#include "Adafruit\_SPIDevice.h"

//#define DEBUG\_SERIAL Serial

/\*!

\* @brief Create an SPI device with the given CS pin and settings

\* @param cspin The arduino pin number to use for chip select

\* @param freq The SPI clock frequency to use, defaults to 1MHz

\* @param dataOrder The SPI data order to use for bits within each byte,

\* defaults to SPI\_BITORDER\_MSBFIRST

\* @param dataMode The SPI mode to use, defaults to SPI\_MODE0

\* @param theSPI The SPI bus to use, defaults to &theSPI

\*/

Adafruit\_SPIDevice::Adafruit\_SPIDevice(int8\_t cspin, uint32\_t freq,

BusIOBitOrder dataOrder,

uint8\_t dataMode, SPIClass \*theSPI) {

#ifdef BUSIO\_HAS\_HW\_SPI

\_cs = cspin;

\_sck = \_mosi = \_miso = -1;

\_spi = theSPI;

\_begun = false;

\_spiSetting = new SPISettings(freq, dataOrder, dataMode);

\_freq = freq;

\_dataOrder = dataOrder;

\_dataMode = dataMode;

#else

// unused, but needed to suppress compiler warns

(void)cspin;

(void)freq;

(void)dataOrder;

(void)dataMode;

(void)theSPI;

#endif

}

/\*!

\* @brief Create an SPI device with the given CS pin and settings

\* @param cspin The arduino pin number to use for chip select

\* @param sckpin The arduino pin number to use for SCK

\* @param misopin The arduino pin number to use for MISO, set to -1 if not

\* used

\* @param mosipin The arduino pin number to use for MOSI, set to -1 if not

\* used

\* @param freq The SPI clock frequency to use, defaults to 1MHz

\* @param dataOrder The SPI data order to use for bits within each byte,

\* defaults to SPI\_BITORDER\_MSBFIRST

\* @param dataMode The SPI mode to use, defaults to SPI\_MODE0

\*/

Adafruit\_SPIDevice::Adafruit\_SPIDevice(int8\_t cspin, int8\_t sckpin,

int8\_t misopin, int8\_t mosipin,

uint32\_t freq, BusIOBitOrder dataOrder,

uint8\_t dataMode) {

\_cs = cspin;

\_sck = sckpin;

\_miso = misopin;

\_mosi = mosipin;

#ifdef BUSIO\_USE\_FAST\_PINIO

csPort = (BusIO\_PortReg \*)portOutputRegister(digitalPinToPort(cspin));

csPinMask = digitalPinToBitMask(cspin);

if (mosipin != -1) {

mosiPort = (BusIO\_PortReg \*)portOutputRegister(digitalPinToPort(mosipin));

mosiPinMask = digitalPinToBitMask(mosipin);

}

if (misopin != -1) {

misoPort = (BusIO\_PortReg \*)portInputRegister(digitalPinToPort(misopin));

misoPinMask = digitalPinToBitMask(misopin);

}

clkPort = (BusIO\_PortReg \*)portOutputRegister(digitalPinToPort(sckpin));

clkPinMask = digitalPinToBitMask(sckpin);

#endif

\_freq = freq;

\_dataOrder = dataOrder;

\_dataMode = dataMode;

\_begun = false;

}

/\*!

\* @brief Release memory allocated in constructors

\*/

Adafruit\_SPIDevice::~Adafruit\_SPIDevice() {

if (\_spiSetting)

delete \_spiSetting;

}

/\*!

\* @brief Initializes SPI bus and sets CS pin high

\* @return Always returns true because there's no way to test success of SPI

\* init

\*/

bool Adafruit\_SPIDevice::begin(void) {

if (\_cs != -1) {

pinMode(\_cs, OUTPUT);

digitalWrite(\_cs, HIGH);

}

if (\_spi) { // hardware SPI

#ifdef BUSIO\_HAS\_HW\_SPI

\_spi->begin();

#endif

} else {

pinMode(\_sck, OUTPUT);

if ((\_dataMode == SPI\_MODE0) || (\_dataMode == SPI\_MODE1)) {

// idle low on mode 0 and 1

digitalWrite(\_sck, LOW);

} else {

// idle high on mode 2 or 3

digitalWrite(\_sck, HIGH);

}

if (\_mosi != -1) {

pinMode(\_mosi, OUTPUT);

digitalWrite(\_mosi, HIGH);

}

if (\_miso != -1) {

pinMode(\_miso, INPUT);

}

}

\_begun = true;

return true;

}

/\*!

\* @brief Transfer (send/receive) a buffer over hard/soft SPI, without

\* transaction management

\* @param buffer The buffer to send and receive at the same time

\* @param len The number of bytes to transfer

\*/

void Adafruit\_SPIDevice::transfer(uint8\_t \*buffer, size\_t len) {

//

// HARDWARE SPI

//

if (\_spi) {

#ifdef BUSIO\_HAS\_HW\_SPI

#if defined(SPARK)

\_spi->transfer(buffer, buffer, len, nullptr);

#elif defined(STM32)

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

\_spi->transfer(buffer[i]);

}

#else

\_spi->transfer(buffer, len);

#endif

return;

#endif

}

//

// SOFTWARE SPI

//

uint8\_t startbit;

if (\_dataOrder == SPI\_BITORDER\_LSBFIRST) {

startbit = 0x1;

} else {

startbit = 0x80;

}

bool towrite, lastmosi = !(buffer[0] & startbit);

uint8\_t bitdelay\_us = (1000000 / \_freq) / 2;

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

uint8\_t reply = 0;

uint8\_t send = buffer[i];

/\*

Serial.print("\tSending software SPI byte 0x");

Serial.print(send, HEX);

Serial.print(" -> 0x");

\*/

// Serial.print(send, HEX);

for (uint8\_t b = startbit; b != 0;

b = (\_dataOrder == SPI\_BITORDER\_LSBFIRST) ? b << 1 : b >> 1) {

if (bitdelay\_us) {

delayMicroseconds(bitdelay\_us);

}

if (\_dataMode == SPI\_MODE0 || \_dataMode == SPI\_MODE2) {

towrite = send & b;

if ((\_mosi != -1) && (lastmosi != towrite)) {

#ifdef BUSIO\_USE\_FAST\_PINIO

if (towrite)

\*mosiPort = \*mosiPort | mosiPinMask;

else

\*mosiPort = \*mosiPort & ~mosiPinMask;

#else

digitalWrite(\_mosi, towrite);

#endif

lastmosi = towrite;

}

#ifdef BUSIO\_USE\_FAST\_PINIO

\*clkPort = \*clkPort | clkPinMask; // Clock high

#else

digitalWrite(\_sck, HIGH);

#endif

if (bitdelay\_us) {

delayMicroseconds(bitdelay\_us);

}

if (\_miso != -1) {

#ifdef BUSIO\_USE\_FAST\_PINIO

if (\*misoPort & misoPinMask) {

#else

if (digitalRead(\_miso)) {

#endif

reply |= b;

}

}

#ifdef BUSIO\_USE\_FAST\_PINIO

\*clkPort = \*clkPort & ~clkPinMask; // Clock low

#else

digitalWrite(\_sck, LOW);

#endif

} else { // if (\_dataMode == SPI\_MODE1 || \_dataMode == SPI\_MODE3)

#ifdef BUSIO\_USE\_FAST\_PINIO

\*clkPort = \*clkPort | clkPinMask; // Clock high

#else

digitalWrite(\_sck, HIGH);

#endif

if (bitdelay\_us) {

delayMicroseconds(bitdelay\_us);

}

if (\_mosi != -1) {

#ifdef BUSIO\_USE\_FAST\_PINIO

if (send & b)

\*mosiPort = \*mosiPort | mosiPinMask;

else

\*mosiPort = \*mosiPort & ~mosiPinMask;

#else

digitalWrite(\_mosi, send & b);

#endif

}

#ifdef BUSIO\_USE\_FAST\_PINIO

\*clkPort = \*clkPort & ~clkPinMask; // Clock low

#else

digitalWrite(\_sck, LOW);

#endif

if (\_miso != -1) {

#ifdef BUSIO\_USE\_FAST\_PINIO

if (\*misoPort & misoPinMask) {

#else

if (digitalRead(\_miso)) {

#endif

reply |= b;

}

}

}

if (\_miso != -1) {

buffer[i] = reply;

}

}

}

return;

}

/\*!

\* @brief Transfer (send/receive) one byte over hard/soft SPI, without

\* transaction management

\* @param send The byte to send

\* @return The byte received while transmitting

\*/

uint8\_t Adafruit\_SPIDevice::transfer(uint8\_t send) {

uint8\_t data = send;

transfer(&data, 1);

return data;

}

/\*!

\* @brief Manually begin a transaction (calls beginTransaction if hardware

\* SPI)

\*/

void Adafruit\_SPIDevice::beginTransaction(void) {

if (\_spi) {

#ifdef BUSIO\_HAS\_HW\_SPI

\_spi->beginTransaction(\*\_spiSetting);

#endif

}

}

/\*!

\* @brief Manually end a transaction (calls endTransaction if hardware SPI)

\*/

void Adafruit\_SPIDevice::endTransaction(void) {

if (\_spi) {

#ifdef BUSIO\_HAS\_HW\_SPI

\_spi->endTransaction();

#endif

}

}

/\*!

\* @brief Assert/Deassert the CS pin if it is defined

\* @param value The state the CS is set to

\*/

void Adafruit\_SPIDevice::setChipSelect(int value) {

if (\_cs != -1) {

digitalWrite(\_cs, value);

}

}

/\*!

\* @brief Write a buffer or two to the SPI device, with transaction

\* management.

\* @brief Manually begin a transaction (calls beginTransaction if hardware

\* SPI) with asserting the CS pin

\*/

void Adafruit\_SPIDevice::beginTransactionWithAssertingCS() {

beginTransaction();

setChipSelect(LOW);

}

/\*!

\* @brief Manually end a transaction (calls endTransaction if hardware SPI)

\* with deasserting the CS pin

\*/

void Adafruit\_SPIDevice::endTransactionWithDeassertingCS() {

setChipSelect(HIGH);

endTransaction();

}

/\*!

\* @brief Write a buffer or two to the SPI device, with transaction

\* management.

\* @param buffer Pointer to buffer of data to write

\* @param len Number of bytes from buffer to write

\* @param prefix\_buffer Pointer to optional array of data to write before

\* buffer.

\* @param prefix\_len Number of bytes from prefix buffer to write

\* @return Always returns true because there's no way to test success of SPI

\* writes

\*/

bool Adafruit\_SPIDevice::write(const uint8\_t \*buffer, size\_t len,

const uint8\_t \*prefix\_buffer,

size\_t prefix\_len) {

beginTransactionWithAssertingCS();

// do the writing

#if defined(ARDUINO\_ARCH\_ESP32)

if (\_spi) {

if (prefix\_len > 0) {

\_spi->transferBytes((uint8\_t \*)prefix\_buffer, nullptr, prefix\_len);

}

if (len > 0) {

\_spi->transferBytes((uint8\_t \*)buffer, nullptr, len);

}

} else

#endif

{

for (size\_t i = 0; i < prefix\_len; i++) {

transfer(prefix\_buffer[i]);

}

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

transfer(buffer[i]);

}

}

endTransactionWithDeassertingCS();

#ifdef DEBUG\_SERIAL

DEBUG\_SERIAL.print(F("\tSPIDevice Wrote: "));

if ((prefix\_len != 0) && (prefix\_buffer != nullptr)) {

for (uint16\_t i = 0; i < prefix\_len; i++) {

DEBUG\_SERIAL.print(F("0x"));

DEBUG\_SERIAL.print(prefix\_buffer[i], HEX);

DEBUG\_SERIAL.print(F(", "));

}

}

for (uint16\_t i = 0; i < len; i++) {

DEBUG\_SERIAL.print(F("0x"));

DEBUG\_SERIAL.print(buffer[i], HEX);

DEBUG\_SERIAL.print(F(", "));

if (i % 32 == 31) {

DEBUG\_SERIAL.println();

}

}

DEBUG\_SERIAL.println();

#endif

return true;

}

/\*!

\* @brief Read from SPI into a buffer from the SPI device, with transaction

\* management.

\* @param buffer Pointer to buffer of data to read into

\* @param len Number of bytes from buffer to read.

\* @param sendvalue The 8-bits of data to write when doing the data read,

\* defaults to 0xFF

\* @return Always returns true because there's no way to test success of SPI

\* writes

\*/

bool Adafruit\_SPIDevice::read(uint8\_t \*buffer, size\_t len, uint8\_t sendvalue) {

memset(buffer, sendvalue, len); // clear out existing buffer

beginTransactionWithAssertingCS();

transfer(buffer, len);

endTransactionWithDeassertingCS();

#ifdef DEBUG\_SERIAL

DEBUG\_SERIAL.print(F("\tSPIDevice Read: "));

for (uint16\_t i = 0; i < len; i++) {

DEBUG\_SERIAL.print(F("0x"));

DEBUG\_SERIAL.print(buffer[i], HEX);

DEBUG\_SERIAL.print(F(", "));

if (len % 32 == 31) {

DEBUG\_SERIAL.println();

}

}

DEBUG\_SERIAL.println();

#endif

return true;

}

/\*!

\* @brief Write some data, then read some data from SPI into another buffer,

\* with transaction management. The buffers can point to same/overlapping

\* locations. This does not transmit-receive at the same time!

\* @param write\_buffer Pointer to buffer of data to write from

\* @param write\_len Number of bytes from buffer to write.

\* @param read\_buffer Pointer to buffer of data to read into.

\* @param read\_len Number of bytes from buffer to read.

\* @param sendvalue The 8-bits of data to write when doing the data read,

\* defaults to 0xFF

\* @return Always returns true because there's no way to test success of SPI

\* writes

\*/

bool Adafruit\_SPIDevice::write\_then\_read(const uint8\_t \*write\_buffer,

size\_t write\_len, uint8\_t \*read\_buffer,

size\_t read\_len, uint8\_t sendvalue) {

beginTransactionWithAssertingCS();

// do the writing

#if defined(ARDUINO\_ARCH\_ESP32)

if (\_spi) {

if (write\_len > 0) {

\_spi->transferBytes((uint8\_t \*)write\_buffer, nullptr, write\_len);

}

} else

#endif

{

for (size\_t i = 0; i < write\_len; i++) {

transfer(write\_buffer[i]);

}

}

#ifdef DEBUG\_SERIAL

DEBUG\_SERIAL.print(F("\tSPIDevice Wrote: "));

for (uint16\_t i = 0; i < write\_len; i++) {

DEBUG\_SERIAL.print(F("0x"));

DEBUG\_SERIAL.print(write\_buffer[i], HEX);

DEBUG\_SERIAL.print(F(", "));

if (write\_len % 32 == 31) {

DEBUG\_SERIAL.println();

}

}

DEBUG\_SERIAL.println();

#endif

// do the reading

for (size\_t i = 0; i < read\_len; i++) {

read\_buffer[i] = transfer(sendvalue);

}

#ifdef DEBUG\_SERIAL

DEBUG\_SERIAL.print(F("\tSPIDevice Read: "));

for (uint16\_t i = 0; i < read\_len; i++) {

DEBUG\_SERIAL.print(F("0x"));

DEBUG\_SERIAL.print(read\_buffer[i], HEX);

DEBUG\_SERIAL.print(F(", "));

if (read\_len % 32 == 31) {

DEBUG\_SERIAL.println();

}

}

DEBUG\_SERIAL.println();

#endif

endTransactionWithDeassertingCS();

return true;

}

/\*!

\* @brief Write some data and read some data at the same time from SPI

\* into the same buffer, with transaction management. This is basicaly a wrapper

\* for transfer() with CS-pin and transaction management. This /does/

\* transmit-receive at the same time!

\* @param buffer Pointer to buffer of data to write/read to/from

\* @param len Number of bytes from buffer to write/read.

\* @return Always returns true because there's no way to test success of SPI

\* writes

\*/

bool Adafruit\_SPIDevice::write\_and\_read(uint8\_t \*buffer, size\_t len) {

beginTransactionWithAssertingCS();

transfer(buffer, len);

endTransactionWithDeassertingCS();

return true;

}