

TFT Library

Majenko Technologies

31 March 2014

Contents

1	Theory of Operation	2
2	Primitive Drawing Functions	5
2.1	Lines and Pixels	5
2.2	Rectangles	6
2.3	Circles	6
2.4	Triangles	6
3	Advanced Drawing Facilities	8
4	Text and Fonts	9
5	Framebuffers	11
5.1	Color and Image Management	11
5.2	Sprites	12
6	Touch Screens	15
7	TFT Interface	16
8	TFTCommunicator Interface	17
9	Font Library	18
10	Color Library	19

Chapter 1

Theory of Operation

The TFT library is a modular system in which a number of different classes interact with each other to describe a specific display system. It relies heavily on *C++ polymorphism* (class inheritance) to group different objects together into drivers with a common interface. There are four basic types of object, each with its own special base class.

TFT objects are drivers for specific hardware display devices. They define how you display data on the screen, control the screen, and otherwise interact at a software level with the screen. The TFT base class also contains all the primitive drawing operations for drawing lines, boxes, circles, etc. TFT drivers communicate with the physical hardware through a TFTCommunicator based object.

TFTCommunicator objects define the physical interface to a display, be that SPI, 16-bit parallel, abstracted through an IO expander, etc. The TFT driver sends data and commands to the display through the standard interface these objects provide. This gives the maximum flexibility for any screen to be driven through any communication medium. It does have the trade-off of slightly reduced throughput though, so some objects have been combined into “Super” display objects. These contain both the TFT control code (though not the primitive drawing code) and the TFTCommunicator code and are highly optimised for high speed throughput. They are mainly used specifically for pre-defined display systems with their own embedded microcontroller.

Complementing these display drivers are also the Touch drivers. These communicate, again through TFTCommunicator objects (where such communication is apt), with the touch screen interface of a display.

The TFT class is also the parent class of a number of virtual display drivers. These can be used exactly as other display drivers, but they don’t have any physical output of their own and are used purely from within your own program for special purposes. They include the Aggregator display which can join a number of physical displays into one single virtual display, and the Framebuffer devices which can be used to build up an internal “off-screen” buffer image prior to high-speed pushing of the entire image to a display. The Framebuffer devices require a backing data store to work in, which uses the Datastore based objects. Datastore objects provide a standard interface for storing image data in a number of locations. The simplest is to store the data in the internal SRAM of the microcontroller, however external SRAM using SPI can also be used with

no change to the internal programming of your software.

Every device has a constructor, and those that require an extra device to operate, such as a TFTCommunicator device, have this extra device passed as a parameter to the constructor.

Every device also has an `initializeDevice()` member function to configure the hardware and software. This must be called once from setup for the top-level devices. It is the device's responsibility to then call the `initializeDevice()` functions of any devices it relies on.

A simple program, using the Adafruit “1.8” TFT Shield with Joystick and *microSD*, might look like this:

```
#include <TFT.h>
DSPI0 spi;
TFTDSPI mySPI(spi, 10, 8);
ST7735 tft(mySPI, ST7735::BlackTab);
void setup() {
    tft.initializeDevice();
    tft.fillScreen(Color::Black);
}

void loop() {
    tft.setPixel(
        random(0, tft.getWidth()),
        random(0, tft.getHeight()),
        random(0, 65536)
    );
}
```

Let's take a quick look at how this program works. Quite simply it plots random color pixels at random locations on the display, but let's break it down line by line.

```
#include <TFT.h>
```

This includes the TFT library into your program, just like any other library.

```
DSPI0 spi;
```

Create a new *DSPI* object called “spi” which communicates on DSPI channel 0. This is usually pins 10-13 on boards with an Arduino-like footprint.

```
TFTDSPI mySPI(spi, 10, 8);
```

This links the *DSPI* object created above to a *TFTCommunicator* object. It takes two extra parameters for the main Chip Select pin (10) and the Register/Data select pin (8). These two pins are automatically controlled by the TFT library to access the TFT display.

```
ST7735 tft(mySPI, ST7735::BlackTab);
```

This creates the actual TFT object for controlling the display. The Adafruit 1.8” display uses the *ST7735* chip, so we use that driver. This specific driver takes an extra parameter to define which actual display is on the module, since

they have released a number of different versions with different displays. See the ST7735 driver information for more detail on this option.

```
tft.initializeDevice();
```

This initializes the TFT display. It also initializes the *TFTDSPI* object “mySPI” by calling the *initializeDevice()* on that class. The *TFTDSPI* class then also configures the *DSPI* object passed to it earlier.

```
tft.fillScreen(Color::Black);
```

We want to start off with a blank canvass. Without this option the screen will be displaying garbage, or an image left over from the last time a program ran. Filling the whole screen with `Color::Black` (see the *Color Library* for a list of available color names) erases the whole contents of the screen. You can also use numeric values between 0 and 65535 or hexadecimal between 0x0000 and 0xFFFF. For the meaning of the values please see the *Color Model* section.

```
tft.setPixel(random(0, tft.getWidth()), random(0, tft.getHeight()),  
             random(0, 65535));
```

The *setPixel()* member function logically sets a single pixel on the screen to a specific color. In this case we choose the location randomly. The bounds of our random numbers are defined by the size of the screen, which we enquire using the *getWidth()* and *getHeight()* member functions.

Chapter 2

Primitive Drawing Functions

Most of the standard basic drawing functions are supported by the TFT library - lines, circles, rectangles, even rounded rectangles (rectangles with rounded corners). Most (where applicable) have both outline and filled versions.

2.1 Lines and Pixels

The simplest primitives are lines and pixels. A pixel is a single point on the screen and can be set to a specific color with the member function¹

```
void TFT.setPixel(x, y, color);
```

X is the horizontal coordinate going from left to right, and Y is the vertical coordinate going from top to bottom. Color is the 16-bit color value, or one of the predefined color names in the *Color Library*.

Lines can be drawn between two points on the screen using the member function

```
void TFT.drawLine(x1, y1, x2, y2, color);
```

X1 and Y1 mark one end of the line, and X2 and Y2 mark the other end. Again this is left-to-right and top-to-bottom coordinates.

Two special functions are provided for drawing horizontal and vertical lines. These kind of lines can often be drawn far faster by the TFT hardware than the function which draws other angled lines, so using these functions can greatly increase the speed of your display. They only take the coordinate of one end of the line (top for vertical lines, left for horizontal) and the distance the line should stretch across or down the screen. For vertical lines the function is

```
void TFT.drawVerticalLine(x, y, height, color);
```

And for horizontal lines it is

```
void TFT.drawHorizontalLine(x, y, width, color);
```

¹TFT in all these functions is a placeholder for the name you assign to your TFT device object.

2.2 Rectangles

Rectangles are drawn by providing one corner coordinate and the size of the rectangle. You can opt for filled or unfilled versions of the rectangle drawing functions. To draw an unfilled rectangle the function is:

```
void TFT.drawRectangle(x, y, width, height, color);
```

where X and Y provide the upper-left corner and the width and height stretch to the right and down the screen from that point. To draw a filled rectangle the function name changes to `fillRectangle` but the parameters remain the same:

```
void TFT.fillRectangle(x, y, width, height, color);
```

You can round the corners of a rectangle, both empty and filled, with the `drawRoundRect` and `fillRoundRect` functions:

```
void TFT.drawRoundRect(x, y, width, height, radius, color);
```

```
void TFT.fillRoundRect(x, y, width, height, radius, color);
```

In both these functions the parameters are as for the normal rectangles, with the addition of the radius parameter. This sets the radius in pixels of the curvature of the corners.

One special rectangle drawing function takes no parameters save a color. This fills the entire screen with the color and is very useful for clearing the screen of all data:

```
void TFT.fillScreen(color);
```

2.3 Circles

Unlike the rectangle functions, the coordinates for a circle have their origin at the center of the circle, not one corner (a circle has no corners...). As usual both a filled and an unfilled circle drawing function is provided:

```
void TFT.drawCircle(x, y, radius, color);
```

```
void TFT.fillCircle(x, y, radius, color);
```

The radius parameter is the number of pixels (when counted horizontally or vertically) from the center of the circle to the edge. With the X and Y coordinates at the center of the circle an example circle at position `x=100, y=100` with a radius of 10 pixels would extend 10 pixels in all directions. That is the rectangular area the circle fits in would stretch from `x=90,y=90` to `x=110,y=110`.

2.4 Triangles

Simple triangles, both filled and unfilled, can be drawn with these two functions:

```
void TFT.drawTriangle(x1, y1, x2, y2, x3, y3, color);
```

```
void TFT.fillTriangle(x1, y1, x2, y2, x3, y3, color);
```

The three sets of coordinates (x1,y1), (x2,y2) and (x3,y3) provide the three vertices of the triangle.

Chapter 3

Advanced Drawing Facilities

One of the most powerful drawing facilities the TFT library provides, and also the hardest to use well, is the windowing facility. This is not quite the same concept as windows on your computer desktop, yet at the same time isn't too far removed.

The window functions allow you to open a virtual window, or viewport, on the TFT device and in effect upload the content of that window directly to the TFT. Once a window has been opened you have to send all the pixel color information for the content of that window as a stream, and that color information will be displayed in the rectangular region defined by the window. This allows for incredibly fast updates of regions of the screen. It is best suited for display data that can easily be programatically calculated on the fly, or for filling with data pre-calculated and stored in memory.

Opening a window is much like drawing a rectangle but without any color, and you use the function:

```
void TFT.openWindow(x, y, width, height);
```

No changes to the display are made at this point, however all subsequent color data sent will be displayed within this area. You should now send enough pixels to fill the area, that is width x height pixels worth of data. The function

```
void TFT.windowData(color);
```

sends the raw pixel color data. If you open a window of size 10,10 you should send $10 \times 10 = 100$ pixels worth of data, which means `.windowData()` should be called 100 times. If you fail to send enough data (under-run) no changes past the last pixel sent will occur. If you send too much data (over-run) the display will loop back to the start of the window and overwrite the pixel data you have already sent.

Once you have sent all your pixel data you should close the window with

```
void TFT.closeWindow();
```

Pixel data is displayed starting at the top-left of the window and progresses across (left to right) and down (top to bottom) the window, much like reading this page.

Chapter 4

Text and Fonts

The TFT library utilises its own special format for embedding fonts into your software. It also comes with a pre-defined library of useful fonts (see *Font Library*).

Selecting the current font to use is as simple as calling the TFT member function

```
void TFT.setFont(font);
```

where *font* is a pointer to the font data to use (or one of the pre-defined fonts, such as `Fonts::Topaz`). *By default no font is selected, so this function must be called before any printing of text can take place.*

The color of the text (and its background) can be set using:

```
void TFT.setTextColor(color);
```

or

```
void TFT.setTextColor(foreground, background);
```

In the latter version, if the foreground and background are the same color the background becomes transparent.

Swapping the foreground and background colors can be performed using

```
void TFT.invertTextColor();
```

The current text color can be enquired using the function

```
uint16_t TFT.getTextColor();
```

By default the text will automatically wrap around the edge of the screen - that is, when text reaches the right hand margin it will automatically start a new line. If this behaviour is not desired you can turn it off (or on again) using:

```
void TFT.setTextWrap(false);
```

Change *false* to *true* to turn wrapping back on again.

Positioning the text on the screen is done using the function

```
void TFT.setCursor(x, y);
```

which starts all text printing with the upper-left corner at coordinates x,y. You can enquire the current coordinates (which automatically update as text is printed) using the functions:

```
int16_t TFT.setCursorX();
```

and

```
int16_t TFT.setCursorY();
```

You can calculate how much room a string will take up when rendered in the currently selected font using the functions:

```
uint16_t TFT.stringWidth(string);
```

and

```
uint16_t TFT.stringHeight(string);
```

They return the expected dimensions of a piece of text when rendered onto the screen. This is especially useful for calculating where to position a piece of text for centering, etc.

The TFT library uses the standard Print library for all its text handling, so printing any values can be done the same way as you would for other Print based output devices, such as the Serial device, using the functions

```
void TFT.print(...);
```

and

```
void TFT.println(...);
```

Chapter 5

Framebuffers

Framebuffers are special virtual TFT devices which store their image data in memory instead of displaying it on a screen. As well as all the standard TFT functions they also provide a few specialised functions. Framebuffers typically operate using a different color model to a TFT screen, either to conserve memory or to allow the use of enhanced effects or functions.

Because framebuffers store all their image data in memory (or other Datastore object) they have instant access to the existing image data for modification. This allows for such things as alpha blending (merging of two colors together in varying proportions), moving of portions of the image around, overlaying of one set of data over another, etc. The most advanced example of the latter is the sprite facility that framebuffers provide. Sprites are small images which exist above the normal canvas of the framebuffer. They are independant entities that exist outside the normal drawing area but are overlaid when the framebuffer is displayed. Sprites can be automatically animated and moved around the framebuffer with no need to update the content of the framebuffer.

5.1 Color and Image Management

Framebuffers provide a few extra standard functions for manipulating colors, chiefly:

```
uint16_t Framebuffer.colorAt(x, y);
```

and

```
uint16_t Framebuffer.bgColorAt(x, y);
```

The former function returns the 16-bit color value at a location in the framebuffer. If a sprite exists at that location it returns the color of that point within the sprite. This is most commonly used internally to render the framebuffer to a real TFT screen.

The latter function returns the same value from the framebuffer, but it ignores any sprites that may be in the framebuffer. This is good for manipulating the image data in the framebuffer.

The entire content of a framebuffer can be pushed out to a TFT display using one of the `update()` member functions:

```
void Framebuffer.update(tft);
```

```
void Framebuffer.update(tft, x, y);
```

The first version of the function will paste the contents of the framebuffer to the upper-left corner of the TFT screen *tft*, and the second version will paste it to a specific location within the TFT screen (upper-left corner of the framebuffer will be at *x,y*).

Depending on the color model employed by the framebuffer it may provide extra color management functions (such as palette management).

5.2 Sprites

As mentioned in the previous section sprites exist in an independant portion of the framebuffer. You can best think of the way sprites work as looking down on a landscape from high up above the clouds. The landscape is the framebuffer's image data, and the clouds that float between you and the ground are the sprites. Sprites were heavily used in early home computers and games consoles as they could be easily overlaid on a screen using hardware which made them very fast and used very few system resources. Unfortunately TFT screens don't have these kind of hardware sprite facilities, but the TFT library does emulate them at the framebuffer level. While not as efficient as hardware sprites they nonetheless have their uses, and are certainly more efficient than having to write them into the framebuffer or screen replacing data that is already there and subsequently having to replace that data when a sprite moves.

Sprites are stored in an array of bytes, with each byte representing one pixel within the sprite. The array can contain multiple copies of the sprite data each with its own array of pixels, and which copy is displayed can be easily selected or cycled through to create simple animations.

A word of caution on sprites, though. At the moment they are only officially supported on the 8-bit palletised Framebuffer object. This is due to the fact that currently they rely on the 8-bit color model of that object.

A typical sprite array might look something like this:

```
const uint8_t alien1[] =
    "..o.....o.."
    "...o....o..."
    "..oooooooo.."
    ".oo.ooo.oo."
    "oooooooooooo"
    "o.oooooooo.o"
    "o.o.....o.o"
    "...oo.oo..."

    "..o.....o.."
    "o.o....o.o"
    "o.oooooooo.o"
    "ooo.ooo.ooo"
    "oooooooooooo"
    ".ooooooooooo"
    "...o.....o.."
    ".o.....o.o";
```

Exactly how you choose to represent or store your sprite data is entirely up to you. The example above, though, is represented as a single long string (though broken onto multiple lines for readability) of the characters “.” and “o” which each represent the color indexes 46 and 111 (purely chosen for ease of text entry). The sprite itself is 11 x 8 pixels in size, but contains 2 frames, so there is actually $11 \times 8 \times 2 = 176$ characters. If you squint you can just about see that it’s an old Space Invader’s alien in its two positions.

A new sprite is added to the framebuffer by passing the pointer to the data, the size, the background color (yes, sprites are by default transparent) and the number of frames in the sprite to the `Framebuffer.addSprite()` function:

```
struct sprite *Framebuffer.addSprite(sprite, width, height,
                                     background, frames);
```

The above example would be added as something like:

```
struct sprite *mySprite = myFramebuffer.addSprite(alien1, 11, 8, '.',
                                                    2);
```

It is important to keep hold of the value returned by the `addSprite` function as it is used in all future manipulations of the sprite. If the sprite failed to be created the function will return `NULL`.

You can set the position within the framebuffer of your newly created sprite by using the `moveTo()` function:

```
void Framebuffer.moveTo(sprite, x, y);
```

which, for the above example, might be:

```
myFramebuffer.moveTo(mySprite, 20, 20);
```

You can also directly access the X and Y coordinate variables within your sprite structure, for example:

```
mySprite->xpos = 20;
```

or

```
mySprite->ypos ++;
```

You can also perform “delta” movements of the sprite, i.e., moving the sprite by an amount instead of to a coordinate, using the `moveBy()` function:

```
void Framebuffer.moveBy(sprite, dx, dy);
```

where dx and dy are the distance to move the sprite in the X and Y planes.

Animating your sprite is as simple as repeatedly calling one of the animation functions on it:

```
void Framebuffer.animate(sprite);
```

```
void Framebuffer.animatePingPong(sprite);
```

The first function will step through the animation frame in sequence. When the animation reaches the end it will start again from the beginning. The second function, however, will start playing the animation backwards when it reaches the end instead. For example, given a sprite with four frames, the `animate()`

function would cycle through the frames 0, 1, 2, 3, 0, 1, 2, 3, 0, 1... whereas `animatePingPong()` would cycle the frames thus: 0, 1, 2, 3, 2, 1, 0, 1, 2, 3, 2, 1, 0, 1... etc.

A sprite can be removed completely from the framebuffer using the `removeSprite` function:

```
void Framebuffer.removeSprite(sprite);
```

Another great feature of sprites is collision detection. You can easily find if another sprite is colliding with your sprite using the function `collidesWith()`:

```
struct sprite *Framebuffer.collidesWith(sprite);
```

This function returns the first sprite found (if any) which is touching or sharing the same space with the sprite provided. If no sprite is colliding with it the function returns `NULL`.

As well as collision detection, you can also determine if there is a sprite at any given location. The function

```
struct sprite *Framebuffer.spriteAt(x, y);
```

will return the sprite which has the coordinates `x,y` located within its bounds. If there is no sprite at that location the function returns `NULL`.

Sprites also provide a small amount of internal data storage (8 bytes) which can be used for storing information about that specific sprite. It's especially handy for identifying multiple copies of the same sprite, storing movement information on a per-sprite basis, or even something like the amount of health a monster has left. You can set data using:

```
void Framebuffer.setSprite(sprite, slot, value);
```

and read the data using:

```
int8_t Framebuffer.getSprite(sprite, slot);
```

The framebuffer also provides iterative access to all the sprites using `firstSprite()` and `nextSprite()`, for example:

```
for (struct sprite *s = myFramebuffer.firstSprite(); s; s =
    myFramebuffer.nextSprite()) {
    // do something with sprite "s"
}
```

In that example the sprite pointer “s” starts off pointing to the first sprite. While it is actually pointing to a valid sprite, do things with it, then get the next sprite and repeat.

Both functions return `NULL` if there are no sprites to work with, and `nextSprite()` returns `NULL` when it has reached the end of the list of sprites.

Chapter 6

Touch Screens

```
initializeDevice  
  sample  
    x  
    y  
    pressure
```


Chapter 7

TFT Interface

Required functions

- setPixel
- drawHorizontalLine
- drawVerticalLine
- initializeDevice
- displayOn
- displayOff
- invertDisplay

Chapter 8

TFTCommunicator Interface

readCommandX
 readDataX
 writeCommandX
 writeDataX
 streamStart
 streamEnd
 streamCommandX
 streamDataX
 blockData
 initializeDevice

Chapter 9

Font Library

Arial, sizes 8 to 40 in steps of 2
















































Arial Bold in sizes 8 to 40 in steps of 2
















































Chapter 10

Color Library

The color names are taken from the standard X11R6 color library (aka rgb.txt).

Name	Red	Green	Blue	16-bit Hex	Color
Snow	255	250	250	0xFFDF	
GhostWhite	248	248	255	0xFFDF	
WhiteSmoke	245	245	245	0xF7BE	
Gainsboro	220	220	220	0xDEFB	
FloralWhite	255	250	240	0xFFDE	
OldLace	253	245	230	0xFFBC	
Linen	250	240	230	0xFF9C	
AntiqueWhite	250	235	215	0xFF5A	
PapayaWhip	255	239	213	0xFF7A	
BlanchedAlmond	255	235	205	0xFF59	
Bisque	255	228	196	0xFF38	
PeachPuff	255	218	185	0xFED7	
NavajoWhite	255	222	173	0xFEf5	
Moccasin	255	228	181	0xFF36	
Cornsilk	255	248	220	0xFFDB	
Ivory	255	255	240	0xFFFFE	
LemonChiffon	255	250	205	0xFFD9	
Seashell	255	245	238	0xFFBD	
Honeydew	240	255	240	0xF7FE	
MintCream	245	255	250	0xF7FF	
Azure	240	255	255	0xF7FF	
AliceBlue	240	248	255	0xF7DF	
Lavender	230	230	250	0xE73F	
LavenderBlush	255	240	245	0xFF9E	
MistyRose	255	228	225	0xFF3C	
White	255	255	255	0xFFFFF	
Black	0	0	0	0x0000	
DarkSlateGray	47	79	79	0x2A69	
DimGray	105	105	105	0x6B4D	
SlateGray	112	128	144	0x7412	
LightSlateGray	119	136	153	0x7453	

Gray	190	190	190	0xBDF7	
LightGray	211	211	211	0xD69A	
MidnightBlue	25	25	112	0x18CE	
Navy	0	0	128	0x0010	
NavyBlue	0	0	128	0x0010	
CornflowerBlue	100	149	237	0x64BD	
DarkSlateBlue	72	61	139	0x49F1	
SlateBlue	106	90	205	0x6AD9	
MediumSlateBlue	123	104	238	0x7B5D	
LightSlateBlue	132	112	255	0x839F	
MediumBlue	0	0	205	0x0019	
RoyalBlue	65	105	225	0x435C	
Blue	0	0	255	0x001F	
DodgerBlue	30	144	255	0x1C9F	
DeepSkyBlue	0	191	255	0x05FF	
SkyBlue	135	206	235	0x867D	
LightSkyBlue	135	206	250	0x867F	
SteelBlue	70	130	180	0x4416	
LightSteelBlue	176	196	222	0xB63B	
LightBlue	173	216	230	0xAEDC	
PowderBlue	176	224	230	0xB71C	
PaleTurquoise	175	238	238	0xAF7D	
DarkTurquoise	0	206	209	0x067A	
MediumTurquoise	72	209	204	0x4E99	
Turquoise	64	224	208	0x471A	
Cyan	0	255	255	0x07FF	
LightCyan	224	255	255	0xE7FF	
CadetBlue	95	158	160	0x5CF4	
MediumAquamarine	102	205	170	0x6675	
Aquamarine	127	255	212	0x7FFA	
DarkGreen	0	100	0	0x0320	
DarkOliveGreen	85	107	47	0x5345	
DarkSeaGreen	143	188	143	0x8DF1	
SeaGreen	46	139	87	0x2C4A	
MediumSeaGreen	60	179	113	0x3D8E	
LightSeaGreen	32	178	170	0x2595	
PaleGreen	152	251	152	0x9FD3	
SpringGreen	0	255	127	0x07EF	
LawnGreen	124	252	0	0x7FE0	
Green	0	255	0	0x07E0	
Chartreuse	127	255	0	0x7FE0	
MediumSpringGreen	0	250	154	0x07D3	
GreenYellow	173	255	47	0xAFE5	
LimeGreen	50	205	50	0x3666	
YellowGreen	154	205	50	0x9E66	
ForestGreen	34	139	34	0x2444	
OliveDrab	107	142	35	0x6C64	

DarkKhaki	189	183	107	0xBDAD	
Khaki	240	230	140	0xF731	
PaleGoldenrod	238	232	170	0xEF55	
LightGoldenrodYellow	250	250	210	0xFFDA	
LightYellow	255	255	224	0xFFFFC	
Yellow	255	255	0	0xFFE0	
Gold	255	215	0	0xFEA0	
LightGoldenrod	238	221	130	0xEEF0	
Goldenrod	218	165	32	0xDD24	
DarkGoldenrod	184	134	11	0xBC21	
RosyBrown	188	143	143	0xBC71	
IndianRed	205	92	92	0xCAEB	
SaddleBrown	139	69	19	0x8A22	
Sienna	160	82	45	0xA285	
Peru	205	133	63	0xCC27	
Burlywood	222	184	135	0xDDD0	
Beige	245	245	220	0xF7BB	
Wheat	245	222	179	0xF6F6	
SandyBrown	244	164	96	0xF52C	
Tan	210	180	140	0xD5B1	
Chocolate	210	105	30	0xD343	
Firebrick	178	34	34	0xB104	
Brown	165	42	42	0xA145	
DarkSalmon	233	150	122	0xECAF	
Salmon	250	128	114	0xFC0E	
LightSalmon	255	160	122	0xFD0F	
Orange	255	165	0	0xFD20	
DarkOrange	255	140	0	0xFC60	
Coral	255	127	80	0xFBEA	
LightCoral	240	128	128	0xF410	
Tomato	255	99	71	0xFB08	
OrangeRed	255	69	0	0xFA20	
Red	255	0	0	0xF800	
HotPink	255	105	180	0xFB56	
DeepPink	255	20	147	0xF8B2	
Pink	255	192	203	0xFE19	
LightPink	255	182	193	0xFDB8	
PaleVioletRed	219	112	147	0xDB92	
Maroon	176	48	96	0xB18C	
MediumVioletRed	199	21	133	0xC0B0	
VioletRed	208	32	144	0xD112	
Magenta	255	0	255	0xF81F	
Violet	238	130	238	0xEC1D	
Plum	221	160	221	0xDD1B	
Orchid	218	112	214	0xDB9A	
MediumOrchid	186	85	211	0xBABA	
DarkOrchid	153	50	204	0x9999	

DarkViolet	148	0	211	0x901A	
BlueViolet	138	43	226	0x895C	
Purple	160	32	240	0xA11E	
MediumPurple	147	112	219	0x939B	
Thistle	216	191	216	0xDDFB	
Gray0	0	0	0	0x0000	
Gray1	3	3	3	0x0000	
Gray2	5	5	5	0x0020	
Gray3	8	8	8	0x0841	
Gray4	10	10	10	0x0841	
Gray5	13	13	13	0x0861	
Gray6	15	15	15	0x0861	
Gray7	18	18	18	0x1082	
Gray8	20	20	20	0x10A2	
Gray9	23	23	23	0x10A2	
Gray10	26	26	26	0x18C3	
Gray11	28	28	28	0x18E3	
Gray12	31	31	31	0x18E3	
Gray13	33	33	33	0x2104	
Gray14	36	36	36	0x2124	
Gray15	38	38	38	0x2124	
Gray16	41	41	41	0x2945	
Gray17	43	43	43	0x2945	
Gray18	46	46	46	0x2965	
Gray19	48	48	48	0x3186	
Gray20	51	51	51	0x3186	
Gray21	54	54	54	0x31A6	
Gray22	56	56	56	0x39C7	
Gray23	59	59	59	0x39C7	
Gray24	61	61	61	0x39E7	
Gray25	64	64	64	0x4208	
Gray26	66	66	66	0x4208	
Gray27	69	69	69	0x4228	
Gray28	71	71	71	0x4228	
Gray29	74	74	74	0x4A49	
Gray30	77	77	77	0x4A69	
Gray31	79	79	79	0x4A69	
Gray32	82	82	82	0x528A	
Gray33	84	84	84	0x52AA	
Gray34	87	87	87	0x52AA	
Gray35	89	89	89	0x5ACB	
Gray36	92	92	92	0x5AEB	
Gray37	94	94	94	0x5AEB	
Gray38	97	97	97	0x630C	
Gray39	99	99	99	0x630C	
Gray40	102	102	102	0x632C	
Gray41	105	105	105	0x6B4D	

Gray42	107	107	107	0x6B4D	
Gray43	110	110	110	0x6B6D	
Gray44	112	112	112	0x738E	
Gray45	115	115	115	0x738E	
Gray46	117	117	117	0x73AE	
Gray47	120	120	120	0x7BCF	
Gray48	122	122	122	0x7BCF	
Gray49	125	125	125	0x7BEF	
Gray50	127	127	127	0x7BEF	
Gray51	130	130	130	0x8410	
Gray52	133	133	133	0x8430	
Gray53	135	135	135	0x8430	
Gray54	138	138	138	0x8C51	
Gray55	140	140	140	0x8C71	
Gray56	143	143	143	0x8C71	
Gray57	145	145	145	0x9492	
Gray58	148	148	148	0x94B2	
Gray59	150	150	150	0x94B2	
Gray60	153	153	153	0x9CD3	
Gray61	156	156	156	0x9CF3	
Gray62	158	158	158	0x9CF3	
Gray63	161	161	161	0xA514	
Gray64	163	163	163	0xA514	
Gray65	166	166	166	0xA534	
Gray66	168	168	168	0xAD55	
Gray67	171	171	171	0xAD55	
Gray68	173	173	173	0xAD75	
Gray69	176	176	176	0xB596	
Gray70	179	179	179	0xB596	
Gray71	181	181	181	0xB5B6	
Gray72	184	184	184	0xBDD7	
Gray73	186	186	186	0xBDD7	
Gray74	189	189	189	0xBDF7	
Gray75	191	191	191	0xBDF7	
Gray76	194	194	194	0xC618	
Gray77	196	196	196	0xC638	
Gray78	199	199	199	0xC638	
Gray79	201	201	201	0xCE59	
Gray80	204	204	204	0xCE79	
Gray81	207	207	207	0xCE79	
Gray82	209	209	209	0xD69A	
Gray83	212	212	212	0xD6BA	
Gray84	214	214	214	0xD6BA	
Gray85	217	217	217	0xDEDB	
Gray86	219	219	219	0xDEDB	
Gray87	222	222	222	0XDEFB	
Gray88	224	224	224	0xE71C	

Gray89	227	227	227	0xE71C	
Gray90	229	229	229	0xE73C	
Gray91	232	232	232	0xEF5D	
Gray92	235	235	235	0xEF5D	
Gray93	237	237	237	0xEF7D	
Gray94	240	240	240	0xF79E	
Gray95	242	242	242	0xF79E	
Gray96	245	245	245	0xF7BE	
Gray97	247	247	247	0xF7BE	
Gray98	250	250	250	0xFFDF	
Gray99	252	252	252	0xFFFF	
Gray100	255	255	255	0xFFFF	
DarkGray	169	169	169	0xAD55	
DarkBlue	0	0	139	0x0011	
DarkCyan	0	139	139	0x0451	
DarkMagenta	139	0	139	0x8811	
DarkRed	139	0	0	0x8800	
LightGreen	144	238	144	0x9772	