PUBLIC

TULIPP

H2020-ICT-O4-2015
Grant Agreement n° 688403

D3.3: Generic RTOS APIs and libraries documentation

Lead Author: Antonio Paolillo, HIPPEROS S.A.
with contributions from:

Organisation no.	Organisation name	Participant Name
4	HIPPEROS	Olivier Desenfans
4	HIPPEROS	Vladimir Svoboda
4	HIPPEROS	Paul Rodriguez
2	TUD	Julian Haase
2	TUD	Habib ul Hasan Khan
2	TUD	Diana Göhringer

Reviewers

Organisation no.	Organisation name	Participant Name
2	TUD	Julian Haase
3	SUN	Emilie Wheatley

PUBLIC

1 Document Description

Deliverable number	3.3
Deliverable title	Generic RTOS APIs and libraries documentation
Work Package	3
Deliverable nature	Report
Dissemination level	Public
Contractual delivery date	
Actual delivery	
Version	1.0

	Written by	**Approved by**
Name Signature	TULIPP consortium	

2 Version history

Version	Date	Editors	Description
0.1	2018-08-08	Antonio Paolillo	Initial draft
0.2	2018-08-21	Julian Haase	TUD's contributions and adding suggestions after review
1.0	2018-10-12	Antonio Paolillo	Document finalisation

3 Abstract

This deliverable represents the work carried out in the context of Task T3.3. This task covers the following content. HIPPEROS implemented the standard APIs and libraries needed to facilitate efficient and rapid implementation of applications on top of the RTOS. We defined how the tools can map the application logic to the RTOS configuration and policies and implemented the required services for lightweight tracing. TUD implemented the device drivers needed to make the FPGA IP cores developed in T4.3 accessible through the RTOS. Collectively these functionalities form an energy efficient, high performance processing platform that is able to support the Generic Application Design Guidelines for image processing.

4 Table of Content

5 Introduction

Work package 3 focuses on implementing an operating system that matches the TULIPP guidelines and concepts from WP1, considers the choices done in WP2 and WP4 and deliver an integrated real-time operating system to WP5.

To ensure this, we have to work on the following actions:

- The design and development of a parallel real-time operating system that can handle the target platform specifics and constraints and is able to optimize low-power processing (tasks T3.1 and T3.2).
- The creation of the required standard APIs and runtime libraries for the RTOS (task T3.3).
- The extensions of the operating system with support for the hardware and software requirements of image processing (tasks T3.2 and T3.3).
- The instantiation of the reference platform.

Deliverable D3.3 represents the work carried out in task T3.3 spanning from M07 to M33 of the TULIPP project.

The objective of this task is to implement the standard APIs and libraries needed to facilitate efficient and rapid implementation of applications on top of the RTOS with support from NTNU. Furthermore, we defined how the tools can map the application logic to the RTOS configuration and policies and implement the required services. TUD implemented the device drivers needed to make the FPGA IP cores developed in T4.3 accessible through the RTOS.

6 HIPPEROS RTOS Application Programming Interface

6.1 Introduction

This chapter is intended as an introduction to the HIPPEROS RTOS API for user applications. It provides an overview of the different features provided to design, boot and run a real-time application. Links are made to the HIPPEROS Technical Reference Manual (TRM) for technical information about the different topics.

HIPPEROS is highly configurable; most features can be (de)activated using build options.

6.2 Application Design

The HIPPEROS RTOS defines three levels of abstraction for user applications: tasks, processes and threads.

The concept of task allows to define properties of a user program. These properties are adopted at runtime to define the behaviour of the process: scheduling parameters (ex: period, deadline), resource allocation (ex: stack size), …

Tasks are not defined in the source code itself. HIPPEROS uses a XML file called the application task set. This file is parsed to generate binary data that will be read by the operating system at boot time. See "Chapter 2: Defining the task set of the application" in the TRM for more information.

A process is a running instance of a task. It encompasses a full program. It disposes of its own resources (specified by the associated task) and may have its own address space (depending on the configuration).

Finally, threads are parallel execution contexts existing within a process. A process can start several threads.

6.3 Loading applications

HIPPEROS supports two schemes for loading applications on the target: linking the application with the kernel or using independent executables for each application.

6.3.1 Static linking

This method is best suited for targets requiring a small memory footprint or that do not have a file system (typical for small systems booting from an on-chip ROM). Using this method, the kernel and applications are linked into one single executable file. The size of the binary is significantly reduced, but the price to pay is that it is impossible to use separate address spaces for user applications using this scheme (see below).

6.3.2 Independent binaries

Using independent binaries means that each application is compiled as its own ELF executable file. Using this scheme enables full application memory protection (see below).

6.4 Scheduling

Deliverable D3.2 describes the supported scheduling policies and algorithms. To summarize, HIPPEROS supports Rate Monotonic and Earliest Deadline First pre-emptive scheduling for single- and multi-core targets. Scheduling can be partitioned (each thread is allocated to a specific core) or global (the kernel can move threads between cores). We also describe in the same document the use of mixed-criticality scheduling techniques. This allows to define different criticality levels for tasks inside a single system.

6.5 Memory Protection

HIPPEROS supports different levels of memory protection:

- Isolated address spaces.
- Single address space.
- No protection.

6.5.1 Full memory isolation

In this configuration, each process has its own address space. It cannot interact with the address space of other processes unless it explicitly uses shared memory regions. The deliverable D3.2 provides in-detail explanations on how this configuration works. Using this configuration, each process has its own page table and virtual memory mapping.

6.5.2 Single address space

The single address space configuration enables minimal memory protection using a single page table for the whole system. A single virtual memory mapping is created for the whole system. The kernel is protected from access by the applications and applications cannot access memory regions that are not explicitly mapped (and therefore cannot modify hardware registers by mistake). However, a process can easily access data and code belonging to another process.

6.5.3 No MMU

For information, HIPPEROS also supports configurations with the MMU disabled. This is mostly useful for targets that do not possess a MMU altogether. Processes are still running in user mode but can access and modify the whole memory space. For targets that do have a MMU we strongly recommend using some form of paging.

6.6 API overview

6.6.1 Process/thread life cycle

Processes are automatically started by the operating system after the boot. An important point about the HIPPEROS process life cycle is that the time constraints on the process are entirely specified in the task set. One must explicitly state:

- The recurrence: whether the process is periodic, aperiodic (activation triggered by another process) or unique.
- Timing constraints: the period (for periodic processes), the offset (allows delaying the first activation), the deadline and the WCET (Worst-Case Execution Time). If a deadline or WCET is specified, the kernel monitors the process and acts if an overrun is detected. These monitors are called time guards.

When a process is started, the kernel starts a thread at the entry point for the process: the master thread.

For unique processes, the application model is similar to the standard C application model: the master thread starts at the main function and runs until it exits (by calling exit or simply returning from the main function). Here is an example:

```
int unique_main(int argc, char* argv[])
{
    application();

    /* Complete clean-up at the end (exit is called implicitly) */
    return EXIT_SUCCESS;
}
```

Things are a bit more complex for periodic processes. Usually, these start with an initialisation phase that must only be executed once. Then the process starts performing its periodic operations, usually inside an infinite loop. A periodic process usually never exits. For this use case HIPPEROS provides the suspend system call. Suspend will tell the operating system that the current iteration of the process is over. This tells the OS to reset the deadline and WCET time guards.

Here is an example of a periodic process making use of the suspend system call:

```
int periodic_main(int argc, char* argv[])
{
    while(1) {
        application();
        h_suspend();
    }

    /* Unreachable */
    return EXIT_SUCCESS;
}
```

6.6.2 Inter-Process Communication (IPC)

HIPPEROS provides two mechanisms for inter-process communication: copy-buffer IPC (channels) and synchronous IPC (shared memory).

Copy-buffer IPC (see chapter 4.8 of the TRM) allows to exchange small messages through the kernel. The messages are copied to avoid alteration.

Synchronous IPC is used for exchanging large messages without the need for copies. Synchronous IPC is equivalent to a shared memory region guarded by process-level mutexes. See chapter 4.9 of the TRM for more details.

6.6.3 Spawning new threads

Once started, the master thread can spawn new threads at will. Threads start with their own stack and share the same address space. HIPPEROS provides an implementation of the pthread API for this purpose. See chapter 4.6: Thread management in the TRM for more details.

6.6.4 Critical sections

HIPPEROS provides semaphores and mutexes for inter-thread signaling. Both mechanisms use the POSIX APIs. See chapter 4.10 of the TRM for a description of the semaphore API. HIPPEROS uses the pthread API for mutexes.

7 Supported Libraries and Drivers

In this chapter, we present the libraries and drivers supported by the operating system and available in the HIPPEROS-TULIPP shipment.

7.1 HIPPEROS API

As presented in the last chapter, the HIPPEROS native API is a library that must be statically linked with applications. This library is the interface to the low-level feature implemented in the OS kernel. It allows, among others, to issue system calls, access thread-local storage and share resources with other tasks or applications such as mutexes, semaphores, IPC, etc.

7.2 C Standard Library

For tasks written in the C language, HIPPEROS provides support for the C Standard Library. We implemented a fork of the newlib implementation of the C standard library and added support for the HIPPEROS RTOS. Support for standard math operation (sqrt, sin, cos, etc.) is also provided (the "lib math").

The code is freely accessible on github: https://github.com/hipperos/newlib

The HIPPEROS support in newlib is implemented by using the HIPPEROS native API, which is the official library (see above) supported by HIPPEROS to access the run-time information about the system and the running user tasks.

7.3 C++

HIPPEROS provides support for tasks written in the C++ language. The C++ library is implemented and linked with the toolchains and therefore all standard features of C++ are allowed (including dynamic memory allocation, standard template library, etc.)

7.4 File system

All targets supported by HIPPEROS provide a SD/MMC port. We implemented support for the FAT16/32 file systems, by using the FatFs library (http://elm-chan.org/fsw/ff/00index_e.html). Having a micro-kernel architecture, we implemented the HIPPEROS file system as a service running in user mode. The file system is linked to a SD/MMC driver of the HIPPEROS driver library (see below).

7.5 Network

HIPPEROS supports TCP/IP and UDP/IP stacks. We use the LwIP library, which is a lightweight network stack targeting embedded systems (https://savannah.nongnu.org/projects/lwip/).

Again, the network start runs a user-space service and is linked to the Ethernet driver of the target platform.

7.6 Drivers

HIPPEROS provides a library containing all supported device drivers. To use it, services or user tasks must statically link this library.

This library provides support for the following devices:

- Clocks and platform timers;
- Ethernet network device;
- FPGA and PCAP (in the case of Zynq platforms);
- GPIO;
- I2C;
- SD/MMC;
- Serial port;
- SPI.

7.7 OpenMP

OpenMP is standard framework to ease the development of parallel multi-threaded applications, i.e. running simultaneously on several cores of the target platform. It consists in compiler pragmas injected into user code and a run-time library implementing the interface between the compiler OpenMP pragma support and the underlying operating system.

As required by some of the TULIPP use cases, HIPPEROS implemented basic support for the OpenMP library. HIPPEROS implemented HOMPRTL, the HIPPEROS port for the OpenMP Run-Time Library. This library defines the functions that are used by OpenMP code when the pragmas are translated by the compiler with the OpenMP flags enabled. HIPPEROS currently only support the “#pragma omp parallel”. Future work includes a wider port of OpenMP to run-time.

7.8 OpenCV

OpenCV is a library that provides building blocks for image processing and computer vision applications. Recently, Xilinx tools allow to turn OpenCV calls into hardware accelerators (in the FPGA Programmable Logic), making this library a target of choice for high performance low-power embedded image processing applications.

The HIPPEROS support of a subset of OpenCV, targeting embedded applications, is a work-in-progress.

8 User Space Drivers Implementation Details

In a micro-kernel based operating system, the device drivers provided for user applications all run in user space. Compared to the monolithic approach, this reduces the complexity and size of the kernel and prevents system-wide crashes due to faulty drivers. It is nonetheless expected to suffer from a performance penalty as the drivers must rely on system calls to the kernel to perform a number of operations that are only a function call away in a monolithic kernel.

This chapter identifies all the issues faced when writing device drivers in a microkernel-based OS and proposes solutions as well as system call APIs to solve them. It is advised to read the deliverable D3.2 and previous material in this deliverable on OS architecture and paging before reading the following.

8.1 Virtual memory mapping

Most devices and controllers are exposed to the user using memory mappings. A range of addresses is reserved in the addressable physical memory range of the processor for each device, one word of memory representing a register of the device.

When the operating system uses paging and memory virtualization, it is impossible for a user application to access devices using their address in physical memory (it would, at best, cause a memory exception). It is necessary to map the device memory map in virtual memory space before trying to access it.

As the kernel is responsible of managing the page tables, drivers must be able to request the mapping of a physical address range in virtual memory. Device memory mappings usually require specific attributes in the page table entry (one of them being that the entry cannot be cached). Our solution to this is to provide the following system call:

```
Word_t h_mm_mapDevice(Address_t pAddr,
                      Size_t length,
                      UWord_t prot,
                      UWord_t cache,
                      Address_t* vAddr)
```

This system call maps a device memory range with the given protection (read-only, read/write) and cache attributes (architecture-specific) and provides the virtual address at which the requesting thread can access the device once the mapping is performed.

8.2 Memory allocation

Memory allocation is usually frowned upon in mission-critical real-time systems. This is because allocators have a highly variable response time depending on whether the kernel must be called to allocate more physical memory to the process. Furthermore, memory allocations can fail when there is no more memory to allocate. If a memory leak occurs the whole application system could break down.

The usual workaround is to use a pseudo-allocator. The total amount of memory required for each thread is computed offline at design time and a simpler, more predictable allocator is used to manage the memory for each thread/process. The kernel is not in charge of allocating physical memory at run-time and therefore there is no need to provide system calls to do so.

However, there is a very specific need regarding DMA (Direct Memory Access) buffers. These are memory areas where agents external to the processor (think disk or network controllers) can read and write data without involving the CPU. Drivers can then start a DMA transfer and give the CPU back for other uses while the data is transferred.

The first issue with DMA buffers is that devices usually require the physical address of the buffer to access them (devices usually do not have access to the virtual memory mapping). While the kernel could provide a system call to request the translation of a virtual address to a physical address, this is expensive and even unsafe (knowledge of physical addresses can be used for security exploits).

The solution we propose is to provide a system call to perform physical memory allocation directly in user space. A block of physical memory is allocated and its physical address is returned. The driver can then map this block in virtual memory to access the buffer by using the `mapDevice` system call described above. This is the system call used to allocate physical memory:

```
Word_t h_mm_palloc(Size_t length, UWord_t attributes, Address_t* pAddr)
```

This system call allocates a range in physical memory and returns the start address. This system call is generic and can be used for allocating physical space in RAM for any purpose. However, when allocating DMA buffers a special attribute must be passed as some platforms reserve an address range specifically for that purpose.

One must note that DMA controllers use the same memory bus as the processor. Using DMA therefore increases traffic on the bus and can cause contention. This coupling can be unacceptable for highly critical systems. Furthermore, drivers that use DMA must be trusted as they allocate physical memory directly and could harm the system if this is done without care.

8.3 Cache primitives

Most processors rely on data caches to accelerate processing by offloading read/write operations to the cache controller. Unless caches are deactivated (a common occurrence in mission-critical systems as caches tend to be highly unpredictable beasts), drivers need primitives to access the cache and force a number of operations:

- Cleaning the cache: especially when using DMA, the driver must make sure that the data it wrote to memory was actually written in the RAM of the system before starting the transfer. As the cache is configured as write-back on most systems, the driver must take care to clean the contents of the buffer from the cache and ask the cache controller to write the data to main memory.
- Invalidating the cache: in some use cases the contents of the cache must be purely and simply discarded. This is called cache invalidation.

Cache primitives do not require executing code in privileged mode. This is why we can provide them as a user library alongside the drivers. Therefore, there will be no performance penalty when compared to a monolithic kernel.

8.4 Interrupts

Interrupts are a must-have for device drivers. There are two ways to handle events when writing device drivers. The first is polling. Software waits actively on a software flag raised in one of the registers of the device. The second is interrupts. The device signals that an event occurred to the CPU. The CPU then executes code to manage the event: the interrupt handler. The advantage is obvious: instead of wasting CPU cycles while waiting for the event, interrupts allow the user to use the CPU for other tasks until the expected event occurs. CPUs handle interrupts using a hardware device called the interrupt controller. The kernel is in charge of configuring and using the interrupt controller.

User-space drivers must be able to block and wait on interrupts originating from devices, releasing the CPU for other applications in the meantime. Interrupt handlers run in privileged mode, meaning they are part of the kernel code. Part of the problem is that there is often a small bit of device-specific code involved in handling the interrupt. For example, it is often necessary to clear the interrupt flag in one of the registers of the device. Without doing this, the interrupt will not be cleared in the registers of the device and the interrupt will be triggered continuously in the interrupt controller of the processor. As we wish to keep the kernel as small and generic as possible, this code must be executed from the user space device driver.

Our solution to this is to provide a single system call for device drivers:

```
Word_t h_irq_wait(UWord_t interruptId)
```

Using this call, the driver signals that it is waiting on the given interrupt identifier and can be pre-empted. By default, the kernel disables all device interrupts individually (in the interrupt controller of the processor). When a driver calls `h_irq_wait`, the kernel enables that interrupt. The interrupt handler is designed to disable the device interrupt in the interrupt controller of the processor as soon as it occurs and leaves it to the driver to perform clean-up operations such as clearing interrupt flags in the registers of the device.

This solution is simple in that it only requires a single system call and is generic at the kernel-level. The kernel must only know of the main interrupt controller(s) of the processor (which it has to anyway for things such as the system timer). Device-specific code is isolated in the user space driver.

9 Development Environment

Within the shipment of a HIPPEROS RTOS package, we provide all scripts required to set-up an Ubuntu machine as a HIPPEROS application development environment. We provide the “HIPPEROS SDK”. This includes scripts, cross toolchains and dependencies to be able to build a set of tasks that will be the user application running on top of the HIPPEROS RTOS.

The setup of a development environment for embedded devices development is usually a costly task as one cannot rely on most of the resources of the package manager of the operating system (apt on Ubuntu for example) of the development machine. This development environment consists of several components:

- A cross-compiler and the associated programs
- RTOS binary and libraries
- Debugging and deployment tools

With each tool come dependencies to specific versions of libraries. Moreover, the application being developed may depend on different versions/builds of some of these tools. This can easily lead to difficulties when updating, breakage with the native package manager, and as stated before the setup of the environment may take a long time.

Additionally, the embedded application is usually built on a dedicated build server (example: Jenkins) which also needs to be set up and maintained, meet certain performance requirements, and even more importantly it must be trusted, especially in a certified context.

To mitigate these issues HIPPEROS now provides an SDK solution based on containers. Containers suit the needs of easy installation, low maintenance cost, isolation and performance, as explained below.

9.1 Containers

At HIPPEROS we wanted to develop a solution that suits certified environments but also less constrained ones. We wanted also the solution to be efficient. This brings several constraints:

- Isolation
- Performance
- Variety of installation machines

While virtual machine solutions (like VirtualBox or VMWare) provide isolation and can be installed nearly everywhere, they have a significant performance cost.

Containers are similar to virtual machines, except that there is no emulation. The processes running in the container do not see the external world, but they run as fast as the processes running on the native host.

The HIPPEROS developers and the build machines use this solution.

9.2 Installation process

In the HIPPEROS package, along with the documentation, the RTOS binaries and libraries there is an installation script that must be executed on the host machine. This script automates the installation of the environment, there is not much to do on the development machine as nearly everything is in the container image. It must essentially:

- install Docker (and its dependencies);
- install udev rules and an associated script that ensures that the boards connected to the host machine are also accessible from the container.

The installation only takes a few minutes. Additionally, scripts are provided to execute commands in a container, to open a new terminal in an existing container... These scripts ensure that the right version of the container image is used.

9.3 How containers work

Containers are instantiated using images built upfront. Most of the time, the state of the container is not saved, which makes it a stateless environment. The interaction with the external world occurs through:

- mount points: directories of the host file system also mounted in the container.
- exposed ports: it is possible to run a server in a container and expose it to the host.

Each HIPPEROS build provides its own scripts defining the container image to use, meaning that the application developer/build server can use different HIPPEROS builds simultaneously without risking breaking its installation.

These scripts facilitate the instantiation of containers and terminals in these containers.

9.4 Drawbacks

The main drawback of this solution is its main strength: the isolation. It is not possible to access tools installed on the native host from the container. There are however several ways to circumvent that limitation. Required tools can be installed into the container environment thus preserving the isolation. Alternatively, the tools may be made to communicate with the container through files or network ports, though not many tools are designed for this purpose.

10 Toolchains and External Tools

10.1 Toolchains

As HIPPEROS is an operating system for embedded targets, architecture-specific code is usually compiled on a host computer with a Linux machine targeting the embedded system. For this purpose, cross toolchains are used. A cross toolchain allows to compile program on the host that generates code for the embedded target architecture.

In the context of the TULIPP project, the target architectures are ARMv7a (Zynq-7000 Processing System) and ARMv8 (UltraScale+ APU).

Our current methodology regarding toolchains consists in packaging toolchains specific for the HIPPEROS development environment. In TULIPP, we provide the GNU-based triplet arm-hipperos-eabi. Users are then able to invoke GCC (and associated linkers, binutils, etc.) to compile user tasks and use standard C library calls. Additionally, we support the LLVM Clang compiler.

10.2 External tools

The Xilinx toolchains is of particular interest for the TULIPP platform, especially the SDx/SDSoC software.

SDx/SDSoC allows to compile C/C++ software and to automatically generate hardware accelerators to optimise the user applications. This means that a user task using SDx is composed of binary code to carry-on the software (on the PS side) and a bitstream to load into the FPGA carrying the hardware accelerator (on the PL FPGA side).

One of the objectives (heavily related to WP4) is to integrate the HIPPEROS application build system with the SDx flow. To that aim, it is required:

- To use the sdscc compiler in the HIPPEROS toolchains configuration scripts. sdscc is an upper layer on top of GCC allowing to invoke the SDx flow when compiling a single C file.
- To support the sdslib in HIPPEROS. The sdslib is a run-time library required to map the software code running on the PS to the hardware design running in the PL. It is heavily related to driver code and very low-level in nature. Unfortunately, this library is closed-source. However, HIPPEROS and SUNDANCE contacted Xilinx about this and the release of the source code (under NDA) is in progress.

The support of the SDx workflow as a standard HIPPEROS toolchain is a work-in-progress task.

11 Device Drivers for the FPGA IP cores

In this chapter we describe the results and concepts using FPGA IP cores by an RTOS. While in the section of dynamic partial reconfiguration (DPR) the RTOS is based on HIPPEROS, we used FreeRTOS in the section debugging support. Because every FPGA IP core needs its own driver related to its underlying hardware the following description shows the basic concept how to develop and to use these device drivers. The designed FPGA IP cores from task T4.3 use an Advanced eXtensible Interface (AXI)-Lite interface, so that following description will describe the concept of a device driver which will access such an IP core. For other IP cores it is easy to adapt the code depending on the given interface.

11.1 Dynamic partial reconfiguration

At first, the general concept of DPR is discussed in this section. Afterwards a standalone (without the RTOS) image processing application based on the TULIPP platform is shown as an example from task T4.3 followed by the generic description how to use DPR and the FPGA IP cores with HIPPEROS.

DPR offers the flexibility to reconfigure a design with different Reconfigurable Modules (RMs) at runtime reusing the same hardware resources on the FPGA floorplan. A DPR design consists of a static part and one or more Reconfigurable Partitions (RPs) that are dynamically reconfigured with the desired Reconfigurable Module (RM), depending on the design. A set of partial bitstreams are generated for all of the RMs of the reconfigurable system. Xilinx's DPR technique is used by dynamically reconfiguring the RPs through the Processor Configuration Access Port (PCAP). The PL can be configured and reconfigured by the software running on the PS using the PCAP path in the Device Configuration Interface (DevC) [1]. The AXI-PCAP bridge is the main component in the configuration of the PL for deployment of bitstreams. The PCAP bridge converts 32-bit AXI formatted data into the 32-bit PCAP protocol and vice versa [2]. The DMA-Controller (DMA) included in the PCAP transfers the data of the bitstream between the memory device, usually the Double Data Rate (DDR) memory, into the First In First Out (FIFO) registers of the PCAP bridge over an AXI-Interconnect. The DevC driver functions have to set up the PCAP bridge into the correct mode and initialize the DMA transfer to send the full and the partial bitstream from the memory device to the FIFOs. In non-secure mode, the transfer rate through the PCAP bridge is around 145MB/s. The PL configuration module can accept data at the rate of 32 bits per PCAP clock, but the overall throughput is limited by the PS AXI interconnect [1].

The design shown in Figure 1 includes an AXI-DMA, a FIFO and two HLS cores, which will act as RMs. It has been implemented on the TULIPP hardware instance, which is based on the EMC^2-Z7030 board. Two case studies are presented here. One core is a pass-through and the second one is a sepia filter. They have been designed to have the same entity to apply the DPR technique. Therefore, they are

PUBLIC

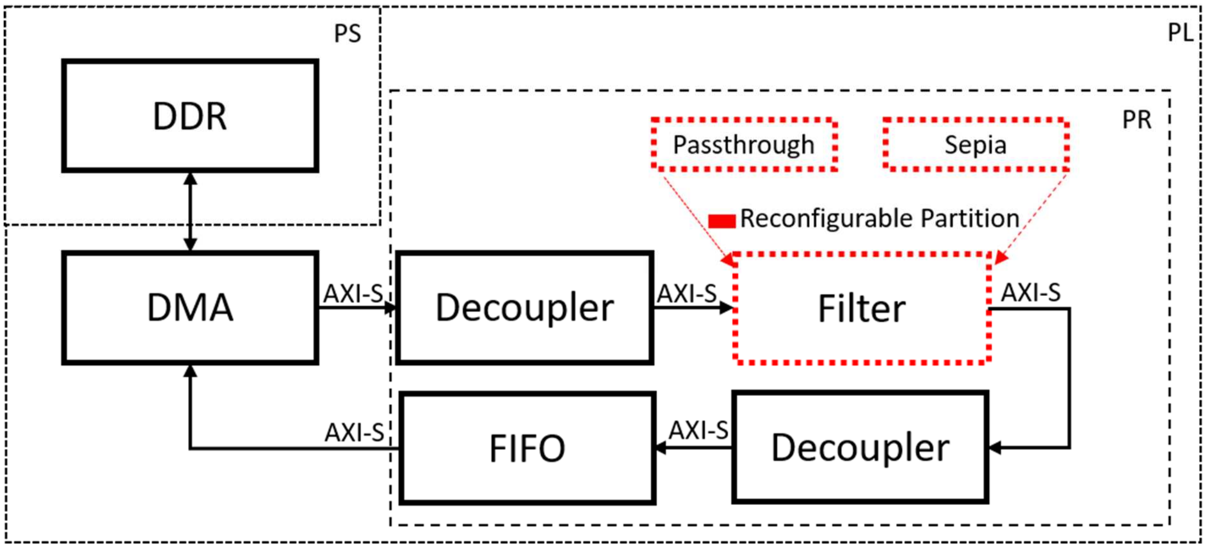

Figure 1 Static design of the standalone application

composed by an input and an output AXI-Stream interface for streaming the images. At the beginning, one RM is reconfigured. Then, an image is read from a Secure Digital (SD) card, loaded into memory and streamed via the AXI-DMA to the filter. Its output is read back from the AXI-DMA to later save the result into the SD card. It is important to highlight that any filter, based on the SDSoC-Image-Processing-Library developed by TUD [3], can be included in this design to be partially reconfigured.

During runtime two RM can be changed so that either the pass-through or sepia filter runs after reconfiguration. The images used for the case studies are 256x256 RGB in 32bit. At the beginning, all available partial bitstreams (1,250Kbytes each) are loaded from the SD card into the memory. Then, the PCAP-controller is initialized to reconfigure only one core at runtime. Despite having two cores, their reconfigurable times are the same due to the fact that all partial bitstream have the same size. The same case study is also running with FreeRTOS [7].

HIPPEROS provides support for the DPR of the FPGA which offers the flexibility to reconfigure a design with different Reconfigurable Modules (RMs) at runtime reusing the same hardware resources on the FPGA floorplan. This is particularly useful when running multiple computation-intensive applications in parallel or when the size of the FPGA is limited. Therefore, the design can fit on a smaller FPGA, which results in lower power consumption and costs. With HIPPEROS support TUD's team successfully evaluated the driver code for DPR. These general concept is described in the next part. This proof of concept is done with a minimal design to avoid issues which can occur during development of complex image processing pipelines.

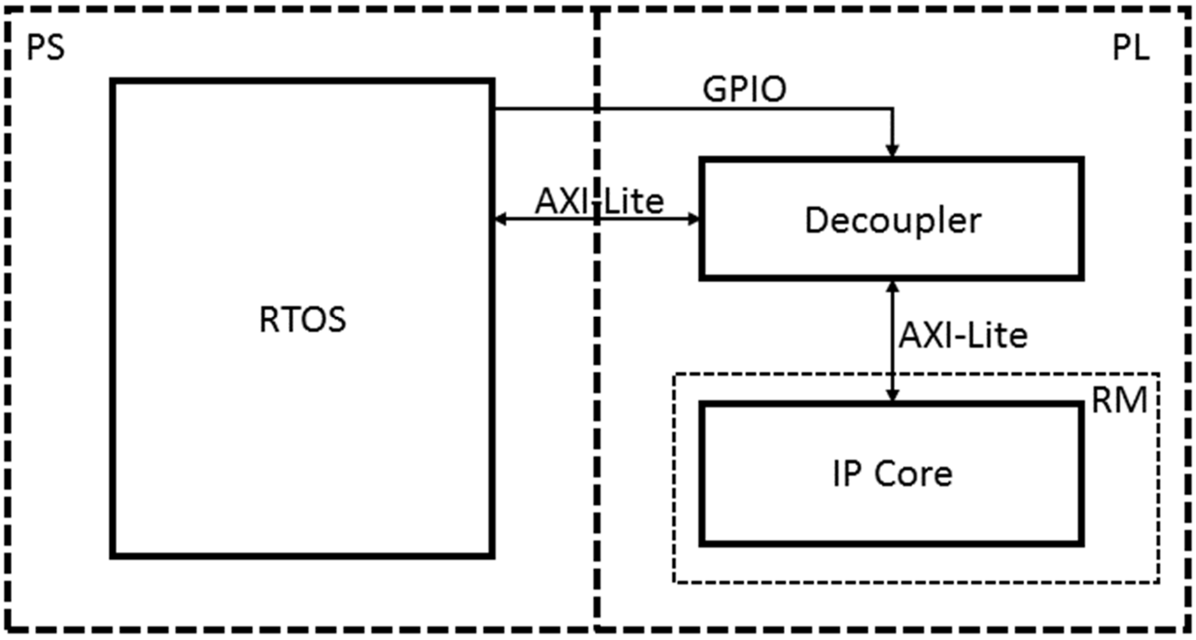

Figure 2 Generic design for DPR using the AXI-Lite interface

The generic block diagram is presented in Figure 2. The RTOS, in this case HIPPEROS, is running on the PS side and controlling the hardware of the whole system. The hardware on the PL consists of a static and a partial part. The Reconfigurable Module (RM) is the FPGA IP core which can be changed dynamically during runtime. The decoupler is part of the static design and need to decouple the used interface during reconfiguration time. It is recommended to use a decoupler, otherwise the hardware can run into undefined states driven by some unwanted signals from the RM. The decoupler is controlled by the RTOS trough the given GPIO. If the port is set to high, then the interface is decoupled. If the port is set to let low, then the RM is connected via the given interface to the PS. The decoupler is provide by Xilinx IP library. For using the GPIO, it is needed to load the GPIO driver of the given RTOS.

To be able to load a bitstream into the FPGA from the CPU, the system must provide the PCAP. The HIPPEROS-TULIPP distribution comes with the PCAP driver built-in directly usable. The user can put its collection of static and partial bitstreams into the SD card of the target board and load these bitstreams at run-time using the driver. The loading from the SD card into memory is done by the Generic FAT File System Module [4] and the provided drivers from HIPPEROS. The first loaded bitstream always has to be the static one which contains the entire hardware of the PL. After that every partial bitstream can be loaded.

For managing the hardware accelerated task a scheduler is needed. Therefore, a lot of schedule schemes are designed for reconfigurable processor architectures [5]. The particular functional tasks will be reconfigured, controlled by an operating system (OS). The OS does not know the run-time behavior of the tasks. The decision of exchanging a task can be made after running the task and takes place according to the principle of First-Come-First-Serve (FCFS) or Shortest-Job-First (SJF).

11.2 Debugging support

During the design phase, many bugs are expected. If the bugs are encountered during simulation phase, they are easy to solve. However, in many cases the bugs are encountered during the hardware phase. Hardware validation and verification are the most critical step because of the hardware invisibility. Even in case of Integrated Logic Analyzer (ILA), the monitored signals are visible only for few clock cycles depending upon the hardware resources. We presented a debugging system which can provide complete visibility and lossless stream of data effectively unlimited debug window [6]. The processor-based debugging system is utilized to collect the data from onboard trace buffers. Once the trace buffers are full, the DUT is stopped by the clock manager and then the data is transferred to the terminal through the available communication port. After the data transfer, the debugging system starts clocking the DUT again and the iterative process can continue. The data can be viewed by any waveform viewer.

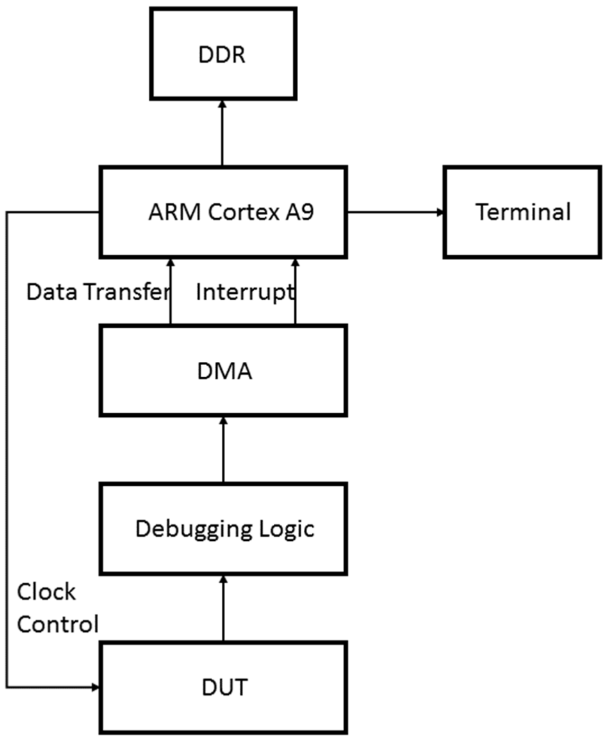

Figure 3 Hardware platform for debugging system

PUBLIC

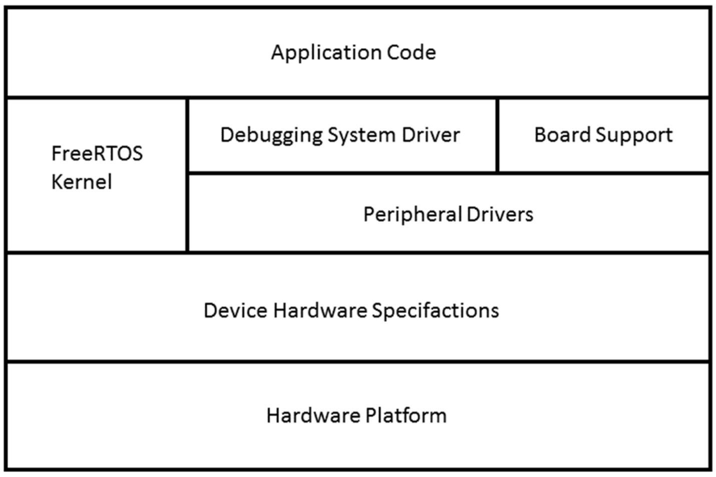

Figure 4 Debugging system software

The debugging system comprises of three modules namely the hardware platform shown in Figure 3, the debugging system software driver and a GUI running on the terminal. The software driver is generic and has been developed using C/C++ programming language. It was evaluated with FreeRTOS, but can be easily ported to other operating system. The block diagram of the driver is presented in Figure 4. The debugging system driver is based upon the peripheral drivers used in the debugging system. The debugging system contains the DMA, interrupt controller and GPIOs. Based upon the same analogy, the main ingredients of the driver are interrupt setup, DMA initialization, debugging system re-initialization in the interrupt handler and the data transmission through serial data communication.

In order to setup the debugging system, we used the same base addresses as are mentioned in the specification files. The debugging system also needs to communicate with the terminal in order to transfer data. We have used serial data communication so that other available communication channels like Ethernet, PCI etc. can be used for other application. At the terminal side, the GUI is used to setup the serial port through its link establishment button. Then it starts receiving the data. Once the data has been received and no more data transfer is required, the communication can be stopped through the Halt button. Then, the debugging data is plotted. Once the debugging is complete, the GUI can be used to close the serial port and exit the debugging. The front panel of the GUI is shown in Figure 5.

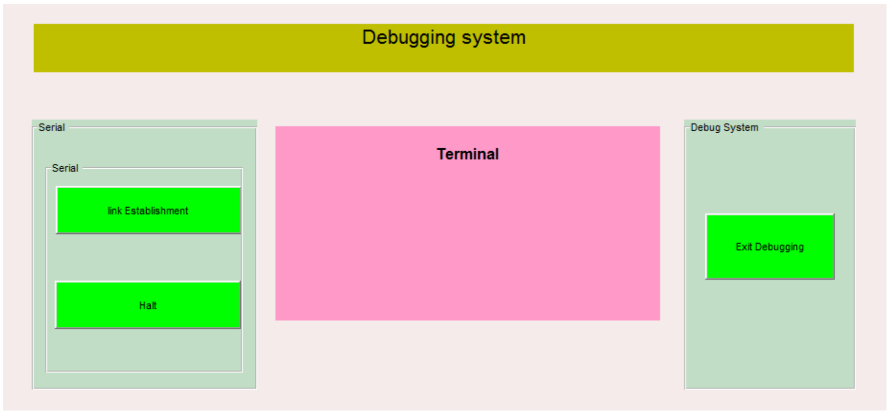

Figure 5 Graphical user interface (GUI)

12 References

[1] “Zynq-7000 SoC Technical Reference Manual UG585 (V1.12.2).” Technical Reference Manual, Zynq-7000 All Programmable SoC, Xilinx, Inc., July 2018.

[2] M. A. Kadi, P. Rudolph, D. Göhringer, and M. Hübner, “Dynamic and partial reconfiguration of zynq 7000 under linux,” in Proc. of the International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp. 1–5, Dec 2013.

[3] L.Kalms, SDSoC-Image-Processing-Library, https://github.com/tulipp-eu/tulipp-tool-chain

[4] CHaN, FatFs - Generic FAT Filesystem Module, http://elm-chan.org/fsw/ff/00index_e.html

[5] H. Walder, M. Platzner, “Online Scheduling for Block-partitioned Reconfigurable Devices”, In Design, Automation and Test in Europe, March 3-7, 2003.

[6] H. ul Hasan Khan and D. Göhringer, “Fpga debugging by a device start and stop approach,” in Proc. of the International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1–6, Nov 2016.

[7] A. Podlubne, J. Haase, L. Kalms, G. Akgün, Muhammad Ali, H. ul Hasan Khan, A. Kamal and D. Göhringer, “Low Power Image Processing Applications on FPGAs using Dynamic Voltage Scaling and Partial Reconfiguration,” In Proc. of the Conference on Design and Architectures for Signal and Image Processing (DASIP), Aug 2018. (Submitted)