EDSIM51™ DI Examples

This page contains example programs (mostly in assembly, but a few in C) for excercising the EdSim51 peripherals.

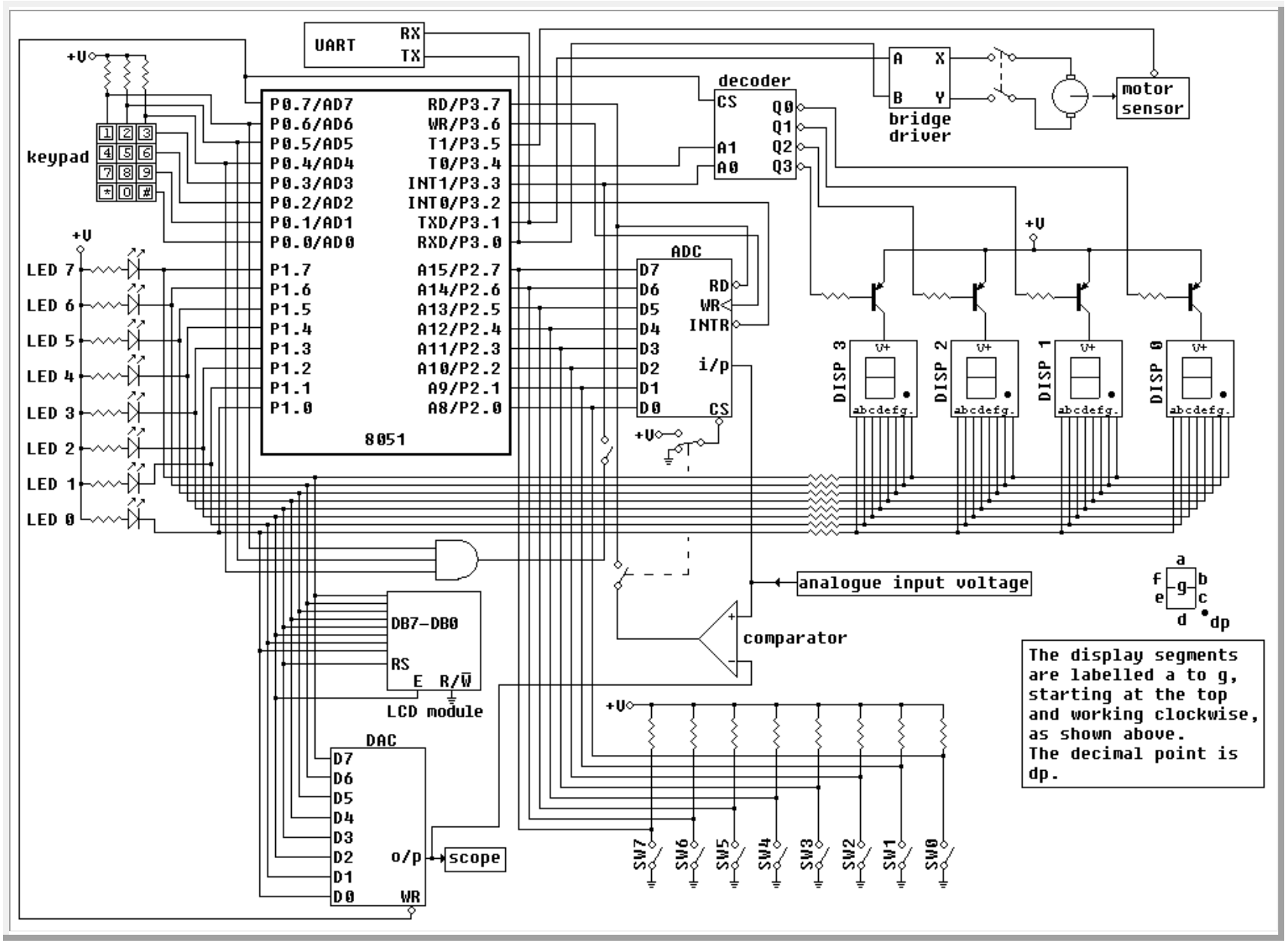
First, download for free the **EdSim51DI simulator** - it's free and is very easy to install.

The simulator does not run in real time, of course. When a program is running, the amount of elapsed time (as far as the 8051 is *aware*) is displayed in the field above the source code. The user can set the number of instructions executed between updates to the simulator GUI by selecting a value from the **Update Freq.** menu. Certain update frequencies suit some programs better than others.

For example, if a program is multiplexing the four 7-segment displays, then it is best to run this program with an update frequency of 1. But if a program is sending data to the serial port, then a higher update frequency might be used in order to see the data arrive promptly.

All of these programs have been written for 12 MHz 8051 system clock. For other clock frequencies, time delays would need to be modified accordingly.

The peripheral logic diagram is shown below. This diagram (and the extracts further down) relates to the standard peripheral interface. The user can move the peripherals to other port pins. A new logic diagram is then available from the simulator itself.



---

1. Binary Pattern on the Port 1 LEDs

```
; This program displays the binary pattern
; from 0 to 255 (and back to 0) on the LEDs
; interfaced with port 1.

; A 1 in the pattern is represented by the LED on,
; while a 0 in the pattern is represented by the LED off.

; However, logic 0 on a port 1 pin turns on the LED,
; therefore it is necessary to write the inverse of the
; pattern to the LEDs. The easiest way to do this is
; to send the data FFH to 0 (and back to FFH) to the LEDs.

; Since port 1 is initially at FFH all we need to do is
; continuously decrement port 1.

start:
        DEC P1          ; decrement port 1
        JMP start       ; and repeat
```

When running this program, best viewed with **Update Freq.** set to **1**.

---

2. Echoing the Switches to the LEDs

```
; This program very simply echoes the
; switches on P2 to the LEDs on P1.

; When a switch is closed a logic 0 appears
; on that P2 pin, which is then copied to
; that P1 bit which turns on that LED.
; Therefore, a closed switch is seen as a lit
; LED and vice versa.

start:
        MOV P1, P2      ; move data on P2 pins to P1
        JMP start       ; and repeat
```

When running this program, best viewed with **Update Freq.** set to **1**.

---

3. Multiplexing the 7-segment Displays

```
; This program multiplexes the number 1234
; on the four 7-segment displays.

; Note: a logic 0 lights a display segment.

start:
        SETB P3.3               ; |
        SETB P3.4               ; | enable display 3
        MOV P1, #11111001B      ; put pattern for 1 on display
        CALL delay
        CLR P3.3                ; enable display 2
        MOV P1, #10100100B      ; put pattern for 2 on display
        CALL delay
        CLR P3.4                ; |
        SETB P3.3               ; | enable display 1
        MOV P1, #10110000B      ; put pattern for 3 on display
        CALL delay
        CLR P3.3                ; enable display 0
        MOV P1, #10011001B      ; put pattern for 4 on display
        CALL delay
        JMP start               ; jump back to start

; a crude delay
delay:
        MOV R0, #200
        DJNZ R0, $
        RET
```

When running this program, best viewed with **Update Freq.** set to **100**.

---

4. LCD Module

The example program for programming the LCD module is written in C. It was developed and compiled using the Keil uVision IDE. If you wish to write your own C programs for the 8051, get the free evaluation version of uVision.

The EdSim51 simulator can **only parse assembly programs**. It cannot compile C programs, therefore do not try to copy and paste the program below into EdSim51. Instead, you should compile the program in uVision3 and use the Intel HEX output file. This type of file can be loaded into EdSim51.

When the program below is running in the simulator, the user can shift the display right and left and can return the cursor home by using switches 5,

6 and 7 (see the comment in the main program for details). Since the simulator is not real-time, to be able to see the display shift left and right, it is best to set the simulator update frequency to 100.

```
#include <reg51.h>

sbit DB7 = P1^7;
sbit DB6 = P1^6;
sbit DB5 = P1^5;
sbit DB4 = P1^4;
sbit RS = P1^3;
sbit E = P1^2;

sbit clear = P2^4;
sbit ret = P2^5;
sbit left = P2^6;
sbit right = P2^7;

void returnHome(void);
void entryModeSet(bit id, bit s);
void displayOnOffControl(bit display, bit cursor, bit blinking);
void cursorOrDisplayShift(bit sc, bit rl);
void functionSet(void);
void setDdRamAddress(char address);

void sendChar(char c);
void sendString(char* str);
bit getBit(char c, char bitNumber);
void delay(void);

void main(void) {

        functionSet();
        entryModeSet(1, 0); // increment and no shift
        displayOnOffControl(1, 1, 1); // display on, cursor on and blinking on
        sendString("EdSim51 LCD Module Simulation");
        setDdRamAddress(0x40); // set address to start of second line
        sendString("Based on Hitachi HD44780");

        // The program can be controlled via some of the switches on port 2.
        // If switch 5 is closed the cursor returns home (address 0).
        // Otherwise, switches 6 and 7 are read - if both switches are open or both switches
        //      are closed, the display does not shift.
        // If switch 7 is closed, continuously shift left.
        // If switch 6 is closed, continuously shift right.
        while (1) {
                if (ret == 0) {
```

```
                        returnHome();
                }
                else {
                        if (left == 0 && right == 1) {
                                cursorOrDisplayShift(1, 0); // shift display left
                        }
                        else if (left == 1 && right == 0) {
                                cursorOrDisplayShift(1, 1); // shift display right
                        }
                }
        }
}

// LCD Module instructions ------------------------------------------
// To understand why the pins are being set to the particular values in the functions
// below, see HD44780.pdf.

void returnHome(void) {
        RS = 0;
        DB7 = 0;
        DB6 = 0;
        DB5 = 0;
        DB4 = 0;
        E = 1;
        E = 0;
        DB5 = 1;
        E = 1;
        E = 0;
        delay();
}

void entryModeSet(bit id, bit s) {
        RS = 0;
        DB7 = 0;
        DB6 = 0;
        DB5 = 0;
        DB4 = 0;
        E = 1;
        E = 0;
        DB6 = 1;
        DB5 = id;
        DB4 = s;
        E = 1;
        E = 0;
        delay();
}
```

```
void displayOnOffControl(bit display, bit cursor, bit blinking) {
        DB7 = 0;
        DB6 = 0;
        DB5 = 0;
        DB4 = 0;
        E = 1;
        E = 0;
        DB7 = 1;mov
        DB6 = display;
        DB5 = cursor;
        DB4 = blinking;
        E = 1;
        E = 0;
        delay();
}

void cursorOrDisplayShift(bit sc, bit rl) {
        RS = 0;
        DB7 = 0;
        DB6 = 0;
        DB5 = 0;
        DB4 = 1;
        E = 1;
        E = 0;
        DB7 = sc;
        DB6 = rl;
        E = 1;
        E = 0;
        delay();
}

void functionSet(void) {
        // The high nibble for the function set is actually sent twice. Why? See 4-bit operation
        // on pages 39 and 42 of HD44780.pdf.
        DB7 = 0;
        DB6 = 0;
        DB5 = 1;
        DB4 = 0;
        RS = 0;
        E = 1;
        E = 0;
        delay();
        E = 1;
        E = 0;
        DB7 = 1;
        E = 1;
        E = 0;
```

```
        delay();
}

void setDdRamAddress(char address) {
        RS = 0;
        DB7 = 1;
        DB6 = getBit(address, 6);
        DB5 = getBit(address, 5);
        DB4 = getBit(address, 4);
        E = 1;
        E = 0;
        DB7 = getBit(address, 3);
        DB6 = getBit(address, 2);
        DB5 = getBit(address, 1);
        DB4 = getBit(address, 0);
        E = 1;
        E = 0;
        delay();
}

void sendChar(char c) {
        DB7 = getBit(c, 7);
        DB6 = getBit(c, 6);
        DB5 = getBit(c, 5);
        DB4 = getBit(c, 4);
        RS = 1;
        E = 1;
        E = 0;
        DB7 = getBit(c, 3);
        DB6 = getBit(c, 2);
        DB5 = getBit(c, 1);
        DB4 = getBit(c, 0);
        E = 1;
        E = 0;
        delay();
}

// -- End of LCD Module instructions
// ------------------------------------------------------------------

void sendString(char* str) {
        int index = 0;
        while (str[index] != 0) {
                sendChar(str[index]);
                index++;
        }
}
```

```
bit getBit(char c, char bitNumber) {
        return (c >> bitNumber) & 1;
}

void delay(void) {
        char c;
        for (c = 0; c < 50; c++);
}
```

LCD Module Assembly Program Example
The example below sends the text *ABC* to the display.

```
; put data in RAM
        MOV 30H, #'A'
        MOV 31H, #'B'
        MOV 32H, #'C'
        MOV 33H, #0     ; end of data marker


; initialise the display
; see instruction set for details


        CLR P1.3                ; clear RS - indicates that instructions are being sent to the module

; function set
        CLR P1.7                ; |
        CLR P1.6                ; |
        SETB P1.5               ; |
        CLR P1.4                ; | high nibble set

        SETB P1.2               ; |
        CLR P1.2                ; | negative edge on E

        CALL delay              ; wait for BF to clear
                                ; function set sent for first time - tells module to go into 4-bit mode
; Why is function set high nibble sent twice? See 4-bit operation on pages 39 and 42 of HD44780.pdf.

        SETB P1.2               ; |
        CLR P1.2                ; | negative edge on E
                                ; same function set high nibble sent a second time

        SETB P1.7               ; low nibble set (only P1.7 needed to be changed)
```

```
        SETB P1.2               ; |
        CLR P1.2                ; | negative edge on E
                                ; function set low nibble sent
        CALL delay              ; wait for BF to clear


; entry mode set
; set to increment with no shift
        CLR P1.7                ; |
        CLR P1.6                ; |
        CLR P1.5                ; |
        CLR P1.4                ; | high nibble set

        SETB P1.2               ; |
        CLR P1.2                ; | negative edge on E

        SETB P1.6               ; |
        SETB P1.5               ; |low nibble set

        SETB P1.2               ; |
        CLR P1.2                ; | negative edge on E

        CALL delay              ; wait for BF to clear


; display on/off control
; the display is turned on, the cursor is turned on and blinking is turned on
        CLR P1.7                ; |
        CLR P1.6                ; |
        CLR P1.5                ; |
        CLR P1.4                ; | high nibble set

        SETB P1.2               ; |
        CLR P1.2                ; | negative edge on E

        SETB P1.7               ; |
        SETB P1.6               ; |
        SETB P1.5               ; |
        SETB P1.4               ; | low nibble set

        SETB P1.2               ; |
        CLR P1.2                ; | negative edge on E

        CALL delay              ; wait for BF to clear
```

```
; send data
        SETB P1.3               ; clear RS - indicates that data is being sent to module
        MOV R1, #30H            ; data to be sent to LCD is stored in 8051 RAM, starting at location 30H
loop:
        MOV A, @R1              ; move data pointed to by R1 to A
        JZ finish               ; if A is 0, then end of data has been reached - jump out of loop
        CALL sendCharacter      ; send data in A to LCD module
        INC R1                  ; point to next piece of data
        JMP loop                ; repeat

finish:
        JMP $


sendCharacter:
        MOV C, ACC.7            ; |
        MOV P1.7, C             ; |
        MOV C, ACC.6            ; |
        MOV P1.6, C             ; |
        MOV C, ACC.5            ; |
        MOV P1.5, C             ; |
        MOV C, ACC.4            ; |
        MOV P1.4, C             ; | high nibble set

        SETB P1.2               ; |
        CLR P1.2                ; | negative edge on E

        MOV C, ACC.3            ; |
        MOV P1.7, C             ; |
        MOV C, ACC.2            ; |
        MOV P1.6, C             ; |
        MOV C, ACC.1            ; |
        MOV P1.5, C             ; |
        MOV C, ACC.0            ; |
        MOV P1.4, C             ; | low nibble set

        SETB P1.2               ; |
        CLR P1.2                ; | negative edge on E

        CALL delay              ; wait for BF to clear

delay:
        MOV R0, #50
        DJNZ R0, $
        RET
```

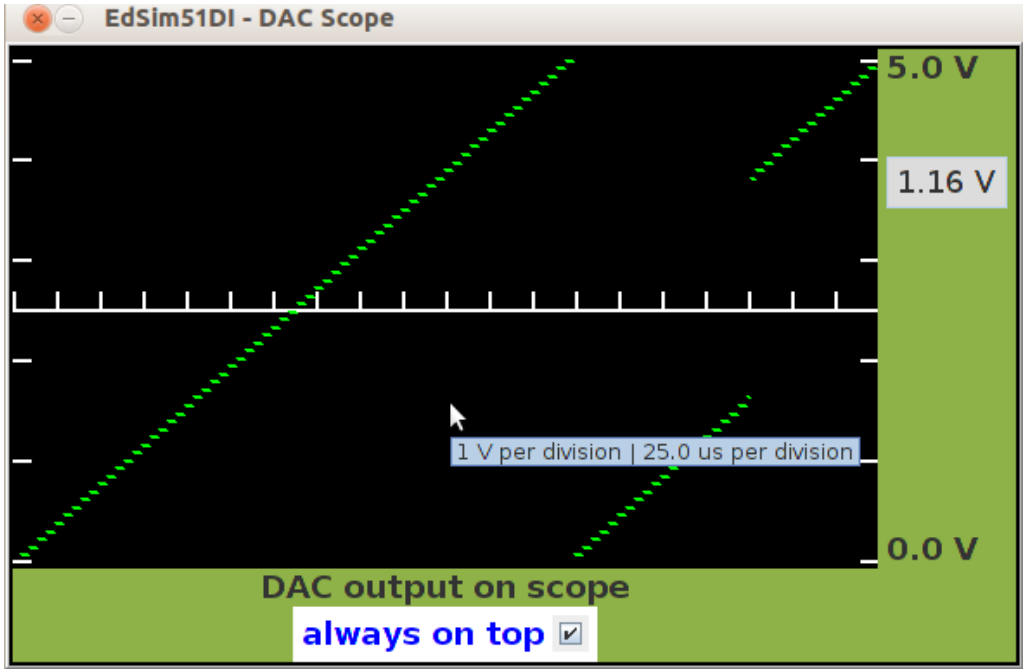
5. Ramp Signal on the DAC Output

```
; This program generates a ramp on the DAC
; output.

; You can try adding values other than 8
; to the accumulator to see what this does
; to the ramp signal.

        CLR P0.7        ; enable the DAC WR line
loop:
        MOV P1, A       ; move data in the accumulator to the ADC inputs (on P1)
        ADD A, #4       ; increase accumulator by 4
        JMP loop        ; jump back to loop
```

When running this program, best viewed with **Update Freq.** set to **1**.
The output on the scope should be something like this:



6. Taking Samples from the ADC and Displaying them on the Scope via the DAC

Note: When using the ADC, the switches in the switch bank must be open (the switches are grey when closed, grey when closed). This is because the switch bank and the ADC share the same port.

```
; This program reads the analogue input
; voltage on the ADC and displays it on
; the scope via the DAC.

; A sample is taken from the ADC every 50 us
; This is achieved by setting timer 0
; to interrupt the main program every 50 us.
; The timer 0 ISR then initiates an ADC
; conversion.

; When the conversion is complete the
; ADC interrupt line goes low. This line
; is interfaced with the 8051 external 0
; interrupt line. The external 0 ISR
; therefore takes the reading from the ADC
; on P2 and passes it to the DAC on P1.

; Therefore, while the program is running,
; the scope voltage level should be the
; same as the ADC input voltage.
; However, when a change is made to the
; ADC input voltage it will take some time
; for the scope to update (ie: until the
; next timer 0 interrupt).

; Note: when running this program make sure
; the ADC is enabled (not the comparator).

ORG 0                          ; reset vector
        JMP main               ; jump to the main program

ORG 3                          ; external 0 interrupt vector
        JMP ext0ISR            ; jump to the external 0 ISR

ORG 0BH                        ; timer 0 interrupt vector
        JMP timer0ISR          ; jump to timer 0 ISR

ORG 30H                        ; main program starts here
main:
        SETB IT0               ; set external 0 interrupt as edge-activated
```

```
        SETB EX0                ; enable external 0 interrupt
        CLR P0.7                ; enable DAC WR line
        MOV TMOD, #2            ; set timer 0 as 8-bit auto-reload interval timer

        MOV TH0, #-50           ; | put -50 into timer 0 high-byte - this reload value,
                                ; | with system clock of 12 MHz, will result in a timer 0 overflow every 50 us

        MOV TL0, #-50           ; | put the same value in the low byte to ensure the timer starts counting from
                                ; | 236 (256 - 50) rather than 0

        SETB TR0                ; start timer 0
        SETB ET0                ; enable timer 0 interrupt
        SETB EA                 ; set the global interrupt enable bit
        JMP $                   ; jump back to the same line (ie: do nothing)

; end of main program


; timer 0 ISR - simply starts an ADC conversion
timer0ISR:
        CLR P3.6                ; clear ADC WR line
        SETB P3.6               ; then set it - this results in the required positive edge to start a conversion
        RETI                    ; return from interrupt


; external 0 ISR - responds to the ADC conversion complete interrupt
ext0ISR:
        CLR P3.7                ; clear the ADC RD line - this enables the data lines
        MOV P1, P2              ; take the data from the ADC on P2 and send it to the DAC data lines on P1
        SETB P3.7               ; disable the ADC data lines by setting RD
        RETI                    ; return from interrupt
```

When running this program with **Update Freq.** set to **1**, the student will observe the time delay between a change in the ADC input voltage appearing on the scope. In order to see this change appear on the scope more quickly, select a higher update frequency.

A Further exercise for the student might be to display, using multiplexing, the actual input signal voltage to the ADC on the 7-segment displays.

---

7. Scanning the Keypad

The following program shows how to scan the keypad. The program halts (to be precise, sits in an endless loop) once a key is pressed.

```
; This program scans the keypad.

; While no key is pressed the program
; scans row0, row1, row2, row3 and back to
; row0, continuously.

; When a key is pressed the key number
; is placed in R0.

; For this program, the keys are numbered
; as:

;       +----+----+----+
;       | 11 | 10 |  9 |       row3
;       +----+----+----+
;       |  8 |  7 |  6 |       row2
;       +----+----|----+
;       |  5 |  4 |  3 |       row1
;       +----+----+----+
;       |  2 |  1 |  0 |       row0
;       +----+----+----+
;        col2 col1 col0

; The pressed key number will be stored in
; R0. Therefore, R0 is initially cleared.
; Each key is scanned, and if it is not
; pressed R0 is incremented. In that way,
; when the pressed key is found, R0 will
; contain the key's number.

; The general purpose flag, F0, is used
; by the column-scan subroutine to indicate
; whether or not a pressed key was found
; in that column.
; If, after returning from colScan, F0 is
; set, this means the key was found.

start:

        MOV R0, #               ; clear R0 - the first key is key0

        ; scan row0
        SETB P0.3               ; set row3
        CLR P0.0                ; clear row0
        CALL colScan            ; call column-scan subroutine
        JB F0, finish           ; | if F0 is set, jump to end of program
                                ; | (because the pressed key was found and its number is in  R0)
```

```
        ; scan row1
        SETB P0.0               ; set row0
        CLR P0.1                ; clear row1
        CALL colScan            ; call column-scan subroutine
        JB F0, finish           ; | if F0 is set, jump to end of program
                                ; | (because the pressed key was found and its number is in  R0)

        ; scan row2
        SETB P0.1               ; set row1
        CLR P0.2                ; clear row2
        CALL colScan            ; call column-scan subroutine
        JB F0, finish           ; | if F0 is set, jump to end of program
                                ; | (because the pressed key was found and its number is in  R0)

        ; scan row3
        SETB P0.2               ; set row2
        CLR P0.3                ; clear row3
        CALL colScan            ; call column-scan subroutine
        JB F0, finish           ; | if F0 is set, jump to end of program
                                ; | (because the pressed key was found and its number is in  R0)


        JMP start               ; | go back to scan row 0
                                ; | (this is why row3 is set at the start of the program
                                ; | - when the program jumps back to start, row3 has just been scanned)

finish:
        JMP $                   ; program execution arrives here when key is found - do nothing


; column-scan subroutine
colScan:
        JNB P0.4, gotKey        ; if col0 is cleared - key found
        INC R0                  ; otherwise move to next key
        JNB P0.5, gotKey        ; if col1 is cleared - key found
        INC R0                  ; otherwise move to next key
        JNB P0.6, gotKey        ; if col2 is cleared - key found
        INC R0                  ; otherwise move to next key
        RET                     ; return from subroutine - key not found
gotKey:
        SETB F0                 ; key found - set F0
        RET                     ; and return from subroutine
```

It may appear as if this program does nothing. Remember, the program simply scans the keypad until a key is pressed. It then places the key number in R0 and enters an endless loop, doing nothing. Therefore, to see that it has performed the task correctly, examine the contents of R0 after a key is

pressed. Also remember that, if the update frequency is set to 1, it will take a short amount of time for they pressed key's number to appear in R0.

Key number: the key number mentioned here is not the number **on** the key, but the key's position in the matrix. The # key is key number 0, 0 key is key number 1, * key is key number 2, and so on.

Further exercises for the student might be:

- Modify the program so that the keypad is scanned continuously (ie: it doesn't stop after one key-press).
- Write extra code that displays the key symbol on one of the 7-segment displays. For example, if key 4 is pressed, the number 8 appears on the display. If key 10 is pressed the number 2 appears on the display. (Note: the symbols # and * cannot be displayed on a 7-segment display, but some special characters could be invented and displayed instead.

---

8. Transmitting Data on the 8051 Serial Port

```
; This program sends the text abc down the
; 8051 serial port to the external UART at 4800 Baud.

; To generate this baud rate, timer 1 must overflow
; every 13 us with SMOD equal to 1 (this is as close as
; we can get to 4800 baud at a system clock frequency
; of 12 Mz).

; The data is sent with even parity,
; therefore for it to be received correctly
; the external UART must be set to Even Parity


        CLR SM0                 ; |
        SETB SM1                ; | put serial port in 8-bit UART mode

        MOV A, PCON             ; |
        SETB ACC.7              ; |
        MOV PCON, A             ; | set SMOD in PCON to double baud rate

        MOV TMOD, #20H          ; put timer 1 in 8-bit auto-reload interval timing mode
        MOV TH1, #243           ; put -13 in timer 1 high byte (timer will overflow every 13 us)
        MOV TL1, #243           ; put same value in low byte so when timer is first started it will overflow after 13 us
        SETB TR1                ; start timer 1

        MOV 30H, #'a'           ; |
```

```
        MOV 31H, #'b'           ; |
        MOV 32H, #'c'           ; | put data to be sent in RAM, start address 30H

        MOV 33H, #0             ; null-terminate the data (when the accumulator contains 0, no more data to be sent)
        MOV R0, #30H            ; put data start address in R0
again:
        MOV A, @R0              ; move from location pointed to by R0 to the accumulator
        JZ finish               ; if the accumulator contains 0, no more data to be sent, jump to finish
        MOV C, P                ; otherwise, move parity bit to the carry
        MOV ACC.7, C            ; and move the carry to the accumulator MSB
        MOV SBUF, A             ; move data to be sent to the serial port
        INC R0                  ; increment R0 to point at next byte of data to be sent
        JNB TI, $               ; wait for TI to be set, indicating serial port has finished sending byte
        CLR TI                  ; clear TI
        JMP again               ; send next byte
finish:
        JMP $                   ; do nothing
```

Take note of what happens if the external UART is not set to Even Parity (ie: run the program with the UART on **`No Parity`**, then run it again with it set to
`Odd Parity`). With the help of the ASCII table, see if you can explain the actual data that is displayed.

It takes quite some time (from the 8051's *perspective* of time) for data to be sent to the UART. If the user wishes to see the data arrive at the UART quickly, it may be best, when running this program, to select an **Update Freq.** of **100** or **1000**.

Further exercises for the student might be:

- Modify the program to transmit data with no parity.
- Modify the program to transmit data with odd parity.
- The above program uses busy-waiting. In other words, it sends a byte to the serial port, then sits in a loop (testing the TI flag) waiting for the serial port to say it's ready for another byte. Rewrite the program making use of the serial port interrupt.

---

9. Receiving Data on the Serial Port

```
; This program receives data on the 8051
; serial port. Once the program's
; initialisation is complete, it waits for
; data arriving on the RXD line (data
; that was transmitted by the external
; UART).

; Any text typed in the external UART is
; sent once the Tx Send button is pressed.
; As each character is transmitted fully,
; it disappears from the Tx window.
; The data is appended with the return
; character (0D HEX).

; The default external UART baud rate is 19,200. To
; generate this baud rate, TH1 must be set to -3.
; If the system clock is 12 MHz, the error when attempting
; to generate such a high baud rate such that very often
; garbage is received. Therefore, for 19,200 Baud, the
; system clock should be set to 11.059 HMz.

; The program is written using busy-waiting.
; It continuously tests the RI flag. Once
; this flag is set by the 8051 hardware
; (a byte has been received) the program
; clears it and then moves the byte from
; SBUF to A.
; The received byte's value is checked to
; see if it is the terminating character (0D HEX).
; If it is the program jumps to the finish,
; otherwise it moves the byte to data memory
; and returns to waiting for the next byte.

        CLR SM0                 ; |
        SETB SM1                ; | put serial port in 8-bit UART mode

        SETB REN                ; enable serial port receiver

        MOV A, PCON             ; |
        SETB ACC.7              ; |
        MOV PCON, A             ; | set SMOD in PCON to double baud rate

        MOV TMOD, #20H          ; put timer 1 in 8-bit auto-reload interval timing mode
        MOV TH1, #0FDH          ; put -3 in timer 1 high byte (timer will overflow every 3 us)
        MOV TL1, #0FDH          ; put same value in low byte so when timer is first started it will overflow after
approx. 3 us
```

```
        SETB TR1                ; start timer 1
        MOV R1, #30H            ; put data start address in R1
again:
        JNB RI, $               ; wait for byte to be received
        CLR RI                  ; clear the RI flag
        MOV A, SBUF             ; move received byte to A
        CJNE A, #0DH, skip      ; compare it with 0DH - it it's not, skip next instruction
        JMP finish              ; if it is the terminating character, jump to the end of the program
skip:
        MOV @R1, A              ; move from A to location pointed to by R1
        INC R1                  ; increment R1 to point at next location where data will be stored
        JMP again               ; jump back to waiting for next byte
finish:
        JMP $                   ; do nothing
```

Further exercises for the student might be:

- Modify the program to receive data with even parity.
- Modify the program to receive data with odd parity.
- Using parity checking, light LED0 (on port 1) if an error is detected in a received byte.
- The above program uses busy-waiting. In other words, it sits in a loop (testing the RI flag) waiting for a byte to be received. Rewrite the program making use of the serial port interrupt.
- Combine the transmit and receive programs to send back the data received from the external UART. In other words, whatever is sent from the UART **Tx** window should appear in the UART **Rx** window.

It takes quite some time (from the 8051's *perspective* of time) for data to be received from the UART. If the user wishes to see the data arrive in 8051 RAM quickly, it may be best, when running this program, to select an **Update Freq.** of **100** or **1000**.

10. The Motor

```
; This program exercises the motor.
; The motor is rotated in a clockwise
; direction and the number of revolutions
; is displayed on Display 0 (the 7-segment
; display). The display only shows up to
; nine revolutions and then resets.

; The motor sensor is connected to P3.5,
; which is the external clock source for
; timer 1. Therefore, timer 1 is put into
; event counting mode. In this way, the
; the timer increments once every motor
; revolution.

; The value in timer 1 low byte is moved
; to A and this value together with the
; data pointer (DPH and DPL) are used to
; get the 7-segment code from program memory.
; The code is then sent to P1 to put the
; appropriate number on the Display 0.

; The motor can be changed from clockwise
; to anti-clockwise by pressing SW0 (on P2.0).
; The motor direction is stored in F0 (1 for
; clockwise, 0 for anti-clockwise). The
; value in F0 is sent to Display 0's decimal
; point (P1.7). This indicates the motor's
; direction - if the decimal point is lit,
; the motor is rotating anti-clockwise, while
; if it is not lit the motor is rotating
; clockwise.

; The value in F0 is compared with the
; value of SW0. If they are the same the
; motor direction does not need to be
; changed. If they are not the same it means
; the user has pressed SW0 and the motor
; direction must be reversed. When this
; happens the new motor direction is then
; stored in F0.


        MOV TMOD, #50H                  ; put timer 1 in event counting mode
        SETB TR1                        ; start timer 1
```

```
        MOV DPL, #LOW(LEDcodes)               ; | put the low byte of the start address of the
                                              ; | 7-segment code table into DPL

        MOV DPH, #HIGH(LEDcodes)              ; put the high byte into DPH

        CLR P3.4                              ; |
        CLR P3.3                              ; | enable Display 0
again:
        CALL setDirection                     ; set the motor's direction
        MOV A, TL1                            ; move timer 1 low byte to A
        CJNE A, #10, skip                     ; if the number of revolutions is not 10 skip next instruction
        CALL clearTimer                       ; if the number of revolutions is 10, reset timer 1
skip:
        MOVC A, @A+DPTR                       ; | get 7-segment code from code table - the index into the table is
                                              ; | decided by the value in A
                                              ; | (example: the data pointer points to the start of the
                                              ; | table - if there are two revolutions, then A will contain two,
                                              ; | therefore the second code in the table will be copied to A)

        MOV C, F0                             ; move motor direction value to the carry
        MOV ACC.7, C                          ; and from there to ACC.7 (this will ensure Display 0's decimal point
                                              ; will indicate the motor's direction)

        MOV P1, A                             ; | move (7-seg code for) number of revolutions and motor direction
                                              ; | indicator to Display 0

        JMP again                             ; do it all again

setDirection:
        PUSH ACC                              ; save value of A on stack
        PUSH 20H                              ; save value of location 20H (first bit-addressable
                                              ;       location in RAM) on stack
        CLR A                                 ; clear A
        MOV 20H, #0                           ; clear location 20H
        MOV C, P2.0                           ; put SW0 value in carry
        MOV ACC.0, C                          ; then move to ACC.0
        MOV C, F0                             ; move current motor direction in carry
        MOV 0, C                              ; and move to LSB of location 20H (which has bit address 0)

        CJNE A, 20H, changeDir                ; | compare SW0 (LSB of A) with F0 (LSB of 20H)
                                              ; | - if they are not the same, the motor's direction needs to be reversed

        JMP finish                            ; if they are the same, motor's direction does not need to be changed
changeDir:
        CLR P3.0                              ; |
        CLR P3.1                              ; | stop motor
```

```
        CALL clearTimer             ; reset timer 1 (revolution count restarts when motor direction changes)
        MOV C, P2.0                 ; move SW0 value to carry
        MOV F0, C                   ; and then to F0 - this is the new motor direction
        MOV P3.0, C                 ; move SW0 value (in carry) to motor control bit 1
        CPL C                       ; invert the carry

        MOV P3.1, C                 ; | and move it to motor control bit 0 (it will therefore have the opposite
                                    ; | value to control bit 1 and the motor will start
                                    ; | again in the new direction)
finish:
        POP 20H                     ; get original value for location 20H from the stack
        POP ACC                     ; get original value for A from the stack
        RET                         ; return from subroutine

clearTimer:
        CLR A                       ; reset revolution count in A to zero
        CLR TR1                     ; stop timer 1
        MOV TL1, #0                 ; reset timer 1 low byte to zero
        SETB TR1                    ; start timer 1
        RET                         ; return from subroutine

LEDcodes:       ; | this label points to the start address of the 7-segment code table which is
                ; | stored in program memory using the DB command below
        DB 11000000B, 11111001B, 10100100B, 10110000B, 10011001B, 10010010B, 10000010B, 11111000B, 10000000B, 10010000B
```

Note: The above program is a good illustration of what can go wrong if the sampling frequency is too low. Try running the program with the motor at full speed (use the slider to the right of the motor to increase the motor speed). You should notice the motor completes a number of revolutions before the display updates. In other words, the motor's highest speed is too fast for the 8051 running at system clock of 12 MHz.

Since the program only shows up to nine revolutions (displayed on the 7-segment display) and then starts counting again, it is best to run this program with an **Update Freq.** of **1**. This allows the user observe the display count up from 0 to 9 and back to 0 again. It also illustrates the delay between the motor arm passing the sensor and the display update. Similarly, the delay between pressing the switch for a direction change and the actual direction change occurring is evident. The student will learn that, while in real time these delays are not noticeable, to the microcontroller these operations take quite some time.

Further exercises for the student might be:

- With the above program only up to nine revolutions are counted, then the display resets. Use multiplexing on the 7-segment displays to count up to 99 revolutions. (Remember, when testing your program you do not have to sit idly by waiting for the motor to revolve a hundred times. You can pause the program, manually change the value in the timer 1 low byte - say to 98 - and then click **Run** to start the program again.)