TFT Extension Version 2

Contents

[Overview 1](#_Toc417737683)

[Button Types 1](#_Toc417737684)

[Primary Functions 1](#_Toc417737685)

[Secondary Functions 2](#_Toc417737686)

[Special Functions 3](#_Toc417737687)

[Getting Started 5](#_Toc417737688)

# Overview

This library is designed to be an extension to Henning Karlsen's UTFT and UTouch libraries. Essentially it is quite similar to his UTFT\_Buttons library, because it too has its own button types. This library is a more simpler yet more robust version of my first version, but it wasn't really intended for beginners. It was designed to simplify the main functions of my last library, mainly the TouchButton, TouchCircle and TouchTriangle function along with the Toggling and delay functions too. In addition to simplifying these functions, the library also used a significantly less amount of precious SRAM, which is usually the limiting factor for any Arduino board. More SRAM can lead to less bugs down the road, especially if an excessive amount of Strings are used with the Hard/SoftwareSerial libraries (that are not stored in the Flash memory) being used throughout the rest of the sketch. Same goes for the Ethernet, WIFI or GSM examples + libraries.

## Button Types

I kept with the usual Box, Circle and Triangle type buttons but I also added a special type of button that will be discussed later.

# Primary Functions

* Touch
* Toggle
* Delay

**Touch:** The bare basic functionality of a button, you press it or hold it and it will return true until it is released.

**Toggle:** This will hold the button's state (pressed or released) until it is pressed again.

**Delay:** This one will look to see if it is being touched within its coordinates and start a timer, if the user entered time has passed, the button will return true. By default the timer is set to 1000, (1000 milliseconds) or 1 second. *(1000 ms = 1s provided the user has not adjusted the Arduino's interrupt routine or timers)* The user can also set their own delay time if needed. The delay range is between 0 and 4294967295 (4294967 seconds or 71582 minutes or 1193 hours or about 49 days, yeesh O\_o)

\*\*Example sketch is given\*\*

# Secondary Functions

* Coords
* Colors
* Padding
* Text
* HeaderText
* FooterText
* HeadFootText \*combination of the above two\*
* Draw
* SetState
* ReDraw
* Polygon \*Unique to only the Triangle class\*

**Coords:** As one may be able to tell, this function is used to set the coordinates of the button on the screen. This function although used by all the different button types is unique for each.

* Box buttons take 2 pairs of X and Y coordinates.
* Circle buttons take one set of X and Y coordinates + a Radius (in pixels).
* Triangle buttons are special as they can accept either 3 pairs of X and Y coordinates **or** one set of X and Y coordinates + a base length and a degree.

**Colors:** Like the Coords function, this too has a slight difference between the buttons, however only by one variable. Both the Circle and Triangle buttons take two colors (pressed and released) and set whether the button is to be filled or not. Box buttons have an additional flag, *round,* which as one would expect, sets whether the button will have rounded edges or not.

\*The user can also set just one color to be used as both pressed and released states.

\*Also note that all the colors in this library require the value to be in 565 format (0x0 - 0xFFFF). If by some chance the user is using a modified, old or fake version of the UTFT library, the macro RGB\_to\_565 is given to convert 3 bytes into one unsigned int or word variable.

**Padding:** Only two of the three types of buttons support this function at this time, Box and Circle. This function takes two conditions, a padding value (in pixels and does not check if the value exceeds the size of the button) and a color. The default padding value set by the constructor is 2 pixels and the color is WHITE (0xFFFF)

\*I haven't figured out how to add a padding to the triangle buttons yet, given two types of coordinates can be used.

**Text:** This function is the same for all types of buttons. It takes 4 conditions, a color, a size (Big or Small, does not work with custom fonts) and a text indicating pressed and released.

\*Like my first library version, it does not check if the font goes beyond the boundaries of the button.

\* Also like the colors, the user can enter just one string of text.

**HeaderText**: Allows text to be displayed above the button

**FooterText**: Allows Text to be displayed below the button

**HeadFootText**: combines both into one function.

**Draw:** As the name implies, it draws the button on the screen. This also takes one condition, a flag variable to tell it to draw the button with or without text.

\*Default is set to include the text if there is any.

**SetState:** This one is not really a function that will be used too often, but what it does is it allows the user to manually set the state of the button. So if the user wants the button to be seen as "Pressed" when it is first drawn, it can be done.

**ReDraw:** Another function that may not be used too often, unless maybe menus are being used. It simply sets a variable to allow the button to be redrawn on the screen. This is because the coding for the draw function is to draw the button once, then only draw again if the state has changed. This function resets the variable *lastState* thus allowing the button to be redrawn on the screen.

**Polygon:** Polygon is a function only available to the triangle class, and allows the user to draw a polygon with equal length sides, however, it does not act as a button…yet. I may break it out into its own class in the future. Why I didn’t make it a touchable function in my last library, I don’t know. The only reason it is not a function now is because it is late and I don't want to make it its own class just yet. But hey, I did make another class that can become any polygon the user wants, keep reading.

# Special Functions

In my opinion, no library would really be complete unless the user gets a little more than core features. With that said I have added a few special types of features.

* Radio (Radio Buttons to be more specific)
* Slider
* Swipe
* CustomButton
* MobileKeyboard
* ProgressBar (only vertical at this time)

\*\*Example sketch is given\*\*

**Radio:** Like my first library version, I included a way for buttons to be arranged in groups but only allowing one to be seen as pressed out of the group. For those unfamiliar with the concept of radio buttons, this was something used on old style car radios, where only one station was selected out of 6 buttons. If a button was pressed, it would remain pressed until another was pressed, then the new button would be the one remaining pressed as all the others were depressed.

This class allows for one group of buttons to be used at one time, so only Box buttons, Circle buttons or Triangle buttons. *User is not allowed to mix the button types*.

**Slider:** This class is basically a simple bar slider that can work either vertically or horizontally depending on what is set as the orientation in the slider's constructor. It can also be size to any width or height (within the dimensions of the screen of course).

Like the main buttons, this also uses Coords, Colors, Padding, Draw, Touch, SetState with a few extras like SetDirection which are Left to Right, Right to Left, or Top to Bottom, Bottom to Top (LTR, RTL or TTB, BTT) depending on the orientation set in the constructor. It also has SetRange, SetValue and GetValue, which may be self explanatory but I will discuss them too, don't worry.

The first functions mentioned like Coords, Colors, Padding … etc, are exactly the same as the one used for the Box buttons. (If you haven't read about them already, please go back and do so) The other functions, again, should be self explanatory but here is the break down.

* **SetDirection:**  This is a simple boolean variable that sets whether the slider fills from either the top to the bottom or vice versa, or from the left to the right or vice versa depending on the orientation set in the constructor.
* **SetRange:** This takes 3 variables, low (long), high (long) and increments (byte). Low is obviously the lowest value that will be returned from the slider, and yes it can be a negative value. High is the highest value that will be returned. Increments is the factor in which the range will be run through. Currently the defaults are 0 for low, 10 for high and 1 for increments. In this case when the slider is moved, it will go through 0 - 10,by increments of 1, so if you change increments to 2, it will increase by a factor of 2.

\*increments cannot be set to anything less than 1, if user enters 0 as a value, it will automatically be set to 1.

* **SetValue:** This does just what the name implies, it will allow the user to manually set the value of the slider to whatever is set within the sliders value range.

**Swipe:** Another one of my past functions that I thought might be useful. This only requires two functions to work, Coords, which is exactly like the Coords function used for the Box buttons and the function SwipeFromArea.

* **SwipeFromArea:** This special function allows the user to register a swiping motion from the specified coordinates in any of the 8 directions. The user can also specify a specific direction with the keywords **Swipe\_** and the direction (up, down, left right, and even upLeft, upRight, downLeft and downRight) Example: *Swipe\_up*. The user can return any swiped direction if **Free\_Swipe** is set as an argument.

A new feature to this function that I thought could maybe come in handy is to be able to mix any of the 8 directions. This means a user can check for both Swipe\_up and Swipe\_down with the OR operator **|**. To do this, simply enter (Swipe\_up | Swipe\_down) and if either direction is swiped, the function will return true.

\*One other thing I forgot to mention was the length of the swipe to be checked. In addition to the direction(s), the user also needs to specify a length (in pixels).

\*The defaults are Free\_Swipe and 50.

**CustomButton:**  This class was quite the pain in the butt to get to work, but I finally got it! This class is an exceptionally special feature to this library as it uses the three main button functions, Touch, Toggle and Delay, as well as Colors and Draw. However the thing with this class is that the user needs to pass the buttons coordinates into the Touch, Toggle and Delay functions as there is no Coords function. Also the way the coordinates are passed in and the type (int, byte, long) the coordinates need to be declare as, is also special. This one is kinda difficult to explain, so I will include an example sketch to show the user just how to use this class. I do have one request, and that is to please not make random, garbled polygons and expect them to work because they might not.

**MobileKeyboard:** I added in my keyboard from my last library. It can work on screens varying from 3.2" up to 7". Default is currently set to 320x240 displays, and can allow the keyboard to be stretched for bigger buttons.

See Keyboard example.

**ProgressBar**: Like my sliders, this allows the user to map their input data to a visual bar. It allows for 3 colors to be assigned to a percentage.

# Getting Started

To use any of the classes with the exception of the Radio buttons, the user must first have the base class that takes in the UTFT and UTouch class references. Below is a simple sketch that shows the user everything needed to get a single Box button to work.

|  |
| --- |
| #include <UTFT.h>  #include <UTouch.h>  #include <TFT\_ExtensionV2.h>  UTFT myGLCD(ITDB32S, 38, 39, 40, 41); // my display pins  UTouch myTouch( 6, 5, 4, 3, 2); // my display pin  Base B(&myGLCD, &myTouch); // Base class, NEEDED!!!  Box myButton(&B); // The type of button, "Box" and the button object itself "myButton". The **&B** is a reference to the base class that allows everything in the Box class to work. Without this, the display will either glitch or just not work at all. (No actual harm will occur, but the screen just won't work until the user has a reference to the base class)  void setup()  {  myGLCD.InitLCD(LANDSCAPE); // standard display initialize function  myGLCD.clrScr(); // A nice fresh screen  myTouch.InitTouch(LANDSCAPE); // standard touch initialize function  myTouch.setPrecision(PREC\_MEDIUM); // Touch precision  myGLCD.fillScr(BLACK); // make the entire screen black, because why not.  myButton.Coords(10, 10, 60, 60); // The button coordinates, x1, y1, x2, y2. **Do not confuse with X, Y, width, height as they are not used in this library**  myButton.Colors(GREEN, RED, FILL, ROUNDED); // In my first library I used ROUNDED and NOTROUNDED, now it is changed to ROUNDED and SQUARED.  myButton.Text(BLACK, Big, "A", "B"); // optional  myButton.Padding(3, BLUE); // optional  myButton.Draw(); // draw it the first time, optional but without it, you won't see the button until it is pressed once.  }  void loop()  {  /\* Only use one at a time \*/  myButton.Touch();  // myButton.Toggle();  // myButton.Delay(500);  } |

End notes:

Trying to make the keyboard class smaller and more efficient, example KB\_Test will eventually be the final result once it is complete.