Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

RAPORT

la Programarea aplicațiilor incorporate și independente de platformă

Lucrare de laborator Nr. 6

Tema: Simple application using studied topics: ”Electronic Voting System”

A efectuat st. gr. FAF-141: Oxana Dunav

A verificat: Andrei Bragarenco

Chişinău 2016

Topic

Simple application using studied topics: ”Electronic Voting System”

Purpose

- Create a simple application using the studied topics
- Understand better embedded systems

Task

Create an electronic voting system that will keep track of 4 voting people and display the solution on the screen. Use a button to reset the counters to zero.

Domain

➔ Embedded systems

An embedded system is a computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints. It is *embedded* as part of a complete device often including hardware and mechanical parts. Embedded systems control many devices in common use today. Ninety-eight percent of all microprocessors are manufactured as components of embedded systems.

Modern embedded systems are often based on microcontrollers (i.e. CPUs with integrated memory or peripheral interfaces),[7] but ordinary microprocessors (using external chips for memory and peripheral interface circuits) are also common, especially in more-complex systems. In either case, the processor(s) used may be types ranging from general purpose to those specialised in certain class of computations, or even custom designed for the application at hand. A common standard class of dedicated processors is the digital signal processor (DSP).

Embedded systems range from portable devices such as digital watches and MP3 players, to large stationary installations like traffic lights, factory controllers, and largely complex systems like hybrid vehicles, MRI, and avionics. Complexity varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large chassis or enclosure.

➔ Microcontroller

A microcontroller (or MCU, short for microcontroller unit) is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of Ferroelectric RAM, NOR flash or OTP ROM is also often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications consisting of various discrete chips.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems.

When you write a program for your microcontroller you are really writing a program that is executed by the MC CPU (central processing unit)

In the simplest sense, a CPU is that part of the microcontroller that executes instructions. It does this in a series of steps:

- Fetch an instruction from the "next instruction" memory location pointer
- Execute that instruction
- Advance the "next instruction" pointer accordingly Every computer program is just a repetitive execution of this sequence.

➔ LCD Module

LCD Modules can present textual information to user. It’s like a cheap “monitor” that you can hook in all of your gadgets. They come in various types. The most popular one can display 2 lines of 16 characters.

LCD Module Pin Configuration
1 VSS (GND Supply)
2 VCC (+5V)
3 VEE (Contrast Adjust)
4 RS
5 R/W
6 E
7 DB0
8 DB1
9 DB2
10 DB3
11 DB4

12 DB5

13 DB6

14 DB7

15 LED +

16 LED -

➔ GPIO

Every micro-controller has GPIO ports. GPIO stands for general purpose input output. These GPIO ports are used to take input on a micro-controller pin or output a value on micro-controller pin.

AVR is 8 bit microcontroller. All its ports are 8 bit wide. Every port has 3 registers associated with it each one with 8 bits. Every bit in those registers configure pins of particular port. Bit0 of these registers is associated with Pin0 of the port, Bit1 of these registers is associated with Pin1 of the port, …. and like wise for other bits.

These three registers are as follows : (x can be replaced by A,B,C,D as per the AVR you are using)

- DDRx register
- PORTx register
- PINx register

➔ DDRx register

DDRx (Data Direction Register) configures data direction of port pins. Means its setting determines whether port pins will be used for input or output. Writing 0 to a bit in DDRx makes corresponding port pin as input, while writing 1 to a bit in DDRx makes corresponding port pin as output.

Example:

to make all pins of port A as input pins :
DDRA = 0b00000000;

to make all pins of port A as output pins :
DDRA = 0b11111111;

to make lower nibble of port B as output and higher nibble as input :
DDRB = 0b00001111;

➔ PINx register

PINx (Port IN) used to read data from port pins. In order to read the data from port pin, first you have to change port's data direction to input. This is done by setting bits in DDRx to zero. If port is made output, then reading PINx register will give you data that has been output on port pins.

Now there are two input modes. Either you can use port pins as tri stated inputs or you can activate internal pull up. It will be explained shortly.

To read data from port A.

DDRA = 0x00 //set port for input

X = PINA // input

➔ PORTx register

PORTx is used for two purposes.

1) To output data : when port is configured as output
2) To activate/deactivate pull up resistors – when port is configures as input

To output data

When you set bits in DDRx to 1, corresponding pins becomes output pins. Now you can write data into respective bits in PORTx register. This will immediately change state of output pins according to data you have written.

In other words to output data on to port pins, you have to write it into PORTx register. However do not forget to set data direction as output.

Example :

To output 0xFF data on port b

```
DDRB = 0b11111111;      //set all pins of port b as outputs

PORTB = 0xFF;           //write data on port
```

To output data in variable x on port a

```
DDRA = 0xFF;            //make port A as output

PORTA = x;              //output variable on port
```

To output data on only 0th bit of port c

```
DDRC.0 = 1;       //set only 0th pin of port c as output

PORTC.0 = 1;1     //make it high.
```

➔ To activate/deactivate pull up resistors

When you set bits in DDRx to 0, i.e. make port pins as inputs, then corresponding bits in PORTx register are used to activate/deactivate pull-up registers associated with that pin. In order to activate pull-up resister, set bit in PORTx to 1, and to deactivate (i.e to make port pin tri stated) set it to 0.

In input mode, when pull1-up is enabled, default state of pin becomes '1'. So even if you don't connect anything to pin and if you try to read it, it will read as 1. Now, when you externally drive that pin to zero(i.e. connect to ground / or pull-down), only then it will be read as 0.

However, if you configure pin as tri state. Then pin goes into state of high impedance. We can say, it is now simply connected to input of some OpAmp inside the uC and no other circuit is driving it from uC. Thus pin has very high impedance. In this case, if pin is left floating (i.e. kept unconnected) then even small static charge present on surrounding objects can change logic state of pin. If you try to read corresponding bit in pin register, its state cannot be predicted. This may cause your program to go haywire, if it depends on input from that particular pin.

Thus while, taking inputs from pins / using micro-switches to take input, always enable pull-up resistors on input pins.

NOTE : while using on chip ADC, ADC port pins must be configured as tri stated input.

Example :

To make port a as input with pull-ups enabled and read data from port a

```
DDRA = 0x00;       //make port a as input

PORTA = 0xFF;      //enable all pull-ups

y = PINA;          //read data from port a pins
```

To make port b as tri stated input

```
DDRB  = 0x00;      //make port b as input
```

PORTB = 0x00; //disable pull-ups and make it tri state

To make lower nibble of port a as output, higher nibble as input with pull-ups enabled

DDRA = 0x0F; //lower nib> output, higher nib> input

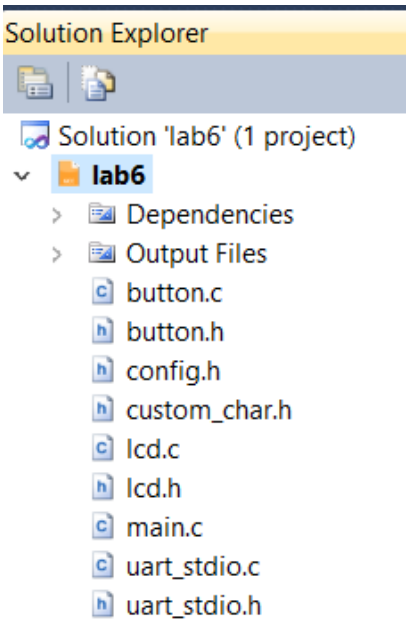
PORTA = 0xF0; //lower nib> set output pins to 0, higher nib> enable pull-ups

Used Resources

LM016L–LCD

Solution

The *Project Structure* looks in this way



lcd.h / lcd.c

Contains declaration and implementation of LCD Driver. It has functions that implements the writing strings and numbers on the screen and others.

button.h /button.c

Contains declaration and implementation of the buttons that control the voting.

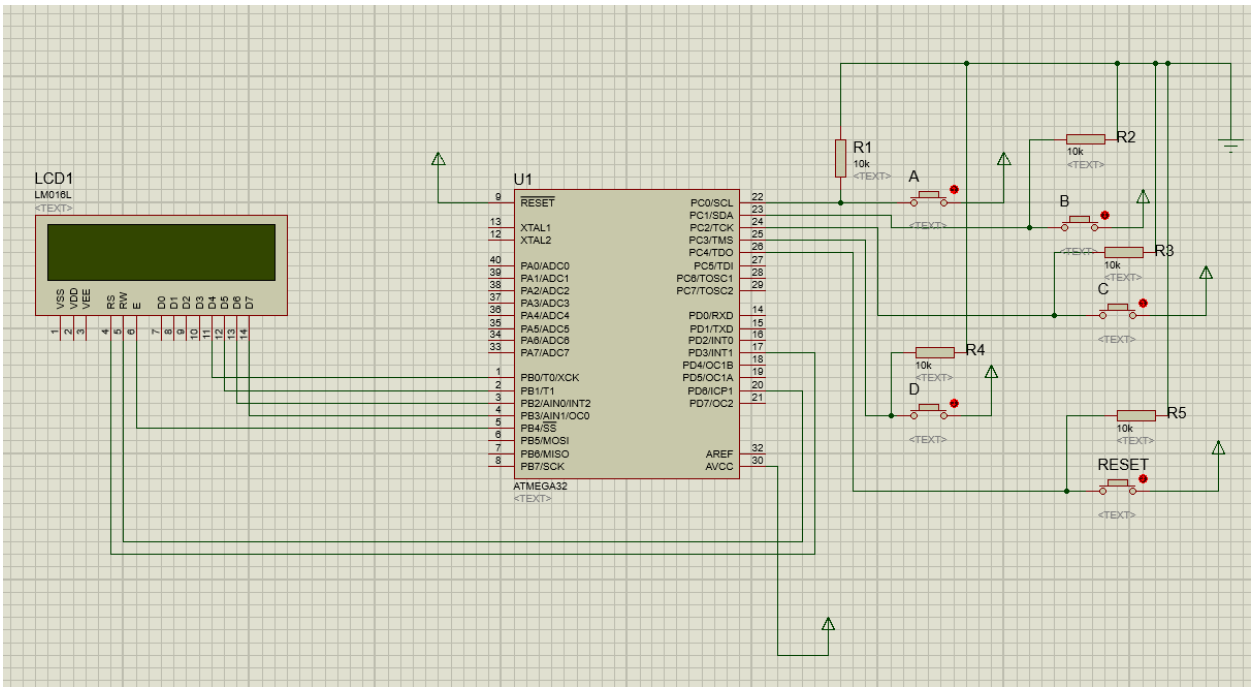
Main Program

Main Program is responsible for :

- Initialization of 5 buttons , which are responsible for the voting system(4 buttons for candidates, 1 for reset)
- Initialization of LCD module
- Initialization of voting scores
- program updates the scores in dependency of which button was pressed

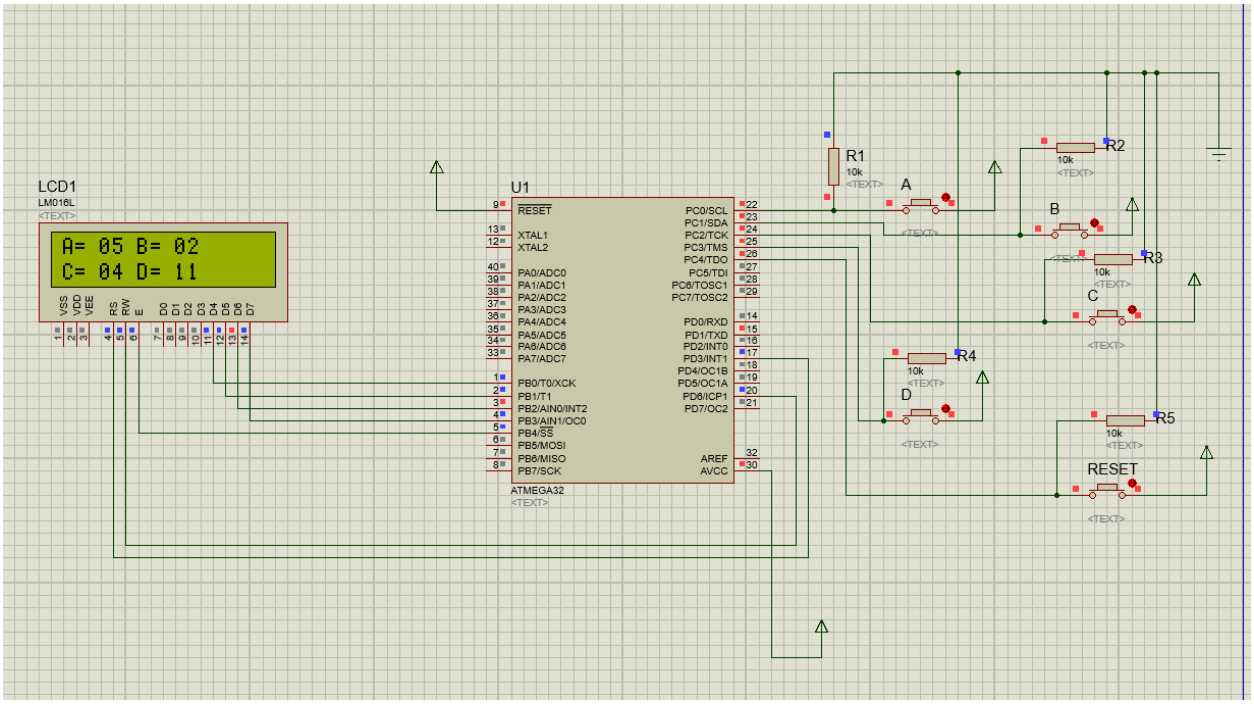
Schematics

For our laboratory work we need ATMega32 MCU, 5 Push Buttons, resistors and LCD.

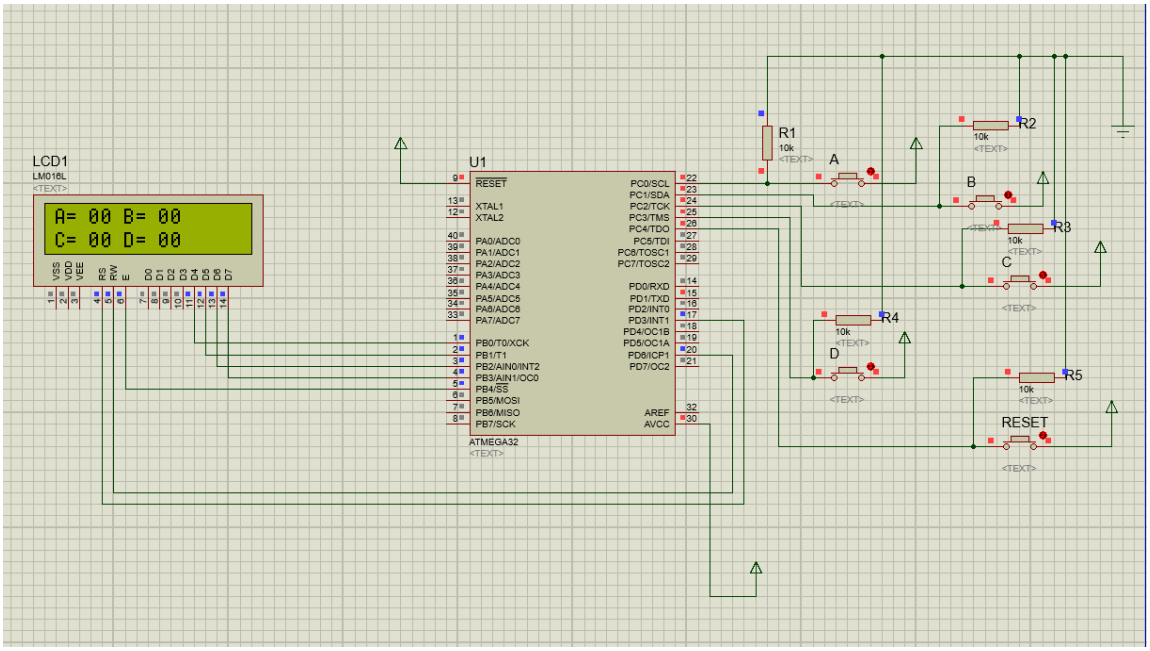


Simulation Result

Result of voting system on the screen.



Voting system after pressing button RESET



Conclusion

During this laboratory work I applied my knowledge gained in this course to implement a simple electronic voting system.

This laboratory work has helped me understand better the embedded systems and I realized how many things you can do with it. It has many applications in our lives and it is interesting for me to study more things about embedded systems in the future.

Appendix

main.c

```
#include <avr/io.h>
#include "button.h"
#include "lcd.h"
#include <avr/delay.h>

int main(void) {

	int aScore = 0;
	int bScore = 0;
	int cScore = 0;
	int dScore = 0;
	initButtonOne();
	initButtonTwo();
	initButtonThree();
	initButtonFour();
	initButtonReset();
```

```
	uart_stdio_Init();

	//Initialize LCD module
    LCDInit(LS_NONE);

    //Clear the screen
    LCDClear();
	LCDWriteStringXY(0,0,"A=");
	LCDWriteStringXY(6,0,"B=");
	LCDWriteStringXY(0,1,"C=");
	LCDWriteStringXY(6,1,"D=");

	LCDWriteIntXY(2,0,aScore,2);
	LCDWriteIntXY(8,0,bScore,2);
	LCDWriteIntXY(2,1,cScore,2);
	LCDWriteIntXY(8,1,dScore,2);

    while(1) {
		_delay_ms(10);

		if(!isButtonOnePressed()) {
			aScore++;
			LCDWriteIntXY(2,0,aScore,2);

		}
		if(!isButtonTwoPressed()) {
			bScore++;
			LCDWriteIntXY(8,0,bScore,2);
		}
		if(!isButtonThreePressed()) {
			cScore++;
			LCDWriteIntXY(2,1,cScore,2);
		}
		if(!isButtonFourPressed()) {
			dScore++;
			LCDWriteIntXY(8,1,dScore,2);
		}
		if(!isButtonResetPressed()) {
			aScore = 0;
			bScore = 0;
			cScore = 0;
			dScore = 0;
			LCDWriteIntXY(2,0,aScore,2);
			LCDWriteIntXY(8,0,bScore,2);
			LCDWriteIntXY(2,1,cScore,2);
			LCDWriteIntXY(8,1,dScore,2);
		}
	}

}
```

button.h

```
#ifndef BUTTON_H_
#define BUTTON_H_
#include <avr/io.h>
```

```
void initButtonOne();
void initButtonTwo();
void initButtonThree();
void initButtonFour();
void initButtonReset();

int isButtonOnePressed();
int isButtonTwoPressed();
int isButtonThreePressed();
int isButtonFourPressed();
int isButtonResetPressed();

#endif /* BUTTON_H_ */
```

button.c

```
#include "button.h"

///Makes first 4 pins of PORTC as Input
void initButtonOne() {
	DDRC &= ~(1 << PORTC0);
}

void initButtonTwo() {
	DDRC &= ~(1 << PORTC1) ;
}

void initButtonThree() {
	DDRC &= ~(1 << PORTC2) ;
}
void initButtonFour() {
	DDRC &= ~(1 << PORTC3) ;
}

void initButtonReset(){
	DDRC &= ~(1 << PORTC4) ;
}

int isButtonOnePressed() {
	return PINC & (1<<PORTC0);
}

int isButtonTwoPressed() {
	return PINC & (1<<PORTC1);
}

int isButtonThreePressed() {
	return PINC & (1<<PORTC2);
}
int isButtonFourPressed() {
	return PINC & (1<<PORTC3);
}

int isButtonResetPressed() {
	return PINC & (1<<PORTC4);
```

```
}
```

lcd.h

```
#include <avr/io.h>

#ifndef _LCD_H
#define _LCD_H

#include "config.h"


//************************************************

#define LS_BLINK 0B00000001
#define LS_ULINE 0B00000010
#define LS_NONE       0B00000000



/*************************************************
                    F U N C T I O N S
*************************************************/



void LCDInit(uint8_t style);

void LCDWriteString(const char *msg);
void LCDWriteFString(const char *msg);

void LCDWriteInt(int val,int8_t field_length);
void LCDGotoXY(uint8_t x,uint8_t y);

//Low level
void LCDByte(uint8_t,uint8_t);
#define LCDCmd(c) (LCDByte(c,0))
#define LCDData(d) (LCDByte(d,1))

void LCDBusyLoop();


/*************************************************
                    F U N C T I O N S     E N D
*************************************************/


/*************************************************
      M A C R O S
*************************************************/
#define LCDClear() LCDCmd(0b00000001)
#define LCDHome() LCDCmd(0b00000010);

#define LCDWriteStringXY(x,y,msg) {\
 LCDGotoXY(x,y);\
 LCDWriteString(msg);\
```

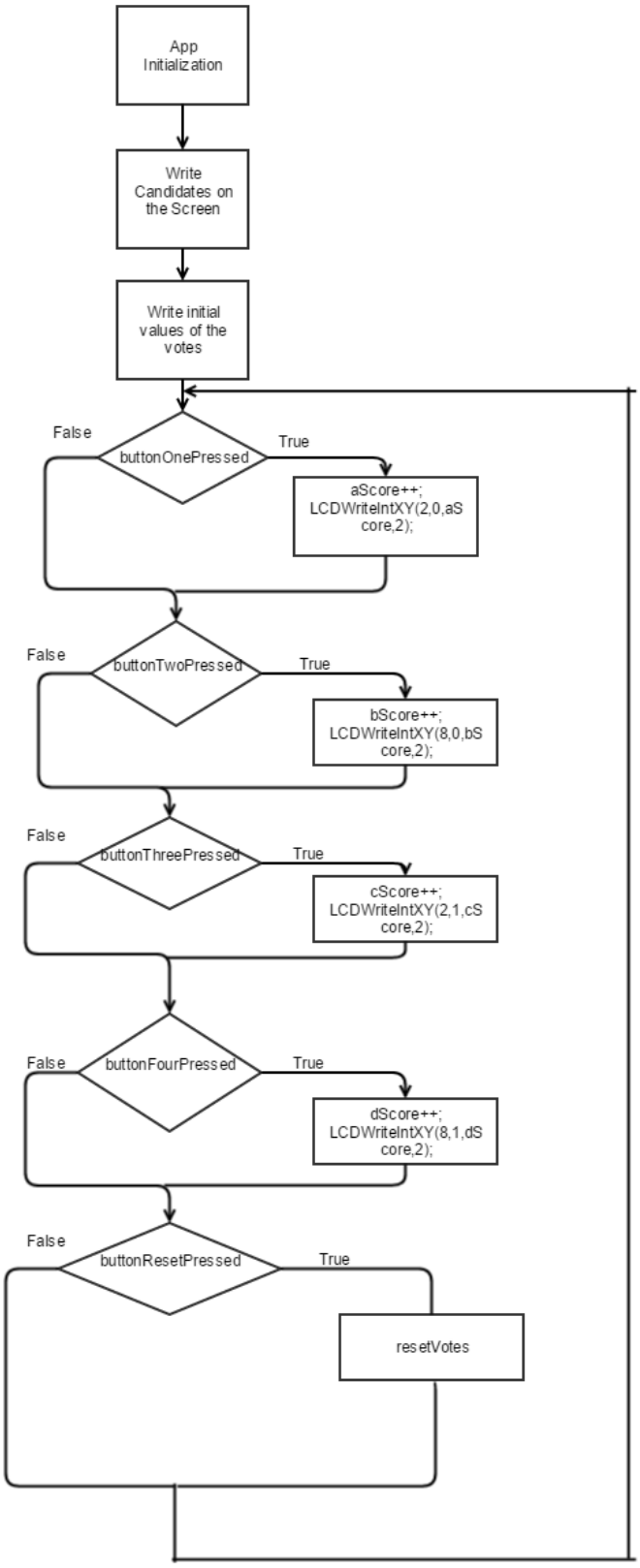
```
}

#define LCDWriteFStringXY(x,y,msg) {\
LCDGotoXY(x,y);\
LCDWriteFString(msg);\
}

#define LCDWriteIntXY(x,y,val,fl) {\
 LCDGotoXY(x,y);\
 LCDWriteInt(val,fl);\
}
/*************************************************/

#endif
```

FlowChart


App Initialization
Write Candidates on the Screen
Write initial values of the votes
False
buttonOnePressed
True
aScore++; LCDWriteIntXY(2,0,aScore,2);
False
buttonTwoPressed
True
bScore++; LCDWriteIntXY(8,0,bScore,2);
False
buttonThreePressed
True
cScore++; LCDWriteIntXY(2,1,cScore,2);
False
buttonFourPressed
True
dScore++; LCDWriteIntXY(8,1,dScore,2);
False
buttonResetPressed
True
resetVotes