

Getting Started with the AndyMark LED/Arduino Kit (AM-2645)

This kit includes the 8.2ft RGB LED light strip (AM-2640)
and Arduino Ethernet Controller (AM-2287)

T. Chou
FRC Team 3940 "Cybertooth"
Northwestern High School
Kokomo, IN
3-24-2014 Rev1.2

CONTENTS

Getting Started with Arduino & RGB LED strips	4
Wiring Notes and Diagram	4
The Arduino IDE (Integrated Development Environment)	5
FastLED libraries	5
Installing your FastLED library:	5
Identifying your Arduino board type	6
Communicating with your Arduino board	7
Powering your Arduino Board	8
Powering the AM-2640 LED Strip	8
The Arduino Ethernet Input and Output Pins	9
The 14 Digital Pins	9
The 6 Analog Pins	10
Using the Arduino text (sketch) editor:	10
Changing default font size in Arduino editor:	11
To verify & upload your code	11
Arduino Language Reference	11
Using FastLED libraries	12
Setting up the leds	12
Include FastLED library, define number of LEDs, Data & Clock pins	12
Define memory block for LED data (LED Array)	13
Setup for correct Chipset	13
FastLED library commands	14
The main loop in Arduino code	14
Writing to an led	14
Changing an led's color	15
'Moving' a lit LED – the traveling dot	15

controlling leds with external controls.....	16
How to set an LED's color	16
Set RGB Color	16
Set HSV Color.....	18
Read RGB Data From Serial	19
Color Math.....	20
Adding and Subtracting Colors	20
Dimming and Brightening Colors.....	20
Constraining Colors Within Limits	22
Misc Color Functions	22
Example Arduino Sketches for LED strips.....	23
LightChaser Example	23
Digital Input Control Example.....	25
AndyMark LightChaser Example 2.....	27

GETTING STARTED WITH ARDUINO & RGB LED STRIPS

Note that these RGB (Red Green Blue) LEDs require a controller such as an Arduino board, they will not work just by supplying power and ground. Each LED is individually addressable and configurable in terms of color and intensity via a SPI (Serial Peripheral Interface) Data line and SPI Clk (Clock) so each LED has an IC next to it to decode the SPI bus.

If you're trying to get the LED strip working in the simplest way possible (the KISS principle) that'll be defined here but if you want to get fancy and start controlling the Arduino with the robot through Ethernet or controlling LED colors tied to some robot action (non-KISS methods) that'll be defined as well.

WIRING NOTES AND DIAGRAM

Remember on the AM-2640 LED strip's yellow wire is ground! (Don't blame us they come that way)

Yellow wire is ground!



- Red wire is +5V in
- Green wire is SPI Data (Goes to Arduino Pin 11 for the KISS method)
- Blue wire is SPI Clock (Goes to Arduino Pin 13 for the KISS method)



- Power Converter
- **Yellow is +5V out!**
 - Red wire is +10V-30V in
 - Black wires are ground



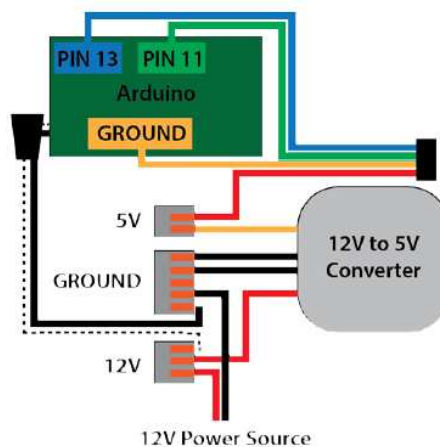
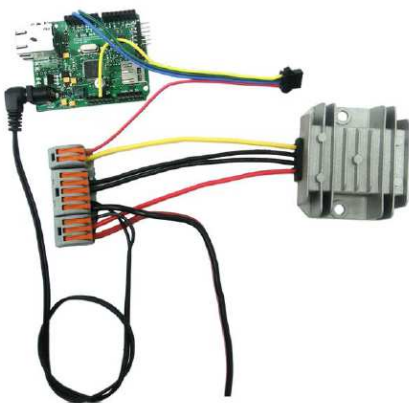
DC Power Cable

- Coaxial Shield is ground
- Center Wire is +V



- Arduino Ethernet
- Pin 13 SPI Clock for the KISS method
 - Pin 11 SPI Data for the KISS method

Note the wiring diagrams below are for the KISS method.



THE ARDUINO IDE (INTEGRATED DEVELOPMENT ENVIRONMENT)

For Arduino info, visit this official Getting Started website: http://arduino.cc/en/Guide/HomePage#Uxe8I_IdV8E

- Download the Arduino IDE (Integrated Development Environment) Software and install it and the necessary drivers onto your PC for your OS (Operating System – Windows, Apple or Linux) from the website above following their instructions.
- Read about the Arduino Development Environment in the website above.
- Read about Arduino Libraries in the website above.

FASTLED LIBRARIES

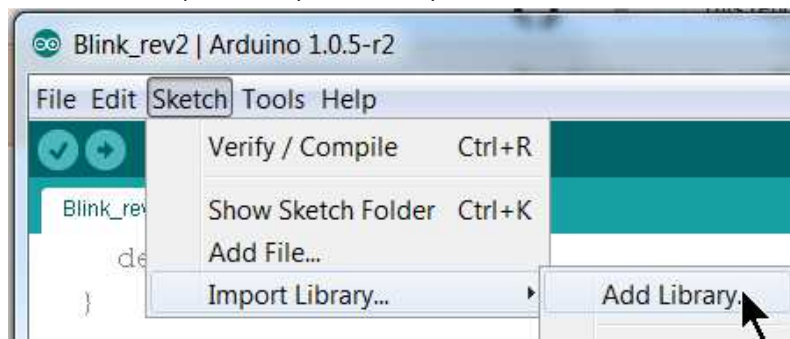
These LED strips require installing the FastLED libraries – a collection of pre-written code written for all the popular LED chipsets that your Arduino program will call, making programming your Arduino easier for you. These are available from <http://fastled.io/blog/> which will lead you to their GitHub site:

<https://github.com/FastLED/FastLED/releases> where you can download the library .zip file (FastLED.zip).

Note that the FastLED library was formerly known as the “FastSPI_LED2” libraries available at <https://code.google.com/p/fastspi/> and <https://code.google.com/p/fastspi/wiki/CRGBreference> and as of March 2014 these sites are still maintained for reference only. Please refer now to <http://fastled.io/blog/> and <https://github.com/FastLED/FastLED/releases> for the latest info.

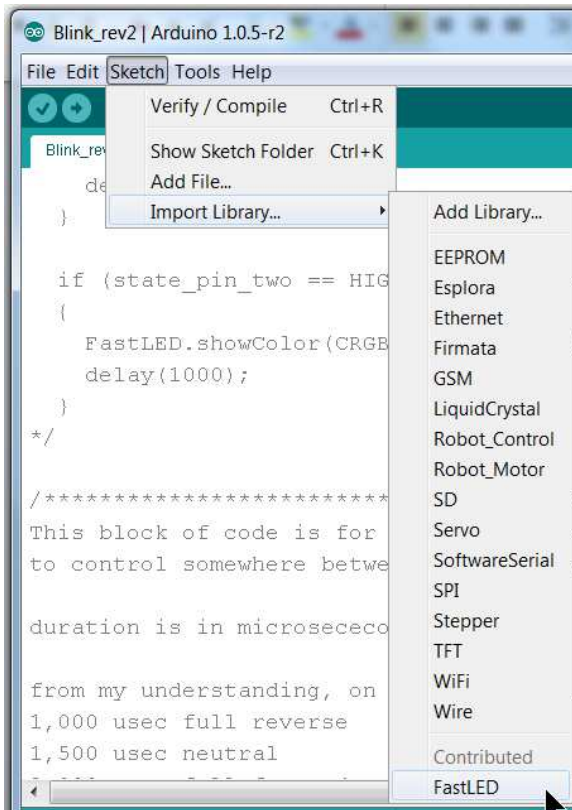
INSTALLING YOUR FASTLED LIBRARY:

- Download FastLED.zip
- Go to Sketch: Import Library: Add Library

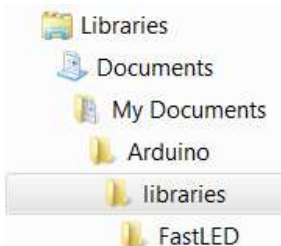


- Find your FastLED.zip file and select it.

- If it successfully installs, you'll see it listed under the Contributed heading.



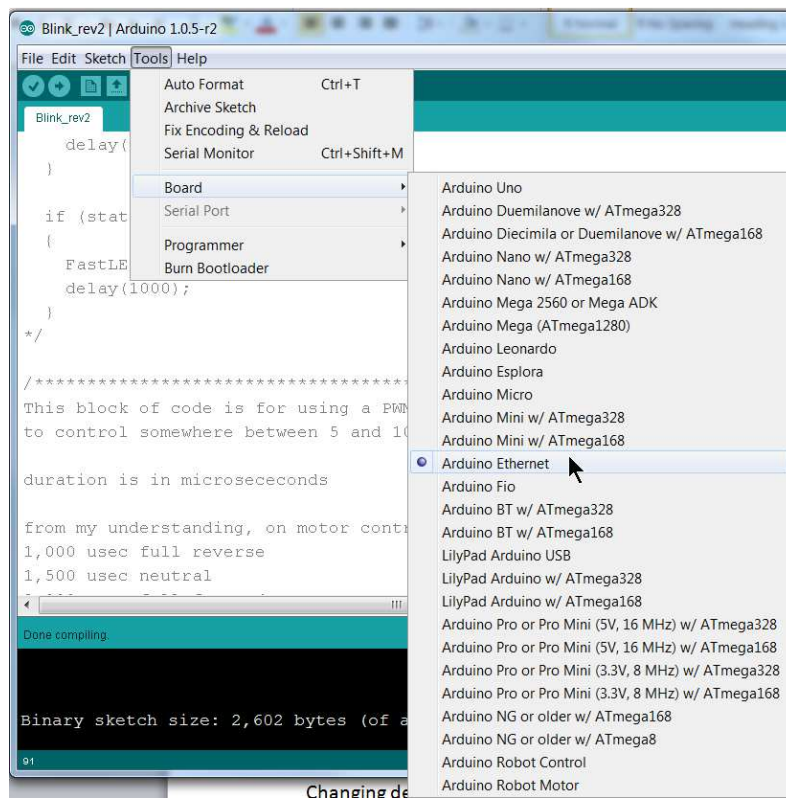
- The actual location of your FastLED library will be in



- This location under My Documents allows you to keep these FastLED libraries installed in case you update your Arduino Environment software as the default libraries (located in C:\Program Files (x86)\Arduino\libraries) will get replaced every time you re-install the Arduino Environment.
- Note that the WS2801 is the chipset in the AM-2640 LED strip.

IDENTIFYING YOUR ARDUINO BOARD TYPE

After installing the FastLED libraries you will need to choose the type of Arduino board you have. The AndyMark am-2287 is an "Arduino Ethernet" compatible (which is essentially a surface mount Arduino Uno / Arduino Pro hybrid with an Ethernet Shield). Shields are boards that plug on top of the Arduino PC board extending its capabilities. Choose your board type as shown below:



COMMUNICATING WITH YOUR ARDUINO BOARD

KISS method: Use the am-2415 FTDI USB-to-serial cable shown here and plug it in as shown (black wire facing the Digital Pins side):



Additional info for non- KISS method:

The Arduino Ethernet differs from other Arduino boards in that it does not have a FTDI (Future Technology Devices International) USB-to-serial chip on board. The FTDI chip is used to convert RS-232 TTL (Transistor Transistor Logic) serial transmissions to USB (Universal Serial Bus) signals. Since the Arduino's native communication is TTL serial, it requires a FTDI chip to connect to USB devices. The Arduino Ethernet accomplishes this via a FTDI USB-to-serial

cable (which contains the FTDI chip inside it) as shown above (am-2416) which plugs into the 6 pin serial programming header.

Notice that on the Arduino Ethernet board, Digital Pins 0 and 1 are shared pins which are used to receive (RX) and transmit (TX) TTL serial data and also used for other digital input functions. Since these USB communications are actually converted to TTL serial transmissions, Digital Pins 0 and 1 are used for communications (i.e. programming), so be careful if you're planning on using these pins for other purposes as you could step on (interfere with) communications (e.g., you'll get programming failures). This is described in more detail in the 14 Digital Pins section below.

Note that this 6 pin serial programming header is compatible with other FTDI USB cables and features support for automatic reset, allowing sketches to be uploaded without pressing the reset button on the board.

Additionally, when plugged into a FTDI-style USB adapter, the Arduino Ethernet can be powered off the adapter.

POWERING YOUR ARDUINO BOARD

KISS method: Power your Arduino board from 12v with the supplied DC Barrel Jack power cord. Get 12v from your robot's power distribution board.

More detailed info for those not using the KISS method:

There are essentially two power nodes on the Arduino, Vin and 5v. Vin is connected to the DC Barrel Jack and to the on-board 5V regulator's input. The 5v node is connected to the 5v Pin, the output of the regulator, and USB.

Therefore, when plugged into a USB-to-Serial adapter, the Arduino Ethernet is capable of being powered from this adapter. The Arduino has an auto-switch circuit to switch between 5v or Vin if the voltage on the Vin node is $>7v$, hence the recommendation of 7v-12v for the input plug voltage. So if you provide between 7v-12v on the DC Barrel Jack Input plug, the Arduino will automatically use that as the power source.

A third option for powering this Arduino Ethernet is via an optional Power over Ethernet (PoE) module.

POWERING THE AM-2640 LED STRIP

KISS method: Get 5v from your robot's power distribution board or use the AndyMark CPR-360 (am-0899) 12-24v to 5v power converter to power the LED strip.

Warning: The WS2801 chipset is extremely sensitive to voltage spikes. This strip runs off of 5V MAX! Applying more than 5V will damage/destroy you LED strip! Do NOT make / change connections while the circuit is powered and generally follow good engineering practices when handling this product taking precautions against electrostatic discharge (ESD). AndyMark is currently investigating protection methods, it may be advisable to connect a 100 μF capacitor between the ground and power lines on the power input. If the strip does get damaged, it may just be the first LED that is blown so try cutting this first segment off and resoldering the connector to the second segment.

Additional info for non- KISS method:

The AM-2640 LED strip is a ShenZhen Shiji Lighting Co WS2801 chipset based strip which requires 5v. DO NOT try to power the whole strip off the Arduino 5v regulator as it cannot support the 1.5 A current draw at full bright white, so you still need to run a separate 5v power supply for the LED strip!

Note that because the chipset operates off a SPI (Serial Peripheral Interface) bus, the 5v reference is critical for its messaging so the GND reference must not float. Therefore, the GND reference between the Arduino power supply and LED strip power supply must be the same. If you observe problems with erratic LED behavior, it may be necessary to have a common ground between the Arduino power supply and the LED strip power (i.e., connect a wire between a GND pin on the Arduino with the GND on the LED strip.)

For example, powering the Arduino off a laptop USB and the LED strip off a grounded 5v supply plugged into your grounded 120v AC household may work just fine as long as the laptop is also plugged into a grounded 120v AC power supply (grounds are common due to the 120v AC grounded outlet), but once the laptop is unplugged and running off its battery, the LEDs will start behaving erratically as the Arduino GND and LED GND references are no longer common. Connect a wire between a GND pin on the Arduino with the GND on the LED strip or make sure both the 12v and 5v power sources share a common ground through the 120v outlet (i.e., make sure both use grounded 120v outlets and plugs).

If you're following the KISS method then both your 12v and 5v supply should be coming from the same source (your robot battery) so they'll already share a common ground and you won't have any grounding issues.

THE ARDUINO ETHERNET INPUT AND OUTPUT PINS

See <http://arduino.cc/en/Main/ArduinoBoardEthernet#.Uyia3ldV8E> for more information.

THE 14 DIGITAL PINS

Additional info for non- KISS method:

Each of the 14 digital pins on the Ethernet board can be used as an input or output, using [pinMode\(\)](#), [digitalWrite\(\)](#), and [digitalRead\(\)](#) functions. They operate at 5 volts. Each pin can provide or receive a maximum of 40 mA and has an internal pull-up resistor (disconnected by default) of 20-50 kOhms. In addition, some pins have specialized functions:

- Serial: 0 (RX) and 1 (TX). Used to receive (RX) and transmit (TX) TTL serial data.

Warning! Digital pins 0 and 1 are used for communicating with the Arduino board (e.g., when programming) so it's possible that you could interfere with this if you have these pins configured for your uses (e.g., pulled high or low) so be aware of this and you might want to avoid using Pins 0 and 1 if at all possible

If you see some error like this when you're uploading your sketch (programming) this could be your problem: "avrdude: stk500_getsync(): not in sync: resp=0x00"

- External Interrupts: 2 and 3. These pins can be configured to trigger an interrupt on a low value, a rising or falling edge, or a change in value. See the [attachInterrupt\(\)](#) function for details.

- PWM: 3, 5, 6, 9, and 10. Provide 8-bit PWM output with the [analogWrite\(\)](#) function.

These are the pins that provide a “digitally created” hardware PWM output. PWM is essentially an analog signal, but in this case it is created by digital means, hence it’s output on these digital pins. Although any of the digital pins can effectively be programmed to output a PWM signal, any pins other than 3,5,6,9 & 10 would require extensive processing power while these would not.

- SPI: 10 (SS Slave Select), 11 (MOSI MasterOutSlaveIn), 12 (MISO MasterInSlaveOut), 13 (SCK Serial Clock). These pins support SPI communication using the [SPI library](#).

Note that these same pins 10-13 on this board are also reserved for Ethernet communications, so if you’re using the Ethernet port (e.g. , you’re wanting to control the LEDs via Ethernet by the robot) you cannot use Port 11 & 13 for the SPI data and clock as described in the examples in this document. The Arduino is the Master the LED strip is the Slave, so by default Port 11 (MOSI) is the LED strip data pin.

Luckily, the FastLED libraries can define other ports for SPI data & clock so you’ll have to do this if you want to use Ethernet, but be aware that you must setup for the chipset differently if you’re defining your own SPI data & clock pins. This will be discussed in detail below in the [Using FastLED Libraries / Setting Up the LEDs](#) section.

- LED: 9. There is a built-in LED connected to digital pin 9. When the pin is HIGH value, the LED is on, when the pin is LOW, it's off. On most other arduino boards, this LED is found on pin 13. It is on pin 9 on the Ethernet board because pin 13 is used as part of the SPI connection.

THE 6 ANALOG PINS

Additional info for non- KISS method:

The Ethernet board has 6 analog inputs, labeled A0 through A5, each of which provide 10 bits of resolution (i.e., 1024 different values). By default they measure from ground to 5 volts, though is it possible to change the upper end of their range using the AREF pin and the [analogReference\(\)](#) function. Additionally, some pins have specialized functionality:

- TWI: A4 (SDA) and A5 (SCL). Support TWI communication using the [Wire library](#).

There are a couple of other pins on the board:

- AREF. Reference voltage for the analog inputs. Used with [analogReference\(\)](#).
- Reset. Bring this line LOW to reset the microcontroller. Typically used to add a reset button to shields which block the one on the board.

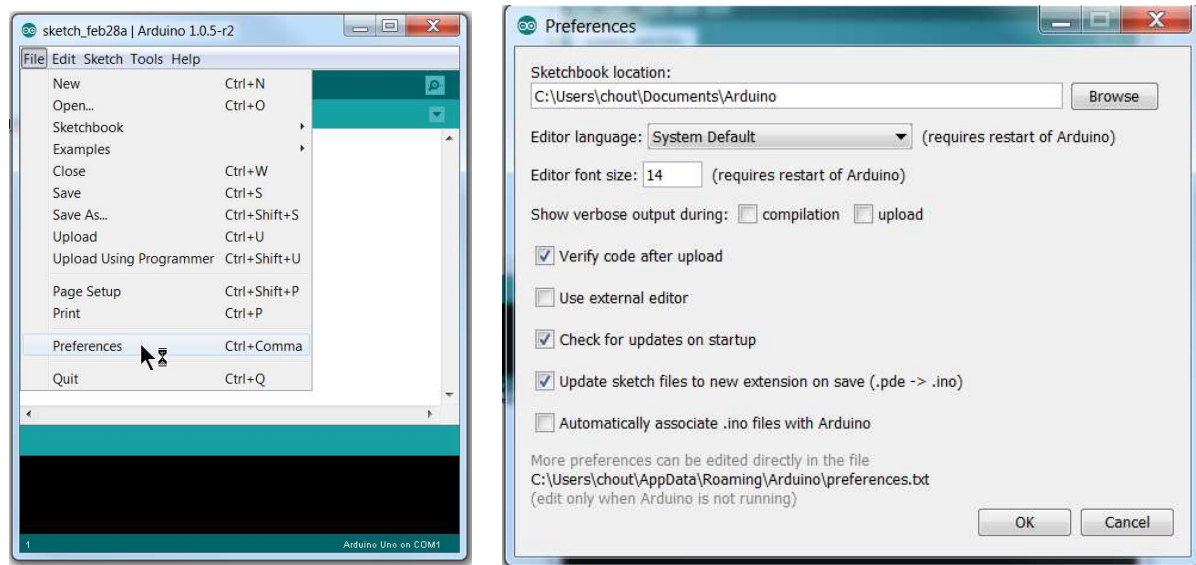
See also the [mapping between Arduino pins and ATmega328 ports](#).

USING THE ARDUINO TEXT (SKETCH) EDITOR:

Software written using Arduino are called sketches. These sketches are written in the text editor of the Arduino IDE and are saved with the file extension .ino.

CHANGING DEFAULT FONT SIZE IN ARDUINO EDITOR:

For us mentors who need a larger font size:



TO VERIFY & UPLOAD YOUR CODE



Use the check mark icon to verify your sketch (code) for errors without uploading, and use the right arrow icon to verify and upload your code to the Arduino.

ARDUINO LANGUAGE REFERENCE

The Arduino language is based on C/C++. For details concerning the Arduino programming language refer to this webpage:

<http://arduino.cc/en/Reference/HomePage#.UyihbPIdV8F>

It links against [AVR Libc](#) and allows the use of any of its functions; see its [user manual](#) for details. See the [libraries page](#) for interfacing with particular types of hardware. Try the list of [community-contributed code](#) for code examples.

USING FASTLED LIBRARIES

These examples use the FastLED library, FastLED.h, formerly known as FastSPI_LED2.h

With LEDs you have basically 3 color spaces you can choose to use:

RGB = Red Green Blue

HSL = Hue Saturation Lightness

HSV = Hue Saturation Value (aka HSB for Brightness or HSI for Intensity)

KISS method: Stick with RGB.

The HSV color space is a bit easier to navigate and provide transitions between colors compared to using RGB. When defining colors with RGB you're mixing the Red, Green, and Blue color values. When using HSV you're defining the hue of the color (that is, where it is on the color wheel), how saturated it is, and how bright it is.

Refer to this website for info on using the FastLED libraries: <https://github.com/FastLED/FastLED/wiki/Basic-usage>

SETTING UP THE LEDS

INCLUDE FASTLED LIBRARY, DEFINE NUMBER OF LEDS, DATA & CLOCK PINS

KISS method: Beginning lines of our .ino file:

```
#include < FastLED.h>
#define NUM_LEDS 80
```

You need to call the FastLED library and define the number of LEDs in the strip. (The strip is originally an 80 LED strip. If you cut it shorter, change NUM_LEDS.)

Additional info for non- KISS method:

The default SPI data and clock pins are 11 and 13 as described in the 14 Digital Pins section above, so there is no need to define these. If you decide to use the Ethernet and need to move these to other digital pins, you'll need to define them, so include these statements as your next lines of code (replace pin 3 & 4 with whatever you choose):

```
// Data pin that led data will be written out over
#define DATA_PIN 3      //Green wire from AM-2640's power connector

// Clock pin SPI
#define CLOCK_PIN 4      //Blue wire from AM-2640's power connector
```

Note that FastLED.h was previously known as FastSPI_LED2.h so you'll still see some Arduino LED sketches refer to this older library, but currently that library is just a shell with a redirect to FastLED.h with a warning that "This file is going away, please use FastLED.h in the future!"

For LED chips like Neopixels, which have a data line, ground, and power, you just need to define DATA_PIN. For LED chipsets that are SPI based (four wires - data, clock, ground, and power), like the LPD8806 or WS2801 you may need to define both DATA_PIN and CLOCK_PIN if you're not using the default ones.

DEFINE MEMORY BLOCK FOR LED DATA (LED ARRAY)

Next, we need to set up the block of memory that will be used for storing and manipulating the led data (i.e., define the LED array):

```
CRGB leds[NUM_LEDS];
```

This sets up an array that we can manipulate to set/clear LED data. For more details, see <https://github.com/FastLED/FastLED/wiki/Pixel-reference>

Note that a “CRGB” is an object defined in the FastLED library that represents a color in RGB color space. Typically, when using this library, each LED strip is represented as an array of CRGB colors, one color for each LED pixel.

SETUP FOR CORRECT CHIPSET

Now, let's actually setup our LEDs, which is a single line of code in our setup function:

KISS method:

```
void setup() {  
    //This is the chipset in the AM-2640 LED strip using the default Data/CLK pins  
    FastLED.addLeds<WS2801, RGB>(leds, NUM_LEDS);  
}
```

This tells the library the chipset to use, and those LEDs will use the LED array `leds`, and there are `NUM_LEDS` (i.e., 80) of them. Note that the AndyMark AM-2640 is a WS2801 chipset and we are not defining the Data and Clock pins since we're using the default ones (Pins 11 & 13).

Additional info for non- KISS method:

If you cannot use the default SPI Data & CLK pins and have defined your own, you'll need to use this setup instead:

```
void setup() {  
    //This is the chipset in the AM-2640 LED strip using your defined Data/CLK pins  
    FastLED.addLeds<WS2801, DATA_PIN, CLOCK_PIN, RGB>(leds, NUM_LEDS);  
}
```

OTHER COMMON LED STRIP CHIPSETS SUPPORTED BY THE FASTLED LIBRARY

Additional info for non- KISS method:

If you are not using the WS2801 chipset here are other chipsets supported by the FastLED library:

```
// FastLED.addLeds<TM1803, DATA_PIN, RGB>(leds, NUM_LEDS);  
// FastLED.addLeds<TM1804, DATA_PIN, RGB>(leds, NUM_LEDS);  
// FastLED.addLeds<TM1809, DATA_PIN, RGB>(leds, NUM_LEDS);
```

```
// FastLED.addLeds<WS2811, DATA_PIN, RGB>(leds, NUM_LEDS);
// FastLED.addLeds<WS2812, DATA_PIN, RGB>(leds, NUM_LEDS);
// FastLED.addLeds<WS2812B, DATA_PIN, RGB>(leds, NUM_LEDS);
// FastLED.addLeds<UCS1903, DATA_PIN, RGB>(leds, NUM_LEDS);
// FastLED.addLeds<SM16716, RGB>(leds, NUM_LEDS);
// FastLED.addLeds<LPD8806, RGB>(leds, NUM_LEDS);
// FastLED.addLeds<SM16716, DATA_PIN, CLOCK_PIN, RGB>(leds, NUM_LEDS);
// FastLED.addLeds<LPD8806, DATA_PIN, CLOCK_PIN, RGB>(leds, NUM_LEDS);
// FastLED.addLeds<NEOPIXEL, DATA_PIN>(leds, NUM_LEDS);
```

FASTLED LIBRARY COMMANDS

The following are FastLED library commands you'll use:

```
FastLED.clear(); //This turns off all LEDs
```

```
FastLED.setBrightness(led_brightness); //This sets the brightness for all LEDs
//led_brightness represents a number between 0-255
//where 0 is dark and full brightness is 255
//so 50% brightness would be 128
```

```
FastLED.show(); //This sends the setting/commands you've previously set to the LEDs
//so if you previously set an LED to be red, this turns it on
```

```
//The showColor method sets all the LEDs in the strip to the same color
FastLED.showColor(CRGB(127, 0, 0)); 127=50% brightness red
FastLED.showColor(CRGB(255, 64, 191)); //255=100% red brightness, 64=25% green, 191=75% blue
FastLED.showColor(CRGB(5, 0, 5)); //2% red brightness + 0% green + 2% blue = 2% purple (violet)
FastLED.showColor(CRGB::Magenta); //use standard named web/HTML color code
```

THE MAIN LOOP IN ARDUINO CODE

The main loop in Arduino code is simply referred to as `loop()`, as in:

```
void loop() {
    // Put main loop function calls and code here. The loop repeats endlessly.
}
```

WRITING TO AN LED

Making your LEDs actually show colors is a two part process with this library. First, you set the values of the entries in the `leds` array to whatever colors you want. Then you tell the library to show your data. Your animation/code/patterns will pretty much consist of this cycle. Decide what you want everything to display, set it, then tell the LED strip to display it. Let's do something very simple, and set the first LED to red:

```
void loop() {
    leds[0] = CRGB::Red;
```

```
FastLED.show();  
delay(30);  
}
```

Upload this sketch to your arduino and it will set the first LED to red, over and over again so it'll stay a solid red.

Next, let's make it blink.

CHANGING AN LED'S COLOR

You can change the value that you set to a led between calls to show, and the next time you call show the new value will get written out. So, we can set the value of that first LED to Red, let it sit there for a second, then set it back to black, let it sit there for a second, and just keep it looping like that, over and over.

```
void loop() {  
  
    // Turn the first led red for 1 second  
    leds[0] = CRGB::Red;  
    FastLED.show();  
    delay(1000);  
  
    // Set the first led back to black for 1 second  
    leds[0] = CRGB::Black;  
    FastLED.show();  
    delay(1000);  
}
```

'MOVING' A LIT LED – THE TRAVELING DOT

Now a single LED is blinking. Next we'll move an LED dot down the length of the strip, a traveling dot. We want to set the first LED to Blue, show the LEDs, set that LED back to Black, delay a little bit, then start this over with the second LED, and so on down the line, until we've shown all the LEDs:

```
void loop() {  
  
    for(int dot = 0; dot < NUM_LEDS; dot++) {  
        leds[dot] = CRGB::Blue;  
        FastLED.show();  
        // clear this led for the next time around the loop  
        leds[dot] = CRGB::Black;  
        delay(30);  
  
        // Notice that there is no need to use the "FastLED.show();" to change the LED color this 2nd time  
        // This is because the next time through the loop the previous LED (leds[dot-1] if you will) has  
        // been set to black and the current Led (leds[dot]) is set to Blue. The FastLED.show command  
        // "activates" both the new led color command and the previous led Black command. For clarity you may  
        // want to place another FastLED.show() command after each color set command.
```

```
}  
}
```

CONTROLLING LEDS WITH EXTERNAL CONTROLS

If we have a [potentiometer](#) connected to the arduino on analog pin 2 that ranges in value from 0-1023, we can use that potentiometer value to decide how many LEDs to have light. We can use the arduino [map](#) function to go from 0-1023 to 0-NUM_LEDS. Here's what that loop function might look like:

```
void loop() {  
  int val = analogRead(2);  
  int numLedsToLight = map(val, 0, 1023, 0, NUM_LEDS);  
  
  // First, clear the existing led values  
  FastLED.clear();  
  
  for(int led = 0; led < numLedsToLight; led++) {  
    leds[led] = CRGB::Blue;  
  }  
  FastLED.show();  
}
```

Now you have something that will change the number of LEDs that are on based on what your potentiometer is set to.

HOW TO SET AN LED'S COLOR

There are lots of ways to set an LED's color; this page gives very short examples of several of them. See <https://github.com/FastLED/FastLED/wiki/Pixel-reference> for more information.

SET RGB COLOR

KISS method: Look at Examples 1, 2, 3 and 4

Additional info for non- KISS method:

Note that a "CRGB" is an object defined in the FastLED library that represents a color in RGB color space. It contains

- a one byte value (0-255) representing the amount of red,
- a one byte value (0-255) representing the amount of green,
- a one byte value (0-255) representing the amount of blue in a given color.

The FastLED library has declared CRGB to be an enumerated variable of sorts in that the named HTML web colors are already mapped to their corresponding byte values, allowing you to assign LED colors by simply linking it to a color name (Example 3 below).

Example 1: set individual R, G, and B fields, the classic way:

```
leds[i].r = 255; //255 = full brightness red
leds[i].g = 68;  //68 = 26.67% brightness green
leds[i].b = 221; //221 = 87% brightness blue
```

Each of these following examples does exactly the same thing:

```
// The three color channel values can be referred to as "red", "green", and "blue"...
leds[i].red  = 255;
leds[i].green = 68;
leds[i].blue  = 221;
```

```
// ...or, using the shorter synonyms "r", "g", and "b"...
leds[i].r = 255;
leds[i].g = 68;
leds[i].b = 221;
```

```
// ...or as members of a three-element array:
leds[i][0] = 255; // red
leds[i][1] = 68;  // green
leds[i][2] = 221; // blue
```

- Example 2: set color from red, green, and blue components all at once

```
leds[i] = CRGB( 255, 68, 221);
```

- Example 3: set RGB from a standard named web/HTML color code

(See http://en.wikipedia.org/wiki/Web_colors or http://en.wikipedia.org/wiki/X11_color_names)

```
leds[i] = CRGB::HotPink;
```

- Example 4: set RGB using 'setRGB' and three values at once

```
leds[i].setRGB( 255, 64, 191); //255=100% red brightness, 64=25% green, 191=75% blue
leds[i].setRGB( 5, 0, 5); //2% red brightness + 0% green + 2% blue = 2% purple (violet)
```

- Example 5: set RGB from a single (hex) color code (hex triplet)

```
leds[i] = 0xFF44DD; //FF44DD = RGB(255,68,221)
```

- Example 6: copy RGB color from another led

```
leds[i] = leds[j];
```

If you are copying a large number of colors from one (part of an) array to another, the standard library function `memcpy` can be used to perform a bulk transfer; the `CRGB` object "is trivially copyable".

```
// Copy ten led colors from leds[src .. src+9] to leds[dest .. dest+9]
memcpy( &leds[dest], &leds[src], 10 * sizeof( CRGB ) );
```

- Example 7: use new 'fill_solid', telling it to fill just one led. (V2) Note that this is a pretty silly way to set one pixel, but it lets us illustrate the existence of `fill_solid`, a new convenience function the library provides.

```
fill_solid( &(leds[i]), 1 /*number of leds*/, CRGB( 255, 68, 221) )
```

SET HSV COLOR

KISS method: Ignore HSV color

Six ways to set an LED's color from HSV (Hue, Saturation, Value). In general, they mostly involve assigning a `CHSV` color to a `CRGB` color; the colorspace conversion happens through an automatic call to `hsv2rgb_rainbow`. It's worth noting that a 'spectrum' and a 'rainbow' are different things; rainbows are more visually color-balanced, and have more yellows and oranges than spectra do. You probably want to start with 'rainbow', and only switch to 'spectrum' if you have a specific need.

- Example 1: Using a 'rainbow' color with hue 0-255, saturating 0-255, and brightness (value) 0-255 (V2)

```
// Simplest, preferred way: assignment from a CHSV color
leds[i] = CHSV( 224, 187, 255);

// Alternate syntax
leds[i].setHSV( 224, 187, 255);
```

- Example 2: Setting to a pure, bright, fully-saturated rainbow hue

```
leds[i].setHue( 224);
```

- Example 3: Using a 'spectrum' color with hue 0-255

```
CHSV spectrumcolor;
spectrumcolor.hue =      222;
spectrumcolor.saturation = 187;
spectrumcolor.value =    255;
hsv2rgb_spectrum( spectrumcolor, leds[i] );
```

- Example 4: Using a 'spectrum' color with hue 0-191

```
CHSV spectrumcolor192;
spectrumcolor192.hue =    166;
spectrumcolor192.saturation = 187;
spectrumcolor192.value =  255;
hsv2rgb_raw( spectrumcolor192, leds[i] ); //raw
```

- Example 5: use new 'fill_solid', telling it to fill just one led. (V2) Note that this is a pretty silly way to set one pixel, but it lets us illustrate the existence of fill_solid, a new convenience function the library provides.

```
fill_solid( &(leds[i]), 1 /*number of leds*/, CHSV( 224, 187, 255) );
```

- Example 6: With fill_rainbow. (V2) Note that this is a pretty silly way to set one pixel, but it lets us illustrate the existence of fill_rainbow, a new convenience function the library provides.

```
fill_rainbow( &(leds[i]), 1 /*led count*/, 222 /*starting hue*/);
```

READ RGB DATA FROM SERIAL

A "CRGB" is nothing more than a convenient wrapper for three bytes of raw RGB data: one byte of red, one byte of green, and one byte of blue. You are welcome, and invited, to directly access the underlying memory. These examples show how you read binary RGB data directly from a Serial stream into your array of LED values:

- Read a single pixel of RGB data directly into one CRGB (V2).

```
Serial.readBytes( (char*)&leds[i], 3); // read three bytes: r, g, and b.
```

- Read a whole strip's worth of RGB data directly into the leds array at once (V2)

```
Serial.readBytes( (char*)leds, NUM_LEDS * 3);
```

COLOR MATH

<https://code.google.com/p/fastspi/wiki/CRGBreference>

KISS Method: Ignore color math

If you wanted to add a little bit of red to an existing LED color, you could do this:

```
// Here's all that's needed to add "a little red" to an existing LED color:  
leds[i] += CRGB( 20, 0, 0);
```

If you've ever done this sort of thing by hand before, you may notice something missing: the check for the red channel overflowing past 255. Traditionally, you've probably had to do something like this:

```
// Add a little red, the old way.  
uint16_t newRed;  
newRed = leds[i].r + 20;  
if( newRed > 255) newRed = 255; // prevent wrap-around  
leds[i].r = newRed;
```

This kind of add-and-then-check-and-then-adjust-if-needed logic is taken care of for you inside the library code for adding two CRGB colors, inside operator+ and operator+=. All of the math operations defined on the CRGB colors are automatically protected from wrap-around, overflow, and underflow.

ADDING AND SUBTRACTING COLORS

```
// Add one CRGB color to another.  
leds[i] += CRGB( 20, 0, 0);
```

```
// Add a constant amount of brightness to all three (RGB) channels.  
leds[i] += 20;
```

```
// Add a constant "1" to the brightness of all three (RGB) channels.  
leds[i]++;
```

```
// Subtract one color from another.  
leds[i] -= CRGB( 20, 0, 0);
```

```
// Subtract a constant amount of brightness from all three (RGB) channels.  
leds[i] -= 20;
```

```
// Subtract a constant "1" from the brightness of all three (RGB) channels.  
leds[i]--;
```

DIMMING AND BRIGHTENING COLORS

There are two different methods for dimming a color: "video" style and "raw math" style. Video style is the default, and is explicitly designed to never accidentally dim any of the RGB channels down from a lit LED (no matter how dim) to an UNlit LED -- because that often comes out looking wrong at low brightness levels. The "raw math" style will eventually fade to black.

Colors are always dimmed down by a fraction. The dimming fraction is expressed in 256ths, so if you wanted to dim a color down by 25% of its current brightness, you first have to express that in 256ths. In this case, $25\% = 64/256$.

```
// Dim a color by 25% (64/256ths)
// using "video" scaling, meaning: never fading to full black
leds[i].fadeLightBy( 64 );
```

You can also express this the other way: that you want to dim the pixel to 75% of its current brightness. $75\% = 192/256$. There are two ways to write this, both of which will do the same thing. The first uses the `%=` operator; the rationale here is that you're setting the new color to "a percentage" of its previous value:

```
// Reduce color to 75% (192/256ths) of its previous value
// using "video" scaling, meaning: never fading to full black
leds[i] %= 192;
```

The other way is to call the underlying scaling function directly. Note the "video" suffix.

```
// Reduce color to 75% (192/256ths) of its previous value
// using "video" scaling, meaning: never fading to full black
leds[i].nscale8_video( 192);
```

If you want the color to eventually fade all the way to black, use one of these functions:

```
// Dim a color by 25% (64/256ths)
// eventually fading to full black
leds[i].fadeToBlackBy( 64 );
```

```
// Reduce color to 75% (192/256ths) of its previous value
// eventually fading to full black
leds[i].nscale8( 192);
```

A function is also provided to boost a given color to maximum brightness while keeping the same hue:

```
// Adjust brightness to maximum possible while keeping the same hue.
leds[i].maximizeBrightness();
```

Finally, colors can also be scaled up or down using multiplication and division.

```
// Divide each channel by a single value
leds[i] /= 2;

// Multiply each channel by a single value
leds[i] *= 2;
```

CONSTRAINING COLORS WITHIN LIMITS

The library provides a function that lets you 'clamp' each of the RGB channels to be within given minimums and maximums. You can force all of the color channels to be at least a given value, or at most a given value. These can then be combined to limit both minimum and maximum.

```
// Bring each channel up to at least a minimum value. If any channel's
// value is lower than the given minimum for that channel, it is
// raised to the given minimum. The minimum can be specified separately
// for each channel (as a CRGB), or as a single value.
leds[i] |= CRGB( 32, 48, 64);
leds[i] |= 96;

// Clamp each channel down to a maximum value. If any channel's
// value is higher than the given maximum for that channel, it is
// reduced to the given maximum. The minimum can be specified separately
// for each channel (as a CRGB), or as a single value.
leds[i] &= CRGB( 192, 128, 192);
leds[i] &= 160;
```

MISC COLOR FUNCTIONS

The library provides a function that 'inverts' each RGB channel. Performing this operation twice results in the same color you started with.

```
// Invert each channel
leds[i] = -leds[i];
```

The library also provides functions for looking up the apparent (or mathematical) brightness of a color.

```
// Get brightness, or luma (brightness, adjusted for eye's sensitivity to
// different light colors. See http://en.wikipedia.org/wiki/Luma\_\(video\) )
uint8_t luma = leds[i].getLuma();
uint8_t avgLight = leds[i].getAverageLight();
```

EXAMPLE ARDUINO SKETCHES FOR LED STRIPS

LIGHTCHASER EXAMPLE

```
//This LightChaser example moves a colored LED down a strip with each color moving faster

#include <FastLED.h>    //Must include the FastLED library!
#define NUM_LEDS 80     //How many LEDs in your strip?

CRGB leds[NUM_LEDS];    //Define LED array (i.e. memory block for LED data)

void setup() {
    //This is the chipset in the AM-2640 LED strip for KISS method
    FastLED.addLeds<WS2801, RGB>(leds, NUM_LEDS);
}

void loop() {    //Main Loop, loops endlessly
    //Call the function color_chase, passing through the value "Green" and "100" for 100 msec, etc.
    color_chase(CRGB::Green, 100);
    color_chase(CRGB::BlueViolet, 80);
    color_chase(CRGB::Red, 60);
    color_chase(CRGB::White, 40);
    color_chase(CRGB::Yellow, 30);
    color_chase(CRGB::Cyan, 20);
    color_chase(CRGB::Blue, 10);
    color_chase(CRGB::Fuchsia, 5);
    //Note: go to http://en.wikipedia.org/wiki/Web\_colors for all valid color names
    //This main loop will keep looping
}

void color_chase(uint32_t color, uint8_t wait)
    //This declares a function named color_chase
    //color is the first variable name being passed a value defined as an unsigned integer of 32 bits (uint32_t)
    //wait is the second variable name being passed a value defined as an unsigned integer of 8 bits (uint8_t)
{
    FastLED.clear();           //Clear all LEDs
    FastLED.setBrightness(255); //Set global LED brightness to full

    // Move a single led
    for(int led_number = 0; led_number < NUM_LEDS; led_number++)

    {

        // Turn our current led ON, then show the leds
        leds[led_number] = color;

        // Show the leds (only one of which is set to white, from above)
        FastLED.show();

        // Wait a little bit
        delay(wait);

        // Turn our current LED back to black for the next loop around
        leds[led_number] = CRGB::Black;

        // Notice that there is no need to use the "FastLED.show();" to change the LED color this 2nd time
        // This is because the next time through the loop the previous LED (leds[dot-1] if you will) has
        // been set to black and the current Led (leds[dot]) is set to Blue. The FastLED.show command
        // "activates" both the new led color command and the previous led Black command. For clarity you may
        // want to place another FastLED.show() command after each color set command.
    }
}
```

Missing Dot Chase Example

//This Missing Dot Chase example moves a unlit LED down a strip of colors as the intensity gets dimmer
//with each progressive color change and the speed gets faster.

```
#include <FastLED.h>    //Must include the FastLED library!
#define NUM_LEDS 80     //How many LEDs in your strip?

CRGB leds[NUM_LEDS];    //Define LED array (i.e. memory block for LED data)

void setup() {
    //This is the chipset in the AM-2640 LED strip for KISS method
    FastLED.addLeds<WS2801, RGB>(leds, NUM_LEDS);
}

void loop() {    //Main Loop, loops endlessly

    //Call the function missing_dot_chase, passing through the value "White" and "100" for 100 msec, etc.
    missing_dot_chase(CRGB::White, 100);
    missing_dot_chase(CRGB::Red, 80);
    missing_dot_chase(CRGB::Yellow, 60);
    missing_dot_chase(CRGB::Green, 40);
    missing_dot_chase(CRGB::Cyan, 30);
    missing_dot_chase(CRGB::Blue, 20);
    missing_dot_chase(0x3000cc, 10) ; //This is another way of dictating the color purple via hex value
}

//Move an "empty" dot down the strip
//This declares a function named missing_dot_chase
//color is the first variable name being passed a value defined as an unsigned integer of 32 bits (uint32_t)
//wait is the second variable name being passed a value defined as an unsigned integer of 8 bits (uint8_t)
void missing_dot_chase(uint32_t color, uint8_t wait)
{
    int led_number;

    //Loop to start LED brightness @ 100/256 intensity, with each loop brightness is decreased in half
    //as long as led_brightness is > 10
    for (int led_brightness = 100; led_brightness > 10; led_brightness/=2)
    {
        FastLED.setBrightness(led_brightness);

        // Start by turning all LEDs on to whatever color is set to:
        for(led_number = 0; led_number < NUM_LEDS; led_number++) leds[led_number] = color;

        // Then run a dark LED down the line turning off/on an LED incrementally one at a time:
        for(led_number = 0; led_number < NUM_LEDS; led_number++)

        {
            leds[led_number] = CRGB::Black; // Set new LED 'off'

            if( led_number > 0 && led_number < NUM_LEDS)
            {
                leds[led_number-1] = color; // Set previous LED 'on'
            }
            FastLED.show();
            delay(wait);
        }
    }
}
```


DIGITAL INPUT CONTROL EXAMPLE

```
//This sketch lights up the LED strip in colors corresponding to 2 digital inputs

#include <FastLED.h> //Must include the FastLED library!
#define NUM_LEDS 2 //How many LEDs in your strip?

//Note that I mistakenly used Digital Pins 0 & 1 so I run the risk of interference when programming.
//Simply use 2 other Digital Pins instead – I was too lazy to rewrite this example code.
//Or make sure Pins 0 & 1 are floating when you program.

//Note there are 2 ways to define the Digital Pins
//const int PIN_ZERO = 0; //Arduino Digital Input 0 assigned to variable PIN_ZERO
//const int PIN_ONE = 1; //Arduino Digital Input 1 assigned to variable PIN_ONE
//The above is not the preferred way (uses more memory or something)
//The preferred method is below using #define
//Note the lack of the = sign and the lack of the ; at the end
#define PIN_ZERO 0
#define PIN_ONE 1

//The IDE somehow automatically knows that these are digital pins.
//If we had meant to assign analog pins instead would we designate then with a “a” prefix
//#define PIN_ZERO a0

CRGB leds[NUM_LEDS]; //Define LED array (i.e. memory block for LED data)

void setup() {
    //This is the chipset in the AM-2640 LED strip for KISS method
    FastLED.addLeds<WS2801, RGB>(leds, NUM_LEDS);

    // initialize the input pins for digital input method
    pinMode(PIN_ZERO, INPUT); //Note that Arduino pins default to inputs so this is not necessary
    pinMode(PIN_ONE, INPUT); //Note that Arduino pins default to inputs so this is not necessary

    FastLED.clear(); //Clear all LEDs
    FastLED.setBrightness(255); //Set global LED brightness to full (could be omitted in this example as brightness is overridden later)
}

//Initialize variables
int state_pin_zero;
int state_pin_one;

void loop() { //Main Loop, loops endlessly
    /*****
    This block of code is for using pins 0 and 1 on the Arduino board as digital inputs to control up to 4 states of LED programs.
    *****/

    state_pin_zero = digitalRead(PIN_ZERO);
    state_pin_one = digitalRead(PIN_ONE);

    if (state_pin_one == LOW && state_pin_zero == LOW)
    {
        FastLED.showColor(CRGB(127, 0, 0)); //if pin1/pin0=low/low LED=red
        delay(1000);
    }

    if (state_pin_one == LOW && state_pin_zero == HIGH)
    {
        FastLED.showColor(CRGB(0, 127, 0)); //if pin1/pin0=low/high LED=green
        delay(1000);
    }

    if (state_pin_one == HIGH && state_pin_zero == LOW)
    {

```

```
FastLED.showColor(CRGB(0, 0, 127));    //if pin1/pin0=high/low LED=blue
delay(1000);
}

if (state_pin_one == HIGH && state_pin_zero == HIGH)
{
  FastLED.showColor(CRGB(127, 0, 127));  //if pin1/pin0=high/high LED=purple
  delay(1000);
}
}
```

ANDYMARK LIGHTCHASER EXAMPLE 2

```
//Some light chase hacks for the AM-2640 5V, Addressable LED strips http://www.andymark.com/product-p/am-2640.htm based on the
WS2801 chipset
//We ran this demo off of our AM-2287 Arduino Ethernet http://www.andymark.com/product-p/am-2287.htm
//http://arduino.cc/en/Main/ArduinoBoardEthernet
//The FastLED library we use here supports multiple chipsets
//This code requires that the fastspi library be put in your arduino\libraries folder
//Arduino info on how to install software libraries http://arduino.cc/en/Guide/Libraries
//AndyMark, Inc.
//Craig Kessler 12/3/2013, 3/17/2014

/**NOTE: This strip runs off of 5V MAX!!!. Applying more than 5V will damage/destroy you LED strip!*/
//DO NOT try to power the whole strip (80 LEDs) off the arduino 5v regulator.
//At full bright white, the strip can draw 1.5Amps or so. This will overheat or burnout the regulator.
//Remember on the AM-2640 LED strip's yellow wire is ground! (don't blame us they come that way)
//Make sure you connect the Yellow ground from the LED strip to the Arduino ground.
//Communications to the LEDs requires a common ground to work.

//We recommend running these led strips off of the AM-0899 10-30Vin to 5V 5A out stepdown converter http://www.andymark.com/product-p/am-0899.htm
//If you are using the AndyMark AM-2297 Arduino Ethernet board then make sure you select Tools>Board>Arduino Ethernet from the Arduino
IDE menu
//If you are new to working with Arduino a good place to start is here http://arduino.cc/en/Guide/HomePage
//Another new training resource provided by a 3rd party is here: http://www.arduino4classroom.com/index.php/arduino-101

//CSK 3/17/2013 Libraries new location
//https://github.com/FastLED/FastLED
//https://github.com/FastLED/FastLED/wiki/Overview

#include "FastSPI_LED2.h"
//CSK 3/17/2014 FastSPI library has been updated. The new header file name is just FastLED.h. FastSPI_LED2.h is now just a shell with an
include for FastLED.h

#include "FastLED.h"

//Tell it how many leds are in the strip. AndyMark's 2.5 meter strip has 80 leds
#define NUM_LEDS 80

// This is an array of leds. One item for each led in your strip.
CRGB leds[NUM_LEDS];

//CSK 3/17/2014 I moved these to pins that don't conflict with Ethernet functions in case you want to control LEDs via Ethernet
#define DATA_PIN 3 //Green wire from AM-2640's power connector
// Clock pin SPI
#define CLOCK_PIN 5 //Blue wire from AM-2640's power connector
#define MAX_BRIGHTNESS 255

//This function is used to setup things like pins, Serial ports etc.
//Here we specify which chipset our LEDs run off of by our choice of config function
void setup()

{
    // Uncomment one of the following lines for your leds arrangement.
    // FastLED.addLeds<TM1803, DATA_PIN, RGB>(leds, NUM_LEDS);
    // FastLED.addLeds<TM1804, DATA_PIN, RGB>(leds, NUM_LEDS);
    // FastLED.addLeds<TM1809, DATA_PIN, RGB>(leds, NUM_LEDS);
    //FastLED.addLeds<WS2811, DATA_PIN, RGB>(leds, NUM_LEDS);
    // FastLED.addLeds<WS2812, DATA_PIN, RGB>(leds, NUM_LEDS);
    // FastLED.addLeds<WS2812B, DATA_PIN, RGB>(leds, NUM_LEDS);
    // FastLED.addLeds<UCS1903, DATA_PIN, RGB>(leds, NUM_LEDS);
    //FastLED.addLeds<WS2801, RGB>(leds, NUM_LEDS);
    // FastLED.addLeds<SM16716, RGB>(leds, NUM_LEDS)
    // FastLED.addLeds<LPD8806, RGB>(leds, NUM_LEDS);
}
```

```

    /***This is the chipset in the AM-2640 LED strip***/
    FastLED.addLeds<WS2801, DATA_PIN, CLOCK_PIN, RGB>(leds, NUM_LEDS);

    // FastLED.addLeds<SM16716, DATA_PIN, CLOCK_PIN, RGB>(leds, NUM_LEDS);
    // FastLED.addLeds<LPD8806, DATA_PIN, CLOCK_PIN, RGB>(leds, NUM_LEDS);
}

void loop()

{
    //This is kind of Arduino's equivalent to Main() in a standard C program
    //This, as the name implies, loops endlessly.
    //https://code.google.com/p/fastspi/wiki/CRGBreference
    cylon(CRGB::Red, 25, 5);
    color_chase(CRGB::Green, 15);
    color_chase(CRGB::BlueViolet, 15);
    color_chase(CRGB::Red, 15);
    color_chase(CRGB::Yellow, 15);
    color_chase(CRGB::Green, 15);
    color_chase(CRGB::Cyan, 15);
    color_chase(CRGB::Blue, 15);
    missing_dot_chase(CRGB::White, 20);
    missing_dot_chase(CRGB::Red, 20);
    missing_dot_chase(CRGB::Yellow, 20);
    missing_dot_chase(CRGB::Green, 20);
    missing_dot_chase(CRGB::Cyan, 20);
    missing_dot_chase(CRGB::Blue, 20);
    missing_dot_chase(0x3000cc, 20);
}

//These are the functions we have defined to do chase patterns. They are actually called inside the loop() above
//They are meant to demonstrate things such as setting LED colors, controlling brightness
void color_chase(uint32_t color, uint8_t wait)
{
    //clear() turns all LEDs off
    FastLED.clear();
    //The brightness ranges from 0-255
    //Sets brightness for all LEDs at once
    FastLED.setBrightness(MAX_BRIGHTNESS);

    // Move a single led
    for(int led_number = 0; led_number < NUM_LEDS; led_number++)
    {
        // Turn our current led ON, then show the leds
        leds[led_number] = color;

        // Show the leds (only one of which has a color set, from above)
        // Show turns actually turns on the LEDs
        FastLED.show();
        // Wait a little bit
        delay(wait);
        // Turn our current led back to black for the next loop around
        leds[led_number] = CRGB::Black;
    }
}

//Move an "empty" dot down the strip
void missing_dot_chase(uint32_t color, uint8_t wait)
{
    //Step thru some brightness levels from max to 10. led_brightness/=2 is a cryptic shorthand way of saying led_brightness =
    led_brightness/2

    for (int led_brightness = MAX_BRIGHTNESS; led_brightness > 10; led_brightness/=2)

```

```

{
    FastLED.setBrightness(led_brightness);

    // Start by turning all pixels on:
    //for(int led_number = 0; led_number < NUM_LEDS; led_number++) leds[led_number] = color;
    //https://github.com/FastLED/FastLED/wiki/Controlling-leds

    fill_solid(leds, NUM_LEDS, color);

    // Then display one pixel at a time:
    for(int led_number = 0; led_number < NUM_LEDS; led_number++)
    {
        leds[led_number] = CRGB::Black; // Set new pixel 'off'
        if( led_number > 0 && led_number < NUM_LEDS)
        {
            leds[led_number-1] = color; // Set previous pixel 'on'
        }
        FastLED.show();
        delay(wait);
    }
}

//Cylon - LED sweeps back and forth, with the color, delay and number of cycles of your choice
void cylon(CRGB color, uint16_t wait, uint8_t number_of_cycles)
{
    FastLED.setBrightness(255);
    for (uint8_t times = 0; times<=number_of_cycles; times++)
    {
        // Make it look like one LED is moving in one direction
        for(int led_number = 0; led_number < NUM_LEDS; led_number++)
        {
            //Apply the color that was passed into the function
            leds[led_number] = color;
            //Actually turn on the LED we just set
            FastLED.show();
            // Turn it back off
            leds[led_number] = CRGB::Black;
            // Pause before "going" to next LED
            delay(wait);
        }
        // Now "move" the LED the other direction
        for(int led_number = NUM_LEDS-1; led_number >= 0; led_number--)
        {
            //Apply the color that was passed into the function
            leds[led_number] = color;
            //Actually turn on the LED we just set
            FastLED.show();
            // Turn it back off
            leds[led_number] = CRGB::Black;
            // Pause before "going" to next LED
            delay(wait);
        }
    }
}

```