```
/*****************************************************
  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)
  #if defined(CONFIG_IDF_TARGET_ESP32S3)
    #include "Processors/TFT_eSPI_ESP32_S3.c" // Tested with SPI and 8 bit parallel
  #elif defined(CONFIG_IDF_TARGET_ESP32C3)
    #include "Processors/TFT_eSPI_ESP32_C3.c" // Tested with SPI (8 bit parallel will probably work too!)
  #else
    #include "Processors/TFT_eSPI_ESP32.c"
  #endif
#elif defined (ARDUINO_ARCH_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)   || defined (ARDUINO_ARCH_MBED) // 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:           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:           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 (locked) {
    locked = false; // Flag to show SPI access now unlocked
#if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS)
&& !defined(TFT_PARALLEL_8_BIT) && !defined(RP2040_PIO_INTERFACE)
    spi.beginTransaction(SPISettings(SPI_FREQUENCY, MSBFIRST, TFT_SPI_MODE));
#endif
    CS_L;
    SET_BUS_WRITE_MODE;   // Some processors (e.g. ESP32) allow recycling the tx buffer when rx is not used
  }
}

// Non-inlined version to permit override
void TFT_eSPI::begin_nin_write(void){
  if (locked) {
    locked = false; // Flag to show SPI access now unlocked
#if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS)
&& !defined(TFT_PARALLEL_8_BIT) && !defined(RP2040_PIO_INTERFACE)
    spi.beginTransaction(SPISettings(SPI_FREQUENCY, MSBFIRST, TFT_SPI_MODE));
#endif
    CS_L;
```

```
    SET_BUS_WRITE_MODE;   // Some processors (e.g. ESP32) allow recycling the tx buffer when rx is not used
  }
}

/***************************************************************************************
** 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(!inTransaction) {       // Flag to stop ending transaction 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;
      SET_BUS_READ_MODE;     // In case bus has been configured for tx only
#if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS) && !defined(TFT_PARALLEL_8_BIT) && !defined(RP2040_PIO_INTERFACE)
      spi.endTransaction();
#endif
    }
  }
}

// Non-inlined version to permit override
inline void TFT_eSPI::end_nin_write(void){
  if(!inTransaction) {       // Flag to stop ending transaction 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;
      SET_BUS_READ_MODE;     // In case SPI has been configured for tx only
#if defined (SPI_HAS_TRANSACTION) && defined (SUPPORT_TRANSACTIONS) && !defined(TFT_PARALLEL_8_BIT) && !defined(RP2040_PIO_INTERFACE)
      spi.endTransaction();
#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) && !defined(RP2040_PIO_INTERFACE)
  if (locked) {
    locked = false;
    spi.beginTransaction(SPISettings(SPI_READ_FREQUENCY, MSBFIRST, TFT_SPI_MODE));
    CS_L;
  }
#else
  #if !defined(TFT_PARALLEL_8_BIT) && !defined(RP2040_PIO_INTERFACE)
    spi.setFrequency(SPI_READ_FREQUENCY);
  #endif
   CS_L;
#endif
  SET_BUS_READ_MODE;
}


/***************************************************************************************
** 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) && !defined(RP2040_PIO_INTERFACE)
  if(!inTransaction) {
    if (!locked) {
      locked = true;
      CS_H;
      spi.endTransaction();
    }
  }
#else
  #if !defined(TFT_PARALLEL_8_BIT) && !defined(RP2040_PIO_INTERFACE)
    spi.setFrequency(SPI_FREQUENCY);
  #endif
   if(!inTransaction) {CS_H;}
#endif
  SET_BUS_WRITE_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 whole 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:           clipAddrWindow
** Description:             Clip address window x,y,w,h to screen and viewport
***************************************************************************************/
bool TFT_eSPI::clipAddrWindow(int32_t *x, int32_t *y, int32_t *w, int32_t *h)
{
  if (_vpOoB) return false; // Area is outside of viewport

  *x+= _xDatum;
  *y+= _yDatum;

  if ((*x >= _vpW) || (*y >= _vpH)) return false;   // Area is outside of viewport

  // Crop drawing area bounds
  if (*x < _vpX) { *w -= _vpX - *x; *x = _vpX; }
  if (*y < _vpY) { *h -= _vpY - *y; *y = _vpY; }

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

  if (*w < 1 || *h < 1) return false; // No area is inside viewport

  return true;   // Area is wholly or partially inside viewport
}

/***************************************************************************************
** Function name:           clipWindow
** Description:             Clip window xs,yx,xe,ye to screen and viewport
***************************************************************************************/
bool TFT_eSPI::clipWindow(int32_t *xs, int32_t *ys, int32_t *xe, int32_t *ye)
{
  if (_vpOoB) return false; // Area is outside of viewport

  *xs+= _xDatum;
  *ys+= _yDatum;
  *xe+= _xDatum;
  *ye+= _yDatum;

  if ((*xs >= _vpW) || (*ys >= _vpH)) return false;   // Area is outside of viewport
  if ((*xe <   _vpX) || (*ye <   _vpY)) return false;   // Area is outside of viewport

  // Crop drawing area bounds
  if (*xs < _vpX) *xs = _vpX;
  if (*ys < _vpY) *ys = _vpY;

  if (*xe > _vpW) *xe = _vpW - 1;
  if (*ye > _vpH) *ye = _vpH - 1;

  return true;   // Area is wholly or partially inside viewport
}

/***************************************************************************************
** Function name:           TFT_eSPI
** Description:             Constructor , we must use hardware SPI pins
***************************************************************************************/
TFT_eSPI::TFT_eSPI(int16_t w, int16_t h)
{
  _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  = last_cursor_x = bg_cursor_x = 0;
  textfont  = 1;
  textsize  = 1;
  textcolor   = bitmap_fg = 0xFFFF; // White
  textbgcolor = bitmap_bg = 0x0000; // Black
  padX        = 0;                  // No padding

  _fillbg     = false;    // Smooth font only at the moment, force text background fill

  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
  _cp437     = true;       // Legacy GLCD font bug fix
  _utf8      = true;       // UTF8 decoding enabled

#if defined (FONT_FS_AVAILABLE) && defined (SMOOTH_FONT)
  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:           initBus
** Description:             initialise the SPI or parallel bus
***************************************************************************************/
void TFT_eSPI::initBus(void) {

#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

// In parallel mode and with the RP2040 processor, the TFT_WR line is handled in the   PIO
#if defined (TFT_WR) && !defined (ARDUINO_ARCH_RP2040) && !defined
(ARDUINO_ARCH_MBED)
  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);

  #if   !defined (ARDUINO_ARCH_RP2040)   && !defined (ARDUINO_ARCH_MBED)// PIO
manages pins
    // 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);
  #endif

  PARALLEL_INIT_TFT_DATA_BUS;

#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)
  {
    initBus();

#if !defined (ESP32) && !defined(TFT_PARALLEL_8_BIT) && !defined(ARDUINO_ARCH_RP2040)
&& !defined (ARDUINO_ARCH_MBED)
  // 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 (ARDUINO_ARCH_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) && !defined(RP2040_PIO_INTERFACE)
    #if defined (TFT_MOSI) && !defined (TFT_SPI_OVERLAP)
&& !defined(ARDUINO_ARCH_RP2040) && !defined (ARDUINO_ARCH_MBED)
      spi.begin(TFT_SCLK, TFT_MISO, TFT_MOSI, -1); // This will set MISO to input
    #else
      spi.begin(); // This will set MISO to input
    #endif
  #endif
#endif
    lockTransaction = false;
    inTransaction = false;
    locked = true;

    INIT_TFT_DATA_BUS;


#if defined (TFT_CS) && !defined(RP2040_PIO_INTERFACE)
  // Set to output once again in case MISO is used for CS
  pinMode(TFT_CS, OUTPUT);
  digitalWrite(TFT_CS, HIGH); // Chip select high (inactive)
#elif defined (ARDUINO_ARCH_ESP8266) && !defined (TFT_PARALLEL_8_BIT) && !defined
(RP2040_PIO_SPI)
  spi.setHwCs(1); // Use hardware SS toggling
#endif


  // Set to output once again in case MISO is used for DC
#if defined (TFT_DC) && !defined(RP2040_PIO_INTERFACE)
    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 !defined(RP2040_PIO_INTERFACE)
    // Set to output once again in case MISO is used for TFT_RST
    pinMode(TFT_RST, OUTPUT);
  #endif
  if (TFT_RST >= 0) {
    writecommand(0x00); // Put SPI bus in known state for TFT with CS tied low
    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
#endif

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

  begin_tft_write();

  tc = tc; // Suppress warning

  // This loads the driver specific initialisation code  <<<<<<<<<<<<<<<<<<<<<< ADD NEW
DRIVERS TO THE LIST HERE <<<<<<<<<<<<<<<<<<<<<<<
#if    defined (ILI9341_DRIVER) || defined(ILI9341_2_DRIVER) || defined (ILI9342_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"

#elif defined (RM68120_DRIVER)
     #include "TFT_Drivers/RM68120_Init.h"

#elif defined (HX8357B_DRIVER)
    #include "TFT_Drivers/HX8357B_Init.h"

#elif defined (HX8357C_DRIVER)
    #include "TFT_Drivers/HX8357C_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 HERE <<<<<<<<<<<<<<<<<<<<<<<
#if    defined (ILI9341_DRIVER) || defined(ILI9341_2_DRIVER) || defined (ILI9342_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"

#elif defined (RM68120_DRIVER)
     #include "TFT_Drivers/RM68120_Rotation.h"

#elif defined (HX8357B_DRIVER)
    #include "TFT_Drivers/HX8357B_Rotation.h"

#elif defined (HX8357C_DRIVER)
    #include "TFT_Drivers/HX8357C_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
***************************************************************************************/
#ifndef RM68120_DRIVER
void TFT_eSPI::writecommand(uint8_t c)
{
  begin_tft_write();

  DC_C;

  tft_Write_8(c);

  DC_D;

  end_tft_write();

}
#else
void TFT_eSPI::writecommand(uint16_t c)
{
  begin_tft_write();

  DC_C;

  tft_Write_16(c);

  DC_D;

  end_tft_write();

}
void TFT_eSPI::writeRegister(uint16_t c, uint8_t d)
{
  begin_tft_write();

  DC_C;

  tft_Write_16(c);

  DC_D;

  tft_Write_8(d);

  end_tft_write();

}
```

```
#endif

/***************************************************************************************
** 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;
#if defined(TFT_PARALLEL_8_BIT) || defined(RP2040_PIO_INTERFACE)

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

  busDir(GPIO_DIR_MASK, INPUT);

  CS_L;

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

  busDir(GPIO_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) || defined(RP2040_PIO_INTERFACE)

  if (!inTransaction) { CS_L; } // CS_L can be multi-statement

  readAddrWindow(x0, y0, 1, 1);

  // Set masked pins D0- D7 to input
  busDir(GPIO_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_DRIVER)// Read 3 bytes

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

    if (!inTransaction) { CS_H; } // CS_H can be multi-statement

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

    return rgb;

  #else // ILI9481 or ILI9486 16 bit read

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

    if (!inTransaction) { CS_H; } // CS_H can be multi-statement

    // Set masked pins D0- D7 to output
    busDir(GPIO_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 anti-aliased 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(); // Sets CS low

  readAddrWindow(x0, y0, 1, 1);

  #ifdef TFT_SDA_READ
    begin_SDA_Read();
  #endif

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

  //#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 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) || defined(RP2040_PIO_INTERFACE)

  CS_L;

  readAddrWindow(x, y, dw, dh);

  data += dx + dy * w;

  // Set masked pins D0- D7 to input
  busDir(GPIO_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(GPIO_DIR_MASK, OUTPUT);

#else // SPI interface

  // 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;

  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, sx = x;
    bool move = true;
    uint16_t np = 0;

    while (len--)
    {
      if (transp != *ptr)
      {
        if (move) { move = false; sx = px; }
        lineBuf[np] = *ptr;
        np++;
      }
      else
      {
        move = true;
        if (np)
        {
          setWindow(sx, y, sx + np - 1, y);
          pushPixels((uint16_t*)lineBuf, np);
          np = 0;
        }
      }
      px++;
      ptr++;
    }
    if (np) { setWindow(sx, y, sx + np - 1, y); 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;


  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, sx = x;
    bool move = true;

    uint16_t np = 0;

    while (len--) {
      uint16_t color = pgm_read_word(ptr);
      if (transp != color) {
        if (move) { move = false; sx = px; }
        lineBuf[np] = color;
        np++;
      }
      else {
        move = true;
        if (np) {
          setWindow(sx, y, sx + np - 1, y);
          pushPixels(lineBuf, np);
          np = 0;
        }
      }
      px++;
      ptr++;
    }
    if (np) { setWindow(sx, y, sx + np - 1, y); 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 *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 = (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 / image

  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 / image

    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, uint16_t *cmap)
{
  PI_CLIP;

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


  // 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, sx = x;
      bool move = true;
      uint16_t np = 0;
```

```
    while (len--) {
      if (transp != *ptr) {
        if (move) { move = false; sx = px; }
        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) {
          setWindow(sx, y, sx + np - 1, y);
          pushPixels(lineBuf, np);
          linePtr = (uint8_t*)lineBuf;
          np = 0;
        }
      }
      px++;
      ptr++;
    }

    if (np) { setWindow(sx, y, sx + np - 1, y); 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, sx = 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; sx = px;
        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; sx = px;
        }
        lineBuf[np] = cmap[index];
        np++; // added a pixel
      }
      else {
        move = true;
        if (np) {
          setWindow(sx, y, sx + np - 1, y);
          pushPixels(lineBuf, np);
          np = 0;
        }
      }
      px++;

      if (len--)
      {
        index = color & 0x0F; // the odd number is 3 .. 0
        if (index != transp) {
```

```
            if (move) {
              move = false; sx = px;
             }
            lineBuf[np] = cmap[index];
            np++;
          }
          else {
            move = true;
            if (np) {
              setWindow(sx, y, sx + np - 1, y);
              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) {
        setWindow(sx, y, sx + np - 1, y);
        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;

    for (int32_t yp = dy;   yp < dy + dh; yp++)
    {
      int32_t px = x, sx = x;
      bool move = true;
      for (int32_t xp = dx; xp < dx + dw; xp++)
      {
        if (data[(xp>>3)] & (0x80 >> (xp & 0x7))) {
          if (move) {
            move = false;
            sx = px;
          }
          np++;
        }
        else {
          move = true;
          if (np) {
            setWindow(sx, y, sx + np - 1, y);
            pushBlock(bitmap_fg, np);
            np = 0;
          }
        }
        px++;
      }
      if (np) { setWindow(sx, y, sx + np - 1, y); pushBlock(bitmap_fg, np); np = 0; }
      y++;
      data += ww;
    }
  }
  _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) || defined(RP2040_PIO_INTERFACE)

  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, uint16_t bgcolor)
{
  //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, uint16_t bgcolor)
{
  //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
***************************************************************************************/
// Smooth fonts use the background colour for anti-aliasing and by default the
// background is not filled. If bgfill = true, then a smooth font background fill will
// be used.
void TFT_eSPI::setTextColor(uint16_t c, uint16_t b, bool bgfill)
{
  textcolor   = c;
  textbgcolor = b;
  _fillbg     = bgfill;
}


/***************************************************************************************
** 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 loop

    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
      while (*string++) str_width += 6;
#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;

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

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

  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+7);

    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
#else
  #ifndef LOAD_GFXFF
    // Avoid warnings if fonts are disabled
    x = x;
    y = y;
    color = color;
    bg = bg;
    size = size;
  #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;
  // Temporary solution is to include the RP2040 code here
  #if (defined(ARDUINO_ARCH_RP2040)   || defined (ARDUINO_ARCH_MBED))
&& !defined(RP2040_PIO_INTERFACE)
    // For ILI9225 and RP2040 the slower Arduino SPI transfer calls were used, so need to swap
back to 16 bit mode
    while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
    hw_write_masked(&spi_get_hw(SPI_X)->cr0, (16 - 1) << SPI_SSPCR0_DSS_LSB,
SPI_SSPCR0_DSS_BITS);
  #endif
#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 (ARDUINO_ARCH_MBED))
    #if !defined(RP2040_PIO_INTERFACE)
      // Use hardware SPI port, this code does not swap from 8 to 16 bit
      // to avoid the spi_set_format() call overhead
      while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
      DC_C;
      #if !defined (SPI_18BIT_DRIVER)
        #if   defined (RPI_DISPLAY_TYPE) // RPi TFT type always needs 16 bit transfers
          hw_write_masked(&spi_get_hw(SPI_X)->cr0, (16 - 1) << SPI_SSPCR0_DSS_LSB,
SPI_SSPCR0_DSS_BITS);
        #else
          hw_write_masked(&spi_get_hw(SPI_X)->cr0, (8 - 1) << SPI_SSPCR0_DSS_LSB,
SPI_SSPCR0_DSS_BITS);
        #endif
      #endif
      spi_get_hw(SPI_X)->dr = (uint32_t)TFT_CASET;

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

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

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

      while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
      DC_C;
      spi_get_hw(SPI_X)->dr = (uint32_t)TFT_RAMWR;
```

```
      while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
      #if !defined (SPI_18BIT_DRIVER)
        hw_write_masked(&spi_get_hw(SPI_X)->cr0, (16 - 1) << SPI_SSPCR0_DSS_LSB,
SPI_SSPCR0_DSS_BITS);
      #endif
      DC_D;
    #else
      // This is for the RP2040 and PIO interface (SPI or parallel)
      WAIT_FOR_STALL;
      tft_pio->sm[pio_sm].instr = pio_instr_addr;

      TX_FIFO = TFT_CASET;
      TX_FIFO = (x0<<16) | x1;
      TX_FIFO = TFT_PASET;
      TX_FIFO = (y0<<16) | y1;
      TX_FIFO = TFT_RAMWR;
    #endif
  #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;

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

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

  // Temporary solution is to include the RP2040 optimised code here
#if (defined(ARDUINO_ARCH_RP2040)  || defined (ARDUINO_ARCH_MBED))
&& !defined(RP2040_PIO_INTERFACE)
  // Use hardware SPI port, this code does not swap from 8 to 16 bit
  // to avoid the spi_set_format() call overhead
  while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
  DC_C;
  hw_write_masked(&spi_get_hw(SPI_X)->cr0, (8 - 1) << SPI_SSPCR0_DSS_LSB,
SPI_SSPCR0_DSS_BITS);
  spi_get_hw(SPI_X)->dr = (uint32_t)TFT_CASET;

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

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

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

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

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

  // Flush the rx buffer and reset overflow flag
  while (spi_is_readable(SPI_X)) (void)spi_get_hw(SPI_X)->dr;
  spi_get_hw(SPI_X)->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

#if (defined (MULTI_TFT_SUPPORT) || defined (GC9A01_DRIVER)) && !defined
(ILI9225_DRIVER)
  addr_row = 0xFFFF;
  addr_col = 0xFFFF;
#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(TFT_PARALLEL_16_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) || defined (ARDUINO_ARCH_MBED)) && !defined
(SSD1351_DRIVER)

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

  #if !defined(RP2040_PIO_INTERFACE)
    while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};

    #if  defined (RPI_DISPLAY_TYPE) // RPi TFT type always needs 16 bit transfers
      hw_write_masked(&spi_get_hw(SPI_X)->cr0, (16 - 1) << SPI_SSPCR0_DSS_LSB,
SPI_SSPCR0_DSS_BITS);
    #else
      hw_write_masked(&spi_get_hw(SPI_X)->cr0, (8 - 1) << SPI_SSPCR0_DSS_LSB,
SPI_SSPCR0_DSS_BITS);
    #endif

    if (addr_col != x) {
      DC_C;
      spi_get_hw(SPI_X)->dr = (uint32_t)TFT_CASET;
```

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

  if (addr_row != y) {
    DC_C;
    spi_get_hw(SPI_X)->dr = (uint32_t)TFT_PASET;
    while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
    DC_D;
    spi_get_hw(SPI_X)->dr = (uint32_t)y>>8;
    spi_get_hw(SPI_X)->dr = (uint32_t)y;
    spi_get_hw(SPI_X)->dr = (uint32_t)y>>8;
    spi_get_hw(SPI_X)->dr = (uint32_t)y;
    addr_row = y;
    while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
  }

  DC_C;
  spi_get_hw(SPI_X)->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(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
    DC_D;
    tft_Write_8N(r); tft_Write_8N(g); tft_Write_8N(b);
  #else
    while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
    DC_D;
    #if  defined (RPI_DISPLAY_TYPE) // RPi TFT type always needs 16 bit transfers
      spi_get_hw(SPI_X)->dr = (uint32_t)color;
    #else
      spi_get_hw(SPI_X)->dr = (uint32_t)color>>8;
      spi_get_hw(SPI_X)->dr = (uint32_t)color;
    #endif
  #endif
  while (spi_get_hw(SPI_X)->sr & SPI_SSPSR_BSY_BITS) {};
#else
  // This is for the RP2040 and PIO interface (SPI or parallel)
  WAIT_FOR_STALL;
  tft_pio->sm[pio_sm].instr = pio_instr_addr;
  TX_FIFO = TFT_CASET;
    TX_FIFO = (x<<16) | x;
    TX_FIFO = TFT_PASET;
    TX_FIFO = (y<<16) | y;
    TX_FIFO = TFT_RAMWR;
    //DC set high by PIO
    #if  defined (SPI_18BIT_DRIVER)
      TX_FIFO = ((color & 0xF800)<<8) | ((color & 0x07E0)<<5) | ((color & 0x001F)<<3);
    #else
      TX_FIFO = color;
    #endif

  #endif

#else

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

    SPI_BUSY_CHECK;

  #if defined (SSD1351_DRIVER)
    if (rotation & 0x1) { swap_coord(x, y); }
    // 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(TFT_PARALLEL_16_BIT) || !defined(ESP32)
    DC_D; tft_Write_16(color);
  #else
    DC_D; tft_Write_16N(color);
  #endif
#endif

  end_tft_write();
}

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

  SPI_BUSY_CHECK;
  tft_Write_16N(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();
}


/***************************************************************************************
** Description:   Constants for anti-aliased line drawing on TFT and in Sprites
***************************************************************************************/
constexpr float PixelAlphaGain    = 255.0;
constexpr float LoAlphaTheshold  = 1.0/32.0;
constexpr float HiAlphaTheshold  = 1.0 - LoAlphaTheshold;

/***************************************************************************************
** Function name:           drawPixel (alpha blended)
** Description:             Draw a pixel blended with the screen or bg pixel colour
***************************************************************************************/
uint16_t TFT_eSPI::drawPixel(int32_t x, int32_t y, uint32_t color, uint8_t alpha, uint32_t bg_color)
{
  if (bg_color == 0x00FFFFFF) bg_color = readPixel(x, y);
  color = alphaBlend(alpha, color, bg_color);
  drawPixel(x, y, color);
  return color;
}

/***************************************************************************************
```

```
** Function name:              fillSmoothCircle
** Description:                Draw a filled anti-aliased circle
***************************************************************************************/
void TFT_eSPI::fillSmoothCircle(int32_t x, int32_t y, int32_t r, uint32_t color, uint32_t bg_color)
{
  if (r <= 0) return;

  inTransaction = true;

  drawFastHLine(x - r, y, 2 * r + 1, color);
  int32_t xs = 1;
  int32_t cx = 0;

  int32_t r1 = r * r;
  r++;
  int32_t r2 = r * r;

  for (int32_t cy = r - 1; cy > 0; cy--)
  {
    int32_t dy2 = (r - cy) * (r - cy);
    for (cx = xs; cx < r; cx++)
    {
      int32_t hyp2 = (r - cx) * (r - cx) + dy2;
      if (hyp2 <= r1) break;
      if (hyp2 >= r2) continue;
      float alphaf = (float)r - sqrtf(hyp2);
      if (alphaf > HiAlphaTheshold) break;
      xs = cx;
      if (alphaf < LoAlphaTheshold) continue;
      uint8_t alpha = alphaf * 255;

      if (bg_color == 0x00FFFFFF) {
        drawPixel(x + cx - r, y + cy - r, color, alpha, bg_color);
        drawPixel(x - cx + r, y + cy - r, color, alpha, bg_color);
        drawPixel(x - cx + r, y - cy + r, color, alpha, bg_color);
        drawPixel(x + cx - r, y - cy + r, color, alpha, bg_color);
      }
      else {
        uint16_t pcol = drawPixel(x + cx - r, y + cy - r, color, alpha, bg_color);
        drawPixel(x - cx + r, y + cy - r, pcol);
        drawPixel(x - cx + r, y - cy + r, pcol);
        drawPixel(x + cx - r, y - cy + r, pcol);
      }
    }
    drawFastHLine(x + cx - r, y + cy - r, 2 * (r - cx) + 1, color);
    drawFastHLine(x + cx - r, y - cy + r, 2 * (r - cx) + 1, color);
  }
  inTransaction = lockTransaction;
  end_tft_write();
}


/***************************************************************************************
** Function name:              fillSmoothRoundRect
** Description:                Draw a filled anti-aliased rounded corner rectangle
***************************************************************************************/
void TFT_eSPI::fillSmoothRoundRect(int32_t x, int32_t y, int32_t w, int32_t h, int32_t r, uint32_t
color, uint32_t bg_color)
{
  inTransaction = true;
  int32_t xs = 0;
  int32_t cx = 0;

  // Limit radius to half width or height
  if (r < 0) r = 0;
  if (r > w/2) r = w/2;
  if (r > h/2) r = h/2;

  y += r;
  h -= 2*r;
  fillRect(x, y, w, h, color);
  h--;
  x += r;
  w -= 2*r+1;
  int32_t r1 = r * r;
  r++;
  int32_t r2 = r * r;

  for (int32_t cy = r - 1; cy > 0; cy--)
  {
    int32_t dy2 = (r - cy) * (r - cy);
    for (cx = xs; cx < r; cx++)
    {
      int32_t hyp2 = (r - cx) * (r - cx) + dy2;
      if (hyp2 <= r1) break;
      if (hyp2 >= r2) continue;
      float alphaf = (float)r - sqrtf(hyp2);
      if (alphaf > HiAlphaTheshold) break;
      xs = cx;
      if (alphaf < LoAlphaTheshold) continue;
      uint8_t alpha = alphaf * 255;

      drawPixel(x + cx - r, y + cy - r, color, alpha, bg_color);
      drawPixel(x - cx + r + w, y + cy - r, color, alpha, bg_color);
      drawPixel(x - cx + r + w, y - cy + r + h, color, alpha, bg_color);
      drawPixel(x + cx - r, y - cy + r + h, color, alpha, bg_color);
    }
    drawFastHLine(x + cx - r, y + cy - r, 2 * (r - cx) + 1 + w, color);
```

```
    drawFastHLine(x + cx - r, y - cy + r + h, 2 * (r - cx) + 1 + w, color);
  }
  inTransaction = lockTransaction;
  end_tft_write();
}

/***************************************************************************************
** Function name:           drawSpot - maths intensive, so for small filled circles
** Description:             Draw an anti-aliased filled circle at ax,ay with radius r
***************************************************************************************/
void TFT_eSPI::drawSpot(float ax, float ay, float r, uint32_t fg_color, uint32_t bg_color)
{
  // Filled circle can be created by the wide line function with zero line length
  drawWedgeLine( ax, ay, ax, ay, r, r, fg_color, bg_color);
}

/***************************************************************************************
** Function name:           drawWideLine - background colour specified or pixel read
** Description:             draw an anti-aliased line with rounded ends, width wd
***************************************************************************************/
void TFT_eSPI::drawWideLine(float ax, float ay, float bx, float by, float wd, uint32_t fg_color, uint32_t
bg_color)
{
  drawWedgeLine( ax, ay, bx, by, wd/2.0, wd/2.0, fg_color, bg_color);
}

/***************************************************************************************
** Function name:           drawWedgeLine - background colour specified or pixel read
** Description:             draw an anti-aliased line with different width radiused ends
***************************************************************************************/
void TFT_eSPI::drawWedgeLine(float ax, float ay, float bx, float by, float ar, float br, uint32_t fg_color,
uint32_t bg_color)
{
  if ( (ar < 0.0) || (br < 0.0) )return;
  if ( (abs(ax - bx) < 0.01f) && (abs(ay - by) < 0.01f) ) bx += 0.01f;   // Avoid divide by zero

  // Find line bounding box
  int32_t x0 = (int32_t)floorf(fminf(ax-ar, bx-br));
  int32_t x1 = (int32_t) ceilf(fmaxf(ax+ar, bx+br));
  int32_t y0 = (int32_t)floorf(fminf(ay-ar, by-br));
  int32_t y1 = (int32_t) ceilf(fmaxf(ay+ar, by+br));

  if (!clipWindow(&x0, &y0, &x1, &y1)) return;

  // Establish x start and y start
  int32_t ys = ay;
  if ((ax-ar)>(bx-br)) ys = by;

  float rdt = ar - br; // Radius delta

  float alpha = 1.0f;
  ar += 0.5;

  uint16_t bg = bg_color;
  float xpax, ypay, bax = bx - ax, bay = by - ay;

  begin_nin_write();
  inTransaction = true;

  int32_t xs = x0;
  // Scan bounding box from ys down, calculate pixel intensity from distance to line
  for (int32_t yp = ys; yp <= y1; yp++) {
    bool swin = true;   // Flag to start new window area
    bool endX = false; // Flag to skip pixels
    ypay = yp - ay;
    for (int32_t xp = xs; xp <= x1; xp++) {
      if (endX) if (alpha <= LoAlphaTheshold) break;   // Skip right side
      xpax = xp - ax;
      alpha = ar - wedgeLineDistance(xpax, ypay, bax, bay, rdt);
      if (alpha <= LoAlphaTheshold ) continue;
      // Track edge to minimise calculations
      if (!endX) { endX = true; xs = xp; }
      if (alpha > HiAlphaTheshold) {
        if (swin) { setWindow(xp, yp, width()-1, yp); swin = false; }
        pushColor(fg_color);
        continue;
      }
      //Blend color with background and plot
      if (bg_color == 0x00FFFFFF) {
        bg = readPixel(xp, yp); swin = true;
      }
      if (swin) { setWindow(xp, yp, width()-1, yp); swin = false; }
      pushColor(alphaBlend((uint8_t)(alpha * PixelAlphaGain), fg_color, bg));
    }
  }

  // Reset x start to left side of box
  xs = x0;
  // Scan bounding box from ys-1 up, calculate pixel intensity from distance to line
  for (int32_t yp = ys-1; yp >= y0; yp--) {
    bool swin = true;   // Flag to start new window area
    bool endX = false; // Flag to skip pixels
    ypay = yp - ay;
    for (int32_t xp = xs; xp <= x1; xp++) {
      if (endX) if (alpha <= LoAlphaTheshold) break;   // Skip right side of drawn line
      xpax = xp - ax;
      alpha = ar - wedgeLineDistance(xpax, ypay, bax, bay, rdt);
      if (alpha <= LoAlphaTheshold ) continue;
      // Track line boundary
```

```
      if (!endX) { endX = true; xs = xp; }
      if (alpha > HiAlphaTheshold) {
        if (swin) { setWindow(xp, yp, width()-1, yp); swin = false; }
        pushColor(fg_color);
        continue;
      }
      //Blend color with background and plot
      if (bg_color == 0x00FFFFFF) {
        bg = readPixel(xp, yp); swin = true;
      }
      if (swin) { setWindow(xp, yp, width()-1, yp); swin = false; }
      pushColor(alphaBlend((uint8_t)(alpha * PixelAlphaGain), fg_color, bg));
    }
  }

  inTransaction = lockTransaction;
  end_nin_write();
}

// Calculate distance of px,py to closest part of line
/***************************************************************************************
** Function name:           lineDistance - private helper function for drawWedgeLine
** Description:             returns distance of px,py to closest part of a to b wedge
***************************************************************************************/
inline float TFT_eSPI::wedgeLineDistance(float xpax, float ypay, float bax, float bay, float dr)
{
  float h = fmaxf(fminf((xpax * bax + ypay * bay) / (bax * bax + bay * bay), 1.0f), 0.0f);
  float dx = xpax - bax * h, dy = ypay - bay * h;
  return sqrtf(dx * dx + dy * dy) + h * dr;
}


/***************************************************************************************
** 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:           fillRectVGradient
** Description:             draw a filled rectangle with a vertical colour gradient
***************************************************************************************/
void TFT_eSPI::fillRectVGradient(int16_t x, int16_t y, int16_t w, int16_t h, uint32_t color1, uint32_t
color2)
{
  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;

  begin_nin_write();

  float delta = -255.0/h;
  float alpha = 255.0;
  uint32_t color = color1;

  while (h--) {
    drawFastHLine(x, y++, w, color);
    alpha += delta;
    color = alphaBlend((uint8_t)alpha, color1, color2);
  }

  end_nin_write();
}


/***************************************************************************************
** Function name:           fillRectHGradient
** Description:             draw a filled rectangle with a horizontal colour gradient
***************************************************************************************/
void TFT_eSPI::fillRectHGradient(int16_t x, int16_t y, int16_t w, int16_t h, uint32_t color1,
uint32_t color2)
{
  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;

  begin_nin_write();
```

```
  float delta = -255.0/w;
  float alpha = 255.0;
  uint32_t color = color1;

  while (w--) {
    drawFastVLine(x++, y, h, color);
    alpha += delta;
    color = alphaBlend((uint8_t)alpha, color1, color2);
  }

  end_nin_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
***************************************************************************************/
/* // Not all processors support buffered write
#ifndef ARDUINO_ARCH_ESP8266 // Avoid ESP8266 board package bug
size_t TFT_eSPI::write(const uint8_t *buf, size_t len)
{
  inTransaction = true;

  uint8_t *lbuf = (uint8_t *)buf;
  while(*lbuf !=0 && len--) write(*lbuf++);

  inTransaction = lockTransaction;
  end_tft_write();
  return 1;
}
#endif
*/
/***************************************************************************************
** 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;

//vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv DEBUG vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
  //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 {
    if (textwrapX && (cursor_x + cwidth * textsize > width())) {
      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*sizeof(void *)) );
      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 * textsize ;

  int32_t w = width;
  int32_t pX       = 0;
  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();
      inTransaction = true;

      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++;
          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);
        }
      }
    }
    inTransaction = lockTransaction;
    end_tft_write();
  }
  // End of RLE font rendering
#endif

#if !defined (LOAD_FONT2) && !defined (LOAD_RLE)
  // Stop warnings
  flash_address = flash_address;
  w = w;
  pX = pX;
  pY = pY;
  line = line;
  clip = clip;
#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;
      case MC_DATUM:
        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) {
    setCursor(poX, poY);

    bool fillbg = _fillbg;
    // If padding is requested then fill the text background
    if (padX && !_fillbg) _fillbg = true;

    while (n < len) {
      uint16_t uniCode = decodeUTF8((uint8_t*)string, &n, len - n);
      drawGlyph(uniCode);
    }
    _fillbg = fillbg; // restore state
    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);
    }
  }

//vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv DEBUG vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
// 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

//vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv DEBUG vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
// 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
  bool negative = false;
```

```
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
  negative = true;
}

floatNumber += rounding; // Round up or down

if (dp == 0) {
  if (negative) floatNumber = -floatNumber;
  return drawNumber((long)floatNumber, poX, poY, font);
}

// 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 (uint16_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) && ! defined (RP2040_PIO_INTERFACE)
SPIClass& TFT_eSPI::getSPIinstance(void)
{
  return spi;
}
#endif


/***************************************************************************************
** Function name:           verifySetupID
** Description:             Compare the ID if USER_SETUP_ID defined in user setup file
***************************************************************************************/
bool TFT_eSPI::verifySetupID(uint32_t id)
{
#if defined (USER_SETUP_ID)
  if (USER_SETUP_ID == id) return true;
#else
  id = id; // Avoid warning
#endif
  return false;
}

/***************************************************************************************
** 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 (USER_SETUP_INFO)
  tft_settings.setup_info = USER_SETUP_INFO;
#else
  tft_settings.setup_info = "NA";
#endif

#if defined (USER_SETUP_ID)
  tft_settings.setup_id = USER_SETUP_ID;
#else
  tft_settings.setup_id = 0;
#endif

#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) || defined(TFT_PARALLEL_16_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
  #ifdef TFT_SPI_PORT
    tft_settings.port = TFT_SPI_PORT;
  #else
    tft_settings.port = 255;
  #endif
  #ifdef RP2040_PIO_SPI
    tft_settings.interface = 0x10;
  #else
    tft_settings.interface = 0x0;
  #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) || defined(TFT_PARALLEL_16_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"
#endif

#include "Extensions/Button.cpp"

#include "Extensions/Sprite.cpp"

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

#ifdef AA_GRAPHICS
  #include "Extensions/AA_graphics.cpp"   // Loaded if SMOOTH_FONT is defined by user
#endif
////////////////////////////////////////////////////////////////////////////////////
```