Hardware Software Interface – Coursework 2.

1. Daniyar Nazarbayev

2. Chester Corpuz

~~3. Abdullah Mohammed El Sayed~~

Table of Contents

1. Code.

1.1 Pure C code.

```
#define STRB 12

#define RS 6

#define DATA_1 20

#define DATA_2 21

#define DATA_3 26

#define DATA_4 16

#define DATA_5 19

#define DATA_6 13

#define DATA_7 5

#define DATA_8 7


#define BUTTON 14

#define YELLOW_LED 18

#define RED_LED 15


#define CLEAR_SCREEN 1

#define RETURN_HOME 2


#define ENTRY_MODE 4

#define ENTRY_INCREMENT 2

#define ENTRY_DECREMENT 0

#define ENTRY_SHIFT 1

#define ENTRY_STAY 0


#define DISPLAY1 8

#define DISPLAY1_ON 4
```

```
#define DISPLAY1_OFF 0
#define DISPLAY1_CURSOR_ON 2
#define DISPLAY1_CURSOR_OFF 0
#define DISPLAY1_BLINK_ON 1
#define DISPLAY1_BLINK_OFF 0

#define DISPLAY2 16
#define DISPLAY2_SCROLL_ON 8
#define DISPLAY2_SCROLL_OFF 0
#define DISPLAY2_RIGHT 4
#define DISPLAY2_LEFT 0

#define FUNCTION_SET 32
#define FUNCTION_4_BIT 0
#define FUNCTION_8_BIT 16
#define FUNCTION_2_LINES 8
#define FUNCTION_1_LINE 0
#define FUNCTION_FONT_SMALL 0
#define FUNCTION_FONT_LARGE 1

// OR it with 0-63
#define CHARACTER_GENERATION 64

// OR it with 0-127
// line 1 = 0x0 to 0xF
// line 2 = 0x40 to 0x4F
#define DDRAM 128

#define COMMAND 0
```

```
#define CHARACTER 1

#define OUTPUT 1
#define INPUT 0

#define SUCCESS 0
#define FAILURE 1

#define PAGE_SIZE          (4*1024)
#define BLOCK_SIZE         (4*1024)

#include <stdio.h>
#include <stdarg.h>
#include <stdint.h>
#include <stdlib.h>
#include <ctype.h>
#include <poll.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/ioctl.h>
#include <unistd.h>
```

```
struct pins;

void init_lcd();
void delay(unsigned int);
void strobe();
void send_data(int, unsigned char);
void write_to_data_pins(unsigned char);
void send_string(char[], int);
void position();
void reset_lcd();
void set_4_pin_mode();
void delayMicroseconds (unsigned int);
void command_4bits(unsigned char);
void change_position(int, int);

void gpiomode_w(int, int);
void init_gpio();
int gpiomode_r(int);
void gpioset(int);
void gpioclear(int);
int gpiolev(int);
int gpioeds_r(int);
void gpioeds_w(int, unsigned int);
void gpiofen(int, unsigned int);
void gpioren(int, unsigned int);
void reset_gpio();

void ctrlc_handler(int);
```

```
void blink(int, int);
int press_blink(int, int, int);
int detect_presses(int);
void print_lcd(char *);


static volatile unsigned int gpiobase ;
static volatile uint32_t *gpio ;
static struct pins * lcd_pins;
static int debug;

struct pins{
        int x, y;
        int data[8];
        int rs;
        int strb;
        int bit_mode;
};

//=============================================================================
================
//============================ HIGH LEVEL FUNCTIONS
==============================================
//=============================================================================
================
int main(int argc, char** argv){

// DEBUG MODE
// shows print statements in the terminal while the game is running
if(argc > 1 && (strcmp(argv[1], "d") == 0)){
```

```
        debug = 1;
} else {
        debug = 0;
}

//============================== LCD SETUP
=====================================================
        lcd_pins = malloc(sizeof(struct pins));
        signal(SIGINT, ctrlc_handler);
        init_gpio();
        init_lcd(4);

        if(lcd_pins->bit_mode == 4){
                set_4_pin_mode();
        }

        send_data(COMMAND, FUNCTION_SET | (lcd_pins->bit_mode == 4 ? FUNCTION_4_BIT :
FUNCTION_8_BIT) | FUNCTION_2_LINES | FUNCTION_FONT_SMALL);
        send_data(COMMAND, DISPLAY1 | DISPLAY1_ON | DISPLAY1_CURSOR_ON |
DISPLAY1_BLINK_ON);
        send_data(COMMAND, DISPLAY2 | DISPLAY2_SCROLL_OFF | DISPLAY2_RIGHT);
        send_data(COMMAND, CLEAR_SCREEN);
        //send_data(COMMAND, RETURN_HOME);
        change_position(0, 0);
        send_data(COMMAND, ENTRY_MODE | ENTRY_INCREMENT | ENTRY_STAY);

//============================ GENERATE RANDOM SEQUENCE
================================================

        srand(time(NULL));
```

```
	int sequence[3];
	int i;
	for(i = 0; i < 3; i++){
		sequence[i] = (rand() % 3) + 1;
	}


	if(debug) printf("SECRET: %d %d %d\n", sequence[0],sequence[1],sequence[2]);

//======================================== GAME
=====================================================

	int button, led1, led2;

	button = BUTTON;
	led1 = RED_LED;
	led2 = YELLOW_LED;

	gpiomode_w(led1, OUTPUT);
	gpiomode_w(led2,OUTPUT);
	gpiomode_w(button, INPUT);

	char buf[32];
	char * str = "MASTERMIND";
	print_lcd(str);
	delay(2000);
	send_data(COMMAND, CLEAR_SCREEN);

while(1){
```

```
int guessnum = 0;
int guesses[3];
if(debug) printf("INPUT PHASE\n");

//INPUT 3 TIMES
while(guessnum < 3){
if(debug) printf("Enter input %d\n",guessnum + 1);

        send_data(COMMAND, CLEAR_SCREEN);
        print_lcd("INPUT PHASE");
        change_position(1, 0);
        memset(buf, 0, strlen(buf));
        if(debug) sprintf(buf, "Enter number %d\n", guessnum + 1);
        print_lcd(buf);

        guesses[guessnum] = press_blink(button, led1, led2);
        guessnum++;
}
if(debug) printf("Input completed. Guesses: %d %d %d\n", guesses[0], guesses[1], guesses[2]);

send_data(COMMAND, CLEAR_SCREEN);
print_lcd("Input completed.");
// END OF INPUT
blink(2, led1);

if(debug) printf("Checking answers...\n");

//ANSWER CHECKING
```

```
int c;
int exact = 0;
int approx = 0;
for(c = 0; c < 3; c++){
        if(guesses[c] == sequence[c]){
                exact++;
        }else{
                int d;
                for(d = c; d < 3; d++){
                        if(guesses[c] == sequence[d] && guesses[c] != sequence[c]){
                                approx++;
                        }
                }
        }
}
if(debug) printf("Exact matches: %d  Approximate matches: %d\n", exact, approx);

send_data(COMMAND, CLEAR_SCREEN);
if(debug) sprintf(buf, "Exact matches: %d\n", exact);
print_lcd(buf);
memset(buf, 0, strlen(buf));

blink(exact, led2);

//BLINK ONCE
blink(1, led1);

change_position(1,0);
if(debug) sprintf(buf, "Part matches: %d\n", approx);
```

```
		print_lcd(buf);
		blink(approx, led2);

		send_data(COMMAND, CLEAR_SCREEN);
		if(exact == 3){
			print_lcd("YOU WIN! :)");
			if(debug) printf("Winner, Winner, Chicken Dinner\n");
			delay(3000);
			break;
		}

		blink(3, led1);
		print_lcd("NEXT ROUND.");
		delay(3000);
}

		reset_lcd();
		reset_gpio();

		exit(SUCCESS);
}

void blink(int times, int pin){
		int t;
		for(t = 0; t < times; t++){
			gpioset(pin);
			delay(700);
			gpioclear(pin);
			delay(700);
```

```
        }
}

int press_blink(int button, int led1, int led2){

        int presses = 0;
        int attempt = 0;

        // REPEATS if button was pressed more than 3 times.
        do {
                if(debug && (attempt != 0)){
                        printf("%d presses. Maximum of 3 allowed. Re-enter input again.\n", presses);
                } else if (debug){

                }
                presses = detect_presses(button);
                attempt++;
        } while(presses > 3);

        if(debug) printf("Button pressed %d times.\n",presses);

        blink(1, led1);

        //LED BLINKS
        blink(presses, led2);

        return presses;
}
int detect_presses(int button){
```

```
	gpiofen(button, 1);
	gpioeds_w(button, 1);

	int counter = 0;
	int i = 0;
	while(i<40){
	if(debug) printf("%d\n", i);
	if(gpioeds_r(button)==1){
		gpioeds_w(button, 1);
		counter++;
	}
	delay(100);
	i++;
	}
	if(debug) printf("Input received.\n");
	send_data(COMMAND, CLEAR_SCREEN);
	if(debug) print_lcd("Input received");
	return counter;
}

void print_lcd(char * s){
	send_string(s, strlen(s));
}

//============================ CTRL C HANDLER
================================================

void ctrlc_handler(int sig) {
	reset_lcd();
```

```
        reset_gpio();
        exit(FAILURE);
}

//=============================================================================================
//============================ LCD LIBRARY (+ DELAY FUNCTIONS) ====================================
//=============================================================================================
void set_4_pin_mode(){
        int i = 0;
        while(i < 3){
        //0011
        command_4bits(3);
        delay(35);
        i++;
        }
        //0010
        command_4bits(2);

        if(debug) printf("4x2 pin mode set\n");
}

void command_4bits(unsigned char command){
        gpioclear(lcd_pins->rs);
        write_to_data_pins(command);
        strobe();
}
```

```
void init_lcd(int mode){
	if(mode==4){
		lcd_pins->data[0] = DATA_5;
		lcd_pins->data[1] = DATA_6;
		lcd_pins->data[2] = DATA_7;
		lcd_pins->data[3] = DATA_8;
	} else if (mode == 8){
		lcd_pins->data[0] = DATA_1;
		lcd_pins->data[1] = DATA_2;
		lcd_pins->data[2] = DATA_3;
		lcd_pins->data[3] = DATA_4;
		lcd_pins->data[4] = DATA_5;
		lcd_pins->data[5] = DATA_6;
		lcd_pins->data[6] = DATA_7;
		lcd_pins->data[7] = DATA_8;
	}
	lcd_pins->rs = RS;
	lcd_pins->strb = STRB;
	lcd_pins->bit_mode = mode;
	lcd_pins->x = 0;
	lcd_pins->y = 0;

	int i = 0;
	while(i < mode){
		gpiomode_w(lcd_pins->data[i], OUTPUT);
		gpioclear(lcd_pins->data[i]);
		i++;
	}
	gpiomode_w(lcd_pins->rs, OUTPUT);
```

```
        gpioclear(lcd_pins->rs);
        gpiomode_w(lcd_pins->strb, OUTPUT);
        gpioclear(lcd_pins->strb);
}

void delayMicroseconds (unsigned int howLong)
{
  struct timespec sleeper ;
  unsigned int uSecs = howLong % 1000000 ;
  unsigned int wSecs = howLong / 1000000 ;

  /**/ if (howLong ==   0)
    return ;
#if 0
  else if (howLong  < 100)
    delayMicrosecondsHard (howLong) ;
#endif
  else
  {
    sleeper.tv_sec  = wSecs ;
    sleeper.tv_nsec = (long)(uSecs * 1000L) ;
    nanosleep (&sleeper, NULL) ;
  }
}

// milliseconds
void delay (unsigned int howLong)
{
  struct timespec sleeper, dummy ;
```

```
  sleeper.tv_sec  = (time_t)(howLong / 1000) ;
  sleeper.tv_nsec = (long)(howLong % 1000) * 1000000 ;

  nanosleep (&sleeper, &dummy) ;
}

// high to low
void strobe(){
        gpioset(lcd_pins->strb);
        delayMicroseconds(50);
        gpioclear(lcd_pins->strb);
        delayMicroseconds(50);
}

void send_data(int mode, unsigned char data){
        // 1 is for sending characters
        if(mode == CHARACTER){
                gpioset(lcd_pins->rs);
        // 0 is for sending commands
        } else if (mode == COMMAND) {
                gpioclear(lcd_pins->rs);

                if((data == CLEAR_SCREEN) || (data == RETURN_HOME)){
                        lcd_pins->x = 0;
                        lcd_pins->y = 0;
                }
        }
        if(lcd_pins->bit_mode == 4){
```

```
                write_to_data_pins((data >> 4) & 0x0F);
                strobe();
                write_to_data_pins(data & 0x0F);
                strobe();
        } else if(lcd_pins->bit_mode == 8){
                write_to_data_pins(data & 0xFF);
                strobe();
        }

        delay(30);
}

void write_to_data_pins(unsigned char data){
        int i = 0;
        while(i < lcd_pins->bit_mode){
                if((data >> i) & 1 == 1){
                        gpioset(lcd_pins->data[i]);
                } else{
                        gpioclear(lcd_pins->data[i]);
                }
                i++;
        }
}

void send_string(char str[], int str_length){
        int i=0;
        while(i<str_length){
                send_data(CHARACTER, str[i]);
                i++;
```

```
                position();
        }
}

void position(){
        lcd_pins->x += 1;
        // 0-15 column numbering
        if(lcd_pins->x > 15){
                lcd_pins->x = 0;
                lcd_pins->y += 1;
                send_data(COMMAND, DDRAM | 0x40);
        }
        // 0-1 row numbering
        if(lcd_pins->y > 1){
                send_data(COMMAND, CLEAR_SCREEN);
                lcd_pins->y = 0;
                lcd_pins->x = 0;
        }
}

// row - 0 or 1
// col - 0 to (including) 15
void change_position(int row, int col){
        if((row==0 || row==1) && (col>=0 && col<=15)){
                if(row==1){
                        send_data(COMMAND, DDRAM | (0x40 + col));
                } else{
                        send_data(COMMAND, DDRAM | col);
                }
```

```
	lcd_pins->x = col;
	lcd_pins->y = row;
	} else{
		exit(FAILURE);
	}
}

void reset_lcd(){
	send_data(COMMAND, CLEAR_SCREEN);
}

//================================================================================
========
//============================== GPIO LIBRARY
================================================

//================================================================================
========

void init_gpio(){
	int fd;
	gpiobase = 0x3F200000;
	if ((fd = open ("/dev/mem", O_RDWR | O_SYNC | O_CLOEXEC) ) < 0){
		if(debug) printf("less than 0\n");
	exit(FAILURE);
	}

	gpio = (uint32_t *)mmap(0, BLOCK_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
gpiobase) ;
	if ((int32_t)gpio == -1){
		if(debug) printf("equals -1\n");
```

```
        exit(FAILURE);
        }
}

// 000 - input
// 001 - output
// function takes input 0 to 7
void gpiomode_w(int pin, int mode){
        if(mode>=0 && mode<=7 && pin>=0 && pin<=53){
                int select = pin/10;
                int bits = (pin - (select*10)) * 3;

                // should be universal, since (x | 0000) should be equal to x
                *(gpio+select) = (*(gpio+select) & ~(7 << bits)) | (mode << bits);
        } else {
                exit(FAILURE);
        }
}

int gpiomode_r(int pin){
        if(pin>=0 && pin<=53){
                int select = pin/10;
                int bits = (pin - (select*10)) * 3;

                return (*(gpio+select) << bits) & (7 << 0);
        } else{
                return -1;
        }
}
```

```
// takes input 1 or 0
void gpioset(int pin){
	if(pin>=0 && pin<=53){
		// GPSET1
		// 32 to 53
		if(pin>31){
		pin -= 32;
		*(gpio+8) = 1 << pin;
		// GPSET0
		// 0 to 31
		} else{
		*(gpio+7) = 1 << pin;
		}
	} else {
		exit(FAILURE);
	}
}

// takes input 1 or 0
void gpioclear(int pin){
	if(pin>=0 && pin<=53){
		// GPCLR1
		// 32 to 53
		if(pin>31){
		pin -= 32;
		*(gpio+11) = 1 << pin;
		// GPCLR0
		// 0 to 31
```

```
                } else{
                *(gpio+10) = 1 << pin;
                }
        } else {
                exit(FAILURE);
        }
}

// takes input 1 or 0
int gpiolev(int pin){
        if(pin>=0 && pin<=53){
                // GPLEV1
                // 32 to 53
                if(pin>31){
                pin -= 32;
                return (*(gpio+14) >> pin) & (1);
                // GPLEV0
                // 0 to 31
                } else{
                return (*(gpio+13) >> pin) & (1);
                }
        } else {
                exit(FAILURE);
        }
}

int gpioeds_r(int pin){
        if(pin>=0 && pin<=53){
                // GPEDS1
```

```
                // 32 to 53
                if(pin>31){
                pin -= 32;
                return (*(gpio+17) >> pin) & (1);
                // GPEDS0
                // 0 to 31
                } else{
                return (*(gpio+16) >> pin) & (1);
                }
        } else {
                exit(FAILURE);
        }
}

void gpioeds_w(int pin, unsigned int bit){
        if((bit==1 || bit==0) && pin>=0 && pin<=53){
                // GPEDS1
                // 32 to 53
                if(pin>31){
                pin -= 32;
                *(gpio+17) = (*(gpio+17) & ~(1 << pin)) | (bit << pin);
                // GPEDS0
                // 0 to 31
                } else{
                *(gpio+16) = (*(gpio+16) & ~(1 << pin)) | (bit << pin);
                }
        } else {
                exit(FAILURE);
        }
```

```
}

void gpioren(int pin, unsigned int bit){
if((bit==1 || bit==0) && pin>=0 && pin<=53){
                        // GPREN1
                        // 32 to 53
                        if(pin>31){
                        pin -= 32;
                        *(gpio+20) = (*(gpio+20) & ~(1 << pin)) | (bit << pin);
                        // GPREN0
                        // 0 to 31
                        } else{
                        *(gpio+19) = (*(gpio+19) & ~(1 << pin)) | (bit << pin);
                        }
        } else {
                        exit(FAILURE);
        }
}

// falling edge
// when LEV changes from 1 to 0
void gpiofen(int pin, unsigned int bit){
if((bit==1 || bit==0) && pin>=0 && pin<=53){
                        // GPRFEN1
                        // 32 to 53
                        if(pin>31){
                        pin -= 32;
                        *(gpio+23) = (*(gpio+23) & ~(1 << pin)) | (bit << pin);
                        // GPFEN0
```

```
                // 0 to 31
                } else{
                *(gpio+22) = (*(gpio+22) & ~(1 << pin)) | (bit << pin);
                }
        } else {
                exit(FAILURE);
        }
}

void reset_gpio(){
        int gpio_pins[] = {2, 3, 4, 14, 15, 17, 18, 27, 22, 23, 24, 10, 9, 11, 8, 7, 25, 5, 6, 12, 13, 19, 16, 26,
20, 21};
        int i = 0;
        int size = sizeof(gpio_pins)/sizeof(gpio_pins[0]);
        while(i < size){
                gpiomode_w(gpio_pins[i], INPUT);
                gpiofen(gpio_pins[i], 0);
                gpioren(gpio_pins[i], 0);
                gpioclear(gpio_pins[i]);
                i++;
        }
}
```

1.2 Inline Assembly component.

```
#include <stdio.h>
#include <stdarg.h>
#include <stdint.h>
#include <stdlib.h>
#include <ctype.h>
#include <poll.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/ioctl.h>
#include <unistd.h>
#define SUCCESS 0
#define FAILURE 1

#define PAGE_SIZE               (4*1024)
#define BLOCK_SIZE              (4*1024)
static volatile unsigned int gpiobase ;
uint32_t *gpio ;

void init_gpio(){
        int fd;
```

```
	gpiobase = 0x3F200000;
	if ((fd = open ("/dev/mem", O_RDWR | O_SYNC | O_CLOEXEC) ) < 0){
		printf("less than 0\n");
	exit(FAILURE);
	}

	gpio = (uint32_t *)mmap(0, BLOCK_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
gpiobase) ;
	if ((int32_t)gpio == -1){
		printf("equals -1\n");
	exit(0);
	}
}



gpiosetASM(int pin){
	if(pin > 31){
		printf("Please use a on board pin.\n");
		exit(0);
	}

	int temp;
	int shift = 1<<pin;
	uint32_t * base = gpio;
	asm volatile(
	"LDR	R3, %[gpio]\n"
	"ADD	R3, R3, #0x1C\n"
	"MOV	R2, %[shift]\n"
	"STR	R2, [R3, #0]\n"
```

```
        "MOV   %[temp], R3\n"
        :[temp] "=r" (temp)
        :[gpio] "m" (base)
        ,[shift] "r" (shift)
        );

}

gpioclearASM(int pin){
        if(pin > 31){
                printf("Please use a on board pin.\n");
                exit(0);
        }

        int temp;
        int shift = 1<<pin;
        uint32_t * base = gpio;
        asm volatile(
        "LDR     R3, %[gpio]\n"
        "ADD     R3, R3, #0x28\n"
        "MOV   R2, %[shift]\n"
        "STR     R2, [R3, #0]\n"

        "MOV   %[temp], R3\n"
        :[temp] "=r" (temp)
        :[gpio] "m" (gpio)
        ,[shift] "r" (shift)
        );
```

```
}

gpiomodeASM(int pin, int mode){
	if(pin > 31 || mode > 1){
		printf("ERROR: Invalid pin or mode\n");
		exit(0);
	}


	int firstnum = (pin/10) * 4;
	int bits = (pin - ((firstnum/4)*10) * 3);
	int bic = 0b111<<bits;
	int orr = 1<<bits;

	if(mode < 1){
		int temp;
		uint32_t * base = gpio;
		asm volatile(
		"LDR	R3, %[gpio]\n"
		"LDR	R2, %[gpio]\n"
		"ADD	R3, R3, %[firstnum]\n"
		"LDR	R2, [R2, #0]\n"
		"BIC	R2, R2, %[bic]\n"
		"STR	R2, [R3, #0]\n"

		"MOV	%[temp], %[firstnum]\n"
		:[temp] "=r" (temp)
```

```
        :[gpio] "m" (base)
        ,[firstnum] "r" (firstnum)
        ,[bic] "r" (bic)
        );
        printf("%d",temp);
}else{
        uint32_t * temp;
        uint32_t * base = gpio;
        asm volatile(
        "LDR    R3, %[gpio]\n"
        "LDR    R2, %[gpio]\n"
        "ADD    R3, R3, %[firstnum]\n"
        "LDR    R2, [R2, #0]\n"
        "BIC    R2, R2, %[bic]\n"
        "STR    R2, [R3, #0]\n"

        //SET OUTPUT
        "LDR    R3, %[gpio]\n"
        "LDR    R2, %[gpio]\n"
        "ADD    R3, R3, %[firstnum]\n"
        "ADD    R2, R2, %[firstnum]\n"
        "LDR    R2, [R2, #0]\n"
        "ORR    R2, R2, %[orr]\n"
        "STR    R2, [R3, #0]\n"

        "MOV   %[temp], R3\n"
        :[temp] "=r" (temp)
        :[gpio] "m" (base)
        ,[firstnum] "r" (firstnum)
```

```
                ,[orr] "r" (orr)
                ,[bic] "r" (bic)
                );

        }
}

void gpsel(uint32_t *gpio){
        //sel pin 19

        uint32_t * temp;
        asm volatile(
        "LDR    R3, %[gpio]\n"
        "LDR    R2, %[gpio]\n"
        "ADD    R3, R3, #4\n"
        "LDR    R2, [R2, #0]\n"
        "BIC    R2, R2, #0b111<<27\n"
        "STR    R2, [R3, #0]\n"

        //SET OUTPUT
        "LDR    R3, %[gpio]\n"
        "LDR    R2, %[gpio]\n"
        "ADD    R3, R3, #4\n"
        "ADD    R2, R2, #4\n"
        "LDR    R2, [R2, #0]\n"
        "ORR    R2, R2, #1<<27\n"
        "STR    R2, [R3, #0]\n"

        //GPSET
```

```
        "LDR     R3, %[gpio]\n"
        "ADD     R3, R3, #0x1C\n"
        "MOV    R4, #1\n"
        "MOV    R2, R4, LSL#19\n"
        "STR     R2, [R3, #0]\n"

        //GPCLEAR
        "LDR     R3, %[gpio]\n"
        "ADD     R3, R3, #0x28\n"
        "MOV    R4, #1\n"
        "MOV    R2, R4, LSL#19\n"
        "STR     R2, [R3, #0]\n"

        "MOV    %[temp], R3\n"
        :[temp] "=r" (temp)
        :[gpio] "m" (gpio)
        );
        printf("%p\n", temp);
}
int main(){
        init_gpio();
        gpiomodeASM(19,1);
        printf("%p\n", gpio);
        printf("GPIO Initialized.\n");
}
```

2. Explanation.

There are about 500 lines of code in our file. All of it is written by us, except for the delay and delay microseconds functions, which were copied from the lab.

The code could have been divided into 2 separate header files: gpio and lcd lib, but gpio is dependent on a static variable, and so is lcd. Since static variables are not allowed in header files, this means that we should add an extra parameter to all the functions in the file, which just seems very redundant. Same can be said about the lcd library. Considering that we must put all our code into this document, we decided not to bother and put it all into one file.

To be honest, an OOP language would have handled this much better.

Also, the code runs successfully on both pi 2 and 3. No changes were necessary to make it run on Pi 3 (the code that was used was pure C though).

2.1 Wiring.

In a nutshell, electricity is created by connecting positive and negative voltages together. This creates a force that makes electrons flow from negative to positive. The current will always try to find the path of least resistance. Having a circuit without any resistors creates a short circuit, which is basically unstable.

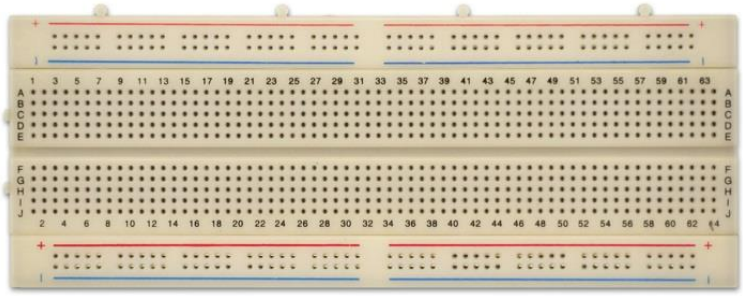
I wanted to include a paragraph on wiring, since a lot of mistakes were made that prevented the coursework from going further. Hopefully future students will get the heads up.

There are about 28 GPIO pins out of the total of 40 in Raspberry Pi 2. GPIO pins can either be set to 1 (3.3 V) or 0 (0 V). Every other pin is constant voltage – 0 V, 3.3 V, 5 V. One can make a circuit just with those constant voltage pins, and make their LCD or LED at least light up for example. That is very helpful if you want to test whether the circuit is functional or not, and I want to present some methods we used to do that.

In our case, there were two problems. First one was related to the breadboard. For quite some time, we were not able to make the button or LCD labs work. The problem was not the code, it was not even the wiring actually – it was inconsistency.

Problem 1.

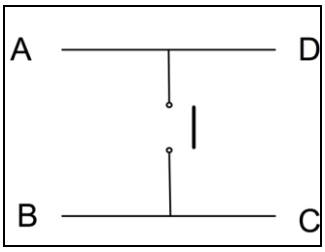
It is said that the plus and minus lines on each side of the breadboard are connected, but that is not quite true. It may be true for other breadboard designs, but not for ours. Our breadboard design has four different plus/minus blocks, rather than two on each side. There is actually a gap in the middle that indicates that.



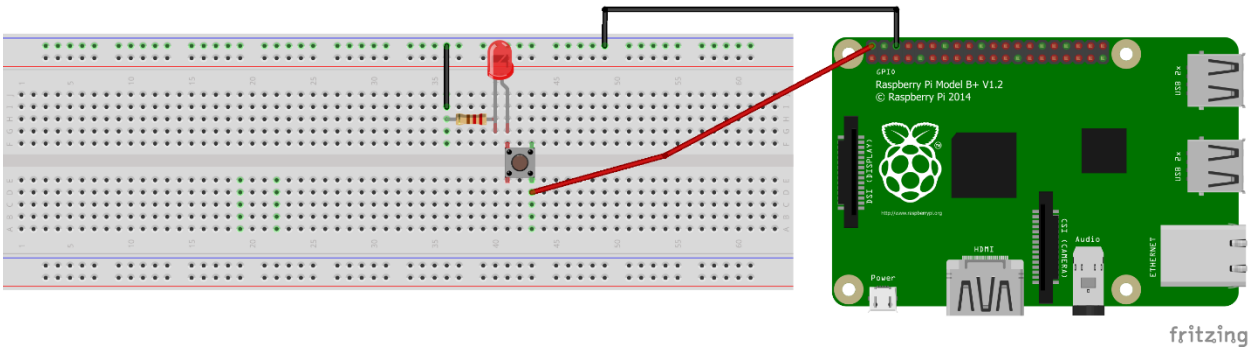
Our LCD was actually off the entire time. We thought that running the code would turn on the backlight, but actually all we needed was to connect four wires.

Problem 2.

The other problem was the button. There is no polarity on the button. A button, otherwise known as a switch, basically acts as a bridge and extends the range of the lines you connect them to, and when pressed, it also interconnects those two lines. The image below gives a visual representation.



In our case, the button was just not pressed in hard enough into the breadboard, but how does one test that? What I did is, I used an LED and a button.



Basically, this is an open circuit that should begin to function only when the button is pressed, so basically when the circuit is closed.

2.2 GPIO library.

When we first tried running the button lab on our raspberry pi's - it did not work, so I decided to write my own GPIO library. I thought that it was the code that was the problem (and there is actually a mistake in that lab), but in our case, it was the wiring.

In the end, having a library helped us out a lot though.

Here are the basic rules of GPIO.

One can set a mode for a pin, and there are 8 different modes altogether. The two main ones are input (0) and output (1). Everything else is considered special case.

If the mode is output, one can set the voltage of that pin. Using the set register, one sets the voltage to 3.3 V, and using the clear register, one sets the voltage to 0 V.

If the mode is input, the pin is used to read voltage. The maximum voltage a pin can take is only 3.3 V. Anything more than that kills the pin. One must be careful since there are 5 V pins on the raspberry pi. LEV register is used to read the voltage.

When the pin is in input, set and clear have no effect it on it, but you will get the corresponding result if you were to change the pin to output. On the web, they say that the input mode blocks out set and clear registers.

All the pins start in input mode. It is considered unsafe to leave pins in any other mode other than input after the program ends, and one has to reset the pins back to input after the program ends. Restarting the pi sets all pins back to input mode.

Safety.

I also want to talk about safety a bit. Like I said, when the program ends, you must set back the pins to input mode. The lab code provided does not do that. One can easily destroy the LCD, LED or even the pi itself, if he were to run a few of the provided labs subsequently. In fact, I even witnessed something like that happening to one of our peers – the LCD apparently short circuited and the brightness completely dimmed.

Also, in the button code, the final line of the code is supposed to do a clear, but register (gpio+7) belongs to GPSET0.

```
 }
// Clean up: write LOW
*(gpio + 7) = 1 << (23 & 31) ;
```

In our code, I implemented functions like reset_gpio() to deal with that, and also a ctrl+c handler, in case the program ends prematurely.

Interrupts.

That pretty much covers the basics of GPIO, but those are not the only registers. There is a total of 41 registers on a pi, and I want to focus on EDS, REN, FEN, HEN, LEN now, which are used for interrupts.

EDS stands for event detect status. While REN stands for rising edge, which indicates that LEV was changed from 0 to 1. FEN for falling edge, which is the opposite. HEN for high, LEN for low.

Why use them? Well, LEV just doesn't give precise results, while EDS does.

You can enable one or all four registers for a specific pin, and whenever the event is detected, EDS is set to 1. Then you should reset the EDS, so that you can detect the event again. What is strange about resetting is that you have to write 1 to EDS to reset it. 1 is used to show whether an event was detected or not, and writing 1 is used to reset EDS – very counterintuitive. Actually at first I could not make it work because of that. Found out about it only once I checked the BCM2835 manual.

REN is sort of bugged. It would sometimes detect an event even when it did not happen, which gave imprecise results. FEN managed to give exact number of button presses meanwhile, which is why we decided to use it.

The image below is taken from the book – *Modern Assembly Language Programming with ARM Processor by Larry D. Pyeatt.*

Offset	Name	Description	Size	R/W
00_{16}	GPFSEL0	GPIO Function Select 0	32	R/W
04_{16}	GPFSEL1	GPIO Function Select 1	32	R/W
08_{16}	GPFSEL2	GPIO Function Select 2	32	R/W
$0C_{16}$	GPFSEL3	GPIO Function Select 3	32	R/W
10_{16}	GPFSEL4	GPIO Function Select 4	32	R/W
14_{16}	GPFSEL5	GPIO Function Select 5	32	R/W
$1C_{16}$	GPSET0	GPIO Pin Output Set 0	32	W
20_{16}	GPSET1	GPIO Pin Output Set 1	32	W
28_{16}	GPCLR0	GPIO Pin Output Clear 0	32	W
$2C_{16}$	GPCLR1	GPIO Pin Output Clear 1	32	W
34_{16}	GPLEV0	GPIO Pin Level 0	32	R
38_{16}	GPLEV1	GPIO Pin Level 1	32	R
40_{16}	GPEDS0	GPIO Pin Event Detect Status 0	32	R/W
44_{16}	GPEDS1	GPIO Pin Event Detect Status 1	32	R/W
$4C_{16}$	GPREN0	GPIO Pin Rising Edge Detect Enable 0	32	R/W
50_{16}	GPREN1	GPIO Pin Rising Edge Detect Enable 1	32	R/W
58_{16}	GPFEN0	GPIO Pin Falling Edge Detect Enable 0	32	R/W
$5C_{16}$	GPFEN1	GPIO Pin Falling Edge Detect Enable 1	32	R/W
64_{16}	GPHEN0	GPIO Pin High Detect Enable 0	32	R/W
68_{16}	GPHEN1	GPIO Pin High Detect Enable 1	32	R/W
70_{16}	GPLEN0	GPIO Pin Low Detect Enable 0	32	R/W
74_{16}	GPLEN1	GPIO Pin Low Detect Enable 1	32	R/W
$7C_{16}$	GPAREN0	GPIO Pin Async. Rising Edge Detect 0	32	R/W
80_{16}	GPAREN1	GPIO Pin Async. Rising Edge Detect 1	32	R/W
88_{16}	GPAFEN0	GPIO Pin Async. Falling Edge Detect 0	32	R/W
$8C_{16}$	GPAFEN1	GPIO Pin Async. Falling Edge Detect 1	32	R/W
94_{16}	GPPUD	GPIO Pin Pull-up/down Enable	32	R/W
98_{16}	GPPUDCLK0	GPIO Pin Pull-up/down Enable Clock 0	32	R/W
$9C_{16}$	GPPUDCLK1	GPIO Pin Pull-up/down Enable Clock 1	32	R/W

2.3 LCD driver.

Apparently, a library that is used to communicate with a peripheral is called as a driver, hence the title.

We already had functioning code for the LCD that was provided in the lab, but personally it was hard for me to make out anything of it, and for that reason, I decided to create my own code for the LCD.

But first I had to understand basics behind the LCD. It was not easy. There was a scarce amount of material on that. Barely any code written in C, and everything was so abstract. At the time, I did not even understand the purpose of RS or STRB pins, but in the end, I found something to grab onto.

The links that helped in the making of the code.

The link below features a diagram that been a great help initially in understanding what RS does.

http://www.firmcodes.com/microcontrollers/8051-3/interfacing-lcd-with-8051/lcd-commands-and-understanding-of-lcd/

commands:

https://mil.ufl.edu/3744/docs/lcdmanual/commands.html

http://www.dinceraydin.com/lcd/commands.htm

initialization with 4 pins.

http://www.taoli.ece.ufl.edu/teaching/4744/labs/lab7/LCD_V1.pdf

line addressing:

http://web.alfredstate.edu/weimandn/lcd/lcd_addressing/lcd_addressing_index.html

Finally, the manual for HD44780 (pages 40-46) and the LCD lab code itself were a great help. In fact, I think that my code is basically the recreation of the LCD lab code, with slight adjustments.

To sum it up.

RS stands for register select. There are 2 registers on the LCD – data and command registers. If RS is 0, command register is chosen, otherwise the data register.

Commands are used to manipulate the LCD, while data are used to print characters.

Strobe (STRB) otherwise known as Enable (E) is basically the trigger for the LCD, so that it would read the data pins and do whatever it should do. To trigger, one must set strobe to 1 and then back to 0. Falling edge is detected and LCD starts reading the data pins.

R/W stands for Read/Write. When it is 0, it is write, otherwise it is read. Since our RW pin is connected to ground, which is 0 V, then the LCD is in write mode the entire time. By the way, there is only two read operations: one is for reading characters off the screen, and the other one is getting the current position on the LCD. Both can be recreated with a few variables in the code.

That is pretty much it. There are 8 data pins, and the LCD may either run in 8 or 4-bit mode. In 4-bit mode, data pins 5,6,7,8 are used. Also, one must run a certain pattern to use the LCD in 4-bit mode (for the pattern, look at the link above). Initially, the LCD only sees 4 bits, and it should be configured to do all operations in 4x2 chunks.

In our case, our LCD can be run in both 4 bit and 8 bit modes, courtesy of writing code from scratch.

There are also functions position() and changePosition() in the code. Position() is for keeping track of row, col and automatically shifting to the next line or the start when it reaches the end of the line. ChangePosition() is for explicitly changing the position to whatever column or line you want.

2.4 Inline Assembly.

The inline assembly code is incomplete and inferior to the C code. We had to hardcode in the values (pins) to make it properly work. We use about 9-13 pins (dependent on the bit mode) in our coursework.

Putting assembly in C.

So essentially using assembly in a C program can be done with a simple function called asm(), or asm volatile() in our case which should be used in situations where you are dealing with risky values such as conducting bit operations on memory addresses or using non negative numbers. Using asm, you can just put regular arm assembly instructions as is and manipulate registers. With extended asm, using input and output operands, we can pass arguments into these assembly instructions as well as get a result, which we can then use in the regular C code.

Our code.

We included our functions made with inline assemblies which are the basic gpio set, clear and setting the gpio mode. The way this all works is that we use the normal C to map the gpio virtual memory and this sets the variable gpio to the base address of the mapped memory. We then use this base address when we conduct the bit shift operations by loading it into registers and conducting the shifts and indexing using assembly. The basic idea behind the use of gpio pins and the registers follows that of the regular C methods but it's just done in assembly. For example, you would just load a base address into a register, add to it in order to move to the appropriate register, take the contents of that register and duplicate it, conduct a bit clear (BIC) on that register to make a mask and store that back into the register as well as shifting the bits. We also have a gpsel function which basically contains a run through of setting a hardcoded pin (19) as output and setting that pin on and then clearing it. To experiment with this, the user can remove the commented part which clears the pin and the led connected to pin 19 should remain on. They can then put back the code for clearing the pin to turn it off. The gpio set and clear functions take parameters and function correctly. The only problem is doing the gpio mode which I believe the problem was due to passing an argument into the assembly ADD function which caused errors. Many solutions were tried and tested such as using MOV to send the argument to a register and using that register to ADD but it still did not function.

Final note.

Overall inline assembly proved to be challenging in a sense that the documentation online was not very user friendly. The other challenges faced included finding a way to properly mmap the gpio addresses using pure assembly code. Also, it made things unnecessarily complicated as one could've achieved these functionalities with fewer lines and more easily with a library or even using pure C code. However, it did provide a better understanding of the manipulation of memory and registers.

2.5 The game.

Mastermind is a simple game. We generate three random numbers, then the player will have to guess them. The range of the numbers is from 1 to 3. After the player gives his input three times, we tell him the approximate and the exact amount of right he managed to get. The approximate is when the number is correct, but it is not in the right position. After that, if he has not already won, then he can use deduction or the probability theory to get the correct answer.

You really only need a while true loop with a break statement to make it.

Random number generator.

For our random generator, we used the UNIX time for the seed – srand(time(NULL)).

For limiting the range, we used the modulo – (rand() % 3) +1.

rand() % 3 will give numbers from range 0 to 2. Plus 1 is there to set a higher range (1 to 3). We are not quite sure whether 0 should be allowed in the game or not, and for that reason we did not use rand() % 4.

Conclusion.

Everything else is very straight forward. There is a certain pattern LEDs must do throughout the game (as listed in the CW paper), and the code does that.

There is also a debug mode. Just add “d” as a argv[1] parameter, and it will turn on print statements, that will show up in the terminal. They are basically a way to see what is happening behind the scenes, without looking at the LCD or the LEDs.

3. Work distribution.

Daniyar – wiring, GPIO library, LCD library, report.

Chester – inline assembly, the game and the main().

Abdullah – nothing.

We can provide the work Abdullah has contributed as proof. There is nothing useful one could extract from that.