UPPSALA
UNIVERSITET

UPTEC F17 010
Examensarbete 30 hp
Mars 2017

Digital camera technology for off-highway vehicles

Robert Zak

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Digital camera technology for off-highway vehicles

Robert Zak

Off-highway vehicles are on the verge of switching from analog to digital video camera technology (VCT), which offers better video quality and new features but adds complexity to the system. This thesis project aims to implement the digital VCT to the display computer CCpilot VA intended for off-highway vehicles. In this project the differences between analog and digital VCTs were reviewed and then a demo displaying a live digital camera video feed on the embedded Linux based display computer CCpilot VA was implemented with Qt and QML.

More specifically, different GStreamer pipelines were tested, as Qt uses GStreamer to play video, and camera settings were changed using the ISO 17215 standard. The demo displayed a live digital camera video feed with high quality, low latency and high frame rate on the VA by using a GStreamer pipeline utilizing hardware decoding. The results have shown that digital video cameras perform better than analog cameras, primarily because digital cameras have better video quality. The attempts to simultaneously display a video feed and a Graphical User Interface created by Qt have been made. However, they were only successful with poor video performance. A zero-copy link between the GStreamer pipeline's decoder and sink element must be used to obtain good video performance.

Handledare: Anders Svedberg
Ämnesgranskare: Ping Wu
Examinator: Tomas Nyberg
ISSN: 1401-5757, UPTEC F17 010

Populärvetenskaplig sammanfattning

I många moderna personbilar finns backkameror för att underlätta backning och parkering av bilen. Dessa kameror är sedan några år oftast digitala istället för analoga, vilket ger en skarpare bild och underlättar bildanalys. Detta i kombination med säkerhetslagstiftning och ett förväntat lägre pris gör digitala videokameror intressanta för terrängfordonsindustrin. Förutom backkameror kan kameror bland annat placeras för att se vad som finns i en grävskopa eller nära fordonets kanter. Speciella kameror behövs dock för att klara av de hårda påfrestningar kamerorna kommer att utsättas för i terrängfordon och dessa kameror är ännu i prototypstadiet

Signalen från en digital videokamera måste avkodas i en dator före den kan visas på en skärm, och denna avkodning kan ske på olika sätt. För att belasta datorn minimalt har vissa datorer speciell hårdvara, till exempel en VPU, där videosignalen kan avkodas utan att det belastar huvudprocessorn, CPUn. Crosscontrols skärmdator CCpilot VA har en VPU och det var därför intressant att undersöka om den modellen klarar av att i realtid visa video från en digital videokamera samtidigt som den kan utföra andra processorkrävande uppgifter.

I det här examensarbetet beskrivs och jämförs analog och digital videokamerateknologi samt ISO 17215 standarden för att kommunicera med IP-kameror. CCpilot VA använder GStreamer för att visa video och genom att ändra det kommando som startar videovisningen kan VPUn användas för att avkoda videon. Olika kommandon testades och ett kommando som gav bra videokvalitet, hög uppdateringsfrekvens, låg fördröjning och som endast begränsat belastade CPUn hittades. Användargränssnittet för CCpilot VA skapas i Qt, men försöken att samtidigt visa både ett användargränssnitt skapat i Qt och video med bra kvalitet misslyckades. Efter att ha modifierat Qts multimediamodul kunde VPUn användas och video visas samtidigt som ett användargränssnitt, men på grund av att kopplingen mellan det avkodande elementet och det videovisande elementet involverade onödig kopiering av videoströmmen belastades CPUn hårt och videoprestandan blev dålig. Med GStreamer kan ett enkelt gränssnitt skapas och visas samtidigt som video med bra kvalitet visas utan att CPUn blir alltför belastad.

Contents

Table 0.1: Abbreviations and acronyms

AHD	Analog High Definition
API	Application Programming Interface
AUTOSAR	AUTomotive Open System ARchitecture
CAN	Controller Area Network
CCpilot VA	A display computer manufactured by Crosscontrol
CPU	Central Processing Unit
GUI	Graphical User Interface
H.264	A video compression format
IP	Internet Protocol
IPU	Image Processing Unit
IPv4	Internet Protocol version 4
IP67/IP68	International Protection Marking 67/68
JPEG	An image compression format
MJPEG	Motion JPEG, a video compression format
NTSC	A broadcasting standard
OS	Operating System
PAL	A broadcasting standard
QML	Qt Meta Language or Qt Modeling Language
Qt	A cross-platform application development framework
RTP	Real-time Transport Protocol
SOME/IP	Scalable service-Oriented MiddlewarE over IP
SW	Software
UDP	User Datagram Protocol
USB	Universal Serial Bus
VA	See CCpilot VA
VCIC	Video Communication Interface for Cameras
VCT	Video Camera Technology
VPU	Video Processing Unit
WVGA	A graphics display resolution

1 Introduction

1.1 Background

Camera technology in the off-highway vehicle industry is now making a paradigm shift; moving from analog to digital camera technology. Digital camera solutions are becoming more and more interesting in this industry because of safety legislations, functionality, performance and an expected price decrease. The private car industry has already integrated digital camera solutions into existing products, but for larger vehicles, such as agricultural and forestry machines, integration into existing products is only beginning now. Special cameras are needed to fit this industry, where the operating environment will be harsh and demanding, and the camera suppliers are now in the prototype and launch phase.

The move to digital cameras enables new features and increased image quality, but it also adds new challenges and the integration is more complex. To integrate a digital camera requires, among other things, an operating system that supports live-streaming video, that some camera parameters are known to the receiving computer, that appropriate software is available and that the hardware can handle video streaming. One of the new features offered by digital cameras made for the off-highway industry is the possibility to configure certain settings, such as video resolution and compression type, programmatically through the ISO 17215 standard.

Crosscontrol, as a display computer supplier, needs to understand this technology domain to support our customers in the best way and also be able to act as a trusted advisor for how to integrate and use these cameras in our customers' applications. The new features and advantages as well as the new challenges need to be investigated in order to know if the switch from analog to digital cameras is possible and justified.

Crosscontrol has previously implemented a video application for displaying an analog camera video feed on one of their display computers and now wants to investigate the possibility of using a digital camera instead of an analog camera. The application should be created using Qt and QML and be able to display a live video feed from a digital camera on a Crosscontrol display computer.

1.2 Purpose and goals

The goals of this thesis are to review the differences between analog and digital video cameras and to investigate if it is possible to connect digital cameras to the display computer CCpilot VA and show the live camera feed with high quality, low latency and high frame rate without putting too much load on the CPU [1]. If it is possible to show a good camera feed, the final goal is to implement the video feed as a part of the graphical user interface created in the software Qt.

1.3 Tasks and methodology

Information about analog and digital video cameras was found through a literature review and compared to highlight the differences between the two technologies. To investigate the possibility of satisfactory showing the camera feed on the VA was a multiple step process, which started by reading the existing pre-release documentation of the specific camera prototypes that were to be used and by examining the software and drivers available on the VA. A test command for streaming the camera feed using the software GStreamer was found in the documentation and the needed GStreamer plugins were confirmed to be installed on the VA. A first test was conducted after one of the cameras was powered up and connected to the same network as the VA.

The test command was hardware independent and did not take advantage of hardware decoding, which meant that the CPU had to do the decoding. The next step was to read the documentation and tutorials of GStreamer as well as relevant online forum discussions in order to better understand the parameters used and thus be able to use hardware decoding. Advantages and disadvantages of different video formats and coding standards were gathered through a literature review and compared with the possible camera settings and the plugins available to the GStreamer version used by the VA. The ISO 17215 standard, which is used to communicate with the cameras, was learned to enable configuration of the camera settings, such as video resolution, compression type and stream protocol.

Qt uses standard C++ with extensions and have multiple available modules, such as Qt Multimedia. Basic understanding of Qt was gained through the Qt Documentation and basic C++ knowledge was refreshed through online tutorials. Some parts of the Qt Multimedia module were modified to enable custom GStreamer commands and a program for changing camera parameters was written, which included setting up an UDP connection and sending according to the ISO 17215 standard.

2 Video camera technology

2.1 Analog video cameras

An analog camera's lens focuses the image on a sensor, which continuously is scanned to produce a time-varying analog signal that consists of the scene intensity information and the corresponding synchronization pulses. These signals are displayed in real time on an analog monitor or stored on analog media, such as analog video tapes, and are therefore hard to directly process and analyze. A third option is to convert the analog signal to a digital signal, which is what happens when an analog signal is displayed on a digital monitor [26], [29].

2.1.1 Video quality

Traditional analog video is restricted to an equivalent resolution of around 0.2 megapixels because of limitations of the NTSC/PAL standard. Removing the limitations of the standard heightens the resolution to the equivalent of 0.41 megapixels for PAL and 0.35 megapixels for NTSC [2]. There exists some technologies that brings high definition analog video, such as AHD, which delivers up to 2 megapixels resolution. These technologies output analog signals over coaxial cables, but the image sensors of the cameras outputs a digital signal, which is converted to an analog signal inside the cameras, and special hardware is often needed to display the video feed [3], [4]. Lens quality and light gathering capabilities are other important parameters that decide the video quality, but these parameters are not as strongly coupled to whether the camera is analog or digital as the resolution is [2].

2.1.2 Transmission and security

Analog signals are usually sent over coaxial cableset cables and the and the signals are not amenable to compression, which further impose a limitation on the resolution. The analog signal is vulnerable to external signal disturbances, which reduces the video quality gradually when the disturbances increase. The maximum recommended length for a coaxial cable without amplifier varies between 61 and 550 meters depending on the coaxial type used and is up to 1980 meters with a amplifier.

One cable per camera is needed to get the analog signals from multiple cameras to the monitor. The monitor usually only have one or two analog inputs, which means that if more cameras are used, an analog matrix switcher is needed between the cameras and the monitor, which can handle multiple analog inputs and output a chosen input or a combination of multiple inputs. The analog cables can only transmit the camera signal and analog cameras do thus need separate cables for controlling and powering the camera. If an analog signal is sent over a cable, a wire or a node in the system must be physically

reachable in order to eavesdrop. Analog signals can be sent wirelessly, but the scrambling techniques offer only a limited security against eavesdropping [29].

2.2 Digital video cameras

A digital camera has the same working principle as an analog camera, but the produced signal is digital and thus easier to process. The signal can for example be compressed before it is sent from the camera, which reduces the necessary bandwidth and storage size. Since the signal is digital already in the camera, video analysis can be made by the camera, for example can moving objects and humans be detected. These smart cameras can then send this information together with the video signal to the receiving units. A digital camera that sends and receive data via a computer network is called an IP camera [29].

2.2.1 Video quality

The video resolution of digital cameras is very high and is increasing with new cameras. As an example, the IndigoVision Ultra 5K Fixed Camera delivers a video resolution of 20 megapixels [5]. If everything except the resolution is the same for two cameras, the camera with the lower resolution will have the better light gathering capabilities and thus less noise. To achieve a higher resolution with the same sensor size, each light gathering component, the photosite, has to be smaller. It is possible to increase the size of the sensor, thus preventing the need to reduce the size of the photosites, but a larger sensor costs more. Furthermore, two lenses with the same size and resolution might produce video with different quality and it is therefore necessary to compare the output of different cameras to decide which one that has the highest quality [2].

2.2.2 Transmission and security

Digital signals from IP cameras are usually sent over Ethernet cables and the signals can be compressed up to 50 or 100 times without losing much quality, which reduces the size of the signal and thus the required bandwidth. Digital signals are immune to many external disturbances, but when a certain number of errors occur, the signal is gone, meaning that nothing from the signal can be displayed. The maximum length of a Category 5 or 6 Ethernet cable following the TIA-EIA-568-A standard is 100 meters, but there is no maximum length specified for connecting multiple Ethernet cables by switches or Ethernet extenders [6], [7].

Each digital camera is connected with one cable, but for IP cameras the Ethernet cables can be connected to an IP switch, which can send multiple camera outputs on a single Ethernet cable if the bandwidth is sufficient. Some IP cameras require only one cable, because the Ethernet cable can carry camera output, control signals and power for the camera simultaneously. Currently a maximum of 25.5 Watts can be supplied by the Ethernet cable, but an updated version that will be able to supply more than 50 Watts is under development

[8], [38]. A node or a wire must be physically reachable to eavesdrop on a digital signal sent in a cable [9]. Digital signals can be sent wirelessly and encryption techniques enable high security against eavesdropping [29], [2].

2.3 Digital video encoding standards

Digital cameras encode the video stream in a specific video format, such as MJPEG or H.264. The different formats have different characteristics, including compression rate, complexity and quality, which make them suitable for different tasks. Video that is to be live streamed needs to have reasonably low complexity, as this reduces the computational need, and thus the delay time, of both the encoding camera and the decoding computer that displays the video. The cameras used in this theses encode the video into MJPEG, but the camera manufacturers are working on making H.264 encoding possible as well. A higher compression rate reduces the bandwidth needed to send the signal, but usually at the cost of a higher compression complexity. The video compression can be either lossy, which reduces the quality be removing information, or lossless, which only removes redundant bits and thus reduces the size without losing information. While information is lost with lossy compression, this information might not be perceivable by humans watching the video and the removal of this information enables a much higher compression rate than lossless compression [35].

2.3.1 MJPEG

Motion JPEG, MJPEG, is a intraframe only video compression format that encodes a series of digital images to a video sequence. The JPEG compression method is applied to each of the individual images, but no compression is done to a series of images, as is done in other video compression formats. This makes the complexity of the compression low, and it does therefore require less computational power to encode and decode the MJPEG format compared with more complex compression formats. MJPEG videos are robust, since there is no compression on a series of images, which means that images dropped during transmission will not effect coming images. The cost of having low complexity and not considering series of images is that the compression ratio is low, which require high-bandwidth communications [34].

2.3.2 H.264

H.264 is a interframe video compression format, which means that it takes advantage of the similarities between neighboring images. It is therefore often possible to achieve a much higher compression ratio than for MJPEG, but the ratio is dependent on how similar the neighboring images are. The dependency between images make H.264 less robust than MJPEG, which means that images lost during transmission might cause later successfully transmitted images to be lost. The high compression ratio makes H.264 more suitable for low-bandwidth

communications or higher definition video on high-bandwidth communications than MJPEG. The H.264 compression is more complex than MJPEG and thus require more computational power, which might increase the latency [35].

2.4 ISO 17215: Road vehicles — Video communication interface for cameras (VCIC)

The ISO 17215 standard "has been established in order to define the use cases, the communication protocol, and the physical layer requirements of a video communication interface for cameras which covers the needs of driver assistance applications" [27]. The SOME/IP protocol, which uses UDP to transport the messages with IPv4 on a physical Ethernet layer, is used to control the camera. An overview of the ISO 17215 standard can be seen in fig. 2.1. The API is independent of the programming language, which means that the developer can choose a preferred language.

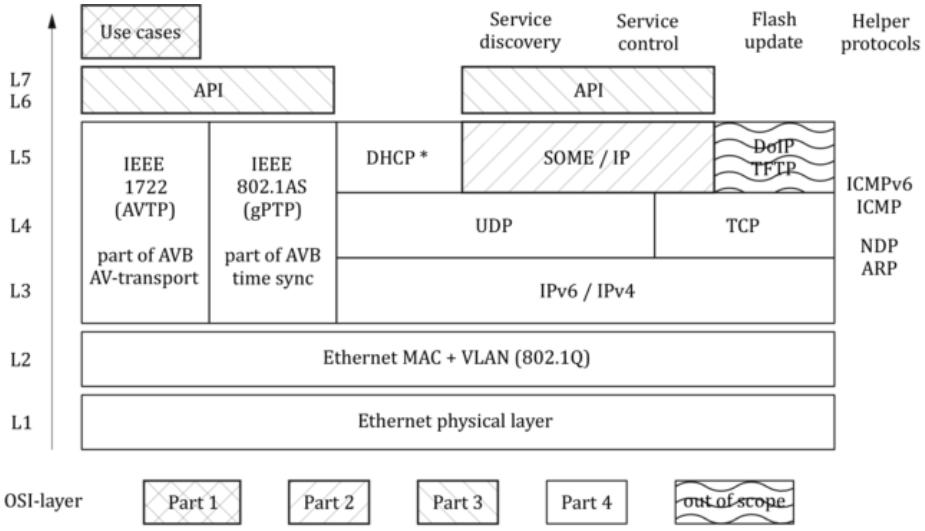

Figure 2.1: An overview of ISO 17215 standard

Settings that can be modified using the 17215 standard include the camera IP address, input and output resolution, video format and stream protocol. The standard is independent of the camera used and specifies which bytes should be sent in which order as well as the corresponding success and error messages. Most of the settings can be changed without rebooting the camera, for example the output resolution. This enables a more dynamic use of the camera stream, where multiple camera streams can be resized to fit the screen, without increasing the processing load on the receiving system [33] or needing to restart the camera.

2.4.1 The SOME/IP protocol

SOME/IP is a protocol created for the automotive industry that allows service-based communication and is highly scalable, because it "allows efficient use of unicast and multicast" [10]. It is compatible with multiple operating systems being used in the automotive industry, including AUTOSAR and Linux [28]. The protocol defines a payload and a header, which contains the method ID, service ID, length, client ID, session ID, protocol and interface version, message type and return code, while the ISO 17215 standard defines how the header and payload should be set for different functions. The structure of the SOME/IP protocol can be seen in table 2.1. Both the header and the payload of the SOME/IP protocol is in turn the payload of the protocol below, in this case UDP.

Table 2.1: The structure of the SOME/IP protocol

Message ID (Service ID [16 bits], Method ID [16bits])			
Length [32 bits]			
Protocol Vers.[8 bits]	Interface Vers.[8 bits]	Message Type[8 bits]	Return Code[8 bits]
Payload [variable length]			

2.4.2 UDP

UDP is a fast internet protocol without reliability checking, which reduces the overhead cost at the price of not being able to guarantee the delivery of packets. There is also no protection to prevent packets from arriving in the wrong order or multiple times. These features make it suitable for real-time applications where small data losses are preferred over delay, such as live video streaming. The UDP header contains the source and destination port, the length and a checksum with the structure seen in table 2.2. The source port defines which port the receiving end should reply to, the destination port is the port on the receiving end that the packets are sent to, the length is the length of the header plus the length of the data to be sent and the checksum is used for error-checking [30].

Table 2.2: The structure of the User Datagram Protocol, UDP

Source Port[16 bits]	Destination Port[16 bits]
Length[16 bits]	Checksum[16 bits]
Payload [variable length]	

2.4.3 An example ISO 17215 command

One of the ISO 17215 functions are called SubscribeROIVideo and it starts the video transmission of a previously defined Region Of Interest. A Region of Interest is a complete video setting with defined video parameters, such as input region, output resolution and type of video compression, and each camera can store a specific number of these complete video settings. Each Region Of Interest is stored at a specific ROIIndex, and the SubscribeROIVideo function must specify which ROIIndex to subscribe to. The ISO 17215 document specifies the structure and values of the header, which contains in the following order: Service ID, MethodID, Length, ClientID, SessionID, ProtocolVersion, InterfaceVersion, MessageType and ReturnCode. Table 2.3 explains how to set the values.

Table 2.3: The structure, values and an explanation of how to set the values of the SOME/IP header.

Variable	Value	Explanation
Service ID [2 bytes]	17215	The SOME/IP service ID
MethodID [2 bytes]	305	The SubscribeROIVideo method ID
Length [4 bytes]	12	The number of bytes coming *after* Length
ClientID [2 bytes]	1494	Should be unique in the network
SessionID [2 bytes]	14	Sets the order of the functions
ProtocolVersion [1 byte]	1	Should be 1
InterfaceVersion [1 byte]	1	Should be 1
MessageType [1 byte]	0	0 means request and 128 means response
ReturnCode [1 byte]	0	0 means OK
ROI index [4 bytes]	4	Which ROI index to stream

The settings to be changed, in this case the ROIIndex, is set after the header with the syntax also specified in the ISO 17215 document, and must in this case be an uint32. To start the video transmission with the Reigon Of Interest defined in ROIIndex 4 the following should be sent, without the commas: 0x433f, 0x0131, 0x0000000c, 0x05d6, 0x000e, 0x01, 0x01, 0x00, 0x00, 0x00000004. This is the same as the header in table 2.3, but in hexadecimal numbers, and with the ROIIndex set to 4 with the uint32 0x00000004. The number of bytes *after* the length value is 12, and therefore the length is set to 12. Some functions, such as SetRegionOfInterest, can not be used directly, but require the SetCamExclusive function to be used before it and EraseCamExclusive after it. In such a case it is important to set the SessionID correctly, for example by setting it to 12 for SetCamExclusive, 13 for SetRegionOfInterest and 14 for EraseCamExclusive.

One reason for this is that the SOME/IP functions are the payload of UDP, which, as stated in chapter 2.4.2, do not guarantee the packets to arrive in the right order.

2.5 Differences between digital and analog video cameras

2.5.1 Video quality

Even very low end digital cameras have better video resolution than analog cameras, because the quality of the analog signal is very restricted. If the best cables are used and the limitations of the NTSC/PAL standards are removed, the maximum resolution is the equivalent of 0.41 megapixel, which can be compared with the resolution of up to 20 megapixels for digital video cameras. The extra resolution makes it possible to cover a wider area with one camera and grants the ability to zoom into the video, while still having adequate video resolution. Higher resolution is an important and easily countable way of increasing the video quality, but resolution is not the sole metric that determines the video quality. Therefore the general answer is that digital cameras have better video quality, even though some high end analog cameras might produce better quality than low end digital cameras.

2.5.2 Transmission

The distance an analog signal can be sent in a coaxial cable is more than five times longer than a digital signal can be sent in an Ethernet cable, but when using amplifiers or switches the maximum length for an analog signal is still restricted to 1980 meters, while digital signal does not have a maximum limit. The compression available for digital signals enables more information, such as higher resolution, to be sent in digital signals than in analog signals. The quality of analog signals gets gradually worse when the noise increases, while digital signals remain the best quality up to a certain noise level, at which to information in the signal is completely lost. Ethernet cables are able to transfer not only the video signal, but also control signals and power for the camera. This reduces the number of cables needed compared to analog cameras, where separate cables are needed for control signals and power. Both analog and digital signals can be transmitted wirelessly, but IP networks, which can use digital encryption, have a much higher level of security than analog systems. IP networks can transmit over the internet, and thus quickly send a video to a remote location or a handheld device.

2.5.3 Security

When the signals are sent over a cable, a wire or a node in the system must be physically reachable in order to launch an attack. Since the cameras will be used in the off-highway vehicle industry, the wires and nodes should mostly be safely located within the chassis. An extra layer of protection, which can secure the transmission even if a wire or node is intercepted, is added by scrambling or

encrypting the signal. Digital encryption can be made much more secure than analog scrambling, which also is true if the signal is transmitted wirelessly [29].

2.5.4 Reliability

Both analog and digital cameras are reliable. If the signal from a digital camera contains to many error and cant be shown, the monitor will usually display the last viewable image. This is dangerous, because the viewer might think that the video is being updated and take actions accordingly. For example might a rear view camera on a vehicle show that nothing is behind the vehicle, even though a human is walking behind the vehicle. To prevent this, a warning can be displayed if the video stream contains to many errors. An analog camera will instead display more noise as the error rate increases, which gradually will make it impossible to recognize what the video is showing.

2.5.5 Intelligence

The signal needs to be digital in order to analyze it, and it is therefore easier to use a digital camera. Smart cameras can analyze the video directly, which reduce the computational power needed by the computer that receives the signals. The analyze can for example detect that something has moved in the video or detect if the vehicle is about to switch lane [11], [12].

2.5.6 Costs

A cost comparison study of analog and digital video surveillance done in 2010 found that digital IP cameras cost roughly 50% more than analog cameras [2]. The price of digital cameras has dropped since then, making them cost the same or even lower than analog cameras. One of the camera manufacturers developing cameras for the off-highway vehicle industry stated that their digital cameras, when released, would cost less than their analog cameras [31].

3 Implementation

3.1 Overview of the system

A digital IP video camera was connected to the display computer CCpilot VA. The video feed was first displayed directly with GStreamer and then with Qt, which uses GStreamer to play video. The CPU load, frame rate and latency was measured for different GStreamer and camera settings. The camera settings were changed through UDP communication as defined in the ISO 17215 standard. The Qt multimedia module was modified to achieve good video performance, but the attempts to simultaneously display both a good quality video feed and a GUI were not successful. An overview of the system can be seen in fig. 3.1.

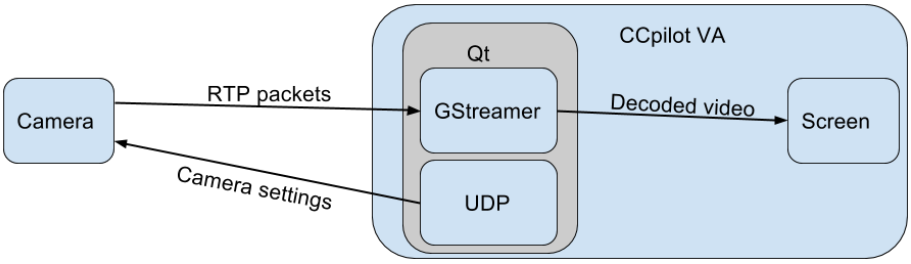

Figure 3.1: An overview of the system. RTP packets and camera settings are sent over an Ethernet cable and the video is decoded by GStreamer and displayed on the screen.

The two digital IP video cameras used in the thesis and the CCpilot VA are described in section 3.2. Section 3.3 gives an introduction to GStreamer, presents an example GStreamer pipeline and concludes with an introduction to Qt. The setups of the cameras, the setup of the VA and three different GStreamer commands to display a video feed from the camera on the VA is presented in section 3.4. Section 3.5 describes how the CPU load, frame rate and latency of the cameras were tested. An implementation to change some camera parameters using the ISO 17215 standard is presented in section 3.6 together with an explanation on how to use it. The chapter ends with section 3.7, which describes how Qt multimedia was changed to obtain a good quality video feed and the attempts to simultaneously display both a good quality video feed and a GUI created by Qt.

3.2 Digital cameras and CCpilot VA

3.2.1 Digital cameras

Two IP camera prototypes from different manufacturers have been used in the thesis. Both are in the developing process and do currently only support

MJPEG video encoding, but both manufacturers are developing support for H.264 encoding. The ISO 17215 standard is supported by both cameras and it is thus possible to configure some of the camera parameters, but some of the standard's functions are not available on both cameras. It is, for example, only possible to change the resolution of one of the cameras, although it probably also will be possible to change it in the final version of the other camera. The camera with fixed resolution has an output resolution of 1280x720 pixels while the other has a maximum output resolution of 1280x960 pixels.

3.2.2 The display computer CCpilot VA

The Crosscontrol CCpilot is a range of display computers in different sizes and materials. A few different hardware setups exist, but most of the devices in the CCpilot range have the processor i.MX 537 and these devices are often used in industry sectors were a live camera feed would be appreciated. The CCpilot VA is one of the devices using the i.MX 537 and, because a VA unit was available at the Uppsala office, it was chosen as the device to be used in this master thesis.

The CCpilot VA, which can be seen in fig. 3.2, is a display computer running Linux with the Freescale i.MX 537 ARM processor and a 7" WVGA, 800 x 480 pixels, resolution full-color display. It has multiple interfaces, including USB, Ethernet, CAN and analog video input, and 10 configurable soft keys. The model used also featured a touch screen. The computer is designed to be reliable and robust even in tough environments and comply with a broad range of regulatory requirements and is IP66 and IP67 classified.

Figure 3.2: The CCpilot VA.

3.2.2.1 The hardware of the VA

The main component of the VA is the Freescale System-on-Chip i.MX 537, which is an "advanced multimedia and power-efficient implementation of the ARM Cortex -A8 core". The core processing speed is up to 800 MHz and it has dedicated hardware accelerators, such as VPU and IPU, to enhance the multimedia capabilities. The i.MX 537 is however an industrial grade System-on-Chip first launched in 2011 and is thus not powerful by today's standards [13]. The VA has 512 MB DDR3 RAM memory and 512 MB industrial grade SLC NAND flash for data storage.

3.2.2.2 The software of the VA

Freescale has released a number of packages with drivers and Linux kernels for the i.MX 537, but since it is getting old and considered to be a good and stable release, they have stopped releasing new packages. The VA is running the latest released package, which contains Linux with kernel version 2.6.35.3 and has default interfaces for accessing most of the hardware, such as Ethernet and USB. The CCaux API, created by Crosscontrol, provides access to device specific hardware, for instance controlling the backlight settings of the display or reading pushbuttons. There is no windowing system, such as X Window System, that creates the GUI, but the applications must write directly to the framebuffer, which is then drawn to the display. The framebuffer is the memory that holds the pixels that should be displayed and by writing directly to it the overhead cost

from using a windowing system is omitted. The VA is equipped with GStreamer 0.10.36 and has most of the plugins in plugins-good and fsl-plugin version 2.0.3, including mfw_vpudecoder and mfw_isink, installed. The GUI as well as the backend of the application is created in Qt 5.6 and the custom UDP class is created in Qt 5.5.

3.3 Software

3.3.1 GStreamer

"GStreamer is a library for constructing graphs of media-handling components" [14]. It can, as in this project, be used for playback of streaming video. GStreamer 0.10 has been used instead of any newer version because of limitations in the hardware and operating system of the target system [37]. GStreamer is very flexible due to its pipeline design, where a chain of elements, each with its own specific task, are linked together. A high level illustration of a pipeline, where some details have been omitted for clarity, can be seen in fig. 3.3. Filesrc, Avidemux, Jpegdec and Autovideosink are separate elements that are linked together with pads in a pipeline. What can be sent through the pads are decided by its capabilities, which are called caps, and only two elements whose pads are matching can be linked together.

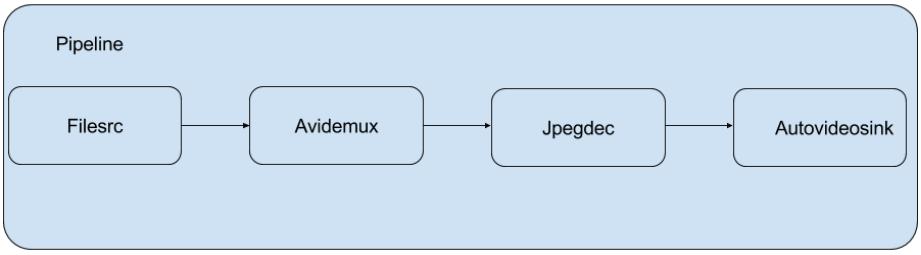

Figure 3.3: A high level example of a GStreamer pipeline.

The command line syntax for playing a media file with GStreamer, and thus creating a pipeline, starts with gst-launch. Then the elements are listed in the same order as they will appear in the pipeline, with the element that finds the media first and the element that exposes the media last. Some elements have specific properties that must be set, such as where the media that the first element will find is located. It is also sometimes necessary to specify the caps in the pipeline, and this must be done between the elements whose caps are set. The command to play a file with the pipeline in fig. 3.3 can be seen in eq. 1.

```
gst-launch filesrc location= /opt/toy_plane_liftoff.avi !
video/x-msvideo ! avidemux name =demux demux.video_00 !          (1)
jpegdec! autovideosink
```

In the pipeline, the ! are the pads, filesrc, avidemux , jpegdec and autovideosink are elements and video/x-msvideo is a cap. Filesrc tells GStreamer to read a local file at the location specified with the property "location", while the next element, avidemux, is a demultiplexer that divides the incoming avi stream into one video stream and one audio stream. The incoming stream is named "demux" with the "name" property and the video part of "demux" is sent to the pad. The video part of the file was known to be compressed as MJPEG and therefore jpegdec was used, as that element decodes JPEG images. The decoded images are sent to autovideosink where they are displayed as a video. Note that most capabilities are not explicitly set in the pipeline. For example, jpegdec and autovideosink does automatically negotiate the caps and does in this example set them to "video/x-raw-yuv". GStreamer pipelines can either be created from the command line or in a C/C++ file [15].

3.3.1.1 GStreamer elements

The most important object class in GStreamer is the element. As seen in the presented pipeline in eq. 1, an element have a specific purpose and is linked to its neighboring elements. The purpose can for example be to load a file, encode, decode, divide or combine streams, payload into packets, extract from packets, apply a filter, convert from one colorspace to another or present a stream on a screen. The elements are linked together to create a pipeline that can perform a specific task, such as playing a local file or saving an incoming stream to a file. A GStreamer plugin adds one or multiple elements that can be used in the pipeline, for instance, the JPeg plugin adds jpegdec and jpegenc for decoding and encoding JPEG, respectively. Multiple elements are connected to perform a more complex task, such as reading a media file, separate the video stream and the audio stream, decode the video stream and display the decoded stream on a sink, as seen in eq. 1.

3.3.1.2 GStreamer pads

The elements are connected to each other by their pads. A pad can either be a source, which is an output, or a sink, which is an input. A source of one element must be connected to a sink on its neighbor element, although an element might have multiple sources or sinks and thus must be connected to multiple neighbor elements. All elements, except the source and sink elements, have both source and sink pads, but the source element does only have source pads and the sink element does only have sink pads. Each pad has specific capabilities, that is, can only handle specific data types. The capabilities, or caps, of the source pad must match those of the sink pad, if the elements are to be connected. A decoded audio stream can not be connected with a videosink and neither can a H.264 video stream be connected to a JPEG decoder, as the caps of the JPEG decoder's source pad is limited to a JPEG stream. GStreamer can often negotiate the caps automatically and thus connect two elements, but sometimes the caps of a pad must be set explicitly in order to connect the elements.

3.3.1.3 GStreamer pipelines

Multiple connected elements can be combined into a bin, which performs a more complex task than the individual elements do themselves. A JPEG decoder and a videosink can be combined into a decode and display bin, which mostly acts a sink element, but receives a JPEG stream instead of the decoded stream. Messages from the children elements are forwarded to the bin and and thus it is possible to query the bin instead of the individual elements. State messages, such as ready, paused and playing, are forwarded from the bin to all the children elements. A pipeline is a top-level bin, to which all elements and bins must be added, that manage synchronization and state updates for all its children. A GStreamer pipeline can be graphically represented as in fig. 3.4.

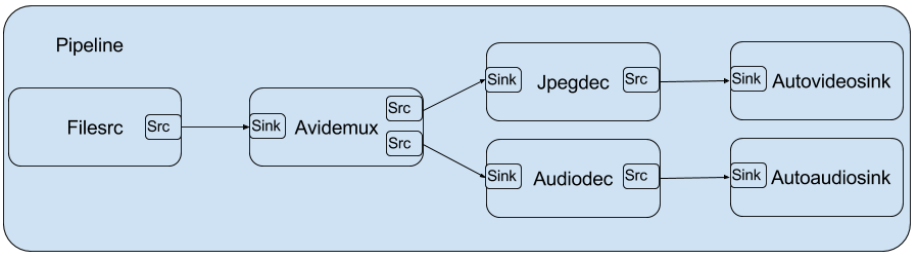

Figure 3.4: An illustrative example of a GStreamer pipeline to play a video file.

3.3.1.4 GStreamer in C/C++

The GStreamer library must be included and initialized before its functions can be used in a C/C++ application. The pipeline concept, including the pieces used, is the same as when running GStreamer from the command line, but the creation of the pipeline requires more details and some extra steps. All elements have to be created, which can be done with the gst_element_factory_make() function. It takes the type of element and the name the element will get as the input parameters, for example gst_element_factory_make("avidemux", "demux") will create a avidemux element with the name demux.

When using elements, it is often not necessary to add and link pads one-to-one manually, because the elements can usually take care of this themselves. It might, however, be necessary to set the caps of some pads manually, which can be done by creating a capsfilter element, setting its caps property and add the element to the pipeline. The input to the caps property can be created with the gst_caps_new_simple() function and can for example be set to "video/x-msvideo" with gst_caps_new_simple("video/x-msvideo", NULL).

A property of an element is set with the g_object_set() function, which takes the element, the name of the property, the value of the property and NULL as input parameters. The location parameter of the filesrc element is thus set to /opt/toy_plane_liftoff.avi with g_object_set(filesrc, "location", "/opt/toy_plane_liftoff.avi", null). The last step is to create the pipeline, add

all elements to it and link the elements.

A pipeline can be created with the gst_pipeline_new() function and be given a name through the input parameter, as in gst_pipeline_new("name-of-the-pipeline"). The elements are most easily added to the pipeline with the gst_bin_add_many() function and then linked to each other with the gst_element_link_many() function, which however is unable to link elements that can have multiple source or sink pads. The avidemux element can have both a video and an audio source pad and does thus require a slightly more complicated linking function. The elements before and including the avidemux element is linked together with each other in one group and the elements after avidemux is linked together in another group.

The function g_signal_connect links the two groups of elements together at run time by finding out which of avidemux's source pads that are compatible with the next element's sink pad. In this case the next element is a video decoder element and thus the video source pad of avidemux is linked to the sink pad of the decoder element. Finally the pipeline's state is set to GST_STATE_PLAYING to start playing the video [15]. The pipeline in eq. 1 is created in C++ in eq. 2.

```
#include <gst/gst.h>
static void on_pad_added(GstElement *element, GstPad *pad, gpointer data)
{
GstPad *sinkpad;
GstElement *decoder = (GstElement *) data;
sinkpad = gst_element_get_static_pad (decoder, "sink");
gst_pad_link (pad, sinkpad);
gst_object_unref(sinkpad);
}
int main (int argc, char *argv[ ])
{
GstElement *pipeline, *src, *filter, *mux, *dec, *sink;
GstCaps *filtercaps;
gst_init (&argc, &argv);
pipeline = gst_pipeline_new("pipeline-name");
src = gst_element_factory_make("filesrc", "source");
filter = gst_element_factory_make("capsfilter", "filter");
mux = gst_element_factory_make("avidemux", "demultiplexer");
dec = gst_element_factory_make("jpegdec", "decoder");
sink = gst_element_factory_make("autovideosink", "sink");
g_object_set(src, "location", "/opt/toy_plane_liftoff.avi", NULL);
filtercaps = gst_caps_new_simple("video/x-msvideo", NULL);
g_object_set(filter, "caps", filtercaps, NULL);
gst_caps_unref(filtercaps);
gst_bin_add_many(GST_BIN (pipeline), src, filter, mux, dec, sink, NULL);
gst_element_link_many(src, filter, mux, NULL);
gst_element_link(dec, sink);
g_signal_connect (mux, "pad-added", G_CALLBACK (on_pad_added), dec);
gst_element_set_state (pipeline, GST_STATE_PLAYING);
}
```

(2)

3.3.2 Qt

"Qt is a cross-platform application development framework for desktop, embedded and mobile" [16]. Because of this, the application can be released natively for different platforms without rewriting the source code, which reduces developing

time and eases code maintenance. Qt is the industry standard when developing for embedded Linux, partly because it in many cases is free to use, and as many of Crosscontrol's customers are using it, Crosscontrol want to use it as well [37]. GUIs can either be written in C++ using the Qt Widgets module or in QML using the QtQuick module. "QML is a user interface specification and programming language" designed by the Qt Project to be a highly readable language enabling "components to be interconnected in a dynamic manner" and has a syntax that resembles CSS, HTML and JavaScript [17]. Components can be created or changed dynamically in response to user input, as can be seen in the QML in eq. 3. In the example a red rectangle is created, that changes color to blue when clicked, and with the centered text "Hello, World!".

```
Rectangle {
  width: 200
  height: 100
  color: "red"
  Text {
    anchors.centerIn: parent
    text: "Hello, World!"                                    (3)
  }
 MouseArea {
    anchors.fill: parent
    onClicked: parent.color = "blue"
  }
}
```

3.4 Setups

3.4.1 Setting up the cameras

The first test was done with an IP camera prototype from Orlaco. It was connected with an Ethernet cable to a private network and powered up. By default the camera was set to broadcast the video stream on port 50004 to all available IP addresses using RTP running over UDP as the network protocol and MJPEG as the compression type. The camera feed could thus be viewed by all computers on the same private network by listening to port 50004 and interpreting the stream the right way. The second camera prototype, that was from Mekra Lang, was connected in the same way and had the same parameters, except for the video stream port that was set to 5004.

3.4.2 Setting up the VA

The VA was connected to power and the private network through an ethernet cable. It starts automatically on power up and broadcasts it's Serial number, Firmware version, VA device type and IP address, but no graphical application was configured to autostart and thus only a black screen was displayed. Crosscontrol has created the LinX software suite, which is a virtual machine running Kubuntu Linux [32]. The image is set up to be ready for developing for the CCpilot line by having the Integrated Development Environment (IDE) Qt creator installed and the relevant libraries linked to the building software. The VA is accessed through the network protocol Secure Shell (SSH) and can thus be controlled from the virtual machine.

3.4.3 The GStreamer commands

The camera documentation proposed a GStreamer command, which can be seen in eq. 4, that only uses standard plugins, because it is supposed to work on all platforms that have GStreamer. These plugins were already installed on the VA and running the command started a full screen video playback of the camera stream. The received video had good resolution and low latency, but low frame rate and high CPU load. This is because the command did not utilize the hardware decoding capabilities of the i.MX 537, but instead used software decoding done in the CPU.

```
gst-launch udpsrc port=50004 ! application/x-rtp,
encoding-name=JPEG, payload=26 ! rtpjpegdepay !          (4)
jpegdec ! autovideosink
```

Udpsrc port 50004 tells GStreamer to read UDP packets on port 50004 while the next element, rtpjpegdepay, extracts JPEG video from RTP packets. Since rtpjpegdepay can not handle all UDP packets, but only those both using the RTP network protocol and containing JPEGs, the pads are not directly matching. The first element needs caps to restrict the UDP packets to only those sent using the RTP protocol that contains JPEGs. The encoding name and payload for different formats sent using the RTP protocol can be found on the Internet Engineering Task Force's web site [18]. JPegdec decodes the stream of jpges and autovideosink present the stream of decoded jpegs on the sink that GStreamer finds most suitable [15].

3.4.3.1 Enabling hardware decoding in GStreamer

The pipeline design of GStreamer makes it easy to change one part of a pipeline while keeping the other parts. The jpegdec element is designed to do a software decoding of the JPEGs and is thus needed to be exchanged to a decoder element that utilizes hardware decoding. The decoder element must be designed for the specific hardware being used, in this case the Freescale i.MX 537, because

hardware decoding is hardware specific. Freescale provides an API for utilizing hardware decoding, which can be used for creating a custom GStreamer decoder element, but they have also released a complete GStreamer plugin that contains the hardware decoder element mfw_vpudecoder for the i.MX 537. A pipeline using the mfw_vpudecoder element can be found in eq. 5, where the property codec-type=7 of the mfw_vpudecoder element sets the codec to MJPEG.

```
gst-launch udpsrc port=50004 ! application/x-rtp,
encoding-name=JPEG, payload=26 ! rtpjpegdepay !          (5)
mfw_vpudecoder codec-type=7 ! autovideosink
```

3.4.3.2 Zero-copy between the decoder and the sink

The CPU load might still be high, even if a hardware decoder is being used. This happens if the decoded frames are stored to a local memory buffer by the VPU, then copied to another memory buffer by the CPU and then again copied by the CPU to a memory buffer used by the sink element. The solution is called zero-copy and is achieved by using Direct Memory Access, which prevents the CPU from copying the buffer by granting access for a hardware subsystem to the buffer [36]. Freescale provides the sink elements mfw_v4lsink and mfw_isink that both are able to present the decoded MJPG stream on the screen directly from the buffer used by the VPU and thus eliminating the need to copy the buffer. The pipeline in eq. 6 is utilizing both hardware decoding and a zero-copy link from the decoder to the sink. This resulted in higher frame rate and lower CPU usage when using either of the cameras, but the camera from Mekra Lang had both higher frame and lower CPU usage. Frame rate and CPU usage for the two cameras can be seen in chapter 4.2.

```
gst-launch udpsrc port=50004 ! application/x-rtp,
encoding-name=JPEG, payload=26 ! rtpjpegdepay !          (6)
mfw_vpudecoder codec-type=7 ! mfw_v4lsink
```

3.5 Testing the CPU load, frame rate and latency

The different aspects of the video performance was tested on the VA using the methods presented in this chapter. The CPU load was measured by running a GStreamer command and then using the Linux *top* command, which show all processes as well as their CPU load. The GStreamer command was running for one minute and the minimum and maximum CPU load for the GStreamer process during that time was used as the limits for the CPU load. The mfw_v4lsink element displays the average frame rate when the command is aborted and this value after one minute's playback was used as the frame rate. It was not important to get the exact latency of the camera and therefore a simple setup was used, where the camera was filming a stopwatch on a computer

screen and displaying the video feed on a VA. Another camera was filming both the computer screen and the VA screen and by comparing the difference between the stop watch on the computer screen and on the VA screen, the latency was determined. A fast 60 frames per second camera was used to film both screens and the result of the setup can seen in fig. 3.5. The command used for testing with hardware decoder can be found in eq. 6, with the port property set according to the camera being tested. The same command, but with mfw_vpudecoder replaced with jpegdec, was used to test with software decoder.

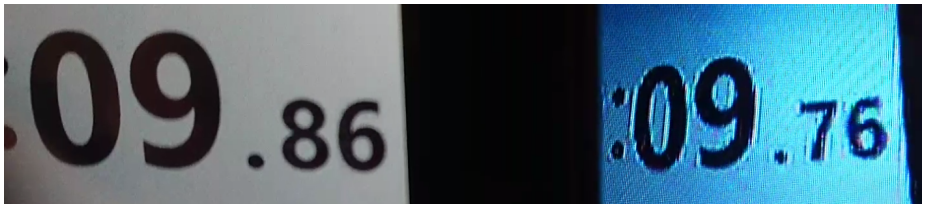

Figure 3.5: A computer screen showing a stopwatch was filmed with the camera to be tested and the feed was displayed on the VA. A second high speed camera was then used to film both the computer screen and the VA screen, and by calculating the difference between the stopwatch on the computer and the VA, the latency was determined.

3.6 Communicating with the camera

The ISO 17215 standard specifies how to communicate with cameras used in road vehicles and defines the API. The camera functions are sent using the SOME/IP protocol, which in turn is the payload of the UDP packets. Qt provides a UDP socket through the QUdpSocket class that manage the UDP header, including length and checksum calculations, as well as the underlying IPv4 and Ethernet protocol headers. This makes it easier to send UDP packets as it removes the task of manually calculating headers.

3.6.1 Creating a custom UDP class in Qt

After reading a tutorial, a custom UDP class called MyUDP, was created in Qt to communicate with the camera [19]. The constructor of the class takes the IP address of the camera as a QHostAddress and the port number of the camera as an integer as input arguments. It stores the QHostAddress as qAddress and the port number as a sendPort and later uses both to send UDP messages to the camera. The constructor then creates an instance of QUdpSocket and binds the socket to a local IPv4 address, also in the form a QHostAdress. The socket is connected to the instance of MyUDP, which enables the socket to forward an incoming message to the instance of MyUDP. The incoming UDP message is received and interpreted by the custom function readyRead.

When the QUdpSocket is bound and connected, it can send UDP datagrams, which is done by creating the payload and specifying the IP address and port number of the receiving camera. The payload is defined as a QByteArray, which is a convenient ByteArray class available in Qt, and the IP address and port number are both given by the camera manufacturers. The manufacturers also specify the possible values for the different parameters, such as input region or output resolution, in the form of bit values for the ISO 17215 functions in the payload. Since each ISO 17215 function needs the UDP datagram to be different, each ISO 17215 function needs an own function in MyUDP to set the payload correctly.

The values that are to be set with a ISO 17215 function is passed as arguments to the corresponding function in MyUDP. As an example, the ISO 17215 function SetRegionOfInterest sets the parameters for a specified Region of Interest. A Region of Interest is, as stated in chapter 2.4.3, a complete video setting with defined video parameters and each camera can store a specific number of these complete video settings. The number of settings as well as the possible values for the video parameters are specified by the camera manufacturer. The corresponding function created in MyUDP is called SetROI and takes the index of the Region of Interest to be changed as an integer and the video parameters as a char array as input arguments. The chars must be sent as hexadecimal values in the order specified in ISO 17215. The input arguments are combined with the unchangeable parameters of SetRegionOfInterest into a QByteArray with the order specified in ISO 17215. The QByteArray is sent with the Qt function writeDatagram by the bound and connected QUdpSocket by specifying the IP address and port of the camera. An example C++ application to change the camera resolution can be found in appendix A.

3.6.2 Using the custom UDP class in Qt

The MyUDP class is used by creating an instance of MyUDP with the parameters set to the camera to be communicated with. The instance can then use the functions defined in the MyUDP class to query or change the settings of the camera, for example the camera resolution. The cameras response, which can be the value of the setting, a success message or an error message, is presented to the user. The next step is to integrate the communication to a GUI, which will enable the user to use the functions in MyUDP by pressing the corresponding button.

3.7 Showing video in Qt

To show video in Qt is, in specific cases, very easy. The main C++ file needs to include the necessary Q headers, in this case the QApplication, QMainWindow, QQmlApplicationEngine and QtMultimedia headers, initiate the QApplication and QQmlApplicationEngine and finally load the QML file. The QML file defines the GUI, including the existing objects and their respective size, color and properties. The Video object is a combination of the MediaPlayer object

and VideoOutput object, that simplifies video playback by being able to both decode and display video. Finally the display size, source file and play parameter of the video object is set. The project is built and deployed, which, in specific cases, will display the video on the device.

3.7.1 Properties of QtMultimedia

The Video object is a part of the QtMultimedia module. A key component of the Qt Multimedia module is the QGstreamerPlayerSession, which creates the GStreamer pipeline and provides functions used by higher level components to, for instance, load a video file or set the playing state. The pipeline is fixed to use the "stand-alone everything-in-one abstraction for an audio and/or video player" element playbin2 [20]. The element tries to find and use the best available pipeline for each incoming video stream, but it does have some shortcomings and does only allow minor modifications of the pipeline used. It is possible to specify the sink to be used, and this is set by QGstreamerPlayerSession to be QVideoSurfaceGstSink.

3.7.2 The shortcomings of the playbin2 element

It is not possible to play a RTP stream with playbin2, because the RTP stream requires configuration settings not available in the stream itself [21]. These settings have to be passed as caps, as is done in eq. 5, but it is not possible to pass these caps to playbin2. Since QtMultimedia uses playbin2, it is not possible to play RTP streams with the Video object or any other QtMultimedia object in Qt. It is also not possible to specify which decoder that should be used, which makes it impossible to choose a hardware decoder. Using the Video object in Qt might thus result in high CPU load and low frame rate. The fixed pipeline also prevents more elaborate pipelines from being used, which may be required in real applications, such as simultaneously stream the video on a screen and record it to a file. The Video object was tested by playing a locally stored video file, which worked but with high CPU load and low frame rate.

3.7.3 Changing QGstreamerPlayerSession to use a custom pipeline

The solution to overcome the shortcomings of the playbin2 element is to not use it, but instead create a custom GStreamer pipeline. QGstreamerPlayerSession is written in C++ and the pipeline can thus be created with the same methods as proposed in chapter 3.3.1.4 and be the same as the pipeline presented in chapter 3.4.3.2. The needed elements, which are udpsrc, rtpjpegdepay, mfw_vpudecoder, mfw_isink and pipeline, are created in the constructor of QGstreamerPlayerSession. The loadFromUri function was modified to accept a port number from the QML Video object's property *source*, which is an url QML basic type. The loadFromUri function extracts the port number as an integer from the value of the url type. This is not following good coding standards and should be resolved by adding a *port* property to the Video object, but since this is only a proof

of concept it was enough to be able to set the port from the QML. Lastly, the play function was modified to set the necessary caps, add the elements to the pipeline and link the elements.

3.7.4 Problems with the mfw_isink using Qt

The modified QGstreamerPlayerSession was built and transferred to the VA unit and a simple setup containing a background color and a small Video object was created in Qt. When the Qt project was built and deployed to the VA, the background color was first shown together with the empty Video video object, as seen in fig. 3.6. Then the video started, but in full screen and not in the Video object's box. This was undesired, because the video was covering the GUI and thus making the GUI useless.

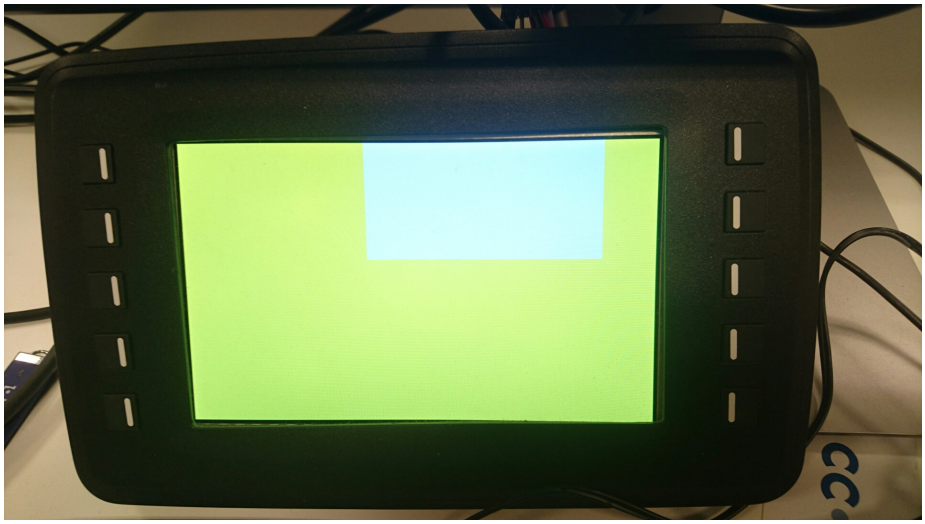

Figure 3.6: A GUI with a yellow background color and a white box. which is an empty Video object.

The mfw_isink has properties to set the size and position of the video display. In the next attempt, these properties were set to roughly match the size and position of the Video object box. Again, this did not yield the desired result, as seen in fig. 3.7. Even though the video display roughly was in the right position and had the right size as the Video object's box, the video was still blocking the GUI by adding a black background around the video stream. If the VA would have had a windowing system, mfw_isink could have been configured to write to an existing window, which in this case would have been the window created by Qt. Since a windowing system is lacking, mfw_isink has to write directly to the framebuffer and does, in a way, create a full screen window by writing to the whole framebuffer.

Figure 3.7: The video stream with the size and position parameters set is adding a black background around the video stream and thus blocking the GUI.

3.7.5 Using the QtMultimedia default video sink

Qt Multimedia uses QVideoSurfaceGstSink by default. The pipeline was changed to use this sink element, but otherwise the same pipeline as before, inlcuding the mfw_vpudecoder element. This yielded the desired result, as seen if fig. 3.8, but not the desired performance. The frame rate was low and the CPU load high, because the link between mfw_vpudecoder and QVideoSurfaceGstSink was not zero-copy. The best solution would be to link mfw_isink to Qt and thus make the video from the sink appear in the Video object, which would enable straightforward QML manipulation of the Video object, including size, positioning and text overlays.

Figure 3.8: Using the default Qt video sink yields the desired result, showing both the GUI and video object that can be manipulated directly from QML, but with bad video performance.

An easy way to achieve this would be to make mfw_isink use the existing surface, or window, created by Qt instead of creating it's own surface. This was however not easily done, because mfw_isink uses video_surface.c to create a surface upon which the video is drawn, while QVideoSurfaceGstSink uses QVideoSurface.cpp. Therefore mfw_isink would need to be adapted to use another surface element, the QVideoSurface, which might be possible to do, but it would be time consuming and was deemed to be outside the scope of this thesis. Another possible solution was to make QVideoSurfaceGstSink use the decoded video stream from mfw_vpudecoder directly, without using the CPU to copy it to the default memory used by QVideoSurfaceGstSink, which would make the link zero-copy. To accomplish this was also deemed outside the scope, as it requires thorough knowledge of Linux's memory management as well as a good understanding of the specific hardware being used, the mfw_vpudecoder element and the QVideoSurfaceGstSink.

A third option, that would lack the possibility to manipulate the video directly through QML, was to make mfw_isink only write the actual video content to the framebuffer and not add the black background. While it might be possible to remove the background, or at least make it transparent, it would require changes of some classes used by mfw_isink, probably including the video_surface class. Since the GUI and GStreamer would be writing to the same framebuffer at the same time, flickering of the screen might occur and therefore a fourth option was chosen. Instead of making the two applications write to the same framebuffer, it would be better to make them write to their own framebuffer. The i.MX 537 supports two separate framebuffers and it

should thus be possible to make the GUI write to the background framebuffer and GStreamer to write to foreground, or overlay, framebuffer. The changes made to QGstreamerPlayerSession can be found in appendix B.

3.7.6 Trying to enable a second framebuffer

The i.MX 537 does have two separate framebuffers and both should be switched on by default. Listing the framebuffers under /dev showed fb0, fb1, fb2 and fb3, while there in /sys/class/graphics existed fb0, fb1 and fbcon. The fbs under /dev are the actual device files for the graphics hardware and the fbs under /sys/class/graphics "provides a means to export kernel data structures, their attributes, and the linkages between them to userspace" [22]. It is thus possible to get and set settings of the hardware devices under /dev by using the corresponding entity under /sys. Executing the command "fbset -fb /dev/fb0" presented the status of framebuffer 0, which also can be found by the command "cat /sys/class/graphics/fb0/mode". "fbset -fb /dev/fb1" or fb2 did however both yield "device not found", which indicates that those devices are not properly initialized.

The Linux department at Crosscontrol, who have made the custom Linux image used by the VA, suggested to try to enable the second framebuffer through changing the boot arguments. This was done by connecting a VA debug card to both the VA and another computer, which enabled the booting to be stopped and the boot arguments to be changed. The bootargs contains a video setting, that initially was set to video=mxcdi0fb:RGB24,800x480@60_VA. It was found at the Freescale internet forum that it could be set twice to enable a second framebuffer, as in video=mxcdi0fb:RGB24,800x480@60_VA,bpp=32 video=mxcdi2fb:RGB24,800x480@60_VA,bpp=32. This was also tried with video=mxcdi1fb while keeping the other settings the same, but this did not enable the second framebuffer either.

4 Results

4.1 Digital cameras are better than analog cameras

Digital cameras have been found to be better than analog cameras through a literature review. The digital cameras have better video quality and security options, the video is easier to transmit, store and automatically analyze and the cameras cost less than their analog counterparts. Digital and analog cameras have different reliability characteristics, where the video from digital cameras either is shown without any noise or not shown at all and video from analog cameras gets more and more noisy as the error rate increases.

4.2 The VA can satisfactory connect to a digital camera

It is possible on the VA to show a live digital camera feed with high quality, low latency and high frame rate without putting too much load on the CPU. It was found that the camera from Mekra Land loaded the CPU much less than the camera from Orlaco and that the CPU load decreased with decreased video resolution. The frame rate was high when hardware decoding was utilized and the latency was always low, even when software decoding was utilized. The frame rate and CPU load for different camera video output resolution and the two cameras when using both hardware and software decoding can be found in table 4.1. The latency was less than 10 ms in all tests, no matter which camera, type of decoding or video resolution that was used.

Table 4.1: The Frames Per Second and CPU load on the VA for different camera output resolutions and cameras with hardware decoding and software decoding, respectively. The GStreamer command used was: *gst-launch udpsrc port=50004 ! application/x-rtp, encoding-name=JPEG,payload=26 ! rtpjpegdepay ! mfw_ vpudecoder ! mfw_ v4lsink* for the hardware decoded values and the same except for that *mfw_ vpudecoder* was replaced with *jpegdec* for the software decoded values.

	With HW decode		With SW decode	
Resolution	FPS	CPU load(%)	FPS	CPU load(%)
Orlaco				
640x480	26	17-28	9	44-50
800x480	26	31-40	9	58 -64
800x600	26	32-40	7	59 -64
1024x768	25	58-65	5	81 -87
1280x720	19	60-78	4	87- 91
Mekra Lang				
1280x720	26	16-27	7	62-66

4.3 Partially integrated camera feed in Qt

The final goal of this thesis was to integrate the camera feed into Qt, which only partially was achieved. The file QGstreamerPlayerSession.cpp was modified to programmatically create a custom GStreamer pipeline, and that file is used by QtMultimedia when creating a Video object in QML. The video stream is thus started by Qt and by changing a few more functions in QGstreamerPlayerSession.cpp it would be possible to stop the camera feed and change to another camera feed. It is also possible to change camera parameters, such as video output resolution, through Qt, but no GUI has been made to simplify the change of camera parameters.

The camera feed can be almost entirely integrated into Qt by using the QVIdeoSurfaceGstSink instead of a Freescale sink. This makes the camera feed appear in the Video object, which can be altered directly through QML, but using the QVIdeoSurfaceGstSink gives bad video performance and high CPU load because the link from the Freescale hardware decoder to QVideoSurfaceGstSink requires copying buffers through the CPU. This solution would require some more changes in QGstreamerPlayerSession.cpp to be entirely integrated into Qt, because some of the functions in QGstreamerPlayerSession.cpp uses methods only available on a playbin element and that element has been replaced by a custom pipeline.

A Freescale sink must be used to get good video performance and low CPU

load, as the Freescale sinks have a zero-copy link to Freescale's hardware decoder elements. These sinks do, however, take over the whole screen even if the camera feed itself is not full screen, as seen in fig. 4.1. This happens because the VA lacks a windowing system and the sink must therefore write directly to the framebuffer, and the sink is not designed to write to only a part of the framebuffer.

Figure 4.1: The video stream with the size and position parameters set is taking over the screen by adding a black background around the video stream and thus blocking the GUI.

The attempt to enable a second framebuffer, which would have made it possible to separate the GUI and the video stream into different framebuffers, failed. It was therefore not possible to display both the video feed and a Qt GUI at the same time. Since mfw_isink supports overlay on itself, it was possible to add images on top of the video stream, which did not increase the CPU load noticeably. It was therefore possible to create a simple GUI with images of buttons on top of the video, as seen in fig 4.2, and it should be possible to link the images of buttons to real buttons in Qt, but this was not investigated. A button overlay was achieved by first running the GStreamer video command seen in eq. 7, which creates a video feed on a video_surface, and then running another GStreamer command seen in eq 8, that creates the image on the already created video_surface. The freeze element freezes the image on the screen, as an image otherwise would be displayed on only one frame and then disappear. To get multiple distinct buttons, with video feed between them, the second command can be repeated with different image files and settings on the mfw_isink element. It is also possible to create an image the contains multiple buttons and thus add multiple buttons with only one command, but the video feed can then not be seen between the buttons.

Figure 4.2: A small button is overlayed in the top left corner on top of the video feed.

```
gst-launch -v udpsrc port=5004 ! application/x-rtp,
encoding-name=JPEG, payload=26 ! rtpjpegdepay !                    (7)
mfw_vpudecoder codec-type=7 ! mfw_isink
```

```
gst-launch -v filesrc location= /opt/rec.jpg !
queue ! jpegdec ! freeze ! mfw_isink                               (8)
disp-width=80 disp-height=48 sync=false
```

5 Conclusions and suggestions for further work

The first two goals of the thesis, to review the differences between analog and digital cameras and to investigate if it was possible to satisfactory connect digital cameras to the VA, were achieved. The final goal, that was to implement the video feed as a part of the GUI created by Qt, was partially achieved. The camera feed was partially integrated into Qt and could be started by Qt, but it was not possible to view the camera feed and the GUI simultaneously.

5.1 Reviewing the differences between analog and digital cameras

There is much literature on the subject of analog and digital cameras, but since the digital cameras have improved drastically over the years, the challenge was to find relevant and updated information. The general camera information in chapter 2 have been gathered from this literature. Digital cameras for the off-highway vehicle industry does only exist in the form of prototypes, and there is therefore no literature on this specific kind of digital cameras. Instead literature from the closed-circuit television, or video surveillance, industry have been used for industry specific camera information, as the cameras used in that industry are fairly similar to those that will be used in the off-highway vehicle industry. The closed-circuit television industry begun the switch from analog cameras to digital cameras a few years ago and there exists some literature on the industry specific difference between analog and digital cameras.

5.2 Satisfactory connecting the digital camera to the VA

The digital cameras used were prototypes and their documentation was in pre-release state. Most of the information in the documentation lacked explanations and it was therefore necessary to either reverse-engineer the settings by trying different values or to try to find explanations elsewhere. The GStreamer command was, for example, given without any comments or descriptions, and it was thus needed to find more information about GStreamer elsewhere. The creators of GStreamer are providing extensive documentation, including tutorials and examples, and there is also a large internet forum community answering GStreamer related questions. These two sources were very helpful in understanding and constructing a GStreamer pipeline, and some hardware specific pipeline advice was found on Freescale's internet forum. The GStreamer documentations also included examples of how to transform a command line pipeline into a C++ pipeline.

The reason why the CPU load for the GStreamer process is substantial, even when hardware decoding is used, is that everything up to the decoding element in the pipeline is done by the CPU, including receiving and depayloading the packets. That the CPU load is three times as high when the Orlaco camera is used than when the Mekra Lang camera is used, even if the video resolution is the same, is presumably because the packets sent by the Oralco camera are

payloaded in way that loads the CPU more when depayloading than the packets sent by the Mekra Lang camera are.

The ISO 17215 standard explains in a very abstract manner how to communicate with the camera and no concrete implementation examples of it were found online which might be because the standard is relatively new, with the first edition of it being published in April 2014. The camera manufacturers did however present a few concrete examples, which made it easier to understand the standard and to create own implementations of it. The QUdpSocket class in Qt was very convenient, as it managed the UDP as well as all lower protocols automatically, leaving only the SOME/IP protocol to be set manually. The camera from Oralaco had a high CPU load, even with hardware decoding and a zero-copy link to the sink, at the default video resolution of 1280x720 pixels, but the CPU load was halved when the resolution was changed to 800x480 pixels. This was important, because the original CPU load of up to 78% from GStreamer was too high to be able to have Qt running smoothly simultaneously. It was interesting to find that the camera from Mekra Lang loaded the CPU with only one third of the load from the Orlaco camera at the same video resolution, and that the video stream from the Mekra Lang camera on top of that had a higher frame rate.

5.3 Partially integrating the camera feed into Qt

The QtMultimedia module is a ready made module that is supposed to be able to manage all multimedia content by default, and it is therefore not designed to be manipulated. This is probably the reason why no documentation was found on how to manipulate it or how the different parts of the module is connected to each other and to other parts of Qt. The first step, which was to change the default GStreamer pipeline into a custom one, was relatively straight forward as all required changes could be made in only two files, the QGstreamerPLayerSession.cpp and its header file.

The next step was to examine the possibility of making Freescale's hardware sink appear as a Qt sink and thus being able to control from Qt. As no documentation on how the parts were connected was found, the links had to be reverse-engineered, which was rather time consuming. It was ultimately discovered that the Qt sink draw the video on a QVideoSurface, while the Freescale sinks draw the video on a video_surface. Adapting the Freescale sinks to use QVideoSurface is a possible solution, which would integrate the camera feed into Qt, but it requires knowledge of how GStreamer sinks are created and of how QVideoSurface is constructed. It might therefore be easier to alter QVideoSurfaceGstSink to access the decoded video stream directly from the buffer that mfw_vpudecoder is outputting to, without using the CPU and thus creating a zero-copy link. This solution would also integrate the camera feed into Qt, but it does require knowledge of Linux's memory management, the specific hardware being used and QVideoSurfaceGstSink.

Two options that would not integrate the camera feed into Qt per se, but that would display both the video feed and the Qt GUI simultaneously, are to

make the Freescale sinks write only the actual video content to the framebuffer and to make the sinks write to another framebuffer than Qt is writing to. The first option would require knowledge of how the GStreamer sink is writing to the framebuffer, and it would presumably be very complicated to make the sink not require full screen in an operating system without a windowing system. The second option should have been easy to implement, as having two framebuffers available is the default option on the i.MX 537. It was, however, not possible to write to the second framebuffer, not even with support from those working with Linux questions at Crosscontrol. On the Freescale internet forum, where users get help with Freescale products from both other users as well as Freescale staff, no one had posted a help request with this specific issue. Some users were requesting help with the next steps regarding the use of two framebuffers and were thus apparently using two framebuffers.

5.4 Suggestions for further work

Four different options for completing the final goal of integrating the camera feed into to Qt have been presented in the thesis and the option of using a second framebuffer have been attempted. Although the attempt was not successful, it is presumably the most uncomplicated approach. The possibility of activating the second framebuffer by an ioctl call should be investigated as well as other approaches of activating the framebuffer. When the second framebuffer is activated, the height and width function in QtMultimedia should be adapted to changing the height and width of the video feed framebuffer instead of the then non-existent QVideoSurface. If it is not possible to position the framebuffer directly, the same effect can be achieved by having a full screen framebuffer with transparent borders around the video feed and positioning the feed correctly with respect to the full screen.

Another option that might solve many of the problems is to use newer hardware that supports a newer version of GStreamer. GStreamer version 1.x comes with some windowing capabilities that possibly enables the positing of the sink even in an OS without a windowing system. When searching for solutions to GStreamer related problems on forums, the first suggestion often was to upgrade to GStreamer 1.x, as it deliver multiple improvements over 0.x versions [23]. GStreamer 1.x has been stable since 2012 and 0.x has been unsupported since Mars 2013 [24], [25].

Appendices

A The files needed to communicate with the camera

main.cpp

```
#include <QApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>
#include <QQuickItem>
#include <QtQml>
#include <QFontDatabase>
#include "myudp.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QFontDatabase database;
    int id = database.addApplicationFont("/usr/lib/fonts/DejaVuSans.ttf");
    if (id != -1){
        QStringList families = QFontDatabase::applicationFontFamilies(id);
        QGuiApplication::setFont(QFont(families.at(0)));
    }

    // get the target setting from the .pro-file:
#if defined(TARGET_ARM)
    bool targetARM = true;
#else
    // virtual machine
    bool targetARM = false;
#endif

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/qml/main.qml")));

    // this will set the "targetARM" property to true if the software is built for ARM
    QObject *rootObject = engine.rootObjects().first();
    rootObject->setProperty("targetARM", targetARM);


// 0x00, 0x00 is the first vaule => 0x0000 = p1x
  // input region p1x, p1y, p2x, p2y, output resolution width, height
   char outP800x640[12] ={0x00,0x00, 0x00,0x00, 0x05,0x00, 0x03,0x00, 0x03,0x20, 0x01,0xe0};
   char outP640x480[12] ={0x00,0x00, 0x00,0x00, 0x05,0x00, 0x03,0xc0, 0x02,0x80, 0x01,0xe0};
   char outP800x480[12] ={0x00,0x00, 0x00,0x00, 0x05,0x00, 0x03,0x00, 0x03,0x20, 0x01,0xe0};
   char outP800x600[12] ={0x00,0x00, 0x00,0x00, 0x05,0x00, 0x03,0xc0, 0x03,0x20, 0x02,0x58};
   char outP1024x768[12]={0x00,0x00, 0x00,0x00, 0x05,0x00, 0x03,0xc0, 0x04,0x00, 0x03,0x00};
   char outP1280x720[12]={0x00,0x00, 0x00,0x00, 0x05,0x00, 0x02,0xd0, 0x05,0x00, 0x02,0xd0};
   char outP1280x768[12]={0x00,0x00, 0x00,0x00, 0x05,0x00, 0x03,0x00, 0x05,0x00, 0x03,0x00};
   char outP1280x960[12]={0x00,0x00, 0x00,0x00, 0x05,0x00, 0x03,0xc0, 0x05,0x00, 0x03,0xc0};

    QHostAddress host;
    host.setAddress("10.131.16.147");
   MyUDP client(0,host, 17215);
    client.SetROI(4, outPut1280x720);
  client.ChangeROI(4);
```

```
    return app.exec();
}
```

MyUDP.cpp

```
#include "myudp.h"

MyUDP::MyUDP(QObject *parent, QHostAddress qAddressIn, int sendPortIn) :
    QObject(parent)
{
    qAddress= qAddressIn;
    sendPort = sendPortIn;
    socket = new QUdpSocket(this);
socket->bind(QHostAddress::LocalHost );
    connect(socket, SIGNAL(readyRead()), this, SLOT(readyRead()));

    qDebug() << "Bound: " << (socket->state() == socket->BoundState);
}

bool MyUDP::ChangeROI(int roiIndex)
{
    QByteArray Data;
    char char00= 0x00; // cant directly input 0x00 in Data.append()
    if (roiIndex >-1 && roiIndex <10)  {
        char setRoi = roiIndex;


        Data.append(0x43);Data.append(0x3f);
        Data.append(0x01);Data.append(0x31);
        Data.append(char00);Data.append(char00);Data.append(char00);Data.append(0x0c);
        Data.append(0x05);Data.append(0xd6);
        Data.append(char00);Data.append(0x0e);
        Data.append(0x01);

        Data.append(0x01);
        Data.append(char00);
        Data.append(char00);
        Data.append(char00);Data.append(char00);Data.append(char00);Data.append(setRoi);
        qDebug() << "Message_size: " << Data.size();
        QByteArray text = Data.toHex();
        qDebug() << "Message is: " << text;

         // qaddress = ip of receiving, sendPost = port at receiving
        qint64 bytesSent = socket->writeDatagram(Data, qAddress, sendPort);

        if (-1 == bytesSent) {
            qDebug() << "Message not sent";
            return false;
        }
        else {
            qDebug() << "Socket: " << socket << ". Bytes sent: " << bytesSent;
            return true;
        }
    }
    else {
        return false;
    }
```

```
}

bool MyUDP::SetROI(int roiIndex, char videoSetting[]) {
    QByteArray Data;
    char char00= 0x00; // cant directly input 0x00 in Data.append()
    if (roiIndex >-1 && roiIndex <10  ) {
        // Method: set_cam_exclusive(17) 0x0011
        Data.append(0x43);Data.append(0x3f);
        Data.append(char00);Data.append(0x11);
        Data.append(char00);Data.append(char00);Data.append(char00);Data.append(0x0c);
        Data.append(0x05);Data.append(0xd6);
        Data.append(char00);Data.append(0x0a);

        Data.append(0x01);
        Data.append(0x01);
        Data.append(char00);
        Data.append(char00);
        Data.append(char00);Data.append(char00);Data.append(char00);Data.append(0x02);

        // Method set region of itnerest(257) 0x0101
        Data.append(0x43);Data.append(0x3f);
        Data.append(0x01);Data.append(0x01);
        Data.append(char00);Data.append(char00);Data.append(char00);Data.append(0x2e);
        Data.append(0x05);Data.append(0xd6);
        Data.append(char00);Data.append(0x0c);
        // row 2
        Data.append(0x01);
        Data.append(0x01);
        Data.append(char00);
        Data.append(char00);
        Data.append(char00);Data.append(char00);Data.append(char00);Data.append(roiIndex);
        Data.append(videoSetting[0]);Data.append(videoSetting[1]);   // P1x
        // row 3
        Data.append(videoSetting[2]);Data.append(videoSetting[3]); // P1y
        Data.append(videoSetting[4]);Data.append(videoSetting[5]); //P2x
        Data.append(videoSetting[6]);Data.append(videoSetting[7]);     //P2y
        Data.append(0x01);
        Data.append(char00);
        Data.append(char00);Data.append(char00);
        // row 4
        Data.append(videoSetting[8]);Data.append(videoSetting[9]); // outputWidth
        Data.append(videoSetting[10]);Data.append(videoSetting[11]);   // OutputHeight
        Data.append(char00);Data.append(0x1e);Data.append(char00);Data.append(char00);
        Data.append(char00);
        Data.append(0x02);
        // row 5
        Data.append(char00);Data.append(char00);Data.append(char00);Data.append(0x32);
        Data.append(0x01); // 01 = JPEG, 02 = H.264
        Data.append(char00);
        Data.append(char00);
        Data.append(char00);
        Data.append(0x04);
        Data.append(0x01);
        // row 6
        Data.append(char00);Data.append(0xff);

        // Method: erase_cam_exclusive(25) 0x0019
```

```
        Data.append(0x43);Data.append(0x3f);
        Data.append(char00);Data.append(0x19);
        Data.append(char00);Data.append(char00);Data.append(char00);Data.append(0x08);
        Data.append(0x05);Data.append(0xd6);
        Data.append(char00);Data.append(0x0d);
       // row 2
        Data.append(0x01);
        Data.append(0x01);
        Data.append(char00);
        Data.append(char00);

        qDebug() << "Message_size: " << Data.size();
        QByteArray text = Data.toHex();
        qDebug() << "Message is: " << text;

        qint64 bytesSent = socket->writeDatagram(Data, qAddress, sendPort);
        if (-1 == bytesSent) {
            qDebug() << "Message not sent";
            return false;
        }
        else {
            qDebug() << "Socket: " << socket << ". Bytes sent: " << bytesSent;
            return true;
        }
    }
    else {
        return false;
    }
}

void MyUDP::readyRead()
{
    // when data comes in
    QByteArray buffer;
    buffer.resize(socket->pendingDatagramSize());
    qDebug() <<"Size of incoming msg: " <<  buffer.size();

    QHostAddress sender;
    quint16 senderPort;
    qint64 bytesRec= socket->readDatagram(buffer.data(), buffer.size(),
                      &sender, &senderPort);

    qDebug() << "Message: " << buffer.size();
    qDebug() << "Message from: " << sender.toString();
    qDebug() << "Message port: " << senderPort;
    qDebug() << "Message hex: " << buffer.toHex();
    qDebug() << "Message: " << bytesRec;
}

void MyUDP::readPendingDatagrams()
{
    while (socket->hasPendingDatagrams()) {
        QByteArray datagram;
        datagram.resize(socket->pendingDatagramSize());
        QHostAddress sender;
        quint16 senderPort;
```

```
        qDebug() << "Incoming message size: " << datagram.size();
        qDebug("test");
        qDebug() << "Message before replace: " << datagram.toHex();
        datagram.replace('\0', "\\0");
        qDebug() << "Incoming message size: " << datagram.size();
        qDebug() << "Message from: " << sender.toString();
        qDebug() << "Message port: " << senderPort;
    }
}
```

B The changes to QGstreamerPlayerSesseion

qgstreamerplayersession.ccp

```
@@ -56,7 +56,7 @@
 #include <QtCore/qdir.h>
 #include <QtCore/qstandardpaths.h>

-//#define DEBUG_PLAYBIN
+#define DEBUG_PLAYBIN
 //#define DEBUG_VO_BIN_DUMP

 QT_BEGIN_NAMESPACE
@@ -110,6 +110,15 @@ QGstreamerPlayerSession::QGstreamerPlayerSession(QObject *parent)
       m_pendingState(QMediaPlayer::StoppedState),
       m_busHelper(0),
       m_playbin(0),
+
+ //Crosscontrol code
+     mcc_udpsource(0),
+     mcc_rtpjpegdepay(0),
+     mcc_hw_decoder(0),
+    // mcc_sw_decoder(0),
+      mcc_hw_sink(0),
+ // end Crosscontrol code
+
       m_videoSink(0),
 #if !GST_CHECK_VERSION(1,0,0)
       m_usingColorspaceElement(false),
@@ -145,7 +154,16 @@ QGstreamerPlayerSession::QGstreamerPlayerSession(QObject *parent)
     Q_ASSERT(result == TRUE);
     Q_UNUSED(result);

-    m_playbin = gst_element_factory_make(QT_GSTREAMER_PLAYBIN_ELEMENT_NAME, NULL);
+    //  m_playbin = gst_element_factory_make(QT_GSTREAMER_PLAYBIN_ELEMENT_NAME, NULL);
+ // Crosscontrol code
+     mcc_udpsource = gst_element_factory_make("udpsrc","mcc_udpsource");
+     mcc_rtpjpegdepay=gst_element_factory_make("rtpjpegdepay","mcc_rtpjpegdepay");
+     mcc_hw_decoder = gst_element_factory_make ("mfw_vpudecoder", "mcc_hw_decoder");
+    // mcc_sw_decoder = gst_element_factory_make ("jpegdec ", "mcc_sw_decoder");
+     mcc_hw_sink = gst_element_factory_make ("mfw_isink", "mcc_hw_sink");
+     m_playbin = gst_pipeline_new ("stream-player");
+ // end Crosscontrol code
+
     if (m_playbin) {
         //GST_PLAY_FLAG_NATIVE_VIDEO omits configuration of ffmpegcolorspace and
videoscale,
         //since those elements are included in the video output bin when necessary.
@@ -225,8 +243,9 @@ QGstreamerPlayerSession::QGstreamerPlayerSession(QObject *parent)
         m_bus = gst_element_get_bus(m_playbin);
         m_busHelper = new QGstreamerBusHelper(m_bus, this);
         m_busHelper->installMessageFilter(this);
-
-        g_object_set(G_OBJECT(m_playbin), "video-sink", m_videoOutputBin, NULL);
+
+       //        g_object_set(G_OBJECT(m_playbin), "video-sink", m_videoOutputBin, NULL);
+  // Crosscontrol code
+  // end Crosscontrol code
```

```
        g_signal_connect(G_OBJECT(m_playbin), "notify::source",
G_CALLBACK(playbinNotifySource), this);
        g_signal_connect(G_OBJECT(m_playbin), "element-added",
G_CALLBACK(handleElementAdded), this);
@@ -310,7 +329,10 @@ void QGstreamerPlayerSession::loadFromStream(const QNetworkRequest
&request, QIO
         m_tags.clear();
         emit tagsChanged();

-        g_object_set(G_OBJECT(m_playbin), "uri", "appsrc://", NULL);
+        //g_object_set(G_OBJECT(m_playbin), "uri", "appsrc://", NULL);
+ // Crosscontrol code
+        g_object_set(G_OBJECT(mcc_udpsource), "uri", "appsrc://", NULL);
+ // end Crosscontrol code

         if (!m_streamTypes.isEmpty()) {
             m_streamProperties.clear();
@@ -334,7 +356,7 @@ void QGstreamerPlayerSession::loadFromUri(const QNetworkRequest
&request)

 #if defined(HAVE_GST_APPSRC)
     if (m_appSrc) {
-        m_appSrc->deleteLater();
+        m_appSrc->deleteLater();
         m_appSrc = 0;
     }
 #endif
@@ -343,7 +365,11 @@ void QGstreamerPlayerSession::loadFromUri(const QNetworkRequest
&request)
         m_tags.clear();
         emit tagsChanged();

-        g_object_set(G_OBJECT(m_playbin), "uri", m_request.url().toEncoded().constData(),
NULL);
+        // g_object_set(G_OBJECT(m_playbin), "uri", m_request.url().toEncoded().constData(),
NULL);
+ // Crosscontrol code
+        // g_object_set(mcc_udpsource,"location","/opt/toy_plane_liftoff.avi",NULL);
+        g_object_set(G_OBJECT(mcc_udpsource),"port",
m_request.url().toString().midRef(7).toInt(),NULL); // Very ugly, should be
replaced by a port property in the Video QML type
+ //end Crosscontrol code

         if (!m_streamTypes.isEmpty()) {
             m_streamProperties.clear();
@@ -564,8 +590,13 @@ void QGstreamerPlayerSession::setVideoRenderer(QObject *videoOutput)
 #endif

     GstElement *videoSink = 0;
-    if (m_renderer && m_renderer->isReady())
-        videoSink = m_renderer->videoSink();
+    if (m_renderer && m_renderer->isReady()){
+            // videoSink = m_renderer->videoSink();
+  // Crosscontrol code
+            // Uncomment row below to use hw-sink
+            videoSink= mcc_hw_sink;
```

```
+  // end Crosscontrol code
+        }

      if (!videoSink)
          videoSink = m_nullVideoSink;
@@ -742,6 +773,7 @@ void QGstreamerPlayerSession::finishVideoOutputChange()
      }

      removeVideoBufferProbe();
+

      gst_bin_remove(GST_BIN(m_videoOutputBin), m_videoSink);

@@ -894,6 +926,46 @@ bool QGstreamerPlayerSession::play()
      qDebug() << Q_FUNC_INFO;
 #endif

+
+
+
+
+qDebug() << "INSIDE QGSTREAMPERPLAYERSESSION:PLAY()";
+ // Crosscontrol code
+ /* play from file
+                 g_object_set(G_OBJECT(mcc_udpsource),"location",
"/opt/toy_plane_liftoff.avi",NULL);
+                  g_object_set(G_OBJECT(mcc_udpsource),"typefind",true,NULL);
+                          */
+
+       // g_object_set(G_OBJECT(mcc_udpsource),"port",5004,NULL);
+       g_object_set (G_OBJECT (mcc_udpsource), "caps",
gst_caps_from_string("application/x-rtp, encoding-name=JPEG, payload=26"), NULL);
+       g_object_set (G_OBJECT (mcc_hw_sink), "disp-height",200, NULL);
+       g_object_set (G_OBJECT (mcc_hw_sink), "disp-width",300, NULL);
+       g_object_set (G_OBJECT (mcc_hw_sink), "axis-left",300, NULL);
+
+              /* must add elements to pipeline before linking them */
+          // should try with mcc_hw_decoder and mcc_hw_sink later
+       gst_bin_add_many (GST_BIN (m_playbin), mcc_udpsource, mcc_rtpjpegdepay,
mcc_hw_decoder, m_videoOutputBin, NULL);  //mcc_hw_sink
+       qDebug() << "Crosscontrol ELEMENTS added to bin";
+
+        /* link */
+        //if (!gst_element_link_many (mcc_udpsource, mcc_rtpjpegdepay,
mcc_hw_decoder, m_videoOutputBin, NULL)) {
+
+        if (!gst_element_link(mcc_udpsource, mcc_rtpjpegdepay)) {
+              g_warning ("Failed to link first elements!");
+  }
+        if (!gst_element_link(mcc_hw_decoder ,m_videoOutputBin)) {  // mcc_hw_sink
+              g_warning ("Failed to link third  elements!");
+  }
+        if (!gst_element_link( mcc_rtpjpegdepay, mcc_hw_decoder)) {
+              g_warning ("Failed to link second  elements!");
+  }
+
+  /*    if (!gst_element_link_many (mcc_udpsource, mcc_rtpjpegdepay, mcc_hw_decoder,
```

```
mcc_hw_sink, NULL)) {
+       g_warning ("Failed to link elements!");
+       } */
+//end Crosscontrol code
+
      m_everPlayed = false;
      if (m_playbin) {
          m_pendingState = QMediaPlayer::PlayingState;
@@ -1474,9 +1546,10 @@ void QGstreamerPlayerSession::updateDuration()
 {
      gint64 gstDuration = 0;
      int duration = -1;
-
-     if (m_playbin && qt_gst_element_query_duration(m_playbin, GST_FORMAT_TIME, &gstDuration))
+
+     if (m_playbin && qt_gst_element_query_duration(m_playbin, GST_FORMAT_TIME, &gstDuration))
          duration = gstDuration / 1000000;
+     }

      if (m_duration != duration) {
          m_duration = duration;
```

References

[1] http://crosscontrol.com/en-US/Products/CrossControl/Product-groups/Display-computers/CCpilot-VA-1.aspx. accessed 2016-10-25 10:34:05.

[2] http://www.monitoryourassets.com/ip-vs-analog/. accessed 2016-06-07 09:51:32.

[3] http://www.securityelectronicsandnetworks.com/articles/2015/01/19/video-surveillance-technologies-strengths-and-weaknesses. accessed 2017-01-05 17:00:12.

[4] http://proto-x.net/en/tech/element/ahd-technology-720p-1080p-quality-via-a-coax-at-the-distance-of-500-m-without-any-latency-and-losses/. accessed 2017-01-05 17:01:58.

[5] https://www.indigovision.com/LinkClick.aspx?fileticket=mlIrEU7lcds%3D&portalid=0. accessed 2017-01-05 17:33:09.

[6] https://web.archive.org/web/20130314150935/https://www.gocsc.com/UserFiles/File/Panduit/Panduit098765.pdf. accessed 2017-01-08 11:05:21.

[7] http://cctvdesign.online/tutorials/cabling/maximum-wire-distance-for-ip-cameras/. accessed 2017-01-08 11:06:45.

[8] IEEE Standards Association, IEEE 802.3-2015 - IEEE Standard for Ethernet, Section 2, Table 33-18, item 4.

[9] http://www.automation.com/library/articles-white-papers/industrial-ethernet/building-a-secure-ethernet-environment. accessed 2016-10-20 11:02:15.

[10] https://vector.com/portal/medien/cmc/events/commercial_events/Automotive_Ethernet_Symposium_AES14/AES14_04_Voelker_BMW_Lecture.pdf. accessed 2016-01-04 16:02:42.

[11] http://www.asus.com/Car-Electronics/RECO-Smart-Car-and-Portable-Cam/. accessed 2016-06-07 16:00:12.

[12] https://www.orlaco.com/media/news/mirroreye-makes-mirrorless-trucks-possible-1. accessed 2016-06-07 16:00:45.

[13] http://cache.nxp.com/assets/documents/data/en/application-notes/AN4274.pdf. accessed 2016-12-15 15:48:29.

[14] https://gstreamer.freedesktop.org/. accessed 2016-08-31 09:36:26.

[15] https://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/manual.pdf. accessed 2016-08-31 11:67:37.

[16] https://wiki.qt.io/About_Qt. accessed 2016-10-26 13:56:26.

[17] http://doc.qt.io/qt-5/qmlapplications.html. accessed 2016-10-27 10:57:18.

[18] https://www.ietf.org/assignments/rtp-parameters/rtp-parameters.xml. accessed 2016-08-30 16:02:15.

[19] http://www.bogotobogo.com/Qt/Qt5_QUdpSocket.php. accessed 2016-07-04 15:13:51.

[20] https://www.freedesktop.org/software/gstreamer-sdk/data/docs/2012.5/gst-plugins-base-plugins-0.10/gst-plugins-base-plugins-playbin2.html. accessed 2016-11-16 14:25:02.

[21] http://gstreamer-devel.966125.n4.nabble.com/How-to-play-rtp-stream-with-playbin-td4675659.html. accessed 2016-11-16 15:25:48.

[22] https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt. accessed 2016-12-19 12:56:37.

[23] https://lwn.net/Articles/464270/. accessed 2016-12-21 20:14:21.

[24] https://gstreamer.freedesktop.org/features/index.html. accessed 2016-12-21 20:05:56.

[25] https://lists.freedesktop.org/archives/gstreamer-announce/2013-March/000273.html. accessed 2016-12-21 20:09:12.

[26] Shahriar Akramullah. *Digital Video Concepts, Methods, and Metrics: Quality, Compression, Performance, and Power Trade-off Analysis*. A Press, Berkeley, CA, 2014.

[27] ISO. *ISO 17215 Road vehicles — Video communication interface for cameras (VCIC) Parts 1-4*. ISO, 1st edition, 2014.

[28] Kirsten Matheus Kirsten Matheus, Thomas Konigseder. *Automotive Ethernet*. Cambridge University Press, University Printing House, Cambridge, United Kingdom, 2014.

[29] Herman Kruegle. *CCTV Surveillance: Video Practices and Technology*. Butterworth Heinemann, US, 2nd edition, 2006.

[30] James F. Kurose and Keith W. Ross. *Computer networking: a top-down approach*. Pearson, Boston, Massachusetts, US, 5th, international edition, 2010.

[31] Ken Lindfors. *Personal interview*. December 22, 2016.

[32] Maximatecc. *LinX Virtual Development Machine - Getting Started*. Maximatecc, Document Revision: 1.3, 2015.

[33] Orlaco. *EMOS Customer technical information document.* Orlaco, Document Revision: 0.9, 2016.

[34] Iain E. Richardson. *Video Codec Design: Developing Image and Video Compression Systems.* Wiley, Hoboken, N.J, US, 2nd edition, 2002.

[35] Iain E. Richardson. *The H.264 advanced video compression standard.* Wiley, Hoboken, N.J, US, 2nd edition, 2010.

[36] Dragan Stancevic. Zero copy i: User-mode perspective. *Linux Journal*, Issue 105, Jan, 2003.

[37] Anders Svedberg. *Personal interview.* August 31, 2016.

[38] J. Wu, H. Wu, C. Li, W. Li, X. He, and C. Xia. Advanced four-pair architecture with input current balance function for power over ethernet (poe) system. *IEEE Transactions on Power Electronics*, 28(5):2343–2355, May 2013.