Modeling and measuring the performance of a surveillance camera

Norbert Verhagen

September 21, 2011

Masters Thesis
2IM90

System Architecture and Networking
Department of Mathematics and Computer Science
Eindhoven University of Technology

Supervisors
Dr. ir. R.J. Bril
M. Holenderski MSc.

Abstract

The performance of an embedded system is determined by the available resources and the efficiency and effectiveness with which these resources are used. This thesis investigates a specific embedded system: a surveillance camera. The focus is on identifying which factors determine the performance with regard to a specific performance metric: the frame rate of the video stream produced by the camera. To determine these factors a performance model for the camera is created. This model relates the time spent on the different steps required to produce a video frame, the critical path in the application, to the frame rate of the video stream. The video task performs most of these steps: the retrieval of the raw image from the sensor, the encoding of the raw image, the wrapping of the encoded image into packets (packetizing) and the transmission of the packets over the network. The other tasks in the application either support or interfere with the video task.

The performance model is based on 3 other models: the resource, operating system and application models. The first model expresses the resources of the camera, i.e. processor, memories and network interface. The second model presents the operating system: μC/OS-II. The OS specifies the characteristics of the tasks in the system and the scheduling of the tasks. It also starts a number of tasks for its own operation. The third model expresses the application. It starts several tasks, including the video task, with the characteristics defined by the OS.

To ensure the accuracy of the performance model it is validated by comparing the results of the model to the results of measurements on the actual system.

Based on the performance model, 3 main impactors on the performance have been identified: encoding, packetizing and the available bandwidth on the network. When the network has sufficient available bandwidth, encoding and packetizing are the main impactors. Encoding is dominant when low resolutions and low quality levels are selected for the video, packetizing is dominant for the high resolution and high quality levels. The transmission over the network becomes a significant factor in the performance when the available bandwidth is insufficient. Each of these 3 impactors are investigated in detail and approaches for reducing their impact are presented. The approaches include a comparison of encoders, an investigation in reducing of the number of data transfers between memories during packetizing, and an approach for handling insufficient bandwidth conditions.

Contents

1 Introduction

Embedded systems are typically resource constrained for cost-effectiveness reasons. Whether it is the speed of the processor or the size of the memory, the resources are limited. Efficient and effective resource use is therefore critical for the performance of a such a system.

This thesis investigates a particular embedded system: a surveillance camera. The focus is on the frame rate of the video, a specific performance metric. The factors of the camera that determine the frame rate are investigated by modeling different aspects of the camera, such as the resources (hardware), the software and the performance. The performance model is based on the hardware and software models and relates the time spent on creating a video frame to the resulting frame rate. It is validated by comparing its results to measurements performed on the camera. After validation the performance model is used to identify specific functions during the creation of a video frame at which the performance can be improved and approaches for improvement are given.

The next sections in this chapter explore the context and background of the problem, followed by the problem description and the goals. Next, the approach to this project is determined and the contributions of this thesis are presented. This chapter is concluded with the overview of this theses and the glossary and the conventions that are used throughout this thesis.

1.1 Context & Background

The SAN group has been researching the real-time aspects of a multimedia embedded processing system as part of the ITEA/CANTATA project [2]. This project concerns a security camera designed by the industrial partner VDG security [13],[14].

The camera is in the professional security domain. Based on [47], there are three categories in this domain: operator-controlled video surveillance, basic automated video surveillance, and smart video surveillance. The camera is in the first category. There is no object recognition, which is required for the basic automated video surveillance category and there is no specific object tracking capability required for the smart video surveillance category. The security domain and the classification within the domain gives rise to specific requirements on the camera. In [52] the quality of the video is shown to be important. The frame rate of the video is also important, see [37].

Parallel to the professional security domain is the domain of consumer video streaming applications. This domain also incorporates devices that produce video and stream it over a network. However, the requirements are

different. User satisfaction, defined in terms of the quality of service (QoS) provided by the device, is the most important requirement. Based on [34], [32] and [28] the users perception on the QoS is influenced by the quality level of the video, the frame rate and fluctuations in the frame rate. Compared to the requirements for the security domain, there is overlap: the quality and frame rate. However, the trade-off between the two is different. In the security domain higher quality and lower frame rate is acceptable [52], while in the video streaming domain a lower quality is preferred over a lower frame rate [28].

The camera is a *best-effort* system. The defining characteristic of such a system is that no guarantees are given on when a particular task is to be finished, i.e. no guaranteed frame rate. *Real-Time* systems on the other hand, do give guarantees by introducing *deadlines*, specific moments at which a specific task should be finished. However, Real-Time systems concepts such as resource reservations [46], which allow the availability of specific resources at specific times to be guaranteed, and scheduling, which allows control over which task is executed when, are relevant for best-effort systems. These allow a more efficient and effective use of the available resources.

The starting point of this project was [35]. In [35] an improvement method to a security camera was proposed that allows the camera to provide a higher frame rate given fluctuating bandwidth conditions on the network. The existing situation was such that when the available bandwidth was insufficient the camera would idle until the bandwidth became sufficient. The method splits up the workload of the camera into two tasks: task 1 performs the encoding and task 2 performs the wrapping of the video frame into packets and the transmission of the packets. Task 2 has the higher priority, therefore task 2 may preempt task 1 at the cost of a context switch on preemption and a context switch on return, for details on scheduling see Section 2.1. Reservations are introduced to guarantee the processor to task 1 even if the bandwidth is fluctuating and to allow task 2 to take advantage of moments when the network is available. Fixed Priority Scheduling with Deferred Preemption (see Section 2.1) is used to schedule the tasks, in order to prevent task 2 from interrupting task 1 at moments during execution when this would result in large preemption overheads due to the data intensive characteristic of the encoder. For example, if the encoder is preempted right after a data transfer between the main memory and the local memory is completed, the preempting task may replace the data in the local memory. This would require the complete data transfer to be performed again after the encoder is allowed to continue.

The advantage of the solution from [35] is that it makes no changes to the application but only to the resource management of the OS. However,

the camera used in [35] is different from the camera investigated in this thesis. Preliminary measurements on the camera used in this project indicated that it does not idle when the bandwidth is insufficient. Also, the measurements revealed the frame rate of the video at high resolution and high video quality levels to be relatively low. Therefore an investigation was started to determine why the performance is as it is.

1.2 Problem Description & Goals

Based on [52] and [37] the quality and the frame rate of the video is of main interest to a user of the camera. The frame rate is determined by 4 main factors, Figure 1 gives an overview.

1. The rate at which the sensor can capture raw video frames.
2. The selected resolution.
3. The selected quality level.
4. The available bandwidth of the network.

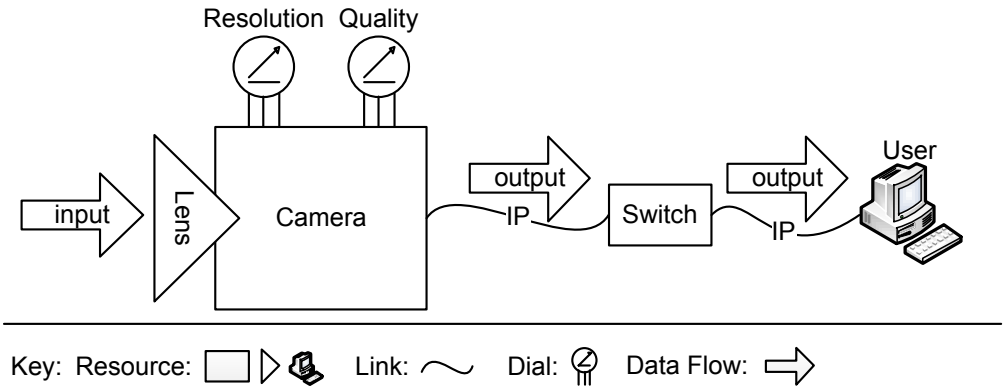


Figure 1: Camera: Top Level Overview

The capture rate of the sensor determines the upper bound of the video frame rate, as each encoded frame is based on exactly one raw frame (no interpolation is performed by the encoder). The resolution and the quality level are factors, as an increase in resolution or quality is an increase in the size of the raw video image. Thus requiring more data to be encoded, packetized and transmitted, increasing the time spent on one video frame thus reducing the frame rate. The available bandwidth of the network is a

factor, as it can reduce the speed with which the video frames are transmitted to the user. Considering that the size of the video frames remains the same but the capacity of the network to transmit that data is reduced, the result is a lower frame rate. In this thesis the network is considered up to and including the switch to which the camera is connected.

During encoding and packetizing intensive use is made of the memories in the system. Considering that memory transfers are relatively slow compared to the processor, these are investigated as well.

The goal is to express the entire process that determines the frame rate in a performance model, for a set resolution and quality level, and given a sensor capture rate and a network with fluctuating bandwidth. Based on the performance model, points in the operation of the camera that lower the frame rate can be identified and approaches to achieve improvements proposed.

1.3 Approach

The first step is the description of a system model, which identifies the core concepts such as tasks, processors and scheduling. The concepts are separated in the task and resource models and mappings between the models. The system model is then used as a basis for the creation of a resource, operating system, and application model. Which respectively describe the hardware and the OS and application that comprise the camera. The resource and software models are used to create a performance model, which shows the sequence of steps the camera executes to create a video frame and how long each step takes. Next, the performance model is validated by comparing the results of the PM to the results of the measurements taken on the camera. Finally, steps that can be improved are identified and proposals for improvement are given.

1.4 Contributions

The contributions of this thesis consist of several models, measurements on the camera and a set of proposals for improving the performance of the camera:

- System model
- Resource model (Hardware model)
- Software models (for both the operating system and application)

- Performance model
- Measurements and measurement results used to validate the performance model
- Listing of performance improvement approaches

1.5 Overview

This chapter briefly discussed the context and the motivation for the project, followed by the problem that forms the core of this thesis as well as the approach taken to solve it and the contributions that result it.

Chapter 2 presents the system model that describes the system in terms of mappings between tasks and resources. Chapter 3 contains the description of the hard- and software that make up the camera presented in the form of a resource, operating system and application model. Chapter 4 focuses on the performance model of the system and of the video task in particular. It is followed by Chapter 5 containing the validation of the performance model. Chapter 6 looks at the potential methods of improving the performance of the camera. The conclusion and the opportunities for future work are presented in the final chapter.

1.6 Glossary

Term	Description
CS	Context Switch
DMA	Direct Memory Access
EDF	Earliest Deadline First scheduling
FPDS	Fixed Priority Scheduling with Deferred Preemption
FPNS	Fixed Priority Non-Preemptive Scheduling
FPPS	Fixed Priority Preemptive Scheduling
FPS	Frames per second
GMAC	Gigabit Media Access Controller
HD	High Definition
ISEF	Instruction Set Fabric
ISR	Interrupt Service Routine
LMS	Local Memory System
MJPEG	Motion JPEG
NIC	Network Interface Card
PM	Performance Model
QoS	Quality of Service
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTCP	Real-Time Control Protocol
RTP	Real-Time Transport Protocol
RTSP	Real-Time Streaming Protocol
SBIOS	Stretch BIOS
VLIW	Very Long Instruction Word
WCET	Worst-Case Execution Time

Table 1: Glossary

1.7 Conventions

A listing of the conventions used:

- Use "." for fields of an object.
- *function_name()* to indicate function calls.
- *variable_name* to indicate variables.

2 System Model

This chapter provides the foundation for the next chapters, it identifies the key concepts required for the models in Chapters 3 and 4 and collects them into a system model.

The system model consists of a task model, which expresses the characteristics of tasks and a resource model expressing the properties of the processor, memories and network. It also maps a collection of tasks to the different resources, respectively scheduling, data to memory mapping and transmission. The mappings identify specific problems that occur when using the resources. Scheduling identifies the problem of having multiple tasks that need to be executed on a processor that is capable of handling only one task at a time. Data to memory mapping concerns the issue of data storage and data transfers between memories, considering that tasks often require input data and produce output data. The transmission mapping identifies which tasks require network access and therefore are dependent on the available bandwidth on the network. Figure 2 shows the system model.

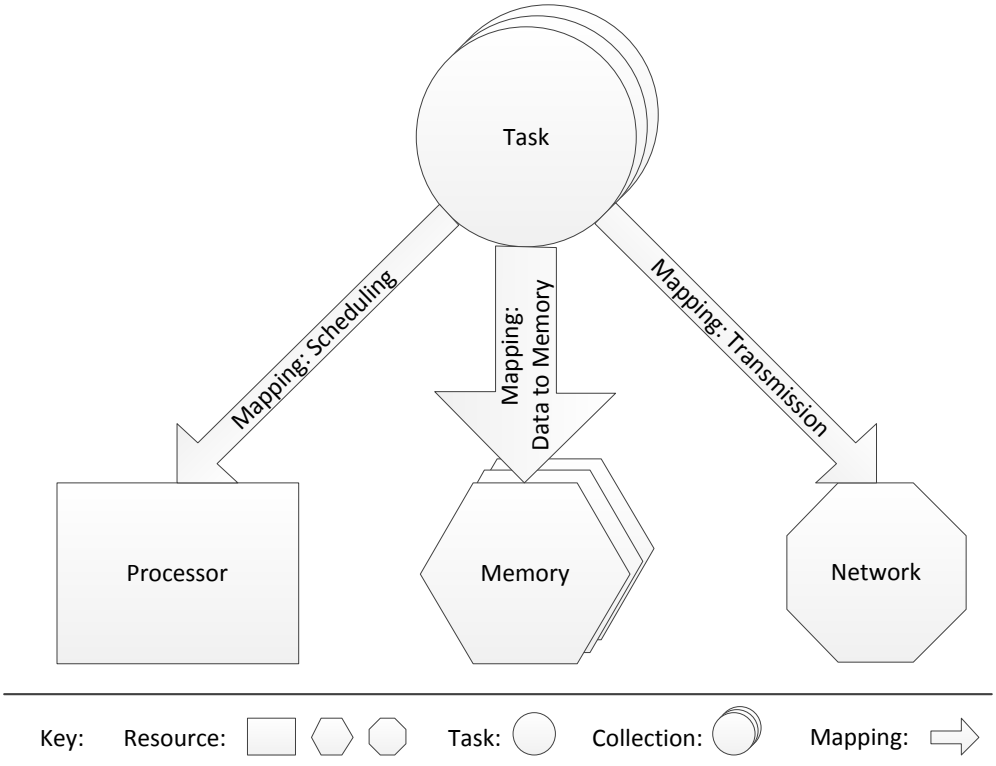


Figure 2: System Model

The following section takes a look at related work on the topic of system models. System models in general are addressed and some examples of system models of video processing applications are presented. The final topic in the related work section is the scheduling of tasks on a processor. Sections 2.2, 2.3 consider respectively the resources and tasks in more detail. Section 2.4 focusses on the mapping of tasks to the resources.

2.1 Related Work

The system model presented in the previous section is a *Model of Computation*, based on [30]. In general, models of computation consist of a collection of entities and the specification of the relationship between the entities. The entities can be expressed at different abstraction levels depending on the purpose of the model. For instance, a processor can be expressed as an entity that takes an input and produces an output, but also as a 32-bit Xtensa LX VLIW processor.

In [56] a system model of a video processing application is presented as a single video processing task with an input, output and controller process. While ideal to express the functioning of a particular task, the video task in this case, the model abstracts over the other tasks and interferences that are relevant for the performance of the camera. In [35] a model is used that focuses on two tasks, the video task and the network task, and also takes the hardware in account. Unfortunately, this model also disregards the other tasks and interferences in the system.

Many different approaches on the scheduling of tasks on processors exist [23, 43]. However, there are two main classes: *dynamic priority scheduling* and *fixed priority scheduling*. *Earliest Deadline First* (EDF) is an example of dynamic priority scheduling. This scheduling approach selects the task that has to finish first to run on the processor, as a consequence the priority of a task may change while the system is in operation. Conversely, in fixed priority scheduling the priority of a task is not allowed to change. An example is *Fixed Priority Preemptive Scheduling* (FPPS) which is used in the system model, see Section 2.4.1. Closely related is *Fixed Priority Non-preemptive Scheduling* (FPNS). In this approach, priorities of tasks are also fixed, but no preemptions are allowed to occur. If during execution of a job a higher priority job becomes available, it will have to wait until the running job has finished. The middle ground between FPPS and FPNS is held by *Fixed Priority Scheduling with Deferred Preemption* (FPDS) [22]. A task can be preempted, but only at specific points during the execution of the task.

2.2 Resource Model

In the system model, three resources are considered: processors, memories and networks:

- Processors perform the execution of tasks at a specific pace: the processor speed. This speed is expressed in the number of instructions that can be executed per second, i.e. in Hz. Defined as: Processor P with a speed $S(P)$ in instructions per second.

- Memories contain the data required by tasks. They have a fixed capacity and a fixed speed at which the data can be accessed. The capacity is expressed in bytes and the access time in the number of bytes that can be accessed per second. Defined as: the system contains X memories and memory M_x with $1 \leq x \leq X$ has a capacity $Capacity(M_x)$. The available capacity of memory M_x is $AvailableCapacity(M_x)$.

  Data can be transferred between memories via connections between memories. In the system model the assumption is made that all memories are connected to all memories. The connections have a transfer speed $TFS(M_{x1}, M_{x2})$ expressed in bytes per second.

- Networks are used to transmit data from one device to another. They have a fixed capacity: the amount of data that can be send in a particular time. Which is expressed in bytes per second. Defined as: Network N with a capacity $NetworkCapacity(N)$ and an available capacity of $AvailableNetworkCapacity(N)$.

2.3 Task Model

An application is comprised of several tasks, each consisting of a sequence of functions to be executed on a processor.

A task has a unique name, τ_i with I the number of tasks in the system and $1 \leq i \leq I$. It also has the following characteristics:

- A priority, expressed in integers with a smaller value meaning higher priority, and with 1 as the highest priority.

- Is preemptive or non-preemptive

- A period T_i, expressed in seconds, or no period.

- A workload, i.e. a sequence of instructions to perform.

- Uses a block of data d_i of a size $SizeOf(d_i)$ expressed in bytes.

A task has a fixed *priority*. Also, the priority is *unique*. The priority is used by the *scheduler*, see Section 2.4, to select which task will run when several tasks are ready to run. A task is *preemptive* if another task is allowed to interrupt the task while it is being executed. If a task cannot be preempted it is *non-preemptive*, such a task will run to its completion before another task is allowed to run.

An instance of a task is called a *job*. It is identified as job j of task i: $j_{i,j}$. If a task has a period T, the period defines the inter arrival time of its jobs. A new job is *released* at the beginning of each period of that task. The *release pattern* therefore is: $j_{i,j}$ arrives at time $j * T_i$, where T_i is the period of τ_i. A job does not need to start immediately on activation, the moment the job actually starts its work is the *start time*. The moment the job finishes its work is the *end time*. Conversely, a task can also be a-periodic, the release time of its jobs being unpredictable.

A job requires a certain amount of time to perform its workload, which can consist of many functions and each function can consist of many individual instructions. The time can be expressed in *Computation Time*. This is the time of the execution of only the instructions of the job, without any interference from other tasks. The computation time of τ_i is C_i. It is defined as the average of the computation times of all jobs of τ_i. Computation time $JC_{i,j}$ is the time job j of τ_i requires to perform its workload. This time is dependent on the speed of the processor τ_i is mapped to and determined by the computation time of each individual instruction. Each instruction is identified as instruction ι of job j and task i: $\iota_{i,j,\iota}$. The computation time of an instruction is expressed by $C_{i,j,\iota}$ the computation time of a job is therefore the sum of all instructions executed by that job, with job j having instructions $1 \leq k \leq K$, this is $\sum_{k=1}^{K}(C_{i,j,k})$.

Interference can be in the form of preemptions by other tasks (see Section 2.4) or interference from other sources such as interrupts. It can be expressed in the form of *Blocking Time* $JB_{i,j}$ for the blocking time of job j and B_i for the average of the blocking times of all jobs of τ_i.

The workload of a task can consist of several consecutive functions that form sub-sequences within the task. For instance, a task can perform some calculation followed by transmission of the results over a network. The functions of this task can be placed in either the 'calculation' sub-sequence or the 'transmission' sub-sequence. By introducing *sub-tasks*, each sub-sequence of functions can then be referred to as a sub-task. Defined as $s\tau_{i,k}$ with K the number of sub-tasks belonging to τ_i and $1 \leq k \leq K$. The sub-task inherits the priority and the preemptiveness of the of the task. The computation time of $s\tau_{i,k}$ is defined as $C_{i,k}$ and the blocking time as $B_{i,k}$. If a task is split up

into a sequence of sub-tasks the release of the first sub-task is equal to the release of the job of that task i.e. the start of the period for that job: $j * T_i$ for $s\tau_{i,1}$. For the next sub-task $s\tau_{i,k}$ the release is: $j * T_i + \sum_{n=1}^{k-1}(C_{i,n} + B_{i,n})$.

The average frame rate (AFr) can be expressed by the system model as: $AFr = \frac{1}{C_{video}+B_{video}}$, with C_{video} and B_{video} in seconds and AFr in frames per second.

2.4 Mapping

The mapping addresses three issues: scheduling, see Section 2.4.1, the mapping of data to memories and the data transfers between memories, see Section 2.4.2, and finally the mapping of tasks to the network in Section 2.4.3.

2.4.1 Scheduling

The function that decides which task can run on the processor[1] at what point in time is the *scheduler*. Under FPPS the available task with the highest priority is always set to run on the processor by the scheduler. If during the execution of a job, another job with a higher priority becomes available, the running job is preempted and the CPU executes the higher priority job. After the higher priority job has finished, the scheduler assigns the processor back to the available job with the highest priority. Changing the execution of one job to another is called a *context switch* (CS). Note that a task is mapped to a processor and therefore all jobs of that task are mapped to that processor as well.

For an example of a scheduling consider the following case: processor P and tasks τ_1 to τ_5 with corresponding priorities 1 to 5. The scheduler uses FPPS. P is executing τ_5 when τ_2 becomes available. As the priority of τ_2 is higher than τ_5, τ_2 will preempt τ_5 and start execution on P.

2.4.2 Data to Memory Mapping

A block of data d_i is used by τ_i and stored in a particular memory M_x. The condition on the storage in a particular memory is that the memory has sufficient capacity to actually contain the data. Expressed by the condition: $SizeOf(d_i) \leq AvailableCapacity(M_x)$.

A block of data can be transferred from one memory to another if these memories are connected and if the receiving memory has sufficient available

[1]For clarity, only scheduling for single processor cases is considered. Scheduling for multi-processors is out of the scope of this thesis.

capacity. The speed with which this transfer is performed is dependent on the TFS of the two memories.

For an example of a data to memory mapping, consider τ_1 with data block d_1 and memories M_1 and M_2. The $SizeOf(d_1) = 10$ bytes and $AvailableCapacity(M_1) = 5$ bytes and $AvailableCapacity(M_2) = 50$ bytes. If τ_1 attempts to store d_1 in M_1 it will fail as $SizeOf(d_1) > AvailableCapacity(M_1)$. For M_2 the condition does hold, therefore d_1 can be stored in M_2.

The transfer of d_1 from M_1 to M_2 requires the size of d_1 to be checked against the available capacity of M_2 if the condition is met the transfer is performed successfully. With a $TFS(M_1, M_2) = 5$ bytes per second, the duration of the transfer is then $\frac{d_1}{TFS(M_1,M_2)} = 2$ seconds.

2.4.3 Transmission

A block of data d_i is sent over the network N by task τ_i. The condition is that the network has sufficient available capacity to allow the transmission to take place. The capacity of the network is expressed in bytes per second, therefore the time t (in seconds) in which d_i is pushed on the network is relevant. This is expressed by the condition: $\frac{SizeOf(d_i)}{t} \leq AvailableNetworkCapacity(N)$. If the condition is not met the task will block until the available capacity becomes sufficient.

3 The Camera

This chapter discusses the relevant features of the hard and software on which the performance model in Section 4 is based. First, related work is explored in Section 3.1 on the topics of video surveillance systems, encoders and network protocols. The section is followed by the resource, operating system and application model sections. These sections focus on the features relevant for the *critical path* in the application, which is the sequence of sub-tasks required for the production of a video frame including the interference caused by other tasks on the sub-tasks. The resource model, which describes the relevant hardware of the camera, is presented in Section 3.2. The relevant features of the operating system are described in Section 3.3. These include the task model, describing the characteristics of the tasks used in both the OS as well as the application. The tasks started by the OS are investigated and the scheduling of tasks is explored. The final section (3.4) looks at the application and the video task in particular. The tasks that can preempt the video task are investigated. The mapping of the data to the memories and to the network is addressed also, focussed on the video task.

3.1 Related Work

3.1.1 Video Surveillance Systems

Many different video surveillance systems exist, two systems are explored to gain insight in the general approach taken when designing and implementing a surveillance system. Investigated are a smart video surveillance system and an operator-controlled video surveillance system.

The system proposed in [54] goes beyond the capabilities of the camera, it adds the feature of for instance object recognition and is therefore a smart video surveillance system. It is based on a 100-MHz Philips TriMedia TM-1300 processor and uses the MJPEG encoder. The system is directly connected to a PC, it does not use an IP network. The system has two parallel processes: the video process and the recognition process. The video process receives a raw video frame, copies it, encodes it, and outputs it to the PC. The recognition process receives the copy of the raw frame, performs its algorithms and provides output to the PC.

The system presented in [51] is comparable to the camera investigated in this thesis, it is also an operator-controlled video surveillance system. It has the same capabilities but consists of different hardware (a DaVinci embedded platform, made by Texas Instruments [12]) and uses a different encoder (H.264). The system uses RTP to transmit the data. The system runs two

programs on the MontaVista Linux operating system [7]: the encoder program and the LIVE555 media server program [4]. The encoder accesses a raw video frame placed in a buffer by the sensor, encodes the video and places the encoded frame in another buffer. The media server retrieves the encoded frame from the buffer and transmits it over the network.

3.1.2 Encoder

The camera uses Motion JPEG (MJPEG) [17] to encode the video, which is based on the still image JPEG (Joint Photographic Experts Group) standard. No defacto standard exists for MJPEG. However, there are a number of documented standards, see [17], [1] and [6]. MJPEG belongs to the class of *intra-frame* compression schemes, the encoding of the current frame uses no knowledge of the previous frames. As result, the video quality is directly related to the complexity of the video frame and is independent of the amount of motion in the video. Summarizing, a video encoded with MJPEG is no more than a stream of still pictures.

Many other encoders exist. Such as the encoders that use an *inter-frame* compression scheme, which does use knowledge of the previous frames and the performance is motion dependent. Examples are MPEG [31] or H.264 [53]. MJPEG does not achieve as high a compression as these more modern video formats, but it does place lower requirements on memory and processor.

3.1.3 Network Protocols

The camera makes use of three Real Time protocols: RTP, RTCP and RTSP. RTP (Real-Time Transport Protocol) is used to send the actual media over the network. RTSP (Real-Time Session Protocol) handles the creation and destruction of RTSP *sessions*. In these sessions the video is streamed to the user using RTP, while providing the user with some control over the stream. For example it allows the user to pauze the stream and continue at a later point in time. Lastly, RTCP (Real-Time Control Protocol) is used to report statistics on RTP.

RTP and RTCP were developed by the Audio-Video Transport Working Group of the Internet Engineering Task Force (IETF), the most recent publication is RFC3550 [19]. RTSP was developed by the Multiparty Multimedia Session Control Working Group (MMUSIC WG) of the IETF, the most recent publication is RFC2326 [16].

All three of the protocols are application layer protocols and usually make use of TCP or UDP protocols for transmission. RTP was specifically developed for real time data transmission, such as video and audio. However,

there is no bandwidth reservation mechanism nor is the quality of service guaranteed. RTCP can be used to determine the current quality of service, it does not provide methods to actually improve it.

An alternative to RTSP is the use of SIP (Session Initiation Protocol) [18], both protocols have the purpose of initiating and directing the video stream. SIP, however, allows the user to be mobile, by redirecting the video to the users current location [55].

3.2 Resource Model

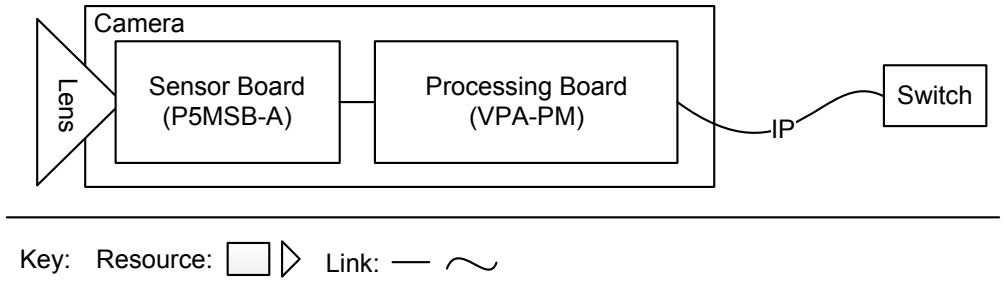


Figure 3: P5MSB-A & VPA-PM

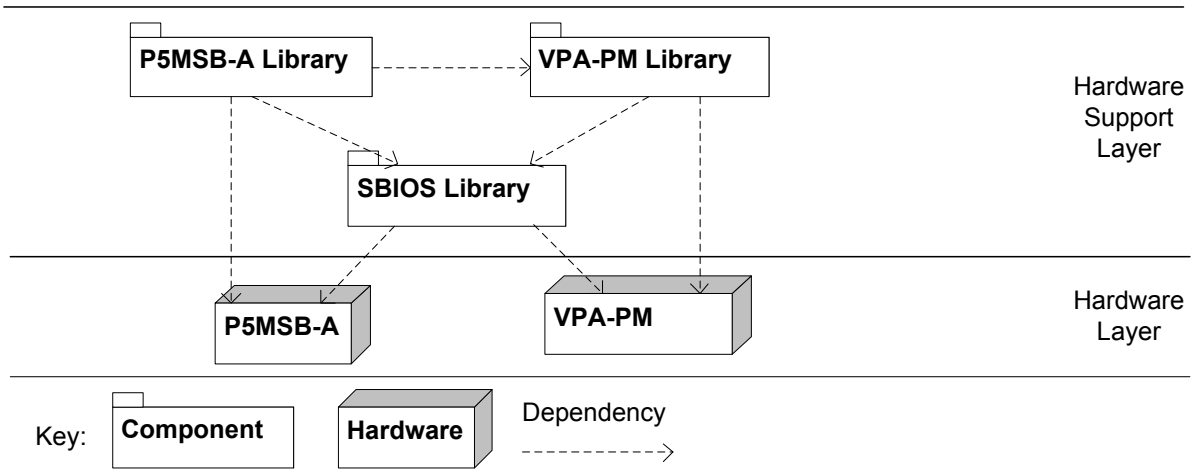


Figure 4: Hardware Layer Overview. The dependencies between the components express use-relations.

The camera is composed of two separate boards: a Sensor Board (the P5MSB-A) and a Video Processing Module Board (the VPA-PM). Figure 3 shows a general overview of the camera and is followed by Figure 4 which

displays the hardware and the software that directly supports the hardware. The next sections consider both boards in detail, focussing on the hardware components that are most relevant to the critical path: the processor, the memories and the physical methods of transferring data from memory to memory. The other hardware components that impact the performance are mentioned briefly.

3.2.1 Sensor Board

The sensor board consists of two main components: a High Definition (HD) sensor and a data port, see Figure 5. The lens focusses the image on the sensor that captures the video. It is a 5 Mega pixel (MP) sensor and supports a maximum resolution of 2592x1944 pixels [8]. However, it is the application, see Section 3.4, that determines the resolution that is actually used. After a video frame has been captured, it is transferred via the data port to the processing board where it is temporarily stored in a buffer, see Section 3.2.2.

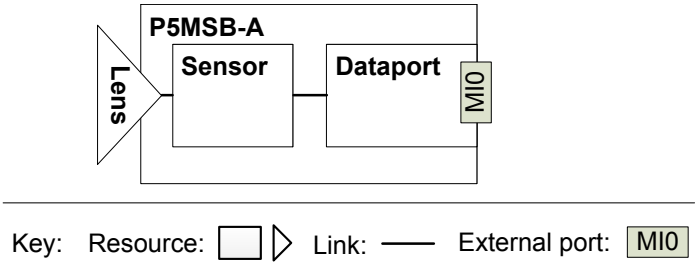


Figure 5: Sensor Board (P5MSB-A)

The sensor board is accompanied by a library, the P5MSB-A library, containing all the necessary functions and data structures for the operation of the sensor board. It is dependent on the library of the processing board.

3.2.2 Processing Board

The processing board forms the backbone of the camera, as it performs the encoding and the transmission of the video. It is comprised of several hardware components, the most relevant are shown in Figure 6. The core of the VPA-PM is a Stretch S6105 processor [39, 40]. This processor actually consists of 2 separate processors, the main processor (the S6SCP processor) and the Programmable Accelerator (the S6AUX processor). Section 3.2.2.1 contains the details. Both processors have individual caches and individual local memories.

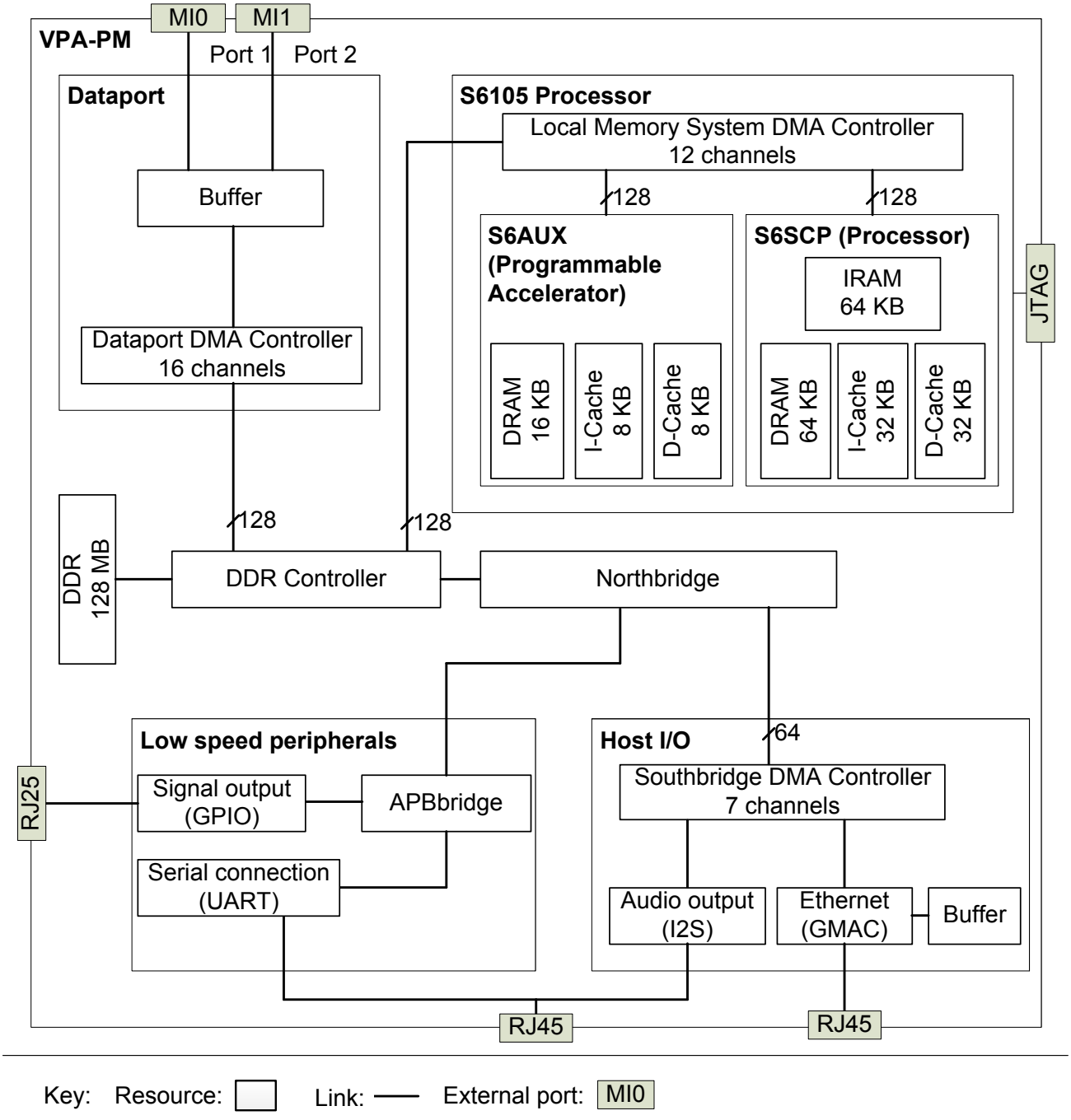


Figure 6: Processing Board (VPA-PM)

The main memory (DDR) can be accessed by both, Section 3.2.2.2 contains the details. The VPA-PM contains two data ports, only one of which is used to accept the incoming video data collected by the sensor, the second port is unused. The low speed peripherals and the Host I/O component provide the hardware needed for the output of the processing board. The serial and the ethernet ports can also be used to interact with the camera, see Section 3.2.2.3.

The VPA-PM board is supported by the VPA-PM library, see Figure 4 (on page 19), containing the functionality needed to initialize the components on the board, including the bootloader. The VPA-PM library has a dependency on the Stretch BIOS (SBIOS) library.

The next sections consider the processor, the memories and the ports of the VPA-PM in more detail.

3.2.2.1 Processors

The S6SCP processor is a 300 MHz, 32-bit Xtensa LX VLIW processor and is based on a RISC architecture. It has a 32 KB instruction cache and 32 KB data cache. The S6SCP processor also has a local memory, the Dual Port RAM of size 64 KB, also referred to as data RAM (DRAM). The main feature of the DRAM is that it can be accessed by the processor and the memory controller, see Section 3.2.2.2, at the same time, but not at the same place.

The Instruction Set Extension Fabric (ISEF) feature of the S6SCP processor (not shown in Figure 6) allows the instruction set of the processor to be extended with new instructions written in C/C++. Sets of extended instructions are organized in ISEF configurations which can be loaded and unloaded when needed. The extra instructions, used for video encoding on the camera but can be custom made for any application in general, increase the performance of the application by replacing tens or hundreds simple instructions with a single extended instruction. The disadvantage is that only one ISEF configuration can be loaded at a given time. Loading another configuration is accomplished by a DMA transfer causing additional traffic on the bus. The ISEF feature also includes an additional memory: the IRAM. It is connected directly to the local memory DMA controller and it can be used to store data, e.g. intermediate results or lookup tables, to increase the performance of the application even further.

The S6AUX processor is a coprocessor used to speed up video encoding on the camera, but can also be used for other purposes. It is a 300 MHz RISC processor with a 8 KB instruction cache, 8 KB data cache and 16 KB data RAM. Communication between the SCP and AUX processors is performed by passing messages, which are stored in queues in the DRAM

of both processors. A section of 256 bytes in both data RAMs is allocated specifically for this purpose.

3.2.2.2 Memories & DMA

The VPA-PM has several more memories than the caches and local memories mentioned in the previous section. The board supports the main memory (DDR) with a capacity of 128 MB operating at 333 MHz. It is managed by the DDR controller. The data ports and the Gigabit Media Access Controller (GMAC) have special purpose fifo buffers of unknown size and unknown speed. The video data received from the sensor board via one of the data ports (MI0 or MI1) is stored in the data port buffer also of unknown size and unknown speed. The memory on the GMAC is used to temporarily store incoming and outgoing packets. Table 2 shows an overview of all the relevant memories on the critical path with their general purpose, sizes and speeds. The available documentation [38, 39, 45] did not specify the sizes and speeds of all memories, these are marked with 'unknown' in the table.

Memory	Size	Speed	Purpose
I- and D cache	32 KB each	unknown	Instruction and Data caches
Data Port Buffer	unknown	unknown	Temporarily store data received via the data ports
IRAM	64 KB	unknown	Fast but small on chip memory for the ISEF
DRAM	64 KB	unknown	Fast but small on chip general purpose memory
DDR	128 MB	333 MHz	Large but slow main memory
GMAC Buffer	unknown	unknown	Temporarily store packets that are sent to or received from the network

Table 2: Memories Overview

The memories and caches of both the S6AUX processor as well as the S6SCP processor are connected to the local memory system (LMS), which is one of three DMA controllers on the board. The other two are the data port DMA controller and the southbridge DMA controller, see Figure 6. The data port DMA controller is used to transfer the video data from the sensor board that has been loaded into the data port buffer to the main memory. The southbridge DMA controller is used to transfer data from the local or

global memory to the GMAC, from where it is sent over the network. The number of lines between the controllers are also shown in Figure 6, there are either 64 or 128 lines. The number of available DMA channels also varies, from 7 channels for the southbridge DMA controller to 16 channels for the data port DMA controller. The number of channels determine the maximum number of concurrent DMA transfers between two memories as each DMA transfer requires one channel. It is possible to perform a sequence of DMA transfers over one channel, the sequence is then temporarily stored in either the main memory or in the DRAM in the form of individual memory-to-memory transfer requests. After a DMA transfer is completed, the channel can be closed and becomes available for another DMA transfer. While the lines and the number of channels give an indication of the capacity of the links between the memories when using DMA, the actual transfer speeds are unknown as these are not specified in [38, 39, 45].

3.2.2.3 Ports

The VPA-PM has several in and output ports, see Figure 6. The ethernet port is used to transmit the video stream and allows access to the website that is hosted on the camera, see Chapter 3.4 for the details. The second RJ45 port is not only the audio port but also the serial port that gives information on the status of the board and provides the interface to start the camera in debug mode. In this mode the software on the VPA-PM can be changed and debugged via the JTAG (IEEE Std 1149.1) port, the specifics are in [20]. For information on how to use the serial and JTAG ports see [50]. The RJ25 port is used for the signaling interface, allowing the camera to send messages to a designated location, for instance to a central server. These messages contain status information and can for instance be used to report when the camera is going to reboot.

3.3 Operating System

The operating system used on the camera is μC/OS-II [42], a real time preemptive operating system designed for embedded systems, created by Micrium [5]. It is extensively documented, starting from the high level concepts down to individual functions.

μC/OS-II was developed with portability in mind. Different ports to different CPU architectures exist, including OpenRISC, ARM and Stretch processors. The OS is composed of several required and optional components and each component consists of a component or one or more modules. Components are used to express a software architectural separation on a high

level, e.g. μC/OS-II extensions or the network suite. Modules are the parts the components consist of. Figure 7 provides an overview of the components, modules and dependencies between them. The kernel component and the configuration and port specific modules are necessary, these provide the basic OS functionality. The port module is specific to the processor on which the OS runs. The kernel component also has optional modules (not shown), these can be individually included or excluded at compile time, based on whether the functionality they provide is needed or not. The functionality includes task synchronization primitives like Semaphores, Mutexes and Flags to manage data access by multiple tasks and also inter-task communication methods such as mailboxes and queues. Not all of the optional modules mentioned are used; semaphores and flags are used, while the modules for mutexes, mailboxes and queues are not. Additional modules to extend the functionality of the OS are available, usually in the form of libraries. For the network interaction the OS is extended with modules for HTTP, DHCP, TCP/IP and UDP. The details of their use are in Section 3.4.

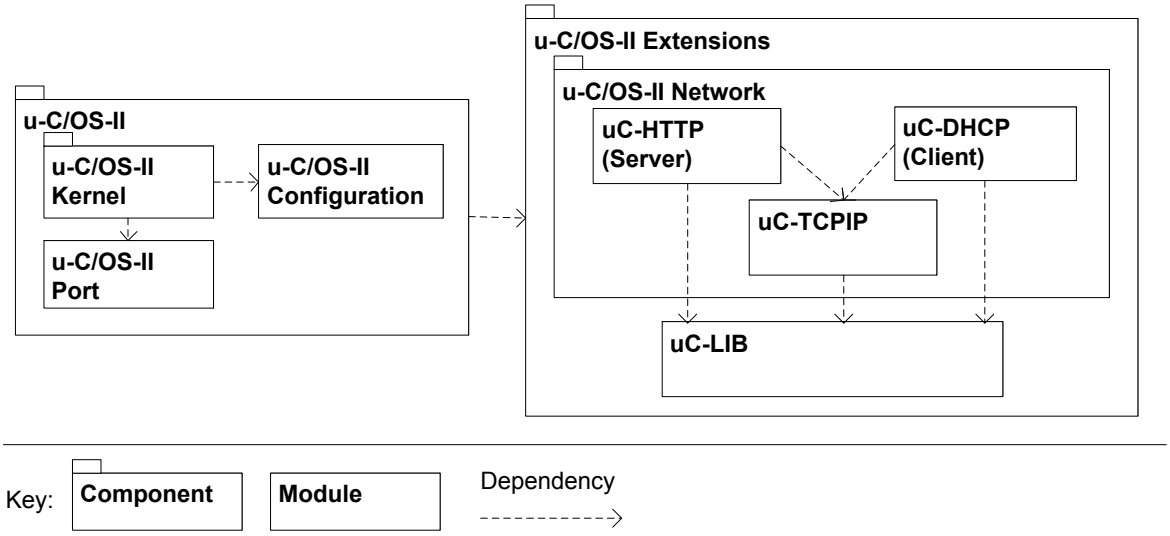


Figure 7: OS Overview. The dependencies between the components express use-relations.

In the following section the task model of μC/OS-II is presented. Also, μC/OS-II is responsible for the scheduling of all tasks, of both the OS as well as the application, therefore scheduling is discussed in the context of the OS. The mapping of data to memories and the transmission is not relevant in the OS context, as none of the OS tasks are on the critical path, and are therefore ignored. These topics are discussed in the context of the application section, see Section 3.4.

3.3.1 OS Task Model

In μC/OS-II each task typically is an infinite loop of several functions, see Figure 8 for an example. Each iteration of this loop can be considered a job of that task, with the check on the guard as the release time of the job.

```
APP_VIDEO_IN_task ()
{
  while (TRUE) do {
    app_VIDEO_IN_task_calibration(p_app_video_in);

    app_VIDEO_IN_task_encoding(p_app_video_in);
  }
}
```

Figure 8: μC/OS-II Task Pseudocode

It has a unique priority and a stack for the storing of local variables, function calls and interrupt nesting. All tasks also have a state, the possible states are shown in Figure 9.

A task starts as dormant and becomes ready after creation. The scheduler, see Section 3.3.2, selects the highest priority task from the ready group and allows it to run. If during execution a task is unable to continue, perhaps due to a preemption or a semaphore, it will be placed in the waiting group and the next highest priority task will be selected by the scheduler to run. During operation a task can be interrupted by an Interrupt Service Routine (ISR). ISRs are discussed after the details on the tasks, at the end of this section.

Usually a μC/OS-II task is preemptive, considering that the scheduler uses FPPS, see Section 3.3.2 for the details. However, the scheduler can be temporarily disabled with the function *OSSchedLock()*. This allows a task to run non-preemptively. However, a call to *OSSchedLock()* must always be eventually followed by a call to the function *OSSchedUnlock()*, to re-enable the scheduler to prevent deadlock.

A μC/OS-II task does not have an explicit period. To construct a periodic task the *OSTimeDly()* function is used. It allows the task to delay itself for a set time. This time is relative, i.e. the task can delay itself for 5 seconds. By adding such a delay at the end of the loop the task consists of a periodic behavior is approximated.

By default μC/OS-II allows up to 64 tasks to run, including the μC/OS-II system tasks. These are the core, statistics, idle and timer management tasks. The core task is the main task, its purpose is to start the other tasks. The statistics task collects information on all other tasks in the system, but only does so when there are no other tasks that can run (excluding the idle task).

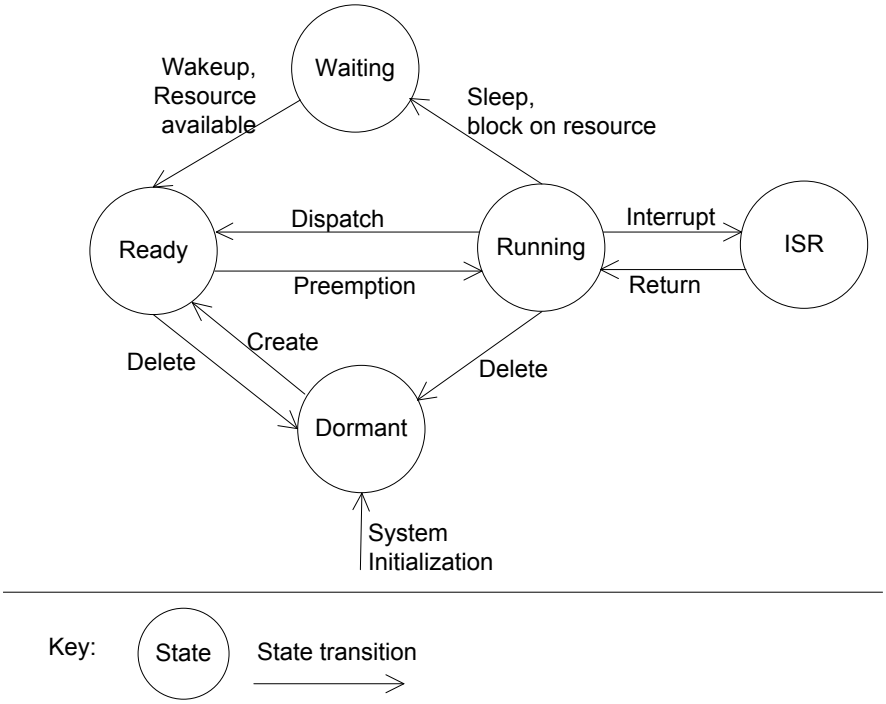


Figure 9: μC/OS-II Task States

The idle task receives processor time only when there are no other tasks available and, as the name suggests, spends processor cycles doing no useful work. The timer management task manages the timers in the OS, these are functions that count down to a specific moment in time and call a function when that moment arrives. Every instance the timer management task is called, also referred to as *updated*, the management task inspects all the active timers and those that have expired execute their assigned functions.

For the normal operation of the camera several OS tasks are required, Table 3 gives an overview, including the component or module from which they originate. The tasks are preemptive and lack a period with the exception of the tasks *OSTmr_Task* and *NetOS_Tmr_Task* these both have a period of 5 ms. The computation times of and the data used by these tasks are not listed as they are not on the critical path.

The two TCPIP tasks provide services for the sending and receiving of packets over an IP network. The HTTP (server) task manages the website that is hosted on the camera, see also Section 3.4. The DHCP (client) task handles the interaction with a DHCP server to acquire a dynamic IP address, if a DHCP server is available on the network. If no DHCP server is detected, the camera assumes a static IP address and the DHCP (client) task becomes

Task Name	Task Purpose	Task Priority	Component or Module
CORE_TASK	μC/OS-II core task	3	μC/OS-II kernel
OSTmr_Task	μC/OS-II timer management task	5	μC/OS-II kernel
NetOS_IF_RxTask	uC-TCPIP receive task	20	μC/OS-II TCPIP
NetOS_Tmr_Task	uC-TCPIP timer task	21	μC/OS-II TCPIP
HTTPs_OS_Task	uC-HTTP server task	25	μC/OS-II HTTP
DHCPc_OS_Task	uC-DHCP client task	26	μC/OS-II DHCP
OS_TaskStat	μC/OS-II statistic task	62	μC/OS-II kernel
OS_TaskIDLE	μC/OS-II idle task	63	μC/OS-II kernel

Table 3: OS Tasks Overview

dormant.

Apart from tasks, μC/OS-II also manages Interrupt Service Routines (ISRs). These are routines that interrupt a task or ISR on the occurrence of a specific event. Each event has a corresponding ISR to handle it. Whenever such an event occurs the processor saves its context (i.e. what is contained in the registers) and executes the functions defined in the ISR. The ISR is therefore executed on the stack of the running task. With the function *OS_CRITICAL_ENTER()* a task or ISR can temporarily disable the other ISRs and run uninterrupted and non-preemptively. However, a call to *OS_CRITICAL_ENTER()* must always be eventually followed by a call to *OS_CRITICAL_EXIT()*, to re-enable the ISRs. ISR nesting is limited by μC/OS-II, it allows nesting up to 255 times and ISRs are handled on a last in first out (LIFO) basis. Table 4 provides an overview of the ISRs declared in the OS.

ISR Name	ISR Purpose	Component or Module
OSTickISR	OS tick	μC/OS-II kernel
OSCtxSw	OS context switch	μC/OS-II kernel

Table 4: OS ISRs Overview

The *OSTickISR* is set at a specific frequency by the developer, it provides a periodic time tick to allow the OS to keep track of delays and timeouts. The *OSTickISR* also activates the *OSTmr_Task* to perform an update on the timers. The second kernel ISR, *OSCtxSw*, performs the required transfers of the data in the CPU registers on a context switch. The data in the CPU registers of the current (i.e. preempted) task are saved on the stack of the task and replaced by the CPU registers data from the stack of the preempting

task.

3.3.2 OS Mapping: Scheduling

The OS is only deployed on the main processor, therefore all OS tasks are scheduled to the S6SCP processor. The auxiliary processor does not use an OS, but it is used by the encoder, see Section 3.4.1.3.

The μC/OS-II scheduler is based on FPPS. To switch from one task to another the scheduler performs a context switch. This entails saving the data in the processor registers of the current task to the memory and restoring the processor registers of the new task from the memory.

3.4 Application

The application deployed on the camera uses the functionality of the hardware and the operating system to do its work. Two services are provided to the user. First, the video stream together with the audio and signaling streams. Second, it provides a method for the user to configure the camera by means of a website. While the camera allows the streaming of video to up to 4 users at the same time, the application description and the performance model, see Section 4, assume a maximum of 1 user. This is based on the fact that adding additional users would only reduce the performance of the camera.

The focus of this chapter is on the critical path of the video production process, more specifically: the steps required to produce and transmit the video. These are described in depth, while processes that support or interfere with the video production process, e.g. the audio stream and the website, are mentioned but not explored in great detail.

Figure 10 gives an overview of the layers of the camera, it shows that the application actually consists of two separate applications: the S6SCP application and the S6AUX application. Each is deployed to its specific processor, the S6SCP application is deployed, with μC/OS-II, to the S6SCP processor. Therefore the tasks of the S6SCP application are scheduled exclusively on the S6SCP processor. The S6AUX application is deployed to the S6AUX processor. The S6SCP application is the main application and uses the S6AUX application to speed up the encoding of the video, see Section 3.4.1.3. Also note that the S6AUX application does not use the operating system. It is entirely dependent on the S6SCP application.

The S6SCP application itself is divided into several components and makes use of several modules, see Figure 11 (on page 31).

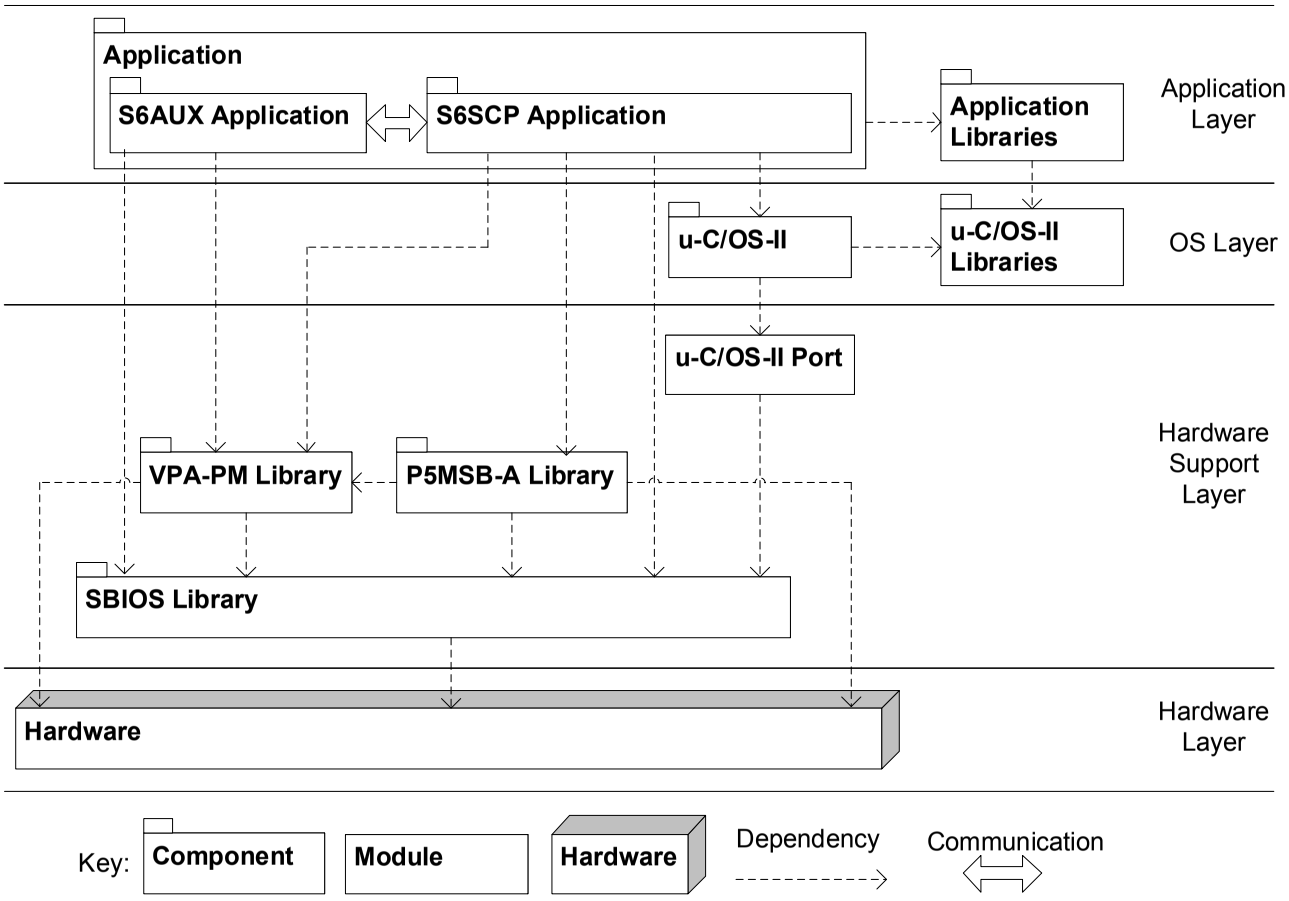


Figure 10: Application High Level Overview. The dependencies between the components express use-relations.

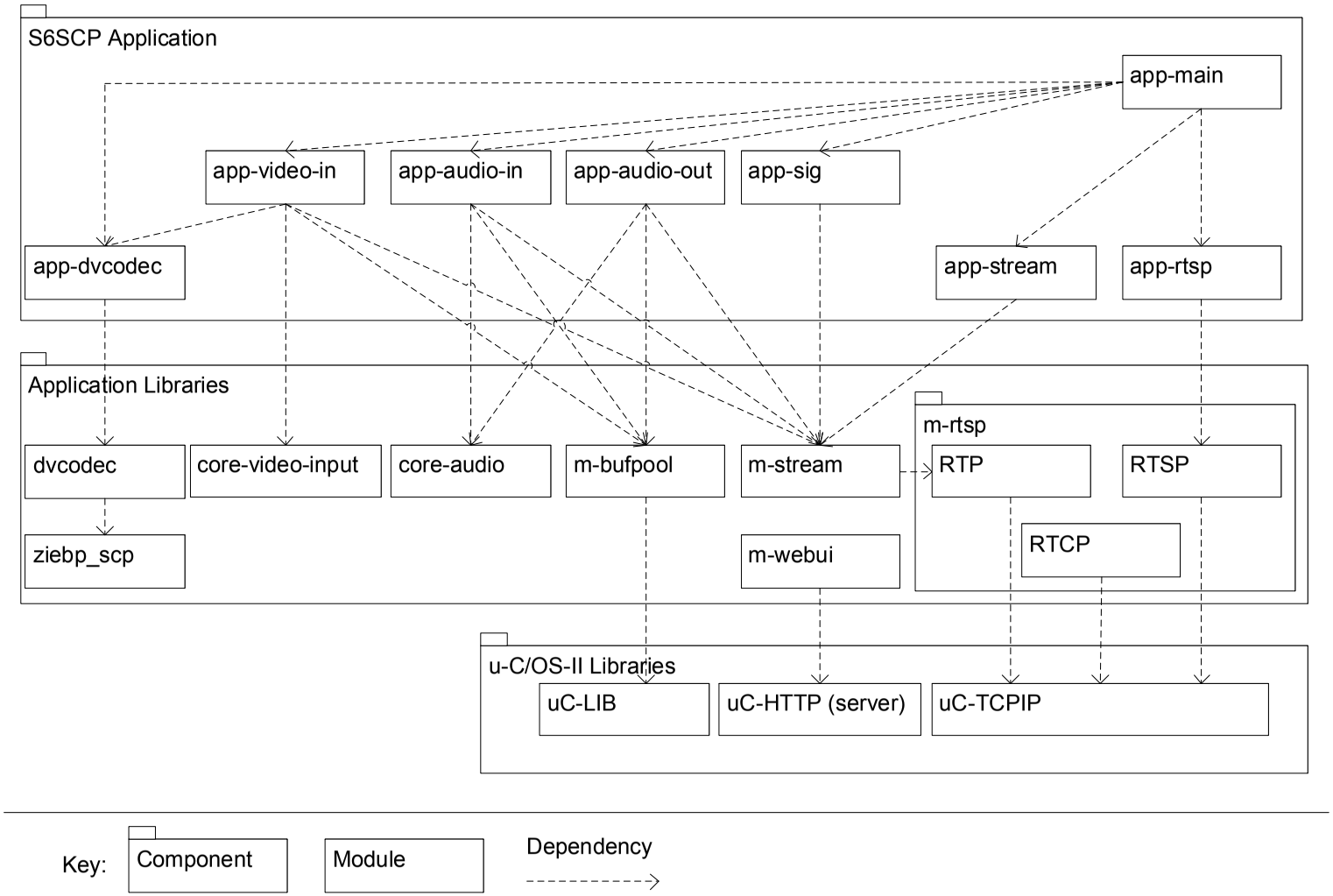


Figure 11: Application Detailed Level Overview. The dependencies between the components express use-relations.

The central modules are: *app-video-in*, *app-main*, *app-rtsp*, *app-audio-in*, *app-audio-out* and *app-sig*, these start their corresponding tasks, see Section 3.4.1. The remaining modules *app-dvcodec* and *app-stream* contain the configuration of respectively the encoder and the stream. The core task is created by the *app-main* module and starts the other tasks, both the OS tasks as well as the application tasks. The console task, from the VPA-PM library, manages the access to the camera via the serial port.

The video module, *app-video-in*, handles the video. The raw video processing is handled by *core-video-input*, encoding is located in the *dvcodec* and *ziebp_scp* modules, the storing of the encoded video is in the *m-bufpool* module and the packetizing and transmission in *app-stream*. The creation of the video task is also in the *app-video-in* module, it depends on all of these modules to perform its work. The details on the video task are in Section 3.4.1.1.

The two audio modules, *app-audio-in* and *app-audio-out* provide the audio functionality, consisting of respectively the audio signal transmitted over the network and the asynchronous audio signal outputted on the audio port. The workload of the audio module consists of the initialization of the audio encoder and audio buffer, and the creation of the two audio tasks. The audio-in task handles the audio that is sent over the network, the audio-out task provides the direct output. Both tasks retrieve the raw audio from the *core-audio* module. The audio-out task uses two ISRs for the interaction with the hardware: *sx_i2s_tc_irq* and *sx_i2s_err_irq*.

The signaling module, *app-sig*, can be used to send messages, for instance before a system reset a message can be sent to report this oncoming reset.

The network modules, *app-rtsp* and *app-stream* provide the functions necessary for the opening, closing and usage of the streams required by the video, audio and signaling components. They make use of the three Real Time protocols mentioned in Section 3.1.3: RTP, RTCP and RTSP. The video, audio and signaling components have their own individual streams. Access to the streams is managed with a semaphore, as the maximum allowed number of users is 4 and the stream can send to only one user at a time.

The website hosted on the camera allows the user to configure the different options of the camera. It contains settings for the encoder, network and miscellaneous features of the camera. The encoder settings include the quality and resolution of the video, but also brightness and more advanced options. The network settings allow changing of the network setup such as static and dynamic IP addresses and subnet masks. The miscellaneous features of the camera include the enabling or disabling of services such as the direct audio output. The website is maintained by the HTTP (server) task, see Section 3.3.

3.4.1 Application Task Model

The S6SCP Application uses the same task model as the OS, see Section 3.3, the task characteristics are therefore the same. The application does start several tasks of its own, Table 5 provides an overview. These tasks are also preemptive and lack a period with the exception of the audio-out task, it has a period of $15ms$. The computation time of and the data used by the video task are both extremely important as the video task is on the critical path. Therefore, the video task is explored in depth in the following sections. The computation time is determined during the validation in Chapter 5 based on the measurements performed in that chapter. The data usage is explored in the data to memory mapping section (Section 3.4.2). The network issues are discussed in Section 3.4.3. The computation time and the data usage of the other tasks are not listed as they are not on the critical path.

Application Task	Task Purpose	Task Priority	Component or Module
CORE_OS_Task	Application core task	22	app-main
CONSOLEs_OS_Task	Console server task	27	VPA-PM library
RTSP_Task	RTSP task	41	m-rtsp
APP_AUDIO_IN_OS_Task	Audio task	42	app-audio-in
APP_AUDIO_OUT_OS_Task	Direct audio output (Audio-out task)	43	app-audio-out
APP_VIDEO_IN_OS_Task	Video task	44	app-video-in

Table 5: Application Tasks Overview

A remark on the priorities of the tasks: While the video task is the task that does all the work concerning the video, it has the lowest priority of all of the application tasks. This is unavoidable given the current implementation of the system. If the video task was given a higher priority, the tasks with the lower priorities would, in normal circumstances, not get any CPU time, because the video task is greedy and never finishes. If, for instance, the video task was given a higher priority than the HTTP (server) task, the website would never be accessible as no CPU time would ever be assigned to the HTTP (server) task.

The S6SCP Application also declares several ISRs, see Table 6 for an overview.

Of the ISRs mentioned in Table 6 only *sx_dp_irq* and *S_ppi_tx* are on the critical path, see respectively Sections 3.4.1.2 and 3.4.1.4. The two audio ISRs and *S_ppi_rx* interfere with the critical path.

Application ISR	ISR Purpose	Component
sx_dp_irq	Data port interrupt (used by the video task)	SBIOS
sx_i2s_tc_irq	Direct audio (used by the audio-out task)	SBIOS
sx_i2s_err_irq	Direct audio error (used by the audio-out task)	SBIOS
S_ppi_rx	GMAC packet received (used by transmission)	SBIOS
S_ppi_tx	GMAC packet send (used by transmission)	SBIOS

Table 6: Application ISRs Overview

3.4.1.1 Video Task: Initialization

Before the video task can begin execution, a number of modules need to be initialized, starting with the buffers in the main memory. Two bufferpools are created, one to contain raw video frames and one to contain several encoded video frames. A bufferpool contains a sequence of individual buffers, each containing a single frame. The raw frame bufferpool consists of a place in the main memory with enough capacity for two raw frames, i.e. it consists of two buffers. The size of this bufferpool is fixed at 10 MB in the main memory. Considering the bufferpool can contain a maximum of 2 raw frames, the size of one raw frame is in the order of 5 MB.

The bufferpool for the encoded video is an instantiation of the class BUFpool with a size sufficient to contain several encoded frames. The size of the bufferpool is fixed at 8 MB, located in the main memory. The size of the encoded frames themselves is variable, it depends on the resolution, quality and contents of the video. The buffers are positioned consecutively in the bufferpool, when the bufferpool is completely filled, the placing mechanism wraps around to the beginning of the bufferpool and starts replacing the oldest buffers. In the current implementation the use of the encoded video bufferpool is limited:

1. After a frame is encoded and placed in the buffer it is immediately sent and no new frame is encoded until the transmission of the previous frame is complete. As a consequence the encoded video bufferpool contains a maximum of one frame that is ready to be sent, but no more. Therefore a bufferpool with a size of one encoded frame would have been sufficient.

2. To ensure exclusive access to the data in the video buffer pool, the semaphore bp-sem is created. It permits a task or ISR to either read or write to the buffer pool but not both at the same time. However, as only the video task has access to the encoded video buffer pool, no need exists to ensure exclusive access.

The two limitations do serve a purpose: future proofing. Both simplify the introduction of a network task to handle the transmission part of the video task as a possible method of improvement, see Section 6.2.

The final modules that need to be initialized are the encoder, during which the settings of the video are set, and the RTSP stream. After the RTSP stream is initialized the user will be able to connect to the camera using the RTSP protocol. Important to note is that the video task does not use the RTSP task to send the video, rather it uses the configuration set by the RTSP task. Therefore, the video task must wait for the RTSP task to complete its initialization.

The video task itself consists of two functions: *app_VIDEO_IN_task_calibration()* and *app_VIDEO_IN_task_encoding()*. The former performs the calibration of the camera, but is executed rarely. The latter comprises the main part of the video task and this function is therefore considered as the video task in the remainder of this thesis. *app_VIDEO_IN_task_encoding()* consists of a set-up part and a while loop. The set-up part is the initialization of the sensor board, performed once each time the video task is started. After initialization, the sensor board starts capturing raw video data and transferring it to the data port buffer, see Section 3.2.1.

After the set-up, the video task starts to produce the video frames, the start of the critical path. Note that the video frames are produced regardless of whether a user is connected to the camera (using RTSP).

The video task consists of 3 subtasks: the retrieval of the raw video frame from the data port buffer (input), the encoding of the frame (codec) and the packetizing and sending of the encoded frame over the video stream (stream), see Figure 12. The next three sections focus on the details of the three subtasks.

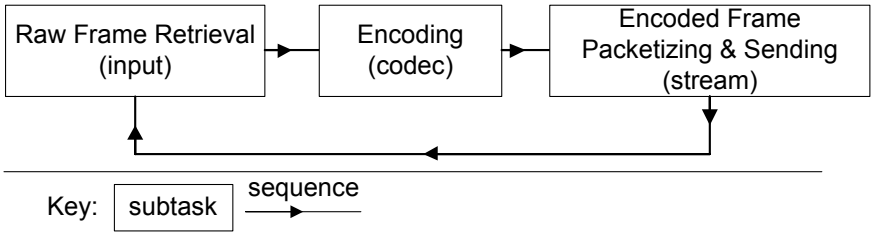


Figure 12: Video Task High Level Overview

3.4.1.2 Input Subtask

The input subtask of the video task consists of the transferring of the raw frame from the data port buffer to the main memory with a DMA transfer and some processing on the raw frame. After the initialization of the sensor board the sensor starts to capture one raw frame every 40 ms. The data port buffer can contain up to two raw frames and the oldest of the two is replaced by the next. This rate of 40 ms dictates the maximum frame rate of the video at 25 Frames per Second (FPS). If the input subtask tries to retrieve a raw frame from the buffer when none is available the video task will block until the data port ISR *sx_dp_irq* is handled, indicating the availability of a new raw frame.

3.4.1.3 Codec Subtask

The codec subtask of the video task consists mainly of allocating space in the buffer pool for the encoded video and the actual encoding using the MJPEG encoder. To increase the speed of encoding the encoder uses the ISEF feature of the processor and the auxiliary processor S6AUX, see Section 3.2. If the S6AUX is disabled, the video task will deadlock. The encoding of the raw frame is performed in blocks, due to the limited size of the S6SCP DRAM and S6AUX DRAM. First a block from the raw video frame, located in the main memory, is transferred to the S6SCP DRAM where it is encoded. During encoding data is transferred between the S6SCP DRAM and the S6AUX DRAM and possibly also between the main memory and the S6AUX DRAM. However, the exact sequence of transfers is unknown as the source code of the encoder is unavailable. After completion, the encoded block is transferred from the S6SCP DRAM to the encoded video frame buffer in the main memory. The sequence is repeated until the entire raw frame is encoded. The details on the transfer methods are in Section 3.4.2.

3.4.1.4 Stream Subtask

The network subtask of the video task is concerned with two objectives: the packetizing of the encoded frame followed by the transmission of the individual packets.

The packetizing process is determined by the network protocols, more specifically, by the datagram of the packet. Figure 13 shows the layout of the packet that is used to carry the video data. The RTP protocol defines the maximum packet size at 1420 bytes, including the RTP header. As each packet must also contain the JPEG header, the resulting maximum payload is 1400 bytes per packet.

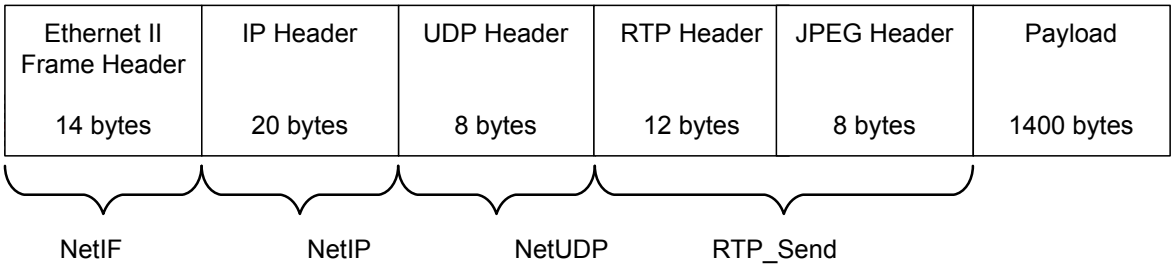


Figure 13: Packet Layout

The first step in the packetizing process is the acquisition of the first 1400 bytes from the encoded frame in the encoded frame buffer. The JPEG and RTP headers are added to this data block and the (incomplete) packet is stored in the *send buffer*, located in the main memory. The packet is then transferred to the *network buffer*, also located in the main memory. Subsequently, the UDP, IP and ethernet headers are created and added to the packet. The method of transferring the partial packet first to the *send buffer* and then to the *network buffer* is probably due to the use of third-party software. The creation and addition of the JPEG and RTP headers is implemented by VDG, while the creation and addition of the UDP, IP and ethernet headers is part of the TCP library created for the operating system (μC/OS-II TCPIP), see Figure 7 (on page 25).

After adding all headers to the packet, it is ready for transmission. In the current implementation the packet is sent straightaway, it is immediately transferred to the GMAC buffer after creation. Once the transmission has been completed, the next 1400 bytes of the encoded frame are acquired followed by the addition of the headers and the sending of the packet. This process is repeated until the end of the video frame is reached. The completion of the transmission of each of the packets is indicated by the handling of an ISR: *S_ppi_tx*. During the creation and sending of packets several memcopies and DMA transfers are performed, the next Section 3.4.2 gives a detailed description of these. Issues specific to the transmission are discussed in Section 3.4.3.

If there are multiple users connected to the camera the process of packetizing and transmission is repeated for each individual user before the next video frame is encoded.

3.4.2 Application Mapping: Data to Memory Mapping

As mentioned in Section 3.4 the S6SCP application is mapped to the S6SCP processor. The S6SCP application makes use of the memories listed in Table 2 (on page 23), the usage sequence of these memories is directly related to the critical path, see Section 3.4. Due to unavailability of the source, the S6AUX is ignored in this context. The memory transfers to and from the S6AUX DRAM are assumed to be performed concurrently with the transfers to and from the (S6SCP) DRAM. Figures 14 and 15 (on page 39) show the memory transfers performed by the video task. For clarity, the number of individual transfers are not shown, but rather the control and dataflow from one memory to another during the different subtasks of the video task. Table 7 lists the individual memcopies performed for the creation of one packet.

	Function and Purpose
memcopy 1:	*RTP_send_JPEG()* Adds RTP and JPEG headers to the packet, the first packet of a frame already contains the JPEG header, therefore for the first packet only one memcopy is needed, for all others 2 memcopies are needed
memcopy 2:	*NetConn_AddrLocalGet()* Copies the local address
memcopy 3:	*NetSock_TxDataHandlerDatagram()* Copies the source port and address (2 memcopies)
memcopy 4:	*NetConn_AddrRemoteGet()* Copies the remote address
memcopy 5:	*NetSock_TxDataHandlerDatagram()* Copies the destination port and address (2 memcopies)
memcopy 6:	*NetUDP_TxAppDataHandler()* Copies the packet into the Network buffer (in the main memory)
memcopy 7:	*NetIF_TxPktPrepareFrame()* Copies the source MAC address into the packet
memcopy 8:	*NetARP_CacheHandler()* Copies the hardware address into the packet

Table 7: Memcopy Overview for a packet

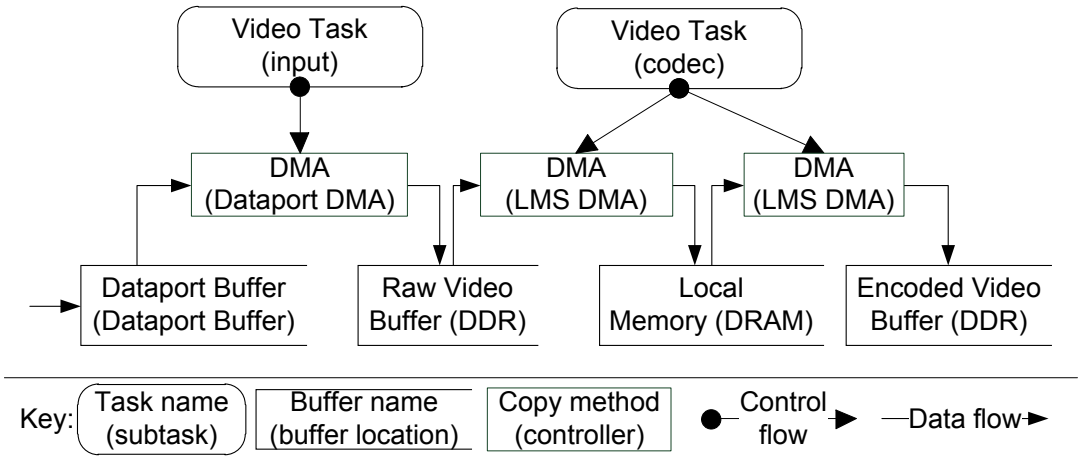


Figure 14: Input & Codec Subtasks Controlflow & Dataflow for a video frame

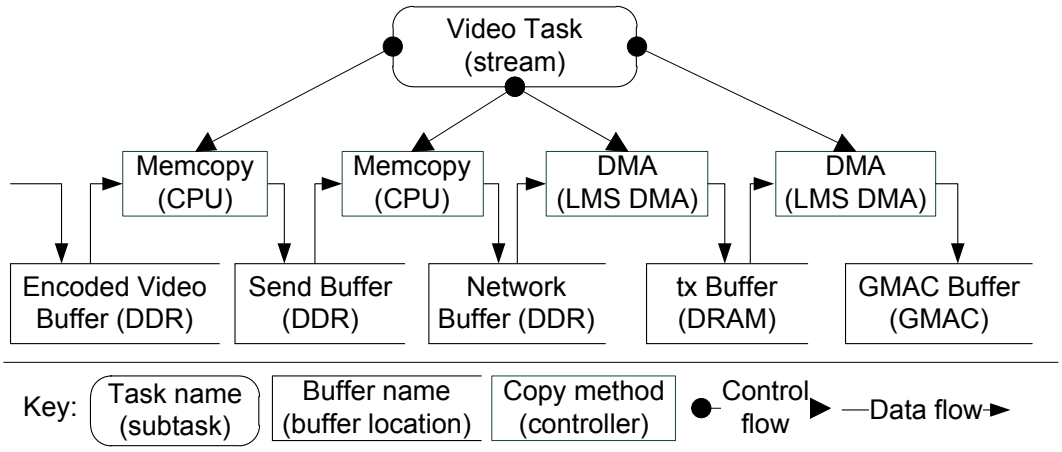


Figure 15: Stream Subtask Controlflow & Dataflow for a packet

Figure 16 shows the flow of data from one memory to another, separated in distinct pathways. It gives an overview of the hardware components required for the individual data transfers performed by the video task. During the creation and sending of each video frame, path 1 is used once for each frame, paths 2, 3, and 4 are used many times during encoding and packetizing, and path 5 is used once for each packet. The paths 2, 3 and 4 allow a choice between the use of a DMA transfer or a memcopy. Therefore, some paths (for instance path 2) has a DMA variant and a memcopy variant. Paths 1 and 5 do not have a memcopy variant, only a DMA variant.

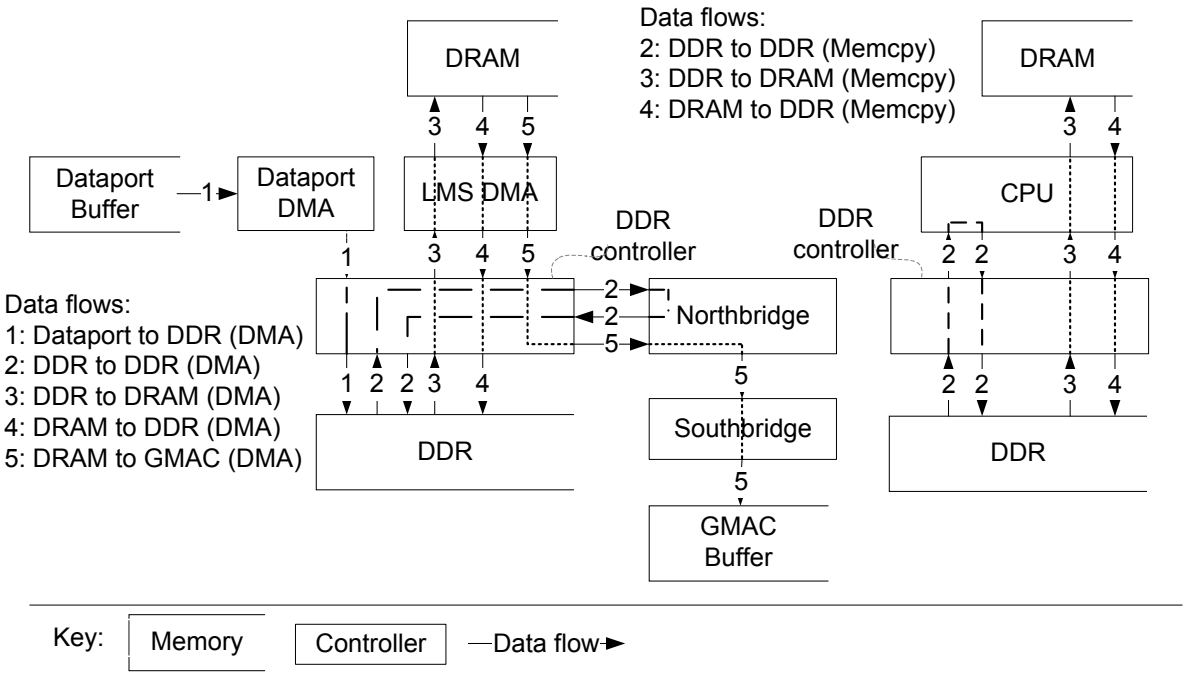


Figure 16: Application to Memory Mapping

3.4.3 Application Mapping: Transmission

Several of the Application tasks shown in Table 5 (on page 33) make use of the network:

- *RTSP_Task*
- *APP_AUDIO_IN_OS_Task* & *APP_AUDIO_OUT_OS_Task*
- *APP_VIDEO_IN_OS_Task* (the video task)

Of these tasks only the video task is on the critical path, the others are therefore ignored the context of this transmission mapping. The main issue is that the video task does not perform the available network capacity check ($\frac{SizeOf(d_i)}{t} \leq AvailableNetworkCapacity(N)$), see section 2.4.3. It merely performs a network availability check. As a consequence, the video task may attempt to send a packet on a network that has insufficient capacity. This may result in the packet being lost at the switch and reducing the frame rate of the video.

4 Performance Model

The performance model relates computation time of the video task, C_{video}, increased with the time spent on preemptions and interruptions, B_{video}, to the frame rate of the video. The performance model can then be used to identify points for improvement in the operation of the system and predict the impact of these changes. The PM is validated by measuring the performance of the actual system and comparing the measurement results to the results of the PM. Chapter 5 details the validation. After the performance model has been validated, it is used to identify the performance improvement points, the approaches for improvement are in Chapter 6.

In the next two sections related work on performance models is explored and the metrics and parameters of the performance model are introduced. In Section 4.3 the high level model is introduced. It expresses the performance of the camera on the task and sub-task level. Next, critical performance points in the codec and stream sub-tasks are identified and examined more closely, resulting in a detailed level performance model. In the final section of this chapter several hypotheses on which factors in the two performance models are potential points for improvement are presented.

4.1 Related Work

In [49] three categories of performance models are presented: *Simulation Based Performance Models*, *Holistic Scheduling Analysis* and *Compositional Models*. Simulation based performances models make use of simulators or trace-based simulation. They have the advantage that they can be made as complex as necessary. A disadvantage is that simulation may fail to produce the worst-case performance, as this is often related to incidental occurrences that may be missed by the simulators. Holistic scheduling analysis draws on the formal methods research performed on the topics of for instance the scheduling of shared resources. The disadvantage is that each new kind of application requires a new analysis. Compositional performance models consist of a combination of a collection of separate performance models each describing a particular aspect of the system. The disadvantage is the heterogeneity of embedded systems, as each system consists of its own special combination of aspects. Based on these categories, the PM presented in this chapter fits into the compositional scheduling analysis category.

In [26] simulation based performance model is presented. A simulator uses a model of the hard- and software of an embedded system as input and produces a set of Event-Traces. These provide information about timing and communication of each component. The last step is the input of these

Event-Traces in the system performance analyzer, which is configured by the architecture of the system. The output of the performance analyzer is a collection of performance metrics, these include overall performance but also execution time for each software block. Unfortunately, the required hardware and software models of the camera are not available and creating these is time consuming.

In [24] a compositional performance model is presented, concerning a video streaming server. It is based on a set of benchmarks measuring specific performance metrics such as single file acces (all clients access the same file on the server) and unique file access (all clients access a uniquely different file on the server). While not an embedded system, the principle of creating a performance model based on a set benchmarks is useful for embedded systems as well.

4.2 Performance Model: Metrics & Parameters

The performance model relates the time spent on the three sub-tasks of the video task (input, codec and stream) to the frame rate of the video stream, corrected for the interferences from preemptions from other tasks, blocking and interrupts. The setup and initialization times of the camera and of the individual tasks are ignored in the model, as the camera is intended to be active for long periods at a time and restarts are incidental.

Both the resolution and the quality level of the video have a significant impact on the computation time required to create a frame as well as on the size of a frame. The resolution is fixed at 1920x1080, the maximum resolution. The lower resolutions are disregarded because the frame rate at these resolutions is acceptable. The quality of the video is of great influence on the time spent on the three sub-tasks of the video task and is therefore a parameter of the PM. While the available bandwidth of the network is an important factor in the performance, it is not a parameter of the PM. The implementation of the application is such that, during the execution of the critical path, the available bandwidth of the network has influence only on the network availability check. See Section 4.4.2 for the details.

The PM assumes only a single user can connect at a time, for the reason explained in Section 3.4.

4.3 Performance Model: High Level

The high level description of the video task, see Figure 12 (on page 35), is taken as the starting point for the PM. This figure displays the 3 sub-tasks required for the production of one video frame. However, one sub-task was disregarded in Figure 12. After the encoded frame has been sent, the video and stream configurations are checked for changes (e.g. whether the resolution of the video has been changed by the user). These checks comprise the workload of the fourth sub-task: the finalization (fin) sub-task. As the purpose of the finalization sub-task is not directly related with the production of a frame it was ignored for the high level description. Nevertheless, the execution does take time and is therefore relevant to the performance model.

The high level model does not specifically include the interferences from other tasks and ISRs, scheduling and context switching overheads and low level interferences like cache-miss overheads. The possible interference from the network is also ignored, as the available bandwidth is assumed to be sufficient. The focus is solely on the production of one frame and the sending of that frame over an interference free network, to provide a base case for the system.

Equation (1) calculates the total computation time to produce and send a single frame, expressed by C_{frame_total}. Of each sub-task the corresponding computation time C is needed (in seconds), the sequence number of the frame is expressed by i and the quality level by q. The quality level ranges from 0 to 100.

$$C_{frame\_total}[i,q] = C_{input}[i] + C_{codec}[i,q] + C_{stream}[i,q] + C_{fin}[i] \qquad (1)$$

Equation (1) shows that the codec and stream sub-tasks are dependent on the quality level of the video, while sub-tasks input and fin are independent, their computation time is the same regardless the quality. This is due to the fact that the quality of the video has no influence on the amount of data the sensor captures and therefore the input sub-task has the same workload independent from the quality. The workload of the fin sub-task does not include any work on a video frame and is therefore independent from the quality level. This is validated in Section 5.4.1.

Equation (2) calculates the average frame rate of the video, Fr, in FPS. $input\_rate$ expresses the rate at which a raw frame is captured by the sensor in FPS. This is 25 FPS, see Section 3.4.1.2. C_{frame_total} is expressed in cycles. Ps expresses the speed of the processor in Hertz (Hz). As mentioned in Section 3.2, the processor speed for the camera is 300×10^6 Hz.

$$Fr = max(input\_rate, \frac{1}{\frac{C_{frame\_total}}{Ps}}) \quad (2)$$

4.4 Performance Model: Detailed Level

Unfortunately, equation (1) is too general to be of much use, it can only show in a broad sense what to improve. By splitting the sub-tasks into sub-components more detailed information can be gained, see Figure 17 (on page 46).

Note that for each frame $C_{stream\_packet\_build}$, $C_{stream\_packet\_send}$ and $C_{stream\_packet\_fin}$ and their sub-components are executed $n[q]$ times. Once for each packet of that frame, i.e. a frame is wrapped into $n[q]$ packets. The other sub-tasks and sub-components are executed only once per frame.

Two sub-tasks are not split up into sub-components: input and fin. The input sub-task could not be split up further due to the unavailability of the source code. The fin sub-task has an insignificant computation time compared to the other sub-tasks, see Section 5.4.1 and for that reason was not split up further. The next sections look at the codec and stream sub-tasks in more detail.

4.4.1 The Codec Sub-task

The computation time of the codec sub-task can be split up in 4 sub-components (see also Figure 17 on page 46):

- $C_{codec\_jpeg\_len}$, determines the size of the encoded frame.
- $C_{codec\_clk\_NTP}$, determines the time stamp of the encoded frame.
- $C_{codec\_buffer\_alloc}$, allocates space for the encoded frame in the buffer.
- C_{codec_step}, performs the actual encoding.

Sub-component *codec_step* is the dominating factor, see Section 5.4.2. Unfortunately it can not be split up further due to unavailability of the source code. The computation time of the other sub-components are insignificant compared to C_{codec_step}, see also Section 5.4.2.

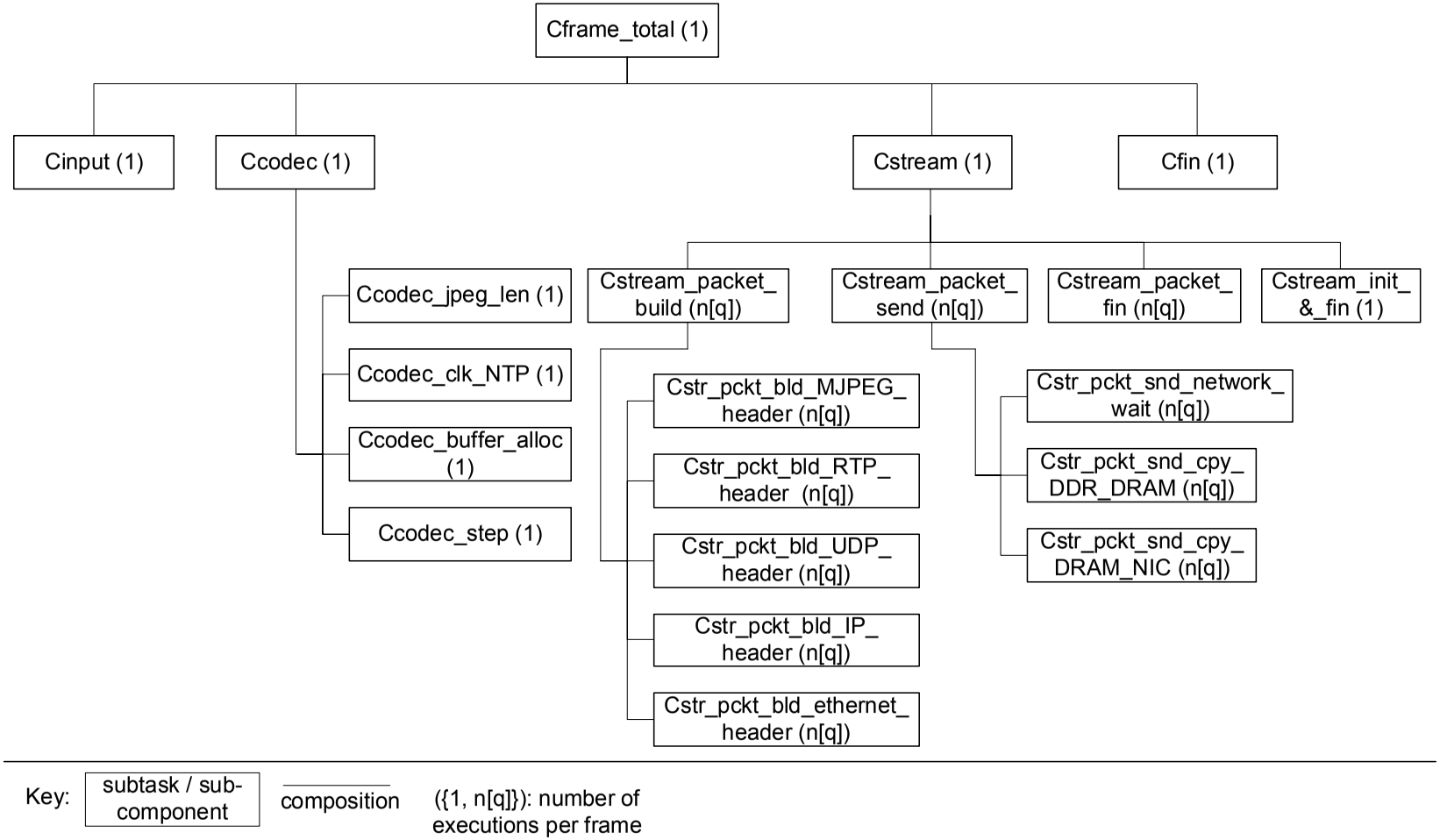


Figure 17: Computation Time Sub-tasks & Sub-Components

4.4.2 The Stream Sub-task

The computation time of sub-task stream can also be split up in 4 sub-components (see also Figure 17 on page 46):

- $C_{stream\_packet\_build}$, builds the packet by creating the 5 required headers.
- $C_{stream\_packet\_send}$, transmits the packet over the network, by checking if the network is available, copying the packet from the main memory (DDR) to the local memory (DRAM) and from the DRAM to the GMAC.
- $C_{stream\_packet\_fin}$, performs the finalization incurred for each packet, this includes the updates on the buffers after the packet has been sent.
- $C_{stream\_init\_\&\_fin}$, checks the stream for errors and clears the network buffers after the packet has been sent.

Sub-components $stream\_packet\_fin$ and $stream\_init\_\&\_fin$ are not split up further, as the impact of these is relatively small. The computation time for sub-components *stream_packet_build* and *stream_packet_send* dominate C_{stream}. See Section 5.4.2 for the details.

The dominating sub-components are explored in more detail. The division of *stream_packet_build* is based on the 5 headers needed for the construction of the packet, see Figure 13 (on page 37). Sub-component *stream_packet_send* is split up based on the network availability check and the copy actions required to transfer the packet from the main memory to the GMAC.

The computation time of *stream_packet_build* consists of the following sub-components:

- $C_{stream\_packet\_build\_MJPG\_header}$
- $C_{stream\_packet\_build\_RTP\_header}$
- $C_{stream\_packet\_build\_UDP\_header}$
- $C_{stream\_packet\_build\_IP\_header}$
- $C_{stream\_packet\_build\_ethernet\_header}$

The construction of the individual headers can be further split up into a computation component and a copy component. This allows the performance model to express how much time the construction of a header takes and how much time the copying of that header into the packet takes, see Figure 18A

(on page 49). The copy actions are performed with memcopies, see Section 3.4.2. Note that a memcopy not only takes up processing time, but also time on the bus, causing it to have additional influence on the performance by possibly causing delays due to unavailability of the bus.

Based on Figure 18A the calculation and copy time of individual headers are expressed. By summing the calculation times of the headers, the total header calculation time can be expressed as $C_{stream\_packet\_build\_computation}$ and by summing the copy times the total header copy time can be expressed as $C_{stream\_packet\_build\_copy}$. This allows a decomposition of $C_{stream\_packet\_build}$ into $C_{stream\_packet\_build\_computation}$ and $C_{stream\_packet\_build\_copy}$, see Figure 18B (on page 49).

The computation time for sub-component *stream_packet_send* can be split up into three components, see Figure 17 (on page 46):

- $C_{stream\_packet\_send\_wait}$, when the network is unavailable for transmission, the network component waits until it is available. The execution of this check takes time, even when the network is clear the time will not be 0.

- $C_{stream\_packet\_send\_copy\_DDR\_DRAM}$, the copying of the packet from the main memory (DDR) to the local memory (DRAM). This copy is performed with a memcopy.

- $C_{stream\_packet\_send\_copy\_DRAM\_GMAC}$, the copying of the packet from the local memory (DRAM) to the GMAC. This copy is performed with a DMA transfer.

Once the packet has arrived at the GMAC it is sent. This takes up no processing time on the CPU and is therefore not explicitly mentioned in the performance model.

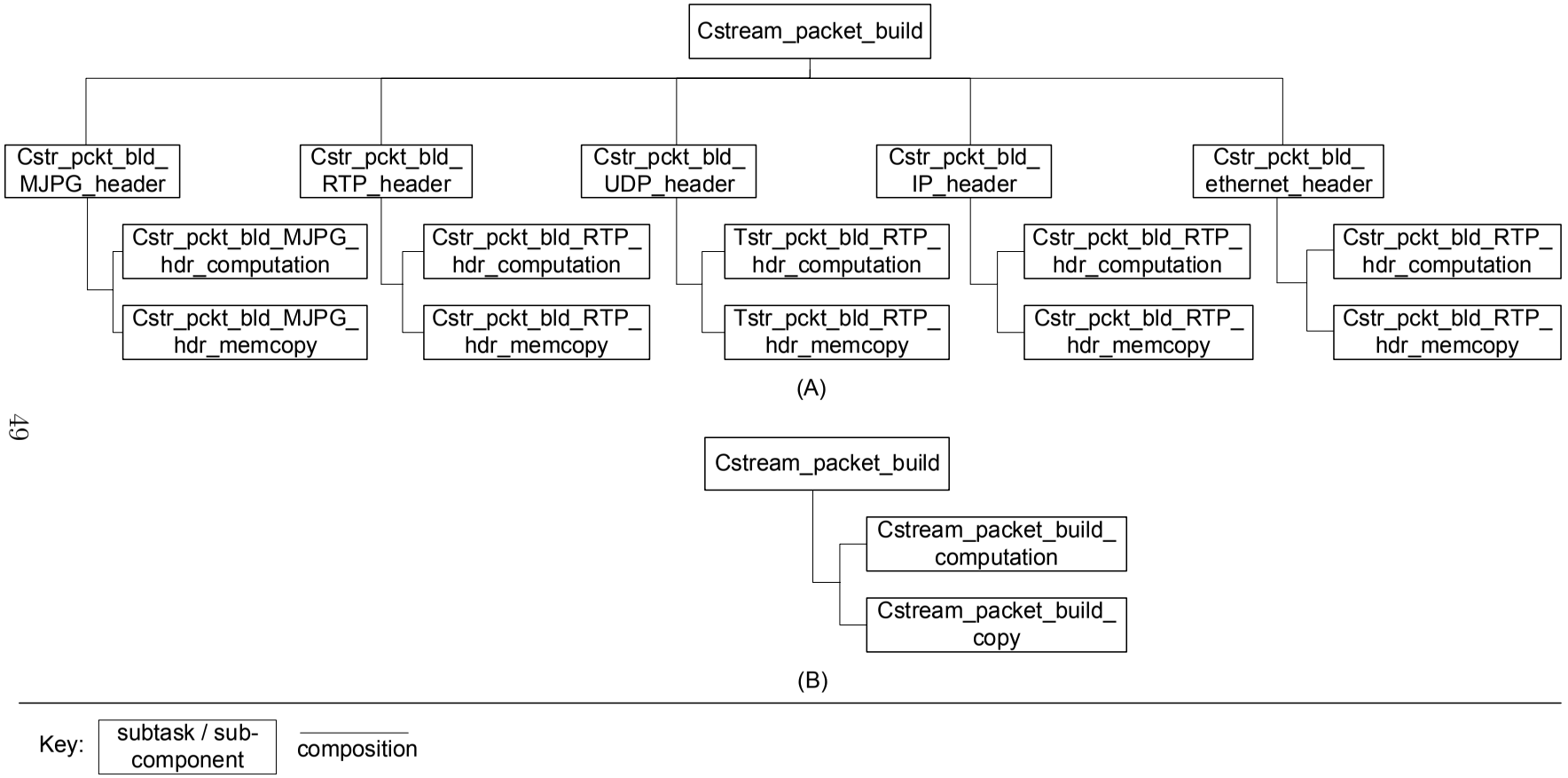


Figure 18: Computation Time Packet Build Sub-Components

4.4.3 Detailed Level Performance Model

Expanding the equations given in Section 4.3 with the information from Sections 4.4.1 and 4.4.2 gives equation (3). With C the computation time, i the sequence number of the frame, q the quality level and n the number of packets that make up the frame.

$$
\begin{aligned}
&C_{\text{frame\_total}}[i,q] \equiv \\
&\quad C_{\text{input}}[i] + \\
&\quad C_{codec\_jpeg\_len}[i] + C_{codec\_clk\_NTP}[i] + C_{codec\_buffer\_alloc}[i] + C_{codec\_step}[i,q] + \\
&\quad \sum_{p=1}^{n[q]} (C_{\text{stream\_packet\_build}} + C_{\text{stream\_packet\_send}} + C_{\text{stream\_packet\_fin}})[i] + C_{\text{stream\_init\_\&\_fin}}[i] + \\
&\quad C_{\text{fin}}[i]
\end{aligned}
\tag{3}
$$

Expanding $C_{stream\_packet\_build}$ for the individual headers, see Figure 18A (page 49), gives:

$$
\begin{aligned}
&C_{stream\_packet\_build} \equiv \\
&C_{stream\_packet\_build\_MJPG\_header\_computation} + C_{stream\_packet\_build\_MJPG\_header\_memcopy} + \\
&C_{stream\_packet\_build\_RTP\_header\_computation} + C_{stream\_packet\_build\_RTP\_header\_memcopy} + \\
&C_{stream\_packet\_build\_UDP\_header\_computation} + C_{stream\_packet\_build\_UDP\_header\_memcopy} + \\
&C_{stream\_packet\_build\_IP\_header\_computation} + C_{stream\_packet\_build\_IP\_header\_memcopy} + \\
&C_{stream\_packet\_build\_ethernet\_header\_computation} + C_{stream\_packet\_build\_ethernet\_header\_memcopy}
\end{aligned}
\tag{4}
$$

Expanding $C_{stream\_packet\_build}$ while abstracting over the headers, see Figure 18B (page 49), gives:

$$
\begin{aligned}
&C_{stream\_packet\_build} \equiv \\
&C_{stream\_packet\_build\_computation} + C_{stream\_packet\_build\_copy}
\end{aligned}
\tag{5}
$$

Expanding $C_{stream\_packet\_send}$, see Figure 17 (page 46), gives:

$$
\begin{aligned}
&C_{stream\_packet\_send} \equiv \\
&C_{stream\_packet\_send\_wait} + \\
&C_{stream\_packet\_send\_copy\_DDR\_DRAM} + C_{stream\_packet\_send\_copy\_DRAM\_NIC}
\end{aligned}
\tag{6}
$$

Comparing equations (1) and (3) reveals that in equation (1) C_{stream} has quality as a parameter and in equation (3) this parameter appears absent.

All of the sub-components are independent from the quality. The reason is that the size of the packets is fixed, see Section 3.4.1.4. The workload of $C_{\text{stream\_packet\_build}}$, $C_{\text{stream\_packet\_send}}$, and $C_{\text{stream\_packet\_fin}}$ is therefore on average constant. The exception is the last packet of each video frame, this packet may be smaller than the maximum size. During the validation of the detailed performance model, see Section 5.4.2, the independence is confirmed based on the measurement results. The number of packets, $n[q]$, is determined by the quality.

Three of the 4 sub-components of the codec sub-task: *codec_jpeg_len*, *codec_clk_NTP* and *codec_buffer_alloc* are also independent from the quality.

The equations presented in this chapter are based on computation time, interference from preemptions by other tasks and ISRs is disregarded. The interference from bus unavailability during copy actions is also ignored. The preemption interference is relatively easy to account for by determining the tasks and ISRs that preempt the video task and detecting when they preempt and for how long a period, see Section 5.4.

The delays caused by bus unavailability are more difficult to determine. It is not possible to determine these delays in the codec sub-task due to the unavailability of the source code of the encoder. Required is the exact sequence of memory transfers with their sources and destinations, before interferences on these transfers can be detected. In de stream sub-task the transfers are determinable, however, the memory transfers in this sub-task consist mainly of memcopies. Considering that a memcopy is performed by the CPU, it can only be interfered with by a concurrent DMA transfer initiated by the stream sub-task itself or by another task or ISR. Sections 3.4.1.4 and 3.4.2 show that at the time of the memcopies, the stream sub-task has initiated no DMA transfers. In Section 5.4 the interfering tasks and ISRs are determined and these also do not initiate DMA transfers.

4.5 Performance Improvement Hypotheses

Both the high level as well as the detailed level performance models give rise to a number of hypotheses on potential improvements:

1. The input sub-task.
2. The codec sub-task.
3. The stream sub-task.

    3a. $C_{stream\_packet\_build\_copy}$.

3b. $C_{stream\_packet\_build\_computation}$.

3c. $C_{stream\_packet\_send\_wait}$, especially under fluctuating bandwidth conditions.

3d. $C_{stream\_packet\_send\_copy\_DDR\_DRAM}$.

3e. $C_{stream\_packet\_send\_copy\_DRAM\_NIC}$.

4. Preemptions by other tasks and interruptions by ISRs on the codec sub-task cause significant overhead, making context switching a point for improvement.

If the stream sub-task is not a point for improvement, then the sub-components of the stream task are also not improvement points.

5 Performance Model Validation

The validation of the performance model presented in the previous chapter consists of the following steps:

1. Determination of the metrics to be measured.
2. Selection of a measurement method and the determination of the overheads & uncertainties of that method.
3. Creation of the testing environment.
4. Validation of the performance model based on the measurement results.

Point 1 is dictated by the purpose of the performance model. Required are the computation time of tasks and of specific functions within these tasks. The model relates these computation times to the frame rate of the video, as expressed by equations (1) and (3). To validate both equations, the frame rate must also be measured. The computation time measurements are used with the equations to calculate the frame rate, which is then compared to the measured frame rate. By performing the frame rate measurements on the PC that receives the video stream from the camera, the influence of the network can be taken into account. Given a clear network with sufficient available bandwidth, the frame rate at the camera and the frame rate at the PC is the same. If the network is congested, there will be a difference dependent on the amount of congestion.

The next section looks in more detail at different methods of performing measurements on embedded systems. In the following sections the measurement approach, the testing environment and the actual validation of the performance model are presented.

5.1 Related work: Performance Measurement Approaches

Performance measurement approaches can be categorized as invasive or non-invasive. Invasive methods change the operation of the system to acquire the measurements, non-invasive methods take a passive approach, the existing system remains unchanged. Non-invasive methods have a severe limitation; the system is considered a black box and only the input to the system and output of the system can be used to acquire measurements, making it impossible to measure individual functions. Exceptions do exist, such as a bus-spie, a process that can read the values on the bus, but often these require additional hardware and software.

The application for the camera is created with the development environment Stretch IDE, it provides a simulator [11] which allows measurements to be taken non-invasively, however the implementation of the application greatly complicates the use of the simulator. The main problem is that the application expects input from the sensor, which is not available in the simulator. This can be circumvented by changing the application to use a raw frame, one that has been captured and stored at an earlier time, over and over again. However, this causes the non-invasive character of the measurement approach to be lost and the collected results are based on one single raw frame instead of an average over many raw frames, making the relevance of the results questionable.

Stretch IDE also provides a measurement method called profiling [11]. It allows the collection of data on the performance of the application by sampling the program counter, which points at the next instruction that is to be executed, at set intervals. Profiling is designed to "identify hot-spots in the application", these are sets of instructions where the majority of the execution time is spent, the same places where optimization will have the largest impact on the performance. Profiling can be used on both the simulator as well as a real system. However, to use profiling on an actual system, referred to as statistical profiling, the application must be extended (at compile time) with a linker support package that supports profiling, thereby changing the application. Also, the information gathered from an actual system is more limited compared to a simulated system. From a real system only histogram type information can be gathered, e.g. which instructions are executed how many times. During testing the data is temporarily stored in a buffer and downloaded after execution has ended, the communication is performed over the JTAG port, see Section 3.2.

The developer of μC/OS-II provides a measurement application called μC/Probe [9] to work within the operating system. This program can be used to extract data from a running embedded system application, via a serial, ethernet, USB or JTAG port. The type of data consists of the values of global variables, which have been selected by the developer beforehand. This method also introduces overhead, the amount is dependent on the transmission method used, the serial connection for instance has a greater impact on the system than the JTAG connection. Also, the application has to be extended with a set of functions that enable and perform the sending of the data.

5.2 The Measurement Approach

Measuring the performance of the camera or an embedded system in general is not as straightforward as it may seem. The problems are comparable to the issues that arise when estimating the WCET of an embedded system [29]. The basic idea is that the program flow, i.e. the critical path, is used to generate a trace of the system. This trace contains al the instructions performed while executing the critical path, including the interference caused by other processes in the application and the OS. The next step is the correction of the calculation for the hardware interference caused by memory transfers and the network.

A listing of the challenges with measuring the performance of an embedded system:

1. Determining what is to be measured and what are the requirements on the measurement? What can influence or interfere with the measurement? How do the measurements influence the behavior of the system?
2. How to store the measurement results?
3. How to retrieve the measurement results?
4. How to make measurement reproducible?

Point 1 is defined by the metrics chosen to express the performance, i.e. the frame rate of the video, see Section 4.2. Therefore, the time it takes to perform a particular operation, for instance the encoding of a video frame or the transmission of a packet, is critical. The primary requirements on the measurements for the camera are a known accuracy level of the measurements and the ability to measure individual functions. The accuracy is influenced by the overhead introduced by the measurement itself and by the interference on the measured functions. Interference can be caused by the application itself, in the form of other tasks or interrupts therefore measurements should be performed in critical sections (with *OS_CRITICAL_ENTER()* and *OS_CRITICAL_EXIT()*. It can also be caused by the hardware, in the form of interference on the shared bus or by cache misses. The hardware also has influence on the measurement itself, performing a measurement and storing the result in a particular memory can give a different result then when storing the result in another memory. See Section 5.2.1 for the details. To increase the accuracy of the measurement result, not individual measurement results are used, but the average of a large sample set of measurement results. The standard deviation on the set of measurement results is used to determine the

reliability of the measurement result. A standard deviation that is relatively low ($\leq$ 10% of the average) makes the measurement reliable.

The ability to measure individual functions is necessary due to the implementation of the application, for instance, the encoding of the video is performed with a single function call. Finally, performing a measurement is disruptive to the behavior of the system. The disruption can be small, if for instance the measurement can be performed with a few short instructions. However, if the measurement takes significant time, the behavior is changed thereby invalidating the measurement.

Points 2 and 3 occur due to the embedded nature of the system, usually there is no screen directly attached to the system on which to display the measurement results. Therefore, the data needs to be stored and sent to an outside party in such a way that it minimizes interference on the measurements themselves. For the storage the most suitable data structure and memory to store the data structure in needs to be determined, and for the output the most suitable output port of the system.

Point 4 is satisfied by providing the implementation of the application with the performance measurements added to it and the definition of the environment (see Section 5.3). Based on these two points the measurements are reproducible.

While the bulk of the measurements are performed on the camera, the determination of the frame rate of the video is the noteworthy exception. This measurement is performed on the PC connected to the camera. For the connection to the stream and to display the video VLC [15] is used, an open source media player. The benchmarking software FRAPS [3] is used to determine the frame rate of the video, as VLC does not do this automatically.

For the time measurements on the camera any one of the methods described in Section 5.1 can be used. However, based on the disadvantages of each of the methods, another method was chosen: the *printf()*, see Section 5.2.1. This is a completely open method, as it does not rely on third party software, and is well suited for time measurements on individual functions. However, it is less suited for system overview measurements, for instance to determine when the video task is preempted by another task or ISR. Grasp, see Section 5.2.2, was designed with the intent to reveal exactly this, making it ideal for the system overview measurements. Grasp is also completely open, however, it does require an additional library to be added to the application. The next two subsections describe the *printf()* method and Grasp in more detail, addressing the uncertainties of both measurements.

5.2.1 The *printf()* Method

One direct way of performance measuring is the invasive method of adding *printf()* statements to the source of an application, resulting in the direct output of the measurement results. There are several drawbacks, but these can be overcome. The main drawback is the overhead that is incurred due to the act of outputting. The size of the overhead depends on what the *printf()* outputs, i.e. how the *printf()* is implemented. Where the *printf()* function outputs to, this can be a screen or some other connection like a serial port, is also relevant. While the output to a screen can be relatively quick, the output over a serial port is comparatively slow. One solution is to store the measurement values temporarily in the memory and output all values at a time outside of the measurement. This may also cause an overhead, the storing of the values requires memory access which in turn can interfere with the normal operation of the device. However, the interference caused is smaller than the interference caused by the direct-output method, considering that the direct output requires data transfers to peripheral devices, such as the serial port and the storing method can store the data in one of the local memories. The calculation time of the function can be determined by determining the time before starting the execution of a function and after completion, followed by calculating the difference. Followed by correcting the difference for any interference by other tasks or ISRs. If the begin and end points can be placed in such a way that the time measured does not include the time spent on storing the values, the result will be unaffected by the storage overhead.

The performance model reflects the amount of time a certain operation in the video task takes, whether it is one of the sub-tasks (eg. codec or stream) or a function in one of the four sub-tasks. Figure 19 shows the pseudo code of a measurement using the *printf()* method to determine exactly such an interval. The code is based on the analysis of Section 5.1. It addresses the three steps: acquiring, storing and outputting of the data.

In total, 1000 measurement points are collected in the example of Figure 19 and stored in the array *log*. The number 1000 is selected to gain a large enough sample for the measurements to be reliable. Note that 3 assignments are performed: to the variables *start* and *stop* and to array *log*. The assignment to the variable *start* is part of the measurement, as the value is determined before it is assigned to *start*. The assignment to *stop* occurs outside of the measurement, as it is performed after the stop-time has been determined. The same applies to the assignment to array *log*. As a result, the time necessary for storing the value for *start* in the memory is an overhead in the measurement. This overhead is determined below.

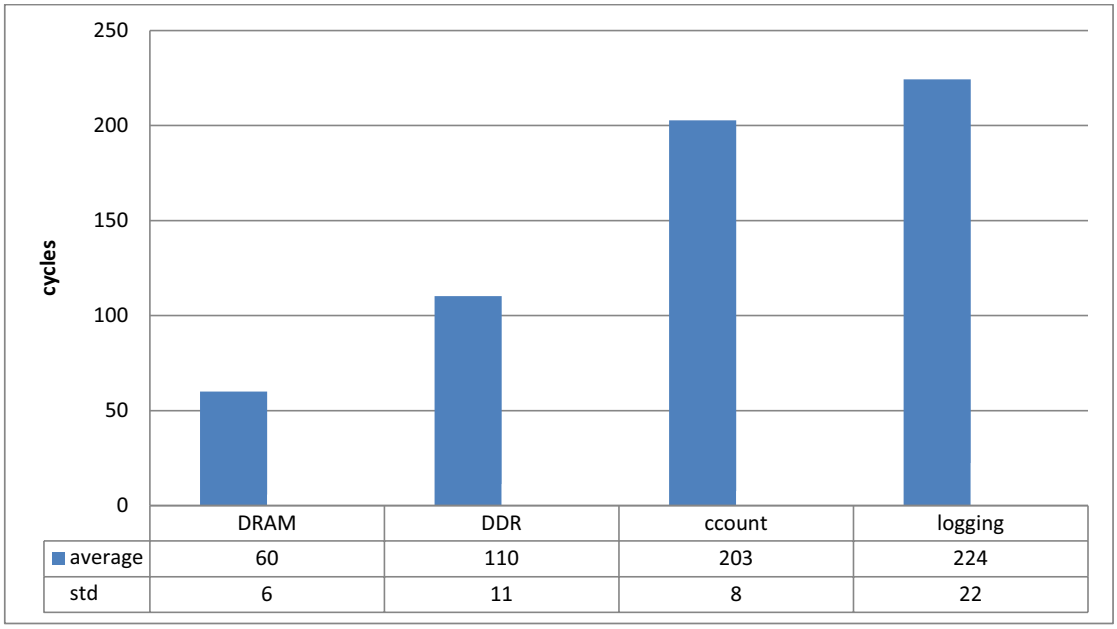
```
for (i=0; i<1000; i++){
  OS_CRITICAL_ENTER

    start = start time of the measurement;

    execution of the functions to be measured;

    stop = stop time of the measurement;

  OS_CRITICAL_EXIT

  log[i] = stop - start;
}

for (i=0; i<1000; i++){
  printf(log[i]);
}
```

Figure 19: Measurement Pseudocode

The measurements are output all at once at a later point in time using a *printf()* statement. This can be immediately after the measurements have been performed or when it is convenient to do so. The output occurs outside of the measurement and therefore does not interfere with the measurement. To eliminate interference from other tasks and ISRs, the measurement can be performed in a *OS_CRITICAL* section.

The SBIOS library [10] provides the current time with the function *sx_get_ccount()*. This function can be called at any point in the application to report the number of processor cycles executed up until that point. The number is retrieved from a register at a very low overhead. The main overhead is incurred when storing the result of *sx_get_ccount()* in the memory. There are two possible places: the value can be stored in the main memory (DDR) or in the local memory (DRAM). The data cache cannot be directly addressed. By comparing the time spent on storing a value in either memory, the most suitable location can be selected. The determination of the two overheads is performed by placing two calls to *sx_get_ccount()* immediately after one another and storing the results of each call in either the main memory or in the local memory. Unfortunately, these two function calls cannot be simply placed after each other in a loop, it is necessary to separate each set of two calls from each other due to the cache. If each of the sets were executed directly after one another, the values would remain in the data cache and be written back to the memory at a later time, rendering the measurements useless. By performing one measurement after the production of one frame the separation between the measurements is sufficient for the measurements to actually include the write back to the memory. To eliminate interference

from other tasks and ISRs the tests were performed in an *OS_CRITICAL* section. For both tests 5000 sets of calls to *sx_get_ccount()* were measured to get a large sample. The results are shown in Figure 20, displayed are the average and the standard deviation.



	DRAM	DDR	ccount	logging
average	60	110	203	224
std	6	11	8	22

Figure 20: ccount & Grasp Overhead

Figure 20 shows a clear difference. When the variables are stored in the DRAM the average cycle time is 60 cycles and when the DDR is used the average cycle time is 110 cycles. These results indicate that it makes sense to use the DRAM as much as possible, with the remark that the incurred overhead is 60 cycles. The uncertainty introduced by storing the variable in the DRAM is also smaller: 6 cycles instead of 11. Based on this uncertainty, measurements in the order of 600 cycles or higher are sufficiently large to ignore the uncertainty, as the uncertainty then amounts to less than 1%. This level of 600 cycles will be referred to as the uncertainty threshold of the *printf()* method. This uncertainty is incurred only once each measurement, because only the storing of the value of *start*, see Figure 19, is inside the measurement. The storing of *stop* is outside the measurement.

The tests performed for the validation fit into 2 categories: frame-centric tests and packet-centric tests. The frame-centric tests are primarily the high level tests, tested are the input, codec and stream sub-tasks. The packet-centric tests are the low level tests, these focus on determining the time required for the various functions executed during the stream sub-task, for

instance the time needed to transfer a packet from the DRAM to the GMAC buffer. The only exception is the test on C_{codec} and sub-components, this detailed test is also a frame-centric test. The frame-centric tests are performed on a 500 frame basis, the packet-centric tests on a 1000 packet basis to gain a large sample. By designing test runs that are partially overlapping, all necessary data can be gathered and the measurements can be easily checked for consistency. See Section 5.4.1 for more detail.

5.2.2 The Grasp Method

Grasp [36] takes a comparable approach, events in the system are stored with the time at which they occurred and output at a later time. Events include the start, preemption and completion of a task, state changes of semaphores and custom events created by the developer. The main benefit of using Grasp is that is provides a visual overview of the behavior of the tasks in the application. The automatic calculation of various statistics such as WCET is also a useful feature. The downside is that the collection of all the relevant metrics cause a significant overhead on the system as a whole, as every individual event is stored and timestamped.

Compared to the *printf()* method, Grasp is more invasive, as it requires the addition of the recorder library to the application and the addition of several function calls to both the application and the operating system. The function calls are necessary to log the time of the events initiated by either the OS or the application. The time is determined by a call to the SBIOS library function *sx_get_ccount()*, see Section 5.2.1 for more detail. The architecture of the application extended with Grasp is shown in Figure 21.

The events saved by the recorder in the main memory are output in the form of a trace via the serial port. This trace can then be loaded into the player to see the results of the measurement. The interference of Grasp on the system is twofold: there is an overhead when an event is logged by the recorder and there is a large overhead when the recorded log is written to the output. The latter occurs outside the measurement and can be ignored. The former cannot, it is included in the measurement. To determine the overhead both the *printf()* method and Grasp are used to determine the time it takes to log one event into the recorder. To eliminate interference from other tasks and ISRs the tests were performed in an *OS_CRITICAL* section. For both tests 5000 sets of calls to *log_message()* were measured. Figure 20 (on page 59) shows the results, displayed are the average and the standard deviation.

Figure 20 reveals a comparable average time of 200 cycles and 225 cycles. The difference is even smaller when the uncertainty of the *printf()* method is taken into account, as 200 cycles is well below the threshold of 600 cycles.

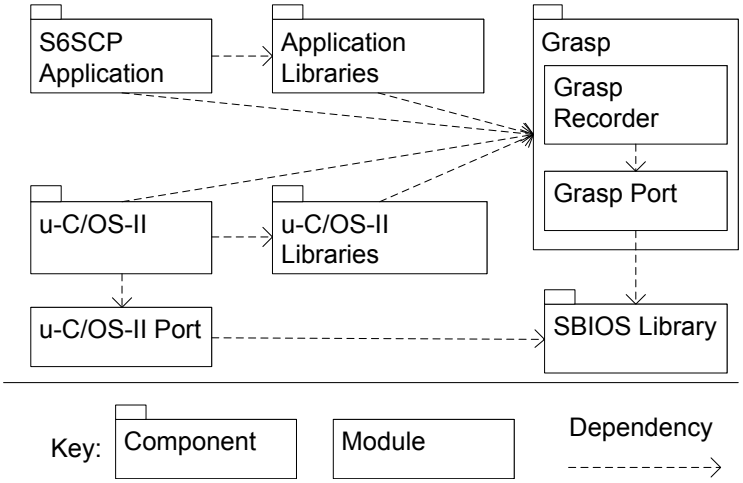


Figure 21: The Grasp Architecture

The *printf()* uncertainty of 6 cycles is added to the standard deviation of the ccount measurement (8), resulting in an uncertainty of 14 cycles.

The overhead for the use of Grasp is significantly higher compared to the 60 cycles incurred by the *printf()* method, therefore the time measurements needed for the validation are performed with the *printf()* method. This higher overhead is less of a problem for the overview of the timing behavior and interactions of the tasks and ISRs in the application and operating system, because the exact time is of less importance. The uncertainty of Grasp (22 cycles) is also larger when compared to the *printf()* method. The measurements should be in the order of 2200 cycles for the uncertainty to be ignored, as the uncertainty then constitutes less than 1% of the measurement.

5.3 Test Environment

The purpose of the test environment is to obtain reliable test results. For this, the sources of interference need to be eliminated as much as possible and when elimination is not possible the interference needs to be accounted for. There are both internal as well as external factors that can cause interference. The internal factors are due to the application itself, the external factors are caused by the interaction of other systems with the camera and the environment in which the camera is located. Both categories can be managed, but with different approaches.

The internal factors can be controlled in two ways: by turning such factors off when possible and by performing measurements in *OS_CRITICAL* sections to prevent preemptions by other tasks and ISRs. During the tests

for the validation the direct audio output (audio_out) and signalling streams have been disabled. This makes the measurements slightly optimistic. However, the focus is on the critical path of the video process, the audio and signaling streams provide a secondary service and require significantly less CPU time compared to the video task. The measurement methods themselves cause interference, for this reason when a *printf()* measurement is performed the Grasp recorder is turned off. The reverse is also true, when a Grasp measurement is performed the *printf()* calls are disabled.

The external factors are more difficult to control. The main external factor: the content of the image captured by the sensor. It is not only determined by what is in front of the camera, but also by the settings for the encoder (entered on the website, see Section 3.4), the focus of the lens and the amount of ambient light. The content of the image itself can be fixed by placing an image in front of the lens of the camera and keeping the lens focussed on that image. As long as neither the image nor the camera is moved any interference from motion is eliminated. Note that motion should not have any influence, due to the intra-frame nature of the MJPEG encoder. See Section 3.4.1.3 for the details. The encoder settings, lens focus, and light levels need to remain constant during the different measurement runs for the measurements of different runs to be comparable. The number of packets making up a single frame is a good indication of this, as a comparable number of packets indicates a comparable environment. For instance, a decrease of the ambient light level will result in a larger number of packets per frame. These factors are especially important for the frame-centric measurements, as the time to encode a frame depends on the complexity of that frame.

The website is also a source of external interference, if during a test the website is accessed the HTTP (server) task will preempt the video task and invalidate the measurement. Therefore, the website must not be accessed during tests that are not performed in a *OS_CRITICAL* section. The network itself is an important external factor as well, to prevent interference from the network the camera is connected to a switch to which only the receiving pc is connected. The available bandwidth is 100 Mbit, far greater than the maximum bandwidth the camera can require, which is in the order of 20 to 30 Mbit. Figure 22 shows the setup. Technically, the switch is superfluous. However, for the measurements given a network with fluctuating available bandwidth the switch is necessary. This allows a 2nd PC to generate additional traffic directed at a 3rd PC, which then interferes with the video stream from the camera directed to the user. See Section 5.4.3 for the details.

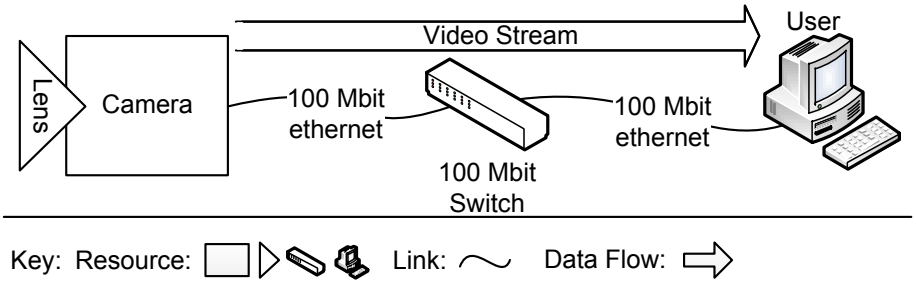


Figure 22: Measurement Setup

5.4 Validation

Chapter 4 introduced two performance models, the high level and the detailed level models. Both performance models are validated in the next two sections. The validation is based on the assumption that the result of a measurement on a task is equal to the result of the sum of all individual measurement results of all sub-tasks of that task. The difference should be negligible. The same should hold for a sub-task and all its sub-components. An example: τ_i consists of $s\tau_{i,1}$ and $s\tau_{i,2}$. The relation $C_i = C_{i,1} + C_{i,2}$ should then hold.

The validation is performed under 3 assumptions:

1. The interference caused by ISRs is negligible.
2. The interference caused by other tasks, including the context switching overhead is negligible.
3. The measurement does not significantly interfere with the behavior of the system.

Points 1 and 2 are addressed in the remainder of this section. Point 3 was addressed in Section 5.2.

Grasp is used to ascertain the remaining interferences after setting up the environment of Section 5.3. Figure 23 displays the average behavior of the video task for the production of 1 frame. Shown are the tasks and ISRs that preempt the video task and the start of the different sub-tasks of the video task. The activation of *OSTickISR* is marked with '1', '2' marks *OSCtxSw*, '3' marks the availability of a new raw frame indicated by *sx_dp_irq*. The remaining ISRs are *S_ppi_tx*, marking the completion of a packet transmission.

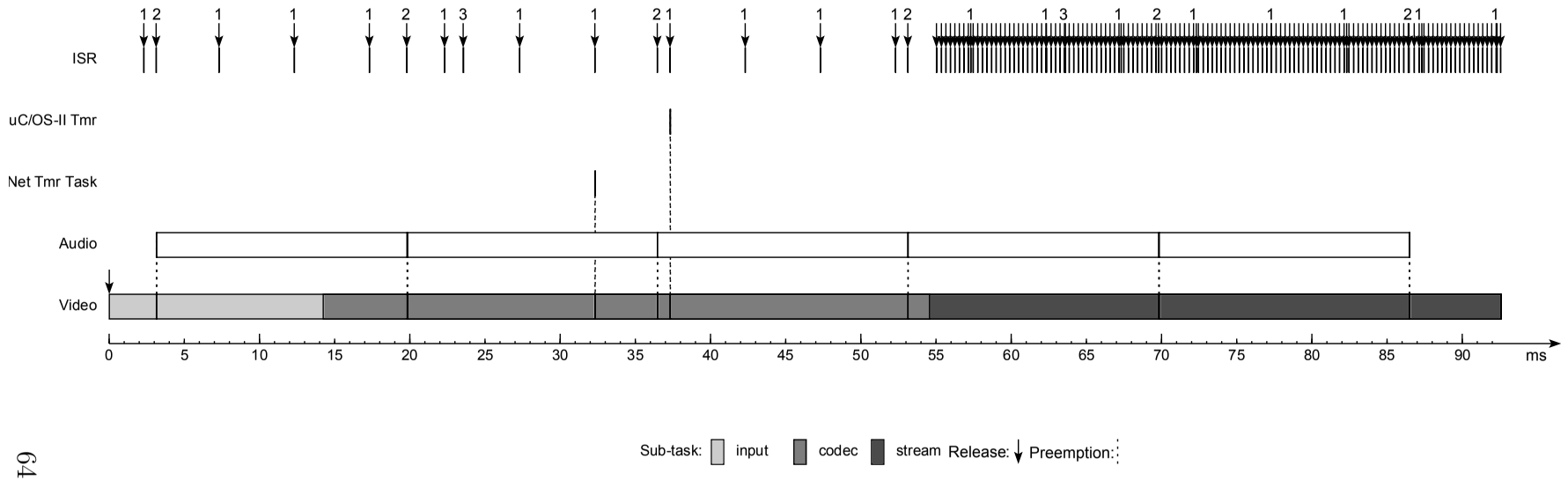


Figure 23: System Overview

Figure 23 clearly indicates that not all interferences have been eliminated. However, the delay is in the order of 0.5% of the total time. The audio task, responsible for the audio stream cannot be disabled on the website, neither can the other two tasks. To prevent these from preempting the video task during the tests, the *OSSchedLock()* and *OSSchedUnlock()* can be used. The ISRs interrupting the video task are required for the correct functioning of the video task and cannot be disabled.

The measurements for the high level model and the detailed level model were not performed in *OS_CRITICAL* sections, the input, codec and stream sub-tasks of the video task all rely on interactions with the before mentioned ISRs. Therefore, interference from other tasks and ISRs is possible. However, the interference is either negligible when long duration measurements are performed (e.g. sub-task level) or easily recognizable by a spike in the measurements for the short duration measurements. The interferences will only interfere with a small subset of the measurement due to the periods of the interferences: 5ms for *OSTickISR*, 16ms for the audio task and *OSCtxSw*, and 40ms for *sx_dp_irq*. The short duration measurements are in the order of 1000 cycles, which is 3μs, much smaller than the period of the most frequent interference, *OSTickISR*. For this reason extreme outliers have been removed from the measurement data.

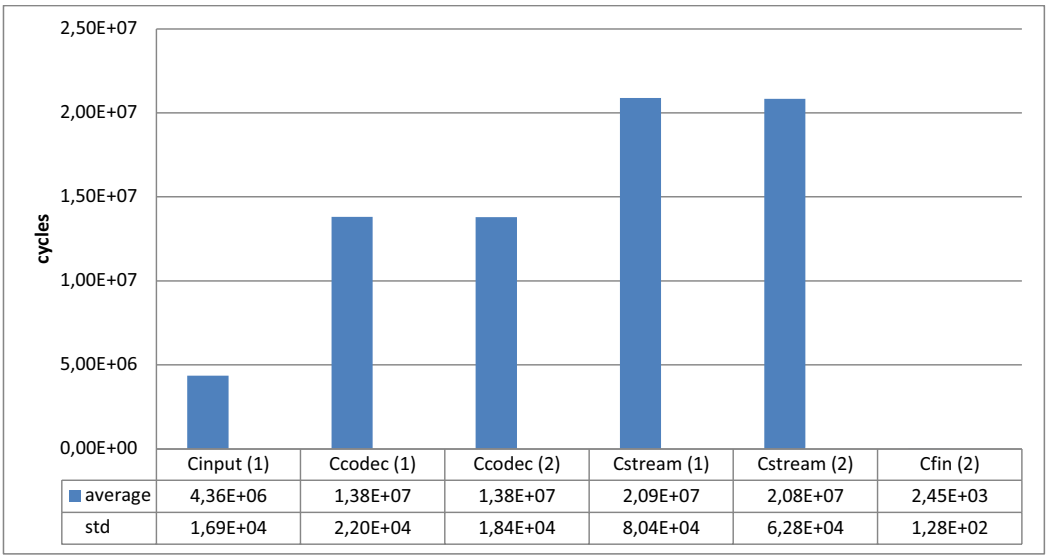
The next 3 sections validate the high level performance model, the detailed level performance model and the detailed performance model when the available bandwidth on the network is insufficient.

5.4.1 High Level Performance Model Validation

This section validates the high level performance model presented in Section 4.3. The high level performance model has a fixed resolution of 1920x1080 and the quality level as parameter q, therefore several test runs need to be performed for the different quality levels. Quality levels range from 0 to 100 and have been tested at intervals of 10. For additional accuracy for the high quality levels, measurements for $q = \{75, 85, 95\}$ have also been performed. The measurements for quality level 90 are explored in depth followed by an overview of the other quality levels. This quality level has been selected as is gives a good impression of the performance issues of the camera. To stress the importance of the ambient light level, additional measurements for $q = \{50, 75, 80, 85, 90\}$ have been performed in low light conditions.

The measurements for the high level model for quality level 90 have been performed in two runs, each consisting of 500 frames. The resolution was set at 1920x1080 and quality at level 90. Each run measured 500 frames. The first run determined the time spent on the input, codec and stream

sub-tasks of the video task. The calculation of the difference and the storing of the values was performed during the finalization sub-task. The second run determined the time spent on the codec, stream and finalization sub-tasks. The calculation and storing was performed during the input sub-task. This approach allows a consistency check on the measurements, the times measured for the codec and stream sub-tasks should be of the same order. For an additional check the number of packets per frame was also recorded for both measurements. Figure 24 displays the results of run (1) and (2), shown are the average cycle times and the standard deviation per frame. For both measurements the average number of packets per frame was 214. The recorded frame rate for both measurements was on average 8 FPS.

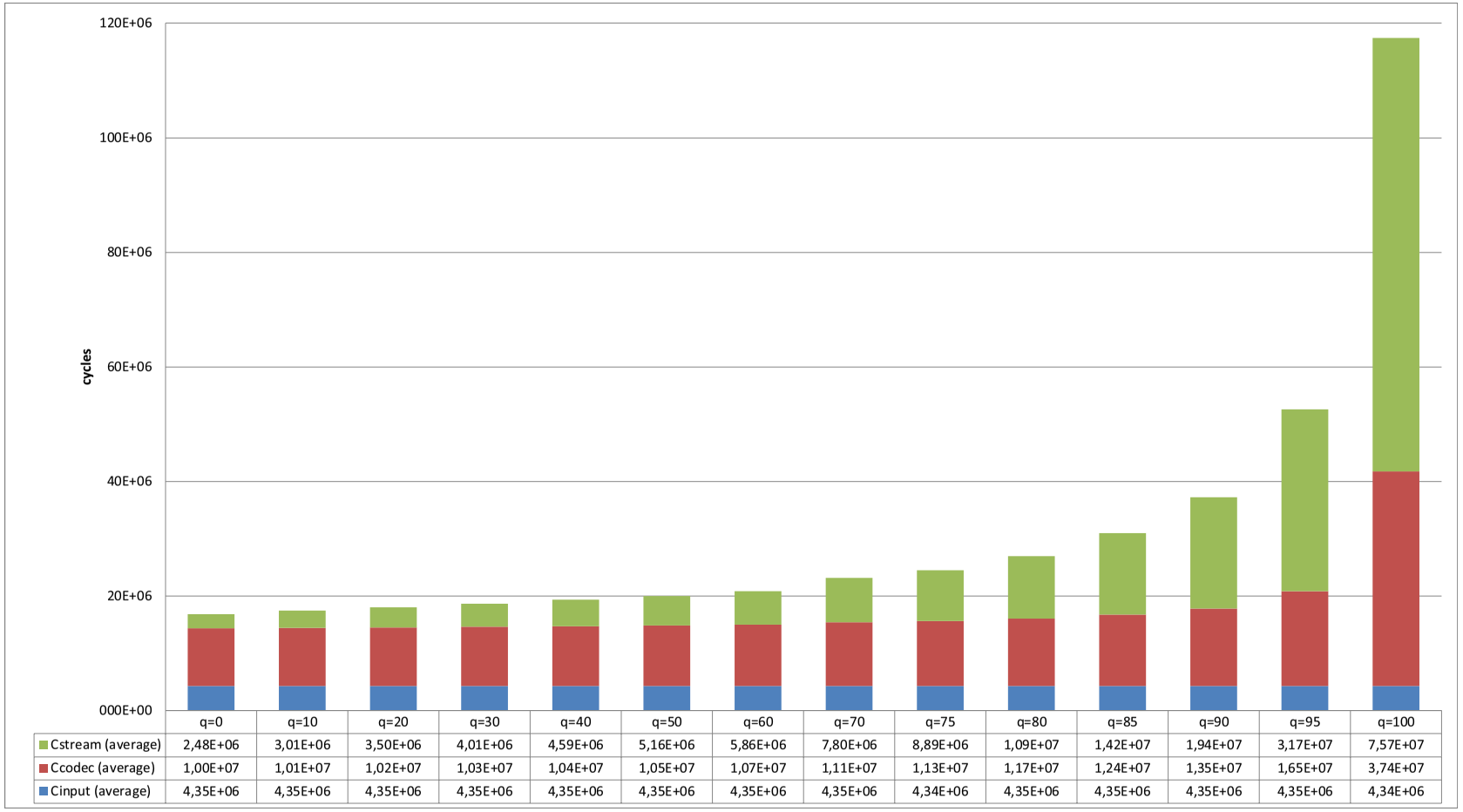


	Cinput (1)	Ccodec (1)	Ccodec (2)	Cstream (1)	Cstream (2)	Cfin (2)
average	4,36E+06	1,38E+07	1,38E+07	2,09E+07	2,08E+07	2,45E+03
std	1,69E+04	2,20E+04	1,84E+04	8,04E+04	6,28E+04	1,28E+02

Figure 24: High Level Measurements Results

The results of both runs are consistent with each other. The differences between the averages are resp. 0.2% and 0.3% for the codec and stream sub-tasks. The number of packets is the same and the frame rates are equal. The results show that the variations in the duration of each sub-task are relatively small and that the finalization sub-task is negligible compared to the input, codec and stream sub-tasks. Only C_{fin} is below the *printf()* uncertainty threshold, but it is of such small relative size that it can be ignored.

The validation of the performance model for quality level 90 is straightforward, inserting the values for the input, codec, stream and finalization sub-tasks into equation (1) and (2) results in a calculated frame rate of 8 FPS. Equal to the measured frame rate.



	q=0	q=10	q=20	q=30	q=40	q=50	q=60	q=70	q=75	q=80	q=85	q=90	q=95	q=100
Cstream (average)	2,48E+06	3,01E+06	3,50E+06	4,01E+06	4,59E+06	5,16E+06	5,86E+06	7,80E+06	8,89E+06	1,09E+07	1,42E+07	1,94E+07	3,17E+07	7,57E+07
Ccodec (average)	1,00E+07	1,01E+07	1,02E+07	1,03E+07	1,04E+07	1,05E+07	1,07E+07	1,11E+07	1,13E+07	1,17E+07	1,24E+07	1,35E+07	1,65E+07	3,74E+07
Cinput (average)	4,35E+06	4,35E+06	4,35E+06	4,35E+06	4,35E+06	4,35E+06	4,35E+06	4,35E+06	4,34E+06	4,35E+06	4,35E+06	4,35E+06	4,35E+06	4,34E+06

Figure 25: Overview Quality Level 0 to 100

	q=0	q=10	q=20	q=30	q=40
C_{input} (average)	$4{,}35\times10^{06}$	$4{,}35\times10^{06}$	$4{,}35\times10^{06}$	$4{,}35\times10^{06}$	$4{,}35\times10^{06}$
C_{codec} (average)	$1{,}00\times10^{07}$	$1{,}01\times10^{07}$	$1{,}02\times10^{07}$	$1{,}03\times10^{07}$	$1{,}04\times10^{07}$
C_{stream}(average)	$2{,}48\times10^{06}$	$3{,}01\times10^{06}$	$3{,}50\times10^{06}$	$4{,}01\times10^{06}$	$4{,}59\times10^{06}$
C_{input} (std)	$4{,}11\times10^{04}$	$1{,}36\times10^{04}$	$3{,}76\times10^{04}$	$2{,}20\times10^{04}$	$1{,}39\times10^{04}$
C_{codec} (std)	$1{,}23\times10^{04}$	$4{,}18\times10^{04}$	$1{,}16\times10^{04}$	$4{,}28\times10^{04}$	$4{,}16\times10^{04}$
C_{stream}(std)	$1{,}35\times10^{04}$	$2{,}20\times10^{04}$	$1{,}45\times10^{04}$	$1{,}75\times10^{04}$	$1{,}50\times10^{04}$
Packets (average)	25	31	36	41	47
Measured Frame Rate (average) (FPS)	18	17	17	16	16
Calculated Frame Rate (average) (FPS)	18	17	17	16	16

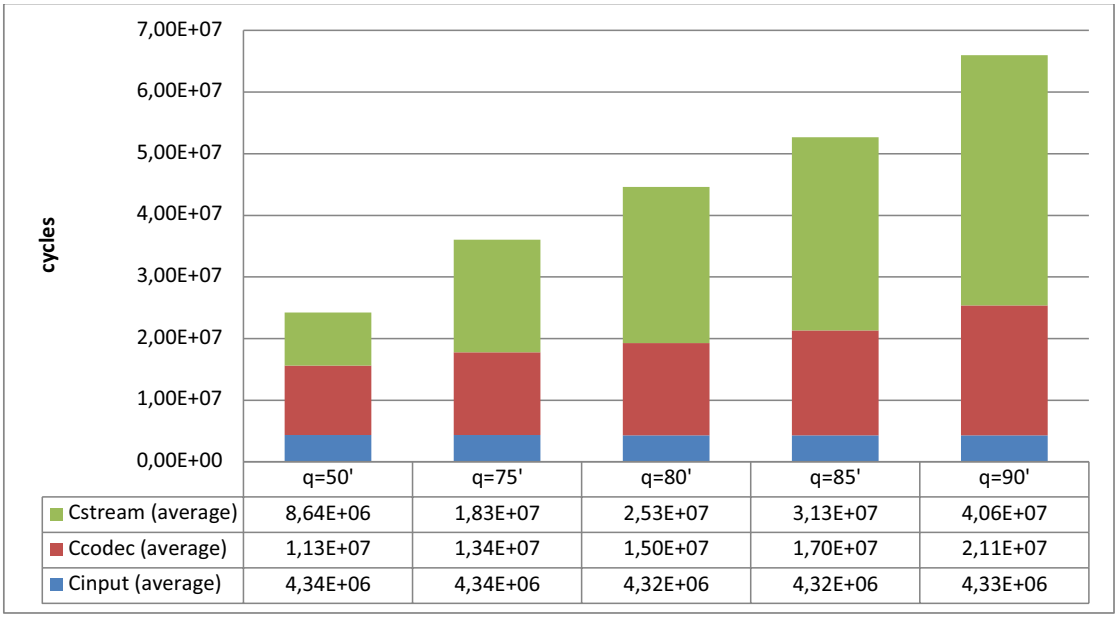
	q=50	q=60	q=70	q=75	q=80
C_{input} (average)	$4{,}35\times10^{06}$	$4{,}35\times10^{06}$	$4{,}35\times10^{06}$	$4{,}34\times10^{06}$	$4{,}35\times10^{06}$
C_{codec} (average)	$1{,}05\times10^{07}$	$1{,}07\times10^{07}$	$1{,}11\times10^{07}$	$1{,}13\times10^{07}$	$1{,}17\times10^{07}$
C_{stream}(average)	$5{,}16\times10^{06}$	$5{,}86\times10^{06}$	$7{,}80\times10^{06}$	$8{,}89\times10^{06}$	$1{,}09\times10^{07}$
C_{input} (std)	$1{,}06\times10^{04}$	$1{,}18\times10^{04}$	$1{,}25\times10^{04}$	$8{,}45\times10^{03}$	$1{,}37\times10^{04}$
C_{codec} (std)	$4{,}26\times10^{04}$	$1{,}44\times10^{04}$	$4{,}29\times10^{04}$	$4{,}32\times10^{04}$	$5{,}76\times10^{04}$
C_{stream}(std)	$3{,}51\times10^{04}$	$1{,}98\times10^{04}$	$2{,}35\times10^{04}$	$4{,}89\times10^{04}$	$3{,}04\times10^{04}$
Packets (average)	53	60	80	91	112
Measured Frame Rate (average) (FPS)	15	14	13	12	11
Calculated Frame Rate (average) (FPS)	15	14	13	12	11

	q=85	q=90	q=95	q=100
C_{input} (average)	$4{,}35\times10^{06}$	$4{,}35\times10^{06}$	$4{,}35\times10^{06}$	$4{,}34\times10^{06}$
C_{codec} (average)	$1{,}24\times10^{07}$	$1{,}35\times10^{07}$	$1{,}65\times10^{07}$	$3{,}74\times10^{07}$
C_{stream}(average)	$1{,}42\times10^{07}$	$1{,}94\times10^{07}$	$3{,}17\times10^{07}$	$7{,}57\times10^{07}$
C_{input} (std)	$2{,}04\times10^{04}$	$1{,}47\times10^{04}$	$3{,}76\times10^{04}$	$2{,}53\times10^{04}$
C_{codec} (std)	$6{,}17\times10^{04}$	$4{,}13\times10^{04}$	$9{,}68\times10^{04}$	$1{,}10\times10^{05}$
C_{stream}(std)	$9{,}01\times10^{04}$	$6{,}70\times10^{04}$	$2{,}54\times10^{05}$	$9{,}34\times10^{06}$
Packets (average)	146	199	326	772
Measured Frame Rate (average) (FPS)	10	8	6	3
Calculated Frame Rate (average) (FPS)	10	8	6	3

Table 8: Overview Quality Level 0 to 100: Average Cycles, Standard Deviation (std) Cycles, Packets & Frame Rate

The results for quality levels 0 to 100 are shown in Figure 25 (on page 67), the finalization sub-task is ignored due to its small relative duration. The measurements for each of the quality levels were performed in one run, the storing of the variables was performed in the finalization sub-task. In Table 8 (on page 68) the standard deviation, average number of packets per frame and the average frame rate and are shown.

To illustrate the impact of ambient light, a subset of the measurements is performed under reduced light conditions. The quality levels $q = \{50, 75, 80, 85, 90\}$ have been remeasured and are marked with a '. See Figure 26 and Table 9.



	q=50'	q=75'	q=80'	q=85'	q=90'
Cstream (average)	8,64E+06	1,83E+07	2,53E+07	3,13E+07	4,06E+07
Ccodec (average)	1,13E+07	1,34E+07	1,50E+07	1,70E+07	2,11E+07
Cinput (average)	4,34E+06	4,34E+06	4,32E+06	4,32E+06	4,33E+06

Figure 26: Overview Quality Level 50 to 90 in low light conditions

Comparing the results of Table 8 and 9 clearly show that the ambient light has a significant impact. For instance, quality level $q = 90$ and $q = 90'$ show that the time for encoding and sending a single video frame nearly doubles for $q = 90'$. Two observations regarding the data can be made:

1. The duration of the input sub-task is constant regardless of the quality level or the ambient light, while the codec and stream sub-tasks vary significantly.

2. The ambient light is of greater influence than the quality level on the duration of the codec and stream sub-tasks and as a result on the frame rate of the video. The standard deviation of the measurements

increases significantly with low ambient light. As a consequence, the number of packets per frame is not a good indication of the quality level of the video. However, it is a good indication of the workload for both the codec and stream sub-tasks, as a higher number of packets always translates into a higher computation time for both sub-tasks.

For the validation of the high level process model the values for C_{input}, C_{codec}, C_{stream} and the frame rate from Table 8 and 9 are entered into equations (1) and (2). The results are also shown in Table 8 and 9.

	q=50'	q=75'	q=80'	q=85'	q=90'
C_{input} (average)	$4{,}34\times10^{06}$	$4{,}34\times10^{06}$	$4{,}32\times10^{06}$	$4{,}32\times10^{06}$	$4{,}33\times10^{06}$
C_{codec} (average)	$1{,}13\times10^{07}$	$1{,}34\times10^{07}$	$1{,}50\times10^{07}$	$1{,}70\times10^{07}$	$2{,}11\times10^{07}$
C_{stream} (average)	$8{,}64\times10^{06}$	$1{,}83\times10^{07}$	$2{,}53\times10^{07}$	$3{,}13\times10^{07}$	$4{,}06\times10^{07}$
C_{input} (std)	$1{,}43\times10^{04}$	$2{,}62\times10^{04}$	$3{,}01\times10^{04}$	$1{,}91\times10^{04}$	$3{,}50\times10^{04}$
C_{codec} (std)	$1{,}61\times10^{04}$	$8{,}34\times10^{04}$	$8{,}46\times10^{04}$	$1{,}75\times10^{05}$	$2{,}76\times10^{05}$
C_{stream} (std)	$4{,}72\times10^{04}$	$3{,}56\times10^{05}$	$3{,}83\times10^{05}$	$3{,}76\times10^{05}$	$6{,}05\times10^{05}$
Packets(average)	88	187	259	321	416
Measured Frame Rate (average) (FPS)	12	8	7	6	5
Calculated Frame Rate (average) (FPS)	12	8	7	6	5

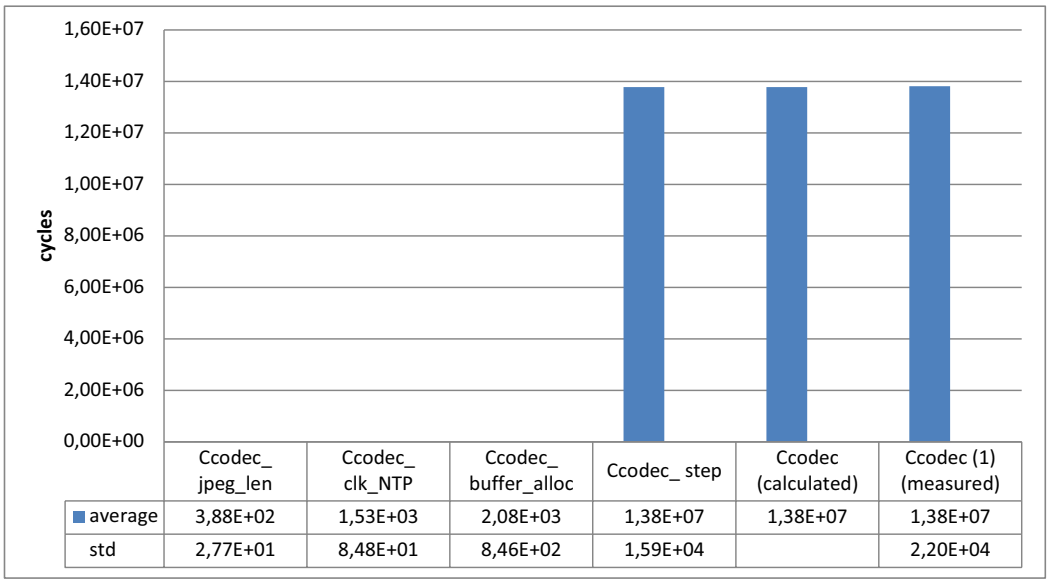
Table 9: Overview Quality Level 50 to 90 in low light conditions: Average Cycles, Standard Deviation (std) Cycles, Packets & Frame Rate

5.4.2 Detailed Level Performance Model Validation

The detailed level performance model focuses on C_{codec} and C_{stream} and is expressed by equations (3) to (6). In this section C_{codec} is validated first followed by C_{stream}. The claim that $C_{\text{stream\_packet\_build}}$, $C_{\text{stream\_packet\_send}}$ and $C_{\text{stream\_packet\_fin}}$ are independent from the quality of the video is also validated.

The measurements for C_{codec} are performed in one run, the storing of the values was performed during the finalization sub-task. To make sure the measurements are comparable with the high level measurements shown in Figure 24 (on page 66) both the number of packets per frame and the frame rate was measured. The computation times of the 4 sub-components of C_{codec} were added, the result is shown as *Ccodec (calculated)*. The high level measurement *Ccodec (1)* is repeated for comparison. Figure 27 displays the results of the measurement for resolution 1920x1080 at quality level 90. In total 500 frames were measured. Shown are the average cycle times and

the standard deviation per frame. The average number of packets per frame was 213, the average frame rate was 8 FPS.



	Ccodec_ jpeg_len	Ccodec_ clk_NTP	Ccodec_ buffer_alloc	Ccodec_ step	Ccodec (calculated)	Ccodec (1) (measured)
average	3,88E+02	1,53E+03	2,08E+03	1,38E+07	1,38E+07	1,38E+07
std	2,77E+01	8,48E+01	8,46E+02	1,59E+04		2,20E+04

Figure 27: C_{codec} Measurements Results

The results show that the core of the workload lies with C_{codec_step} and that the workload of the other sub-components of C_{codec} is negligible. Only $C_{codec\_jpeg\_len}$ is below the uncertainty threshold (600 cycles, see Section 5.2.1, but it is of such small relative size that it can be ignored. Considering the number of packets for this measurement is comparable to the number of packets for the high level model measurements both measurements should be consistent. The *codec (calculation)* and the *codec (measured)* bars show that this is indeed the case, validating C_{codec}. The difference between the calculated C_{codec} and the measured C_{codec} is 0,23%.

The measurements for C_{stream} are performed in three runs due to the switch from a frame-centric measurement to a packet-centric measurement. The first measurement is performed to determine the initialization and the finalization of the send function up to and starting from the loop that creates the packets. This initialization and finalization is incurred only once every frame. The second measurement focusses on the build phase of the send function: the creation and the copying of the headers into the packet in the various buffers. There is again a finalization, incurred once every packet. The third and final measurement determines the waiting time on the network and the time spent on copying the packet from the DDR to the DRAM and from the DRAM to the buffer on the GMAC. Again there is an initialization and

a finalization incurred once every packet.

The first measurement is frame centric, the number of packets is measured and also the frame rate. The average number of packets was 214 and the average frame rate 8 FPS. *Cstream (calculated)* is the result of the sum of $C_{\text{stream\_packet\_\{build + send + fin\}}}$ and $C_{\text{stream\_init\&fin}}$

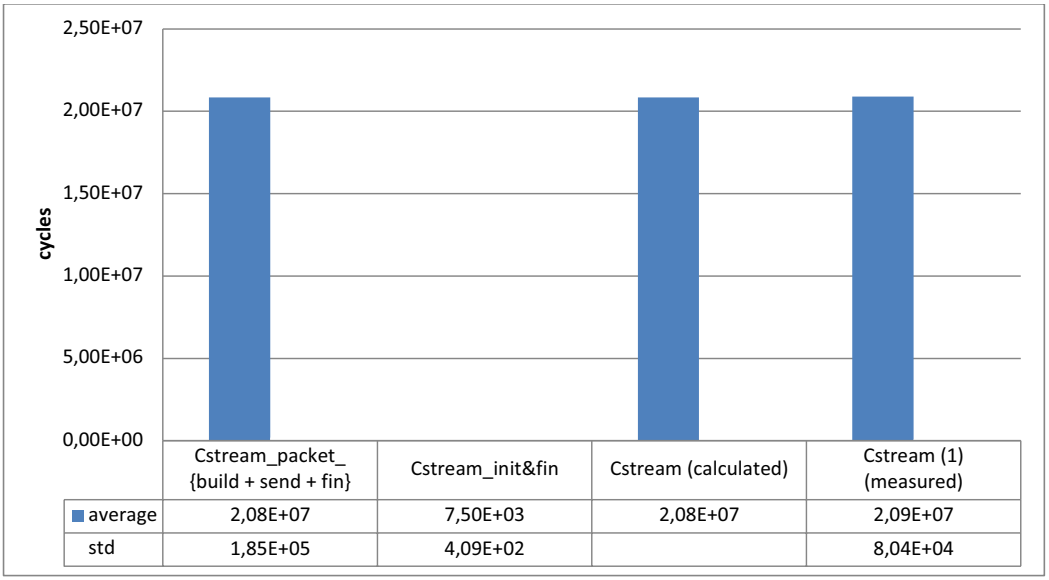
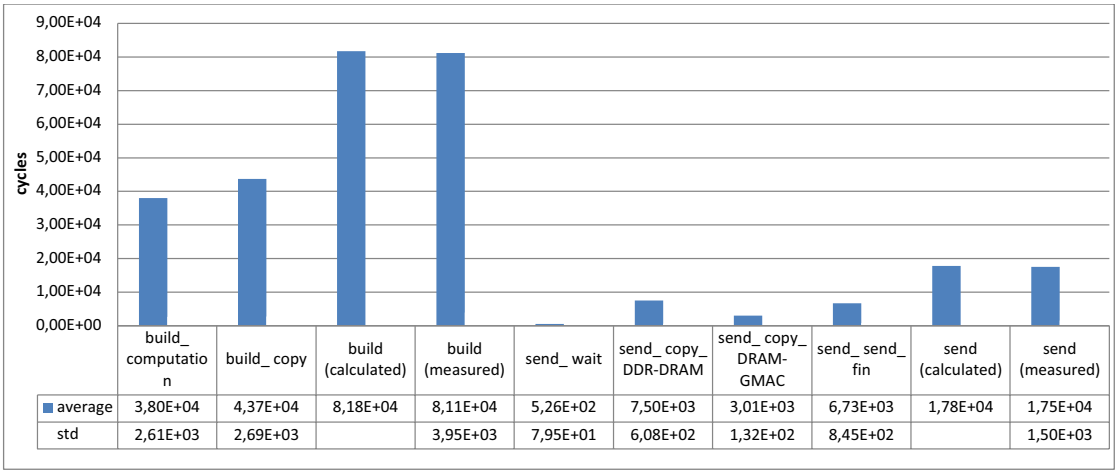


Figure 28: C_{stream} Frame-Centric Measurements Results

The results show that the initialization and finalization are small compared to the time spent on building and transmitting the actual packet. The number of packets is again comparable to the number of packets for the high level model measurements allowing another consistency check. The *stream (calculation)* and the *stream (measured)* bars display comparable times, the difference is 0,25%.

The second and third measurements are packet-centric, the number of packets per frame are not measured, neither is the frame rate. The second measurement focusses on the build phase of packet construction. It includes the time used for creating the headers, the time for copying the headers into the packet in the buffers. The third measurement determines the waiting time on the network, the transfer times of the packet from memory to memory and the finalization for each packet (*send_send_fin*). Both measurements again overlap, the second measurement also measures the total time for $C_{stream\_packet\_send}$ and the third measurement the total time for $C_{stream\_packet\_build}$ allowing a consistency check between the measurements. The results of both measurements are in Figure 29. Shown are the average cycle times and the

standard deviation. The value for *build (calculated)* is the result of the sum of *build_computation* and *build_copy*. The value for *send (calculated)* is the result of the sum of the sub-components of $C_{stream\_packet\_send}$ and *send_send_fin*.



	build_ computation	build_ copy	build (calculated)	build (measured)	send_ wait	send_ copy_ DDR-DRAM	send_ copy_ DRAM-GMAC	send_ send_ fin	send (calculated)	send (measured)
average	3,80E+04	4,37E+04	8,18E+04	8,11E+04	5,26E+02	7,50E+03	3,01E+03	6,73E+03	1,78E+04	1,75E+04
std	2,61E+03	2,69E+03		3,95E+03	7,95E+01	6,08E+02	1,32E+02	8,45E+02		1,50E+03

Figure 29: C_{stream_packet} Packet-Centric Measurements Results

The differences between the calculated and the measured values for $C_{stream\_packet\_build}$ and $C_{stream\_packet\_send}$ are respectively 0,76% and 1,29%. These differences are higher compared to the previous differences for several reasons. Both measurements are the result of a sum of individual measurements. Moreover, the measured values are smaller, with several below the uncertainty threshold. As a result, their standard deviations are relatively higher. All of these factors contribute to the higher relative uncertainty and therefore increased differences.

The detailed performance model states that $C_{stream\_packet\_build}$, $C_{stream\_packet\_send}$ and $C_{stream\_packet\_fin}$ are independent from the quality of the video. Considering that the bulk of C_{stream} is determined by $C_{stream\_packet\_\{build\ +\ send\ +\ fin\}}$, see Figure 28, the increase of C_{stream} should be linear with an increase of the number of packets. Figure 30 shows the measurements on C_{stream} for the different quality levels and the corresponding number of packets per frame. For comparison C_{codec} is added as well. For both C_{codec} and C_{stream} the trendlines are shown with the function on which they are based. The trendlines were determined with MS Excel.

Figure 30 shows that C_{stream} increases linear with an increase in number of packets, for C_{codec} this is clearly not the case, an exponential function has a better fit.

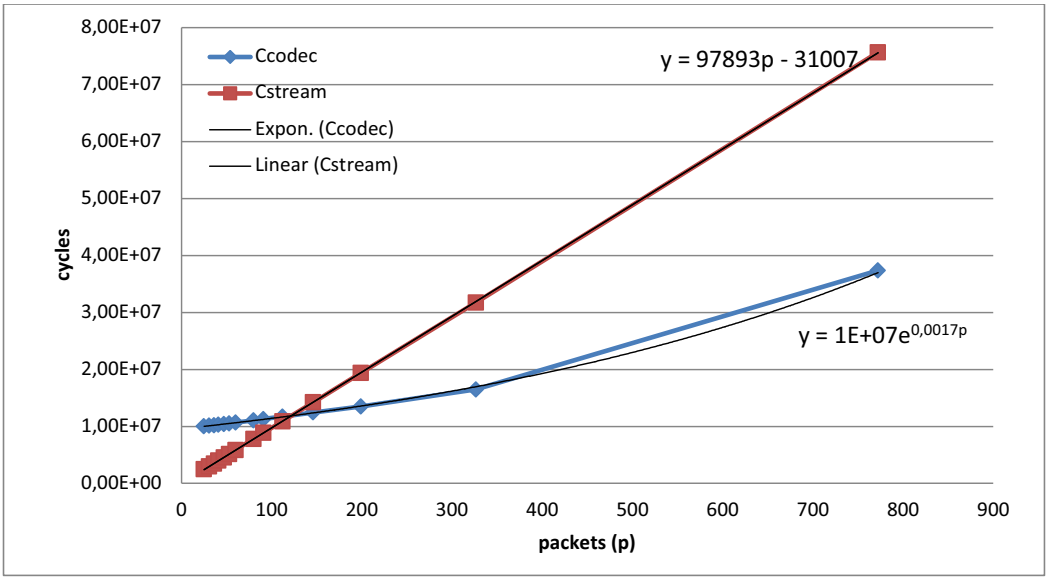


Figure 30: Computation Time & Packets

The validation of the detailed model is performed by entering the calculated values presented in this section in equations (3) to (6) repeated in equation (7). The results are shown in equation (8).

$$
\begin{aligned}
C_{\text{frame\_total}}[i,q] \equiv\ & C_{\text{input}}[i] \\
& + C_{\text{codec\_jpeg\_len}}[i] + C_{\text{codec\_clk\_NTP}}[i] + C_{\text{codec\_buffer\_alloc}}[i] + C_{\text{codec\_step}}[i,q] \\
& + \sum_{p=1}^{n[q]} (C_{\text{stream\_packet\_build\_computation}} + C_{\text{stream\_packet\_build\_copy}} \\
& \qquad + C_{\text{stream\_packet\_send\_wait}} \\
& \qquad + C_{\text{stream\_packet\_send\_copy\_DDR\_DRAM}} + C_{\text{stream\_packet\_send\_copy\_DRAM\_NIC}} \\
& \qquad + C_{\text{stream\_packet\_fin}})[i] \\
& + C_{\text{stream\_init\_\&\_fin}}[i] \\
& + C_{\text{fin}}[i]
\end{aligned}
\tag{7}
$$

$$
\begin{aligned}
C_{\text{frame\_total}}[i,q] &= 4,36 \times 10^{06} \\
&\quad + 3,88 \times 10^{02} + 1,53 \times 10^{03} + 2,08 \times 10^{03} + 1,38 \times 10^{07} \\
&\quad + \sum_{p=1}^{214} (3,80 \times 10^{04} + 4,37 \times 10^{04} \\
&\qquad + 5,26 \times 10^{02} \\
&\qquad + 7,50 \times 10^{03} + 3,01 \times 10^{03} \\
&\qquad + 6,73 \times 10^{03}) \\
&\quad + 7,50 \times 10^{03} \\
&\quad + 2,45 \times 10^{03} \\
&= 3,95 \times 10^{07}
\end{aligned}
\tag{8}
$$

The resulting frame rate is 8 FPS, equal to the measured frame rate. Comparing the calculated cycle times to the measured cycle times shows a difference of 1,21%.

5.4.3 Detailed Level Performance Model Validation with Insufficient Bandwidth

The measurements up to this point have been performed in an environment in which the available bandwidth on the network was sufficient. This is expressed by the consistent low times measured for $C_{stream\_packet\_send\_wait}$. When the available bandwidth is not sufficient the behavior of the system as a whole changes dramatically, shown by the Grasp trace in Figure 31. Shown are the tasks and ISRs that preempt the video task and the start of the different sub-tasks of the video task. The activation of *OSTickISR* is marked with '1', '2' marks *OSCtxSw*, '3' marks the availability of a new raw frame indicated by *sx_dp_irq*. The remaining ISRs are *S_ppi_tx*, marking the completion of a packet transmission. Figure 31 is accompanied by a measurement on $C_{stream\_packet\_send\_wait}$, see Figure 32.

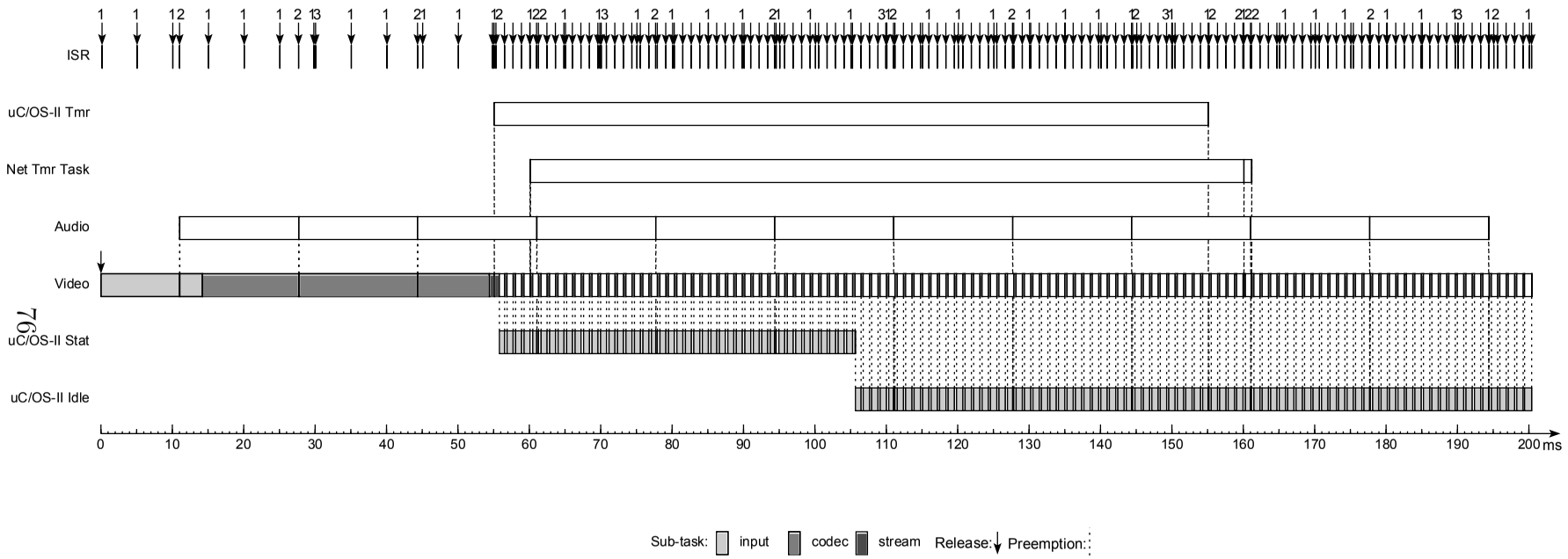


Figure 31: System overview when available bandwidth is insufficient

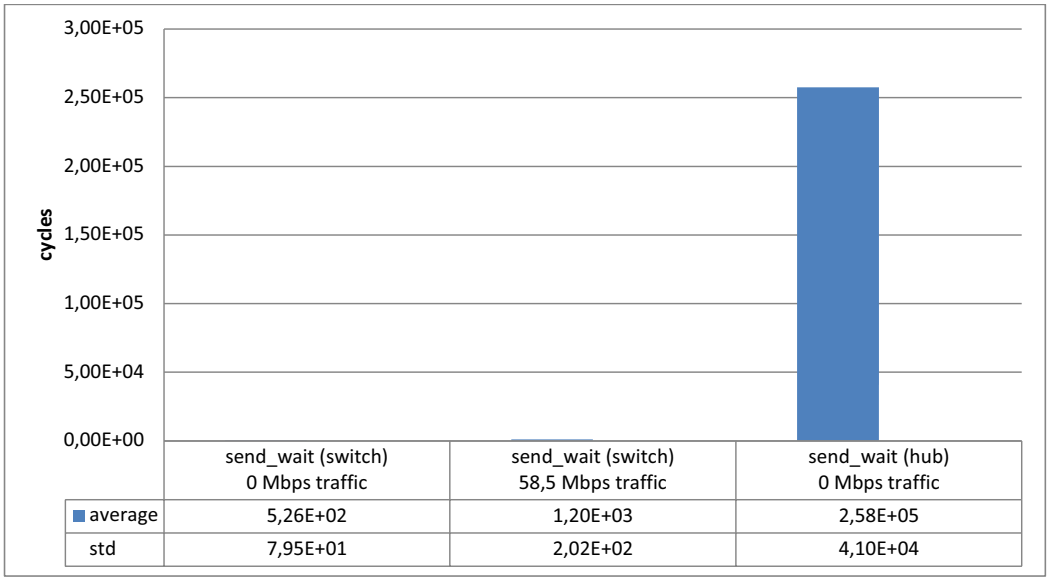


Figure 32: $C_{stream\_packet\_send\_wait}$ when available bandwidth is insufficient

To obtain the results for Figure 32 the resolution was set at 1920x1080 and the quality level at 90. The measurement from the previous section for $C_{stream\_packet\_send\_wait}$ is repeated as the 0 Mbps traffic measurement. For the 58,5 Mbps test, the 100 Mbit switch was saturated with 58,5 Mbps traffic, the maximum amount of traffic that does not instantly disrupt the video stream. The measurement setup is shown in Figure 33. The frame rate fluctuated between 6 an 8 FPS. Instead of 500 cycles, the times for $C_{stream\_packet\_send\_wait}$ are in the order of 1000 cycles. Even though the times have doubled, the behavior of Figure 31 was not achieved. At first glance it appears that the remainder of the bandwidth is sufficient for the video stream. In theory this is correct, however, in practice increasing the traffic beyond the 58,5 Mbps level causes an immediate disruption of the stream. Possibly caused by an inability of the switch of handling the large number of packets. The overloaded state of the switch is not recognized by the camera though, indicated by the low $C_{stream\_packet\_send\_wait}$ measurement results.

To achieve the behavior of Figure 31 the switch from the original measurement setup (Figure 22 on page 63) was replaced with a 10 Mbit hub. The main difference between a switch and a hub is that the switch provides a separate *collision domain* to each of its links and a hub provides a shared collision domain over all its links. As a consequence, when using a switch a link always has the maximum bandwidth available and in an overload situation packets are dropped by the switch. When using a hub every link shares the bandwidth, in an overload situation a device is unable of pushing

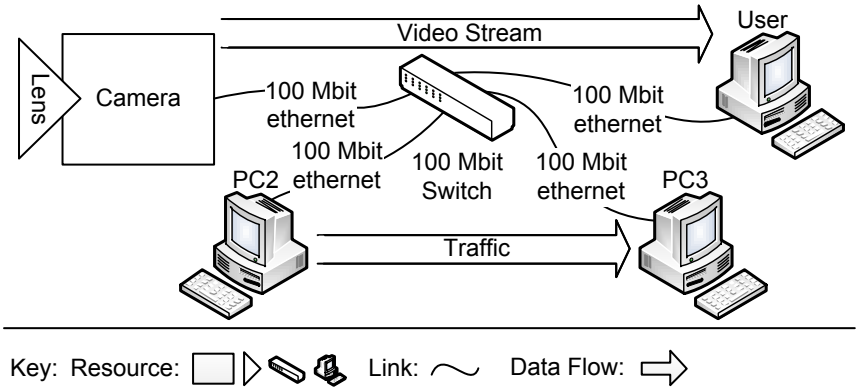


Figure 33: Insufficient Available Bandwidth Measurement Setup

a packet onto the link.

Considering that 10 Mbit is insufficient to carry a video stream at quality level 90, the bandwidth required is in the order of 20 Mbps[1], no additional traffic is needed on the hub. Instead of the waiting times of 500 cycles, the waiting times are in the order of 250000 cycles, causing the camera to idle when the network bandwidth is insufficient. The frame rate fluctuated around 3 FPS, a significant drop compared to the 8 FPS measured for the clear network case.

[1]The required bandwidth can be measured, but also calculated. The fixed size of a packet is 1500 bytes or $1500 * 8 = 12000$ bits, including all headers. The number of packets per frame is 215 for resolution 1920x1080 and quality level 90. The frame rate is 8 FPS. The required bandwidth is then $12000 * 215 * 8 = 20640000$ bps or 21 Mbit.

6 Identification of the Improvement Points and Improvement Approaches

The goal of identifying the improvement points in the performance is to serve as a starting point for achieving a higher frame rate.

The performance model from Chapter 4 can be used to calculate the necessary reduction of computation time to achieve a higher frame rate by entering a frame rate and calculating the total cycle time, see Table 10. The frame rate of 8 FPS is taken as the base case for this calculation, the frame rate of the video for resolution 1920x1080 and quality level 90 used in the validation of the performance model. The improvements necessary to achieve a higher frame rate are presented relative to this level of performance.

Frame Rate (FPS)	*frame_total* Time (cycles)	Improvement (cycles)	Relative Improvement (%)
8	$3{,}75\times10^7$		
9	$3{,}33\times10^7$	$4{,}17\times10^6$	11
10	$3{,}00\times10^7$	$7{,}50\times10^6$	20
11	$2{,}73\times10^7$	$1{,}02\times10^7$	27
12	$2{,}50\times10^7$	$1{,}25\times10^7$	33
13	$2{,}31\times10^7$	$1{,}44\times10^7$	38
14	$2{,}14\times10^7$	$1{,}61\times10^7$	43
15	$2{,}00\times10^7$	$1{,}75\times10^7$	47
16	$1{,}88\times10^7$	$1{,}88\times10^7$	50

Table 10: Frame rates and required performance improvements compared to base case of 8 FPS for resolution 1920x1080 and quality level 90

Table 10 shows that significant reductions in the cycle times are necessary for an increase in frame rate. To achieve a frame rate of 10 FPS a 20% improvement is necessary. To achieve a frame rate of 15 FPS, an improvement of nearly 50% is necessary. The reductions in cycle times can be achieved in two ways: (1) The calculation time of a sub-task or sub-component of a sub-task can be reduced directly, (2) operations may be parallelized. The codec sub-task is a good example of parallelization, it uses not only the main processor for the encoding process but also the auxiliary processor, see Section 3.4.1.3.

The next section evaluates the performance improvement hypotheses from Section 4.5 and determines the main improvement points in the performance, the following section proposes approaches to achieve performance improvements.

6.1 Performance Improvement Points

Based on the high level model, see Section 4.3 the codec and the stream sub-tasks of the video task are the main actors. Together these take up 75% to 95% of the total time for the production and transmission of one video frame for resolution 1920x1080 and quality level 0 to 100 given sufficient available bandwidth. At the lower quality levels the codec sub-task usually dominates, depending on the amount of ambient light, at the higher quality levels the stream sub-task dominates, see Figure 25 (on page 67). As a consequence, the lower quality levels benefit more from a performance improvement in encoding and the higher quality levels from an improvement of the stream sub-task. Based on this, hypotheses 1, 2, and 3, see Section 4.5, can be evaluated. Hypotheses 1 is false, the input sub-task is not dominant therefore not an improvement point. Hypotheses 2 and 3 are both conditionally true. Hypotheses 2 is true for quality levels $q < 85$, and false for the quality levels $q \geq 85$. Hypotheses 3 is true for quality levels $q \geq 85$ and false for quality levels $q < 85$. Note that the pivot point is dependent on the ambient light levels. For lower light levels the pivot point lies at a lower quality level as shown by Table 9 (on page 70).

Unfortunately, the source code of the encoder is unavailable. As a consequence, the details on the specific improvement points inside the encoder are also unavailable. For that reason the encoder is considered a 'black box' and is to be improved as a whole. Conversely, the stream sub-task is entirely open, making more detailed determinations of improvement points possible. Based on the performance model shown in Section 4.4, the computation time for the build phase of a packet ($C_{stream\_packet\_build}$) consists of a computation component ($C_{stream\_packet\_build\_computation}$) and a copying component ($C_{stream\_packet\_build\_copy}$). Comparing the values of $C_{stream\_packet\_build\_computation}$ and $C_{stream\_packet\_build\_copy}$ to the total time of $C_{stream\_packet\_build}$, see Table 29 (on page 73), show that on average 55% of the time is spent on copying the packet. As copying is an overhead, this is an important improvement point. The creation of a packet takes the remaining 45% of the time, a significant factor and therefore also candidate for improvement. Therefore hypotheses 3a and 3b are both true.

The computation time for *stream_packet_send*, the sending phase of the stream sub-task, consists of the waiting time $C_{stream\_packet\_send\_wait}$ and the time for the two copy actions $C_{stream\_packet\_send\_copy\_DDR\_DRAM}$ and $C_{stream\_packet\_send\_copy\_DRAM\_NIC}$. Sub-component *stream_packet_send_wait* is not an improvement point when the network has sufficient bandwidth available. However, it is one when insufficient bandwidth is available, see Section 5.4.3. Hypotheses 3c is therefore conditionally true.

The two copy actions *stream_packet_send_copy_DDR_DRAM* and *stream_packet_send_copy_DRAM_NIC* are performed with differing methods. The former is implemented by a memcopy, the latter by a DMA transfer. *stream_packet_send_copy_DDR_DRAM* delays the execution of the critical path, no other operation can be performed during the memcopy, and is therefore an improvement point. On the other hand, *stream_packet_send_copy_DRAM_NIC* is performed in the background by a DMA transfer. It does not delay the critical path and the computation time is 50% less when compared to $C_{stream\_packet\_send\_copy\_DDR\_DRAM}$, see Figure 29 (on page 73). The focus is therefore on the improvement of $C_{stream\_packet\_send\_copy\_DDR\_DRAM}$, making hypotheses 3d true and hypotheses 3e false.

Hypotheses 4 is false considering that the measured overhead of other tasks and ISRs is in the order of 0.5%, see Section 5.4.

An overview of the performance improvement points:

1. For the lower quality levels the codec sub-task is the main improvement point, for the higher quality levels the stream sub-task.

2. In the stream sub-task the time spent on memcopies is the main improvement point. This includes $C_{stream\_packet\_build\_copy}$ and $C_{stream\_packet\_send\_copy\_DDR\_DRAM}$. However, the time spent on $C_{stream\_packet\_build\_computation}$ is also significant.

3. When the network is congested *stream_packet_send_wait* is a serious improvement point

6.2 Performance Improvement Approaches

Several improvement methods are discussed in the following sections, starting with approaches for the encoder and followed by approaches for the stream sub-task. The approaches for dealing with a congested network conclude this section.

6.2.1 Encoder

The encoder is considered a black box as the source code is unavailable, see Section 3.4.1.3. The only method of improving the encoder therefore is to replace it entirely by another encoder. However, this changes the output of the camera, as a video stream encoded with MJPEG is expected and a video stream encoded with another encoder is provided.

The encoder used by the camera for the encoding sub-task is based on MJPEG, see Section 3.4.1.3. In [33] an overview is given of several encoders,

including MJPEG, MPEG and H.264. For more detailed information on MPEG see [31] and [53] for H.264. The comparison of the compression ratios and the computational intensity shows the tradeoff that needs to be considered when selecting an encoder. MJPEG does not provide a high compression when compared to H.264 but it does have a significantly lower computational intensity. There is an additional tradeoff specific to the camera, a lower compression rate will result in a larger frame size, as a consequence the time that needs to be spent on packetizing and sending the frame will also be larger. The optimal encoder should be selected based on the combination of these factors (CPU intensity and compression ratio).

Even though the H.264 standard is relatively new, there already exist implementations in the surveillance domain making use of this standard see [51]. Unfortunately, the hardware used in [51] is quite different from the camera, making comparisons impossible. The main drawback of H.264 is the computational intensity, in [48] optimizations are presented that reduce the CPU load. These optimizations include both structural changes as well as functional changes, achieving a speedup in the order of 2x when compared to the presented benchmark application.

Based on [33] the alternative encoders are more CPU intensive and therefore decrease the performance of the camera. However, these higher computation times are offset by increased compression ratios and a resulting lower number of packets. Based on the CPU use and encoding ratio, rough estimates can be given for the performance on the camera. The calculation below is based on the assumption that the CPU usage times are directly related to the CPU cycles spent on the codec sub-task and the encoding ratio is directly related to the number of packets.

Encoder	CPU Usage (%)	Encoding Ratio	C_{codec} (cycles)	Number of packets	C_{stream} (cycles)	C_{frame_total} (cycles)	Frame Rate (FPS)
None	0	1 to 1	0	3584	$3{,}50\times10^8$	$3{,}54\times10^7$	0,85
MJPEG	22	10 to 1	$1{,}80\times10^7$	358	$3{,}50\times10^7$	$5{,}73\times10^7$	5,23
MPEG-4	50	30 to 1	$4{,}09\times10^7$	119	$1{,}17\times10^7$	$5{,}69\times10^7$	5,27
MPEG-2	85	30 to 1	$6{,}95\times10^7$	119	$1{,}17\times10^7$	$8{,}55\times10^7$	3,51
H.264[2]	70	60 to 1	$5{,}72\times10^7$	60	$5{,}84\times10^7$	$6{,}74\times10^7$	4,45

Table 11: Encoders Overview

Table 11 presents the results. The base case for the calculation is a

[2]The CPU usage for H.264 is based on a lower resolution compared to the other encoders and is therefore optimistic compared to the others.

situation in which no encoding is performed and a raw frame (i.e. a non-encoded frame) is sent. The first step is the determination of the number of packets of a single raw frame. The size of a raw frame is 5MB, see Section 3.4.1.1. Given a maximum payload of 1400 bytes per packet, see Section 3.4.1.4, the resulting number of packets is 3584. With the function derived in Figure 30 the number of cycles for C_{stream} can be calculated. The total time, C_{frame_total}, can be calculated with equation (1). Finally, based on C_{frame_total} the frame rate can be calculated with equation (2).

Based on [33], the MJPEG encoder has a encoding ratio of 10 to 1. The number of packets for an encoded video frame is therefore 3584. By using the function derived in Figure 30 (on page 74) the calculation time for C_{codec} can be determined. The determination of C_{frame_total} is analogous to the calculation of C_{frame_total} for the base case. The calculation for the remaining cases (MPEG-2, MPEG-4 and H.264) is similar to the calculation for MJPEG, with the exception of the calculation for C_{codec}. Instead of determining C_{codec} based on the number of packets, C_{codec} is determined based on the CPU usage. C_{codec} for MJPEG is taken as 22% CPU usage, as determined by [33], and C_{codec} for the remaining cases is calculated accordingly.

Even when considering the increased compression most of the results remain unsatisfactory, exchanging the current MJPEG codec for another of the listed encoders is usually detrimental to the performance. However, the encoding ratios presented in [33] are *typical* encoding ratios for still images. The numbers are highly dependent on the actual image that is being encoded, as more complex images will have a reduced encoding ratio. The amount of motion is also a big impactor on the encoding ratio as well as the CPU usage. With the exception of MJPEG, MPEG-2, MPEG-4 and H.264 are inter-frame compression schemes and therefore dependent on motion. In situations in which motion is absent the compression ratios of MPEG-2, MPEG-4 and H.264 can be significantly higher, in the order of 1000 to 1 depending on the content of the image.

6.2.2 Stream

The improvements for the stream sub-task can be categorized as either improvements to the copying behavior or improvements to the computation behavior. First, the improvements for the copying behavior are discussed, followed by improvements to the computation behavior. For each individual improvement the impact on the frame rate of the video is calculated using the equations presented in Chapter 4.

Two of the 3 improvements to the copying are relatively straightforward, these are discussed first.

1. Currently, the packet is copied from the encoded video buffer to the send buffer and from the send buffer to the network buffer during the creation of the headers. Considering these buffers are located in the same memory (DDR), either the send buffer or the network buffer is superfluous; adding a header to a packet in one buffer is comparable to adding the same header to the same packet in another buffer. Therefore, one of the copy actions can be avoided.

2. The packet is copied from the network buffer in the main memory (DDR) to the tx buffer in the local memory (DRAM). This transfer is performed with a memcopy while it is also possible to use a DMA transfer. This allows the transfer to be completed in less time and in the background, freeing the CPU to do additional work.

The first improvement requires either a substantial rewrite of the RTP library, as it needs to be adapted to use the buffer implemented in the μC/OS-II network component, or it would require the μC/OS-II network component to be adapted to use the buffer implemented in the RTP library. The latter adaptation requires more effort than the former, based on the relative sizes of the RTP library and the μC/OS-II network component. If the copy from encoded video buffer to the send buffer is eliminated, $1{,}12\times10^4$ cycles per packet can be avoided. Compared to the total copying time, $C_{stream\_packet\_build\_copy}$, this is 25% of the total copy time. The transfer from send buffer to the network buffer is of the same order. Using either method, based on 214 packets per frame and equations (3) and (5), C_{stream} is reduced from $2{,}09\times10^7$ cycles to $1{,}85\times10^7$ cycles. Resulting in an increase of 0,50 FPS.

Exchanging the memcopy with a DMA transfer, as proposed in improvement 2, is relatively trivial to implement. The copying time is reduced from $7{,}50\times10^3$ cycles to $3{,}00\times10^3$ cycles, comparable to the DMA transfer from DRAM to GMAC. A reduction of $4{,}50\times10^3$ cycles per packet, results in an increase of 0,19 FPS, given 214 packets per frame and equations (3) and (6).

Based on [25] the total number of copy actions can be reduced, at the cost of a complete redesign of the network code and GMAC driver, though the GMAC driver can be left unchanged at the cost of a small performance loss. In the best case the entire network stack can be redesigned to eliminate all memcopies during *stream_packet_build*, resulting in the elimination of $C_{stream\_packet\_build\_copy}$. However, the copy actions during *stream_packet_send* remain necessary, but are relatively small. The complexity of *stream_packet_send_copy_DDR_DRAM* will increase as the packet is scattered throughout the main memory instead of being in one place. Likely, this will result in a higher $C_{stream\_packet\_send\_copy\_DDR\_DRAM}$. Based on Figure 29, the reduction is

$4{,}37 \times 10^4$ cycles per packet. Given 214 packets per frame and equations (3) and (5), the result is an improvement of 2,36 FPS. If not all memcopies can be removed, even the elimination of a few will result in an improvement, as shown by the calculation on the removal of the memcopy to the send buffer.

The computation time for *stream_packet_build_computation* consists of many small individual operations, however one operation stands out: the calculation of the checksum of the UDP packet. This operation takes up, on average, 80% of the total computation time of $C_{stream\_packet\_build\_computation}$. As the checksum is an optional component of a UDP packet, the removal of this calculation would reduce the cycles per packet by $3{,}04 \times 10^4$ cycles. Given 214 packets per frame and equations (3) and (5), the improvement results in an increase of 1,5 FPS.

If the checksum cannot be removed [44] proposes several optimizations, achieving a 30% performance improvement for the UDP protocol on two different platforms. These optimizations include the more intensive use of caches, redesign of general-purpose code to protocol specific code and the reduction of memory accesses. Unfortunately, this 30% cannot be directly translated to the $C_{stream\_packet\_build}$, as the differences in implementation are unknown.

6.2.3 Insufficient Bandwidth

In [35] a method is proposed to cope with a situation in which the available bandwidth fluctuates. The application is changed to allow packets to be sent when the network is free, possibly preempting the encoder in the process. When the network is not available, no attempts are made to send packets and the encoder is allowed to do additional work until the network becomes available. Unfortunately, the proposed solution cannot be implemented as-is, as it contains a number of incorrect assumptions:

1. *The scheduler is scheduled with Fixed Priority Non-preemptive Scheduling (FPNS).* This is not the case on the camera. The scheduling used by μC/OS-II is Fixed Priority Preemptive Scheduling (FPPS).

2. *There are two tasks in the system: a video and a network task.* In the application this is one task: the video task. It performs the operations of both the video and network task from [35].

3. *The network task busy waits for the network to accept a packet when the required bandwidth is unavailable.* Assuming the situation of Figure 33 (on page 78), this is only the case when the camera is connected to a hub instead of a switch, see Section 5.4.1. If a switch is used, the

current standard, the network task does not busy wait. It continues sending packets. In this case, the problem of congestion on the network is relegated to the switch. It will start dropping packets in case of an overload situation.

4. *The video task is the dominating factor in the system.* This is usually true for the lower quality levels, depending on the amount of ambient light, see Section 5.4.1. For the higher quality levels the network task is the dominating factor.

The first two points can be addressed quickly. Exchanging FPPS for Fixed Priority Scheduling with Deferred Preemption (FPDS) will, in the worst case, result in the same performance as with FPPS by allowing preemptions at all times during the runtime of the task [21]. If the preemption points are selected carefully, a performance gain can be achieved by preventing context switching overheads from occurring.

Introducing a network task to perform the stream sub-task of the video task is relatively simple. It requires the send function to be called from the new network task instead of the video task. In addition to the overhead of having an extra task in the application, a small extra overhead is introduced: the parameters required for the send function call need to be communicated from the video task to the network task. Fortunately, μC/OS-II has standard methods for the communication in the form of mailboxes and queues, see Section 3.3

The last two points are more difficult to deal with. Point 3 assumes that the check whether the network is busy is for free; when the network is busy the task simply blocks. When a switch is used this is not the case, therefore an additional check on the network needs to be performed to determine the available bandwidth. This additional check is an extra overhead and will reduce the performance.

Point 4 is the most critical one, for the case in which the network task (the stream sub-task) is the dominating factor, the proposed solution will not improve the performance much. In this case, the video task may continue while the network task is blocked, but because the network task takes substantially longer, the video task will fill the encoded video buffer quickly and then block until the network task has finished sending a frame. The resulting behavior is equal to the original behavior of the video task and as a consequence: the original performance.

Based on point 4, the method proposed in [35] is unsuitable for the optimization of the camera for the high resolution and high quality levels. For the lower resolutions and lower quality levels, when the encoder dominates

the performance of the system, the method could improve the performance. However, the implementation is not straightforward. It requires an extension of the operating system with FPDS and servers, see [27], to function.

7 Conclusion

In this thesis, the performance of a surveillance camera is investigated with respect to the frame rate of the video stream produced by the camera. A total of 5 models are presented: the system, resource, operating system, application and performance models. Each model focusses on a different aspect of the camera. The first model is the system model. This model identifies the core concepts on a high level. Concepts such as tasks and their characteristics and the resources available for these tasks such as a processor, several memories and a network and their characteristics are addressed. The mapping between the tasks and the resources is also investigated, in the form of a task to processor mapping (scheduling), task to memory mapping (data to memory mapping) and a task to network mapping (transmission).

Models 2, 3 and 4 express the core concepts on the level of the camera. Model 2, the resource model, presents the resources of the camera and their characteristics. Model 3, the operating system (μC/OS-II) model, focusses on the characteristics of the tasks in the camera system. The tasks started by the OS are mentioned and the scheduling of all tasks is explored. Model 4, the application model, presents the tasks started by the application and concentrates on the video task. The task that performs the encoding, packetizing and sending of the video.

Model 5, the performance model, relates the time spent on the critical path to the resulting frame rate of the video. The performance model is validated by performing measurements on the camera and comparing the results to the results of the performance model. The validation consists of two sets of measurements: In the first set the resolution of the video is set at the maximum resolution, the available bandwidth on the network is sufficient and the quality level is varied. In the second set the resolution and the quality level are fixed and the available bandwidth is varied. In the validation, tasks that preempt the video task and interferences such as interrupts are eliminated when possible or accounted for.

Based on the performance model and the measurements several opportunities for performance improvement are investigated. The comparison of the MJPEG, MPEG-2, MPEG-4 and H.264 encoders revealed that the MJPEG encoder provides the highest frame rate given a resolution of 1920x1080, a quality level of 90 and a network with sufficient available bandwidth. The methods for improving the packetizing process include the alteration of data copying behavior and the elimination of several data copy actions. With these methods the frame rate can be improved in direct relation to the number of packets per frame. The higher the number of packets the higher the frame rate improvement. Given an average of 214 packets per frame and

a network with sufficient available bandwidth, a frame rate increase of 2,36 FPS on an original frame rate of 8 FPS is achievable.

The method for improving the performance of the camera when the available bandwidth fluctuates described in [35] is investigated as well. The method is shown to contain a number of incorrect assumptions that make it inapplicable to the camera investigated in this thesis.

8 Future Work

This section contains some avenues for future work. Topics include implementation, analysis and optimization.

Section 6.2 proposed several methods for improving the performance of the camera. Especially the improvements proposed for the stream sub-task, some of which show significant increases in performance and are worth the effort of implementation. The proposals for the encoder sub-task and for coping with insufficient available bandwidth require additional analysis before commencing with the implementation.

The analysis on the encoder has two categories: (1) the determination of performance improvement points inside the encoder and (2) the comparison of the encoder to alternative encoders. Clarity concerning the memory transfer behavior of the encoder is necessary to determine whether this is a factor in the performance, the calculation itself may also be a factor. The encoder comparison is based on the comparison of computational intensities from [33]. These were measured on a different platform and therefore may not hold for the stretch platform. Also the comparison is relative to the stretch implementation of the MJPEG codec, it is optimized for the stretch platform as it makes use of both the auxiliary processor and the ISEF. It is possible that the other encoders are more or less suited to the stretch platform, resulting in a breakdown of the relation of computational intensities proposed in [33]. Therefore, to achieve a more reliable comparison a more detailed model of the encoders is necessary.

The analysis of handling insufficient bandwidth conditions indicated that currently the camera does not detect this event and continues sending the video regardless of the effectiveness. Therefore a method is needed to detect whether the network is available or not. The requirements on this method are a low computational intensity and the ability to detect rapid changes in the available bandwidth. The computational intensity should be low, as it is an investment of CPU time. The investment must not be larger than the original CPU time loss. The detection must be quick enough to detect the instances of available bandwidth when they occur, if the detection is too

slow, the instance may have passed before the camera can respond. In [41] the network availability detection is approached from another angle. Instead of detecting the available bandwidth at the camera, the network is managed by a central PC. Thereby removing the need for network detection by the camera, but introducing a management overhead.

The improvements proposed in Section 6.2 are improvements that are not specific to the stretch platform. Performance can be gained by optimizing the software for this platform. For instance, μC/OS-II is ported to the stretch architecture, it is not optimized for the platform. Making more intensive use of the local memories instead of the main memory could improve the performance of the camera as a whole. Also, the special features of the stretch platform, the auxiliary processor S6AUX and the ISEF, are only used by the encoder. The other sub-tasks of the video task, input and stream, may also benefit.

Acknowledgements

I would like to thank Rick Koeleman from VDG Security for answering all the questions on the camera and Nick de Koning from Prodrive for the help in getting the camera operational.

References

[1] Apple MJPEG standard. http://developer.apple.com/library/mac/#documentation/QuickTime/QTFF/QTFFPreface/qtffPreface.html.

[2] CANTATA Project Homepage. http://www.win.tue.nl/san/projects/cantata/ and http://www.hitech-projects.com/euprojects/cantata/.

[3] FRAPS Homepage. http://www.fraps.com.

[4] Live555 media server. http://www.live555.com/mediaServer/.

[5] Micrium Homepage. http://www.micrium.com.

[6] Microsoft MJPEG standard. http://www.fileformat.info/format/bmp/spec/b7c72ebab8064da48ae5ed0c053c67a4/view.htm.

[7] MontaVista Embeded Linux. http://www.mvista.com/.

[8] MT9P031: 1/2.5-Inch 5Mp Digital Image Sensor Datasheet. http://www.aptina.com.

[9] muC/Probe Homepage. http://micrium.com/page/products/tools/probe.

[10] Stretch BIOS (html). Included with Stretch IDE.

[11] Stretch IDE manual (html). Included with Stretch IDE.

[12] Texas instrument, tmdsevm6446: Dm6446 digital video evaluation module. http://focus.ti.com/docs/toolsw/folders/print/tmdsevm6446.html.

[13] *VDC-5000HD User manual*.

[14] VDG Homepage. http://www.vdg-security.com/.

[15] VLC Player Homepage. http://www.videolan.org.

[16] Real Time Streaming Protocol (RTSP), 1998. http://tools.ietf.org/html/rfc2326.

[17] RTP Payload Format for JPEG-compressed Video, 1998. http://tools.ietf.org/html/rfc2435.

[18] SIP: Session Initiation Protocol, 1999. `http://tools.ietf.org/html/rfc2543`.

[19] RTP: A Transport Protocol for Real-Time Applications, 2003. `http://tools.ietf.org/html/rfc3550`.

[20] IEEE standard for reduced-pin and enhanced-functionality test access port and boundary-scan architecture. *IEEE Std 1149.7-2009*, pages c1 –985, 10 2010.

[21] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 269 –279, july 2007.

[22] Alan Burns. Preemptive priority-based scheduling: An appropriate engineering approach. In *Advances in Real-Time Systems, chapter 10*, pages 225–248. Prentice Hall, 1994.

[23] Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2nd edition edition, 2004.

[24] L. Cherkasova and L. Staley. Building a performance model of streaming media applications in utility data center environment. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 52 – 59, may 2003.

[25] M.L. Chiang and Y.C. Li. Lyranet: A zero-copy TCP/IP protocol stack for embedded systems. *Real-Time Systems*, 34(1):5–18, 2006.

[26] M.K. Chung, S. Na, and C.M. Kyung. System-level performance analysis of embedded system using behavioral C/C++ model. In *VLSI Design, Automation and Test, 2005.(VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on*, pages 188–191. IEEE, 2005.

[27] W. Cools. Extending μCOS-II with FPDS and reservations, master thesis, eindhoven university of technology. Master's thesis, 2010.

[28] N. Cranley, P. Perry, and L. Murphy. User perception of adapting video quality. *International journal of human-computer studies*, 64(8):637–647, 2006.

[29] J. Engblom, A. Ermedahl, M. Sjdin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):437–455, 2003.

[30] D.D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Verlag, 2009.

[31] D.L. Gall. MPEG: A video compression standard for multimedia applications. *Commun. ACM*, 34(4):46–58, 1991.

[32] G. Ghinea and J.P. Thomas. Quality of perception: user quality of service in multimedia presentations. *Multimedia, IEEE Transactions on*, 7(4):786–789, 2005.

[33] J. Golston. Comparing media codecs for video content. In *Proc. Embedded System Conf*, 2004.

[34] S.R. Gulliver and G. Ghinea. Changing frame rate, changing satisfaction? [multimedia quality of perception]. In *Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on*, volume 1, pages 177 – 180 Vol.1, june 2004.

[35] M. Holenderski, R.J. Bril, and J.J. Lukkien. Using fixed priority scheduling with deferred preemption to exploit fluctuating network bandwidth. In *Work in Progress session of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[36] M. Holenderski, M.M.H.P. van den Heuvel, R.J. Bril, and J.J. Lukkien. Grasp: Tracing, visualizing and measuring the behavior of real-time systems. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, page 37, 2010.

[37] J. Honovich. Security manager's guide to video surveillance. *V3. IPVideoMarket. info.[online, ebook] Available: http://ipvideomarket. info/book*, 2009.

[38] Stretch inc. S6000 datasheet (html), 2009. included with Stretch IDE.

[39] Stretch inc. S6000 family architecture overview, 2009. `http://www.stretchinc.com/_files/s6ArchitectureOverview.pdf`.

[40] Stretch inc. S6000 family datasheet, 2009.

[41] H. Karatoy. A real time networked camera system, 2011.

[42] Jean J. Labrosse. *Micro C/OS II : the real-time kernel*. Newnes, 2002.

[43] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[44] C. Partridge and S. Pink. A faster udp. *IEEE/ACM Transactions on Networking (TON)*, 1(4):429–440, 1993.

[45] Fariza Dion Prasetyo. Vpa-pm library reference manual, 2009. Confidential.

[46] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE proceedings series*, pages 150–164. Society of Photo-Optical Instrumentation Engineers, 1997.

[47] M.H. Sedky, M. Moniri, and C.C. Chibelushi. Classification of smart video surveillance systems for commercial applications. In *Advanced Video and Signal Based Surveillance, 2005. AVSS 2005. IEEE Conference on*, pages 638 – 643, sept. 2005.

[48] M. Shafique, L. Bauer, and J. Henkel. Optimizing the H. 264/AVC video encoder application structure for reconfigurable and application-specific platforms. *Journal of Signal Processing Systems*, 60(2):183–210, 2010.

[49] L. Thiele and E. Wandeler. Performance analysis of distributed embedded systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 10–10. ACM, 2007.

[50] N.J.C. Verhagen. Manual on (re)programming the VDC-5000HD camera, 2011.

[51] N. Vun and M. Ansary. Implementation of an embedded H.264 live video streaming system. In *Consumer Electronics (ISCE), 2010 IEEE 14th International Symposium on*, pages 1 –4, june 2010.

[52] P.E. Walters. Cctv systems thinking-systems practice. In *Security and Detection, 1995., European Convention on*, pages 64 –69, may 1995.

[53] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h. 264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003.

[54] W. Wolf, B. Ozer, and T. Lv. Smart cameras as embedded systems. *Computer*, 35(9):48–53, 2002.

[55] Dapeng Wu, Y.T. Hou, Wenwu Zhu, Ya-Qin Zhang, and J.M. Peha. Streaming video over the internet: approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3):282 –300, mar 2001.

[56] C.C. Wüst, L. Steffens, W.F.J. Verhaegh, R.J. Bril, and C. Hentschel. Qos control strategies for high-quality video processing. *Real-Time Systems*, 30(1):7–29, 2005.

9 Appendix A: Manual on (re)programming the Camera

Manual on (re)programming the Camera

Norbert Verhagen

September 20, 2011

Abstract

Preceding the actual (re)programming of the camera several physical connections need to be established and a number of programs need to be run. All actions necessary to prepare the camera for programming are described in this document, including the hard and software required for each step. Following the preparation, the (re)programming itself is addressed and finally the errors, failures and exceptions that occurred while working with the camera are mentioned and possible solutions are given. This guide is written for MS Windows XP, however, many of the programs have alternate versions for other operating systems.

Contents

1 Introduction

Reprogramming the camera, more specifically: updating the firmware of the camera, requires multiple pieces of hardware, drivers and several applications. Section 2 provides a listing of all that is required. The ensuing sections contain an in depth description of how and when to use the listed components, each focussing on a specific piece of hardware or software. Section 3 centers on the serial cable connection and hyperterminal. Section 4 explains the usage of the JTAG connection and Section 5 focuses on the Stretch IDE development environment and on the OCD deamon. The Web Update Application and its purpose is the topic of Section 6. The final sections 7 and 8 contain a few final remarks and some errors that were encountered while working with the camera.

The firmware for the camera actually consists of two applications: the S6SCP application for the main processor and the S6AUX application for the auxiliary processor. Together they are referred to as the 'main application'. Both the S6SCP and S6AUX applications have individual projects for the development environment and can be compiled individually. However, both individual applications cannot function without one another. Therefore, when using the camera both must be available on the camera.

Alongside the main application there is a bootloader and a backup application installed on the camera. The bootloader handles the booting and the boot options, see also Section 3. The backup application is executed when the main application is either unavailable or when the CRC check on the main application during the boot sequence fails.

The main application hosts a website on which new firmware, a new backup application and a new bootloader can be uploaded. However, to prevent version conflicts, an additional application is available: the Web Update Application. See Section 6 for the details. This application has its own project in the development environment, as it is completely independent from the main application. Its only function is to provide a website on which new firmware (only the S6SCP and S6AUX applications) can be uploaded. To replace the bootloader and the backup application the main application's website is needed.

There are two distinct methods of reprogramming the camera, as shown in Figure 1. The flowchart shows which actions need to be performed, with the numbers between parentheses referring to the corresponding sections in this document.

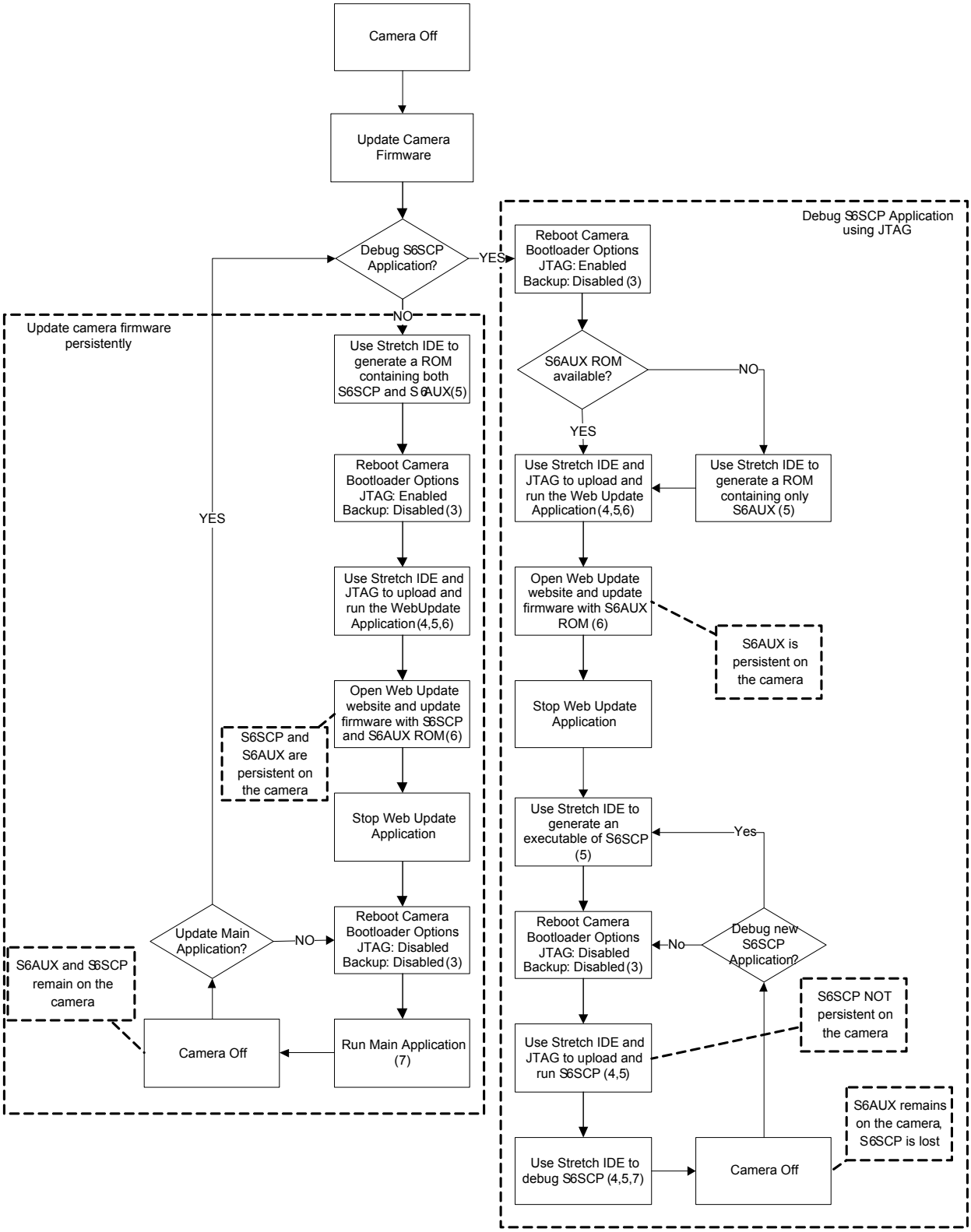


Figure 1: Updating Firmware Flowchart

The main differences between the two methods is that one method updates the firmware persistently, the S6SCP and S6AUX applications remain on the camera even after a reboot or when the camera is turned off. The consequence is that the development environment cannot be used for debugging. The second method does allow for debugging, but does not update all of the firmware persistently. After a reboot the S6SCP application is lost and must be uploaded again, the S6AUX application does remain on the camera.

2 Required Hard and Software

Before starting the process of reprogramming, a listing of the required hard and software is given in the next subsections.

2.1 Hardware

- The camera. More precisely: the VPA-PM board with the P5MSB-A attached to one of the dataports, see Figure 2.
- The adapter to power the camera.
- The Catapult EJ-1 or EJ-2 ethernet to JTAG device, with power supply.
- The Catapult Flying Leads & JTAG print.
- A serial to RJ45 cable and when no serial port is available on the connecting computer, a serial to USB cable.
- 3 UTP patch cables.
- A switch.
- A computer or laptop with a serial or USB port and an ethernet port. The computer should be capable of running Windows XP SP2 or capable of running Windows XP SP2 in a virtual machine.

2.2 Software

- Windows XP SP2 or SP3 either native or in a Virtual Machine.
- If needed, the Serial to USB cable driver (`http://www.conceptronic.net`)

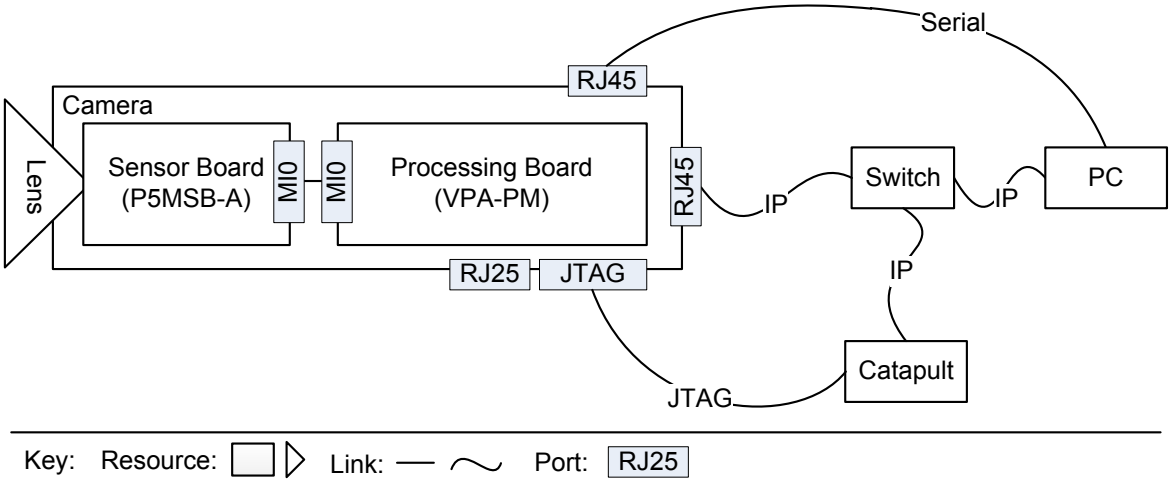


Figure 2: Camera Components & Environment

- Hyperterminal (included with Windows XP) or an equivalent program such as putty (`http://www.putty.org`).
- Mozilla Firefox 3.x (`http://www.mozilla.com/en-US/firefox/fx/`)
- Catapult configuration tool (`http://www.byte-tools.com`, in the downloads section, need an account to download)
- Xtensa OCD Deamon 7.1.0 (`http://stretchinc.com`, in the downloads section, need an account to download)
- Stretch IDE (`http://stretchinc.com`, in the downloads section, need an account to download)[1]
- Apple QuickTime (`http://www.apple.com/quicktime/download/`)
- VLC player (`http://www.videolan.org/vlc/`)
- VMWare (`http://www.vmware.com`, need an account to download)
- Source of the application, containing the S6SCP and S6AUX projects, from VDG
- Source of the WebUpdate project, from VDG

[1]Stretch IDE also requires a machine specific licence.

3 Hyperterminal and the Debug Mode

The first step in reprogramming the camera is to set the camera in 'debug mode' [4]. In this mode the JTAG port can be enabled and set to receive new software. It is also possible to force the camera to start the backup application.

Before starting, make sure the camera is switched *off*. Next, connect the serial to RJ45 cable to the unshielded RJ45 port (the audio port) on the camera. Then plug the serial end of the cable into the computer. However, as most new computers no longer have a serial port, a serial to USB cable could be needed with the corresponding driver, see Section 2. After the driver is installed an additional COM port should be available in Windows, which can be used as any other COM port. A note for Virtual Machine (VM) users: If after installing the driver there is no new COM port or if there is an error during the installation of the driver, it is possible that the USB connection is still connected to the host OS and not to the VM. Check the connected devices to the VM and make sure the USB to serial connection is connected to the VM.

To make a connection over the serial cable, a program is needed that can communicate over this type of cable. The standard Windows program for such connections is hyperterminal. After starting hyperterminal, a connection can be made over a COM port by clicking on the 'new connection' button. The 'connection description' window will pop up. On this window a name for the connection should be entered, followed by a click on the 'ok' button. The 'connect to' popup window will appear, here the correct COM port should be selected, again followed by a click on the 'ok' button. The last window to pop up is the 'COM properties' window, on here the settings listed in Table 1 should be entered, followed by a final click on the 'ok' button.

Field	Value
Bits per second	115200
Databits	8
Parity	None
Stopbits	1
Flowcontrol	none

Table 1: Serial Port Connection Details

Subsequently, the new connection can be called by clicking on the 'call' button on the toolbar of hyperterminal, even when the camera is unpowered. After the connection has started, the camera can be powered up. Immedi-

ately thereafter hyperterminal should display the bootloader followed by the question to enter debug, the user then has a very short amount of time to press any key and enter the debug mode. If the moment to enter the debug mode is missed, the camera can be reset by turning the power off and back on.

In debug mode, 3 options are displayed:

1. Enable/Disable Backup Firmware
2. Enable/Disable JTAG
3. Reboot Camera

After setting an option, the camera automatically reboots with the new setting. If the Backup Firmware option is enabled, the camera will reboot and then execute the backup application. If JTAG is enabled, the camera will reboot and after the boot sequence has been completed the message 'waiting for debugger to load the second-stage application' will be displayed on hyperterminal, signaling the readiness of the camera to receive over the JTAG connection. The details are in sections 4 and 5. Option 3 instantly reboots the camera. Options 1 and 2 are persistent, they remain set to the value entered even after a reboot.

Even though the serial connection is only necessary when changing the bootloader options, it is useful to keep the serial connection active when uploading new firmware or when debugging the application. Errors that occur are usually reported via the serial port. Other useful information is also given. For instance, the IP address of the camera is reported over the serial connection during the startup phase of the main application and the Web Update application.

A couple of remarks on the usage of hyperterminal:

- It is possible to record the output of the camera displayed on hyperterminal to a file. On the menu 'transfer' in the hyperterminal window select 'capture text', enter a filename and click the 'start' button. From this point on, all new text displayed on hyperterminal will also be written to the selected file. To stop recording, go to the menu 'transfer', select 'capture text' and click on 'stop'.
- Hyperterminal will hang once in a while, usually this is noticeable when no new text is displayed when new text is expected. The only solution is to end the hyperterminal process and restart hyperterminal. The camera need not be reset, the serial connection can be restarted during runtime. The same applies to the serial cable, it can be attached and detached while the camera is on.

4 JTAG: Usage of the ByteTools Catapult

First, a warning on the use of the ByteTools Catapult: the Catapult is *fragile*. Connecting the wires incorrectly or failing to follow the steps for connecting and disconnecting the Catapult to the camera could result in a permanently damaged Catapult. The strict sequence of steps is given at the end of this section, after the Catapult has been looked at in detail.

Before physically connecting the Catapult to the camera, the Catapult can be connected to by a computer by using an ethernet cable. When no DHCP server is available, the device will default to the factory settings, see Table 2 [2].

Field	Factory Default Catapult EJ-1	Factory Default Catapult EJ-2
IP Address	192.168.15.120	192.168.6.120
network mask	255.255.255.0	255.255.255.0
gateway	192.168.15.1	192.168.6.1
serial number	041244	050157

Table 2: Catapult Connection Details

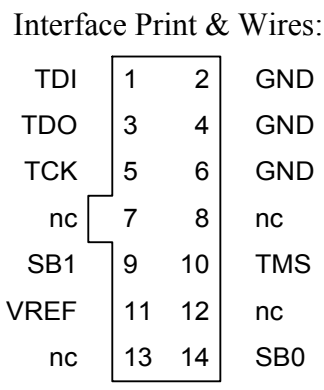
Make sure the IP of the connecting computer is in the range (pay attention to the subnet mask of the connecting PC), for instance assume the static IP 192.168.15.126 when the EJ-1 Catapult is on factory default.

After plugging in the Catapult (still not connected to the camera), the red waring LED may start to flash, this is normal, and is explained below. Regardless of the flashing LED, some time after plugging in the Catapult, the device should become pingable. The Catapult configuration tool, the software accompanying the device, should now be able to connect to the device as well. After the device has been located by the tool, click on 'test' to perform a self test. It is possible to assign the catapult a static IP by selecting the device in the window and clicking on the 'connect' button. On the popup window click on the 'force unlock' button, wait for the confirmation. Then press the 'modify' button and enter a new IP address on the new window. The static IP can be used to put the catapult in the same IP range as the camera, which is useful if no DHCP is available and a hub or switch is used. The Catapult can be reset to its factory defaults by selecting 'restore factory defaults' from the configuration menu and entering the serial number (the serial number is on the casing of the device).

Alternatively, the catapult hosts a website on which settings can also be changed. The settings include entering a static or dynamic IP address

and allow the updating of the Catapult's firmware and the resetting of the Catapult. In addition, some statistics on the network usage of the catapult are shown. The website can be reached on the IP address of the Catapult.

The flashing red warning LED, usually occurring when the Catapult is not connected to the camera, is indicative that the target VREF (the voltage reference on the target camera) is not correct. The LED should stop flashing after the device has been connected to the camera (the method is listed at the end of this section) and both have been powered up. If the LED continues to flash after the powering of both devices, the flying leads may be connected incorrectly to the interface print. The proper connection is shown in Figure 3 (different from the connection listed here [4]). If, after connecting the Catapult to the camera and powering up both, the red warning LED continues to flash it may be that a different interface print is provided. Shifting all the connections by adding two (ie. TMS should be connected to pin 1 but now connect it to pin 3, see [4]) may solve the problem. However, this should only be attempted as a last resort and make sure that both the Catapult and the camera are powered off and unconnected when changing the flying leads. In general: take great care and double-check the connections and pay particular attention to the VREF and GND wires. If any other problems occur, a good place to start is the ByteTools wiki, see [3].



Label	Description	Direction[2]
TDI	JTAG TDI	Output
TDO	JTAG TDO	Input
TMS	JTAG TMS	Output
TCK	JTAG TCK	Output
SB0	Sideband 0	Bidir
SB1	Sideband 1	Bidir
VREF	JTAG IO voltage	Input
GND	Signal Ground	Input
nc	Not used	

Figure 3: JTAG pins

[2]Direction of this signal is relative to the Catapult. For example, a signal with the direction 'Output' is an output from Catapult to the target board.

The strict sequence (do not deviate from this sequence!) to connect the Catapult to the camera:

1. The camera and Catapult are unconnected and both are powered off.
2. Connect the Catapult to the camera, by connecting the print to the JTAG connector on the camera, see Figure 4 [4].
3. Power the Catapult.
4. Power the camera.

When turning off the camera and Catapult follow this strict sequence (again: do not deviate from this sequence!):

1. The camera and Catapult are connected and both are powered on.
2. Power off the camera.
3. Power off the Catapult.
4. Disconnect the print from the JTAG connector on the camera.

It is possible to hard-reset the camera (i.e. power off the camera and then powering it on again) without powering off the Catapult when both are connected. However, make sure that when the Catapult is not powered it is never connected to a camera that is powered.

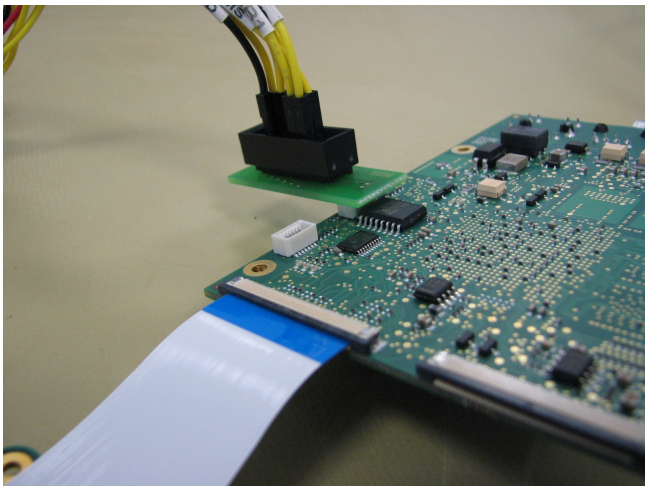


Figure 4: Correct JTAG connection

5 Stretch IDE and OCD Deamon

The development environment for the camera application is Stretch IDE. It allows the user to compile the application and transmit it to the camera over the JTAG connection, see Section 5.1. It also offers the user the feature of debugging the application by setting breakpoints and by providing a method to inspect the values of variables, see Section 5.2. The exact steps needed to debug the S6SCP application are also listed in that section. In the remainder of this section, the setup of the physical and software connections are looked at in more detail. Note that the environment runs on MS Windows XP and the licence is machine dependent. To circumvent this, a virtual machine can be build with Windows XP (SP2 or SP3) as operating system and Stretch IDE can then be installed in the VM. A licence is still needed, but it is created for this virtual machine. The VM itself can be copied to other systems, and Stretch IDE can be used on these systems without needing licences specific to these systems.

Before a connection can be made between Stretch IDE and the camera via the JTAG connection, two items need taking care of: The Xtensa OCD Deamon (debug) needs to be running and a 'board connection' in Stretch IDE needs to be available. First, the OCD Deamon. Before it can be started, the configuration file of the OCD Deamon needs to be altered. The configuration file is shown in Section 9.1, usually the only change that is needed is the substitution of the IP address of the Catapult listed in the file with the correct IP. After starting the OCD Deamon, it will automatically connect to the Catapult. For a successful connection, both the Catapult as well as the camera need to be plugged in and connected to each other, see Section 4. The Deamon will display an error message if the connection could not be established.

A 'board connection' in Stretch IDE can be created by selecting 'new board connection' under the menu 'tools', which displays a popup window after clicking. On this window the connection settings shown in Table 3 should be entered.

Connection Name	S6SCP
Connection type	JTAG (OCD)
Host name	localhost
Port	20000
Board type	S6105CAM-bootloader

Table 3: Stretch IDE Connection Details

Note that the host name is 'localhost' and not the IP of the Catapult, as

Stretch IDE connects to the OCD Deamon (running on the local host) and not directly to the Catapult. Also, the 'board connection' is saved and has to be created only once.

After completing these two steps, the application can be uploaded and run on the camera via the JTAG connection, by clicking on the 'debug' button on the toolbar of Stretch IDE and having the Xtensa OCD Deamon running in the background. The uploading can take some time, 1.5 to 2 minutes for an upload is not uncommon.

The S6SCP application requires the S6AUX application to function correctly. This complicates matters when using the JTAG connection to run and debug the S6SCP application. Required is that a stand-alone ROM, see Section 5.1, of the S6AUX application is uploaded to the camera as firmware and that during the bootsequence of the camera the JTAG connection is *disabled*. If the JTAG connection is enabled and the attempt is made to run and debug the S6SCP application over the JTAG connection, the S6SCP application will deadlock.

5.1 Executables, ROMs and Libraries

To run and debug the S6SCP application, an executable of the S6SCP application needs to be generated. To effect this, open the S6SCP project in Stretch IDE and select on the 'project' menu 'active target'. On the popup window, select from the pull down menu target is 'remote'. After selecting 'build' from the 'build' menu a .exe file will be generated. Select 'start debugging' on the 'debug' menu to upload the executable to the camera over the JTAG connection and to start debugging the S6SCP application. Note that the S6AUX rom must be on the camera when debugging the S6SCP application, see Figure 1. How to generate a rom is explained below.

For debugging the S6SCP application, a ROM of the S6AUX application is needed and to upload the main application permanently to the camera a ROM containing both applications (S6AUX and S6SCP) needs to be available. To generate a ROM from an application, select on the 'project' menu the 'active target' field. On the popup window select target is 'romable' from the pull down menu. The project properties, accessible via the 'project' menu and then selecting 'project properties', will show that the extension of the generated file will still be .exe. This is normal. During post-processing the .exe is wrapped into a .rom file. For the S6AUX ROM this is straight forward, only the .exe of the S6AUX application is needed. For the ROM containing the main application, the .exe of the S6SCP application as well as the .exe of the S6AUX application are necessary. If the executable of the S6AUX application is unavailable it can be created with the S6AUX project. For

this, open the S6AUX project in Stretch IDE. Select 'active target' from the 'project' menu and on the popup window select 'romable' from the pull down menu. Compile the project by selecting 'build' from the 'build' menu and a ROM with the extension .exe will be generated.

The S6SCP application depends on several libraries. These are contained in two files: a .a file and an _d.a file. Both files contain the entire library but are used in different circumstances. The .a variant is used when an executable of the application is generated and the _d.a variant is used when the application is compiled into a ROM. This is especially relevant when changes need to be made to the source of libraries, as both variants need to be generated. The process of generating is analogous to generating an executable. Open the library project in Stretch IDE and select on the 'project' menu 'active target'. Select 'remote-debug' for generating a .a file and select 'remote-release' for generating an _d.a file. The files are created when 'build' from the 'build' menu is selected.

5.2 Debugging in Stretch IDE

The debugging options in Stretch IDE are accessible from the 'debug' menu, some of the features have quick access buttons on the toolbar and some are also accessible by right clicking in the editor section of the Stretch IDE window. The application can be started, paused and stopped. Breakpoints can be set, with the condition that it is only possible to set breakpoints when the application is not running or is paused. From the same menu the 'step into', 'step over' and 'step out' features are available.

Values of variables can be inspected by right clicking on the variable and selecting 'add watch', an additional window will be opened on which the selected variables are listed and their values are displayed during run time when the application is paused.

The steps for debugging the S6SCP application:

1. Camera & Catapult are connected and powered, see Section 4 for the details.

2. Upload the S6AUX ROM to the camera using the Web Update Application, see Section 6. Note that the S6AUX ROM remains persistent on the camera and needs to be uploaded only once.

3. Set the camera debug mode options, see Section 3, to: 'backup firmware' is disabled, 'JTAG' is *disabled*. Note that these options are persistent and remain set even after a reboot of the camera. After the settings have been entered, the camera automatically reboots.

4. Start the OCD Deamon.
5. Open the S6SCP project in Stretch IDE.
6. Create the 'board connection', see Section 5. Note that this board connection is saved and needs to be created only once.
7. Click on the 'debug' button to start debugging.
8. When finished debugging, click on the 'stop' button to end the debugging process. The camera automatically reboots.

6 Web Update Application

The Web Update Application offers a simple way of persistently updating the firmware on the camera. It is possible to upload the main application (both S6SCP and S6AUX in a single ROM) or to upload a ROM containing just the S6AUX application (necessary for debugging of the S6SCP application). The Web Update Application consists solely of a website with the option of uploading and installing firmware. It is designed to be uploaded and run via the JTAG connection, temporarily replacing the application on the camera. The Web Update Application requires the camera to be in debug mode, see Section 3, with the backup application disabled and the JTAG connection *enabled*. After these settings have been entered over the serial connection, the application can be uploaded and run by using the JTAG connection and Stretch IDE, a comparable method is used as for the uploading and running of the S6SCP application, see Section 5.2. The exact steps are listed at the end of this section.

After the camera has booted into the Web Update application, the website can be reached on the IP address reported via the serial port. On the website the new firmware can be selected by clicking on 'browse'. Be careful to upload only a ROM containing the correct firmware and not any other file, as the website does not check the file that is uploaded! By clicking on 'submit' the firmware is uploaded and installed. The process can be followed via the serial port, it will report progress and either success or failure. On the successful upload of the new firmware, reboot the camera, enter debug mode and disable the JTAG connection. The camera will now boot into the newly uploaded firmware. If the uploading of the firmware fails an error is displayed on a popup on the website, a more detailed description of the error is reported via the serial connection.

The steps for uploading a ROM using the Web Update Application:

1. Camera & Catapult are connected and powered, see Section 4 for the details.

2. Set the camera debug mode options, see Section 3, to: 'backup firmware' is disabled, 'JTAG' is *enabled*. Note that these options are persistent and remain set even after a reboot of the camera. After the settings have been entered, the camera automatically reboots.

3. Start the OCD Deamon, see Section 5.

4. Open the Web Update project in Stretch IDE.

5. Create the 'board connection', see Section 5. Note that this board connection is saved and needs to be created only once.

6. Click on the 'debug' button to start the Web Update Application.

7. After the camera has started the Web Update Application (reported over the serial port) log on to the Web Update website by entering the IP address of the camera in the webbrowser. Note that the IP address is mentioned in the output over the serial port.

8. Upload the ROM using the website. As with all firmware updates, do not interrupt the uploading process of the ROM in any way. Failure to do so can result in a permanently damaged camera.

9. After the upload has finished, reported by the website and over the serial port, stop the Web Update Application by clicking the 'stop' button in Stretch IDE. The camera automatically reboots.

10. Set the camera debug mode options, see Section 3, to: 'backup firmware' is disabled, 'JTAG' is *disabled*. After the settings have been entered, the camera automatically reboots and starts the firmware that has been uploaded.

7 Application

Some notes on the S6SCP application itself:

1. If the P5MSB-A is not connected to one of the dataports on the VPA-PM the S6SCP application will crash immediately.

2. After the camera has booted, the application will first try to connect to a DHCP server to acquire an IP address. If none is found the camera assumes a static IP if this is set on the application's website. If no static IP is set, the camera assumes an IP in the 169.x.x.x range. The IP that has been assumed is reported during the booting of the camera via the serial port, see Section 3

3. The application's website contains 5 tab pages[3]:

    1. Info, which contains general information on the application and the hardware
    2. Video, options specific to the video, also contains the jpeg image which is refreshed every few moments.
    3. Live, containing the actual videostream. To be able to see the videostream the Quicktime plugin is required.
    4. Network, options specific to the network. Here a static IP address can be assigned to the camera, see 9.2. This page also contains the options to switch on or of the Video Multicast, the audio stream (both in and out) and the signaling interface.
    5. Misc, which provides access to the logs, the possibility to upload a new bootloader and the option to reboot the camera.

4. It is also possible to view the stream in the VLC player or in the QuickTime Player. The address of the stream: `rtsp://192.168.6.146/media`. See Section 9.2 for the IP address of the camera or look at the IP that is reported over the serial connection during the startup of the main application.

In [1] a more in depth description of the use of the camera and also the main application is given.

[3]The available tab pages depend on the version of the application. Older versions have 5 tab pages, newer versions have fewer tab pages and different contents.

8 Failures, Errors and Possible Solutions

This section contains a number of failures and errors that were encountered.

- Without the sensor board (P5MSB-A), the application crashes immediately. Correctly attach the sensor board to one of the dataports on the processing board to solve this error.
- If the video_in task blocks on function *jpeg_open()*, this may indicate that the program for the auxiliary processor (S6AUX) is not loaded. Usually this is the result of the bootloader setting JTAG is set to enabled, see Section 3. By setting JTAG to disabled followed by a reboot of the camera, the application should then no longer block on the function *jpeg_open()*.

References

[1] *VDC-5000HD User manual*.

[2] Byte Tools, *Byte Tools quick start*, `http://www.byte-tools.com/btejquickstart.html`.

[3] ———, *Byte Tools wiki*, `http://www.byte-tools.com/wiki/doku.php`.

[4] Prodrive, *Prodrive memo: connecting the prodrive s6105 processor board*.

9 Appendices

9.1 Appendix A: OCD Deamon Configuration File

The configuration file is shown below. Make sure the IP address ('ipaddr') of the Catapult is correct.

```
<configuration>
  <controller id='Controller0' module='catapult' speed='12500000'
    ipaddr='192.168.6.140'/>
  <driver id='XtensaDriver0' module='xtensa' step-intr='mask,stepover' />
  <chain controller='Controller0'>
    <tap id='TAP0' irwidth='19' bypass='0x0ffff' />
    <tap id='TAP2' irwidth='5' />
    <tap id='TAP3' irwidth='5' />
  </chain>
  <system module='jtag'>
    <component id='Component0' tap='TAP2' config='tensilica' />
    <component id='Component1' tap='TAP3' config='tensilica' />
  </system>
  <device id='Xtensa0' component='Component0' driver='XtensaDriver0' />
  <device id='Xtensa1' component='Component1' driver='XtensaDriver0' />
  <application id='GDBStub' module='gdbstub' port='20000'>
    <target device='Xtensa0' />
    <target device='Xtensa1' />
  </application>
</configuration>
```

9.2 Appendix B: Camera Configuration

The configuration of the camera as entered on the website of the camera:

Setting	Value
IP Address	192.168.15.122
Subnet Mask	255.255.255.0
Gateway	192.168.15.1