BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING

FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

DESIGN AND IMPLEMENTATION OF DISPLAY SNIFFER ON EMBEDDED TARGETS

NÁVRH A IMPLEMENTACE NÁSTROJE PRO ANALÝZU OBRAZOVÝCH DAT VESTAVĚNÝCH SYSTÉMŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR   Bc. Samuel Lipták
AUTOR PRÁCE

SUPERVISOR   doc. Ing. Jiří Krejsa, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2023

Assignment Master's Thesis

Institut:	Institute of Solid Mechanics, Mechatronics and Biomechanics
Student:	**Bc. Samuel Lipták**
Degree programm:	Mechatronics
Branch:	no specialisation
Supervisor:	**doc. Ing. Jiří Krejsa, Ph.D.**
Academic year:	2023/24

As provided for by the Act No. 111/98 Coll. on higher education institutions and the BUT Study and Examination Regulations, the director of the Institute hereby assigns the following topic of Master's Thesis:

Design and implementation of display sniffer on embedded targets

Brief Description:

An integral part of the automated testing of embedded systems containing a graphical interface is the verification of the compliance of the graphical design with the real implementation. A commonly used technique is the use of camera systems, but this is calibration and maintenance intensive and inevitably leads to degradation of image data. One way to replace these systems is to use specialized tools that analyze directly the data stream on the bus between the processor and the display. The essence of this thesis is the design and implementation of such a tool on the PYNQ platform.

Master's Thesis goals:

1. Research commonly used interfaces between display and processor in embedded systems
2. Design and implement reception of image data using PYNQ platform
3. Design and implement the interface for propagation of image data to test framework.

Recommended bibliography:

Cem Ünsalan, Bora Tar: Digital System Design with FPGA: Implementation Using Verilog and VHDL, McGraw Hill, 2017

Alégroth, E., Feldt, R. On the long-term use of visual gui testing in industrial practice: a case study. Empir Software Eng 22, 2017, pp. 2937–2971.

Deadline for submission Master's Thesis is given by the Schedule of the Academic year 2023/24

In Brno,

L. S.

_____   _____

prof. Ing. Jindřich Petruška, CSc.   doc. Ing. Jiří Hlinka, Ph.D.
Director of the Institute   FME dean

Abstrakt

Táto diplomová práca sa zaoberá overením funkcionality grafického rozhrania v integrovaných systémoch. Súčasné prístupy využívajúce kamerové systémy pre optickú kontrolu sú charakterizované nedostatočnou spoľahlivosťou, vysokými nákladmi, náročnou údržbou a náročnosťou na priestorové umiestnenie.

Cieľom tejto práce je analyzovať a navrhnúť nový prístup k získavaniu grafických dát, ktorý bude založený na spoľahlivej technológii. Konkrétne riešenie využíva technológiu FPGA (Field-Programmable Gate Array) a celý systém je implementovaný na vývojovej platforme PYNQ. Táto platforma zároveň obsahuje server s API, čo umožňuje jednoduchší prístup k získaným dátam.

Výsledkom tejto práce je nová metóda overenia funkcionality grafického rozhrania vstavaných systémov, ktorá bude spĺňať požadované kritériá spoľahlivosti a účinnosti. Takýto prístup by mohol nájsť uplatnenie v priemysle a prispieť k zlepšeniu kvality a efektívnosti kontroly kvality integrovaných systémov.

Summary

This thesis deals with the verification of the functionality of the graphical interface in embedded systems. Current approaches using camera systems for optical inspection are characterised by a lack of reliability, high cost, maintenance difficulties and spatial challenges.

The aim of this work is to analyse and propose a new approach to graphical data acquisition, based on a reliable technology. The specific solution uses FPGA (Field-Programmable Gate Array) technology and the whole system is implemented on the PYNQ development platform. This platform also includes a server with an API, which allows easier access to the acquired data.

The result of this work is a new verification method of the graphical interface of embedded systems, which will meet the required reliability and efficiency criteria. Such an approach may find application in industry and contribute to improving the quality and efficiency of quality control of embedded systems.

Kľúčové slová

grafické rozhranie, vstavané systémy, FPGA, PYNQ, kontrola kvality

Keywords

graphical interface, embedded systems, FPGA, PYNQ, quality control

Bibliographic citation

LIPTÁK, S. *Design and implementation of display sniffer on embedded targets.* Brno: Brno University of Technology, Faculty of Mechanical Engineering, 2024. 70 pages, Master's thesis supervisor: doc. Ing Jiří Krejsa, Ph.D..

I declare that this thesis is my original work, I prepared it independently under the guidance of doc. Ing Jiří Krejsa, Ph.D. and using the cited literature.

Samuel Lipták

Brno . . . .

Contents

Introduction

Quality control is an important part of embedded systems development. We can test electrical, mechanical and software properties as well as the overall functionality of such a system. This thesis primarily focuses on the quality control of graphical components, with a primary emphasis on displays within the system.

Displays and other graphical interfaces play a crucial role of providing user with information, instructions or warnings. For this reason, correct functioning of graphical interfaces must be assured by intensive quality control.

Gathering data for these purposes can prove rather difficult for various reasons. If testing includes monitoring instructions for the display, the testing environment might not catch errors caused by faulty display driver. Another solution could be to monitor graphical output directly by external camera, but this method is complex, not always reliable and very space-consuming. Another solution would be to monitor dataflow which describes the exact data that are sent to display pixels. This dataflow is extremely fast and contains a large amount of data compared to instructions sent by microcontroller.

The aim of this work is to develop a system that will ensure reliable and accurate image data that is acquired from the communication between the display controller and the display itself. Based on this requirement, the chosen medium for obtaining the necessary data is FPGA technology, specifically the PYNQ Z-2 development board. The software for the acquisition, processing and subsequent distribution of the image data based on DVI-D protocol is designed and implemented directly on PYNQ Z-2.

1 Problem analysis

This thesis is focusing on three tasks:

1. Research commonly used display-to-processor interfaces for embedded display peripherals

2. Design and implement image data acquisition using a PYNQ platform

3. Design and implement a suitable interface for passing image data to the test framework

The frame gathering system should be superior to other used methods in speed, quality of gathered data and reliability. Speed is measured in FPS (frames per second). This unit describes how many frames can display refresh per second. Quality of data compares the theoretical real and received data and evaluate their similarity. The replicability of the above units is evaluated by reliability. Following table sets a minimum that the final system must handle.

Res	FPS
100x100	15
500x400	14
960x544	12
1280x720	10

Table 1.1: Minimum requirements for final system [source: author]

1.1 Quality control in embedded development

Quality control in embedded development is a crucial mechanism for ensuring functionality of final embedded solution. The process involves:

- Defining clear conditions which must be met

- Create testing environment

- Use testing environment to ensure the system satisfies defined conditions

The objective is to identify any problems which have negative impact on the effectivity or overall functionality of the final system. Mechanical and electrical tests are not included in this thesis.

By using tools such as testing frameworks which manually or automatically perform software tests on developed system, the user can evaluate system's performance and quality. These tests consist of testing framework inserting known inputs to a system and then comparing the expected outputs (defined by conditions) to real outputs.
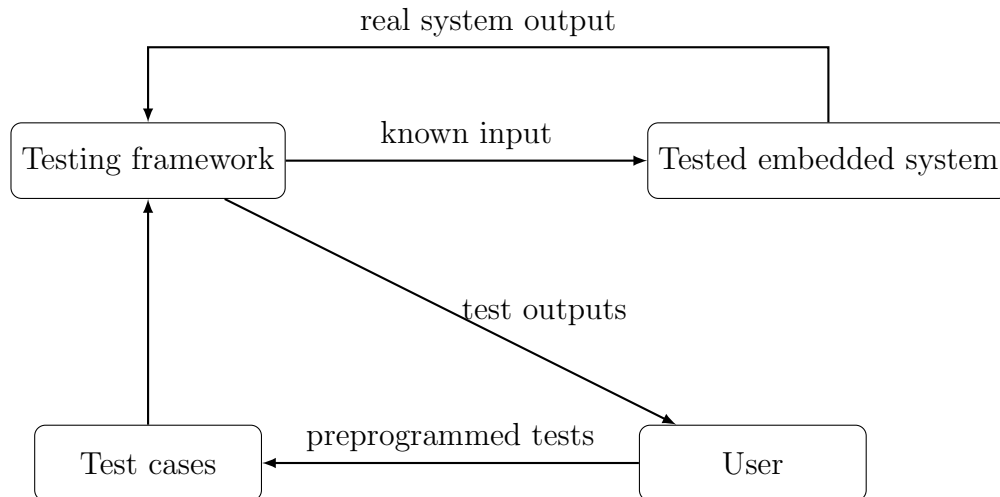

Figure 1.1: Simple diagram of quality control loop [source: author]

Process displayed in figure 1.1 consists of few simple steps:

1. User defines test cases based on tested conditions

2. Testing framework is connected to tested embedded system

3. Testing framework starts test cases with expected outputs

4. Real outputs are compared to expected outputs

5. Results are provided to user who can update test cases or fix errors in embedded system

This process is straightforward for simple digital or basic analog signals. If user wants to test more advanced peripherals such as audio, high frequency signals or graphical outputs, advanced signal processing systems must be used to process the *real system output*.

The processing must be non-destructive meaning that the details and quality must not be lost due to destructive compression or other signal simplification methods.

1.2 Display quality control

Displays are one of the most important graphical peripherals in embedded systems. Displays are used for informing, warning or instructing people. Graphical outputs are used in a plethora of applications, and thus, quality control of such devices is a relevant topic.

In order to design a data gathering mechanism for testing framework, a display type must be defined.
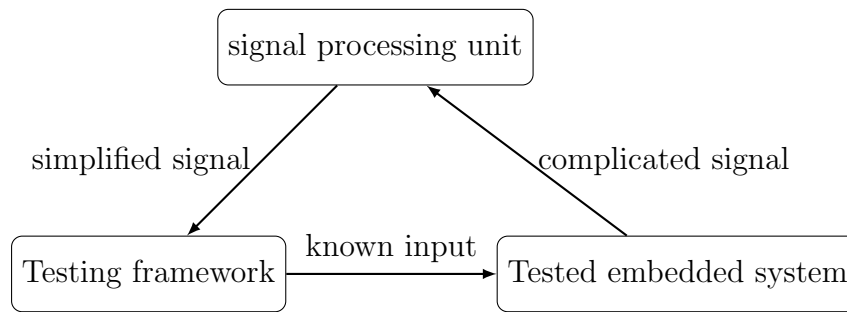
Figure 1.2: Signal processing unit position in data gathering loop [source: author]

1.2.1 Technologies of display devices

Display devices can implement different technologies for outputting graphical data. Most common display technologies are:

- LCD

- OLED

- LED

- E-ink

LCD displays

LCD or **Liquid Crystal Display** are widely used in devices such as TVs, monitors or smartphones.

These displays utilize a unique technology that involves liquid crystals positioned between layers of glass or plastic. The liquid crystals, which are rod-shaped molecules, can twist and untwist in response to an applied electric current. [1] LCD can display monochrome as well as color images.

Figure 1.3: Monochrome LCD display [4]

This property allows for precise control over the passage of light through individual pixels. Color is achieved by employing colour filters and dividing pixels into subpixels of red, green, and blue. [1]
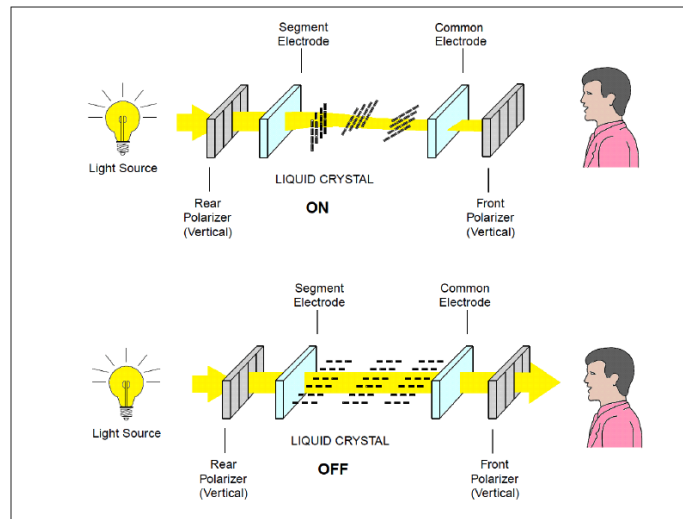
Figure 1.4: LCD working principle [3]

OLED displays

OLED or **Organic Light Emitting Diode** displays are similar to LCD technology. OLED displays are composed of light-emitting pixels thus not requiring back-illumination as LCD.
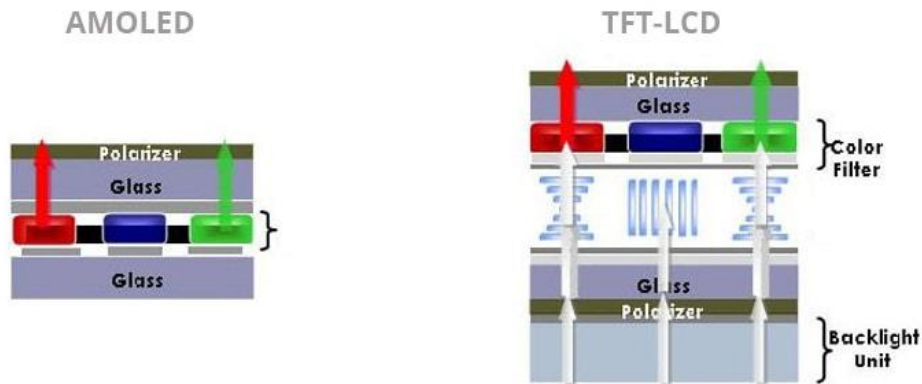

Figure 1.5: Difference between LED and (AM)OLED display [5]

AMOLED is a branch of OLED technology. The working principle is the same but AMOLED uses more advanced colour control mechanisms. OLEDs produce images by lighting or dimming specific RGB pixels similar to classic light emitting diode, hence, image data is also an analog signal describing the intensity of each RGB component. [2]
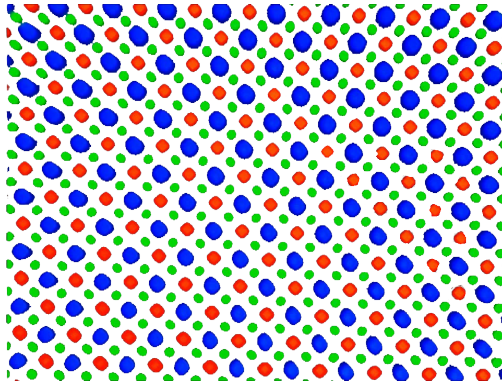
Figure 1.6: One of many possible structures of OLED display under microscope [6]

LED displays

LED displays use **light emitting diodes** to display information. This type of displays is mostly used if simple information is to be shown such as numbers and basic letters. LED displays are also used in conjunction with LCD displays as back illumination.

LED displays are simple devices controlled by digital inputs and can be used to display colour of monochrome images especially on large formats.
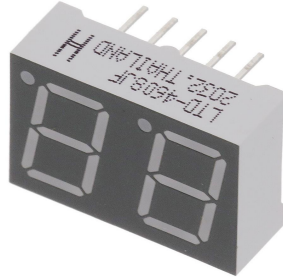
Figure 1.7: 8 segment monochrome LED display [7]

E-ink displays

E-ink displays use very different display technology compared to LCD or OLED displays. E-ink or electronic paper use microcapsules containing charged black particles suspended in white or clear fluid.

These particles can be triggered and moved to top of the capsule displaying black pixel. These displays have extremely low power consumption and are bistable, meaning that energy is not needed to maintain displayed content. [8]

Cross-section of E-ink display is shown in figure 1.8
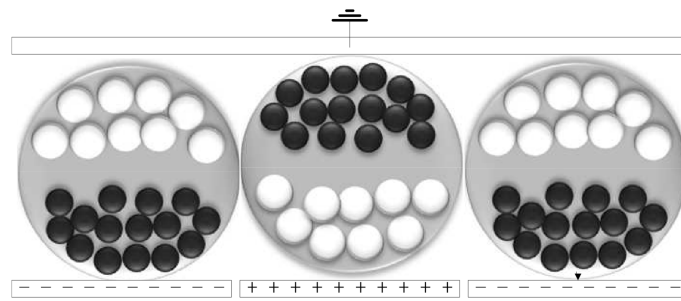
Figure 1.8: E-ink capsules [8]

1.2.2   Display dataflow

Controlling any display can be achieved by various means.

Dataflow in this thesis is the communication between user programmed controller and the display graphic controller which can be either part of embedded system or a part of display as described in figure 1.9.

The difference between command dataflow and frame dataflow is discussed in chapters 2.2 and 2.3.
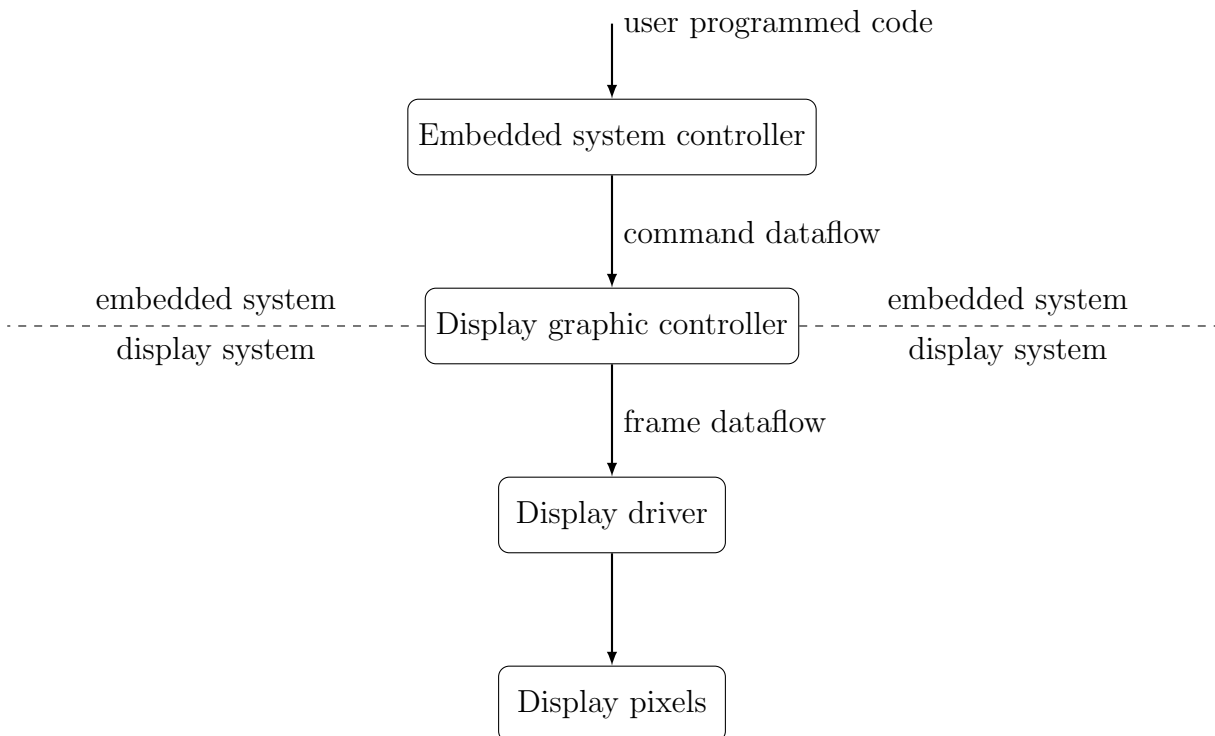
Figure 1.9: Simple diagram of display dataflow [source: author]

Frames or pixel data are stored as bytes or bits. If the display is monochrome, the value of a pixel can be stored as 1 or 0 in a single bit. If a display has a colour screen, the value

of each pixel must be represented by multiple values.

Commonly used is the RGB format which stores the *intensity* of each pixel as n bit number. The number of bits is determined by hardware and software capabilities of display driver. Commonly used is 16-bit, 18-bit or 24-bit RGB format. This means that value of each color is stored in this 16,18 or 24 bit number.

Figure 1.10 represents 24-bit color format.
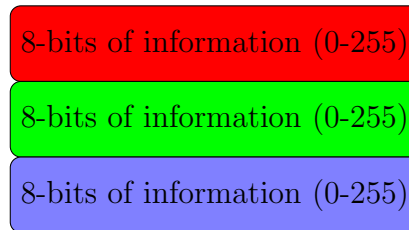
Figure 1.10: Single 24-bit pixel color value storage [source: author]

The ratio in which the colours are stored is defined in the name of a format. For example **RGB565** tells that the red value is represented by 5 bits, green value by 6 bits and blue by 5 bits, together combined in 16 bit (2 bytes) number. RGB888 stores all values equally $(3*8)$ in 24-bit (3 bytes) number.

Serial protocols

Serial protocols use the *serialization* of data which are then sent as a string of bits according to chosen protocol.

Most notable and commonly used serial protocols are:

- UART

- IIC (I^2C)

- SPI

- CAN

Serial protocols are usually more connection-efficient requiring only few physical connections compared to parallel protocols. Serialization of data causes significantly slower data transfer since only one bit can be transferred each clock cycle.

Serialization of data is shown in simplified figure 1.11. As shown in the figure serial data is 24 times slower (for RGB888 color format). Serial protocols are mostly used for monochrome displays where only ones and zeros are transferred thus every pixel is represented by one bit, or if low FPS of the color display is sufficient.

Parallel protocols

Parallel protocols are designed to maximize dataflow. To achieve this, sent data are distributed into multiple data lines. The distribution can differ. For example display driver *ST7920* can be set to accept 4-bit parallel interface where a 8-bit information is split into two 4-bit numbers sent one after another. ST7920 can also be set to accept 8-bit parallel interface and 8 datalines are needed to transfer all 8-bit at the same time. [9] Parallel data sending is shown in simplified figure 1.11.
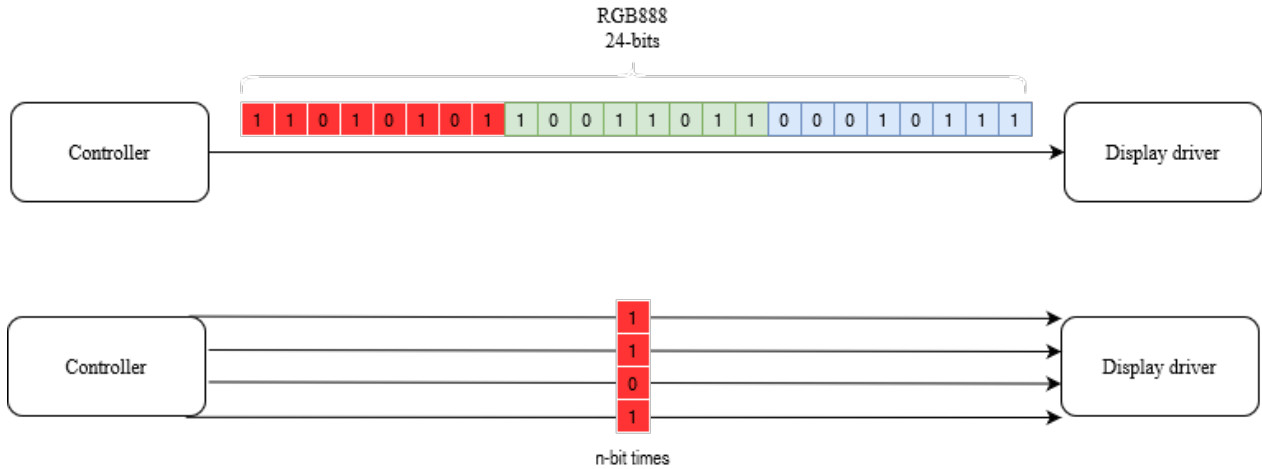
Figure 1.11: Serial and parallel dataflow example [source: author]

Display parallel interface (DPI)

Signal name	Used for
Clock	Synchronizing all transferred data, signalizes to move to next pixel
Data enable	Signalizing valid pixel data
H-sync	Signalizing driver to move to new line
V-sync	Signalizing driver frame ended
Red [7:0]	8 signal lines defining value of red color (0-255)
Green [7:0]	8 signal lines defining value of green color (0-255)
Blue [7:0]	8 signal lines defining value of blue color (0-255)

Table 1.2: Signals transferred by display parallel interface [source: author]

Display parallel interface is a type of parallel protocol where data is sent via n datalines and 4 synchronization signals. Number n is defined by color format. For example as previously discussed 24-bit color format will require 24 datalines. All signals that DPI protocol transferring RGB888 color format uses is summarized in table 1.2.

Given timing diagrams show basic functioning of DPI protocol for better understanding.
Figure 1.12 shows how display receives information about pixel value. Each clock cycle
a value is assigned to red, green and blue value. These values are written to display only
when data enable signal is active. H-sync and V-sync signals are static while pixel values are
transferred.
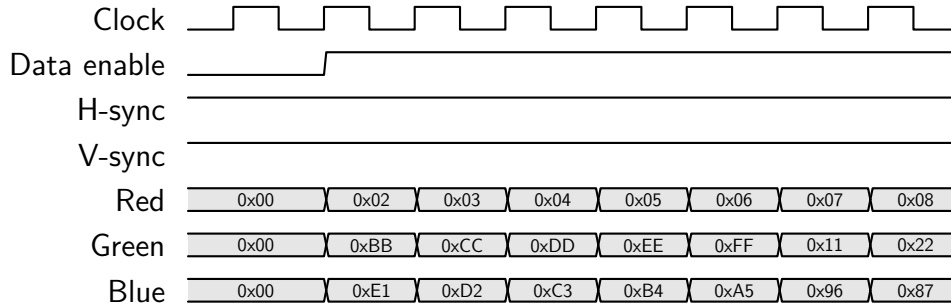
Figure 1.12: Timing diagram of pixel values information [source: author]

Figure 1.13 shows how display receives information to move to another line. Low value
of data enable signalizes that display will receive no more pixel data.
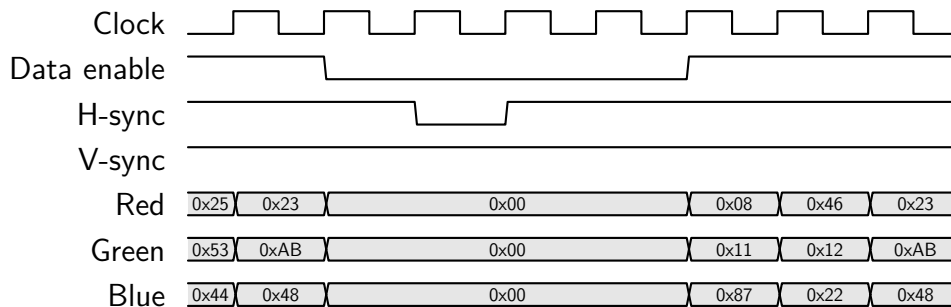
Figure 1.13: Timing diagram of new line [source: author]
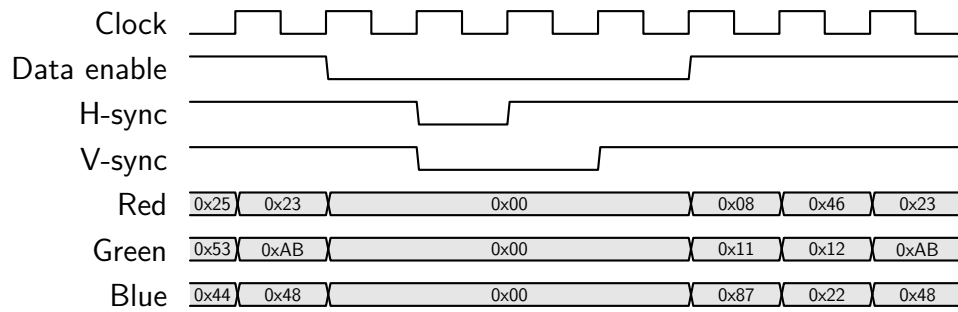
Figure 1.14: Timing diagram of end of frame [source: author]

Figure 1.14 shows how display receives information that the whole frame has ended. The display driver will move the pixel pointer to the beginning of a whole frame.

Figures are simplified, however, real timing functioning is the same.

2 Image data acquiring

The *real system output* as defined in figure 1.1 can be acquired by a plethora of methods. Following section discusses the technology behind each method, its advantages and disadvantages.

2.1 Photographic equipment application

The easiest method to implement is to use photographic equipment. Digital camera interfaces are present in any modern computing unit and passing data to testing framework is very simply implemented, especially, if testing framework is based on popular programming language such as Python or Rust. Camera system can be integrated to the testing process by simply capturing the produced image by the display.
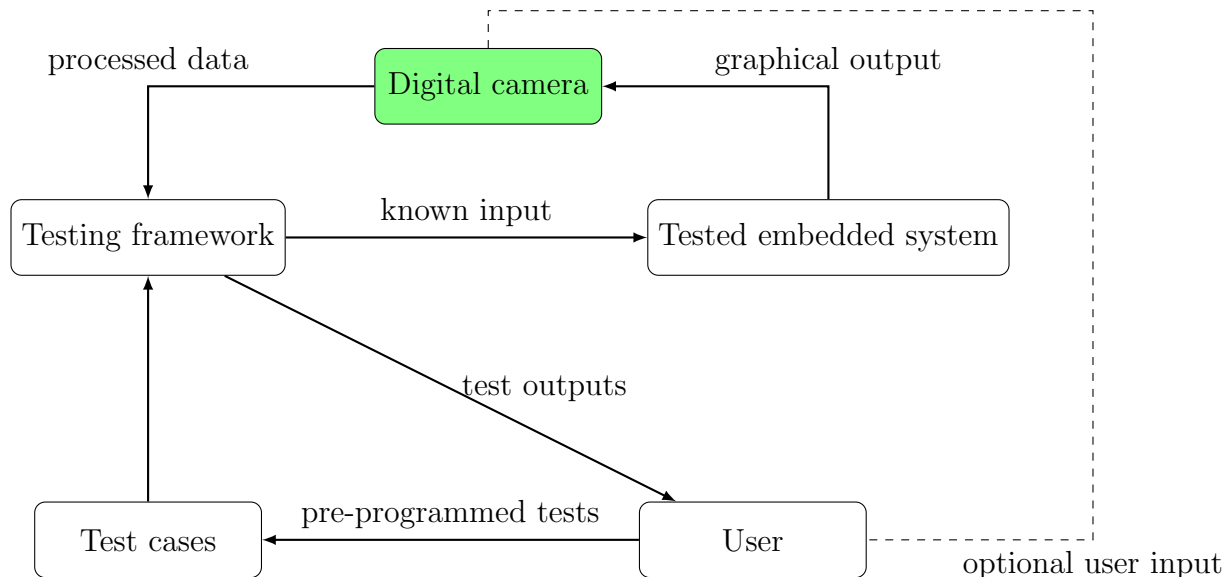
Figure 2.1: Diagram of capturing data by camera [source: author]

2.1.1 Camera system

Camera system, in context of this thesis, consists of a camera sensor and camera lens. Both are crucial when dealing with accurate data aggregation.

Camera sensor

Camera sensors can be split into two basic technologies:

- CCD

- CMOS

Both types serve the same purpose, to receive and process light particles coming from external source.

All sensors are composed of light sensitive semiconductors (photodiodes). Both sensor types need additional components such as amplifiers, filters or other components which are used to convert analog value of received light particles to digital value which is subsequently used as image data.

CMOS

CMOS sensor is used in most commercial digital cameras. These sensors consist of small blocks representing pixels which are equipped with:

- photosensitive diode

- electron trap

- analog amplifier

- analog to digital converter

Each pixel can produce final digital value of each pixel, basically no other components are needed to read image data.

Figure 2.2: Diagram of CMOS chip [16]

Most CMOS sensor use *rolling shutter*. This process of capturing image data in which each row of pixels is exposed to light separately causes an unwanted effect if CMOS sensor

is capturing a fast moving image. The effect of rolling shutter can be overlooked if data on display are static.
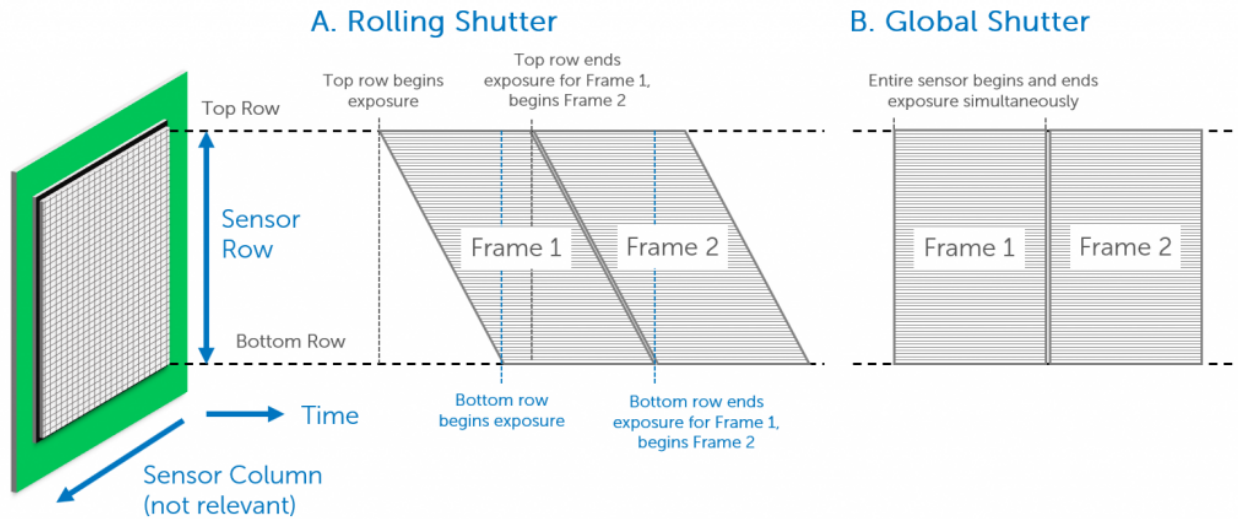
Figure 2.3: Rolling vs global shutter [17]
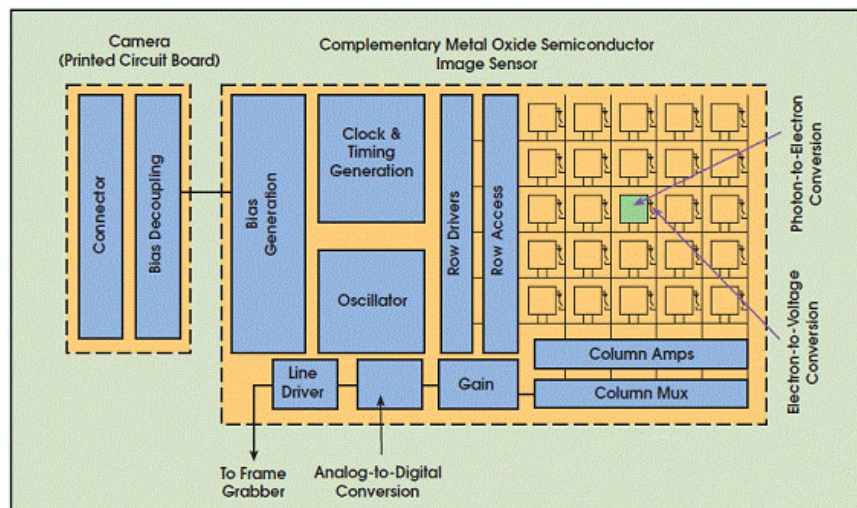
CCD

CCD sensor is typical for its ability to capture whole frame in one instance. CCD sensors utilize a method where the charge built on each photodiode is passed to common output which is then passed to amplifiers and analog to digital converters. The advantage to this technique which can be defined as *global shutter* is that there is no motion blur and CCD chips suffer less from image noise since the pixel area can be used more effectively[10].

In conclusion, both sensors are suitable for the purpose of display output data capturing. The only exceptions are displays with very fast refresh rate and fast moving objects, then a CCD sensor with good timing is preferred. Both sensors need to be calibrated and the cost is comparable.

2.1.2 Data degradation

Data degradation is one of the main drawbacks of camera system frame gathering. Data degradation can occur in camera sensor resolution and camera sensor color mask.

Camera sensor resolution

As mentioned in chapter 2.1.1, camera sensor is made from 2D-array of pixels. If the testing framework requires exact pixel color information then the camera sensor must have a resolution of equal (if aspect ratio is the same) or greater than tested display.

$$K_{sensor} \geq K_{display} \tag{2.1}$$
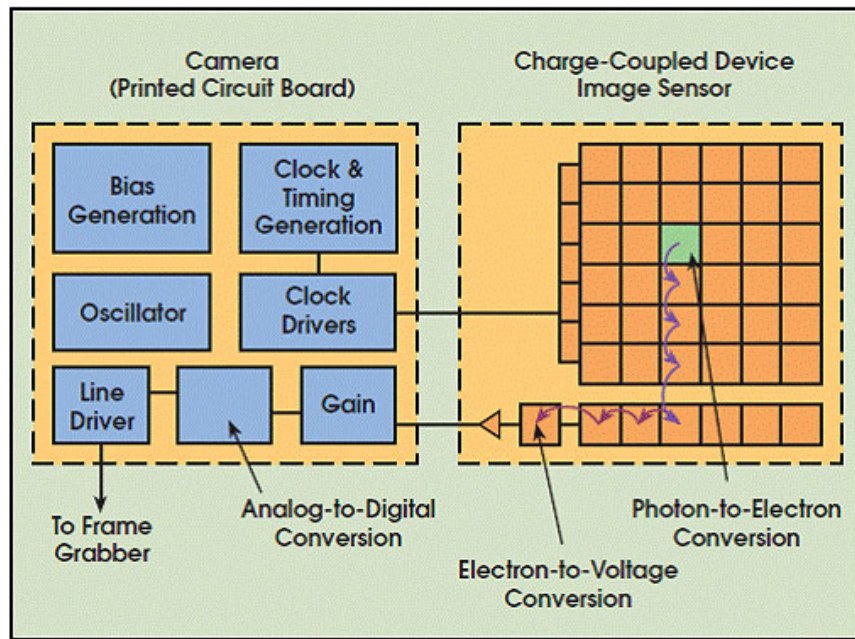
Figure 2.4: Diagram of CCD chip [16]

Where K = resolution in pixels.

Average camera sensor has a resolution of 12 Mpx which corresponds to 4056x3040 px[11] and maximum resolution defined in table 1.1 is 1920x1080 px. The resolution difference is not a problem for this application.

Example of position of display in camera frame is in figure 2.5
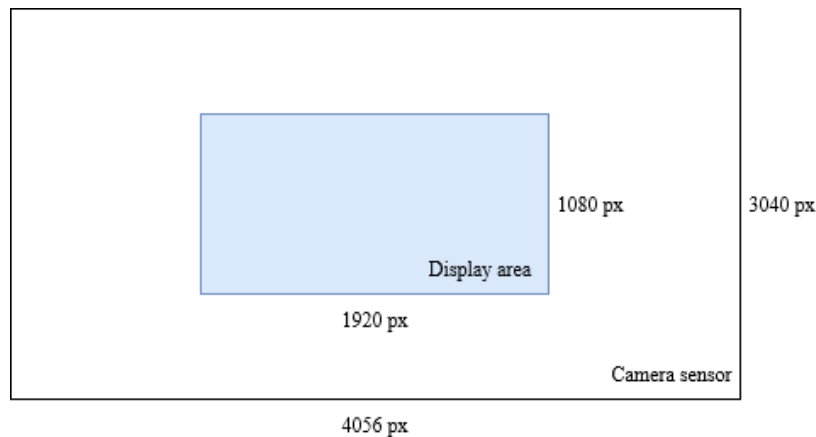
Figure 2.5: Example of display area in camera area [source: author]

However because the resolution is rarely identical, the gathered frame must undergo intense postprocessing to get precise pixel information gathered from the display or alternatively the whole setup must be exactly calibrated such that the display screen area exactly corresponds to pixel size of the camera sensor.

Camera sensor color mask

Camera sensor pixels does not distinguish between colors. The color information is obtained by filtering the light by Brayer mask or similar filter. This mask is placed on top of photodiodes and filters incoming light based on wavelength (color).
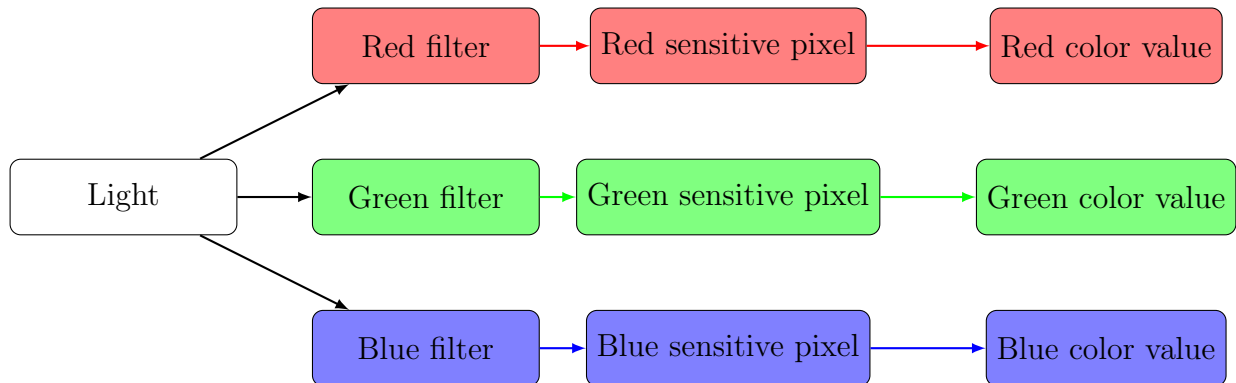
Figure 2.6: Color selection and color value extraction diagram [source: author]

This causes a photodiode to react only when light of specific spectrum passes the Bayer mask. The pattern of bayer mask is shown on figure 2.7.
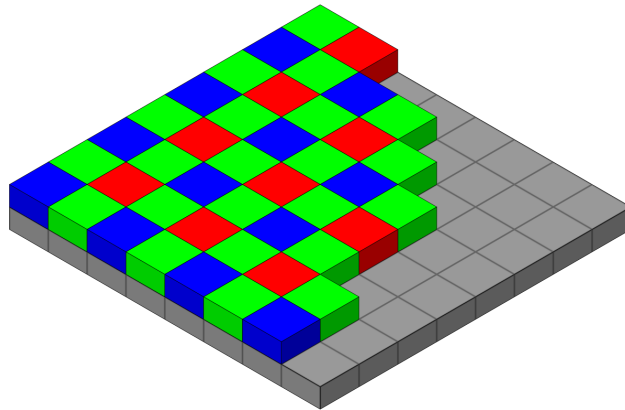
Figure 2.7: Bayer mask pattern layed on photodiodes [12]

The pattern is mosaic with a predominance of green pixels which was decided by Kodak in 1976 to mimic the physiology of human eyesight. Because the pattern is mosaic, a postprocessing called demosaicing must be applied.

This process interpolates the lost or missing color data. This postprocessing loads the data with an error which can produce inaccurate results. If the frame capturing system uses postprocessing such as sharpening or even color manipulation, the resulting data are unusable due to their inaccuracy and the fact that they do not represent the real output of the display.

To overcome these issues, very intense and sometimes impossible data preprocessing and
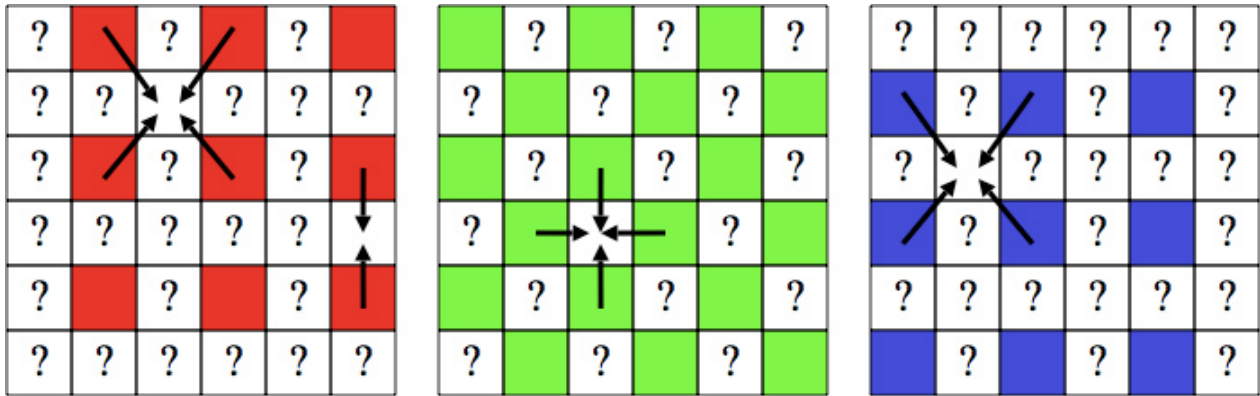
Figure 2.8: Interpolation of color data [15]

calibration must be applied before the data are used in testing framework. This will result in either incorrect results of testing framework or the need to design less strict testing framework which might lead to subtle errors.

However, if identical display data is not required, this problem could be overlooked.

2.1.3   Space occupation

Space occupation of any testing technology is important if the testing is done on multiple displays. As mentioned in chapter 2.1.2 camera sensor should be offset from a display to match pixel area. Even if postprocessing of interpolating colors was acceptable, the camera lens distortion is most notable on edges of frame and sensor offsetting is encouraged.

For these reasons, the overall space occupation of such a testing stand is much larger compared to other methods which require only minimal additional hardware for testing.

2.1.4   Image distortion

When capturing frame through a camera, lens distortion might occur. Lens distortion is a type of data degradation which happens because camera lenses bend the incoming light.

Lens distortion can be manifested in two main forms: barrel distortion and pincushion distortion. Barrel distortion causes straight lines to appear curved outward, like the sides of a barrel. On the other hand, pincushion distortion makes straight lines curve inward, resembling the shape of a pincushion. [21]

Additionally, chromatic aberration can contribute to image distortion. This occurs when different colors of light focus at slightly different points, resulting in color fringing along high-contrast edges. [22]

These problems can and must be fixed by heavy image postprocessing, expensive lenses and more distance between sensor and display which increase space occupation of this system. All of these solution might produce additional unwanted errors or damage data integrity during postprocessing.

Figure 2.9: Example of lens distortion and chromatic aberration [23]

2.2 Serial debugging

Serial debugging is another solution to gathering the output data. This process is based on the fact that embedded system controller must send commands to display driver as shown on figure 1.9.

2.2.1 Serial communication with display

Due to limited connection capacities of any microcontroller, serial protocols are widely used in embedded systems. Such protocols (more discussed in chapter 1.2.2) can be used to send exact information about what is requested to be shown on display, or preprogrammed commands are used. In either way, this communication is programmed by the user and is exactly defined.

Example of such communication is ST7920 display controller IC, which can communicate with MCU via SPI protocol. ST7920 has preprogrammed font and cursor position registers so the user only needs to send information what string is to be shown on display and its initial position. In contrast, the graphical option requires the MCU to send exact information about which pixel is to be turned on or off. For colour displays the dataflow of such commands are much more data heavy.

For this reason some display graphic controllers come preprogrammed or if they are part of embedded system, the user can pre-program them to send predefined data to display driver which greatly reduces the information needed to be sent from the MCU. This comes at the cost that command dataflow does not contain exact pixel values which is fatal when trying to reconstruct the image.

2.2.2 Image reconstruction

Image reconstruction when using serial communication is rather simple procedure due to previously mentioned display graphic controller responsible for translating the commands into actual image data.

Since the frame and display graphic controller behavior is predefined. It is as simple as catching a command to display some information via serial communication and matching it to requested command.

Testing framework output

Embedded system controller

command dataflow

Serial data sniffer — Reconstructed image → Testing framework input

command dataflow

embedded system
- - - - - - - - - Display graphic controller - - - - - - - - - embedded system
display system   display system

frame dataflow

Display driver

Display pixels
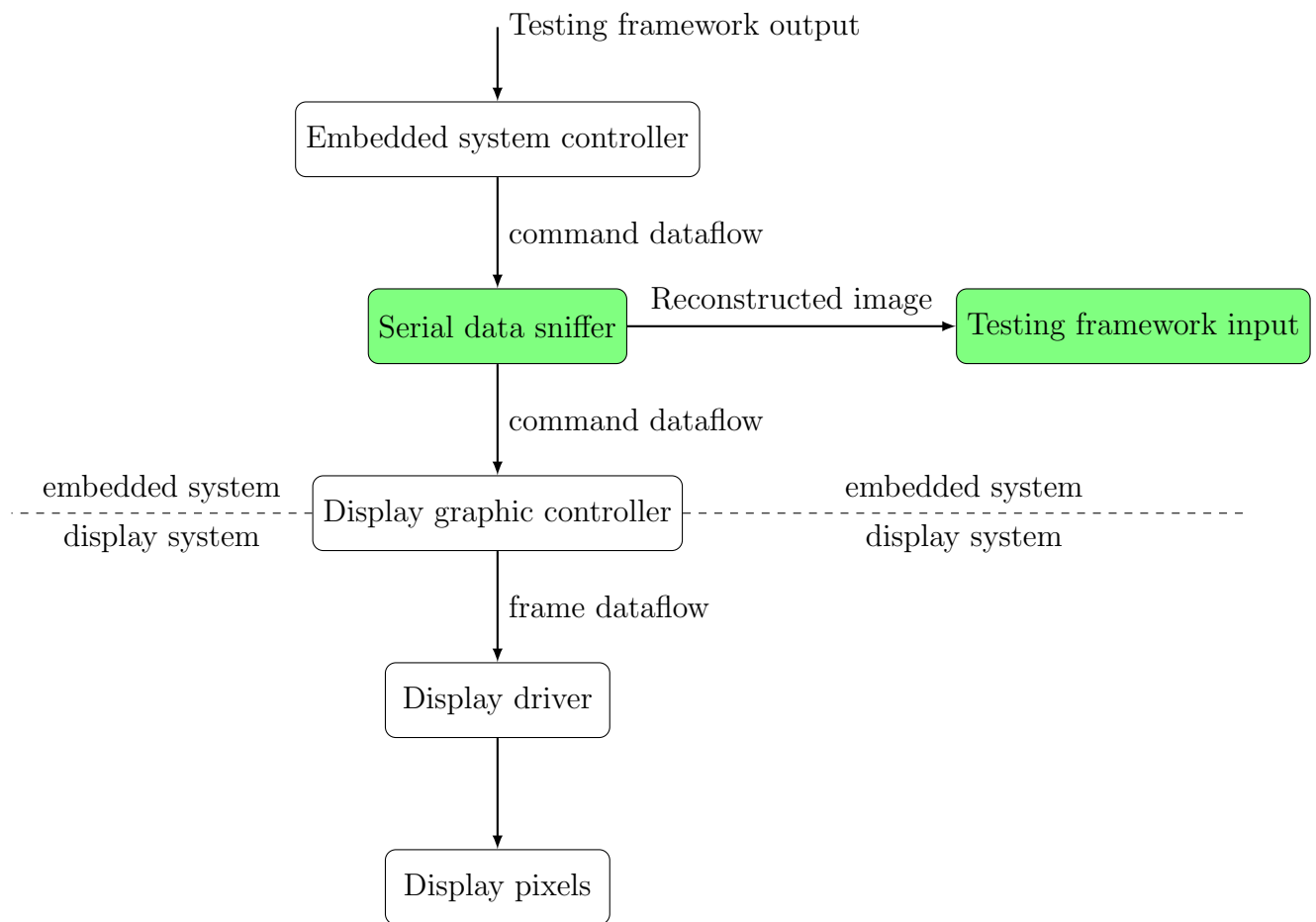
Figure 2.10: Serial dataflow capture diagram [source: author]

The biggest flaw of this method is, that serial debugging cannot see errors that might be produced by the display driver or the display itself since it is monitoring only the commands sent.

2.3 Frame grabbing hardware
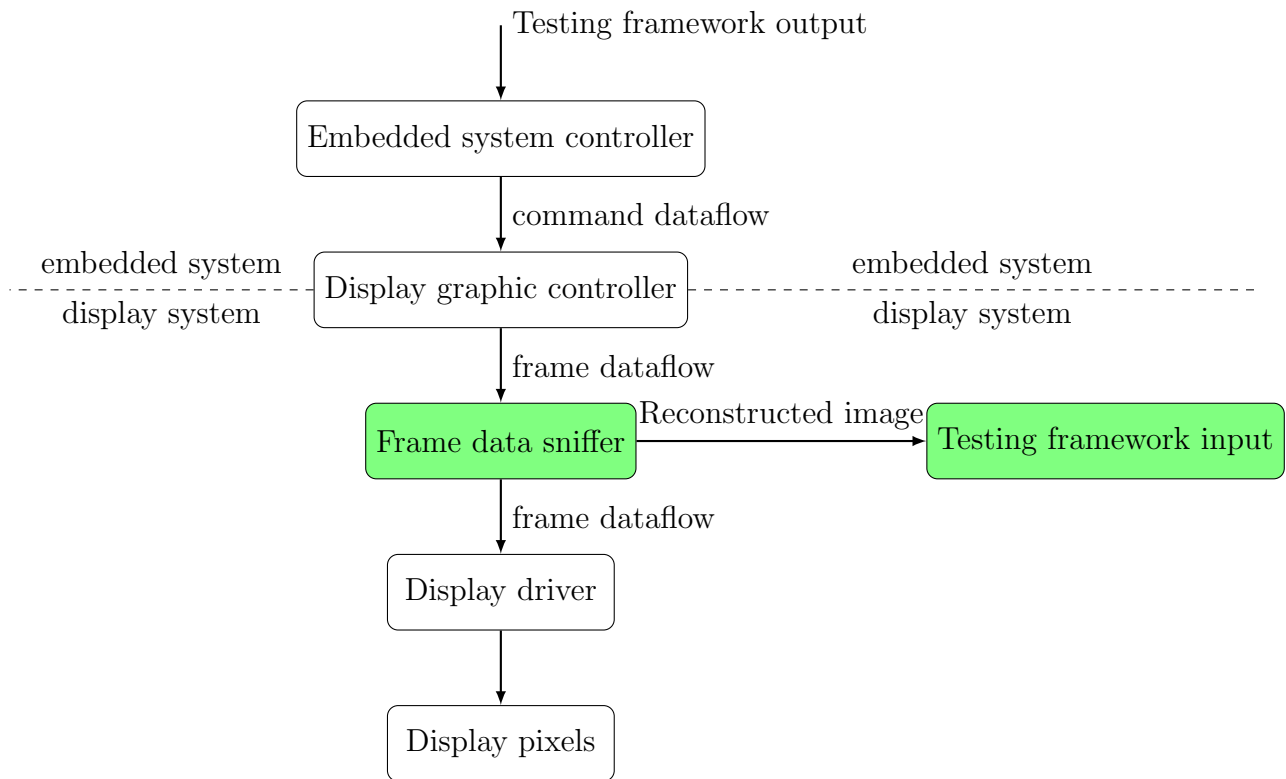
Figure 2.11: Frame dataflow capture diagram [source: author]

The method which is used in solution in this thesis is frame grabbing hardware. This method is similar to serial debugging with the difference of sniffed dataflow. Serial debugging focuses on *command dataflow* while the frame grabbing hardware is focusing on *frame dataflow*.

Frame grabbing hardware is a type of hardware device designed to capture video signals and convert them into digital format that can be processed by testing framework in this case.

2.3.1 Video signal protocols

Video signal protocol is a standardized form of transmitting video data. Standardization makes the protocols reliable in compatibility and interoperability. Video signal protocols can be separated into two basic categories.

- Analog video protocols

- Digital video protocols

Analog video protocols

Analog video protocols are an old method of transmitting video. Today mostly digital protocols are used. Analog protocols use voltage level that represent an intensity of some color and brightness. Analog protocols also include timing and synchronization signals. An example of analog video protocol is VGA.

- Old method

- Not usable for high resolutions

- Uses analog voltage levels

- Susceptibility to interference

- Timing is challenging

- Limited colour accuracy

- Signal degradation based on high distances

Digital video protocols

Digital video protocols are a method of transmitting video via digital signals. The video signal is transmitted in discrete manner using "HIGH" and "LOW" voltage levels. Video signal is represented as encoded digital data which can be transferred parallelly or serialized, see chapter 1.2.2 for explanation. Digital video protocols include for example **HDMI, DisplayPort, DVI, DPI** and many more. Digital protocols can handle higher resolution, higher framerates and better image quality as analog protocols.

- Binary representation (digital signal)

- Supports high resolution

- Resistant to signal degradation over long distances

- Full color accuracy

- Resistant to interference

- Adjustable for various use cases

2.3.2   Image reconstruction

Image reconstruction when using frame grabbing hardware is based on "simulating" display driver. Since frame grabbing hardware can capture raw graphical information the simulation is a simple decoding of received signal. This method can be applied to analog or digital signals provided that analog signal is converted into digital signal that can be processed by a decoding unit.

2.3.3 DPI data

As an example a DPI protocol, described in 1.2.2, data would be decoded by following steps:

1. Capture DPI data

2. Synchronize data (if needed)

3. Decode data

4. Save and output frame

The maximum speed a frame grabbing hardware can go through all needed steps is defined in frames per second. Perfect system would have FPS of display and FPS of grabbed frames equal.

Capturing DPI data

The fastest signal defining highest frequency in DPI protocol is the clock signal. The clock signal is used to synchronize the whole protocol. Frequency of clock signal can be defined by the following formula:

$$f_{clock} = H \times W \times f_{display} \tag{2.2}$$

where f_{clock} is frequency of the clock signal in Hz, H is height of display in pixels, similarly W is width of display in pixels and $f_{display}$ described the refresh rate of the display in Hz.

According to the requirements, table 1.1 the maximum expected frequency of display is

$$f_{clock,fullHD} = H \times W \times f_{display}$$

$$f_{clock,fullHD} = 1920 \times 1080 \times 24$$

$$f_{clock,fullHD} \approx 50 \times 10^6 \tag{2.3}$$

The minimum **theoretical** required clock frequency of frame grabbing hardware is equal to DPI clock frequency. DPI protocol validates data on rising edge of clock. If frame grabbing hardware was a little offset from the display clock, the waveforms would look as displayed in following figure:

However since the frame grabbing hardware must simultaneously process captured data, the frame grabbing clock frequency must be higher. The exact ratio is discussed in chapter 3.2.3.

Figure 2.12: Theoretical timing diagram of frame grabbing hardware [source: author]

Synchronizing data

The synchronization of data step is required only if display clock and display data are not exactly synchronized. As an example, if display data are slower than display clock and frame grabbing clock is faster than the difference a data mixup can occur. When this error is present on the display system, the simplest solution is to offset frame grabbing data read by a fixed value. Following figures display this error and its correction.
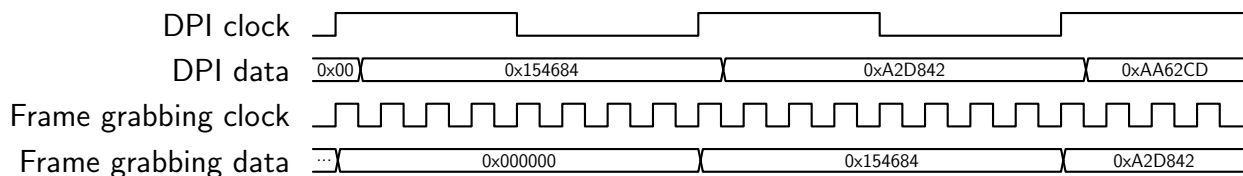
Figure 2.13: Data synchronization error [source: author]
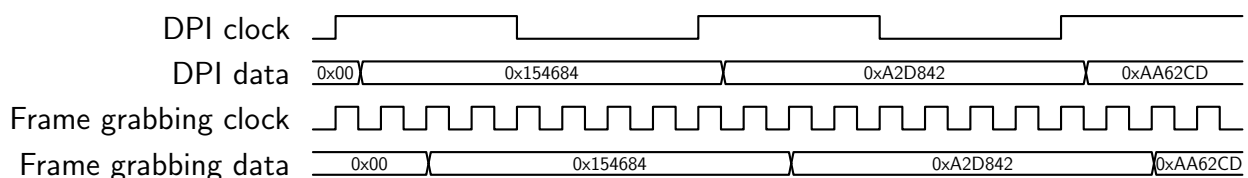
Figure 2.14: Data synchronization error correction [source: author]

Figure 2.13 displays a situation when frame grabbing clock acquires data on rising edge of display clock. Since display data are slower, the frame grabbing hardware captures an old pixel value which causes the output data to be corrupted and unusable.

Figure 2.14 displays one of possible fixes/protection against this error. Frame grabbing hardware will wait for a specified time (3 clock signals in waveform) before capturing display data.

Decoding captured data

For decoding DPI image data all signals listed in table 1.2 must be used. All data are saved into frame array. Frame grabbing hardware reacts to display clock with delay according to figure 2.14 if needed. Display data are concatenated into single value with size dependant on color format (8,16,24,... bit value). Frame grabbing hardware then reacts to *H-SYNC* that signalizes new line, and line index of frame data is increased. Another important signal is *V-SYNC* which signalizes that whole frame has been transferred and frame grabbing hardware can proceeed to the next step.

Details about this process are discussed in chapter 3.2.4.

Output gathered frame

Frame grabbing hardware saves the gathered data in internal memory in an array or similar memory structure. These data could be directly used to evaluate quality in testing framework, but sometimes other image formats, such as PNG, is preferred. Both formats, raw or translated, need to be somehow transferred to the testing framework.

For this reason a frame grabbing hardware must be able to output gathered data effectively and in lossless format. Data can be transferred directly via datalines (USB, ...) or some on board protocol (SPI,I2C, parallel,...) if frame grabbing hardware is located on the same system as testing framework. However, if these two subsystems are separated or more testing frameworks need access to the grabbed data, a server is a more flexible solution. A local server able to provide data to any testing framework when requested is preferable solution since these two subsystems do not have to be integrated together and programmer can access data manually if needed either for manual inspection or any other reason.

Server and data access setup is discussed in chapter 4.3

2.4 Chosen technology

Based on chapters 2.1,2.3 and 2.2, table 2.1 was constructed to compare available frame grabbing technologies.

Technology	Practicality	Data integrity	Reliability	Scalability	Max. display resolution
Camera	Costly and spaceconsuming	Depends on sensor and environment	Depends on build quality	Very low due to space occupation problems	Based on lens and camera sensor, has upper limit based on the available space
Serial debug	Cheap, easy integration	Very low or none	Very high, produces consistent results	Very high, easy to apply to muliple systems	No limits
Frame grabbing hardware	Costly, complicated integration	1:1 data values, does not check physical display	Very high, based on high-end technology	Very high, compact and easy to expand	No limits

Table 2.1: Available technology comparison [source: author]

Frame grabbing hardware was used as a solution for its exact and reliable data values. Additionally, frame grabbing hardware is superior in compactness, scalability and no display resolution or data protocol limitation. Disadvantages such as high cost and complicated integration can be reduced by using the right technologies to simplify user experience. If display correctness testing is a requirement, frame grabbing hardware can be paired with camera technology.

3 FPGA implementation

Based on the requirements for speed discussed in chapter 3.2.3 and the practical data output requirement discussed in chapter 2.3.3 PYNQ-Z2 development board was chosen for it's FPGA circuit implementation along the processing system. [13].
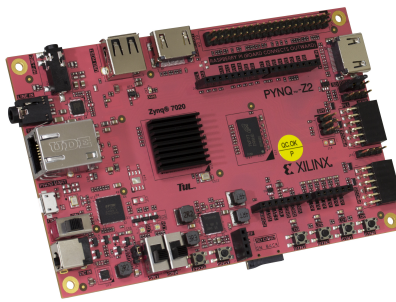
Figure 3.1: PYNQ-Z2 [13]

3.1 PYNQ-Z2 programmable logic

PYNQ-Z2 features a programmable logic circuit based on a Xilinx Artix-7. FPGA is a type of integrated circuit which can be configured by programmer to perform specific tasks by programming digital logic gates and arrays. FPGAs excel at parallel processing, versatility and speed.

3.1.1 FPGA on PYNQ

PYNQ-Z2 takes the versatility of FPGA technology a step further by implement it on a system with "PYNQ overlay" which is capable of simple interconnection of FPGA and processing system. FPGA for PYNQ can be programmed using commercially available IDEs such as Vivado, which is used for this application.

Table 3.1 summarizes some important specifications regarding FPGA technology on PYNQ-Z2.

Maximum clock frequency	200 MHz*
Logic slices	13300
fast RAM	630 kB
DSP slices	220

Table 3.1: PYNQ-Z2 FPGA basic specifications [13]

*The actual maximum clock frequency is defined by the complexity and how time demanding a designed circuit is.

3.2 FPGA programming

Chosen technology for gathering display data is frame grabbing hardware based on FPGA technology. FPGA technology provides fast, effective and robust solution for extremely fast data manipulation. The FPGA "circuit" is designed as a block diagram in Vivado IDE using VHDL programming language must be capable of the following points:
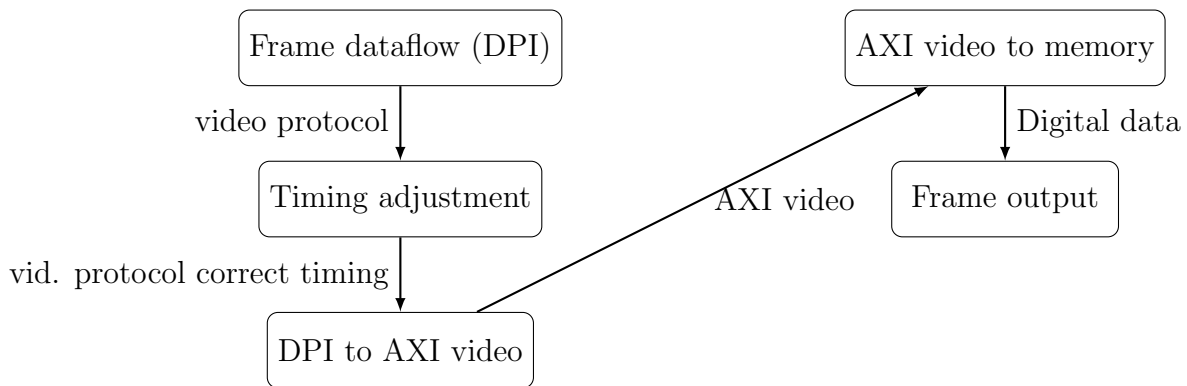
Figure 3.2: Simplified FPGA data processing [source: author]

1. Gather DPI data

2. Analyze DPI data and adjust timing if needed

3. Convert DPI data into internal AXI protocol

4. Save frame data into buffer RAM memory

5. Enable processing system to access the frame via interconnections

6. Enable processing system to access information about DPI protocol (frequency, resolution, etc.)

The final block diagram is displayed in figure 3.3.

Figure 3.3: Display sniffer block diagram in Vivado IDE [source: author]

3.2.1 Vivado

Vivado is an integrated development environment designed by Xilinx which also provides an integrated circuit used on PYNQ-Z2 board. Vivado is used for design, verification and implementation of FPGA circuits.

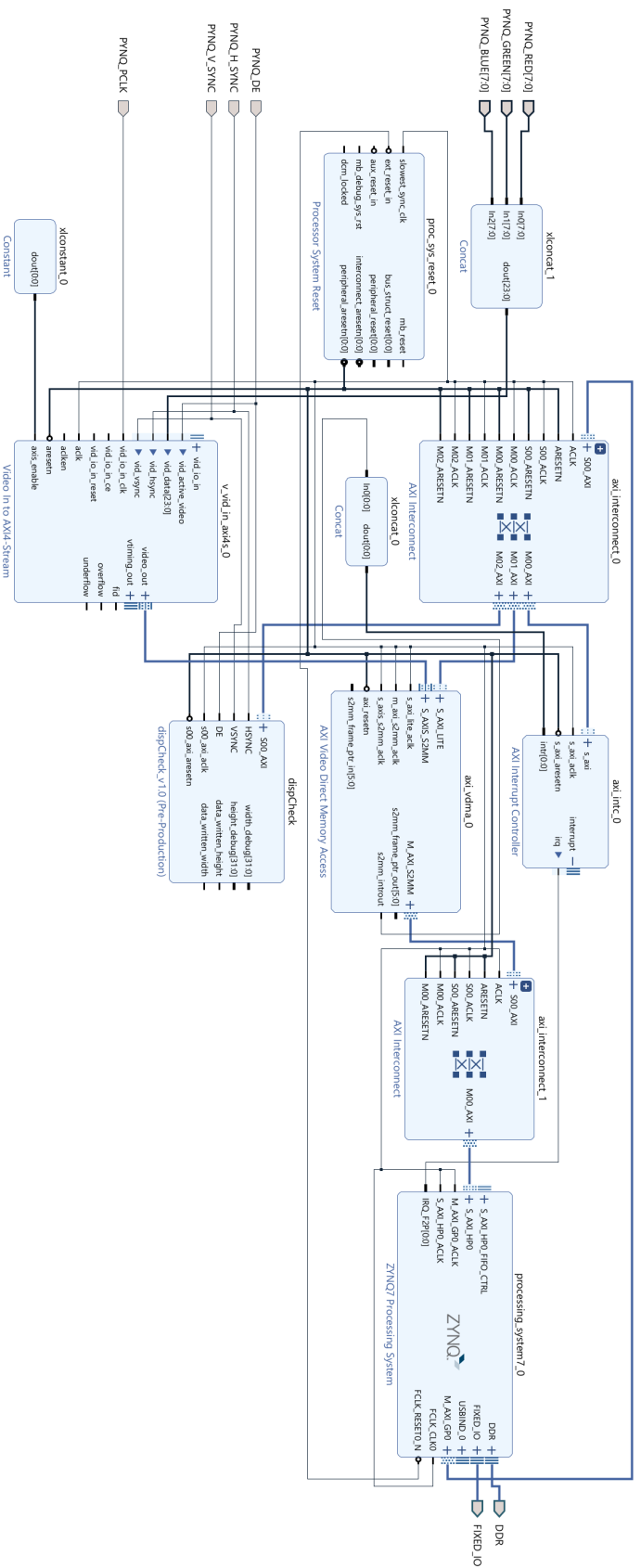Circuit is designed as a block design where each block is programmed in VHDL or Verilog programming language. Some blocks are preprogrammed by Xilinx and some blocks are programmed specifically for this thesis. Following subsections discuss each part of the FPGA design.

Compiling output for FPGA

Compiling output data for FPGA is different than for other programming languages.

For example when C languages are compiled, the compiler follows steps such as **preprocessing**, **compilation** and **linking** [24].

Compiling code for hardware description languages such as VHDL used in this project for FPGA system requires different approach. Compilation of VHDL consists of:

1. Analysis: this phase involves reading and parsing the VHDL source files and building an intermediate representation, any parsing or syntax errors are caught in this step.

2. Elaboration: This step instantiates all components and elaborates the design hierarchy.

3. Synthesis: Until this step, hardware was interchangeable. Synthesis step generates netlist of all elements on specific hardware.

4. Implementation: Implementation retrieves the netlist and generates routes that will be later programmed on the target hardware.

5. Generating bitstream: While synthesis and implementation steps defined all connections and settings on FPGA target, the generated bitstream contains all the information needed to program the FPGA circuit. This file with extension *.bit* is loaded into processing system and can be used later to setup the programmable logic of mentioned PYNQ-Z2.

3.2.2 AXI protocol

AXI stands for *Advanced eXtensible Interface* and is a widely used protocol in the field of digital design and FPGA. It is a protocol and set of rules defining how each block and module communicates within digital system, in this case FPGA. It is crucial to highlight the advantages of this protocol since its fast and reliable. AXI uses master-slave architecture, where master initiates transaction and slave responds.

AXI protocol has subsets, each designed and perfected for a specific task.

AXI4 is a standard and is used when high frequency and performance is needed. AXI4 includes features such as burst transfers or separate channels.

AXI4-Lite is a simplified version of AXI4, it is designed for a simpler control or low-bandwidth applications. It has a reduced complexity compared to AXI4 and additional features, such as burst transfers or separate channels, are not present.

AXI4-Stream is designed for streaming data interfaces. Unlike AXI4, which is more

transaction-oriented, AXI-Stream is optimized for continuous streams of data.

AXI4-Stream Video is a subset of AXI-Stream and is also stream-oriented protocol optimized for streaming video data. This protocol includes conventions generally used in video processing applications.

AXI-Interconnect is not a protocol itself but it is an important component used to connect multiple AXI masters and slaves.
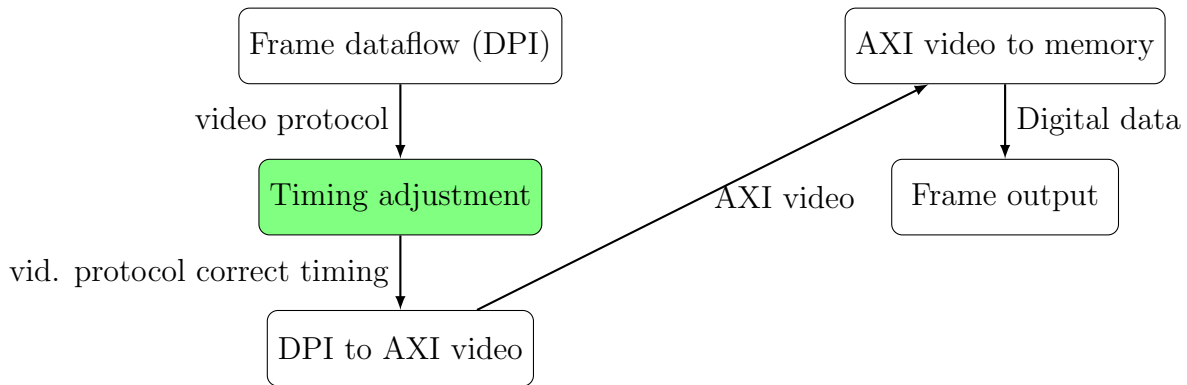
3.2.3 Data timing correction using FPGA

Figure 3.4: FPGA data process timing correction block [source: author]

Data timing correction problem is a common issue that occurs anywhere data is being sampled or gathered in any form.

In theory, we should be able to gather digital signal with *sensor* having the same sampling frequency as the *source* data. However, in reality this is not true and the *source* must have higher frequency to correctly gather information about/from the *source*.

For this reason a measurement was conducted where the impact of different ratios of $\frac{f_{sensor}}{f_{source}}$ was measured.

For measurement, a frequency generator was used as a source. PYNQ with custom-made frequency analyzer was responsible for frequency detection. Graphs showing the results from measurements demonstrate what frequency the FPGA circuit measured with specifically selected prescaler of sampling clock frequency.

Graphs of final measurements are displayed in figure(s) 3.5

(a) Source set to 31.5 Hz   (b) Source set to 122 Hz   (c) Source set to 500 Hz

(d) Source set to 4 kHz   (e) Source set to 7.8 kHz   (f) Source set to 31 kHz
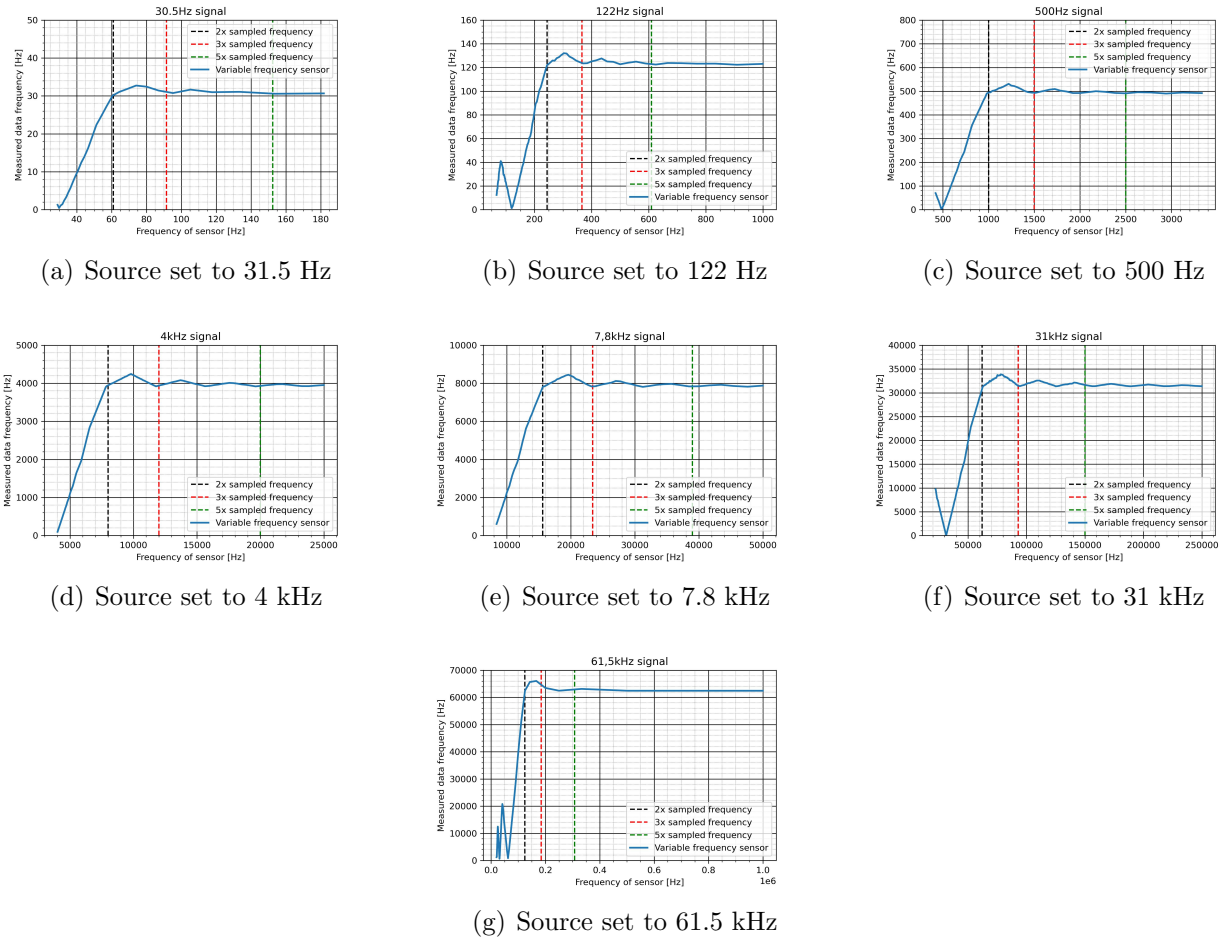
(g) Source set to 61.5 kHz

Figure 3.5: Measurements of frequency identification correctness based on sensor and source frequency ratio [source: author]

From these measurements the recommended ratio of f_{sensor} to f_{source} is 5. Which means that the PYNQ board should sample the display with 5 times higher frequency.

3.2.4   Display parallel interface to AXI protocol

After the DPI data enter the FPGA and are correctly timed, they must be translated to protocol the internal wiring of FPGA can work with. The protocol chosen is **AXI-stream video** since it is the easiest to implement for video data transfer.

Axi-stream protocol uses different naming for each signal. Table 3.2 summarizes the comparison between DPI data and AXI-stream data.

The protocols are similar in functioning with the only difference in effectivity and speed in which AXI is better since it is an internal communication protocol.

This conversion is handled by *Video to AXI4-Stream* IP. This IP must be set up before generating bitstream and cannot be edited later in the process. Because of this disadvantage, every display protocol and colour format must have individual bitstream generated. Since
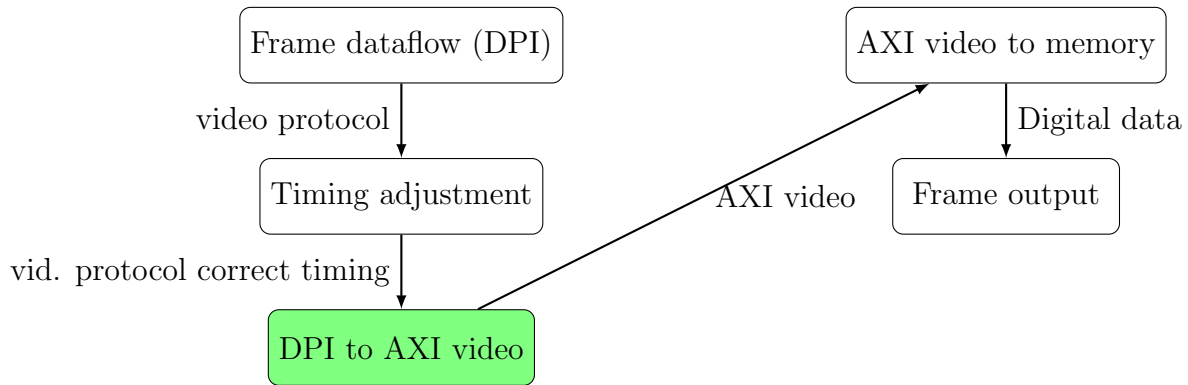
Figure 3.6: FPGA data process DPI to AXI block [source: author]

DPI	AXI	Purpose
Data enable	valid	Signalizes valid pixel data
H-sync	tuser	Synchronizes end of line
V-sync	tlast	Synchronizes end of frame
Data	data	Colour values, changes with each pixel

Table 3.2: DPI to AXI signals [source: author]

the bitstreams can be uploaded by the processing system this disadvantage is not detrimental to the project.

3.2.5   Display check IP

Display check IP is a quintessential part of the whole design. This IP is responsible for monitoring information about the connected display, statuses of all data transfers, statuses of errors that do not cause system crash and report all the information to processing system. The processing system needs this information for proper functioning and whole system setup.
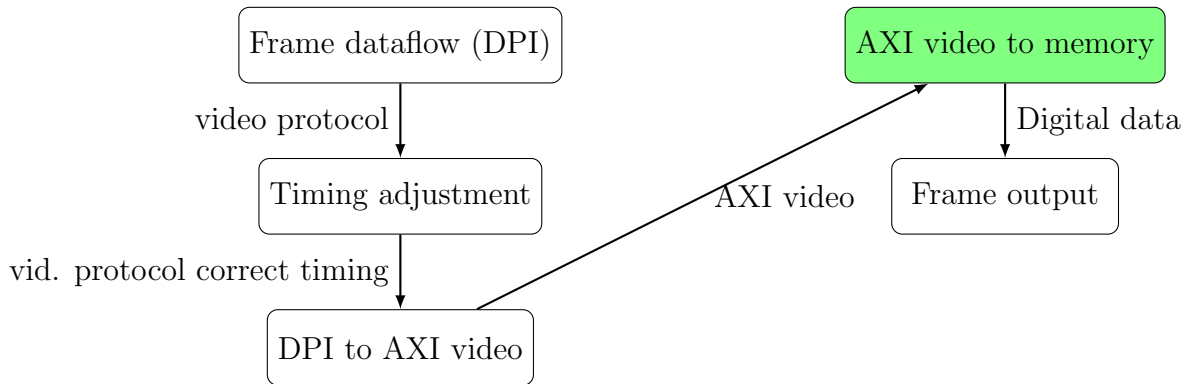
3.2.6   Video direct memory access (VDMA)

Figure 3.7: FPGA data process AXI to memory block [source: author]

Video direct memory access or VDMA is a block commonly used in the Xilinx Vivado development environment. VDMA enables the efficient transfer of video data between different memory locations without the need for constant intervention from the processing unit. It is often employed in applications such as video streaming, image/video processing, and graphics rendering.

By using VDMA common problems with memory accessing are solved beforehand, allowing the processing unit to simply adjust VDMA settings on the fly and gather raw frame data by directly accessing parts of memory where the frame is stored.
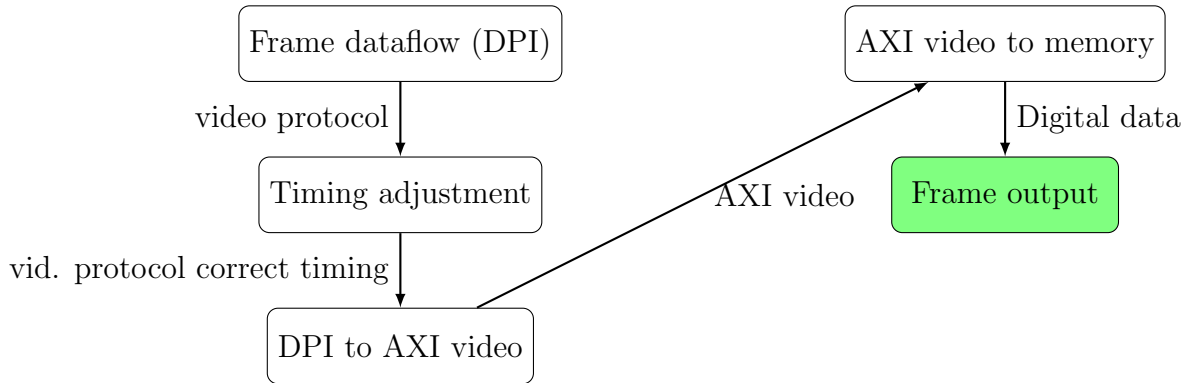
4 Display sniffer server

Figure 4.1: FPGA data process frame output block [source: author]

When frame data arrive to the system memory and are saved in known memory location handled by VDMA (3.2.6), they must be further processed and delivered to the user.

The output of frame can be handled by plethora of methods. For the highest possible simplicity and practicality for the user, a **server-client** method is used.

This method ensures that the data can be accessed by any device connected to the same network as the PYNQ testing station.
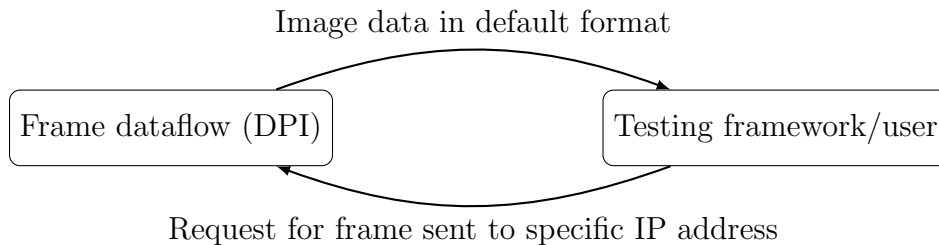

Figure 4.2: PYNQ and testing framework connection [source: author]

4.1 PYNQ-Z2 processing system

Further data manipulation is handled by PYNQ processing system.

The processing system can be described by few key points:

- Dual-core ARM

- Clocked at 650 MHz

- Capable of running operating system

- Multiple connectivity options (USB, Ethernet, UART, etc..)

- PYNQ operating system based on Linux and Python environment

- Can interact with programmable logic via PS-PL interconnects

4.1.1 PS-PL interconnect

PS-PL interconnects are a crucial parts of PYNQ main processing unit. Processing system and programmable logic cannot communicate and exchange data directly. This data manipulation and communication must be handled by another subsystem. This subsystem ensured correct timing and error handling. Following simplified diagram describes the internal layout of ZYNQ.

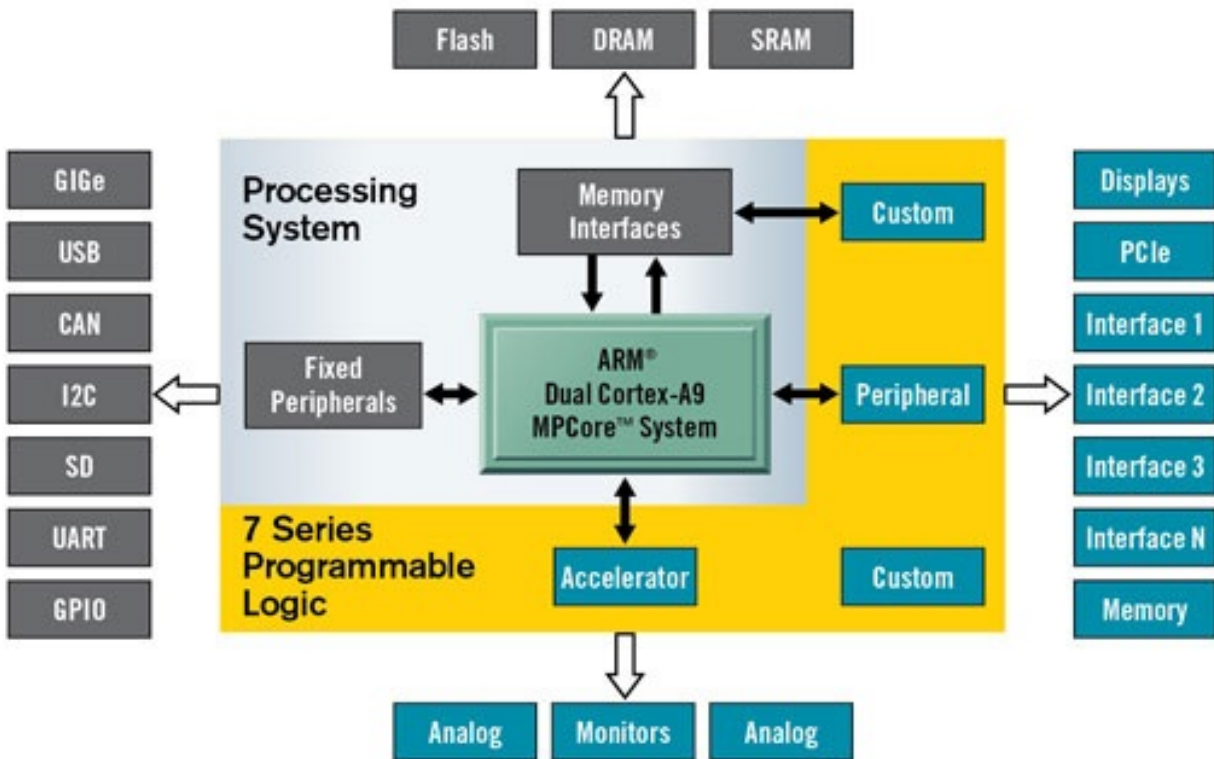Figure 4.3: Simplified internal ZYNQ diagram [18]

The PS-PL interconnect allows processing system to access any memory parts from programmable logic region. As mentioned in chapter 3.2.6 where VDMA is discussed, the VDMA is responsible for saving image data to specific memory location and then pass their location and size to processing system which can read it via the PS-PL interconnect directly from the memory.

4.1.2 Operating system

The processing system is hosting an PYNQ Linux operating system (OS). This operating system manages hardware connections and provides a software interface for applications.

This OS can provide almost all capabilities exactly as any other Linux based OS, extended by PYNQ overlay, enabling the access to PL part of ZYNQ chip via Python libraries. PYNQ Linux can be managed either by connecting directly via USB or by ethernet.

For data access, software programming and overall manipulation jupyter notebook environment or classic command line interface is used.

4.2 Data manipulation

Before the data in a form of images can be accessed or requested by the user, these data must follow several steps:

1. Activate frame data gathering

2. Retrieve frame data from PL

3. Convert raw frame data into image

4. Apply post-processing to converted image

5. Save or send final image

4.2.1 Data gathering

Data gather is done by initiating FPGA overlay, VDMA and calling VDMA instance in python script. VDMA must be setup with specific expected resolution. This initiation can be changed any time.

VDMA is then started when frame data are to be gathered. Raw frame data are requested and after they are received, VDMA is stopped to prevent memory overflow.

Simplified example of VDMA use case is described in code snippet 1.

```
1   # import required packages
2   import pynq
3   from pynq import allocate, Overlay
4   import pynq.lib.video as pynq_video_lib
5
6   # initiate FPGA overlay and vdma instances
7   overlay = Overlay("bitstream.bit")
8   vdma = overlay.axi_vdma_0
9
10  # setup vdma module
11  vdma.readchannel.mode = pynq_video_lib.VideoMode(frame_width,
12                                      frame_height,
13                                      bits_per_pixel)
14
15  # read frame using asynchronous function
16  vdma.readchannel.start()
17  raw_frame_data = await vdma.readchannel.readframe_async()
18  vdma.readchannel.stop()
```

Code snippet 1: Simplified python code for VDMA handling [source: author]

4.2.2 Data conversion and post-processing

After raw frame data are saved in a variable, they need to be processed into usable image format. After testing all formats, BMP was chosen. When raw frame data were processed the BMP format proved to be the most effective in speed and quality. More details about the testing in chapter 5.2.4. When image is saved as BMP, it is transformed into bytes format in order to send it to user.

```
1   # create Image instance from raw_frame_data
2   bmp_image = Image.fromarray(raw_frame_data)
3
4   # save image instance and convert it to bytes format
5   with io.BytesIO() as buf:
6       image.save(buf, format='BMP')
7       image_bytes = buf.getvalue()
8   return Response(image_bytes, media_type = "image/bmp") # return bytes data to user
```

Code snippet 2: Simplified Python code raw frame postprocessing [source: author]

Another post-processing features such as image cropping or colour value editing can be done using the *Image* object functions.

4.3 Server API

As mentioned in chapter 4, **server-client** method of data accessing is implemented.

In order to make the server manipulation and data accessing as simple as possible a server API is implemented.

4.3.1 FastAPI python package

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python programming language. It is designed to be easy to use, fast to run, and to produce fast code [25]. Some key features and aspects of FastAPI include:

- **Fast**, supports asynchronism

- **Secure**, build-in features for higher security

- **Websockets**, in addition to HTTP, FastAPI supports WebSocket connections

- **Auto documentation**, useful when project gets larger

4.3.2 Server API

FastAPI is capable of maintaining server that is hosted on mentioned OS of PYNQ development board.

Code snippet 3 shows a simplified example of such framework implementation with an example function. If user or system wants to access wrapped function and retrieve data returned by the function, IP address with the function path must be called. If input data are to be sent, following format is used.

192.168.2.207:8000/path/data=my_custom_data_input

```
1   # import fastAPI
2   from fastapi import FastAPI
3
4   # create framework app object
5   app = FastAPI()
6
7   @app.get("/") # web address to call wrapped function
8   def read_root():
9       return {"online"} # anything returned will be "sent" to user
10
```

Code snippet 3: Simplified python code for fastAPI utilization [source: author]

Display sniffer has several public functions accessible by user. Table 4.1 summarizes all of them with brief comment. Details about these processes are discussed further below. Every external method is handled by fastAPI and wrapped functions. Internal methods cannot be accessed by the user.

Function	Used for	Returns
root (initial function)	default landing page, initiates resolution detection	current settings of server
set_color_format	used for setting display color format	1 or error with details
set_resolution	used for defining display resolution	1 or error with details
set_overlay	used for loading FPGA circruit	1 or error with details
start_vdma	used for starting VDMA frame gathering	1 or error with details
stop_vdma	used for stopping VDMA, must be used after frame reading	1 or error with details
read_video	used for retrieving multiple frames in rapid succession	list of frames with video statistics or error with details
read_frame	used for retrieving single frame	image in specified format or error with details
detect_resolution	used for manually detecting resolution of display	1 or error with details
reset_server	used for software reset, erases all settings	1 or error with details
get_possible_settings	used for retrieving all possible settings	list of settings or error with details

Table 4.1: Table of usable function in display sniffer API [source: author]

Root

Root function is the default function that gets called when the user accesses display sniffer server. Root is used for checking if connection was established, if server is running and to retrieve server settings. This function should be called when testing framework is initiated or if connection/settings had to be checked.

Root is accessed by:
SERVER_IP:PORT/
Root input data:

- **None**

Root conducts following steps:

1. sets up default overlay

2. detects resolution of connected display

3. returns settings of display sniffer server

Set colour format

Colour format function is used for defining colour format of connected display. This colour format must be known in advance. This function addresses a problem of different colour format processing. Server has multiple bitstreams with different color format pre-cessing capabilities included.
Color format function is accessed by:
> **SERVER_IP:PORT/setColorFormat/colorFormat={color_format}**
> Color format function input data:

- **color_format** (RGB888, RGB565, etc.)

  Color format function conducts following steps:

1. checks if input color format is valid

2. updates color format setting for correct bitstream

3. returns True (1) or error with details

Set resolution

Resolution setting is crucial for correct VDMA functioning. If the user does not know the resolution of connected display, automatic resolution detection might be used. The resolution includes porch regions which are empty pixels. The resolution must be set including porch region which is later removed at the client side.
> Set resolution function is accessed by:
> **SERVER_IP:PORT/setResolution/**
displayWidth={width}/displayHeight={height}
> Set resolution function input data:

- **width** width of display + front porch in pixels

- **height** height of display + top porch in pixels

  Set resolution function conducts following steps:

1. Checks if height, width are valid values (int!=0)

2. Updates internal resolution setting

3. Returns True (1) or error with details

Set overlay

Set overlay is responsible for calling overlay set procedure. This procedure will "upload" a desired FPGA circuit to PL side of ZYNQ.

Set resolution function is accessed by:
SERVER_IP:PORT/setOverlay
Set overlay function input data:

- **None**

Set overlay function conducts following steps:

1. Check if colour format was chosen

2. Check if resolution is set

3. Stops VDMA if it is running

4. Loads bitstream from memory and uploads it to PL

5. Initiates used IPs

6. Sets up VDMA (resolution, color format)

7. Returns True (1) or error with details

Start/Stop vdma

Starting and stopping VDMA is crucial for the correct functioning of the whole system. Before each reading session, VDMA must be started. If no frame will be read in upcoming time, the manual stopping of VDMA is recommended.

Start/stop vdma function is accessed by:
SERVER_IP:PORT/startVDMA
or
SERVER_IP:PORT/stopVDMA
Start/stop vdma function input data:

- **None**

Start/stop vdma function conducts following steps:

1. starts/stops the VDMA module

2. returns True (1) if operation was successful

Read frame

Read frame is a basic function that can be used to retrieve frame data from display sniffer. To use read frame function, correct overlay must be set first.

Read frame function is accessed by:

SERVER_IP:PORT/readFrame/image_format={image_format}

Read frame function input data:

- **image_format** format of returned image (BMP, Numpy array, JPEG, ...)

Read frame function conducts following steps:

1. checks if requested image format is valid

2. checks if resolution is set

3. checks if overlay is set

4. checks if VDMA is running

5. reads data from VDMA (async function)

6. converts raw data into requested format

7. returns frame in requested format

Read video

Read video is similar to read frame function. Its purpose is to capture multiple frames as fast as possible. To save time for postprocessing raw dat, the frames are read in the requested FPS speed and saved locally. After all frames are saved they are processed together and sent back to user as a list of frames with information about speed and real FPS.

Read video function is accessed by:

SERVER_IP:PORT/readVideo/frames={frames}
/fps={fps}/image_format={image_format}

Read video function input data:

- **frames**, how many frames the system will read

- **fps**, requested FPS

- **image_format**, defines final image format (BMP,Numpy array, JPEG,...)

Read video function conducts following steps:

1. checks if requested image format is valid

2. checks if FPS request is valid

3. checks if number of frames is valid

4. checks if resolution is set

5. checks if overlay is set

6. calculates needed time period for one frame

7. reads frame and appends it to the raw frames list

8. records timing information to the list of information

9. waits for specified time or immediately starts reading a new frame

10. repeats 7.,8.,9. until desired number of frames is read

11. starts postprocessing of raw frames

12. calculates statistics such as average FPS, max FPS, etc.

13. returns processed data and information as [{information dictionary},[processed frames]]

Detect resolution

Detect resolution is an useful function when user needs to manually check the connected display resolution. For detecting resolution, a default overlay is used. This overlay is not capable of outputting any frames, its only purpose is to detect the resolution or if the display is connected at all.

Detect resolution function is accessed by:
SERVER_IP:PORT/detectResolution
Detect resolution function input data:

- **None**

Detect resolution function conducts following steps:

1. sets up default overlay

2. initiates dispCheck module

3. reads internal registers of dispCheck that store resolution

4. updates resolution information and returns it to user

Reset server

Software reset of server is useful if display is being changed, setup process is incorrect or if a new user is connecting to already running server that is in an unknown state.

Reset server function is accessed by:
SERVER_IP:PORT/softReset
Reset server function input data:

- **None**

Reset server function conducts following steps:

1. stops VDMA if its running

2. resets internal settings

3. uploads default overlay

Get possible settings

This function is used only for users that are not familiarized with the display sniffer system. Users can call on this function to receive a dictionary of all possible settings such as available colour formats, available image formats, current internal settings, etc.

Get possible settings function is accessed by:
SERVER_IP:PORT/getPossibleSettings
Get possible settings function input data:

- **None**

Get possible settings function conducts following steps:

1. Returns the information about available options in dictionary format

4.3.3 Client class

Client class is a control layer written in Python language. This layer is responsible for calling and executing functions mentioned in chapter 4.3.2 in a safe and controlled manner.

The documentation for client control layer can be found on the project github. [26]

Following code is an example of how to use display sniffer with following variables:

1. server ip = 192.168.2.127

2. display color format = RGB888

3. display resolution = unknown

```python
1   # import packages
2   from display_sniffer_handler import PynqHandler
3   from PIL import Image
4
5   # create display sniffer object
6   display = pynq_handler("192.168.2.127") # when creating object, IP must be specified
7
8   # check if server is online and receive settings including display resolution
9   server_status = display.server_check() # server_status has information about resolution and settings
10
11  # set display settings
12  display.height = server_status["frame_height"] # detected resolution
13  display.width = server_status["frame_width"] # detected resolution
14  display.colorFormat = "RGB888"
15  display.frontPorch = 0 # must be manually corrected
16  display.topPorch = 0 # must be manually corrected
17
18  # call for FPGA setup
19  display.setup(10) # number of seconds to wait
20
21  # request image in BMP format from server
22  image = display.readOneFrame(wait_time=10, mode="bmp", type="image")
23
```

Code snippet 4: Example of client connection to display sniffer [source: author]

5 Testing

After the system was completed, basic testing was done to further validate the functionality and effectivity.

5.1 Simulating video data

Since connecting real display would consume too much space and debugging would be challenging, other methods of simulating video data were chosen.

Simulating video data can be internally inside FPGA circuit or by external sources.

5.1.1 Test pattern generator (TPG)

The Test Pattern Generator IP Core is designed to produce test patterns essential for Video System initialization, assessment, and troubleshooting. Offering a diverse range of test patterns, the core facilitates users in debugging and evaluating color, quality, edge, and motion performance, addressing potential quality issues within the video system. It can be seamlessly integrated into an AXI4-Stream video interface, providing users with an option to either pass through the system video signals or incorporate test patterns as needed [19].

TPG is capable of

- Different resolutions (64x64 up to 8192 x 4320)

- Generating static or dynamic color patterns

- Outputting video data in a form of AXI-video

- Generating multiple colour formats

- Generating multiple colour depths

Example of some TPG patterns:

This IP is ideal for testing the connection between VDMA on PL side and Python script on PS side. TPG cannot generate DPI video data.

5.1.2 Raspberry Pi

For testing DPI video data gathering and processing, an external video image generator must be used.

For this purpose Raspberry Pi is used. Raspberry Pi is a compact and affordable mini computer. Despite its small size, it boasts significant computing power, making it capable of various tasks, including serving as a basic computer. Raspberry Pi is equipped with external

(a) Test pattern ID: 0x09   (b) Test pattern ID: 0x0B   (c) Test pattern ID: 0x0F
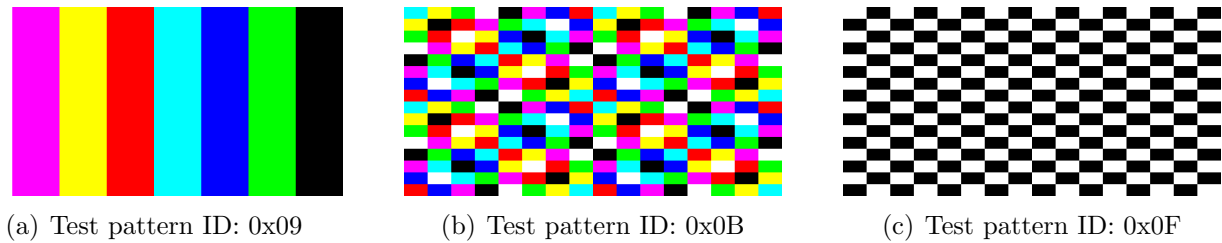
Figure 5.1: Test pattern generator correct patterns [source: author]

GPIO pins capable of either controlling external devices or receiving and transmitting various data using different protocols such as **UART, SPI, IIC** or for our use **DPI**. Its low cost and versatility have made it a popular choice among hobbyists, educators, and tech enthusiasts worldwide.

Figure 5.2: Raspberry Pi Zero [20]

Raspberry Pi is connected directly to PYNQ development board using Raspberry header. Raspberry is set to output graphical data as DPI protocol via GPIO pins. Resolution, display frequency, DPI timings and DPI setting as a whole is defined in *config.txt* by defining *dpi_timings*.

5.2 Test results

5.2.1 Python TPG pattern reading

First test was conducted to determine the speed and reliability of VDMA and Python reading sequence shown in code snippet 1.

Figure 5.3: TPG to VDMA to Python testing pipeline [source: author]

First TPG was set to produce patterns as in figure 5.1. Data from TPG are directly connected to VDMA and no other signal processing is done.

Color shifting

Problem encountered while conducting this test which is worth noting was colour data shifting. Figure 5.4 displays the error. Patterns are supposed to look like the patterns in figure 5.1.
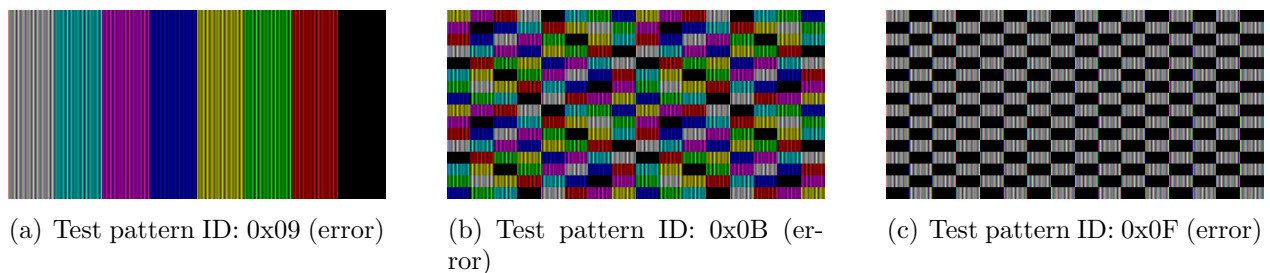

(a) Test pattern ID: 0x09 (error)   (b) Test pattern ID: 0x0B (error)   (c) Test pattern ID: 0x0F (error)

Figure 5.4: Test pattern generator incorrect patterns [source: author]

This error was caused by an incorrect setup of **AXI** protocol in FPGA. AXI protocol can transfer data with two general bit widths. This setting defines how many bits of data are transferred in one clock cycle. **Correct setting is 64 bits**. If the setting is set to 32 bits, the colour information will shift and the resulting image will not be correct.

Memory overflow

Another discovery worth noting is the fact that VDMA, when handled incorrectly, will overflow memory and cause fatal errors and segmentation faults. This will cause the whole system to crash, sometimes needing a complete SD card re-etching.

To prevent this, a commands *vdma.readchannel.start()* and *vdma.readchannel.stop()* must be used before and after writing data to memory. Testing showed that the start/stop calls have close to no effect on speed and effectivity.

5.2.2 Testing DPI video data using Raspberry Pi

Raspberry Pi is capable of transmitting DPI video data using GPIO pins. The resolution and framerate are customizable and for these reasons, Raspberry Pi is used to test the whole system.

During testing, no bugs or errors were discovered.

Test results

System was tested with resolutions defined in table 1.1. Test results are shown in the table and in the graph.

Resolution	Time for one frame [ms]	Average FPS of 60 frame video
100x100	354	18.07
200x200	362	18.16
500x400	357	18.12
960x544	470	18.10

Table 5.1: Results of Raspberry Pi video data testing in table [source: author]

Results show that the system effectivity when capturing video is not affected by selected resolution. However, when capturing a single frame, the resolution causes the system to slow down by an insignificant amount.

5.2.3 Reading real display data

Final testing of FPGA and frame gathering subsystem is testing with real display. For this purpose colour display with resolution 320x240 was used.

DPI timing error

When using read display, timing errors discussed in chapter 2.3.3 and shown in figure 2.13 occurred. The result of this error is shown in figure 5.5.

(a) Display with incorrect timing data

(b) Display with correct timing data
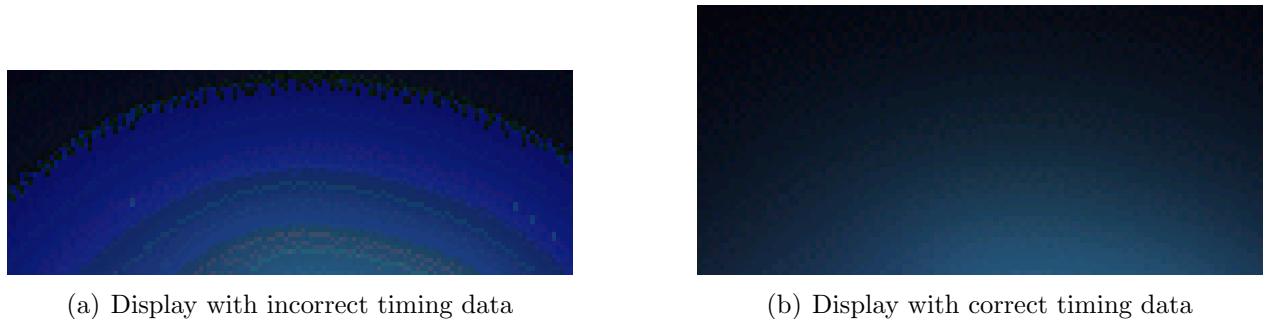
Figure 5.5: DPI timing error in real display data [source: author]

This error was fixed by implementing dynamic timing correction (DTC) system. This system works as shown in figure 2.14 with an exception of dynamic clock delay. The DTC subsystem will determine the frequency of connected display and adjust the timing correction accordingly. If the dynamic correction is not enough, a delay variable can be set by user.

5.2.4   Image data output

The speed of sending data to the user was also tested. Tested variables were image format sent and format of multiple frames.

Image format

After the PS receives raw frame data, they must be converted into usable image format. The image is saved and manipulated using *Image* package in Python script.

Three image formats were considered:

- JPEG

- PNG

- BMP

JPEG format was immediately excluded from the options for its automatic loss compression. This means that the JPEG format saves the image with size reducing algorithms that have the unwanted effect of reducing the quality and precision of the transmitted data. This approach might be used when streaming video but the system was not designed to handle such tasks.

PNG format is a good choice for its lossless compression. However, PNG supports transparent images hence adding more complexity to transmitted data which is not needed since displays are not transparent. PNG format represents raw data with exact precision and thus the PNG image will contain the quality needed and correct representation of raw data.

BMP is format similar to PNG with the difference of support for transparency. BMP also doesn't use compression by default. This causes the BMP images to be overall larger than PNG images. For data representation, the BMP format can as PNG format represent the raw frame values correctly and without any loss in precision.

After testing all formats, BMP was chosen. When raw frame data were processed the BMP format proved to be the most effective in speed and quality. Figure 5.6 shows the comparison of PNG and BMP speeds.

The results show that speed of BMP and JPEG format is nearly the same. Since JPEG uses image compressed and thus loses some quality. Only BMP format is recommended.
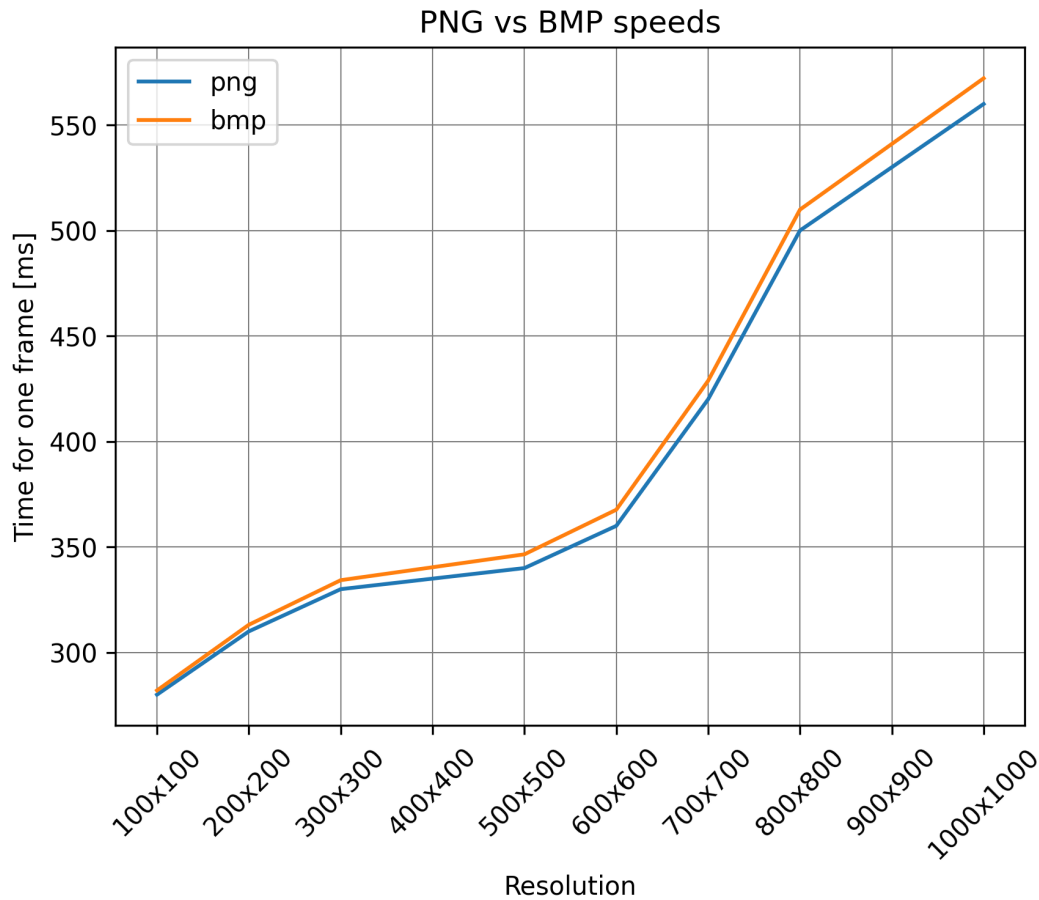
Figure 5.6: PNG vs BMP image format processing and transmitting speeds [source: author]

Multiple frames format

If multiple consequent frames are required a one frame reading method could be less effective since the images are being processed right as they are requested.

To speed up this process a multiple frames request method was created. This method ensures the maximum frame gathering speed. This process captured raw frames into a long buffer. After the requested number of frames is captured, the processing of each will start.

The frames are then sent to the user as a list of frames. The speed differences of single frame requesting and multiple frames requesting are listed in table 5.2.

Resolution	Time for one frame [ms]	Mean time of one frame for 60 frames [ms]
100x100	300	48
200x200	320	50
600x400	400	52
960x544	450	55

Table 5.2: Single vs multiple frames gathering speeds [source: author]

5.3 Opportunities for improvement

The whole system was optimized as much as possible in given time. The timings of all steps are shown as a pie chart in figure 5.7 and table 5.3 compares the minimum system requirements to actual system capabilities.

Resolution	minimum FPS	actual FPS
100x100	15	24
500x400	14	20
960x544	12	18
1280x720	10	15

Table 5.3: Minimum requirements compared to real capabilities (using multiple frames function) [source: author]
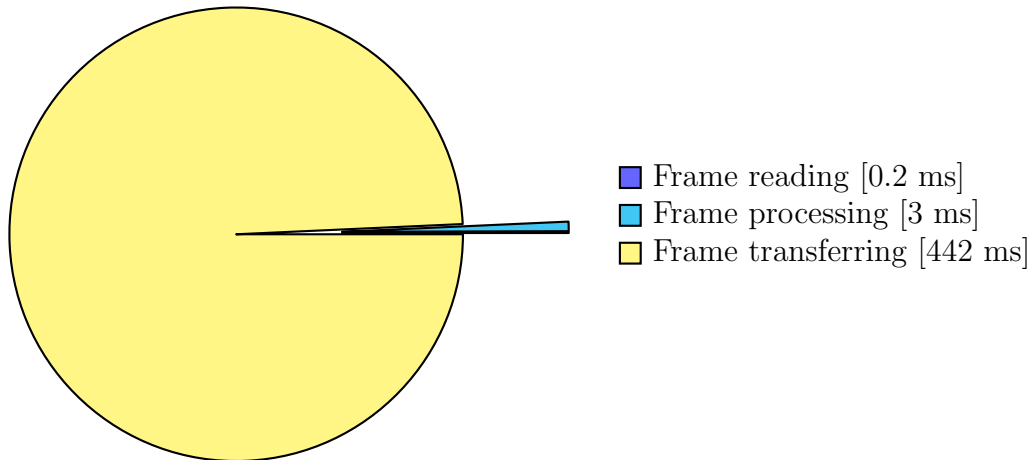
Figure 5.7: Overall timing pie chart for 960x544 resolution [source: author]

It's obvious from these timings that the transmitting is the step that slows down whole system the most.

5.3.1 Compressing data in FPGA

To send a single frame faster and more effectively a compression on FPGA side of PYNQ system should be used. Compressing the raw data instead of compressing data on PS size of PYNQ will affect the speed minimally.

Compressing image data in FPGA is not an easy task, and since the current solution satisfies the minimum requirements this upgrade was not implemented. By compressing, the data sending of frame should be faster.

5.3.2   Video streaming and video compression

If the user requests a video or many frames in a rapid succession, the current system will not be as effective as it could be. Current solution using FastAPI might prove to be too slow for video streaming.

Compressing video in processing system with the combination of compressing frames in FPGA will result in extremely small sized packages that can be sent faster. Also, a new method of transmitting data to user should be used. Instead of using APIs, maybe direct TCP or UDP protocols might increase the maximum FPS provided by the system.

6 Conclusion

In this thesis I have analyzed the available options for quality control of image outputs of embedded devices that are in a form of displays. I have analyzed different types of displays and ways to acquire data in a form that can be accepted by the test framework.

These methods were then analyzed and the option of acquiring image data through frame grabbing hardware was selected.

To implement the frame grabbing hardware (display sniffer), I used the PYNQ-Z2 development board which main processing unit is a powerful Artix-7 manufactured by Xilinx. This processor is able to implement FPGA circuit subsystem and processing subsystem unit, which is capable of running a Linux-based operating system.

These two seemingly separate subsystems are able to communicate with each other due to the internal layout of the processor, and thus various programs written in Python on the operating system side of the Artix-7 chip can take advantage of the flexibility and speed of the FPGA circuit.

This advantage is leveraged in the form of a division of tasks between the two subsystems, where the FPGA circuit is responsible for collecting the image data and storing it in the shared memory of the Artix chip. The server running on the operating system side is then able to access the almost complete data from the FPGA side. This data is then post-processed and sent to the user via the internet implementing fastAPI Python module.

The FPGA circuit, in addition to collecting graphical data, is able to identify the display, its resolution and also forward this information to the user for an easier setup of the system.

The system limits and minimum requirements were checked as part of the work. The objectives of the work and the minimum requirements were met.

At the end of the thesis, the shortcomings of the system were evaluated. These problems are time-consuming and not required to meet the minimum requirements. The thesis has potential for further development of the system in the form of compressing image data on the FPGA side, implementing a more efficient method of sending finished image data from the system, or implementing CI/CD pipelines directly on the PYNQ-Z2 board and thus skipping sending data through the Internet altogether.

All the objectives of the thesis have been achieved and some have been extended or fulfilled beyond the minimum requirements.

List of Figures

List of Tables

List of source codes

Abbreviations

DPI	Display parallel interface
SPI	Serial peripheral interface
I2C	Inter integrated circuit
USB	Universal serial bus
FPGA	Field-programmable gate array
FPS	Frames per second
AXI	Advanced extensible interface
IP	Intellectual property
VDMA	Video direct memory access
OS	Operating system
TPG	Test pattern generator
DTC	Dynamic timing correction

Bibliography

[1] Sciencedirect, 2023. Online. Sciencedirect.com. Available at: `https://www.sciencedirect.com/topics/computer-science/liquid-crystal-display`

[2] TSUJIMURA, Takatoshi, 2017. Oled display fundamentals and applications. 2017. John Wiley.

[3] LCD Display - Fundamentals, 2022. Online. GEORGE, Ligo. Electrosome. Available at: `https://electrosome.com/lcd-display-fundamentals/`

[4] 16x2 LCD displej 1602 modrý + I2C převodník, 2024. Online. In: Laskakit. Available at: `https://www.laskakit.cz/16x2-lcd-displej-1602-i2c-prevodnik/`

[5] AMOLED or TFT: a new choice for display designers, 2017. Online. In: Andersdx. Available at: `https://www.andersdx.com/advantages-of-amoled/`

[6] Samsung S8+ AMOLED display under the microscope, 2019. Online. In: Hackaday. Available at: `https://hackaday.io/project/166935-optical-scanning-microscope/log/170202-samsung-s8-amoled-display-under-the-microscope`

[7] LTD-4608JF, 2024. Online. Mouser. Available at: mouser.cz

[8] Peng Fei Bai, Robert Andrew Hayes, Mingliang Jin, Lingling Shui, Zi Chuan Yi, Li Wang, Xiao Zhang, and Guofu Zhou, "Review of Paper-Like Display Technologies (Invited Review)," Progress In Electromagnetics Research, Vol. 147, 95-116, 2014. doi:10.2528/PIER13120405

[9] Sitronix, 2002. Online. ST7920: Documentation. Online. 3rd edition. Available at: `https://www.laskakit.cz/user/related_files/st7920_chinese.pdf`

[10] LITWILLER, Dave, 2001. CCD vs. CMOS: Facts and Fiction. Online. In: Duke. Available at: `https://courses.cs.duke.edu/fall11/cps274/papers/Littwiller01.pdf`

[11] SONY, 2018. IMX477-AACK: Documentation. Online. Sony Semiconductor Solutions Corporation. Available at: `https://www.sony-semicon.com/files/62/pdf/p-13_IMX477-AACK_Flyer.pdf`

[12] The Bayer arrangement of color filters on the pixel array of an image sensor, 2023. Online. In: Wikipedia. Available at: `https://en.wikipedia.org/wiki/Bayer_filter#/media/File:Bayer_pattern_on_sensor.svg`

[13] TUL PYNQ™-Z2 board, 2018. Online. Tulembedded. Available at: `https://www.tulembedded.com/fpga/ProductsPYNQ-Z2.html`

[14] XILINX, 2020. Zynq-7000 SoC: Documentation. Online. V1.21. Available at: `https://docs.xilinx.com/v/u/en-US/ds187-XC7Z010-XC7Z020-Data-Sheet`

[15] Linear interpolation approach to demosaicing, 2019. Online. In: Illinois.edu. Available at: `https://slazebni.cs.illinois.edu/spring19/assignment0.html`

[16] Background Information on CCD and CMOS Technology, 2023. Online. In: www.tedpella.com. Available at: `https://www.tedpella.com/cameras_html/ccd_cmos.aspx`

[17] Top Considerations When Buying A Microscopy Camera: PART 6: Global vs. Rolling Shutter, 2022. Online. In: Accu-scope.com. Available at: `https://accu-scope.com/news/top-considerations-when-buying-a-microscopy-camera-part-6-global-vs-rolling-shutter/`

[18] Pynq online documentation, 2022. Online. In: Pynq.readthedocs.io. Available at: `https://pynq.readthedocs.io/en/v2.3/pynq_overlays.html`

[19] XILINX, 2024. Test Pattern Generator: Documentation. Online. In: Xilinx.com. Available at: `https://www.xilinx.com/products/intellectual-property/tpg.html`

[20] Raspberry Pi Zero WH, 2024. Online. In: Rpishop.cz. Available at: `https://rpishop.cz/raspberry-pi-zero/685-raspberry-pi-zero-wh.html`

[21] Image distortion, 2022. Online. In: Www.image-engineering.de. Available at: `https://www.image-engineering.de/library/image-quality/factors/1062-distortion`

[22] BERKENFELD, Diane, 2024. Chromatic Aberration. Online. In: Www.nikonusa.com. Available at: `https://www.nikonusa.com/en/learn-and-explore/a/products-and-innovation/chromatic-aberration.html`

[23] FUSTER, Eduardo, 2024. Chromatic Aberration: What is it and How to Avoid it. Online. In: Photoworldtours.com. Available at: `https://photoworldtours.com/chromatic-aberration/`

[24] Compilation process in c, 2021. Online. In: Javatpoint.com. Available at: `https://www.javatpoint.com/compilation-process-in-c`

[25] FastAPI: Documentation, 2024. Online. In: Fastapi.tiangolo.com. Available at: `https://fastapi.tiangolo.com/`

[26] LIPTÁK, Samuel, 2024. Display-sniffer-client-files: Source Files. Online. In: Gitlab.com. Available at: `https://gitlab.com/Lipetka/display-sniffer-client-files`