



MSc THESIS

Speeding up MPEG-4 colorspace conversion

Abstract



CE-MS-2004-12

It is envisioned that the correct answer for future performance enhancement of embedded systems is through dedicated hardware for the computation intensive kernels of the targeted applications. A currently recognized set of applications aimed in the new generation mobile devices are the multi-media encoders and decoders. In order to preserve the flexibility of such systems, solutions involving general-purpose processors and reconfigurable hardware are emerging. To affirm such approach, the SMOKE (Speeding up MPEG-4 Operational Kernels on Excalibur) project was initiated. This thesis presents only a part of the SMOKE project. SMOKE's main goal is to provide a full-featured hardware accelerated MPEG-4 decoder. The MPEG-4 decoder selected for this project is based upon the opensource codec XviD. The hardware platform that was used for development during the SMOKE project is DAMP. The DAMP development board is specially designed for multimedia applications and is based on the Excalibur device from Altera. This device integrates an ARM922T processor and an FPGA fabric into a single chip. To support the development on the DAMP platform, the Linux kernel was ported to the development board. Furthermore, a stand-alone environment was created to validate the operation of the hard-

ware accelerators. The *inverse discrete cosine transform (IDCT)* and *colorspace conversion (CSPC)* kernels were identified as the two most computationally intensive parts of the MPEG-4 decoder. Both kernels are responsible for more than 40% of the total execution time. This project focuses on the acceleration of the *colorspace conversion* kernel. The hardware/software (HW/SW) co-design exploration of the *colorspace conversion* accelerator is presented. By placing the colorspace conversion kernel in hardware a speedup of 1.24 was realized. When both the hardware accelerated IDCT and CSPC kernels are used by the MPEG-4 decoder the measured speedup is 1.40.

Speeding up MPEG-4 colorspace conversion
An implementation of a hardware accelerated MPEG-4
decoder on DAMP.

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Jonathan Hofman
born in Gouda, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Speeding up MPEG-4 colorspace conversion

by Jonathan Hofman

Abstract

It is envisioned that the correct answer for future performance enhancement of embedded systems is through dedicated hardware for the computation intensive kernels of the targeted applications. A currently recognized set of applications aimed in the new generation mobile devices are the multi-media encoders and decoders. In order to preserve the flexibility of such systems, solutions involving general-purpose processors and reconfigurable hardware are emerging. To affirm such approach, the SMOKE (Speeding up MPEG-4 Operational Kernels on Excalibur) project was initiated. This thesis presents only a part of the SMOKE project. SMOKE's main goal is to provide a full-featured hardware accelerated MPEG-4 decoder. The MPEG-4 decoder selected for this project is based upon the opensource codec XviD. The hardware platform that was used for development during the SMOKE project is DAMP. The DAMP development board is specially designed for multimedia applications and is based on the Excalibur device from Altera. This device integrates an ARM922T processor and an FPGA fabric into a single chip. To support the development on the DAMP platform, the Linux kernel was ported to the development board. Furthermore, a stand-alone environment was created to validate the operation of the hardware accelerators. The *inverse discrete cosine transform (IDCT)* and *colorspace conversion (CSPC)* kernels were identified as the two most computationally intensive parts of the MPEG-4 decoder. Both kernels are responsible for more than 40% of the total execution time. This project focuses on the acceleration of the *colorspace conversion* kernel. The hardware/software (HW/SW) co-design exploration of the *colorspace conversion* accelerator is presented. By placing the colorspace conversion kernel in hardware a speedup of *1.24* was realized. When both the hardware accelerated IDCT and CSPC kernels are used by the MPEG-4 decoder the measured speedup is *1.40*.

Laboratory : Computer Engineering
Codenummer : CE-MS-2004-12

Committee Members :

Advisor: Georgi Gaydadjiev, CE, TU Delft

Chairperson: Stamatis Vassiliadis, CE, TU Delft

Member: F.L. Muller

Member: N.V. Budko, EM, TU Delft

*To Gitty for her love and patience,
and to my parents for their endless support.*

Contents

List of Figures	ix
List of Tables	xi
Acknowledgements	xiii
1 Introduction	1
1.1 Objectives and contributions	2
1.2 Thesis framework	3
2 Software Environment	5
2.1 Toolchain	5
2.1.1 The C library	5
2.1.2 Building a platform specific toolchain	6
2.2 Bootloader	6
2.3 Runtime Environment	7
2.4 The MPEG-4 decoder	8
3 Building the DAMP SDK	9
3.1 Porting Newlib to DAMP	9
3.1.1 The Newlib stubs	9
3.1.2 Adding filesystem capabilities	11
3.2 Initializing the environment	14
3.2.1 Enabling the MMU	14
3.2.2 Installing the interrupt table	14
3.2.3 Setting up the C environment	16
3.2.4 Starting the application	16
3.3 Building standalone applications	16
4 Modifying Linux	17
4.1 Configuring and building the kernel	17
4.2 Modifications to the kernel sources	19
4.2.1 DAMP specific modifications	19
4.2.2 Setting up the memory map	20
4.2.3 Modifying the ethernet driver	21
4.3 Booting the kernel on DAMP	24
4.3.1 Booting from the flash filesystem	25
4.3.2 Booting from the network filesystem	26
4.4 Building hardware drivers	27
4.4.1 The difference between user space and kernel space	27

4.4.2	Character device drivers	27
5	Hardware Environment	33
5.1	The DAMP platform	33
5.2	The hardware/software interface	35
5.2.1	Implementing MOLEN on Excalibur	35
5.2.2	Memory mapped interfaces	38
5.3	The VGA Interface	42
5.3.1	The VGA signals	42
5.3.2	The DAMP VGA hardware	45
5.3.3	Implementing the VGA controller	45
5.3.4	Software drivers	47
6	Speeding up the Colorspace conversion	51
6.1	Colorspaces	51
6.2	Colorspace conversions	52
6.2.1	Software colorspace conversion	54
6.2.2	Hardware colorspace conversion	55
6.3	The modifications of the software	57
7	Experimental results	59
7.1	Test methodology	59
7.2	Speed up calculations	60
7.3	Measurements	61
7.3.1	Kernel speedup	61
7.3.2	MPEG-4 decoder speedup	62
8	Conclusions	65
8.1	Summary	65
8.2	Main contributions	66
8.3	Future Work	67
	Bibliography	70
A	The character device framework	71
A.1	Character device driver source	71
A.2	Device driver load script	73
A.3	Device driver unload script	74
B	AMBA Slave	75
C	The colorspace conversion	79
C.1	Software colorspace conversion	79
C.2	Hardware colorspace conversion	80
D	The DMA controller VHDL source	83

E	The VGA software driver for Linux	89
E.1	The Linux VGA driver (H-file)	89
E.2	The Linux VGA driver (C-file)	90
E.3	VGA driver load script	99
E.4	VGA driver unload script	100
F	The VFS sources	101
F.1	VFS	101
F.2	Terminal driver	103
F.3	memfs	104

List of Figures

1.1	Pentium 4 clockspeed/performance relation [31].	1
3.1	The layered structure of the filesystem implementation of the DAMP SDK.	11
3.2	The call path when the <i>fwrite</i> function is called.	13
3.3	The different methods for installing the interrupt vector table.	15
4.1	The signal path of the external interrupt before and after modification.	22
4.2	The layout of the flash memory	25
4.3	The locations of the arguments of the <i>read</i> operation.	30
5.1	A block diagram of the excalibur device.	34
5.2	The organization of the MOLEN architecture	36
5.3	The interface of the DMA controller.	40
5.4	The statediagram of the DMA control FSM.	41
5.5	The statediagram of the statemachine for the DMA controller responsible for the AHB transfers.	43
5.6	The scanning of a VGA screen.	44
5.7	The timing relations of the <i>hsync</i> and <i>vsync</i> signals.	44
5.8	Blockdiagram of the VGA controller.	46
5.9	The internal components of the VGA driver.	46
6.1	The quality loss after converting an image to yv12.	53
6.2	The different formats for storing images in the luma-chroma colorspace.	55
6.3	Block diagram of the VGA driver supporting yv12	57
7.1	The execution times of the different configurations of the decoder.	62

List of Tables

2.1	The profiling results of the XviD decoder	8
3.1	The Newlib library stubs implemented for the DAMP SDK.	10
5.1	The different input formats and their minor types	48
7.1	The speedup of the colorspace conversion.	61
7.2	The speedup of the IDCT.	62
7.3	The measured speedup of the XviD codec.	63
7.4	The speedup of the XviD codec.	63

Acknowledgements

A large number of people have supported me during my thesis project or my study at the Computer Engineering group. Of course, it is not possible to mention all of them, but nevertheless I have compiled a short list.

First of all I want to thank Guido de Goede. During the project we have worked intensively together. He has been a good colleague on who I could depend. He is a good friend, who made working on the project more pleasant. Even when we had to make long days and had to miss the summer vacation he always kept his big smile and his happy mood, which was one of the reasons I enjoyed working together.

The second person to whom I am deeply grateful is my advisor, Ir Georgi Gaydadjiev. His patience, enthusiasm and his great experience has helped us to overcome difficulties. I want thank him for the great effort he made by reviewing the thesis within a very short time period, because deadlines had to be met.

I owe great gratitude to Dr. Sorin Cotofana who has been a mentor to me during my study at the Computer Engineering group. He has always provided me with reflection and other insights at the field. Also I am thankful for the time and effort he has spend on the CSIDC competition, which I, together with two other students, have attended. Even though the result of our attendance to the competition was not the finals in Washington, I have learned a great deal by it.

Furthermore, I want to thank Prof. Stamatis Vassiliadis, for his encouraging and enthusiastic motivation he gave me during my study at the Computer Engineering group. I want to thank Bert Meijs for always wanting to help with computer related problems and for providing my colleague and me with all the computer facilities we needed for the SMOKE project. I would also thank all the other members of the Computer Engineering group, which I had a very good time with.

Last but not least I want to thank my family and friends, who have always supported me.

Jonathan Hofman
Delft, The Netherlands
January 5, 2005

Introduction

Since many years the processor industry is challenged to increase performance of processors rapidly. When faster processors find their way to the market, more demanding applications are developed. Until now the performance increase has followed the Moore's law¹. This law states that the performance of processors will double every 18 months. This was mainly realized by using smaller (sub micron) technologies. The performance is usually thought to be equivalent to the clockspeed of a processor. Deeper analysis, however, reveals that the latter is a misconception. Even though the clockspeed of processors still apply to Moore's Law, the performance, measured by how fast applications can execute, does not. The trend in performance against clockspeed is depicted in figure 1.1. In this figure the relation between the clockspeed and the performance of a Pentium 4 processor is shown. The *optimal* line shows a linear relation between clockspeed and performance. This linear relation is the most optimal behavior. The real relation between clockspeed and performance, measured by the *Specfp2000* benchmark suite, is less optimal. The results have been matched to both an *Amdahl* and *exponential decay* function, which give a good relation between the clockspeed and the performance.

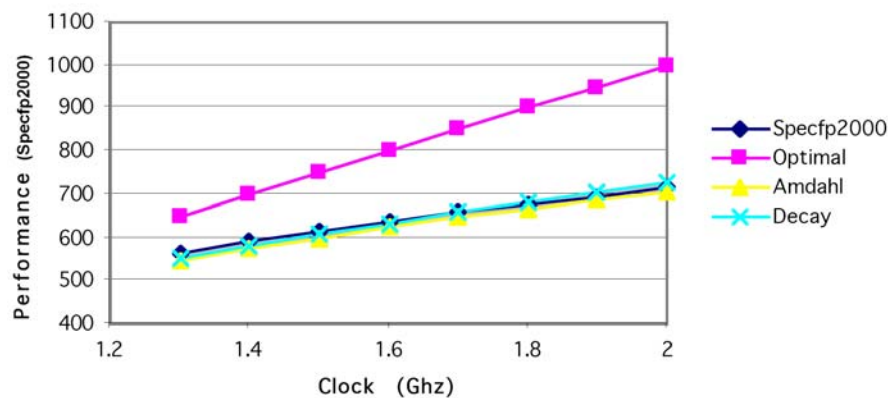


Figure 1.1: Pentium 4 clockspeed/performance relation [31].

To be able to cope with future computation intensive applications new paradigms are emerging. One approach for speeding up high demanding applications is to design Application Specific Integrated Circuits (ASICs). In ASICs, the high demanding computations can be optimized for best performance results. Unfortunately, even though the ASIC approach works fine in an embedded environment (where a system is applied to perform a specific task) ASICs are not applicable for General Purpose Machines (GPM).

¹Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

A GPM is a computing device that can be applied to a large variety of tasks. Because it is unknown in advance for which tasks the GPM's will be used, ASIC's are not suitable.

Since the mid nineties Reconfigurable Computing (RC) is becoming an increasingly popular computing paradigm. It combines the flexibility of general purpose processors with the computational power of an ASIC approach. Where nowadays the hardware in a GPM is fixed and the software is responsible for adapting the environment to perform a specific task, RC will in the future make also the hardware adaptable to the task it needs to perform at any given moment. The latter gives great potential for the handheld mobile devices, which are becoming more and more GPM's nowadays. These devices are currently restricted in the amount of dedicated hardware that can be added in the form of ASIC's. Certainly when compared to a PC platform (where via add-on cards hardware accelerators can be added and replaced), reconfigurable hardware can give handheld devices the flexibility needed.

1.1 Objectives and contributions

The purpose of this thesis is to provide a full-featured hardware accelerated MPEG-4 decoder on an Excalibur [8] platform. The decoder will be implemented in two different environments. One running Linux as operating system and one running the decoder as a stand alone application with no operating system.

An existing *opensource* codec will be used for the implementation of the decoder, which is extended to support hardware acceleration of certain computation intensive kernels. The opensource codec chosen is XviD [36]. One of the reasons to choose XviD is its design for portability. Most of the opensource MPEG-4 projects available focus on i386² platforms only. Furthermore, XviD is used by a large community of users and has therefore proved its usefulness. In [14] a more in depth discussion of the choice for the XviD codec is provided.

This master thesis is part of the SMOKE (Speeding up MPEG-4 Operational Kernels on Excalibur) [20] project. SMOKE is part of the MOLEN project (initiated at the Computer Engineering Group of the University of Technology of Delft) and focuses on acceleration of a real life MPEG-4 decoder on a reconfigurable hardware platform. MOLEN on the other hand focuses more on the Instruction Set Architecture (ISA) level that should make this integration of hardware and software possible. In the future, SMOKE could form the basis of the implementation of a full featured MPEG-4 decoder on the MOLEN architecture.

The main contributions of this thesis are:

- An implementation of the colorspace conversion of the XviD codec in hardware.
- An implementation of an environment for running the accelerated XviD decoder as standalone application.
- An implementation of the Linux operating system on the DAMP platform.
- An implementation of a DMA controller for the Altera Excalibur device.

²i386 stands for the well excepted Intel (compatible) processors used in the desktop PC's nowadays.

- A feasibility study for implementing the MOLEN architecture on DAMP.
- An implementation of a VGA controller for the DAMP platform.

1.2 Thesis framework

The organization of this thesis is as follows. Chapter 2 gives a short introduction to two software tools used for the development. Furthermore, the two environments (Linux and stand alone) that are used to run the decoder and the decoder itself will be investigated. In Chapter 3, the stand alone environment will be explored in more details. Chapter 4 will present the Linux environment that was modified for the DAMP board. The hardware environment on which these runtime environments run will be presented in Chapter 5. One of the computation intensive kernels of the MPEG-4 decoder will be accelerated using specially designed hardware. The hardware accelerator design will be presented in Chapter 6. Chapter 7 will present the speedup realized by the hardware accelerated MPEG-4 decoder. Finally, Chapter 8 concludes the thesis and gives some recommendations and future directions.

Software Environment

2

In this chapter the software environment that was used during the project is explained. Section 2.1 will emphasize on the used tools (compilers, runtime library, etc) and how they were obtained. Section 2.2 will discuss the boot process for the DAMP platform in more depth. After the boot process has been performed a *'run-time environment'* is activated to support the execution of the decoder application. The runtime environments used are Linux and standalone execution respectively. An introduction to these environments will be given in Section 2.3. In Section 2.4 the MPEG-4 decoder used for the SMOKE project is introduced.

2.1 Toolchain

In order to develop software for a platform, a set of tools is required. This set of tools is commonly referred as a toolchain. Such a toolchain consists of compiler, linker and runtime library. Usually, some optional tools like a debugger are available to support the development process. The software that will run on the DAMP platform will be developed on a IBM compatible PC (referred as PC in this document), therefore a cross-toolchain is needed. The cross-toolchain runs on a host platform, but generates code for a target platform, that is different from that of the host.

Several commercial toolchains for the ARM are available, unfortunately only at very high cost. Therefore, the free available *GCC* [6] draws attention. *GCC* stands for GNU Compiler Collection. It is the toolchain used for many Linux distributions, including the one distributed by Redhat. A large number of processors are supported by *GCC*, including ARM which is available on the DAMP platform.

Even though *GCC* supports a large number of programming languages, for this project only the C language is used. This because, even though experts will never agree on what programming language is the best to use, C is the most accepted language in the industry for embedded software development.

2.1.1 The C library

Besides the compiler a runtime library is needed. The library that is usually used in combination with *GCC* is *glibc* that supports many standards (like *POSIX*) and fits seamlessly together with the Linux operating system. The drawback of this, is the large footprint (in codesize) and the great amount of effort that is needed for porting the library to different platforms or operating systems. For embedded systems, the memory footprint is usually a major consideration. However this is not the case for DAMP, because it has a large SDRAM and network support for loading programs from a remote

location that can contain a harddisk. The portability, however, becomes an issue when running programs on another (non-unix like) operating system or without an operating system at all. Several alternatives to glibc are available. For example uClibc [32], diet libc [15] and Newlib [23]. All of these libraries target on a small codesize, but not all of them focus on portability as well. This is for example well expressed in one of the readme files accompanying the sources of the uClibc library:

If you want to port uClibc to support some non-UN*X-like Operating System, you should probably stop using crack¹. It is bad for you. ;-)

Diet libc is also targeting a Linux environment. NewLib on the contrary has special support for porting to other operating systems or for running with no operating system at all [19]. As we will see later, two different software environments will be used in this project. One is using Linux as operating system and one running with no operating system at all. For this reason we will use two toolchains. Both based on GCC, but with using different libraries. The one used for compiling software for the Linux environment will be based on the glibc library. For the standalone environment the Newlib library will be used, because of its easy portability. The porting of Newlib to run without operating system will be covered in more depth in Chapter 3.

2.1.2 Building a platform specific toolchain

As said before, a cross-toolchain is needed to compile programs on a normal PC and run them on the targeted hardware (in this case DAMP). To obtain a cross-toolchain for a given platform one can use prebuild toolchains which are compatible with the target platform. Such prebuild toolchains are made available by individuals via the internet. The other option is to build a cross-toolchain from scratch. Building a cross-toolchain can be a tedious task. Fortunately, several scripts are available which perform this task without the builder requiring extended knowledge of the buildprocess. For the SMOKE project the arm-linux-gcc (version 2.9.95) toolchain for Linux and arm-elf-gcc (2.95-arm9-020528) for the standalone version were used. In addition, a cross-toolchain has been build from scratch for the newest versions (3.4) of GCC.

2.2 Bootloader

Before the software application can be executed the environment needs to be initialized. This process is also referred to as the boot process. During the boot several hardware modules are initialized. For example the clock sources are set to provide the correct clock frequencies, the SDRAM controller is started and the memory map is configured. Detailed information about the boot process can be found in [11]. When the boot process is finished an application can be loaded and executed.

When the application runs without an operating system there is no special need for a separate bootloader. The bootloader's functionality could be implemented into the standalone application itself. Nevertheless, using a bootloader in this case is still useful.

¹*Crack is a highly addictive form of cocaine that is typically smoked. The term crack refers to the crackling sound heard when the substance is heated [24]*

The DAMP environment incorporates flash memory for non-volatile storage. This would be the logical location to store an application. Unfortunately, programming the flash takes up a reasonable amount of time and the flash memory has a limited number of write cycles. Both arguments make the flash less attractive during the debugging cycle of an application. In this case, a solution where a highspeed interface is used to load a program into the SDRAM and execute this application from the SDRAM would be useful.

In order to fulfill the above requirements, a bootloader with networking support has been ported to the DAMP. The bootloader used is ARMboot [3]. Several other (more advanced) bootloaders are available on the Internet, but because ARMboot was already ported to the EPXA boards from Altera (DAMP is similar to these boards) the effort for porting to DAMP was reduced to setting some variables to a different value. Most of these changes are discussed in [38]. Besides the changes described in [38], the file containing the system wide configuration (*include/configs/config_damp.h*) was modified to support the CS8900A ethernet controller.

2.3 Runtime Environment

After the hardware has been initialized by the bootloader, the runtime environment can be started. This can be the operating system or the environment in which the standalone application will run. Each of these environments has their advantages and disadvantages.

A disadvantage of using an operating system is the overhead that is introduced when communicating with the hardware. The operating system often has protection mechanisms to prevent direct access to the hardware by user applications. These applications need to access the hardware via system calls to the operating system. This, in contrast to a application in the standalone environment, which has full control over the memory and hardware peripherals. The application in the standalone environment will therefore communicate faster with the hardware, improving the systems performance.

Another disadvantage of using an operating system, is the overhead introduced by the task scheduling. The switching between different tasks takes time. An operating system like Linux supports multitasking. This means that multiple processes can run on a single processor system. The operating system will interrupt a task that is running too long and give another process control of the processor. This type of scheduling is called pre-emptive scheduling. The Linux kernel itself can not be interrupted during processing and is therefore not pre-emptive [5]. The standalone environment does not support multiple tasks and therefore has no time penalty due to task switching.

The advantage of using an operating system however, is the basic support for operations needed by a large number of applications, like file access, memory protection, network support, etc. When building a standalone application all these operations need to be added to the application. For example when a filesystem is needed, the software to access the hardware on which the filesystem is located and the software for the filesystem itself need to be linked into the executable. This makes the application more complex and more vulnerable to bugs, then it would be when there was an operating

function	percentage of total execution time
colorspace conversion	26.92%
IDCT	19.73%

Table 2.1: The profiling results of the XviD decoder

system available.

Another advantage of using an operating system is, that porting of applications requires less programming effort. This is due to the fact that the operating system introduces a layer of abstraction to the platform specific hardware. For example, Linux is ported from an Intel architecture, to an ARM architecture. An network application that runs on the Intel architecture, can probably be ported to the ARM architecture, because Linux provides a general method for accessing network interfaces. Even though the underlying hardware can be quite different.

2.4 The MPEG-4 decoder

For the MPEG-4 decoder selection several opensource projects where explored. The XviD decoder was selected for the SMOKE project. One of the main reasons for choosing the XviD decoder, was its great portability. Without any modification of the source code it could be compiled for the DAMP platform. Furthermore, the high development activity and the large community using the XviD decoder, prove that XviD is a mature product.

The XviD decoder will be accelerated by placing some computation intensive kernels in hardware. To identify which kernels of the XviD decoder are most likely to gain from an implementation in hardware, the XviD decoder was profiled. For the profiling the tools available in GCC where used. As a result of the profiling two computation intensive kernels where identified. In Table 2.1 the profiling results for these two kernels are depicted.

Both, the colorspace conversion, as the IDCT will be accelerated using hardware modules. The implementation of the IDCT accelerator is presented in [14]. This thesis will focus on the acceleration of the colorspace conversion. In [14] a more detailed discussion of the different decoders, the selection criteria for these decoders and profiling of the XviD decoder can be found.

Building the DAMP SDK

In this chapter the environment that is build to run the XviD decoder without an operating system is explained in more depth. This enviroment is called the *DAMP Software Development Kit* (DAMP SDK for short). Because no operating system is available, all system calls need to be handled, e.g. by a runtime library. In some cases the application needs to be modified to avoid using system calls, which are hard to implement. As stated in Section 2.1.1 Newlib was chosen as library for the standalone environment, because of its portability. In this chapter the porting of Newlib and the working of the standalone environment is presented.

This chapter is orginized as follows. In Section 3.1 is explained how Newlib can be used in a standalone environment. The software that was added to Newlib is also covered in this section. The initialization procedure of the environment will be explained in Section 3.2. Finally, the process of building an application with the DAMP SDK is shown in Section 3.3.

3.1 Porting Newlib to DAMP

The Newlib library is an ANSI C library, that is intended for use on embedded systems [23]. The library can easily be ported, because the system dependent operations are grouped into 17 function calls. These function calls are referred to as stubs. Only these system dependent stubs need to be hooked to the function that implements their functionality on the specific platform. Certain stubs do not need to be implemented, if their functionality is not required by the application. The stubs need to be linked together with the application and the library. Because the stubs do not have to be compiled into the toolchain, it is possible to use the same toolchain for compiling to different runtime environments. So can for example the toolchain be used to compile programs for when the platform is running a monitor program, or when the platform is running a realtime operating system.

The Newlib code is re-entrant, meaning that Newlib will work in a multithreaded environment. When the Newlib library is used in a multithreaded environment, the stubs also need to be re-entrant. For the DAMP SDK multithreading will not be used because there is no operating system that implements this feature. Therefore, the non-reentrant stubs are used.

3.1.1 The Newlib stubs

The Newlib stubs are the interface of the library to the underlying environment. For the DAMP SDK not all stubs have to be implemented. Therefore the default behavior for

stub	description
<code>_open</code>	Open a file.
<code>_read</code>	Read from a file.
<code>_write</code>	Write to a file.
<code>_close</code>	Close a file.
<code>isatty</code>	Query whether output stream is a terminal.
<code>_sbrk</code>	Increase program data space (used by <code>malloc</code>).
<code>_times</code>	Timing information for current process.
<code>_gettimeofday</code>	Returns the current time.

Table 3.1: The Newlib library stubs implemented for the DAMP SDK.

these stubs is implemented, as given by the Newlib documentation. Often this behavior is simply the return of a standard value. An example for the implementation of the DAMP SDK is the `_fork` stub. This stub is in a POSIX environment used to create a clone of the current processing context. For the DAMP SDK the `fork` command has no meaning, because the DAMP SDK does not support multiple processes. Therefore the stub does not need an implementation. When the `fork` stub is called, the value `-1` is returned to indicate an error. Of course, an application that utilizes library functions that make use of this stub is not likely to work. In Table 3.1 the stubs that have been implemented for the DAMP SDK are given.

The `_open`, `_read`, `_write` and `_close` stubs are used for handling of files. Also the writing and reading to the terminal is done via the `_read` and `_write` stubs. From the point of view of the library the terminal is a special kind of file. When the library functions are used that read or write characters from or to a terminal (for example `printf`), the `_read` and `_write` stubs need to be implemented. The `isatty` stub is used to verify if a file represents a terminal. The `_open` and `_close` stubs need to be implemented, when besides a terminal also a real filesystem is implemented. For the DAMP SDK a filesystem has been developed. This will more deeply be covered in Section 3.1.2.

The `_sbrk` stub is needed when `malloc` and its related functions are used in the application. The `_sbrk` stub is used for the heap space management. When the application is low on dynamic allocated memory the `_sbrk` function is called to increase the heap space. In this stub it is wise to detect if no stack collision occurs by enlarging the heap space. When in the DAMP SDK a stack collision occurs, a message is sent to the terminal and the application is aborted.

The `_times` and `_gettimeofday` stubs provide timing information for the time related functions of the library. These functions require a time reference in order to return a meaningful result. For the time reference one of the timers of the Excalibur device is used. This timer is initialized during the initialization of the environment to count with a resolution of one millisecond. Because the timer has a word size of 32 bits, it will overflow after 50 days of operation. Because the DAMP board is only used in a laboratory environment, this does not form a restriction. Because the timer is not synchronized with a clock on the outside world, no absolute time reference is available. Even though the name of the function `_gettimeofday` may suggest otherwise. The

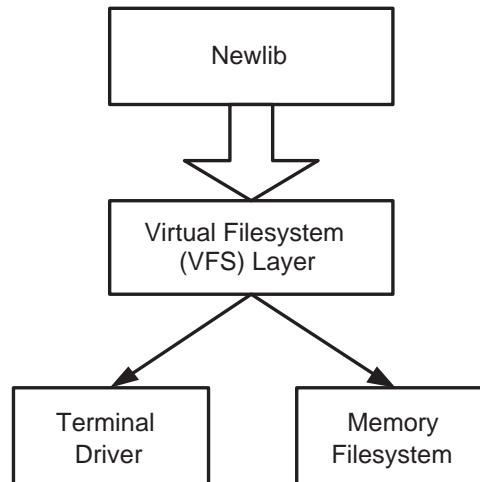


Figure 3.1: The layered structure of the filesystem implementation of the DAMP SDK.

time functions in the library therefore can only be used for time differences, which in the case of the XviD decoder is not a problem.

3.1.2 Adding filesystem capabilities

Most applications make use of files. For the decoder used in SMOKE this is not different. The decoder uses an input file that contains the encoded videostream. Furthermore the decoder is capable of saving the decoded image as separate files. In order to run the decoder as a standalone application without a large number of modifications, effort has been made to implement a filesystem layer.

The implementation of the filesystem is based on a layered architecture. The layers introduce an abstraction that makes the filesystem implementation more flexible. From the point of view of the library a single filesystem is accessed, while in fact the filesystem can consist of multiple small filesystems and hardware drivers. The smaller filesystems can all use their own methods and hardware for storing files. For the upper abstraction layer they only need to implement a predefined set of functions. The current implementation of the abstraction layer supports a single (root) filesystem and a number of device drivers only. The model of the filesystem can be extended to support the connection of multiple filesystem (the so called mounting in Unix environments). In Figure 3.1 the layered filesystem architecture is shown. In this figure a filesystem that stores files in the memory and a hardware driver that reads and writes to the terminal are shown. The same abstraction (in a more advanced form) for accessing filesystems can be found in many operating systems, e.g. Linux.

The layer that hides the details of the physical filesystem implementation for the library, is the *Virtual FileSystem* (VFS) layer. The VFS layer implements the four stubs of the Newlib library that are related to files. These stubs where `_open`, `_read`, `_write` and `_close`. A number of extra stubs for filesystem management are available and will

need an implementation when the application requires more advanced filesystem support. Examples are the `_link` and `_unlink` stubs for creating or removing a reference to a file. The connection of the VFS to the library will be called the frontend, while the backend is the interface to the filesystem implementation, or the filesystem driver.

The frontend uses a path to the file or a filedescriptor to identify the files. The path is a string containing the hierarchy and the filename within the filesystem. The path is used for the `_open`, `_link` and `_unlink` stubs. The other stubs use a filedescriptor instead of the path to identify a file. The filedescriptor is a number, that is assigned to a file after it is opened. The VFS maintains a list with open files. The assignment of a filedescriptor to a file is performed dynamically. So when a file is closed and opened again it will probably have a different value for its filedescriptor. Some filedescriptors however are reserved for a special purpose. The `stdout` that is used by the library for writing to the terminal, is an example of such a file with a special filedescriptor. The library expects the `stdout` filedescriptor to have a value of one.

The backend of the VFS is connected dynamically to the filesystem driver at runtime. The dynamic connection is created by the use of function pointers. The VFS maintains for every filedescriptor a set functionpointers that implement the file operations (for example a pointer to the `read` or `write` procedure) for the specific file. For the operations that do not operate on an open file, and therefore have no entry in the filedescriptor list, the operation is passed to the rootfilesystem driver. The rootfilesystem is registered during the initialization of the VFS. Because the routines are dispatched to the rootfilesystem, they will not function for files that are placed on another filesystem than the rootfilesystem. This is also the reason why the current implementation does not support multiple filesystems. This restriction could easily be resolved by dispatching the calls to the corresponding filesystemdriver based on the path that is passed as an argument. For the SMOKE project multiple filesystems are not relevant and therefore not implemented.

The following two functions are used to create the dynamic connection between the VFS and the implementing filesystem drivers.

```
int vfs_register_fd(int fd, file_ops_t *fops);
void vfs_register_root_fs(file_ops_t *fops);
```

The `vfs_register_root_fs` function installs the default routines of the rootfilesystem. These routines are used for the stubs that do not operate on a open file. The argument of the `vfs_register_root_fs` is a structure containing pointers to the file operations of the rootfilesystem. The `vfs_register_fd` creates an entry in the filedescriptor list. This function is used for creating the entries for the `stdin`, `stdout` and `stderr`. These filestreams are not opened by the library, but are expected to be already available. They are therefore created during the initialization of the environment. The first argument of the `vfs_register_fd` function is the filedescriptor number. This should for example be one for the `stdin`. The second argument is a structure with pointers to the file operations that can be performed on this file.

The following operations are performed when the file stubs are called. First a file have to be opened. Therefore the `_open` stub is called. This stub calls the `vfs_open` function, which is part of the VFS abstraction layer. The `vfs_open` function will create a new entry in the filedescriptor list and call the `open` function of the rootfilesystem,

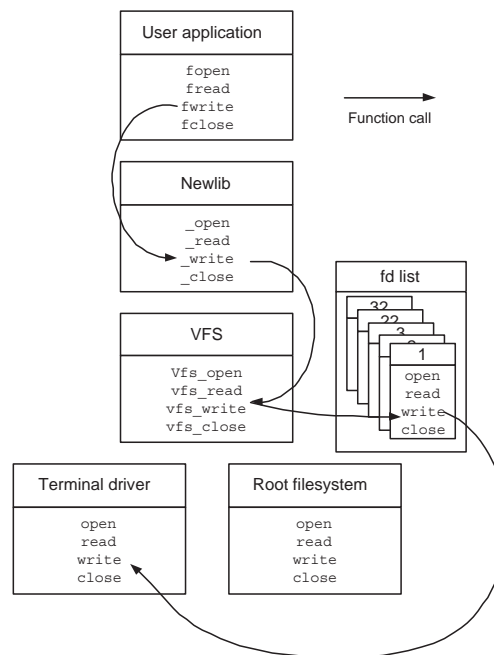


Figure 3.2: The call path when the *fwrite* function is called.

to connect the right file operations to this filedescriptor entry. Finally the `_open` stub returns the filedescriptor value to the calling library function. When the file is opened the `_read`, `_write` and `_close` operations can be performed on this open file. When for example the `_read` stub is called, the `vfs_read` function will search the filedescriptor list for the corresponding filedescriptor (passed as an argument to the `_read` stub). When the entry is found the corresponding function is called that is in the list of functions. In this case the pointer to the read operation of the file will be used. The `_write` and `_close` are handled the same way. The only difference for the `_close` call is, that after the `close` function in the pointer is called, the entry in the filedescriptor list is removed. In Figure 3.2 the described process for the *write* function is graphically explained.

For DAMP the following two filesystem drivers have been implemented. The first is the terminal driver that implements the routines for the *stdin*, *stdout* and *stderr* filestreams. The second filesystem driver implements a memory filesystem. This memory filesystems can reserve a part of the memory to store files. In Appendix F, the sources of the VFS, the terminal driver and the memoryfilesystem can be found. The filesystem has been tested on a PC platform and did operate as expected. Unfortunately, the memory filesystem has not worked on DAMP. The terminal, however, does operate correctly using the VFS implementation. Due to time limitations it was not possible to solve the problems with the memoryfilesystem on DAMP.

3.2 Initializing the environment

Before the application can be started in the standalone environment, the hardware and software environments need to be initialized. Most of the hardware is already initialized by the bootloader, but for example the instruction cache and the timer that is used as time reference need to be enabled when the SDK is started. Also the stackpointer that is needed by the C environment requires initialization.

3.2.1 Enabling the MMU

The ARM 922T processor has data and instruction caches. Both are disabled by default after a system reset. The instruction cache can be enabled without any dependencies. The *Memory Management Unit* (MMU) is to be enabled, before the data cache can be activated. This memory management unit is used to translate virtual addresses to physical addresses and to apply security policies to the memory management. The virtual addressing and security policies are used by operating systems to prevent faulty processes to destabilize the entire system. For an embedded environment that is only running a single application without an operating system, the MMU is not really needed. Nevertheless, the MMU is required, when the data cache is used. Therefore the MMU will be enabled, with the virtual addresses that map to the same physical addresses and no restriction to memory access.

The MMU uses a table in the main memory to store the address mappings and security policies. The MMU divides the memory in a number of blocks that share the same mapping and security policy. So for example a block of memory with virtual addresses ranging from 0_{HEX} to $FFFF_{HEX}$ will map to a physical range from $F0000_{HEX}$ to $F0FFFF_{HEX}$. The MMU supports different block sizes to divide the memory. For the DAMP SDK the largest block size is used because this will give the least overhead and is less complicated to implement.

The code for enabling the MMU was obtained from an Altera reference design. This code was modified to setup the memory map for the SDRAM instead for the SRAM. In the same file also the C environment is initialized which will be explained in more depth in Section 3.2.3. After the MMU is initialized the data cache can be enabled.

3.2.2 Installing the interrupt table

The DAMP SDK is started by the bootloader. This bootloader is placed at address 0_{HEX} of the memory. The bootloader also installs the interrupt vector table. Because the ARMboot code cannot be overwritten during the loading of the SDK, it is not possible for the SDK to be loaded at the beginning of the memory. This is no problem for the code, because the linker can be instructed to add an offset to the address of the programcode. However, for the interrupt table this introduces a problem. The interrupt table must be placed at address 0_{HEX} . The application will not be able to respond to interrupts, if it does not place the right interrupt table at address 0_{HEX} .

There are several solutions to solve this problem. The first solution is to replace the interrupt vector table of the bootloader during the initialization of the application. This

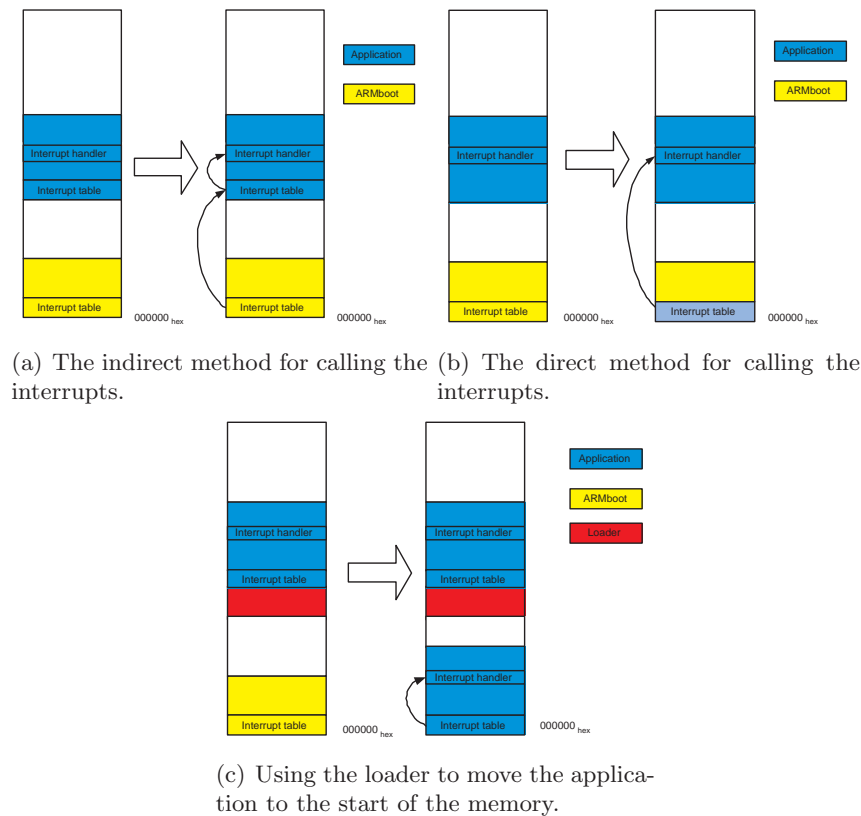


Figure 3.3: The different methods for installing the interrupt vector table.

table can be replaced with a table that jumps to the interrupt table that is together with the application placed at an offset in the memory, or with a table that directly jumps to the application interrupt handlers. The second solution is to use a *loader*, that copies the application to the start of the memory and executes it. The advantage of the last solution is that the application does not need to be aware of the fact that it is loaded by a bootloader. The application has full control of the environment. In Figure 3.3 the different solutions are graphically presented. Even though the last solution requires the copying of the entire application to a new location in the memory (which of course takes time) this solution was chosen, because the application itself does not need to be modified to solve the problem with the interrupt table.

In order to copy the application to the start of the memory, a *loader* was developed. The *loader* performs two simple tasks. First it copies the application to the start of the memory. Then it starts the application that it has just copied by jumping to address zero (the reset interrupt vector). The binary of the application is embedded in the *loader's* executable. In order to link the application binary into the *loader* executable, a special tool, called `bin2c`, was developed. This tool generates a C file from a plain binary file. The C file contains a constant character array that contains the data of the binary file. The generated C file is compiled and linked with the *loader*.

3.2.3 Setting up the C environment

The applications that are build with the DAMP SDK are written in C. Before an application can be executed, the library and the stackpointer are initialized. The stackpointer have to be set to the end of the memory. Because the processor has different modes (for example a supervisor and user mode), the stackpointers need to be set for these different operation modes. After the stackpointers are initialized a call is made to the initialization function of the library stubs. This initialization function is called `initialise_lib`. It enables the timer that is used as clock reference and initializes the filesystem.

3.2.4 Starting the application

After the C environment is initialized the `initialize_environment` function is called. This function first configures the FPGA with the configuration data that is included in the executable. The configuration is obtained from a SBI file that is generated with the Quartus environment. To include the configuration data the same method is used as for the the *loader* application (see Section 3.2.2), with the only difference that instead of using the `bin2c` tool a tool called `sbi2c` (also specially written for the DAMP SDK) is used. This tool performs some sanity checks on the input file and also places some additional constants in the C file that contain information about the configuration file.

After the FPGA is configured the `main` function of the application can be called. If needed the main function can be called with the required arguments. When the main application returns the return code is displayed and the environment enters an endless loop. In future implementation of the DAMP SDK a simple shell like behavior could be created in order to load the main application and pass arguments to it.

3.3 Building standalone applications

The DAMP SDK consists of a number of configuration files and needs to be build in a number of steps. In order to facilitate this process a makefile was created. This makefile first compiles the needed tools (`bin2c` and `sbi2c`), next it compiles the separate sources and in the process the SBI file is converted to a C file. After all the sources are compiled the executable is linked. The obtained object file is converted by a binary file that can be executed on the DAMP platform. This binary file is converted with the `bin2c` and compiled into the *loader*. The *loader* is linked with an offset so it can be loaded behind the bootloader in the memory. The final step in the build process is to convert the *loader* into a binary file.

When the entire build process is finished the binary image of the *loader* can be loaded via BOOTP in the DAMP board with keeping the linked offset in mind. The load process to the DAMP board is performed by the bootloader. After the application is loaded to the memory of the board, the bootloader jumps to the first instruction of the *loader*.

One of the environments in which the decoder will run, is the Linux operating system. The Linux kernel needs to be build for the DAMP platform. The process of porting and building the kernel is explained in Section 4.1. Because no port was available for DAMP, several small modifications of the kernel sources where needed. These modifications are stated in Section 4.2. After the kernel is built, the right kernel arguments and a filesystem are needed in order to boot the kernel on the target platform. In Section 4.3 the booting of the kernel is elaborated.

4.1 Configuring and building the kernel

Linux is becoming an increasingly popular operating system for desktop PC's, but it is also gaining popularity for embedded systems. One of the reasons is the fact that it can be used at no cost. The fact that Linux is open source, makes it possible to modify it for use on any non standard platform. There is a large community supporting the Linux kernel development. Within this community, several subcommunities focus on different target platform. These subcommunities contribute patches to the main kernel development line to add support for specific platforms. It is possible that these patches are merged with the main development line in later versions.

The first step in porting the Linux kernel to an non standard platform is finding an already supported platform that has the most similarities with the new platform [35]. This platform is usually referred as *machine* or *sub-architecture*. The architecture is the CPU-type or category in which the system design can be placed. The architecture in the case of DAMP is ARM. The ARM processor is well supported by the Linux kernel. A special community focusses on the ARM development line for the Linux kernel [29]. The patch available at this community has support for the EPXA1DB and EPXA10DB development boards from Altera. The sources and configuration files for the EPXA1DB will be used as a foundation for the DAMP port. The EPXA1DB board contains an EPXA1 Excalibur chip and therefore is closest to the DAMP platform.

Before the kernel can be configured for the EPXA1DP platform the kernel's main makefile is to be modified. The variables `ARCH` and `CROSS-COMPILE` need to be set to `ARM` and `arm-linux-` respectively. Setting the `ARCH` variable ensures that the target platform will be ARM even when it is compiled on another architecture. When compiling on another than the target architecture a cross-compiler is needed. In the case of the SMOKE project a PC (i386) running RedHat Linux 8.0 was used as development platform. Therefore also the `CROSS-COMPILE` variable needed to be set to the used cross-toolchain `arm-linux-gcc` (version 2.95.3).

Next, the kernel is configured to compile for the EPXA1DB development board by running the following commands in the root of the kernel source directory:

```
make epxa1db_config
make oldconfig
```

The first command configures the kernel sources by making the right symbolic links and by installing the machine's default configuration file. The last command makes sure there are no configuration variables left uninitialized. The latter could happen when new settings for the kernel have been added since the machine configuration file has been submitted. In case of uninitialized variables it is usually safe to set them to *No* [21]. The default configuration for the EPXA1DB was modified. Modifying the kernel configuration can be performed by running the following command:

```
make menuconfig
```

This presents a menu structured interface to configure the kernel. The following changes in the configuration have been made:

- Both data and instruction cache where enabled.
- The driver for the Cirrus CS9800A Ethernet chip was enabled (the driver for the SMC91111 network chip was disabled).
- The NFS driver was enabled.
- Support for using NFS as a root filesystem was enabled.
- Extensive kernel debugging information was enabled.

Besides the changes mentioned above, several small changes where made to add some additional features to the kernel which in the end where not used.

The final steps are to compile and build the kernel. Actually, before compiling and building the kernel some of the kernel sources need to be modified to support the DAMP platform. These modifications are presented in Section 4.2. After the modifications the following commands are needed for for the compilation and building of the kernel.

```
make clean
make dep
make zImage
make modules
```

The last command (`make modules`) is not needed, because all drivers are statically linked into the kernel, instead of dynamic loaded at run-time. During the configuration it is possible to select if parts are dynamically loaded as module or statically linked into the kernel. The advantage of modules is that they can be replaced without recompiling the entire kernel and they keep the kernel image small. Also the hardware configuration of a platform does not need to be known during kernel compilation, because support for this hardware can be loaded when the kernel is booted. This is method used for many PC (i386) distributions available. On the other hand there is an disadvantage of using modules. Modules can not be used early in the boot process, because there is still no access to the root filesystem where these modules reside. When for example

support for the root filesystem is available by the use of a module there is conflict. The root filesystem is needed to load the module, but the module is needed to access the root filesystem. To overcome this restriction a special construction can be used. It is possible to instruct the kernel to load a special kind of ramdisk, called `initrd` [1]. This `initrd` is loaded very early in the boot process, making it possible to load modules into the kernel that are needed to continue the boot process. But of course statically linking the drivers into the kernel, is in this case more practical. Especially when the necessary drivers are known at compile time, the size of the kernel image does not really matter and the drivers are not likely to be updated in the near future. This is the case for DAMP and therefore the support for e.g. NFS and the ethernet controller are statically compiled into the kernel.

When the compilation process is successful a compressed kernel image (called a `zImage`) can be found in the `arch/arm/boot` directory relative to the root directory of the kernel source. This `zImage` consist the compressed kernel and code to extract the compressed kernel and execute it afterward. The `zImage` need to be executed by the bootloader. The bootloader requires the `zImage` to be packed into an image with a special header that contains an identifier (in order to recognize the image as kernel image) and a checksum to verify if the `zImage` is not corrupt. A special tool, called `mkimage`, for adding this header to the `zImage` is included in the distribution of ARMboot.

The kernel image is placed after the bootloader in the flash. One of the main reasons for placing the kernel into the flash memory instead of loading it via BOOTP¹ (which is also supported by ARMboot) was that support for the ethernet controller was added later on to the DAMP board during the SMOKE project. In the meantime the board had only flash available as non-volatile storage medium. For placing the kernel image into the flash after the bootloader it needs to be converted to a hex file with a start address shifted. A special script was written to call the tools to convert the kernel image in a hexfile containing the bootable image needed by ARMboot with a shifted start address. The generated hexfile is programmed into the flash memory through the JTAG interface using the `exc_flash_programmer` tool part of the Quartus development environment.

4.2 Modifications to the kernel sources

As mentioned in Section 4.1, small modifications to the kernel sources where needed to support the DAMP platform. Even though most modifications seem trivial, they require a deeper understanding of the Linux kernel internals.

4.2.1 DAMP specific modifications

An important modification to the kernel source is to replace the headerfile containing all the configuration settings of the Excalibur. This configuration file is automatically

¹*BOOTstrap Protocol* (BOOTP) which allows a diskless client machine to discover its own IP address, the address of a server host, and the name of a file to be loaded into memory and executed [13].

generated when the *MegaWizard Plug-in Manager* (which is part of the Quartus development environment) for the Excalibur is used to configure the chip. Examples of configuration settings are the clockspeed of the ARM processor, the memory mapping, the SDRAM clockspeed and more. When the Quartus environment is also used for compiling programs for Excalibur, a bootloader is added to the code to initialize the Excalibur to reflect the settings made by the MegaWizard. A headerfile is also generated by the MegaWizard, to use the configuration settings in the C applications that run on the Excalibur chip.

The Linux kernel needs to be aware of the configuration of the Excalibur device. It for example needs to know the location of the registers, the clockspeed of the ARM processor, etc. The headerfile that is generated by the MegaWizard is used to obtain all these settings. The kernel will not work properly, if the configuration that is resembled by this headerfile does not resemble the configuration that has been set by the bootloader (ARMboot). The latter was the case for the initial Linux version for DAMP, resulting in the faulty operation of the UART interface. The baudrate for the UART is derived from the processors clockspeed. Because the headerfile containing the configuration settings used by Linux was outdated, the wrong clockspeed was used to derive the value for the baudrate division register. In order to prevent problems in the future, a symbolic link was made to the headerfile containing the configuration data used by ARMboot. When the configuration of the Excalibur device is changed, the headerfile in the ARMboot sources will be updated to resemble the new configuration. When the kernel is recompiled the updated headerfile will automatically be used.

4.2.2 Setting up the memory map

As many operating systems Linux uses virtual memory addressing. One of the advantages of virtual memory addressing is that the operating system is independent of the physical memory layout.

As a result of this virtual addressing it is not possible to reference memory mapped hardware devices by their physical addresses. Instead the virtual addresses that translate to these physical addresses should be used. Of course these virtual addresses need to be known by the drivers that use this hardware. Also the kernel needs to know that these memory mapped devices are available and it need to make table entries that map the virtual addresses to their physical counterparts. Therefore it is possible to setup a memory map during the initialization of the kernel.

There are several memory mapped hardware modules that need to be available to the kernel or the device drivers. The control registers of the Excalibur device where already mapped by the EPXA1DB port of the kernel. The following devices have been added to this mapping:

- The control registers of the ethernet controller.
- The SDRAM and DPRAM of the Excalibur (which are used as an interface to the PLD hardware).
- The AMBA slave devices that reside in the PLD.

The kernel has two methods for mapping a physical memory range to a virtual range. The first method creates a static mapping. This means that a virtual address range is always mapped to a certain physical address range. The static mapping is created during the initialization of the kernel. The second method creates dynamic mapping. A call to the function `io_remap`, with the physical memory range as argument, returns a pointer to the begin of the virtual memory range. The dynamic method can be used in device drivers to map regions only when they are needed. The static mapping always creates the mapping, even when it is not used. The static method is used for the memory map for DAMP, because this method was also used by the EPXA1DB port.

The static mapping can be made by modifying the machine dependent file² `mm.c`. In this file the machine dependent memory map is initialized by calling the function `iortable_init`. The argument of this function is a structure containing a description of the mappings that need to be made.

4.2.3 Modifying the ethernet driver

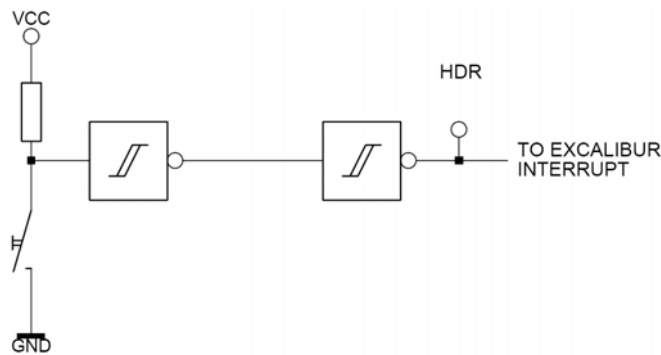
To add networking support to Linux several problems needed to be resolved. A few were the result of the imperfections of the DAMP board. These problems with the hardware even required some physical modifications to DAMP as explained in Section 4.2.3.1. Also some software modifications need to be made to the Linux driver of the ethernet controller. These modifications are explained in Section 4.2.3.2.

4.2.3.1 Hardware modifications to DAMP for ethernet support

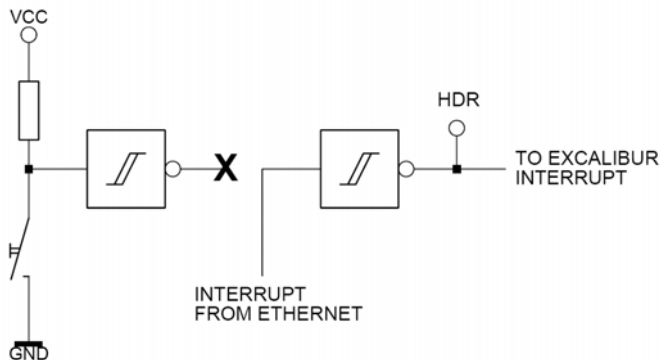
The current version of DAMP, that is used for the SMOKE project, contains a few imperfections. One of these imperfections is the too small footprint on the PCB for the network controller. The controller is therefore connected with wires to the extension headers of the DAMP board.

Besides the too small footprint, there is another problem when using the ethernet controller together with Linux. The available Linux driver for the ethernet chip assumes the controller operates in an interrupt driven mode. Unfortunately on DAMP the interrupt pin of the ethernet controller is not connected. Even worse is the fact that the only external interrupt of Excalibur is occupied by one of the user I/O pushbuttons. Fortunately, the pushbuttons are not needed during the SMOKE project, so the interrupt can be used for another task. However, assigning the interrupt to another source (in this case the ethernet controller) introduces some difficulties. Even though the interrupt is available on a extension header pin, it can not directly be forced by another source. This is because the signal of the pushbutton is fed through two schmitt-triggers to obtain a well defined voltage level for the digital circuitry. The current interrupt circuitry is depicted in Figure 4.1(a). The schmitt-trigger places a hard low or high value on the interrupt line. Connecting another source to this interrupt line via the extension header pin would result in a short circuit and could damage one or both of

²All machine depended C files can be found in the `arch/arm/mach-epxa` directory. For other machines these files can be found in the `arch/arm/mach-XXX`, where XXX stands for the machine name that is used.



(a) The signal path of the external interrupt



(b) The modified signal path of the external interrupt

Figure 4.1: The signal path of the external interrupt before and after modification.

the sources. In this case the only solution is to physically disconnect the schmitt-trigger from the interrupt line.

Another option for connecting the interrupt of the ethernet controller to the Excalibur, is to use the interrupts available in the reconfigurable hardware of Excalibur. One of these interrupts could then be routed to one of the available I/O pins. Unfortunately this would imply the restriction that the ethernet controller can only be used when the reconfigurable hardware is configured to support this. Therefore it was decided to physically disconnect one of the schmitt-triggers. The external interrupt was connected from the conflicting source. Actually the interrupt of the ethernet controller was connected between the two schmitt-triggers, because the ethernet interrupt signal needed to be inverted. This because the interrupt of the ethernet controller is active high, while the external interrupt of the Excalibur device is active low. In Figure 4.1(b) the modified interrupt circuitry is depicted.

A quite different approach for solving the problem with the interrupt, is to modify the ethernet driver to use polling instead of an interrupt. Polling is also used by the ARMboot. Unfortunately polling is rather inefficient. The ethernet controller is accessed by the CPU, even when there has been no network activity. For ARMboot the speed is not an issue, because it can only perform one task at the time. When it

is transferring a file, the CPU has to wait until the transfer is completed. Therefore it does not matter if polling is used. This is quite different when running an operating system like Linux. Besides networking, the operating system has to perform a large number of other tasks and polling a network driver would slow down the entire system unnecessary. For this reason the solution to modify the Linux network driver to support polling instead of interrupts was abandoned.

4.2.3.2 Modifications to the ethernet driver for Linux

Two type of modifications to the ethernet driver for Linux needed to be made. The first type of modifications are related to system depended variables like the base address³ and irq number. The second type of modifications where related to the platform differences. The driver was probably made for an i386 architecture, which has some differences in how devices are connected to the processor in comparison to the ARM architecture.

For the first category three modifications needed to be made. First, the MAC address is added. On DAMP, there is no eeprom connected to the ethernet controller for storing its configuration (like a MAC address). The ethernet controller therefore needs to be configured by the processor during the initialization. The MAC address used for DAMP is the one that is also used by the LART [25] boards. This is no problem because DAMP is connected to a local network behind a router. When more DAMP boards are build, it would be wise to use a MAC addresses of some old ISA network card that is not used any longer.

The other modifications in this category are related to the resources used by the ethernet controller. Two constants need to be set to the right values. These constants are `CIRRUS_DEFAULT_IO` and `CIRRUS_DEFAULT_IRQ`. The first variable is related to the mapping of the control register of the ethernet controller in the system memory. In Section 4.2.2 is already explained how the physical address range of the ethernet controller was mapped to a virtual address range. For the ethernet controller the start of the address range, is not the same as the base address used in the driver. This because the ethernet control registers are placed with an default offset of `300HEX` relative to its base address. This value is therefore added to start of the address range (defined as `ETHER_BASE`) in the kernel sources performing the memory mapping.

The second variable (`CIRRUS_DEFAULT_IRQ`) contains the IRQ number to which the ethernet controller is connected. Excalibur has an interrupt controller with different operating modes [10]. For SMOKE, the default mode is sufficient. In this mode all available interrupts are represented by a priority register and several status and masking bits. The values stored in the priority register determine which one of the interrupts is handled first when more than one interrupt is asserted. The priority value is also used to identify the current active interrupt. This value can be read from the `INT_ID` register. Via this register the interrupt routine can quickly identify which interrupt it needs to handle. The Linux implementation for the EPXA1DB board places during the initialization up going priority values in the priority registers. These priority numbers are the same numbers used in Linux to identify the interrupt. So because the counting

³The *base address* of a device is the first memory address on which the control registers are mapped.

of the initialization starts at zero and the external interrupt is the seventh priority register the interrupt used in Linux is six. Therefore the `CIRRUS_DEFAULT_IRQ` is set to six. Fortunately the interrupt number of the external pin is already defined by the constant `IRQ_EXT` and is used instead of six for clarity and portability. The definitions of both variables are stated below.

```
#define CIRRUS_DEFAULT_IO ETHER_BASE + 0x300
#define CIRRUS_DEFAULT_IRQ IRQ_EXT
```

All the mentioned modifications were related to system dependent variables in the driver source. The second type of modifications required changes of the ethernet driver source. This due to the differences in the platform for which the driver was written. The original targeting platform was probably an i386. On the i386 a special bus for IO devices is available. On the ARM, no such dedicated IO bus is available. Here all IO devices are connected to the memory bus. For Linux, these differences are visible for device drivers. Different functions are used to operate on the IO bus or on the memory bus. For the ethernet driver, this is the case for the `check_io_region` and the `request_io_region` functions. These functions are used for resource management. They prevent multiple drivers accessing the same memory locations⁴. Because the ARM architecture has no special IO bus these functions make no sense and therefore will always fail. The ethernet controller on DAMP is connected to the memory bus and therefore the counterparts of the `check_io_region` and `request_io_region` need to be used. These are `check_mem_region` and `request_mem_region` and they are called with the same arguments. Fortunately, besides the different functions for the IO connected or memory mapped devices no additional modifications to the ethernet driver source were required.

4.3 Booting the kernel on DAMP

Besides a kernel image, a root filesystem is needed in order to boot the operating system and run applications. For DAMP, there are several possibilities where the root filesystem could be located. These are the flash memory, the SDRAM or a remote network share. Both, the network share and the flash memory were implemented as root filesystem location. The SDRAM on the other hand is not used, because of its volatile nature. The SDRAM would require a filesystem image reload after system power down. Also the modifications to the files would be lost after each power down.

Besides the location, some basic content of the root filesystem is needed. Furthermore a commandline argument to the kernel needs to be given to adapt the kernel to the filesystem used. The settings and modifications for both the flash filesystem and the network filesystem are presented in section Section 4.3.1 and Section 4.3.2 respectively.

⁴It needs to be noted, that it is the responsibility of the driver developer to check (via the function `check_io_region`) if an region is available. The kernel has no way of preventing a driver from accessing regions that are already claimed by other drivers.

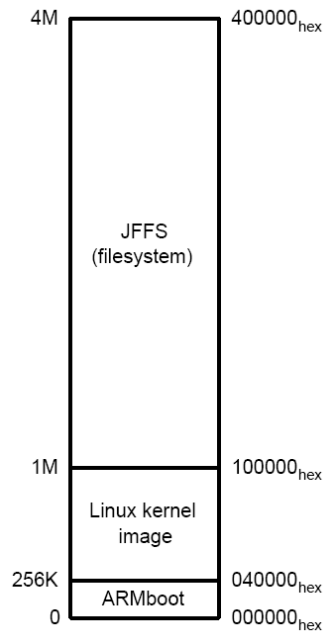


Figure 4.2: The layout of the flash memory

4.3.1 Booting from the flash filesystem

The first and most preferable location for the root filesystem is the flash memory. When using the flash memory, Linux can boot stand alone (without a network connection). Unfortunately the flash memory is limited to four megabytes in the current DAMP revision. One megabyte is used for the bootloader and the kernel image. In the other three megabytes the root filesystem can be placed. In Figure 4.2 a layout of the flash memory is given. The three megabytes form a large restriction on the number of applications and files that can be stored. Therefore other alternatives besides the flash memory where explored.

A special kernel subsystem is used to access the flash memory. This subsystem is called MTD (which stands for Memory Technology Device) [17] and is developed for memory devices, especially flash memories. Before the free available flash memory can be accessed by the driver a device map (containing the physical memory locations of the MTD devices) is needed. After modifying the device map, the kernel needs to be recompiled, because the driver is statically linked into the kernel. The device map can be found in the *driver/mtd/map/epxa-flash.c* file relative to the root of the kernel source.

Besides a root filesystem location and a driver to access it, a driver for the filesystem type is needed. In the case for the flash memory, JFFS (Journaling Flash File System) is used. This filesystem is specially adapted to the properties of flash memories. For example, JFFS takes into account the limited erase-cycles of a flash memory. It therefore tries to equally divide the erase cycles over the different flash memory segments⁵, even

⁵Flash memories are divided in a number of segments. These segments form the basic quantities in which the flash memory can be erased. Even when a few bytes in a segments need to be changed, the

when the same file is continuously modified. This process is called *wear levelling* [34].

After the filesystem is available to the kernel, the correct commandline options need to be given to the kernel to use this filesystem as root filesystem. Furthermore, the filesystem has to contain a basic set of device nodes⁶, configuration files and applications. For the filesystem that was placed in the flash memory a distribution from Altera was used as foundation. Due to the limited size of the flash memory, only a small number of applications could be selected. The files needed to boot the kernel are the device nodes for the storage medium hardware (in this case the MTD driver) and the terminal for user I/O. Furthermore, a shell like *bash* is needed.

The following commandline is used to boot the kernel with the flash memory as root filesystem.

```
console=ttyUA0,115200 mem=240M root=/dev/mtdblock0 init=/bin/bash
```

The `console` argument tells the kernel that user I/O goes through the UART, which is controlled by the `ttyUA0` driver. The baudrate for the UART is set to 115200 baud per second. The `mem` argument is used to set the available main memory. This value is lower than the true available memory (256MB), because the kernel crashed at higher values. It is unclear why the kernel crashes, but due to time limitation the problem has not been further investigated. The root filesystem can be set with the `root` argument. Finally the argument `init` tells the kernel that it needs to start `/bin/bash` instead of the default application `init`, after the root filesystem is mounted.

4.3.2 Booting from the network filesystem

The other location where the root filesystem can be placed is a network share. The network share is accessed using NFS (Network File System). Even though the network share requires the DAMP board to be connected to a network, it has the advantage that it makes the storage capacity of a standard desktop PC available to the DAMP board. The network filesystem therefore overcomes the main disadvantage of using a flash filesystem on the current revision of the DAMP board.

In order to use NFS as root filesystem, support for using NFS shares as root filesystem needs to be compiled into the kernel as explained in Section 4.1. Furthermore, the network driver needs to be statically linked into the kernel as discussed in section Section 4.2.3.2. On the host machine (that is running an NFS server) the filesystem content is placed. For this filesystem the Montavista Hard Hat 2.0 Linux distribution was used. This distribution is available as a set of RPM packages. The required files were extracted from these packages with a tool called *alien*. A script was written to automate the extraction process.

For booting from the network filesystem two more arguments are added to the commandline, that was used for booting from the flash memory filesystem. The commandline that was used to boot Linux with NFS as root filesystem is stated below.

entire segment is erased and re-written.

⁶Device nodes are special files that in fact give access to the hardware drivers by using the standard filesystem calls.

```
root=/dev/nfs mem=240M console=ttyUA0,115200
nfsroot=192.168.2.1:/tftpboot/%s/filesystem
ip=192.168.2.25:192.168.2.1:192.168.2.1:255.255.255.0:damp:eth0:off
```

The arguments `root`, `nfsroot` and `ip` are needed to enable the NFS support. The `root` argument tells the kernel that the NFS driver needs to be used for accessing the root filesystem. The location where the root filesystem can be found is given by the `nfsroot` argument. Finally, the `ip` arguments states the configuration settings for the ethernet device (like the ip address, gateway, hostname, etc) it needs to use.

4.4 Building hardware drivers

The Linux operating system protects hardware resources against unauthorized access. Furthermore, it gives each process its own runtime environment, to protect the system against the malfunctioning of a single process. The applications can access hardware modules through special software components, called drivers.

4.4.1 The difference between user space and kernel space

In Linux the distinction is made between *user space* and *kernel space*. The user space is the environment in which all applications run. Each user space program has its own address space and runs in the *protected* mode of the processor. The user space applications have limited rights on the platform and therefore have no access to the hardware devices. On the other hand the software that runs in kernel space has full control over the platform, because it runs in the *supervisor* mode of the processor. All the code that runs in kernel space uses the same address space.

To access hardware resources from a user space application, a driver for this resource needs to be available. This driver is part of the kernel and therefore runs in kernel space. To add a driver to the kernel two methods are available. The first method is to statically link the driver into the kernel. The second method uses dynamic linking to load the driver into the kernel at runtime. In this case the driver is called a kernel module. Both options were already discussed in Section 4.1. For drivers that are under development and are likely to change often, the module approach is far more convenient.

4.4.2 Character device drivers

The accelerators for the kernels of the MPEG-4 decoder are implemented in hardware. When the MPEG-4 decoders is running on the Linux platform, it has no direct access to the hardware. This because Linux prevents applications from accessing resources directly. In order to access the hardware accelerators, a driver needs to be implemented. The driver type that will be used for the kernels is the *character device driver*. The character driver behaves, from the point of view of the user application, as a file. The standard file operations like reading and writing can be performed (if they are supported

by the driver).

4.4.2.1 Accessing character drivers from user space

Each device driver is accessed as a file by the user application. For this reason an entry for the device driver needs to be made in the filesystem. These entries can be made with the tool `mknod`. An entry in the filesystem for a device driver just looks like a file and is called a device node. All the operations that are performed on this device node, are passed to the corresponding driver. The kernel needs to know which driver is connected to each device node. Therefore the `mknod` tool takes two arguments that are both numbers. The first number is the *major* device number. The major number is used to identify the driver. The second argument to the `mknod` tool is the *minor* number. This number can be used by the driver to create small variations on its behavior. For example the minor numbers above 128 for a backup type driver, rewind the tape after the device file is closed, while the minor numbers below the 128 leave the tape at its current position after the device file is closed.

Of course a major number must be assigned to the driver, in order to be connected to the corresponding device nodes. Actually, the driver registers itself to a major number when it is loaded into the kernel. Unfortunately, the major, as well as the minor numbers, are limited to 256 values. The free available major numbers are becoming rare. Therefore the use of hard coded values for the major number is strongly discouraged [30]. The kernel provides a dynamic method for obtaining a free major number. Every time the driver is loaded it asks the kernel which major number is available. The number that is returned by the kernel, is used by the driver to register itself. A disadvantage of dynamically assigning a major number to a driver, is that the device nodes for this driver become invalid when a new major number is assigned to the driver. The device nodes for this driver then need to be recreated with the new major number. Hard coded minor numbers do not introduce a conflict with other drivers, because the minor numbers are only used within a single driver.

4.4.2.2 Implementing the file operations

During the initialization of driver, besides the major number, also the routines that handle the file operations for a device node need to be registered. The driver passes a structure with function pointers, that point to the different file operations, to the kernel. This way the kernel can dispatch the required operation to the driver. Not all the operations that can be performed by an application on a file need to be implemented by the driver. The function pointer in the structure can be left uninitialized, if the operation related to this function pointer has no meaning for the hardware. For example the *seek* operation can not be implemented for a terminal driver, because the input and output is sequential.

The drivers that will be used for the hardware accelerators of the XviD decoder are mainly responsible for transferring data between the hardware and the decoder. Three methods have been investigated for implementing these data transfers. The first method

uses the standard *read* and *write* routines. The *write* routine can be used to send data to the driver, while the *read* routine can be used to obtain data from the driver. The second method is using the general *ioctl* function. The *ioctl* function in this project is used for passing a command and a pointer to a buffer to the driver. The last method that was investigated, is the the *mmap* function. This function can be used to map the memory locations of the hardware registers and buffers to the memory space of the user program. The user program can in this case directly write to the hardware memory locations.

Using the read/write routines

The *read* and *write* routines can be used to copy data to or from the hardware. Both functions have similar principle of operation. This translates in the almost equal prototypes for these functions:

```
ssize_t read(struct file *filp, char *buff, size_t count,
             loff_t *offp);
ssize_t write(struct file *filp, const char *buff, size_t
             count, loff_t *offp);
```

The first argument of both functions is a pointer to a file structure. This structure contains information about the open file. The `buff` argument contains a pointer to the user space buffer. For the *read* function, the driver should copy to this buffer, while for the *write* function the driver should read from this buffer. The `count` parameter contains the size of the user space buffer. The last argument (`offp`) passes the offset from where the driver should read or write. This, because the reading or writing does not have to start from the beginning of the file, but can start at arbitrary position.

In most cases that the *read* or *write* functions of a device driver are called, it is required to transfer data between user and kernel space. The copying to and from user space can not be performed by the simple `memcpy` function. The copying between user and kernel space must be performed by the `copy_to_user` and `copy_from_user` functions. These functions copy from kernel to user space, or from user to kernel space respectively. These special functions are needed, because in user space a different address space is used, than in kernel space. In Figure 4.3 the location of the arguments is shown for the *read* function.

Using the ioctl function

The *ioctl* function can be used to pass device specific commands to the driver. The use of the *ioctl* function is very general. The driver function that handles the *ioctl* call, has the following prototype:

```
int ioctl(struct inode *inode, struct file *filp,
          unsigned int cmd, unsigned long arg);
```

The `inode` and `filp` both contain the information about the file. The `cmd` argument is used to pass a command number from the user application to the driver. The

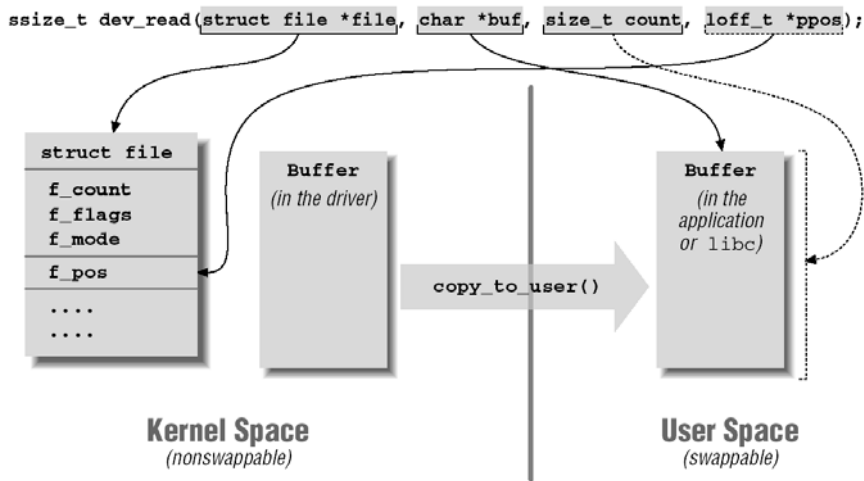


Figure 4.3: The locations of the arguments of the *read* operation.

`arg` argument is used to pass an optional argument from user space to the driver. The kernel does not perform any processing on both the `cmd` and `arg` arguments. Also the return value is directly returned to user space without any processing by the kernel. Because of such direct passing of arguments the `ioctl` function can easily be adapted to almost any task that need to be performed by the driver. A pointer to a structure can be passed as argument, when more than one argument have to be passed to the driver. Also a pointer to a buffer can be passed via this method.

Using memory mapping

The `mmap` function can give user application direct access to the memory mapped registers and buffers of the hardware. The `mmap` function will map a part of the applications address space to the physical addresses of the registers and buffers of a hardware device. The user application has via the mapping direct access to the hardware.

Unfortunately, the `mmap` function has a few restrictions. First, of all the mapping is performed on blocks that are the size of pages in the memory management. Secondly, the start of the region must be aligned with the page size. The first restriction is solved by the kernel by making the region as large as the multiple of the page size. However the second restriction remains.

The kernel performs a great deal of preprocessing on the `mmap` call before it is passed to the driver. The prototype of `mmap` handler in the driver is rather different from the call that is made in user space. The prototype of the `mmap` function in the driver is declared as:

```
int mmap(struct file *filp, struct vm_area_struct *vma);
```

The `filp` arguments is the same as for all the other functions that where covered. It contains information on the open file. The `vma` pointer is a pointer to a structure that

contains information about the virtual addresses that are used to access the device driver. To this addresses the driver should map the hardware registers. The actual mapping can be performed by the `remap_page_range` function.

4.4.2.3 The character driver framework

For the SMOKE hardware accelerators a number of drivers where written. These drivers where implemented as modules. In Appendix A, the framework for a module based driver is given. When this driver is loaded, it registers itself and its filehandlers for the *open*, *close*, *read* and *write* procedures. Furthermore, Appendix A also presents two scripts for loading and unloading the driver. The load script loads the module into kernel and creates the corresponding device nodes. The unload script unloads the module from the kernel and removes its device nodes.

Hardware Environment

In this chapter, the hardware environment that is used for the SMOKE project is discussed. The hardware platform used is called DAMP. DAMP is based on the Altera Excalibur device, which contains an ARM processor and FPGA fabric. The integration of a general purpose processor and FPGA into a single device, makes Excalibur well suited for hardware software co-design targeting computation intensive applications with a large number of data transfers, like streaming video decoding.

This chapter is organized as follows. First, an introduction to the DAMP platform is given in Section 5.1. In Section 5.2, a few possible interfaces between the ARM software and the FPGA hardware are discussed. In Section 5.3 the hard- and software that will be used to control the VGA monitor is presented. The VGA monitor will be used to display the decoded images. The hardware that controls this VGA monitor uses one of the interfaces that is discussed in Section 5.2. The VGA hardware will also form the foundation for the hardware accelerator that will be presented in Chapter 6.

5.1 The DAMP platform

As said in the introduction, the hardware platform used for SMOKE is DAMP [38][16][37]. The DAMP board was developed at the Computer Engineering group of Delft University of Technology. DAMP is based on the Altera Excalibur device [7] and is developed to be a substitute for the EPXA1DB development board, that is available from Altera. The main goal of the DAMP project was to develop a reference board that was specially suited for (mobile) multimedia applications. This with the restriction that the board should be cheaper and contain more peripherals than the EPXA1DB board. In order to be suitable to multimedia applications, DAMP provides the necessary peripherals commonly used by stand-alone multimedia applications. Examples of these peripherals are a VGA interface, a DVD quality audio codec, a large SDRAM memory and a network interface.

The core of the DAMP board is the Excalibur device. The Excalibur device features a 32-bit ARM922T processor and FPGA fabric that is based on the APEX 20KE FPGA's device from Altera. The FPGA is often referred as PLD by Altera, which is short for *Programmable Logic Device*. Besides the processor and the PLD several other components are available on chip, like an interrupt controller, an UART, general purpose timers, single and dual port memory, etc. Furthermore, an integrated SDRAM controller is available to connect external SDRAM devices. All these components are connected via two AMBA AHB busses [2] as depicted in Figure 5.1.

The AMBA AHB bus is specially designed for high-performance and high clock frequency systems. On a single bus multiple masters can reside. When multiple masters are connected, arbitration is performed by the bus to prevent simultaneous access by

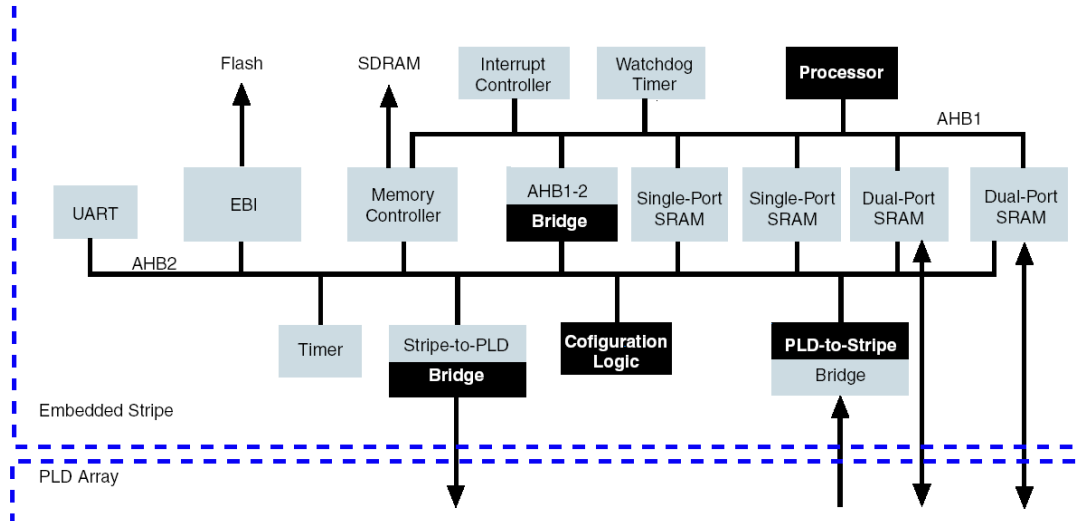


Figure 5.1: A block diagram of the excalibur device.

two or more masters. On the first bus (AHB 1) the ARM processor is the only master. Therefore the processor can always access the bus without waiting for another master to finish. The AHB 1 bus gives the processor access to the internal and external memories at transfer rates equal to the processor's clockspeed.

Three masters are connected to the second bus (AHB 2). In Figure 5.1 all the masters are shown in black and all the slaves in gray. The first master on the AHB 2 is the *AHB 1-2 bridge*. The AHB 1-2 bridge gives the processor access to the components that are only available on the AHB 2 bus, like the EBI interface¹. However, this bridge is unidirectional and does not give masters on the AHB 2 bridge access to slaves on the AHB 1 bridge. The second master is the *PLD-to-stripe bridge*. This bridge enables AHB masters in the PLD to access slaves that are connected to the AHB 2 bus. Finally, the last master is the *configuration logic*. The configuration logic is used to externally configure the PLD and the internal memories with JTAG or an E²PROM device. All AHB 2 masters have access to all the memories, the slaves in the PLD via the stripe-to-PLD bridge, the timer and the UART.

The AHB 2 bus is synchronized with the AHB 1 bus, but runs at half of the AHB 1 clock speed. The AHB masters and slaves that are located in the PLD on the contrary are not synchronized to the AHB 1 and AHB 2 clock, because they have their own clock sources. Therefore, special synchronization logic have been added to both PLD-bridges. These bridges introduce at least a clock delay of two AHB 2 clock cycles [9]. As a result, memory access from the PLD suffers from this delay. The only exception is the access to the DPRAM's (dual port memories). Depending on the configuration of the Excalibur device one of the two ports can directly be accessed from inside the PLD. For this port no synchronization is required.

¹The EBI interface gives access to external memory modules, like flash memory, SRAM or ROM. It is also possible to connect external memory mapped hardware modules. For example on the DAMP board the flash memories and the ethernet controller are connected to the EBI interface.

The frequency of the clocks that can be applied to the Excalibur device, depends on its speedgrade. For DAMP the EPXA1F672C3 device was used. This is the device with the slowest speedgrade. The maximum clock frequency for the processor clock is 133 MHz (compared to 200 MHz for devices with the fastest speedgrade). Besides a distinction in speed a distinction in the size of the FPGA fabric and the on-chip memories can be made. The EPXA1F672C3 is the smallest of three types available in the Excalibur device family. It contains a 100k gates FPGA fabric, while the largest device contains a 1M gates FPGA fabric.

5.2 The hardware/software interface

The XviD decoder will make use of specialized hardware acceleration modules to accelerate the decoding process. In order to communicate with this hardware a interface is needed. For the SMOKE project two different paradigms were explored. The first is to use the MOLEN architecture, as is explained in Section 5.2.1. In Section 5.2.2 the second method, that makes use of memory mapping of control and data registers, is discussed.

5.2.1 Implementing MOLEN on Excalibur

The MOLEN architecture is a relative new concept with as main target to incorporate reconfigurable hardware into a general purpose processing environment. It is developed at the Computer Engineering group of Delft University of Technology. The MOLEN architecture has successfully been implemented on the *Virtex-II Pro* from Xilinx and used to accelerate several multimedia algorithms. In the next sections the basic concepts of the MOLEN architecture are discussed. In addition, an evaluation is made if the MOLEN architecture can be efficiently implemented on the Excalibur device.

5.2.1.1 The MOLEN paradigm

The MOLEN architecture expands the general purpose processing paradigm. In order to increase performance of the general purpose processing environment, acceleration units are added [33]. These acceleration units can be hardwired or be placed in reconfigurable hardware. So the MOLEN organisation consists of an processor core and a reconfigurable core. In order to accelerate programs by using specialized accelerators in this reconfigurable core, only five additional instructions need to be added to the core processor's instruction set. A special selection module, called *arbiter*, partly evaluates all fetched instructions and determines if they need to be executed by the general purpose processor or by the reconfigurable core. The MOLEN instructions are handled by the reconfigurable core, while all other instructions are still handled by the core processor.

The MOLEN instructions perform a *p-set*, *c-set*, *movtx*, *movfx* and *execute* operation. The *p-set* and *c-set* instructions are used to configure the hardware with the

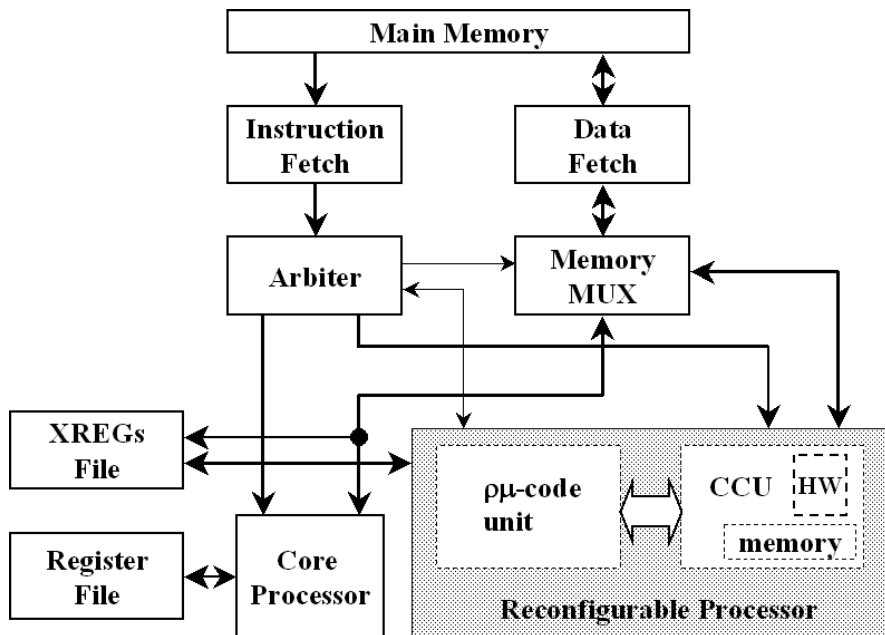


Figure 5.2: The organization of the MOLEN architecture

needed accelerator. The configuration data is stored in the system main memory. The first address of this configuration data is part of the instruction. The `movtx` and `movfx` instructions are used to transfer data between the processor and reconfigurable core. For this purpose a special register file, called XREGs, is available. The `movtx` and `movfx` can be used by the processor core to store data to or load data from in the XREGs file. This register file can be accessed by both, the processor core and the reconfigurable processor. The execute instruction is used to start the actual processing on the accelerator. The operations that need to be performed by the accelerator are coded in microcode. This microcode is located at the memory address that is embedded in the execute instruction. The basic organization of the MOLEN architecture is depicted in Figure 5.2.

5.2.1.2 Implementing the arbiter

One of the crucial elements of the MOLEN architecture is the arbiter. The arbiter is placed between the processor and the memory in order to perform the selection process (see also Figure 5.2). Of course, there is no hardwired arbiter available in the Excalibur. Therefore the arbiter needs to be placed inside the PLD. Intercepting the instructions that are fetched from the memory by the processor now becomes a problem, because it is not possible to reroute the memory bus signals of the ARM processor to the arbiter placed in the PLD. Actually, there are only three methods for the processor to communicate with the PLD. One is via the stripe-to-PLD bridge, the second is via the DPRAM and the third is via the GPIO register. Only the stripe-to-PLD bridge gives the arbiter the possibility to intercept instructions that are fetched by the processor. In this case the arbiter would be connected to the stripe-to-PLD bridge as an AHB

slave and behave like a memory device, from which the processor can fetch instructions. When the arbiter obtains a request for an instruction from the processor, it must first obtain the instruction from the real memory. After the instruction has been fetched, the arbiter will verify if the instruction is a MOLEN or a normal instruction. Dependent on the type of instruction that is obtained, the arbiter will depatch the instruction to the reconfigurable core or to the processor core.

The programcode is very likely to be placed in one of the internal memories of the Excalibur device or the external SDRAM, before it is executed. This unfortunately introduces an conflict when the arbiter tries to fetch the instruction. The processor has obtained the control over the AHB 2 bus in order to fetch the instruction from the arbiter. The processor will hold the bus until it obtains a reply from the arbiter. In the meanwhile the arbiter needs to obtain the instruction from the memory, but needs to wait until the processor releases the bus. The system will not be able to recover from this deadlock situation, unless counter measures are taken. One solution is to send a dummy instruction to the processor, in the main while the arbiter can obtain the real instruction from the memory. A better way to ensure the processor releases the bus, is to send a RETRY response to the processor. When a slave device returns a RETRY the master will retry to obtain the bus and perform the transfer again. The AHB 2 masters are arbitrated using a round-robin scheme [8], therefore all other masters are severed first before the bus is given back to the processor. Another solution to prevent the deadlock, is to place the programcode in a dedicated memory that is directly accessible by the PLD. For example, the programcode could be copied before execution to the DPRAM. Then the arbiter could use the DPRAM port, that is dedicated to the PLD, to obtain the instructions. Of course this only works when the DPRAM is large enough to hold the programcode. When the DPRAM is not large enough, external memory could be added using the IO-pins of the PLD. For the EPXA10 devices it would also have been possible to bypass the on-chip SDRAM controller and place one in the PLD to access the on-board SDRAM memory. Unfortunately, the EPXA1 (that is used for DAMP) does not support this feature.

5.2.1.3 The limitations of Excalibur for MOLEN

Even though it is possible to implement an arbiter on Excalibur, there are a number of drawbacks. Both proposed implementations, the one using the available memories or the one using a dedicated memory to the PLD, suffer from a serious time penalties. The first delay is introduced by the AHB 2 clock speed, which is half of the AHB 1 clock speed. The second delay is the result of the synchronization inside the stripe-to-PLD bridge. This bridge synchronizes between the clock domains of the AHB 2 bus and the AHB bus inside the PLD. The delay that is introduced for read transfers, is two or three cycles of the destination bus its clock [9]. Depending on the chosen solution, the same delay for the PLD-to-stripe bridge needs to be added.

It can be concluded that the arbiter introduces a large delay, that will at least double the program execution time. This becomes even a bigger problem when Linux (or another operating system) is used as execution environment. The entire operating

system will suffer from the delay, even when it does not use any of the accelerators. Fortunately there are Linux kernels that can perform *eXecution In Place* (or XIP for short) [22]. Instead of loading a user program to the main memory before it is executed, the program can directly execute from the memory location where it is stored. This method is originally meant for embedded environments, where programs are often stored in ROM and the RAM size is limited. For MOLEN the arbiter can be seen as the ROM device holding a program. In this case all programs are executed outside the arbiter, except the ones that are run using XIP from the arbiter location. Even though the delay introduced by the arbiter can be reduced, the chances that a hardware accelerator can compensate for the remaining degraded performance are very small.

Another problem that makes the implementation of the MOLEN architecture difficult, is the lack of partial reconfigurability of the Altera PLD. Actually, it is unclear if the Excalibur device can be partly reconfigured or not, because there is no good documentation available about this subject. Even if it would be possible to partly reconfigure the Excalibur device, the development tools do not support this.

The MOLEN architecture is based on the possibility to reconfigure the PLD device to add different acceleration modules. This reconfiguration is performed by the $\varphi\mu$ -unit. Because the $\varphi\mu$ -unit is not available in the Excalibur device, it needs to be placed inside the PLD. Unfortunately, reconfiguration of the Excalibur device will also reconfigure the $\varphi\mu$ -unit, which introduces a conflict. Furthermore, the configuration time for the entire device is at least eight milliseconds. This reconfiguration time is too long to reconfigure the device during the execution of the program.

The impossibility to partly reconfigure the device, the large reconfiguration time penalty, but most of all the large delay introduced by the use of the arbiter in Excalibur, are the reasons not to use the MOLEN architecture for SMOKE. Nevertheless, it is possible to implement the arbitration principle of the MOLEN architecture on Excalibur.

5.2.2 Memory mapped interfaces

A commonly used approach to access hardware modules is via memory mapped registers. The mapped registers are often used to transfer control information or small amounts of data between the hardware and the software. The fact that the registers are mapped to normal memory locations, makes accessing these registers by the software equal to accessing locations of variable. In Section 5.2.2.1 two implementations of a memory mapped interface for the Excalibur device are presented.

When large amounts of data need to be transferred to or from the hardware, a more advanced mechanism can be used. In most cases the data that is transferred to the hardware is already available in the memory. Using the processor to transfer data from the memory to the hardware is not efficient. In this case the use of a separate module that performs the copying is more efficient. This module is called a DMA controller. When the DMA controller is used to transfer data between the memory and the hardware, the processor is available for performing other tasks. The use of an DMA controller on Excalibur is discussed in Section 5.2.2.2.

5.2.2.1 Passive interfaces

For the Excalibur device two methods are available for creating a memory mapped interface. The first method uses an AHB slave inside the PLD to connect registers to the memory bus. The second method uses the DPRAM (Dual Port RAM) for transferring data between the processor and the hardware.

The first method for creating a memory mapped interface, is to connect an AHB slave to the memory bus. In this slave, a number of registers are available. The registers can be accessed by the processor (via the memorybus) and by the hardware inside the PLD. The AHB slave is implemented in the PLD and connected via the stripe-to-PLD bridge to the AHB 2 bus. In order to connect the AHB slave to the PLD bridge several components for bus control are needed. These components can automatically be generated by the SOPC builder, which is part of the Quartus environment. The advantage that the AHB slave is implemented inside the PLD is, that the registers can internally be connected to any other hardware module inside the PLD. In Appendix B, the VHDL source of an AHB slave from an Altera reference design is given.

The second method for creating a memory mapped interface, is to utilize the DPRAM. The reason that the DPRAM can be used as a interface to the hardware, is that one of the DPRAM ports can directly be accessed from inside the PLD. The other port of the DPRAM can be accessed by the the processor and other masters on the AHB 2 bus. The memory locations of the DPRAM can be used to exchange data between the hardware and the processor. From the point of view of the processor the hardware behaves like a memory mapped device. However, from the point of view of the hardware the DPRAM does not behave in the same way as the registers in the AHB slave did. The registers of the AHB slave could be connected to other hardware modules. This is not possible for the DPRAM, because only one value can be read each clock cycle. Furthermore, the hardware can not directly respond to changes that are made to the memory locations of the DPRAM. When the hardware needs to respond to changes in the DPRAM, it should sequentially sample the DPRAM memory locations to notice if values are changed. For this reason the DPRAM is not very efficient to transfer control information to the hardware.

5.2.2.2 Active interfaces

Instead of performing the memory transfers under processor controll, a more advanced method can be used. A specialized module, called a DMA controller, can perform the transfer of data between different components. Using a DMA controller to transfer large blocks of data can increase the performance of the system, because during the transfer the processor is available to perform other tasks. The DMA controller is only suitable for large data transfers, because the initialization of the DMA controller itself introduces some overhead.

To start a DMA transfer, some information is needed by the DMA controller. Examples of such information are the start address and the size of the data block that must be transfered. There are several methods for providing the DMA controller with this information. One of the methods is to use a number of memory mapped registers,

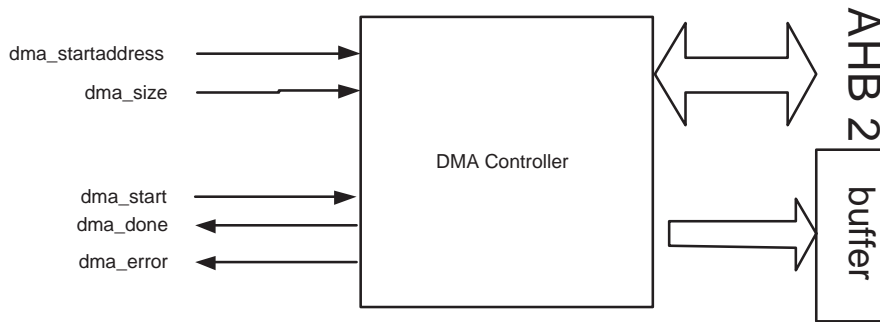


Figure 5.3: The interface of the DMA controller.

that are set by the processor. The DMA controller can read these registers and perform the transfer when it obtains the start signal from the processor. Another possibility is to place a list of transfers in the memory. The DMA controller can read the entries in this list and perform the transfers until it reaches the the end of the list. After the last transfer has been performed, the DMA controller will wait until the processor adds a new transfer to the list. In some applications it is not the processor that initializes the transfers, but other hardware modules. The hardware modules will provide the DMA controller with required information to perform the transfer. The different methods for providing the DMA controller with transfer information are discussed in more detail in [12].

For the SMOKE project a DMA controller has been developed. The DMA controller is placed inside the PLD and can perform transfers via the PLD-to-stripe bridge. The transfers are initiated by another hardware module, that is also located inside the PLD. The designed DMA controller is an alternative to the DMA controller that is available in an Altera reference design. The reason a new DMA controller was designed, is that the reference design from Altera did not implement the complete AHB bus protocol. Furthermore, the design of the DMA controller was difficult to separate from the the rest of the reference design. Our DMA controller contains a full implementation of the AMBA AHB protocol and is modularly designed. Because of the modular design, the DMA controller can easily be used for different applications. In Appendix D, the complete VHDL source of the DMA controller can be found.

The DMA controller is controlled via a number of signals. These signals are *dma_startaddress*, *dma_size*, *dma_start*, *dma_done* and *dma_error*. The interface of the DMA controller is depicted in Figure 5.3. The *dma_startaddress* and *dma_size* give the location and the size of the source data block respectively. Also a number of signals are connected to a local buffer, to store the obtained data. Furthermore, several of control signals are available. The control signals of the DMA controller use handshaking to prevent unintended transfers. The *dma_start* signal initiates a DMA transfer, while the *dma_done* and *dma_error* return the status of the transfer. When *dma_done* becomes high the DMA transfer is completed. Only when the *dma_done* becomes low again a new transfer can be started. The *dma_done* becomes low when the *dma_start* is low. The *dma_error* signal becomes high when an bus error is detected. When an error occurs the

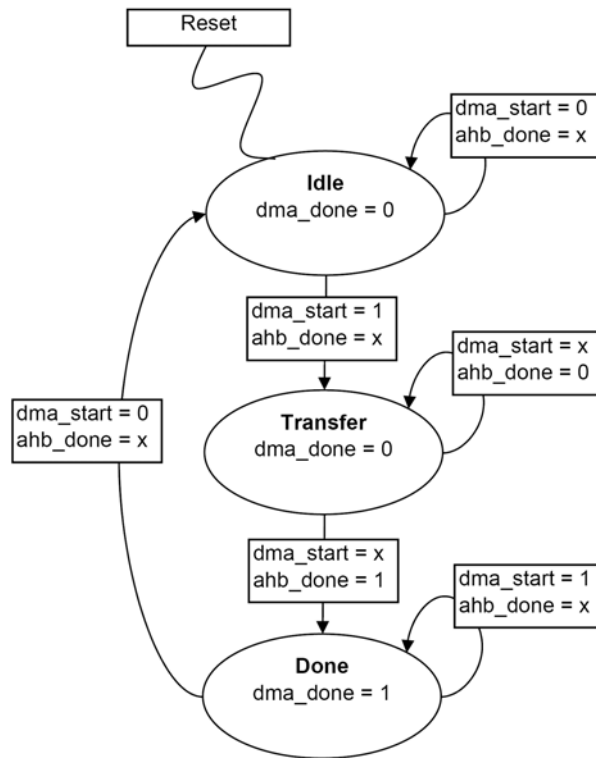


Figure 5.4: The statediagram of the DMA control FSM.

transfer is stopped. The hardware using the DMA controller can decide how to respond to this error. For example, it can try to initiate the same transfer again or continue with a new transfer. Starting a transfer, by changing *dma_start* from low to high, will reset the error condition.

The DMA controller consists of two finite state machines. One state machine handles the handshaking with the initiating module and is called the control FSM. The second state machine implements the actual data transfer and is referred as the AHB FSM. Together with only a few counters and some glue logic these two FSM's form the DMA controller.

The control FSM consists of three states. The following states can be identified, an *idle* state, a *transfer* state and a *done* state. When the DMA controller is ready to start a new transfer it is in the *idle* state. During the transfer it is in the *transfer* state. It will only leave the *transfer* state if the transfer is completed, as signaled by *ahb_done* that is generated by the AHB FSM. When the control FSM leaves the *transfer* state it goes to the *done* state, where it stays until the *dma_start* signal has become low. The state diagram of the control FSM is given in Figure 5.4.

The AHB statemachine is far more complex, than the control FSM. This partly due to the fact the AHB statemachine needs to respond every clock cycle to a number of responses of the slaves. Depended on the responses of the slave, the current address

counter for the AHB bus, as well as the address counter for the local buffer memory needs to be increased, keep the same value or even be decreased. In order to keep the number of states small, the change of the addresses is not depended on the state, but on the transitions between the states. In Figure 5.5 the statediagram for the AHB FSM is given. In this diagram the address operations are in bold font attached to the transfer arrows. It would require a too large introduction of the AHB protocol to explain the working of the AHB FSM, but it is worth noting that the implemented DMA controller has reduced functionality, compared to the proposed design. This, because the first versions of the entire SMOKE design were too large to fit in the available Excalibur device. The size of the DMA controller was reduced by ignoring the SPLIT or RETRY responses of the slave. It is safe to omit these two responses, because the slave that is accessed by the DMA controller in the SMOKE design can not generate these responses.

5.3 The VGA Interface

A VGA monitor can be connected to the DAMP board. This VGA monitor will be used to show the decoded images. In this section, a short introduction to the VGA signals, needed for the communication with the monitor, and the hard- and software required to display images on the VGA screen are presented.

5.3.1 The VGA signals

An image on a VGA screen is build up from a two dimensional array of pixels. Every pixel can have its own color. The monitor illuminates the pixels one for one and line by line. This process is called scanning. The scanning process is depicted in Figure 5.6. By repeating the scanning process at a high speed, an image is created.

The VGA monitor is controlled using 5 signals. Three of these signals represent the red, green and blue intensity of the pixel that is currently written. The signals have analog values, where 0 Volts is equal to black and 0.7 Volts is equal to maximum brightness. The two other signals are digital and used to synchronize the horizontal and vertical scanning. The period of the horizontal synchronization signal determines the time that is required to write a single line. The vertical synchronization signal determines the time used to write an entire frame.

The time used to write a single pixel, depends on the refresh rate (the number of times per second the images is rewritten), the resolution and the time needed by the monitor to synchronize. The VGA standard states that the refresh rate of the image should be 59.94 Hz and the resolution of the image is 640 by 480 pixels. The pixels should be written with a frequency of 25.175MHz to the monitor. In Figure 5.7 the components of the horizontal and vertical synchronization signals and their related timing are presented. The duration of the components is expressed in the number of pixels or lines. Besides the horizontal and vertical synchronization signals, also a *hblank* and a *vblank* signal is depicted. These signals are not used by the VGA monitor, but by the hardware components of the VGA controller. The DAMP VGA interface will be designed according to the VGA standard,

Inputs
 hgrant
 hready
 reset_n
 reset_n
 hresp[1..0]
 boundary1k
 lasttransfer

Outputs
 hbusreq
 htrans
 ahb_done

Operations
 address
 store_data

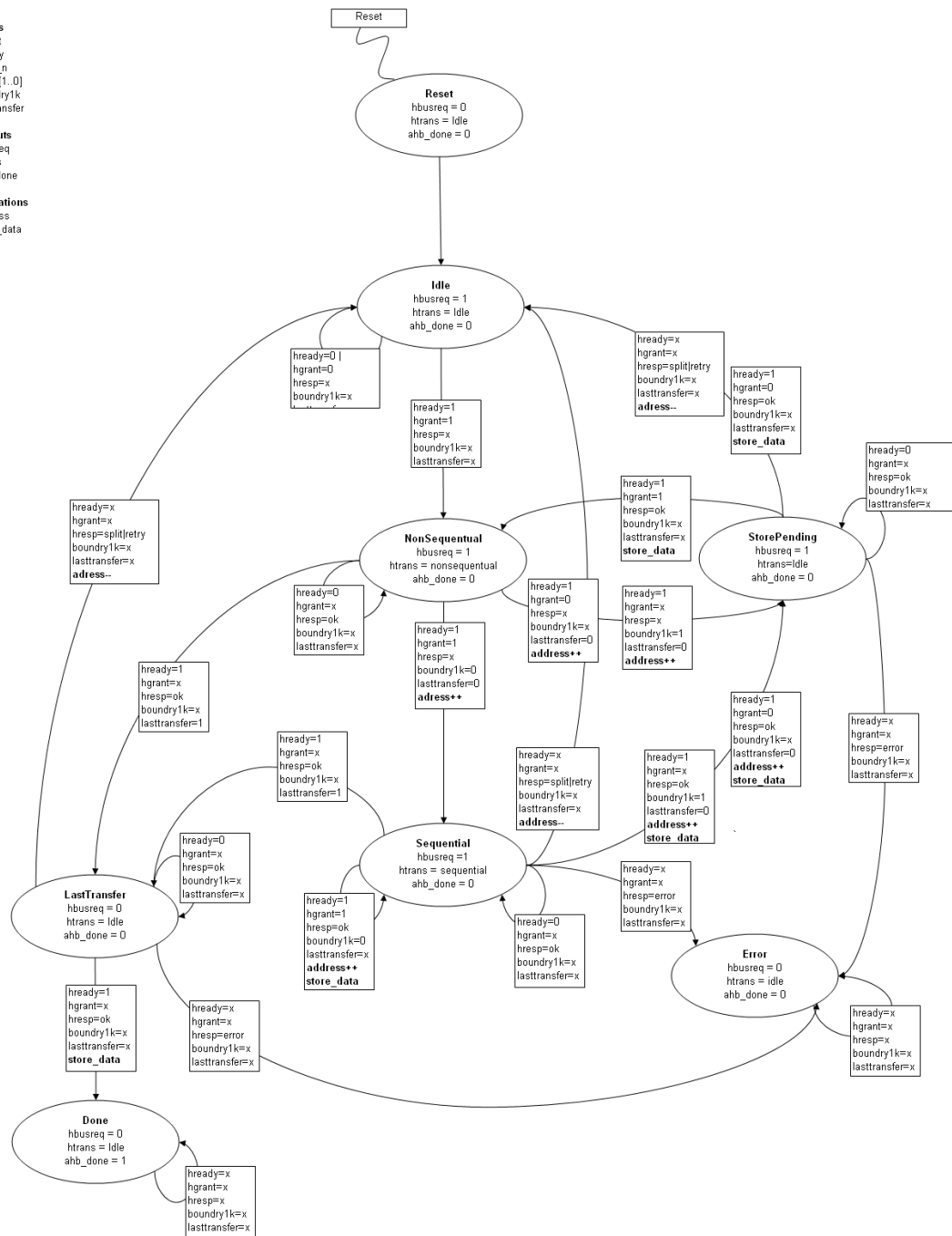


Figure 5.5: The statediagram of the statemachine for the DMA controller responsible for the AHB transfers.

with a small deviation by using a 25 MHz pixel clock, instead of a 25.175 MHz clock. The 25 MHz clock is used, because this signal is available from the external clock sources.

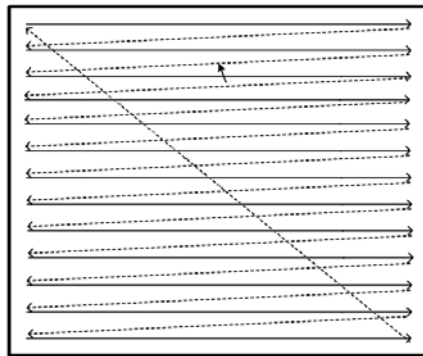


Figure 5.6: The scanning of a VGA screen.

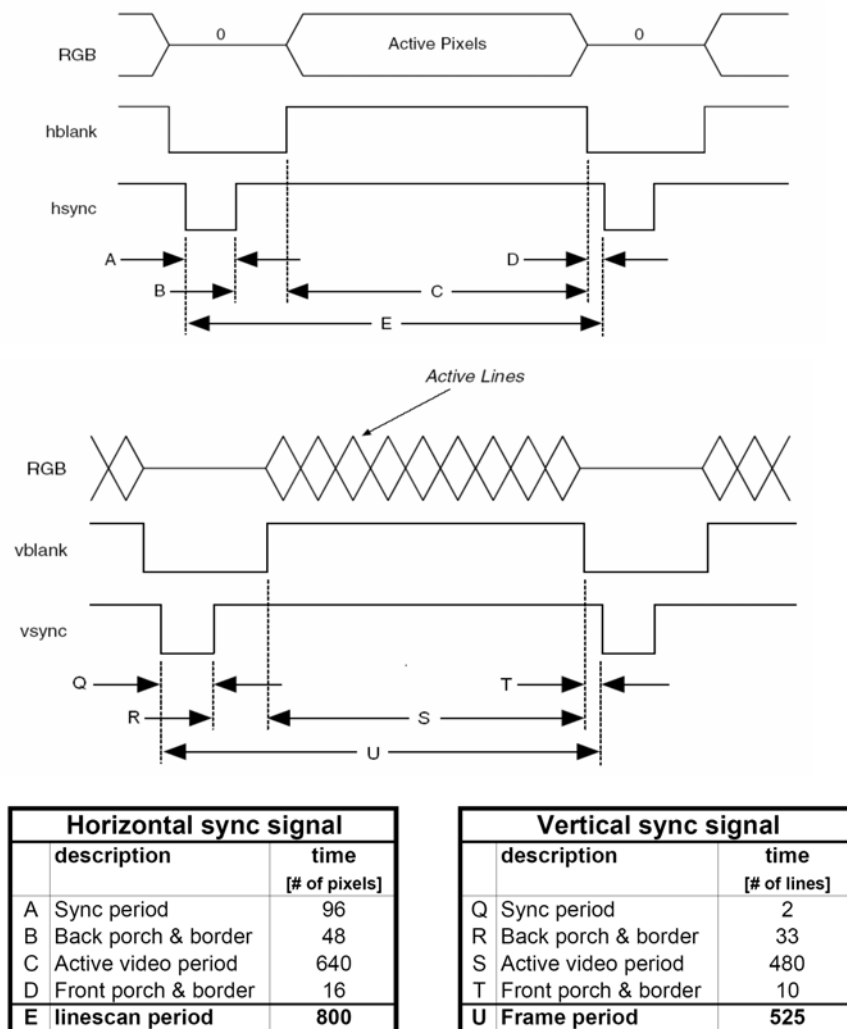


Figure 5.7: The timing relations of the *hsync* and *vsync* signals.

5.3.2 The DAMP VGA hardware

On the DAMP board only the digital to analog converters (DAC) to generate the analog Red, Green and Blue signals and buffers with TTL output for the *hsync* and *vsync* signals are available. The DAC's have a resolution of 3 bits for the red and green channel. For the blue channel only a 2 bits DAC is available. The reason that one of the colors is only encoded using two bits, is to fit the entire color information of one pixel into a single byte. The encoding used is referred as RGB8 in this thesis. The RGB8 format stores in the most 3 significant bits the red value, the green value is stored and the following three bits and the blue value occupies the last two bits. Because 8 bits are used to encode the color information of a pixel, only 256 colors can be shown.

The DAC's and the TTL buffers are connected to the VGA connector on one side and to the PLD on the other side. The hardware that is needed to feed the DAC's with the pixel information and that generates the *vsync* and *hsync* signals is placed inside the PLD. So it is not possible to use the VGA interface, without configuring the PLD.

5.3.3 Implementing the VGA controller

The VGA controller consists of 3 main parts, a framebuffer, a DMA controller and a VGA driver. The framebuffer is used to store the image that is shown on the monitor. This framebuffer also forms the interface between the processor and the VGA hardware. The DMA controller is used to transfer the image data from the framebuffer to the VGA driver. When a line is written to the monitor the DMA controller will be instructed to load the data of the next line. The VGA driver sends the pixels obtained by the DMA controller with the correct timing to the VGA monitor. Furthermore, the VGA driver generates the necessary control signals for the DMA controller and the VGA monitor. The processor can control the VGA driver by setting some registers, that are connected to the ARM processor via the stripe-to-PLD bridge. A block diagram of the VGA controller is depicted in figure 5.8.

The framebuffer is located in the SRAM of the Excalibur device, while both the DMA controller and the VGA driver are placed in the PLD. Because of the limited size of the SRAM only frames with a maximum resolution of 160 by 120 pixels can be displayed. A framebuffer located in the external SDRAM would eliminate this restriction. Unfortunately the current prototype of DAMP does not allow highspeed memory transfers from the SDRAM. Highspeed memory transfers are required for the correct operation of the VGA driver.

The VGA driver itself consist of several modules (depicted in figure 5.9). It contains a linebuffer, for holding the pixel information of the current line, pixel- and linecounters, and some additional glue logic. The pixelcounter is directly connected to the linebuffer, which feeds the DAC's with the color information of the current pixel. The pixelcounter also generates the *hsync* signal and the clock signal of the linecounter. Furthermore it generates the *hblank* signal that is high when the pixels are written to the VGA monitor. This is called the the active period of a line. The *hblank* signal is not used by VGA monitor, but by the other modules of the VGA controller. For example, the DMA controller needs to be started after the last pixels is written to the VGA interface. The

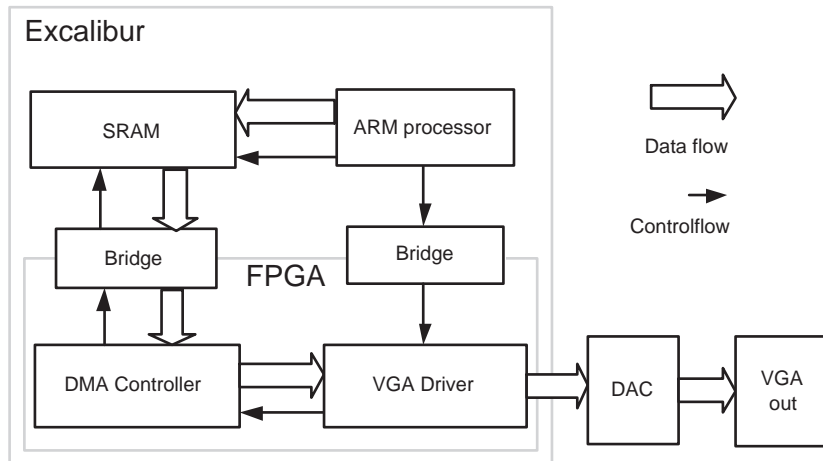


Figure 5.8: Blockdiagram of the VGA controller.

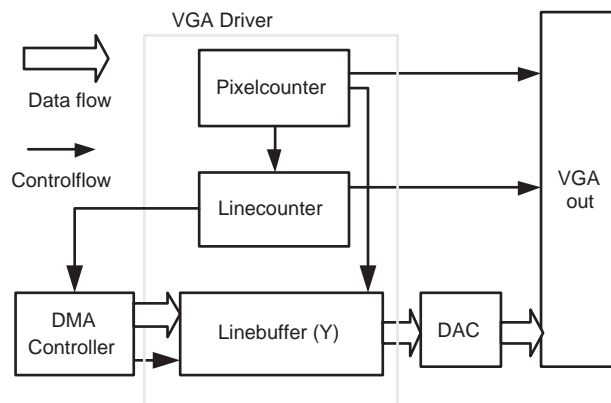


Figure 5.9: The internal components of the VGA driver.

DMA controller uses the high to low transition of the *hblank* signal to identify the end of the active period. The linecounter is connected to the DMA controller, which needs to refresh the linebuffer after a line is written. With the current linenumber provided by the linecounter, the address of the new line within the framebuffer can be calculated. The linebuffer is then filled with the data for the next line, before the first pixel is written. Furthermore the linecounter is also responsible for the generation of the *vsync* and *vblank* signal. The *vblank* signal is high when lines are written to the VGA monitor. This is called the active period of the line. The *vblank* signal is only used by the other hardware modules and is not used by the VGA monitor.

5.3.3.1 Hardware/software interface

The communication between the processor and the VGA driver is performed via the framebuffer that is placed in the SRAM. The DMA controller that was described in

Section 5.2.2.2 is used to transfer the data from the framebuffer to the VGA hardware modules. The DMA is under the control of the VGA hardware.

The VGA hardware is controlled using an AHB slave that contains two registers. The first register is used to store the address of the framebuffer. This address can dynamically be modified. The second register is used for two control bits. Bit one of the control register is used to enable or disable the VGA hardware. By writing a one to this control bit the hardware can be enabled, while writing a zero will disable the hardware. Bit two of the control register is used to set the operating mode of the colorspace conversion module. Writing a zero to this bit will set the colorspace conversion module in yv12 mode, while a one will set the module in RGB8 mode. The remaining bits of the control register (including bit zero) are not used.

5.3.4 Software drivers

The access to the VGA controller depends on the environment on which the software runs. For the standalone environment no special software is needed to copy the image data to the frame buffer. For Linux however, accessing the framebuffer is more complicated. In Linux the access to the framebuffer is made via the VGA driver that was written for this purpose. The decoder application was modified to support both environments.

5.3.4.1 Standalone environment

For the standalone environment accessing the framebuffer is not different than accessing any other memory location. When the decoder finished decoding a frame, it will convert the BGR24 result to RGB8 and copy the image to the framebuffer. On DAMP, the decoder can not provide the images real-time. If the decoder would deliver the images faster than would be required for real-time playback, some additional flow control need to be added.

5.3.4.2 Linux drivers

A Linux driver was build to give the decoder application access to the VGA framebuffer. This driver implements a number of minor devices. Each of these minor devices accept as input a different image format. In Table 5.1 the different accepted formats and their related minor numbers are given. If the input format is different from the format used in the framebuffer (which is RGB8), the driver converts the input to RGB8. The advantage that the driver can support different input formats is, that the conversion to RGB8 does not need to be implemented in the applications that use the VGA driver. In this way code duplication is avoided.

The VGA driver will implement two of the three interface methods that were discussed in Section 4.4. These methods are the *read/write* routines and the *ioctl* routine. The *mmap* was not implemented, because initial tests showed that the mapped

minor number	input format
0	default handler, has no file in or output
1	for testing purposes
2	for testing purposes
3	reserved
4	RGB8
5	RGB24
6	BGR24 (output of the decoder after colorspace conversion)

Table 5.1: The different input formats and their minor types

memory could only be used when reading and writing was performed with a significant delay. Due to time limitations and the availability of good alternatives, this method was not further investigated.

Implementing the read/write routines

The read/write routines were implemented because they can easily be used in combination with available commandline tools. So for example it is possible to use the program `cat` to send an image to the VGA driver. The following invocation can be used for sending an image called `penguin.rgb8` to the VGA driver (assuming that a device node for the RGB8 input is available at `/dev/vga_driver_rgb8`).

```
cat penguin.rgb8 > /dev/vga_driver_rgb8
```

A tool, called `bmp2raw`, was developed to generate from a 24 bits bitmap with a resolution of 160 by 120 pixels the different image formats accepted by the driver (RGB8, RGB24 and BGR24).

For sending images to the framebuffer the write routines of the driver are used. For all the minor devices a write routine is implemented. On the contrary, the read routines are not implemented for all the minor devices. Only minor device two has an implementation for the read routine. This minor device is only used for testing purposes. The reason that the read routine was not implemented for all the minor devices is, because this routine has little meaning. The image inside the framebuffer will not be changed by the hardware. Therefore the image that would be read from the framebuffer, would always be the same as the image that was written to it.

The write function for the RGB8 minor device copies the data from user space directly to the frame buffer. For the other two routines this is not possible, because the image needs to be converted before it can be copied to the framebuffer. The image is therefore first placed in a buffer, converted to RGB8 and then it is stored into the framebuffer.

Reading and writing to files can be performed in parts. In the case of the VGA driver this means that the write function will be called multiple times, before the entire image is sent to the VGA driver. The offset argument is used to keep track of the position where the data needs to be written. The same offset argument is used by the RGB24 and BGR24 devices to detect if the entire image is obtained. When for the RGB24 and

BGR24 devices the entire image is obtained, the image is converted to RGB8 and copied to the framebuffer.

Implementing the *ioctl* routine

The *ioctl* function was implemented to reduce the number of system calls that were needed to transfer the image data to the framebuffer. As said before, the write function will be called multiple times before the entire image is written to the framebuffer. This will reduce the performance of the application. Furthermore, writing multiple images to the framebuffer, requires rewinding the filestream after each image is written. This makes the write routine less efficient. The *ioctl* routine can be used as an alternative to the write function. In this case the *ioctl* function is called with two arguments. The first argument is a command number to inform the driver which operation it needs to perform. The second argument contains a pointer to a memory buffer (in user space) that contains the image that needs to be transferred to the framebuffer. The *ioctl* implementation for the VGA driver will copy the image to kernel space, if needed it will convert the image to the right format and store the image in the framebuffer. The entire image is transferred in a single system call. This reduces the overhead introduced by communication between the user application and the hardware.

The *ioctl* function however can not be used by generic applications, which was the case for the write function. Only applications that have special support for this specific implementation of the *ioctl* function of the VGA driver, can use the *ioctl* interface. The decoder has been modified to support the *ioctl* functions of the different minor devices of the VGA driver. In Appendix E, the source code of the VGA driver can be found.

Speeding up the Colorspace conversion

6

As a result of the profiling presented in Section 2.4, the colorspace conversion was identified as one of the most computation intensive kernels of the XviD decoder. To speed up the decoder, a hardware accelerator for the colorspace conversions was build.

The organization of this chapter is as follows: first the concept of colorspace will be explained in Section 6.1. Section 6.2 will discuss in more depth the colorspace conversion. Finally, in Section 6.3 the software modifications made to the VGA driver and to the XviD decoder are presented.

6.1 Colorspaces

A colorspace is an abstract mathematical model for describing colors as a set of different components (all represented by a value). A colorspace has no meaning without a mapping to a reference colorspace. The reference standard normally used is the CIE XYZ colorspace [27]. This colorspace can represent all the colors that are visible for the human eye. Because the CIE XYZ colorspace is too complex or impractical to use in normal applications, colorspace that are adapted to the application have been developed. These colorspace can be categorized in different groups. The colorspace of a particular group use the same principle for describing colors. In this thesis these groups are referred as colormodels. Examples of these colormodels are the RGB model (the group of colorspace based upon a combination of the three primary colors Red, Green and Blue (RGB)), the luma-chroma model (the group of colorspace based on a brightness component and components that describe the color differences (chroma)) and the HSV model (the group of colorspace based upon Hue, Saturation and Value (brightness) components).

For digital decoders, a colorspace based upon the luma-chroma colormodel is frequently used. In the literature the used colormodel is often referred as YUV, but YUV is in fact a colorspace used for analog video. Even more unfortunate are the resembles of the Y in YUV, with the Y in the CIE XYZ colorspace. The Y used in the CIE XYZ colorspace stands for the luminance. Even though both values have an relation with the brightness of an image, they do not have the same value for the same color. This, because the CIE XYZ colorspace is calculated by the use of linearly distributed RGB values, while the Y values in case of YUV are calculated with gamma corrected¹ R'G'B' values. The quotation mark are used to denote that RGB values have been subject to the non-linear gamma correction. In the literature not enough care is taken to prevent the ambiguity by the use of strict notation. In most cases the terms YUV and luminance are misplaced in the context of digital imaging [28]. The

¹Gamma correction is an non linear function that is applied to the color components of an colorspace, to compensate for the non-linear color behavior of the displaying devices.

correct terms are $Y' C_B C_R$ and luma respectively. The quotation mark for the Y is used to denote the difference with the Y used in the CIE XYZ colorspace. While for both chroma representations C_B and C_R the use of the quotation is not needed, because they have no ambiguous meaning. The $Y' C_B C_R$ colorspace and its conversion to R'G'B' are defined by **ITU-601** standard.

The XviD codec uses $Y' C_B C_R$ as its native colorspace. This was confirmed by comparing the conversion matrices used by XviD, with the conversion matrices used for $Y' C_B C_R$ as stated in [26]. Even though XviD uses $Y' C_B C_R$ as its native colorspace, it unfortunately refers to YUV in most of the source files. For this thesis the correct terms $Y' C_B C_R$ and luma will be used where appropriate, even when the source files or documentation of XviD state otherwise.

The luma-chroma colormodel can make special use of the properties of the human eye. The human eye is far more sensitive to brightness than to color [4]. This property makes it possible to lower the resolutions for the color information, without resulting in noticeable differences for the human eye. Because the luma-chroma colormodel separates the color information from the brightness, it is well suited to perform this kind of (lossy) compression. For example, for every pixel one value can be stored for the luma, while for only every four pixels one value can be stored for each of the chroma values. This results in images that are half the size of images stored in the RGB colormodel.

6.2 Colorspace conversions

In the XviD decoder the colorspace conversion is responsible for outputting the decoded image in the colorspace and storage format² that is requested by the calling application. The XviD decoder can output the decoded image into several colorspace and formats, for instance, B'G'R' (Blue, Green and Red) and i420 (luma, chroma blue, chroma red). These conversions are linear transformations of the native colorspace $Y' C_B C_R$ used by XviD, which can be implemented as matrix multiplications. When only the format is different, but the target colorspace is based on $Y' C_B C_R$, a reordering of the bytes, subsampling or interpolation as explained later is sufficient. The colorspace conversions or format transformations can be reversed, but will in most cases result in small errors. The consequence of these errors is the loss of image quality. In Figure 6.1, the quality loss as a result of a conversion to an yv12 image is depicted. In the zoomed image, the artifacts of the conversion can clearly be seen.

The XviD decoder uses yv12 as native format for storing images. The yv12 format uses $Y' C_B C_R$ as native colorspace, which is part of the luma-chroma colormodel. The yv12 format stores for every pixel one value for the luma and it stores for every square block of four pixels one value for each of the chroma parameters. Storing only one value for multiple pixels is called subsampling. There are several methods to obtain the subsampled value. For example the subsampled chroma value for the four pixels can be the average of the chroma values of the four pixels. An other method for subsampling

²In this context a format represents the way the image is stored in the memory or on disk. The same colorspace can be stored using different formats, or in other words memory layouts



(a) The original image before colorspace conversion.



(b) The image after conversion to the yv12 colorspace.



(c) A zoomed version of the original image before colorspace conversion.



(d) A zoomed version of the image after conversion to the yv12 colorspace.

Figure 6.1: The quality loss after converting an image to yv12.

that consumes less computation time than the averaging method, is to use the value of the first pixel for example. This method assumes that the nearby pixels will almost have the same color. When converting the image back into another format or colorspace, an interpolation needs to be performed to generate one luma and two chroma values for every pixel. The simplest form of interpolation is to use the subsampled value for all four pixels. More advanced forms of interpolation are available, but they are more computation intensive. The subsampling and interpolation methods have influence on the quality of the restored image.

Beside the subsampling the layout is part of the format. Two types of layouts are frequently used for luma-chroma colorspace. These are the planar layout and the packed layout. The planar method uses separate arrays to store the different values for luma and both chroma's, as depicted in Figure 6.2(a). While the packed layout on the other hand stores all data in a single array, where the data of a single pixel is packed together

as depicted in Figure 6.2(c). The planar layout is very useful when the chroma values are subsampled. In the case of subsampling, the packed layout becomes irregular because not for every pixel the same amount of values are saved. This makes the packed method more difficult to process in case of subsampled images. For the planar layout, the subsampling has no effect on the regularity of the stored image. The only consequence is, that the arrays for both the chroma values are becoming smaller (see Figure 6.2(b)). Of course, there is a solution for making the processing of the packed format regular and therefore better to process. This solution is to handle the stored image as a set of macro blocks. Every macro block contains the information of a small set of pixels. The yv12 format uses a planar layout to store an image.

In Section 5.3 it was explained that the VGA driver works in the RGB8³ colorspace. Therefore a conversion from yv12 to RGB8 needs to be performed. Because the XviD decoder does not support the RGB8 colorspace, the closely related BGR colorspace is used. The driver for the VGA interface is modified to support BGR input and it converts the BGR input to RGB8 before it is written to the framebuffer. This process is explained in Section 6.3. In Section 6.2.1 the software version of the colorspace conversion from yv12 to BGR and from BGR to RGB8 is explained. In Section 6.2.2, the hardware version of this colorspace conversion is given.

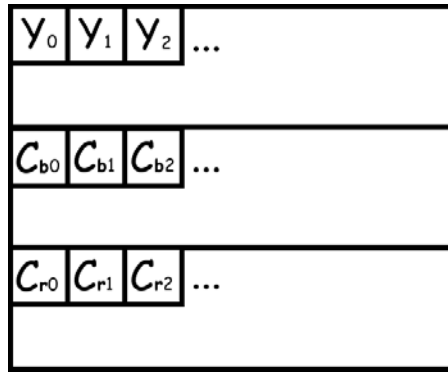
6.2.1 Software colorspace conversion

The colorspace conversion available in XviD decoder performs a yv12 to BGR conversion. The result of this colorspace conversion is closely related to the RGB8 format used by the VGA controller. The BGR colorspace saves tree 8 bit values for every pixel. These values represent the Blue, Green and Red components respectively. In order to be capable to display the image with the DAMP VGA controller the 8 bit values need to be reduced to 3 bit values for the Red and Green component and a 2 bit value for the Blue component. This operation is performed by the driver by selecting the most significant bits of the different color components. Furthermore the bytes are reordered to RGB instead of BGR in the same operation.

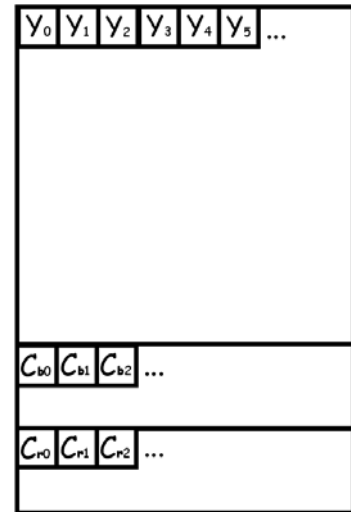
The colorspace conversion can be described by the matrix multiplication stated in Equation 6.1. All input values are between 0 and 255. The source code of XviD uses numerous macros to implement the different colorspace conversions. A human readable version of the yv12 to BGR conversion used by the XviD decoder, is given in Appendix C. Besides the code that performs the matrix manipulation, also code for clipping has been added. It is possible that the result values are outside the 8 bit range. It is undesirable that for these cases the under- or overflow results are used. Therefore the values are saturated to the minimum or maximum value.

$$\begin{pmatrix} B'_{255} \\ G'_{255} \\ R'_{255} \end{pmatrix} = \begin{pmatrix} 1.164 & 2.018 & \\ 1.164 & -0.391 & -0.813 \\ 1.164 & & 1.596 \end{pmatrix} * \begin{pmatrix} Y' - 16 \\ C_R - 128 \\ C_B - 128 \end{pmatrix} \quad (6.1)$$

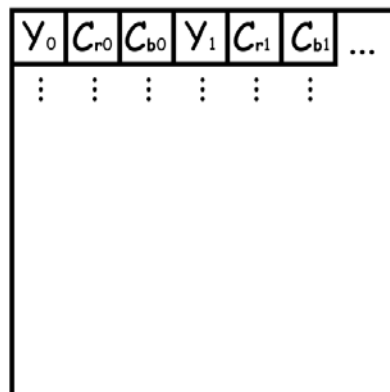
³The RGB8 colorspace is not commonly accepted as colorspace, but was defined in Section 5.3.



(a) An image stored in a luma-chroma colorspace using the planar method.



(b) A subsampled image stored in a luma-chroma colorspace using the planar method.



(c) An image stored in a luma-chroma colorspace using the packed method.

Figure 6.2: The different formats for storing images in the luma-chroma colorspace.

6.2.2 Hardware colorspace conversion

As stated in Section 2.4, the software colorspace conversion consumes a significant amount of computation time, while the same can be achieved with a relatively small hardware implementation. Therefore only a few modules need to be added to the basic VGA controller as described in Section 5.3. It needs to be noted, that the colorspace conversion acceleration can only be used, when the decoded images are displayed on the VGA screen. It is not possible to use the accelerator when the images are for example

stored to a file. This is due to the fact the accelerator is tightly integrated with the VGA controller. Fortunately this restriction is of little importance, because for an embedded platform displaying images is more relevant operation than storing these images. Furthermore, displaying images is more bound to timing constrains, compared to writing the images to a file.

The hardware colorspace conversion is performed just before the pixels are fed to the DAC. So the luma and two chroma values for the current pixel are fed to an conversion modules, which outputs the R'_3, G'_3, B'_2 values to the DAC. An alternative solution would be to convert the entire framebuffer at once, by a hardware module located in the reconfigurable fabric. After the entire framebuffer is converted, the current VGA controller can be used to display the image. Unfortunately, this approach would introduce a delay before the image can be displayed. Furthermore it would introduce heavier load on the memory bus, because multiple transfers to and from the FPGA fabric of the same data are needed. This would eventually degrade the performance of the entire system. Therefore the first solution was selected.

The first solution requires the datapad to be modified to support the separate luma and chroma values. This is the result of the planar storage method used by the yv12 colorspace. For the RGB8 colorspace, all the values needed to display a pixel were stored in a single byte. The linebuffer, therefore, could be filled with consecutive bytes from the image. The yv12 colorspace on the other hand stores the values needed for the conversion of a single pixel on three different locations in the memory. It would be very inefficient to load single bytes from different locations, because the DMA controller has to be reinitialized after the transfer of each byte and also the transfer modes of the AMBA bus are not fully exploited. Therefore two linebuffers are added, to be able to store the values for the luma and chroma for an entire line separately. Now, the DMA controller can transfer the values of the different components for an entire line. The linebuffer that was already available to hold the RGB8 values has the most storage capacity and is therefore used to store the luma values of the current line. The other two linebuffers are smaller because only the half number of bytes for a single line is needed to store the two chroma values for the current line. Also the refresh rate of these linebuffers is half of the refresh rate of the luma buffer. In order to fill the three linebuffers with data from different locations, a statemachine for controlling the DMA controller is added. In addition logic is added to calculate the different address locations.

The final modification made to add yv12 support, is placing a colorspace conversion module between the output of the linebuffers and the DAC. Because the design of new VGA controller still contains the old modules, it is still possible to display images that are in the RGB8 colorspace. For this purpose, the colorspace module can be switched between RGB8 mode (pass-through) or yv12 mode by the means of setting a controlbit in the controlregister of the VGA controller.

The colorspace conversion module is designed using VHDL. In Appendix C, the source of the colorspace conversion module can be found. For the conversion, the matrix multiplication as stated in Equation 6.2 is used.

$$\begin{pmatrix} B'_{255} \\ G'_{255} \\ R'_{255} \end{pmatrix} = \begin{pmatrix} 149 & 258 & \\ 149 & -50 & -104 \\ 149 & & 204 \end{pmatrix} * \begin{pmatrix} Y' - 16 \\ C_R - 128 \\ C_B - 128 \end{pmatrix} * \frac{1}{128} \quad (6.2)$$

The input and output values are in the range between 0 and 255. All operations on these values are performed by using integer arithmetic. Because the values of the matrix used in Equation 6.1 are too small to perform the calculations using integer arithmetic without introducing large rounding errors, the values are upscaled by a factor 128 and downscaled again after all operations are completed. Such a matrix with scaled multiplication factors is also presented by [26], but differs from the scaling factor used here. In [26] a scaling factor of 256 was used, instead of the scaling factor of 128 used for our hardware accelerator. During the development, it was chosen to lower the scaling factor to 128 to save hardware area. The result of the lower scaling will not be visible due to the low number of colors that can be displayed by the DAC.

In Figure 6.3, the block design of the modified VGA controller is given. The gray shaded boxes are added for the yv12 support. When a comparison is made to the original VGA controller design (Figure 5.8), it can be seen that several new modules have been added to the VGA controller. The old datapath still exists to support the RGB8 colorspace.

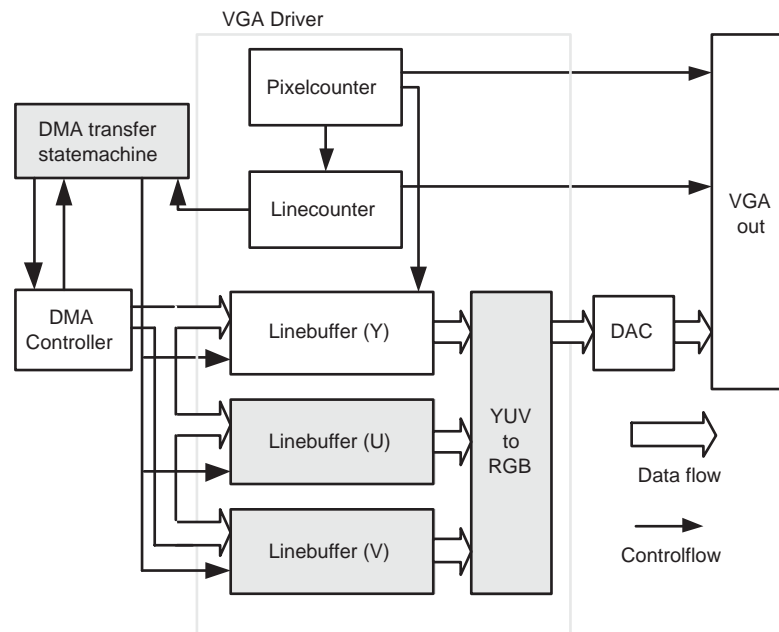


Figure 6.3: Block diagram of the VGA driver supporting yv12. The gray blocks are added for the yv12 support.

6.3 The modifications of the software

The modifications to the XviD decoder to add support for the colorspace module are minimal. The XviD decoder was modified to support sending yv12 images to the frame-buffer. In the commandline argument of the XviD decoder it is possible to select if the output images are in the bgr format, or in the yv12 format. The first option selects

that the decoder should use software decoding. The second option instructs the decoder to use the hardware for the colorspace conversion. Depending on the environment that is used, the data is send directly to the framebuffer (standalone environment), or to a driver (in case of Linux). In the standalone environment the decoder has to initialize the VGA driver to the correct output mode (rgb or yv12). In case of Linux, the initialization is handled by the driver.

For Linux it was not necessary to build a new driver for the colorspace conversion hardware. The driver that was already available for the VGA hardware could be used. This driver is described in Section 5.3.4. Only a small number of modifications were required. First of all an additional minor device was added to the driver. This minor device (with a minor number of 3) accepts yv12 formatted input. The write function of this minor device is not different from that of the rgb8 minor device. The only difference between the yv12 device and the rgb8 device is the *open* function. This open function changes the mode of the colorspace conversion module to rgb8 or yv12, depended on the device node that is opened. So when the rgb8, rgb24 or bgr24 device nodes are opened, the VGA hardware is switched to rgb8 mode. The rgb24 and bgr24 of course still need to be converted to rgb8. When the yv12 device is opened the yv12 mode of the hardware is selected.

These modifications were sufficient to enable the yv12 support to the decoder application. In Appendix E, the complete source of the VGA driver with yv12 support can be found.

Experimental results

In this chapter, the speedup of the XviD decoder is discussed. This speedup is the result of the colorspace conversion accelerator as described in Chapter 6. Besides the speedup of the colorspace conversion kernel, the speedup of the IDCT kernel is presented. The IDCT accelerator study is described in [14]. The performance results for both kernels will give a complete overview of the speedup that was realized with the SMOKE project.

The organization of this chapter is as follows. In Section 7.1, the methodology that was used for the measurements is elaborated. The speedup calculations based on the Amdahl's law are explained in Section 7.2. Finally, in Section 7.3 the results of the measurements and the calculated speedups are presented.

7.1 Test methodology

To calculate the speedups of the XviD codec, the execution time with and without hardware accelerators is needed. The average decoding time of a frame will be used in the calculations to follow. The decoding time consists of the time needed by the XviD decoder to decode a single frame from the input stream and to transfer the resulting image to the VGA hardware. The time for initializing the decoder and the file access times are not considered.

For the measurements, four test sequences containing MPEG-4 compressed video data were used. The test sequences were created using the XviD encoder that is also part of the XviD project. The reference video files that were used for the test sequences were obtained from the Technical University of Munich [18]. Each of the test sequences consists of 20 frames.

The time measurement is performed using the `clock` function that is part of the C library. On the ARM platform this function has a resolution of 10 milliseconds. This resolution is relatively low compared to the execution time of the XviD decoder. The influence of rounding errors is reduced by averaging the decoding time of the different frames.

Before the speedup of the XviD decoder is evaluated, the speedup of the colorspace conversion accelerator is determined separately. As indicated earlier, the clock resolution is too low to measure the execution time of a single colorspace conversion. To overcome this restriction the kernel is executed for twenty frames. From the total execution time the average execution time of a single frame is calculated. The input data that is used for the multiple execution of the kernel, is extracted from the test sequences. The original XviD decoder was modified to store the input values of the kernel in a file. The extraction process is performed for the four test sequences. As a result of this four test sequences four *colorspace data files* are obtained. These files contain the data for the 20

frames that are normally fed by the XviD decoder to the colorspace conversion module. A test application was written to execute the colorspace conversion kernel using the data obtained from the colorspace data files. From the total execution time of the test application the average execution time for the colorspace conversion kernel to process a single frame can be calculated.

In the scope of the SMOKE project also the IDCT kernel was accelerated as discussed in [14]. The results of the IDCT kernel acceleration are obtained from [14] and will be used in this chapter without further explanation.

The execution time of the XviD decoder for four different configurations is measured. The following configurations of the XviD decoder were used.

- Software IDCT, Software colorspace conversion (SW IDCT, SW CSPC).
- Hardware IDCT, Software colorspace conversion (HW IDCT, SW CSPC).
- Software IDCT, Hardware colorspace conversion (SW IDCT, HW CSPC).
- Hardware IDCT, Hardware colorspace conversion (HW IDCT, HW CSPC).

The hardware accelerators can be enabled or disabled by setting commandline switches passed to the XviD decoder. The above four different configurations were investigated using the four different test sequences as introduced earlier.

7.2 Speed up calculations

The calculations of the speedups are based upon Amdahl's law. In Equation 7.1, the formula for calculating the speedup when a single kernel is accelerated in hardware is presented.

$$S_i = \frac{T_{sw}}{T_{sw} - T_{sw,i} + T_{hw,i}}, \quad (7.1)$$

where T_{sw} is the execution time of the decoder when none of the parts are accelerated by hardware. $T_{sw,i}$ is the execution time of a single kernel in software and $T_{hw,i}$ the execution time of the same kernel in hardware. This formula can be rewritten as follows,

$$S_i = \frac{1}{1 - \alpha_i + \frac{\alpha_i}{s_i}}. \quad (7.2)$$

where $\alpha_i = \frac{T_{sw,i}}{T_{sw}}$ and $s_i = \frac{T_{sw,i}}{T_{hw,i}}$. The factor α_i is the percentage of the total execution time that is spent by the kernel. The factor s_i represents the kernel speedup.

When multiple kernels are accelerated using hardware, the speedup is given by

$$S = \frac{T_{sw}}{T_{hw}} = \frac{T_{sw}}{T_{sw} - \sum T_{sw,i} + \sum T_{hw,i}}, \quad (7.3)$$

where T_{hw} is the total execution time when all the kernels are implemented in hardware. This function can be rewritten, using α_i and s_i too. The formula then becomes:

$$S = \frac{1}{1 - \sum \alpha_i + \sum \frac{\alpha_i}{s_i}}. \quad (7.4)$$

The maximum speedup can be calculated by assuming $\sum T_{hw,i} = \sum \frac{\alpha_i}{s_i} \rightarrow 0$ (the hardware executes in zero time).

7.3 Measurements

In this section, the results of the measurements are presented. In Section 7.3.1 the speedup of the color space conversion and the IDCT are presented. In Section 7.3.2 the speedup of the XviD decoder is presented.

7.3.1 Kernel speedup

Colorspace conversion

The colorspace conversion kernel was tested using the colorspace data files (obtained from the modified XviD kernel). The special test application performs both the software version and the hardware version of the colorspace conversion. The results of the measurements using the colorspace conversion kernel are presented in Table 7.1. The values represent the average kernel execution time per frame for every reference stream. The execution time of the software version consists of the colorspace conversion and the transfer time to the VGA framebuffer. The execution time of the hardware version does only consist of the memory transfer, because no calculations are performed by the processor. It is expected, when the speed of the DAMP SDRAM increases, the difference between the software and the hardware colorspace conversion to become even larger. This is because the memory transfer penalty becomes much smaller compared to the execution time needed for the colorspace conversion. When a faster version of the DAMP platform is available, it would be interesting to perform the tests again.

	mobcol	parkrun	shields	stockholm	average
	[ms/frame]	[ms/frame]	[ms/frame]	[ms/frame]	[ms/frame]
software	135.0	136.1	133.4	134.7	134.8
hardware	37.6	37.6	37.6	37.7	37.6
speed-up	3.59	3.62	3.55	3.58	3.59

Table 7.1: The speedup of the colorspace conversion.

Based upon the speedup of the kernel and the profiling results presented in Section 2.4 the expected speedup of the XviD decoder can be calculated. For this calculation, Equation 7.2 is applied. For α_i the profiling result of the colorspace conversion kernel is used, which is almost 27% (see Section 2.4). The average speedup used for s_i is 3.59, obtained from Table 7.1. The XviD decoder speedup is calculated as 1.24. This speedup is close to the theoretical maximum estimated as 1.36.

IDCT

For the IDCT, a similar methodology was used. The results of the measurements are stated in Table 7.2. The values were obtained from [14].

From the results in Table 7.2, the speedup of the XviD decoder can be estimated.

	mobcol	parkrun	shields	stockholm	average
	[$\mu\text{s}/\text{idct}$]	[$\mu\text{s}/\text{idct}$]	[$\mu\text{s}/\text{idct}$]	[$\mu\text{s}/\text{idct}$]	[$\mu\text{s}/\text{idct}$]
software	340	371	315	309	334
hardware	142	221	181	174	180
speed-up	2.39	1.67	1.74	1.78	1.85

Table 7.2: The speedup of the IDCT.

For α_i the profiling result for the IDCT is used, which is almost 20% (Section 2.4). The average speedup of 1.85 (stated in Table 7.2) for the IDCT is used for s_i . With the use of Equation 7.2, a speedup for the XviD decoder of 1.10 is estimated. The maximum speedup for IDCT is determined as 1.25.

7.3.2 MPEG-4 decoder speedup

The individual speedups of the hardware kernels have been determined in the last section. With these speedups estimations have been made (using Amdahl's law) of the speedup of the XviD decoder. The estimated speedups will be verified by measuring the execution time of the XviD application for the different hardware configurations. The same test sequences will be used for the benchmarking of the XviD decoder, as for the test sequences that were used when extracting the colorspace data files. In Figure 7.1, the execution times for the different configurations of the XviD decoder for the four reference files are shown. The speedups can be calculated from the different execution times, using the relation $S = \frac{T_{sw}}{T_{hw}}$ as presented in Equation 7.3. The speedups are listed in Table 7.3.

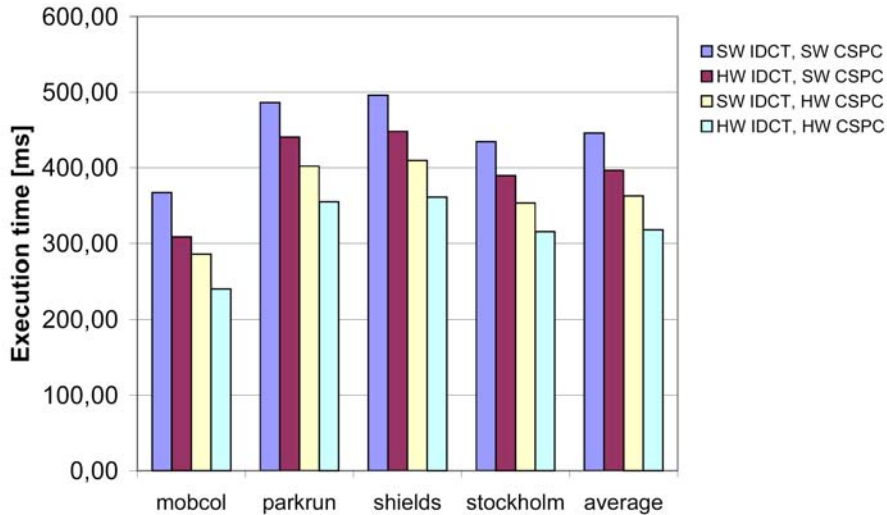


Figure 7.1: The execution times of the different configurations of the decoder.

From the speedups of the four different reference files, the average speedups were calcu-

	mobcol	parkrun	shields	stockholm	average
SW IDCT,SW CSPC	1.00	1.00	1.00	1.00	1.00
SW IDCT,HW CSPC	1.29	1.21	1.21	1.23	1.23
HW IDCT,SW CSPC	1.19	1.10	1.11	1.11	1.12
HW IDCT,HW CSPC	1.53	1.37	1.37	1.38	1.40

Table 7.3: The measured speedup of the XviD codec.

lated as presented on the last column in Table 7.3. The average speedups are compared to the expected speedups and the theoretical speedups. The results are presented in Table 7.4. The expected speedups are based upon the individual speedups of the different kernels (as calculated in Section 7.3.1). The speedup when both kernels are used (the last row of Table 7.4) can be calculated using Equation 7.4. For the theoretical speedup column it is assumed that the execution time of the hardware accelerators equals to zero.

	estimated speedup	measured speedup	maximal speedup
SW IDCT,SW CSPC	1.00	1.00	1.00
SW IDCT,HW CSPC	1.24	1.23	1.36
HW IDCT,SW CSPC	1.10	1.12	1.25
HW IDCT,HW CSPC	1.40	1.40	1.89

Table 7.4: The speedup of the XviD codec.

As can be seen from Table 7.4, the estimated values for the software IDCT and hardware colorspace conversion are good approximations of the measured values. For the hardware IDCT and software CSPC this is not the case. The most likely explanations for this phenomenon is that reference files require more IDCT operations than the files that were used for profiling. In this case, the value of α_i becomes larger resulting in a higher speedup.

Conclusions

The use of a platform that combines a general purpose processor and reconfigurable hardware is assumed to be the answer to the increasingly more demanding multimedia applications in embedded environments. This project aimed to prove the validity of this assumption.

This chapter is organized as follows. In 8.1 an summary of the thesis is presented. The contributions of this thesis are stated in Section 8.2. Section 8.3 concludes by giving recommendations for further research.

8.1 Summary

Chapter 2 introduced the software environment used for the SMOKE project. The compiler tools that were used for software development are part of GNU GCC. The runtime environments that were used to execute of the MPEG-4 decoder are standalone operation, or the Linux operating system. The MPEG-4 decoder that runs on these environments is a modified version of the open-source codec XviD. The XviD codec is a mature product and can easily be ported to different platforms.

The *DAMP software development kit (DAMP SDK)* that supports the standalone execution of the MPEG-4 decoder was presented in Chapter 3. For the DAMP SDK the Newlib C-library was used. This library can easily be ported to different runtime environments, even without operating system. The library can be adapted to the runtime environment by adding support for the platform specific operations, e.g. file reading and writing. A basic filesystem was developed for the DAMP SDK. Furthermore the initialization of the DAMP environment was described.

In Chapter 3 the Linux environment for DAMP was introduced. This chapter described the porting of Linux to the DAMP platform. The configuration of the kernel and the modifications to the kernel sources were presented. Furthermore, a description was given of the bootprocess and the rootfilesystems that were used. In order to access the hardware accelerators, kernel drivers were build. The different methods that are provided by the Linux kernel for hardware access and the ones used for the SMOKE project were described.

In Chapter 5 the hardware platform that was used for SMOKE was introduced. This platform contains the Altera Excalibur device and peripherals that are well suited for Multimedia applications. The Excalibur device integrates an ARM922T processor and an FPGA fabric on a single chip. Two methodologies for communicating between the processor and the hardware were discussed. Finally, the VGA hardware that was used for displaying the images on a VGA monitor was presented. The VGA controller design consists of a hardware design and the accompanying software.

Chapter 6 introduced the kernel of the MPEG-4 decoder that was accelerated

using hardware. The software colorspace conversion and the hardware accelerator were presented. Furthermore, the modifications to the software drivers and the MPEG-4 application were discussed.

The experimental results were presented in Chapter 7. In this chapter it was shown that the speedup of the colorspace conversion kernel is 3.59. This results in a speedup of 1.23 for the entire MPEG-4 decoder. Besides the colorspace conversion kernel, other kernels have been accelerated as part of the SMOKE project. The results of these kernels were also presented. A total speedup of 1.40 for the XviD decoder was realized.

8.2 Main contributions

The main contributions of this thesis can be summarized as follows:

- An accelerator for the colorspace conversion kernel of the MPEG-4 decoder was presented. A detailed discription of hardware/software co-design was given. Besides the hardware implementation of the accelerator a software driver was build for the Linux operating system to give the decoder access to the accelerator hardware. With an average speedup of 3.59, the accelerator can speedup the entire MPEG-4 decoder by 1.23.
- The Linux operating system has been ported the DAMP platform. This platform was used to execute and test the decoder. Furthermore, a number of drivers were created to give the decoder access to the hardware accelerators.
- A special standalone environment for executing applications without the use of an operating system was created. The environment and the accompanying tools are called the DAMP SDK. The DAMP SDK contains of a C-library, initialization code and a filesystem implementation. To support the development a number of tools and a makefile for the build process of the environment were created. The DAMP SDK has been used to verify the operation of the hardware accelerators and the MPEG-4 decoder.
- Several methods have been investigated for communicating between the software and the hardware. For the communication between the VGA and the processor a DMA controller has been designed and implemented.
- A feasibility study of MOLEN architecture implementation on DAMP was performed. A method for implementing MOLEN on DAMP is presented and the advances and disadvantages have been analized.
- A VGA controller has been implemented for the DAMP board. This VGA controller was modified later to facilitate the hardware accelerator functionality.

8.3 Future Work

During the project a number of research directions were abandoned or could not be performed due to limitations in time or hardware. Nevertheless, a deeper investigation of these directions would prove useful in future. The following directions are recommended for future research.

- The speedups of the accelerators and the MPEG-4 decoder should be performed on a revised version of the DAMP board. It is expected that the speedup will increase due to higher memory bandwidth, because the overhead introduced for hardware/software communication is most likely to become smaller.
- The MOLEN architecture could be implemented on the DAMP platform with or without the use of an external memory controller. The impact for implementing MOLEN on the the runtime environment, the hardware drivers and the decoder application should be investigated and reported.
- The VGA controller should be modified to support higher resolution images by placing the framebuffer inside the SDRAM. Placing the framebuffer in the SDRAM can only be performed on a DAMP board that supports highspeed memory transfers. Furthermore, to increase the quality of the images shown on the VGA monitor, the current video DACs should be replaced with three 8 bits DACs.
- The DAMP SDK should be extended to support debugging, networking and provide a shell interface for loading and executing applications.
- In addition to the DAMP SDK, also a light weighted realtime operating system could be ported to the DAMP platform. A good example of such an operating system would be eCos, which is an opensource, royalty-free, real-time operating system intended for embedded applications.

Bibliography

- [1] Werner Almesberger and Hans Lermen, *Using the initial ram disk (initrd)*, 2000.
- [2] ARM, *AMBATM specification rev. 2.0*, May 1999.
- [3] ARMboot, <http://armboot.sourceforge.net/>.
- [4] P. Bourke, *Ycc colour space and image compression*, November 2000, <http://astronomy.swin.edu.au/pbourke/colour/ycc/>.
- [5] Daniel P. Bovet and Marco Cesati, *Understanding the linux kernel*, O'Reilly & Associates, Inc., January 2001.
- [6] GNU Compiler Collection, <http://gcc.gnu.org>.
- [7] Altera Corporation, *Excalibur device overview*, May 2002.
- [8] ———, *Excalibur hardware reference manual*, November 2002.
- [9] ———, *Excalibur solutions using the embedded stripe bridges*, June 2002.
- [10] ———, *Excalibur solutions using the interrupt controller*, December 2002.
- [11] ———, *Booting excalibur devices*, March 2003.
- [12] ———, *Using excalibur dma controllers for video imaging*, February 2003.
- [13] Bill Croft and John Gilmore, *Rfc 951 - bootstrap protocol*, September 1985.
- [14] Guido de Goede, *Accelerating the xvid idct on damp*, Master's thesis, Delft University of Technology, November 2004, p. 48.
- [15] diet libc, <http://www.fefe.de/dietlibc/>.
- [16] J. Eilers, *Damp: Design of the delft altera-based multimedia platform*, Master's thesis, Delft University of Technology, October 2003.
- [17] Memory Technology Device (MTD) Subsystem for Linux, <http://www.linux-mtd.infradead.org/>.
- [18] Technische Universität München Lehrstuhl für Datenverarbeitung, *Mpeg test sequences*, <ftp://ftp.ldv.e/technik.tu-muenchen.de/pub/test-sequences/>.
- [19] Bill Gatliff, *Porting and using newlib in embedded systems*, (2001).
- [20] J. Hofman, G. de Goede, G. Gaydadjiev, and S. Vassiliadis, *Smoke - speeding up mpeg-4 operational kernels on excalibur*, Proceedings ProRISC 2004, November 2004.
- [21] Russell King, *Kernel compilation*, June 2004.

- [22] John Lombardo, *Embedded linux*, New Riders Publishers, June 2002.
- [23] newlib, <http://sources.redhat.com/newlib/>.
- [24] Office of National Drug Control Policy (USA), *Crack*, <http://www.whitehousedrugpolicy.gov/drugfact/crack/>.
- [25] The LART Pages, <http://www.lart.tudelft.nl/>.
- [26] Charles Poynton, *A technical introduction to digital video*, John Wiley & Sons, Inc, 1996.
- [27] ———, *A guided tour of color space*, (1997).
- [28] ———, *Yuv and luminance considered harmful: A plea for precise terminology in video*, (2001).
- [29] The ARM Linux Project, <http://www.arm.linux.org.uk/>.
- [30] Alessandro Rubini and Jonatan Corbet, *Linux device drivers*, second ed., O'Reilly & Associates, Inc., June 2001.
- [31] H. J. Sips and Hai-Xiang Lin, *Lecture slides, introduction to high performance computing*, 2004.
- [32] uClibc, <http://www.uclibc.org/>.
- [33] S. Vassiliadis, S. Wong, and S. D. Cotofana, *The molen $\rho\mu$ -coded processor*, in 11th International Conference on Field-Programmable Logic and Applications (FPL), Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147, August 2001, pp. 275–285.
- [34] David Woodhouse, *Jffs: The journalling flash file system*, October 2001.
- [35] Wookey and Tak-Shing, *Porting the linux kernel to a new arm platform*, 2002.
- [36] XviD, <http://www.xvid.org>.
- [37] W. Zwart, *Tr-damp: Testing and redesigning the delft altera-based multimedia platform*, Master's thesis, Delft University of Technology, October 2003.
- [38] W. Zwart, J. Eilers, G. N. Gaydadjiev, and S. D. Cotofana, *Damp - delft altera-based multimedia platform*, Proceedings ProRISC 2002, November 2002, pp. 587–594.

The character device framework



A.1 Character device driver source

```
#ifndef __KERNEL__
# define __KERNEL__
#endif
#ifndef MODULE
# define MODULE
#endif

#include <linux/module.h>
#include <linux/version.h>

#include <linux/sched.h>
#include <linux/kernel.h> /* printk() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/tqueue.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/arch/irqs.h>

#include <linux/vmalloc.h>
#include <linux/ioctl.h>

int module_major = 0;
MODULE_PARM(module_major, "i");
MODULE_AUTHOR("Jonathan Hofman");

struct file_operations *module_fop_array[]={
    &module_default_fops, /* minor 0 */
    &module_minor1_fops, /* minor 1 */
    &module_minor2_fops, /* minor 2 */
};

/* the device used to store private data */
Module_dev module_minor1_dev =
{
    device_name: "module_minor1",

    /* more private data fields can be placed here */
};

Module_dev module_minor2_dev =
{
    device_name: "module_minor2",

    /* more private data fields can be placed here */
};

/* file handling routines */

/* default routines */
int module_default_open(struct inode *inode, struct file *filp){
    int minor;
    minor = MINOR(inode->i_rdev);
    printk(KERN_INFO "%s: module_default_open is called\n",module_name);

    if(minor > MODULE_MAX_MINOR)
        return -ENODEV;

    /* specify which operations it will perform (control or line_buffer) */
    filp->f_op = module_fop_array[minor];

    return filp->f_op->open(inode, filp); /* dispatch to specific open */
}
```

```

int module_default_release(struct inode *inode, struct file *filp){
    printk(KERN_INFO "%s: module_default_release is called\n",module_name);
    MOD_DEC_USE_COUNT;
    return 0;
}

struct file_operations module_default_fops = {
    open:    module_default_open,
    release: module_default_release,
};

/* minor 1 routines */

int module_minor1_open(struct inode *inode, struct file *filp){
    DEBUG(printk(KERN_INFO "%s: module_minor1_open is called\n",module_name));

    filp->private_data = &module_minor1_dev;

    return 0;
}

int module_minor1_release(struct inode *inode, struct file *filp){
    Module_minor1_dev *dev = (Module_minor1_dev*) filp->private_data;

    printk(KERN_INFO "%s: module_minor1_release is called\n",dev->device_name);

    return damp_vga_fops.release(inode,filp);
}

ssize_t module_minor1_read(struct file *filp, char *buf, size_t count, loff_t *f_pos){
    Module_minor1_dev *dev = (Module_minor1_dev*) filp->private_data;

    printk(KERN_INFO "%s: module_minor1_read is called\n",dev->device_name);

    return 0;
}

ssize_t module_minor1_write(struct file *filp, const char *buf, size_t count,
    loff_t *f_pos){
    Module_minor1_dev *dev = (Module_minor1_dev*) filp->private_data;

    printk(KERN_INFO "%s: module_minor1_write is called\n",dev->device_name);
    return 0;
}

int module_minor1_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg){
    Module_minor1_dev *dev = (Module_minor1_dev*) filp->private_data;

    DEBUG(printk(KERN_INFO "%s: module_minor1_ioctl is called\n",dev->device_name));
}

struct file_operations damp_vga_data_fops = {
    read:    module_minor1_read,
    write:   module_minor1_write,
    open:    module_minor1_open,
    release: module_minor1_release,
    ioctl:   module_minor1_ioctl,
};

/* minor 2 routines */

int module_minor2_open(struct inode *inode, struct file *filp){
    DEBUG(printk(KERN_INFO "%s: module_minor2_open is called\n",module_name));

    filp->private_data = &module_minor2_dev;

    return 0;
}

int module_minor2_release(struct inode *inode, struct file *filp){
    Module_minor2_dev *dev = (Module_minor2_dev*) filp->private_data;

    printk(KERN_INFO "%s: module_minor2_release is called\n",dev->device_name);

    return damp_vga_fops.release(inode,filp);
}

ssize_t module_minor2_read(struct file *filp, char *buf, size_t count, loff_t *f_pos){
    Module_minor2_dev *dev = (Module_minor2_dev*) filp->private_data;

    printk(KERN_INFO "%s: module_minor2_read is called\n",dev->device_name);

    return 0;
}

```

```

ssize_t module_minor2_write(struct file *filp, const char *buf, size_t count,
    loff_t *f_pos){
    Module_minor2_dev *dev = (Module_minor2_dev*) filp->private_data;

    printk(KERN_INFO "%s: module_minor2_write is called\n",dev->device_name);
    return 0;
}

int module_minor2_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg){
    Module_minor2_dev *dev = (Module_minor2_dev*) filp->private_data;

    DEBUG(printk(KERN_INFO "%s: module_minor2_ioctl is called\n",dev->device_name));
}

struct file_operations damp_vga_data_fops = {
    read:    module_minor2_read,
    write:   module_minor2_write,
    open:    module_minor2_open,
    release: module_minor2_release,
    ioctl:   module_minor2_ioctl,
};

/* module initialisation and finit functions */

static int module_init(void){
    int result;

    result = register_chrdev(module_major, module_name, &module_default_fops);
    if (result < 0) {
        printk(KERN_INFO "%s: can't get major number\n",module_name);
        return result;
    }
    if (module_major == 0)
        module_major = result; /* dynamic */

    printk(KERN_INFO "%s: Using major number %i \n",module_name,vga_linebuffer_major);

    SET_MODULE_OWNER(&module_default_fops);

    return 0;
}

static void module_cleanup(void){
    unregister_chrdev(module_major, module_name);
}

int init_module (void) /* Loads a module in the kernel */
{
    printk("Hello kernel \n");
    module_init();
    return 0;
}

void cleanup_module(void) /* Removes module from kernel */
{
    module_cleanup();
    printk("GoodBye Kernel \n");
}

```

A.2 Device driver load script

```

#!/bin/sh
module="module"
device="module"
mode="664"

# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod -f ./${module} $* || exit 1

major='cat /proc/devices | awk "\$2==\"$device\" {print \$1}"'

# Remove stale nodes and replace them, then give gid and perms
# Usually the script is shorter, it's simple that has several devices in it.

rm -f /dev/${device}
rm -f /dev/${device}_1
rm -f /dev/${device}_2

mknod /dev/${device} c $major 0
mknod /dev/${device}_1 c $major 1
mknod /dev/${device}_2 c $major 2

```

```
chmod $mode /dev/${device}
chmod $mode /dev/${device}_1
chmod $mode /dev/${device}_2
```

A.3 Device driver unload script

```
#!/bin/sh
module="module"
device="module"

# invoke rmmmod with all arguments we got
/sbin/rmmmod $module $* || exit 1

# Remove stale nodes
rm -f /dev/${device}
rm -f /dev/${device}_1
rm -f /dev/${device}_2
```

AMBA Slave

B

```
-- Copyright (C) 1991-2002 Altera Corporation
-- Any megafunction design, and related netlist (encrypted or decrypted),
-- support information, device programming or simulation file, and any other
-- associated documentation or information provided by Altera or a partner
-- under Altera's Megafunction Partnership Program may be used only
-- to program PLD devices (but not masked PLD devices) from Altera. Any
-- other use of such megafunction design, netlist, support information,
-- device programming or simulation file, or any other related documentation
-- or information is prohibited for any other purpose, including, but not
-- limited to modification, reverse engineering, de-compiling, or use with
-- any other silicon devices, unless such use is explicitly licensed under
-- a separate agreement with Altera or a megafunction partner. Title to the
-- intellectual property, including patents, copyrights, trademarks, trade
-- secrets, or maskworks, embodied in any such megafunction design, netlist,
-- support information, device programming or simulation file, or any other
-- related documentation or information provided by Altera or a megafunction
-- partner, remains with Altera, the megafunction partner, or their respective
-- licensors. No other licenses, including any licenses needed under any third
-- party's intellectual property, are provided herein.
--
--
-- Synopsis:
--This design implements an AHB slave peripheral which contains a
--simple bank of 8 registers. The register bank is intended to
--interface with a DMA controller and a VGA driver which used in
--conjunction with each other support driving still images and
--video from external SDRAM or any other memory source within
--the system address space.
--
-- Documentation:
--See video_driver.doc for more information on the register map and
--the functions associated with individual registers.
--
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY slave_interface IS
PORT (

-- AHB interface
hresetn : IN std_logic;
hclock : IN std_logic;
hwrite : IN std_logic;
hsel : IN std_logic;
htrns : IN std_logic_vector(1 downto 0);
hsize : IN std_logic_vector(1 downto 0);
hburst : IN std_logic_vector(2 downto 0);
haddress : IN std_logic_vector(31 downto 0);
hwdata : IN std_logic_vector(31 downto 0);
hready : OUT std_logic;
hresp : OUT std_logic_vector(1 downto 0);
hrdata : OUT std_logic_vector(31 downto 0);

-- Inteface to DMA Controller and VGA Driver
status : IN std_logic_vector(31 downto 0);
current_address : IN std_logic_vector(31 downto 0);
buffer_address : OUT std_logic_vector(31 downto 0);
image_dimensions : OUT std_logic_vector(31 downto 0);
control : OUT std_logic_vector(31 downto 0)
);
END slave_interface;

ARCHITECTURE rtl OF slave_interface IS
-- Memory Map
-- ADDRESS NAME DESCRIPTION
-- 0x00 buffer_addr This register contains the address from which the DMA
--should begin an image xfer from
-- 0x04 image_dimensions This address contains the number of lines in the image and
--the numbe of pixels per line
-- 0x08 control_reg This register is used to enable or disable the DMA, and
--clear or enable IRQs
-- 0x0C current_address The current address that the DMA is reading from
-- 0x10 status This register is used to check the current status of the DMA
-- 0x14 reserved Reserved for future use
```

```

-- 0x18 reserved Reserved for future use
-- 0x1C reserved Reserved for future use

SIGNAL buffer_address_reg : std_logic_vector(31 downto 0);
SIGNAL image_dimensions_reg : std_logic_vector(31 downto 0);
SIGNAL control_reg : std_logic_vector(31 downto 0);

SIGNAL internal_write : std_logic;
SIGNAL internal_address : std_logic_vector(2 downto 0);

TYPE state_type IS (address,data);
SIGNAL state : state_type;

BEGIN

-- We are always ready and we always respond with an OKAY responses to the initiating master.
hready <= '1';
hresp <= "00";

-- Create a FSM to control the internal read and write signals to the register bank.
PROCESS(hclock,hresetn)
BEGIN
IF hresetn = '0' THEN
internal_write <= '0';
state <= address;
ELSIF rising_edge(hclock) THEN
CASE state IS
WHEN address =>
IF hsel = '1' AND htrans = "10" THEN
IF hwrite = '1' THEN
internal_write <= '1';
ELSE
internal_write <= '0';
END IF;
state <= data;
ELSE
internal_write <= '0';
state <= address;
END IF;
END IF;
WHEN data =>
-- Remain in data state on burst transfers
IF htrans = "11" THEN
IF hwrite = '1' THEN
internal_write <= '1';
ELSE
internal_write <= '0';
END IF;
state <= data;
ELSE
internal_write <= '0';
state <= address;
END IF;
END IF;
WHEN others =>
internal_write <= '0';
state <= address;
END CASE;
END IF;
END PROCESS;

-- Create the Register Bank
PROCESS(hclock,hresetn)
BEGIN
IF hresetn = '0' THEN
internal_address <=(others => '0');
buffer_address_reg <= (others => '0');
image_dimensions_reg <= (others => '0');
control_reg <= (others => '0');
hrdata <= (others => '0');
ELSIF rising_edge(hclock) THEN
internal_address <= haddress(4 downto 2);
IF internal_write = '1' THEN
CASE internal_address IS
WHEN "000" =>
buffer_address_reg <= hwdata;
WHEN "001" =>
image_dimensions_reg <= hwdata;
WHEN "010" =>
control_reg <= hwdata;
-- Not all of the registers are writeable. This design does not
-- return any errors if the processor tries to write to a non-
-- writeable register. Instead the design will ignore writes to
-- non-writeable registers.
WHEN others =>
null;
END CASE;
END IF;
END IF;

```

```
IF hsel = '1' AND hwrite = '0' THEN
CASE haddress(4 downto 2) IS
WHEN "000" =>
hrdata <= buffer_address_reg;
WHEN "001" =>
hrdata <= image_dimensions_reg;
WHEN "010" =>
hrdata <= control_reg;
WHEN "011" =>
hrdata <= current_address;
WHEN "100" =>
hrdata <= status;
WHEN "101" =>
hrdata <= (others => '0');
WHEN "110" =>
hrdata <= (others => '0');
WHEN "111" =>
hrdata <= (others => '0');
WHEN others =>
hrdata <= (others => '0');
END CASE;
END IF;
END PROCESS;

-- These registers get assigned to output pins because they are used by the DMA controller
-- and the VGA driver
buffer_address <= buffer_address_reg;
image_dimensions <= image_dimensions_reg;
control <= control_reg;

END rtl;
```




The colorspace conversion

C.1 Software colorspace conversion

```
int32_t RGB_Y_tab[256];
int32_t B_U_tab[256];
int32_t G_U_tab[256];
int32_t G_V_tab[256];
int32_t R_V_tab[256];

void
colorspace_init(void)
{
    int32_t i;

    for (i = 0; i < 256; i++) {
        RGB_Y_tab[i] = ((uint16_t) ((1.164) * (1L<<13) + 0.5)) * (i - 16);
        B_U_tab[i] = ((uint16_t) ((2.018) * (1L<<13) + 0.5)) * (i - 128);
        G_U_tab[i] = ((uint16_t) ((0.391) * (1L<<13) + 0.5)) * (i - 128);
        G_V_tab[i] = ((uint16_t) ((0.813) * (1L<<13) + 0.5)) * (i - 128);
        R_V_tab[i] = ((uint16_t) ((1.596) * (1L<<13) + 0.5)) * (i - 128);
    }
}

void yv12_to_bgr_c(uint8_t * x_ptr, int x_stride, uint8_t * y_ptr, uint8_t * u_ptr, uint8_t * v_ptr, int y_stride, int uv_stride, int width, int height,
{
    int fixed_width = (width + 1) & ~1;
    int x_dif = x_stride - (3)*fixed_width;
    int y_dif = y_stride - fixed_width;
    int uv_dif = uv_stride - (fixed_width / 2);
    int x, y;
    if (vflip)
    {
        x_ptr += (height - 1) * x_stride;
        x_dif = -(3)*fixed_width - x_stride;
        x_stride = -x_stride;
    }
    for (y = 0; y < height; y+=(2))
    {
        for (x = 0; x < fixed_width; x+=(2))
        {
            int rgb_y;
            int b_u0 = B_U_tab[ u_ptr[0] ];
            int g_uv0 = G_U_tab[ u_ptr[0] ] + G_V_tab[ v_ptr[0] ];
            int r_v0 = R_V_tab[ v_ptr[0] ];
            rgb_y = RGB_Y_tab[ y_ptr[(0)*y_stride + 0] ];
            x_ptr[(0)*x_stride+(0)] = ((0)>(((255)<((rgb_y + b_u0) >> 13)?(255):
            ((rgb_y + b_u0) >> 13)))?((0):(((255)<((rgb_y + b_u0) >> 13)?(255):((rgb_y + b_u0) >> 13)))));
            x_ptr[(0)*x_stride+(1)] = ((0)>(((255)<((rgb_y - g_uv0) >> 13)?(255):
            ((rgb_y - g_uv0) >> 13)))?((0):(((255)<((rgb_y - g_uv0) >> 13)?(255):((rgb_y - g_uv0) >> 13)))));
            x_ptr[(0)*x_stride+(2)] = ((0)>(((255)<((rgb_y + r_v0) >> 13)?(255):
            ((rgb_y + r_v0) >> 13)))?((0):(((255)<((rgb_y + r_v0) >> 13)?(255):((rgb_y + r_v0) >> 13)))));
            if ((3)>3) x_ptr[(0)*x_stride+(0)] = 0;
            rgb_y = RGB_Y_tab[ y_ptr[(0)*y_stride + 1] ];
            x_ptr[(0)*x_stride+(3)+(0)] = ((0)>(((255)<((rgb_y + b_u0) >> 13)?(255):
            ((rgb_y + b_u0) >> 13)))?((0):(((255)<((rgb_y + b_u0) >> 13)?(255):((rgb_y + b_u0) >> 13)))));
            x_ptr[(0)*x_stride+(3)+(1)] = ((0)>(((255)<((rgb_y - g_uv0) >> 13)?(255):
            ((rgb_y - g_uv0) >> 13)))?((0):(((255)<((rgb_y - g_uv0) >> 13)?(255):((rgb_y - g_uv0) >> 13)))));
            x_ptr[(0)*x_stride+(3)+(2)] = ((0)>(((255)<((rgb_y + r_v0) >> 13)?(255):
            ((rgb_y + r_v0) >> 13)))?((0):(((255)<((rgb_y + r_v0) >> 13)?(255):((rgb_y + r_v0) >> 13)))));
            if ((3)>3) x_ptr[(0)*x_stride+(3)+(0)] = 0;
            rgb_y = RGB_Y_tab[ y_ptr[(1)*y_stride + 0] ];
            x_ptr[(1)*x_stride+(0)] = ((0)>(((255)<((rgb_y + b_u0) >> 13)?(255):
            ((rgb_y + b_u0) >> 13)))?((0):(((255)<((rgb_y + b_u0) >> 13)?(255):((rgb_y + b_u0) >> 13)))));
            x_ptr[(1)*x_stride+(1)] = ((0)>(((255)<((rgb_y - g_uv0) >> 13)?(255):
            ((rgb_y - g_uv0) >> 13)))?((0):(((255)<((rgb_y - g_uv0) >> 13)?(255):((rgb_y - g_uv0) >> 13)))));
            x_ptr[(1)*x_stride+(2)] = ((0)>(((255)<((rgb_y + r_v0) >> 13)?(255):
            ((rgb_y + r_v0) >> 13)))?((0):(((255)<((rgb_y + r_v0) >> 13)?(255):((rgb_y + r_v0) >> 13)))));
            if ((3)>3) x_ptr[(1)*x_stride+(0)] = 0;
            rgb_y = RGB_Y_tab[ y_ptr[(1)*y_stride + 1] ];
            x_ptr[(1)*x_stride+(3)+(0)] = ((0)>(((255)<((rgb_y + b_u0) >> 13)?(255):
            ((rgb_y + b_u0) >> 13)))?((0):(((255)<((rgb_y + b_u0) >> 13)?(255):((rgb_y + b_u0) >> 13)))));
            x_ptr[(1)*x_stride+(3)+(1)] = ((0)>(((255)<((rgb_y - g_uv0) >> 13)?(255):
            ((rgb_y - g_uv0) >> 13)))?((0):(((255)<((rgb_y - g_uv0) >> 13)?(255):((rgb_y - g_uv0) >> 13)))));
        }
    }
}
```

```

x_ptr[(1)*x_stride+(3)+(2)] = ((0)<((255)<(rgb_y + r_v0) >> 13)?(255):
((rgb_y + r_v0) >> 13))?(0):((255)<(rgb_y + r_v0) >> 13)?(255):((rgb_y + r_v0) >> 13));
if ((3)>3) x_ptr[(1)*x_stride+(3)+(0)] = 0;;
x_ptr += (2)*(3);
y_ptr += (2);
u_ptr += (2)/2;
v_ptr += (2)/2;
}
  x_ptr += x_dif + (2 -1)*x_stride;
  y_ptr += y_dif + (2 -1)*y_stride;
  u_ptr += uv_dif + ((2/2)-1)*uv_stride;
  v_ptr += uv_dif + ((2/2)-1)*uv_stride; //hierdoor moet uv_stride 160 opgegeven worden ipv 80 zodat hij wel steeds opschuift
}
}

```

C.2 Hardware colorspace conversion

```

LIBRARY ieee,work;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.VGA_constants.ALL;

entity yuv2rgb is
port
(
  enable : in std_logic;
  clk : in std_logic;
  rgb_yuv : in std_logic;
  Y : in std_logic_vector(7 downto 0);
  U : in std_logic_vector(7 downto 0);
  V : in std_logic_vector(7 downto 0);
  R : out std_logic_vector(2 downto 0);
  G : out std_logic_vector(2 downto 0);
  B : out std_logic_vector(1 downto 0)
);
end yuv2rgb;

architecture behaviour of yuv2rgb is
  signal reg_R,reg_G : std_logic_vector(2 downto 0);
  signal reg_B : std_logic_vector(1 downto 0);
begin

process(enable,rgb_yuv,Y,U,V)
  variable int_y,int_u,int_v: integer;
  variable int_r,int_g,int_b,tmp: integer;
begin
  int_y := to_integer(unsigned(Y)) - 16;
  int_u := to_integer(unsigned(U)) - 128;
  int_v := to_integer(unsigned(V)) - 128;

  int_b := (149*int_y + 258*int_u)/128;
  int_g := (149*int_y - 104*int_v - 50*int_u)/128;
  int_r := (149*int_y + 204*int_v)/128;

if rgb_yuv = '1' then
  reg_R <= Y(7 downto 5);
  reg_G <= Y(4 downto 2);
  reg_B <= Y(1 downto 0);
else
  if int_r < 0 then
    reg_R <= (others => '0');
  elsif int_r > 255 then
    reg_R <= (others => '1');
  else
    reg_R <= std_logic_vector(to_unsigned(int_r,8))(7 downto 5);
  end if;

  if int_g < 0 then
    reg_G <= (others => '0');
  elsif int_g > 255 then
    reg_G <= (others => '1');
  else
    reg_G <= std_logic_vector(to_unsigned(int_g,8))(7 downto 5);
  end if;

  if int_b < 0 then
    reg_B <= (others => '0');
  elsif int_b > 255 then
    reg_B <= (others => '1');
  else
    reg_B <= std_logic_vector(to_unsigned(int_b,8))(7 downto 6);
  end if;
end if;

if enable = '0' then

```

```
R <= (others => '0');
G <= (others => '0');
B <= (others => '0');
elsif clk = '1' and clk'event then
    R <= reg_R;
    G <= reg_G;
    B <= reg_B;
end if;
end process;
end behaviour;
```


The DMA controller VHDL

source

D

```
-----
-- Jonathan Hofman, 2004 --
-- Support for the SPLIT and RETRY responses has been disabled. --
-- It can easily be disabled by uncommenting the corresponding --
-- constructions. --
-----

library ieee;
use ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;

library work;
use work.AMBA_definitions.all;

entity VGA_DMA_controller is
port (

-- AHB interface
reset_n : in std_logic;
clk : in std_logic;

hwrite : out std_logic;
htrans : out std_logic_vector(1 downto 0);
hsize : out std_logic_vector(2 downto 0);
hburst : out std_logic_vector(2 downto 0);
haddr : out std_logic_vector(31 downto 0);
hwdata : out std_logic_vector(31 downto 0);
hresp : in std_logic_vector(1 downto 0);
hgrant : in std_logic;
hbusreq : out std_logic;
hlock : out std_logic;
hprot : out std_logic_vector(3 downto 0);
hready : in std_logic;
hrdata : in std_logic_vector(31 downto 0);

-- Inteface to linebuffer
linebuffer_data : out std_logic_vector(31 downto 0);
linebuffer_address : out std_logic_vector(5 downto 0);
linebuffer_wr_en : out std_logic;
linebuffer_wrclock : out std_logic;

-- Interface to video controller
dma_startaddress : in std_logic_vector(31 downto 0);
dma_size : in std_logic_vector(5 downto 0); -- size in words! - we can transfer 1k words (4k bytes)
dma_start : in std_logic;
dma_done : out std_logic;
dma_error : out std_logic
);
end VGA_DMA_controller;

architecture behaviour of VGA_DMA_controller is
signal int_linebuffer_address_delayed : std_logic_vector(5 downto 0);
signal int_linebuffer_address : std_logic_vector(5 downto 0);
signal int_ahb_address : std_logic_vector(31 downto 0);
signal int_linebuffer_data : std_logic_vector(31 downto 0);
signal int_dma_size : std_logic_vector(5 downto 0);
signal ahb_done : std_logic;
signal ahb_error : std_logic;
signal ahb_reset : std_logic;
signal dma_size_enable : std_logic;

type ahb_state_type is (asReset,asIdle,asNonseq,asSeq,asLastTransfer,asDone,asStorePending,asError);
signal ahb_state, old_ahb_state : ahb_state_type;

type control_state_type is (csIdle,csTransfer,csDone);
signal control_state, next_control_state : control_state_type;

-- function declarations --
function boundry1k return boolean is
begin
return int_ahb_address(9 downto 2) = "11111111"; --check 1k boundry
end function boundry1k;
```

```

function lasttransfer return boolean is
begin
    return int_ahb_address = dma_startaddress + ((int_dma_size - 1) & "00");
end function lasttransfer;

procedure inc_ahb_address is
begin
    int_ahb_address <= int_ahb_address + 4;
end procedure inc_ahb_address;

procedure dec_ahb_address is
begin
    int_ahb_address <= int_ahb_address - 4;
end procedure dec_ahb_address;

procedure store_data is
begin
    int_linebuffer_address_delayed <= int_linebuffer_address_delayed + 1;
    linebuffer_wr_en <= '1';
    int_linebuffer_data <= hrddata;
end procedure store_data;

procedure no_store_data is
begin
    linebuffer_wr_en <= '0';
end;

procedure reset_addresscounters is
begin
    int_linebuffer_address_delayed <= (others => '0');
    int_ahb_address <= dma_startaddress;
end;

begin
    -- always transfer 32 bits
    hsize <= H_32BIT;
    -- we always perform an unspecified length burst
    hburst <= H_INCR;
    -- the linebuffer clock is fed by the AMBA bus clock
    linebuffer_wrclock <= clk;
    -- the ahb_error_signal is directly fed to the output
    dma_error <= ahb_error;
    -- we are always reading
    hwrite <= '0';
    -- we will never read so we don't care
    hwdata <= (others => 'X');
    -- we will perform locked transfers for now
    hlock <= '1';
    --
    hprot <= HPROT_DATA or HPROT_PRIV or HPROT_BUF or HPROT_CACHE;

control_fsm: process(clk,reset_n,control_state,ahb_done,dma_start)
begin
    if reset_n = '0' then
        control_state <= csIdle;
    elsif clk'event and clk = '1' then
        control_state <= next_control_state;
    end if;

    case control_state is
        when csIdle =>
            -- output signals
            dma_done <= '0';
            ahb_reset <= '1';
            dma_size_enable <= '1';
        -- new state
        if dma_start = '1' then
            next_control_state <= csTransfer;
        else
            next_control_state <= csIdle;
        end if;
        when csTransfer =>
            -- output signals
            dma_done <= '0';
            ahb_reset <= '0';
            dma_size_enable <= '0';
        -- new state
        if ahb_done = '1' then
            next_control_state <= csDone;
        else
            next_control_state <= csTransfer;
        end if;
    end case;

when csDone =>
    -- output signals
    dma_done <= '1';
    ahb_reset <= '1';

```

```

        dma_size_enable <= '0';
    -- new state
    if dma_start = '0' then
        next_control_state <= csIdle;
    else
        next_control_state <= csDone;
    end if;
when others =>
    null;
end case;
end process;

ahb_fsm: process(clk,ahb_reset)
begin
    if ahb_reset = '1' then
        ahb_state <= asReset;
    elsif clk'event and clk = '1' then
        case ahb_state is
            when asReset=>
                -- new state
                reset_addresscounters;
                ahb_state <= asIdle;
                when asIdle =>
                    -- new state
                    if hready = '0' or hgrant = '0' then
                        ahb_state <= asIdle;
                    elsif hready = '1' and hgrant = '1' then
                        ahb_state <= asNonSeq;
                    else
                        ahb_state <= asError;
                    end if;
                    no_store_data;
                    when asNonSeq =>
                        -- new state
                        if hready = '0' then
                            ahb_state <= asNonSeq;
                        elsif hready = '1' and lasttransfer then
                            ahb_state <= asLastTransfer;
                        elsif hready = '1' and not(lasttransfer) and ((hgrant = '0') or boundry1k) then
                            ahb_state <= asStorePending;
                            inc_ahb_address;
                        elsif hready = '1' and not(lasttransfer) and hgrant = '1' and not(boundry1k) then
                            ahb_state <= asSeq;
                            inc_ahb_address;
                        end if;
                    end if;
                    no_store_data;
                    when asSeq =>
                        -- new state
                        if hresp = H_ERROR then
                            ahb_state <= asError;
                            no_store_data;
                        --elsif hresp = H_SPLIT or hresp = H_RETRY then
                        -- ahb_state <= asIdle;
                        -- dec_ahb_address;
                        -- no_store_data;
                        elsif hresp = H_OK and hready = '0' then
                            ahb_state <= asSeq;
                            no_store_data;
                        elsif hresp = H_OK and hready = '1' and lasttransfer then
                            ahb_state <= asLastTransfer;
                            store_data;
                        elsif hresp = H_OK and hready = '1' and not(lasttransfer) and ((hgrant = '0') or boundry1k) then
                            ahb_state <= asStorePending;
                            inc_ahb_address;
                            store_data;
                            elsif hresp = H_OK and hready = '1' and not(lasttransfer) and (hgrant = '1') and not (boundry1k) then
                                ahb_state <= asSeq;
                                inc_ahb_address;
                                store_data;
                            end if;
                        when asStorePending =>
                            -- new state
                        if hresp = H_ERROR then
                            ahb_state <= asError;
                            no_store_data;
                        --elsif hresp = H_SPLIT or hresp = H_RETRY then
                        -- ahb_state <= asIdle;
                        -- dec_ahb_address;
                        -- no_store_data;
                        elsif hresp = H_OK and hready = '0' then
                            ahb_state <= asStorePending;
                            no_store_data;
                        elsif hresp = H_OK and hready = '1' and hgrant = '1' then
                            ahb_state <= asNonSeq;
                            store_data;
                        elsif hresp = H_OK and hready = '1' and hgrant = '0' then

```

```

    ahb_state <= asIdle;
    store_data;
end if;
when asLastTransfer =>
-- new state
if hresp = H_ERROR then
    ahb_state <= asError;
    no_store_data;
--elsif hresp = H_SPLIT or hresp = H_RETRY then
--    ahb_state <= asIdle;
--    dec_ahb_address;
--    no_store_data;
elsif hresp = H_OK and hready = '0' then
    ahb_state <= asLastTransfer;
    no_store_data;
elsif hresp = H_OK and hready = '1' then
    ahb_state <= asDone;
    store_data;
end if;
when asDone =>
-- new state
    ahb_state <= asDone;
    no_store_data;
    when asError =>
-- new state
        ahb_state <= asError;
no_store_data;
    when others =>
null;
end case;
end if;
end process;

decode_ahb_fsm: process(ahb_state)
begin
--calculate the output signals depending on the states (outside the clock event)
case ahb_state is
    when asReset=>
-- output signals
        hbusreq <= '0';
        htrans <= H_IDLE;
        ahb_done <= '0';
        ahb_error <= '0';
--    hburst <= (others => 'X');
    when asIdle =>
-- output signals
        hbusreq <= '1';
        htrans <= H_IDLE;
        ahb_done <= '0';
--    hburst <= (others => 'X');
    when asNonSeq =>
-- output signals
        hbusreq <= '1';
        htrans <= H_NONSEQ;
        ahb_done <= '0';
        ahb_error <= '0';
--    if boundry1k then
--        hburst <= H_SINGLE;
--    else
--        hburst <= H_INCR;
    end if;
    when asSeq =>
-- output signals
        hbusreq <= '1';
        htrans <= H_SEQ;
        ahb_done <= '0';
        ahb_error <= '0';
--    if boundry1k then
--        hburst <= H_SINGLE;
--    else
--        hburst <= H_INCR;
    end if;
    when asStorePending =>
-- output signals
        hbusreq <= '1';
        htrans <= H_IDLE;
        ahb_done <= '0';
        ahb_error <= '0';
--    hburst <= H_INCR;
    when asLastTransfer =>
-- output signals
        hbusreq <= '0';
        htrans <= H_IDLE;
        ahb_done <= '0';
        ahb_error <= '0';
--    hburst <= (others => 'X');
    when asDone =>

```

```
-- output signals
hbusreq <= '0';
htrans <= H_IDLE;
ahb_done <= '1';
ahb_error <= '0';
-- hburst <= (others => 'X');
when asError =>
-- output signals
hbusreq <= '0';
htrans <= H_IDLE;
ahb_done <= '1';
ahb_error <= '1';
when others =>
null;
end case;
end process;

delayed_linebufferaddress: process(clk,reset_n)
begin
if reset_n = '0' then
int_linebuffer_address <= (others => '0');
elsif clk'event and clk = '1' then
int_linebuffer_address <= int_linebuffer_address_delayed;
end if;
end process;

registered_dma_size: process(clk,reset_n)
begin
if reset_n = '0' then
int_dma_size <= (others => '0');
elsif clk'event and clk = '1' then
if dma_size_enable = '1' then
int_dma_size <= dma_size;
else
int_dma_size <= int_dma_size;
end if;
end if;
end process;

linebuffer_address <= int_linebuffer_address;
haddr <= int_ahb_address;
linebuffer_data <= int_linebuffer_data;

end behaviour;
```


The VGA software driver for Linux



E.1 The Linux VGA driver (H-file)

```
#ifndef _DAMP_VGA_H_
#define _DAMP_VGA_H_

#include <linux/ioctl.h>

#undef DEBUGING
//#define DEBUGING 1

#ifdef DEBUGING
#define DEBUG(val) val
#else
#define DEBUG(val)
#endif
#define DEBUG(val)

const char* module_name = "vga_driver";

/* memory locations */
/*
 * Here's our address range, and a place to store the ioremap'd base.
 */
#define VGA_CONTROL_BASE (PLD_BASE)
#define VGA_CONTROL_SIZE (0x1000) /* 0x400 words = 0x1000 bytes */
#define VGA_CONTROL_END (VGA_LINE_BUFFER_BASE + VGA_LINE_BUFFER_SIZE - 1)

#define VGA_FRAME_BASE (SRAM_BASE)
#define VGA_FRAME_SIZE (0x8000) /* 0x400 words = 0x1000 bytes */
#define VGA_FRAME_END (VGA_LINE_BUFFER_BASE + VGA_LINE_BUFFER_SIZE - 1)
#define VGA_LINEWIDTH 160
#define VGA_LINEHEIGHT 120
#define YUV_FRAME_SIZE ((VGA_LINEWIDTH*VGA_LINEHEIGHT*3)/2)
#define RGB8_FRAME_SIZE (VGA_LINEWIDTH*VGA_LINEHEIGHT)
#define RGB24_FRAME_SIZE ((VGA_LINEWIDTH*VGA_LINEHEIGHT)*3)

#define VGA_DMA_CONTROL *((unsigned int*) VGA_CONTROL_BASE + 0)
#define VGA_DMA_START_ADDRESS_REG *((unsigned int*) VGA_CONTROL_BASE + 1)
#define VGA_DMA_SIZE_REG *((unsigned int*) VGA_CONTROL_BASE + 2)

#define VGA_CTRL_MODE_DISABLE 0x0
#define VGA_CTRL_MODE_RGB 0xF
#define VGA_CTRL_MODE_YUV 0x2

#define VGA_IOC_MAGIC 'k'
#define VGA_IOC_MAXNR 3

#define VGA_IOC_YUV _IO(VGA_IOC_MAGIC, 0)
#define VGA_IOC_RGB8 _IO(VGA_IOC_MAGIC, 1)
#define VGA_IOC_RGB24 _IO(VGA_IOC_MAGIC, 2)

/* filehandlers implemented by the driver for diferent minor */

extern struct file_operations damp_vga_fops; /* default */
extern struct file_operations damp_vga_control_fops; /* minor 0 */
extern struct file_operations damp_vga_data_fops; /* minor 1 */
extern struct file_operations damp_vga_data_raw_fops; /* minor 2 */
extern struct file_operations damp_vga_data_yuv_fops; /* minor 3 */
extern struct file_operations damp_vga_data_rgb8_fops; /* minor 4 */
extern struct file_operations damp_vga_data_rgb24_fops; /* minor 5 */
extern struct file_operations damp_vga_data_bgr24_fops; /* minor 6 */

/* here the device info is stored */
typedef struct damp_vga_dev {
    char* device_name;
    char* buffer;
    unsigned int size;
    unsigned int pos;
    struct semaphore sem; /* mutual exclusion semaphore */
} Damp_vga_dev;
```

```

/* auxiliary functions */
void write_RGB(void* addr, int R, int G, int B);
unsigned long int RGB24(unsigned int red, unsigned int green, unsigned int blue);
unsigned char RGB8(unsigned char red, unsigned char green, unsigned char blue);

#define DAMP_VGA_MAX_MINOR 6
#define DAMP_VGA_IRQ IRQ_PLD1
#endif _DAMP_VGA_H_

```

E.2 The Linux VGA driver (C-file)

```

#ifndef __KERNEL__
# define __KERNEL__
#endif
#ifndef MODULE
# define MODULE
#endif

#include <linux/module.h>
#include <linux/version.h>

#include <linux/sched.h>
#include <linux/kernel.h> /* printk() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/tqueue.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/arch/irqs.h>

#include <linux/vmalloc.h>
#include <linux/ioctl.h>

#include "vga_driver.h"
// #include "bitmap.h"
#include "penguin.yuv.h"

int vga_linebuffer_major = 0;
MODULE_PARM(vga_linebuffer_major, "i");
MODULE_AUTHOR("Jonathan Hofman");

struct file_operations *damp_vga_fop_array[] = {
    &damp_vga_control_fops, /* minor 0 */
    &damp_vga_data_fops, /* minor 1 */
    &damp_vga_data_raw_fops, /* minor 2 */
    &damp_vga_data_yuv_fops, /* minor 3 */
    &damp_vga_data_rgb8_fops, /* minor 4 */
    &damp_vga_data_rgb24_fops, /* minor 5 */
    &damp_vga_data_bgr24_fops, /* minor 6 */
};

/* the device structures used to store private data */

Damp_vga_dev damp_vga_control_dev = {
    device_name: "damp_vga_control",
    buffer: NULL,
};

Damp_vga_dev damp_vga_data_dev = {
    device_name: "damp_vga_data",
    buffer: NULL,
};

Damp_vga_dev damp_vga_data_raw_dev = {
    device_name: "damp_vga_data_raw",
    buffer: NULL,
};

Damp_vga_dev damp_vga_data_rgb24_dev = {
    device_name: "damp_vga_data_rgb24",
    buffer: NULL,
};

Damp_vga_dev damp_vga_data_bgr24_dev = {
    device_name: "damp_vga_data_bgr24",
    buffer: NULL,
};

unsigned long int read_io(void* addr){
    return readl(addr);
}

```

```

}

void write_io(void* addr, unsigned long int data){
    writel(data, addr);
}

void zero_framebuffer(void){
    int i;
    unsigned long int *io_ptr = (unsigned long int*) VGA_FRAME_BASE;

    for(i=0;i<(VGA_FRAME_SIZE);i=i+4)
    {
        write_io(io_ptr,0x0);
        io_ptr++;
    }
}

void write_RGB(void* addr, int R, int G, int B){
    RGB8(R,G,B);
    write_io(addr,RGB8(R,G,B));
}

void write_RGB_to_buf(void* addr, int R, int G, int B){
    RGB8(R,G,B);
    *((unsigned long int*)addr) = RGB8(R,G,B);
}

unsigned long int RGB24(unsigned int red, unsigned int green, unsigned int blue){
    unsigned long int tmp;
    tmp = (unsigned long int) red;
    tmp = (tmp << 8) | green;
    tmp = (tmp << 8) | blue;

    return tmp;////((unsigned long int) red && 0xFF) << 16) | ((green & 0xFF) << 8) | (blue & 0xFF);
}

unsigned char RGB8(unsigned char red, unsigned char green, unsigned char blue){
    return (red & 0xE0) | ((green & 0xE0) >> 3) | ((blue & 0xC0) >> 6);
}

/* file handling routines */

/* default routines */

int damp_vga_open(struct inode *inode, struct file *filp){
    int minor;

    DEBUG(printf(KERN_INFO "%s: damp_vga_open is called\n",module_name));

    minor = MINOR(inode->i_rdev);

    if(minor > DAMP_VGA_MAX_MINOR)
        return -ENODEV;

    /* specify which operations it will perform (control or line_buffer) */
    filp->f_op = damp_vga_fop_array[minor];

    return filp->f_op->open(inode, filp); /* dispatch to specific open */
}

int damp_vga_release(struct inode *inode, struct file *filp){
    DEBUG(printf(KERN_INFO "%s: damp_vga_release is called\n",module_name));
    MOD_DEC_USE_COUNT;
    return 0;
}

struct file_operations damp_vga_fops = {
    open:    damp_vga_open,
    release: damp_vga_release,
};

/* data routines */

int vga_data_open(struct inode *inode, struct file *filp){
    DEBUG(printf(KERN_INFO "%s: vga_linebuffer_open is called\n",module_name));

    filp->private_data = &damp_vga_data_dev;
    return 0;
}

int vga_data_release(struct inode *inode, struct file *filp){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    unsigned int count;

    DEBUG(printf(KERN_INFO "%s: vga_linebuffer_release is called\n",dev->device_name));
    return damp_vga_fops.release(inode,filp);
}

```

```

}

ssize_t vga_data_read(struct file *filp, char *buf, size_t count, loff_t *f_pos){
    Damp_vga_dev *dev = filp->private_data;
    unsigned int cnt;
    size_t result = 0;

    DEBUG(printk(KERN_INFO "%s: vga_data_read is called\n",dev->device_name));

    /* --- SEM --- we will use a semaphore here so always exit by freeing it */
    {
        if (down_interruptible(&(dev->sem))) /* don't let other processes disturbe the read */
            return -ERESTARTSYS;

        if (*f_pos >= dev->size){
            goto finally;
        }
        if ((*f_pos + count) > dev->size){
            cnt = dev->size;
        } else {
            cnt = (*f_pos + count);
        }

        if (copy_to_user(buf,dev->buffer+*f_pos,cnt))
        {
            result = -EFAULT;
            goto finally;
        }
        *f_pos += cnt;
        result = cnt;
    }
    finally: up(&dev->sem);
    /* --- END SEM --- */

    return result;
}

ssize_t vga_data_write(struct file *filp, const char *buf, size_t count,
    loff_t *f_pos){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    char* dptr;
    unsigned int size;
    size_t result = 0;

    DEBUG(printk(KERN_INFO "%s: vga_data_write is called\n",dev->device_name));

    /* --- SEM --- we will use a semaphore here so always exit by freeing it */
    {
        if (down_interruptible(&(dev->sem))) /* don't let other processes disturbe the read */
            return -ERESTARTSYS;

        if (*f_pos != dev->pos){
            goto finally;
        }

        /* TODO: improve the rather inefficient way for allocating memory */
        if (dev->size < (*f_pos + count)){
            size = (*f_pos + count);
            dptr = kmalloc(size,GFP_KERNEL);
            if(!dptr){
                goto finally;
            }
            dev->buffer = memcpy(dptr,dev->buffer,dev->size);
            dev->size = size;
        }

        if (copy_from_user(dev->buffer+dev->pos,buf,count)){
            result = -EFAULT;
            goto finally;
        }
        *f_pos += count;
        dev->pos = *f_pos;
        result = count;
    }
    finally: up(&dev->sem);
    /* --- END SEM --- */

    return result;
}

struct file_operations damp_vga_data_fops = {
    read:    vga_data_read,
    write:   vga_data_write,
    open:    vga_data_open,
    release: vga_data_release,
};

```

```

/* raw data routines */

int vga_data_raw_open(struct inode *inode, struct file *filp){
    DEBUG(printk(KERN_INFO "%s: vga_data_raw_open is called\n",module_name));

    filp->private_data = &damp_vga_data_raw_dev;
    return 0;
}

int vga_data_raw_release(struct inode *inode, struct file *filp){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;

    DEBUG(printk(KERN_INFO "%s: vga_data_raw_release is called\n",dev->device_name));
    return damp_vga_fops.release(inode,filp);
}

ssize_t vga_data_raw_read(struct file *filp, char *buf, size_t count, loff_t *f_pos){
    Damp_vga_dev *dev = filp->private_data;
    unsigned int cnt;
    size_t result = 0;

    DEBUG(printk(KERN_INFO "%s: vga_data_raw_read is called\n",dev->device_name));

    /* --- SEM --- we will use a semaphore here so always exit by freeing it */
    {
        if (down_interruptible(&(dev->sem))) /* don't let other processes disturbe the read */
            return -ERESTARTSYS;

        if (*f_pos >= VGA_FRAME_SIZE){
            goto finally;
        }
        if ((*f_pos + count) > VGA_FRAME_SIZE){
            cnt = VGA_FRAME_SIZE - *f_pos;
        } else{
            cnt = (*f_pos + count);
        }

        if (copy_to_user(buf,(unsigned char*)VGA_FRAME_BASE+*f_pos,cnt)){
            result = -EFAULT;
            goto finally;
        }
        *f_pos += cnt;
        result = cnt;
    }
    finally: up(&dev->sem);
    /* --- END SEM --- */

    return result;
}

ssize_t vga_data_raw_write(struct file *filp, const char *buf, size_t count,
    loff_t *f_pos)
{
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    unsigned int cnt;
    size_t result = 0;

    DEBUG(printk(KERN_INFO "%s: vga_data_raw_write is called\n",dev->device_name));

    /* --- SEM --- we will use a semaphore here so always exit by freeing it */
    {
        if (down_interruptible(&(dev->sem))) /* don't let other processes disturbe the read */
            return -ERESTARTSYS;

        if (*f_pos >= VGA_FRAME_SIZE){
            result = -EFAULT;
            goto finally;
        } else if ((*f_pos + count) > VGA_FRAME_SIZE){
            cnt = VGA_FRAME_SIZE - *f_pos;
        } else {
            cnt = count;
        }

        if (copy_from_user(((char*)VGA_FRAME_BASE)+*f_pos,buf,cnt)){
            result = -EFAULT;
            goto finally;
        }
        *f_pos += cnt;
        result = cnt;
    }
    finally: up(&dev->sem);
    DEBUG(printk(KERN_INFO "%s: result = %u\n",dev->device_name,result));
    /* --- END SEM --- */
    return result;
}

```

```

}

struct file_operations damp_vga_data_raw_fops = {
    read:    vga_data_raw_read,
    write:   vga_data_raw_write,
    open:    vga_data_raw_open,
    release: vga_data_raw_release,
};

/* data routines for rgb8 */

int vga_data_rgb8_open(struct inode *inode, struct file *filp){
    filp->private_data = &damp_vga_data_raw_dev;
    DEBUG(printk(KERN_INFO "%s: vga_data_rgb8_open is called\n",module_name));

    /* Put driver into RGB 8 mode */
    VGA_DMA_CONTROL = VGA_CTRL_MODE_RGB;

    return 0;
}

int vga_data_rgb8_release(struct inode *inode, struct file *filp){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;

    DEBUG(printk(KERN_INFO "%s: vga_data_rgb8_release is called\n",dev->device_name));
    return damp_vga_fops.release(inode,filp);
}

int vga_data_rgb8_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    size_t result = 0;
    DEBUG(printk(KERN_INFO "%s: vga_data_rgb8_ioctl is called\n",dev->device_name));

    if(_IOC_TYPE(cmd) != VGA_IOC_MAGIC){
        return -ENOTTY;
    }
    if(_IOC_NR(cmd) > VGA_IOC_MAXNR){
        return -ENOTTY;
    }

    /* should perform some memory checking */
    switch(cmd){
        case VGA_IOC_RGB8:{
            VGA_DMA_CONTROL = VGA_CTRL_MODE_RGB;
            if (copy_from_user(((char*)VGA_FRAME_BASE), (char*)arg,RGB8_FRAMESIZE)){
                result = -EFAULT;
                return result;
            }
        } break;
        default: {
            return -ENOTTY;
        }
    }
    return 1;
}

struct file_operations damp_vga_data_rgb8_fops = {
    write:    vga_data_raw_write,
    open:     vga_data_rgb8_open,
    release:  vga_data_rgb8_release,
    ioctl:    vga_data_rgb8_ioctl,
};

/* data routines for vy12 */

int vga_data_yuv_open(struct inode *inode, struct file *filp){
    filp->private_data = &damp_vga_data_raw_dev;
    DEBUG(printk(KERN_INFO "%s: vga_data_yuv_open is called\n",module_name));

    /* Put driver into YUV mode */
    VGA_DMA_CONTROL = VGA_CTRL_MODE_YUV;

    return 0;
}

int vga_data_yuv_release(struct inode *inode, struct file *filp){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;

    DEBUG(printk(KERN_INFO "%s: vga_data_yuv_release is called\n",dev->device_name));
    return damp_vga_fops.release(inode,filp);
}

int vga_data_yuv_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    size_t result = 0;
    DEBUG(printk(KERN_INFO "%s: vga_data_yuv_ioctl is called\n",dev->device_name));

```

```

    if(_IOC_TYPE(cmd) != VGA_IOCTL_MAGIC){
        return -ENOTTY;
    }
    if(_IOC_NR(cmd) > VGA_IOCTL_MAXNR){
        return -ENOTTY;
    }

    /* should perform some memory checking */
    switch(cmd){
    case VGA_IOCTL_YUV:{
        VGA_DMA_CONTROL = VGA_CTRL_MODE_YUV;
        if (copy_from_user(((char*)VGA_FRAME_BASE), (char*)arg, YUV_FRAMESIZE)){
            result = -EFAULT;
            return result;
        }
    } break;
    default: {
        return -ENOTTY;
    }
    }
    return 1;
}

struct file_operations damp_vga_data_yuv_fops = {
    write:    vga_data_raw_write,
    open:    vga_data_yuv_open,
    release:  vga_data_yuv_release,
    ioctl:   vga_data_yuv_ioctl,
};

/* data routines for rgb24 */

int vga_data_rgb24_open(struct inode *inode, struct file *filp){
    Damp_vga_dev *dev = &damp_vga_data_rgb24_dev;

    filp->private_data = dev;

    DEBUG(printk(KERN_INFO "%s: vga_data_rgb24_open is called\n", module_name));

    dev->buffer = kmalloc(RGB24_FRAMESIZE, GFP_KERNEL);
    if(!dev->buffer){
        DEBUG(printk(KERN_INFO "%s: could not allocate temp buf.\n", dev->device_name));
        return -EFAULT;
    }

    /* Put driver into RGB mode */
    VGA_DMA_CONTROL = VGA_CTRL_MODE_RGB;

    return 0;
}

int vga_data_rgb24_release(struct inode *inode, struct file *filp)
{
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;

    kfree(dev->buffer);
    DEBUG(printk(KERN_INFO "%s: vga_data_rgb24_release is called\n", dev->device_name));
    return damp_vga_fops.release(inode, filp);
}

ssize_t vga_data_rgb24_write(struct file *filp, const char *buf, size_t count,
    loff_t *f_pos){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    unsigned int cnt;
    size_t result = 0;

    DEBUG(printk(KERN_INFO "%s: vga_data_rgb24_write is called\n", dev->device_name));

    /* --- SEM --- we will use a semaphore here so always exit by freeing it */
    {
        int i;
        if (down_interruptible(&(dev->sem))) /* don't let other processes disturbe the read */
            return -ERESTARTSYS;

        if (*f_pos >= RGB24_FRAMESIZE){
            result = -EFAULT;
            goto finally;
        }else if ((*f_pos + count) > RGB24_FRAMESIZE){
            cnt = RGB24_FRAMESIZE - *f_pos;
        }else{
            cnt = count;
        }

        if (copy_from_user(dev->buffer+*f_pos, buf, cnt)){
            DEBUG(printk(KERN_INFO "%s: failed to copy memory from userspace.\n", dev->device_name));
            result = -EFAULT;
        }
    }
}

```

```

        goto finally;
    }

    *f_pos += cnt;
    result = cnt;
    if (*f_pos >= RGB24_FRAME_SIZE){
        for(i=0;i<(RGB24_FRAME_SIZE/3);i++){
            *(((char*)VGA_FRAME_BASE)+i) = RGB8(dev->buffer[3*i],dev->buffer[3*i+1],dev->buffer[3*i+2]);
        }
    }
}
finally: up(&dev->sem);

DEBUG(printk(KERN_INFO "%s: result = %u\n",dev->device_name,result));
/* --- END SEM --- */
return result;
}

int vga_data_rgb24_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    size_t result = 0;
    DEBUG(printk(KERN_INFO "%s: vga_data_rgb24_ioctl is called\n",dev->device_name));

    if(_IOC_TYPE(cmd) != VGA_IOCTL_MAGIC){
        return -ENOTTY;
    }
    if(_IOC_NR(cmd) > VGA_IOCTL_MAXNR){
        return -ENOTTY;
    }

    /* should perform some memory checking */
    switch(cmd){
        case VGA_IOCTL_RGB24:{
            int i;

            VGA_DMA_CONTROL = VGA_CTRL_MODE_RGB;
            if (copy_from_user(dev->buffer,(char*)arg,RGB24_FRAME_SIZE)){
                result = -EFAULT;
                return result;
            }

            for(i=0;i<(RGB24_FRAME_SIZE/3);i++){
                *(((char*)VGA_FRAME_BASE)+i) = RGB8(dev->buffer[3*i],dev->buffer[3*i+1],dev->buffer[3*i+2]);
            }

        } break;
        default: {
            return -ENOTTY;
        }
    }
    return 1;
}

struct file_operations damp_vga_data_rgb24_fops = {
    write:    vga_data_rgb24_write,
    open:    vga_data_rgb24_open,
    release:  vga_data_rgb24_release,
    ioctl:   vga_data_rgb24_ioctl,
};

/* data routines for bgr24 */

int vga_data_bgr24_open(struct inode *inode, struct file *filp){
    Damp_vga_dev *dev = &damp_vga_data_bgr24_dev;

    filp->private_data = dev;

    DEBUG(printk(KERN_INFO "%s: vga_data_bgr24_open is called\n",module_name));

    dev->buffer = kmalloc(RGB24_FRAME_SIZE,GFP_KERNEL);
    if(!dev->buffer){
        DEBUG(printk(KERN_INFO "%s: could not allocate temp buf.\n",dev->device_name));
        return -EFAULT;
    }

    /* Put driver into RGB mode */
    VGA_DMA_CONTROL = VGA_CTRL_MODE_RGB;

    return 0;
}

int vga_data_bgr24_release(struct inode *inode, struct file *filp)
{
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;

    kfree(dev->buffer);
}

```

```

    DEBUG(printk(KERN_INFO "%s: vga_data_bgr24_release is called\n",dev->device_name));
    return damp_vga_fops.release(inode,filp);
}

ssize_t vga_data_bgr24_write(struct file *filp, const char *buf, size_t count,
    loff_t *f_pos)
{
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    unsigned int cnt;
    size_t result = 0;

    DEBUG(printk(KERN_INFO "%s: vga_data_bgr24_write is called\n",dev->device_name);)

    /* --- SEM --- we will use a semaphore here so always exit by freeing it */
    {
        int i;
        if (down_interruptible(&(dev->sem))) /* don't let other processes disturbe the read */
            return -ERESTARTSYS;

        if (*f_pos >= RGB24_FRAMESIZE){
            result = -EFAULT;
            goto finally;
        }else if ((*f_pos + count) > RGB24_FRAMESIZE){
            cnt = RGB24_FRAMESIZE - *f_pos;
        }else{
            cnt = count;
        }

        DEBUG(printk(KERN_INFO "%s: count = %u; fpos = %Li; cnt = %u\n",dev->device_name,count,*f_pos,cnt);)

        if (copy_from_user(dev->buffer+*f_pos,buf,cnt)){
            DEBUG(printk(KERN_INFO "%s: failed to copy memory from userspace.\n",dev->device_name);)
            result = -EFAULT;
            goto finally;
        }

        DEBUG(printk(KERN_INFO "%s: count = %u; fpos = %Li; cnt = %u\n",dev->device_name,count,*f_pos,cnt);)

        *f_pos += cnt;
        result = cnt;
        if (*f_pos >= RGB24_FRAMESIZE){
            for(i=0;i<(RGB24_FRAMESIZE/3);i++){
                *(((char*)VGA_FRAME_BASE+i) = RGB8(dev->buffer[3*i+2],dev->buffer[3*i+1],dev->buffer[3*i]);
            }
        }
        finally: up(&dev->sem);

        DEBUG(printk(KERN_INFO "%s: result = %u\n",dev->device_name,result));
        /* --- END SEM --- */
        return result;
    }
}

int vga_data_bgr24_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg){
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    size_t result = 0;
    DEBUG(printk(KERN_INFO "%s: vga_data_bgr24_ioctl is called\n",dev->device_name));

    if(_IOC_TYPE(cmd) != VGA_IOCTL_MAGIC){
        return -ENOTTY;
    }
    if(_IOC_NR(cmd) > VGA_IOCTL_MAXNR){
        return -ENOTTY;
    }

    /* should perform some memory checking */
    switch(cmd){
        case VGA_IOCTL_RGB24:{
            int i;

            VGA_DMA_CONTROL = VGA_CTRL_MODE_RGB;
            if (copy_from_user(dev->buffer,(char*)arg,RGB24_FRAMESIZE)){
                result = -EFAULT;
                return result;
            }

            for(i=0;i<(RGB24_FRAMESIZE/3);i++){
                *(((char*)VGA_FRAME_BASE+i) = RGB8(dev->buffer[3*i+2],dev->buffer[3*i+1],dev->buffer[3*i]);
            }

        } break;
        default: {
            return -ENOTTY;
        }
    }
}

```

```

return 1;
}

struct file_operations damp_vga_data_bgr24_fops = {
    write:   vga_data_bgr24_write,
    open:    vga_data_bgr24_open,
    release: vga_data_bgr24_release,
    ioctl:   vga_data_bgr24_ioctl,
};

/* control routines */

int vga_control_open(struct inode *inode, struct file *filp){
    DEBUG(printk(KERN_INFO "%s: vga_linebuffer_open is called\n",module_name));

    filp->private_data = &damp_vga_control_dev;

    /* TODO: we should initialise the control registers here */
    return 0;
}

int vga_control_release(struct inode *inode, struct file *filp)
{
    DEBUG(printk(KERN_INFO "%s: vga_linebuffer_release is called\n",module_name));
    return 0;
}

ssize_t vga_control_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
    Damp_vga_dev *dev = (Damp_vga_dev*) filp->private_data;
    size_t result = 0;

    DEBUG(printk(KERN_INFO "%s: vga_control_read is called\n",dev->device_name));

    /* --- SEM --- we will use a semaphore here so always exit by freeing it */
    {
        if (down_interruptible(&(dev->sem))) /* don't let other processes disturb the read */
            return -ERESTARTSYS;

        if ((*f_pos != dev->pos) || (count != VGA_CONTROL_SIZE)) /* refactor this code dev->pos is probably obsolete */
        {
            goto finally;
        }

        if (copy_to_user(buf,(void*)VGA_CONTROL_BASE,count))
        {
            result = -EFAULT;
            goto finally;
        }
    }
    finally: up(&dev->sem);
    /* --- END SEM --- */

    return result;
}

ssize_t vga_control_write(struct file *filp, const char *buf, size_t count,
    loff_t *f_pos){
    Damp_vga_dev *dev = filp->private_data;
    size_t result = 0;

    DEBUG(printk(KERN_INFO "%s: vga_control_write is called\n",dev->device_name));

    /* --- SEM --- we will use a semaphore here so always exit by freeing it */
    {
        if (down_interruptible(&(dev->sem))) /* don't let other processes disturb the read */
            return -ERESTARTSYS;

        if ((*f_pos != 0) || (count != VGA_CONTROL_SIZE))
        {
            goto finally;
        }

        if (copy_from_user((void*)VGA_CONTROL_BASE,buf,count))
        {
            result = -EFAULT;
            goto finally;
        }
        *f_pos += count;
        result = count;
    }
    finally: up(&dev->sem);
    /* --- END SEM --- */
    return result;
}

```

```

struct file_operations damp_vga_control_fops = {
    read:    vga_control_read,
    write:   vga_control_write,
    open:    vga_control_open,
    release: vga_control_release,
};

/* module initialisation and finit functions */

static int damp_vga_init(void){
    int result;

    result = register_chrdev(vga_linebuffer_major, module_name, &damp_vga_fops);
    if (result < 0) {
        printk(KERN_INFO "%s: can't get major number\n",module_name);
        return result;
    }
    if (vga_linebuffer_major == 0)
        vga_linebuffer_major = result; /* dynamic */

    printk(KERN_INFO "%s: Using major number %i \n",module_name,vga_linebuffer_major);
    if (check_mem_region(VGA_FRAME_BASE, VGA_FRAME_SIZE)){
        printk(KERN_INFO "%s: requested io region unavailable \n",module_name);
        return -EFAULT;
    }

    if (!request_mem_region (VGA_FRAME_BASE,VGA_FRAME_SIZE,module_name)){
        printk (KERN_ERR "%s: failed to request region\n",module_name);
        return -EBUSY;
    }

    /* initialise semaphores in device structs */
    sema_init(&damp_vga_data_dev.sem, 1);
    sema_init(&damp_vga_data_raw_dev.sem, 1);
    sema_init(&damp_vga_data_rgb24_dev.sem, 1);
    sema_init(&damp_vga_data_bgr24_dev.sem, 1);
    sema_init(&damp_vga_control_dev.sem, 1);

    SET_MODULE_OWNER(&damp_vga_fops);

    {
        VGA_DMA_START_ADDRESS_REG = 0x20000000;
        VGA_DMA_SIZE_REG = 160/4;
        VGA_DMA_CONTROL = 0x2;
    }

    zero_framebuffer();
    return 0;
}

static void damp_vga_cleanup(void)
{
    VGA_DMA_CONTROL = 0x00000000;
    release_mem_region (VGA_FRAME_BASE,VGA_FRAME_SIZE);
    unregister_chrdev(vga_linebuffer_major, module_name);
}

int init_module (void) /* Loads a module in the kernel */
{
    DEBUG(printk("Hello kernel \n"));
    damp_vga_init();
    return 0;
}

void cleanup_module(void) /* Removes module from kernel */
{
    damp_vga_cleanup();
    DEBUG(printk("GoodBye Kernel \n"));
}

```

E.3 VGA driver load script

```

#!/bin/sh
module="vga_driver"
device="vga_driver"
mode="664"

# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod -f ./${module} ${*} || exit 1

major='cat /proc/devices | awk "\$2==\"$device\" {print \$1}"'

```

```
# Remove stale nodes and replace them, then give gid and perms
# Usually the script is shorter, it's simple that has several devices in it.
```

```
rm -f /dev/${device}
rm -f /dev/${device}_data
rm -f /dev/${device}_raw
rm -f /dev/${device}_yuv
rm -f /dev/${device}_rgb8
rm -f /dev/${device}_rgb24
rm -f /dev/${device}_bgr24

mknod /dev/${device} c $major 0
mknod /dev/${device}_data c $major 1
mknod /dev/${device}_raw c $major 2
mknod /dev/${device}_yuv c $major 3
mknod /dev/${device}_rgb8 c $major 4
mknod /dev/${device}_rgb24 c $major 5
mknod /dev/${device}_bgr24 c $major 6

chmod $mode /dev/${device}
chmod $mode /dev/${device}_data
chmod $mode /dev/${device}_raw
chmod $mode /dev/${device}_yuv
chmod $mode /dev/${device}_rgb8
chmod $mode /dev/${device}_rgb24
chmod $mode /dev/${device}_bgr24
```

E.4 VGA driver unload script

```
#!/bin/sh
module="vga_driver"
device="vga_driver"

# invoke rmmmod with all arguments we got
/sbin/rmmmod $module $* || exit 1

# Remove stale nodes
rm -f /dev/${device}
rm -f /dev/${device}_data
rm -f /dev/${device}_raw
rm -f /dev/${device}_yuv
rm -f /dev/${device}_rgb8
rm -f /dev/${device}_rgb24
rm -f /dev/${device}_bgr24
```

F

The VFS sources

F.1 VFS

```
#include <vfs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*
This file contains the abstraction layer between the
physical filesystem and the application library. */

/*
Will not be able to mount filesystems, can only use one (and of course
the terminal)

int mount_fs(fs_ops_t* fs_ops){
}

void unmount_fs(int fs_number){
}

//void open
*/

#define FD_START 1000

linkedlist_t *file_list;
int fd_count;
file_ops_t *root_fs_ops;

void vfs_init(void){
    nr_txchar('z');
    file_list = fidslist_create();
    nr_txchar('y');
}

void vfs_finit(void){
    fidslist_destroy(file_list);
}

/* fids (file identifier structure) functions */

fids_t* fids_create(int fd,const char* path, int flags, int mode){
    fids_t *result;
    result = (fids_t*)malloc(sizeof(fids_t));
    if(!result){
        nr_printf("fids_create: error could not allocate fids memory.");
        return NULL;
    }

    result->fd = fd;
    result->path = (char*)malloc(strlen(path)+1);

    if(!result->path){
        nr_printf("fids_create: error could not allocate path memory.");
        return NULL;
    }

    strcpy(result->path,path);
    result->flags = flags;
    result->mode = mode;
    result->file_ops = root_fs_ops;
    result->offset = 0;
    return result;
};

void fids_destroy(fids_t *fids){
    if(!fids){
        nr_printf("fids_destroy: trying to free a NULL\r\n");
    }
}
```

```

    free(fids->path);
    free(fids);
}

int get_free_fd(void);

/* fids (file identifier structure) list functions */
linkedlist_t* fidslist_create(void){
return linkedlist_create();
}

void fidslist_destroy(linkedlist_t *list){
linkedlist_destroy(list);
}

/* testen of nested function */
fids_t* fidslist_get_item_via_fd(linkedlist_t *list, int fd){
int fidlist_is_fd(void *data){
return fd == ((fids_t*)data)->fd;
}

return (fids_t*)linkedlist_search(list,fidlist_is_fd);
}

int fidslist_is_item(linkedlist_t *list, fids_t *fids){
int fidlist_is_item(void *data){
return ((fids_t*)data) == fids;
}

return linkedlist_search(list,fidlist_is_item) != NULL;
}

void fidslist_add(linkedlist_t *list, fids_t *item){
linkedlist_add(list,item);
}

void fidslist_remove(linkedlist_t *list, fids_t *item){
linkedlist_remove(list,item);
}

int get_free_fd(void){
/* when fd_count overflows we introduce a major bug!! Need to fix this in the future */
int result;
if(fd_count<FD_START){
fd_count = FD_START;
}
result = fd_count++;
return result;
}

/* vfs basic routines */

extern void dump_memory(const char *ptr, int size);

int vfs_open(const char *path, int flags, int mode){
int result;
fids_t *fids;

fids = fids_create(get_free_fd(),path,flags,mode);
if(!fids){
return -1;
}
result = fids->file_ops->open(fids);
if(result > 0){
fidslist_add(file_list,fids);
} else {
fids_destroy(fids);
}

return result;
}

int vfs_close(int fd){
int result = -1;
fids_t *fids;

/* set up errno for lib */
fids = fidslist_get_item_via_fd(file_list,fd);
if(fids){
result = fids->file_ops->close(fids);
if(result > 0){
fidslist_remove(file_list,fids);
fids_destroy(fids);
}
}
}

```

```

}
return result;
}

long vfs_write(int fd, const char *ptr, int len){
fids_t *fids;

fids = fidslist_get_item_via_fd(file_list,fd);
if(fids){
return fids->file_ops->write(fids,ptr,len);
} else {
/* set up errno for lib */
return -1;
}
}

long vfs_read(int fd, char *ptr, int len){
fids_t *fids;

fids = fidslist_get_item_via_fd(file_list,fd);
if(fids){
return fids->file_ops->read(fids,ptr,len);
} else {
/* set up errno for lib */
return -1;
}
}

int vfs_register_fd(int fd, file_ops_t *fops){
// int result;
fids_t *fids;

fids = fids_create(fd,"",0,0);
if(!fids){
return -1;
}

fids->file_ops = fops;
fidslist_add(file_list,fids);

return 1;
}

void vfs_register_root_fs(file_ops_t *fops){
root_fs_ops = fops;
}

```

F.2 Terminal driver

```

#include <vfs.h>
#include "terminal_driver.h"

file_ops_t terminal_fops = {
open: terminal_open,
close: terminal_close,
write: terminal_write,
read: terminal_read,
};

void terminal_init(void){
vfs_register_fd(0, &terminal_fops);
vfs_register_fd(1, &terminal_fops);
vfs_register_fd(2, &terminal_fops);
vfs_register_fd(3, &terminal_fops);
}

void terminal_finit(void){
}

int terminal_open(fids_t *fids){
return 1;
}

int terminal_close(fids_t *fids){
return -1;
}

long terminal_write(fids_t *fids, const char *ptr, int len){
int x=len;
while (x > 0){
if(*ptr == '\n'){
nr_txchar ('\r');
}
}
}

```

```

nr_txchar (*ptr++);
x--;
}
return len - x;
}

long terminal_read(fids_t *fids, char *ptr, int len){
int x = len;
while (x>0){
*ptr++ = nr_rxchar ();
x--;
}
if (x < 0){
return -1;
}

/* x == len is not an error, at least if we want feof() to work. */
return len - x;
}

```

F.3 memfs

```

/*
Jonathan Hofman 2004

This file gives a limited (memory) filesystem for embedded purposes.
*/

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

linkedlist_t *memfile_list;

file_ops_t memfs_fops = {
open: memfs_open,
close: memfs_close,
write: memfs_write,
read: memfs_read,
};

void memfs_init(void){
memfile_list = memfilelist_create();
}

void memfs_finit(void){
memfilelist_destroy(memfile_list);
}

linkedlist_t* memfilelist_create(void){
return linkedlist_create();
}

void memfilelist_destroy(linkedlist_t *list){
linkedlist_destroy(list);
}

memfile_t* memfilelist_get_item_via_fd(linkedlist_t *list, int fd){
int memfile_is_fd(void *data){
return fidslist_get_item_via_fd(((memfile_t*)data)->fids_list,fd) != NULL;
}

return (memfile_t*)linkedlist_search(list,memfile_is_fd);
}

memfile_t* memfilelist_get_item_via_fids(linkedlist_t *list, fids_t *fids){
int memfile_is_fd(void *data){
return fidslist_is_item(((memfile_t*)data)->fids_list,fids);
}

return (memfile_t*)linkedlist_search(list,memfile_is_fd);
}

memfile_t* memfilelist_get_item_via_path(linkedlist_t *list, const char *path){
int memfile_is_path(void *data){
return !strcmp(((memfile_t*)data)->path,path);
}

return (memfile_t*)linkedlist_search(list,memfile_is_path);
}

```

```

void memfilelist_add(linkedlist_t *list, memfile_t *item){
linkedlist_add(list,item);
}

void memfilelist_remove(linkedlist_t *list, memfile_t *item){
linkedlist_remove(list,item);
}

memfile_t* memfile_create(linkedlist_t *list, char* path){
memfile_t* result;

result = (memfile_t*)malloc(sizeof(memfile_t));
if(!result){
nr_printf("memfile_create: could not allocate memfile");
return NULL;
}

result->fids_list = fidslist_create();
if(!result->fids_list){
nr_printf("memfile_create: could not allocate fidslist");
free(result);
return NULL;
}

result->path = (char*)malloc(strlen(path));
if(!result->path){
nr_printf("memfile_create: could not allocate fidslist");
fidslist_destroy(result->fids_list);
free(result);
return NULL;
}
strcpy(result->path,path);

memfilelist_add(list,result);
result->usage_count = 0;
result->size = 0;

return result;
}

void memfile_destroy(linkedlist_t *list, memfile_t *item){
if(!item){
nr_printf("memfile_destroy: called with item == NULL");
return;
}
memfilelist_remove(list,item);
fidslist_destroy(item->fids_list);
free(item->path);
free(item);
}

int memfs_open(fids_t *fids){
memfile_t* fileptr;

nr_printf("memfs_open: fids = %s, %i, %li, %lx\nr", fids->path,fids->fd,fids->offset,(long)fids->priv);
fileptr = memfilelist_get_item_via_path(memfile_list,fids->path);

if(!(fids->mode | __SWR | __SAPP | __SRD)){
nr_printf("memfs_open: invalid mode!\r\n");
}

if(fids->flags & __SWR){
if(!fileptr){
fileptr = memfile_create();
if(!fileptr){
//nr_txchar("+");
nr_printf("memfs_open: fileptr is NULL\r\n");
return -EIO;
} else {
fileptr->path = (char*)malloc(strlen(fids->path));
strcpy(fileptr->path,fids->path);
memfilelist_add(memfile_list,fileptr);
}
}
if(fids->flags & __SAPP){
fids->offset = fileptr->size - 1;
}
if(fids->flags & __SRD){
if(!fileptr){
return -ENOENT;
}
}
}
}

```

```

/* TODO: handle different file modes (r,w,rb,wb,r+,w+,append enz.) */

if(!fileptr){
    nr_printf("memfs_open: file not found.\r\n");
    fileptr = memfile_create(memfile_list,fids->path);
    nr_printf("memfs_open: memfile created at %lx.\r\n", (long)fileptr);
    if(!fileptr){
        nr_printf("memfs_open: fileptr is NULL.\r\n");
        errno = EIO;
        return -1;
    };
}

fids->priv = fileptr;

fileptr->usage_count++;
fidslist_add(fileptr->fids_list,fids);
return fids->fd;
}

int memfs_close(fids_t *fids){
    memfile_t* fileptr;

    fileptr = (memfile_t*)fids->priv;
    if(!fileptr){
        nr_printf("memfs_close: memfile is NULL\r\n");
        errno = ENOENT;
        return -1;
    }

    /*TODO: checken of that fids nog steeds deel uit maakt van de list, usage count mag niet kleiner dan 0 */
    if(!fidslist_is_item(fileptr->fids_list,fids)){
        nr_printf("memfs_close: Trying to delete a fids wich does not exist\r\n");
        return -1;
    }

    if(!(fileptr->usage_count > 0)){
        nr_printf("memfs_close: Trying to decrease usage count below zero.\r\n");
        return -1;
    }

    fileptr->usage_count--;
    fidslist_remove(fileptr->fids_list,fids);
    return 0;
}

extern void dump_memory(const char *ptr, int size);

long memfs_read(fids_t *fids, char *ptr, int len){
    memfile_t* fileptr;
    int result;

    nr_printf("memfs_read: fids = %s, %i, %li, %lx\r\n", fids->path,fids->fd,fids->offset, (long)fids->priv);
    fileptr = (memfile_t*)fids->priv;
    if(!fileptr){
        errno = EIO;
        return -1;
    }

    nr_printf("memfs_read: memfile = %s, %i, %lx\r\n", fileptr->path,fileptr->size, (long)fileptr->start_mem);
    if(fids->offset >= fileptr->size){
        /*TODO: check how to return end of file */
        errno = EOF;
        return -1;
    } else if(fids->offset+len > fileptr->size){
        result = fileptr->size - fids->offset;
    } else {
        result = len;
    }

    /* overlappende stukken ??, memmove? */
    memcpy(ptr, (char*)(fileptr->start_mem) + fids->offset, result);
    fids->offset += result;
    return result;
}

long memfs_write(fids_t *fids, const char *ptr, int len){
    memfile_t* fileptr;

    nr_printf("memfs_write: fids = %s, %i, %li, %lx\r\n", fids->path,fids->fd,fids->offset, (long)fids->priv);

    dump_memory(ptr, len);
    fileptr = (memfile_t*)fids->priv;
    if(!fileptr){
        fileptr = memfilelist_get_item_via_fd(memfile_list,fids->fd);
    }
}

```

```
nr_printf("memfile: %lx\r\n",fileptr);
if(!fileptr){
nr_printf("memfs_write: fileptr is null\r\n");
errno = ENOENT;
return -1;
}
}

nr_printf("memfs_write: before resize\r\n");
if(fids->offset+len > fileptr->size){
void* new_loc;
new_loc = realloc(fileptr->start_mem,fids->offset+len);
if(!new_loc){
errno = ENOENT;
return -1;
}
fileptr->start_mem = new_loc;
fileptr->size = fids->offset+len;
}

nr_printf("memfs_write: before memcpy\r\n");
memcpy((char*)(fileptr->start_mem) + fids->offset,ptr,len);
fids->offset += len;
nr_printf("memfs_write: memfile = %s, %i, %lx\r\n", fileptr->path,fileptr->size,(long)fileptr->start_mem);
nr_printf("memfs_write: before return\r\n");
return len;
}
```


Curriculum Vitae



Jonathan Hofman was born in Gouda, the Netherlands, on April 8, 1981. He attended the Coenecoopcollege high school in Waddinxveen, from which he graduated in 1999. He took the exams in the subjects: Dutch, English, Mathematics, Physics, Chemistry, Economics, Business Economics and Social study. In the same year, he was admitted to Electrical Engineering faculty of the Delft University of Technology in the Netherlands.

In 2002, he joined the Computer Engineering laboratory, led by Prof. Stamatis Vassiliadis, to start his Msc graduation project under the supervision of Ir. Georgi Gaydadjiev. His thesis is titled *Speeding up MPEG-4 colorspace conversion*. His research interests include: embedded operating systems, embedded software design, reconfigurable hardware.