Lab 1. Serial Port 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 ICC12;
- To organize a device driver of useful serial port I/O routines.

Review
- Introduction chapter of this laboratory manual,
- Chapter 14 of the M68HC812A4 Technical Summary, in particular look up all the I/O ports associated with SC0: SC0BDH, SC0BDL, SC0CR1...,
- The 6812 Cross compiler documentation, ImageCraft ICC12 manual,
- 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,

Starter files
- HC12.H, SCI12.C, SCI12.H, SCIDEMO.C

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 SCI12.H and SCI12.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 programs is defined in the SCI12.H file) from the mechanisms (how the programs are implemented, which is defined in the SCI12.C file.) In this lab, you will be using, then extending software that performs I/O using the 6812 serial port. The 6812 Serial Port 0 (PS1/TxD0 Transmitter and PS0/RxD0 Receiver) is the interface port that implements the serial communication between the Adapt812 board and COM2 of the PC. This lab involves the implementation of specific routines explicitly defined in this lab assignment. Some of the routines are already written. If a routine already exists, your job is to document it and integrate it. I.e., make it easy to use. You should create a serial port device driver that will be easy to add to your subsequent programs. 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 serial I/O uses the 6812 Serial Port 0.

The 6812 has a two-channel serial communications interface. On the 68HC812 Serial Port is located at addresses 0x00C0 to 0x00CF. Connector J4, the 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.

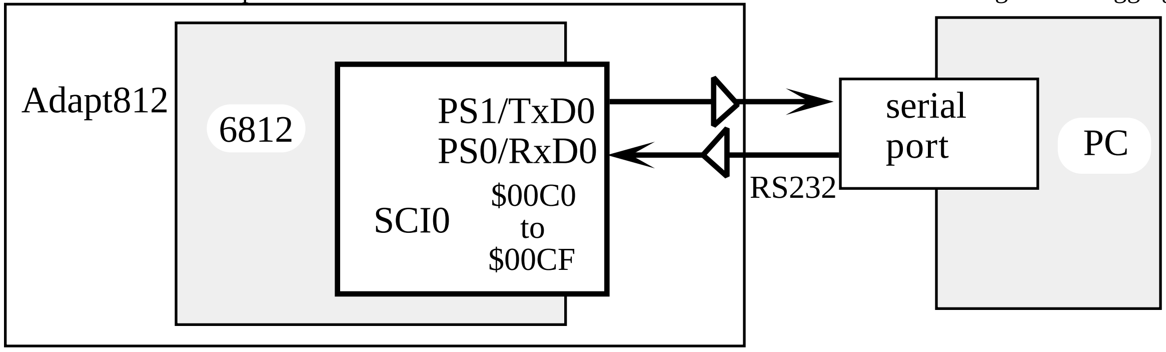

Figure 1.1. The serial port hardware connects the 6812 SCI0 with the PC COM2.

To receive a character from the PC using serial channel A (operator types on the PC keyboard), your software must wait until **RDF** (receive data full) is set. **RDF** is located in Bit 5 of the `SC0SR1` register.

```
char InChar(void){
  while ((SC0SR1 & RDF) == 0);
  return(SC0DRL);}
```

To transmit a character from the 6812 to the PC using serial channel A (data shows up on the terminal window of the ICC12 application), your software must wait until **TBE** (transmit buffer empty) is set. **TBE** is located in bit 7 of the `SC0SR1` register.

```
void OutChar(char data){
  while ((SC0SR1 & TBE) == 0);
  SC0DRL = data;
}
```

We will illustrate the conversion between ASCII strings and binary numbers by developing high level functions in C. The purpose of introducing C programs is to introduce the conversion process. Later we will develop equivalent software in assembly language. In the first example, let `Data` be a fixed length string of 3 ASCII characters. Each entry of `Data` is an ASCII character 0 to 9. Let `Data[0]` be the ASCII code for the hundred's digit, `Data[1]` be the ten's digit and `Data[2]` be the one's digit. Let `N` be an unsigned integer. We will also need indices, `i,j`. In C we could define them:

```
char Data[3];
unsigned int i,j,temp,N;
```

To convert this string of 3 decimal digits into binary we can simply calculate

```
       N = 100*(Data[0]-0x30) + 10*(Data[1]-0x30) + (Data[2]-0x30);
```

that could also be calculated

```
       N = (Data[2]-0x30) + 10*((Data[1]-0x30) + 10*(Data[0]-0x30)) ;
```

If `Data` were a string of 5 decimal digits we could put the above function into a loop

```
       N = 0;
       for (i=0; i<5 ;i++) N = 10*N + (Data[i]-0x30);
```

If `Data` were a variable length string of ASCII characters terminated with a null character (value=0), we could convert it to binary using

```
//---------------------------Str2UDec----------------------------
// Convert ASCII string to unsigned decimal
unsigned int Str2UDec(unsigned char data[]){
  unsigned int N = 0;   // the number
  unsigned int i = 0;   // index into
  while (Data[i]!=0) {
    N = 10*N + (Data[i]-0x30);
    i++; }
  return N;}
```

In the following example, the function `InChar()` returns an ASCII character from the input device. The function `InUDec()` will accept characters from the input device until a carriage return in typed.

```
//---------------------------InUDec----------------------------
// InUDec accepts ASCII input in unsigned decimal format
//     and converts to a 16 bit unsigned number with a maximum value of 65535
// If you enter a number above 65535, it will truncate without reporting the error
// Backspace will remove last digit typed
unsigned int InUDec(void){   unsigned number=0, length=0;
  unsigned char character;
  while ((character=InChar())!=CR){    // accepts until carriage return input
// The next line checks that the input is a digit, 0-9.
// If the character is not 0-9, it is ignored and not echoed
    if(character>='0' && character<='9' ) {
      number=10*number+character-'0';   // this line overflows if above 65535
      length++;
      OutChar(character);
    }
// If the input is a backspace, then the return number is
// changed and a backspace is outputted to the screen
    else if (character==BS && length){
      number/=10;
      length--;
      OutChar(character);
    }
  }
  return number;
}
```

To convert an unsigned integer into a fixed length string of ASCII characters, we could use the integer divide. Assume N is an unsigned integer less than or equal to 999:

```
       Data[0] = N/100 + 0x30;
       N = N % 100;                /* N is now between 0 and 99 */
       Data[1] = N/10 + 0x30;
       N = N % 10;                 /* N is now between 0 and 9 */
       Data[2] = N + 0x30;
```

To convert an unsigned integer into a variable length string of ASCII characters, we convert the digits in reverse order, then switch them:

```
//----------------------UDec2Str -------------------------------------
// Unsigned decimal, N, to ASCII string, Data
unsigned char Data[10];
void UDec2Str(unsigned int N){ unsigned int i,j,uN; unsigned char temp;
  uN=N;     // a copy that we can change
  i = 0;    // i will contain the total number of ASCII characters
  do {
    Data[i] = uN % 10 + 0x30;   // Start with the one’s digit
    uN = uN / 10 ;
    i++; }
  while (uN != 0 );
  for (j=0 ; j< i/2 ; j++ ) {
    temp=Data[j];     // Reverse order
    Data[j]=Data[i-1-j];
    Data[i-1-j]=temp; }
  Data[i]=0; }         // Null terminated ASCII string
```

The following example performs the same unsigned conversion, but uses recursion. The function `OutCh()` send one ASCII character to the output device. Notice that the function calls itself

```
//----------------------OutUDec------------------------------------
// Output a 16 bit number in unsigned decimal format
// Variable format 1 to 5 digits with no space before or after
// This function uses recursion to convert a decimal number of
//   unspecified length as an ASCII string
void OutUDec(unsigned int number){
  if (number>=10){
    OutUDec(number/10);
    OutUDec(number%10);}
  else
    OutChar(number+'0');
}
```

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**. 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 may be stated in alternatives (n_{alt}), in bits (n_{bits}), in bytes (n_{bytes}), or in decimal digits (n_{digits}.)

$$n_{alt} = 2^{n_{bits}} = 10^{n_{digits}}$$
$$n_{bits} = 8 \bullet n_{bytes}$$

The precision of a fixed-point number (n_{bytes}) is determined by the number of bytes used to store the variable integer. The precision may be one or more bytes. 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**. This value is fixed, and can not be changed. The fixed constant is not stored in memory. Usually we add this fixed content to our 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 ≡ (variable integer)•(fixed constant)

The **resolution** of a number (Δ) is the smallest difference that can be represented. In the case of fixed-point numbers, the resolution is equal to the fixed constant.

Δ = fixed constant

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. Sometimes it may be convenient to use **decimal fixed-point**. Fixed decimal point could be used in a microcomputer based digital voltmeter. The integer **m** is fixed and is not stored in memory.

decimal fixed-point number = (variable integer) • 10^m

Δ = 10^m for some integer m

Assume that this digital voltmeter had a 16 bit 2's complement ADC converter such that

ADC Analog Input (volts)	ADC Digital Output
-10.000	-10000
-5.000	-5000
0.000	0
0.001	1
+5.000	5000
+10.000	10000

The voltage could then be conveniently stored in the computer as a 16 bit decimal fraction, with m=-3, Δ=0.001. The variable integer part would simply be the ADC digital output.

Variable Integer	Fixed Constant	Value(volts)
-10000	0.001	-10.000
-1	0.001	-0.001
0	0.001	0.000
1	0.001	0.001
2	0.001	0.002
3	0.001	0.003
250	0.001	0.250
500	0.001	0.500
9999	0.001	9.999

Preparation (do this before your lab period)

There is no hardware for this lab. Please download from the class web site the three starter files:

SCI12.H	header file for serial port device driver
SCI12.C	implementation file for serial port device driver
SCIDEMO.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. Separate these programs from the rest of software (the routines that test these functions) into a *device driver* so that you can use them in other labs.

Procedure (do this during your lab period)

Part (1) Experiment with the different features of ICC12 and the DOS-level program DL12.EXE. Familiarize yourself with the various options and features available in the editor/assembler/terminal. Edit/compile/download/run the starter program working through all aspects of software development. In particular, learn how to:

• create hard copy listings of your source and object code;
• save your source code on floppy disk;
• compile, download, and execute a program.

Part (2) Extend the device driver by adding the following two functions.

```
int InFDec(void);  // Reads in an signed 16 bit binary fixed point number
void OutFDec(int); // Output as signed 16 bit binary fixed point number
```

You will use these two functions in subsequent labs to input/output temperature data. The fixed-point constant is 1/2. The `InFDec()` function should provide for flexible operation. The input/output specifications of these two functions are illustrated in the following table.

User types in to `InFDec`	Actual value or Output of `OutFDec`	Output of `InFDec` or Input to `OutFDec`
0	+0.0	\$0000 0
+25	+25.0	\$0032 50
70.5	+70.5	\$008D 141
124.87	+125.0	\$00FA 250
+10.74	+10.5	\$0015 21
+10.76	+11.0	\$0016 22
-0.5	-0.5	\$FFFF -1
-12.4	-12.5	\$FFE7 -25
-12.6	-12.5	\$FFE7 -25
-12.9	-13.0	\$FFE6 -26
-55	-55.0	\$FF92 -110

Table 1.1. Examples of 16 bit signed binary fixed point with $\Delta=2^{-1}$.

Notice that `InFDec` handles an optional + sign, and decimal point. `InFDec` also rounds the input to the closest fixed-point result (e.g., 10.7 goes to 10.5 and 10.8 goes to 11.0). You may assume the input the user enters (neglecting the decimal point) ranges from -32768 to +32767. In other words 10.1234 will not be valid, while 10.123 is OK. In the comments of the software, please discuss why you chose your particular implementation method over the other available choices. In particular, you are free to use iterative or recursive algorithms. You are free to modify the prototypes as well as handle illegal inputs in any way you feel is appropriate. Since you will be using these routines in conjunction with the DS1620 temperature chip, you could limit the range to -55 to +125. One the other hand, to make it more general, please make the range from -16384.000 to +16383.500.

Learn how to:

- examine and modify registers, I/O ports, and global variables;
- examine the stack;
- use breakpoints in the system.

An algorithm for `InFDec` is as follows. Let `data` be the signed 16-bit decimal fixed point output parameter

```
// 1) initialize sign=0, integer=0, fraction=0
// 2) input the first character, character=InChar()
//      if character is a minus sign, set the sign=-1 and character=InChar()
//      if character is a plus sign, character=InChar()
// 3) input the integer part, repeat until the character is '.' or <enter>
//      if character is between '0' and '9' then integer=10*integer+(character-0x30)
//      read next, i.e, character=InChar()
// 4) if character is not '.', skip to step 7
// 5) input the first character after the '.', character=InChar()
//      if character is between '5' and '9' then fraction=1
// 6) repeat until the character is <enter>
//      read next, i.e, character=InChar()
// 7) combine the integer and  fraction parts into the data
//     data=sign*(2*integer+fraction)
```

An algorithm for `OutFDec` is as follows. Let `data` be the signed 16-bit decimal fixed point input parameter

```
// 1) output sign and find the magnitude
  if(data<0)
    // output a minus sign and set the magnitude equal to -data
  else
     // output a plus sign and set the magnitude equal to +data
// 2) divide the magnitude into  integer and  fraction parts
  integer=magnitude/10;
  fraction=magnitude%10;
// 3) output the integer part without any leading or trailing spaces
// 4) output a decimal point
// 5) output the fraction part as a 1 digit decimal number
```

Part (3) Organize the **I/O routines into a device driver**.

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

- demonstrate your ability to set and clear breakpoints;
- show the TA you know how to observe global and local (stack) variables;
- 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.

Hints
1) Run the existing program 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 have found some compiler bugs in earlier versions of ICC12. I.e., sometimes the compiler will not create the proper executable code. ImageCraft has been very responsive in correcting bugs, so 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.)
4) You may find it useful to read the DS1620 data sheets to get a feel for the context of the fixed-point routines you are developing.