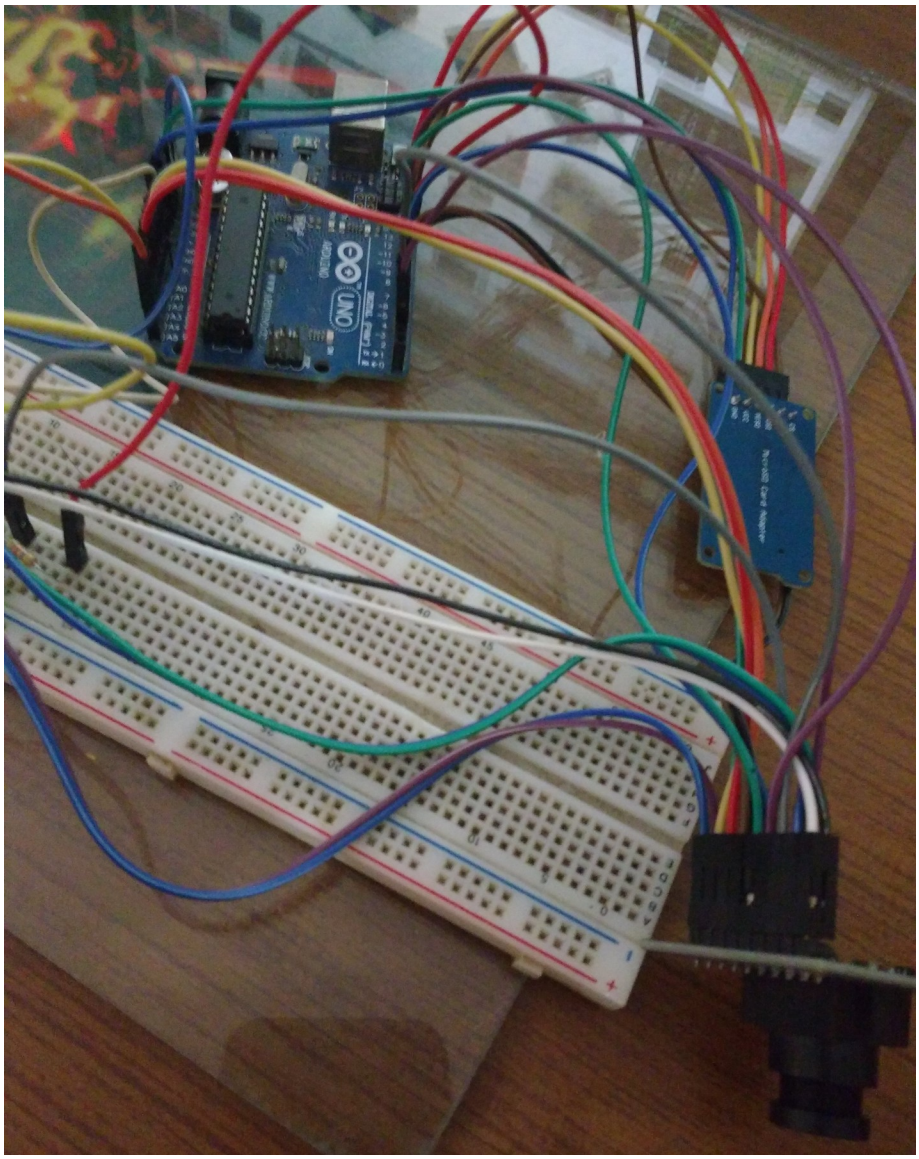
IMAGE CAPTURE USING OV7670 non-FIFO WITH ARDUINO UNO AND SD CARD MODULE

Hardik Kalasua



AIM

Capture a QVGA(320x240) image with OV7670 non-FIFO Camera on an Arduino Uno and save it onto an SD card. The Same has been achieved using the Arduino SD and Wire library.

INTRODUCTION

Camera Module: OV7670 non-FIFO.

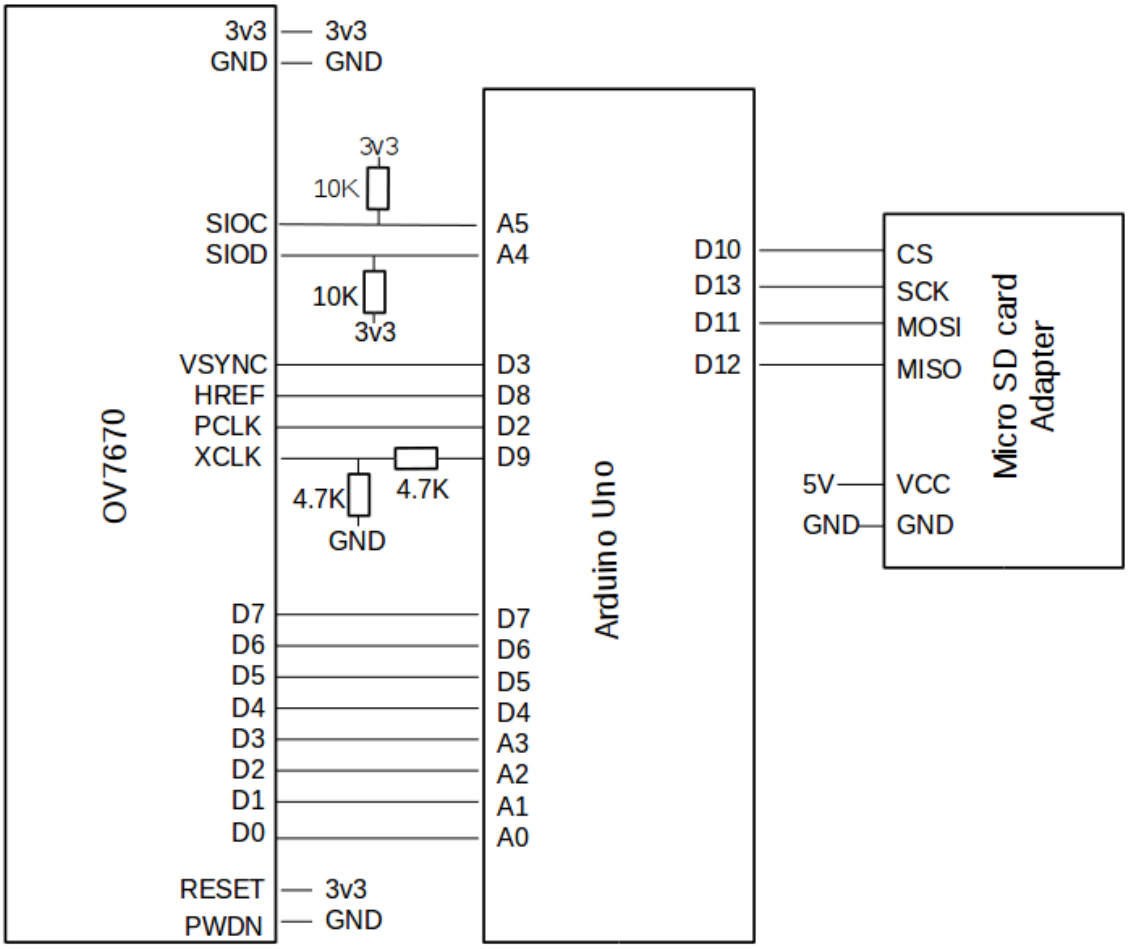
The OV7670 camera module can be understood as consisting of two parts:

1. SCCB (I2C) Controller Section
2. Camera Section
   - **SCCB Interface:** This is basically an I2C interface with which one can access all the control registers of the camera which control various aspects of camera operation like white balance, exposure, resolution etc. One need to write appropriate values to these registers for the camera to work in the desired fashion. We are going to use the Arduino Wire library for this purpose.
   - **Camera Sensor Output:** This is the D0-D7 pinout where the image data is obtained one byte at a time.

SD Card Module: This is a basic SD card adapter module operating at 3.3V. However, an onboard voltage regulator allows us to connect it to Arduino 5V supply. The module communicates with Arduino via its SPI interface. We are going to use the Arduino SD library for this purpose.

Arduino Uno: The very small number of pins on the Uno board makes it a difficult task to control the camera module with SD module. Since most of the pins on Arduino are multiplexed to provide other functions, this doesn't leave us with much choice over the way we can connect the modules.
Another limitation is the small amount of memory(32Kb) on the Atmega328 chip driving arduino.

The Figure below shows the connections between the modules.



XCLK

The camera module requires a system clock input at the pin XCLK. This clock is essential for the SCCB interface to work and for any other camera operation. Note that this clock is different from the SCCB input clock. We are generating an 8Mhz PWM signal at digital pin 9 of the Arduino board using Arduino timer 1.

We start by defining the function to setup XCLK

```
void XCLK_SETUP(void){
```

Next, we need to set the pin 9 as output

```
pinMode(9, OUTPUT);
```

Now we need to initialize Timer 1

We need to set bits WGM13, WGM12, WGM11 & WGM10 in TCCR1A and TCCR1B registers to 1 for Fast PWM mode. Then set bit COM1A0 to 1 to Toggle OC1A on compare match and CS10 bit for clock select with no prescaling.

```
TCCR1A = (1 << COM1A0) | (1 << WGM11) | (1 << WGM10);
TCCR1B = (1 << WGM13) | (1 << WGM12) | (1 << CS10);
```

Next, Set Output Compare Register 1A(OCR1A) = 0.This will lead to a match on every clock cycle and the OC1A output pin will toggle on every match instance. Therefore, the generated waveform will have half the frequency of the driving clock i.e. 8Mhz.
OC1A pin- PB1 (alternate function) pin i.e. Arduino pin 9

```
OCR1A = 0;
}
```

I2C INTERFACE

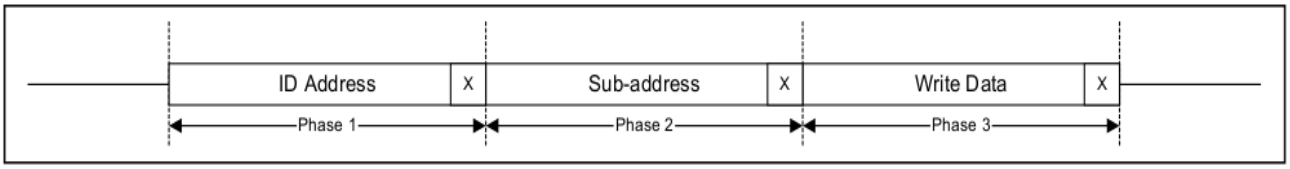
Each I^2C bus consists of two signals: SCL and SDA. SCL is the clock signal, and SDA is the data signal. The clock signal is always generated by the current bus master; some slave devices may force the clock low at times to delay the master sending more data (or to require more time to prepare data before the master attempts to clock it out).

the I^2C bus drivers are “open drain”, meaning that they can pull the corresponding signal line low, but cannot drive it high. Thus, there can be no bus connection where one device is trying to drive the line high while another tries to pull it low, eliminating the potential for damage to the drivers or excessive power dissipation in the system. Each signal line has a pull-up resistor on it, to restore the signal to high when no device is asserting it low.

Since the devices on the bus don’t actually drive the signals high, I^2C allows for some flexibility in connecting devices with different I/O voltages. In general, in a system where one device is at a higher voltage than another, it may be possible to connect the two devices via I^2C without any level shifting circuitry in between them. The trick is to connect the pull-up resistors to the lower of the two voltages. This only works in some cases, where the lower of the two system voltages exceeds the high-level input voltage of the the higher voltage system–for example, a 5V Arduino and a 3.3V accelerometer.

WRITING TO OV7670

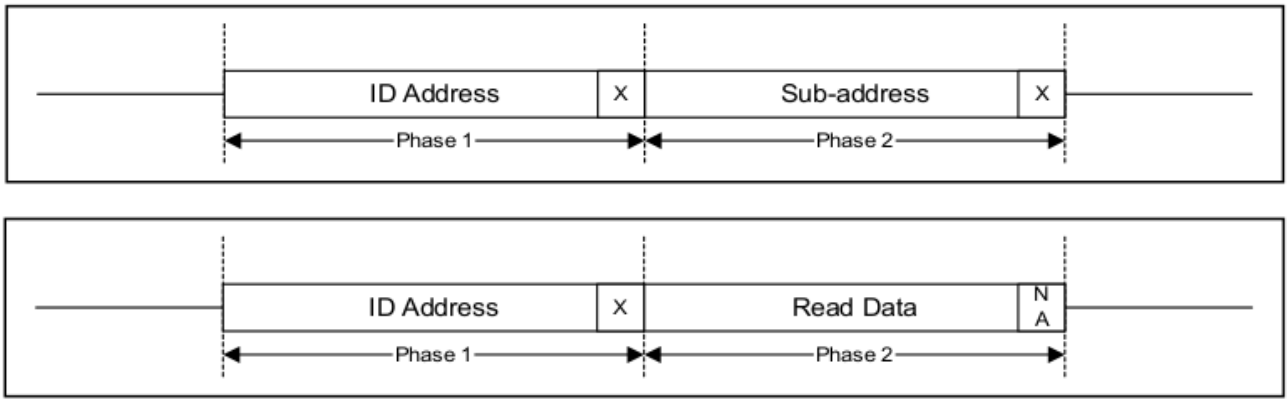
Data is written to the OV7670 Registers in a 3-phase write transmission cycle. The 3-phase write transmission cycle is a full write cycle such that the master can write one byte of data to a specific slave. The ID address identifies the specific slave that the master intends to access. The sub-address identifies the register location. The write data contains 8-bit data that the master intends to overwrite the content of this specific address. The 9th bit of the three phases will be Don’t-Care bits.




```
void WriteOV7670(byte regID, byte regVal){
      7-bit device address(ID Address)
      Wire.beginTransmission(0x21);
      regID-8-bit Register address where data is to be written
      Wire.write(regID);
      regVal-8-bit data to be written
      Wire.write(regVal);
      Wire.endTransmission();
      delay(1);
}
```

READING FROM OV7670

The read cycle consists of 2-phase write transmission cycle followed by 2-phase read transmission cycle. The purpose of issuing a 2-phase write transmission cycle is to identify the sub-address of some specific slave from which the master intends to read data for the following 2-phase read transmission cycle. The 9th bit of the two write transmission phases will be Don’t-Care bits.



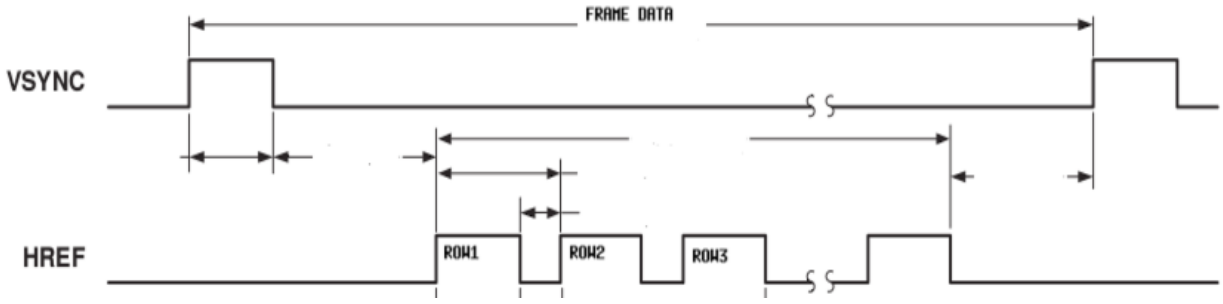

```
void ReadOV7670(byte regID){
        Wire.beginTransmission(0x21);
        Wire.write(regID);
        Wire.endTransmission();

        Wire.requestFrom(0x21, 1);
        Serial.print("Read request Status:");
        Serial.println(Wire.available());
        Serial.print(regID,HEX);
        Serial.print(":");
        Serial.println(Wire.read(),HEX);
}
```

RECEIVING IMAGE BYTES

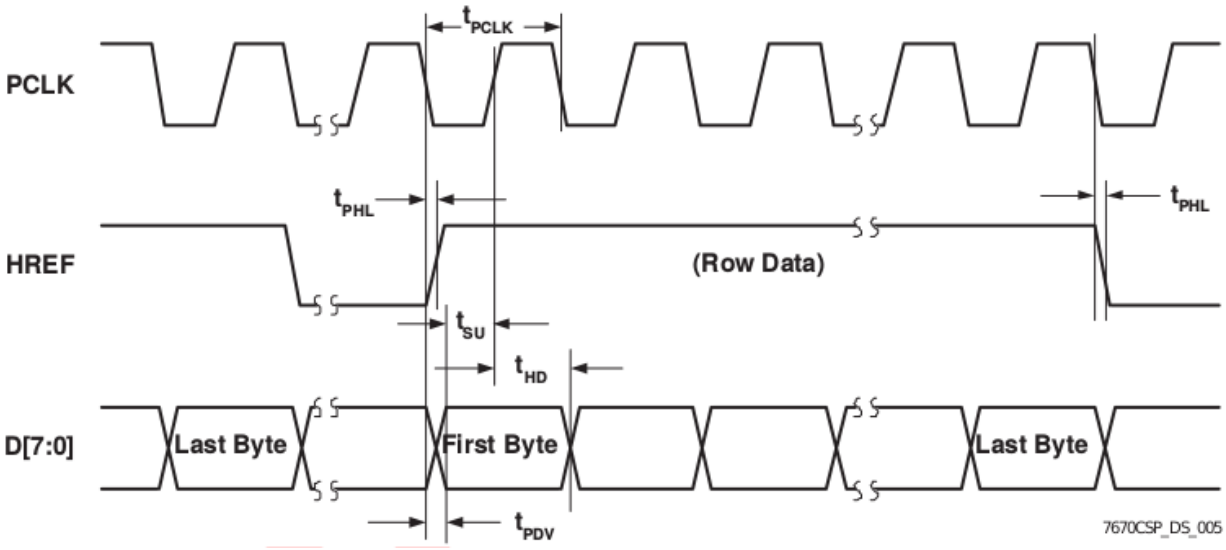
The OV7670 sends the data in a parallel synchronous format. According to the datasheet, the system clock (XCLK) frequency must be between 10 and 48 MHz although it works fine with 8 MHz clock which can be generated on Arduino Uno at pin 9 using timer 1.

After a clock signal has been applied to the XCLK pin, the OV7670 will start driving its VSYNC, HREF and D0-D7 pins.

The falling edge of VSYNC signals the start of a frame, and its rising edge signals the end of a frame. The image format used in this project is QVGA (320 x 240). That means each QVGA frame has 240 rows and each row has 320 pixels. Each row data must be captured during the high state of HREF i.e. D0-D7 must be sampled only when HREF is high. The rising edge of HREF signals the start of a row, and the falling edge of HREF signals the end of the row.



D0-D7 must be sampled at the rising edge of the PCLK signal. All these bytes sampled when HREF was high, correspond to the pixels in one row.



Note that ***one byte is not a pixel***, and in the format chosen i.e. YUV422, two bytes correspond to a pixel where the first byte has luminescence information.

The 240 lines, equivalent to a frame, are captured during the low state of VSYNC.

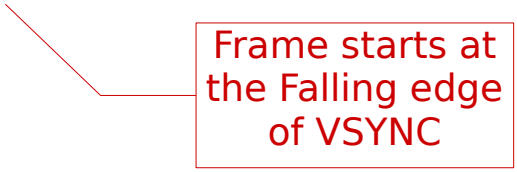
During HREF high state, we must capture 320 pixels, equivalent to a line.

Now, how fast the data is being sent depends on the frequency of PCLK. By default, the PCLK will have the same frequency of XCLK, however prescalers and PPLs can be configured using the SCCB, to produce a PCLK of different frequency. This is particularly important because writing data to SD card is timeconsuming and not a very fast process. This puts up timing restrictions on how fast data can be sampled. In this project, PCLK has been slowed down by using maximum allowed prescaler bits.

```
WriteOV7670(0x11, 0x1F); //Range 00-1F
```

SAVING DATA ONTO SD CARD

```
void QVGA_Image(String title){
  int h,w;
  File dataFile = SD.open(title, FILE_WRITE);
  while (!(PIND & 8)); // wait for HIGH
  while ((PIND & 8)); // wait for LOW
    h = 240;
  while (h--){
       w = 320;
      byte dataBuffer[320];
    while (w--){
      while ((PIND & 4));   // wait for LOW
        dataBuffer[319-w] = (PINC & 15) | (PIND & 240);
      while (!(PIND & 4));  // wait for HIGH
      while ((PIND & 4));   // wait for LOW
      while (!(PIND & 4));  // wait for HIGH
    }
    dataFile.write(dataBuffer,320);
  }
    dataFile.cose();
    delay(100);
}
```



BYTE 1 captures the luminescence of the image and therefore it is being sampled to obtain a greyscale image. BYTE 2 is being discarded due to memory limitations of the Arduino Uno board.

WRITING A BMP IMAGE

A python script writes the raw image data obtained to a BMP image file. The first 54 bytes in a BMP file is the header data that sets the file parameters. All following bytes are pixel data. Each pixel has 3 bytes of information corresponding to the Blue, Green and Red channels. Therefore to obtain a greyscale image, all 3 bytes are written with the same luminescence data.

Row size for a BMP image is given by the formula:

Row Size = Floor((BitsPerPixel*ImageWidth+31) / 32)*4

Each row consists of 1 row of image pixel data plus some padding bytes. In this case where image width is 320 pixels and height is 240 pixels, row size:

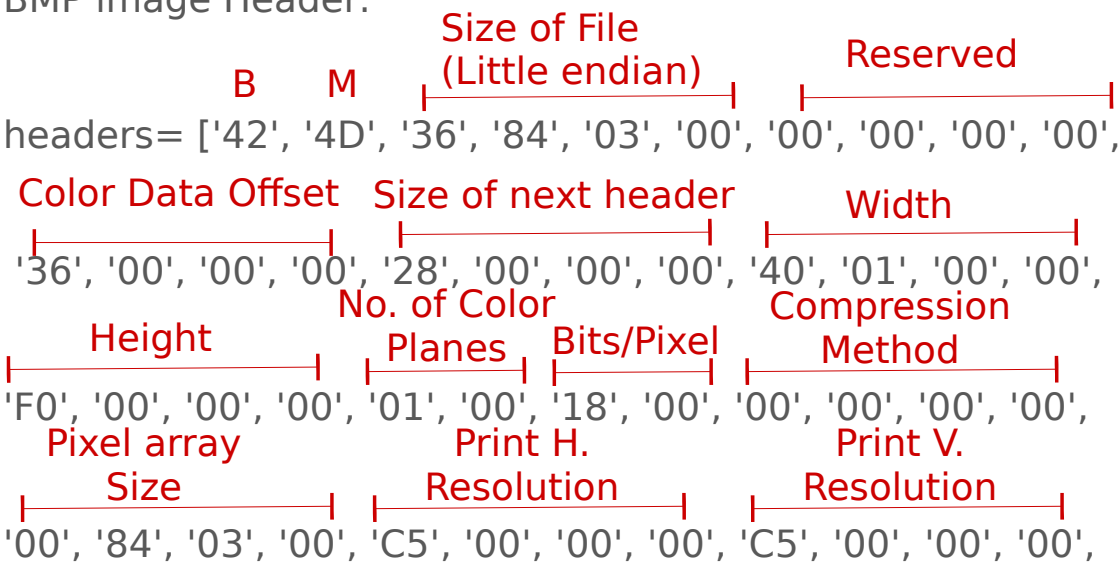
row size = floor((24*320+31)/32)*4 = 960 bytes

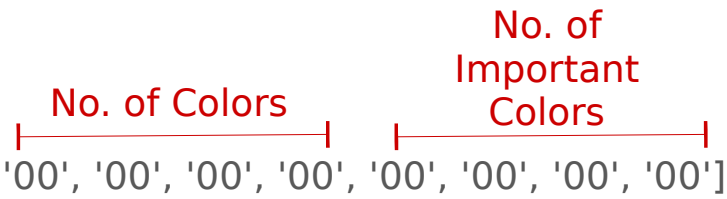
Padding bytes = bytes in a row – ImageWidth*3 = 0

So, in this case, no padding bytes needed.
Therefore the size of a BMP file can be calculated as:

Size = Header bytes + imageHeight*RowSize

BMP image Header:





SUMMARY

So to summerise, these are the steps needed to capture an image with OV7670:

1) Connect the modules according to the schematics.
2) Generate 8mhz XCLK at digital pin 9.
3) Initialize SCCB interface
4) Reset OV7670 register values
5) Write Registers for QVGA settings
6) Write registers for YUV422 settings
7) Initialize SD module and make sure it is working
8) Capture one frame raw data to SD card
9) Write the raw data into a BMP file to obtain the final image

LIMITATIONS

1) Few no. of I/O pins on Arduino Uno board
2) Small amount of memory(32Kb) on Uno board.
3) 8mhz of maximum generated clock speed by the uno board which is less than the specified operating range of OV7670 camera.
4) Slow writing speed to the SD card.

REFERENCES

Aparicio, J. *Hacking the OV7670 camera module*. http://embeddedprogrammer.blogspot.in/2012/07/hacking-ov7670-camera-module-sccb-cheat.html

APPLICATION NOTE Serial Camera Control Bus Functional Specification. (2002). 2nd ed. [ebook] Omnivision Technologies. http://www4.cs.umanitoba.ca/~jacky/Teaching/Courses/74.795-LocalVision/ReadingList/ov-sccb.pdf

BMP file format. https://en.wikipedia.org/wiki/BMP_file_format

ComputerNerd/ov7670-no-ram-arduino-uno. https://github.com/ComputerNerd/ov7670-no-ram-arduino-uno

I2C – learn.sparkfun.com. https://learn.sparkfun.com/tutorials/i2c

OV7670/OV7171 CMOS VGA (640x480) CAMERA CHIP Sensor. (2006). [ebook] OmniVision Technologies, Inc. https://www.voti.nl/docs/OV7670.pdf

YUV Colorspace. http://softpixel.com/~cwright/programming/colorspace/yuv/

Arduino Serial Peripheral Interface. https://www.tutorialspoint.com/arduino/arduino_serial_peripheral_interface.htm