MARMARA UNIVERSITY
ENGINEERING FACULTY

EE 4065

Introduction to Embedded Digital Image Processing

Homework Report

NAME:	*Baran*	*Muhammet Yücel*
SURNAME:	*ORMAN*	*ÇELİK*
NUMBER:	*150721063*	*150721024*

1. Question

For the first part of the homework, we had to take a grayscale image, store it as a header file, and then find it in the microcontroller's memory.

How We Did It:

Our image was 320x240 pixels. In grayscale (1 byte per pixel), this is **76,800 bytes**.

This was our first big problem. Our STM32F446RE chip only has 128KB of RAM. If we put this giant array in RAM, we would have almost no space left for anything else, and the program would probably crash.

So, we decided to store the image in **Flash memory** (the program memory) instead of RAM.

To do this, we used two special C keywords: `const` and `extern`.

`const`: We told the compiler that our image array is `const` (constant). This tells C, "This data will never change." Because it won't change, the compiler stores it in the Flash memory (which is Read-Only) instead of copying it to RAM when the program starts.

`extern`: To keep our project clean, we made two files:

`image.c`: This file holds the actual, giant array of all 76,800 pixel numbers (like 0x6e, 0x3a, 0x5f...).

`image.h`: This is the header file. We just put one line here: extern const uint8_t maymun[76800];. This line just tells the compiler that the array named maymun exists somewhere else in the project. We included this header in main.c.

Finding the Image in Memory:

- We ran the program in Debug mode and paused it.
- We opened the **"Memory"** window.
- We typed the address of our array (`&maymun`) into the memory window's address bar.
- It showed us the data starting at an address like `0x08002498`. Addresses that start with `0x08...` are in **Flash memory**. This proved that our `const` trick worked and we saved our RAM.
- We could also see the first pixel numbers right in the memory window, matching the data in our `image.c` file. This showed the image was loaded correctly.

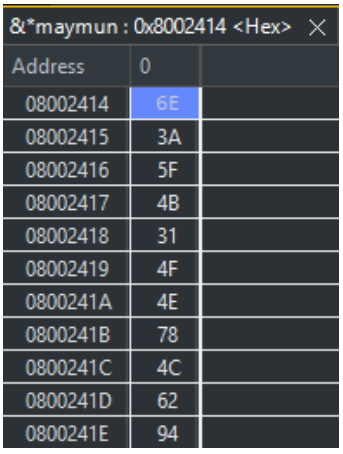
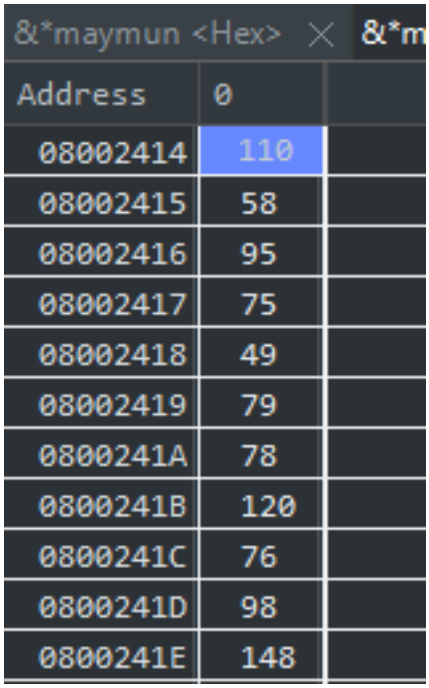


Figure 1 Pixel values of original image in memory as hexadecimals



&*maymun <Hex>	&*m
Address	0
08002414	110
08002415	58
08002416	95
08002417	75
08002418	49
08002419	79
0800241A	78
0800241B	120
0800241C	76
0800241D	98
0800241E	148

Figure 2 Pixel values of original image in memory as unsigned integers

```
 7  #include "image.h"
 8
 9  const uint8_t maymun[76800] = {
10    0x6e, 0x3a, 0x5f, 0x4b, 0x31, 0x4f, 0x4e, 0x78, 0x4c, 0x62, 0x94, 0x65,
11    0x5e, 0x57, 0x44, 0x53, 0x34, 0x5a, 0x46, 0x3c, 0x29, 0x23, 0x58, 0x61,
12    0x50, 0x79, 0x48, 0x55, 0x35, 0x51, 0x63, 0x4b, 0x59, 0x27, 0x3f, 0x5f,
13    0x2d, 0x5f, 0x76, 0x66, 0x35, 0x67, 0x76, 0x50, 0x6a, 0x5c, 0x46, 0x5c,
14    0x28, 0x33, 0x61, 0x5a, 0x49, 0x5c, 0xaa, 0x8d, 0x84, 0x7e, 0x54, 0x69,
15    0x2a, 0x45, 0x33, 0x54, 0x5e, 0x72, 0x77, 0x9d, 0x78, 0x78, 0x76, 0x69,
16    0x22, 0x56, 0x5c, 0x33, 0x31, 0x92, 0x8f, 0x69, 0x38, 0x5d, 0x66, 0x78,
17    0x4c, 0x71, 0x66, 0x3b, 0x2b, 0x26, 0x50, 0x4e, 0x3e, 0x69, 0x57, 0x5b,
```

Figure 3 Pixel array of the image

Before doing any image processing, we needed a simple way to manage our image data. Our image is not just a bunch of numbers; it has a size and a type.

We used a typedef struct to create a new custom data type called ImageTypeDef. We can think of this like a passport for the image.

```
/**
 * @brief Görüntü verisi ve özelliklerini tutan ana yapı
 */
typedef struct _image{
    uint8_t *pData;      // Görüntü piksellerinin tutulduğu dizinin pointer'ı
    uint16_t width;      // Görüntü genişliği
    uint16_t height;     // Görüntü yüksekliği
    uint32_t size;       // Görüntü boyutu (byte cinsinden)
    uint8_t  format;     // Görüntü formatı (IMAGE_FORMAT_... sabitleri)
} ImageTypeDef;
```

Figure 4 Data initializations of image on a "struct" architecture

2. Question

a) Negative Image

This transformation reverses the brightness of every pixel. Black becomes white, and white becomes black.

Mathematical Formula:

$$s = 255 - r$$

(where r is the old pixel value, and s is the new pixel value)

Expected New Pixel Values (First 3):

- $r = 110 \Rightarrow s = 255 - 110 = 145$
- $r = 58 \Rightarrow s = 255 - 58 = 197$
- $r = 58 \Rightarrow s = 255 - 95 = 160$

Result Verification: We checked the memory window for the destination buffer (img_hedef.pData). The first three pixels were correctly changed to **145, 197, and 160**, proving the calculation worked.

```
void IMAGE_apply_negative(ImageTypeDef *pSrcImg, ImageTypeDef *pDstImg)
{
    // 1. Güvenlik Kontrolleri
    // Görüntü pointer'ları geçerli mi?
    if (pSrcImg == NULL || pDstImg == NULL) return;
    // Görüntü verilerinin pointer'ları geçerli mi?
    if (pSrcImg->pData == NULL || pDstImg->pData == NULL) return;
    // Görüntü boyutları aynı mı?
    if (pSrcImg->size != pDstImg->size) return;
    // Görüntü formatları GRAYSCALE mi? (Bu formül sadece 1-byte'lık pikseller içindir)
    if (pSrcImg->format != IMAGE_FORMAT_GRAYSCALE || pDstImg->format != IMAGE_FORMAT_GRAYSCALE)
    {
        return;
    }

    // 2. Asıl İşlem
    // Kaynak görüntüdeki her pikseli tek tek döngüye al
    for (uint32_t i = 0; i < pSrcImg->size; i++)
    {
        // Kaynak pikseli oku
        uint8_t old_pixel = pSrcImg->pData[i];

        // Negatifini hesapla
        uint8_t new_pixel = 255 - old_pixel;

        // Yeni pikseli hedef görüntüye yaz
        pDstImg->pData[i] = new_pixel;
    }
}

/**
 * @brief  Bir görüntüye eşikleme uygular.
 * @note   Sadece GRAYSCALE formatı destekler.
 * @param  pSrcImg: Kaynak görüntü
 * @param  pDstImg: Hedef görüntü
 * @param  threshold: Eşik değeri (0-255).
 */
```

Figure 5 Code block for the Negative Image Function

&*img_hedef.pData <Unsigned Integer>

Address	0
20000238	91
20000239	C5
2000023A	A0
2000023B	B4
2000023C	CE
2000023D	B0
2000023E	B1
2000023F	87
20000240	B3
20000241	9D
20000242	6B

Figure 6 Negative image pixel values in memory as hexadecimals

&*img_hedef.pData : 0x20000238 <Unsigned Integer>

Address	0
20000238	145
20000239	197
2000023A	160
2000023B	180
2000023C	206
2000023D	176
2000023E	177
2000023F	135
20000240	179
20000241	157
20000242	107

Figure 7 Negative image pixel values in memory addresses as unsigned integers

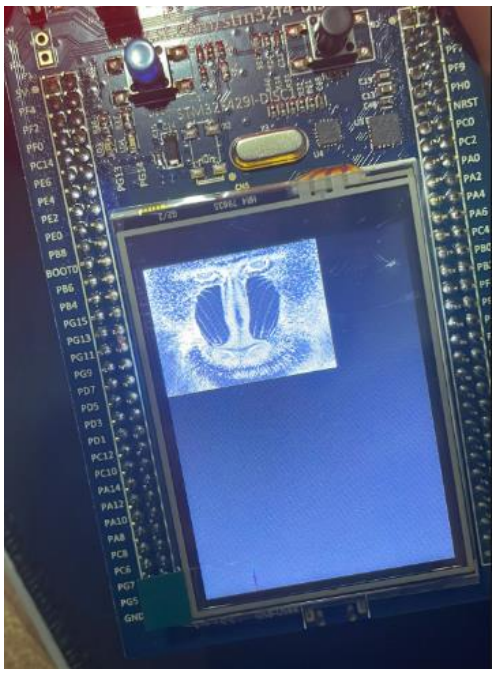


Figure 8 Negatie Image display on LCD TFT screen

b) Thresholding the Image

Thresholding is the simplest way to turn a grayscale image into a pure black-and-white image. We used 128 as the threshold value.

Mathematical Formula:

$$s = \begin{cases} 255 \; if \; r > 128 \\ 0 \; if \; r \leq 128 \end{cases}$$

Expected New Pixel Values (First 3):

- $r = 110 \leq 128; s \Rightarrow 0$
- $r = 58 \leq 128; \; s \Rightarrow 0$
- $r = 95 \leq 128; \; s \Rightarrow 0$

Result Verification: After running the function, we checked the destination memory. We saw that the array now only contained **00 and FF** (Hex) values, confirming that every pixel was correctly converted to pure black or pure white.

```c
void IMAGE_apply_threshold(ImageTypeDef *pSrcImg, ImageTypeDef *pDstImg, uint8_t threshold)
{
    // 1. Güvenlik Kontrolleri
    if (pSrcImg == NULL || pDstImg == NULL) return;
    if (pSrcImg->pData == NULL || pDstImg->pData == NULL) return;
    if (pSrcImg->size != pDstImg->size) return;
    if (pSrcImg->format != IMAGE_FORMAT_GRAYSCALE || pDstImg->format != IMAGE_FORMAT_GRAYSCALE)
    {
        return;
    }

    // 2. Asıl İşlem
    uint8_t new_pixel;
    for (uint32_t i = 0; i < pSrcImg->size; i++)
    {
        // Kaynak pikseli oku
        uint8_t old_pixel = pSrcImg->pData[i];

        // Eşikleme kuralını uygula
        if (old_pixel > threshold)
        {
            new_pixel = 255; // Beyaz
        }
        else
        {
            new_pixel = 0;   // Siyah
        }

        // Yeni pikseli hedef görüntüye yaz
        pDstImg->pData[i] = new_pixel;
    }
}

/**
 * @brief  Bir görüntüye gama düzeltmesi uygular.
 * @note   Sadece GRAYSCALE formatı destekler. FPU (float) işlemleri kullanır.
 * @param  pSrcImg: Kaynak görüntü
 * @param  pDstImg: Hedef görüntü
 * @param  gamma: Uygulanacak gama değeri
 */
```

Figure 9 Code block for the Applying Threshold Function5

&*img_hedef.pData : 0x20000238 <Unsigned Integer>

Address	0
20000238	0
20000239	0
2000023A	0
2000023B	0
2000023C	0
2000023D	0
2000023E	0
2000023F	0
20000240	0
20000241	0
20000242	255

Figure 106 Pixel values of thresholded image in memory as unsigned integer

&*img_hedef.pData <Unsigned Integer>

Address	0
20000238	00
20000239	00
2000023A	00
2000023B	00
2000023C	00
2000023D	00
2000023E	00
2000023F	00
20000240	00
20000241	00
20000242	FF

Figure 117 Pixel values of thresholded image in memory as hexadecimals

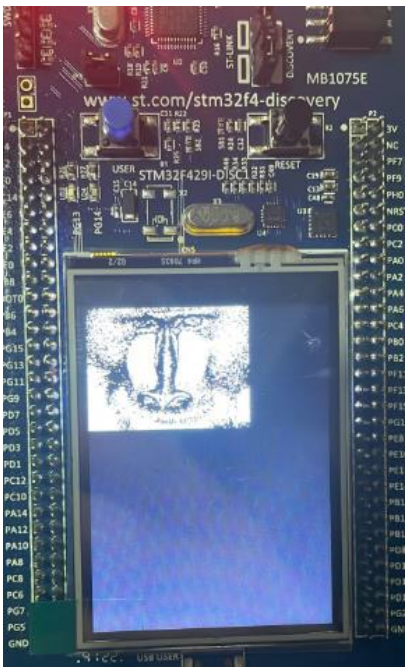


Figure 12 Threshold Applied Image display on LCD TFT screen

c) Gamma Correction for γ = 3 and γ = 1/3

Gamma correction is used to change the overall brightness and contrast non-linearly. We use floating-point math for this.

Mathematical Formula:

$$s = 255 \cdot \left(\frac{\Gamma}{255}\right)^{\gamma}$$

1. Gamma = 3.0 (Darkening): By using a Gamma value greater than 1, the function makes the overall image much darker.

Expected New Pixel Values (First 3):

- $s = 255 \cdot \left(\frac{110}{255}\right)^{\gamma=3} \approx 20$
- $s = 255 \cdot \left(\frac{58}{255}\right)^{\gamma=3} \approx 3$
- $s = 255 \cdot \left(\frac{95}{255}\right)^{\gamma=3} \approx 13$

Verification: The first three pixels became **20, 3, and 13**, confirming the darkening effect.

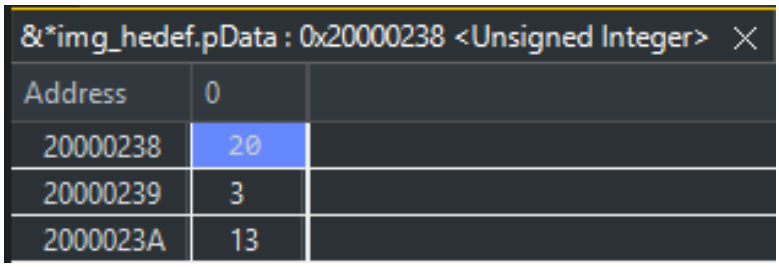


Figure 13 Question 2c, pixel values of gamma corrected image in memory as unsigned integers for Gamma = 3

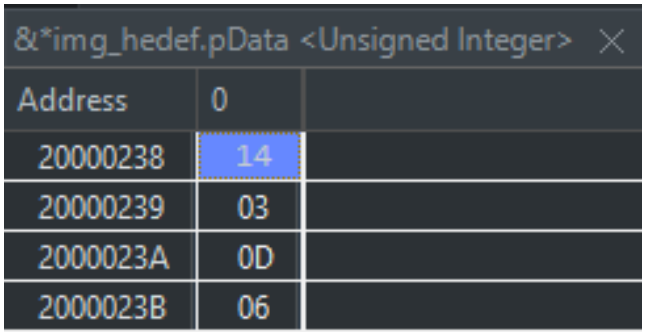


Figure 14 Question 2c, pixel values of gamma corrected image in memory as hexadecimals for Gamma = 3

2. Gamma = 1/3 (Brightening): By using a Gamma value less than 1 (0.333), the function makes the overall image much brighter.

Expected New Pixel Values (First 3):

- $s = 255 \cdot \left(\frac{110}{255}\right)^{\gamma=1/3} \approx 192$
- $s = 255 \cdot \left(\frac{58}{255}\right)^{\gamma=1/3} \approx 155$
- $s = 255 \cdot \left(\frac{95}{255}\right)^{\gamma=1/3} \approx 183$

Verification: The first three pixels became **192, 155, and 183**, confirming the brightening effect.

```
void IMAGE_apply_gamma(ImageTypeDef *pSrcImg, ImageTypeDef *pDstImg, float gamma)
{
    // 1. Güvenlik Kontrolleri
    if (pSrcImg == NULL || pDstImg == NULL) return;
    if (pSrcImg->pData == NULL || pDstImg->pData == NULL) return;
    if (pSrcImg->size != pDstImg->size) return;
    if (pSrcImg->format != IMAGE_FORMAT_GRAYSCALE || pDstImg->format != IMAGE_FORMAT_GRAYSCALE)
    {
        return;
    }

    // 2. Asıl İşlem

    // Gama formülündeki (1 / gamma) kısmını döngüden önce bir kez hesaplayalım.
    // Bu, 76,800 çarpma işleminden tasarruf sağlar.
    float exponent = gamma;

        for (uint32_t i = 0; i < pSrcImg->size; i++)
    {
        // Orijinal pikseli al
        uint8_t old_pixel = pSrcImg->pData[i];

        // Formülü uygula: encoded = ((original / 255) ^ (1 / gamma)) * 255

        // 1. (original / 255): Pikseli 0.0-1.0 arasına normalize et
        //    (float)old_pixel -> 'integer' olan 110'u, 'float' olan 110.0f'a çevirir
        //    Bunu yapmazsak C, 110/255 işlemini 0 (sıfır) olarak hesaplar!
        float norm_pixel = (float)old_pixel / 255.0f;

        // 2. (... ^ (1 / gamma)): 'powf' fonksiyonu ile üs al
        //    powf() -> 'float' versiyonudur, FPU için daha hızlıdır.
        float corrected_pixel = powf(norm_pixel, exponent);

        // 3. (... * 255): Tekrar 0-255 aralığına genişlet ve uint8_t'ye çevir
        uint8_t new_pixel = (uint8_t)(corrected_pixel * 255.0f);

        // Yeni pikseli hedef görüntüye yaz
        pDstImg->pData[i] = new_pixel;
    }
}
```

Figure 15 Code block for the Gamma Correction Function

&*img_hedef.pData <Unsigned Integer>

Address	0
20000238	C0
20000239	9B
2000023A	B7
2000023B	A9

Figure 168 Question 2c, pixel values of gamma corrected image in memory as hexadecimals for Gamma = 1/3

&*img_hedef.pData : 0x20000238 <Unsigned Integer>

Address	0
20000238	192
20000239	155
2000023A	183

Figure 17 Question 2c, pixel values of gamma corrected image in memory as unsigned integers for Gamma = 1/3

Figure 18 Gamma Corrected Image display on LCD TFT screen for Gamma = 3

9

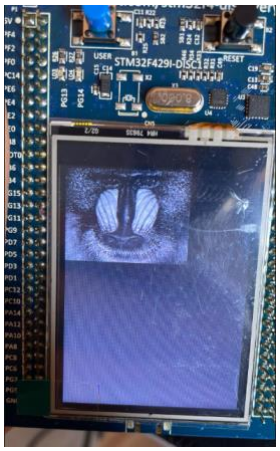


Figure 19 Gamma Corrected Image display on LCD TFT screen for Gamma = 1/310

d) Piecewise linear transformations for the part in b

This is a more flexible way to adjust contrast. It can **stretch** the contrast of a specific gray range. We chose $r_1 = 50$, $r_2 = 200$.

- *Mathematical Formula:*

$$s = \begin{cases} 0 \text{ if } r < 50 \\ 255 \cdot \frac{r-50}{200-5} \text{ if } 50 \leq r \leq 200 \\ 255 \text{ if } r > 200 \end{cases}$$

Expected New Pixel Values (First 3):

- $r = 110 \Rightarrow 155 \cdot \frac{110-50}{150} \approx 102$
- $r = 58 \Rightarrow 155 \cdot \frac{58-50}{150} \approx 13$
- $r = 95 \Rightarrow 155 \cdot \frac{95-50}{150} \approx 76$

Result Verification: The first three pixels became **102, 13, and 76**. This shows the function correctly shifted and stretched the values based on the formula, increasing the contrast within the 50 to 200 gray range.

```
void IMAGE_apply_piecewise_linear(ImageTypeDef *pSrcImg, ImageTypeDef *pDstImg, uint8_t r1, uint8_t r2)
{
    // 1. Güvenlik Kontrolleri
    if (pSrcImg == NULL || pDstImg == NULL) return;
    if (pSrcImg->pData == NULL || pDstImg->pData == NULL) return;
    if (pSrcImg->size != pDstImg->size) return;
    if (pSrcImg->format != IMAGE_FORMAT_GRAYSCALE || pDstImg->format != IMAGE_FORMAT_GRAYSCALE)
    {
        return;
    }
    // r1'in r2'den küçük olduğundan emin ol
    if (r1 >= r2) return;

    // 2. Asıl İşlem

    // (float)(r2 - r1) işlemini döngüden önce bir kez hesapla
    float denominator = (float)(r2 - r1);

    for (uint32_t i = 0; i < pSrcImg->size; i++)
    {
        uint8_t old_pixel = pSrcImg->pData[i];
        uint8_t new_pixel;

        if (old_pixel < r1)
        {
            new_pixel = 0;
        }
        else if (old_pixel > r2)
        {
            new_pixel = 255;
        }
        else
        {
            // Formül: 255 * ( (old_pixel - r1) / (r2 - r1) )
            // Kayan noktalı (float) hesaplama yapmak zorundayız
            float norm_pixel = (float)(old_pixel - r1) / denominator;
            new_pixel = (uint8_t)(norm_pixel * 255.0f);
        }

        // Yeni pikseli hedef görüntüye yaz
        pDstImg->pData[i] = new_pixel;
    }
}
```

Figure 20 Code block for the Piecewise Linear Transformation Function11

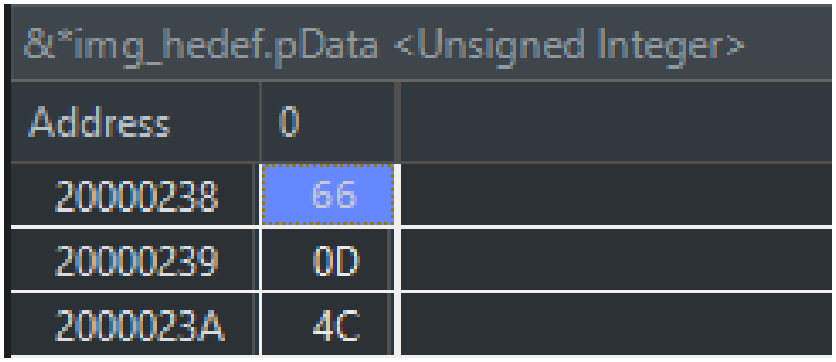


Figure 21 Pixel values for linear transformed image in memory as unsigned integer

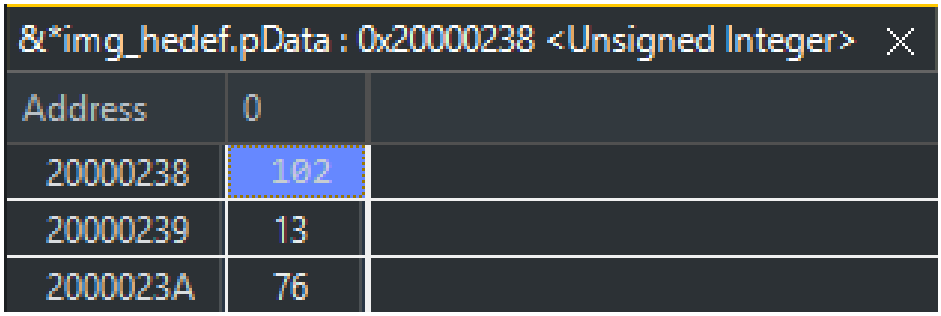


Figure 22 Pixel values for linear transformed image in memory as hexadecimals

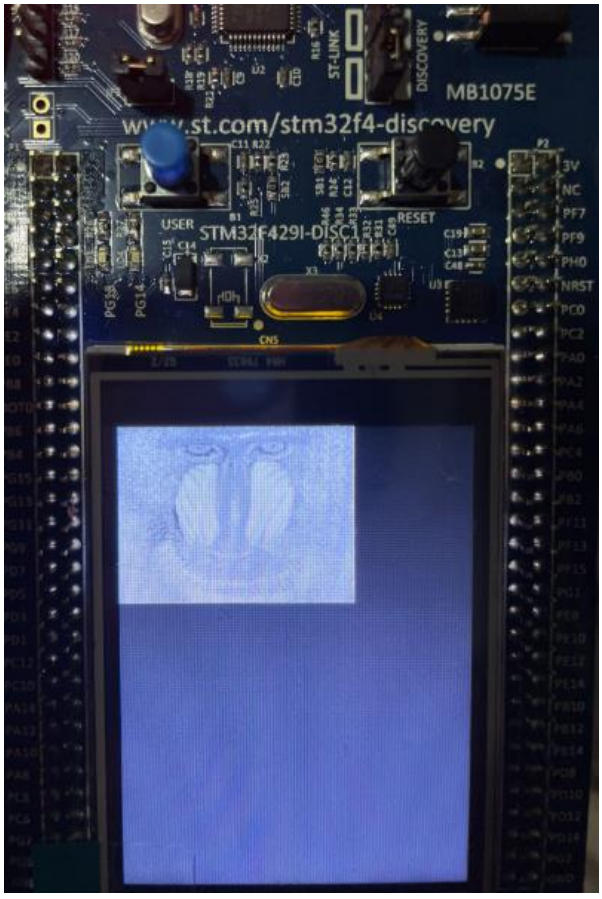


Figure 2312 Piecewise Linear Transformated Image display on LCD TFT screen for the range of 200 and 50

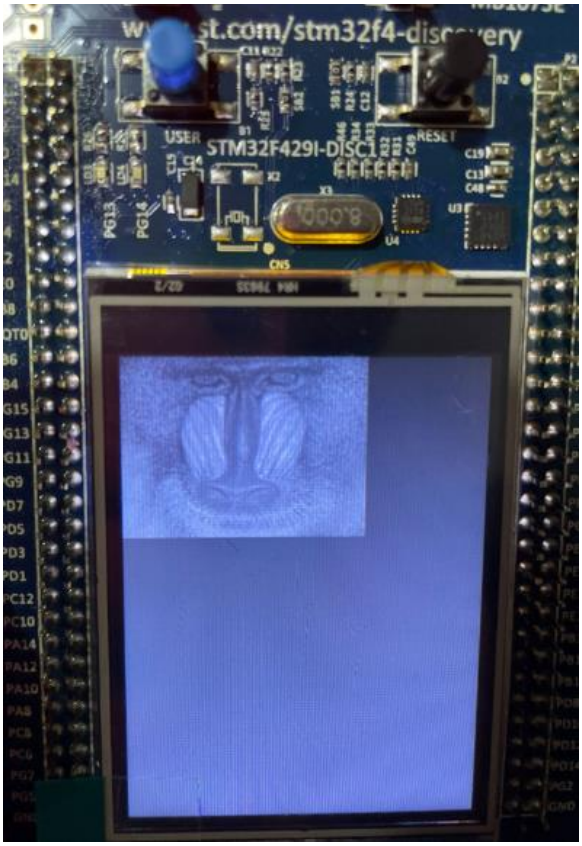


Figure 2413 Piecewise Linear Transformated Image display on LCD TFT screen for the range of 50 and 200 (Inverse Slope)