 0 (High … Low)	7 6 5 4 3 2 1 0 (High … Low)	
0 0 0 0 * 0 0 0	0 0 0	0 0 0	* * * 1 1 1 1 0
		0 0 1	1 0 0 0 1
		0 1 0	1 0 0 0 1
		0 1 1	1 1 1 1 0
		1 0 0	1 0 1 0 0
		1 0 1	1 0 0 1 0
		1 1 0	1 0 0 0 1
		1 1 1	* * * 0 0 0 0 0 — Cursor position
0 0 0 0 * 0 0 1	0 0 1	0 0 0	* * * 1 0 0 0 1
		0 0 1	0 1 0 1 0
		0 1 0	1 1 1 1 1
		0 1 1	0 0 1 0 0
		1 0 0	1 1 1 1 1
		1 0 1	0 0 1 0 0
		1 1 0	0 0 1 0 0
		1 1 1	* * * 0 0 0 0 0 — Cursor position
		0 0 0	* * *
		0 0 1	
0 0 0 0 * 1 1 1	1 1 1	1 0 0	
		1 0 1	
		1 1 0	
		1 1 1	* * *

Character pattern (1)

Character pattern (2)

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 re-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. The following two functions can be used to create a single character at a given location or to create all eight available characters from a table of 64 row-definition bytes defining all eight characters.

- *LCD_CharGen ()* is a routine that builds a single custom character for the ASCII code passed to it as a parameter (0x00 to 0x07), with the character definition pattern beginning at a location pointed to as a parameter.
- *LCD_CharGen8()* is a routine that builds eight custom characters, with their definition patterns beginning at a location pointed to as a parameter. (If you don't need all eight characters, just fill the unused row bytes with nulls to produce blank characters.) This function doesn't need an ASCII code, since it fills 0x00 to 0x07.

LCD_CharGen Example

```
/*****************************************************************************
*       Variables
*****************************************************************************/

char cNameString[14] = "P Ross Taylor";

char ASCII_1[8] =
{
    0b00000100,
    0b00001110,
    0b00011011,
    0b00010001,
    0b00011011,
    0b00011011,
    0b00001110,
    0b00000100
};

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


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

/*****************************************************************************
*       Initializations
*****************************************************************************/

    LCD_Init();
    LCD_Ctrl(0b00001100);   //Cursor off

    LCD_CharGen(1,ASCII_1); //Generate a single ASCII Char as 0x01


    for (;;)        //endless program loop
    {
/*****************************************************************************
*       Main Program Code
*****************************************************************************/

        LCD_Pos(2,5);               //Row 3, Column 6
        LCD_String(cNameString);    //Display name

        LCD_Pos(0,10);              //Row 1, Column 11
        LCD_Char(1);                //Display custom character


        HALT;                       //Halt program in endless loop

    }
}
```

Notice that *LCD_CharGen()* is run only once, in the initializations. That's where ASCII character 0x01 is created. To use this character, we simply display ASCII character 0x01 by using our *LCD_Char()* function. The results are seen below. Notice how the pattern in the first row relates to the pixel map shown in the *Variables* space above.

LCD_CharGen8 Example

Here's the author's version of *LCD_CharGen8()*. If you're an enterprising type, you will probably realize that you don't need to modify this much to make *LCD_CharGen()*. The eight-character version writes 64 bytes into CGRAM, starting with address 0x00 and going to address 0x3F.

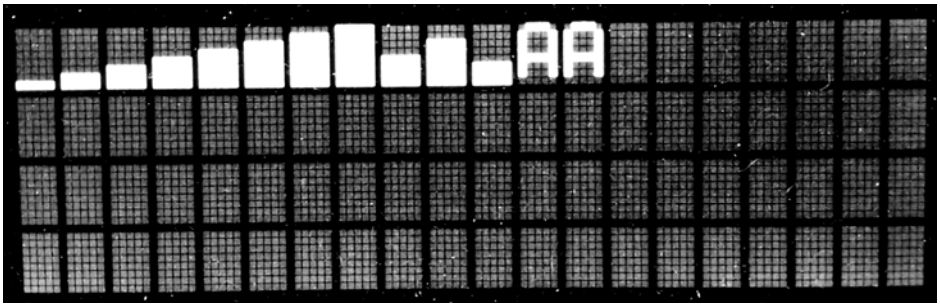
```
void LCD_CharGen8(char *cDefs)
{
    unsigned char cLine;

    LCD_Ctrl(0b01000000);   //start of ASCII(0)

    for(cLine=0;cLine<64;cLine++)
    {
        LCD_Char(*cDefs++);
    }

    LCD_Ctrl(0b10000000);   //back to DDRAM, home location
}
```

To keep the entire useful code together, the following images have been shrunk probably beyond your ability to read them on paper. However, you can zoom in on the electronic version to see more detail.

```
/*******************************
*       Variables
*******************************/
unsigned char cCount;

char ASCII_0to7[64] =
{       //ASCII char 0
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
            //ASCII char 1
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
            //ASCII char 2
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
    0b00011111,
            //ASCII char 3
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
            //ASCII char 4
    0b00000000,
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
            //ASCII char 5
    0b00000000,
    0b00000000,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
            //ASCII char 6
    0b00000000,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
            //ASCII char 7
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111,
    0b00011111
};
```

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

/*********************************************************************
*       Initializations
*********************************************************************/

    LCD_Init();
    LCD_Ctrl(0b00001100);   //Cursor off

    LCD_CharGen8(ASCII_0to7);   //Generate all eight custom characters

    for (;;)        //endless program loop
    {
/*********************************************************************
*       Main Program Code
*********************************************************************/

        LCD_Pos(0,0);               //Home position

        for(cCount=0;cCount<8;cCount++)
        {
            LCD_Char(cCount);       //Display chars in order
        }
        LCD_Char(3);                //Display three randomly chosen
        LCD_Char(5);
        LCD_Char(2);
        LCD_Char(0x41);             //Same procedure for normal ASCII
        LCD_Char('A');              //Alternate way to display ASCII

        HALT;                       //Halt program in endless loop

    }
}
```

Notice again that *LCD_CharGen8()* is only run once in the initializations, and creates all eight custom ASCII characters: 0x00 through 0x07. Once created, these characters can be displayed in the same way as any of the other pre-defined ASCII characters.



ASCII Code Manipulation

The LCD display and the Serial Communication Interface (still to come) work primarily with ASCII values. The following table shows the standard 7-bit ASCII codes.

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. Eight-bit ASCII includes another 128 characters ($80 to $FF) which are called "extended ASCII" and are not standardized. Trying them out will produce different results on different displays and terminals, so you're welcome to play around with them if you have a fairly high tolerance for frustration. For the Hitachi 44780 display, the following table from the datasheet shows the characters that can be displayed:

HD44780U

Table 4 Correspondence between Character Codes and Character Patterns (ROM Code: A00)

Upper 4 Bits / Lower 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	ß	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	q	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	×	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		−	=	M	]	m	}			ュ	ス	ヘ	ン	₺	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	°	ö	■

Note: The user can specify any pattern for character-generator RAM.

The note at the bottom reiterates the fact that you can create your own characters for ASCII codes 0 through 7. Apparently, they can also be accessed using ASCII codes 8 through F, but they would be the same characters as 0 through 7.

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.) You should discover that there is only one bit different. This makes manipulation of uppercase and lowercase values pretty simple. The only thing you need to watch out for is that you only want to change this bit if the value is a valid alphabet character – otherwise, you could be seriously messing up a number or punctuation mark!

In your Misc_Lib.h header file, you'll notice that prototypes have been included for two routines designed to handle changing the case of an ASCII character:

- ToUpper()
- ToLower()

Your instructor will want you to have these routines completed and working, not only for the LCD, but also for subsequent use with dumb terminals attached to an SCI comm port.

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. It's a code!

Converting regular digits (0123456789) to ASCII is easy – just add 0x30 to the digit or OR the digit with %00110000.

Converting the hexadecimal values ABCDEF to ASCII is similar, but with a different offset. For these, you need to add 0x37.

In your *Misc_Lib*, you will want to write HexToASCII() so that you can convert individual numeric digits to ASCII code in order to send the results to the LCD or other equipment that displays ASCII codes. Make sure that your function only converts true numerals (0 – 9 and A – F or a – f), and that it can't be broken if a two-nibble or two-digit value is sent to it. Incidentally, you can simplify the handling of A – F and a – f by using either ToUpper() or ToLower() inside your HexToASCII() routine.

Also in your *Misc_Lib*, you will want to write ASCIIToHex() that takes the codes for valid ASCII codes (0x30 – 0x39 and 0x40 – 0x45) and converts them to real numbers (0 – 9 and A – F). Again, don't' mess with any values outside of these ranges.

Your ASCIIToHex() function should return *0* if non-valid (i.e. non-numeric) keys are pressed. It will then be up to your handling of the returned value in the main() program to determine what to do with a returned 0. You may choose to work with the returned zero, or you may want to set up a trapping routine that determines when zero represents an invalid response and when it actually means zero.

The Serial Communications Interface

Your 9S12X 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 9S12X (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 9S12X SCI modules are able to send 8-bit or 9-bit data payloads. This provides a fair bit of flexibility:

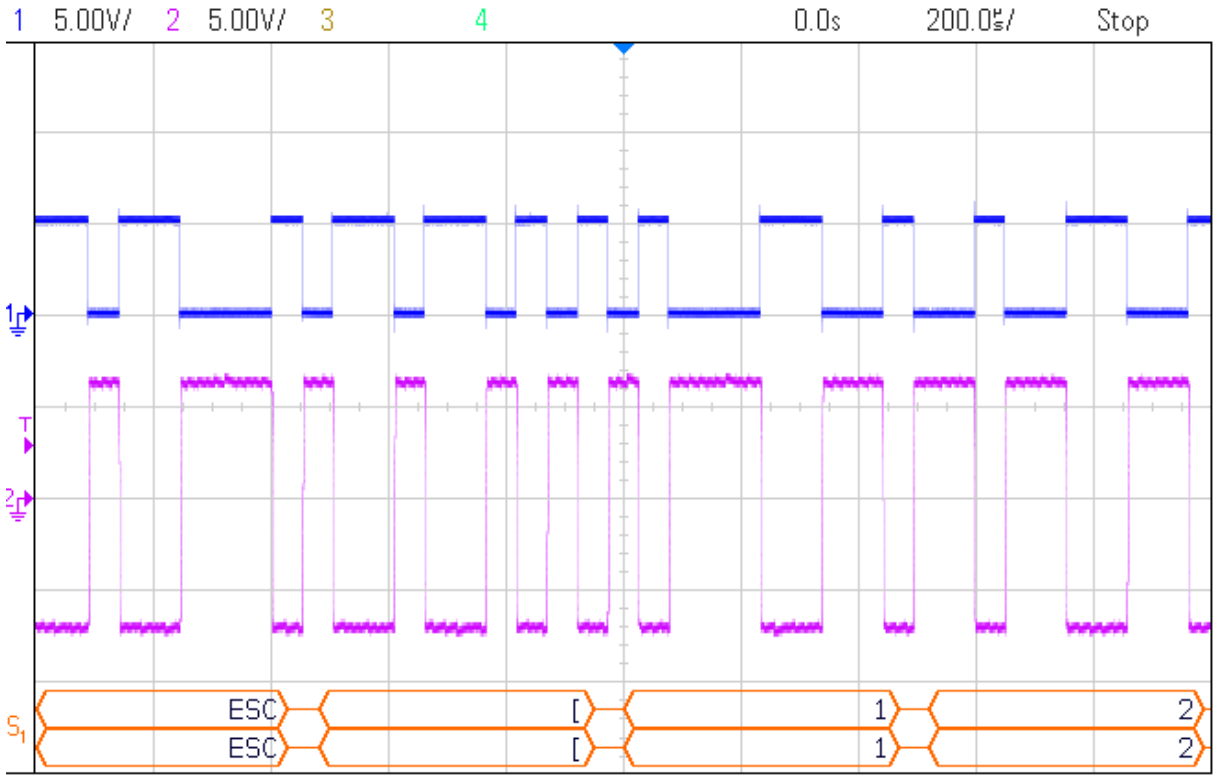
- 9-bit mode provides for 9 actual data bits (very rarely used) or 8 data bits and a parity bit for error checking
- 8-bit mode provides for 8 actual data bits or 7 data bits and a parity bit

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.

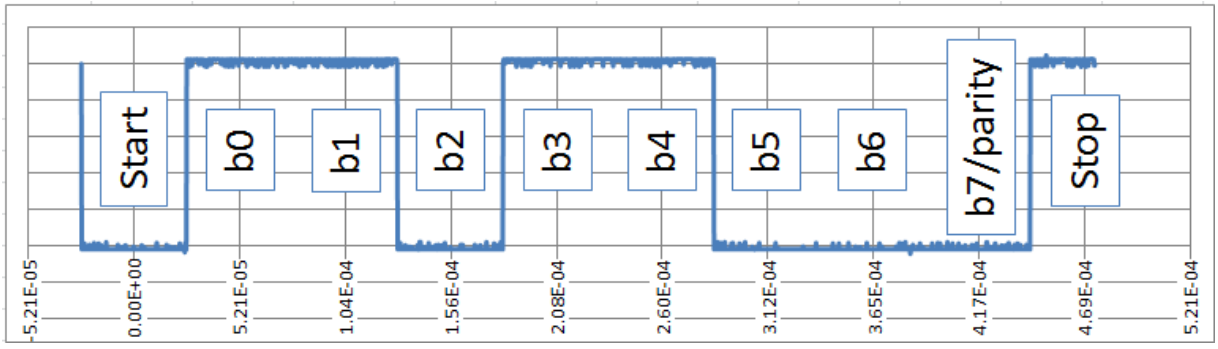
Also, we will be working with the simplest electrical connection possible between our board and the terminal – three wires only: Ground, Transmit Data, and Receive Data. This means that we will not be using the bundle of handshaking wires made available in the RS-232C standard.

Note: 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 9S12X'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 9S12X 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

Initializing the Serial Communications Interface

To activate an SCI module on the 9S12X, you typically need to configure just three registers. A full description of the activities of these registers is found in chapter 11 of the 9S12X "Data Sheet". Look these up if you want more a more detailed understanding of the operation of these registers than is provided below.

The first register, SCIBD, is a sixteen-bit register (SCIBDH/SCIBDL) that controls the baud rate for the module, and requires a 13-bit value as a clock divisor.

11.3.2.1 SCI Baud Rate Registers (SCIBDH, SCIBDL)

	7	6	5	4	3	2	1	0
R W	IREN	TNP1	TNP0	SBR12	SBR11	SBR10	SBR9	SBR8
Reset	0	0	0	0	0	0	0	0

Figure 11-3. SCI Baud Rate Register (SCIBDH)

	7	6	5	4	3	2	1	0
R W	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
Reset	0	0	0	0	0	0	0	0

Figure 11-4. SCI Baud Rate Register (SCIBDL)

Read: Anytime, if AMAP = 0. If only SCIBDH is written to, a read will not return the correct data until SCIBDL is written to as well, following a write to SCIBDH.

Write: Anytime, if AMAP = 0.

NOTE

Those two registers are only visible in the memory map if AMAP = 0 (reset condition).

The SCI baud rate register is used by to determine the baud rate of the SCI, and to control the infrared modulation/demodulation submodule.

Table 11-1. SCIBDH and SCIBDL Field Descriptions

Field	Description
7 IREN	**Infrared Enable Bit** — This bit enables/disables the infrared modulation/demodulation submodule. 0 IR disabled 1 IR enabled
6:5 TNP[1:0]	**Transmitter Narrow Pulse Bits** — These bits enable whether the SCI transmits a 1/16, 3/16, 1/32 or 1/4 narrow pulse. See Table 11-2.
4:0 7:0 SBR[12:0]	**SCI Baud Rate Bits** — The baud rate for the SCI is determined by the bits in this register. The baud rate is calculated two different ways depending on the state of the IREN bit. The formulas for calculating the baud rate are: When IREN = 0 then, SCI baud rate = SCI bus clock / (16 x SBR[12:0]) When IREN = 1 then, SCI baud rate = SCI bus clock / (32 x SBR[12:1]) **Note:** The baud rate generator is disabled after reset and not started until the TE bit or the RE bit is set for the first time. The baud rate generator is disabled when (SBR[12:0] = 0 and IREN = 0) or (SBR[12:1] = 0 and IREN = 1). **Note:** Writing to SCIBDH has no effect without writing to SCIBDL, because writing to SCIBDH puts the data in a temporary location until SCIBDL is written to.

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 actual baud rate is the bus frequency (8 MHz on your board) divided by 16 divided by the 13-bit value provided in SCIBDH:SCIBDL. By the way, you can simply write a sixteen-bit value to SCIBD. Since there are a number of SCI ports available, the prototype file distinguishes between them by inserting a number into the register name. The one we want is SCI0BD. This port is connected to the 9-pin RS-232 connector on your 9S12X board.

If you want to access the other SCI ports, SCI1 is connected to the infrared hardware on your 9S12X board, and the other ports are available at the break-out headers – just look up the appropriate pin numbers for TX and RX for the channel you're interested in. There are six SCI ports available in total!

If you decide to use the infrared channel for wireless point-of-sight communication, you'll also need to manage the upper three bits of SCI1BDH. For standard wired communication, these can be all be cleared to zero.

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 SCI0BD?

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 modules are 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.

The next configuration register to consider is the SCI Control Register 1, shown on the following page.

11.3.2.2 SCI Control Register 1 (SCICR1)

	7	6	5	4	3	2	1	0
R W	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT
Reset	0	0	0	0	0	0	0	0

Figure 11-5. SCI Control Register 1 (SCICR1)

Read: Anytime, if AMAP = 0.

Write: Anytime, if AMAP = 0.

NOTE

This register is only visible in the memory map if AMAP = 0 (reset condition).

Table 11-3. SCICR1 Field Descriptions

Field	Description
7 LOOPS	**Loop Select Bit** — LOOPS enables loop operation. In loop operation, the RXD pin is disconnected from the SCI and the transmitter output is internally connected to the receiver input. Both the transmitter and the receiver must be enabled to use the loop function. 0 Normal operation enabled 1 Loop operation enabled The receiver input is determined by the RSRC bit.
6 SCISWAI	**SCI Stop in Wait Mode Bit** — SCISWAI disables the SCI in wait mode. 0 SCI enabled in wait mode 1 SCI disabled in wait mode
5 RSRC	**Receiver Source Bit** — When LOOPS = 1, the RSRC bit determines the source for the receiver shift register input. See Table 11-4. 0 Receiver input internally connected to transmitter output 1 Receiver input connected externally to transmitter
4 M	**Data Format Mode Bit** — MODE determines whether data characters are eight or nine bits long. 0 One start bit, eight data bits, one stop bit 1 One start bit, nine data bits, one stop bit
3 WAKE	**Wakeup Condition Bit** — WAKE determines which condition wakes up the SCI: a logic 1 (address mark) in the most significant bit position of a received data character or an idle condition on the RXD pin. 0 Idle line wakeup 1 Address mark wakeup
2 ILT	**Idle Line Type Bit** — ILT determines when the receiver starts counting logic 1s as idle character bits. The counting begins either after the start bit or after the stop bit. If the count begins after the start bit, then a string of logic 1s preceding the stop bit may cause false recognition of an idle character. Beginning the count after the stop bit avoids false idle character recognition, but requires properly synchronized transmissions. 0 Idle character bit count begins after start bit 1 Idle character bit count begins after stop bit
1 PE	**Parity Enable Bit** — PE enables the parity function. When enabled, the parity function inserts a parity bit in the most significant bit position. 0 Parity function disabled 1 Parity function enabled
0 PT	**Parity Type Bit** — PT determines whether the SCI generates and checks for even parity or odd parity. With even parity, an even number of 1s clears the parity bit and an odd number of 1s sets the parity bit. With odd parity, an odd number of 1s clears the parity bit and an even number of 1s sets the parity bit. 1 Even parity 1 Odd parity

This register controls the main communications behaviours of the module. For example, we probably 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.

So, we probably want ***SCI0CR1 = 0b00000000***!

(Note the typo in the description of "Parity Type Bit". I guess errors are to be expected in a 1300 page document!)

The final configuration register to consider is the SCI Control Register 2.

11.3.2.6 SCI Control Register 2 (SCICR2)

	7	6	5	4	3	2	1	0
R W	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
Reset	0	0	0	0	0	0	0	0

Figure 11-9. SCI Control Register 2 (SCICR2)

Read: Anytime

Write: Anytime

Table 11-9. SCICR2 Field Descriptions

Field	Description
7 TIE	**Transmitter Interrupt Enable Bit** — TIE enables the transmit data register empty flag, TDRE, to generate interrupt requests. 0 TDRE interrupt requests disabled 1 TDRE interrupt requests enabled
6 TCIE	**Transmission Complete Interrupt Enable Bit** — TCIE enables the transmission complete flag, TC, to generate interrupt requests. 0 TC interrupt requests disabled 1 TC interrupt requests enabled
5 RIE	**Receiver Full Interrupt Enable Bit** — RIE enables the receive data register full flag, RDRF, or the overrun flag, OR, to generate interrupt requests. 0 RDRF and OR interrupt requests disabled 1 RDRF and OR interrupt requests enabled
4 ILIE	**Idle Line Interrupt Enable Bit** — ILIE enables the idle line flag, IDLE, to generate interrupt requests. 0 IDLE interrupt requests disabled 1 IDLE interrupt requests enabled
3 TE	**Transmitter Enable Bit** — TE enables the SCI transmitter and configures the TXD pin as being controlled by the SCI. The TE bit can be used to queue an idle preamble. 0 Transmitter disabled 1 Transmitter enabled
2 RE	**Receiver Enable Bit** — RE enables the SCI receiver. 0 Receiver disabled 1 Receiver enabled
1 RWU	**Receiver Wakeup Bit** — Standby state 0 Normal operation. 1 RWU enables the wakeup function and inhibits further receiver interrupt requests. Normally, hardware wakes the receiver by automatically clearing RWU.
0 SBK	**Send Break Bit** — Toggling SBK sends one break character (10 or 11 logic 0s, respectively 13 or 14 logics 0s if BRK13 is set). Toggling implies clearing the SBK bit before the break character has finished transmitting. As long as SBK is set, the transmitter continues to send complete break characters (10 or 11 bits, respectively 13 or 14 bits). 0 No break characters 1 Transmit break characters

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.

So, for now, the best choice for configuration is ***SCI0CR2 = 0b00001100***.

SCI0 Library

The following is the SCI0_Lib.h header file that will probably be provided to you by your instructor.

```
//Com port SCI0 initialization and commands
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//May 2015

void SCI0_Init(unsigned long);  //any valid baud rate can be passed to this; 8-bit, 1 Stop, No parity, no interrupts
void SCI0_Init9600(void);       //8-bit, 1 Stop, No parity, no interrupts
void SCI0_Init19200(void);      //8-bit, 1 stop, No parity, no interrupts
void SCI0_TxChar(unsigned char);
unsigned char SCI0_RxChar(void);    //Non-blocking; returns NULL if no new valid character is available
void SCI0_TxString(char *);     //Requires a NULL-terminated ASCII string in the main program
```

To begin with, you should create the appropriate functions in your SCI0_Lib.c to match the prototypes for the following two initialization routines. Once these are working, you should develop the first routine, *SCI0_Init(lBaud)*, which provides you the flexibility of operating in any of the valid baud rates, but requires that you know what these valid rates are.

- SCI0_Init9600
- SCI0_Init19200

Unfortunately, you won't be able to verify these initialization routines until you've created functions to communicate through the SCI0 port. That's our next item of discussion.

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 the current status of the port. The following selection from the data sheet has been abbreviated to discuss the only two bits that are significant to us at this point. If you need enhanced error handling, consult the full discussion in the data sheet.

11.3.2.7 SCI Status Register 1 (SCISR1)

The SCISR1 and SCISR2 registers provides inputs to the MCU for generation of SCI interrupts. Also, these registers can be polled by the MCU to check the status of these bits. The flag-clearing procedures require that the status register be read followed by a read or write to the SCI data register.It is permissible to execute other instructions between the two steps as long as it does not compromise the handling of I/O, but the order of operations is important for flag clearing.

	7	6	5	4	3	2	1	0
R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
W								
Reset	1	1	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 11-10. SCI Status Register 1 (SCISR1)

Read: Anytime

Write: Has no meaning or effect

Table 11-10. SCISR1 Field Descriptions

Field	Description
7 TDRE	**Transmit Data Register Empty Flag** — TDRE is set when the transmit shift register receives a byte from the SCI data register. When TDRE is 1, the transmit data register (SCIDRH/L) is empty and can receive a new value to transmit.Clear TDRE by reading SCI status register 1 (SCISR1), with TDRE set and then writing to SCI data register low (SCIDRL). 0 No byte transferred to transmit shift register 1 Byte transferred to transmit shift register; transmit data register empty
5 RDRF	**Receive Data Register Full Flag** — RDRF is set when the data in the receive shift register transfers to the SCI data register. Clear RDRF by reading SCI status register 1 (SCISR1) with RDRF set and then reading SCI data register low (SCIDRL). 0 Data not available in SCI data register 1 Received data available in SCI data register

When transmitting, check for availability of the port using the Transmit Data Register Empty Flag (TDRE) in bit 7 before attempting to write to the port. If TDRE is a '1', then it is OK to write a new byte to the SCIDRL. NOTE: TDRE does not indicate that transmission is complete – it only indicates that the transmit data register is empty, which is good enough for us. Bit 6 (missing from the above clip) indicates when transmission is actually complete.

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.

There's no point in reading the register to get a received byte until a valid new one has actually been received. You can check to see if a byte has been received since the last *read* 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 new byte of data has been received by the module. So, 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. In actual operation, the byte may never be sent due to a failure in communication, so 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. In an Assembly Language routine, it would be typical to use a condition-code register bit, such as *Carry*, to indicate whether or not a new byte has been received. However, in *C*, that isn't a workable plan. Instead, you may want to return a *NULL* character (ASCII code 0), as that's a very unlikely character to have appearing in a transmitted file. (In some protocols, the actual transmission of a delimiting character like this is indicated by sending the character twice; the programmer would need to do an error-trapping routine that would recognize this condition and handle the character as a special case.) In our case, we'll simply write our *main* program so that it treats any *NULL* returned as an indication that no valid character is present, so we'll ignore that result.

At this point, you will want to write the following functions indicated in your library header file:

- SCI0_TxChar
- SCI0_RxChar

Once you've written these, you can do a loop-back test by running *SCI0_RxChar* to receive a character from your computer's keyboard, then sending that character by running *SCI0_TxChar* to send the character back to your computer, operating as a "dumb terminal". The software you will likely be asked to use is called Tera Term Pro.

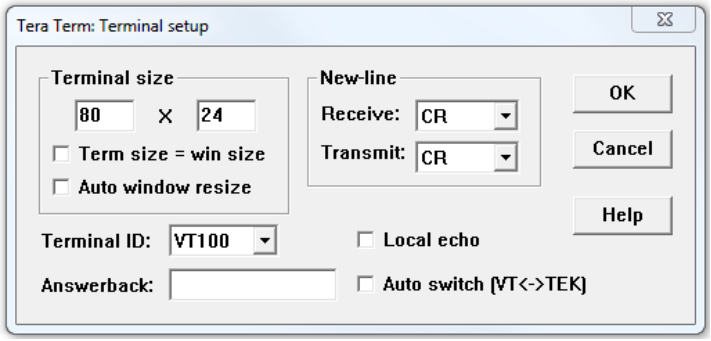
Terminal Emulation

This is your chance to make a highly-advanced and super-fast computer act like a 1960's remote terminal, designed to allow mere mortals to communicate with main-frame computers like the UNIVAC! Here are a couple of pictures of real terminals, which you are going to emulate. The one on the left uses paper instead of a monitor!

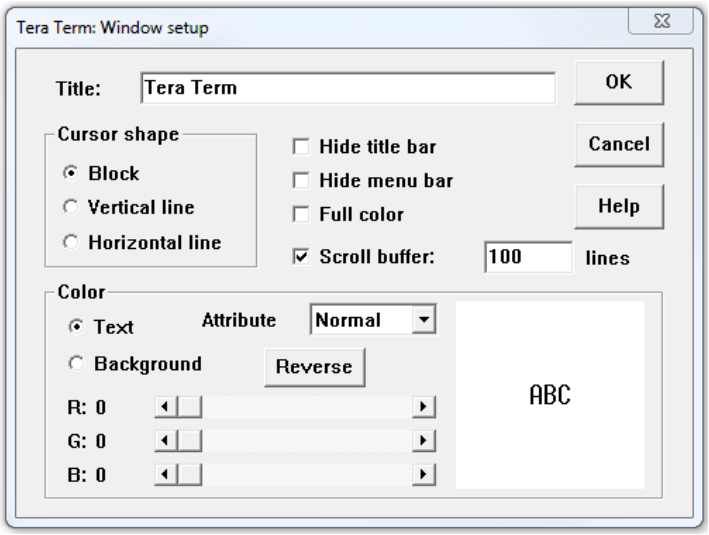
The term "dumb terminal" was coined to indicate that, in this configuration, your computer will be doing nothing other than sending and receiving ASCII characters, just like the terminals shown on the previous page. Of course, since our computers are multi-tasking devices, they will actually be doing a gazillion things in the background; however, the terminal emulator window will not be doing anything other than acting as a dumb terminal.

Inside Tera Term, you will need to set up a number of characteristics so that the dumb terminal can communicate with your 9S12 development kit. These are found in the "Setup" menu.

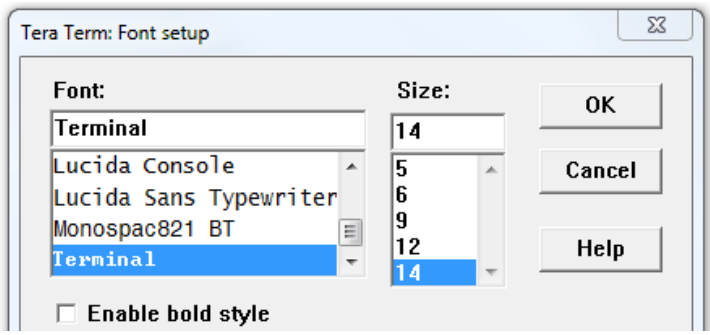
First, you need to make Tera Term into the right dumb terminal. Here's the "Terminal" setup you want:



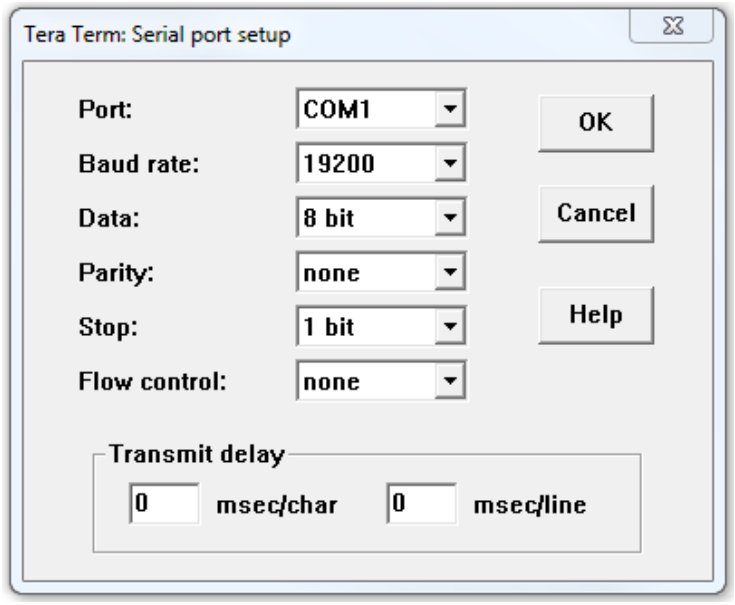
Next, you'll probably want your "Window" setup to look like the following:



The following "Font" setup makes your display fairly readable:

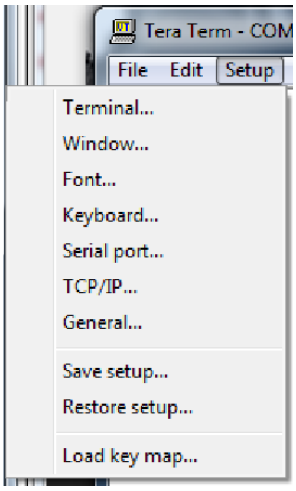


Last but certainly not least, you need to do the following "Serial Port" setup:



This setup matches the settings for the *SCI0_Init19200* initialization routine in your library. If you want to communicate at a different baud rate, this is where you would select that.

Before you leave Tera Term Pro, you will want to save these configuration changes so that you don't have to do this every time you run the dumb terminal.



From the "Setup" menu, select "Save setup…" and simply accept the defaults. This will overwrite the default setup file with your new settings, and Tera Term will start up the way you want it to for the rest of the semester (unless your computer gets re-Ghosted).

Once you've done this configuration, have written a program that initializes the 9S12X, have connected your 9S12X board to the computer using a 9-pin Comm cable, and have written a small program to receive a character from the keyboard and transmit it back to the dumb terminal, you should be able to verify that your functions are working properly (or not). Have fun!

The VT100/VT52 Terminal

The terminal program on the PC side of the connection should now be set up to emulate a VT100 terminal. These terminals were able to display received 7-bit ASCII characters, manage a cursor, and send characters from a keyboard through an RS-232 connection.

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. They 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 control the terminal.

The following small table shows a few of the sequences of interest. A more complete, but not entirely trustworthy, 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, or 0x1B)	**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

Previously, you created ToUpper() and ToLower() routines. One example of using these routines would be in handling the response to a "Y/N" question, received using SCI0_RxChar(). The operator could just as easily enter 'y' instead of 'Y', or 'n' instead of 'N', so your program should respond to either case. This is easily done by simply running the response through ToUpper() and responding to the uppercase value returned.

SCI0_TxString

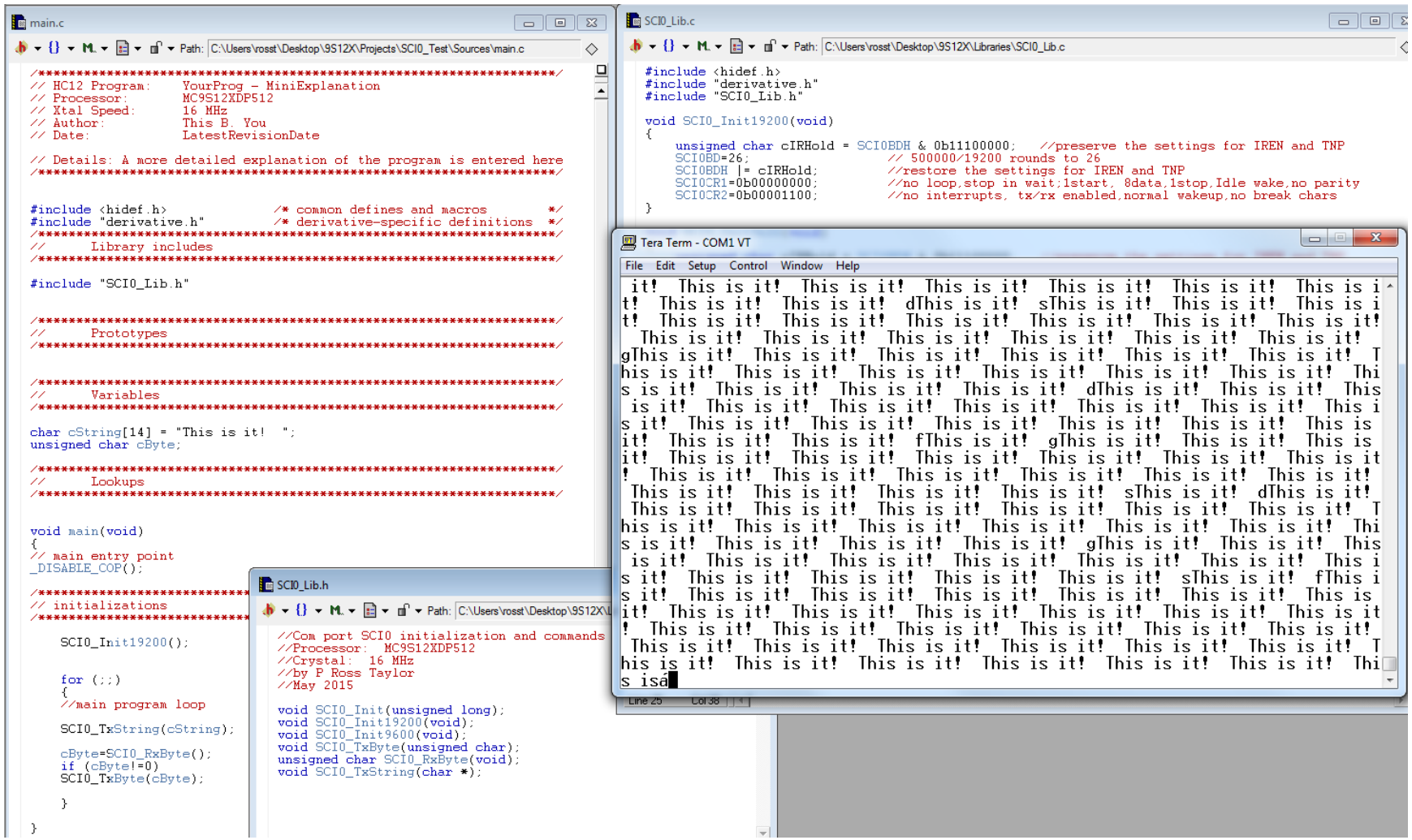
Quite often, you'll want to send multiple characters to the dumb terminal (or to other peripherals or equipment that's looking for ASCII characters). For example, you probably don't want to send the characters for the previously-listed escape sequences one-by-one every time you want to control the terminal. As with the LCD, the best way to send these sequences, as well as longer strings of characters, is by using a ***null-terminated string*** of ASCII characters. The *SCI0_TxString* function you will be creating can handle strings of any length, since it is expecting a string that ends with the *NULL* character (ASCII code 0). The routine transmits each character, then checks to see if the character was a *NULL*. If it is a *NULL*, program execution exits the function.

You should be able to generalize what you did with LCD_String(*) to develop SCI0_TxString(*). If you need further assistance, your instructor can give you direction.

The simple string management program shown on the next page should be a good reference for your work with the SCI port. Here are some pertinent things to notice:

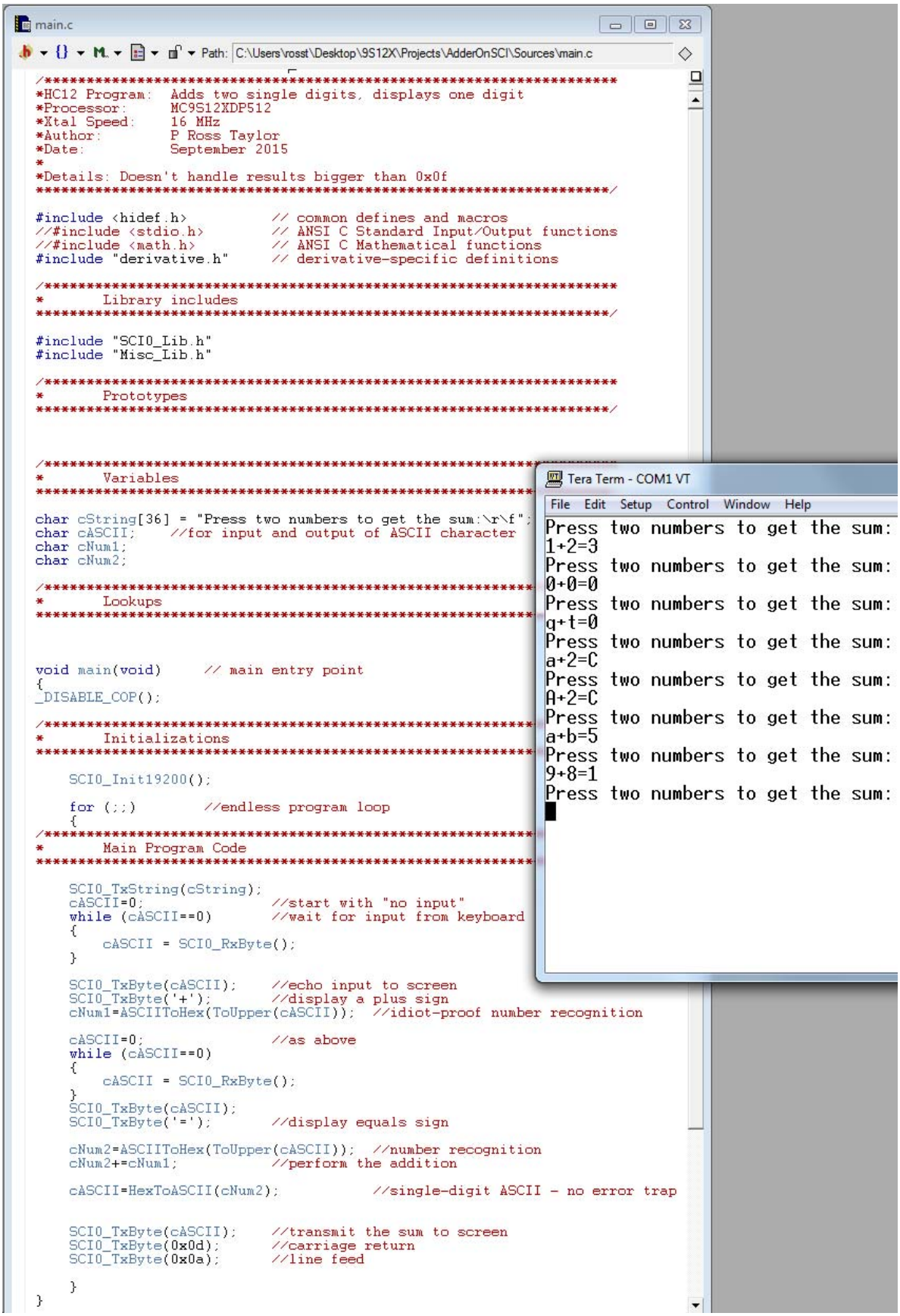
1. The author's version of *SCI0_Init19200* is shown, from which you can build the other initialization routines required. Although it's probably not necessary for most of what you'll be doing, you'll notice that the contents of the upper three bits of the sixteen-bit baud rate register *SCI0BD* have been preserved, in case there's a chance that the infrared settings might be significant. They really shouldn't be, because this SCI port is connected to a wired comm port.
2. Notice that you need to calculate the value for the baud rate register, *SCI0BD*.
3. Note that there was no need to add a *NULL* character to the end of the string – the IDE does that automatically. However, in the declaration of the array size, room had to be provided for the *NULL*.
4. The undeclared *for (;;){}* loop provides an endless loop, which shows up on the screen as an endless printout of the string.
5. Notice that occasionally, there's an unexpected letter appearing on screen. These characters have been entered from the keyboard using *SCI0_RxByte()*. Note that we read the SCI port, check to see if it contains a valid character from the keyboard, and if it does, we print it.

Note: The header information shown on the next page hasn't been updated to show who wrote it, when, and what it does. As Stan and Jan Berenstain once famously said "That is what you should not do. So let that be a lesson to you." (The Bike Lesson by Stan and Jan Berenstain, 1964 Random House, Inc.)


main.c
Path: C:\Users\rosst\Desktop\9S12X\Projects\SCI0_Test\Sources\main.c
// 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 "SCI0_Lib.h"
// Prototypes
// Variables
char cString[14] = "This is it! ";
unsigned char cByte;
// Lookups
void main(void)
{
// main entry point
_DISABLE_COP();
// initializations
SCI0_Init19200();
for (;;)
{
//main program loop
SCI0_TxString(cString);
cByte=SCI0_RxByte();
if (cByte!=0)
SCI0_TxByte(cByte);
}
}
SCI0_Lib.h
Path: C:\Users\rosst\Desktop\9S12X\
//Com port SCI0 initialization and commands
//Processor: MC9S12XDP512
//Crystal: 16 MHz
//by P Ross Taylor
//May 2015
void SCI0_Init(unsigned long);
void SCI0_Init19200(void);
void SCI0_Init9600(void);
void SCI0_TxByte(unsigned char);
unsigned char SCI0_RxByte(void);
void SCI0_TxString(char *);
SCI0_Lib.c
Path: C:\Users\rosst\Desktop\9S12X\Libraries\SCI0_Lib.c
#include <hidef.h>
#include "derivative.h"
#include "SCI0_Lib.h"
void SCI0_Init19200(void)
{
unsigned char cIRHold = SCI0BDH & 0b11100000; //preserve the settings for IREN and TNP
SCI0BD=26; // 500000/19200 rounds to 26
SCI0BDH |= cIRHold; //restore the settings for IREN and TNP
SCI0CR1=0b00000000; //no loop,stop in wait;1start, 8data,1stop,Idle wake,no parity
SCI0CR2=0b00001100; //no interrupts, tx/rx enabled,normal wakeup,no break chars
}
Tera Term - COM1 VT
File Edit Setup Control Window Help
it! This is it! This is it! This is it! This is it! This is it! This is i
t! This is it! This is it! dThis is it! sThis is it! This is it! This is i
t! This is it! This is it! This is it! This is it! This is it! This is it!
This is it! This is it! This is it! This is it! This is it! This is it!
gThis is it! This is it! This is it! This is it! This is it! This is it! T
his is it! This is it! This is it! This is it! This is it! This is it! Thi
s is it! This is it! This is it! This is it! dThis is it! This is it! This
is it! This is it! This is it! This is it! This is it! This is it! This i
s it! This is it! This is it! This is it! This is it! This is it! This is
it! This is it! This is it! fThis is it! gThis is it! This is it! This is
it! This is it! This is it! This is it! This is it! This is it! This is it
! This is it! This is it! This is it! This is it! This is it! This is it!
This is it! This is it! This is it! This is it! sThis is it! dThis is it!
This is it! This is it! This is it! This is it! This is it! This is it! T
his is it! This is it! This is it! This is it! This is it! This is it! Thi
s is it! This is it! This is it! This is it! gThis is it! This is it! This
is it! This is it! This is it! This is it! This is it! This is it! This i
s it! This is it! This is it! This is it! This is it! sThis is it! fThis i
s it! This is it! This is it! This is it! This is it! This is it! This is
it! This is it! This is it! This is it! This is it! This is it! This is it
! This is it! This is it! This is it! This is it! This is it! This is it!
This is it! This is it! This is it! This is it! This is it! This is it! T
his is it! This is it! This is it! This is it! This is it! This is it! Thi
s isa

The next page is a screen capture of a simple "adder" that demonstrates, among other things, the way in which ASCII values from the keyboard need to be manipulated in order to do simple math, and how the results need to be manipulated back to ASCII codes in order to display them in a manner that is meaningful to human operators. Here are some pertinent things to notice:

1. In the string declaration, the escape characters '\r\f' are interpreted as a carriage return and line feed respectively, moving the text display on screen down to the beginning of the next line.
2. The string array size is declared to be one larger than the number of characters, to allow for the *NULL* terminator.
3. Single characters can be sent to the terminal by putting single quotes around them.
4. Special characters can be sent to the terminal as hexadecimal numbers, or, for that matter, binary, octal, or decimal numbers. 0x0d and 0x0a are the ASCII codes for carriage return and line feed respectively.
5. In order to add two numbers together, they must be true numbers, not ASCII codes; In order to display a true number, it must be converted to ASCII for the terminal.
6. This simple program doesn't check for non-valid (i.e. non-numeric) entries. A better program would reject these and would wait for a valid input.
7. Note that the larger hex values (A – F) can be entered as either lowercase or uppercase, and are interpreted correctly.
8. This simple routine can't handle results that are larger than a single digit (i.e. 0x10 up to 0x1E, which is 0x0F + 0x0F). Only the lower digit is displayed. A better program would send two characters, or would at least trap the error.


main.c
Path: C:\Users\rosst\Desktop\9S12X\Projects\AdderOnSCI\Sources\main.c
/***
*HC12 Program: Adds two single digits, displays one digit
*Processor: MC9S12XDP512
*Xtal Speed: 16 MHz
*Author: P Ross Taylor
*Date: September 2015
*
*Details: Doesn't handle results bigger than 0x0f
***/
#include <hidef.h>   // common defines and macros
//#include <stdio.h>   // ANSI C Standard Input/Output functions
//#include <math.h>   // ANSI C Mathematical functions
#include "derivative.h"   // derivative-specific definitions
/***
*   Library includes
***/
#include "SCI0_Lib.h"
#include "Misc_Lib.h"
/***
*   Prototypes
***/
/***
*   Variables
***/
char cString[36] = "Press two numbers to get the sum:\r\f";
char cASCII;   //for input and output of ASCII character
char cNum1;
char cNum2;
/***
*   Lookups
***/
void main(void)   // main entry point
{
_DISABLE_COP();
/***
*   Initializations
***/
    SCI0_Init19200();
    for (;;)   //endless program loop
    {
/***
*   Main Program Code
***/
    SCI0_TxString(cString);
    cASCII=0;   //start with "no input"
    while (cASCII==0)   //wait for input from keyboard
    {
        cASCII = SCI0_RxByte();
    }
    SCI0_TxByte(cASCII);   //echo input to screen
    SCI0_TxByte('+');   //display a plus sign
    cNum1=ASCIIToHex(ToUpper(cASCII)); //idiot-proof number recognition
    cASCII=0;   //as above
    while (cASCII==0)
    {
        cASCII = SCI0_RxByte();
    }
    SCI0_TxByte(cASCII);
    SCI0_TxByte('=');   //display equals sign
    cNum2=ASCIIToHex(ToUpper(cASCII)); //number recognition
    cNum2+=cNum1;   //perform the addition
    cASCII=HexToASCII(cNum2);   //single-digit ASCII - no error trap
    SCI0_TxByte(cASCII);   //transmit the sum to screen
    SCI0_TxByte(0x0d);   //carriage return
    SCI0_TxByte(0x0a);   //line feed
    }
}
Tera Term - COM1 VT
File Edit Setup Control Window Help
Press two numbers to get the sum:
1+2=3
Press two numbers to get the sum:
0+0=0
Press two numbers to get the sum:
q+t=0
Press two numbers to get the sum:
a+2=C
Press two numbers to get the sum:
A+2=C
Press two numbers to get the sum:
a+b=5
Press two numbers to get the sum:
9+8=1
Press two numbers to get the sum:

Interrupts

Up to this point, you've been using a technique called "polling" when designing your software for the 9S12X. 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 other processes temporarily take control when another event 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.

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.

Interrupts in S12XCPU Assembly Language

Although in this course we won't spend more time programming in S12XCPU Assembly Language, we will revisit it here in order to gain an understanding of how interrupts work.

This is a good time to compare what's required, in S12XCPU Assembly Language, 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

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

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.

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.)

The following code snippets show how to set up and use the SCI Receive interrupt to read a character from the keyboard and echo it to the screen, while spending the remainder of the time in a power-down WAIT condition.

```
;Initializations
      JSR   SCI0Init192

      BSET  SCI0SR1,%00100000   ;clear receive flag
      BSET  SCI0CR2,%00100000   ;enable RIE interrupt

      CLI                       ;enable interrupts

Main:

RegularLoop:
       WAI
       BRA   RegularLoop

;*****************************************************************
;*    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 from keyboard
    JSR   SCI0TxByte        ;echo to screen
    RTI

;*****************************************************************
;*    Interrupt Vectors
;*****************************************************************
    ORG     $FFD6     ;SCI0 SCI1CR2
    DC.W    SCI_Echo_ISR
```

The screen capture below shows just the top of a three-page listing of interrupt vectors, enough to show where the microcontroller jumps to if an interrupt occurs on Timer Channel 0, which we're using in this example:

Table 1-12. Interrupt Vector Locations (Sheet 1 of 3)

Vector Address[1]	XGATE Channel ID[2]	Interrupt Source	CCR Mask	Local Enable
$FFFE	—	System reset or illegal access reset	None	None
$FFFC	—	Clock monitor reset	None	PLLCTL (CME, SCME)
$FFFA	—	COP watchdog reset	None	COP rate select
Vector base + $F8	—	Unimplemented instruction trap	None	None
Vector base+ $F6	—	SWI	None	None
Vector base+ $F4	—	$\overline{\text{XIRQ}}$	X Bit	None
Vector base+ $F2	—	$\overline{\text{IRQ}}$	I bit	IRQCR (IRQEN)
Vector base+ $F0	$78	Real time interrupt	I bit	CRGINT (RTIE)
Vector base+ $EE	$77	Enhanced capture timer channel 0	I bit	TIE (C0I)
Vector base + $EC	$76	Enhanced capture timer channel 1	I bit	TIE (C1I)
Vector base+ $EA	$75	Enhanced capture timer channel 2	I bit	TIE (C2I)

The default Vector base is $FF00, so you would add this to the Vector Address column value.

It's possible to have multiple Interrupt Service Routines running simultaneously. The code shown on the following page counts up endlessly on the bottom four digits of the seven-segment display, maintains a timer, grabs characters from console when interrupted by the SCI0 Receive Interrupt, and clears the display in response to an interrupt generated on a switch connected to bit 0 of PortJ.

```
;*****************************************************************
;* HC12 Program:    Creates an upcounter based on an interrupt timer    *
;* Processor:   MC9S12XDP512                                     *
;* Xtal Speed:  16 MHz                                           *
;* Author:      P Ross Taylor                                    *
;* Date:        April 2014                                       *
;*                                                               *
;* Details: Also responds to interrupts from the SCI Rx (keyboard)      *
;*                                                               *
;*****************************************************************

;export symbols
        XDEF        Entry           ;export'Entry' symbol
        ABSENTRY    Entry           ;for absolute assembly: app entry point

;include derivative specific macros
        INCLUDE 'derivative.inc'

;*****************************************************************
;*      Equates                                                  *
;*****************************************************************


;*****************************************************************
;*      Variables                                                *
;*****************************************************************
        ORG         RAMStart        ;Address $2000
 TimCounter:        DC.W    1

;*****************************************************************
;*      Code Section                                             *
;*****************************************************************
        ORG         ROM_4000Start   ;Address $4000 (FLASH)
Entry:
        LDS         #RAMEnd+1       ;initialize the stack pointer

Main:
;Initializations
            JSR     SevSeg_Init
            JSR     SCI0Init192

            MOVB    #%10000000,TSCR1   ;enable timer module 0
            LDAB    #%00000111         ;2^7 prescaler
            STAB    TSCR2              ;prescaler set to Dus^(2^D)
            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
            BSET    TIE,%00000001      ;enable OC0 interrupt

            BSET    SCI0SR1,%00100000  ;clear receive flag
            BSET    SCI0CR2,%00100000  ;enable RIE interrupt

            BCLR    DDRJ,%00000001     ;ensure input at PortJ0
            BSET    PPSJ,%00000001     ;set polarity for rising edge (key press)
            BSET    PIFJ,%00000001     ;clear PortJ0 flag
            BSET    PIEJ,%00000001     ;enable PortJ0 interrupt

            CLI                        ;enable interrupts

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


;*****************************************************************
;*      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

;*****************************************************************
;*      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
        RTI

;*****************************************************************
;*      Switch_ISR                                               *
;*Responds to a press of the PortJ0 switch                      *
;*****************************************************************

Switch_ISR:
        BSET    PIFJ,%00000001      ;clear interrupt flag
        LDD     #0                  ;ready to blank top display
        STD     TimCounter
        RTI

;*****************************************************************
;*      Constants                                                *
;*****************************************************************
        ORG         ROM_C000Start   ;second block of ROM

;*****************************************************************
;*      Look-Up Tables                                           *
;*****************************************************************

;*****************************************************************
;*      SCI VT100 Strings                                        *
;*****************************************************************

;*****************************************************************
;*      Absolute Library Includes                                *
;*****************************************************************

        INCLUDE "Misc_Lib.inc"
        INCLUDE "SCI0_Lib.inc"
        INCLUDE "SevSeg_Lib.inc"

;*****************************************************************
;*      Interrupt Vectors                                        *
;*****************************************************************
        ORG         $FFD6           ;SCI0 SCI1CR2
        DC.W        SCI_Echo_ISR

        ORG         $FFCE           ;Port J
        DC.W        Switch_ISR

        ORG         $FFEE           ;TIE C0I
        DC.W        Timer_ISR

        ORG         $FFFE
        DC.W        Entry           ;Reset Vector
```

Interrupts using ANSI C

Interrupt handling in ANSI C is different from handling interrupts 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 that everything gets stacked upon entry and then everything gets pulled back off the stack at the end of the ISR), we'll have to use global variables for anything we want to send to or receive from an interrupt routine.

Recall 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.

For the SCI port, the most useful interrupt is the Receive Data Register Full (RDRF), which we have used in our polling routines. Since people type pretty slowly, compared to the processing rate of a microcontroller, and since people tend to take relatively long breaks to think about what they are typing or to get a coffee, leaving the microcontroller in a blocking routine while it waits for a new character is an incredible waste of time and computing power.

If you don't want to try and find the Vector Handlers in the 1300 page data sheet, you can open up the "mc9s12xdp512.h" file that always shows up in your projects when you make the correct initial selections. Here's a snippet out of that file, which should be generally useful for the interrupt-driven parts of this course.

```
#define VectorNumber_Vporth             25U
#define VectorNumber_Vportj             24U
#define VectorNumber_Vatd1              23U
#define VectorNumber_Vatd0              22U
#define VectorNumber_Vsci1              21U
#define VectorNumber_Vsci0              20U
#define VectorNumber_Vspi0              19U
#define VectorNumber_Vtimpaie           18U
#define VectorNumber_Vtimpaaovf         17U
#define VectorNumber_Vtimovf            16U
#define VectorNumber_Vtimch7            15U
#define VectorNumber_Vtimch6            14U
#define VectorNumber_Vtimch5            13U
#define VectorNumber_Vtimch4            12U
#define VectorNumber_Vtimch3            11U
#define VectorNumber_Vtimch2            10U
#define VectorNumber_Vtimch1            9U
#define VectorNumber_Vtimch0            8U
```

Here are some important facts to use in setting up an SCI0 Receive interrupt routine:

- SCI0 interrupts are handled by "VectorNumber_Vsci0". (Be careful, as the cross-compiler is case sensitive.)
- RDRF is bit5 of SCI0SR1.
- Flags are cleared by writing a "1" to them.
- The receive interrupt enable, RIE, is bit5 of SCI0CR2.
- The interrupts you've selected are ultimately enabled using "EnableInterrupts", which is the equivalent of "CLI".
- You can use "asm WAI" to put the microcontroller to sleep, waiting for an interrupt.

Points 2 to 5 of the things that are needed for an interrupt-driven program are done for you in the following screen capture. Notice, in particular, the syntax of the declaration of the vector handler and its associated interrupt service routine – the top line of this code. The function is void(void) because everything is placed on the stack and retrieved from the stack. If you want to have the ISR work on values or provide results, use global variables as the memory access points.

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

interrupt VectorNumber_Vsci0 void SCI_Echo(void)
  {
  //put your code here
    SCI0SR1|=0b00100000;          //clear the flag
  }

/**************************************************************************/
```

At this point, you should be able to write an ANSI C equivalent of the "receive and echo" program shown earlier in S12XCPU Assembly Language.

By the way, you may have noticed that, in one of the ISRs the flag was cleared at the beginning of the routine and in the other the flag was cleared at the end. This doesn't matter with Motorola-based microcontrollers, as the interrupt is automatically disabled as long as the interrupt is being serviced.

Input-Driven Interrupt

Often, we want our microcontroller to respond to simple external events, such as can be detected by a logic change on an input.

On our board, Port J can be used to generate interrupts in response to a change in the signals connected to it.

The microcontroller board has PortJ0 and PortJ1 connected to push-button switches. Of course, you could also wire one or both of these to a circuit of your own design, or an add-on peripheral, that generates binary logic levels as external interrupts.

The following is from page 819 of the "Data Sheet":

0x0268	Port J Data Register (PTJ)	Read / Write[1]
0x0269	Port J Input Register (PTIJ)	Read
0x026A	Port J Data Direction Register (DDRJ)	Read / Write[1]
0x026B	Port J Reduced Drive Register (RDRJ)	Read / Write[1]
0x026C	Port J Pull Device Enable Register (PERJ)	Read / Write[1]
0x026D	Port J Polarity Select Register (PPSJ)	Read / Write[1]
0x026E	Port J Interrupt Enable Register (PIEJ)	Read / Write[1]
0x026F	Port J Interrupt Flag Register (PIFJ)	Read / Write[1]

In order to use PortJ0 and/or PortJ1, we need to define them as inputs, using DDRJ.

Also, we need to specify if we want interrupts to be generated on a rising or falling edge of the input signal using PPSJ, where 0 is for a falling edge and 1 is for a rising edge.

Once the port is set up, we can enable the interrupt for our particular channel using PIEJ.

Interrupt events will be reported using PIFJ, which will have to be cleared before further interrupts can be detected.

The Interrupt Vector for PortJ is $FFCE, and the Interrupt Vector Handler for ANSI C is VectorNumber_Vportj.

There are some distinct advantages to using an interrupt service routine for binary input signals, particularly if they are generated by switches.

For one, we don't need to worry about long activation time for the switch, because a single event gets us into the service routine, and further events won't have any effect until we clear the interrupt flag and exit the ISR. Holding the switch down doesn't generate any more edges, so no further action is detected until the switch is reactivated. Also, events are only initiated by the edge we've chosen, so, for example, if we've chosen a rising edge to detect switch press, the falling edge for the switch release condition will be ignored.

Switch bounce is also significantly reduced, as the switch will probably stabilize during the operation of the interrupt service routine. If your switch bounces upon release, however, you may get an unwanted event after the service routine has finished its operation, which you may need to find a solution for. This will probably vary from one application to another.

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 9S12X data sheet. In the current version, this is on page 310. Here it is for quick reference:

Chapter 7 Enhanced Capture Timer (S12ECT16B8CV2)

7.1.3 Block Diagram

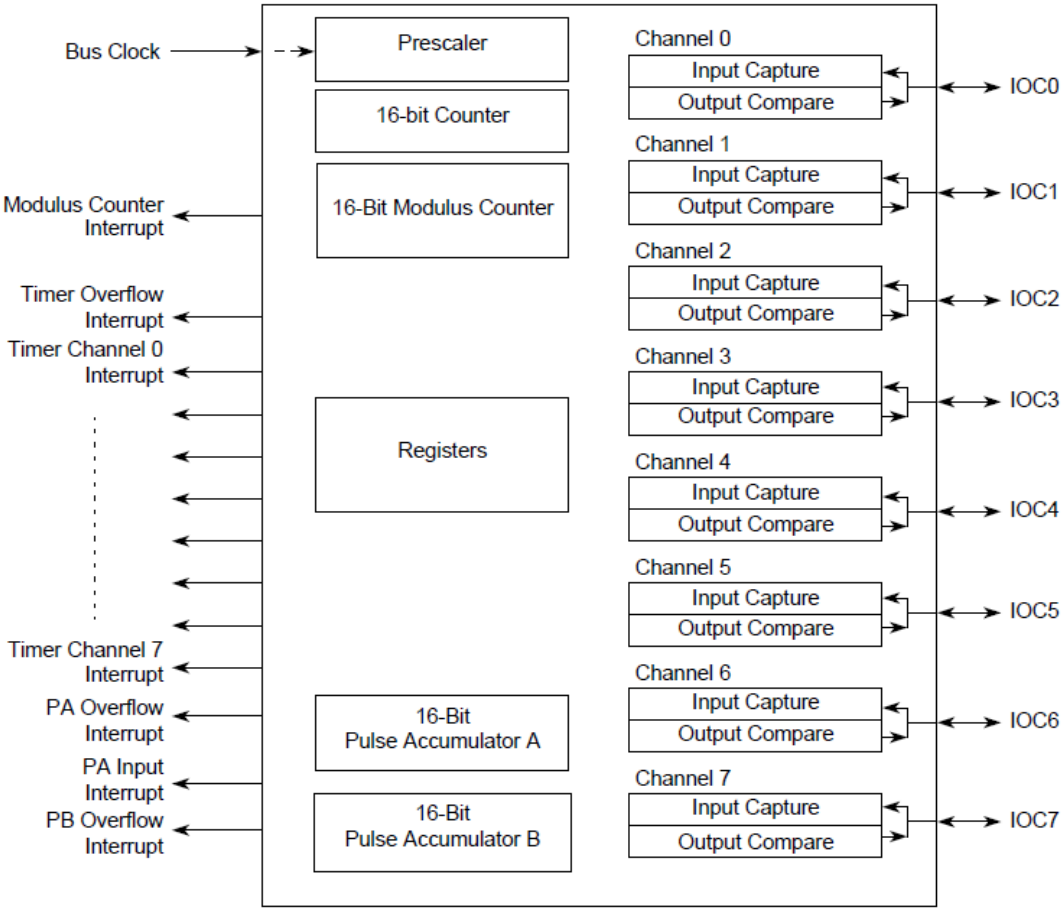


Figure 7-1. ECT Block Diagram

In other versions of the 9S12 microcontroller, there are multiple enhanced capture timers available. However, in the MC9S12XDP512, the manufacturers made room for multiple CAN Bus modules by removing all but one of the timer modules. In those versions of the controller, you need to specify which module you want (e.g. TIM0_?????). We don't have to do that with this version of the controller.

At the core of the timer module exists a 16-bit counter, the current count value being available at the 16-bit location TCNT (for us, this is at addresses 0x0044 and 0x0045). 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.

There are a number of registers that need to either be written to for control or read from to determine the current status of the timer module. Here's a screen-shot showing all of the available registers. Pretty daunting!

7.3 Memory Map and Register Definition

This section provides a detailed description of all memory and registers.

7.3.1 Module Memory Map

The memory map for the ECT module is given below in Table 7-1. The address listed for each register is the address offset. The total address for each register is the sum of the base address for the ECT module and the address offset for each register.

Table 7-1. ECT Memory Map

Address Offset	Register	Access
0x0000	Timer Input Capture/Output Compare Select (TIOS)	R/W
0x0001	Timer Compare Force Register (CFORC)	R/W[1]
0x0002	Output Compare 7 Mask Register (OC7M)	R/W
0x0003	Output Compare 7 Data Register (OC7D)	R/W
0x0004	Timer Count Register High (TCNT)	R/W[2]
0x0005	Timer Count Register Low (TCNT)	R/W[2]
0x0006	Timer System Control Register 1 (TSCR1)	R/W
0x0007	Timer Toggle Overflow Register (TTOV)	R/W
0x0008	Timer Control Register 1 (TCTL1)	R/W
0x0009	Timer Control Register 2 (TCTL2)	R/W
0x000A	Timer Control Register 3 (TCTL3)	R/W
0x000B	Timer Control Register 4 (TCTL4)	R/W
0x000C	Timer Interrupt Enable Register (TIE)	R/W
0x000D	Timer System Control Register 2 (TSCR2)	R/W
0x000E	Main Timer Interrupt Flag 1 (TFLG1)	R/W
0x000F	Main Timer Interrupt Flag 2 (TFLG2)	R/W
0x0010	Timer Input Capture/Output Compare Register 0 High (TC0)	R/W[3]
0x0011	Timer Input Capture/Output Compare Register 0 Low (TC0)	R/W[3]
0x0012	Timer Input Capture/Output Compare Register 1 High (TC1)	R/W[3]
0x0013	Timer Input Capture/Output Compare Register 1 Low (TC1)	R/W[3]
0x0014	Timer Input Capture/Output Compare Register 2 High (TC2)	R/W[3]
0x0015	Timer Input Capture/Output Compare Register 2 Low (TC2)	R/W[3]
0x0016	Timer Input Capture/Output Compare Register 3 High (TC3)	R/W[3]
0x0017	Timer Input Capture/Output Compare Register 3 Low (TC3)	R/W[3]
0x0018	Timer Input Capture/Output Compare Register 4 High (TC4)	R/W[3]
0x0019	Timer Input Capture/Output Compare Register 4 Low (TC4)	R/W[3]
0x001A	Timer Input Capture/Output Compare Register 5 High (TC5)	R/W[3]
0x001B	Timer Input Capture/Output Compare Register 5 Low (TC5)	R/W[3]
0x001C	Timer Input Capture/Output Compare Register 6 High (TC6)	R/W[3]
0x001D	Timer Input Capture/Output Compare Register 6 Low (TC6)	R/W[3]

Table 7-1. ECT Memory Map (continued)

Address Offset	Register	Access
0x001E	Timer Input Capture/Output Compare Register 7 High (TC7)	R/W[3]
0x001F	Timer Input Capture/Output Compare Register 7 Low (TC7)	R/W[3]
0x0020	16-Bit Pulse Accumulator A Control Register (PACTL)	R/W
0x0021	Pulse Accumulator A Flag Register (PAFLG)	R/W
0x0022	Pulse Accumulator Count Register 3 (PACN3)	R/W
0x0023	Pulse Accumulator Count Register 2 (PACN2)	R/W
0x0024	Pulse Accumulator Count Register 1 (PACN1)	R/W
0x0025	Pulse Accumulator Count Register 0 (PACN0)	R/W
0x0026	16-Bit Modulus Down Counter Register (MCCTL)	R/W
0x0027	16-Bit Modulus Down Counter Flag Register (MCFLG)	R/W
0x0028	Input Control Pulse Accumulator Register (ICPAR)	R/W
0x0029	Delay Counter Control Register (DLYCT)	R/W
0x002A	Input Control Overwrite Register (ICOVW)	R/W
0x002B	Input Control System Control Register (ICSYS)	R/W[4]
0x002C	Reserved	--
0x002D	Timer Test Register (TIMTST)	R/W[2]
0x002E	Precision Timer Prescaler Select Register (PTPSR)	R/W
0x002F	Precision Timer Modulus Counter Prescaler Select Register (PTMCPSR)	R/W
0x0030	16-Bit Pulse Accumulator B Control Register (PBCTL)	R/W
0x0031	16-Bit Pulse Accumulator B Flag Register (PBFLG)	R/W
0x0032	8-Bit Pulse Accumulator Holding Register 3 (PA3H)	R/W[5]
0x0033	8-Bit Pulse Accumulator Holding Register 2 (PA2H)	R/W[5]
0x0034	8-Bit Pulse Accumulator Holding Register 1 (PA1H)	R/W[5]
0x0035	8-Bit Pulse Accumulator Holding Register 0 (PA0H)	R/W[5]
0x0036	Modulus Down-Counter Count Register High (MCCNT)	R/W
0x0037	Modulus Down-Counter Count Register Low (MCCNT)	R/W
0x0038	Timer Input Capture Holding Register 0 High (TC0H)	R/W[5]
0x0039	Timer Input Capture Holding Register 0 Low (TC0H)	R/W[5]
0x003A	Timer Input Capture Holding Register 1 High(TC1H)	R/W[5]
0x003B	Timer Input Capture Holding Register 1 Low (TC1H)	R/W[5]
0x003C	Timer Input Capture Holding Register 2 High (TC2H)	R/W[5]
0x003D	Timer Input Capture Holding Register 2 Low (TC2H)	R/W[5]
0x003E	Timer Input Capture Holding Register 3 High (TC3H)	R/W[5]
0x003F	Timer Input Capture Holding Register 3 Low (TC3H)	R/W[5]

1. Always read 0x0000.
2. Only writable in special modes (test_mode = 1).
3. Writes to these registers have no meaning or effect during input capture.
4. May be written once when not in test00mode but writes are always permitted when test00mode is enabled.
5. Writes have no effect.

Clearly, we can only touch on a small subset of all the capabilities of this very important module, so let's get started.

Initially, we will be working with seven of the registers:

TSCR1	Timer System Control Register 1
TSCR2	Timer System Control Register 2
TIOS	Timer Input Capture/Output Compare Select
TCTL2	Timer Control Register 2 (This is half of a 16-bit TCTL register)
TCNT	Timer Count Register
TC0	Timer Input Capture/Output Compare Register 0
TFLG1	Main Timer Interrupt Flag 1

Once we have the timer working, you might be interested in checking out the functionality of Input Capture, which requires TCTL3 and TCTL4, and reports the clock value when an external event happens. You could also try Pulse Accumulation, which requires PACTL and the 16-bit Pulse Accumulator Count register PACN32, and counts the number of external events detected during a period of time. Input capture can be used to determine the **period** of a periodic waveform, whereas pulse accumulation can be used to determine the **frequency** of a periodic waveform. Of course, there are a lot of other applications for these, too, such as determining revolutions per minute (RPM) of a rotating shaft or the time between two external events such as the time between front tires and back tires of a car crossing a sensor to determine its speed.

Timer Initialization

To initialize the timer, we need to do the following:

1. Enable the timer module.
2. Determine the rate at which we want the timer to count up, based on a prescaler from the main bus clock.
3. Set up the timer to operate in "Output Compare" mode for one of the 8 channels – for now, we'll use IOC0.
4. Connect an output signal to an external pin available on the microcontroller kit.
5. Clear the Output Compare Flag so the timer is ready to announce the first Output Compare instance.

Here are the registers we'll be working with in the initialization for Output Compare on Channel IOC0:

TSCR1	R	TEN	TSWAI	TSFRZ	TFFCA	PRNT	0	0	0
	W								
TSCR2	R	TOI	0	0	0	TCRE	PR2	PR1	PR0
	W								
TIOS	R	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0
	W								
TCTL1	R	OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4
	W								
TCTL2	R	OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0
	W								
TCNT (High)	R	TCNT15	TCNT14	TCNT13	TCNT12	TCNT11	TCNT10	TCNT9	TCNT8
	W								
TCNT (Low)	R	TCNT7	TCNT6	TCNT5	TCNT4	TCNT3	TCNT2	TCNT1	TCNT0
	W								
TC0 (High)	R	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
	W								
TC0 (Low)	R	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	W								
TFLG1	R	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F
	W								

For our basic initialization routine, we want to do the following:

1. Turn on the timer.
2. Set the prescaler so that the timer's count interval is 8 μs (to begin with).
3. Set up Channel 0 for Output Compare.
4. Connect Channel 0 to its corresponding pin in PortT, PT0, in Toggle Mode.
5. Clear the capture flag for Channel 0.

In your "Misc_Lib.c" library, begin a function to match the following header which should be in the corresponding "Misc_Lib.h" file:

void TimInit8us(void);

1. Turn on the Timer using TSCR1, but don't mess up the contents of the other bits in this register. (You'll need to OR or AND the contents of the register for this).
2. In TSCR2, you'll notice that there are only three bits available for the prescaler. These bits represent the power of 2 that you want to divide the clock frequency by. Your options, then, are 1, 2, 4, 8, 16, 32, 64, or 128. Dividing the clock frequency means the same as multiplying the period, so this is also the power of 2 that you multiply the 125 ns clock period by. So, for the desired 8 µs, we want:
$$\frac{8\mu s}{125ns} = 64 = 2^6$$

   …so the prescaler should be 6, or 110_2.

   You will need to create this prescaler without messing up the other bits, so "OR" in the 1s, and "AND" out the 0s.

   Alternatively, you could store the entire thing temporarily, clear out the prescaler bits in the temporary copy, write the desired values into the prescaler, then OR the original bits back into the register. You'll probably find the first way of doing this to be the easier of the two.
3. Set up TIOS to enable Output Compares on Channel 0. Consistent with other operations in this microcontroller, a 1 represents output and a 0 represents input. Again, don't mess up the other bits in this register.
4. Now to connect Channel 0's Output Compare events to a pin that we can monitor with an oscilloscope. Notice that the TCTL register is 16-bit, even though there are only 8 channels. That's because each channel has four options available to it, hence the need for 2 bits. Here's the table that explains this operation:

Table 7-10. Compare Result Output Action

OMx	OLx	Action
0	0	Timer disconnected from output pin logic
0	1	Toggle OCx output line
1	0	Clear OCx output line to zero
1	1	Set OCx output line to one

   We want to toggle PT0 in response to events on Channel 0, so we want to use 01_2 for the bottom two bits of TCTL2. Again, we don't want to meddle with the settings of the other bits in TCTL2, so this will require AND-ing and OR-ing.
5. Finally, we want to clear the interrupt flag associated with Channel 0. For many peripherals, a flag is "cleared" by "setting" it. In other words, to make a flag go to 0, you write a 1 to it. Do this to the appropriate bit in TFLG1 using OR.

If you followed the steps above, you should have ended up with a function that looks very much like the following:

```
void TimInit8us(void)
{
    TSCR1 |= 0b10000000;    //enable timer module
    TSCR2 &= 0b11111000;    //set prescale to Bus/64 (8 us per tick)
    TSCR2 |= 0b00000110;    //...continued
    TIOS  |= 0b00000001;    //set IOS0 to output compare
    TCTL2 &= 0b11111100;    //set PT0 to toggle mode
    TCTL2 |= 0b00000001;    //...continued
    TFLG1 |= 0b00000001;    //clear flag
}
```

Hopefully, you took the time to go through the preceding steps instead of looking ahead and just putting this code into your library, because in doing so, you will have learned quite a bit about the operation of the timer.

At this point, you should be able to use an oscilloscope to see if your initialization routine worked, because PT0 will be changing states once for each complete cycle of the count in TCNT.

Why? Here's a quick explanation. A register called 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. So, without us manipulating the value in TC0, the board will produce a square wave with a period of 1.048576 s, or a frequency of 0.954 Hz. Check to see if that's what's happening at PT0, by observing the signal at Pin 9 of the microcontroller using your oscilloscope.

We will typically manipulate the amount of time between output compare events by changing the contents of TC0 for every desired time period.

If your 8 µs initialization routine worked, add an initialization routine called "TimInit1us". This should be very similar to the one you've just finished, except for the timing. Make appropriate changes to match the label. This one should produce a free-running frequency of 7.63 Hz.

Add another routine called "TimInit125ns". This one should free-run at about 61 Hz.

Setting the Timer Compare Event Duration

Every time TCNT matches TC0, the corresponding flag is set in TFLG1. If you want to know when an output compare event has occurred, poll for the event flag in the TFLG1 register. Another way to handle this, which we will soon investigate, is to enable interrupts and allow the timer to interrupt the main program whenever the flag is set.

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. Alternatively, you could use timer interrupts, which we will soon address.

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. With a delay, the amount of time required to set up the delay and access it would be additional to the time spent waiting for the timer, but, unless the delay is extremely short, this is probably insignificant.

Intervals

First, 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 current clock value, our 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. Since interrupts and interval timing work so well together, we'll leave a full discussion of interval timing until later in the course.

Delay Function for Misc_Lib

In your "Misc_Lib.h" header file, you will find the following prototype:

```
void Sleep_ms(unsigned int);  //requires TimInit8us setup; blocking delay
```

This function is appropriately called "Sleep", because it will block and wait until the time interval has elapsed – nothing else, other than interrupt-driven behaviour, will happen once this function has been called, until the timer runs out. Unfortunately, the micro doesn't actually go into low power "sleep" mode, because it's madly checking the timer module!

Here's some information necessary to completing this function:

- 1 ms represents 125 8 µs timer ticks.
- Since we don't know what value is present in the counter (TCNT) when we access this routine, we will need to read that and add one millisecond (125 counts) to it for the first interval.
- Once the first interval is set up, we can enter a *for* loop to handle the rest of the milliseconds, in which we will clear the flag, wait for it to trigger, and add 125 to the previous target present in the timer compare register (TC0).

Again, you can check your work against the following solution:

```
void Sleep_ms(unsigned int iTime)   //requires TimInit8us()
{
    unsigned int iCount;
    TC0=TCNT+125;                   //first target -- 125 counts at 8 us = 1 ms
    for (iCount = 1;iCount<=iTime;iCount++)
    {
        TFLG1 |= 0b00000001;        //clear flag
        while((TFLG1&0b00000001)==0);   //BLOCKING wait for flag
        TC0+=125;                   //next target -- based on previous target for accuracy
    }
}
```

A good test of this routine would be to set up the program to toggle all three LEDs on the board after a delay. With an oscilloscope, you could probe the control line for one of the LEDs to see if the timing is what you expect it to be. Try values like 1 ms, 2 ms, 10 ms, 100 ms, 1000 ms, and 65000 ms (if you can wait that long!). Your results should be accurate to within the limitations of your oscilloscope.

Notice in the Sleep_ms() routine that the first value for TC0 comes from TCNT, because when you enter a Delay, you don't know what the starting time is. Consequently, there will be a small period of time lost between when you call this delay and when it actually starts, so repetitive calls of this routine will run a bit more slowly than a true interval timer.

However, notice that inside the loop, the subsequent values are based on the previous contents of TC0. That makes these true Intervals, because they are not affected by the processing time for managing the loop.

Interrupt-Driven Timer

Having an interval timer running on interrupts is a great idea. Your program can carry out any number of tasks, either dependent on or independent of the timer, without needing to poll the timer module to see if the interval is over.

The following example creates an interrupt-driven timer on Timer Channel 0. For an S12XCPU Assembly Language version of this code, look back to the code following the introduction to interrupts for the SCI receive operation.

Here are steps 1 to 5, written into the part of the skeleton file dedicated to ISRs:

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

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

Notice the declaration statement in the top line: "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.

```
      TIE |= 0b00000001;      /*enable channel 0 interrupts*/
      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.

The following endless loop watches for the press of the MID switch, one check per timer cycle, as established in the ISR on the previous page:

```
      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. Unlike an endless loop, the WAI command actually puts the microcontroller into a low-power state, so it conserves energy, which is particularly important for a battery-operated application.

The following page shows the code snippets discussed above all together as a single printout.

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

#include "SW_LED_Lib.h"


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


void    TimerSetup(int iTimeVal,byte bPrescaler);

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

volatile int    iTimeVal = 62500;
byte            bPrescaler = 0b00000101;

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


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

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

    SW_LED_Init();

    TimerSetup(iTimeVal,bPrescaler);
    EnableInterrupts;

    for (;;)
    {
        //main program loop

        if(SwCk()==0b00000001) PT1AD1&=0b00011111;  /*clear LEDs on MID press*/
        else PT1AD1 += 0b00100000;                  /*count up on the LEDs*/

        asm WAI;
    }

}



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

void TimerSetup(int iTimeVal,byte bPrescaler)
{
    TSCR1 = 0b10000000;             /*turn on timer module*/
    TSCR2 &= 0b11111000;            /*start by clearing the prescaler bits*/
    TSCR2 |= bPrescaler;            /*now set the prescaler*/
    TIOS |= 0b00000001;             /*IOS0 set to output compare*/
    TCTL2 &= 0b11111100;            /*start by clearing bits for output to PT0*/
    TCTL2 |= 0b00000001;            /*set low bit for TC0 for toggle (01) on PT0*/
    TC0 = TCNT + iTimeVal;          /*first interval set up*/
    TIE |= 0b00000001;              /*enable channel 0 interrupts*/
    TFLG1 |= 0b00000001;            /*clear TC0 flag*/
}

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

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

Real-Time Loop

(Optional topic) 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 yet – wait until the end of the semester, when you have more peripherals to work with. 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, shown first in S12XCPU Assembly Language, then in ANSI C.

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

      WAI

      BRA         RTL

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

```
for(;;)
{
      SevSegTask();
      SecTask();
      ADCDACTask();
      VoutTask();

      asm WAI;
}

//*************************************************************
//*         Timer Interrupt Service Request                   *
//*************************************************************
interrupt VectorNumber_Vtimch0 void TimerInterval(void)
{
  TC0 += 1250;                          // next time
  TFLG1 |= 0b00000001;                  // acknowledge the interrupt
}
```

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.

Input Capture and Pulse Accumulation

(Optional topic) 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 electrical 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.

TIOS	R	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0
	W								

This time, we want Channel 7 to be an "Input Compare", so it should be a 0.

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).

TCTL3	R	EDG7B	EDG7A	EDG6B	EDG6A	EDG5B	EDG5A	EDG4B	EDG4A
	W								
TCTL4	R	EDG3B	EDG3A	EDG2B	EDG2A	EDG1B	EDG1A	EDG0B	EDG0A
	W								

There are two bits associated with each channel, since there are four possible options.

Table 7-12. Edge Detector Circuit Configuration

EDGxB	EDGxA	Configuration
0	0	Capture disabled
0	1	Capture on rising edges only
1	0	Capture on falling edges only
1	1	Capture on any edge (rising or falling)

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.

The following page shows a code snippet that displays the period, in microseconds, of a signal connected to PT7 (pin 18 of the microcontroller).

```
    TimInit1us();
    SevSeg_Init();

    TIOS &=0b01111111;  //put channel 7 into Input Compare mode
    TCTL3&=0b01111111;  //first part of making channel 7 rising edge
    TCTL3|=0b01000000;  //second part of making channel 7 rising edge

    TFLG1|=0b10000000;  //clear the flag
    iStart=TCNT;        //first timer reading

    for (;;)        //endless program loop
    {
/*****************************************************************************
*       Main Program Code
*****************************************************************************/

        while((TFLG1&&0b10000000)==0);  //wait for rising edge on channel 7
        iDiff= TC7-iStart;
        iStart=TC7;
        SevSeg_Top4(HexToBCD(iDiff));
        TFLG1|=0b10000000;  //clear the flag
    }
```

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 the 16-bit Pulse Accumulator A, connected to PT7. Here is its control register, PACTL:

PACTL	R	0	PAEN	PAMOD	PEDGE	CLK1	CLK0	PA0VI	PAI
	W								

Table 7-18. PACTL Field Descriptions

Field	Description
6 PAEN	**Pulse Accumulator A System Enable** — PAEN is independent from TEN. With timer disabled, the pulse accumulator can still function unless pulse accumulator is disabled. 0 16-Bit Pulse Accumulator A system disabled. 8-bit PAC3 and PAC2 can be enabled when their related enable bits in ICPAR are set. Pulse Accumulator Input Edge Flag (PAIF) function is disabled. 1 16-Bit Pulse Accumulator A system enabled. The two 8-bit pulse accumulators PAC3 and PAC2 are cascaded to form the PACA 16-bit pulse accumulator. When PACA in enabled, the PACN3 and PACN2 registers contents are respectively the high and low byte of the PACA. PA3EN and PA2EN control bits in ICPAR have no effect. Pulse Accumulator Input Edge Flag (PAIF) function is enabled. The PACA shares the input pin with IC7.
5 PAMOD	**Pulse Accumulator Mode** — This bit is active only when the Pulse Accumulator A is enabled (PAEN = 1). 0 Event counter mode 1 Gated time accumulation mode
4 PEDGE	**Pulse Accumulator Edge Control** — This bit is active only when the Pulse Accumulator A is enabled (PAEN = 1). Refer to Table 7-19. For PAMOD bit = 0 (event counter mode). 0 Falling edges on PT7 pin cause the count to be incremented 1 Rising edges on PT7 pin cause the count to be incremented For PAMOD bit = 1 (gated time accumulation mode). 0 PT7 input pin high enables bus clock divided by 64 to Pulse Accumulator and the trailing falling edge on PT7 sets the PAIF flag. 1 PT7 input pin low enables bus clock divided by 64 to Pulse Accumulator and the trailing rising edge on PT7 sets the PAIF flag. If the timer is not active (TEN = 0 in TSCR1), there is no divide-by-64 since the ÷64 clock is generated by the timer prescaler.
3:2 CLK[1:0]	**Clock Select Bits** — For the description of PACLK please refer to Figure 7-70. If the pulse accumulator is disabled (PAEN = 0), the prescaler clock from the timer is always used as an input clock to the timer counter. The change from one selected clock to the other happens immediately after these bits are written. Refer to Table 7-20.
2 PAOVI	**Pulse Accumulator A Overflow Interrupt Enable** 0 Interrupt inhibited 1 Interrupt requested if PAOVF is set
0 PAI	**Pulse Accumulator Input Interrupt Enable** 0 Interrupt inhibited 1 Interrupt requested if PAIF is set

We will need to enable Pulse Accumulator A. We also typically need to tell it that we're going to use it as an Event Counter. We also need to indicate whether it will respond to rising edges or falling edges, and how we will use the clock source.

Table 7-19. Pin Action

PAMOD	PEDGE	Pin Action
0	0	Falling edge
0	1	Rising edge
1	0	Divide by 64 clock enabled with pin high level
1	1	Divide by 64 clock enabled with pin low level

Table 7-20. Clock Selection

CLK1	CLK0	Clock Source
0	0	Use timer prescaler clock as timer counter clock
0	1	Use PACLK as input to timer counter clock
1	0	Use PACLK/256 as timer counter clock frequency
1	1	Use PACLK/65536 as timer counter clock frequency

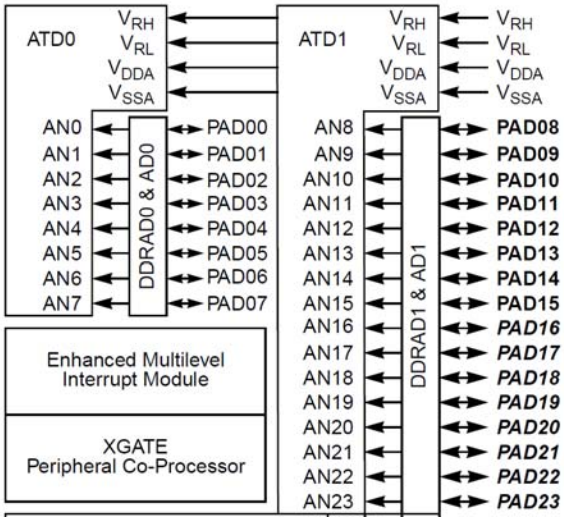
It's fairly obvious that this pulse accumulator can be used in a lot of different ways. We will just use it in its simplest mode: responding to a rising edge, using the timer prescaler as the counter clock. At this point, we aren't using any interrupts, so we'll inhibit the two interrupts. Hopefully, you've determined that we want to put the value 0b01010000 into PACTL (the first bit isn't used, and is always 0).

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. Here's a bit of code that counts events on PT7 (pin 18) for a full second, then displays the frequency, in Hz, on the seven segment display.

```
/*****************************************************************
*       Initializations
*****************************************************************/
    SevSeg_Init();
    TimInit8us();

    PACTL=0b01010000;   //rising edge event counter using timer prescaler

    for (;;)        //endless program loop
    {
/*****************************************************************
*       Main Program Code
*****************************************************************/

    PACN32=0;       //clear counter
    Sleep_ms(1000);
    SevSeg_Top4(HexToBCD(PACN32));

    }
```

A To D Conversion

The 9S12XDP512 has two fairly complex Analog to Digital Conversion blocks. Here's the top left corner of the block diagram, which we've looked at previously.



Notice that PAD00 to PAD07 are associated with ATD0, and PAD08 to PAD15, the pins that are connected on our board to the LEDs and Switches, could also be attached to ATD1. So, of the two available converters, we're going to be using ATD0.

This peripheral is highly configurable, and can do a lot of things. Again, we'll just scratch the surface of its capabilities.

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 8 external voltages simultaneously.
- It can sample on command 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.
- You can ask for multiple samples from one channel, one sample from each of the channels, or a number of samples from a number of channels (within limits!).
- You can choose where the results end up, since the result registers aren't directly tied to the input channels. In one mode (FIFO) it continuously wraps through the channels placing the next available value in the next output register; in another mode, it always starts at result channel 0 and runs until it gets to the last result you've asked it for; it can also start filling at a result register of your choice, wrapping around until it gets to the last result you've asked for.
- 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 unsigned or signed values (Single quadrant or 4-quadrant), and left or right justified values.

The block diagram for ATD0 shows some of the capabilities of this converter, along with the internal devices that make possible these capabilities.

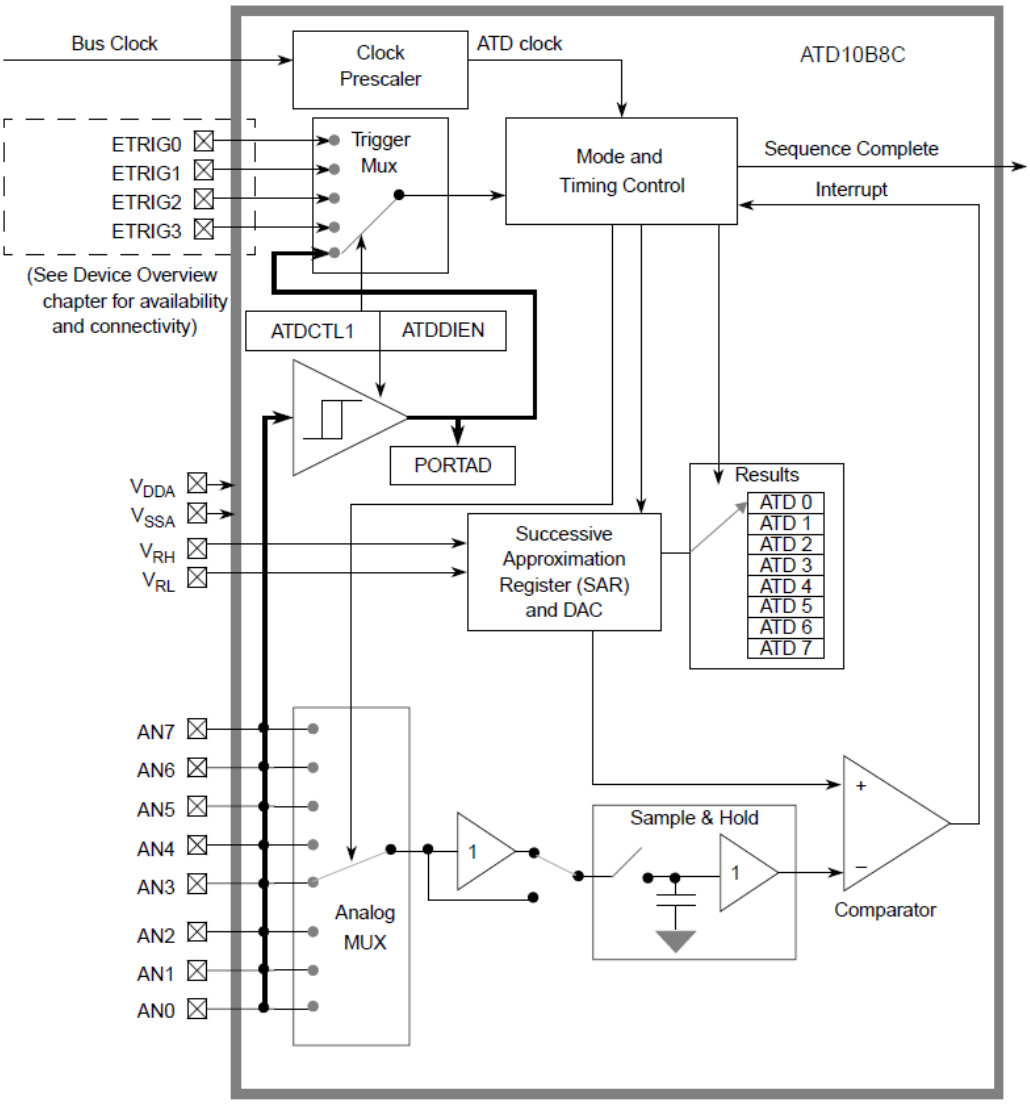


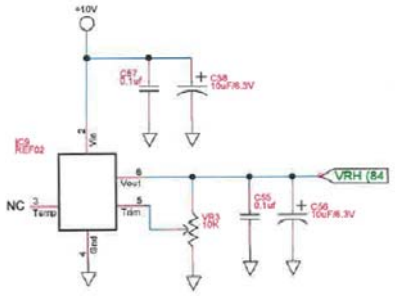
Figure 5-1. ATD Block Diagram

Setting up V_{RH}

In other courses, you've come to think of the REF02 integrated circuit as a fixed 5.0 V reference. The designers of the IC knew that it would be very difficult to make all of their products produce exactly 5.0 V, so they included a *TRIM* pin that can be used to adjust the output slightly. We have taken advantage of this feature, and have incorporated an output trimming circuit that can be used to set the REF02 output to 5.120 V_{DC}, which gives us a useful step size. With a 10-bit A to D converter, the step size is

$$StepSize = \frac{V_{ref}}{2^n} = \frac{5.120}{2^{10}} = 5.000mV/step$$

Here's the circuitry involved. The capacitors are there to filter out noise and variations in the power supplies. What's important to our discussion is *VR3*, the trim potentiometer.



On your board, you will need to connect a digital voltmeter between ground and *VREF*, and adjust *VR3* to set *VREF* to 5.120 V_{DC}.

Configuring ATD0

The full details for the registers we're using are found in Chapter 5 of the "Data Sheet". Here's a summary that includes the ones we'll be using.

Register Name		Bit 7	6	5	4	3	2	1	Bit 0
ATDCTL0	R	0	0	0	0	0	WRAP2	WRAP1	WRAP0
	W								
ATDCTL1	R	ETRIGSEL	0	0	0	0	ETRIGCH2	ETRIGCH1	ETRIGCH0
	W								
ATDCTL2	R	ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIGE	ASCIE	ASCIF
	W								
ATDCTL3	R	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0
	W								
ATDCTL4	R	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0
	W								
ATDCTL5	R	DJM	DSGN	SCAN	MULT	0	CC	CB	CA
	W								
ATDSTAT0	R	SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0
	W								
ATDDR0H	10-BIT	0	0	0	0	0	0	BIT 9 MSB	BIT 8
	8-BIT	0	0	0	0	0	0	0	0
	W								
ATDDR0L	10-BIT	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
	8-BIT	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
	W								
ATDDIEN	R	IEN7	IEN6	IEN5	IEN4	IEN3	IEN2	IEN1	IEN0
	W								

In the datasheet, the registers are named "ATDCTLx", etc. But since there are two A to D converters in this controller, we need to insert a "0" after "ATD" in each case to specify that we're using ATD0 (not ATD1, which is connected to the switches and LEDs).

Remember DDR1AD1 and ATD1DIEN1? We needed to digitally-enable the inputs in order to get a digital signal into them. However, to receive analog signals, we need to have DDRAD and ATDDIEN or, in our case, DDR1AD0 and ATD0DIEN, cleared to LOW for the pins to be enabled as analog inputs instead. Since the unit defaults to 0, we shouldn't have to worry about DDR1AD0 and ATD0DIEN, unless somewhere else in software we've set these bits to "1". In any case, it's good practice to make sure the pins are configured correctly.

Your instructor will probably want you to create a library for A to D conversion, called "ATD0_Lib.c", with its prototype header file "ATD0_Lib.h". In this library, the first entry should be an initialization routine, ATD0_Init().

Notice in the list of registers that there are six control registers: ATDCTL0 through ATDCTL5. The first two are for modes we're not going to use. So, let's start with ATD0CTL2 (notice the "0" inserted in the register name). You should probably refer to the register descriptions in Chapter 5 of the "Data Sheet" to determine what each bit should be in the following registers, if they aren't obvious.

- For ATD0CTL2, 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.
- In the "Data Sheet", it says that, once ATD0CTL2 has been set up and the A to D Converter has been powered up, we need to wait at least 50 µs before anything else can be configured. You probably don't want to be forced to include your Misc_Lib every time you run the A to D converter, so it makes sense to create a simple delay using a clock-cycle-based loop. We've done this before using "asm" commands to: 1) load an accumulator with a desired number of loops, then 2) execute the "DBNE" Assembly Language command (three clock cycles) until the counter runs out.
- For ATD0CTL3, we want 8 conversions per sequence, we want the converter to start at our selected register (which we will soon set to 0) rather than being "first-in first-out", and we want the A to D converter to finish conversions before freezing on a break. This one probably requires a look at the description in the "Data Sheet":

Table 5-8. Conversion Sequence Length Coding

S8C	S4C	S2C	S1C	Number of Conversions per Sequence
0	0	0	0	8
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	X	X	X	8

Table 5-9. ATD Behavior in Freeze Mode (Breakpoint)

FRZ1	FRZ0	Behavior in Freeze Mode
0	0	Continue conversion
0	1	Reserved
1	0	Finish current conversion, then freeze
1	1	Freeze Immediately

(continued)

- For ATD0CTL4, 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. This one definitely needs some information from the "Data Sheet":

Table 5-10. ATDCTL4 Field Descriptions

Field	Description
7 SRES8	**A/D Resolution Select** — This bit selects the resolution of A/D conversion results as either 8 or 10 bits. The A/D converter has an accuracy of 10 bits; however, if low resolution is required, the conversion can be speeded up by selecting 8-bit resolution. 0 10-bit resolution 8-bit resolution
6–5 SMP[1:0]	**Sample Time Select** — These two bits select the length of the second phase of the sample time in units of ATD conversion clock cycles. Note that the ATD conversion clock period is itself a function of the prescaler value (bits PRS4–0). The sample time consists of two phases. The first phase is two ATD conversion clock cycles long and transfers the sample quickly (via the buffer amplifier) onto the A/D machine's storage node. The second phase attaches the external analog signal directly to the storage node for final charging and high accuracy. Table 5-11 lists the lengths available for the second sample phase.
4–0 PRS[4:0]	**ATD Clock Prescaler —** These 5 bits are the binary value prescaler value PRS. The ATD conversion clock frequency is calculated as follows: $ATDclock = \frac{[BusClock]}{[PRS + 1]} \times 0.5$ **Note:** The maximum ATD conversion clock frequency is half the bus clock. The default (after reset) prescaler value is 5 which results in a default ATD conversion clock frequency that is bus clock divided by 12. Table 5-12 illustrates the divide-by operation and the appropriate range of the bus clock.

Table 5-11. Sample Time Select

SMP1	SMP0	Length of 2nd Phase of Sample Time
0	0	2 A/D conversion clock periods
0	1	4 A/D conversion clock periods
1	0	8 A/D conversion clock periods
1	1	16 A/D conversion clock periods

Table 5-12. Clock Prescaler Values

Prescale Value	Total Divisor Value	Max. Bus Clock[1]	Min. Bus Clock[2]
00000	Divide by 2	4 MHz	1 MI lz
00001	Divide by 4	8 MHz	2 MHz
00010	Divide by 6	12 MHz	3 MHz
00011	Divide by 8	16 MHz	4 MHz
00100	Divide by 10	20 MHz	5 MHz
00101	Divide by 12	24 MHz	6 MHz
00110	Divide by 14	28 MHz	7 MHz
00111	Divide by 16	32 MHz	8 MHz
01000	Divide by 18	36 MHz	9 MHz
01001	Divide by 20	40 MHz	10 MHz
01010	Divide by 22	44 MHz	11 MHz
01011	Divide by 24	48 MHz	12 MHz
01100	Divide by 26	52 MHz	13 MHz
01101	Divide by 28	56 MHz	14 MHz
01110	Divide by 30	60 MHz	15 MHz
01111	Divide by 32	64 MHz	16 MHz
10000	Divide by 34	68 MHz	17 MHz
10001	Divide by 36	72 MHz	18 MHz
10010	Divide by 38	76 MHz	19 MHz
10011	Divide by 40	80 MHz	20 MHz
10100	Divide by 42	84 MHz	21 MHz
10101	Divide by 44	88 MHz	22 MHz
10110	Divide by 46	92 MHz	23 MHz
10111	Divide by 48	96 MHz	24 MHz
11000	Divide by 50	100 MHz	25 MHz
11001	Divide by 52	104 MHz	26 MHz
11010	Divide by 54	108 MHz	27 MHz
11011	Divide by 56	112 MHz	28 MHz
11100	Divide by 58	116 MHz	29 MHz
11101	Divide by 60	120 MHz	30 MHz
11110	Divide by 62	124 MHz	31 MHz
11111	Divide by 64	128 MHz	32 MHz

[1] Maximum ATD conversion clock frequency is 2 MHz. The maximum allowed bus clock frequency is shown in this column.

[2] Minimum ATD conversion clock frequency is 500 kHz. The minimum allowed bus clock frequency is shown in this column.

- For ATD0CTL5, we want our results to be right-justified unsigned values, we'll run in continuous scan mode sampling all eight channels, and we want to have the results of each conversion sequence start at result register 0 so that the result register will match the input channel.

If you've correctly interpreted the information on the previous page, you should have ended up with something like the following:

```
//ATD0 Library Files
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//December 2014

#include <hidef.h>
#include "derivative.h"
#include "ATD0_Lib.h"


void ATD0_Init(void)
{
    DDR1AD0  = 0b00000000;  //enable all channels as inputs
    ATD0DIEN = 0b00000000;  //ensure they are Analog

    ATD0CTL2 = 0b11100000;
/*                  ||||||||
                    ||||||| --interrupt flag (input - don't care)
                    |||||| ---interrupt disabled
                    ||||| ----external trigger disabled
                    |||| -----external trigger polarity (don't care)
                    ||| ------external trigger interrupt disabled
                    || -------ATD continues in wait mode
                    | --------fast flag -- clears on read
                     ---------ATD power up
*/

    asm LDX #134;           //need a 50 us delay before continuing
    asm DBNE X,*;

    ATD0CTL3 = 0b00000010;
/*                  ||||||||
                    ||||||| --\
                    |||||| ---/ finish conversion before freezing
                    ||||| ----result maps into corresponding register
                    |||| -----\
                    ||| ------ \
                    || ------- / 8 conversions per sequence
                    | --------/
                     ---------dont care
*/

    ATD0CTL4 = 0b00100110;
/*                  ||||||||
                    ||||||| --\
                    |||||| --- \
                    ||||| ----  \ bus clock divide by 14 = 571.4 kHz (1.75 us period)
                    |||| -----  /  with 4 clocks per sample, this is 7 us per sample
                    ||| ------/   which is the required minimum
                    || -------\
                    | --------/ 4 A/D conversion clock periods per sample
                     ---------10-bit resolution
*/
    ATD0CTL5 = 0b10110000;
/*                  ||||||||
                    ||||||| --\  starts filling the result registers
                    |||||| --- / at the bottom, mapping Channelx to DRx
                    ||||| ----/
                    |||| -----(don't care)
                    ||| ------sample all 8 channels
                    || -------Continuous scan conversion
                    | --------unsigned (single quadrant)
                     ---------right-justified
*/
}
```

Using ATD0

Now that the A to D converter has been initialized, we want to use it. For this, we'll need to connect an appropriate signal to the channel of choice, and run a routine that makes the A to D converter go through a conversion sequence, then reads the resulting digital value. The minimum requirement for this course is to perform A to D conversions using AN0, which, as you should verify from the schematic, is connected to Pin 67 through a 1 kΩ resistor to minimize the chance of damaging this input pin with an incorrect input signal. On your board, you should be able to see the eight resistors used to protect channels AN0 through AN7, and you should see a pin soldered into the header for Pin 67.

In your ATD_Lib, you will need to write a simple routine called "AtoD_AN0". It will wait for the SCF flag (Sequence Complete Flag) which is b7 of ATD0STAT0, then it will read the appropriate Data Register (in this case, ATD0DR0). Remember that we're doing 10-bit conversions, so this result will have to be read as a 16-bit (int) value. Also, we've chosen single-quadrant operation, so the result will always be positive (unsigned).

You will need to write a test routine that reads the input voltage and displays it somewhere, probably the seven-segment display for simplicity. To verify your results, though, you will need to write code to do the math and value manipulation necessary to convert the result to an actual voltage, which you can verify with a digital multi-meter (DMM).

Pulse-Width Modulation

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 9S12X has a highly-configurable eight-channel PWM module. We will only begin to scratch the surface of the capabilities of this module.

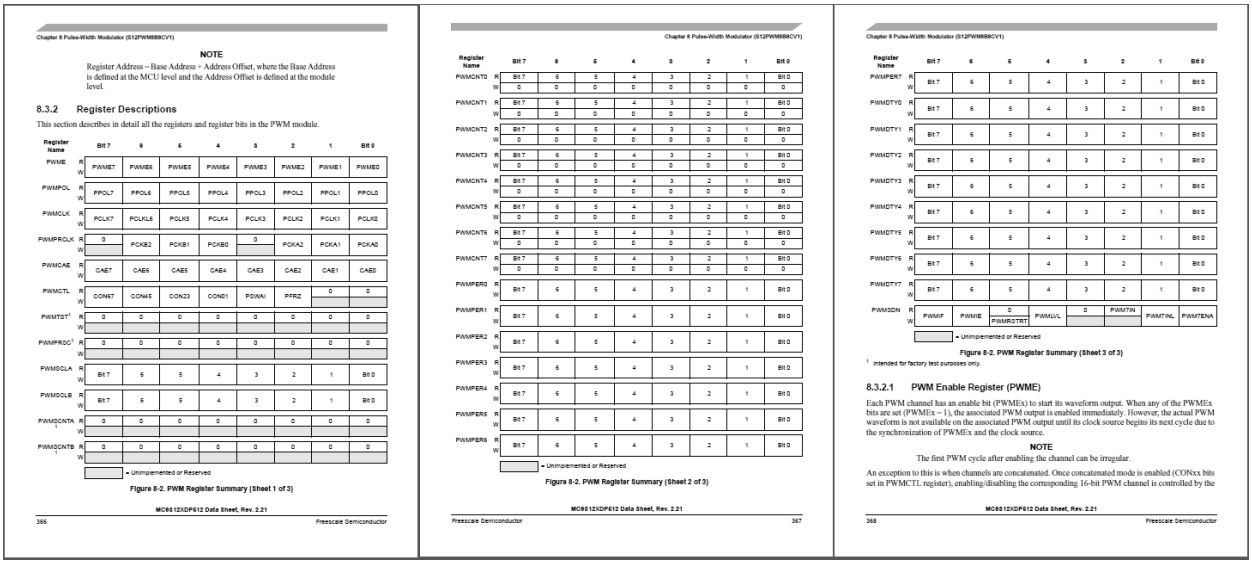
The PWM module is used to create waveforms with programmable period and duty. There are a number of uses for programmable waveforms, most residing well outside the scope of this course.

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 9S12X, the PWM subsystem is configured through a series of registers, shown in a screen capture on the next page to give you a sense of the complexity of this module. We’ll learn about the registers of interest to you as you need them.

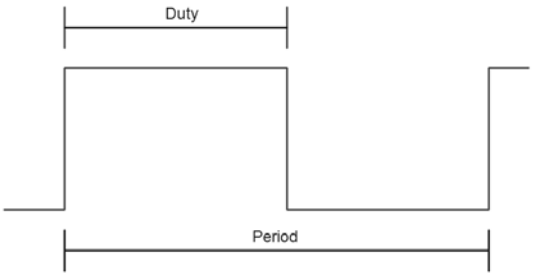


There are eight fairly independent PWM channels in the MC9S12XDP512 – "fairly" independent, because you can control a lot of characteristics independently; however, the clocks, although highly configurable, are shared by four channels each, which can be challenging if you want to control devices requiring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

Generating Waveforms

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.



The ***PWMPOL*** register, which determines the polarity of the pulse, is shown below.

PWMPOL								
	R	PPOL7	PPOL6	PPOL5	PPOL4	PPOL3	PPOL2	PPOL1	PPOL0
	W								

Strangely, the default condition of each PWM channel is negative or inverted polarity. For most of our applications, we want to ***set*** the bits in PWMPOL to make them normal polarity.

The next characteristics of the waveform all relate to timing. You've learned in other courses that the most basic aspects of waveform timing are frequency and its inverse, the period. The timing characteristics are based on clocks internal to the PWM module.

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.

PWMCLK									
	R	PCLK7	PCLKL6	PCLK5	PCLK4	PCLK3	PCLK2	PCLK1	PCLK0
	W								

Table 8-3. PWMCLK Field Descriptions

Field	Description	Field	Description
7 PCLK7	**Pulse Width Channel 7 Clock Select** 0 Clock B is the clock source for PWM channel 7. 1 Clock SB is the clock source for PWM channel 7.	3 PCLK3	**Pulse Width Channel 3 Clock Select** 0 Clock B is the clock source for PWM channel 3. 1 Clock SB is the clock source for PWM channel 3.
6 PCLK6	**Pulse Width Channel 6 Clock Select** 0 Clock B is the clock source for PWM channel 6. 1 Clock SB is the clock source for PWM channel 6.	2 PCLK2	**Pulse Width Channel 2 Clock Select** 0 Clock B is the clock source for PWM channel 2. 1 Clock SB is the clock source for PWM channel 2.
5 PCLK5	**Pulse Width Channel 5 Clock Select** 0 Clock A is the clock source for PWM channel 5. 1 Clock SA is the clock source for PWM channel 5.	1 PCLK1	**Pulse Width Channel 1 Clock Select** 0 Clock A is the clock source for PWM channel 1. 1 Clock SA is the clock source for PWM channel 1.
4 PCLK4	**Pulse Width Channel 4 Clock Select** 0 Clock A is the clock source for PWM channel 4. 1 Clock SA is the clock source for PWM channel 4.	0 PCLK0	**Pulse Width Channel 0 Clock Select** 0 Clock A is the clock source for PWM channel 0. 1 Clock SA is the clock source for PWM channel 0.

It's important to note which basic clock is being used, A or B, and that '0' selects the basic clock whereas '1' selects the prescaled version of that basic clock.

The next register to configure is ***PWMPRCLK***. This register determines how clock A and/or clock B is divided down from the bus clock. PWMPRCLK affects the basic clock speeds, and so affects the frequency of all waveforms produced by the PWM module. Notice that both prescalers are contained in the same register, but at different bit locations. Be careful with this! Also, notice that there are only three bits associated with each prescaler. That's because these bits represent the power of 2 for the prescaler. We'll refer to the prescaler as 2^{PREx}, where 'x' is either A or B.

PWMPRCLK									
	R	0	PCKB2	PCKB1	PCKB0	0	PCKA2	PCKA1	PCKA0
	W								

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. We'll refer to these as PWMSCLx, where 'x' is either A or B. This provides the scaling factor, allowing for fine control over the settings chosen for the A or B clock previously. The information for PWMSCLA has been provided; PWMSCLB behaves identically, but for clock SB.

PWMSCLA divides down Clock A to generate Clock SA, and PWMSCLB divides down Clock B to generate Clock SB.

8.3.2.9 PWM Scale A Register (PWMSCLA)

PWMSCLA is the programmable scale value used in scaling clock A to generate clock SA. Clock SA is generated by taking clock A, dividing it by the value in the PWMSCLA register and dividing that by two.

Clock SA = Clock A / (2 * PWMSCLA)

NOTE

When PWMSCLA = $00, PWMSCLA value is considered a full scale value of 256. Clock A is thus divided by 512.

Any value written to this register will cause the scale counter to load the new scale value (PWMSCLA).

	7	6	5	4	3	2	1	0
R W	Bit 7	6	5	4	3	2	1	Bit 0
Reset	0	0	0	0	0	0	0	0

Figure 8-11. PWM Scale A Register (PWMSCLA)

Read: Anytime

Write: Anytime (causes the scale counter to load the PWMSCLA value)

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 0x00 means 256. This does not seem to be true. The biggest divisor available seems to be 2 x 255 = 510.

In addition to the clock rates chosen, the waveforms you will be generating are managed by byte-sized period and duty values, controlled using two more registers: ***PWMPERn*** and ***PWMDTYn***, where "n" is the number of our selected channel. Channel 0 is shown below:

		Bit 7	6	5	4	3	2	1	Bit 0
PWMPER0	R W								
PWMDTY0	R W								

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. For this shortest period, the duty could only be 1, since either 0 or 2 would produce a DC signal.

Typically, the duty cycle of a signal is defined as a ratio or percentage, as shown below:

$$d = \frac{t_p}{T} \quad \text{or} \quad d = \frac{t_p}{T} \times 100\%$$

So, the duty cycle for the PWM module, as a ratio, would be

$$d = \frac{PWMDTYn}{PWMPERn}$$

A very common duty cycle is 50%, which represents a true square wave. For this, PWMDTYn would be half of PWMPERn.

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 PWMPERn, then enter the number of clock cycles (less than PWMPERn for obvious reasons) into PWMDTYn.

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

$$f = \frac{8MHz}{2^{PREx} PWMPERn} \quad \text{and} \quad T = 125ns \times 2^{PREx} PWMPERn$$

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

$$f = \frac{8MHz}{2^{PREx} \times 2 \times PWMSCLx \times PWMPERn} \quad \text{and} \quad T = 125ns \times 2^{PREx} \times 2 \times PWMSCLx \times PWMPERn$$

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

The last thing to do is actually turn on the channel, which is done by setting the corresponding bit in the ***PWME*** register:

PWME	R W	PWME7	PWME6	PWME5	PWME4	PWME3	PWME2	PWME1	PWME0

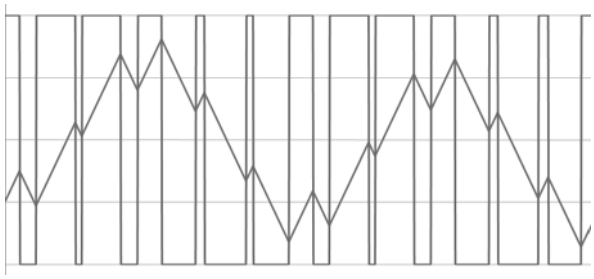
We've left discussion of this register to the end because, unlike many of the modules we've worked with to this point, we sometimes don't want to have the PWM channel turned on all the time. Manipulating this register allows you to turn your signals on and off under software control.

The code snippet on the following page turns a 1 kHz square wave, sent to the speaker, on and off once per second. If you build this code, you can monitor the signal by probing the left side of JP1 on the board with an oscilloscope. Also, with JP1 installed and VR2 turned up, you should be able to hear the tone.

```
/**************************************************************************
*       Initializations
**************************************************************************/

    TimInit8us();

    PWMPOL =0b11111111;     //positive polarity, all channels
    PWMCLK|=0b01000000;     //Use SB as the clock for channel 6 (speaker)
    /* 1 kHz = 8000000/(2^2 x 2 x 4 x 250)  */
    PWMPRCLK &=0b00001111;  //clear PRE-B before setting
    PWMPRCLK |=0b00100000;  //PRE-B = 2^2
    PWMSCLB = 4;            //Scale = 2 x 4
    PWMPER6 = 250;          //keep the period number large for accuracy
    PWMDTY6 = 125;          //50% duty cycle


    for (;;)        //endless program loop
    {
/**************************************************************************
*       Main Program Code
**************************************************************************/

    Sleep_ms(500);          //half second on, half second off
    PWME^=0b01000000;       //...by toggling PWME6

    }
```

True Pulse-Width Modulation

(Optional topic) For many applications, including motor control, 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, alternating parabolas, you name it! The figure below shows this process.

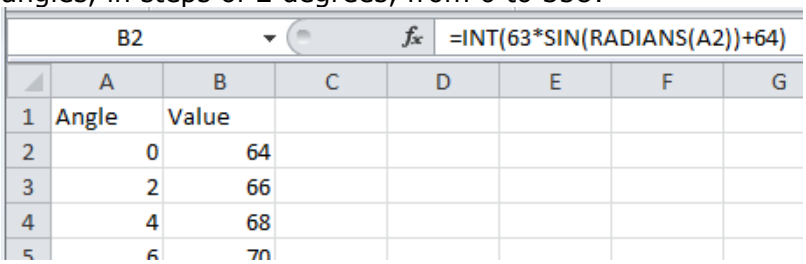


The "jagged sinusoid" shown above would be the result of a simple integration of the PWM signal on which it is juxtaposed. Note that, when the duty cycle is greater than 50%, the integrated signal rises; when the duty cycle is less than 50%, the integrated signal falls.

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.

The following page shows a sample section of code, with the resulting waveforms seen at the output of the PWM (channel 1) and after the integrator (channel 2). Note the following:

1. The values for the duty cycles in the lookup table "cWave[180]" were generated using the formula shown in the "f_x" line of the Excel screen shot below, applied to angles, in steps of 2 degrees, from 0 to 358:

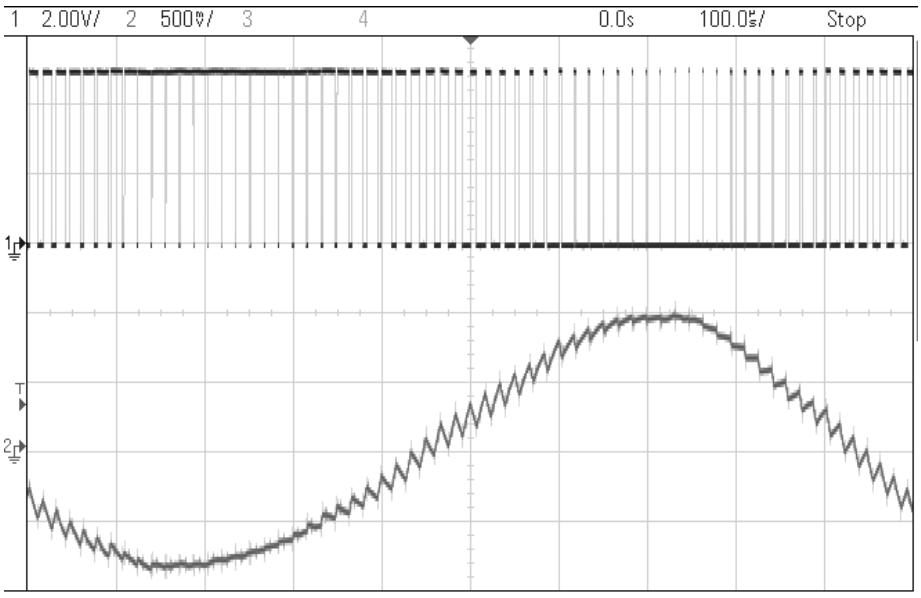


B2 | f_x | =INT(63*SIN(RADIANS(A2))+64)

	A	B	C	D	E	F	G
1	Angle	Value					
2	0	64					
3	2	66					
4	4	68					
5	6	70					

   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 9S12X microcontroller isn't set up at this time to use floating-point decimals; we don't include 360º, as that's the same as 0º in our repeating sine wave.
2. Both the PWM module and the Timer are running as fast as possible – the Bus clock.
3. The PWM module is set to run at 62.5 kHz, well above the audible range.
4. Using a timer interval of 44 counts, each 125 ns long, it takes 990 µs to run through all 180 values for the duty cycle in the table, for a frequency of 1.01 kHz on the resulting sine wave. The endless "for" loop starts at the top of the table again.
5. The PWM frequency is constant – only the duty cycle changes sinusoidally.
6. The speaker is driven by a transistor switch, which is an inverter, so the sine wave rises when the duty cycle is close to 1/128 and falls when it is close to 127/128.
7. The jagged edge of the integrator can be seen on the sine wave.
8. Channel 2 is AC coupled to block the DC offset generated by the transistor switch.
9. The speaker volume control is set to maximum to drive the integrator appropriately.

```
/*****************************************************************
*       Variables
*****************************************************************/

unsigned char cCounter;


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

unsigned char cWave[180]=
{
    64,66,68,70,72,74,77,79,81,83,85,87,89,91,93,95,
    97,99,101,102,104,106,107,109,110,112,113,114,116,117,118,119,
    120,121,122,123,123,124,125,125,126,126,126,126,126,127,126,126,
    126,126,126,125,125,124,123,123,122,121,120,119,118,117,116,114,
    113,112,110,109,107,106,104,102,101,99,97,95,93,91,89,87,
    85,83,81,79,77,74,72,70,68,66,64,61,59,57,55,53,
    50,48,46,44,42,40,38,36,34,32,30,28,26,25,23,21,
    20,18,17,15,14,13,11,10,9,8,7,6,5,4,4,3,
    2,2,1,1,1,1,1,1,1,1,1,1,2,2,3,
    4,4,5,6,7,8,9,10,11,13,14,15,17,18,20,21,
    23,25,26,28,30,32,34,36,38,40,42,44,46,48,50,53,
    55,57,59,61
};

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

/*****************************************************************
*       Initializations
*****************************************************************/

    TimInit125ns();             //set up Timer Channel 0 for bus clock speed

    PWMCLK   &=0b10111111;      //Use B Clock directly
    PWMPRCLK &=0b10001111;      //Set B clock to Bus Clock (8 MHz)
    PWMPER6 = 128;              //128 clock cycles = 62.5 kHz
    PWMDTY6 = 64;               //initially 50% duty cycle
    PWME     |=0b01000000;      //turn on channel 6 (speaker)

    TC0=TCNT+44;                //first timer interval:  44 x 180 x 125 ns = 990 us
                                //for a 1.01 kHz sine wave

    for (;;)        //endless program loop
    {
/*****************************************************************
*       Main Program Code
*****************************************************************/

        for(cCounter = 0;cCounter<180;cCounter++)   //endlessly loops through
        {                                           //the entire set of 180 values
            PWMDTY6= cWave[cCounter];               //get the current duty value
            while((TFLG1&0b00000001)==0);           //wait for the timer flag
            TC0+=44;                                //set up the next interval
            TFLG1|=0b00000001;                      //clear the flag
        }
    }
}
```



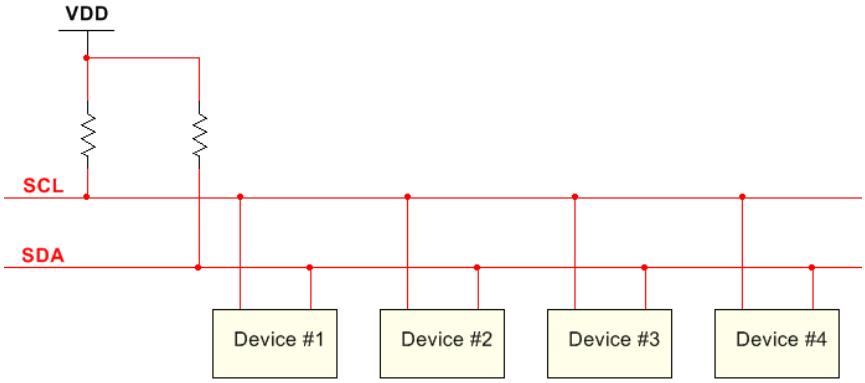
I²C Bus

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):

- M41T81 Real-time Clock (with battery backup – the battery is under the LCD display)
- 24AA512 Memory – 512kB of EEPROM
- LTC2633HZ12 Dual channel 12-bit Digital to Analog Converter (DAC)
- MPL3115A2 Precision Altimeter/Barometer/Thermometer
- LSM303DLHC Accelerometer/Compass

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 must not 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.

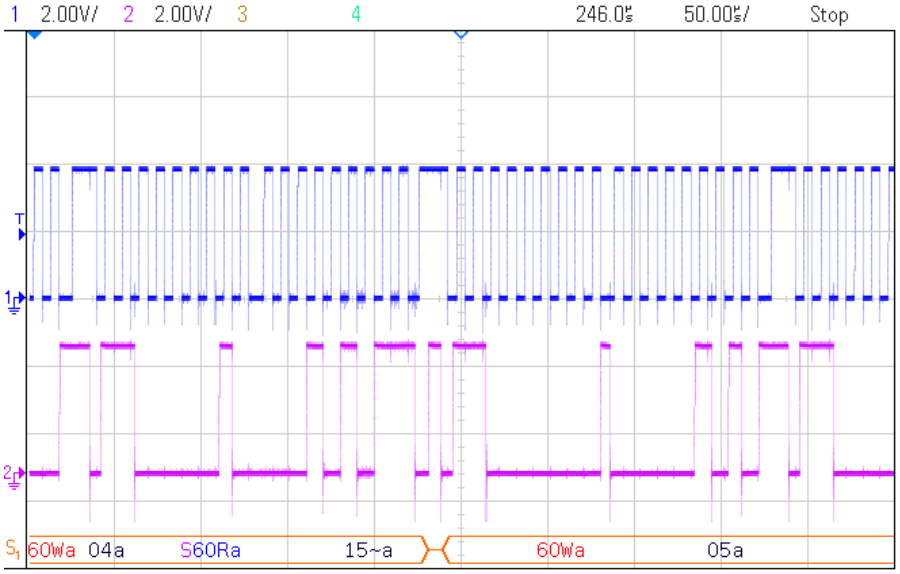
Another feature of this system is its ability to communicate at different speeds for different devices. The device talking at any point in time generates the clock, which synchronizes communication with the receiving device. The typical maximum speed for the I^2C bus is 100 kb/s, but some devices have 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, 00100xxr. 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: 0010000r, 0010001r, and 0010010r. 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, to talk to these devices, you would need to treat these three addresses as 0x20, 0x22, and 0x24. However, as is made obvious by the oscilloscope in this author's cubicle, they are officially only 7-bit addresses with the low bit missing, and should be thought of as 0x10, 0x11, and 0x12!

Here's a screen shot from the author's oscilloscope of the 9S12X talking to a device with the seven-bit address 1100000r which we would have to treat as 0xC0, but is officially 0x60.



The top trace is SCL. Notice that it starts and stops, depending on how the bus is being used. The lower trace is SDA, and contains communication both from the 9S12X, acting as a "master", and the device at address 0b1100000r, acting as a slave. The line at the bottom shows that the master told the device it was going to Write something to it "60W", sent 0x04 "04" to it, indicated a restart "S", told the device it was going to Read from it "60R", so the device put 0x15 "15"on the bus. (The "a" and "~a" are ACK and NACK handshaking tools – each master Command expects an ACK (acknowledge), but communication back from the slave is followed by a NACK (not acknowledge) instead.)

Basic I²C Communication Using the 9S12X

On our board, the 9S12X will always be the Master, and the peripherals on the board will be Slaves. Here's a table of the addresses for the available devices:

Part	***Part Number***	***I²C Address (7-bit)***	***I²C Address (with R/W as r)***
Real Time Clock	M41T81	0x68	0b1101000r
512 kbit Memory (64kB)	24AA512	0x50	0b1010000r
Dual 12-bit DAC	LTC2633HZ12	0x10	0b0010000r
Pressure/Temperature	MPL3115A2	0x60	0b1100000r
Accelerometer	LSM303DLHC	0x19	0b0011001r
Compass	LSM303DLHC	0x1E	0b0011110r

To use the bus, our micro has to do the following:

1. Check to see if the bus is available by checking the "IBB" bit (b5) of the status register. This is the "Bus Busy" bit, and is SET when the bus is being used.
2. If it is, indicate "Start" to take control of the bus. (Incidentally, "Start" involves a negative-going transition on SDA while SCL is HIGH. This is generated in our I²C controller by setting the Master and Tx bits in IBCR.)
3. Notify the desired slave by its address, while at the same time indicating, typically, that we're going to write to it.
4. Send a byte that contains the internal address of the register we want to put something into 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. Indicate a "Stop" to free up the bus for another device to take control (of course, on our board there's no one else to take control, because there's just one device that has the brains to be Master, and that's our 9S12X). A "Stop" involves a positive-going transition on SDA while SCL is HIGH. (In our I²C module, clear the Tx bit in the IBCR.)

Reading

5. Indicate a "Restart", which will allow us to keep control of the bus, and allows us to issue a new command to the slave of our choice. (There's a Restart bit in IBCR.)
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. Indicate 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.

Variations on the theme:

- If you need to write more than one byte to a device that knows how to do that (e.g. one that auto-increments the internal address), you can just keep writing data bytes until you're done, then indicate a Stop. One example of this is for memory devices that require a 16-bit address, like the 24AA512. For these, we have to send two bytes to establish the starting address of 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 how to do that, you can just keep reading data bytes until you're done before indicating a Stop. One example of this is the MPL3115A2, which can send its information in five consecutive bytes. Other devices need to transmit 16-bit values in two bytes.
- … There seem to be an almost-infinite number of variations on the theme. The datasheets for each I²C device will provide necessary information and timing diagrams to help you establish a working relationship with that device.

You will want to start a new library and library header called "IIC0_Lib". The IIC0 part is because there are two I²C busses on our controller, and the devices on board are wired up to I²C-0. These are the primary functions you'll eventually have in it:

```
void IIC0_Init(void);
void IIC0_Write(unsigned char cAddr, unsigned char cReg, unsigned char cData);
unsigned char IIC0_Read(unsigned char cAddr, unsigned char cReg);
void IIC0_WriteDAC(unsigned char cAddr, unsigned char cCommand,  unsigned int iData);
```

In the header file, you may choose not to specify the names of the parameters passed, to provide you with more flexibility. Suitable names have been provided above to indicate what the various parameters do.

The Init routine sets up the I²C-0 port in the 9S12XDP512. In the "Data Sheet", this is discussed in detail in Chapter 9, parts of which are included in the discussion below for convenience.

There are five registers used by the I²C controller. To specify which I²C module we're interacting with, we need to tack IIC0_ in front of the register names.

Chapter 9 Inter-Integrated Circuit (IICV2) Block Description

9.3.2 Register Descriptions

This section consists of register descriptions in address order. Each description includes a standard register diagram with an associated figure number. Details of register bit and field function follow the register diagrams, in bit order.

Register Name		Bit 7	6	5	4	3	2	1	Bit 0
IBAD	R	ADR7	ADR6	ADR5	ADR4	ADR3	ADR2	ADR1	0
	W								
IBFD	R	IBC7	IBC6	IBC5	IBC4	IBC3	IBC2	IBC1	IBC0
	W								
IBCR	R	IBEN	IBIE	MS/$\overline{SL}$	Tx/$\overline{Rx}$	TXAK	0	0	IBSWAI
	W						RSTA		
IBSR	R	TCF	IAAS	IBB	IBAL	0	SRW	IBIF	RXAK
	W								
IBDR	R	D7	D6	D5	D4	D3	D2	D1	D0
	W								

= Unimplemented or Reserved

Figure 9-2. IIC Register Summary

IIC0_IBAD – This is the I²C slave address assigned to the 9S12. We don't have to worry about this one, as we'll always be the Master, not the Slave.

IIC0_IBFD – This is the Frequency Divider Register to set up the communication rate. 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, which is five pages long! Here's a piece of it that will help us figure out what we need to put into IIC0_IBFD:

Chapter 9 Inter-Integrated Circuit (IICV2) Block Description

Table 9-5. IIC Divider and Hold Values (Sheet 2 of 5)

IBC[7:0] (hex)	SCL Divider (clocks)	SDA Hold (clocks)	SCL Hold (start)	SCL Hold (stop)
3E	3072	513	1534	1537
3F	3840	513	1918	1921
MUL=2				
40	40	14	12	22
41	44	14	14	24
42	48	16	16	26
43	52	16	18	28
44	56	18	20	30
45	60	18	22	32
46	68	20	26	36
47	80	20	32	42
48	56	14	20	30
49	64	14	24	34
4A	72	18	28	38
4B	80	18	32	42
4C	88	22	36	46
4D	96	22	40	50
4E	112	26	48	58

Given the peripherals installed on our board, your instructors (primarily Simon Walker) have determined that 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. (All of that information is found in the data sheets for the various devices, and values have to be chosen for the slowest device on the bus.) Given that we have an 8 MHz bus clock, you should be able to determine that the appropriate value for IIC0_IBFD is 0x47.

IIC0_IBCR – We want to enable I²C, 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 I²C must be enabled before changing any of the other bits in this register, so we have to turn that bit ON first by itself, then make sure the interrupts and WAIT mode bits are turned OFF after that – two writes to this register.

If you've been following this discussion, you should be able to verify the IIC0_Init() routine shown in the start to the IIC0_Lib.c file, shown below.

```
//IIC0 Library Files
//Processor:  MC9S12XDP512
//Crystal:  16 MHz
//by P Ross Taylor
//May 2014

#include <hidef.h>
#include "derivative.h"
#include "IIC0_Lib.h"

void IIC0_Init(void)
{
    IIC0_IBFD=0x47;          //100 kHz, SDA Hold = 20 cks, SCL Hold Start = 32 SCL Hold Stop = 42
    IIC0_IBCR|=0b10000000;   //enable the bus - must be done first
    IIC0_IBCR&=0b10111110;   //no interrupts, normal WAI
}
```

Two bits are important in the Status Register, IIC0_IBSR: *b5*, IBB, is the I²C Bus Busy flag, and *b1*, IBIF, the I²C Interrupt Flag, which is set whenever a transfer is complete (whether or not we have interrupts enabled). The IBIF flag needs to be cleared by writing a 1 to it. For some devices, such as the 24AA412 EEPROM, you will also need to monitor RXAK, the "receive acknowledge" bit if you want to access more than one byte of memory at a time.

Writing

We need to be able to write out commands to the peripherals. Here's the prototype for the Write command from the header file:

```
void IIC0_Write(unsigned char cAddr, unsigned char cReg, unsigned char cData);
```

The Write function 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, as indicated by a LOW on the IBB bit, b5.
2. Once the bus is free, change the micro to "Master" mode, set to "Transmit". These bits are in the Control Register, IIC0_IBCR.
3. Place the device address on the bus with the LSB set to "Write" mode.
4. Wait for the Byte Transfer Complete process, as 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, and don't clear the IBIF flag yet.
8. Stop transmitting and exit "Master" mode, using the Control Register.
9. Finally, clear the IBIF flag.

Take some time to see how the above discussion is handled in the following function.

```
void IIC0_Write(unsigned char cAddr, unsigned char cReg, unsigned char cData)
{
    while(IIC0_IBSR & 0b00100000);      //wait for not busy flag
    IIC0_IBCR |= 0b00110000;            //micro as master, start transmitting

    IIC0_IBDR = cAddr & 0b11111110;     //place address on bus with /Write
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBDR = cReg;                   //locate desired register
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBDR = cData;                  //send data
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBCR &= 0b11001111;            //stop transmitting, exit Master mode
    IIC0_IBSR |= 0b00000010;            //clear flag

}
```

Note: This routine, and the others in this set of Course Notes, are handled as simply as possible, and do not provide means of escape if something goes wrong with communication – the system may simply freeze, waiting for a flag that never comes up. If you have an application where the system needs to be essentially fail-proof, you will need to incorporate ways of handling various unexpected exceptions, particularly by avoiding blocking loops (of which there are four in the previous code alone!). Your instructor or someone with a lot of experience with this (*i.e.* Simon Walker) might be willing to help you with this.

Reading

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.

```
unsigned char IIC0_Read(unsigned char cAddr, unsigned char cReg);
```

The contents of that register are returned to the main program as a byte. Here's the procedure, the first six parts of which were also part of the "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. The last byte received from a device is supposed to have a NACK 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.
12. Here's a curious fact: in order to initiate a Read, you need to read the I^2C 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 IBIF flag.
16. Finally, you can read the real data out of the I^2C Data Register and return it to the main program.

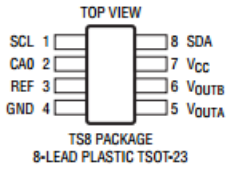
Take some time to see how the above discussion is handled in the function shown below.

```
unsigned char IIC0_Read(unsigned char cAddr, unsigned char cReg)
{
    unsigned char cData;

    while(IIC0_IBSR & 0b00100000);      //wait for not busy flag
    IIC0_IBCR |= 0b00110000;            //micro as master, start transmitting

    IIC0_IBDR = cAddr & 0b11111110;     //place address on bus with /Write
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBDR = cReg;                   //locate desired register
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBCR |= 0b00000100;            //restart

    IIC0_IBDR = (cAddr | 0b00000001);   //place address on bus with Read
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBCR |= 0b00001000;            //reading 1 unsigned char only
    IIC0_IBCR &= 0b11101111;            //receive unsigned char
    cData = IIC0_IBDR;                  //not actually -- starts the process
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBCR &= 0b11011111;            //stop
    IIC0_IBSR |= 0b00000010;            //clear flag
    cData = IIC0_IBDR;                  //for real this time

    return cData;
}
```

LTC2633HZ12 I²C DAC – 16-bit Data Writes

The LTC2633HZ12 DAC that's been added to your board is a dual 12-bit DAC, set up so that its internal reference is effectively 4.096 V. DACA and DACB, the outputs of the two internal DACs, are available to the right of the DAC module on your microcontroller kit, with a ground pin positioned between them.

The pinout of the IC is as follows:



On the board, "CA0" is connected to ground to make the device's slave address end in "00" – i.e. 0x10 or 0b0010000r. (Check back a few pages to recall why this is the case.) If you want, you can add up to two more LTC2633HZ12 modules: leaving CA0 floating makes the address 0x11 or 0b0010001r, and connecting CA0 to +5 V makes it 0x12 or 0b0010010r.

The data sheet for this device goes into a lot of detail about how to use it, but the following snippet is particularly informative for us in terms of how to send information to the DAC.

LTC2633

OPERATION

The format of the three data bytes is shown in Figure 3. The first byte of the input word consists of the 4-bit command, followed by the 4-bit DAC address. The next two bytes contain the 16-bit data word, which consists of the 12-, 10- or 8-bit input code, MSB to LSB, followed by 4, 6 or 8 don't-care bits (LTC2633-12, LTC2633-10 and LTC2633-8 respectively). A typical LTC2633 write transaction is shown in Figure 4.

The command bit assignments (C3-C0) and address (A3-A0) assignments are shown in Tables 3 and 4. The first four commands in the table consist of write and update operations. A write operation loads a 16-bit data word from the 32-bit shift register into the input register. In an update operation, the data word is copied from the input register to the DAC register. Once copied into the DAC register, the data word becomes the active 12-, 10-, or 8-bit input code, and is converted to an analog voltage at the DAC output. Write to and update combines the first two commands. The update operation also powers up the DAC if it had been in power-down mode. The data path and registers are shown in the Block Diagram.

Table 3. Command Codes

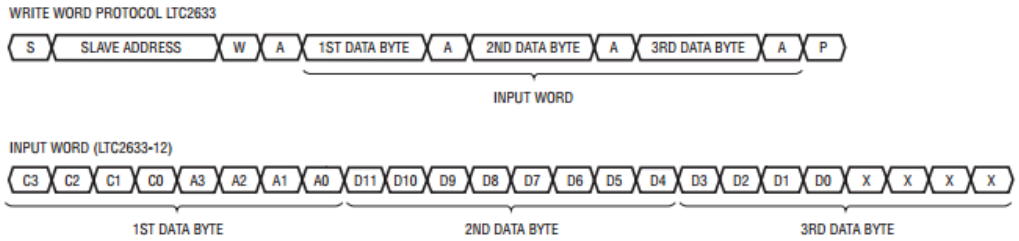
COMMAND*				
C3	C2	C1	C0	
0	0	0	0	Write to Input Register n
0	0	0	1	Update (Power-Up) DAC Register n
0	0	1	0	Write to Input Register n, Update (Power-Up) All
0	0	1	1	Write to and Update (Power-Up) DAC Register n
0	1	0	0	Power-Down n
0	1	0	1	Power-Down Chip (All DAC's and Reference)
0	1	1	0	Select Internal Reference (Power-Up Reference)
0	1	1	1	Select External Reference (Power-Down Internal Reference)
1	1	1	1	No Operation

*Command codes not shown are reserved and should not be used.

Table 4. Address Codes

ADDRESS (n)*				
A3	A2	A1	A0	
0	0	0	0	DAC A
0	0	0	1	DAC B
1	1	1	1	All DACs

* Address codes not shown are reserved and should not be used.



Since it's a 12-bit device, it's obviously going to need two bytes of data sent to it. In fact, as an I²C device, it's going to need three bytes total: A Command byte and two bytes of data, as shown on the previous page. Of course, this comes after identifying the DAC itself on the I²C bus using its address.

We'll send the address as a parameter each time to be consistent with other functions in our I²C library, although it should always be 0x20 – based on the seven-bit I²C address of 0x10, which translates to 0b0010000r, where 'r' is the Read/Write bit.

From the data sheet, the device defaults to the mode in which the DAC reference is 4.096 V. Also, the command we're going to use to write to the device includes a power-up, so we don't need to do any initializing.

The Command byte is actually made up of one nibble for the command and one nibble to specify which DAC channel or channels you're addressing. The instructors in this course have played with this device a fair bit, and have determined that the simplest way to send a value to one of the DAC channels so that it appears instantly is to use the "Write to and update DAC register n" command, 0b0011. Then, the lower nibble will be 0b0000 for DAC A, 0b0001 for DAC B, 0r 0b1111 to set both DACs to the same value.

Like a number of I²C devices (another example is the MPL3115A2 barometer), this device expects its data to arrive "left-justified", meaning that the 12 bits it's expecting are the upper 12, not the lower 12 of the 16 bits in an *unsigned int*. (Incidentally, this is to keep it compatible with other members of the family that have more bits, which improve the resolution by using the lower (i.e. finer resolution) bits.) So we need to send a byte that contains the 8 upper bits, and a byte that contains the 4 lower bits followed by four zeros.

We need to write a new version of the I²C "Write" command that fits the following header:

```
void IIC0_WriteDAC(unsigned char cAddr, unsigned char cCommand, int iData);
```

We also need to know what the step size is for this DAC.

$$StepSize = \frac{V_{ref}}{2^n} = \frac{4.096}{2^{12}} = 1mV/step$$

When we send a numeric value to the DAC, it will be a number representing the voltage in millivolts, and it will be in the format we're used to: right-justified hexadecimal. So, the first thing we need to do in our function for writing to the DAC is to convert the incoming value to left-justified format, which simply means moving it from the lower 12 bits to the upper 12 bits of a 16-bit value. Once that's been done, we need to send the result out as two 8-bit bytes, since we can only send 8 bits at a time on the I²C bus.

Based on the timing diagram on the previous page, the three bytes can be sent one after the other, as long as we wait for the I²C flag to indicate that the device is ready for another byte. (Incidentally, this is the sequential write process discussed previously, and can be used for other devices we've touched on that have this as one of their modes of operation.)

Again, if you've been following the previous discussion, you should come up with something like the code on the following page. Don't just copy this: Work it through to make sure you understand what it does. You may also want to come up with more sophisticated ways of moving the data to the right place in the *int* variable and parsing out the two bytes, such as using regular division and MOD division.

```
void IIC0_WriteDAC(unsigned char cAddr, unsigned char cCommand, int iData)
{
  iData*=16;              //move data into the upper 12 bits

  while(IIC0_IBSR & 0b00100000);    //wait for not busy flag
  IIC0_IBCR |= 0b00110000;      //micro as master, start transmitting

  IIC0_IBDR = cAddr & 0b11111110;   //place address on bus with /Write
  while(!(IIC0_IBSR & 0b00000010)); //wait for flag
  IIC0_IBSR |= 0b00000010;      //clear flag

  IIC0_IBDR = cCommand;       //send the desired command
  while(!(IIC0_IBSR & 0b00000010)); //wait for flag
  IIC0_IBSR |= 0b00000010;      //clear flag

  IIC0_IBDR = (unsigned char)(iData/256);   //send first unsigned char of the data
  while(!(IIC0_IBSR & 0b00000010)); //wait for flag
  IIC0_IBSR |= 0b00000010;      //clear flag

  IIC0_IBDR = (unsigned char)(iData&0b0000000011111111);  //send second unsigned char of the data
  while(!(IIC0_IBSR & 0b00000010)); //wait for flag
  IIC0_IBCR &= 0b11001111;      //stop transmitting, exit Master mode
  IIC0_IBSR |= 0b00000010;      //clear flag
}
```

The following code snippet shows a very simple implementation of the preceding function that generates two 4,095-step ramp waves: a rising ramp on DAC A and a falling ramp on DAC B. You could use this to check your IIC0_WriteDAC() function. You will need to declare an *unsigned int* variable called iDataOut.

```
IIC0_Init();

  for (;;)
  {
  IIC0_WriteDAC(0x20,0b00110000,iDataOut);
  IIC0_WriteDAC(0x20,0b00110001,0x0FFF-iDataOut++);
  }

}
```

MPL3115A2: Standard 8-bit Reads and Writes

The MPL3115A2 "Precision Altimeter" gives the opportunity to do simple I2C communication to verify the Read and Write functions above, since its internal registers are few enough to have only 8-bit addresses, and the register contents are also 8-bit locations.

The following is a flowchart from the MPL3115A2 data sheet that shows how to set up the device and how to read the internal registers. For this course, we'll just be using the "Polling" side of the flowchart, so it's not as bad as it seems at first glance.

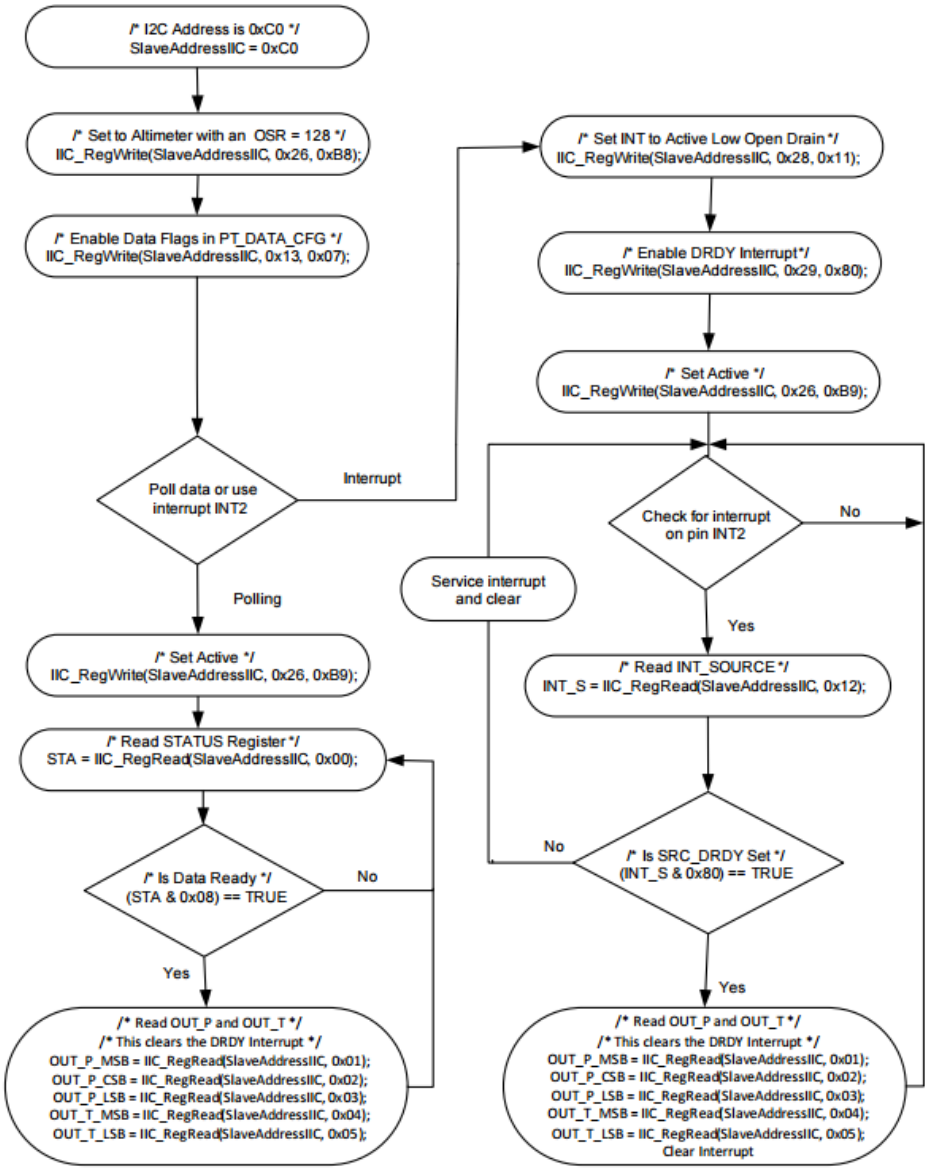


Figure 5. Polling or Interrupt - No FIFO

MPL3115A2

The registers we need to use are summarized below:

7 Register Descriptions

Table 9. Register Address Map

Register Address	Name	Reset	Reset when STBY to Active	Type	Auto-Increment Address		Comment	
0x00	Sensor Status Register (STATUS)(1)(2)	0x00	Yes	R	0x01		Alias for DR_STATUS or F_STATUS	
0x01	Pressure Data Out MSB (OUT P_MSB)(1)(2)	0x00	Yes	R	0x02	0x01	Bits 12-19 of 20-bit real-time Pressure sample.	Root pointer to Pressure and Temperature FIFO data.
0x02	Pressure Data Out CSB (OUT_P_CSB)(1)(2)	0x00	Yes	R	0x03		Bits 4-11 of 20-bit real-time Pressure sample	
0x03	Pressure Data Out LSB (OUT_P _LSB)(1)(2)	0x00	Yes	R	0x04		Bits 0-3 of 20-bit real-time Pressure sample	
0x04	Temperature Data Out MSB (OUT_T_MSB)(1)(2)	0x00	Yes	R	0x05		Bits 4-11 of 12-bit real-time Temperature sample	
0x05	Temperature Data Out LSB (OUT_T _LSB)(1)(2)	0x00	Yes	R	0x00		Bits 0-3 of 12-bit real-time Temperature sample	
0x13	PT Data Configuration Register (PT_DATA_CFG)(1)(3)	0x00	No	R/W	0x14		Data event flag configuration	
0x26	Control Register 1 (CTRL_REG1)(1)(4)	0x00	No	R/W	0x27		Modes, Oversampling	
0x27	Control Register 2 (CTRL_REG2)(1)(4)	0x00	No	R/W	0x28		Acquisition time step	
0x28	Control Register 3 (CTRL_REG3)(1)(4)	0x00	No	R/W	0x29		Interrupt pin configuration	
0x29	Control Register 4 (CTRL_REG4)(1)(4)	0x00	No	R/W	0x2A		Interrupt enables	
0x2A	Control Register 5 (CTRL_REG5)(1)(4)	0x00	No	R/W	0x2B		Interrupt output pin assignment	

Here are some points about the device, the flowchart, and the registers involved.

- The device's I2C address is 0x60 (i.e. 0b1100000r), so we need to communicate with it using 0xC0. Instead of making a variable to hold the slave address as shown in the flowchart, we can just put 0xC0 in the slave address parameter field.
- We'll be using our function "IIC0_Write" in place of their "IIC_RegWrite" function.
- During configuration, we need to set up Control Register 1 (0x26). The other registers default to conditions that are acceptable for us at this point. Here are the bits of CTRL_REG1:

Table 57. CTRL_REG1 Register

	7	6	5	4	3	2	1	0
R	ALT	RAW	OS2	OS1	OS0	0	OST	SBYB
W						RST		
Reset	0	0	0	0	0	0	0	0

Following the flowchart, we want to set this up for an "over-sampling rate" of 128 by setting the OSn bits to produce 2^7. Also, the flowchart suggests setting *b7*, which puts the device into "Altimeter" mode instead of "Barometer" mode. This results in the value 0b10111000, or, as indicated in the flowchart, 0xB8. We actually want to operate in "Barometer" mode, so we'll send 0b00111000 instead.

- The flowchart then sends 0b00000111 (0x07) to the PT_DATA_CFG register to enable data flags so that pressure events (PDEFE) or temperature events (TDEFE) are available, using Data Ready Event Mode (DREM).

Table 37. PT_DATA_CFG Register

	7	6	5	4	3	2	1	0
R	0	0	0	0	0	DREM	PDEFE	TDEFE
W								
Reset	0	0	0	0	0	0	0	0

- The last step in the initialization part of the flowchart takes us back to CTRL_REG1, where we take it out of standby (SBYB) by putting a 1 in *b0*. Since there isn't a good way of just use OR to change a single bit, we overwrite the register with the new value: 0b00111001. This is shown in the flowchart as 0xB9, but that was for Altimeter mode, so, for Barometer mode, we don't want the MSB set.
- Simon Walker indicates that, for reliable operation, we now need to read the Status register once before entering the main loop that repeatedly checks the status register, then reads the data registers when valid data is indicated.
- Inside the loop, as indicated in the flowchart, we wait until the MPL3115A2 reports the availability of good data, then we read the registers we're interested in. For pressure information, these would be 0x01, 0x02, and 0x03. For temperature information, these would be 0x04 and 0x05.
- The bits of the status register (0x00) are as follows:

Table 12. DR_STATUS Register

	7	6	5	4	3	2	1	0
R	PTOW	POW	TOW	0	PTDR	PDR	TDR	0
W								
Reset	0	0	0	0	0	0	0	0

The only bit we need to concern ourselves with is *b3*: PTDR stands for "Pressure and Temperature Data Ready". We have to wait for this bit to be SET before we can read valid data from the other registers. (If you wanted, you could watch for "PDR" or "TDR" instead, if you only needed pressure or temperature data.)

The values in the data registers are presented in a slightly complicated format, in that they are left-justified (occupying the upper bits and leaving the least significant bits as zeros), and they have a fractional component.

The pressure data is in Pascals (Pa), and is provided as 20-bits in Q18.2 format. This means that the most significant 18 bits are the hexadecimal value of the pressure in Pascals, and the other two bits are, in order, ½ (2^{-1}) and ¼ (2^{-2}) Pascal weightings.

For example, the three-byte value for pressure can be interpreted as follows:

0x01 = 0b0110 1010
0x02 = 0b0111 0001
0x03 = 0b1001 0000

Pressure = 0b011010100111000110.01 = 108,998.25 Pa

The temperature is even more complicated, as it is returned as 2's complement negative 12-bit fractional data in ºC, provided in Q8.4 format (although the data sheet says Q12.4, but that would 16 bits). The whole-degrees portion is in the first byte (Register 0x04), and the fractional part is the high nibble of Register 0x05, accurate to 1/16th of a degree (2^{-4}).

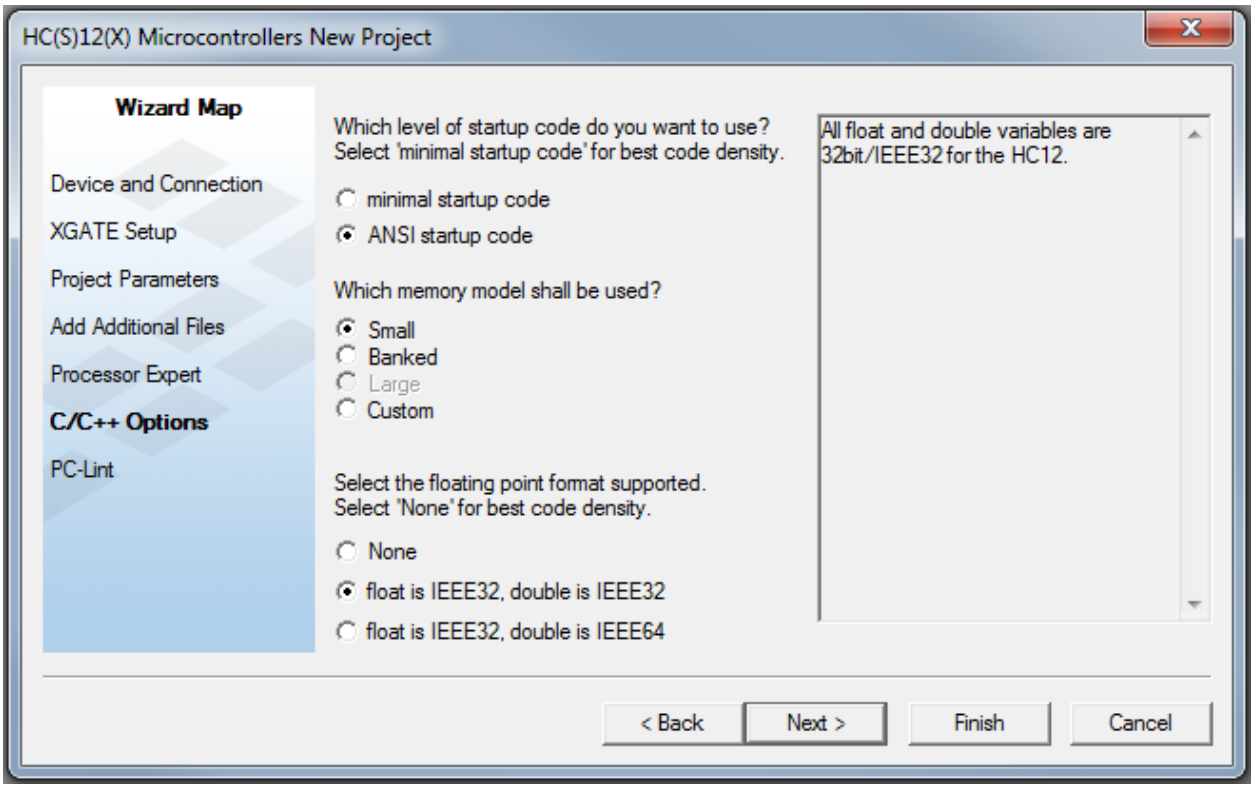
If you just want to display whole degrees, you can ignore Register 0x05 and just use Register 0x04. To display negative temperatures in a form suitable for human consumption (pretty important in Alberta!), you would need to perform a 2's complement conversion on the negative values to find their magnitude, then simply insert a "negative" sign in front of the value.

Handling the full temperature this way, including the fractional component, is beyond this course, but you might want to investigate it for future use of your microcontroller kit.

Floating-Point Math in ANSI C

Having gone through an example of how to handle multi-bit values from a precise sensor using integer math, it may be a good time to show that the 9S12, operating in ANSI C, can also do floating point math. One reason we've been avoiding this until now is that floating point math requires fairly large amounts of memory, takes a lot of clock cycles, and involves some fairly complex functions. However, there are times when it just makes sense to work with real numbers rather than integers, as long as you are aware that download times will increase, you might run out of memory, and most everything is going to slow down.

To begin with, when you are setting up a new project, you need to specify that you are going to be using floating point math, and in which format. One of the screens you've been going to in order to select the memory model also gives you the option of working in floating point:



From experience, your instructors recommend choosing "float is IEEE32, double is IEEE32", as shown above: the other option has been problematic.

Also, in the skeleton file used when creating a new project, you will probably need to invoke the ANSI C <stdio.h> library, and, depending on what mathematical functions you intend to use, you may also need to invoke the <math.h> library.

<stdio.h>

Here are the functions available in <stdio.h>. You can find it at C:\Program Files (x86)\Freescale\CWS12v5.1\lib/hc12c\include\math.h.

fopen	fclose	fgetpos
fsetpos	freopen	fseek
scanf	remove	ftell
sscanf	rename	rewind
vsscanf	tmpfile	fgetc
puts	tmpnam	fread
printf	fflush	fwrite
fprintf	setbuf	fgets
vfprintf	setvbuf	fputs
sprintf		fscanf
vsprintf		ungetc
set_printf		gets
vprintf		

Since there's no file structure, or even mass storage device, on your microcontroller board, none of the file-related commands are of any direct use to you, and since your micrcocontroller board doesn't have a keyboard or monitor, none of the standard user-interface device (console) commands do anything directly, either. So, what use is this library to you? You could potentially set up your board with some "Standard I/O" devices (keyboard interface, video interface, external memory), and then you would have to define the parameters of these devices so the board knew what to access using the console commands. This would be a significant challenge (beyond this course), but not impossible.

However, there are other functions in this library that are of direct use to you. Commands related to string manipulation fit this category; "sprintf", for example, is a function you can use to format strings, particularly those with floating-point numbers in them.

Here's an informative example from a program that has previously received a 12-bit 2's complement value from the X channel of an accelerometer, placed into the variable "iX":

```
if(sprintf(DispString,"X = %+4.3f g     ",iX/1000.0)>0)
{
    LCD_Pos(0,0);
    LCD_String(DispString);
}
```

Note the following:

- All sprintf returns is a flag to indicate a valid result – hence the "if" statement.
- The variable "DispString" was declared previously in the setup for this program as a 21-byte char array, and is used as the target for sprintf. (21 provides enough room for the 20 characters on a line of the display, followed by a NULL terminator.)
- The contents of the string are contained inside double quotes.
- The "%" symbol indicates that formatting commands for a fillable field follow.
- "+" indicates that the sign of the number will be shown, both positive and negative.
- "4.3f" indicates that we want to display a floating point number made up of four digits, of which three will follow the decimal point.
- In the calculation of the value to be placed in the field, we're dividing by 1000.0 (not just 1000) to do an implicit cast of the result into floating point format.

Go here for a complete description of the "sprintf" command:
http://www.tutorialspoint.com/c_standard_library/c_function_sprintf.htm

<math.h>

This library has a lot of useful functions in it, as shown below.

frexp	pow	asinf
ldexp	sqrt	acosf
modf	ceil	atanf
frexpf	floor	atan2f
ldexpf	fabs	log10f
modff	fmod	expf
cos	sincos	logf
cosh	sncsh	powf
sin	sqrt_r	sqrtf
sinh	pow_i	ceilf
tan	exp_r	floorf
tanh	log_r	fabsf
asin	cosf	fmodf
acos	coshf	sincosf
atan	sinf	sncshf
atan2	sinhf	sqrtf_r
log10	tanf	powf_i
exp	tanhf	expf_r
log		logf_r

Again, a lot of information about the use and formatting of each of these is available on the Internet when you need to use them.

Of all the functions in this library, you're most likely to use the trigonometry and exponential groups. It should come as no surprise that the trig functions are radians-based, so, if you want to work in degrees, you'll need to do the appropriate conversions.

$$Degrees = Radians\left(\frac{180}{\pi}\right)$$

The math.h library provides a value for π, but it's a bit awkward: _M_PI. You might want to assign that to a slightly more workable variable name – just make sure it's a *float*. There are other constants available, too. A complete listing can be found in the header file itself, under C:\Program Files (x86)\Freescale\CWS12v5.1\lib/hc12c\include\math.h.

M41T81 Real-Time Clock – Standard 8-bit Reads and Writes

Back to I^2C devices: The M41T81 Real-Time Clock is designed to do either multi-byte communication (like the EEPROM), or individual register reads to get the information we're looking for, and individual register writes to change the time, date, or control bits.

As with the other devices we've covered in this course, there are a lot of features we don't have time to cover in this brief overview. However, the full details can be found in the data sheet for this device, available from the Internet or in Moodle for this course.

The designers of this device chose to present the data in BCD, which is quite helpful, in that we don't need to do conversions before we display its results.

Since the results are in BCD, you need to work with the two nibbles in a byte to get the full number (e.g.) to get 35 s, you read register 0x01, in which the upper nibble will contain 3 (once ST is masked off) and the lower nibble will be 5.

The M41T81 has likely been running on your board ever since it was assembled, and it contains a lot of information, which, at this point, is almost guaranteed to be incorrect. This device has its own internal 32.768 kHz crystal, so it doesn't rely on the bus clock or an external crystal oscillator. It has a backup battery with a circuit that detects when the main board power is turned off, so it continues to maintain the time and any settings when the board is turned off. It keeps track of time and date in hundredths of seconds, seconds, minutes, hours, day of week, date, months, years, and even centuries (although just the twenty-first and twenty-second centuries!), with leap years built in, so it is a true calendar as well as a clock.

Along with current time and date, the device has the capability of being used as an alarm, as a square wave generator, and as a microcontroller monitoring device called a "watchdog". Here's a list of its internal registers, along with a description of the labels used for control bits that are scattered throughout the registers:

M41T81 **Clock operation**

Table 2. Clock register map(1)

Addr	D7	D6	D5	D4	D3	D2	D1	D0	Function/range BCD format	
00h	0.1 seconds				0.01 seconds				Seconds	00-99
01h	ST	10 seconds			Seconds				Seconds	00-59
02h	0	10 minutes			Minutes				Minutes	00-59
03h	CEB	CB	10 hours		Hours (24 hour format)				Century/ hours	0-1/00-23
04h	0	0	0	0	0	Day of week			Day	01-7
05h	0	0	10 date		Date: day of month				Date	01-31
06h	0	0	0	10M	Month				Month	01-12
07h	10 years				Year				Year	00-99
08h	OUT	FT	S	Calibration					Control	
09h	0	BMB4	BMB3	BMB2	BMB1	BMB0	RB1	RB0	Watchdog	
0Ah	AFE	SQWE	ABE	Al 10M	Alarm month				Al month	01-12
0Bh	RPT4	RPT5	Al 10 date		Alarm date				Al date	01-31
0Ch	RPT3	HT	Al 10 hour		Alarm hour				Al hour	00-23
0Dh	RPT2	Alarm 10 minutes			Alarm minutes				Al min	00-59
0Eh	RPT1	Alarm 10 seconds			Alarm seconds				Al sec	00-59
0Fh	WDF	AF	0	0	0	0	0	0	Flags	
10h	0	0	0	0	0	0	0	0	Reserved	
11h	0	0	0	0	0	0	0	0	Reserved	
12h	0	0	0	0	0	0	0	0	Reserved	
13h	RS3	RS2	RS1	RS0	0	0	0	0	SQW	

Keys:
S = Sign bit
FT = Frequency test bit
ST = Stop bit
0 = Must be set to '0'
BMB0-BMB4 = Watchdog multiplier bits
CEB = Century enable bit
CB = Century bit
OUT = Output level
ABE = Alarm in battery backup mode enable bit
AFE = Alarm flag enable flag
RB0-RB1 = Watchdog resolution bits
RPT1-RPT5 = Alarm repeat mode bits
WDF = Watchdog flag (read only)
AF = Alarm flag (read only)
SQWE = Square wave enable
RS0-RS3 = SQW frequency
HT = Halt update bit

Notice again that the data is presented in Binary-Coded Decimal (BCD), using two nibbles for each item. The number of bits required for the most significant digit depends on the size of the particular digit, so, for example, the month only needs a zero or a one for its first digit, the day of the month requires two bits for its first digit to cover the possibilities of 0, 1, 2, or 3, and the year requires all four bits to represent values from 0 to 9.

Since the other bits aren't needed for the data, some of them get used for control and reporting functions. A very important one is "ST", which is the MSB of the "seconds" register. This bit stops the clock crystal, and the registers hold the last available data.

Another bit that's very important is *b6* of the Alarm Hour register: "HT". When the power on the board goes down and the Real Time Clock switches to battery, this bit is set to halt the updating of the registers, while the clock continues to run in the background. This allows the user to write code to read the registers on power-up to find out when the power failed, before reactivating the normal operation in which the proper time will be reported.

So, if you want to know when the power went down, you can read the time registers before clearing the HT bit to get the power-down information; in any case, you will need to clear the "HT" bit to let the clock report the current time.

The "ST" bit must be cleared to allow the clock crystal to run. Once this bit has been cleared, it will stay cleared until it is deliberately set, so it's best to check to see if it needs to be cleared before doing anything to it. If you go through the process of clearing it unnecessarily, you may lose the occasional second on the clock, since this bit is in the "seconds" register. Why? Because to clear the ST bit, we need to read the seconds register, clear the ST bit in the read-in value, then write the resulting value back into the seconds register. If the seconds register updates during this process, the value written back in will be the old value, which is one second behind. Also, if you clear the ST bit before you clear the HT bit, you will be reading in the seconds value that was held for reporting the power-down, and will write that back in, overwriting the current seconds value with an old (and incorrect) value.

Unlike interrupt flags, these bits are cleared by writing '0' to them.

The Real-Time Clock's I^2C address is0x68, which translates into 0b1101000r for our purposes. The code snippet on the following page shows how to start the clock if it isn't running, how to allow it to report current data, and how to read in current data from the registers associated with seconds, minutes, hours, date, month, and year. The actual process of displaying the data is not shown, as that would be dependent on the display device chosen.

```
    HasHT=IIC0_Read(0xD0,0x0C);
    IIC0_Write(0xD0,0x0C,(HasHT&0b10111111));   //clear the Halt bit

    Sec=IIC0_Read(0xD0,0x01);                   //if set, clear the ST bit
    if((Sec&0b10000000)!=0) IIC0_Write(0xD0,0x01,(Sec&0b01111111));

    //SetTime(0x30,0x25,0x10,0x02,0x07,0x12,0x15);

    for (;;)        //endless program loop
    {
/*********************************************************************
*       Main Program Code
*********************************************************************/

        Sec=IIC0_Read(0xD0,0x01)&0b01111111;
        Min=IIC0_Read(0xD0,0x02)&0b01111111;
        Hr=IIC0_Read(0xD0,0x03)&0b00111111;
        Date=IIC0_Read(0xD0,0x05)&0b00111111;
        Mth=IIC0_Read(0xD0,0x06)&0b00011111;
        Year=IIC0_Read(0xD0,0x07);

        UpdateDisplay();
    }
```

In this code snippet, notice there's a "SetTime" function call that has been commented out. The code was run once with that line included, then the version of the code with it excluded was down-loaded to the microcontroller so that further resets or power-ups will not set the time back to the numbers hard-coded into this routine. Clearly, a more sophisticated means of setting the clock would be useful – for example, a function that responds to a switch press if the user wants to set the time.

Notice that all of the values sent to the "SetTime" function are indicated as hexadecimal: that's because BCD, which is what the clock is expecting, isn't decimal – it's a binary (or hex) code used to represent decimal values. So, 0x31 represents 31 minutes in the second byte of the function call.

Position Information with the LSM303DLHC – Standard 8-bit Reads and Writes

(Optional topic) Another device that can be controlled and accessed using eight-bit address and data reads and writes is theLSM303DLHC eCompass Module. (It can also be accessed using multi-byte sequences, but we'll stick with the easier approach.) This unit contains a three-axis accelerometer, a three-axis magnetometer, and, probably because everyone else is doing it, another temperature sensor. Here's a clip from the datasheet.

1.2 Pin description

Figure 2. Pin connections

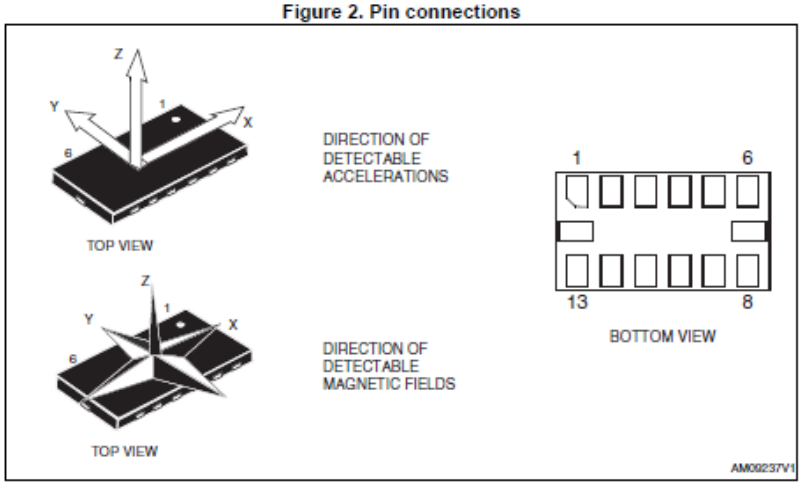


Table 2. Pin description

Pin#	Name	Function
1	Vdd_IO	Power supply for I/O pins
2	SCL	Signal interface I^2C serial clock (SCL)
3	SDA	Signal interface I^2C serial data (SDA)
4	INT2	Inertial interrupt 2
5	INT1	Inertial interrupt 1
6	C1	Reserved capacitor connection (C1)
7	GND	0 V supply
8	Reserved	Leave unconnected
9	DRDY	Data ready
10	Reserved	Connect to GND
11	Reserved	Connect to GND
12	SETP	S/R capacitor connection (C2)
13	SETC	S/R capacitor connection (C2)
14	Vdd	Power supply

3-Axis Accelerometer

You should recall, from earlier physics-related science courses, that objects near the earth's surface accelerate at a rate of approximately 9.81 m/s^2 if allowed to fall freely. This is referred to as 1.0 g (not to be confused with the SI unit for grams – we just ran out of letters!). The accelerometers in the LSM303DLHC report acceleration in milli-g's, which shows up in the datasheet as *mg*, again, not to be confused with milligrams.

The seven-bit I^2C address for the accelerometer is 0x19, so the eight-bit representation of this is 0b0011001r.

There are a bunch of internal registers associated with the accelerometer:

Name	Type	Register Hex
Reserved (do not modify)		00 - 1F
CTRL_REG1_A	rw	20
CTRL_REG2_A	rw	21
CTRL_REG3_A	rw	22
CTRL_REG4_A	rw	23
CTRL_REG5_A	rw	24
CTRL_REG6_A	rw	25
REFERENCE_A	rw	26
STATUS_REG_A	r	27
OUT_X_L_A	r	28
OUT_X_H_A	r	29
OUT_Y_L_A	r	2A
OUT_Y_H_A	r	2B
OUT_Z_L_A	r	2C
OUT_Z_H_A	r	2D
FIFO_CTRL_REG_A	rw	2E
FIFO_SRC_REG_A	r	2F
INT1_CFG_A	rw	30
INT1_SRC_A	r	31
INT1_THS_A	rw	32
INT1_DURATION_A	rw	33

Name	Type	Register Hex
INT1_CFG_A	rw	30
INT1_SRC_A	r	31
INT1_THS_A	rw	32
INT1_DURATION_A	rw	33
INT2_CFG_A	rw	34
INT2_SRC_A	r	35
INT2_THS_A	rw	36
INT2_DURATION_A	rw	37
CLICK_CFG_A	rw	38
CLICK_SRC_A	rw	39
CLICK_THS_A	rw	3A
TIME_LIMIT_A	rw	3B
TIME_LATENCY_A	rw	3C
TIME_WINDOW_A	rw	3D
Reserved (do not modify)		3E-3F

That's a lot of registers! At least they still fit in an 8-bit internal address space, so we don't need a new routine to set up this IC. The registers on the following pages the ones that are significant to us at this point, but you may find you can make use of some of the more esoteric features of this chip, such as the free-fall sensors (possibly used to park a hard-drive on a falling laptop), or the "click" sensor (possibly used to determine if someone is tapping an interactive display).

7.1.1 CTRL_REG1_A (20h)

Table 18. CTRL_REG1_A register

ODR3	ODR2	ODR1	ODR0	LPen	Zen	Yen	Xen

Table 19. CTRL_REG1_A description

ODR[3:0]	Data rate selection. Default value: 0000 (0000: power-down, others: refer to *Table 20*)
LPen	Low-power mode enable. Default value: 0 (0: normal mode, 1: low-power mode)
Zen	Z-axis enable. Default value: 1 (0: Z-axis disabled, 1: Z-axis enabled)
Yen	Y-axis enable. Default value: 1 (0: Y-axis disabled, 1: Y-axis enabled)
Xen	X-axis enable. Default value: 1 (0: X-axis disabled, 1: X-axis enabled)

ODR[3:0] is used to set the power mode and ODR selection. In the following table bit selection of ODR [3:0] for all frequencies is shown.

Table 20. Data rate configuration

ODR3	ODR2	ODR1	ODR0	Power mode and ODR selection
0	0	0	0	Power-down mode
0	0	0	1	Normal / low-power mode (1 Hz)
0	0	1	0	Normal / low-power mode (10 Hz)
0	0	1	1	Normal / low-power mode (25 Hz)
0	1	0	0	Normal / low-power mode (50 Hz)
0	1	0	1	Normal / low-power mode (100 Hz)
0	1	1	0	Normal / low-power mode (200 Hz)
0	1	1	1	Normal / low-power mode (400 Hz)
1	0	0	0	Low-power mode (1.620 kHz)
1	0	0	1	Normal (1.344 kHz) / low-power mode (5.376 kHz)

From this set of tables, we can determine the values needed to enable the three axis sensors, turn on the accelerometer, and set up its refresh rate. Note that the default condition, 0b00000111 (found in the Register address map table of the data sheet), disables the accelerometer, so we have to deal with this register. For our purposes, a speed of 100 Hz in normal mode, with all three axes enabled is suitable: 0b01010111.

For now, we'll leave control registers 2_A, 3_A, 5_A, and 6_A as they are – they deal with interrupts and some of the features that are less useful to us at this point.

Table 21. CTRL_REG2_A register

HPM1	HPM0	HPCF2	HPCF1	FDS	HPCLICK	HPIS2	HPIS1

Default 0b00000000

Table 24. CTRL_REG3_A register

I1_CLICK	I1_AOI1	I1_AOI2	I1_DRDY1	I1_DRDY2	I1_WTM	I1_OVERRUN	--

Default 0b00000000

Table 28. CTRL_REG5_A register

BOOT	FIFO_EN	--	--	LIR_INT1	D4D_INT1	LIR_INT2	D4D_INT2

Default 0b00000000

Table 30. CTRL_REG6_A register

I2_CLICKen	I2_INT1	I2_INT2	BOOT_I1	P2_ACT	--	H_LACTIVE	--

Default 0b00000000

7.1.4 CTRL_REG4_A (23h)

Table 26. CTRL_REG4_A register

BDU	BLE	FS1	FS0	HR	0(1)	0(1)	SIM

1. This bit must be set to '0' for correct operation of the device.

Table 27. CTRL_REG4_A description

BDU	Block data update. Default value: 0 (0: continuous update, 1: output registers not updated until MSB and LSB have been read
BLE	Big/little endian data selection. Default value 0. (0: data LSB @ lower address, 1: data MSB @ lower address)
FS[1:0]	Full-scale selection. Default value: 00 (00: ±2 *g*, 01: ±4 *g*, 10: ±8 *g*, 11: ±16 *g*)
HR	High-resolution output mode: Default value: 0 (0: high-resolution disable, 1: high-resolution enable)
SIM	SPI serial interface mode selection. Default value: 0 (0: 4-wire interface, 1: 3-wire interface).

For this, you need to know something you may have learned earlier: the difference between Motorola-type and Intel-type microprocessors. Motorola-type processors are referred to as "Big-Endian", as sixteen-bit values are accessed MSbyte first, LSbyte last; Intel-type processors are referred to as "Little-Endian", as sixteen-bit values are accessed LSbyte first, MSbyte last. Although the register tables tell us that the MSbyte for each accelerometer appears at the lower of the two addresses, that isn't necessarily the case: in Little-Endian mode (the default), the lower of the two addresses is actually the LSbyte, which can be very confusing. So, we want to put this device into Big-Endian mode.

Also, we need to determine the full-scale readings for the accelerometer. This is a good place to note that the values are returned as ***16-bit 2's complement*** signed integers – but we'll be reading them as two eight-bit values. More on that later. The *FS* bits determine the range that can be covered by the device. For high sensitivity, we'll choose the ±2 g scale (00). Given that this is a 12-bit device, the step size is $stepsize = \frac{4g}{2^{12}-1}$, or 0.977 mg/step (*Don't believe this value too quickly!)*

0.997 looks eerily close to 1 mg/step. In fact, in the datasheet, Table 3, the sensitivity is shown as below:

LA_So	Linear acceleration sensitivity	FS bit set to 00		1		mg/LSB
		FS bit set to 01		2		
		FS bit set to 10		4		
		FS bit set to 11		12		

The datasheet doesn't indicate which of these values is correct – it could be that the "±2 g" is an approximate value and the sensitivity is actually 1 mg/step for a true scale of ±2.047 g, or it could be that the full scale is accurate, and the sensitivity is rounded. We'll assume that the 1 mg/step is correct, as this seems to be verified empirically.

A suitable entry for CTRL_REG4_A is 0b01001000.

The data shows up in the following registers:

7.1.9 OUT_X_L_A (28h), OUT_X_H_A (29h)

X-axis acceleration data. The value is expressed in two's complement.

7.1.10 OUT_Y_L_A (2Ah), OUT_Y_H_A (2Bh)

Y-axis acceleration data. The value is expressed in two's complement.

7.1.11 OUT_Z_L_A (2Ch), OUT_Z_H_A (2Dh)

Z-axis acceleration data. The value is expressed in two's complement.

7.1.12 FIFO_CTRL_REG_A (2Eh)

Data is available as indicated by the status register:

7.1.8 STATUS_REG_A (27h)

Table 34. STATUS_A register

ZYXOR	ZOR	YOR	XOR	ZYXDA	ZDA	YDA	XDA

The bit we're interested in is *ZYXDA*, which tells us that all three values are available. For simplicity, we'll typically work with a blocking loop that waits for this flag to come TRUE. However, this could result in the program hanging if something is wrong with the I²C bus or the accelerometer IC.

Since the I²C bus only handles eight-bit values, we'll need to read two bytes to get a complete value. We can either do that by using our existing IIC0_Read() routine twice, or we can make a new routine that reads the two bytes and combines them into a single sixteen-bit value. If you want to go that route, here's one version of a working routine.

```
int IIC0_ReadD16(unsigned char cAddr, unsigned char cReg)
{
    int iData;
    char cRead;

    while(IIC0_IBSR & 0b00100000);      //wait for not busy flag
    IIC0_IBCR |= 0b00110000;            //micro as master, start transmitting

    IIC0_IBDR = cAddr & 0b11111110;     //place address on bus with /Write
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBDR = cReg;                   //locate desired register
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBCR |= 0b00000100;            //restart

    IIC0_IBDR = (cAddr | 0b00000001);   //place address on bus with Read
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBCR &= 0b11100111;            //read byte with ACK
    cRead = IIC0_IBDR;                  //fake read
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag
    iData = IIC0_IBDR*256;              //read first byte

    IIC0_IBCR |= 0b00001000;            //read byte with NAK
    cRead = IIC0_IBDR;                  //fake read
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBCR &= 0b11011111;            //stop
    IIC0_IBSR |= 0b00000010;            //clear flag
    iData +=IIC0_IBDR;                  //second byte

    return iData;
}
```

The data you read back will be formatted at 16-bit 2's complement, but it's actually 12-bit 2's complement left justified. In other words, the bits you're interested in are in the upper three nibbles. A quick way to fix this is to divide by 16, which will do an arithmetic shift left by 4, keeping track of the sign of the number.

3-Axis Magnetometer and Temperature Sensor

(Optional topic) Working with the earth's magnetic field is a surprisingly complex problem, as it is three-dimensional, and quite weak compared to the magnetic fields produced by electrical currents and magnets in equipment. If you need to make a proper compass, you will find it not to be a trivial exercise. In this course, we will simply look at what you need to do to get the values from the magnetometer – it's up to you as to what you want to do with them!

The magnetometer is a separate device inside the LSM303DLHC, at a different I^2C address and with different configurations. For example, there's no "Big-Endian/Little-Endian" issue: the registers are just MSbyte-LSbyte, in that order.

The magnetometer (and the temperature sensor) address is 0x1E, or 0b0011110r.

There are three control registers, all of which need our attention.

7.2.1 CRA_REG_M (00h)

Table 70. CRA_REG_M register

TEMP_EN	0(1)	0(1)	DO2	DO1	DO0	0(1)	0(1)

1. This bit must be set to '0' for correct operation of the device.

Table 71. CRA_REG_M description

TEMP _EN	Temperature sensor enable. 0: temperature sensor disabled (default), 1: temperature sensor enabled
DO[2:0]	Data output rate bits. These bits set the rate at which data is written to all three data output registers (refer to *Table 72*). Default value: 100

Table 72. Data rate configurations

DO2	DO1	DO0	Minimum data output rate (Hz)
0	0	0	0.75
0	0	1	1.5
0	1	0	3.0
0	1	1	7.5
1	0	0	15
1	0	1	30
1	1	0	75
1	1	1	220

A value of 0b00011000 in CRA sets the device up for a 75 Hz refresh rate, with no temperature sensor. 0b10011000 enables the temperature sensor, if you want it.

7.2.2 CRB_REG_M (01h)

Table 73. CRB_REG_M register

GN2	GN1	GN0	0[1]	0[1]	0[1]	0[1]	0[1]

1. This bit must be set to '0' for correct operation of the device.

Table 74. CRB_REG_M description

GN[2:0]	Gain configuration bits. The gain configuration is common for all channels (refer to *Table 75)*

Table 75. Gain setting

GN2	GN1	GN0	Sensor input field range [Gauss]	Gain X, Y, and Z [LSB/Gauss]	Gain Z [LSB/Gauss]	Output range
0	0	1	±1.3	1100	980	0xF800–0x07FF (-2048 to +2047)
0	1	0	±1.9	855	760	
0	1	1	±2.5	670	600	
1	0	0	±4.0	450	400	
1	0	1	±4.7	400	355	
1	1	0	±5.6	330	295	
1	1	1	±8.1	230	205	

The CRB register sets up the sensitivity of the magnetometer. Although the middle column claims to set the gains equally for all three channels, the next column over indicates that the Z channel has a different sensitivity. We'll go with the assumption that the Z column was put in the datasheet for a purpose, so the middle column must only apply to X and Y.

The units are unusual: LSB/gauss. This is the inverse of the step size, so the bigger the number, the more sensitive the device, as shown in the fourth column. Another thing that's unusual is the use of "gauss", a holdover from an old measurement system based on centimetres/grams/seconds (CGS) rather than the SI system's metre/kilogram/seconds (MKS) standard. In the MKS system, the tesla is used, and is 10,000 times bigger than a gauss.

For greatest sensitivity, we'll use a value of 0b00100000 for CRB. This makes the step-size for the X and Y channels 0.909 mG/step, or 90.9 nT/step. The Z channel sensitivity is 1.02 mG/step, or 102 nT/step.

7.2.3 MR_REG_M (02h)

Table 76. MR_REG_M register

0[1]	0[1]	0[1]	0[1]	0[1]	0[1]	MD1	MD0

1. This bit must be set to '0' for correct operation of the device.

The MR register defaults to 0b00000011, which puts the magnetometers into sleep mode. Values of 00 are needed in the *MD1* and *MD0* bits to put the device into "Continuous conversion" mode.

Once all of that is set up, the data can be read when the LSB of the status register goes HIGH:

7.2.7 SR_REG_M (09h)

Table 79. SR_REG_M register

--	--	--	--	--	--	LOCK	DRDY

The data is available as shown below:

7.2.4 OUT_X_H_M (03), OUT_X_L_M (04h)

X-axis magnetic field data. The value is expressed as two's complement.

7.2.5 OUT_Z_H_M (05), OUT_Z_L_M (06h)

Z-axis magnetic field data. The value is expressed as two's complement.

7.2.6 OUT_Y_H_M (07), OUT_Y_L_M (08h)

Y-axis magnetic field data. The value is expressed as two's complement.

This time, the data is right-justified! This means there's no need to shift the data to the right – it arrives as a proper 2's complement signed number, with the upper four bits stuffed appropriately.

If you want to use the temperature sensor and you've enabled it earlier, its values are available as shown below:

7.2.9 TEMP_OUT_H_M (31h), TEMP_OUT_L_M (32h)

Table 84. TEMP_OUT_H_M register

TEMP11	TEMP10	TEMP9	TEMP8	TEMP7	TEMP6	TEMP5	TEMP4

Table 85. TEMP_OUT_L_M register

TEMP3	TEMP2	TEMP1	TEMP0	–	--	--	–

Table 86. TEMP_OUT resolution

TEMP[11:0]	Temperature data (8 LSB/deg - 12-bit resolution). The value is expressed as two's complement.

Note that this is left-justified, so you'll need to divide by 16 to move it into proper position. The value is 2's complement signed, and has a resolution of "8 LSB/deg", or a step size of 0.125 ºC/step. That means that the three LSB's are fractional: 1/2, 1/4, and 1/8.

Device with 16-bit Internal Addresses (e.g. EEPROM) – Write and Read Functions

(Optional topic) In order to use the EEPROM on your board, you need 16-bit address versions of these two routines, shown below:

```
void IIC0_WriteA16(unsigned char cAddr, int iAddr, unsigned char cData)
{
    unsigned char cUpper = (unsigned char)(iAddr/256);
    unsigned char cLower = (unsigned char)(iAddr&0b0000000011111111);
    while(IIC0_IBSR & 0b00100000);      //wait for not busy flag
    IIC0_IBCR |= 0b00110000;            //micro as master, start transmitting

    IIC0_IBDR = (cAddr & 0b11111110);   //place address on bus with /Write
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBDR = cUpper;                 //upper unsigned char of address
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBDR = cLower;                 //lower unsigned char of address
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBDR = cData;                  //send data
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0 IBCR &= 0b11001111;            //stop transmitting, exit Master mode
    IIC0_IBSR |= 0b00000010;            //clear flag

}

unsigned char IIC0_ReadA16(unsigned char cAddr, int iAddr)
{
    unsigned char cData;
    unsigned char cUpper = (unsigned char)(iAddr/256);
    unsigned char cLower = (unsigned char)(iAddr&0b0000000011111111);

    while(IIC0_IBSR & 0b00100000);      //wait for not busy flag
    IIC0_IBCR |= 0b00110000;            //micro as master, start transmitting

    IIC0_IBDR = (cAddr & 0b11111110);   //place address on bus with /Write
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBDR = cUpper;                 //upper unsigned char of address
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBDR = cLower;                 //lower unsigned char of address
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBCR |= 0b00000100;            //restart

    IIC0_IBDR = (cAddr | 0b00000001);   //place address on bus with Read
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBSR |= 0b00000010;            //clear flag

    IIC0_IBCR |= 0b00001000;            //reading 1 unsigned char only
    IIC0_IBCR &= 0b11101111;            //receive unsigned char
    cData = IIC0_IBDR;                  //not actually -- starts the process
    while(!(IIC0_IBSR & 0b00000010));   //wait for flag
    IIC0_IBCR &= 0b11011111;            //stop
    IIC0_IBSR |= 0b00000010;            //clear flag
    cData = IIC0_IBDR;                  //for real this time

    return cData;
}
```

If you want to transfer an entire array of bytes between the micro and the EEPROM, you would want to make versions of these routines that can write or read a bunch of bytes sequentially. The EEPROM is designed to operate in a special paging mode, in which a full "page" of 128 bytes is read from or written to the device. This requires starting at a page-delineated address, and also involves pointers to string arrays for the microcontroller. The information as to how to manage "Page Write" or "Page Read" transfers is available in the data sheet for the 24AA512 EEPROM. This topic goes beyond the scope of this course, but the following gives you a starting point. Multiple reads and writes involve issuing an I^2C ACK signal between each byte transferred, with a NAK signal at the end; unlike normal single byte transfers, which only issue NAK signals. With the 24AA512, since it operates more slowly than the I^2C bus, we need to wait until it's ready for the next byte. The only way to handle this is to send the byte and see if it's acknowledged; if it isn't, we send the

byte again and keep doing so until it is acknowledged. With all of this hand-shaking, the possibility of hanging up the program waiting for a flag, an ACK or a NAK looms large. It's best to write software that will only wait so long, then returns an error code to indicate that the system has failed. As previously mentioned, Simon Walker has written an extensive set of I^2C library components that handle multiple reads and writes, along with page reads and writes, all of which have escape routes in case of failure. If you find yourself using I^2C devices on a regular basis, you should talk him about how to use his library components.

I^2C Reliability Measures

As indicated in the previous discussion, you probably discovered, with the simple routines created for your library, that the I^2C bus sometimes goes insane (mostly when you're troubleshooting, as it's pretty dependable in normal operation), and your program will hang up waiting for a flag, often in IIC0_IBSR. A partial solution to this, which you would see implemented in Simon Walker's library, is to put a counter into the *while(!(IIC0_IBSR&0b00000010))* loop so that after a certain number of tries, say 5000 or so, you exit the loop and return an error code. A typical error code is 0b11111111 (i.e. 0xFF), which, as a signed number, is –1, and as a Boolean value, is ~0. You may also want to come up with more sophisticated "try-catch" routines that allow your program to continue operating when the I^2C bus goes down, including prompting an operator to cycle the power on the board, if necessary.

Parting Words

You have now touched on some of the capabilities of a very powerful microcontroller and a selection of associated peripherals that were built into your microcontroller kit. You've learned, with varying levels of proficiency, how to use a fairly 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!