PCB

This PCB was designed with EasyEDA and then manufactured by JLCPCB (I know they're a meme but they're actually really good).

BOM (Bill of Materials):

R1:	10KΩ	
R2:	10KΩ	
R4:	220Ω	

R3, R5: Either use a 68Ω resistor or use two parallel 100Ω to give an equivalent 50Ω resistance which is acceptable.

R6:	4.7KΩ or 5.1KΩ	
R7:	4.7KΩ or 5.1KΩ	
U1:	Arduino Nano	
U2:	Camera	(OV7670
U3:	IIC OLED Screen	
U4:	VGA terminal	(CH4RHNVGA)
J1:	Female pin headers for arduino	
J2:	Pins for SB1	
J3:	Pins for SB2	
J4:	Pins for SB3	
SB1:	Short to enable Camera	
SB2:	Short to enable OLED	
SB3:	Short to enable VGA	

Note: You may need to mount SB2 upside down so it does not hit the camera module

Solder Bridges are for the pin headers - use the cap to short them and select a project. There are mounting holes dotted around in case you would, for some reason, like to screw it down into place.

Use the female header pins for the arduino by soldering them onto the PCB. Then you can place the Arduino into the headers so it is removable - it also stops you having to solder the delicate Arduino.

Switching Between Projects

Each project has an SB (solder bridge) connection. Once the headers are soldered, place a wire between the two or short them otherwise to enable power to the project (on the projects, the GND or VCC for the components are isolated through the headers). You can't run more than one project at the same time - the camera takes up too many pins and requires a different logic level to the OLED screen, and VGA uses too much memory and all available hardware timers, so it's not possible to use it with anything else.

Incompatibility Explanation Table

	VGA	OLED	Camera
VGA	■	Too little Memory!	Too little memory!
OLED	Too little memory!	■	Conflicting pins!
Camera	Too little memory!	Conflicting pins!	■

About

The projects in this document were designed as a proof of concept device that would give a practical implementation of theory.

Practical implementation	**Theory discussed**
OLED I2C screen	I2C packet structures
OV7670 camera stream	Logic family voltage level shifting
Arduino VGA output	VGA and CRT monitor timing and theory

Project 1: OLED I2C Screen

- Usually one colour
- 0.96” display
- 128x64 px display (not quite 8K, I’ll admit)

Part 1: I2C packet structures

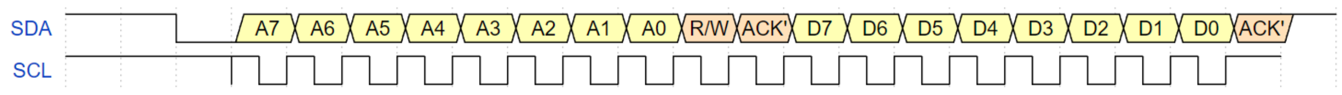
Begin my rant about I2C. First of all, its know by all of these names:

- I2C
- IIC } Inter Integrated Circuit
- I^{2}C
- TWI Two Wire Interface
-

Seriously. Could they not just pick one? The next part is the fact that I2C requires pull-up resistors because it’s open-drain, but some manufactures add them on the host board, some on the slave device and some on none (you add them externally). Two pairs of pull up resistors isn’t a good idea because it causes current to always flow from the pull up resistors to the ground of the host (which is negative current from the perspective of the host, which it often cannot tolerate). Equally, the resistance of the resistors needed depend on the speed of the bus, which depends on the peripherals used. As well as this, I2C has the problem that every time ACK is sent, the direction of the bus changes in the middle of the packet. Also it limits I2C to half duplex transfer (see below)

Simplex	Data sent either one way or the other	TV or radio comms PS/2 Other v. old protocols
Half-duplex	During one packet, data is sent both ways. Data is not sent both ways at the same time	PCI I2C USB 2.0 or previous some UART setups
Full-duplex	Data has the ability to be sent both ways at the same time. Devices receive and send data at the same time.	SPI USB 3.0 or later some UART setups

But, that being said, I2C has a lot of uses, mainly the fact that I2C only needs 2 wires for data (technically, VCC and GND are optional extras). SPI is simpler to implement, but needs more logic control to do full duplex, and needs more connections. I2C is better for this project, because A) SPI OLEDs were more expensive and B) I2C is more interesting to show the structure of.



(made using wavedrom editor and free time - when SCL first changes at A7 this is a mistake but I can't fix it for some reason. SCL would actually stay high rather than pulse low for virtually 0s)

Essentially, I2C spends most of its time at VCC (which is weird because this obviously uses more power). The addresses are sent MSB→LSB (as with data). Weirdly, ACK is actually $\overline{ACK}$, which in theory means that by pulling down the SCL and SDA lines to GND would actually make an I2C host think a device was working and attached - just a write-only I2C device.

Pulling down the SDA line while SCL is high (Why is the clock defaulted to VCC?), triggers the start of an I2C transaction. Then the addresses are sent on the rising edge and then the R/W flag, which enables devices to easily switch between reading and writing. The ACK is the difficult part of the I2C protocol to implement on a hardware level because it requires the direction of the bus to change to slave→master, which obviously means synchronisation has to be exact. The data is then sent one byte at the time. Note there is no limit on how long it is, only that an ACK cycle happens at the end of each byte transmitted. In the case of just 1 byte transmitted (which is fairly standard), this is a bit strange - having 2 ACK bits means that the only different info you could get compared to 1 is that the address was correct but the data was corrupted? in which case the transmission failed, so only one ACK bit would have told you that. The stop bit is the weirdest because it requires SCL to go HIGH before SDA does the same. That won't be a pain to time correctly.

And then the whole process starts again. I2C is very flexible in this regard because the bus can switch between multiple slaves and masters, so the slave can transmit data in the same way as the master if it wants to. I2C is also flexible because it can be implemented in a software-only regard.

Part 2: I2C OLED

Short SB2 and disconnect SB1 and SB3.

Prerequisite libraries (use library manager):

- adafruit_SSD1306

- adafruit_GFX

You must also include the <Wire.h> (I2C) library.

Class definitions:
#define SCREEN_WIDTH 64
#define SCREEN_HEIGHT 32
#define OLED_RESET   -1
// This OLED has no reset pin, so it is shared with the Arduino RESET. So we define RESET as -1# (or -1 if sharing Arduino reset pin)
#define OLED_RESET   -1 // Reset pin # (or -1 if sharing Arduino reset pin)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

Setup() Functions:

To set up the OLED, this code is placed in the setup function.

```
if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
   Serial.println(F("SSD1306 allocation failed"));
   for(;;);          // this is a "naked for loop" - it is the same as while(true);
}
```

Library macros:

BLACK	(colour reference)
WHITE	(colour reference)

The useful library functions are:

display.display();
Display the contents of the OLED buffer and clear buffer. Doesn't need to be called every time the screen is updated, but good to use after batch operations.

display.clear();
Clears the display (and the buffer I believe.) to BLACK

display.drawPixel(int x, int y, int colour)
set a pixel at the coordinates to a colour.

display.drawLine(int startX, int startY, int endX, int endY, int colour)
Draws a straight line between two points with the colour selected.

display.width()
Returns the width of the display

display.height()
Returns the height of the display

Have fun. I think you know how to program the rest.

Project 2: OV7670 Camera Module

Mmm VGA resolution camera. Real 2000s YouTube feel…

When you realise this camera's resolution is 640x480 (VGA standard)

Part 1: Pinout

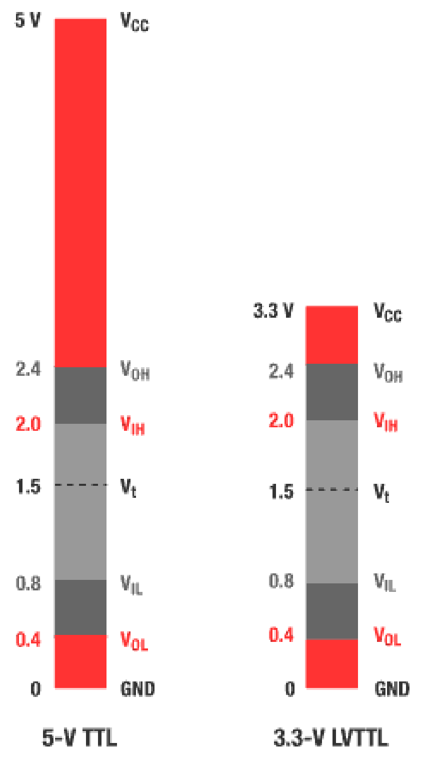
The OV7670 has the following pinout:

3V3:	3.3V power. because nothing is 5V compatible
SIOC:	because calling it SCL was apparently too difficult.
SIOD:	Because calling it SDA was apparently too difficult.
VSYNC	Vertical refresh of the camera's data
HREF	Horizontal refresh of the camera's data
PCLK	Further clock referencing and sync
XCLK	As above, but from arduino to camera
D7->D0	Output pixel data from camera to arduino
PWDN	Active low signal to pause operation. Tied to 3V3 on this PCB.
RST	Active low signal to reset the camera. Tied to 3V3 on this PCB.

Part 2: Level shifting

The arduino uses 5V logic (well technically anything over 3V is high on the system). However, the camera uses 3V3 logic. So for XCLK we need to talk about level shifting.

Level shifting is the process of changing a voltage level scheme. The arduino uses TTL logic, but the camera uses LVTTL. The different levels are shown below:



The terminology is as follows:

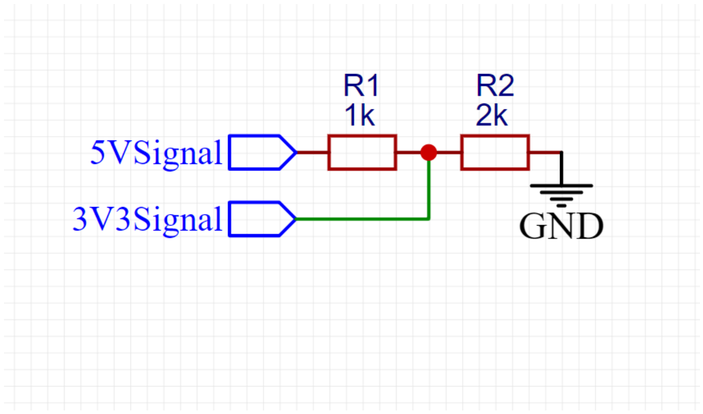
V_{OL}	Low (0) voltage output
V_{IL}	Low (0) voltage input
V_t	Threshold voltage
V_{IH}	High (1) voltage input
V_{OH}	High (1) voltage output

Note: TTL = transistor-transistor logic
LVTTL = low voltage transistor-transistor logic

transistor-transistor logic simply describes how the logic is formed. I prefer CMOS because it requires considerably lower current than TTL. As you can see, the inputs are not compatible with each other, since the 5V output from the arduino will damage the 3.3V input of the camera. Level shifting is the process of changing these voltages. There are two ways to do it.

Method 1 - Voltage dividers

If, as in this case, your application is at a relatively low speed and unidirectional, you can just use resistors to step down the voltage. For example, a 5V to 3V3 voltage can be made like so:



We can easily prove this works using the voltage divider equation.

$$V_{out} = V_{in}\left(\frac{R2}{R1+R2}\right)$$

$$R1 = 1000,\ R2 = 2000$$

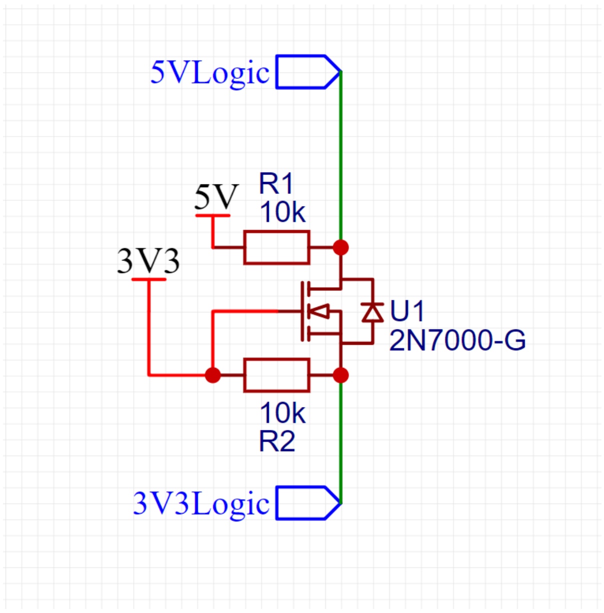
$$V_{out} = 5\left(\frac{2000}{1000+2000}\right)$$

$$V_{out} = 5\left(\frac{2}{3}\right)$$

$V_{out} = 3.3......$ which is close enough. And when V_{IN} = 0V (for a 0), then there is no voltage in the system so V_{out} must also be 0V. The resistances are arbitrary provided the ratio is correct: for a 5V system, the ratio of R1:R2 must be 1:2. This is one of the drawbacks of this system - for each input voltage, the required resistors change.

Method 2 - MOSFETs

By using MOSFETs (n-type for this case), we can create a bidirectional level shifter that can work with many different voltage levels without the need for altering the circuit. For example, the below circuit changes 5V logic to 3.3V:



The great thing about this circuit is that the system works both ways and can be dynamically adjusted by changing the power inputs (e.g the 3V3 connection could be swapped for 1.8V and then you would have a different voltage protocol with the same setup). . It also draws less current because MOSFETs are voltage controlled.

Note: To work properly, the upper side in this example must be a greater voltage than the lower side. This is because otherwise the body diode of the N-MOSFET will allow current to flow from the higher to lower voltage.

Comparison of the systems

	Resistors	**MOSFETs**
Components:	2	3
Cost:	V. cheap	Cheap
Directional:	One way	Bidirectional
Speed:	Fast	V. Fast
Adjustability:	Fixed inputs and outputs	Fully dynamic
Current draw / power	Depends, but can be high	Low in most cases

In the case of this project, we use the resistor method because it is simpler. You can see it on the PCB as resistors R6 and R7.

Part 3: Using the camera

With the theory out of the way, we can now implement the camera. In this case, we use a serial connection to send each bit of the image one by one over the USB connection to the computer which can then save this or display it.

Libraries

- Adafruit_ST7735 (not needed for this example but still a dependency).
- LiveOV7060

To install LiveOV7060, download the github repo here, extract it, recompress the /src/lib/LiveOV7670 to a zip file and then use the Sketch/include library/add .zip library in the IDE.

Obviously I didn't write this library. Mainly because I don't program 24/7, and the camera has so many registers and config registers, that I would need several header files just full of I2C addresses, which I cannot be bothered to write.

To upload the relevant sketch, do the following:

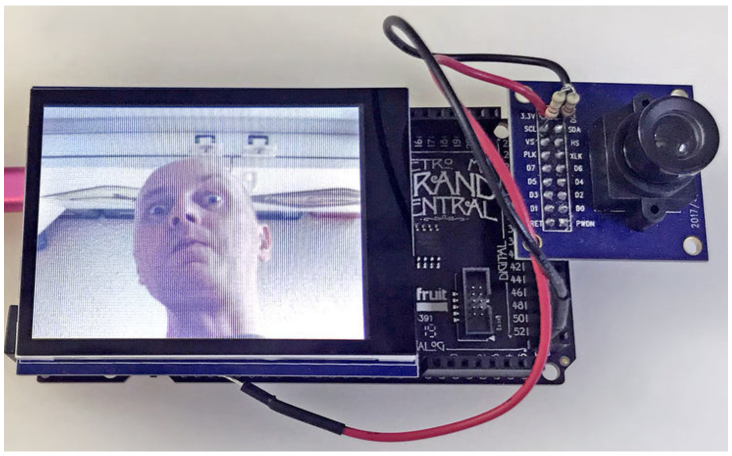
1. Open "Path to downloaded zip"/src/LiveOV7670/LiveOV7670.ino in the Arduino IDE
2. Open the file "setup.h" in the sketch and change the EXAMPLE definition to 3
3. Open the file "ExampleUART.h" and change the UART mode to one you prefer
4. Upload the sketch
5. Use the serial port reader here to monitor the images that are sent.

If this doesn't work, there are many other codebases online with the same pinout. Weirdly, different modules seem to like different libraries to work. But there is another thing to try first:

- For the voltage divider, first use 4k7 / 4k7
- If that doesn't work, use 1k / 2k
- If *that* doesn't work, scope the XLCK signal and if it doesn't rise above 3V, just short out the resistor.

Other codebases:

Ethan2023/CameraV2



One last thing… can we talk about the 2nd result on google when you search “OV7670 with arduino”:

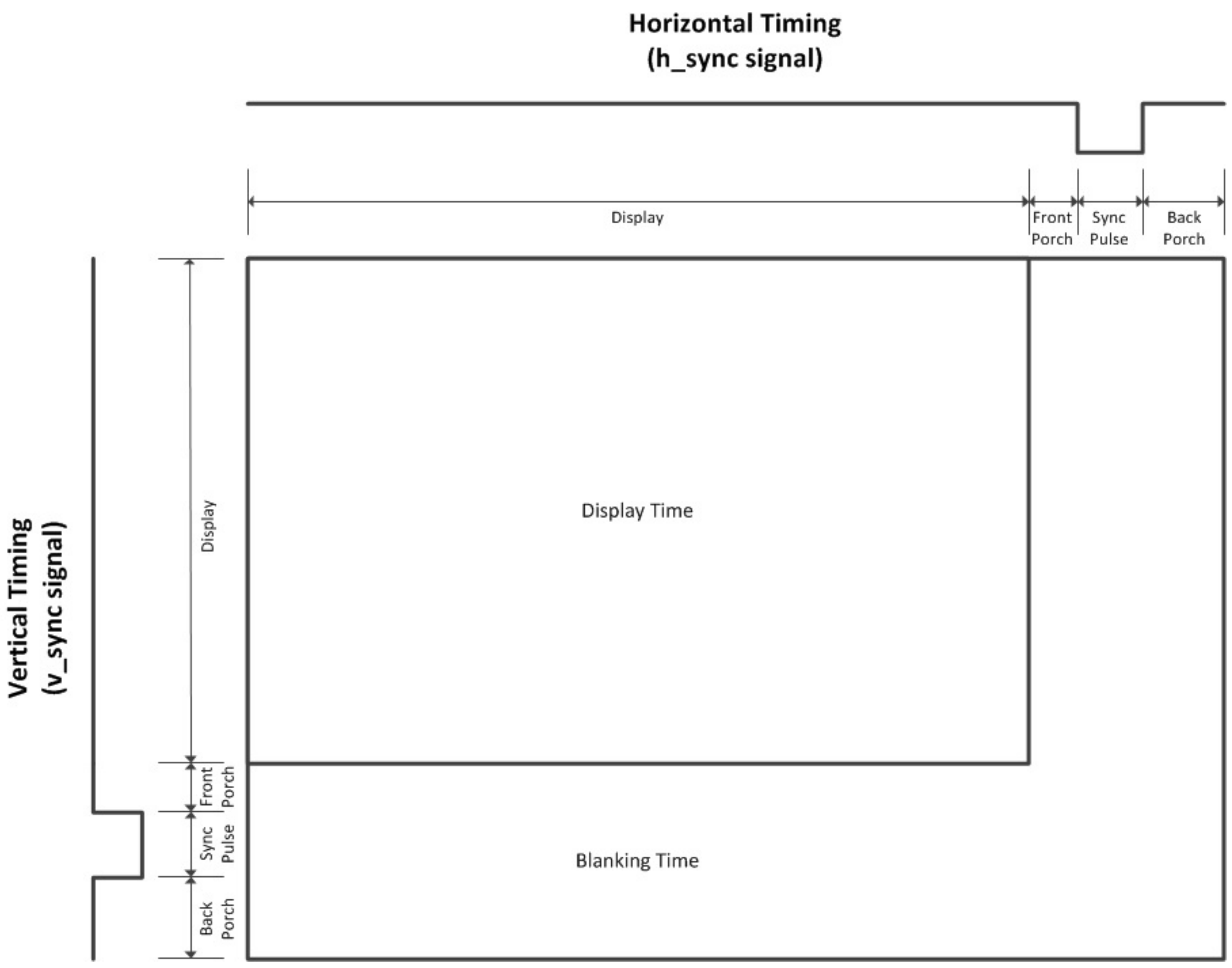
That's not what I would put online publicly if I were him.

Project 3: VGA with Arduino

Warning! generating VGA uses **all** of the hardware timers and typically 90% or more SRAM. If you do seriously want to use a VGA output, I'd use two arduino's - one for calculating screen data and sending it over I2C or UART, and another with the VGA sketch to actually render the image. On a MEGA, you have more SRAM so should be fine - but the problem with the MEGA is that it's just far too big to be actually useful inside a dedicated system build, and can't be placed on a PCB

Important note: The VGA terminal will feel tight when you press it into the PCB. You still need to solder it in place. Make sure to also solder the mounting legs to the holes in the PCB.

Part 1: VGA timing



This is the timing for a VGA signal. It's entirely analogue, and uses a predefined clock rate. We should be aiming for ~25.175MHz, but this is an Arduino (16MHz for historical reasons, actually designed for 20MHz and can be run at ~30MHz. Someone did 63.5MHz here, but I don't have any liquid nitrogen). For this reason, you're likely to see slightly fuzzy images as the pixels move in and out of sync, but this project is more a proof of concept.

Starting with HSYNC. HSYNC is active-low and is used to tell the monitor to move onto the next scanline. Remember VGA was designed for CRT monitors, which are damaged by trying to shoot electrons at high voltage outside of the viewable area. The viewable area is 640px (or 640 rising edges of the clock), which is the same as 25.422045680238 μs. For simplicity, I'll measure in px The front porch is the time between the viewport and the sync pulse, which for this resolution is conveniently 16 pixels - easy to time against the clock with a 4 bit counter. To synchronise, the HSYNC line is pulled low and then after 96 pixels it is then pulled high. Then, the time taken to the counter reset (next line) is 48 px - this is called the back porch. After this, the next horizontal "scanline" is sent to the monitor. We can calculate the rate each scanline occurs by using: $refresh_{vertical} = \frac{f_{system\ clock}}{px_{h\ total}} = \frac{25{,}175{,}000}{800} = 31.46875\ KHz.$ Therefore, the VSYNC counter refreshes ~31,000 times per second.

VSYNC is exactly the same but with different timings. It occurs less frequently because VSYNC only does its sync pattern after all horizontal lines have been drawn, The timings are shown below:

	HSYNC Pixels	VSYNC Pixels
Visible Area	640	480
Front Porch	16	10
Sync Pulse	96	2
Back Porch	48	33
Whole Frame	800	525

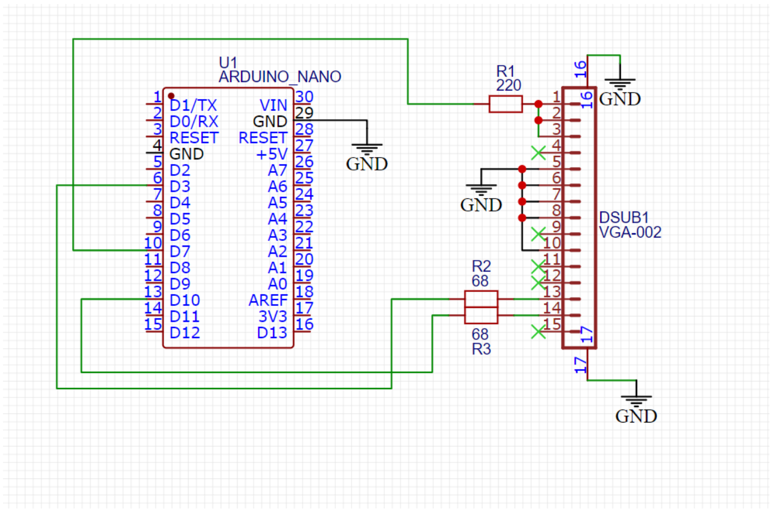
Part 2: Implementing on Arduino

The required libraries for this are:

- Adafruit GFX (get from the library manager)

- VGAX (get from here and install using the “add .zip library”)

Schematic:



Note the shorted RGB lines to give monochrome colour - this is just because adding colours is too complex. Someone has created a library which uses 2 of the 3 colour lines but you are limited to 120x60 resolution which isn’t much better than an OLED screen. You could use another microcontroller to generate colour information by using BJT transistors to limit the voltage across each video pin, but remember you need to create different colours for only the text in most cases, so you would also need to decode the video info on the second microcontroller. Pins 16 and 17 are the mounting holes, which can be grounded as a shield terminal.

VGA pinout:

1	Video RED
2	Video GREEN
3	Video BLUE
4,11	*Often reserved for shield connection*
5,6,7,8,10	GND
9	*Often reserved for 5V*
12	*Often reserved for I2C SDA*
13	Video HSYNC
14	Video VSYNC
15	*Often reserved for I2C SCL*

I2C is occasionally used to configure the monitor at the end of the cable, e.g. settings or ROM that may hold important data. It can also be used with a monitor hosting the bus to tell the computer what it can and can't display, as well as its supported resolutions etc. I’ve found you only need to connect the HSYNC and VSYNC pins for a monitor to detect a source, and then the video pins (1,2,3) to show an image. This project outputs the universally required 640x480, so the I2C lines aren't needed because we know the monitor has to support it to be VGA compatible.

Obviously with the PCB just short SB3, disconnect SB1 and SB2 and upload the relevant sketch. VGA uses the PWM pins to generate the signal (3,7,10). If you look at the library's source code, it actually uses the assembly references of PORTB and PORTD etc. instead of digitalWrite() to make the code run as fast as possible. Example code is shown below:

Important note: VGA library uses all of the hardware timers of an arduino in order to keep track of sync signals. If you want to use the delay function, use the *VGA.delay(int uS);* function because delay(int uS) will be ignored since all hardware timers are being used..

```
---
#include <Adafruit_GFX.h>
#include <VGA.h>
VGA display = VGA();

void setup(){
        Serial.begin(9600);
        display.begin();
        display.clearDisplay();
        display.setTextSize(1);
        display.setTextColor(WHITE);
}
void loop(){
        if (serial.available(){
                String message = Serial.readString();
                if (message == “circle”)
                        display.drawCircle(100,100, random(50,150), WHITE);
                else if (message == “clear”)
                        display.clearDisplay();
                else if (message == “line”)
                        display.drawLine(0,0, random(50,100), random(50,100), WHITE);
        }
}
```

Most draw functions also have a “fill” function e.g. fillRect(), fillCircle() etc.

VGA output is actually very useful. Compared to HDMI, the ability for VGA to run on such slow systems is very handy - HDMI requires the pixel clock to often be in the hundred-MHz or GHz range. You can always use a cheap VGA to HDMI converter to change over the signal - but remember that these converters use active microprocessors because VGA is an analog protocol but HDMI is a digital signal.

Do be aware that the HDMI port on your laptop if output only in most cases

VGA further cool designs

This ESP32 with 2 cores and 8 MB of QSPI PSRAM (Quad SPI (SPI with 4 data lines) Pseudo-static RAM) is capable of multiple resolutions, including 720p support! But it has 4000 times the RAM of this system…

This video card by Ben Eater (a great channel for low level hardware design) is capable of displaying a colour image from an EEPROM chip on a 640x480 display. But good luck making it dynamic.

This version of VGA from an arduino uses 2 bit colour but sacrifices resolution to do so.

This version of VGA from an arduino uses one IC to give a 320x300 resolution in a much more stable manner.