TFT_eSPI Library by Bodmer

PDF of commands syntax made by G.Ponte

open Bookmarks for see single command

Look at end for see images of connections

For Esp32 Dev Kit 1 and Ili9488 touch screen

Fast link to

TFT_eSP

Touch functions

Smooth font functions

Button class

Sprite class

```
/*****************************************************************************************
** Function name:           TFT_eSPI Library
** Description:             Arduino TFT graphics library targeted at 32 bit
*****************************************************************************************/
/***************************************************
  Arduino TFT graphics library targeted at 32 bit
  processors such as ESP32, ESP8266 and STM32.

  This is a stand-alone library that contains the
  hardware driver, the graphics functions and the
  proportional fonts.

  The larger fonts are Run Length Encoded to reduce their
  size.

  Created by Bodmer 2/12/16
  Last update by Bodmer 20/03/20
 ****************************************************/


#include "TFT_eSPI.h"

#if defined (ESP32)
  #include "Processors/TFT_eSPI_ESP32.c"
#elif defined (ESP8266)
  #include "Processors/TFT_eSPI_ESP8266.c"
#elif defined (STM32) // (_VARIANT_ARDUINO_STM32_) stm32_def.h
  #include "Processors/TFT_eSPI_STM32.c"
#elif defined (ARDUINO_ARCH_RP2040) // Raspberry Pi Pico
  #include "Processors/TFT_eSPI_RP2040.c"
#else
  #include "Processors/TFT_eSPI_Generic.c"
#endif

#ifndef SPI_BUSY_CHECK
  #define SPI_BUSY_CHECK
#endif

// Clipping macro for pushImage
#define PI_CLIP                             \
  if (_vpOoB) return;                       \
  x+= _xDatum;                              \
  y+= _yDatum;                              \
                                            \
  if ((x >= _vpW) || (y >= _vpH)) return;   \
                                            \
  int32_t dx = 0;                           \
  int32_t dy = 0;                           \
  int32_t dw = w;                           \
  int32_t dh = h;                           \
                                            \
  if (x < _vpX) { dx = _vpX - x; dw -= dx; x = _vpX; } \
  if (y < _vpY) { dy = _vpY - y; dh -= dy; y = _vpY; } \
                                            \
  if ((x + dw) > _vpW ) dw = _vpW - x;      \
  if ((y + dh) > _vpH ) dh = _vpH - y;      \
                                            \
  if (dw < 1 || dh < 1) return;
```

```
/***************************************************************************************
** Function name:           begin_tft_write (was called spi_begin)
** Description:             Start SPI transaction for writes and select TFT
***************************************************************************************/
inline void TFT_eSPI::begin_tft_write(void){
#if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS) && !defined(TFT_PARALLEL_8_BIT)
  if (locked) {
    locked = false; // Flag to show SPI access now unlocked
    spi.beginTransaction(SPISettings(SPI_FREQUENCY, MSBFIRST, TFT_SPI_MODE)); // RP2040 SDK -> 68us delay!
    CS_L;
    SET_BUS_WRITE_MODE;  // Some processors (e.g. ESP32) allow recycling the tx buffer when rx is not used
  }
#else
  CS_L;
  SET_BUS_WRITE_MODE;
#endif

}
```

```
/***************************************************************************************
** Function name:           end_tft_write (was called spi_end)
** Description:             End transaction for write and deselect TFT
***************************************************************************************/
inline void TFT_eSPI::end_tft_write(void){
#if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS) && !defined(TFT_PARALLEL_8_BIT)
  if(!inTransaction) {      // Flag to stop ending tranaction during multiple graphics calls
    if (!locked) {          // Locked when beginTransaction has been called
      locked = true;        // Flag to show SPI access now locked
      SPI_BUSY_CHECK;       // Check send complete and clean out unused rx data
      CS_H;
      spi.endTransaction(); //  RP2040 SDK -> 0.7us delay
    }
    SET_BUS_READ_MODE;      // In case SPI has been configured for tx only
  }
#else
  if(!inTransaction) {SPI_BUSY_CHECK; CS_H; SET_BUS_READ_MODE;}
#endif
}
```

```
/***************************************************************************************
** Function name:           begin_tft_read  (was called spi_begin_read)
** Description:             Start transaction for reads and select TFT
***************************************************************************************/
// Reads require a lower SPI clock rate than writes
inline void TFT_eSPI::begin_tft_read(void){
  DMA_BUSY_CHECK; // Wait for any DMA transfer to complete before changing SPI settings
#if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS) && !defined(TFT_PARALLEL_8_BIT)
  if (locked) {
    locked = false;
    spi.beginTransaction(SPISettings(SPI_READ_FREQUENCY, MSBFIRST, TFT_SPI_MODE));
    CS_L;
  }
#else
  #if !defined(TFT_PARALLEL_8_BIT)
    spi.setFrequency(SPI_READ_FREQUENCY);
  #endif
   CS_L;
#endif
  SET_BUS_READ_MODE;
}
```

```
/***************************************************************************************
** Function name:           setViewport
** Description:             Set the clipping region for the TFT screen
***************************************************************************************/
void TFT_eSPI::setViewport(int32_t x, int32_t y, int32_t w, int32_t h, bool vpDatum)
{
  // Viewport metrics (not clipped)
  _xDatum  = x; // Datum x position in screen coordinates
  _yDatum  = y; // Datum y position in screen coordinates
  _xWidth  = w; // Viewport width
  _yHeight = h; // Viewport height

  // Full size default viewport
  _vpDatum = false; // Datum is at top left corner of screen (true = top left of viewport)
  _vpOoB   = false; // Out of Bounds flag (true is all of viewport is off screen)
  _vpX = 0;         // Viewport top left corner x coordinate
  _vpY = 0;         // Viewport top left corner y coordinate
  _vpW = width();   // Equivalent of TFT width  (Nb: viewport right edge coord + 1)
  _vpH = height();  // Equivalent of TFT height (Nb: viewport bottom edge coord + 1)

  // Clip viewport to screen area
  if (x<0) { w += x; x = 0; }
  if (y<0) { h += y; y = 0; }
  if ((x + w) > width() ) { w = width()  - x; }
  if ((y + h) > height() ) { h = height() - y; }

  //Serial.print(" x=");Serial.print( x);Serial.print(", y=");Serial.print( y);
  //Serial.print(", w=");Serial.print(w);Serial.print(", h=");Serial.println(h);

  // Check if viewport is entirely out of bounds
  if (w < 1 || h < 1)
  {
    // Set default values and Out of Bounds flag in case of error
    _xDatum = 0;
    _yDatum = 0;
    _xWidth  = width();
    _yHeight = height();
    _vpOoB = true;      // Set Out of Bounds flag to inhibit all drawing
    return;
  }

  if (!vpDatum)
  {
    _xDatum = 0; // Reset to top left of screen if not using a viewport datum
    _yDatum = 0;
    _xWidth  = width();
    _yHeight = height();
  }

  // Store the clipped screen viewport metrics and datum position
  _vpX = x;
  _vpY = y;
  _vpW = x + w;
  _vpH = y + h;
  _vpDatum = vpDatum;

  //Serial.print(" _xDatum=");Serial.print( _xDatum);Serial.print(", _yDatum=");Serial.print( _yDatum);
  //Serial.print(", _xWidth=");Serial.print(_xWidth);Serial.print(", _yHeight=");Serial.println(_yHeight);

  //Serial.print(" _vpX=");Serial.print( _vpX);Serial.print(", _vpY=");Serial.print( _vpY);
  //Serial.print(", _vpW=");Serial.print(_vpW);Serial.print(", _vpH=");Serial.println(_vpH);

}
```

```
/***************************************************************************************
** Function name:           checkViewport
** Description:             Check if any part of specified area is visible in viewport
***************************************************************************************/
// Note: Setting w and h to 1 will check if coordinate x,y is in area
bool TFT_eSPI::checkViewport(int32_t x, int32_t y, int32_t w, int32_t h)
{
  if (_vpOoB) return false;
  x+= _xDatum;
  y+= _yDatum;

  if ((x >= _vpW) || (y >= _vpH)) return false;

  int32_t dx = 0;
  int32_t dy = 0;
  int32_t dw = w;
  int32_t dh = h;

  if (x < _vpX) { dx = _vpX - x; dw -= dx; x = _vpX; }
  if (y < _vpY) { dy = _vpY - y; dh -= dy; y = _vpY; }

  if ((x + dw) > _vpW ) dw = _vpW - x;
  if ((y + dh) > _vpH ) dh = _vpH - y;

  if (dw < 1 || dh < 1) return false;

  return true;
}
```

```
/***************************************************************************************
** Function name:           resetViewport
** Description:             Reset viewport to whle TFT screen, datum at 0,0
***************************************************************************************/
void TFT_eSPI::resetViewport(void)
{
  // Reset viewport to the whole screen (or sprite) area
  _vpDatum = false;
  _vpOoB   = false;
  _xDatum = 0;
  _yDatum = 0;
  _vpX = 0;
  _vpY = 0;
  _vpW = width();
  _vpH = height();
  _xWidth  = width();
  _yHeight = height();
}
```

```
/***************************************************************************************
** Function name:           getViewportX
** Description:             Get x position of the viewport datum
***************************************************************************************/
int32_t  TFT_eSPI::getViewportX(void)
{
  return _xDatum;
}
```

```
/***************************************************************************************
** Function name:           getViewportY
** Description:             Get y position of the viewport datum
***************************************************************************************/
int32_t  TFT_eSPI::getViewportY(void)
{
  return _yDatum;
}
```

```
/**************************************************************************************
** Function name:           getViewportWidth
** Description:             Get width of the viewport
**************************************************************************************/
int32_t TFT_eSPI::getViewportWidth(void)
{
  return _xWidth;
}
```

```
/**************************************************************************************
** Function name:           getViewportHeight
** Description:             Get height of the viewport
**************************************************************************************/
int32_t TFT_eSPI::getViewportHeight(void)
{
  return _yHeight;
}
```

```
/***************************************************************************************
** Function name:           getViewportDatum
** Description:             Get datum flag of the viewport (true = viewport corner)
***************************************************************************************/
bool  TFT_eSPI::getViewportDatum(void)
{
  return _vpDatum;
}
```

```
/***************************************************************************************
** Function name:           frameViewport
** Description:             Draw a frame inside or outside the viewport of width w
***************************************************************************************/
void TFT_eSPI::frameViewport(uint16_t color, int32_t w)
{
  // Save datum position
  bool _dT = _vpDatum;

  // If w is positive the frame is drawn inside the viewport
  // a large positive width will clear the screen inside the viewport
  if (w>0)
  {
    // Set vpDatum true to simplify coordinate derivation
    _vpDatum = true;
    fillRect(0, 0, _vpW - _vpX, w, color);               // Top
    fillRect(0, w, w, _vpH - _vpY - w - w, color);       // Left
    fillRect(_xWidth - w, w, w, _yHeight - w - w, color); // Right
    fillRect(0, _yHeight - w, _xWidth, w, color);        // Bottom
  }
  else
  // If w is negative the frame is drawn outside the viewport
  // a large negative width will clear the screen outside the viewport
  {
    w = -w;

    // Save old values
    int32_t _xT = _vpX; _vpX = 0;
    int32_t _yT = _vpY; _vpY = 0;
    int32_t _wT = _vpW;
    int32_t _hT = _vpH;

    // Set vpDatum false so frame can be drawn outside window
    _vpDatum = false; // When false the full width and height is accessed
    _vpH = height();
    _vpW = width();

    // Draw frame
    fillRect(_xT - w - _xDatum, _yT - w - _yDatum, _wT - _xT + w + w, w, color); // Top
    fillRect(_xT - w - _xDatum, _yT - _yDatum, w, _hT - _yT, color);             // Left
    fillRect(_wT - _xDatum, _yT - _yDatum, w, _hT - _yT, color);                 // Right
    fillRect(_xT - w - _xDatum, _hT - _yDatum, _wT - _xT + w + w, w, color);     // Bottom

    // Restore old values
    _vpX = _xT;
    _vpY = _yT;
    _vpW = _wT;
    _vpH = _hT;
  }

  // Restore vpDatum
  _vpDatum = _dT;
}
```

```
/***************************************************************************************
** Function name:           end_tft_read (was called spi_end_read)
** Description:             End transaction for reads and deselect TFT
***************************************************************************************/
inline void TFT_eSPI::end_tft_read(void){
#if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS) && !defined(TFT_PARALLEL_8_BIT)
  if(!inTransaction) {
    if (!locked) {
      locked = true;
      CS_H;
      spi.endTransaction();
    }
  }
#else
  #if !defined(TFT_PARALLEL_8_BIT)
    spi.setFrequency(SPI_FREQUENCY);
  #endif
   if(!inTransaction) {CS_H;}
#endif
  SET_BUS_WRITE_MODE;

// The ST7796 appears to need a 4ms delay after a CGRAM read, otherwise subsequent writes will fail!
#ifdef ST7796_DRIVER
  delay(4);
#endif
}
```

```
/***************************************************************************************
** Function name:           Legacy - deprecated
** Description:             Start/end transaction
***************************************************************************************/
  void TFT_eSPI::spi_begin()      {begin_tft_write();}
  void TFT_eSPI::spi_end()       {  end_tft_write();}
  void TFT_eSPI::spi_begin_read()  {begin_tft_read(); }
  void TFT_eSPI::spi_end_read()    {  end_tft_read(); }
```

```
/***************************************************************************************
** Function name:           TFT_eSPI
** Description:             Constructor , we must use hardware SPI pins
***************************************************************************************/
TFT_eSPI::TFT_eSPI(int16_t w, int16_t h)
{

// The control pins are deliberately set to the inactive state (CS high) as setup()
// might call and initialise other SPI peripherals which would could cause conflicts
// if CS is floating or undefined.
#ifdef TFT_CS
  pinMode(TFT_CS, OUTPUT);
  digitalWrite(TFT_CS, HIGH); // Chip select high (inactive)
#endif

// Configure chip select for touchscreen controller if present
#ifdef TOUCH_CS
  pinMode(TOUCH_CS, OUTPUT);
  digitalWrite(TOUCH_CS, HIGH); // Chip select high (inactive)
#endif

#ifdef TFT_WR
  pinMode(TFT_WR, OUTPUT);
  digitalWrite(TFT_WR, HIGH); // Set write strobe high (inactive)
#endif

#ifdef TFT_DC
  pinMode(TFT_DC, OUTPUT);
  digitalWrite(TFT_DC, HIGH); // Data/Command high = data mode
#endif

#ifdef TFT_RST
  if (TFT_RST >= 0) {
    pinMode(TFT_RST, OUTPUT);
    digitalWrite(TFT_RST, HIGH); // Set high, do not share pin with another SPI device
  }
#endif

#if defined (TFT_PARALLEL_8_BIT)

  // Make sure read is high before we set the bus to output
  pinMode(TFT_RD, OUTPUT);
  digitalWrite(TFT_RD, HIGH);

  // Set TFT data bus lines to output
  pinMode(TFT_D0, OUTPUT); digitalWrite(TFT_D0, HIGH);
  pinMode(TFT_D1, OUTPUT); digitalWrite(TFT_D1, HIGH);
  pinMode(TFT_D2, OUTPUT); digitalWrite(TFT_D2, HIGH);
  pinMode(TFT_D3, OUTPUT); digitalWrite(TFT_D3, HIGH);
  pinMode(TFT_D4, OUTPUT); digitalWrite(TFT_D4, HIGH);
  pinMode(TFT_D5, OUTPUT); digitalWrite(TFT_D5, HIGH);
  pinMode(TFT_D6, OUTPUT); digitalWrite(TFT_D6, HIGH);
  pinMode(TFT_D7, OUTPUT); digitalWrite(TFT_D7, HIGH);

  CONSTRUCTOR_INIT_TFT_DATA_BUS;

#endif

  _init_width  = _width  = w; // Set by specific xxxxx_Defines.h file or by users sketch
  _init_height = _height = h; // Set by specific xxxxx_Defines.h file or by users sketch

  // Reset the viewport to the whole screen
  resetViewport();

  rotation  = 0;
  cursor_y  = cursor_x  = 0;
  textfont  = 1;
  textsize  = 1;
  textcolor   = bitmap_fg = 0xFFFF; // White
  textbgcolor = bitmap_bg = 0x0000; // Black
  padX = 0;             // No padding
  isDigits   = false;   // No bounding box adjustment
  textwrapX  = true;    // Wrap text at end of line when using print stream
  textwrapY  = false;   // Wrap text at bottom of screen when using print stream
  textdatum = TL_DATUM; // Top Left text alignment is default
  fontsloaded = 0;

  _swapBytes = false;   // Do not swap colour bytes by default

  locked = true;          // Transaction mutex lock flag to ensure begin/endTranaction pairing
  inTransaction = false;   // Flag to prevent multiple sequential functions to keep bus access open
  lockTransaction = false; // start/endWrite lock flag to allow sketch to keep SPI bus access open

  _booted   = true;     // Default attributes
```

```
  _utf8     = true;     // UTF8 decoding enabled

#ifdef FONT_FS_AVAILABLE
  fs_font  = true;     // Smooth font filing system or array (fs_font = false) flag
#endif

#if defined (ESP32) && defined (CONFIG_SPIRAM_SUPPORT)
  if (psramFound()) _psram_enable = true; // Enable the use of PSRAM (if available)
  else
#endif
  _psram_enable = false;

  addr_row = 0xFFFF;  // drawPixel command length optimiser
  addr_col = 0xFFFF;  // drawPixel command length optimiser

  _xPivot = 0;
  _yPivot = 0;

// Legacy support for bit GPIO masks
  cspinmask = 0;
  dcpinmask = 0;
  wrpinmask = 0;
  sclkpinmask = 0;

// Flags for which fonts are loaded
#ifdef LOAD_GLCD
  fontsloaded  = 0x0002; // Bit 1 set
#endif

#ifdef LOAD_FONT2
  fontsloaded |= 0x0004; // Bit 2 set
#endif

#ifdef LOAD_FONT4
  fontsloaded |= 0x0010; // Bit 4 set
#endif

#ifdef LOAD_FONT6
  fontsloaded |= 0x0040; // Bit 6 set
#endif

#ifdef LOAD_FONT7
  fontsloaded |= 0x0080; // Bit 7 set
#endif

#ifdef LOAD_FONT8
  fontsloaded |= 0x0100; // Bit 8 set
#endif

#ifdef LOAD_FONT8N
  fontsloaded |= 0x0200; // Bit 9 set
#endif

#ifdef SMOOTH_FONT
  fontsloaded |= 0x8000; // Bit 15 set
#endif
}
```

```
/***************************************************************************************
** Function name:           begin
** Description:             Included for backwards compatibility
***************************************************************************************/
void TFT_eSPI::begin(uint8_t tc)
{
 init(tc);
}
```

```
/***************************************************************************************
** Function name:           init (tc is tab colour for ST7735 displays only)
** Description:             Reset, then initialise the TFT display registers
***************************************************************************************/
void TFT_eSPI::init(uint8_t tc)
{
  if (_booted)
  {
#if !defined (ESP32) && !defined(TFT_PARALLEL_8_BIT) && !defined(ARDUINO_ARCH_RP2040)
  // Legacy bitmasks for GPIO
  #if defined (TFT_CS) && (TFT_CS >= 0)
    cspinmask = (uint32_t) digitalPinToBitMask(TFT_CS);
  #endif

  #if defined (TFT_DC) && (TFT_DC >= 0)
    dcpinmask = (uint32_t) digitalPinToBitMask(TFT_DC);
  #endif

  #if defined (TFT_WR) && (TFT_WR >= 0)
    wrpinmask = (uint32_t) digitalPinToBitMask(TFT_WR);
  #endif

  #if defined (TFT_SCLK) && (TFT_SCLK >= 0)
    sclkpinmask = (uint32_t) digitalPinToBitMask(TFT_SCLK);
  #endif

  #if defined (TFT_SPI_OVERLAP) && defined (ESP8266)
    // Overlap mode SD0=MISO, SD1=MOSI, CLK=SCLK must use D3 as CS
    //    pins(int8_t sck, int8_t miso, int8_t mosi, int8_t ss);
    //spi.pins(       6,         7,         8,         0);
    spi.pins(6, 7, 8, 0);
  #endif

  spi.begin(); // This will set HMISO to input

#else
  #if !defined(TFT_PARALLEL_8_BIT)
    #if defined (TFT_MOSI) && !defined (TFT_SPI_OVERLAP) && !defined(ARDUINO_ARCH_RP2040)
      spi.begin(TFT_SCLK, TFT_MISO, TFT_MOSI, -1);
    #else
      spi.begin();
    #endif
  #endif
#endif
    lockTransaction = false;
    inTransaction = false;
    locked = true;

    INIT_TFT_DATA_BUS;



#ifdef TFT_CS
  // Set to output once again in case ESP8266 D6 (MISO) is used for CS
  pinMode(TFT_CS, OUTPUT);
  digitalWrite(TFT_CS, HIGH); // Chip select high (inactive)
#elif defined (ESP8266) && !defined (TFT_PARALLEL_8_BIT)
  spi.setHwCs(1); // Use hardware SS toggling
#endif



  // Set to output once again in case ESP8266 D6 (MISO) is used for DC
#ifdef TFT_DC
    pinMode(TFT_DC, OUTPUT);
    digitalWrite(TFT_DC, HIGH); // Data/Command high = data mode
#endif

    _booted = false;
    end_tft_write();
  } // end of: if just _booted

  // Toggle RST low to reset
#ifdef TFT_RST
  if (TFT_RST >= 0) {
    digitalWrite(TFT_RST, HIGH);
    delay(5);
    digitalWrite(TFT_RST, LOW);
    delay(20);
    digitalWrite(TFT_RST, HIGH);
  }
  else writecommand(TFT_SWRST); // Software reset
#else
  writecommand(TFT_SWRST); // Software reset
```

```
  delay(150); // Wait for reset to complete

  begin_tft_write();

  tc = tc; // Supress warning

  // This loads the driver specific initialisation code  <<<<<<<<<<<<<<<<<<<< ADD NEW DRIVERS TO THE L
#if   defined (ILI9341_DRIVER) || defined(ILI9341_2_DRIVER)
    #include "TFT_Drivers/ILI9341_Init.h"

#elif defined (ST7735_DRIVER)
    tabcolor = tc;
    #include "TFT_Drivers/ST7735_Init.h"

#elif defined (ILI9163_DRIVER)
    #include "TFT_Drivers/ILI9163_Init.h"

#elif defined (S6D02A1_DRIVER)
    #include "TFT_Drivers/S6D02A1_Init.h"

#elif defined (ST7796_DRIVER)
    #include "TFT_Drivers/ST7796_Init.h"

#elif defined (ILI9486_DRIVER)
    #include "TFT_Drivers/ILI9486_Init.h"

#elif defined (ILI9481_DRIVER)
    #include "TFT_Drivers/ILI9481_Init.h"

#elif defined (ILI9488_DRIVER)
    #include "TFT_Drivers/ILI9488_Init.h"

#elif defined (HX8357D_DRIVER)
    #include "TFT_Drivers/HX8357D_Init.h"

#elif defined (ST7789_DRIVER)
    #include "TFT_Drivers/ST7789_Init.h"

#elif defined (R61581_DRIVER)
    #include "TFT_Drivers/R61581_Init.h"

#elif defined (RM68140_DRIVER)
□#include "TFT_Drivers/RM68140_Init.h"

#elif defined (ST7789_2_DRIVER)
    #include "TFT_Drivers/ST7789_2_Init.h"

#elif defined (SSD1351_DRIVER)
    #include "TFT_Drivers/SSD1351_Init.h"

#elif defined (SSD1963_DRIVER)
    #include "TFT_Drivers/SSD1963_Init.h"

#elif defined (GC9A01_DRIVER)
     #include "TFT_Drivers/GC9A01_Init.h"

#elif defined (ILI9225_DRIVER)
     #include "TFT_Drivers/ILI9225_Init.h"

#endif

#ifdef TFT_INVERSION_ON
  writecommand(TFT_INVON);
#endif

#ifdef TFT_INVERSION_OFF
  writecommand(TFT_INVOFF);
#endif

  end_tft_write();

  setRotation(rotation);

#if defined (TFT_BL) && defined (TFT_BACKLIGHT_ON)
  pinMode(TFT_BL, OUTPUT);
  digitalWrite(TFT_BL, TFT_BACKLIGHT_ON);
#else
  #if defined (TFT_BL) && defined (M5STACK)
    // Turn on the back-light LED
    pinMode(TFT_BL, OUTPUT);
    digitalWrite(TFT_BL, HIGH);
  #endif
#endif
```

```
/***************************************************************************************
** Function name:           setRotation
** Description:             rotate the screen orientation m = 0-3 or 4-7 for BMP drawing
***************************************************************************************/
void TFT_eSPI::setRotation(uint8_t m)
{

  begin_tft_write();

    // This loads the driver specific rotation code  <<<<<<<<<<<<<<<<<<<< ADD NEW DRIVERS TO THE LIST
#if   defined (ILI9341_DRIVER) || defined(ILI9341_2_DRIVER)
    #include "TFT_Drivers/ILI9341_Rotation.h"

#elif defined (ST7735_DRIVER)
    #include "TFT_Drivers/ST7735_Rotation.h"

#elif defined (ILI9163_DRIVER)
    #include "TFT_Drivers/ILI9163_Rotation.h"

#elif defined (S6D02A1_DRIVER)
    #include "TFT_Drivers/S6D02A1_Rotation.h"

#elif defined (ST7796_DRIVER)
    #include "TFT_Drivers/ST7796_Rotation.h"

#elif defined (ILI9486_DRIVER)
    #include "TFT_Drivers/ILI9486_Rotation.h"

#elif defined (ILI9481_DRIVER)
    #include "TFT_Drivers/ILI9481_Rotation.h"

#elif defined (ILI9488_DRIVER)
    #include "TFT_Drivers/ILI9488_Rotation.h"

#elif defined (HX8357D_DRIVER)
    #include "TFT_Drivers/HX8357D_Rotation.h"

#elif defined (ST7789_DRIVER)
    #include "TFT_Drivers/ST7789_Rotation.h"

#elif defined (R61581_DRIVER)
    #include "TFT_Drivers/R61581_Rotation.h"

#elif defined (RM68140_DRIVER)
□#include "TFT_Drivers/RM68140_Rotation.h"

#elif defined (ST7789_2_DRIVER)
    #include "TFT_Drivers/ST7789_2_Rotation.h"

#elif defined (SSD1351_DRIVER)
    #include "TFT_Drivers/SSD1351_Rotation.h"

#elif defined (SSD1963_DRIVER)
    #include "TFT_Drivers/SSD1963_Rotation.h"

#elif defined (GC9A01_DRIVER)
     #include "TFT_Drivers/GC9A01_Rotation.h"

#elif defined (ILI9225_DRIVER)
     #include "TFT_Drivers/ILI9225_Rotation.h"

#endif

  delayMicroseconds(10);

  end_tft_write();

  addr_row = 0xFFFF;
  addr_col = 0xFFFF;

  // Reset the viewport to the whole screen
  resetViewport();
}
```

```
/***************************************************************************************
** Function name:           commandList, used for FLASH based lists only (e.g. ST7735)
** Description:             Get initialisation commands from FLASH and send to TFT
***************************************************************************************/
void TFT_eSPI::commandList (const uint8_t *addr)
{
  uint8_t  numCommands;
  uint8_t  numArgs;
  uint8_t  ms;

  numCommands = pgm_read_byte(addr++);   // Number of commands to follow

  while (numCommands--)                 // For each command...
  {
    writecommand(pgm_read_byte(addr++)); // Read, issue command
    numArgs = pgm_read_byte(addr++);     // Number of args to follow
    ms = numArgs & TFT_INIT_DELAY;       // If hibit set, delay follows args
    numArgs &= ~TFT_INIT_DELAY;          // Mask out delay bit

    while (numArgs--)                   // For each argument...
    {
      writedata(pgm_read_byte(addr++));  // Read, issue argument
    }

    if (ms)
    {
      ms = pgm_read_byte(addr++);       // Read post-command delay time (ms)
      delay( (ms==255 ? 500 : ms) );
    }
  }

}
```

```
/***************************************************************************************
** Function name:           spiwrite
** Description:             Write 8 bits to SPI port (legacy support only)
***************************************************************************************/
void TFT_eSPI::spiwrite(uint8_t c)
{
  begin_tft_write();
  tft_Write_8(c);
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           writecommand
** Description:             Send an 8 bit command to the TFT
***************************************************************************************/
void TFT_eSPI::writecommand(uint8_t c)
{
  begin_tft_write();

  DC_C;

  tft_Write_8(c);

  DC_D;

  end_tft_write();

}
```

```
/**************************************************************************************
** Function name:           writedata
** Description:             Send a 8 bit data value to the TFT
**************************************************************************************/
void TFT_eSPI::writedata(uint8_t d)
{
  begin_tft_write();

  DC_D;        // Play safe, but should already be in data mode

  tft_Write_8(d);

  CS_L;        // Allow more hold time for low VDI rail

  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           readcommand8
** Description:             Read a 8 bit data value from an indexed command register
***************************************************************************************/
uint8_t TFT_eSPI::readcommand8(uint8_t cmd_function, uint8_t index)
{
  uint8_t reg = 0;
#ifdef TFT_PARALLEL_8_BIT

  writecommand(cmd_function); // Sets DC and CS high

  busDir(dir_mask, INPUT);

  CS_L;

  // Read nth parameter (assumes caller discards 1st parameter or points index to 2nd)
  while(index--) reg = readByte();

  busDir(dir_mask, OUTPUT);

  CS_H;

#else // SPI interface
  // Tested with ILI9341 set to Interface II i.e. IM [3:0] = "1101"
  begin_tft_read();
  index = 0x10 + (index & 0x0F);

  DC_C; tft_Write_8(0xD9);
  DC_D; tft_Write_8(index);

  CS_H; // Some displays seem to need CS to be pulsed here, or is just a delay needed?
  CS_L;

  DC_C; tft_Write_8(cmd_function);
  DC_D;
  reg = tft_Read_8();

  end_tft_read();
#endif
  return reg;
}
```

```
/***************************************************************************************
** Function name:           readcommand16
** Description:             Read a 16 bit data value from an indexed command register
***************************************************************************************/
uint16_t TFT_eSPI::readcommand16(uint8_t cmd_function, uint8_t index)
{
  uint32_t reg;

  reg  = (readcommand8(cmd_function, index + 0) <<  8);
  reg |= (readcommand8(cmd_function, index + 1) <<  0);

  return reg;
}
```

```
/***************************************************************************************
** Function name:           readcommand32
** Description:             Read a 32 bit data value from an indexed command register
***************************************************************************************/
uint32_t TFT_eSPI::readcommand32(uint8_t cmd_function, uint8_t index)
{
  uint32_t reg;

  reg  = ((uint32_t)readcommand8(cmd_function, index + 0) << 24);
  reg |= ((uint32_t)readcommand8(cmd_function, index + 1) << 16);
  reg |= ((uint32_t)readcommand8(cmd_function, index + 2) <<  8);
  reg |= ((uint32_t)readcommand8(cmd_function, index + 3) <<  0);

  return reg;
}
```

```
/***************************************************************************************
** Function name:           read pixel (for SPI Interface II i.e. IM [3:0] = "1101")
** Description:             Read 565 pixel colours from a pixel
***************************************************************************************/
uint16_t TFT_eSPI::readPixel(int32_t x0, int32_t y0)
{
  if (_vpOoB) return 0;

  x0+= _xDatum;
  y0+= _yDatum;

  // Range checking
  if ((x0 < _vpX) || (y0 < _vpY) ||(x0 >= _vpW) || (y0 >= _vpH)) return 0;

#if defined(TFT_PARALLEL_8_BIT)

  CS_L;

  readAddrWindow(x0, y0, 1, 1);

  // Set masked pins D0- D7 to input
  busDir(dir_mask, INPUT);

#if  !defined (SSD1963_DRIVER)
  // Dummy read to throw away don't care value
  readByte();
#endif

  // Fetch the 16 bit BRG pixel
  //uint16_t rgb = (readByte() << 8) | readByte();

  #if defined (ILI9341_DRIVER)  || defined(ILI9341_2_DRIVER) || defined (ILI9488_DRIVER) || defined (SSD1963_

    // Read window pixel 24 bit RGB values and fill in LS bits
    uint16_t rgb = ((readByte() & 0xF8) << 8) | ((readByte() & 0xFC) << 3) | (readByte() >> 3);

    CS_H;

    // Set masked pins D0- D7 to output
    busDir(dir_mask, OUTPUT);

    return rgb;

  #else // ILI9481 or ILI9486 16 bit read

    // Fetch the 16 bit BRG pixel
    uint16_t bgr = (readByte() << 8) | readByte();

    CS_H;

    // Set masked pins D0- D7 to output
    busDir(dir_mask, OUTPUT);

    #ifdef ILI9486_DRIVER
      return  bgr;
    #else
      // Swap Red and Blue (could check MADCTL setting to see if this is needed)
      return  (bgr>>11) | (bgr<<11) | (bgr & 0x7E0);
    #endif

  #endif

#else // Not TFT_PARALLEL_8_BIT

  // This function can get called during antialiased font rendering
  // so a transaction may be in progress
  bool wasInTransaction = inTransaction;
  if (inTransaction) { inTransaction= false; end_tft_write();}

  uint16_t color = 0;

  begin_tft_read();

  readAddrWindow(x0, y0, 1, 1); // Sets CS low

  #ifdef TFT_SDA_READ
    begin_SDA_Read();
  #endif

  // Dummy read to throw away don't care value
  tft_Read_8();

  //#if !defined (ILI9488_DRIVER)
```

```
      // Read the 2 bytes
      color = ((tft_Read_8()) << 8) | (tft_Read_8());
    #else
      // Read the 3 RGB bytes, colour is actually only in the top 6 bits of each byte
      // as the TFT stores colours as 18 bits
      uint8_t r = tft_Read_8();
      uint8_t g = tft_Read_8();
      uint8_t b = tft_Read_8();
      color = color565(r, g, b);
    #endif

/*
  #else

    // The 6 colour bits are in MS 6 bits of each byte, but the ILI9488 needs an extra clock pulse
    // so bits appear shifted right 1 bit, so mask the middle 6 bits then shift 1 place left
    uint8_t r = (tft_Read_8()&0x7E)<<1;
    uint8_t g = (tft_Read_8()&0x7E)<<1;
    uint8_t b = (tft_Read_8()&0x7E)<<1;
    color = color565(r, g, b);

  #endif
*/
  CS_H;

  #ifdef TFT_SDA_READ
    end_SDA_Read();
  #endif

  end_tft_read();

  // Reinstate the transaction if one was in progress
  if(wasInTransaction) { begin_tft_write(); inTransaction = true; }

  return color;

#endif
}

void TFT_eSPI::setCallback(getColorCallback getCol)
{
  getColor = getCol;
}
```

```
/***************************************************************************************
** Function name:           read rectangle (for SPI Interface II i.e. IM [3:0] = "1101")
** Description:             Read 565 pixel colours from a defined area
***************************************************************************************/
void TFT_eSPI::readRect(int32_t x, int32_t y, int32_t w, int32_t h, uint16_t *data)
{
  PI_CLIP ;

#if defined(TFT_PARALLEL_8_BIT)

  CS_L;

  readAddrWindow(x, y, dw, dh);

  data += dx + dy * w;

  // Set masked pins D0- D7 to input
  busDir(dir_mask, INPUT);

  #if defined (ILI9341_DRIVER)  || defined(ILI9341_2_DRIVER) || defined (ILI9488_DRIVER) // Read 3 bytes
    // Dummy read to throw away don't care value
    readByte();

    // Fetch the 24 bit RGB value
    while (dh--) {
      int32_t lw = dw;
      uint16_t* line = data;
      while (lw--) {
        // Assemble the RGB 16 bit colour
        uint16_t rgb = ((readByte() & 0xF8) << 8) | ((readByte() & 0xFC) << 3) | (readByte() >> 3);

        // Swapped byte order for compatibility with pushRect()
        *line++ = (rgb<<8) | (rgb>>8);
      }
      data += w;
    }

  #elif  defined (SSD1963_DRIVER)
    // Fetch the 18 bit BRG pixels
    while (dh--) {
      int32_t lw = dw;
      uint16_t* line = data;
      while (lw--) {
        uint16_t bgr = ((readByte() & 0xF8) >> 3);; // CS_L adds a small delay
        bgr |= ((readByte() & 0xFC) << 3);
        bgr |= (readByte() << 8);
        // Swap Red and Blue (could check MADCTL setting to see if this is needed)
        uint16_t rgb = (bgr>>11) | (bgr<<11) | (bgr & 0x7E0);
        // Swapped byte order for compatibility with pushRect()
        *line++ = (rgb<<8) | (rgb>>8);
      }
      data += w;
    }

  #else // ILI9481 reads as 16 bits
    // Dummy read to throw away don't care value
    readByte();

    // Fetch the 16 bit BRG pixels
    while (dh--) {
      int32_t lw = dw;
      uint16_t* line = data;
      while (lw--) {
      #ifdef ILI9486_DRIVER
        // Read the RGB 16 bit colour
        *line++ = readByte() | (readByte() << 8);
      #else
        // Read the BRG 16 bit colour
        uint16_t bgr = (readByte() << 8) | readByte();
        // Swap Red and Blue (could check MADCTL setting to see if this is needed)
        uint16_t rgb = (bgr>>11) | (bgr<<11) | (bgr & 0x7E0);
        // Swapped byte order for compatibility with pushRect()
        *line++ = (rgb<<8) | (rgb>>8);
      #endif
      }
      data += w;
    }
  #endif

  CS_H;

  // Set masked pins D0- D7 to output
  busDir(dir_mask, OUTPUT);
```

```
  // This function can get called after a begin_tft_write
  // so a transaction may be in progress
  bool wasInTransaction = inTransaction;
  if (inTransaction) { inTransaction= false; end_tft_write();}

  uint16_t color = 0;

  begin_tft_read();

  readAddrWindow(x, y, dw, dh);

  data += dx + dy * w;

  #ifdef TFT_SDA_READ
    begin_SDA_Read();
  #endif

  // Dummy read to throw away don't care value
  tft_Read_8();

  // Read window pixel 24 bit RGB values
  while (dh--) {
    int32_t lw = dw;
    uint16_t* line = data;
    while (lw--) {

  #if !defined (ILI9488_DRIVER)

    #if defined (ST7796_DRIVER)
      // Read the 2 bytes
      color = ((tft_Read_8()) << 8) | (tft_Read_8());
    #else
      // Read the 3 RGB bytes, colour is actually only in the top 6 bits of each byte
      // as the TFT stores colours as 18 bits
      uint8_t r = tft_Read_8();
      uint8_t g = tft_Read_8();
      uint8_t b = tft_Read_8();
      color = color565(r, g, b);
    #endif

  #else

      // The 6 colour bits are in MS 6 bits of each byte but we do not include the extra clock pulse
      // so we use a trick and mask the middle 6 bits of the byte, then only shift 1 place left
      uint8_t r = (tft_Read_8()&0x7E)<<1;
      uint8_t g = (tft_Read_8()&0x7E)<<1;
      uint8_t b = (tft_Read_8()&0x7E)<<1;
      color = color565(r, g, b);
  #endif

      // Swapped colour byte order for compatibility with pushRect()
      *line++ = color << 8 | color >> 8;
    }
    data += w;
  }

  //CS_H;

  #ifdef TFT_SDA_READ
    end_SDA_Read();
  #endif

  end_tft_read();

  // Reinstate the transaction if one was in progress
  if(wasInTransaction) { begin_tft_write(); inTransaction = true; }
#endif
}
```

```
/***************************************************************************************
** Function name:           push rectangle
** Description:             push 565 pixel colours into a defined area
***************************************************************************************/
void TFT_eSPI::pushRect(int32_t x, int32_t y, int32_t w, int32_t h, uint16_t *data)
{
  bool swap = _swapBytes; _swapBytes = false;
  pushImage(x, y, w, h, data);
  _swapBytes = swap;
}
```

```
/***************************************************************************************
** Function name:           pushImage
** Description:             plot 16 bit colour sprite or image onto TFT
***************************************************************************************/
void TFT_eSPI::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, uint16_t *data)
{
  PI_CLIP;

  begin_tft_write();
  inTransaction = true;

  setWindow(x, y, x + dw - 1, y + dh - 1);

  data += dx + dy * w;

  // Check if whole image can be pushed
  if (dw == w) pushPixels(data, dw * dh);
  else {
    // Push line segments to crop image
    while (dh--)
    {
      pushPixels(data, dw);
      data += w;
    }
  }

  inTransaction = lockTransaction;
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           pushImage
** Description:             plot 16 bit sprite or image with 1 colour being transparent
***************************************************************************************/
void TFT_eSPI::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, uint16_t *data, uint16_t transp)
{
  PI_CLIP;

  begin_tft_write();
  inTransaction = true;

  data += dx + dy * w;

  int32_t xe = x + dw - 1, ye = y + dh - 1;

  uint16_t  lineBuf[dw]; // Use buffer to minimise setWindow call count

  // The little endian transp color must be byte swapped if the image is big endian
  if (!_swapBytes) transp = transp >> 8 | transp << 8;

  while (dh--)
  {
    int32_t len = dw;
    uint16_t* ptr = data;
    int32_t px = x;
    bool move = true;
    uint16_t np = 0;

    while (len--)
    {
      if (transp != *ptr)
      {
        if (move) { move = false; setWindow(px, y, xe, ye); }
        lineBuf[np] = *ptr;
        np++;
      }
      else
      {
        move = true;
        if (np)
        {
          pushPixels((uint16_t*)lineBuf, np);
          np = 0;
        }
      }
      px++;
      ptr++;
    }
    if (np) pushPixels((uint16_t*)lineBuf, np);

    y++;
    data += w;
  }

  inTransaction = lockTransaction;
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           pushImage - for FLASH (PROGMEM) stored images
** Description:             plot 16 bit image
***************************************************************************************/
void TFT_eSPI::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, const uint16_t *data)
{
  // Requires 32 bit aligned access, so use PROGMEM 16 bit word functions
  PI_CLIP;

  begin_tft_write();
  inTransaction = true;

  data += dx + dy * w;

  uint16_t  buffer[dw];

  setWindow(x, y, x + dw - 1, y + dh - 1);

  // Fill and send line buffers to TFT
  for (int32_t i = 0; i < dh; i++) {
    for (int32_t j = 0; j < dw; j++) {
      buffer[j] = pgm_read_word(&data[i * w + j]);
    }
    pushPixels(buffer, dw);
  }

  inTransaction = lockTransaction;
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           pushImage - for FLASH (PROGMEM) stored images
** Description:             plot 16 bit image with 1 colour being transparent
***************************************************************************************/
void TFT_eSPI::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, const uint16_t *data, uint16_t transp)
{
  // Requires 32 bit aligned access, so use PROGMEM 16 bit word functions
  PI_CLIP;

  begin_tft_write();
  inTransaction = true;

  data += dx + dy * w;

  int32_t xe = x + dw - 1, ye = y + dh - 1;

  uint16_t  lineBuf[dw];

  // The little endian transp color must be byte swapped if the image is big endian
  if (!_swapBytes) transp = transp >> 8 | transp << 8;

  while (dh--) {
    int32_t len = dw;
    uint16_t* ptr = (uint16_t*)data;
    int32_t px = x;
    bool move = true;

    uint16_t np = 0;

    while (len--) {
      uint16_t color = pgm_read_word(ptr);
      if (transp != color) {
        if (move) { move = false; setWindow(px, y, xe, ye); }
        lineBuf[np] = color;
        np++;
      }
      else {
        move = true;
        if (np) {
          pushPixels(lineBuf, np);
          np = 0;
        }
      }
      px++;
      ptr++;
    }
    if (np) pushPixels(lineBuf, np);

    y++;
    data += w;
  }

  inTransaction = lockTransaction;
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           pushImage
** Description:             plot 8 bit or 4 bit or 1 bit image or sprite using a line buffer
***************************************************************************************/
void TFT_eSPI::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, const uint8_t *data, bool bpp8,  uint16_t *cm
{
  PI_CLIP;

  begin_tft_write();
  inTransaction = true;
  bool swap = _swapBytes;

  setWindow(x, y, x + dw - 1, y + dh - 1); // Sets CS low and sent RAMWR

  // Line buffer makes plotting faster
  uint16_t  lineBuf[dw];

  if (bpp8)
  {
    _swapBytes = false;

    uint8_t  blue[] = {0, 11, 21, 31}; // blue 2 to 5 bit colour lookup table

    _lastColor = -1; // Set to illegal value

    // Used to store last shifted colour
    uint8_t msbColor = 0;
    uint8_t lsbColor = 0;

    data += dx + dy * w;
    while (dh--) {
      uint32_t len = dw;
      uint8_t* ptr = (uint8_t*)data;
      uint8_t* linePtr = (uint8_t*)lineBuf;

      while(len--) {
        uint32_t color = pgm_read_byte(ptr++);

        // Shifts are slow so check if colour has changed first
        if (color != _lastColor) {
          //         =====Green=====     ===============Red==============
          msbColor = (color & 0x1C)>>2 | (color & 0xC0)>>3 | (color & 0xE0);
          //         =====Green=====    =======Blue======
          lsbColor = (color & 0x1C)<<3 | blue[color & 0x03];
          _lastColor = color;
        }

       *linePtr++ = msbColor;
       *linePtr++ = lsbColor;
      }

      pushPixels(lineBuf, dw);

      data += w;
    }
    _swapBytes = swap; // Restore old value
  }
  else if (cmap != nullptr) // Must be 4bpp
  {
    _swapBytes = true;

    w = (w+1) & 0xFFFE;   // if this is a sprite, w will already be even; this does no harm.
    bool splitFirst = (dx & 0x01) != 0; // split first means we have to push a single px from the left of the sprite / ima

    if (splitFirst) {
      data += ((dx - 1 + dy * w) >> 1);
    }
    else {
      data += ((dx + dy * w) >> 1);
    }

    while (dh--) {
      uint32_t len = dw;
      uint8_t * ptr = (uint8_t*)data;
      uint16_t *linePtr = lineBuf;
      uint8_t colors; // two colors in one byte
      uint16_t index;

      if (splitFirst) {
        colors = pgm_read_byte(ptr);
        index = (colors & 0x0F);
        *linePtr++ = cmap[index];
        len--;
        ptr++;
```

```
    while (len--)
    {
      colors = pgm_read_byte(ptr);
      index = ((colors & 0xF0) >> 4) & 0x0F;
      *linePtr++ = cmap[index];

      if (len--)
      {
        index = colors & 0x0F;
        *linePtr++ = cmap[index];
      } else {
        break;  // nothing to do here
      }

      ptr++;
    }

    pushPixels(lineBuf, dw);
    data += (w >> 1);
   }
   _swapBytes = swap; // Restore old value
  }
  else // Must be 1bpp
  {
   _swapBytes = false;
   uint8_t * ptr = (uint8_t*)data;
   uint32_t ww =  (w+7)>>3; // Width of source image line in bytes
   for (int32_t yp = dy;  yp < dy + dh; yp++)
   {
    uint8_t* linePtr = (uint8_t*)lineBuf;
    for (int32_t xp = dx; xp < dx + dw; xp++)
    {
     uint16_t col = (pgm_read_byte(ptr + (xp>>3)) & (0x80 >> (xp & 0x7)) );
     if (col) {*linePtr++ = bitmap_fg>>8; *linePtr++ = (uint8_t) bitmap_fg;}
     else     {*linePtr++ = bitmap_bg>>8; *linePtr++ = (uint8_t) bitmap_bg;}
    }
    ptr += ww;
    pushPixels(lineBuf, dw);
   }
  }

  _swapBytes = swap; // Restore old value
  inTransaction = lockTransaction;
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           pushImage
** Description:             plot 8 bit or 4 bit or 1 bit image or sprite using a line buffer
***************************************************************************************/
void TFT_eSPI::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, uint8_t *data, bool bpp8,  uint16_t *cmap)
{
  PI_CLIP;

  begin_tft_write();
  inTransaction = true;
  bool swap = _swapBytes;

  setWindow(x, y, x + dw - 1, y + dh - 1); // Sets CS low and sent RAMWR

  // Line buffer makes plotting faster
  uint16_t  lineBuf[dw];

  if (bpp8)
  {
    _swapBytes = false;

    uint8_t  blue[] = {0, 11, 21, 31}; // blue 2 to 5 bit colour lookup table

    _lastColor = -1; // Set to illegal value

    // Used to store last shifted colour
    uint8_t msbColor = 0;
    uint8_t lsbColor = 0;

    data += dx + dy * w;
    while (dh--) {
      uint32_t len = dw;
      uint8_t* ptr = data;
      uint8_t* linePtr = (uint8_t*)lineBuf;

      while(len--) {
        uint32_t color = *ptr++;

        // Shifts are slow so check if colour has changed first
        if (color != _lastColor) {
          //         =====Green=====     ===============Red==============
          msbColor = (color & 0x1C)>>2 | (color & 0xC0)>>3 | (color & 0xE0);
          //         =====Green=====    =======Blue======
          lsbColor = (color & 0x1C)<<3 | blue[color & 0x03];
          _lastColor = color;
        }

       *linePtr++ = msbColor;
       *linePtr++ = lsbColor;
      }

      pushPixels(lineBuf, dw);

      data += w;
    }
    _swapBytes = swap; // Restore old value
  }
  else if (cmap != nullptr) // Must be 4bpp
  {
    _swapBytes = true;

    w = (w+1) & 0xFFFE;   // if this is a sprite, w will already be even; this does no harm.
    bool splitFirst = (dx & 0x01) != 0; // split first means we have to push a single px from the left of the sprite / ima

    if (splitFirst) {
      data += ((dx - 1 + dy * w) >> 1);
    }
    else {
      data += ((dx + dy * w) >> 1);
    }

    while (dh--) {
      uint32_t len = dw;
      uint8_t * ptr = data;
      uint16_t *linePtr = lineBuf;
      uint8_t colors; // two colors in one byte
      uint16_t index;

      if (splitFirst) {
        colors = *ptr;
        index = (colors & 0x0F);
        *linePtr++ = cmap[index];
        len--;
        ptr++;
```

```
    while (len--)
    {
      colors = *ptr;
      index = ((colors & 0xF0) >> 4) & 0x0F;
      *linePtr++ = cmap[index];

      if (len--)
      {
        index = colors & 0x0F;
        *linePtr++ = cmap[index];
      } else {
        break;  // nothing to do here
      }

      ptr++;
    }

    pushPixels(lineBuf, dw);
    data += (w >> 1);
   }
   _swapBytes = swap; // Restore old value
  }
  else // Must be 1bpp
  {
   _swapBytes = false;

   uint32_t ww =  (w+7)>>3; // Width of source image line in bytes
   for (int32_t yp = dy;  yp < dy + dh; yp++)
   {
    uint8_t* linePtr = (uint8_t*)lineBuf;
    for (int32_t xp = dx; xp < dx + dw; xp++)
    {
     uint16_t col = (data[(xp>>3)] & (0x80 >> (xp & 0x7)) );
     if (col) {*linePtr++ = bitmap_fg>>8; *linePtr++ = (uint8_t) bitmap_fg;}
     else     {*linePtr++ = bitmap_bg>>8; *linePtr++ = (uint8_t) bitmap_bg;}
    }
    data += ww;
    pushPixels(lineBuf, dw);
   }
  }

  _swapBytes = swap; // Restore old value
  inTransaction = lockTransaction;
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           pushImage
** Description:             plot 8 or 4 or 1 bit image or sprite with a transparent colour
***************************************************************************************/
void TFT_eSPI::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, uint8_t *data, uint8_t transp, bool bpp8, uint
{
  PI_CLIP;

  begin_tft_write();
  inTransaction = true;
  bool swap = _swapBytes;

  int32_t xe = x + dw - 1, ye = y + dh - 1;

  // Line buffer makes plotting faster
  uint16_t  lineBuf[dw];

  if (bpp8) { // 8 bits per pixel
    _swapBytes = false;

    data += dx + dy * w;

    uint8_t  blue[] = {0, 11, 21, 31}; // blue 2 to 5 bit colour lookup table

    _lastColor = -1; // Set to illegal value

    // Used to store last shifted colour
    uint8_t msbColor = 0;
    uint8_t lsbColor = 0;

    while (dh--) {
      int32_t len = dw;
      uint8_t* ptr = data;
      uint8_t* linePtr = (uint8_t*)lineBuf;

      int32_t px = x;
      bool move = true;
      uint16_t np = 0;

      while (len--) {
        if (transp != *ptr) {
          if (move) { move = false; setWindow(px, y, xe, ye);}
          uint8_t color = *ptr;

          // Shifts are slow so check if colour has changed first
          if (color != _lastColor) {
            //         =====Green=====    ===============Red==============
            msbColor = (color & 0x1C)>>2 | (color & 0xC0)>>3 | (color & 0xE0);
            //         =====Green=====    ========Blue======
            lsbColor = (color & 0x1C)<<3 | blue[color & 0x03];
            _lastColor = color;
          }
          *linePtr++ = msbColor;
          *linePtr++ = lsbColor;
          np++;
        }
        else {
          move = true;
          if (np) {
            pushPixels(lineBuf, np);
            linePtr = (uint8_t*)lineBuf;
            np = 0;
          }
        }
        px++;
        ptr++;
      }

      if (np) pushPixels(lineBuf, np);
      y++;
      data += w;
    }
  }
  else if (cmap != nullptr) // 4bpp with color map
  {
    _swapBytes = true;

    w = (w+1) & 0xFFFE; // here we try to recreate iwidth from dwidth.
    bool splitFirst = ((dx & 0x01) != 0);
    if (splitFirst) {
      data += ((dx - 1 + dy * w) >> 1);
    }
    else {
      data += ((dx + dy * w) >> 1);
```

```
  while (dh--) {
    uint32_t len = dw;
    uint8_t * ptr = data;

    int32_t px = x;
    bool move = true;
    uint16_t np = 0;

    uint8_t index;  // index into cmap.

    if (splitFirst) {
      index = (*ptr & 0x0F);  // odd = bits 3 .. 0
      if (index != transp) {
        move = false; setWindow(px, y, xe, ye);
        lineBuf[np] = cmap[index];
        np++;
      }
      px++; ptr++;
      len--;
    }

    while (len--)
    {
      uint8_t color = *ptr;

      // find the actual color you care about.  There will be two pixels here!
      // but we may only want one at the end of the row
      uint16_t index = ((color & 0xF0) >> 4) & 0x0F;  // high bits are the even numbers
      if (index != transp) {
        if (move) {
          move = false; setWindow(px, y, xe, ye);
        }
        lineBuf[np] = cmap[index];
        np++; // added a pixel
      }
      else {
        move = true;
        if (np) {
          pushPixels(lineBuf, np);
          np = 0;
        }
      }
      px++;

      if (len--)
      {
        index = color & 0x0F; // the odd number is 3 .. 0
        if (index != transp) {
          if (move) {
            move = false; setWindow(px, y, xe, ye);
          }
          lineBuf[np] = cmap[index];
          np++;
        }
        else {
          move = true;
          if (np) {
            pushPixels(lineBuf, np);
            np = 0;
          }
        }
        px++;
      }
      else {
        break;  // we are done with this row.
      }
      ptr++;  // we only increment ptr once in the loop (deliberate)
    }

    if (np) {
      pushPixels(lineBuf, np);
      np = 0;
    }
    data += (w>>1);
    y++;
  }
}
else { // 1 bit per pixel
  _swapBytes = false;

  uint32_t ww =  (w+7)>>3; // Width of source image line in bytes
  uint16_t np = 0;
```

```
    {
      int32_t px = x;
      bool move = true;
      for (int32_t xp = dx; xp < dx + dw; xp++)
      {
        if (data[(xp>>3)] & (0x80 >> (xp & 0x7))) {
          if (move) {
            move = false;
            setWindow(px, y, xe, ye);
          }
          np++;
        }
        else {
          if (np) {
            pushBlock(bitmap_fg, np);
            np = 0;
            move = true;
          }
        }
        px++;
      }
      y++;
      data += ww;
      if (np) { pushBlock(bitmap_fg, np); np = 0; }
    }
  }
  _swapBytes = swap; // Restore old value
  inTransaction = lockTransaction;
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           setSwapBytes
** Description:             Used by 16 bit pushImage() to swap byte order in colours
***************************************************************************************/
void TFT_eSPI::setSwapBytes(bool swap)
{
  _swapBytes = swap;
}
```

```
/***************************************************************************************
** Function name:           getSwapBytes
** Description:             Return the swap byte order for colours
***************************************************************************************/
bool TFT_eSPI::getSwapBytes(void)
{
  return _swapBytes;
}
```

```
/***************************************************************************************
** Function name:           read rectangle (for SPI Interface II i.e. IM [3:0] = "1101")
** Description:             Read RGB pixel colours from a defined area
***************************************************************************************/
// If w and h are 1, then 1 pixel is read, *data array size must be 3 bytes per pixel
void  TFT_eSPI::readRectRGB(int32_t x0, int32_t y0, int32_t w, int32_t h, uint8_t *data)
{
#if defined(TFT_PARALLEL_8_BIT)

  uint32_t len = w * h;
  uint8_t* buf565 = data + len;

  readRect(x0, y0, w, h, (uint16_t*)buf565);

  while (len--) {
    uint16_t pixel565 = (*buf565++)<<8;
    pixel565 |= *buf565++;
    uint8_t red   = (pixel565 & 0xF800) >> 8; red   |= red   >> 5;
    uint8_t green = (pixel565 & 0x07E0) >> 3; green |= green >> 6;
    uint8_t blue  = (pixel565 & 0x001F) << 3; blue  |= blue  >> 5;
    *data++ = red;
    *data++ = green;
    *data++ = blue;
  }

#else  // Not TFT_PARALLEL_8_BIT

  begin_tft_read();

  readAddrWindow(x0, y0, w, h); // Sets CS low

  #ifdef TFT_SDA_READ
    begin_SDA_Read();
  #endif

  // Dummy read to throw away don't care value
  tft_Read_8();

  // Read window pixel 24 bit RGB values, buffer must be set in sketch to 3 * w * h
  uint32_t len = w * h;
  while (len--) {

  #if !defined (ILI9488_DRIVER)

    // Read the 3 RGB bytes, colour is actually only in the top 6 bits of each byte
    // as the TFT stores colours as 18 bits
    *data++ = tft_Read_8();
    *data++ = tft_Read_8();
    *data++ = tft_Read_8();

  #else

    // The 6 colour bits are in MS 6 bits of each byte, but the ILI9488 needs an extra clock pulse
    // so bits appear shifted right 1 bit, so mask the middle 6 bits then shift 1 place left
    *data++ = (tft_Read_8()&0x7E)<<1;
    *data++ = (tft_Read_8()&0x7E)<<1;
    *data++ = (tft_Read_8()&0x7E)<<1;

  #endif

  }

  CS_H;

  #ifdef TFT_SDA_READ
    end_SDA_Read();
  #endif

  end_tft_read();

#endif
}
```

```
/***************************************************************************************
** Function name:           drawCircle
** Description:             Draw a circle outline
***************************************************************************************/
// Optimised midpoint circle algorithm
void TFT_eSPI::drawCircle(int32_t x0, int32_t y0, int32_t r, uint32_t color)
{
  if ( r <= 0 ) return;

  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

    int32_t f     = 1 - r;
    int32_t ddF_y = -2 * r;
    int32_t ddF_x = 1;
    int32_t xs    = -1;
    int32_t xe    = 0;
    int32_t len   = 0;

    bool first = true;
    do {
      while (f < 0) {
        ++xe;
        f += (ddF_x += 2);
      }
      f += (ddF_y += 2);

      if (xe-xs>1) {
        if (first) {
          len = 2*(xe - xs)-1;
          drawFastHLine(x0 - xe, y0 + r, len, color);
          drawFastHLine(x0 - xe, y0 - r, len, color);
          drawFastVLine(x0 + r, y0 - xe, len, color);
          drawFastVLine(x0 - r, y0 - xe, len, color);
          first = false;
        }
        else {
          len = xe - xs++;
          drawFastHLine(x0 - xe, y0 + r, len, color);
          drawFastHLine(x0 - xe, y0 - r, len, color);
          drawFastHLine(x0 + xs, y0 - r, len, color);
          drawFastHLine(x0 + xs, y0 + r, len, color);

          drawFastVLine(x0 + r, y0 + xs, len, color);
          drawFastVLine(x0 + r, y0 - xe, len, color);
          drawFastVLine(x0 - r, y0 - xe, len, color);
          drawFastVLine(x0 - r, y0 + xs, len, color);
        }
      }
      else {
        ++xs;
        drawPixel(x0 - xe, y0 + r, color);
        drawPixel(x0 - xe, y0 - r, color);
        drawPixel(x0 + xs, y0 - r, color);
        drawPixel(x0 + xs, y0 + r, color);

        drawPixel(x0 + r, y0 + xs, color);
        drawPixel(x0 + r, y0 - xe, color);
        drawPixel(x0 - r, y0 - xe, color);
        drawPixel(x0 - r, y0 + xs, color);
      }
      xs = xe;
    } while (xe < --r);

  inTransaction = lockTransaction;
  end_tft_write();             // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           drawCircleHelper
** Description:             Support function for drawRoundRect()
***************************************************************************************/
void TFT_eSPI::drawCircleHelper( int32_t x0, int32_t y0, int32_t rr, uint8_t cornername, uint32_t color)
{
  if (rr <= 0) return;
  int32_t f     = 1 - rr;
  int32_t ddF_x = 1;
  int32_t ddF_y = -2 * rr;
  int32_t xe    = 0;
  int32_t xs    = 0;
  int32_t len   = 0;

  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  while (xe < rr--)
  {
    while (f < 0) {
      ++xe;
      f += (ddF_x += 2);
    }
    f += (ddF_y += 2);

    if (xe-xs==1) {
      if (cornername & 0x1) { // left top
        drawPixel(x0 - xe, y0 - rr, color);
        drawPixel(x0 - rr, y0 - xe, color);
      }
      if (cornername & 0x2) { // right top
        drawPixel(x0 + rr    , y0 - xe, color);
        drawPixel(x0 + xs + 1, y0 - rr, color);
      }
      if (cornername & 0x4) { // right bottom
        drawPixel(x0 + xs + 1, y0 + rr    , color);
        drawPixel(x0 + rr, y0 + xs + 1, color);
      }
      if (cornername & 0x8) { // left bottom
        drawPixel(x0 - rr, y0 + xs + 1, color);
        drawPixel(x0 - xe, y0 + rr    , color);
      }
    }
    else {
      len = xe - xs++;
      if (cornername & 0x1) { // left top
        drawFastHLine(x0 - xe, y0 - rr, len, color);
        drawFastVLine(x0 - rr, y0 - xe, len, color);
      }
      if (cornername & 0x2) { // right top
        drawFastVLine(x0 + rr, y0 - xe, len, color);
        drawFastHLine(x0 + xs, y0 - rr, len, color);
      }
      if (cornername & 0x4) { // right bottom
        drawFastHLine(x0 + xs, y0 + rr, len, color);
        drawFastVLine(x0 + rr, y0 + xs, len, color);
      }
      if (cornername & 0x8) { // left bottom
        drawFastVLine(x0 - rr, y0 + xs, len, color);
        drawFastHLine(x0 - xe, y0 + rr, len, color);
      }
    }
    xs = xe;
  }
  inTransaction = lockTransaction;
  end_tft_write();           // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           fillCircle
** Description:             draw a filled circle
***************************************************************************************/
// Optimised midpoint circle algorithm, changed to horizontal lines (faster in sprites)
// Improved algorithm avoids repetition of lines
void TFT_eSPI::fillCircle(int32_t x0, int32_t y0, int32_t r, uint32_t color)
{
  int32_t  x  = 0;
  int32_t  dx = 1;
  int32_t  dy = r+r;
  int32_t  p  = -(r>>1);

  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  drawFastHLine(x0 - r, y0, dy+1, color);

  while(x<r){

    if(p>=0) {
      drawFastHLine(x0 - x, y0 + r, dx, color);
      drawFastHLine(x0 - x, y0 - r, dx, color);
      dy-=2;
      p-=dy;
      r--;
    }

    dx+=2;
    p+=dx;
    x++;

    drawFastHLine(x0 - r, y0 + x, dy+1, color);
    drawFastHLine(x0 - r, y0 - x, dy+1, color);

  }

  inTransaction = lockTransaction;
  end_tft_write();             // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           fillCircleHelper
** Description:             Support function for fillRoundRect()
***************************************************************************************/
// Support drawing roundrects, changed to horizontal lines (faster in sprites)
void TFT_eSPI::fillCircleHelper(int32_t x0, int32_t y0, int32_t r, uint8_t cornername, int32_t delta, uint32_t color)
{
  int32_t f     = 1 - r;
  int32_t ddF_x = 1;
  int32_t ddF_y = -r - r;
  int32_t y     = 0;

  delta++;

  while (y < r) {
    if (f >= 0) {
      if (cornername & 0x1) drawFastHLine(x0 - y, y0 + r, y + y + delta, color);
      if (cornername & 0x2) drawFastHLine(x0 - y, y0 - r, y + y + delta, color);
      r--;
      ddF_y += 2;
      f     += ddF_y;
    }

    y++;
    ddF_x += 2;
    f     += ddF_x;

    if (cornername & 0x1) drawFastHLine(x0 - r, y0 + y, r + r + delta, color);
    if (cornername & 0x2) drawFastHLine(x0 - r, y0 - y, r + r + delta, color);
  }
}
```

```
/***************************************************************************************
** Function name:           drawEllipse
** Description:             Draw a ellipse outline
***************************************************************************************/
void TFT_eSPI::drawEllipse(int16_t x0, int16_t y0, int32_t rx, int32_t ry, uint16_t color)
{
  if (rx<2) return;
  if (ry<2) return;
  int32_t x, y;
  int32_t rx2 = rx * rx;
  int32_t ry2 = ry * ry;
  int32_t fx2 = 4 * rx2;
  int32_t fy2 = 4 * ry2;
  int32_t s;

  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  for (x = 0, y = ry, s = 2*ry2+rx2*(1-2*ry); ry2*x <= rx2*y; x++) {
    // These are ordered to minimise coordinate changes in x or y
    // drawPixel can then send fewer bounding box commands
    drawPixel(x0 + x, y0 + y, color);
    drawPixel(x0 - x, y0 + y, color);
    drawPixel(x0 - x, y0 - y, color);
    drawPixel(x0 + x, y0 - y, color);
    if (s >= 0) {
      s += fx2 * (1 - y);
      y--;
    }
    s += ry2 * ((4 * x) + 6);
  }

  for (x = rx, y = 0, s = 2*rx2+ry2*(1-2*rx); rx2*y <= ry2*x; y++) {
    // These are ordered to minimise coordinate changes in x or y
    // drawPixel can then send fewer bounding box commands
    drawPixel(x0 + x, y0 + y, color);
    drawPixel(x0 - x, y0 + y, color);
    drawPixel(x0 - x, y0 - y, color);
    drawPixel(x0 + x, y0 - y, color);
    if (s >= 0)
    {
      s += fy2 * (1 - x);
      x--;
    }
    s += rx2 * ((4 * y) + 6);
  }

  inTransaction = lockTransaction;
  end_tft_write();             // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           fillEllipse
** Description:             draw a filled ellipse
***************************************************************************************/
void TFT_eSPI::fillEllipse(int16_t x0, int16_t y0, int32_t rx, int32_t ry, uint16_t color)
{
  if (rx<2) return;
  if (ry<2) return;
  int32_t x, y;
  int32_t rx2 = rx * rx;
  int32_t ry2 = ry * ry;
  int32_t fx2 = 4 * rx2;
  int32_t fy2 = 4 * ry2;
  int32_t s;

  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  for (x = 0, y = ry, s = 2*ry2+rx2*(1-2*ry); ry2*x <= rx2*y; x++) {
    drawFastHLine(x0 - x, y0 - y, x + x + 1, color);
    drawFastHLine(x0 - x, y0 + y, x + x + 1, color);

    if (s >= 0) {
      s += fx2 * (1 - y);
      y--;
    }
    s += ry2 * ((4 * x) + 6);
  }

  for (x = rx, y = 0, s = 2*rx2+ry2*(1-2*rx); rx2*y <= ry2*x; y++) {
    drawFastHLine(x0 - x, y0 - y, x + x + 1, color);
    drawFastHLine(x0 - x, y0 + y, x + x + 1, color);

    if (s >= 0) {
      s += fy2 * (1 - x);
      x--;
    }
    s += rx2 * ((4 * y) + 6);
  }

  inTransaction = lockTransaction;
  end_tft_write();             // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           fillScreen
** Description:             Clear the screen to defined colour
***************************************************************************************/
void TFT_eSPI::fillScreen(uint32_t color)
{
  fillRect(0, 0, _width, _height, color);
}
```

```
/***************************************************************************************
** Function name:           drawRect
** Description:             Draw a rectangle outline
***************************************************************************************/
// Draw a rectangle
void TFT_eSPI::drawRect(int32_t x, int32_t y, int32_t w, int32_t h, uint32_t color)
{
  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  drawFastHLine(x, y, w, color);
  drawFastHLine(x, y + h - 1, w, color);
  // Avoid drawing corner pixels twice
  drawFastVLine(x, y+1, h-2, color);
  drawFastVLine(x + w - 1, y+1, h-2, color);

  inTransaction = lockTransaction;
  end_tft_write();             // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           drawRoundRect
** Description:             Draw a rounded corner rectangle outline
***************************************************************************************/
// Draw a rounded rectangle
void TFT_eSPI::drawRoundRect(int32_t x, int32_t y, int32_t w, int32_t h, int32_t r, uint32_t color)
{
  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  // smarter version
  drawFastHLine(x + r  , y    , w - r - r, color); // Top
  drawFastHLine(x + r  , y + h - 1, w - r - r, color); // Bottom
  drawFastVLine(x    , y + r  , h - r - r, color); // Left
  drawFastVLine(x + w - 1, y + r  , h - r - r, color); // Right
  // draw four corners
  drawCircleHelper(x + r    , y + r    , r, 1, color);
  drawCircleHelper(x + w - r - 1, y + r    , r, 2, color);
  drawCircleHelper(x + w - r - 1, y + h - r - 1, r, 4, color);
  drawCircleHelper(x + r    , y + h - r - 1, r, 8, color);

  inTransaction = lockTransaction;
  end_tft_write();             // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           fillRoundRect
** Description:             Draw a rounded corner filled rectangle
***************************************************************************************/
// Fill a rounded rectangle, changed to horizontal lines (faster in sprites)
void TFT_eSPI::fillRoundRect(int32_t x, int32_t y, int32_t w, int32_t h, int32_t r, uint32_t color)
{
  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  // smarter version
  fillRect(x, y + r, w, h - r - r, color);

  // draw four corners
  fillCircleHelper(x + r, y + h - r - 1, r, 1, w - r - r - 1, color);
  fillCircleHelper(x + r    , y + r, r, 2, w - r - r - 1, color);

  inTransaction = lockTransaction;
  end_tft_write();             // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           drawTriangle
** Description:             Draw a triangle outline using 3 arbitrary points
***************************************************************************************/
// Draw a triangle
void TFT_eSPI::drawTriangle(int32_t x0, int32_t y0, int32_t x1, int32_t y1, int32_t x2, int32_t y2, uint32_t color)
{
  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  drawLine(x0, y0, x1, y1, color);
  drawLine(x1, y1, x2, y2, color);
  drawLine(x2, y2, x0, y0, color);

  inTransaction = lockTransaction;
  end_tft_write();             // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           fillTriangle
** Description:             Draw a filled triangle using 3 arbitrary points
***************************************************************************************/
// Fill a triangle - original Adafruit function works well and code footprint is small
void TFT_eSPI::fillTriangle ( int32_t x0, int32_t y0, int32_t x1, int32_t y1, int32_t x2, int32_t y2, uint32_t color)
{
  int32_t a, b, y, last;

  // Sort coordinates by Y order (y2 >= y1 >= y0)
  if (y0 > y1) {
    swap_coord(y0, y1); swap_coord(x0, x1);
  }
  if (y1 > y2) {
    swap_coord(y2, y1); swap_coord(x2, x1);
  }
  if (y0 > y1) {
    swap_coord(y0, y1); swap_coord(x0, x1);
  }

  if (y0 == y2) { // Handle awkward all-on-same-line case as its own thing
    a = b = x0;
    if (x1 < a)      a = x1;
    else if (x1 > b) b = x1;
    if (x2 < a)      a = x2;
    else if (x2 > b) b = x2;
    drawFastHLine(a, y0, b - a + 1, color);
    return;
  }

  //begin_tft_write();         // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  int32_t
  dx01 = x1 - x0,
  dy01 = y1 - y0,
  dx02 = x2 - x0,
  dy02 = y2 - y0,
  dx12 = x2 - x1,
  dy12 = y2 - y1,
  sa   = 0,
  sb   = 0;

  // For upper part of triangle, find scanline crossings for segments
  // 0-1 and 0-2.  If y1=y2 (flat-bottomed triangle), the scanline y1
  // is included here (and second loop will be skipped, avoiding a /0
  // error there), otherwise scanline y1 is skipped here and handled
  // in the second loop...which also avoids a /0 error here if y0=y1
  // (flat-topped triangle).
  if (y1 == y2) last = y1;  // Include y1 scanline
  else          last = y1 - 1; // Skip it

  for (y = y0; y <= last; y++) {
    a   = x0 + sa / dy01;
    b   = x0 + sb / dy02;
    sa += dx01;
    sb += dx02;

    if (a > b) swap_coord(a, b);
    drawFastHLine(a, y, b - a + 1, color);
  }

  // For lower part of triangle, find scanline crossings for segments
  // 0-2 and 1-2.  This loop is skipped if y1=y2.
  sa = dx12 * (y - y1);
  sb = dx02 * (y - y0);
  for (; y <= y2; y++) {
    a   = x1 + sa / dy12;
    b   = x0 + sb / dy02;
    sa += dx12;
    sb += dx02;

    if (a > b) swap_coord(a, b);
    drawFastHLine(a, y, b - a + 1, color);
  }

  inTransaction = lockTransaction;
  end_tft_write();             // Does nothing if Sprite class uses this function
}
```

```
/**************************************************************************************
** Function name:           drawBitmap
** Description:             Draw an image stored in an array on the TFT
**************************************************************************************/
void TFT_eSPI::drawBitmap(int16_t x, int16_t y, const uint8_t *bitmap, int16_t w, int16_t h, uint16_t color)
{
  //begin_tft_write();          // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  int32_t i, j, byteWidth = (w + 7) / 8;

  for (j = 0; j < h; j++) {
    for (i = 0; i < w; i++ ) {
      if (pgm_read_byte(bitmap + j * byteWidth + i / 8) & (128 >> (i & 7))) {
        drawPixel(x + i, y + j, color);
      }
    }
  }

  inTransaction = lockTransaction;
  end_tft_write();              // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           drawBitmap
** Description:             Draw an image stored in an array on the TFT
***************************************************************************************/
void TFT_eSPI::drawBitmap(int16_t x, int16_t y, const uint8_t *bitmap, int16_t w, int16_t h, uint16_t fgcolor, uint
{
  //begin_tft_write();          // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  int32_t i, j, byteWidth = (w + 7) / 8;

  for (j = 0; j < h; j++) {
    for (i = 0; i < w; i++ ) {
      if (pgm_read_byte(bitmap + j * byteWidth + i / 8) & (128 >> (i & 7)))
          drawPixel(x + i, y + j, fgcolor);
      else drawPixel(x + i, y + j, bgcolor);
    }
  }

  inTransaction = lockTransaction;
  end_tft_write();              // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           drawXBitmap
** Description:             Draw an image stored in an XBM array onto the TFT
***************************************************************************************/
void TFT_eSPI::drawXBitmap(int16_t x, int16_t y, const uint8_t *bitmap, int16_t w, int16_t h, uint16_t color)
{
  //begin_tft_write();          // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  int32_t i, j, byteWidth = (w + 7) / 8;

  for (j = 0; j < h; j++) {
    for (i = 0; i < w; i++ ) {
      if (pgm_read_byte(bitmap + j * byteWidth + i / 8) & (1 << (i & 7))) {
        drawPixel(x + i, y + j, color);
      }
    }
  }

  inTransaction = lockTransaction;
  end_tft_write();              // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           drawXBitmap
** Description:             Draw an XBM image with foreground and background colors
***************************************************************************************/
void TFT_eSPI::drawXBitmap(int16_t x, int16_t y, const uint8_t *bitmap, int16_t w, int16_t h, uint16_t color, uint1
{
  //begin_tft_write();          // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  int32_t i, j, byteWidth = (w + 7) / 8;

  for (j = 0; j < h; j++) {
    for (i = 0; i < w; i++ ) {
      if (pgm_read_byte(bitmap + j * byteWidth + i / 8) & (1 << (i & 7)))
          drawPixel(x + i, y + j,   color);
      else drawPixel(x + i, y + j, bgcolor);
    }
  }

  inTransaction = lockTransaction;
  end_tft_write();              // Does nothing if Sprite class uses this function
}
```

```
/***************************************************************************************
** Function name:           setCursor
** Description:             Set the text cursor x,y position
***************************************************************************************/
void TFT_eSPI::setCursor(int16_t x, int16_t y)
{
  cursor_x = x;
  cursor_y = y;
}
```

```
/***************************************************************************************
** Function name:           setCursor
** Description:             Set the text cursor x,y position and font
***************************************************************************************/
void TFT_eSPI::setCursor(int16_t x, int16_t y, uint8_t font)
{
  textfont = font;
  cursor_x = x;
  cursor_y = y;
}
```

```
/***************************************************************************************
** Function name:           getCursorX
** Description:             Get the text cursor x position
***************************************************************************************/
int16_t TFT_eSPI::getCursorX(void)
{
  return cursor_x;
}
```

```
/***************************************************************************************
** Function name:           getCursorY
** Description:             Get the text cursor y position
***************************************************************************************/
int16_t TFT_eSPI::getCursorY(void)
{
  return cursor_y;
}
```

```
/***************************************************************************************
** Function name:           setTextSize
** Description:             Set the text size multiplier
***************************************************************************************/
void TFT_eSPI::setTextSize(uint8_t s)
{
  if (s>7) s = 7; // Limit the maximum size multiplier so byte variables can be used for rendering
  textsize = (s > 0) ? s : 1; // Don't allow font size 0
}
```

```
/***************************************************************************************
** Function name:           setTextColor
** Description:             Set the font foreground colour (background is transparent)
***************************************************************************************/
void TFT_eSPI::setTextColor(uint16_t c)
{
  // For 'transparent' background, we'll set the bg
  // to the same as fg instead of using a flag
  textcolor = textbgcolor = c;
}
```

```
/***************************************************************************************
** Function name:           setTextColor
** Description:             Set the font foreground and background colour
***************************************************************************************/
void TFT_eSPI::setTextColor(uint16_t c, uint16_t b)
{
  textcolor   = c;
  textbgcolor = b;
}
```

```
/***************************************************************************************
** Function name:           setPivot
** Description:             Set the pivot point on the TFT
***************************************************************************************x*/
void TFT_eSPI::setPivot(int16_t x, int16_t y)
{
  _xPivot = x;
  _yPivot = y;
}
```

```
/***************************************************************************************
** Function name:           getPivotX
** Description:             Get the x pivot position
***************************************************************************************/
int16_t TFT_eSPI::getPivotX(void)
{
  return _xPivot;
}
```

```
/***************************************************************************************
** Function name:           getPivotY
** Description:             Get the y pivot position
***************************************************************************************/
int16_t TFT_eSPI::getPivotY(void)
{
  return _yPivot;
}
```

```
/***************************************************************************************
** Function name:           setBitmapColor
** Description:             Set the foreground foreground and background colour
***************************************************************************************/
void TFT_eSPI::setBitmapColor(uint16_t c, uint16_t b)
{
  if (c == b) b = ~c;
  bitmap_fg = c;
  bitmap_bg = b;
}
```

```
/***************************************************************************************
** Function name:           setTextWrap
** Description:             Define if text should wrap at end of line
***************************************************************************************/
void TFT_eSPI::setTextWrap(bool wrapX, bool wrapY)
{
  textwrapX = wrapX;
  textwrapY = wrapY;
}
```

```
/**************************************************************************************
** Function name:           setTextDatum
** Description:             Set the text position reference datum
**************************************************************************************/
void TFT_eSPI::setTextDatum(uint8_t d)
{
  textdatum = d;
}
```

```
/***************************************************************************************
** Function name:           setTextPadding
** Description:             Define padding width (aids erasing old text and numbers)
***************************************************************************************/
void TFT_eSPI::setTextPadding(uint16_t x_width)
{
  padX = x_width;
}
```

```
/***************************************************************************************
** Function name:           setTextPadding
** Description:             Define padding width (aids erasing old text and numbers)
***************************************************************************************/
uint16_t TFT_eSPI::getTextPadding(void)
{
  return padX;
}
```

```
/***************************************************************************************
** Function name:           getRotation
** Description:             Return the rotation value (as used by setRotation())
***************************************************************************************/
uint8_t TFT_eSPI::getRotation(void)
{
  return rotation;
}
```

```
/***************************************************************************************
** Function name:           getTextDatum
** Description:             Return the text datum value (as used by setTextDatum())
***************************************************************************************/
uint8_t TFT_eSPI::getTextDatum(void)
{
  return textdatum;
}
```

```
/***************************************************************************************
** Function name:           width
** Description:             Return the pixel width of display (per current rotation)
***************************************************************************************/
// Return the size of the display (per current rotation)
int16_t TFT_eSPI::width(void)
{
  if (_vpDatum) return _xWidth;
  return _width;
}
```

```
/***************************************************************************************
** Function name:           height
** Description:             Return the pixel height of display (per current rotation)
***************************************************************************************/
int16_t TFT_eSPI::height(void)
{
  if (_vpDatum) return _yHeight;
  return _height;
}
```

```
/***************************************************************************************
** Function name:           textWidth
** Description:             Return the width in pixels of a string in a given font
***************************************************************************************/
int16_t TFT_eSPI::textWidth(const String& string)
{
  int16_t len = string.length() + 2;
  char buffer[len];
  string.toCharArray(buffer, len);
  return textWidth(buffer, textfont);
}

int16_t TFT_eSPI::textWidth(const String& string, uint8_t font)
{
  int16_t len = string.length() + 2;
  char buffer[len];
  string.toCharArray(buffer, len);
  return textWidth(buffer, font);
}

int16_t TFT_eSPI::textWidth(const char *string)
{
  return textWidth(string, textfont);
}

int16_t TFT_eSPI::textWidth(const char *string, uint8_t font)
{
  int32_t str_width = 0;
  uint16_t uniCode  = 0;

#ifdef SMOOTH_FONT
  if(fontLoaded) {
    while (*string) {
      uniCode = decodeUTF8(*string++);
      if (uniCode) {
        if (uniCode == 0x20) str_width += gFont.spaceWidth;
        else {
          uint16_t gNum = 0;
          bool found = getUnicodeIndex(uniCode, &gNum);
          if (found) {
            if(str_width == 0 && gdX[gNum] < 0) str_width -= gdX[gNum];
            if (*string || isDigits) str_width += gxAdvance[gNum];
            else str_width += (gdX[gNum] + gWidth[gNum]);
          }
          else str_width += gFont.spaceWidth + 1;
        }
      }
    }
    isDigits = false;
    return str_width;
  }
#endif

  if (font>1 && font<9) {
    char *widthtable = (char *)pgm_read_dword( &(fontdata[font].widthtbl ) ) - 32; //subtract the 32 outside the loo

    while (*string) {
      uniCode = *(string++);
      if (uniCode > 31 && uniCode < 128)
      str_width += pgm_read_byte( widthtable + uniCode); // Normally we need to subtract 32 from uniCode
      else str_width += pgm_read_byte( widthtable + 32); // Set illegal character = space width
    }

  }
  else {

#ifdef LOAD_GFXFF
    if(gfxFont) { // New font
      while (*string) {
        uniCode = decodeUTF8(*string++);
        if ((uniCode >= pgm_read_word(&gfxFont->first)) && (uniCode <= pgm_read_word(&gfxFont->last ))) {
          uniCode -= pgm_read_word(&gfxFont->first);
          GFXglyph *glyph  = &(((GFXglyph *)pgm_read_dword(&gfxFont->glyph))[uniCode]);
          // If this is not the  last character or is a digit then use xAdvance
          if (*string  || isDigits) str_width += pgm_read_byte(&glyph->xAdvance);
          // Else use the offset plus width since this can be bigger than xAdvance
          else str_width += ((int8_t)pgm_read_byte(&glyph->xOffset) + pgm_read_byte(&glyph->width));
        }
      }
    }
    else
#endif
    {
#ifdef LOAD_GLCD
```

```
#endif
    }
  }
  isDigits = false;
  return str_width * textsize;
}
```

```
/***************************************************************************************
** Function name:           fontsLoaded
** Description:             return an encoded 16 bit value showing the fonts loaded
***************************************************************************************/
// Returns a value showing which fonts are loaded (bit N set =  Font N loaded)

uint16_t TFT_eSPI::fontsLoaded(void)
{
  return fontsloaded;
}
```

```
/***************************************************************************************
** Function name:           fontHeight
** Description:             return the height of a font (yAdvance for free fonts)
***************************************************************************************/
int16_t TFT_eSPI::fontHeight(int16_t font)
{
#ifdef SMOOTH_FONT
  if(fontLoaded) return gFont.yAdvance;
#endif

#ifdef LOAD_GFXFF
  if (font==1) {
    if(gfxFont) { // New font
      return pgm_read_byte(&gfxFont->yAdvance) * textsize;
    }
  }
#endif
  return pgm_read_byte( &fontdata[font].height ) * textsize;
}

int16_t TFT_eSPI::fontHeight(void)
{
  return fontHeight(textfont);
}
```

```
/***************************************************************************************
** Function name:           drawChar
** Description:             draw a single character in the GLCD or GFXFF font
***************************************************************************************/
void TFT_eSPI::drawChar(int32_t x, int32_t y, uint16_t c, uint32_t color, uint32_t bg, uint8_t size)
{
  if (_vpOoB) return;

  int32_t xd = x + _xDatum;
  int32_t yd = y + _yDatum;

  if (c < 32) return;
#ifdef LOAD_GLCD
//>>>>>>>>>>>>>>>>>>
  #ifdef LOAD_GFXFF
  if(!gfxFont) { // 'Classic' built-in font
  #endif
//>>>>>>>>>>>>>>>>>>

  if ((xd >= _vpW)               || // Clip right
     ( yd >= _vpH)               || // Clip bottom
     ((xd + 6 * size - 1) < _vpX)  || // Clip left
     ((yd + 8 * size - 1) < _vpY))    // Clip top
    return;

  bool fillbg = (bg != color);
  bool clip = xd < _vpX || xd + 6  * textsize >= _vpW || yd < _vpY || yd + 8 * textsize >= _vpH;

  if ((size==1) && fillbg && !clip) {
    uint8_t column[6];
    uint8_t mask = 0x1;
    begin_tft_write();

    setWindow(xd, yd, xd+5, yd+8);

    for (int8_t i = 0; i < 5; i++ ) column[i] = pgm_read_byte(font + (c * 5) + i);
    column[5] = 0;

    for (int8_t j = 0; j < 8; j++) {
      for (int8_t k = 0; k < 5; k++ ) {
        if (column[k] & mask) {tft_Write_16(color);}
        else {tft_Write_16(bg);}
      }
      mask <<= 1;
      tft_Write_16(bg);
    }

    end_tft_write();
  }
  else {
    //begin_tft_write();          // Sprite class can use this function, avoiding begin_tft_write()
    inTransaction = true;

    for (int8_t i = 0; i < 6; i++ ) {
      uint8_t line;
      if (i == 5)
        line = 0x0;
      else
        line = pgm_read_byte(font + (c * 5) + i);

      if (size == 1 && !fillbg) { // default size
        for (int8_t j = 0; j < 8; j++) {
          if (line & 0x1) drawPixel(x + i, y + j, color);
          line >>= 1;
        }
      }
      else {  // big size or clipped
        for (int8_t j = 0; j < 8; j++) {
          if (line & 0x1) fillRect(x + (i * size), y + (j * size), size, size, color);
          else if (fillbg) fillRect(x + i * size, y + j * size, size, size, bg);
          line >>= 1;
        }
      }
    }
    inTransaction = lockTransaction;
    end_tft_write();              // Does nothing if Sprite class uses this function
  }

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>
  #ifdef LOAD_GFXFF
  } else { // Custom font
  #endif
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#endif // LOAD_GLCD
```

```
#ifdef LOAD_GFXFF
   // Filter out bad characters not present in font
   if ((c >= pgm_read_word(&gfxFont->first)) && (c <= pgm_read_word(&gfxFont->last ))) {
     //begin_tft_write();          // Sprite class can use this function, avoiding begin_tft_write()
     inTransaction = true;
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

     c -= pgm_read_word(&gfxFont->first);
     GFXglyph *glyph  = &(((GFXglyph *)pgm_read_dword(&gfxFont->glyph))[c]);
     uint8_t  *bitmap = (uint8_t *)pgm_read_dword(&gfxFont->bitmap);

     uint32_t bo = pgm_read_word(&glyph->bitmapOffset);
     uint8_t  w  = pgm_read_byte(&glyph->width),
              h  = pgm_read_byte(&glyph->height);
              //xa = pgm_read_byte(&glyph->xAdvance);
     int8_t   xo = pgm_read_byte(&glyph->xOffset),
              yo = pgm_read_byte(&glyph->yOffset);
     uint8_t  xx, yy, bits=0, bit=0;
     int16_t  xo16 = 0, yo16 = 0;

     if(size > 1) {
       xo16 = xo;
       yo16 = yo;
     }

     // GFXFF rendering speed up
     uint16_t hpc = 0; // Horizontal foreground pixel count
     for(yy=0; yy<h; yy++) {
       for(xx=0; xx<w; xx++) {
         if(bit == 0) {
           bits = pgm_read_byte(&bitmap[bo++]);
           bit  = 0x80;
         }
         if(bits & bit) hpc++;
         else {
          if (hpc) {
             if(size == 1) drawFastHLine(x+xo+xx-hpc, y+yo+yy, hpc, color);
             else fillRect(x+(xo16+xx-hpc)*size, y+(yo16+yy)*size, size*hpc, size, color);
             hpc=0;
           }
         }
         bit >>= 1;
       }
       // Draw pixels for this line as we are about to increment yy
       if (hpc) {
         if(size == 1) drawFastHLine(x+xo+xx-hpc, y+yo+yy, hpc, color);
         else fillRect(x+(xo16+xx-hpc)*size, y+(yo16+yy)*size, size*hpc, size, color);
         hpc=0;
       }
     }

     inTransaction = lockTransaction;
     end_tft_write();              // Does nothing if Sprite class uses this function
   }
#endif

#ifdef LOAD_GLCD
  #ifdef LOAD_GFXFF
  } // End classic vs custom font
  #endif
#endif

}
```

```
/***************************************************************************************
** Function name:           setAddrWindow
** Description:             define an area to receive a stream of pixels
***************************************************************************************/
// Chip select is high at the end of this function
void TFT_eSPI::setAddrWindow(int32_t x0, int32_t y0, int32_t w, int32_t h)
{
  begin_tft_write();

  setWindow(x0, y0, x0 + w - 1, y0 + h - 1);

  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           setWindow
** Description:             define an area to receive a stream of pixels
***************************************************************************************/
// Chip select stays low, call begin_tft_write first. Use setAddrWindow() from sketches
void TFT_eSPI::setWindow(int32_t x0, int32_t y0, int32_t x1, int32_t y1)
{
  //begin_tft_write(); // Must be called before setWindow
  addr_row = 0xFFFF;
  addr_col = 0xFFFF;

#if defined (ILI9225_DRIVER)
  if (rotation & 0x01) { swap_coord(x0, y0); swap_coord(x1, y1); }
  SPI_BUSY_CHECK;
  DC_C; tft_Write_8(TFT_CASET1);
  DC_D; tft_Write_16(x0);
  DC_C; tft_Write_8(TFT_CASET2);
  DC_D; tft_Write_16(x1);

  DC_C; tft_Write_8(TFT_PASET1);
  DC_D; tft_Write_16(y0);
  DC_C; tft_Write_8(TFT_PASET2);
  DC_D; tft_Write_16(y1);

  DC_C; tft_Write_8(TFT_RAM_ADDR1);
  DC_D; tft_Write_16(x0);
  DC_C; tft_Write_8(TFT_RAM_ADDR2);
  DC_D; tft_Write_16(y0);

  // write to RAM
  DC_C; tft_Write_8(TFT_RAMWR);
  DC_D;
#elif defined (SSD1351_DRIVER)
  if (rotation & 1) {
    swap_coord(x0, y0);
    swap_coord(x1, y1);
  }
  SPI_BUSY_CHECK;
  DC_C; tft_Write_8(TFT_CASET);
  DC_D; tft_Write_16(x1 | (x0 << 8));
  DC_C; tft_Write_8(TFT_PASET);
  DC_D; tft_Write_16(y1 | (y0 << 8));
  DC_C; tft_Write_8(TFT_RAMWR);
  DC_D;
#else
  #if defined (SSD1963_DRIVER)
    if ((rotation & 0x1) == 0) { swap_coord(x0, y0); swap_coord(x1, y1); }
  #endif

  #ifdef CGRAM_OFFSET
    x0+=colstart;
    x1+=colstart;
    y0+=rowstart;
    y1+=rowstart;
  #endif

  // Temporary solution is to include the RP2040 optimised code here
  #if defined(ARDUINO_ARCH_RP2040) && !defined(TFT_PARALLEL_8BIT)
    while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
    DC_C;
    #if !defined (SPI_18BIT_DRIVER)
      spi_set_format(spi0,  8, (spi_cpol_t)0, (spi_cpha_t)0, SPI_MSB_FIRST);
    #endif
    spi_get_hw(spi0)->dr = (uint32_t)TFT_CASET;

    while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
    DC_D;
    spi_get_hw(spi0)->dr = (uint32_t)x0>>8;
    spi_get_hw(spi0)->dr = (uint32_t)x0;
    spi_get_hw(spi0)->dr = (uint32_t)x1>>8;
    spi_get_hw(spi0)->dr = (uint32_t)x1;

    while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
    DC_C;
    spi_get_hw(spi0)->dr = (uint32_t)TFT_PASET;

    while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
    DC_D;
    spi_get_hw(spi0)->dr = (uint32_t)y0>>8;
    spi_get_hw(spi0)->dr = (uint32_t)y0;
    spi_get_hw(spi0)->dr = (uint32_t)y1>>8;
    spi_get_hw(spi0)->dr = (uint32_t)y1;

    while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
```

```
    spi_get_hw(spi0)->dr = (uint32_t)TFT_RAMWR;

    while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
    #if !defined (SPI_18BIT_DRIVER)
      spi_set_format(spi0, 16, (spi_cpol_t)0, (spi_cpha_t)0, SPI_MSB_FIRST);
    #endif
    DC_D;

  #else
    SPI_BUSY_CHECK;
    DC_C; tft_Write_8(TFT_CASET);
    DC_D; tft_Write_32C(x0, x1);
    DC_C; tft_Write_8(TFT_PASET);
    DC_D; tft_Write_32C(y0, y1);
    DC_C; tft_Write_8(TFT_RAMWR);
    DC_D;
  #endif // RP2040 SPI
#endif
  //end_tft_write(); // Must be called after setWindow
}
```

```
/***************************************************************************************
** Function name:           readAddrWindow
** Description:             define an area to read a stream of pixels
***************************************************************************************/
void TFT_eSPI::readAddrWindow(int32_t xs, int32_t ys, int32_t w, int32_t h)
{
  //begin_tft_write(); // Must be called before readAddrWindow or CS set low

  int32_t xe = xs + w - 1;
  int32_t ye = ys + h - 1;

  addr_col = 0xFFFF;
  addr_row = 0xFFFF;

#ifdef CGRAM_OFFSET
  xs += colstart;
  xe += colstart;
  ys += rowstart;
  ye += rowstart;
#endif

#if defined (SSD1963_DRIVER)
  if ((rotation & 0x1) == 0) { swap_coord(xs, ys); swap_coord(xe, ye); }
#endif

  // Temporary solution is to include the RP2040 optimised code here
#if defined(ARDUINO_ARCH_RP2040) && !defined(TFT_PARALLEL_8BIT)
  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
  DC_C;
  spi_set_format(spi0,  8, (spi_cpol_t)0, (spi_cpha_t)0, SPI_MSB_FIRST);
  spi_get_hw(spi0)->dr = (uint32_t)TFT_CASET;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
  DC_D;
  spi_get_hw(spi0)->dr = (uint32_t)xs>>8;
  spi_get_hw(spi0)->dr = (uint32_t)xs;
  spi_get_hw(spi0)->dr = (uint32_t)xe>>8;
  spi_get_hw(spi0)->dr = (uint32_t)xe;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
  DC_C;
  spi_get_hw(spi0)->dr = (uint32_t)TFT_PASET;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
  DC_D;
  spi_get_hw(spi0)->dr = (uint32_t)ys>>8;
  spi_get_hw(spi0)->dr = (uint32_t)ys;
  spi_get_hw(spi0)->dr = (uint32_t)ye>>8;
  spi_get_hw(spi0)->dr = (uint32_t)ye;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
  DC_C;
  spi_get_hw(spi0)->dr = (uint32_t)TFT_RAMRD;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
  //spi_set_format(spi0, 8, (spi_cpol_t)0, (spi_cpha_t)0, SPI_MSB_FIRST);
  DC_D;

  // Flush the rx buffer and reset overflow flag
  while (spi_is_readable(spi0)) (void)spi_get_hw(spi0)->dr;
  spi_get_hw(spi0)->icr = SPI_SSPICR_RORIC_BITS;

#else
  // Column addr set
  DC_C; tft_Write_8(TFT_CASET);
  DC_D; tft_Write_32C(xs, xe);

  // Row addr set
  DC_C; tft_Write_8(TFT_PASET);
  DC_D; tft_Write_32C(ys, ye);

  // Read CGRAM command
  DC_C; tft_Write_8(TFT_RAMRD);

  DC_D;
#endif // RP2040 SPI

  //end_tft_write(); // Must be called after readAddrWindow or CS set high
}
```

```
/***************************************************************************************
** Function name:           drawPixel
** Description:             push a single pixel at an arbitrary position
***************************************************************************************/
void TFT_eSPI::drawPixel(int32_t x, int32_t y, uint32_t color)
{
  if (_vpOoB) return;

  x+= _xDatum;
  y+= _yDatum;

  // Range checking
  if ((x < _vpX) || (y < _vpY) ||(x >= _vpW) || (y >= _vpH)) return;

#ifdef CGRAM_OFFSET
  x+=colstart;
  y+=rowstart;
#endif

  begin_tft_write();

#if defined (ILI9225_DRIVER)

  if (rotation & 0x01) { swap_coord(x, y); }

  SPI_BUSY_CHECK;

  // Set window to full screen to optimise sequential pixel rendering
  if (addr_row != 0x9225) {
    addr_row = 0x9225; // addr_row used for flag
    DC_C; tft_Write_8(TFT_CASET1);
    DC_D; tft_Write_16(0);
    DC_C; tft_Write_8(TFT_CASET2);
    DC_D; tft_Write_16(175);

    DC_C; tft_Write_8(TFT_PASET1);
    DC_D; tft_Write_16(0);
    DC_C; tft_Write_8(TFT_PASET2);
    DC_D; tft_Write_16(219);
  }

  // Define pixel coordinate
  DC_C; tft_Write_8(TFT_RAM_ADDR1);
  DC_D; tft_Write_16(x);
  DC_C; tft_Write_8(TFT_RAM_ADDR2);
  DC_D; tft_Write_16(y);

  // write to RAM
  DC_C; tft_Write_8(TFT_RAMWR);
  #if defined(TFT_PARALLEL_8_BIT) || !defined(ESP32)
    DC_D; tft_Write_16(color);
  #else
    DC_D; tft_Write_16N(color);
  #endif

  // Temporary solution is to include the RP2040 optimised code here
#elif defined (ARDUINO_ARCH_RP2040)

  // Since the SPI functions do not terminate until transmission is complete
  // a busy check is not needed.
  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
  DC_C;
  spi_set_format(spi0,  8, (spi_cpol_t)0, (spi_cpha_t)0, SPI_MSB_FIRST);
  spi_get_hw(spi0)->dr = (uint32_t)TFT_CASET;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS){};
  DC_D;
  spi_get_hw(spi0)->dr = (uint32_t)x>>8;
  spi_get_hw(spi0)->dr = (uint32_t)x;
  spi_get_hw(spi0)->dr = (uint32_t)x>>8;
  spi_get_hw(spi0)->dr = (uint32_t)x;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
  DC_C;
  spi_get_hw(spi0)->dr = (uint32_t)TFT_PASET;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
  DC_D;
  spi_get_hw(spi0)->dr = (uint32_t)y>>8;
  spi_get_hw(spi0)->dr = (uint32_t)y;
  spi_get_hw(spi0)->dr = (uint32_t)y>>8;
  spi_get_hw(spi0)->dr = (uint32_t)y;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
```

```
  spi_get_hw(spi0)->dr = (uint32_t)TFT_RAMWR;

  #if defined (SPI_18BIT_DRIVER) // SPI 18 bit colour
    uint8_t r = (color & 0xF800)>>8;
    uint8_t g = (color & 0x07E0)>>3;
    uint8_t b = (color & 0x001F)<<3;
    while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
    DC_D;
    tft_Write_8N(r); tft_Write_8N(g); tft_Write_8N(b);
  #else
    while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};
    DC_D;
    spi_get_hw(spi0)->dr = (uint32_t)color>>8;
    spi_get_hw(spi0)->dr = (uint32_t)color;
  #endif
/*
  // Subsequent pixel reads work OK without draining the FIFO...
  // Drain RX FIFO, then wait for shifting to finish (which may be *after*
  // TX FIFO drains), then drain RX FIFO again
  while (spi_is_readable(spi0))
      (void)spi_get_hw(spi0)->dr;
  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS)
      tight_loop_contents();
  while (spi_is_readable(spi0))
      (void)spi_get_hw(spi0)->dr;
//*/

//  Subsequent pixel reads work without this
//  spi_get_hw(spi0)->icr = SPI_SSPICR_RORIC_BITS;

  while (spi_get_hw(spi0)->sr & SPI_SSPSR_BSY_BITS) {};

  // Next call will start with 8 bit command so changing to 16 bit not needed here
  //spi_set_format(spi0,  16, (spi_cpol_t)0, (spi_cpha_t)0, SPI_MSB_FIRST);

#else

#if defined (SSD1351_DRIVER) || defined (SSD1963_DRIVER)
  if ((rotation & 0x1) == 0) { swap_coord(x, y); }
#endif

  SPI_BUSY_CHECK;

#if defined (MULTI_TFT_SUPPORT) || defined (GC9A01_DRIVER)
  // No optimisation
  DC_C; tft_Write_8(TFT_CASET);
  DC_D; tft_Write_32D(x);
  DC_C; tft_Write_8(TFT_PASET);
  DC_D; tft_Write_32D(y);
#elif defined (SSD1351_DRIVER)
  // No need to send x if it has not changed (speeds things up)
  if (addr_col != x) {
    DC_C; tft_Write_8(TFT_CASET);
    DC_D; tft_Write_16(x | (x << 8));
    addr_col = x;
  }

  // No need to send y if it has not changed (speeds things up)
  if (addr_row != y) {
    DC_C; tft_Write_8(TFT_PASET);
    DC_D; tft_Write_16(y | (y << 8));
    addr_row = y;
  }
#else
  // No need to send x if it has not changed (speeds things up)
  if (addr_col != x) {
    DC_C; tft_Write_8(TFT_CASET);
    DC_D; tft_Write_32D(x);
    addr_col = x;
  }

  // No need to send y if it has not changed (speeds things up)
  if (addr_row != y) {
    DC_C; tft_Write_8(TFT_PASET);
    DC_D; tft_Write_32D(y);
    addr_row = y;
  }
#endif
  DC_C; tft_Write_8(TFT_RAMWR);

  #if defined(TFT_PARALLEL_8_BIT) || !defined(ESP32)
    DC_D; tft_Write_16(color);
  #else
    DC_D; tft_Write_16N(color);
```

```
#endif

  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           pushColor
** Description:             push a single pixel
***************************************************************************************/
void TFT_eSPI::pushColor(uint16_t color)
{
  begin_tft_write();

  tft_Write_16(color);

  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           pushColor
** Description:             push a single colour to "len" pixels
***************************************************************************************/
void TFT_eSPI::pushColor(uint16_t color, uint32_t len)
{
  begin_tft_write();

  pushBlock(color, len);

  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           startWrite
** Description:             begin transaction with CS low, MUST later call endWrite
***************************************************************************************/
void TFT_eSPI::startWrite(void)
{
  begin_tft_write();
  lockTransaction = true; // Lock transaction for all sequentially run sketch functions
  inTransaction = true;
}
```

```
/**************************************************************************************
** Function name:           endWrite
** Description:             end transaction with CS high
**************************************************************************************/
void TFT_eSPI::endWrite(void)
{
  lockTransaction = false; // Release sketch induced transaction lock
  inTransaction = false;
  DMA_BUSY_CHECK;          // Safety check - user code should have checked this!
  end_tft_write();         // Release SPI bus
}
```

```
/***************************************************************************************
** Function name:           writeColor (use startWrite() and endWrite() before & after)
** Description:             raw write of "len" pixels avoiding transaction check
***************************************************************************************/
void TFT_eSPI::writeColor(uint16_t color, uint32_t len)
{
  pushBlock(color, len);
}
```

```
/***************************************************************************************
** Function name:           pushColors
** Description:             push an array of pixels for 16 bit raw image drawing
***************************************************************************************/
// Assumed that setAddrWindow() has previously been called
// len is number of bytes, not pixels
void TFT_eSPI::pushColors(uint8_t *data, uint32_t len)
{
  begin_tft_write();

  pushPixels(data, len>>1);

  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           pushColors
** Description:             push an array of pixels, for image drawing
***************************************************************************************/
void TFT_eSPI::pushColors(uint16_t *data, uint32_t len, bool swap)
{
  begin_tft_write();
  if (swap) {swap = _swapBytes; _swapBytes = true; }

  pushPixels(data, len);

  _swapBytes = swap; // Restore old value
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           drawLine
** Description:             draw a line between 2 arbitrary points
***************************************************************************************/
// Bresenham's algorithm - thx wikipedia - speed enhanced by Bodmer to use
// an efficient FastH/V Line draw routine for line segments of 2 pixels or more
void TFT_eSPI::drawLine(int32_t x0, int32_t y0, int32_t x1, int32_t y1, uint32_t color)
{
  if (_vpOoB) return;

  //begin_tft_write();       // Sprite class can use this function, avoiding begin_tft_write()
  inTransaction = true;

  //x+= _xDatum;             // Not added here, added by drawPixel & drawFastXLine
  //y+= _yDatum;

  bool steep = abs(y1 - y0) > abs(x1 - x0);
  if (steep) {
    swap_coord(x0, y0);
    swap_coord(x1, y1);
  }

  if (x0 > x1) {
    swap_coord(x0, x1);
    swap_coord(y0, y1);
  }

  int32_t dx = x1 - x0, dy = abs(y1 - y0);;

  int32_t err = dx >> 1, ystep = -1, xs = x0, dlen = 0;

  if (y0 < y1) ystep = 1;

  // Split into steep and not steep for FastH/V separation
  if (steep) {
    for (; x0 <= x1; x0++) {
      dlen++;
      err -= dy;
      if (err < 0) {
        if (dlen == 1) drawPixel(y0, xs, color);
        else drawFastVLine(y0, xs, dlen, color);
        dlen = 0;
        y0 += ystep; xs = x0 + 1;
        err += dx;
      }
    }
    if (dlen) drawFastVLine(y0, xs, dlen, color);
  }
  else
  {
    for (; x0 <= x1; x0++) {
      dlen++;
      err -= dy;
      if (err < 0) {
        if (dlen == 1) drawPixel(xs, y0, color);
        else drawFastHLine(xs, y0, dlen, color);
        dlen = 0;
        y0 += ystep; xs = x0 + 1;
        err += dx;
      }
    }
    if (dlen) drawFastHLine(xs, y0, dlen, color);
  }

  inTransaction = lockTransaction;
  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           drawFastVLine
** Description:             draw a vertical line
***************************************************************************************/
void TFT_eSPI::drawFastVLine(int32_t x, int32_t y, int32_t h, uint32_t color)
{
  if (_vpOoB) return;

  x+= _xDatum;
  y+= _yDatum;

  // Clipping
  if ((x < _vpX) || (x >= _vpW) || (y >= _vpH)) return;

  if (y < _vpY) { h += y - _vpY; y = _vpY; }

  if ((y + h) > _vpH) h = _vpH - y;

  if (h < 1) return;

  begin_tft_write();

  setWindow(x, y, x, y + h - 1);

  pushBlock(color, h);

  end_tft_write();
}
```

```
/**************************************************************************************
** Function name:           drawFastHLine
** Description:             draw a horizontal line
**************************************************************************************/
void TFT_eSPI::drawFastHLine(int32_t x, int32_t y, int32_t w, uint32_t color)
{
  if (_vpOoB) return;

  x+= _xDatum;
  y+= _yDatum;

  // Clipping
  if ((y < _vpY) || (x >= _vpW) || (y >= _vpH)) return;

  if (x < _vpX) { w += x - _vpX; x = _vpX; }

  if ((x + w) > _vpW) w = _vpW - x;

  if (w < 1) return;

  begin_tft_write();

  setWindow(x, y, x + w - 1, y);

  pushBlock(color, w);

  end_tft_write();
}
```

```
/**************************************************************************************
** Function name:           fillRect
** Description:             draw a filled rectangle
**************************************************************************************/
void TFT_eSPI::fillRect(int32_t x, int32_t y, int32_t w, int32_t h, uint32_t color)
{
  if (_vpOoB) return;

  x+= _xDatum;
  y+= _yDatum;

  // Clipping
  if ((x >= _vpW) || (y >= _vpH)) return;

  if (x < _vpX) { w += x - _vpX; x = _vpX; }
  if (y < _vpY) { h += y - _vpY; y = _vpY; }

  if ((x + w) > _vpW) w = _vpW - x;
  if ((y + h) > _vpH) h = _vpH - y;

  if ((w < 1) || (h < 1)) return;

  //Serial.print(" _xDatum=");Serial.print( _xDatum);Serial.print(", _yDatum=");Serial.print( _yDatum);
  //Serial.print(", _xWidth=");Serial.print(_xWidth);Serial.print(", _yHeight=");Serial.println(_yHeight);

  //Serial.print(" _vpX=");Serial.print( _vpX);Serial.print(", _vpY=");Serial.print( _vpY);
  //Serial.print(", _vpW=");Serial.print(_vpW);Serial.print(", _vpH=");Serial.println(_vpH);

  //Serial.print(" x=");Serial.print( y);Serial.print(", y=");Serial.print( y);
  //Serial.print(", w=");Serial.print(w);Serial.print(", h=");Serial.println(h);

  begin_tft_write();

  setWindow(x, y, x + w - 1, y + h - 1);

  pushBlock(color, w * h);

  end_tft_write();
}
```

```
/***************************************************************************************
** Function name:           color565
** Description:             convert three 8 bit RGB levels to a 16 bit colour value
***************************************************************************************/
uint16_t TFT_eSPI::color565(uint8_t r, uint8_t g, uint8_t b)
{
  return ((r & 0xF8) << 8) | ((g & 0xFC) << 3) | (b >> 3);
}
```

```
/***************************************************************************************
** Function name:           color16to8
** Description:             convert 16 bit colour to an 8 bit 332 RGB colour value
***************************************************************************************/
uint8_t TFT_eSPI::color16to8(uint16_t c)
{
  return ((c & 0xE000)>>8) | ((c & 0x0700)>>6) | ((c & 0x0018)>>3);
}
```

```
/***************************************************************************************
** Function name:           color8to16
** Description:             convert 8 bit colour to a 16 bit 565 colour value
***************************************************************************************/
uint16_t TFT_eSPI::color8to16(uint8_t color)
{
  uint8_t  blue[] = {0, 11, 21, 31}; // blue 2 to 5 bit colour lookup table
  uint16_t color16 = 0;

  //       =====Green=====     ===============Red==============
  color16  = (color & 0x1C)<<6 | (color & 0xC0)<<5 | (color & 0xE0)<<8;
  //       =====Green=====    =======Blue======
  color16 |= (color & 0x1C)<<3 | blue[color & 0x03];

  return color16;
}
```

```
/***************************************************************************************
** Function name:           color16to24
** Description:             convert 16 bit colour to a 24 bit 888 colour value
***************************************************************************************/
uint32_t TFT_eSPI::color16to24(uint16_t color565)
{
  uint8_t r = (color565 >> 8) & 0xF8; r |= (r >> 5);
  uint8_t g = (color565 >> 3) & 0xFC; g |= (g >> 6);
  uint8_t b = (color565 << 3) & 0xF8; b |= (b >> 5);

  return ((uint32_t)r << 16) | ((uint32_t)g << 8) | ((uint32_t)b << 0);
}
```

```
/***************************************************************************************
** Function name:           color24to16
** Description:             convert 24 bit colour to a 16 bit 565 colour value
***************************************************************************************/
uint32_t TFT_eSPI::color24to16(uint32_t color888)
{
  uint16_t r = (color888 >> 8) & 0xF800;
  uint16_t g = (color888 >> 5) & 0x07E0;
  uint16_t b = (color888 >> 3) & 0x001F;

  return (r | g | b);
}
```

```
/***************************************************************************************
** Function name:           invertDisplay
** Description:             invert the display colours i = 1 invert, i = 0 normal
***************************************************************************************/
void TFT_eSPI::invertDisplay(bool i)
{
  begin_tft_write();
  // Send the command twice as otherwise it does not always work!
  writecommand(i ? TFT_INVON : TFT_INVOFF);
  writecommand(i ? TFT_INVON : TFT_INVOFF);
  end_tft_write();
}


/***************************************************************************************
** Function name:           setAttribute
** Description:             Sets a control parameter of an attribute
***************************************************************************************/
void TFT_eSPI::setAttribute(uint8_t attr_id, uint8_t param) {
   switch (attr_id) {
          break;
       case CP437_SWITCH:
          _cp437 = param;
          break;
       case UTF8_SWITCH:
          _utf8  = param;
          decoderState = 0;
          break;
       case PSRAM_ENABLE:
#if defined (ESP32) && defined (CONFIG_SPIRAM_SUPPORT)
          if (psramFound()) _psram_enable = param; // Enable the use of PSRAM (if available)
          else
#endif
          _psram_enable = false;
          break;
       //case 4: // TBD future feature control
       //    _tbd = param;
       //    break;
   }
}


/***************************************************************************************
** Function name:           getAttribute
** Description:             Get value of an attribute (control parameter)
***************************************************************************************/
uint8_t TFT_eSPI::getAttribute(uint8_t attr_id) {
   switch (attr_id) {
       case CP437_SWITCH: // ON/OFF control of full CP437 character set
          return _cp437;
       case UTF8_SWITCH: // ON/OFF control of UTF-8 decoding
          return _utf8;
       case PSRAM_ENABLE:
          return _psram_enable;
       //case 3: // TBD future feature control
       //    return _tbd;
       //    break;
   }

   return false;
}
```

```
/***************************************************************************************
** Function name:           decodeUTF8
** Description:             Serial UTF-8 decoder with fall-back to extended ASCII
***************************************************************************************x*/
uint16_t TFT_eSPI::decodeUTF8(uint8_t c)
{
  if (!_utf8) return c;

  // 7 bit Unicode Code Point
  if ((c & 0x80) == 0x00) {
    decoderState = 0;
    return c;
  }

  if (decoderState == 0) {
    // 11 bit Unicode Code Point
    if ((c & 0xE0) == 0xC0) {
      decoderBuffer = ((c & 0x1F)<<6);
      decoderState = 1;
      return 0;
    }
    // 16 bit Unicode Code Point
    if ((c & 0xF0) == 0xE0) {
      decoderBuffer = ((c & 0x0F)<<12);
      decoderState = 2;
      return 0;
    }
    // 21 bit Unicode  Code Point not supported so fall-back to extended ASCII
    // if ((c & 0xF8) == 0xF0) return c;
  }
  else {
    if (decoderState == 2) {
      decoderBuffer |= ((c & 0x3F)<<6);
      decoderState--;
      return 0;
    }
    else {
      decoderBuffer |= (c & 0x3F);
      decoderState = 0;
      return decoderBuffer;
    }
  }

  decoderState = 0;

  return c; // fall-back to extended ASCII
}
```

```
/***************************************************************************************
** Function name:           decodeUTF8
** Description:             Line buffer UTF-8 decoder with fall-back to extended ASCII
***************************************************************************************x*/
uint16_t TFT_eSPI::decodeUTF8(uint8_t *buf, uint16_t *index, uint16_t remaining)
{
  uint16_t c = buf[(*index)++];
  //Serial.print("Byte from string = 0x"); Serial.println(c, HEX);

  if (!_utf8) return c;

  // 7 bit Unicode
  if ((c & 0x80) == 0x00) return c;

  // 11 bit Unicode
  if (((c & 0xE0) == 0xC0) && (remaining > 1))
    return ((c & 0x1F)<<6) | (buf[(*index)++]&0x3F);

  // 16 bit Unicode
  if (((c & 0xF0) == 0xE0) && (remaining > 2)) {
    c = ((c & 0x0F)<<12) | ((buf[(*index)++]&0x3F)<<6);
    return  c | ((buf[(*index)++]&0x3F));
  }

  // 21 bit Unicode not supported so fall-back to extended ASCII
  // if ((c & 0xF8) == 0xF0) return c;

  return c; // fall-back to extended ASCII
}
```

```
/***************************************************************************************
** Function name:           alphaBlend
** Description:             Blend 16bit foreground and background
***************************************************************************************x*/
uint16_t TFT_eSPI::alphaBlend(uint8_t alpha, uint16_t fgc, uint16_t bgc)
{
  // For speed use fixed point maths and rounding to permit a power of 2 division
  uint16_t fgR = ((fgc >> 10) & 0x3E) + 1;
  uint16_t fgG = ((fgc >>  4) & 0x7E) + 1;
  uint16_t fgB = ((fgc <<  1) & 0x3E) + 1;

  uint16_t bgR = ((bgc >> 10) & 0x3E) + 1;
  uint16_t bgG = ((bgc >>  4) & 0x7E) + 1;
  uint16_t bgB = ((bgc <<  1) & 0x3E) + 1;

  // Shift right 1 to drop rounding bit and shift right 8 to divide by 256
  uint16_t r = (((fgR * alpha) + (bgR * (255 - alpha))) >> 9);
  uint16_t g = (((fgG * alpha) + (bgG * (255 - alpha))) >> 9);
  uint16_t b = (((fgB * alpha) + (bgB * (255 - alpha))) >> 9);

  // Combine RGB565 colours into 16 bits
  //return ((r&0x18) << 11) | ((g&0x30) << 5) | ((b&0x18) << 0); // 2 bit greyscale
  //return ((r&0x1E) << 11) | ((g&0x3C) << 5) | ((b&0x1E) << 0); // 4 bit greyscale
  return (r << 11) | (g << 5) | (b << 0);
}
```

```
/***************************************************************************************
** Function name:           alphaBlend
** Description:             Blend 16bit foreground and background with dither
***************************************************************************************x*/
uint16_t TFT_eSPI::alphaBlend(uint8_t alpha, uint16_t fgc, uint16_t bgc, uint8_t dither)
{
  if (dither) {
    int16_t alphaDither = (int16_t)alpha - dither + random(2*dither+1); // +/-4 randomised
    alpha = (uint8_t)alphaDither;
    if (alphaDither <  0) alpha = 0;
    if (alphaDither >255) alpha = 255;
  }

  return alphaBlend(alpha, fgc, bgc);
}
```

```
/***************************************************************************************
** Function name:           alphaBlend
** Description:             Blend 24bit foreground and background with optional dither
***************************************************************************************x*/
uint32_t TFT_eSPI::alphaBlend24(uint8_t alpha, uint32_t fgc, uint32_t bgc, uint8_t dither)
{

  if (dither) {
    int16_t alphaDither = (int16_t)alpha - dither + random(2*dither+1); // +/-dither randomised
    alpha = (uint8_t)alphaDither;
    if (alphaDither <  0) alpha = 0;
    if (alphaDither >255) alpha = 255;
  }

  // For speed use fixed point maths and rounding to permit a power of 2 division
  uint16_t fgR = ((fgc >> 15) & 0x1FE) + 1;
  uint16_t fgG = ((fgc >>  7) & 0x1FE) + 1;
  uint16_t fgB = ((fgc <<  1) & 0x1FE) + 1;

  uint16_t bgR = ((bgc >> 15) & 0x1FE) + 1;
  uint16_t bgG = ((bgc >>  7) & 0x1FE) + 1;
  uint16_t bgB = ((bgc <<  1) & 0x1FE) + 1;

  // Shift right 1 to drop rounding bit and shift right 8 to divide by 256
  uint16_t r = (((fgR * alpha) + (bgR * (255 - alpha))) >> 9);
  uint16_t g = (((fgG * alpha) + (bgG * (255 - alpha))) >> 9);
  uint16_t b = (((fgB * alpha) + (bgB * (255 - alpha))) >> 9);

  // Combine RGB colours into 24 bits
  return (r << 16) | (g << 8) | (b << 0);
}
```

```
/***************************************************************************************
** Function name:           write
** Description:             draw characters piped through serial stream
***************************************************************************************/
size_t TFT_eSPI::write(uint8_t utf8)
{
  if (_vpOoB) return 1;

  uint16_t uniCode = decodeUTF8(utf8);

  if (!uniCode) return 1;

  if (utf8 == '\r') return 1;

#ifdef SMOOTH_FONT
  if(fontLoaded) {
    if (uniCode < 32 && utf8 != '\n') return 1;

    drawGlyph(uniCode);

    return 1;
  }
#endif

  if (uniCode == '\n') uniCode+=22; // Make it a valid space character to stop errors
  else if (uniCode < 32) return 1;

  uint16_t cwidth = 0;
  uint16_t cheight = 0;

//vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv DEBUG vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
  //Serial.print((uint8_t) uniCode); // Debug line sends all printed TFT text to serial port
  //Serial.println(uniCode, HEX); // Debug line sends all printed TFT text to serial port
  //delay(5);                     // Debug optional wait for serial port to flush through
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ DEBUG ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
#ifdef LOAD_GFXFF
  if(!gfxFont) {
#endif
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

#ifdef LOAD_FONT2
  if (textfont == 2) {
    if (uniCode > 127) return 1;

    cwidth = pgm_read_byte(widtbl_f16 + uniCode-32);
    cheight = chr_hgt_f16;
    // Font 2 is rendered in whole byte widths so we must allow for this
    cwidth = (cwidth + 6) / 8;  // Width in whole bytes for font 2, should be + 7 but must allow for font width change
    cwidth = cwidth * 8;        // Width converted back to pixels
  }
  #ifdef LOAD_RLE
  else
  #endif
#endif

#ifdef LOAD_RLE
  {
    if ((textfont>2) && (textfont<9)) {
      if (uniCode > 127) return 1;
      // Uses the fontinfo struct array to avoid lots of 'if' or 'switch' statements
      cwidth = pgm_read_byte( (uint8_t *)pgm_read_dword( &(fontdata[textfont].widthtbl ) ) + uniCode-32 );
      cheight= pgm_read_byte( &fontdata[textfont].height );
    }
  }
#endif

#ifdef LOAD_GLCD
  if (textfont==1) {
      cwidth =  6;
      cheight = 8;
  }
#else
  if (textfont==1) return 1;
#endif

  cheight = cheight * textsize;

  if (utf8 == '\n') {
    cursor_y += cheight;
    cursor_x  = 0;
  }
  else {
```

```
      cursor_y += cheight;
      cursor_x = 0;
    }
    if (textwrapY && (cursor_y >= (int32_t) height())) cursor_y = 0;
    cursor_x += drawChar(uniCode, cursor_x, cursor_y, textfont);
  }

//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
#ifdef LOAD_GFXFF
  } // Custom GFX font
  else {
    if(utf8 == '\n') {
      cursor_x  = 0;
      cursor_y += (int16_t)textsize * (uint8_t)pgm_read_byte(&gfxFont->yAdvance);
    } else {
      if (uniCode > pgm_read_word(&gfxFont->last )) return 1;
      if (uniCode < pgm_read_word(&gfxFont->first)) return 1;

      uint16_t   c2    = uniCode - pgm_read_word(&gfxFont->first);
      GFXglyph *glyph = &(((GFXglyph *)pgm_read_dword(&gfxFont->glyph))[c2]);
      uint8_t   w     = pgm_read_byte(&glyph->width),
                h     = pgm_read_byte(&glyph->height);
      if((w > 0) && (h > 0)) { // Is there an associated bitmap?
        int16_t xo = (int8_t)pgm_read_byte(&glyph->xOffset);
        if(textwrapX && ((cursor_x + textsize * (xo + w)) > width())) {
          // Drawing character would go off right edge; wrap to new line
          cursor_x  = 0;
          cursor_y += (int16_t)textsize * (uint8_t)pgm_read_byte(&gfxFont->yAdvance);
        }
        if (textwrapY && (cursor_y >= (int32_t) height())) cursor_y = 0;
        drawChar(cursor_x, cursor_y, uniCode, textcolor, textbgcolor, textsize);
      }
      cursor_x += pgm_read_byte(&glyph->xAdvance) * (int16_t)textsize;
    }
  }
#endif // LOAD_GFXFF
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

  return 1;
}
```

```
/***************************************************************************************
** Function name:           drawChar
** Description:             draw a Unicode glyph onto the screen
***************************************************************************************/
  // TODO: Rationalise with TFT_eSprite
  // Any UTF-8 decoding must be done before calling drawChar()
int16_t TFT_eSPI::drawChar(uint16_t uniCode, int32_t x, int32_t y)
{
  return drawChar(uniCode, x, y, textfont);
}

  // Any UTF-8 decoding must be done before calling drawChar()
int16_t TFT_eSPI::drawChar(uint16_t uniCode, int32_t x, int32_t y, uint8_t font)
{
  if (_vpOoB || !uniCode) return 0;

  if (font==1) {
#ifdef LOAD_GLCD
  #ifndef LOAD_GFXFF
    drawChar(x, y, uniCode, textcolor, textbgcolor, textsize);
    return 6 * textsize;
  #endif
#else
  #ifndef LOAD_GFXFF
    return 0;
  #endif
#endif

#ifdef LOAD_GFXFF
    drawChar(x, y, uniCode, textcolor, textbgcolor, textsize);
    if(!gfxFont) { // 'Classic' built-in font
    #ifdef LOAD_GLCD
      return 6 * textsize;
    #else
      return 0;
    #endif
    }
    else {
      if((uniCode >= pgm_read_word(&gfxFont->first)) && (uniCode <= pgm_read_word(&gfxFont->last) )) {
        uint16_t   c2    = uniCode - pgm_read_word(&gfxFont->first);
        GFXglyph *glyph = &(((GFXglyph *)pgm_read_dword(&gfxFont->glyph))[c2]);
        return pgm_read_byte(&glyph->xAdvance) * textsize;
      }
      else {
        return 0;
      }
    }
#endif
  }

  if ((font>1) && (font<9) && ((uniCode < 32) || (uniCode > 127))) return 0;

  int32_t width  = 0;
  int32_t height = 0;
  uint32_t flash_address = 0;
  uniCode -= 32;

#ifdef LOAD_FONT2
  if (font == 2) {
    flash_address = pgm_read_dword(&chrtbl_f16[uniCode]);
    width = pgm_read_byte(widtbl_f16 + uniCode);
    height = chr_hgt_f16;
  }
  #ifdef LOAD_RLE
  else
  #endif
#endif

#ifdef LOAD_RLE
  {
    if ((font>2) && (font<9)) {
      flash_address = pgm_read_dword( (const void*)(pgm_read_dword( &(fontdata[font].chartbl ) ) + uniCode*size
      width = pgm_read_byte( (uint8_t *)pgm_read_dword( &(fontdata[font].widthtbl ) ) + uniCode );
      height= pgm_read_byte( &fontdata[font].height );
    }
  }
#endif

  int32_t xd = x + _xDatum;
  int32_t yd = y + _yDatum;

  if ((xd + width * textsize < _vpX || xd >= _vpW) && (yd + height * textsize < _vpY || yd >= _vpH)) return width

  int32_t w = width;
```

```
  int32_t pY      = y;
  uint8_t line = 0;
  bool clip = xd < _vpX || xd + width  * textsize >= _vpW || yd < _vpY || yd + height * textsize >= _vpH;

#ifdef LOAD_FONT2 // chop out code if we do not need it
  if (font == 2) {
    w = w + 6; // Should be + 7 but we need to compensate for width increment
    w = w / 8;

    if (textcolor == textbgcolor || textsize != 1 || clip) {
      //begin_tft_write();          // Sprite class can use this function, avoiding begin_tft_write()
      inTransaction = true;

      for (int32_t i = 0; i < height; i++) {
        if (textcolor != textbgcolor) fillRect(x, pY, width * textsize, textsize, textbgcolor);

        for (int32_t k = 0; k < w; k++) {
          line = pgm_read_byte((uint8_t *)flash_address + w * i + k);
          if (line) {
            if (textsize == 1) {
              pX = x + k * 8;
              if (line & 0x80) drawPixel(pX, pY, textcolor);
              if (line & 0x40) drawPixel(pX + 1, pY, textcolor);
              if (line & 0x20) drawPixel(pX + 2, pY, textcolor);
              if (line & 0x10) drawPixel(pX + 3, pY, textcolor);
              if (line & 0x08) drawPixel(pX + 4, pY, textcolor);
              if (line & 0x04) drawPixel(pX + 5, pY, textcolor);
              if (line & 0x02) drawPixel(pX + 6, pY, textcolor);
              if (line & 0x01) drawPixel(pX + 7, pY, textcolor);
            }
            else {
              pX = x + k * 8 * textsize;
              if (line & 0x80) fillRect(pX, pY, textsize, textsize, textcolor);
              if (line & 0x40) fillRect(pX + textsize, pY, textsize, textsize, textcolor);
              if (line & 0x20) fillRect(pX + 2 * textsize, pY, textsize, textsize, textcolor);
              if (line & 0x10) fillRect(pX + 3 * textsize, pY, textsize, textsize, textcolor);
              if (line & 0x08) fillRect(pX + 4 * textsize, pY, textsize, textsize, textcolor);
              if (line & 0x04) fillRect(pX + 5 * textsize, pY, textsize, textsize, textcolor);
              if (line & 0x02) fillRect(pX + 6 * textsize, pY, textsize, textsize, textcolor);
              if (line & 0x01) fillRect(pX + 7 * textsize, pY, textsize, textsize, textcolor);
            }
          }
        }
        pY += textsize;
      }

      inTransaction = lockTransaction;
      end_tft_write();
    }
    else { // Faster drawing of characters and background using block write

      begin_tft_write();

      setWindow(xd, yd, xd + width - 1, yd + height - 1);

      uint8_t mask;
      for (int32_t i = 0; i < height; i++) {
        pX = width;
        for (int32_t k = 0; k < w; k++) {
          line = pgm_read_byte((uint8_t *) (flash_address + w * i + k) );
          mask = 0x80;
          while (mask && pX) {
            if (line & mask) {tft_Write_16(textcolor);}
            else {tft_Write_16(textbgcolor);}
            pX--;
            mask = mask >> 1;
          }
        }
        if (pX) {tft_Write_16(textbgcolor);}
      }

      end_tft_write();
    }
  }

  #ifdef LOAD_RLE
  else
  #endif
#endif  //FONT2

#ifdef LOAD_RLE  //674 bytes of code
  // Font is not 2 and hence is RLE encoded
  {
    begin_tft_write();
```

```
w *= height; // Now w is total number of pixels in the character
if (textcolor == textbgcolor && !clip) {

  int32_t px = 0, py = pY; // To hold character block start and end column and row values
  int32_t pc = 0; // Pixel count
  uint8_t np = textsize * textsize; // Number of pixels in a drawn pixel

  uint8_t tnp = 0; // Temporary copy of np for while loop
  uint8_t ts = textsize - 1; // Temporary copy of textsize
  // 16 bit pixel count so maximum font size is equivalent to 180x180 pixels in area
  // w is total number of pixels to plot to fill character block
  while (pc < w) {
    line = pgm_read_byte((uint8_t *)flash_address);
    flash_address++;
    if (line & 0x80) {
      line &= 0x7F;
      line++;
      if (ts) {
        px = xd + textsize * (pc % width); // Keep these px and py calculations outside the loop as they are slow
        py = yd + textsize * (pc / width);
      }
      else {
        px = xd + pc % width; // Keep these px and py calculations outside the loop as they are slow
        py = yd + pc / width;
      }
      while (line--) { // In this case the while(line--) is faster
        pc++; // This is faster than putting pc+=line before while()?
        setWindow(px, py, px + ts, py + ts);

        if (ts) {
          tnp = np;
          while (tnp--) {tft_Write_16(textcolor);}
        }
        else {tft_Write_16(textcolor);}
        px += textsize;

        if (px >= (xd + width * textsize)) {
          px = xd;
          py += textsize;
        }
      }
    }
    else {
      line++;
      pc += line;
    }
  }
}
else {
  // Text colour != background and textsize = 1 and character is within viewport area
  // so use faster drawing of characters and background using block write
  if (textcolor != textbgcolor && textsize == 1 && !clip)
  {
    setWindow(xd, yd, xd + width - 1, yd + height - 1);

    // Maximum font size is equivalent to 180x180 pixels in area
    while (w > 0) {
      line = pgm_read_byte((uint8_t *)flash_address++); // 8 bytes smaller when incrementing here
      if (line & 0x80) {
        line &= 0x7F;
        line++; w -= line;
        pushBlock(textcolor,line);
      }
      else {
        line++; w -= line;
        pushBlock(textbgcolor,line);
      }
    }
  }
  else
  {
    int32_t px = 0, py = 0;  // To hold character pixel coords
    int32_t tx = 0, ty = 0;  // To hold character TFT pixel coords
    int32_t pc = 0;          // Pixel count
    int32_t pl = 0;          // Pixel line length
    uint16_t pcol = 0;       // Pixel color
    bool     pf = true;      // Flag for plotting
    while (pc < w) {
      line = pgm_read_byte((uint8_t *)flash_address);
      flash_address++;
      if (line & 0x80) { pcol = textcolor; line &= 0x7F; pf = true;}
      else { pcol = textbgcolor; if (textcolor == textbgcolor) pf = false;}
      line++;
```

```
        tx = x + textsize * px;
        py = pc / width;
        ty = y + textsize * py;

        pl = 0;
        pc += line;
        while (line--) {
          pl++;
          if ((px+pl) >= width) {
            if (pf) fillRect(tx, ty, pl * textsize, textsize, pcol);
            pl = 0;
            px = 0;
            tx = x;
            py ++;
            ty += textsize;
          }
        }
        if (pl && pf) fillRect(tx, ty, pl * textsize, textsize, pcol);
      }
    }
   }
   inTransaction = lockTransaction;
   end_tft_write();
  }
  // End of RLE font rendering
#endif
  return width * textsize;    // x +
}
```

```
/***************************************************************************************
** Function name:           drawString (with or without user defined font)
** Description :            draw string with padding if it is defined
***************************************************************************************/
// Without font number, uses font set by setTextFont()
int16_t TFT_eSPI::drawString(const String& string, int32_t poX, int32_t poY)
{
  int16_t len = string.length() + 2;
  char buffer[len];
  string.toCharArray(buffer, len);
  return drawString(buffer, poX, poY, textfont);
}
// With font number
int16_t TFT_eSPI::drawString(const String& string, int32_t poX, int32_t poY, uint8_t font)
{
  int16_t len = string.length() + 2;
  char buffer[len];
  string.toCharArray(buffer, len);
  return drawString(buffer, poX, poY, font);
}

// Without font number, uses font set by setTextFont()
int16_t TFT_eSPI::drawString(const char *string, int32_t poX, int32_t poY)
{
  return drawString(string, poX, poY, textfont);
}

// With font number. Note: font number is over-ridden if a smooth font is loaded
int16_t TFT_eSPI::drawString(const char *string, int32_t poX, int32_t poY, uint8_t font)
{
  int16_t sumX = 0;
  uint8_t padding = 1, baseline = 0;
  uint16_t cwidth = textWidth(string, font); // Find the pixel width of the string in the font
  uint16_t cheight = 8 * textsize;

#ifdef LOAD_GFXFF
  #ifdef SMOOTH_FONT
    bool freeFont = (font == 1 && gfxFont && !fontLoaded);
  #else
    bool freeFont = (font == 1 && gfxFont);
  #endif

  if (freeFont) {
    cheight = glyph_ab * textsize;
    poY += cheight; // Adjust for baseline datum of free fonts
    baseline = cheight;
    padding =101; // Different padding method used for Free Fonts

    // We need to make an adjustment for the bottom of the string (eg 'y' character)
    if ((textdatum == BL_DATUM) || (textdatum == BC_DATUM) || (textdatum == BR_DATUM)) {
      cheight += glyph_bb * textsize;
    }
  }
#endif


  // If it is not font 1 (GLCD or free font) get the baseline and pixel height of the font
#ifdef SMOOTH_FONT
  if(fontLoaded) {
    baseline = gFont.maxAscent;
    cheight  = fontHeight();
  }
  else
#endif
  if (font!=1) {
    baseline = pgm_read_byte( &fontdata[font].baseline ) * textsize;
    cheight = fontHeight(font);
  }

  if (textdatum || padX) {

    switch(textdatum) {
      case TC_DATUM:
        poX -= cwidth/2;
        padding += 1;
        break;
      case TR_DATUM:
        poX -= cwidth;
        padding += 2;
        break;
      case ML_DATUM:
        poY -= cheight/2;
        //padding += 0;
        break;
```

```
      poX -= cwidth/2;
      poY -= cheight/2;
      padding += 1;
      break;
    case MR_DATUM:
      poX -= cwidth;
      poY -= cheight/2;
      padding += 2;
      break;
    case BL_DATUM:
      poY -= cheight;
      //padding += 0;
      break;
    case BC_DATUM:
      poX -= cwidth/2;
      poY -= cheight;
      padding += 1;
      break;
    case BR_DATUM:
      poX -= cwidth;
      poY -= cheight;
      padding += 2;
      break;
    case L_BASELINE:
      poY -= baseline;
      //padding += 0;
      break;
    case C_BASELINE:
      poX -= cwidth/2;
      poY -= baseline;
      padding += 1;
      break;
    case R_BASELINE:
      poX -= cwidth;
      poY -= baseline;
      padding += 2;
      break;
   }
  }


  int8_t xo = 0;
#ifdef LOAD_GFXFF
  if (freeFont && (textcolor!=textbgcolor)) {
    cheight = (glyph_ab + glyph_bb) * textsize;
    // Get the offset for the first character only to allow for negative offsets
    uint16_t c2 = 0;
    uint16_t len = strlen(string);
    uint16_t n = 0;

    while (n < len && c2 == 0) c2 = decodeUTF8((uint8_t*)string, &n, len - n);

    if((c2 >= pgm_read_word(&gfxFont->first)) && (c2 <= pgm_read_word(&gfxFont->last) )) {
      c2 -= pgm_read_word(&gfxFont->first);
      GFXglyph *glyph = &(((GFXglyph *)pgm_read_dword(&gfxFont->glyph))[c2]);
      xo = pgm_read_byte(&glyph->xOffset) * textsize;
      // Adjust for negative xOffset
      if (xo > 0) xo = 0;
      else cwidth -= xo;
      // Add 1 pixel of padding all round
      //cheight +=2;
      //fillRect(poX+xo-1, poY - 1 - glyph_ab * textsize, cwidth+2, cheight, textbgcolor);
      fillRect(poX+xo, poY - glyph_ab * textsize, cwidth, cheight, textbgcolor);
    }
    padding -=100;
   }
#endif

  uint16_t len = strlen(string);
  uint16_t n = 0;

#ifdef SMOOTH_FONT
  if(fontLoaded) {
    if (textcolor!=textbgcolor) fillRect(poX, poY, cwidth, cheight, textbgcolor);
/*
    // The above only works for a single text line, not if the text is going to wrap...
    // So need to use code like this in a while loop to fix it:
    if (textwrapX && (cursor_x + width * textsize > width())) {
      cursor_y += height;
      cursor_x = 0;
    }
    if (textwrapY && (cursor_y >= (int32_t)height())) cursor_y = 0;
    cursor_x += drawChar(uniCode, cursor_x, cursor_y, textfont);
*/
```

```
    while (n < len) {
      uint16_t uniCode = decodeUTF8((uint8_t*)string, &n, len - n);
      drawGlyph(uniCode);
    }
    sumX += cwidth;
    //fontFile.close();
  }
  else
#endif
  {
    while (n < len) {
      uint16_t uniCode = decodeUTF8((uint8_t*)string, &n, len - n);
      sumX += drawChar(uniCode, poX+sumX, poY, font);
    }
  }

//vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv DEBUG vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
// Switch on debugging for the padding areas
//#define PADDING_DEBUG

#ifndef PADDING_DEBUG
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ DEBUG ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  if((padX>cwidth) && (textcolor!=textbgcolor)) {
    int16_t padXc = poX+cwidth+xo;
#ifdef LOAD_GFXFF
    if (freeFont) {
      poX +=xo; // Adjust for negative offset start character
      poY -= glyph_ab * textsize;
      sumX += poX;
    }
#endif
    switch(padding) {
      case 1:
        fillRect(padXc,poY,padX-cwidth,cheight, textbgcolor);
        break;
      case 2:
        fillRect(padXc,poY,(padX-cwidth)>>1,cheight, textbgcolor);
        padXc = poX - ((padX-cwidth)>>1);
        fillRect(padXc,poY,(padX-cwidth)>>1,cheight, textbgcolor);
        break;
      case 3:
        if (padXc>padX) padXc = padX;
        fillRect(poX + cwidth - padXc,poY,padXc-cwidth,cheight, textbgcolor);
        break;
    }
  }


#else

//vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv DEBUG vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
// This is debug code to show text (green box) and blanked (white box) areas
// It shows that the padding areas are being correctly sized and positioned

  if((padX>sumX) && (textcolor!=textbgcolor)) {
    int16_t padXc = poX+sumX; // Maximum left side padding
#ifdef LOAD_GFXFF
    if ((font == 1) && (gfxFont)) poY -= glyph_ab;
#endif
    drawRect(poX,poY,sumX,cheight, TFT_GREEN);
    switch(padding) {
      case 1:
        drawRect(padXc,poY,padX-sumX,cheight, TFT_WHITE);
        break;
      case 2:
        drawRect(padXc,poY,(padX-sumX)>>1, cheight, TFT_WHITE);
        padXc = (padX-sumX)>>1;
        drawRect(poX - padXc,poY,(padX-sumX)>>1,cheight, TFT_WHITE);
        break;
      case 3:
        if (padXc>padX) padXc = padX;
        drawRect(poX + sumX - padXc,poY,padXc-sumX,cheight, TFT_WHITE);
        break;
    }
  }
#endif
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ DEBUG ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

return sumX;
}
```

```
/***************************************************************************************
** Function name:           drawCentreString (deprecated, use setTextDatum())
** Descriptions:            draw string centred on dX
***************************************************************************************/
int16_t TFT_eSPI::drawCentreString(const String& string, int32_t dX, int32_t poY, uint8_t font)
{
  int16_t len = string.length() + 2;
  char buffer[len];
  string.toCharArray(buffer, len);
  return drawCentreString(buffer, dX, poY, font);
}

int16_t TFT_eSPI::drawCentreString(const char *string, int32_t dX, int32_t poY, uint8_t font)
{
  uint8_t tempdatum = textdatum;
  int32_t sumX = 0;
  textdatum = TC_DATUM;
  sumX = drawString(string, dX, poY, font);
  textdatum = tempdatum;
  return sumX;
}
```

```
/***************************************************************************************
** Function name:           drawRightString (deprecated, use setTextDatum())
** Descriptions:            draw string right justified to dX
***************************************************************************************/
int16_t TFT_eSPI::drawRightString(const String& string, int32_t dX, int32_t poY, uint8_t font)
{
  int16_t len = string.length() + 2;
  char buffer[len];
  string.toCharArray(buffer, len);
  return drawRightString(buffer, dX, poY, font);
}

int16_t TFT_eSPI::drawRightString(const char *string, int32_t dX, int32_t poY, uint8_t font)
{
  uint8_t tempdatum = textdatum;
  int16_t sumX = 0;
  textdatum = TR_DATUM;
  sumX = drawString(string, dX, poY, font);
  textdatum = tempdatum;
  return sumX;
}
```

```
/***************************************************************************************
** Function name:           drawNumber
** Description:             draw a long integer
***************************************************************************************/
int16_t TFT_eSPI::drawNumber(long long_num, int32_t poX, int32_t poY)
{
  isDigits = true; // Eliminate jiggle in monospaced fonts
  char str[12];
  ltoa(long_num, str, 10);
  return drawString(str, poX, poY, textfont);
}

int16_t TFT_eSPI::drawNumber(long long_num, int32_t poX, int32_t poY, uint8_t font)
{
  isDigits = true; // Eliminate jiggle in monospaced fonts
  char str[12];
  ltoa(long_num, str, 10);
  return drawString(str, poX, poY, font);
}
```

```
/***************************************************************************************
** Function name:           drawFloat
** Descriptions:            drawFloat, prints 7 non zero digits maximum
***************************************************************************************/
// Assemble and print a string, this permits alignment relative to a datum
// looks complicated but much more compact and actually faster than using print class
int16_t TFT_eSPI::drawFloat(float floatNumber, uint8_t dp, int32_t poX, int32_t poY)
{
  return drawFloat(floatNumber, dp, poX, poY, textfont);
}

int16_t TFT_eSPI::drawFloat(float floatNumber, uint8_t dp, int32_t poX, int32_t poY, uint8_t font)
{
  isDigits = true;
  char str[14];               // Array to contain decimal string
  uint8_t ptr = 0;            // Initialise pointer for array
  int8_t  digits = 1;         // Count the digits to avoid array overflow
  float rounding = 0.5;       // Round up down delta

  if (dp > 7) dp = 7; // Limit the size of decimal portion

  // Adjust the rounding value
  for (uint8_t i = 0; i < dp; ++i) rounding /= 10.0;

  if (floatNumber < -rounding) {    // add sign, avoid adding - sign to 0.0!
    str[ptr++] = '-'; // Negative number
    str[ptr] = 0; // Put a null in the array as a precaution
    digits = 0;   // Set digits to 0 to compensate so pointer value can be used later
    floatNumber = -floatNumber; // Make positive
  }

  floatNumber += rounding; // Round up or down

  // For error put ... in string and return (all TFT_eSPI library fonts contain . character)
  if (floatNumber >= 2147483647) {
    strcpy(str, "...");
    return drawString(str, poX, poY, font);
  }
  // No chance of overflow from here on

  // Get integer part
  uint32_t temp = (uint32_t)floatNumber;

  // Put integer part into array
  ltoa(temp, str + ptr, 10);

  // Find out where the null is to get the digit count loaded
  while ((uint8_t)str[ptr] != 0) ptr++; // Move the pointer along
  digits += ptr;                  // Count the digits

  str[ptr++] = '.'; // Add decimal point
  str[ptr] = '0';   // Add a dummy zero
  str[ptr + 1] = 0; // Add a null but don't increment pointer so it can be overwritten

  // Get the decimal portion
  floatNumber = floatNumber - temp;

  // Get decimal digits one by one and put in array
  // Limit digit count so we don't get a false sense of resolution
  uint8_t i = 0;
  while ((i < dp) && (digits < 9)) { // while (i < dp) for no limit but array size must be increased
    i++;
    floatNumber *= 10;       // for the next decimal
    temp = floatNumber;      // get the decimal
    ltoa(temp, str + ptr, 10);
    ptr++; digits++;         // Increment pointer and digits count
    floatNumber -= temp;     // Remove that digit
  }

  // Finally we can plot the string and return pixel length
  return drawString(str, poX, poY, font);
}
```

```
/***************************************************************************************
** Function name:           setFreeFont
** Descriptions:            Sets the GFX free font to use
***************************************************************************************/

#ifdef LOAD_GFXFF

void TFT_eSPI::setFreeFont(const GFXfont *f)
{
  if (f == nullptr) { // Fix issue #400 (ESP32 crash)
    setTextFont(1); // Use GLCD font
    return;
  }

  textfont = 1;
  gfxFont = (GFXfont *)f;

  glyph_ab = 0;
  glyph_bb = 0;
  uint16_t numChars = pgm_read_word(&gfxFont->last) - pgm_read_word(&gfxFont->first);

  // Find the biggest above and below baseline offsets
  for (uint8_t c = 0; c < numChars; c++) {
    GFXglyph *glyph1  = &(((GFXglyph *)pgm_read_dword(&gfxFont->glyph))[c]);
    int8_t ab = -pgm_read_byte(&glyph1->yOffset);
    if (ab > glyph_ab) glyph_ab = ab;
    int8_t bb = pgm_read_byte(&glyph1->height) - ab;
    if (bb > glyph_bb) glyph_bb = bb;
  }
}
```

```
/**************************************************************************************
** Function name:           setTextFont
** Description:             Set the font for the print stream
**************************************************************************************/
void TFT_eSPI::setTextFont(uint8_t f)
{
  textfont = (f > 0) ? f : 1; // Don't allow font 0
  gfxFont = NULL;
}

#else
```

```
/**************************************************************************************
** Function name:           setFreeFont
** Descriptions:            Sets the GFX free font to use
**************************************************************************************/

// Alternative to setTextFont() so we don't need two different named functions
void TFT_eSPI::setFreeFont(uint8_t font)
{
  setTextFont(font);
}
```

```
/***************************************************************************************
** Function name:           setTextFont
** Description:             Set the font for the print stream
***************************************************************************************/
void TFT_eSPI::setTextFont(uint8_t f)
{
  textfont = (f > 0) ? f : 1; // Don't allow font 0
}
#endif
```

```
/***************************************************************************************
** Function name:           getSPIinstance
** Description:             Get the instance of the SPI class
***************************************************************************************/
#if !defined (TFT_PARALLEL_8_BIT)
SPIClass& TFT_eSPI::getSPIinstance(void)
{
  return spi;
}
#endif
```

```
/***************************************************************************************
** Function name:           getSetup
** Description:             Get the setup details for diagnostic and sketch access
***************************************************************************************/
void TFT_eSPI::getSetup(setup_t &tft_settings)
{
// tft_settings.version is set in header file

#if defined (PROCESSOR_ID)
  tft_settings.esp = PROCESSOR_ID;
#else
  tft_settings.esp = -1;
#endif

#if defined (SUPPORT_TRANSACTIONS)
  tft_settings.trans = true;
#else
  tft_settings.trans = false;
#endif

#if defined (TFT_PARALLEL_8_BIT)
  tft_settings.serial = false;
  tft_settings.tft_spi_freq = 0;
#else
  tft_settings.serial = true;
  tft_settings.tft_spi_freq = SPI_FREQUENCY/100000;
  #ifdef SPI_READ_FREQUENCY
    tft_settings.tft_rd_freq = SPI_READ_FREQUENCY/100000;
  #endif
#endif

#if defined(TFT_SPI_OVERLAP)
  tft_settings.overlap = true;
#else
  tft_settings.overlap = false;
#endif

  tft_settings.tft_driver = TFT_DRIVER;
  tft_settings.tft_width  = _init_width;
  tft_settings.tft_height = _init_height;

#ifdef CGRAM_OFFSET
  tft_settings.r0_x_offset = colstart;
  tft_settings.r0_y_offset = rowstart;
  tft_settings.r1_x_offset = 0;
  tft_settings.r1_y_offset = 0;
  tft_settings.r2_x_offset = 0;
  tft_settings.r2_y_offset = 0;
  tft_settings.r3_x_offset = 0;
  tft_settings.r3_y_offset = 0;
#else
  tft_settings.r0_x_offset = 0;
  tft_settings.r0_y_offset = 0;
  tft_settings.r1_x_offset = 0;
  tft_settings.r1_y_offset = 0;
  tft_settings.r2_x_offset = 0;
  tft_settings.r2_y_offset = 0;
  tft_settings.r3_x_offset = 0;
  tft_settings.r3_y_offset = 0;
#endif

#if defined (TFT_MOSI)
  tft_settings.pin_tft_mosi = TFT_MOSI;
#else
  tft_settings.pin_tft_mosi = -1;
#endif

#if defined (TFT_MISO)
  tft_settings.pin_tft_miso = TFT_MISO;
#else
  tft_settings.pin_tft_miso = -1;
#endif

#if defined (TFT_SCLK)
  tft_settings.pin_tft_clk  = TFT_SCLK;
#else
  tft_settings.pin_tft_clk  = -1;
#endif

#if defined (TFT_CS)
  tft_settings.pin_tft_cs   = TFT_CS;
#else
  tft_settings.pin_tft_cs   = -1;
#endif
```

```
#if defined (TFT_DC)
  tft_settings.pin_tft_dc  = TFT_DC;
#else
  tft_settings.pin_tft_dc  = -1;
#endif

#if defined (TFT_RD)
  tft_settings.pin_tft_rd  = TFT_RD;
#else
  tft_settings.pin_tft_rd  = -1;
#endif

#if defined (TFT_WR)
  tft_settings.pin_tft_wr  = TFT_WR;
#else
  tft_settings.pin_tft_wr  = -1;
#endif

#if defined (TFT_RST)
  tft_settings.pin_tft_rst = TFT_RST;
#else
  tft_settings.pin_tft_rst = -1;
#endif

#if defined (TFT_PARALLEL_8_BIT)
  tft_settings.pin_tft_d0 = TFT_D0;
  tft_settings.pin_tft_d1 = TFT_D1;
  tft_settings.pin_tft_d2 = TFT_D2;
  tft_settings.pin_tft_d3 = TFT_D3;
  tft_settings.pin_tft_d4 = TFT_D4;
  tft_settings.pin_tft_d5 = TFT_D5;
  tft_settings.pin_tft_d6 = TFT_D6;
  tft_settings.pin_tft_d7 = TFT_D7;
#else
  tft_settings.pin_tft_d0 = -1;
  tft_settings.pin_tft_d1 = -1;
  tft_settings.pin_tft_d2 = -1;
  tft_settings.pin_tft_d3 = -1;
  tft_settings.pin_tft_d4 = -1;
  tft_settings.pin_tft_d5 = -1;
  tft_settings.pin_tft_d6 = -1;
  tft_settings.pin_tft_d7 = -1;
#endif

#if defined (TFT_BL)
  tft_settings.pin_tft_led = TFT_BL;
#endif

#if defined (TFT_BACKLIGHT_ON)
  tft_settings.pin_tft_led_on = TFT_BACKLIGHT_ON;
#endif

#if defined (TOUCH_CS)
  tft_settings.pin_tch_cs   = TOUCH_CS;
  tft_settings.tch_spi_freq = SPI_TOUCH_FREQUENCY/100000;
#else
  tft_settings.pin_tch_cs   = -1;
  tft_settings.tch_spi_freq = 0;
#endif
}


////////////////////////////////////////////////////////////////////////////////////
#ifdef TOUCH_CS
  #include "Extensions/Touch.cpp"
  #include "Extensions/Button.cpp"
#endif

#include "Extensions/Sprite.cpp"

#ifdef SMOOTH_FONT
  #include "Extensions/Smooth_font.cpp"
#endif
```

```
/*****************************************************************************
** Function name:           TOUCH
** Description:             Define TOUCH_CS is the user setup file to enable this code
*****************************************************************************/
```

```
/***************************************************************************************
** Function name:           begin_touch_read_write - was spi_begin_touch
** Description:             Start transaction and select touch controller
***************************************************************************************/
// The touch controller has a low SPI clock rate
inline void TFT_eSPI::begin_touch_read_write(void){
  DMA_BUSY_CHECK;
  CS_H; // Just in case it has been left low
  #if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS)
    if (locked) {locked = false; spi.beginTransaction(SPISettings(SPI_TOUCH_FREQUENCY, MSBFIRST, SPI_MODE0))
  #else
    spi.setFrequency(SPI_TOUCH_FREQUENCY);
  #endif
  SET_BUS_READ_MODE;
  T_CS_L;
}
```

```
/***************************************************************************************
** Function name:           end_touch_read_write - was spi_end_touch
** Description:             End transaction and deselect touch controller
***************************************************************************************/
inline void TFT_eSPI::end_touch_read_write(void){
  T_CS_H;
  #if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS)
    if(!inTransaction) {if (!locked) {locked = true; spi.endTransaction();}}
  #else
    spi.setFrequency(SPI_FREQUENCY);
  #endif
  //SET_BUS_WRITE_MODE;
}
```

```
/***************************************************************************************
** Function name:           Legacy - deprecated
** Description:             Start/end transaction
***************************************************************************************/
void TFT_eSPI::spi_begin_touch() {begin_touch_read_write();}
void TFT_eSPI::spi_end_touch()   {  end_touch_read_write();}
```

```
/***************************************************************************************
** Function name:           getTouchRaw
** Description:             read raw touch position.  Always returns true.
***************************************************************************************/
uint8_t TFT_eSPI::getTouchRaw(uint16_t *x, uint16_t *y){
  uint16_t tmp;

  begin_touch_read_write();

  // Start YP sample request for x position, read 4 times and keep last sample
  spi.transfer(0xd0);                   // Start new YP conversion
  spi.transfer(0);                      // Read first 8 bits
  spi.transfer(0xd0);                   // Read last 8 bits and start new YP conversion
  spi.transfer(0);                      // Read first 8 bits
  spi.transfer(0xd0);                   // Read last 8 bits and start new YP conversion
  spi.transfer(0);                      // Read first 8 bits
  spi.transfer(0xd0);                   // Read last 8 bits and start new YP conversion

  tmp = spi.transfer(0);                // Read first 8 bits
  tmp = tmp <<5;
  tmp |= 0x1f & (spi.transfer(0x90)>>3);   // Read last 8 bits and start new XP conversion

  *x = tmp;

  // Start XP sample request for y position, read 4 times and keep last sample
  spi.transfer(0);                      // Read first 8 bits
  spi.transfer(0x90);                   // Read last 8 bits and start new XP conversion
  spi.transfer(0);                      // Read first 8 bits
  spi.transfer(0x90);                   // Read last 8 bits and start new XP conversion
  spi.transfer(0);                      // Read first 8 bits
  spi.transfer(0x90);                   // Read last 8 bits and start new XP conversion

  tmp = spi.transfer(0);                // Read first 8 bits
  tmp = tmp <<5;
  tmp |= 0x1f & (spi.transfer(0)>>3);    // Read last 8 bits

  *y = tmp;

  end_touch_read_write();

  return true;
}
```

```
/**************************************************************************************
** Function name:           getTouchRawZ
** Description:             read raw pressure on touchpad and return Z value.
**************************************************************************************/
uint16_t TFT_eSPI::getTouchRawZ(void){

  begin_touch_read_write();

  // Z sample request
  int16_t tz = 0xFFF;
  spi.transfer(0xb0);               // Start new Z1 conversion
  tz += spi.transfer16(0xc0) >> 3;  // Read Z1 and start Z2 conversion
  tz -= spi.transfer16(0x00) >> 3;  // Read Z2

  end_touch_read_write();

  return (uint16_t)tz;
}
```

```
/***************************************************************************************
** Function name:           validTouch
** Description:             read validated position. Return false if not pressed.
***************************************************************************************/
#define _RAWERR 20 // Deadband error allowed in successive position samples
uint8_t TFT_eSPI::validTouch(uint16_t *x, uint16_t *y, uint16_t threshold){
  uint16_t x_tmp, y_tmp, x_tmp2, y_tmp2;

  // Wait until pressure stops increasing to debounce pressure
  uint16_t z1 = 1;
  uint16_t z2 = 0;
  while (z1 > z2)
  {
    z2 = z1;
    z1 = getTouchRawZ();
    delay(1);
  }

  //  Serial.print("Z = ");Serial.println(z1);

  if (z1 <= threshold) return false;

  getTouchRaw(&x_tmp,&y_tmp);

  //  Serial.print("Sample 1 x,y = "); Serial.print(x_tmp);Serial.print(",");Serial.print(y_tmp);
  //  Serial.print(", Z = ");Serial.println(z1);

  delay(1); // Small delay to the next sample
  if (getTouchRawZ() <= threshold) return false;

  delay(2); // Small delay to the next sample
  getTouchRaw(&x_tmp2,&y_tmp2);

  //  Serial.print("Sample 2 x,y = "); Serial.print(x_tmp2);Serial.print(",");Serial.println(y_tmp2);
  //  Serial.print("Sample difference = ");Serial.print(abs(x_tmp - x_tmp2));Serial.print(",");Serial.println(abs(y_tm

  if (abs(x_tmp - x_tmp2) > _RAWERR) return false;
  if (abs(y_tmp - y_tmp2) > _RAWERR) return false;

  *x = x_tmp;
  *y = y_tmp;

  return true;
}
```

```
/***************************************************************************************
** Function name:           getTouch
** Description:             read callibrated position. Return false if not pressed.
***************************************************************************************/
#define Z_THRESHOLD 350 // Touch pressure threshold for validating touches
uint8_t TFT_eSPI::getTouch(uint16_t *x, uint16_t *y, uint16_t threshold){
  uint16_t x_tmp, y_tmp;

  if (threshold<20) threshold = 20;
  if (_pressTime > millis()) threshold=20;

  uint8_t n = 5;
  uint8_t valid = 0;
  while (n--)
  {
    if (validTouch(&x_tmp, &y_tmp, threshold)) valid++;;
  }

  if (valid<1) { _pressTime = 0; return false; }

  _pressTime = millis() + 50;

  convertRawXY(&x_tmp, &y_tmp);

  if (x_tmp >= _width || y_tmp >= _height) return false;

  _pressX = x_tmp;
  _pressY = y_tmp;
  *x = _pressX;
  *y = _pressY;
  return valid;
}
```

```
/***************************************************************************************
** Function name:           convertRawXY
** Description:             convert raw touch x,y values to screen coordinates
***************************************************************************************/
void TFT_eSPI::convertRawXY(uint16_t *x, uint16_t *y)
{
  uint16_t x_tmp = *x, y_tmp = *y, xx, yy;

  if(!touchCalibration_rotate){
    xx=(x_tmp-touchCalibration_x0)*_width/touchCalibration_x1;
    yy=(y_tmp-touchCalibration_y0)*_height/touchCalibration_y1;
    if(touchCalibration_invert_x)
      xx = _width - xx;
    if(touchCalibration_invert_y)
      yy = _height - yy;
  } else {
    xx=(y_tmp-touchCalibration_x0)*_width/touchCalibration_x1;
    yy=(x_tmp-touchCalibration_y0)*_height/touchCalibration_y1;
    if(touchCalibration_invert_x)
      xx = _width - xx;
    if(touchCalibration_invert_y)
      yy = _height - yy;
  }
  *x = xx;
  *y = yy;
}
```

```
/***************************************************************************************
** Function name:           calibrateTouch
** Description:             generates calibration parameters for touchscreen.
***************************************************************************************/
void TFT_eSPI::calibrateTouch(uint16_t *parameters, uint32_t color_fg, uint32_t color_bg, uint8_t size){
  int16_t values[] = {0,0,0,0,0,0,0,0};
  uint16_t x_tmp, y_tmp;



  for(uint8_t i = 0; i<4; i++){
    fillRect(0, 0, size+1, size+1, color_bg);
    fillRect(0, _height-size-1, size+1, size+1, color_bg);
    fillRect(_width-size-1, 0, size+1, size+1, color_bg);
    fillRect(_width-size-1, _height-size-1, size+1, size+1, color_bg);

    if (i == 5) break; // used to clear the arrows

    switch (i) {
      case 0: // up left
        drawLine(0, 0, 0, size, color_fg);
        drawLine(0, 0, size, 0, color_fg);
        drawLine(0, 0, size , size, color_fg);
        break;
      case 1: // bot left
        drawLine(0, _height-size-1, 0, _height-1, color_fg);
        drawLine(0, _height-1, size, _height-1, color_fg);
        drawLine(size, _height-size-1, 0, _height-1 , color_fg);
        break;
      case 2: // up right
        drawLine(_width-size-1, 0, _width-1, 0, color_fg);
        drawLine(_width-size-1, size, _width-1, 0, color_fg);
        drawLine(_width-1, size, _width-1, 0, color_fg);
        break;
      case 3: // bot right
        drawLine(_width-size-1, _height-size-1, _width-1, _height-1, color_fg);
        drawLine(_width-1, _height-1-size, _width-1, _height-1, color_fg);
        drawLine(_width-1-size, _height-1, _width-1, _height-1, color_fg);
        break;
      }

    // user has to get the chance to release
    if(i>0) delay(1000);

    for(uint8_t j= 0; j<8; j++){
      // Use a lower detect threshold as corners tend to be less sensitive
      while(!validTouch(&x_tmp, &y_tmp, Z_THRESHOLD/2));
      values[i*2  ] += x_tmp;
      values[i*2+1] += y_tmp;
      }
    values[i*2  ] /= 8;
    values[i*2+1] /= 8;
  }


  // from case 0 to case 1, the y value changed.
  // If the measured delta of the touch x axis is bigger than the delta of the y axis, the touch and TFT axes are switc
  touchCalibration_rotate = false;
  if(abs(values[0]-values[2]) > abs(values[1]-values[3])){
    touchCalibration_rotate = true;
    touchCalibration_x0 = (values[1] + values[3])/2; // calc min x
    touchCalibration_x1 = (values[5] + values[7])/2; // calc max x
    touchCalibration_y0 = (values[0] + values[4])/2; // calc min y
    touchCalibration_y1 = (values[2] + values[6])/2; // calc max y
  } else {
    touchCalibration_x0 = (values[0] + values[2])/2; // calc min x
    touchCalibration_x1 = (values[4] + values[6])/2; // calc max x
    touchCalibration_y0 = (values[1] + values[5])/2; // calc min y
    touchCalibration_y1 = (values[3] + values[7])/2; // calc max y
  }

  // in addition, the touch screen axis could be in the opposite direction of the TFT axis
  touchCalibration_invert_x = false;
  if(touchCalibration_x0 > touchCalibration_x1){
    values[0]=touchCalibration_x0;
    touchCalibration_x0 = touchCalibration_x1;
    touchCalibration_x1 = values[0];
    touchCalibration_invert_x = true;
  }
  touchCalibration_invert_y = false;
  if(touchCalibration_y0 > touchCalibration_y1){
    values[0]=touchCalibration_y0;
    touchCalibration_y0 = touchCalibration_y1;
    touchCalibration_y1 = values[0];
```

```
  }

  // pre calculate
  touchCalibration_x1 -= touchCalibration_x0;
  touchCalibration_y1 -= touchCalibration_y0;

  if(touchCalibration_x0 == 0) touchCalibration_x0 = 1;
  if(touchCalibration_x1 == 0) touchCalibration_x1 = 1;
  if(touchCalibration_y0 == 0) touchCalibration_y0 = 1;
  if(touchCalibration_y1 == 0) touchCalibration_y1 = 1;

  // export parameters, if pointer valid
  if(parameters != NULL){
    parameters[0] = touchCalibration_x0;
    parameters[1] = touchCalibration_x1;
    parameters[2] = touchCalibration_y0;
    parameters[3] = touchCalibration_y1;
    parameters[4] = touchCalibration_rotate | (touchCalibration_invert_x <<1) | (touchCalibration_invert_y <<2);
  }
}
```

```
/***************************************************************************************
** Function name:           setTouch
** Description:             imports calibration parameters for touchscreen.
***************************************************************************************/
void TFT_eSPI::setTouch(uint16_t *parameters){
  touchCalibration_x0 = parameters[0];
  touchCalibration_x1 = parameters[1];
  touchCalibration_y0 = parameters[2];
  touchCalibration_y1 = parameters[3];

  if(touchCalibration_x0 == 0) touchCalibration_x0 = 1;
  if(touchCalibration_x1 == 0) touchCalibration_x1 = 1;
  if(touchCalibration_y0 == 0) touchCalibration_y0 = 1;
  if(touchCalibration_y1 == 0) touchCalibration_y1 = 1;

  touchCalibration_rotate = parameters[4] & 0x01;
  touchCalibration_invert_x = parameters[4] & 0x02;
  touchCalibration_invert_y = parameters[4] & 0x04;
}
```

```
/***************************************************************************************
** Function name:           SMOTH FONTS
** Description:             This is part of the TFT_eSPI class and is associated with anti-aliased font functions
***************************************************************************************x*/
```

```
/***************************************************************************************
** Function name:           loadFont
** Description:             loads parameters from a font vlw array in memory
****************************************************************************************x*/
void TFT_eSPI::loadFont(const uint8_t array[])
{
  if (array == nullptr) return;
  fontPtr = (uint8_t*) array;
  loadFont("", false);
}

#ifdef FONT_FS_AVAILABLE
```

```
/**************************************************************************************
** Function name:           loadFont
** Description:             loads parameters from a font vlw file
**************************************************************************************x*/
void TFT_eSPI::loadFont(String fontName, fs::FS &ffs)
{
  fontFS = ffs;
  loadFont(fontName, false);
}
#endif
```

```
/***************************************************************************************
** Function name:           loadFont
** Description:             loads parameters from a font vlw file
***************************************************************************************x*/
void TFT_eSPI::loadFont(String fontName, bool flash)
{
  /*
    The vlw font format does not appear to be documented anywhere, so some reverse
    engineering has been applied!

    Header of vlw file comprises 6 uint32_t parameters (24 bytes total):
      1. The gCount (number of character glyphs)
      2. A version number (0xB = 11 for the one I am using)
      3. The font size (in points, not pixels)
      4. Deprecated mboxY parameter (typically set to 0)
      5. Ascent in pixels from baseline to top of "d"
      6. Descent in pixels from baseline to bottom of "p"

    Next are gCount sets of values for each glyph, each set comprises 7 int32t parameters (28 bytes):
      1. Glyph Unicode stored as a 32 bit value
      2. Height of bitmap bounding box
      3. Width of bitmap bounding box
      4. gxAdvance for cursor (setWidth in Processing)
      5. dY = distance from cursor baseline to top of glyph bitmap (signed value +ve = up)
      6. dX = distance from cursor to left side of glyph bitmap (signed value -ve = left)
      7. padding value, typically 0

    The bitmaps start next at 24 + (28 * gCount) bytes from the start of the file.
    Each pixel is 1 byte, an 8 bit Alpha value which represents the transparency from
    0xFF foreground colour, 0x00 background. The sketch uses a linear interpolation
    between the foreground and background RGB component colours. e.g.
        pixelRed = ((fgRed * alpha) + (bgRed * (255 - alpha))/255
    To gain a performance advantage fixed point arithmetic is used with rounding and
    division by 256 (shift right 8 bits is faster).

    After the bitmaps is:
       1 byte for font name string length (excludes null)
       a zero terminated character string giving the font name
       1 byte for Postscript name string length
       a zero/one terminated character string giving the font name
       last byte is 0 for non-anti-aliased and 1 for anti-aliased (smoothed)


    Glyph bitmap example is:
    // Cursor coordinate positions for this and next character are marked by 'C'
    // C<------- gxAdvance ------->C  gxAdvance is how far to move cursor for next glyph cursor position
    // |                       |
    // |                       |   ascent is top of "d", descent is bottom of "p"
    // +-- gdX --+           ascent
    // |       +-- gWidth--+     |   gdX is offset to left edge of glyph bitmap
    // |   +     x@.........@x  +  |   gdX may be negative e.g. italic "y" tail extending to left of
    // |   |     @@.........@@  |  |   cursor position, plot top left corner of bitmap at (cursorX + gdX)
    // |   |     @@.........@@ gdY |   gWidth and gHeight are glyph bitmap dimensions
    // |   |     .@@@.....@@@@  |  |
    // | gHeight ....@@@@@..@@  +  +    <-- baseline
    // |   |     ...........@@     |
    // |   |     ...........@@     |   gdY is the offset to the top edge of the bitmap
    // |   |     .@@.......@@. descent plot top edge of bitmap at (cursorY + yAdvance - gdY)
    // |   +     x..@@@@@@@..x     |   x marks the corner pixels of the bitmap
    // |                       |
    // +-------------------------+   yAdvance is y delta for the next line, font size or (ascent + descent)
    //                           some fonts can overlay in y direction so may need a user adjust value

  */

  if (fontLoaded) unloadFont();

#ifdef FONT_FS_AVAILABLE
  if (fontName == "") fs_font = false;
  else { fontPtr = nullptr; fs_font = true; }

  if (fs_font) {
    spiffs = flash; // true if font is in SPIFFS

    if(spiffs) fontFS = SPIFFS;

    // Avoid a crash on the ESP32 if the file does not exist
    if (fontFS.exists("/" + fontName + ".vlw") == false) {
      Serial.println("Font file " + fontName + " not found!");
      return;
    }

    fontFile = fontFS.open( "/" + fontName + ".vlw", "r");
```

```
    fontFile.seek(0, fs::SeekSet);
  }
#else
  // Avoid unused varaible warning
  fontName = fontName;
  flash = flash;
#endif

  gFont.gArray   = (const uint8_t*)fontPtr;

  gFont.gCount   = (uint16_t)readInt32(); // glyph count in file
                   readInt32(); // vlw encoder version - discard
  gFont.yAdvance = (uint16_t)readInt32(); // Font size in points, not pixels
                   readInt32(); // discard
  gFont.ascent   = (uint16_t)readInt32(); // top of "d"
  gFont.descent  = (uint16_t)readInt32(); // bottom of "p"

  // These next gFont values might be updated when the Metrics are fetched
  gFont.maxAscent  = gFont.ascent;   // Determined from metrics
  gFont.maxDescent = gFont.descent;  // Determined from metrics
  gFont.yAdvance   = gFont.ascent + gFont.descent;
  gFont.spaceWidth = gFont.yAdvance / 4;  // Guess at space width

  fontLoaded = true;

  // Fetch the metrics for each glyph
  loadMetrics();
}
```

```
/***************************************************************************************
** Function name:           loadMetrics
** Description:             Get the metrics for each glyph and store in RAM
***************************************************************************************x*/
//#define SHOW_ASCENT_DESCENT
void TFT_eSPI::loadMetrics(void)
{
  uint32_t headerPtr = 24;
  uint32_t bitmapPtr = headerPtr + gFont.gCount * 28;

#if defined (ESP32) && defined (CONFIG_SPIRAM_SUPPORT)
  if ( psramFound() )
  {
    gUnicode  = (uint16_t*)ps_malloc( gFont.gCount * 2); // Unicode 16 bit Basic Multilingual Plane (0-FFFF)
    gHeight   =  (uint8_t*)ps_malloc( gFont.gCount );    // Height of glyph
    gWidth    =  (uint8_t*)ps_malloc( gFont.gCount );    // Width of glyph
    gxAdvance =  (uint8_t*)ps_malloc( gFont.gCount );    // xAdvance - to move x cursor
    gdY       =  (int16_t*)ps_malloc( gFont.gCount * 2); // offset from bitmap top edge from lowest point in any char
    gdX       =   (int8_t*)ps_malloc( gFont.gCount );    // offset for bitmap left edge relative to cursor X
    gBitmap   = (uint32_t*)ps_malloc( gFont.gCount * 4); // seek pointer to glyph bitmap in the file
  }
  else
#endif
  {
    gUnicode  = (uint16_t*)malloc( gFont.gCount * 2); // Unicode 16 bit Basic Multilingual Plane (0-FFFF)
    gHeight   =  (uint8_t*)malloc( gFont.gCount );    // Height of glyph
    gWidth    =  (uint8_t*)malloc( gFont.gCount );    // Width of glyph
    gxAdvance =  (uint8_t*)malloc( gFont.gCount );    // xAdvance - to move x cursor
    gdY       =  (int16_t*)malloc( gFont.gCount * 2); // offset from bitmap top edge from lowest point in any charact
    gdX       =   (int8_t*)malloc( gFont.gCount );    // offset for bitmap left edge relative to cursor X
    gBitmap   = (uint32_t*)malloc( gFont.gCount * 4); // seek pointer to glyph bitmap in the file
  }

#ifdef SHOW_ASCENT_DESCENT
  Serial.print("ascent  = "); Serial.println(gFont.ascent);
  Serial.print("descent = "); Serial.println(gFont.descent);
#endif

#ifdef FONT_FS_AVAILABLE
  if (fs_font) fontFile.seek(headerPtr, fs::SeekSet);
#endif

  uint16_t gNum = 0;

  while (gNum < gFont.gCount)
  {
    gUnicode[gNum]  = (uint16_t)readInt32(); // Unicode code point value
    gHeight[gNum]   =  (uint8_t)readInt32(); // Height of glyph
    gWidth[gNum]    =  (uint8_t)readInt32(); // Width of glyph
    gxAdvance[gNum] =  (uint8_t)readInt32(); // xAdvance - to move x cursor
    gdY[gNum]       =  (int16_t)readInt32(); // y delta from baseline
    gdX[gNum]       =   (int8_t)readInt32(); // x delta from cursor
    readInt32(); // ignored

    //Serial.print("Unicode = 0x"); Serial.print(gUnicode[gNum], HEX); Serial.print(", gHeight  = "); Serial.println(gH
    //Serial.print("Unicode = 0x"); Serial.print(gUnicode[gNum], HEX); Serial.print(", gWidth  = "); Serial.println(gW
    //Serial.print("Unicode = 0x"); Serial.print(gUnicode[gNum], HEX); Serial.print(", gxAdvance  = "); Serial.println
    //Serial.print("Unicode = 0x"); Serial.print(gUnicode[gNum], HEX); Serial.print(", gdY  = "); Serial.println(gdY[gN

    // Different glyph sets have different ascent values not always based on "d", so we could get
    // the maximum glyph ascent by checking all characters. BUT this method can generate bad values
    // for non-existant glyphs, so we will reply on processing for the value and disable this code for now...
    /*
    if (gdY[gNum] > gFont.maxAscent)
    {
      // Try to avoid UTF coding values and characters that tend to give duff values
      if (((gUnicode[gNum] > 0x20) && (gUnicode[gNum] < 0x7F)) || (gUnicode[gNum] > 0xA0))
      {
        gFont.maxAscent   = gdY[gNum];
#ifdef SHOW_ASCENT_DESCENT
        Serial.print("Unicode = 0x"); Serial.print(gUnicode[gNum], HEX); Serial.print(", maxAscent  = "); Serial.println
#endif
      }
    }
    */

    // Different glyph sets have different descent values not always based on "p", so get maximum glyph descent
    if (((int16_t)gHeight[gNum] - (int16_t)gdY[gNum]) > gFont.maxDescent)
    {
      // Avoid UTF coding values and characters that tend to give duff values
      if (((gUnicode[gNum] > 0x20) && (gUnicode[gNum] < 0xA0) && (gUnicode[gNum] != 0x7F)) || (gUnicode[gNu
      {
        gFont.maxDescent   = gHeight[gNum] - gdY[gNum];
#ifdef SHOW_ASCENT_DESCENT
```

```
#endif
    }
   }

   gBitmap[gNum] = bitmapPtr;

   bitmapPtr += gWidth[gNum] * gHeight[gNum];

   gNum++;
   yield();
  }

  gFont.yAdvance = gFont.maxAscent + gFont.maxDescent;

  gFont.spaceWidth = (gFont.ascent + gFont.descent) * 2/7;  // Guess at space width
}
```

```
/**************************************************************************************
** Function name:           deleteMetrics
** Description:             Delete the old glyph metrics and free up the memory
**************************************************************************************x*/
void TFT_eSPI::unloadFont( void )
{
  if (gUnicode)
  {
    free(gUnicode);
    gUnicode = NULL;
  }

  if (gHeight)
  {
    free(gHeight);
    gHeight = NULL;
  }

  if (gWidth)
  {
    free(gWidth);
    gWidth = NULL;
  }

  if (gxAdvance)
  {
    free(gxAdvance);
    gxAdvance = NULL;
  }

  if (gdY)
  {
    free(gdY);
    gdY = NULL;
  }

  if (gdX)
  {
    free(gdX);
    gdX = NULL;
  }

  if (gBitmap)
  {
    free(gBitmap);
    gBitmap = NULL;
  }

  gFont.gArray = nullptr;

#ifdef FONT_FS_AVAILABLE
  if (fs_font && fontFile) fontFile.close();
#endif

  fontLoaded = false;
}
```

```
/***************************************************************************************
** Function name:           readInt32
** Description:             Get a 32 bit integer from the font file
***************************************************************************************x*/
uint32_t TFT_eSPI::readInt32(void)
{
  uint32_t val = 0;

#ifdef FONT_FS_AVAILABLE
  if (fs_font) {
    val |= fontFile.read() << 24;
    val |= fontFile.read() << 16;
    val |= fontFile.read() << 8;
    val |= fontFile.read();
  }
  else
#endif
  {
    val |= pgm_read_byte(fontPtr++) << 24;
    val |= pgm_read_byte(fontPtr++) << 16;
    val |= pgm_read_byte(fontPtr++) << 8;
    val |= pgm_read_byte(fontPtr++);
  }

  return val;
}
```

```
/***************************************************************************************
** Function name:           getUnicodeIndex
** Description:             Get the font file index of a Unicode character
***************************************************************************************x*/
bool TFT_eSPI::getUnicodeIndex(uint16_t unicode, uint16_t *index)
{
  for (uint16_t i = 0; i < gFont.gCount; i++)
  {
    if (gUnicode[i] == unicode)
    {
      *index = i;
      return true;
    }
  }
  return false;
}
```

```
/***************************************************************************************
** Function name:           drawGlyph
** Description:             Write a character to the TFT cursor position
***************************************************************************************x*/
// Expects file to be open
void TFT_eSPI::drawGlyph(uint16_t code)
{
  uint16_t fg = textcolor;
  uint16_t bg = textbgcolor;

  if (code < 0x21)
  {
    if (code == 0x20) {
      //if (fg!=bg) fillRect(cursor_x, cursor_y, gFont.spaceWidth, gFont.yAdvance, bg);
      cursor_x += gFont.spaceWidth;
      return;
    }

    if (code == '\n') {
      cursor_x = 0;
      cursor_y += gFont.yAdvance;
      if (textwrapY && (cursor_y >= height())) cursor_y = 0;
      return;
    }
  }

  uint16_t gNum = 0;
  bool found = getUnicodeIndex(code, &gNum);

  if (found)
  {

    if (textwrapX && (cursor_x + gWidth[gNum] + gdX[gNum] > width()))
    {
      cursor_y += gFont.yAdvance;
      cursor_x = 0;
    }
    if (textwrapY && ((cursor_y + gFont.yAdvance) >= height())) cursor_y = 0;
    if (cursor_x == 0) cursor_x -= gdX[gNum];

    uint8_t* pbuffer = nullptr;
    const uint8_t* gPtr = (const uint8_t*) gFont.gArray;

#ifdef FONT_FS_AVAILABLE
    if (fs_font)
    {
      fontFile.seek(gBitmap[gNum], fs::SeekSet); // This is taking >30ms for a significant position shift
      pbuffer =  (uint8_t*)malloc(gWidth[gNum]);
    }
#endif

    int16_t cy = cursor_y + gFont.maxAscent - gdY[gNum];
    int16_t cx = cursor_x + gdX[gNum];

    int16_t  xs = cx;
    uint32_t dl = 0;
    uint8_t pixel;

    startWrite(); // Avoid slow ESP32 transaction overhead for every pixel

    //if (fg!=bg) fillRect(cursor_x, cursor_y, gxAdvance[gNum], gFont.yAdvance, bg);

    for (int y = 0; y < gHeight[gNum]; y++)
    {
#ifdef FONT_FS_AVAILABLE
      if (fs_font) {
        if (spiffs)
        {
          fontFile.read(pbuffer, gWidth[gNum]);
          //Serial.println("SPIFFS");
        }
        else
        {
          endWrite();    // Release SPI for SD card transaction
          fontFile.read(pbuffer, gWidth[gNum]);
          startWrite();  // Re-start SPI for TFT transaction
          //Serial.println("Not SPIFFS");
        }
      }
#endif
      for (int x = 0; x < gWidth[gNum]; x++)
      {
#ifdef FONT_FS_AVAILABLE
        if (fs_font) pixel = pbuffer[x];
```

```
#endif
      pixel = pgm_read_byte(gPtr + gBitmap[gNum] + x + gWidth[gNum] * y);

      if (pixel)
      {
        if (pixel != 0xFF)
        {
          if (dl) {
            if (dl==1) drawPixel(xs, y + cy, fg);
            else drawFastHLine( xs, y + cy, dl, fg);
            dl = 0;
          }
          if (getColor) bg = getColor(x + cx, y + cy);
          drawPixel(x + cx, y + cy, alphaBlend(pixel, fg, bg));
        }
        else
        {
          if (dl==0) xs = x + cx;
          dl++;
        }
      }
      else
      {
        if (dl) { drawFastHLine( xs, y + cy, dl, fg); dl = 0; }
      }
    }
    if (dl) { drawFastHLine( xs, y + cy, dl, fg); dl = 0; }
  }

  if (pbuffer) free(pbuffer);
  cursor_x += gxAdvance[gNum];
  endWrite();
 }
 else
 {
  // Not a Unicode in font so draw a rectangle and move on cursor
  drawRect(cursor_x, cursor_y + gFont.maxAscent - gFont.ascent, gFont.spaceWidth, gFont.ascent, fg);
  cursor_x += gFont.spaceWidth + 1;
 }
}
```

```
/***************************************************************************************
** Function name:           showFont
** Description:             Page through all characters in font, td ms between screens
***************************************************************************************x*/
void TFT_eSPI::showFont(uint32_t td)
{
  if(!fontLoaded) return;

  int16_t cursorX = width(); // Force start of new page to initialise cursor
  int16_t cursorY = height();// for the first character
  uint32_t timeDelay = 0;    // No delay before first page

  fillScreen(textbgcolor);

  for (uint16_t i = 0; i < gFont.gCount; i++)
  {
    // Check if this will need a new screen
    if (cursorX + gdX[i] + gWidth[i] >= width())  {
      cursorX = -gdX[i];

      cursorY += gFont.yAdvance;
      if (cursorY + gFont.maxAscent + gFont.descent >= height()) {
        cursorX = -gdX[i];
        cursorY = 0;
        delay(timeDelay);
        timeDelay = td;
        fillScreen(textbgcolor);
      }
    }

    setCursor(cursorX, cursorY);
    drawGlyph(gUnicode[i]);
    cursorX += gxAdvance[i];
    //cursorX +=  printToSprite( cursorX, cursorY, i );
    yield();
  }

  delay(timeDelay);
  fillScreen(textbgcolor);
  //fontFile.close();

}
```

```
/***************************************************************************************
** Function name:           BUTTON
** Description:             Code for the GFX button UI element
***************************************************************************************/
TFT_eSPI_Button::TFT_eSPI_Button(void) {
  _gfx       = nullptr;
  _xd        = 0;
  _yd        = 0;
  _textdatum = MC_DATUM;
  _label[9]  = '\0';
}
```

```
/***************************************************************************************
** Function name:           initButton
** Description:             Classic initButton() function: pass center & size
***************************************************************************************/
void TFT_eSPI_Button::initButton(
 TFT_eSPI *gfx, int16_t x, int16_t y, uint16_t w, uint16_t h,
 uint16_t outline, uint16_t fill, uint16_t textcolor,
 char *label, uint8_t textsize)
{
  // Tweak arguments and pass to the newer initButtonUL() function...
  initButtonUL(gfx, x - (w / 2), y - (h / 2), w, h, outline, fill,
    textcolor, label, textsize);
}
```

```
/***************************************************************************************
** Function name:           initButtonUL
** Description:             Newer function instead accepts upper-left corner & size
***************************************************************************************/
void TFT_eSPI_Button::initButtonUL(
 TFT_eSPI *gfx, int16_t x1, int16_t y1, uint16_t w, uint16_t h,
 uint16_t outline, uint16_t fill, uint16_t textcolor,
 char *label, uint8_t textsize)
{
  _x1          = x1;
  _y1          = y1;
  _w           = w;
  _h           = h;
  _outlinecolor = outline;
  _fillcolor    = fill;
  _textcolor    = textcolor;
  _textsize     = textsize;
  _gfx          = gfx;
  strncpy(_label, label, 9);
}
```

```
/***************************************************************************************
** Function name:           setLabelDatum
** Description:             Adjust text datum and x, y deltas
***************************************************************************************/
void TFT_eSPI_Button::setLabelDatum(int16_t x_delta, int16_t y_delta, uint8_t datum)
{
  _xd       = x_delta;
  _yd       = y_delta;
  _textdatum = datum;
}
```

```
/***************************************************************************************
** Function name:           drawButton
** Description:             Draw Button on screen
***************************************************************************************/
void TFT_eSPI_Button::drawButton(bool inverted, String long_name) {
  uint16_t fill, outline, text;

  if(!inverted) {
    fill    = _fillcolor;
    outline = _outlinecolor;
    text    = _textcolor;
  } else {
    fill    = _textcolor;
    outline = _outlinecolor;
    text    = _fillcolor;
  }

  uint8_t r = min(_w, _h) / 4; // Corner radius
  _gfx->fillRoundRect(_x1, _y1, _w, _h, r, fill);
  _gfx->drawRoundRect(_x1, _y1, _w, _h, r, outline);

  _gfx->setTextColor(text, fill);
  _gfx->setTextSize(_textsize);

  uint8_t tempdatum = _gfx->getTextDatum();
  _gfx->setTextDatum(_textdatum);
  uint16_t tempPadding = _gfx->getTextPadding();
  _gfx->setTextPadding(0);

  if (long_name == "")
    _gfx->drawString(_label, _x1 + (_w/2) + _xd, _y1 + (_h/2) - 4 + _yd);
  else
    _gfx->drawString(long_name, _x1 + (_w/2) + _xd, _y1 + (_h/2) - 4 + _yd);

  _gfx->setTextDatum(tempdatum);
  _gfx->setTextPadding(tempPadding);
}

bool TFT_eSPI_Button::contains(int16_t x, int16_t y) {
  return ((x >= _x1) && (x < (_x1 + _w)) &&
          (y >= _y1) && (y < (_y1 + _h)));
}

void TFT_eSPI_Button::press(bool p) {
  laststate = currstate;
  currstate = p;
}
```

```
/**************************************************************************************************
** Function name:           isPressed()
** Description:             return (currstate)
**************************************************************************************************/
bool TFT_eSPI_Button::isPressed()    { return currstate; }
```

```
/**************************************************************************************
** Function name:           justPressed()
** Description:             return (currstate && !laststate)
**************************************************************************************/
bool TFT_eSPI_Button::justPressed()  { return (currstate && !laststate); }
```

```
/**************************************************************************************
** Function name:           justReleased()
** Description:             return (currstate && !laststate)
**************************************************************************************/
bool TFT_eSPI_Button::justReleased() { return (!currstate && laststate); }
```

```
/***************************************************************************************
** Function name:           TFT_eSprite
** Description:             Class constructor
***************************************************************************************/
// The following class creates Sprites in RAM, graphics can then be drawn in the Sprite
// and rendered quickly onto the TFT screen. The class inherits the graphics functions
// from the TFT_eSPI class. Some functions are overridden by this class so that the
// graphics are written to the Sprite rather than the TFT.
// Coded by Bodmer, see license file in root folder

// Color bytes are swapped when writing to RAM, this introduces a small overhead but
// there is a nett performance gain by using swapped bytes.

TFT_eSprite::TFT_eSprite(TFT_eSPI *tft)
{
  _tft = tft;     // Pointer to tft class so we can call member functions

  _iwidth    = 0; // Initialise width and height to 0 (it does not exist yet)
  _iheight   = 0;
  _bpp = 16;
  _swapBytes = false;   // Do not swap pushImage colour bytes by default

  _created = false;
  _vpOoB   = true;

  _xs = 0;  // window bounds for pushColor
  _ys = 0;
  _xe = 0;
  _ye = 0;

  _xptr = 0; // pushColor coordinate
  _yptr = 0;

  _colorMap = nullptr;

  _psram_enable = true;
}
```

```
/***************************************************************************************
** Function name:           createSprite
** Description:             Create a sprite (bitmap) of defined width and height
***************************************************************************************/
// cast returned value to (uint8_t*) for 8 bit or (uint16_t*) for 16 bit colours
void* TFT_eSprite::createSprite(int16_t w, int16_t h, uint8_t frames)
{

  if ( _created ) return _img8_1;

  if ( w < 1 || h < 1 ) return nullptr;

  _iwidth  = _dwidth  = _bitwidth = w;
  _iheight = _dheight = h;

  cursor_x = 0;
  cursor_y = 0;

  // Default scroll rectangle and gap fill colour
  _sx = 0;
  _sy = 0;
  _sw = w;
  _sh = h;
  _scolor = TFT_BLACK;

  _img8   = (uint8_t*) callocSprite(w, h, frames);
  _img8_1 = _img8;
  _img8_2 = _img8;
  _img    = (uint16_t*) _img8;
  _img4   = _img8;

  if ( (_bpp == 16) && (frames > 1) ) {
    _img8_2 = _img8 + (w * h * 2 + 1);
  }

  // ESP32 only 16bpp check
  //if (esp_ptr_dma_capable(_img8_1)) Serial.println("DMA capable Sprite pointer _img8_1");
  //else Serial.println("Not a DMA capable Sprite pointer _img8_1");
  //if (esp_ptr_dma_capable(_img8_2)) Serial.println("DMA capable Sprite pointer _img8_2");
  //else Serial.println("Not a DMA capable Sprite pointer _img8_2");

  if ( (_bpp == 8) && (frames > 1) ) {
    _img8_2 = _img8 + (w * h + 1);
  }

  if ( (_bpp == 4) && (_colorMap == nullptr)) createPalette(default_4bit_palette);

  // This is to make it clear what pointer size is expected to be used
  // but casting in the user sketch is needed due to the use of void*
  if ( (_bpp == 1) && (frames > 1) )
  {
    w = (w+7) & 0xFFF8;
    _img8_2 = _img8 + ( (w>>3) * h + 1 );
  }

  if (_img8)
  {
    _created = true;
    rotation = 0;
    setViewport(0, 0, _dwidth, _dheight);
    setPivot(_iwidth/2, _iheight/2);
    return _img8_1;
  }

  return nullptr;
}
```

```
/**************************************************************************************
** Function name:           getPointer
** Description:             Returns pointer to start of sprite memory area
**************************************************************************************/
void* TFT_eSprite::getPointer(void)
{
  if (!_created) return nullptr;
  return _img8_1;
}
```

```
/**************************************************************************************
** Function name:           created
** Description:             Returns true if sprite has been created
**************************************************************************************/
bool TFT_eSprite::created(void)
{
  return _created;
}
```

```
/***************************************************************************************
** Function name:           ~TFT_eSprite
** Description:             Class destructor
***************************************************************************************/
TFT_eSprite::~TFT_eSprite(void)
{
  deleteSprite();

#ifdef SMOOTH_FONT
  if(fontLoaded) unloadFont();
#endif
}
```

```
/***************************************************************************************
** Function name:           callocSprite
** Description:             Allocate a memory area for the Sprite and return pointer
***************************************************************************************/
void* TFT_eSprite::callocSprite(int16_t w, int16_t h, uint8_t frames)
{
  // Add one extra "off screen" pixel to point out-of-bounds setWindow() coordinates
  // this means push/writeColor functions do not need additional bounds checks and
  // hence will run faster in normal circumstances.
  uint8_t* ptr8 = nullptr;

  if (frames > 2) frames = 2; // Currently restricted to 2 frame buffers
  if (frames < 1) frames = 1;

  if (_bpp == 16)
  {
#if defined (ESP32) && defined (CONFIG_SPIRAM_SUPPORT)
    if ( psramFound() && _psram_enable && !_tft->DMA_Enabled)
    {
      ptr8 = ( uint8_t*) ps_calloc(frames * w * h + frames, sizeof(uint16_t));
      //Serial.println("PSRAM");
    }
    else
#endif
    {
      ptr8 = ( uint8_t*) calloc(frames * w * h + frames, sizeof(uint16_t));
      //Serial.println("Normal RAM");
    }
  }

  else if (_bpp == 8)
  {
#if defined (ESP32) && defined (CONFIG_SPIRAM_SUPPORT)
    if ( psramFound() && _psram_enable ) ptr8 = ( uint8_t*) ps_calloc(frames * w * h + frames, sizeof(uint8_t));
    else
#endif
    ptr8 = ( uint8_t*) calloc(frames * w * h + frames, sizeof(uint8_t));
  }

  else if (_bpp == 4)
  {
    w = (w+1) & 0xFFFE; // width needs to be multiple of 2, with an extra "off screen" pixel
    _iwidth = w;
#if defined (ESP32) && defined (CONFIG_SPIRAM_SUPPORT)
    if ( psramFound() && _psram_enable ) ptr8 = ( uint8_t*) ps_calloc(((frames * w * h) >> 1) + frames, sizeof(uin
    else
#endif
    ptr8 = ( uint8_t*) calloc(((frames * w * h) >> 1) + frames, sizeof(uint8_t));
  }

  else // Must be 1 bpp
  {
    //_dwidth   Display width+height in pixels always in rotation 0 orientation
    //_dheight  Not swapped for sprite rotations
    // Note: for 1bpp _iwidth and _iheight are swapped during Sprite rotations

    w =  (w+7) & 0xFFF8; // width should be the multiple of 8 bits to be compatible with epdpaint
    _iwidth = w;         // _iwidth is rounded up to be multiple of 8, so might not be = _dwidth
    _bitwidth = w;       // _bitwidth will not be rotated whereas _iwidth may be

#if defined (ESP32) && defined (CONFIG_SPIRAM_SUPPORT)
    if ( psramFound() && _psram_enable ) ptr8 = ( uint8_t*) ps_calloc(frames * (w>>3) * h + frames, sizeof(uint8_
    else
#endif
    ptr8 = ( uint8_t*) calloc(frames * (w>>3) * h + frames, sizeof(uint8_t));
  }

  return ptr8;
}
```

```
/***************************************************************************************
** Function name:           createPalette (from RAM array)
** Description:             Set a palette for a 4-bit per pixel sprite
***************************************************************************************/
void TFT_eSprite::createPalette(uint16_t colorMap[], uint8_t colors)
{
  if (_colorMap != nullptr)
  {
    free(_colorMap);
  }

  if (colorMap == nullptr)
  {
    // Create a color map using the default FLASH map
    createPalette(default_4bit_palette);
    return;
  }

  // Allocate and clear memory for 16 color map
  _colorMap = (uint16_t *)calloc(16, sizeof(uint16_t));

  if (colors > 16) colors = 16;

  // Copy map colors
  for (uint8_t i = 0; i < colors; i++)
  {
    _colorMap[i] = colorMap[i];
  }
}
```

```
/**************************************************************************************
** Function name:           createPalette (from FLASH array)
** Description:             Set a palette for a 4-bit per pixel sprite
**************************************************************************************/
void TFT_eSprite::createPalette(const uint16_t colorMap[], uint8_t colors)
{
  if (colorMap == nullptr)
  {
    // Create a color map using the default FLASH map
    colorMap = default_4bit_palette;
  }

  // Allocate and clear memory for 16 color map
  _colorMap = (uint16_t *)calloc(16, sizeof(uint16_t));

  if (colors > 16) colors = 16;

  // Copy map colors
  for (uint8_t i = 0; i < colors; i++)
  {
    _colorMap[i] = pgm_read_word(colorMap++);
  }
}
```

```
/**************************************************************************************
** Function name:           frameBuffer
** Description:             For 1 bpp Sprites, select the frame used for graphics
**************************************************************************************/
// Frames are numbered 1 and 2
void* TFT_eSprite::frameBuffer(int8_t f)
{
  if (!_created) return nullptr;

  if ( f == 2 ) _img8 = _img8_2;
  else          _img8 = _img8_1;

  if (_bpp == 16) _img = (uint16_t*)_img8;

  //if (_bpp == 8) _img8 = _img8;

  if (_bpp == 4) _img4 = _img8;

  return _img8;
}
```

```
/***************************************************************************************
** Function name:           setColorDepth
** Description:             Set bits per pixel for colour (1, 8 or 16)
***************************************************************************************/
void* TFT_eSprite::setColorDepth(int8_t b)
{
  // Do not re-create the sprite if the colour depth does not change
  if (_bpp == b) return _img8_1;

  // Validate the new colour depth
  if ( b > 8 ) _bpp = 16;  // Bytes per pixel
  else if ( b > 4 ) _bpp = 8;
  else if ( b > 1 ) _bpp = 4;
  else _bpp = 1;

  // Can't change an existing sprite's colour depth so delete it
  if (_created) free(_img8_1);

  // If it existed, re-create the sprite with the new colour depth
  if (_created)
  {
    _created = false;
    return createSprite(_dwidth, _dheight);
  }

  return nullptr;
}
```

```
/***************************************************************************************
** Function name:           getColorDepth
** Description:             Get bits per pixel for colour (1, 8 or 16)
***************************************************************************************/
int8_t TFT_eSprite::getColorDepth(void)
{
  if (_created) return _bpp;
  else return 0;
}
```

```
/**************************************************************************************
** Function name:           setBitmapColor
** Description:             Set the 1bpp foreground foreground and background colour
**************************************************************************************/
void TFT_eSprite::setBitmapColor(uint16_t c, uint16_t b)
{
  if (c == b) b = ~c;
  _tft->bitmap_fg = c;
  _tft->bitmap_bg = b;
}
```

```
/**************************************************************************************
** Function name:           setPaletteColor
** Description:             Set the 4bpp palette color at the given index
**************************************************************************************/
void TFT_eSprite::setPaletteColor(uint8_t index, uint16_t color)
{
  if (_colorMap == nullptr || index > 15) return; // out of bounds

  _colorMap[index] = color;
}
```

```
/**************************************************************************************
** Function name:           getPaletteColor
** Description:             Return the palette color at 4bpp index, or 0 on error.
**************************************************************************************/
uint16_t TFT_eSprite::getPaletteColor(uint8_t index)
{
  if (_colorMap == nullptr || index > 15) return 0; // out of bounds

  return _colorMap[index];
}
```

```
/**************************************************************************************
** Function name:           deleteSprite
** Description:             Delete the sprite to free up memory (RAM)
**************************************************************************************/
void TFT_eSprite::deleteSprite(void)
{
  if (_colorMap != nullptr)
  {
    free(_colorMap);
□_colorMap = nullptr;
  }

  if (_created)
  {
    free(_img8_1);
    _img8 = nullptr;
    _created = false;
    _vpOoB   = true;  // TFT_eSPI class write() uses this to check for valid sprite
  }
}
```

```
/***************************************************************************************
** Function name:           pushRotated - Fast fixed point integer maths version
** Description:             Push rotated Sprite to TFT screen
***************************************************************************************/
#define FP_SCALE 10
bool TFT_eSprite::pushRotated(int16_t angle, uint32_t transp)
{
  if ( !_created || _tft->_vpOoB) return false;

  // Bounding box parameters
  int16_t min_x;
  int16_t min_y;
  int16_t max_x;
  int16_t max_y;

  // Get the bounding box of this rotated source Sprite relative to Sprite pivot
  if ( !getRotatedBounds(angle, &min_x, &min_y, &max_x, &max_y) ) return false;

  uint16_t sline_buffer[max_x - min_x + 1];

  int32_t xt = min_x - _tft->_xPivot;
  int32_t yt = min_y - _tft->_yPivot;
  uint32_t xe = _dwidth << FP_SCALE;
  uint32_t ye = _dheight << FP_SCALE;
  uint32_t tpcolor = transp;

  if (transp != 0x00FFFFFF) {
    if (_bpp == 4) tpcolor = _colorMap[transp & 0x0F];
    tpcolor = tpcolor>>8 | tpcolor<<8; // Working with swapped color bytes
  }
    _tft->startWrite(); // Avoid transaction overhead for every tft pixel

  // Scan destination bounding box and fetch transformed pixels from source Sprite
  for (int32_t y = min_y; y <= max_y; y++, yt++) {
    int32_t x = min_x;
    uint32_t xs = (_cosra * xt - (_sinra * yt - (_xPivot << FP_SCALE)) + (1 << (FP_SCALE - 1)));
    uint32_t ys = (_sinra * xt + (_cosra * yt + (_yPivot << FP_SCALE)) + (1 << (FP_SCALE - 1)));

    while ((xs >= xe || ys >= ye) && x < max_x) { x++; xs += _cosra; ys += _sinra; }
    if (x == max_x) continue;

    uint32_t pixel_count = 0;
    do {
      uint32_t rp;
      int32_t xp = xs >> FP_SCALE;
      int32_t yp = ys >> FP_SCALE;
      if (_bpp == 16) {rp = _img[xp + yp * _iwidth]; }
      else { rp = readPixel(xp, yp); rp = (uint16_t)(rp>>8 | rp<<8); }
      if (tpcolor == rp) {
        if (pixel_count) {
          // TFT window is already clipped, so this is faster than pushImage()
          _tft->setWindow(x - pixel_count, y, x, y);
          _tft->pushPixels(sline_buffer, pixel_count);
          pixel_count = 0;
        }
      }
      else {
        sline_buffer[pixel_count++] = rp;
      }
    } while (++x < max_x && (xs += _cosra) < xe && (ys += _sinra) < ye);
    if (pixel_count) {
      // TFT window is already clipped, so this is faster than pushImage()
      _tft->setWindow(x - pixel_count, y, x, y);
      _tft->pushPixels(sline_buffer, pixel_count);
    }
  }

  _tft->endWrite(); // End transaction

  return true;
}
```

```
/***************************************************************************************
** Function name:           pushRotated - Fast fixed point integer maths version
** Description:             Push a rotated copy of the Sprite to another Sprite
***************************************************************************************/
// Not compatible with 4bpp
bool TFT_eSprite::pushRotated(TFT_eSprite *spr, int16_t angle, uint32_t transp)
{
  if ( !_created  || _bpp == 4) return false; // Check this Sprite is created
  if ( !spr->_created  || spr->_bpp == 4) return false;  // Ckeck destination Sprite is created

  // Bounding box parameters
  int16_t min_x;
  int16_t min_y;
  int16_t max_x;
  int16_t max_y;

  // Get the bounding box of this rotated source Sprite
  if ( !getRotatedBounds(spr, angle, &min_x, &min_y, &max_x, &max_y) ) return false;

  uint16_t sline_buffer[max_x - min_x + 1];

  int32_t xt = min_x - spr->_xPivot;
  int32_t yt = min_y - spr->_yPivot;
  uint32_t xe = _dwidth << FP_SCALE;
  uint32_t ye = _dheight << FP_SCALE;
  uint32_t tpcolor = transp;

  if (transp != 0x00FFFFFF) {
    if (_bpp == 4) tpcolor = _colorMap[transp & 0x0F];
    tpcolor = tpcolor>>8 | tpcolor<<8; // Working with swapped color bytes
  }

  bool oldSwapBytes = spr->getSwapBytes();
  spr->setSwapBytes(false);

  // Scan destination bounding box and fetch transformed pixels from source Sprite
  for (int32_t y = min_y; y <= max_y; y++, yt++) {
    int32_t x = min_x;
    uint32_t xs = (_cosra * xt - (_sinra * yt - (_xPivot << FP_SCALE)) + (1 << (FP_SCALE - 1)));
    uint32_t ys = (_sinra * xt + (_cosra * yt + (_yPivot << FP_SCALE)) + (1 << (FP_SCALE - 1)));

    while ((xs >= xe || ys >= ye) && x < max_x) { x++; xs += _cosra; ys += _sinra; }
    if (x == max_x) continue;

    uint32_t pixel_count = 0;
    do {
      uint32_t rp;
      int32_t xp = xs >> FP_SCALE;
      int32_t yp = ys >> FP_SCALE;
      if (_bpp == 16) rp = _img[xp + yp * _iwidth];
      else { rp = readPixel(xp, yp); rp = (uint16_t)(rp>>8 | rp<<8); }
      if (tpcolor == rp) {
        if (pixel_count) {
          spr->pushImage(x - pixel_count, y, pixel_count, 1, sline_buffer);
          pixel_count = 0;
        }
      }
      else {
        sline_buffer[pixel_count++] = rp;
      }
    } while (++x < max_x && (xs += _cosra) < xe && (ys += _sinra) < ye);
    if (pixel_count) spr->pushImage(x - pixel_count, y, pixel_count, 1, sline_buffer);
  }
  spr->setSwapBytes(oldSwapBytes);
  return true;
}
```

```
/***************************************************************************************
** Function name:           getRotatedBounds
** Description:             Get TFT bounding box of a rotated Sprite wrt pivot
***************************************************************************************/
bool TFT_eSprite::getRotatedBounds(int16_t angle, int16_t *min_x, int16_t *min_y,
                                   int16_t *max_x, int16_t *max_y)
{
  // Get the bounding box of this rotated source Sprite relative to Sprite pivot
  getRotatedBounds(angle, width(), height(), _xPivot, _yPivot, min_x, min_y, max_x, max_y);

  // Move bounding box so source Sprite pivot coincides with TFT pivot
  *min_x += _tft->_xPivot;
  *max_x += _tft->_xPivot;
  *min_y += _tft->_yPivot;
  *max_y += _tft->_yPivot;

  // Return if bounding box is outside of TFT viewport
  if (*min_x > _tft->_vpW) return false;
  if (*min_y > _tft->_vpH) return false;
  if (*max_x < _tft->_vpX) return false;
  if (*max_y < _tft->_vpY) return false;

  // Clip bounding box to be within TFT viewport
  if (*min_x < _tft->_vpX) *min_x = _tft->_vpX;
  if (*min_y < _tft->_vpY) *min_y = _tft->_vpY;
  if (*max_x > _tft->_vpW) *max_x = _tft->_vpW;
  if (*max_y > _tft->_vpH) *max_y = _tft->_vpH;

  return true;
}
```

```
/**************************************************************************************
** Function name:           getRotatedBounds
** Description:             Get destination Sprite bounding box of a rotated Sprite wrt pivot
**************************************************************************************/
bool TFT_eSprite::getRotatedBounds(TFT_eSprite *spr, int16_t angle, int16_t *min_x, int16_t *min_y,
                                                     int16_t *max_x, int16_t *max_y)
{
  // Get the bounding box of this rotated source Sprite relative to Sprite pivot
  getRotatedBounds(angle, width(), height(), _xPivot, _yPivot, min_x, min_y, max_x, max_y);

  // Move bounding box so source Sprite pivot coincides with destination Sprite pivot
  *min_x += spr->_xPivot;
  *max_x += spr->_xPivot;
  *min_y += spr->_yPivot;
  *max_y += spr->_yPivot;

  // Test only to show bounding box
  // spr->fillSprite(TFT_BLACK);
  // spr->drawRect(min_x, min_y, max_x - min_x + 1, max_y - min_y + 1, TFT_GREEN);

  // Return if bounding box is completely outside of destination Sprite
  if (*min_x > spr->width()) return true;
  if (*min_y > spr->height()) return true;
  if (*max_x < 0) return true;
  if (*max_y < 0) return true;

  // Clip bounding box to Sprite boundaries
  // Clipping to a viewport will be done by destination Sprite pushImage function
  if (*min_x < 0) min_x = 0;
  if (*min_y < 0) min_y = 0;
  if (*max_x > spr->width())  *max_x = spr->width();
  if (*max_y > spr->height()) *max_y = spr->height();

  return true;
}
```

```
/**************************************************************************************
** Function name:           rotatedBounds
** Description:             Get bounding box of a rotated Sprite wrt pivot
**************************************************************************************/
void TFT_eSprite::getRotatedBounds(int16_t angle, int16_t w, int16_t h, int16_t xp, int16_t yp,
                                   int16_t *min_x, int16_t *min_y, int16_t *max_x, int16_t *max_y)
{
  // Trig values for the rotation
  float radAngle = -angle * 0.0174532925; // Convert degrees to radians
  float sina = sin(radAngle);
  float cosa = cos(radAngle);

  w -= xp; // w is now right edge coordinate relative to xp
  h -= yp; // h is now bottom edge coordinate relative to yp

  // Calculate new corner coordinates
  int16_t x0 = -xp * cosa - yp * sina;
  int16_t y0 =  xp * sina - yp * cosa;

  int16_t x1 =  w * cosa - yp * sina;
  int16_t y1 = -w * sina - yp * cosa;

  int16_t x2 =  h * sina + w * cosa;
  int16_t y2 =  h * cosa - w * sina;

  int16_t x3 =  h * sina - xp * cosa;
  int16_t y3 =  h * cosa + xp * sina;

  // Find bounding box extremes, enlarge box to accomodate rounding errors
  *min_x = x0-2;
  if (x1 < *min_x) *min_x = x1-2;
  if (x2 < *min_x) *min_x = x2-2;
  if (x3 < *min_x) *min_x = x3-2;

  *max_x = x0+2;
  if (x1 > *max_x) *max_x = x1+2;
  if (x2 > *max_x) *max_x = x2+2;
  if (x3 > *max_x) *max_x = x3+2;

  *min_y = y0-2;
  if (y1 < *min_y) *min_y = y1-2;
  if (y2 < *min_y) *min_y = y2-2;
  if (y3 < *min_y) *min_y = y3-2;

  *max_y = y0+2;
  if (y1 > *max_y) *max_y = y1+2;
  if (y2 > *max_y) *max_y = y2+2;
  if (y3 > *max_y) *max_y = y3+2;

  _sinra = round(sina * (1<<FP_SCALE));
  _cosra = round(cosa * (1<<FP_SCALE));
}
```

```
/***************************************************************************************
** Function name:           pushSprite
** Description:             Push the sprite to the TFT at x, y
***************************************************************************************/
void TFT_eSprite::pushSprite(int32_t x, int32_t y)
{
  if (!_created) return;

  if (_bpp == 16)
  {
    bool oldSwapBytes = _tft->getSwapBytes();
    _tft->setSwapBytes(false);
    _tft->pushImage(x, y, _dwidth, _dheight, _img );
    _tft->setSwapBytes(oldSwapBytes);
  }
  else if (_bpp == 4)
  {
    _tft->pushImage(x, y, _dwidth, _dheight, _img4, false, _colorMap);
  }
  else _tft->pushImage(x, y, _dwidth, _dheight, _img8, (bool)(_bpp == 8));
}
```

```
/***************************************************************************************
** Function name:           pushSprite
** Description:             Push the sprite to the TFT at x, y with transparent colour
***************************************************************************************/
void TFT_eSprite::pushSprite(int32_t x, int32_t y, uint16_t transp)
{
  if (!_created) return;

  if (_bpp == 16)
  {
    bool oldSwapBytes = _tft->getSwapBytes();
    _tft->setSwapBytes(false);
    _tft->pushImage(x, y, _dwidth, _dheight, _img, transp );
    _tft->setSwapBytes(oldSwapBytes);
  }
  else if (_bpp == 8)
  {
    transp = (uint8_t)((transp & 0xE000)>>8 | (transp & 0x0700)>>6 | (transp & 0x0018)>>3);
    _tft->pushImage(x, y, _dwidth, _dheight, _img8, (uint8_t)transp, (bool)true);
  }
  else if (_bpp == 4)
  {
    _tft->pushImage(x, y, _dwidth, _dheight, _img4, (uint8_t)(transp & 0x0F), false, _colorMap);
  }
  else _tft->pushImage(x, y, _dwidth, _dheight, _img8, 0, (bool)false);
}
```

```
/***************************************************************************************
** Function name:           pushToSprite
** Description:             Push the sprite to another sprite at x, y
***************************************************************************************/
// Note: The following sprite to sprite colour depths are currently supported:
//    Source    Destination
//    16bpp  -> 16bpp
//    16bpp  ->  8bpp
//     8bpp  ->  8bpp
//     4bpp  ->  4bpp (note: color translation depends on the 2 sprites pallete colors)
//     1bpp  ->  1bpp (note: color translation depends on the 2 sprites bitmap colors)

bool TFT_eSprite::pushToSprite(TFT_eSprite *dspr, int32_t x, int32_t y)
{
  if (!_created) return false;
  if (!dspr->created()) return false;

  // Check destination sprite compatibility
  int8_t ds_bpp = dspr->getColorDepth();
  if (_bpp == 16 && ds_bpp != 16 && ds_bpp !=  8) return false;
  if (_bpp ==  8 && ds_bpp !=  8) return false;
  if (_bpp ==  4 && ds_bpp !=  4) return false;
  if (_bpp ==  1 && ds_bpp !=  1) return false;

  bool oldSwapBytes = dspr->getSwapBytes();
  dspr->setSwapBytes(false);
  dspr->pushImage(x, y, _dwidth, _dheight, _img, _bpp);
  dspr->setSwapBytes(oldSwapBytes);

  return true;
}
```

```
/***************************************************************************************
** Function name:           pushToSprite
** Description:             Push the sprite to another sprite at x, y with transparent colour
***************************************************************************************/
// Note: The following sprite to sprite colour depths are currently supported:
//    Source    Destination
//    16bpp  -> 16bpp
//    16bpp  ->  8bpp
//     8bpp  ->  8bpp
//     1bpp  ->  1bpp

bool TFT_eSprite::pushToSprite(TFT_eSprite *dspr, int32_t x, int32_t y, uint16_t transp)
{
  if ( !_created  || !dspr->_created) return false; // Check Sprites exist

  // Check destination sprite compatibility
  int8_t ds_bpp = dspr->getColorDepth();
  if (_bpp == 16 && ds_bpp != 16 && ds_bpp !=  8) return false;
  if (_bpp ==  8 && ds_bpp !=  8) return false;
  if (_bpp ==  4 || ds_bpp ==  4) return false;
  if (_bpp ==  1 && ds_bpp !=  1) return false;

  bool oldSwapBytes = dspr->getSwapBytes();
  uint16_t sline_buffer[width()];

  transp = transp>>8 | transp<<8;

  // Scan destination bounding box and fetch transformed pixels from source Sprite
  for (int32_t ys = 0; ys < height(); ys++) {
    int32_t ox = x;
    uint32_t pixel_count = 0;

    for (int32_t xs = 0; xs < width(); xs++) {
      uint16_t rp = 0;
      if (_bpp == 16) rp = _img[xs + ys * width()];
      else { rp = readPixel(xs, ys); rp = rp>>8 | rp<<8; }
      //dspr->drawPixel(xs, ys, rp);

      if (transp == rp) {
        if (pixel_count) {
          dspr->pushImage(ox, y, pixel_count, 1, sline_buffer, _bpp);
          ox += pixel_count;
          pixel_count = 0;
        }
        else ox++;
      }
      else {
        sline_buffer[pixel_count++] = rp;
      }
    }
    if (pixel_count) dspr->pushImage(ox, y, pixel_count, 1, sline_buffer);
    y++;
  }
  dspr->setSwapBytes(oldSwapBytes);
  return true;
}
```

```
/***************************************************************************************
** Function name:           pushSprite
** Description:             Push a cropped sprite to the TFT at tx, ty
***************************************************************************************/
bool TFT_eSprite::pushSprite(int32_t tx, int32_t ty, int32_t sx, int32_t sy, int32_t sw, int32_t sh)
{
  if (!_created) return false;

  // Perform window boundary checks and crop if needed
  setWindow(sx, sy, sx + sw - 1, sy + sh - 1);

  /* These global variables are now populated for the sprite
  _xs = x start coordinate
  _ys = y start coordinate
  _xe = x end coordinate (inclusive)
  _ye = y end coordinate (inclusive)
  */

  // Calculate new sprite window bounding box width and height
  sw = _xe - _xs + 1;
  sh = _ye - _ys + 1;

  if (_ys >= _iheight) return false;

  if (_bpp == 16)
  {
    bool oldSwapBytes = _tft->getSwapBytes();
    _tft->setSwapBytes(false);

    // Check if a faster block copy to screen is possible
    if ( sx == 0 && sw == _dwidth)
      _tft->pushImage(tx, ty, sw, sh, _img + _iwidth * _ys );
    else // Render line by line
      while (sh--)
        _tft->pushImage(tx, ty++, sw, 1, _img + _xs + _iwidth * _ys++ );

    _tft->setSwapBytes(oldSwapBytes);
  }
  else if (_bpp == 8)
  {
    // Check if a faster block copy to screen is possible
    if ( sx == 0 && sw == _dwidth)
      _tft->pushImage(tx, ty, sw, sh, _img8 + _iwidth * _ys, (bool)true );
    else // Render line by line
    while (sh--)
      _tft->pushImage(tx, ty++, sw, 1, _img8 + _xs + _iwidth * _ys++, (bool)true );
  }
  else if (_bpp == 4)
  {
    // Check if a faster block copy to screen is possible
    if ( sx == 0 && sw == _dwidth)
      _tft->pushImage(tx, ty, sw, sh, _img4 + (_iwidth>>1) * _ys, false, _colorMap );
    else // Render line by line
    {
      int32_t ds = _xs&1; // Odd x start pixel

      int32_t de = 0;     // Odd x end pixel
      if ((sw > ds) && (_xe&1)) de = 1;

      uint32_t dm = 0;     // Midsection pixel count
      if (sw > (ds+de)) dm = sw - ds - de;
      sw--;

      uint32_t yp = (_xs + ds + _iwidth * _ys)>>1;
      _tft->startWrite();
      while (sh--)
      {
        if (ds) _tft->drawPixel(tx, ty, readPixel(_xs, _ys) );
        if (dm) _tft->pushImage(tx + ds, ty, dm, 1, _img4 + yp, false, _colorMap );
        if (de) _tft->drawPixel(tx + sw, ty, readPixel(_xe, _ys) );
        _ys++;
        ty++;
        yp += (_iwidth>>1);
      }
      _tft->endWrite();
    }
  }
  else // 1bpp
  {
    // Check if a faster block copy to screen is possible
    if ( sx == 0 && sw == _dwidth)
      _tft->pushImage(tx, ty, sw, sh, _img8 + (_bitwidth>>3) * _ys, (bool)false );
    else // Render line by line
    {
```

```
      while (sh--)
      {
        _tft->pushImage(tx, ty++, sw, 1, _img8 + (_bitwidth>>3) * _ys, (bool)false );
      }
      _tft->endWrite();
    }
  }

  return true;
}
```

```
/**************************************************************************************
** Function name:           readPixelValue
** Description:             Read the color map index of a pixel at defined coordinates
**************************************************************************************/
uint16_t TFT_eSprite::readPixelValue(int32_t x, int32_t y)
{
  if (_vpOoB  || !_created) return 0xFF;

  x+= _xDatum;
  y+= _yDatum;

  // Range checking
  if ((x < _vpX) || (y < _vpY) ||(x >= _vpW) || (y >= _vpH)) return 0xFF;

  if (_bpp == 16)
  {
    // Return the pixel colour
    return readPixel(x - _xDatum, y - _yDatum);
  }

  if (_bpp == 8)
  {
    // Return the pixel byte value
    return _img8[x + y * _iwidth];
  }

  if (_bpp == 4)
  {
    if (x >= _dwidth) return 0xFF;
    if ((x & 0x01) == 0)
      return _img4[((x+y*_iwidth)>>1)] >> 4;   // even index = bits 7 .. 4
    else
      return _img4[((x+y*_iwidth)>>1)] & 0x0F; // odd index = bits 3 .. 0.
  }

  if (_bpp == 1)
  {
    // Note: _dwidth and _dheight bounds not checked (rounded up -iwidth and _iheight used)
    if (rotation == 1)
    {
      uint16_t tx = x;
      x = _dheight - y - 1;
      y = tx;
    }
    else if (rotation == 2)
    {
      x = _dwidth - x - 1;
      y = _dheight - y - 1;
    }
    else if (rotation == 3)
    {
      uint16_t tx = x;
      x = y;
      y = _dwidth - tx - 1;
    }
    // Return 1 or 0
    return (_img8[(x + y * _bitwidth)>>3] >> (7-(x & 0x7))) & 0x01;
  }

  return 0;
}
```

```
/**************************************************************************************
** Function name:           readPixel
** Description:             Read 565 colour of a pixel at defined coordinates
**************************************************************************************/
uint16_t TFT_eSprite::readPixel(int32_t x, int32_t y)
{
  if (_vpOoB  || !_created) return 0xFFFF;

  x+= _xDatum;
  y+= _yDatum;

  // Range checking
  if ((x < _vpX) || (y < _vpY) ||(x >= _vpW) || (y >= _vpH)) return 0xFFFF;

  if (_bpp == 16)
  {
    uint16_t color = _img[x + y * _iwidth];
    return (color >> 8) | (color << 8);
  }

  if (_bpp == 8)
  {
    uint16_t color = _img8[x + y * _iwidth];
    if (color != 0)
    {
    uint8_t  blue[] = {0, 11, 21, 31};
      color =   (color & 0xE0)<<8 | (color & 0xC0)<<5
              | (color & 0x1C)<<6 | (color & 0x1C)<<3
              | blue[color & 0x03];
    }
    return color;
  }

  if (_bpp == 4)
  {
    if (x >= _dwidth) return 0xFFFF;
    uint16_t color;
    if ((x & 0x01) == 0)
      color = _colorMap[_img4[((x+y*_iwidth)>>1)] >> 4];   // even index = bits 7 .. 4
    else
      color = _colorMap[_img4[((x+y*_iwidth)>>1)] & 0x0F]; // odd index = bits 3 .. 0.
    return color;
  }

  // Note: Must be 1bpp
  // _dwidth and _dheight bounds not checked (rounded up -iwidth and _iheight used)
  if (rotation == 1)
  {
    uint16_t tx = x;
    x = _dheight - y - 1;
    y = tx;
  }
  else if (rotation == 2)
  {
    x = _dwidth - x - 1;
    y = _dheight - y - 1;
  }
  else if (rotation == 3)
  {
    uint16_t tx = x;
    x = y;
    y = _dwidth - tx - 1;
  }

  uint16_t color = (_img8[(x + y * _bitwidth)>>3] << (x & 0x7)) & 0x80;

  if (color) return _tft->bitmap_fg;
  else       return _tft->bitmap_bg;
}
```

```
/***************************************************************************************
** Function name:           pushImage
** Description:             push image into a defined area of a sprite
***************************************************************************************/
void  TFT_eSprite::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, uint16_t *data, uint8_t sbpp)
{
  if (data == nullptr || !_created) return;

  PI_CLIP;

  if (_bpp == 16) // Plot a 16 bpp image into a 16 bpp Sprite
  {
    // Pointer within original image
    uint8_t *ptro = (uint8_t *)data + ((dx + dy * w) << 1);
    // Pointer within sprite image
    uint8_t *ptrs = (uint8_t *)_img + ((x + y * _iwidth) << 1);

    if(_swapBytes)
    {
      while (dh--)
      {
        // Fast copy with a 1 byte shift
        memcpy(ptrs+1, ptro, (dw<<1) - 1);
        // Now correct just the even numbered bytes
        for (int32_t xp = 0; xp < (dw<<1); xp+=2)
        {
          ptrs[xp] = ptro[xp+1];;
        }
        ptro += w<<1;
        ptrs += _iwidth<<1;
      }
    }
    else
    {
      while (dh--)
      {
        memcpy(ptrs, ptro, dw<<1);
        ptro += w << 1;
        ptrs += _iwidth << 1;
      }
    }
  }
  else if (_bpp == 8 && sbpp == 8) // Plot a 8 bpp image into a 8 bpp Sprite
  {
    // Pointer within original image
    uint8_t *ptro = (uint8_t *)data + (dx + dy * w);
    // Pointer within sprite image
    uint8_t *ptrs = (uint8_t *)_img + (x + y * _iwidth);

    while (dh--)
    {
      memcpy(ptrs, ptro, dw);
      ptro += w;
      ptrs += _iwidth;
    }
  }
  else if (_bpp == 8) // Plot a 16 bpp image into a 8 bpp Sprite
  {
    uint16_t lastColor = 0;
    uint8_t  color8    = 0;
    for (int32_t yp = dy; yp < dy + dh; yp++)
    {
      int32_t xyw = x + y * _iwidth;
      int32_t dxypw = dx + yp * w;
      for (int32_t xp = dx; xp < dx + dw; xp++)
      {
        uint16_t color = data[dxypw++];
        if (color != lastColor) {
          // When data source is a sprite, the bytes are already swapped
          if(!_swapBytes) color8 = (uint8_t)((color & 0xE0) | (color & 0x07)<<2 | (color & 0x1800)>>11);
          else color8 = (uint8_t)((color & 0xE000)>>8 | (color & 0x0700)>>6 | (color & 0x0018)>>3);
        }
        lastColor = color;
        _img8[xyw++] = color8;
      }
      y++;
    }
  }
  else if (_bpp == 4)
  {
    // The image is assumed to be 4 bit, where each byte corresponds to two pixels.
    // much faster when aligned to a byte boundary, because the alternative is slower, requiring
    // tedious bit operations.
```

```
    uint8_t *ptr = (uint8_t *)data;

    if ((x & 0x01) == 0 && (dx & 0x01) == 0 && (dw & 0x01) == 0)
    {
      x = (x >> 1) + y * sWidth;
      dw = (dw >> 1);
      dx = (dx >> 1) + dy * (w>>1);
      while (dh--)
      {
        memcpy(_img4 + x, ptr + dx, dw);
        dx += (w >> 1);
        x += sWidth;
      }
    }
    else  // not optimized
    {
      for (int32_t yp = dy; yp < dy + dh; yp++)
      {
        int32_t ox = x;
        for (int32_t xp = dx; xp < dx + dw; xp++)
        {
          uint32_t color;
          if ((xp & 0x01) == 0)
            color = (ptr[((xp+yp*w)>>1)] & 0xF0) >> 4; // even index = bits 7 .. 4
          else
            color = ptr[((xp-1+yp*w)>>1)] & 0x0F;      // odd index = bits 3 .. 0.
          drawPixel(ox, y, color);
          ox++;
        }
        y++;
      }
    }
  }

  else // 1bpp
  {
    // Plot a 1bpp image into a 1bpp Sprite
    uint32_t ww =  (w+7)>>3; // Width of source image line in bytes
    uint8_t *ptr = (uint8_t *)data;
    for (int32_t yp = dy;  yp < dy + dh; yp++)
    {
      uint32_t yw = yp * ww;              // Byte starting the line containing source pixel
      int32_t ox = x;
      for (int32_t xp = dx; xp < dx + dw; xp++)
      {
        uint16_t readPixel = (ptr[(xp>>3) + yw] & (0x80 >> (xp & 0x7)) );
        drawPixel(ox++, y, readPixel);
      }
      y++;
    }
  }
}
```

```
/***************************************************************************************
** Function name:           pushImage
** Description:             push 565 colour FLASH (PROGMEM) image into a defined area
***************************************************************************************/
void  TFT_eSprite::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, const uint16_t *data)
{
#ifdef ESP32
  pushImage(x, y, w, h, (uint16_t*) data);
#else
  // Partitioned memory FLASH processor
  if (data == nullptr || !_created) return;

  PI_CLIP;

  if (_bpp == 16) // Plot a 16 bpp image into a 16 bpp Sprite
  {
    for (int32_t yp = dy; yp < dy + dh; yp++)
    {
      int32_t ox = x;
      for (int32_t xp = dx; xp < dx + dw; xp++)
      {
        uint16_t color = pgm_read_word(data + xp + yp * w);
        if(_swapBytes) color = color<<8 | color>>8;
        _img[ox + y * _iwidth] = color;
        ox++;
      }
      y++;
    }
  }

  else if (_bpp == 8) // Plot a 16 bpp image into a 8 bpp Sprite
  {
    for (int32_t yp = dy; yp < dy + dh; yp++)
    {
      int32_t ox = x;
      for (int32_t xp = dx; xp < dx + dw; xp++)
      {
        uint16_t color = pgm_read_word(data + xp + yp * w);
        if(_swapBytes) color = color<<8 | color>>8;
        _img8[ox + y * _iwidth] = (uint8_t)((color & 0xE000)>>8 | (color & 0x0700)>>6 | (color & 0x0018)>>3);
        ox++;
      }
      y++;
    }
  }

  else if (_bpp == 4)
  {
    #ifdef TFT_eSPI_DEBUG
    Serial.println("TFT_eSprite::pushImage(int32_t x, int32_t y, int32_t w, int32_t h, const uint16_t *data) not impl
    #endif
    return;
  }

  else // Plot a 1bpp image into a 1bpp Sprite
  {
    x-= _xDatum;   // Remove offsets, drawPixel will add
    y-= _yDatum;
    uint16_t bsw =  (w+7) >> 3; // Width in bytes of source image line
    uint8_t *ptr = ((uint8_t*)data) + dy * bsw;

    while (dh--) {
      int32_t odx = dx;
      int32_t ox  = x;
      while (odx < dx + dw) {
        uint8_t pbyte = pgm_read_byte(ptr + (odx>>3));
        uint8_t mask = 0x80 >> (odx & 7);
        while (mask) {
          uint8_t p = pbyte & mask;
          mask = mask >> 1;
          drawPixel(ox++, y, p);
          odx++;
        }
      }
      ptr += bsw;
      y++;
    }
  }
#endif // if ESP32 check
}
```

```
/***************************************************************************************
** Function name:           setWindow
** Description:             Set the bounds of a window in the sprite
***************************************************************************************/
// Intentionally not constrained to viewport area, does not manage 1bpp rotations
void TFT_eSprite::setWindow(int32_t x0, int32_t y0, int32_t x1, int32_t y1)
{
  if (x0 > x1) swap_coord(x0, x1);
  if (y0 > y1) swap_coord(y0, y1);

  int32_t w = width();
  int32_t h = height();

  if ((x0 >= w) || (x1 < 0) || (y0 >= h) || (y1 < 0))
  { // Point to that extra "off screen" pixel
    _xs = 0;
    _ys = _dheight;
    _xe = 0;
    _ye = _dheight;
  }
  else
  {
    if (x0 < 0) x0 = 0;
    if (x1 >= w) x1 = w - 1;
    if (y0 < 0) y0 = 0;
    if (y1 >= h) y1 = h - 1;

    _xs = x0;
    _ys = y0;
    _xe = x1;
    _ye = y1;
  }

  _xptr = _xs;
  _yptr = _ys;
}
```

```
/***************************************************************************************
** Function name:           pushColor
** Description:             Send a new pixel to the set window
***************************************************************************************/
void TFT_eSprite::pushColor(uint32_t color)
{
  if (!_created ) return;

  // Write the colour to RAM in set window
  if (_bpp == 16)
    _img [_xptr + _yptr * _iwidth] = (uint16_t) (color >> 8) | (color << 8);

  else  if (_bpp == 8)
    _img8[_xptr + _yptr * _iwidth] = (uint8_t )((color & 0xE000)>>8 | (color & 0x0700)>>6 | (color & 0x0018)>>3

  else if (_bpp == 4)
  {
    uint8_t c = (uint8_t)color & 0x0F;
    if ((_xptr & 0x01) == 0) {
      _img4[(_xptr + _yptr * _iwidth)>>1] = (c << 4) | (_img4[(_xptr + _yptr * _iwidth)>>1] & 0x0F);  // new colo
    }
    else {
      _img4[(_xptr + _yptr * _iwidth)>>1] = (_img4[(_xptr + _yptr * _iwidth)>>1] & 0xF0) | c; // new color is the
    }
  }

  else drawPixel(_xptr, _yptr, color);

  // Increment x
  _xptr++;

  // Wrap on x and y to start, increment y if needed
  if (_xptr > _xe)
  {
    _xptr = _xs;
    _yptr++;
    if (_yptr > _ye) _yptr = _ys;
  }

}
```

```
/***************************************************************************************
** Function name:           pushColor
** Description:             Send a "len" new pixels to the set window
***************************************************************************************/
void TFT_eSprite::pushColor(uint32_t color, uint16_t len)
{
  if (!_created ) return;

  uint16_t pixelColor;

  if (_bpp == 16)
    pixelColor = (uint16_t) (color >> 8) | (color << 8);

  else  if (_bpp == 8)
    pixelColor = (color & 0xE000)>>8 | (color & 0x0700)>>6 | (color & 0x0018)>>3;

  else pixelColor = (uint16_t) color; // for 1bpp or 4bpp

  while(len--) writeColor(pixelColor);
}
```

```
/**************************************************************************************
** Function name:           writeColor
** Description:             Write a pixel with pre-formatted colour to the set window
**************************************************************************************/
void TFT_eSprite::writeColor(uint16_t color)
{
  if (!_created ) return;

  // Write 16 bit RGB 565 encoded colour to RAM
  if (_bpp == 16) _img [_xptr + _yptr * _iwidth] = color;

  // Write 8 bit RGB 332 encoded colour to RAM
  else if (_bpp == 8) _img8[_xptr + _yptr * _iwidth] = (uint8_t) color;

  else if (_bpp == 4)
  {
    uint8_t c = (uint8_t)color & 0x0F;
    if ((_xptr & 0x01) == 0)
      _img4[(_xptr + _yptr * _iwidth)>>1] = (c << 4) | (_img4[(_xptr + _yptr * _iwidth)>>1] & 0x0F);  // new colo
    else
      _img4[(_xptr + _yptr * _iwidth)>>1] = (_img4[(_xptr + _yptr * _iwidth)>>1] & 0xF0) | c; // new color is the
  }

  else drawPixel(_xptr, _yptr, color);

  // Increment x
  _xptr++;

  // Wrap on x and y to start, increment y if needed
  if (_xptr > _xe)
  {
    _xptr = _xs;
    _yptr++;
    if (_yptr > _ye) _yptr = _ys;
  }
}
```

```
/***************************************************************************************
** Function name:           setScrollRect
** Description:             Set scroll area within the sprite and the gap fill colour
***************************************************************************************/
// Intentionally not constrained to viewport area
void TFT_eSprite::setScrollRect(int32_t x, int32_t y, int32_t w, int32_t h, uint16_t color)
{
  if ((x >= _iwidth) || (y >= _iheight) || !_created ) return;

  if (x < 0) { w += x; x = 0; }
  if (y < 0) { h += y; y = 0; }

  if ((x + w) > _iwidth ) w = _iwidth  - x;
  if ((y + h) > _iheight) h = _iheight - y;

  if ( w < 1 || h < 1) return;

  _sx = x;
  _sy = y;
  _sw = w;
  _sh = h;

  _scolor = color;
}
```

```
/***************************************************************************************
** Function name:           scroll
** Description:             Scroll dx,dy pixels, positive right,down, negative left,up
***************************************************************************************/
void TFT_eSprite::scroll(int16_t dx, int16_t dy)
{
  if (abs(dx) >= _sw || abs(dy) >= _sh)
  {
    fillRect (_sx, _sy, _sw, _sh, _scolor);
    return;
  }

  // Fetch the scroll area width and height set by setScrollRect()
  uint32_t w  = _sw - abs(dx); // line width to copy
  uint32_t h  = _sh - abs(dy); // lines to copy
  int32_t iw  = _iwidth;       // rounded up width of sprite

  // Fetch the x,y origin set by setScrollRect()
  uint32_t tx = _sx; // to x
  uint32_t fx = _sx; // from x
  uint32_t ty = _sy; // to y
  uint32_t fy = _sy; // from y

  // Adjust for x delta
  if (dx <= 0) fx -= dx;
  else tx += dx;

  // Adjust for y delta
  if (dy <= 0) fy -= dy;
  else
  { // Scrolling down so start copy from bottom
    ty = ty + _sh - 1; // "To" pointer
    iw = -iw;          // Pointer moves backwards
    fy = ty - dy;      // "From" pointer
  }

  // Calculate "from y" and "to y" pointers in RAM
  uint32_t fyp = fx + fy * _iwidth;
  uint32_t typ = tx + ty * _iwidth;

  // Now move the pixels in RAM
  if (_bpp == 16)
  {
    while (h--)
    { // move pixel lines (to, from, byte count)
      memmove( _img + typ, _img + fyp, w<<1);
      typ += iw;
      fyp += iw;
    }
  }
  else if (_bpp == 8)
  {
    while (h--)
    { // move pixel lines (to, from, byte count)
      memmove( _img8 + typ, _img8 + fyp, w);
      typ += iw;
      fyp += iw;
    }
  }
  else if (_bpp == 4)
  {
    // could optimize for scrolling by even # pixels using memove (later)
    if (dx >  0) { tx += w; fx += w; } // Start from right edge
    while (h--)
    { // move pixels one by one
      for (uint16_t xp = 0; xp < w; xp++)
      {
        if (dx <= 0) drawPixel(tx + xp, ty, readPixelValue(fx + xp, fy));
        if (dx >  0) drawPixel(tx - xp, ty, readPixelValue(fx - xp, fy));
      }
      if (dy <= 0)  { ty++; fy++; }
      else  { ty--; fy--; }
    }
  }
  else if (_bpp == 1 )
  {
    if (dx >  0) { tx += w; fx += w; } // Start from right edge
    while (h--)
    { // move pixels one by one
      for (uint16_t xp = 0; xp < w; xp++)
      {
        if (dx <= 0) drawPixel(tx + xp, ty, readPixelValue(fx + xp, fy));
        if (dx >  0) drawPixel(tx - xp, ty, readPixelValue(fx - xp, fy));
      }
```

```
      else  { ty--; fy--; }
    }
  }
  else return; // Not 1, 4, 8 or 16 bpp

  // Fill the gap left by the scrolling
  if (dx > 0) fillRect(_sx, _sy, dx, _sh, _scolor);
  if (dx < 0) fillRect(_sx + _sw + dx, _sy, -dx, _sh, _scolor);
  if (dy > 0) fillRect(_sx, _sy, _sw, dy, _scolor);
  if (dy < 0) fillRect(_sx, _sy + _sh + dy, _sw, -dy, _scolor);
}
```

```
/**************************************************************************************
** Function name:           fillSprite
** Description:             Fill the whole sprite with defined colour
**************************************************************************************/
void TFT_eSprite::fillSprite(uint32_t color)
{
  if (!_created || _vpOoB) return;

  // Use memset if possible as it is super fast
  if(_xDatum == 0 && _yDatum == 0  &&  _xWidth == width())
  {
    if(_bpp == 16) {
      if ( (uint8_t)color == (uint8_t)(color>>8) ) {
        memset(_img,  (uint8_t)color, _iwidth * _yHeight * 2);
      }
      else fillRect(_vpX, _vpY, _xWidth, _yHeight, color);
    }
    else if (_bpp == 8)
    {
      color = (color & 0xE000)>>8 | (color & 0x0700)>>6 | (color & 0x0018)>>3;
      memset(_img8, (uint8_t)color, _iwidth * _yHeight);
    }
    else if (_bpp == 4)
    {
      uint8_t c = ((color & 0x0F) | (((color & 0x0F) << 4) & 0xF0));
      memset(_img4, c, (_iwidth * _yHeight) >> 1);
    }
    else if (_bpp == 1)
    {
      if(color) memset(_img8, 0xFF, (_bitwidth>>3) * _dheight + 1);
      else      memset(_img8, 0x00, (_bitwidth>>3) * _dheight + 1);
    }
  }
  else fillRect(_vpX - _xDatum, _vpY - _yDatum, _xWidth, _yHeight, color);
}
```

```
/***************************************************************************************
** Function name:           width
** Description:             Return the width of sprite
***************************************************************************************/
// Return the size of the display
int16_t TFT_eSprite::width(void)
{
  if (!_created ) return 0;

  if (_bpp > 1) {
    if (_vpDatum) return _xWidth;
    return _dwidth;
  }

  if (rotation & 1) {
    if (_vpDatum) return _xWidth;
    return _dheight;
  }

  if (_vpDatum) return _xWidth;
  return _dwidth;
}
```

```
/**************************************************************************************
** Function name:           height
** Description:             Return the height of sprite
**************************************************************************************/
int16_t TFT_eSprite::height(void)
{
  if (!_created ) return 0;

  if (_bpp > 1) {
    if (_vpDatum) return _yHeight;
    return _dheight;
  }

  if (rotation & 1) {
    if (_vpDatum) return _yHeight;
    return _dwidth;
  }

  if (_vpDatum) return _yHeight;
  return _dheight;
}
```

```
/***************************************************************************************
** Function name:           setRotation
** Description:             Rotate coordinate frame for 1bpp sprite
***************************************************************************************/
// Does nothing for 4, 8 and 16 bpp sprites.
void TFT_eSprite::setRotation(uint8_t r)
{
  if (_bpp != 1) return;

  rotation = r;

  if (rotation&1) {
    resetViewport();
  }
  else {
    resetViewport();
  }
}
```

```
/*****************************************************************************
** Function name:           getRotation
** Description:             Get rotation for 1bpp sprite
*****************************************************************************/
uint8_t TFT_eSprite::getRotation(void)
{
  return rotation;
}
```

```
/**************************************************************************************
** Function name:           drawPixel
** Description:             push a single pixel at an arbitrary position
**************************************************************************************/
void TFT_eSprite::drawPixel(int32_t x, int32_t y, uint32_t color)
{
  if (!_created || _vpOoB) return;

  x+= _xDatum;
  y+= _yDatum;

  // Range checking
  if ((x < _vpX) || (y < _vpY) ||(x >= _vpW) || (y >= _vpH)) return;

  if (_bpp == 16)
  {
    color = (color >> 8) | (color << 8);
    _img[x+y*_iwidth] = (uint16_t) color;
  }
  else if (_bpp == 8)
  {
    _img8[x+y*_iwidth] = (uint8_t)((color & 0xE000)>>8 | (color & 0x0700)>>6 | (color & 0x0018)>>3);
  }
  else if (_bpp == 4)
  {
    uint8_t c = color & 0x0F;
    int index = (x+y*_iwidth)>>1;;
    if ((x & 0x01) == 0) {
      _img4[index] = (uint8_t)((c << 4) | (_img4[index] & 0x0F));
    }
    else {
      _img4[index] =  (uint8_t)(c | (_img4[index] & 0xF0));
    }
  }
  else // 1 bpp
  {
    if (rotation == 1)
    {
      uint16_t tx = x;
      x = _dwidth - y - 1;
      y = tx;
    }
    else if (rotation == 2)
    {
      x = _dwidth - x - 1;
      y = _dheight - y - 1;
    }
    else if (rotation == 3)
    {
      uint16_t tx = x;
      x = y;
      y = _dheight - tx - 1;
    }

    if (color) _img8[(x + y * _bitwidth)>>3] |=  (0x80 >> (x & 0x7));
    else       _img8[(x + y * _bitwidth)>>3] &= ~(0x80 >> (x & 0x7));
  }
}
```

```
/***************************************************************************************
** Function name:           drawLine
** Description:             draw a line between 2 arbitrary points
***************************************************************************************/
void TFT_eSprite::drawLine(int32_t x0, int32_t y0, int32_t x1, int32_t y1, uint32_t color)
{
  if (!_created || _vpOoB) return;

  //_xDatum and _yDatum Not added here, it is added by drawPixel & drawFastxLine

  bool steep = abs(y1 - y0) > abs(x1 - x0);
  if (steep) {
    swap_coord(x0, y0);
    swap_coord(x1, y1);
  }

  if (x0 > x1) {
    swap_coord(x0, x1);
    swap_coord(y0, y1);
  }

  int32_t dx = x1 - x0, dy = abs(y1 - y0);;

  int32_t err = dx >> 1, ystep = -1, xs = x0, dlen = 0;

  if (y0 < y1) ystep = 1;

  // Split into steep and not steep for FastH/V separation
  if (steep) {
    for (; x0 <= x1; x0++) {
      dlen++;
      err -= dy;
      if (err < 0) {
        err += dx;
        if (dlen == 1) drawPixel(y0, xs, color);
        else drawFastVLine(y0, xs, dlen, color);
        dlen = 0; y0 += ystep; xs = x0 + 1;
      }
    }
    if (dlen) drawFastVLine(y0, xs, dlen, color);
  }
  else
  {
    for (; x0 <= x1; x0++) {
      dlen++;
      err -= dy;
      if (err < 0) {
        err += dx;
        if (dlen == 1) drawPixel(xs, y0, color);
        else drawFastHLine(xs, y0, dlen, color);
        dlen = 0; y0 += ystep; xs = x0 + 1;
      }
    }
    if (dlen) drawFastHLine(xs, y0, dlen, color);
  }
}
```

```
/***************************************************************************************
** Function name:           drawFastVLine
** Description:             draw a vertical line
***************************************************************************************/
void TFT_eSprite::drawFastVLine(int32_t x, int32_t y, int32_t h, uint32_t color)
{
  if (!_created || _vpOoB) return;

  x+= _xDatum;
  y+= _yDatum;

  // Clipping
  if ((x < _vpX) || (x >= _vpW) || (y >= _vpH)) return;

  if (y < _vpY) { h += y - _vpY; y = _vpY; }

  if ((y + h) > _vpH) h = _vpH - y;

  if (h < 1) return;

  if (_bpp == 16)
  {
    color = (color >> 8) | (color << 8);
    int32_t yp = x + _iwidth * y;
    while (h--) {_img[yp] = (uint16_t) color; yp += _iwidth;}
  }
  else if (_bpp == 8)
  {
    color = (color & 0xE000)>>8 | (color & 0x0700)>>6 | (color & 0x0018)>>3;
    while (h--) _img8[x + _iwidth * y++] = (uint8_t) color;
  }
  else if (_bpp == 4)
  {
    if ((x & 0x01) == 0)
    {
      uint8_t c = (uint8_t) (color & 0xF) << 4;
      while (h--) {
        _img4[(x + _iwidth * y)>>1] = (uint8_t) (c | (_img4[(x + _iwidth * y)>>1] & 0x0F));
        y++;
      }
    }
    else {
      uint8_t c = (uint8_t)color & 0xF;
      while (h--) {
        _img4[(x + _iwidth * y)>>1] = (uint8_t) (c | (_img4[(x + _iwidth * y)>>1] & 0xF0)); // x is odd; new color
        y++;
      }
    }
  }
  else
  {
    x -= _xDatum; // Remove any offset as it will be added by drawPixel
    y -= _yDatum;
    while (h--)
    {
      drawPixel(x, y++, color);
    }
  }
}
```

```
/**************************************************************************************
** Function name:           drawFastHLine
** Description:             draw a horizontal line
**************************************************************************************/
void TFT_eSprite::drawFastHLine(int32_t x, int32_t y, int32_t w, uint32_t color)
{
  if (!_created || _vpOoB) return;

  x+= _xDatum;
  y+= _yDatum;

  // Clipping
  if ((y < _vpY) || (x >= _vpW) || (y >= _vpH)) return;

  if (x < _vpX) { w += x - _vpX; x = _vpX; }

  if ((x + w) > _vpW) w = _vpW - x;

  if (w < 1) return;

  if (_bpp == 16)
  {
    color = (color >> 8) | (color << 8);
    while (w--) _img[_iwidth * y + x++] = (uint16_t) color;
  }
  else if (_bpp == 8)
  {
    color = (color & 0xE000)>>8 | (color & 0x0700)>>6 | (color & 0x0018)>>3;
    memset(_img8+_iwidth * y + x, (uint8_t)color, w);
  }
  else if (_bpp == 4)
  {
    uint8_t c = (uint8_t)color & 0x0F;
    uint8_t c2 = (c | ((c << 4) & 0xF0));
    if ((x & 0x01) == 1)
    {
      drawPixel(x - _xDatum, y - _yDatum, color);
      x++; w--;
      if (w < 1)
        return;
    }

    if (((w + x) & 0x01) == 1)
    {
      // handle the extra one at the other end
      drawPixel(x - _xDatum + w - 1, y - _yDatum, color);
      w--;
      if (w < 1) return;
    }
    memset(_img4 + ((_iwidth * y + x) >> 1), c2, (w >> 1));
  }
  else {
    x -= _xDatum; // Remove any offset as it will be added by drawPixel
    y -= _yDatum;

    while (w--)
    {
      drawPixel(x++, y, color);
    }
  }
}
```

```
/***************************************************************************************
** Function name:           fillRect
** Description:             draw a filled rectangle
***************************************************************************************/
void TFT_eSprite::fillRect(int32_t x, int32_t y, int32_t w, int32_t h, uint32_t color)
{
  if (!_created || _vpOoB) return;

  x+= _xDatum;
  y+= _yDatum;

  // Clipping
  if ((x >= _vpW) || (y >= _vpH)) return;

  if (x < _vpX) { w += x - _vpX; x = _vpX; }
  if (y < _vpY) { h += y - _vpY; y = _vpY; }

  if ((x + w) > _vpW) w = _vpW - x;
  if ((y + h) > _vpH) h = _vpH - y;

  if ((w < 1) || (h < 1)) return;

  int32_t yp = _iwidth * y + x;

  if (_bpp == 16)
  {
    color = (color >> 8) | (color << 8);
    uint32_t iw = w;
    int32_t ys = yp;
    if(h--)  {while (iw--) _img[yp++] = (uint16_t) color;}
    yp = ys;
    while (h--)
    {
      yp += _iwidth;
      memcpy( _img+yp, _img+ys, w<<1);
    }
  }
  else if (_bpp == 8)
  {
    color = (color & 0xE000)>>8 | (color & 0x0700)>>6 | (color & 0x0018)>>3;
    while (h--)
    {
      memset(_img8 + yp, (uint8_t)color, w);
      yp += _iwidth;
    }
  }
  else if (_bpp == 4)
  {
    uint8_t c1 = (uint8_t)color & 0x0F;
    uint8_t c2 = c1 | ((c1 << 4) & 0xF0);
    if ((x & 0x01) == 0 && (w & 0x01) == 0)
    {
      yp = (yp >> 1);
      while (h--)
      {
        memset(_img4 + yp, c2, (w>>1));
        yp += (_iwidth >> 1);
      }
    }
    else if ((x & 0x01) == 0)
    {

      // same as above but you have a hangover on the right.
      yp = (yp >> 1);
      while (h--)
      {
        if (w > 1)
          memset(_img4 + yp, c2, (w-1)>>1);
        // handle the rightmost pixel by calling drawPixel
        drawPixel(x+w-1-_xDatum, y+h-_yDatum, c1);
        yp += (_iwidth >> 1);
      }
    }
    else if ((w & 0x01) == 1)
    {
      yp = (yp + 1) >> 1;
      while (h--) {
        drawPixel(x-_xDatum, y+h-_yDatum, color & 0x0F);
        if (w > 1)
          memset(_img4 + yp, c2, (w-1)>>1);
        // same as above but you have a hangover on the left instead
        yp += (_iwidth >> 1);
      }
    }
```

```
    {
      yp = (yp + 1) >> 1;
      while (h--) {
        drawPixel(x-_xDatum, y+h-_yDatum, color & 0x0F);
        if (w > 1) drawPixel(x+w-1-_xDatum, y+h-_yDatum, color & 0x0F);
        if (w > 2)
          memset(_img4 + yp, c2, (w-2)>>1);
        // maximal hacking, single pixels on left and right.
        yp += (_iwidth >> 1);
      }
    }
  }
  else
  {
    x -= _xDatum;
    y -= _yDatum;
    while (h--)
    {
      int32_t ww = w;
      int32_t xx = x;
      while (ww--) drawPixel(xx++, y, color);
      y++;
    }
  }
}
```

```
/***************************************************************************************
** Function name:           drawChar
** Description:             draw a single character in the Adafruit GLCD or freefont
***************************************************************************************/
void TFT_eSprite::drawChar(int32_t x, int32_t y, uint16_t c, uint32_t color, uint32_t bg, uint8_t size)
{
  if ( _vpOoB || !_created ) return;

  if ((x >= _vpW - _xDatum)                 || // Clip right
      (y >= _vpH - _yDatum)                 || // Clip bottom
      ((x + 6 * size - 1) < (_vpX - _xDatum)) || // Clip left
      ((y + 8 * size - 1) < (_vpY - _yDatum)))   // Clip top
    return;

  if (c < 32) return;
#ifdef LOAD_GLCD
//>>>>>>>>>>>>>>>>>>
#ifdef LOAD_GFXFF
  if(!gfxFont) { // 'Classic' built-in font
#endif
//>>>>>>>>>>>>>>>>>>

  bool fillbg = (bg != color);

  if ((size==1) && fillbg)
  {
    uint8_t column[6];
    uint8_t mask = 0x1;

    for (int8_t i = 0; i < 5; i++ ) column[i] = pgm_read_byte(font + (c * 5) + i);
    column[5] = 0;

    int8_t j, k;
    for (j = 0; j < 8; j++) {
      for (k = 0; k < 5; k++ ) {
        if (column[k] & mask) {
          drawPixel(x + k, y + j, color);
        }
        else {
          drawPixel(x + k, y + j, bg);
        }
      }

      mask <<= 1;

      drawPixel(x + k, y + j, bg);
    }
  }
  else
  {
    for (int8_t i = 0; i < 6; i++ ) {
      uint8_t line;
      if (i == 5)
        line = 0x0;
      else
        line = pgm_read_byte(font + (c * 5) + i);

      if (size == 1) // default size
      {
        for (int8_t j = 0; j < 8; j++) {
          if (line & 0x1) drawPixel(x + i, y + j, color);
          line >>= 1;
        }
      }
      else {  // big size
        for (int8_t j = 0; j < 8; j++) {
          if (line & 0x1) fillRect(x + (i * size), y + (j * size), size, size, color);
          else if (fillbg) fillRect(x + i * size, y + j * size, size, size, bg);
          line >>= 1;
        }
      }
    }
  }

//>>>>>>>>>>>>>>>>>>>>>>>>>>
#ifdef LOAD_GFXFF
  } else { // Custom font
#endif
//>>>>>>>>>>>>>>>>>>>>>>>>>>
#endif // LOAD_GLCD

#ifdef LOAD_GFXFF
    // Filter out bad characters not present in font
    if ((c >= pgm_read_word(&gfxFont->first)) && (c <= pgm_read_word(&gfxFont->last ))
```

```
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    c -= pgm_read_word(&gfxFont->first);
    GFXglyph *glyph  = &(((GFXglyph *)pgm_read_dword(&gfxFont->glyph))[c]);
    uint8_t  *bitmap = (uint8_t *)pgm_read_dword(&gfxFont->bitmap);

    uint32_t bo = pgm_read_word(&glyph->bitmapOffset);
    uint8_t  w  = pgm_read_byte(&glyph->width),
             h  = pgm_read_byte(&glyph->height);
             //xa = pgm_read_byte(&glyph->xAdvance);
    int8_t   xo = pgm_read_byte(&glyph->xOffset),
             yo = pgm_read_byte(&glyph->yOffset);
    uint8_t  xx, yy, bits=0, bit=0;
    int16_t  xo16 = 0, yo16 = 0;

    if(size > 1) {
      xo16 = xo;
      yo16 = yo;
    }

    uint16_t hpc = 0; // Horizontal foreground pixel count
    for(yy=0; yy<h; yy++) {
      for(xx=0; xx<w; xx++) {
        if(bit == 0) {
          bits = pgm_read_byte(&bitmap[bo++]);
          bit  = 0x80;
        }
        if(bits & bit) hpc++;
        else {
          if (hpc) {
            if(size == 1) drawFastHLine(x+xo+xx-hpc, y+yo+yy, hpc, color);
            else fillRect(x+(xo16+xx-hpc)*size, y+(yo16+yy)*size, size*hpc, size, color);
            hpc=0;
          }
        }
        bit >>= 1;
      }
      // Draw pixels for this line as we are about to increment yy
      if (hpc) {
        if(size == 1) drawFastHLine(x+xo+xx-hpc, y+yo+yy, hpc, color);
        else fillRect(x+(xo16+xx-hpc)*size, y+(yo16+yy)*size, size*hpc, size, color);
        hpc=0;
      }
    }
  }
#endif


#ifdef LOAD_GLCD
  #ifdef LOAD_GFXFF
  } // End classic vs custom font
  #endif
#endif

}
```

```
/***************************************************************************************
** Function name:           drawChar
** Description:             draw a unicode glyph onto the screen
***************************************************************************************/
  // TODO: Rationalise with TFT_eSPI
  // Any UTF-8 decoding must be done before calling drawChar()
int16_t TFT_eSprite::drawChar(uint16_t uniCode, int32_t x, int32_t y)
{
  return drawChar(uniCode, x, y, textfont);
}

  // Any UTF-8 decoding must be done before calling drawChar()
int16_t TFT_eSprite::drawChar(uint16_t uniCode, int32_t x, int32_t y, uint8_t font)
{
  if (_vpOoB || !uniCode) return 0;

  if (font==1) {
#ifdef LOAD_GLCD
  #ifndef LOAD_GFXFF
    drawChar(x, y, uniCode, textcolor, textbgcolor, textsize);
    return 6 * textsize;
  #endif
#else
  #ifndef LOAD_GFXFF
    return 0;
  #endif
#endif

#ifdef LOAD_GFXFF
    drawChar(x, y, uniCode, textcolor, textbgcolor, textsize);
    if(!gfxFont) { // 'Classic' built-in font
    #ifdef LOAD_GLCD
      return 6 * textsize;
    #else
      return 0;
    #endif
    }
    else {
      if((uniCode >= pgm_read_word(&gfxFont->first)) && (uniCode <= pgm_read_word(&gfxFont->last) )) {
        uint16_t   c2    = uniCode - pgm_read_word(&gfxFont->first);
        GFXglyph *glyph = &(((GFXglyph *)pgm_read_dword(&gfxFont->glyph))[c2]);
        return pgm_read_byte(&glyph->xAdvance) * textsize;
      }
      else {
        return 0;
      }
    }
#endif
  }

  if ((font>1) && (font<9) && ((uniCode < 32) || (uniCode > 127))) return 0;

  int32_t width  = 0;
  int32_t height = 0;
  uint32_t flash_address = 0;
  uniCode -= 32;

#ifdef LOAD_FONT2
  if (font == 2) {
    flash_address = pgm_read_dword(&chrtbl_f16[uniCode]);
    width = pgm_read_byte(widtbl_f16 + uniCode);
    height = chr_hgt_f16;
  }
  #ifdef LOAD_RLE
  else
  #endif
#endif

#ifdef LOAD_RLE
  {
    if ((font>2) && (font<9)) {
      flash_address = pgm_read_dword( (const void*)(pgm_read_dword( &(fontdata[font].chartbl ) ) + uniCode*size
      width = pgm_read_byte( (uint8_t *)pgm_read_dword( &(fontdata[font].widthtbl ) ) + uniCode );
      height= pgm_read_byte( &fontdata[font].height );
    }
  }
#endif

  int32_t xd = x + _xDatum;
  int32_t yd = y + _yDatum;

  if ((xd + width * textsize < _vpX || xd >= _vpW) && (yd + height * textsize < _vpY || yd >= _vpH)) return width

  int32_t w = width;
```

```
  int32_t pY      = y;
  uint8_t line = 0;
  bool clip = xd < _vpX || xd + width  * textsize >= _vpW || yd < _vpY || yd + height * textsize >= _vpH;

#ifdef LOAD_FONT2 // chop out code if we do not need it
  if (font == 2) {
    w = w + 6; // Should be + 7 but we need to compensate for width increment
    w = w / 8;

    for (int32_t i = 0; i < height; i++)
    {
      if (textcolor != textbgcolor) fillRect(x, pY, width * textsize, textsize, textbgcolor);

      for (int32_t k = 0; k < w; k++)
      {
        line = pgm_read_byte((uint8_t *)flash_address + w * i + k);
        if (line) {
          if (textsize == 1) {
            pX = x + k * 8;
            if (line & 0x80) drawPixel(pX, pY, textcolor);
            if (line & 0x40) drawPixel(pX + 1, pY, textcolor);
            if (line & 0x20) drawPixel(pX + 2, pY, textcolor);
            if (line & 0x10) drawPixel(pX + 3, pY, textcolor);
            if (line & 0x08) drawPixel(pX + 4, pY, textcolor);
            if (line & 0x04) drawPixel(pX + 5, pY, textcolor);
            if (line & 0x02) drawPixel(pX + 6, pY, textcolor);
            if (line & 0x01) drawPixel(pX + 7, pY, textcolor);
          }
          else {
            pX = x + k * 8 * textsize;
            if (line & 0x80) fillRect(pX, pY, textsize, textsize, textcolor);
            if (line & 0x40) fillRect(pX + textsize, pY, textsize, textsize, textcolor);
            if (line & 0x20) fillRect(pX + 2 * textsize, pY, textsize, textsize, textcolor);
            if (line & 0x10) fillRect(pX + 3 * textsize, pY, textsize, textsize, textcolor);
            if (line & 0x08) fillRect(pX + 4 * textsize, pY, textsize, textsize, textcolor);
            if (line & 0x04) fillRect(pX + 5 * textsize, pY, textsize, textsize, textcolor);
            if (line & 0x02) fillRect(pX + 6 * textsize, pY, textsize, textsize, textcolor);
            if (line & 0x01) fillRect(pX + 7 * textsize, pY, textsize, textsize, textcolor);
          }
        }
      }
      pY += textsize;
    }
  }

  #ifdef LOAD_RLE
  else
  #endif
#endif  //FONT2

#ifdef LOAD_RLE  //674 bytes of code
  // Font is not 2 and hence is RLE encoded
  {
    w *= height; // Now w is total number of pixels in the character
    int16_t color = textcolor;
    if (_bpp == 16) color = (textcolor >> 8) | (textcolor << 8);
    else if (_bpp == 8) color = ((textcolor & 0xE000)>>8 | (textcolor & 0x0700)>>6 | (textcolor & 0x0018)>>3);

    int16_t bgcolor = textbgcolor;
    if (_bpp == 16) bgcolor = (textbgcolor >> 8) | (textbgcolor << 8);
    else if (_bpp == 8) bgcolor = ((textbgcolor & 0xE000)>>8 | (textbgcolor & 0x0700)>>6 | (textbgcolor & 0x018

    if (textcolor == textbgcolor && !clip && _bpp != 1) {
      int32_t px = 0, py = pY; // To hold character block start and end column and row values
      int32_t pc = 0; // Pixel count
      uint8_t np = textsize * textsize; // Number of pixels in a drawn pixel

      uint8_t tnp = 0; // Temporary copy of np for while loop
      uint8_t ts = textsize - 1; // Temporary copy of textsize
      // 16 bit pixel count so maximum font size is equivalent to 180x180 pixels in area
      // w is total number of pixels to plot to fill character block
      while (pc < w) {
        line = pgm_read_byte((uint8_t *)flash_address);
        flash_address++;
        if (line & 0x80) {
          line &= 0x7F;
          line++;
          if (ts) {
            px = xd + textsize * (pc % width); // Keep these px and py calculations outside the loop as they are slow
            py = yd + textsize * (pc / width);
          }
          else {
            px = xd + pc % width; // Keep these px and py calculations outside the loop as they are slow
            py = yd + pc / width;
```

```
        while (line--) { // In this case the while(line--) is faster
          pc++; // This is faster than putting pc+=line before while()?
          setWindow(px, py, px + ts, py + ts);

          if (ts) {
            tnp = np;
            while (tnp--) writeColor(color);
          }
          else writeColor(color);

          px += textsize;

          if (px >= (xd + width * textsize)) {
            px = xd;
            py += textsize;
          }
        }
      }
      else {
        line++;
        pc += line;
      }
    }
  }
  else {
    // Text colour != background and textsize = 1 and character is within viewport area
    // so use faster drawing of characters and background using block write
    if (textcolor != textbgcolor && textsize == 1 && !clip && _bpp != 1)
    {
      setWindow(xd, yd, xd + width - 1, yd + height - 1);

      // Maximum font size is equivalent to 180x180 pixels in area
      while (w > 0) {
        line = pgm_read_byte((uint8_t *)flash_address++); // 8 bytes smaller when incrementing here
        if (line & 0x80) {
          line &= 0x7F;
          line++; w -= line;
          while (line--) writeColor(color);
        }
        else {
          line++; w -= line;
          while (line--) writeColor(bgcolor);
        }
      }
    }
    else
    {
      int32_t px = 0, py = 0;  // To hold character pixel coords
      int32_t tx = 0, ty = 0;  // To hold character TFT pixel coords
      int32_t pc = 0;          // Pixel count
      int32_t pl = 0;          // Pixel line length
      uint16_t pcol = 0;       // Pixel color
      bool     pf = true;      // Flag for plotting
      while (pc < w) {
        line = pgm_read_byte((uint8_t *)flash_address);
        flash_address++;
        if (line & 0x80) { pcol = textcolor; line &= 0x7F; pf = true;}
        else { pcol = textbgcolor; if (textcolor == textbgcolor) pf = false;}
        line++;
        px = pc % width;
        tx = x + textsize * px;
        py = pc / width;
        ty = y + textsize * py;

        pl = 0;
        pc += line;
        while (line--) {
          pl++;
          if ((px+pl) >= width) {
            if (pf) fillRect(tx, ty, pl * textsize, textsize, pcol);
            pl = 0;
            px = 0;
            tx = x;
            py ++;
            ty += textsize;
          }
        }
        if (pl && pf) fillRect(tx, ty, pl * textsize, textsize, pcol);
      }
    }
  }
 }
 // End of RLE font rendering
#endif
```

```
}

#ifdef SMOOTH_FONT
```

```
/***************************************************************************************
** Function name:           drawGlyph
** Description:             Write a character to the sprite cursor position
***************************************************************************************/
//
void TFT_eSprite::drawGlyph(uint16_t code)
{
  uint16_t fg = textcolor;
  uint16_t bg = textbgcolor;

  if (code < 0x21)
  {
    if (code == 0x20) {
      cursor_x += gFont.spaceWidth;
      return;
    }

    if (code == '\n') {
      cursor_x = 0;
      cursor_y += gFont.yAdvance;
      if (textwrapY && (cursor_y >= height())) cursor_y = 0;
      return;
    }
  }

  uint16_t gNum = 0;
  bool found = getUnicodeIndex(code, &gNum);

  if (found)
  {

    bool newSprite = !_created;

    if (newSprite)
    {
      createSprite(gWidth[gNum], gFont.yAdvance);
      if(fg != bg) fillSprite(bg);
      cursor_x = -gdX[gNum];
      cursor_y = 0;
    }
    else
    {
      if( textwrapX && ((cursor_x + gWidth[gNum] + gdX[gNum]) > width())) {
        cursor_y += gFont.yAdvance;
        cursor_x = 0;
      }

      if( textwrapY && ((cursor_y + gFont.yAdvance) > height())) cursor_y = 0;

      if ( cursor_x == 0) cursor_x -= gdX[gNum];
    }

    uint8_t* pbuffer = nullptr;
    const uint8_t* gPtr = (const uint8_t*) gFont.gArray;

#ifdef FONT_FS_AVAILABLE
    if (fs_font) {
      fontFile.seek(gBitmap[gNum], fs::SeekSet); // This is slow for a significant position shift!
      pbuffer =  (uint8_t*)malloc(gWidth[gNum]);
    }
#endif

    int16_t  xs = 0;
    uint16_t dl = 0;
    uint8_t pixel = 0;
    int32_t cgy = cursor_y + gFont.maxAscent - gdY[gNum];
    int32_t cgx = cursor_x + gdX[gNum];

    for (int32_t y = 0; y < gHeight[gNum]; y++)
    {
#ifdef FONT_FS_AVAILABLE
      if (fs_font) {
        fontFile.read(pbuffer, gWidth[gNum]);
      }
#endif
      for (int32_t x = 0; x < gWidth[gNum]; x++)
      {
#ifdef FONT_FS_AVAILABLE
        if (fs_font) {
          pixel = pbuffer[x];
        }
        else
#endif
        pixel = pgm_read_byte(gPtr + gBitmap[gNum] + x + gWidth[gNum] * y);
```

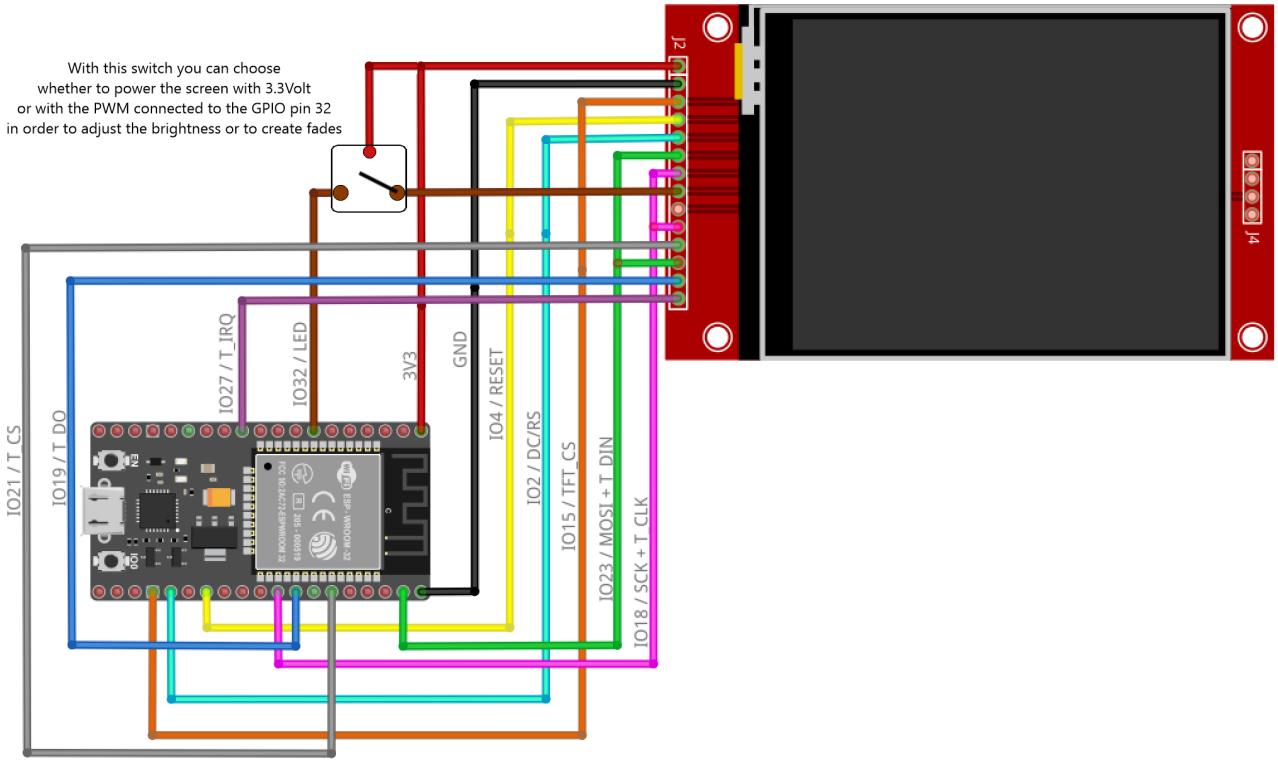
```
        if (pixel)
        {
          if (pixel != 0xFF)
          {
            if (dl) { drawFastHLine( xs, y + cgy, dl, fg); dl = 0; }
            if (_bpp != 1) {
              if (fg == bg) drawPixel(x + cgx, y + cgy, alphaBlend(pixel, fg, readPixel(x + cgx, y + cgy)));
              else drawPixel(x + cgx, y + cgy, alphaBlend(pixel, fg, bg));
            }
            else if (pixel>127) drawPixel(x + cgx, y + cgy, fg);
          }
          else
          {
            if (dl==0) xs = x + cgx;
            dl++;
          }
        }
        else
        {
          if (dl) { drawFastHLine( xs, y + cgy, dl, fg); dl = 0; }
        }
      }
      if (dl) { drawFastHLine( xs, y + cgy, dl, fg); dl = 0; }
    }

    if (pbuffer) free(pbuffer);

    if (newSprite)
    {
      pushSprite(cgx, cursor_y);
      deleteSprite();
    }
    cursor_x += gxAdvance[gNum];
  }
  else
  {
    // Not a Unicode in font so draw a rectangle and move on cursor
    drawRect(cursor_x, cursor_y + gFont.maxAscent - gFont.ascent, gFont.spaceWidth, gFont.ascent, fg);
    cursor_x += gFont.spaceWidth + 1;
  }
}
```

```
/*****************************************************************************
** Function name:           printToSprite
** Description:             Write a string to the sprite cursor position
*****************************************************************************/
void TFT_eSprite::printToSprite(String string)
{
  if(!fontLoaded) return;
  printToSprite((char*)string.c_str(), string.length());
}
```

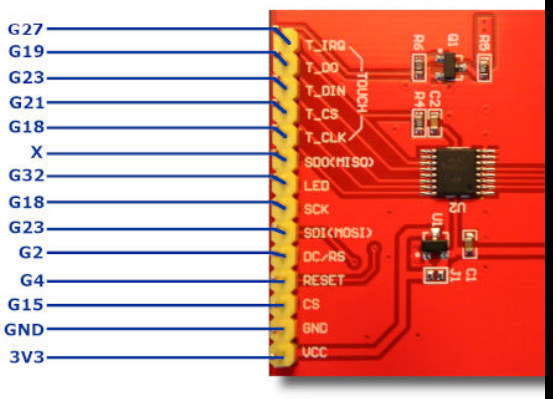
```
/**************************************************************************************
** Function name:           printToSprite
** Description:             Write a string to the sprite cursor position
**************************************************************************************/
void TFT_eSprite::printToSprite(char *cbuffer, uint16_t len) //String string)
{
  if(!fontLoaded) return;

  uint16_t n = 0;
  bool newSprite = !_created;

  if (newSprite)
  {
    int16_t sWidth = 1;
    uint16_t index = 0;

    while (n < len)
    {
      uint16_t unicode = decodeUTF8((uint8_t*)cbuffer, &n, len - n);
      if (getUnicodeIndex(unicode, &index))
      {
        if (n == 0) sWidth -= gdX[index];
        if (n == len-1) sWidth += ( gWidth[index] + gdX[index]);
        else sWidth += gxAdvance[index];
      }
      else sWidth += gFont.spaceWidth + 1;
    }

    createSprite(sWidth, gFont.yAdvance);

    if (textcolor != textbgcolor) fillSprite(textbgcolor);
  }

  n = 0;

  while (n < len)
  {
    uint16_t unicode = decodeUTF8((uint8_t*)cbuffer, &n, len - n);
    //Serial.print("Decoded Unicode = 0x");Serial.println(unicode,HEX);
    //Serial.print("n = ");Serial.println(n);
    drawGlyph(unicode);
  }

  if (newSprite)
  { // The sprite had to be created so place at TFT cursor
    pushSprite(_tft->cursor_x, _tft->cursor_y);
    deleteSprite();
  }
}
```

```
/**************************************************************************************
** Function name:           printToSprite
** Description:             Print character in a Sprite, create sprite if needed
**************************************************************************************/
int16_t TFT_eSprite::printToSprite(int16_t x, int16_t y, uint16_t index)
{
  bool newSprite = !_created;
  int16_t sWidth = gWidth[index];

  if (newSprite)
  {
    createSprite(sWidth, gFont.yAdvance);

    if (textcolor != textbgcolor) fillSprite(textbgcolor);

    drawGlyph(gUnicode[index]);

    pushSprite(x + gdX[index], y, textbgcolor);
    deleteSprite();
  }

  else drawGlyph(gUnicode[index]);

  return gxAdvance[index];
}
#endif
```

```
/**********************************************************************************************
** Function name:           CONNECTIONS
** Description:             Diagrams
**********************************************************************************************/
```

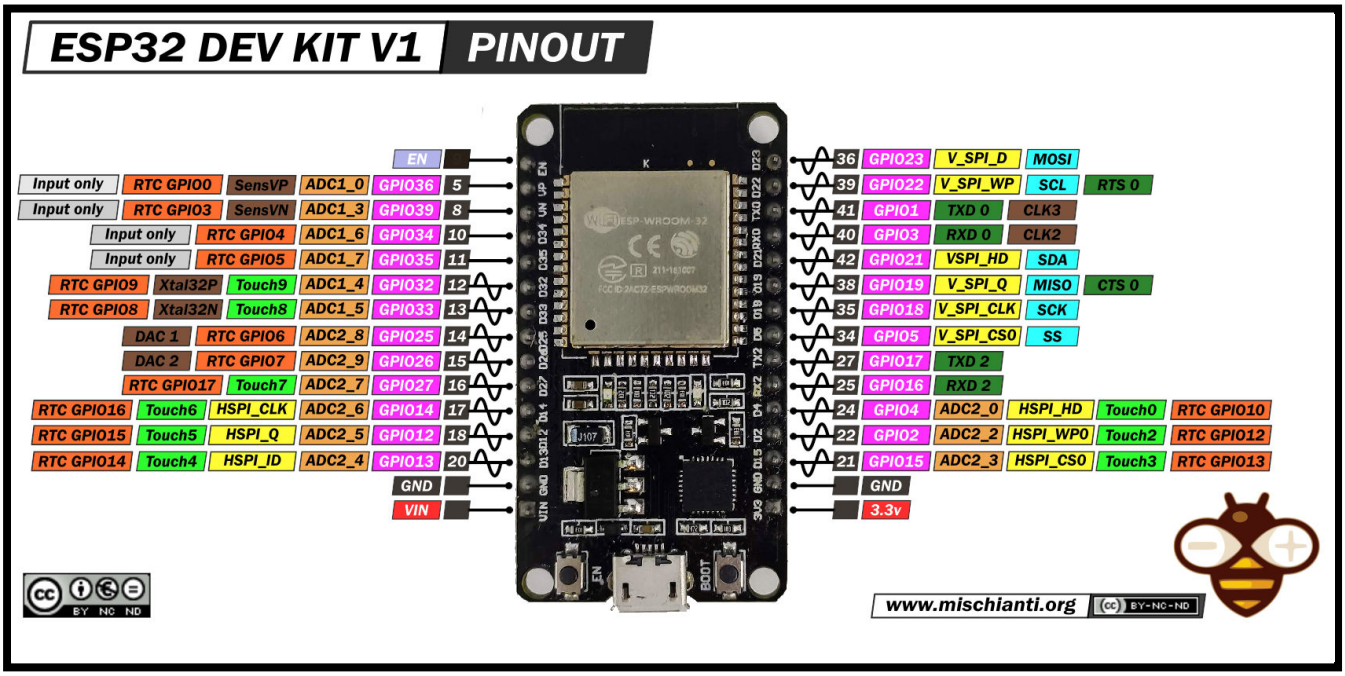


TFT Pin	ESP32 Pin
VCC	3V3
GND	GND
CS	GPIO15
RESET	GPIO4
DC/RS	GPIO2
SDI (MOSI)	GPIO23
SCK	GPIO18
LED	GPIO32
SDO (MISO)	N.C.
T_CLK	GPIO18
T_CS	GPIO21
T_DIN	GPIO23
T_DO	GPIO19
T_IRQ	GPIO27



If you use PWM to GPIO 32, you need put this lines in SETUP of sketch:

```
ledcSetup(0, 5000, 8);
#ifdef TFT_BL
  ledcAttachPin(TFT_BL, 0);
#else
  ledcAttachPin(32, 0);
#endif
  ledcWrite(0, 255); // Start @ initial Brightnes
```



```
/**********************************************************************************
** Function name:           SETUP FILES
** Description:             Setup Files
**********************************************************************************/

>>>>>>>>>>>>> User_Setup.h <<<<<<<<<<<

#define ILI9488_DRIVER     // WARNING: Do not connect ILI9488 display SDO to MISO if other devices share the S
#define TFT_BL   32            // LED back-light control pin
#define TFT_BACKLIGHT_ON HIGH  // Level to turn ON back-light (HIGH or LOW)
#define TFT_MISO 19
#define TFT_MOSI 23
#define TFT_SCLK 18
#define TFT_CS   15  // Chip select control pin
#define TFT_DC    2  // Data Command control pin
#define TFT_RST   4  // Reset pin (could connect to RST pin)
#define TOUCH_CS 21     // Chip select pin (T_CS) of touch screen
#define LOAD_GLCD   // Font 1. Original Adafruit 8 pixel font needs ~1820 bytes in FLASH
#define LOAD_FONT2  // Font 2. Small 16 pixel high font, needs ~3534 bytes in FLASH, 96 characters
#define LOAD_FONT4  // Font 4. Medium 26 pixel high font, needs ~5848 bytes in FLASH, 96 characters
#define LOAD_FONT6  // Font 6. Large 48 pixel font, needs ~2666 bytes in FLASH, only characters 1234567890:-.a
#define LOAD_FONT7  // Font 7. 7 segment 48 pixel font, needs ~2438 bytes in FLASH, only characters 12345678
#define LOAD_FONT8  // Font 8. Large 75 pixel font needs ~3256 bytes in FLASH, only characters 1234567890:-.
#define LOAD_GFXFF  // FreeFonts. Include access to the 48 Adafruit_GFX free fonts FF1 to FF48 and custom fonts
#define SMOOTH_FONT

// Define the SPI clock frequency, this affects the graphics rendering speed. Too
// fast and the TFT driver will not keep up and display corruption appears.
// With an ILI9341 display 40MHz works OK, 80MHz sometimes fails
// With a ST7735 display more than 27MHz may not work (spurious pixels and lines)
// With an ILI9163 display 27 MHz works OK.

// #define SPI_FREQUENCY   1000000
// #define SPI_FREQUENCY   5000000
// #define SPI_FREQUENCY  10000000
// #define SPI_FREQUENCY  20000000
#define SPI_FREQUENCY  27000000
 //#define SPI_FREQUENCY  40000000
// #define SPI_FREQUENCY  55000000 // STM32 SPI1 only (SPI2 maximum is 27MHz)
// #define SPI_FREQUENCY  80000000

// Optional reduced SPI frequency for reading TFT
//#define SPI_READ_FREQUENCY  20000000
#define SPI_READ_FREQUENCY  16000000

// The XPT2046 requires a lower SPI clock rate of 2.5MHz so we define that here:
#define SPI_TOUCH_FREQUENCY  2500000

// The ESP32 has 2 free SPI ports i.e. VSPI and HSPI, the VSPI is the default.
// If the VSPI port is in use and pins are not accessible (e.g. TTGO T-Beam)
// then uncomment the following line:
//#define USE_HSPI_PORT
```

```
/*****************************************************************************
** Function name:           User_Setup_Select.h
** Description:             Setup Files
*****************************************************************************/

// Customised User_Setup files are stored in the "User_Setups" folder.

#ifndef USER_SETUP_LOADED //  Lets PlatformIO users define settings in
                          //  platformio.ini, see notes in "Tools" folder.

// Only ONE line below should be uncommented.  Add extra lines and files as needed.

//#include <User_Setup.h>           // Default setup is root library folder
#include <User_Setups/Setup21_ILI9488.h>          // Setup file for ESP32 and ILI9488 SPI bus TFT
#endif // USER_SETUP_LOADED

//////////////////////////////////////////////////////////////////////////////////
//                                                                              //
//     DON'T TINKER WITH ANY OF THE FOLLOWING LINES, THESE ADD THE TFT DRIVERS     //
//       AND ESP8266 PIN DEFINITONS, THEY ARE HERE FOR BODMER'S CONVENIENCE!       //
//                                                                              //
//////////////////////////////////////////////////////////////////////////////////

// Identical looking TFT displays may have a different colour ordering in the 16 bit colour
#define TFT_BGR 0   // Colour order Blue-Green-Red
#define TFT_RGB 1   // Colour order Red-Green-Blue

// Legacy setup support, RPI_DISPLAY_TYPE replaces RPI_DRIVER
#if defined (RPI_DRIVER)
  #if !defined (RPI_DISPLAY_TYPE)
    #define RPI_DISPLAY_TYPE
  #endif
#endif

// Invoke 18 bit colour for selected displays
#if !defined (RPI_DISPLAY_TYPE) && !defined (TFT_PARALLEL_8_BIT) && !defined (ESP32_PARALLEL)
  #if defined (ILI9481_DRIVER) || defined (ILI9486_DRIVER) || defined (ILI9488_DRIVER)
    #define SPI_18BIT_DRIVER
  #endif
#endif

// Load the right driver definition - do not tinker here !
#if   defined (ILI9341_DRIVER) || defined(ILI9341_2_DRIVER)
    #include <TFT_Drivers/ILI9341_Defines.h>
    #define  TFT_DRIVER 0x9341
#elif defined (ILI9488_DRIVER)
    #include <TFT_Drivers/ILI9488_Defines.h>
    #define  TFT_DRIVER 0x9488
                        // <<<<<<<<<<<<<<<<<<<<<<<<< ADD NEW DRIVER HERE
                        // XYZZY_init.h and XYZZY_rotation.h must also be added in TFT_eSPI.cpp
#elif defined (XYZZY_DRIVER)
    #include "TFT_Drivers/XYZZY_Defines.h"
    #define  TFT_DRIVER 0x0000
#else
    #define  TFT_DRIVER 0x0000
#endif
```

```
/**********************************************************************************
** Function name:           Setup21_ILI9488.h
** Description:             Setup Files
**********************************************************************************/

#define ILI9488_DRIVER
//#define TFT_INVERSION_OFF
#define TFT_MISO 19 // (leave TFT SDO disconnected if other SPI devices share MISO)
#define TFT_MOSI 23
#define TFT_SCLK 18
#define TFT_CS    15  // Chip select control pin
#define TFT_DC    2  // Data Command control pin
#define TFT_RST   4  // Reset pin (could connect to RST pin)
#define  TOUCH_CS 21
#define LOAD_GLCD   // Font 1. Original Adafruit 8 pixel font needs ~1820 bytes in FLASH
#define LOAD_FONT2  // Font 2. Small 16 pixel high font, needs ~3534 bytes in FLASH, 96 characters
#define LOAD_FONT4  // Font 4. Medium 26 pixel high font, needs ~5848 bytes in FLASH, 96 characters
#define LOAD_FONT6  // Font 6. Large 48 pixel font, needs ~2666 bytes in FLASH, only characters 1234567890:-.a
#define LOAD_FONT7  // Font 7. 7 segment 48 pixel font, needs ~2438 bytes in FLASH, only characters 12345678
#define LOAD_FONT8  // Font 8. Large 75 pixel font needs ~3256 bytes in FLASH, only characters 1234567890:-.
#define LOAD_GFXFF  // FreeFonts. Include access to the 48 Adafruit_GFX free fonts FF1 to FF48 and custom fonts
#define SMOOTH_FONT
// Define the SPI clock frequency, this affects the graphics rendering speed. Too
// fast and the TFT driver will not keep up and display corruption appears.
// With an ILI9341 display 40MHz works OK, 80MHz sometimes fails
// With a ST7735 display more than 27MHz may not work (spurious pixels and lines)
// With an ILI9163 display 27 MHz works OK.

// #define SPI_FREQUENCY   1000000
// #define SPI_FREQUENCY   5000000
// #define SPI_FREQUENCY  10000000
// #define SPI_FREQUENCY  20000000
#define SPI_FREQUENCY  27000000
 //#define SPI_FREQUENCY  40000000
// #define SPI_FREQUENCY  55000000 // STM32 SPI1 only (SPI2 maximum is 27MHz)
// #define SPI_FREQUENCY  80000000

// Optional reduced SPI frequency for reading TFT
//#define SPI_READ_FREQUENCY  20000000
#define SPI_READ_FREQUENCY  16000000

// The XPT2046 requires a lower SPI clock rate of 2.5MHz so we define that here:
#define SPI_TOUCH_FREQUENCY  2500000

// The ESP32 has 2 free SPI ports i.e. VSPI and HSPI, the VSPI is the default.
// If the VSPI port is in use and pins are not accessible (e.g. TTGO T-Beam)
// then uncomment the following line:
//#define USE_HSPI_PORT
```