

11 Interfacing

This chapter deals with interfacing various devices to the 8051 family of micro-controllers. The list here is endless but the basic add-ons such as simple LEDs, switches, keypads, LCDs, DC motors (including servos and stepper motors) are all well covered with example programs.

11.1 Interfacing add-ons to the 8051

The 8051 on its own can be of little use unless we somehow manage to connect it to the real world. Minimally we would need some form of output device, such as an LED or a buzzer and an input interface which might even be a simple ON-OFF switch. Before going further, let us mention two important notes:

- A common fault when interfacing devices (even if simple) or other boards to the 8051 is to forget to connect the ground of the external device to the ground of the 8051 board. This would result in floating signals which would give indeterminate results.
- We should also remember when using the 8051 ports that port 0 needs external pull-up resistors whilst ports 1, 2 and 3 do not need any since they have them already internally wired. These pull-up resistors are not always shown in the following diagrams since it depends to which port we are connecting the interface circuit.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"
Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



With these notes in mind, we can list and describe a number of interface components which we can connect to the 8051.

11.2 LEDs

The simplest output indicator which we can connect would be a Light Emitting Diode (LED). We can connect the LED to either light up when the port pin is High (see Figure 11-1) or to light up when the port pin is Low (see Figure 11-2).

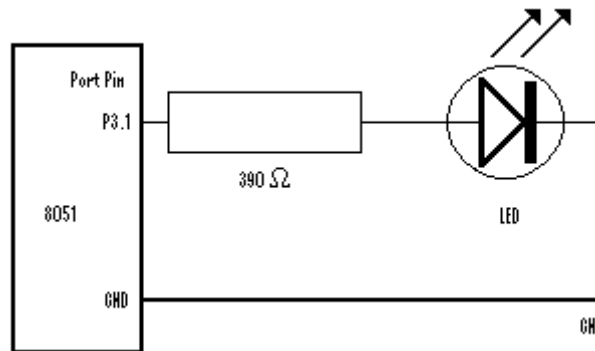


Figure 11-1 Port Driving LED (pin High = LED on)

The option shown in Figure 11-2 is better since the port is being used to sink the current rather than providing the source voltage.

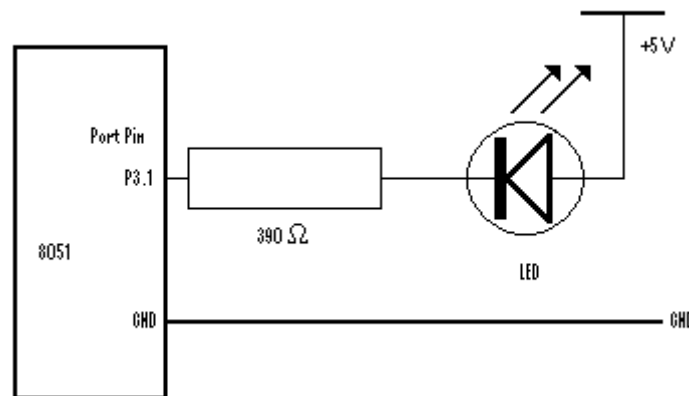


Figure 11-2 Port Sinking LED (pin Low = LED on)

We now list a section of code in C using Keil μ Vision4 for the circuit shown in **Figure 11-2**.

It will flicker the LED, switching it off for 1 second and then on for another second and so on until the microcontroller circuit is switched off.

```
#include <REG52.H>
void msdelay( unsigned int );

sbit LED1 = P3^1;      // refer to bit P3.1 (port 3 bit 1) as LED1

#define led_on 0
#define led_off 1

void main(){

    LED1=0;           // set pin 1 of PORT3 as output

    while(1){        //infinite loop

        LED1 = 1;     //pins high, LED is off, or use LED1 = led_off;
        msdelay(250); // some delay
        LED1 = 0;     // pin low, LEDs are on, or use LED1 = led_on;
        msdelay(250); // some delay

    }

}

//delay function
void msdelay(unsigned int value){

    unsigned int x,y;
    for(x=0;x<value;x++)
        for(y=0;y<1275;y++);
}

}
```

In C programs we cannot be sure of software delays, because they depend a lot on how the compiler optimizes the loops. As soon as we make some changes in the compiling options, the delay time changes.

A better option would be to use the in-built micro-controller timers if we want to have exact delays. Shown below is a function equivalent to a 1 second delay using timer 0, assuming we have an 11.0592 MHz crystal clock driving the micro-controller. The idea is to make a 50ms timer delay and repeat it for 20 times ($20 \times 50\text{ms} = 1000\text{ms} = 1\text{s}$) so as to obtain the required one second delay. The timer would be counting at the rate of $12/11.0592$ micro-seconds per count. Thus we need 46080 counts to get the required 50ms delay, and therefore, as we recall, we need to load the timer with the value of 19456 (which is $65536 - 46080$) or 4C00 hex since our timer would be counting up until it overflows.

```

delay_1s()                // using Timer 0 to get a 1 sec delay
{
int d;

TMOD &= 0xF0;            // clear Timer 0 mode settings, temporarily to mode 0
TMOD |= 0x01;           // set Timer 0 in mode 1, 16-bit
TF0 = 0;                // clear Timer 0 overflow flag

for (d=0; d<=20; d++)    // repeat 20 times
{
    TL0 = 0x00;          // load it for 50ms overflow delay
    TH0 = 0x4C;          // 4C00 hex = 19456
    TR0 = 1;            // start Timer 0.
    while (TF0 == 0);   // run until TF0 = 1, indicating overflow, waiting 50ms

    TR0 = 0;            // stop Timer 0
    TF0 = 0;            // reset the Timer 0 overflow flag
}
}

```

This type of problem is very simple to write using the PaulOS RTOS. Just one task would be needed to implement this LED flickering action:

What do you want to do?

No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site www.volvogroup.com. We look forward to getting to know you!

VOLVO
AB Volvo (publ)
www.volvogroup.com

VOLVO TRUCKS | RENAULT TRUCKS | MACK TRUCKS | VOLVO BUSES | VOLVO CONSTRUCTION EQUIPMENT | VOLVO PENTA | VOLVO AERO | VOLVO IT
VOLVO FINANCIAL SERVICES | VOLVO 3P | VOLVO POWERTRAIN | VOLVO PARTS | VOLVO TECHNOLOGY | VOLVO LOGISTICS | BUSINESS AREA ASIA

Download free eBooks at bookboon.com



Click on the ad to read more

```

void Task_LED (void) {

while(1){          //infinite loop

    LED1 = 1;          //pins high, LED is off, or use LED1 = led_off;
    OS_WAITT_A(0,0, 250); // 250 millisecond delay
    LED1 = 0;          // pin low, LEDs are on, or use LED1 = led_on;
    OS_WAITT_A(0,0, 250); // 250 millisecond delay

    }

}

```

11.2.1 Seven-Segment LED Displays

Another simple output indicator which we can use is the familiar 7-segment LED display. There are basically two types of such displays, either the so-called Common Cathode (all the cathodes or negative connections are connected together to one common ground [GND] terminal) or the Common Anode type where all the anodes (or positive connections) are connected to one common supply [Vcc] terminal as shown in Figure 11-3. Apart from the 7 segments (a-g) forming the digit, some displays have an optional 8th segment which we could use to represent a decimal point (dp).

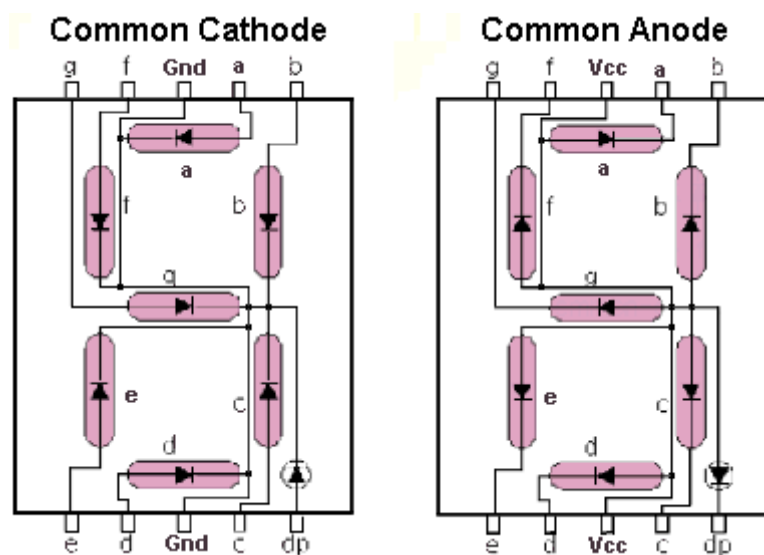
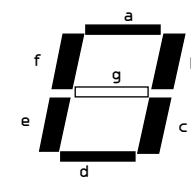
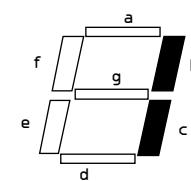
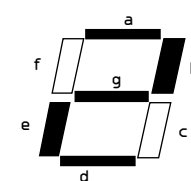
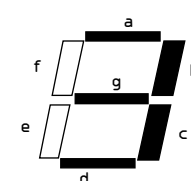
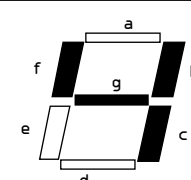
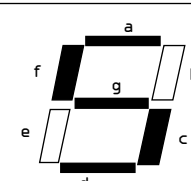


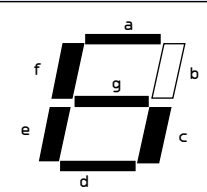
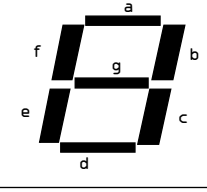
Figure 11-3 7-segmet LED displays

In order to switch on the required decimal digit, we can connect the 7 or 8 segment diodes to the 8-bit port of the 8051 as we have already seen in the example with just one LED in **Figure 11-2**.

The software would be written in such a way so as to switch on the required LEDs to display our decimal number. Thus to display the number 3, we would need to light up segments a, b, c, d and g and switch off the other segments. We should remember that with this direct drive method, the port must keep on presenting the same data to the 7-segment display, otherwise the display would change.

The following Table 11-1 shows how we can display the various digits. The 2nd and 3rd column in this Table shows the output byte for the port, depending on the way the segments are connected to the port..

| Digit | gfedcba 6543210 | abcdefg 6543210 | a | b | c | d | e | f | g | |
|-------|--------------------|--------------------|-----|-----|-----|-----|-----|-----|-----|---|
| 0 | 0x3F | 0x7E | on | on | on | on | on | on | off |  |
| 1 | 0x06 | 0x30 | off | on | on | off | off | off | off |  |
| 2 | 0x5B | 0x6D | on | on | off | on | on | off | on |  |
| 3 | 0x4F | 0x79 | on | on | on | on | off | off | on |  |
| 4 | 0x66 | 0x33 | off | on | on | off | off | on | on |  |
| 5 | 0x6D | 0x5B | on | off | on | on | off | on | on |  |

| Digit | gfedcba 6543210 | abcdefg 6543210 | a | b | c | d | e | f | g | |
|-------|--------------------|--------------------|-----|-----|-----|-----|-----|-----|-----|---|
| 6 | 0x7D | 0x5F | on | off | on | on | on | on | on |  |
| 7 | 0x07 | 0x70 | on | on | on | off | off | off | off |  |
| 8 | 0x7F | 0x7F | on | on | on | on | on | on | on |  |
| 9 | 0x6F | 0x7B | on | on | on | on | off | on | on |  |
| A | 0x77 | 0x77 | on | on | on | off | on | on | on |  |
| b | 0x7C | 0x1F | off | off | on | on | on | on | on |  |
| C | 0x39 | 0x4E | on | off | off | on | on | on | off |  |
| d | 0x5E | 0x3D | off | on | on | on | on | off | on |  |

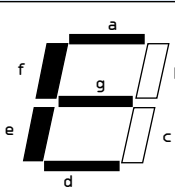
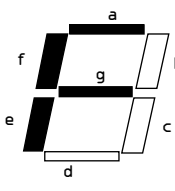
| Digit | gfedcba 6543210 | abcdefg 6543210 | a | b | c | d | e | f | g | |
|-------|--------------------|--------------------|----|-----|-----|-----|----|----|----|---|
| E | 0x79 | 0x4F | on | off | off | on | on | on | on |  |
| F | 0x71 | 0x47 | on | off | off | off | on | on | on |  |

Table 11-1 LED 7 segment connections

We can also multiplex more than one 7-segment display by using a circuit as shown in Figure 11-4. One port supplies the data to all the displays, whilst the transistors T1–T4 switch on one display at a time as programmed by port 2. The first digit display would be left on for a few milliseconds and then switched off. The data is then changed to reflect the second digit display which is then switched on also for a few milliseconds. All the digits would be similarly switched on and off and this strobing action is repeated indefinitely so as to the viewer all the displays would appear to be lighted up continuously. A sample code program is listed to describe the program flow. We could also write the program using an RTOS where a OS_WAITT_A() command would be used to replace the delay function, thus the processor can be doing something else while waiting and driving the display.

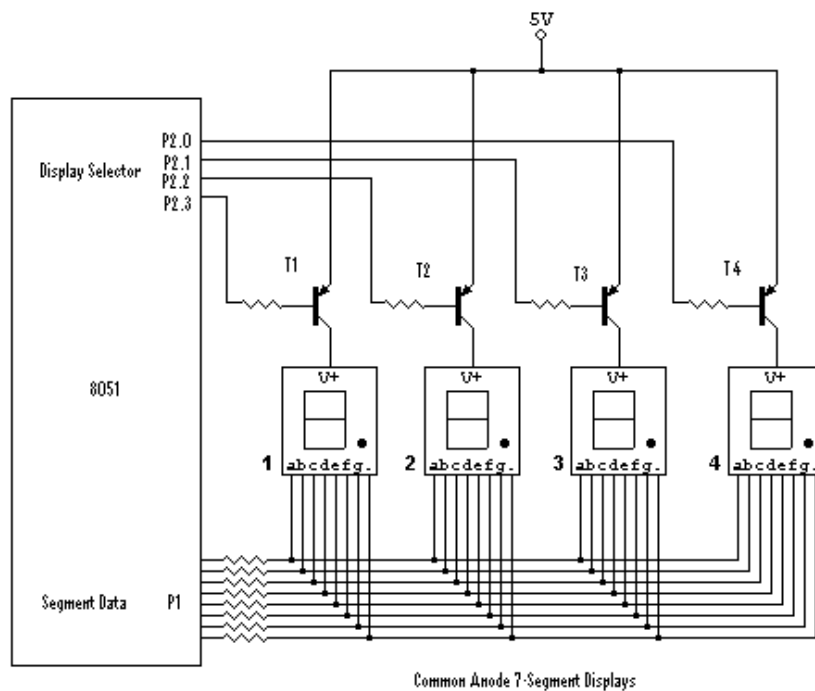


Figure 11-4 Multiplexing displays


```
sbit digit0 = P2^0;
sbit digit1 = P2^1;
sbit digit2 = P2^2;
sbit digit3 = P2^3;

// Assuming segment a is connected to bit P1.6, segment b to bit P1.5 etc, then from Table 11-1
// we can select the segments to light up for each decimal digit 0–9 by sending the correct
// segment data from the array segment[].
// The digit can be selected by outputting a 1 on ONE pin from P2.0 to P2.3
unsigned char segments[10] = {0x7E, 0x30, 0x6D, 0x79, 0x33, 0x5B, 0x5F,
                             0x70, 0x7F, 0x7B};

P2 &= 0xF0;          // switch off all digits
while(1)            // keep on looping
{
P1 = segments[0];    // send data to reflect the segments which need to be lighted up
// in this case the number shown would be 0
digit0 = 1;         // switch on digit 0
delay();            // wait for some time, calling the delay function
digit0 = 0;         // switch off digit 0
// Now repeat for the second 7 segment LED digit
P1 = segments[1];    // send data to reflect the segments which need to be lighted up
// in this case the number shown would be 1
Digit1 = 1;         // switch on digit 1
delay();            // wait for some time, calling the delay function
digit1 = 0;         // switch off digit 1
// and so on for the other digits.
.....
.....
}
```

To make programming easier and at the same time provide a data latching (memory) capability, avoiding the need to keep on strobing the data, various 7-segment driver ICs were developed, the 4511 being one of them. These generally have 4 data input pins (D_1 to D_4) to represent the digit number which we want to display, D_1 being the least significant bit. Some are decimal drivers, accepting a 4-bit BCD (binary coded decimal number 0–9). Numbers greater than 9 (10–15) would show as blank. There are also Hex drivers which can display the normal 0–9 decimal digits and also a, b, c, d, e and f with the limitations of the 7-segment display. Thus A, C (not all drivers), E and F are shown as capital letters, whereas b, d (and sometimes c) are shown as small letters. The latching (latch enable or LE pin) mechanism ensures that once the data is latched on the IC (by putting LE low for a few micro-seconds, done in software by setting the port pin which is connected to this LE terminal from high to low and then back to high), then there is no need to keep the data at the 4511 input pins; the display would remain showing the latched digit data until some new data is latched to that same LED driver.

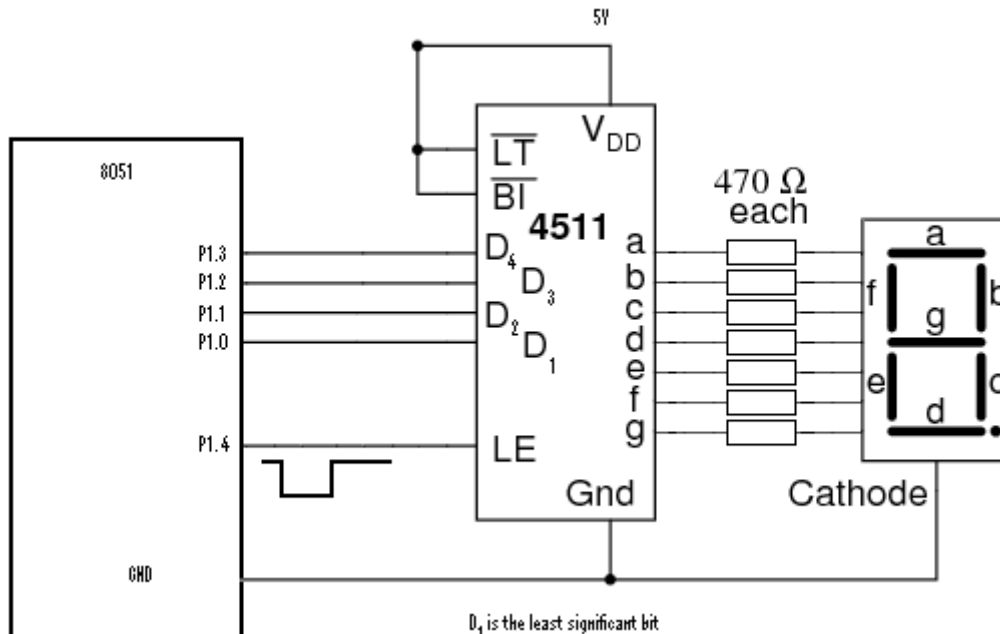


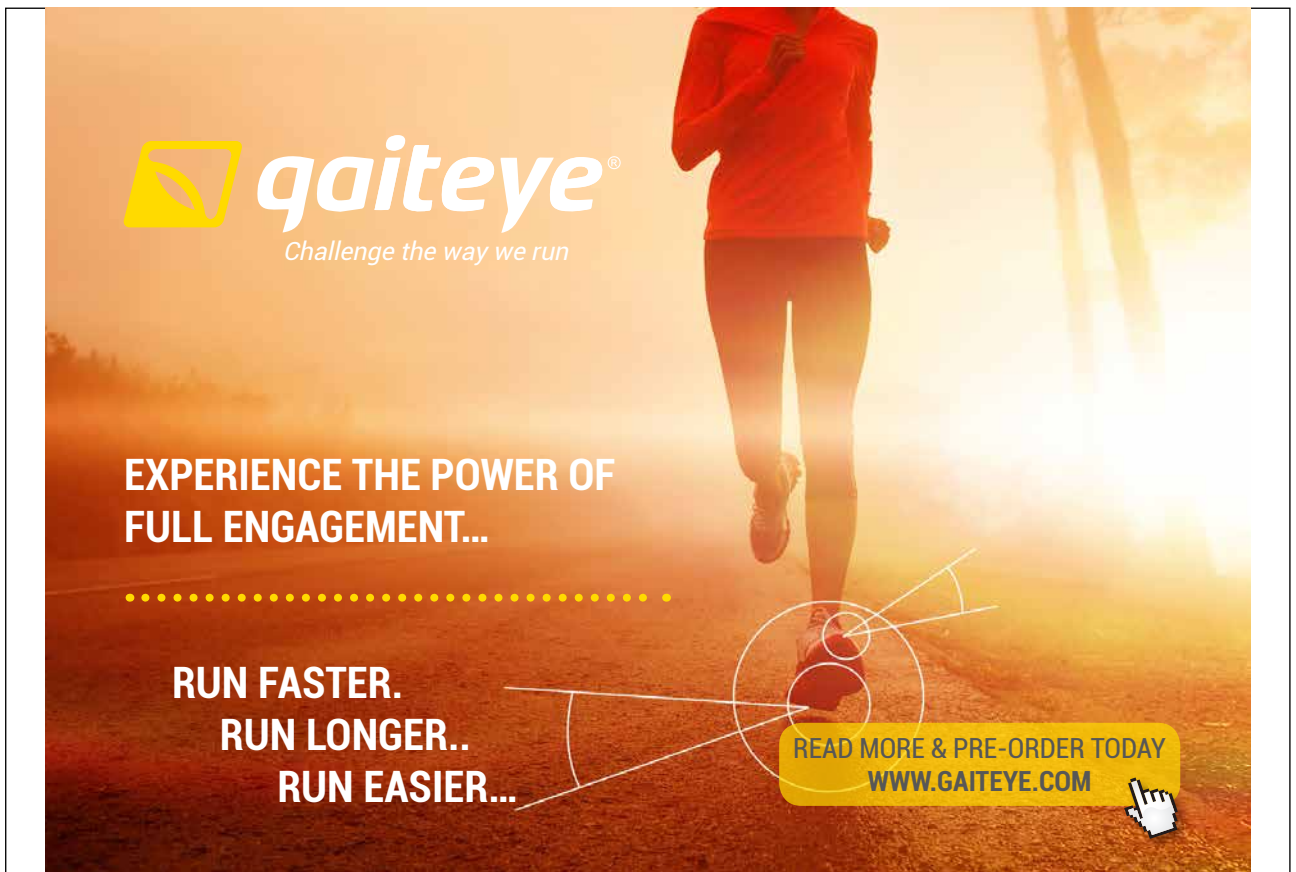
Figure 11-5 LED BCD driver

If we have the port P1 connections as shown in Figure 11-5 then we can display the number 7 with the following simple C code:

```
sbit LE = P1^4;
LE = 1;           // ensure latch is High
P1 &= 0xF0;      // clear lower 4 data bits
P1 |= 0x07;      // set the correct data bits (in this case 7)
LE = 0;         // toggle the Latch Enable bit
LE = 1;
```

Two other control pins are usually available. The LT (lamp test) pin is usually used just to check that all the segments are working, and when set to low, the number 8 is displayed, irrespective of the D_1 - D_4 input conditions. The BL (blanking input) pin is used to blank the display and is usually used to blank the leading zeroes in a multi-digit display. If not required, these two control signals are usually connected directly to the positive supply as shown in Figure 11-5.

We can also use the latch enable pin to multiplex more than one digit display to the same port. By latching sequentially different 7-segment digits, we can easily have a 6-digit display to use as a clock to display HH:MM:SS (the colon [:] can be obtained by using 4 separate LEDs, permanently on). Figure 11-6 shows how we can connect two 7 segment displays using the 4511 BCD-to-7-segment driver, which we can easily extend to more digits as required. The BCD data coming out of pins P1.0 to P1.3 is common to all the digits but the display is selected by pulsing the correct LE pin, using P1.4 or P1.5



The advertisement features a background image of a person running on a path during a sunrise or sunset. The GaiTeye logo is in the top left, with the tagline 'Challenge the way we run'. The main text reads 'EXPERIENCE THE POWER OF FULL ENGAGEMENT...' followed by 'RUN FASTER. RUN LONGER.. RUN EASIER...'. A yellow button in the bottom right corner says 'READ MORE & PRE-ORDER TODAY' and 'WWW.GAITEYE.COM' with a hand cursor icon.

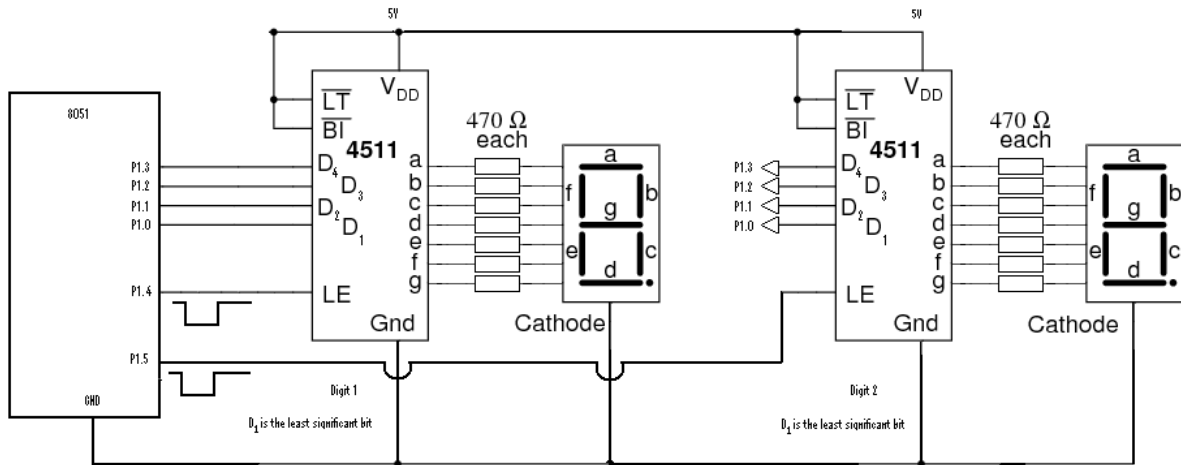


Figure 11-6 Multiplexing 4511s

For the circuit shown above in Figure 11-6 we can easily write a function which will handle everything:

```

sbit LE1 = P1^4;
sbit LE2 = P1^5;

Void Display(unsigned char Digit, unsigned char BCD_Data) {
    P1 &= 0xF0;      // clear lower 4 bits
    P1 |= BCD_Data;  // place data on output lines
    if (Digit == 1) {
        LE1 = 1;     // latch data to digit 1
        LE1 = 0;
        LE1 = 1;
    }
    if (Digit == 2) {
        LE2 = 1;     // latch data to digit 2
        LE2 = 0;
        LE2 = 1;
    }
}

```

11.3 Input Switches

The simplest input is the switch, as shown in Figure 11-7. Here we can easily see that whenever the switch is open, the microcontroller port pin would be effectively connected to the 5V supply through the 10k ohm resistor. The microcontroller would read a high logic level or a 1. Closing the switch would ground the pin and the microcontroller would read a zero logic level. The port pin would be programmed for the input mode by initially writing a 1 to that pin.

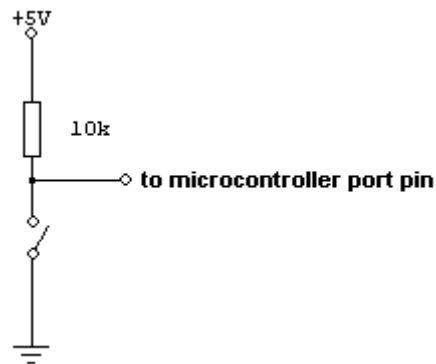


Figure 11-7 Switch (normally open, high on port pin)

On the other hand, in Figure 11-8 we can easily see that whenever the switch is open (normal position), the microcontroller port pin would be effectively connected to the ground through the 10k ohm resistor. The microcontroller would read a low logic level or a 0. Closing the switch would connect the pin to the 5V rail and the microcontroller would read a high logic level or a 1.

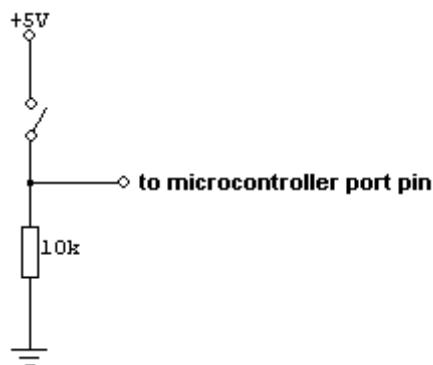


Figure 11-8 Switch (normally open, low on port pin)

11.3.1 Switch Bounce

When a physical switch is closed the contacts bounce opened and closed rapidly for about 20 to 30 ms, as illustrated below in Figure 11-9. The opening of a switch is normally clean and without bounce.

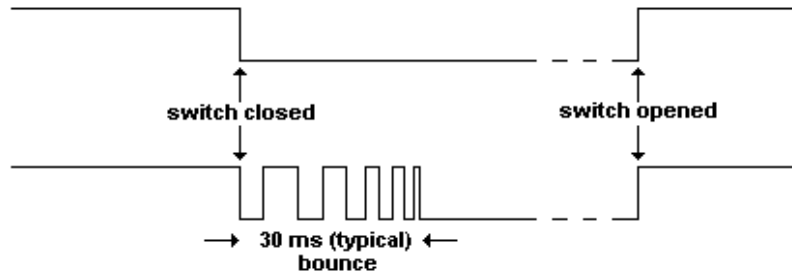


Figure 11-9 Switch bounce

While switch-close bounce is a very short time in human terms it is a very long time for a micro-controller (the basic 8051 running on a system clock of 12 MHz executes a 1-byte instruction in 1 μ s). Without switch de-bouncing, the microcontroller would 'think' the switch was opened and closed repeatedly. Imagine if a push-button switch was being used to increment the output to a digital to analogue convertor. The software routine to poll the push button switch (expecting an off-on-off action on the push switch, returning a one when pressed, otherwise wait) would normally be:

- Wait while the switch is off
- Wait while the switch is on
- Switch can now be taken as pressed (off-on-off) and return a '1'



The main program would then normally stay in the endless:

```
{  
    Call switch polling routine (outlined above)  
    Increment voltage routine  
}
```

If the switch was connected to the microcontroller without any switch de-bouncing mechanism, then a user pressing the switch once would actually result in the DAC output voltage being increased many times because the microcontroller would respond as if the switch had been pressed many times.

De-bouncing mechanisms can be implemented:

1. By means of a software delay of around 30ms between two successive readings of the switch (to let the bouncing die down), whilst it is being polled. If the switch readings agree, then the switch is really on.
 - Wait while the switch is off
 - Wait 30ms
 - Exit if switch is off (return a '0'), else
 - Wait while the switch is on
 - Switch can now be taken as pressed (off-on-off) and return a '1'
2. Another software technique is to connect the switch to an interrupt pin instead of polling it routinely. It would be easier if a normally-high output from the switch is used and connected to the external interrupt, negative-edge triggered mode. As soon as the switch is pressed, we would have a high-to-low transition which would trigger an external interrupt. The ISR is called where we would immediately disable the external interrupt (otherwise we would have lots of them due to bouncing), wait for 30ms and then read the switch again. If we still read an ON condition, then we have detected a valid switch-on event and proceed accordingly. We can then enable the interrupts again and exit the ISR once finished with the response required. If the second reading shows an OFF condition, then we can take it as a glitch (or still bouncing) and that no switch has been pressed and once again we enable the interrupts again and exit the ISR without taking further action. If the bouncing is still going on, we would detect another interrupt and automatically repeat the ISR again.
3. By hardware, usually using a one shot device, which means that as soon as the switches flickers to the on position, the output of the one-shot will remain steadily on and bouncing is thus eliminated.

11.4 Keypad

Multiple switches (or keypads and keyboards) are normally connected in the form of a matrix where the vertical lines (columns) and horizontal lines (rows) are connected to the controller ports (either directly or via pull-up resistors) as shown in Figure 11-10. The port connections can be programmed to act as either input or output lines as required in order to be able to decide which key, if any, has been pressed.

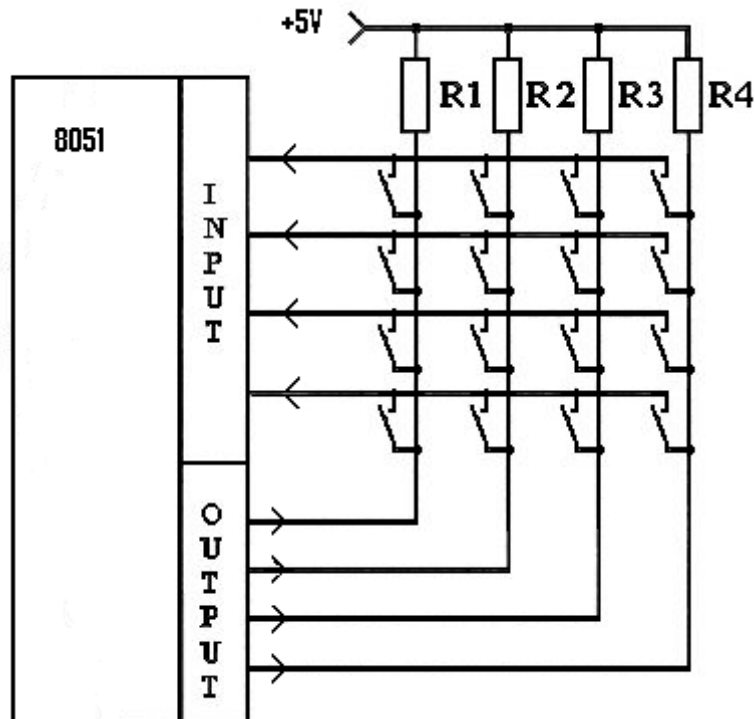


Figure 11-10 Keypad switch matrix

The method to detect a key press is as follows:

1. We set all output columns bits to 0.
2. The input row pins are then read.
3. If any row pin is a zero, then we know that a key in that row is being pressed, although we cannot tell yet which one of the four it is. If the input is not zero, we just have to wait and keep on reading the input port, waiting for a key press (going back to step 2).
4. If in the input row reading we do indeed detect a zero, then usually a bouncing delay is initiated so as to eliminate any bouncing or erroneous key contact (unless the bouncing is being taken care of by other hardware devices).
5. We read once again the input after this delay, and if the same row is giving a zero then we can start the process to determine exactly which column switch in that row is being pressed (the correct row is now known). If we do not detect a zero in any row, then we take it that it was a glitch and go back to step 2, waiting for a key press.

6. We can determine which key is being pressed by setting the input to zero for one column at a time and reading the row state until we read a zero. When the correct column is determined, then we have effectively decoded the key press, since we had already determined the row in step 5.

11.4.1 Keypad: interrupts vs polling

Instead of using this algorithm, where we are effectively waiting (whilst reading the port input) for any key press, we can modify the circuit to that shown in Figure 11-11. Note that in this figure, the rows are the output bits (P1.0 to P1.3) of the port, while the higher nibble of the port (P1.4 to P1.7) act as the input to read the column values.

All the rows are first set to zero and the external INT0 interrupt is enabled. The column input signals are ANDed together to provide an external INT0 interrupt low logic signal whenever any column goes low (negative edge triggered, activated when the signal goes from high to low). The INT0 interrupt service routine (ISR) would then be activated so that we can determine which key is being pressed.

www.sylvania.com

We do not reinvent the wheel we reinvent light.

Fascinating lighting offers an infinite spectrum of possibilities: Innovative technologies and new markets provide both opportunities and challenges. An environment in which your expertise is in high demand. Enjoy the supportive working atmosphere within our global group and benefit from international career paths. Implement sustainable ideas in close cooperation with other specialists and contribute to influencing our future. Come and join us in reinventing light every day.

Light is OSRAM

OSRAM SYLVANIA



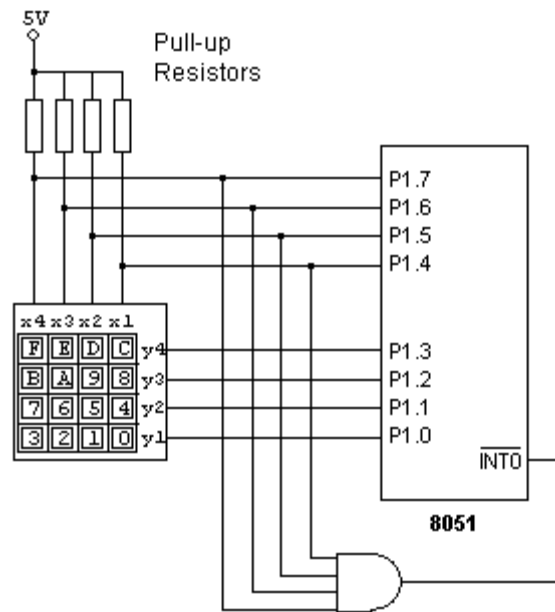


Figure 11-11 Interrupt keypad interface

We set all output row bits to 0, and enable the external negative-edge INT0 interrupt in the main program. We cannot obviously keep on looping and waiting within the ISR itself so the algorithm is modified as described below.

The ISR, activated whenever there is a key press would then perform the following:

1. The external interrupt is disabled. This is especially important in this case, since the bouncing effect of a switch would otherwise cause repeated interrupts.
2. A de-bounce delay (typically 30 ms) is initiated so as to wait for any bouncing or erroneous key contact to die down.
3. The input column pins are then read.
4. If any column pin is a zero, then we know that a key in that column is being pressed, although we cannot tell yet which one of the four it is. If we do not detect a zero on any input line, then the interrupt was probably caused by some glitch or intermittent key contact and we jump immediately to step 7 to exit the ISR.
5. If in the input column reading we do indeed detect a zero, then we can start the process to determine exactly which row switch in that column is being pressed (the correct column is now known from the input data pattern).
6. We can determine which actual key is being pressed by setting the input to zero for one row at a time and reading the column state until we read a zero. When the correct row is determined, then we have effectively decoded the key press, since we had already determined the column in step 5.
7. Enable once again the external INT0 interrupt, and exit the ISR.

10.5 LCD Display

A Liquid Crystal Display (LCD) provides a versatile output screen where normal text and graphics can be displayed, thus providing more versatility than the simple LED devices mentioned above. LCD displays come in many different versions, but here we shall deal with the cheap and simple 2 or 4 line display, providing 16 or 20 characters per line capability. It can be programmed to run either in the 8-bit data or in the 4-bit data mode if we do not have the luxury of using an 8-bit port dedicated to supply just the data bits to the LCD.

Figure 11-12 shows how we can connect a standard LCD (such as the Hitachi HD44780) to an 8051 microcontroller. Apart from the ground, supply, back lighting and contrast pins, we would need 8 data bits (D0–D7) in 8-bit mode or just 4 data bits (D4–D7) in the 4-bit mode so that we can communicate with the LCD. There are also 3 additional control signals RS, R/W and E (or EN) which we need to connect to the 8051 to provide the required hand-shaking control signals.

- RS is the register select signal, so that the LCD would know whether we are sending data to be displayed or sending a command intended to give some instructions to the LCD.
- R/W, as the name implies is the Read or Write signal which determines the direction of the data flow (reading from the LCD or writing to the LCD).
- E (or EN) is the enable pin, which has to be toggled so that any data is latched on to the device.



360°
thinking.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.



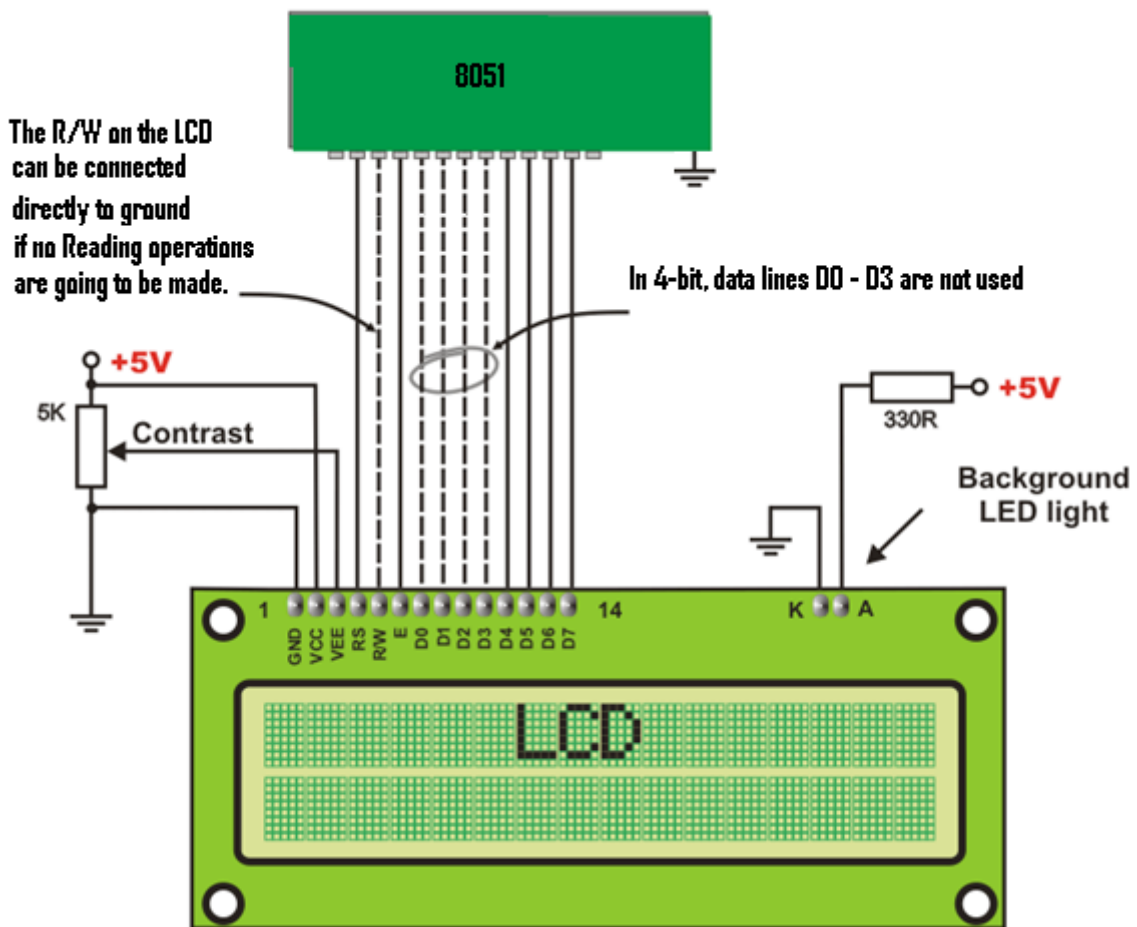


Figure 11-12 Standard LCD connections

The Read capability is mainly used to read the status of the LCD so that we can make sure that the LCD is ready to receive the next data or instruction. This is because the LCD takes some time to perform the required instructions, and not all instructions take the same amount of time to be executed. Hence the need to read the status of the LCD and wait for the LCD ready signal before proceeding. In many applications, we may only be required to write to the LCD, without the need to read anything. In this case we may simply initiate a fixed delay between issuing commands or data transfers, so as to be sure that the LCD has finished from the previous command, without the need to check the LCD status. Since the Write pin is active low, we can connect this pin permanently to ground in such cases. So, if we are only writing to the LCD and if we are using the 4-bit mode, we would then need only 6 bits (4-bits data, the EN and RS control signals) to communicate with the LCD. The LCD ground line naturally has to be common with the 8051 ground.

11.5.1 Programming the HD44780

In order to write a command or data, the following sequence of commands needs to be made, depending on the mode of operation of the LCD:

| 8-Bit Write Sequence |
|---|
| Make Sure "EN" is 0 or low |
| Set "R/S" to 0 for a command, or 1 for data/characters |
| Put the data/command on D7-0 |
| Set "EN" (EN= 1 or High) |
| Wait At Least 450 ns!!! |
| Clear "EN" (EN= 0 or Low) |
| Wait 5ms for command writes, and 200us for data writes. |

Table 11-2 LCD 8-bit write sequence

| 4-Bit Write Sequence |
|---|
| Make Sure "EN" is 0 or low |
| Set "R/S" to 0 for a command, or 1 for data/characters |
| Put the HIGH BYTE of the data/command on D7-4 |
| Set "EN" (EN= 1 or High) |
| Wait At Least 450 ns!!! |
| Clear "EN" (EN= 0 or Low) |
| Wait 5ms for command writes, and 200us for data writes. |
| Put the LOW BYTE of the data/command on D7-4 |
| Wait At Least 450 ns!!! |
| Clear "EN" (EN= 0 or Low) |
| Wait 5ms for command writes, and 200us for data writes. |

Table 11-3 LCD 4-bit write sequence

11.6 LCD Command Set

There are certain instructions or commands which we need to get familiar with in order to be able to program or setup the LCD display. The R/S and R/W control lines are also used depending on the type of the command required. Dedicated functions can be written which can take care of the initialisation and LCD mode setup as explain in the following sub-sections.

| R/S | R/W | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Instruction/Description |
|-----|-----|----|----|----|----|----|----|----|----|--|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Clear Display and Home the Cursor |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | * | Return Cursor and LCD to Home Position |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | ID | S | Set Cursor Move Direction |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B | Enable Display/Cursor |
| 0 | 0 | 0 | 0 | 0 | 1 | SC | RL | * | * | Move Cursor/Shift Display |
| 0 | 0 | 0 | 0 | 1 | DL | N | F | * | * | Set Interface Length |
| 0 | 0 | 0 | 1 | A | A | A | A | A | A | Move Cursor into CGRAM |
| 0 | 0 | 1 | A | A | A | A | A | A | A | Move Cursor to Display |
| 0 | 1 | BF | * | * | * | * | * | * | * | Poll the "Busy Flag" |
| 1 | 0 | D | D | D | D | D | D | D | D | Write a Character to the Display at the Current Cursor Position |
| 1 | 1 | D | D | D | D | D | D | D | D | Read the Character on the Display at the Current Cursor Position |

Table 11-4 LCD Command set

SIMPLY CLEVER

ŠKODA



We will turn your CV into an opportunity of a lifetime



Do you like cars? Would you like to be a part of a successful brand? We will appreciate and reward both your enthusiasm and talent. Send us your CV. You will be surprised where it can take you.

Send us your CV on www.employerforlife.com



The bit abbreviations used in Table 11-4 for the different commands are explained in the following list:

“*” - Not Used/Ignored. This bit can be either “1” or “0”

Set Cursor Move Direction:

ID - Increment the Cursor After Each Byte Written to Display if set
S - Shift Display when Byte Written to Display if set

Enable Display/Cursor

D - Turn Display On(1)/Off(0)
C - Turn Cursor On(1)/Off(0)
B - Cursor Blink On(1)/Off(0)

Move Cursor/Shift Display

SC - Display Shift On(1)/Off(0)
RL - Direction of Shift Right(1)/Left(0)

Set Interface Length

DL - Set Data Interface Length 8(1)/4(0)
N - Number of Display Lines 1(0)/2(1)
F - Character Font 5x10(1)/5x7(0)

Poll the “Busy Flag”

BF - This bit is set while the LCD is processing

Move Cursor to CGRAM/Display

A - Address

Read/Write ASCII to the Display

D - Data

We now provide some basic initialisation code for the 8-bit and for the 4-bit connection so that we can interface and communicate with this LCD.

| | General Initialisation | Example Initialisation |
|----------------------------------|---|---|
| 1 | Wait 20ms for LCD to power up | |
| 2 | Write D7-0 = 30 hex, with RS = 0 | |
| 3 | Wait 5ms | |
| 4 | Write D7-0 = 30 hex, with RS = 0, again | |
| 5 | Wait 200us | |
| 6 | Write D7-0 = 30 hex, with RS = 0, one more time | |
| 7 | Wait 200us | |
| 8 | Write Command “Set Interface” | Write 38 hex (8-Bits, 2-lines) |
| 9 | Write Command “Enable Display/Cursor” | Write 08 hex (don’t shift display, hide cursor) |
| 10 | Write Command “Clear and Home” | Write 01 hex (clear and home display) |
| 11 | Write Command “Set Cursor Move Direction” | Write 06 hex (move cursor right) |
| 12 | -- | Write 0C hex (turn on display) |
| Display is ready to accept data. | | |

Table 11-5 LCD 8-bit mode initialisation

11.6.1 The 8-bit mode LCD initialisation sample program

From the above tables, we can write some basic initialisation program for the LCD, starting with the 8-bit mode of operation. In this program we are making certain assumptions regarding the port pin connections to the LCD lines as can be seen from the initial remarks found in the code listing.

```

/* Assume that LCD-RS is connected to bit 0 of Port 2 (or LCD_CTRL_PORT)*/
/*          0 = Command, 1 = Data          */
/* Assume that LCD-RW is connected to bit 1 of Port 2 (or LCD_CTRL_PORT) */
/*          0 = Write, 1 = Read          */
/* Assume that LCD-EN is connected to bit 2 of Port 2 (or LCD_CTRL_PORT) */
/* A high (1) to low (0) transition is needed to latch data/command */
#define LCD_CTRL_PORT P2
sbit RSbit = LCD_CTRL_PORT^0;
sbit RWbit = LCD_CTRL_PORT^1;
sbit ENbit = LCD_CTRL_PORT^2;

/* If we only use the Control Port just for this purpose, we can send any one of the */
/* following defined items to set all three control lines simultaneously */
/* bit      2      1      0      */
#define ClearLines      0x00      /* EN = 0, RW = 0, RS = 0 */
#define LatchCommand1   0x04      /* EN = 1, RW = 0, RS = 0 */
#define LatchCommand2   0x00      /* EN = 0, RW = 0, RS = 0 */
#define LatchData1      0x05      /* EN = 1, RW = 0, RS = 1 */
#define LatchData2      0x01      /* EN = 0, RW = 0, RS = 1 */
#define ReadDataLines1  0x06      /* EN = 1, RW = 1, RS = 0 */
#define ReadDataLines2  0x02      /* EN = 0, RW = 1, RS = 0 */

/* Assume that the 8-bits data are connected to Port 1 (or LCD_DATA_PORT) */
#define LCD_DATA_PORT P1

/*****/
void LCD_SOFT_WAIT (int x)
{
    unsigned int i,j;
    for(j=1; j<=x; j++){
        for(i=0; i<=120; i++){ /* JUST A DELAY */
        }
    }
}

```



```
/******  
void LCD_SHORT_WAIT (void)  
{  
    unsigned char i;  
    i++;  
    i++;  
}  
  
/******  
  
/* This Wait If Busy routine can be used ONLY after the initialisation */  
void LCD_WAIT_IF_BUSY()  
{  
    unsigned char Status;  
    LCD_DATA_PORT = 0xFF;          /* set DATA port to input mode */  
    do  
    {  
        RWbit = 1; RSbit = 0; ENbit = 1;    /* set reading mode */  
        /* or LCD_CTRL_PORT = ReadDataLine2; */  
        LCD_SHORT_WAIT();  
        ENbit = 0;    /* or LCD_CTRL_PORT = ReadDataLine1; */  
        Status = LCD_DATA_PORT;  
    } while ((Status & 0x80) == 0x80);  
  
    ENbit = 1;  
    LCD_DATA_PORT = 0x00;          /* set DATA port to output mode */  
}  
  
/******  
/******  
  
void LCD_SEND_INIT(char ch) /* send display init to lcd */  
{  
    LCD_DATA_PORT = ch;  
    ENbit = 1; RWbit = 0; RSbit = 0;    // command sending mode  
    LCD_SHORT_WAIT();  
    ENbit = 0;  
    LCD_SOFT_WAIT(20);    /* wait for at least 5 milliseconds */  
    ENbit = 1;  
    /* cannot check busy line yet, not until the initialisation has finished */  
}
```

```

/*****/

void LCD_Send_Command(char ch)    /* write display command to lcd */
{
    LCD_WAIT_IF_BUSY();
    LCD_DATA_PORT = ch;
    ENbit = 1; RWbit = 0; RSbit = 0;    // command sending mode
    LCD_SHORT_WAIT();
    ENbit = 0;
}    /* end lcd write */

/*****/

void LCD_Send_Data(char ch)        /* write display data to lcd */
{
    LCD_WAIT_IF_BUSY();
    LCD_DATA_PORT = ch;
    ENbit = 1; RWbit = 0; RSbit = 1;    // data sending mode
    LCD_SHORT_WAIT();
    ENbit = 0;
}    /* end lcd write */

/*****/

/* 8-bit mode */
void LCD_INIT(void)    /* reset lcd display */
{
    LCD_CTRL_PORT = LCD_DATA_PORT = 0;    /* set both 8251 ports as output */
    LCD_SOFT_WAIT(50);    /* wait a few milliseconds, after power on */
    ENbit = 0; RWbit = 0; RSbit = 0;    // clear control lines
    LCD_SEND_INIT(0x38);    /* get attention */
    LCD_SEND_INIT(0x38);    /* set mode to 8 bit DATA 2 lines, 5x7 dots */
    LCD_SEND_COMMAND(0x0C);    /* Display On, Cursor Off and Blinking off */
    LCD_SEND_COMMAND(0x01);    /* Clear Display */
    LCD_SEND_COMMAND(0x06);    /* Set Entry Mode */
}    /* end of lcd initialisation */

```

11.6.2 4-bit mode LCD Initialisation

We have to remember that in this 4-bit mode any Data/Command writes of one-byte size are handled using:

send high-nibble, delay, send low-nibble, delay

sequence, where 1 nibble is equivalent to 4 bits.

Download free eBooks at bookboon.com

| | General Initialisation | Example Initialisation |
|----------------------------------|--|---|
| 1 | Wait 20ms for LCD to power up | |
| 2 | Write D7-4 = 3 hex, with RS = 0 | |
| 3 | Wait 5ms | |
| 4 | Write D7-4 = 3 hex, with RS = 0, again | |
| 5 | Wait 200us | |
| 6 | Write D7-4 = 3 hex, with RS = 0, one more time | |
| 7 | Wait 200us | |
| 8 | Write D7-4 = 2 hex, to enable four-bit mode | |
| 9 | Wait 5ms | |
| 10 | Write Command "Set Interface" | Write 28 hex (4-Bits, 2-lines) |
| 11 | Write Command "Enable Display/Cursor" | Write 08 hex (don't shift display, hide cursor) |
| 12 | Write Command "Clear and Home" | Write 01 hex (clear and home display) |
| 13 | Write Command "Set Cursor Move Direction" | Write 06 hex (move cursor right) |
| 14 | -- | Write 0C hex (turn on display) |
| Display is ready to accept data. | | |

Table 11-6 LCD 4-bit mode initialisation

11.6.3 The 4-bit mode LCD initialisation sample program

Here we assume that the control signals are connected to the lower 3 bits (RS to bit 0, RW to bit 1 and EN to bit 2), while the 4 data lines (D4–D7) are connected to the upper four bits of the port. D4 to port bit 4, D5 to port bit 5 and so on.

```
#define LCD_PORT    P2
sbit RSbit = LCD_PORT^0;
sbit RWbit = LCD_PORT^1;
sbit ENbit = LCD_PORT^2;
#define    LCD_EN    0x04
#define    LCD_RW    0x02
#define    LCD_RS    0x01

// The 4 data lines (D4–D7) are connected to the upper four bits of the port.
// D4 to port bit 4, D5 to port bit 5 and so on.
```

```
void LCD_Wait_If_Busy (void)      /* wait for lcd if busy */
{
// The Busy Flag is the most significant bit of the received data
char c,d;
LCD_PORT = 0xF0;                // set port upper nibble to input mode
do {
ENbit = 1;
RWbit = 1;                      // prepare for a Write operation
lcd_soft_wait(5);
c = (LCD_PORT & 0xF0);         /* read high data nibble */
ENbit = 0;
RWbit = 0;
lcd_soft_wait(5);
ENbit = 1;
RWbit = 1;                      // prepare for a Write operation
lcd_soft_wait(5);
d = ( LCD_PORT & 0xF0);        /* read low data nibble, in Port.4 – Port.7 bits */
ENbit = 0;
RWbit = 0;
d = d>>4;                       /* move it to the lower nibble
c = c + d;                       /* combine nibbles to form 8-bit data */
} while (c & 0x80);            /* wait for Busy Flag (BF) line to go low */
LCD_PORT = 0x00;                // set all port pins to output mode again
}                               /*end lcd busy wait */

void LCD_Send_Data(char ch)      /* write display data to lcd */
{
LCD_Wait_If_Busy();
LCD_PORT = ((ch & 0xf0) | LCD_EN | LCD_RS);    /* send character high nibble */
ENbit = 0;
lcd_soft_wait(3);
LCD_PORT = (((ch & 0x0f) << 4) | LCD_EN | LCD_RS); /* send character high nibble */
ENbit = 0;
}                               /* end lcd data write */

void LCD_Send_Command(char ch)  /* write display command to lcd */
```

```

{
LCD_Wait_If_Busy();
LCD_PORT = ((ch & 0xf0) | LCD_EN);          /* send character high nibble */
ENbit = 0;
lcd_soft_wait(3);
LCD_PORT = (((ch & 0x0f) << 4) | LCD_EN); /* send character low nibble */
ENbit = 0;
}          /* end lcd write command function */
void LCD_Send_Init_Command(char ch)          /* write display initialization commands to lcd
*/
// Cannot use LCD_Wait_If_Busy routine yet.
{
LCD_PORT = ((ch & 0xf0) | LCD_EN);          /* send character high nibble */
ENbit = 0;
lcd_soft_wait(5);
LCD_PORT = (((ch & 0x0f) << 4) | LCD_EN);    /* send character low nibble */
ENbit = 0;
lcd_soft_wait(5);
}          /* end lcd write command function */

void LCD_Init_4(void)          /* reset lcd display */
{
lcd_soft_wait(10);          /* wait at least 15ms after power on*/
LCD_Send_Init_Command (0x33);    /* get attention */
lcd_soft_wait(5);          /* wait */
LCD_Send_Init_Command (0x32);    /* get attention */
lcd_soft_wait(10);          /* wait */
LCD_Send_Init_Command (0x20);    /* 4 bit DATA transfer from now on */
lcd_soft_wait(5);
// LCD_Send_Init_Command (0x28); /* 4 bit data, 2 lines, 5x7 dots */
LCD_Send_Init_Command (0x2C);    /* 4 bit data, 2 lines, 5x10 dots */
LCD_Send_Command (0x06);        /* Move Cursor to the right. Do not shift display */
LCD_Send_Command (0x0C);        /* Display On, Cursor and Blinking off */
// LCD_Send_Command (0x08);      /* Display, Cursor and Blinking Off */
// LCD_Send_Command (0x0F);      /* Display, Cursor and Blinking on */
LCD_Send_Command (0x01);        /* Clear Display */
}          /* end lcd initialize */

```

In any mode, in order to write a text string to the LCD, instead of writing a letter at a time we can write a routine. In the sample program below we are assuming that the length of the text fits into the LCD display.

```
void LCD_Write_String (char *s)
{
    while (*s) { /* Write all characters within string, checking for the end of string char /0 */
        LCD_Send_Data(*s++); /* Send character to LCD display */
    }
}
```

Similarly we can then write various other routines so that we can centre our text, write at any row or column position, display a moving text and so on.

11.7 DC Motor

A simple DC motor can be connected to the 8051 as shown in Figure 11-13. Since the motor takes some appreciable amount of current, especially when switching on, we cannot drive it directly through the port. We normally use a transistor such as the BD139 (or mechanical relay) to switch it on and off, as shown in Figure 11-13. The type of the transistor used depends on the motor specification, mainly the current that it takes. Since this current would all be passing through the transistor, Q1 must be able to handle the power without overheating. A heat-sink is also used in most case to keep the temperature of the transistor within limits. The diode across the motor is needed in order to provide a path for the back emf generated by the motor itself.

I joined MITAS because
I wanted **real responsibility**

The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com

Month 16
I was a construction
supervisor in
the North Sea
advising and
helping foremen
solve problems

Real work
International opportunities
Three work placements

MAERSK



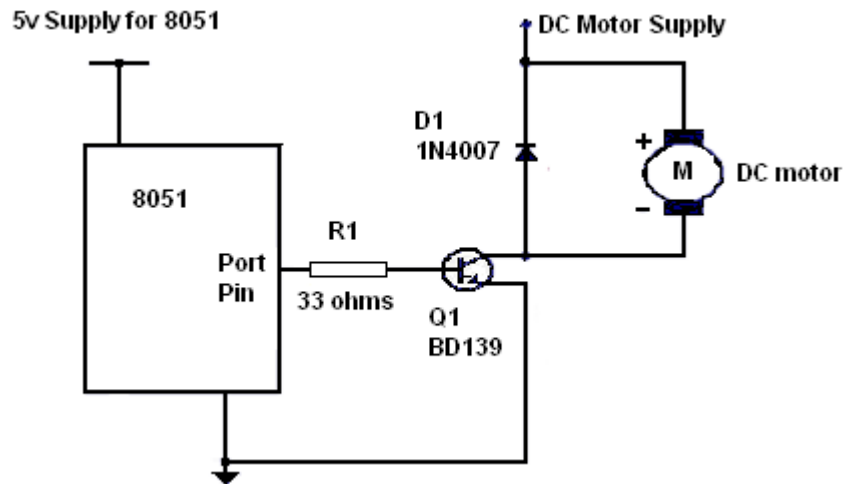


Figure 11-13 DC Motor interfacing

The supply for the dc motor is normally a separate supply which can handle the power requirements of the motor and moreover reduces the glitches on the 8051 supply rail.

Apart from just switching it ON (running at maximum speed) when we need the motor and then switching OFF when we are done with it, we can also make it run at variable speeds by switching it ON and OFF with a pulse train (or Pulse Width Modulation [PWM] signal), varying the ON pulse width relative to the OFF time. The inertia of the motor armature and whatever it is driving, will keep the motor turning even during the OFF cycle. The greater the ON time, the faster it goes, since the average voltage of the signal would be higher, as shown in Figure 11-14.

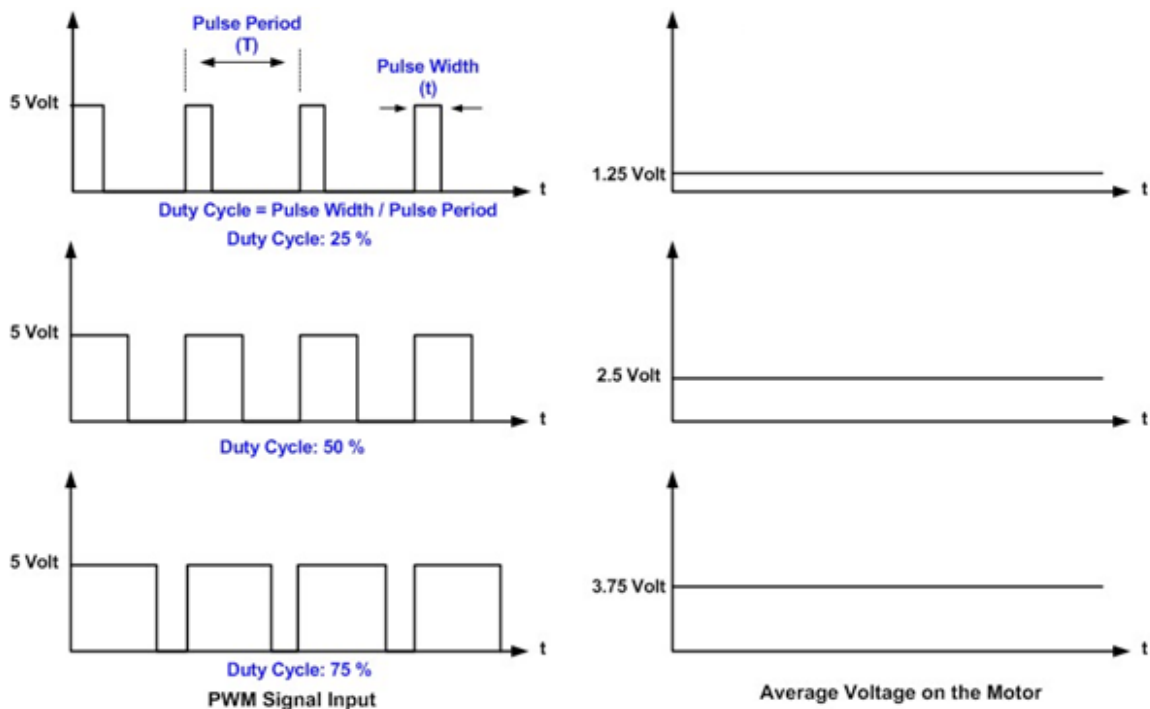


Figure 11-14 PWM used to control DC motor speed

Having a very low mark (1 or ON) to space (0 or OFF) ratio could result in the motor not turning at all. It depends a lot on the type of motor and how free is the armature to rotate. So we can expect that the mark-to-space ratio would need to be above 30% for the motor to start turning and overcome friction etc.

An example which can be adapted to this setup is given in section 11.8 when discussing the H-bridge connection. The principle of using PWM to adjust and control the motor speed is still the same.

11.8 DC motor using H-Bridge

If we add an H-bridge to our circuit, we can now also change the direction of rotation of the motor, apart from controlling its speed. The H-bridge operation can be best explained with reference to the following figures which describe the operation of the dc motor. The switches shown would actually be transistor switches and they could be switched ON and OFF by means of signals coming out of the 8051 port.

Figure 11-15 shows the motor in the OFF position, where all the switches are off.

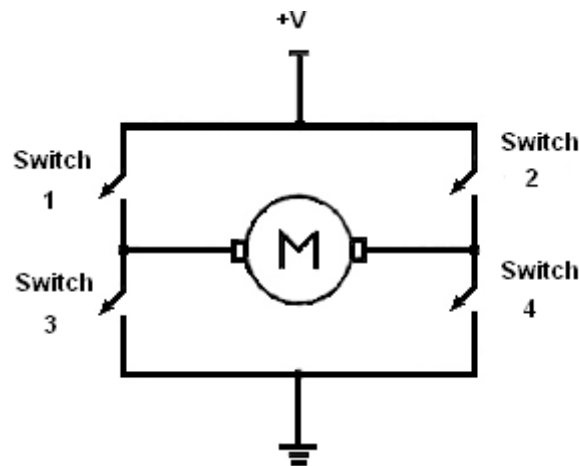


Figure 11-15 Motor Off

If now switches 1 and 4 are switched ON, leaving the others off, the motor would turn at full speed in one direction say clockwise, as shown in Figure 11-16.

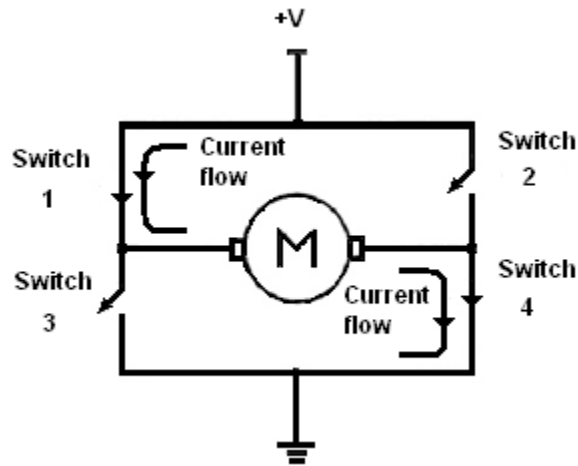


Figure 11-16 Motor Clockwise Rotation

On the other hand, if we switch ON 2 and 3, and leaving switches 1 and 4 OFF as shown in Figure 11-17 the motor would turn at full speed in the opposite direction, since the supply would now be inverted with respect to the motor terminals.

ie business school

#1 EUROPEAN BUSINESS SCHOOL
FINANCIAL TIMES 2013

#gobeyond

MASTER IN MANAGEMENT

Because achieving your dreams is your greatest challenge. IE Business School's Master in Management taught in English, Spanish or bilingually, trains young high performance professionals at the beginning of their career through an innovative and stimulating program that will help them reach their full potential.

- Choose your area of specialization.
- Customize your master through the different options offered.
- Global Immersion Weeks in locations such as London, Silicon Valley or Shanghai.

Because you change, we change with you.

www.ie.edu/master-management | mim.admissions@ie.edu | f t in YouTube



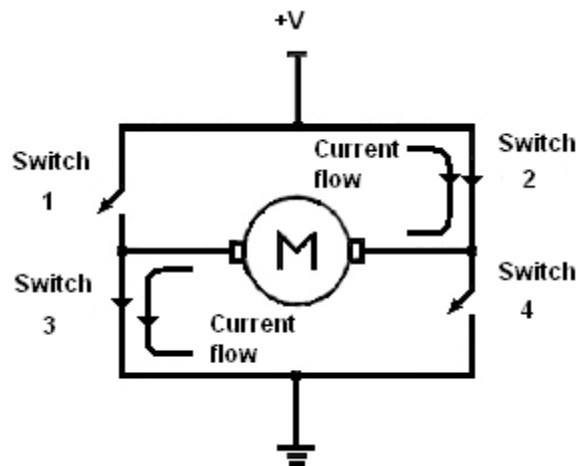


Figure 11-17 Motor Anti-Clockwise Rotation

Thus we can see that the switches normally operate in pairs since switches 1 and 4 switch on and off together and the same thing with switches 2 and 3. We would therefore require two signals which moreover are always of the opposite logic with respect to each other (one is the complement of the other). Hence theoretically we could do with one signal and its complement (which we can obtain by using an inverter). However, the need to avoid having all the switches ON at the same time (which can happen during the transition due to the propagation delay), it would be best if we use two separate signals (S1 and S2) to control the two separate pairs of switches as shown in Figure 11-18, making sure that we switch off one pair before switching on the other pair.

Once again, if instead of switching these sets of switches permanently ON, we supply them with a PWM signal, we now have the capability to control BOTH the speed and the direction of the DC motor.

A typical H-bridge, using discrete components is shown in Figure 11-18, with the transistors acting as the switches, being driven from the 8051 ports. We have to ensure by means of our software program not to have both transistors on either side of the motor ON at the same time, otherwise we would be short-circuiting the motor supply.

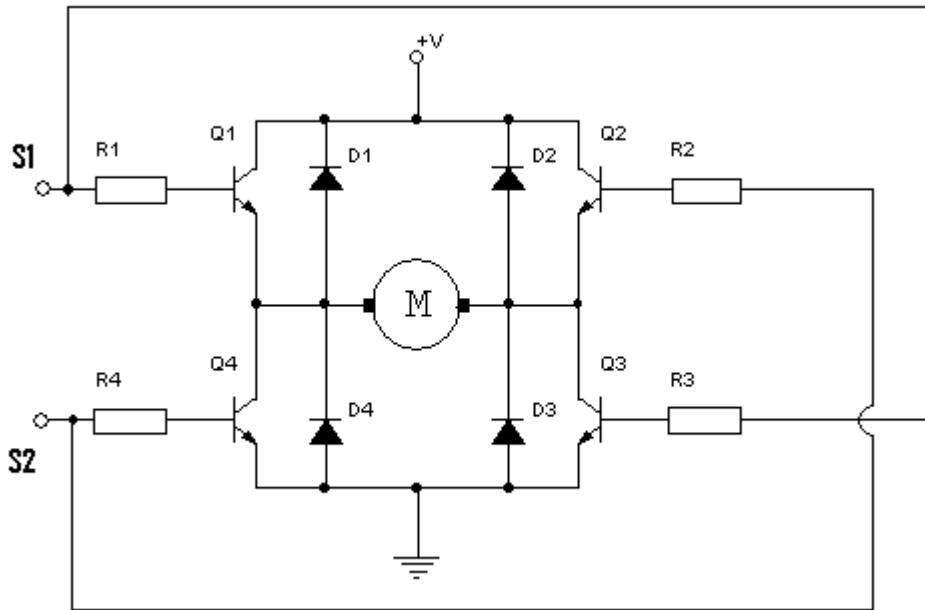


Figure 11-18 H-Bridge circuit with discrete devices

Thus, before switching from one set of transistors to the other set in order to change the direction, we must make sure to switch off ALL the transistors first. The algorithm to control the speed and direction is very simple and we describe it briefly here with the source code for a routine which controls the circuit shown in Figure 11-18. Speed can take a value between 0 and 100 representing zero (off) to 100% full speed. Direction can be either C (clockwise) or A (anti-clockwise). Duration would be the length of time in milliseconds that the motor has to be in that state.

We are assuming that we have a timer routine called `ms_delay(unsigned long delay)` which would wait for the specified amount of milliseconds.

```
// H Bridge
#include <reg51.h>
sbit S1 = P1^0;
sbit S2 = P1^1;

void ms_delay(unsigned long);

// The following routine controls the motor, setting it at the required
// direction and speed for the specified time duration.
// The speed, although theoretically has the range 0–100, might need a value
// greater than 30 for the motor to actually start turning and overcome friction etc.
// The PWM signal has a periodic time of 100ms.
// Motor always exits the routine in the OFF condition.
```

```
void MotorControl(char Direction, unsigned char Speed, unsigned long Duration) {
    unsigned long milliSeconds;

    milliSeconds = 0;

    if (Speed == 0) {
        S1 = S2 = 0;
        ms_delay(Duration);
    }
        // switch off motor completely

    else if ((Speed == 100) && ((Direction == 'A') || (Direction == 'a'))){
        S1 = 0;
        S2 = 1;
        ms_delay(Duration);
        // full speed anti-clockwise, no PWM required
        S1 = S2 = 0;
    }

    else if ((Speed == 100) && ((Direction == 'C') || (Direction == 'c'))){
        S1 = 1;
        S2 = 0;
        ms_delay(Duration);
        // full speed clockwise, no PWM required
        S1 = S2 = 0;
    }

    else{
        // 0 < speed < 100 hence PWM is required
        milliSeconds = 0;
        // used for timing the duration of the PWM
        while(milliSeconds < Duration)
        {
            // first switch off one pair of transistors, then turn on the other pair of transistors
            // to avoid shorting the power supply
            if ((Direction == 'A') || (Direction == 'a')) {S1 = 0; S2 = 1;}
            if ((Direction == 'C') || (Direction == 'c')) {S2 = 0; S1 = 1}
            ms_delay((unsigned long)Speed);
            S1 = 0; S2 = 0;
            ms_delay((unsigned long)(100 - Speed));
            milliSeconds += 100UL; // add one PWM period to check duration
        }
    }
}
```

```
void ms_delay(unsigned long delay_ms) {
// Assuming clock is 11.0592 MHz, then 921 timer counts
// would take approximately 1 millisecond
// Hence timer registers will be loaded with (65536 - 921)
// i.e. 64615, so that it will overflow after 1 millisecond
// TH0 = 64615/256 = 252
// TL0 = 64615%256 = 103

    TMOD &= 0xF0;
    TMOD |= 0x01;          // set Timer 0 in 16-bit mode 1
    ET0 = 0;              // disable Timer 0 interrupts just in case
    while (delay_ms > 0) {
        TH0 = 252;
        TL0 = 103;        // load Timer 0 registers for 1 millisecond delay
        TF0 = 0;          // clear Timer 0 overflow flag
        TR0 = 1;          // start Timer 0
        while (!TF0);     // wait for Timer 0 overflow
        delay_ms--;       // decrement 1 millisecond
        TR0 = 0;          // stop Timer 0
    }
    TF0 = 0;              // Reset flag before exit
}

void main(void) {
    S1 = S2 = 0;          // start with motor off

    MotorControl('A',1000UL); // motor stopped for 1 second
    MotorControl('C',90,4000UL); // motor clockwise at 90%, for 4 seconds
    MotorControl('A',50,3000UL); // motor anti-clockwise at 50%, for 3 seconds
    MotorControl('C',10,2000UL); // motor clockwise at 10%, for 2 seconds
    MotorControl('A',100,2500UL); // motor anti-clockwise at 100%, for 2.5 seconds

    while(1);            // stay here when finished
}
```

The H-bridge is so much in use that special ICs from a wide range of manufacturers have been designed. Shown in Figure 11-19 is a typical IC, the L292D which has the capability to drive 2 dc motors separately. Datasheets for this and similar devices are readily available on the internet, which fully describe the operation complete with examples.

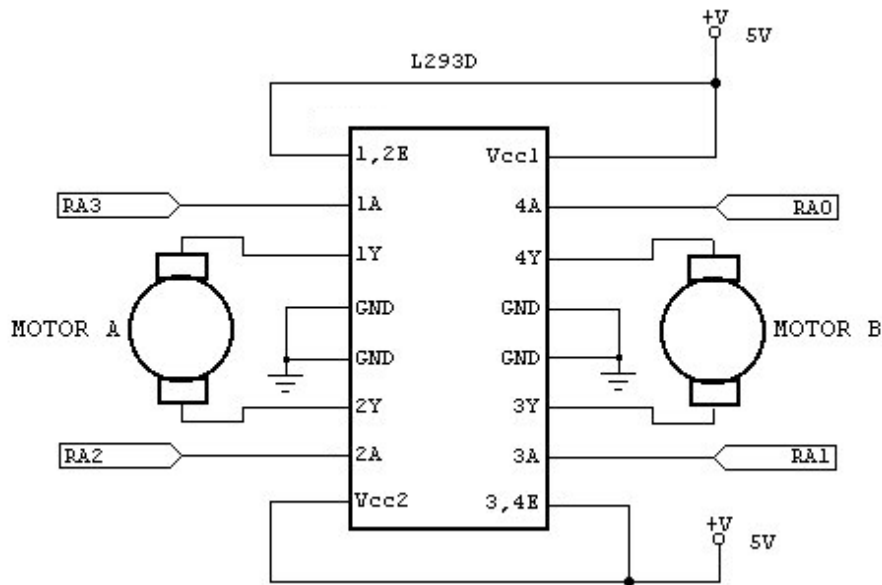


Figure 11-19 L293D H-bridge connection

"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"
Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download

11.9 Model Servo Control

Radio Controlled (RC) model servo motors, of the type shown in Figure 11-20 can also be very easily controlled using the 8051. They are widely used in RC aero models and miniature robotics. These types of motors require a PWM signal very similar to the one explained above in sections 11.7 and 11.8. We need to have a PWM period of 20ms and we need to vary the ON time in the range of 1 to 2ms. A 1ms pulse would result in a full right movement say while a 2ms ON pulse would turn the servo arm to the full left position (a 1.5ms ON pulse would place the servo arm in the centre or neutral position).



Figure 11-20 RC Servo (www.parallaxinc.com)

Just three connections are needed as shown in Figure 11-21, two for the supply (usually around 5V, red is positive and black is ground) and the third wire (usually white or yellow) is where the PWM signal is fed from the micro-controller port pin. We should always remember to connect the ground of the servo to the ground of the micro-controller, since we would normally be feeding the servo from a separate higher capacity supply source.

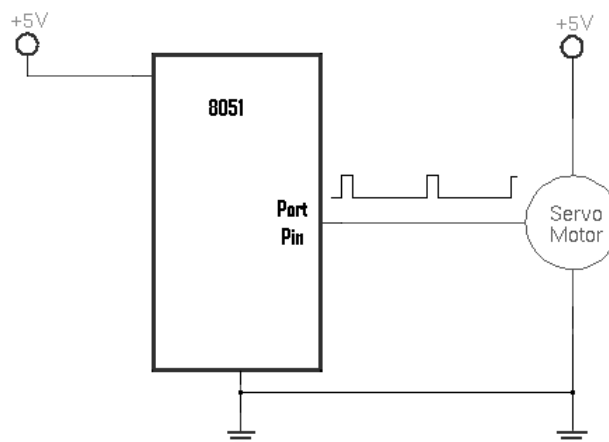


Figure 11-21 RC Servo connection

Servos like all other motors, consume a lot of power especially when under load and it therefore would make sense to use a separate power supply just for the servo motor which would also reduce the interference on the 8051 supply lines.

We can also find servos which are slightly modified so that instead of just turning plus or minus 90 degrees, they are able to turn continuously. For example, a 1ms pulse would cause the servo to turn continuously clockwise and a 2ms pulse would turn it continuously anti-clockwise. In order to stop the servo, we would need to feed it with a 1.5ms pulse train, still using 20ms PWM periodicity.

11.10 Stepper Motor

The stepper motor (see Figure 11-22) is one of the commonly used motors for precise angular movement. The advantage of using a stepper motor is that the angular position of the motor shaft can be controlled without the need of any feedback mechanism. They are widely used in industrial and commercial applications as well as in drive systems of autonomous robots.



Figure 11-22 Typical Stepper Motors

They are commonly found in dot-matrix or ink-jet printers to drive the printing head and feed forward the paper. By switching on the appropriate coils (see Figure 11-23), we can make the armature to rotate to and then stop at a specified rotation angle, so as it would align with the stator magnetic field. Moreover, if the whole 360 degree sequence is continuously repeated, the stepper motor can be made to turn at the required speed and in the required direction. The program would just have to determine which coils are to be energised and for how long.

Various ICs are available to drive these stepper motors and the L297 (or similar) stepper motor controller IC in conjunction with the L298 (or similar) dual H-bridge IC can be used.

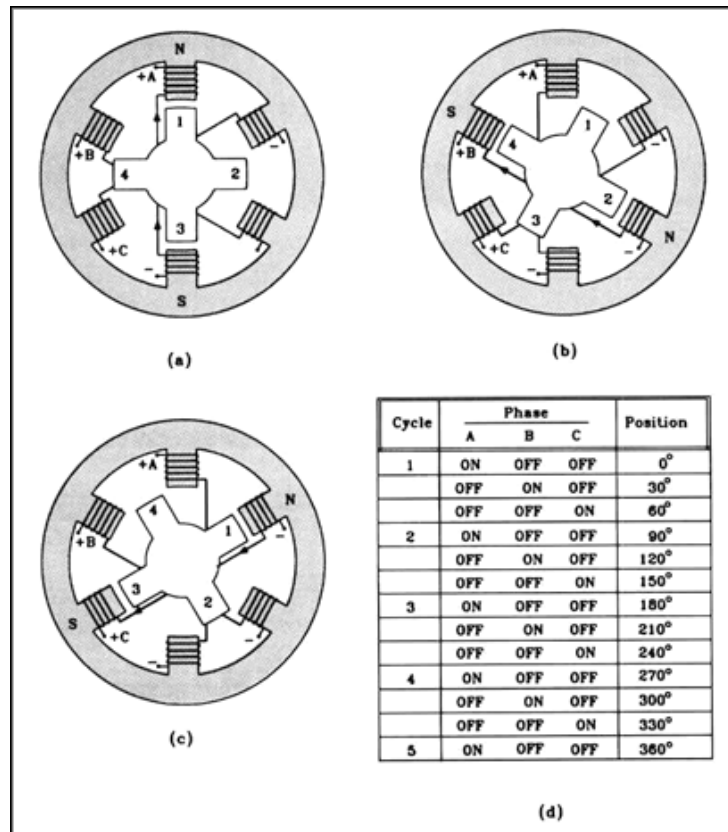


Figure 11-23 Stepper Motor sequence (zone.ni.com)

Excellent Economics and Business programmes at:



university of
 groningen




“The perfect start of a successful, international career.”

CLICK HERE
to discover why both socially and academically the University of Groningen is one of the best places for a student to be

www.rug.nl/feb/education

