Integration of a Motion-JPEG video decoder
a practical study

Frédéric Pétrot, Mounir Benabdenbi, and Adrien Prost-Boucle

October 19, 2011

Contents

Chapter 1

Presentation

The goal of this activity is to discover the different steps of a MP-SoC hardware and software design processus through the integration of a Motion-JPEG decoder application on a multiprocessor hardware platform.

1.1 Introdution

Three steps will be covered. First, we will get more familiar with the SOCLib cycle-accurate/bit-accurate hardware simulation environment by connecting several devices to a pre-existing hardware platform. Then, we will try to understand the hardware requirement in order to execute a parallelized version of our MJPEG decoder and modify the previous platform accordingly. This will also be the occasion to start working with the operating system. Finally, we will go deeper into the operating systems intrinsics and use this knowledge to enhance existing drivers and write new ones.

1.2 Work schedule

1. Hardware simulation - part 1 (3 courses)
    - Get familiar with the SOCLib environment
    - Add new devices
    - Validate the new devices with small C functions
2. Hardware simulation - part 2 (2 courses)
    - Understand the requirements of a parallel architecture
    - Validate your platform with a parallel version of the MJPEG
    - Compare several executions of the MJPEG with different parameters
3. Operating system (2 courses)
    - Understand the driver intrinsics
    - Write a framebuffer driver
    - Validate with the MJPEG application
4. Optimization (3 courses)
    - Add one or several DMA devices in the parallel architecture
    - Add DMA support into the communication drivers
    - Validate with the MJPEG application

Reporting After each part and at the end of the activity, a preliminary report has to be handed over. Although only the last report will be taken into account for graduation, you are expected to take into account your previous mistakes. **Complete or partial duplications of previous reports would seriously impact your final grade.**

Chapter 2

Reporting

This chapter presents, in a few words, the challenges involved in the different proposed activities. Please read carefully, as hints on how your intermediate and final reports must be constructed are given.

2.1 Basic rules

- **DO NOT INCLUDE SOURCE CODE** in your report.
- If you feel that your algorithm needs to be illustrated, **USE PSEUDO-CODE**.
- **DO NOT RECOPY THE SUBJECT**.
- For your final report, **DO NOT MERELY COPY AND PASTE PREVIOUS REPORTS**.

2.2 Activity 1 — Hardware simulation, part 1

This activity insists on your ability to understand existing foreign source code, extend it, and validate your extensions. Consequently, in your report, you need to emphasize on your understanding of the problem at hands and your solution to this problem.

2.3 Activity 2 — Hardware simulation, part 2

This activity insists on your ability to run benchmarks and process their results, as well as your understanding of the problematic linked to hardware multi-processor platforms. It also calls upon your skills at thinking for yourself and drawing your own conclusion. Consequently, your analysis will have to be organized by theme (such as parallel performances, latency, etc.), and for each of them you will have to discuss the results, emit hypothesis, and draw a conclusion.

2.4 Activity 3 — Operating System

This activity insists on your ability to quickly understand a broad software architecture and decide where and how your own contribution fits. Hence, your report must contains details of your understanding of the operating system architecture and how your driver fits in.

2.5 Activity 4 — Optimization

This activity calls upon your ability to design efficient hardware/software interfaces that can benefit both hardware cost and software performance. In your report, you will promote your hardware design choices and highlight the modifications you brought to the original device driver. You will also show how these modifications improve the overall performances, and discuss the obtained results.

Chapter 3

Activity 1 — Hardware simulation, part 1

In this activity, you will learn how to define and extend an hardware simulation platform using the SoCLib environment. You will also learn how to manipulate hardware devices only with a few lines of C code.

3.1 Hardware platform

The first hardware platform you are going to work with is really simple. It contains several MIPS R3000 processors, interfaced with a data cache and an instruction cache. It also contains a memory controller, a terminal emulation device, a system timer, a system mailbox, and an advanced interrupt controller. These components are interconnected with a generic network-on-chip, called the generic micro-network.

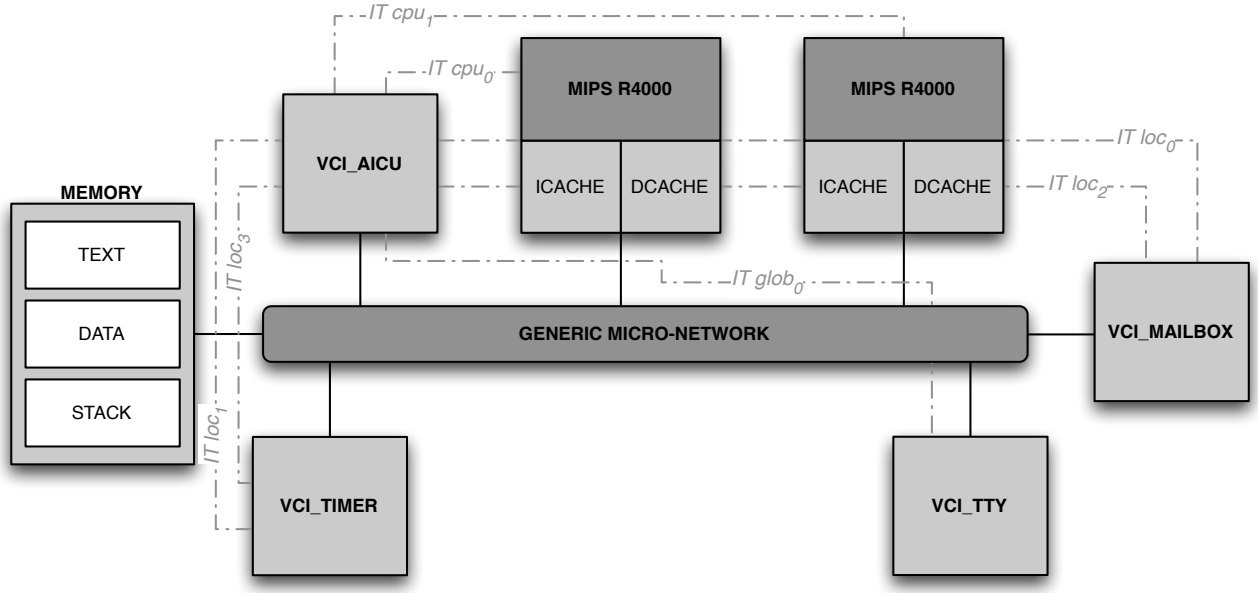


Figure 3.1: The original platform

Files The files related to this platform are located here: `/softslin/TPSoC3A/Platforms/SystemPlatform`. The actual SoCLib implementation is contained in the file `top.cpp`. It is highly advised to carefully read and understand how this file has been written, and how the different components interact with each other.

WARNING Before starting to work with SoCLib or, in the future, any kind of software components, you need to source the `install.sh` located here: `/softslin/TPSoC3A`.

NOTA BENE You are going to work with a new architecture of SystemC components. As with any new architecture, you are expected to follow the **RTFM** rule. **RTFM** stands for **Read The Freaking Manual**. The documentation regarding the SoCLib components is located at the following address: `https://www.soclib.`

`fr`. If you are not familiar with a standard tool such as (and not limited to) `make`, `gcc`, `gdb`, you need to refer to appropriate sources of documentation (`man`, `google`) **on your own**.

3.2 First steps

1. Create a work directory wherever you want in your home folder.
2. Copy the folder `/softslin/TPSoC3A/Platforms/SystemPlatform` in this directory.
3. Also, copy the folder `/softslin/TPSoC3A/Applications/SimpleApplication` in this directory.
4. In the `SimpleApplication` directory, type `source ./install.sh` and then `apes-compose`.
5. In the `SystemPlatform` directory, type `make`.
6. Finally, type `./simulation.x ../SimpleApplication/SimpleApp`.

3.3 Default hardware devices

The initial platform already contains several hardware devices: a terminal emulation device, a system timer, a system mailbox, and an advanced interrupt controller. These components are configured as follows:

1. The terminal emulator has a base address of `0xC1000000`, a segment size of `0x40`, and an interrupt line connected to the global interrupt line 0.
2. The system timer has a base address of `0xC2000000`, a segment size of `0x100`, and an interrupt line connected to the local interrupt line 1.
3. The system mailbox has a base address of `0xC3000000`, a segment size of `0x400`, and an interrupt line connected to the local interrupt line 0.
4. The advanced interrupt controller has a base address of `0xC4000000`, a segment size of `0x1000`. Documentation concerning this component can be found here:
   `http://tima-sls.imag.fr/www/research/soclib/vci_aicu/`.

Your first job will be to write series of small applications **that validate** the following devices: the terminal emulator, the system timer, and the mailbox. To do so, copy the `SimpleApplication` and modify the source files accordingly. Your application needs to validate each function of your devices. For the timer and the mailbox, basic functions are already present in the HAL presented in appendix D. You are vividly encouraged to use theses functions in your own applications.

3.4 Media hardware devices

The devices present by default in the platform are not sufficient to properly execute the parallel MJPEG application. Two extra components need to be added :

1. A block device controller, with a base address at `0xC5000000`, a segment size of `0x100`, and an interrupt line connected to the global interrupt line 1.
2. A framebuffer controller, with a base address at `0xC6000000` and a segment size of `0x100000`.

Your second job will be to integrate these new components into the basic platform. In the previous section, you had to implement small applications to validate existing hardware devices. Do the same to validate your own devices. Once you have finished the integration and the validation of each component, you can try to validate your final platform with the MJPEG application, located here:
`/softslin/TPSoC3A/Applications/ParallelMjpeg`.

NOTA BENE Please refer to appendix A for information concerning the integration of new components, the execution of the simulator, and the debug of an application.

Chapter 4

Activity 2 — Hardware simulation, part 2

In this activity, you will study and learn the requirements of a Synchronous Multi-Processor (SMP) architecture, hardware-wise and software-wise.

4.1 Hardware platform

The platform you need to start with is depicted figure 3.1. This is the platform you should have got to at the end of the previous activity. If your platform has not reached its final state yet, you can continue to work on it for a little while. If at mid-session, you are still not done, you **must** stop working on your platform and start this activity with a fresh, functional architecture. Ask your teacher.

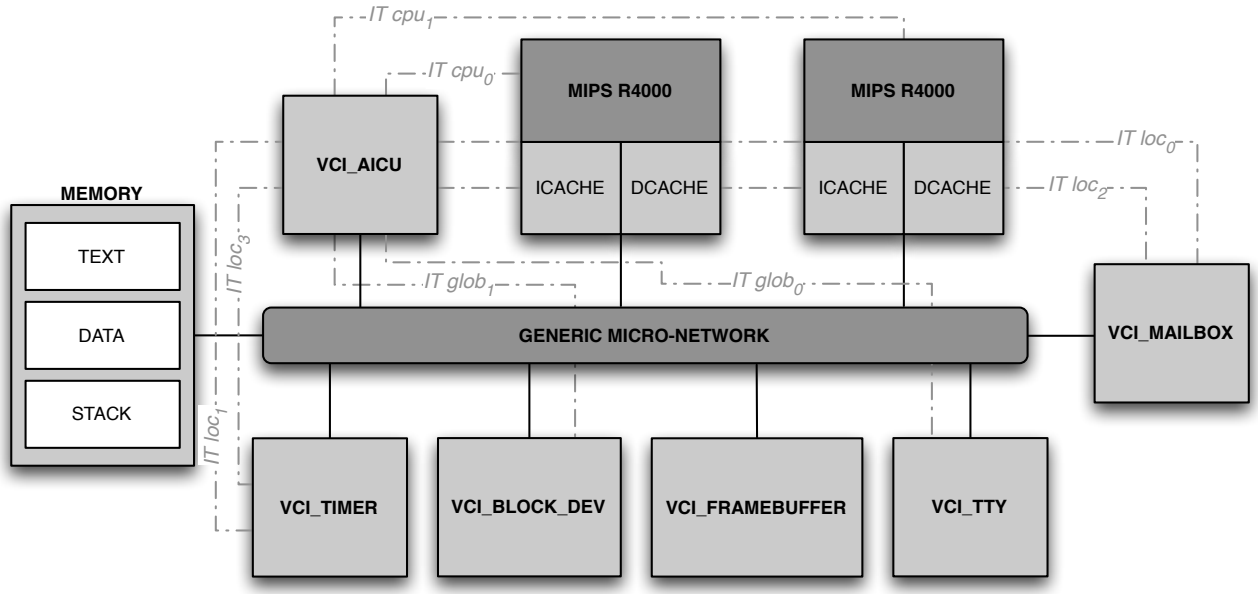


Figure 4.1: The multimedia platform

4.2 Understanding synchronization in SMP architecture

As a starter, you will study one major constraint of programming a low-level application on a multiprocessor architecture: synchronization. Starting with a 2-processor platform (using `SystemPlatform` is enough) and a copy of `SimpleApplication`, you will first analyze the booting process of the application.

```
if processor.id == 0 then puts "hello, "
else if processor.id == 1 then puts "world!"
endif

forever
```

Then, you will write the algorithm above and, using the methods of the `Processor` component, try to synchronize its execution in order to correctly display `"Hello, world!"`.

4.3 Performance analysis

As you probably already noticed during your previous work and your careful reading of the SoCLib documentation, the GMN, the caches, and the MJPEG application can be tuned. The GMN accepts parameters for its FIFO size and its latency, the caches accept parameters for their capacity in number of lines and their line size in blocks, and the application can accept different number of IDCT tasks. By modifying those parameters at will, gather as much information as you can on the behavior of the MJPEG application, for each of the configuration your developed in the previous section.

Chapter 5

Activity 3 — Operating System

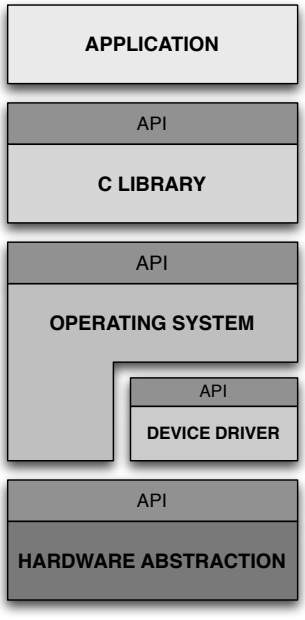
This activity is focused on the device driver intrinsics of the DNA operating system. You will have to get familiar with the structures and methods involved in the devices abstraction process, then you will write a driver for the framebuffer hardware device and you will modify the parallel MJPEG application in such a way that you will use your driver instead of a basic `memcpy` to write to the framebuffer.

5.1 Files

You shall base your work on a pre-existant device driver. Examples of such drivers can be found here: `/softslin/TPSoC3A/Apes/Components/`. You will have a total of eight hours to write this driver. Most of this time will be necessary to understand DNA's driver mechanism, while only a couple of hours will be necessary to implement and test your driver.

5.2 The device driver mechanism

This section describes the inner mechanism of the driver management in the DNA operating system. In this project, the application is structured as depicted in the figure presented on the right: at the top level, the application using the C library API; the C library, using the DNA's API to access system functionalities; the operating system, using a hardware abstraction layer to access processor- and platform-specific operations.
Inside the operating system, modular components can be present and are called *modules*. Three kinds of modules exist: device drivers, filesystem, and extensions. These modules are managed by the support component of the kernel, while the filesystems are used by the VFS component and the drivers are used by the device filesystem - *DevFS*. The subsections below present the management and the control programming interface of device drivers.



5.2.1 Management interface

At boot time, the operating system loads each module and calls its initialization function. To do so, each module needs to export a `driver_t` variable. This variable contains several elements such as the name of the module, and a set of management functions like `init_hardware` and `init_driver`, used by the *DevFS* to create the device file.

5.2.2 Control interface

The control interface exports functions to control and interact with the hardware device. The required functions are presented below:

- `open`: called when the application calls `fopen` on the device path. This function is responsible for the operation at open time.

- `close`: called when the application calls `fclose` on the FILE* item. This function closes the opened file.
- `read`: called when the application calls `fread` on the device FILE. This function reads data from the hardware device.
- `write`: is called when the application calls `fwrite` on the device FILE. This function writes data to the hardware device.
- `control`: usually called when the application calls `ioctl` on the device ID.

Each of them must be correctly implemented. They define as a whole the driver behavior, and are specific to the device.

5.3 Writing and integrating your own driver

You need to copy the driver skeleton in your application's source directory. It will automatically be compiled with the application and included in your binary. However, you still need to add the reference to your `driver_t` in your *ldscript*. When implementing your driver's functions, all the accesses to the hardware need to be done using an HAL primitive described in appendix D.

Chapter 6

Activity 4 — Optimization

This activity focuses on the optimization of existing communication drivers using Direct Memory Access (DMA) devices. DMA devices are automatic memory transfer devices which role is to alleviate the processor during memory transfer tasks.

NOTA BENE This activity requires more autonomy than the previous one. This means that only the coarse-grained goal will be explained. It will be your responsibility to browse SoCLib's documentation to select the right hardware component, and to rely on your past DNA experience to perform the fine-grained tasks.

6.1 Preliminary actions

- Study in details the implementation of the parallel MJPEG.
- Gather data about the communications.
- Study how DMA devices can improve the communications.
- Decide of a preliminary modifications plan.

6.2 Modification of the platform

- Look-up the SoCLib documentation for a DMA device.
- Add one or several DMA devices to the multimedia platform.
- Validate your modifications.

6.3 Modification of the communication drivers

Before writing any piece of code, you need to precisely study the current implementation of the communication drivers and decide which part of the driver can be actually improved. Once you modified the communication drivers, run some benchmarks and quantify the improvements.

Appendix A

Simulation tool-tips

Once and for all, you need to read the documentation associated with the tools you will be using. This appendix provides a few tool tips to get you started with the simulation environment.

A.1 Platform extension caveats

In order to painlessly add an hardware component, you can find below the most important points of the device integration process that need to be double-checked:

1. Objects declaration, signals declaration.
2. Objects instantiation and their parameters.
3. Signals connection.
4. Entry of the component in the segmentation table.
5. Number of initiators and targets in the micro-network.

Most of the issues you will encounter when adding an hardware component are related to one of these five elements, so pay attention to each of them. If your device is an initiator, you may also need to register it into the cache system as being a non-coherent initiator. This operation is done as follows:

```
cctab -> register_non_coherent_initiator (maptab .  indexForId (DEV_TARGET_ID))
```

A.2 Simulation execution

The provided platforms understand two environment variables: `SIMULATION_N_CPUS` contains the desired number of processors in the platform and SIMULATION_N_CYCLES contains the desired number of simulation cycles. Hence, a 10000 cycles simulation with 3 processors can be started this way:

```
$ SIMULATION_N_CPUS=3 SIMULATION_N_CYCLES=10000 ./simulation.x
```

Additionally, the platform contains a default logging device that can be used to easily output debug information. This device is located at address `0xC0000000` and provides a single 32-bit register that merely acts as a data sink. You can either access it directly or use the `platfom_debug_puts` method provided by the HAL.

A.3 GDB, The GNU debugger

The sole and only tool available to debug your cross-compiled software is GDB. By default, your simulator creates an interface with GDB, called GDB server, that can be used to access the simulated processors. To connect GDB to your simulator, launch your simulator and type:

```
$ mips-sls-dnaos-gdb MY_APP -command PLATFORM_PATH/commands.gdb
```

A.3.1 GDB server configuration

The GDB server proposed in SoCLib can be configured in many ways: CPUs can be frozen, watchpoint can be set, etc. Please refer to SoCLib GDB's documentation: `http://www.soclib.fr/trac/dev/wiki/Tools/GdbServer`.

A.3.2 GDB commands digest

`man gdb` can give you a complete overview of GDB's functionalities. However, here is a little cheat sheet:

- Launch the execution: `$ c{ontinue}`
- Set a breakpoint: `$ b{reak} <SYMBOL_NAME> | <LINE_NUMBER>`
- Delete a breakpoint: `$ db <BP_NUMBER>`
- Breakpoint information: `$ info break`
- Display the current source: `$ list`
- Go to next line of code: `$ n{ext}`
- Go to next line of code, step into functions: `$ s{tep}`
- Print a variable/a register: `$ p{rint} <SYMBOL_NAME> | <REGISTER_NAME>`
- Display a variable/a register: `$ p{rint} <SYMBOL_NAME> | <REGISTER_NAME>`
- Register information: `$ info register`
- Threads/CPU information: `$ info thread`
- Threads/CPU switch `$ thread <THREAD_NUMBER>`

Appendix B

The Motion-JPEG decoder application

The Motion-JPEG video format is composed of a succession of JPEG still pictures. It is used by several digital cameras and camcorders to store video clips of a relatively small size. With Motion-JPEG, each frame of video is captured separately and compressed using the JPEG algorithm. JPEG stands for Joint Photographic Experts Group, a standardization committee that gave its name to the still picture compression algorithm it defined. It is a lossy compression algorithm, meaning that the decompressed image is not totally identical to the original image. Its goal is to reduce the size of natural color images as much as possible without affecting the quality of the image as experienced by the human eyes.

B.1 Algorithm overview

The JPEG compression algorithm splits an image in blocks of 8×8 pixels, then translates each block into the frequency domain, eliminates the high using a per image filter, and compresses the resulting blocks using an Huffman encoding with a per image dictionary. The blocks are also called macroblock or MCU for MacroBlock Unit. The resulting bitstream is made of sequences of raw data separated by markers that identify data. The format of the sections is given in section C.

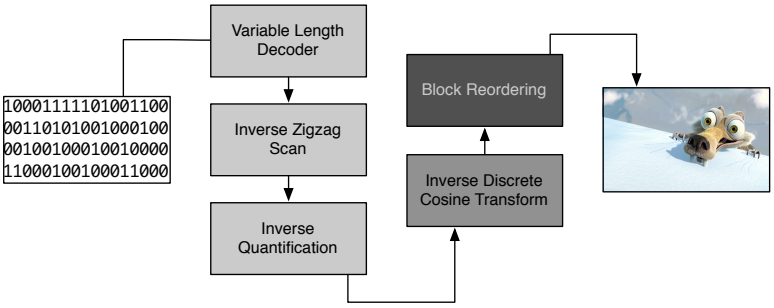


Figure B.1: The JPEG decoder

Figure B.1 presents the sequence of operations required to process a compressed JPEG picture. The decompression of a JPEG picture is done through three sequential steps: data decoding, data decompression, and data reassembling.

B.1.1 Data decoding

This step performs a rough analysis of the input stream in order to extract the JPEG markers and identify the corresponding data. For each image, it executes the following operations:

- **Variable Length Decoding**: using the Huffman tables from the DHT section, and the image size from the SOF section
- **Inverse Quantization and Inverse Zigzag Scan**: using the inverse quantization table from the DQT Section), some of the SOF Section, and some of the SOS section

Variable length decoding

The variable length decoding operation is based on Huffman codes. Huffman codes are called *prefixed* code, because no symbol can be the prefix of another symbol. Figure B.2 illustrates this.

a	11
b	0
c	101
d	1001
e	1000

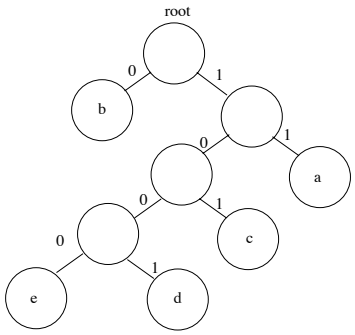


Figure B.2: Decoding the bitstream `0001101011000` leads to `bbbabce`

Inverse Quantization and Inverse Zigzag Scan

The inverse quantization operation multiplies each value in a macroblock by a constant which value depends on the position of the value in the block Each image of the flow has its own constant table in order to optimize the compression. The inverse zigzag scan reorders the data of a macroblock as depicted figure B.3.

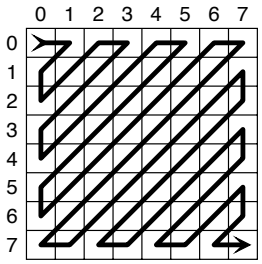


Figure B.3: The inverse zigzag scan operation

B.1.2 Data decompression

This steps performs the Inverse Discrete Cosine Transform operation (IDCT). It is implemented as a doubly nested sum, thanks to Loëffler's algorithm, that uses a minimal number of multiplications to accomplish this computation.

B.1.3 Data reassembling

This step reorganizes the macroblocks coming from the data decompression step into a picture. Once the picture is finalized, it is sent to the framebuffer.

B.2 Parallel decoder

The parallel implementation of the MJPEG decoder, proposed in this case study, separates the input stream into multiple frames following the SOF and EOF tags, feeds these frames to several sequential JPEG decoders, and reassembles and displays the decoded frames. The principal advantage of this implementation is its scalability. Indeed, a linear performance gain is obtained when the number of processor present in the target hardware platform is increased.

Appendix C

Information about the JPEG format

A JPEG marker is a `short int` (16 bits or 2 bytes) whose first byte is `0xFF` and the second one is different from `0xFF`. The table below gives the markers specified in the normative document DIS 10918.1. The markers are aligned on a byte boundary.

C.1 Stream scope markers

Code	ID	Description
0x01	TEM	
0x02 ... 0xbf	Reserved (not used)	
0xc0	SOF0	Baseline DCT (Huffman)
0xc1	SOF1	Extended sequential DCT (Huffman)
0xc2	SOF2	Progressive DCT (Huffman)
0xc3	SOF3	Spatial (Lossless) DCT (Huffman)
0xc4	DHT	Define Huffman Tables
0xc5	SOF5	Differential sequential DCT (Huffman)
0xc6	SOF6	Differential progressive DCT (Huffman)
0xc7	SOF7	Differential spatial DCT (Huffman)
0xc8	JPG	Reserved for JPEG extensions
0xc9	SOF9	Extended sequential DCT (arithmetic)
0xca	SOF10	Progressive DCT (arithmetic)
0xcb	SOF11	Spatial (lossless) DCT (arithmetic)
0xcc	DAC	Arithmetic Conditionning info
0xcd	SOF13	Differential sequential DCT (arithmetic)
0xce	SOF14	Differential progressive DCT (arithmetic)
0xcf	SOF15	Differential lossless (arithmetic)
0xd0 ... 0xd7	RST0 ... RST7	Restart Interval Termination
0xd8	SOI	Start Of Image (beginning of datastream)
0xd9	EOI	End Of Image (end of datastream)
0xda	SOS	Start Of Scan (begins compressed data)
0xdb	DQT	Define Quantization tables
0xdc	DNL	
0xdd	DRI	Define Restart Interval
0xde	DHP	
0xdf	EXP	
0xe0 ... 0xef	APP0 ... APP15	Application-specific marker
0xf0 ... 0xfd	JPG0 ... JPG13	
0xfe	COM	Comment

C.2 SOFx - Start Of Frame section format

0x00	2	SOFx Marker identifier (FFCx)
0x02	2	Length in byte of this section, including this data.
0x04	1	Data bit precision (usually 8, for baseline JPEG)
0x05	2	Image height in pixels
0x07	2	Image width in pixels
0x09	1	N = Number of components (eg. 3 for RGB, 1 for Grayscale)
0x10	3N	N times : - 1 byte : Component index - 1 byte : H (4bits high) and V(4bits low) sampling factors - 1 byte : Quantization table associated

C.3 DHT - Define Huffman Tables section format

0x00	2	DHT Marker identifier (FFC4)
0x02	2	Length in byte of this section, including this data.
0x04	1	Huffman Table information : bit 0..3 : index (0..3, otherwise error) bit 4 : type (0=DC, 1=AC) bit 5..7 : not used, must be 0
0x05	16	Number of symbols with codes of length 1..16. The sum of these bytes is the total number of codes and must be <= 256
0x15	x	Table containing the symbols in order of increasing code length.

C.4 DAC - Define Arithmetic Components section format

0x00	2	DAC Marker identifier (FFCC)
0x02	2	Length in byte of this section, including this data.
0x04	2N	N times : 1 byte : Index - bit 0..3 : index (0..3, otherwise error) - bit 4 : type (0=DC, 1=AC) - bit 5..7 : not used, must be 0 1 byte : Value

C.5 SOS - Start Of Scan section format

0x00	2	SOS Marker identifier (FFDA)
0x02	2	Length in byte of this section, including this data.
0x04	1	N = Number of components 2N+6 must be equal to Length
0x05	2N	N times : 1 byte : Component index 1 byte : (4bits) DC table index (4bits) AC table index
...	1	Ss : Start of spectral or predictor selection
...	1	Se : End of spectral selection
...	1	Ah : 4bit, Successive approximation bit position high Al : 4bit, Successive approximation bit position low or poin transform

C.6 DQT - Define Quantization Table section format

0x00	2	DQT Marker identifier (0xFFDB)
0x02	2	Length in byte of this section, including this data.
0x04	1	4bits high : precision (0=8bits, 1=16bits) 4bits low : quantization table index
0x05	64	Quantization values stored in zig-zag format

C.7 DRI - Define Restart Interval section format

0x00	2	DRI Marker identifier (0xFFDD)
0x02	2	Length in byte of this section, including this data. Must be 4.
0x04	2	Restart interval

C.8 APPx - Application-specific marker section format

0x00	2	APPx Marker identifier (0xFFEx)
0x02	2	Length in byte of this section, including this data
. 0x04	x	Data

C.9 COM - Comment section format

0x00	2	COM Marker identifier (0xFFFE)
0x02	2	Length in byte of this section, including this data.
0x04	x	Data

Appendix D

Hardware Abstractions Layer

D.1 Platform abstraction interface

D.1.1 *Endianness* management

Definitions

definition `PLATFORM_IS_LITTLE_ENDIAN`

Summary This definition specifies that all shared data is of type *little endian*.

definition `PLATFORM_IS_BIG_ENDIAN`

Summary This definition specifies that all shared data is of type *big endian*.

D.1.2 Multiprocessor management

Method

function `platform_mp_cpu_count` **return** `Integert32`
`type :` **in** `enumerate - The processor's type`

Summary This function returns the number of processors for a specified *type*.

D.1.3 Standard debug output

Method

procedure `platform_debug_puts`
`string :` **in** `String - The string to print on the debug output`

Summary This procedure prints *string* on the platform debug device.

D.2 CPU abstraction interface

D.2.1 *Endianness* management

Methods

procedure `cpu_endian_is_big_{16,32,64,...}`
`value :` **inout** `Integer{16,32,64,...}`

Summary Depending on the processor's *endianness*, this procedure reorders the bytes of the {16,32,64,...} bits word *value* in a *big-endian* fashion.

```
procedure cpu_endian_is_little_{16,32,64,...}
    value : inout Integer{16,32,64,...}
```

Summary Depending on the processor's *endianness*, this procedure reorders the bytes of the {16,32,64,...} bits word *value* in a *little-endian* fashion.

```
procedure cpu_endian_concat
    size   : in Integer32
    result : out Integer{16,32,64,...}
    low    : in Integer{8,16,32,...}
    high   : in Integer{8,16,32,...}
```

Summary This procedure concatenates two {8,16,32,...} bits elements into one {16,32,64,...} bits elements that matches the processor's endianness.

```
procedure cpu_endian_split
    size  : in Integer32
    value : in Integer{16,32,64,...}
    low   : out Integer{8,16,32,...}
    high  : out Integer{8,16,32,...}
```

Summary This procedure splits one {16,32,64,...} bits element matching the processor's endianness into two {8,16,32,...} bits elements.

D.2.2 Execution context management

Definition

```
definition CPU_CONTEXT_SIZE
```

Summary This definition defines the context's size of the processor, in bytes.

Type

```
type cpu_context_t
```

Summary This type represents the execution context of the processor. It ususally contains a copy of the processor's registers involved in the execution of an execution thread, usually all the general-purpose registers, some special registers, and the status register.

Methods

```
procedure cpu_context_init
    context   : out access cpu_context_t
    stack     : in access array of Integer8
    size      : in Integer32 - The stack's size
    entry     : in access procedure
    arguments : in access record
```

Summary This procedure initializes an execution context with a *stack* of a specific *size*, an *entry* point, and some *arguments* that will be passed to the entry point. Note that the stack must have been allocated. The remaining fields of the context are usually set to 0.

```
procedure cpu_context_load
    context : in access cpu_context_t
```

Summary This procedure loads a specific *context*.

```
procedure cpu_context_save
    context :  out access cpu_context_t
    entry   :  in access
```

Summary This procedure saves the current execution context into *context*. The *entry* argument contains the address from which the execution thread needs to be restarted.

```
procedure cpu_context_switch
    from  :  out access cpu_context_t
    to    :  in access cpu_context_t
```

Summary This procedure saves the current execution context into *from* and loads the context *to*. It is merely a combination of the functions `cpu_context_save` and `cpu_context_load`. However, it is not suited to perform finely-grained switch operations.

D.2.3 I/O management

Methods

```
procedure cpu_read
    size    :  in Integer32
    address :  in access Integer{8,16,...}
    result  :  out Integer{8,16,...}
```

Summary This procedure reads a {8,16,...}-bit integer from a system-wide *address* to the variable *result*.

```
procedure cpu_read_uncached
    size    :  in Integer32
    address :  in access Integer{8,16,...}
    result  :  out Integer{8,16,...}
```

Summary This procedure reads a {8,16,...}-bit integer from a system-wide *address* to *result*. The data cache, if any, is bypassed.

```
procedure cpu_read_vector
    mode  :  in VectorMode
    from  :  in access <VectorMode>
    to    :  out access <VectorMode>
    size  :  in Integer32
```

Summary This procedure reads a vector of size *size* from the system-wide location *from* and writes it to the local region *to*. This operation is executed in a specific *mode*, which can be either Integer{8,16,...}, Float, or Double.

```
procedure cpu_write
    size    :  in Integer32
    address :  in access Integer{8,16,...}
    value   :  out Integer{8,16,...}
```

Summary This procedure writes a {8,16,...}-bit integer *value* to a system-wide *address*.

```
procedure cpu_write_uncached
     size    :  in Integer32
     address :  in access Integer{8,16,...}
     value   :  out Integer{8,16,...}
```

Summary This procedure writes a {8,16,...} integer *value* from a system-wide *address*. The data cache, if any, is bypassed.

```
procedure cpu_write_vector
     mode  :  in VectorMode
     to    :  in access <VectorMode>
     from  :  out access <VectorMode>
     size  :  in Integer32
```

Summary This procedure reads a vector of size *size* from the local region location *from* and writes it to the system-wide location *to*. This operation is executed in a specific *mode*, which can be either Integer{8,16,...}, Float, or Double.

D.2.4 Interrupts management

Definition

```
definition CPU_N_IT
```

Summary This definition contains the number of hardware interrupts of the processor. This is a loose definition, since it generally defines the number of available hardware interrupts from the abstraction. This flexibility is particularly helpful when the processor's interrupts are controlled by an external interrupt controller. In that case, the abstraction can be extended to the interrupt controller.

Types

```
type interrupt_id_t
```

Summary This type represents an interrupt line of a processor or of a processor's subsystem.

```
type interrupt_status_t
```

Summary This type represents the interrupt status of a processor or of a processor's subsystem.

```
type interrupt_handler_t
```

Summary This type represents the handler of a processor interrupt.

Methods

```
procedure cpu_trap_enable
     vector :  in interrupt_id_t
```

Summary This procedure unmasks the interruption indexed by *vector*.

```
procedure cpu_trap_disable
     vector :  in interrupt_id_t
```

Summary This procedure masks the interruption indexed by *vector*.

```
procedure cpu_trap_attach_isr
     vector  :  in interrupt_id_t
     handler :  in interrupt_handler_t
```

Summary This procedure attaches an interrupt *handler* to a specific interrupt *vector*.

D.2.5 Exception management

Types

```
type exception_id_t
```

Summary This type represents an exception of a processor.

```
type exception_handler_t
```

Summary This type represents the handler of a processor exception.

Method

```
procedure cpu_trap_attach_esr
     vector  :  in exception_id_t
     handler :  in exception_handler_t
```

Summary This procedure attaches an exception *handler* to a specific exception *vector*.

D.2.6 Multiprocessor management

Methods

```
function cpu_mp_count return Integer32
```

Summary This function returns the number of CPUs identical to the one calling the function.

```
function cpu_mp_id return Integer32
```

Summary This function returns the unique processor's identifier.

```
procedure cpu_mp_wait
```

Summary When this procedure is called, the processor spins until an other processor calls `cpu_mp_proceed`.

```
procedure cpu_mp_proceed
```

Summary This procedure releases processors waiting on the `cpu_mp_wait` call.

```
procedure cpu_mp_send_ipi
     cpuid   :  in Integer32
     message :  in Integer32
     data    :  in access record
```

Summary This procedure sends an IPI *message* to the processor with id *cpuid* with its associated *data*.

D.2.7 Timer management

Types

```
type bigtime_t
```

Summary This type represents time in nanoseconds, coded on 64 bits.

Methods

```
procedure cpu_timer_set
     cpuid    :  in Integer32
     deadline :  in bigtime_t
```

Summary This procedure sets the next dealine of *cpuid*'s timer to *deadline* nanoseconds.

```
procedure cpu_timer_get
     cpuid    :  in Integer32
     deadline :  out bigtime_t
```

Summary This procedure gets the current value of *cpuid*'s timer into *deadline*.

```
procedure cpu_timer_cancel
     cpuid :  in Integer32
```

Summary This procedure cancels the pending deadline on timer *cpuid*.

D.2.8 Synchronization management

Methods

```
function cpu_test_and_set return Integer32
     lock :  out Integer32
```

Summary This function performs a *test-and-set* operation on the given *lock*.

```
function cpu_compare_and_swap return Integer32
     lock :  inout Integer32
     old  :  in Integer32
     new  :  in Integer32
```

Summary This function performs a compare-and-swap on the specified *lock*, using the two *old* and *new* values as reference.

D.2.9 Power management

Method

```
procedure cpu_power_wake_on_interrupt
```

Summary This procedure puts the processor in a *wake-on-interrupt* mode.

D.2.10 Cache management

Method

```
procedure cpu_cache_sync
```

Summary This procedure flushes the write buffer of the processor's caches.

```
procedure cpu_cache_invalidate
     cache_type :  in cache_type_t
     address    :  in access
     words      :  in Integer32
```

Summary Depending on *cache_type*, this procedure invalidates the lines containing the addresses from *base* to *base* + *words* from either the instruction cache or the data cache.