Electronic Systems - Report

Diogo Faria Correia
MIEEC
Faculdade de Engenharia da Univeridade do Porto
Porto, Portugal
up201504726@fe.up.pt

Pedro Miguel Coutinho Augusto
MIEEC
Faculdade de Engenharia da Univeridade do Porto
Porto, Portugal
up201503495@fe.up.pt

I. Introduction

This report addresses the three lab projects developed during the course SELE - Electronic Systems (EEC0152): *Asynchronous Communication System*, *Controlling a PIC32MX320F128H using JTAG* and *Controlling a SPI device using ATmega328P*.
In the first project a set of *ATmega328P* micro-controllers were used to communicate with each other using the built-in *UART* port on a Master/Slave protocol paradigm.
In the second project the same micro-controller was used to control another one, this time a PIC32MX320F128H, utilizing its JTAG port.
For the last project the *ATmega328P* was used to control a device using SPI or I2C. The chosen device was the *PCD8544* screen driver. This module uses SPI as its only interface.

II. Async9 - Asynchronous Communication System

This system follows the Master/Slave model in which there is one device (the master) that controls all the other (the slaves). Both types of devices are based in the ATmega328P MCU and MAX485 transceivers. The addresses of each node are defined via 4 dip switches (4 bit addresses) that select the firmware version (master or slave). The frames of information have 9 bits where the 9th is used to differentiate the address and data frames. The goal was to use the asynchronous communication system to blink each LED of the slave nodes by pressing the pressure buttons connected to the master node. The work is split in two main work areas:

A. Board Development

This section explains the design of the schematic and PCB for the master/slave Hardware nodes.

1) Schematic: Beside the needed resistors, capacitors and 16MHz crystal, the only ICs used are the microcontroller **ATMega328p** and the **MAX485 transceiver**. Only the master needs the extra 2 buttons (control buttons) to blink the LEDs of the slaves but as the PCB is unique for both master/slave the schematic has these 2 buttons as well as the usual reset button. Only the reset button has the pull-up resistor. The control buttons and the DIP switches don't have pull-up resistors because the pull-up effect (logic 0 @ Vcc) can be set while developing the firmware. The female headers are needed to add the input pins (Vcc and Gnd) and the output pins (Tx pins for both slaves).

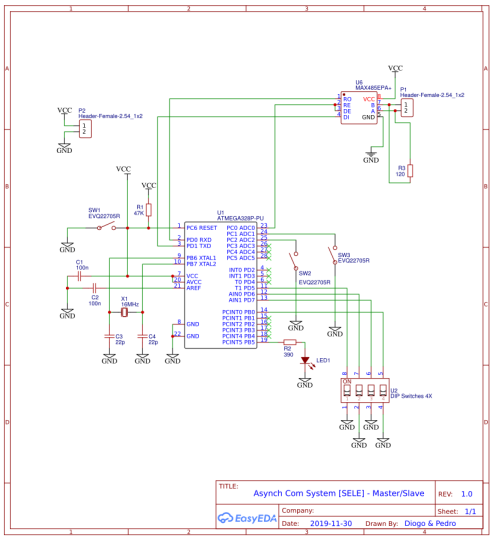

Fig. 1: Node schematic.

2) PCB: Following the schematic design, came the PCB design. To avoid track crossing and positioning problems, the top layer and the bottom layers were used. The initial idea was to use the bottom layer as the Ground plane but as all the components have through-hole packages, this was not possible. Instead, all the components are placed in the top layer with the signal nets except for the ground net that is placed in the bottom layer.

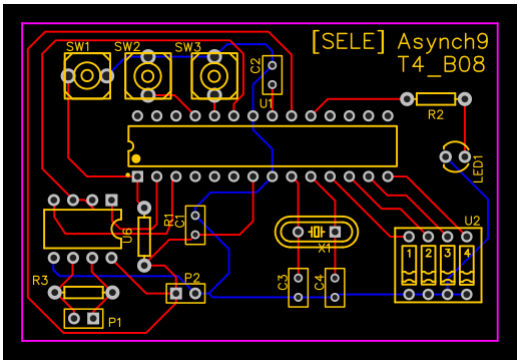

Fig. 2: Node PCB.

As there are not RF components in the circuit, strict constraints regarding track length or width weren't applied (width wasn't altered and was left as default - 10 mil 0.26mm).

B. Async9 Communication Protocol

1) Setup: As said before, Async9 communication system follows the Master/Slave paradigm. A 4-bit address is used in order to identify the device as a Master or a Slave. The chosen addresses must respect the following table:

Type	Address
Master	0xF
Slave	0x0 - 0xE

The address is configured by a set of 4 physical switches and must be configured before the device is plugged in.

```
uint8_t readADDR(){
  int addr0 = digitalRead(ADDR0_PIN);
  int addr1 = digitalRead(ADDR1_PIN);
  int addr2 = digitalRead(ADDR2_PIN);
  int addr3 = digitalRead(ADDR3_PIN);
  return addr0+addr1*2+addr2*4+addr3*8;
}
```

To communicate between relatively long distances, Async9 uses a differential signalling method called TIA-485. The transceiver needs to be configured into reception/transmission mode.

```
if(myADDR == MASTER_ADDR){
    digitalWrite(WREN_PIN, HIGH);
} else digitalWrite(WREN_PIN, LOW);
```

In the ATmega328P microcontroller the USART communication protocol must be configured too. The baud rate and the number of bits inside the datagram (in this case 9 bits) must be configured. This can be achieved by setting the values of UBRR0 and UCSZ0 flags. The UBRR0 code is obtained by using the following formula:

$$UBRR0 = \frac{f_{osc}}{16 * BAUDRATE} - 1 \qquad (1)$$

The UCSZ0 flag's value is available on Table 19-7 of the ATmega328P's datasheet.

UCSZ02	UCSZ02	UCSZ02	Char size
0	0	0	5-bits
0	0	1	6-bits
0	1	0	7-bits
0	1	1	8-bits
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bits

To configure the number of bits on a datagram we need to set the UCSR0 flag composed by 3 bits. The first two most significant bits are available on USART Control and Status Register C and the least significant bit on USART Control and Status Register B.

```
long ubrr = FOSC/(16*BAUD)-1;
UBRR0H = (unsigned char) (ubrr>>8);
UBRR0L = (unsigned char) ubrr;
UCSR0C |= (3 << UCSZ01);
UCSR0B |= (1 << UCSZ02);
```

Other FLAGs that need to be set on ATmega38P are the TXEN0 (in case of being a master) or RXEN0 and MPCM0 (in case of being a slave). TXEN0 and RXEN0 are used to enable the serial port in transmission and reception mode. The MPCM0 is used to set the microcontroller in Multi-processor mode, in which all the incoming frames that do not contain address information are ignored. This FLAG will be set to one until the device is called by its master, after that it will be set to zero until the master decides to call another device.

```
if(myADDR == MASTER_ADDR){
    //master
    UCSR0B |= (1 << TXEN0);
} else {
    //slave
    UCSR0A = (1 << MPCM0);
    UCSR0B |= (1 << RXEN0);
}
```

2) Loop: The loop is relatively simple. After setup, the immediate action for a node to take is to check its address

```
if(myADDR == MASTER_ADDR){
  ...
```

if it is the Master node it will update the state of the buttons and read the most recent state

```
lastBUT1 = nowBUT1;
lastBUT2 = nowBUT2;
nowBUT1 = digitalRead(BUT1_PIN);
nowBUT2 = digitalRead(BUT2_PIN);
...
```

4 different states may occur: Button that triggers communication with the slave1 was pressed

```
(lastBUT1 == 1) && (nowBUT1 == 0)
```

Button that triggers communication with the slave1 was released

```
(lastBUT1 == 0) && (nowBUT1 == 1)
```

Button that triggers communication with the slave2 was pressed

```
(lastBUT2 == 1) && (nowBUT2 == 0)
```

Button that triggers communication with the slave2 was released

```
(lastBUT2 == 0) && (nowBUT2 == 1)
```

For each of these 4 situations, if the previous stored address is different from the one of the recently pressed button, the previous stored address will be updated with the address of the

new slave associated with the button pressed. This assures that an address frame is only sent if there is another slave being addressed (by pressing the other button). The same is done if the button is being released - allows to switch one of the LEDs off if the two LEDS are ON at the time of the release.

```
if((lastBUTx == 1) && (nowBUTx == 0)){
    //Buttonx was pressed
    if(lstADDR != SLAVEx_ADDR){
    //Update address
    lstADDR = SLAVEx_ADDR;
    send_addr(SLAVEx_ADDR);
}
send_data(HIGH);
```

In the example above, we can see that if the button has been pressed, a logic 1 will be sent as a data frame. If released, a logic 0 will be sent as a data frame. These were the actions taken in case the node is a Master. If in the beginning of the loop the node confirms it is a slave, it will check for frames in the communication line and most importantly verify what type of frame it is. If it detects an address frame with its address, it will clear the MPCM0 bit in UCSR0A otherwise will wait for the next address frame directed to it and keep the MPCM0 setting. After receiving all data frames directed to it, if a new address frame arrives, it will set the MPCM0 bit and wait for a new address frame from master.

```
bit9 = get_data(&data);
if(bit9 == 1){
    if (myADDR == data){
        UCSR0A &= ~(1<<MPCM0);
    } else if((myADDR != data)){
        UCSR0A |= (1<<MPCM0);
    }
    lstADDR = data;
}
```

If a data frame is received, it will turn the LED ON or OFF accordingly:

```
if((bit9 == 0) && (data == HIGH))
   digitalWrite(LED_PIN, HIGH);
else if((bit9 == 0) && (data == LOW))
digitalWrite(LED_PIN, LOW);
```

III. Controlling a PIC32MX320F128H using JTAG

A. Using ATmega328P as a JTAG Programmer for the PIC32

The implementation of the JTAG Programmer for the PIC32MX320F128H is based on a main program which calls functions from the library **jtag.h**.

This library may serve as a basic JTAG Programmer implementation and can be used to facilitate the communication between the ATmega328P and other IEEE 1149.11 compliant devices.

- **TAP(TMS, TDI) -** Changes to the next state of the TAP Controller's state machine according to the provided TMS and TDI values. Also returns the TDO value.
- **resetTAP() -** Resets the TAP Controller's State Machine.

To interact specifically with the PIC32MX320D128H, the following functions act as a middleware that executes some tasks on the PIC32MX320D128H through IEEE 1149.11 TAP.

- **sir(opcode) -** Sets the PIC32MX320D128H's Instruction Register with the provided opcode.
- **sdr(pin, input, output, control) -** Sets the values on the input, output and control cells of the boundary scan.
- **sdrTDO(jtagPin) -** Returns the value present on **jtagPin** cell position.
- **registerTDO() -** Returns the values present on the the boundary scan cells.

B. Examples

1) Getting the PIC32's IDCODE: In order to obtain the IDCODE, according with the PIC32MX320D128H.bsdl document, the instruction *00001* has to be set on the instruction register. After that, the value stored on the IDCODE register will be available on TDO's cell.

```
sir(1);
idcode = registerTDO();
```

2) Controlling the ChipKit's Uno32 LED5: To control the LED LD5 of the ChipKit board connected to the pin 43 of the ChipKit Uno32 and pin 68 of the PIC32, one needs to change the value of the 18th output cell on the boundary scan. For that, the boundary scan was set in the **External Test** mode using the **00110** opcode.

```
//Turning on the LED:
sir(6);
sdr(18,0,1,1);

//Turning off the LED:
sir(6);
sdr(18,0,0,1);
```

3) Getting the state of a button: In order to obtain the value of the button connected to the pin 29 of the ChipKit Uno32 and pin 63 of the PIC32, one needs to check the 3rd cell of the boundary scan. For that, the boundary scan was set on **sample and preload** mode using the **00010** opcode.

```
sir(2);
statebtn = sdrTDO(3);
```

IV. Controlling a SPI device using ATmega328P

A. SPI - Serial Peripheral Interface

This communication protocol follows a Master/Slave paradigm and uses four different signals to enable communication between the Master device and its Slaves: *MISO*, *MOSI*, *SCLK* and *SS*.

a) : The *MISO*, Master Input Slave Output, and the *MOSI*, Master Output Slave Input, are used to transfer data bytes between master-slaves and slaves-master respectively. The *SCLK*, Serial Clock, and the *SS*, Slave Selector, are used to synchronize communication between devices. The SS signal is active LOW and should be repeated for each new slave or not, accordingly with the chosen topology. Due to the nature of our project, the discussion between different topologies can and will be ignored on this report as there is only one slave.

b) : Fortunately the ATmega328P has a SPI module built-in. To use the built-in SPI module it was necessary to activate it using the reserved microcontroller register. This process is executed when the SPInit() function is called.

```
void SPInit(){
    DDRB |= (1 << SS) | (1 << MOSI) | (1 << SCK);
    SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR0);
    SPSR = (1 << SPI2X);
}
```

c) : There are four SPI data modes, each one a combination of SCK phase and polarity. No mode was specified on the SPInit() function therefore the default one was selected. On the default mode, mode 0, the data signal is sampled during its rising edge and set up during the falling edge.

d) : The clock signal must be set at the lowest frequency in the range of all the slave devices using the same interface (in this case, there is only one device). According to the PCD8544 data-sheet, figure 3, the SLCK frequency can range from 0MHz to 4MHz. To ensure the good functioning of the system the 2MHz frequency was chosen.

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	4 MHz
0	0	1	1 MHz
1	0	1	2 MHz

TABLE I: SPCR register bit values for frequencies up to 4MHz

e) : To send commands to the slave, in this case there is only one - PCD8544 Nokia 5110 LCD controller, one simply has to store the data on the SPDR register and wait for it to be shifted out to the slave.

```
void SPISend(uint8_t payload){
    SPDR = payload;
    while(!(SPSR & (1 << SPIF)));
}
```

B. PCD8544 - The LCD Driver

The *PCD8544* driver chip supports an LCD display with 48 lines per 84 columns. Regardless of using SPI, the PCD8544 needs two more signals: a reset signal and a mode selector (command/data).

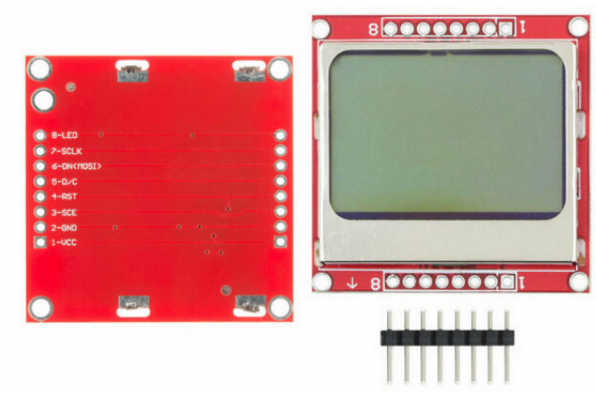

Fig. 3: SPI device - PCD8544 LCD driver.

a) : Immediately following power-on, the contents of all internal registers and of the RAM are undefined - a Reset signal must be applied before starting its usage.

b) : The mode select signal (DC) allows to choose between the command/addressing mode (DC = 0) or the data mode (DC = 1). The command/addressing mode allows to chose between horizontal or vertical addressing as well as choose the instruction set to be used (NORMAL mode was used in this application).

```
void PCD8544SendCommand(uint8_t command){
digitalWrite(DC, LOW);
SPISend(command);
}
```

The data mode simply writes data to be displayed on the LCD screen.

```
void PCD8544SendData(uint8_t data){
digitalWrite(DC, HIGH);
SPISend(data);
}
```

C. Displaying data onscreen

Making use of the function PCD8544SendData(), it is very easy to display data on screen.

STEP	SERIAL BUS BYTE									DISPLAY	OPERATION
	D/C̄	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
1	1	0	0	0	1	1	1	1	1		data write Y and X are initialized to 0 by default, so they are not set here

Fig. 4: Horizontal addressing example.

Each time the function is called, it will put the 8 bit array onscreen in the position next to the previous position. If the vertical addressing mode is chosen, *PCD8544SendCommand(0b00100010)*,the data will be placed in the next 8 pixels below the current position, if the horizontal addressing is chosen instead, *PCD8544SendCommand(0b00100000)*, the data will be placed

in the next 8 pixels to the right of the current position. In figure 4, it is shown how to display a 5 bit column, from the max data byte.

D. Defuse the Bomb minigame

To make use of the SPI enabled LCD, a small application was developed - *Defuse the Bomb*, the minigame.

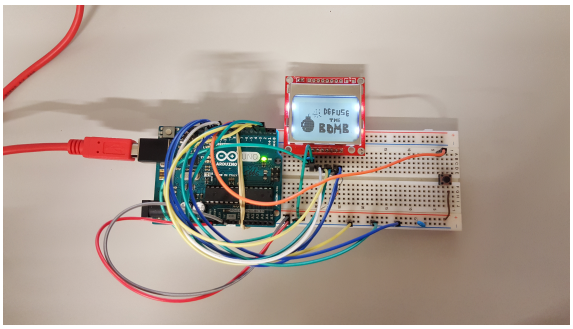

Fig. 5: *Defuse the Bomb* breadboard setup.

1) Game concept: The game simulates a life and death scenario where the player has to disarm a bomb by cutting (or in this case disconnecting) one of the three wires available. When powered on, the player is greeted with 3 screens:

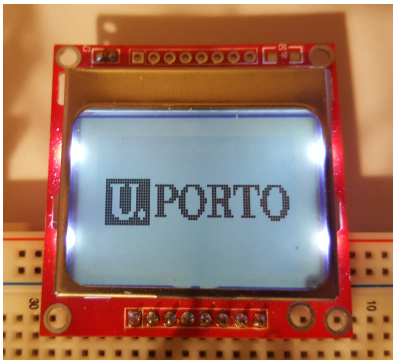

Fig. 6: Uporto game screen.

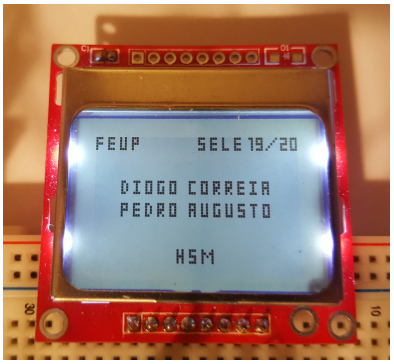

Fig. 7: About game screen.

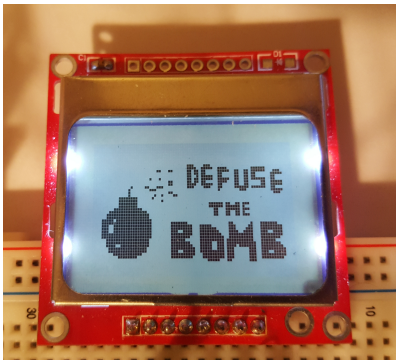

Fig. 8: Game title screen.

The type of image displayed onscreen dictates the addressing mode utilized, e.g, the *About* screen in figure 7 is displayed with the horizontal addressing mode as the alphanumeric characters shown were implemented utilizing this type of addressing.

In a first approach all bitmap images were stored on arrays. This method was deprecated due to the increasing memory usage. Instead, only the set of bytes with useful information were stored in arrays and the remaining ones were substituted with for-cycles.

```
zByte(172);
for (int i=0; i < 187; i++){
    PCD8544SendData(uporto[i]);
    delay(5);
}
```

The game's logic is simple and is based around the following components/concepts:

- **Countdown timer-** After the three loading screens, the player is greeted with a decreasing countdown of 30 seconds, figure 9. When the timer finishes, reaches time 0s, the player loses the game and an explosion screen appears, figure 10.

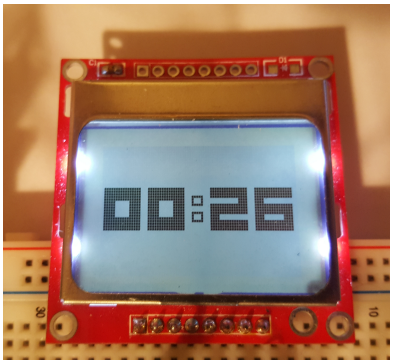

Fig. 9: Countdown screen.

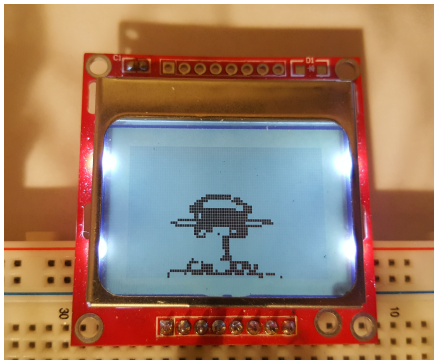

Fig. 10: Explosion screen.

Fig. 12: Win screen.

- **Defuse Wires -** To prevent the incoming explosion, the player has 3 wires to choose from (figure 11):

  Bomb disarmament wire - If the player disconnects the bomb disarmament wire, the player is victorious and the Victory screen is displayed, figure 12.

  Two Bomb trigger wires - If the player disconnects one of the bomb trigger wires, the counter is decreased and the player has to make a choice between the remaining two wires. If the following disconnected wire happens to be the other bomb trigger wire it is game over - the explosion screen appears, figure 10.
  Between games, the wires functions randomly exchange.

- **Reset Button -** To reset the game, the player can simply press the button on the breadboard. This will reset the counter to 30s whatever the game state may be.

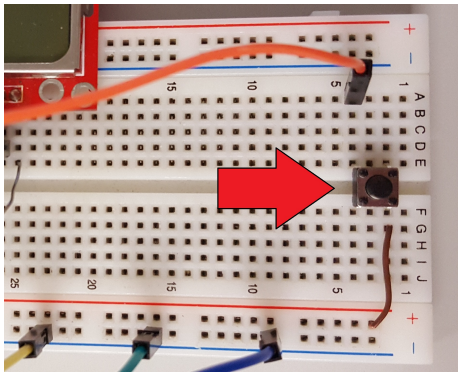

Fig. 13: Reset button.

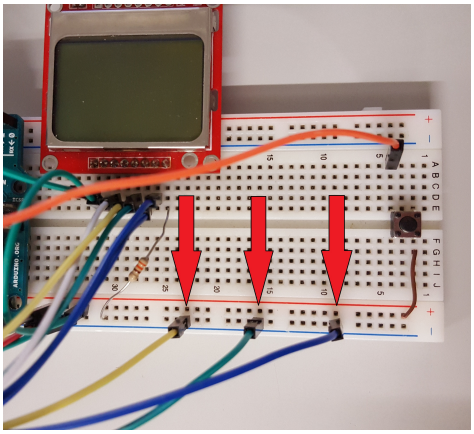

Fig. 11: Game wires.

2) Board Development: In addition to the software development, an arduino shield was designed with roughly the same Hardware set up in the breadboard used for testing and demonstration purposes. The extra hardware element added to the shield was a switch that allows to turn the LCD's backlight ON and OFF.

3) Schematic: As it is a simple shield, there are only a few extra components beside the LCD - the reset button, the backlight switch and the female headers for the wires. The reset button doesn't have a pull-up resistor because the pull-up effect (logic 0 @ Vcc) was set while developing the firmware. There are no headers for the Vcc and Gnd signals as these are already supplied by the arduino to the shield.

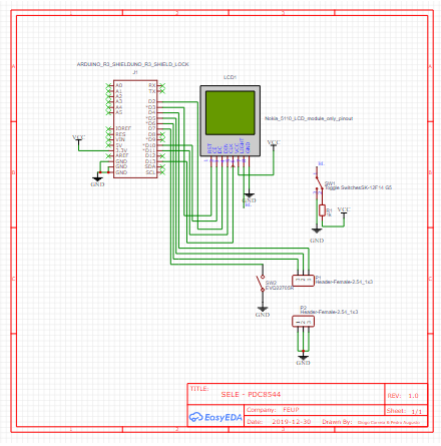

Fig. 14: *Defuse the Bomb* arduino shield schematic.

4) PCB: As with the Async9 project, section II, the top layer and the bottom layers were used, to avoid track crossing and positioning problems. Only the SCLK signal and gnd signals are routed in the bottom layer. All the other signals are routed in the top layer.

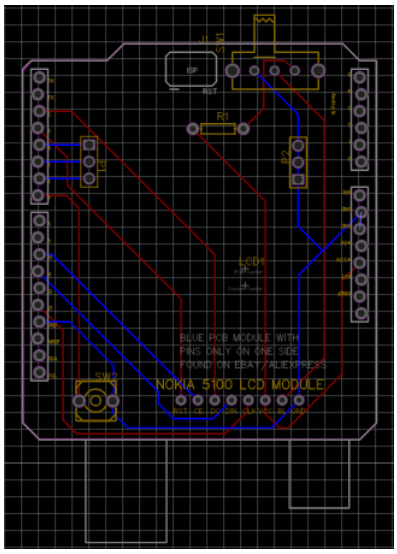

Fig. 15: *Defuse the Bomb* arduino shield PCB.

No restrictions were applied on track length or width - the default of 10 mil or 0.26mm was kept.

V. Conclusion

The report demonstrates three different projects that involved the ATMega328P microcontroller which could be considered a market standard, specially for simple applications. With the arduino uno board and programming directly the ATMega328p registers, we were able to learn about asynchronous communication, JTAG and synchronous communication, most specifically Serial Peripheral Interface or SPI. Besides the firmware developed for all three projects, two different PCBs were designed which could replace the breadboard setups if manufactured. All the different projects were successfully completed.