

## توضیحات کد – Documentation

در ابتدا به توضیح دادن فرآیند انجام کار ها و سپس به توضیح هر کدام از فایل ها و توابع پرداخته می شود.

- توضیح فرآیند و ساختار کلی برنامه

در ابتدا بعد از اجرای فایل `main.py` صفحه ی اصلی برنامه باز می شود که در آن تعدادی ابزار در نوار سمت چپ مربوط به دستکاری عکس و تعدادی دکمه در نوار پایین مربوط به آپلود عکس و کار با دوربین مشاهده می شود. قالب طراحی این صفحه از فایل `main.ui` که در پوشه ی `UI Files` واقع شده است بارگزاری می شود. برای هر صفحه ای که باز می شود یک کلاس از نوع `Qwidget` تعریف می شود. برای اطلاعات بیشتر در رابطه با کلاس های کتابخانه ی `pyQt5` و ساخت آنها با برنامه ی `QtDesign` می توان لینک زیر را مشاهده کرد.

[https://www.tutorialspoint.com/pyqt5/pyqt5\\_using\\_qt\\_designer.htm](https://www.tutorialspoint.com/pyqt5/pyqt5_using_qt_designer.htm)

بعد از به نمایش در آمدن این صفحه تابع `init` اجرا می شود. این تابع وظیفه ی انجام تنظیمات اولیه ی این صفحه را بر دوش دارد. از کارهای انجام شده در این تابع می توان به گزینه های زیر اشاره نمود:

- مقدار دهی اولیه متغیر ها
  - بارگزاری فایل `ui` و آیکون پنجره
  - پیدا کردن تمام اجزای داخل صفحه و نسبت دادن آنها به یک متغیر (دکمه ها و دیگر اجزا) (این کار با استفاده از تابع `findChild` انجام می شود).
  - ایجاد ارتباط بین المان های داخل صفحه و توابع کارایی آنها (مثلا به هنگام کلیک شدن روی هر دکمه) (این کار با استفاده از تابع `connect` انجام می شود. به این صورت که هر گاه اتفاق مورد انتظار بیفتد تابع نسبت داده شده اجرا می شود).
  - ساختن فضای ایجاد نمودار های هیستوگرام
- بدین صورت تمام المان ها در برنامه قابل دسترس می شوند و همچنین به توابع مربوط به خودشان متصل می شوند. هر گاه المان جدیدی به ظاهر برنامه اضافه شود قبل از نوشتن تابع کارکرد آن باید مراحل بالا برای آن المان طی شود.
- به غیر از این صفحه ۳ پنجره ی دیگر نیز در برنامه وجود دارد که هنگام فراخوانی هر کدام از آنها دقیقا به همین منوال در تابع `init` مربوط به کلاس خودشان کار های بالا به ترتیب انجام می شود. این سه پنجره عبارتند از:

- پنجره ی `alert` برای نشان دادن یک متن جهت هشدار. مثلا به هنگام چک کردن اتصال دوربین
- پنجره ی `Camera_Tools` که با کلیک روی دکمه ی `camera tools` فراخوانده می شود و حاوی اجزای مربوط به تنظیمات دوربین و زمان بندی عکس برداری می باشد.
- پنجره ی `channel edit` که با کلیک روی دکمه ی `channel edit` فراخوانده می شود و برای تغییر دادن بازه ی رنگی کانال های قرمز، آبی و سبز تصویر به کار می رود.

همچنین علاوه بر فایل های ui ، در پوشه ی style فایل هایی وجود دارد که برخی از تنظیمات ظاهری هر صفحه ها مشخص می کند. پوشه ی دیگری با نام DigiCam در بین فایل های برنامه دیده می شود که در آن فایل های مربوط به اتصال با دوربین قرار داده شده است. یکی از آنها تعریف کلاس Camera می باشد. در پنجره ی main (برای دکمه ی take picture ) و همچنین پنجره ی camera tools که نیاز به ارتباط با دوربین داریم یک نمونه از این کلاس نگهداری می شود. فایل دیگر هم با نام FileSystem.py به انجام کارهای سیستمی ارتباط با دوربین می پردازد که به جزئیات آن کاری نداریم.

#### - توضیحات فایل های برنامه

- فایل main.py : این فایل حاوی تمام محتویات صفحه ی اصلی می باشد. پنجره ی اصلی برنامه به صورت یک کلاس ذخیره شده است. تمام اجزای صفحه متغیر های محلی این کلاس و کارهای قابل انجام در پنجره همگی متد های این کلاس هستند.
- فایل Image.py : در این فایل تمام مشخصات کلاس image که بیشتر مربوط به انجام عملیات های مختلف روی عکس می باشد را نگه می دارد. هر کدام از این عملیات به صورت یک تابع منحصر به فرد تعریف شده است.
- فایل libraries.py : در این فایل تمام کتابخانه هایی که استفاده شده و همچنین آدرس فایل اجرایی برنامه ی DigiCamControl جهت اتصال به دوربین ذخیره شده است. این فایل در همه فایل های دیگر import شده است.
- فایل های setup.py و build.py : این دو فایل مربوط به برنامه ی DigiCamControl هستند و کاری به محتوای آنها نداریم.
- فایل sub\_window\_classes.py : در این فایل کد های مربوط به پنجره های فرعی برنامه نوشته شده است. هر کدام از این پنجره ها به صورت یک کلاس نوشته شده اند.
- فایل threads.py : در بعضی از فرآیندها نیاز به همگام سازی انجام کارها می باشد. مثل هنگامی که دکمه ای کلیک می شود و قبل از تمام شدن فرآیند مورد نظر نیاز به انجام فرآیند دیگری است. اینگونه فرآیند ها هر کدام به صورت یک کلاس و در این فایل نوشته شده است.
- فایل Camera.py : این فایل در پوشه ی DigiCam واقع در مسیر اصلی برنامه قرار دارد. این فایل رابط بین برنامه اصلی و دوربین است و تمام دستور هایی که لازم به فرستاده شدن به دوربین دارند از این فایل فرستاده می شوند.

- انجام کار ها داخل برنامه به صورت زیر انجام می پذیرد:

۱. در صفحه ی اصلی با کلیک کردن دکمه ی Browse image می توان یک عکس روی صفحه آپلود کرد. این کار با تابع زیر انجام می شود.

در این تابع pic حاوی لیستی از آدرس فایل های انتخاب شده است و چون فقط یک عکس میخواهیم لود کنیم آدرس خانه ی اول آن را ذخیره و آن عکس را در بارگزاری می کنیم. عکس در صفحه یک نمونه از کلاس Image می باشد. در خط های بعدی عکس پشتیبان (برای زمان هایی که دکمه ی reset کلیک می شود) ذخیره می شود، عکس برای پنجره ی channel edit فرستاده می شود و همچنین تابع update\_image صدا زده می شود که کار آن نمایش دوباره ی عکس و همچنین به روز رسانی نمودار های هیستوگرام می باشد.

```
# Browsing files
def on_click(self):
    pic, _ = QFileDialog.getOpenFileNames(self, "Choose Image File", "",
                                          "Image Files (*.jpg *.png
                                          *.jpeg *.ico);;All Files (*)")
    if pic:
        self.pictureName = pic[0].split('/')[0]
        self.image = Image(cv2.imread(pic[0], 1) )
        self.channel_edit_window.set_image(self.image)
        self.backup = cv2.imread(pic[0], 1)
        self.update_image()
```

فایل main.py خط ۱۶۷

۲. دکمه های بالای نمودار ها در صفحه ی اصلی

از این دکمه ها برای اعمال فیلتر های B/W ، invert ، Auto Contrast ، Auto Sharpen ، چرخش و قرینه کردن تصویر و همچنین زوم و آن زوم استفاده می شود. به غیر از دکمه های مربوط به زوم کردن چرخه ی کلی تمام فرآیند ها مشابه یکدیگر می باشد. به این صورت که بعد از کلیک ابتدا تابع مربوطه در پنجره ی اصلی اجرا

```
# connecting buttons to Image class
# Flips buttons function
def vertical_flip(self):
    self.image.v_flip()
    self.update_image()
def horizontal_flip(self):
    self.image.h_flip()
    self.update_image()

# Rotating buttons function
def clockwise_rotate(self):
    self.image.rotate_clockwise()
    self.update_image()
def unclockwise_rotate(self):
    self.image.rotate_unclockwise()
    self.update_image()
```

فایل main.py خط ۲۱۶

```
# Invert button function
def invert(self):
    self.image.auto_invert()
    self.update_image()
# Auto Contrast button function
def auto_contrast(self):
    self.image.auto_contrast()
    self.update_image()
# Auto Sharpen button function
def auto_sharpen(self):
    self.image.auto_sharpen()
    self.update_image()
# Black and White button function
def black_and_white(self):
    self.image.blackAndWhite()
    self.update_image()
```

فایل main.py خط ۲۳۳

می‌شود و در آن تابع فرآیند مورد نظر روی عکسی که یک آبجکت از کلاس Image است اجرا می‌شود و سپس تصویر روی پنجره به همراه ۳ نمودار هیستوگرام آپدیت می‌شوند.

اکنون به توضیح دادن تابع update\_image() و سپس به توضیح توابع داخل کلاس Image می‌پردازیم.

هر زمان که تغییری روی عکس صورت بگیرد تابع نام برده شده صدا زده می‌شود. این تابع وظیفه‌ی به روز رسانی پنجره‌ی اصلی که کاربر آن را می‌بیند را بر عهده دارد.

```
# updating the screen
def update_image(self):
    self.current_image =
QPixmap(qimage2ndarray.array2qimage(cv2.cvtColor(self.image.get_image(), cv2.COLOR_BGR2RGB)))
    self.scene = QGraphicsScene()
    self.current_image = self.current_image.scaledToHeight(864)
    self.scene_img = self.scene.addPixmap(self.current_image)
    self.imageArea.setScene(self.scene)
    self.update_histograms()
```

فایل main.py خط ۲۰۷

اگر به ساختار نشان دادن عکس در صفحه نگاه بیندازیم (در کتابخانه‌ی PyQt5) فهمیده می‌شود که برای این کار لازم است یک آبجکت از نوع QGraphicsView داشته باشیم که ما آن را در زمان ساختن ui قرار داده ایم. حال باید یک آبجکت از نوع QGraphicsScene داشته باشیم که عکس روی آن قرار بگیرد. (با تابع addPixmap) سپس باید آبجکت دوم را روی آبجکت اول ست کنیم که این کار با تابع setScene انجام داده شده است. نکته‌ی دیگر آنکه برای عکس‌ها باید به صورت QImage ذخیره شده باشند تا بتوان بدون مشکل آن‌ها را در پنجره مورد نظر نمایش داد. برای این کار هم از یک کتابخانه به نام QImage2ndarray استفاده کردیم تا np.ndarray ها را در زمان نمایش به فرمت خواسته شده تبدیل کند. به این صورت هر بار که تغییری روی عکس رخ می‌دهد ابتدا عکس جدید را از کلاس Image می‌گیریم. (self.image.get\_image()) سپس فرمت آن را تغییر می‌دهیم. یک scene جدید می‌سازیم. عکس را روی آن قرار می‌دهیم و scene جدید را روی QGraphicsView که داشتیم ست می‌کنیم. در این بین نکته‌ی حائز اهمیت ابعاد عکس می‌باشد. ابعاد عکس خروجی دوربین ۳۴۵۶ \* ۵۱۸۴ پیکسل می‌باشد. این ابعاد بسیار بزرگ هستند و نمایش آنها در پنجره‌ی اصلی امکان‌پذیر نیست. ابعاد فضای مورد نظر برنامه برای نشان دادن عکس ۸۶۴ \* ۱۲۹۶ پیکسل می‌باشد. (یعنی یک چهارم عکس اصلی) به همین جهت در هنگام نشان دادن عکس آن را scale می‌کنیم. لازم به ذکر است عکس با همان ابعاد اصلی در برنامه ذخیره می‌شود و تمام تغییرات هم روی عکس با ابعاد اصلی اعمال می‌شود و فقط در زمان نشان دادن است که ابعاد عکس کوچک می‌شوند.

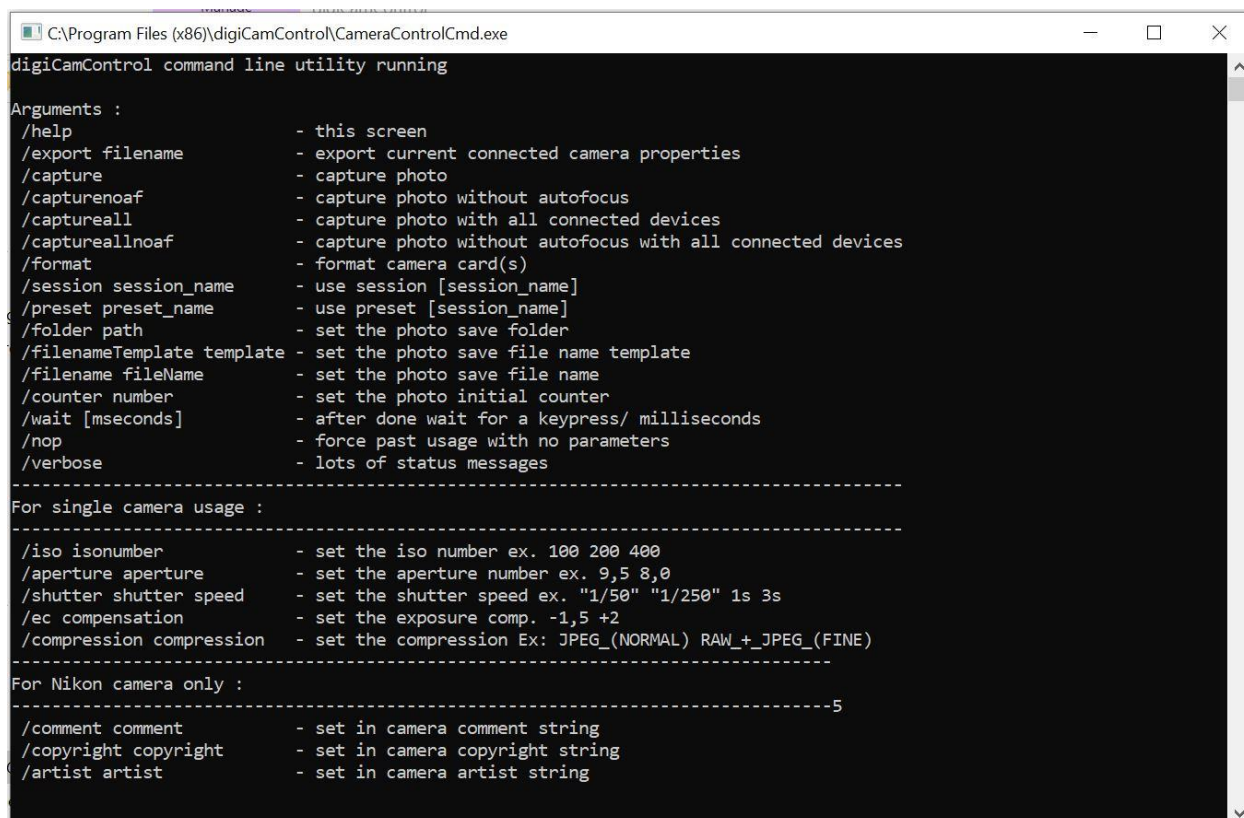
### ۳. دکمه ی Camera Connectivity Check

در تابع مربوط به این دکمه چک می شود که دوربین متصل باشد. این کار با تابع `check_camera()` انجام می شود. به این صورت که دستور ذخیره کردن اطلاعات دوربین به دوربین فرستاده می شود. اگر در این بین به مشکلی خوردیم یعنی دوربینی وصل نیست اما اگر اطلاعات بدون مشکل ذخیره شوند یعنی دوربین متصل داریم. چون دستور مستقیم برای چک کردن اتصال دوربین نداریم مجبور به استفاده از این ایده شده ایم.

```
# checking camera connectivity
def check_camera(self):
    self.camera.show_camera_info("connectivity_check_@.txt")
    try:
        f = open("connectivity_check_@.txt")
        f.close()
        os.remove("connectivity_check_@.txt")
        self.alert_sub_window.set_text("Camera is Available!")
        self.alert_sub_window.show()
    except IOError:
        self.alert_sub_window.set_text("Camera is not Connected!")
        self.alert_sub_window.show()
```

فایل main.py خط ۱۸۲

به طور کل تمام دستوراتی که میتوان به دوربین داد از فایل `CameraControlcmd.exe` واقع در محل نصب برنامه ی DigiCam قابل مشاهده است. این دستورات به اقسام زیر هستند.



```
C:\Program Files (x86)\digiCamControl\CameraControlCmd.exe
digiCamControl command line utility running

Arguments :
/help                - this screen
/export filename     - export current connected camera properties
/capture            - capture photo
/capturenoaf        - capture photo without autofocus
/captureall         - capture photo with all connected devices
/captureallnoaf     - capture photo without autofocus with all connected devices
/format            - format camera card(s)
/session session_name - use session [session_name]
/preset preset_name - use preset [session_name]
/folder path        - set the photo save folder
/fileNameTemplate template - set the photo save file name template
/fileName fileName  - set the photo save file name
/counter number     - set the photo initial counter
/wait [mseconds]    - after done wait for a keypress/ milliseconds
/nop                - force past usage with no parameters
/verbose            - lots of status messages
-----
For single camera usage :
-----
/iso isonumber      - set the iso number ex. 100 200 400
/aperture aperture  - set the aperture number ex. 9,5 8,0
/shutter shutter speed - set the shutter speed ex. "1/50" "1/250" 1s 3s
/ec compensation     - set the exposure comp. -1,5 +2
/compression compression - set the compression Ex: JPEG_(NORMAL) RAW+_JPEG_(FINE)
-----
For Nikon camera only :
-----
/comment comment    - set in camera comment string
/copyright copyright - set in camera copyright string
/artist artist       - set in camera artist string
```

#### ۴. دکمه ی take picture

در تابع مربوط به این دکمه ابتدا چک می شود که دوربین وصل باشد. حال برای عکس گرفتن بعد از چک کردن اتصال دوربین عملیات های مربوط به مرتب کردن نام عکس ها را انجام می دهیم و سپس دستور عکس گرفتن را با صدا کردن یک تابع از کلاس camera ارسال می کنیم. بعد از آن همان عکس را روی پنجره ی اصلی بارگزاری می کنیم تا در لحظه آن را ببینیم. سپس تابع به روز رسانی صفحه را که پیشتر توضیح دادیم صدا می کنیم.

```
# take picture command
def take_picture(self):
    self.check_camera()
    # set the name of the picture
    self.picture_index = self.camera.capture_single_image(autofocus=True)
    self.pictureName = "Capture_"+str(self.picture_index)+".jpg"
    address = "captures\\Capture_"+str(self.picture_index)+".jpg"
    # save image and its backup
    self.image = Image(cv2.imread(address, 1))
    self.backup = cv2.imread(address, 1)
    self.update_image()
```

فایل main.py خط ۱۹۵

برای تغییر دادن دستور ها (به عنوان مثال تعیین انجام فوکوس دستی یا اتوماتیک قبل از عکس برداری ) باید تغییراتی در فایل camera.py انجام دهیم. به عنوان نمونه تابع Capture\_single\_image به شکل زیر کار می کند.

```
def capture_single_image(self, autofocus: bool = False) -> None:
    """Captures a single image. Iterates the image index to ensure a unique name
    for each image taken.

    Args:
        autofocus (bool, optional): [description]. Defaults to False."""
    # Make the correct camera command depending on autofocus being enabled.
    if autofocus:
        self.command_camera("/capture")
    else:
        self.command_camera("/capturenoaf")

    # Increment the image index
    self.image_index += 1
    return self.image_index-1
```

فایل Camera.py خط ۱۳۰

که در اینجا می توانیم دستوری که می خواهیم به دوربین بدهیم را به هر کدام از دستورات دیگری که می خواهیم (که در عکس صفحه قبل قابل مشاهده است) تغییر بدهیم. همچنین میتوانیم در این کلاس هر تابع جدیدی نیز بنویسیم. جلوتر با تابع های جدید که در این فایل اضافه شده آشنا می شویم.

## ۵. دکمه ی Camera Tools

همانطور که از نام این دکمه مشخص است با کلیک بر روی آن پنجره ی جدیدی باز می شود که در آن ابزار های کار با دوربین قابل مشاهده هستند.

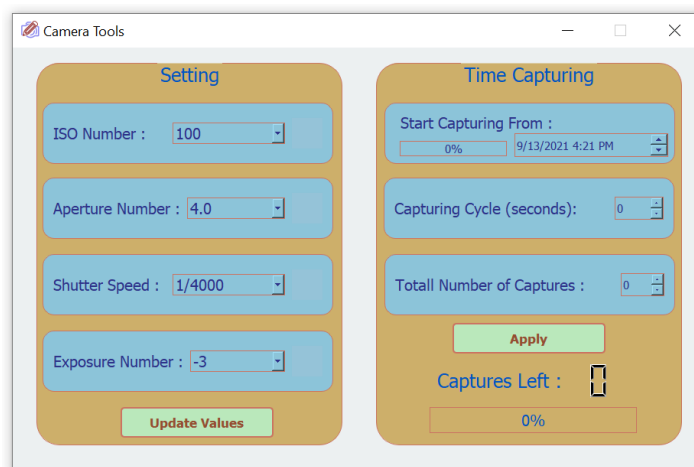
```
# camera setting window
def camera_setting(self):
    self.camera_setting_window.show()
```

فایل main.py خط ۱۷۸

هر کدام از پنجره های مرتبط با برنامه به صورت یک متغیر از کلاس بزرگ تر انتخاب شده اند و در اینجا بعد از کلیک روی دکمه صرفا آن پنجره را نمایش می دهیم.

با باز شدن پنجره ی جدید می توانیم به ۲ دسته دستور پردازیم. سمت چپی ها مربوط به تغییرات تنظیمات دوربین و سمت راستی ها هم مربوط به زمان بندی برای عکس گرفتن.

در این پنجره هم دقیقا مثل پنجره اصلی هنگام ساخته شدن همه متغیر های لازم ریست و ساخته می شوند و دکمه ها و دیگر اجزای صفحه هر یک به متغیری از پنجره جدید نسبت داده می شوند.



همانطور که میبینید در این پنجره ها اجزایی داریم. هر کدام از این اجزا هم به تابع های مستقیم خود وصل شده اند. حال به کارکرد هر کدام می پردازیم. با فشار دادن دکمه ی **update values** تابع زیر اجرا می شود. در این تابع ابتدا اتصال دوربین چک می شود. سپس اطلاعات وارد شده در هر کدام از فیلدهای مربوط به تنظیمات استخراج می شوند. در مرحله بعد یک پردازش جدید ساخته می شود که کار آن این است که ابتدا گیف های کنار هر مورد را که نشانه ی در حال لود بودن است را بزرگ کند (در ابتدا سایز این گیف خیلی کوچک است تا دیده نشود) سپس هر کدام از موارد تغییر تنظیمات را به دوربین ارسال کند و بعد از اتمام هر کدام یک سیگنال برا برنامه ی اصلی بفرستد تا اعلام کند کار این قسمت تمام شده و اندازه ی گیف به حالت قبلی برگردد. کدهای مربوط به پردازش هم در اینجا آمده است.

```
# derive input data from combo boxes for updating values
def retrieve_setting_data(self):
    camera_connectivity_state = self.check_camera()
    if (camera_connectivity_state=='yes'):
        iso = self.ISOComboBox.currentText()
        aperture = self.ApertureComboBox.currentText()
        exposure = self.ExpoComboBox.currentText()
        shutter = self.ShutterComboBox.currentText()
        speed = "1s"

    self.scalegif()
    self.LoadingThread = LoadingGifThread(iso, aperture, exposure, shutter, speed)
    self.LoadingThread.start()
    self.LoadingThread.gif1.connect(lambda: self.LoadingGif1.setScaledSize(QtCore.QSize(1, 1)))
    self.LoadingThread.gif2.connect(lambda: self.LoadingGif2.setScaledSize(QtCore.QSize(1, 1)))
    self.LoadingThread.gif3.connect(lambda: self.LoadingGif3.setScaledSize(QtCore.QSize(1, 1)))
    self.LoadingThread.gif4.connect(lambda: self.LoadingGif4.setScaledSize(QtCore.QSize(1, 1)))
    self.LoadingThread.alert.connect(self.show_updated_alert)
```

فایل sub\_window\_classes.py خط ۹۸

در انتهای کار پردازش نام برده شده دستوری به دوربین می فرستد که طبق آن تمام تنظیمات لحظه ای دوربین در یک فایل txt در آدرس مشخص شده ذخیره می شوند.

بعد از آن هم یک پنجره باز می شود و انجام موفق این مورد را اعلام می کند.



```

# QThread class for handling gif icons whenever update button pressed
class LoadingGifThread(QThread):
    # One signal per Each gif
    gif1 = pyqtSignal()
    gif2 = pyqtSignal()
    gif3 = pyqtSignal()
    gif4 = pyqtSignal()
    alert = pyqtSignal()

    def __init__(self, iso, aperture, exposure, shutter, speed) -> None:
        super(LoadingGifThread, self).__init__()
        self.camera = Camera(control_cmd_location=CAMERA_CONTROL_CMD_PATH,
save_folder="captures\\", collection_name="Capture")
        self.i = iso
        self.a = aperture
        self.o = exposure
        self.s = shutter
        self.p = speed

    def run(self):
        # Turn down each gifs after it commands
        self.camera.set_iso_number(self.i)
        self.gif1.emit()
        self.camera.set_aperture_number(self.a)
        self.gif2.emit()
        self.camera.set_exposure_comp_number(self.o)
        self.gif3.emit()
        self.camera.set_shutter_speed_number(self.s, self.p)

        # Saving Camera setting details each time settings change
        t = datetime.datetime.now()
        current_time = str(t.strftime("%d")) + "." + str(t.strftime("%m")) + "." +
str(t.strftime("%Y")) + "." + str(
        t.strftime("%H")) + "." + str(t.strftime("%M"))
        try:
            os.mkdir(os.path.join(os.getcwd(), "Camera-setting-records"))
        except OSError as error:
            pass
        record_name = "Camera-setting-records\\" + current_time + ".txt"
        self.camera.show_camera_info(record_name)
        self.gif4.emit()
        self.alert.emit()

```

فایل threads.py خط ۳

در بخش سمت چپ این پنجره اجزای دیگری وجود دارند که کارهای مربوط به عکس برداری زمان دار را انجام می دهند. با نحوه ی گرفتن یک عکس در یکی از تابع های پیشین آشنا شدیم. در اینجا هم پایه ی گرفتن عکس همان است و تنها تفاوت این است که برای راه شروع زمان عکس گرفتن، تعداد عکس ها و فاصله زمانی بین هر دو عکس ورودی میگیریم و با استفاده از یک سری پردازش که کار نشان داده زمان سپری شده روی **progress bar** ها و عکس برداری را همزمان با هم انجام می دهند. با توجه به زمان گرفتن هر عکس نام آن مشخص می شود و در فایل مربوطه ذخیره می شود.

بعد از کلیک روی دکمه ی **apply** تابعی با نام **startCapturing** شروع به اجرا شدن می کند.

```
# waiting until capturing cycle start time
def startCapturing(self):
    self.multi_capture_number = 0
    t = datetime.datetime.now()
    start_time = self.StartTime.dateTime().toPyDateTime()
    wait_to_start = int((start_time - t).total_seconds())

    self.waitForStartThread = WaitingToStartThread(wait_to_start)
    self.waitForStartThread.update.connect(self.setWaitProgressValue)
    self.waitForStartThread.end.connect(self.capturing)
    self.waitForStartThread.start()
```

فایل sub\_window\_classes.py خط ۲۶۲

همانطور که در تصویر مشخص است ابتدا مقدار زمانی که باید منتظر باشیم تا عکس برداری شروع شود را محاسبه می کنیم و سپس یک پردازش با نام **WaitingToStartThread** می سازیم و به عنوان ورودی مقدار منتظر ماندن را به آن می دهیم. این پردازش برای این طراحی شده است که همزمان با منتظر ماندن یکی از **progress bar** ها را جلو ببرد. کد های مربوط به این پردازش را اینجا مشاهده می کنید.

```
# QThread class for handling Waiting time between pressing apply button and start capturing
class WaitingToStartThread(QThread):
    update = pyqtSignal(int)
    end = pyqtSignal()
    def __init__(self, waiting_len) -> None:
        super(WaitingToStartThread, self).__init__()
        self.duration = waiting_len

    def run(self):
        for i in range(self.duration):
            time.sleep(1)
            val = ((i + 1) / self.duration) * 100
            self.update.emit(val)
        self.end.emit()
```

فایل threads.py خط ۴۴

بعد از هربار فرستاده شدن سیگنال **update** تابع **setWaitProgressValue** که مربوط به به روز رسانی شی مربوطه است صدا زده می شود و بعد از فرستاده شدن سیگنال **end** تابع **capturing** اجرا می شود. این تابع عملاً عکس برداری را آغاز می کند.

اول LCDNumber که تعداد عکس های باقی مانده را نشان می دهد مقدار دهی می کند و سپس شروع زمان انتظار بین هر عکس و عکس بعدی را با مقیاس دهم ثانیه به تابع waitAndCapture می دهد.

```
# start the capturing cycle
def capturing(self):
    camera_connectivity_state = self.check_camera()
    if (camera_connectivity_state == 'yes'):
        n = self.TotallCaptureNumber.value()
        k = n
        self.LCDNumber.display(k)
        self.LCDNumber.repaint()
        waiting = self.CapturingCycleSpinBox.value()
        self.waitAndCapture(waiting*10)
```

فایل sub\_window\_classes.py خط ۲۷۳

در تابع بعدی فقط یک پردازش ساخته می شود. این پردازش از نوع WaitingToCaptureThread است و کارش انجام همزمان صبر بین عکس ها و همچنین پر کردن progress bar مربوطه می باشد.

```
# waiting for capturing each photo
def waitAndCapture(self, t):
    self.waitToCaptureThread = WaitingToCaptureThread(t)
    self.waitToCaptureThread.update.connect(self.setWaitProgressValue2)
    self.waitToCaptureThread.end.connect(self.capture)
    self.waitToCaptureThread.start()
```

فایل sub\_window\_classes.py خط ۲۸۵

کد های مربوط به این پردازش هم زیر همینجا قابل مشاهده است.

```
# QThread class for handling Waiting time between each capture
class WaitingToCaptureThread(QThread):
    end = pyqtSignal(int)
    update = pyqtSignal(int)
    def __init__(self, waiting_len) -> None:
        super(WaitingToCaptureThread, self).__init__()
        self.duration = waiting_len
    def run(self):
        for i in range(self.duration):
            time.sleep(0.1)
            val = ((i + 1) / self.duration) * 100
            self.update.emit(val)
        self.end.emit(self.duration)
```

فایل threads.py خط ۴۴

اگر دقت کنید سیگنال تمام شدن این پردازش به تابعی به نام capture متصل شده است. در نتیجه با تمام شدن زمان صبر کردن این تابع اجرا می شود. در این تابع ابتدا عملیات های مربوط به عکس برداری صورت می گیرد. (نام گذاری عکس و فرستادن دستور عکس گرفتن به دوربین) سپس عدد LCDNumber به روز رسانی می شود و در انتها اگر تعداد عکس ها تمام نشده بود دوباره تابع waitAndCapture صدا زده می شود تا همین روند برای گرفتن عکس بعدی تکرار شود.

```
# take picture command and update lcd number
def capture(self, time):
    # taking picture
    self.camera.reset_image_index(self.multi_capture_number)
    t = datetime.datetime.now()
    current_time = str(t.strftime("%d")) + "." + str(t.strftime("%m")) + "." +
str(t.strftime("%Y")) + "." + str(
    t.strftime("%H")) + "." + str(t.strftime("%M"))
    self.camera.set_save_folder(current_time + "\\")
    t = self.camera.capture_single_image(autofocus=True)
    self.multi_capture_number += 1

    #updating lcd number
    k = self.LCDNumber.value()
    self.LCDNumber.display(k-1)
    self.LCDNumber.repaint()
    if self.LCDNumber.value() > 0:
        self.waitAndCapture(time)
    else:
        self.clear()
```

فایل sub\_window\_classes.py خط ۲۹۲

#### ۶. دکمه های save و reset

همانطور که از نام این دکمه ها مشخص است با فشار دادن دکمه ی save عکس ادیت شده ذخیره می شود. با فشار دادن دکمه ی reset هم عکس اولیه آپلود شده روی پنجره قبل از همه ی تدوین ها دوباره لود می شود. این دو کار با استفاده از توابع زیر اجرا می شوند.

```
# re-loading backup photo
def reset(self):
    self.gamaSlider.setValue(10)
    self.image = Image(self.backup)
    self.update_image()

# saving current photo into the file
def save(self):
    try:
        os.mkdir(os.path.join(os.getcwd(), "Edit-Captures"))
    except OSError as error:
        pass
    name = "Edit-Captures\\" + self.pictureName + "(Edit).JPG"
    cv2.imwrite(name, self.image.get_image())
```

فایل sub\_window\_classes.py خط ۲۹۲

#### ۷. بزرگ نمایی و کوچک نمایی تصویر (زوم و آنزوم)

برای نشان دادن بزرگی عکس دو متغیر داریم. اولی ابعاد ابتدایی و دیفالت عکس را مشخص می کند. دومی هم نسبت بزرگ و کوچک شدن تصویر را به ازای هر بار زوم و آنزوم مشخص می کند.

```
# zoom parameters for scaling image
self.zoom_scale = 1
self.zoom_parameter = 0.2
```

فایل main.py خط ۵۰

حال برای زوم و آنزوم دو روش داریم. یا از دکمه ها استفاده کنیم یا از اسکرول موس. دو تابع `zooming_in` و `zooming_out` داریم که بعد از فشار دادن دکمه یا تکان دادن اسکرول صدا زده می شوند. این دو تابع به صورت زیر تعریف می شوند.

```
# zooming functions
def zooming_in(self):
    self.current_image = QPixmap(
        QImage2ndarray.array2qimage(cv2.cvtColor(self.image.get_image(), cv2.COLOR_BGR2RGB)))
    self.scene = QGraphicsScene()
    h = 864 * self.zoom_scale * (1+self.zoom_parameter)
    self.zoom_scale = self.zoom_scale * (1+self.zoom_parameter)
    self.current_image = self.current_image.scaledToHeight(int(h))
    self.scene_img = self.scene.addPixmap(self.current_image)
    self.imageArea.setScene(self.scene)
    self.update_histograms()
def zooming_out(self):
    self.current_image = QPixmap(
        QImage2ndarray.array2qimage(cv2.cvtColor(self.image.get_image(), cv2.COLOR_BGR2RGB)))
    self.scene = QGraphicsScene()
    h = 864 * self.zoom_scale * (1-self.zoom_parameter)
    self.zoom_scale = self.zoom_scale * (1 - self.zoom_parameter)
    self.current_image = self.current_image.scaledToHeight(int(h))
    self.scene_img = self.scene.addPixmap(self.current_image)
    self.imageArea.setScene(self.scene)
    self.update_histograms()
```

فایل main.py خط ۱۳۵

کاری که این دو تابع انجام می دهند لود کردن عکس با ابعاد متفاوت بر روی همان قاب قبلی با سایز سابق است. به همین دلیل قسمت کمتر یا بیشتری از عکس قابل مشاهده می شود که احساس زوم یا آنزوم شدن به کاربر دست می دهد.

تابع دیگری هم با نام `zooming_rst` وجود دارد که با همین سیستم کار می کند. برای هندل کردن اسکرول موس هم از تابع زیر استفاده می کنیم.

```
# handling wheel zoom
def wheelEvent(self, event):
    a = event.angleDelta().y()
    self.zooming_in() if a>0 else self.zooming_out()
```

فایل main.py خط ۱۳۰

۸. نشان دادن هیستوگرام ها در پنجره اصلی

برای این کار بعد از هر بار اعمال تغییر روی عکس تابع `update_histograms` صدا زده می شود. در این تابع به تفکیک هر کدام از هیستوگرام های چنل های مختلف رنگی به روز رسانی می شوند. به عنوان نمونه برای چنل قرمز تابع زیر انجام می شود. نمودار های دیگر کانال های رنگی نیز دقیقاً همینطور کشیده می شوند.

```
# Updating Histograms Values
def red_channel_hist(self):
    self.red_channel_plot.clear()
    red_channel = cv2.calcHist([self.image.get_image()], [2], None, [256], [0, 256])
    rc = []
    for i in red_channel:
        rc.append(i[0]//1000)
    self.red_channel_plot.setBackground((141, 9, 37))
    self.red_channel_plot.plot(rc, pen='w')
    self.red_channel_plot.setLabel('bottom', 'Red Channel')
```

فایل `main.py` خط ۹۷

تقسیم بر هزار برای جمع شدن بازه ی پیکسل هاست و هیچ نکته ی خاصی ندارد.

۹. تغییر گاما

هر زمان که `slider` مربوط به ضریب گاما تغییر کند تابع زیر صدا زده می شود. این تابع مقدار تعیین شده را می گیرد و آن را به آجکت کلاس عکس می فرستد که در آنجا تغییرات مربوطه اعمال شود.

```
# Handling changes in gama value
def gama_slider_value_change(self):
    t = self.gamaSlider.value() / 10
    self.image.gama_adjust(t)
    self.update_image()
```

فایل `main.py` خط ۹۱

در کلاس عکس هم این تغییر به صورت زیر انجام می شود. این کار با استفاده از یک `Look Up Table` انجام می پذیرد.

```
# Gama Adjust Function
def gama_adjust(self, gamma):
    if gamma==0:
        gamma+=0.1
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255
                      for i in np.arange(0, 256)]).astype("uint8")
    # apply gamma correction using the lookup table
    self.img = cv2.LUT(self.backup, table)
```

فایل `Image.py` خط ۱۲۰

## ۱۰. دکمه ی Edit Channels

با کلیک کردن روی این دکمه پنجره ی جدیدی باز می شود که در آن میتوان محدوده ی کانال های رنگی را تغییر داد. این پنجره هم به صورت یک متغیر از کلاس اصلی ذخیره شده و در هنگام فشار دادن دکمه پنجره نمایش داده می شود. در هنگام نمایش این پنجره عکسی که هم اکنون روی پنجره ی اصلی هست روی این متغیر (پنجره جدید) ست می شود.

```
# opening channel edit window
def channel_edit(self):
    self.channel_edit_window.set_image(self.image)
    self.channel_edit_window.show()
```

فایل main.py خط ۸۶

در صفحه ی جدید تعدادی اسلایدر برای تغییر دادن محدوده ی کانال های رنگی دادیم. هر کدام از آنها به تابع خاص خود متصل هستند. در این صفحه هم یک نمونه از کلاس Image داریم و همچنین توابع مربوط به به روز رسانی عکس و هیستوگرام ها داریم. هر کدام از اسلایدر ها به صورت زیر کار می کنند.

```
# Handle changes in sliders
def red_range_slider_value_change(self):
    start, end = self.red_range_slider.value()
    self.image.strech_red_channel(start, end)
    self.update_image()
    self.update_histograms()
```

فایل sub\_window\_classes.py خط ۴۱۱

ابتدا مقدار های جدید هر اسلایدر گرفته می شود. سپس در کلاس عکسی که داریم تابع strech\_red\_channel اعمال می شود. (با توجه به رنگ مربوطه) این تابع به صورت زیر کار می کند.

```
# Channel Stretching Functions
def strech_red_channel(self, start=0, end=255):
    b, g, r = cv2.split(self.backup)
    r[r < start] = start
    r[r > end] = end
    self.img = cv2.merge([b, g, r])
```

فایل Image.py خط ۷۴

به این صورت که تمام پیکسل هایی که مقدارشان خارج از محدوده ی جدید وجود دارد به نزدیک ترین مرز محدوده منتقل می شوند. برای هر سه کانال رنگی همین الگوریتم اعمال می شود.

تمامی این تغییرات روی عکسی که در حال تماشای آن هستیم اعمال می شود و روی عکس اصلی ذخیره نمی شود. (اگر ذخیره شود امکان بازگشت وجود ندارد) برای همین دکمه ی apply کنار هر اسلایدر وجود دارد تا در صورت رسیدن به تغییری ایده آل آن را روی عکس اصلی ذخیره کنیم. در اینجا نکته ی قابل توجه این است که یک متغیر گلوبال داخل کلاس Image وجود دارد که دسترسی به آن از تمام نمونه های این کلاس در همه ی پنجره ها ممکن است. با استفاده از این متغیر (که عکس اصلی را روی آن ذخیره می کنیم) می توانیم تغییر های اعمال شده روی پنجره ی اصلی هم اعمال کنیم. نام این متغیر گلوبال img می باشد.

با زدن دکمه ی apply تابع زیر در فایل کلاس image اجرا می شود.

```
# Applying channel changes
def apply_red_channel(self):
    b1, g1, r1 = cv2.split(self.backup)
    b2, g2, r2 = cv2.split(self.img)
    self.backup = cv2.merge([b1, g1, r2])
    img = self.img
```

فایل Image.py خط ۱۰۰

اسلایدر دیگری هم در پنجره ی مورد نظر قرار دارد که روی هر سه کانال قرمز آبی و سبز تاثیر یکسان دارد. تمام توابع آن شبیه توابع گذشته می باشد تنها تفاوتش این است که روی هر سه کانال به صورت همزمان اجرا می شود.

```
def rgb_range_slider_value_change(self):
    start, end = self.rgb_range_slider.value()
    self.green_range_slider.setValue([start, end])
    self.red_range_slider.setValue([start, end])
    self.blue_range_slider.setValue([start, end])
    self.image.strech_rgb_channel(start, end)
    self.update_image()
    self.update_histograms()
```

فایل sub\_window\_classes.py خط ۴۲۷