Hochschule Konstanz
Department of Computer Science

Patrick Reichle
Lukas Seehuber
Ubiquitous Computing Lab

Degree: Applied Informatics (AIN)
Lecture: Ubiquitous Computing

Report for the Laboratory 2: Communication between IoT devices

The main objective of this exercise is to understand the communication between IoT devices (in this case, Arduino), including encoding the data, using a checksum and input/output devices. Our task is to connect two Arduinos using the I2C interface for transmitting and receiving control commands, which control a RGB LED.

1.1 Control and properties of RGB LED on one Arduino

First we implemented the control for the RGB LED via a Virtual Terminal on only one Arduino. Therefore we connected the RGB LED with 220 ohm resistors for each of the three colour pins to protect the LEDs inside from high currents, as described in the previous laboratory where we connected a single LED.

We also need to connect the RGB LEDs common cathode (we use the CC type) to a ground to close the circuit as shown in the Figure below.[1]

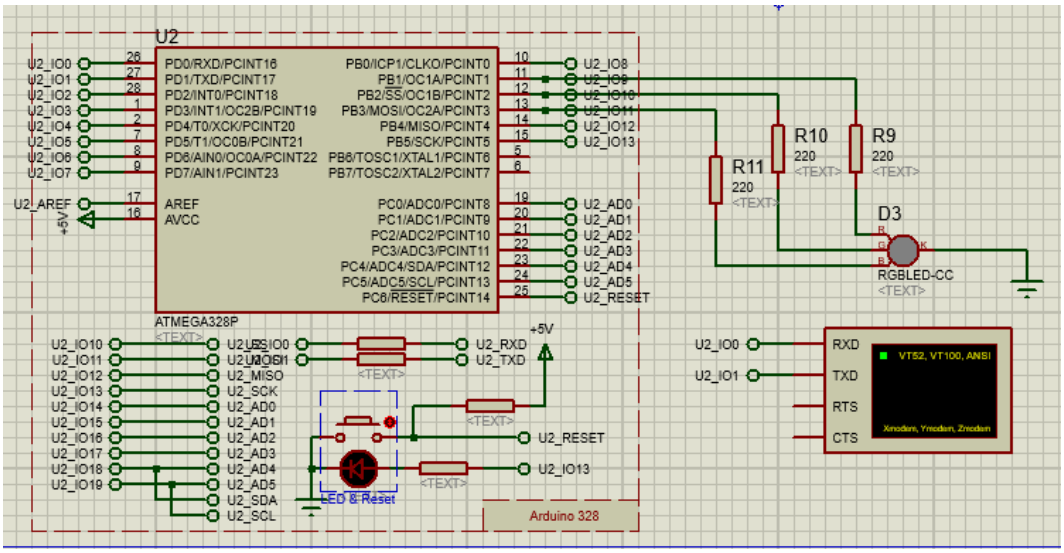

Figure 1: Setup with one Arduino to control RGB LED via a Virtual Terminal.

The Virtual Terminal is connected to the Arduino pins 0 and 1 with RX and TX switched for input/output connectivity again as described in the implementation of the previous laboratory.

[1] "Arduino RGB LED Tutorial", https://create.arduino.cc/projecthub/muhammad-aqib/arduino-rgb-led-tutorial-fc003e accessed on 01.07.2020

```
void RGB_color(int red_light_value, int green_light_value, int blue_light_value)
 {
  analogWrite(red_light_pin, red_light_value);
  analogWrite(green_light_pin, green_light_value);
  analogWrite(blue_light_pin, blue_light_value);
}

void RGB_rainbow() {
  RGB_color(255, 0, 0); // Red
  delay(500);
  RGB_color(255, 127, 0); // Orange
  delay(500);
  RGB_color(255, 255, 0); // Yellow
  delay(500);
  RGB_color(0, 255, 0); // Green
  delay(500);
  RGB_color(0, 0, 255); // Blue
  delay(500);
  RGB_color(75, 0, 130); // Indigo
  delay(500);
  RGB_color(148, 0, 211); // Violet
  delay(500);

}
```

Figure 2: Method for setting colours and the rainbow implementation.

```
void loop() {
  if (command == 'y') {
    RGB_color(255, 255, 0); // Yellow
    delay(1000);
  } else if (command == 'v') {
    RGB_color(148, 0, 211); // Violet
    delay(1000);
  } else if (command == 'w') {
    RGB_color(255, 255, 255); // White
    delay(1000);
  } else if (command == 'b') {
    RGB_rainbow();
    // no delay, function long enough
  } else if (command == 'o') {
    RGB_color(0, 0, 0); // off
    delay(1000);
    // delay could be omitted for higher response
  }
}
```

Figure 3: Implementation of the colour setting methods in the main loop.

The code in Figure 2 sets the current (between 0 and 255) individually for each pin to ensure the right “mix” of colours. This way the LED can represent a vast variety of different shades of colours. In Figure 4 you can see the input “v” parsing the command to set the RGB LED colour to violet. This is ensured by setting the PWM supporting Arduino pin connected to the red pin of the RGB LED via analogWrite() to value 148, accordingly is the green pin set to 0 and the blue pin set to 211, to represent the colour mixture of violet. As discussed in the first laboratory, it is important to connect the RGB LED pins to PWM supporting pins in order to be able to change the brightness of each colour by signalling

with different duty cycles as mentioned in the previous laboratory, where we used this mechanism to produce different pitches at a speaker.

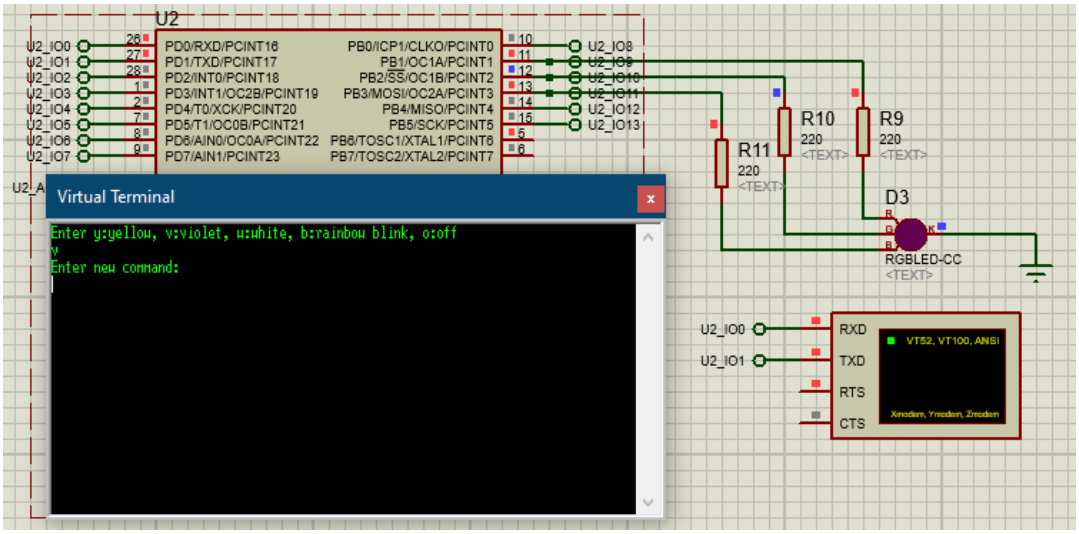

Figure 4: Simulation after typing the command “v” for violet, ready to read the next command.

1.2 Encoding/decoding and checksum algorithms

```
byte encodeCommand(char cmd) {
  if (cmd == 'y') {
    return B00000001;
  } else if (cmd == 'v'){
    return B00000011;
  } else if (cmd == 'w'){
    return B00000111;
  } else if (cmd == 'b'){
    return B00001111;
  } else if (cmd == 'o'){
    return B00011111;
  }
  return B00000000;
}
```

Figure 5: Encoding method, which converts the char-type input “*cmd*” to a byte.

We encode our commands by assigning each command a pattern of n 0s and 8-n 1s at the end. One byte per command is more than enough, since one byte could encode up to 256 different commands.

```
byte CRC8(byte* bytes, int len) {
  const byte generator = 0x07;
  byte crc = 0;

  for (int j = 0; j < len; j++) {
    crc ^= bytes[j];

    for (int i = 0; i < 8; i++) {
      if ((crc & 0x80) != 0) {
        crc = (byte)((crc << 1) ^ generator);
      }
      else {
        crc <<= 1;
      }
    }
  }
  return crc;
}
```

Figure 6: Implementation of the CRC8-algorithm.

The CRC8 checksum is the remainder of the polynomial division of the message by the so-called generator polynomial (we use 0x07). After calculating the checksum, the encoded command and the checksum are transmitted to the second Arduino as shown in Figure 9.

Our algorithm shown in Figure 6 receives an array of bytes as input. The polynomial division is done by XOR-ing the intermediate result with the generator polynomial if the leading bit is 1. If the bit is 0 we simply shift by one bit. After all bits are processed the intermediate result holds the checksum.

After receiving the message, as shown below, the CRC8 algorithm is run on the command and the checksum. If the result is 0 the message has been transmitted correctly. Note that there is always a slight possibility of corrupted data, that produces the same result, but this is very unlikely to happen here, as we use set byte codes, that don't interfere with each other. However the sender and receiver have to use the same generator polynomial for the algorithm to work properly.

```
void receiveCommand(int bytes) {
  byte enc = Wire.read();
  byte crc = Wire.read();
  byte check[] = {enc,crc};
  byte res = CRC8(check,2);
  if (res != 0) {
    mySerial.println("Received corrupted command!");
  } else {
    if (enc == B00000001) {
      command = 'y'; // Yellow
    } else if (enc == B00000011) {
      command = 'v'; // Violet
    } else if (enc == B00000111) {
      command = 'w'; // White
    } else if (enc == B00001111) {
      command = 'b';
    } else if (enc == B00011111) {
      command = 'o'; // off
    }
  }
}
```

Figure 7: Receiving the message, checking for corruption and finally decoding the command.

1.3 Final Setup with two Arduinos

With our basic RGB LED controls working, we now connect the first Arduino to the previous one in order to make it the input device, with the Virtual Terminal now connected to it as shown below.

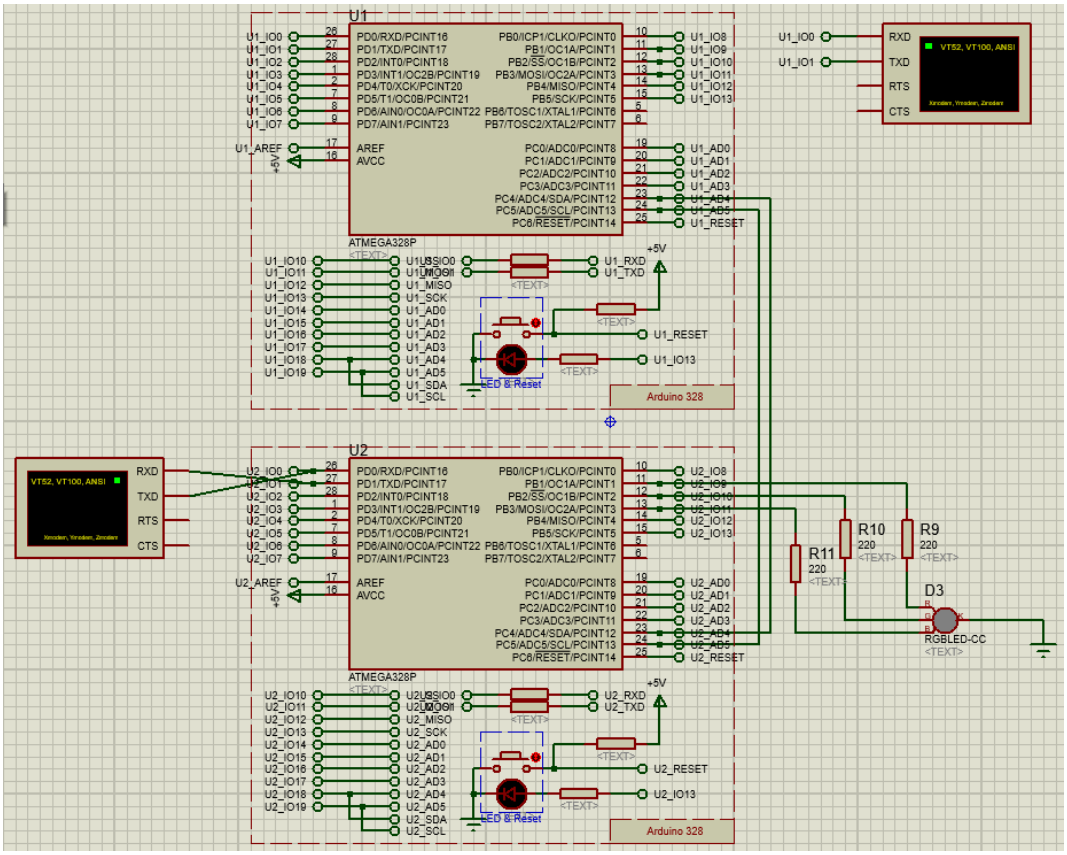

Figure 8: Final setup with two Arduinos connected via master and slave connection (I2C). First Arduino connected to an input terminal and second Arduino with RGB LED as output device.

As a debugging tool we still have another Virtual Terminal connected to the second Arduino as shown above, which is only used as an output device to have more information about possible errors in the transmission and encoding/decoding algorithms, but it doesn't affect the running process at all.

```
void sendCommand(char cmd) {
 byte enc = encodeCommand(cmd);
 byte encArr[] = {enc};
 byte crc = CRC8(encArr, 1);
 byte data[] = {enc, crc};
 Wire.beginTransmission(9); // transmit to device #9
 Wire.write(data, 2);
 Wire.endTransmission();
}
```

Figure 9: Transmission of the encoded message with the calculated checksum via I2C to the second Arduino.

For the communication between the two Arduinos we used the I2C interface. It is implemented in the Wire library and in order for it to work we need to connect the SDA (data line) and SCL (clock line) pins of the two Arduinos to ensure proper communication from master to slave.[2] The master device (the Arduino that reads the commands from the terminal) uses the beginTransmission(address) and write() methods to send the message to the device at 'address' (in our case the address is 9). The write()-method takes an array of bytes as argument. After that we end the transmission with the endTransmission() method. The slave device must listen on the same address and registers an event handler function that is called every time the slave receives a message. The message is read and tested for corruption. If the message is valid the command is decoded and executed.

The second Arduino, now responsible for decoding is still setting the RGB LED colours as before, but now according to the received information from the first Arduino.

1.4 Control commands and corresponding simulation

To control the colour of the RGB LED we implemented five commands, four of them can set colours, even though one is the colour black, hence this is similar to “off”, and the fifth command calls a method that lets the RGB LED alternately blink through the colours of the rainbow. The commands with corresponding simulation output of the RGB LED in the final setup are shown below.

The available commands are:

- 'y' → set LED colour to yellow (Figure 10)
- 'v' → set LED colour to violet (Figure 11)
- 'w' → set LED colour to white (Figure 12)
- 'b' → blink LED in rainbow colours (Figure 14)
- 'o' → turn LED off (Figure 13)

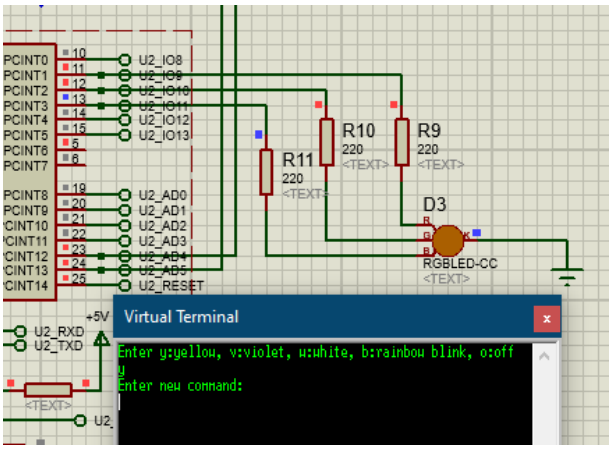

Figure 10: Command “y” for yellow.

[2] “Wire Library”, https://www.arduino.cc/en/Reference/Wire accessed on 02.07.2020

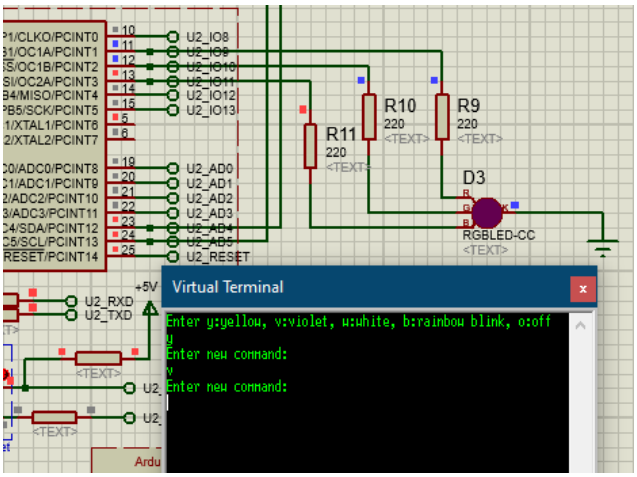

Figure 11: Command “v” for violet.

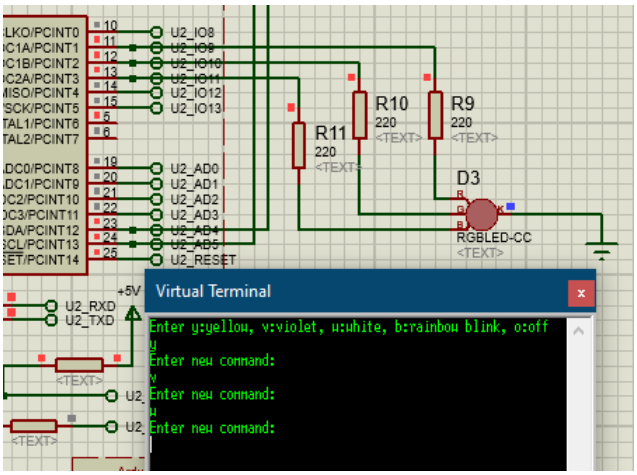

Figure 12: Command “w” for white.

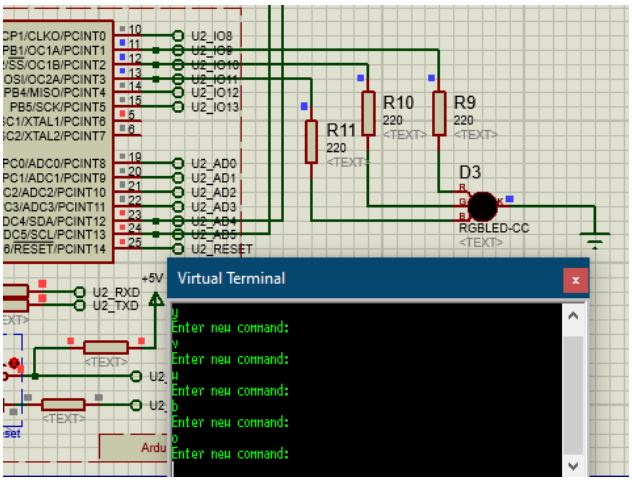

Figure 13: Command “o” for off/black.

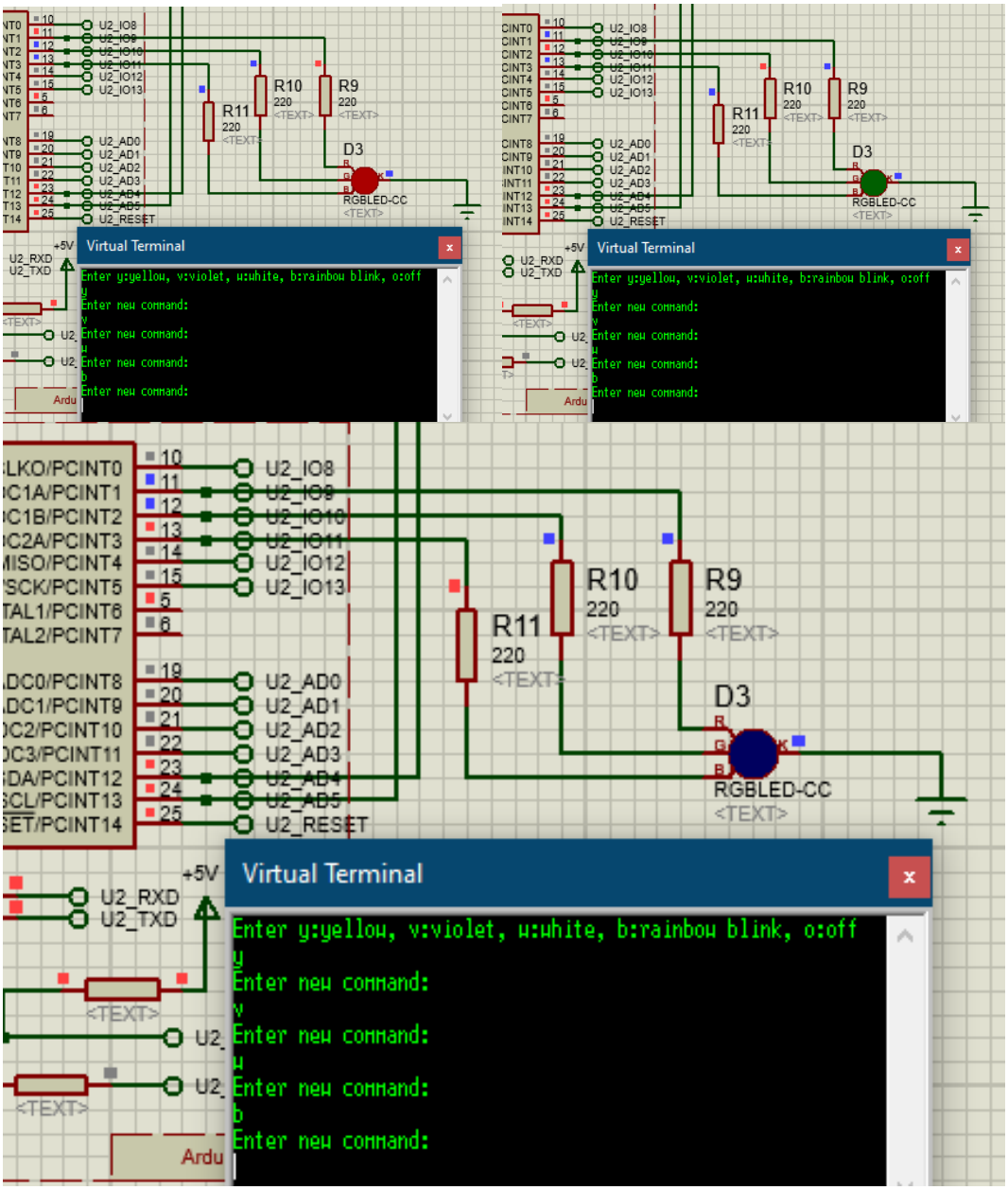

Figure 14: Command “b” for blinking the colours of the rainbow alternately. The pictures obviously don’t show a representation of all 7 colours concerning space. Representative the first colour red the fourth colour green and fifth colour blue are shown here.

1.5 Code and Setup on GitHub

The code for the two Arduinos “input” and “output” as well as the Proteus setup for the simulation can be accessed at:

https://github.com/patrick-reichle/UbiCom/tree/master/lab2