OS References & Documentation

**Kieran Warren 100332570**

**Systems Programming**

# Demo

The demo consists of a number of different features to show off as many implemented features as possible.

Upon load up, you can select a resolution of your choice in console mode. For best viewing, 400x300 is recommended. However, any resolution combination will work up to 400x600, but may cut off some parts of the demo or stretch incorrectly due to Bochs resolution scaling. By default, chain4mode is disabled but can be enabled in the code. Of course, this limits resolution to 320x200.

Once selected the OS will load in the selected resolution with user mode enabled by default and will load up a standard 256 colour palette similar to those found here: <https://jonasjacek.github.io/colors/>. This can be easily customised to personal preference in the CreateColourPalette method.

The OS is then divided up into a few different sections. The first is the colour palette demo, displaying a single line for each colour on the palette.

The second section is the rectangle demo. It demos the draw and fill rectangle methods with a variety of sizes and colours.

The third is a demo for drawing a variety of filled and non-filled circles.

The fourth section demos polygon drawing. Using F1-F10 you can manipulate the polygons in different ways. Increase the number of edges, the size, position and rotation of the polygon. Press any of these keys to display the polygons. Polygons may flash due to refresh of the screen between draw operations.

The final section is a demo of text printing and keyboard input. You can type in text here yourself. But limited to A-Z, 0-9 due to technical issues with the extended character set in a compressed format.

The demo runs infinitely in the input loop which can update the polygons and input text.

# Testing

To test robustness of each algorithm used. I created a few different tests to cover many scenarios. For example, testing draw line required 12 different lines to comfortable test all possible lines. These included lines facing left, right, up and down as well as in each octet. Most drawing methods break down to simple line drawing, so these put the draw line through its paces to confidently cover all possibilities. And to test some of the more complex methods at a higher level, I used a small test project in HTML canvas and JavaScript to make debugging issues easier, then applying the working code to the project.

To test resolutions, I used a similar test suite to the demo suite to check everything display in a variety of resolutions with chain 4 mode disabled, and then at 320x200 with it enabled.

To test colour palette settings, I used a simple for loop for 0 through 255 drawing a line for each. The same as the one in the demo. I tried a few different generation setups, but this one felt most flexible with all colours reasonably well represented including grayscale.

Character printing from a pre-loaded bitmap proved somewhat of a challenge, whilst logic was relatively simple, iterating through the array, it was difficult pre-loading the image in an easily readable way. Using similar code, loading image bitmaps would be relatively simple.

To be able to implement more than 3 parameters in a user-friendly method, I chose to share registers between two 16 bit values, usually Vectors. These can then be reversed by using bit shifting and a bitwise AND.

# Draw

Container for all drawing function implementations.

## SetPixel(uint16\_t x, uint16\_t y, uint8\_t colour)

Sets specified x, y position on screen to the specified colour. Can react to chain4mode being enabled or disabled. If disabled, some additional work is required to retrieve the correct plane and position in memory.

## ClearScreen(uint8\_t colour)

Clears the entire screen with the specified colour. Implements a well optimized version which takes around half the time of the first implementation to clear a 400x600 resolution screen.

## DrawHorizontalLine(Vector2 start, uint16\_t length, uint8\_t colour) (Ring 0)

Draws a horizontal line from specified start position of chosen length and colour. Has two different methods based on chain4mode being enabled or not. With chain4mode enabled, no additional calculations are required for pixel positions. Therefore, it’s more efficient to just run through the specified memory position as fast as possible. With chain4mode, it will execute SetPixel per pixel to correctly specify position. Ensure start position is the left most position on the line.

## DrawUserHorizontalLine(uint32\_t start, uint16\_t length, uint8\_t colour)

The receiving end of the ring 3 to ring 0 transfer code. Vector2 made of 2 x 16-bit is passed as a single 32-bit value and needs to be reversed before executing DrawHorizontalLine. Not recommended for manual use.

## DrawVerticalLine(Vector2 start, uint16\_t length, uint8\_t colour) (Ring 0)

Functions the same as horizontal line when chain4mode is disabled. Iterates between start position and the line length, executing SetPixel for each pixel. Only use when confident line is straight and downward facing.

## DrawUserVerticalLine(uint32\_t, uint16\_t length, uint8\_t colour)

The receiving end of the ring 3 to ring 0 transfer code. Vector2 made of 2 x 16-bit is passed as a single 32-bit value and needs to be reversed before executing DrawVerticalLine. Not recommended for manual use.

## DrawLine(Vector2 start, Vector2 end, uint8\_t colour) (Ring 0)

More robust method for drawing a line between any 2 points on screen. Can use DrawHorizontalLine or DrawVerticalLine if provided line is found to be straight. If not it will fall back on Bresenham’s Line Drawing Algorithm. Iterating through the line, it will find the closest pixel to plot for each position across the line. This method could be further optimized for common diagonal lines such as 45 degrees where pattern is simple and the additional calculations used in Bresenham’s Line Algorithm can be avoided.

## DrawUserLine(uint32\_t start, uint32\_t end, uint8\_t colour)

The receiving end of the ring 3 to ring 0 transfer code. Receives the 2 merged vectors and splits them up. Then calling DrawLine with the vectors. Not recommended for manual use.

## DrawRectangle(Rectangle rect, uint8\_t colour) (Ring 0)

Draws an empty rectangle using the Rectangle struct passed in. Uses the optimized DrawHorizontalLine and DrawVerticalLine and also the least possible number of variable changes between lines.

## DrawUserRectangle(uint32\_t start, uint32\_t size, uint8\_t colour)

The receiving end of the ring 3 to ring 0 transfer code. Receives the merged vectors for start and size and calls DrawRectangle. Not recommended for manual use.

## FillRectangle(Rectangle rect, uint8\_t colour) (Ring 0)

Fills rectangle at specified position and of specified size. Uses the DawHorizontalLine for extra performance if chain4mode is enabled.

## FillUserRectangle(uint32\_t centre, uint32\_t size, uint8\_t colour)

Ring 3 to ring 0 transfer method. Breaks up 32-bit values into 16 bit vectors and then a rectangle. Then calls FillRectangle. Not recommended for manual use.

## DrawCircle(Vector2 centre, uint16\_t radius, uint8\_y colour) (Ring 0)

Draws a circle using midpoint circle algorithm. Divides the circle into octets to reduce iterations, can then plot pixel on the same point of each octet by adding/subtracting the position differently for each one.

## DrawUserCirlce(uint32\_t centre, uint16\_t radius, uint8\_t colour)

Converts 32-bit centre to Vector2 and then calls DrawCircle. Not recommended for manual use.

## FillCircle(Vector2 centre, uint16\_t radius, uint8\_t colour) (Ring 0)

Adapts the midpoint circle algorithm by drawing a line between the left and right most positions in each Y direction.

## FillUserCircle(uint32\_t centre, uint16\_t radius, uint8\_t colour)

Converts 32-bit centre to Vector2 and then calls FillCircle. Not recommended for manual use.

## DrawPolygon(uint16\_t xPoints[], uint16\_t yPoints, uin8\_t colour) (Ring 0)

Draws a polygon from the specified x and y points. Drawing line between each point and point +1. Then last and first to complete the polygon.

## DrawUserPolygon(uint16\_t xPoints[], uint16\_t yPoints[], uint32\_t sizeCol)

Converts the merged colour and number of sides into two separate 16-bit values then calls DrawPolygon.

## FillPolygon(uint16\_t xPoints[], uint16\_t yPoints, uin8\_t colour) (Ring 0)

Fills a polygon with the specified points. Creates a rectangle from the smallest and largest values of the polygon. Then tests if it is within the polygon to choose if it should be filled.

## FillUserPolygon(uint16\_t xPoints[], uint16\_t yPoints[], uint32\_t sizeCol)

Converts the merged colour and number of sides into two separate 16-bit values then calls FillPolygon.

## InPolygon(uint16\_t x, uint16\_t y, uint16\_t xPoints[], uint16\_t yPoints[], uint16\_t length)

Checks to see if a point is inside the bound of the given polygon. An adaptation of the algorithm created by Randolph Franklin in Fortran.

## Reverse32BitMergeVector2(uint32\_t a)

Reverses the 32 bit merge for user functions. Bit shift right by 16 to get first value. Then use bitwise AND with 0x0000FFFF to get the second. Returns a Vector2.

# Print

Contains any functions relating to printing text to the screen (when not in text mode)

## WriteCharacter(char c, uint16\_t x, uint16\_t y, uint8\_t colour) (Ring 0)

Writes the specified character from the font bitmap to the chosen position. Limited to A-Z 0-9 due to memory issues when loading extended bitmap.

## WriteUserCharacter(char c, uint32\_t position, uint8\_t colour)

Converts 32-bit position to Vector2 for ring 3 to ring 0 transfer. Then executes WriteCharacter. Not recommended for manual use.

## WriteText(const char\* text, uint16\_t x, uint16\_t y, uint8\_t colour) (Ring 0)

Writes a string to a specified position. Moving along in the x-axis for each character, calling WriteCharacter for each.

## WriteUserText(const char\* text, uint16\_t x, uint16\_t y, uint8\_t colour)

Converts 32-bit position to Vector2 for ring 3 to ring 0 transfer. Then executes WriteText. Not recommended for manual use.

## ToStringUint16(uint16\_t i, uint16\_t base) (Any Ring)

Converts a 16-bit unsigned integer to a string. Can be used in any mode.

# User

Ring 3 implementations for most major functions. Along with the already implemented console methods, this includes drawing methods and text writing methods. Many of these methods use the MergeTwo16Bit method to create a 32 bit register ready value from two 16 bit values. Extracted at the other end.

Draw methods are implemented through interrupt 81 while character writing is using interrupt 82.

Here is the list of methods designed to run from User Mode:

* User\_SetPixel(unsigned int x, unsigned int y, uint8\_t colour) User\_DrawLine(uint16\_t startX, uint16\_t startY, uint16\_t endX, uint16\_t endY, uint8\_t colour)
* User\_ClearScreen(uint8\_t colour)
* User\_DrawHorizontalLine(uint16\_t startX, uint16\_t starty, uint16\_t endX, uint16\_t endY, uint8\_t colour)
* User\_DrawVerticalLine(uint16\_t startX, uint16\_t startY, uint16\_t length, uint8\_t colour) User\_DrawRectangle(uint16\_t startX, uint16\_t startY, uint16\_t width, uint16\_t height, uint8\_t colour)
* User\_FillRectangle(uint16\_t startX, uint16\_t startY, uint16\_t width, uint16\_t height, uint8\_t colour)
* User\_DrawCircle(uint16\_t centreX, uint16\_t centreY, uint16\_t radius, uint8\_t colour)
* User\_FillCircle(uint16\_t centreX, uint16\_t centreY, uint16\_t radius, uint8\_t colour)
* User\_DrawPolygon(uint16\_t xPoints[], uint16\_t yPoints[], uint8\_t colour)
* User\_FillPolygon(uint16\_t xPoints[], uint16\_t yPoints[], uint8\_t colour)
* User\_WriteText(const char\* text, uint16\_t x, uint16\_t y, uint8\_t colour)
* User\_WriteCharacter(char c, uint16\_t x, uint16\_t y, uint8\_t colour)
* User\_WriteText(char\* str, uint16\_t x, uint16\_t y, uint8\_t colour) (Ring 3)

## MergeTwo16Bit(uint16\_t a, uint16\_t b)

Bitshifts the first value by 16 left. Then adding second value to it to create the merged values.

# Math

Helper file with common maths functions, could be extended further to have other common maths functions.

## cos(float a)

Finds the cosine value of the input value (in Radians)

## sin(float a)

Finds the sine value (in Radians)

## abs(float a)

Returns the absolute value