SLOT MACHINE

Computer Hardware Project

by

Sarper Önal

A project
presented to the Athlone Institute of Technology
in fulfilment of the
requirement for the final year of
Computer Engineering

Teacher: Kevin McDermott
Student: Sarper Önal
No: A00141571

Table of Contents

Chapter 1: Introduction....3
What is Slot Machine....3
What is Microcontroller....3
Programming environments....4
Embedded design....4
Interrupts....5
Programs....5
Other microcontroller features....5
Chapter 2: Background Research....6
LCD Background....6
HANDLING THE EN CONTROL LINE....7
INITIALIZING THE LCD....9
CLEARING THE DISPLAY....10
Writing a text to the LCD....11
A "HELLO WORLD" PROGRAM....12
CURSOR POSITIONING....12
Code Background....15
Buttons Background: 17
Chapter 3: Hardware Design....18
Chapter 4: Software Design....19
Chapter 5: Future Development....20
Chapter 6: Testing Evaluation....20
Code....21

Chapter 1: Introduction:

What is Slot Machine?

Slot machines are a game of chance where the player inserts money into the machine, presses the spin button and watches the reels spin. Instead of using a reels spin I made it with some electronic components, such as LCD button and 8051 board. The objective of the game is to get a winning combination on the pay line- this will normally be a line across the middle of the reels. If you manage to get a winning combination on the slot-machine, you will be paid according to the slot-machines pay table.

Slot Machine is the semester two project focused on the random pictures,This simple machine has 4 buttons, one Screen and 8051 board,There are 3 random shapes you have to get 3 of them same by pressing buttons,If you get 3 of them same, you win!,If you couldn't get 3 shape same, you lose!

What is Microcontroller?

A microcontroller (also MCU or µC) is a functional computer system-on-a-chip. It contains a processor core, memory, and programmable input/output peripherals. Microcontrollers include an integrated CPU, memory (a small amount of RAM, program memory, or both) and peripherals capable of input and output.

It emphasises high integration, in contrast to a microprocessor which only contains a CPU (the kind used in a PC). In addition to the usual arithmetic and logic elements of a general purpose microprocessor, the microcontroller integrates additional elements such as read-write memory for data storage, read-only memory for program storage, Flash memory for permanent data storage, peripherals, and input/output interfaces. At clock speeds of as little as 32KHz, microcontrollers often operate at very low speed compared to microprocessors, but this is adequate for typical applications. They consume relatively little power (milliwatts or even microwatts), and will generally have the ability to retain functionality while waiting for an event such as a button press or interrupt. Power consumption while sleeping (CPU clock and peripherals disabled) may be just nanowatts, making them ideal for low power and long lasting battery applications.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, remote controls, office machines, appliances, power tools, and toys. By reducing the size, cost, and power consumption compared to a design using a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to electronically control many more processes.

Programming environments

Microcontrollers were originally programmed only in assembly language, but various high-level programming languages are now also in common use to target microcontrollers. These languages are either designed specially for the purpose, or versions of general purpose languages such as the C programming language. Compilers for general purpose languages will typically have some restrictions as well as enhancements to better support the unique characteristics of microcontrollers. Some microcontrollers have environments to aid developing certain types of applications. Microcontroller vendors often make tools freely available to make it easier to adopt their hardware.

Many microcontrollers are so quirky that they effectively require their own non-standard dialects of C, such as SDCC for the 8051, which prevent using standard tools (such as code libraries or static analysis tools) even for code unrelated to hardware features. Interpreters are often used to hide such low level quirks.

Interpreter firmware is also available for some microcontrollers. For example, BASIC on the early microcontrollers Intel 8052; BASIC and FORTH on the Zilog Z8as well as some modern devices. Typically these interpreters support interactive programming.

Simulators are available for some microcontrollers, such as in Microchip's MPLAB environment. These allow a developer to analyse what the behaviour of the microcontroller and their program should be if they were using the actual part. A simulator will show the internal processor state and also that of the outputs, as well as allowing input signals to be generated. While on the one hand most simulators will be limited from being unable to simulate much other hardware in a system, they can exercise conditions that may otherwise be hard to reproduce at will in the physical implementation, and can be the quickest way to debug and analyse problems.

Recent microcontrollers are often integrated with on-chip debug circuitry that when accessed by an In-circuit emulator via JTAG, allow debugging of the firmware with a debugger.

Embedded design

The majority of computer systems in use today are embedded in other machinery, such as auto mobiles, telephones, appliances, and peripherals for computer systems. These are called embedded systems. While some embedded systems are very sophisticated, many have minimal requirements for memory and program length, with no operating system, and low software complexity. Typical input and output devices include switches, relays, solenoids, LED's, small or custom LCD displays, radio frequency devices, and sensors for data such as temperature, humidity, light level etc. Embedded systems

usually have no keyboard, screen, disks, printers, or other recognisable I/O devices of a personal computer, and may lack human interaction devices of any kind.

Interrupts

It is mandatory that microcontrollers provide real time response to events in the embedded system they are controlling. When certain events occur, an interrupt system can signal the processor to suspend processing the current instruction sequence and to begin an interrupt service routine (ISR). The ISR will perform any processing required based on the source of the interrupt before returning to the original instruction sequence. Possible interrupt sources are device dependent, and often include events such as an internal timer overflow, completing an analogue to digital conversion, a logic level change on an input such as from a button being pressed, and data received on a communication link. Where power consumption is important as in battery operated devices, interrupts may also wake a microcontroller from a low power sleep state where the processor is halted until required to do something by a peripheral event.

Programs

Microcontroller programs must fit in the available on-chip program memory, since it would be costly to provide a system with external, expandable, memory. Compilers and assembly language are used to turn high-level language programs into a compact machine code for storage in the microcontroller's memory. Depending on the device, the program memory may be permanent, read-only memory that can only be programmed at the factory, or program memory may be field-alterable flash or erasable read-only memory.

Other microcontroller features

Since embedded processors are usually used to control devices, they sometimes need to accept input from the device they are controlling. This is the purpose of the analogue to digital converter. Since processors are built to interpret and process digital data, i.e. 1s and 0s, they won't be able to do anything with the analog signals that may be being sent to it by a device. So the analogue to digital converter is used to convert the incoming data into a form that the processor can recognise. There is also a digital to analogue converter that allows the processor to send data to the device it is controlling.

In addition to the converters, many embedded microprocessors include a variety of timers as well. One of the most common types of timers is the Programmable Interval Timer, or PIT for short. A PIT just counts down from some value to zero. Once it reaches zero, it sends an interrupt to the processor indicating that it has finished counting. This is useful for devices such as thermostats, which periodically test the temperature around them to see if they need to turn the air conditioner on, the heater on, etc.

Time Processing Unit or TPU for short. Is essentially just another timer, but more

sophisticated. In addition to counting down, the TPU can detect input events, generate output events, and other useful operations.

Dedicated Pulse Width Modulation (PWM) block makes it possible for the CPU to control power converters, resistive loads, motors, etc., without using lots of CPU resources in tight timer loops.

Universal Asynchronous Receiver/Transmitter (UART) block makes it possible to receive and transmit data over a serial line with very little load on the CPU.

Chapter 2: Background Research :

LCD Background:

Frequently, an 8051 program must interact with the outside world using input and output devices that communicate directly with a human being. One of the most common devices attached to an 8051 is an LCD display. Some of the most common LCD's connected to the 8051 are 16x2 and 20x2 displays. This means 16 characters per line by 2 lines and 20 characters per line by 2 lines, respectively.

Fortunately, a very popular standard exists which allows us to communicate with the vast majority of LCD's regardless of their manufacturer. The standard is referred to as HD44780U, which refers to the controller chip which receives data from an external source (in this case, the 8051) and communicates directly with the LCD.

44780 BACKGROUND

The 44780 standard requires 3 control lines as well as either 4 or 8 I/O lines for the data bus. The user may select whether the LCD is to operate with a 4-bit data bus or an 8-bit data bus. If a 4-bit data bus is used the LCD will require a total of 7 data lines (3 control lines plus the 4 lines for the data bus). If an 8-bit data bus is used the LCD will require a total of 11 data lines (3 control lines plus the 8 lines for the data bus). Furthermore I didn't use any 44780

The three control lines are referred to as **EN**, **RS**, and **RW**.

The **EN** line is called "Enable." This control line is used to tell the LCD that you are sending it data. To send data to the LCD, your program should make sure this line is low (0) and then set the other two control lines and/or put data on the data bus. When the other lines are completely ready, bring **EN** high (1) and wait for the minimum amount of time required by the LCD data-sheet (this varies from LCD to LCD), and end by bringing it low (0) again.

The **RS** line is the "Register Select" line. When RS is low (0), the data is to be treated as a command or special instruction (such as clear screen, position cursor, etc.).

When RS is high (1), the data being sent is text data which should be displayed on the screen. For example, to display the letter "T" on the screen you would set RS high.

The **RW** line is the "Read/Write" control line. When RW is low (0), the information on the data bus is being written to the LCD. When RW is high (1), the program is effectively querying (or reading) the LCD. Only one instruction ("Get LCD status") is a read command. All others are write commands--so RW will almost always be low.

Finally, the data bus consists of 4 or 8 lines (depending on the mode of operation selected by the user). In the case of an 8-bit data bus, the lines are referred to as DB0, DB1, DB2, DB3, DB4, DB5, DB6, and DB7.

HANDLING THE EN CONTROL LINE

As we mentioned above, the EN line is used to tell the LCD that you are ready for it to execute an instruction that you've prepared on the data bus and on the other control lines. Note that the EN line must be raised/lowered before/after each instruction sent to the LCD regardless of whether that instruction is read or write, text or instruction. In short, you must always manipulate EN when communicating with the LCD. EN is the LCD's way of knowing that you are talking to it. If you don't raise/lower EN, the LCD doesn't know you're talking to it on the other lines.

Thus, before we interact in any way with the LCD we will always bring the **EN** line low with the following instruction:

CLR EN

And once we've finished setting up our instruction with the other control lines and data bus lines, we'll always bring this line high:

SETB EN

The line must be left high for the amount of time required by the LCD as specified in its data-sheet. This is normally on the order of about 250 nanoseconds, but check the data-sheet. In the case of a typical 8051 running at 12 MHz, an instruction requires 1.08 microseconds to execute so the EN line can be brought low the very next instruction. However, faster microcontrollers (such as the DS89C420 which executes an instruction in 90 nanoseconds given an 11.0592 MHz crystal) will require a number of NOPs to create a delay while EN is held high. The number of NOPs that must be inserted depends on the microcontroller you are using and the crystal you have selected.

The instruction is executed by the LCD at the moment the EN line is brought low with a final CLR EN instruction.

CHECKING THE BUSY STATUS OF THE LCD

As previously mentioned, it takes a certain amount of time for each instruction to be executed by the LCD. The delay varies depending on the frequency of the crystal attached to the oscillator input of the 8051 as well as the instruction which is being executed.

While it is possible to write code that waits for a specific amount of time to allow the LCD to execute instructions, this method of "waiting" is not very flexible. If the crystal frequency is changed, the software will need to be modified. Additionally, if the LCD itself is changed for another LCD which, although 44780 compatible, requires more time to perform its operations, the program will not work until it is properly modified.

A more robust method of programming is to use the "Get LCD Status" command to determine whether the LCD is still busy executing the last instruction received.

The "Get LCD Status" command will return to us two tidbits of information; the information that is useful to us right now is found in DB7. In summary, when we issue the "Get LCD Status" command the LCD will immediately raise DB7 if it's still busy executing a command or lower DB7 to indicate that the LCD is no longer occupied. Thus our program can query the LCD until DB7 goes low, indicating the LCD is no longer busy. At that point we are free to continue and send the next command.

Since we will use this code every time we send an instruction to the LCD, it is useful to make it a subroutine. Let's write the code:

```
WAIT_LCD:
        CLR EN ;Start LCD command
        CLR RS ;It's a command
        SETB RW ;It's a read command
        MOV DATA,#0FFh ;Set all pins to FF initially
        SETB EN ;Clock out command to LCD
        MOV A,DATA ;Read the return value
        JB ACC.7,WAIT_LCD ;If bit 7 high, LCD still busy
        CLR EN ;Finish the command
        CLR RW ;Turn off RW for future commands
        RET
```

Thus, our standard practice will be to send an instruction to the LCD and then call our **WAIT_LCD** routine to wait until the instruction is completely executed by the LCD. This will assure that our program gives the LCD the time it needs to execute instructions and also makes our program compatible with any LCD, regardless of how fast or slow it is.

The above routine does the job of waiting for the LCD, but were it to be used in a real application a very definite improvement would need to be made: as written, if the LCD never becomes "not busy" the program will effectively "hang," waiting for DB7 to go low. If this never happens, the program will freeze. Of course, this should never happen and **won't** happen when the hardware is working properly. But in a real application it would be wise to put some kind of time limit on the delay--for example, a maximum of 256 attempts to wait for the busy signal to go low. This would guarantee that even if the LCD hardware fails, the program would not lock up.

INITIALIZING THE LCD

Before you may really use the LCD, you must initialise and configure it. This is accomplished by sending a number of initialisation instructions to the LCD.

The first instruction we send must tell the LCD whether we'll be communicating with it with an 8-bit or 4-bit data bus. We also select a 5x8 dot character font. These two options are selected by sending the command 38h to the LCD as a command. As you will recall from the last section, we mentioned that the **RS** line must be low if we are sending a command to the LCD. Thus, to send this 38h command to the LCD we must execute the following 8051 instructions:

```
CLR RS
MOV DATA,#38h
SETB EN
CLR EN
LCALL WAIT_LCD
```

The LCD command 38h is really the sum of a number of option bits. The instruction itself is the instruction 20h ("Function set"). However, to this we add the values 10h to indicate an 8-bit data bus plus 08h to indicate that the display is a two-line display.

We've now sent the first byte of the initialization sequence. The second byte of the initialization sequence is the instruction 0Eh. Thus we must repeat the initialization code from above, but now with the instruction. Thus the the next code segment is:

```
CLR RS
MOV DATA,#0Eh
SETB EN
CLR EN
LCALL WAIT_LCD
```

The command 0Eh is really the instruction 08h plus 04h to turn the LCD on. To that an additional 02h is added in order to turn the cursor on.

The last byte we need to send is used to configure additional operational parameters of the LCD. We must send the value 06h.

```
CLR RS
MOV DATA,#06h
SETB EN
CLR EN
LCALL WAIT_LCD
```

The command 06h is really the instruction 04h plus 02h to configure the LCD such that every time we send it a character, the cursor position automatically moves to the right.

So, in all, our initialization code is as follows:

```
INIT_LCD:
        CLR RS
        MOV DATA,#38h
        SETB EN
        CLR EN
        LCALL WAIT_LCD
        CLR RS
        MOV DATA,#0Eh
        SETB EN
        CLR EN
        LCALL WAIT_LCD
        CLR RS
        MOV DATA,#06h
        SETB EN
        CLR EN
        LCALL WAIT_LCD
        RET
```

Having executed this code the LCD will be fully initialized and ready for us to send display data to it.

CLEARING THE DISPLAY

When the LCD is first initialized, it's always a good idea to Clean screen yourself so that you can be completely sure that the display is the way you want it.

An LCD command exists to accomplish this function. Not surprisingly, it is the command 01h. Since clearing the screen is a function we very likely will wish to call more than once, it's a good idea to make it a subroutine:

```
CLEAR_LCD:
        CLR RS
        MOV DATA,#01h
        SETB EN
        CLR EN
        LCALL WAIT_LCD
        RET
```

How that we've written a "Clear Screen" routine, we may clear the LCD at any time by simply executing an **LCALL CLEAR_LCD**.

Executing the "Clear Screen" instruction on the LCD also positions the cursor in the upper left-hand corner as we would expect.

WRITING TEXT TO THE LCD

Now we get to the real meat of what we're trying to do: All this effort is really so we can display text on the LCD. Really, we're pretty much done.

Once again, writing text to the LCD is something we'll almost certainly want to do over and over--so let's make it a subroutine.

```
WRITE_TEXT:
        SETB RS
        MOV DATA,A
        SETB EN
        CLR EN
        LCALL WAIT_LCD
        RET
```

The **WRITE_TEXT** routine that we just wrote will send the character in the accumulator to the LCD which will, in turn, display it. Thus to display text on the LCD all we need to do is load the accumulator with the byte to display and make a call to this routine. Pretty easy, huh?

A "HELLO WORLD" PROGRAM

Now that we have all the component subroutines written, writing the classic "Hello World" program--which displays the text "Hello World" on the LCD is a relatively trivial matter. Consider:

```
LCALL INIT_LCD
LCALL CLEAR_LCD
MOV A,#'H'
LCALL WRITE_TEXT
MOV A,#'E'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'O'
LCALL WRITE_TEXT
MOV A,#' '
LCALL WRITE_TEXT
MOV A,#'W'
LCALL WRITE_TEXT
MOV A,#'O'
LCALL WRITE_TEXT
MOV A,#'R'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'D'
LCALL WRITE_TEXT
```

The above "Hello World" program should, when executed, initialize the LCD, clear the LCD screen, and display "Hello World" in the upper left-hand corner of the display.

CURSOR POSITIONING

The above "Hello World" program is simplistic in the sense that it prints its text in the upper left-hand corner of the screen. However, what if we wanted to display the word "Hello" in the upper left-hand corner but wanted to display the word "World" on the second line at the tenth character? This sounds simple--and actually, it **is** simple. However, it requires a little more understanding of the design of the LCD.

The 44780 contains a certain amount of memory which is assigned to the display. All the text we write to the 44780 is stored in this memory, and the 44780 subsequently reads this memory to display the text on the LCD itself. This memory can be represented with the following "memory map":

In the above memory map, the area shaded in blue is the visible display. As you can see, it measures 16 characters per line by 2 lines. The numbers in each box is the memory address that corresponds to that screen position.

Thus, the first character in the upper left-hand corner is at address 00h. The following character position (character #2 on the first line) is address 01h, etc. This continues until we reach the 16th character of the first line which is at address 0Fh.

However, the first character of line 2, as shown in the memory map, is at address 40h. This means if we write a character to the last position of the first line and then write a second character, the second character will **not** appear on the second line. That is because the second character will effectively be written to address 10h--but the second line begins at address 40h.

Thus we need to send a command to the LCD that tells it to position the cursor on the second line. The "Set Cursor Position" instruction is 80h. To this we must add the address of the location where we wish to position the cursor. In our example, we said we wanted to display "World" on the second line on the tenth character position.

Referring again to the memory map, we see that the tenth character position of the second line is address 4Ah. Thus, before writing the word "World" to the LCD, we must send a "Set Cursor Position" instruction--the value of this command will be 80h (the instruction code to position the cursor) plus the address 4Ah. 80h + 4Ah = CAh. Thus sending the command CAh to the LCD will position the cursor on the second line at the tenth character position:

```
CLR RS
MOV DATA,#0CAh
SETB EN
CLR EN
LCALL WAIT_LCD
```

The above code will position the cursor on line 2, character 10. To display "Hello" in the upper left-hand corner with the word "World" on the second line at character position 10 just requires us to insert the above code into our existing "Hello World" program. This results in the following:

```
LCALL INIT_LCD
LCALL CLEAR_LCD
MOV A,#'H'
LCALL WRITE_TEXT
MOV A,#'E'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'O'
LCALL WRITE_TEXT
CLR RS
MOV DATA,#0CAh
SETB EN
CLR EN
LCALL WAIT_LCD
MOV A,#'W'
LCALL WRITE_TEXT
MOV A,#'O'
LCALL WRITE_TEXT
MOV A,#'R'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'D'
LCALL WRITE_TEXT
```

SUMMARY

This background has presented the underlying concepts of programming an LCD display. this background should get me going in the right direction.

Code Background:

As previously mentioned in ***LCD Background*** will use the above code to write something on LCD, but that is not about just writing something on the LCD there will be buttons,random numbers,delays and so on. I will use random numbers to handle with random shapes in my Slot Machine so lets talk about random numbers

There will be a button to start game, and I am going to use this button to generate random numbers by calculating the time while the button holding and this will give me a random number because it calculates milliseconds. Let's write the code:

```
;----------------------<Random Number Generator1>-------------
RANDOM:
        MOV   R1,#00H
INC_:  INC    R1
        MOV   A,P2
        CJNE  A,#0F7H,RND        ;STAY UNTIL P2.3 "1"
        CJNE  R1,#03H,INC_
        SJMP  RANDOM

;----------------------</Random Number Generator1>-------------

;those lines are if closes
;if R1 has 01h it will store '1' into acc
;elseif R1 has 02h it will store '2' into acc
;elseif R1 has 02h it will store '3' into acc
;----------------------<Random picture selsct1>-----------------------------------------

RND:  INC     R7      ;this is my counter for numbers it tells me which number i am writing
        MOV   A,R1
        CJNE  R1,#01H,NEXT1
        MOV   R1,#'@'
        CALL  WRITE_1
NEXT1:CJNE        R1,#02H,NEXT2
        MOV   R1,#'&'
        CALL  WRITE_1
NEXT2:
        MOV   R1,#'#'
        CALL  WRITE_1
```

```
;----------------------</Random picture selsct1>----------------------------------------
```

On this code I am incrementing R1 form “0” to “3”, and when it is “1” it calls WRITE_1 and it writes “@” when it is 2 it calls WRITE_1 and it writes “&” when it is 3 it calls WRITE_1 and it writes“#” it means that I am not using “0” I am using 1, 2, 3.

Before I start my project I check how can I draw my own shapes on to LCD screen. And I found some information about it. I am going to show it.

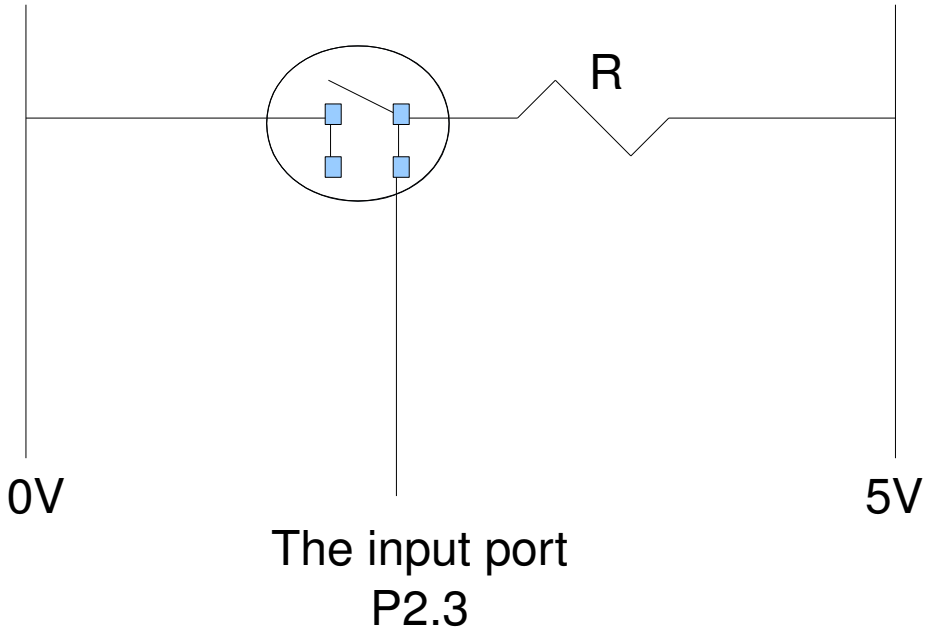
- **Relationship between Character Code and CGRAM**

Character code								CGRAM Address						CGRAM Data								Pattern number
D7	D6	D5	D4	D3	D2	D1	D0	A5	A4	A3	A2	A1	A0	P7	P6	P5	P4	P3	P2	P1	P0	
0	0	0	0	x	0	0	0	0	0	0	0	0	0	x	x	x	0	1	1	1	0	pattern 1
											0	0	1	x	x	x	1	0	0	0	1	
											0	1	0	x	x	x	1	0	0	0	1	
											0	1	1	x	x	x	1	1	1	1	1	
											1	0	0	x	x	x	1	0	0	0	1	
											1	0	1	x	x	x	1	0	0	0	1	
											1	1	0	x	x	x	1	0	0	0	1	
											1	1	1	x	x	x	0	0	0	0	0	
0	0	0	0	x	1	1	1	0	0	0	0	0	0	x	x	x	1	0	0	0	1	pattern8
											0	0	1	x	x	x	1	0	0	0	1	
											0	1	0	x	x	x	1	0	0	0	1	
											0	1	1	x	x	x	1	1	1	1	1	
											1	0	0	x	x	x	1	0	0	0	1	
											1	0	1	x	x	x	1	0	0	0	1	
											1	1	0	x	x	x	1	0	0	0	1	
											1	1	1	x	x	x	0	0	0	0	0	

This picture explains everything about how to draw our shapes on to LCD.

Buttons Background:

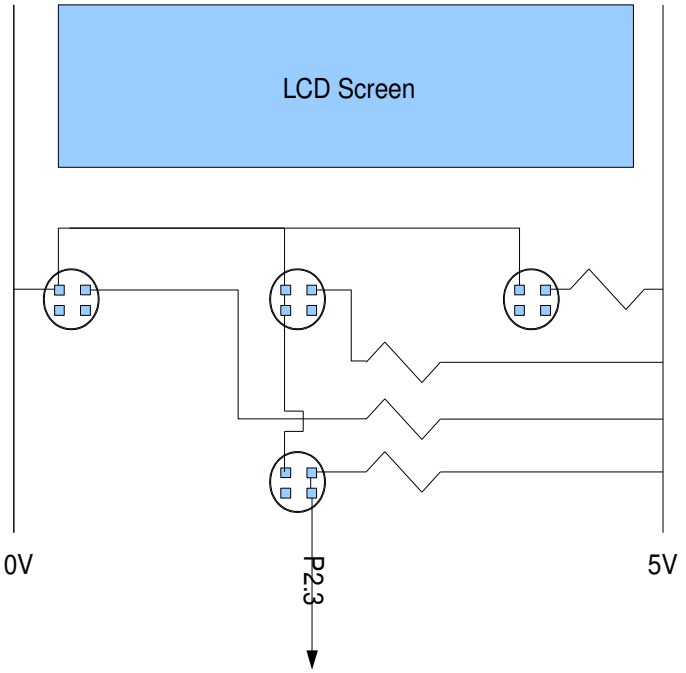
Working with buttons is really straight forward the only thing you should do is use a switch and connect one leg to resistor and other leg to 0V , and connect resistors free leg to 5V, the main purpose of using the resistor to handle the direct connection between 0V and 5V and save the battery.



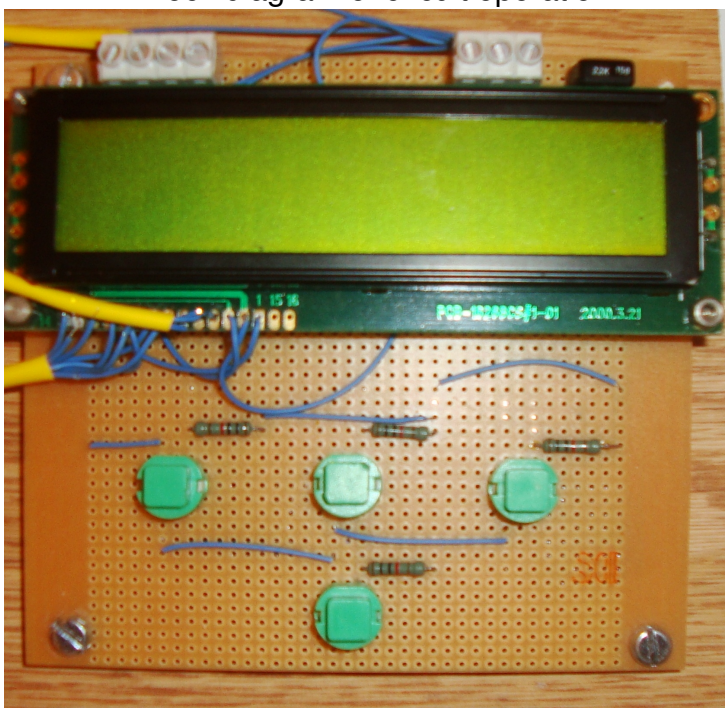
After you connect all of those the last thing is connect it to the on of our ports , when we press button it connects the 0V to one of our ports leg and when we stop holding it it connects 5V to our ports leg. That is what we want from a button though , it gives 0V when you press it and gives 5V as normal.

Chapter 3: Hardware Design :

My hardware was really straight forward I used resistors, one LCD, 4 Buttons, and cables, I am going to show you my circuit diagram.



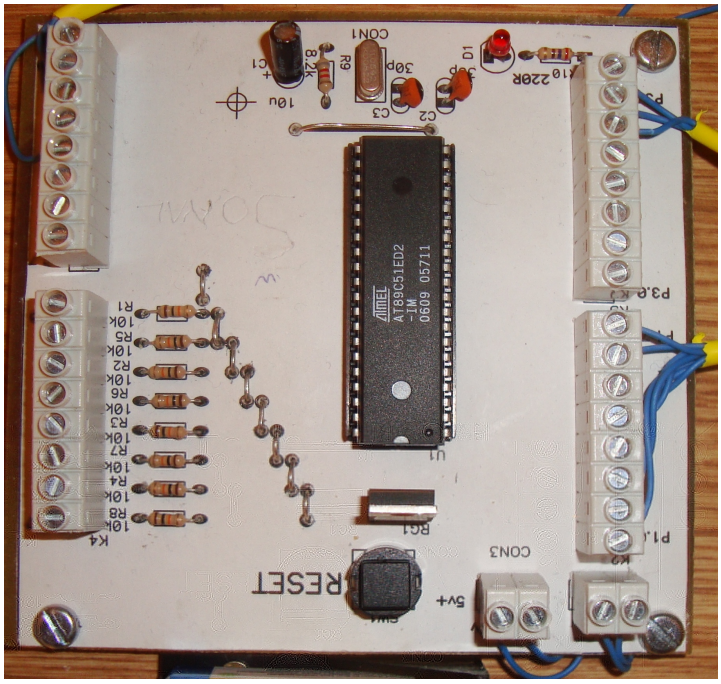
Block diagram of circuit operation



Finished displayed circuit board

If you are doing the same project and trying to copy this board, use drawing, not the picture because I solder some cables from the back as well.

I would like to show my 8051 board as well



Those background's and research's than are enough to start from the right point.

Chapter 4: Software Design :

Here is the Algorithm

If button is pressed than start counting up from 1 to 3
when it stop holding than select the number
Compare the number and the characters (for example for 1 write up “@”, for 2 “#” for 3 “&”)
Show this character for 2 more loop and keep it on the screen for 2 more loop
Show in the screen “please press the button 2 times more”

Start loop again for random pictures

Check if 3 pictures (numbers) are same or not?
If they are same than wrote “Lucky Day” to screen if not give “Out of Luck”

Chapter 5: Future Development :

There should be a LED to count how many “Lucky” outputs you get

I should draw Fruits on the LCD display and there should be Apple, Banana, and Orange

I might draw my circuit diagram on the computer and print it than use it instead of a ready one.

Chapter 6: Testing Evaluation :

About hardware I didn't test it, because it was working when I finish it,But I spend %50 of my time on Software testing about Fruits (about how I can draw them, I succeeded I could draw my my own banana on it :) but because of the way how I write my letters to LCD I couldn't use the shapes, If I use the DB's on the first part of my program , I could use the shapes then.

What I should do is

```
Display:mov DPTR,#data_out      ;move the output/data in the data pointer
        push Acc                ;push the accumulator
        movC A,@A+DPTR          ;output the value to the data pointer
        mov P0,A                ;move the contents of port1 into the accumulator
        CLR A                   ;clear the accumulator
        ACALL       check       ;

org 300h
data_out: db “L”,”U”,”C”,”K”,”Y”, ; Those are the letters I will wrote
```