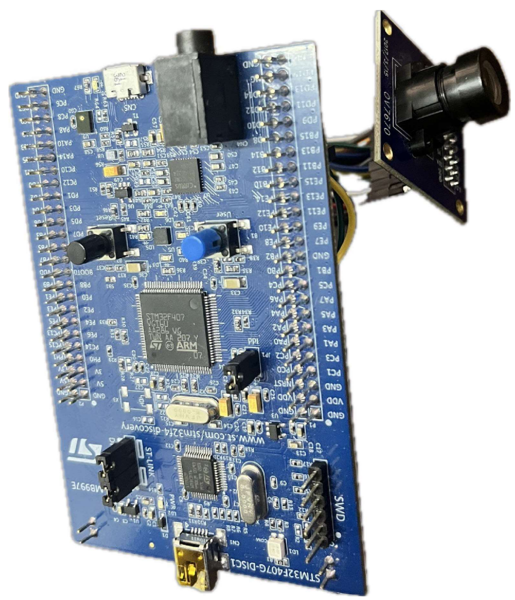
Hardware Used in Project

STM32F407G Discovery Board

- ARM 32-bit M4 CPU
- Frequency up to 168 MHz
- 192 kB RAM (64 kB CCM RAM)
- SWD and JTAG debug interfaces
- I2C, USART, UART interfaces
- DCMI interface available
- Built in STLink V2



OV7670 Camera Module

- I/O Tolerance 2.45V – 3.0V
- Image array of 656 x 488 pixels
- Supports YUV/YCbCr/RGB Formats
- Supports resolutions from VGA and below (QCIF Supported)
- Input clock 10 MHz – 48 MHz
- Frame rates up to 30 FPS
- Built in image processing (exposure/gamma etc.)
- Uses SCCB interface for camera parametrization (similar to I2C)

Development Tools

Software Tools

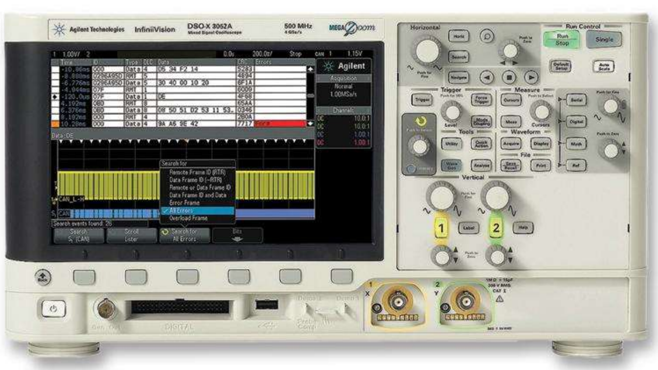
STM32 CubeIDE

TERA TERM

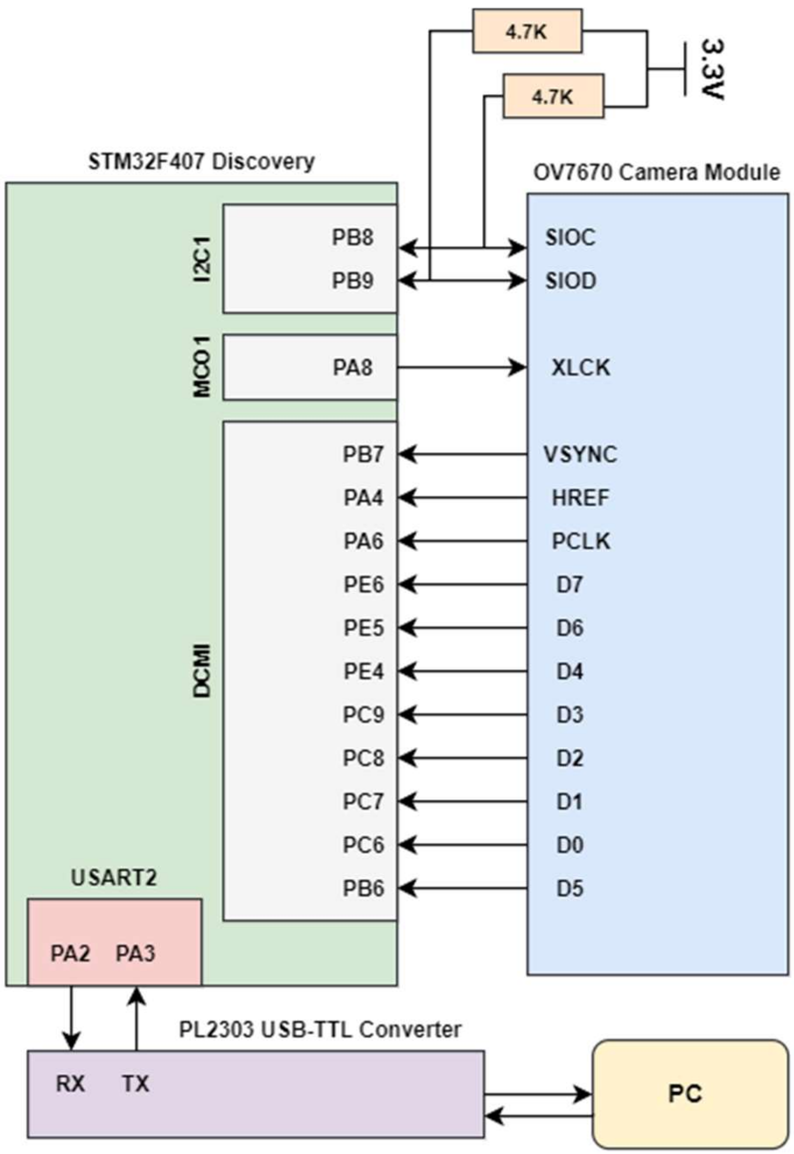
WaveForms

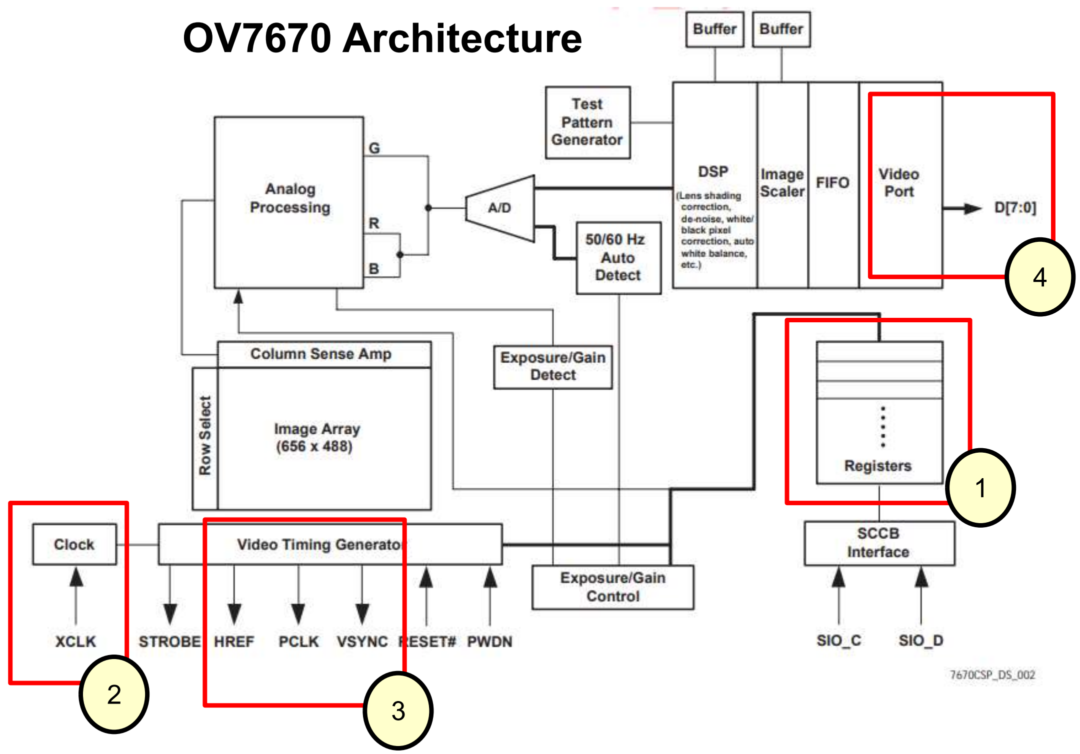
STM32 Standard Peripheral Libraries

Hardware Tools



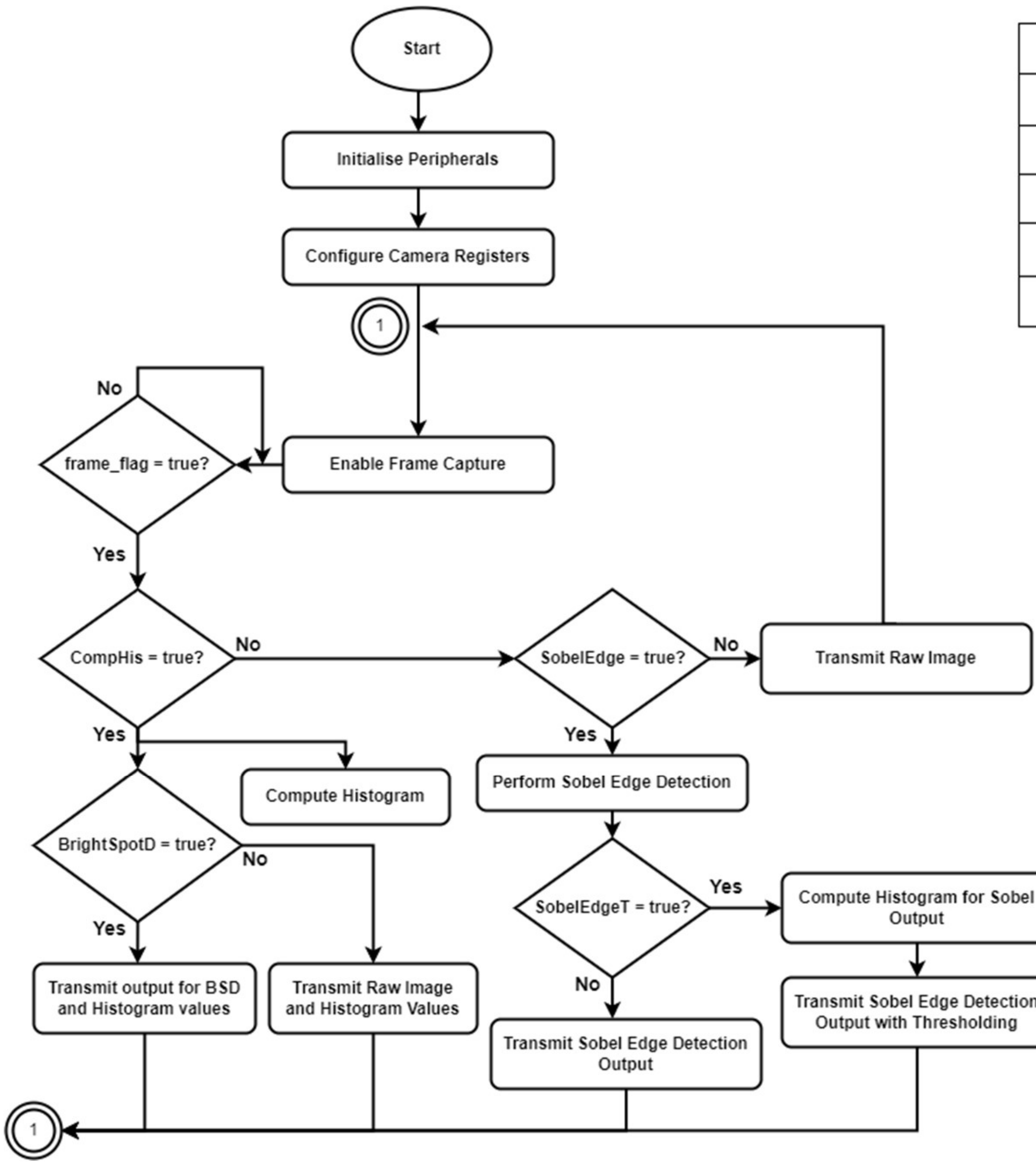
Overview of Hardware Design





1. OV7670 registers are configured using the SCCB interface. (color format, resolution etc.)
1. Clock input from MCU is provided via XCLK.
2. Clock output from OV7670 used for data acquisition is generated via Video Timing Generator (VTG).
3. Image data is received via Video Port (VP) through 8 lines.

Overview of Firmware Design



Command	Flag Set	Response
L	sendLine_flag	Transmit a new line of pixels via USART2
H	CompHis	Enable histogram computation
D	BrightSpotD	Enable bright spot detection
E	SobelEdge	Enable Sobel edge detection
T	SobelEdgeT	Enable Sobel edge detection with adaptive thresholding

Default Operation Mode:

- Stream unprocessed frames to PC

Histogram Mode:

- Compute Histogram and statistics σ,μ
- Transmit σ and μ to PC

BSD Mode:

- Enable Histogram to compute threshold, σ+μ
- Perform adaptive thresholding on image and transmit output frames

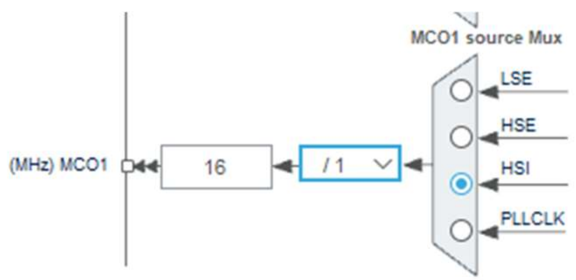
Sobel Edge Detection Mode:

- Perform Sobel Edge Detection
- Perform thresholding if 'T' command received and transmit output frames

Configuring OV7670 Camera Module

Microcontroller Clock Output (MCO)

According to OV's Datasheet, a clock of 10 - 48 MHz is required, but past implementations have been most stable with an 8 - 16 MHz input clock.



- STM32F407's MCO1 is used to supply input clock, with clock sourced from **HSI (16 MHz)**
- High Speed **PLL Clock (33-168 MHz)** is not used as it causes issues with SCCB communication.

OV7670 Register Configuration

Before setting up the communication interface to configure the camera registers, the register settings were first defined according to past implementations [6].

Register Name	Address	Description	Value to Write
COM7	0x12	Reset all register values	0x80
COM7	0x12	Set output resolution to QCIF, YUV422	0x8
CLKRC	0x11	Adjust PCLK to allow frame rate of 15 FPS	0x01

Important register values that define the output resolution, colour format and PCLK settings.

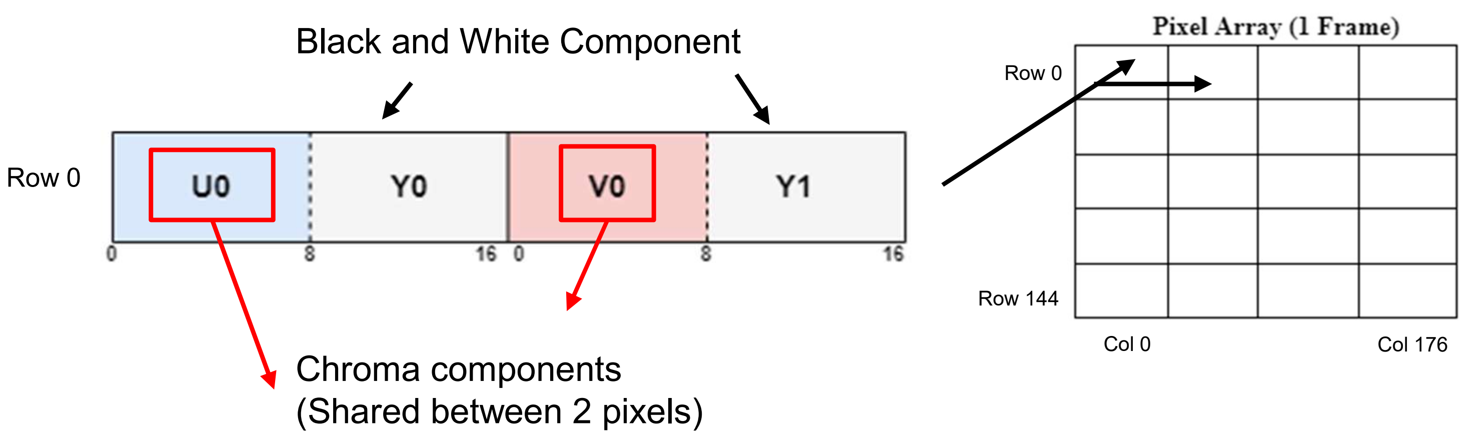
Data Output Format

The OV7670 Camera Module supports several frame resolutions and color modes. We will be using the following settings to best suit our applications, which is for a mobile robot.

Resolution	Colour Format	Bytes Per 2 Pixel (B)	Frame Size (kB)
QCIF (176 x 144)	YUV422	4	51 kB

The YUV422 format has 3 components, namely the luminance (Y) and chroma (U,V) components with each component consuming 1 byte for each pixel.

However, what makes this format memory-saving is that the UV components are shared by every 2 adjacent pixels.

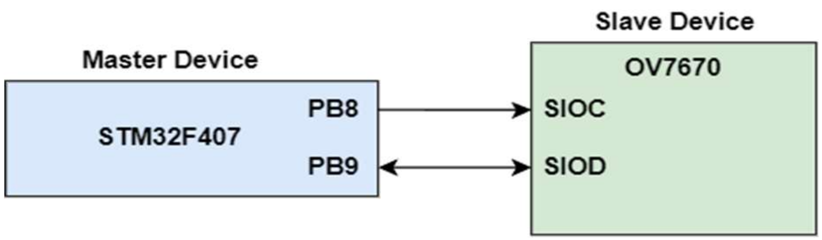


Configuring OV7670 Camera Module

I2C Communication Interface

The STM32F407's I^{2}C1 peripheral was used to write values to the OV7670 register via its SCCB Interface.

I2C Interface Connection



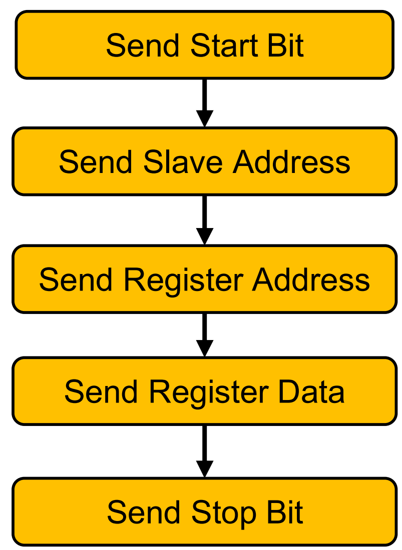
Modifications to I2C Interface to accommodate SCCB

- Clock speed fixed at 100 KHz
- No checking/waiting for NACK/ACK
- Only 1 data byte sent in single transmission

Comparison between I2C and SCCB

Features	I²C	SCCB
Bus Lines	SDA (Serial Data), SCL (Serial Clock)	SDA (Serial Data), SCL (Serial Clock)
Addressing	7/10 Bit Addressing	8 Bit Addressing
Supported Clock Speeds	Standard Mode: 100 KHz Fast Mode: 400 KHz	100 KHz (Only supports single speed)
Data Format	8-Bit Data with NACK/ACK	8-Bit Data with Don't Care Bit

I2C Data Transmission Process



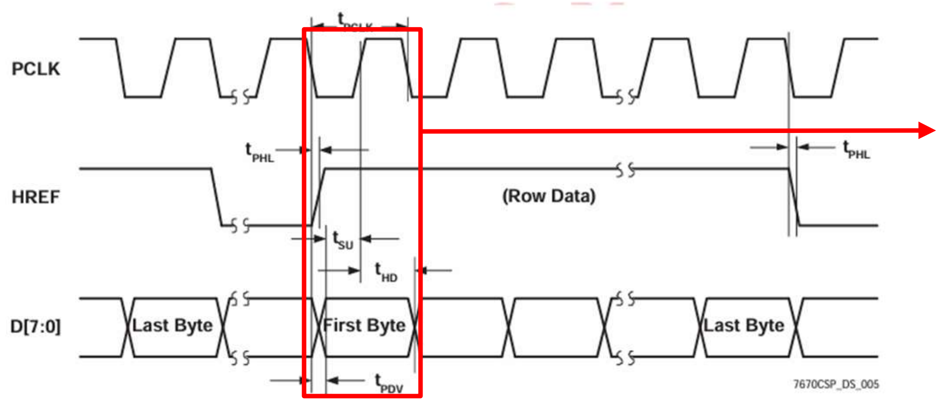
The status of transmission is verified by checking the SFR bits for each I2C task

If any transmission timeout occurs, the error is logged via UART

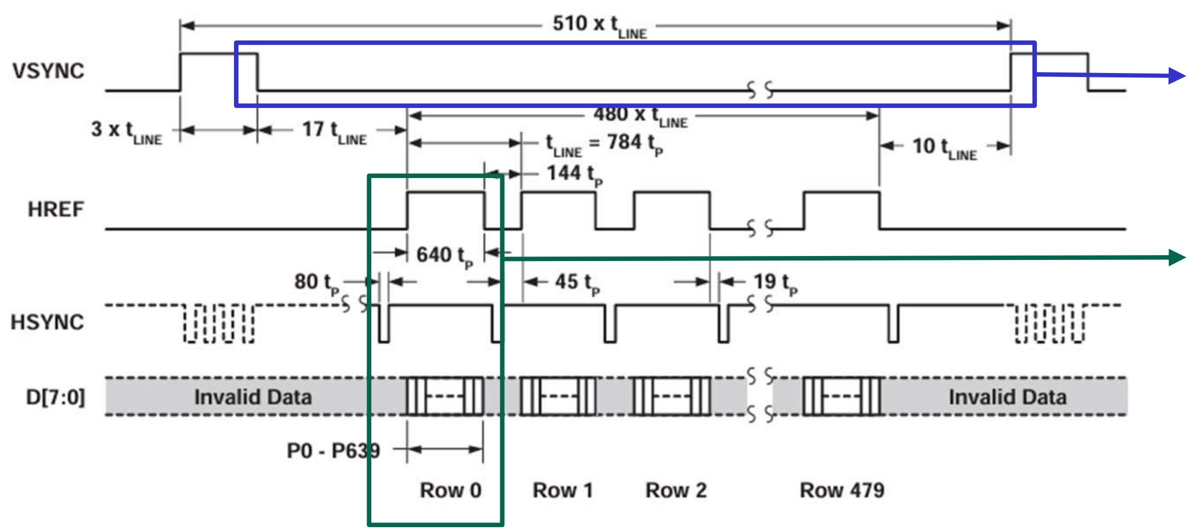
OV7670 Timing Diagram

Once the clock input is provided, we will begin seeing the data flow from pins D7 to D0, as well as the synchronization signals from HREF and VSYNC.

The Figure below shows the expected signals from the OV7670 on successful initialization



- D7 to D0 must be sampled at the **rising edge** of PCLK
- D7 to D0 must be sampled ONLY when HREF is **high.**

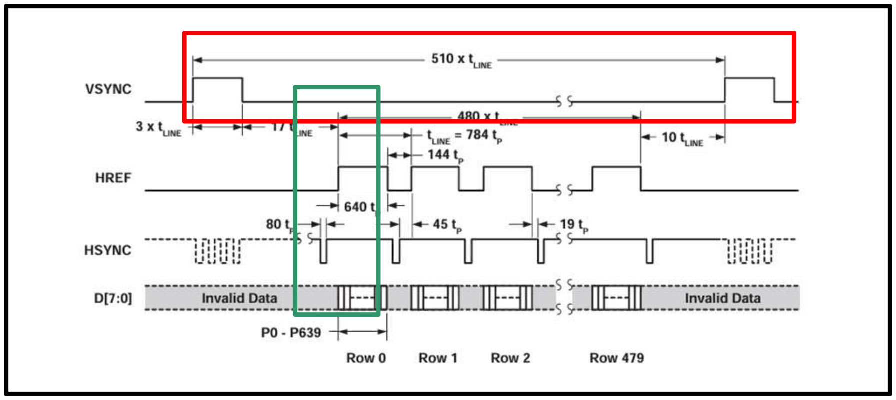


- **Falling edge** of VSYNC indicate start of frame and vice versa.
- This means that an entire frame will be captured during VSYNC **low state**.

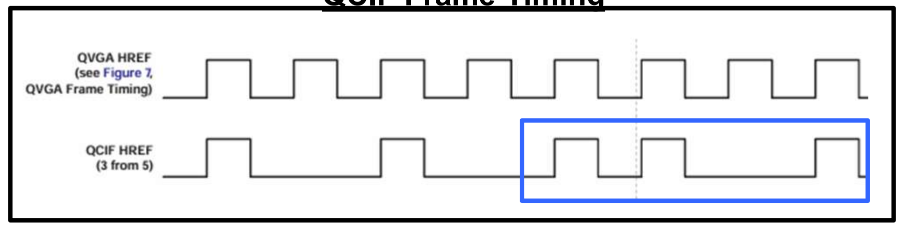
- The **rising edge** of HREF indicates the start of a new row of pixels.
- A complete row of pixels is sampled during the **high state** of HREF, given VSYNC remains **low**.

Verification of Camera Configuration

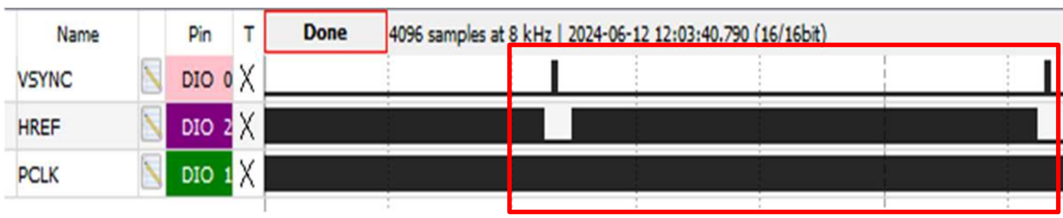
OV7670 Frame Timing



QCIF Frame Timing

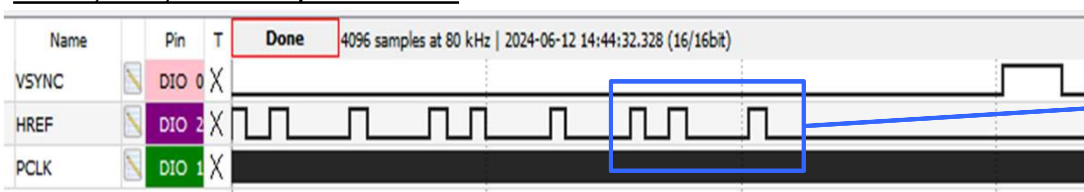


VSYNC,HREF, PCLK Sampled at 8 KHz



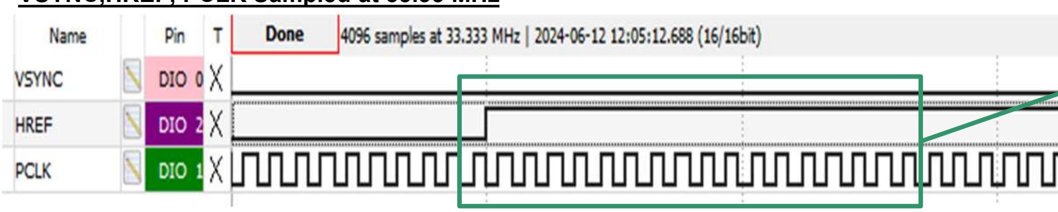
- HREF signals are high when VSYNC is low
- PCLK is free-running

VSYNC,HREF, PCLK Sampled at 80 KHz



- Only 3/5 HREF signals are produced, indicating resolution is QCIF

VSYNC,HREF, PCLK Sampled at 33.33 MHz



- Pattern remains the same when sampled at more than twice PCLK frequency (33 MHz.)

Frame Capture - DCMI Architecture

The Digital Camera Interface (DCMI) which is embedded within the STM32F4 controllers which allows easier connection and transfer of data from a camera module to the MCU.

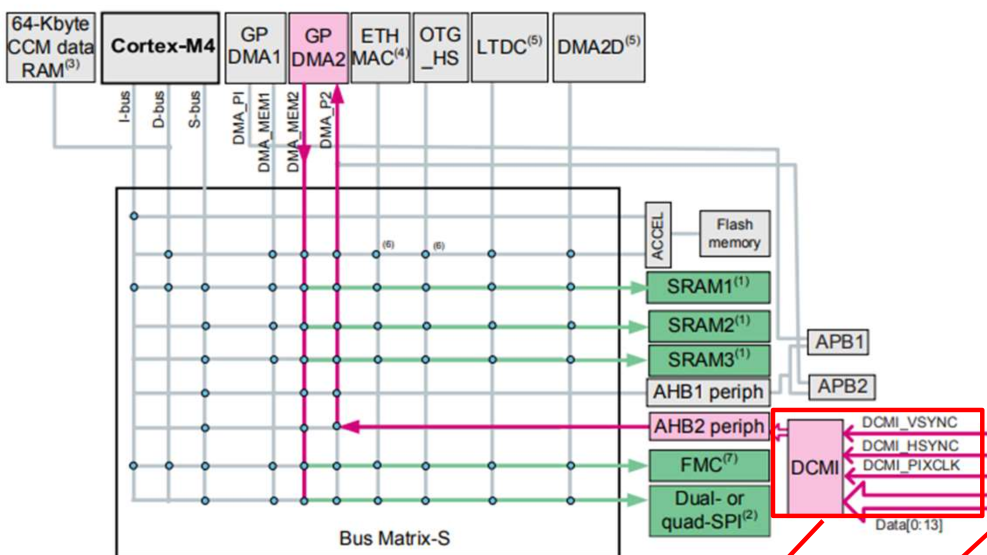
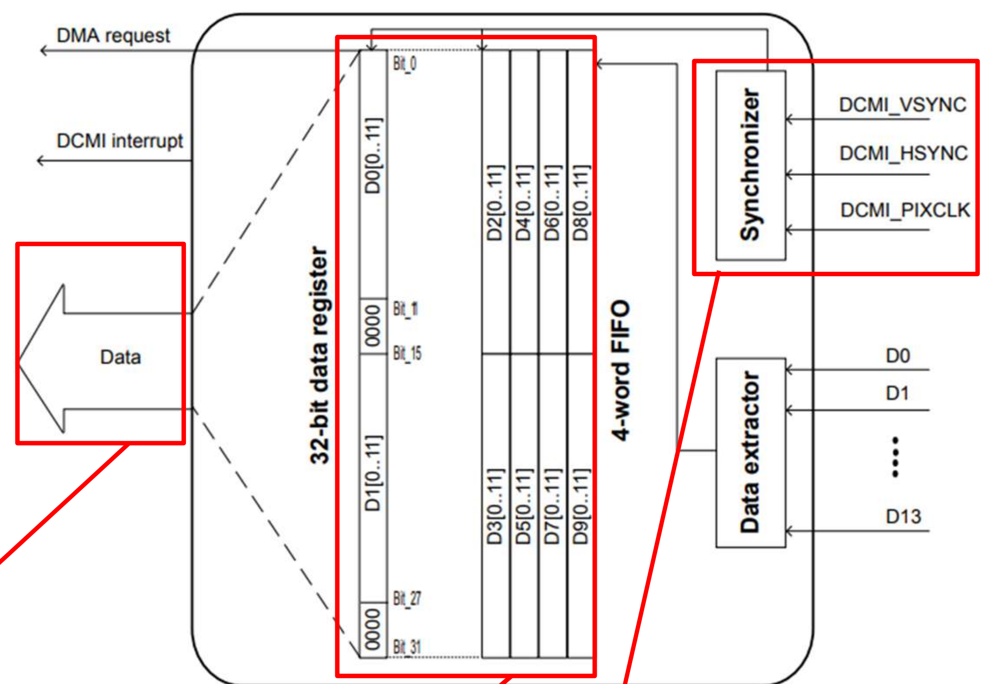


Figure 17. DCMI block diagram



The data and synchronization signal pins are connected to the DCMI pins on the F4 controller

Data from the DCMI data register is transferred to SRAM through Direct Memory Access (DMA).

A 4-word FIFO is used to adapt the transfer rate to the Advanced High-performance Bus (AHB).

The built in synchronizer manages the synchronization of received data based on the input signals and initialized settings.

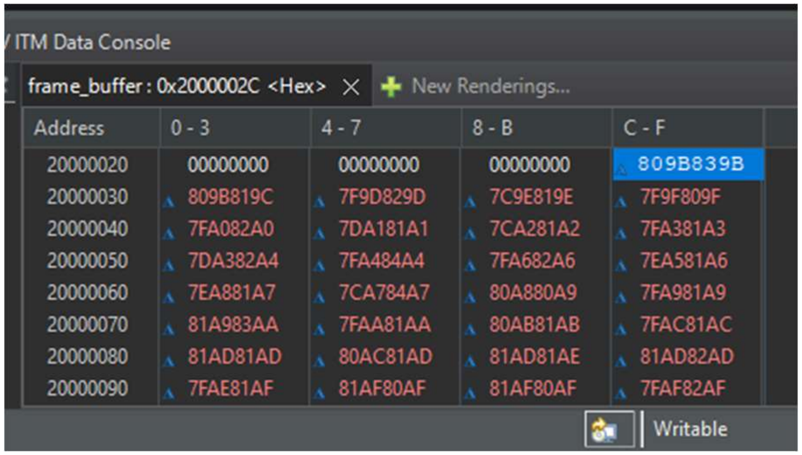
Frame Capture - DCMI Initialization

The DCMI Interface and DMA are initialized using the function DCMI_configure as shown below:

```
DCMI_DeInit();
DCMI_InitStructure.DCMI_CaptureMode = DCMI_CaptureMode_SnapShot;
DCMI_InitStructure.DCMI_ExtendedDataMode = DCMI_ExtendedDataMode_8b;
DCMI_InitStructure.DCMI_CaptureRate = DCMI_CaptureRate_All_Frame;
DCMI_InitStructure.DCMI_PCKPolarity = DCMI_PCKPolarity_Rising;
DCMI_InitStructure.DCMI_HSPolarity = DCMI_HSPolarity_High;
DCMI_InitStructure.DCMI_VSPolarity = DCMI_VSPolarity_Low;
DCMI_InitStructure.DCMI_SynchroMode = DCMI_SynchroMode_Hardware;
DCMI_Init(&DCMI_InitStructure);
DCMI_ITConfig(DCMI_IT_FRAME, ENABLE);
DCMI_ITConfig(DCMI_IT_OVF, ENABLE);
DCMI_ITConfig(DCMI_IT_ERR, ENABLE);

// DMA config
DMA_DeInit(DMA2_Stream1);
DMA_InitStructure.DMA_Channel = DMA_Channel_1;
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) (&DCMI->DR);
DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t) frame_buffer;
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;
DMA_InitStructure.DMA_BufferSize = IMG_ROWS * IMG_COLUMNS;
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
DMA_InitStructure.DMA_Priority = DMA_Priority_High;
DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Enable;
DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
DMA_Init(DMA2_Stream1, &DMA_InitStructure);
DMA_ITConfig(DMA2_Stream1, DMA_IT_TC, ENABLE);
DMA_ITConfig(DMA2_Stream1, DMA_IT_TE, ENABLE);
```

- Capture mode is set to snapshot, we will only be grabbing one frame at a time.
- Data from DCMI Data register will be transferred to our buffer (frame_buffer) using DMA.
- Interrupt generated by DCMI once frame captured and transferred to memory by DMA.

The figure below shows the memory location occupied by frame buffer after a frame is transferred, indicating successful transfer from DCMI DR to buffer.



ITM Data Console

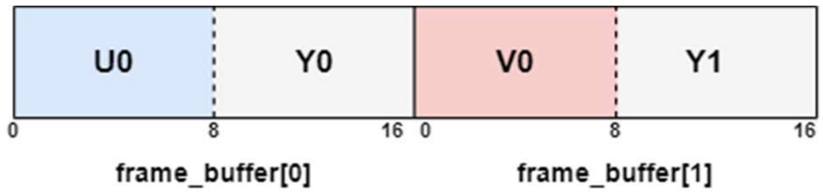
frame_buffer : 0x2000002C <Hex> New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
20000020	00000000	00000000	00000000	809B839B
20000030	809B819C	7F9D829D	7C9E819E	7F9F809F
20000040	7FA082A0	7DA181A1	7CA281A2	7FA381A3
20000050	7DA382A4	7FA484A4	7FA682A6	7EA581A6
20000060	7EA881A7	7CA784A7	80A880A9	7FA981A9
20000070	81A983AA	7FAA81AA	80AB81AB	7FAC81AC
20000080	81AD81AD	80AC81AD	81AD81AE	81AD82AD
20000090	7FAE81AF	81AF80AF	81AF80AF	7FAF82AF

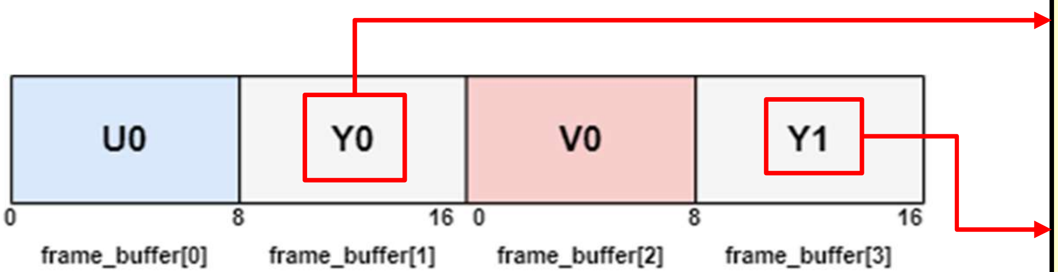
Writable

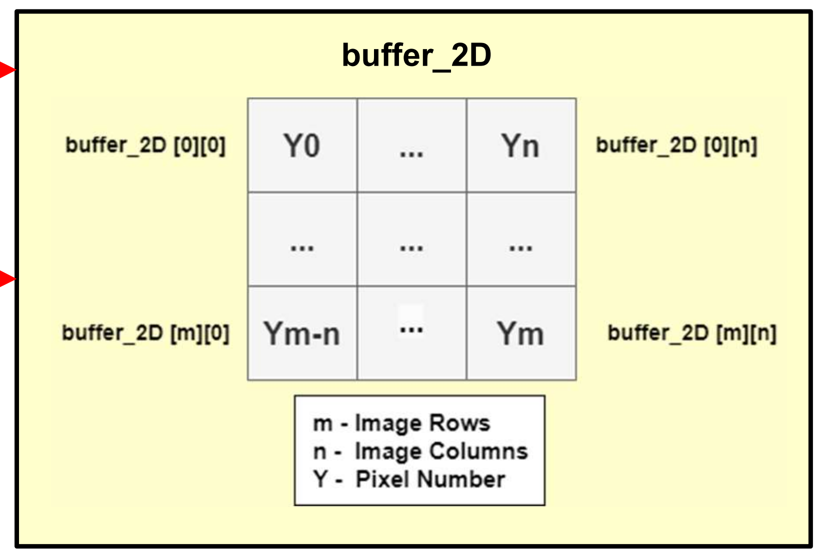
Frame Transmission – Greyscale Conversion

Once a frame is captured, it is stored in the **frame_buffer** in the following format:



Greyscale conversion is needed to allow **faster streaming**, and for processing as most algorithms work with grayscale images. It is done by the **YUVtoGrey_2D** function.



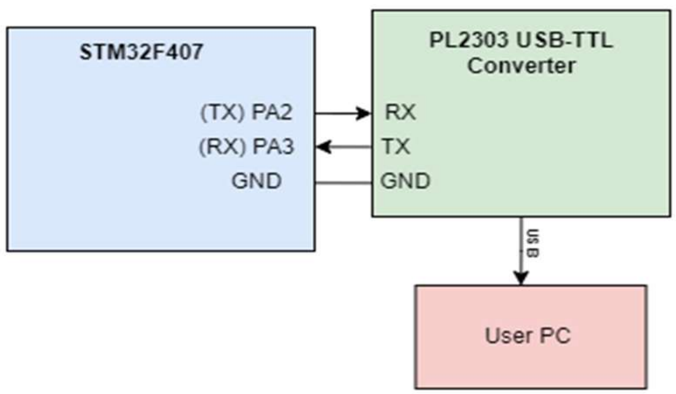


- The frame buffer is **casted to 8-bit representation** for easier extraction.
- The luminance values are extracted from frame_buffer and stored in 2D representation in buffer_2D.

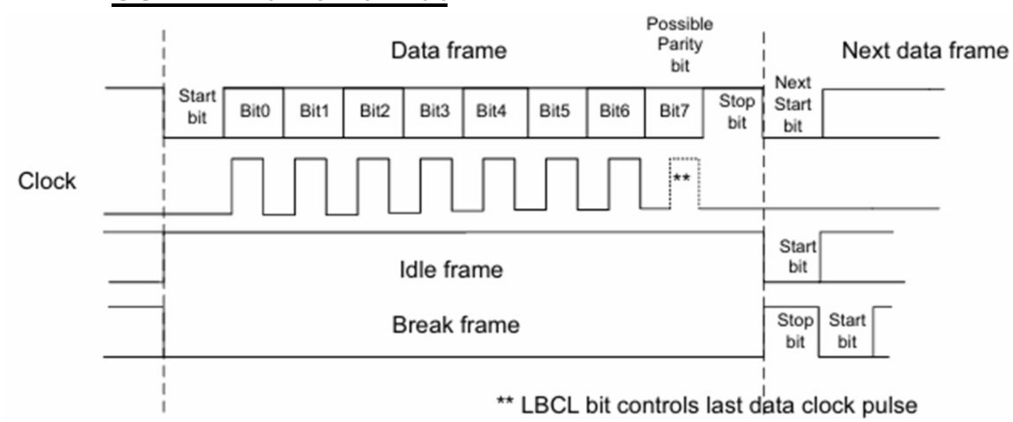
Frame Transmission – USART Interface

To stream frames, the MCU's USART peripheral is used alongside a USB to TTL converter that is used to interface to the PC

USART and PC Interfacing



USART Frame Format



USART Peripheral Settings

Parameter	Settings
Baud Rate	230400
Word Length	8 Bits
Stop Bits	1 Bit
Parity	None
RX Interrupt	Enabled

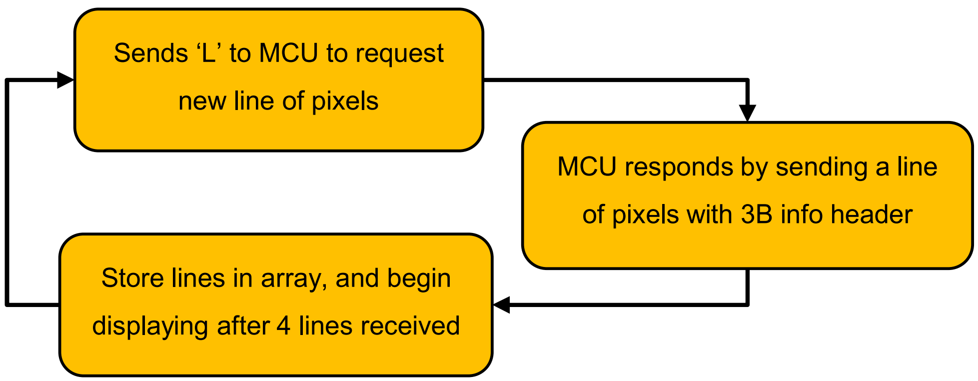
USART Write/Read Functions Used

Function	Description
uart2_write	Transmit raw decimal values between 0-255
Serial_log	Transmit character string for debugging/error log
Serial_logi	Transmit decimals as character string
Serial_read	Read data received by USART

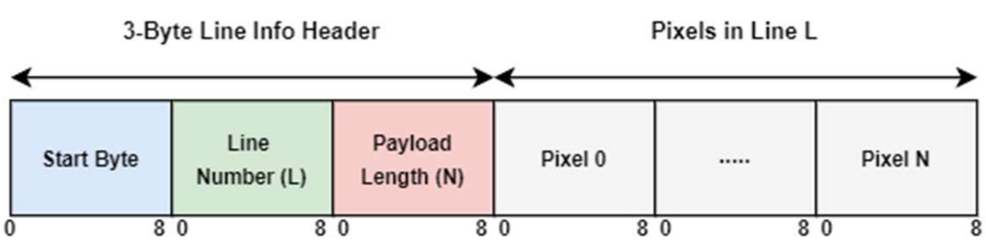
Frame Transmission – Python Program

On PC side, a Python program written by Prof. Fabian Kung [10] was used to **receive frame data line-by-line** and display it using PyGame.

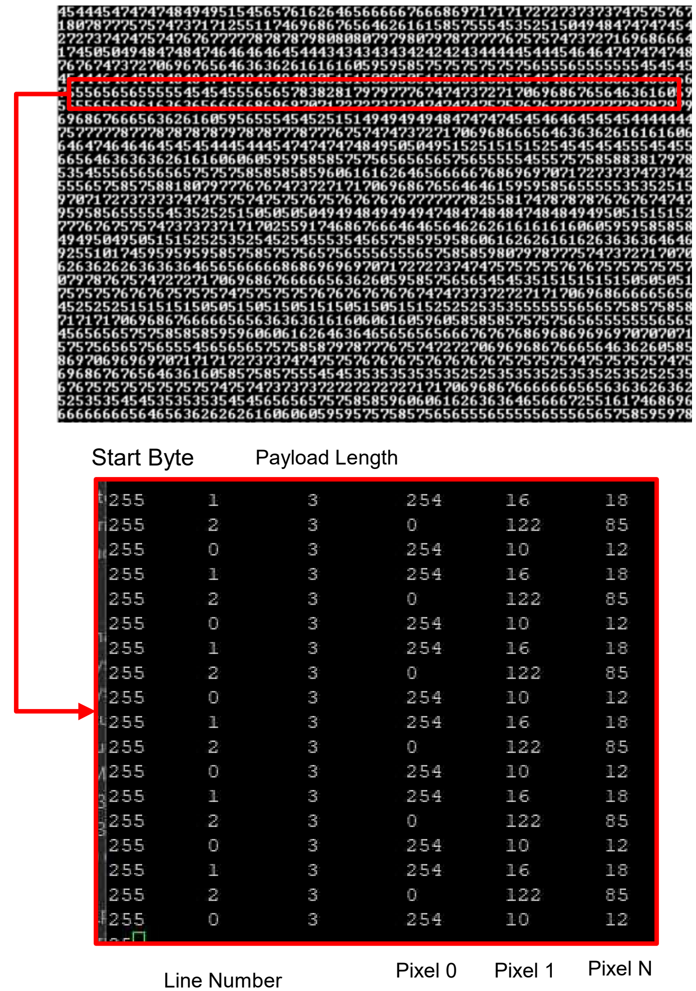
Python Program Flow



Line Transmission Format



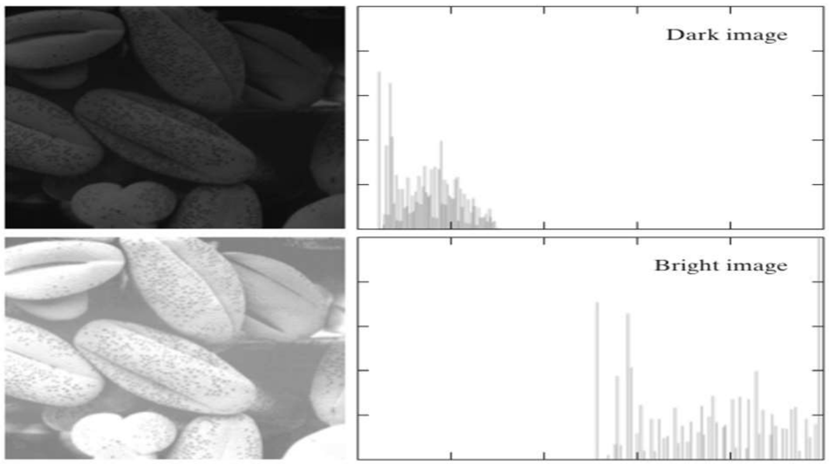
Example of Line Transmission



CV Algorithm - Histogram

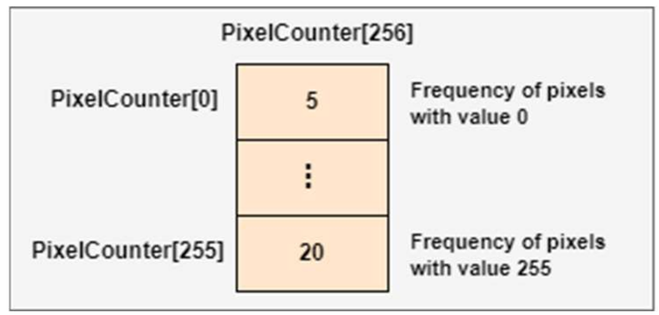
The simplest CV algorithm implemented is the Histogram which computes the frequency of different gray levels within the image.

Example of Image Histogram Plot



Gray Level Counter

An array is implemented as a counter to calculate the frequency of each pixel gray level



Implemented in **YUVtoGrey_2D**

Brightness of surrounding can be determined by analyzing distribution of pixel intensity in Histogram.

High μ = Bright Surrounding
Low μ = Dark Surrounding

Threshold can be calculated through **μ+σ**

Used to filter out lower intensity pixels, and only retain pixels within high gray level range of image

Statistics from Histogram

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=0}^{255}(i-\mu)^2 \times f(\mathrm{i})}$$

$$\mu = \frac{1}{N}\sum_{i=0}^{255} i \times f(\mathrm{i})$$

μ - Mean
σ - Standard Deviation
i - Gray level
f(*i*) - Freq of *i*
N - Total No. Pixels

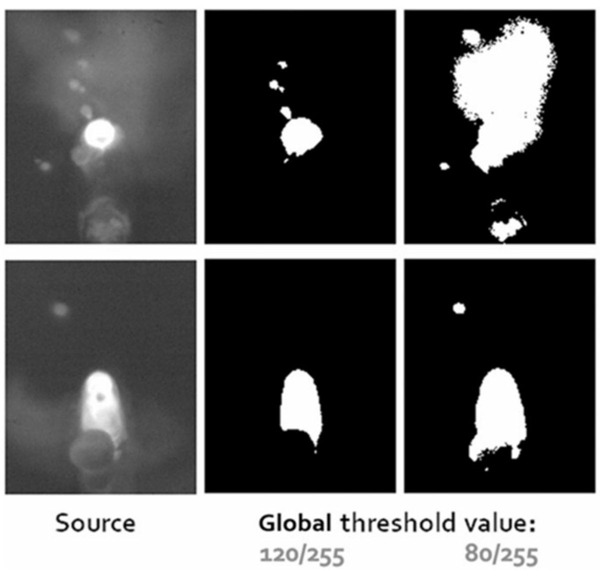
Implemented in **Histogram(Counter)**

CV Algorithm – Bright Spot Detection

Bright Spot Detection or BSD Algorithm works by isolating pixels with higher gray levels from those that with lower levels.

Global Thresholding

A single threshold, T is used for all pixels within the image.



Pixels are checked and turned white/black depending on its value and threshold.

If **$Pixel_N$ > T** ---- $Pixel_N$ = **255 (White)**

If **$Pixel_N$ > T** ---- $Pixel_N$ = **0 (Black)**

Adaptive Global Thresholding

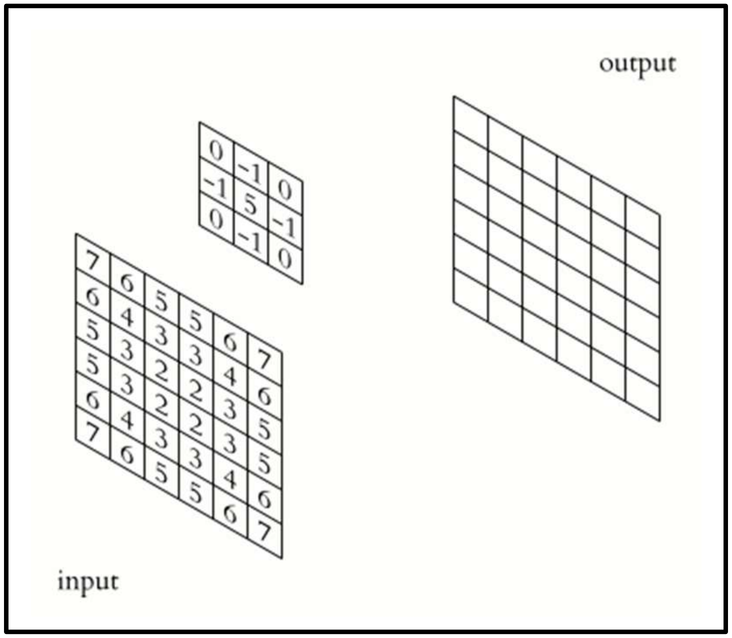
Threshold T is computed on an image-to-image basis using **Histogram** function.

Threshold, T = $\mu+\sigma$

The computed threshold is used to filter out bright spots within the image

Kernel Convolution

Kernel Convolution



Kernel Convolution involves **sliding the kernel** over **each pixel** in the image and multiplying the center pixel and its neighbors by the values within the kernel.

Image Kernels

1	2	1
2	4	2
1	2	1

Gaussian Kernel

-1	0	1
-2	0	2
-1	0	1

Sobel X Kernel

-1	-2	-1
0	0	0
1	2	1

Sobel Y Kernel

Mathematical Model of Kernel Convolution

$$G(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} I(x+i, y+j) \cdot K(i+1, j+1) \quad (3.3)$$

- $G(x, y)$ is the output image
- $I(x, y)$ is the input image
- $K(i, j)$ *is the* 3x3 kernel
- (x, y) *is the coordinates of a specific pixel*
- i, j *range from* -1 *to* 1, *covering the 3x3 neighbourhood of pixels*

CV Algorithm – Sobel Edge Detection

The Sobel Edge Detection algorithm **uses kernel convolution** to compute the spatial gradient of an image.

Edges occur in images when there is a steep intensity gradient

buffer_2D

23	55	87	93
11	43	46	78
54	66	69	9
12	14	18	45

*

-1	0	1
-2	0	2
-1	0	1

Sobel X Kernel

=

Gx

Highlights Horizontal Edges

buffer_2D

23	55	87	93
11	43	46	78
54	66	69	9
12	14	18	45

*

-1	-2	-1
0	0	0
1	2	1

Sobel Y Kernel

=

Gy

Highlights Vertical Edges

The Gx and Gy components are combined to obtain the magnitude of gradient (highlights edges in all directions)

$$M(i,j) = \sqrt{(\,G_X(i,j)\,)^2 + (\,G_Y(i,j)\,)^2}$$

Kernel Convolution in SobelEdgeDetection Function

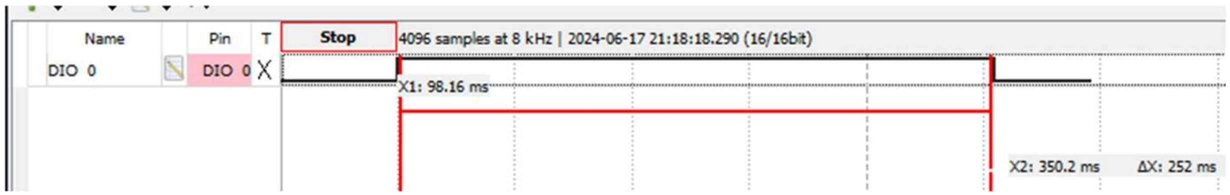
```
for (int i = 1; i < IMG_ROWS - 1; i++) {
    for (int j = 1; j < IMG_COLUMNS - 1; j++) {
        gx = (-1 * input[i - 1][j - 1]) + (1 * input[i - 1][j + 1])
           + (-2 * input[i][j - 1])     + (2 * input[i][j + 1])
           + (-1 * input[i + 1][j - 1]) + (1 * input[i + 1][j + 1]);

        gy = (-1 * input[i - 1][j - 1]) + (-2 * input[i - 1][j]) + (-1 * input[i - 1][j + 1])
           + (1 * input[i + 1][j - 1])  + (2 * input[i + 1][j])  + (1 * input[i + 1][j + 1]);

        magnitude = sqr_root((gx * gx) + (gy * gy));
```

- The Gx, Gy and magnitude are computed on a **pixel-to-pixel basis**
- This eliminates the need for multiple buffers for Gx, Gy and M
- The final output is the magnitude which is stored in **one buffer**

Histogram is used to compute the threshold for the Sobel magnitude and **thresholding is done for a clearer image of edges**.

Analysis of Results

Timing measurements are done by **toggling a test pin** before and after the process to be timed and measuring the **pulse width** using the Digilent Logic Analyzer.



The time taken for executing general tasks, as well as image processing algorithms is shown below:

Task	Time Taken
Configuring OV7670 registers	65.39 s
Capturing and storing frame in buffer	252.00 ms
Transmitting unprocessed frame (230400BD)	1.25 s

Image Processing Algorithm	Time Taken
Histogram	2.00 ms
Bright Spot Detection (Thresholding)	50.00 ms
Kernel Convolution (Gaussian Blur)	1.05 s
Sobel Edge Detection	3.09 s

Analysis of Results

Unprocessed Streaming

Test Image 1

Grayscale Output

- The CV Module is able to capture the test image clearly with no visible distortion or noise.
- No visible contrast mismatch or blurring of frame.

Bright Spot Detection

Test Image 2

BSD with Adaptive Thresholding

- Bright spots in image induced by LED lights can be captured accurately.
- No noise is falsely detected as bright spots.

Analysis of Results

<u>Sobel Edge Detection</u>

Test Image 1

Sobel Edge Detection

Sobel Edge Detection with Thresholding

- Edges within image are well-defined
- Prominent features of image are retained.
- Background is clean with minimal noise.

Analysis of Results

Sobel Edge Detection (More images)