CPE316 – Embedded Systems Final Project Report

Smart drop off box

Semester II (2021-2022)

|  |  |
| --- | --- |
| **Team [Team x] members:** | |
| * Yavuz Çelikel 1810213041 | * Abedelkareem Abusafia 1810213547 |
| * Mehmet Can Özcan 1810213067 | * Yehia Akeela 1810213011 |
| * Batuhan Yamantürk 1810213068 | * Mehmethan Ozansoy 1810213047 |
| * Tolga Rodoplu 1810213040 |  |

Contents

[1. Introduction 3](#_Toc105038065)

[2. Related Works 3](#_Toc105038066)

[3. Project Design. 3](#_Toc105038067)

[3.1. Project Mechanism 3](#_Toc105038068)

[4. Conclusion and Results 4](#_Toc105038069)

[5. Lesson Learnt 4](#_Toc105038070)

[6. References 4](#_Toc105038071)

[7. Attachments 4](#_Toc105038072)

[7.1. Project Images 4](#_Toc105038073)

[7.2. Code Modules 6](#_Toc105038074)

## Introduction

Our product simply is a password secured box to deliver your packages. The courier will get the package with a password of the box, Enter the password put the package and last close it. You will open the box whenever you want without need of direct contact with the courier. The whole process will be traced and owner will be notified using telegram bot.

## Related Works

A previous product of Eufy company drop of box.

## Project Design.

### Project Mechanism

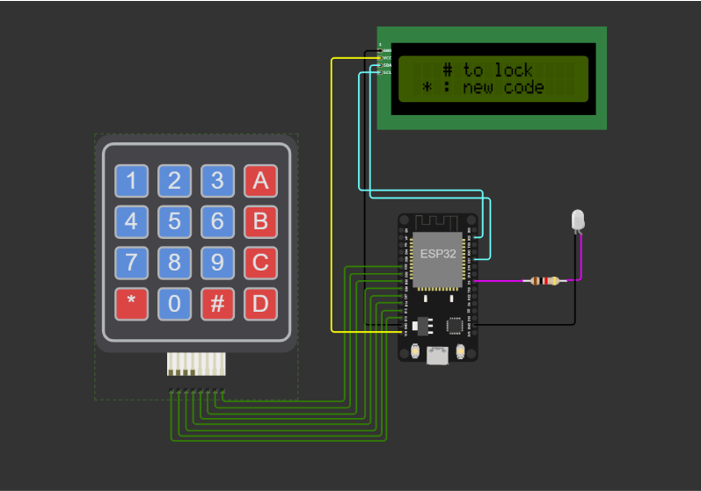
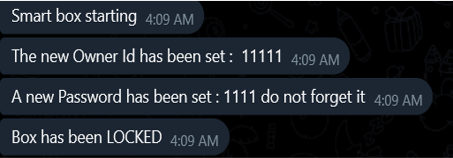
 When owner firstly get the product, he will set an ID to use for security restrictions and set the initial password (can be changed late) and both ID and password will be kept in telegram's bot conversation (track them).to open the box you need to enter the password correctly. in order to change the password, the user must enter correct ID. When the box is locked the circuit of led (Electromagnetic Lock) will be closed ,the user will be notified in telegram and when correct password entered the circuit will be opened (again the user will be notified). The project uses a lot if real-time error handling in code (password mismatch mismatch etc.).

Figure Telegram Bot

Figure set Id

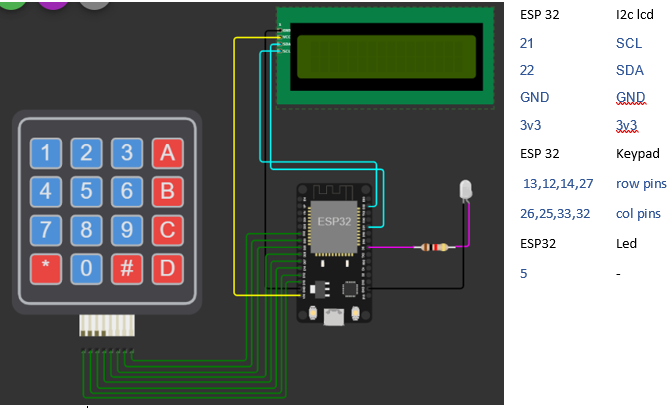


Figure Ports

## Conclusion and Results

The product works well.it can be modified further more adding hardware pieces like camera and memory. More functionalities for telegram bot can be added. But overall, the product logic is strong and achieved our main goals.

## Lesson Learnt

The field of Embedded systems is very wide and perfectly resemble the idea of Engineering which is finding solutions for problems. Taking this course widen our vision about what a small piece of code written in a simple hardware component can solve critical problems.

## References

https://docs.wokwi.com/parts

## Attachments

### Project Images

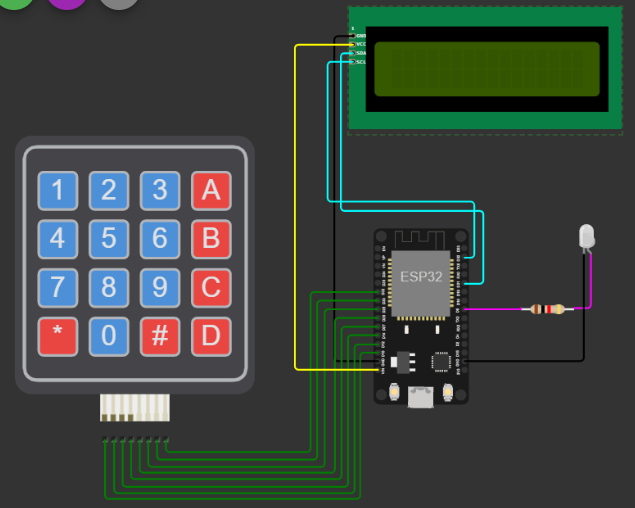


Figure Box is Opened (led off)

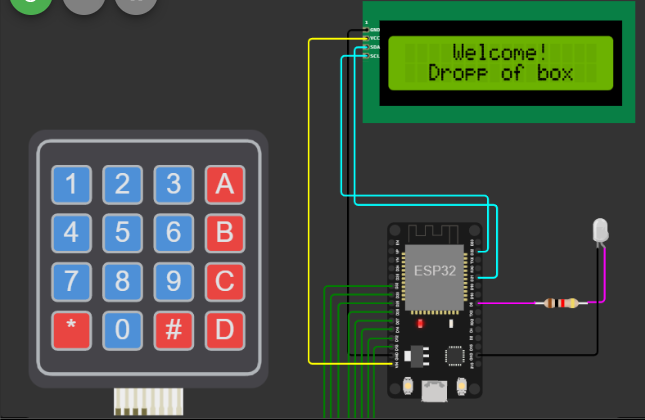


Figure Welcome massage

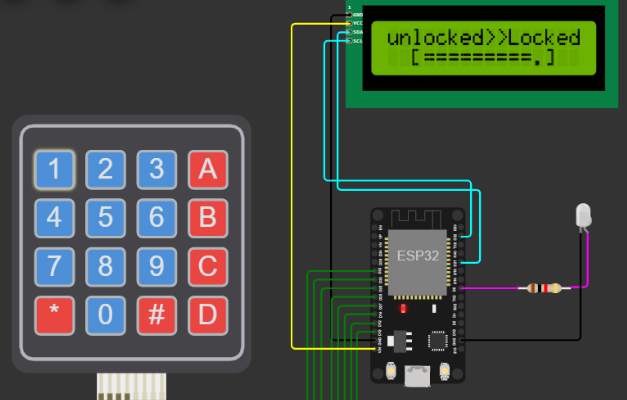


Figure Box is being locked

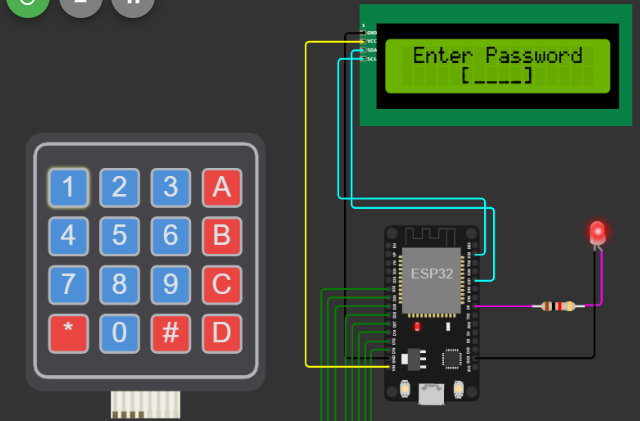


Figure Box is Locked (led on)

### Code Modules

#### main.py

from time import sleep\_ms, ticks\_ms,sleep

from machine import I2C, Pin,PWM

from i2c\_lcd import I2cLcd

from keypad import keypad

import utelegram

import network

import time

#Global Variables

isLocked = False

boxCode = ''

hasCode = False

ownerId = ''

hasOwner = False

#LCD initialization

AddressOfLcd = 0x27

i2c = I2C(scl=Pin(22), sda=Pin(21), freq=400000)

lcd = I2cLcd(i2c, AddressOfLcd, 2, 16)

#led initialization

led = Pin(5,Pin.OUT)

led.value(0)

#KeyPad initialization

rowPins = [13,12,14,27]

colPins = [26,25,33,32]

keypad = keypad(rowPins,colPins)

#Network & Bot

print("Connecting to WiFi", end="")

sta\_if = network.WLAN(network.STA\_IF)

sta\_if.active(True)

sta\_if.connect('Wokwi-GUEST', '')

while not sta\_if.isconnected():

  print(".", end="")

  time.sleep(0.1)

print(" Connected!")

TOKEN = '5155092636:AAH5O0HYIHyP5tSySRVDLpDxGEECQOahKCc'

chat = '537775139'

bot = utelegram.Bot(TOKEN)

def lock():

  global isLocked

  isLocked = True

  led.value(1)

  bot.send\_message(chat,'Box has been LOCKED')

def unlock():

  global isLocked

  global boxCode

  isLocked = False

  led.value(0)

  bot.send\_message(chat,'Box has been OPENED')

def check\_password(userCode) :

  global boxCode

  if (boxCode != ''):

    if (userCode == boxCode):

      return True

  else :

    return False

def check\_owner(userId) :

  global ownerId

  if (ownerId != ''):

    if (userId == ownerId):

      return True

  else :

    return False

def showStartupMessage() :

  lcd.move\_to(4, 0)

  lcd.putstr("Welcome!")

  sleep\_ms(200)

  lcd.move\_to(0, 1)

  massage = "  Dropp of box"

  for i in range (0,14):

    lcd.putchar(massage[i])

    sleep\_ms(100)

  bot.send\_message(chat,'Smart box starting')

  sleep(0.2)

def inputSecretCode() :

  lcd.move\_to(5, 1)

  lcd.putstr("[\_\_\_\_]")

  lcd.move\_to(6, 1)

  result = ""

  while (len(result) < 4) :

    key = keypad.getKey()

    if (key >= '0' and key <= '9') :

      lcd.putchar('\*')

      result += key

  return result

def inputOwner() :

  lcd.move\_to(5, 1)

  lcd.putstr("[\_\_\_\_\_]")

  lcd.move\_to(6, 1)

  userInput = ""

  while (len(userInput) < 5) :

    key = keypad.getKey()

    if (key != '\*' and key != '#') :

      lcd.putchar('\*')

      userInput += key

  return userInput

def waitScreen(delayMil) :

  lcd.move\_to(2, 1)

  lcd.putstr("[..........]")

  lcd.move\_to(3, 1)

  for i in range (0,10):

    sleep\_ms(delayMil)

    lcd.putstr('=')

def setNewCode() :

  global boxCode

  global hasCode

  lcd.clear()

  lcd.move\_to(0, 0)

  lcd.putstr("Enter new code:")

  newCode = inputSecretCode()

  lcd.clear()

  lcd.move\_to(0, 0)

  lcd.putstr("Confirm new code")

  confirmCode = inputSecretCode()

  if (newCode == confirmCode):

    boxCode = newCode

    hasCode = True

    waitScreen(200)

    bot.send\_message(chat,"{} {} {}".format('A new Password has been set :', boxCode,'do not forget it'))

    return True

  else :

    lcd.clear()

    lcd.move\_to(1, 0)

    lcd.putstr("Code mismatch")

    lcd.move\_to(0, 1)

    lcd.putstr("Box not locked!")

    sleep\_ms(700)

    return False

def setOwner() :

  global hasOwner

  global ownerId

  lcd.clear()

  lcd.move\_to(0, 0)

  lcd.putstr("Enter owner Id:")

  newOwner = inputOwner()

  lcd.clear()

  lcd.move\_to(0, 0)

  lcd.putstr("   Confirm Id")

  confirmID = inputOwner()

  if (newOwner == confirmID):

    ownerId = newOwner

    hasOwner = True

    waitScreen(200)

    bot.send\_message(chat,"{} {}".format('The new Owner Id has been set : ', ownerId))

    return True

  else :

    lcd.clear()

    lcd.move\_to(1, 0)

    lcd.putstr("ID mismatch")

    lcd.move\_to(1, 1)

    lcd.putstr("ID not set")

    sleep\_ms(700)

    return False

def showUnlockMessage() :

  lcd.clear()

  lcd.move\_to(4, 0)

  lcd.putstr("Unlocked!")

  sleep(0.5)

def boxUnlockedLogic():

  global boxCode

  global isLocked

  global hasCode

  global ownerId

  global hasOwner

  lcd.clear()

  lcd.move\_to(2, 0)

  lcd.putstr(" # to lock")

  newOwnerNeeded = True

  newCodeNeeded = True

  if (hasCode == True and hasOwner == True):

    lcd.move\_to(0, 1)

    lcd.putstr(" \* : new code")

    newCodeNeeded = False

    newOwnerNeeded = False

  key = keypad.getKey()

  while (key != '\*' and key != '#') :

    key = keypad.getKey()

  readyToLock = True

  if (key == '\*' ) :

    if (hasOwner):

      lcd.clear()

      lcd.clear()

      lcd.putstr("    Enter Id")

      userInput = inputOwner()

      rightId = check\_owner(userInput)

      waitScreen(300)

      if (rightId) :

        readyToLock = setNewCode()

      else :

        lcd.clear()

        lcd.move\_to(0, 0)

        lcd.putstr("ID mismatch")

        waitScreen(700)

  if (newOwnerNeeded):

    readyToLock = setOwner()

  if (newCodeNeeded == True and hasOwner) :

    readyToLock = setNewCode()

  '''

  if (newOwnerNeeded == True and hasCode) :

    readyToLock = setOwner()

  '''

  if (readyToLock == True) :

    lcd.clear()

    lcd.move\_to(0, 0)

    lcd.putstr("unlocked")

    lcd.putstr(">>")

    lcd.putstr("Locked")

    waitScreen(300)

    lock()

def boxLockedLogic() :

  lcd.clear()

  lcd.putstr("  Box Locked! ")

  sleep(1)

  lcd.clear()

  lcd.putstr(" Enter Password")

  userCode = inputSecretCode()

  unlockedSuccessfully = check\_password(userCode)

  waitScreen(300)

  if (unlockedSuccessfully) :

    showUnlockMessage()

    unlock()

  else :

    lcd.clear()

    lcd.move\_to(0, 0)

    lcd.putstr("Access Denied!")

    waitScreen(700)

## main calls

showStartupMessage()

while True:

  if (isLocked == True) :

    boxLockedLogic()

  else :

    boxUnlockedLogic()

* + 1. keypad.py

from machine import Pin

from time import sleep

class keypad :

    def \_\_init\_\_(self, r, c):

        # CONSTANTS

        self.KEY\_UP   = 0

        self.KEY\_DOWN = 1

        self.keys = [['1', '2', '3', 'A'], ['4', '5', '6', 'B'], ['7', '8', '9', 'C'], ['\*', '0', '#', 'D']]

        # Pin names for Pico

        self.rows = r

        self.cols = c

        # set pins for rows as outputs

        self.row\_pins = [Pin(pin\_name, mode=Pin.OUT) for pin\_name in self.rows]

        # set pins for cols as inputs

        self.col\_pins = [Pin(pin\_name, mode=Pin.IN, pull=Pin.PULL\_DOWN) for pin\_name in self.cols]

    def init(self):

      for row in range(0,4):

          for col in range(0,4):

              self.row\_pins[row].off()

    def scan(self,row, col):

      """ scan the keypad """

     # set the current column to high

      self.row\_pins[row].on()

      self.key = None

      # check for keypressed events

      if self.col\_pins[col].value() == self.KEY\_DOWN:

          self.key = self.KEY\_DOWN

      if self.col\_pins[col].value() == self.KEY\_UP:

          self.key = self.KEY\_UP

      self.row\_pins[row].off()

      # return the key state

      return self.key

         #print("starting")

        # set all the columns to low

    def getKey(self):

      self.init()

      self.last\_key\_press  = None

      while self.last\_key\_press == None :

          for row in range(4):

              for col in range(4):

                  self.key = self.scan(row, col)

                  if self.key == self.KEY\_DOWN:

                      #print("Key Pressed", keys[row][col])

                      self.last\_key\_press = self.keys[row][col]

                      #i = i + 1

                      sleep(0.1)

                  self.init()

      return self.last\_key\_press

7.2.3

import ujson, ure, time, gc, urequests, \_thread

from machine import Timer

from time import sleep

class Bot():

    '''

    Base class for interacting with telegram api

    '''

    def \_\_init\_\_(self, token):

        self.url = 'https://api.telegram.org/bot' + token

        self.last\_update = 0

        self.command\_handlers = {}

        self.callback\_handlers = {}

        self.message\_handlers = {}

        self.conversation\_handlers = []

        self.\_get\_updates()

    def \_get\_updates(self):

        '''

        Gets all the updates from the telegram api and stores

        latest id for next iteration

        '''

        parameters = {

            'offset': self.last\_update + 1,

            'timeout': 2,

            'allowed\_updates': ['messages']

            }

        try:

            response = urequests.post(self.url + '/getUpdates', json=parameters)

            data = response.json()

            response.close()

            if data['result']:

                self.last\_update = data['result'][-1]['update\_id'] #storing last update id

                return [Update(self, update) for update in data['result']]

            return None

        except Exception as e:

            print('\_get\_updates: ',e)

            return  None

    def \_handle\_update(self, update):

        '''

        Function that chooses the right function to handle the update,

        based on the previously defined handlers

        '''

        text = update.message['text']

        if update.is\_callback:

            self.callback\_handlers[update.callback\_data](update)

        if text.startswith('/'): #is a command

            #get first word (useful for future implementation of commands with arguments)

            command = text.split(' ')[0].replace('/','')

            for c in self.conversation\_handlers:

                if command in c.steps[c.active][0].keys():

                    next\_step = c.steps[c.active][0][command](update)

                    c.go\_to\_step(next\_step)

                    return

            if command in set(self.command\_handlers.keys()):

                self.command\_handlers[command](update)

                return

        else:

            for c in self.conversation\_handlers:

                    for expression in c.steps[c.active][1].keys():

                        if ure.match(expression, text):

                            next\_step = c.steps[c.active][1][expression](update)

                            c.go\_to\_step(next\_step)

                            return

            for expression in set(self.message\_handlers.keys()):

                #handling messagges

                if ure.match(expression, text):

                    self.message\_handlers[expression](update)

                    return

    def \_read(self):

        '''

        main bot read function

        '''

        updates = self.\_get\_updates()

        if updates:

            for update in updates:

                self.\_handle\_update(update)

        gc.collect() #in case automatic gc is disabled

        return

    def \_loop(self, period=100):

        while True:

            self.\_read()

            sleep(1)

    def start\_loop(self, main\_function=None, args=(), period=100):

        """

        main function used to start the bot in a different thread.

        """

        if main\_function:

            \_thread.start\_new\_thread(main\_function, args)

        \_thread.start\_new\_thread(self.\_loop(), (period,))

    def add\_message\_handler(self, regular\_expression):

        '''

        Decorator to add a message handler with regex validation

        '''

        def decorator(function):

            self.message\_handlers[regular\_expression] = function

        return decorator

    def add\_callback\_handler(self, callback\_data):

        '''

        Decorator to add a callback handler

        '''

        def decorator(function):

            self.callback\_handlers[callback\_data] = function

        return decorator

    def add\_command\_handler(self, command):

        '''

        Decorator to add a command handler, (write command without '/' as argument)

        '''

        def decorator(function):

            self.command\_handlers[command] = function

        return decorator

    def add\_conversation\_handler(self, conversation):

        '''

        Decorator to add a conversation handler

        '''

        self.conversation\_handlers.append(conversation)

    def send\_message(self, chat\_id, text, parse\_mode='MarkdownV2', reply\_markup=None):

        parameters = {

            'chat\_id': chat\_id,

            'text': text.replace('.', '\.'),

            'parse\_mode': parse\_mode

        }

        if reply\_markup:

            parameters['reply\_markup'] = reply\_markup.data

        try:

            message = urequests.post(self.url + '/sendMessage', json=parameters)

            assert message

            message.close()

        except Exception:

            print('message not sent')

    def update\_message(self, chat\_id, message\_id, text, parse\_mode='MarkdownV2', reply\_markup=None):

        parameters = {

            'chat\_id': chat\_id,

            'message\_id' : message\_id,

            'text': text,

            'parse\_mode': parse\_mode,

        }

        if reply\_markup:

            parameters['reply\_markup'] = reply\_markup.data

        try:

            message = urequests.post(self.url + '/editMessageText', json=parameters)

            print(message.text)

            assert message

            message.close()

        except Exception:

            print('update not sent')

class Conversation():

    """

    Conversation class used for conversations with multiple steps

    STEPS MUST BE DEFINED AT INITIALIZATION, EACH STEP CAN HAVE MULTIPLE HANDLERS

    ENTRY STEP IS ADDED BY DEFAULT AND IS USED TO START THE CONVERSATION

    every function used as a handler should return the next conversation step

    """

    def \_\_init\_\_(self, steps: list = []):

        self.END = 0

        self.steps = {

            'ENTRY': [{},{}]

        }

        self.active = 'ENTRY'

        for step in steps:

            self.steps[step] = [{},{}]

    def add\_command\_handler(self, step, command):

        '''

        Decorator to add a command handler to a specific step,

        (write command without '/' as argument)

        '''

        def decorator(function):

            self.steps[step][0][command] = function

            return decorator

        def add\_message\_handler(self, step, regular\_expression):

        '''

        Decorator to add a message handler to a specific step,

        with regex validation

        '''

        def decorator(function):

            self.steps[step][1][regular\_expression] = function

        return decorator

    def go\_to\_step(self, step):

        if step == 0:

            self.active = 'ENTRY'

        elif step in self.steps.keys():

            self.active = step

        else:

            print('[ERROR] No step named {s} defined, staying at current step'.format(step))

    def end(self):

        self.active = 'ENTRY'

class ReplyKeyboardMarkup():

    '''

    class used to as custom reply\_markup to send custom keyboards

    '''

    def \_\_init\_\_(self, keyboard, resize\_keyboard=False, one\_time\_keyboard=False, selective=False):

        self.data = {

            'keyboard': [[k.data for k in row] for row in keyboard],

            'resize\_keyboard': resize\_keyboard,

            'one\_time\_keyboard': one\_time\_keyboard,

            'selective': selective

            }

class InlineKeyboardMarkup():

    '''

    class used to as custom reply\_markup to send custom keyboards

    '''

    def \_\_init\_\_(self, keyboard):

        self.data = {

            'inline\_keyboard': [[k.data for k in row] for row in keyboard]

            }

class KeyboardButton():

    '''

    class used to create button objects used with ReplyKeyboardMarkup

    '''

    def \_\_init\_\_(self, text, request\_contact=False, request\_location=False):

        self.data = {

            'text': text,

            'request\_contact': request\_contact,

            'request\_location': request\_location

            }

class InlineKeyboardButton():

    '''

    class used to create button objects used with ReplyKeyboardMarkup

    '''

    def \_\_init\_\_(self, text,url="",callback\_data = ""):

        self.data = {

            'text': text,

            'url': url,

            'callback\_data' : callback\_data

            }

class Update():

    '''

    class with basic methods for updates

    '''

    def \_\_init\_\_(self, b, update):

        self.update\_id = update['update\_id']

        self.bot = b

        self.is\_callback = False

        self.callback\_data = ""

        try:

            if update['callback\_query']:

                    print("IS A CALLBACK")

                    self.is\_callback = True

                    self.message = update['callback\_query']['message']

                    self.callback\_data = update['callback\_query']['data']

        except KeyError as e:

            print("Not a Callback")

            self.message = update['message']

    def reply(self, text, parse\_mode='MarkdownV2', reply\_markup=None):

        self.bot.send\_message(self.message['chat']['id'], text, parse\_mode=parse\_mode, reply\_markup=reply\_markup)

    def edit(self,text, parse\_mode='MarkdownV2', reply\_markup=None):

        self.bot.update\_message(self.message['chat']['id'] , self.message['message\_id'], text, parse\_mode= parse\_mode , reply\_markup= reply\_markup)