Spring 5-2018

Real-Time Streaming Video and Image Processing on Inexpensive Hardware with Low Latency

Richard L. Gregg
University of Nebraska - Lincoln, rick.gregg@prsmip.com

Real-Time Streaming Video And Image Processing On Inexpensive Hardware With Low Latency

by

Richard L. Gregg

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Telecommunications Engineering

Under the Supervision of Professor Dongming Peng

Lincoln, Nebraska

May 2018

REAL-TIME STREAMING VIDEO AND IMAGE PROCESSING ON INEXPENSIVE HARDWARE WITH LOW LATENCY

Richard L. Gregg, M.S.

University of Nebraska, 2018

Advisor: Dongming Peng

The use of resource constrained inexpensive hardware places restrictions on the design of streaming video and image processing system performance. In order to achieve acceptable frame-per-second (fps) performance with low latency, it is important to understand the response time requirements that the system needs to meet. For humans to be able to process and react to an image there should not be more than a 100ms delay between the time a camera captures an image and subsequently displays that image to the user.

In order to accomplish this goal, several design considerations need to be taken into account that limit the use of high level abstractions in favor of techniques that optimize performance. The reference design shown in this work uses embedded Linux on commercially available hardware costing $150. Performance is optimized by employing Linux user-space to kernel level functions including Video for Linux 2 (V4L2) and the Linux frame buffer. Optimized algorithms for color space conversion and image processing using a Haar Discrete Wavelet Transform (DWT) are also presented.

The results of this work show that real-time streaming video and image processing performance of 10 fps to 15 fps with 67ms to 100ms of latency can be achieved on embedded Linux using low cost hardware, kernel level abstractions and optimized algorithms.

TABLE OF CONTENTS

LISTS OF MULTIMEDIA OBJECTS

FIGURES

TABLES

DEDICATION

I am indebted to my beautiful wife Janeen Macrino, without whose love, sacrifices, and loyalty neither life nor work would bring fulfillment. She has given me words of consolation when I needed them and a well-ordered home where love is a reality.

- Martin Luther King, Jr.

AUTHOR'S ACKNOWLEDGMENTS

I entered the undergraduate predecessor of this program at the University of Nebraska at Omaha in May 1976 and graduated with my Bachelors degree. Over the years I have accomplished much academically and professionally. In 2011, I came home to start my graduate work in the same program. I wanted to learn anew. And from the best.

To my Thesis committee Drs. Dongming Peng, Hamid Sharif, Kuan Zhang, and Harvey Siy - thank you for challenging me to learn and allowing me to work with you in accomplishing this goal. It has been a privilege and an honor.

To my current and former instructors from my undergraduate years Drs. Bing Chen and Hamid Sharif, Professors Roger Sash, Ed Hollingsworth (retired), and Charles Sedlacek (1931-2016). Thank you for giving me the skills that I needed to become a respected scientist, engineer, inventor, entrepreneur and patent author.

To my former undergraduate classmates Bob Anderson, Tom Vaiskunas, Chuck Arbaugh, Dave Stone, Ken Johnson (deceased), Doug Grote and Tracy Osborn - thank you for encouraging me not to fall too far behind the pace you were setting.

To my very first Engineering mentor John Gaukel (1944-2014). Thank you for hiring me before I ever had a degree and taking a chance on me. You never limited my creativity. You let me make mistakes and learn from them. You taught me that anything is possible.

I remember a time in fifth grade 50 years ago when all of my classmates were given the classic career day assignment to ask their father's what they did for a profession and to make a living. My father, who was an Optician and a small business owner simply told me with great pride that "he helped people to see better." Given that I have embarked on an area of research that uses technology to accomplish the same goal for visual impairment may be more than just a little bit coincidental.

To my parents Ann and Lee (1921-1985) Gregg. The foundation on which all else has been built. I love you both very much.

1. INTRODUCTION

The use of on-demand streaming video has become popular in entertainment services such as YouTube, Crackle, HBO Now, Hulu Plus, Amazon Prime and, of course, Netflix. These services store and stream pre-recorded content and provide an alternative to cable and satellite on-demand video services at a lower cost. Live streaming video has gained adoption in video conferencing services such as Skype, WebEx, GoToMeeting, and Apple's FaceTime. Video conferencing offers benefits that increase collaboration and lower the costs of travel while increasing productivity. These streaming video services are run on very large Content Delivery Networks (CDN) or Telecommunications Networks that are expensive to deploy and global in scale. There has been a large body of scientific and academic work over the past many years focused on network architectures and protocols to improve Quality-of-Service (QoS) performance for on-demand streaming video and near real-time streaming video conferencing.

Digital image processing uses computer algorithms to perform image processing on digital images. Image processing is used in medical (x-ray, MRI, ultrasound), environmental (weather, space, land) and manufacturing (quality control, sorting, assembly) applications. Digital video processing combines streaming video with image processing to identify and track a moving target, provide autonomous vehicle guidance and detect motion in security applications. Streaming video image processing is computationally expensive and is performed in both offline and online environments.

Real-time streaming video image processing places additional constraints on streaming video and image processing performance by requiring a response time within a specified time constraint. In order to more accurately define real-time streaming video performance requirements, the specific application needs to be considered. In Robert Miller's 1968 classic paper entitled "*Response Time In Man-Computer Conversational Transactions*" [1], Miller described three different orders of magnitude of responsiveness:

1. A response time of 100ms is perceived as instantaneous.
2. Response times of 1 second or less are fast enough for users to feel they are interacting freely with the information.
3. Response times greater than 10 seconds completely lose the user's attention.

As human beings we have the ability to observe and experience the passage of time. However, our brains limit our sensory perception in a way that prevents us from reacting to our perceptions within a short timeframe commonly known as reaction time.

The average human reaction time is 250ms on average. Reaction time is a complex subject and includes several different facets of mental processing including sensory perception [2]. Sensory perception allows our senses to receive incoming audio or visual data from the outside world. For example, consider an image a Welder sees when initially striking an arc on a welding torch and beginning a welding operation. Latency is the delay between action and reaction. The time that welding image takes to travel down the optic nerve into the visual cortex is incredibly fast, on the order of 13ms [3] or about 1 in 75 frames per second. As the brain receives the incoming data stream, an asynchronous process acknowledges the input and admits it into our consciousness. Now aware of the incoming data stream, another part of the brain applies context to the stream so that a decision can be made about how to react which happens very quickly.

Increasing latency above 13ms has a negative impact on human performance for a given task. While imperceptible at first, added latency continues to degrade human processing ability until approaching 75ms to 100ms. At this point, we become conscious that input has become too slow and adapt to conditions by anticipating input rather than simply reacting to input. For example, Massive Multiplayer Online Gaming (MMOG) presents immersive and lifelike experiences to their users. Games with very realistic environments, including those using Virtual Reality (VR) technology have very strict data stream latency requirements. In a Virtual Reality environment, low latency is fundamentally important to deliver optimal experiences that the eyes and brain accept as real. Recent data shows that in VR gaming environments [4], a game is unplayable with a latency of 300ms and becomes degraded at 150ms. Player performance can be affected at 100ms suggesting a target for latency performance between 50ms to 100ms. However, a delay of even 100ms can reduce player performance in games by a measureable amount that forces players into predicting movements.

Due to the emergence of very powerful, low-cost, and energy-efficient processors, it has become possible to incorporate practical real-time streaming video image processing into embedded systems. In addition to VR headsets, examples include surveillance applications using IP network cameras, vehicle backup and collision avoidance systems, and Augmented Reality Smart Glasses (ARSG) [5].

It is important to distinguish between augmented reality and virtual reality as these two concepts are not the same. In virtual reality, users are immersed or semi-immersed in a virtual world and cannot typically see the real world around them. In contrast, augmented reality merges the virtual and real worlds by overlaying information on the user's perception of the real world by 1) combining real and virtual objects, 2) the ability to interact in real-time; and 3) the ability to use 3D objects. The major breakthrough in augmented reality occurred in 2016 when the popular game Pokemon Go was released around the world. One of the more notable earlier attempts at augmented reality was Google Glass. While Google Glass missed in the consumer realm, it made an impact on enterprises by popularizing the concept of Smart Glasses.

According to Forrester Research [6], the Smart Glasses market is real and tangible for enterprises. About 14 million US workers will use Smart Glasses in their jobs by 2025 creating $30 billion in US Smart Glasses hardware revenue through 2025. The interest and momentum building around enterprise adoption of Smart Glasses is being driven by positive ROI expectations. Leveraging Smart Glasses, a company can reduce costs by 15-25%, a substantial savings in any business [7]. For decades industrial enterprises have looked at lean-manufacturing processes, training, robotics and automation as the primary strategies for improving operational performance. While these have yielded significant benefits, a gap remains: many activities in an industrial environment still need humans to be directly hands-on. Given the nature of these jobs it had previously not been possible nor practical to provide the workforce with all the data, applications and access right where it is most impactful – working with their hands, with tools, while moving around.

In this paper, the concept of Mediated Reality Smart Glasses (MRSG) is introduced to improve operator vision during a manual welding operation. As opposed to Virtual Reality (VR) and Augmented Reality (AR), Mediated Reality (MR) alters one's perception of reality by changing what someone is actually seeing in a typically immersive environment. Conceptually, the traditional welding helmet auto-darkening filter (ADF) cartridge is replaced with a cartridge having the same form factor as the ADF. Known as a mediated reality welding (MRW) cartridge, this ADF replacement contains low-cost embedded hardware running a real-time streaming video image processing application that improves operator vision during a manual welding operation. The mediated reality welding cartridge is thought to be one of the first improvements in operator vision since the auto-darkening filter was patented in 1975 [8] and introduced into the market in 1981.

One of the major factors for driving the adoption of Mediated Reality Welding into the market, and the subject of this Thesis, is to determine if real-time streaming video image processing is possible on inexpensive hardware with low latency. The target price range to the consumer for a MRW replacement cartridge is $300 to $500 based on the current market price for ADF replacement cartridges. This low price point constrains the tradeoffs that must be considered between hardware cost, software design and real-time streaming video image processing performance. This paper examines how those tradeoffs are addressed, provides a reference design with performance results, concludes offering some additional recommendations for the current work, and identifies future work.

2. BACKGROUND

In my review of the relevant published academic papers, there has been much written about streaming video, image processing and embedded systems, but not typically together. The papers reviewed focus on high level design, Internet Protocol (IP) camera surveillance systems and specific image processing applications such as ultrasound or object detection. I found very little work in the Augmented Reality (AR) area that was relevant to the topic of this Thesis. Consider the following:

R. Hill et al., in "*Measuring Latency for Video Surveillance Systems",* discuss the increased flexibility and benefits offered by Internet Protocol (IP) network cameras making them a common choice for installation in surveillance networks. A common limitation of IP cameras is their high latency when compared to their analog counterparts. This historically caused some reluctance to install or upgrade to digital cameras and has slowed the adoption of live, intelligent analysis techniques (i.e. image processing) into video surveillance systems [9]. This work shows that IP camera video streaming latency across a network is in the range of 120ms to 200ms but ignores the performance constraints added by additional image processing algorithms; and unlike this work, relies on the use of a network in the overall streaming video system.

In their paper entitled "*Research on Embedded Video Monitoring System Based on Linux*", Q. Li et al., disclose the use of hardware components to decode YUV422 video and then subsequently encode the video into MPEG-4 [10] using a custom Video for Linux driver, but fails to disclose any streaming video and image processing performance results and effectively teaches away from this work which is more cost effective by using a look up table loaded into memory to perform the corresponding encoding functions offered by the additional hardware and custom driver disclosed in this paper.

Poudel, et al., notes that real time computer vision applications like video streaming on cell phones, remote surveillance and virtual reality have stringent performance requirements which can be severely constrained by limited resources. In their paper entitled "*Optimization of Computer Vision Algorithms for Real Time Platforms*" [11], the authors propose that the use of optimized algorithms is vital to meet real-time requirements because computer vision algorithms are computationally intensive and resource exhaustive. However, unlike this work, the aforementioned paper uses OpenCV and dual core architectures to achieve the performance results.

The reader is informed that the results from the author's benchmarking tests suggest that exploiting all the available on chip hardware resources and assigning computation intensive tasks to dedicated hardware is one of the main techniques to achieving real time performance. The research disclosed by this Thesis purposefully does not use high level image processing libraries such as OpenCV, additional dedicated streaming video and image processing hardware; and is relegated to a single core processor. Instead, this work achieves optimization on more inexpensive resource constrained hardware through user-space to kernel level C language optimizations for streaming video and image processing without the use of dedicated hardware. In fact, the Graphics Processing Unit (GPU) capability of the BeagleBone Black single board computer used in this research is not utilized in order to further constrain the hardware resources.

In the paper "*Design and Realization of Image Processing System Based on Embedded Platforms*" [12], a hardware and software architecture is introduced to support image processing on an embedded system. The architecture uses a 32-bit ARM processor with a Linux distribution and provides for a primitive set of application programming interfaces (API) to support functions such as geometric transformation, edge detection, and contour tracing. No details are provided regarding algorithm implementation or image processing performance although the author's claim that "the image processing system based on embedded platform can be installed in hand held devices to satisfy the user's need for image processing at a lower cost. Testing results indicate that the system is highly efficient." No testing results are provided to support the claims made by the author, but do suggest the design of embedded image processing systems plays an active role in the "application domains of the relative technology." In addition, there is no disclosure of the use of streaming video in this paper.

With the rapid development of the computer, network, image processing and transmission technology, the application of embedded technology in video monitoring has become more widely adopted. D. Li et al., in a paper entitled "*Design of Embedded Video Capture System Based on ARM9*" [13], presents a design based on a S3C2440 hardware platform and Linux operation system using a mesh V2000 camera for video collection, combined with V4L video interface technology and MPEG-4 video coding and decoding and video transmission technology, aiming at design of a low-cost high-performance program. The paper elaborates the development process of a USB camera in an embedded Linux operation system, the use of MPEG-4 video coding technology and the network transmission realization of video data. The authors claim their design realizes rapid video acquisition and real-time transmission with stable performance and lower cost. However, this paper fails to teach the reader how any of these performance objectives are accomplished and certainly does not illuminate the reader with the performance results of their system.

An interesting paper put forth by Schlessman et al., entitled "*Tailoring Design for Embedded Computer Vision Applications*," [14] discusses the tradeoffs that need to be considered when designing embedded computer vision systems. The authors propose a design methodology that focuses on two critical challenges when developing embedded image processing algorithms: 1) numerical characteristics and 2) memory. Numerical characteristics mentions the data types that can be used to write and optimize image processing algorithms while memory refers to the amount of memory that will be consumed by selecting a specific data type(s) to implement image processing algorithms. The authors propose that the both numerical characteristics and memory interact because high-precision numerical representations require more memory and therefore memory bandwidth. This is something I found to be true during my research for this Thesis. The authors observe that "many algorithm designers liberally use doubled-precision, floating point arithmetic to avoid dealing with numerical problems, which incurs substantial memory and energy. They often use MATLAB or the OpenCV library which provide library functions for very abstract operations obscuring implementation costs."

The methodology was then applied to two common computer vision applications: 1) optical flow analysis and 2) background subtraction. Both algorithms started their development life inside of MATLAB to get an understanding of the tasks the algorithm needed to perform and what data types could be used. In embedded computer vision, data representation should be based on integer values whenever possible. But, if rational values had to be used, the selection of fixed or floating point values had to be determined. The algorithms were converted to the C language using MATLAB Coder and instrumented using SimpleScaler or VTune to determine performance. The algorithms were then converted to the C language and further instrumented and optimized for performance. The authors show that using their methodology, the number of floating point instructions is greatly reduced (zero for the two algorithms identified), memory consumption is reduced by a factor of three and the CPU speed required for a compatible level of performance is cut in half. Although I used a more intuitive, iterative approach based on my years of experience in MATLAB and C language coding skills, the methodology disclosed in this paper enforces more of a structured discipline and should definitely be considered for future work in this area.

In "*A Real-Time Remote Video Streaming Platform for Ultrasound Imaging*" [15], M. Ahmadi et al., propose a real-time streaming video platform which allows specialist physicians to remotely monitor ultrasound exams. An ultrasound steam is captured using a Phillips Sparq Ultrasound probe and transmitted through a wireless network. In addition, the system is equipped with a camera to track the position of the ultrasound probe. The system uses Video for Linux to control and capture the video stream which is displayed on the popular Linux desktop application Gstreamer. The video was compressed using Motion JPEG and transmitted from the Linux server over a local wireless network to a Galaxy Note 10 tablet. A latency of less than one second was achieved with a resolution close to high definition (HD). While this is an interesting application, its implementation fundamentally teaches away from the work presented herein. It is worth noting again the meaning of the term "real-time" as it applies to a specific streaming video image processing application.

Finally for consideration is the S. Saypadith et al., paper "*Optimized Human Detection on the Embedded Computer Vision System*." [16]. Presented in this paper is a real-time human detection technique that is capable of real-time image processing on a Raspberry Pi using a Histogram of Oriented Gradients (HOG) image processing algorithm to differentiate a human from a scene and fed into a Support Vector Machine (SVM). The Raspberry Pi is a resource constrained embedded Linux single board computer similar to the BeagleBone Black used in this research. Interestingly, the use of a Haar variant (the Haar Discrete Wavelet Transform is used in this Thesis) could be used for detecting objects such as human bodies and faces [17].

Streaming video was not used in this paper. Instead, reference video from the Town Center and CAVIAR data sets was processed by the optimized OpenCV HOG and SVM image processing algorithms disclosed in [18]. The maximum performance attained was approximately 2.9 frames-per-second (fps) with less accuracy than other HOG algorithms but within acceptable limits. According to the authors, "Although, accuracy of proposed system is slightly low, it can be used in outdoor applications like pedestrian detection and surveillance systems. The Raspberry Pi based solution has advantages over other smart solutions depending on the problem. Given a limited number of fps on nominal resolution, such low-cost independent and portable solution can be employed. However, for visual data at higher fps and resolution, Raspberry Pi might not be a good choice." It would be interesting to use the results of this work to determine if the performance of such a system could be improved.

None of the reviewed papers define what the term “real-time” means in a streaming video image processing environment. Surprisingly, there does not appear to be a single reference that illustrates a detailed reference implementation teaching the reader about design considerations, optimization tradeoffs and provides a specific reference implementation with supporting performance data that answers the question “*Can real-time streaming video image processing be implemented on inexpensive hardware with low latency?*”

3. MOTIVATION

After my survey of the literature, one very important aspect of this work is the ability to perform in-camera analysis [19]. This can be an important consideration for large scale computer vision systems. Some multi-camera systems send video to the cloud which consumes significant bandwidth. In-camera analysis systems can ensure that raw video never leaves the camera. This additional level of privacy created by the lack of a video record may be an attractive alternative in many situations. Another emerging application for this work is the replacement of conventional mechanical process control sensors in manufacturing environments by a single camera combined with machine vision algorithms [20]. The work disclosed in this Thesis can also be used in the design of AR/VR/MR headsets; and finally, any application that requires improving eye sight for visually impaired individuals or in visually impaired environments is a certainly a target for this research.

The principal motivation for this Thesis is a result of the future work identified in the paper entitled "*Mediated Reality Welding*" [21] by the author. This paper disclosed the development of image processing algorithms used to improve operator vision during a manual welding operation. In this work, video of a welding operation was captured through the use of an ADF and a video camera. The captured video was used as input to image processing algorithms utilizing compositing, region-of-interest (ROI), object tracking; and object subtraction and substitution resulting in a Mediated Reality Welding output video. This work resulted in several pending patent applications in the US [22], EPO [23], Japan [24] and China [25].

The output video was generated on a workstation class personal computer in an offline fashion without any considerations given to real-time performance. These algorithms demonstrated that the decades old visual environment presented to an operator by a welding helmet can be improved. The following figures shows the difference between ADF and MRW at the same instant in time when the arc is first struck by the Operator's MIG welding torch and the ADF goes dark requiring the welder to intuitively follow the weld puddle. With the use of MRW, the Operator's vision of the welding operation is clearly enhanced.

Figure 1 - Auto Darkening Filter

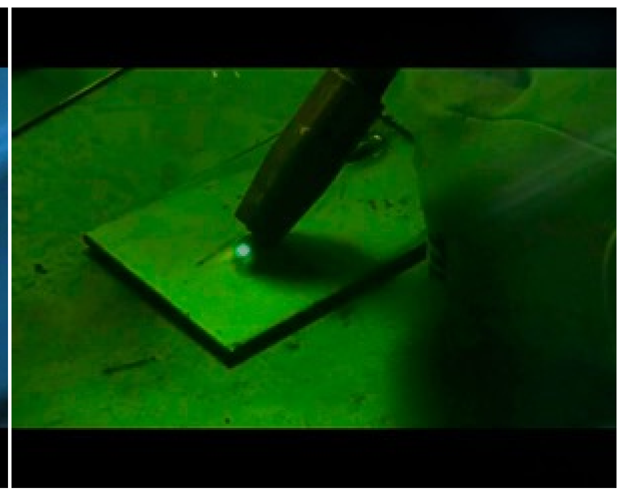


Figure 2 - Mediated Reality Welding

Forrester Research forecasts that 14M US workers will use ARSG in their jobs by 2025 representing $30B in US ARSG hardware revenue [26]. The use of ARSG is predicted to reduce costs by 15-25% percent [27], a substantial savings in any business. Cost reduction will occur by impacting four primary variables: 1) *Labor Productivity* - accelerate activity speed and reduce idle time, 2) *Quality and Defects* - reduce defects and lower associated rework and scrap costs, 3) *Resource Utilization* - improve resource utilization and lower labor costs by accelerating new-hire ramp-up time, up-skill current workers, optimize the access and use of experts across a company, reduce downtime and enable lower cost resources to perform high skill work; and 4) *Risk* - Decrease the number and severity of unplanned events. Additional benefits include improvements in process optimization and flexibility promoting a high-quality and consistent operation.

For some niche jobs, like welding, Smart Glasses are incredibly relevant to the point where I believe that the majority of welders will use them as tools by 2025 if MRW can be commercially developed. In my previous paper, several areas of future work were identified including the open question asking if "implementation of the mediated reality welding system using real-time optimized algorithms onto low-cost hardware in the cartridge form factor currently used by welding helmets" is possible.

4. HARDWARE ARCHITECTURE

When taking into account the requirements for developing prototype hardware to study the feasibility of Mediated Reality Welding, the objectives for selection of the hardware components necessitated that the hardware be: 1) low-cost, 2) have a small footprint 3) be commercially available off-the-shelf; and 4) have the ability to support the capture of streaming video from a camera, process the video using image processing algorithms and display the result on an integrated display.

These goals were accomplished for a total expenditure of $150 by selecting the: 1) BeagleBone Black Single Board Computer ($50), the 2) Logitech HD Pro Webcam C920 ($50) and; 3) 4D Systems 4D 4.3" LCD CAPE for the BeagleBone Black ($50). The following figure shows the hardware used in this research.

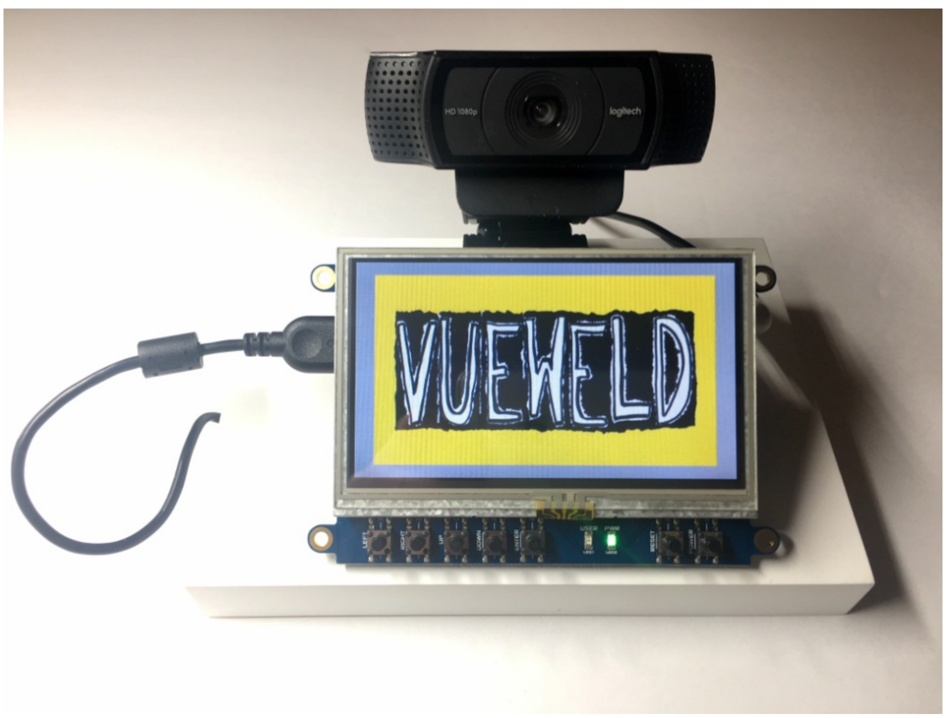
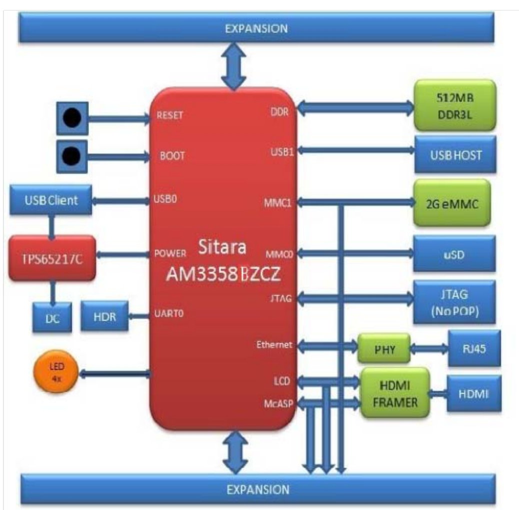


Figure 3 - System Hardware

4.1 BEAGLEBONE BLACK SINGLE BOARD COMPUTER

The BeagleBone Black [28][29] is a compact, low-cost, open-source Linux computing platform that provides a USB 2.0 host port to support the Logitech C920 Webcam and two (2) 46 pin expansion headers P8 and P9 to accommodate the 4D Systems LCD display. The BeagleBone Black ships with the Angstrom Linux distribution installed although other Linux distributions such as Ubuntu and Debian are available. Angstrom is a stable and lean Linux distribution that is widely used on embedded systems. The following figure and table illustrates the block diagram and for the BeagleBone Black.



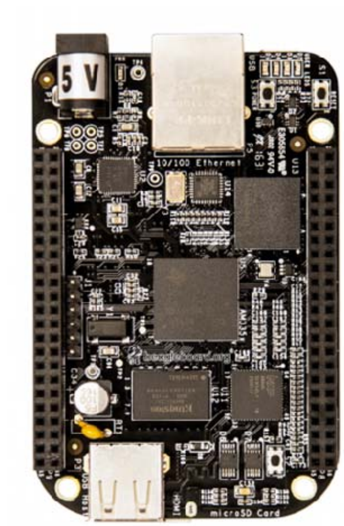


Figure 4 – BeagleBone Black Single Board Computer Block Diagram

The BeagleBone Black single board computer hardware reference design was designed, openly published and produced by Texas Instruments as a way of demonstrating Texas Instruments' AM335X system-on-a-chip (SOC). The AM335X SOC uses a 32-bit ARM Cortex-A8 processor core with a programmable CPU clock frequency range from 600 MHz to 1000 MHz. The following table illustrates the specifications for the BeagleBone Black single board computer.

BeagleBone Black Single Board Computer	
Feature	**Specification**
CPU	TI Sitara AM3358BZCZ100 @ 300, 600, 800, 1000 MHZ, 2000 MIPS @ 1GHZ **(Note: Single Core CPU).**
Graphics Engine	SGX530 3D, 20M Polygons/S **(not supported by Angstrom Linux)**
SDRAM Memory	512MB DDR3L 800MHZ
Onboard Flash	4GB, 8bit Embedded MMC
USB 2.0 Host	USB Type A female connector with full LS/FS/HS Host support
Ethernet	10/100 RJ45
SD/MMC	microSD
Video Out	16 bit RGB565 HDMI, 1280x1024 (max) OR 16 bit LCD via expansion headers
Power	5VDC 1A power supply plugged into DC connector
Expansion	Power 5V, 3.3V , VDD ADC(1.8V), 3.3V I/O on all signals McASP0, SPI1, I2C, GPIO(69 max), LCD, GPMC, MMC1, MMC2, 7 AIN(1.8V MAX), 4 Timers, 4 Serial Ports, CAN0, EHRPWM(0,2),XDMA Interrupt, Power button, Expansion Board ID (Up to 4 can be stacked)

Table 1 – BeagleBone Black Single Board Computer Specifications

4.2 CAMERA

The Logitech HD Pro Webcam C920 [30] is USB 2.0 compliant with a maximum frame rate of 1080p at 30 frames per second. The camera supports USB video device class or UVC for streaming video. UVC (uvcvideo) is built into the Angstrom Linux distribution installed on the BeagleBone Black (/dev/video0).

Logitech HD Pro C920 Webcam	
Feature	**Specification**
Resolution	160 x 90 to 2304 x 1536 @ 2 – 30 fps
Format	YUV422 (raw), H.264 (compressed), MJPEG (compressed)
Control	Brightness, contrast, saturation, white balance, gain, sharpness, backlight compensation, auto exposure, manual exposure, pan, tilt, auto focus, manual focus, and zoom.
Interface	USB 2.0

Table 2 – Logitech HD Pro C920 Webcam Specifications

Pixel resolution, frame rate, video format and several camera controls can be set programmatically by software on the Logitech HD Pro Webcam C920. Pixel resolutions from 160 x 90 to 2304 x 1536 at 2 to 30 frames per second are supported. Video formats include YUV422 (raw), H.264 (compressed), and MJPEG (compressed). Camera control features including brightness, contrast, saturation, white balance, gain, sharpness, backlight compensation, auto exposure, manual exposure, pan, tilt, auto focus, manual focus, and zoom.

4.3 DISPLAY

4D 4.3" LCD BeagleBone Black Cape	
Feature	Specification
Resolution	480 x 272
Format	16-bit RGB565
Control	7 push buttons including left, right, up, down, enter, reset and power.
Interface	BeagleBone Black P8/P9

Table 3 – 4D 4.3" LCD BeagleBone Black Cape

The 4D Systems 4D 4.3" LCD CAPE [31] is specifically designed for the BeagleBone Black. The CAPE features a 4.3" TFT LCD 480 x 272 pixel resolution 16-bit RGB565 display. The CAPE includes 7 push buttons including left, right, up, down, enter, reset and power. This research used the 4DCAPE-43 which is the non-resistive touch version of the display. The display mounts directly to the BeagleBone black expansion headers.

The TI Sitara AM335X provides I/O support for 16-bit RGB565 displays. The BeagleBone Black can support only one display at a time, either a HDMI display via the NXP TDA19988 HDMI Transmitter chip or an LCD display via the P8 and P9 expansion headers.

5. SOFTWARE DESIGN CONSIDERATIONS

The overall software design goal for this research was to focus on algorithm optimization for both the streaming video and image processing functionality with the primary focus on frame-per-second (fps) performance. There are several methods and libraries available on Linux when considering what to use for video capture and display.

5.1 OPERATING SYSTEM

Angstrom Linux is the default Linux distribution that ships with the BeagleBone Black. There are many reasons why Linux has been adopted as the operating system in many embedded systems products beyond its traditional stronghold in server applications. Examples of these embedded systems include cellular phones, video games, digital cameras, network switches and wireless communications gear. Embedded Linux is also being widely adopted in AR/VR/MR headsets and Internet of Things (IoT) applications.

Because of the numerous economic and technical benefits, there is strong growth in the adoption of Linux for embedded devices. This trend has crossed virtually all markets and technologies. Some of the reasons for the growth of embedded Linux are 1) Linux supports a vast variety of hardware devices, probably more than any other operating system; 2) Linux supports a large number of applications and networking protocols; 3) Linux can be deployed without the royalties required by traditional proprietary embedded operating systems; 4) Linux has attracted an impressive population of active developers, enabling rapid support of new hardware architectures, platforms, and devices; 5) An increasing number of hardware and software vendors, including virtually all the top-tier chip manufacturers and independent software vendors now support Linux.

5.2 STREAMING VIDEO

The Logitech C920 USB Webcam was chosen as the video camera for this research. This camera supports the USB video device class (UVC) [32]. In fact, UVC is the typical way for any Linux distribution to use a USB Webcam. Angstrom Linux provides the UVC device (`uvcvideo`) at `/dev/video0`. This appears as a file, but is actually an interface to the device driver. Although it is not a real file, it can be opened, read and written to.

Video4Linux2 (V4L2) [33] is a collection of device drives and an API that supports streaming video on Linux systems and uses the UVC driver. The 4VL2 framework has been made an integral part of the standard Linux kernel. The 4VL2 API allows manipulation of various video devices for capture as well as output. The API is mostly implemented as a set of `ioctl()` calls in the Linux operating system. The `ioctl()` function provides an input/output control that uses a system (kernel) call for device specific input/output operations and other operations that cannot be expressed by regular system calls. As a result of the wide adoption, good documentation, code examples; and most importantly, the 4VL2 API works directly from the user-space to the system kernel level providing optimal performance for this work.

5.3 IMAGE PROCESSING

Several image processing libraries including the Open Source Computer Vision Library (OpenCV) and MATLAB Coder were considered for use in this research. OpenCV [34][35] is released under a Berkeley Software Distribution (BSD) license and hence it's free for both academic and commercial use. OpenCV has C++, Python and Java interfaces; and supports Windows, Linux, Mac OS, iOS and Android. OpenCV was designed for computational efficiency and with a strong focus on real-time applications. Written in optimized C/C++, the library can take advantage of multi-core processing. OpenCV can take advantage of the hardware acceleration of the underlying heterogeneous computing platform and supports V4L2 making it an excellent choice for embedded Linux computer vision applications.

Since the initial work for algorithm development was accomplished using MATLAB, the use of MATLAB Coder [36] was considered as a tool to produce source code for this work. MATLAB Coder generates C and C++ code from MATLAB code for a variety of hardware platforms, from desktop systems to embedded hardware. It supports most of the MATLAB language and a wide range of MATLAB toolboxes. Generated code can be inserted into software development projects as source code, static libraries, or dynamic libraries. The author, in the evaluation of MATLAB Coder, found that it did not completely support many of the computer and vision image processing functions that are supported by MATLAB including many functions used in the author's previous work, generated non-optimized code; and was very expensive to purchase. For the aforementioned reasons, MATLAB Coder was not chosen for this research.

The Haar Discrete Wavelet Transform (DWT) [37][38][39][40][41] is an efficient way to perform both lossless and lossy image compression. Lossless and lossy compression are terms that describe whether or not, in the compression of a image, all original data can be recovered when the image is uncompressed. With lossless compression, every single bit of data that was originally in the image remains after the file is uncompressed.

The Haar DWT relies on averaging and differencing values in an image matrix to produce a matrix which is sparse or nearly sparse. A sparse matrix is a matrix in which a large portion of its entries are zero. A sparse matrix can be stored in an efficient manner, leading to smaller file sizes. Wavelets provide a mathematical way of encoding information in such a way that it is layered according to level of detail. This layering facilitates approximations at various intermediate stages and can be stored using a lot less space than the original data.

The Haar DWT is one of the simplest and basic transformations from the space/time domain to a local frequency domain thereby revealing the space/time variant spectrum. This makes the Haar DWT a candidate for a wide variety of applications using signal and image compression. In our research, the Haar DWT is used to benchmark image processing performance.

Unfortunately, Haar DWT support was not available in OpenCV or MATLAB Coder requiring the author to thoroughly understand the Haar DWT and write optimized C code from scratch. In addition, it was important to get an understanding of how to actually write optimized image processing algorithms for future work.

5.4 VIDEO DISPAY

When considering access to the graphics hardware and the display [42][43][44], the most common graphics architecture in Linux ix X11. However, this is not the only way that Linux has to display graphics. For purposes of this work, rending graphics in xlib, xcb, GTK+ or QT5 is at too high of an abstraction level that will most likely negatively affect FPS performance. This leaves some of the lower level abstractions for potential use including: 1) X Server (X11) direct connection, 2) direct rending manager (DRM) dumb buffers; and 3) the Linux frame buffer device (fbdev).

X Server (X11) Direct Connection – Connecting to the X11 X Server is by far the most common method of displaying graphics on Linux systems. It has been around a long time and is in use on virtually every Linux system. However, X11 is far from a simple and easy to use protocol. Typically, an application using X11 for graphics will use a very high level widget library, such as GTK+ or QT5, which in turn use Xlib or XCB to establish connection and handle communication with the X Server. A simpler application might only use Xlib or XCB if the programmer has enough skill. XCB is currently accepted as the lowest level method possible of communicating with the X Server.

The X11 protocol uses the client server model for communication. This means that, if we can open sockets, we can connect to the X Server on our own, without relying on Xlib or XCB to facilitate communication. We will just have to handle the X11 protocol in the application software. When writing an X Server, this would be a very daunting and nearly impossible task given the scope of all the X Server is expected to handle; however, writing a client is a much simpler task since only certain parts of the protocol will need to be implemented. Given the complexity and high level abstractions required, this approach was eliminated from future consideration.

Direct Rending Manager (DRM) Dumb Buffers - DRM is a much more feature rich interface, and provides a lot more options which also means it's a lot more complicated to use. DRM offers control over the graphics hardware, which is great for hardware acceleration and also has a mechanism for double buffering. DRM has a feature called "dumb buffers" which is essentially a frame buffer. It's supposedly the easiest to set up, but in reality is very complicated. DRM provides a kernel interface which was given serious consideration given the fps performance challenges of this work; however, most DRM applications use libdrm which makes software development a lot easier but also introduces a higher level of abstraction which may impact performance. Because the goal is to eliminate any use of user libraries, keep the implementation simple and stay as low-level as possible, DRM was eliminated from consideration in this work.

The Linux frame buffer device (fbdev) - The Linux frame buffer is often talked about, but rarely actually used. One of the main reasons for this is that documentation and examples are fairly hard to come by. Like many things, the people that know how to program for the frame buffer are few and far between. First of all, the Linux kernel must be built with support for the correct frame buffer driver. Angstrom Linux provides the frame buffer device at `/dev/fb0`. This appears as a file, but is actually an interface to the device driver. Although it is not a real file it can be opened, read and written to.

Frame buffer drivers are especially interesting for embedded systems, where memory footprint is essential, or when the intended applications do not require advanced graphics acceleration. At the core, a frame buffer driver implements the following functionality: 1) mode setting, and 2) optional 2D acceleration. Mode setting consists of configuring the frame buffer to get a picture on the screen. This includes choosing the video resolution and refresh rates. Frame buffer drivers can provide basic 2D acceleration used to accelerate the Linux console. The Linux frame buffer interfaces with the graphics hardware and display directly at the Linux kernel level which should minimize latency and have a favorable fps performance. Frame buffers remain the simplest and easiest to use when considering all of the other alternatives. Given the fps and latency performance challenges of this work, the Linux frame buffer was chosen for this research.

6. SOFTWARE DEVELOPMENT ENVIRONMENT

In the software development environment for this research, the BeagleBone Black is the target system and a Dell M6600 Precision Workstation was used as the development host system. Since the BeagleBone Black uses Angstrom Linux by default, the distribution that shipped with the board was updated to the latest version, 3.8.13. The GNU toolchain was installed on the BeagleBone Black and used to compile the C language source code on the target system using GCC and debugged using GBD. The BeagleBone Black was connected to an IP based LAN accessible by the Dell M6600 workstation. The workstation uses Windows 10 as the host operating system configured with an Oracle VM VirtualBox 5.2.8 containing an Ubuntu 16.04 LTS guest machine. In addition to the GNU toolchain also installed on the workstation, several other image processing tools were used for development, test and debug including 7yuv [45] and FFmpeg [46] both of which were used extensively; and ImageMagick [47]. The BeagleBone Black and Dell M6600 Workstation communicated using SFTP and SSH over an IP network. The workstation's Windows 10 host operating system had MATLAB version 9.3.0.713579 (R2017b) installed with the MATLAB computer vision, image processing and wavelet toolboxes in order to perform the initial development of the Haar Discrete Wavelet Transform (DWT) algorithm before converting this algorithm to optimized C code.

7. STREAMING VIDEO AND IMAGE PROCESSING

The primary technical issues related to performance in real-time streaming video image processing performance are to create low-latency paths in the hardware, operating system, and application software to achieve an acceptable frames-per-second (fps) rate. There are several stages of processing required to make the pixels captured by a camera visible on a video display. The delays contributed by each of these processing steps produce the total delay, known as *end-to-end latency*. But, the biggest contributors to video latency are the processing stages that require temporal storage of data (i.e. buffering), color space conversion (encoding) and image processing. Converting from frames to time depends on the video's frame rate. For example, a delay of one frame in 30 fps video corresponds to 1/30th of a second (33.3ms) of latency.

The following figure illustrates the pseudo code for the software developed for this research. The full source code listing can be found in Appendix A (sv5.c).

```
/*Initialization*/
Frame Buffer Initialization (/dev/fb0)
Camera Initialization (/dev/video0)
YUV422 to RGB565 Look Up Table Initialization
Performance Measurement Initialization
/*Streaming Video and Image Processing Loop*/
while(true){
  Get Frame from Camera
  Convert from YUV422 to RGB565
  Perform Haar DWT Image Processing
  Display Image
}/*eo while*/
Cleanup
```

Figure 5 – Streaming Video and Image Processing Program Flow

7.1 FRAME BUFFER INITIALIZATION

During frame buffer initialization, the frame buffer device (`/dev/fb0`) is opened and its properties are read into two data structures using `ioctl()` [48]. Properties include *x* by *y* pixel screen resolution (i.e. 480 x 272), the horizontal line length based on the number of pixels in a line times the pixel size (480 x 2 bytes/pixel = 960 bytes); and the number of bits per pixel. In this case, the frame buffer device supports a pixel size of 16-bits per pixel using the RGB565 color space. The pixel size and color space characteristics are defined by the hardware design of the AM335X processor and need to be supported by the chosen LCD display.

Finally the frame buffer itself is created by `mmap()` [49][50] and a file handle to the frame buffer is returned from the kernel to the user-space application. The `mmap()` function call will be used frequently in this work. The `mmap()` function provides a user-space interface to the kernel that allows an application to map a file or a device into virtual memory. The programmer can then access the file or device directly through memory, identically to any other chunk of memory-resident data. Using `mmap()` it is also possible to allow writes from the memory region to directly to a file on disk. This feature was a feature used in generating the YUV422 to RGB565 color space conversion table used in this work (Appendix B). The following figure illustrates the code snippet used to initialize the frame buffer.

```
/* data structures */
struct fb_fix_screeninfo finfo;
struct fb_var_scrconinfo vinfo;
/* oper framebuffer device */
fb_fd = open("/dev/fb0",O_RDWR);
/* get variable frame buffer properties*/
ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);
/* got fixcd framc buffcr proportics */
ioctl(fb_fd, FBIOGET_FSCREENINFO, &finfo);
/* set variable frame buffer properties */
ioctl(fb_fd, FBIOPUT_VSCREENINFO, &vinfo);
/* allocate framebuffer and map to kernel */
screensize = vinfc.yres_virtual *
fin.lire_length;
fbp = mmap(0, screensize, PRCT_READ | PROT_WRITE,
MAP_SHARED, fb_fd, 0);
```

Figure 6 – Frame Buffer Initialization Code Snippet

7.2 CAMERA INITIALIZATION

The Logitech C920 Webcam is initialized in this step. The camera device is opened (`/dev/video0`) and a file handle is returned to the application. Next a call is made to V4L2 using `ioctl()` to get several data structure that enable the program to get and set the properties of the camera including capabilities, format and buffer use. For example, the application can determine if the camera supports streaming video and what types of memory buffers the camera supports. In addition, camera properties can be set by the application using V4L2. In this case, the application tells the camera to use the YUV422 color space to capture images. The reason the YUV422 color space was chosen is because unlike H.264 and MJPG, which is also supported by the C920, YUV422 is not compressed. This is advantageous since it would take extra processing to decompress the H.264 or MJPEG image data captured by the camera. Instead, YUV422 raw image data is utilized.

Another camera property set by the application is the pixel width and height. Recall that the LCD display being used has a resolution of 480 x 272 (HVGA), However, the closest to HVGA resolution provided by the camera is WQVGA which has a resolution of 432 x 240. The application program tells the camera to use WQVGA at 30 fps which is the maximum frame rate that can be provided at any resolution by the C920.

V4L2 can support a number of memory models (e.g. DMA, memory mapped, etc.) and multiple video buffers. In this case, a call to `ioctl()` is used to request that one memory mapped buffer be allocated for use by the application for video capture. Next, `mmap()` is used to get a file pointer to a kernel space buffer used by the C920 to place image data that can be used by the user space application. Remember that this buffer will contain raw YUV422 432 x 240 color space image data that will need to be converted to RGB565 color space data for use by the 480 x 272 display. Finally, `ioctl()` informs V4L2 that the camera is ready to begin streaming video. The following illustrates the code snippet used during camera initialization.

```
/* data structures */
struct v4l2_capability v4l2_cap;
struct v4l2_format v4l2_fmt;
struct v4l2_requestbuffers v4l2_reqbuf;
struct v4l2_buffer v4l2_buf;
/* open camera */
vid_fd = open("/dev/video0", O_RDWR );
/* get camera capabilities */
ioctl(vid_fd, VIDIOC_QUERYCAP, &v4l2_cap);
/* set camera format */
ioctl(vid_fd, VIDIOC_S_FMT, &v4l2_fmt);
/* request buffers */
ioctl(vid_fd, VIDIOC_REQBUFS, &v4l2_reqbuf);
/* query buffer status */
ioctl(vid_fd, VIDIOC_QUERYBUF, &v4l2_buf);
/* allocate webcam buffer and map to kernel */
cbp = mmap(NULL, v4l2_buf.length,PROT_READ |
PROT_WRITE, MAP_SHARED, vid_fd, v4l2_buf.m.offset);
/* turn on video streaming */
ioctl(vid_fd, VIDIOC_STREAMON, &v4l2_reqbuf.type);
```

Figure 7 – Camera Initialization Code Snippet

7.3 YUV422 TO RGB565 LOOK UP TABLE INITIALIZATION

First, a word about color spaces is in order. A range of colors can be created by the primary colors of pigment and these colors then define a specific color space. Color space is an abstract mathematical model which simply describes the range of colors as tuples of numbers, typically as 3 or 4 values or color components (e.g. RGB; R=Red, G=Green, B=Blue). Each color in the system is represented by a single pixel. RGB is a color space which uses red, green and blue to elaborate a color model. An RGB color space can simply be interpreted as all possible colors which can be made up from three colors for red, green and blue. In the case of RGB888, each pixel is 24 bits (3 bytes), made up of a 8-bit red value from 0-255, a 8-bit green value from 0-255 and a 8-bit blue value from 0-255. As a result RGB888 has 256 x 256x 256 or 16,777,216 colors available in its color space.

On the other hand, RGB565, which is the color space supported by the AM335X processor, is made up of a 5-bit red value from 0-31, a 6-bit green value from 0-63, and a 5-bit blue value from 0-31. RGB565 therefore, has 32 x 64 x 32 or 65,536 colors available in its color space. Each RGB565 pixel is 16 bits (2 bytes).

The C920 camera captures images in the YUV422 (YCbCr) color space. Unlike RGB, YUV422 defines a 4 byte macro-pixel (U0,YO,VO,Y1). Each 4 byte macro-pixel represents 2 image pixels. The first 2 byte pixel is calculated using Y0,U0,V0. The second 2 byte pixel is calculated using Y1,U0,V0. Y0 and Y1 are luminance values (Y) that can be used alone to produce a grayscale image. Color (chrominance - CbCr) is added to the luminance value through the U0 and V0. U0 is also referred to as Cr which is the red value and V0 is also referred to as Cb which is the blue value. In order to convert YUV422 to RGB888 or RGB565, there are three formulas (one each for red, green and blue) using floating point arithmetic that need to be calculated. These calculations are specified by the ITU-R 601 standard [51]. Initially, floating point calculations were used to convert the YUV422 image from the camera to a RGB565 image to be used by the frame buffer to display the image. As will be shown in the PERFORMANCE RESULTS section of this Thesis, there was a significant performance impact using the floating point routines. In order to optimize the YUV422 to RGB565 conversion, a program (lut.c – Appendix B) was written to generate a look up table to perform the conversion. This look up table is 33,554,432 bytes (33MB) in size.

During initialization, the yuv2rgb.lut file is read into virtual memory using `mmap()`. In order to ensure that all 33MB of the table is read into memory so that the application doesn't rely on virtual memory page swaps to access the table during run time which would decrease performance, a for loop is executed to read the entire table into memory one page (4096 bytes) at a time until all pages (8,192 pages) are read. The `convert3()` function executed in the streaming video and image processing loop then simply uses elegant, yet simple, pointer arithmetic to quickly convert one YUV422 macro pixel into two RGB565 pixels. The following figure illustrates the YUV422 to RGB565 look up table initialization.

```
/* read yuv2rgb.lut into virtual memory for random
access*/
lut_fd = open("/home/root/yuv2rgb.lut", O_RDWR);
uint16_t *lut_ptr = mmap(NULL, lut_size, PROT_READ,
MAP_PRIVATE | MAP_POPULATE, lut_fd, 0);
/* use madvise() for mmap() performance improvement*/
madvise(lut_ptr, lut_size, MADV_WILLNEED);
/* pre-read table into memory to avoid virtual memory
read access latency during streaming */
for(i=0; i<lut_size/4096; i++){
        lseek(lut_fd, (long)(i*4096), SEEK_SET);
        read(lut_fd, lut_buf, 1);
}/*eo for*/
```

Figure 8 – YUV422 to RGB565 Look Up Table Initialization Code Snippet

7.4 PERFORMANCE MEASUREMENT INITIALIZATION

The application code for this research was instrumented with the `clock_gettime()` software function [52] to record the processing time intervals for several discrete events which include: 1) initialization, 2) capture frame, 3) encode frame, 4) image processing; and 5) display frame. This was accomplished by simply commenting out the function that didn't need to be included in the measurement, recompiling the source code (Appendix A) and then taking the difference between the cumulative of time of each specific event using a sample size of 100. This data then was used in latency and frame-per-second (fps) calculations. Performance data was taken several times during the development phase of this work and analyzed to make additional design tradeoffs and optimizations. The results of the performance data can be reviewed in the PERFORMANCE RESULTS section of this Thesis.

The software approach used to collect performance data itself is subject to a certain amount of latency, but presumed not to be large enough to have a dramatic impact on performance. However, it would have been useful to use a simple port bit and measure performance using an oscilloscope. This effort has been left open for investigation. The following code snippet shows the initialization and execution of performance measurement.

```
/* data structures */
struct timespec fps_start_time,fps_end_time,
init_time_start,init_time_end;
/* begin initialization */
if (tlog) clock_gettime(CLOCK_REALTIME, &init_time_start);
/* perform initialization steps … */
if(tlog) clock_gettime(CLOCK_REALTIME, &init_time_end);
/* end of initialization
if(tlog) clock_gettime(CLOCK_REALTIME, &fps_start_time);
/* streaming loop */
while(true){
     /* streaming */
}/*eo while*/
if(tlog){
     clock_gettime(CLOCK_REALTIME, &fps_end_time);
     /* get initialization time */
     t_diff = (init_time_end.tv_sec - init_time_start.tv_sec)
     + (double)(init_time_end.tv_nsec -
     init_time_start.tv_nsec)/1000000000.0d;
     t_diff = t_diff/SAMPLE_SIZE;
     printf("initialization time: %f sec\n", t_diff);
     /* above code repeated  for fps time(not shown) */
}/*eo if*/
```

Figure 9 – Performance Measurement Initialization Code Snippet

7.6 STREAMING VIDEO AND IMAGE PROCESSING LOOP

At this point, the application enters an infinite while loop unless performance recording is turned on (`tlog=1`) that: 1) gets a frame from the camera, 2) converts the frame from the YUV422 to RGB565 color space using a look up table, 3) optionally converts the streaming video image using a Haar DWT; and 4) places the image in the frame buffer for display on the LCD display. Depending upon the options that have been selected, the display will show either normal streaming camera video or a Haar DWT conversion of the streaming camera video.

7.6.1 GET FRAME FROM CAMERA

An `ioctl()` function using V4L2 is given a buffer for the camera to use. This call waits until an image has been captured and is in the buffer before completing. Once complete, a second `ioctl()` function using V4L2 is used to read the camera image from the buffer. The image captured by the camera is now ready to be converted from the YUV422 color space to the RGB565 color space. The following illustration shows the capture of a frame from the camera.

```
/* provide buffer and wait */
ioctl(vid_fd, VIDIOC_QBUF, &v4l2_buf);
/* get filled buffer */
ioctl(vid_fd, VIDIOC_DQBUF, &v4l2_buf);

/* yuv422 to rgb565 color space conversion */
convert3(cbp, rgb565ptr, lut_ptr);
/* haar dwt image processing */
HaarDwt((uint16_t*)rgb565ptr, imgout_ptr);
/* display image */
display_LCD4(fbp,(uint8_t*)imgout_ptr);
```

Figure 10 - Get Frame from Camera

Several options including WAIT, NOWAIT and the use of DMA buffers versus memory mapped buffers were considered when capturing frames in order to ensure that there would not be a negative performance impact based on the V4L2 capture approach used. These approaches were investigated during CAMERA INITIALIZATION. It was determined that the application program does need to WAIT until the camera buffer is available before it can be de-queued for subsequent processing. Use of the NOWAIT option during `open()`caused a buffer not ready error and the use of DMA buffers instead of memory mapped buffers did not increase performance. This is likely due to the resource constrained nature of the hardware being used in this research. A consistent delay of 34ms to capture a frame was observed under all optimization scenarios. In summary, memory mapped buffers (`v4l2_reqbuf.memory = V4L2_MEMORY_MMAP`) and opening the camera device with the WAIT option (`fd = open("/dev/video0", O_RDWR)`)was used in this work providing acceptable performance and reliability. Further optimization of camera to frame capture latency has been left open for investigation.

7.6.2 CONVERT FROM YUV422 TO RGB565

The `convert3()`function takes the YUV422 (Y0,U0,V0) image from the camera (`*cbp`) and converts it to an RGB565 image for the display using the following pointer arithmetic to quickly access a color space conversion look up table where `pbuf` is the base table pointer (location zero) and `rgb565` is the 2 byte RGB565 pixel returned by the look up table: `rgb565 = *(pbuf + ((Y0*256*256)+(U0*256)+V0);`

```
int convert3(void *cbp, uint8_t *rgb565ptr, uint16_t *pbuf){
     /* cbp - camera buffer, rgb565ptr - display buffer, pbuf - yuv422 to rgb565 lut */
     int i=0; uint8_t *yuvptr = (uint8_t *)cbp; uint8_t Y0,Y1,U0,V0;
     uint16_t rgb565=0xffff;
     /* convert yuv422 camera buffer to rgb565 display buffer */
     for(i=0; i<YUYV_SIZE; i++){
          Y1 = *yuvptr;  /*Y1*/
          i++; yuvptr++;
          V0 = *yuvptr;  /*V0*/
          i++; yuvptr++;
          Y0 = *yuvptr;  /*Y0*/
          i++; yuvptr++;
          U0 = *yuvptr;  /*U0*/
          yuvptr++;      /*next 4 byte macropixel*/
          /* convert yuv422 to rgb565 */
          rgb565 = *(pbuf + ((Y0*256*256)+(U0*256)+V0));
          *rgb565ptr = rgb565;
          rgb565ptr++;
          *rgb565ptr = rgb565 >> 8;
          rgb565ptr++;
          rgb565 = *(pbuf + ((Y1*256*256)+(U0*256)+V0));
          *rgb565ptr = rgb565;
          rgb565ptr++;
          *rgb565ptr = rgb565 >> 8;
          rgb565ptr++;
     }/*eo for*/
     return(0);
}/*eo convert3*/
```

Figure 11 – Convert YUV422 to RGB565 Using a Look Up Table

Initially, the pointer arithmetic of the look up table (figure 11) were replaced in the `convert3()` function with a call to a floating point color space conversion routine that converts YUV422 to RGB888 then to RGB565. This same conversion function was used to generate the look up table that was ultimately used in this work (Appendix B). The following figure shows the details of the floating point color space conversion algorithm.

```
/*
** ITU-R 601
** convert yuv422 to rgb888
*/
r = 1.164*(float)(Y-16) + 1.596*(float)(V-128);
if(r<0) r=0;
if(r>255) r=255;
red = (uint8_t)r;

g = 1.164*(float)(Y-16) - 0.813*(float)(V-128) - 0.391*(float)(U-128);
if(g<0) g=0;
if(g>255) g=255;
green = (uint8_t)g;

b = 1.164*(float)(Y-16) + 2.018*(float)(U-128);
if(b<0) b=0;
if(b>255) b=255;
blue = (uint8_t)b;

/*
** convert rgb888 to rgb565
*/
r16 = ((red >>3) & 0x1f) << 11;
g16 = ((green >> 2) & 0x3f) << 5;
b16 = (blue >> 3) & 0x1f;
rgb565 = r16 | g16 | b16;
```

Figure 12 – YUV422 to RGB565 Floating Point Conversion Algorithm

The impact of performance using floating point calculations in a resource constrained environment is severe. In this work, color space conversion latency decreased from 235ms to 66ms at a 600Mhz CPU clock speed. This was accomplished by substituting the use of a CPU cycle intensive floating point algorithm to a look up table memory access cycle mechanism. The reader is referred to the Schlessman et al., paper entitled "*Tailoring Design for Embedded Computer Vision Applications*" [14] for a further discussion on modeling the tradeoffs between floating point and integer data type use in embedded applications.

7.6.3 PERFORM HAAR DWT IMAGE PROCESSING

The `HaarDWT()` function takes a raw RGB565 image, performs an optimized Haar Discrete Wavelet Transform (DWT) image processing operation and returns the result in a raw RGB565 format for rendering on the LCD display. A standalone version of the optimized Haar DWT algorithm using the standard Lena test image [53] is available for review in Appendix C.

A wavelet transform decomposes a time-frequency signal into a set of frequency (basis) functions known as wavelets. Wavelets (i.e. a small wave) provide a very simple and efficient way to analyze signals in the time-frequency domain. Wavelets can be used for many different purposes including audio, image and video compression; speech recognition, de-noising signals; and motion detection and tracking. JPEG-2000 is a popular example of an image compression algorithm that uses a DWT.

A Discrete Wavelet Transform separates the high and low frequency portions of a signal through the use of filters. A one level DWT passes a signal through a high pass (H) and low pass (L) filter producing two signals which are then downsized by a factor of two. For example, a DWT of an image pixel size x by y produces two images each $x/2$ by $y/2$. A second level or 2-DWT would repeat the process again creating four $x/4$ by $y/4$ images: low-low (LL), low-high (LH), high-low (HL), high-high (HH).

The Haar DWT is one of the more efficient wavelet transformations. A Haar DWT decomposes each signal into two components, one is called the average (approximation) and the other is known as the difference (detail). The Haar DWT has a number of advantages: 1) it is conceptually simple and fast; and 2) it is memory efficient since it can be calculated in place without a temporary array. The following block diagram shows the Haar DWT.

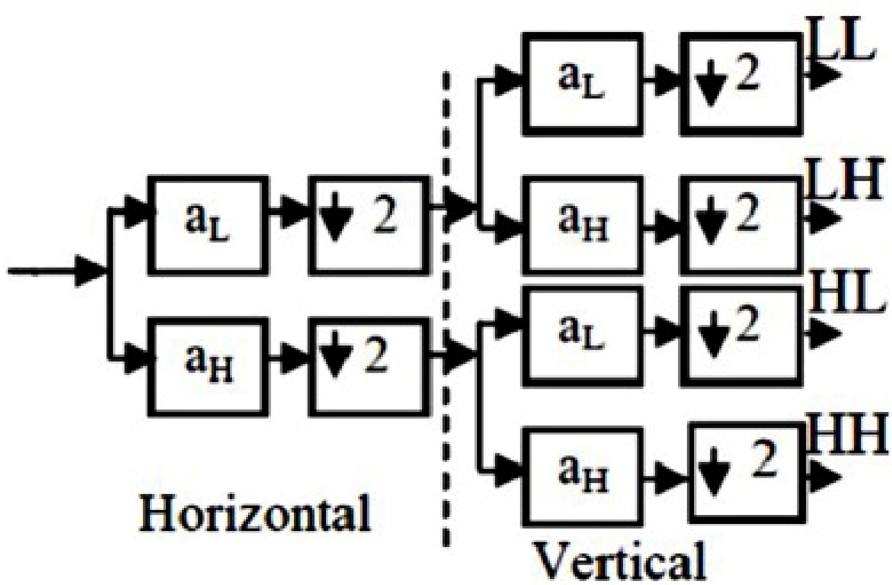


Figure 13 - Haar DWT Block Diagram

The Haar DWT was used in this research to benchmark real-time streaming video image processing performance. The main challenge was to write an optimized Haar DWT image processing algorithm that would maximize fps results. In order to ensure that the correct results were achieved, I initially used the MATLAB `HaarDwt()` function, then wrote a discrete Haar DWT function in MATLAB, converted the MATLAB logic to the C language and then optimized the C code (Appendix C) for the best performance possible. I used the standard Lena test image so I could compare my results with the work of other experts in the field to ensure my results were correct. Finally, the optimized Haar DWT code was integrated as a function in the main source code for this research

(Appendix A). The following figure illustrates the standard Lena test image used as input to the optimized Haar DWT algorithm (Appendix C) and the results that were obtained.

Standard Lena Reference Image

Haar DWT Lena
(using optimized algorithm - Appendix C)

Figure 14 - Haar DWT

7.6.3.1 Haar 2-DWT RGB565 Algorithm Optimization

The Haar 2-DWT algorithm uses integer calculations and is optimized by using two nested loops – an outer loop used simply to index the image row count and an inner loop that performs the transform by processing two rows and two columns at a time using a sliding window R1C1, R1C2, R2C1 and R2C2 until all the rows and columns of the image are processed. The algorithm within the same inner loop: 1) calculates the sliding window, 2) separates the RGB color channels by splitting the RGB565 sliding window pixels into individual red, green and blue values; 3) performs the LL, LH, HL, and HH transform using the sliding window for each red, green and blue value; 4) packs the result back into four separate LL, LH, HL, and HH RGB565 pixels; and 5) stores the results in a memory buffer organized with four quadrants, one for each LL, LH, HL and HH result. The quadrant buffer is then passed to the `Display_LCD()` function for rendering on the LCD display. The following figure illustrates a pseudo code summary of the optimized Haar DWT algorithm. Each of the individual processing steps will be discussed in the next few sections of this paper.

```
/* process two rows at a time until all rows processed */
for (i=0; i<NUM_ROWS; i++){
    Calculate Quadrant Row Offset (QUAD_ROW_OFFSET*h)
    Set Output Buffer Column to Zero (k=0)
    /* process two columns at a time until all columns processed */
    for(j=0; j<NUM_COLS; j++){
        Get Sliding Window R1C1,R1C2,R2C1,R2C2
        Point To Beginning of Next Two Columns (j++)
        Unpack RGB565 Pixels into Red, Green, Blue
        Calculate Red LL,LH,HL,HH Pixels
        Calculate Green LL,LH,HL,HH Pixels
        Calculate Blue LL,LH,HL,HH Pixels
        Pack Into LL,LH,HL,HH RGB565 Pixels
        Put LL,LH,HL,HH RGB565 Pixels into Quadrant Buffer
        Point To Next Column In Output Buffer (k++)
    }/*eo for*/
    Point To Beginning Of Next Two Input Buffer Rows (i++)
    Point To Next Row In Output Buffer (h++)
}/*eo for*/
```

Figure 15 - Haar 2-DWT RGB565 Optimized Algorithm Pseudo Code

7.6.3.2 Get Sliding Window R1C1, R1C2, R2C1, R2C2

A sliding window R1C1, R1C2, R2C1 and R2C2 is constructed using pointer arithmetic to transform the RGB565 image (`imgin_ptr`) two rows and two columns at a time until all of the rows and columns of the image are processed. The following figure illustrates how the sliding window is generated.

```
/*
** haar dwt
** this haar dwt uses a sliding window r1c1,r1c2,r2c1,r2c2
** to traverse the entire rgb565 image.
**
** H_ROW_WIDTH*i points to the first row (r1)
** H_ROW_WIDTH*(i+1) points to the second row (r2)
** index j points to the first column (c1) in the row (r1,r2)
** index j+1 points to the second column (c2) in the row (r1,r2)
*/
for(j=0; j<NUM_COLS; j++){
    /*
    ** sliding window
    ** get rgb565 pixels r1c1,r1c2,r2c1,r2c2
    */
    r1c1 = *(imgin_ptr+(((H_ROW_WIDTH*i)+j)));          //r1c1
    r1c2 = *(imgin_ptr+(((H_ROW_WIDTH*i)+(j+1))));      //r1c2
    r2c1 = *(imgin_ptr+(((H_ROW_WIDTH*(i+1))+j)));      //r2c1
    r2c2 = *(imgin_ptr+(((H_ROW_WIDTH*(i+1))+(j+1)))); //r2c2
```

Figure 16 - Get Sliding Window R1C1 ,R1C2 ,R2C1 ,R2C2

7.6.3.3 Unpack RGB565 Pixels into Red, Green, Blue

The next step in the algorithm unpacks the RGB565 pixels contained in the sliding window into separate red, green and blue color channels. The following figure illustrates the RGB565 unpacking operation.

```
for(j=0; j<NUM_COLS; j++){
    Get Sliding Window R1C1,R1C2,R2C1,R2C2
    /*
    ** unpack rgb565 pixels into red, green, blue
    */
    red_r1c1 = (((r1c1 & 0xf800)>>3)>>8);
    grn_r1c1 = ((r1c1 & 0x07e0)>>5);
    blu_r1c1 = (r1c1 & 0x001f);

    red_r1c2 = (((r1c2 & 0xf800)>>3)>>8);
    grn_r1c2 = ((r1c2 & 0x07e0)>>5);
    blu_r1c2 = (r1c2 & 0x001f);

    red_r2c1 = (((r2c1 & 0xf800)>>3)>>8);
    grn_r2c1 = ((r2c1 & 0x07e0)>>5);
    blu_r2c1 = (r2c1 & 0x001f);

    red_r2c2 = (((r2c2 & 0xf800)>>3)>>8);
    grn_r2c2 = ((r2c2 & 0x07e0)>>5);
    blu_r2c2 = (r2c2 & 0x001f);
```

Figure 17 - Unpack RGB55 Pixels into Red, Green, Blue

7.6.3.4 Calculate Red, Green, Blue LL, LH, HL, HH Pixels

The Haar DWT transform is calculated for each red, green, and blue value in the current sliding window. If the result of the LL, LH, HL, or HH filter calculation is greater than the maximum value of the individual red, green or blue pixel values, the result is set to the maximum value for the specific color channel. For red and blue the maximum value is 31 (0x1f); for green the maximum value is 63 (0x3f). The absolute value of the LH, HL and HH filter calculations is used to make sure the results are not less than zero. Finally, the result of the LH, HL, and HH filter calculations is multiplied by ten (10) to make the results more visible to the viewer. The following shows the formulas used to calculate the LL, LH, HL and HH filter values:

```
LL =     (((R1C1 + R2C1)/2) + ((R1C2 + R2C2)/2)/2)
LH = abs (((R1C1 + R2C1)/2) - ((R1C2 + R2C2)/2)/2) x 10
HL = abs (((R1C1 - R2C1)/2) + ((R1C2 - R2C2)/2)/2) x 10
HH = abs (((R1C1 - R2C1)/2) - ((R1C2 - R2C2)/2)/2) x 10
```

```
for(j=0; j<NUM_COLS; j++){
    Get Sliding Window R1C1,R1C2,R2C1,R2C2
    Point To Beginning of Next Two Columns (j++)
    Unpack RGB565 Pixels into Red,Green,Blue
    /* calculate red LL,LH,HL,HH pixels (repeat for green, blue - not shown) */
    /* low-low (LL) */
    lp1 = (red_r1c1+red_r2c1)/2; if(lp1>0x1f) lp1=0x1f;
    lp2 = (red_r1c2+red_r2c2)/2; if(lp2>0x1f) lp2=0x1f;
    lp3 = (lp1+lp2)/2; if(lp3>0x1f) lp3=0x1f;
    red_ll = lp3;
    /* low-high (LH) */
    hp1 = abs((lp1-lp2)/2); if(hp1>0x1f) hp1=0x1f;
    red_lh = hp1*10;
    /* high-low (HL) */
    hp1 = abs((red_r1c1-red_r2c1)/2); if(hp1>0x1f) hp1=0x1f;
    hp2 = abs((red_r1c2-red_r2c2)/2); if(hp2>0x1f) hp2=0x1f;
    lp1 = (hp1+hp2)/2; if(lp1>0x1f) lp1=0x1f;
    red_hl = lp1*10;
    /* high-high (HH) */
    hp3 = abs((hp1-hp2)/2); if(hp3>0x1f) hp3=0x1f;
    red_hh = hp3*10;
```

Figure 18 - Calculate Red, Green, Blue LL, LH, HL, HH Pixels

7.6.3.5 Pack into LL, LH, HL, HH RGB565 Pixels

The individual results from the red, green and blue LL, LH, HL, and HH filter calculations are packed back into a complete RGB565 LL, LH, HL, and HH 16-bit pixel and are then stored in the quadrant memory buffer which is shown in the following figures.

```
for(j=0; j<NUM_COLS; j++){
     Get Sliding Window R1C1,R1C2,R2C1,R2C2
     Point To Beginning of Next Two Columns (j++)
     Unpack RGB565 Pixels into Red,Green,Blue
     Calculate Red,Green,Blue LL,LH,HL,HH Pixels
     /*
     ** pack into ll,lh,hl,hh rgb565 pixels
     */
     rgb565_ll = ((red_ll << 11) | (grn_ll << 5) | (blu_ll));
     rgb565_lh = ((red_lh << 11) | (grn_lh << 5) | (blu_lh));
     rgb565_hl = ((red_hl << 11) | (grn_hl << 5) | (blu_hl));
     rgb565_hh = ((red_hh << 11) | (grn_hh << 5) | (blu_hh));
```

Figure 19 - Pack into LL, LH, HL, HH RGB565 Pixels

7.6.3.6 Put RGB565 LL, LH, HL, HH Pixels into Quadrant Buffer

```
/* process two rows at a time until all rows processed */
for (i=0; i<NUM_ROWS; i++){
    Calculate Quadrant Row Offset (QUAD_ROW_OFFSET*h)
    Set Output Buffer Column to Zero (k=0)
    /* process two columns at a time until all columns processed */
    for(j=0; j<NUM_COLS; j++){
        Get Sliding Window R1C1,R1C2,R2C1,R2C2
        Point To Beginning of Next Two Columns (j++)
        Unpack RGB565 Pixels into Red,Green,Blue
        Calculate Red,Green,Blue LL,LH,HL,HH Pixels
        Pack into LL,LH,HL,HH RGB565 Pixels
        *(imgout_ptr+(quad_row_offset+k))=rgb565_ll;
        *(imgout_ptr+(QUAD_COL_OFFSET+(quad_row_offset+k)))=rgb565_lh;
        *(imgout_ptr+((QUAD_ROW_ORIGIN)+(quad_row_offset+k)))=rgb565_hl;
        *(imgout_ptr+((QUAD_ROW_ORIGIN)+QUAD_COL_OFFSET+(quad_row_offset+k)))=rgb565_hh;
        Point to Next Column In Output Buffer (k++)
    }/*eo for*/
    Point To Beginning Of Next Two Input Buffer Rows (i++)
    Point To Next Row In Output Buffer (h++)
}/*eo for*/
```

Figure 20 - Put RGB565 LL, LH, HL, HH Pixels into Quadrant Buffer

7.6.4 DISPLAY IMAGE

The `display_LCD4()` function copies the RGB565 image to the frame buffer where it is instantaneously displayed on the LCD panel. The `display_LCD4()` function also needs to accommodate the difference in size between the WQVGA (432 x 240) format provided by the camera and the HVGA (480 x 272) format supported by the display by simply centering the image on the display. There is no significant impact on the display function that needed to be considered during this research. The following shows the `display_LCD4()` function.

```
int display_LCD4(void *fbp, void *filebuf){
     int i=0,j=0,idx=0;
     uint16_t *fbptr = fbp;
     uint16_t *filebuf_ptr = filebuf;

     idx = HVGA_WIDTH*((HVGA_HEIGHT-WQVGA_HEIGHT)/2);
     /* copy display buffer to frame buffer */
     for(i=0; i<WQVGA_HEIGHT; i++){      /*row*/
          idx=idx+((HVGA_WIDTH-WQVGA_WIDTH)/2);
          for(j=0; j<WQVGA_WIDTH; j++){ /*col*/
               *(fbptr+idx)=*filebuf_ptr;
               idx++;
               filebuf_ptr++;
          }/*eo for*/
          idx=idx+((HVGA_WIDTH-WQVGA_WIDTH)/2);
     }/*eo for*/
     return(0);
}/*eo Display_LCD4*/
```

Figure 21 – `Display_LCD()`

7.7 CLEANUP

During the cleanup phase of the application program, streaming is deactivated, the camera (`/dev/video0`) and frame buffer (`/dev/fb0`) devices and look up table file are closed and all allocated memory is released back to the operating system. The program then exits to the operating system.

8. PERFORMANCE RESULTS

The BeagleBone Black CPU supports clock speeds of 300 MHz, 600 MHz, 800 MHz, and 1000 MHz all under programmatic control. When taking the performance measurements for this research, the clock speed settings were set manually and then verified using the `cpufreq-info` command from the Linux console.

The following figure shows how the clock frequency was set to 600 Mhz by using the `echo` command from the console and then checked with the `cpufreq-info` command.

```
root@beaglebone:/sys/devices/system/cpu/cpu0/cpufreq# echo "userspace" > scaling_governor
root@beaglebone:/sys/devices/system/cpu/cpu0/cpufreq# echo "600000" > scaling_max_freq
root@beaglebone:/sys/devices/system/cpu/cpu0/cpufreq# echo "600000" > scaling_min_freq
root@beaglebone:/sys/devices/system/cpu/cpu0/cpufreq# echo "600000" > scaling_setspeed
root@beaglebone:/sys/devices/system/cpu/cpu0/cpufreq# cpufreq-info

cpufrequtils 008: cpufreq-info (C) Dominik Brodowski 2004-2009
Report errors and bugs to cpufreq@vger.kernel.org, please.
analyzing CPU 0:
  driver: generic_cpu0
  CPUs which run at the same hardware frequency: 0
  CPUs which need to have their frequency coordinated by software: 0
  maximum transition latency: 300 us.
  hardware limits: 300 MHz - 1000 MHz
  available frequency steps: 300 MHz, 600 MHz, 800 MHz, 1000 MHz
  available cpufreq governors: conservative, ondemand, userspace, powersave, performance
  current policy: frequency should be within 600 MHz and 600 MHz.
                  The governor "userspace" may decide which speed to use
                  within this range.
  current CPU frequency is 600 MHz (asserted by call to hardware).
  cpufreq stats: 300 MHz:nan%, 600 MHz:nan%, 800 MHz:nan%, 1000 MHz:nan%
root@beaglebone:/sys/devices/system/cpu/cpu0/cpufreq#
```

Figure 22 - Set/Check CPU Frequency

Measurements of latency and fps were taken for each of the available CPU clock frequencies upon completion of the following events: 1) initialization, 2) camera image capture, 3) encoding – image color space conversion from YUV422 to RGB565, 4) Haar DWT image processing, and 5) display image. A sample size of 100 was used in the latency and fps calculations.

8.1 FLOATING POINT COLOR SPACE CONVERSION

The initial performance measurement of the system was limited to basic streaming video without any image processing. The clear bottleneck was observed to be the use of the floating point algorithm used for YUV422 to RGB565 color space conversion. Initialization and display time were seen as minimal and well within acceptable limits. A consistent frame capture time of 34ms was observed across all CPU clock frequencies which aroused curiosity and is a topic of further investigation, but was felt acceptable to the overall latency and fps performance of the system. At this point, overall performance was deemed unacceptable and lead to the investigation of: 1) an alternative color space conversion approach, and 2) the use of compiler optimization. The following table illustrates the initial basic streaming video performance of the system.

Event	CPU Speed			
	300 Mhz	600 Mhz	800 Mhz	1000 Mhz
Initialization	.0023s	.0019s	.0018s	.0017s
Capture Frame	.034s (29.4 fps)	.034s (29.4 fps)	.034s (29.4 fps)	.034s (29.4fps)
Encode Frame	1.31s (0.76 fps)	.457s (2.2 fps)	.333s (3.0 fps)	.251s (4.0 fps)
Display Frame	**1.36s (0.73 fps)**	**.468s (2.1 fps)**	**.333s (3.0 fps)**	**.266s (3.8 fps)**

Table 4 - Basic Streaming Video
(Performance Using Floating Point Calculations Without Compiler Optimization)

The C920 Webcam as configured for use in this research provides frames at a rate of 30 frames-per-second (fps). The following graph shows the theoretical 30 fps line in black and plots the measurement events (capture, encode, display, etc.) on the graph at the CPU clock speeds of 300 MHz (red plot), 600 MHz (green plot), 800 MHz (blue plot) and 1000 MHz (magenta plot).

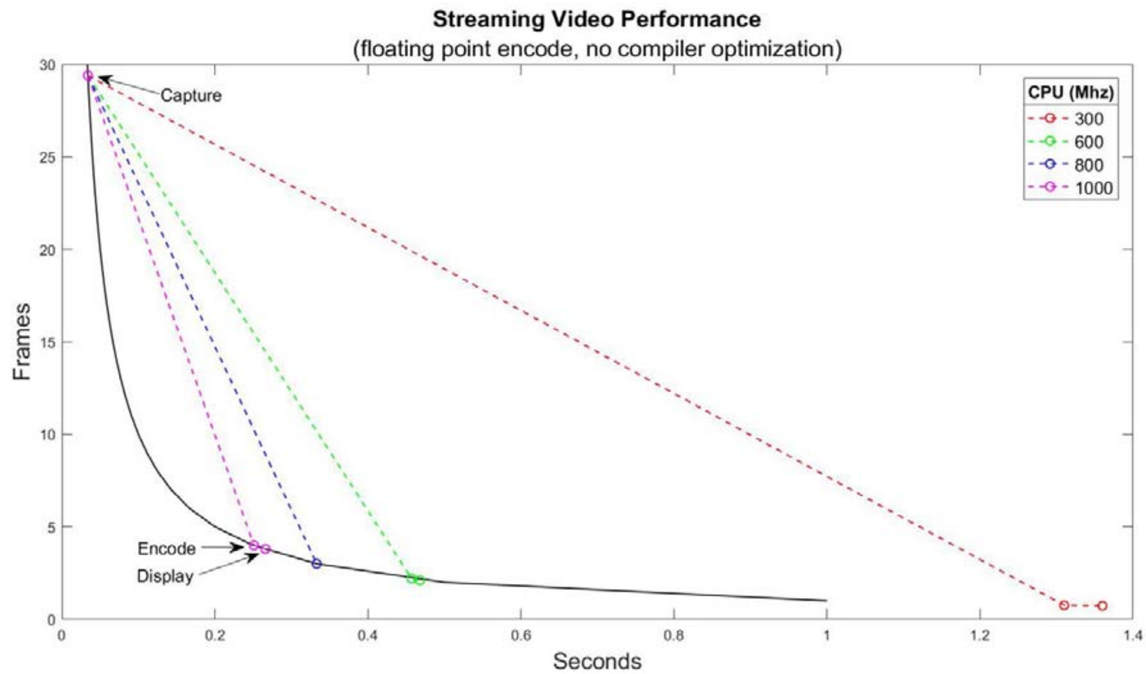


Figure 23 – Basic Streaming Video Performance (Table 4)

8.2 COMPILER OPTIMIZATION

The simple use of compiler optimization [54][55] (`gcc -O3 sv5.c -o sv5 -lrt`) had an immediate positive impact on the performance results. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. The compiler performs optimization based on the knowledge it has of the program and using the `-O3` option includes an aggressive set of optimizations that incur a space-time tradeoff in favor of time, such as loop unrolling and automatic function in-lining. The following table shows a significant increase in basic streaming video performance using floating point color space conversion from 3.8 (Table 4) to 6.0 fps at 1000 MHz.

Event	CPU Speed			
	300 Mhz	600 Mhz	800 Mhz	1000 Mhz
Initialization	.0023s	.0020s	.0018s	.0017s
Capture Frame	.0340s (29.4 fps)	.0340s (29.4 fps)	.0340s (29.4 fps)	.0340s (29.4 fps)
Encode Frame	0.768s (1.3 fps)	0.269s (3.7 fps)	0.201s (5.0 fps)	0.167s (5.9 fps)
Display Frame	**0.795s (1.2 fps)**	**0.288s (3.5 fps)**	**0.201s (4.9 fps)**	**0.167s (6.0 fps)**

Table 5 – Basic Streaming Video
(Performance Using Floating Point Calculations And Compiler Optimization)

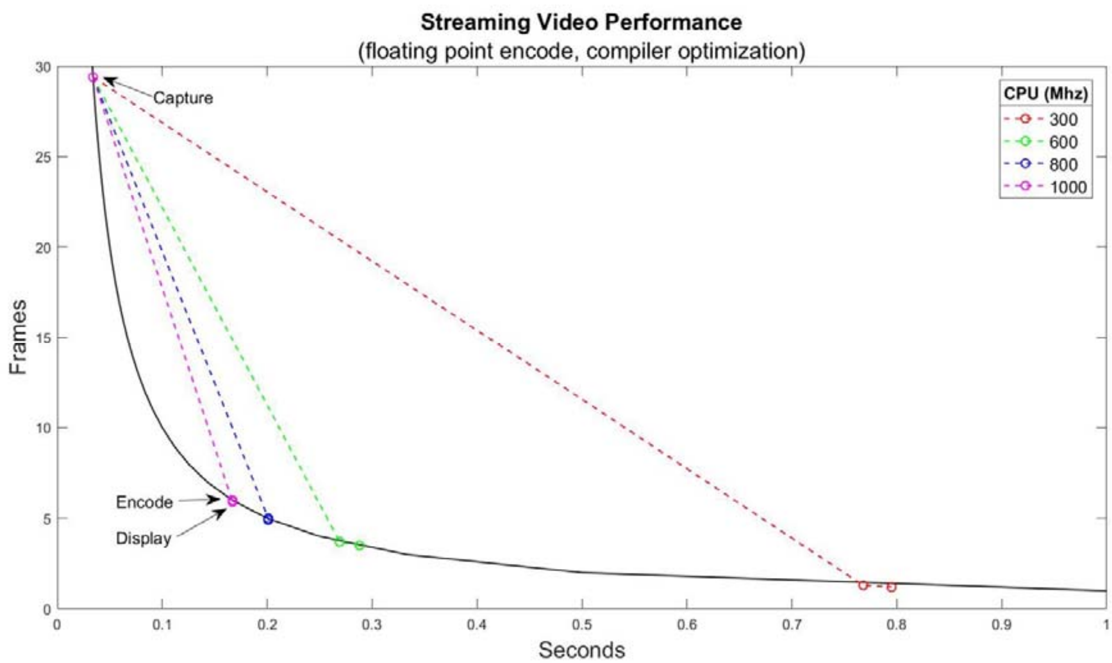


Figure 24 – Basic Streaming Video Performance (Table 5)

8.3 LOOK UP TABLE COLOR SPACE CONVERSION

The next implemented optimization was the switch from the use of a floating point color space conversion algorithm that consumes CPU cycles to a look up table color space conversion that predominantly consumes memory access cycles. The performance increases from 3.8 fps (Table 4) to 10.0 fps (Table 6) at 1000 MHz without compiler optimization and from 3.8 fps (Table 4) to 14.8 fps (Table 8) at 1000 MHz using compiler optimization. Table 7 shows that by eliminating the look up table memory access operation pointer arithmetic and therefore the color space conversion, there is little overall degradation in performance from frame capture to display from 600 MHz to 1000 MHz. Clearly, if the camera provided a color space that the processor and display would support directly, performance would improve substantially. As will be shown, the color space conversion performance has a much larger impact on latency and fps than the use of the Haar DWT image processing algorithm which did not seem intuitively obvious at first glance.

Event	CPU Speed			
	300 Mhz	600 Mhz	800 Mhz	1000 Mhz
Initialization	.0023s	.0020s	.0018s	.0017s
Capture Frame	.034s (29.4 fps)	.034s (29.4 fps)	.034s (29.4 fps)	.034s (29.4fps)
Encode Frame	.200s (5.0 fps)	.100s (10.0 fps)	.100s (10.0 fps)	.093s (10.7 fps)
Display Frame	**.238s (4.2 fps)**	**.130s (7.7 fps)**	**.101s (10.0 fps)**	**.101s (10.0 fps)**

Table 6 - Basic Streaming Video
(Performance Using Look Up Table Without Compiler Optimization)

Event	CPU Speed			
	300 Mhz	600 Mhz	800 Mhz	1000 Mhz
Initialization	.0023s	.0020s	.0018s	.0017s
Capture Frame	.0340s (29.4 fps)	.0340s (29.4 fps)	.0340s (29.4 fps)	.0340s (29.4 fps)
Encode Frame	.0348s (28.7 fps)	.0339s (29.5 fps)	.0340s (29.4 fps)	.0340s (29.4 fps)
Display Frame	.0570s (17.6 fps)	.0339s (29.5 fps)	.0340s (29.4 fps)	.0340s (29.4 fps)

Table 7 - Basic Streaming Video
(Performance Without Encode using LUT Pointer Arithmetic to Access LUT and Compiler Optimization – Display all White Pixels)

Event	CPU Speed			
	300 Mhz	600 Mhz	800 Mhz	1000 Mhz
Initialization	.0023s	.0020s	.0018s	.0017s
Capture Frame	.034s (29.4 fps)	.034s (29.4 fps)	.034s (29.4 fps)	.034s (29.4 fps)
Encode Frame	.134s (7.5 fps)	.071s (13.9 fps)	.067s (14.8 fps)	.0670s (14.9 fps)
Display Frame	.135s (7.4 fps)	**.072s (13.8 fps)**	**.068s (14.7 fps)**	**.0675s (14.8 fps)**

Table 8 - Basic Streaming Video
(Performance with LUT and Compiler Optimization)

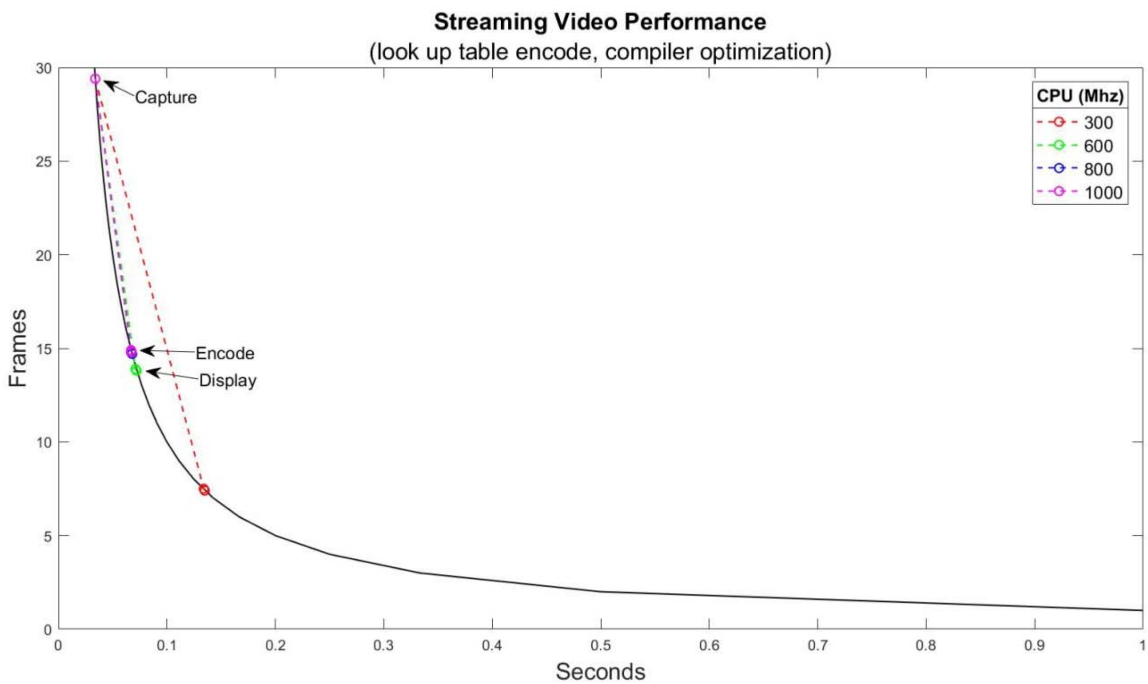


Figure 25 – Basic Streaming Video Performance (Table 8)

8.4 STREAMING VIDEO AND IMAGE PROCESSING

At this stage, the optimized Haar DWT image processing function is added to the optimized basic streaming video capabilities to provide a complete real-time streaming video and image processing system. Notice that the color space conversion takes from 33ms at 1000 MHz to 54ms at 600 MHz while the Haar DWT image processing algorithm takes from 15ms at 1000 MHz to 11ms at 600 MHz. This result indicates that color space conversion remains the largest drag on latency and fps performance in the system while image processing runs in a consistently efficient manner from 600 MHz to 1000 MHz. Initialization time is approximately doubled over the previous scenarios because the entire 33MB look up table is read into memory. However, this is considered to be negligible in comparison to other performance metrics. Overall performance ranged from 10.0 fps at 600 MHz to 11.8 fps at 1000 MHz. The target goal was to achieve real-time streaming video and image processing latency of no more than 100ms which was accomplished at a clock speed of 600 MHz.

Event	CPU Speed			
	300 Mhz	600 Mhz	800 Mhz	1000 Mhz
Initialization	.009s	.004s	.003s	.003s
Capture Frame	.034s (29.4 fps)	.034s (29.4 fps)	.034s (29.4 fps)	.034s (29.4 fps)
Encode Frame	.139s (7.2 fps)	.088s (11.4 fps)	.073s (13.7 fps)	.067s (14.9 fps)
Haar DWT Frame	.168s (5.9 fps)	.099s (10.1 fps)	.084s (11.9 fps)	.082s (12.2 fps)
Display Frame	.189s (5.3 fps)	**.100s (10.0 fps)**	**.087s (11.5 fps)**	**.085s (11.8 fps)**

Table 9 - Streaming Video and Image Processing
(Performance with LUT Completely Read into Memory, Haar DWT and Compiler Optimization)

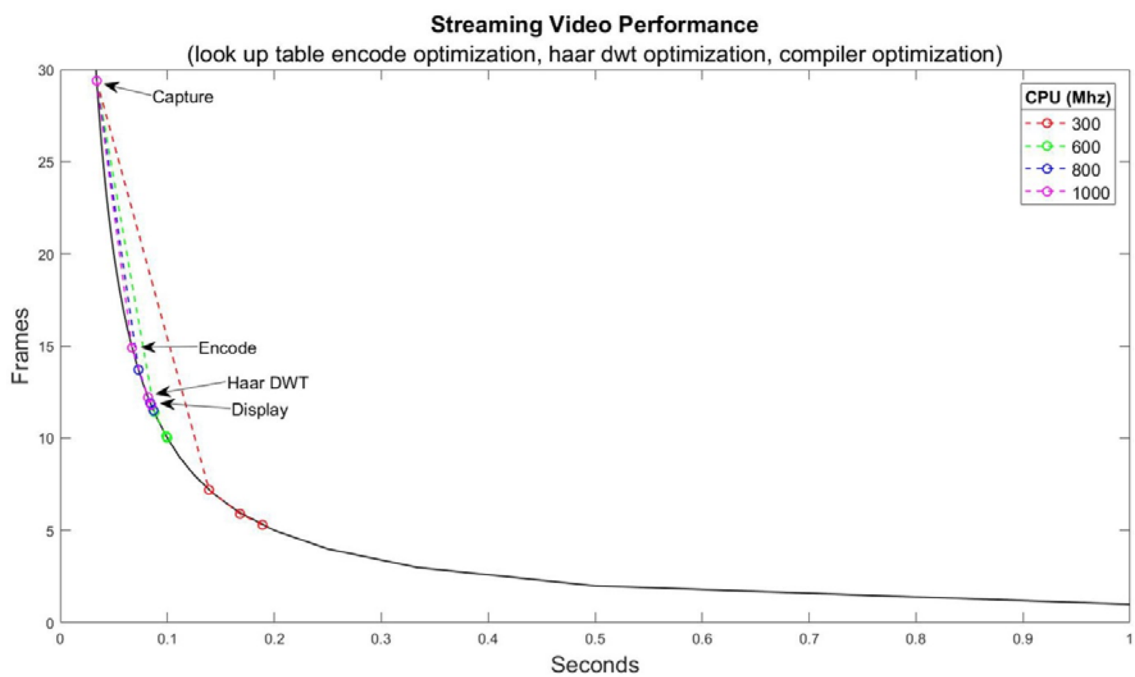


Figure 26 - Streaming Video and Image Processing Performance (Table 9)

8.5 ANALYSIS OF PERFORMANCE RESULTS

The results of this research have shown that real-time streaming video and image processing on inexpensive hardware with low latency is attainable. Basic streaming video performed consistently at 15 fps with a latency of 70ms from 600 MHz to 1000 MHz. Streaming video and image processing performed at 10 fps with 100ms latency at 600 MHz and attained a maximum performance of 12 fps with 85ms latency at 1000 MHz. These results are well within the 100ms target goal and were attained on a single core processor with no GPU support.

There is a tradeoff between clock speed and power consumption that needs to be considered in applications above 600 MHz since an increase in clock speed does not exponentially increase fps performance. In this work, CPU cycles used in floating point color space calculations were traded off for memory cycles when accessing a color space conversion look up table. This may explain why there was not much increase in performance above 600 MHz.

A 34ms camera to frame capture delay was consistent across all optimization and measurement scenarios and is left open for further investigation. The use of compiler optimization is an absolute necessity and color space conversion is the largest drag on performance. The Haar DWT image processing algorithm had an almost a negligible effect (10ms – 15ms) on system performance when compared to the 34ms camera to frame capture delay and the color space conversion (33ms – 66ms) latency.

9. CONCLUSIONS

In conclusion, this Thesis affirmatively answered the question "*can real-time streaming video and image processing be implemented on inexpensive hardware with low latency?*" Consider the following:

1. "*Real-time*" in the context of the human response time goal required during a manual welding operation was achieved with a streaming video and image processing rate of 10 frames-per-second (fps). Basic streaming video without image processing resulted in performance of 15 fps.
2. "*Inexpensive hardware*" was successfully used to obtain the real-time fps and latency performance results required to meet the presumed $300 to $500 market price for replacing an existing ADF cartridges with a MRW cartridge. The commercially available off-the-shelf hardware used in this research cost $150, well within the cost target needed to meet the market price for replacement MRW cartridges.
3. "*Low latency*" of 70ms for basic streaming video and 100ms for streaming video and image processing was achieved.

10. RECOMMENDATIONS

Open questions remain regarding potential performance enhancements of the current work. The following recommendations have been identified in that regard:

1. Eliminate color space conversion by selecting a camera with the same or closer color space as the processor/display hardware. For example, the Logitech C910 Webcam has a RGB888 format. Would this be a faster conversion than YUV422 to RGB565?
2. Instrument time measurement times using a scope bit and an oscilloscope since the performance of the software used to measure performance was itself not measured. As a result, I do not know how much impact a software time measurement instrumentation approach impacted the results.
3. Use of a tool that can program performance analysis and hardware-software co-verification such as SimpleScaler or VTune.
4. Examine the feasibility of implementing image processing in the BeagleBone Black SGX530 GPU and frame buffer 2D acceleration.
5. Investigate the cause of possible memory bus bandwidth performance when using a look up table.
6. Investigate the optimization on performance by eliminating duplicate color space conversion entries in the YUV422 to RGB565 look up table.
7. Investigate the performance impact of logical shift over multiplication operations in the color space look up table pointer arithmetic calculation:
   ```
   rgb565 = *(pbuf + ((Y0*256*256)+(U0*256)+V0));
   ```
8. Investigate camera to frame capture delay.

11. FUTURE WORK

Additional future work to determine the ultimate feasibility of Mediated Reality Welding is necessary to determine its commercial and technical viability. Additional topics for consideration include, but are not limited to the following:

1. Evaluation of a low-cost smart camera design suitable for the harsh lighting conditions found in the welding environment.
2. Analysis of overall power consumption and energy management.
3. Possible use of dynamic voltage and frequency scaling (DVFS) to save energy consumption is software codec’s an image processing algorithms.
4. Development of image processing algorithms that accommodate operator visual impairment (i.e. users wearing eye glasses to correct vision).
5. Use of FPGAs and multi-core architectures to improve performance.
6. Quality of Environment (QoE) evaluation of human interaction in the VR/AR/MR environment using performance metrics.

BIBLIOGRAPHY

[1] R. Miller, "Response Time in Man-Computer Conversational Transactions", Fall Joint Computer Conference, 1968, International Business Machines.

[2] M. Tovee, "How Fast Is The Speed Of Thought", Neuronal Processing, Current Biology, Vol. 4. No. 12, 1994.

[3] M. Potter, et al., "Detecting Meaning in RSVP at 13ms per picture", Attention, Perception & Psychophysics, February 2014, Vol. 76 Issue 2, pp. 270-279.

[4] T. Burger, "How Fast is Realtime? Human Perception and Technology, https://www.pubnub.com/blog/2015-02-09-how-fast-is-realtime-human-perception-and-technology/ , Pubnub, February 9, 2015.

[5] A. Syberfeldt, O. Danielsson and P. Gustavsson, "Augmented Reality Smart Glasses in the Smart Factory: Product Evaluation Guidelines and Review of Available Products," in IEEE Access, vol. 5, pp. 9118-9130, 2017.

[6] J.P. Gownder, "How Enterprise Smart Glasses Will Drive Workforce Enablement – Forecast: U Enterprise Adoption And Usage Of Smart Glasses", Forrester Research, April 21, 2016.

[7] Magrid Abraham, “Augmented Reality Is Already Improving Worker Performance”, Marco Annunziata, Harvard Business Review, March 13, 2017.

[8] Mark Gordon, "Welding Helmet with Eye Piece Control”, US Patent #3.873,804, March 25, 1975.

[9] R. Hill, C. Madden, A. v. d. Hengel, H. Detmold and A. Dick, "Measuring Latency for Video Surveillance Systems", 2009 Digital Image Computing: Techniques and Applications, Melbourne, VIC, 2009, pp. 89-95.

[10] Q. Li, P. Liu and C. Li, "Research on Embedded Video Monitoring System Based on Linux”, 2009 International Conference on Computer Engineering and Technology, Singapore, 2009, pp. 478-481.

[11] P. Poudel and M. Shirvaikar, "Optimization of Computer Vision Algorithms for Real Time Platforms”, 2010 42nd Southeastern Symposium on System Theory (SSST), Tyler, TX, 2010, pp. 51-55.

[12] Y. Liping and S. Kai, "Design and Realization of Image Processing System Based on Embedded Platform", 2010 International Forum on Information Technology and Applications, Kunming, 2010, pp. 446-449.

[13] Denan Li and Zhiyun Xiao, "Design of Embedded Video Capture System Based on ARM9”, 2011 International Conference on Electric Information and Control Engineering, Wuhan, 2011, pp. 2092-2095.

[14] J. Schlessman and M. Wolf, "Tailoring Design for Embedded Computer Vision Applications", in Computer, vol. 48, no. 5, pp. 58-62, May 2015.

[15] M. Ahmadi, W. J. Gross and S. Kadoury, "A Real-Time Remote Video Streaming Platform for Ultrasound Imaging", 2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Orlando, FL, 2016, pp. 4383-4386.

[16] S. Saypadith, W. Ruangsang and S. Aramvith, "Optimized Human Detection on the Embedded Computer Vision System," 2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), Kuala Lumpur, 2017, pp. 1707-1711.

[17] R. Lienhart and J. Maydt, "An Extended Set of Haar-Like Features for Rapid Object Detection”, Proceedings. International Conference on Image Processing, 2002, pp. I-900-I-903 vol.1.

[18] M. Noman, M. H. Yousaf and S. A. Velastin, "An Optimized and Fast Scheme for Real-Time Human Detection Using Raspberry Pi”, 2016 International Conference on

Digital Image Computing: Techniques and Applications (DICTA), Gold Coast, QLD, 2016, pp. 1-7.

[19] J. Schlessman and M. Wolf, "Tailoring Design for Embedded Computer Vision Applications", in Computer, vol. 48, no. 5, pp. 58-62, May 2015.

[20] C. Bachhuber, E. Steinbach, M. Freundl and M. Reisslein, "On the Minimization of Glass-to-Glass and Glass-to-Algorithm Delay in Video Communication", in IEEE Transactions on Multimedia, vol. 20, no. 1, pp. 238-252, Jan. 2018.

[21] R. Gregg, "Mediated Reality Welding", University of Nebraska, May 8, 2014.

[22] R. Gregg, "Method And System For Mediated Reality Welding", US Patent Application #14/704562, Filed May 5, 2015.

[23] R. Gregg, "Method And System For Mediated Reality Welding", European Patent Office (EPO) Application #15789635.8, Filed May 6, 2015.

[24] R. Gregg, "Method And System For Mediated Reality Welding", Japan Patent Application #2017-511543, Filed May 6, 2015.

[25] R. Gregg, "Method And System For Mediated Reality Welding", China Patent Application #PCT/US2015/029338, Filed May 6, 2015.

[26] J.P. Gownder, "How Enterprise Smart Glasses Will Drive Workforce Enablement – Forecast: US Enterprise Adoption And Usage Of Smart Glasses", Forrester Research, April 21, 2016.

[27] M. Abraham and M. Annunziata, "Augmented Reality Is Already Improving Worker Performance", Harvard Business Review, March 13, 2017.

[28] G. Cooley, "BeagleBone Black System Reference Manual – Rev C.1", www.beaglenoard.org, May 22, 2014.

[29] D. Molloy, "Exploring BeagleBone – Tools and Techniques for Building with Embedded Linux", John Wiley & Sons, 2015.

[30] http://support.logitech.com/en_us/product/hd-pro-webcam-c920

[31] https://www.4dsystems.com.au/product/4DCAPE_43/

[32] "Universal Serial Bus Device Class Definition for Video Devices-Revision 1.5", http://www.usb.org/developers/docs/devclass_docs/, August 9, 2012.

[33] "Linux Media Kernel API Documentation", https://www.linuxtv.org/docs.php

[34] "OpenCV", https://opencv.org/

[35] G. Bradski and A. Kuehler, "Learning OpenCV", www-cs.ccny.cuny.edu/~wolberg/capstone/opencv/LearningOpenCV.pdf, O'Reilly Media, Inc., 2008.

[36] "MATLAB Coder", https://www.mathworks.com/products/matlab-coder/features.html

[37] "Haar Wavelet Image Compression", https://people.math.osu.edu/husen.1/teaching/572/image_comp.pdf

[38] C Mulcahy, et al.,"Image Compression Using Haar Wavelet Transform, Spelman Science and Mathematics Journal, 1997.

[39] K. Talukder and K. Harada, "Haar Wavelet Based Approach for Image Compression and Quality Assessment of Compressed Image", IAENG International Journal of Applied Mathematics, Volume 36, Issue 1, 2007.

[40] C. Mulcahy, "Image Compression Using the Haar Wavelet Transformation", Spelman Science and Math Journal.

[41] “Discrete Wavelet Analysis”, https://www.mathworks.com/help/wavelet/discrete-wavelet-analysis.html, MathWorks, Inc.

[42] “Low Level Graphics on Linux”, http://betteros.org/tut/graphics1.php

[43] S. Marchesion, “Linux Graphics Drivers: an Introduction”, https://people.freedesktop.org/~marcheu/linuxgraphicsdrivers.pdf, March 15, 2012.

[44] G. Uytterhoeven, “The Frame Buffer Device”, https://www.kernel.org/doc/Documentation/fb/framebuffer.txt, May 10, 2001.

[45] “Datahammer 7yuv Raw Image Data and Hex Editor”, http://datahammer.de/

[46] “FFmpeg”, https://www.ffmpeg.org/

[47] “ImageMagick”, https://www.imagemagick.org/script/index.php

[48] “ioctl() man page”, http://man7.org/linux/man-pages/man2/ioctl.2.html

[49] “mmap() man page”, http://man7.org/linux/man-pages/man2/mmap.2.html

[50] R. Love, “Linux System Programming”, O’Reilly Media, Inc., September 2007.

[51] “Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios”, Recommendation ITU-R BT.601-7, March 2011.

[52] “clock_gettime() man page”, https://linux.die.net/man/3/clock_gettime

[53] “Lena Test Image”, http://www.lenna.org

[54] “gcc -o / -O option flags”, https://www.rapidtables.com/code/linux/gcc/gcc-o.html#optimization

[55] “Options That Control Optimization”, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

STREAMING VIDEO AND IMAGE PROCESSING SOURCE CODE (sv5.c)

```c
/*
** sv5.c
** write streaming video from a webcam into the framebuffer for
** real-time display on beaglebone black
**
** april 29, 2018 - rlg
*/

/*
** NOTE: need to run as root in tty1 (chvt 1) because framebuffer is at
** the linux kernel level and only available in tty
**
** DEBUG compile using: gcc -g3 sv5.c -o sv5 -lrt
** OPTIMIZE compile using: gcc -O3 sv5.c -o sv5 -lrt
**
** set cpu frequency governor to "performance" on start-up
** echo userspace > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
*/

#include <linux/fb.h>
#include <stdio.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <linux/videodev2.h>

/*
** 4D LCD display resolution 480 x 272
** HVGA (Half-size VGA) screens have 480×320 pixels (3:2 aspect ratio),
** 480×360 pixels (4:3 aspect ratio), 480×272 (~16:9 aspect ratio)
** or 640×240 pixels (8:3 aspect ** ratio).
*/
#define HVGA_WIDTH      480
#define HVGA_HEIGHT     272
#define FRAMEBUF_SIZE   HVGA_WIDTH*HVGA_HEIGHT*2
/*
** WQVGA resolution is 432 x 240
** Logitech c920 USB Camera
*/
#define WQVGA_WIDTH     432
#define WQVGA_HEIGHT    240
#define FILEBUF_SIZE    WQVGA_WIDTH*WQVGA_HEIGHT*2
#define RGB565_SIZE     WQVGA_WIDTH*WQVGA_HEIGHT*2
#define RGB888_SIZE     WQVGA_WIDTH*WQVGA_HEIGHT*3
#define YUYV_SIZE       WQVGA_WIDTH*WQVGA_HEIGHT*2
#define GRAYSCALE_SIZE  WQVGA_WIDTH*WQVGA_HEIGHT
/*
** SVGA resolution is 800 x 600
*/
#define SVGA_WIDTH      800
#define SVGA_HEIGHT     600
/*
** WUXGA resolution is 1920 x 1200
*/
#define WUXGA_WIDTH     1920
```

```c
#define WUXGA_HEIGHT    1200
/*
** benchmark timing sample size
*/
#define SAMPLE_SIZE     100
/*
** RGB565
*/
#define WHITE   0xffff
#define YELLOW  0xffe0
#define CYAN    0x07ff
#define GREEN   0x07e0
#define MAGENTA 0xf81f
#define RED     0xf800
#define BLUE    0x001f
#define BLACK   0x0000
#define GRAY    0xc618
/*
** function declarations
*/
uint16_t rgb888_to_rgb565(uint32_t);
uint32_t yuv422_to_rgb888(uint8_t Y, uint8_t U, uint8_t V);
uint16_t yuv422_to_rgb565(uint8_t Y, uint8_t U, uint8_t V);
int display_HDMI(void *fbp, uint8_t *rgb565ptr);
int display_LCD4(void *fbp, void *filebuf);
int convert2(void *cbp, uint8_t *rgb565ptr);
int convert3(void *cbp, uint8_t *rgb565ptr, uint16_t *pbuf);
int RGBColorBars_HDMI(void *fbp);
int RGBColorBars_LCD4(void *fbp);
int RGBDisplayFile_HDMI(void *fbp, char *filepath);
int ReadRGBFile(void *filebuf, char *fpath);
int WriteRGBFile(void *filebuf, char *fpath);
int init_fb_color(void *fbp, uint16_t color);

/*
** haar dwt constants and function declaration
*/
int HaarDwt(uint16_t *imgin_ptr, uint16_t *imgout_ptr);

/*
** general purpose variables
*/
uint8_t *orig_rgb565ptr=NULL, *rgb565ptr=NULL;

/*
** framebuffer variables
*/

int height=0,width=0,step,channels;

int i,j,k;
uint8_t r,g,b, pixel;
int x=0,y=0,idx=0;
uint16_t rgbp, rgb;
long screensize=0;
struct fb_fix_screeninfo finfo;
struct fb_var_screeninfo vinfo;
int fb_fd=0;
uint16_t *fbp=NULL;
void *filebuf=NULL;

/*
```

```
** yuv422 to rgb565 look up table (lut) variables
*/
int lut_size = 256*256*256*2;
int lut_fd=0;

/*
** webcam video for linux (v4l2) variables
*/
int vid_fd;
struct v4l2_capability v4l2_cap;
struct v4l2_format v4l2_fmt;
struct v4l2_requestbuffers v4l2_reqbuf;
struct v4l2_buffer v4l2_buf;
void *cbp = NULL;

/*
** timing declarations
*/
struct timespec fps_start_time,
                fps_end_time,
                init_time_start,
                init_time_end;
double t_diff=0, fps=0;
int count=SAMPLE_SIZE;
int tlog=1;     /*1=timing on, 0=timing off*/

/***************************************
** main()
***************************************/

int main(void)
{
        if (tlog) clock_gettime(CLOCK_REALTIME, &init_time_start);

        /*
        ** open the framebuffer
        */

        if((fb_fd = open("/dev/fb0",O_RDWR))<0)
        {
                perror("open\n");
                exit(1);
        }/*eo if*/

        /*
        ** Initialize the framebuffer
        */
        ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);
        vinfo.grayscale=0;
        vinfo.bits_per_pixel=16;
        if (ioctl(fb_fd, FBIOPUT_VSCREENINFO, &vinfo) < 0){
                printf("FBIOPUT_VSCREENINFO error %d\n", errno);
                exit(1);
        }/*eo if*/

        /*
        ** get the framebuffer properties
        */
        ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);
        ioctl(fb_fd, FBIOGET_FSCREENINFO, &finfo);

        /*
```

```
        ** calculate the framebuffer screensize
        */
        screensize = vinfo.yres_virtual * finfo.line_length;

        /*
        ** get the address for the framebuffer
        */
        fbp = NULL;
        fbp = mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fb_fd, 0);
        if (fbp == MAP_FAILED){
                printf("framebuffer - mmap failed errno %d\n", errno);
                exit(1);
        }/*eo if*/

        /*
        ** set up camera
        */

        /*
        ** open webcam
        ** video0 = Logitech C920 USB Webcam on Beagle Bone Black
        */
        if((vid_fd = open("/dev/video0", O_RDWR )) < 0)
        {
                perror("webcam open");
                exit(1);
        }/*eo if*/

        /*
        ** get webcam capabilites
        */
        if(ioctl(vid_fd, VIDIOC_QUERYCAP, &v4l2_cap) < 0)
        {
                perror("VIDIOC_QUERYCAP");
                exit(1);
        }/*eo if*/


        if(!(v4l2_cap.capabilities & V4L2_CAP_VIDEO_CAPTURE))
        {
                fprintf(stderr, "The device does not handle single-planar video\n");
                exit(1);
        }//eo if


        if(!(v4l2_cap.capabilities & V4L2_CAP_STREAMING))
        {
                fprintf(stderr, "The device does not handle frame streaming\n");
                exit(1);
        }//eo if

        /*
        ** set the webcam format
        */
        memset(&v4l2_fmt, 0, sizeof(v4l2_fmt));
        v4l2_fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        v4l2_fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
        v4l2_fmt.fmt.pix.width =  WQVGA_WIDTH;  /*432*/
        v4l2_fmt.fmt.pix.height = WQVGA_HEIGHT; /*240*/

        if(ioctl(vid_fd, VIDIOC_S_FMT, &v4l2_fmt) <0)
```

```
        {
                perror("VIDIOC_S_FMT");
                exit(1);
        }/*eo if*/

        /*
        ** request buffer(s)
        ** initiate memory mapped I/O. Memory mapped buffers are located in device
        ** memory and must be allocated before they can be mapped into the applications
        ** I/O space.
        */
        memset(&v4l2_reqbuf, 0, sizeof(v4l2_reqbuf));
        v4l2_reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        v4l2_reqbuf.memory = V4L2_MEMORY_MMAP;
        v4l2_reqbuf.count = 1;

        if(ioctl(vid_fd, VIDIOC_REQBUFS, &v4l2_reqbuf) < 0)
        {
                perror("VIDIOC_REQBUFS");
                exit(0);
        }/*eo if*/

        /*
        ** query the status of the buffers (why?)
        */
        memset(&v4l2_buf, 0, sizeof(v4l2_buf));
        v4l2_buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        v4l2_buf.memory = V4L2_MEMORY_MMAP;
        v4l2_buf.index = 0;

        if(ioctl(vid_fd, VIDIOC_QUERYBUF, &v4l2_buf) < 0)
        {
                perror("VIDIOC_QUERYBUF");
                exit(1);
        }/*eo if*/

        /*
        ** map webcam buffer to kernel space
        */
        cbp = mmap(NULL, v4l2_buf.length,PROT_READ | PROT_WRITE,
                   MAP_SHARED, vid_fd, v4l2_buf.m.offset);

        if(cbp == MAP_FAILED)
        {
                printf("camera - mmap failed errno=%d\n", errno);
                exit(1);
        }/*eo if*/

        /*
        ** start v4l2 streaming
        */
        memset(&v4l2_buf, 0, sizeof(v4l2_buf));
        v4l2_buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        v4l2_buf.memory = V4L2_MEMORY_MMAP;
        v4l2_buf.index = 0;

        int type = v4l2_reqbuf.type;
        int result = ioctl(vid_fd, VIDIOC_STREAMON, &type);
        if(result < 0)
        {
```

```c
		perror("VIDIOC_STREAMON");
		int error = errno;
		exit(1);
	}/*eo if*/

	/*
	** set up rgb565file buffer
	*/
	orig_rgb565ptr = rgb565ptr = (uint8_t*)malloc(RGB565_SIZE);
	memset(rgb565ptr, 0, RGB565_SIZE);

	/*
	** use yuv2rgb look up table
	*/
	lut_fd = open("/home/root/yuv2rgb.lut", O_RDWR);
	if(lut_fd < 0){
		printf("yuv2rgb.lut open failed errno=%d\n", errno);
		return(0);
	}/*eo if*/

	/*
	** read yuv2rgb.lut into virtual memory for random access by convert3()
	** changed MAP_SHARED to MAP_RIVATE | MAP_POPULATE for 0.1s per frame performance improvemnt
	*/
	uint16_t *lut_ptr = mmap(NULL, lut_size, PROT_READ, MAP_PRIVATE | MAP_POPULATE, lut_fd, 0);
	if(lut_ptr == MAP_FAILED){
		printf("lut - mmap failed errno=%d\n", errno);
		return(0);
	}/*eo if*/

	/*
	** use madvise() for mmap() performance improvement
	*/
	madvise(lut_ptr, lut_size, MADV_WILLNEED);

	/*
	** read yuv2rgb look up table into memory
	*/
	uint8_t *lut_buf;
	for(i=0; i<lut_size/4096; i++){
		lseek(lut_fd, (long)(i*4096), SEEK_SET);
		read(lut_fd, lut_buf, 1);
	}/*eo for*/


	/*
	** clear console and turn off cursor
	** to turn console on use printf("\033[?25h");
	*/
	printf("\033[3J");
	fflush(stdout);
	printf ("\033[?25l");
	fflush(stdout);

	/*
	** LCD4
	*/
	init_fb_color(fbp, GRAY);

	/*
```

```c
	** allocate output buffer for haar dwt result
	*/
	uint16_t *imgout_ptr = malloc(sizeof(uint8_t)*432*240*2);
	memset(imgout_ptr, 0xff, 432*240*2);

	/*
	** end of initialization
	*/
	if(tlog) clock_gettime(CLOCK_REALTIME, &init_time_end);

	/*****************************************************
	*****************************************************
	** begin streaming loop
	*****************************************************
	*****************************************************/
	if(tlog) clock_gettime(CLOCK_REALTIME, &fps_start_time);
	while(count){

		/*
		** provide camera with a buffer to fill
		*/
		v4l2_buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
		v4l2_buf.memory = V4L2_MEMORY_MMAP;
		result = ioctl(vid_fd, VIDIOC_QBUF, &v4l2_buf);
		if( result < 0)
		{
			perror("VIDIOC_QBUF");
			exit(1);
		}/*eo if*/

		/*
		** the buffer has been filled by the video camera
		** get the buffer for futher processing
		*/
		result = ioctl(vid_fd, VIDIOC_DQBUF, &v4l2_buf);
		if( result < 0)
		{
			perror("VIDIOC_DQBUF");
			printf("ERRNO %d\n", errno);
			exit(1);
		}/*eo if*/

		/*
		** convert yuyv422 to rgb565 using look up table
		*/
		convert3(cbp, rgb565ptr, lut_ptr);

		/*
		** display basic video stream
		*/
		//display_LCD4(fbp, rgb565ptr);


		/*
		** process image using haar dwt
		*/
		HaarDwt((uint16_t*)rgb565ptr, imgout_ptr);

		/*
		** display haar dwt video stream
		*/
		display_LCD4(fbp,(uint8_t*)imgout_ptr);
```

```c

		/*
		** timing
		*/
		if(tlog) count--;

	}/*eo while*/

	/***************************************************
	****************************************************
	** end streaming loop
	****************************************************
	***************************************************/

	/*
	** benchmark timing
	*/
	if(tlog){
		clock_gettime(CLOCK_REALTIME, &fps_end_time);
		/*
		** initialization time
		*/
		t_diff = (init_time_end.tv_sec - init_time_start.tv_sec) +
		(double)(init_time_end.tv_nsec - init_time_start.tv_nsec)/1000000000.0d;
		t_diff = t_diff/SAMPLE_SIZE;
		printf("initialization time: %f sec\n", t_diff);

		/*
		** frames per second
		*/
		t_diff = (fps_end_time.tv_sec - fps_start_time.tv_sec) +
		(double)(fps_end_time.tv_nsec - fps_start_time.tv_nsec)/1000000000.0d;
		t_diff = t_diff/SAMPLE_SIZE;
		fps = (double)1/t_diff;
		printf("%f sec/frame %f frames/sec\n", t_diff, fps);
	}/*eo if*/


	/*
	** deactivate streaming
	*/
	if (ioctl(vid_fd, VIDIOC_STREAMOFF, &type) < 0)
	{
		perror("VIDIOC_STREAMOFF");
		exit(1);
	}//eo if

	/*
	** clean up
	*/

	/*
	** close webcam
	*/
	close(vid_fd);
	munmap(cbp, v4l2_buf.length);

	/*
	** close framebuffer
	*/
```

```c
	close(fb_fd);
	munmap(fbp, screensize);

	/*
	** yuv422 to rgb565 conversion buffer
	*/
	free(rgb565ptr);
	if(filebuf) free(filebuf);

	/*
	** lut
	*/
	close(lut_fd);
	munmap(lut_ptr,(lut_size));

	printf("done\n");
	return 0;

}/*eo main****************************/

/*
** display_HDMI
*/
int display_HDMI(void *fbp, uint8_t *rgb565ptr){

	int x=0,y=0,idx=0;
	uint8_t pxlsb=0,pxmsb=0;
	uint8_t *orig_rgb565ptr = rgb565ptr;	/*debug*/

	idx = idx+(WUXGA_WIDTH*(HVGA_WIDTH-WQVGA_WIDTH));

	for(y=0; y<WQVGA_HEIGHT; y++){
		for(x=0; x<WQVGA_WIDTH; x++){
			*(uint8_t*)(fbp+idx+(HVGA_HEIGHT-WQVGA_HEIGHT))
						= *rgb565ptr;
			idx++;
			rgb565ptr++;
			*(uint8_t*)(fbp+idx+32) = *rgb565ptr;
			idx++;
			rgb565ptr++;

		}/*eo for*/
		idx = idx+((WUXGA_WIDTH -WQVGA_WIDTH)*2);
	}/*eo for*/

	return(0);

}/*eo display_HDMI*/

/*
** display the frame buffer on LCD4
*/
int display_LCD4(void *fbp, void *filebuf){


	int i=0,j=0,idx=0;
	uint16_t *fbptr = fbp;
	uint16_t *filebuf_ptr = filebuf;
	uint16_t pixel=0;

	idx = HVGA_WIDTH*((HVGA_HEIGHT-WQVGA_HEIGHT)/2);

```

```
        for(i=0; i<WQVGA_HEIGHT; i++){          /*row*/

                idx=idx+((HVGA_WIDTH-WQVGA_WIDTH)/2);

                for(j=0; j<WQVGA_WIDTH; j++){   /*col*/

                        *(fbptr+idx)=*filebuf_ptr;
                        idx++;
                        filebuf_ptr++;

                }/*eo for*/

                idx=idx+((HVGA_WIDTH-WQVGA_WIDTH)/2);

        }/*eo for*/

        return(0);

}/*eo Display_LCD4*/

/*
** convert2
** uses floating point calculations to convert yuv422 to rgb565
*/
int convert2(void *cbp, uint8_t *rgb565ptr){

        int i=0;
        uint8_t *yuvptr = NULL;
        uint8_t *orig_yuvptr = NULL;
        uint8_t Y0,Y1,U0,V0;
        uint8_t pixel;
        uint16_t rgb565;

        orig_yuvptr = yuvptr = (uint8_t *)cbp;

        for(i=0; i<YUYV_SIZE; i++){
                Y1 = *yuvptr;   /*Y1*/
                i++;
                yuvptr++;
                V0 = *yuvptr;   /*V0*/
                i++;
                yuvptr++;
                Y0 = *yuvptr;   /*Y0*/
                i++;
                yuvptr++;
                U0 = *yuvptr;   /*U0*/
                yuvptr++;       /*next 4 byte macropixel*/

                /*
                ** convert yuv422 to rgb565
                */
                rgb565 = yuv422_to_rgb565(Y0,U0,V0);

                pixel = rgb565;
                *rgb565ptr = pixel;
                rgb565ptr++;

                pixel = rgb565 >> 8;
                *rgb565ptr = pixel;
                rgb565ptr++;

                /*
```

```
                ** convert yuv422 to rgb565
                */
                rgb565 = yuv422_to_rgb565(Y1,U0,V0);

                pixel = rgb565;
                *rgb565ptr = pixel;
                rgb565ptr++;

                pixel = rgb565 >> 8;
                *rgb565ptr = pixel;
                rgb565ptr++;

        }/*eo for*/

        return(0);

}/*eo convert2*/

/*
** convert3
** uses yuv2rgb.lut look up table to convert yuv422 to rgb565
*/
int convert3(void *cbp, uint8_t *rgb565ptr, uint16_t *pbuf){

        int i=0;
        uint8_t *yuvptr = NULL;
        uint8_t Y0,Y1,U0,V0;
        uint8_t pixel;
        uint16_t rgb565=0xffff;

        yuvptr = (uint8_t *)cbp;

        for(i=0; i<YUYV_SIZE; i++){
                Y1 = *yuvptr;   /*Y1*/
                i++;
                yuvptr++;
                V0 = *yuvptr;   /*V0*/
                i++;
                yuvptr++;
                Y0 = *yuvptr;   /*Y0*/
                i++;
                yuvptr++;
                U0 = *yuvptr;   /*U0*/
                yuvptr++;       /*next 4 byte macropixel*/

                /*
                ** convert yuv422 to rgb565
                */
                rgb565 = *(pbuf + ((Y0*256*256)+(U0*256)+V0));

                pixel = rgb565;
                *rgb565ptr = pixel;
                rgb565ptr++;

                pixel = rgb565 >> 8;
                *rgb565ptr = pixel;
                rgb565ptr++;


                /*
                ** convert yuv422 to rgb565
                */
```

```
                    rgb565 = *(pbuf + ((Y1*256*256)+(U0*256)+V0));

                    pixel = rgb565;
                    *rgb565ptr = pixel;
                    rgb565ptr++;

                    pixel = rgb565 >> 8;
                    *rgb565ptr = pixel;
                    rgb565ptr++;

        }/*eo for*/

        return(0);

}/*eo convert3*/

/*
** yuv422 to rgb888 conversion
*/
uint32_t yuv422_to_rgb888(uint8_t Y, uint8_t U, uint8_t V){

        uint32_t  rgb_red=0, rgb_blue=0, rgb_green=0;
        uint32_t rgb888=0;

        /*
        ** ITU-R 708
        */

        /*
        float r = ( (1.164*(float)(Y-16)) + (2.115*(float)(V-128)) );
        if(r<0) r=0;
        if(r>255) r=255;
        rgb_red = (uint32_t)r;

        float g = ( (1.164*(float)(Y-16)) - (0.534*(float)(V-128)) -
                    (0.213*(float)(U-128)) );
        if(g<0) g=0;
        if(g>255) g=255;
        rgb_green = (uint32_t)g;

        float b = ( (1.164*(float)(Y-16)) + (1.793*(float)(U-128)) );
        if(b<0) b=0;
        if(b>255) b=255;
        rgb_blue = (uint32_t)b;
        */

        /*
        ** ITU-R 601
        */
        float r = 1.164*(float)(Y-16) + 1.596*(float)(V-128);
        if(r<0) r=0;
        if(r>255) r=255;
        rgb_red = (uint32_t)r;

        float g = 1.164*(float)(Y-16) - 0.813*(float)(V-128) - 0.391*(float)(U-128);
        if(g<0) g=0;
        if(g>255) g=255;
        rgb_green = (uint32_t)g;

        float b = 1.164*(float)(Y-16) + 2.018*(float)(U-128);
        if(b<0) b=0;
```

```
        if(b>255) b=255;
        rgb_blue = (uint32_t)b;

        /*
        ** create packed rgb888
        */
        rgb_red = rgb_red << 8;
        rgb_green = rgb_green << 16;
        rgb_blue = rgb_blue << 24;
        rgb888 = rgb888 | rgb_red | rgb_green | rgb_blue;

        return(rgb888);

}/*eo yuv422_to_rgb888*/

/*
** convert yuv422 to rgb565
*/
uint16_t yuv422_to_rgb565(uint8_t Y, uint8_t U, uint8_t V){

        float r,g,b;
        uint8_t red = 0, green=0, blue=0;
        uint16_t r16=0, g16=0, b16=0, rgb565=0, bgr565=0;

        /*
        ** ITU-R 601
        */
        r = 1.164*(float)(Y-16) + 1.596*(float)(V-128);
        if(r<0) r=0;
        if(r>255) r=255;
        red = (uint8_t)r;

        g = 1.164*(float)(Y-16) - 0.813*(float)(V-128) - 0.391*(float)(U-128);
        if(g<0) g=0;
        if(g>255) g=255;
        green = (uint8_t)g;

        b = 1.164*(float)(Y-16) + 2.018*(float)(U-128);
        if(b<0) b=0;
        if(b>255) b=255;
        blue = (uint8_t)b;

        /*
        ** rgb565 format used in x86
        */
        /*
        r16 = ((red >>3) & 0x1f) << 11;
        g16 = ((green >> 2) & 0x3f) << 5;
        b16 = (blue >> 3) & 0x1f;
        rgb565 = r16 | g16 | b16;

        return(rgb565);
        */

        /*
        ** bgr565 format used in ARM & Beaglebone Black
        */

        b16 = ((blue >>3) & 0x1f) << 11;
        g16 = ((green >> 2) & 0x3f) << 5;
        r16 = (red >> 3) & 0x1f;
        bgr565 = b16 | g16 | r16;
```

```c

        return(bgr565);


}/*eo yuv422_to_rgb565*/

/*
** convert rgb888 to rgb565
*/
uint16_t rgb888_to_rgb565(uint32_t rgb888){

        uint8_t red = 0, green=0, blue=0;
        uint16_t r=0, g=0, b=0, rgb565=0;

        /* uint32_t rgb888 format
        ** msb 24-31 red, 16-23 green, 8-15 blue, lsb 0-7 0x00
        */

        red = rgb888 >> 24;
        green = rgb888 >> 16;
        blue = rgb888 >> 8;

        /*
        red = rgb888 >> 8;
        green = rgb888 >> 16;
        blue = rgb888 >> 24;
        */

        /* uint16_t rgb565 format
        ** msb 11-15 red, 5-10 green, 0-4 blue
        */
        r = ((red >>3) & 0x1f) << 11;
        g = ((green >> 2) & 0x3f) << 5;
        b = (blue >> 3) & 0x1f;

        rgb565 = r | g | b;

        return(rgb565);

}/*eo rgb888_to_rgb565*/

/*
** display RGB Color Bars on BeagleBone HDMI
*/
int RGBColorBars_HDMI(void *fbp){

        int x=0,y=0,idx=0;
        uint8_t pxlsb=0,pxmsb=0;

        idx = idx+(WUXGA_WIDTH*(HVGA_WIDTH-WQVGA_WIDTH));

        for(y=0; y<WQVGA_HEIGHT; y++){
                for(x=0; x<WQVGA_WIDTH; x++){
                        if(x>= 0 && x<= 53){
                                pxmsb = 0xff;   /*white*/
                                pxlsb = 0xff;
                        }/*eo if*/
                        if(x>= 54 && x<= 107){
                                pxmsb = 0xff;   /*yellow*/
                                pxlsb = 0xe0;
                        }/*eo if*/
                        if(x>= 108 && x<= 161){
```

```c
                                pxmsb = 0x07;   /*cyan*/
                                pxlsb = 0xff;
                        }/*eo if*/
                        if(x>= 162 && x<= 215){
                                pxmsb = 0x07;   /*green*/
                                pxlsb = 0xe0;
                        }/*eo if*/
                        if(x>= 216 && x<= 269){
                                pxmsb = 0xf8;   /*magenta*/
                                pxlsb = 0x1f;
                        }/*eo if*/
                        if(x>= 270 && x<= 323){
                                pxmsb = 0xf8;   /*red*/
                                pxlsb = 0x00;
                        }/*eo if*/
                        if(x>= 324 && x<= 377){
                                pxmsb = 0x00;   /*blue*/
                                pxlsb = 0x1f;
                        }/*eo if*/
                        if(x>= 378 && x<= 431){
                                pxmsb = 0x00;   /*black*/
                                pxlsb = 0x00;
                        }/*eo if*/
                        *(uint8_t*)(fbp+idx+(HVGA_HEIGHT-WQVGA_HEIGHT))
                                    = pxlsb; /*pixel lsb*/
                        idx++;
                        *(uint8_t*)(fbp+idx+32) = pxmsb; /*pixel msb*/
                        idx++;
                }/*eo for*/
                idx = idx+((WUXGA_WIDTH -WQVGA_WIDTH)*2);
        }/*eo for*/

        return(0);

}/*eo RGBColorBars_HDMI*/

/*
** display color bars on the BeagleBone LCD4
*/
int RGBColorBars_LCD4(void *fbp){


        int i=0,j=0,idx=0;
        uint16_t *fbptr = fbp;
        uint16_t pixel=0;

        idx = HVGA_WIDTH*((HVGA_HEIGHT-WQVGA_HEIGHT)/2);

        for(i=0; i<WQVGA_HEIGHT; i++){          /*row*/

                idx=idx+((HVGA_WIDTH-WQVGA_WIDTH)/2);

                for(j=0; j<WQVGA_WIDTH; j++){   /*col*/

                        if(j>=   0 && j<=  53) pixel = WHITE;
                        if(j>=  54 && j<= 107) pixel = YELLOW;
                        if(j>= 108 && j<= 161) pixel = CYAN;
                        if(j>= 162 && j<= 215) pixel = GREEN;
                        if(j>= 216 && j<= 269) pixel = MAGENTA;
                        if(j>= 270 && j<= 323) pixel = RED;
                        if(j>= 324 && j<= 377) pixel = BLUE;
                        if(j>= 378 && j<= 431) pixel = BLACK;
```

```
				*(fbptr+idx)=pixel;
				idx++;

			}/*eo for*/

			idx=idx+((HVGA_WIDTH-WQVGA_WIDTH)/2);

		}/*eo for*/

		return(0);

}/*eo RGBColorBars_LCD4*/


/*
** opens and display raw rgb565 files using HDMI
*/
int RGBDisplayFile_HDMI(void *fbp, char *filepath){

		int x=0,y=0,idx=0;
		uint8_t pxlsb=0,pxmsb=0;

		FILE *fp=NULL;
		int errnum=0, fsize=0;
		struct stat filestat;
		uint8_t *fbuf=NULL, *orig_fbuf=NULL;

		fp = fopen(filepath, "r");
		if(fp == NULL){
			errnum = errno;
			printf("error opening file: %s\n", strerror(errnum));
			return (1);
		}/*eo if*/
		fstat(fileno(fp), &filestat);
		fsize = filestat.st_size;
		orig_fbuf = fbuf = (uint8_t*)malloc(fsize);
		memset(fbuf, 0, fsize);
		fread(fbuf,sizeof(uint8_t),fsize,fp);
		fclose(fp);

		idx = idx+(WUXGA_WIDTH*(HVGA_WIDTH-WQVGA_WIDTH));

		for(y=0; y<WQVGA_HEIGHT; y++){
			for(x=0; x<WQVGA_WIDTH; x++){
				pxlsb = *fbuf;
				fbuf++;
				pxmsb = *fbuf;
				fbuf++;
				*(uint8_t*)(fbp+idx+(HVGA_HEIGHT-WQVGA_HEIGHT))
						= pxlsb; /*pixel lsb*/
				idx++;
				*(uint8_t*)(fbp+idx+32) = pxmsb; /*pixel msb*/
				idx++;
			}/*eo for*/
			idx = idx+((WUXGA_WIDTH -WQVGA_WIDTH)*2);
		}/*eo for*/
		free(fbuf);
		return(0);

}/*eo RGBDisplayFile_HDMI*/

```

```

/*
** Read a file into a buffer for display
** file needs to be:
** WQVGA_WIDTH  432
** WQVGA_HEIGHT 240
** FILEBUF_SIZE WQVGA_WIDTH*WQVGA_HEIGHT*2
*/
int ReadRGBFile(void *filebuf, char *fpath){

        FILE *fd=NULL;
        int fsize=0;

        /*
        ** read file into a buffer
        */
        fd = fopen(fpath, "r");
        if(fd == NULL){
                printf("error %d opening file %s\n", errno, fpath);
                return(-1);
        }/*eo if*/
        fseek(fd, 0, SEEK_END);
        fsize = ftell(fd);
        rewind(fd);
        fread(filebuf,sizeof(uint8_t),fsize,fd);
        fclose(fd);

        return(0);

}/*eo ReadRGBFile*/

/*
** initialize frame buffer color
*/
int init_fb_color(void *fbp, uint16_t color){

        uint16_t *fbptr=fbp;
        int i=0;

        /*
        ** initialize frame buffer color
        */
        fbptr=fbp;
        for(i=0; i<HVGA_WIDTH*HVGA_HEIGHT; i++){
                *fbptr = color;
                fbptr++;
        }/*eo for*/

        return(0);

}/*eo init_fb_color*/

/*
** write rgb565 buffer to a file
*/
int WriteRGBFile(void *filebuf, char *fpath){

        FILE *fd=NULL;

        /*
        ** write buffer to file
        */
```

```
        fd = fopen(fpath, "w");
        fwrite(filebuf,sizeof(uint8_t),RGB565_SIZE,fd);
        fclose(fd);

        return(0);

}/*eo WriteRGBFile*/

/*
** HaarDwt
** Transform rgb565 image to haar dwt rgb565 image
**
** input variables:
** uint16_t *imgin_ptr is the input buffer
** uint16_t *imgout_ptr is the output buffer
*/
int HaarDwt(uint16_t *imgin_ptr, uint16_t *imgout_ptr)
{

        #define QUAD_ROW_ORIGIN 432/2*240
        #define QUAD_COL_OFFSET 432/2
        #define QUAD_ROW_OFFSET 432
        #define H_NUM_ROWS      240
        #define H_NUM_COLS      432
        #define H_ROW_WIDTH     432

        /*
        ** input buffer indicies
        */
        int i=0,j=0;

        /*
        ** output buffer indicies
        */
        int k=0,h=0;
        int quad_row_offset=0;

        /*
        ** haar dwt sliding window r1c1,r1c2,r2c1,r2c2
        ** contains 2 byte rgb565 pixel
        */
        int r1c1=0,r1c2=0,r2c1=0,r2c2=0;

        /*
        ** haar dwt r,g,b sliding window pixels
        ** the 2 byte rgb565 pixel is broken into r,g,b
        ** pixels for seperate r,g,b channel processing
        */
        int red_r1c1=0,red_r1c2=0,red_r2c1=0,red_r2c2=0;
        int grn_r1c1=0,grn_r1c2=0,grn_r2c1=0,grn_r2c2=0;
        int blu_r1c1=0,blu_r1c2=0,blu_r2c1=0,blu_r2c2=0;

        /*
        ** haar dwt low pass (lp), high pass (hp) results
        ** used to calculate ll,lh,hl,hh results
        */
        int lp1=0,lp2=0,lp3=0;
        int hp1=0,hp2=0,hp3=0;

        /*
        ** store r,g,b pixels by haar dwt ll,lh,hl,hh results
        */
```

```
        uint8_t red_ll=0, red_lh=0, red_hl=0, red_hh=0;
        uint8_t grn_ll=0, grn_lh=0, grn_hl=0, grn_hh=0;
        uint8_t blu_ll=0, blu_lh=0, blu_hl=0, blu_hh=0;

        /*
        ** combine seperate r,g,b ll,lh,hl,hh pixels into single 2 byte
        ** rgb565 ll,lh,hl,hh pixels
        */
        uint16_t rgb565_ll=0, rgb565_lh=0, rgb565_hl=0, rgb565_hh=0;

        /*
        ** haar dwt
        ** this haar dwt uses a sliding window r1c1,r1c2,r2c1,r2c2
        ** to traverse the entire rgb565 image.
        **
        ** H_ROW_WIDTH*i points to the first row (r1)
        ** H_ROW_WIDTH*(i+1) points to the second row (r2)
        ** index j points to the first column (c1) in the row (r1,r2)
        ** index j+1 points to the second column (c2) in the row (r1,r2)
        */

        /*
        ** two rows at a time until all rows processed
        */
        for(i=0; i<H_NUM_ROWS; i++){

                /*
                ** calculate the output buffer row offset (quad_row_offset)
                ** and initialize output buffer column index (k) before
                ** processing the rows and columns
                */
                quad_row_offset = QUAD_ROW_OFFSET*h;
                k=0;

                /*
                ** two columns at a time until all columns processed
                */
                for(j=0; j<H_NUM_COLS; j++){

                        /*
                        ** sliding window
                        ** get rgb565 pixels r1c1,r1c2,r2c1,r2c2
                        */
                        r1c1 = *(imgin_ptr+(((H_ROW_WIDTH*i)+j)));          //r1c1
                        r1c2 = *(imgin_ptr+(((H_ROW_WIDTH*i)+(j+1))));      //r1c2
                        r2c1 = *(imgin_ptr+(((H_ROW_WIDTH*(i+1))+j)));      //r2c1
                        r2c2 = *(imgin_ptr+(((H_ROW_WIDTH*(i+1))+(j+1)))); //r2c2

                        /*
                        ** input buffer
                        ** point to beginning of next two columns
                        */
                        j++;

                        /*
                        ** unpack rgb565 pixels into red, green, blue
                        */
                        red_r1c1 = (((r1c1 & 0xf800)>>3)>>8);
```

```c
grn_r1c1 = ((r1c1 & 0x07e0)>>5);
blu_r1c1 = (r1c1 & 0x001f);

red_r1c2 = (((r1c2 & 0xf800)>>3)>>8);
grn_r1c2 = ((r1c2 & 0x07e0)>>5);
blu_r1c2 = (r1c2 & 0x001f);

red_r2c1 = (((r2c1 & 0xf800)>>3)>>8);
grn_r2c1 = ((r2c1 & 0x07e0)>>5);
blu_r2c1 = (r2c1 & 0x001f);

red_r2c2 = (((r2c2 & 0xf800)>>3)>>8);
grn_r2c2 = ((r2c2 & 0x07e0)>>5);
blu_r2c2 = (r2c2 & 0x001f);

/*
** calculate red ll,lh,hl,hh pixels
*/
lp1 = (red_r1c1+red_r2c1)/2; if(lp1>0x1f) lp1=0x1f;
lp2 = (red_r1c2+red_r2c2)/2; if(lp2>0x1f) lp2=0x1f;
lp3 = (lp1+lp2)/2; if(lp3>0x1f) lp3=0x1f;
red_ll = lp3;

hp1 = abs((lp1-lp2)/2); if(hp1>0x1f) hp1=0x1f;
red_lh = hp1*10;

hp1 = abs((red_r1c1-red_r2c1)/2); if(hp1>0x1f) hp1=0x1f;
hp2 = abs((red_r1c2-red_r2c2)/2); if(hp2>0x1f) hp2=0x1f;
lp1 = (hp1+hp2)/2; if(lp1>0x1f) lp1=0x1f;
red_hl = lp1*10;

hp3 = abs((hp1-hp2)/2); if(hp3>0x1f) hp3=0x1f;
red_hh = hp3*10;

/*
** calculate green ll,lh,hl,hh pixels
*/
lp1 = (grn_r1c1+grn_r2c1)/2; if(lp1>0x3f) lp1=0x3f;
lp2 = (grn_r1c2+grn_r2c2)/2; if(lp2>0x3f) lp2=0x3f;
lp3 = (lp1+lp2)/2; if(lp3>0x3f) lp3=0x3f;
grn_ll = lp3;

hp1 = abs((lp1-lp2)/2); if(hp1>0x3f) hp1=0x3f;
grn_lh = hp1*10;

hp1 = abs((grn_r1c1-grn_r2c1)/2); if(hp1>0x3f) hp1=0x3f;
hp2 = abs((grn_r1c2-grn_r2c2)/2); if(hp2>0x3f) hp2=0x3f;
lp1 = (hp1+hp2)/2; if(lp1>0x3f) lp1=0x3f;
grn_hl = lp1*10;

hp3 = abs((hp1-hp2)/2); if(hp3>0x3f) hp3=0x3f;
grn_hh = hp3*10;

/*
** calculate blue ll,lh,hl,hh pixels
*/
lp1 = (blu_r1c1+blu_r2c1)/2; if(lp1>0x1f) lp1=0x1f;
lp2 = (blu_r1c2+blu_r2c2)/2; if(lp2>0x1f) lp2=0x1f;
lp3 = (lp1+lp2)/2; if(lp3>0x1f) lp3=0x1f;
blu_ll = lp3;

hp1 = abs((lp1-lp2)/2); if(hp1>0x1f) hp1=0x1f;
```

```c
				blu_lh = hp1*10;

				hp1 = abs((blu_r1c1-blu_r2c1)/2); if(hp1>0x1f) hp1=0x1f;
				hp2 = abs((blu_r1c2-blu_r2c2)/2); if(hp2>0x1f) hp2=0x1f;
				lp1 = (hp1+hp2)/2; if(lp1>0x1f) lp1=0x1f;
				blu_hl = lp1*10;

				hp3 = abs((hp1-hp2)/2); if(hp3>0x1f) hp3=0x1f;
				blu_hh = hp3*10;

				/*
				** pack into ll,lh,hl,hh rgb565 pixels
				*/
				rgb565_ll = ((red_ll << 11) | (grn_ll << 5) | (blu_ll));
				rgb565_lh = ((red_lh << 11) | (grn_lh << 5) | (blu_lh));
				rgb565_hl = ((red_hl << 11) | (grn_hl << 5) | (blu_hl));
				rgb565_hh = ((red_hh << 11) | (grn_hh << 5) | (blu_hh));

				/*
				** put rgb565 pixel into ll,lh,hl,hh quadrant in
				** output buffer
				*/

				/*ll*/
				*(imgout_ptr+(quad_row_offset+k))=rgb565_ll;
				/*lh*/
				*(imgout_ptr+(QUAD_COL_OFFSET+(quad_row_offset
	+k)))=rgb565_lh;
				/*hl*/
				*(imgout_ptr+((QUAD_ROW_ORIGIN)+(quad_row_offset
	+k)))=rgb565_hl;
				/*hh*/
				*(imgout_ptr+((QUAD_ROW_ORIGIN)+QUAD_COL_OFFSET
	+(quad_row_offset+k)))=rgb565_hh;

				/*
				** output buffer
				** point to next column
				*/
				k++;

			}/*eo for*/

			/*
			** input buffer
			** point to beginning of next two rows
			*/
			i++;

			/*
			** output buffer
			** point to next row
			*/
			h++;

		}/*eo for*/

		return(0);

}/*eo HaarDwt*/
```

YUV422 TO RGB565 LUT GENERATION SOURCE CODE (lut.c)

```c
/*
** lut.c
** create yuv422 to rgb565 look up table
** write table to disk file
**
** february 10, 2018 - rlg
*/

#include <stdio.h>
#include <stdint.h>
#include <sys/mman.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#define Y_SIZE 256
#define U_SIZE 256
#define V_SIZE 256

uint16_t yuv422_to_rgb565(uint8_t Y, uint8_t U, uint8_t V);

int main(void){
	uint8_t y=0,u=0,v=0;
	int i,j,k;
	long count=0;
	uint16_t pixel;
	int err=0;
	int fd=0;
	int size = 256*256*256*2;

	fd = open("/home/rgregg/Desktop/yuv2rgb.lut", O_RDWR | O_CREAT | O_TRUNC, (mode_t)0600);
	if(fd < 0){
		err = errno;
		printf("open failed errno=%d\n", errno);
		return(0);
	}/*eo if*/

	if( lseek(fd, size-1, SEEK_SET) < 0){
		err = errno;
		printf("lseek failed errno=%d\n", errno);
		return(0);
	}/*eo if*/

	if( write(fd, "", 1) < 0){
		err = errno;
		printf("lseek failed errno=%d\n", errno);
		return(0);
	}/*eo if*/

	uint16_t *pbuf = mmap(NULL, size, PROT_WRITE, MAP_SHARED,fd,0);
	if(pbuf == MAP_FAILED){
		err = errno;
		printf("mmap failed errno=%d\n", errno);
		return(0);
	}/*eo if*/


	/*
```

```
        ** create yuv2rgb look up table
        */
        uint16_t *pb = pbuf;
        for(i=0; i<Y_SIZE; i++){
                y = (uint8_t)i;
                for(j=0; j<U_SIZE; j++){
                        u = (uint8_t)j;
                        for(k=0; k<V_SIZE; k++){
                                v = (uint8_t)k;
                                count++;
                                pixel  = yuv422_to_rgb565(y,u,v);
                                printf("%ld Y=%d U=%d V=%d RGB=%04x\n",count,y,u,v,pixel);
                                *pb = pixel;
                                pb++;
                        }/*eo for*/

                }/*eo for*/

        }/*eo for*/

        /*
        ** write yuv2rgb look up table to disk
        */
        msync(pbuf, size, MS_SYNC);

        /*
        ** clean up
        */
        close(fd);
        munmap(pbuf,(size));
        printf("done\n");

        return(0);

}/*eo main*/

/*
** convert yuv422 to rgb565
*/
uint16_t yuv422_to_rgb565(uint8_t Y, uint8_t U, uint8_t V){

        float r,g,b;
        uint8_t red = 0, green=0, blue=0;
        uint16_t r16=0, g16=0, b16=0, rgb565=0, bgr565=0;

        /*
        ** ITU-R 601
        */
        r = 1.164*(float)(Y-16) + 1.596*(float)(V-128);
        if(r<0) r=0;
        if(r>255) r=255;
        red = (uint8_t)r;

        g = 1.164*(float)(Y-16) - 0.813*(float)(V-128) - 0.391*(float)(U-128);
        if(g<0) g=0;
        if(g>255) g=255;
        green = (uint8_t)g;

        b = 1.164*(float)(Y-16) + 2.018*(float)(U-128);
        if(b<0) b=0;
        if(b>255) b=255;
```

```
        blue = (uint8_t)b;

        /*
        ** rgb565 format used in x86
        */
        /*
        r16 = ((red >>3) & 0x1f) << 11;
        g16 = ((green >> 2) & 0x3f) << 5;
        b16 = (blue >> 3) & 0x1f;
        rgb565 = r16 | g16 | b16;
        */

        /*
        ** bgr565 format used in ARM & Beaglebone Black
        */
        b16 = ((blue >>3) & 0x1f) << 11;
        g16 = ((green >> 2) & 0x3f) << 5;
        r16 = (red >> 3) & 0x1f;
        bgr565 = b16 | g16 | r16;

        return(bgr565);

}/*eo yuv422_to_rgb888*/
```

OPTIMIZED HAAR DWT SOURCE CODE (haar4.c)

```c
/*
** haar4.c
** perform haar dwt on rgb565 image - optimized
**
** richard l. gregg
** march 20, 2018
*/

#include <stdio.h>
#include <errno.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#define QUAD_ROW_ORIGIN         256*512
#define QUAD_COL_OFFSET         256
#define QUAD_ROW_OFFSET         512

#define H_ROW_WIDTH             512
#define H_COL_HEIGHT            512

int HaarDwt(uint16_t *imgin_ptr, uint16_t *imgout_ptr);

int main(void){

        FILE *fd=NULL;
        int fsize=0;
        uint16_t *imgin_ptr=NULL, *imgout_ptr=NULL;

        /*
        ** read lena_rgb565.raw file into a buffer
        */
        fd = fopen("/home/rgregg/Desktop/lena_rgb565.raw", "r");
        if(fd == NULL){
                printf("error %d opening file lena_rgb565.raw", errno);
                return(-1);
        }/*eo if*/
        fseek(fd, 0, SEEK_END);
        fsize = ftell(fd);
        rewind(fd);
        imgin_ptr = malloc(sizeof(uint8_t)*fsize);
        fread(imgin_ptr,sizeof(uint8_t),fsize,fd);
        fclose(fd);

        /*
        ** allocate output buffer for haar dwt result
        */
        imgout_ptr = malloc(sizeof(uint8_t)*512*512*2);
        memset(imgout_ptr, 0xff, 512*512*2);

        /*
        ** perform haar dwt on rgb565 image
        */
        HaarDwt(imgin_ptr, imgout_ptr);

        /*
        ** write haar dwt output buffer to a file
        */
        fd = fopen("/home/rgregg/Desktop/lena_haar_rgb565_opt.raw", "w");
        if(fd == NULL){
                printf("error %d writing file lena_haar_rgb565_opt.raw", errno);
                return(-1);
```

```
        }/*eo if*/
        fwrite(imgout_ptr, sizeof(uint8_t), sizeof(uint8_t)*512*512*2, fd);
        fclose(fd);

        /*
        ** clean up
        */
        free(imgin_ptr);
        free(imgout_ptr);

        printf("done\n");
        return(0);

}/*eo main*/

/*
** HaarDwt
** Transform rgb565 image to haar dwt rgb565 image
**
** input variables:
** uint16_t *imgin_ptr is the input buffer
** uint16_t *imgout_ptr is the output buffer
*/
int HaarDwt(uint16_t *imgin_ptr, uint16_t *imgout_ptr)
{
        /*
        ** input buffer indicies
        */
        int i=0,j=0;

        /*
        ** output buffer indicies
        */
        int k=0,h=0;
        int quad_row_offset=0;

        /*
        ** haar dwt sliding window r1c1,r1c2,r2c1,r2c2
        ** contains 2 byte rgb565 pixel
        */
        int r1c1=0,r1c2=0,r2c1=0,r2c2=0;

        /*
        ** haar dwt r,g,b sliding window pixels
        ** the 2 byte rgb565 pixel is broken into r,g,b
        ** pixels for seperate r,g,b channel processing
        */
        int red_r1c1=0,red_r1c2=0,red_r2c1=0,red_r2c2=0;
        int grn_r1c1=0,grn_r1c2=0,grn_r2c1=0,grn_r2c2=0;
        int blu_r1c1=0,blu_r1c2=0,blu_r2c1=0,blu_r2c2=0;

        /*
        ** haar dwt low pass (lp), high pass (hp) results
        ** used to calculate ll,lh,hl,hh results
        */
        int lp1=0,lp2=0,lp3=0;
        int hp1=0,hp2=0,hp3=0;

        /*
        ** store r,g,b pixels by haar dwt ll,lh,hl,hh results
        */
        uint8_t red_ll=0, red_lh=0, red_hl=0, red_hh=0;
```

```
    uint8_t grn_ll=0, grn_lh=0, grn_hl=0, grn_hh=0;
    uint8_t blu_ll=0, blu_lh=0, blu_hl=0, blu_hh=0;

    /*
    ** combine seperate r,g,b ll,lh,hl,hh pixels into single 2 byte
    ** rgb565 ll,lh,hl,hh pixels
    */
    uint16_t rgb565_ll=0, rgb565_lh=0, rgb565_hl=0, rgb565_hh=0;

    /*
    ** haar dwt
    ** this haar dwt uses a sliding window r1c1,r1c2,r2c1,r2c2
    ** to traverse the entire rgb565 image.
    **
    ** H_ROW_WIDTH*i points to the first row (r1)
    ** H_ROW_WIDTH*(i+1) points to the second row (r2)
    ** index j points to the first column (c1) in the row (r1,r2)
    ** index j+1 points to the second column (c2) in the row (r1,r2)
    */

    /*
    ** two rows at a time until all rows processed
    */
    for(i=0; i<H_ROW_WIDTH; i++){

        /*
        ** calculate the output buffer row offset (quad_row_offset)
        ** and initialize output buffer column index (k) before
        ** processing the rows and columns
        */
        quad_row_offset = QUAD_ROW_OFFSET*h;
        k=0;

        /*
        ** two columns at a time until all columns processed
        */
        for(j=0; j<H_COL_HEIGHT; j++){

            /*
            ** sliding window
            ** get rgb565 pixels r1c1,r1c2,r2c1,r2c2
            */
            r1c1 = *(imgin_ptr+((H_ROW_WIDTH*i)+j));            /*r1c1*/
            r1c2 = *(imgin_ptr+((H_ROW_WIDTH*i)+(j+1)));        /*r1c2*/
            r2c1 = *(imgin_ptr+((H_ROW_WIDTH*(i+1))+j));        /*r2c1*/
            r2c2 = *(imgin_ptr+((H_ROW_WIDTH*(i+1))+(j+1)));    /*r2c2*/

            /*
            ** input buffer
            ** point to beginning of next two columns
            */
            j++;

            /*
            ** unpack rgb565 pixels into red, green, blue
            */
            red_r1c1 = (((r1c1 & 0xf800)>>3)>>8);
            grn_r1c1 = ((r1c1 & 0x07e0)>>5);
```

```c
            blu_r1c1 = (r1c1 & 0x001f);

            red_r1c2 = (((r1c2 & 0xf800)>>3)>>8);
            grn_r1c2 = ((r1c2 & 0x07e0)>>5);
            blu_r1c2 = (r1c2 & 0x001f);

            red_r2c1 = (((r2c1 & 0xf800)>>3)>>8);
            grn_r2c1 = ((r2c1 & 0x07e0)>>5);
            blu_r2c1 = (r2c1 & 0x001f);

            red_r2c2 = (((r2c2 & 0xf800)>>3)>>8);
            grn_r2c2 = ((r2c2 & 0x07e0)>>5);
            blu_r2c2 = (r2c2 & 0x001f);

            /*
            ** calculate red ll,lh,hl,hh pixels
            */
            lp1 = (red_r1c1+red_r2c1)/2; if(lp1>0x1f) lp1=0x1f;
            lp2 = (red_r1c2+red_r2c2)/2; if(lp2>0x1f) lp2=0x1f;
            lp3 = (lp1+lp2)/2; if(lp3>0x1f) lp3=0x1f;
            red_ll = lp3;

            hp1 = abs((lp1-lp2)/2); if(hp1>0x1f) hp1=0x1f;
            red_lh = hp1;

            hp1 = abs((red_r1c1-red_r2c1)/2); if(hp1>0x1f) hp1=0x1f;
            hp2 = abs((red_r1c2-red_r2c2)/2); if(hp2>0x1f) hp2=0x1f;
            lp1 = (hp1+hp2)/2; if(lp1>0x1f) lp1=0x1f;
            red_hl = lp1;

            hp3 = abs((hp1-hp2)/2); if(hp3>0x1f) hp3=0x1f;
            red_hh = hp3;

            /*
            ** calculate green ll,lh,hl,hh pixels
            */
            lp1 = (grn_r1c1+grn_r2c1)/2; if(lp1>0x3f) lp1=0x3f;
            lp2 = (grn_r1c2+grn_r2c2)/2; if(lp2>0x3f) lp2=0x3f;
            lp3 = (lp1+lp2)/2; if(lp3>0x3f) lp3=0x3f;
            grn_ll = lp3;

            hp1 = abs((lp1-lp2)/2); if(hp1>0x3f) hp1=0x3f;
            grn_lh = hp1;

            hp1 = abs((grn_r1c1-grn_r2c1)/2); if(hp1>0x3f) hp1=0x3f;
            hp2 = abs((grn_r1c2-grn_r2c2)/2); if(hp2>0x3f) hp2=0x3f;
            lp1 = (hp1+hp2)/2; if(lp1>0x3f) lp1=0x3f;
            grn_hl = lp1;

            hp3 = abs((hp1-hp2)/2); if(hp3>0x3f) hp3=0x3f;
            grn_hh = hp3;

            /*
            ** calculate blue ll,lh,hl,hh pixels
            */
            lp1 = (blu_r1c1+blu_r2c1)/2; if(lp1>0x1f) lp1=0x1f;
            lp2 = (blu_r1c2+blu_r2c2)/2; if(lp2>0x1f) lp2=0x1f;
            lp3 = (lp1+lp2)/2; if(lp3>0x1f) lp3=0x1f;
            blu_ll = lp3;

            hp1 = abs((lp1-lp2)/2); if(hp1>0x1f) hp1=0x1f;
            blu_lh = hp1;
```

```c

                        hp1 = abs((blu_r1c1-blu_r2c1)/2); if(hp1>0x1f) hp1=0x1f;
                        hp2 = abs((blu_r1c2-blu_r2c2)/2); if(hp2>0x1f) hp2=0x1f;
                        lp1 = (hp1+hp2)/2; if(lp1>0x1f) lp1=0x1f;
                        blu_hl = lp1;

                        hp3 = abs((hp1-hp2)/2); if(hp3>0x1f) hp3=0x1f;
                        blu_hh = hp3;

                        /*
                        ** pack into ll,lh,hl,hh rgb565 pixels
                        */
                        rgb565_ll = ((red_ll << 11) | (grn_ll << 5) | (blu_ll));
                        rgb565_lh = ((red_lh << 11) | (grn_lh << 5) | (blu_lh));
                        rgb565_hl = ((red_hl << 11) | (grn_hl << 5) | (blu_hl));
                        rgb565_hh = ((red_hh << 11) | (grn_hh << 5) | (blu_hh));

                        /*
                        ** put rgb565 pixel into ll,lh,hl,hh quadrant in
                        ** output buffer
                        */

                        /*ll*/
                        *(imgout_ptr+(quad_row_offset+k))=rgb565_ll;
                        /*lh*/
                        *(imgout_ptr+(QUAD_COL_OFFSET+(quad_row_offset+k)))=rgb565_lh;
                        /*hl*/
                        *(imgout_ptr+((QUAD_ROW_ORIGIN)+(quad_row_offset+k)))=rgb565_hl;
                        /*hh*/
                        *(imgout_ptr+((QUAD_ROW_ORIGIN)+QUAD_COL_OFFSET+(quad_row_offset+k)))=rgb565_hh;

                        /*
                        ** output buffer
                        ** point to next column
                        */
                        k++;

                }/*eo for*/

                /*
                ** input buffer
                ** point to beginning of next two rows
                */
                i++;

                /*
                ** output buffer
                ** point to next row
                */
                h++;

        }/*eo for*/

        return(0);

}/*eo HaarDwt*/

```