/\*!

\* @file Adafruit\_SPITFT.cpp

\*

\* @mainpage Adafruit SPI TFT Displays (and some others)

\*

\* @section intro\_sec Introduction

\*

\* Part of Adafruit's GFX graphics library. Originally this class was

\* written to handle a range of color TFT displays connected via SPI,

\* but over time this library and some display-specific subclasses have

\* mutated to include some color OLEDs as well as parallel-interfaced

\* displays. The name's been kept for the sake of older code.

\*

\* Adafruit invests time and resources providing this open source code,

\* please support Adafruit and open-source hardware by purchasing

\* products from Adafruit!

\* @section dependencies Dependencies

\*

\* This library depends on <a href="https://github.com/adafruit/Adafruit\_GFX">

\* Adafruit\_GFX</a> being present on your system. Please make sure you have

\* installed the latest version before using this library.

\*

\* @section author Author

\*

\* Written by Limor "ladyada" Fried for Adafruit Industries,

\* with contributions from the open source community.

\*

\* @section license License

\*

\* BSD license, all text here must be included in any redistribution.

\*/

#if !defined(\_\_AVR\_ATtiny85\_\_) // Not for ATtiny, at all

#include "Adafruit\_SPITFT.h"

#if defined(\_\_AVR\_\_)

#if defined(\_\_AVR\_XMEGA\_\_) // only tested with \_\_AVR\_ATmega4809\_\_

#define AVR\_WRITESPI(x) \

for (SPI0\_DATA = (x); (!(SPI0\_INTFLAGS & \_BV(SPI\_IF\_bp)));)

#else

#define AVR\_WRITESPI(x) for (SPDR = (x); (!(SPSR & \_BV(SPIF)));)

#endif

#endif

#if defined(PORT\_IOBUS)

// On SAMD21, redefine digitalPinToPort() to use the slightly-faster

// PORT\_IOBUS rather than PORT (not needed on SAMD51).

#undef digitalPinToPort

#define digitalPinToPort(P) (&(PORT\_IOBUS->Group[g\_APinDescription[P].ulPort]))

#endif // end PORT\_IOBUS

#if defined(USE\_SPI\_DMA) && (defined(\_\_SAMD51\_\_) || defined(ARDUINO\_SAMD\_ZERO))

// #pragma message ("GFX DMA IS ENABLED. HIGHLY EXPERIMENTAL.")

#include "wiring\_private.h" // pinPeripheral() function

#include <Adafruit\_ZeroDMA.h>

#include <malloc.h> // memalign() function

#define tcNum 2 // Timer/Counter for parallel write strobe PWM

#define wrPeripheral PIO\_CCL // Use CCL to invert write strobe

// DMA transfer-in-progress indicator and callback

static volatile bool dma\_busy = false;

static void dma\_callback(Adafruit\_ZeroDMA \*dma) { dma\_busy = false; }

#if defined(\_\_SAMD51\_\_)

// Timer/counter info by index #

static const struct {

Tc \*tc; // -> Timer/Counter base address

int gclk; // GCLK ID

int evu; // EVSYS user ID

} tcList[] = {{TC0, TC0\_GCLK\_ID, EVSYS\_ID\_USER\_TC0\_EVU},

{TC1, TC1\_GCLK\_ID, EVSYS\_ID\_USER\_TC1\_EVU},

{TC2, TC2\_GCLK\_ID, EVSYS\_ID\_USER\_TC2\_EVU},

{TC3, TC3\_GCLK\_ID, EVSYS\_ID\_USER\_TC3\_EVU},

#if defined(TC4)

{TC4, TC4\_GCLK\_ID, EVSYS\_ID\_USER\_TC4\_EVU},

#endif

#if defined(TC5)

{TC5, TC5\_GCLK\_ID, EVSYS\_ID\_USER\_TC5\_EVU},

#endif

#if defined(TC6)

{TC6, TC6\_GCLK\_ID, EVSYS\_ID\_USER\_TC6\_EVU},

#endif

#if defined(TC7)

{TC7, TC7\_GCLK\_ID, EVSYS\_ID\_USER\_TC7\_EVU}

#endif

};

#define NUM\_TIMERS (sizeof tcList / sizeof tcList[0]) ///< # timer/counters

#endif // end \_\_SAMD51\_\_

#endif // end USE\_SPI\_DMA

// Possible values for Adafruit\_SPITFT.connection:

#define TFT\_HARD\_SPI 0 ///< Display interface = hardware SPI

#define TFT\_SOFT\_SPI 1 ///< Display interface = software SPI

#define TFT\_PARALLEL 2 ///< Display interface = 8- or 16-bit parallel

// CONSTRUCTORS ------------------------------------------------------------

/\*!

@brief Adafruit\_SPITFT constructor for software (bitbang) SPI.

@param w Display width in pixels at default rotation setting (0).

@param h Display height in pixels at default rotation setting (0).

@param cs Arduino pin # for chip-select (-1 if unused, tie CS low).

@param dc Arduino pin # for data/command select (required).

@param mosi Arduino pin # for bitbang SPI MOSI signal (required).

@param sck Arduino pin # for bitbang SPI SCK signal (required).

@param rst Arduino pin # for display reset (optional, display reset

can be tied to MCU reset, default of -1 means unused).

@param miso Arduino pin # for bitbang SPI MISO signal (optional,

-1 default, many displays don't support SPI read).

@note Output pins are not initialized; application typically will

need to call subclass' begin() function, which in turn calls

this library's initSPI() function to initialize pins.

\*/

Adafruit\_SPITFT::Adafruit\_SPITFT(uint16\_t w, uint16\_t h, int8\_t cs, int8\_t dc,

int8\_t mosi, int8\_t sck, int8\_t rst,

int8\_t miso)

: Adafruit\_GFX(w, h), connection(TFT\_SOFT\_SPI), \_rst(rst), \_cs(cs),

\_dc(dc) {

swspi.\_sck = sck;

swspi.\_mosi = mosi;

swspi.\_miso = miso;

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

#if defined(CORE\_TEENSY)

#if !defined(KINETISK)

dcPinMask = digitalPinToBitMask(dc);

swspi.sckPinMask = digitalPinToBitMask(sck);

swspi.mosiPinMask = digitalPinToBitMask(mosi);

#endif

dcPortSet = portSetRegister(dc);

dcPortClr = portClearRegister(dc);

swspi.sckPortSet = portSetRegister(sck);

swspi.sckPortClr = portClearRegister(sck);

swspi.mosiPortSet = portSetRegister(mosi);

swspi.mosiPortClr = portClearRegister(mosi);

if (cs >= 0) {

#if !defined(KINETISK)

csPinMask = digitalPinToBitMask(cs);

#endif

csPortSet = portSetRegister(cs);

csPortClr = portClearRegister(cs);

} else {

#if !defined(KINETISK)

csPinMask = 0;

#endif

csPortSet = dcPortSet;

csPortClr = dcPortClr;

}

if (miso >= 0) {

swspi.misoPort = portInputRegister(miso);

#if !defined(KINETISK)

swspi.misoPinMask = digitalPinToBitMask(miso);

#endif

} else {

swspi.misoPort = portInputRegister(dc);

}

#else // !CORE\_TEENSY

dcPinMask = digitalPinToBitMask(dc);

swspi.sckPinMask = digitalPinToBitMask(sck);

swspi.mosiPinMask = digitalPinToBitMask(mosi);

dcPortSet = &(PORT->Group[g\_APinDescription[dc].ulPort].OUTSET.reg);

dcPortClr = &(PORT->Group[g\_APinDescription[dc].ulPort].OUTCLR.reg);

swspi.sckPortSet = &(PORT->Group[g\_APinDescription[sck].ulPort].OUTSET.reg);

swspi.sckPortClr = &(PORT->Group[g\_APinDescription[sck].ulPort].OUTCLR.reg);

swspi.mosiPortSet = &(PORT->Group[g\_APinDescription[mosi].ulPort].OUTSET.reg);

swspi.mosiPortClr = &(PORT->Group[g\_APinDescription[mosi].ulPort].OUTCLR.reg);

if (cs >= 0) {

csPinMask = digitalPinToBitMask(cs);

csPortSet = &(PORT->Group[g\_APinDescription[cs].ulPort].OUTSET.reg);

csPortClr = &(PORT->Group[g\_APinDescription[cs].ulPort].OUTCLR.reg);

} else {

// No chip-select line defined; might be permanently tied to GND.

// Assign a valid GPIO register (though not used for CS), and an

// empty pin bitmask...the nonsense bit-twiddling might be faster

// than checking \_cs and possibly branching.

csPortSet = dcPortSet;

csPortClr = dcPortClr;

csPinMask = 0;

}

if (miso >= 0) {

swspi.misoPinMask = digitalPinToBitMask(miso);

swspi.misoPort = (PORTreg\_t)portInputRegister(digitalPinToPort(miso));

} else {

swspi.misoPinMask = 0;

swspi.misoPort = (PORTreg\_t)portInputRegister(digitalPinToPort(dc));

}

#endif // end !CORE\_TEENSY

#else // !HAS\_PORT\_SET\_CLR

dcPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(dc));

dcPinMaskSet = digitalPinToBitMask(dc);

swspi.sckPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(sck));

swspi.sckPinMaskSet = digitalPinToBitMask(sck);

swspi.mosiPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(mosi));

swspi.mosiPinMaskSet = digitalPinToBitMask(mosi);

if (cs >= 0) {

csPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(cs));

csPinMaskSet = digitalPinToBitMask(cs);

} else {

// No chip-select line defined; might be permanently tied to GND.

// Assign a valid GPIO register (though not used for CS), and an

// empty pin bitmask...the nonsense bit-twiddling might be faster

// than checking \_cs and possibly branching.

csPort = dcPort;

csPinMaskSet = 0;

}

if (miso >= 0) {

swspi.misoPort = (PORTreg\_t)portInputRegister(digitalPinToPort(miso));

swspi.misoPinMask = digitalPinToBitMask(miso);

} else {

swspi.misoPort = (PORTreg\_t)portInputRegister(digitalPinToPort(dc));

swspi.misoPinMask = 0;

}

csPinMaskClr = ~csPinMaskSet;

dcPinMaskClr = ~dcPinMaskSet;

swspi.sckPinMaskClr = ~swspi.sckPinMaskSet;

swspi.mosiPinMaskClr = ~swspi.mosiPinMaskSet;

#endif // !end HAS\_PORT\_SET\_CLR

#endif // end USE\_FAST\_PINIO

}

/\*!

@brief Adafruit\_SPITFT constructor for hardware SPI using the board's

default SPI peripheral.

@param w Display width in pixels at default rotation setting (0).

@param h Display height in pixels at default rotation setting (0).

@param cs Arduino pin # for chip-select (-1 if unused, tie CS low).

@param dc Arduino pin # for data/command select (required).

@param rst Arduino pin # for display reset (optional, display reset

can be tied to MCU reset, default of -1 means unused).

@note Output pins are not initialized; application typically will

need to call subclass' begin() function, which in turn calls

this library's initSPI() function to initialize pins.

\*/

#if defined(ESP8266) // See notes below

Adafruit\_SPITFT::Adafruit\_SPITFT(uint16\_t w, uint16\_t h, int8\_t cs, int8\_t dc,

int8\_t rst)

: Adafruit\_GFX(w, h), connection(TFT\_HARD\_SPI), \_rst(rst), \_cs(cs),

\_dc(dc) {

hwspi.\_spi = &SPI;

}

#else // !ESP8266

Adafruit\_SPITFT::Adafruit\_SPITFT(uint16\_t w, uint16\_t h, int8\_t cs, int8\_t dc,

int8\_t rst)

: Adafruit\_SPITFT(w, h, &SPI, cs, dc, rst) {

// This just invokes the hardware SPI constructor below,

// passing the default SPI device (&SPI).

}

#endif // end !ESP8266

#if !defined(ESP8266)

// ESP8266 compiler freaks out at this constructor -- it can't disambiguate

// beteween the SPIClass pointer (argument #3) and a regular integer.

// Solution here it to just not offer this variant on the ESP8266. You can

// use the default hardware SPI peripheral, or you can use software SPI,

// but if there's any library out there that creates a 'virtual' SPIClass

// peripheral and drives it with software bitbanging, that's not supported.

/\*!

@brief Adafruit\_SPITFT constructor for hardware SPI using a specific

SPI peripheral.

@param w Display width in pixels at default rotation (0).

@param h Display height in pixels at default rotation (0).

@param spiClass Pointer to SPIClass type (e.g. &SPI or &SPI1).

@param cs Arduino pin # for chip-select (-1 if unused, tie CS low).

@param dc Arduino pin # for data/command select (required).

@param rst Arduino pin # for display reset (optional, display reset

can be tied to MCU reset, default of -1 means unused).

@note Output pins are not initialized in constructor; application

typically will need to call subclass' begin() function, which

in turn calls this library's initSPI() function to initialize

pins. EXCEPT...if you have built your own SERCOM SPI peripheral

(calling the SPIClass constructor) rather than one of the

built-in SPI devices (e.g. &SPI, &SPI1 and so forth), you will

need to call the begin() function for your object as well as

pinPeripheral() for the MOSI, MISO and SCK pins to configure

GPIO manually. Do this BEFORE calling the display-specific

begin or init function. Unfortunate but unavoidable.

\*/

Adafruit\_SPITFT::Adafruit\_SPITFT(uint16\_t w, uint16\_t h, SPIClass \*spiClass,

int8\_t cs, int8\_t dc, int8\_t rst)

: Adafruit\_GFX(w, h), connection(TFT\_HARD\_SPI), \_rst(rst), \_cs(cs),

\_dc(dc) {

hwspi.\_spi = spiClass;

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

#if defined(CORE\_TEENSY)

#if !defined(KINETISK)

dcPinMask = digitalPinToBitMask(dc);

#endif

dcPortSet = portSetRegister(dc);

dcPortClr = portClearRegister(dc);

if (cs >= 0) {

#if !defined(KINETISK)

csPinMask = digitalPinToBitMask(cs);

#endif

csPortSet = portSetRegister(cs);

csPortClr = portClearRegister(cs);

} else { // see comments below

#if !defined(KINETISK)

csPinMask = 0;

#endif

csPortSet = dcPortSet;

csPortClr = dcPortClr;

}

#else // !CORE\_TEENSY

dcPinMask = digitalPinToBitMask(dc);

dcPortSet = &(PORT->Group[g\_APinDescription[dc].ulPort].OUTSET.reg);

dcPortClr = &(PORT->Group[g\_APinDescription[dc].ulPort].OUTCLR.reg);

if (cs >= 0) {

csPinMask = digitalPinToBitMask(cs);

csPortSet = &(PORT->Group[g\_APinDescription[cs].ulPort].OUTSET.reg);

csPortClr = &(PORT->Group[g\_APinDescription[cs].ulPort].OUTCLR.reg);

} else {

// No chip-select line defined; might be permanently tied to GND.

// Assign a valid GPIO register (though not used for CS), and an

// empty pin bitmask...the nonsense bit-twiddling might be faster

// than checking \_cs and possibly branching.

csPortSet = dcPortSet;

csPortClr = dcPortClr;

csPinMask = 0;

}

#endif // end !CORE\_TEENSY

#else // !HAS\_PORT\_SET\_CLR

dcPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(dc));

dcPinMaskSet = digitalPinToBitMask(dc);

if (cs >= 0) {

csPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(cs));

csPinMaskSet = digitalPinToBitMask(cs);

} else {

// No chip-select line defined; might be permanently tied to GND.

// Assign a valid GPIO register (though not used for CS), and an

// empty pin bitmask...the nonsense bit-twiddling might be faster

// than checking \_cs and possibly branching.

csPort = dcPort;

csPinMaskSet = 0;

}

csPinMaskClr = ~csPinMaskSet;

dcPinMaskClr = ~dcPinMaskSet;

#endif // end !HAS\_PORT\_SET\_CLR

#endif // end USE\_FAST\_PINIO

}

#endif // end !ESP8266

/\*!

@brief Adafruit\_SPITFT constructor for parallel display connection.

@param w Display width in pixels at default rotation (0).

@param h Display height in pixels at default rotation (0).

@param busWidth If tft16 (enumeration in header file), is a 16-bit

parallel connection, else 8-bit.

16-bit isn't fully implemented or tested yet so

applications should pass "tft8bitbus" for now...needed to

stick a required enum argument in there to

disambiguate this constructor from the soft-SPI case.

Argument is ignored on 8-bit architectures (no 'wide'

support there since PORTs are 8 bits anyway).

@param d0 Arduino pin # for data bit 0 (1+ are extrapolated).

The 8 (or 16) data bits MUST be contiguous and byte-

aligned (or word-aligned for wide interface) within

the same PORT register (might not correspond to

Arduino pin sequence).

@param wr Arduino pin # for write strobe (required).

@param dc Arduino pin # for data/command select (required).

@param cs Arduino pin # for chip-select (optional, -1 if unused,

tie CS low).

@param rst Arduino pin # for display reset (optional, display reset

can be tied to MCU reset, default of -1 means unused).

@param rd Arduino pin # for read strobe (optional, -1 if unused).

@note Output pins are not initialized; application typically will need

to call subclass' begin() function, which in turn calls this

library's initSPI() function to initialize pins.

Yes, the name is a misnomer...this library originally handled

only SPI displays, parallel being a recent addition (but not

wanting to break existing code).

\*/

Adafruit\_SPITFT::Adafruit\_SPITFT(uint16\_t w, uint16\_t h, tftBusWidth busWidth,

int8\_t d0, int8\_t wr, int8\_t dc, int8\_t cs,

int8\_t rst, int8\_t rd)

: Adafruit\_GFX(w, h), connection(TFT\_PARALLEL), \_rst(rst), \_cs(cs),

\_dc(dc) {

tft8.\_d0 = d0;

tft8.\_wr = wr;

tft8.\_rd = rd;

tft8.wide = (busWidth == tft16bitbus);

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

#if defined(CORE\_TEENSY)

tft8.wrPortSet = portSetRegister(wr);

tft8.wrPortClr = portClearRegister(wr);

#if !defined(KINETISK)

dcPinMask = digitalPinToBitMask(dc);

#endif

dcPortSet = portSetRegister(dc);

dcPortClr = portClearRegister(dc);

if (cs >= 0) {

#if !defined(KINETISK)

csPinMask = digitalPinToBitMask(cs);

#endif

csPortSet = portSetRegister(cs);

csPortClr = portClearRegister(cs);

} else { // see comments below

#if !defined(KINETISK)

csPinMask = 0;

#endif

csPortSet = dcPortSet;

csPortClr = dcPortClr;

}

if (rd >= 0) { // if read-strobe pin specified...

#if defined(KINETISK)

tft8.rdPinMask = 1;

#else // !KINETISK

tft8.rdPinMask = digitalPinToBitMask(rd);

#endif

tft8.rdPortSet = portSetRegister(rd);

tft8.rdPortClr = portClearRegister(rd);

} else {

tft8.rdPinMask = 0;

tft8.rdPortSet = dcPortSet;

tft8.rdPortClr = dcPortClr;

}

// These are all uint8\_t\* pointers -- elsewhere they're recast

// as necessary if a 'wide' 16-bit interface is in use.

tft8.writePort = portOutputRegister(d0);

tft8.readPort = portInputRegister(d0);

tft8.dirSet = portModeRegister(d0);

tft8.dirClr = portModeRegister(d0);

#else // !CORE\_TEENSY

tft8.wrPinMask = digitalPinToBitMask(wr);

tft8.wrPortSet = &(PORT->Group[g\_APinDescription[wr].ulPort].OUTSET.reg);

tft8.wrPortClr = &(PORT->Group[g\_APinDescription[wr].ulPort].OUTCLR.reg);

dcPinMask = digitalPinToBitMask(dc);

dcPortSet = &(PORT->Group[g\_APinDescription[dc].ulPort].OUTSET.reg);

dcPortClr = &(PORT->Group[g\_APinDescription[dc].ulPort].OUTCLR.reg);

if (cs >= 0) {

csPinMask = digitalPinToBitMask(cs);

csPortSet = &(PORT->Group[g\_APinDescription[cs].ulPort].OUTSET.reg);

csPortClr = &(PORT->Group[g\_APinDescription[cs].ulPort].OUTCLR.reg);

} else {

// No chip-select line defined; might be permanently tied to GND.

// Assign a valid GPIO register (though not used for CS), and an

// empty pin bitmask...the nonsense bit-twiddling might be faster

// than checking \_cs and possibly branching.

csPortSet = dcPortSet;

csPortClr = dcPortClr;

csPinMask = 0;

}

if (rd >= 0) { // if read-strobe pin specified...

tft8.rdPinMask = digitalPinToBitMask(rd);

tft8.rdPortSet = &(PORT->Group[g\_APinDescription[rd].ulPort].OUTSET.reg);

tft8.rdPortClr = &(PORT->Group[g\_APinDescription[rd].ulPort].OUTCLR.reg);

} else {

tft8.rdPinMask = 0;

tft8.rdPortSet = dcPortSet;

tft8.rdPortClr = dcPortClr;

}

// Get pointers to PORT write/read/dir bytes within 32-bit PORT

uint8\_t dBit = g\_APinDescription[d0].ulPin; // d0 bit # in PORT

PortGroup \*p = (&(PORT->Group[g\_APinDescription[d0].ulPort]));

uint8\_t offset = dBit / 8; // d[7:0] byte # within PORT

if (tft8.wide)

offset &= ~1; // d[15:8] byte # within PORT

// These are all uint8\_t\* pointers -- elsewhere they're recast

// as necessary if a 'wide' 16-bit interface is in use.

tft8.writePort = (volatile uint8\_t \*)&(p->OUT.reg) + offset;

tft8.readPort = (volatile uint8\_t \*)&(p->IN.reg) + offset;

tft8.dirSet = (volatile uint8\_t \*)&(p->DIRSET.reg) + offset;

tft8.dirClr = (volatile uint8\_t \*)&(p->DIRCLR.reg) + offset;

#endif // end !CORE\_TEENSY

#else // !HAS\_PORT\_SET\_CLR

tft8.wrPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(wr));

tft8.wrPinMaskSet = digitalPinToBitMask(wr);

dcPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(dc));

dcPinMaskSet = digitalPinToBitMask(dc);

if (cs >= 0) {

csPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(cs));

csPinMaskSet = digitalPinToBitMask(cs);

} else {

// No chip-select line defined; might be permanently tied to GND.

// Assign a valid GPIO register (though not used for CS), and an

// empty pin bitmask...the nonsense bit-twiddling might be faster

// than checking \_cs and possibly branching.

csPort = dcPort;

csPinMaskSet = 0;

}

if (rd >= 0) { // if read-strobe pin specified...

tft8.rdPort = (PORTreg\_t)portOutputRegister(digitalPinToPort(rd));

tft8.rdPinMaskSet = digitalPinToBitMask(rd);

} else {

tft8.rdPort = dcPort;

tft8.rdPinMaskSet = 0;

}

csPinMaskClr = ~csPinMaskSet;

dcPinMaskClr = ~dcPinMaskSet;

tft8.wrPinMaskClr = ~tft8.wrPinMaskSet;

tft8.rdPinMaskClr = ~tft8.rdPinMaskSet;

tft8.writePort = (PORTreg\_t)portOutputRegister(digitalPinToPort(d0));

tft8.readPort = (PORTreg\_t)portInputRegister(digitalPinToPort(d0));

tft8.portDir = (PORTreg\_t)portModeRegister(digitalPinToPort(d0));

#endif // end !HAS\_PORT\_SET\_CLR

#endif // end USE\_FAST\_PINIO

}

// end constructors -------

// CLASS MEMBER FUNCTIONS --------------------------------------------------

// begin() and setAddrWindow() MUST be declared by any subclass.

/\*!

@brief Configure microcontroller pins for TFT interfacing. Typically

called by a subclass' begin() function.

@param freq SPI frequency when using hardware SPI. If default (0)

is passed, will fall back on a device-specific value.

Value is ignored when using software SPI or parallel

connection.

@param spiMode SPI mode when using hardware SPI. MUST be one of the

values SPI\_MODE0, SPI\_MODE1, SPI\_MODE2 or SPI\_MODE3

defined in SPI.h. Do NOT attempt to pass '0' for

SPI\_MODE0 and so forth...the values are NOT the same!

Use ONLY the defines! (Pity it's not an enum.)

@note Another anachronistically-named function; this is called even

when the display connection is parallel (not SPI). Also, this

could probably be made private...quite a few class functions

were generously put in the public section.

\*/

void Adafruit\_SPITFT::initSPI(uint32\_t freq, uint8\_t spiMode) {

if (!freq)

freq = DEFAULT\_SPI\_FREQ; // If no freq specified, use default

// Init basic control pins common to all connection types

if (\_cs >= 0) {

pinMode(\_cs, OUTPUT);

digitalWrite(\_cs, HIGH); // Deselect

}

pinMode(\_dc, OUTPUT);

digitalWrite(\_dc, HIGH); // Data mode

if (connection == TFT\_HARD\_SPI) {

#if defined(SPI\_HAS\_TRANSACTION)

hwspi.settings = SPISettings(freq, MSBFIRST, spiMode);

#else

hwspi.\_freq = freq; // Save freq value for later

#endif

hwspi.\_mode = spiMode; // Save spiMode value for later

// Call hwspi.\_spi->begin() ONLY if this is among the 'established'

// SPI interfaces in variant.h. For DIY roll-your-own SERCOM SPIs,

// begin() and pinPeripheral() calls MUST be made in one's calling

// code, BEFORE the screen-specific begin/init function is called.

// Reason for this is that SPI::begin() makes its own calls to

// pinPeripheral() based on g\_APinDescription[n].ulPinType, which

// on non-established SPI interface pins will always be PIO\_DIGITAL

// or similar, while we need PIO\_SERCOM or PIO\_SERCOM\_ALT...it's

// highly unique between devices and variants for each pin or

// SERCOM so we can't make those calls ourselves here. And the SPI

// device needs to be set up before calling this because it's

// immediately followed with initialization commands. Blargh.

if (

#if !defined(SPI\_INTERFACES\_COUNT)

1

#else

#if SPI\_INTERFACES\_COUNT > 0

(hwspi.\_spi == &SPI)

#endif

#if SPI\_INTERFACES\_COUNT > 1

|| (hwspi.\_spi == &SPI1)

#endif

#if SPI\_INTERFACES\_COUNT > 2

|| (hwspi.\_spi == &SPI2)

#endif

#if SPI\_INTERFACES\_COUNT > 3

|| (hwspi.\_spi == &SPI3)

#endif

#if SPI\_INTERFACES\_COUNT > 4

|| (hwspi.\_spi == &SPI4)

#endif

#if SPI\_INTERFACES\_COUNT > 5

|| (hwspi.\_spi == &SPI5)

#endif

#endif // end SPI\_INTERFACES\_COUNT

) {

hwspi.\_spi->begin();

}

} else if (connection == TFT\_SOFT\_SPI) {

pinMode(swspi.\_mosi, OUTPUT);

digitalWrite(swspi.\_mosi, LOW);

pinMode(swspi.\_sck, OUTPUT);

digitalWrite(swspi.\_sck, LOW);

if (swspi.\_miso >= 0) {

pinMode(swspi.\_miso, INPUT);

}

} else { // TFT\_PARALLEL

// Initialize data pins. We were only passed d0, so scan

// the pin description list looking for the other pins.

// They'll be on the same PORT, and within the next 7 (or 15) bits

// (because we need to write to a contiguous PORT byte or word).

#if defined(\_\_AVR\_\_)

// PORT registers are 8 bits wide, so just need a register match...

for (uint8\_t i = 0; i < NUM\_DIGITAL\_PINS; i++) {

if ((PORTreg\_t)portOutputRegister(digitalPinToPort(i)) ==

tft8.writePort) {

pinMode(i, OUTPUT);

digitalWrite(i, LOW);

}

}

#elif defined(USE\_FAST\_PINIO)

#if defined(CORE\_TEENSY)

if (!tft8.wide) {

\*tft8.dirSet = 0xFF; // Set port to output

\*tft8.writePort = 0x00; // Write all 0s

} else {

\*(volatile uint16\_t \*)tft8.dirSet = 0xFFFF;

\*(volatile uint16\_t \*)tft8.writePort = 0x0000;

}

#else // !CORE\_TEENSY

uint8\_t portNum = g\_APinDescription[tft8.\_d0].ulPort, // d0 PORT #

dBit = g\_APinDescription[tft8.\_d0].ulPin, // d0 bit in PORT

lastBit = dBit + (tft8.wide ? 15 : 7);

for (uint8\_t i = 0; i < PINS\_COUNT; i++) {

if ((g\_APinDescription[i].ulPort == portNum) &&

(g\_APinDescription[i].ulPin >= dBit) &&

(g\_APinDescription[i].ulPin <= (uint32\_t)lastBit)) {

pinMode(i, OUTPUT);

digitalWrite(i, LOW);

}

}

#endif // end !CORE\_TEENSY

#endif

pinMode(tft8.\_wr, OUTPUT);

digitalWrite(tft8.\_wr, HIGH);

if (tft8.\_rd >= 0) {

pinMode(tft8.\_rd, OUTPUT);

digitalWrite(tft8.\_rd, HIGH);

}

}

if (\_rst >= 0) {

// Toggle \_rst low to reset

pinMode(\_rst, OUTPUT);

digitalWrite(\_rst, HIGH);

delay(100);

digitalWrite(\_rst, LOW);

delay(100);

digitalWrite(\_rst, HIGH);

delay(200);

}

#if defined(USE\_SPI\_DMA) && (defined(\_\_SAMD51\_\_) || defined(ARDUINO\_SAMD\_ZERO))

if (((connection == TFT\_HARD\_SPI) || (connection == TFT\_PARALLEL)) &&

(dma.allocate() == DMA\_STATUS\_OK)) { // Allocate channel

// The DMA library needs to alloc at least one valid descriptor,

// so we do that here. It's not used in the usual sense though,

// just before a transfer we copy descriptor[0] to this address.

if (dptr = dma.addDescriptor(NULL, NULL, 42, DMA\_BEAT\_SIZE\_BYTE, false,

false)) {

// Alloc 2 scanlines worth of pixels on display's major axis,

// whichever that is, rounding each up to 2-pixel boundary.

int major = (WIDTH > HEIGHT) ? WIDTH : HEIGHT;

major += (major & 1); // -> next 2-pixel bound, if needed.

maxFillLen = major \* 2; // 2 scanlines

// Note to future self: if you decide to make the pixel buffer

// much larger, remember that DMA transfer descriptors can't

// exceed 65,535 bytes (not 65,536), meaning 32,767 pixels max.

// Not that we have that kind of RAM to throw around right now.

if ((pixelBuf[0] = (uint16\_t \*)malloc(maxFillLen \* sizeof(uint16\_t)))) {

// Alloc OK. Get pointer to start of second scanline.

pixelBuf[1] = &pixelBuf[0][major];

// Determine number of DMA descriptors needed to cover

// entire screen when entire 2-line pixelBuf is used

// (round up for fractional last descriptor).

int numDescriptors = (WIDTH \* HEIGHT + (maxFillLen - 1)) / maxFillLen;

// DMA descriptors MUST be 128-bit (16 byte) aligned.

// memalign() is considered obsolete but it's replacements

// (aligned\_alloc() or posix\_memalign()) are not currently

// available in the version of ARM GCC in use, but this

// is, so here we are.

if ((descriptor = (DmacDescriptor \*)memalign(

16, numDescriptors \* sizeof(DmacDescriptor)))) {

int dmac\_id;

volatile uint32\_t \*data\_reg;

if (connection == TFT\_HARD\_SPI) {

// THIS IS AN AFFRONT TO NATURE, but I don't know

// any "clean" way to get the sercom number from the

// the SPIClass pointer (e.g. &SPI or &SPI1), which

// is all we have to work with. SPIClass does contain

// a SERCOM pointer but it is a PRIVATE member!

// Doing an UNSPEAKABLY HORRIBLE THING here, directly

// accessing the first 32-bit value in the SPIClass

// structure, knowing that's (currently) where the

// SERCOM pointer lives, but this ENTIRELY DEPENDS on

// that structure not changing nor the compiler

// rearranging things. Oh the humanity!

if (\*(SERCOM \*\*)hwspi.\_spi == &sercom0) {

dmac\_id = SERCOM0\_DMAC\_ID\_TX;

data\_reg = &SERCOM0->SPI.DATA.reg;

#if defined SERCOM1

} else if (\*(SERCOM \*\*)hwspi.\_spi == &sercom1) {

dmac\_id = SERCOM1\_DMAC\_ID\_TX;

data\_reg = &SERCOM1->SPI.DATA.reg;

#endif

#if defined SERCOM2

} else if (\*(SERCOM \*\*)hwspi.\_spi == &sercom2) {

dmac\_id = SERCOM2\_DMAC\_ID\_TX;

data\_reg = &SERCOM2->SPI.DATA.reg;

#endif

#if defined SERCOM3

} else if (\*(SERCOM \*\*)hwspi.\_spi == &sercom3) {

dmac\_id = SERCOM3\_DMAC\_ID\_TX;

data\_reg = &SERCOM3->SPI.DATA.reg;

#endif

#if defined SERCOM4

} else if (\*(SERCOM \*\*)hwspi.\_spi == &sercom4) {

dmac\_id = SERCOM4\_DMAC\_ID\_TX;

data\_reg = &SERCOM4->SPI.DATA.reg;

#endif

#if defined SERCOM5

} else if (\*(SERCOM \*\*)hwspi.\_spi == &sercom5) {

dmac\_id = SERCOM5\_DMAC\_ID\_TX;

data\_reg = &SERCOM5->SPI.DATA.reg;

#endif

#if defined SERCOM6

} else if (\*(SERCOM \*\*)hwspi.\_spi == &sercom6) {

dmac\_id = SERCOM6\_DMAC\_ID\_TX;

data\_reg = &SERCOM6->SPI.DATA.reg;

#endif

#if defined SERCOM7

} else if (\*(SERCOM \*\*)hwspi.\_spi == &sercom7) {

dmac\_id = SERCOM7\_DMAC\_ID\_TX;

data\_reg = &SERCOM7->SPI.DATA.reg;

#endif

}

dma.setPriority(DMA\_PRIORITY\_3);

dma.setTrigger(dmac\_id);

dma.setAction(DMA\_TRIGGER\_ACTON\_BEAT);

// Initialize descriptor list.

for (int d = 0; d < numDescriptors; d++) {

// No need to set SRCADDR, DESCADDR or BTCNT --

// those are done in the pixel-writing functions.

descriptor[d].BTCTRL.bit.VALID = true;

descriptor[d].BTCTRL.bit.EVOSEL = DMA\_EVENT\_OUTPUT\_DISABLE;

descriptor[d].BTCTRL.bit.BLOCKACT = DMA\_BLOCK\_ACTION\_NOACT;

descriptor[d].BTCTRL.bit.BEATSIZE = DMA\_BEAT\_SIZE\_BYTE;

descriptor[d].BTCTRL.bit.DSTINC = 0;

descriptor[d].BTCTRL.bit.STEPSEL = DMA\_STEPSEL\_SRC;

descriptor[d].BTCTRL.bit.STEPSIZE =

DMA\_ADDRESS\_INCREMENT\_STEP\_SIZE\_1;

descriptor[d].DSTADDR.reg = (uint32\_t)data\_reg;

}

} else { // Parallel connection

#if defined(\_\_SAMD51\_\_)

int dmaChannel = dma.getChannel();

// Enable event output, use EVOSEL output

DMAC->Channel[dmaChannel].CHEVCTRL.bit.EVOE = 1;

DMAC->Channel[dmaChannel].CHEVCTRL.bit.EVOMODE = 0;

// CONFIGURE TIMER/COUNTER (for write strobe)

Tc \*timer = tcList[tcNum].tc; // -> Timer struct

int id = tcList[tcNum].gclk; // Timer GCLK ID

GCLK\_PCHCTRL\_Type pchctrl;

// Set up timer clock source from GCLK

GCLK->PCHCTRL[id].bit.CHEN = 0; // Stop timer

while (GCLK->PCHCTRL[id].bit.CHEN)

; // Wait for it

pchctrl.bit.GEN = GCLK\_PCHCTRL\_GEN\_GCLK0\_Val;

pchctrl.bit.CHEN = 1; // Enable

GCLK->PCHCTRL[id].reg = pchctrl.reg;

while (!GCLK->PCHCTRL[id].bit.CHEN)

; // Wait for it

// Disable timer/counter before configuring it

timer->COUNT8.CTRLA.bit.ENABLE = 0;

while (timer->COUNT8.SYNCBUSY.bit.STATUS)

;

timer->COUNT8.WAVE.bit.WAVEGEN = 2; // NPWM

timer->COUNT8.CTRLA.bit.MODE = 1; // 8-bit

timer->COUNT8.CTRLA.bit.PRESCALER = 0; // 1:1

while (timer->COUNT8.SYNCBUSY.bit.STATUS)

;

timer->COUNT8.CTRLBCLR.bit.DIR = 1; // Count UP

while (timer->COUNT8.SYNCBUSY.bit.CTRLB)

;

timer->COUNT8.CTRLBSET.bit.ONESHOT = 1; // One-shot

while (timer->COUNT8.SYNCBUSY.bit.CTRLB)

;

timer->COUNT8.PER.reg = 6; // PWM top

while (timer->COUNT8.SYNCBUSY.bit.PER)

;

timer->COUNT8.CC[0].reg = 2; // Compare

while (timer->COUNT8.SYNCBUSY.bit.CC0)

;

// Enable async input events,

// event action = restart.

timer->COUNT8.EVCTRL.bit.TCEI = 1;

timer->COUNT8.EVCTRL.bit.EVACT = 1;

// Enable timer

timer->COUNT8.CTRLA.reg |= TC\_CTRLA\_ENABLE;

while (timer->COUNT8.SYNCBUSY.bit.STATUS)

;

#if (wrPeripheral == PIO\_CCL)

// CONFIGURE CCL (inverts timer/counter output)

MCLK->APBCMASK.bit.CCL\_ = 1; // Enable CCL clock

CCL->CTRL.bit.ENABLE = 0; // Disable to config

CCL->CTRL.bit.SWRST = 1; // Reset CCL registers

CCL->LUTCTRL[tcNum].bit.ENABLE = 0; // Disable LUT

CCL->LUTCTRL[tcNum].bit.FILTSEL = 0; // No filter

CCL->LUTCTRL[tcNum].bit.INSEL0 = 6; // TC input

CCL->LUTCTRL[tcNum].bit.INSEL1 = 0; // MASK

CCL->LUTCTRL[tcNum].bit.INSEL2 = 0; // MASK

CCL->LUTCTRL[tcNum].bit.TRUTH = 1; // Invert in 0

CCL->LUTCTRL[tcNum].bit.ENABLE = 1; // Enable LUT

CCL->CTRL.bit.ENABLE = 1; // Enable CCL

#endif

// CONFIGURE EVENT SYSTEM

// Set up event system clock source from GCLK...

// Disable EVSYS, wait for disable

GCLK->PCHCTRL[EVSYS\_GCLK\_ID\_0].bit.CHEN = 0;

while (GCLK->PCHCTRL[EVSYS\_GCLK\_ID\_0].bit.CHEN)

;

pchctrl.bit.GEN = GCLK\_PCHCTRL\_GEN\_GCLK0\_Val;

pchctrl.bit.CHEN = 1; // Re-enable

GCLK->PCHCTRL[EVSYS\_GCLK\_ID\_0].reg = pchctrl.reg;

// Wait for it, then enable EVSYS clock

while (!GCLK->PCHCTRL[EVSYS\_GCLK\_ID\_0].bit.CHEN)

;

MCLK->APBBMASK.bit.EVSYS\_ = 1;

// Connect Timer EVU to ch 0

EVSYS->USER[tcList[tcNum].evu].reg = 1;

// Datasheet recommends single write operation;

// reg instead of bit. Also datasheet: PATH bits

// must be zero when using async!

EVSYS\_CHANNEL\_Type ev;

ev.reg = 0;

ev.bit.PATH = 2; // Asynchronous

ev.bit.EVGEN = 0x22 + dmaChannel; // DMA channel 0+

EVSYS->Channel[0].CHANNEL.reg = ev.reg;

// Initialize descriptor list.

for (int d = 0; d < numDescriptors; d++) {

// No need to set SRCADDR, DESCADDR or BTCNT --

// those are done in the pixel-writing functions.

descriptor[d].BTCTRL.bit.VALID = true;

// Event strobe on beat xfer:

descriptor[d].BTCTRL.bit.EVOSEL = 0x3;

descriptor[d].BTCTRL.bit.BLOCKACT = DMA\_BLOCK\_ACTION\_NOACT;

descriptor[d].BTCTRL.bit.BEATSIZE =

tft8.wide ? DMA\_BEAT\_SIZE\_HWORD : DMA\_BEAT\_SIZE\_BYTE;

descriptor[d].BTCTRL.bit.SRCINC = 1;

descriptor[d].BTCTRL.bit.DSTINC = 0;

descriptor[d].BTCTRL.bit.STEPSEL = DMA\_STEPSEL\_SRC;

descriptor[d].BTCTRL.bit.STEPSIZE =

DMA\_ADDRESS\_INCREMENT\_STEP\_SIZE\_1;

descriptor[d].DSTADDR.reg = (uint32\_t)tft8.writePort;

}

#endif // \_\_SAMD51

} // end parallel-specific DMA setup

lastFillColor = 0x0000;

lastFillLen = 0;

dma.setCallback(dma\_callback);

return; // Success!

// else clean up any partial allocation...

} // end descriptor memalign()

free(pixelBuf[0]);

pixelBuf[0] = pixelBuf[1] = NULL;

} // end pixelBuf malloc()

// Don't currently have a descriptor delete function in

// ZeroDMA lib, but if we did, it would be called here.

} // end addDescriptor()

dma.free(); // Deallocate DMA channel

}

#endif // end USE\_SPI\_DMA

}

/\*!

@brief Allow changing the SPI clock speed after initialization

@param freq Desired frequency of SPI clock, may not be the

end frequency you get based on what the chip can do!

\*/

void Adafruit\_SPITFT::setSPISpeed(uint32\_t freq) {

#if defined(SPI\_HAS\_TRANSACTION)

hwspi.settings = SPISettings(freq, MSBFIRST, hwspi.\_mode);

#else

hwspi.\_freq = freq; // Save freq value for later

#endif

}

/\*!

@brief Call before issuing command(s) or data to display. Performs

chip-select (if required) and starts an SPI transaction (if

using hardware SPI and transactions are supported). Required

for all display types; not an SPI-specific function.

\*/

void Adafruit\_SPITFT::startWrite(void) {

SPI\_BEGIN\_TRANSACTION();

if (\_cs >= 0)

SPI\_CS\_LOW();

}

/\*!

@brief Call after issuing command(s) or data to display. Performs

chip-deselect (if required) and ends an SPI transaction (if

using hardware SPI and transactions are supported). Required

for all display types; not an SPI-specific function.

\*/

void Adafruit\_SPITFT::endWrite(void) {

if (\_cs >= 0)

SPI\_CS\_HIGH();

SPI\_END\_TRANSACTION();

}

// -------------------------------------------------------------------------

// Lower-level graphics operations. These functions require a chip-select

// and/or SPI transaction around them (via startWrite(), endWrite() above).

// Higher-level graphics primitives might start a single transaction and

// then make multiple calls to these functions (e.g. circle or text

// rendering might make repeated lines or rects) before ending the

// transaction. It's more efficient than starting a transaction every time.

/\*!

@brief Draw a single pixel to the display at requested coordinates.

Not self-contained; should follow a startWrite() call.

@param x Horizontal position (0 = left).

@param y Vertical position (0 = top).

@param color 16-bit pixel color in '565' RGB format.

\*/

void Adafruit\_SPITFT::writePixel(int16\_t x, int16\_t y, uint16\_t color) {

if ((x >= 0) && (x < \_width) && (y >= 0) && (y < \_height)) {

setAddrWindow(x, y, 1, 1);

SPI\_WRITE16(color);

}

}

/\*!

@brief Swap bytes in an array of pixels; converts little-to-big or

big-to-little endian. Used by writePixels() below in some

situations, but may also be helpful for user code occasionally.

@param src Source address of 16-bit pixels buffer.

@param len Number of pixels to byte-swap.

@param dest Optional destination address if different than src --

otherwise, if NULL (default) or same address is passed,

pixel buffer is overwritten in-place.

\*/

void Adafruit\_SPITFT::swapBytes(uint16\_t \*src, uint32\_t len, uint16\_t \*dest) {

if (!dest)

dest = src; // NULL -> overwrite src buffer

for (uint32\_t i = 0; i < len; i++) {

dest[i] = \_\_builtin\_bswap16(src[i]);

}

}

/\*!

@brief Issue a series of pixels from memory to the display. Not self-

contained; should follow startWrite() and setAddrWindow() calls.

@param colors Pointer to array of 16-bit pixel values in '565' RGB

format.

@param len Number of elements in 'colors' array.

@param block If true (default case if unspecified), function blocks

until DMA transfer is complete. This is simply IGNORED

if DMA is not enabled. If false, the function returns

immediately after the last DMA transfer is started,

and one should use the dmaWait() function before

doing ANY other display-related activities (or even

any SPI-related activities, if using an SPI display

that shares the bus with other devices).

@param bigEndian If true, bitmap in memory is in big-endian order (most

significant byte first). By default this is false, as

most microcontrollers seem to be little-endian and

16-bit pixel values must be byte-swapped before

issuing to the display (which tend toward big-endian

when using SPI or 8-bit parallel). If an application

can optimize around this -- for example, a bitmap in a

uint16\_t array having the byte values already ordered

big-endian, this can save time here, ESPECIALLY if

using this function's non-blocking DMA mode.

\*/

void Adafruit\_SPITFT::writePixels(uint16\_t \*colors, uint32\_t len, bool block,

bool bigEndian) {

if (!len)

return; // Avoid 0-byte transfers

// avoid paramater-not-used complaints

(void)block;

(void)bigEndian;

#if defined(ESP32)

if (connection == TFT\_HARD\_SPI) {

if (!bigEndian) {

hwspi.\_spi->writePixels(colors, len \* 2); // Inbuilt endian-swap

} else {

hwspi.\_spi->writeBytes((uint8\_t \*)colors, len \* 2); // Issue bytes direct

}

return;

}

#elif defined(ARDUINO\_NRF52\_ADAFRUIT) && \

defined(NRF52840\_XXAA) // Adafruit nRF52 use SPIM3 DMA at 32Mhz

if (!bigEndian) {

swapBytes(colors, len); // convert little-to-big endian for display

}

hwspi.\_spi->transfer(colors, NULL, 2 \* len); // NULL RX to avoid overwrite

if (!bigEndian) {

swapBytes(colors, len); // big-to-little endian to restore pixel buffer

}

return;

#elif defined(ARDUINO\_ARCH\_RP2040)

spi\_inst\_t \*pi\_spi = hwspi.\_spi == &SPI ? spi0 : spi1;

if (!bigEndian) {

// switch to 16-bit writes

hw\_write\_masked(&spi\_get\_hw(pi\_spi)->cr0, 15 << SPI\_SSPCR0\_DSS\_LSB,

SPI\_SSPCR0\_DSS\_BITS);

spi\_write16\_blocking(pi\_spi, colors, len);

// switch back to 8-bit

hw\_write\_masked(&spi\_get\_hw(pi\_spi)->cr0, 7 << SPI\_SSPCR0\_DSS\_LSB,

SPI\_SSPCR0\_DSS\_BITS);

} else {

spi\_write\_blocking(pi\_spi, (uint8\_t \*)colors, len \* 2);

}

return;

#elif defined(USE\_SPI\_DMA) && \

(defined(\_\_SAMD51\_\_) || defined(ARDUINO\_SAMD\_ZERO))

if ((connection == TFT\_HARD\_SPI) || (connection == TFT\_PARALLEL)) {

int maxSpan = maxFillLen / 2; // One scanline max

uint8\_t pixelBufIdx = 0; // Active pixel buffer number

#if defined(\_\_SAMD51\_\_)

if (connection == TFT\_PARALLEL) {

// Switch WR pin to PWM or CCL

pinPeripheral(tft8.\_wr, wrPeripheral);

}

#endif // end \_\_SAMD51\_\_

if (!bigEndian) { // Normal little-endian situation...

while (len) {

int count = (len < maxSpan) ? len : maxSpan;

// Because TFT and SAMD endianisms are different, must swap

// bytes from the 'colors' array passed into a DMA working

// buffer. This can take place while the prior DMA transfer

// is in progress, hence the need for two pixelBufs.

swapBytes(colors, count, pixelBuf[pixelBufIdx]);

colors += count;

// The transfers themselves are relatively small, so we don't

// need a long descriptor list. We just alternate between the

// first two, sharing pixelBufIdx for that purpose.

descriptor[pixelBufIdx].SRCADDR.reg =

(uint32\_t)pixelBuf[pixelBufIdx] + count \* 2;

descriptor[pixelBufIdx].BTCTRL.bit.SRCINC = 1;

descriptor[pixelBufIdx].BTCNT.reg = count \* 2;

descriptor[pixelBufIdx].DESCADDR.reg = 0;

while (dma\_busy)

; // Wait for prior line to finish

// Move new descriptor into place...

memcpy(dptr, &descriptor[pixelBufIdx], sizeof(DmacDescriptor));

dma\_busy = true;

dma.startJob(); // Trigger SPI DMA transfer

if (connection == TFT\_PARALLEL)

dma.trigger();

pixelBufIdx = 1 - pixelBufIdx; // Swap DMA pixel buffers

len -= count;

}

} else { // bigEndian == true

// With big-endian pixel data, this can be handled as a single

// DMA transfer using chained descriptors. Even full screen, this

// needs only a relatively short descriptor list, each

// transferring a max of 32,767 (not 32,768) pixels. The list

// was allocated large enough to accommodate a full screen's

// worth of data, so this won't run past the end of the list.

int d, numDescriptors = (len + 32766) / 32767;

for (d = 0; d < numDescriptors; d++) {

int count = (len < 32767) ? len : 32767;

descriptor[d].SRCADDR.reg = (uint32\_t)colors + count \* 2;

descriptor[d].BTCTRL.bit.SRCINC = 1;

descriptor[d].BTCNT.reg = count \* 2;

descriptor[d].DESCADDR.reg = (uint32\_t)&descriptor[d + 1];

len -= count;

colors += count;

}

descriptor[d - 1].DESCADDR.reg = 0;

while (dma\_busy)

; // Wait for prior transfer (if any) to finish

// Move first descriptor into place and start transfer...

memcpy(dptr, &descriptor[0], sizeof(DmacDescriptor));

dma\_busy = true;

dma.startJob(); // Trigger SPI DMA transfer

if (connection == TFT\_PARALLEL)

dma.trigger();

} // end bigEndian

lastFillColor = 0x0000; // pixelBuf has been sullied

lastFillLen = 0;

if (block) {

while (dma\_busy)

; // Wait for last line to complete

#if defined(\_\_SAMD51\_\_) || defined(ARDUINO\_SAMD\_ZERO)

if (connection == TFT\_HARD\_SPI) {

// See SAMD51/21 note in writeColor()

hwspi.\_spi->setDataMode(hwspi.\_mode);

} else {

pinPeripheral(tft8.\_wr, PIO\_OUTPUT); // Switch WR back to GPIO

}

#endif // end \_\_SAMD51\_\_ || ARDUINO\_SAMD\_ZERO

}

return;

}

#endif // end USE\_SPI\_DMA

// All other cases (bitbang SPI or non-DMA hard SPI or parallel),

// use a loop with the normal 16-bit data write function:

if (!bigEndian) {

while (len--) {

SPI\_WRITE16(\*colors++);

}

} else {

// Well this is awkward. SPI\_WRITE16() was designed for little-endian

// hosts and big-endian displays as that's nearly always the typical

// case. If the bigEndian flag was set, data is already in display's

// order...so each pixel needs byte-swapping before being issued.

// Rather than having a separate big-endian SPI\_WRITE16 (adding more

// bloat), it's preferred if calling function is smart and only uses

// bigEndian where DMA is supported. But we gotta handle this...

while (len--) {

SPI\_WRITE16(\_\_builtin\_bswap16(\*colors++));

}

}

}

/\*!

@brief Wait for the last DMA transfer in a prior non-blocking

writePixels() call to complete. This does nothing if DMA

is not enabled, and is not needed if blocking writePixels()

was used (as is the default case).

\*/

void Adafruit\_SPITFT::dmaWait(void) {

#if defined(USE\_SPI\_DMA) && (defined(\_\_SAMD51\_\_) || defined(ARDUINO\_SAMD\_ZERO))

while (dma\_busy)

;

#if defined(\_\_SAMD51\_\_) || defined(ARDUINO\_SAMD\_ZERO)

if (connection == TFT\_HARD\_SPI) {

// See SAMD51/21 note in writeColor()

hwspi.\_spi->setDataMode(hwspi.\_mode);

} else {

pinPeripheral(tft8.\_wr, PIO\_OUTPUT); // Switch WR back to GPIO

}

#endif // end \_\_SAMD51\_\_ || ARDUINO\_SAMD\_ZERO

#endif

}

/\*!

@brief Check if DMA transfer is active. Always returts false if DMA

is not enabled.

@return true if DMA is enabled and transmitting data, false otherwise.

\*/

bool Adafruit\_SPITFT::dmaBusy(void) const {

#if defined(USE\_SPI\_DMA) && (defined(\_\_SAMD51\_\_) || defined(ARDUINO\_SAMD\_ZERO))

return dma\_busy;

#else

return false;

#endif

}

/\*!

@brief Issue a series of pixels, all the same color. Not self-

contained; should follow startWrite() and setAddrWindow() calls.

@param color 16-bit pixel color in '565' RGB format.

@param len Number of pixels to draw.

\*/

void Adafruit\_SPITFT::writeColor(uint16\_t color, uint32\_t len) {

if (!len)

return; // Avoid 0-byte transfers

uint8\_t hi = color >> 8, lo = color;

#if defined(ESP32) // ESP32 has a special SPI pixel-writing function...

if (connection == TFT\_HARD\_SPI) {

#define SPI\_MAX\_PIXELS\_AT\_ONCE 32

#define TMPBUF\_LONGWORDS (SPI\_MAX\_PIXELS\_AT\_ONCE + 1) / 2

#define TMPBUF\_PIXELS (TMPBUF\_LONGWORDS \* 2)

static uint32\_t temp[TMPBUF\_LONGWORDS];

uint32\_t c32 = color \* 0x00010001;

uint16\_t bufLen = (len < TMPBUF\_PIXELS) ? len : TMPBUF\_PIXELS, xferLen,

fillLen;

// Fill temp buffer 32 bits at a time

fillLen = (bufLen + 1) / 2; // Round up to next 32-bit boundary

for (uint32\_t t = 0; t < fillLen; t++) {

temp[t] = c32;

}

// Issue pixels in blocks from temp buffer

while (len) { // While pixels remain

xferLen = (bufLen < len) ? bufLen : len; // How many this pass?

writePixels((uint16\_t \*)temp, xferLen);

len -= xferLen;

}

return;

}

#elif defined(ARDUINO\_NRF52\_ADAFRUIT) && \

defined(NRF52840\_XXAA) // Adafruit nRF52840 use SPIM3 DMA at 32Mhz

// at most 2 scan lines

uint32\_t const pixbufcount = min(len, ((uint32\_t)2 \* width()));

uint16\_t \*pixbuf = (uint16\_t \*)rtos\_malloc(2 \* pixbufcount);

// use SPI3 DMA if we could allocate buffer, else fall back to writing each

// pixel loop below

if (pixbuf) {

uint16\_t const swap\_color = \_\_builtin\_bswap16(color);

// fill buffer with color

for (uint32\_t i = 0; i < pixbufcount; i++) {

pixbuf[i] = swap\_color;

}

while (len) {

uint32\_t const count = min(len, pixbufcount);

writePixels(pixbuf, count, true, true);

len -= count;

}

rtos\_free(pixbuf);

return;

}

#else // !ESP32

#if defined(USE\_SPI\_DMA) && (defined(\_\_SAMD51\_\_) || defined(ARDUINO\_SAMD\_ZERO))

if (((connection == TFT\_HARD\_SPI) || (connection == TFT\_PARALLEL)) &&

(len >= 16)) { // Don't bother with DMA on short pixel runs

int i, d, numDescriptors;

if (hi == lo) { // If high & low bytes are same...

onePixelBuf = color;

// Can do this with a relatively short descriptor list,

// each transferring a max of 32,767 (not 32,768) pixels.

// This won't run off the end of the allocated descriptor list,

// since we're using much larger chunks per descriptor here.

numDescriptors = (len + 32766) / 32767;

for (d = 0; d < numDescriptors; d++) {

int count = (len < 32767) ? len : 32767;

descriptor[d].SRCADDR.reg = (uint32\_t)&onePixelBuf;

descriptor[d].BTCTRL.bit.SRCINC = 0;

descriptor[d].BTCNT.reg = count \* 2;

descriptor[d].DESCADDR.reg = (uint32\_t)&descriptor[d + 1];

len -= count;

}

descriptor[d - 1].DESCADDR.reg = 0;

} else {

// If high and low bytes are distinct, it's necessary to fill

// a buffer with pixel data (swapping high and low bytes because

// TFT and SAMD are different endianisms) and create a longer

// descriptor list pointing repeatedly to this data. We can do

// this slightly faster working 2 pixels (32 bits) at a time.

uint32\_t \*pixelPtr = (uint32\_t \*)pixelBuf[0],

twoPixels = \_\_builtin\_bswap16(color) \* 0x00010001;

// We can avoid some or all of the buffer-filling if the color

// is the same as last time...

if (color == lastFillColor) {

// If length is longer than prior instance, fill only the

// additional pixels in the buffer and update lastFillLen.

if (len > lastFillLen) {

int fillStart = lastFillLen / 2,

fillEnd = (((len < maxFillLen) ? len : maxFillLen) + 1) / 2;

for (i = fillStart; i < fillEnd; i++)

pixelPtr[i] = twoPixels;

lastFillLen = fillEnd \* 2;

} // else do nothing, don't set pixels or change lastFillLen

} else {

int fillEnd = (((len < maxFillLen) ? len : maxFillLen) + 1) / 2;

for (i = 0; i < fillEnd; i++)

pixelPtr[i] = twoPixels;

lastFillLen = fillEnd \* 2;

lastFillColor = color;

}

numDescriptors = (len + maxFillLen - 1) / maxFillLen;

for (d = 0; d < numDescriptors; d++) {

int pixels = (len < maxFillLen) ? len : maxFillLen, bytes = pixels \* 2;

descriptor[d].SRCADDR.reg = (uint32\_t)pixelPtr + bytes;

descriptor[d].BTCTRL.bit.SRCINC = 1;

descriptor[d].BTCNT.reg = bytes;

descriptor[d].DESCADDR.reg = (uint32\_t)&descriptor[d + 1];

len -= pixels;

}

descriptor[d - 1].DESCADDR.reg = 0;

}

memcpy(dptr, &descriptor[0], sizeof(DmacDescriptor));

#if defined(\_\_SAMD51\_\_)

if (connection == TFT\_PARALLEL) {

// Switch WR pin to PWM or CCL

pinPeripheral(tft8.\_wr, wrPeripheral);

}

#endif // end \_\_SAMD51\_\_

dma\_busy = true;

dma.startJob();

if (connection == TFT\_PARALLEL)

dma.trigger();

while (dma\_busy)

; // Wait for completion

// Unfortunately blocking is necessary. An earlier version returned

// immediately and checked dma\_busy on startWrite() instead, but it

// turns out to be MUCH slower on many graphics operations (as when

// drawing lines, pixel-by-pixel), perhaps because it's a volatile

// type and doesn't cache. Working on this.

#if defined(\_\_SAMD51\_\_) || defined(ARDUINO\_SAMD\_ZERO)

if (connection == TFT\_HARD\_SPI) {

// SAMD51: SPI DMA seems to leave the SPI peripheral in a freaky

// state on completion. Workaround is to explicitly set it back...

// (5/17/2019: apparently SAMD21 too, in certain cases, observed

// with ST7789 display.)

hwspi.\_spi->setDataMode(hwspi.\_mode);

} else {

pinPeripheral(tft8.\_wr, PIO\_OUTPUT); // Switch WR back to GPIO

}

#endif // end \_\_SAMD51\_\_

return;

}

#endif // end USE\_SPI\_DMA

#endif // end !ESP32

// All other cases (non-DMA hard SPI, bitbang SPI, parallel)...

if (connection == TFT\_HARD\_SPI) {

#if defined(ESP8266)

do {

uint32\_t pixelsThisPass = len;

if (pixelsThisPass > 50000)

pixelsThisPass = 50000;

len -= pixelsThisPass;

delay(1); // Periodic delay on long fills

while (pixelsThisPass--) {

hwspi.\_spi->write(hi);

hwspi.\_spi->write(lo);

}

} while (len);

#elif defined(ARDUINO\_ARCH\_RP2040)

spi\_inst\_t \*pi\_spi = hwspi.\_spi == &SPI ? spi0 : spi1;

color = \_\_builtin\_bswap16(color);

while (len--)

spi\_write\_blocking(pi\_spi, (uint8\_t \*)&color, 2);

#else // !ESP8266 && !ARDUINO\_ARCH\_RP2040

while (len--) {

#if defined(\_\_AVR\_\_)

AVR\_WRITESPI(hi);

AVR\_WRITESPI(lo);

#elif defined(ESP32)

hwspi.\_spi->write(hi);

hwspi.\_spi->write(lo);

#else

hwspi.\_spi->transfer(hi);

hwspi.\_spi->transfer(lo);

#endif

}

#endif // end !ESP8266

} else if (connection == TFT\_SOFT\_SPI) {

#if defined(ESP8266)

do {

uint32\_t pixelsThisPass = len;

if (pixelsThisPass > 20000)

pixelsThisPass = 20000;

len -= pixelsThisPass;

yield(); // Periodic yield() on long fills

while (pixelsThisPass--) {

for (uint16\_t bit = 0, x = color; bit < 16; bit++) {

if (x & 0x8000)

SPI\_MOSI\_HIGH();

else

SPI\_MOSI\_LOW();

SPI\_SCK\_HIGH();

SPI\_SCK\_LOW();

x <<= 1;

}

}

} while (len);

#else // !ESP8266

while (len--) {

#if defined(\_\_AVR\_\_)

for (uint8\_t bit = 0, x = hi; bit < 8; bit++) {

if (x & 0x80)

SPI\_MOSI\_HIGH();

else

SPI\_MOSI\_LOW();

SPI\_SCK\_HIGH();

SPI\_SCK\_LOW();

x <<= 1;

}

for (uint8\_t bit = 0, x = lo; bit < 8; bit++) {

if (x & 0x80)

SPI\_MOSI\_HIGH();

else

SPI\_MOSI\_LOW();

SPI\_SCK\_HIGH();

SPI\_SCK\_LOW();

x <<= 1;

}

#else // !\_\_AVR\_\_

for (uint16\_t bit = 0, x = color; bit < 16; bit++) {

if (x & 0x8000)

SPI\_MOSI\_HIGH();

else

SPI\_MOSI\_LOW();

SPI\_SCK\_HIGH();

x <<= 1;

SPI\_SCK\_LOW();

}

#endif // end !\_\_AVR\_\_

}

#endif // end !ESP8266

} else { // PARALLEL

if (hi == lo) {

#if defined(\_\_AVR\_\_)

len \*= 2;

\*tft8.writePort = hi;

while (len--) {

TFT\_WR\_STROBE();

}

#elif defined(USE\_FAST\_PINIO)

if (!tft8.wide) {

len \*= 2;

\*tft8.writePort = hi;

} else {

\*(volatile uint16\_t \*)tft8.writePort = color;

}

while (len--) {

TFT\_WR\_STROBE();

}

#endif

} else {

while (len--) {

#if defined(\_\_AVR\_\_)

\*tft8.writePort = hi;

TFT\_WR\_STROBE();

\*tft8.writePort = lo;

#elif defined(USE\_FAST\_PINIO)

if (!tft8.wide) {

\*tft8.writePort = hi;

TFT\_WR\_STROBE();

\*tft8.writePort = lo;

} else {

\*(volatile uint16\_t \*)tft8.writePort = color;

}

#endif

TFT\_WR\_STROBE();

}

}

}

}

/\*!

@brief Draw a filled rectangle to the display. Not self-contained;

should follow startWrite(). Typically used by higher-level

graphics primitives; user code shouldn't need to call this and

is likely to use the self-contained fillRect() instead.

writeFillRect() performs its own edge clipping and rejection;

see writeFillRectPreclipped() for a more 'raw' implementation.

@param x Horizontal position of first corner.

@param y Vertical position of first corner.

@param w Rectangle width in pixels (positive = right of first

corner, negative = left of first corner).

@param h Rectangle height in pixels (positive = below first

corner, negative = above first corner).

@param color 16-bit fill color in '565' RGB format.

@note Written in this deep-nested way because C by definition will

optimize for the 'if' case, not the 'else' -- avoids branches

and rejects clipped rectangles at the least-work possibility.

\*/

void Adafruit\_SPITFT::writeFillRect(int16\_t x, int16\_t y, int16\_t w, int16\_t h,

uint16\_t color) {

if (w && h) { // Nonzero width and height?

if (w < 0) { // If negative width...

x += w + 1; // Move X to left edge

w = -w; // Use positive width

}

if (x < \_width) { // Not off right

if (h < 0) { // If negative height...

y += h + 1; // Move Y to top edge

h = -h; // Use positive height

}

if (y < \_height) { // Not off bottom

int16\_t x2 = x + w - 1;

if (x2 >= 0) { // Not off left

int16\_t y2 = y + h - 1;

if (y2 >= 0) { // Not off top

// Rectangle partly or fully overlaps screen

if (x < 0) {

x = 0;

w = x2 + 1;

} // Clip left

if (y < 0) {

y = 0;

h = y2 + 1;

} // Clip top

if (x2 >= \_width) {

w = \_width - x;

} // Clip right

if (y2 >= \_height) {

h = \_height - y;

} // Clip bottom

writeFillRectPreclipped(x, y, w, h, color);

}

}

}

}

}

}

/\*!

@brief Draw a horizontal line on the display. Performs edge clipping

and rejection. Not self-contained; should follow startWrite().

Typically used by higher-level graphics primitives; user code

shouldn't need to call this and is likely to use the self-

contained drawFastHLine() instead.

@param x Horizontal position of first point.

@param y Vertical position of first point.

@param w Line width in pixels (positive = right of first point,

negative = point of first corner).

@param color 16-bit line color in '565' RGB format.

\*/

void inline Adafruit\_SPITFT::writeFastHLine(int16\_t x, int16\_t y, int16\_t w,

uint16\_t color) {

if ((y >= 0) && (y < \_height) && w) { // Y on screen, nonzero width

if (w < 0) { // If negative width...

x += w + 1; // Move X to left edge

w = -w; // Use positive width

}

if (x < \_width) { // Not off right

int16\_t x2 = x + w - 1;

if (x2 >= 0) { // Not off left

// Line partly or fully overlaps screen

if (x < 0) {

x = 0;

w = x2 + 1;

} // Clip left

if (x2 >= \_width) {

w = \_width - x;

} // Clip right

writeFillRectPreclipped(x, y, w, 1, color);

}

}

}

}

/\*!

@brief Draw a vertical line on the display. Performs edge clipping and

rejection. Not self-contained; should follow startWrite().

Typically used by higher-level graphics primitives; user code

shouldn't need to call this and is likely to use the self-

contained drawFastVLine() instead.

@param x Horizontal position of first point.

@param y Vertical position of first point.

@param h Line height in pixels (positive = below first point,

negative = above first point).

@param color 16-bit line color in '565' RGB format.

\*/

void inline Adafruit\_SPITFT::writeFastVLine(int16\_t x, int16\_t y, int16\_t h,

uint16\_t color) {

if ((x >= 0) && (x < \_width) && h) { // X on screen, nonzero height

if (h < 0) { // If negative height...

y += h + 1; // Move Y to top edge

h = -h; // Use positive height

}

if (y < \_height) { // Not off bottom

int16\_t y2 = y + h - 1;

if (y2 >= 0) { // Not off top

// Line partly or fully overlaps screen

if (y < 0) {

y = 0;

h = y2 + 1;

} // Clip top

if (y2 >= \_height) {

h = \_height - y;

} // Clip bottom

writeFillRectPreclipped(x, y, 1, h, color);

}

}

}

}

/\*!

@brief A lower-level version of writeFillRect(). This version requires

all inputs are in-bounds, that width and height are positive,

and no part extends offscreen. NO EDGE CLIPPING OR REJECTION IS

PERFORMED. If higher-level graphics primitives are written to

handle their own clipping earlier in the drawing process, this

can avoid unnecessary function calls and repeated clipping

operations in the lower-level functions.

@param x Horizontal position of first corner. MUST BE WITHIN

SCREEN BOUNDS.

@param y Vertical position of first corner. MUST BE WITHIN SCREEN

BOUNDS.

@param w Rectangle width in pixels. MUST BE POSITIVE AND NOT

EXTEND OFF SCREEN.

@param h Rectangle height in pixels. MUST BE POSITIVE AND NOT

EXTEND OFF SCREEN.

@param color 16-bit fill color in '565' RGB format.

@note This is a new function, no graphics primitives besides rects

and horizontal/vertical lines are written to best use this yet.

\*/

inline void Adafruit\_SPITFT::writeFillRectPreclipped(int16\_t x, int16\_t y,

int16\_t w, int16\_t h,

uint16\_t color) {

setAddrWindow(x, y, w, h);

writeColor(color, (uint32\_t)w \* h);

}

// -------------------------------------------------------------------------

// Ever-so-slightly higher-level graphics operations. Similar to the 'write'

// functions above, but these contain their own chip-select and SPI

// transactions as needed (via startWrite(), endWrite()). They're typically

// used solo -- as graphics primitives in themselves, not invoked by higher-

// level primitives (which should use the functions above for better

// performance).

/\*!

@brief Draw a single pixel to the display at requested coordinates.

Self-contained and provides its own transaction as needed

(see writePixel(x,y,color) for a lower-level variant).

Edge clipping is performed here.

@param x Horizontal position (0 = left).

@param y Vertical position (0 = top).

@param color 16-bit pixel color in '565' RGB format.

\*/

void Adafruit\_SPITFT::drawPixel(int16\_t x, int16\_t y, uint16\_t color) {

// Clip first...

if ((x >= 0) && (x < \_width) && (y >= 0) && (y < \_height)) {

// THEN set up transaction (if needed) and draw...

startWrite();

setAddrWindow(x, y, 1, 1);

SPI\_WRITE16(color);

endWrite();

}

}

/\*!

@brief Draw a filled rectangle to the display. Self-contained and

provides its own transaction as needed (see writeFillRect() or

writeFillRectPreclipped() for lower-level variants). Edge

clipping and rejection is performed here.

@param x Horizontal position of first corner.

@param y Vertical position of first corner.

@param w Rectangle width in pixels (positive = right of first

corner, negative = left of first corner).

@param h Rectangle height in pixels (positive = below first

corner, negative = above first corner).

@param color 16-bit fill color in '565' RGB format.

@note This repeats the writeFillRect() function almost in its entirety,

with the addition of a transaction start/end. It's done this way

(rather than starting the transaction and calling writeFillRect()

to handle clipping and so forth) so that the transaction isn't

performed at all if the rectangle is rejected. It's really not

that much code.

\*/

void Adafruit\_SPITFT::fillRect(int16\_t x, int16\_t y, int16\_t w, int16\_t h,

uint16\_t color) {

if (w && h) { // Nonzero width and height?

if (w < 0) { // If negative width...

x += w + 1; // Move X to left edge

w = -w; // Use positive width

}

if (x < \_width) { // Not off right

if (h < 0) { // If negative height...

y += h + 1; // Move Y to top edge

h = -h; // Use positive height

}

if (y < \_height) { // Not off bottom

int16\_t x2 = x + w - 1;

if (x2 >= 0) { // Not off left

int16\_t y2 = y + h - 1;

if (y2 >= 0) { // Not off top

// Rectangle partly or fully overlaps screen

if (x < 0) {

x = 0;

w = x2 + 1;

} // Clip left

if (y < 0) {

y = 0;

h = y2 + 1;

} // Clip top

if (x2 >= \_width) {

w = \_width - x;

} // Clip right

if (y2 >= \_height) {

h = \_height - y;

} // Clip bottom

startWrite();

writeFillRectPreclipped(x, y, w, h, color);

endWrite();

}

}

}

}

}

}

/\*!

@brief Draw a horizontal line on the display. Self-contained and

provides its own transaction as needed (see writeFastHLine() for

a lower-level variant). Edge clipping and rejection is performed

here.

@param x Horizontal position of first point.

@param y Vertical position of first point.

@param w Line width in pixels (positive = right of first point,

negative = point of first corner).

@param color 16-bit line color in '565' RGB format.

@note This repeats the writeFastHLine() function almost in its

entirety, with the addition of a transaction start/end. It's

done this way (rather than starting the transaction and calling

writeFastHLine() to handle clipping and so forth) so that the

transaction isn't performed at all if the line is rejected.

\*/

void Adafruit\_SPITFT::drawFastHLine(int16\_t x, int16\_t y, int16\_t w,

uint16\_t color) {

if ((y >= 0) && (y < \_height) && w) { // Y on screen, nonzero width

if (w < 0) { // If negative width...

x += w + 1; // Move X to left edge

w = -w; // Use positive width

}

if (x < \_width) { // Not off right

int16\_t x2 = x + w - 1;

if (x2 >= 0) { // Not off left

// Line partly or fully overlaps screen

if (x < 0) {

x = 0;

w = x2 + 1;

} // Clip left

if (x2 >= \_width) {

w = \_width - x;

} // Clip right

startWrite();

writeFillRectPreclipped(x, y, w, 1, color);

endWrite();

}

}

}

}

/\*!

@brief Draw a vertical line on the display. Self-contained and provides

its own transaction as needed (see writeFastHLine() for a lower-

level variant). Edge clipping and rejection is performed here.

@param x Horizontal position of first point.

@param y Vertical position of first point.

@param h Line height in pixels (positive = below first point,

negative = above first point).

@param color 16-bit line color in '565' RGB format.

@note This repeats the writeFastVLine() function almost in its

entirety, with the addition of a transaction start/end. It's

done this way (rather than starting the transaction and calling

writeFastVLine() to handle clipping and so forth) so that the

transaction isn't performed at all if the line is rejected.

\*/

void Adafruit\_SPITFT::drawFastVLine(int16\_t x, int16\_t y, int16\_t h,

uint16\_t color) {

if ((x >= 0) && (x < \_width) && h) { // X on screen, nonzero height

if (h < 0) { // If negative height...

y += h + 1; // Move Y to top edge

h = -h; // Use positive height

}

if (y < \_height) { // Not off bottom

int16\_t y2 = y + h - 1;

if (y2 >= 0) { // Not off top

// Line partly or fully overlaps screen

if (y < 0) {

y = 0;

h = y2 + 1;

} // Clip top

if (y2 >= \_height) {

h = \_height - y;

} // Clip bottom

startWrite();

writeFillRectPreclipped(x, y, 1, h, color);

endWrite();

}

}

}

}

/\*!

@brief Essentially writePixel() with a transaction around it. I don't

think this is in use by any of our code anymore (believe it was

for some older BMP-reading examples), but is kept here in case

any user code relies on it. Consider it DEPRECATED.

@param color 16-bit pixel color in '565' RGB format.

\*/

void Adafruit\_SPITFT::pushColor(uint16\_t color) {

startWrite();

SPI\_WRITE16(color);

endWrite();

}

/\*!

@brief Draw a 16-bit image (565 RGB) at the specified (x,y) position.

For 16-bit display devices; no color reduction performed.

Adapted from https://github.com/PaulStoffregen/ILI9341\_t3

by Marc MERLIN. See examples/pictureEmbed to use this.

5/6/2017: function name and arguments have changed for

compatibility with current GFX library and to avoid naming

problems in prior implementation. Formerly drawBitmap() with

arguments in different order. Handles its own transaction and

edge clipping/rejection.

@param x Top left corner horizontal coordinate.

@param y Top left corner vertical coordinate.

@param pcolors Pointer to 16-bit array of pixel values.

@param w Width of bitmap in pixels.

@param h Height of bitmap in pixels.

\*/

void Adafruit\_SPITFT::drawRGBBitmap(int16\_t x, int16\_t y, uint16\_t \*pcolors,

int16\_t w, int16\_t h) {

int16\_t x2, y2; // Lower-right coord

if ((x >= \_width) || // Off-edge right

(y >= \_height) || // " top

((x2 = (x + w - 1)) < 0) || // " left

((y2 = (y + h - 1)) < 0))

return; // " bottom

int16\_t bx1 = 0, by1 = 0, // Clipped top-left within bitmap

saveW = w; // Save original bitmap width value

if (x < 0) { // Clip left

w += x;

bx1 = -x;

x = 0;

}

if (y < 0) { // Clip top

h += y;

by1 = -y;

y = 0;

}

if (x2 >= \_width)

w = \_width - x; // Clip right

if (y2 >= \_height)

h = \_height - y; // Clip bottom

pcolors += by1 \* saveW + bx1; // Offset bitmap ptr to clipped top-left

startWrite();

setAddrWindow(x, y, w, h); // Clipped area

while (h--) { // For each (clipped) scanline...

writePixels(pcolors, w); // Push one (clipped) row

pcolors += saveW; // Advance pointer by one full (unclipped) line

}

endWrite();

}

// -------------------------------------------------------------------------

// Miscellaneous class member functions that don't draw anything.

/\*!

@brief Invert the colors of the display (if supported by hardware).

Self-contained, no transaction setup required.

@param i true = inverted display, false = normal display.

\*/

void Adafruit\_SPITFT::invertDisplay(bool i) {

startWrite();

writeCommand(i ? invertOnCommand : invertOffCommand);

endWrite();

}

/\*!

@brief Given 8-bit red, green and blue values, return a 'packed'

16-bit color value in '565' RGB format (5 bits red, 6 bits

green, 5 bits blue). This is just a mathematical operation,

no hardware is touched.

@param red 8-bit red brightnesss (0 = off, 255 = max).

@param green 8-bit green brightnesss (0 = off, 255 = max).

@param blue 8-bit blue brightnesss (0 = off, 255 = max).

@return 'Packed' 16-bit color value (565 format).

\*/

uint16\_t Adafruit\_SPITFT::color565(uint8\_t red, uint8\_t green, uint8\_t blue) {

return ((red & 0xF8) << 8) | ((green & 0xFC) << 3) | (blue >> 3);

}

/\*!

@brief Adafruit\_SPITFT Send Command handles complete sending of commands and

data

@param commandByte The Command Byte

@param dataBytes A pointer to the Data bytes to send

@param numDataBytes The number of bytes we should send

\*/

void Adafruit\_SPITFT::sendCommand(uint8\_t commandByte, uint8\_t \*dataBytes,

uint8\_t numDataBytes) {

SPI\_BEGIN\_TRANSACTION();

if (\_cs >= 0)

SPI\_CS\_LOW();

SPI\_DC\_LOW(); // Command mode

spiWrite(commandByte); // Send the command byte

SPI\_DC\_HIGH();

for (int i = 0; i < numDataBytes; i++) {

if ((connection == TFT\_PARALLEL) && tft8.wide) {

SPI\_WRITE16(\*(uint16\_t \*)dataBytes);

dataBytes += 2;

} else {

spiWrite(\*dataBytes); // Send the data bytes

dataBytes++;

}

}

if (\_cs >= 0)

SPI\_CS\_HIGH();

SPI\_END\_TRANSACTION();

}

/\*!

@brief Adafruit\_SPITFT Send Command handles complete sending of commands and

data

@param commandByte The Command Byte

@param dataBytes A pointer to the Data bytes to send

@param numDataBytes The number of bytes we should send

\*/

void Adafruit\_SPITFT::sendCommand(uint8\_t commandByte, const uint8\_t \*dataBytes,

uint8\_t numDataBytes) {

SPI\_BEGIN\_TRANSACTION();

if (\_cs >= 0)

SPI\_CS\_LOW();

SPI\_DC\_LOW(); // Command mode

spiWrite(commandByte); // Send the command byte

SPI\_DC\_HIGH();

for (int i = 0; i < numDataBytes; i++) {

if ((connection == TFT\_PARALLEL) && tft8.wide) {

SPI\_WRITE16(\*(uint16\_t \*)dataBytes);

dataBytes += 2;

} else {

spiWrite(pgm\_read\_byte(dataBytes++));

}

}

if (\_cs >= 0)

SPI\_CS\_HIGH();

SPI\_END\_TRANSACTION();

}

/\*!

@brief Adafruit\_SPITFT sendCommand16 handles complete sending of

commands and data for 16-bit parallel displays. Currently somewhat

rigged for the NT35510, which has the odd behavior of wanting

commands 16-bit, but subsequent data as 8-bit values, despite

the 16-bit bus (high byte is always 0). Also seems to require

issuing and incrementing address with each transfer.

@param commandWord The command word (16 bits)

@param dataBytes A pointer to the data bytes to send

@param numDataBytes The number of bytes we should send

\*/

void Adafruit\_SPITFT::sendCommand16(uint16\_t commandWord,

const uint8\_t \*dataBytes,

uint8\_t numDataBytes) {

SPI\_BEGIN\_TRANSACTION();

if (\_cs >= 0)

SPI\_CS\_LOW();

if (numDataBytes == 0) {

SPI\_DC\_LOW(); // Command mode

SPI\_WRITE16(commandWord); // Send the command word

SPI\_DC\_HIGH(); // Data mode

}

for (int i = 0; i < numDataBytes; i++) {

SPI\_DC\_LOW(); // Command mode

SPI\_WRITE16(commandWord); // Send the command word

SPI\_DC\_HIGH(); // Data mode

commandWord++;

SPI\_WRITE16((uint16\_t)pgm\_read\_byte(dataBytes++));

}

if (\_cs >= 0)

SPI\_CS\_HIGH();

SPI\_END\_TRANSACTION();

}

/\*!

@brief Read 8 bits of data from display configuration memory (not RAM).

This is highly undocumented/supported and should be avoided,

function is only included because some of the examples use it.

@param commandByte

The command register to read data from.

@param index

The byte index into the command to read from.

@return Unsigned 8-bit data read from display register.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

uint8\_t Adafruit\_SPITFT::readcommand8(uint8\_t commandByte, uint8\_t index) {

uint8\_t result;

startWrite();

SPI\_DC\_LOW(); // Command mode

spiWrite(commandByte);

SPI\_DC\_HIGH(); // Data mode

do {

result = spiRead();

} while (index--); // Discard bytes up to index'th

endWrite();

return result;

}

/\*!

@brief Read 16 bits of data from display register.

For 16-bit parallel displays only.

@param addr Command/register to access.

@return Unsigned 16-bit data.

\*/

uint16\_t Adafruit\_SPITFT::readcommand16(uint16\_t addr) {

#if defined(USE\_FAST\_PINIO) // NOT SUPPORTED without USE\_FAST\_PINIO

uint16\_t result = 0;

if ((connection == TFT\_PARALLEL) && tft8.wide) {

startWrite();

SPI\_DC\_LOW(); // Command mode

SPI\_WRITE16(addr);

SPI\_DC\_HIGH(); // Data mode

TFT\_RD\_LOW(); // Read line LOW

#if defined(HAS\_PORT\_SET\_CLR)

\*(volatile uint16\_t \*)tft8.dirClr = 0xFFFF; // Input state

result = \*(volatile uint16\_t \*)tft8.readPort; // 16-bit read

\*(volatile uint16\_t \*)tft8.dirSet = 0xFFFF; // Output state

#else // !HAS\_PORT\_SET\_CLR

\*(volatile uint16\_t \*)tft8.portDir = 0x0000; // Input state

result = \*(volatile uint16\_t \*)tft8.readPort; // 16-bit read

\*(volatile uint16\_t \*)tft8.portDir = 0xFFFF; // Output state

#endif // end !HAS\_PORT\_SET\_CLR

TFT\_RD\_HIGH(); // Read line HIGH

endWrite();

}

return result;

#else

(void)addr; // disable -Wunused-parameter warning

return 0;

#endif // end !USE\_FAST\_PINIO

}

// -------------------------------------------------------------------------

// Lowest-level hardware-interfacing functions. Many of these are inline and

// compile to different things based on #defines -- typically just a few

// instructions. Others, not so much, those are not inlined.

/\*!

@brief Start an SPI transaction if using the hardware SPI interface to

the display. If using an earlier version of the Arduino platform

(before the addition of SPI transactions), this instead attempts

to set up the SPI clock and mode. No action is taken if the

connection is not hardware SPI-based. This does NOT include a

chip-select operation -- see startWrite() for a function that

encapsulated both actions.

\*/

inline void Adafruit\_SPITFT::SPI\_BEGIN\_TRANSACTION(void) {

if (connection == TFT\_HARD\_SPI) {

#if defined(SPI\_HAS\_TRANSACTION)

hwspi.\_spi->beginTransaction(hwspi.settings);

#else // No transactions, configure SPI manually...

#if defined(\_\_AVR\_\_) || defined(TEENSYDUINO) || defined(ARDUINO\_ARCH\_STM32F1)

hwspi.\_spi->setClockDivider(SPI\_CLOCK\_DIV2);

#elif defined(\_\_arm\_\_)

hwspi.\_spi->setClockDivider(11);

#elif defined(ESP8266) || defined(ESP32)

hwspi.\_spi->setFrequency(hwspi.\_freq);

#elif defined(RASPI) || defined(ARDUINO\_ARCH\_STM32F1)

hwspi.\_spi->setClock(hwspi.\_freq);

#endif

hwspi.\_spi->setBitOrder(MSBFIRST);

hwspi.\_spi->setDataMode(hwspi.\_mode);

#endif // end !SPI\_HAS\_TRANSACTION

}

}

/\*!

@brief End an SPI transaction if using the hardware SPI interface to

the display. No action is taken if the connection is not

hardware SPI-based or if using an earlier version of the Arduino

platform (before the addition of SPI transactions). This does

NOT include a chip-deselect operation -- see endWrite() for a

function that encapsulated both actions.

\*/

inline void Adafruit\_SPITFT::SPI\_END\_TRANSACTION(void) {

#if defined(SPI\_HAS\_TRANSACTION)

if (connection == TFT\_HARD\_SPI) {

hwspi.\_spi->endTransaction();

}

#endif

}

/\*!

@brief Issue a single 8-bit value to the display. Chip-select,

transaction and data/command selection must have been

previously set -- this ONLY issues the byte. This is another of

those functions in the library with a now-not-accurate name

that's being maintained for compatibility with outside code.

This function is used even if display connection is parallel.

@param b 8-bit value to write.

\*/

void Adafruit\_SPITFT::spiWrite(uint8\_t b) {

if (connection == TFT\_HARD\_SPI) {

#if defined(\_\_AVR\_\_)

AVR\_WRITESPI(b);

#elif defined(ESP8266) || defined(ESP32)

hwspi.\_spi->write(b);

#elif defined(ARDUINO\_ARCH\_RP2040)

spi\_inst\_t \*pi\_spi = hwspi.\_spi == &SPI ? spi0 : spi1;

spi\_write\_blocking(pi\_spi, &b, 1);

#else

hwspi.\_spi->transfer(b);

#endif

} else if (connection == TFT\_SOFT\_SPI) {

for (uint8\_t bit = 0; bit < 8; bit++) {

if (b & 0x80)

SPI\_MOSI\_HIGH();

else

SPI\_MOSI\_LOW();

SPI\_SCK\_HIGH();

b <<= 1;

SPI\_SCK\_LOW();

}

} else { // TFT\_PARALLEL

#if defined(\_\_AVR\_\_)

\*tft8.writePort = b;

#elif defined(USE\_FAST\_PINIO)

if (!tft8.wide)

\*tft8.writePort = b;

else

\*(volatile uint16\_t \*)tft8.writePort = b;

#endif

TFT\_WR\_STROBE();

}

}

/\*!

@brief Write a single command byte to the display. Chip-select and

transaction must have been previously set -- this ONLY sets

the device to COMMAND mode, issues the byte and then restores

DATA mode. There is no corresponding explicit writeData()

function -- just use spiWrite().

@param cmd 8-bit command to write.

\*/

void Adafruit\_SPITFT::writeCommand(uint8\_t cmd) {

SPI\_DC\_LOW();

spiWrite(cmd);

SPI\_DC\_HIGH();

}

/\*!

@brief Read a single 8-bit value from the display. Chip-select and

transaction must have been previously set -- this ONLY reads

the byte. This is another of those functions in the library

with a now-not-accurate name that's being maintained for

compatibility with outside code. This function is used even if

display connection is parallel.

@return Unsigned 8-bit value read (always zero if USE\_FAST\_PINIO is

not supported by the MCU architecture).

\*/

uint8\_t Adafruit\_SPITFT::spiRead(void) {

uint8\_t b = 0;

uint16\_t w = 0;

if (connection == TFT\_HARD\_SPI) {

return hwspi.\_spi->transfer((uint8\_t)0);

} else if (connection == TFT\_SOFT\_SPI) {

if (swspi.\_miso >= 0) {

for (uint8\_t i = 0; i < 8; i++) {

SPI\_SCK\_HIGH();

b <<= 1;

if (SPI\_MISO\_READ())

b++;

SPI\_SCK\_LOW();

}

}

return b;

} else { // TFT\_PARALLEL

if (tft8.\_rd >= 0) {

#if defined(USE\_FAST\_PINIO)

TFT\_RD\_LOW(); // Read line LOW

#if defined(\_\_AVR\_\_)

\*tft8.portDir = 0x00; // Set port to input state

w = \*tft8.readPort; // Read value from port

\*tft8.portDir = 0xFF; // Restore port to output

#else // !\_\_AVR\_\_

if (!tft8.wide) { // 8-bit TFT connection

#if defined(HAS\_PORT\_SET\_CLR)

\*tft8.dirClr = 0xFF; // Set port to input state

w = \*tft8.readPort; // Read value from port

\*tft8.dirSet = 0xFF; // Restore port to output

#else // !HAS\_PORT\_SET\_CLR

\*tft8.portDir = 0x00; // Set port to input state

w = \*tft8.readPort; // Read value from port

\*tft8.portDir = 0xFF; // Restore port to output

#endif // end HAS\_PORT\_SET\_CLR

} else { // 16-bit TFT connection

#if defined(HAS\_PORT\_SET\_CLR)

\*(volatile uint16\_t \*)tft8.dirClr = 0xFFFF; // Input state

w = \*(volatile uint16\_t \*)tft8.readPort; // 16-bit read

\*(volatile uint16\_t \*)tft8.dirSet = 0xFFFF; // Output state

#else // !HAS\_PORT\_SET\_CLR

\*(volatile uint16\_t \*)tft8.portDir = 0x0000; // Input state

w = \*(volatile uint16\_t \*)tft8.readPort; // 16-bit read

\*(volatile uint16\_t \*)tft8.portDir = 0xFFFF; // Output state

#endif // end !HAS\_PORT\_SET\_CLR

}

TFT\_RD\_HIGH(); // Read line HIGH

#endif // end !\_\_AVR\_\_

#else // !USE\_FAST\_PINIO

w = 0; // Parallel TFT is NOT SUPPORTED without USE\_FAST\_PINIO

#endif // end !USE\_FAST\_PINIO

}

return w;

}

}

/\*!

@brief Issue a single 16-bit value to the display. Chip-select,

transaction and data/command selection must have been

previously set -- this ONLY issues the word.

Thus operates ONLY on 'wide' (16-bit) parallel displays!

@param w 16-bit value to write.

\*/

void Adafruit\_SPITFT::write16(uint16\_t w) {

if (connection == TFT\_PARALLEL) {

#if defined(USE\_FAST\_PINIO)

if (tft8.wide)

\*(volatile uint16\_t \*)tft8.writePort = w;

#else

(void)w; // disable -Wunused-parameter warning

#endif

TFT\_WR\_STROBE();

}

}

/\*!

@brief Write a single command word to the display. Chip-select and

transaction must have been previously set -- this ONLY sets

the device to COMMAND mode, issues the byte and then restores

DATA mode. This operates ONLY on 'wide' (16-bit) parallel

displays!

@param cmd 16-bit command to write.

\*/

void Adafruit\_SPITFT::writeCommand16(uint16\_t cmd) {

SPI\_DC\_LOW();

write16(cmd);

SPI\_DC\_HIGH();

}

/\*!

@brief Read a single 16-bit value from the display. Chip-select and

transaction must have been previously set -- this ONLY reads

the byte. This operates ONLY on 'wide' (16-bit) parallel

displays!

@return Unsigned 16-bit value read (always zero if USE\_FAST\_PINIO is

not supported by the MCU architecture).

\*/

uint16\_t Adafruit\_SPITFT::read16(void) {

uint16\_t w = 0;

if (connection == TFT\_PARALLEL) {

if (tft8.\_rd >= 0) {

#if defined(USE\_FAST\_PINIO)

TFT\_RD\_LOW(); // Read line LOW

if (tft8.wide) { // 16-bit TFT connection

#if defined(HAS\_PORT\_SET\_CLR)

\*(volatile uint16\_t \*)tft8.dirClr = 0xFFFF; // Input state

w = \*(volatile uint16\_t \*)tft8.readPort; // 16-bit read

\*(volatile uint16\_t \*)tft8.dirSet = 0xFFFF; // Output state

#else // !HAS\_PORT\_SET\_CLR

\*(volatile uint16\_t \*)tft8.portDir = 0x0000; // Input state

w = \*(volatile uint16\_t \*)tft8.readPort; // 16-bit read

\*(volatile uint16\_t \*)tft8.portDir = 0xFFFF; // Output state

#endif // end !HAS\_PORT\_SET\_CLR

}

TFT\_RD\_HIGH(); // Read line HIGH

#else // !USE\_FAST\_PINIO

w = 0; // Parallel TFT is NOT SUPPORTED without USE\_FAST\_PINIO

#endif // end !USE\_FAST\_PINIO

}

}

return w;

}

/\*!

@brief Set the software (bitbang) SPI MOSI line HIGH.

\*/

inline void Adafruit\_SPITFT::SPI\_MOSI\_HIGH(void) {

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

#if defined(KINETISK)

\*swspi.mosiPortSet = 1;

#else // !KINETISK

\*swspi.mosiPortSet = swspi.mosiPinMask;

#endif

#else // !HAS\_PORT\_SET\_CLR

\*swspi.mosiPort |= swspi.mosiPinMaskSet;

#endif // end !HAS\_PORT\_SET\_CLR

#else // !USE\_FAST\_PINIO

digitalWrite(swspi.\_mosi, HIGH);

#endif // end !USE\_FAST\_PINIO

}

/\*!

@brief Set the software (bitbang) SPI MOSI line LOW.

\*/

inline void Adafruit\_SPITFT::SPI\_MOSI\_LOW(void) {

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

#if defined(KINETISK)

\*swspi.mosiPortClr = 1;

#else // !KINETISK

\*swspi.mosiPortClr = swspi.mosiPinMask;

#endif

#else // !HAS\_PORT\_SET\_CLR

\*swspi.mosiPort &= swspi.mosiPinMaskClr;

#endif // end !HAS\_PORT\_SET\_CLR

#else // !USE\_FAST\_PINIO

digitalWrite(swspi.\_mosi, LOW);

#endif // end !USE\_FAST\_PINIO

}

/\*!

@brief Set the software (bitbang) SPI SCK line HIGH.

\*/

inline void Adafruit\_SPITFT::SPI\_SCK\_HIGH(void) {

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

#if defined(KINETISK)

\*swspi.sckPortSet = 1;

#else // !KINETISK

\*swspi.sckPortSet = swspi.sckPinMask;

#endif

#else // !HAS\_PORT\_SET\_CLR

\*swspi.sckPort |= swspi.sckPinMaskSet;

#endif // end !HAS\_PORT\_SET\_CLR

#else // !USE\_FAST\_PINIO

digitalWrite(swspi.\_sck, HIGH);

#endif // end !USE\_FAST\_PINIO

}

/\*!

@brief Set the software (bitbang) SPI SCK line LOW.

\*/

inline void Adafruit\_SPITFT::SPI\_SCK\_LOW(void) {

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

#if defined(KINETISK)

\*swspi.sckPortClr = 1;

#else // !KINETISK

\*swspi.sckPortClr = swspi.sckPinMask;

#endif

#else // !HAS\_PORT\_SET\_CLR

\*swspi.sckPort &= swspi.sckPinMaskClr;

#endif // end !HAS\_PORT\_SET\_CLR

#else // !USE\_FAST\_PINIO

digitalWrite(swspi.\_sck, LOW);

#endif // end !USE\_FAST\_PINIO

}

/\*!

@brief Read the state of the software (bitbang) SPI MISO line.

@return true if HIGH, false if LOW.

\*/

inline bool Adafruit\_SPITFT::SPI\_MISO\_READ(void) {

#if defined(USE\_FAST\_PINIO)

#if defined(KINETISK)

return \*swspi.misoPort;

#else // !KINETISK

return \*swspi.misoPort & swspi.misoPinMask;

#endif // end !KINETISK

#else // !USE\_FAST\_PINIO

return digitalRead(swspi.\_miso);

#endif // end !USE\_FAST\_PINIO

}

/\*!

@brief Issue a single 16-bit value to the display. Chip-select,

transaction and data/command selection must have been

previously set -- this ONLY issues the word. Despite the name,

this function is used even if display connection is parallel;

name was maintaned for backward compatibility. Naming is also

not consistent with the 8-bit version, spiWrite(). Sorry about

that. Again, staying compatible with outside code.

@param w 16-bit value to write.

\*/

void Adafruit\_SPITFT::SPI\_WRITE16(uint16\_t w) {

if (connection == TFT\_HARD\_SPI) {

#if defined(\_\_AVR\_\_)

AVR\_WRITESPI(w >> 8);

AVR\_WRITESPI(w);

#elif defined(ESP8266) || defined(ESP32)

hwspi.\_spi->write16(w);

#elif defined(ARDUINO\_ARCH\_RP2040)

spi\_inst\_t \*pi\_spi = hwspi.\_spi == &SPI ? spi0 : spi1;

w = \_\_builtin\_bswap16(w);

spi\_write\_blocking(pi\_spi, (uint8\_t \*)&w, 2);

#else

// MSB, LSB because TFTs are generally big-endian

hwspi.\_spi->transfer(w >> 8);

hwspi.\_spi->transfer(w);

#endif

} else if (connection == TFT\_SOFT\_SPI) {

for (uint8\_t bit = 0; bit < 16; bit++) {

if (w & 0x8000)

SPI\_MOSI\_HIGH();

else

SPI\_MOSI\_LOW();

SPI\_SCK\_HIGH();

SPI\_SCK\_LOW();

w <<= 1;

}

} else { // TFT\_PARALLEL

#if defined(\_\_AVR\_\_)

\*tft8.writePort = w >> 8;

TFT\_WR\_STROBE();

\*tft8.writePort = w;

#elif defined(USE\_FAST\_PINIO)

if (!tft8.wide) {

\*tft8.writePort = w >> 8;

TFT\_WR\_STROBE();

\*tft8.writePort = w;

} else {

\*(volatile uint16\_t \*)tft8.writePort = w;

}

#endif

TFT\_WR\_STROBE();

}

}

/\*!

@brief Issue a single 32-bit value to the display. Chip-select,

transaction and data/command selection must have been

previously set -- this ONLY issues the longword. Despite the

name, this function is used even if display connection is

parallel; name was maintaned for backward compatibility. Naming

is also not consistent with the 8-bit version, spiWrite().

Sorry about that. Again, staying compatible with outside code.

@param l 32-bit value to write.

\*/

void Adafruit\_SPITFT::SPI\_WRITE32(uint32\_t l) {

if (connection == TFT\_HARD\_SPI) {

#if defined(\_\_AVR\_\_)

AVR\_WRITESPI(l >> 24);

AVR\_WRITESPI(l >> 16);

AVR\_WRITESPI(l >> 8);

AVR\_WRITESPI(l);

#elif defined(ESP8266) || defined(ESP32)

hwspi.\_spi->write32(l);

#elif defined(ARDUINO\_ARCH\_RP2040)

spi\_inst\_t \*pi\_spi = hwspi.\_spi == &SPI ? spi0 : spi1;

l = \_\_builtin\_bswap32(l);

spi\_write\_blocking(pi\_spi, (uint8\_t \*)&l, 4);

#else

hwspi.\_spi->transfer(l >> 24);

hwspi.\_spi->transfer(l >> 16);

hwspi.\_spi->transfer(l >> 8);

hwspi.\_spi->transfer(l);

#endif

} else if (connection == TFT\_SOFT\_SPI) {

for (uint8\_t bit = 0; bit < 32; bit++) {

if (l & 0x80000000)

SPI\_MOSI\_HIGH();

else

SPI\_MOSI\_LOW();

SPI\_SCK\_HIGH();

SPI\_SCK\_LOW();

l <<= 1;

}

} else { // TFT\_PARALLEL

#if defined(\_\_AVR\_\_)

\*tft8.writePort = l >> 24;

TFT\_WR\_STROBE();

\*tft8.writePort = l >> 16;

TFT\_WR\_STROBE();

\*tft8.writePort = l >> 8;

TFT\_WR\_STROBE();

\*tft8.writePort = l;

#elif defined(USE\_FAST\_PINIO)

if (!tft8.wide) {

\*tft8.writePort = l >> 24;

TFT\_WR\_STROBE();

\*tft8.writePort = l >> 16;

TFT\_WR\_STROBE();

\*tft8.writePort = l >> 8;

TFT\_WR\_STROBE();

\*tft8.writePort = l;

} else {

\*(volatile uint16\_t \*)tft8.writePort = l >> 16;

TFT\_WR\_STROBE();

\*(volatile uint16\_t \*)tft8.writePort = l;

}

#endif

TFT\_WR\_STROBE();

}

}

/\*!

@brief Set the WR line LOW, then HIGH. Used for parallel-connected

interfaces when writing data.

\*/

inline void Adafruit\_SPITFT::TFT\_WR\_STROBE(void) {

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

#if defined(KINETISK)

\*tft8.wrPortClr = 1;

\*tft8.wrPortSet = 1;

#else // !KINETISK

\*tft8.wrPortClr = tft8.wrPinMask;

\*tft8.wrPortSet = tft8.wrPinMask;

#endif // end !KINETISK

#else // !HAS\_PORT\_SET\_CLR

\*tft8.wrPort &= tft8.wrPinMaskClr;

\*tft8.wrPort |= tft8.wrPinMaskSet;

#endif // end !HAS\_PORT\_SET\_CLR

#else // !USE\_FAST\_PINIO

digitalWrite(tft8.\_wr, LOW);

digitalWrite(tft8.\_wr, HIGH);

#endif // end !USE\_FAST\_PINIO

}

/\*!

@brief Set the RD line HIGH. Used for parallel-connected interfaces

when reading data.

\*/

inline void Adafruit\_SPITFT::TFT\_RD\_HIGH(void) {

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

\*tft8.rdPortSet = tft8.rdPinMask;

#else // !HAS\_PORT\_SET\_CLR

\*tft8.rdPort |= tft8.rdPinMaskSet;

#endif // end !HAS\_PORT\_SET\_CLR

#else // !USE\_FAST\_PINIO

digitalWrite(tft8.\_rd, HIGH);

#endif // end !USE\_FAST\_PINIO

}

/\*!

@brief Set the RD line LOW. Used for parallel-connected interfaces

when reading data.

\*/

inline void Adafruit\_SPITFT::TFT\_RD\_LOW(void) {

#if defined(USE\_FAST\_PINIO)

#if defined(HAS\_PORT\_SET\_CLR)

\*tft8.rdPortClr = tft8.rdPinMask;

#else // !HAS\_PORT\_SET\_CLR

\*tft8.rdPort &= tft8.rdPinMaskClr;

#endif // end !HAS\_PORT\_SET\_CLR

#else // !USE\_FAST\_PINIO

digitalWrite(tft8.\_rd, LOW);

#endif // end !USE\_FAST\_PINIO

}

#endif // end \_\_AVR\_ATtiny85\_\_