CECS 347
PROGRAMMING in C SYLABUS

Writing C Code for the 8051
by Matthew Kramer
(Available on-line at: http://ubermensch.org/Computing/8051/8051-c/#appa)

About the Keil Compiler

Keil Software (http://www.keil.com) publishes one of the most complete development tool suites for 8051 software, which is used throughout industry. For development of C code, their Developer's Kit product includes their C51 compiler, as well as an integrated 8051 simulator for debugging. A demonstration version of this product is available on their website, but it includes several limitations (see next section). This is the software that will be used for CECS-347.

The C programming language was designed for computers, though, and not embedded systems. It does not support direct access to registers, nor does it allow for the reading and setting of single bits, two very important requirements for 8051 software. In addition, most software developers are accustomed to writing programs that will by executed by an operating system, which provides system calls the program may use to access the hardware. However, much code for the 8051 is written for direct use on the processor, without an operating system. To support this, the Keil compiler has added several extensions to the C language to replace what might have normally been implemented in a system call, such as the connecting of interrupt handlers.

The purpose of this manual is to further explain the limitations of the Keil compiler, the modifications it has made to the C language, and how to account for these in developing software for the 8051 micro controller.

Keil Limitations

There are several very important limitations in the evaluation version of Keil's Developer's Kit that users need be aware of when writing software for the 8051.

Object code must be less than 2 Kbytes

The compiler will compile any-sized source code file, but the final object code may not exceed 2 Kbytes. If it does, the linker will refuse to create a final binary executable (or HEX file) from it. Along the same lines, the debugger will refuse any files that are over 2Kbytes, even if they were compiled using a different software package.

Few student projects will cross this 2Kbyte threshold, but programmers should be aware of it to understand why code may no longer compile when the project grows too large.

Program code starts at address 0x4000

All C code compiled and linked using the Keil tools will begin at address 0x4000 in code memory. Such code may not be programmed into devices with less than 16Kbytes of Read-Only Memory. Code written in assembly may circumvent this limitation by using the "origin" keyword to set the start to address 0x0000. No such work-around exists for C programs, though. However, the integrated debugger in the evaluation software may still be used for testing code. Once tested, the code may be compiled by the full version of the Keil software, or by another compiler that supports the C extensions used by Keil.

C Modifications

The Keil C compiler has made some modifications to an otherwise ANSI-compliant implementation of the C programming language. These modifications were made solely to facilitate the use of a higher-level language like C for writing programs on micro controllers.

Variable Types

The Keil C compiler supports most C variable types and adds several of its own.

Standard Types

The evaluation version of the Keil C compiler supports the standard ANSI C variable types, with the exception of the floating-point types. These types are summarized below.

Type	Bits	Bytes	Range
char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32,768 to +32,767
short	16	2	-32,768 to +32,767
unsigned short	16	2	0 to 65,535
int	16	2	-32,768 to +32,767
unsigned int	16	2	0 to 65,535
long	32	4	-2,147,483,648 to +2,147,483,647
unsigned long	32	4	0 to 4,294,697,295

In addition to these variable types, the compiler also supports the **struct** and **union** data structures, as well as type redefinition using **typedef**.

Keil Types

To support a micro controller and embedded systems applications, Keil added several new types to their compiler. These are summarized in the table below.

Type	Bits	Bytes	Range
bit	1	0	0 to 1
sbit	1	0	0 to 1
sfr	8	1	0 to 255

sf16	16	2	0 to 65,535

Of these, only the **bit** type works as a standard variable would. The other three have special behavior that a programmer must be aware of.

bit

This is a data type that gets allocated out of the 8051's bit-addressable on-chip RAM. Like other data types, it may be declared as either a variable. However, unlike standard C types, if may not be used as a pointer. An example of its usage follows.

```
/* declare two bit variables - the compiler will decide which */
/* addresses they are at.  Initialize them to 0 and 1. */
bit testbit1 = 0;
bit testbit2 = 1;

/* set testbit1 to the value in testbit2 */
testbit1 = testbit2;

/* clear testbit2 */
testbit2 = 0;

/* testbit1 is now a 1, and testbit2 is now a 0 */
/* Note that the assignment of testbit2 to testbit1 only copied */
/* the contents of testbit2 into testbit1.  It did *not* change */
/* the location of testbit1 to be the same as testbit2. */
```

sbit, sfr, and sf16

These are special types for accessing 1-bit, 8-bit, and 16-bit special function registers. Because there is no way to indirectly address registers in the 8051, addresses for these variables must be declared outsite of functions within the code. Only the data addressed by the variable may be manipulated in the code. An example follows:

```
/* create an sbit variable that points to pin 0 of port 1 */
/* note that this is done outside of any functions! */
sbit P10 = 0x90;

/* now the functions may be written to use this location */
```

```
void main (void)
{
    /* forever loop, toggling pin 0 of port 1 */
    while (1==1)
    {
        P10 = !P10;
        delay (500); /* wait 500 microseconds */
    }
}
```

Conveniently, the standard special function registers are all defined in the reg51.h file that any developer may include into their source file. Only registers unique to the particular 8051-derivative being used for the project need have these variable declared, such as registers and bits related to a second on-chip serial port.

Keil Variable Extensions

In writing applications for a typical computer, the operating system handles manages memory on behalf of the programs, eliminating their need to know about the memory structure of the hardware. Even more important, most computers having a unified memory space, with the code and data sharing the same RAM. This is not true with the 8051, which has separate memory spaces for code, on-chip data, and external data.

To accommodate for this when writing C code, Keil added extensions to variable declarations to specify which memory space the variable is allocated from, or points to. The most important of these for student programmers are summarized in the following table.

Extension	Memory Type	Related ASM
data	Directly-addressable data memory (data memory addresses 0x00-0x7F)	MOV A, 07Fh
idata	Indirectly-addressable data memory (data memory addresses 0x00-0xFF)	MOV R0, #080h MOV A, R0
xdata	External data memory	MOVX @DPTR
code	Program memory	MOVC @A+DPTR

These extensions may be used as part of the variable type in declaration or casting by placing the extension after the type, as in the example below. If the memory type extension is not specified, the compiler will decide which memory type to use automatically, based on the memory model (SMALL, COMPACT, or LARGE, as specified in the project properties in Keil).

```c
/* This is a function that will calculate and return a checksum of */
/* a range of addresses in code memory, using a simple algorithm */
/* that simply adds each consecutive byte together.  This could be */
/* useful for verifying if the code in ROM got corrupted (like if */
/* the Flash device were wearing out). */

unsigned int checksum (unsigned int start, unsigned int end)
{
    /* first, declare pointers to the start and end of */
    /* the range in code memory. */
    unsigned int code *codeptr, *codeend;

    /* now declare the variable the checksum will be */
    /* calculated in.  Because direct-addressable data */
    /* is faster to access than indirect, and this */
    /* variable will be accessed frequently, we will */
    /* declare it in data memory (instead of idata). */
    /* In reality, if left unspecified, the compiler */
    /* would probably leave it in the accumulator for */
    /* even faster access, but that would defeat the */
    /* point of this example. */
    unsigned int data checksum = 0;

    /* Initialize the codestart and codeend pointers to */
    /* the addresses passed into the function as params. */
    /* because start and end are passed in as values, */
    /* not pointers, they must be cast to the correct */
    /* pointer type */
    codeptr = (unsigned int code *)start;
    codeend = (unsigned int code *)end;

    /* Now perform the checksum calculation, looping */
    /* until the end of the range is reached. */
    while (codeptr <= codeend)
    {
        checksum = checksum + (unsigned int data)*codeptr;
        codeptr++; /* go to the next address */
    }
```

```
    return (checksum);
}
```

Keil Function Extensions

As in most other C compilers, functions may be declared in one of two fashions:

```
unsigned int functionname (unsigned int var)    functionname (var)
{                                                unsigned int var
   ....                                          {
   return (var);                                    ....
}                                                    return (var);
                                                 }
```

Most modern programmers use the first syntax, as do the examples in this document.

Keil provides two important extensions to the standard function declaration to allow for the creation of interrupt handlers and reentrant functions.

interrupt

In writing applications for a typical computer, the operating system provides system calls for setting a function, declared in the standard manner, as the handler for an interrupt. However, in writing code for an 8051 without an operating system, such a system would not be possible using solely C code. To eliminate this problem, the Keil compiler implements a function extension that explicitly declares a function as an interrupt handler. The extension is **interrupt**, and it must be followed by an integer specifying which interrupt the handler is for. For example:

```
/* This is a function that will be called whenever a serial */
/* interrupt occurs. Note that before this will work, interrupts */
/* must be enabled.  See the interrupt example in the appendix. */
void serial_int (void) interrupt 4
{
   ...
}
```

In the example, a function called **serial_int** is set as the handler for interrupt 4, which is the serial port interrupt. The number is calculated by subtracting 3 from the interrupt vector address and dividing by 8. The five standard interrupts for the 8051 are as follows:

Interrupt	Vector address	Interrupt number
External 0	0003h	0
Timer 0	000Bh	1
External 1	0013h	2
Timer 1	001Bh	3
Serial	0023h	4

Other interrupts are dependent on the implementation in the particular 8051-derivative being used in the project, but may be calculated in the same manor using the vector addresses specified by the manufacturer.

using

Since the processor only save the current program counter before executing an interrupt handler, the handler can potentially damage any data that was in the registers prior to the interrupt. This in turn would corrupt the program once the processor goes back to where it left off. To avoid this, the Keil compiler determines which registers will be used by the interrupt handler function, pushes them out to the stack, executes the handler, and then restores the registers from the stack, before returning to the interrupted code. However, this incurs extra time, especially if a lot of registers will be used. It is preferred that as little time be spent in interrupts as possible. To decrease this time, Keil provides an optional extension, **using**, to the **interrupt** extension that tells the compiler to simple change to a new register bank prior to executing the handler, instead of pushing the registers to the stack.

```
/* This is a function that will be called whenever a serial */
/* interrupt occurs. Prior to executing the handler, the */
```

```
/* processor will switch to register bank 1
void serial_int (void) interrupt 4 using 1
{
    ...
}
```

In the 8051, interrupts have two possible priorities: high and lo. If, during the processing of an interrupt, another interrupt of the same priority occurs, the processor will continue processing the first interrupt. The second interrupt will only be processed after the first has finished. However, if an interrupt of a higher priority arrives, the first (low priority) interrupt will itself be interrupted, and not resume until the higher priority interrupt has finished. Because of this, all interrupts of the same priority may use the same register bank.

The **using** extension should be used when quick execution time is of high importance, or when other functions are called from the interrupt handler, as it would otherwise push all of the registers on to the stack prior to calling the function, incurring more time penalties.

reentrant

Similar to the case described for interrupts above, it is possible for a single function to be interrupted by itself. For example, in the middle of normal execution of the function, the interrupt occurs, and that interrupt makes a call to the same function. While the interrupt handler will save the registers before entering this function, no protective measures are taken from overwriting the contents of local variables allocated in data memory. When the interrupt is serviced and control is passed back to normal execution, the corrupted data in those variables could ruin the entire program.

The general term for a function that may be called more than once simultaneously is "reentrant." Accordingly, the **reentrant** extension may be used in a function declaration to force the compiler to maintain a separate data area in memory for each instance of the function. While safe, this does have the

potential to use large area of the rather limited data memory. An example of such a function follows.

```c
/* Because this function may be called from both the main program */
/* and an interrupt handler, it is declared as reentrant to */
/* protect its local variables. */

int somefunction (int param) reentrant
{
   ...
   return (param);
}

/* The handler for External interrupt 0, which uses somefunction() */
void external0_int (void) interrupt 0
{
   ...
   somefunction(0);
}

/* the main program function, which also calls somefunction() */
void main (void)
{
   while (1==1)
   {
      ...
      somefunction();
   }
}
```

Appendix A - Sample Code

Sample code that is fully compilable in the evaluation version of Keil may be found at the following URL:

http://ubermensch.org/Computing/8051/code

Sample Code

Example code for the 8051 Microcontroller

basic.c - A very basic example of writing C code for the 8051.

int.c - A rewrite of the serial example to use interrupts in C.

serial.c - Example of how to read and write data on the 8051 serial port using polling.

Example 1 BASIC.C

```
/**************************************************************************
* basic.c - The basics of writing C code for the 8051 using the Keil
*  development environment.  In this case, a simple program will be
*  constructed to make a binary counter on Port 0.
*/

/*
 * As always with C, the included header files should come first.  Most
 * every project for the 8051 will want to include the file reg51.h.  This
 * header file contains the mapping of registers to names, such as setting
 * P0 (port 0) to address 0x80.  This allows the coder to use the keyword
 * "P0" in their code whenever they wish to access Port 0.  For the complete
 * list of registers named, view the file.
 */

#include
```

```c
/*
 * Other header files may be included after reg51.h, including any headers
 * created by the user.
 */

/*
 * The C program starts with function main().  In the case of a program
 * written using multiple .c files, main can only occur in one of them.
 * Unlike in programming user applications for a standard computer, the
 * main() function in a Keil program for the 8051 takes no inputs and
 * returns no output, thus the declaration has implied void types.
 */

/****************************************************************************
 * main - Program entry point
 *
 * INPUT: N/A
 * RETURNS: N/A
 */

main()
{
    unsigned int i;     /* will be used for a delay loop */

    /* First, Port 0 will be initialized to zero */

    P0 = 0;

    /*
     * Now the counter loop begins.  Because this program is intended
     * to run in an embedded system with no user interaction, and will
     * run forever, the rest of the program is placed in a non-exiting
     * while() loop.
     */

    while (1==1)
    {
```

```c
    /*
                  * This is a very unpredictable method of implementing a delay
                  * loop, but remains the simplest.  More reliable techniques
                  * can be done using the using the built-in timers.  In this
                  * example, though, the for() loop below will run through 60000
                  * iterations before continuing on to the next instruction.
                  * The amount of time required for this loop varies with the
                  * clock frequency and compiler used.
                  */

        for (i = 0; i < 60000; i++) {;}

            /* Increment Port 0 */

            P0 = P0 + 1;
    }

}
```

Example 2   INT.C

```c
/************************************************************************
* int.c - A demonstration of how to write interrupt-driven code for an
*  8051 using the Keil C compiler.  The same techniques may work in other
*  8051 C compilers with little or no modifcation.  This program will
*  combine the functionality of both basic.c and serial.c to allow serial
*  communications to be entirely in the background, driven by the serial
*  interrupt.  This allows the main() function to count on Port 0 without
*  being aware of any ongoing serial communication.
*/

/* included headers */

#include

/* function declarations */
```

```
char getCharacter (void);        /* read a character from the serial port */
void sendCharacter (char);       /* write a character to the serial port */


/*
 * Interrupt handlers:
 * Here the code for the interrupt handler will be placed.  In this
 * example, a handler for the serial interrupt will be written.
 * Examination of the 8051 specs will show that the serial interrupt is
 * interrupt 4.  A single interrupt is generated for both transmit and
 * receive interrupts, so determination of the exact cause (and proper
 * response) must be made within the handler itself.
 * To write an interrupt handler in Keil, the function must be declared
 * void, with no parameters.  In addition, the function specification
 * must be followed by a specification of the interrupt source it is
 * attached to.  The "using" attribute specifies which register bank
 * to use for the interrupt handler.
 */

void serial_int (void) interrupt 4
{
        static charchr = '\0';  /* character buffer */

        /*
         * The interrupt was generated, but it is still unknown why.  First,
         * check the RI flag to see if it was because a new character was
         * received.
         */

        if (RI == 1)            /* it was a receive interrupt */
        {
                chr = SBUF;         /* read the character into our local buffer */
                RI = 0;             /* clear the received interrupt flag */
                TI = 1;             /* signal that there's a new character to send */
        }
        else if (TI == 1)       /* otherwise, assume it was a transmit interrupt */
        {
```

```c
                TI = 0;      /* clear the transmit interrupt flag */
                if (chr != '\0')        /* if there's something in the local buffer... */
                {
                        if (chr == '\r') chr = '\n';        /* convert  to  */
                        SBUF = chr;                  /* put the character into the
transmit buffer */
                        chr = '\0';
                }
        }
}

/* functions */


/****************************************************************************
* main - Program entry point.  This program sets up the timers and
*  interrupts, then simply receives characters from the serial port and
*  sends them back.  Notice that nowhere in the main function is Port 0
*  incremented, nor does it call any other function that may do so.
*  main() is free to do solely serial communications.  Port 0 is handled
*  entirely by the interrupt handler (aside from initialization).
*
* INPUT: N/A
* RETURNS: N/A
*/

main()
{
        /* Before the serial port may be used, it must be configured. */

        /* Set up Timer 0 for the serial port */

        SCON = 0x50;        /* mode 1, 8-bit uart, enable receiver */
        TMOD = 0x20;        /* timer 1, mode 2, 8-bit reload */
        TH1  = 0xFE;        /* reload value for 2400 baud */
        ET0  = 0;           /* we don't want this timer to make interrupts */
        TR1  = 1;           /* start the timer */
        TI   = 1;           /* clear the buffer */
```

```c
/*
 * The compiler automatically installs the interrupt handler, so
 * all that needs to be done to use it is enable interrupts.  First,
 * speficially enable the serial interrupt, then enable interrupts.
 */

ES  = 1;            /* allow serial interrupts */
EA  = 1;            /* enable interrupts */

/* initialize Port 0 to 0, as in basic.c */

P0 = 0;

/*
 * Loop forever, increasing Port 0.  Again, note nothing is done
 * with the serial port in this loop.  Yet simulations will show
 * that the software is perfectly capable of maintaining serial
 * communications while this counting proceeds.
 */

while (1==1)
{
        unsigned int        i;
        for (i = 0; i < 60000; i++) {;}     /* delay */
        P0 = P0 + 1;            /* increment Port 0 */
}
}
```

Example 3 SERIAL.C

```c
/**************************************************************************
* serial.c - A demonstration of how to access the serial port on an
*  8051 using C code.  To avoid using interrupts, this example polls
*  the interrupt flags in the serial port to know when the serial port
*  is ready.
*/
```

```
/* included headers */

#include   /* register names */

/*
 * function declarations - Here the functions in our code are declared.
 *  In C, this is only necessary if the actual implementation of the
 *  function is performed in a separate file or after any function that
 *  calls it in its own file.  In this program, these functions will
 *  all be implemented within this file.  However, for asthetics,
 *  functions will be implemented in order of highest-level to lowest.
 *  By nature, this creates a scenario where the functions will be
 *  called by code placed above the actual implementation, so the
 *  functions must first be declared here.
 */

char getCharacter (void);        /* read a character from the serial port */
void sendCharacter (char);       /* write a character to the serial port */

/* functions */

/*************************************************************************
* main - Program entry point.  This program will simply receive characters
*  from the serial port, then send them back.
*
* INPUT: N/A
* RETURNS: N/A
*/

main()
{
        char      chr;        /* variable to hold characters in */

        /* Before the serial port may be used, it must be configured. */

        /*
```

```
 * The serial controll register configures the method of operation
 * for the serial port.  The value used below is 0x50 (or 50h in
 * 8051 lingo), referring to the bits within the SCON register,
 * this does the following:
 *  MODE = 010 (8-bit UART serial buffer, no stop bit - this is typical)
 *  REN = 1 (enabled receiving)
 *  TB8, RB8 = 00 (unused in this mode)
 *  RI,TI = 00 (start the interupt flags as ready to receive and send)
 */

SCON = 0x50;                    /* mode 1, 8-bit uart, enable receiver */

/*
 * Because a standard serial port transmits no clocking signal, both
 * ends of the serial connection must agree on a clock frequency,
 * which is then generated internally at each end.  For this example,
 * a baud rate of 2400 bits per second will be used.  The timer must be
 * configured accordingly.
 * The formula for determining the reload value based on desired baud
 * rate and clock frequency is:
 *  TH1 = 256 - clock frequency (in Hz) / (384 * baud rate)
 * For 2400bps and a 11.02Mhz clock:
 *  TH1 = 256 - 11,020,000 / (384 * 2400) = 255 = 0xFE
 */

TMOD = 0x20;                    /* timer 1, mode 2, 8-bit reload */
TH1  = 0xFE;                    /* reload value for 2400 baud */

/* Setting TR1 will start the timer, and serial communications */

TR1  = 1;

/*
 * Set the Transmit Interrupt flag to send the the character in
 * the serial buffer, clearing it for use by the program.
 */
```

```c
        TI  = 1;

        /*
         * Now the program isr eady to send and receive data on the serial
         * port. Because it is going to do this indefinitely (until the
         * device is effectively turned off), the rest of the program will
         * be in an infinite while() loop.
         */

        while (1==1)
        {

                /* read the next character from the serial port */

                chr = getCharacter ();

                /* send it back to the original sender */

                sendCharacter (chr);
        }
}


/**************************************************************************
 * getCharacter - Waits for a new character to arrive in the serial port,
 *  then reads it.
 *
 * INPUT: N/A
 * RETURNS: newly received character
 */

char getCharacter (void)
{
        char chr;  /* variable to hold the new character */

        /*
         * Wait until the serial port signals a new character has arrived.
         * It does so by setting the Received interrupt flag, RI, which
```

```
          * this routine loops on until it is set to 1.  This is known
          * as polling.
    */

          while (RI != 1) {;}

          /* now read the value in the serial buffer into the local variable */

          chr = SBUF;

          /*
           * Once the character is read, the serial port must be told that it is
           * free to receive a new character.  This is done by clearing the
           * Received Interrupt flag.
           */

          RI = 0;

          /* the character is then returned to the calling function. */

          return(chr);
}

/**************************************************************************
* sendCharacter - Waits until the serial port is ready to send a new
*  character, then sends it.
*
* INPUT:
*          chr - The character to send
*
* RETURNS: N/A
*/

void sendCharacter
(
   char    chr        /* character to send */
)
```

```c
{
    /*
     * Because of the way terminal programs for serial ports work, we want
     * to replace carriage returns with line feeds.
     */

    if (chr == '\r') chr = '\n';

    /*
     * Wait until the serial port signals the previous character has
     * been sent. It does so by setting the Transmit interrupt flag, TI,
     * which this routine loops on until it is set to 1.
     */

    while (TI != 1) {;}

    /*
     * Clear the Transmit Interrupt flag to prepare the serial port
     * to send a new character.
     */

    TI = 0;

    /* Write the character into the serial port buffer register, SBUF */

    SBUF = chr;

    /*
     * The serial port hardware takes over from here, and the program
     * may continue with other operations.
     */

    return;
}
```

Example 4 : Implementing a 4 bit Counter using an 8051 and Interfacing it to an LCD

Prof. Frank Vahid

Purpose:

In this lab, you will learn how to write a simple C program for 80C51 micro-controller, compile it using C51 compiler, and emulate it on an emulator using Pro51. The program will be used to control a simple 4-bit up-down counter, capable of counting from 0 to 15. At each step the count should be displayed in decimal format on the LCD.

Assignment:

In this lab :

- You will design a 4-bit Up-Down counter using the C programming language for the 8051 micro-controller and display the count on an LCD.

- You will then test and run your program on the 8051.

- The 4 bit counter has the following functionality:
  - The counter has the following input pins :
    a. *Reset :* when high resets the counter *dataout* to ``0000''
    b. *Updown :* decides whether the counter counts up or down.
    c. *Load :* makes the counter count from the 4 bit input *Datain*
    d. *Datain :* which is a 4 bit input count
  - The counter has a 4 bit *Dataout* to reflect the count.
  - The count has to be sent to the LCD and displayed in decimal format.

Apparatus Required:

4. 4.7k resistors(8)
5. 1k resistor(1)
6. DIP switch
7. LCD
8. 5V power supply
9. Philips PDS51 development board \

Schematic:
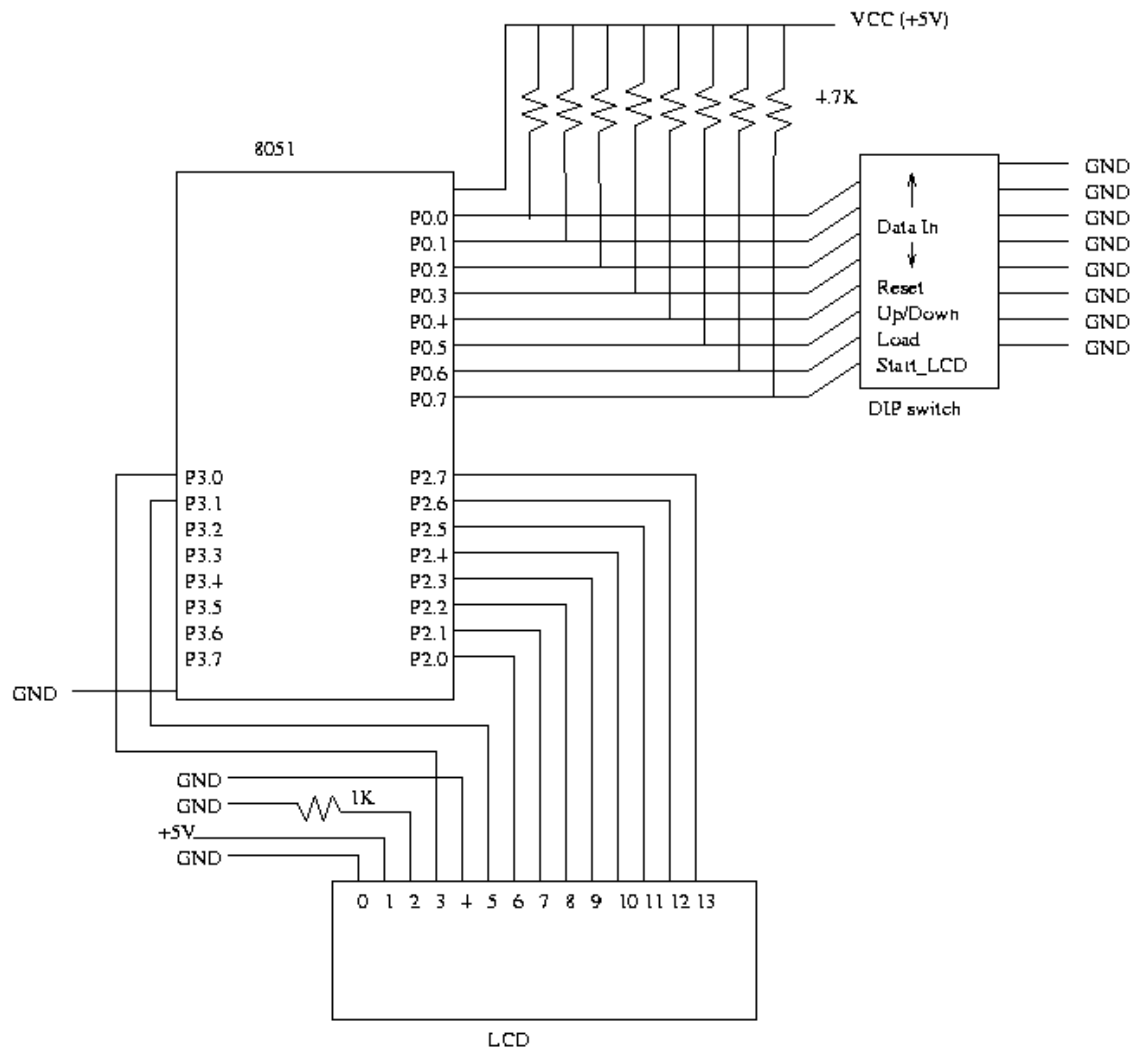
Program:

```
#pragma SMALL DB OE
#include
#include "io.h"

/* P0, P1, P2 and P3 are predefined port names and are bit addressable */
```

```c
sbit reset = P0^4; /* bit 4 of Port 0 */
sbit up_down = P0^5;
sbit load = P0^6;
sbit Start_LCD = P0^7; /* bit 7 of Port 3 */

/* Delay function */
void delay() {

    int i, j;

    for(i=0; i<1000; i++)
        for(j=0; j<100; j++)
            i = i + 0;
}

/* Function to output the decimal value of the count on the LCD */
void PrintInt(unsigned char i) {
   char ch[4];

   /* Write code to convert the count to a string value and use the
        PrintString function provided in io.c */

    PrintString(ch);
}

void main(void) {

    unsigned char count = 0;

    InitIO();  /* Initialize the LCD */
    while (1) {
        if (Start_LCD == 1) {
            ClearScreen();
            PrintString("Ready...");
            delay();
        }
```

```
else if (reset == 1) {

        /* Output 0 on the LCD */

    }
    else if (load == 1) {

        /* Output the current value of Datain on the LCD */

    }
    else {
        /* Check the Up/Down pin for 1 or 0 count up or down
           accordingly. Display each value on the LCD */
    }
  }
}
```

Steps to be followed:

10. Wire up the circuit as shown in the schematic.
    Note: Port 0 should not be used for output because it does cannot sufficiently drive the LCD.
11. Map your network drive to

        P:\\Peart\cs122

12. Run the batch file cs122.bat
13. Get the IO files to control the LCD. The functions specified in these files are used to handle initialization and other special functions of the LCD.
    o io.c
    o io.h
14. Open up a DOS window and edit your program under *C:* For eg:
15.   C:\Temp\count.c

16. Compile your programs
17.   c51 count.c
18.   c51 io.c

This would generate object files: count.obj, io.obj

19. Link the object files to create your executable file.
20.   bl51 count.obj, io.obj to count.omf

Example 5 : Implementing a Calculator Using Peripherals Like a Keypad and LCD

Prof. Frank Vahid

Purpose:

In this lab, you will build a simple calculator using the keypad as an input and the LCD as an output peripheral. After debugging and testing your program, you will have to burn the compiled program to a standalone 8051 chip at the end of this lab.

Description:

Keypads are often used as a primary input device for embedded micro controllers. The keypads actually consist of a number of switches, connected in a row/column arrangement as shown in Fig 1.

In order for the micro controller to scan the keypad, it outputs a nibble to force one (only one) of the columns low and then reads the rows to see if any buttons in that column have been pressed. The rows are pulled up by the internal weak pull-ups in the 8051 ports. Consequently, as long as no buttons are pressed, the micro controller sees a logic high on each of the pins attached to the keypad rows. The nibble driven onto the columns always contains only a single 0. The only way the micro controller can find a 0 on any row pin is for the keypad button to be pressed that connects the column set to 0 to a row. The controller knows which column is at a 0-level and which row reads 0, allowing it to determine which key is pressed. For the keypad, the pins from left to right are: R1, R2, R3, R4, C1, C2, C3, C4.
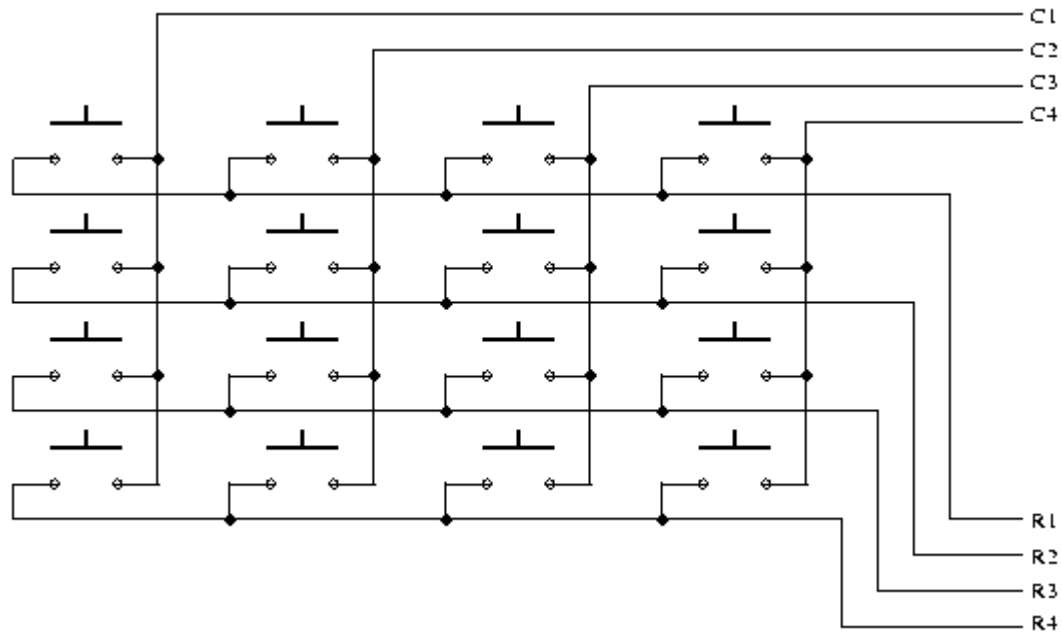
**Fig 1. Keypad connection**

Assignment:

In this lab :

- You will design a integer calculator with a keypad input. Your design should be able to read the operands (integers 0 to 9) and the operator from the keypad. You can design any combination of the input sequence as you like. For example, you may wish to input in the way:
  1. pick the first operand
  2. pick the operator
  3. pick the second operand

Apparatus Required:

1. 1k resistor(1)
2. keypad
3. LCD
4. 12MHz Crystal
5. 5V power supply
6. Philips PDS51 development board
7. Programmer LCPX5X40

Schematic:
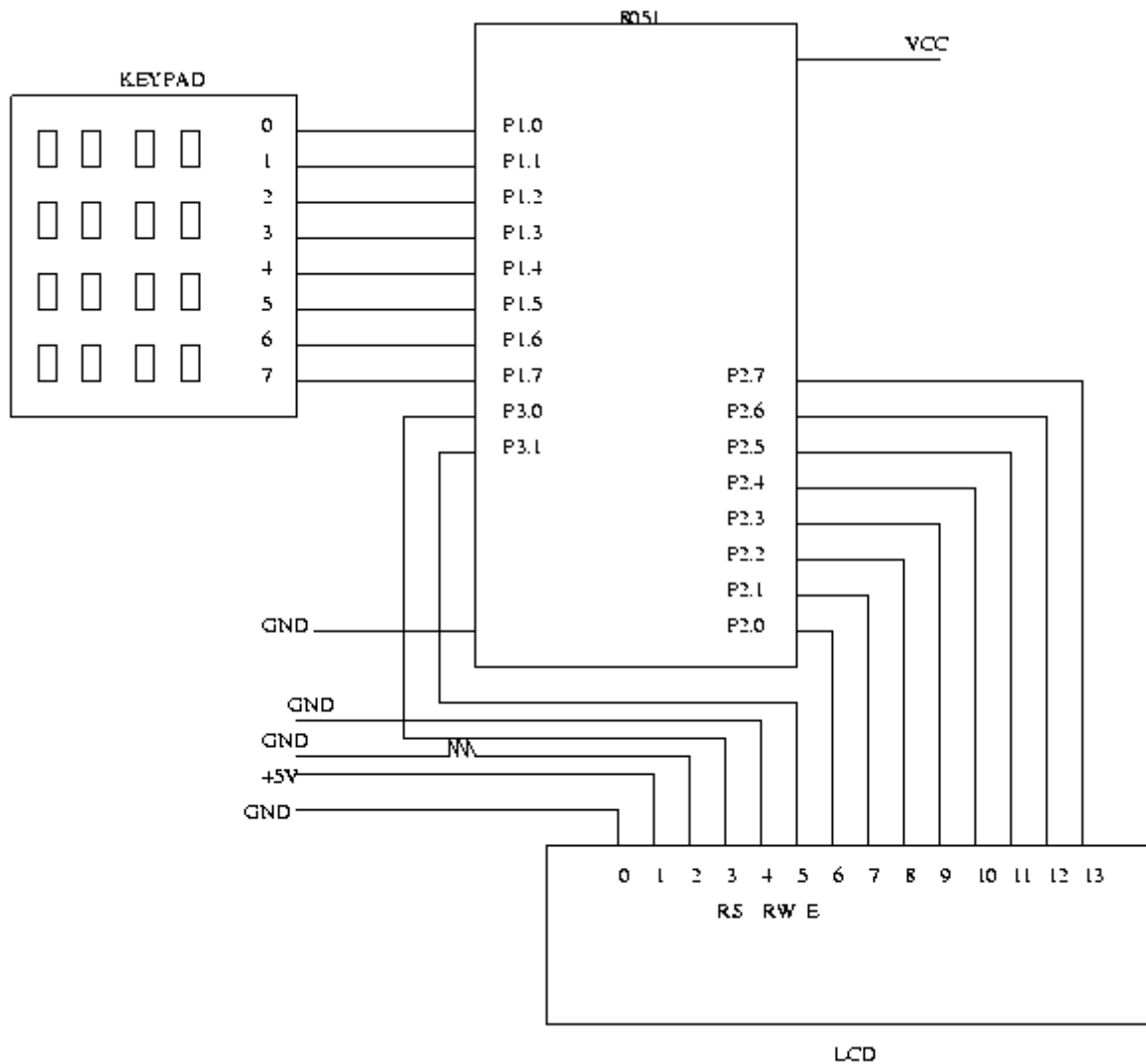
Program:

/* To implement a integer calculator using a keypad and LCD */

#pragma SMALL DB OE
#include <reg51.h>
#include "io.h"

/* The functions to initialize and control the LCD are assumed to be in the file io.c */

/* Function to output the decimal value of the result on the LCD */

```c
void PrintInt(int i) {

    .

    .

    .

}


/* Routine to scan the key pressed */
unsigned char key_scan()
{
    unsigned char i, j, temp1, temp2;

    while( 1 ) /* keep waiting for a key to be pressed */

        for(i=0; i<4; i++) {

            /* Set each row to 0 */
            P1 = 0xff & ~(1<<i);

            /* Scan each column to see which key was pressed */
            for (j=4; j<8; j++) {

                /* Code to determine the position of the
                   key which was pressed */
                /* return(position) */
            }
        }
}

void main() {

    /* You can have a conversion table to convert the key position into a
       valid number which indicates the operator or operand value. For eg:
       the numbers 0 to 9 are valid operands and 100 to 103 denote
       addition, subtraction, multiplication and division respectively */

    char conv_table[] = {
```

```c
        1, 2,  3, 100 /* add */,
        4, 5,  6, 101 /* sub */,
        7, 8,  9, 102 /* mul */,
       -1, 0, -1, 103 /* div */
};
char num1, num2, op;
int result;

InitIO();

while(1) {

    ClearScreen();

    /* read num1 */
    GotoXY(0, 0);
    PrintString("num1 : ");
    do {

        num1 = conv_table[key_scan()];
    }
    while( num1 < 0 || num1 > 9 );

    /* read a valid operation */
    GotoXY(0, 0);
    PrintString("op :   ");
    do {

        op = conv_table[key_scan()];
    }
    while( op < 100 );

    /* read num2 */
    GotoXY(0, 0);
    PrintString("num2 : ");
    do {
```

```
            num2 = conv_table[key_scan()];
        }
        while( num2 < 0 || num2 > 9 );


        /* compute result */
        if( op == 100 ) {

            /* Add numbers and display result on the LCD */
        }
        else if( op == 101 ) {
            .
            .
            .
            .
        /* Continue similarly for other operations */

        }
    }
}
```

Example 6 : Serial Communication

Prof. Frank Vahid

Purpose:

To establish a serial communication link between the PC and the 8051.

Description:

Serial communication is often used either to control or to receive data from an embedded microprocessor. Serial communication is a form of I/O in which the bits of a byte begin transferred appear one after the other in a timed sequence on a single wire. Serial communication has become the standard for intercomputer communication. In this lab, we'll try to build a serial link between 8051 and PC using RS232.

RS232C

The example serial waveforms in Fig 1 show the waveform on a single conductor to transmit a byte (0x41) serially. The upper waveform is the TTL-level waveform seen at the transmit pin of 8051. The lower waveform shows the same waveform converted to RS232C levels. The voltage level of the RS232C are used to assure error-free transmission over greater distances than would be possible with TTL levels.

As shown in Fig 1, each byte is preceded by a start bit and followed by one stop bit. The start and stop bits are used to synchronize the serial recivers. The data byte is always transmitted *least-significant-bit first*. For error checking it is possible to include a parity bit as well, just prior to the stop bit. The bits are transmitted at specific time intervals determined by the **baud rate** of the serial signal. The baud rate is the reciprocal of the time to send 1 bit. Error-free serial communication requires that the baud rate, number of data bits, number of stop bits, and presence or absence of a parity bit be the same at the transmitter and at the receiver.
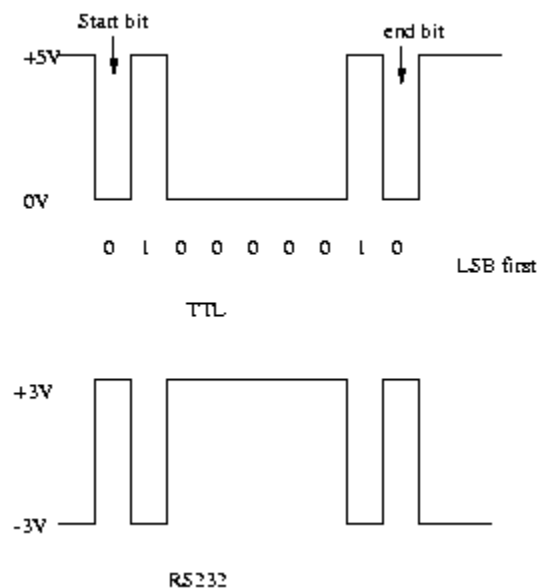
Fig 1. Serial Waveforms

Using the Serial Port

8051 provides a transmit channel and a receive channel of serial communication. The transmit data pin (TXD) is specified at P3.1, and the receive data pin (RXD) is at P3.0. The serial signals provided on these pins are TTL signal levels and must be *boosted* and *inverted* through a suitable converter(LT1130CN is used in this lab) to comply with RS232 standard.

All modes are controlled through SCON, the Serial CONtrol register. The SCON bits are defined as SM0, SM1, SM2, REN, TB8, RB8, TI, RI from MSB to LSB. The timers are controlled using TMOD, the Timer MODe register, and TCON, the Timer CONtrol register.

Register Descriptions

SCON bit definitions

SCON Serial Control Register

```
(msb)                  (lsb)
 ----------------------------
|SMO|SM1|SM2|REN|TB8|RB8|TI|RI|
 ----------------------------
```

SMO, SM1, SM2 Serial Mode COntrol Bits

```
SM0 SM1  Mode Baud Rate
-----------------------
 0  0   0   fosc/12
 0  1   1   variable
 1  0   2   fosc/32 or fosc/64
```

SM2 Multiprocessor Mode Control Bit

1 = Multi-processor mode

0 = Normal mode

REN Receiver Enable Bit

1 = Receive Enable
0 = Receive Disabled

TB8 9th Transmit Bit
   Enabled only in modes 2 and 3

RB8 9th Bit Received
   Used in modes 2 and 3

RI, TI Serial Interrrupts

   RI is set to indicate receipt of a serial word and TI
   is set to indicate completion of a serial transmission.

TMOD Timer Mode Register

   |Gate|C/T|M1|M0|Gate|C/T\|M1|M0|

   |<-Timer 1----><---Timer 0 --->

Gate Gating Control.
   0= Timer enabled
   1 = Timer enabled if INTx\ is high

C/T\ Counter or Timer Selector
   0 = Internal count source (clock/12)
   1 = External count source (Tx pin)

M1, M0 Mode Control

```
M1 M0       Mode
---------------------------------
 0  0   Mode 0, 13 bit count mode
 0  1   Mode 1, 16 bit count mode
 1  0   Mode 2, Auto reload mode
 1  1   Mode 3, Multiple mode
```

TCON Timer Control Receiver Register

```
--------------------------
|TF1|TR1|TF0|TR0| | | | |
--------------------------
<-Timer Controls><-Unused for timers
```

TRx Timer x run control
   0 = Timer not running
   1 = Timer running

TFx Timer x flag
   0 = timer has not rolled over
   1 = timer has rolled over

Formula to load the value of TH1 corresponding to required baud rate

$$\frac{\text{Clock Frequency (12 MHz)}}{12 \times 32 \times (256 - TH1)} = \text{Baud Rate}$$

RS232 connector

PCs have 9pin/25pin male SUB-D connectors. The pin layout is as follows (seen from outside your PC):

```
   1                13     1     5

  _____      _____
  \ . . . . . . . . . . . . /   \ . . . . . /
   \ . . . . . . . . . . . /     \ . . . . /
    -------------------------     -----------
   14                 25     6     9
```
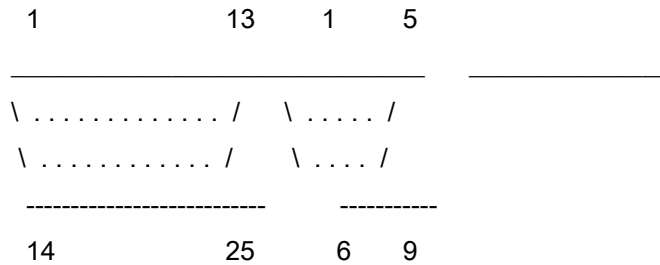
```
 Name (V24)  25pin  9pin  Dir  Full name          Remarks
-------------------------------------------------------------------------
   TxD      2    3   o   Transmit Data
   RxD      3    2   i   Receive Data
   RTS      4    7   o   Request To Send
   CTS      5    8   i   Clear To Send
   DTR     20    4   o   Data Terminal Ready
   DSR      6    6   i   Data Set Ready
   RI      22    9   i   Ring Indicator
   DCD      8    1   i   Data Carrier Detect
   GND      7    5   -   Signal ground
   -        1    -   -   Protective ground     Don't use this one
                                               for signal ground!
```

- The most important lines are RxD, TxD, and GND. Others are used with modems, printers and plotters to indicate internal states.

Apparatus Required:

1. 0.1 µF capacitors(6)
2. LT1130CN
3. connector and cable
4. 5V power supply
5. Philips PDS51 development board

Schematic:
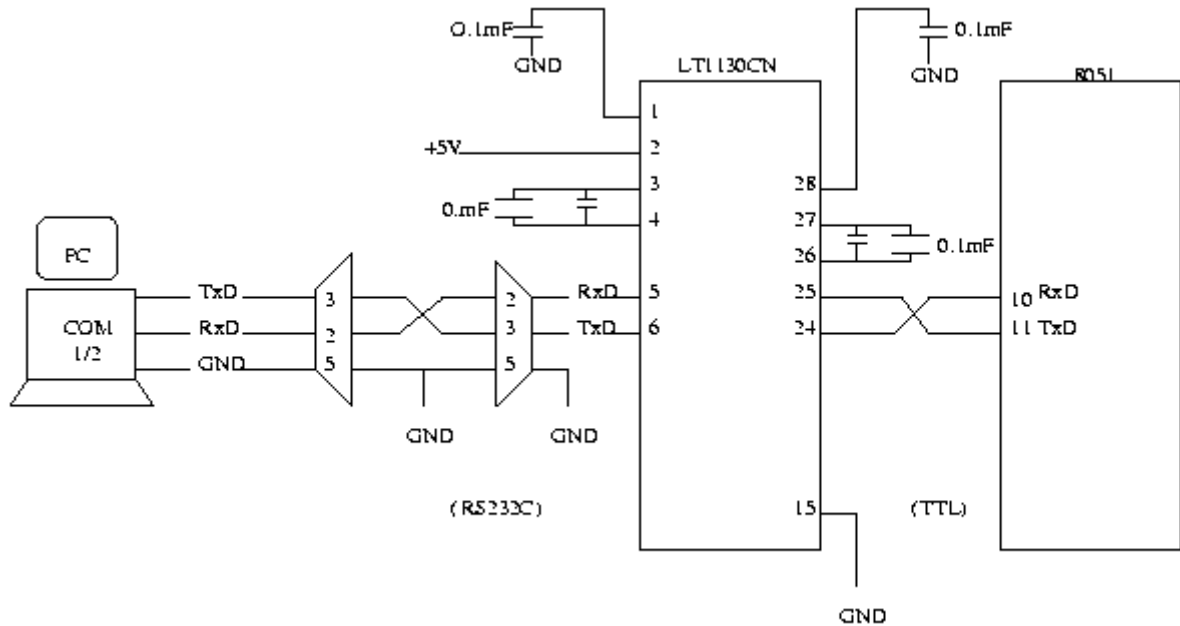
Program:

```
#pragma SMALL DB OE
#include <reg51.h>

unsigned char ReceiveSerial() {

        unsigned char c;

        TMOD = 0x20;        /* configure timer for the correct baud rate */
    TH1 = 0xe6;     /* 1200 bps for 12 MHz clock */
    TCON = 0x00;    /* Set timer to not running */

        SCON = 0x50;    /* Set Serial IO to receive and normal mode */
        TR1 = 1;   /* start timer to Receive */
        while( (SCON & 0x01) == 0 ) /* wait for receive data */;
        c = SBUF;
        return c;
}

void SendSerial(unsigned char c) {
```

```
        /* initialize..set values for TMOD, TH1 and TCON */
        /* set the Tx interrupt in SCON to indicate sending data */
        /* start timer */
        /* write character to SBUF */
        /* wait for completion of sent data */
}


void main(void) {

        unsigned char c;

        while( 1 ) {

        /* Use ReceiveSerial to read in a character 'c' */
        /* Do some computation on 'c' */
        /* Send the result using SendSerial() */
        }
}
```

Example 7 : Analog to Digital Conversion
Prof. Frank Vahid

Purpose:

To be able to implement analog to digital conversion using the ADC0804LCN 8-bit A/D converter. You will design a circuit and program the chip so that when an analog signal is given as input, the equivalent digital voltage is displayed on an LCD display. Thus, in effect, your circuit should function like a simple voltmeter.

Description:

The ability to convert analog signals to digital and vice-versa is very important in signal processing. The objective of an A/D converter is to determine the output digital word corresponding to an analog input signal.

The [Datasheet](#) for ADC0804LCN shows the pinout and a typical application schematic. The A/D converter operates on the successive approximation principle. Analog switches are closed sequentially by successive-approximation logic until the analog differential input volatge[$V_{in(+)}$ - $V_{in(-)}$] matches a voltage derived from a tapped resistor string across the reference voltage.

The normal operation proceeds as follows. On the high-to-low transition of the WR input, the internal SAR latches and the shift-register stages are reset, and the INTR output will be set high. As long as the CS input and WR input remain low, the A/D will remain in a reset state. Conversion will start from 1 to 8 clock periods after at least one of these inputs makes a low-to-high transition. After the requisite number of clock pulses to complete the conversion, the INTR pin will make a high-to-low transition. This can be used to interrupt a processor, or otherwise signal the availability of a new conversion. A RD operation(with CS low) will clear the INTR line high again. The device may be operated in the free-running mode by connecting INTR to the WR input with CS=0.

Since this is an 8-bit A/D converter, a voltage from 0-5V. O will be repersented as 0000 0000 (0 in decimal) and 5V is represented as 1111 1111 (256 in decimal). To convert a value X volts to decimal, use the following formula:

```
  X * 5.0
  -------
   256
```

To get a better resolution, and display the vlaue as a floating point number, you can multiply the numerator by a factor of 100, 1000 etc. and then print the voltage accordingly.

Apparatus Required:

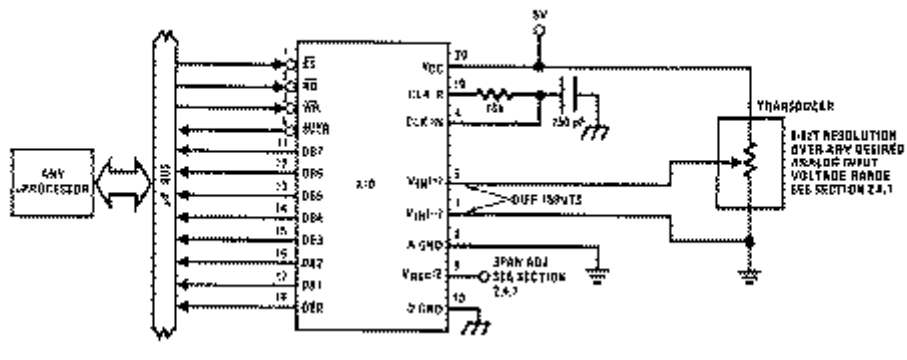1. ADC0804LCN
2. 10k resistor
3. 1k resistor
4. 5V power supply

Refer to the pinout and the application example in the Datasheet to connect the ADC0804. The LCD can be connected as was done in the earlier labs.
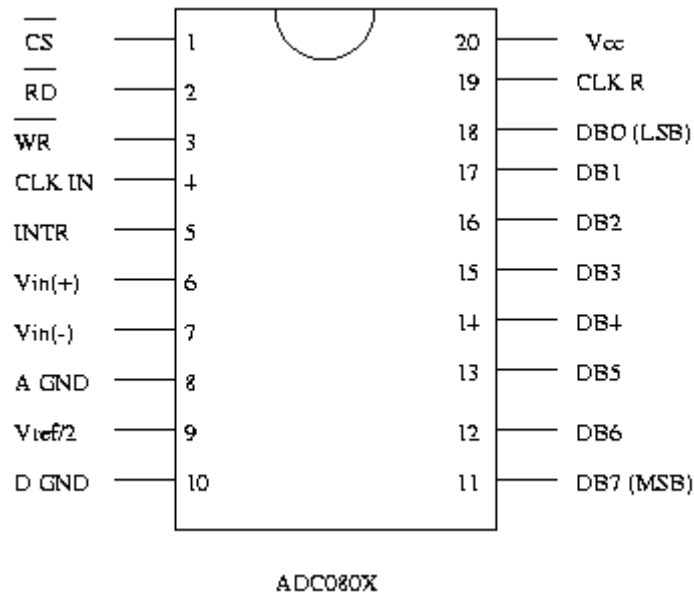
Figure 1. Connection Diagram

Figure 2. Typical Applications
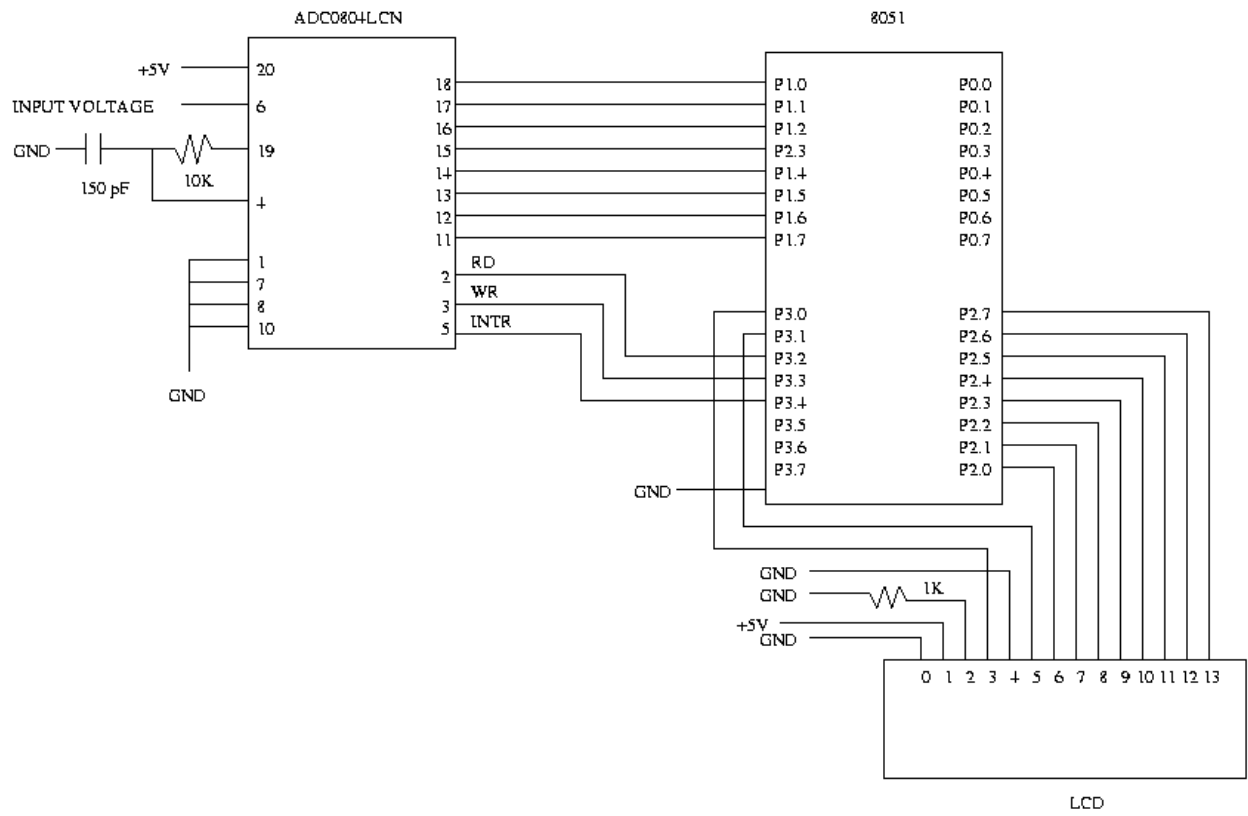
Figure 3. A/D Schematic

Program:

```
#include <reg51.h>
#include "io.h"



sbit READ = P3^2; /* Define these according to how you have connected the */
sbit WRITE = P3^3;/* RD, WR, and INTR pins                           */
sbit INTR = P3^4;



void main( void ) {

        unsigned char adVal;
```

```
        unsigned long volts;
        InitIO();


        READ = 1;
        WRITE = 1;
        INTR = 1;
        ClearScreen();

    while(1) {

        /* Make a low-to-high transition on the WR input */

        while( INTR == 1 );    /* wait until the INTR signal makes  */
                    /* high-to-low transition indicating */
                    /* completion of conversion          */



        /* Read the voltage value from the port */
        READ = 0;
        adVal = P1;
        READ = 1;

        /* Compute the digital value of the volatge read */

        /* Print the value of the voltage in decimal form */
        }
}
```

Example 8 : Controlling a Stepper Motor

Prof. Frank Vahid

Purpose:

The purpose of this lab is to control a stepper motor, with instructions received from the PC via a serial communication link to the 8051.

Description:

Stepper motors are often used in conjunction with microcontrollers to provide precisely controlled motor power. Stepper motors are devices that rotate a precise number of degrees for each "step" applied, as opposed to regular motors, which simply rotate continuously when power is applied. Driving a stepper motor involves applying a series of voltages to the four(typically) coils of the stepper motor. The coils are energized one or two at a time to cause the motor to rotate one step.

Assignment:

In this lab, you'll design a program to do the following jobs:

1. Set up the stepper motor and stepper motor driver circuit
2. Interface the PC to the 8051 through a serial communication link. Control the movement of the stepper motor by characters received from the PC.
   - When the character 'l' is received from the PC, make the stepper motor turn left. Also display the message "Moving left" on an LCD interfaced to the 8051.
   - When the character 'r' is received from the PC, make the stepper motor turn in the opposite direction and the mesasge "Moving Right" should be displayed on the LCD.

Apparatus Required:

1. MJE3055T NPN transistors (4)
2. MJE2955T PNP transistors (4)
3. MC3479P Stepper motor driver
4. LB82773-M1 Bipolar Stepper Motor
5. 1k resistors (9)
6. 47K resistor
7. 0.1μF capacitors (6)
8. Serial communication cable, connectors
9. LT1130CN
10. LCD
11. 5V power supply
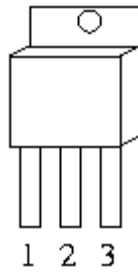12. Philips PDS51 development board

Schematic:

Figure 1. Pinout for transistors

Figure 2. Stepper Motor Schematic

Program:

#pragma SMALL DB OE

#include <reg51.h>

```c
unsigned char ReceiveSerial() {

        unsigned char c;

        TMOD = 0x20;        /* configure timer for the correct baud rate */
    TH1 = 0xe6;    /* 1200 bps for 12 MHz clock */
    TCON = 0x00;    /* Set timer to not running */

        SCON = 0x50;    /* Set Serial IO to receive and normal mode */
        TR1 = 1;  /* start timer to Receive */
        while( (SCON & 0x01) == 0 ) /* wait for receive data */;
        c = SBUF;
        return c;
}

void SendSerial(unsigned char c) {

    /* initialize..set values for TMOD, TH1 and TCON */
    /* set the Tx interrupt in SCON to indicate sending data */
    /* start timer */
    /* write character to SBUF */
    /* wait for completion of sent data */
}

void main(void) {

        unsigned char c;

        while( 1 ) {

      /* Use ReceiveSerial to read in a character */
      /* Depending on character make the motor move left or right */
      /* and display the direction on the LCD */
    }
}
```

1. Set up the circuit for the stepper motor as shown in the above schematic. Note that the circuit requires two different voltages but you need a common ground. *CAUTION: Transistors can get very hot.*
2. Interface the LCD and 8051 as was done in earlier labs. Pins 3 and 5 of the LCD are connected to different pins on the 8051. You will need to make the appropriate changes to the io.c file.
3. Set up serial communication between the PC and 8051. *(See Lab 3 and make the appropriate changes.)*
4. The stepper motor moves when Clk (pin 7 on the MC3479P chip) is toggled. It's direction depends on whether CW/CCW (pin 10 on the MC3479P chip) is set to 0 or 1.
5. Run your program on the emulator.
   - To transmit data go to Neighborhood Network -> Peart -> cs122 -> Serial
   - Copy the executable onto your desktop
   - Enter the letter you wish to transmit and click "Transmit" several times. If everything is done correctly, your stepper motor should be moving.

Program IO.C

```
#pragma SMALL DB OE




/*---------------------------------------------------------------------*/




#include <reg51.h>


#include "io.h"




/*---------------------------------------------------------------------*/
```

```c
sfr DATA_BUS = 0xa0;

sbit RS = 0xb0;

sbit E  = 0xb1;


/*------------------------------------------------------------------------*/


static void EnableLCD(int t) {


    unsigned char i;


    E = 1; for(i=0; i<t; i++) i = i;

    E = 0; for(i=0; i<t; i++) i = i;

}


/*------------------------------------------------------------------------*/


void InitIO(void) {
```

```c
    RS=0;

    DATA_BUS=0x38; EnableLCD(255);

    DATA_BUS=0x38; EnableLCD(255);

    DATA_BUS=0x38; EnableLCD(255);

    DATA_BUS=0x01; EnableLCD(255);

    DATA_BUS=0x0d; EnableLCD(255);

    DATA_BUS=0x06; EnableLCD(255);

    RS = 1;

}


/*-------------------------------------------------------------------*/


void ClearScreen(void) {


    RS=0;

    DATA_BUS=0x01; EnableLCD(255);

    RS = 1;

}
```

```c
/*-----------------------------------------------------------------------*/

void GotoXY(unsigned char r, unsigned char c) {

    RS=0;

    DATA_BUS=0x02; EnableLCD(255);

    for(r=r*40+c, c=0; c<r; c++)

        DATA_BUS = 0x14, EnableLCD(45);

    RS=1;

}

/*-----------------------------------------------------------------------*/

void PutChar(char c) {

    DATA_BUS = c; EnableLCD(45);

}
```

```
/*-------------------------------------------------------------------*/
```

void PrintString(const char* s) {

   while(*s) DATA_BUS = *(s++), EnableLCD(45);

}

Program h.c

```
#ifndef __io_h__
#define __io_h__

#define HIDE_CURSOR     0,25

void InitIO(void);
void ClearScreen(void);
void GotoXY(unsigned char r, unsigned char c);
void PutChar(char c);
void PrintString(const char* s);

#endif
```