INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

FORMULA STUDENT RACING CHAMPIONSHIP: IMPLEMENTATION OF THE ON-BOARD VIDEO ACQUISITION AND ENCODING SYSTEM

Sílvio Fernandes Calunda

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Júri

Presidente:	Doutor Marcelino Bicho dos Santos
Orientador:	Doutor Nuno Filipe Valentim Roma
Vogais:	Doutor Nuno Filipe Valentim Roma
	Doutor Moisés Simões Piedade
	Doutor Francisco André Corrêa Alegria

Outubro de 2010

Acknowledgments

I would like to thank Prof. Nuno Roma for giving me the opportunity to work under his supervision, for believing in me and for being so helpful and always available to help me throughout the entire project.

A very special thanks to Paulo Mendes, who has been a very good friend and very supportive when I need him most.

For you, my relatives and friends who have always supported me, I hope I have made you all proud.

Abstract

A complete on-board audio and video acquisition and encoding system, to be integrated in a Formula Student racing prototype, was developed and implemented in the scope of this dissertation. Real-time on-board video scenes, taken from several possible positions in the car, are captured with an off-the-shelf low cost webcam and subsequently transmitted to the pits by using a dedicated Wi-Fi communication infrastructure. To comply with the strict restrictions imposed by this application environment, the conceived system was implemented by using a mini-ITX embedded board, equipped with an ultra low-power Via C7 processor, running a small Linux distribution installed on a flash-based memory device. According to the conducted experimental assessment, the conceived system is able to capture and encode video streams at a rate of 14 fps using the MPEG-4 video standard.

Keywords

- On-board Audio and Video;
- Multimedia Signals Acquisition and Encoding;
- Multimedia Streaming;
- Portable Embedded Systems.

Resumo

Um sistema completo de acquisição e codificação de áudio e vídeo a bordo, foi desenvolvido no âmbito desta dissertação, para ser integrado no protótipo de um carro de corrida, numa competição designada por Fórmula Student. As sequências de vídeo a bordo em tempo real são adquiridas com uma webcam comercial e de baixo custo, e são posteriormente transmitidas para as boxes, utilizando uma infra-estrutura de comunicação Wi-Fi dedicada. Para cumprir com as rígidas restrições impostas no âmbito desta aplicação, o sistema concebido foi implementado integrando uma placa mini-ITX, equipada com um processador VIA C7 de muito baixa potência, executando uma pequena distribuição Linux instalada num dispositivo baseado numa memória flash. Segundo a avaliação experimental realizada, o sistema concebido é capaz de capturar e codificar streams de vídeo numa taxa de 14 fps usando o padrão de vídeo MPEG-4.

Palavras Chave

- Áudio e Vídeo a Bordo;
- Aquisição e Codificação de Sinais Multimédia;
- Fluxo de Multimédia;
- Sistemas Embebidos Portáteis.

Contents

List of Figures

List of Tables

IST Instituto Superior Técnico

PC personal computer

TV television

FIA Federation Internationale de l'Automobile

HD High-definition

POV Point-Of-View

GPS Global Positioning System

SD Secure Digital

USB Universal Serial Bus

DVD Digital Versatile Disc

ASP Advanced Simple Profile

PAL Phase Alternating Line

NTSC National Television Standards Committee

MPEG-4 Motion Picture Experts Group Layer-4 Video

MP2 MPEG-1 Audio Layer II

DS0 Digital Signal 0

IDE Integrated Device Electronics

SATA Serial Advanced Technology Attachment

CRT Cathode Ray Tube

DVI Digital Video Interface

LVDS Low-voltage differential signaling

CPU central processing unit

SCSI Small Computer System Interface

CD Compact Disc

ALSA Advanced Linux Sound Architecture

OSS Open Sound System

UVC USB Video Class

A/D Analog-to-Digital

MP3 MPEG-1 Layer III

AAC Advanced Audio Coding

FLAC Free Lossless Audio CODEC

fps frames per second

SECAM Séquentiel Couleur Avec Mémoire

NTSC National Transmission Standards Committee

CCD Charge-Coupled Device

CMOS Complementary Metal-Oxide Semiconductor

PCM Pulse-Code Modulation

WAVE Waveform Audio File Format

MC Motion compensation

DVB Digital Video Broadcasting

UWB Ultra-wideband

WPAN Wireless Personal Area Network

WLAN Wireless Local Area Network

ISM Industrial, scientific and medical

RF Radio Frequency

DSP Digital Signal Processor

API application programming interface

POSIX Portable Operating System Interface

MIDI Musical Instrument Digital Interface

SMP Symmetric Multiprocessing

D/A Digital-to-analog

DMA Direct memory access

IRQ Interrupt Request

FM Frequency Modulation

VBR variable bitrate

RDS Radio Data System

VBI Vertical Blanking Interval

AM Amplitude Modulation

JPEG Joint Photographic Experts Group

JFIF JPEG File Interchange Format

RGB Red, Green, Blue

ASCII American Standard Code for Information Interchange

PPM Portable Pixel Map

PGM Portable Gray Map

GIF Graphic Interchange Format

PNG Portable Network Graphics

TIFF Tagged Image File Format

Targa Truvision Advanced Raster Graphics Array

QP quantization parameter

DCT Discrete Cosine Transform

QP quantization parameter

RLE run-length encoding

CIF Common Intermediate Format

QCIF Quarter Common Intermediate Format

VGA Video Graphics Array

1

Introduction

Contents

1.1 Scope

The FST Project was created in 2001 by a group of students from Instituto Superior Técnico (IST), in order to design and implement a prototype to participate in the European Formula Student competitions, also known as the "University Formula One". The team started its participation in an UK Formula Student competition, in 2002, with its first prototype, the FST 01.

With nine years of existence, the FST project has already built three prototypes. In 2008, the FST 03 was completed and participated in the German Formula Student competition, and it was the first Portuguese prototype to complete the most important and dynamic proof, the Endurance. In 2009, a new team joined to the FST Project, who is working at the moment in the development of the first Portuguese electrical Formula Student prototype, the FST 04e [1].

Figure 1.1: Dynamic test of the FST 03 prototype, by Pedro Lamy, in Kartódromo Internacional de Palmela, Portugal (01/04/2009).

1.2 State Of The Art and Available Technologies

Communications are vital in any business. In Formula One, most communications are supported by radio systems. Although radio systems are not as highlighted as often are other parts of the car, such as the engine or the tyres, they play a major role to the success of any team. The in-car communication system allows the driver to give and receive information, while out on track.

1.2.1 Audio System

Formula One teams adopt radio systems to exchange vital messages between the pits and the drivers. Before the introduction of radio communications, there was a time when everything was done by pit signals. If a team wanted to send a message to a driver, he had to read it from the pit board, and he only had one chance every lap to do so, by which time it might be too late. Pit

Figure 1.2: Driver's earplugs and microphone.

boards are still in use nowadays, but the most important communications are mainly transmitted by radio [2].

Today, the audio system has three main functions [3]:

- It allows the team to send new instructions to the driver; this may either be a new strategy, information about the driver's position, alert about unexpected events or conditions, etc.
- It is used by the driver to send feedback to the engineers, prior to a pit stop, allowing appropriate set-up changes to be prepared by the engineers and mechanics.
- In extreme circumstances, its use is used to warn about an unscheduled stop caused by a racing incident, puncture or system failure.

In addition to the on-board radio equipment, the driver often wears earplugs (see Figure 1.2) and a microphone is placed inside the helmet. The microphones are very small - something like 5 mm diameter and 2 mm thickness - and extremely light.

Communicating in the race environment is not easy at all. Audio noise (produced by the loud engines), mechanical noise (caused by the vibration of the car), and electrical noise (generated by electrical circuits, such as generators and voltage regulators) may have to be handled. To circumvent this, the engineers currently use noise-canceling technology, that requires double-face microphones. One of the microphone faces is directed to the noise, while the other face is used to capture the driver's voice. As such, there are two signals to be processed: noise and voice. The noise is suppressed by comparing and subtracting both signals.

Adding to all these problems is the huge number of radios operating at the same time, which may generate interference. In order to solve some of the inconveniences often introduced by analog systems, some teams have migrated their systems to the digital domain. As an example, the McLaren team has been working for many years alongside the official supplier Kenwood, in order to develop its own in-car communication system [4]. The MP4-19B's radio system, designated by CBX-780, was located under the drivers' knees on the floor of the car, while the microphone and ear pieces were fitted inside the helmet. At the pits, there is a digital intercommunication system which converts the audio into digital domain, a controller implemented by personal computer (PC) for monitoring and set-up of the system, and repeater units which broadcast the radio signals

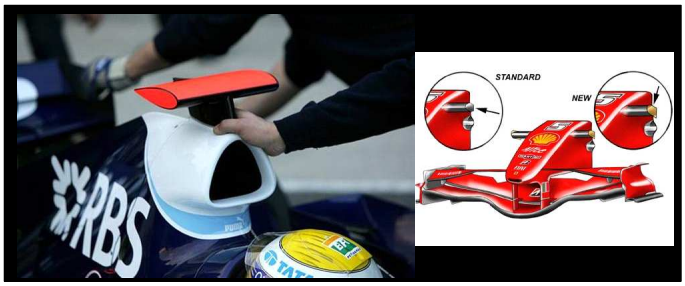

Figure 1.3: On-board camera placements.

across the circuit [3]. Digital radio systems are also more suited for encrypting data [4]. As an example, SELEX Communications was selected by Scuderia Ferrari to develop and supply a secure solution for radio communications for the 2006-2008 racing seasons [5, 6].

1.2.2 Video System

On-board video systems tend to be less focused by the Formula One teams than audio systems. One of the main reasons is because it does not play such an important role in win races as the other systems do. Therefore, it is very difficult to find information about the currently adopted video systems. In practice, on-board cameras are usually used to give a better perspective from the driver's point of view.

In the early days, cameras were only mounted on F1 cars during testing, but since the 80's, more and more cameras have been fitted. Currently, along the Grand Prix weekend, all cars must be fitted with two cameras or camera housings to provide on-board television (TV) footage. The cars have a total of six camera mounting points. One camera housing is always mounted on top of the air box, immediately behind the driver's head. The position of the second housing is chosen by the Federation Internationale de l'Automobile (FIA) in consultation with the relevant team and driver [7]. See some examples in Figure 1.3.

Some companies, such as TV2 International, are providing broadcasters and production companies with the latest compact digital microwave systems, which are capable of transmitting live images from an on-board camera directly to a small number of ground receivers, avoiding the need for a helicopter to relay the signals [8]. Broadcast Sports, a North American provider of compact, wireless High-definition (HD) broadcast camera systems, claims to be one of the first outdoor broadcast companies to employ Point-Of-View (POV) cameras for auto racing. The company has announced that it has installed Cavium's PureVu H.264 HD video processors in race cars [9, 10]. The PureVu Video Processor compresses 1080p quality video to sub 50 Mbps, capable to be sent with high reliability over a standard Wi-Fi network [11].

1.2.3 Industrial and Commercial Prototypes

There are not many commercial products available of in-car video and audio communication systems. Usually, Formula One establishes partnerships with multinational companies which provide these services, and the technologies that they use are usually kept secret.

Nevertheless, some prototypes are publicly available. As an example, Racelogic, a designer and manufacturer of *cutting edge* electronic systems for the automotive industry, based in the United Kingdom, has developed the Video VBOX in-car video system [12]. This equipment combines a powerful Global Positioning System (GPS) data logger with a high quality solid-state video recorder. The system takes multiple cameras and combines them with a graphical overlay. The resulting video is streamed onto a SD card or USB drive, with a DVD quality level.

The videos recorded by Video VBOX adopt the .avi multimedia container, meaning that they can be played in any standard media player. In order to enhance the video and graphics encoding efficiency, MPEG-4 compression is used to achieve the best balance between quality and bandwidth.

In the following, it is presented the most important features:

- Video Standard - Motion Picture Experts Group Layer-4 Video (MPEG-4) Advanced Simple Profile (ASP) at level 5 (at about 600 KB/s up to 2 MB/s);
- Resolution Options - PAL DVD, with 720 x 576 pixels at 25 frames per second (default) and NTSC DVD, with 720 x 480 pixels at 30 frames per second;
- Sound - Two external mono microphone connections, mixed and recorded as mono audio inputs in the video file. MPEG-1 Audio Layer II (MP2) encoded into the video stream;
- Storage Options - Secure Digital (SD) card and/or optional USB adapter for USB flash drives;
- Video VBOX Set-Up Software - Windows configuration software for screen layout and recording options.

Video VBOX is available in two different versions: the Video VBOX Pro and the Video VBOX Lite (see Figure 1.4). Both are designed to be rugged and reliable, and can be used on cars, bikes, boats and planes. However, these products are quite expensive to be used by FST Project. As an example, the “Video VBOX Lite and Single Camera Kit” option costs about €1250. The “Video VBOX Pro and Two Camera Kit” costs more than €2500.

1.3 Objectives

The main goal of this project is to develop a set of software components, integrated in a single application, that handles the interface with the hardware that deal with the audio/video acquisition,

Figure 1.4: Racelogic Video VBOX Pro (Left) and Video VBOX Lite (Right).

as well as with the communication system installed in the car, in order to acquire audio and images/video data and send it to the pits through a wireless connection. All the acquired data should be properly encoded and compressed before transmission, so that the required bandwidth can be as reduced as possible. Moreover, the team engineer should also be able to remotely control the data transmission using a simple interface tool. Therefore, the developed functionalities are as follows:

- Two-way audio communication;
- One-way image/video communication.

To achieve this goal, the following tasks must be accomplished:

- Signal acquisition using a microphone and a webcam;
- Signal encoding;
- Signal transmission through the communication channel.

To circumvent the physical constraints of the application (e.g.: vibration, space restrictions, etc.), as well as to handle the access and the interface with the peripherals, an embedded Linux distribution should be adopted. It is also necessary to develop an efficient user interface of this system to be used at the base station.

1.4 Original Contributions

The main contributions of this project are:

- To provide a low-cost and open source solution for in-car audio and video systems: these kind of systems usually cost about 10 times more than the budget for this project;
- To offer many design options for data encoding: the characteristics of the conceived system may be simply changed and/or configured by the programmer , by changing a simple line of code where the parameters of the codecs are set;
- To offer flexibility, quality and low transmission rates.

1.5 Document Organization

The first chapters act as a brief introduction to on-board audio and video systems and present some of the available technologies and architectures that will be used to implement this project. The middle chapters feature the actual implementation of the system and of the user interface. The final chapters focus on the evaluation of the system's performance.

Chapter 1 presents the Formula Student Racing Championship, the available technologies for on-board audio and video in Formula One and commercial prototypes. Chapter 2 discusses the requirements that the system must meet and the best solutions for hardware (to be installed in the racing car) and software (for signal acquisition, encoding and streaming). Chapter 3, 4, 5 focus on programming audio and image/video devices in Linux, as well as the implementation of the audio and image/video acquisition and encoding systems. Chapter 6 features the integration of the system and its implementation, covering aspects concerned with the transmission system, the hardware for the wireless communication and the structure of the messages to be sent. Moreover, it also presents the user interface, how it works and how to configure it. This chapter also shows the several controls and parameterization options available to the user. Chapter 7 reports the performance analysis evaluation of the system, the conducted testing results and draw the final conclusions.

2

System Architecture

Contents

2.1 Introduction

The system architecture is based on a client/server structure, implemented using a direct host-to-host communication network. This chapter discusses the main elements that constitute the system, such as the components that communicate through a wireless connection (most commonly referred to as *stations* or *nodes*); the hardware and software architecture for each station; the requirements that the system must meet and the protocol used in the signals transmission.

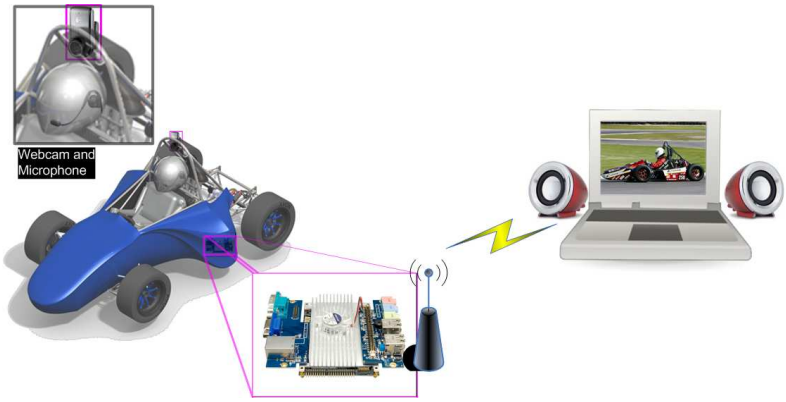

Figure 2.1: System architecture.

2.2 System Architecture

The implemented system is composed by two nodes: the **mobile station** (installed in the racing car) and the **base station** (Pit Wall[1]).

Figure 2.2 shows a block diagram that illustrates the several software modules that constitute the mobile station and therefore, must be implemented. The *pits* block represents the base station, to which the mobile station should communicate, through a wireless link. The next block represents the first stage within the mobile station - the *message formating and protocol implementation*. The message protocol was defined by the FST Project team and it is suitable for all signals being transmitted between both stations, which includes other sensors that are not discussed in this project. This subject will be further discussed in Section 6.4.1.

In the second stage of the mobile station are represented two multiplexing blocks: the *incoming data multiplexing* and the *outgoing data multiplexing*. The first block de-multiplexes the data comming from the *pits*, which is either audio data or user control signals. The incoming audio is encoded and compressed, in order to reduce the transmission bandwidth needed to stream it. Therefore, the next stages involves decoding and playback the audio data. The last stage (the fifth

[1] In Formula One Pit Wall is where the team owner, managers and engineers stay during the race and communicate with the driver.

stage), represents the hardware used by the driver in order to listen to the sound being received - the earplugs. This hardware was previously described in Section 1.2.1. The controller signals (listed on the bottom left side of the diagram) are briefly resumed in Section 2.3.

The second block of the multiplexing stage collects the signals being sent from the mobile station to the *pits*. These signals, which are audio and images/video signals, are being captured from the driver's microphone and the webcam, respectively. Both the microphone and the webcam will be further discussed in Section 2.6. Furthermore, the signals are encoded before they reach the multiplexing stage, in order to be transmitted to the pits.

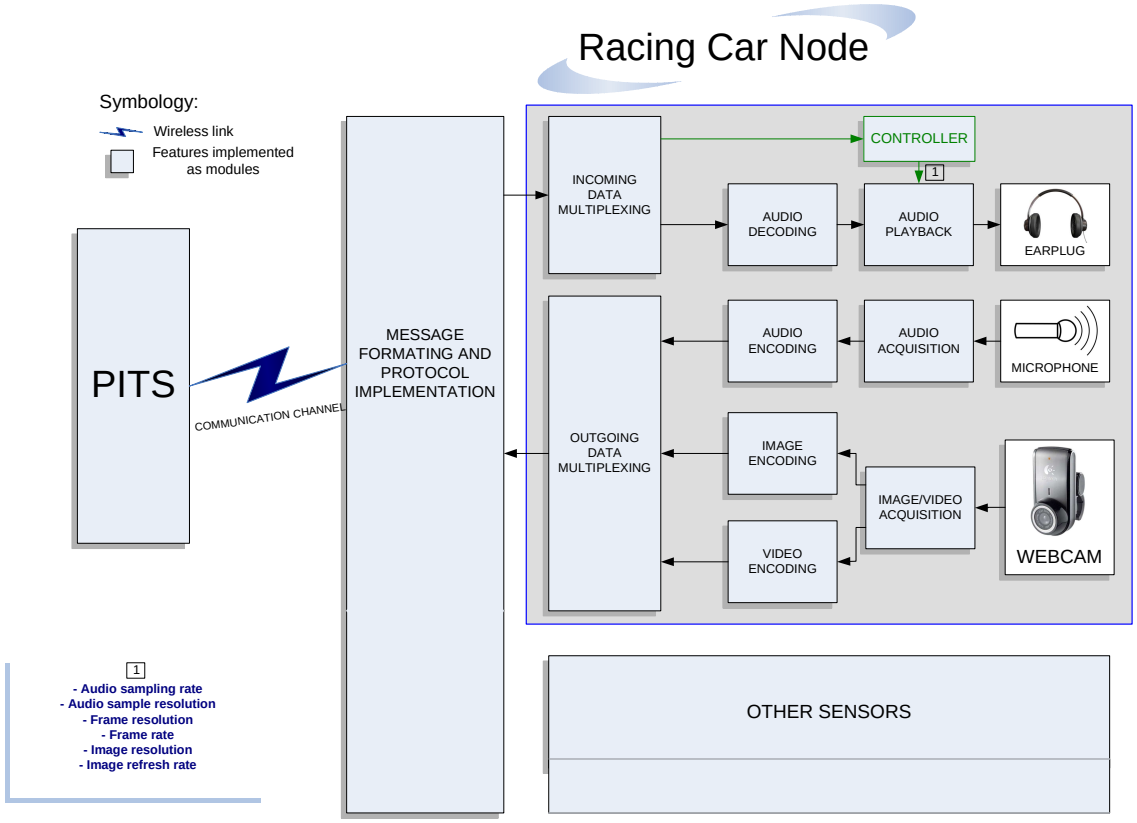

Figure 2.2: Mobile station block diagram.

Figure 2.3 shows a block diagram that illustrates the several software modules that constitute the base station. The *car* block represents the mobile station previously described. In the base station, the *outgoing data multiplexing* block is multiplexing either the audio signal that is being captured from the team engineer's microphone, or the the control signals that he is generating using the conceived user interface, which will be further discussed in Section 6.5. These signals will be sent to the mobile station. The audio data must be encoded before being transmitted.

The *incoming data multiplexing* block is de-multiplexing either the encoded audio data or the encoded images and video data being received. As such, this data must be decoded in order to playback its content. The decoded audio signal shall be sent to the speakers on the team

engineer's laptop, so that he can listen to the driver. The decoded images and video signals will be presented in the user interface screen.

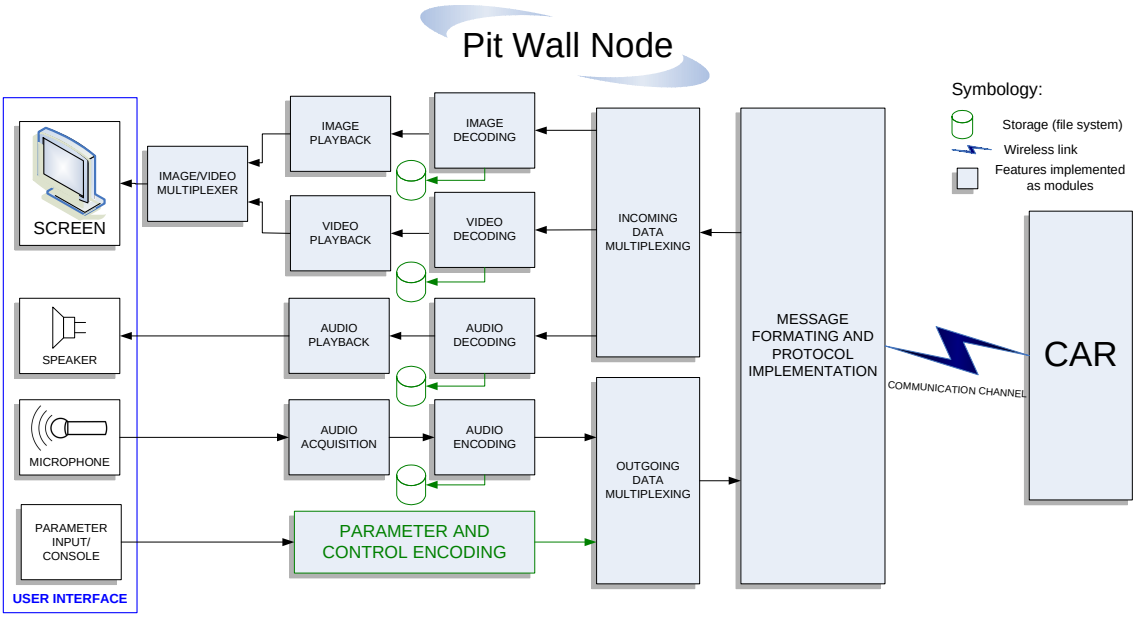

Figure 2.3: Base station block diagram.

2.3 Requirements

To achieve the proposed objective, several requirements must be met in order to circumvent the constraints of the application. In the following paragraphs it will be given a brief description of each of these requirements.

Physical constraints

Similarly to most components in a Formula One racing car, the in-car audio/video communication system must be as small and light as possible, which makes the packaging of the equipment an important consideration. In addition, the system must be robust to vibration.

Communication system

The data/messages exchanged in the car/pit network must use a pre-defined protocol which will be further discussed in Section 6.4.1. Since the communication is achieved through a Wi-Fi connection, network limitations such as a short communication range (about 60 m) and a 54 Mbps maximum data rate must be taken into account when implementing the system.

Bandwidth Requirements

All the acquired data must be properly compressed before transmission, so that the required bandwidth can be as reduced as possible. In telephony, a standard audio signal for a single phone call is encoded as 8000 analog samples per second, of 8 bits each, giving a 64 kbit/s uncompressed Pulse-Code Modulation (PCM) signal known as Digital Signal 0 (DS0) [13].

In case of video signal, let us enumerate some technologies for purposes of comparison:

1. Videophone (QCIF, 10 fps): 8-16 kbps(or 1-2 KB/s);
2. Video conference (CIF, 15 fps): 128-384 kbps (or 16-48 KB/s);
3. Video CD (CIF, 30 fps): 1.25 Mbps (or 156.25 KB/s).

Computational restrictions

The computational restrictions are also very important, requiring that the executed software applications shall be as "light" as possible. The processor must be fully compatible with Linux operating systems, in order to easily handle the access to the peripherals. It must also be small and combine robust performance with ultra low power consumption and highly efficient heat dissipation.

Operating system

The operating system should be installed in an USB flash-disk, rather than in a conventional hard drive, which would be impossible in rough conditions such as the car vibration. Universal Serial Bus (USB) flash disks do not have this limitations. Moreover, they allow the storage of the acquired data, since they are not read-only devices. The adopted operating system must also have available the drivers for audio/video acquisition and for the webcam.

User interface

The user interface shall be as simple as possible and must allow the user to configure the most important parameters of audio and image/video which are resumed in Figure 2.4.

Webcam

The camera must have a USB interface and must be able to acquire images/video at a high frame rate.

Option		Parameter Discription
IMAGES		
Capture Images (snapshots) in JPEG format	Refresh Rate	Set delay (in seconds) between Image acquisitions
	Resolution	Set frame heigth in pixels
		Set frame width in pixels
VIDEO Stream		
Capture a continuos sequence of video	Frame Rate	Set frame rate. Range is [0.2 - 30] fps. Optimal value is 14 fps
	Resolution	Set frame heigth in pixels
		Set frame width in pixels
AUDIO Stream		
Capture a continuos sequence of audio	Sampling Rate	Set sampling rate. The lower and upper limit are 4000 Hz and 48000 Hz

Figure 2.4: Configurable signal parameters.

Budget

The budget imposed by the FST championship is very low. Consequently, the hardware that composes the communication system must be acquired with particular attention to this requisite.

2.4 Racing Car (Mobile Station)

The mobile station includes an ultra compact Pico-ITX board connected to a Wi-Fi USB network adapter. This computational board is responsible for acquiring the audio and image/video signals from the car (through a microphone and a webcam), encode and compress the acquired signals, store the encoded data in a local flash memory and/or send it to the laptop/computer at the pits through the wireless network.

2.4.1 Hardware

The prototype vehicle is equipped with a 12 V battery that powers all the electrical systems and is recharged by the alternator attached to the engine. A VIA EPIA-P700 Pico-ITX board [14] was selected and mounted in the car in order to meet the involved computational restrictions. It is an extremely compact 10 cm x 7.2 cm Pico-ITX board, with a 1GHz VIA C7® processor [15]. Such feature also meets the constraints of space and weight (weighing about 455 grams). The system memory consists of a 667/533 MHz DDR2, with 1GB of storage space. The C7 processor is a native x86 platform, fully compatible with Microsoft® and Linux operating systems, which will be further discussed in Section 2.4.2.A. In addition to the processor, the main board also includes several peripherals and accessories, as shown in Figure 2.5.

The main key features are the following:

- Incorporates one Integrated Device Electronics (IDE) and one Serial Advanced Technology Attachment (SATA) controllers;

- Incorporates one Gigabit Ethernet adapter;
- Support for Cathode Ray Tube (CRT), Digital Video Interface (DVI) and Low-voltage differential signaling (LVDS) displays;
- Four USB 2.0 ports;
- Support for HD audio;
- 12V DC-input;
- Low profile design: all back panel I/O in pin header format, except for the IDE and SATA connectors.

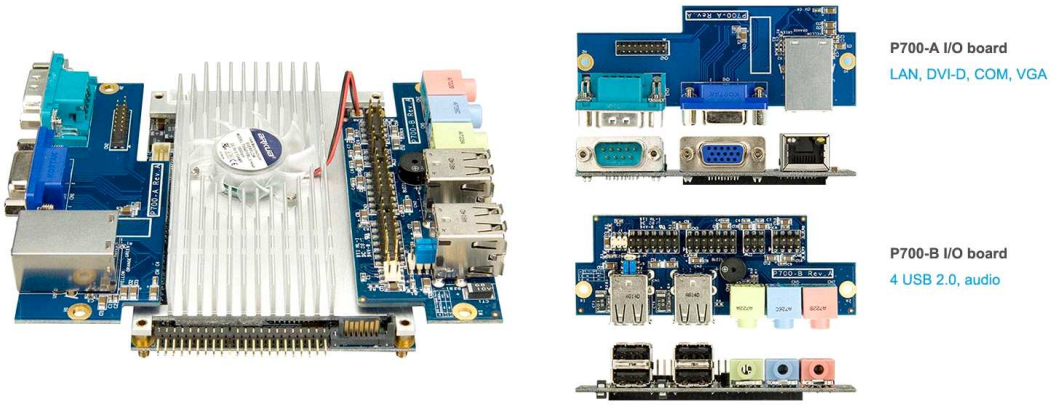

Figure 2.5: EPIA-P700 accessories.

2.4.2 Software

Besides the operating system, the installed flash memory also stores the developed application software that controls both the microphone and the webcam and encodes the acquired signals.

2.4.2.A Operating System

One of the advantages of using an operating system is that it greatly facilitates the access to the peripherals, along with other procedures that control the resources of the board. The selected VIA EPIA-P700 embedded board is fully compatible with most versions of Microsoft® and Linux operating systems, as previously mentioned.

When choosing the operating system to be adopted, the relevant features that should be considered for this particular project are:

- The price (the lower the better);
- The computational requirements, in terms of central processing unit (CPU) and memory usage;
- The size and storage requirements;
- The ability to be installed in a USB flash-disk drive, with a minimalistic configuration, in order to circumvent the physical constraints (in particular, the vibrations).

Since the cost is a main concern of this project, the first choice favored the selection of a linux distribution over a Microsoft operating system, because its cost is almost free. Initially, there were two possibilities that were considered: the **iMedia Embedded Linux** and **Knoppix** distributions. They have quite similar features concerning the requirements of this project. In the following, it will be given a brief overview of both of these distributions.

iMedia Embedded Linux

iMedia Embedded Linux[16] is a hybrid between a small embedded linux distribution and a full featured linux distribution. While not having all the common restrictions of an embedded linux distribution (like compatibility issues, in what respects the hardware/architectures, non standard libraries, file systems, etc.) it is still much lighter in terms of disk space and on CPU and memory usage than a full featured linux distribution. As an example, a minimum installation of Fedora Linux distribution is about 400 MB while iMedia is about 8 MB. Moreover, iMedia distribution offers the possibility to run on most hardware platforms as a full featured linux distribution. iMedia is usually most suitable for low power purposes, such as embedded mini-ITX systems or embedded PC designs.

The main key features are:

- Flexible and customizable linux platform;
- Support for advanced wireless and home networking;
- Intuitive and user-friendly interface;
- Extensive multimedia tools to play and acquire most audio/video formats;
- Installs anywhere: USB stick, USB hard drive, USB ZIP, Compact Flash, SATA, Small Computer System Interface (SCSI), IDE, etc;
- Desktop reliability and security.

Knoppix

The KNOPPIX distribution is a compilation of GNU/Linux software, which runs completely from Compact Disc (CD), Digital Versatile Disc (DVD) or flash disks. It automatically detects and supports a wide range of graphics adapters, sound cards, USB devices and other common peripheral devices.

Knoppix was the chosen distribution, mainly because it is more popular and presents a higher development activity. Furthermore, its main key features meet the requirements for this particular project:

- Possibility of execution in a solid state storage media;
- Availability of drivers for audio capture, such as Advanced Linux Sound Architecture (ALSA) or Open Sound System (OSS) (these are discussed in Section 3.2;
- Availability of drivers for video acquisition, such as Video4Linux (these are discussed in Section 4.1;
- Availability of webcam drivers, such as USB Video Class (UVC) interface are also available. The UVC specification defines video streaming functionality on the USB. This specification is very important, since it allows the interconnectivity of webcams in computers even without the installation of proprietary drivers [17].

2.5 Pit Wall (Base Station)

2.5.1 Hardware

The base station consists of a PC (laptop), with an integrated Wi-Fi interface. The laptop at the pit wall is used to control, monitor and store the data received from the board installed in the racing car.

2.5.2 Software

The user controls the signal acquisition by “remote control”, through a simple interface which will be discussed in Section 6.5. In other words, the user at the pit controls, in real time, what is happening on track, by controlling the behavior of the board in the car. He decides *which* data to acquire, *when* and *how* the data is received (in what format).

2.6 Signal Acquisition

2.6.1 Audio Signals

Sound waves consist of continuous variations in air pressure. They are usually acquired using a microphone device. These transducers convert acoustic signals (sound waves) into electrical energy (the audio signal), as illustrated in Figure 2.6. However, the electrical current that is generated is usually very small (mV). To be useful for recording and processing purposes, the signal must be amplified. Unfortunately, any electrical noise produced by the microphone will be also amplified. As such, slight amounts of noise are intolerable. Noise also includes unwanted pickup of mechanical vibration through the body of the microphone, which is very common during a car race.

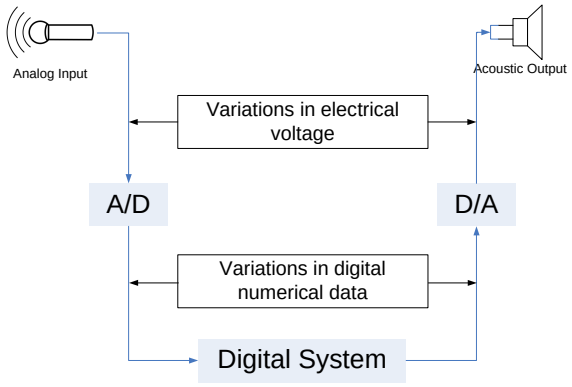

Figure 2.6: Analog to digital audio.

The variations in electrical voltage (analog signal) are then converted into variations in digital numerical data (digital signal) by an Analog-to-Digital (A/D) converter. The A/D runs at a specified sampling rate and converts the samples at a known bit resolution. For example, CD audio has a sampling rate of 44.1 kHz and 16-bit resolution for each channel. Microphones usually only provide one channel. For stereo signals there are two channels: “left” and “right”, which imposes the usage of two microphone devices.

The digital audio signal may then be either stored or transmitted. Digital audio storage can be on a CD, a hard drive, USB flash drive, or any other digital data storage device. Audio data compression techniques - such as MPEG-1 Layer III (MP3), Advanced Audio Coding (AAC), Ogg Vorbis, or Free Lossless Audio CODEC (FLAC) - are commonly employed to reduce the file size.

2.6.2 Image/video Signals

Video can be defined as an ordered sequence of images varying at a certain rate. The frame rate, which denotes the number of still pictures per unit of time, ranges from 6 or 8 frames per

second (fps) for old mechanical cameras to 120 or more frames per second for new high performance professional cameras. Several television standards have been defined around the world. The Phase Alternating Line (PAL) (Europe, Asia, Australia, etc.) and Séquentiel Couleur Avec Mémoire (SECAM) (France, Russia, parts of Africa etc.) standards specify 25 frame/s, while National Transmission Standards Committee (NTSC) (USA, Canada, Japan, etc.) specifies 29.97 frame/s.

Most analog cameras capture light onto photographic films, while digital cameras capture images using electronic image sensors to convert light into electrons (typically a Charge-Coupled Device (CCD) or Complementary Metal-Oxide Semiconductor (CMOS) sensor chip). After the conversion is done, the next step is to read the value (accumulated charge) of each cell in the image.

Given the requirements of this project, one single digital camera will be used. As such, the video acquisition is based on an USB webcam, interfaced to the processing board using the Video4Linux device driver, provided by the Linux distribution. This matter will be further discussed in Section 4.1. The selected camera is a Logitech Portable Webcam (QuickCam Pro for Notebooks) illustrated in Figure 2.7.

This camera fits most requirements, except for its design that makes it hard to fix in the racing car completely steady. Another important feature to be referred as well is that this camera is also UVC compatible[17].

This camera features:

- HD video capture (up to 960 x 720 [2] pixels);
- Up to 30 frames per second;
- Hi-Speed USB 2.0 certified;
- 5.08 cm width;
- 7.62 cm height;
- 227 grams weight.

2.7 Signal Encoding

Signal encoding is the process of putting a sequence of data into a special format suitable for transmission or storage purposes. The main goal is to represent data as efficient as possible (less amount of bits), with the best quality.

[2] This value was not observed experimentally, since it is only achieved in Windows platforms. The maximum resolution available is 640 x 480 pixels.

Figure 2.7: Logitech QuickCam Pro for Notebooks.

After acquiring the data there is the need to compress it, mainly because it is required to reduce the memory space and the transmission bandwidth needed to stream it. However, when transmitting compressed data, one must ensure that both the sender and the receiver understand the encoding scheme. Designing this data compression schemes involves many factors such as:

- the degree of compression;
- the amount of distortion that may be introduced;
- the computational resources required to compress and uncompress the data.

Usually audio can be easily compressed at a 10:1 ratio without perceptible loss of quality. For video the compression ratio is even higher, about 100:1. Such rates are more difficult to be achieved for still images: with a 10:1 ratio the quality loss is already perceptible.

2.7.1 Audio

As mentioned before, audio capture is achieved by using a microphone transducer. The captured data, represented in the analog domain, is then digitalized using a PCM.

To obtain a PCM from an analog waveform, the analog signal amplitude is sampled at regular time intervals using a sample-and-hold device. The sampling rate, or number of samples that are acquired per second, should be twice the maximum frequency of the analog waveform in cycles per second or Hertz, in order to be able to recovery - Nyquist-Shannon sampling theorem. The instantaneous amplitude of the analog signal at each sample is rounded off to the nearest of several specific and predetermined levels. This process is called *quantization*.

PCM is often used in the context of digital telephony, to represent speech signals (8000 Hz sampling rate, 8-bit quantization accuracy and 64 kbps[3] data rate). Nevertheless other sampling rates and quantization accuracies may also be adopted.

[3] **kbps** or kbit/s.

To reduce the bandwidth or storage space required to accommodate PCM audio signals, several data compression techniques may be applied. Among the most often used compression formats are the MP3, the AAC, the Ogg Vorbis, etc. These are lossy encoding techniques and they are quite efficient at compressing data. Anyway, there are some requirements that must be fulfilled:

- the decoded audio should sound as close as possible to the original audio before compression;
- low complexity to enable software decoders or inexpensive hardware decoders with low power consumption;
- flexibility for different application scenarios.

Nevertheless other lossless encoding formats are also available, such as Waveform Audio File Format (WAVE). Let us just say for now that despite the WAVE format being lossless, it offers no compression, so the main limitation is the size of the output file that is usually very large. Therefore, this format does not meat the requirements of this project.

2.7.2 Images and Video

In the current project, the hardware responsible for acquiring this type of signals is the webcam that was previously presented in Section 2.6.2. This digital camera converts the signal directly to a digital output, without the need of an external A/D converter. As will be further discussed in Chapter 4, these devices acquire the image/video data in a Raw pixel format, which is obtained almost directly from the image sensor of the camera. As a consequence, image and video data must be encoded in order to be useful for the intended purposes.

There are several different techniques in which image data can be compressed. These techniques may be *lossless* or *lossy*. Lossless compression means that the decompressed image is exactly the same as the original image. In lossy compression, the decompressed image is as close to the original image as possible, but not exactly the same. For Internet use, as well as for other applications with limited channel capacity, the two most common compressed image formats are the Joint Photographic Experts Group (JPEG) format (usually lossy) and the Graphic Interchange Format (GIF) format (lossless). In the following, it will be given a brief overview of the JPEG process. This format shall be further described in Chapter 4.

1. The image is broken into 8 x 8 blocks of pixels;
2. The Discrete Cosine Transform (DCT) is applied to each block;
3. Each block is compressed through quantization;

4. The quantized coefficients are encoded using a run-length entropic code.

The DCT applies a transformation from spatial domain to frequency domain. Therefore, each pixel from the 8 x 8 block is coded separately by transforming into an 8 x 8 frequency space. As such, more bits are assigned to some frequencies than others. This may be advantages because the eye can not easily see errors in the high frequencies, for example. For more compression, the higher frequencies can simply be omitted.

Quantization reduces the amount of information in the high frequency components, by dividing each component in the frequency domain by a constant value defined for that component, and then rounding to the nearest integer. Many of the higher frequency components are rounded to zero, and many of the rest become small positive or negative numbers, which takes many fewer bits to store. Quantization is the main reason for both the compression and the loss of quality. In particular, JPEG takes advantage of the fact that most components are rounded to zero, by encoding the components in a zig-zag pattern and by employing a run-length encoding (RLE) algorithm that groups similar frequencies together. Finally, it uses then using Huffman coding to encode the result of the RLE, with special purpose Huffman tables.

Video compression can be seen as a combination of spatial *image compression* and temporal *motion compensation* techniques to reduce the redundancy and irrelevancy of the image data. The redundancy is presented as correlated and predictable information between neighboring pixels (*spacial redundancies*), or similarities between adjacent frames (*temporal redundancies*). Its reduction generally conducts to *lossless compression* methods. On the other hand, the compression methods that reduce irrelevancy take advantage of the fact that part of the information is imperceptible to the human eye or even insignificant to the human brain. As a consequence, these techniques give rise to *lossy compression* methods. Besides quantization, *Chroma subsampling* is another example of this method and will be extensively explained in Chapter 4. Motion compensation (MC) describes a given image block in terms of translation of a corresponding block in a reference image. The reference image may be previous in time or even from the future. Figure 2.8 illustrates an example of a MC procedure.

Several video coding standards have been proposed in the last three decades, since the ITU-T H.120, presented in 1984 to the ITU-T H.264, presented in 2003 (refer to the timeline in Figure 2.9). The most popular applications include videoconferencing, videotelephony, DVD Video, Blu-Ray, Digital Video Broadcasting (DVB), video on mobile phones (3GP) and Internet (DivX, Xvid ...), among others. Figure 2.9 shows the evolution of video coding standards.

As it was referred before, H.264 is the latest digital video standard and is also expected to become the video standard of choice in the coming years. However, despite presenting a much greater encoding efficiency than MPEG-4 Part 2 (it needs fewer than up to 50% of the bits to encode a video sequence), it requires much higher computational requirements [18]. As a consequence, the strict computational restrictions imposed by the adopted computational system led to

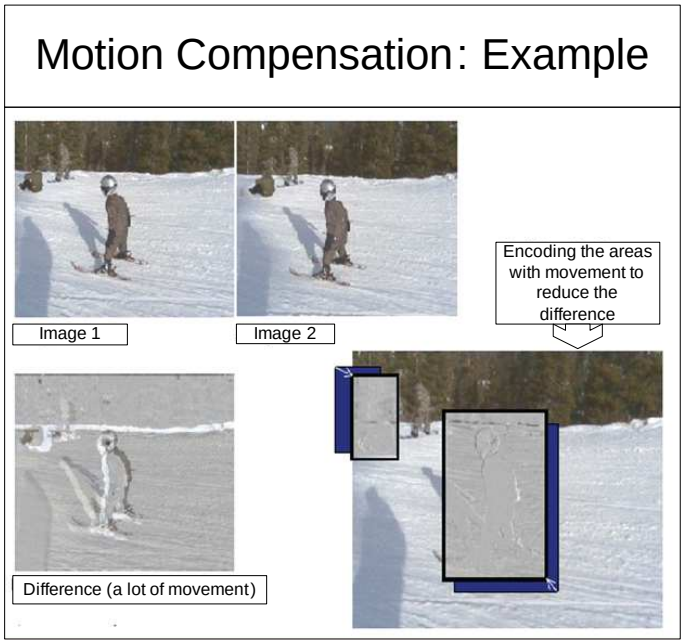

Figure 2.8: Motion compensation example.

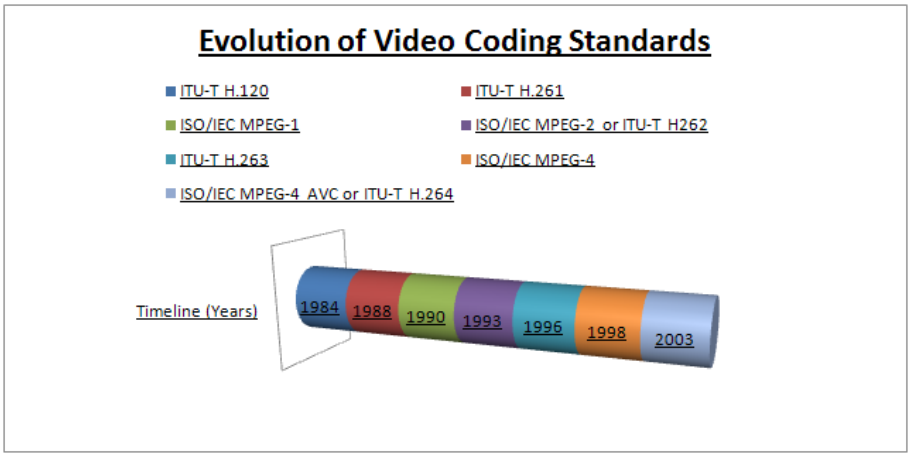

Figure 2.9: Evolution of video coding standards.

the selection of the MPEG-4 Part 2 encoder to be implemented in the conceived system.

2.8 Communication Link

Nowadays there is a wide variety of different wireless data communication technologies available. Some examples are Ultra-wideband (UWB), Bluetooth, ZigBee, and Wireless USB suitable for use in Wireless Personal Area Network (WPAN) systems. They are usually applied for short range communications between paired devices that are typically controlled by a single user. A keyboard might communicate with a computer, or a mobile phone with a handsfree kit, using any of these technologies. For broadband applications, **Wi-Fi** is the most successful technology intended for use as a Wireless Local Area Network (WLAN) system.

The previous prototypes of the FST Project adopted a ZigBee transceiver to transmit the data acquired from several sensors placed in the engine, suspension, etc. The most interesting features of these transceivers are low power consumption and low price. In particular, the Max Stream's XBee-Pro OEM was the selected module used at the time. It operates in the ISM 2.4 GHz frequency band and provides a reliable communication link between the mobile and the base station.

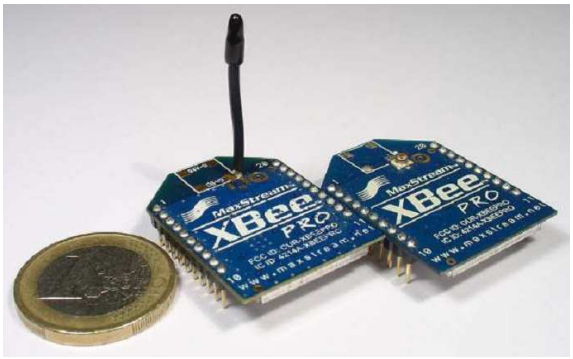

Figure 2.10: XBee-PRO transceivers fitted with a whip antenna and a male U.FL antenna connector.

However, these transceivers impose a very important limitation. The offered data rate is restricted to only 250 kbps as shown is Table 2.1 [19]. This makes it difficult to transmit encoded audio and video (mainly video), taking into account that the channel would be shared to send signals from the other sensors mentioned above.

As a consequence, in the current version of the FST prototype it was decided to migrate the communication technology and adopt the Wi-Fi technology in order to suppress such important bandwidth limitation. As it will be seen in Section 6.3.1, such technology reveals to be the most appropriate in what concerns the bandwidth and latency. Furthermore, the required transceiver devices are widely available in the market, with an acquisition cost that is affordable with the strict budget imposed by the FST championship.

Specifications	XBee-PRO
Indoor/Urban Range	Up to 100 m
Outdoor Line-of-Sight Range	Up to 1500 m
Transmit Power Output (software selectable)	60 mW (18 dBm) conducted
RF Data Rate	250 kbps
Supply Voltage	2.8 - 3.4 V
Operating Frequency	ISM 2.4 GHz
Dimensions	2.438cm x 3.294cm

Table 2.1: XBee-PRO specifications.

3

Audio Acquisition and Encoding

Contents

This chapter addresses the basics of audio programming in Linux. The three main operations that will have to be considered are capturing (recording), playing and encoding. The first two are implemented by handling the Digital Signal Processor (DSP) device. In the following sections it will be provided a brief description about the main Linux sound drivers and devices and how they are programmed in order to offer the first two options. The aim is to implement a number of applications to record, play and store the sound data for further processing. Then, it will be presented a current alternative to compress and store the audio information. At the respect, it will be described the usage of some of the current publicly available libraries of applications to encode and compress the sound signal.

3.1 Interaction With the Operating System

The most common approach to control a device under Linux consists in using a small set of functions, usually called *system calls*, implemented by the operating system's kernel. By using this standardized method of accessing the devices, the particular details of each device are masked to the programmers under a standardized and well defined interface, thus preventing the programmers from having to learn new functions for each type of device.

The *open* system call establishes the access to the device, returning a file descriptor to be used for subsequent calls. The *read* and *write* functions receive data from and send data to a device, respectively. Finally, the *ioctl* routine is a catch-all function to perform other operations that do not fit in the read/write model. In the following paragraphs it will be given a brief description of each of these system calls in more detail. Such an overview will facilitate the presentation about the set of operating system devices and interfaces with the device drivers that will follow, in the next sections.

The *open* system call

This system call follows the format:

```
int open(const char *pathname, int flags, int mode);
```

It is used to gain access to a system device, allowing the user to subsequently operate on it with other system calls. The device can be an existing one that is to be opened for reading, writing, or both. This function can also be used to create a new file. The *pathname* parameter specifies the name of the device to be opened. It can be either a regular file or a device file (such as */dev/dsp*, corresponding to the Digital Signal Processor).

The *flags* parameter indicates the mode that should be used for opening the file, and takes one of the following values:

- O_RDONLY - open for read only;

- O_WRONLY - open for write only;
- O_RDWR - open for both read and write.

In addition, some flags can be added (bitwise OR) with the ones above to control other aspects of open operation. A number of flags are defined, most of which are device-specific and not important to the discussion that is presented here.

The third *mode* parameter is optional - it specifies the permissions to be used when creating a new file and is only used when the O_CREAT option is given.

When successful, the function returns an integer file descriptor (a small positive number) to be used in subsequent system calls to reference the opened device. If the function fails for some reason, the call returns the -1 value and sets the variable *errno* with a value that indicates the reason for such failure.

The *read* system call

The prototype of this function is:

```
int read(int fd, char *buf, size_t count);
```

This system call returns data from a file or device. The first parameter is a file descriptor, obtained from a previous call to the *open* function. The *buf* parameter points to a memory, where the retrieved data will be stored. The *char* * definition for the buffer is a convenience way to cover all kinds of data. This argument is often casted to another data type, such as a data structure, that represents the particular kind of data that will be returned. The *count* parameter indicates the maximum number of bytes to be read.

If successful, this function returns the actual number of bytes that were read, which is sometimes less than *count*. On error, the value -1 is returned and the global variable *errno* is set to a value indicating the error cause.

Calling this function can cause a process to block until the data is available.

The write system call

The *write* system call is used to write some data to a file or device. It takes the following form:

```
size_t write(int fd, const char *buf, size_t count);
```

This function is analogous to read, but sends data to a file or device. The parameter *fd* is the opened file descriptor, *buf* points to the data to be written, and *count* indicates the number of bytes to be written. This function returns the number of bytes actually written, or -1 if an error occurred. Like the read call, the process may be blocked by the kernel until the data has been successfully written.

The ioctl system call

The *ioctl* system call is a catch-all function that takes the form:

```
int ioctl(int fd, int request, ...);
```

This function is used to perform miscellaneous operations on a file or device that do not fit into the *read* or *write* system calls. Each request may either set some behavior of the device, return information, or even both. As such, it is often device-specific.

The first parameter is a file descriptor, obtained when the device was opened. The second is an integer value indicating the type of ioctl request being made. There is usually a third parameter, which is dependent on the issued ioctl request being made.

The close system call

The last of the set of basic functions retrieved in this section follows the format:

```
int close(int fd);
```

The *close* system call notifies the kernel that the access to a file or device is no longer required, allowing any related resources to be freed up. Since there is usually a fixed limit on the number of files/devices that any process may open at any time, it is good practice to close the files or devices whenever they are no longer required.

3.2 Linux Sound

Drivers (OSS, ALSA)

Most Linux distributions include one of the following two sound drivers: the older OSS, which works with every UNIX-like system, and the newer ALSA.

The OSS provides a cross platform application programming interface (API) and device drivers for most consumer and professional audio devices for UNIX and Portable Operating System Interface (POSIX) based operating systems, including Linux. Owing to its open architecture, applications developed on one supporting operating system platform can be easily recompiled on any other platform [20].

On the other hand, the ALSA is a Linux kernel component for audio support in Linux. It is replacing the OSS and individual projects that were providing device drivers for sound cards. ALSA provides audio and MIDI functionality to the Linux operating system and offers the following significant features [21]:

- Efficient support for all types of audio interfaces, from consumer sound cards to professional multichannel audio interfaces.
- Fully modularized sound drivers.

- Symmetric Multiprocessing (SMP) and thread-safe design.
- User space[1] library (alsa-lib) to simplify the development of new applications and provide higher level of functionality.
- Support for the older OSS API, providing binary compatibility for most OSS programs.

ALSA provides better support for Linux and offers more features. Moreover, it is regarded as allowing a faster driver development. For such reason, this project was focused on the ALSA system.

Devices and Interfaces

Several analog inputs are usually provided - microphone, audio CD, and line level input are the most common. On the other side there are the analog outputs - usually, the speaker and line level outputs. An output mixer device combines the various signal sources and sends them to the output devices.

Similarly, an input mixer accepts several analog signals, combines them, and feeds an analog to digital converter to be digitized. Many sound cards also provide CD-ROM, Musical Instrument Digital Interface (MIDI), and joystick interfaces.

To develop new applications that access the sound devices using the driver API, the corresponding program source code should include the sound header file *linux/soundcard.h*. In the follows, it will be presented a brief overview about each of the devices provided by the sound driver. Due to the particular interest for the development of the current project, a special attention will be devoted to the */dev/dsp* (the DSP) device.

/dev/sndstat Device

The device */dev/sndstat/* is the simplest device provided by the sound driver. It is a read-only device file, whose only purpose is to report information about the sound driver in human-readable form. It is useful for checking the hardware configuration (Direct memory access (DMA) channel, Interrupt Request (IRQ) number, etc.) and for finding out the version of the sound driver being used.

/dev/audio Device

The */dev/audio* device is similar to */dev/dsp*. Which will be described bellow. It is intended to keep compatibility with the audio device on workstations made by Sun Microsystems, where it has the same name. The device uses μ-law encoding. The main purpose is to support basic user commands (such as *cat file.au >/dev/audio*) to play Sun μ-law encoded files.

[1] **User space** is the memory area where all user mode applications work.

As a consequence, it is better to use the */dev/dsp* interface instead. Only one of */dev/audio* and */dev/dsp* is available at any time, as they are just different software interfaces to the same hardware device. There is also a */dev/audio1* device available for those cards that have a second sampling device.

/dev/mixer Device

The mixer device contains two mixer circuits, the input mixer and the output mixer. The input mixer accepts analog inputs from a number of different signal sources (mixer channels). While some sound cards only allow one channel to be connected as a recording source, others allow any combination of inputs. Independently on the number of input channels, the signals are then to the mixer, which essentially adds them together. The resulting signal is then transferred to the A/D converter where it is digitized for further processing. It is worth noting that up until the A/D converter, all of the processed signals are in analog form.

The output mixer works in a similar manner. The several output signals are fed to the output mixer. After passing through a gain controller the signals are then transferred to the Digital-to-analog (D/A) converter. The last step is to send the resulting output signal to the speakers or other analog outputs.

Programming the mixer device consists in setting the desired levels for the gain controller and the several switches that select the recording source. Other than that, the mixer operates continuously, using scarce computing resources to operate. More than one process can open the mixer at a time.

/dev/sequencer Device

The */dev/sequencer* device allows the programming of Frequency Modulation (FM) or wavetable synthesizer built into the sound card or external devices connected to a MIDI bus. It is intended for computer music applications.

/dev/dsp Device

The */dev/dsp* is the digital sampling and digital converter device, and probably the most important for multimedia applications. Writing to the device accesses the D/A converter, to produce the sound. Reading from the device activates the A/D converter, for sound recording and analysis. Generally speaking the device should be opened either for read only or for write only. It is possible to perform both read and write on the device, although with some restrictions, which will not be covered in this project. As such, it is assumed that only one process can have the DSP device opened at a time. Attempts by another process to open it will fail with an error code of EBUSY.

The sound cards may use a dedicated DSP chip, or may implement the functions with a number of discrete devices. Some sounds cards provide more than one digital sampling device. In this case, a second device is available as */dev/dsp1*. Unless noted otherwise, this device operates in the same manner as the */dev/dsp*.

When an application program invokes the *read* system call, the sound data is transferred to a memory buffer of the calling program. The data acquisition is entirely controlled by the kernel driver, making the actual sampling rate absolutely defined and entirely independent of the speed at which the application program reads it. When reading from the */dev/dsp* there is no end-of-file condition. If data is read too slowly (less than the sampling rate), the excess data will be discarded, resulting in gaps in the digitized sound. If data is read too quickly, the kernel sound driver will block the process until the required amount of data is available.

The input source depends on the mixer settings (not addressed in this project); the default is the microphone input. The format of the digitized data depends on which the *ioctl* calls that have been used to set up the device. Nevertheless, each time the device is opened, its parameters are set to default values: 8-bit unsigned samples, using one channel (mono), and an 8 kHz sampling rate.

In a similar way, writing a sequence of digital sample values to the DSP device produces sound output. Again, the format can be defined using *ioctl* system calls, with defaults similar to those given above for the *read* system call (8-bit unsigned data, mono, 8 kHz sampling rate). In general the parameters should be set *after* opening the device, and *before* any calls to *read* or *write* sound data. Moreover, the parameters should be set in the order in which they are described below.

As it was referred in Section 3.1, all DSP *ioctl* calls take a third argument that is a pointer to an integer. Constant is not allowed; a variable must be used. The call will return -1 if an error occurs, and set the global variable `errno`.In the event that the hardware does not support the exact value of a specified parameter, the sound driver will try to set the parameter to the closest allowable value. Similarly, if a parameter is out of range, the driver will set it to the closest value (i.e., the upper or lower limit). For example, attempting to use a 16-bit sampling resolution with an 8-bit sound card will result in the driver selecting 8 bit, but no error will be returned.

All of the *ioctl* system calls parameters for the DSP device use identifiers starting with SOUND_-PCM. Calls in the form SOUND_PCM_READ_*XXX* are used to return just the current value of a parameter. To change the values, the *ioctl* calls use the general template SOUND_PCM_-WRITE_*XXX*. As discussed above, these calls also return the selected value, which is not necessarily the same as the value passed to the sound driver. The *ioctl* constants are defined in the header file *linux/soundcard.h*. Some of the most significant are briefly described below:

SOUND_PCM_WRITE_BITS - Sets the sample resolution, in bit. Valid choices are 8 and 16, but some cards do not support 16.

SOUND_PCM_READ_BITS - Returns the current sample resolution, which should be either 8

or 16 bits.

SOUND_PCM_WRITE_CHANNELS - Sets the number of channels: 1 for mono, 2 for stereo. When running in stereo mode, the left and right data sampled are interleaved when read or written, in the format left-right-left-right.... Some sound cards do not support stereo.

SOUND_PCM_READ_CHANNELS - Returns the current number of channels, either 1 or 2.

SOUND_PCM_WRITE_RATE - Sets the sampling rate in samples per second. All sound cards have a limit on the allowed range; the driver will round the rate to the nearest frequency supported by the hardware, returning the actual (rounded) rate in the argument. Typical lower limits are 4 kHz; upper limits are 13, 15, 22, or 44.1 kHz.

SOUND_PCM_READ_RATE - Returns the current sampling rate. This is the rate used by the kernel, which may not be exactly the rate given in a previous call to SOUND_PCM_WRITE_RATE, because of the previously discussed rounding.

3.3 Sound Programming in Linux

Programming the sound interface in Linux usually corresponds to the development of software code that implement the capturing (and recording) and playback procedures. In the following subsections, it will be given a brief overview of the involved tasks and configuration procedures.

3.3.1 Capture (Recording)

The implementation of the procedures involves the development of a software routine that parameterizes the DSP device of the sound card in order to perform PCM capture and recording. Such procedure should be able to record the sound data in a memory buffer and subsequently accommodate it in a storage media using a straight Raw format. Figure 3.1 shows a block diagram that illustrates the software routine.

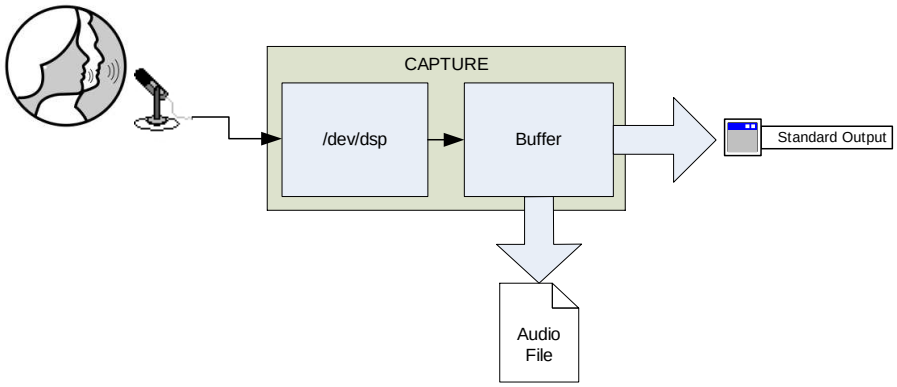

Figure 3.1: Capture process.

3.3.2 Playback

Similarly to what was previously done for the recording procedure, the playback task maybe simply implemented using a software routine that programs the DSP device to perform sound playback. In practice, it does the inverse operation by writing the raw audio samples to the sound card until the whole data has been completely transferred. The block diagram in Figure 3.2 illustrates such procedure.

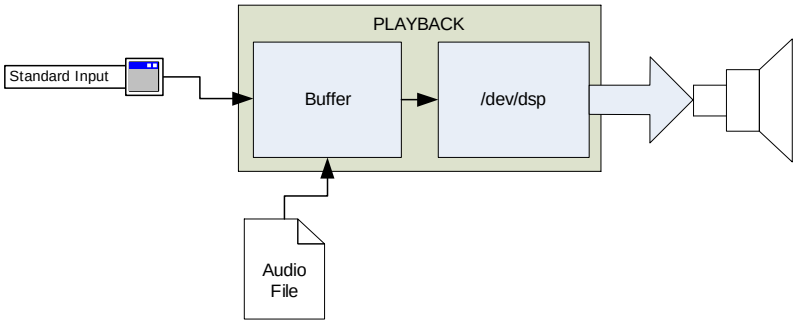

Figure 3.2: Playback process.

3.4 Audio Encoding

As it was previously mentioned, the acquired audio samples can be either stored in an uncompressed form, or may be compressed in order to reduce the amount of required storage space. This aspect may be particularly important, for data transmission.

3.4.1 Compressed Audio Encoding

Along the past decades, several audio compression algorithms have been proposed to enable audio data to be saved more efficiently. A compressor codec takes an original uncompressed audio track and exploits its intrinsic proprieties in order to reduce its size. Hence, because of the smaller size of the encoded files, the speed requirements of the storage devices may be greatly reduced, as well as transmission bandwidth. To playback the compressed audio data, a decompression algorithm is generally used.

Several audio compression formats are available, such as MP3, Ogg Vorbis, AAC, and other digital audio formats. MP3 is currently the most popular. Nevertheless, unlike MP3, Ogg Vorbis is patent and license-free. In the following, it will be given a brief description of these two formats.

MP3

MP3 is a popular digital audio encoding format, designed to greatly reduce the amount of data required to represent audio. Yet, it often provides a faithful reproduction of the original uncom-

pressed audio to most listeners. An MP3 file can be constructed either at higher or lower bit rates, with higher or lower resulting quality. The compression is achieved by reducing the accuracy of certain parts of sound that are deemed beyond the auditory resolution ability of most people. As a consequence, it is usually referred as a lossy compression format, and it is commonly referred to as perceptual encoding [22]. It internally provides a representation of sound within a short-term time/frequency analysis window, by using psychoacoustic models to discard or reduce the precision of those components that are less audible to human hearing, and encoding the remaining information in an efficient manner.

Several systems with a limited channel capacity use this format. In 1995, MP3 was selected as the audio format for the WorldSpace satellite digital audio broadcasting system [23]. It is also very usefull in several other applications. For example, as a way to store music and transfer music files over the internet.

OGG

Ogg is a multimedia container format, and the native stream format of the Xiph.org multimedia codecs. Although this container format may include other multimedia data besides audio (e.g. video), the term 'Ogg' is commonly used to refer to the audio file format Ogg Vorbis, that is, Vorbis-encoded audio in the Ogg container. Ogg is a stream oriented container, meaning that it can be written and read in one single pass, making it a natural fit for Internet streaming and implementations based on processing pipelines. This stream orientation is the major design difference over other file-based container formats [24].

3.4.2 Codecs

A **codec** is defined as a piece of software capable of encoding and/or decoding a digital data stream or signal. With such tool, the system is not only capable of recognizing the multimedia file format, but it also allows to decompress the encoded file.

For encoding using the MP3 file format, LAME software framework is currently considered one of the best encoders at mid-high bitrates and at variable bitrate (VBR). Like all MP3 encoders, LAME implements some technology that is currently covered by patents owned by the Fraunhofer Society and other entities [25]. As a consequence, some restrictions are applied to the usage of such tool for commercial applications.

Oggenc is part of vorbis-tools and it is the official Ogg Vorbis encoder by the Xiph.org Foundation. Both the LAME and the oggenc codecs are open-source.

Refer to Appendix A where the most relevant options among several available parameters within each codec are briefly described.

3.4.3 Evaluation of the Codecs Performance

To conduct the assessment and evaluation of the LAME and oggenc encoders, several audio samples with 3 seconds duration were captured and compressed. Each sample is recorded with a specific sampling rate and compressed using both codecs. In Figure 3.3 it is presented a comparison between the sizes of the output files generated by the two codecs, for several sampling rates. Table 3.1 shows the variation of the bit rate of the compressed files for several sampling rates. LAME compresses with a lower bit rate then oggenc most of the time, and with the same quality. The values are quite similar when the sampling rate increases. For 3 seconds of 8.0 kHz, 8-bit mono PCM, *capture* produces 23.4 KB of uncompressed audio data. LAME compresses it to 3.1 KB while oggenc compresses it to 10.4 KB. If the sampling rate is 44.1 kHz instead of 8.0 kHz, the size of the compressed data is now 9.5 KB and 10.9 KB for LAME and oggenc, respectively.

As a consequence, it was considered that the LAME encoder is the best solution, taking into account the objectives of this project.

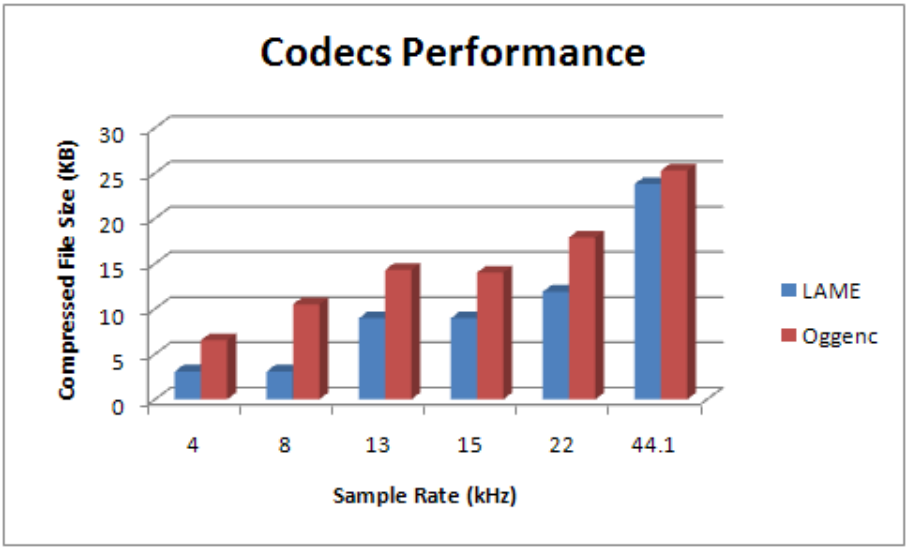

Figure 3.3: Comparison of the LAME and oggenc audio encoders for the same output quality.

		Sampling Rate (kHz)					
		4	**8**	**13**	**15**	**22**	**44.1**
Average Bit Rate (kbps)	LAME	8	8	24	24	32	64
	Oggenc	12.7	23.2	32.1	30.4	40.1	60.9

Table 3.1: Comparison of the LAME and Oggenc encoders.

3.5 Implementation of the Audio Acquisition and Encoding Module

The audio acquisition process and the audio encoding process can both be executed in parallel, each one running in a different *thread*. To transfer the acquired sound data from the acquisition (*capture*) process to the encoding process, it was decided to adapt a "named" pipe, provided by the operating system. Such approach allowed an easy integration of the developed software with the adopted open-source encoding tools that were selected to conceive the implemented system. As such, the *capture* program records audio data and writes it to a "named" pipe (NP1), while the encoder reads the data from the pipe and compresses it. Meanwhile, the compressed data can be sent to another pipe (NP2). Subsequently, the compressed data will be read from NP2 and sent to the communication channel.

In order to achieve this goal, the following tasks must be accomplished:

1. Define a static buffer to temporarily hold the sound data (unsigned samples);
2. Create the pipes with the `mkfifo()` system call;
3. Open the DSP device for reading, with the `open()` system call;
4. Set the audio parameters with the required *ioctl* system calls. The parameters to be set are the sample resolution (8-bit), the number of channels (1) and the sampling rate (configurable between 8 kHz and 44.1 kHz);
5. Open the "named" pipes with the `open()` system call. NP1 is opened for writing and NP2 is opened for reading;
6. Run the processing loop. Within the loop, execute the following tasks:
   (a) Read the sound data into the buffer with the `read()` system call;
   (b) Write the data to the pipe (NP1) with the `write()` system call.
7. Run the encoder (in another process) by using the `system()` function, to encode the data from NP1 and send the compressed data to NP2.
8. Send the data from NP2 to the base station.

It is important to mention that NP2 must be opened in non-blocking mode (O_NONBLOCK). Otherwise, the process will block until another process starts reading the data from this pipe.

Another important aspect to be referred as well is that in Linux versions before 2.6.11, the capacity of a pipe was the same as the system page size (e.g. 4096 bytes on x86). Since Linux 2.6.11, the pipe capacity is 65536 bytes [26]. Therefore, the applications must ensure that the data

transferred from the buffer is being correctly sent to the pipe. Otherwise, if the pipe capacity is lower than the buffer size, there is no way of knowing if the data is being properly sent. Moreover, the application should be designed so that the reading process consumes data as soon as it is available, so that a writing process does not have to remain blocked. The following pseudo code presents a brief description of how this problem was solved.

```
while(buffer_size)
   if(buffer_size > pipe_buf)
      write pipe_buf to NP1;

   else
      write buffer_size bytes to NP1;

   buffer_size = buffer_size - pipe_buf;
   read from NP2
   further processing...
```

4

Image Acquisition and Encoding

Contents

This chapter discusses the implementation of a computer program to acquire and encode a still image with a camera device in the Linux operating system. In this particular project, the adapted camera is the webcam (Logitech QuickCam Pro for Notebooks), previously described in section 2.6.2. The conceived program makes use of an internal kernel API designed for such purpose: the Video4Linux2 API. The aim is to implement an application that programs the capture interface (a Video4Linux2 device) in order to acquire images from the webcam. Then, two alternatives will be presented to store the image information: compressed and uncompressed. Finally, it will be described a tool to encode and compress the image data.

4.1 Interaction With the Operating System

4.1.1 Drivers (V4L,V4L2)

Video For Linux (often written as "**Video4Linux**" or abbreviated as "**V4L**" is a driver API. Essentially, the V4L API is a kernel interface for analog video capture and output drivers, and is applicable to most video streams not related to DVB devices or graphics cards (though it also entails a few oddities such as analog radio and Radio Data System (RDS) services). Several types of devices fall within the scope of V4L. Some examples are webcams, MPEG encoders/decoders, digital video streamers, analog TV tuners and video capture devices, among others.

The V4L API is currently in its second version (commonly referred to as "**V4L2**"). One of the aspects under this version of the API is that V4L2 drivers include a compatibility mode for V4L1 based applications - though the support for this can be incomplete and it is therefore recommended that devices designed for V4L2 only use this mode.

V4L2 was designed to support a wide variety of devices, although only some of which are truly "video" in nature [27]:

- The **video capture interface** grabs video data from a tuner or camera device. This interface will be further emphasized in this chapter.
- The **video output interface** allows applications to drive peripherals which can provide video images - perhaps in the form of a television signal - outside of the computer.
- The **video overlay interface**, is a variant of the capture interface, whose job is to facilitate the direct display of video data from a capture device. Video data is transferred directly from the capture device to the display, without passing through the system's CPU.
- The **VBI interfaces** provide access to data transmitted during the video blanking interval. There are two of them, the "raw" and "sliced" interfaces, which differ in the amount of processing of the VBI data performed in hardware.

- The **radio interface** provides access to audio streams from Amplitude Modulation (AM) and FM tuner devices.

4.1.2 V4L2 Devices

The **V4L2 API Specification** [28] describes the API from the user-space point of view, although the V4L2 drivers implement that API directly. Consequently, most of the structures are the same and the semantics of the V4L2 calls are clearly laid out. To use this API, the following header file must be included in the source of the application program:

```
#include <linux/videodev2.h>
```

4.1.2.A Device Types

As it was previously referred, the V4L2 driver is a suite of related specifications for different types of video devices and video-related data. The type of device determines what kind of data is passed via the *read* and the *write* system calls, and/or the set of *ioctl* commands that the driver supports. Several device types have been defined or are planned. In particular, the V4L2 devices are defined as Unix char type devices, with a major device number of 81. They are identified by the set of names shown in Table 4.1 in the */dev* tree. The V4L2 module automatically assigns minor numbers in load order, depending on the registered device type. Conventionally, V4L2 video capture devices are accessed through character device special files named */dev/video* and */dev/video0* to */dev/video63* with major number 81 and minor numbers 0 to 63. */dev/video* is typically a symbolic link to the preferred video device.

Table 4.1: V4L device types and names.

Device Name	Type of Device
/dev/video	Video capture interface
/dev/vfx	Video effects interface
/dev/codec	Video codec interface
/dev/vout	Video output interface
/dev/radio	AM/FM radio devices
/dev/vtx	Teletext interface chips
/dev/vbi	Data services interface

Figure 4.1 shows the interaction between a user application and a V4L2 driver, by using the video capture interface provided by the */dev/video* character device.

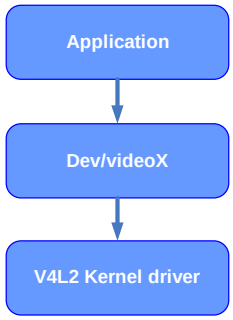

Figure 4.1: Interaction between a user application and the V4L2 driver.

4.1.2.B Device Minor Numbers

Because V4L2 is a catch-all for a wide variety of devices on different kinds of buses, there is no good way for it to automatically assign device minor numbers in a logical and consistent way. Therefore, minor numbers have to be given to V4L2 device drivers in the *insmod* command line. A single V4L2 device driver module can potentially support multiple devices of a given type, and multiple types of devices. For example, a driver may support capture, vbi, and codec device types, in which case minor numbers will need to be specified for all three types of devices. The command line parameters to set minor numbers are the same as the device node name prefixed with "unit_". For example "unit_video" for */dev/video* (video capture) devices, or "unit_codec" for codec devices.

The device nodes might be as exemplified in Table 4.2:

Table 4.2: V4L device numbers.

node	major	minor
/dev/video0	81	0
/dev/video1	81	1
/dev/codec0	81	10
/dev/codec1	81	11

V4L2 applications open and scan the existing devices to find what they are looking for. Capability queries define what each interface supports, which will be further discussed in this chapter.

4.1.3 Using the V4L2 devices: overview

Generally, the interaction between the applications and the driver follows four major processes:

- Opening and closing devices;

- Querying capabilities;
- Image format negotiation;
- Read from or write to a device - Input/Output (I/O);

In the following, it will be given a brief overview of each process.

4.1.3.A Opening and Closing Devices

To open and close V4L2 generic devices, the user application use the *open* and the *close* system calls, respectively. The *open* adopts the usual prototype:

```
int open(struct inode *inod, struct file *file)
```

The *stuct inode* (i.e. */dev/videoX*, where *videoX* is the *inode*), related to the desired device is passed as argument, so that the driver can identify the corresponding minor. Then, by using this minor number, the driver can access the proper device and retrieve its private data, which usually contains information specific to that device. The V4L2 devices can be opened more than once simultaneously, either by the same application, or by different applications. After the *open*, all the other system calls access the private data of the device via the *struct *file* pointer. The return value is returned to the application.

4.1.3.B Querying Capabilities

Because V4L2 covers a wide variety of devices, not all aspects of the API are equally applicable to all types of devices. Furthermore, devices of the same type may have different capabilities.

As a consequence, upon its opening, the VIDIOC_QUERYCAP *ioctl* should be used to query the functions and I/O methods supported by the device. All V4L2 drivers must support VIDIOC_QUERYCAP *ioctl*.

4.1.3.C Image Format Negotiation

Different I/O devices may exchange different kinds of data with the application programs, such as video images, raw or sliced VBI data, RDS datagrams, etc. The image format negotiation allows the application to ask for a particular data format, upon which the driver selects and reports the best the hardware can do to satisfy such request.

A single mechanism exists to negotiate all data formats using the aggregate `v4l2_format` structure and the VIDIOC_G_FMT *ioctl* and the VIDIOC_S_FMT *ioctl*, which must be supported by all drivers. The VIDIOC_S_FMT *ioctl* is a major turning-point in the initialization sequence. Prior to this point, multiple applications can concurrently access the same device to select the current input, change controls or modify other properties. The first VIDIOC_S_FMT assigns a

logical stream (video data, Vertical Blanking Interval (VBI) data etc.) exclusively to one file descriptor. The expression “exclusive” means that no other application (more precisely no other file descriptor) can grab this stream or change the device properties inconsistently with the negotiated parameters. Accordingly, this method prevents that multiple file descriptors (which grabbed different logical streams) interfere each other. As an example, when video overlay is about to start or is already in progress, simultaneous video capturing may be restricted to the same cropping and image size.

When the applications omit this VIDIOC_S_FMT *ioctl*, this locking effect is implicit in the following steps, corresponding to the selection of an I/O method with the VIDIOC_REQBUFS *ioctl* or implicit with the first *read* or *write* system call. Generally, only a single logical stream can be assigned to a file descriptor. The exception is observed in some drivers that permit simultaneous video capturing and overlay using the same file descriptor for compatibility with V4L and earlier versions of V4L2.

4.1.3.D Input/Output

The V4L2 API defines several different methods to read from or write to a device. All drivers exchanging data with the applications must support at least one of them. The classic I/O method based on the *read* and the *write* system calls is automatically selected after opening a V4L2 device. When the driver does not support this method, attempts to read or write will fail. As such, other methods must be negotiated. There are two streaming methods, either with memory mapped or user buffers. To select the streaming I/O method, the applications should call the VIDIOC_REQBUFS *ioctl*.

A – Read/Write Input and output devices support the classic *read* and *write* system calls, respectively, when the V4L2_CAP_READWRITE flag is set in the capabilities field of `struct v4l2_capability` returned by the VIDIOC_QUERYCAP *ioctl*.

In this mode the drivers need the CPU to copy the data to/from user space. However, this is not necessary always true, since they may also make use of DMA mechanisms to transfer data to or from user memory. Consequently, under such specific considerations this I/O method is not necessarily less efficient than other methods that merely exchange buffer pointers. Nevertheless, it is still considered inferior, since no meta-information (such as frame counters or timestamps) is passed. This information is necessary to recognize frame dropping and to synchronize with other data streams. Even so, this is also the simplest I/O method, requiring little or no setup to exchange data.

B – Streaming I/O (Memory Mapping) The input and output devices support this I/O method when the V4L2_CAP_STREAMING flag is set in the capabilities field of `struct v4l2_capability`

returned by the VIDIOC_QUERYCAP *ioctl*. In particular, to determine if the memory mapping I/O method is supported, the applications must call the VIDIOC_REQBUFS *ioctl*.

In this streaming I/O method only pointers to buffers are exchanged between the application and driver: the data itself is not copied. Memory mapping is primarily intended to map buffers in the device's memory into the application's address space. As an example, the device memory can be the frame memory on a video capture card.

To allocate device buffers, applications should call the VIDIOC_REQBUFS *ioctl* with the desired number of buffers and the corresponding buffer type (for example V4L2_BUF_TYPE_-VIDEO_CAPTURE). This *ioctl* can also be used to change the number of buffers or to free the allocated memory, provided that none of the buffers is still mapped.

Before the applications can access the buffers, they must map them into their address space with the `mmap()` function. The location of the buffers in device memory can be determined with the VIDIOC_QUERYBUF *ioctl*. The `m.offset` and `length` values returned in the `v4l2_buffer` structure are then passed to the `mmap()` function. Since these buffers are allocated in physical memory (as opposed to virtual memory which could be swapped out to disk), applications should not change the offset and length values. Moreover, as soon as the buffers are no longer required, they should be released by the applications with the `munmap()` function.

C – Streaming I/O (User Pointers) The input and output devices support this I/O method when the V4L2_CAP_STREAMING flag is set in the capabilities field of `struct v4l2_capability` returned by the VIDIOC_QUERYCAP *ioctl*. In particular, to determine if the user pointer method is supported, applications should call the VIDIOC_REQBUFS *ioctl*.

This I/O method combines advantages of the read/write and memory mapping methods. Buffers are allocated by the application itself and can reside either in virtual or shared memory. Only pointers to data are exchanged, where these pointers and the corresponding meta-information are passed in `struct v4l2_buffer`. The driver is switched into user pointer I/O mode by calling the VIDIOC_REQBUFS with the desired buffer type. No buffers are allocated beforehand. Consequently, they are not indexed and cannot be queried (as opposed with the mapped buffers with the VIDIOC_QUERYBUF *ioctl*).

Buffer addresses and sizes are passed on the fly with the VIDIOC_QBUF *ioctl*. Since buffers are commonly cycled along the time, applications can pass different addresses and sizes at each VIDIOC_QBUF call. If required by the hardware, the driver may swap memory pages within physical memory to create a continuous memory space. This happens transparently to the application in the virtual memory subsystem of the kernel. In the event that buffer pages have been swapped out to disk, they are brought back and finally locked in physical memory for DMA. The driver can unlock the memory pages at any time between the completion of the DMA and this *ioctl*. The memory is also unlocked when VIDIOC_STREAMOFF or, VIDIOC_REQBUFS are called, or

when the device is closed. Before applications can release the buffer's memory, they must ensure that the buffers are not being used.

4.1.3.E Buffer Queuing System

The management of the several buffers used to exchange data between the driver and the user space memory is implemented by means of a queuing system. For capturing applications, the user program enqueues a number of empty buffers, to start capturing and enter the read loop. Then, the application waits until a buffer has been filled by the driver and dequeued. Finally, it re-enqueues the buffer when the data is no longer needed. In contrast, output applications fill and enqueue buffers. As soon as enough buffers are stacked up, the driver initiates the output process. In the write loop, when the application runs out of free buffers it must wait until an empty buffer can be dequeued and reused.

There are three fundamental states that a buffer can be in [27]:

- *In the driver's incoming queue* - The buffers are placed in this queue by the application in the expectation that the driver will do something useful with them. For a video capture device, buffers in the incoming queue will be empty, waiting for the driver to fill them with video data. For an output device, these buffers will have frame data to be sent to the device;
- *In the driver's outgoing queue* - Buffers in this state were already processed by the driver and are waiting for the application to claim them. For capture devices, outgoing buffers will have new frame data, and for output devices, these buffers are empty;
- *In neither queue* - In this state, the buffer is owned by the user program and will not normally be touched by the driver. This is the only time when the application should do anything with the buffer. One shall call this state the "user space" state.

These states, as well as the operations which cause transitions between them, are illustrated in the diagram of Figure 4.2:

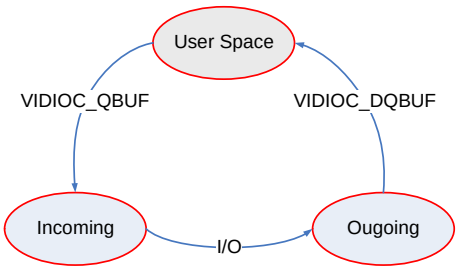

Figure 4.2: Buffer fundamental states.

To enqueue and dequeue a buffer applications use the VIDIOC_QBUF and VIDIOC_DQBUF

ioctl. The status of a buffer being mapped, enqueued, full or empty can be determined at any time using the VIDIOC_QUERYBUF *ioctl*.

The applications should suspend the execution until one or more buffers can be dequeued. By default VIDIOC_DQBUF blocks when no buffer is in the outgoing queue. When the O_-NONBLOCK flag was used as attribute in the *open* system call, VIDIOC_DQBUF will return immediately with an EAGAIN error code if no buffer is available. The `select()` or `poll()` functions are always available.

To start and stop capturing or output, the applications should call the VIDIOC_STREAMON and VIDIOC_STREAMOFF *ioctl*. The VIDIOC_STREAMOFF removes all buffers from both queues as a side effect. As such, if an application needs to synchronize with another event, it should examine the `struct v4l2_buffer` *timestamp* of the captured buffers before enqueuing buffers then output.

A – The v4l2_buffer structure Pixel data is passed between the application and the driver in a data structure of type v4l2_buffer, whose description is the following:

```
struct v4l2_buffer
    {
        __u32                   index;
        enum v4l2_buf_type      type;
        __u32                   bytesused;
        __u32                   flags;
        enum v4l2_field         field;
        struct timeval          timestamp;
        struct v4l2_timecode    timecode;
        __u32                   sequence;

        /* memory location */
        enum v4l2_memory        memory;
        union {
                __u32           offset;
                unsigned long   userptr;
        } m;
        __u32                   length;
        __u32                   input;
        __u32                   reserved;
    };
```

The `index` field is a sequence number identifying the buffer. It is only used with memory-mapped buffers. The buffers start with index 0 and increase its value sequentially. The `type` field describes the type of the buffer, usually V4L2_BUF_TYPE_VIDEO_CAPTURE or V4L2_BUF_-TYPE_VIDEO_OUTPUT.

The size of the buffer is given by `length` (in bytes). The size of the image data contained within the buffer is found in `bytesused` (`bytesused` $\leq$ `length`). For capture devices, the driver will set `bytesused`. For output devices the application must set this entry. `field` describes which field of the image is stored in the buffer. The `timestamp` entry, for input devices, indicates when the frame

was captured. For output devices, the driver should not send the frame out before the time found in this field. A `timestamp` of zero means "as soon as possible". The driver will set `timestamp` with the time that the first byte of the frame was transferred to the device - or as close to that time as it can get. `timecode` can be used to hold a timecode value, useful for video editing applications [28]. The driver also maintains an incrementing count of frames passing through the device. It stores the current `sequence` number in sequence as each frame is transferred. For input devices, the application can check this field to detect dropped frames. `memory` tells whether the buffer is memory-mapped or user-space. For memory-mapped buffers, `m.offset` describes where the buffer is to be found. For user-space buffers, `m.userptr` is the user-space address of the buffer. The `input` field can be used to quickly switch between inputs on a capture device - assuming the device supports quick switching between frames. The `reserved` field should be set to zero.

Finally, there are several `flags` defined:

- V4L2_BUF_FLAG_MAPPED: indicates that the buffer has been mapped into user space - only applicable to memory-mapped buffers;
- V4L2_BUF_FLAG_QUEUED: the buffer is in the driver's incoming queue;
- V4L2_BUF_FLAG_DONE: the buffer is in the driver's outgoing queue;
- V4L2_BUF_FLAG_KEYFRAME: the buffer holds a key frame - useful in compressed streams;
- V4L2_BUF_FLAG_PFRAME and V4L2_BUF_FLAG_BFRAME are also used with compressed streams - indicate predicted or difference frames;
- V4L2_BUF_FLAG_TIMECODE: the `timecode` field is valid;
- V4L2_BUF_FLAG_INPUT: the `input` field is valid.

B – Buffer setup Once a streaming application has performed its basic setup, it will turn to the task of organizing its I/O buffers. The first step is to establish a set of buffers with the VIDIOC_REQBUFS *ioctl*, which is turned by V4L2 into a call to the driver's `vidioc_reqbufs()` method:

```
int (*vidioc_reqbufs) (struct file *file, void *private_data,
                       struct v4l2_requestbuffers *req);
```

The `v4l2_requestbuffers` structure follows the format:

```
struct v4l2_requestbuffers
    {
        __u32                   count;
        enum v4l2_buf_type      type;
        enum v4l2_memory        memory;
        __u32                   reserved[2];
    };
```

The `type` field describes the type of I/O to be done. It will usually be either V4L2_BUF_-TYPE_VIDEO_CAPTURE for a video acquisition device or V4L2_BUF_TYPE_VIDEO_OUTPUT for an output device, among others. If the application wants to use memory-mapped buffers, it will set `memory` to V4L2_MEMORY_MMAP and `count` to the number of buffers it wants to use. If the driver does not support memory-mapped buffers, it will return the error code `EINVAL`. Otherwise, it will allocate the requested buffers internally and return zero.

To work properly, the driver may require a minimum number of buffers that must be enqueued at any time. Apart of this, no limit exists on the number of buffers that the applications can enqueue in advance, or dequeue and process [28]. It is important to note that the driver is not required to allocate exactly the requested number of buffers. In many cases, there is a minimum number of buffers which makes sense. If the application requests fewer than the minimum, it may actually get more buffers than it asked for.

Setting `count` to zero is a way for the application to request that all existing buffers be released. In this case, the driver must stop any DMA operations before releasing the buffers. It is not possible to free buffers if they are current mapped into user space.

In contrast, if user-space buffers are to be used, the only fields which matter are the buffer type and a value of V4L2_MEMORY_USERPTR in the `memory` field. The application does not need to specify the number of buffers that it intends to use. Since the allocation will be happening in user space, the driver needs not to care about it to. Provided that the driver supports user-space buffers, it will return zero; otherwise the error code `EINVAL` will be returned.

4.2 Video Capture Interface

Besides the set of programming aspects concerning to the V4L2 device driver described in the previous section, other aspects concerning to the video capture interface are particularly relevant for this project. Therefore, the common API elements and the basic concepts particularly applied to the **video capture interface** will be now presented. With this interface, applications can control the capturing process and move images from the driver into user space.

4.2.1 Driver Parameterization

4.2.1.A Querying Capabilities

Devices supporting the video capture interface set the `V4L2_CAP_VIDEO_CAPTURE` flag in the `capabilities` field of `struct v4l2_capability`, returned by the `VIDIOC_QUERYCAP` *ioctl*. In order to read images, the device must support at least one of the read/write or streaming I/O methods. In this particular case of the Logitech QuickCam Pro webcam device, which will be used in this project, only memory mapping is supported.

Image Format Negotiation

The result of a capture operation is determined by cropping and the image format parameters. The former selects an area of the acquired frame, while the latter defines how images are stored in memory (i. e. Red, Green, Blue (RGB) or YUV formats), the number of bits per pixel, the image width and height. Together they also define how images are scaled in the process.

To query the current image format definition, applications should set the `type` field of a `struct v4l2_format` to `V4L2_BUF_TYPE_VIDEO_CAPTURE` and call the `VIDIOC_G_FMT` *ioctl* with a pointer to this structure. The driver will then fill `pix` member of the `fmt` union enclosed within `struct v4l2_format`. In order to request different parameters, applications should set the `type` field of `struct v4l2_format` as above and initialize all fields of the `vbi` member of `fmt` union, enclosed within `struct v4l2_pix_format`. An alternative and more expedite solution consists in just modifying the results of VIDIOC_G_FMT, and call the VIDIOC_S_FMT *ioctl* with a pointer to this structure. The driver will then adjust the specified parameters and returns then just as VIDIOC_G_FMT does.

4.2.2 YUV Formats

YUV is the native format of TV broadcast and composite video signals. It separates the brightness information (Y) from the color information (U/Cb and V/Cr). The color information consists of blue and red *color difference* signals. The green component is reconstructed by subtracting U and V from the brightness component.

The adoption of the YUV format comes from the fact that early television only transmitted brightness information. Hence, to add color in a way compatible with existing receivers, a new signal carrier was added to transmit the color difference signals. One important property of this format is the fact that the YUV format the U and V components usually have lower resolution than the Y component, taking advantage of a property of the human visual system, which is more sensitive to brightness information than to color. This technique, known as **chroma subsampling**, has been widely used to reduce the data rate of video signals.

There are several subsampling schemes for the YUV format but only three are considered relevant for this project - 4:4:4, 4:2:2 and 4:2:0 (see Figure 4.3). In the following, it will be given a brief overview about each of the schemes.

YCbCr is commonly denoted YCbCr in digital domain. As such, from this point of this document one shall use the YCbCr notation.

4.2.2.A YCbCr 4:4:4

Each of the three YCbCr components have the same sample rate. This scheme is sometimes used in high-end film scanners and cinematic postproduction. Figure 4.4 shows an example of an

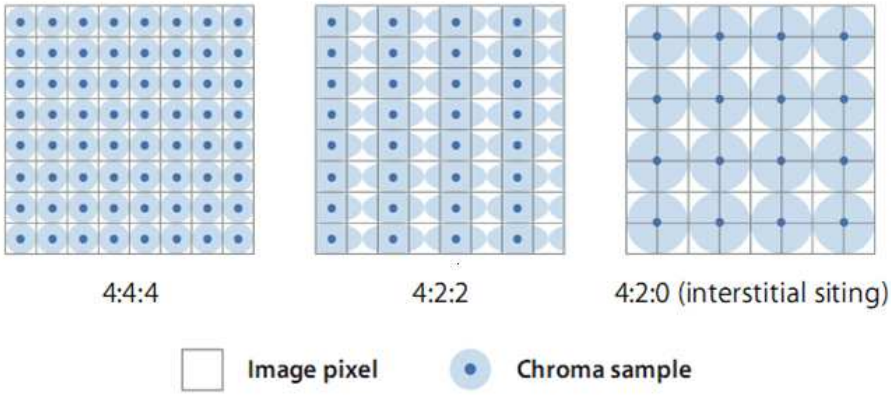

Figure 4.3: Types of subsampling.

image along with its Y, Cb, and Cr components.

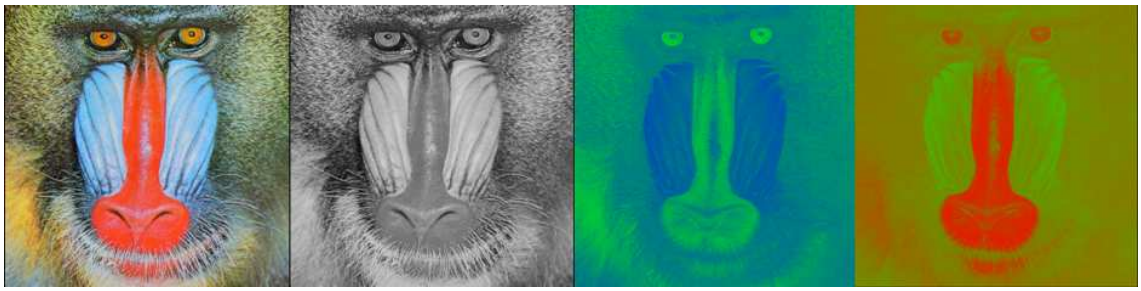

Figure 4.4: YCbCr components.

YCbCr 4:2:2

The two chroma components are sampled at half the rate of luma in the horizontal direction. This reduces the bandwidth of the video signal by one-third with little to no visual difference.

Many high-end digital video formats and interfaces make use of this scheme (e.g.: AVC-Intra 100, Digital Betacam, DVCPRO50 and DVCPRO HD, Digital-S, etc.).

4.2.2.B YCbCr 4:2:0

This scheme is used in JPEG/JPEG File Interchange Format (JFIF) encodes of still frames and in most video compression standards, such as H.26x/MPEG-x, in H.261. Cb and Cr components are each subsampled by a factor of 2 in the horizontal and vertical directions.

There are several variants of 4:2:0 subsampling.

- In MPEG-2, Cb and Cr are cosited horizontally, i.e. Cb and Cr are sited between pixels in the vertical direction (sited interstitially).

- In JPEG/JFIF, H.261, and MPEG-1, Cb and Cr are sited interstitially, i.e. halfway between alternate luma samples.

Despite the wide set of digital image formats available, the webcam device that was considered for this project (Logitech Quickcam Pro) only supports the YCbCr 4:2:2 format, which is identified by the V4L2 driver with V4L2_PIX_FMT_YUYV identifier. Refer to Appendix B.1, where this format is briefly described.

4.3 Image Encoding

The acquired images can be either stored in an uncompressed form, or may be compressed in order to reduce the amount of storage space, as well as the transmission bandwidth.

4.3.1 Uncompressed Image Encoding

As it was previously referred, among all the available sample formats, the only format supported by the webcam used in this project is the YCbCr 4:2:2. However, most image formats (e.g.: Portable Pixel Map (PPM), etc.) adopt the RGB color space in their encoding, which requires the implementation of conversion procedure between the YCbCr 4:2:2 and the RGB sample formats. In the following, it will be given a brief overview of both of these sample formats.

4.3.1.A RGB Color Space

A given color in the RGB color space is described by indicating how much the red, green, and blue primary components is included. Each color is expressed as an RGB triplet (r,g,b), where each component can vary from zero to a defined maximum value. If all the components are at zero, the result is black; if all are at their maximum value, the result is the brightest representable white.

These ranges may be quantified in several different ways:

- From 0 to 1, with any fractional value in between, used in theoretical analyses, and in systems that use floating-point representations;
- As a percentage, to represent each color component from 0% to 100%;
- In computing, the component values are often stored as integer numbers, in the range between 0 and 255, often adopted in computing by using a single 8-bit byte to represent each color component with 256 distinct values.
- High-end digital image equipment can deal with the integer range 0 and 65535 for each color component, by employing 16-bit words instead of 8-bit bytes.

To exemplify these notations, the full intensity red is represented using several different RGB representations in Table 4.3.

Table 4.3: Example of RGB notations.

Notation	RGB triplet
Arithmetic	(1.0, 0.0, 0.0)
Percentage	(100%, 0%, 0%)
Digital 8-bit per channel	(255, 0, 0)
Digital 16-bit per channel	(65535, 0, 0)

A – YCbCr to RGB Conversion The conversion between the YCbCr color space and the RGB color space is conducted by a simple arithmetic combination to the acquired YCbCr 4:4:4 sample pixels. The following formulas show how to obtain the pixel value in one format from the pixel value in the other format (more precisely YCbCr 4:4:4 to RGB 4:4:4) [29]:

$$
\begin{aligned}
R &= \text{clip}((\ 298 * C + 409 * E + 128) >> 8) \\
G &= \text{clip}((\ 298 * C - 100 * D - 208 * E + 128) >> 8) \\
B &= \text{clip}((\ 298 * C + 516 * D + 128) >> 8)
\end{aligned}
$$

Where the following coefficients are used in this conversion process:

$$
\begin{aligned}
C &= Y - 16 \\
D &= U - 128 \\
E &= V - 128
\end{aligned}
$$

`clip()` denotes clipping a value to the range of 0 to 255.

The programmer must ensure that the pixel values for each component are defined within the ranges described in Table 4.4, before using the previous formulas. It is worth noting that there are other formulas used to convert YCbCr to RGB. However, they are less efficient, since they use more multiplications, which were replaced by simpler bit-shifts and addition operations.

Table 4.4: Values of the components.

Component	Min Value	Max Value
Y	0	255
Cb	-128	127
Cr	-128	127
R	0	255
G	0	255
B	0	255

B – YCbCr 4:2:2 to YCbCr 4:4:4 Conversion As referred before, the adopted webcam does not support the acquisition of pixel data in the YCbCr 4:4:4 sample format. As a consequence, in order to convert the acquired YCbCr 4:2:2 format to RGB, first it is necessary to convert the YCbCr data to YCbCr 4:4:4, and then convert from YCbCr 4:4:4 to RGB.

Converting YCbCr 4:2:2 to YCbCr 4:4:4 means doubling the horizontal resolution of the Cr and Cb components, so that they have the same horizontal resolution as the Y component. This is simply achieved by duplicating each cell containing the chroma components, as shown in Figure 4.5.

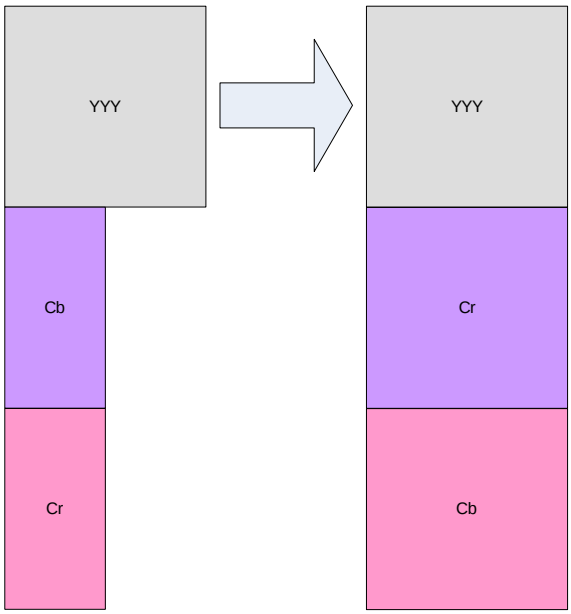

Figure 4.5: Conversion between YCbCr 4:2:2 to YCbCr 4:4:4 color spaces.

4.3.1.B PPM File Format

PPM is a simple and commonly used uncompressed RGB color image representation. Its definition is as follows:

- A "magic number" for identifying the file type - the PPM file's magic number is the two characters "P3";
- Whitespace;
- The image width, formatted as American Standard Code for Information Interchange (ASCII) characters in decimal;
- Whitespace;
- The image height, again in ASCII decimal;

- Whitespace;
- The maximum color-component value, again in ASCII decimal;
- Whitespace;
- Width * height pixel data - the RGB components of each pixel value are represented with three ASCII decimal values, between 0 and the specified maximum value, starting at the top-left corner of the pixmap, and proceeding in normal text reading order. The value of 0 represents black, and the maximum value represents the white.

Characters inserted between a "#" and the next end-of-line are ignored (comments). However, no line should be longer than 70 characters. An example of a color RGB image represented using the PPM format is presented in Table 4.5 and the corresponding image is shown in Figure 4.6. There is a newline character at the end of each line[30].

Table 4.5: PPM format example.

```
P3
# P3 means colors are in ASCII, 3 columns and 2 rows
# then 255 for max color, then RGB triplets
3 2
255
255   0     0     0   255     0     0     0   255
255   255   0   255   255   255     0     0     0
```

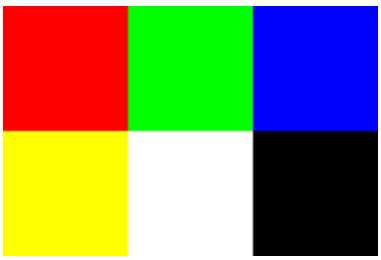

Figure 4.6: PPM example.

The great disadvantage of this image format is the considerable size required to accommodate it in the file system. As an example, a PPM image with a resolution of 640 x 480 pixels has a file size of 900 KB. When encoded into the JPEG standard image (described below), the file size can be reduced to 24.4 KB. For this reason, the PPM format is generally applied as an intermediate format used to store and pre-process the image before converting to a more efficient format.

4.3.2 Compressed Image Encoding

Several image compression formats are currently available, some of them represent the original image without introducing any degradation, such as the GIF, the Portable Network Graphics (PNG), and the Tagged Image File Format (TIFF). However, their lossless nature compromises the achieved compression level. As a consequence, some other lossy compression image formats have been adopted, such as the JPEG, providing a much greater encoding efficiency.

As such, JPEG is designed for compressing either full-color or gray-scale images of natural real-world scenes. This compression algorithm is "lossy," meaning that the decompressed image is not quite the same as the original. The offered compression is achieved by exploiting known limitations of the human visual system, notably the fact that small color changes are perceived less accurately than small changes in brightness. Thus, JPEG is more intended for compressing images that will be looked at by humans.

A useful property of JPEG is that the degree of lossiness can be varied by adjusting some compression parameters. This means that the image maker can trade off the file size against the output image quality. As such, it is possible to represent images with “extremely” small files if the poor output quality is not a problem. This is useful for certain applications, such as indexing image archives. Conversely, if the quality is a main concern, and the output quality at the default compression setting is not enough, it is possible to increase the quality until it is satisfactory. This way there will be lesser compression.

Another interesting and important aspect of JPEG is that decoders can trade off the decoding speed against image quality, by using faster but inaccurate approximations to the required calculations.

4.3.2.A JPEG Codec

For encoding using the JPEG format, the **cjpeg** encoder is a great solution. It is part of a coder/decoder package (cjpeg and djpeg respectively). Cjpeg compresses the image file and produces a JPEG/JFIF file on the standard output.. The currently supported input file formats are: PPM, Portable Gray Map (PGM), GIF, Truvision Advanced Raster Graphics Array (Targa), and RLE. The image quality is adjusted by setting the *-quality* option. Refer to appendix B.2, where this option is briefly described and an example is presented.

4.3.2.B Evaluation of the Codec Performance

The acquired YCbCr images must be converted to PPM, in order to be compressed by the codec. This is achieved by first converting the YCbCr image to RGB color space and then by accommodating the RGB pixel data in a PPM file.

Table 4.6 shows the resulting file size, after applying the cjpeg encoder to execute a lossless

uncompressed PPM image file information, considering the amount of image data being acquired with the *-quality* option set to default (75).

By comparing between compressed and uncompressed data, it is possible to verify that the results are better than the initially expected before testing (the 10:1 ratio previously described in Section 2.7).

RESOLUTION	DATA SIZE (KB)	
	UNCOMPRESSED (PPM)	COMPRESSED (JPEG)
QCIF (176 x 144 pixels)	74.277	3.16
CIF (352 x 288 pixels)	297.027	6.885
VGA (640 x 480)	900.027	26.256

Table 4.6: Bandwidth of image signals.

If it is intended to get even higher compression, this is easily achieved simple by adjusting the codec's parameter (reducing the image quality). Nevertheless, the present results are acceptable.

4.4 Implementation of the Image Acquisition and Encoding Module

The image acquisition and encoding module was implemented by a software routine divided in two functional blocks. The first is responsible for the configuration of the video capture device and related negotiations with the V4L driver, required after opening and closing the device: querying capabilities, image format negotiation, etc. The second block is responsible to respectively acquire and encode the captured frames. To properly control this cycle, so that images are acquired in a certain and periodic interval, a software timer and the corresponding signal handler routine was implemented. This periodic interval is denoted the image refresh rate.

In order to achieve this goal, the following tasks must be accomplished:

1. Setup both the interval-timer and the signal handler routine with the `setitimer()` and the `signal()` system calls, respectively;
2. Open the video capture device (*/dev/video0*) with the `open()` system call;
3. Initialize the video capture device:
   (a) Query the device capabilities, with the VIDIOC_QUERYCAP *ioctl*;
   (b) Negotiate the image format, with the VIDIOC_S_FMT *ioctl*;
4. Initialize the memory-mapping scheme:

(a) Request the driver to accommodate the required frames in memory mapping mode, with the VIDIOC_REQBUFS *ioctl*;

(b) Setup the buffers characteristics, with the VIDIOC_QUERYBUF *ioctl*;

5. Run the acquisition loop, by executing the following tasks:

(a) Image capture:

i. Request the driver to enqueue the frame buffer with the VIDIOC_QBUF *ioctl*;

ii. Start the acquisition chain with the VIDIOC_STREAMON *ioctl*;

iii. Wait and dequeue the acquired frame, with the VIDIOC_DQBUF *ioctl*;

iv. As soon as the frame is processed, call the VIDIOC_QBUF *ioctl* to re-enqueue the buffer again;

(b) Image encoding:

i. Convert the acquired YCbCr 4:2:2 image into the YCbCr 4:4:4 pixel format;

ii. Convert the YCbCr 4:4:4 image into RGB color space;

iii. Add the PPM header to the RGB image data;

iv. Run the cjpeg encoder with the `system()` system call, in order to encode the PPM image data into a JPEG image;

(c) Send the encoded data to the base station;

(d) Wait until the next interruption of the timer.

Refer to Appendix B.3, where the pseudo code for the YCbCr 4:2:2 to YCbCr 4:4:4 conversion is presented, and refer to the Appendix B.4, were the YCbCr 4:4:4 to RGB conversion is presented. The PPM header to be concatenated to a 640 x 480 RGB image data follows the format:

```
P6
# PPM header: Image resolution is 640 x 480 pixels
640 480
255
```

5

Video Acquisition and Encoding

Contents

This chapter discusses the main concepts about video programming in Linux. In the following sections it will be provided a brief description about the main operations that had to be considered: video acquisition, and encoding.

5.1 Programming Video Acquisition in Linux

Video can be defined as an ordered sequence of still images. As such, the involved procedures included the implementation of a software application that programs and interacts with the video capture interface provided by the V4L2 driver. A detailed presentation of the interface provided by this driver was already presented and discussed in the previous chapter. In order to acquire images from the webcam to capture video. The involved tasks and configurations are exactly the same: opening and closing the device, querying capabilities of the device, image format negotiation and read the frames. The only difference refers to the implementation of the acquisition loop, where the adopted timer has to be properly setup in order to achieve a signaling note corresponding to the desired frame rate of the acquired video sequence.

As previously mentioned in Section 4.2, the frames that are acquired with the adopted webcam must be output in raw YUV format (more precisely YCbCr 4:2:2), due to the limitations of the camera. Nevertheless, these frames shall be used either to playback in the base station (laptop) or to be encoded and compressed. However, most video players and encoders only support the YCbCr 4:2:0 format in their most basic parameterizations. Therefore, the conceived application must start by converting the acquired frames, from the YCbCr 4:2:2 to the YCbCr 4:2:0 format.

YCbCr 4:2:2 to YCbCr 4:2:0 format Conversion

In order to convert the 4:2:2 image format (corresponding to the V4L2 format V4L2_PIX_FMT_YUYV identifier, described in Appendix B.1) to the 4:2:0 format, the new components Cr' and Cb' must be re-sampled in order to have half the horizontal resolution that the Y component. For example, Cr'00 is the mean value between Cr00 and Cr10; Cr'01 is the mean value between Cr01 and Cr11 and so on, as illustrated in Figure 5.1.

5.2 Video Encoding

In Section 2.7 it was given a brief overview on the process of encoding and compressing the acquired frames, in order to reduce the amount of storage space, as well as to reduce transmission bandwidth. In the following, it will be given a brief description about the selected video compression standard, the MPEG-4 Part 2 video encoder.

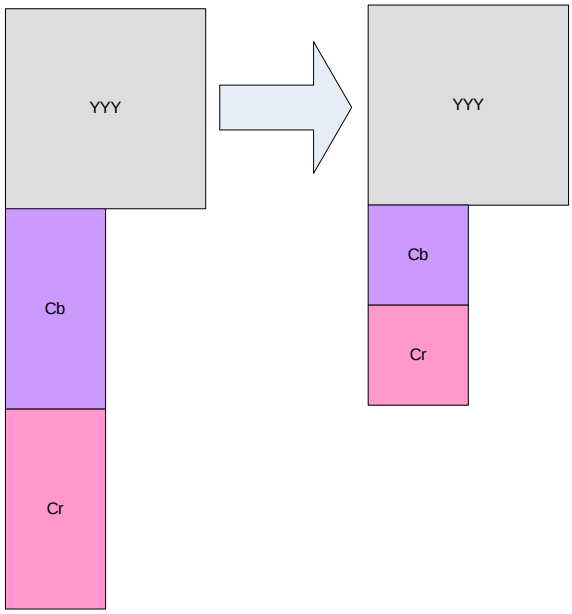

Figure 5.1: 4:2:2 to 4:2:0 format conversion.

5.2.1 Compressed Video Encoding

MPEG-4 Part 2 uses motion compensation followed by DCT, quantization and entropy coding. Just as other video standards. it is also concerned with the offered set of capabilities organized in *profiles* (sets of algorithmic features) and *levels* (performance classes).

5.2.1.A Profiles and Levels

There are at least 19 profiles defined in MPEG-4[31]. The simplest being the *Simple* profile, for low-complexity coding of rectangular video frames (which is the profile that was focused in this project) to the most complex, the *Core Studio* for object-based coding of high quality video. Each profile can have 4 or more levels, which are used to place constraints on paremeters of the bit stream such as picture size, bit rate, number of objects and frame rates.

5.2.1.B Transform and Quantization

After the application of the prediction techniques, each residual frame (sample) is transformed using 8x8 pixel block DCT. The transform converts the samples into another domain in which they are represented by transform coefficients. The coefficients are quantized to remove insignificant values, leaving a small number of significant coefficients that provide more compact representation of the residual frame.

Quantization is the main process in which large compression is achieved. This is achieved by dividing each coefficient by an integer value. This process reduces the precision of the transform

coefficients according to a quantization parameter (QP). There are 31 values defined for this parameter. Bit-rate vs quality tradeoff is achieved y varying the QP. Setting QP to a high value means that more coefficients are set to zero, resulting in higher compression at the expense of poor decoded image quality. Setting QP to a low value means that more non-zero coefficients remain after the quantization, resulting in better decoded image quality but lower compression. The simple profile uses scalar quantization, where all coefficients are scaled using the same quantization parameter [32]. Another profile may allow a user defined matrix quantizer, where each coefficient can have a unique scalar.

5.2.2 Video Encoding Software (codec)

For encoding using the MPEG-4 Part 2 file format, FFmpeg software framework is very popular. It incorporates several codecs, some developed by teams that are completely independent from each other. FFmpeg is a very fast video and audio converter. It is a free software with a command line interface that is designed to be intuitive. Moreover, it is open source and is always being developed and improved. FFmpeg is developed under GNU/Linux, but it can be compiled under most operating systems [33]. Refer to Appendix C.2 to learn more about the parameterization. To encode MPEG-4 Part 2 the command line should be the following:

```
ffmpeg -f rawvideo -r framerate -s widthxheight -pix_fmt yuv420p -i IN_FILE
-vcodec mpeg4 -r framerate -qscale q -y OUT_FILE
```

The codec used by the ffmpeg framework to encode MPEG-4 Part 2 is implemented through the mpeg4 library, defined in the *-vcodec* parameter.

The *-qscale* (or quantization scale) represents the quantization parameter described in Section 5.2.1.B. The lower the *qscale* value, the better the quality. The available *qscale* values range from 1 (highest quality) to 31 (lowest quality).

5.2.3 Evaluation of the Video Codec Performance

```
ffmpeg -f rawvideo -r framerate -s widthxheight -pix_fmt yuv420p -i IN_PIPE
-vcodec mpeg4 -r framerate -qscale q -y OUT_PIPE
```

The applications exchange video data by using “named” pipes (as it was done with the audio data in Section 3.5). IN_PIPE is the pipe where the captured frames are written, in YCbCr format. Then ffmpeg encodes the video data to MPEG-4 and writes it back to OUT_PIPE.

Table 5.1 shows the resulting file size and data rate, after applying the ffmpeg encoder to execute two lossless uncompressed YCbCr 4:2:0 files: one with QCIF resolution, whose size is 14.5 MB, and the other with CIF resolution, whose size is 43.5 MB.

By comparing the sizes between both the compressed and the uncompressed video data, it is possible to verify that the results for compressed data using a quantizer scale value of 10 (or

greater) are closer to the results that were initially expected (the 100:1 ratio previously described in Section 2.7). Moreover, values greater than 13 produces low quality video. Nevertheless, this is still acceptable considering the compromise between bandwidth and video quality. However, if necessary it is possible to set this value to a satisfactory value or, instead of assigning a constant value, one can set a upper limit by setting the *-qmax* parameter to a desired value; 10 is a good value.

RESOLUTION	Quantizer Scale	Compressed Size (KB)	Data Rate (kbps)
QCIF	5	598	367.6
	10	246	150.9
	15	148	91.2
	20	107	66.0
CIF	5	1395	1143.0
	10	578	473.1
	15	355	291.2
	20	263	215.3

Table 5.1: Bandwidth of video signals.

5.3 Implementation of the Video Acquisition and Encoding Module

The implementation of the video acquisition and encoding involves several tasks that were already descried in the previous modules in chapters 3 and 4. The acquisition process and the encoding process shall run each in a different thread. Therefore, both processes may be running in parallel and shall communicate with each other through a “named” pipe. As such, the acquisition module captures the video data and writes it to a pipe (IN_PIPE). Meanwile, the encoder compresses the YCbCr video data from IN_PIPE and writes it to another pipe (OUT_PIPE). Therefore, in order to achieve this goal, the following tasks must be accomplished:

1. Set both the interval-timer and the signal handler with the `setitimer()` and the `signal()` system calls, respectively;
2. Create the pipes with the `mkfifo()` function;
3. Open the video capture device (*/dev/video0*) with the *open* system call;
4. Initialize device and memory-mapping (same as in Section 4.4);
5. Run in a loop. Within the loop execute the following tasks:
   (a) Start capturing (same as in Section 4.4);

(b) Encode the image data:

i. Convert the YCbCr 4:2:2 frame to YCbCr 4:2:0;

ii. Run the encoder with the `system()` function in order to encode the YCbCr video data into MPEG-4;

(c) Wait until the next interruption;

Refer to Appendix C.3, where the pseudo code for the YCbCr 4:2:2 to YCbCr 4:2:0 conversion is presented.

6

System Integration and Implementation

Contents

In this chapter it is presented the integration of the several modules that were developed and described in the previous chapters. The conceived system acquires all signals in parallel, encode and compress the sampled data and sends them to the other node of the communication link - this shall be the *server* of the system, which is installed in the mobile station (car). On the other hand, there is also a *client* which receives the data in the base station (pit), stores it in a media device and playback to the user. The communication is achieved through a TCP socket supported in a wireless link, as will be described further in Section 6.3. Finally, the conceived user interface will be described.

6.1 Developed Modules

The modules that were developed in the racing car node are the following: audio decoding and playback module, audio acquisition and encoding module, image acquisition and encoding module, video acquisition and encoding module, controller - receives the parameters (from the base station) that control the audio playback and the audio and image/video acquisition.

The modules that were developed in the pit wall node are the following: image decoding and playback module, video decoding and playback module, image/video multiplexer module - this module selects which data to playback, between image and video (this procedure is useful, since the image and video shall be shown in the same display (see Figure 6.2) - audio decoding and playback module, audio acquisition and encoding module, parameter and control encoding module - reads the parameters from the user.

Moreover, the message format and protocol module was also developed in both nodes. This module will be further discussed in Section 6.4.1.

6.2 Modules Integration

6.2.1 Racing Car Node

The flow chart presented in Figure C.1 in Appendix C.4 illustrates the general operation of the server process. The set of tasks implemented in this procedure can be summarize as follows:

- Create a socket to establish the communication link with the client;
- Start the execution loop - each iteration comprises:
  - The server either receives the parameters to set up the devices and start capturing (if it is the first iteration), or receives the options from the client (pit) to control the acquisition of the signals. The parameters are: audio sampling rate, audio sample resolution, frame resolution, frame rate, image resolution, image refresh rate;

  - After acquiring and encoding the data, it is sent to the client trough the TCP socket. The integration of the acquisition and encoding module was previously described in sections 3.5, 4.4 and 5.3.

6.2.2 Pit Wall Node

The flow chart in Figure C.2 in Appendix C.4 depicts the operation of the client process. Its procedure can be summarize as follows:

- The program is initially blocked, waiting for the user to insert the required options which set up the parameters described in Figure 2.4;
- Creation of the socket to communicate with the server. The program quits in case of failure, otherwise it sends the user options to the server.
- Start the operation loop. Within each iteration, it executes the following tasks:
  - Check if the user is changing the input parameters that control the acquisition process in the server side;
  - If a new parameter value is inserted corresponding to a new configuration, the control message is sent to the server which may send encoded data;
  - After the data is received, the storage and playback tasks are executed;
  - The user may stop the loop by exiting the program, which actually exits both the client and the server processes.

In order to playback the received data (audio, image or video), it shall be written to a corresponding “named” pipe. Furthermore, *mplayer* (an external application) is used to read the data from these pipes. This application allows the user to directly visualize and/or listen to the received stream, since the decoding process is executed internally.

Refer to Appendix C.1, where the most relevant options among several available parameters are briefly described.

6.3 Communication Structure

After the audio and image/video data are acquired and compressed at the server side, the signals are ready to be transmitted to the pit. The communications between the car and the pit are established through a Wi-Fi connection, supported by an USB adapter.

6.3.1 Hardware

Figure 6.1 illustrates the selected hardware device, the SMC EZ Connect G (SMCWUSB-G) [34], which is a 2.4 GHz 54 Mbps wireless adapter for USB 2.0. The adapter offers a stable connectivity in noisy environments, assuring a good transmission link with minimum interferences. This product can take full advantage of the 802.11g bandwidth and its compact and sleek design (27 grams and 88.9 x 28.5mm) makes it convenient and easy to install. Moreover, since the connection is made via USB, makes it possible to use an USB cable so that it can be placed anywhere in the car, in order to improve the communication conditions.

It was not possible to find any information regarding to the operating range. However, an overview from a similar PCI adapter (SMCWPCI-G2) should give us an approximate idea of the communication range that the device can achieve. (see Table 6.1 [34, 35]).

	1 Mbps	54 Mbps
Outdoor*	350 m	60 m
Indoor*	75 m	40 m
Sensitivity(2.412 - 2.484 GHz)	-92 dBm	-65 dBm

Table 6.1: Technical specifications of the USB adapter (specifications corresponding to the SMCWPCI-G2 adapter).

Figure 6.1: Adopted wireless USB adapter.

Since the regulations of the FST competition do not allow any devices being placed in the track, the workstation must be targeted to a wireless ad hoc network. The ad hoc network does not rely on a preexisting infrastructure, such as routers in wired networks or access points in managed (infrastructure) wireless networks.

6.4 Communication Protocol

The communication between the client and the server is achieved through a stream TCP socket . These sockets are full-duplex byte streams, similar to pipes and they ensure that data is

not lost or duplicated.

Nevertheless, there may be problems of re-establishment in case of connection losses. Therefore, a dedicated protocol was defined which will be described in the following section.

6.4.1 Message Format and Protocol

To guarantee that the data being transmitted from the car to the pit is not lost due to eventual connection losses of the wireless communication link a reliable communication mechanism was developed by another IST student in the scope of his MSc Thesis - Formula Student Racing Championship: design and implementation of the management and graphical interface of the telemetry system [36]. Such communication scheme was incorporated in the transmission module of this project in order to implement a reliable packeted protocol whose packet structure is in Table 6.2.

16 bits	8 bits	16 bits	Amount indicated by Size	8 bits
Package Number	Size	ID	Data	Checksum

Table 6.2: Package structure used by the adopted communication system.

The Package Structure:

- Package Number: The numbering of the packages starts at 0 and is automatically incremented with each new package. This field is useful for retransmission requests and package re.ordering when a package arrive to its destiny;
- Size: The size of the actual data, which is limited to a range of 0 to 255 bytes;
- ID: There are three types of packages - data package, acknowledge package and retransmission package. They are also identified by a priority status, as shown in Table 6.3;
- Car: The number of the car where the package is coming from, in case there is more than one car. This field is optional, since alternative solutions can equally be applied (e.g.: assign a different port to each car);
- Data: This is the actual data (audio or image/video), whose size is variable. The size is usually small, so that the amounts of retransmitted data being retransmitted (in case of error) is always small;
- Checksum: Allows to check the integrity of the package.

The data messages exchanged between the car and the pit include other signals acquired by several sensors in the car. Their goal is to obtain several mechanical parameters, such as tire

Priority Status	ID (bit)
High	0 - 32768
Medium	32769 - 49152
Low	49153 - 65536

Table 6.3: Package priority.

pressure and temperature, structure deformation and suspension displacement. Since all these signals are being transmitted at the same time, it was decided to define a set of priorities in order to establish a package discarding criteria whenever the offered bandwidth or the communication fidelity is not enough to assure the transmission of the whole set of signals. As such, it was considered that the signals coming from mechanical sensors should have higher priority, since they carry very important information. The multimedia signals generated in this project are assigned the **low** priority (images/video messages) and the **medium** priority (audio messages).

The IDs assigned to these signals are as follows:

- 49152 (Medium) - Audio.
- 49153 (Low) - Images;
- 49154 (Low) - Video.

To guarantee that no data is lost, the system is designed so that the packages that have not been acknowledged from the receptor are stored in a list, which is pre-initialized with a certain storage space. Its size is increased only when necessary. When the acknowledge is received the package is removed from the list. Otherwise, the sender will receive a retransmission request.

To avoid network congestion, in case of a transmission error or temporary connection loss (which causes the delay or loss of the packages), low and medium priority messages are the first to be discarded.

6.5 User Interface

Besides the multimedia acquisition, encoding and transmission modules, whose description was previously presented in chapters 3, 4, 5, respectively, it was also developed an integrated user interface to allow the user, at the pits, to control and playback the set of signals that are acquired (see Figure 6.2). The interface also allows the user to setup the set of parameters previously shown in Figure 2.4 and control the behavior of each signal. In other words, the user may control *when* to transmit or receive audio signals (whenever he wants to talk to the driver); *when* to receive an instant image (whenever he wants to get a snapshot) and also *when* to receive the acquired video in real time.

6.5.1 Layout

Figure 6.2 illustrates the conceived user interface. In following, it will be presented a brief description of the several pop-up menus, buttons and the main display.

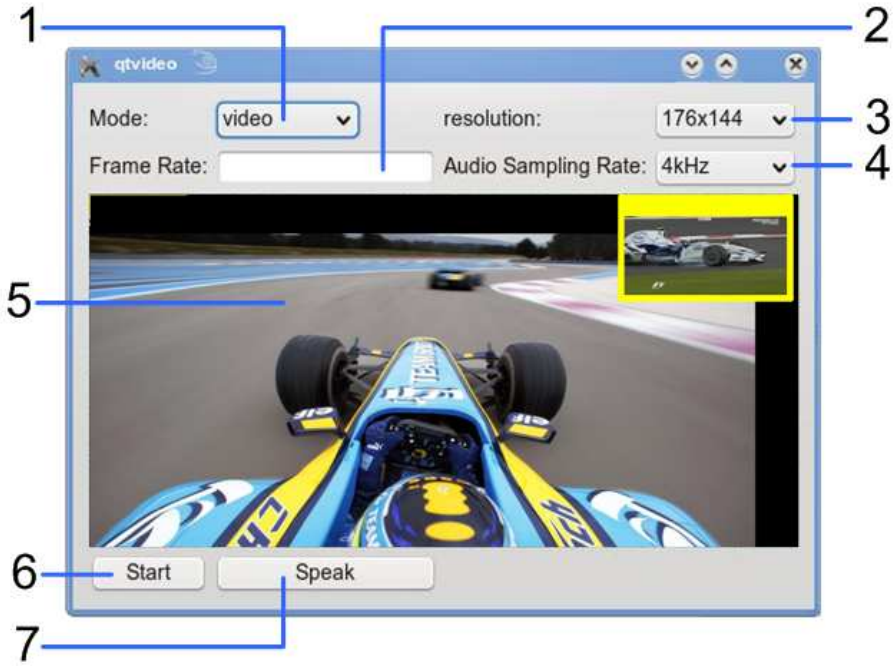

Figure 6.2: Conceived user interface.

1. Mode - Pop-up menu to choose between video or snapshot (image).
2. Text box - Text box to type the desired value of the video frame rate (how many frames per second) or the refresh rate (in seconds) of the images.
3. Resolution - Pop-up menu to choose the desired image/video resolution.
4. Audio sampling rate - Pop-up menu to choose one of the available sampling rates.
5. Display - Main display for playing video or for showing the snapshots.
6. Start - Press this button in order to start the acquisition of video or snapshots.
7. Speak - Press this button to talk to the driver.

6.5.2 Operating Functions

Upon startup, all the options are set to their default values. The user may just run the application with the default values or, if desired, change the necessary parameters.

In the following, it will be given a description of each function in detail.

Audio Options

The audio communication works just like a walkie talkie - push the button to talk and release it to listen to the incoming audio. By clicking the *Speak* button, the audio from the engineer's microphone in the pit is sent to the car, and when it is not pressed the audio coming from the driver's microphone is sent to the pit so that the engineer may listen to it. The only parameter that can be set up is the audio sampling rate. In order to set the sampling rate, the user only has to click the pop-up menu and choose one of the available values, as shown in Figure 6.3.

These parameters are also sent to the mobile station through the TCP socket, using the protocol stated in Table 6.2.

Figure 6.3: Audio sampling rate (pop-up menu).

Image Options

In order to capture still images from the car's webcam, the *snapshot* option must be set in the *Mode* pop-up menu. When this option is chosen, the *Frame Rate* indication in the text box changes to *Refresh Rate*. The refresh rate is the periodic delay between image acquisition. For example, if the user types 0.5 in the text box, it means that he wants to acquire an image every 0.5 seconds (in other words two pictures per second). If the user only wants a single image, just type 0.

Video Options

To start acquiring video at the car, the video option in the *Mode* pop-up menu must be set. The user can control when to play or pause it, by simply using the *Start* button. The parameters to be set for this signal are the video resolution and the frame rate. The resolution value must be chosen within the values available in the pop-up menu. The frame rate can be set by typing the desired value in the text box (see Figure 6.4).

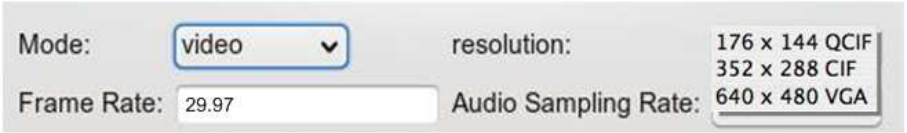

Figure 6.4: Video parameters.

Storage

One of the features described in previous chapters is the possibility of storing the acquired data in the flash memory installed in the mobile station. However, this feature is not controlled by the user. As such, the (encoded and compressed) data that is sent to the base station, is also automatically stored in the local flash memory and properly identified.

7

Performance Analysis and Conclusions

Contents

In this concluding chapter, it is presented a final discussion about the conducted analysis and evaluation of the achieved system performance. In the following, it will be given a summary of the obtained results, as well as some limitations of the conceived implementation.

7.1 System Evaluation and Performance Analysis

7.1.1 Evaluation of the Audio System

The audio acquisition and encoding module works as expected. However, the main limitation is concerned with the presence of noise that mixes with the voice signal. The main sources of noise are the engines, air and car vibration. Nevertheless, the first one may not be a problem in the near future, since the most recent prototype is equipped with an electric and more silent engine. Moreover, other design spaces may still be considered, in what concerns a proper integration of the headphones and microphone in the driver's helmet. Nevertheless, this implementation does not deal with the noise problem.

7.1.2 Evaluation of Image Acquisition

The system works as expected. Nevertheless, since the camera scans the entire picture line by line (since it has no memory), occasionally the result suffers from a motion blur, caused by the delay (latency) of the pixel transmission between the camera and the PC. As such, the lower the delay, the better the image quality. This allows to conclude that when a camera captures a moving object, the sharpness of the frozen image will depend both on the technology of the webcam and the speed of the pixel transmission channel.

7.1.3 Evaluation of the Video System

For purposes of analysis the values used are to be within 0.2 fps to 30.0 fps. When capturing frames the maximum frame rate that allows the application to run without losing frames is 14.0 fps. Higher rates may cause the lost of frames.

7.1.4 Signal Transmission (Wireless Connection)

Since the racing circuit is not always available for testing the conceived system, a good solution for testing the wireless connection and to overcome this protocol limitation is to choose a place that is not necessarily a racing circuit but that has the same obstacles that may interfere with the communication between both stations. As such, it was considered the worst case scenario, by conducting the tests in a place with several obstacles such as concrete structures, trees and uneven ground.

Figure 7.1: Testing area of the communication system: the green zone denote places where there is acquisition of signal; the red zone is where the signal is lost.

Figure 7.1 shows the area around the Tagus-Park campus of IST - Porto Salvo, Portugal; where the tests were conducted. The base station is located about 2 to 3 meters high. The mobile station moves around the area colored as green and red. The green area is where the signal sent from the base station is strong enough to reconnect (connection range). The red area is where the signal gets so weak that in case of a connection lost, it becomes impossible to re-connect. The “X” marks the exact spot, where the connection was lost in this particular test. The main reason is due to the presence of concrete structure and trees between the two nodes. Moreover, in this particular location the spot is located in a deeper area, about 2 meters below the ground level; to simulate the worst case scenario. The signal output power is 15 dBm. Around the green area the average data rate is 24 Mbps, decreasing to 18 Mbps in the border between the green and red zones, which is about 150 meters away from the base station. When the mobile station is close to the "X" spot (which is about 30 meters further), the data rate is about 2 Mbps. The highest data rate in the red zone is 11 Mbps. Refer to the graph in Figure 7.2.

The highest distance from the pit (base station) to another point in the racing circuit where the tests are usually conducted is about 300 meters. Nevertheless, this is easily overcome by placing the base station in the middle of the racing circuit. Moreover, in the racing circuit there are definitely less obstacles and the ground is more even.

7.2 Conclusions

The designed system for in-car audio and video capture is now a reality. The conceived application provides an interface that allows the pit team to monitor the race through the camera

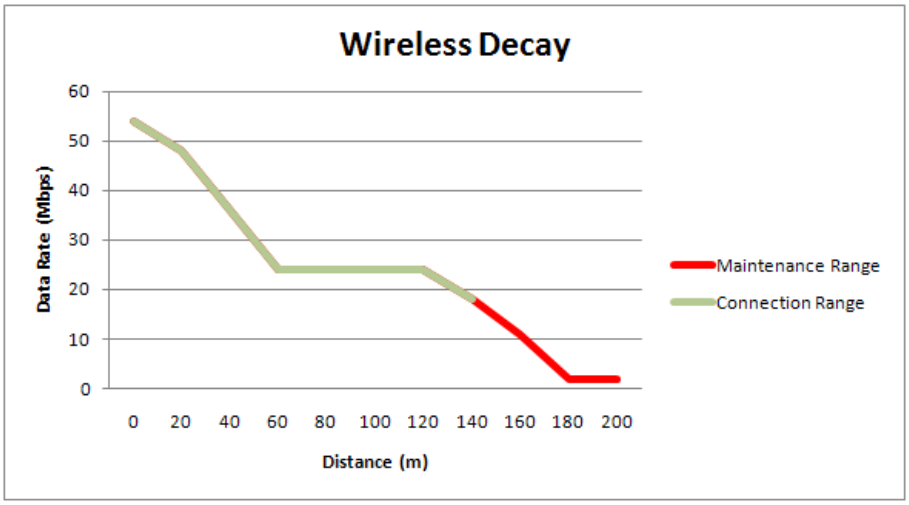

Figure 7.2: Wireless signal decay. Transmission power of the output is 15 dBm.

installed in the car and to perform audio communication with the driver via Wi-Fi. The software running in the mobile station is installed in a USB flash-disk drive, running a Knoppix Linux distribution, in a low-power VIA EPIA-P700 board that meets all the requirements such as space and weight. In fact, all components that compose the system's hardware in the car meet these requirements.

The bandwidth constraints were overcome by efficiently compressing the acquired signals before transmitting. The audio signal is captured in 8-bit PCM format and encoded to MP3, being transmitted at about 8 kbit/s with good quality. The video signal is acquired in YCbCr 4:2:0 pixel format at a maximum frame rate of 14 fps and encoded in MPEG-4 Part 2. The compression ratio is about 100:1 leading to a 150 kbps bit-stream to transmit 30 QCIF frames per second. Occasionally, the captured images are affected by motion blur, when capturing moving objects, which is a problem that could not be solved with the adopted webcam. However the bandwidth requirements are met. The compression ratio is about 10:1 for still images.

Finally, instead of encoding the whole set of the system parameters in the source code, the conceived user interface offers the possibility and flexibility to change and adapt the system characteristics, such as the frame resolution for image and video acquisition, or the sampling rate for audio acquisition. As such, it is possible to achieve a better balance when the bandwidth is not enough.

7.2.1 Budget Report

In order to comply with the strict budget restrictions, the entire software that was adopted was either cost free, including the codecs and the operating system, or was specially developed. The hardware was also acquired taking into account the ratio performance/price. Table 7.1 provides a

budget report for the hardware used in the entire project.

Item		
Hardware	**Description**	**Price** (€)
Embedded Board	Pico-ITX VIA P700-10L	222.00
Memory	KINGSTON SO-DDRII 1024MB 667MHz	13.67
Webcam	Logitech QuickCam Pro for Notebooks	49.79
Flash Memory	KINGSTON DataTraveler 101C USB 2.0 4GB	9.49
	Total	**294.95**

Table 7.1: Budget report.

7.2.2 Future Work

An important aspect that still deserves further research in the future is concerned with the possibility to incorporate audio noise-canceling technology. This can be done, for instance by using a sound card that provides more than one digital sampling device, */dev/dsp* and */dev/dsp1*. This way, one can use two microphones, one facing the driver to capture the speech and the other facing in the opposite direction, looking for noise. The noise can be suppressed by comparing the signals from both sources.

Another important aspect is concerned with the possibility to use another type of webcam. In particular, devices with internal memory that allows acquiring the frames at once, in order to avoid the motion blur.

Since the signal quality was not the main concern in this project, in contrast to other aspects such as bandwidth and the system efficiency in portable embedded systems; another possible improvement to this project is to add more options to the user interface. In such a case, if quality is a main concern (for instance), the user may be able to configure other parameters that are only available to the programmer. For example, the encoding standard and the parameterization of the codecs.

Bibliography

[1] P. FST. (2010, June) A história do projecto fst. Projecto FST. [Online]. Available: http://www.projectofst.com/

[2] f1technical, “Radio communications,” August 2006. [Online]. Available: http://www.f1technical.net/features/3759

[3] f1technical, “Car communication systems,” October 2005. [Online]. Available: http://www.f1technical.net/features/1230

[4] K. CORPORATION. (2010) The importance of radio systems in races.

[5] S. Communications. (2008, June) Selex communications participates and sponsors the 2008 edition of the tetra world congress in hong kong. http://www.selex-comms.com/internet/index.php?section=CORP&open0=5&open1=24&showentry=4536. SELEX Communications.

[6] M. International, “Scuderia ferrari selects selex communications to supply tetra solution,” December 2006.

[7] F. O. A. Ltd. (2009) Television cameras and timing transponders. http://www.formula1.com/inside_f1/rules_and_regulations/technical_regulations/5274/. Retrieved 2010.06.15.

[8] T. International. (2010) On-board camera systems. http://www.tv2international.com/onboard.html. TV2 International. Retrieved 2010.06.15.

[9] Nathesh, “Broadcast sports adds cavium hd video processors on-board race cars,” September 2009, nathesh, TMCnet Contributor.

[10] (2010) Specialized camera systems. http://www.broadcastsportsinc.com/sdhd-digital-wireless-on-board-cameras.aspx. BROADCAST SPORTS INC. Retrieved 2010.06.15.

[11] C. Networks. (2010) Purevu tm cnw5602 super low latency full-hd video processor soc preliminary product brief. http://www.caviumnetworks.com/pdfFiles/CNW5602_PB-1.0.pdf. Cavium Networks. Retrieved 2010.06.15.

[12] Racelogic. Retrieved 2009.09.18. [Online]. Available: http://www.racelogic.co.uk/

[13] (1996, August) Glossary of telecommunication terms.

[14] EPIA-P700 Mainboard, 1st ed., VIA Technologies, Inc., December 2008. [Online]. Available: http://www.via.com.tw/en/products/embedded/ProductDetail.jsp?id=690

[15] Via c7 processor specifications. http://www.via.com.tw/en/products/processors/c7/specs.jsp. VIA Technologies, Inc.

[16] iMedia Embedded Linux: Small Embedded Linux distribution for mini-ITX / x86 systems., Ituner Networks Corp - Mini-Box.com, 2007.

[17] Linux uvc driver and tools. http://www.ideasonboard.org. ideas on board SPRL. [Online]. Available: http://www.ideasonboard.org/uvc/

[18] T. L. Ce Xu and T. Tay, "H.264/avc codec: Instruction-level complexity analysis," in Proceedings of IASTED International Conference on Internet and Multimedia Systems, and Applications (IMSA 2005), Hawaii, August 2005, pp. 341–346.

[19] XBee™/XBee-PRO™OEM RF Modules Data Sheet, MaxStream, 2006.

[20] O. S. S. ™. http://www.opensound.com. [Online]. Available: http://www.opensound.com/oss.html

[21] (2008, January) Advanced linux sound architecture (alsa). http://www.alsa-project.org/. [Online]. Available: http://www.alsa-project.org/main/index.php/Main_Page

[22] J. S. R. A. B. L. M. H. N. Jayant, N.; Johnston, "Signal compression based on models of human perception," Proceedings of the IEEE, vol. 81, pp. 1385 – 1422, October 1993.

[23] The history of mp3. http://www.mp3licensing.com/mp3/history.html.

[24] The ogg container format. http://xiph.org. Xiph.Org. Retrieved 2009.09.28. [Online]. Available: http://xiph.org/ogg/

[25] The lame project. http://lame.sourceforge.net/. Retrieved 2009.09.25.

[26] pipe(7) - linux man page. http://linux.die.net. die.net. [Online]. Available: http://linux.die.net/man/7/pipe

[27] J. Corbet, "The lwn.net video4linux2 api series," October 2006. [Online]. Available: http://lwn.net/Articles/203924/

[28] H. V. M. R. Bill Dirks, Michael H. Schimek, Video for Linux Two API Specification: Revision 0.24. Free Software Foundation, 2008.

[29] (2004, April) Converting between yuv and rgb. http://msdn.microsoft.com/. Microsoft Corporation. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms893078.aspx

[30] V. R. W. Murray, James D., Encyclopedia of Graphics File Formats, 2nd Edition, 2nd ed., O'Reilly, Ed.

[31] I. E. G. Richardson, H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia, Wiley, Ed.

[32] R. Soheili, "Mpeg-4 part 2 video coding," July 2004.

[33] FFmpeg Documentation, http://ffmpeg.org/ffmpeg-doc.html, August 2010.

[34] SMCWUSB-G EZ Connect^{TM}g 2.4GHz 802.11g Wireless USB 2.0 Adapter, SMC®Networks, retrieved 2009.07.04.

[35] SMCWPCI-G2 EZ Connect^{TM}g 802.11g 54Mbps Wireless PCI Adapter, SMC®Networks, retrieved 2010.07.04.

[36] P. F. Mendes, "Formula student racing championship: design and implementation of the management and graphical interface of the telemetry system," Master's thesis, Instituto Superior Técnico (IST) - Universidade Técnica de Lisboa (UTL), 2010.

A

Audio Encoding Tools

Contents

This appendix provides the most relevant options among the several available parameters that are offered by each considered audio encoder.

A.1 LAME audio compressor

SYNOPSIS

lame [options] <infile><outfile>

OPTIONS

Input options:

-**r**

Assume the input file is raw PCM. Sampling rate and mono/stereo/jstereo must be specified on the command line. Without -r, LAME will perform several *fseek()'s* on the input file, looking for WAV and AIFF headers.

-**s** *sfreq*

sfreq = 8/11.025/12/16/22.05/24/32/44.1/48

Required only for raw PCM input files. Otherwise it will be determined from the header of the input file.

LAME will automatically resample the input file to one of the supported MP3 sampling rates if necessary.

Operational options:

-**m** *mode*

mode = s, j, f, d, m

(s)tereo

(j)oint stereo

(f)orced joint stereo

(d)ual channels

(m)ono

Joint-stereo is the default mode for stereo files with VBR when **-V** is more than 4 or fixed bitrates of 160kbps or less. At higher fixed bitrates or higher VBR settings, the default is stereo.

(m)ono

The input will be encoded as a mono signal. If the acquired input is a stereo signal, it will be downsampled to mono. The downmix is calculated as the sum of the left and right channels, attenuated by 6 dB.

-**b** *n*

For MPEG1 (sampling frequencies of 32, 44.1 and 48 kHz)

n = 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320

For MPEG2 (sampling frequencies of 16, 22.05 and 24 kHz)

n = 8, 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 128, 144, 160

Default is 128 for MPEG1 and 64 for MPEG2.

EXAMPLES

Encoding mono 8.0 kHz raw PCM from standard input, 64 kbps output:

lame -r -m m -**b** *64* -**s** 8.0 - sound.mp3

Encoding mono 8.0 kHz raw PCM from input pipe (NP1) to output (NP2):

lame -r -m m -**s** 8.0 NP1 NP2

A.2 Vorbis Tools

NAME

oggenc - encode audio into the Ogg Vorbis format

SYNOPSIS

oggenc [-**r**][-**B** *raw* input sample size] [-**C** *raw* input number of channels] [-**R** raw input samplerate] [-**b** nominal bitrate] [-**o** *output_file*] *input_files* ...

OPTIONS

Input options:

-**r**, −−raw

Assume input data is raw little-endian audio data with no header information. If other options are not specified, defaults to 44.1kHz stereo 16 bits. See next three options for how to change this.

-**B** n, −−raw-bit=n

Sets raw mode input sample size in bit. Default is 16.

-**C** n, −−raw-chan=n

Sets raw mode input number of channels. Default is 2.

-**R** n, −−raw-rate=n

Sets raw mode input sampling rate. Default is 44100.

-**b** n, −−bitrate=n

Sets target bitrate to n (in kb/s). The encoder will attempt to encode at approximately this bitrate.

-**o** output_file, −−output=output_file

Write the Ogg Vorbis stream to output_file (only valid if a single input file is specified).

EXAMPLES

Encoding mono 8.0 kHz, 16-bit raw PCM from standard input, 24 kbps output:

oggenc -r -R 8000 -**B** 16 -**C** 1 -**b** 24 - -**o** sound.ogg

B

Images Processing and Encoding Tools

Contents

B.1 V4L2 API V4L2_PIX_FMT_YUYV ('YUYV') Format Definition

Name

V4L2_PIX_FMT_YUYV - Packed format with 1/2 horizontal chroma resolution, also known as YUV 4:2:2.

Description

In this format, each set of four bytes encodes two pixels, corresponding to two Y's, one Cb and one Cr components. Each Y goes to one of the pixels, and the Cb and Cr components belong to both pixels. As such, the Cr and Cb components have half the horizontal resolution of the Y component. V4L2_PIX_FMT_YUYV is known in the Windows environment as YUY2.

Example: V4L2_PIX_FMT_YUYV 4 x 4 pixel image

Byte Order. Each cell is one byte.

start + 0:	Y'00	Cb00	Y'01	Cr00	Y'02	Cb01	Y'03	Cr01
start + 8:	Y'10	Cb10	Y'11	Cr10	Y'12	Cb11	Y'13	Cr11
start + 16:	Y'20	Cb20	Y'21	Cr20	Y'22	Cb21	Y'23	Cr21
start + 24:	Y'30	Cb30	Y'31	Cr30	Y'32	Cb31	Y'33	Cr31

B.2 CJPEG Encoder

NAME

cjpeg - compress an image file to a JPEG file

SYNOPSIS

cjpeg [options] [filename]

OPTIONS

-**quality N**

Scale quantization tables to adjust image quality. Quality is 0 (worst) to 100 (best); default is 75.

EXAMPLE

This example compresses the PPM input file input.ppm and saves the output as out.jpg:

```
cjpeg input.ppm > out.jpg
```

B.3 YCbCr 4:2:2 to YCbCr 4:4:4 Conversion

In the following, it will be presented the pseudo code corresponding to the routine that implements the YCbCr 4:2:2 to YCbCr 4:4:4 conversion.

```
/*variables*/
p; // p is YUV422 frame. Size = width*height + width*heigh/2
Y, U, V; //Y, Cb and Cr planes. Size of each plane = width*height

/*Convert YUV422 to YUV444*/
for(j=0;j<height;j++)
  for(i=0;i<width;i+=2)
    k=(int)(j/2);
    Y[j*width + i] = p[j*width*2 + i*2];/*even elements of Y plane*/
    Y[j*width + i + 1] = p[j*width*2 + i*2 + 2];/*odd elements of Y plane*/
    /*duplicate the elements of Cb and Cr plane*/
    U[(2*k)*width + i] = U[(2*k+1)*width + i] = p[j*width*2 + i*2 + 1];
    V[(2*k)*width + i] = V[(2*k+1)*width + i] = p[j*width*2 + i*2 + 3];
```

B.4 YCbCr 4:4:4 to RGB

In the following, it will be presented the pseudo code corresponding to the routine that implements the YCbCr 4:4:4 to RGB conversion.

```
/*variables*/
Y, U, V; /*Y, Cb and Cr planes. Size of each plane = width*height*/
RGB; /*store converted data*/

/*Convert YUV444 to RGB*/
j=0;
for(i=0;i<width*height;i++){
  C = Y[i] - 16;
  if(i%2==0){/*even elements*/
    D = U[i] - 128;
    E = V[i] - 128;
  }
```

```
  else{/*odd elements*/
    D =  U[i-1] - 128;
    E =  V[i-1] - 128;
  }
  /*limit pixel value to 255*/
  RGB[j++] = clip(( 298 * C           + 409 * E + 128) >> 8);
  RGB[j++] = clip(( 298 * C - 100 * D - 208 * E + 128) >> 8);
  RGB[j++] = clip(( 298 * C + 516 * D           + 128) >> 8);
}
```

C

Video Processing and Encoding Tools

Contents

C.1 Mplayer

SYNOPSIS

./mplayer [options]

OPTIONS

-**cache** <**kBytes**>

This option specifies how much memory (in kBytes) to use when precaching a file or URL. Especially useful on slow media.

-**demuxer** < [+]**name**>

Force demuxer type. Use a '+' before the name to force it, this will skip some checks! Give the demuxer name as printed by -demuxer help. For backward compatibility it also accepts the demuxer ID as defined in libmpdemux/demuxer.h.

-**rawvideo** <**option1:option2:...**>

This option lets you play raw video files. You have to use -demuxer rawvideo as well.

Available options are:

fps=<value>

rate in frames per second (default: 25.0) sqcif|qcif|cif|4cif|pal|ntsc

set standard image size

w=<value>

image width in pixels

h=<value>

image height in pixels

i420|yv12|yuy2|y8

The camera supports only i420 (YUV 4:2:0) and yuy2 (YUV 4:2:2).

set colorspace

format=<value>

colorspace (fourcc) in hex

size=<value>

frame size in Bytes

EXAMPLE: mplayer sample-720x576.yuv -demuxer rawvideo -rawvideo w=720:h=576

Play a raw YUV sample.

C.2 FFmpeg

Name

ffmpeg - FFmpeg video converter

Synopsis

ffmpeg [[infile options][-i infile]]... [outfile options] outfile...

Options

Main options

-f fmt

Force format.

-i filename

input filename

-y

Overwrite output files.

Video Options

-b bitrate

Set the video bitrate in bit/s (default = 200 kb/s).

-r fps

Set frame rate (Hz value, fraction or abbreviation), (default = 25).

-s size

Set frame size. The format is wxh (ffserver default = 160x128, ffmpeg default = same as source). The following abbreviations are recognized:

sqcif 128x96

qcif 176x144

cif 352x288

4cif 704x576

Advanced Video Options

-pix_fmt format

Set pixel format.

-qscale q

Use fixed video quantizer scale (VBR)

-qmin q

minimum video quantizer scale (VBR)

-qmax q

maximum video quantizer scale (VBR)

EXAMPLE: ffmpeg -f rawvideo -**r** framerate -**s** widthxheight -**pix_fmt** yuv420p -**i** IN_PIPE -**vcodec** mpeg4 -**r** framerate -**y** OUT_PIPE

Encode the raw YCbCr data from the input pipe (IN_PIPE) to MPEG-4 and writes it to the output pipe (OUT_PIPE).

C.3 YCbCr 4:2:2 to YCbCr 4:2:0 Conversion

In the following, it will be presented the pseudo code corresponding to the routine that implements the YCbCr 4:2:2 to YCbCr 4:2:0 conversion.

```
/*variables*/
p; // p is YUV422 frame. Size = width*height + width*heigh/2
Y, U, V; /*Y, Cb and Cr planes. Size of Y plane = width*height*/
         /* Cb plane and Cr, same size = width*height/4 */

/*Convert YUV422 to YUV444*/
for(j=0;j<height;j++)
    for(i=0;i<width;i+=2)
      Y[j*width + i] = p[j*width*2 + i*2];/*even elements of Y plane*/
      Y[j*width + i + 1] = p[j*width*2 + i*2 + 2];/*odd elements of Y plane*/
      U[(j/2)*(width/2) + i/2] = p[j*width*2 + i*2 + 1];
      V[(j/2)*(width/2) + i/2] = p[j*width*2 + i*2 + 3];
```

C.4 Flow Charts: Server and Client

The flow charts presented in Figure C.1 and Figure C.2 illustrates the general operation of the server process and the client process respectively.

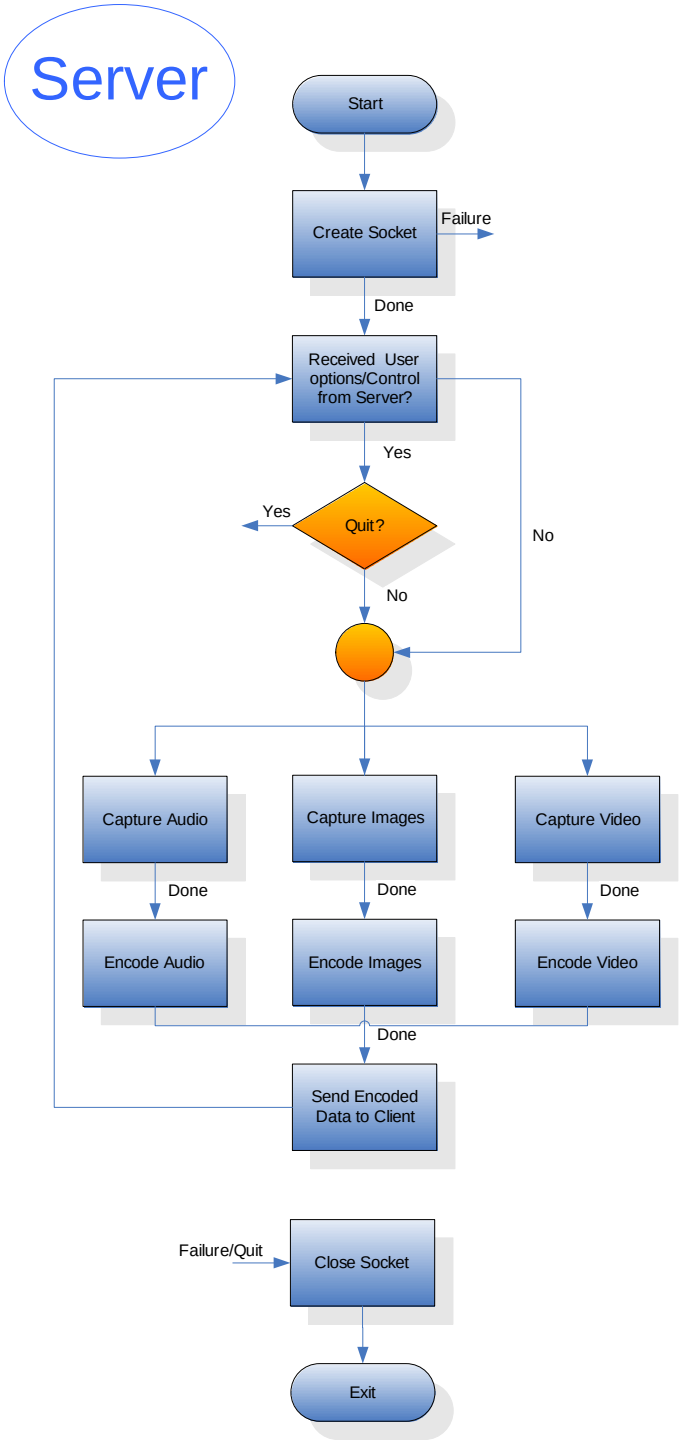

Figure C.1: Flow chart of the server operation.

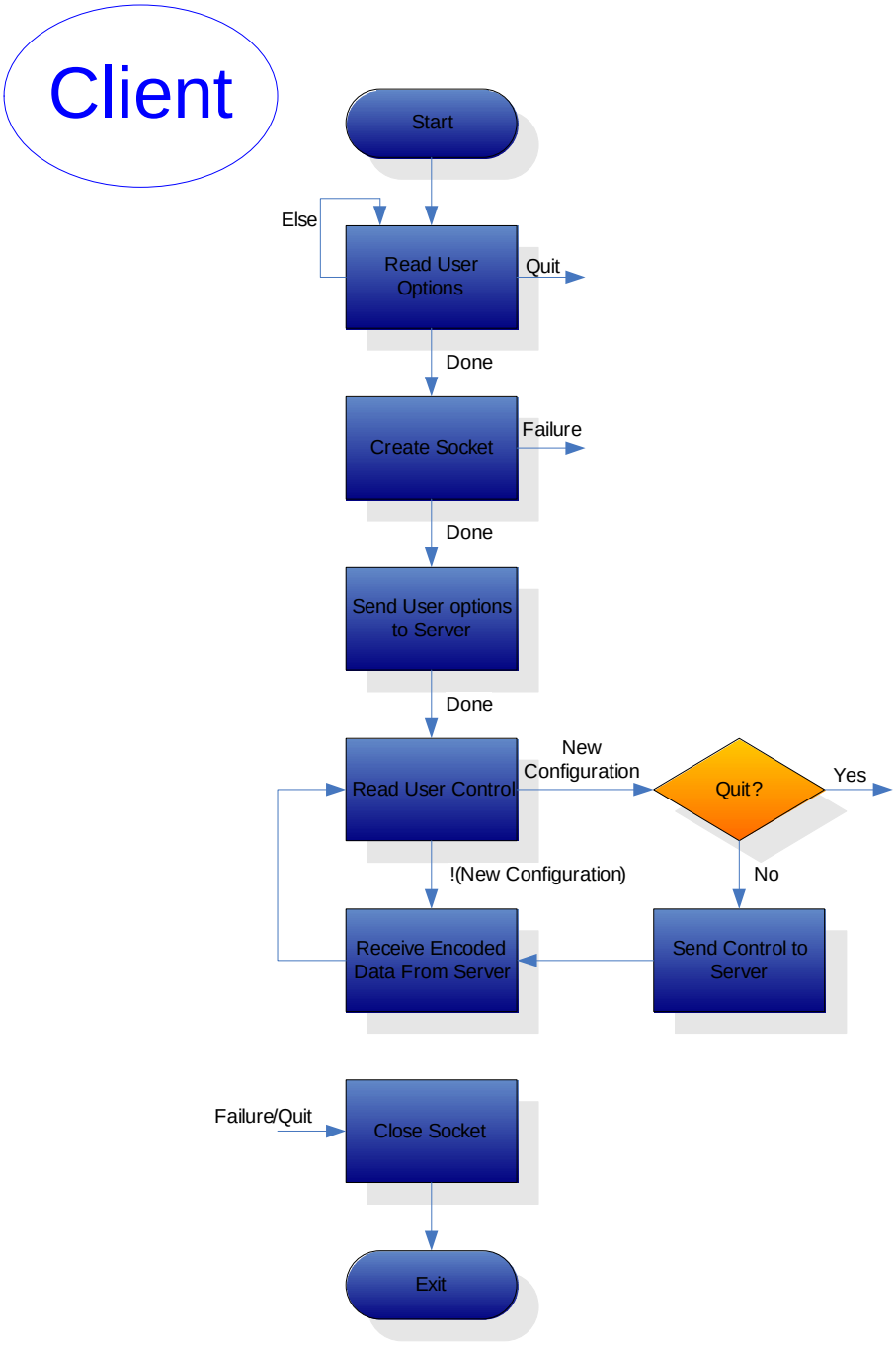

Figure C.2: Flow chart of the client operation.