Lab 2e. Fixed-point Serial I/O Device Driver

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- To introduce the lab equipment;
- To familiarize yourself with Metrowerks CodeWarrior for the 6812;
- To organize a device driver of useful serial port I/O routines.

Review
- "How to program…" section located at the beginning of this laboratory manual,
- Chapter 14 of the M68HC812A4 Technical Summary, or the **`S12SCIV2.pdf`** (9S12C32), in particular look up all the I/O ports associated with your SCI
- Read "Developing C Programs using ICC11/ICC12/Hiware" on the TExaS CD or at http://www.ece.utexas.edu/~valvano/embed/toc1.htm
- Valvano Sections 1.1, 1.5, 1.7, 2.1, 2.2, 2.3, 2.5, and 2.7.2 from the book Embedded Microcomputer Systems: Real Time Interfacing,
- Valvano Excerpts from Introduction to Embedded Microcomputer Systems: Motorola 6811 and 6812 Simulation located in the lab manual.

Starter files
- **`SCI`** project

Background

The objective of this lab is to introduce the programming environment and extend a device driver of useful I/O routines that will be used in the subsequent labs. A device driver is a set of functions that facilitate the usage of an I/O port. In particular, the **`SCI.H`** and **`SCI.C`** files constitute the device driver for the 6812 SCI port. An important factor in device driver design is to separate the policies of the interface (how to use the software is defined in the **`SCI.H`** file) from the mechanisms (how the programs are implemented, which is defined in the **`SCI.C`** file.) In this lab, you will be using, then extending software that performs I/O using the 6812 serial port. Both the MC68HC812A4 and 9S12C32 have serial ports to implement communication between the 6812 board and COM2 of the PC, as shown in Figure 2.1. PS1 is the **TxD** Transmitter and PS0 is the **RxD** Receiver. Both systems have RS232 drivers, which are connected via a 9-pin serial cable to the COM2 of the PC. This lab involves the implementation of specific software routines explicitly defined in this lab assignment. You are encouraged to modify the specific names of the routines, the calling sequences, and the parameter formats, as long as the general concept is implemented. In this lab, all I/O uses the 6812 serial port.

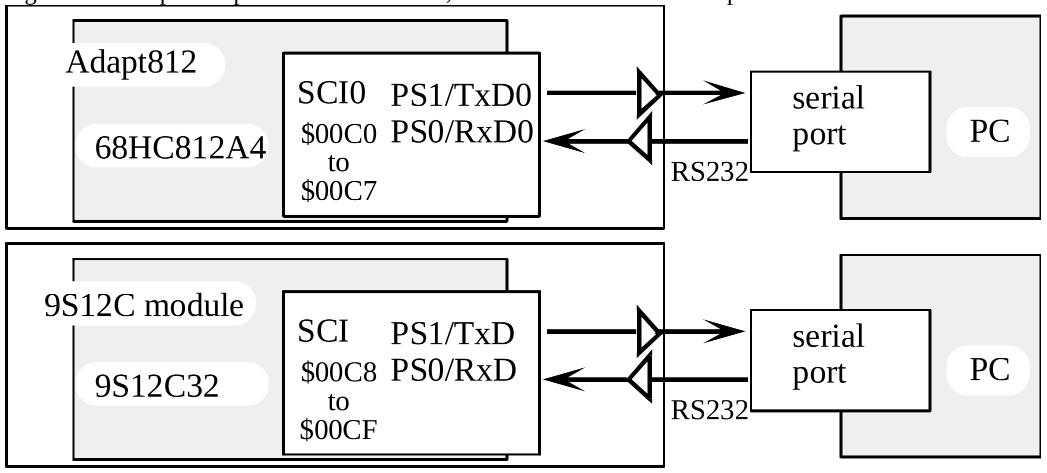

Figure 2.1. The serial port hardware connects the 6812 SCI0 with the PC COM2.

The MC68HC812A4 has two serial communication channels, called SCI0 and SCI1. We will not be using SCI1 at all this semester. On the MC68HC812A4, serial port SCI0 is located at addresses 0x00C0 to 0x00C7. Connector J4 on the Adapt812 board, via a serial DB9 connector, will be connected to the PC COM2 port. Recall that the PC COM1 port is connected to the Kevin Ross BDM and is used for downloading and debugging.

The 9S12C32 has only one serial communication channel, called SCI. The 9S12C32 serial port is located at addresses 0x00C8 to 0x00CF. This one serial port will be used for both the on-chip debugger and your communication software.

Verify the baud rate of your 6812 software matches the baud rate of the terminal program running on the PC. To transmit information from the PC to your software, the operator types on the PC keyboard with focus in the terminal program, and your 6812 software calls the function **SCI_InChar**. The following sequence of events occurs as one character is communicated:

the PC keyboard software encodes the typed key into ASCII,
the PC terminal window transmits the 8-bit ASCII code out the COM2 serial port
  at 38400 bits/sec for the MC68HC812A4 or
  at 19200 bits/sec for the 9S12C32,
the 6812 SCI port accepts the serial transmission,
the **RDRF** (receive data register full) flag is set,
the **SCI_InChar** function waits for **RDRF** to be set
  RDRF is bit 5 of **SC0SR1** on the MC68HC812A4 or
  RDRF is bit 5 of **SCISR1** on the 9S12C32,
the ASCII character is read from the receive serial data register
  SC0DRL on the MC68HC812A4 or
  SCIDRL on the 9S12C32,
the ASCII code is returned.

If you type one or two characters, it doesn't matter which occurs first: your software calling **SCI_InChar**, or the operator typing on the keyboard. If your software calls first, it will wait for **RDRF**. If the operator types first, up to two ASCII codes will be safely stored in the 6812 SCI hardware (one in the shift register, one in the data register).

```
// MC68HC812A4 version
#define RDRF 0x20
unsigned char SCI_InChar(void){
  while((SC0SR1 & RDRF) == 0){};
  return(SC0DRL);
}
```

```
// 9S12C32 version
#define RDRF 0x20
unsigned char SCI_InChar(void){
  while((SCISR1 & RDRF) == 0){};
  return(SCIDRL);
}
```

To transmit a character from the 6812 to the PC, your software calls **SCI_OutChar**. The following sequence of events occurs:

the function **SCI_OutChar** waits for **TDRE** (transmit data register empty) to be set,
  TDRE is bit 7 of **SC0SR1** on the MC68HC812A4 or
  TDRE is bit 7 of **SCISR1** on the 9S12C32,
the function **SCI_OutChar** writes the 8-bit ASCII code to the transmit data register
  SC0DRL on the MC68HC812A4 or
  SCIDRL on the 9S12C32,
the SCI hardware transmits the data to the PC COM2 serial port
  at 38400 bits/sec for the MC68HC812A4 or
  at 19200 bits/sec for the 9S12C32,
the COM2 port on the PC accepts the serial transmission,
the PC software displays the ASCII code on the terminal window.

```
// MC68HC812A4 version
#define TDRE 0x80
void SCI_OutChar(unsigned char data){
  while((SC0SR1 & TDRE) == 0){};
  SC0DRL = data;
}
```

```
// 9S12C32 version
#define TDRE 0x80
void SCI_OutChar(unsigned char data){
  while((SCISR1 & TDRE) == 0){};
  SCIDRL = data;
}
```

Notice that **SCI_InChar** reads from the *data register* and **SCI_OutChar** writes to the *data register*. There are actually two separate serial data registers at the same address (the receiver data register is read-only, and the transmit data register is write-only.) In particular, it is possible for data to flow in both directions at the same time. There are actually two shift registers too. The two flags, **TDRE** and **RDRF**, specify the status of the corresponding data registers.

This lab will use the string input and string output functions

```
//------------------------SCI_InString-----------------------
// This function accepts ASCII characters from the serial port
void SCI_InString(unsigned char *string, unsigned short max) {
unsigned int length=0;
unsigned char character;
  character = SCI_InChar();
  while(character!=CR){
    if(character==BS){
      if(length){
        string--;
        length--;
        SCI_OutChar(BS);
      }
    }
    else if(length<max){
      *string++ = character;
      length++;
      SCI_OutChar(character);
    }
    character = SCI_InChar();
  }
  *string = 0;
}
//------------------------SCI_OutString-----------------------
// Output String (NULL termination)
void SCI_OutString(unsigned char *pt){
  while(*pt){
    SCI_OutChar(*pt);
    pt++;
  }
}
```

Maintenance Tip: *Even though the machine will process data in binary, we can specify numbers in many formats (e.g., binary, decimal, hexadecimal, etc.) Use the format that makes your software easiest to understand. There is no one format that is best for all situations.*

We will use fixed-point numbers when we wish to express values in our software that have noninteger values. A **fixed-point number** contains two parts. The first part is a **variable integer**, called **I**. This integer may be signed or unsigned. An unsigned fixed-point number is one that has an unsigned variable integer. A signed fixed-point number is one that has a signed variable integer. The **precision** of a number is the total number of distinguishable values that can be represented. The precision of a fixed-point number is determined by the number of bits used to store the variable integer. On the 6812, we typically use 8 bits or 16 bits. Extended precision can be implemented, but the execution speed will be slower because the calculations will have to be performed using software algorithms rather than with hardware instructions. This integer part is saved in memory and is manipulated by software. These manipulations include but are not limited to add, subtract, multiply, divide, convert to BCD, convert from BCD. The second part of a fixed-point number is a **fixed constant**, called D. This value is fixed, and can not be changed. The fixed constant is not stored in memory. Usually we specify the value of this fixed content using software comments to explain our fixed-point algorithm. The value of the fixed-point number is defined as the product of the two parts:

fixed-point number ≡ **I**•D

The **resolution** of a number is the smallest difference in value that can be represented. In the case of fixed-point numbers, the resolution is equal to the fixed constant (Δ). Sometimes we express the resolution of the number as its units. For example, a decimal fixed-point number with a resolution of 0.001 volts is really the same thing as an integer with units of mV. When interacting with humans it is convenient to use **decimal fixed-point**. With decimal fixed-point the fixed constant is a power of 10.

decimal fixed-point number = $\mathbf{I} \bullet 10^{\mathbf{m}}$ for some fixed integer **m**

Again, the integer **m** is fixed and is not stored in memory. Decimal fixed-point will be easy to display, while **binary fixed-point** will be easier to use when performing mathematical calculations. With binary fixed-point the fixed constant is a power of 2.

binary fixed-point number = $\mathbf{I} \bullet 2^{\mathbf{n}}$ for some fixed integer **n**

In the next example, we will develop the equations that a 6812 would need to implement a digital scale. Assume the range is 0 to 3 cm, and the system uses the 6812's ADC to perform the measurement. The 8-bit ADC analog input range is 0 to +5 V, and the ADC digital output varies 0 to 255 respectively. Let **x** be the distance to be measured in cm, $\mathbf{V_{in}}$ be the analog voltage and **N** be the 8-bit digital ADC output, then the equations that relate the variables are

$V_{in} = 5 * N/255 = 0.019607843 * N$ and $x = 3 * V_{in}/5 = 0.6 * V_{in}$

thus

$x = 3 * N/255 = 0.0117647 * N$

From this equation, we can see that the smallest change in distance that the ADC can detect is about 0.01 cm. In other words, the distance must increase or decrease by 0.01 cm for the digital output of the ADC to change by at least one bit. It would be inappropriate to save the distance as an integer, because the only integers in this range are 0, 1, 2 and 3. Since the 6812 does not support floating point, the distance data will be saved in fixed-point format. Decimal fixed-point is chosen because the distance data for this voltmeter will be displayed for a human to read. A fixed-point resolution of Δ=0.01 cm should be chosen because it best matches the resolution determined by the hardware. Table 2.1 shows the performance of the system with the resolution of Δ=0.01 cm. The table shows us that we need to store the variable part of the fixed-point number in a 16-bit unsigned variable.

x distance in cm	$\mathbf{V_{in}}$ (V) Analog input	**N** ADC output	**I** (Δ= 0.01cm) variable part
0	0.00	0	0
0.01	0.02	1	1
0.60	1.00	51	60
1.50	2.50	128	150
3.00	5.00	255	300

Table 2.1. Performance data of a microcomputer-based distance measurement.

It is very important to carefully consider the order of operations when performing multiple integer calculations. There are two mistakes that can happen. The first error is **overflow**, and it is easy to detect. Overflow occurs when the result of a calculation exceeds the range of the number system. The following fixed-point calculation, although mathematically correct, has an overflow bug

```
    I = (300*N)/255;
```

because when `N` is greater than 218, `300*N` exceeds the range of a 16-bit unsigned integer. If possible, we try to reduce the size of the integers. In this case, an equivalent calculation can be performed without overflow

```
    I = (60*N)/51;
```

No overflow occurs with this second equation using unsigned 16-bit math, because the maximum value of `60*N` is 15300. If you can not rework the problem to eliminate overflow, the best solution is to use promotion. Promotion is the process of performing the operation in a higher precision. For example, in C we cast the input as `unsigned long`, and cast the result as `unsigned short`

```
    I = (unsigned short)((300*(unsigned long)N)/255);
```

When speed is important we can implement the calculation in assembly

```
ldd  N
ldx  #300
emul      32-bit Y:D is 300*N
ldx  #255
ediv      16-bit Y is (300*N)/255
sty  I
```

The other error is called **drop out**. Drop out occurs after a right shift or a divide, and the consequence is that an intermediate result looses its ability to represent all of the values. It is very important to divide last when performing multiple integer calculations. If you divided first, e.g.,

```
I=60*(N/51);
```

then the values of I would be only 0, 60, 120, … or 300. Since the integer divide ignores the remainder, we can perform the following fixed-point calculation to convert N into I.

```
I = (60*N+25)/51;
```

The addition of "25" has the effect of rounding to the closest integer. The value 25 is selected because it is about one half of the divisor. For example, the calculation (60*N)/51=4 for N=4, whereas the "(60*4+25)/51" calculation yields the better answer of 5. The display algorithm is simple:

1) display (`I/100`) as a single digit value
2) display a decimal point
3) display (`I%100`) as a two digital value
4) display the units "`cm`"

When adding or subtracting two fixed-point numbers with the same D, we simply add or subtract their integer parts. First, let x,y,z be three fixed-point numbers with the same D. To perform z=x+y, we simply calculate K=I+J. Similarly, to perform z=x-y, we simply calculate K=I-J. When adding or subtracting fixed-point numbers with different fixed parts, then we must first convert two the inputs to the format of the result before adding or subtracting. This is where binary fixed-point is more convenient, because the conversion process involves shifting rather than multiplication/division.

In this next example, let x,y,z be three binary fixed-point numbers with the different Ds. In particular, we define x to be $\mathbf{I}\bullet 2^{-5}$, y to be $\mathbf{J}\bullet 2^{-2}$, and z to be $\mathbf{K}\bullet 2^{-3}$. To convert x, to the format of z, we divide I by 4 (right shift twice). To convert y, to the format of z, we multiply J by 2 (left shift once). To perform z=x+y, we calculate

K=(I>>2)+(J<<1)

For the general case, we define x to be $\mathbf{I}\bullet 2^{n}$, y to be $\mathbf{J}\bullet 2^{m}$, and z to be $\mathbf{K}\bullet 2^{p}$. To perform any general operation, we derive the fixed-point calculation by starting with desired result. For addition, we have z=x+y. Next, we substitute the definitions of each fixed-point parameter

$$\mathbf{K}\bullet 2^{p}=\mathbf{I}\bullet 2^{n}+\mathbf{J}\bullet 2^{m}$$

Lastly, we solve for the integer part of the result

$$\mathbf{K}=\mathbf{I}\bullet 2^{n-p}+\mathbf{J}\bullet 2^{m-p}$$

For multiplication, we have z=x•y. Again, we substitute the definitions of each fixed-point parameter

$$\mathbf{K}\bullet 2^{p}=\mathbf{I}\bullet 2^{n}\bullet\mathbf{J}\bullet 2^{m}$$

Lastly, we solve for the integer part of the result

$$\mathbf{K}=\mathbf{I}\bullet\mathbf{J}\bullet 2^{n+m-p}$$

For division, we have z=x/y. Again, we substitute the definitions of each fixed-point parameter

$$\mathbf{K}\bullet 2^{p}=\mathbf{I}\bullet 2^{n}/\mathbf{J}\bullet 2^{m}$$

Lastly, we solve for the integer part of the result

$$\mathbf{K}=\mathbf{I}/\mathbf{J}\bullet 2^{n-m-p}$$

Again, it is very important to carefully consider the order of operations when performing multiple integer calculations. We must worry about overflow and drop out. In particular, in the division example, if (n-m-p) is positive then the left shift (**I**•2^{n-m-p}) should be performed before the divide (/**J**).

We can use these fixed-point algorithms to perform complex operations using the integer functions of our 6812. For example, consider the following digital filter calculation.

```
y = x -0.0532672*x1 + x2 + 0.0506038*y1-0.9025*y2;
```

In this case, the variables **`y`**, **`y1`**, **`y2`**, **`x`**, **`x1`**, and **`x2`** are all integers, but the constants will be expressed in binary fixed-point format. The value -0.0532672 will be approximated by -14•2^{-8}. The value 0.0506038 will be approximated by 13•2^{-8}. Lastly, the value -0.9025 will be approximated by -231•2^{-8}. The fixed-point implementation of this digital filter is

```
y = x + x2 + (-14*x1+13*y1-231*y2)/256;
```

Preparation (do this before your lab period)

There is no hardware for this lab. Please download from the class web site the SCI project:

`SCI.H`	header file for serial port device driver
`SCI.C`	implementation file for serial port device driver
`main.C`	main program that tests the device driver

A "syntax-error-free" hardcopy listing for procedure Part (2) of the software is required as preparation. The TA will check this off at the beginning of the lab period. You are required to do your editing before lab. The debugging will be done during lab. Document clearly the operation of the routines. Create a project that includes five files:

`Fix.h`	prototypes for the functions **`Fix_In`**, **`Fix_Out`**,
`Fix.c`	implementations of the two functions,
`main.c`	a main program that tests the new functions that you wrote.
`SCI.H`	unchanged from the example SCI project
`SCI.C`	unchanged from the example SCI project

Separate driver functions (**`Fix.h`**, **`Fix.c`**) from the software that tests the driver (**`main.c`**), similar to the way **`SCI.C`** is separated in the SCI project. The comments included in **`Fix.h`** are intended for the client (programmers that will use your functions.) The comments included in **`Fix.c`** are intended for the coworkers (programmers that will debug/modify your functions.)

The fixed-point constant is 0.01. The full-scale range is from 0 to 655.34. The **`Fix_In`** function should convert an ASCII string into the integer part of the fixed-point number. The input/output specifications of **`Fix_In`** are illustrated in the Table 2.2. Your test program will call **`SCI_InString`** to get ASCII strings from the operator. Your **`Fix_In`** program does not perform any SCI input or output.

Input String	Actual value	Return parameter of `Fix_In()`
"0"	0.00	0
".2"	0.20	20
"5.05"	5.05	5050
"10.720"	10.72	1072
"0.0023"	0.00	0
"142.595"	142.60	14260
"142.604"	142.60	14260
"601"	601.00	60100
"655.34"	655.34	65534

Table 1.2. Examples of 16-bit unsigned decimal fixed point input with ?=10^{-2}.

`Fix_In` should round to the closest fixed-point result (e.g., 1.595 rounds to 1.60 and 1.604 also rounds to 1.60). You may use 16-bit unsigned math. In particular, some numbers like 1.2345678 might be considered illegal because they cause overflow of intermediate results. In the comments of the software, please discuss why you chose your particular implementation method over the other available choices. You are free to modify the prototypes in any way you feel is appropriate. You must check for illegal inputs, returning 65535, which is defined as an illegal number. The main program then can decide what to do on an illegal input. For example,

```
unsigned short GetFix(void){ // possible user program that uses your driver
unsigned short result;       // fixed-point resolution 0.01
unsigned char buffer[16];
  do{
    SCI_OutChar(CR); SCI_OutString("Input a number from 0 to 655.34 ");
    SCI_InString(buffer,15); // get a string from the operator
    result = Fix_In(buffer); // convert string to fixed point
  while(result ==65535);     // repeat until legal
  return result;
}
```

Your second function converts the integer part of a fixed-point number into an ASCII string. This function also performs no SCI input/output. The input/output specifications for **`Fix_Out`** are illustrated in the Table 2.3.

Input to **`Fix_Out`**	Output of **`Fix_Out`**
`$0000 0`	`"0.00"`
`$0032 50`	`"0.50"`
`$008D 141`	`"1.41"`
`$00FA 250`	`"2.50"`
`$0781 1921`	`"19.21"`
`$3039 12345`	`"123.45"`
`$7FFF 32767`	`"327.67"`
`$FFFE 65534`	`"655.34"`
`$FFFF 65535`	`"***.**"`

Table 2.3. Examples of 16-bit unsigned decimal fixed point output with ?=10^{-2}.

You are free to develop a testing file in whatever style you wish. This main program has three important roles. First, you will use it to test all the features of your program. Second, a judge in a lawsuite can subpoena this file. In a legal sense, this file documents to the world the extent to which you verified the correctness of your program. When one of your programs fails in the marketplace, and you get sued for damages, your degree of liability depends on whether you took all the usual and necessary steps to test your software, and the error was unfortunate but unforeseeable, or whether you rushed the product to market without the appropriate amount of testing and the error was a foreseeable consequence of your greed and incompetence. Third, if you were to sell your software package (**`Fix.c`** and **`Fix.h`**), your customer can use this file to understand how to use your package, its range of functions, and its limitations. For example

```
#include "SCI.h"
#include "fix.h"
unsigned short n,m;     // fixed-point resolution 0.01
unsigned char buffer[16];
void main(void){
  SCI_Init(38400);   // initialize SCI and set the baud rate
  while(1){
    do{
      SCI_OutChar(CR);
      SCI_OutString("Input a fixed-point number from 0 to 655.34 ");
      SCI_InString(buffer,15); // get a string from the operator
      n = Fix_In(buffer);      // convert string to fixed point
    while(n==65535);           // repeat until legal
    SCI_OutString("The integer part is ");
    SCI_OutUDec(n); SCI_OutChar(CR);   // display integer part
    SCI_OutString("Input an integer from 0 to 65535 ");
    m = SCI_InUDec();    // get am integer from the operator
    Fix_Out(m,buffer);   // convert fixed-point to string
    SCI_OutString("The fixed-point value is ");
    SCI_OutString(buffer); SCI_OutChar(CR);   // display value
  }
}
```

Procedure (do this during your lab period)
Part (1) Experiment with the different features of Metrowerks and the debugger. Familiarize yourself with the various options and features available in the editor/assembler/terminal. Edit, compile, download, and run the SCI project working through all aspects of software development. In particular, learn how to:

- create hard copy listings of your source code;
- open and read the assembly listing files;
- save your source code on floppy disk;
- compile, download, and execute a program.

Part (2) Debug your device driver (`Fix.h Fix.c main.c`).

Part (3) Draw a call graph showing software modules (`main` `Fix` and `SCI`), and the `SCI` hardware module. There should be no call arrow from `Fix` to `SCI`. Separating `Fix` from `SCI` will allow you to use these routines in a later lab when the input comes from a keyboard attached directly to the 6812 and the output goes to a LCD display also attached directly to the 6812. There should be no direct call arrow from `main` to the `SCI` hardware module.

Deliverables (exact components of the lab report)
A) Objectives (1/2 page maximum)
B) Hardware Design (none for this lab)
C) Software Design (no software printout in the report)
Call graph of the system
D) Measurement Data (none for this lab)
E) Analysis and Discussion (1 page maximum)

Checkout (show this to the TA)
You should be able to demonstrate correct operation of each routine:

- show the TA you know how to observe global variables and I/O ports using the debugger;
- demonstrate to the TA you know how to observe assembly language code;
- verify proper input/outputs of the I/O functions;
- verify the proper handling of illegal formats;
- demonstrate your software does not crash.

Your software files will be copied onto the TA's zip drive during checkout.

Hints
1) Run the existing SCI project first, then make small changes and test. Make backups of the previous versions, so that when you add something that doesn't work, you can go back to a previous working version and try a new approach. Please add documentation that makes it easier to change and use in the future. Your job is to organize these routines to facilitate subsequent laboratories.
2) You must always look at the assembly language created by the compiler to verify the appropriate function. We are using a new compiler, and bugs have been found in earlier versions. I.e., sometimes the compiler will not create the proper executable code. If you think you've found a bug, email the source and assembly listing to the TA explaining where the bug is.
3) The SCI, like most I/O peripherals, must be initialized before they can be used. For a serial device, we usually specify the baud rate, number of data bits, type of parity if desired, the number of stop bits, and the synchronization method (gadfly or interrupts.) The 6812 and PC must be operating at the same baud rate.
4) You may find it useful read to the calculator and position measurement labs to get a feel for the context of the fixed-point routines you are developing. In particular, you should be able to use these functions in these two labs without additional modification.