```
// Arduino "bridge" code between host computer and WS2801-based digital
// RGB LED pixels (e.g. Adafruit product ID #322).  Intended for use
// with USB-native boards such as Teensy or Adafruit 32u4 Breakout;
// works on normal serial Arduinos, but throughput is severely limited.
// LED data is streamed, not buffered, making this suitable for larger
// installations (e.g. video wall, etc.) than could otherwise be held
// in the Arduino's limited RAM.

// Some effort is put into avoiding buffer underruns (where the output
// side becomes starved of data).  The WS2801 latch protocol, being
// delay-based, could be inadvertently triggered if the USB bus or CPU
// is swamped with other tasks.  This code buffers incoming serial data
// and introduces intentional pauses if there's a threat of the buffer
// draining prematurely.  The cost of this complexity is somewhat
// reduced throughput, the gain is that most visual glitches are
// avoided (though ultimately a function of the load on the USB bus and
// host CPU, and out of our control).

// LED data and clock lines are connected to the Arduino's SPI output.
// On traditional Arduino boards, SPI data out is digital pin 11 and
// clock is digital pin 13.  On both Teensy and the 32u4 Breakout,
// data out is pin B2, clock is B1.  LEDs should be externally
// powered -- trying to run any more than just a few off the Arduino's
// 5V line is generally a Bad Idea.  LED ground should also be
// connected to Arduino ground.

// --------------------------------------------------------------------
//   This file is part of Adalight.
```

```
//   Adalight is free software: you can redistribute it and/or modify
//   it under the terms of the GNU Lesser General Public License as
//   published by the Free Software Foundation, either version 3 of
//   the License, or (at your option) any later version.

//   Adalight is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU Lesser General Public License for more details.

//   You should have received a copy of the GNU Lesser General Public
//   License along with Adalight.  If not, see
//   <http://www.gnu.org/licenses/>.
// --------------------------------------------------------------------

#include <SPI.h>

// LED pin for Adafruit 32u4 Breakout Board:
//#define LED_DDR  DDRE
//#define LED_PORT PORTE
//#define LED_PIN  _BV(PORTE6)
// LED pin for Teensy:
//#define LED_DDR  DDRD
//#define LED_PORT PORTD
//#define LED_PIN  _BV(PORTD6)
// LED pin for Arduino:
#define LED_DDR  DDRB
```

```
#define LED_PORT PORTB
#define LED_PIN  _BV(PORTB5)

// A 'magic word' (along with LED count & checksum) precedes each block
// of LED data; this assists the microcontroller in syncing up with the
// host-side software and properly issuing the latch (host I/O is
// likely buffered, making usleep() unreliable for latch).  You may see
// an initial glitchy frame or two until the two come into alignment.
// The magic word can be whatever sequence you like, but each character
// should be unique, and frequent pixel values like 0 and 255 are
// avoided -- fewer false positives.  The host software will need to
// generate a compatible header: immediately following the magic word
// are three bytes: a 16-bit count of the number of LEDs (high byte
// first) followed by a simple checksum value (high byte XOR low byte
// XOR 0x55).  LED data follows, 3 bytes per LED, in order R, G, B,
// where 0 = off and 255 = max brightness.

static const uint8_t magic[] = {'A','d','a'};
#define MAGICSIZE  sizeof(magic)
#define HEADERSIZE (MAGICSIZE + 3)

#define MODE_HEADER 0
#define MODE_HOLD   1
#define MODE_DATA   2

// If no serial data is received for a while, the LEDs are shut off
// automatically.  This avoids the annoying "stuck pixel" look when
// quitting LED display programs on the host computer.
```

```
static const unsigned long serialTimeout = 15000; // 15 seconds

void setup()
{
  // Dirty trick: the circular buffer for serial data is 256 bytes,
  // and the "in" and "out" indices are unsigned 8-bit types -- this
  // much simplifies the cases where in/out need to "wrap around" the
  // beginning/end of the buffer.  Otherwise there'd be a ton of bit-
  // masking and/or conditional code every time one of these indices
  // needs to change, slowing things down tremendously.
  uint8_t
    buffer[256],
    indexIn       = 0,
    indexOut      = 0,
    mode          = MODE_HEADER,
    hi, lo, chk, i, spiFlag;
  int16_t
    bytesBuffered = 0,
    hold          = 0,
    c;
  int32_t
    bytesRemaining;
  unsigned long
    startTime,
    lastByteTime,
    lastAckTime,
    t;
```

```
LED_DDR  |=  LED_PIN; // Enable output for LED

LED_PORT &= ~LED_PIN; // LED off


Serial.begin(115200); // Teensy/32u4 disregards baud rate; is OK!


SPI.begin();

SPI.setBitOrder(MSBFIRST);

SPI.setDataMode(SPI_MODE0);

SPI.setClockDivider(SPI_CLOCK_DIV128); // 1 MHz max, else flicker


// Issue test pattern to LEDs on startup.  This helps verify that

// wiring between the Arduino and LEDs is correct.  Not knowing the

// actual number of LEDs connected, this sets all of them (well, up

// to the first 25,000, so as not to be TOO time consuming) to red,

// green, blue, then off.  Once you're confident everything is working

// end-to-end, it's OK to comment this out and reprogram the Arduino.

uint8_t testcolor[] = { 0, 0, 0, 255, 0, 0 };

for(char n=3; n>=0; n--) {

  for(c=0; c<25000; c++) {

    for(i=0; i<3; i++) {

      for(SPDR = testcolor[n + i]; !(SPSR & _BV(SPIF)); );

    }

  }

  delay(1); // One millisecond pause = latch

}


Serial.print("Ada\n"); // Send ACK string to host
```

```
startTime    = micros();
lastByteTime = lastAckTime = millis();

// loop() is avoided as even that small bit of function overhead
// has a measurable impact on this code's overall throughput.

for(;;) {

  // Implementation is a simple finite-state machine.
  // Regardless of mode, check for serial input each time:
  t = millis();
  if((bytesBuffered < 256) && ((c = Serial.read()) >= 0)) {
    buffer[indexIn++] = c;
    bytesBuffered++;
    lastByteTime = lastAckTime = t; // Reset timeout counters
  } else {
    // No data received.  If this persists, send an ACK packet
    // to host once every second to alert it to our presence.
    if((t - lastAckTime) > 1000) {
      Serial.print("Ada\n"); // Send ACK string to host
      lastAckTime = t; // Reset counter
    }
    // If no data received for an extended time, turn off all LEDs.
    if((t - lastByteTime) > serialTimeout) {
      for(c=0; c<32767; c++) {
        for(SPDR=0; !(SPSR & _BV(SPIF)); );
      }
      delay(1); // One millisecond pause = latch
```

```
    lastByteTime = t; // Reset counter
  }
}

switch(mode) {

 case MODE_HEADER:

  // In header-seeking mode.  Is there enough data to check?
  if(bytesBuffered >= HEADERSIZE) {
    // Indeed.  Check for a 'magic word' match.
    for(i=0; (i<MAGICSIZE) && (buffer[indexOut++] == magic[i++]););
    if(i == MAGICSIZE) {
      // Magic word matches.  Now how about the checksum?
      hi  = buffer[indexOut++];
      lo  = buffer[indexOut++];
      chk = buffer[indexOut++];
      if(chk == (hi ^ lo ^ 0x55)) {
        // Checksum looks valid.  Get 16-bit LED count, add 1
        // (# LEDs is always > 0) and multiply by 3 for R,G,B.
        bytesRemaining = 3L * (256L * (long)hi + (long)lo + 1L);
        bytesBuffered -= 3;
        spiFlag       = 0;        // No data out yet
        mode          = MODE_HOLD; // Proceed to latch wait mode
      } else {
        // Checksum didn't match; search resumes after magic word.
        indexOut  -= 3; // Rewind
      }
```

```
  } // else no header match.  Resume at first mismatched byte.
  bytesBuffered -= i;
 }
 break;

case MODE_HOLD:

 // Ostensibly "waiting for the latch from the prior frame
 // to complete" mode, but may also revert to this mode when
 // underrun prevention necessitates a delay.

 if((micros() - startTime) < hold) break; // Still holding; keep buffering

 // Latch/delay complete.  Advance to data-issuing mode...
 LED_PORT &= ~LED_PIN;  // LED off
 mode      = MODE_DATA; // ...and fall through (no break):

case MODE_DATA:

 while(spiFlag && !(SPSR & _BV(SPIF))); // Wait for prior byte
 if(bytesRemaining > 0) {
  if(bytesBuffered > 0) {
    SPDR = buffer[indexOut++];   // Issue next byte
    bytesBuffered--;
    bytesRemaining--;
    spiFlag = 1;
  }
  // If serial buffer is threatening to underrun, start
```

```
      // introducing progressively longer pauses to allow more
      // data to arrive (up to a point).
      if((bytesBuffered < 32) && (bytesRemaining > bytesBuffered)) {
        startTime = micros();
        hold      = 100 + (32 - bytesBuffered) * 10;
        mode      = MODE_HOLD;
          }
     } else {
      // End of data -- issue latch:
      startTime  = micros();
      hold       = 1000;       // Latch duration = 1000 uS
      LED_PORT  |= LED_PIN;     // LED on
      mode       = MODE_HEADER; // Begin next header search
     }
   } // end switch
  } // end for(;;)
}

void loop()
{
  // Not used.  See note in setup() function.
}
```