

Computing Report

All Subroutines

[`getPixel\(row, col, imageAddress\)`](#)

[`putPixel\(row, col, imageAddress\)`](#)

[`rowColToIndex\(row, col\)`](#)

[`getValueFromMask\(pixel, colorMask\)`](#)

General Functions

After considering a few of the challenges set out in the assignment I decided to write a suite of subroutines I thought would be universal. This set was mostly complete on the first revision.

In all the 'proper' functions only the return register is corrupted.

Pixel retrieval and storage

First it was necessary to write subroutines for accessing and setting pixels based on a simple two dimensional system. The main interfaces to this system were the *getPixel* and *putPixel* subroutines which take in a *row*, a *column* and a *image address* and returns the value stored in memory at the corresponding offset from the base *image address*. There are no safety checks in these accesses to a row or column that is too large. Such attempts will simply be executed.

Note: Row and col are unsigned integers. A pixel is an RGB color value of form `00RRGGBB` in hex.

The images are indexed from the top left corner using a zero based index as shown.

	col 0	col 1	col 2	col 3	col 4	col 5
row 0	pixel	pixel	pixel	pixel	pixel	pixel
row 1	pixel	pixel	pixel	pixel	pixel	pixel
row 2	pixel	pixel	pixel	pixel	pixel	pixel
row 3	pixel	pixel	pixel	pixel	pixel	pixel

`getPixel(row, col, imageAddress)`

Retrieves a pixel from the memory address at location (row, col) from the imageAddress.

Parameters:

REGISTER	CONTENT
R0	row
R1	col
R2	imageAddress

Return values:

REGISTER	CONTENT
R0	pixel

putPixel(row, col, imageAddress)

Sets a memory address at location (row, col) from the imageAddress to a given pixel.

Parameters:

REGISTER	CONTENT
R0	row
R1	col
R2	imageAddress

rowColToIndex(row, col)

Both *getPixel* and *putPixel* above are dependent on this method which is just a very short wrapper around the `MLA` instruction:

```
MLA R0, R2, R0, R1
```

The method gets the the picture width using *getPicWidth* and returns `row * width + col`. Wrapping this in a branch subroutine spoils this instructions efficiency but since it is used in both *getPixel* and *setPixel* and since it requires a branch to *getPicWidth* I thought it useful in a smaller subroutine.

Parameters:

REGISTER	CONTENT
R0	row
R1	col

Color Component Manipulation

Since many of the sections in the assignment would require logic that was dependent on the color components of the pixel I wrote two subroutines which made it easy to retrieve and set a specific *color component* of a given pixel. This made all parts of the assignment much quicker to implement as they are used in many places such as Contrasting, Blurring and Grey-scaling.

A color mask must be provided in the form $0xFF \ll 8n \mid 0 \leq n < 3$ where $n = 0$ is blue, $n = 1$ is green and $n = 2$ is red.

The subroutine works with *color components* which lie between the values of 0 and 255 inclusive.

getValueFromMask(pixel, colorMask)

Returns a *color component* determined by a given *color mask*.

Uses `AND` to clear everything not under the mask and shifts the component back to the right location. The '*right location*' is found by shifting the mask as well. Since the mask is a series of ones it will set the carry flag if the operation has gone one step too far, which I use as the end condition.

```
; R0 = RGB
; R1 = mask
AND R0, R0, R1                ; value = RGB & mask
getMaskWhile
    LSRS R1, R1, #4            ; while (mask >> 4
doesn't carry)
    BCS endGetMaskWhile       ; {
    LSR R0, R0, #4            ;   value >> 4
    B getMaskWhile            ; }
endGetMaskWhile
```

Parameters:

REGISTER	CONTENT
R0	pixel
R1	colorMask

Return values:

REGISTER	CONTENT
R0	colorComponent

setValueFromMask(pixel, colorMask, colorComponent)

Sets a *color component*, specified by a given *color mask*, of a pixel and returns that pixel.

The *color component* of the original pixel is cleared using the mask and the provided *color component* is shifted to the correct location using the similar logic to that of the *getValueFromMask* subroutine. The shifted *color component* and the cleared pixel are then simply added together.

Parameters:

REGISTER	CONTENT
R0	pixel
R1	colorMask
R2	colorComponent

Return values:

REGISTER	CONTENT
R0	pixel

Pixel Traversal and Modular Code

This function is particularly cool. Throughout the entire code there exists only one loop for going through all of the pixels in the image. This is because the function *applyToAll* takes in the address of a function which is executed at each pixel location (row, col) of the image. This is very exciting because it vastly shortened the complexity and the time required to write the other effects. The functions that can be used with this I have dubbed '*wrappers*'. They execute pixel sensitive operations. They all have the same interface.

applyToAll(wrapperAddress)

Parameters:

REGISTER	CONTENT
R0	wrapperAddress

The functions are called for all valid combinations of (row, col) using the following lines:

```
; R2 = wrapperAddress
MOV LR, PC
BX R2                                ; execute(wrapperAddress)
```

Wrappers

wrapper(row, col)

Parameters:

REGISTER	CONTENT
R0	row
R1	col

Note: Most of the wrappers use a hard-coded value for the '*copy address*' which is the start of an empty space to which the image can be copied to. My original idea was to keep all relevant pixels in registers or on the stack but this proved to be a little too complicated for my current implementation which tends to have a number of the registers locked up for reference values.

Quick outline of wrappers:

copy(*row*, *col*)

Copies a pixel from the given image to the same location in the duplicate image. Simple use of *getPixel* and *setPixel*.

applyAdjust(*row*, *col*)

Updates the current pixel in the original image to a pixel with contrast and brightness applied.

applyGreyScale(*row*, *col*)

Updates the current pixel in the original image to a pixel with greyscale applied.

applyMotionBlur(*row*, *col*)

Calculates the motion blur value for a pixel in the copied image and updates it in the original.

lensEffectCopy(*row*, *col*)

Calculates the *transformed(row, col)* value, takes the pixel from the copied image and changes the actual (*row*, *col*) pixel to that one.

Misk

divide(*numerator*, *denominator*)

Implements a fast shifting divide function.

Parameters:

REGISTER	CONTENT
R0	numerator
R1	denominator

Return values:

REGISTER	CONTENT
R0	remainder
R1	quotient

Part 1 – Brightness and Contrast

For each pixel the pixel is passed to the *adjustPixelColor* subroutine which breaks the pixel down into its constituent color parts and applies the *adjustColor* subroutine to them.

There were three main design choices I made at this point:

1. Use masks to extract color components instead of loading bytes.
2. Keep brightness and contrast values in memory.
3. Use negative contrast values.

For the first I decided to use masks since I found the colors easier to think about in terms of full words than in bytes. This also made it easier to write since I didn't have to consider the exact position of a '*pixel pointer*' relative to a single pixel, though that work could easily have been passed out to a subroutine. I decided to use masks for this reason. The methods for which are described under the *General Functions* section. The masks are used in the *adjustPixelColor* subroutine to get and set the color values of the pixel.

By rights the brightness and contrast should have been passed as parameters to all the functions however I had originally intended to implement a simple user interface in which I would save the brightness and contrast to specific locations in memory. As such I had declared these values in a memory area. Currently the program doesn't support text based interaction but it wouldn't be difficult to implement.

Negative contrast values were by and large considered to be invalid input for the functions provided:

$$R'_{ij} = (R_{ij} * \alpha) / 16 + \beta$$

Where α is the contrast coefficient and β is the brightness.

However I have decided to allow negative values which will invert the color of the image. For $-16 < \alpha < 0$ the image will be inverted colors and the contrast increased. For $\alpha < -16$ the image will be inverted and the contrast decreased. For $\alpha = -16$ the contrast will remain unchanged but the image will be inverted by the standard method.

Handling the negative values was pretty simple. During the computation of the above function I simply add two extra lines:

```
; R2 = color
; R4 = contrast
; R5 = brightness
MULS R2, R4, R2           ; color *= contrast
ASR R2, R2, #4            ; color /= 16
ADDMI R2, R2, #255        ; invert color if contrast was negative
ADDS R2, R2, R5           ; color += brightness
```

I use `ASR` to keep the sign of the number while dividing by 16 and if the result is negative it is subtracted from 255 thus inverting it.

The final number is checked for validity. Number greater than 255 get set to 255. Since the color inversion can make the color negative we must consider the color value as signed so values less than 0 are set to 0.

adjustPixelColor(pixel)

Breaks pixels down into constituent parts and calls *adjustColor* on them.

Parameters:

REGISTER	CONTENT
R0	pixel

adjustColor(*colorComponent*, *contrast*, *brightness*)

Applies contrast and brightness to the supplied color value. Also applies color inversion.

Parameters:

REGISTER	CONTENT
R0	colorComponent
R1	contrast
R2	brightness

Return values:

REGISTER	CONTENT
R2	colorComponent

Testing

I've tested particular values for contrast and brightness.

Contrast test values:

Contrast Value	Reason
16	Check no change
0	Test 0 value - should be completely black
10	Low contrast - should be darker
22	High contrast - should be more contrasted / starker
-16	Check negatives - should invert color with no effect on contrast

Original / 16	0	10	22	-16
				
Worked	Worked	Worked	Worked	Worked

Brightness test values:

Brightness Value	Reason
0	Check no change
60	Should be brighter
-60	Should be darker
255	Checking overflow - should be completely white
-255	Checking overflow - should be completely black

Original / 0	60	-60	255	-255
				
Worked	Worked	Worked	Worked	Worked

Part 2 - Blur Effect

My method for blur effect can take any size radius. I had considered implementing a kernel based method for this but since kernels run in $O(n^2)$ time and my method would run in $O(n)$ time I decided to go with my current approach. Originally I had wanted to avoid making a duplicate copy of the image by interchanging pixels in the stack:

For n radius push n pixels to the stack. Compute the average. Update the pixel. Pop $n - 1$ pixels from the stack and into a new stack. Pop the last pixel. Pop the $n - 1$ pixels back onto the original stack. Load a new pixel and push it onto the stack. In this method the diagonal would be traversed and only original pixel values would be used.

However, the logic for how many pixels to load in the edge cases became a little too complex for me and since the effect I was going to use in the third part of the assignment was going to need a full copy of the image I decided to make a full copy and work with that.

Implementation

The *motionBlur* method is implemented as a wrapper, explained in the *General Functions* section. First all the pixels need to be copied to a new location. This location is hard-coded which I admit is not an ideal implementation. Once the copy is finished the pixels in the copied image are traversed. The radius provided must be odd. The center pixel being one and two equal number of pixels either side. The center pixel is pushed to the stack and the pixels within the radius are pushed to the stack as well. This is followed by a number detailing the number of pixels pushed. This is because I ignore hypothetical pixels beyond the edge of the image. This means the pixel whose distance to the nearest edge is less than `floor(radius / 2)` will actually have less blur applied. The following example would explain this:

Using a radius of 8 the cells along the diagonal contain the number of pixels considered:

	0	1	2	3	4	5	6	7	8
0	5								
1		6							
2			7						
3				8					
4					8				
5						8			
6							7		
7								6	
8									5

As you can see the value for the pixel (0, 0) only takes into account the pixels (1, 1) through (4, 4) as the pixels (-1, -1) through (-4, -4) are out of bounds.

This stack is passed to the *averageN* subroutine which returns the mean pixel that is to take the place of the pixel in the original image. After this the stack is cleared and a new stack is created. (Please note that this stack system is a remnant of the previously desired functionality, where there is no duplicate of the image.)

applyMotionBlur(row, col)

Calls creates the parameters for the *averageN* subroutine at the given pixel location and writes the result to the original image.

Parameters:

REGISTER	CONTENT
R0	row
R1	col

averageN(pixels...)

This subroutine allows an arbitrary number of pixels to be passed through the stack. The size of the blur is therefore only limited by the size of the stack and, of course, the size of the image. This routine splits the pixels into their constituent colors and finds their mean.

Parameters:

REGISTER	CONTENT
Stack	N - number of pixels in stack
Stack	N pixels

Return values:

REGISTER	CONTENT
R0	averagePixel

Testing

I've tested particular values for the radius.

Contrast test values:

Radius Value - Standard radius value.	Reason
0 - 1	Check no change
2 - 5	Test given example
6 - 13	Check large values
14 - 29	Check large values - should be faster than kernel methods

Original / 0 - 1	2 - 5	16 - 13	14 - 29
			
Worked	Worked	Worked	Worked - Was Faster

Part 3 - Bonus Effect - Lens Blur

For my third part of the assignment I decided to attempt a lens blur effect. This magnifies the image where the coefficient of magnification on a pixel is dependent on its distance from the center. This effect is also called barrel blur. There are many algorithms online for achieving this effect, most of which are in the form of '*lens correction*' which corrects this distortion in real cameras. What I need for this effect is its inverse.

[Excellent resource on this.](#)

From the link above I tried to use this equation:



The first problem was that this equation requires an alteration of the coordinate system from an integer based index to a centered coordinate system that stretches from -1 to 1. Using decimal locations wasn't very useful for me so I attempted to alter the formula to use my default coordinate system. This worked to a certain extent, however, the divisor was regularly less than 1 rendering the equation useless no matter how much I tweaked it.

So abandoning the mathematically beautiful formula above I implemented my own derivation which looks like this:

$$P_x = (\alpha * P_x') / ||P||$$

$$P_y = (\alpha * P_y') / ||P||$$

Where alpha is some coefficient. For $\alpha > 0$ the image is magnified from the center. For $\alpha < 0$ the image undergoes an effect called *Pincushion Distortion* where the image appears to shrink as it nears the center.

$||P||$ is the distance of the pixel from the center of the image. Simply calculated using $\text{sqrt}(x^2 + y^2)$

For the formulas above I had to implement a *square root* function to find the distance from the center. My square root implement Newton's method for integer square root approximation. It is an iterative formula:

$$x = (x' + S/x')/2$$

Where S is the original integer and x is an approximation for the square root. We can stop when $|x - x'| < 1$. If x is an overestimation of the square root S/x can be assumed to be less than the square root. As such the mean of x and S/x is a closer approximation to the square root. We keep applying this until the value of x changes by 1 or less.

```

MOV R11, R0                ; save number
LDR R3, =0                 ; temp = 0
MOV R2, R0                 ; x = S

find_sqr_whl               ; while (previous x != next x) {
    LSR R2, R2, #1         ; x /= 2
    SUBS R4, R2, R3
    BEQ end_sqr_whl        ;
                                return x
    CMP R4, #1
    BEQ end_sqr_whl
    MOV R3, R2              ; temp = x
    MOV R1, R2              ;
    MOV R0, R11             ;
    BL divide               ;
    ADD R2, R2, R1          ; x = x + divide(number, x)
    B find_sqr_whl
end_sqr_whl

MOV R0, R2

```

Before the processing began the row and col indexes had to be normalised to a coordinate system where the origin was in the center of the image as such:

	-4	-3	-2	-1	0	1	2	3	4
4	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel
3	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel
2	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel
1	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel
0	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel
-1	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel
-2	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel
-3	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel
-4	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel	pixel

This is simply achieved by:

```
x = col - width
```

```
y = row - y
```

Now we can calculate magnitudes using (0, 0) as the center of the image.

sqrt(*number*)

Approximates the square root of a number.

Parameters:

REGISTER	CONTENT
R0	number

Return values:

REGISTER	CONTENT
R0	squareRoot

distanceSQR(*y*, *x*)

Finds the square of the distance of an (x, y) value from the origin. Return `x^2 + y^2`

Parameters:

REGISTER	CONTENT
R0	y
R1	x

Return values:

REGISTER	CONTENT
R0	squareDistance

applyLens(*row*, *col*)

Wrapper for applying the lens effect to a pixel. Takes a pixel coordinate, normalises it as follows:

```
SUB R0, R0, R2      ; y -= centery
SUB R1, R1, R3      ; x -= centery
```




Then transforms it using the formulas above. It retrieves the pixel from the copied image at the transformed coordinate and update the original pixel to this value.

Testing

I've tested particular values for the radius.

Contrast test values:

Radius Value - Standard radius value.	Reason
0	No effect
20	Lens effect
-20	Pincushion effect

Original / 0 - 1	20	-20
		
Worked	Worked	Worked but with error

For the pincushion effect we get an error where the values of the transformed `y` and `x` are out of bounds. In the `x` axis it causes a type of wrapping, in the `y` this causes access to memory that is completely zeroed and hence fills in black.