Technical Design Documentation

Contents

[API Documentation: 2](#_Toc501563215)

[Notes: 2](#_Toc501563216)

[Ring 3 -> Ring 0: 2](#_Toc501563217)

[vga.c 2](#_Toc501563218)

[void SetResolution(int width, int height); 2](#_Toc501563219)

[void SetColourPalette(uint8\_t red, uint8\_t green, uint8\_t blue, int index); 2](#_Toc501563220)

[void SetLineStyle(int dashLength); 2](#_Toc501563221)

[void ClearScreen(); 2](#_Toc501563222)

[void SetPixel(unsigned int x, unsigned int y, unsigned int colour); 3](#_Toc501563223)

[bool MoveTo(int x, int y); 3](#_Toc501563224)

[void DrawChar(char c, uint8\_t colour); 3](#_Toc501563225)

[void DrawString(const char\* str, uint8\_t colour); 3](#_Toc501563226)

[bool LineTo(int x, int y, uint8\_t colour); 3](#_Toc501563227)

[void DrawRectangle(int width, int height, uint8\_t colour); 3](#_Toc501563228)

[void FillRectangle(int width, int height, uint8\_t colour); 4](#_Toc501563229)

[void DrawCircle(int radius, uint8\_t colour); 4](#_Toc501563230)

[void FillCircle(int radius, uint8\_t colour); 4](#_Toc501563231)

[void DrawPolygon(int \*coordinates, int numberOfVertices, uint8\_t colour); 4](#_Toc501563232)

[void FillPolygon(int \*coordinates, int numberOfVertices, uint8\_t colour); 4](#_Toc501563233)

[void FloodFill(int startX, int startY, uint8\_t replacementColour); 5](#_Toc501563234)

[edge\_list.c 5](#_Toc501563235)

[void PrintListY(node \*head); 5](#_Toc501563236)

[void PrintListX(node \*head); 6](#_Toc501563237)

[int GetListSize(node \*head); 6](#_Toc501563238)

[node \* GetNodeFromList(node \*head, int index); 6](#_Toc501563239)

[void AddToList(node \*\*head, node \*nodeToAdd); 6](#_Toc501563240)

[bool RemoveFromList(node \*\*head, edge \*edgeToRemove); 6](#_Toc501563241)

[void SwapInEdgeList(node \*head, int first, int second); 6](#_Toc501563242)

[void SortEdgeListByY(node \*edgeListHead); 6](#_Toc501563243)

[void SortEdgeListByX(node \*edgeListHead); 6](#_Toc501563244)

[Testing: 6](#_Toc501563245)

# API Documentation:

## Notes:

I have used your provided versions of console.c and ctring.c, due to the decreased likelihood of any errors or bugs within them. I have however provided my own versions of these titled 'consoleOld.c' and 'stringOld.c', along with their header files 'consoleOld.h' and 'stringOld.h', which as far as my testing has shown when creating them, are bug free.

## Ring 3 -> Ring 0:

VGA driver functions are exposed to the user through User.h in order for calls to the driver to be made from Ring 3 code. Each user function invokes it's Ring 0 counterpart through the use of a software interrupt, with the parameters first being moved into registers. Once an interrupt has been issued of the correct number (0x81) the handler in vgaapi pushes the parameters, calls the appropriate function and adjusts the stack pointer accordingly so that the parameters are removed from the stack and available to the Ring 0 driver function. I have expanded the parameter count that it is possible to handle to 4 by utilising the register 'ESI', however this is only used by SetColourPalette. It would be possible to expand to 5 by using 'EDI' however I have not had cause to do this with my implemented functionality.

## vga.c

### void SetResolution(int width, int height);

Simply changes the global height and width used by other methods in vga.c for drawing. This allows flexibility of the size of the drawing canvas used. However no functionality is available for VGAmodes not used Chain4Mode.

### void SetColourPalette(uint8\_t red, uint8\_t green, uint8\_t blue, int index);

Allows users to specify a new colour to be added to the palette at the index provided. This works by outputting the index to port 0x3C8 and then values for red, green and blue to port 0x3C9. This is done by making use of the HAL, specifically HAL\_OutputByteToPort.

### void SetLineStyle(int dashLength);

The main logic for line styles resides in the LineTo method. By using modulo 'n' on a counter that is incremented with every loop of LineTo, it is possible to miss setting a pixel every 'nth' pixel, thus allowing the user to set the dot/dash length by changing 'n'. SetLineStyle allows the user to specify the dash length and by adding one to this we have an appropriate value of 'n' to use.

### void ClearScreen();

To clear screen we set the entire block of memory to 0, taking into account the variable width and height of the drawing canvas, using the global \_width and \_height.

### void SetPixel(unsigned int x, unsigned int y, unsigned int colour);

First the value of x and y are checked to ensure they are within the drawing canvas, then the videomemory is accessed using the equation "0xA0000 + screenwidth \* y + x", and the appropriate byte is then set to the colour specified.

When chain mode is disabled we must now select the appropriate plane and then determine the offset. To determine the offset all we must do is divide the original equation by 4 to get the following:

(Width \* y + x) / 4

### bool MoveTo(int x, int y);

My implementation uses the concept of a cursor position, as such a value of '\_currentX' and '\_currentY' are maintained globally. Any call to draw a primitive assumes that the start position is the current values of these variables, and so users must ensure to make a call to MoveTo before making draw calls (except for Draw and Fill Rectangle, as the specification made clear that these methods should allow the user to pass in start x and y values). This method allows the user the ability to update the state of x and y.

### void DrawChar(char c, uint8\_t colour);

A char is drawn to canvas using a header containing a font stored within an array that represents a font bitmap. Due to the lack of file system implemented within the OS, I instead used an already created font array (from http://jared.geek.nz/2014/jan/custom-fonts-for-microcontrollers) and adapted the accompanying logic to work with my own implementation.

### void DrawString(const char\* str, uint8\_t colour);

This method simply uses DrawChar repeatedly to draw a string of text, incrementing the x value by the width of a char in this bitmap, after each.

### bool LineTo(int x, int y, uint8\_t colour);

a generalised version of Bresenham's Line Drawing algorithm is utilised to draw lines of all gradients. A brief explanation of Bresenham's:

- First constants of the line are calculated such as the absolute delta of X and Y, along with the signs of the deltas (these are important for generalising the algorithm, by allowing us to decrement/increment depending on the sign of the delta of the axis).

- As the standard algorithm only works for 0 <= gradient <= 1, we use the "swap" logic to deal with lines with a greater delta Y, by swapping the way we deal with x and y values whenever delta Y is larger than delta X.

- The main drawing loops functions by incrementing for each discrete value of the fastest increasing dimension, then first setting a pixel at the current x and y and then, and then based on the error value, deciding whether to increment the secondary dimension or not. This error values allows us to calculate which pixel appears closest to the line for each discrete value of the fastest increasing dimension.

Also here, is the logic for line styles (an explanation of this can be found under the documentation for SetLineStyle).

### void DrawRectangle(int width, int height, uint8\_t colour);

Lines are drawn clockwise around the perimeter of the rectangle using LineTo. It is expected that the start position of the rectangle is set via MoveTo, however for users using User\_DrawRectangle, the start point are passed in and MoveTo is called by the underlying code in user.c

For rectangles not in a standard orientation, polygon functions should be used instead.

### void FillRectangle(int width, int height, uint8\_t colour);

Lines are drawn for each y value that is occupied by the rectangle, starting at the value of \_currentX and ending at \_crruentX + width for each. Note that line style is set to 0 before drawing and then returned to it's previous value upon completion, this is so that the current line style does not lead to weird fill behaviour due to using LineTo.

For rectangles not in a standard orientation, polygon functions should be used instead.

### void DrawCircle(int radius, uint8\_t colour);

Here I have utilised the midpoint circle algorithm (source: https://en.wikipedia.org/wiki/Midpoint\_circle\_algorithm). Points within the first octant of the circle are enumerated, while also mirroring each point to the other 7 octants. As such 8 pixels are set per iteration of the loop.

### void FillCircle(int radius, uint8\_t colour);

The same method as DrawCircle is used however Lines are draw between each horizontally corresponding "mirror pair" in order to fill in the space between mirrored pixels. This algorithm could be optimised further as the current implementation will lead to the re-drawing of certain pixels within the circle, multiple times.

Again, note that line style is set to zero and then returned to its previous value in order to stop any interference with the fill.

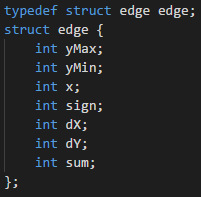
### void DrawPolygon(int \*coordinates, int numberOfVertices, uint8\_t colour);

The vertices of the polygon are to be passed in, in order, and are then looped through, using LineTo to draw lines between each vertex.

### void FillPolygon(int \*coordinates, int numberOfVertices, uint8\_t colour);

An implementation of Scanline is used in order to correctly fill pixels that are within the bounds of the polygon. This implementation was driven by psuedocode from https://hackernoon.com/computer-graphics-scan-line-polygon-fill-algorithm-3cb47283df6.

For each edge of the polygon a bucket is constructed using the following struct:



In order to set aside memory to use for the edge buckets an array of edge structs is initialised as a global. Along with this is also the function:

edge \* mallocEdge() - returns the next free edge from the \_edges array

I chose this method of providing allocated memory for my structs as the physicalmemorymanager only provides and manages chunks of memory in 4k blocks. At the time of implementation I was unfamiliar with the functionality of physicalmemorymanager and wished to move on to new functionality beyond FillPolygon, as such I did not spend the time to make use of the memory allocation facility. I am not entirely certain of the ramification of my method of acquiring memory, however I believe that the compiler will permenantly set aside the amount of memory required for my arrays even when they are set to zero by ResetPolygonMemory. As such I believe this to be an inefficient method.

We then maintain a list two lists, Edge Table and Active List. Edge table contains all the edges to begin, while Active List maintains all edges that intersect with the current scanline. For each scanline first maintenance of the lists occurs followed by drawing lines between each pair of edges that are intersected. Maintenance of the lists involves first removing any edges from the active list where yMax of the edge is equal to the current scan line, followed by adding any edges to the active list where yMin is equal to the current scan line.

Once all edges have been processed, the memory allocated for the edge buckets is reset to 0 using ResetPolygonMemory. This method also resets the memory allocated for list nodes as they are both used solely by FillPolygon.

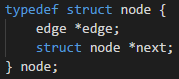
### void FloodFill(int startX, int startY, uint8\_t replacementColour);

Floodfill simply works by recursively checking the north, east, south and west pixels and setting them to the specified colour where the pixel is not already equal to that colour, starting at a point given that is within the shape.

With chain mode disabled, flood fill no longer works. Specifically the error seems related to my method of checking the value of a pixel when determining if the bounds of the shape have been reached. I’m unsure of the specific cause of this. However, as I have scanline already implemented and the deadline nears, I have decided to disable the demo of flood fill and sacrifice the functionality.

## edge\_list.c

In this file is functionality for managing a basic linked list implementation. All these methods work with pointers to existing lists using the following struct:



Like the edge stucts used by FillPolygon, a number of nodes are pre-defined with access to these handled by node \* mallocNode().

### void PrintListY(node \*head);

This function was used for debugging and simply prints out a list of edges by displaying their yMin, yMax and X values. This was useful for checking the contents of the Edge Table.

### void PrintListX(node \*head);

The contents of a list are printed displaying each edges current x value. X is a value that specifies the x intercept for a scanline and is maintained during FillPolygon as the value of scanline increases.

### int GetListSize(node \*head);

This returns the size of a given list by travelling the length of the list and recording the number of nodes.

### node \* GetNodeFromList(node \*head, int index);

This returns the node at the specified index in a list.

### void AddToList(node \*\*head, node \*nodeToAdd);

The given node is added to the end of the list.

### bool RemoveFromList(node \*\*head, edge \*edgeToRemove);

The node containing the given edge is removed from the list. Edge is used instead of node as during FillPolygon it is required to remove an edge from two lists without knowing which node contains a pointer to the edge in both lists.

### void SwapInEdgeList(node \*head, int first, int second);

The edges at the first and second index are swapped within the given list. Note, the nodes themselves are not swapped, just the edges that they point to.

### void SortEdgeListByY(node \*edgeListHead);

The given list is sorted based on the yMin of the edges. This is useful for maintaining the Edge Table in increase yMin order. This is a standard bubble sort implementation.

### void SortEdgeListByX(node \*edgeListHead);

The given list is sorted based on the x value of the edges. This is useful for maintaining the Active List which is required to be in increasing x order to properly identify edge pairs. This is a standard bubble sort implementation.

## Testing:

Testing of many of my utility functions occurred throughout development, often using console mode to print and confirm that the state of relevant values were as expected.

Shape drawing functions were testing by drawing appropriate shapes for each use case that I could imagine. For example FillPolygon was tested for a range of polygons including convex and concave, along with a varying number of vertices.

LineTo was testing in all directions on it’s own as well as during the implementation of drawing calls that built on top of LineTo.