BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF RADIO ELECTRONICS

ÚSTAV RADIOELEKTRONIKY

DESIGN OF ADAPTIVE WIRELESS VIDEO AND DATA TRANSMISSION

NÁVRH ADAPTIVNÍHO BEZDRÁTOVÉHO PŘENOSU VIDEA A DAT

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. Tomáš Lorenc

SUPERVISOR
VEDOUCÍ PRÁCE

doc. Ing. Tomáš Götthans, Ph.D.

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Elektronika a komunikační technologie**

Ústav radioelektroniky

Student: Bc. Tomáš Lorenc ***ID:*** 206777

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Návrh adaptivního bezdrátového přenosu videa a dat

POKYNY PRO VYPRACOVÁNÍ:

V teoretické části práce prostudujte technologie bezdrátového přenosu dat a videa, která mohou pracovat v ISM pásmech. Předpokládejte minimální rozlišení videa 720×480 při 15fps. Při návrhu přenosu, uvažujte též zpětný kanál. Dále prostudujte možnosti zpracovávání videa (komprese) v jednotce NVIDIA Jetson Nano. Pro kompresi zvolte vhodný kodek, výběr zdůvodněte. Realizujte hardware navrženého řešení (návrh, osazení).

Realizovaná zařízení zprovozněte – vysílací a přijímací jednotku. Vytvořte firmware pro obě strany. Uvažujte, že bude vysílací jednotka připojena k NVIDIA Jetson Nano a přijímací jednotka bude připojena k osobnímu počítači. Vytvořte uživatelské rozhraní pro počítač a implementační knihovnu pro NVIDIA Jetson Nano.
Dodatečné informace (doporučení):
Doporučené znalosti: práce s programy CAD, návrh obvodů, schopnosti programovat v MATLABU, C a Python, znalosti v oblasti zpracování signálů, znalosti v oblasti měření.

DOPORUČENÁ LITERATURA:

[1] DOBES J.; ZALUD V. Moderní radiotechnika, BEN - technická literatura, ISBN: 9788073001322

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 22.5.2023

Vedoucí práce: doc. Ing. Tomáš Götthans, Ph.D.

doc. Ing. Lucie Hudcová, Ph.D.
předseda rady studijního programu

ABSTRACT

The aim of this thesis is to build a device that will be able to establish a wireless transmission of a real-time video stream and send the video to a computer where it will be displayed. The device is powered by the NVIDIA Jetson Nano and video is streamed wirelessly through a WiFi interface. In theoretical part described the problem of video transmission and video compression. The used codecs are h264, h265, and VP8. A subjective test to find the minimum acceptable value of video quality was performed. Measurements have been made to find the best codec for real-time video streaming. Measurements were also performed in real conditions. In conclusion, the results of the measurements and the selection of the best codec for real-time video streaming are commented.

KEYWORDS

Real-time video streaming, WiFi video streaming, codec description, h264, h265, VP8, Video, Nvidia Jetson Nano, RTP, GStreamer, USB Camera capture, GstShark, Outdoor measurement conditions, Subjective test.

ABSTRAKT

Cílem této diplomové práce je sestavit zařízení, které bude umět vytvořit bezdrátový přenos real time videa a posílat video do počítače, kde bude zobrazeno. Zařízení je postavené na jednotce NVIDIA Jetson Nano a vysílání videa probíhá bezdrátově pomocí WiFi rozhraní. V teoretické části je popsána problematika přenosu videa a komprese videa. Používají se kodeky h264, h265 a VP8. Byl proveden subjektivní test pro nalezení minimální akceptovatelné hodnoty kvality videa. Bylo provedeno měření, které odhalí nejlepší kodek pro real time přenos videa. Měření probíhalo i v reálných podmínkách. V závěru jsou komentovány výsledky měření a výběr nejlepšího kodeku pro real time video přenos.

KLÍČOVÁ SLOVA

Streamování videa v reálném čase, streamování videa přes WiFi, popis kodeků, h264, h265, VP8, Video, Nvidia Jetson Nano, RTP, GStreamer, snímání kamerou USB, GstShark, venkovní podmínky měření, subjektivní test.

LORENC, Tomáš. *Design of adaptive wireless video and data transmission*. Brno: Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav radioelektroniky, 2023, 69 p. Master's Thesis. Advised by doc. Ing. Tomáš Götthans, Ph.D.

Author's Declaration

Author:	Bc. Tomáš Lorenc
Author's ID:	206777
Paper type:	Master's Thesis
Academic year:	2022/23
Topic:	Design of adaptive wireless video and data transmission

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would especially like to acknowledge my supervisor, doc. Ing. Tomáš Götthans, Ph.D. for his valuable advice, and support with multiple issues I had through the whole way of making this thesis.

Contents

List of Figures

List of Tables

Introduction

Nowadays, it's a matter of course for everyone to play videos on their devices. Playing video has become part of our lives. But to play the video on the user's device, there are a lot of operations behind it. The main reason is that video is very bit rate intensive. And that's why it is necessary to use tools that can reduce the bit rate without degrading the resulting video quality. These tools are called codecs, and their function is to modify the video to make it size as small as possible.

There are two ways to play video. Video can be playback from a recording file or in real-time mode. For example, playback from a recording can be when the video is played from a file stored on a local drive or playback on some internet TV e.g. youtube. It doesn't matter how long it took to encode the video and how long it takes to decode and how large the latency is. The most important thing is that the final quality is as good as possible. Real-time mode means that when the video is captured, it is immediately displayed on the monitor. In most cases, the video source is the output from the camera. A real-time system is more difficult than a system that plays videos from a file. The real-time system has the opposite requirements than a system playing from a file. The latency between video capture and display must be as short as possible regardless of the quality of the video. The problem is more complicated if it is by a wireless connection. An example of the use of real-time video can be with security cameras or drone camera.

The aim of this thesis is to build a device that will be able to establish a wireless transmission of real-time video and send the video to a computer where it will be displayed. On the transmitter unit, the video will be created using a camera and transmitted via WiFi wireless technology to a computer. Using this device, several measurements are made to determine which codec and with which parameters are best for real-time video streaming. The best codec are used to create the final device. The application of this device will be, for example, that it will be attached to a robot or a car and will transmit video from a camera that should capture the scene in front of the vehicle. This means that the transmitter unit will move during the transmission. To ensure that the transmitter has a sufficiently powerful video compression chip, a single board NVIDIA Jetson Nano computer will be used, which has a hardware accelerator for h264, h265, and VP8 codecs. By using a powerful unit, it will also be possible to use it for other tasks in the future, such as obstacle detection.

In the theoretical part, the possibilities of video compression will be explored. An introduction to video is first described. Then the principle of video compression will be described step by step as they occur in the codec. It will describe all the most important parts of compression, which are used by all modern video codecs.

It will be explained what the differences between h264, h265, and VP8 codecs. The next chapter in the theoretical part is focused on the protocols used for video sharing over an IP network. The greatest attention will be given to the description of the principle of the RTP protocol, which is used for real-time transmissions of multimedia content. The final chapter in the theoretical part will be about the design of a device for testing real-time video streaming. The NVIDIA Jetson Nano computer will be described. The NVIDIA Jetson Nano will be used as a transmitter unit, where the camera video capture and video encoding will be done. For work with video under Linux, the GStreamer framework will be used. The functionality of the GStreamer framework will be described in detail. A software program will be created for the Jetson unit and computer. Several tests will be performed using the device to find the best codec for real-time video streaming. The measurement will also take place in real conditions.

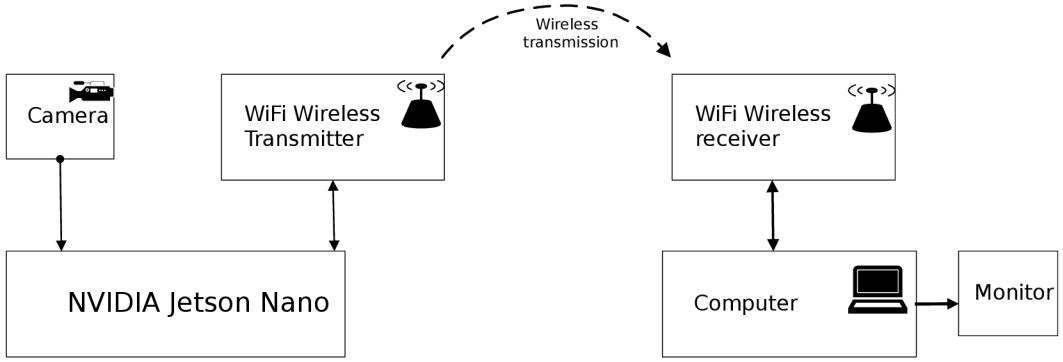


Fig. 1: Device concept block diagram

1 Theoretical Part

1.1 Video processing

The digital video consists of a group of many images that are quickly projected to the viewer. These images are called **Frames**[1]. Due to the imperfection of the human eye, the human brain perceives this projection as a moving image. The faster the Frames are projected, the smoother the motion seems for viewer. This speed is called the **Frame Rate**. It is one of the main parameters of the video. Another important parameter for streaming video is the **Data Rates**. This depends on the frame rate and the resolution of the frame. The **Resolution** of a frame indicates how many pixels are in rows and lines. **Pixel** means the smallest color element in the image and contains a numeric code of color. Color is represented by three colors: red, green and blue (RGB). Almost any color can be created by using different combinations of these three colors.

1.1.1 Compression Parameters

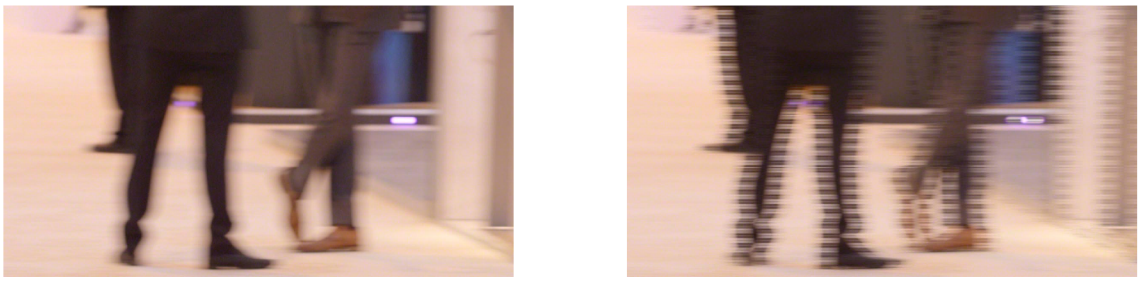
Frame Rate

The frame rate is a parameter that represents how many static frames are projected per time[2]. The most common unit of time is one second, and that's why the basic unit is **FPS** (Frames per Second). The higher the FPS, the smoother the movement appears to people. But if the value is too large, it will be very demanding on computing performance. If the value is too low, the viewer sees the choppy and flickering video. The frame rate must be at least 8 FPS for smooth video perception[1]. For example, most movies have 24 FPS.

Data Rate

The data rate indicates how many data bits are transferred per time. This parameter is important for the design of the communication link. The data transfer rate is typically represented in bits per second (bps), and the amount of bits depends on the quality of the frames and frame rate. The compression efficiency is also determined by the ratio of the original to the new data rate value.

As the scene in the video changes over time, the data rate also changes. This is called **Variable Bit Rate (VBR)** [1]. The advantages of VBR are better quality and better optimization of storage space. Use more bits in a complex scene and use fewer bits for a simple scene to save disk space. But for the playback player and the storage disk, VBR is processing very intensively. The video output can be switched



(a) Progressive (b) Interlaced

Fig. 1.1: Example of Interlaced Error artifacts [3]

to the opposite mode, namely **Constant Bit Rate (CBR)**. At a constant bit rate, the video quality must change over time to keep the same value of bit rate. CBR is very often used because the required data capacity of the communication link can be designed in advance. Though the resulting video quality is reduced and the storage capacity is not optimal (for simple scenes the storage capacity is wasted).

1.1.2 Interlaced / Progressive Video

Interlaced

Interlaced technique[5] is a video distribution and display technique that reduces bit rate and doubles frame rate. The interlacing technique consists in not broadcasting the whole frame at once. The first frame only odd lines are transmitted and the second frame only even lines are transmitted. Again, the imperfections of the human eye are exploited, which cannot detect these missing lines. The disadvantage of this technique is that it creates error artifacts in a very fast moving scene because the fast object is in different positions in each frame. An example of an error artifact can be seen in the figure 1.1.

Progressive

Progressive technique of video distribution is opposite of Interlaced technique. Progressive technique transmits all line of all frames. This method of transmitting is demanding in terms of data rate, but is nowadays already used by modern televisions and computer monitors.

The method of video distribution can be determined by the letter after the resolution value. If there is an "i" (e.g. 1080i), it means that the interleaved method[1] is used and the "p" (e.g. 720p) means the progressive method.

1.1.3 Video Compression

Uncompressed video recording requires a huge amount of data rate. Let's take a simple case for a 1280x720 video at 60 FPS. If you wanted to broadcast this video, the bit rate calculation would be as follows:
$1280(Width) \cdot 720(height) \cdot 3(colorsperpixel) \cdot 8(bitsperpixel) \cdot 60(FPS) = 1,33Gbps.$
With such a high bit rate, it is not possible to establish a data link, so it is necessary to reduce the video bit rate before sending and saving. For this, a method called video compression is used. It is a process in which the data intensity is reduced[1]. How much the data consumption is reduced depends on the choice of codec and also on the requirement of the final video quality.

The tool that provides data compression is called **CODEC**. Codec is an acronym that stands compressor / decompressor[4]. The compressor converts the data into a form that takes up as little space as possible and the decoder reconstructs the data to its original form. Codecs are very demanding on computing performance, so they are often implemented as a hardware accelerator in processors or graphics cards. There are many codecs exist and it is impossible to say which one is the best. In the case of the NVIDIA Jetson Nano, the MPEG2, MPEG4, H.264, H.265, VP8 and VC1 [11] hardware codecs are implemented in the chip. In general, they can be classified according to the fidelity of the quality of the resulting video, the file size, computing performance requirements, popularity, and licensing[1].

1.1.4 Codec

The basic operation of the compression codec is simple. The codec analyzes the video content and converts it into a form that takes up much less space. However, the codec must be able to convert this compressed video beck into its original form. Compression codecs are mainly divided into **lossy** and **lossless** codecs[1]. With lossless compression, the data can be reconstructed exactly as it was before encoding. The reconstructed data is identical to the original data. An example is the Huffman coding [4] method, where the symbols that occur most frequently in the signal are encoded using shorter code words. Lossless compression is not as efficient, so lossy data compression is used for multimedia applications. After using lossy compression, the reconstructed data will never be identical to the original data. In the case of multimedia data, items that cannot be detected by the human senses are most often removed. Below we will discuss a few basic methods used by the video codecs in the NVIDIA Jetson Nano. The block diagram1.2 shows the basic structure of the codec.

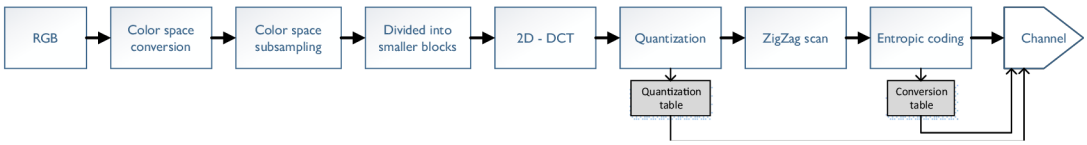


Fig. 1.2: A basic block diagram of Video Compression

1.1.5 Lossy compression

The lossy compression method is a type of encoding that does not need to retrieve exactly the original data during decoded. Lossy compression can reduce the file size several times more than lossless compression[1], while the resulting quality is still satisfactory for the application. Lossy compression is most commonly used for data such as audio, music, and video because it can take advantage of the imperfections of the human senses. This information is called irrelevant data, which means redundant data that will be permanently deleted. Several studies and subjective tests have been conducted to understand the human vision. Based on these, a model **Human Visual System(HVS)** [6] has been constructed to show what the human vision is very sensitive to and what it is less sensitive to. Using this model, the codec determines which components in the image can be completely removed, which components can be compressed at a higher compression level, and which components need to be kept at the best quality.

1.1.6 Human Visual System (HVS)

The human eye is sensitive to wavelengths between 380 nm and 780 nm (visible spectrum)[5]. Any color outside this spectrum is irrelevant and can be removed. The human eye consists of rods and cones. Rods are sensitive to low light levels but only perceive brightness intensity. The cones allow color perception and are best in bright light. The cones are not equally sensitive to all colors in the visible spectrum, but the predominant colors are at 420 nm (blue), 534 nm (green), and 564 nm (red). This is one of the reasons why we describe color using these three colors. A very important fact is that human vision is more sensitive to brightness than to color. Therefore, it is more appropriate to represent the image in the YC_BC_R format than in the basic RGB format[2]. Y stand for luminance (brightness) component and C stands for chroma (colors) components. This format will be described below. Because the eye is more sensitive to brightness than color, color components may be encoded with lower resolution

The HVS model also describes what a person sees more sensitively than less sensitively.[6] The following are some examples of imperfect human vision:

- In fast scenes, the sensitivity of the eye decreases, while in slow scenes it is more sensitive.
- In areas where the difference in color contrast is small, the perception of sensitivity is poor.
- Human vision perceives the finer structures of an image in less detail than coarse structures.

The number of possibilities presented in the HVS model, where human perception of the image is limited, is many times more, but this is not the focus of this thesis. These perception imperfections are used to compress. Structures in the video where the eye sensitivity is high are therefore coded with a lower compression rate and where the eye sensitivity is lower are coded with a higher compression rate.

1.1.7 Color space conversion

As mentioned earlier, HVS is more sensitive to brightness information compared to sensitivity to color information. Therefore, a method is used for image transmission where the brightness component is extracted from the image and transmitted separately. This method of image transmission was already used for analog television. The most widely used color scheme with a separate brightness component is called $\mathbf{YC_BC_R}$ and was part of the NTSC analog television standard[5, 6]. The YC_BC_R color scheme consists of three components. The Y component contains luminance information and corresponds to the gray-level representation of video. The C_B and C_R components contain color information. C_B indicates the colour difference signal between the blue and luminance components. And C_R indicates the colour difference signal between the red and luminance components. The green color is not transferred, but both C_B and C_R components are calculated based on Y, the green color can be calculated and the original RGB values can be restored. An example of how YC_BC_R looks like is in the figure1.3. There are several methods for conversion exist, below is the most used conversion method according to ITU-R BT.601[7].

$$\begin{bmatrix} Y \\ C_B \\ C_R \end{bmatrix} = \begin{bmatrix} 0,299 & 0,587 & 0,144 \\ -0,169 & -0,331 & -0,5 \\ 0,5 & 0,419 & -0,081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{1.1}$$

1.1.8 Color space subsampling

Taking advantage of the imperfection of the HVS, the quality of the color information can be reduced without significant visual loss for the viewer. This makes it possible to use subsampled chrominance formats[2]. Subsampled the chroma component is one of the most basic compression and data-rate reduction technique. It

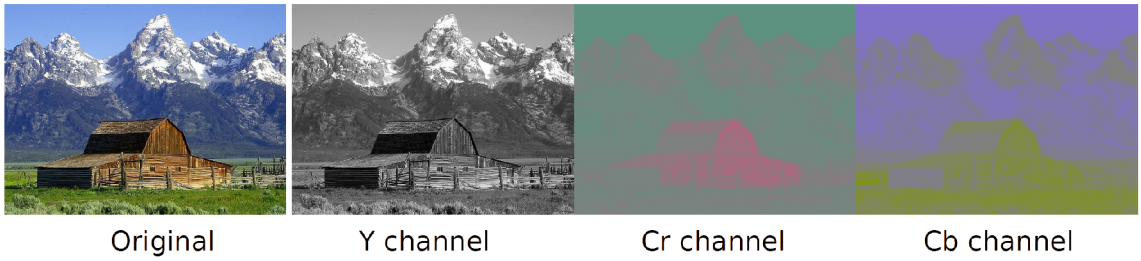


Fig. 1.3: RGB to YCbCr conversion

is used not only for video, but also for static images (JPEG). Subsampling works in such a way that the individual colour points of the chroma component are not all transferred, but fewer. The subsampling used is commonly expressed as 4:a:b ratio and this ratio is defined for an area of 4 x 2 pixels. The first digit indicates the luminance component and in all cases has a value of 4, which means that the luminance component is transmitted at full resolution. The second and third digits (a and b) indicate the number of chrominance samples in the top row and the number in the bottom row. Below is a list of the formats used. The luminance component is always kept at full resolution.

- **Format 4:4:4** - It keeps the chroma components at their original resolution. No subsampled is used, so there is no data compression.
- **Format 4:2:2** - Subsampling the horizontal resolution of the chroma components by half, but the vertical resolution is full.
- **Format 4:2:0** - Subsampling the horizontal, and even vertical resolution of the chroma components by half.
- **Format 4:1:1** - Subsampling the horizontal resolution of the chroma components by a quarter.

1.1.9 Transform Coding – DCT

In the original image, all individual color points are equally significant, so when transferring, all points must be transferred. This method of transmission is not good at all. When the image is subdivided into blocks, it is possible to detect the similarity of each pixels in the small block. This means they have spatial redundancy. To assign different significance to all image points, a method is used that can concentrate the energy into a few coefficients. **Discrete Cosine Transform** (DCT)[5] function is most commonly used in video compression. The result of the transform is to decorrelate the original signal and redistribute the signal energy among a small set of transform coefficients. For image or video frame transformations, the 2D

version of the Discrete Cosine Transform is used and the inverse transform is called the **Inverse Discrete Cosine Transform** (IDCT). The DCT is very similar to the Fourier transform, but the output is real numbers, while the output of the Fourier transform is complex, so the DCT is less computationally demanding. The DCT expresses a finite sequence of discrete frequency points in terms of a sum of cosine functions. Before applying the transformation, the video frame is divided into smaller blocks. The small block size depends on the codec used. For example, for basic JPEG compression, the image is divided into 8 x 8 blocks. Theoretically, it would be possible to transform the whole image at once, but it would be very difficult. DCT transforms each color component separately, so if the image has been converted to the YCbCr color scheme, DCT transforms the three components separately.

The two-dimensional DCT can be expressed as follows:

$$X(u,v) = \alpha(u)\alpha(v) \sum_{m=0}^{M} \sum_{n=0}^{N} x(m,n) \cos\left[\frac{(2m+1)u\pi}{2M}\right] \cos\left[\frac{(2n+1)v\pi}{2N}\right] \tag{1.2}$$

Here, u and v are the horizontal and vertical spatial frequencies, range is same like an input block; $\alpha(u)$ and $\alpha(v)$ are a normalizing factor; $x(m,n)$ is the pixel value at spatial location (m,n). Normalizing factor equal to:

$$\alpha(u) = \begin{cases} \frac{1}{\sqrt{M}} & , u = 0 \\ \sqrt{\frac{2}{M}} & , u > 0 \end{cases} \qquad \alpha(v) = \begin{cases} \frac{1}{\sqrt{N}} & , v = 0 \\ \sqrt{\frac{2}{N}} & , v > 0 \end{cases} \tag{1.3}$$

The decoder uses the IDCT function, which is expressed as follows:

$$x(m,n) = \sum_{m=0}^{M} \sum_{n=0}^{N} \alpha(u)\alpha(v)X(u,v) \cos\left[\frac{(2m+1)u\pi}{2M}\right] \cos\left[\frac{(2n+1)v\pi}{2N}\right] \tag{1.4}$$

When the equations for DCT1.2 and IDCT1.4 function are compared, it can be seen that they are very similar. The same part of both equations is called the **Basic Function of DCT** and looks like this:

$$\alpha(u)\alpha(v) \cos\left[\frac{(2m+1)u\pi}{2M}\right] \cos\left[\frac{(2n+1)v\pi}{2N}\right] \tag{1.5}$$

This part is always constant, it is not recalculated for each point. A constant matrix is created for all samples, which is called the **Basic Function of DCT**. The figure1.4 shows the visualization of the 8x8 matrix Basic Function of DCT. The horizontal frequency increases from left to right and the vertical frequency increases from top to bottom. In the upper left corner the frequency is zero, so the value is constant there (DC coefficient).

The figure1.6 shows an example of power distribution using DCT transform. The left upper figure is the original image that inputs to the DCT function. The

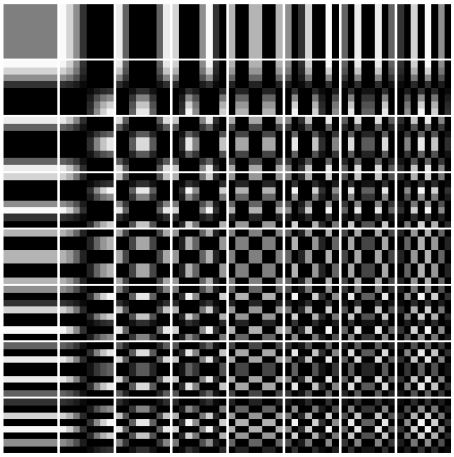


Fig. 1.4: DCT basis functions for an 8×8 input block

table input contains only integer values and each value has a similar significance (energy is evenly distributed). The right upper figure represents the output of the DCT function. The resolution has the same size as the input, but the values are non-integer and the main energy is concentrated in the upper left corner. The DCT function does not reduce the number of bits, quantization and entropy coding must follow.

1.1.10 Quantization

As already mentioned, HVS is less sensitive to high-frequency structures than to low-frequency structures. This feature allows high-frequency information to be truncated without the viewer noticing. For this, a quantization is used, which removes the spatial redundancy in the DCT block[7]. Quantization is a lossy process in which high-frequency information is truncated and cannot be recovered in the inversion operation. There are many types of quantization. The simplest method is division, rounding, and zeroing. In the case of image or video compression, a quantization table is used where the quantization coefficients for all DCT elements are listed. For higher frequency elements, the quantization coefficient is larger because the HVS is lower sensitive. After the DCT is completed, each element is divided by a quantum coefficient from the quantum table according to the specific coordinate. The quantization tables are different, they vary depending on the codec used, what color component it is for, and what compression ratio is set. In most cases, the resulting number is rounded to an integer. Another quantization mechanism is thresholding. If the resulting value is below a specified threshold, it is neglected.

As mentioned, quantization is a lossy process, and if the quantization level is set too high, issues can appear in the resulting video. For example: quantization

(a) Blur (b) Block artifacts

Fig. 1.5: Example of Quantization issues

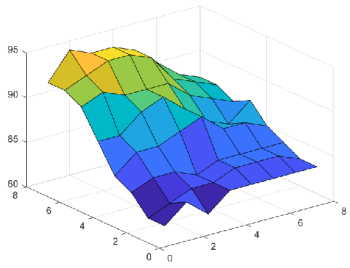
noise, block artifacts or image blur1.5. The figure1.6 shows an example of the power distribution after using quantization and IDCT. The bottom left graph shows the power distribution after quantization is applied. It shows the change in the maximum Z-axis value and the smoothing of the other values. The bottom right graph shows the power distribution after IDFT. The shape is similar to the original image, but the values are more flat.

1.1.11 Entropy Coding

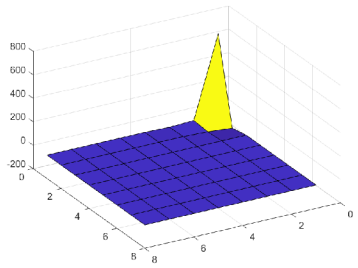
The goal of entropic coding is to encode the transmitted data into a form where it takes up fewer bits, and at the same time, this compression is lossless[7]. This is achieved by using Variable Length Coding (VLC) and a method of reading DCT coefficients. The reading of the coefficient after quantization is done using a ZigZag scan pattern, where the individual coefficients are read diagonally. The result is that a sequence of non-zero values will follow first, and then a sequence of zero values. Therefore, do not transfer a sequence of zeros, but only the count of zeros in the sequence. If there are only zeros until the end of the block, only the EOB (End Of Block) mark is transmitted. The DC component is coded separately so is differentially coded with respect to the previous DC block. The easiest algorithms for entropic coding are Huffman coding and Arithmetic coding. The entropic coder reduces statistical redundancy.

Huffman code

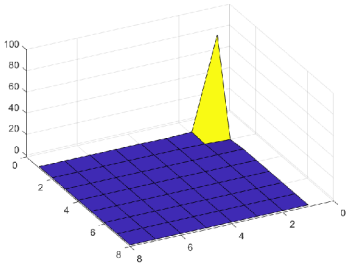
The Huffman coding uses a conversion table where defined all possible input symbols that can occur and an encoding code for them. To create the conversion table, it



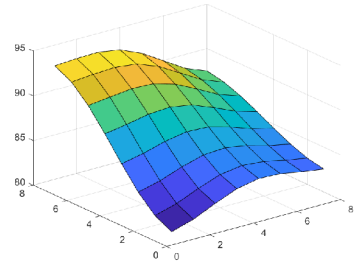
(a) Original Image



(b) DCT



(c) Quantization



(d) IDCT

Fig. 1.6: Example of power distribution using DCT transform and quantization

is necessary to know the probability of occurrence of each input symbol before the encoding begins. Encoding codes are created according to the probability of occurrence of each input symbol. More frequent symbols are represented by a shorter code and symbols that are less frequent are represented by a longer code. Huffman coding is prefix coding, which means that encoders do not need to send how long the code is. The code words are made so that the decoder can recognize where one code symbol ends and where the next one begins. For decoding, the decoder needs to know the conversion table. Because the Conversion Table adapts to the actual input symbols, the table is transmitted with the data. Huffman coding is optimal for symbol-by-symbol coding and the probability of distribution of the input symbols is known in advance. Otherwise, it is better to use other coding methods.

Arithmetic code

Arithmetic code differs from Huffman coding in that it does not separate individual input symbols from each other but encodes the entire message into a single number. The resulting number is a fraction and ranges from 0 to 1. Arithmetic coding assumes that the probability of each symbol occurring is the same. Depending on how the input symbols stream, the resulting number increases or decreases. The requirement is that the decoder must know all possible input symbols that can occur,

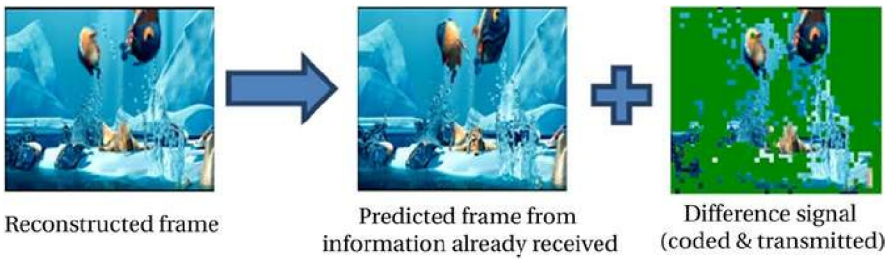


Fig. 1.7: An example of spatial redundancy in a video frame[5]

their combinations and their count. For more efficient coding, probability models are also used, which can adapt to the current stream of input symbols during the coding process. Arithmetic coding is also prefix code.

1.1.12 Predictive Coding

The neighboring frames in the video stream are very similar. This similarity can be used in video compression. This is a compensation of spatial redundancy using inter-frame prediction coding, where the similarity of neighboring frames is exploited[4, 5]. For example, in TV news, where the presenter is in the studio and just reads the news, most of the screen is the same over time and only the presenter's face is changing. Figure 1.7 shows an example of spatial redundancy present in a video frame. Therefore, there is no need to send the whole frames, but only changes to neighbor frames. Only the different information is enough for the decoder to be able to reconstruct the frames. Thanks to this method, the compression ratio can be increased several times, but it also increases the complexity of the conversion. The neighboring frame doesn't only have to be the previous frame, it can also be the future frame. Simultaneously, it is not necessary to refer only to frames that are in close contact, it is also possible to refer to frames that are far. There are many methods that are used. Some methods compare individual pixels, some compare energy distribution after DCT transformation, and others compare color similarities and individual objects in the scene. The method of comparing individual pixels is called Differential Pulse Codemodulation (DPCM) technique. Compares neighbor pixels across frames and sends the difference using a variable-length code.

A popular prediction coding method is **Group of pictures** (GOP). This method was first used with the MPEG-1 codec. Predictive coding is used between frames only within this group. The count of frames in the GOP is determined by the codec. The advantage is that if the decoder receives one GOP corrupted, it will not affect the other GOPs. And it also makes it easy to jump across in the video. Frames in

the GOP block are divided into three types:

- **I-frame (intra-coded frame)** - This frame is compressed without dependencies on other frames and does not need any additional data to reconstruct it. It is required that the GOP block contains at least one I-frame.
- **P-frame (predictive coded picture)** - The frame is encoded using predictive coding. For decoding, it is necessary to know the form of the reference frame. The reference frame can be only the previous frame and only one the I-frame or P-frame.
- **B-frame (bipredictive coded picture)** - The frame is encoded using predictive coding. Can use multiple frames to reference. Reference frames can be either earlier or later. This increases the efficiency of compression, but also increases the complexity of decoding. B-frame cannot be used as a reference frame.

As a result, these images are compressed with high data reduction. Before the GOP block is transmitted, the frames are sorted in a queue according to which frames are referenced to which. Older codecs had a strictly defined GOP block structure (number of individual frames, frame order, number of reference frames, etc.). Newer codecs have much more flexibility in modifying the GOP block structure. If the parameters are selected incorrectly, boundary artifacts may appear in the resulting video.

In a video stream, there are objects or blocks that are the same but have only changed their position in the frame. This is also used in video compression and only the location coordinates of the same block in the reference frame and its movement are transmitted. The movement is called **motion vectors** and defines the movement in the vertical and horizontal dimensions. The process that finds the positions of similar blocks is called motion estimation. Divides the frame into smaller blocks (macroblocks) that can be of various sizes. It then searches for similar blocks across frames and determines motion vectors. It does not have to be the whole block, it can be a combination of several blocks or just part of them. The motion estimation technique is complex for both encoder and decoder, but greatly reduces the data rate.

1.2 Codec in NVIDIA Jetson Nano

The Jetson Nano unit includes three hardware encoders for h264, h265 and VP8 codecs [11].

1.2.1 H.264/MPEG-4 AVC

The h264 code was published in 2003[4, 6]. It is a collaboration between MPEG (Moving Picture Experts Group - ISO) and VCEG (Video Coding Experts Group - ITU). The codec has several names for historical reasons: H.264, AVC (Advanced Video Coding) or MPEG-4 Part 10. Compared to older codecs, the h264 codec is already designed only for compressing video in a rectangular format. The goal of the codec was to be at least 50% more efficient than previous codecs, and to do so without increasing the complexity too much. And this was also achieved. The principle of the codec is based on the previous MPEG-4 Part 2, MPEG-2, H.263, MPEG-1, and H.261. It may seem that the codec h264 is very old, but it is still very much popular. Which also corresponds to the fact that it's still being updated and got its last update in 2021. The h264 codec is used in almost every application: video storage, DVD and Blu-ray discs, Internet TV (YouTube, iTunes), online video calls, and broadcasts TV over terrestrial, cable, and satellite channels.

The codec uses similar features as its previous codecs. Only a few features will be listed. It introduces a hybrid DPCM block coding technique that reduces the spatial and temporal domains. Improves entropic coding with adaptive word length and statistical correlations. Improves movement prediction and movement vectors. The encoder introduces the ability to adaptively size macroBlocks. Improves the encoding of B and P frames that can reference multiple reference frames. Improves Inter-Coding where 16x16, 8x8 or 4x4 blocks can be predicted from blocks in the same frame.

1.2.2 H.265/MPEG-H HEVC

The codec was published in 2013[4, 6]. The codec has several names for historical reasons: H.265, HEVC (High Efficiency Video Coding) or MPEG-H. It is the successor to the h264 codec, and compared to it achieves about 50% better efficiency. This has been achieved by many times more complex algorithms, which required more computer power. The basic version of the codec supports up to 8K resolution. The h265 codec is a licensed product. It is still less popular than its previous h264. It is used in many area, for example video storage, Internet TV, Internet video calls, or terrestrial DVB-T2 TV broadcasting.

The algorithm is again based on its previous design. To achieve even higher compression ratios, most techniques are modified, but only a few will be listed here. The biggest change was the size of the blocks. In the h264 codec the largest block was 16x16, but in the h265 codec the largest block is 64x64. The change can be seen in the image 1.8. Improvements have been made to the prediction, coding, and segmentation of I-frames. Improved entropy coding. Improved reconstruction filter

to reduce artifacts. Intra-frame block prediction was improved by increasing the number of possible motion vector directions from 8 to 33. For faster coding process, multi-threading has been a lot improved.

Fig. 1.8: The difference of splitting the frame into blocks [4]

1.2.3 VP8

This is an open-source codec from 2010[4, 6]. This codec can be used without paying a license fee. It was developed mainly for the needs of web applications. It has been developed for online real-time video calling and Internet TV (e.g. YouTube). The codec is supported by all web browsers. For storage purposes it also developed the Matroska video format. However, when creating a new device, it is preferable to use the successors, which are VP10 or AV1.

The codec is much simpler than the h264 and h265 codecs. Only supports progressive video in YUV 4:2:0 format. It uses similar principles as the h264 codec. It also uses the DCT transformation, but the blocks are strictly sliced to a defined size. It also uses intra-predictive block coding in frames using moving vectors but moving vectors have limited precision. It also uses I, B and P frames to predict the frames in the stream. It widely supports multi-threaded processing.

1.3 Protocols transmission

1.3.1 UDP/TCP

For communication between the video source unit (NVIDIA Jetson Nano) and the receiver (Computer), a communication network with TCP/IP protocol will be used, in which two transport protocols are most used[2]. The first one is UDP (User Datagram Protocol), which implements an unreliable data link. This means that

the sender has no information about the condition in which the message was delivered to the receiver. The advantages of this link are: lightweight, low link capacity requirements, and short response time. It is therefore suitable for streaming multimedia content, where a several packet loss with video or audio will not matter. The opposite of UDP is TCP (Transmission Control Protocol). TCP implements reliable transmission where a channel is established between the sender and the receiver before the actual transmission begins. The TCP protocol guarantees delivery of messages, delivery in the correct order. The disadvantage is higher latency and therefore not suitable for streaming multimedia content. The most used transport protocol in practice is UDP for Real-Time transmissions.

1.3.2 RTP

The main part of this project is transferring video from the Jetson unit to the receiving computer. This is called Streaming Videos. For Streaming Videos in an IP network, the RTP (Real-time Transport Protocol) is most used[2, 4], which provides for the end-to-end transfer of data in the shortest possible time. The protocol is designed to transfer packets of multimedia content over an IP network in real-time and provides mechanisms for jitter compensation, packet loss tolerance, and synchronization. Since UDP packets are used for transmission, packet loss is highly probable. RTP also supports the possibility of using the TCP transport protocol, but this method is not usual in practice.

The RTP protocol divides the multimedia data stream into separate packets and assigns them a header. RTP only describes the format of the packet, not the way how the packet is transmitted over the network. This is handled by the UDP protocol. A multimedia stream can be video, audio or text. For the receiving side to know what codec the transmitter is using, there is a Payload-Type field in the header which indicates the codec used by a numeric code.

The size of the header depends on the RTP protocol version and specification used. The most popular RTP protocol specification is structured into 11 fields. Only the most important fields will be described below. The complete header can be seen in the table 1.1.

- **Sequence number:** Represents the RTP packet identification number. Each packet has a sequence number 1 number greater than the previous packet. When the maximum value is reached, the value is reset to zero. The sequence number is used to identify packets that were not delivered in the correct order or to determine if a packet was lost.
- **Timestamp:** It is determined by the sampling period of the multimedia content. It is used for synchronization on the receiver side. The timestamp

Tab. 1.1: RTP Packet header[2]

Bit	0-1	2	3	4-7	8	9-15	16-31
Offset	Ver.	P.	Ext.	CSRC count	Mark	PT	Sequence number
32	Timestamp						
64	Synchronization Source Identifier (SSRC)						
96	Contributor Identifier (CSRC)						
:	Payload						
:	:						

number is given by sum of the previous packet's timestamp number and the time to produce the next packet. If the contents of a single frame are divided into multiple packets, the timestamp number will be the same for all of these packets.

- **Synchronization source identifier (SSRC):** Identifies the source of the data flow. It is used when there are multiple active streams between stations. Each source is assigned a random ID number.
- **Contributing Source Identifier (CSRC):** It is used when there are multiple sources of RTP content in the network. Each RTP content source is assigned a random number.
- **Payload Type (PT):** It determines what data is transmitted and what compression is used. The code number is given by a table from RTP defines document.

1.3.3 RTCP/RTSP

RTP only provides transmission from the transmitter to the receiver(s). For a transmitter to respond to packet drops or degrading channel conditions, it needs an RTCP (RTP Control Protocol) feedback channel[2]. RTCP is a monitoring tool that sends quality of service (QoS) information from the receiver to the transmitter, such as the number of packets received, data loss, jitter, etc. It does not transmit any multimedia data. The transmitter can then react, for example, by reducing the bit rate (e.g. reducing the resolution of the video). RTCP usually uses a port number one higher than RTP. The problem can occur with large multicast (multiple receivers) where the RTCP bit rate can exceed the limit. Therefore, RTCP limits the bandwidth so that it does not exceed 5% of the total link capacity.

When using RTP and RTCP, a reliable data stream can be created, but the client receiving the content is unable to control it. By control we mean PLAY, RECORD, PAUSE and so on. The Real-Time Streaming Protocol (RTSP) is used to control the

stream[2]. Communication takes place primarily using the TCP transport protocol and communicates bidirectionally. The communication is classified into commands and responses, so that the other side knows if the command was successfully executed or not.

1.4 Device design

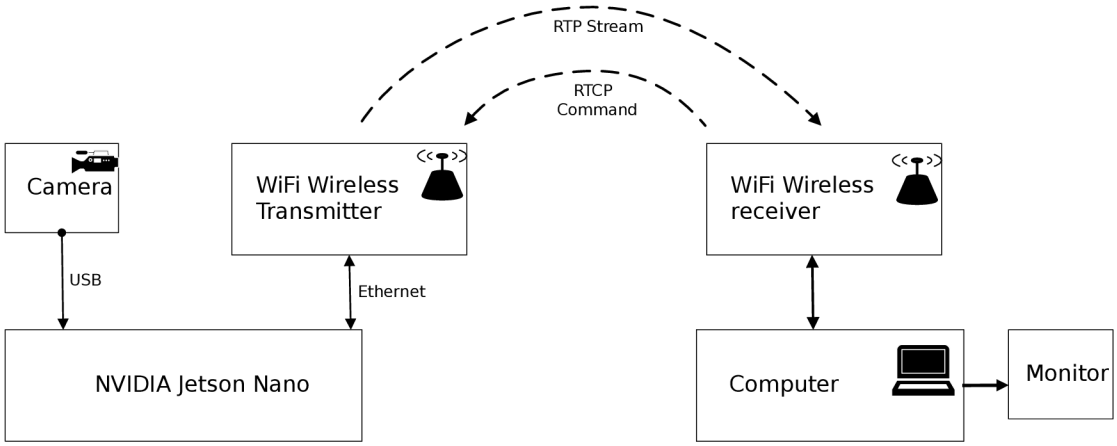


Fig. 1.9: Device block diagram

1.4.1 NVIDIA Jetson series

NVIDIA Company developing a platform called Jetson that focuses for creating embedded systems that need computing power for machine learning and Artificial intelligence[9]. This is a series of processing modules that contain NVIDIA TEGRA mobile chip. Second main advantage is low power consumption and small dimension. The application areas are countless such as object identification, video live processing, high-speed signal processing, machine learning and much more. Thanks for small dimension and low power consumption is perfect choice for car, drones, robots etc.

The NVIDIA TEGRA mobile chip integrates a processing unit (CPU) and powerful graphic unit (GPU) into one package. The graphic unit used in this chip is directly designed for effective processing multimedia and AI (Artificial intelligence). This performance is achieved by using many of hundreds CUDA cores that support parallel computing. This parallel computing is suitable for AI and multimedia processing. Best practical example is screen rendering. The screen is divide to many pieces and each piece is render in parallel[10, 13].

Today is Jetson available in many variants. Jetson Nano has the lowest level of performance around all modules. This module includes a 128-CUDA Core Maxwell architecture GPU unit and quad-core ARM A57 CPU running at 1.43 GHz. The memory size is either 2 GB of 4 GB[10]. For first look this specification is too poor, but it for small AI project is sufficient. High-performance modules are names Xavier and newest Orin. Difference with the Nano is that it has more CUDA and Tensor cores and obviously more memory and better CPU. In marketplace is still possible to buy an old module called TX1 and TX2, but NVIDIA they marked as obsolete and therefore will not be described further. In 2021, NVIDIA company announced Jetson Orin that has 6 time better performance than Jetson Xavier[10]. Detail describe can be found in the table 1.2.

Tab. 1.2: Jetson comparison table[10]

	Jetson Nano	Jetson Xavier NX	Jetson Orin NX
AI Performance	472 GFLOPs	21 TFLOPs	100 TFLOPs
GPU cores	128 CUDA	384 CUDA /48Tensor	1024 CUDA /32Tensor
GPU Architecture	NVIDIA Maxwell	NVIDIA Volta	NVIDIA Ampere
CPU	Quad-Core Arm® Cortex®-A57	6-core NVIDIA Carmel Arm®v8.2	8-core NVIDIA Carmel Arm®v8.2
Memory	2 GB / 4 GB LPDDR4	8 GB / 16 GB LPDDR4x	8 GB / 16 GB LPDDR5x
Storage	16 GB eMMC	16 GB / 32 GB eMMC	64 GB eMMC
Power	5 W / 10 W	10 W / 20 W	10 W / 25 W
Announcement	2018	2017	2021

In this table is describe only main product of Jetson product line and omitted obsolete products. In this thesis use only Jetson Nano therefore will be describe in next paragraph.

1.4.2 Jetson Nano

Connectivity of Jetson Nano is relatively various and fine for most projects. Main communication interface are 1000Mb/s Ethernet port and USB ports. The pity is absence of wireless connectivity over WiFi. It sounds like bagatelle thing, but WiFi speeds up the development of small projects. USB ports are both in version 2.0 and 3.0[8]. The display interface can connect an external monitor via HDMI, DisplayPort or MIPI-DSI, and two monitors can be connected simultaneously. SD card is used for main storage and loading the operating system. A PCI Express port with four lines can be used to connect a high-speed device. For embedded device is very commonly used low-level interface. Jetson Nano equipped UART, SPI, I2C, CSI a GPIO ports. Unfortunately, the CAN Bus is missing between the interfaces. CAN bus is very often used in automotive vehicles[10, 11].

Jetson Nano can work in two performance modes. In energy-saving mode, the consumption is 5W and in performance mode the consumption is 10 W. This difference in consumption is achieved by turning off 2 cores in the CPU and reducing the clock speed in the CPU and GPU[11]. Details are written in the table1.3.

Tab. 1.3: Jetson Nano performance modes[11]

	Performance Mode	Energy-Saving Mode
Power Consmption	10 W	5W
Activate CPU core	4	2
Max. CPU Clock	1479 MHz	918 MHz
Max. GPU Clock	921.6 MHz	640 MHz

Jetson Nano Development Kit

NVIDIA sells a set of development boards for Jetson modules for ease of development of the final product by customers. NVIDIA publishes a detailed description of these boards, and they can be used to build your own product like a template.

The Jetson Nano development board will be used in this thesis. The board provides most of the connectivity that the Jetson module has. Commonly used interfaces include: Ethernet, HDMI, DisplayPort, 4xUSB 3.0, Camera CSI and a 40-pin GPIO connector where there are UART, SPI, I2C[14]. There are two power options. The first option is via the DC jack and the second via the USB connector. On the bottom side of the board is include PCB pads for soldering small battery case if customer want to archived time over power off[12].

Fig. 1.10: Jetson Nano Development Kit[10]

1.4.3 JetPack SDK

NVIDIA provide a software packet for Jetson module called JetPack. Packet Jetpack can be divided into two parts. The first part is the Linux operating system, which is modified to working on the NVIDIA Tegra chip (Linux for Tegra L4T). It's include Linux kernel and also bootloader and drivers. The second part is software libraries (SDKs) that have been optimized to work on Jetson. Primarily contain tools needed to develop a project for AI. In the picture1.11 you can see a list of the most important tools included in the JetPack. In this work, tools from the multimedia container will be used, mainly GStreamer and the video decoding library. All Jetson modules supported the same JetPack and thanks to this you can move your project created on once Jetson to any other Jetson module[11].

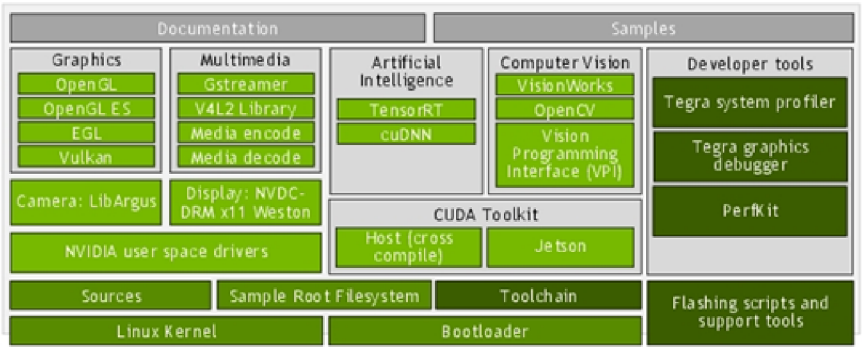


Fig. 1.11: List of the most important tools included in the JetPack[11]

1.4.4 GStreamer

GStreamer is a popular multimedia framework that can be used to create a media processing application[16]. The basic principle of the GStreamer design is the creation of a diagram called a pipeline. It is the connection of a function block (Elements) to a line that represents the flow of data. GStreamer can be used for any multimedia content modification and to provide connection data flow from/to another application. An example of a pipeline can be seen in the picture1.12. Data flow is always one direction and line can be divided and merged. There are three types of elements: sources, filters and sinks[16, 17].

Sources are elements concerned input of data. It always stands as the first element in the pipeline. Such elements are the generation of a video/audio signal, reading from a file of from camera or read stream from the Internet. For example, a source element called videotestsrc can produce a video signal of various formats.

Filter elements have both and outlet pads. They are the elements that change the input data into the final form. An example of such elements is adding a new effect to the audio signal or changing the resolution of the video signal. The second main function is to split pipes and merge them again. These elements are usually called mux and demux. The third important function of filter elements is format conversion. These operations are called encode (conversion to a compressed format) and decode (conversion from a compressed format).

Sink elements are endpoint of the pipeline and always stand at the end. It typically sends data to a sound card, video display, or hard drive[16, 17].

Almost all elements have customizable properties. Some properties are read-only and some can be edited, it depends on the rules. GStreamer makes sure that all function blocks run and communicate with each other after the pipeline completes. It also provides data buffering, queuing ahead of slow blocks, and keeping track of synchronization between blocks. Thanks to this a comprehensive pipeline will be look like one pack processing block and user easy to use.

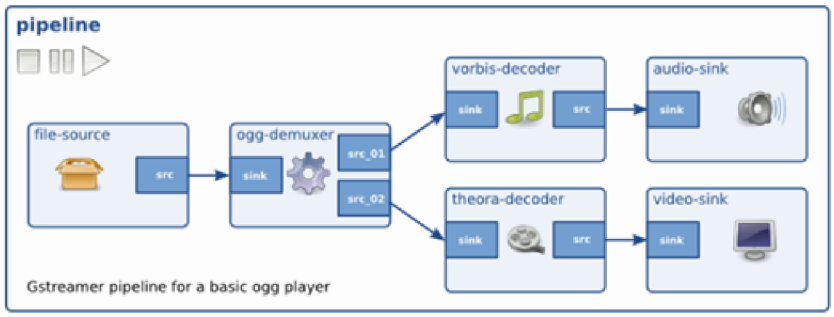


Fig. 1.12: GStreamer pipeline[16]

1.4.5 U-Blox

It was decided to use WiFi at 2.4 GHz for video transmission. The Jetson Nano does not have a wireless transmitter, so it was required to find a device that could provide wireless transmission. Only the Ethernet connector on the Jetson unit can be used for video transfer. So that means that it will have to find a solution to create a bridge from Ethernet to WiFi. The simplest option would be to use a router and switch it to bridge mode. But the planned device is to be battery powered and as small as possible. Since the standard router is difficult to powered from the battery and is relatively large, it was necessary to find an alternative solution. After searching for suitable devices, the following three modules were shortlisted:

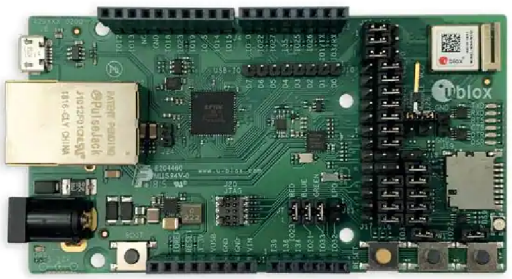


Fig. 1.13: U-blox EVK-NINA-W101 development board[22]

- U-blox NINA-W10,
- U-blox ODIN-W2,
- Microchip WFI32E01PE.

All three modules have very similar parameters. After review, the U-blox NINA-W10 module was selected. The biggest reason for the choice was that the module has an ESP32 processor[21] inside. Since I have previous experience with ESP32 processors, using this processor will be simpler for me. The other modules use an architecture I've never worked with before and would therefore be harder to use.

The NINA-W10 module contains a dual-core ESP32 processor, and the components required for the processor: 16 MB FLASH memory and high-quality crystal. Also includes components for wireless transmission: RF filter, RF amplifier and antenna matching. Thanks to these extra components, the wireless transmission characteristics should be better than with a regular ESP32 module. The module provides support for WiFi transmission at 2.4 GHz with 802.11b/g/n specification[21]. It can transmit with power of 15 dBm (module output without antenna) and receiver sensitivity is -96 dBm. U-blox writes in the datasheet that the module has a maximum throughput of 25 Mbit/s. Of course, this value cannot be trusted, but the value is still many times higher than what is needed for this work. The module does not have a direct input for the Ethernet port, but only an input for the RMII bus. It is necessary to add an Ethernet <-> RMII converter on the PCB. In addition, the module provides many other peripherals, but they will not be in the final product, so they will not be listed. The NINA-W10 version is special in that it is open, and

you can upload custom programs.

After consulting with the supervisor, it was decided to purchase a development board and not to create a custom PCB. The selected development board is EVK-NINA-W101. The development board is equipped with the NINA-W101 module. The NINA-W101 module does not use an internal antenna, but has it connected to a pin and an external antenna can be used. In the box together with the development board there is an antenna (3 dBi monopole), which will be used for measurement.

1.4.6 Camera

A camera will be used to capture the actual video signal. For this work, a video camera from company UP will be used, which works with USB UVC (USB Video Class)[19] protocol. Includes a 2-megapixel sensor[20] and has support across operating systems thanks to the UVC protocol.

2 Implementation Part

2.1 NVIDIA Jetson setup

This project will use an NVIDIA Jetson Nano module as the main transmitting unit and a motherboard will be use a standard development board also from NVIDIA. NVIDIA sells both products as a set called Developer Kit. The Developer Kit includes the module, motherboard and passive heatsink[14], which are already factory assembled. But the developer kit does not include any accessories and you need to buy at least a power adapter and an SD card to get it to work.

2.1.1 Power adapter and SD card

As already mentioned in the theoretical part, the unit can be powered in several ways. The easiest way to deliver power to the unit is through the Micro USB port. The datasheet specifies that the maximum current required for the unit to operate state is 2 A at 5 V[14]. Unfortunately, the USB port does not support Quick Charge function like a modern mobile phone and the unit will not turn on with these types of power adapters. This unsuccessful situation occurred on the first attempt to switch on the unit. Replacing the power adapter with another one fixed it. The unsuitable adapter was replaced by a common adapter with 3 A output current. Another negative aspect of the unit is that it is sensitive to power supply voltage drops. Specifically, when the voltage drops below 4.75[8], the unit becomes unstable. Therefore, it is also necessary to take care about the quality and length of the power USB cable.

The second thing that is needed to get the unit working is to get an SD card. The SD card slot is in microSD format and is located at the bottom side of the module. The manufacturer specifies a minimum recommended size of SD card is 32 GB with UHS-1 speed class[14], which indicates a theoretical write speed of 10 MB/s. For this work, it was chosen a 64 GB SD card from SanDisk Ultra with UHS-1 speed class and A1 standard, which indicates that the card is designed for frequent storage of small blocks. Before inserting it into the unit, it is necessary to install the JetPack operating system from which it will boot.

2.1.2 JetPack instalation

There are two ways to install the JetPack operating system[14]. The first and easier way is to flash the JetPack image directly to the SD card. Download the completed JetPack image in .iso format from the NVIDIA website and then use the writing

software to flash the image to the SD card and create the bootloader partition. Then just plug the SD card into the NVIDIA Jetson unit and connect the power supply. The advantage of this way is simplicity, but the disadvantage is that you cannot customize the JetPack image. The image, which can be downloaded from the NVIDIA website, does not include all the modules that JetPack provides. If the user would like to use these modules, they must install JetPack using the second way. Fortunately, the image contained all the modules needed for this work, so the installation was done using the first way.

The second way to use a tool called NVIDIA SDK Manager. With this tool it is possible to customize JetPack modules. For example, some modules that will not be used can be removed to save disk space. The process of installing the JetPack into the Jetson unit is much more complicated than the first way. It is required to connect the Jetson unit to the computer via a USB cable and switch the unit to the Force Recovery Mod using a several pins on the board. With this NVIDIA SDK Manager tool, it is also possible to install the operating system program in the internal memory, not only on to the SD card. But for the units from the Nano series, installing the operating system on the SD card is the only one possible way to install JetPack, because they don't have their own non-volatile memory[8]. The SD card is the only memory the Jetson Nano units have.

At the time of writing this work, the latest version of JetPack is 5.1.1[10], but this version does not support the Jetson Nano, only the newer XAVIER and ORIN series units. The latest version that supports the Jetson Nano unit is version 4.6.3. The biggest difference between JetPack versions 4 and 5 is that version 4 uses Ubuntu 18, while version 5 uses Ubuntu 20.

2.1.3 Linux preparation

After successfully booting the unit and passing the startup wizard where the username, password, timezone etc are set, it needs to connect the unit to the internet and download the updates using the apt package tool. Internet connection can be done using an Ethernet cable or a WiFi USB dongle. In Ubuntu there are several default applications pre-installed and it is suitable to remove applications that will not be used to save disk space. Then it is the installation of programs according to personal preference. For example, I prefer GEdit as a GUI text editor and Nano as a text editor in the terminal. It is not comfortable to use the command line to navigate through the folders, so I prefer Midnight Commander, which makes it easier to navigate through the folders in the terminal. Midnight Commander is a fully text-based program, so it can also be used in remote SSH access. To monitor

the unit's resource usage and check the status of the hardware video accelerators, was installed the JTop program.

2.1.4 Remote access

There are several ways to control the NVIDA Jetson unit. The first and easiest way is to connect unit to monitor via HDMI/DisplayPort[14] and mouse and keyboard to USB port and control it directly as a regular desktop. JetPack contains the Ubuntu distribution with several desktop environments. As default graphic environment is set to Unity, but you can switch to more lightweight LXDE environment. Longer work is not comfortable, and there can also be situations when it is physically impossible to get to the unit and connect the monitor to it. For this reason, remote access is used, where one computer can control other computers. Nowadays it uses SSH protocol for remote control (Secure Shell)[24]. It is a secure communication channel that works under TCP/IP protocol. It allows to provide remote access to the terminal. The SSH protocol is implemented in the JetPack and is automatically run when the unit is powered on. If the controlling computer has a Linux system, using SSH is easy. Just type `$ ssh <username>@<IP address>` into the terminal and then the system will ask for the password.

It is also useful to be able to share files between the unit and the computer accessed remotely. SSH allows to transfer files, but it is not very handy. It is more convenient to use the FTP (File Transfer Protocol)[25] protocol. The protocol is very simple, reliable, and widely supported. The FTP client is not part of the JetPack system and has been installed. After successful installation it was necessary in the configuration file to enable remote writing, because by default the FTP client is only allowed to read only. For connecting to the unit from a Linux system, you can use the Midnight Commander terminal program. The program is very simple and completely sufficient. The program uses a two-window topology, where on one side of the window you can have folders displayed from your computer and on the other side you can have displayed folders from a remote drive. The FTP protocol has several disadvantages, namely that communication is not secure and sharing small files is not too fast.

There are also options where you can control the unit remotely, not only using the command line, but you can also share the whole desktop with windows. This is possible for example with the VNC (Virtual Network Computing) protocol, but the response time when using it was too long and the control was uncomfortable.

2.1.5 GStreamer setup

In the JetPack pack, GStreamer is included, but only the basic part of it, which cannot be used to create custom applications. When developing applications with GStreamer it is required to download the develop libraries. How to install them is described in the instructions on the NVIDIA document [15]. On the computer the situation is the same, you also need to install additional GStreamer develop libraries to be able to create your own applications.

As mentioned in the theoretical part, the functionality of GStreamer is based on the creation of a pipeline that can transfer large amounts of data very efficiently. This is very well suitable for multimedia content that requires a high data throughput. This work will be working only with video elements. The Jetson Nano unit includes three hardware encoders for h264, h265 and VP8 codecs [11]. Of course, it is possible to use a software encoder, but they would be too slow for real-time video applications. To work with hardware encoders, NVIDIA provides two libraries [15]. One is called OpenMAX and the second is called *Video4Linux2*. Both libraries contain all three codecs. The OpenMAX library is older, and last year NVIDIA announced that they stopped developing it, but it can still be used. On the other hand, the Video4Linux2 (v4l2) library is newer and still supported. It was decided to use only elements from the v4l2 library in this work. The names of encoder in the v4l2 library are called *nvv4l2h264enc* for the h264 codec, *nvv4l2h265enc* for the h265 codec and *nvvv4l2vp8enc* for the VP8 codec. Encoders can be configured using several parameters. Only a few of them will be listed here:

- Setting the bit rate mode (variable or constant output bit rate).
- Setting the average bit rate and maximum peak.
- Setting the interval by inserting I frames.
- Setting the Hardware Preset Level. The mode determines the accuracy of the motion vectors and the size of the macroblocks.
- Setting codec profile settings (Baseline, Main, High). Only works with h264 and h265 codecs.

The input to the encoders is only from NVMM memory and format of the uncompressed video input must be in I420 format only. To work with NVMM memory, it's possible to use the *nvvidconv* function included in JetPack.

RTP and UDP will be used for transmission over the IP network. To transmit video over RTP, the stream must be split into parts and added an RTP header. This is done by an element from the Gstrtppay library [16]. For video encoded with the h264 codec the element is called *rtph264pay*, for the h265 codec it is *rtph265pay* and for the VP8 codec it is *rtpvp8pay*. To let the receiver know how to manipulate with the received video, it is recommended to turn on the option of transmitting

the configuration block with the video. Specifically, these are the SPS (sequence parameter set) and PPS (picture parameter set) configuration blocks. To transmit the configuration block, it is necessary to enable the configuration output in the encoder. By default, it is disabled. Finally, just send the RTP packets using the UDP protocol. For this purpose, the *udpsink* element is used. Since the UDP protocol is used, you just need to specify the destination IP address and port. The port to be used is 5000, which is the standard port for RTP transfers.

The source of the video will be a live camera stream. In this work, the camera works with the USB UVC protocol [20]. To insert the camera video into the GStreamer pipeline, the *v4l2src* element is used [15]. Several parameters need to be set: Camera address, sync clock source, video format, video resolution, and frame rate. Parameters need to be set with regards to what a specific camera can deliver. For example, the camera used in this work for measurements can capture video at these resolutions and frame rates:

```
Index       : 0
Type        : Video Capture
Pixel Format: YUYV 4:2:2
  Size: Discrete 1920x1080
    Interval: Discrete 0.200s (5.000 fps)
    Interval: Discrete 0.333s (3.000 fps)
  Size: Discrete 1280x720
    Interval: Discrete 0.200s (5.000 fps)
  Size: Discrete 640x480
    Interval: Discrete 0.050s (20.000 fps)
  Size: Discrete 320x240
    Interval: Discrete 0.033s (30.000 fps)
Index       : 1
Type        : Video Capture
Pixel Format: 'MJPG' (compressed)
  Size: Discrete 1920x1080
    Interval: Discrete 0.033s (30.000 fps)
    Interval: Discrete 0.040s (25.000 fps)
    Interval: Discrete 0.050s (20.000 fps)
    Interval: Discrete 0.067s (15.000 fps)
    Interval: Discrete 0.100s (10.000 fps)
    Interval: Discrete 0.200s (5.000 fps)
    Interval: Discrete 1.000s (1.000 fps)
  Size: Discrete 1280x720
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 640x480
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 320x240
    Interval: Discrete 0.033s (30.000 fps)
```

This list was found using the following command: `v4l2-ctl -d /dev/video0 --list-formats-ext` As can be seen, the camera can transmit video in two formats. The first is the uncompressed YUYV4:2:2 format, with which high frame rates cannot be delivered. The second format is Motion-JPEG. It is a format with compression. The video quality will be lower, but the frame rate will be higher. To use Motion-JPEG format, it is necessary to decompress the video before any manipulation. In this work, the *nvv4l2decoder* from the v4l2 library[15], which is able to use a hardware accelerator, will be used.

On the receiving computer, the pipeline is like this. The input to the GStreamer pipeline is UDP packets, the *udpsrc* element is used for this [16]. The receiving UDP element needs to set the port and format in which to expect input data. The following function can sort RTP packets into the correct order and delete duplicate messages. From the RTP form, the video is back assembled into a continuous stream using elements from the Gstrtpdepay library (*rtph264depay, rtph265depay, rtpvp8depay*). For decoding it was used an element from the FFMPEG libav library (*avdec_h264, avdec_h265, avdec_vp8*). Before displaying on screen, it is useful to add a *videoscale* element to the pipeline that scales the video resolution. Now the video is decoded and ready to display on the screen using the *xvimagesink* element.

2.1.6 GstShark setup

GstShark is a tool for measuring and analyzing the internal behavior of GStreamer[18]. GstShark can tracers many parameters and events. To illustrate, here is a listing of some of the tracers:

- Buffer utilization monitoring,
- CPU load,
- Element delay tracking,
- Frame rate measures,
- Bit rate measures inside the pipeline,
- and many more

GstShark generates the measured values into a large file. It was required to create a script that can extract the measured values from the generated file and convert them into a format that MATLAB understands. The script was written in Bash language. The script creates classic .mat files and separates them by parameters. Then just open the files in MATLAB and plot the measured values. This can also identify possible sources of instability that may occur when running the GStreamer pipeline.

This tool is not a standard feature of GStreamer and must be installed separately. GstShark is an open-source tool and can be downloaded from GitHub[18]. GstShark

is installed using the make file. To enable GstShark, you need to switch GStreamer to debug mode by writing the parameter `GST_DEBUG="GST_TRACER:7"` before the *gst-launch-1.0* command. This parameter switches GStreamer into debug mode and starts sending pipeline information. GstShark captures this information and writes it to a file. To prevent the generated file not being too large, the user can select only the relevant parameters. It does this by adding another parameter. For example, to measure bit rate and frame rate, the new parameter would look like this: `GST_TRACERS="bitrate;framerate"`.

2.1.7 NTP setup

Time synchronization between the PC and the Jetson unit is very important, because it will make it possible to measure video latency. The synchronization topology will be as follows: the computer will serve as the reference time source and the Jetson unit will synchronize to it. The computer has a Linux distribution Ubuntu and it contains only NTP client but does not contain NTP server, so it needs to be installed. After installation, it is a good idea to set the internet addresses of the NTP servers, preferably the closest ones to you in the configuration file. To do this, you can use the ntppool.org cluster which unifies the NTP server's distributors and recommends the one with the best parameters for the user. In the case of the Czech Republic, just add the address *cz.pool.ntp.org* to the configuration file. The specific address for another country can be found on the web. For a more reliable connection, it is a good idea to add more addresses if some NTP servers are down. In most cases four addresses are enough. When the PC is without an internet connection, NTP will try to synchronize time over internet unsuccessfully repeatedly and will cause the PC to stop synchronizing Jetson unit. To prevent this, the address of the computer itself, i.e. localhost, is added to the configuration file, and the address is modified with the lowest priority *stratum 10* parameter. To activate the setting, the entire NTP process needs to be restarted with the command `$ sudo systemctl restart ntp .`

To set up the Jetson unit as a client, no additional software needs to be installed. In Ubuntu, the program *timedatectl* is used for NTP synchronization client. Again, it is necessary to set up a configuration file where you need to set the IP address of the computer. It is not strictly necessary, but it is a good idea to add parameters for synchronization intervals to the configuration file. The configuration file on the Jetson unit looks like this:

```
NTP=192.168.1.1
RootDistanceMaxSec=5
PollIntervalMinSec=32
PollIntervalMaxSec=2048
```

Again, you need to restart the whole process timedatectl and enable NTP synchronization with these command: `$ sudo systemctl restart systemd-timesyncd` `$ sudo timedatectl set-ntp true` .

When using NTP set up like this, it worked but was unstable. The time between the computer and the unit was only sometimes synchronized. Due to testing, it was necessary to restart the unit multiple times in one day and after restarting the unit, the NTP did not work reliably. After troubleshooting the problem, it was discovered that the NTP protocol has a mechanism that prevents frequent querying. To solve the problem, a new parameter *restrict 192.168.1.0 mask 255.255.255.0 nomodify notrap* was added to the configuration file on the computer that removes this restriction for devices connected to the local network. The complete configuration file on the computer looks like this:

```
server 0.cz.pool.ntp.org
server 1.cz.pool.ntp.org
server 2.cz.pool.ntp.org
server 3.cz.pool.ntp.org
server 127.127.1.0
fudge 127.127.1.0 stratum 10

restrict 192.168.1.0 mask 255.255.255.0 nomodify notrap
broadcast 192.168.1.255
```

2.2 U-BLOX setup

For this project will be used the development kit EVK-NINA-W101. Therefore, there was no need to create a custom PCB board. From the NINA module series, the W10 version was selected, because it has an open processor [21] and allows uploading of custom programs. To create a custom program, it was necessary to install the Espressif SDK version 4.4 [22]. The installation steps are clearly described on the Espressif website [23]. Creating a program for the NINA module is the same as creating a program for a regular ESP32. After installing the Espressif SDK is necessary to install a patch. From U-blox github download several .c files which overwrite the files in the Espressif SDK directory. The patch provides several modified features

for proper function of WiFi and memory. Before compiling the program it is necessary to set the configuration flag CONFIG_SPI_FLASH_USE_LEGACY_IMPL to set.

The NINA module connects to the computer using a converter UART <-> USB. The development board includes many peripherals, but you cannot run them all at the same time. For activating the periphery there are used several jumpers. In this work only WiFi and RMII bus will be used. To activate the RMII bus it is necessary to put the jumpers in the positions specified in the documentation [22]. Since the RMII uses most of the available pins, therefore it is not possible to use any other peripheral. Unfortunately, the RMII peripheral shares a pin with the BOOT pin, which is used when uploading a new program. It is always necessary to swap this jumper before uploading a new program. To upload a program, it must follow these steps: First the jumper on the BOOT pin must be swapped. Then you need to use the RESET and BOOT buttons in that sequence: Press RESET, then press BOOT, then release RESET, then release BOOT. The ESP32 processor gets into uploading mode. After uploading the program, just return the jumper and press the RESET button.

The program for this work is created by following several examples and snippets that can be found in the Espressif SDK documentation [23]. The FreeRTOS operating system is running in the program, which organizes the running of the program. The program works by first initializing both communication peripherals. Then, a set up a data tunnel in FreeRTOS to transport packets from the RMII and WiFi peripheral. The development kit is assembled with a KSZ8081 RMII transceiver chip. Unfortunately for this specific chip Espressif does not supply a ready-made configuration file, but fortunately it is very similar to another chip. The difference is only in the reference clock parameter. This difference was solved by modifying the value in the registry.

The program has been compiled and uploaded to the module. When the module starts, it first waits for active communication through the Ethernet port. Then, it starts a WiFi AP transmitter to which the computer can connect. The program works only on the second ISO/OSI layer. This means that the module cannot send packets to the network.

2.3 Software implementation

Running the GStreamer pipeline from the command line is only used for initial functionality testing. For more advanced operation and customization, it is necessary to create a program for the GStreamer pipeline. It is possible to use C/C++ or Python[16]. The Python was chosen for this work. To link Python and GStreamer,

it is necessary to download and install the library *python3-gst-1.0*. Several programs were created with functions that handled the operation of GStreamer pipelines and broadcast management. It created programs for both the Jetson unit and the computer.

In the Jetson unit, the program will be executed automatically at startup. This solution means that the codec cannot be changed automatically during operation but must always be changed manually. The GStreamer pipelines contains the elements described earlier. The programs are designed with multi-threaded support, with one thread used for the GStreamer pipeline and the second thread used for control. There is also a third thread that was created for measurement purposes. It services the latency measurement element. This thread will of course not be used in normal operation. Separate programs have been created for each of the three codecs.

The computer program is written to be compatible with the PyQt graphical framework. Individual programs were created for each codec separately. As with the Jetson unit, the program is written to allow multi-threading. One thread is used for the GStreamer pipeline, and the second thread is used for control. For measurement purposes, the third thread provides measurements of latency, frame rate, and packets received and lost. GStreamer Elements have been described earlier. To connect PyQt and GStreamer video output, the PyQt module *QMediaPlayer* is used.

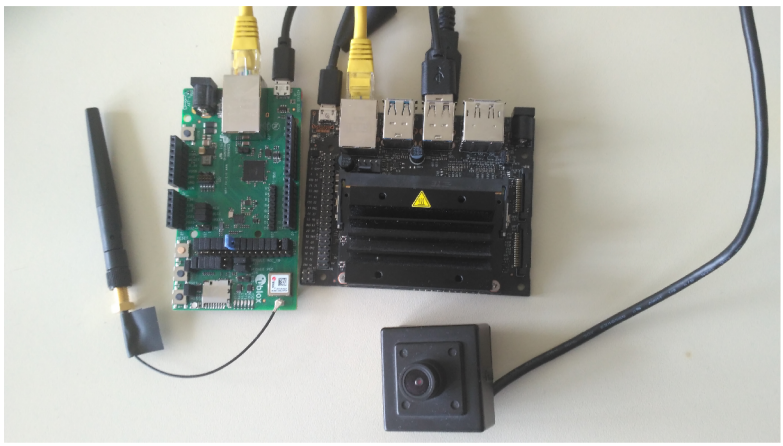


Fig. 2.1: Fully Complete Device

3 Measurement Results

For digital video transmission applications, it will be necessary to select a proper video codec and determine the required bit rate. As mentioned in the theoretical part, the NVIDIA Jetson Nano includes a video encoding hardware accelerator for three codecs: h264, h265 and VP8. In the case of real-time video transmission, the use of a hardware accelerator is crucial. Software encoding is too slow, so using a hardware accelerator will be necessary. This paragraph reports of four measurements that determine the best video codec and data rate for real-time video transmission.

3.1 File Encoding

As a first step, it was required to verify the characteristics of all codecs. To understand the basic characteristics of codecs, it will be done to encode the video into a file. During the encoding process, parameters such as CPU load, output bit rate characteristics, and the time it takes for the encode whole video file will be measured. The input video will be an uncompressed video downloaded from a free test video library *https://media.xiph.org/*. The reason why uncompressed video was chosen is so that it reflects the real characteristics of the codecs and does not have to decode the video before encoding. Details of the uncompressed video can be seen in the table 3.1. The individual frames are in YUV I420 format, which means that the video is only compressed by subsampling both chromatic components by the number 2. This means that in a 2x2 square there are four Y components, one U component and one V component.

Tab. 3.1: Uncompressed video File Parameters

File Parameters	
File name	Elephants dream
Video type	Uncompressed video
Format	.y4m (YUV I420)
Resolution	704x480
Frame Rate	24 FPS
Size	7.7 GB
Time	10 min 53 sec
Average bit rate	97000 kb/s

For the first introduction to the codecs, each codec was set to the default values according to the NVIDIA document[10]. As mentioned earlier, this work will use GStreamer elements from the *nvv4l2* library. The second *gst-omx* library will not

be used because it is no longer supported by NVIDIA and it is also recommended not to be used further[10]. The table 3.2 shows a list of the basic parameters that were set for the first test run. The codecs are set to use a variable bit rate, with a setpoint of 4000 kb/s and a maximum peak that will be 1.2 times the setpoint bit rate. Profiles for h264 and h265 were set to Baseline mode. The VP8 codec does not have any mod switching.

Tab. 3.2: Codec parameters

	h264 (nvv4l2h264enc)	h265 (nvv4l2h265enc)	VP8 (nvv4l2vp8enc)
Bit Rate Control Mode	Variable	Variable	Variable
Bit Rate	4 000 kb/s	4 000 kb/s	4 000 kb/s
Peak Bit Rate	1.2*Bit Rate	1.2*Bit Rate	1.2*Bit Rate
I Frame Interval	30	30	30
Profile	Baseline	Baseline	-

Pipeline in GStreamer was created like this: First, the *filesrc* element was used to read the video from the file. Because it is an uncompressed video, there is no need to use a decoder, just split the video file into frames using the *y4mdec* element. It is a useful practice to use a buffer *queue* to read from a file. Since elements from the *nvv4l2* library only support input from the hardware NVMM buffer, the video stream needs to be switched from the CPU buffer to the NVMM buffer by using the *nvvidconv* element. Stream is now ready to be encoded with one of the encoding elements (*nvv4l2h264enc, nvv4l2h265enc, nvv4l2vp8enc*). For the h264 and h265 codecs, they need to be split into individual blocks using the *h264parse* and *h265parse* elements before saving. Finally, just pack the compressed video into a file. For h264 and h265 codecs, the QuickTime format will be used, and the file will have a .mp4 extension. For the VP8 codec, the Matroska format will be used and the file will have the .mkv extension. Below are the pipelines for all three codecs:

```
$gst-launch-1.0 filesrc location=elephants_dream.y4m ! y4mdec !\
queue ! nvvidconv ! 'video/x-raw(memory:NVMM),format=(string)I420' !\
nvv4l2h264enc ! h264parse ! qtmux ! filesink location=output.mp4

$gst-launch-1.0 filesrc location=elephants_dream.y4m ! y4mdec !\
queue ! nvvidconv ! 'video/x-raw(memory:NVMM),format=(string)I420' !\
nvv4l2h265enc ! h265parse ! qtmux ! filesink location=output.mp4

$gst-launch-1.0 filesrc location=elephants_dream.y4m ! y4mdec !\
queue ! nvvidconv ! 'video/x-raw(memory:NVMM),format=(string)I420' !\
nvv4l2vp8enc ! matroskamux name=mux ! filesink location=output.mkv
```

The GstShark tool was used for the measurements. As can be seen from the table 3.3, the overall processed times are very much the same. This means that when using

a hardware accelerator, the encoding time is equal for all codecs. CPU load was measured to see what effect the CPU has on video encoding, even though the main encoding is done by the hardware accelerator. Since the Jetson unit is switched to Performance Mode, it has all four cores activated. The measured results from all four cores are averaged into one average value, which is written in the table 3.3. However, even though the encoding is done by a hardware accelerator, the CPU is also heavily loaded. As expected, h265 is the most complex to encode. From the figure 3.1, we can see that the h264 and h265 codecs made a bit rate waveform that are very close to the set threshold, but the VP8 codec bit rate is more floating. This can also be seen in the file size, where the VP8 file is smaller than the others. This means that VP8 has worse bit rate control than the h264 and h265 codecs.

Tab. 3.3: Result of file encoding

	h264	h265	VP8
Processing time	73 s	71 s	68 s
Average CPU load	52,97 %	75,43 %	67,15 %
Final file size	305,3 MB	305,4 MB	237,6 MB

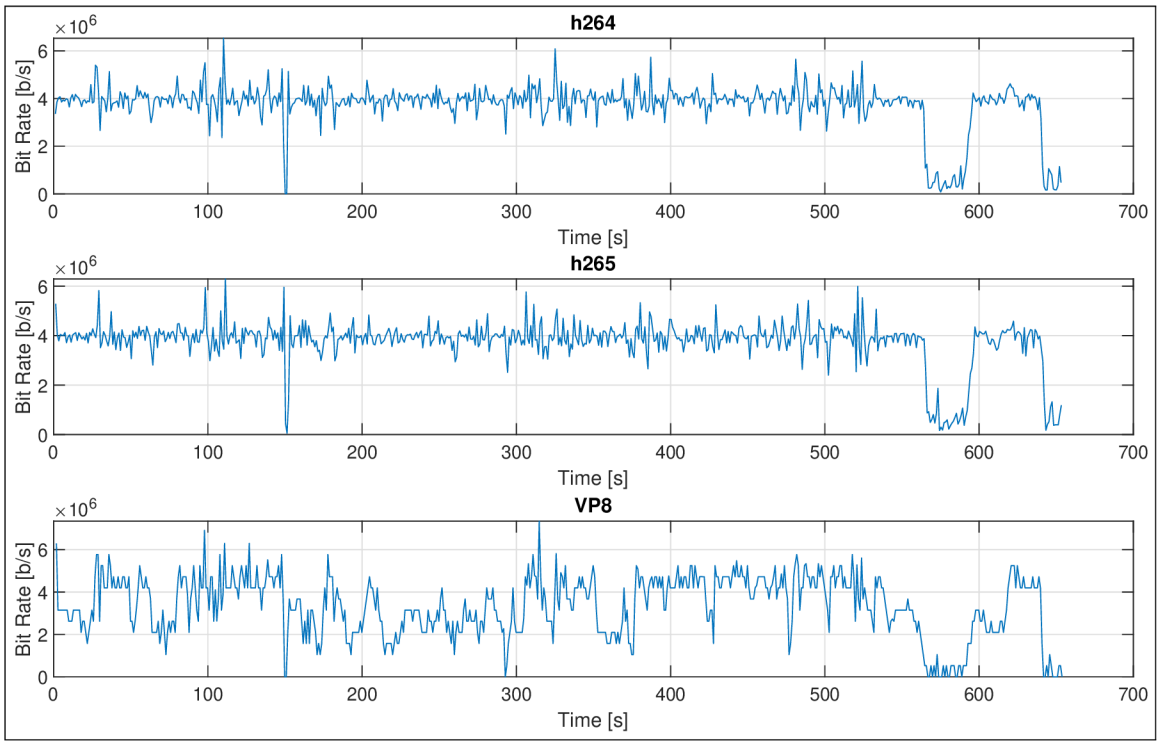


Fig. 3.1: Bit rate of the final file

3.2 Subjective test

The change of bit rate has a direct effect on the change of video quality. For this work, it was decided to make a subjective test to find an optimal bit rate that is still acceptable to humans. Since this work focuses on real-time video transmission, it is therefore necessary to select bit rates as small as possible.

First, for the subjective test, it was necessary to determine the test range of bit rates. The minimum value was determined by increasing from the minimum bit rate until the video was acceptable. For the VP8 codec, the lowest bit rate is 500 kb/s, so the last value is this one. For codecs h264 and h265 it was possible to create video with a lower bit rate, but the quality of the video was so bad that it was decided that the lowest bit rate for codecs h264 and h265 is 300 kb/s. A total of 17 values were selected, with a care to ensure that respondents did not have too many samples. The list of test samples can be seen in the table 3.4.

Tab. 3.4: List of test Bit Rate

Bit Rate [kb/s]				
h264		**h265**		**VP8**
2000		2000		2000
1000		1000		1000
800		800		800
600		600		600
400		400		500
300		300		

(a) Bit Rate = 200 kb/s (b) Bit Rate = 2000 kb/s

Fig. 3.2: Example of sample compression test

The test video sequence was recorded on the camera, specifically the car ride. This view was chosen because the final application of this device will be a video broadcast of a car ride. This view is specific because the scene contains the sky and the road, which change very few times while driving. The surrounding landscape and oncoming cars change the most from the scene. In the figure 3.2 you can see an example of a video compression result with a very low bit rate, such quality was considered unsatisfactory and did not make it into the test samples.

The graph 3.3 shows the waveforms of the real output bit rate with the center bit rate set to 600 kb/s. The encoders were switched to constant bit rate mode. As can be seen from the graph 3.3, the h264 codec has the best constant bit rate waveform and the VP8 codec has the worst waveform.

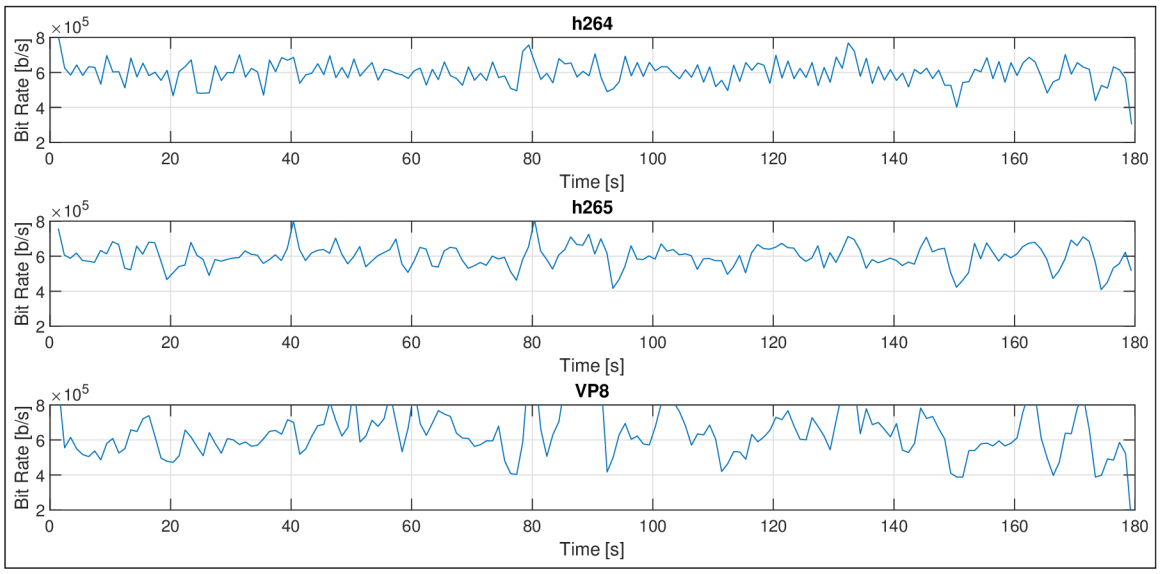


Fig. 3.3: Bit rate of the 600 kb/s sample

The graph 3.4 shows bit rate 300 kb/s is also very interesting. The h265 codec has the best waveform to handle this very low bit rate. The waveform in this case is better with the h265 codec than with h264 as was the case with the previous waveform with a center bit rate at 600 kb/s. The waveform of the h264 codec very often spikes up to 1.5 times the set bit rate. This is another reason why bit rate 300 kb/s was set as the lowest possible in test range. Again, the VP8 codec was the worst waveform, unable to encode video at the 300 kb/s bit rate. It can be seen from the waveform that the lowest bit rate for the VP8 codec is 500 kb/s and therefore this is the lowest value in the test range for the subjective test.

As mentioned earlier, for the subjective test 17 bit rate values were selected and test video samples of duration 26 s with 720p resolution and 30 FPS were created. A sample of the test video is shown in the figure 3.2. A text about the codec

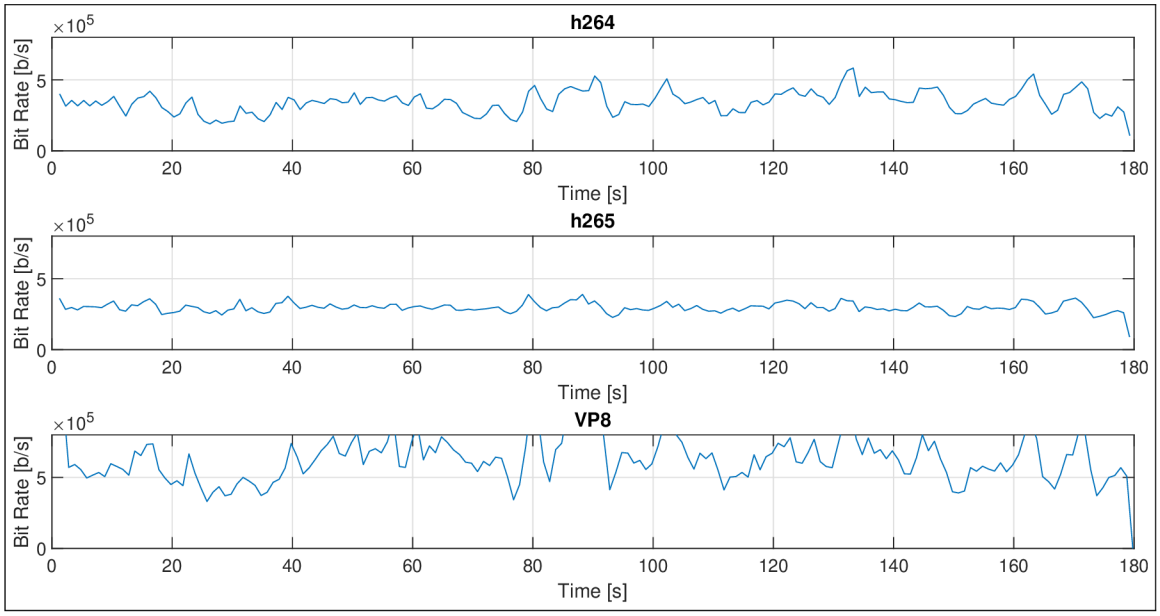


Fig. 3.4: Bit rate of the 300 kb/s sample

used and the bit rate set was inserted in the video for better orientation in the samples. The questionnaire was completed by a total of 10 respondents who rated the scores from 1 (best) to 5 (worst) and could use a floating number. The result of the questionnaire can be seen in a graph 3.5 where the MOS (Mean Opinion Score) is plotted, accompanied by a confidence interval. The MOS can be calculated as follows[4]:

$$MOS = \frac{1}{N}\sum_{i=1}^{N} u_i \tag{3.1}$$

Here, N is number of respondents and u is respondent's result. Confidence interval is calculated according to the ITU-R BT.500 standard. The standard ITU-R BT.500 defines the formula for calculating the confidence interval as follows:

$$\delta = 1.96\frac{S}{\sqrt{N}} \tag{3.2}$$

Here, N is number of respondents and S is standard deviation. The standard deviation can be calculated as follows:

$$S = \sqrt{\sum_{i=1}^{N} \frac{(MOS - u_i)}{N-1}} \tag{3.3}$$

As can be seen from the graph, respondents were less consistent in their ratings for the h264 and h265 codecs, and therefore the confidence interval is quite wide. In contrast, for the codec VP8, respondents very often agreed on their results. This shows that the output quality of h264 and h265 codecs is very subjective.

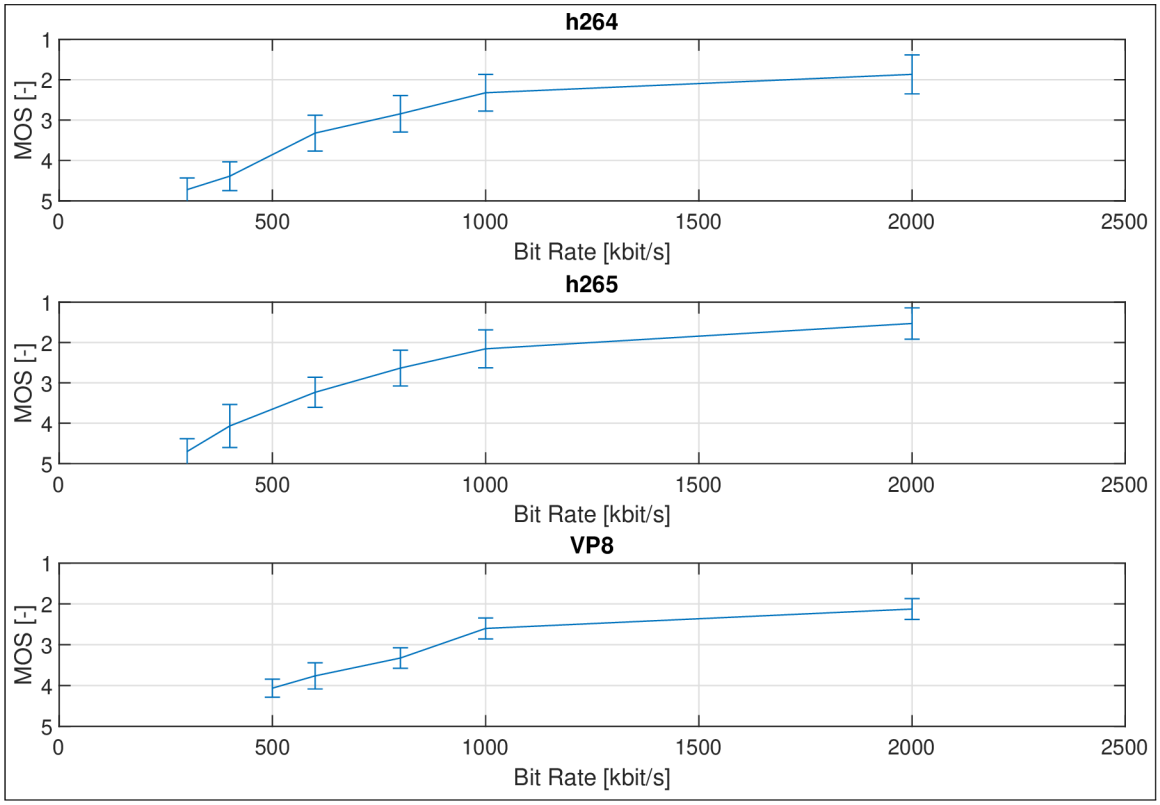


Fig. 3.5: Result of the subjective assessment

Putting all three results into one graph 3.6 shows what was written in the theoretical part. The best codec is h265, which according to a subjective test achieves better video quality at the same bit rate. In the middle place is the h264 codec and the worst codec is VP8. Another thing that can be seen from the graph is that the lower the bit rate used, the smaller the MOS differences become. The result of the subjective test is that the minimum bit rate that is still acceptable for the viewer is around 1000 kb/s.

3.3 Real-time video transmission measurement

The main goal of this work is real-time video transmission, which means that the most important parameter is low latency and stability at the expense of video quality. Real-time video will be transmitted using the RTP protocol and individual packets will be transmitted over the TCP/IP network using the UDP protocol. The disadvantage of using UDP is that the transmitter has no information if the packet arrived well, but the transmission is faster. Video transmission will be tested using three different codecs and for simplicity of measurement, the bit rate will be set to

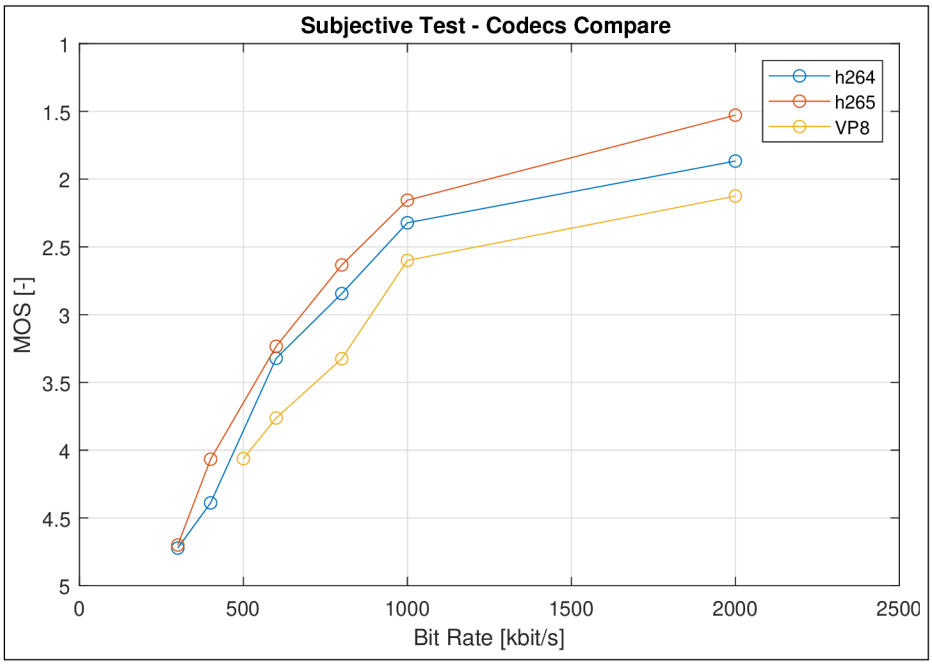


Fig. 3.6: Codecs comparison of average ratings score

1000 kb/s for most situations. The point of this measurement should be to determine which codec is best for real-time video transmission in outdoor environments.

3.3.1 Methodology of measurement

To measure real-time transmission, it was necessary to define methods to measure latency, frame rate, packets received and lost, and bit rate.

Latency

Latency means the amount of time it takes for a frame to go from being processed in the unit to being displayed on the monitor. It may seem like a very simple task, but the implementation is very complex. Packets are sent using the UDP protocol, which has no delivery control and individual packets do not carry information about the time of sending, so cannot be used to measure latency. Another way is to send additional packets containing information about the timestamps when each frame was sent. This method is good, but it increases the data rate and is difficult to implement, because individual frames must be tracked with a timestamp. For the latency measurement, an effort was made to keep the method as simple as possible. The selected method works by inserting the text with the current time into the frame before compression and then also insert the current time after decompression at the receiver. By inserting time directly into each frame, the problem of matching

timestamps to frames is solved. You can see the final implementation in the sample frame 3.7. At the top is the encoding time and at the bottom is the decoding time. For proper operation, it is required to successfully synchronize the time between the unit and the computer using the NTP protocol.

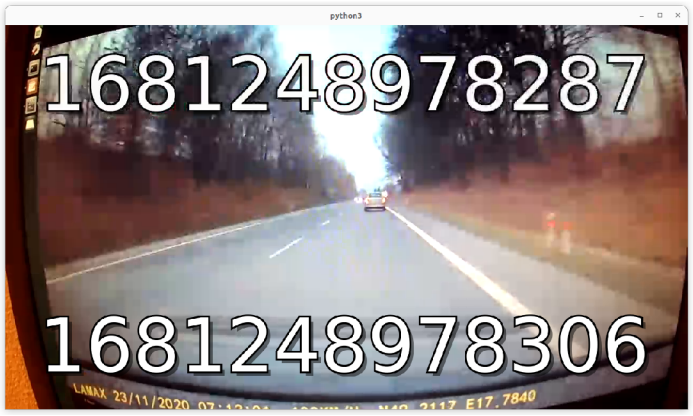


Fig. 3.7: implementation of the latency measurement method

In GStreamer element library, there is no element that can insert the current time into the video, so it was necessary to create one. As mentioned earlier, the GStreamer implementation is written in python. In the Jetson implementation, the frame is always extracted from NVMM memory (which is the memory used by CUDA kernels) into regular RAM before being inserted into the encoder. Then it is very easy to insert the current time into the frame using the available python libraries. For simplicity, the time is in millisecond format. Then the frame is returned to NVMM memory and inserted into the encoder. The situation is similar on a computer side, but the frames don't have to be moved between memories.

In the process of using it, the disadvantage was found that the time value could not be automated read and it was necessary to manually read the times and subtract them

Packet measurement

The goal of this measurement is to get the number of packets that were delivered correctly and on time and the number of packets that were lost, arrived late or arrived multiple times. For this measurement it uses the identification marks in the RTP header and the PTS (presentation timestamp) block where the frame rate is included. Thanks to the identification marks in the RTP header, it is very easy to see how many packets have been lost. Because if a packet arrives and it does not have a mark one higher than the previous one, it means that the packets have been

lost between them. In addition, the marker can be used to determine the correct ordering of packets if they arrive out of order. Thanks to PTS, it is possible to calculate how often each frame should be received.

Frame Rate

For the frame rate measurement, the GstFPSDisplaySink element was used, which is located in the GStreamer library. The element measures the current and average frame rate. The element has two data outputs. The first one is writing the frame rates directly to the video as a text, and the second one is callback function from which the frame rates can be read. In this project the callback function is used, and the measured values are stored in a csv file.

Bit Rate

The bit rate was measured using WireShark. WireShark is a very advanced analysis tool that can be used to analyze all network communications. In the case of this project, only UDP packets were analyzed. Afterwards, the data was saved in a csv file.

3.3.2 Measurement of internal parameters in the unit and in the computer

It is useful to measure the internal states of the Jetson unit and the computer during video processing and transmission. With this measurement, it is possible to know how much performance the video transfer process takes and how much performance is available for other applications. CPU load, elements latency and bit rate between elements were measurements on the Jetson unit. Measurements were made on all three codecs with different output bit rate settings. Latency and bit rate were measurements on the computer.

To measure the CPU load in the Jetson unit, the GstShark measurement tool was used. The value of the average CPU load was based on a measure for 60 s period. Measured for all three codecs and with the bit rate set to 800 kb/s, 1000 kb/s and 2000 kb/s. The video was in 720p resolution, and the frame rate was 30 FPS. As can be seen from the table 3.5, all values are very similar, and it can be determined that the CPU load was the same for all situations. This is due to the fact that a hardware accelerator is used.

To measure the bit rate, the GstShark measurement tool was used. The average bit rate at the element outputs, which are the same for all codecs, is listed in the table 3.6. The length of the measurement period was 60 s and the video was in 720p

Tab. 3.5: Average CPU load in the Jetson unit during real-time transmission

		h264	h265	VP8
800 kb/s		32.65 %	31.46 %	33.18 %
1000 kb/s		33.32 %	31.21 %	32.03 %
2000 kb/s		31.82 %	31.82 %	32.83 %

resolution with 30 FPS. The camera output is compressed with the jpeg codec, so it is necessary to decode the video before any manipulation. To the encoder inputs uncompressed video in I420 format at a bit rate of 330.5 Mbps.

Tab. 3.6: Average bit rate in the Jetson unit real-time transmission

Camera output	Camera output decoder	Measure Latency Element	Encoder input
26.1 Mb/s	330,5 Mb/s	330,5 Mb/s	330,5 Mb/s

The table 3.7 shows the average values of the output bit rate of all encoders. The length of the measurement period was 60 s and the video was in 720p resolution with 30 FPS. Codecs are set to a constant bit rate mode. The graph 3.8 compares the output bit rate of all three codecs for a set bit rate of 1000 kb/s. As can be seen, the h265 codec has the smoothest and most stable waveform. A very interesting waveform has the VP8 codec, which has a large spike after about 8 seconds. The reason for this phenomenon is unknown.

Tab. 3.7: Average bit rate on the encoder output during real-time transmission

	h264	h265	VP8
800 kb/s	792.860 kb/s	791.690 kb/s	793.400 kb/s
1000 kb/s	996.900 kb/s	987.790 kb/s	992.560 kb/s
2000 kb/s	2001.300 kb/s	1997.300 kb/s	1994.719 kb/s

To measure latency, the GstShark measurement tool was used. The elements, which are the same for all codecs, are listed in the table 3.8. The latency element was measured to have a latency of 7 ms in Jetson and 3 ms in the computer. Decoding the video on the computer side takes the same time for all codecs, 5 ms. Thanks to the measured latency inside the unit and the computer, the real latency of the wireless link can be calculated.

The table 3.9 lists the latency of all codecs for 800 kb/s, 1000 kb/s and 2000 kb/s bit rate and the latency for elements that pack the video stream into RTP and UDP packets. As can be seen from the table, the larger the bit rate value is set, the higher

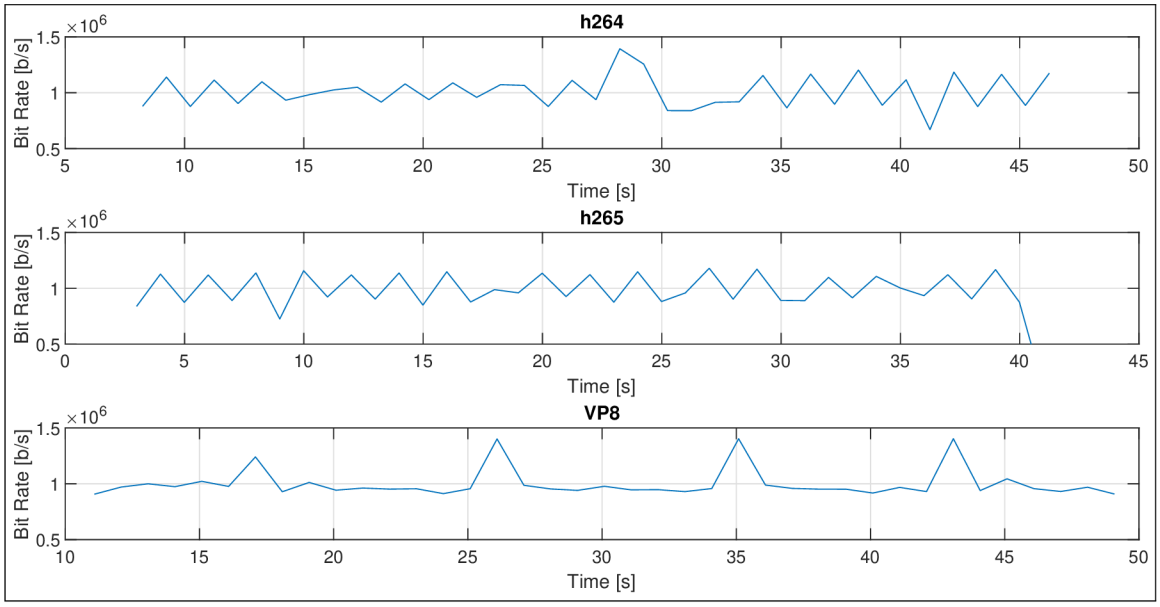


Fig. 3.8: Encoder output bit rate during real-time transmission

Tab. 3.8: Average latency during real-time transmission

Latency in the Jetson unit		
Camera capture	Camera decode	Latency Element
0.073 ms	2.856 ms	7.513 ms

Latency in the computer		
Processing UDP and RTP	Video decode	Latency Element
0.267 ms	5.020 ms	3.061 ms

the overall latency. The h264 codec had the lowest latency and the VP8 codec had the highest.

3.3.3 Outdoor measurement when both stations are stationary

The most important chapter of the measurement is the effect of distance on the parameters of the video. After consulting with the supervisor, it was decided that the WiFi adapter built into the laptop would be used as a receiver. Because using an external antenna would only increase the distance, but the parameter waveform would remain unchanged. On the U-Blox NINA WiFi transmitter a 3 dBi omni-directional antenna was installed and the transmitter was set to maximum output power. For protection reasons, the Jetson unit and the WiFi transmitter were enclosed in a plastic box 3.9. The measurements were done outdoors, and the receiver

Tab. 3.9: Encoder latency during real-time transmission

h264		
	Encoder	UDP and RTP Process
800 kb/s	5.201 ms	0.176 ms
1000 kb/s	5.209 ms	0.184 ms
2000 kb/s	5.934 ms	0.338 ms
h265		
	Encoder	UDP and RTP Process
800 kb/s	5.939 ms	0.210 ms
1000 kb/s	5.923 ms	0.244 ms
2000 kb/s	5.934 ms	0.338 ms
VP8		
	Encoder	UDP and RTP Process
800 kb/s	8.763 ms	0.124 ms
1000 kb/s	8.777 ms	0.148 ms
2000 kb/s	8.852 ms	0.180 ms

always had a direct line of sight to the transmitter. The test video was live output from a video camera with 720p resolution and 30 FPS.

Measurements when both stations were stationary were done by placing the

Fig. 3.9: Device enclosed in a plastic box

Jetson unit in a visible location and the user with the computer moved in steps of 10 meters. At each step, measurements were taken for 60 s. During this time, the average frame rate and latency were measured. The packet count is the sum of packets received and lost during this time.

The first graph 3.10 is for the h264 codec. The graphs show very clearly the characteristic of digital video where small errors can be recovered by the receiver using error correction codes, but when the number of errors exceeds a specific limit, the video will stop. In the case of h264, this limit distance occurred at 170 m. Until this distance, the average frame rate was kept at around 28 FPS. The average latency started at 38 ms and also increased with increasing distance up to 65 ms. According to the subjective ratings of users, the h264 codec was good. Synchronisation at startup was fast and there were relatively few error artefacts or other image problems in the resulting video. Most often the video stopped, but after a while it continued again.

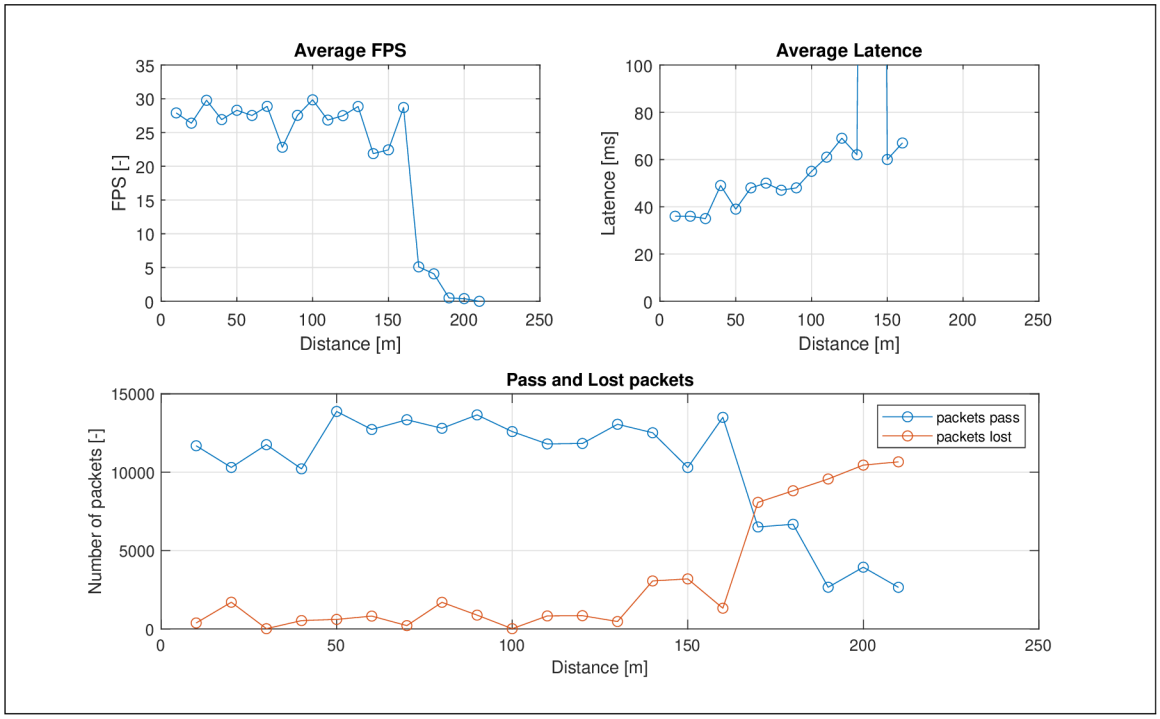


Fig. 3.10: Outdoor measurement result with static stations for h264 codec

The second graph 3.11 is for the h265 codec. The difference from the h264 codec is that the h265 codec has a more stable frame rate and the average rate was close to the maximum frame rate of 30 FPS. Latency started at 50 ms up to 90 ms. Latency is about 20 ms higher than h264. The range where video could be played was 210 m and this is 50 m more than the h264 codec. This shows that the h265 codec has much

more robust methods for repairing errors caused by transmission. Very interesting were the h265 codec errors, which were represented by green horizontal bars or the whole green screen. The frequency of freeze and the number of artifacts was the same as for the h264 codec. Synchronization was fast when the stream started. The transmission made a good subjective impression on user.

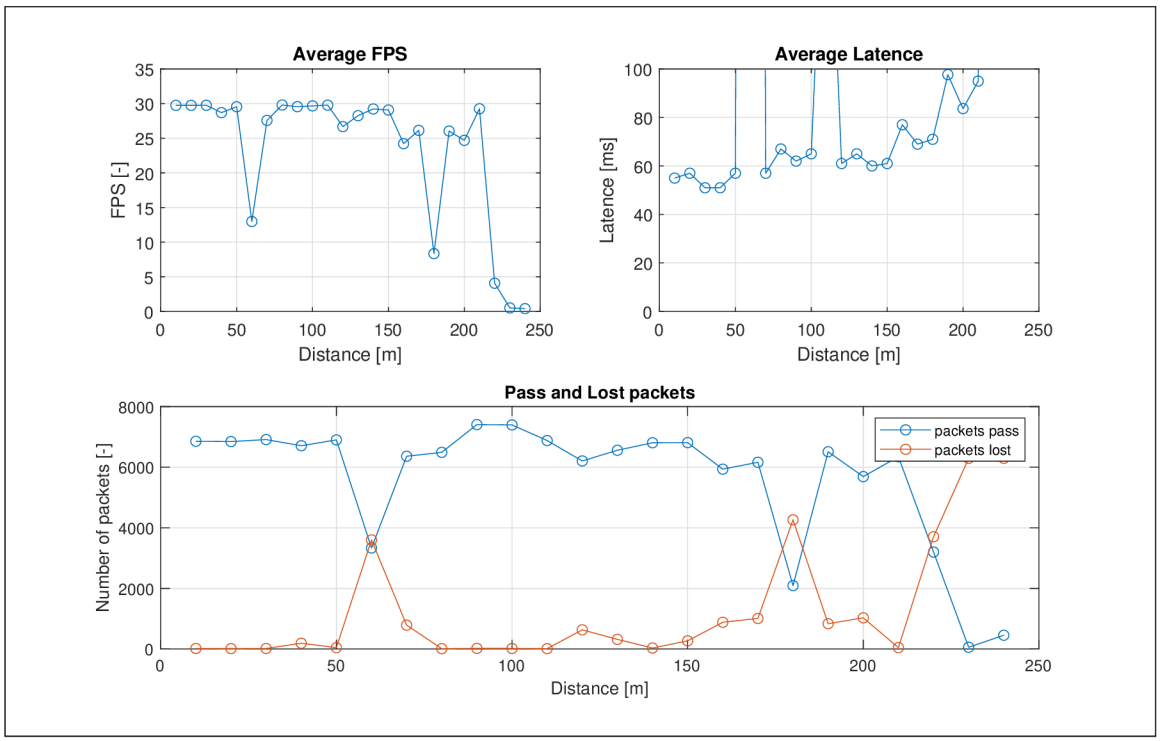


Fig. 3.11: Outdoor measurement result with static stations for h265 codec

The third graph 3.12 is for the VP8 codec. The average frame rate was around 25 FPS. During transmission, there were many freeze that affected the measurements. The maximum distance at which the video transmission could be played was 160 m. This is the same distance as for the h264 codec. From the latency plot it looks like an ideal transmission, where latency does not increase with distance. Which of course is not possible and therefore the latency of VP8 will be ignored. I assign the reason for the error to high frequency of streaming failure. Synchronization took longer when starting the stream, compared to other codecs. The most common artifacts were multiple small white squares spread randomly across the frame. The second most common defect was that the loss of color and then the video was gray. The transmission did not make a good impression on the user.

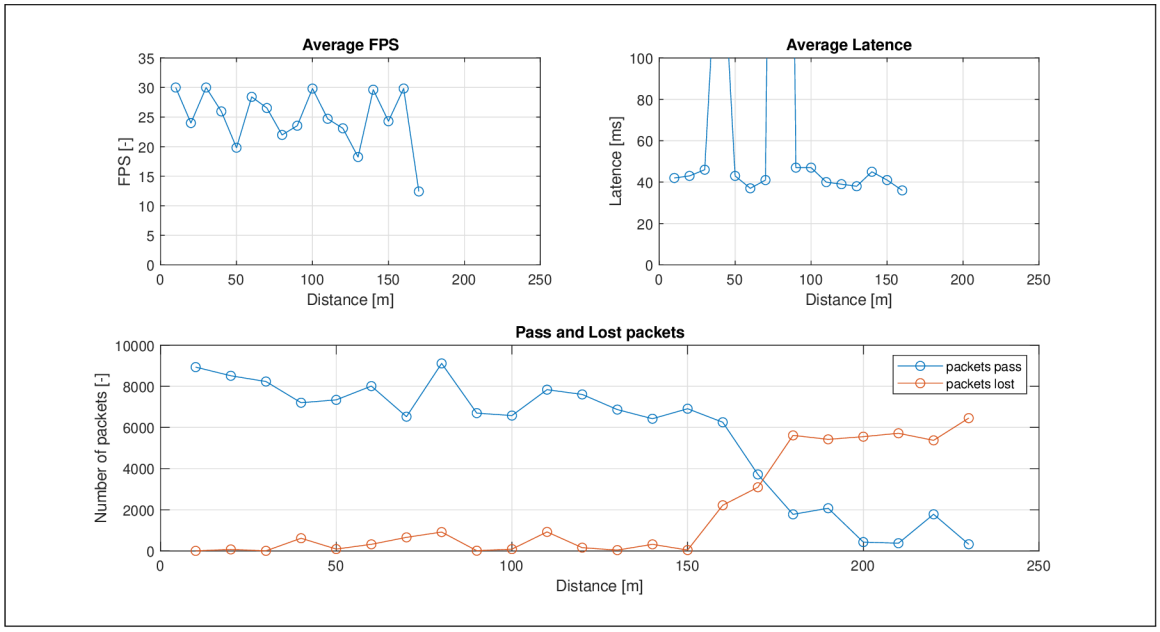


Fig. 3.12: Outdoor measurement result with static stations for VP8 codec

3.3.4 Outdoor measurement when one station is in movement

To verify the functionality of the device, it was done a test where the transmitting unit was moving, and the receiver was stationary. The set-up was exactly the same as the last measurement. The test used an area of 80 x 80 m. Unfortunately, it was a small space, and therefore all three codecs had no problem with the range. The test of each codec took about 4 minutes. Unfortunately, the track line for the moving unit could not be made to always be the same, so the track line for the moving unit changed for each measurement. Unfortunately, the most common reason for the freeze was covering the antenna with body parts. In the graphs below, there is always a selection of a part of the measurements.

The h264 codec had the same characteristics as the last measurement. The initial synchronization was fast and when the quality of signal degraded, the video stopped. When the quality of signal improved the video resumed automatically. The most common artifacts were represented by blurred squares. On the graph 3.13 you can see the bit rate waveform on the receiving computer and the frame rate. There are visible dropouts in the frame rate.

The h265 codec also had the same characteristics as in the previous test. The initial synchronization was fast and when the quality of signal degraded, the video stopped. When the quality of signal improved the video resumed automatically. The most common artifacts are green stripes and blurred squares. The graph 3.14 shows a section of a very bad signal. But it showed that the h265 codec could handle

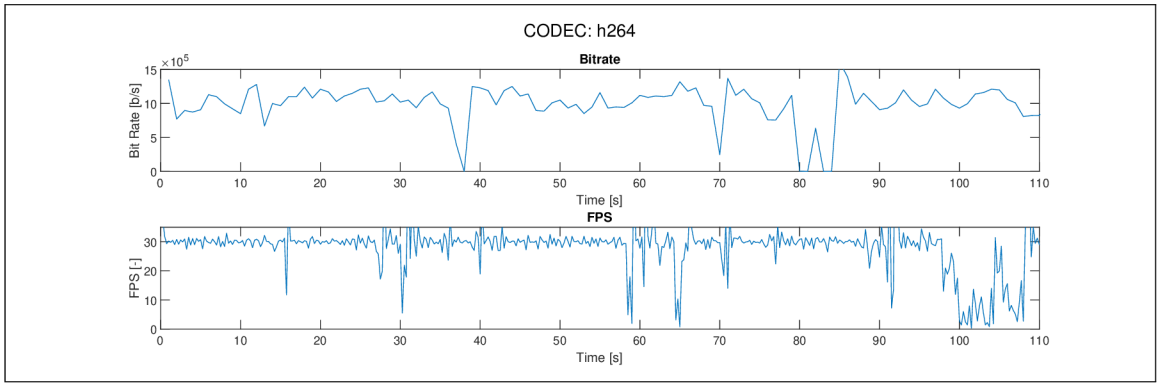


Fig. 3.13: Outdoor measurement result with one move station for h264 codec

the bad signal and would reconstruct the videos even if it contained errors. When testing real-time video, users agreed that transmission using the h265 codec was the best. The video did have defects, but they didn't disturb as much as other codecs.

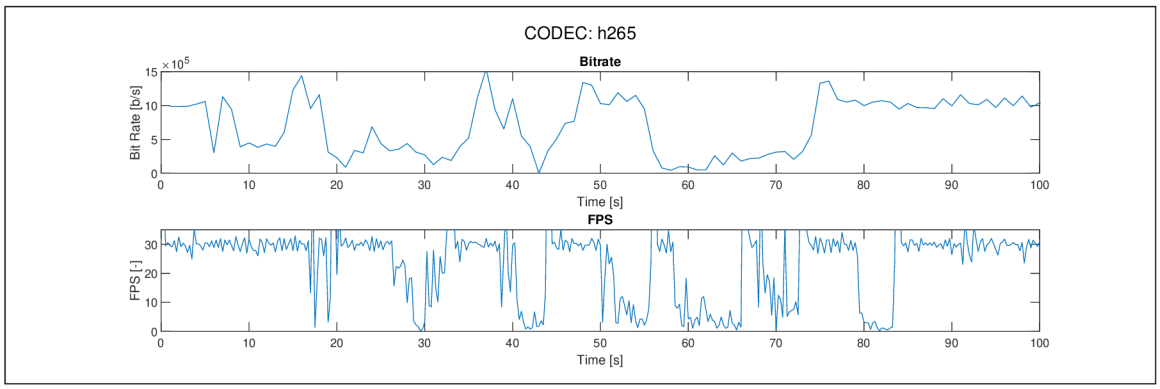


Fig. 3.14: Outdoor measurement result with one move station for h265 codec

The VP8 codec also had the same characteristics as in the previous test. The initial synchronization took longer to synchronize with the input video stream and the video contained many artifacts. A very interesting phenomenon can be seen in the graph 3.15, where it looks like a perfect video transmission even though the quality of signal was not perfect. This is because where the h264 and h265 codecs would stop the video due to missing data, the VP8 codec continued, but the video was all grey. From the user's subjective assessment of view, the transmission was annoying. It can be concluded that the VP8 codec is not suitable for real-time applications.

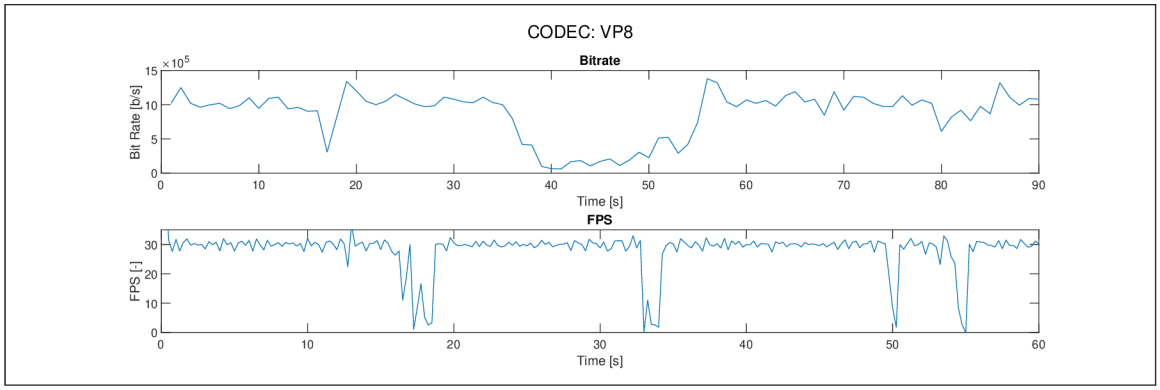
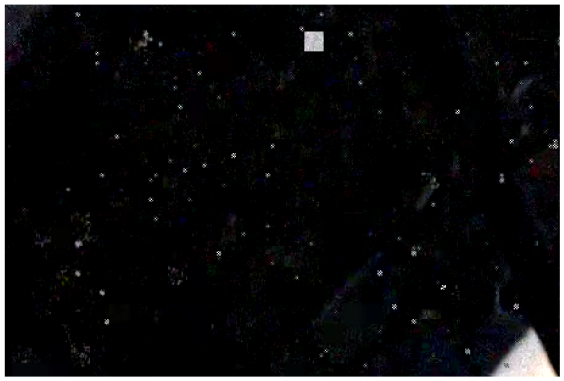


Fig. 3.15: Outdoor measurement result with one move station for vp8 codec

3.3.5 Artefacts in video transmission

(a) Blurred squares in codec h264

(b) Green stripes in h265



(c) Small white squares in VP8

(d) Grey screen when frame I is lost. It happens with all codecs

Fig. 3.16: Examples of Artifacts in video transmission

Conclusion

The aim of this thesis was to design a device that will be able to process video from a camera and wirelessly transmit it to a computer where it can be displayed on a monitor. The device is powered by the NVIDIA Jetson Nano and video is streamed wirelessly through a WiFi interface. In theoretical part are described the problems of video transmission and describe all the most important parts of the compression process. It also explained the differences between h264, h265, and VP8 codecs. The next chapter described the protocols that work over IP network and are used for the real-time transmission of multimedia content. The last chapter of the theoretical part focuses on the analysis of individual parts of the resulting device. As the transmitting unit, where the video is processed, the NVIDIA Jetson Nano computer is used. Features and how to make the unit start-up operation and then how to use it describe in detail. A computer is used as a receiver, where the video is decoded and displayed on a monitor. It was decided to not use an external antenna but to use the antennas integrated into the netbook for measurements. This reduces the range, but the transmitting characteristics remain the same. The GStreamer framework was used to work with the video in the Jetson unit and on the computer. The control of GStreamer and the operation of the whole device was written in Python. The theory around Gstreamer was described and the final pipeline for real-time video streaming was explained. For video encoding, hardware accelerators were used. U-Blox NINA W10 radio module was used for wireless transmitting, which consists of an ESP32 processor and radio apparatus. It describes how to start the module, how to write and upload the program and how the software work. The complete device was successfully assembled and made operational. The software libraries for the Jetson unit and the computer have been written. The library for the computer is modified to be compatible with the PyQt graphical framework. With this device, several measurements were made to compare the performance of the codecs and their settings.

In the measurement part, the codecs were tested with several measurements. The measured parameters are the CPU load of the Jetson unit, the total latency from the camera to the monitor, the bit rate waveform, and the frame rate waveform. When measured in real conditions, stability was measured at distances between stations. A subjective test was done to determine the limit of acceptability of the video quality. A total of 17 test video sequences were created for the experiment, from a bit rate of 300 kb/s to 2000 kb/s. A total of 10 respondents completed the questionnaire. The subjective test proved that the limit of acceptability for 720p video is 1000 kb/s. Another result from the subjective test determined that the h265 codec had the best video quality and the VP8 codec had the worst video quality. This result

agrees with the theoretical predictions. For initial measurement of the performance of all three codecs, they were tested for video encoding to file. From the results, it was discovered that the difference between codecs when using hardware accelerator takes compression time and CPU load similarly. For the h264 and h265 codecs the waveform was nicely stable, but for the VP8 codec the waveform was unstable. For real-time transmission, it is preferable when the bit rate is stable. The next measurement of the codec parameters was to measure the internal behavior of the unit and the computer when the transmission was active. It can be seen from the bit rate waveform that the h264 and h265 codecs have a relatively stable waveform, but the VP8 codec had a waveform with periodic spikes. The inter latency measurements showed the latency of each element in the GStreamer pipeline. The h264 encoder has a latency of 5.4 ms, the h265 has 6.2 ms, and the VP8 has 8.8 ms. When the bit rate increased, the latency also increased. Video decoding on the computer takes 5.2 ms for all codecs.

For measurements in real conditions, total latency, average frame rate, bit rate, and number of received and lost packets. The measurements were performed in two different situations. The first situation was when the transmitter and receiver were not moving, and the second situation was when the transmitter was moving. Thus, when measuring the first situation, the phenomenon of digital videos was evident. When small errors can be repaired by the error correction code, but when the number of errors exceeds a limit, the video stops. For the h264 and VP8 codecs, this limit occurred at 170 m. For the h265 codec, this limit occurred at 210 m. When changing the encoder parameters (bit rate, I-frame frequency), the maximum distance did not change. For latency measurements, the h264 codec had a 20 ms lower average latency than the h265 codec.

In the second situation, when the transmitting unit was in motion, it was measured in an area of 80x80 m. In this area, all codecs had no problem with the range. The VP8 codec took the longest time to initially synchronize the video stream and the video contained many error artifacts. Compared to the other codecs, the VP8 codec did not stop when packets were lost but continued even when the screen was garbled. The h264 codec was fast during the initial synchronization. The video stream was freezing more often than showing error artifacts. The frequency of freezing was relatively frequent. The h265 codec also had fast initial synchronization, and the video was much more stable than the h264 codec. Even with a poor signal, the h265 was able to reconstruct and display the video. The most frequent artifacts in the h265 codec were green horizontal lines. Despite sometimes freezing and showing artifacts, the video stream was the best to watch.

Measurements determined that the best codec to use for real-time streaming is h265. The h265 codec was able to transfer the video at the longest distance. From

the measurements when the transmitting unit was in motion the h265 codec was the best. Compared to the h264 codec, it has a 20 ms higher average latency. Thanks to the use of a hardware accelerator, there was no problem with performance and the encoding time is 6.2 ms.

Bibliography

[1] BEACH, A.; OWEN, A. *Video compression handbook. Second edition.* Berkeley: Peachpit Press, 2019. ISBN 0-13-486621-5

[2] ČÍKA, P. *Multimediální služby.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací 2012. ISBN 978-80-214-4443-0

[3] WIGGINS, P. *Creating interlaced video from progressive footage in Final Cut Pro X.* [online]. Copyright © 2015 FCP.co [cit. 1. 10. 2022]. Dostupné z URL: <`https://fcp.co/final-cut-pro/tutorials/`>

[4] POLÁK, L. *Digitální vysílání a videotechnika: Přednášky.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií.

[5] MURAT TEKALP, A. *Digital Video Processing, 2nd Edition.* Pearson, 2015. ISBN 0-13-399100-8

[6] AKRAMULLAH, S. *Digital Video Concepts, Methods, and Metrics: Quality, Compression, Performance, and Power Trade-off Analysis.* Berkeley: Apress Media, 2014. ISBN 978-1-4302-6713-3

[7] JIROUŠEK, R.; IVÁNEK, J., MÁŠA, P., TOUŠEK, J., VANĚK, N *Principy digitální komunikace.* Voznice: Leda, 2006. ISBN 80-7335-084-x

[8] NVIDIA . *NVIDIA Jetson Nano DATA SHEET* [specification]. Copyright © 2020 NVIDIA Corporation [cit. 1. 10. 2022].

[9] NVIDIA. *Embedded Systems Developer Kits & Modules from NVIDIA Jetson* [online]. Copyright © 2022 NVIDIA Corporation [cit. 1. 10. 2022]. Dostupné z URL: <`https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/`>

[10] NVIDIA. *NVIDIA Developer Center* [online]. Copyright © 2022 NVIDIA Corporation [cit. 1. 10. 2022]. Dostupné z URL: <`https://developer.nvidia.com/embedded/develop/`>

[11] NVIDIA. *NVIDIA Documentation Center* [online]. Copyright © 2022 NVIDIA Corporation [cit. 1. 10. 2022]. Dostupné z URL: <`https://docs.nvidia.com/#nvidia-jetson-software_jetpack/`>

[12] NVIDIA . *NVIDIA Jetson Nano Developer Kit Carrier Board* [specification]. Copyright © 2018 NVIDIA Corporation [cit. 1. 10. 2022].

[13] WU, Elaine. *NVIDIA Jetson Nano Developer Kit Detailed Review.* [online]. Copyright © 2018 Seeed Studio [cit. 1. 10. 2022]. Dostupné z URL: <https://www.seeedstudio.com/blog/2019/04/03/nvidia-jetson-nano-developer-kit-detailed-review/>

[14] NVIDIA . *NVIDIA Jetson Nano Developer Kit User Guide* [specification]. Copyright © 2012 NVIDIA Corporation [cit. 1. 10. 2022].

[15] NVIDIA. *NVIDIA ACCELERATED GSTREAMER USER GUIDE* [online]. Copyright © 2019 NVIDIA Corporation [cit. 1. 10. 2022]. Dostupné z URL: <https://developer.download.nvidia.com/embedded/L4T/r32-2_Release_v1.0/Accelerated_GStreamer_User_Guide.pdf>

[16] GStreamer: open source multimedia framework. *Application Development Manual.* [online]. gstreamer.freedesktop.org [cit. 1. 10. 2022]. Dostupné z URL: <https://gstreamer.freedesktop.org/documentation/application-development/>

[17] RAVIK, Haakon Wilhelm. *A Real-Time Video Retargeting Plugin for GStreamer* . Master thesis. University of Oslo. 2016 [cit. 1. 10. 2022]. Dostupné z URL: <https://www.duo.uio.no/handle/10852/53014>

[18] RidgeRun *GstShark Wiki.* [online]. RidgeRun [cit. 1. 10. 2022]. Dostupné z URL: <https://developer.ridgerun.com/wiki/index.php?title=GstShark>

[19] *USB video device class.* [online]. Wikipedia [cit. 1. 10. 2022]. Dostupné z URL: <https://en.wikipedia.org/wiki/USB_video_device_class/>

[20] *UP HD camera.* [online]. UP Shop [cit. 1. 10. 2022]. Dostupné z URL: <https://up-shop.org/up-hd-camera.html>

[21] U-BLOX. *NINA-W10 series - Data sheet* [online]. Copyright © 2022 u-blox [cit. 1. 5. 2023]. Dostupné z URL: <https://content.u-blox.com/sites/default/files/NINA-W10_DataSheet_UBX-17065507.pdf>

[22] U-BLOX. *NINA-W10 series - System integration manual* [online]. Copyright © 2023 u-blox [cit. 1. 5. 2023]. Dostupné z URL: <https://content.u-blox.com/sites/default/files/NINA-W1_SIM_UBX-17005730.pdf>

[23] ESPRESSIF . *ESP-IDF Programming Guide* [online]. Copyright © 2023 ESPRESSIF [cit. 1. 5. 2023]. Dostupné z URL: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/>

[24] Secure Shell. Wikipedia: the free encyclopedia. [online]. Copyright © 2023 San Francisco (CA): Wikimedia Foundation [cit. 1. 5. 2023]. Dostupné z URL: <https://cs.wikipedia.org/wiki/Secure_Shell>

[25] File Transfer Protocol. Wikipedia: the free encyclopedia. [online]. Copyright © 2023 San Francisco (CA): Wikimedia Foundation [cit. 1. 5. 2023]. Dostupné z URL: <https://cs.wikipedia.org/wiki/File_Transfer_Protocol>