Microcontrollers Programming

StockTracker Project

Made by: Faisal Kassem, STID:404386

Lodz University 2023

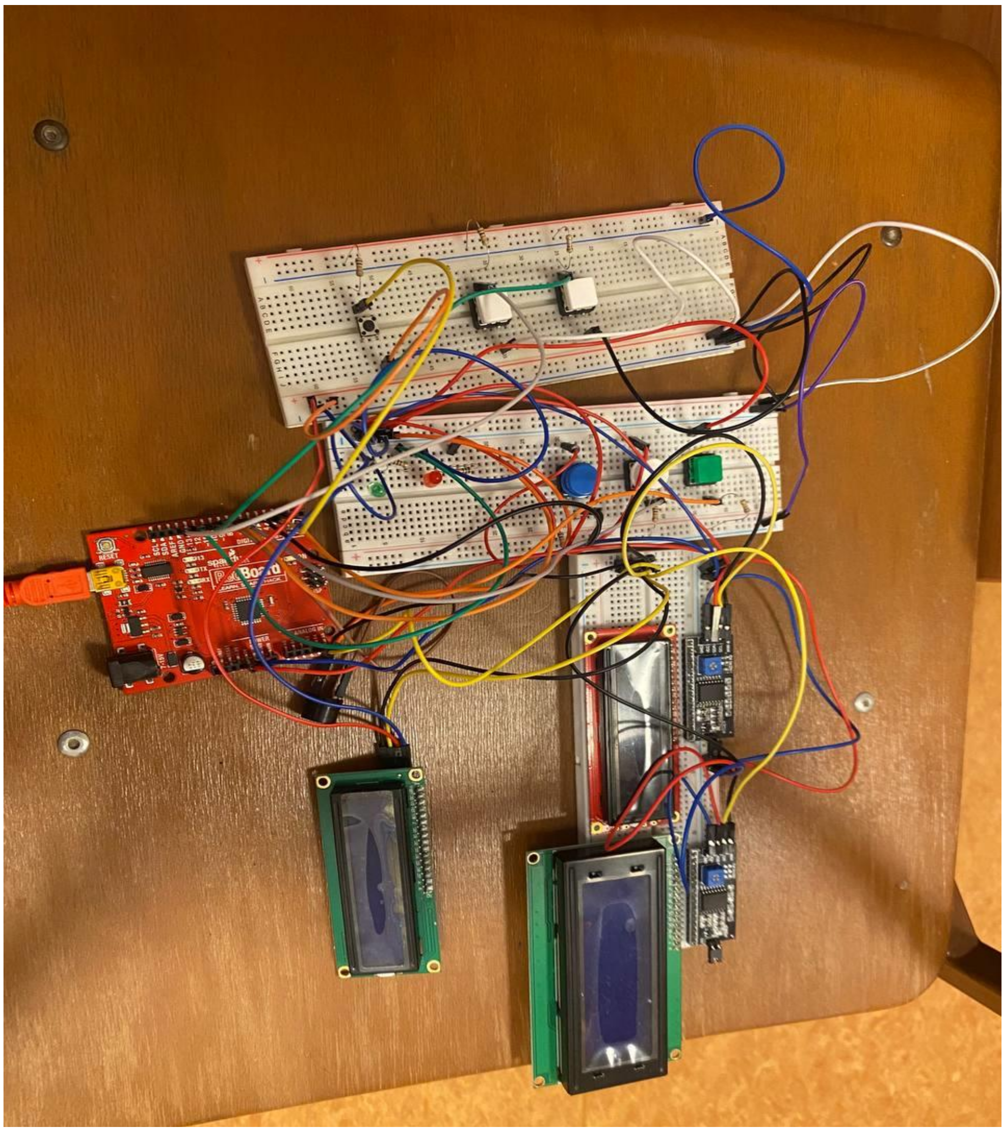
Briefly

StockTracker is an application based on Arduino and Python which aims to ease the stock tracking as well as managing process for the user, by means of constant monitoring and updates of the stock status, together with stock management.

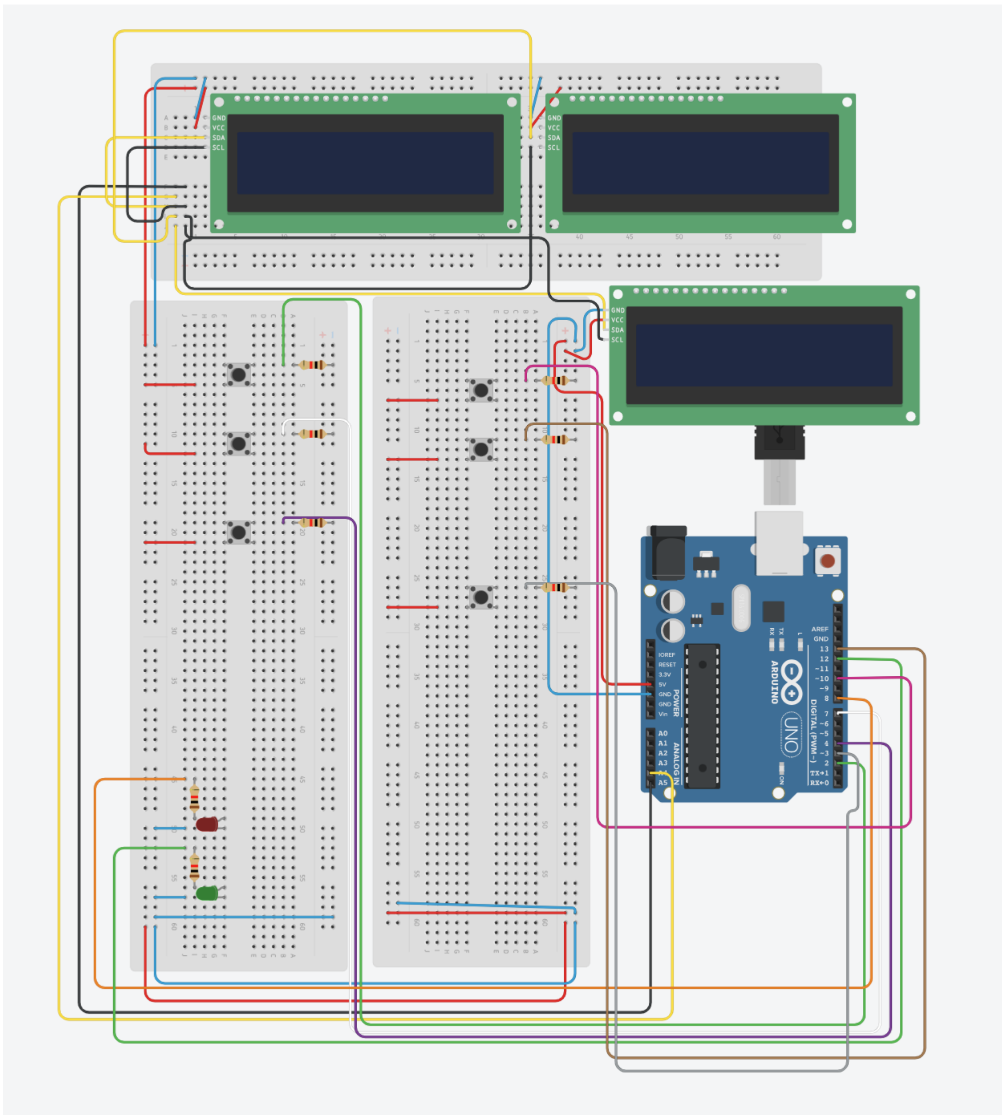
Components used:

- Arduino Uno
- Wires
- 6 Buttons
- Resistors (OHM)
- 2 LEDs (RED, GREEN)
- Resistors (OHM)
- 2 LCD Displays + 2 I2C modules, and LCD(I2C soldered). (Note combinations like: 3 LCD Displays + 3 I2C modules, or 3 LCD(I2C soldered) can be used as well)
- Buzzer

Overview



Schema



1

Part: API

In order to implement the project, we need to select a trading platform which provides API for the data about stocks from their server. For this project we have chosen XTB trading platform (but you can choose any platform with own API which matches our needs, XTB API is working, but there are some issues we will cover during this guide). XTB API provides the instrument to retrieve data about stocks, orders, history, candles (https://en.wikipedia.org/wiki/Candlestick_chart), additionally XTB Platform allows to make trades.

xtb
online trading

References:

API Docs:
http://developers.xstore.pro/documentation/

XTB Platform:

https://www.xtb.com/pl

2

Part: API Wrappers Ways of Working

XTB has provided their API service together with the documentation, that you can follow if you wish to create your own API wrapper (note in order to create it yourself you need knowledge about web sockets, see references in docs: http://developers.xstore.pro/documentation/). Nonetheless, XTB provides their own API wrapper for .NET, JAVA, or PYTHON (http://developers.xstore.pro/api/wrappers/2.5.0), however as we are going to use python for implementation, some part of realization of the official wrapper is not user-friendly to use, and causes issues in most of cases. Thus we will use custom wrapper available on github.

xAPI Protocol Documentation

References:

Github API Wrapper Repository by frederico123579: https://github.com/federico123579/XTBApi

Github API Wrapper Repository by Saitama298:

https://github.com/Saitama298/Python-XTB-API

3

Part: API Wrappers Ways of Working

Initially, when the project was in demo stage, only Github API Wrapper by Saitama298 was user, which allowed to do almost all needed operations (in the project scope: retrieving stock data, account data, balance, and opening positions) except for closing the positions. Later was discovered API Wrapper by Frederico123579 which allowed us to get through this limitation and provide full functionality for the project

References:

Github API Wrapper Repository by frederico123579: https://github.com/federico123579/XTBApi

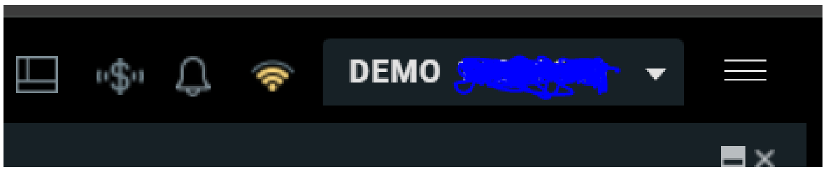
Github API Wrapper Repository by Saitama298:

https://github.com/Saitama298/Python-XTB-API

4

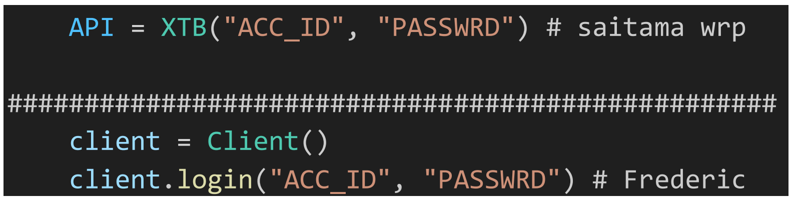
Part: Lastly about API

In order to access XTB data you may need an account. In our project we can use both official or demo account provided by XTB (however for official you will need a bunch of confirmations and document verifications which we will skip in this project for sure). We are going to use demo account (to use official account check documentation by xstore in order to avoid issues in connection, in case of wrappers this situation is handled so you can just switch between account on off/demo by providing different account ids). Thus, to use API you need to pass account id and password you used in registration of your account.



Account Id can be found on as on the picture above in the top left corner of the browser if you use PC. Where it has format like DEMO <ACCOUNT_ID>. <ACCOUNT_ID> is what you need

Python object creations for APIs:



```
    API = XTB("ACC_ID", "PASSWRD") # saitama wrp

##################################################
    client = Client()
    client.login("ACC_ID", "PASSWRD") # Frederic
```

5

Part: Code overview and Serial Communication (arduino <-> PC)

In this guide, we will cover some code part, however to fully browse the code, you access source codes provided with documentation, or on a github attached in the end of the documentation.

Starting from basics, here are the following libraries used in python:

```
import serial #To send to Arduino
import time #For Timers
import sys
from API import XTB # saitama
from XTBApi.api import Client #
Frederico
```

To sustain the communication between PC and Arduino, when setting up the Serial Port on Arduino and connecting via python we have to set same baud rate (https://forum.arduino.cc/t/serial-monitor-baud-rate/689338/3). For this project I preferred default 9600 (but for some fast implementation you may consider to increase this value on both ends)

Python:

```
ser = serial.Serial('COM5', 9600, timeout=0.1)
```

Arduino:

```
#define BRATE 9600
…
void setup() {
…
  Serial.begin(BRATE);
…
}
```

5 Part: Code overview and Serial Communication (arduino <-> PC)

As we have two way communication, let’s firstly consider communication Python (PC) to Arduino.

Firstly, in python we setup what exactly we are willing to send (in case if preferrable it can be expanded):

```python
def determine_indicators():

    #set indicator arrays
    available_indicators = [1, 2, 3, 4, 5]
    target_indicators = []
    choice = 0

    while choice != 5:

        print("Please select your desired indicators: (Note, certain indicators may require
additional Arduino setup.")

        for indicator in available_indicators:

            if indicator == 1:
                print("1. Current Stock Price (###: XX.XX)")

            elif indicator == 2:
                print("2. Change Since Open (+X.XX)")

            elif indicator == 3:
                print("3. Percent Change Since Open (-X.XX%)")

            elif indicator == 4:
                print("4. Day Change Indicator Light (Green light or Red Light)")

            elif indicator == 5:
                print("5. Finish Program Setup")

        choice = int(input("Choice (1-5): "))

        if choice in available_indicators and choice != 5:

            available_indicators.remove(choice)
            target_indicators.append(choice)

        elif choice != 5:
            print("Not a valid selection.")

    #return chosen indicators
    return target_indicators
```

5

Part: Code overview and Serial Communication (arduino <-> PC)

Beforehand, we should also request from user number of tickers to track:

```
num = int(input("Input number of
trackable tickers: "))
        i = 0
        while i < num:
            target = get_target()
            ticker.append(target)
            i+=1
```

It is also necessary to check if particular ticker provided by user exist on the platform. Potential solution: for this in the setup procedure we request all possible options and check if such symbol exists on the platform, if no we shall restart application or proceed.

Examples of data requested in the code is:

```
    symboldata = API.get_Symbol(ticker[curTrack])
```

- Returns information about the Symbol Record.

```
candles = API.get_Candles("D1", ticker[curTrack],
qty_candles=2)
```

- Returns information about Price changes for a Day (see "D1", also exist options like "H1", "M1" and etc.)

```
margin = API.get_Margin(ticker[curTrack], curLot)
```

- Returns how much will be invested for particular stock and volume. (E.g. CRYPTO: ETHEREUM 0.05 – 190 PLN)

We should also follow connection validation for the server:

At most 50 simultaneous connections from the same client address are allowed (an attempt to obtain the 51st connection returns the error EX008). If you need this rule can be lenified please contact the xStore Support Team.

Every new connection that fails to deliver data within one second from when it is established may be forced to close with no notification.

Each command invocation should not contain more than 1kB of data.

User should send requests in 200 ms intervals. This rule can be broken, but if it happens 6 times in a row the connection is dropped.

Each command should be a proper JSON object.

Part: Code overview and Serial Communication (arduino <-> PC)

Thus, to follow connection validation and make runnable program, it was decided to implement two simple additional mechanisms. 1. Timer Mechanism; 2. Primitive Recovery mechanism;

Starting from timer mechanism. We could use Sleep command after each API request (200ms sleep), however it may slow down significantly the program. In the code however we used sleeps in some places to optimize and ease the development process, but recommended to completely refuse from it.

The timer mechanism basically looks like this:

```
…
current_time_API = time.time()
        if current_time_API - start_time_API >=
interval_API:
…
             start_time_API = current_time_API
```

```
…
```

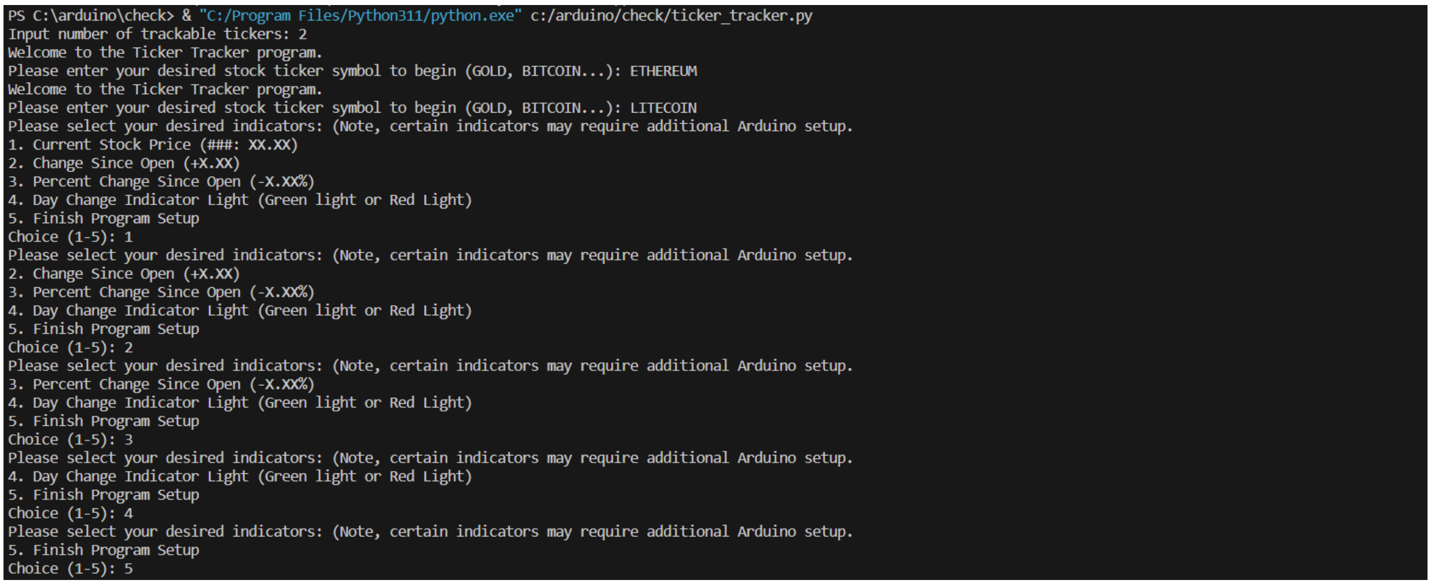
So here we basically just in the real-time of the program check how much time passed since starting time till current time and if it satisfies interval we perform some action.

2. Primitive Recovery In our case is extremely simple: Whenever we catch some exception (except for Keyboard interrupt or ClearCommError (which appears when device is manually disconnected during program running time)) we just restart the program.

```
#run program
if __name__ == "__main__":
    exceptionOccured = False
    while True:
        try:
            if not exceptionOccured:
                main(True)
            else:
                main(False)
                exceptionOccured = False
        except Exception as e:
            print(str(e))
            if isinstance(e, KeyboardInterrupt) or
"ClearCommError" in str(e):
                sys.exit(0)
            else:
                print("ERROR OCCURED: ", e)
                exceptionOccured = True
        time.sleep(2)
```

5 Part: Code overview and Serial Communication (arduino <-> PC)

Examples of console output:



```
PS C:\arduino\check> & "C:/Program Files/Python311/python.exe" c:/arduino/check/ticker_tracker.py
Input number of trackable tickers: 2
Welcome to the Ticker Tracker program.
Please enter your desired stock ticker symbol to begin (GOLD, BITCOIN...): ETHEREUM
Welcome to the Ticker Tracker program.
Please enter your desired stock ticker symbol to begin (GOLD, BITCOIN...): LITECOIN
Please select your desired indicators: (Note, certain indicators may require additional Arduino setup.
1. Current Stock Price (###: XX.XX)
2. Change Since Open (+X.XX)
3. Percent Change Since Open (-X.XX%)
4. Day Change Indicator Light (Green light or Red Light)
5. Finish Program Setup
Choice (1-5): 1
Please select your desired indicators: (Note, certain indicators may require additional Arduino setup.
2. Change Since Open (+X.XX)
3. Percent Change Since Open (-X.XX%)
4. Day Change Indicator Light (Green light or Red Light)
5. Finish Program Setup
Choice (1-5): 2
Please select your desired indicators: (Note, certain indicators may require additional Arduino setup.
3. Percent Change Since Open (-X.XX%)
4. Day Change Indicator Light (Green light or Red Light)
5. Finish Program Setup
Choice (1-5): 3
Please select your desired indicators: (Note, certain indicators may require additional Arduino setup.
4. Day Change Indicator Light (Green light or Red Light)
5. Finish Program Setup
Choice (1-5): 4
Please select your desired indicators: (Note, certain indicators may require additional Arduino setup.
5. Finish Program Setup
Choice (1-5): 5
```

Now let's talk about the way of sending messages to Arduino. There are bunch of implementations for that. One of the implementations I would highly recommend to follow, however without investigating it may be a bit complex for understanding (in this implementation we skipped it, as we have a bunch of data to send to Arduino, but not that large to follow that pattern), is the following implementation by Nick Gammon:

Reference to the website with proper explanation: http://www.gammon.com.au/serial

Or you can also check this one by "Curious Scientist": https://curiousscientist.tech/blog/fast-serial-communication-with-arduino

However, we will follow an example: https://www.programmingelectronics.com/serial-read/?utm_source=YouTube&utm_medium=VideoDescription&utm_campaign=LeadGen&utm_content=Using%20Serial.read()%20with%20Arduino%20|%20Part%202

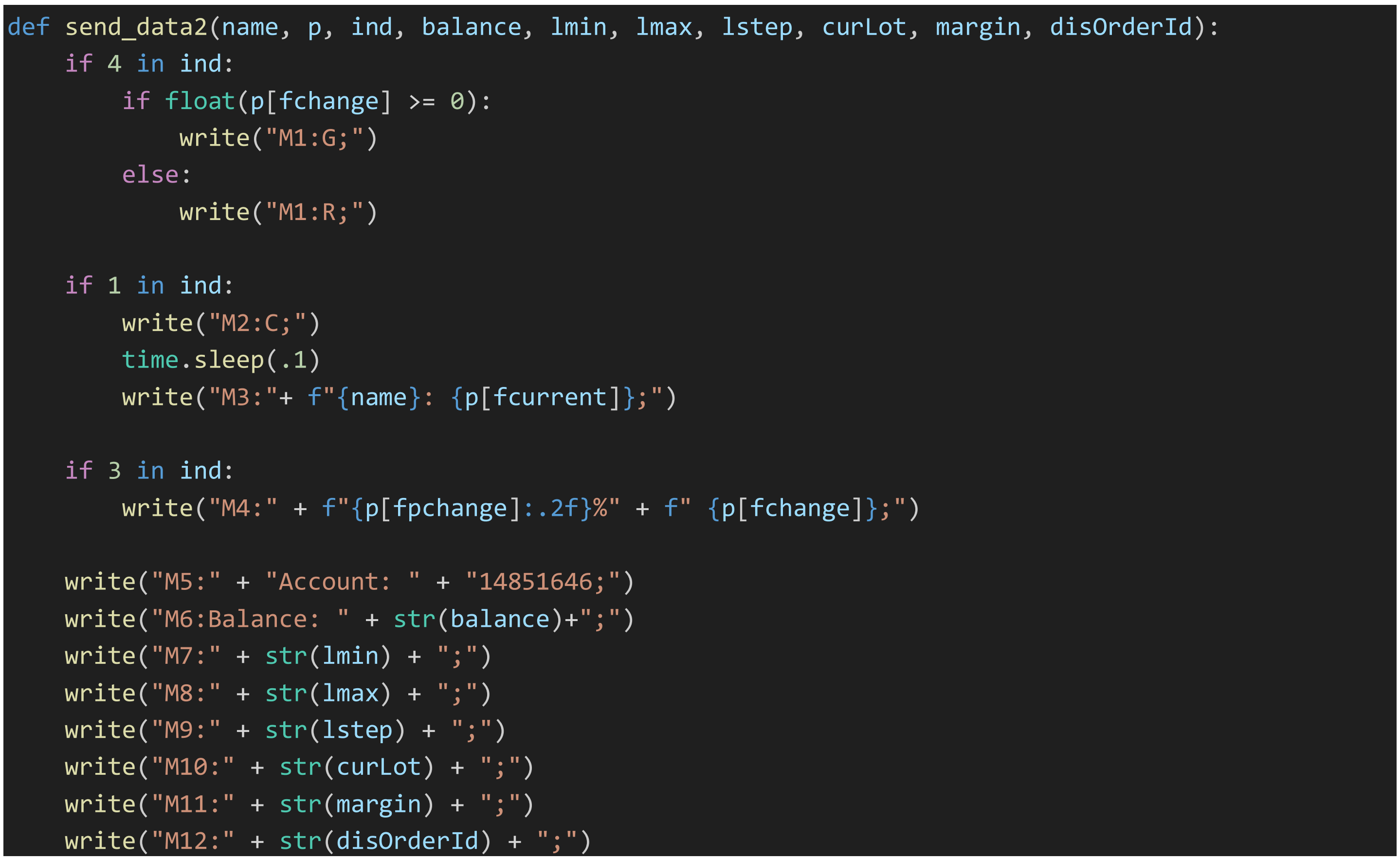
In the example above, as Arduino's Serial.read() function can only read incoming data by byte each piece, they distinguish if messages are different whenever the byte is equal '\n' or if the message length exceed maximum set length. We will use similar approach. The message which is sent has the following format:

M[NUM]:[MESSG];

[NUM] in that context indicates the order of the message, which is sent first, second, third and etc. ':' separates order with the particular message and ';' will be our message separator, if we reach byte equal to ';' that means we received one message. Letter 'M' is just kept for better understanding of data, nothing else more.

5 Part: Code overview and Serial Communication (arduino <-> PC)

On python side the data send looks like this:



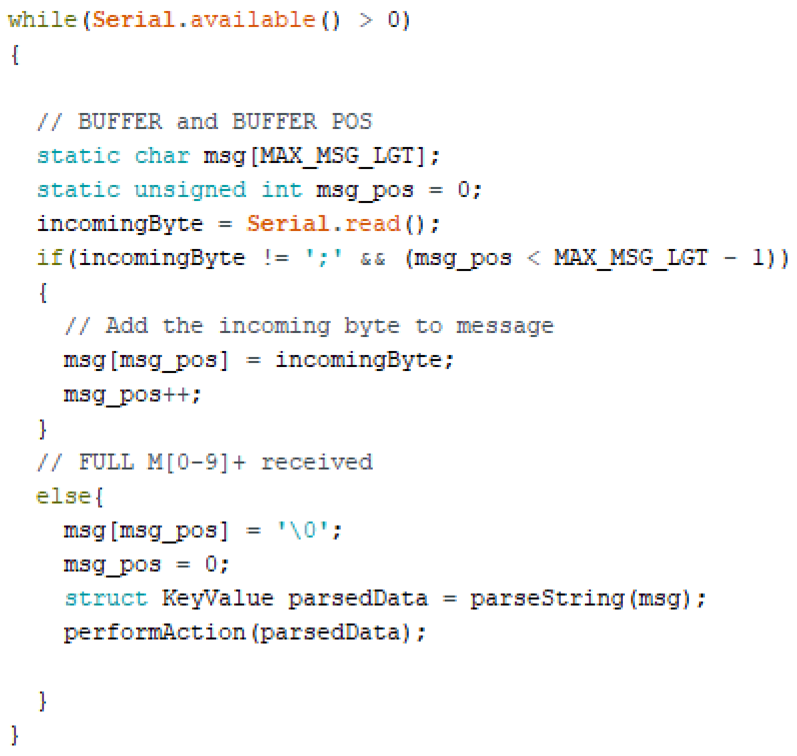
```python
def send_data2(name, p, ind, balance, lmin, lmax, lstep, curLot, margin, disOrderId):
    if 4 in ind:
        if float(p[fchange] >= 0):
            write("M1:G;")
        else:
            write("M1:R;")

    if 1 in ind:
        write("M2:C;")
        time.sleep(.1)
        write("M3:"+ f"{name}: {p[fcurrent]};")

    if 3 in ind:
        write("M4:" + f"{p[fpchange]:.2f}%" + f" {p[fchange]};")

    write("M5:" + "Account: " + "14851646;")
    write("M6:Balance: " + str(balance)+";")
    write("M7:" + str(lmin) + ";")
    write("M8:" + str(lmax) + ";")
    write("M9:" + str(lstep) + ";")
    write("M10:" + str(curLot) + ";")
    write("M11:" + str(margin) + ";")
    write("M12:" + str(disOrderId) + ";")
```

On arduino's side data is processed in the following way:



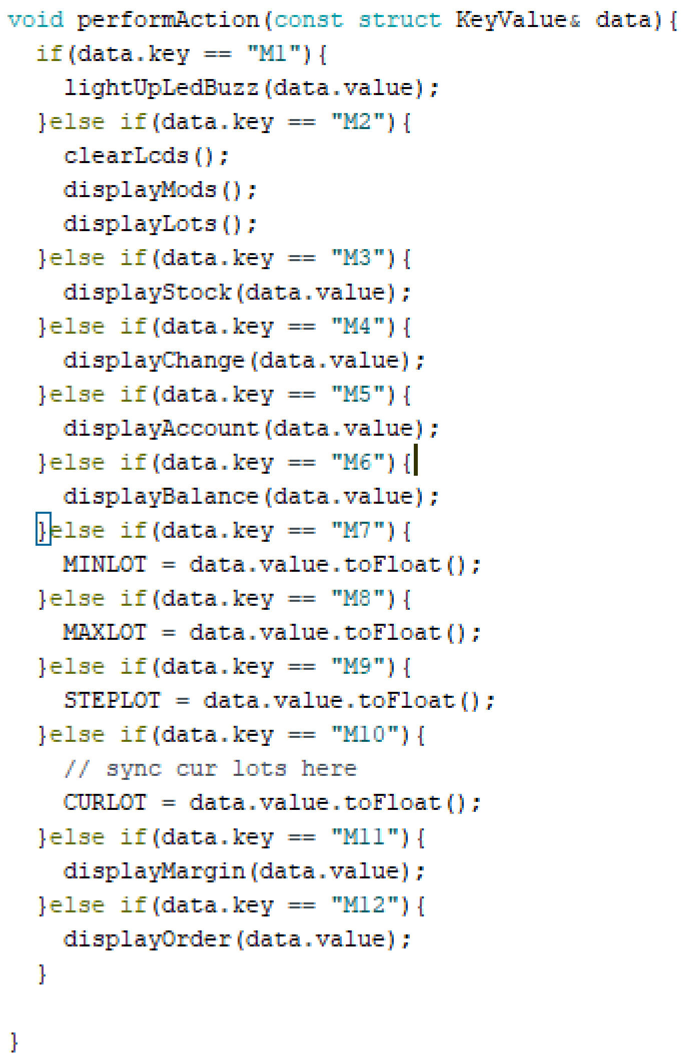
```cpp
while(Serial.available() > 0)
{

  // BUFFER and BUFFER POS
  static char msg[MAX_MSG_LGT];
  static unsigned int msg_pos = 0;
  incomingByte = Serial.read();
  if(incomingByte != ';' && (msg_pos < MAX_MSG_LGT - 1))
  {
    // Add the incoming byte to message
    msg[msg_pos] = incomingByte;
    msg_pos++;
  }
  // FULL M[0-9]+ received
  else{
    msg[msg_pos] = '\0';
    msg_pos = 0;
    struct KeyValue parsedData = parseString(msg);
    performAction(parsedData);

  }
}
```

5 Part: Code overview and Serial Communication (arduino <-> PC)

The way arduino performs actions after message is processed is pretty straightforward, functionality for it look like following:



```
void performAction(const struct KeyValue& data){
  if(data.key == "M1"){
    lightUpLedBuzz(data.value);
  }else if(data.key == "M2"){
    clearLcds();
    displayMods();
    displayLots();
  }else if(data.key == "M3"){
    displayStock(data.value);
  }else if(data.key == "M4"){
    displayChange(data.value);
  }else if(data.key == "M5"){
    displayAccount(data.value);
  }else if(data.key == "M6"){
    displayBalance(data.value);
  }else if(data.key == "M7"){
    MINLOT = data.value.toFloat();
  }else if(data.key == "M8"){
    MAXLOT = data.value.toFloat();
  }else if(data.key == "M9"){
    STEPLOT = data.value.toFloat();
  }else if(data.key == "M10"){
    // sync cur lots here
    CURLOT = data.value.toFloat();
  }else if(data.key == "M11"){
    displayMargin(data.value);
  }else if(data.key == "M12"){
    displayOrder(data.value);
  }

}
```

In order to display this much data, we are required to have multiple lcds. In order to use and deprive us of an imaginable wiring, we shall use I2C modules for Arduino (which also use following library: #include <LiquidCrystal_I2C.h>) and LCDs objects are created in the following way:

LiquidCrystal_I2C lcd(LCD1, 16, 2);

LiquidCrystal_I2C lcd2(LCD2, 20, 4);

LiquidCrystal_I2C lcd3(LCD3, 16, 2);

5 Part: Code overview and Serial Communication (arduino <-> PC)

Where LCD1, LCD2, and LCD3 are different addresses for I2C.

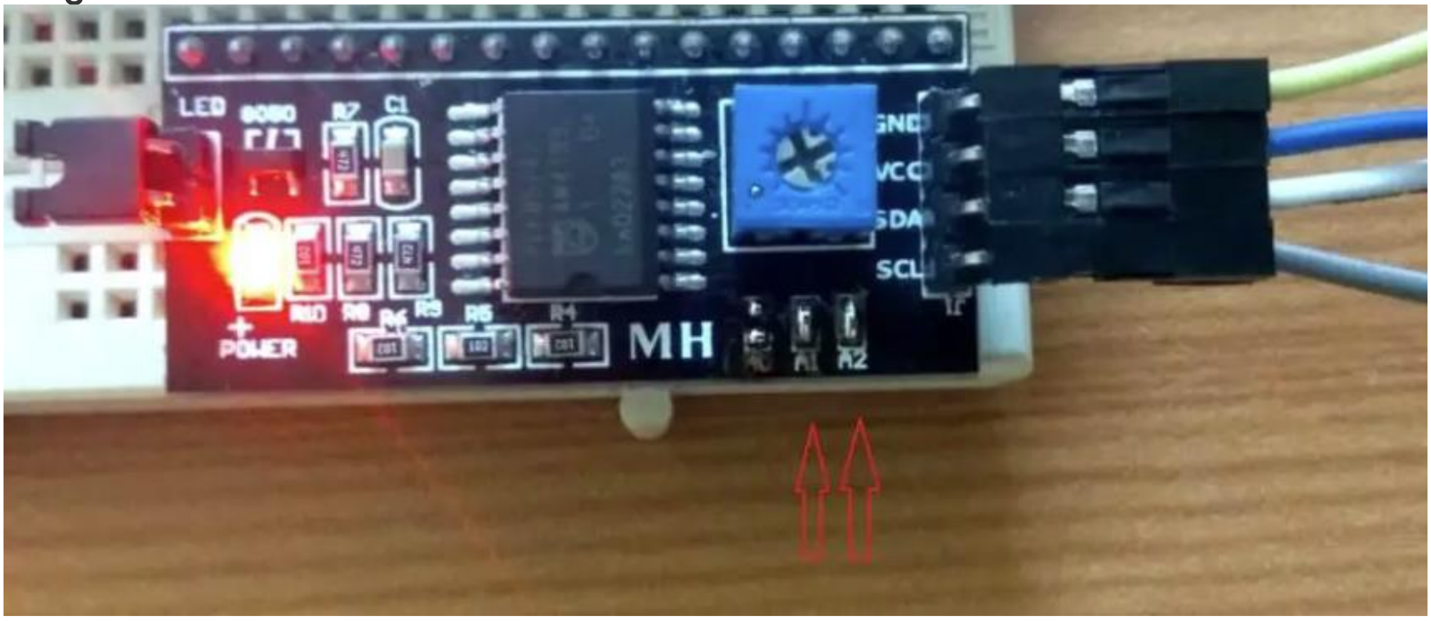
```
#define LCD1 0x22

#define LCD2 0x27

#define LCD3 0x23
```

In order to understand better the I2C module connection I recommend reading this article: https://electronics-project-hub.com/how-to-connect-multiple-lcd-to-arduino/

However, just important to note In order to change I2C module address we have to solder bridges mentioned in tutorial:

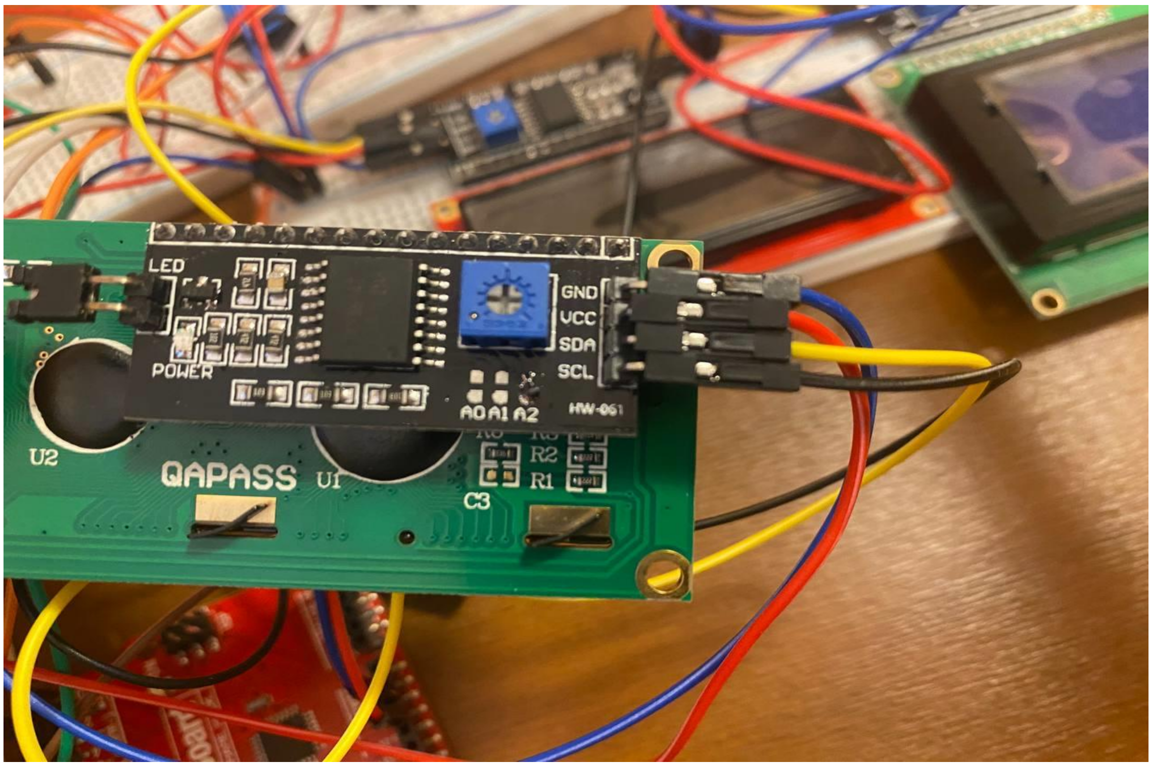


In my case I soldered In following way:

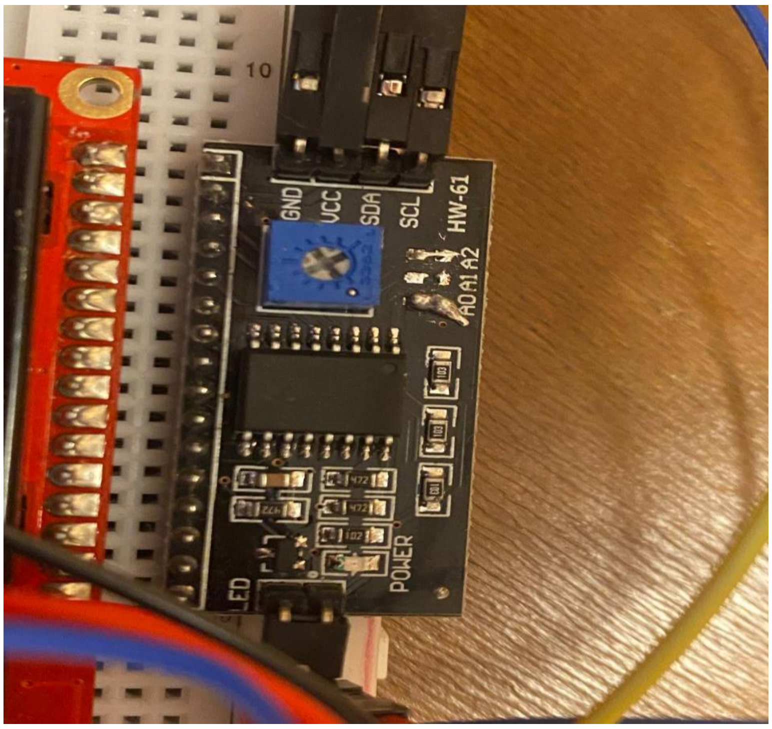
5 Part: Code overview and Serial Communication (arduino <-> PC)

One I2C module is kept intact;

2nd:



3rd:



5 Part: Code overview and Serial Communication (arduino <-> PC)

Time to consider Arduino to PC communication. To send data to PC we use Serial.Write(); command, and the messages are pretty simple, we will consider them further:

In our project, we have 5 buttons that we should consider individually:

#define PIN_BUTST 4

#define PIN_BUTMOD 7

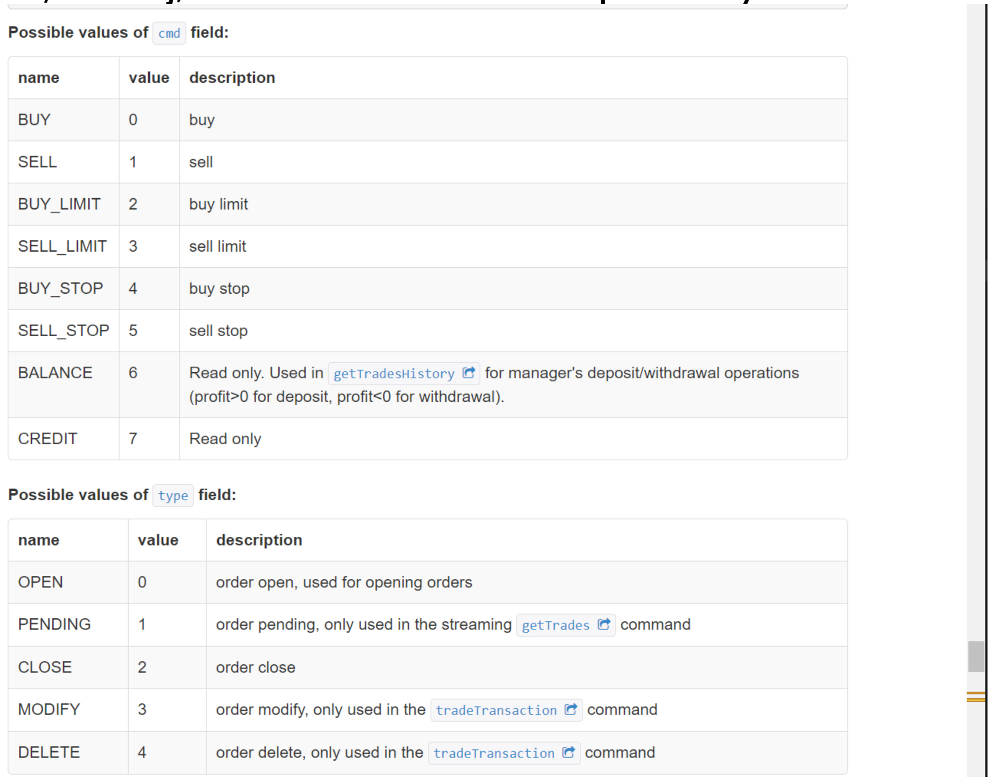
#define PIN_BUTACT 2

#define PIN_BUTORDER 3

#define PIN_BUTINC 13

#define PIN_BUTDEC 10

1. **PIN_BUTST – This button triggers Arduino to send message "STOCK" which we defined on python side to switch to another (or do nothing if number of tickers is 1 i.e.) stock ticker, e.g. ETHEREUM - > LITECOIN. So we will start getting data about Litecoin from the next tick.**
2. **PIT_BUTMOD – In our project implementation we have up to 3 mods (they are the same on both ends so let's check Arduino): char MODS[][13] = { "BUY/OPEN   ", "SELL/OPEN   ", "CLOSE/CLOSE "}; . Number of mods can be expanded by some of those combinations**



Possible values of `cmd` field:

name	value	description
BUY	0	buy
SELL	1	sell
BUY_LIMIT	2	buy limit
SELL_LIMIT	3	sell limit
BUY_STOP	4	buy stop
SELL_STOP	5	sell stop
BALANCE	6	Read only. Used in `getTradesHistory` for manager's deposit/withdrawal operations (profit>0 for deposit, profit<0 for withdrawal).
CREDIT	7	Read only

Possible values of `type` field:

name	value	description
OPEN	0	order open, used for opening orders
PENDING	1	order pending, only used in the streaming `getTrades` command
CLOSE	2	order close
MODIFY	3	order modify, only used in the `tradeTransaction` command
DELETE	4	order delete, only used in the `tradeTransaction` command

5 Part: Code overview and Serial Communication (arduino <-> PC)

3. **PIN_BUTACT – This button triggers Arduino to send message “ACT” which would trigger python to perform API request which i.e. would open a position for the specified by user volume, close the position and etc.**

4. **PIN_BUTORDER – This button is special for closing order functionality. As in order to differentiate between positions, API requires order ID, thus to switch between orders we click this button which sends message “ORDER” which would require python to switch to another order in case if we desire then push button ACT with MOD: CLOSE/CLOSE, which would close the position.**

5. **PIN_BUTINC – we use this button to increase the volume of the stock we are willing to buy. Example, minimal lot for the stock is 0.05, the step of the volume for stock is 0.01, and max is 100. So price varies basing on the volume for the stock, for 0.05 < 0.1 and etc.**

6. **PIN_BUTDEC – works same except it is used to decrease the volume**

How python processes the events:

```python
def read(maxrange, maxorderl, curlot, steplot, maxlot, minlot):
        data = ser.readline().decode().strip()
        global curTrack
        global lastMode
        global doAct
        global orderTrack
        if data:
            list = data.split(" ")
            # for simplicity always increase cur track here
            for i in list:
                if i == "STOCK":
                        curTrack+=1
                        if curTrack == maxrange:
                            curTrack = 0
                if i == "ORDER":
                    if maxorderl > 0:
                        orderTrack += 1
                        print(orderTrack, " ", maxorderl)
                        if orderTrack == maxorderl:
                            orderTrack = 0
                if i in MODELIST:
                    lastMode = i
                if i == "ACT":
                   doAct = True
                if i == "INCLOT":
                    curlot += steplot
                    if curlot > maxlot:
                        curlot = maxlot
                if i == "DECLOT":
                …
```

5 Part: Code overview and Serial Communication (arduino <-> PC)

Actions handling looks like following:

```
        if doAct == True:
            time.sleep(0.2)
            if lastMode == MODELIST[0]:
                status, order = API.make_Trade(ticker[curTrack], 0, 0, curLot)
                print(status)
                print(order)
                orderTrack = 0
            if lastMode == MODELIST[1]:
                status, order = API.make_Trade(ticker[curTrack], 1, 0, curLot)
                print(status)
                print(order)
                orderTrack = 0
            if lastMode == MODELIST[2]:
                if orderId != -1 and olen > 0:
                    close_trade(orderId)
                    orderTrack = 0
            doAct = False
```

From the code perspective we covered all significant parts. Important to note, remember as requests to API are limited, the data is refreshed every some time (4-5 seconds In example).

DEMOS:

2023-07-11 08-26-54.mkv

IMG_3110.MOV

2023-07-11 08-28-52.mkv

IMG_3111.MOV