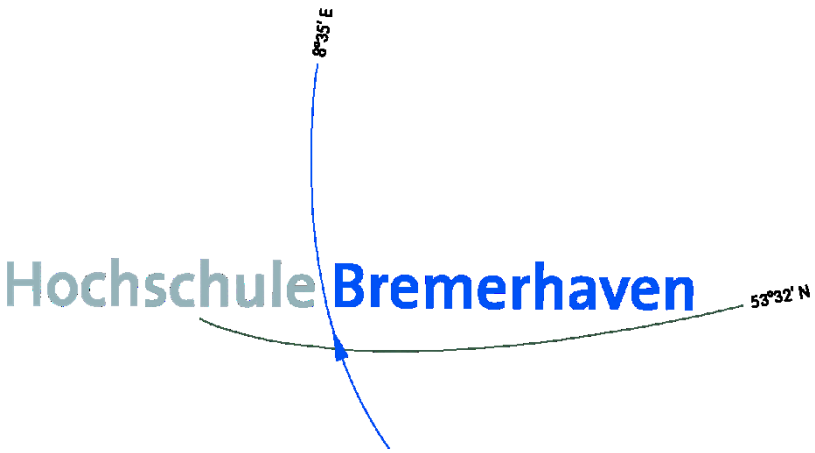

University of Applied Sciences Bremerhaven

M.Sc. Embedded Systems Design

Concept of smart avionics controller involving the integration of network protocols, data acquisition, data processing-storage, and control.

Master Thesis

Yarib Israel Nevárez Esparza

34399

Supervisors:

Prof. Dr.-Ing. Kai Müller

Prof. Dr.-Ing. Karsten Peter

Bremerhaven, Bremen (Germany). October 2017

Abstract

Embedded computer systems are rapidly evolving into smarter devices. The vehicle behind this transition is the integration of operating systems, network protocols, storage, hardware-software capabilities, and more.

This thesis achieves the conception of a smart controller by means of hardware-software codesign on Xilinx Zynq SoC with embedded Linux. The smart controller concept involves the implementation of customized peripherals, Linux device drivers, and the development of an effective software architecture.

The customized peripherals cover the hardware implementation of a configurable SPI, multiple PWMs, and GPIOs on the FPGA side of the Xilinx Zynq SoC. This also includes the adaptation of device tree, as well as the development of Linux kernel modules to support hardware peripherals on the FPGA and externals.

The software architecture is designed with the object-oriented application framework approach. Implemented in C++, and based on software design patterns, the main benefits of this software architecture are modularity, reusability, and extensibility. The application framework encapsulates low-level mechanisms and provides generic templates for end-user applications, components to support multi-threading management, background services, USB input-device handling, TCP/IP networking, infrastructure for remote monitoring and control, and conceptualization for hardware resources like sensors, actuators, and GPIOs.

The application framework allows to plug components together to build up the final user or business applications, it avoids re-designing, re-implementing, and re-validating software components. This design supplies stable base library to extend into a particular application domain by sub-classing into extended application frameworks. This strategy of reuse and inheritance provides an adaptable, low cost and highly effective software development.

Preface

This thesis is submitted to the University of Applied Sciences Bremerhaven to obtain the degree of Master of Science in Embedded Systems Design.

Acknowledgements

A strong passion for exact mathematical sciences, electronics, and software development brought me all the way here.

To my beloved family, my beloved woman, my beloved friends, and to my beloved México.

Thanks to my parents, because I speak sciences as a native language. I am still learning vocabulary.

I would like to thank the University of Applied Sciences Bremerhaven, especially the professors and the personnel involved in the Master Program ESD, for this great international program, which is a contribution from Germany to the world.

Thanks to Deutschlandstipendium, for the financial support during my master studies.

Thanks to E.I.S. Electronics GmbH, for the support of this thesis.

Thanks to Sriram P N, for sending me the link to the website.

Thanks to both Prof. Dr.-Ing. Kai Müller and Prof. Dr.-Ing. Karsten Peter, for being the driving force of the Master Program ESD.

Thanks to Germany, I have always been treated very well.

I hope you will enjoy reading this thesis.

Yarib Israel Nevárez Esparza

Bremerhaven, October 2017

Statutory declaration

I declare that I have developed and written the enclosed Master Thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. The Master Thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.

Yarib Israel Nevárez Esparza

Bremerhaven, October 2017

Contents

List of tables

List of figures

Acronyms and definitions

Acronym	Definition
ADC	Analog to Digital Converter
ADT	Abstract Data Type
AMBA	Advanced Microcontroller Bus Architecture
AP SoC	All Programmable System-on-Chip
API	Application Programming Interface
APU	Application Processor Unit
ARM	Advanced RISC Machine
ARM® Cortex™-A9	32-bit processor, ARMv7-A architecture
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Part
AXI	Advanced eXtensible Interface
Bit	Binary digit, basic unit of information
BSP	Board Support Package
Byte	Eight bits, unit of memory size.
CPHA	Clock phase (SPI)
CPOL	Clock polarity (SPI)
CPSR	Current Program Status Register
CPU	Central Processing Unit
DD	Dependence Diagram (safety)
DIN	Data Input
DOUT	Data Output
DSP	Digital Signal Processor
dtb	Device Tree Blob
dtc	Device Tree Compiler
dts	Device Tree Source
EDL	Eclipse Distribution License
EDT	Expanded Device Tree
ePAPR	Embedded Power Architecture Platform Requirements
EPL	Eclipse Public License
ext4	Fourth extended file system Linux-standard
FAT	File Allocation Table (file system)
FAT32	File Allocation Table with 32-bit cluster
FDT	Flattened Device Tree
FF	Flip-Flop
FIFO	First Input First Output
FIQ	Fast Interrupt
FPGA	Field Programmable Gate Array

FSBL	First Stage Boot Loader
FTA	Fault Tree Analysis
GCC	GNU Compiler Collection
GDB	GNU Debugger
GIT	Version control and collaboration tool
GNU	GNU's Not Unix
GPIO	General Purpose Input Output
GPL	General Public License
GUI	Graphical User Interface
HDL	Hardware Description Language
HDMI	High Definition Multimedia Interface
HID	Human Interface Device
HKMG	High-k Metal Gate (process technology)
HPL	High-Performance, Low-Power
HSAS	Heterogeneous System Architecture (foundation)
HTTP	Hypertext Transfer Protocol
I/O	Input/output
IC	Integrated Circuit
ID	Identity
IDE	Integrated Development Environment
IP	Intellectual Property, Internet Protocol
IPC	Interprocess communication
IRQ	Interrupt
ksps	Kilo Samples Per Second
JTAG	Joint Test Action Group
LED	Light-Emitting Diode
LTE	Long-Term Evolution (telecommunication)
LUT	Look-Up Table
MFC	Microsoft Foundation Classes
MIO	Multiplexed Input Output
MISO	Master Input, Slave Output (SPI)
MOSI	Master Output, Slave Input (SPI)
OO	Object-Oriented
OS	Operating System
OLED	Organic Light-Emitting Diode
PC	Personal Computer
PCB	Printed circuit board
PCI	Peripheral Component Interconnect
PL	Programmable Logic
PLL	Phase-Locked Loop
PMOD	Peripheral Module (interface)

PS	Processing System
PWM	Pulse Width Modulation
QML	Qt Modeling Language
QSPI	Quad Serial Peripheral Interface
Qt	Cross-platform software development for embedded and desktop
RAM	Random Access Memory
RFS	Root File System
RJ45	Registered Jack 45 (Ethernet connector)
SAR	Successive Approximation Register
SCLK	Serial Clock (SPI)
SCSI	Small Computer System Interface
SD	Secure Digital (non-volatile memory card)
SDK	Software Development Kit
SMP	Symmetric Multiprocessing
SoC	System on Chip
SPI	Serial Peripheral Interface
SSBL	Second Stage Bootloader
STL	Standard Template Library (C++)
TCF	Target Communications Framework
TCP	Transmission Control Protocol
TSC	Technical Steering Committee (Linaro)
UART	Universal Asynchronous Receiver-Transmitter
UI	User Interface
UML	Unified Modeling Language
UNIX	Multi-user, multitasking operating system
URL	Uniform Resource Locator
USB	Universal Serial Bus
USB OTG	USB On The Go
VFS	Virtual File System
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
x86	Family of processor architectures based on the Intel 8086 CPU and its Intel 8088 variant
XADC	Xilinx Analog to Digital Converter
XSDK	Xilinx Software Development Kit
ZYBO	Zynq Board
Zynq	All Programmable SoC with hardware and software programmability

Table 1: Acronyms and definitions

Chapter 1. **Introduction**

1.1 Motivations

The field of embedded systems has driven many advances in the industry, especially in the aviation sector. Keeping up with the state of the art technologies like FPGA, SoC, and operating systems are the key to successfully navigate through this changing field.

The motivations for this thesis are coming not only from business, but also from academic points of view, these are described below.

1.1.1 Business

This thesis is part of "The cabin convenient controller unit", which is a project originated from the development department of E.I.S. Electronics GmbH. This project can be defined as a customizable controller of water system functions for aircrafts.

The motivation of this project is to produce a convenient and reusable design of a controller unit, with the state of the art technology in hardware and software.

1.1.2 Academics

As academic motivation, the particular effort of this thesis brings skills and experience in the following areas:

- VHDL and SoC-FPGA design
- Embedded Linux device drivers and application development
- Object-oriented design, and C++
- Software design patterns, and software architecture
- TCP/IP communication and USB HID event handling

1.2 Scope

As academic work, the present thesis produce the SoC-FPGA design, operating system setup for Xilinx Zynq, and software architecture for "The cabin convenient controller unit". However, the implementation of any business logic or any customer application either in hardware or software does not fall within the scope of this thesis.

1.3 Objectives

The primary goals of this thesis are listed:

1. Linux set up for Xilinx Zynq SoC platform
2. Implementation of custom hardware peripherals on the FPGA side of the Zynq SoC
3. Development of Linux device drivers for customized hardware peripherals on FPGA, and externals
4. Design and development of software architecture
5. Development software samples for final user and business applications

6. Development of desktop and embedded software testing tools
7. Testing and evaluation of results

1.4 Methodology

The objectives of this thesis are accomplished under agile methodologies. This comprehends procedures of adaptive planning, evolutionary development, continuous delivery, continuous improvement, and it encourages rapid and flexible response to changes.

1.5 Project contribution

From the development department of E.I.S. Electronics GmbH, “The cabin convenient controller unit” is divided into three master thesis:

1. Hardware PCB design
2. SoC-FPGA design, operating system setup, and software architecture
3. System and functional verification

The present thesis is focused on the SoC-FPGA design, operating system setup, and software architecture.

Chapter 2. **Background**

2.1 Embedded systems

An embedded system is a computerized system which is built for a specific application or purpose. Since its destination is reduced than a general purpose computer, an embedded system has less support for resources that are unrelated to running the application. The hardware and software often have constraints: for instance, a CPU that runs more slowly to save battery power; less memory so it can be cheaper; specific processors that support a subset of peripherals, etc.

In some systems, the software must act deterministically (exactly the same each time) or real-time (always reacting to an event fast enough). Some systems require that the software be fault tolerant with graceful degradation in the face of errors. For example, consider a system here servicing faulty software or broken hardware may be infeasible (e.g. a satellite or a tracking tag on a whale). Other systems require that the software cease operation at the first sign of trouble, often providing clear error messages (a heart monitor should not fail quietly). [1]

2.2 FPGA

An FPGA is an integrated circuit (IC) that can be programmed for different algorithms after fabrication. Modern FPGAs consist of up to two million logic cells that can be configured to implement a variety of software algorithms. Although the traditional FPGA design flow is more similar to a regular IC than a processor, an FPGA provides significant cost advantages in comparison to an IC development effort and offers the same level of performance in most cases. Another advantage of the FPGA when compared to the IC is its ability to be dynamically reconfigured. This process, which is the same as loading a program in a processor, can affect part or all of the resources available in the FPGA fabric.

FPGAs are heterogeneous compute platforms that include Block RAMs, DSP Slices, PCI Express support, and programmable fabric. They enable parallelism and pipelining of applications across the entire platform as all of these computer resources can be used simultaneously.

The basic structure of a FPGA is composed of the following elements:

- Look-up table (LUT) - This element performs logic operations.
- Flip-Flop (FF) - This register element stores the result of the LUT.
- Wires - These elements connect elements to one another.
- Input/output (I/O) pads - These physical ports get data in and out of the FPGA.

The combination of these elements results in the basic FPGA architecture shown in the following figure from Xilinx website. Although this structure is sufficient for the implementation of any algorithm, the efficiency of the resulting implementation is limited in terms of computational throughput, required resources, and achievable clock frequency.

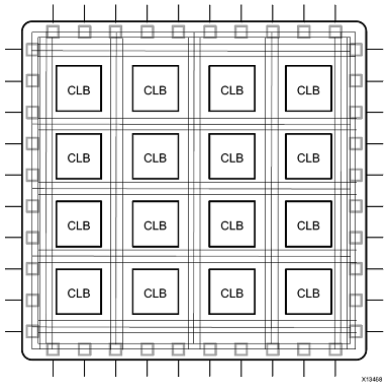

Figure 1: Basic FPGA architecture

Contemporary FPGA architectures incorporate the basic elements along with additional computational and data storage blocks that increase the computational density and efficiency of the device. These additional elements, which are discussed in the following sections, include:

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

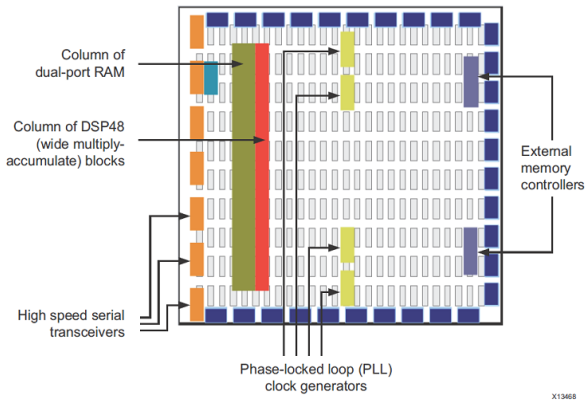

Figure 2: Contemporary FPGA architecture

From Xilinx website, the previous figure shows the combination of these elements on a contemporary FPGA architecture. This provides the FPGA with the flexibility to implement any software algorithm running on a processor. Note that all of these elements across the entire FPGA can be used concurrently. [2]

2.3 Zynq-7000 All Programmable SoC

The Zynq®-7000 family is based on the Xilinx® All Programmable SoC (AP SoC) architecture. These products integrate a feature-rich dual or single-core ARM® Cortex™-A9 MPCore™ based processing system (PS) and Xilinx programmable logic (PL) in a single device, built on a state-of-the-art, high-performance, low-power (HPL), 28 nm, and high-k metal gate (HKMG) process technology. The ARM

Cortex-A9 MPCore CPUs are the heart of the PS which also includes on-chip memory, external memory interfaces, and a rich set of I/O peripherals. [3]

The Zynq-7000 family offers the flexibility and scalability of an FPGA, while providing performance, power, and ease of use typically associated with ASIC and ASSPs. The range of devices in the Zynq-7000 AP SoC family enables designers to target cost-sensitive as well as high-performance applications from a single platform using industry-standard tools. While each device in the Zynq-7000 family contains the same PS, the PL and I/O resources vary between the devices. As a result, the Zynq-7000 AP SoC devices are able to serve a wide range of applications including:

- Automotive driver assistance, driver information, and infotainment
- Broadcast camera
- Industrial motor control, industrial networking, and machine vision
- IP and smart camera
- LTE radio and baseband
- Medical diagnostics and imaging
- Multifunction printers
- Video and night vision equipment

The processor(s) in the PS always boot first, allowing a software-centric approach for PL system boot and PL configuration. The PL can be configured as part of the boot process, or configured at some point in the future. Additionally, the PL can be completely reconfigured or used with partial, dynamic reconfiguration (PR). PR allows configuration of a portion of the PL. This enables optional design changes such as updating coefficients or time-multiplexing of the PL resources by swapping in new algorithms as needed. This latter capability is analogous to the dynamic loading and unloading of software modules. The PL configuration data is referred to as a Bitstream.

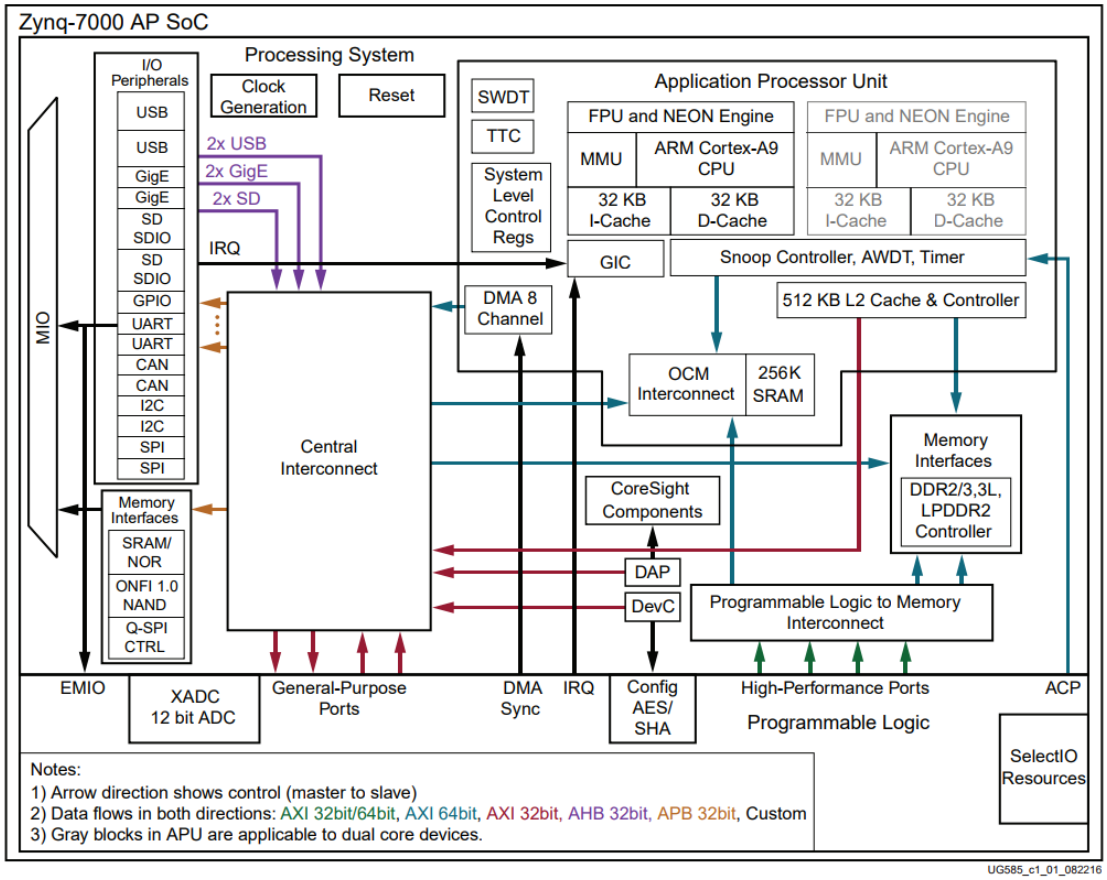

Figure 3: Zynq-7000 AP SoC overview

The Zynq-7000 AP SoC is composed of the following major functional blocks:

- Processing System (PS)
  - Application processor unit (APU)
  - Memory interfaces
  - I/O peripherals (IOP)
  - Interconnect
- Programmable Logic (PL)

2.4 Fundamental concepts of Linux kernel

The core software that manages and allocates system resources (i.e., CPU, memory, and devices). Although it is possible to run programs on a computer without a kernel, the assistance of a kernel highly simplifies the development and use of other programs, and boost the capacity and flexibility available to programmers. The kernel does this by providing a software layer to handle the resources of a computer. The following are the main tasks that the kernel performs:

2.4.1 Process scheduling

Linux is a preemptive multitasking operating system, this means that multiple processes (programs) can reside in memory at the same time and each may get use of the CPU(s). Preemptive means that the kernel process scheduler determines which processes get use of the CPU and for how long.

2.4.2 Memory management

Memory (RAM) is a resource that the kernel must share among processes in an efficient fashion. Linux employs virtual memory management, one of its main advantages is process isolation, it means that a process cannot access memory assigned to another process or the kernel.

2.4.3 File system

Linux is heavily based on the file system concept; almost everything in Linux can be treated as a file. The kernel builds a structured file system on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple file system types, that is, different ways of organizing data on the physical medium. For example, disks may be formatted with the new Linux-standard ext4 file system, the commonly used FAT file system or several others. [4]

2.4.4 Creation and termination of processes

The kernel loads programs into memory, providing it with the resources (e.g., CPU, memory, and files access). The instance of a running program is called a process. Once a process ends execution, the kernel ensures that the resources it used are freed for reuse by later programs.

2.4.5 Access to devices

The devices connected to a computer allow data communication between the system and the external world, allowing input, output, or both. The kernel provides the means with a simplified and standard interface to access the devices, at the same time moderates the access by multiple processes to each device.

2.4.6 Networking

Networking must be managed by the operating system, since most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel. [4]

2.4.7 System call

The system call are the kernel entry points, by using it processes can request the kernel to perform various tasks. The system call can be considered as an application programming interface (API).

In addition to the previously described features, Linux is a multiuser operating systems providing the abstraction of a virtual private computer to each user; it means, each user log on to the system and operate independently of other users. Each user has their own disk storage space (in home folder), users can run programs getting a share of the CPU and its own virtual address space, these programs independently access the devices and transfer data over the network. The kernel resolves possible conflicts in accessing resources, therefore in most of the cases, users and processes are unaware of the conflicts. [5]

2.5 Kernel mode and user mode

Modern processor architectures typically allow the CPU to operate in at least two different modes: user mode and kernel mode (sometimes referred to as supervisor mode). CPU instructions allow switching

between modes. Correspondingly, virtual memory areas can be assigned to user space or kernel space. When running in user mode, the CPU is able to access only memory that is assigned to user space; any attempt to access memory in kernel space result in a hardware exception. When running in kernel mode, the CPU is able to access both user and kernel memory space.

Certain instructions can be executed only while the CPU is running in kernel mode. Examples include executing the halt instruction to stop the system, accessing the memory-management hardware, and initiating device I/O operations. By taking advantage of this CPU feature to set the operating system in supervisor mode (kernel mode), operating system developers ensure that user processes cannot access the instructions and data structures of the kernel, or to execute instructions that adversely affect the system operation. [5]

2.6 Operating modes ARMv7 architecture

The ARM7 processor family has seven modes of operation:

- User mode is the usual ARM program execution state, and it is used for executing most application programs.
- Fast Interrupt (FIQ) mode supports a data transfer or channel process.
- Interrupt (IRQ) mode is used for general-purpose interrupt handling.
- Supervisor mode is a protected mode for the operating system.
- Abort mode is entered after a data or instruction Prefetch Abort.
- System mode is a privileged user mode for the operating system.
- Undefined mode is entered when an undefined instruction is executed.

It can only be entered into System mode from another privileged mode by modifying the mode bit of the Current Program Status Register (CPSR).

Modes other than User mode are collectively known as privileged modes. Privileged modes are used to service interrupts or exceptions, or to access protected resources. [6]

2.7 Kernel modules

The Linux kernel is modular. Modules are pieces of software that can be loaded and unloaded into the kernel upon demand. They dynamically extend kernel functionalities without the need to reboot the system. One advantage is that the module can be under development and test, loading and unloading it without the need of rebuilding or rebooting the main kernel.

Kernel modules obviously run in kernel mode, hence they have access to hardware. Therefore, a device driver is written as a kernel module.

Without modules, we would have to build monolithic kernels and add new functionalities directly into the kernel image itself. Besides having larger kernels. During development, one disadvantage is that it requires rebuild and reboot the kernel every time a new functionality is needed (even to test the smallest change).

2.8 Device drivers

One of the purposes of an operating system is to hide the peculiarities of the system's hardware devices from its users. For example, the Virtual File System presents a uniform view of the mounted file systems irrespective of the underlying physical devices. The Linux kernel device drivers are, essentially, a shared library of privileged, memory resident, and low-level hardware handling routines. It is Linux device drivers that handle the peculiarities of the devices they are managing.

One of the basic features of a device driver is that it abstracts the handling of devices. All hardware devices look like regular files; they can be opened, closed, read and written using the same, standard, system calls that are used to manipulate files. Every device in the system is represented by a device special file. All devices controlled by the same device driver have a common major device number. The minor device numbers are used to distinguish between different devices and their controllers.

Linux supports three types of hardware device: character, block, and network. Character devices are read and written directly without buffering, for example, a GPIO. Block devices can only be written to and read from in multiples of the block size, typically 512 or 1024 bytes. Block devices are accessed via the buffer cache and may be randomly accessed, that is to say, any block can be read or written no matter where it is on the device. Block devices can be accessed via their device special file but more commonly they are accessed via the file system. Only a block device can support a mounted file system, hard disks are block devices. Network devices are accessed via the BSD socket interface and the networking subsystems.

There are many different device drivers in the Linux kernel (that is one of Linux's strengths) but they all share some common attributes:

- **Kernel code.** Device drivers are part of the kernel and, like other code within the kernel, if they go wrong they can seriously damage the system. A badly written driver may even crash the system, possibly corrupting file systems and losing data,
- **Kernel interfaces.** Device drivers must provide a standard interface to the Linux kernel or to the subsystem that they are part of. For example, GPIO driver provides a file I/O interface to the Linux kernel.
- **Kernel mechanisms and services.** Device drivers make use of standard kernel services such as memory allocation, interrupt delivery and wait-queues to operate.
- **Loadable.** Most of the Linux device drivers can be loaded on demand as kernel modules when they are needed and unloaded when they are no longer being used. This makes the kernel very adaptable and efficient with the system's resources.
- **Configurable.** Linux device drivers can be built into the kernel. Which devices are built is configurable when the kernel is compiled.
- **Dynamic.** As the system boots and each device driver is initialized it looks for the hardware devices that it is controlling. It does not matter if the device being controlled by a particular device driver does not exist. In this case, the device driver is simply redundant and causes no harm apart from occupying a little of the system's memory.

2.8.1 Polling and Interrupts

Polling the device usually means reading its status register every so often until the device's status changes to indicate that it has completed the request. As a device driver is part of the kernel it would not be time efficient if a driver were to poll as nothing else in the kernel would run until the device had completed the request, since devices are generally slower than the kernel. Instead, polling device drivers use system timers to have the kernel call a routine within the device driver at some later time. Polling by means of timers is at best approximate, a much more efficient method is to use interrupts.

An interrupt driven device driver is one where the hardware device being controlled will raise a hardware interrupt whenever it needs attention. For example, an Ethernet device driver would interrupt whenever it receives an Ethernet packet from the network. The Linux kernel needs to be able to deliver the interrupt from the hardware device to the correct device driver. This is achieved by the device driver registering its usage of the interrupt with the kernel. It registers the address of an interrupt handling routine and the interrupt number that it wishes to own. You can see which interrupts are being used by the device drivers, as well as how many of each type of interrupts there have been, by looking at */proc/interrupts*.

This requesting of interrupt resources is done at driver initialization time. Some of the interrupts in the system are fixed, in the case of desktop computers, this is a legacy of the IBM PC's architecture. In embedded devices, it depends on its architecture.

How one interrupt is delivered to the CPU itself, it is architecture dependent but on most architectures, the interrupt is delivered in a special mode that stops other interrupts from happening in the system. A device driver should do as little as possible in its interrupt handling routine so that the Linux kernel can dismiss the interrupt and return to what it was doing before it was interrupted. Device drivers that need to do a lot of work as a result of receiving an interrupt can use the kernel's bottom half handlers or task queues to queue routines to be called later on.

2.8.2 Kernel interface

The Linux kernel must be able to interact with drivers in standard ways. Each class of device driver, character, block, and network, provides common interfaces that the kernel uses when requesting services from them. These common interfaces mean that the kernel can treat often very different devices and their device drivers absolutely the same. In other words, each device has its particular mechanism but all of them are treated as files with standard file operations.

Linux allows you to include device drivers at kernel build time via its configuration scripts. When these drivers are initialized at boot time they may not discover any hardware to control. Other drivers can be loaded as kernel modules when they are needed. To cope with this dynamic nature of device drivers, device drivers register themselves with the kernel as they are initialized. Linux maintains tables of registered device drivers as part of its interfaces with them. These tables include pointers to routines and information that support the interface with that class of devices.

2.8.3 Character devices

Character devices, the simplest of Linux's devices, are accessed as files, applications use standard system calls to open them, read from them, write to them and close them exactly as if the device were a file. As a

character device is initialized its device driver registers itself with the Linux kernel by adding an entry into the kernel data structures.

Each entry in the kernel module data structure contains at least two elements; a pointer to the name of the registered device driver and a pointer to a block of file operations. This block of file operations is itself the addresses of routines within the device character device driver each of which handles specific file operations such as open, read, write and close. The contents of */proc/devices* for character devices are taken from the chrdevs vector.

When a special file representing a character device is opened the kernel must set things up so that the correct character device driver's file operation routines will be called. Just like an ordinary file, each device special file is represented by a VFS inode.

Each VFS inode has associated with it a set of file operations and these are different depending on the file system object that the inode represents. Whenever a VFS inode representing a character special file is created, its file operations are set to the default character device operations. The file operations are described in more detail in the following topic.

2.8.4 Block devices

Block devices also support being accessed like files. The mechanisms used to provide the correct set of file operations for the opened block special file are very much the same as for character devices. Linux maintains the set of registered block devices as the blkdevs vector. It, like the chrdevs vector, is indexed using the device's major device number. Unlike character devices, there are classes of block devices. SCSI devices are one such class and IDE devices are another. It is the class that registers itself with the Linux kernel and provides file operations to the kernel. The device drivers for a class of block device provide class-specific interfaces to the class.

2.8.5 Network devices

A network device is an entity that sends and receives packets of data. This is normally a physical device such as an ethernet module. Some network devices though are software only such as the loopback device which is used for sending data to yourself. Each network device is represented by a device data structure. Network device drivers register the devices that they control with Linux during network initialization at kernel boot time. The device data structure contains information about the device and the addresses of functions that allow the various supported network protocols to use the device's services. These functions are mostly concerned with transmitting data using the network device. The device uses standard networking support mechanisms to pass received data up to the appropriate protocol layer.

2.9 Basic kernel module structure

This module defines two functions, one to be invoked when the module is loaded into the kernel (`hello_init`) and one for when the module is removed (`hello_exit`). The `module_init` and `module_exit` lines use special kernel macros to indicate the role of these two functions. Another special macro (`MODULE_LICENSE`) is used to tell the kernel that this module bears a free license; without such a declaration, the kernel complains when the module is loaded.

The `printk` function is defined in the Linux kernel and made available to modules; it behaves similarly to the standard C library function `printf`. The kernel needs its own printing function because it runs by itself, without the help of the C library. The module can call `printk` because, after `insmod` has loaded it, the module is linked to the kernel and can access the kernel's public symbols. The string `KERN_ALERT` is the priority of the message. It is specified a high priority in this module, because a message with the default priority might not show up in terminal, depending on the kernel version is running, the version of the `klogd` daemon, and your configuration. [4]

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
 printk(KERN_ALERT "Hello, world\n");
 return 0;
}
static void hello_exit(void)
{
 printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

The code of the kernel modules that were implemented for the prototype are listed in the Appendix B.

2.10 File operations

An open device is identified internally by a file structure, and the kernel uses the `file_operations` structure to access the driver's functions. The structure, defined in *linux/fs.h* (Linux kernel header files), is an array of function pointers. The operations are mostly in charge of implementing the system calls and are thus named open, read, and so on. We can consider the file to be an "object" and the functions operating on it to be its "methods," using object-oriented programming terminology to denote actions declared by an object to act on itself.

Conventionally, a `file_operations` structure or a pointer to one is called fops (or some variation thereof); which is sent as a pointer as an argument to the `register_chrdev` call (function to register the kernel module as char device). Each field in the structure must point to the function in the driver that implements a specific operation, or be left `NULL` for unsupported operations.

The `file_operations` structure has been slowly getting bigger as new functionality is added to the kernel. The addition of new operations can, of course, create portability problems for device drivers. Instantiations of the structure in each driver used to be declared using standard C syntax, and new operations were normally added to the end of the structure; a simple recompilation of the drivers would place a `NULL` value for that operation, thus selecting the default behaviour, usually what you wanted.

The following list introduces all the operations that an application can invoke on a device.

```
loff_t (*llseek) (struct file *, loff_t, int);
```

The `llseek` method is used to change the current read/write position in a file, and the new position is returned as a (positive) return value. The `loff_t` is a "long offset" and is at least 64 bits wide even on 32-bit platforms. Errors are signalled by a negative return value. If the function is not specified for the driver, one seek relative to end-of-file fails, while other seeks succeed by modifying the position counter in the file structure.

```
size_t (*read) (struct file *, char *, size_t, loff_t *);
```
Used to retrieve data from the device. A null pointer in this position causes the read system call to fail with **-EINVAL** ("Invalid argument"). A non-negative return value represents the number of bytes successfully read (the return value is a "signed size" type, usually the native integer type for the target platform).

```
size_t (*write) (struct file *, const char *, size_t, loff_t *);
```
Sends data to the device. If missing, -EINVAL is returned to the program calling the write system call. The return value, if non-negative, represents the number of bytes successfully written.

```
int (*readdir) (struct file *, void *, filldir_t);
```
This field should be **NULL** for device files; it is used for reading directories, and is only useful to file systems.

```
unsigned int (*poll) (struct file *, struct poll_table_struct *);
```
The poll method is the back end of two system calls, poll and select, both used to inquire if a device is readable or writable or in some special state. Either system call can block until a device becomes readable or writable. If a driver doesn't define its poll method, the device is assumed to be both readable and writable, and in no special state. The return value is a bit mask describing the status of the device.

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```
The **ioctl** system call offers a way to issue device-specific commands (like formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few ioctl commands are recognized by the kernel without referring to the fops table. If the device doesn't offer an ioctl entry point, the system call returns an error for any request that isn't predefined (**-ENOTTY**, "No such ioctl for device"). If the device method returns a non-negative value, the same value is passed back to the calling program to indicate successful completion.

```
int (*mmap) (struct file *, struct vm_area_struct *);
```
mmap is used to request a mapping of device memory to a process's address space. If the device doesn't implement this method, the **mmap** system call returns **-ENODEV**.

```
int (*open) (struct inode *, struct file *);
```
Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this entry is NULL, opening the device always succeeds, but your driver isn't notified.

```
int (*flush) (struct file *);
```
The flush operation is invoked when a process closes its copy of a file descriptor for a device; it should execute (and wait for) any outstanding operations on the device.

```
int (*release) (struct inode *, struct file *);
```
This operation is invoked when the file structure is being released. Like open, release can be missing

```
int (*fsync) (struct inode *, struct dentry *, int);
```
This method is the back end of the **fsync** system call, which a user calls to flush any pending data. If not implemented in the driver, the system call returns **-EINVAL**.

```
int (*fasync) (int, struct file *, int);
```
This operation is used to notify the device of a change in its **FASYNC** flag.

```
int (*lock) (struct file *, int, struct file_lock *);
```
The **lock** method is used to implement file locking; locking is an indispensable feature for regular files, but is almost never implemented by device drivers.

```
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *); ,
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
```

These methods, added late in the 2.3 (kernel version) development cycle, implement scatter/gather read and write operations. Applications occasionally need to do a single read or write operation involving multiple memory areas; these system calls allow them to do so without forcing extra copy operations on the data.

`struct module *owner;`
This field isn't a method like everything else in the `file_operations` structure. Instead, it is a pointer to the module that "owns" this structure; it is used by the kernel to maintain the module's usage count.

2.11 The /proc file system

It is known that Linux treats most things as files and that there are entries in the file system for hardware devices. These */dev* files are used to access hardware in a specific way using low-level system calls.

The software drivers that control hardware can often be configured in certain ways, or are capable of reporting information. In recent years, there has been a trend toward providing a more consistent way of accessing driver information, and, in fact, to extend this to include communication with various elements of the Linux kernel.

Linux provides a special file system, procfs, that is usually made available as the directory */proc*. It contains many special files that allow higher-level access to driver and kernel information. Applications can read and write these files to get information and set parameters as long as they are running with the correct access permissions.

The files that appear in */proc* will vary from system to system, and more are included with each Linux release as more drivers and facilities support the procfs file system.

For the thesis prototype, the kernel modules of controller, ADC, PWMs create entry files in the procfs, these are in fact virtual files (not allocated on disk). These files provide the bridge from the application through the kernel; inside these modules, it is possible to access the corresponding hardware registers (kernel mode capabilities).

2.12 Device tree

The main reason for the existence of Device Tree in Linux is to provide a way to describe non-discoverable hardware. This information was previously hardcoded in source code. Device Tree data can be represented in several different formats. It is derived from the device tree format used by Open Firmware to encapsulate platform information. The device tree data is typically created and maintained in a human-readable format in .dts source files and .dtsi source include files.

The device tree source is compiled into a binary format contained in a .dtb blob file. The format of the data in the .dtb blob file is commonly referred to as a Flattened Device Tree (FDT). The Linux operating system uses the device tree data to find and register the devices in the system. The FDT is accessed in the raw form during the very early phases of boot, but is expanded into a kernel internal data structure known as the Expanded Device Tree (EDT) for more efficient access for later phases of the boot and after the system has completed booting.

Currently, the Linux kernel can read device tree information in the ARM, x86, Microblaze, PowerPC, and Sparc architectures. There is interest in extending support for device trees to other platforms, to unify the handling of platform description across kernel architectures.

The Flattened Device Tree (FDT) is a data structure. It describes a machine hardware configuration. It is derived from the device tree format used by Open Firmware. The format is expressive and able to describe most board design aspects including:

- The number and type of CPUs
- Base addresses and size of RAM
- Busses and bridges
- Peripheral device connections
- Interrupt controllers and IRQ line connections
- Pin multiplexing

Just like initrd images, an FDT image can either be statically linked into the kernel or passed to the kernel at boot time.

2.12.1 Format

The device tree is a simple tree structure of nodes and properties. Properties are key-value pairs, and the node may contain both properties and child nodes. For example, the following is a simple tree in the .dts format:

```
/dts-v1/;
/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        // hex is implied in byte arrays. no '0x' prefix is required
        a-byte-data-property = [01 23 34 56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};
```

2.12.2 Nodes

Every node must have a name in the form `<name>[@<unit-address>]`.

`<name>` is a simple ASCII string and can be up to 31 characters in length. In general, nodes are named according to what kind of device it represents.

The unit-address is included if the node describes a device with an address. In general, the unit address is the primary address used to access the device.

Sibling nodes must be uniquely named, but it is normal for more than one node to use the same generic name as long as the address is different (e.g. `serial@101f1000` & `serial@101f2000`). More details in section 2.2.1 of the ePAPR spec.

Every device in the system is represented by a device tree node. Every node in the tree that represents a device is required to have the compatible property. Compatible is the key an operating system uses to decide which device driver to bind to a device.

Compatible is a list of strings. The first string in the list specifies the exact device that the node represents in the form "`<manufacturer>,<model>`". The following strings represent other devices that the device is compatible with.

This practice allows existing device drivers to be bound to a newer device, while still uniquely identifying the exact hardware. [7]

2.12.3 Addressing

Devices that are addressable use the following properties to encode address information into the device tree:

`reg`
`#address-cells`
`#size-cells`
Each addressable device gets a *reg* which is a list of tuples in the form `reg = <address1 length1 [address2 length2] [address3 length3] ... >`. Each tuple represents an address range used by the device. Each address value is a list of one or more 32 bits integers called cells. Similarly, the length value can either be a list of cells, or empty.

Since both the address and length fields are variable of variable size, the #address-cells and #size-cells properties in the parent node are used to state how many cells are in each field. Or in other words, interpreting a `reg` property correctly requires the parent node's `#address-cells` and `#size-cells` values.

2.12.4 Memory mapped devices

A memory mapped device is assigned a range of addresses that it will respond to. `#size-cells` is used to state how large the length field is in each child `reg` tuple. In the following example, each address value is 1 cell (32 bits), and each length value is also 1 cell, which is typical of 32-bit systems. 64-bit machines may use a value of 2 for `#address-cells` and `#size-cells` to get 64-bit addressing in the device tree. [8]

```
/dts-v1/;

/ {
    #address-cells = <1>;
    #size-cells = <1>;
...
            adc
        {
            compatible = "yarib-adc-1.00.a";
            reg = <0x43C00000 0x10000>;
        } ;
        controller
        {
            compatible = "yarib-controller-1.00.a";
            reg = <0x41200000 0x10000>;
        } ;
        pwm_0
        {
            compatible = "yarib-pwm_0-1.00.a";
```

```
            reg = <0x42800000 0x10000>;
        } ;
        pwm_1
        {
            compatible = "yarib-pwm_1-1.00.a";
            reg = <0x42810000 0x10000>;
        } ;
...
} ;
```

This example shows the memory mapped devices that were added to the device tree of the used platform.

2.13 Daemons

A daemon is a process with the following features:

- It is long-lived. Usually, a daemon is created at system startup and runs until the system is shut down.
- It runs in the background and has no controlling terminal.

Daemons are written to carry out specific tasks, as illustrated by the following examples:

- cron: a daemon that executes commands at a scheduled time.
- sshd: the secure shell daemon, which permits logins from remote hosts using a secure communications protocol.
- httpd: the HTTP server daemon (Apache), which serves web pages.
- inetd: the Internet superserver daemon, which listens for incoming network connections on specified TCP/IP ports and launches appropriate server programs to handle these connections.

The software prototype for this thesis is a daemon application, this means that the application runs in background from system startup until shutdown, with no controlling terminal. And it has access to the device drivers of the peripherals.

To become a daemon, a program performs the following steps [5].

1. Perform a `fork()`, after which the parent exits and the child continues. (As a consequence, the daemon becomes a child of the init process.) This step is done for two reasons:
    a. Assuming the daemon was started from the command line, the parent's termination is noticed by the shell, which then displays another shell prompt and leaves the child to continue in the background.
    b. The child process is guaranteed not to be a process group leader, since it inherited its process group ID from its parent and obtained its own unique process ID, which differs from the inherited process group ID. This is required in order to be able to successfully perform the next step.
2. The child process calls `setsid()` to start a new session and free itself of any association with a controlling terminal.
3. If the daemon never opens any terminal devices thereafter, then we don't need to worry about the daemon reacquiring controlling terminal. If the daemon might later open a terminal device, then we must take steps to ensure that the device does not become the controlling terminal. We can do this in two ways:
    a. Specify the `O_NOCTTY` flag on any `open()` that may apply to a terminal device.

b. Alternatively, and more simply, perform a second `fork()` after the `setsid()` call, and again have the parent exit and the (grand) child continue. This ensures that the child is not the session leader, and the process can never reacquire a controlling terminal

4. Clear the process umask, to ensure that, when the daemon creates files and directories, they have the requested permissions.
5. Change the process's current working directory, typically to the root directory (/). This is necessary because a daemon usually runs until system shutdown; if the daemon's current working directory is on a file system other than the one containing / , then that file system can't be unmounted. Alternatively, the daemon can change its working directory to a location where it does its job or a location defined in its configuration file, as long as it is known that the file system containing this directory never needs to be unmounted.
6. Close all open file descriptors that the daemon has inherited from its parent. (A daemon may need to keep certain inherited file descriptors open, so this step is optional, or open to variation.) This is done for a variety of reasons. Since the daemon has lost its controlling terminal and is running in the background, it makes no sense for the daemon to keep file descriptors 0, 1, and 2 open if these refer to the terminal. Furthermore, we can't unmount any file systems on which the long-lived daemon holds files open. And, as usual, we should close unused open file descriptors because file descriptors are a finite resource.
7. After having closed file descriptors 0, 1, and 2, a daemon normally opens */dev/null* and uses `dup2()` (or similar) to make all those descriptors refer to this device. This is done for two reasons:
    a. It ensures that if the daemon calls library functions that perform I/O on these descriptors, those functions won't unexpectedly fail.
    b. It prevents the possibility that the daemon later opens a file using descriptor 1 or 2, which is then written to and thus corrupted by a library function that expects to treat these descriptors as standard output and standard error.

2.14 Linux services

A Linux service is an application (or set of applications) that runs in the background waiting to be used, or carrying out essential tasks. The system uses the configuration files (and/or directories) in */etc/rc.d* file system to control the services to be started. The Linux start-up process uses parameters from */etc/inittab* to identify the default runlevel and the files that will be used by that runlevel.

Runlevel 1 is the most basic configuration (simple single user access using text interface), while runlevel 5 is the most advanced (multi-user, networking, and a GUI front end). Runlevels 0 and 6 are used for halting and rebooting the system.

There are, however, differences between Linux distributions. Therefore, the documentation should be checked before making any changes. The following table shows a generic list of configurations taken from Linux - The Complete Reference (R.Peterson, Osbourne/McGraw-Hill) [9].

Runlevel	Generic	Debian
0	Halt	Halt
1	Single-user mode	Single-user mode

2	Basic multi-user mode (without networking)	Multi-user mode
3	Full (text based) multi-user mode	-
4	Not used	Multi-user mode
5	Full (GUI based) multi-user mode	Multi-user mode
6	Reboot	Reboot

Table 2: Linux run levels

The software prototype implemented for this thesis is a Linux background service, it begins its execution during Linux starting up, and it stops when shutting down. This prototype was configured as a Linux service. The following code shows the script used in the configuration for the Linux service.

```
#! /bin/sh
### BEGIN INIT INFO
# Author:            Yarib Nevárez
# Required-Start:    controller service
# Required-Stop:     controller service
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Start daemon at boot time
# Description:       Enable service provided by daemon.
### END INIT INFO

case "$1" in
  start)
    echo "Starting controller service."
    insmod /etc/controller-manager/controller
    insmod /etc/controller-manager/adc
    insmod /etc/controller-manager/pwm_0
    insmod /etc/controller-manager/pwm_1
    nice -n -15 /etc/controller-manager/ctrlapp -daemon -server -ir -leakage
    ;;
  stop)
    echo "Stopping controller service."
    killall ctrlapp
    rmmod controller
    rmmod adc
    rmmod pwm_0
    rmmod pwm_1
    ;;
  *)
    echo "Usage: service controller-manager {start|stop}"
    exit 1
    ;;
esac

exit 0
```

This script is stored in */etc/init.d* named *controller-manager*. Afterwards it was set as executable using *chmod 755 /etc/init.d/controller-manager*, and finally installed as a service with *update-rc.d controller-manager defaults*.

For this thesis prototype, all the kernel modules and the daemon application are located in */etc/controller-manager/* (it can be seen in the script). The script should be called with one parameter, it can be either *start* or *stop*. The start parameter inserts the kernel modules into the kernel and finally starts the execution of ctrlapp (background daemon); the application gets some parameters, these define application features that are going to be detailed in next chapters. The stop parameter kills the daemon process, and removes all the kernel modules that were previously inserted by the start parameter.

If it is needed to stop the daemon service, it can be done in the shell terminal at any time with *service controller-manager stop*. If it is needed to uninstall the service it can be done with *update-rc.d -f controller-manager remove*.

2.15 Interprocess communication

Interprocess communication (IPC) is the transfer of data among processes. The five most common types of interprocess communication are:

- Shared memory. This permits processes to communicate by simply reading and writing to a specified memory location.
- Mapped memory. This is similar to shared memory, except that it is associated with a file in the file system.
- Pipes. Sequential communication from one process to a related process.
- FIFOs. These are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the file system.
- Sockets. These support communication between unrelated processes even on different computers.

These types of IPC differ by the following criteria:

- Whether they restrict communication to related processes (processes with a common ancestor), to unrelated processes sharing the same file system, or to any computer connected to a network
- Whether a communicating process is limited to only write data or only read data
- The number of processes permitted to communicate
- Whether the communicating processes are synchronized by the IPC for example, a reading process halts until data is available to read.

For this thesis prototype, sockets are used to communicate processes running in the target device and the desktop computer.

2.16 Sockets

A socket is a bi-directional communication device that can be used to communicate with another process on the same machine or with a process running on other machines.

When you create a socket, you must specify three parameters: communication style, namespace, and protocol.

A communication style controls how the socket treats transmitted data and specifies the number of communication partners. When data is sent through a socket, it is packaged into chunks called packets. The communication style determines how these packets are handled and how they are addressed from the sender to the receiver.

Connection styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender. The addresses of the sender and receiver are fixed at the beginning of the communication when the connection is established.

Datagram styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions. Each packet must be labeled with its destination and is not guaranteed to be delivered. The system guarantees only “best effort,” so packets may disappear or arrive in a different

order than shipping. A datagram-style socket behaves more like postal mail. The sender specifies the receiver's address for each individual message.

A socket namespace specifies how socket addresses are written. A socket address identifies one end of a socket connection. For example, socket addresses in the "local namespace" are ordinary filenames. In "Internet namespace," a socket address is composed of the Internet address (also known as an Internet Protocol address or IP address) of a host attached to the network and a port number. The port number distinguishes among multiple sockets on the same host.

A protocol specifies how data is transmitted. Some protocols are TCP/IP, the primary networking protocols used by the Internet. [10]

2.16.1 Linux sockets API

These are the system calls functions involving sockets:

- **`socket`**, Creates a socket
- **`close`**, Destroys a socket
- **`connect`**, Creates a connection between two sockets
- **`bind`**, Labels a server socket with an address
- **`listen`**, Configures a socket to accept conditions
- **`accept`**, Accepts a connection and creates a new socket for the connection

Sockets are represented by file descriptors.

2.16.2 Creating and destroying sockets

The **`socket`** and **`close`** functions create and destroy sockets, respectively. When a socket is created, there are three parameters: namespace, communication style, and protocol.

- First parameter: **`PF_LOCAL`** or **`PF_UNIX`** specifies the local namespace, and **`PF_INET`** specifies Internet namespaces.
- Second parameter: **`SOCK_STREAM`** for a connection-style socket, and **`SOCK_DGRAM`** for a datagram-style socket.
- Third parameter: the protocol, specifies the low-level mechanism to transmit and receive data. Each protocol is valid for a particular namespace-style combination.

Because there is usually one best protocol for each such pair, 0 is usually the correct protocol. If socket succeeds, it returns a file descriptor for the socket. It can be read from or written to the socket using **`read`**, **`write`**, and so on, as with other file descriptors. When the socket is not needed anymore, it can be called **`close`** to remove it. The following code shows an example:

```
int client_socket;
client_socket = socket(AF_INET , SOCK_STREAM , 0);
close(client_socket);
```

2.16.3 Connecting

To create a connection between two sockets, the client calls **`connect`**, specifying the address of a server socket to connect to. A client is the process initiating the connection, and a server is the process waiting to

accept connections. The client calls connect to initiate a connection from a local socket to the server socket specified by the second argument. The third argument is the length, in bytes, of the address structure pointed to by the second argument. Socket address formats differ according to the socket namespace. [10]

```
char * host_address = "192.168.1.150"; // Server IP address
uint16_t port = 23;
struct sockaddr_in server_address;
memset(&server_address, 0, sizeof(server_address));

client_socket = socket(AF_INET , SOCK_STREAM , 0);

if (client_socket != -1)
{
    server_address.sin_addr.s_addr = inet_addr(host_address);
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(port);
    result = connect(client_socket,
                (const struct sockaddr *) &server_address,
                 sizeof(server_address));
}
```

2.16.4 Sending information

Any technique to write to a file descriptor can be used to write to a socket. The send function, which is specific to the socket file descriptors, provides an alternative to write with a few additional choices (flags); for example, request the link layer that forward progress happened, avoid usage of gateway to send out the packets, nonblocking operation, etc. [11]

```
void * buffer = (void *)"Hola"; // char string
size_t length = 5;
send(client_socket, buffer, length, 0);
```

2.16.5 Server sockets

A server's life cycle consists of the creation of a connection-style socket, binding an address to its socket, placing a call to listen that enables connections to the socket, placing calls to accept incoming connections, and then closing the socket. Data isn't read and written directly via the server socket; instead, each time a program accepts a new connection, Linux creates a separate socket to use in transferring data over that connection. Important functions for this are: `bind`, `listen`, and `accept`.

An address must be bound to the server's socket using `bind` if a client is to find it. Its first argument is the socket file descriptor. The second argument is a pointer to a socket address structure; the format of this depends on the socket's address family. The third argument is the length of the address structure, in bytes. The following function reveals how the `TcpServer` class performs the binding of the server socket.

```
int TcpServer::prepare(void)
{
    int result = EXIT_FAILURE;
    if (server_socket == -1)
    {
        server_socket = socket(AF_INET , SOCK_STREAM , 0);
    }
    if (server_socket != -1)
    {
        if (!bound)
        {
            server_address.sin_family = AF_INET;
            server_address.sin_addr.s_addr = INADDR_ANY;
            server_address.sin_port = htons(host_port);
            result = bind(server_socket,
                    (struct sockaddr *)&server_address,
                    sizeof(server_address));
            bound = result != -1;
        }
        else
```

```
        {
            result = EXIT_SUCCESS;
        }
    }
    return result;
}
```

When an address is bound to a connection-style socket, it must invoke listen to indicate that it is a server. Its first argument is the socket file descriptor. The second argument specifies how many pending connections are queued. If the queue is full, additional connections will be rejected. This does not limit the total number of connections that a server can handle; it limits just the number of clients attempting to connect that have not yet been accepted.

A server accepts a connection request from a client by invoking **accept**. The first argument is the socket file descriptor. The second argument points to a socket address structure, which is filled with the client socket's address. The third argument is the length, in bytes, of the socket address structure. The server can use the client address to determine whether it really wants to communicate with the client. The call to **accept** creates a new socket for communicating with the client and returns the corresponding file descriptor. The original server socket continues to accept new client connections. The following code shows how the server socket returns a client socket after accepting the connection.

```
TcpSocket * TcpServer::accept_connection(void)
{
    TcpSocket * client = NULL;
    if (server_socket != -1)
    {
        int                client_socket;
        struct sockaddr_in client_address;
        int                addr_len;

        listen(server_socket , 1);
        addr_len = sizeof(struct sockaddr_in);
        client_socket = accept(server_socket,
                    struct sockaddr *) &client_address,
                    socklen_t*)&addr_len);

        if (client_socket != -1)
        {
            client = new TcpSocket(client_socket, client_address);
        }
    }
    return client;
}
```

To read data from a socket without removing it from the input queue, it can be used **recv**. It takes the same arguments as **read**, plus an additional FLAGS argument. A flag of **MSG_PEEK** causes data to be read but not removed from the input queue. [10]

2.16.6 Local sockets

Sockets connecting processes on the same computer can use the local namespace represented by the synonyms **PF_LOCAL** and **PF_UNIX**. These are called local sockets or UNIX-domain sockets. Their socket addresses, specified by filenames, are used only when creating connections. The socket's name is specified in **struct sockaddr_un**. You must set the **sun_family** field to **AF_LOCAL**, indicating that this is a local namespace. The **sun_path** field specifies the filename to use and may be, at most, 108 bytes long. The actual length of **struct sockaddr_un** should be computed using the **SUN_LEN** macro. Any filename can be used, but the process must have directory write permissions, which permit adding files to the directory. To connect to a socket, a process must have read permission for the file. Even though different computers may share the same file system, only processes running on the same computer can communicate with local namespace sockets. The only permissible protocol for the local namespace is 0. [10]

2.16.7 Internet domain sockets

UNIX-domain sockets can be used only for communication between two processes on the same computer. Internet domain sockets, on the other hand, may be used to connect processes on different machines connected by a network. Sockets connecting processes through the Internet use the Internet namespace represented by `PF_INET`.The most common protocols are TCP/IP. The Internet Protocol (IP), a low-level protocol, moves packets through the Internet, splitting and rejoining the packets, if necessary. It guarantees only "best-effort" delivery, so packets may vanish or be reordered during transport. Every participating computer is specified using a unique IP number. The Transmission Control Protocol (TCP), layered on top of IP, provides reliable connection-ordered transport. It permits telephone-like connections to be established between computers and ensures that data is delivered reliably and in order.

Internet socket addresses contain two parts: a machine and a port number. This information is stored in a `struct sockaddr_in` variable. Set the `sin_family` field to `AF_INET` to indicate that this is an Internet namespace address. The `sin_addr` field stores the Internet address of the desired machine as a 32-bit integer IP number. A port number distinguishes a given machine's different sockets. Because different machines store multibyte values in different byte orders, use `htons` to convert the port number to network byte order. See the man page for `ip` for more information.

To convert human-readable hostnames, either numbers in standard dot notation (such as 10.0.0.1) or DNS names (such as www.codesourcery.com) into 32-bit IP numbers, it can be used `gethostbyname` .This returns a pointer to the `struct hostent`; the `h_addr` field contains the host's IP number. The previous code examples in fact are internet domain sockets, these were implemented for the thesis prototype, internet server running on the target device.

2.17 Linux distributions

Linux is actually just a kernel. The kernel source is freely distributed to compile and install on a machine and then obtain and install many other freely distributed software programs to make a complete Linux installation. These installations are usually referred to as Linux systems because they consist of much more than just the kernel. Most of the utilities come from the GNU Project of the Free Software Foundation.

In the case of embedded Linux for ARM architectures, Linaro provides distributions already tested and ready to use.

Linaro is an engineering organization that works on free and open-source software such as the Linux kernel, the GNU Compiler Collection (GCC), power management, graphics and multimedia interfaces for the ARM family of instruction sets and implementations thereof as well as for the Heterogeneous System Architecture. The founding of Linaro was announced at Computex in June 2010 by ARM, Freescale Semiconductor, IBM, Samsung, ST-Ericsson, and Texas Instruments in a joint press conference. It also provides engineering and investment in upstream open source projects, a monthly release of tools and software and support to silicon companies in up-streaming code to be used with their system-on-chips (SoC). At some point, Linaro joined the HSA Foundation.

Linaro focuses on the use of the ARM instruction set in its versions 7a (32-bit) and 8 (64-bit) including concrete implementations of these, such as SoC that contain Cortex-A5, Cortex-A7, Cortex-A8, Cortex-

A9, Cortex-A15, Cortex-A53 or Cortex-A57 processor(s). Linaro engineering works with upstream projects on a set of requirements that are determined by the Technical Steering Committee (TSC).

At the end of each month, there is a release of tools and software that acts as a staging tree for code that will be pushed upstream. Linaro works on software that is close to the silicon such as kernel, multimedia, power management, graphics and security. Linaro aims to provide stable, tested tools and code for multiple software distributions to use to reduce low-level fragmentation of embedded Linux software.

For the thesis prototype, it was used the latest versions of Ubuntu and Debian, always performing with excellent results.

2.18 Software architecture

The software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.

The object-oriented application framework is the software structure employed for this thesis. This application framework is specialized to produce smart controllers, its design and development is actually the main goal of this thesis work.

2.18.1 Object-oriented application framework

Object-oriented (OO) application frameworks are a promising technology for conceptualizing proven software designs and implementations in order to reduce the cost and improve the quality of software. A framework is a reusable, “semi-complete” application that can be specialized to produce custom applications.

2.18.1.1 Benefits

The primary benefits of OO application frameworks stem from the modularity, reusability, extensibility, and inversion of control they provide to developers, as described below:

- **Modularity**. Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes. This localization reduces the effort required to understand and maintain existing software.
- **Reusability**. The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer productivity, as well as enhance the quality, performance, reliability, and interoperability of software.
- **Extensibility**. A framework enhances extensibility by providing explicit hook methods that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviours of an application domain from the variations required by instantiations

of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features. [12]

- **Inversion of control**. The run-time architecture of a framework is characterized by an "inversion of control." This architecture enables canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive dispatching mechanism. When events occur, the framework's dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-specific processing on the events. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events.

Developers in certain domains have successfully applied OO application frameworks for years. Early object-oriented frameworks (such as MacApp) originated in the domain of graphical user interfaces (GUIs). The Microsoft Foundation Classes (MFC) is a contemporary GUI framework that has become the *de facto* industry standard for creating graphical applications on PC platforms. Although MFC has limitations (such as lack of portability to non-Microsoft Windows PC platforms), its wide-spread adoption demonstrates the productivity benefits of reusing common frameworks to develop graphical business applications.

Application developers in more complex domains (such as telecommunications, medical systems, and real-time avionics) have traditionally lacked standard "off-the-shelf" frameworks. As a result, developers in these domains largely build, validate, and maintain software systems from scratch. In an era of deregulation and stiff global competition, however, it has become prohibitively costly and time-consuming to develop applications entirely in-house from the ground up.

Frameworks eliminate many tedious, error-prone, and non-portable aspects of creating and managing distributed applications and reusable service components. Frameworks enable programmers to develop and deploy complex applications rapidly and robustly, rather than wrestling endlessly with low-level infrastructure concerns.

2.18.1.2 Scope and classification

Although the benefits and design principles underlying frameworks are largely independent of the domain to which they are applied, it useful to classify frameworks by their scope, as follows:

System infrastructure frameworks. These frameworks simplify the development of portable and efficient system infrastructure such as operating system and communication frameworks (Linux OS and its System call API), and frameworks for user interfaces and language processing tools. System infrastructure frameworks are primarily used internally within a software organization and are not sold to customers directly. [13] [14]

Middleware integration frameworks. These frameworks are commonly used to integrate distributed applications and components. Middleware integration frameworks are designed to enhance the ability of software developers to modularize, reuse, and extend their software infrastructure to work seamlessly in a distributed environment.

Enterprise application frameworks. These frameworks address broad application domains (such as telecommunications, avionics, manufacturing, and financial engineering) and are the cornerstone of

enterprise business activities. Relative to System infrastructure and Middleware integration frameworks, Enterprise frameworks are expensive to develop and/or purchase. However, Enterprise frameworks can provide a substantial return on investment since they support the development of end-user applications and products directly. In contrast, System infrastructure and Middleware integration frameworks focus largely on internal software development concerns. Although these frameworks are essential to rapidly create high-quality software, they typically don't generate substantial revenue for large enterprises. As a result, it's often more cost-effective to buy System infrastructure and Middleware integration frameworks rather than build them in-house. [15]

Regardless of their scope, frameworks are also classified by the techniques used to extend them, which range along a continuum from whitebox frameworks to blackbox frameworks.

Whitebox frameworks. They are rely heavily on OO language features like inheritance and dynamic binding to achieve extensibility. Existing functionality is reused and extended by inheriting from framework base classes and overriding pre-defined hook methods using patterns like Template Method. Blackbox frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by defining components that conform to a particular interface and integrating these components into the framework using patterns like Strategy and Functor.

Whitebox frameworks require application developers to have intimate knowledge of the frameworks' internal structure. Although whitebox frameworks are widely used, they tend to produce systems that are tightly coupled to the specific details of the framework's inheritance hierarchies.

Blackbox frameworks. In contrast to whitebox, blackbox is structured using object composition and delegation more than inheritance. As a result, blackbox frameworks are generally easier to use and extend than whitebox frameworks. However, blackbox frameworks are more difficult to develop since they require framework developers to define interfaces and hooks that anticipate a wider range of potential use-cases. [16]

2.18.1.3 Approach design

Frameworks are closely related to other approaches to reuse, including:

Patterns. Patterns represent recurring solutions to software development problems within a particular context. Patterns and frameworks both facilitate reuse by capturing successful software development strategies. The primary difference is that frameworks focus on reuse of concrete designs, algorithms, and implementations in a particular programming language. In contrast, patterns focus on reuse of abstract designs and software micro-architectures.

Frameworks can be viewed as a concrete reification of families of design patterns that are targeted for a particular application domain. Likewise, design patterns can be viewed as more abstract micro-architectural elements of frameworks that document and motivate the semantics of frameworks in an effective way. When patterns are used to structure and document frameworks, nearly every class in the framework plays a well-defined role and collaborates effectively with other classes in the framework.

Class libraries. Frameworks extend the benefits of OO class libraries in the following ways:

- **Frameworks define "semi-complete" applications that embody domain-specific object structures and functionality**. Components in a framework work together to provide a generic architectural skeleton for a family of related applications. Complete applications can be composed by inheriting from and/or instantiating framework components. In contrast, class libraries are less domain-specific and provide a smaller scope of reuse. For instance, class library components like classes for Strings, vectors, and arrays are relatively low-level and ubiquitous across many application domains.
- **Frameworks are active and exhibit "inversion of control" at run-time**. Class libraries are typically passive, e.g., they perform their processing by borrowing threads of control from self-directed application objects. In contrast, frameworks are active, e.g., they control the flow of control within an application via event dispatching patterns like Reactor and Observer. The "inversion of control" in the run-time architecture of a framework is often referred to as The Hollywood Principle, e.g., "Don't call us, we'll call you."

In practice, frameworks and class libraries are complementary technologies. For instance, frameworks typically utilize class libraries like the C++ Standard Template Library (STL) internally to simplify the development of the framework. Likewise, application-specific code invoked by framework event handlers can utilize class libraries to perform basic tasks such as string processing, file management, and numerical analysis.

Components. Components are self-contained instances of abstract data types (ADTs) that can be plugged together to form complete applications. In terms of OO design, a component is a blackbox that defines a cohesive set of operations, which can be reused based solely upon knowledge of the syntax and semantics of its interface. Compared with frameworks, components are less tightly coupled and can support binary-level reuse. For example, applications can reuse components without having to subclass from existing base classes.

The relationship between frameworks and components is highly synergistic, with neither subordinate to the other. Frameworks can be used to develop components, whereby the component interface provides a Facade for the internal class structure of the framework. Likewise, components can be used as pluggable strategies in blackbox frameworks. In general, frameworks are often used to simplify the development of infrastructure and middleware software, whereas components are often used to simplify the development of end-user application software. Naturally, components are also effective for developing infrastructure and middleware, as well.

2.18.1.4 Challenges

When used in conjunction with patterns, class libraries, and components, OO application frameworks can significantly increase software quality and reduce development effort. However, a number of challenges must be addressed in order to employ frameworks effectively. There are several challenges that must be recognized and resolved such as development effort, learning curve, integratability, maintainability, validation and defect removal, efficiency, and lack of standards, which are outlined below:

Development effort. While developing complex software is hard enough, developing high quality, extensible, and reusable frameworks for complex application domains is even harder. The skills required

to produce frameworks successfully often remain locked in the heads of expert developers. One of the goals of this theme issue is to demystify the software process and design principles associated with developing and using frameworks.

Learning curve. Learning to use an OO application framework effectively requires a considerable investment of effort. For instance, it often takes 6-12 months become highly productive with a GUI framework like Android or Qt, depending on the experience of developers. Typically, hands-on mentoring and training courses are required to teach application developers how to use the framework effectively. Unless the effort required to learn the framework can be amortized over many projects, this investment may not be cost-effective. Moreover, the suitability of a framework for a particular application may not be apparent until the learning curve has flattened.

Integratability. Application development will be increasingly based on the integration of multiple frameworks (e.g. GUIs, communication systems, databases, etc.) together with class libraries, legacy systems, and existing components. However, many earlier generation frameworks were designed for internal extension rather than for integration with other frameworks developed externally. Integration problems arise at several levels of abstraction, ranging from documentation issues to the concurrency/distribution architecture, to the event dispatching model.

Maintainability. Application requirements change frequently. Therefore, the requirements of frameworks often change, as well. As frameworks invariably evolve, the applications that use them must evolve with them.

Framework maintenance activities include modification and adaptation of the framework. Both modification and adaptation may occur on the functional level (i.e., certain framework functionality does not fully meet developers' requirements), as well as on the non-functional level (which includes more qualitative aspects such as portability or reusability).

Framework maintenance may take different forms, such as adding functionality, removing functionality, and generalization. A deep understanding of the framework components and their interrelationships is essential to perform this task successfully. In some cases, the application developers and/or the end-users must rely entirely on framework developers to maintain the framework.

Validation and defect removal. Although a well-designed, modular framework can localize the impact of software defects, validating and debugging applications built using frameworks can be tricky for the following reasons:

- **Generic components are harder to validate in the abstract**. A well-designed framework component typically abstracts away from application-specific details, which are provided via sub-classing, object composition, or template parameterization. While this improves the flexibility and extensibility of the framework, it greatly complicates module testing since the components cannot be validated in isolation from their specific instantiations. Moreover, it is usually hard to distinguish bugs in the framework from bugs in application code.
- **Inversion of control and lack of explicit control flow**. Applications written with frameworks can be hard to debug since the framework's “inverted” flow of control oscillates between the application-independent framework infrastructure and the application-specific method callbacks.

This increases the difficulty of "single-stepping" through the run-time behaviour of a framework within a debugger since the control flow of the application is driven implicitly by callbacks and developers may not understand or have access to the framework code. This is similar to the problems encountered trying to debug a compiler lexical analyser and parser. In these applications, debugging is straightforward when the thread of control is in the user-defined action routines.

Efficiency. Frameworks enhance extensibility by employing additional levels of indirection. For instance, dynamic binding is commonly used to allow developers to subclass and customize existing interfaces. However, the resulting generality and flexibility often reduce efficiency. For instance, in languages like C++ and Java, the use of dynamic binding makes it impractical to support Concrete Data Types (CDTs), which are often required for time-critical software. The lack of CDTs yields an increase in storage layout (e.g., due to embedded pointers to virtual tables), performance degradation (e.g. due to the additional overhead of invoking a dynamically bound method and the inability to inline small methods), and a lack of flexibility (e.g., due to the inability to place objects in shared memory). In these critical real-time applications the efficiency is improved by static compilation.

Lack of standards. Currently, there are no widely accepted standards for designing, implementing, documenting, and adapting frameworks. Often, vendors use industry standards to sell proprietary software under the guise of open systems. Therefore, it's essential for companies and developers to work with standards organizations and middleware vendors to ensure the emerging specifications support true interoperability and define features that meet their software needs.

2.18.1.5 Trends

Trends in framework-related topics:

Reducing framework development effort. Traditionally, reusable frameworks have been developed by generalizing from existing systems and applications. Unfortunately, this incremental process of organic development is often slow and unpredictable since core framework design principles and patterns must be discovered "bottom-up." However, since many good framework exemplars now exist, we expect that the next generation of developers will leverage this collective knowledge to conceive, design, and implement higher quality frameworks more rapidly.

Greater focus on domain-specific enterprise frameworks. Existing frameworks have focused largely on system infrastructure and middleware integration domains. In contrast, there are relatively few widely documented exemplars of enterprise frameworks for key business domains such as manufacturing, banking, insurance, and medical systems. As more experience is gained developing frameworks for these business domains, however, we expect that the collective knowledge of frameworks will be expanded to cover an increasingly wide range of domain-specific topics and an increasing number of enterprise application frameworks will be produced. As a result, benefits of frameworks will become more immediate to application programmers, as well as to infrastructure developers. [17]

Blackbox frameworks. Many framework experts favour black-box frameworks over white-box frameworks since black-box frameworks emphasize dynamic object relationships (via patterns like Bridge and Strategy) rather than static class relationships. Thus, it is easier to extend and reconfigure black-box frameworks dynamically. As developers become more familiar with techniques and patterns for factoring

out common interfaces and components, it is expected that an increasing percentage of black-box frameworks will be produced.

Framework documentation. Accurate and comprehensible documentation is crucial to the success of large-scale frameworks. However, documenting frameworks is a costly activity and contemporary tools often focus on low-level method-oriented documentation, which fails to capture the strategic roles and collaborations among framework components. It is expected that the advent of tools for reverse-engineering the structure of classes and objects in complex frameworks will help to improve the accuracy and utility of framework documentation. Likewise, it is expected to see an increase in the current trend of using design patterns to provide higher-level descriptions of frameworks. [18]

Processes for managing framework development. Frameworks are inherently abstract since they generalize from a solution to a particular application challenge to provide a family of solutions. This level of abstraction makes it difficult to engineer their quality and manage their production. Therefore, it is essential to capture and articulate development processes that can ensure the successful development and use of frameworks. It is known that extensive prototyping and phased introduction of framework technology into organizations is crucial to reducing risk and helping to ensure successful adoption.

Framework economics. The economics of developing framework includes activities such as the following [19]:

- **Determining effective framework cost metrics**. These measure the savings of reusing framework components vs. building applications from scratch.
- **Cost estimation**. This is the activity of accurately forecasting the cost of buying, building, or adapting a particular framework.
- **Investment analysis and justification**. This determines the benefits of applying frameworks in terms of return on investment.

It is expected that the focus on framework economics will help to bridge the gap among the technical, managerial, and financial aspects of making, buying, or adapting frameworks.

2.19 Safety and reliability considerations

Embedded software may be qualitatively included in the Fault Tree Analysis (FTA) or Dependence Diagram (DD) for certain systems and items. In particular, FTA/DD may be necessary to provide adequate analytic visibility of software safety issues for complex systems, especially when credit is taken for the following safety attributes. [20]

- Systems and items which provide fail-safe protection against software errors (The protection may be provided either via other software or via hardware alone.)
- Systems and items in which software provides fail-safe protection against hardware errors or hardware faults
- Systems and items in which software provides protection against latent hardware faults.

Since the software implementation for this thesis prototype does not cause injuries to persons, nor significant material damage or other unacceptable consequences, this safety analysis is not presented in this academic work.

In order to ensure the system performs its required function under regular given conditions, it is suggested to implement a redundant element, for future development.

2.20 Development environment

In addition to understanding the necessary theoretical background, it is important to understand what development tools are available that aid in the implementation of the system. The development and integration of the system's various hardware and software components are made possible through development tools that provide everything from HDL synthesis, loading software into the hardware, test and debugging.

A development environment is a collection of tools for developing, testing and debugging software. The development environment is shown in the following table.

Tool	Version
Desktop RAM 7,7 GB; Disk 93,4 GB Intel® Core™ i5-6200U CPU @ 2.30GHz × 4, 64-bit	Ubuntu 16.04 LTS (64-bit)
Vivado Design Suite. Software suite produced by Xilinx for synthesis and analysis of HDL designs with features for system on a chip development and high-level synthesis.	Vivado v2015.4 (64-bit Linux)
Xilinx Software Development Kit (XSDK). Integrated Design Environment for creating embedded applications on Xilinx microprocessors.	Release Version 2015.4. Linux
Qt Creator Cross-platform C++, JavaScript and QML integrated development environment which is part of the SDK for the Qt GUI Application development framework.	Based on Qt 5.6.1 (GCC 5.4.0 20160609, 64 bit Linux)
GNU toolchain Provided by Xilinx; it includes C/C++ compilers, optimized C/C++ libraries and GDB debugger	Release Version 2015.4. Linux
TCF Target Communications Framework, TCP/IP debugger.	TCF 1.5
GIT Version control and collaboration tool	git version 2.7.4, and web based interface

Table 3: Development environment

2.20.1 Debugging

Debugging is primarily the task of locating and fixing errors within the system. This task is made simpler having appropriate debugging tools. The SDK debugger allows seeing what is happening to a program

while it executes. You can set breakpoints or watchpoints to stop the processor, step through program execution, view the program variables and stack, and view the contents of the memory in the system [21].

For this thesis prototype, TCF (Target Communications Framework) is the preferred debugging tool for development on the Linux userspace. For kernel debugging, it was employed kernel logging.

2.20.2 TCF debugger

TCF (Target Communications Framework) is a universal, extensible, simple, lightweight, vendor agnostic framework for tools and targets to communicate for purpose of debugging, profiling, code patching and performing other device software development tasks. It requires just a single configuration per target (not per tool per target as today in most cases), or no configuration at all when possible. It is designed to have small overhead and footprint on the target side, operates over the transport-agnostic channel abstraction and supports auto-discovery of targets and services.

The channel implementation is TCP/IP, and the autodiscovery implementation is based on UDP broadcasting.

TCF is an open source tool that has to be recompiled on target or cross-compiled for the desired target architecture, all TCF code is licensed under Eclipse Public License (EPL). C agent code is also licensed under Eclipse Distribution License (EDL) (dual-licensing).

2.21 Version control

Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

2.21.1 Benefits

Developing software without using version control is risky, like not having backups. Version control can also enable developers to move faster and it allows software teams to preserve efficiency and agility as the team scales to include more developers.

1. **A complete long-term change history of every file**. Changes include the creation and deletion of files as well as edits to their contents over the years.
2. **Branching and merging.** Team members work concurrently easily, even individuals working on their own can benefit from the ability to work on independent streams of changes, this means their own sandbox.
3. **Traceability**. Being able to trace each change made to the software and connect it to project management and bug tracking software.

Git is the version control and collaboration tool utilized during the development of this project.

2.21.2 GitHub

GitHub is a code hosting platform for version control and collaboration. It allows developers to work together on projects from anywhere.

Repositories and historical evidence.

Repository	Repository web URL
Object-oriented application framework. OO App-framework, controller app, and system tool app for target. Source code: C++, C.	https://github.com/YaribNevarez/app-workspace.git
Linux kernel modules ADC, PWM, Controller kernel modules. Source code: C.	https://github.com/YaribNevarez/Device-drivers-Xilinx-Linux-platform.git
Remote Test Tool Qt GUI remote test tool. Source code: C++	https://github.com/YaribNevarez/TestTool.git
Hardware design. Vivado system on chip design. Source code: VHDL, C.	https://github.com/YaribNevarez/Hardware-accelerator-Linux-platform.git

Table 4: Version control repositories

2.22 Software verification and validation

Software verification is a practice of verifying documents, design, code and the program. It includes all the activities associated with producing high-quality software: inspection, design analysis, and specification analysis. It is a relatively objective process. On the other hand, validation is the process of evaluating the final product to check whether the software meets the customer expectations and requirements. It is a dynamic mechanism for validating and testing the actual product.

A primary purpose of validating and testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product functions properly under all conditions but can establish only that it does not function properly under specific conditions. The scope of software testing often includes the examination of code as well as the execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team.

2.22.1 Development of software tools

The object-oriented application framework (developed for this thesis), is actually used for fast development of software testing tools. These software tools are intended to be capable of testing all the features and use cases of the prototype itself. These software tools can be extended to perform automated test.

Tool	Feature description

Remote access tool (TCP/IP) Desktop executable (x86-64 GNU/Linux, C++)	This application tests remotely all devices of the controller.
System Tool Target executable (ARM GNU/Linux, C++)	This tool tests all features locally on the embedded platform.

Table 5: Development of software tools

Details about the development of these software tools will be described in following chapters.

There is another on-going thesis work focusing on quality assurance and testing for this prototype.

Chapter 3. **Hardware**

The smart controller is a hardware and software implementation, the digital hardware is implemented in the programmable logic of a SoC, and it is intended to give proper interface communication with the external devices; the software part is running on the processing system, which is intended to be the controlling unit.

The ZYBO board was used as the development platform.

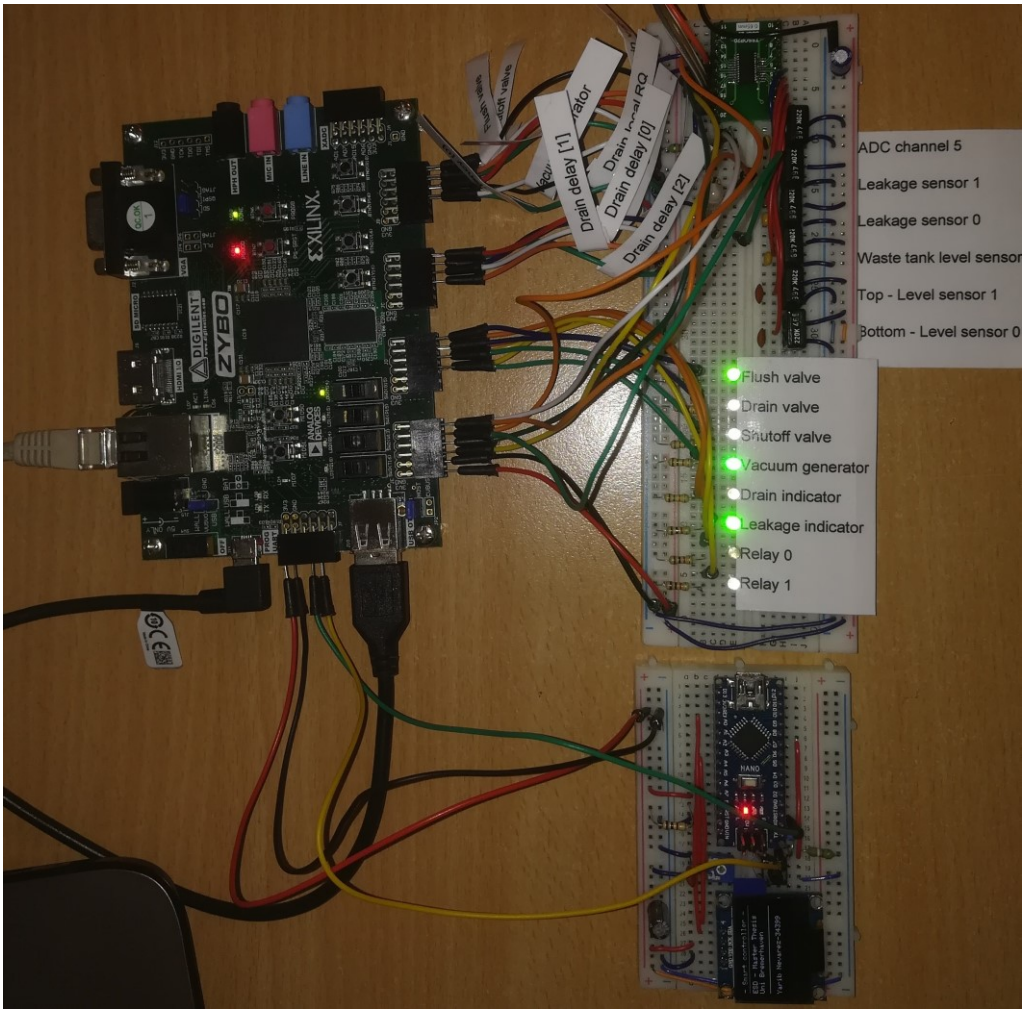

Figure 4: Development platform

3.1 Development platform (ZYBO)

The ZYBO (ZYnq BOard) is a feature-rich, ready-to-use, entry-level embedded software and digital circuit development platform built around the smallest member of the Xilinx Zynq-7000 family, the Z-7010. The Z-7010 is based on the Xilinx All Programmable System-on-Chip (AP SoC) architecture, which tightly integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic. When coupled with the rich set of multimedia and connectivity peripherals available on the ZYBO, the Zynq Z-7010 can host a whole system design. The on-board memories, video and audio I/O, dual-role USB, Ethernet, and SD slot will have your design up-and-ready with no additional hardware needed. Additionally, six Pmod ports are available to put any design on an easy growth path. [22]

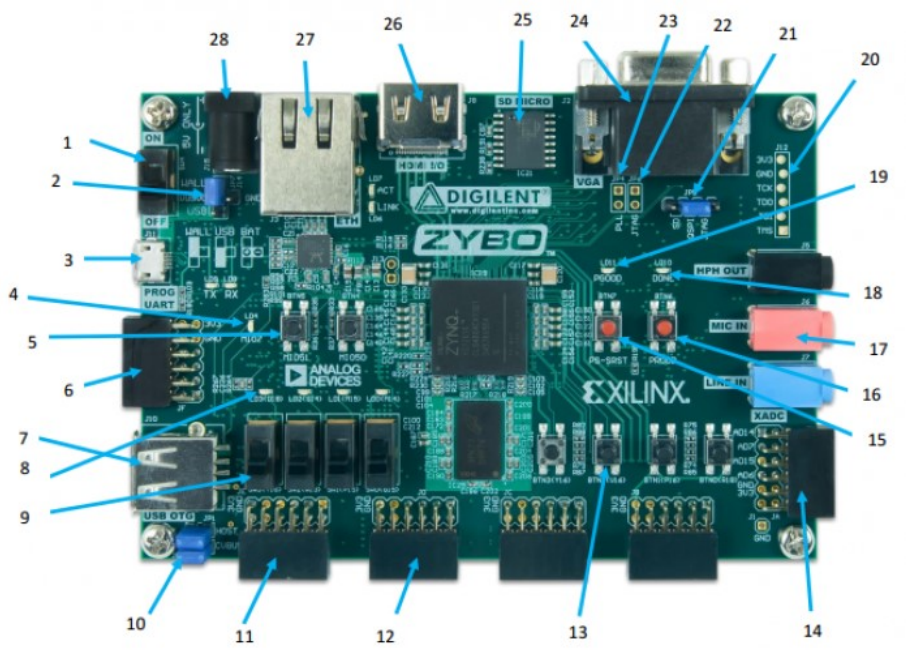

Figure 5: ZYBO platform

Callout	Component description	Callout	Component description
1	Power switch	15	Processor reset pushbutton
2	Power select jumper and battery header	16	Logic configuration reset pushbutton
3	Shared UART/JTAG USB port	17	Audio codec connectors
4	MIO LED	18	Logic configuration done LED
5	MIO Pushbuttons (2)	19	Board power good LED
6	MIO PMOD	20	JTAG port for optional external cable
7	USB OTG connectors	21	Programming mode jumper
8	Logic LEDs (4)	22	Independent JTAG mode enable jumper
9	Logic slide switches (4)	23	PLL bypass Jumper
10	USB OTG host/device select jumpers	24	VGA connector
11	Standard PMOD	25	Micro SD connector (reverse side)
12	High-speed PMOD (3)	26	HDMI sink/source connector
13	Logic pushbuttons (4)	27	Ethernet RJ45 connector
14	XADC PMOD	28	Power jack

Table 6: ZYBO platfrom components

"The cabin convenient controller unit" will be ported to a customized PCB hardware platform, which is also built around the Xilinx Zynq-7000 family.

3.2 Hardware block design

The following diagram shows the overview of the required system design.

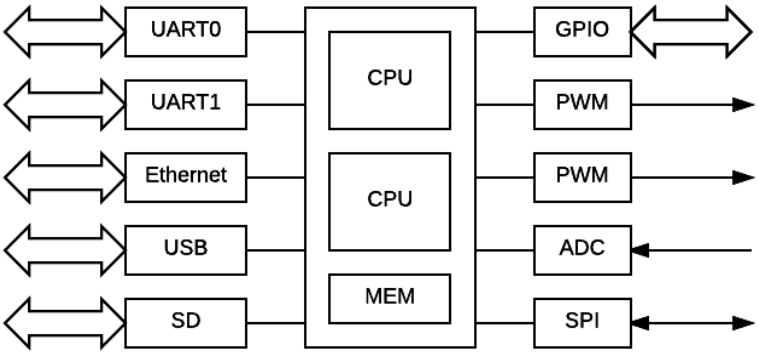

Figure 6: System block diagram

The customization of the processing system permits the configuration of the UART, Ethernet, USB, and QSPI (SD card).

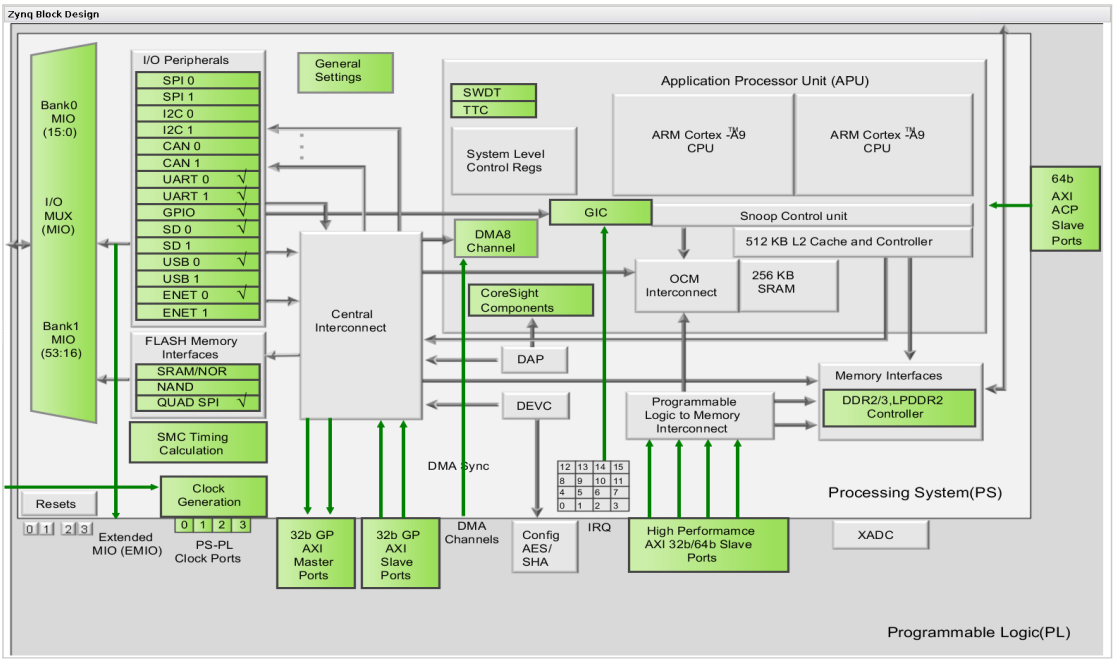

Figure 7: Zynq processing system

The peripherals such as GPIOs, PWMs and SPI are implemented in the programmable logic. The ADC is an external device that is connected through SPI.

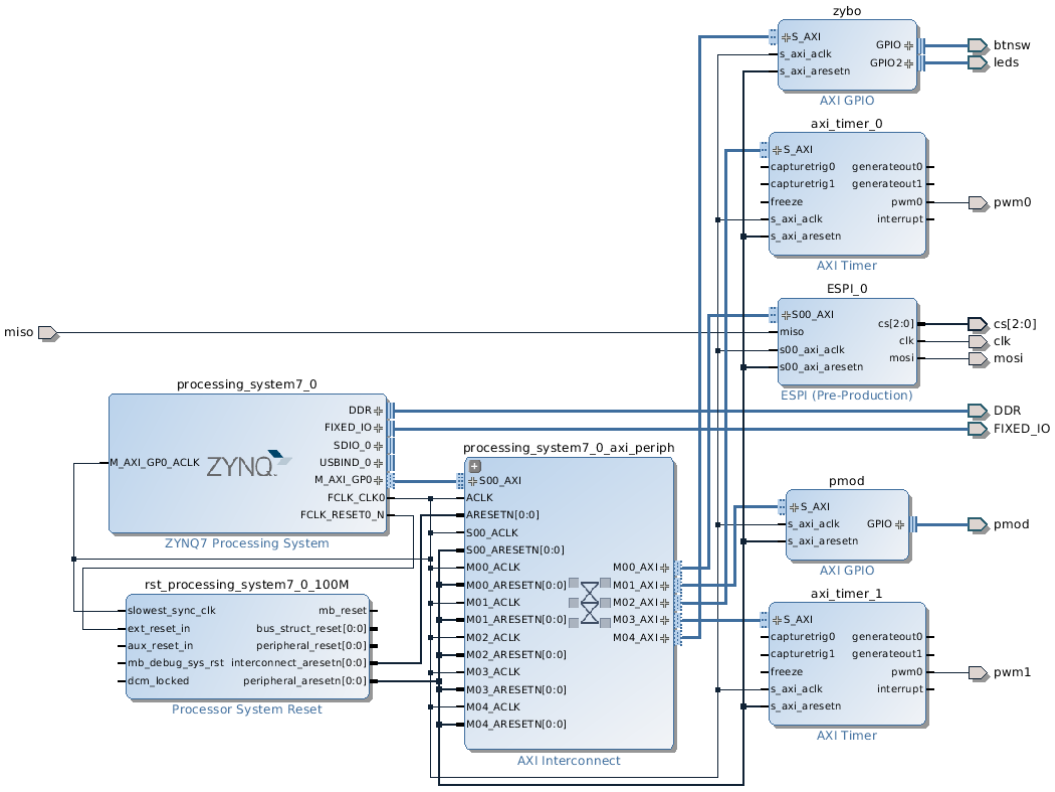

Figure 8: Hardware block design

The following table shows the peripheral addresses generated by the SoC synthesis.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
pmod	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
ESPI_0	S00_AXI	S00_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF
axi_timer_0	S_AXI	Reg	0x4280_0000	64K	0x4280_FFFF
axi_timer_1	S_AXI	Reg	0x4281_0000	64K	0x4281_FFFF
zybo	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF

Table 7: Hardware AXI peripheral address

These peripherals are being accessed by the kernel modules through their corresponding address.

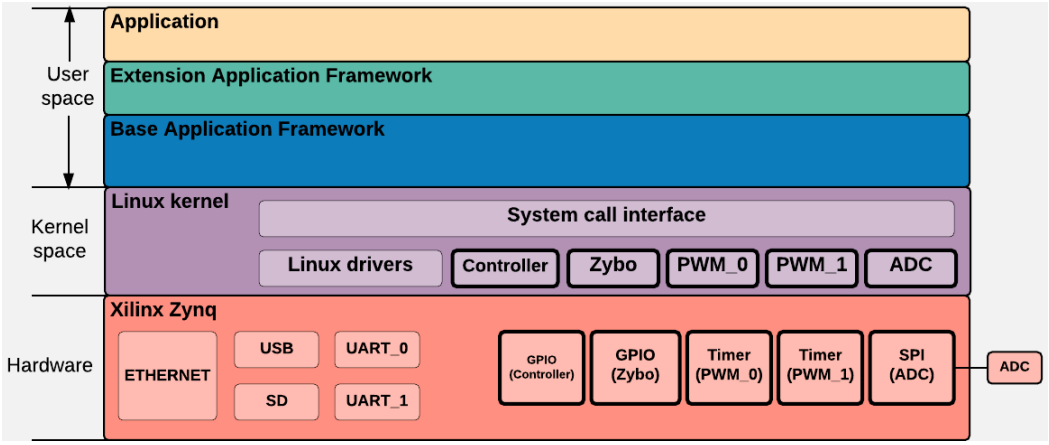

Figure 9: System structure

The hardware IP blocks and their corresponding kernel modules are described in the following paragraphs.

3.3 Enhanced SPI

Inside the ESPI block, it is instantiated an Enhanced SPI which was designed with the following features.

- Baud rate divider for output clock signal (SCLK)
- Configurable data length (8, 16, 24 and 32 bits)
- Flexible settle-time for specific devices
- Configurable clock polarity (CPOL) and clock phase (CPHA).
- Full duplex data transmission.

The VHDL code implemented for the Enhanced SPI is listed below.

```
ENTITY espi IS
    GENERIC (DATA_LENGTH_BIT_SIZE   : INTEGER := 2;
             SETTLE_TIME_SIZE       : INTEGER := 2;
             DATA_SIZE              : INTEGER := 32;
             BAUD_RATE_DIVIDER_SIZE : INTEGER := 8);
    PORT ( clk                : IN STD_LOGIC;
           reset              : IN STD_LOGIC;
           data_length        : IN STD_LOGIC_VECTOR (DATA_LENGTH_BIT_SIZE-1 DOWNTO 0);
           baud_rate_divider  : IN STD_LOGIC_VECTOR (BAUD_RATE_DIVIDER_SIZE-1 DOWNTO 0);
           settle_time        : IN STD_LOGIC_VECTOR (SETTLE_TIME_SIZE-1 DOWNTO 0);
           clock_polarity     : IN STD_LOGIC;
           clock_phase        : IN STD_LOGIC;
           start_transmission : IN STD_LOGIC;
           transmission_done  : OUT STD_LOGIC;
           data_tx  : IN  STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0);
           data_rx  : OUT STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0) := (others => '0');
           spi_clk  : OUT STD_LOGIC;
           spi_MOSI : OUT STD_LOGIC;
           spi_MISO : IN STD_LOGIC;
           spi_cs   : OUT STD_LOGIC);
END espi;
```

The baud rate divider is implemented in the following process.

```
    baud_rate_division_process: PROCESS (clk, reset, baud_rate_divider, settle_time,
clk_pulse)
    VARIABLE baud_rate_counter   : UNSIGNED (BAUD_RATE_DIVIDER_SIZE-1 DOWNTO 0) :=
(others => '0');
    VARIABLE settle_time_counter : UNSIGNED (SETTLE_TIME_SIZE-1 DOWNTO 0)       :=
(others => '0');
    BEGIN
        IF falling_edge(clk) THEN
            clk_pulse <= '0';
            IF reset = '1' OR current_state = SPI_IDLE THEN
                baud_rate_counter := (others => '0');
                settle_time_counter := (others => '0');
            ELSIF baud_rate_divider = CONV_STD_LOGIC_VECTOR(baud_rate_counter,
BAUD_RATE_DIVIDER_SIZE) THEN
                baud_rate_counter := (others => '0');
                ----------------------------------------------------------------
                IF current_state = SPI_REDY OR current_state = SPI_STOP THEN
                    IF settle_time = CONV_STD_LOGIC_VECTOR(settle_time_counter,
SETTLE_TIME_SIZE) THEN
                        clk_pulse <= '1';
                        settle_time_counter := (others => '0');
                    ELSE
                        settle_time_counter := settle_time_counter + 1;
                    END IF;
                ELSE
                    clk_pulse <= '1';
                END IF;
                ----------------------------------------------------------------
            ELSE
                baud_rate_counter := baud_rate_counter + 1;
            END IF;
        END IF;
    END PROCESS;
```

The mechanism for switching between states (SPI as a state machine) is implemented in the following process. This process determines the number of bits that should be transmitted-received based on the selected data length, the shifted data in the internal buffers, and the setup for the next state.

```
    spi_switch_state : PROCESS (clk, current_state, next_state)
    VARIABLE data_length_internal : UNSIGNED (1 DOWNTO 0);
    BEGIN
        IF rising_edge(clk) THEN
            IF reset = '1' THEN
                current_state <= SPI_IDLE;
                i_tx_buffer <= (others => '0');
                i_rx_buffer <= (others => '0');
                data_rx <= (others => '0');
            ELSIF next_state = SPI_REDY THEN -- Get ready immediately
                current_state <= SPI_REDY;
                data_rx <= (others => '0');
                i_rx_buffer <= (others => '0');

                CASE data_length IS
                    WHEN "00" =>
                     i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_0) <=
data_tx(DATA_LENGTH_0-1 downto 0);
                     counter <= DATA_LENGTH_0-1;
                    WHEN "01" =>
                     i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_1) <=
data_tx(DATA_LENGTH_1-1 downto 0);
                     counter <= DATA_LENGTH_1-1;
                    WHEN "10" =>
                     i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_2) <=
data_tx(DATA_LENGTH_2-1 downto 0);
                     counter <= DATA_LENGTH_2-1;
                    WHEN "11" =>
                     i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_3) <=
data_tx(DATA_LENGTH_3-1 downto 0);
                     counter <= DATA_LENGTH_3-1;
                    WHEN OTHERS => NULL;
                END CASE;
            ELSIF clk_pulse = '1' THEN       -- Or Wait for the pulse

                IF clock_phase = '0' THEN
                    -- PUSH INPUT
                    IF next_state = SPI_CLK_ACTIVE THEN
                        i_rx_buffer <= i_rx_buffer(DATA_SIZE-2 DOWNTO 0) & spi_MISO;
                    END IF;
                    -- POP OUTPUT
                    IF next_state = SPI_CLK_IDLE THEN
                        i_tx_buffer <= i_tx_buffer(DATA_SIZE-2 downto 0) & '-';
                        counter <= counter - 1;
                    END IF;
                END IF;

                IF clock_phase = '1' THEN
                    -- PUSH INPUT
                    IF current_state = SPI_CLK_ACTIVE THEN
                        i_rx_buffer <= i_rx_buffer(DATA_SIZE-2 DOWNTO 0) & spi_MISO;
                    END IF;
                    -- POP OUTPUT
                    IF current_state = SPI_CLK_IDLE AND next_state = SPI_CLK_ACTIVE THEN
                        i_tx_buffer <= i_tx_buffer(DATA_SIZE-2 downto 0) & '-';
                        counter <= counter - 1;
                    END IF;
                END IF;

                IF current_state = SPI_STOP THEN
                    data_rx <= i_rx_buffer;
                END IF;

                current_state <= next_state;
            END IF;
        END IF;
    END PROCESS;
```

The SPI state-machine is implemented in the following process.

```
    spi_mechanism :  process (next_state, clock_polarity, current_state,
start_transmission, i_tx_buffer, clock_phase, counter)
    BEGIN
        next_state <= current_state;
        spi_clk <= clock_polarity;
        spi_cs <= '1';
        spi_MOSI <= '0';
        transmission_done <= '1';

        CASE current_state IS
            WHEN SPI_IDLE =>
                IF start_transmission = '1' THEN
                    next_state <= SPI_REDY;
                END IF;
            WHEN SPI_REDY =>
                spi_cs <= '0';
                transmission_done <= '0';
                IF clock_phase = '0' THEN
                    spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                END IF;
                next_state <= SPI_CLK_ACTIVE;
            WHEN SPI_CLK_ACTIVE =>
                spi_cs <= '0';
                transmission_done <= '0';
                spi_clk <= NOT clock_polarity;
                spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                IF counter = 0 THEN
                    next_state <= SPI_STOP;
                ELSE
                    next_state <= SPI_CLK_IDLE;
                END IF;
            WHEN SPI_CLK_IDLE =>
                spi_cs <= '0';
                transmission_done <= '0';
                spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                IF counter = 0 AND clock_phase = '1' THEN
                    next_state <= SPI_STOP;
                ELSE
                    next_state <= SPI_CLK_ACTIVE;
                END IF;
            WHEN SPI_STOP =>
                IF clock_phase = '1' THEN
                    spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                END IF;
                next_state <= SPI_IDLE;
                transmission_done <= '0';
                spi_cs <= '0';
        END CASE;
    END PROCESS;
```

The internal mechanism is summarized in the following state diagram.

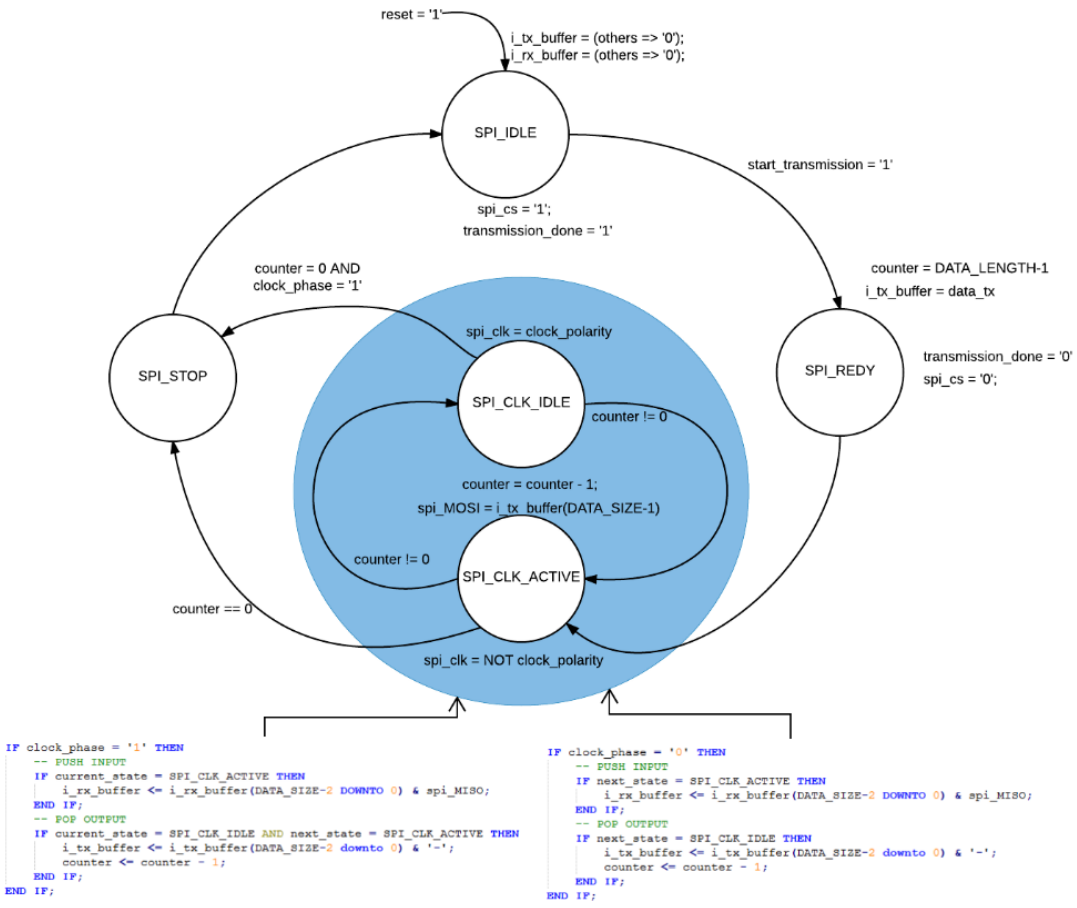

Figure 10: Enhanced SPI – State machine diagram

For the simulation it was instantiated an Enhanced SPI, it was connected `MOSI` and `MISO` lines so the given value in the `data_tx` register appears in `data_rx` register after the transmission (full duplex).

- `data_length` set to two so 24 bits will be transmitted
- `settle_time` set to three to start the transmission after four clock cycles (40ns)
- `baud_rate_divider` set to zero for no clock division (full speed)
- `clock_polarity` set to one making high the idle state of the output clock signal
- `clock_phase` set to zero in order to start the transmission with no phase delay

The data to be transmitted is set to the `data_tx` register (32 bits), in this simulation it set to `a51188a5`, since the data length was setup for having a transmission of 24 bits (3 bytes) and `MOSI` and `MISO` lines are connected, the received data in `data_rx` is `001188a5` (24 bits) after 24 clock cycles.

For more detailed information regarding the SPI protocol, it can be referred to the documentation of standardized SPI protocol.

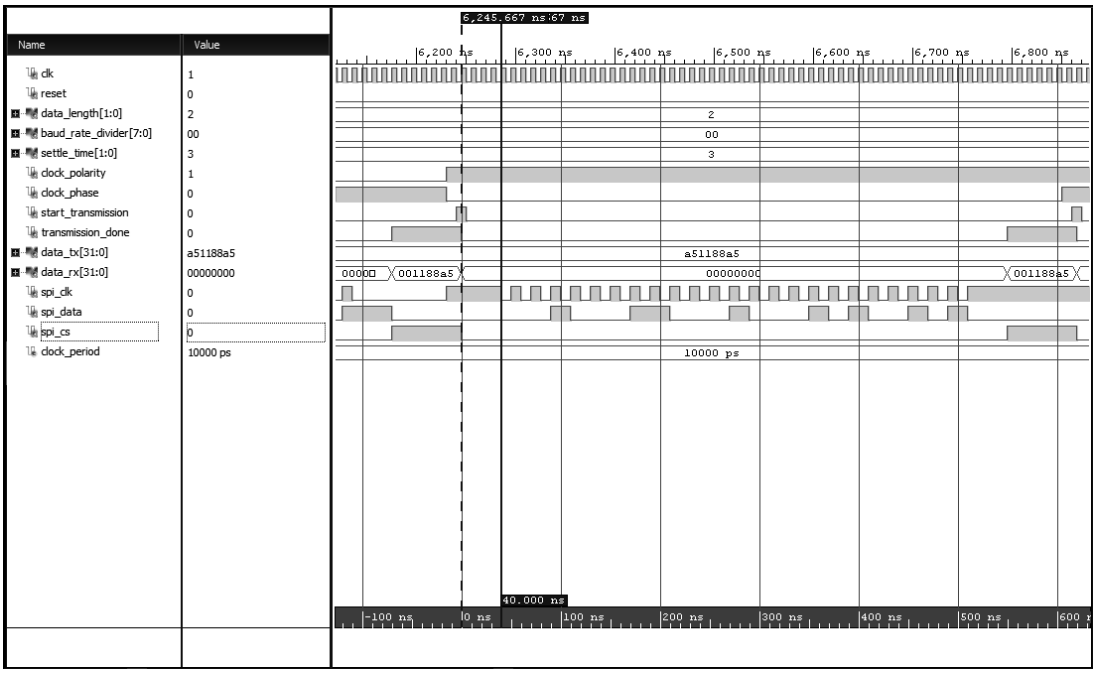

Figure 11: Enhanced SPI - Simulation

3.4 ADC

The MAX11632 is the analog to digital converter (ADC) utilized for this prototype, it is an external SPI device.

The following figure shows the connections between the Zynq device and the MAX11632.

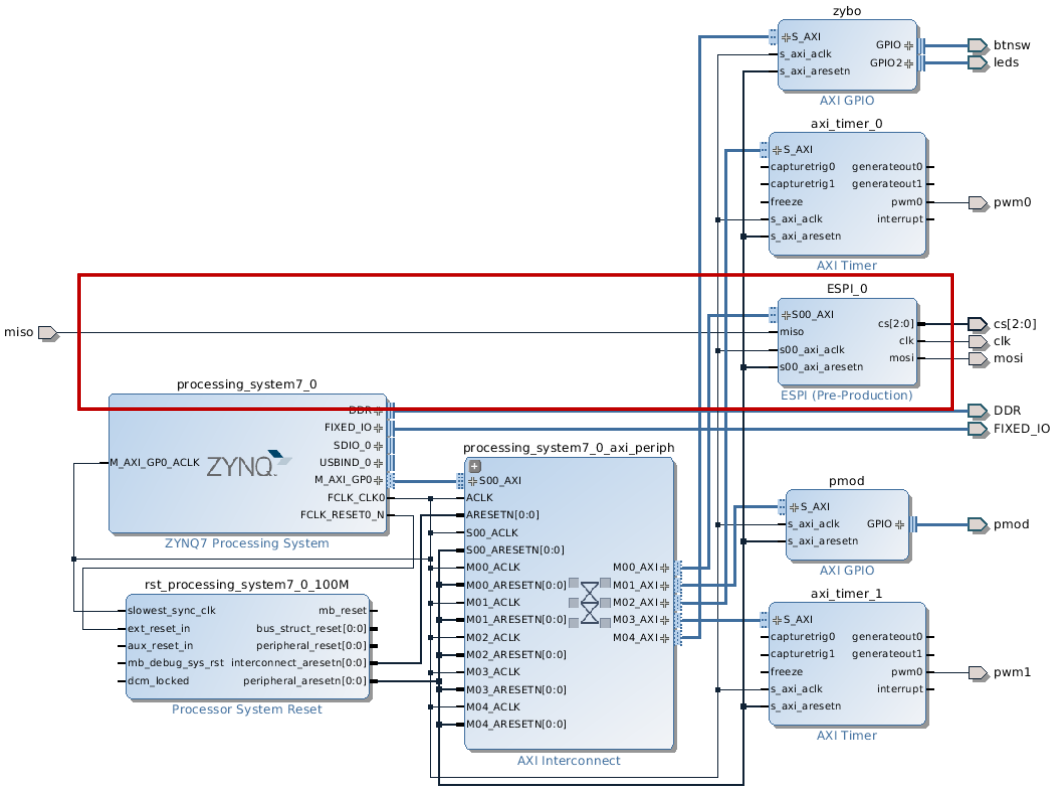

Figure 12: SPI hardware block

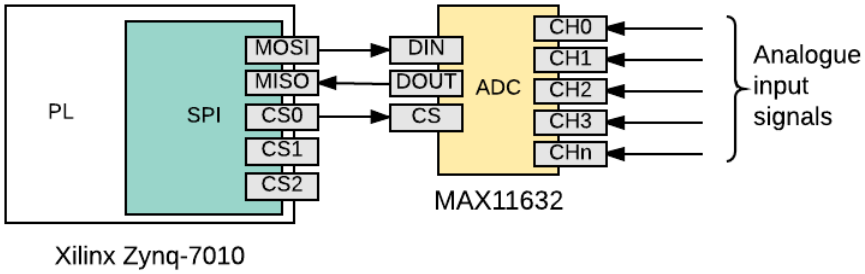

Figure 13: ADC interface

The MAX11632 are serial 12-bit analog-to-digital converters (ADCs) with an internal reference. These devices feature on-chip FIFO, scan mode, internal clock mode, internal averaging, and AutoShutdown™. The maximum sampling rate is 300ksps using an external clock. The MAX11632 has 16 input channels. This device operates from either a +3V supply or a +5V supply, and contain a 10MHz SPI-/QSPI-/MICROWIRE-compatible serial port.

Converter operation. The MAX11632 ADCs use a successive-approximation register (SAR) conversion technique and an on-chip T/H block to convert voltage signals into a 12-bit digital result. This single-ended configuration supports unipolar signal ranges.

Input bandwidth. The ADC's input-tracking circuitry has a 1MHz small-signal bandwidth, so it is possible to digitize high-speed transient events and measure periodic signals with bandwidths exceeding the ADC's sampling rate by using undersampling techniques. Anti-alias prefiltering of the input signals is necessary to avoid high-frequency signals aliasing into the frequency band of interest. [23]

3.4.1 Device driver

The Linux kernel communicates with the ADC device through the SPI, hence it is developed a kernel module to access the SPI registers implemented in the PL of the Zynq device.

The following table shows the internal ADC registers, this information is used for the operation of the ADC.

REGISTER NAME	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
Conversion	1	CHSEL3	CHSEL2	CHSEL1	CHSEL0	SCAN1	SCAN0	X
Setup	0	1	CKSEL1	CKSEL0	REFSEL1	REFSEL0	X	X
Averaging	0	0	1	AVGON	NAVG1	NAVG0	NSCAN1	NSCAN0
Reset	0	0	0	1	$\overline{\text{RESET}}$	X	X	X

X = Don't care.

Figure 14: ADC MAX11632 - Internal registers

The following code shows the implementation of the kernel

```
/* Device Probe function for driver
* ------------------------------------
*  Get the resource structure from the information in device tree.
*  request the memory region needed for the controller, and map it into
*  kernel virtual memory space. Create an entry under /proc file system
*  and register file operations for that entry.
*/
static int driver_probe(struct platform_device *pdev)
{
    struct proc_dir_entry * driver_proc_entry;
    int rc = SUCCESS;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
```

```
    if (!res)
    {
        dev_err(&pdev->dev, "No memory resource\n");
        rc = -ENODEV;
    }

    if (rc == SUCCESS)
    {
        remap_size = res->end - res->start + 1;
        if (!request_mem_region(res->start, remap_size, pdev->name))
        {
            dev_err(&pdev->dev, "Cannot request IO\n");
            rc = -ENXIO;
        }
    }

    if (rc == SUCCESS)
    {
        base_addr = ioremap(res->start, remap_size);
        if (base_addr == NULL)
        {
            dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n", (unsigned
long)res->start);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        driver_proc_entry = proc_create(DRIVER_NAME, 666, NULL,
&proc_driver_operations);
        if (driver_proc_entry == NULL)
        {
            dev_err(&pdev->dev, "Couldn't create proc entry\n");
            iounmap(base_addr);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        printk(KERN_INFO DRIVER_NAME " Mapped at virtual address 0x%08lx\n", (unsigned
long) base_addr);

        major = register_chrdev(0, DRIVER_NAME, &proc_driver_operations);

        spi_initialize();
    }

    return rc;
}
```

At the end of the kernel module initialization, it is initialized the ADC device in order to make it operational.

```
void spi_initialize(void)
{
    spi_set_baud_rate_divider(0x10); // SPI_CLK(f) = 50000000 / 0x10
    spi_set_clock_phase(0);
    spi_set_clock_polarity(0);

    spi_set_cs_force(0);
    spi_set_settle_time(0);
    spi_set_slave_select(0);

    spi_set_data_length(SPI_DATA_LENGTH_8_BITS);

    spi_send_data(RESET);
    spi_send_data(AVERAGING | NO_AVG);
    spi_send_data(SETUP | CLK_EXTERNAL | REF_INTERNAL_ON);

    spi_set_data_length(SPI_DATA_LENGTH_24_BITS);
}
```

The function to read an analogue channel is `u32 adc_read(u8 channel)`, it receives the channel number to read, and it returns the conversion.

```
// base_addr = 0x43C00000

static void outport(u8 offset, u32 data)
{
    wmb();
    iowrite32(data, base_addr + offset);
}

static u32 inport(u8 offset)
{
    wmb();
    return ioread32(base_addr + offset);
}

static u32 spi_send_data(u32 data)
{
    u32 reg;
    do
    {
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    outport(SPI_DATA_REGISTER_INDEX, data);
    return spi_receive_data();
}

u32 adc_read(u8 channel)
{
    channel <<= 3;
    return spi_send_data( ( CONVERSION | channel | NO_SCAN ) << 16 );
}
```

The read and write file operations.

```
static u8           ADC_chanel  = 0;
static DeviceID     device_ID   = 0;
...
/* Write operation for /proc/driver
* ----------------------------------
*/
static ssize_t proc_driver_write(struct file *file,
                                 const char __user * buffer,
                                 size_t buffer_size,
                                 loff_t * position)
{
    ssize_t written_size = 0;

    if ((buffer != NULL) && (buffer_size == sizeof(DeviceID)))
    {
        written_size = buffer_size - copy_from_user((void *) &device_ID, buffer,
sizeof(device_ID));

        ADC_chanel = device_ID;
    }
    return written_size;
}

/* Read operation for /proc/driver
* ----------------------------------
*/
static ssize_t proc_driver_read(struct file *file,
                                char __user * buffer,
                                size_t buffer_size,
                                loff_t * position)
{
    ssize_t read_size = 0;

    if ((buffer != NULL) && (sizeof(IOPacket) <= buffer_size))
    {
        IOPacket packet;
        packet.device_ID = device_ID;
        packet.data = adc_read(ADC_chanel);
        read_size = buffer_size - copy_to_user(buffer, (void *) &packet,
sizeof(IOPacket));
```

```
    }
    return read_size;
}
```

The entire implementation of this device driver can be found in the Appendix B – ADC .

3.4.2 Device tree node

The device node that is added to the device tree:

```
/dts-v1/;
/ {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "xlnx,zynq-7000";
    model = "Xilinx Zynq";
    ...
    ps7_axi_interconnect_0: amba@0 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
        ranges ;
        ...
        adc
        {
            compatible = "yarib-adc-1.00.a";
            reg = <0x43C00000 0x10000>;
        } ;
    } ;
} ;
```

The implementation of the device tree can be found in the Appendix B – Device tree.

3.5 GPIO

The AXI GPIO v2.0 (LogiCORE IP) is the GPIO block utilized for this prototype. For detailed information about this peripheral can be found in datasheet XILINX AXI GPIO v2.0 LogiCORE IP Product Guide. [24] The block design contains two GPIO blocks, one is intended to drive the IO signals on the PMOD connectors (JB, JC, JD) on ZYBO development board. The ZYBO GPIO drives the push buttons, switches, and LEDs on the same board.

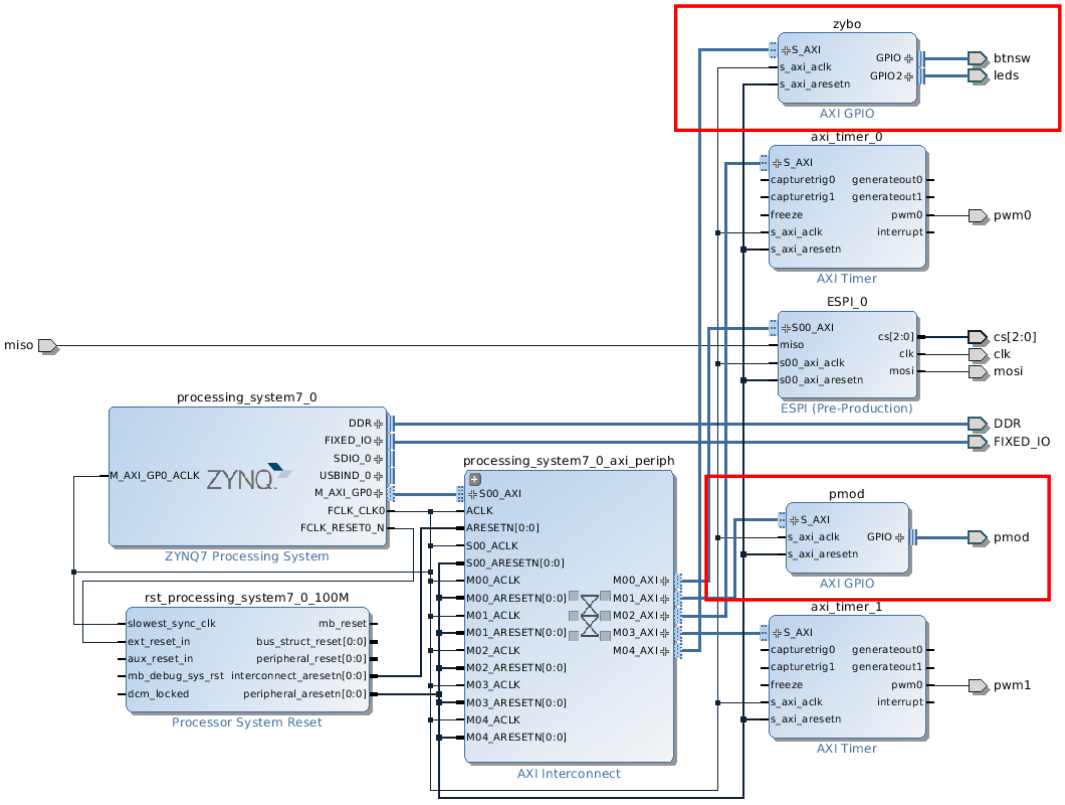

Figure 15: GPIO hardware blocks

The PMOD GPIO is single channel 24-bit port, each bit configurable as either input or output (configured in the kernel), this GPIO is used by the custom application for controlling generic digital IO signals, actuators and sensors.

The ZYBO GPIO is dual channel port. The first channel is configured as 8-bit input, lower nibble for push buttons and higher nibble for switches, the second channel is configured as 4-bit output for the LEDs.

3.5.1 Description

- Supports the AXI4-Lite interface specification
- Supports configurable single or dual GPIO channel(s)
- Supports configurable channel width for GPIO pins from 1 to 32 bits
- Supports dynamic programming of each GPIO bit as input or output
- Supports individual configuration of each channel
- Supports independent reset values for each bit of all registers
- Supports optional interrupt request generation

The top-level block diagram of the AXI GPIO core is shown in the following figure:

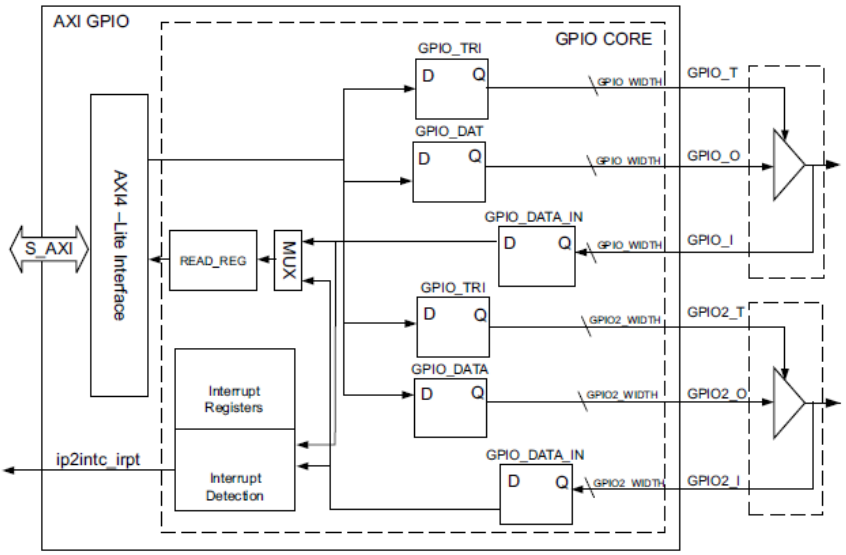

Figure 16: GPIO internal block diagram

3.5.2 Register space

The following table shows the AXI GPIO registers and their addresses:

Address Space Offset[3]	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register.
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register.
0x0008	GPIO2_DATA	R/W	0x0	Channel 2 AXI GPIO Data Register.
0x000C	GPIO2_TRI	R/W	0x0	Channel 2 AXI GPIO 3-state Control.
0x011C	GIER[1]	R/W	0x0	Global Interrupt Enable Register.
0x0128	IP IER[1]	R/W	0x0	IP Interrupt Enable Register (IP IER).
0x0120	IP ISR[1]	R/TOW[2]	0x0	IP Interrupt Status Register.

Notes:

1. Interrupt registers are available only if AXI GPIO is compiled using the **Enable Interrupt** parameter.
2. Toggle-On-Write (TOW) access toggles the status of the bit when a value of 1 is written to the corresponding bit.
3. Address Space Offset is relative to C_BASEADDR assignment.

Table 8: AXI GPIO registers

The AXI GPIO data register (GPIOx_DATA) is used to read the general purpose input ports and write to the general purpose output ports. When a port is configured as input, writing to the AXI GPIO data register has no effect.

Figure 17: AXI GPIO data register

Bits	Field Name	Access Type	Reset Value	Description
[GPIOx_Width-1 :0]	GPIOx_DATA	Read/Write	GPIO: Default Output Value GPIO2: Default Output Value	AXI GPIO Data Register. For each I/O bit programmed as input: • R: Reads value on the input pin. • W: No effect. For each I/O bit programmed as output: • R: Reads to these bits always return zeros • W: Writes value to the corresponding AXI GPIO data register bit and output pin.

Table 9: AXI GPIO data register description

The AXI GPIO 3-state control register (GPIOx_TRI) is used to configure the ports dynamically as input or output. When a bit within this register is set, the corresponding I/O port is configured as an input port. When a bit is cleared, the corresponding I/O port is configured as an output port.

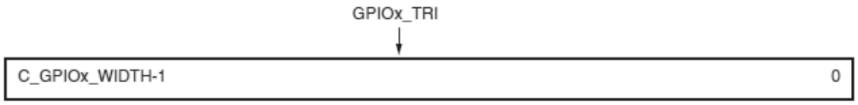

Figure 18: AXI GPIO 3-state control register

Bits	Field Name	Access Type	Reset Value	Description
[GPIOx_Width-1 :0]	GPIOx_TRI	Read/ Write	GPIO: Default Tri State Value GPIO2: Default Tri State Value	AXI GPIO 3-State Control Register. Each I/O pin of the AXI GPIO is individually programmable as an input or output. For each of the bits: 0 = I/O pin configured as output. 1 = I/O pin configured as input.

Table 10: AXI GPIO data register description

The AXI GPIO core can be configured under the control of the Enable Interrupt parameter to generate a level interrupt when a transition occurs in any of the channel inputs. The GPIO interface module includes interrupt detection logic to identify any transition on channel inputs. When a transition is detected, it is indicated to the Interrupt Controller module. The Interrupt Controller module implements the necessary registers to enable and maintain the status of the interrupts.

3.5.3 Device driver for PMOD

The following code shows the implementation of the kernel module.

```
/* Device Probe function for driver
* ------------------------------------
*  Get the resource structure from the information in device tree.
*  request the memory region needed for the controller, and map it into
*  kernel virtual memory space. Create an entry under /proc file system
*  and register file operations for that entry.
*/
static int driver_probe(struct platform_device *pdev)
{
    struct proc_dir_entry * driver_proc_entry;
    int rc = SUCCESS;
```

```
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res)
    {
        dev_err(&pdev->dev, "No memory resource\n");
        rc = -ENODEV;
    }

    if (rc == SUCCESS)
    {
        remap_size = res->end - res->start + 1;
        if (!request_mem_region(res->start, remap_size, pdev->name))
        {
            dev_err(&pdev->dev, "Cannot request IO\n");
            rc = -ENXIO;
        }
    }

    if (rc == SUCCESS)
    {
        base_addr = ioremap(res->start, remap_size);
        if (base_addr == NULL)
        {
            dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n", (unsigned
long)res->start);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        driver_proc_entry = proc_create(DRIVER_NAME, 0, NULL, &proc_driver_operations);
        if (driver_proc_entry == NULL)
        {
            dev_err(&pdev->dev, "Couldn't create proc entry\n");
            iounmap(base_addr);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        printk(KERN_INFO DRIVER_NAME " Mapped at virtual address 0x%08lx\n", (unsigned
long) base_addr);

        major = register_chrdev(0, DRIVER_NAME, &proc_driver_operations);

        configport(0x0000FFFF);
        iowrite32(0x000000, base_addr);
    }

    return rc;
}
```

At the end of the kernel module initialization, it is initialized the GPIO device in order to make it operational with the lower two bytes as input and the rest two as output.

```
...
        configport(0x0000FFFF);
        iowrite32(0x000000, base_addr);
...

static void configport(u32 data)
{
    static u32 current_data = 0;

    if (current_data != data)
    {
        wmb();
        iowrite32(data, base_addr + 1);
        current_data = data;
    }
}
```

The strategy to handle devices through the GPIO is to access individual bit or slides from the GPIO, and assign these to a particular purpose (abstract device).

The macro to read and write a particular slide in the GPIO is the following:

```
#define SET_SLICE_32(register, value, offset, mask)
(register) = (((register) & ~((mask) << (offset))) | (((value) & (mask)) << (offset)))


#define GET_SLICE_32(register, offset, mask)
(((register) & ((mask) << (offset))) >> (offset))
```

The read and write file operations. This function contains the definition of the devices that will be handled by the GPIO, this definition of devices can be found in the following headers files (shared with the user application).

```
...
#define KERNEL_MODULE
#include "../../app-workspace/base_framework/device/iodef.hpp"
#include "../../app-workspace/extension_framework/deviceid.hpp"
…
/* Define Driver Name */
#define DRIVER_NAME "controller"

#define SUCCESS     0


static unsigned long *   base_addr   = NULL;
static struct resource * res         = NULL;
static unsigned long     remap_size  = 0;
static DeviceID          device_ID   = 0;
static u32               output_data = 0;
...
/* Write operation for /proc/driver
* ----------------------------------
*/
static ssize_t proc_driver_write(struct file *file,
                                 const char __user * buffer,
                                 size_t buffer_size,
                                 loff_t * position)
{
    ssize_t written_size = 0;

    if (buffer != NULL)
    {
        if (buffer_size == sizeof(device_ID))
        {
            written_size = buffer_size - copy_from_user((void *) &device_ID, buffer,
sizeof(device_ID));
        }
        else if (buffer_size == sizeof(IOPacket))
        {
            IOPacket packet;
            written_size = buffer_size - copy_from_user((void *) &packet, buffer,
sizeof(IOPacket));

            if (written_size == buffer_size )
            {
                device_ID = packet.device_ID;
                switch (packet.device_ID)
                {
                    case IRSENSOR:
                        break;
                    case FLUSHVALVE:
                        SET_SLICE_32(output_data, packet.data, 16, 0x1);
                        break;
                    case DRAINVALVE:
                        SET_SLICE_32(output_data, packet.data, 17, 0x1);
                        break;
                    case SHUTOFFVALVE:
                        SET_SLICE_32(output_data, packet.data, 18, 0x1);
                        break;
                    case DRAINLOCAL:
                        break;
```

```
                    case EMERGENCY:
                        break;
                    case APPSELECTION:
                        break;
                    case DRAINDELAY:
                        break;
                    case VACUMGEN:
                        SET_SLICE_32(output_data, packet.data, 19, 0x1);
                        break;
                    case DRAININDICATOR:
                        SET_SLICE_32(output_data, packet.data, 20, 0x1);
                        break;
                    case LEAKINDICATOR:
                        SET_SLICE_32(output_data, packet.data, 21, 0x1);
                        break;
                    case RELAY_0:
                        SET_SLICE_32(output_data, packet.data, 22, 0x1);
                        break;
                    case RELAY_1:
                        SET_SLICE_32(output_data, packet.data, 23, 0x1);
                        break;
                    default:;
                }
                outport(output_data);
                * position += written_size;
            }
        }
    }
    return written_size;
}

/* Read operation for /proc/driver
* ----------------------------------
*/
static ssize_t proc_driver_read(struct file *file,
                                char __user * buffer,
                                size_t buffer_size,
                                loff_t * position)
{
    ssize_t read_size = 0;

    if ((buffer != NULL) && (sizeof(IOPacket) <= buffer_size))
    {
        IOPacket packet;
        u32 input_data = inport();
        packet.device_ID = device_ID;

        switch (device_ID)
        {
            case FLUSHVALVE:
                packet.data = GET_SLICE_32(input_data, 0, 1);
                break;
            case DRAINVALVE:
                packet.data = GET_SLICE_32(input_data, 1, 1);
                break;
            case SHUTOFFVALVE:
                packet.data = GET_SLICE_32(input_data, 2, 1);
                break;
            case VACUMGEN:
                packet.data = GET_SLICE_32(input_data, 3, 1);
                break;
            case IRSENSOR:
                packet.data = GET_SLICE_32(input_data, 5, 1);
                break;
            case DRAINLOCAL:
                packet.data = GET_SLICE_32(input_data, 6, 1);
                break;
            case EMERGENCY:
                packet.data = GET_SLICE_32(input_data, 7, 3);
                break;
            case APPSELECTION:
                packet.data = GET_SLICE_32(input_data, 9, 1);
                break;
            case DRAINDELAY:
                packet.data = GET_SLICE_32(input_data, 10, 7);
                break;
            case DRAININDICATOR:
                packet.data = GET_SLICE_32(input_data, 20, 1);
                break;
            case LEAKINDICATOR:
```

```
                packet.data = GET_SLICE_32(input_data, 21, 1);
                break;
            case RELAY_0:
                packet.data = GET_SLICE_32(input_data, 22, 1);
                break;
            case RELAY_1:
                packet.data = GET_SLICE_32(input_data, 23, 1);
                break;
            default:;
        }
        read_size = buffer_size - copy_to_user(buffer, (void *) &packet,
sizeof(IOPacket));
    }
    return read_size;
}
```

The implementation of this device driver can be found in the Appendix B – Controller .

3.5.4 Device tree node for PMOD

The device node added to the device tree for the GPIO is named controller:

```
/dts-v1/;
/ {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "xlnx,zynq-7000";
    model = "Xilinx Zynq";
    ...
    ps7_axi_interconnect_0: amba@0 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
        ranges ;
        ...
        controller
        {
            compatible = "yarib-controller-1.00.a";
            reg = <0x41200000 0x10000>;
        } ;
    } ;
} ;
```

The full implementation of the device tree can be found in the Appendix B – Device tree.

3.5.5 Device driver for ZYBO (pushbuttons, switches and LEDs)

The following code shows the implementation of the kernel module.

```
/* Device Probe function for driver
* ------------------------------------
*  Get the resource structure from the information in device tree.
*  request the memory region needed for the controller, and map it into
*  kernel virtual memory space. Create an entry under /proc file system
*  and register file operations for that entry.
*/
static int driver_probe(struct platform_device *pdev)
{
    struct proc_dir_entry * driver_proc_entry;
    int rc = SUCCESS;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res)
    {
        dev_err(&pdev->dev, "No memory resource\n");
        rc = -ENODEV;
    }

    if (rc == SUCCESS)
    {
        remap_size = res->end - res->start + 1;
        if (!request_mem_region(res->start, remap_size, pdev->name))
```

```
        {
            dev_err(&pdev->dev, "Cannot request IO\n");
            rc = -ENXIO;
        }
    }

    if (rc == SUCCESS)
    {
        base_addr = ioremap(res->start, remap_size);
        if (base_addr == NULL)
        {
            dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n", (unsigned
long)res->start);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        driver_proc_entry = proc_create(DRIVER_NAME, 0, NULL, &proc_driver_operations);
        if (driver_proc_entry == NULL)
        {
            dev_err(&pdev->dev, "Couldn't create proc entry\n");
            iounmap(base_addr);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        printk(KERN_INFO DRIVER_NAME " Mapped at virtual address 0x%08lx\n", (unsigned
long) base_addr);

        major = register_chrdev(0, DRIVER_NAME, &proc_driver_operations);

        output_data = 0;
        outport(output_data, 2);
    }

    return rc;
}
```

At the end of the kernel module initialization, it is initialized the GPIO clearing the second channel (LEDs).

```
...
        output_data = 0;
        outport(output_data, 2);
    }

    return rc;
}
...

static void outport(u32 data, u32 offset)
{
    static u32 current_data = 0;

    if (current_data != data)
    {
        wmb();
        iowrite32(data, base_addr + offset);
        current_data = data;
    }
}
```

The strategy to handle devices through the GPIO is to access slides from the GPIO channels.

The macros to read and write a particular slide in the GPIO are the following:

```
#define SET_SLICE_32(register, value, offset, mask)
(register) = (((register) & ~((mask) << (offset))) | (((value) & (mask)) << (offset)))
```

```
#define GET_SLICE_32(register, offset, mask)
(((register) & ((mask) << (offset))) >> (offset))
```

The read and write file operations. This function contains the definition of the devices that will be handled by the GPIO, this definition of devices can be found in the following headers files (shared with the user application).

```
...
#define KERNEL_MODULE
#include "../../app-workspace/base_framework/device/iodef.hpp"
#include "../../app-workspace/extension_framework/deviceid.hpp"
...
/* Define Driver Name */
#define DRIVER_NAME "zybo"

#define SUCCESS     0


static unsigned long *   base_addr   = NULL;
static struct resource * res         = NULL;
static unsigned long     remap_size  = 0;
static DeviceID          device_ID   = 0;
static u32               output_data = 0;

static int               major;       /* major number we get from the kernel */
...
/* Write operation for /proc/driver
* ---------------------------------
*/
static ssize_t proc_driver_write(struct file *file,
                                 const char __user * buffer,
                                 size_t buffer_size,
                                 loff_t * position)
{
    ssize_t written_size = 0;

    if (buffer != NULL)
    {
        if (buffer_size == sizeof(device_ID))
        {
            written_size = buffer_size - copy_from_user((void *) &device_ID, buffer,
sizeof(device_ID));
        }
        else if (buffer_size == sizeof(IOPacket))
        {
            IOPacket packet;
            written_size = buffer_size - copy_from_user((void *) &packet, buffer,
sizeof(IOPacket));

            if (written_size == buffer_size )
            {
                device_ID = packet.device_ID;
                switch (packet.device_ID)
                {
                    case ZYBO_BUTTONS:
                        break;
                    case ZYBO_SWITCHES:
                        break;
                    case ZYBO_LEDS:
                        SET_SLICE_32(output_data, packet.data, 0, 0xF);
                        break;
                    default:;
                }
                outport(output_data, 2);
                * position += written_size;
            }
        }
    }
    return written_size;
}

/* Read operation for /proc/driver
* --------------------------------
*/
static ssize_t proc_driver_read(struct file *file,
```

```
                                char __user * buffer,
                                size_t buffer_size,
                                loff_t * position)
{
    ssize_t read_size = 0;

    if ((buffer != NULL) && (sizeof(IOPacket) <= buffer_size))
    {
        IOPacket packet;
        u32 input_data = inport(0);
        packet.device_ID = device_ID;

        switch (device_ID)
        {
            case ZYBO_BUTTONS:
                packet.data = GET_SLICE_32(input_data, 0, 0xF);
                break;
            case ZYBO_SWITCHES:
                packet.data = GET_SLICE_32(input_data, 4, 0xF);
                break;
            case ZYBO_LEDS:
                packet.data = GET_SLICE_32(output_data, 0, 0xF);
                break;
            default:;
        }
        read_size = buffer_size - copy_to_user(buffer, (void *) &packet,
sizeof(IOPacket));
    }
    return read_size;
}
```

The entire implementation of this device driver can be found in the Appendix B – ZYBO.

3.5.6 Device tree node for ZYBO

The device node added to the device tree for the GPIO is named zybo:

```
/dts-v1/;
/ {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "xlnx,zynq-7000";
    model = "Xilinx Zynq";
    ...
    ps7_axi_interconnect_0: amba@0 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
        ranges ;
        ...
        zybo
        {
            compatible = "yarib-zybo-1.00.a";
            reg = <0x41210000 0x10000>;
        } ;
    } ;
} ;
```

The entire implementation of the device tree can be found in the Appendix B – Device tree.

3.6 PWM

The PWM that is used for this prototype is an AXI Timer v2.0 configured as PWM. This information is detailed in the Xilinx AXI Timer v2.0 LogiCORE IP Product Guide. [25]

The block design contains two PWM, both have the same configuration.

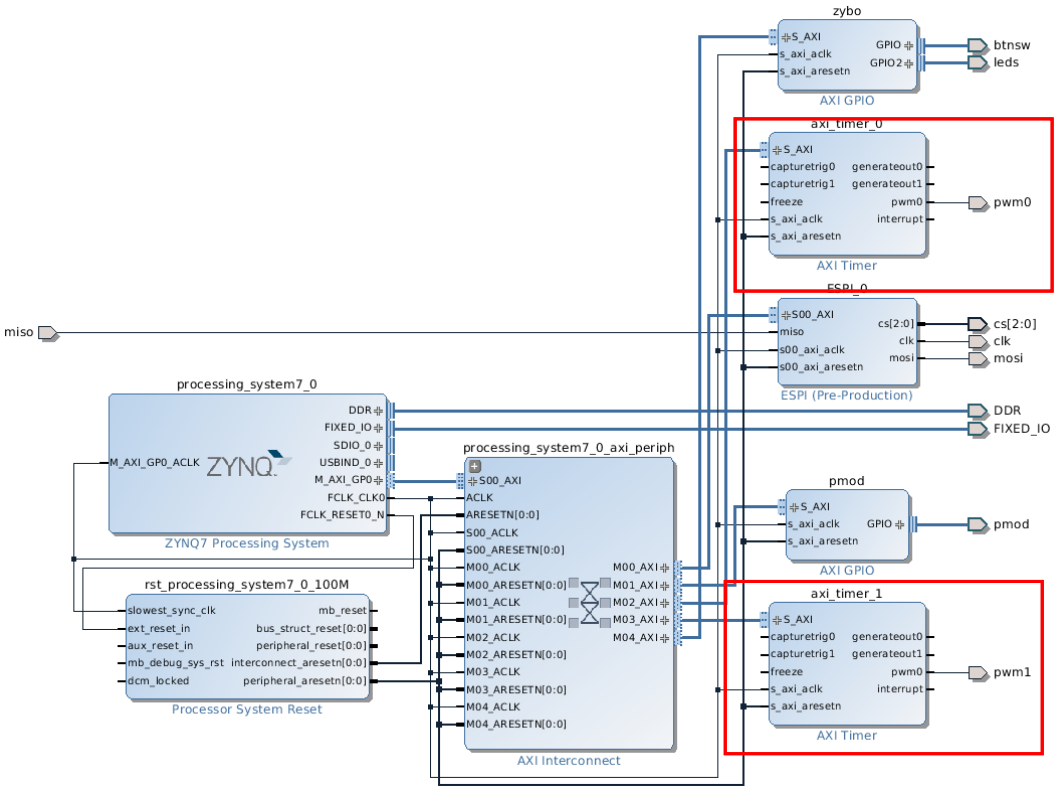

Figure 19: PWM hardware blocks

3.6.1 Description

The AXI Timer/Counter module provides an AXI4-Lite interface to communicate with the host. The timer/counter design has the following key modules:

- **AXI4-Lite interface**: The AXI4-Lite interface module implements an AXI4-Lite slave interface for accessing memory mapped Timer registers. For details about the AXI4-Lite slave interface, see the *LogiCORE IP AXI4-Lite IPIF Data Sheet* (DS765)
- **Timer registers**: The Register block implements a set of 32-bit registers for each timer/counter. This set of register contains load register, timer/counter register and control/status register.
- **32-bit counters**: The Timer/Counter module has two 32-bit counters, each of which can be configured for up/down counts and can be loaded with a value from the load register.
- **Interrupt control**: The Interrupt control module generates a single interrupt depending on the mode of operation.
- **Pulse Width Modulation (PWM)**: The PWM block generates a pulse signal, PWM0, with a specified frequency and duty factor. It uses Timer 0 for PWM0 period, and Timer 1 for the PWM0 output width.

The block diagram of AXI Timer, also known as AXI Timer/Counter, is shown in the following diagram:

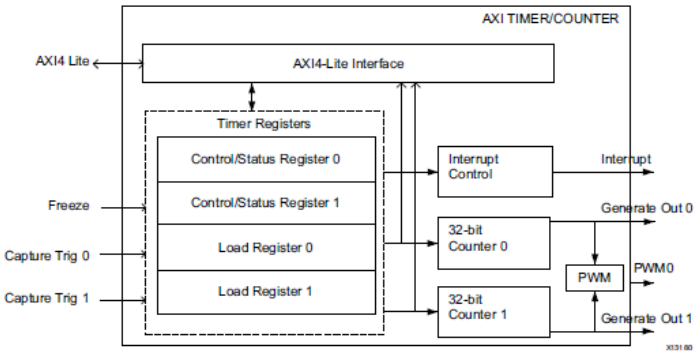

Figure 20: Internal block diagram of AXI Timer

3.6.2 Register space

The following table shows the AXI GPIO registers and their addresses:

Address Offset	Register Name	Description
0h	TCSR0	Timer 0 Control and Status Register
04h	TLR0	Timer 0 Load Register
08h	TCR0	Timer 0 Counter Register
0Ch-0Fh	RSVD	Reserved
10h	TCSR1	Timer 1 Control and Status Register
14h	TLR1	Timer 1 Load Register
18h	TCR1	Timer 1 Counter Register
1Ch-1Fh	RSVD	Reserved

Table 11: AXI Timer registers

Control/Status register 0 (TCSR0) contains the control and status bits for timer module 0.

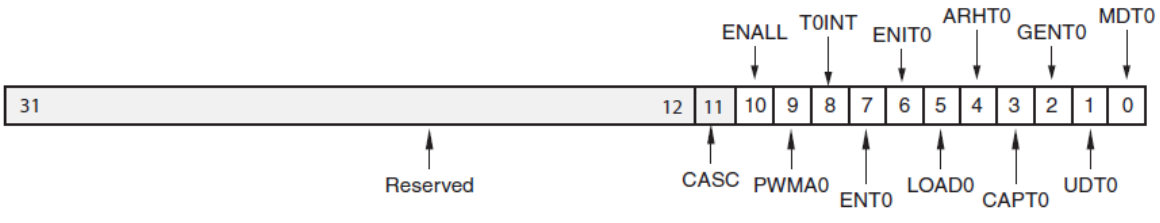

Figure 21: Control/Status register 0

For more detailed information regarding these registers, it can be referred to the datasheet Xilinx AXI Timer v2.0 LogiCORE IP Product Guide. [25].

Load Register (TLR0 and TLR1). When the counter width has been configured as less than 32 bits, the load register value is right-justified in TLR0 and TLR1. The least-significant counter bit is always mapped to load register bit 0.

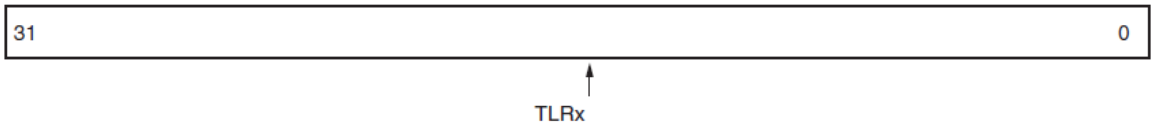

Figure 22: Timer/Counter load register

Timer/Counter register (TCR0 and TCR1). When the counter width has been configured as less than 32 bits, the count value is right-justified in TCR0 and TCR1. In cascade mode, TCR0 has the least significant 32 bits of the 64-bit counter, and TCR1 has the most significant bits.

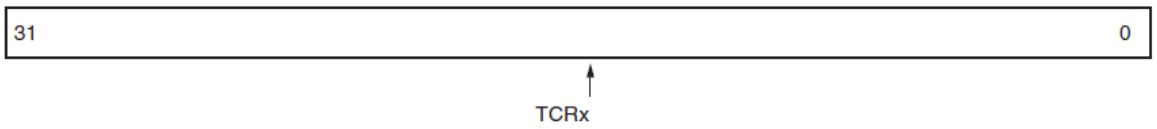

Figure 23: Timer/Counter register

Control/Status Register 1 (TCSR1) contains the control and status bits for timer module 1. This register is used only for loading the TLR1 register in cascade mode.

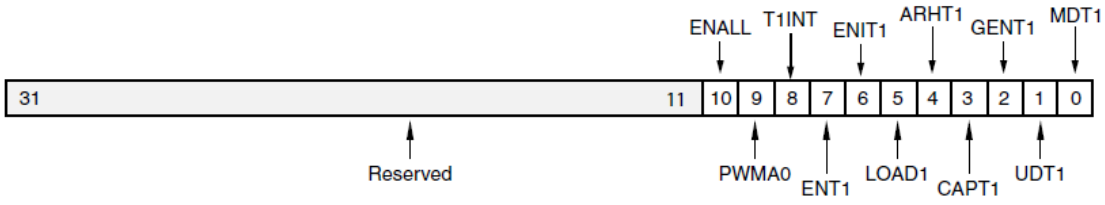

Figure 24: Control/Status register 1

3.6.3 Timer modes

Four modes can be used with the two Timer/Counter modules:

- Generate mode
- Capture mode
- Pulse Width Modulation mode
- Cascade mode

3.6.4 Pulse Width Modulation mode

In Pulse Width Modulation (PWM) mode, two timer/counters are used as a pair to produce an output signal (PWM0) with a specified frequency and duty factor. Timer 0 sets the period and Timer 1 sets the high time for the PWM0 output.

3.6.5 Programming sequence

Pulse Width Modulation (PWM) mode has the following programming sequence:

- The mode for both Timer 0 and Timer 1 must be set to generate mode (bit MDT in the TCSR set to 0).
- The PWMA0 bit in TCSR0 and PWMB0 bit in TCSR1 must be set to 1 to enable PWM mode.

- The GenerateOut signals must be enabled in the TCSR (bit GENT set to 1). The PWM0 signal is generated from the GenerateOut signals of Timer 0 and Timer 1, so these signals must be enabled in both timer/counters.
- The assertion level of the GenerateOut signals for both timers in the pair must be set to Active High.
- The counter can be set to count up or down.

3.6.6 Setting the PWM period and duty factor

The PWM period is determined by the generated value in the Timer 0 load register (TLR0). The PWM high time is determined by the generated value in the Timer 1 load register (TLR1). The period and duty factor are calculated as follows:

When counters are configured to count up (UDT = 0):

```
PWM_PERIOD = (MAX_COUNT - TLR0 + 2) * AXI_CLOCK_PERIOD
PWM_HIGH_TIME = (MAX_COUNT - TLR1 + 2) * AXI_CLOCK_PERIOD
```

When counters are configured to count down (UDT = 1):

```
PWM_PERIOD = (TLR0 + 2) * AXI_CLOCK_PERIOD
PWM_HIGH_TIME = (TLR1 + 2) * AXI_CLOCK_PERIOD
```

Where MAX_COUNT is the maximum count value for the counter, such as 0xFFFFFFFF for a 32-bit counter.

3.6.7 Device driver

The following code shows the implementation of the kernel module. Both PWM0 and PWM1 have the same configuration and their kernel modules are independent instances.

```
#define KERNEL_MODULE
#include "../../app-workspace/base_framework/device/iodef.hpp"
#include "../../app-workspace/extension_framework/deviceid.hpp"

/* Define Driver Name */
#define DRIVER_NAME     "pwm_0"

static unsigned long   * base_addr  = NULL;
static struct resource * res        = NULL;
static unsigned long     remap_size = 0;
static int               major;     // major number we get from the kernel
static DeviceID          device_ID = 0;

#define SUCCESS 0

/* Device Probe function for driver
* ------------------------------------
*  Get the resource structure from the information in device tree.
*  request the memory region needed for the controller, and map it into
*  kernel virtual memory space. Create an entry under /proc file system
*  and register file operations for that entry.
*/
static int driver_probe(struct platform_device *pdev)
{
    struct proc_dir_entry * driver_proc_entry;
    int rc = SUCCESS;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res)
    {
        dev_err(&pdev->dev, "No memory resource\n");
```

```
        rc = -ENODEV;
    }

    if (rc == SUCCESS)
    {
        remap_size = res->end - res->start + 1;
        if (!request_mem_region(res->start, remap_size, pdev->name))
        {
            dev_err(&pdev->dev, "Cannot request IO\n");
            rc = -ENXIO;
        }
    }

    if (rc == SUCCESS)
    {
        base_addr = ioremap(res->start, remap_size);
        if (base_addr == NULL)
        {
            dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n", (unsigned
long)res->start);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        driver_proc_entry = proc_create(DRIVER_NAME, 666, NULL,
&proc_driver_operations);
        if (driver_proc_entry == NULL)
        {
            dev_err(&pdev->dev, "Couldn't create proc entry\n");
            iounmap(base_addr);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        printk(KERN_INFO DRIVER_NAME " Mapped at virtual address 0x%08lx\n", (unsigned
long) base_addr);

        major = register_chrdev(0, DRIVER_NAME, &proc_driver_operations);

        pwm_initialize();
    }

    return rc;
}
```

At the end of the kernel module initialization, it is initialized the AXI Timer to operate as a PWM. The PWM frequency can be defined by using the macros.

```
//////////////////////////////////////////////////////////////////////////////
#define AXI_CLK_FREQUENCY_HZ  50000000 // <--- Define AXI clk frequency !
#define PWM_FREQUENCY         1000     // <--- Define desired PWM frequency !
//////////////////////////////////////////////////////////////////////////////
#define CALCULATE_PERIOD_REGISTER(AXI_FREQ, PWM_FREQ)   (((AXI_FREQ) / (PWM_FREQ)) - 2)
#define AXI_PERIOD_REGISTER
CALCULATE_PERIOD_REGISTER(AXI_CLK_FREQUENCY_HZ, PWM_FREQUENCY)

static void outport(u8 offset, u32 data)
{
    wmb();
    iowrite32(data, base_addr + offset);
}

static u32 inport(u8 offset)
{
    wmb();
    return ioread32(base_addr + offset);
}

static void pwm_set_period(u32 period)
{
```

```
    outport(TLR0, period);
}

static u32  pwm_get_period(void)
{
    return inport(TLR0);
}

static void pwm_set_high_time(u32 high_time)
{
    outport(TLR1, high_time);
}

static u32  pwm_get_high_time(void)
{
    return inport(TLR1);
}

static void pwm_initialize(void)
{
    outport(TCSR0, ENALL0 | PWMA0 | GENT0 | UDT0);
    outport(TCSR1, ENALL1 | PWMB0 | GENT1 | UDT1);

    pwm_set_period(AXI_PERIOD_REGISTER);
    pwm_set_high_time(0);
}
```

The read and write file operations.

```
/* Write operation for /proc/driver
* ---------------------------------
*/
static ssize_t proc_driver_write(struct file *file,
                                 const char __user * buffer,
                                 size_t buffer_size,
                                 loff_t * position)
{
    ssize_t written_size = 0;

    if (buffer != NULL)
    {
        if (buffer_size == sizeof(device_ID))
        {
            written_size = buffer_size - copy_from_user((void *) &device_ID, buffer,
sizeof(device_ID));
        }
        else if (buffer_size == sizeof(IOPacket))
        {
            IOPacket packet;
            written_size = buffer_size - copy_from_user((void *) &packet, buffer,
sizeof(IOPacket));

            if (written_size == buffer_size )
            {
                device_ID = packet.device_ID;
                switch (packet.device_ID)
                {
                    case PWM_0:
                        pwm_set_high_time(packet.data);
                        break;
                    default:;
                }
                * position += written_size;
            }
        }
    }
    return written_size;
}

/* Read operation for /proc/driver
* --------------------------------
*/
static ssize_t proc_driver_read(struct file *file,
                                char __user * buffer,
                                size_t buffer_size,
                                loff_t * position)
{
```

```
    ssize_t read_size = 0;

    if ((buffer != NULL) && (sizeof(IOPacket) <= buffer_size))
    {
        IOPacket packet;
        packet.device_ID = device_ID;
        packet.data = pwm_get_high_time();
        read_size = buffer_size - copy_to_user(buffer, (void *) &packet,
sizeof(IOPacket));
    }
    return read_size;
}
```

The complete implementation of this device driver can be seen in the Appendix B – PWM.

3.6.8 Device tree node

The device nodes added to the device tree for the two PWM are named **pwm_0** and **pwm_1**, both drivers have the same implementation:

```
/dts-v1/;
/ {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "xlnx,zynq-7000";
    model = "Xilinx Zynq";
    ...
    ps7_axi_interconnect_0: amba@0 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
        ranges ;
        ...
        pwm_0
        {
            compatible = "yarib-pwm_0-1.00.a";
            reg = <0x42800000 0x10000>;
        } ;
        pwm_1
        {
            compatible = "yarib-pwm_1-1.00.a";
            reg = <0x42810000 0x10000>;
        } ;
    } ;
} ;
```

The complete implementation of the device tree can be found in the Appendix B – Device tree.

Chapter 4. Software architecture

The software architecture of a system is the set of structures which comprise software elements, relations among them, and their properties.

This software architecture implements a structure of an object-oriented application framework. This chapter describes it.

4.1 Object-oriented application framework

The Object-oriented (OO) application framework is a reusable, "semi-complete" software that can be extended and specialized to produce custom applications.

This software architecture is shown in the following image:

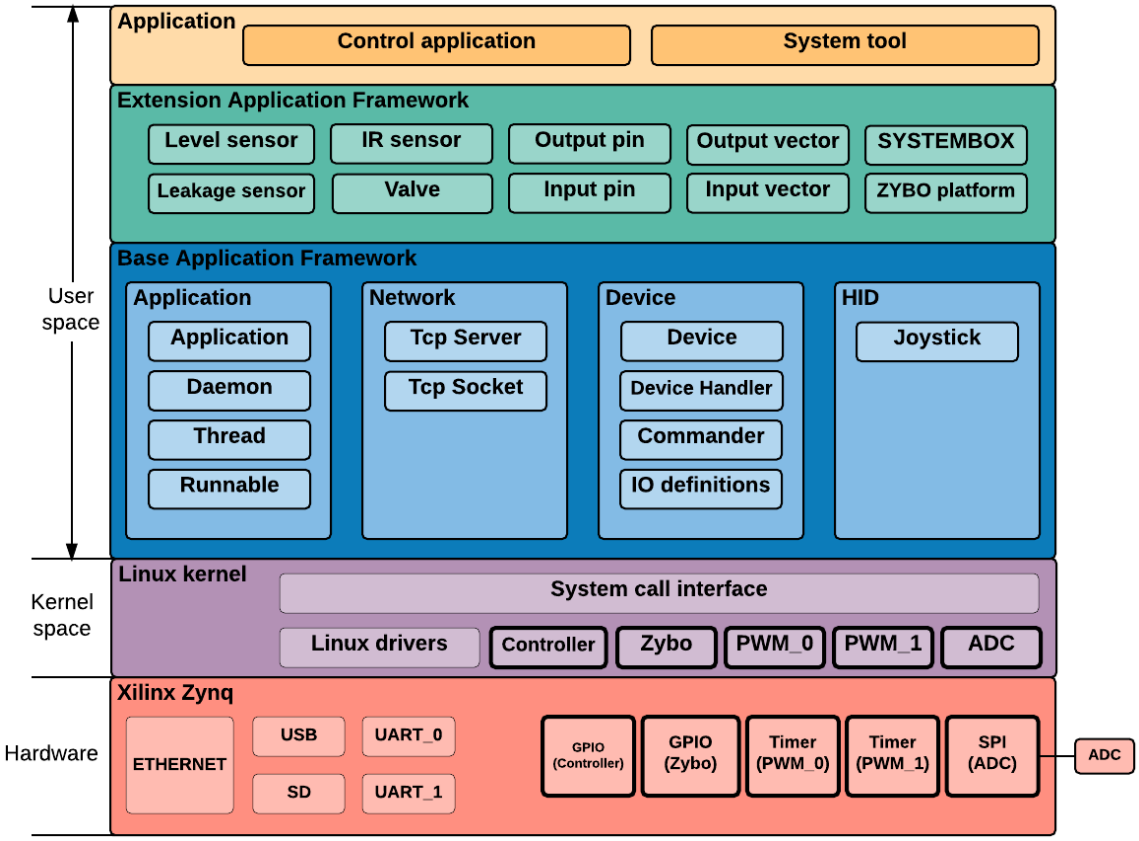

Figure 25: Software architecture

Description of layers.

- **Application**. This layer accomplishes efficient development of rich embedded Linux applications by using the extended application framework, the base application framework, as well as the Linux system call.
- **Extension application framework**. This layer is implemented to support specialized features for specific applications (for this prototype, it is specialized in avionics applications).
- **Base application framework**. This layer is the core of this architecture, it implements in a modular manner the main components of the software.

- **Linux kernel**. In this layer are located the custom device drivers as well as the generic kernel components.
- **Hardware**. This is the physical medium, this layer holds the hardware peripherals.

Chapter 5. **Base application framework**

The base application framework is a whitebox middleware integration framework. This is the foundation and integration layer between the Linux system call, and embedded Linux applications.

This software layer is designed with component-based patterns, it implements decoupled modules. Each module is individual, reusable and independent of each other.

Main characteristics:

- **Modularity**. Decoupled architecture that allows components to remain completely autonomous and unaware of each other.
- **Reusability**. Collection of software elements and generic features that enhance its suitability for reuse
- **Extensibility**. Design that takes future growth into consideration. Extensions can be seen as decoration patterns to add specialized functionalities or enhancements.

The image below displays the modules and classes implemented in this framework.

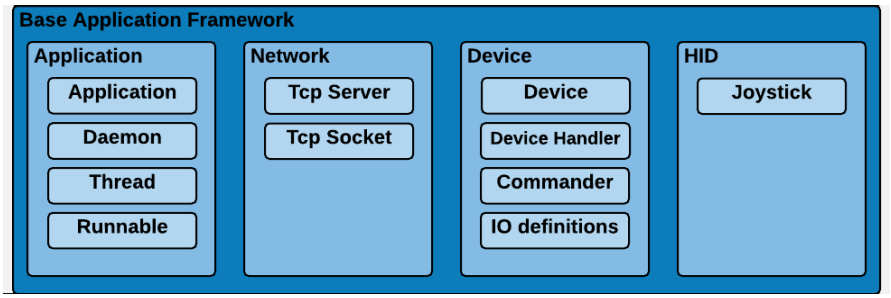

Figure 26: Base application framework

The following diagram and table show the directory structure and header files for each module.

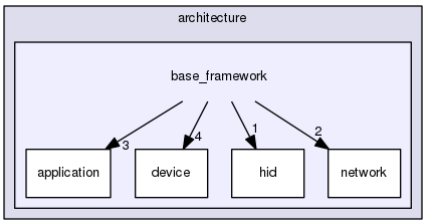

Figure 27: Directory diagram for base application framework

Module name	Module directory	Module header file
Application	application	*m_application.hpp*
Device	device	*m_device.hpp*
HID	hid	*m_hid.hpp*
Network	network	*m_network.hpp*

Table 12: Table of modules for base application framework

Note: In order to use a module in a development project, it is enough to include its corresponding header file and link to its folder in the project.

5.1 Device module

This module implements the support for proper handling of shared devices in multi-threading applications, the interface communication with the kernel modules, and generic command handling of the devices.

The module is displayed in the figure.

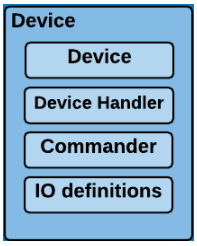

Figure 28: Device module

The header file for this module is *m_device.hpp*, it includes all the header files of the classes in this module.

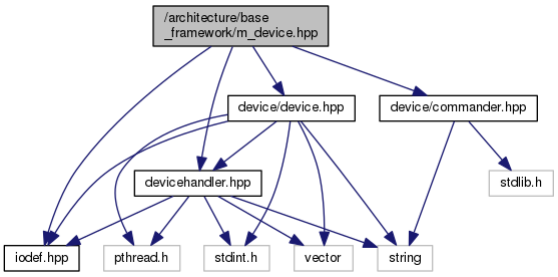

Figure 29: Header file diagram for Device module

5.1.1 IO definitions

The communication between the user space and kernel space is performed via the interchange of data packets.

5.1.1.1 IOPacket structure

The `IOPacket` structure is defined in C language, this structure is shared for compilation in both the kernel modules and user applications. For now this structure contains the device ID and the payload data, both structure members are unsigned integer (32-bit) type defined. Each packet travels between application and kernel caring with an ID and 32-bit data of information.

The data packets can be extended or reduced by increasing the size of the `data` member. These definitions can be found in iodef.hpp.

```
#ifdef __cplusplus
extern "C" {
#endif

#define SUCCESS 0

typedef unsigned int  DeviceID;
typedef unsigned int  Data;

typedef struct
{
    DeviceID device_ID;
    Data     data;
} IOPacket;

typedef struct
```

```
{
    const DeviceID  ID;
    const char *    name;
} DeviceIdentity;

#ifdef __cplusplus
}
#endif
```

The following figure shows the UML diagram of this structure.

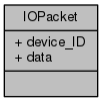

Figure 30: IOPacket structure

Public Attributes

```
DeviceID device_ID
Data data
```

5.1.1.2 DeviceIdentity structure

This structure gives identification to device instances by a unique ID and text string name. The devices are identifiable through the entire system, therefore this structure is defined in *iodef.hpp* (C language) and is shared for dependency compilation in both kernel and userspace applications.

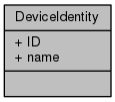

Figure 31: DeviceIdentity structure

Public Attributes

```
const DeviceID ID
const char * name
```

5.1.2 DeviceHandler class

This class is a file handler wrapper, it performs the file operations, as well as the blocking mechanism for shared devices in multiple threading. This class implements the mediator and bridge patterns and mechanisms for kernel-user communication. Any customization in the manner of reading from or writing to a kernel module should be places in this class.

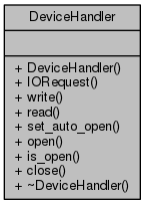

Figure 32: DeviceHandler class

Public Types

```
enum IORequestType { READ, WRITE, QUERY }
```

Public Member Functions

```
DeviceHandler (const char *const file_name)
size_t IORequest (void *buffer, size_t size, IORequestType irq, uint32_t query_ID=0)
size_t write (void *buffer, size_t size)
size_t read (void *buffer, size_t size)
void set_auto_open (bool)
bool open (void)
bool is_open (void)
bool close (void)
~DeviceHandler ()
```

5.1.2.1 Bridge to kernel

The bridge design pattern and the lock for concurrency are implemented in the `IOReques` function:

```
size_t  DeviceHandler::IORequest(void * buffer,
                                 size_t size,
                                 IORequestType irq,
                                 uint32_t query_ID)
{
    size_t result = 0;
    pthread_mutex_lock(&ioMutex);

    if (auto_open)
    {
        open();
    }

    if ((0 < size) & is_open())
    {
        switch (irq)
        {
        case WRITE:
            result = fwrite(buffer, size, 1, file_handler);
            fflush(file_handler);
            break;

        case READ:
            result = fread(buffer, size, 1, file_handler);
            fflush(file_handler);
            break;

        case QUERY:
            fwrite(&query_ID, sizeof(query_ID), 1, file_handler);
            fflush(file_handler);
            result = fread(buffer, size, 1, file_handler);
            break;

        default:;
        }
    }

    if (auto_open)
    {
        close();
    }

    pthread_mutex_unlock(&ioMutex);
    return result;
}
```

5.1.3 Device class

This class implements the base presentation of a device (abstraction layer from a file handler to a device). It holds a unique ID (number) and name (text) for the device. This class wraps, unwraps and verifies the data packets from the kernel. All created instances are registered in an internal static vector.

The following figures display UML class and collaboration diagrams.

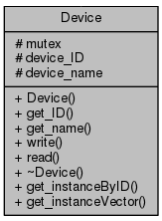

Figure 33: Device class

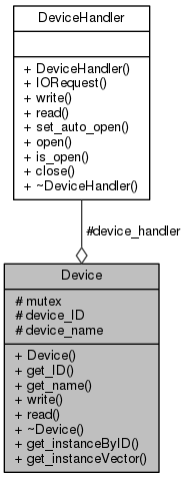

Figure 34: Collaboration diagram for Device class

Public Types

```
typedef std::vector< Device * > InstanceVector
```

Public Member Functions

```
Device (DeviceHandler *device_handler, DeviceIdentity *identity)
DeviceID get_ID (void)
std::string get_name (void)
bool write (uint32_t data)
bool read (uint32_t *data)
virtual ~Device ()
```

Static Public Member Functions

```
static Device * get_instanceByID (DeviceID id)
static InstanceVector & get_instanceVector (void)
```

Protected Attributes

```
pthread_mutex_t mutex
DeviceHandler * device_handler
DeviceID device_ID
std::string device_name
```

5.1.3.1 Device instance identification, registration and retrieval

A reference to each device instance is stored internally in a static vector as singleton pattern, and multition design, the registered instances can be retrieved either by its name string or by its ID. When deleting an instance, it is also deregistered from the vector. The creation of a device instance requires of `DeviceIdentity` and `DeviceHandler` by the constructor as a dependency injection pattern. More than one device may access the same kernel module, it means they will share the same instance of `DeviceHandler` (file handler).

The identification and registration of the instance is done inside the constructor:

```
Device::Device(DeviceHandler * device_handler, DeviceIdentity * identity):
device_handler(device_handler),
```

```
device_ID(0),
device_name("")
{
    pthread_mutex_init(&mutex, NULL);
    if (identity != NULL)
    {
        device_ID   = identity->ID;
        device_name = identity->name;
    }
    get_instanceVector().push_back(this);
}

Retrieval of instance by its ID:
Device * Device::get_instanceByID(DeviceID id)
{
    Device * instance = NULL;
    InstanceVector instances = get_instanceVector();
    for (unsigned i = 0; (i < instances.size()) && (instance == NULL); i ++)
    {
        if ((DeviceID)instances[i]->get_ID() == id)
        {
            instance = instances[i];
        }
    }

    return instance;
}
```

5.1.3.2 Packet wrapping

Wrapping and unwrapping of data packets:

```
bool Device::write(uint32_t data)
{
    bool result = false;
    pthread_mutex_lock(&mutex);
    if (device_handler != NULL)
    {
        IOPacket packet;
        packet.device_ID = device_ID;
        packet.data = data;

        result = device_handler->write(&packet, sizeof(packet)) > 0;
    }
    pthread_mutex_unlock(&mutex);
    return result;
}

bool Device::read(uint32_t * data)
{
    bool result = false;
    pthread_mutex_lock(&mutex);
    if (device_handler != NULL)
    {
        IOPacket packet;

        result = device_handler->IORequest(&packet,
                                           sizeof(packet),
                                           DeviceHandler::QUERY,
                                           device_ID) > 0;

        if (packet.device_ID == device_ID)
        {
            *data = packet.data;
        }
    }
    pthread_mutex_unlock(&mutex);
    return result;
}
```

These figures reveal the interface to the `DeviceHandler`.

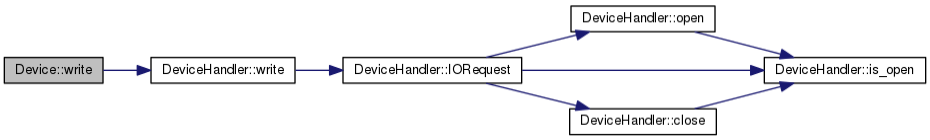

Figure 35: Call diagram for Device::write function

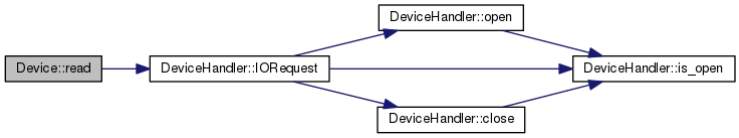

Figure 36: Call diagram for Device::read function

From both diagrams, it is seen that the read function directly calls **`DeviceHandler::IORequest`**, in order to read data from the kernel module it is necessary to first inform the kernel about it, then the kernel module prepares the requested data, and the kernel makes it available during the following read operation. Therefore, reading operation of a device, in reality, is a writing and then reading. This bridge pattern is implemented in **`DeviceHandler::IORequest`**.

5.1.4 Commander class

This class implements text commands to access and control the devices. When requesting a text command, the requested device is identified, then the instance is retrieved, and finally, it is executed either write or read according to the command.

Five command requests are implemented:

Command	Request
`DEVICES`	Returns the number of registered devices
`DEVICE n`	Returns the text name of the device indexed by n. The argument n should be positive and lower than the number retrieved by DEVICES command.
`<DEVICE_NAME> = X`	Calls the write method of the device given by its name. The X value is given as write parameter value.
`<DEVICE_NAME>`	Calls the read method of the device given by its name. The result is returned.
`EXIT`	Exits the commander loop.

Table 13: Commands

Note: If wrong command is given, the commander answers with a help message, described in Chapter 8 (8.1.1).

The commander class is UML diagram:

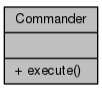

Figure 37: Commander class

Static Public Member Functions

```
static bool execute (std::string command, std::string &answer)
```

5.1.4.1 Command behavioural

The command handler is implemented in the following code:

```
bool Commander::execute(std::string command, std::string & answer)
{
    bool exit = false;

    Device::InstanceVector devices = Device::get_instanceVector();
    answer.clear();

    if (command.compare(0, 4, "EXIT") == 0)
    {
        answer = "Exit commander\n";
        exit = true;
    }
    if (command.compare(0, 7, "DEVICES") == 0)
    {
        char c_answer[128];
        sprintf(c_answer, "%d\n", devices.size());
        answer = c_answer;
    }
    if (command.compare(0, 7, "DEVICE ") == 0)
    {
        uint32_t i = strtoul(command.substr(6).c_str(), NULL, 0);
        if (i < devices.size())
        {
            answer = devices[i]->get_name() + "\n";
        }
        else
        {
            answer = "NO DEVICE\n";
        }
    }
    else
    {
        Device * device = NULL;

        for (unsigned i = 0; i < devices.size(); i ++)
        {
            if (command.compare(0,
                                devices[i]->get_name().size(),
                                devices[i]->get_name()) == 0)
            {
                device = devices[i];
            }
        }

        if (device != NULL)
        {
            uint32_t data;
            bool device_access;
            std::string::size_type token_equal;

            token_equal = command.find("=", device->get_name().size());

            if (token_equal == std::string::npos)
            {
                device_access = device->read(& data);
                answer = "[READ] ";
            }
            else
            {
                data = strtoul(command.substr(token_equal + 1).c_str(), NULL, 0);
                device_access = device->write(data);
                answer = "[WRITE] ";
            }

            if (device_access)
            {
                char c_answer[128];
                sprintf(c_answer, "%s = 0x%X\n", device->get_name().c_str(), data);
                answer += c_answer;
            }
            else
            {
                answer += "Couldn't access device: " + device->get_name() + "\n";
            }
        }
```

```
    }

    if (answer.empty() && (0 < devices.size()))
    {
        answer = "_________________ HELP _________________\n";
        answer += "List of devices:\n\n";
        for (unsigned i = 0; i < devices.size(); i ++)
        {
            answer += devices[i]->get_name() + "\n";
        }
        answer += "\n * Example reading:\n" + devices[0]->get_name() + "     (hit ENTER
key)\n";
        answer += "\n * Example writing:\n" + devices[0]->get_name() + " = 1 (hit ENTER
key)\n";
        answer += "\n * Get number of registered devices:\nDEVICES (hit ENTER key)\n";
        answer += "\n * Get name of a registered device:\nDEVICE 1 (hit ENTER key)\n";
        answer += "\n * Exit commander:\nEXIT (hit ENTER key)\n";
    }

    answer += "_____________________________________________\n";

    return exit;
}
```

5.2 Application module

This module provides the skeletons to for multi-threading Linux applications and background daemon processes.

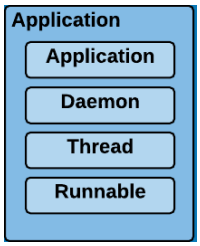

Figure 38: Application module

The header file for this module is *m_application.hpp*.

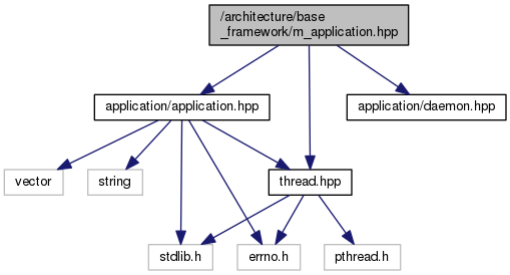

Figure 39: Header file diagram for Application module

5.2.1 Runnable class

Runnable is an abstract class, it provides the prototype pattern for runnable objects.

Figure 40: Runnable class

Public Member Functions

```
virtual int run (void)=0
```

As abstract class, it has `run` as a pure virtual method, this is the running entry point for the extended classes.

Active object design pattern.

```
class Runnable
{
public:
    virtual int run(void) = 0;
};
```

5.2.2 Thread class

This class extends from runnable, and this provides the threading handling to implement the active object patterns. However this class does not implement the pure virtual method (`run`), the result is again an abstract class which is decorated with threading mechanisms.

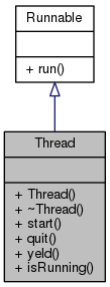

Figure 41: Inheritance diagram for Thread class

Public Member Functions

```
Thread (void)
virtual ~Thread (void)
virtual int start (void)
virtual int quit (void)
virtual int yeld (void)
virtual bool isRunning (void)
```

5.2.2.1 Execution decoupling

The `start` method initializes a thread object from the Linux system call, and then it starts `process_trigger` which is a static private method, it decouples the execution of the `run` method; this method is implemented in extended classes.

```
virtual int Thread::start(void)
{
    int result = -1;
    if (!isRunning())
    {
        result = pthread_create(&thread, NULL, process_trigger, this);
    }
    return result;
}

void * Thread::process_trigger(void * runnable)
{
    if (runnable != NULL)
    {
        pthread_exit((void *)((Runnable *) runnable)->run());
    }
    return EXIT_SUCCESS;
}
```

5.2.3 Application class

This class is the skeleton for multi-threading applications, it also provides the handling of calling arguments.

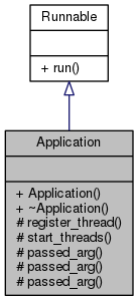

Figure 42: Inheritance diagram for Application class

Public Member Functions

```
Application (int argc=0, char *argv[]=NULL)
virtual ~Application ()
```

Protected Member Functions

```
void register_thread (Thread *thread)
int start_threads (void)
int passed_arg (std::string argument)
std::string passed_arg (int i)
int passed_arg (void)
```

5.2.3.1 Multi-threading handling

This class is an abstract class extending from Runnable, and decorated with the mechanisms of multi-threading management. This management is made by the registration of thread objects in an internal thread vector generally during the object construction, and then the execution of all registered threads is triggered by `start_threads`, they can be yielded, synchronized, and stopped at any time by the application since it owns the threads in its internal thread vector. Finally, the threads are stopped automatically when destroying the instance of the application.

Threads registering:

```
void Application::register_thread(Thread * thread)
{
    threads.push_back(thread);
}
```

Threads triggering:

```
int Application::start_threads(void)
{
    int result = EXIT_SUCCESS;
    for (unsigned int i = 0;
        (i < threads.size()) && (result == EXIT_SUCCESS);
        i ++)
    {
        result = threads[i]->start();
    }
    return result;
}
```

Threads halting:

```
Application::~Application()
{
    while (!threads.empty())
    {
        if (threads.back() != NULL)
            delete threads.back();
        threads.pop_back();
    }
}
```

5.2.4 Daemon class

This class provides the means to convert an application into a background process without any user interface, suitable to be a background service applications.

Figure 43: Daemon class

Public Types

```
enum { BD_NO_CHDIR = 01, BD_NO_CLOSE_FILES = 02, BD_NO_REOPEN_STD_FDS = 04, BD_NO_UMASK0
= 010, BD_MAX_CLOSE = 8192 }
```

Static Public Member Functions

```
static int become (int flags)
```

5.2.4.1 Background process

The fork system call creates a parallel running process, from the system call return two execution contexts, the caller process is terminated and returning the control to the caller process. The second process context continues with the program execution in a background state. This process is performed twice to detach parent processes.

```
    switch (fork())
    {
    case -1:
        return -1;
    case 0:
        break;
    default:
        _exit(EXIT_SUCCESS);
    }
```

5.3 Network module

The Network module provides a TCP/IP interface, this implements the client-server pattern for programming applications that require TCP/IP communication. This is an interprocess communication that can be used either for remote or local communication.

This module defines the transport layer of the TCP/IP model. Internet and Network Interface layers are driven by the own kernel modules and services.

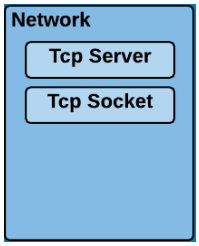

Figure 44: Network module

A TCP connection must be established to a remote host before any data transfer can begin. Once the connection has been established, at any time the peer can close the connection, and data transfer will then stop immediately.

The header file for this module is *m_network.hpp*.

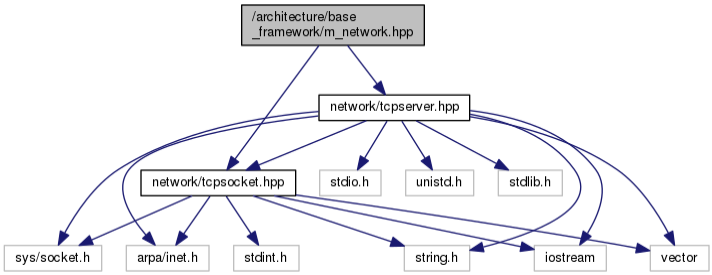

Figure 45: Header file diagram for Network module

5.3.1 TcpSocket class

This class implements a TCP/IP bridge, it encapsulates operational mechanisms, and this only provides interfaces for connection opening, closing, and data sending and receiving.

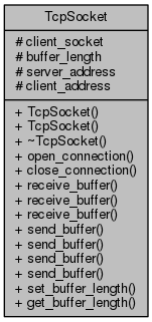

Figure 46: TcpSocket class

Public Member Functions

```
TcpSocket (void)
TcpSocket (int client_socket, struct sockaddr_in client_address)
virtual ~TcpSocket (void)
virtual int open_connection (char *host_address, uint16_t port)
virtual void close_connection (void)
virtual int receive_buffer (void * buffer, size_t length)
virtual int receive_buffer (std::string & string)
virtual int receive_buffer (ByteVector & vector)
virtual int send_buffer (void * buffer, size_t length)
virtual int send_buffer (const char * array)
```

```
virtual int send_buffer (std::string & string)
virtual int send_buffer (ByteVector &vector)
void set_buffer_length (size_t length)
size_t get_buffer_length (void)
```

Protected Types

```
enum { DEFAULT_BUFFER_LENGTH = 255 }
```

Protected Attributes

```
int client_socket
size_t buffer_length
struct sockaddr_in server_address
struct sockaddr_in client_address
```

5.3.1.1 TCP/IP bridge

Methods for opening and closing connection:

```
int TcpSocket::open_connection(char * host_address, uint16_t port)
{
    int result = -1;
    client_socket = socket(AF_INET , SOCK_STREAM , 0);

    if (client_socket != -1)
    {
        server_address.sin_addr.s_addr = inet_addr(host_address);
        server_address.sin_family = AF_INET;
        server_address.sin_port = htons(port);

        result = connect(client_socket,
                         (const struct sockaddr *) &server_address,
                         sizeof(server_address));
    }

    return result;
}

void TcpSocket::close_connection(void)
{
    if (client_socket != -1)
    {
        shutdown(client_socket, SHUT_RDWR);
        close(client_socket);
        client_socket = -1;
    }
}

TcpSocket::~TcpSocket(void)
{
    close_connection();
}

Sending and receiving data:
int TcpSocket::send_buffer(void * buffer, size_t length)
{
    return send(client_socket, buffer, length, 0);
}
int TcpSocket::send_buffer(ByteVector &vector)
{
    return send_buffer((void *)vector.data(), vector.size());
}

int TcpSocket::receive_buffer(ByteVector & vector)
{
    int result;

    ByteVector i_vector(buffer_length);

    result = receive_buffer(i_vector.data(), i_vector.size());
```

```
    if (result != -1)
    {
        vector = i_vector;
        vector.resize(result);
    }

    return result;
}
```

5.3.2 TcpServer class

This class implements a TCP/IP server that handles incoming connections. This class also encapsulates the operations for preparing the server, accepting and closing connections.

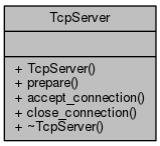

Figure 47: TcpServer class

Public Member Functions

```
TcpServer (uint16_t)
int prepare (void)
TcpSocket * accept_connection (void)
virtual void close_connection (void)
virtual ~TcpServer (void)
```

5.3.2.1 TCP/IP server setup

Server setup, this consist on the creation of a socket from Linux system call, and binding it to the current IP machine address.

```
int TcpServer::prepare(void)
{
    int result = EXIT_FAILURE;
    if (server_socket == -1)
    {
        server_socket = socket(AF_INET , SOCK_STREAM , 0);
    }
    if (server_socket != -1)
    {
        if (!bound)
        {
            server_address.sin_family = AF_INET;
            server_address.sin_addr.s_addr = INADDR_ANY;
            server_address.sin_port = htons(host_port);

            result = bind(server_socket,
                          (struct sockaddr *)&server_address,
                          sizeof(server_address));

            bound = result != -1;
        }
        else
        {
            result = EXIT_SUCCESS;
        }
    }
    return result;
}
```

Accepting connections, the method `accept_connection` makes the server to listen to incoming connections, synchronously the server waits for client connections, and with factory pattern, it creates a socket pipe to communicate through.

```
TcpSocket * TcpServer::accept_connection(void)
{
    TcpSocket * client = NULL;
    if (server_socket != -1)
    {
        int                 client_socket;
        struct sockaddr_in client_address;
        int                 addr_len;

        listen(server_socket , 1);

        addr_len = sizeof(struct sockaddr_in);
        client_socket = accept(server_socket,
                               (struct sockaddr *) &client_address,
                               (socklen_t*)&addr_len);

        if (client_socket != -1)
        {
            client = new TcpSocket(client_socket, client_address);
        }
    }
    return client;
}
```

Closing connections.

```
void TcpServer::close_connection(void)
{
    if (server_socket != -1)
    {
        shutdown(server_socket, SHUT_RDWR);
        close(server_socket);
        server_socket = -1;
    }
}

TcpServer::~TcpServer()
{
    close_connection();
}
```

5.4 HID module

This module provides event handling support of Human Interface Devices (HID). For now, this module handles generic USB Joystick devices, this allows the re-usage of buttons and axis for controlling or testing software.

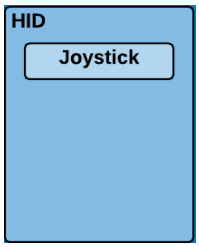

Figure 48: HID module

The header file for this module is *m_hid.hpp*

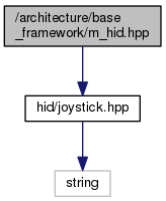

Figure 49: Header file diagram for HID module

5.4.1 Joystick class

This class provides the event handling for generic USB Joystick, it gets how many buttons and axis a device has, and it gets the states of these buttons and axis. This class implements a mediator pattern between the caller and the event handling mechanism.

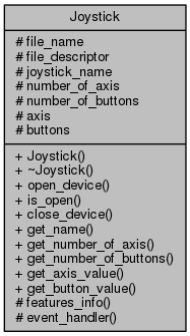

Figure 50: Joystick class

Public Member Functions

```
Joystick (const char *file_name)
virtual ~Joystick (void)
bool open_device (void)
bool is_open (void)
void close_device (void)
std::string get_name (void)
unsigned get_number_of_axis (void)
unsigned get_number_of_buttons (void)
int get_axis_value (int axis)
int get_button_value (int button)
```

Protected Member Functions

```
void features_info (void)
void event_handler (void)
```

Protected Attributes

```
std::string file_name
int file_descriptor
std::string joystick_name
int number_of_axis
int number_of_buttons
int * axis
int * buttons
```

5.4.1.1 HID features

The following method obtains the HID hardware characteristics from the kernel.

```
bool Joystick::open_device(void)
{
```

```
    if (!is_open())
    {
        file_descriptor = open((const char *)file_name.c_str(), O_RDONLY);
        features_info();
    }
    return is_open();
}

void Joystick::features_info(void)
{
    if (is_open())
    {
        char name[128];
        fcntl(file_descriptor, F_SETFL, O_NONBLOCK);
        ioctl(file_descriptor, JSIOCGNAME(128), &name);
        ioctl(file_descriptor, JSIOCGAXES, &number_of_axis);
        ioctl(file_descriptor, JSIOCGBUTTONS, &number_of_buttons);

        joystick_name = name;

        if (0 < number_of_axis)
        {
            axis = new int[number_of_axis];
            memset(axis, 0x0, sizeof(int) * number_of_axis);
        }

        if (0 < number_of_buttons)
        {
            buttons = new int[number_of_buttons];
            memset(buttons, 0x0, sizeof(int) * number_of_buttons);
        }
    }
}
```

5.4.1.2 HID event handling

The event handler pull all events from the kernel event-queue, then the HID hardware states are updated internally, finally the states can be read, this section of code can be modified to trigger call-backs.

```
void Joystick::event_handler(void)
{
    struct js_event js;
    struct js_event js_diff;

    if (is_open())
    {
        do
        {
            memcpy(&js_diff, &js, sizeof(struct js_event));
            read(file_descriptor, &js, sizeof(struct js_event));

            switch (js.type & ~JS_EVENT_INIT)
            {
                case JS_EVENT_AXIS:
                    axis[js.number] = js.value;
                    break;
                case JS_EVENT_BUTTON:
                    buttons[js.number] = js.value;
                    break;
            }
        }
        while (memcmp(&js, & js_diff, sizeof(struct js_event)) != 0);
    }
}
```

This is the method to read the state of a particular axis “a” (axis are numerated).

```
int Joystick::get_axis_value(int a)
{
    int value = 0;
    event_handler();
    if ((a < number_of_axis) && (axis != NULL))
    {
        value = axis[a];
    }
```

```
    return value;
}
```

This is the method to read the state of a particular button “b” (buttons are numerated).

```
int Joystick::get_button_value(int b)
{
    int value = 0;
    event_handler();
    if ((b < number_of_buttons) && (buttons != NULL))
    {
        value = buttons[b];
    }
    return value;
}
```

Chapter 6. **Extension application framework**

The extension application framework is a layer that implements classes for specific application domain, this extends from the base application framework (foundation layer).

This layer is developed to support specialized features of "The cabin convenient controller unit". This layer is fully operational; however, it serves as a template or base layer that will be customized according to future requirements of the controller.

The focus of this layer is summarized in the following points:

- **Platform specifications and definitions**. This is done by the definition of the kernel modules and their paths in the file system, as well as the definition of the device IDs and text names (sensors, actuators, etc.).
- **Specialized components**. These are specialized devices extended from base device class, these extended devices implement abstraction and decorator patterns in order conceptualize valves, sensors, and digital IO.
- **Platform resources**. This holds singleton instances of the devices and resources in the platform. All the valves, sensors, and digital IO instances are used with adaptive plug-in pattern at code level.

The following diagram shows this layer and its components.

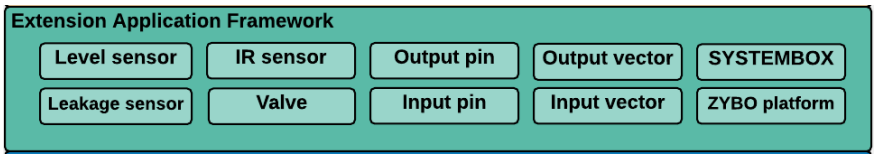

Figure 51: Extension application framework

The next diagrams show the UML hierarchy of the extended devices and the dependencies of this layer.

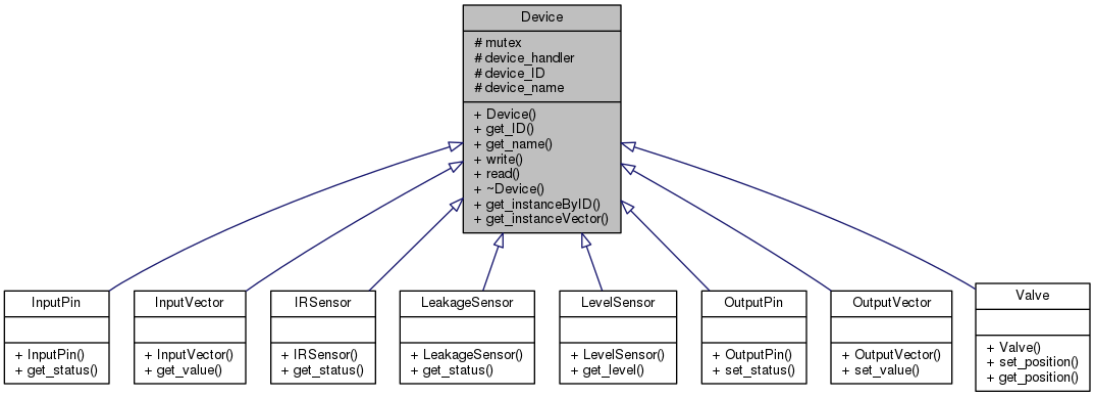

Figure 52: Inheritance diagram for Device class

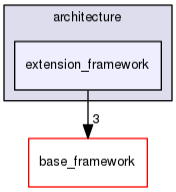

Figure 53: Dependency diagram for extension application framework

The extended devices can be seen as wrappers, the implementation is simple since the device methods are inherited from the `Device` class in the base application framework.

6.1 Platform specifications and definitions

The platform specification and definitions are stated in two source files:

- **zybo.hpp**, holds the kernel module paths, and
- **deviceid.hpp**, contains the device identities.

6.1.1 ZYBO_PLATFORM namespace

Located in *zybo.hpp*, `ZYBO_PLATFORM` contains the definition of the kernel modules and their paths in the file system. From these paths, it creates the singleton instances of the file handlers (`DeviceHandler`) that will be used by the devices as injection dependency pattern.

```
/*
 * zybo.hpp
 *
 *  Created on: Aug 31, 2017
 *      Author: Yarib Nevárez <yarib_007@hotmail.com> - root
 */

#ifndef ZYBO_HPP_
#define ZYBO_HPP_

#include "m_device.hpp"

namespace ZYBO_PLATFORM
{
    namespace DRIVER
    {
        const char * const PWM0    = "/proc/pwm_0";
        const char * const PWM1    = "/proc/pwm_1";
        const char * const CTRL    = "/proc/controller";
        const char * const ADC     = "/proc/adc";
        const char * const ZYBO    = "/proc/zybo";
        const char * const SERIAL0 = "/dev/ttyPS1";
    }

    namespace HANDLER
    {
        DeviceHandler PWM0(DRIVER::PWM0);
        DeviceHandler PWM1(DRIVER::PWM1);
        DeviceHandler CONTROLLER(DRIVER::CTRL);
        DeviceHandler ADC(DRIVER::ADC);
        DeviceHandler ZYBO(DRIVER::ZYBO);
        DeviceHandler SERIAL0(DRIVER::SERIAL0);
    }
}

#endif /* ZYBO_HPP_ */
```

It can be seen in the included files, that this component depends on the Device module (*m_device.hpp*) from the base application framework.

Note: The module header file *m_device.hpp* contains all the header files of the Device module from base application framework.

6.1.2 Device identities

The device identities are defined in the header file: deviceid.hpp. This is a C/C++ header file, it is shared for compilation between kernel and application space, so both have access to the identities. The device identity instances (`DeviceIdentity`) are created, and these are going to be given to the devices as dependency injection. These are the names that later on can be accessed by the commander component (from Device module), and from remote access as well.

```
#ifdef __cplusplus
extern "C" {
#endif

#ifndef KERNEL_MODULE
#include "m_device.hpp"
#endif

typedef enum
{
    CONTROLLER_DEVICE_ID_BEGIN = 0,
    IRSENSOR,
    FLUSHVALVE,
    DRAINVALVE,
    SHUTOFFVALVE,
    DRAINLOCAL,
    EMERGENCY,
    APPSELECTION,
    DRAINDELAY,
    VACUMGEN,
    DRAININDICATOR,
    LEAKINDICATOR,
    RELAY_0,
    RELAY_1,
    CONTROLLER_DEVICE_ID_END
} ControllerDeviceIDs;

typedef enum
{
    ADC_DEVICE_ID_BEGIN = 0,
    LEVELSENSOR_0,
    LEVELSENSOR_1,
    WASTETLEVEL,
    LEAKSENSOR_0,
    LEAKSENSOR_1,
    ADC_CHANNEL_5,
    ADC_CHANNEL_6,
    ADC_CHANNEL_7,
    ADC_CHANNEL_8,
    ADC_CHANNEL_9,
    ADC_CHANNEL_10,
    ADC_DEVICE_ID_END
} ADCDeviceIDs;

typedef enum
{
    PWM_DEVICE_ID_BEGIN = 0,
    PWM_0,
    PWM_1,
    PWM_DEVICE_ID_END
} PWMDeviceIDs;

typedef enum
{
    ZYBO_DEVICE_ID_BEGIN = 0,
    ZYBO_BUTTONS,
    ZYBO_SWITCHES,
    ZYBO_LEDS,
    ZYBO_DEVICE_ID_END
} ZYBODeviceIDs;

typedef enum
```

```
{
    SERIAL_DEVICE_ID_BEGIN = 0,
    SERIAL_0 = 0x80A55A01,
    SERIAL_DEVICE_ID_END
} SerialDeviceIDs;

DeviceIdentity id_LEVELSENSOR_0  = {LEVELSENSOR_0,  "LEVELSENSOR_0" };
DeviceIdentity id_LEVELSENSOR_1  = {LEVELSENSOR_1,  "LEVELSENSOR_1" };
DeviceIdentity id_WASTETLEVEL    = {WASTETLEVEL,    "WASTETLEVEL"   };
DeviceIdentity id_LEAKSENSOR_0   = {LEAKSENSOR_0,   "LEAKSENSOR_0"  };
DeviceIdentity id_LEAKSENSOR_1   = {LEAKSENSOR_1,   "LEAKSENSOR_1"  };
DeviceIdentity id_IRSENSOR       = {IRSENSOR,       "IRSENSOR"      };
DeviceIdentity id_FLUSHVALVE     = {FLUSHVALVE,     "FLUSHVALVE"    };
DeviceIdentity id_DRAINVALVE     = {DRAINVALVE,     "DRAINVALVE"    };
DeviceIdentity id_SHUTOFFVALVE   = {SHUTOFFVALVE,   "SHUTOFFVALVE"  };
DeviceIdentity id_DRAINLOCAL     = {DRAINLOCAL,     "DRAINLOCAL"    };
DeviceIdentity id_EMERGENCY      = {EMERGENCY,      "EMERGENCY"     };
DeviceIdentity id_APPSELECTION   = {APPSELECTION,   "APPSELECTION"  };
DeviceIdentity id_DRAINDELAY     = {DRAINDELAY,     "DRAINDELAY"    };
DeviceIdentity id_VACUMGEN       = {VACUMGEN,       "VACUMGEN"      };
DeviceIdentity id_DRAININDICATOR = {DRAININDICATOR, "DRAININDICATOR"};
DeviceIdentity id_LEAKINDICATOR  = {LEAKINDICATOR,  "LEAKINDICATOR" };
DeviceIdentity id_PWM_0          = {PWM_0,          "PWM_0"         };
DeviceIdentity id_PWM_1          = {PWM_1,          "PWM_1"         };
DeviceIdentity id_RELAY_0        = {RELAY_0,        "RELAY_0"       };
DeviceIdentity id_RELAY_1        = {RELAY_1,        "RELAY_1"       };
DeviceIdentity id_ADC_CHANNEL_5  = {ADC_CHANNEL_5,  "ADC_CHANNEL_5" };
DeviceIdentity id_ADC_CHANNEL_6  = {ADC_CHANNEL_6,  "ADC_CHANNEL_6" };
DeviceIdentity id_ADC_CHANNEL_7  = {ADC_CHANNEL_7,  "ADC_CHANNEL_7" };
DeviceIdentity id_ADC_CHANNEL_8  = {ADC_CHANNEL_8,  "ADC_CHANNEL_8" };
DeviceIdentity id_ADC_CHANNEL_9  = {ADC_CHANNEL_9,  "ADC_CHANNEL_9" };
DeviceIdentity id_ADC_CHANNEL_10 = {ADC_CHANNEL_10, "ADC_CHANNEL_10"};
DeviceIdentity id_ZYBO_BUTTONS   = {ZYBO_BUTTONS,   "ZYBO_BUTTONS"  };
DeviceIdentity id_ZYBO_SWITCHES  = {ZYBO_SWITCHES,  "ZYBO_SWITCHES" };
DeviceIdentity id_ZYBO_LEDS      = {ZYBO_LEDS,      "ZYBO_LEDS"     };
DeviceIdentity id_SERIAL_0       = {SERIAL_0,       "SERIAL_0"      };
#ifdef __cplusplus
}
#endif

#endif /* DEVICEID_HPP_ */
```

6.2 Specialized components

This is done by extensions of the base device in order to implement specialized devices. These extensions implement abstraction and decorator patterns. This approach provides the conceptualization of valves, sensors, and input-output pins. All of these class extensions are minimalistic, they are defined in *devices.hpp* and implemented in *devices.hpp*.

6.2.1 LevelSensor class

Instances of this class read analog signals corresponding to the water level. The injected device handler defines the kernel module that is used.

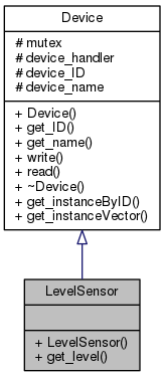

Figure 54: Inheritance diagram for LevelSensor class

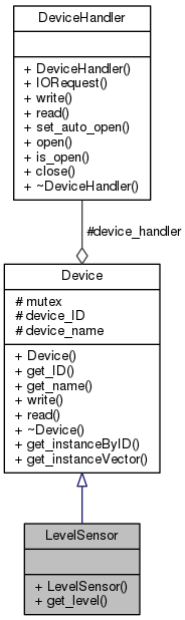

Figure 55: Collaboration diagram for LevelSensor class

Specializations:

- Public Types

```
enum Status { LOW = 0, FULL = 1 }
```

- Public Member Functions

```
LevelSensor (DeviceHandler *device_handler, DeviceIdentity *identity)
Status get_level (void)
```

This class is implemented as a template, it will be customized according to business requirements.

Class declaration:

```
class LevelSensor : public Device
{
public:
    LevelSensor(DeviceHandler *, DeviceIdentity *);

    typedef enum
    {
        LOW = 0,
        FULL = 1
    } Status;

    Status get_level(void);
};
```

Class implementation:

```
LevelSensor::LevelSensor(DeviceHandler * device_handler, DeviceIdentity * identity):
Device(device_handler, identity)
{}

LevelSensor::Status LevelSensor::get_level(void)
{
    Status status;
    uint32_t value;

    read((uint32_t *) & value);

    if (value > 0x7FF) // threshold value
```

```
        status = FULL;
    else
        status = LOW;

    return status;
}
```

6.2.2 LeakageSensor class

Instances of this class read analog signals corresponding to the water leakage. The injected device handler defines the kernel module that is used.

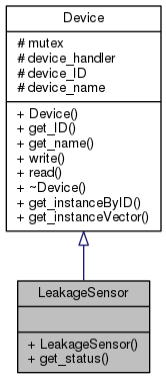

Figure 56: Inheritance diagram for LeakageSensor class

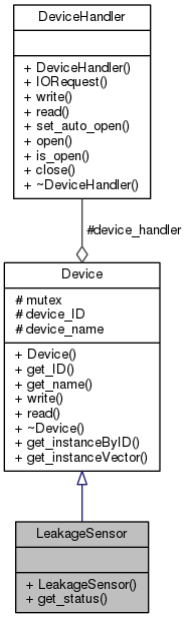

Figure 57: Collaboration diagram for LeakageSensor class

Specializations:

- Public Types

```
enum Status { NO_LEAKING = 0, LEAKING = 1 }
```

- Public Member Functions

```
LeakageSensor (DeviceHandler *device_handler, DeviceIdentity *identity)
Status get_status (void)
```

Class declaration:

```
class LeakageSensor : public Device
{
public:
    LeakageSensor(DeviceHandler *, DeviceIdentity *);

    typedef enum
    {
        NO_LEAKING = 0,
        LEAKING    = 1
    } Status;

    Status get_status(void);
};
```

Class implementation:

```
LeakageSensor::LeakageSensor(DeviceHandler * device_handler, DeviceIdentity * identity):
Device(device_handler, identity)
{}

LeakageSensor::Status LeakageSensor::get_status(void)
{
    Status status;
    uint32_t value;

    read((uint32_t *) & value);

    if (value>0x7FF) // threshold value
        status = LEAKING;
    else
        status = NO_LEAKING;

    return status;
}
```

6.2.3 IRSensor class

Instances of this class read digital signals from an external IR sensor module. The injected device handler defines the kernel module that is used.

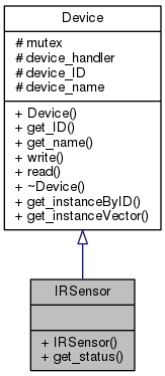

Figure 58: Inheritance diagram for IRSensor class

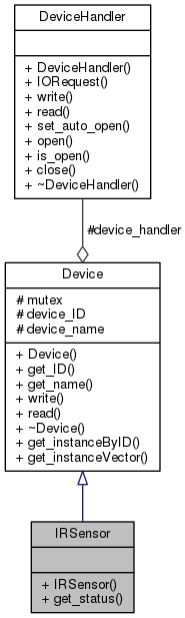

Figure 59: Collaboration diagram for IRSensor class

Specializations:

- Public Types

```
enum Status { ABSENCE = 0, PRESENCE = 1 }
```

- Public Member Functions

```
IRSensor (DeviceHandler *device_handler, DeviceIdentity *identity)
Status get_status (void)
```

Class declaration:

```
class IRSensor : public Device
{
public:
    IRSensor(DeviceHandler *, DeviceIdentity *);

    typedef enum
    {
        ABSENCE  = 0,
        PRESENCE = 1
    } Status;

    Status get_status(void);
};
```

Class implementation:

```
IRSensor::IRSensor(DeviceHandler * device_handler, DeviceIdentity * identity):
Device(device_handler, identity)
{}

IRSensor::Status IRSensor::get_status(void)
{
    Status status;
    read((uint32_t *) & status);
    return status;
}
```

6.2.4 Valve class

Instances of this class read from and write to valve actuators, these are digital signals. The injected device handler defines the kernel module that is used.

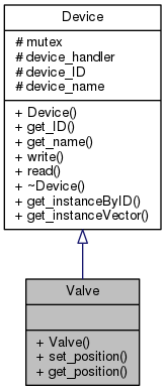

Figure 60: Inheritance diagram for Valve class

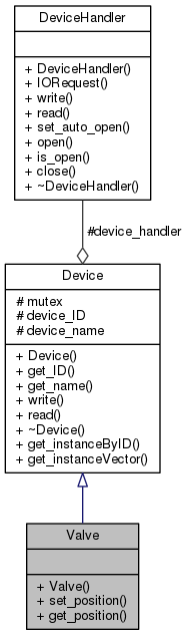

Figure 61: Collaboration diagram for Valve class

Specializations:

- Public Types

```
enum Position { CLOSE = 0, OPEN = 1 }
```

- Public Member Functions

```
Valve (DeviceHandler *device_handler, DeviceIdentity *identity)
bool set_position (Position position)
```

```
Position get_position (void)
```

Class declaration:

```
class Valve : public Device
{
public:
    Valve(DeviceHandler *, DeviceIdentity *);

    typedef enum
    {
        CLOSE = 0,
        OPEN  = 1
    } Position;

    bool     set_position(Position position);
    Position get_position(void);
};
```

Class implementation:

```
Valve::Valve(DeviceHandler * device_handler, DeviceIdentity * identity):
Device(device_handler, identity)
{}

bool Valve::set_position(Position position)
{
    return write((int) position);
}

Valve::Position Valve::get_position(void)
{
    Position position;
    read((uint32_t *) & position);
    return position;
}
```

6.2.5 OutputPin class

Instances of this class write digital signals (1-bit). The injected device handler defines the kernel module that is used.

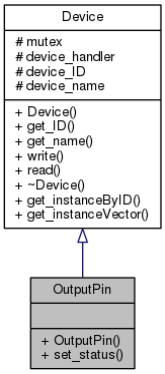

Figure 62: Inheritance diagram for OutputPin class

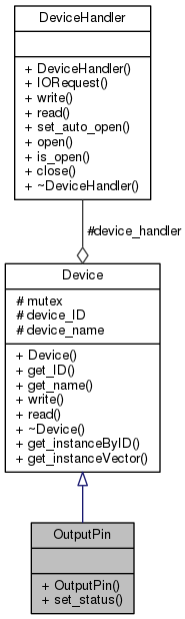

Figure 63: Collaboration diagram for OutputPin class

Specializations:

- Public Types

```
enum Status { OFF = 0, ON = 1 }
```

- Public Member Functions

```
OutputPin (DeviceHandler *device_handler, DeviceIdentity *identity)
bool set_status (Status status)
```

Class declaration:

```
class OutputPin: public Device
{
public:
    OutputPin(DeviceHandler *, DeviceIdentity *);

    typedef enum
    {
        OFF = 0,
        ON  = 1
    } Status;

    bool set_status(Status status);
};
```

Class implementation:

```
OutputPin::OutputPin(DeviceHandler * device_handler, DeviceIdentity * identity):
Device(device_handler, identity)
{}

bool OutputPin::set_status(Status status)
{
    return write((int) status);
}
```

6.2.6 InputPin class

Instances of this class read digital signals (1-bit). The injected device handler defines the kernel module that is used.

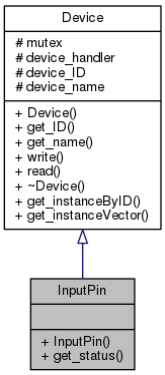

Figure 64: Inheritance diagram for InputPin class

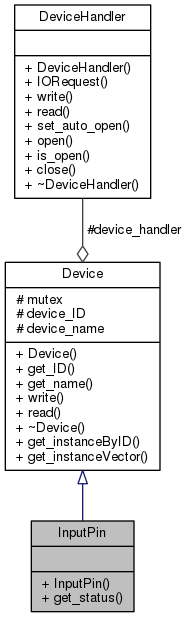

Figure 65: Collaboration diagram for InputPin class

Specializations:

- Public Types

```
enum Status { OFF = 0, ON = 1 }
```

- Public Member Functions

```
InputPin (DeviceHandler *device_handler, DeviceIdentity *identity)
Status get_status (void)
```

Class declaration:

```
class InputPin : public Device
{
public:
    InputPin(DeviceHandler *, DeviceIdentity *);

    typedef enum
    {
        OFF = 0,
        ON  = 1
    } Status;

    Status get_status(void);
};
```

Class implementation:

```
InputPin::InputPin(DeviceHandler * device_handler, DeviceIdentity * identity):
Device(device_handler, identity)
{}

InputPin::Status InputPin::get_status(void)
{
    Status status;
    read((uint32_t *) & status);
```

```
    return status;
}
```

6.2.7 OutputVector class

Instances of this class write digital vectors. The injected device handler defines the kernel module that is used.

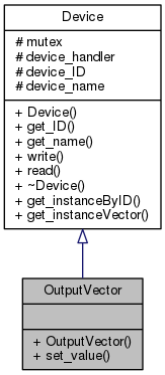

Figure 66: Inheritance diagram for OutputVector class

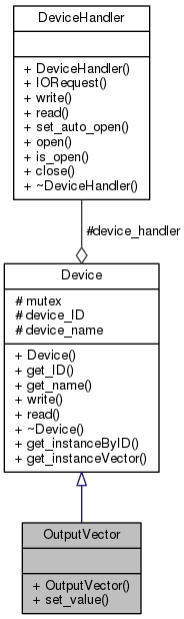

Figure 67: Collaboration diagram for OutputVector class

Specializations:

- Public Member Functions

```
OutputVector (DeviceHandler *device_handler, DeviceIdentity *identity)
bool set_value (int value)
```

Class declaration:

```
class OutputVector : public Device
{
public:
    OutputVector(DeviceHandler *, DeviceIdentity *);

    bool set_value(int value);
};
```

Class implementation:

```
OutputVector::OutputVector(DeviceHandler * device_handler, DeviceIdentity * identity):
Device(device_handler, identity)
{}

bool OutputVector::set_value(int value)
{
    return write(value);
}
```

6.2.8 InputVector class

Instances of this class read digital vectors. The injected device handler defines the kernel module that is used.

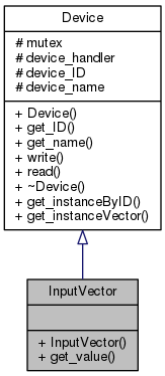

Figure 68: Inheritance diagram for InputVector class

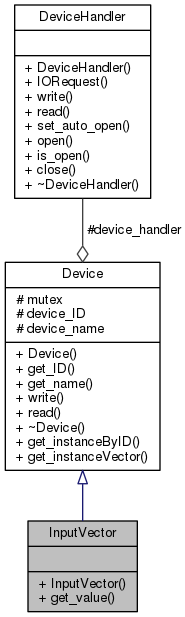

Figure 69: Collaboration diagram for InputVector class

Specializations:

- Public Member Functions

```
InputVector (DeviceHandler *device_handler, DeviceIdentity *identity)
int get_value (void)
```

Class declaration:

```
class InputVector : public Device
{
public:
    InputVector(DeviceHandler *, DeviceIdentity *);

    int get_value(void);
};
```

Class implementation:

```
InputVector::InputVector(DeviceHandler * device_handler, DeviceIdentity * identity):
Device(device_handler, identity)
{}

int InputVector::get_value(void)
{
    int status;
    read((uint32_t *) & status);
    return status;
}
```

6.3 Platform resources.

The singleton instances of the devices and resources in the platform are located within the **SYSTEMBOX** namespace, in the header file *systembox.hpp*.

All the resources available in the system are located in **SYSTEMBOX**. All singleton instances of sensors, valves and digital IO are located in this name space.

As a plug-in design pattern, applications that are willing to utilize any of these singleton resources have to use the **SYSTEMBOX** namespace.

6.3.1 SYSTEMBOX Namespace

This namespace creates and holds the instances of sensors, valves, and digital IO. These are constructed during the loading of the application. The dependency injections for the creation of devices are taken either from **ZYBO_PLATFORM::HANDLER**, or **ZEDBOARD::HANDLER**, or **FLORIDA::HANDLER**. This depends on which pre-compilation variable is defined in the project: **ZYBO_HARDWARE**, or **ZEDBOARD_HARDWARE**, or **FLORIDA_HARDWARE**. This adaptive patterns allow to build the application for a specific hardware platform.

```
/*
 * systembox.hpp
 *
 *  Created on: Aug 29, 2017
 *      Author: Yarib Nevárez <yarib_007@hotmail.com> - root
 */

#ifndef SYSTEMBOX_HPP_
#define SYSTEMBOX_HPP_

#include "devices.hpp"
#include "deviceid.hpp"
#include "zybo.hpp"

#ifdef ZYBO_HARDWARE
    using namespace ZYBO_PLATFORM::HANDLER;
#elif ZEDBOARD_HARDWARE
    using namespace ZEDBOARD::HANDLER;
#elif FLORIDA_HARDWARE
    using namespace FLORIDA::HANDLER;
#endif

namespace SYSTEMBOX
{
    LevelSensor   chamberBottomSensor (&ADC,         &id_LEVELSENSOR_0  );
    LevelSensor   chamberTopSensor    (&ADC,         &id_LEVELSENSOR_1  );
    LevelSensor   wasteTankSensor     (&ADC,         &id_WASTETLEVEL    );
    LeakageSensor leakageSensor_0     (&ADC,         &id_LEAKSENSOR_0   );
    LeakageSensor leakageSensor_1     (&ADC,         &id_LEAKSENSOR_1   );
    IRSensor      irSensor            (&CONTROLLER, &id_IRSENSOR       );
    Valve         flushValve          (&CONTROLLER, &id_FLUSHVALVE     );
    Valve         drainValve          (&CONTROLLER, &id_DRAINVALVE     );
    Valve         shutOffValve        (&CONTROLLER, &id_SHUTOFFVALVE   );
    InputPin      drainLocalReq       (&CONTROLLER, &id_DRAINLOCAL     );
    InputVector   emergencySwitch     (&CONTROLLER, &id_EMERGENCY      );
    InputVector   appSelection        (&CONTROLLER, &id_APPSELECTION   );
    InputVector   drainDelaySelection (&CONTROLLER, &id_DRAINDELAY     );
    OutputPin     vacumGenerator      (&CONTROLLER, &id_VACUMGEN       );
    OutputPin     drainIndicator      (&CONTROLLER, &id_DRAININDICATOR );
    OutputPin     leakageIndicator    (&CONTROLLER, &id_LEAKINDICATOR  );
    OutputVector  pwm_0               (&PWM0,        &id_PWM_0         );
    OutputVector  pwm_1               (&PWM1,        &id_PWM_1         );
    OutputPin     relay_0             (&CONTROLLER, &id_RELAY_0        );
    OutputPin     relay_1             (&CONTROLLER, &id_RELAY_1        );
    InputVector   adc_channel_5       (&ADC,         &id_ADC_CHANNEL_5  );
#ifdef ADC_CHANNELS_6_TO_10
    InputVector   adc_channel_6       (&ADC,         &id_ADC_CHANNEL_6  );
    InputVector   adc_channel_7       (&ADC,         &id_ADC_CHANNEL_7  );
    InputVector   adc_channel_8       (&ADC,         &id_ADC_CHANNEL_8  );
    InputVector   adc_channel_9       (&ADC,         &id_ADC_CHANNEL_9  );
```

```
    InputVector   adc_channel_10      (&ADC,        &id_ADC_CHANNEL_10 );
#endif
    InputVector   zybo_buttons        (&ZYBO,       &id_ZYBO_BUTTONS   );
    InputVector   zybo_switches       (&ZYBO,       &id_ZYBO_SWITCHES  );
    OutputVector  zybo_leds           (&ZYBO,       &id_ZYBO_LEDS      );
    OutputVector  serial_0            (&SERIAL0,    &id_SERIAL_0       );
};

#endif /* SYSTEMBOX_HPP_ */
```

Chapter 7. **Applications**

The Application layer is the place for customer application where the business logic is implemented. The applications are robust and featured by the advantage of reusing components from the entire architecture.

Business logic: Business logic or domain logic is the part of the software that encodes the real-world business rules that determine the processes execution and data handling.

This layer contains two sample applications to test the software architecture, these applications serve as skeleton for future customer application. However, the implementation of any business logic or any customer application does not fall within the scope of this thesis.

Sample applications:

- **Control application.** This application is a sample of a customer application, it implements a multi-threading hardware access, background processing as Linux service, internet monitoring-control, and storage.
- **System tool.** This application is a sample of a testing tool, it implements internet monitoring-control, remote and local command line interface, local hardware monitoring, serial communication, and HID (USB Joystick) handling.

The following figures show the design of the application layer.

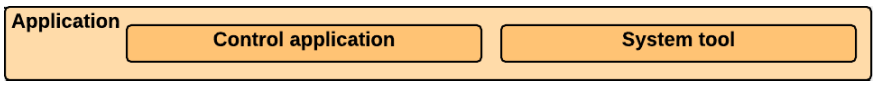

Figure 70: Application Layer

This layer has the dependency of both application frameworks, base and extended, as shown in the following figure in the right side.

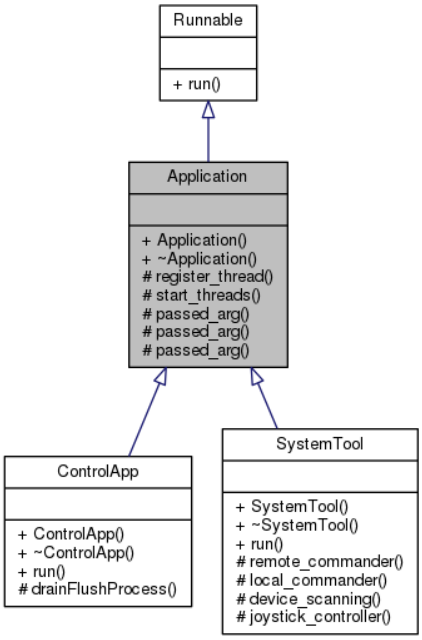

Figure 71: Inheritance diagram for Application class

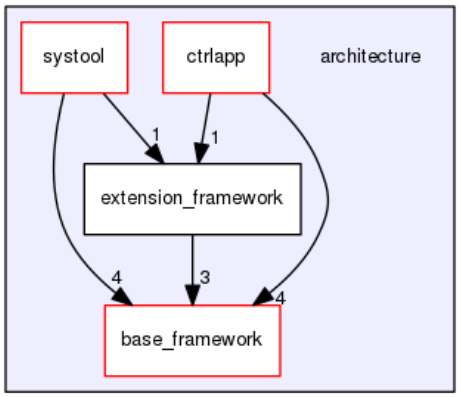

Figure 72: Dependency diagram for Application layer

The following diagram reveals the active object patterns and their inheritance from `Runnable` class.

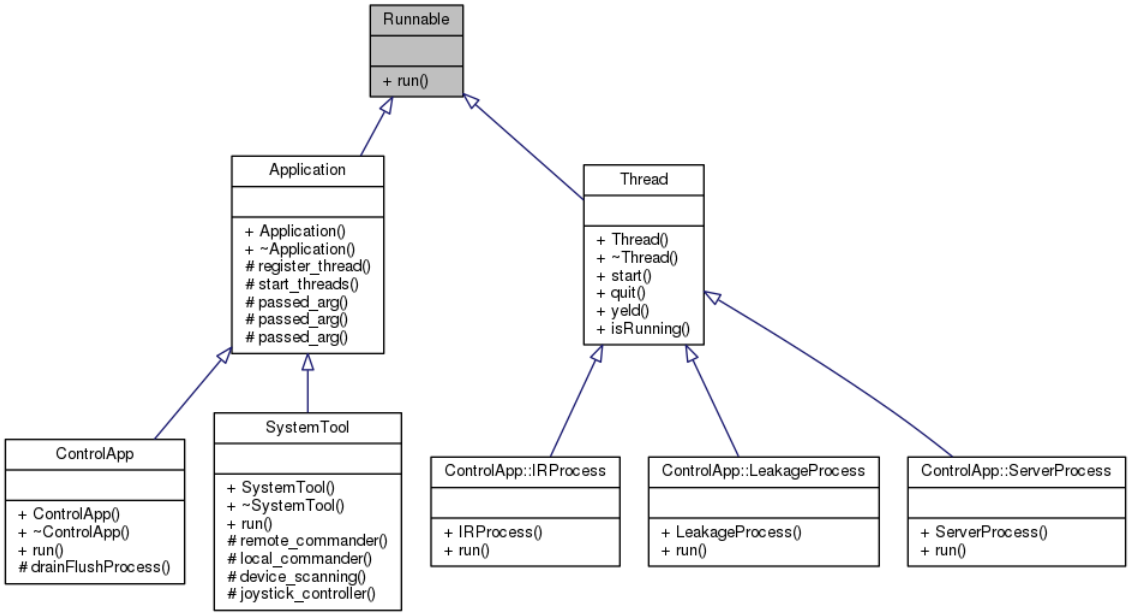

Figure 73: Inheritance diagram for Runnable class in Application layer

7.1 Control application

This application is a sample of a customer application for controlling the water system in an aircraft.

This application is installed in the system as a Linux service, therefore it is coded to be a daemon background application.

The application runs four processes or tasks, each one in different thread. Each process is able to support its own business logic.

Processes:

1. Water drain and flush
2. Cabin infrared detection
3. Water leakage detection
4. TCP/IP Server

The control application is implemented in **`ControlApp`** class that extends form **`Application`** class. This application creates three active objects (threads) to decouple the individual execution of the processes, these are encapsulated as nested classes.

7.1.1 ControlApp class

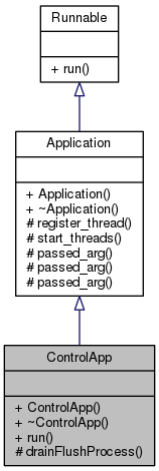

Figure 74: Inheritance diagram for ControlApp class

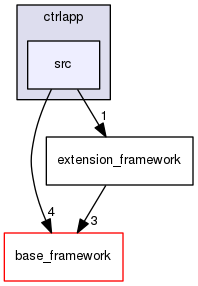

Figure 75: Dependency diagram for ControlApp class

Nested classes

```
class IRProcess
class LeakageProcess
class ServerProcess
```

Public Member Functions

```
ControlApp (int argc=0, char *argv[]=NULL)
~ControlApp ()
virtual int run (void)
```

Protected Member Functions

```
void drainFlushProcess (void)
```

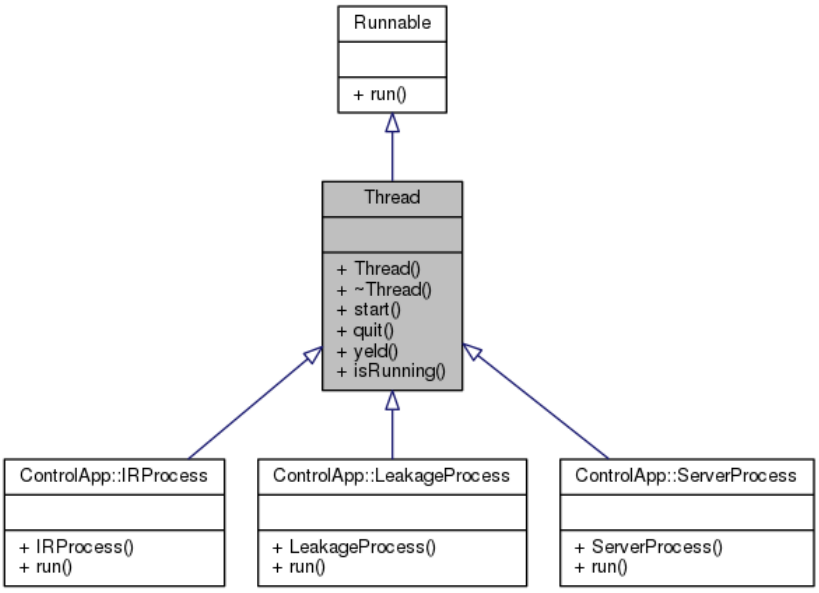

Figure 76: Nested classes in ControlApp class

Class declaration:

```
#include "m_application.hpp"
```

```
class ControlApp: public Application
{
public:
    ControlApp(int argc = 0, char * argv[] = NULL);
    ~ControlApp();

    virtual int run(void);

protected:

    void drainFlushProcess(void);

    class IRProcess: public Thread
    {
    public:
        IRProcess()
        {}
        virtual int run(void);
    };

    class LeakageProcess: public Thread
    {
    public:
        LeakageProcess()
        {}
        virtual int run(void);
    };

    class ServerProcess: public Thread
    {
    public:
        ServerProcess()
        {}
        virtual int run(void);
    };
};
```

As an active object, the application entry point is the **run** function (inherited from **Runnable** class), it is called by the main function of the program.

```
#include "ctrlapp.hpp"

int main(int argc, char * argv[])
{
    ControlApp app(argc, argv);

    return app.run();
}
```

Before including the necessary modules from the architecture, it is defined the usage of ZYBO hardware as a pre-compilation variable, and the resources of the platform (sensors, actuators, etc.) are added to the application by using the **SYSTEMBOX** namespace.

```
#define ZYBO_HARDWARE
#include "m_application.hpp"
#include "m_device.hpp"
#include "m_network.hpp"
#include "systembox.hpp"

using namespace SYSTEMBOX;
```

The application gets the arguments from the system caller, these arguments are handled in the inherited methods of **Application** class. Inside the class constructor, the arguments are used to decide which active object are going to be instantiated and registered in the application.

```
ControlApp::ControlApp(int argc, char * argv[]):
Application(argc, argv)
```

```
{
    if (0 < passed_arg("-ir"))
        register_thread(new IRProcess());

    if (0 < passed_arg("-leakage"))
        register_thread(new LeakageProcess());

    if (0 < passed_arg("-server"))
        register_thread(new ServerProcess());
}
...
```

In the `run` function, it is evaluated if the application will become a daemon to run in background, afterwards it triggers the registered active objects or threads; finally, it is called the function `drainFlushProcess`.

```
int ControlApp::run(void)
{
    if (0 < passed_arg("-daemon"))
    {
        Daemon::become(0);
    }

    start_threads();

    drainFlushProcess();

    return 0;
}
```

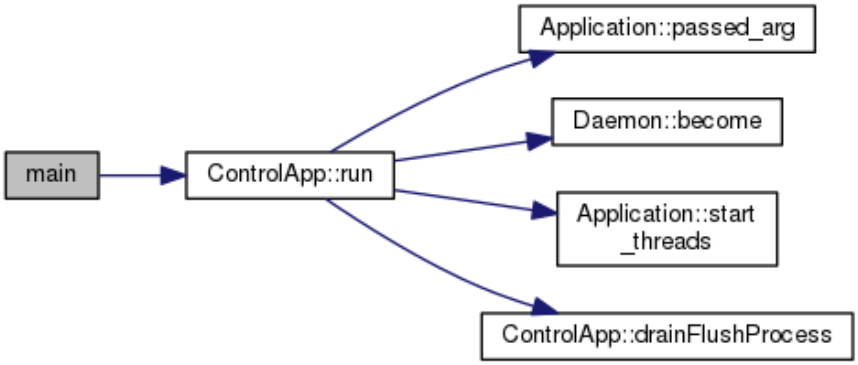

Figure 77: Call diagram for main function in Control application

7.1.2 Water drain and flush

Business logic:

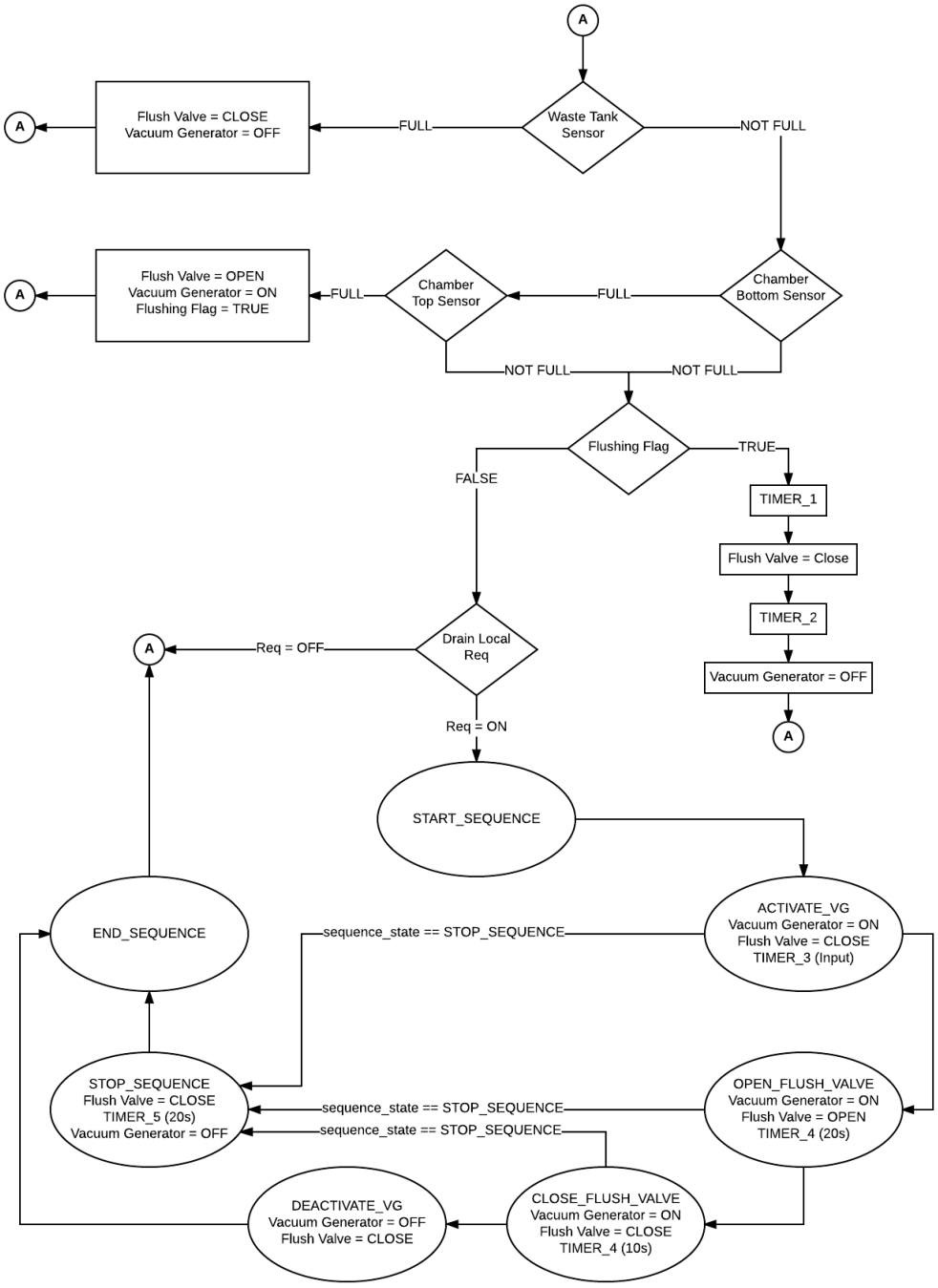

Figure 78: Flow diagram for water drain-flush

Code implementation:

```
void ControlApp::drainFlushProcess()
{
    typedef enum
    {
        START_SEQUENCE,
        ACTIVATE_VG,
        OPEN_FLUSH_VALVE,
        CLOSE_FLUSH_VALVE,
        DEACTIVATE_VG,
        STOP_SEQUENCE,
        END_SEQUENCE
    } DrainSequenceState;

    DrainSequenceState sequence_state;
    bool flushingFlag;
```

```
    zybo_leds.set_value(1);

    for (;;)
    {
////////////////////////////// POWER OFF - BUTTON ////////////////////////////
        if (zybo_buttons.get_value() & 0x01)
        {
            system("poweroff");
        }
////////////////////////////// BUSINESS LOGIC ////////////////////////////////
        if (LevelSensor::FULL != wasteTankSensor.get_level())
        {
            if ((LevelSensor::FULL == chamberTopSensor.get_level()) &&
(LevelSensor::FULL == chamberBottomSensor.get_level()))
            {
                vacumGenerator.set_status(OutputPin::ON);
                flushValve.set_position(Valve::OPEN);
                flushingFlag = true;
            }
            else if (flushingFlag)
            {
                sleep(1);
                flushValve.set_position(Valve::CLOSE);

                sleep(5);
                vacumGenerator.set_status(OutputPin::OFF);

                flushingFlag = false;
            }
            else if (InputPin::ON == drainLocalReq.get_status())
            {
                sequence_state = START_SEQUENCE;

                while (sequence_state != END_SEQUENCE)
                {
                    switch (sequence_state)
                    {
                    case START_SEQUENCE:
                    case ACTIVATE_VG:
                        vacumGenerator.set_status(OutputPin::ON);
                        sleep(drainDelaySelection.get_value());
                        if (sequence_state != STOP_SEQUENCE)
                        {
                            sequence_state = OPEN_FLUSH_VALVE;
                        }
                        break;

                    case OPEN_FLUSH_VALVE:
                        flushValve.set_position(Valve::OPEN);
                        sleep(20);
                        if (sequence_state != STOP_SEQUENCE)
                        {
                            sequence_state = CLOSE_FLUSH_VALVE;
                        }
                        break;

                    case CLOSE_FLUSH_VALVE:
                        flushValve.set_position(Valve::CLOSE);
                        sleep(10);
                        if (sequence_state != STOP_SEQUENCE)
                        {
                            sequence_state = DEACTIVATE_VG;
                        }
                        break;

                    case DEACTIVATE_VG:
                        vacumGenerator.set_status(OutputPin::OFF);
                        sequence_state = END_SEQUENCE;
                        break;
                    case STOP_SEQUENCE:
                        flushValve.set_position(Valve::CLOSE);
                        sleep(20);
                        vacumGenerator.set_status(OutputPin::OFF);
                        sequence_state = END_SEQUENCE;
                    case END_SEQUENCE:
                        break;
                    }
                }
            }
        }
```

```
        else
        {
            flushValve.set_position(Valve::CLOSE);
            vacumGenerator.set_status(OutputPin::OFF);
        }
    }
}
```

This code belongs to the `ControlApplication` class, therefor this is running in the main thread of the program. It does not need the support of an additional active object.

7.1.3 Cabin infrared detection

Business logic:

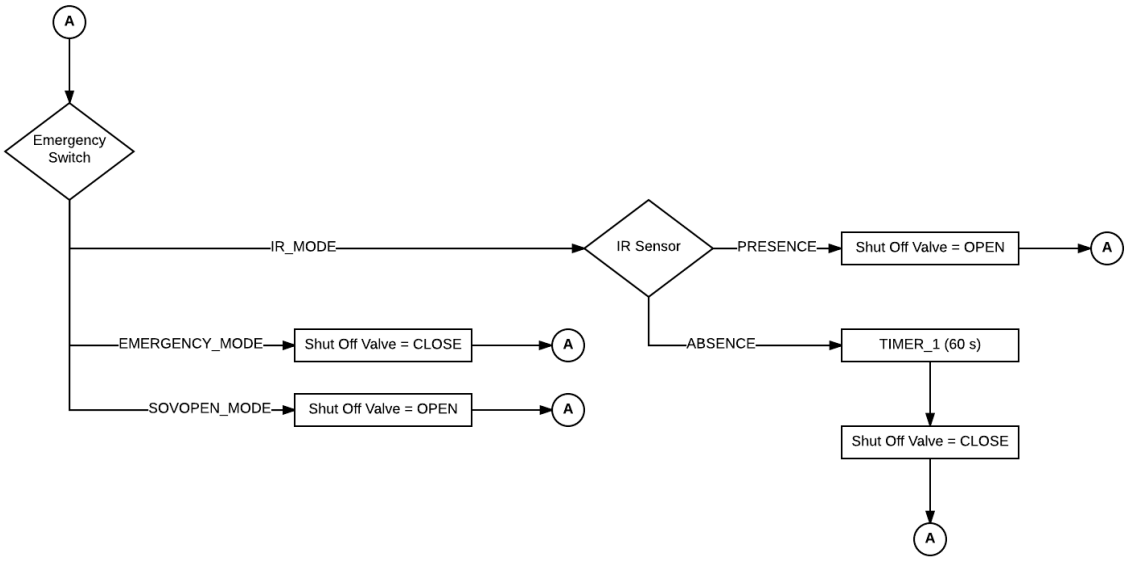

Figure 79: Flow diagram for cabin infrared detection

7.1.3.1 ControlApp::IRProcess class

In order to decouple the execution of this process, the code of the business logic is encapsulated in an active object (extended from `Thread` class).

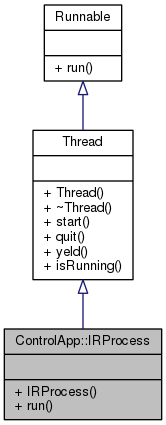

Figure 80: Inheritance diagram for ControlApp::IRProcess class

Public Member Functions

```
IRProcess ()
virtual int run (void)
```

Nested class declaration:

```
    class IRProcess: public Thread
    {
    public:
        IRProcess()
        {}
        virtual int run(void);
    };
```

Nested class implementation:

```
int ControlApp::IRProcess::run(void)
{
    typedef enum
    {
        IR_MODE = 0,
        EMERGENCY,
        SOV_OPEN
    } ESwitchState;

    ESwitchState status;
    const unsigned int ABSENCE_TIMEOUT = 60;
    unsigned int timer = 0;

    for (;;)
    {
        status = (ESwitchState) emergencySwitch.get_value();
        switch(status)
        {
        case IR_MODE:
            switch (irSensor.get_status())
            {
            case IRSensor::PRESENCE:
                shutOffValve.set_position(Valve::OPEN);
                timer = ABSENCE_TIMEOUT;
                break;
            case IRSensor::ABSENCE:
                if (timer == 0)
                {
                    shutOffValve.set_position(Valve::CLOSE);
                }
                else
                {
                    shutOffValve.set_position(Valve::OPEN);
                    sleep(1);
                    timer --;
                }
                break;
            default:;
            }
            break;
        case EMERGENCY:
            shutOffValve.set_position(Valve::CLOSE);
            timer = 0;
            break;
        case SOV_OPEN:
            shutOffValve.set_position(Valve::OPEN);
            timer = 0;
            break;
        default:;
            timer = 0;
        }
        yeld();
    }
    return 0;
}
```

7.1.4 Water leakage detection

Business logic:

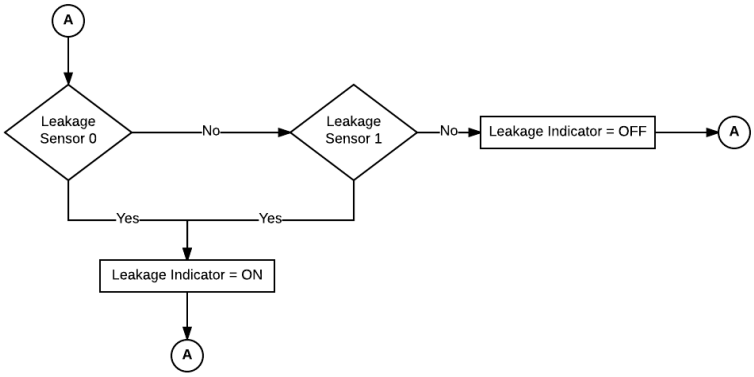

Figure 81: Flow diagram for water leakage detection

7.1.4.1 ControlApp::LeakageProcess class

In order to decouple the execution of this process, the code of the business logic is encapsulated in an active object (extended from **`Thread`** class).

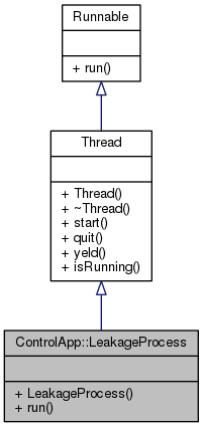

Figure 82: Inheritance diagram for ControlApp::LeakageProcess class

Public Member Functions

```
LeakageProcess ()
virtual int run (void)
```

Nested class declaration:

```
    class LeakageProcess: public Thread
    {
    public:
        LeakageProcess()
        {}
        virtual int run(void);
    };
```

Nested class implementation:

```
int ControlApp::LeakageProcess::run(void)
{
    for (;;)
    {
        if (LeakageSensor::LEAKING == leakageSensor_0.get_status()
```

```
         || LeakageSensor::LEAKING == leakageSensor_1.get_status())
        {
            leakageIndicator.set_status(OutputPin::ON);
        }
        else
        {
            leakageIndicator.set_status(OutputPin::OFF);
        }
        yeld();
    }
    return 0;
}
```

7.1.5 TCP/IP Server

This process implements a synchronous client-server loop. The remote client sends messages as commands, these commands are executed by the commander class in the base application framework. All received commands are stored in the file system as network logging.

7.1.5.1 ControlApp::ServerProcess class

In order to decouple the execution of this process, the code of the business logic is encapsulated in an active object (extended from `Thread` class).

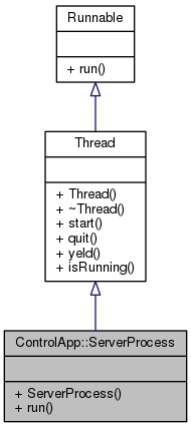

Figure 83: Inheritance diagram for ControlApp::ServerProcess class

Public Member Functions

```
ServerProcess ()
virtual int run (void)
```

Nested class declaration:

```
    class ServerProcess: public Thread
    {
    public:
        ServerProcess()
        {}
        virtual int run(void);
    };
```

Nested class implementation:

```
int ControlApp::ServerProcess::run(void)
{
```

```
    TcpServer * server;
    TcpSocket * client;
    std::string message;
    std::string answer;
    bool exit_flag;

    server = new TcpServer(23);
    if (server != NULL)
    {
        if (server->prepare() != -1)
        {
            for (;;)
            {
                client = server->accept_connection(); // Waiting for connection

                if (client != NULL)
                {
                    do
                    {
                        if (client->receive_buffer(message) > 0)
                        {
                            exit_flag = Commander::execute(message, answer);
                            client->send_buffer(answer);
                            // String sent to the system: echo REMOTE_MESSAGE >>
/root/net_logs.txt
                            system((message.insert(0, "echo ") + " >>
/root/net_logs.txt").c_str());
                        }
                        else
                        {
                            exit_flag = true;
                        }
                        yeld();
                    }
                    while(!exit_flag);

                    delete client;
                }
                yeld();
            }
        }
        delete server;
    }
    perror("\nExit");
    return 0;
}
```

Inside the core of the loop, after receiving a message from the client, this is executed, and then it is logged in the file: */root/net_logs.txt*.

```
exit_flag = Commander::execute(message, answer);
client->send_buffer(answer);
//String sent to the system: echo REMOTE_MESSAGE >> /root/net_logs.txt
system((message.insert(0, "echo ") + " >> /root/net_logs.txt").c_str());
```

This process has access to monitor and control all devices in the system, since all the devices are accessed by the device vector from the `Device` class. This is executed in `Commander::execute`.

With polymorphism, all devices are interfaced with `Device::read` and `Device::write` functions from the base class.

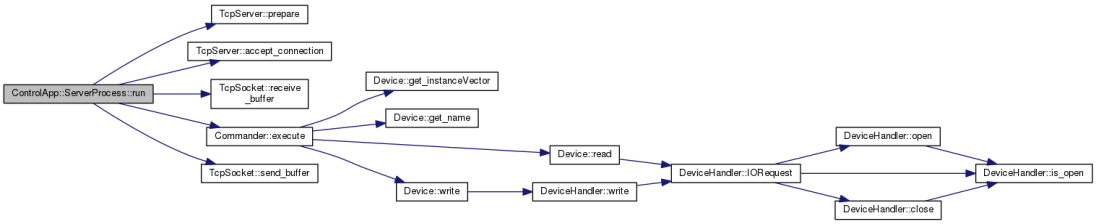

Figure 84: Call diagram for ControlApp::ServerProcess::run

The ZYBO development board provides Ethernet as the network interface.

7.2 System tool

This application is developed using the software architecture, this sample would be used for testing of sensors, actuators and pheripherals of "The cabin convenient controller unit".

This executable is installed in the file system as a Linux application.

The application runs one processes at a time chosen by an option menu.

Options:

1. Network command interface
2. Command interface
3. Device scanning
4. Joystick controller

7.2.1 SystemTool class

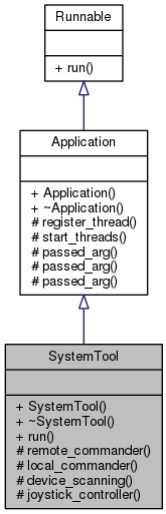

Figure 85: Inheritance diagram for SystemTool class

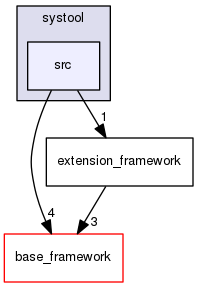

Figure 86: Dependency diagram for SystemTool class

Public Member Functions

```
SystemTool (int argc=0, char *argv[]=NULL)
~SystemTool ()
virtual int run (void)
```

Protected Member Functions

```
void remote_commander (uint16_t server_port=2323)
void local_commander (void)
void device_scanning (void)
void joystick_controller (void)
```

Class declaration:

```
#include "m_application.hpp"
#include "m_network.hpp"

class SystemTool: public Application
{
public:
    SystemTool(int argc = 0, char * argv[] = NULL);
    ~SystemTool();

    virtual int run(void);

protected:
    void remote_commander(uint16_t server_port = 2323);
    void local_commander(void);
    void device_scanning(void);
    void joystick_controller(void);
};
```

The entry point of the application:

```
#include "systool.hpp"

int main(int argc, char * argv[])
{
    SystemTool app(argc, argv);

    return app.run();
}
```

The `run` method displays the options menu in the terminal, it gets the selection from the user, and calls the corresponding method. At the beginning, it is checked the execution arguments of the application, the first argument is considered to select the option from the menu; if there is a second argument, it is considered as a port number for TCP/IP communication services, if this matchs with the first argument.

```
int SystemTool::run(void)
{
    int tcp_port = 0;
    char op;

    system("clear");

    if (passed_arg() >= 2)
    {
        op = passed_arg(1)[0];

        if (passed_arg() > 2)
        {
            tcp_port = strtoul(passed_arg(2).c_str(), NULL, 0);
        }
    }
    else
    {
        std::cout << "\n***** System Tool *****";
        std::cout << "\n\nOptions:";
        std::cout << "\n 1   - Network command interface";
        std::cout << "\n 2   - Command interface";
        std::cout << "\n 3   - Device scanning";
        std::cout << "\n 4   - Joystick controller";
        std::cout << "\n XXX - Exit";
        std::cout << "\n\nSelect: ";

        op = std::cin.get();
    }

    switch(op)
```

```
    {
    case '1':
        if (tcp_port == 0)
        {
            std::cout << "\nTCP/IP port number: ";
            std::cin >> tcp_port;
        }
        remote_commander(tcp_port);
        break;
    case '2':
        local_commander();
        break;
    case '3':
        device_scanning();
        break;
    case '4':
        joystick_controller();
        break;
    default:;
    }

    return EXIT_SUCCESS;
}
```

The figure below shows the call diagram.

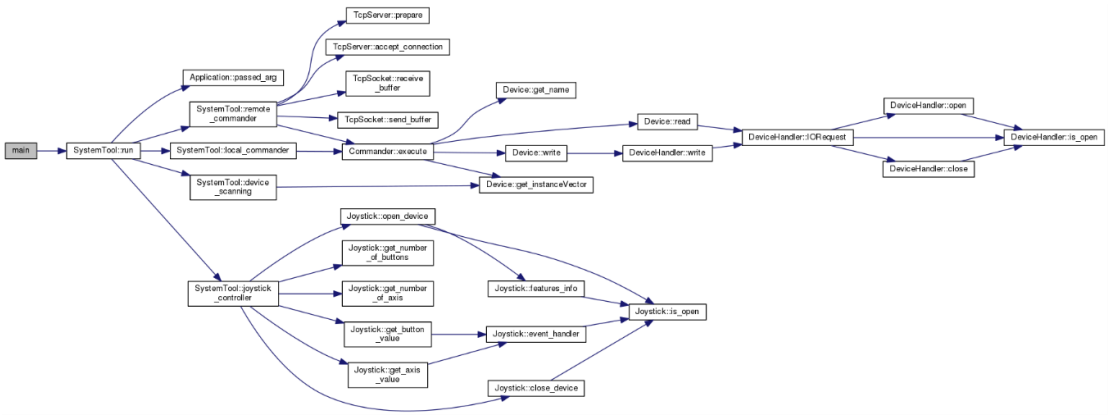

Figure 87: Call diagram for main function in System tool

7.2.2 Network command interface

This feature is implemented in **`SystemTool::remote_commander`**, it implements a synchronous client-server loop.

The remote client sends messages as commands, these commands are executed by **`Commander::execute`** in the base application framework.

Implementation:

```
void SystemTool::remote_commander(uint16_t server_port)
{
    TcpServer * server;
    TcpSocket * client;
    std::string message;
    std::string answer;
    bool exit_flag;

    server = new TcpServer(server_port);
    if (server != NULL)
    {
        if (server->prepare() != -1)
        for (;;)
```

```
        {
            std::cout << "\nWaiting for connection on server port: " << server_port <<
std::endl;
            client = server->accept_connection();

            if (client != NULL)
            {
                do
                {
                    if (client->receive_buffer(message) > 0)
                    {
                        std::cout << "\nClient: " << message.c_str();
                        exit_flag = Commander::execute(message, answer);
                        std::cout << "\nServer: " << answer.c_str();
                        client->send_buffer(answer);
                    }
                    else
                    {
                        exit_flag = true;
                    }
                } while(!exit_flag);

                delete client;
            }
        }
        delete server;
    }

    perror("\nExit");
}
```

This feature has access to monitor and control all devices in the system, since all the devices are accessed by the device vector from the **`Device`** class. This happens inside of the **`Commander::execute`**.

With polymorphism, all devices are interfaced with **`Device::read`** and **`Device::write`** functions from the base class.

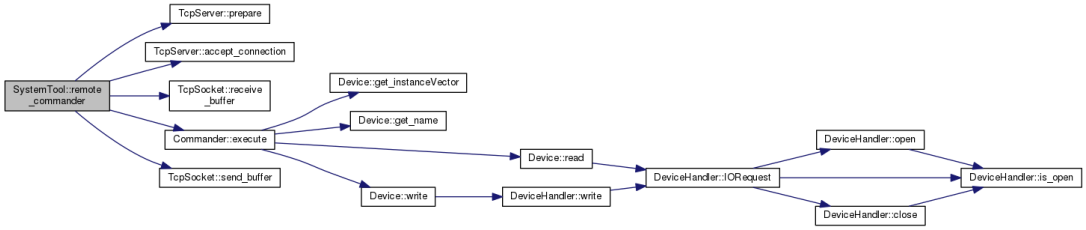

Figure 88: Call diagram for SystemTool::remote_commander

7.2.3 Command interface

This feature is implemented in **`SystemTool::local_commander`**, it implements a command line interface. The user type commands in the terminal, these commands are executed by **`Commander::execute`** in the base application framework.

Implementation:

```
void SystemTool::local_commander(void)
{
    std::string cmd;
    std::string answer;
    bool exit_flag;

    std::cout << "\nWaiting for command ...\n";

    do
    {
```

```
        std::getline(std::cin, cmd);
        exit_flag = Commander::execute(cmd, answer);
        std::cout << answer;
    } while(!exit_flag);
}
```

The parsing of the commands, and the execution is encapsulated in the `Commander` class, it generates a text answer or result, and it is posted in the terminal. In case of an unrecognized command, it returns a help message.

7.2.4 Device scanning

This feature is implemented in `SystemTool::device_scanning`, this function obtains the device vector from `Device` class, and in a loop it is obtained the name of each device, and it is sampled with `read` method. The names and the sample of all devices are displayed in the terminal. Pressing ENTER key breaks the loop.

Implementation:

```
void SystemTool::device_scanning()
{
    unsigned int data;
    int kbhit = 0;

    Device::InstanceVector instances = Device::get_instanceVector();

    fputs("\e[?25l", stdout); /* hide the cursor */
    system("clear");
    std::cout << "___ Scanning devices. Press ENTER to stop ___";
    for(;kbhit == 0;)
    {
        for (unsigned i = 0; i < instances.size(); i ++)
        {
            if (instances[i]->get_ID() != id_SERIAL_0.ID)
            {
                instances[i]->read(&data);
                printf("%c[%d;%df %s = 0x%X         ", 0x1B, i + 2, 10,
                       instances[i]->get_name().c_str(), data);
            }
        }
        ioctl(0, FIONREAD, &kbhit);
    }
    getchar();
    system("clear");
    fputs("\e[?25h", stdout); /* show the cursor */
}
```

Example of sampling the first device:

```
unsigned int data;
Device::InstanceVector instances = Device::get_instanceVector();
instances[0]->read(&data);
printf("0x%X", data);
```

7.2.5 Joystick controller

The purpose of this feature is to reuse the input interface from Joysticks in order to control programs. For this application, the Joystick axis and buttons are utilized to control actuators for testing purposes.

This feature is implemented in `SystemTool::joystick_controller`, it creates an instance of `Joystick` class from the base application framework, and then it reads the events from js0 in the file system. A USB joystick should be attached to the USB port in advanced.

The instance of the `Joystick` class provides the number of buttons and axis and their values, and then these values are assigned to different devices.

```
#define JOY "/dev/input/js0"
void SystemTool::joystick_controller(void)
{
    Joystick js(JOY);
    unsigned int buttons, axis;
    int kbhit = 0;

    fputs("\e[?25l", stdout);
    system("clear");
    std::cout << "___ Scanning Joystick. Press ENTER to stop ___";

    js.open_device();
    buttons = js.get_number_of_buttons();
    axis = js.get_number_of_axis();

    for(;kbhit == 0;)
    {
        for (unsigned i = 0; i < buttons; i ++)
        {
            printf("%c[%d;%df B[%d]=%d ", 0x1B, i + 2, 10,
                   i, js.get_button_value(i));
        }
        for (unsigned i = 0; i < axis; i ++)
        {
            printf("%c[%d;%df A[%d]=%d     ", 0x1B, i + 2 + buttons, 10,
                   i, js.get_axis_value(i));
        }
        ioctl(0, FIONREAD, &kbhit);

        pwm_0.write(js.get_axis_value(0)>>4);
        pwm_1.write(js.get_axis_value(2)>>4);
        serial_0.write(js.get_axis_value(1)>>4);
        flushValve.write(js.get_button_value(0));
        drainValve.write(js.get_button_value(1));
        shutOffValve.write(js.get_button_value(2));
        vacumGenerator.write(js.get_button_value(3));
        drainIndicator.write(js.get_button_value(4));
        leakageIndicator.write(js.get_button_value(5));
        relay_0.write(js.get_button_value(6));
        relay_1.write(js.get_button_value(7));
    }
    js.close_device();
    getchar();
    fputs("\e[?25h", stdout); /* show the cursor */
    system("clear");
}
```

The following code gets the value of the axis and buttons individually, and then it displays the values on the terminal:

```
        for (unsigned i = 0; i < buttons; i ++)
        {
            printf("%c[%d;%df B[%d]=%d ", 0x1B, i + 2, 10,
                   i, js.get_button_value(i));
        }
        for (unsigned i = 0; i < axis; i ++)
        {
            printf("%c[%d;%df A[%d]=%d     ", 0x1B, i + 2 + buttons, 10,
                   i, js.get_axis_value(i));
        }
```

The following assign values from axis and buttons to some devices:

```
        pwm_0.write(js.get_axis_value(0)>>4);
        pwm_1.write(js.get_axis_value(2)>>4);
        serial_0.write(js.get_axis_value(1)>>4);
        flushValve.write(js.get_button_value(0));
        drainValve.write(js.get_button_value(1));
        shutOffValve.write(js.get_button_value(2));
```

```
vacumGenerator.write(js.get_button_value(3));
drainIndicator.write(js.get_button_value(4));
leakageIndicator.write(js.get_button_value(5));
relay_0.write(js.get_button_value(6));
relay_1.write(js.get_button_value(7));
```

The following figure shows the call diagram for this feature.

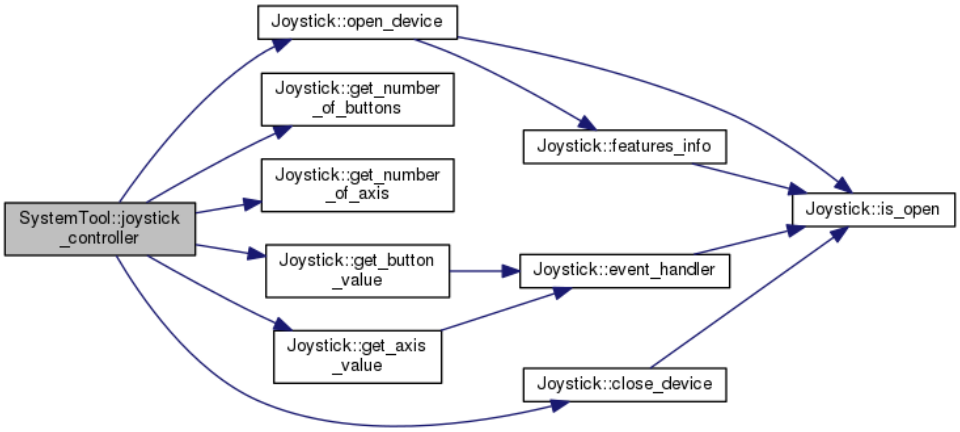

Figure 89: Call diagram for SystemTool::joystick_controller

Chapter 8. Test and results

The software testing tools provide information about the quality of the software product under test. Test techniques include the executon of an application or components of it with the intent of finding defects, and the verification that the software product is fit for use. Software testing involves evaluation of one or more properties of interest.

The number of possible test cases for even simple software components is practically infinite, hence it is important to have a strategy to select tests that are feasible for the available time and resources. Testing planning and execution is not the topic of this thesis; however, an important part of this thesis is the development of software tools for testing during the development and testing processes.

Two software testing tools are developed:

- **System tool**. Embedded Linux application suitable to run testing procedures on the development platform.
- **Remote access tool**. Multiplatform desktop application suitable to run remote testing procedures.

8.1 System tool

As explained in Chapter 7 (section 7.2), this embedded Linux application is installed in the file system: /bin/systool.

To run this application from the serial terminal:

```
root@linaro-ubuntu-desktop:~# systool
```

The application runs one processes at a time chosen from an options menu.

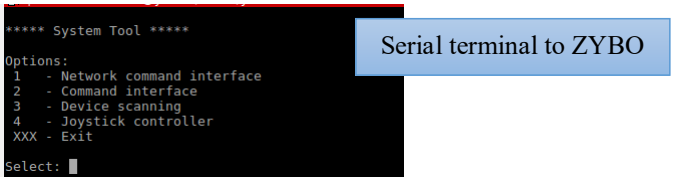

Figure 90: System Tool options menu

8.1.1 Network command interface

The first option starts the network command interface, it asks for the network port to communicate through, afterwards it creates an instance of a server socket, and finally it waits for an incoming client connection.

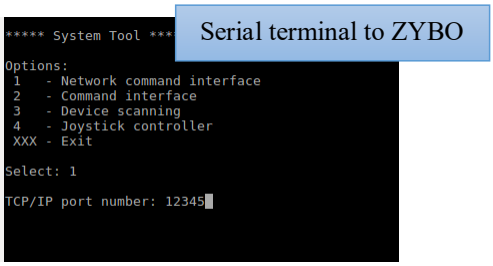

Figure 91: Network command interface

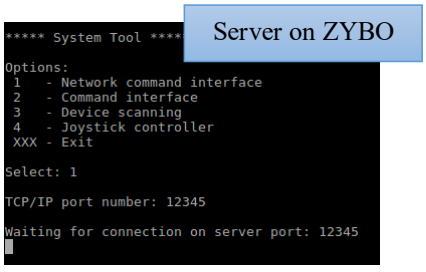

Figure 92: System Tool waiting for connection

To start a connection with the server, the platform should be connected to the network, it is done through Ethernet cable. A remote system can establish communication via Telnet.

8.1.1.1 Telnet connection

Telnet is a terminal emulation program for TCP/IP networks such as the Internet. The Telnet program runs on a computer and it connects to a server on the network. The user then enters text-messages or commands through the Telnet program and these messages will be sent to the remote server to be processed, and the answer from the server is sent back to the Telnet client.

In order to establish any TCP/IP connection, it is needed to know the IP address of the server (on Linux systems, it can be obtained with ifconfig command).

Once having the IP address, it can be started a TCP/IP connection in a remote computer connected to the network. The figure below shows the Telnet connection to the IP 192.168.1.112 (server running on the ZYBO) and the port 12345 (arbitrarily chosen).

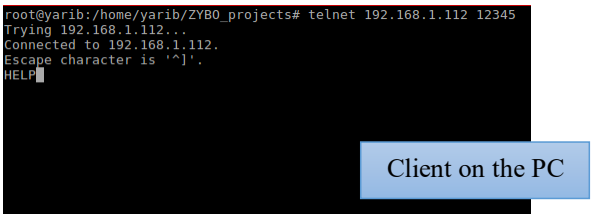

Figure 93: Telnet interface

Once connected, the user (PC) can start sending text commands so the server (ZYBO) processes them, and it sends an answer back. The server (ZYBO) displays the incoming messages and answers on the serial terminal. This is shown in the next figures.

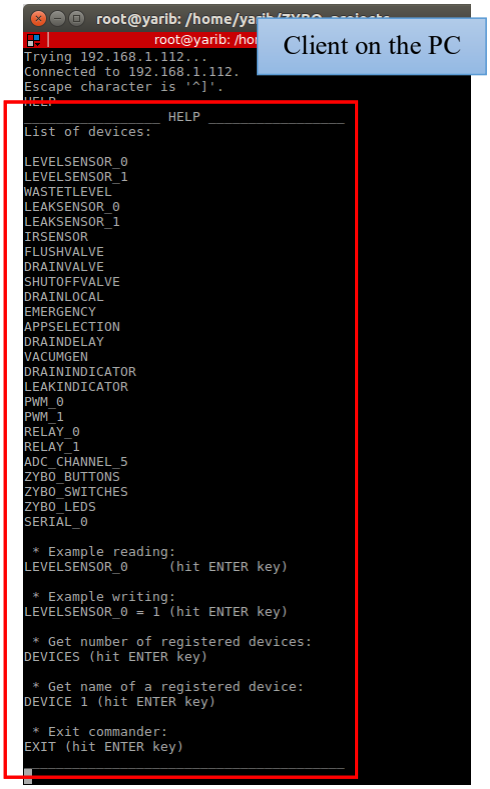

Figure 94: Telnet client

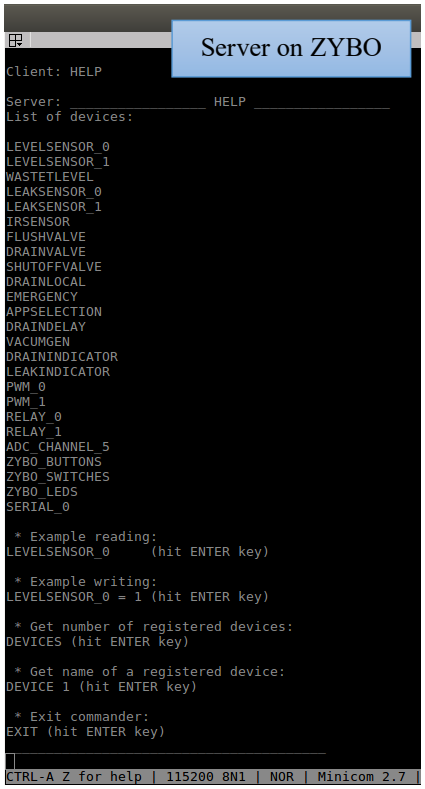

Figure 95: Server display

The help message is sent back from the server as the answer for unrecognized commands. The help message contains the actual list of devices.

8.1.1.2 Remote request for device information

To request the number of devices (when using **SYSTEMBOX**), it is typed the command "DEVICES". In this case the number of devices is 25 (devices in **SYSTEMBOX**).

To request the name of a particular device, it is typed the command "DEVICE *n*", where *n* is the index of the device from the list of devices. If it is requested a number out of the range, the answer is "NO DEVICE".

The following image shows this use cases, the Telnet client is in the left side, the serial terminal connected to ZYBO is in the right side.

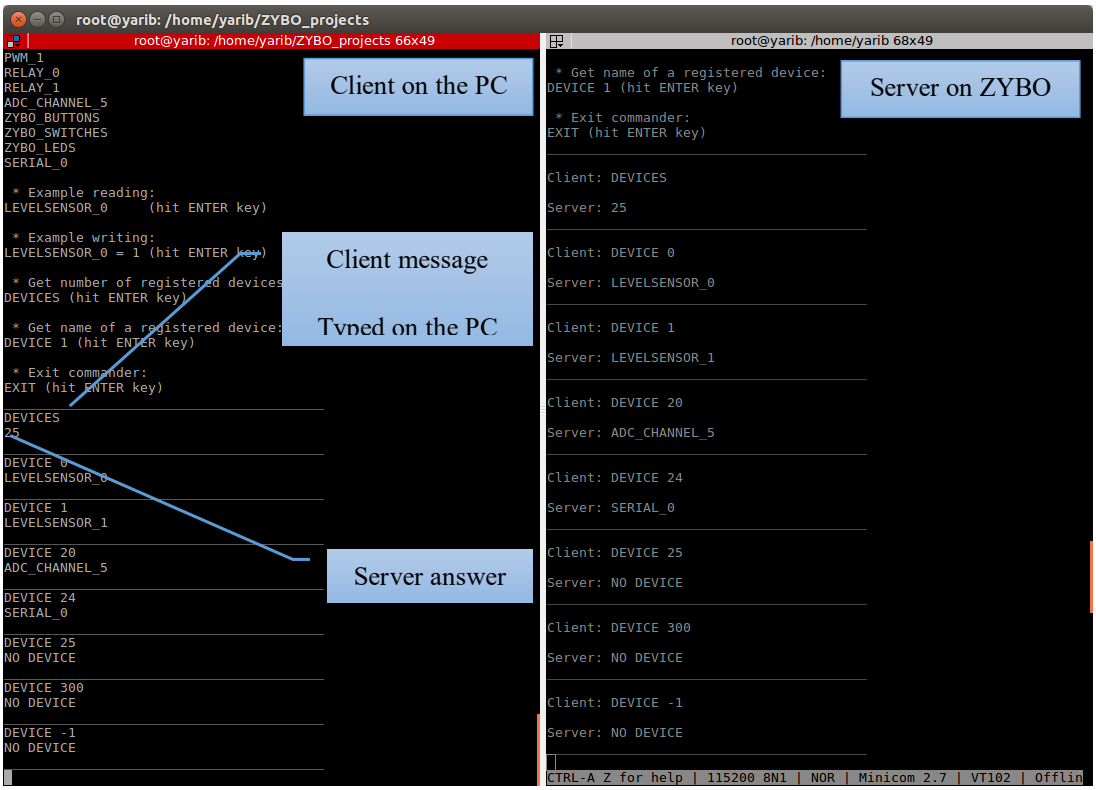

Figure 96: Telnet request for device information

8.1.1.3 Remote device writing

To perform a device writing, it should be typed the name of the device followed by the equals sign, and the value to write.

Remote writing to RELAY_0 and RELAY_1:

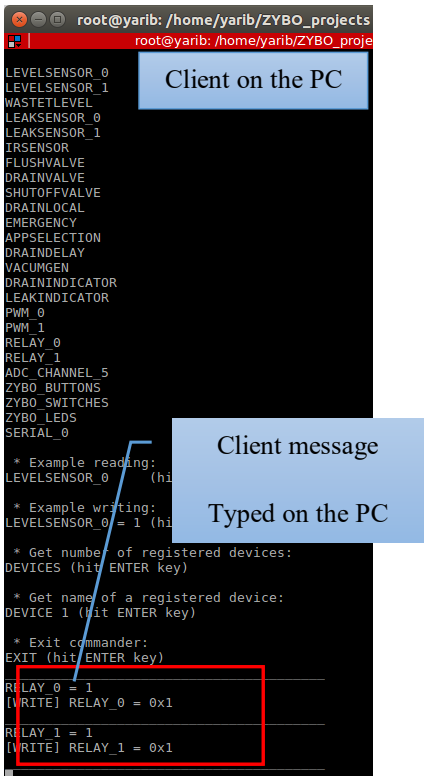

Figure 97: Remote writing to RELAY_0 and RELAY_1

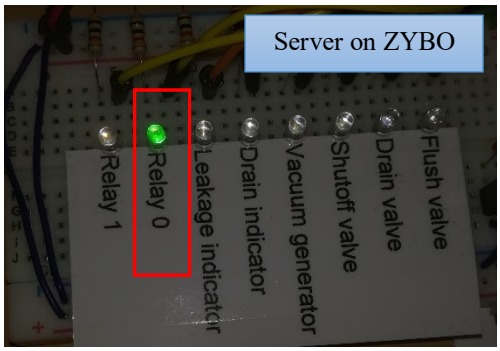

Figure 98: Writing to RELAY_0

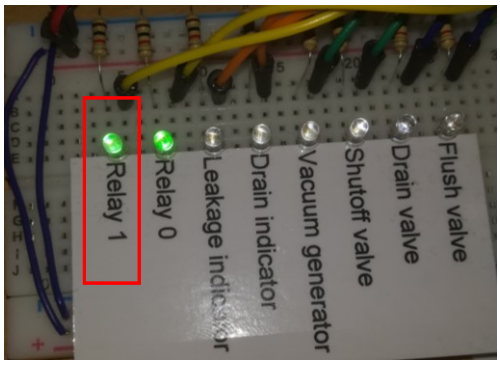

Figure 99: Writing to RELAY_1

Remote writing to SERIAL_0:

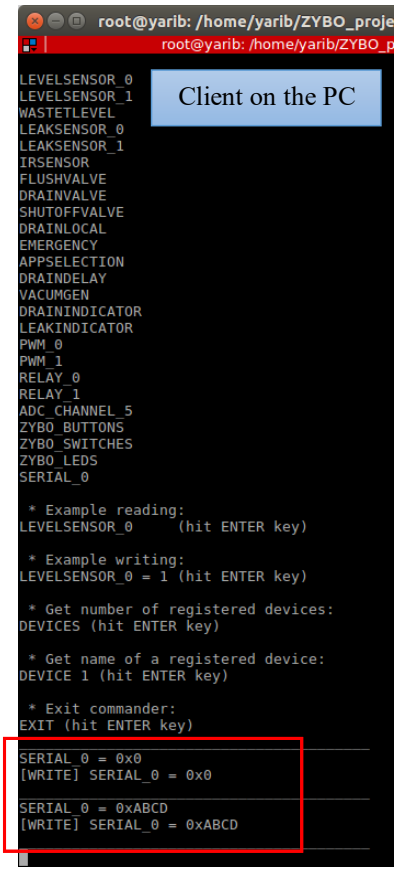

Figure 100: Remote writing to SERIAL_0

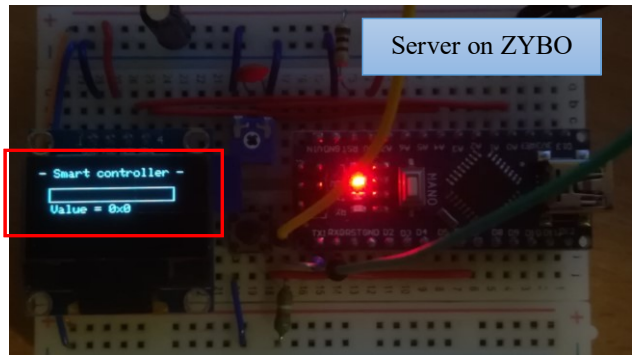

Figure 101: Writing 0x0 to SERIAL_0

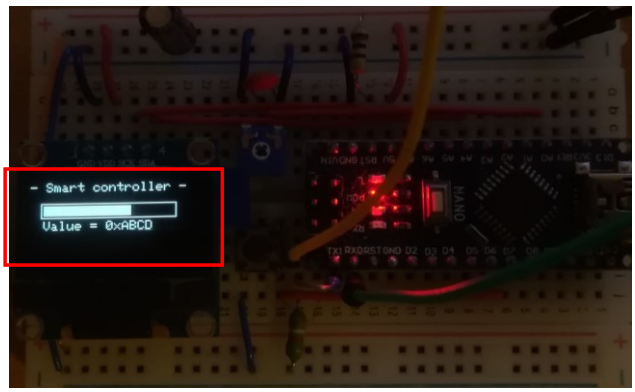

Figure 102: Writing 0xABCD to SERIAL_0

An ATMEL microcontroller is programmed to display values from UART on an OLED display, this device is connected to the ZYBO through the UART 0 of the Zynq SoC.

Remote writing to ZYBO_LEDS:

```
VACUMGEN
DRAININDICATOR
LEAKINDICATOR
PWM_0
PWM_1
RELAY_0
RELAY_1
ADC_CHANNEL_5
ZYBO_BUTTONS
ZYBO_SWITCHES
ZYBO_LEDS
SERIAL_0

 * Example reading:
LEVELSENSOR_0     (hit ENTER key)

 * Example writing:
LEVELSENSOR_0 = 1 (hit ENTER key)

 * Get number of registered devices:
DEVICES (hit ENTER key)

 * Get name of a registered device:
DEVICE 1 (hit ENTER key)

 * Exit commander:
EXIT (hit ENTER key)
_____________________________________
SERIAL_0 = 0x0
[WRITE] SERIAL_0 = 0x0
_____________________________________
SERIAL_0 = 0xABCD
[WRITE] SERIAL_0 = 0xABCD
_____________________________________
SERIAL_0 = 0
[WRITE] SERIAL_0 = 0x0
_____________________________________
ZYBO_LEDS = 0x0A
[WRITE] ZYBO_LEDS = 0xA
_____________________________________
```

Figure 103: Remote writing to ZYBO_LEDS

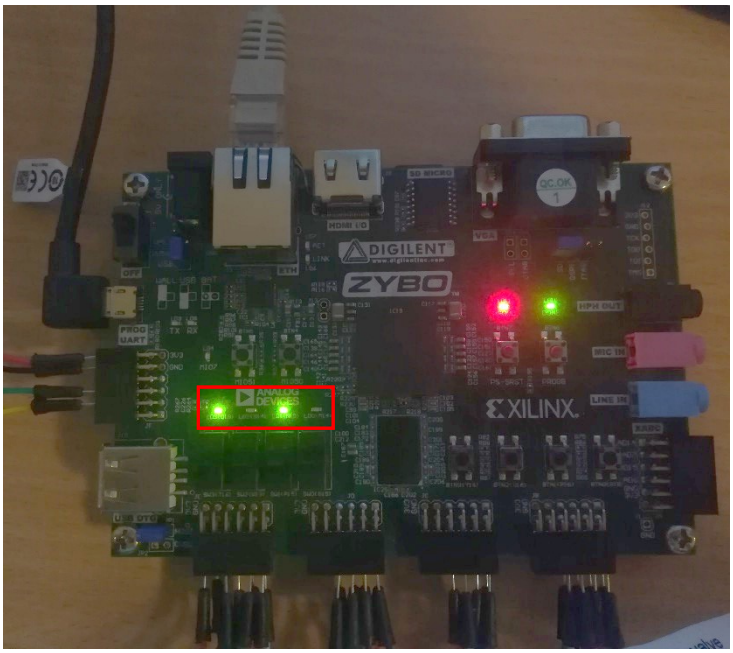

Figure 104: Writing to ZYBO_LEDS

8.1.1.4 Remote device reading

To perform a device reading, it should be typed the name of the device, the server will read the device and it will send the value back.

Remote reading from ZYBO_SWITCHES:

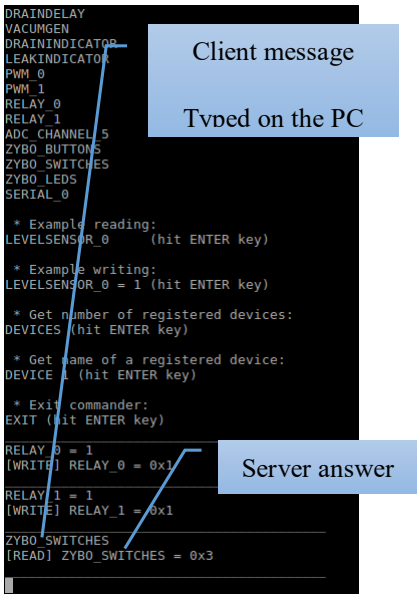

Figure 105: Remote read ZYBO_SWITCHES

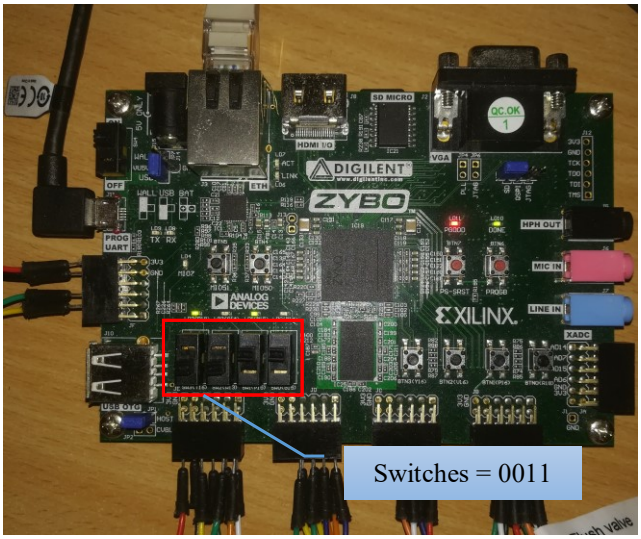

Figure 106: Read ZYBO_SWITCHES

Remote read from WASTETLEVEL:

```
DRAININDICATOR
LEAKINDICATOR
PWM_0
PWM_1
RELAY_0
RELAY_1
ADC_CHANNEL_5
ZYBO_BUTTONS
ZYBO_SWITCHES
ZYBO_LEDS
SERIAL_0

 * Example reading:
LEVELSENSOR_0     (hit ENTER key)

 * Example writing:
LEVELSENSOR_0 = 1 (hit ENTER key)

 * Get number of registered devices:
DEVICES (hit ENTER key)

 * Get name of a registered device:
DEVICE 1 (hit ENTER key)

 * Exit commander:
EXIT (hit ENTER key)
_____________________________________
WASTETLEVEL
[READ] WASTETLEVEL = 0x16
_____________________________________
WASTETLEVEL
[READ] WASTETLEVEL = 0xFFF
_____________________________________
```

Figure 107: Remote read WASTETLEVEL

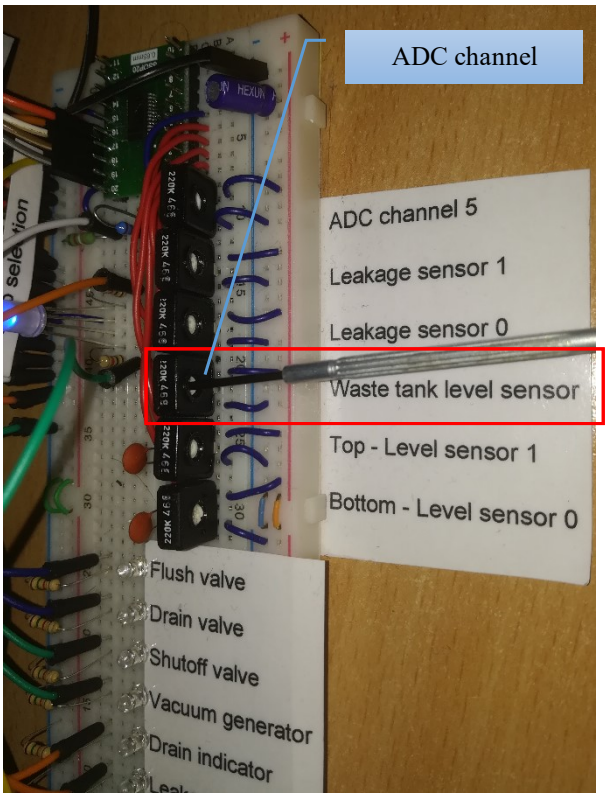

Figure 108: Read WASTETLEVEL

This device reads from the ADC that has potentiometers connected to the input channels.

8.1.1.5 Disconnect from server

In order to disconnect from server, it should be typed "`EXIT`". This command will break the server loop, and the server will wait again for more client connections.

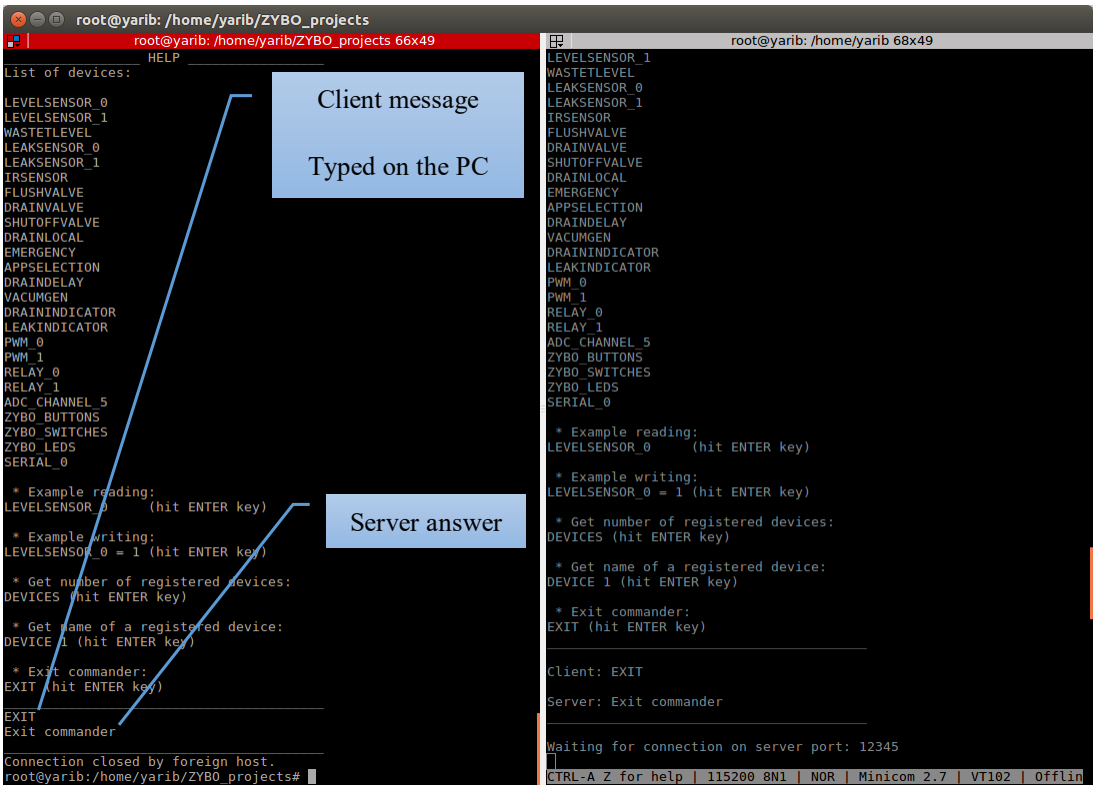

Figure 109: Disconnect from server

8.1.2 Command interface

This is the second option in the menu of the System Tool. The same functionality as the Network command interface, however in this case the user establish a local communication via serial terminal.

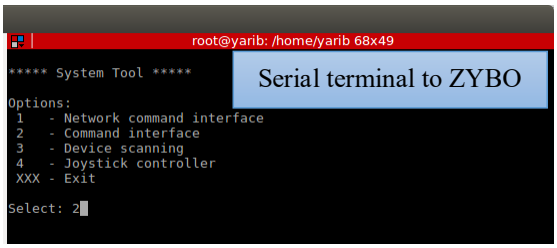

Figure 110: System tool menu, option two (Command interface)

Once selected the option two, the tool will display the help message.

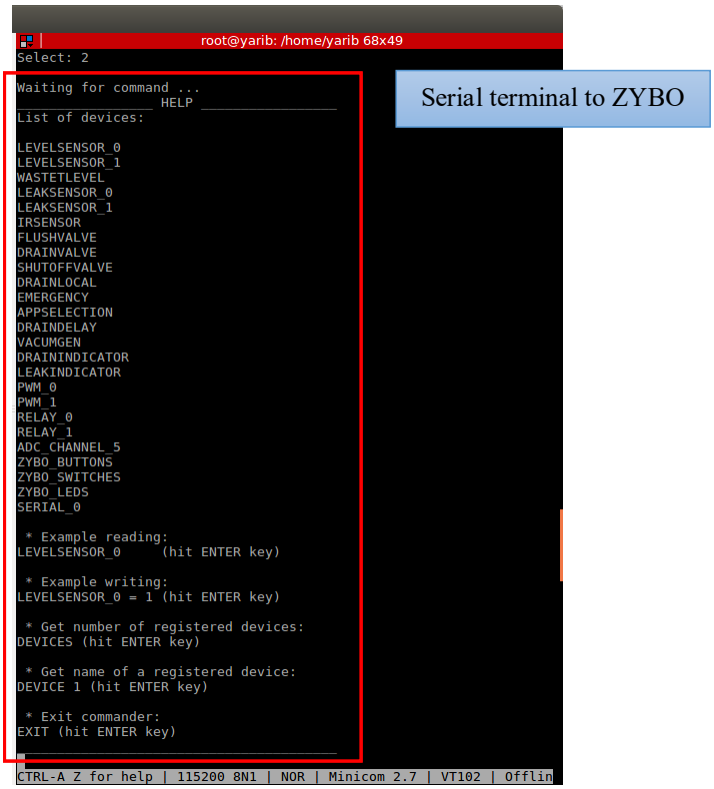

Figure 111: Help message from Command interface

Writing to some devices:

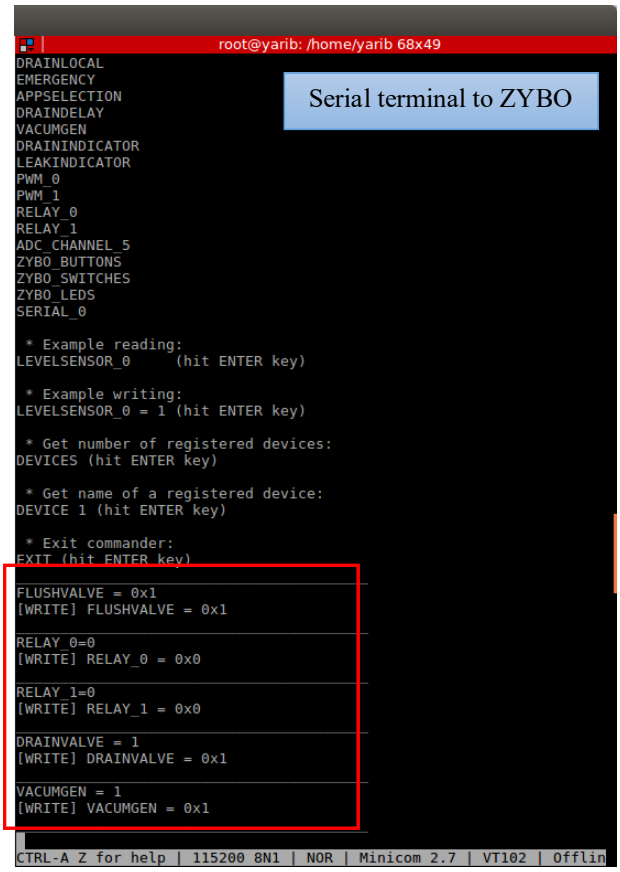

Figure 112: Writing to devices with Command interface

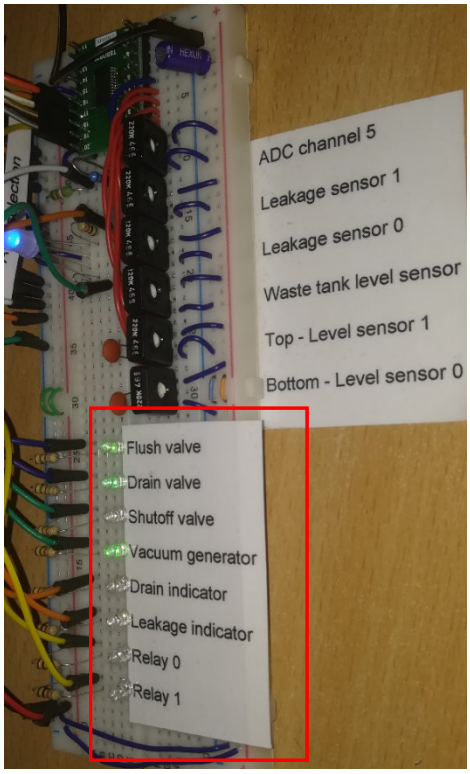

Figure 113: Manipulating devices with Command interface

In order to exit from the Command interface, it should be typed "`EXIT`".

8.1.3 Device scanning

This feature implements a loop that is constantly monitoring all devices and displaying their values in real time.

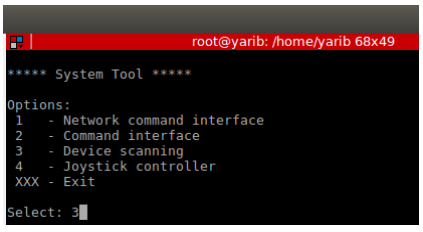

Figure 114: System tool menu, option three (Device scanning)

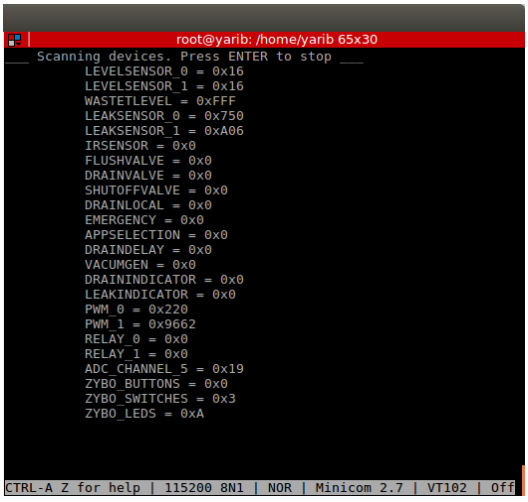

Figure 115: Device scanning

In order to stop the scanning, the user should press ENTER key in the terminal.

8.1.4 Joystick controller

This feature allows to reuse axis and buttons of any USB Joystick to control and test actuators.

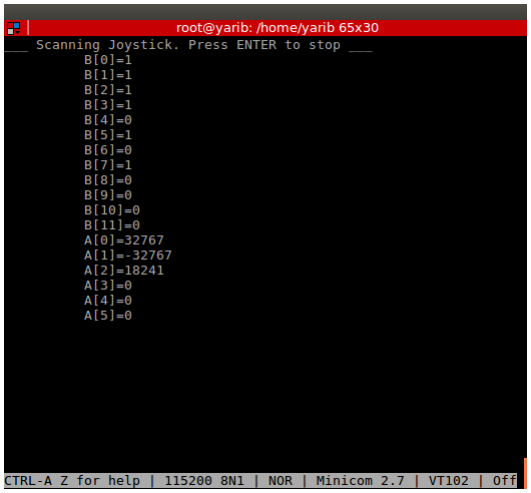

Figure 116: Scanning USB Joystick

Figure 117: Generic USB Joystick

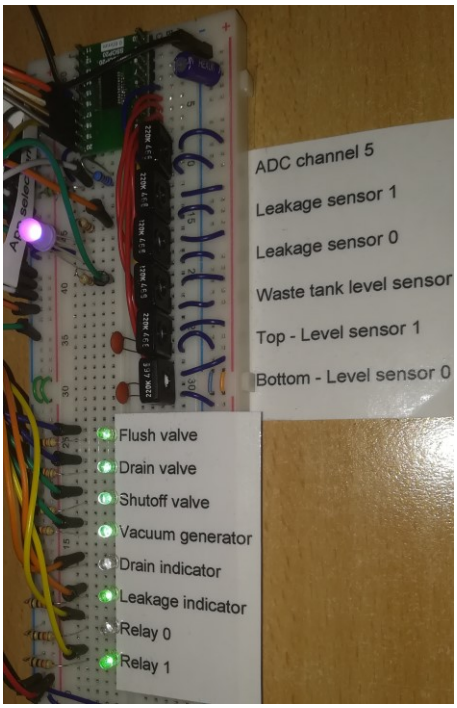

Figure 118: Testing devices with USB Joystick

The button mapping is customizable in the source code of this feature.

```
pwm_0.write(js.get_axis_value(0)>>4);
pwm_1.write(js.get_axis_value(2)>>4);
serial_0.write(js.get_axis_value(1)>>4);
flushValve.write(js.get_button_value(0));
drainValve.write(js.get_button_value(1));
shutOffValve.write(js.get_button_value(2));
vacumGenerator.write(js.get_button_value(3));
drainIndicator.write(js.get_button_value(4));
leakageIndicator.write(js.get_button_value(5));
relay_0.write(js.get_button_value(6));
relay_1.write(js.get_button_value(7));
```

To close the tool, the user should press ENTER key in the terminal.

8.2 Remote access tool

The Remote access tool is a multiplatform desktop application suitable to run remote testing procedures. This tool performs as a TCP/IP client, it is capable to request device information, remote reading and writing. This is a standalone application with graphical user interface (GUI) developed in C++ and Qt application framework.

This application communicates with the server imitating Telnet command messaging. The GUI application builds the text of the command messages, and it sends them to the server, finally, the answers from the server are parsed by the tool and displayed on its GUI.

8.2.1 User interface

The main window application has the following user interface elements:

- **Server IP address and port number**. These fields set the IP address and port of the server to connect
- **Connection controls**. Control buttons to connect and disconnect from the server
- **Connection status**. Provides information about the current status of the connection
- **Number of devices**. Display the number of devices available from the server
- **Device list area**. List of devices from the server, each element in the list is double clickable to open its corresponding device control dialog.
- **Log area**. Area to display client-server messages.
- **Clear log area**. Button to clean the current logs from the log area.

The following figure shows this user interface.

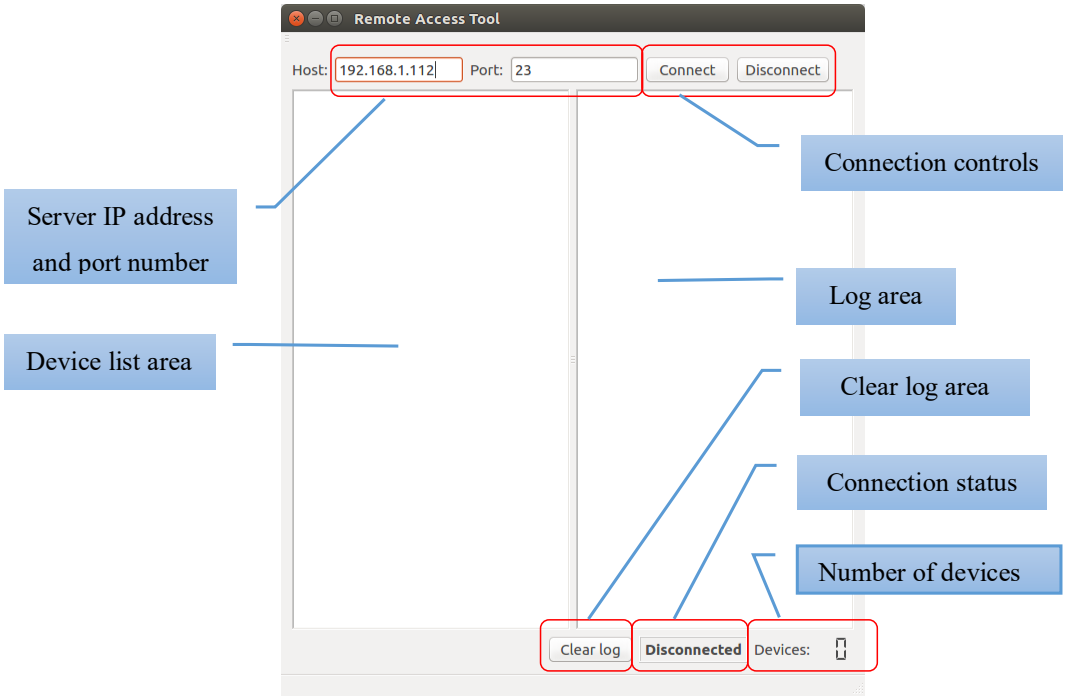

Figure 119: Remote access tool

8.2.2 Device list

Once the application is connected to the server, it performs the following steps:

1. **Request number of devices**. The client application sends the command: "`DEVICES`". Then, the server sends the number of devices as the answer.
2. **Request name of each device**. For each device, the client application sends the command: "`DEVICE` *n*" (*n* is within the range from 0 to the number of devices gotten from the server, minus one). Then, the server sends the name of each requested device.

3. **Create device list**. From the previous information of devices, the tool creates a list of names of devices in the device list area of the UI.
4. **Update UI information fields**. The number of devices, and the logs are displayed in the corresponding fields.

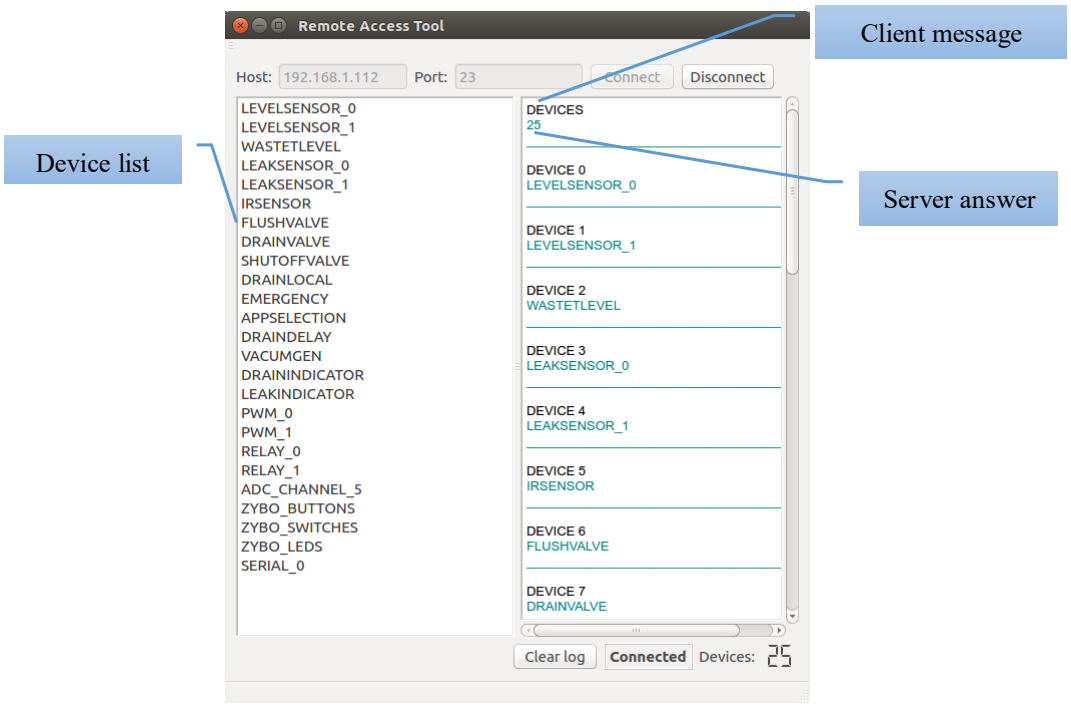

Figure 120: Device list and logs

8.2.3 Device dialog

Once the device list is displayed, the user is able to double click on any of the devices, then a device dialog will show up.

The device dialog, UI elements:

- **Device name**. Device name
- **Read button**. Sends the read message to the server
- **Value from read**. Displays the value gotten from the server (Hexadecimal format)
- **Progress bar**. This progress bar is used for reading ADC values, its range goes from 0 to 0xFFF
- **Timer**. This check box enables a timer that will trigger the read message every 30 milliseconds
- **Value to write**. This editable spin box is the input of integer values for writing to the device
- **Write button**. Send the write message to the server
- **Slider**. This slider modifies the value for writing to the device and it sends the write message to the server on any change event. The range goes from 0 to 0xFFFF

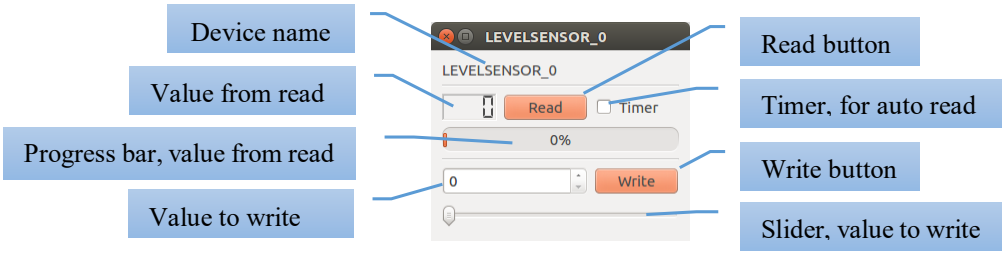

Figure 121: Device dialog

8.2.4 Remote device writing

To perform a device writing, the user should open a device dialog by double clicking it from the device list. Then the user should type the value in the input line of the spin box, and finally, the user should hit enter, or press the write button.

Writing to the VACUUMGEN device:

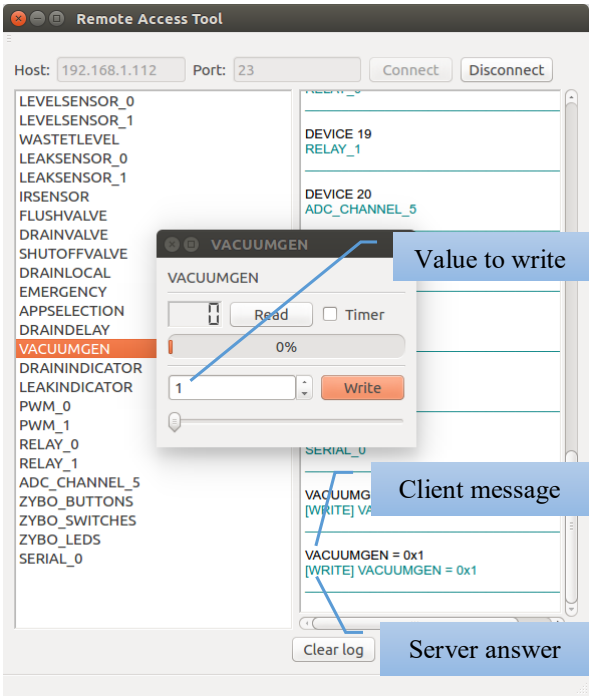

Figure 122: Remote writing to VACUUMGEN

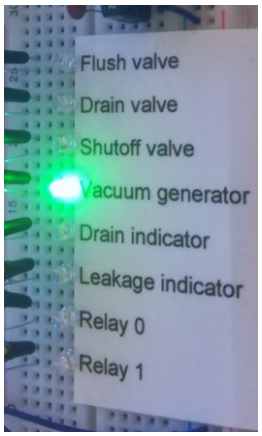

Figure 123: Writing to VACUUMGEN

Writing to ZYBO_LEDS:

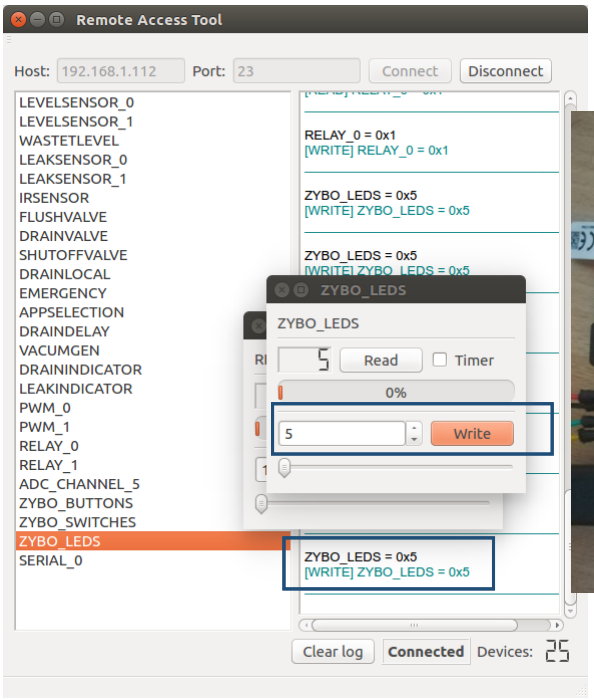

Figure 124: Remote writing to ZYBO_LEDS

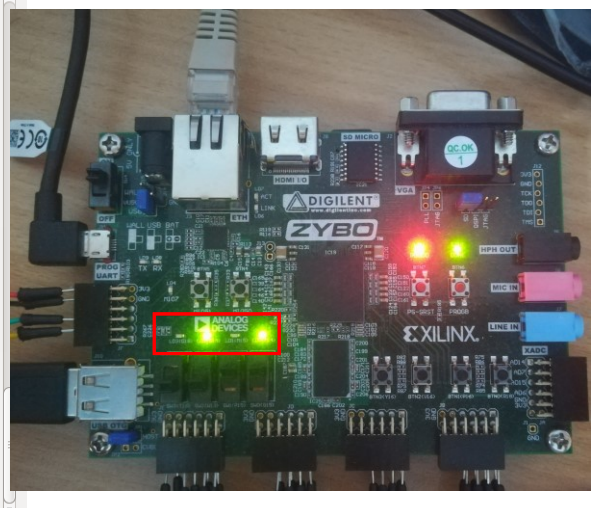

Figure 125: Writing to ZYBO_LEDS

Writing to SERIAL_0:

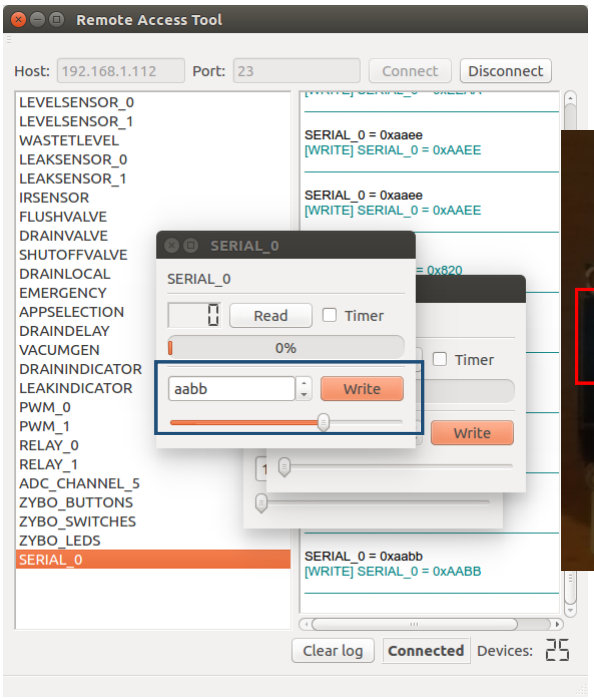

Figure 126: Remote writing to SERIAL_0

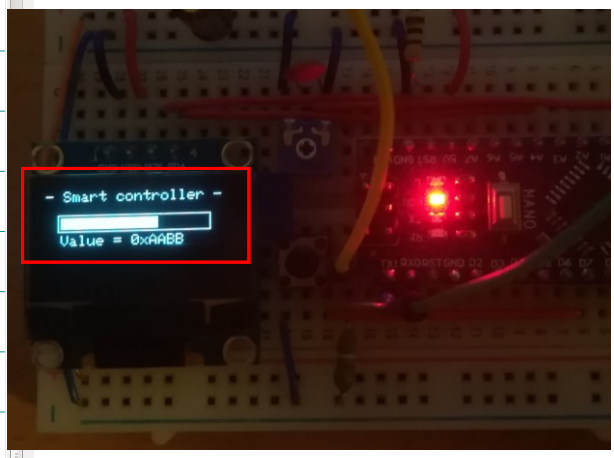

Figure 127: Writing to SERIAL_0

8.2.5 Remote device reading

To perform a device reading, the user should open a device dialog by double clicking it from the device list, then the user should press the read button, optionally the user can tic the timer check box.

Reading from the ZYBO_BUTTONS device:

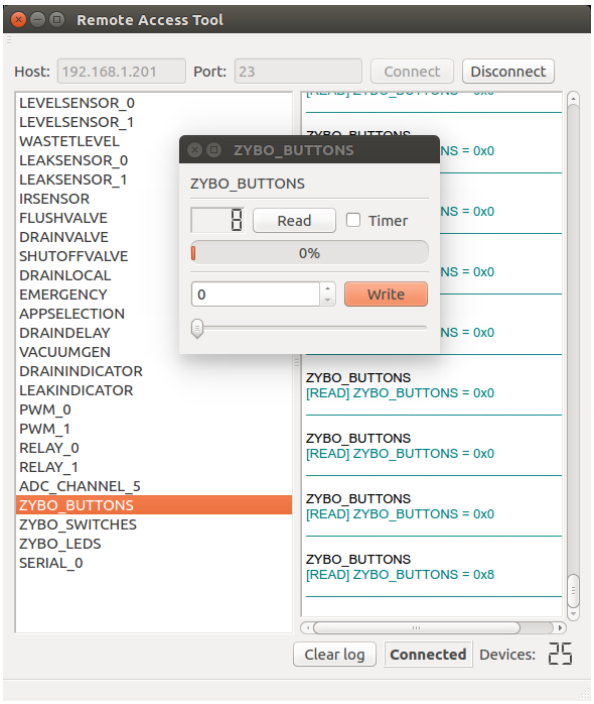

Figure 128: Remote reading from ZYBO_BUTTONS

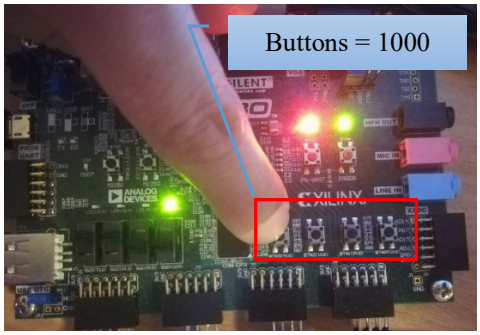

Figure 129: Reading ZYBO_BUTTONS

Reading from both LEVELSENSOR_0 and WASTETLEVEL at the same time with timer:

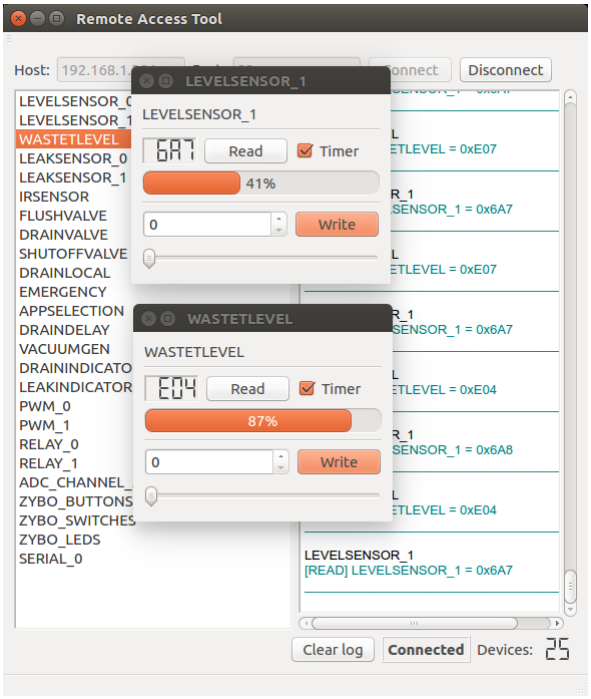

Figure 130: Remote reading from both LEVELSENSOR_0 and WASTETLEVEL at the same time with timer

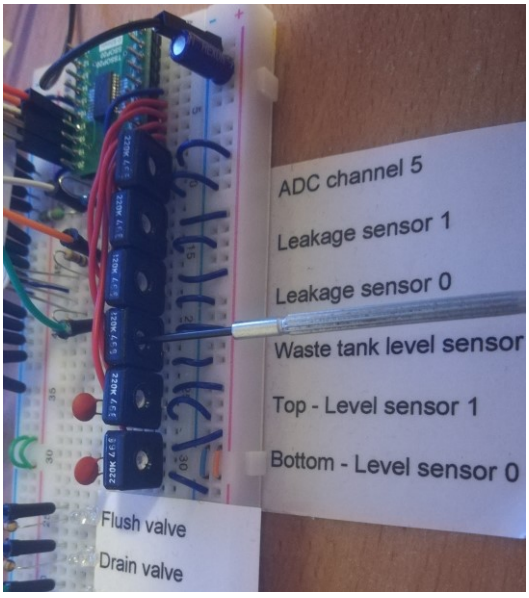

Figure 131: Reading from both LEVELSENSOR_0 and WASTETLEVEL

8.2.6 Error logging

All errors during the client server communication are logged in the same log area. The errors are displayed in red colour.

These are some examples:

On the left side image, the tool is trying to connect to an invalid IP address. On the right side image, the server was powered down.

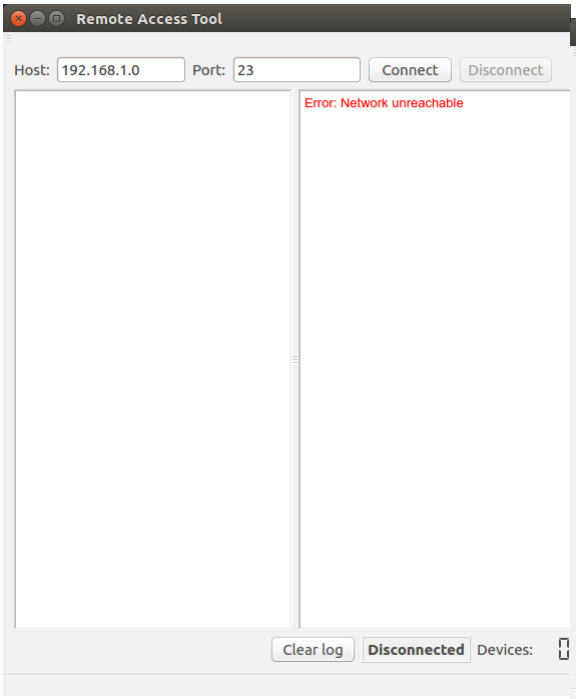

Figure 132: Error logging on invalid server IP address

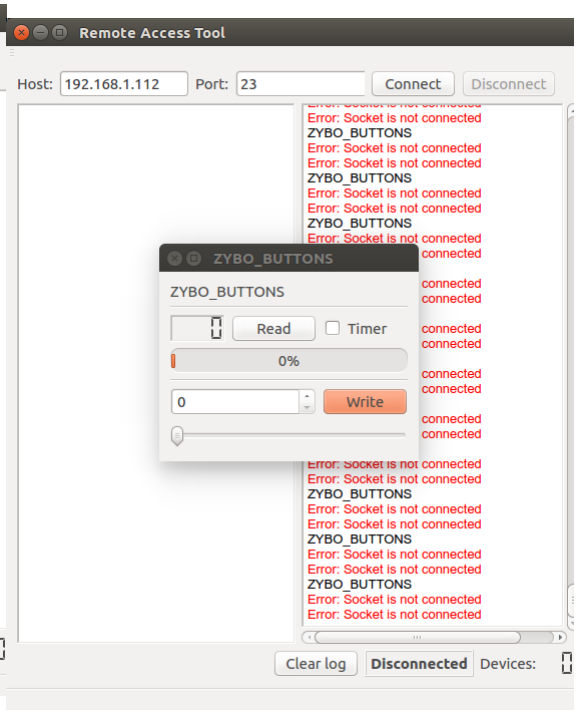

Figure 133: Error logging on server powered down

Error logging when debugging server:

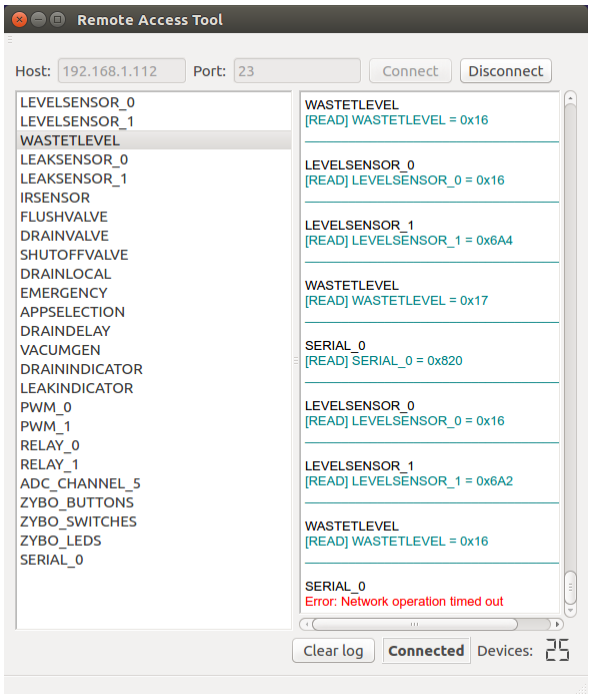

Figure 134: Error logging when debugging server

8.3 Future work

The Remote Access Tool is capable to run as a remote monitoring software during the performance of the business logic threads, threads decoupled and server monitoring.

As software samples, needless to mention, these tools can be extended or modified in order to automate procedures, or perform specialized or business tasks.

There is another master thesis covering the formal planning and execution of testing.

Chapter 9. **Conclusions**

The main effort for this master thesis was the development of a software architecture, which primary benefits are modularity, reusability, and extensibility. This software architecture was designed to allow plugging software components together, in order to promote a highly effective, and fast low-cost software development of final user applications.

Running Linux OS on Zynq SoC brought substantial benefits. The support for networking, file system, and driver modules allowed internet services, storage management, and USB communication with commercial devices. In addition, the scheduling for multi-threading processing in user mode was really great for non-real time applications; nevertheless, if it would have been the case of hard real-time processing, it would have been done in kernel mode or in the FPGA side.

Regarding hardware, the FPGA side of the Zynq SoC was capable to implement customized hardware and reconfigure as required. The prototype for this thesis required ordinary hardware, a large amount of digital IO lines, PWMs, and SPI communication with an ADC. However, this prototype is ready to be expanded with sophisticated hardware, for example, specialized hardware accelerators, signal processing, customized business logic, additional symmetric or asymmetric multiprocessing and more.

In conclusion, as academic work, the present thesis produced the SoC-FPGA design, operating system setup for Xilinx Zynq, and software architecture for “The cabin convenient controller unit”. Named as concept of smart avionics controller.

References

[1] E. White, Making Embedded Systems, O'Reilly, October 2011.

[2] A. P. Xilinx.

[3] Xilinx, Zynq-7000 All Programmable SoC, Technical Reference Manual, September 27, 2016.

[4] A. R. G. K.-H. Jonathan Corbet, Linux Device Drivers, O'Reilly Media, Inc.

[5] M. KerrisK, The Linux Programming interface: a Linux and UNIX System Programming Handbook, San Francisco: No Starch Press, Inc., 2010.

[6] ARM, ARM7TDMI Technical Reference Manual.

[7] P. B. Secretary, "Power.org™ Standard for Embedded Power Architecture™ Platform Requirements," EEE-ISTO, Piscataway, 2011.

[8] L. community, "Embedded Linux Wiki".

[9] O. R.Peterson, Linux - The Complete Reference, McGraw-Hill.

[10] J. O. a. A. S. Mark Mitchell, Advanced Linux Programming, Pearson Education.

[11] L. community, "The Linux man-pages project," 2017.

[12] W. Pree, Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1994.

[13] R. H. C. a. N. Islam, "A Technique for Documenting the Framework of an Object-Oriented System," *Computing Systems, Vol. 6, No. 4.*

[14] D. C. Schmidt, Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Softwarec, MacMillan Computer Publishing.

[15] E. T. Birrer, "Frameworks in the Financial Engineering Domain: An Experience Report," *ECOOP '93 Proceedings, Lecture Notes in Computer Science nr. 707, Springer-Verlag.*

[16] R. J. a. B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming. SIGS.*

[17] R. H. R. J. a. J. V. Erich Gamma, "Design Patterns: Elements of Reusable Software Architecture".

[18] H. H. a. R. J. a. R. Engel, "A Framework for Network Protocol Software," *Proceedings of OOPSLA,* October 1995..

[19] D. S. H. a. M. E. Fayad, "Achieve Bottom-Line Improvements with Enterprise Frameworks," *The Communications of ACM,* October 1997..

[20] ARP4761, AEROSPACE RECOMMENDED PRACTICE, 1996.

[21] Xilinx, “Debugging with System Debugger (TCF)”.

[22] Digilent, ZYBO™ FPGA Board Reference Manual, 2016.

[23] MAXIM, “MAX11626–MAX11629/MAX11632/MAX11633 Data datasheet”.

[24] XILINX, “AXI GPIO v2.0 LogiCORE IP Product Guide,” 2016.

[25] Xilinx, “AXI Timer v2.0, LogiCORE IP Product Guide,” 2016.

[26] R. A. E. M. A. E. R. W. S. Louise H. Crockett, The Zynq Book, Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC.

[27] DIGILENT, Embedded Linux ® Hands-on Tutorial for the ZYBO, July 17, 2014.

[28] AVNET, Ubuntu on Zynq®-7000 All Programmable SoC Tutorial For ZedBoard and Zynq Mini-ITX Development Kitsc, July 2014.

[29] Linaro, “https://www.linaro.org/about/,” [Online].

[30] L. Bass, R. Kazman and P. Clements, Software Architecture in Practice, Third Edition, Addison-Wesley Professional, 2012.

Appendix A Building Linux image for Xilinx Zynq

This Appendix is documenting how to build a bootable Linux image for Xilinx Zynq.

In order to boot Linux on a Zynq-7000 AP device, it is need to have four files present on the boot medium (in this case micro SD card).

1. **BOOT.BIN**. Zynq boot image file
2. **uImage**. Linux kernel image
3. **devicetree.dtb**. Device tree blob
4. **Linux distribution**. File system, GNU tools and libraries, additional software, and documentation

Zynq boot image

The *BOOT.BIN* file is actually the combination of 2 compulsory files, the First Stage Boot Loader (FSBL) and Second Stage Boot Loader (SSBL) executable in linkable format (.elf) files, and an optional Bitstream (.bit) file. The FSBL and SSBL files, as their name suggests, contain the final stages of the bootloader which is uses to load Linux on the device. The Bitstream is the file that is used to configure the programmable logic of the Zynq-7000 AP device. [26]

The ordering of the files that are stitched together to form the boot image is important; the Bitstream file (optional), if required, must be placed after the FSBL file and before the SSBL file.

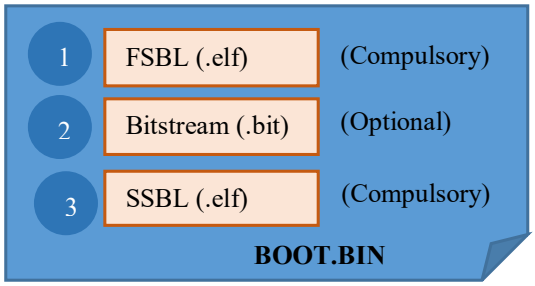

Figure 135: Zynq boot image file (BOOT.BIN)

The creation of the Zynq boot image file is described in the following 4 steps:

1. **Bitstream**

The Bitstream file contains the hardware description for the Zynq Programmable Logic (PL).

In Vivado, after synthesis and implementation of the hardware design, it can be generated the Bitstream. The following images show how to generate and export hardware including the Bitstream.

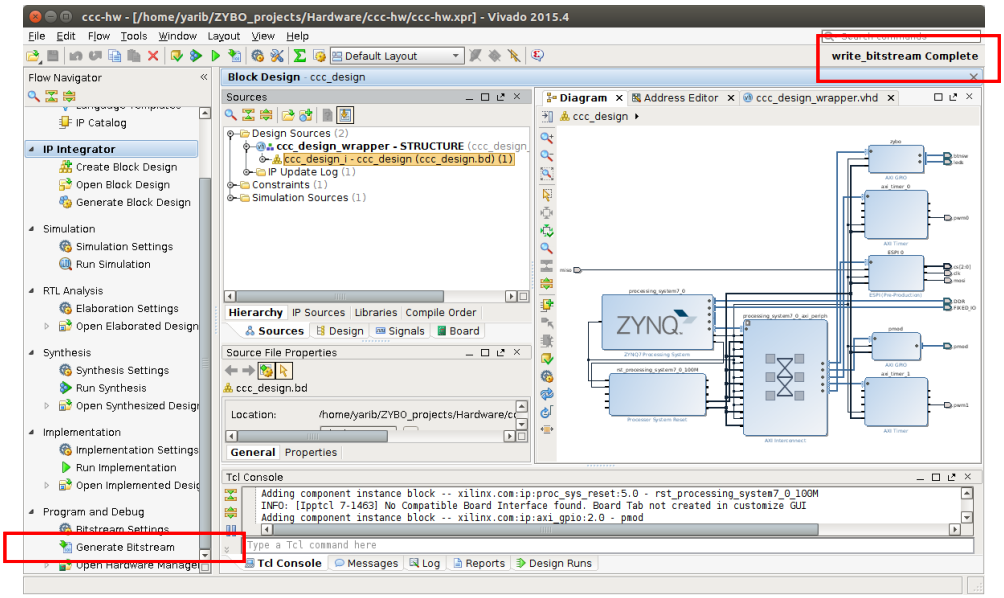

Figure 136: Generate Bitstream in Vivado

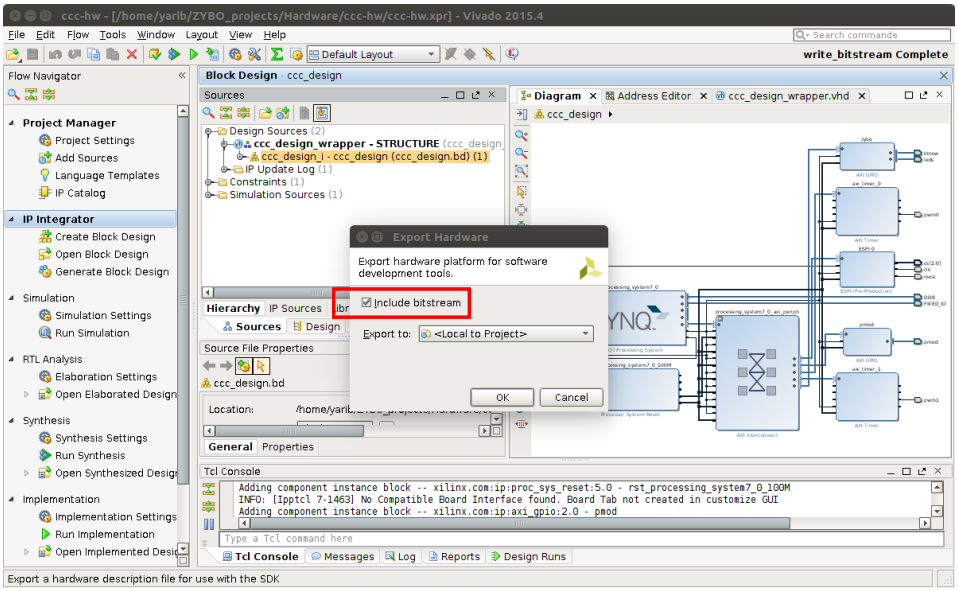

Figure 137: Exporting hardware from Vivado

For more detailed information: The Zynq Book [26].

2. **FSBL**

The FSBL is created in the Xilinx SDK, that is launched from the Vivado hardware design project. Once the SDK is launched from the Vivado project itself, it will contain hardware definitions to create the Board Support Package (BSP), which is needed to create baremetal applications including the FSBL.

The FSBL should be created as a new project from Zynq FSBL template, as shown in the following image.

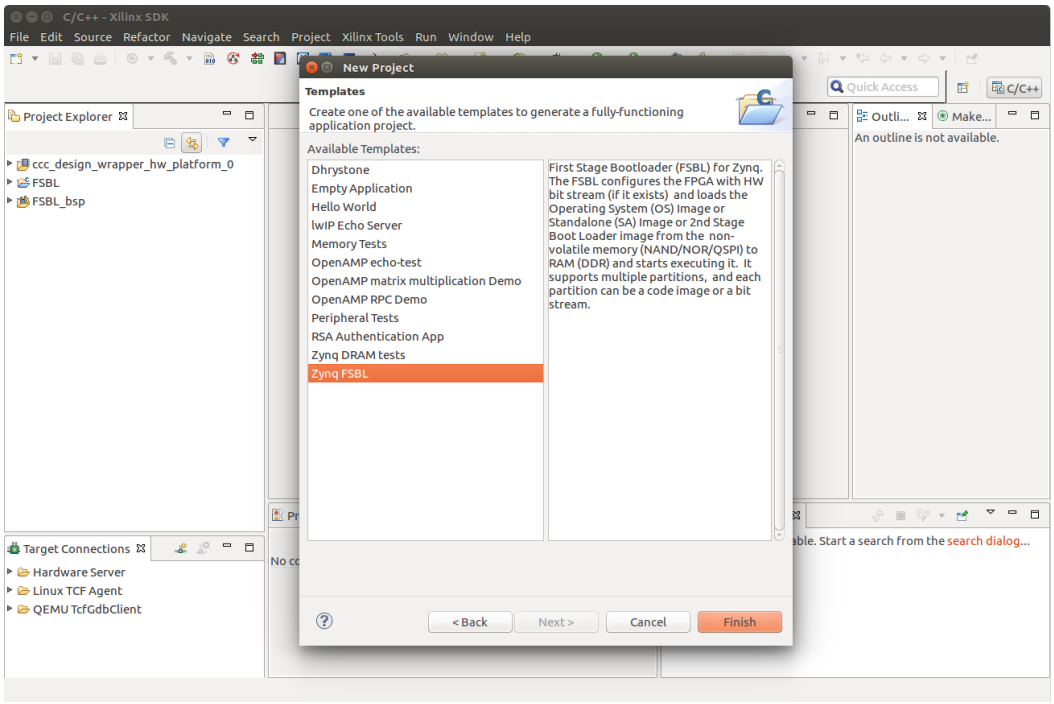

Figure 138: Creation of FSBL

Once the FSBL template is instantiated, it will auto compile and generate the *FSBL.elf* file.

3. **SSBL (u-boot)**

The SSBL is U-Boot, which is the Linux bootloader, the source code is obtained from the Digilent Git repository.

The repository containing the source code is cloned with the following git command:

```
git clone -b master-next https://github.com/DigilentInc/u-boot-Digilent-Dev.git
```

After cloning the repository, this can be compiled with make tool. It should be defined the environment variables to define the processor architecture and the cross compiler name and path:

```
cd u-boot-Digilent-Dev
source /opt/Xilinx/Vivado/2015.4/settings64.sh
export ARCH=arm
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
export LD_LIBRARY_PATH=
make distclean
make zynq_zybo_config
make
file u-boot
cp u-boot ../u-boot.elf
```

Once the compilation process is done, the u-boot file is created (without extension), it should be copied to a path where the *BOOT.BIN* generator can see it.

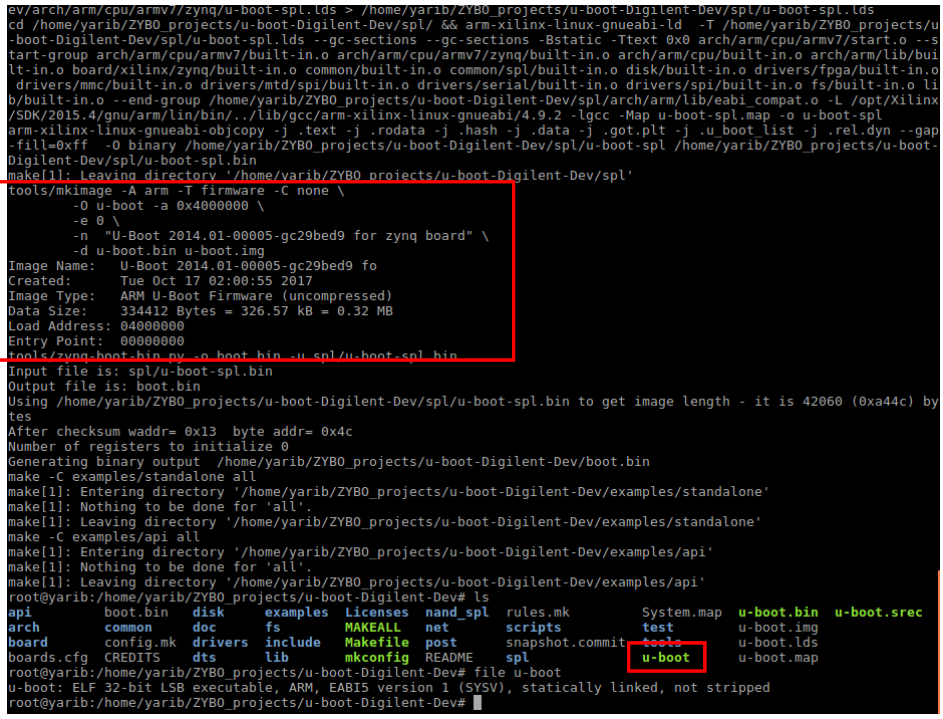

Figure 139: U-boot compilation

4. BOOT.BIN

Finally, the *BOOT.BIN* should be created containing the FSBL, Bitsream, and u-boot. It is used the tool from Xilinx SDK, in the menu Xilinx Tools, the option “Create Boot Image”. The boot images is created as shown in the following image.

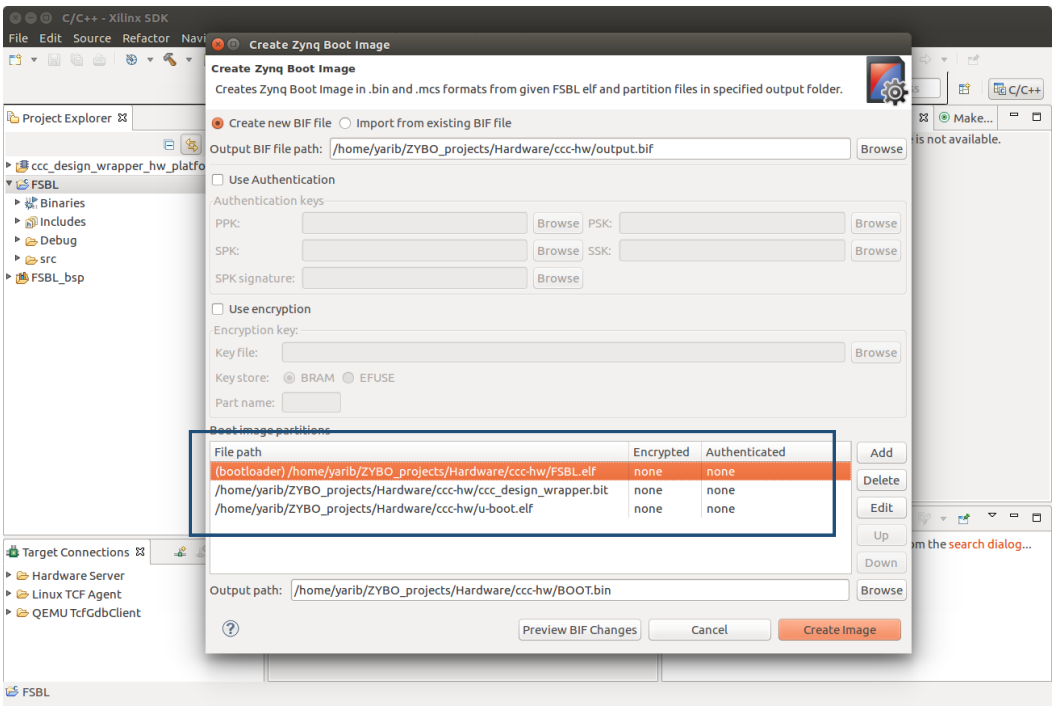

Figure 140: Zynq boot image tool

The files can be added with the “Add” button (it searches in the file system folders), the FSBL.elf is the Zynq bootlader, therefore its partition type is bootloader, and it should be the first in the list of files. The rest of the files are datafile partition type. For more detailed information it can be referred the Xilinx documentation.

Linux kernel image

The kernel is effectively the core of Linux operating system. The source code is obtained from the Digilent Git repository. [27]

Using git tool, the repository containing the source code is cloned with the following command:

```
git clone -b master-next https://github.com/DigilentInc/Linux-Digilent-Dev.git
```

After cloning the repository, this can be compiled with make tool. It should be declared environment variables to define the processor architecture and the cross compiler name and path:

```
cd Linux-Digilent-Dev
source /opt/Xilinx/Vivado/2015.4/settings64.sh
export ARCH=arm
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
export LD_LIBRARY_PATH=
make distclean
make xilinx_zynq_defconfig
make menuconfig
make
cd arch/arm/boot/
file zImage
PATH=$PATH:../ u-boot-Digilent-Dev/tools/
make UIMAGE_LOADADDR=0x8000 uImage
file uImage
```

Once the compilation process is done, the uImage file is created (without extension), it should be copied to the path where the *BOOT.BIN* is located. The following images show the Linux kernel Menuconfig and the kernel compilation.

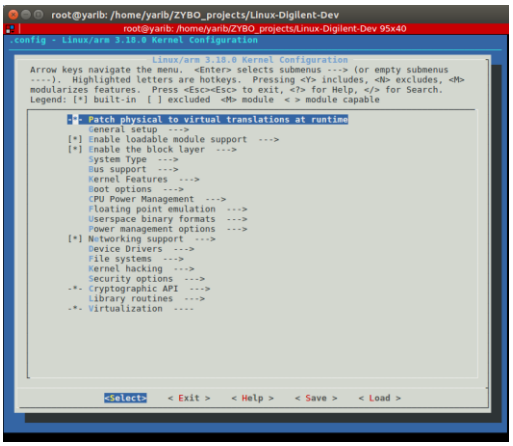

Figure 141: Linux kernel Menuconfig

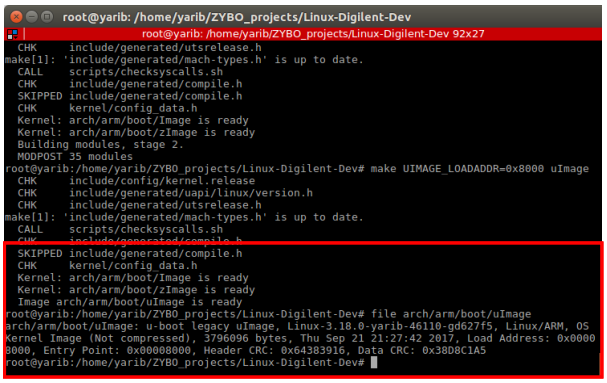

Figure 142: Linux kernel compilation

Device tree

The device tree contains detailed information of the hardware platform. The device tree for ZYBO is found in u-boot-Digilent-Dev repository. It should be added nodes for the new AXI peripherals. The entire device tree source is in Appendix B – Device tree.

```
/dts-v1/;

/ {
    #address-cells = <1>;
    #size-cells = <1>;
...
    chosen
    {
        bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk
rootfstype=ext4 rootwait devtmpfs.mount=0";
                /*bootargs = "console=ttyPS0,115200 root=/dev/ram rw initrd=0x800000,8M
earlyprintk";*/
        linux,stdout-path = "/amba@0/serial@e0001000";
    } ;
...
        adc
        {
            compatible = "yarib-adc-1.00.a";
            reg = <0x43C00000 0x10000>;
        } ;
        controller
        {
            compatible = "yarib-controller-1.00.a";
            reg = <0x41200000 0x10000>;
        } ;
        pwm_0
        {
            compatible = "yarib-pwm_0-1.00.a";
            reg = <0x42800000 0x10000>;
        } ;
        pwm_1
        {
            compatible = "yarib-pwm_1-1.00.a";
            reg = <0x42810000 0x10000>;
        } ;
...
} ;
```

The device tree source (.dts) should be compiled into the device tree blob (.dtb) using the device tree compiler (dtc) tool. The resulting file is named devicetree.dtb, it should be placed where the kernel image and the Zynq boot image are located.

```
./scripts/dtc/dtc -I dts -O dtb -o ../devicetree.dtb arch/arm/boot/dts/zynq-zybo.dts
```

The device tree can be edited to support Linaro file system, it should be modified in boot arguments:

```
bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1";
```

For RAM disk image (default):

```
bootargs = "console=ttyPS0,115200 root=/dev/ram rw initrd=0x800000,8M earlyprintk";
```

If the boot arguments are changed, the u-boot environment variables should be reset in u-boot command line:

```
env default –a –f
printenv sdboot
saveenv
```

Partition SD card for booting

The Zynq processor is booted from data contained on a SD card, and once the Linux environment is operational the root file system will also reside on the SD card. The ZYBO should be configured to boot-up from SD card by placing the jumper JP5 in SD.

The SD card contains two partitions. The first partition is FAT32 format, which can be accessed by either a Windows or Linux system. This partition contains the files used for initial bring-up of the Zynq device. The second partition is ext format, usable only by a Linux systems. On this partition is placed the root file system, required by the Linux kernel to complete the OS boot process.

The SD card partitioning is done using GParted tool. The next figure shows a 32GB SD card partitioning:

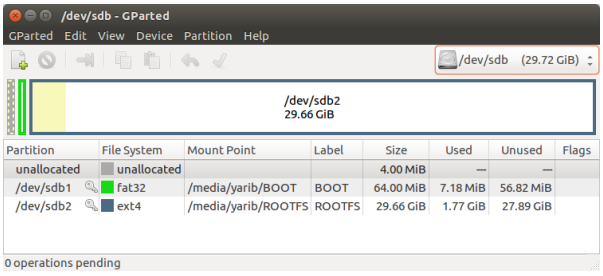

Figure 143: SD card partitioning

Once the SD card is properly partitioned, the booting files (*BOOT.BIN*, *uImage*, *devicetree.dtb*) are copied into the first partition FAT32, and the root file system or Linux distribution is copied into the second partition ext4.

The following figure shows the SD card image:

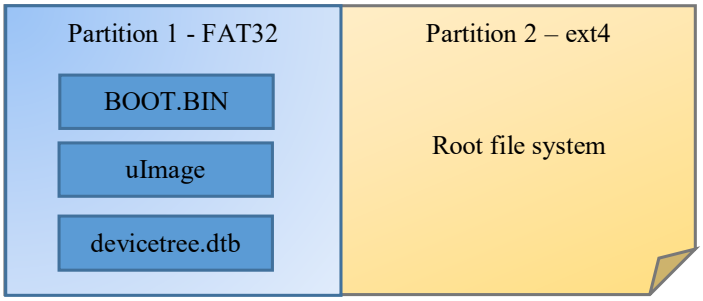

Figure 144: SD card image

Linux distribution

The Root File System (RFS) is contained on the same partition as the root directory in the Linux kernel, and it is the file system on which all other file systems are mounted. The RFS is separate from the Linux kernel, but is required by the kernel to complete the boot process. All file data used by Linux and users, are contained in the Root File System.

The separation of kernel and file system adds additional flexibility to Linux distributions. Distributions largely use a common Linux kernel (with slight variations to support specific elements important to the distribution), but the large part of what makes distributions unique is determined by the makeup of the root file system.

The creation of a complete Root File System from scratch is a complex procedure, and beyond the scope of this thesis.

It is best to start with a Root File System that contains much of what it is already needed for the platform, and then customize it to the specific requirements. Linaro provides a rich root file system that is open licence, which serves the purposes of this thesis. [28]

Linaro is an engineering organization that works on free and open-source software such as the Linux kernel, the GNU Compiler Collection (GCC), power management, graphics and multimedia interfaces for the ARM family of instruction sets and implementations thereof as well as for the Heterogeneous System Architecture. [29]

The latest release of Linaro Ubuntu developer is located at: http://releases.linaro.org/

http://releases.linaro.org/archive/12.11/ubuntu/precise-images/ubuntu-desktop/linaro-precise-ubuntu-desktop-20121124-560.tar.gz

The package should be decompressed with standard Linux tools, and the content should be copied in the second partition (ext4) of the SD card.

The following figure shows the SD card image:

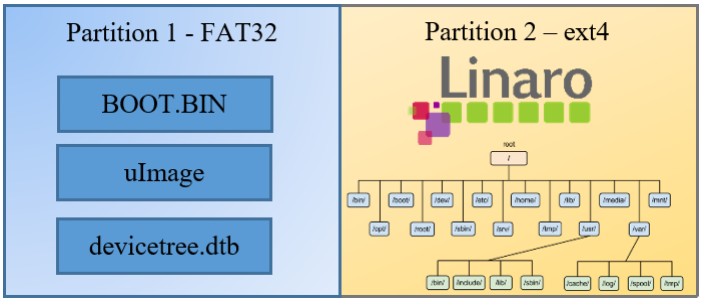

Figure 145: SD card image with Linaro

Debugging tools

The TCF agent is the preferred debugging tool. The repository containing the source code of the TCF agent should be downloaded and compiled in the Zynq environment, it can be cross compiled as well.

The ZYBO board should be connected to the internet through the Ethernet cable. In the serial terminal, the following commands will install the TCF agent:

```
sudo apt-get update
sudo apt-get install git uuid uuid-dev libssl-dev
git clone git://git.eclipse.org/gitroot/tcf/org.eclipse.tcf.agent.git
cd org.eclipse.tcf.agent/agent
make NO_SSL=1 NO_UUID=1
```

Once the debugger is compiled, it can be started with the following command:

```
obj/GNU/Linux/armv6l/Debug/agent -S
```

Appendix B Source code

The source codes attached in this Appendix are the SPI in VHDL, register access macros for the SPI, ZYBO device tree, and the Linux device drivers for the peripherals implemented on the FPGA side of the Zynq SoC and the external ADC.

The source code of the base application framework, the extended application framework, applications and tools are included as code fragments during the chapters of this thesis, the source code files are not included in this Appendix.

Enhanced SPI – VHDL

```
----------------------------------------------------------------------------------
-- Company: Hoschule Bremerhaven
-- Engineer: Yarib Nevárez <Yarib_007@hotmail.com>
--
-- Create Date: 21.11.2016 21:13:55
-- Design Name: ESPI
-- Module Name: espi - Behavioral
-- Revision: 2 Settle time.
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.std_logic_arith.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

ENTITY espi IS
    GENERIC (DATA_LENGTH_BIT_SIZE   : INTEGER := 2;
             SETTLE_TIME_SIZE       : INTEGER := 2;
             DATA_SIZE              : INTEGER := 32;
             BAUD_RATE_DIVIDER_SIZE : INTEGER := 8);
    PORT ( clk                : IN STD_LOGIC;
           reset              : IN STD_LOGIC;
           data_length        : IN STD_LOGIC_VECTOR (DATA_LENGTH_BIT_SIZE-1 DOWNTO 0);
           baud_rate_divider  : IN STD_LOGIC_VECTOR (BAUD_RATE_DIVIDER_SIZE-1 DOWNTO 0);
           settle_time        : IN STD_LOGIC_VECTOR (SETTLE_TIME_SIZE-1 DOWNTO 0);
           clock_polarity     : IN STD_LOGIC;
           clock_phase        : IN STD_LOGIC;
           start_transmission : IN STD_LOGIC;
           transmission_done  : OUT STD_LOGIC;
           data_tx  : IN  STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0);
           data_rx  : OUT STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0) := (others => '0');
           spi_clk  : OUT STD_LOGIC;
           spi_MOSI : OUT STD_LOGIC;
           spi_MISO : IN STD_LOGIC;
           spi_cs   : OUT STD_LOGIC);
END espi;

ARCHITECTURE Behavioral OF espi IS

TYPE SPI_STATE_TYPE IS (SPI_IDLE, SPI_REDY, SPI_CLK_IDLE, SPI_CLK_ACTIVE, SPI_STOP);
SIGNAL current_state, next_state: SPI_STATE_TYPE;

SIGNAL clk_pulse : STD_LOGIC;

SIGNAL i_tx_buffer : STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0) := (others => '0');
SIGNAL i_rx_buffer : STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0) := (others => '0');
SIGNAL counter : INTEGER RANGE 0 TO DATA_SIZE-1 := 0;
```

```
CONSTANT DATA_LENGTH_0 : INTEGER := 8;
CONSTANT DATA_LENGTH_1 : INTEGER := 16;
CONSTANT DATA_LENGTH_2 : INTEGER := 24;
CONSTANT DATA_LENGTH_3 : INTEGER := 32;

--TYPE DATA_LENGTH_ARRAY IS ARRAY (3 downto 0) OF INTEGER;

--CONSTANT DATA_LENGTHS : DATA_LENGTH_ARRAY := (DATA_LENGTH_3,
--                                              DATA_LENGTH_2,
--                                              DATA_LENGTH_1,
--                                              DATA_LENGTH_0);

BEGIN

    baud_rate_division_process: PROCESS (clk, reset, baud_rate_divider, settle_time,
clk_pulse)
    VARIABLE baud_rate_counter   : UNSIGNED (BAUD_RATE_DIVIDER_SIZE-1 DOWNTO 0) :=
(others => '0');
    VARIABLE settle_time_counter : UNSIGNED (SETTLE_TIME_SIZE-1 DOWNTO 0)       :=
(others => '0');
    BEGIN
        IF falling_edge(clk) THEN
            clk_pulse <= '0';
            IF reset = '1' OR current_state = SPI_IDLE THEN
                baud_rate_counter := (others => '0');
                settle_time_counter := (others => '0');
            ELSIF baud_rate_divider = CONV_STD_LOGIC_VECTOR(baud_rate_counter,
BAUD_RATE_DIVIDER_SIZE) THEN
                baud_rate_counter := (others => '0');
                ---------------------------------------------------------------
                IF current_state = SPI_REDY OR current_state = SPI_STOP THEN
                    IF settle_time = CONV_STD_LOGIC_VECTOR(settle_time_counter,
SETTLE_TIME_SIZE) THEN
                        clk_pulse <= '1';
                        settle_time_counter := (others => '0');
                    ELSE
                        settle_time_counter := settle_time_counter + 1;
                    END IF;
                ELSE
                    clk_pulse <= '1';
                END IF;
                ---------------------------------------------------------------
            ELSE
                baud_rate_counter := baud_rate_counter + 1;
            END IF;
        END IF;
    END PROCESS;

    spi_switch_state : PROCESS (clk, current_state, next_state)
    VARIABLE data_length_internal : UNSIGNED (1 DOWNTO 0);
    BEGIN
        IF rising_edge(clk) THEN
            IF reset = '1' THEN
                current_state <= SPI_IDLE;
                i_tx_buffer <= (others => '0');
                i_rx_buffer <= (others => '0');
                data_rx <= (others => '0');
            ELSIF next_state = SPI_REDY THEN -- Get redy immediately
                current_state <= SPI_REDY;
                data_rx <= (others => '0');
                i_rx_buffer <= (others => '0');

                CASE data_length IS
                    WHEN "00" =>
                     i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_0) <=
data_tx(DATA_LENGTH_0-1 downto 0);
                     counter <= DATA_LENGTH_0-1;
                    WHEN "01" =>
                     i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_1) <=
data_tx(DATA_LENGTH_1-1 downto 0);
                     counter <= DATA_LENGTH_1-1;
                    WHEN "10" =>
                     i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_2) <=
data_tx(DATA_LENGTH_2-1 downto 0);
                     counter <= DATA_LENGTH_2-1;
                    WHEN "11" =>
                     i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_3) <=
data_tx(DATA_LENGTH_3-1 downto 0);
                     counter <= DATA_LENGTH_3-1;
                    WHEN OTHERS => NULL;
```

```
            END CASE;
        ELSIF clk_pulse = '1' THEN       -- Or Wait for the pulse

            IF clock_phase = '0' THEN
                -- PUSH INPUT
                IF next_state = SPI_CLK_ACTIVE THEN
                    i_rx_buffer <= i_rx_buffer(DATA_SIZE-2 DOWNTO 0) & spi_MISO;
                END IF;
                -- POP OUTPUT
                IF next_state = SPI_CLK_IDLE THEN
                    i_tx_buffer <= i_tx_buffer(DATA_SIZE-2 downto 0) & '-';
                    counter <= counter - 1;
                END IF;
            END IF;

            IF clock_phase = '1' THEN
                -- PUSH INPUT
                IF current_state = SPI_CLK_ACTIVE THEN
                    i_rx_buffer <= i_rx_buffer(DATA_SIZE-2 DOWNTO 0) & spi_MISO;
                END IF;
                -- POP OUTPUT
                IF current_state = SPI_CLK_IDLE AND next_state = SPI_CLK_ACTIVE THEN
                    i_tx_buffer <= i_tx_buffer(DATA_SIZE-2 downto 0) & '-';
                    counter <= counter - 1;
                END IF;
            END IF;

            IF current_state = SPI_STOP THEN
                data_rx <= i_rx_buffer;
            END IF;

            current_state <= next_state;
        END IF;
    END IF;
END PROCESS;


    spi_mechanism :  process (next_state, clock_polarity, current_state,
start_transmission, i_tx_buffer, clock_phase, counter)
    BEGIN
        next_state <= current_state;
        spi_clk <= clock_polarity;
        spi_cs <= '1';
        spi_MOSI <= '0';
        transmission_done <= '1';

        CASE current_state IS
            WHEN SPI_IDLE =>
                IF start_transmission = '1' THEN
                    next_state <= SPI_REDY;
                END IF;
            WHEN SPI_REDY =>
                spi_cs <= '0';
                transmission_done <= '0';
                IF clock_phase = '0' THEN
                    spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                END IF;
                next_state <= SPI_CLK_ACTIVE;
            WHEN SPI_CLK_ACTIVE =>
                spi_cs <= '0';
                transmission_done <= '0';
                spi_clk <= NOT clock_polarity;
                spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                IF counter = 0 THEN
                    next_state <= SPI_STOP;
                ELSE
                    next_state <= SPI_CLK_IDLE;
                END IF;
            WHEN SPI_CLK_IDLE =>
                spi_cs <= '0';
                transmission_done <= '0';
                spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                IF counter = 0 AND clock_phase = '1' THEN
                    next_state <= SPI_STOP;
                ELSE
                    next_state <= SPI_CLK_ACTIVE;
                END IF;
            WHEN SPI_STOP =>
                IF clock_phase = '1' THEN
                    spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
```

```
                END IF;
                next_state <= SPI_IDLE;
                transmission_done <= '0';
                spi_cs <= '0';
        END CASE;
    END PROCESS;

END Behavioral;
```

Enhanced SPI register access macros

```
/*
 * espi.h
 *
 *  Created on: Sep 13, 2017
 *      Author: Yarib Nevárez
 */

#ifndef ESPI_HW_H_
#define ESPI_HW_H_

// Enhanced SPI driver

#define ACCESS_REGISTER(base, index)        (*((volatile uint32_t *)((base)+4*(index))))
#define REGISTER_GET(reg, mask, shift)      (((mask) & (reg)) >> (shift))
#define REGISTER_SET(reg, mask, shift, val) ((reg) = (~mask & (reg)) | ((mask) &
((val)<<(shift))))


#define SPI_BASEADDR                    XPAR_ESPI_0_0 //_AXI_BASEADDR !!

#define SPI_CONTROL_REGISTER_INDEX      0
#define SPI_CONTROL_REGISTER            ACCESS_REGISTER(SPI_BASEADDR,
SPI_CONTROL_REGISTER_INDEX)
#define SPI_DATA_REGISTER_INDEX         1
#define SPI_DATA                        ACCESS_REGISTER(SPI_BASEADDR,
SPI_DATA_REGISTER_INDEX)

#define SPI_BAUD_RATE_DIVIDER_MASK      0x000000FFu
#define SPI_BAUD_RATE_DIVIDER_SHIFT     0
#define SET_SPI_BAUD_RATE_DIVIDER(val)  REGISTER_SET(SPI_CONTROL_REGISTER,           \
                                                     SPI_BAUD_RATE_DIVIDER_MASK, \
                                                     SPI_BAUD_RATE_DIVIDER_SHIFT,\
                                                     val)
#define GET_SPI_BAUD_RATE_DIVIDER       REGISTER_GET(SPI_CONTROL_REGISTER,           \
                                                     SPI_BAUD_RATE_DIVIDER_MASK, \
                                                     SPI_BAUD_RATE_DIVIDER_SHIFT)


#define SPI_CLOCK_PHASE_MASK            0x00000100u
#define SPI_CLOCK_PHASE_SHIFT           8
#define SET_SPI_CLOCK_PHASE(val)        REGISTER_SET(SPI_CONTROL_REGISTER,     \
                                                     SPI_CLOCK_PHASE_MASK, \
                                                     SPI_CLOCK_PHASE_SHIFT,\
                                                     val)
#define GET_SPI_CLOCK_PHASE             REGISTER_GET(SPI_CONTROL_REGISTER,     \
                                                     SPI_CLOCK_PHASE_MASK, \
                                                     SPI_CLOCK_PHASE_SHIFT)


#define SPI_CLOCK_POLARITY_MASK         0x00000200u
#define SPI_CLOCK_POLARITY_SHIFT        9
#define SET_SPI_CLOCK_POLARITY(val)     REGISTER_SET(SPI_CONTROL_REGISTER,        \
                                                     SPI_CLOCK_POLARITY_MASK, \
                                                     SPI_CLOCK_POLARITY_SHIFT,\
                                                     val)
#define GET_SPI_CLOCK_POLARITY          REGISTER_GET(SPI_CONTROL_REGISTER,        \
                                                     SPI_CLOCK_POLARITY_MASK, \
                                                     SPI_CLOCK_POLARITY_SHIFT)

#define SPI_DATA_LENGTH_8_BITS          0
#define SPI_DATA_LENGTH_16_BITS         1
#define SPI_DATA_LENGTH_24_BITS         2
#define SPI_DATA_LENGTH_32_BITS         3

#define SPI_DATA_LENGTH_MASK            0x00000C00u
#define SPI_DATA_LENGTH_SHIFT           10
#define SET_SPI_DATA_LENGTH(val)        REGISTER_SET(SPI_CONTROL_REGISTER,     \
                                                     SPI_DATA_LENGTH_MASK, \
                                                     SPI_DATA_LENGTH_SHIFT,\
                                                     val)
#define GET_SPI_DATA_LENGTH             REGISTER_GET(SPI_CONTROL_REGISTER,     \
                                                     SPI_DATA_LENGTH_MASK, \
                                                     SPI_DATA_LENGTH_SHIFT)


#define SPI_CS_FORCE_MASK                   0x00001000u
#define SPI_CS_FORCE_SHIFT                  12
#define SET_SPI_CS_FORCE(val)               REGISTER_SET(SPI_CONTROL_REGISTER, \
```

```
                                                SPI_CS_FORCE_MASK,    \
                                                SPI_CS_FORCE_SHIFT,   \
                                                val)
#define GET_SPI_CS_FORCE                    REGISTER_GET(SPI_CONTROL_REGISTER, \
                                                SPI_CS_FORCE_MASK,    \
                                                SPI_CS_FORCE_SHIFT)


#define SPI_SETTLE_TIME_MASK            0x00006000u
#define SPI_SETTLE_TIME_SHIFT           13
#define SET_SPI_SETTLE_TIME(val)        REGISTER_SET(SPI_CONTROL_REGISTER,     \
                                                SPI_SETTLE_TIME_MASK, \
                                                SPI_SETTLE_TIME_SHIFT,\
                                                val)
#define GET_SPI_SETTLE_TIME             REGISTER_GET(SPI_CONTROL_REGISTER,     \
                                                SPI_SETTLE_TIME_MASK, \
                                                SPI_SETTLE_TIME_SHIFT)


#define SPI_TRANSMISSION_DONE_MASK          0x00008000u
#define SPI_TRANSMISSION_DONE_SHIFT         15
#define GET_SPI_TRANSMISSION_DONE           REGISTER_GET(SPI_CONTROL_REGISTER,       \
                                                SPI_TRANSMISSION_DONE_MASK, \
                                                SPI_TRANSMISSION_DONE_SHIFT)


#define SPI_SLAVE_SELECT_MASK           0x00030000u
#define SPI_SLAVE_SELECT_SHIFT          16
#define SET_SPI_SLAVE_SELECT(val)       REGISTER_SET(SPI_CONTROL_REGISTER,      \
                                                SPI_SLAVE_SELECT_MASK, \
                                                SPI_SLAVE_SELECT_SHIFT,\
                                                val)
#define GET_SPI_SLAVE_SELECT                REGISTER_GET(SPI_CONTROL_REGISTER,  \
                                                SPI_SLAVE_SELECT_MASK, \
                                                SPI_SLAVE_SELECT_SHIFT)


#endif /* ESPI_H_ */
```

Device tree

```
/*
 * Device Tree for Zybo board
 * Partially generated by Device Tree Generator 1.1
 *
 * (C) Copyright 2007-2013 Xilinx, Inc.
 * (C) Copyright 2007-2013 Michal Simek
 * (C) Copyright 2007-2012 PetaLogix Qld Pty Ltd
 * (C) Copyright 2014 Digilent, Inc.
 *
 * Michal SIMEK <monstr@monstr.eu>
 * Tinghui Wang <steven.wang@digilentinc.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 *
 */

/dts-v1/;
/ {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "xlnx,zynq-7000";
    model = "Xilinx Zynq";
    aliases {
        ethernet0 = &ps7_ethernet_0;
        serial0 = &ps7_uart_1;
        serial1 = &ps7_uart_0;
        spi0 = &ps7_qspi_0;
    } ;
    chosen {
        bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk
rootfstype=ext4 rootwait devtmpfs.mount=0";
                /*bootargs = "console=ttyPS0,115200 root=/dev/ram rw initrd=0x800000,8M
earlyprintk";*/
        linux,stdout-path = "/amba@0/serial@e0001000";
    } ;
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        ps7_cortexa9_0: cpu@0 {
            bus-handle = <&ps7_axi_interconnect_0>;
            clock-latency = <1000>;
            clocks = <&clkc 2>;
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            interrupt-handle = <&ps7_scugic_0>;
            operating-points = <666667 1000000 333334 1000000 222223 1000000>;
            reg = <0x0>;
        } ;
        ps7_cortexa9_1: cpu@1 {
            bus-handle = <&ps7_axi_interconnect_0>;
            clocks = <&clkc 2>;
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            interrupt-handle = <&ps7_scugic_0>;
            reg = <0x1>;
        } ;
    } ;
    pmu {
        compatible = "arm,cortex-a9-pmu";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 5 4>, <0 6 4>;
        reg = <0xf8891000 0x1000>, <0xf8893000 0x1000>;
        reg-names = "cpu0", "cpu1";
    } ;
    ps7_ddr_0: memory@0 {
```

```
        device_type = "memory";
        reg = <0x0 0x20000000>;
    } ;
    ps7_axi_interconnect_0: amba@0 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
        ranges ;
        ps7_afi_0: ps7-afi@f8008000 {
            compatible = "xlnx,ps7-afi-1.00.a";
            reg = <0xf8008000 0x1000>;
        } ;
        ps7_afi_1: ps7-afi@f8009000 {
            compatible = "xlnx,ps7-afi-1.00.a";
            reg = <0xf8009000 0x1000>;
        } ;
        ps7_afi_2: ps7-afi@f800a000 {
            compatible = "xlnx,ps7-afi-1.00.a";
            reg = <0xf800a000 0x1000>;
        } ;
        ps7_afi_3: ps7-afi@f800b000 {
            compatible = "xlnx,ps7-afi-1.00.a";
            reg = <0xf800b000 0x1000>;
        } ;
        ps7_ddrc_0: ps7-ddrc@f8006000 {
            compatible = "xlnx,zynq-ddrc-1.0";
            reg = <0xf8006000 0x1000>;
            xlnx,has-ecc = <0x0>;
        } ;
        ps7_dev_cfg_0: ps7-dev-cfg@f8007000 {
            clock-names = "ref_clk", "fclk0", "fclk1", "fclk2", "fclk3";
            clocks = <&clkc 12>, <&clkc 15>, <&clkc 16>, <&clkc 17>, <&clkc 18>;
            compatible = "xlnx,zynq-devcfg-1.0";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <0 8 4>;
            reg = <0xf8007000 0x100>;
        } ;
        ps7_dma_s: ps7-dma@f8003000 {
            #dma-cells = <1>;
            #dma-channels = <8>;
            #dma-requests = <4>;
            clock-names = "apb_pclk";
            clocks = <&clkc 27>;
            compatible = "arm,primecell", "arm,pl330";
            interrupt-names = "abort", "dma0", "dma1", "dma2", "dma3",
                "dma4", "dma5", "dma6", "dma7";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <0 13 4>, <0 14 4>, <0 15 4>, <0 16 4>, <0 17 4>, <0 40 4>, <0
41 4>, <0 42 4>, <0 43 4>;
            reg = <0xf8003000 0x1000>;
        } ;
        ps7_ethernet_0: ps7-ethernet@e000b000 {
            #address-cells = <1>;
            #size-cells = <0>;
            clock-names = "ref_clk", "aper_clk";
            clocks = <&clkc 13>, <&clkc 30>;
            compatible = "xlnx,ps7-ethernet-1.00.a";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <0 22 4>;
            phy-handle = <&phy0>;
            phy-mode = "rgmii-id";
            reg = <0xe000b000 0x1000>;
            xlnx,eth-mode = <0x1>;
            xlnx,has-mdio = <0x1>;
            xlnx,ptp-enet-clock = <108333336>;
            mdio {
                #address-cells = <1>;
                #size-cells = <0>;
                phy0: phy@1 {
                    compatible = "realtek,RTL8211E";
                    device_type = "ethernet-phy";
                    reg = <1>;
                } ;
            } ;
        } ;
        ps7_globaltimer_0: ps7-globaltimer@f8f00200 {
            clocks = <&clkc 4>;
            compatible = "arm,cortex-a9-global-timer";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <1 11 0x301>;
```

```
        reg = <0xf8f00200 0x100>;
    } ;
    ps7_gpio_0: ps7-gpio@e000a000 {
        #gpio-cells = <2>;
        clocks = <&clkc 42>;
        compatible = "xlnx,zynq-gpio-1.0";
        emio-gpio-width = <64>;
        gpio-controller ;
        gpio-mask-high = <0xc0000>;
        gpio-mask-low = <0xfe81>;
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 20 4>;
        reg = <0xe000a000 0x1000>;
    } ;
    ps7_iop_bus_config_0: ps7-iop-bus-config@e0200000 {
        compatible = "xlnx,ps7-iop-bus-config-1.00.a";
        reg = <0xe0200000 0x1000>;
    } ;
    ps7_ocmc_0: ps7-ocmc@f800c000 {
        compatible = "xlnx,zynq-ocmc-1.0";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 3 4>;
        reg = <0xf800c000 0x1000>;
    } ;
    ps7_pl310_0: ps7-pl310@f8f02000 {
        arm,data-latency = <3 2 2>;
        arm,tag-latency = <2 2 2>;
        cache-level = <2>;
        cache-unified ;
        compatible = "arm,pl310-cache";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 2 4>;
        reg = <0xf8f02000 0x1000>;
    } ;
    ps7_qspi_0: ps7-qspi@e000d000 {
        clock-names = "ref_clk", "pclk";
        clocks = <&clkc 10>, <&clkc 43>;
        compatible = "xlnx,zynq-qspi-1.0";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 19 4>;
        is-dual = <0>;
        num-cs = <1>;
        reg = <0xe000d000 0x1000>;
        xlnx,fb-clk = <0x1>;
        xlnx,qspi-mode = <0x0>;
        #address-cells = <1>;
        #size-cells = <0>;
        flash@0 {
            compatible = "n25q128";
            reg = <0x0>;
            spi-tx-bus-width = <1>;
            spi-rx-bus-width = <4>;
            spi-max-frequency = <50000000>;
            #address-cells = <1>;
            #size-cells = <1>;
            partition@qspi-fsbl-uboot {
                label = "qspi-fsbl-uboot";
                reg = <0x0 0x400000>;
            };
            partition@qspi-linux {
                label = "qspi-linux";
                reg = <0x400000 0x500000>;
            };
            partition@qspi-device-tree {
                label = "qspi-device-tree";
                reg = <0x900000 0x20000>;
            };
            partition@qspi-user {
                label = "qspi-user";
                reg = <0x920000 0x6E0000>;
            };
        };

    } ;
    ps7_qspi_linear_0: ps7-qspi-linear@fc000000 {
        clock-names = "ref_clk", "aper_clk";
        clocks = <&clkc 10>, <&clkc 43>;
        compatible = "xlnx,ps7-qspi-linear-1.00.a";
        reg = <0xfc000000 0x1000000>;
    } ;
```

```
        ps7_scugic_0: ps7-scugic@f8f01000 {
            #address-cells = <2>;
            #interrupt-cells = <3>;
            #size-cells = <1>;
            compatible = "arm,cortex-a9-gic", "arm,gic";
            interrupt-controller ;
            num_cpus = <2>;
            num_interrupts = <96>;
            reg = <0xf8f01000 0x1000>, <0xf8f00100 0x100>;
        } ;
        ps7_scutimer_0: ps7-scutimer@f8f00600 {
            clocks = <&clkc 4>;
            compatible = "arm,cortex-a9-twd-timer";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <1 13 0x301>;
            reg = <0xf8f00600 0x20>;
        } ;
        ps7_scuwdt_0: ps7-scuwdt@f8f00620 {
            clocks = <&clkc 4>;
            compatible = "xlnx,ps7-scuwdt-1.00.a";
            device_type = "watchdog";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <1 14 0x301>;
            reg = <0xf8f00620 0xe0>;
        } ;
        ps7_sd_0: ps7-sdio@e0100000 {
            clock-frequency = <50000000>;
            clock-names = "clk_xin", "clk_ahb";
            clocks = <&clkc 21>, <&clkc 32>;
            compatible = "arasan,sdhci-8.9a";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <0 24 4>;
            reg = <0xe0100000 0x1000>;
            xlnx,has-cd = <0x1>;
            xlnx,has-power = <0x0>;
            xlnx,has-wp = <0x1>;
        } ;
        ps7_slcr_0: ps7-slcr@f8000000 {
            #address-cells = <1>;
            #size-cells = <1>;
            compatible = "xlnx,zynq-slcr", "syscon";
            ranges ;
            reg = <0xf8000000 0x1000>;
            clkc: clkc@100 {
                #clock-cells = <1>;
                clock-output-names = "armpll", "ddrpll", "iopll", "cpu_6or4x",
"cpu_3or2x",
                    "cpu_2x", "cpu_1x", "ddr2x", "ddr3x", "dci",
                    "lqspi", "smc", "pcap", "gem0", "gem1",
                    "fclk0", "fclk1", "fclk2", "fclk3", "can0",
                    "can1", "sdio0", "sdio1", "uart0", "uart1",
                    "spi0", "spi1", "dma", "usb0_aper", "usb1_aper",
                    "gem0_aper", "gem1_aper", "sdio0_aper", "sdio1_aper", "spi0_aper",
                    "spi1_aper", "can0_aper", "can1_aper", "i2c0_aper", "i2c1_aper",
                    "uart0_aper", "uart1_aper", "gpio_aper", "lqspi_aper", "smc_aper",
                    "swdt", "dbg_trc", "dbg_apb";
                compatible = "xlnx,ps7-clkc";
                fclk-enable = <0xf>;
                ps-clk-frequency = <50000000>;
                reg = <0x100 0x100>;
            } ;
        } ;
        ps7_ttc_0: ps7-ttc@f8001000 {
            clocks = <&clkc 6>;
            compatible = "cdns,ttc";
            interrupt-names = "ttc0", "ttc1", "ttc2";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <0 10 4>, <0 11 4>, <0 12 4>;
            reg = <0xf8001000 0x1000>;
        } ;
        ps7_uart_1: serial@e0001000 {
            clock-names = "uart_clk", "pclk";
            clocks = <&clkc 24>, <&clkc 41>;
            compatible = "xlnx,xuartps", "cdns,uart-r1p8";
            current-speed = <115200>;
            device_type = "serial";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <0 50 4>;
            port-number = <0>;
            reg = <0xe0001000 0x1000>;
```

```
            xlnx,has-modem = <0x0>;
        } ;
        ps7_uart_0: serial@e0000000 {
            clock-names = "uart_clk", "pclk";
            clocks = <&clkc 23>, <&clkc 40>;
            compatible = "xlnx,xuartps", "cdns,uart-r1p8";
            current-speed = <9600>;
            device_type = "serial";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <0 27 4>;
            reg = <0xE0000000 0x1000>;
        };
        ps7_usb_0: ps7-usb@e0002000 {
            clocks = <&clkc 28>;
            compatible = "xlnx,ps7-usb-1.00.a", "xlnx,zynq-usb-1.00.a";
            dr_mode = "host";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <0 21 4>;
            phy_type = "ulpi";
            reg = <0xe0002000 0x1000>;
            xlnx,usb-reset = "MIO 46";
        } ;
        ps7_xadc: ps7-xadc@f8007100 {
            clocks = <&clkc 12>;
            compatible = "xlnx,zynq-xadc-1.00.a";
            interrupt-parent = <&ps7_scugic_0>;
            interrupts = <0 7 4>;
            reg = <0xf8007100 0x20>;
        } ;
        zybo
        {
            compatible = "yarib-zybo-1.00.a";
            reg = <0x41210000 0x10000>;
        } ;
        adc
        {
            compatible = "yarib-adc-1.00.a";
            reg = <0x43C00000 0x10000>;
        } ;
        controller
        {
            compatible = "yarib-controller-1.00.a";
            reg = <0x41200000 0x10000>;
        } ;
        pwm_0
        {
            compatible = "yarib-pwm_0-1.00.a";
            reg = <0x42800000 0x10000>;
        } ;
        pwm_1
        {
            compatible = "yarib-pwm_1-1.00.a";
            reg = <0x42810000 0x10000>;
        } ;
    } ;
} ;
```

Controller

```
/*
 * AXI GPIO v2.0 - Linux driver
 *
 * Yarib Nevárez <yarib_007@hotmail.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

#include <linux/kernel.h>
#include <asm/uaccess.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <asm/uaccess.h>    /* Needed for copy_from_user */
#include <asm/io.h>         /* Needed for IO Read/Write Functions */
#include <linux/proc_fs.h>  /* Needed for Proc File System Functions */
#include <linux/seq_file.h> /* Needed for Sequence File Operations */
#include <linux/platform_device.h>  /* Needed for Platform Driver Functions */
#include <asm/uaccess.h>

#include <linux/ctype.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/fcntl.h>

#define KERNEL_MODULE
#include "../../app-workspace/base_framework/device/iodef.hpp"
#include "../../app-workspace/extension_framework/deviceid.hpp"

/* Define Driver Name */
#define DRIVER_NAME "controller"

#define SUCCESS     0


static unsigned long *   base_addr   = NULL;
static struct resource * res         = NULL;
static unsigned long     remap_size  = 0;
static DeviceID          device_ID   = 0;
static u32               output_data = 0;

static int major;       /* major number we get from the kernel */

static void configport(u32 data)
{
    static u32 current_data = 0;

    if (current_data != data)
    {
        wmb();
        iowrite32(data, base_addr + 1);
        current_data = data;
    }
}

static void outport(u32 data)
{
    static u32 current_data = 0;

    if (current_data != data)
    {
        wmb();
        iowrite32(data, base_addr);
        current_data = data;
    }
}
```

```
static u32 inport(void)
{
    wmb();
    return ioread32(base_addr);
}

#define SET_SLICE_32(register, value, offset, mask) (register) = (((register) & ~((mask)
<< (offset))) | (((value) & (mask)) << (offset)))
#define GET_SLICE_32(register, offset, mask) (((register) & ((mask) << (offset))) >>
(offset))

/* Write operation for /proc/driver
* ----------------------------------
*/
static ssize_t proc_driver_write(struct file *file,
                                 const char __user * buffer,
                                 size_t buffer_size,
                                 loff_t * position)
{
    ssize_t written_size = 0;

    if (buffer != NULL)
    {
        if (buffer_size == sizeof(device_ID))
        {
            written_size = buffer_size - copy_from_user((void *) &device_ID, buffer,
sizeof(device_ID));
        }
        else if (buffer_size == sizeof(IOPacket))
        {
            IOPacket packet;
            written_size = buffer_size - copy_from_user((void *) &packet, buffer,
sizeof(IOPacket));

            if (written_size == buffer_size )
            {
                device_ID = packet.device_ID;
                switch (packet.device_ID)
                {
                    case IRSENSOR:
                        break;
                    case FLUSHVALVE:
                        SET_SLICE_32(output_data, packet.data, 16, 0x1);
                        break;
                    case DRAINVALVE:
                        SET_SLICE_32(output_data, packet.data, 17, 0x1);
                        break;
                    case SHUTOFFVALVE:
                        SET_SLICE_32(output_data, packet.data, 18, 0x1);
                        break;
                    case DRAINLOCAL:
                        break;
                    case EMERGENCY:
                        break;
                    case APPSELECTION:
                        break;
                    case DRAINDELAY:
                        break;
                    case VACUUMGEN:
                        SET_SLICE_32(output_data, packet.data, 19, 0x1);
                        break;
                    case DRAININDICATOR:
                        SET_SLICE_32(output_data, packet.data, 20, 0x1);
                        break;
                    case LEAKINDICATOR:
                        SET_SLICE_32(output_data, packet.data, 21, 0x1);
                        break;
                    case RELAY_0:
                        SET_SLICE_32(output_data, packet.data, 22, 0x1);
                        break;
                    case RELAY_1:
                        SET_SLICE_32(output_data, packet.data, 23, 0x1);
                        break;
                    default:;
                }
                outport(output_data);
                * position += written_size;
            }
        }
```

```
    }
    return written_size;
}

/* Read operation for /proc/driver
* ---------------------------------
*/
static ssize_t proc_driver_read(struct file *file,
                                char __user * buffer,
                                size_t buffer_size,
                                loff_t * position)
{
    ssize_t read_size = 0;

    if ((buffer != NULL) && (sizeof(IOPacket) <= buffer_size))
    {
        IOPacket packet;
        u32 input_data = inport();
        packet.device_ID = device_ID;

        switch (device_ID)
        {
            case FLUSHVALVE:
                packet.data = GET_SLICE_32(input_data, 0, 1);
                break;
            case DRAINVALVE:
                packet.data = GET_SLICE_32(input_data, 1, 1);
                break;
            case SHUTOFFVALVE:
                packet.data = GET_SLICE_32(input_data, 2, 1);
                break;
            case VACUUMGEN:
                packet.data = GET_SLICE_32(input_data, 3, 1);
                break;
            case IRSENSOR:
                packet.data = GET_SLICE_32(input_data, 5, 1);
                break;
            case DRAINLOCAL:
                packet.data = GET_SLICE_32(input_data, 6, 1);
                break;
            case EMERGENCY:
                packet.data = GET_SLICE_32(input_data, 7, 3);
                break;
            case APPSELECTION:
                packet.data = GET_SLICE_32(input_data, 9, 1);
                break;
            case DRAINDELAY:
                packet.data = GET_SLICE_32(input_data, 10, 7);
                break;
            case DRAININDICATOR:
                packet.data = GET_SLICE_32(input_data, 20, 1);
                break;
            case LEAKINDICATOR:
                packet.data = GET_SLICE_32(input_data, 21, 1);
                break;
            case RELAY_0:
                packet.data = GET_SLICE_32(input_data, 22, 1);
                break;
            case RELAY_1:
                packet.data = GET_SLICE_32(input_data, 23, 1);
                break;
            default:;
        }
        read_size = buffer_size - copy_to_user(buffer, (void *) &packet,
sizeof(IOPacket));
    }
    return read_size;
}


/* Callback function when opening file /proc/driver
* ------------------------------------------------------
*  Read the register value of driver file controller, print the value to
*  the sequence file struct seq_file *p. In file open operation for /proc/driver
*  this callback function will be called first to fill up the seq_file,
*  and seq_read function will print whatever in seq_file to the terminal.
*/
static int proc_driver_show(struct seq_file *p, void *v)
{
    seq_printf(p, "\nController kernel module\n");
```

```
    return SUCCESS;
}

/* Open function for /proc/driver
* ------------------------------------
*  When user want to read /proc/driver (i.e. cat /proc/driver), the open function
*  will be called first. In the open function, a seq_file will be prepared and the
*  status of driver will be filled into the seq_file by proc_driver_show function.
*/
static int driver_open(struct inode *inode, struct file *file)
{
    unsigned int size = 16;
    char * buf;
    struct seq_file * m;
    int rc = SUCCESS;

    buf = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
    if (buf != NULL)
    {
        rc = single_open(file, proc_driver_show, NULL);

        if (rc == SUCCESS)
        {
            m = file->private_data;
            m->buf = buf;
            m->size = size;
        }
        else
        {
            kfree(buf);
        }
    }
    else
    {
        printk(KERN_ALERT DRIVER_NAME "%s No memory resource\n", __FUNCTION__);
        rc = -ENOMEM;
    }

    return rc;
}

/* File Operations for /proc/driver */
static const struct file_operations proc_driver_operations =
{
    .open           = driver_open,
    .write          = proc_driver_write,
    .read           = proc_driver_read,
    .llseek         = seq_lseek,
    .release        = single_release
};

/* Shutdown function for driver
* ----------------------------------
*  Before driver shutdown, turn-off all the stuff
*/
static void driver_shutdown(struct platform_device *pdev)
{
    iowrite32(0, base_addr);
}

/* Remove function for driver
* ---------------------------------
*  When driver module is removed, turn off all the stuff first,
*  release virtual address and the memory region requested.
*/
static int driver_remove(struct platform_device *pdev)
{
    driver_shutdown(pdev);

    /* Remove /proc/driver entry */
    remove_proc_entry(DRIVER_NAME, NULL);

    unregister_chrdev(major, DRIVER_NAME);

    /* Release mapped virtual address */
    iounmap(base_addr);

    /* Release the region */
    release_mem_region(res->start, remap_size);
```

```
    return SUCCESS;
}

/* Device Probe function for driver
* ------------------------------------
*  Get the resource structure from the information in device tree.
*  request the memory regioon needed for the controller, and map it into
*  kernel virtual memory space. Create an entry under /proc file system
*  and register file operations for that entry.
*/
static int driver_probe(struct platform_device *pdev)
{
    struct proc_dir_entry * driver_proc_entry;
    int rc = SUCCESS;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res)
    {
        dev_err(&pdev->dev, "No memory resource\n");
        rc = -ENODEV;
    }

    if (rc == SUCCESS)
    {
        remap_size = res->end - res->start + 1;
        if (!request_mem_region(res->start, remap_size, pdev->name))
        {
            dev_err(&pdev->dev, "Cannot request IO\n");
            rc = -ENXIO;
        }
    }

    if (rc == SUCCESS)
    {
        base_addr = ioremap(res->start, remap_size);
        if (base_addr == NULL)
        {
            dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n", (unsigned
long)res->start);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        driver_proc_entry = proc_create(DRIVER_NAME, 0, NULL, &proc_driver_operations);
        if (driver_proc_entry == NULL)
        {
            dev_err(&pdev->dev, "Couldn't create proc entry\n");
            iounmap(base_addr);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        printk(KERN_INFO DRIVER_NAME " Mapped at virtual address 0x%08lx\n", (unsigned
long) base_addr);

        major = register_chrdev(0, DRIVER_NAME, &proc_driver_operations);

        configport(0x0000FFFF);
        iowrite32(0x000000, base_addr);
    }

    return rc;
}

/* device match table to match with device node in device tree */
static const struct of_device_id zynq_pmod_match[] =
{
    {.compatible = "yarib-controller-1.00.a"},
    { },
};

MODULE_DEVICE_TABLE(of, zynq_pmod_match);

/* platform driver structure for device driver */
```

```
static struct platform_driver zynq_pmod_driver =
{
    .driver =
    {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = zynq_pmod_match
    },
    .probe = driver_probe,
    .remove = driver_remove,
    .shutdown = driver_shutdown
};

/* Register device platform driver */
module_platform_driver(zynq_pmod_driver);

/* Module Informations */
MODULE_AUTHOR("Yarib, Inc.");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION(DRIVER_NAME ": Controller module");
MODULE_ALIAS(DRIVER_NAME);
```

ZYBO

```
/*
 * AXI GPIO v2.0 - Linux driver
 *
 * Yarib Nevárez <yarib_007@hotmail.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

#include <linux/kernel.h>
#include <asm/uaccess.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <asm/uaccess.h>    /* Needed for copy_from_user */
#include <asm/io.h>         /* Needed for IO Read/Write Functions */
#include <linux/proc_fs.h>  /* Needed for Proc File System Functions */
#include <linux/seq_file.h> /* Needed for Sequence File Operations */
#include <linux/platform_device.h>  /* Needed for Platform Driver Functions */
#include <asm/uaccess.h>

#include <linux/ctype.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/fcntl.h>

#define KERNEL_MODULE
#include "../../app-workspace/base_framework/device/iodef.hpp"
#include "../../app-workspace/extension_framework/deviceid.hpp"

/* Define Driver Name */
#define DRIVER_NAME "zybo"

#define SUCCESS     0


static unsigned long *   base_addr   = NULL;
static struct resource * res         = NULL;
static unsigned long     remap_size  = 0;
static DeviceID          device_ID   = 0;
static u32               output_data = 0;

static int major;       /* major number we get from the kernel */

// static void configport(u32 data, u32 offset)
// {
//     static u32 current_data = 0;

//     if (current_data != data)
//     {
//         wmb();
//         iowrite32(data, base_addr + offset);
//         current_data = data;
//     }
// }

static void outport(u32 data, u32 offset)
{
    static u32 current_data = 0;

    if (current_data != data)
    {
        wmb();
        iowrite32(data, base_addr + offset);
        current_data = data;
    }
}

static u32 inport(u32 offset)
{
    wmb();
    return ioread32(base_addr + offset);
```

```
}

#define SET_SLICE_32(register, value, offset, mask) (register) = (((register) & ~((mask)
<< (offset))) | (((value) & (mask)) << (offset)))
#define GET_SLICE_32(register, offset, mask) (((register) & ((mask) << (offset))) >>
(offset))

/* Write operation for /proc/driver
* ----------------------------------
*/
static ssize_t proc_driver_write(struct file *file,
                                 const char __user * buffer,
                                 size_t buffer_size,
                                 loff_t * position)
{
    ssize_t written_size = 0;

    if (buffer != NULL)
    {
        if (buffer_size == sizeof(device_ID))
        {
            written_size = buffer_size - copy_from_user((void *) &device_ID, buffer,
sizeof(device_ID));
        }
        else if (buffer_size == sizeof(IOPacket))
        {
            IOPacket packet;
            written_size = buffer_size - copy_from_user((void *) &packet, buffer,
sizeof(IOPacket));

            if (written_size == buffer_size )
            {
                device_ID = packet.device_ID;
                switch (packet.device_ID)
                {
                    case ZYBO_BUTTONS:
                        break;
                    case ZYBO_SWITCHES:
                        break;
                    case ZYBO_LEDS:
                        SET_SLICE_32(output_data, packet.data, 0, 0xF);
                        break;
                    default:;
                }
                outport(output_data, 2);
                * position += written_size;
            }
        }
    }
    return written_size;
}

/* Read operation for /proc/driver
* ----------------------------------
*/
static ssize_t proc_driver_read(struct file *file,
                                char __user * buffer,
                                size_t buffer_size,
                                loff_t * position)
{
    ssize_t read_size = 0;

    if ((buffer != NULL) && (sizeof(IOPacket) <= buffer_size))
    {
        IOPacket packet;
        u32 input_data = inport(0);
        packet.device_ID = device_ID;

        switch (device_ID)
        {
            case ZYBO_BUTTONS:
                packet.data = GET_SLICE_32(input_data, 0, 0xF);
                break;
            case ZYBO_SWITCHES:
                packet.data = GET_SLICE_32(input_data, 4, 0xF);
                break;
            case ZYBO_LEDS:
                packet.data = GET_SLICE_32(output_data, 0, 0xF);
                break;
            default:;
```

```
        }
        read_size = buffer_size - copy_to_user(buffer, (void *) &packet,
sizeof(IOPacket));
    }
    return read_size;
}


/* Callback function when opening file /proc/driver
* ----------------------------------------------------
*  Read the register value of driver file controller, print the value to
*  the sequence file struct seq_file *p. In file open operation for /proc/driver
*  this callback function will be called first to fill up the seq_file,
*  and seq_read function will print whatever in seq_file to the terminal.
*/
static int proc_driver_show(struct seq_file *p, void *v)
{
    seq_printf(p, "\nzybo kernel module\n");
    return SUCCESS;
}

/* Open function for /proc/driver
* ----------------------------------
*  When user want to read /proc/driver (i.e. cat /proc/driver), the open function
*  will be called first. In the open function, a seq_file will be prepared and the
*  status of driver will be filled into the seq_file by proc_driver_show function.
*/
static int driver_open(struct inode *inode, struct file *file)
{
    unsigned int size = 16;
    char * buf;
    struct seq_file * m;
    int rc = SUCCESS;

    buf = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
    if (buf != NULL)
    {
        rc = single_open(file, proc_driver_show, NULL);

        if (rc == SUCCESS)
        {
            m = file->private_data;
            m->buf = buf;
            m->size = size;
        }
        else
        {
            kfree(buf);
        }
    }
    else
    {
        printk(KERN_ALERT DRIVER_NAME "%s No memory resource\n", __FUNCTION__);
        rc = -ENOMEM;
    }

    return rc;
}

/* File Operations for /proc/driver */
static const struct file_operations proc_driver_operations =
{
    .open           = driver_open,
    .write          = proc_driver_write,
    .read           = proc_driver_read,
    .llseek         = seq_lseek,
    .release        = single_release
};

/* Shutdown function for driver
* ---------------------------------
*  Before driver shutdown, turn-off all the stuff
*/
static void driver_shutdown(struct platform_device *pdev)
{
    output_data = 0;
    outport(output_data, 2);
}

/* Remove function for driver
```

```
* ---------------------------------
*  When driver module is removed, turn off all the stuff first,
*  release virtual address and the memory region requested.
*/
static int driver_remove(struct platform_device *pdev)
{
    driver_shutdown(pdev);

    /* Remove /proc/driver entry */
    remove_proc_entry(DRIVER_NAME, NULL);

    unregister_chrdev(major, DRIVER_NAME);

    /* Release mapped virtual address */
    iounmap(base_addr);

    /* Release the region */
    release_mem_region(res->start, remap_size);

    return SUCCESS;
}

/* Device Probe function for driver
* ---------------------------------
*  Get the resource structure from the information in device tree.
*  request the memory regioon needed for the controller, and map it into
*  kernel virtual memory space. Create an entry under /proc file system
*  and register file operations for that entry.
*/
static int driver_probe(struct platform_device *pdev)
{
    struct proc_dir_entry * driver_proc_entry;
    int rc = SUCCESS;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res)
    {
        dev_err(&pdev->dev, "No memory resource\n");
        rc = -ENODEV;
    }

    if (rc == SUCCESS)
    {
        remap_size = res->end - res->start + 1;
        if (!request_mem_region(res->start, remap_size, pdev->name))
        {
            dev_err(&pdev->dev, "Cannot request IO\n");
            rc = -ENXIO;
        }
    }

    if (rc == SUCCESS)
    {
        base_addr = ioremap(res->start, remap_size);
        if (base_addr == NULL)
        {
            dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n", (unsigned
long)res->start);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        driver_proc_entry = proc_create(DRIVER_NAME, 0, NULL, &proc_driver_operations);
        if (driver_proc_entry == NULL)
        {
            dev_err(&pdev->dev, "Couldn't create proc entry\n");
            iounmap(base_addr);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        printk(KERN_INFO DRIVER_NAME " Maped at virtual address 0x%08lx\n", (unsigned
long) base_addr);
```

```
        major = register_chrdev(0, DRIVER_NAME, &proc_driver_operations);

        output_data = 0;
        outport(output_data, 2);
    }

    return rc;
}

/* device match table to match with device node in device tree */
static const struct of_device_id zynq_pmod_match[] =
{
    {.compatible = "yarib-zybo-1.00.a"},
    { },
};

MODULE_DEVICE_TABLE(of, zynq_pmod_match);

/* platform driver structure for device driver */
static struct platform_driver zynq_pmod_driver =
{
    .driver =
    {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = zynq_pmod_match
    },
    .probe = driver_probe,
    .remove = driver_remove,
    .shutdown = driver_shutdown
};

/* Register device platform driver */
module_platform_driver(zynq_pmod_driver);

/* Module Informations */
MODULE_AUTHOR("Yarib, Inc.");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION(DRIVER_NAME ": zybo module");
MODULE_ALIAS(DRIVER_NAME);
```

ADC

```
/*
 * MAX11632 - Maxim 12-Bit SPI ADC Linux driver
 *
 * Yarib Nevárez <yarib_007@hotmail.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

#include <linux/kernel.h>
#include <asm/uaccess.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <asm/uaccess.h>    /* Needed for copy_from_user */
#include <asm/io.h>         /* Needed for IO Read/Write Functions */
#include <linux/proc_fs.h>  /* Needed for Proc File System Functions */
#include <linux/seq_file.h> /* Needed for Sequence File Operations */
#include <linux/platform_device.h>  /* Needed for Platform Driver Functions */
#include <asm/uaccess.h>

#include <linux/ctype.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/fcntl.h>

#include "../../Hardware/ip_repo/ESPI_1.0/src/espi_hw.h"
#define KERNEL_MODULE
#include "../../app-workspace/base_framework/device/iodef.hpp"
#include "../../app-workspace/extension_framework/deviceid.hpp"

/* Define Driver Name */
#define DRIVER_NAME "adc"

#define SUCCESS     0

typedef enum
{
    CONVERSION  = 0x80,
    SETUP       = 0x40,
    AVERAGING   = 0x20,
    RESET       = 0x10
} MAX116XX_Registers;

typedef enum
{
    SCAN_0TON   = 0x00,
    SCAN_NTOH   = 0x02,
    SCAN_N      = 0x04,
    NO_SCAN     = 0x06
} MAX116XX_Scannning;

typedef enum
{
    NO_AVG      = 0x00,
    AVG_4       = 0x10,
    AVG_8       = 0x14,
    AVG_16      = 0x18,
    AVG_32      = 0x1C
} MAX116XX_Avgerage;

typedef enum
{
    CLK_INTERNAL    = 0x00,
    CLK_INTERNAL_1  = 0x10,
    CLK_INTERNAL_2  = 0x20,
    CLK_EXTERNAL    = 0x30
} MAX116XX_CLKselection;

typedef enum
{
    REF_INTERNAL_DELAY  = 0x00,
```

```
    REF_EXTERNAL        = 0x04,
    REF_INTERNAL_ON     = 0x08,
} MAX116XX_Reference;


static unsigned long *   base_addr  = NULL;
static struct resource * res        = NULL;
static unsigned long     remap_size = 0;
static int               major;       /* major number we get from the kernel */
static u8                ADC_chanel = 0;
static DeviceID          device_ID  = 0;

static void outport(u8 offset, u32 data)
{
    wmb();
    iowrite32(data, base_addr + offset);
}

static u32 inport(u8 offset)
{
    wmb();
    return ioread32(base_addr + offset);
}


////////////////////////////////////////////////////////////////////////////

static void spi_set_baud_rate_divider(u8 devider)
{
    u32 reg;
    do
    {
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    REGISTER_SET(reg, SPI_BAUD_RATE_DIVIDER_MASK, SPI_BAUD_RATE_DIVIDER_SHIFT, devider);
    outport(SPI_CONTROL_REGISTER_INDEX, reg);
}

static void spi_set_clock_phase(u8 val)
{
    u32 reg;
    do
    {
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    REGISTER_SET(reg, SPI_CLOCK_PHASE_MASK, SPI_CLOCK_PHASE_SHIFT, val);
    outport(SPI_CONTROL_REGISTER_INDEX, reg);
}

static void spi_set_clock_polarity(u8 val)
{
    u32 reg;
    do
    {
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    REGISTER_SET(reg, SPI_CLOCK_POLARITY_MASK, SPI_CLOCK_POLARITY_SHIFT, val);
    outport(SPI_CONTROL_REGISTER_INDEX, reg);
}

static void spi_set_data_length(u8 val)
{
    u32 reg;
    do
    {
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    REGISTER_SET(reg, SPI_DATA_LENGTH_MASK, SPI_DATA_LENGTH_SHIFT, val);
    outport(SPI_CONTROL_REGISTER_INDEX, reg);
}

static void spi_set_cs_force(u8 val)
{
    u32 reg;
    do
    {
```

```
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    REGISTER_SET(reg, SPI_CS_FORCE_MASK, SPI_CS_FORCE_SHIFT, val);
    outport(SPI_CONTROL_REGISTER_INDEX, reg);
}

static void spi_set_settle_time(u8 val)
{
    u32 reg;
    do
    {
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    REGISTER_SET(reg, SPI_SETTLE_TIME_MASK, SPI_SETTLE_TIME_SHIFT, val);
    outport(SPI_CONTROL_REGISTER_INDEX, reg);
}

// static u8 spi_get_transmission_done(void)
// {
//     u32 reg = inport(SPI_CONTROL_REGISTER_INDEX);
//     return REGISTER_GET(reg, SPI_TRANSMISSION_DONE_MASK,
SPI_TRANSMISSION_DONE_SHIFT);
// }

static void spi_set_slave_select(u8 val)
{
    u32 reg;
    do
    {
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    REGISTER_SET(reg, SPI_SLAVE_SELECT_MASK, SPI_SLAVE_SELECT_SHIFT, val);
    outport(SPI_CONTROL_REGISTER_INDEX, reg);
}

static u32 spi_receive_data(void)
{
    u32 reg;
    do
    {
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    return inport(SPI_DATA_REGISTER_INDEX);
}

static u32 spi_send_data(u32 data)
{
    u32 reg;
    do
    {
        reg = inport(SPI_CONTROL_REGISTER_INDEX);
    }
    while(!(reg & SPI_TRANSMISSION_DONE_MASK));
    outport(SPI_DATA_REGISTER_INDEX, data);
    return spi_receive_data();
}
////////////////////////////////////////////////////////////////////////////
void spi_initialize(void)
{
    spi_set_baud_rate_divider(0x10); // SPI_CLK(f) = 50000000 / 0x10
    spi_set_clock_phase(0);
    spi_set_clock_polarity(0);

    spi_set_cs_force(0);
    spi_set_settle_time(0);
    spi_set_slave_select(0);

    spi_set_data_length(SPI_DATA_LENGTH_8_BITS);

    spi_send_data(RESET);
    spi_send_data(AVERAGING | NO_AVG);
    spi_send_data(SETUP | CLK_EXTERNAL | REF_INTERNAL_ON);

    spi_set_data_length(SPI_DATA_LENGTH_24_BITS);
}
```

```
u32 adc_read(u8 channel)
{
    channel <<= 3;
    return spi_send_data( ( CONVERSION | channel | NO_SCAN ) << 16 );
}
////////////////////////////////////////////////////////////////////////////

/* Write operation for /proc/driver
* ---------------------------------
*/
static ssize_t proc_driver_write(struct file *file,
                                 const char __user * buffer,
                                 size_t buffer_size,
                                 loff_t * position)
{
    ssize_t written_size = 0;

    if ((buffer != NULL) && (buffer_size == sizeof(DeviceID)))
    {
        written_size = buffer_size - copy_from_user((void *) &device_ID, buffer,
sizeof(device_ID));

        switch (device_ID)
        {
            case LEVELSENSOR_0:
            ADC_chanel = 0;
            break;

            case LEVELSENSOR_1:
            ADC_chanel = 1;
            break;

            case WASTETLEVEL:
            ADC_chanel = 2;
            break;

            case LEAKSENSOR_0:
            ADC_chanel = 3;
            break;

            case LEAKSENSOR_1:
            ADC_chanel = 4;
            break;

            case ADC_CHANNEL_5:
            ADC_chanel = 5;
            break;

            case ADC_CHANNEL_6:
            ADC_chanel = 6;
            break;

            case ADC_CHANNEL_7:
            ADC_chanel = 7;
            break;

            case ADC_CHANNEL_8:
            ADC_chanel = 8;
            break;

            case ADC_CHANNEL_9:
            ADC_chanel = 9;
            break;

            case ADC_CHANNEL_10:
            ADC_chanel = 10;
            break;

            default:;
        }
    }
    else
    {   // Not handled, allow flushing
        written_size = buffer_size;
    }
    return written_size;
}

/* Read operation for /proc/driver
* ---------------------------------
```

```
*/
static ssize_t proc_driver_read(struct file *file,
                                char __user * buffer,
                                size_t buffer_size,
                                loff_t * position)
{
    ssize_t read_size = 0;

    if ((buffer != NULL) && (sizeof(IOPacket) <= buffer_size))
    {
        IOPacket packet;
        packet.device_ID = device_ID;
        packet.data = adc_read(ADC_chanel);
        read_size = buffer_size - copy_to_user(buffer, (void *) &packet,
sizeof(IOPacket));
    }
    else
    {   // Not handled, allow flushing
        read_size = buffer_size;
    }
    return read_size;
}


/* Callback function when opening file /proc/driver
* ----------------------------------------------------
*  Read the register value of driver file controller, print the value to
*  the sequence file struct seq_file *p. In file open operation for /proc/driver
*  this callback function will be called first to fill up the seq_file,
*  and seq_read function will print whatever in seq_file to the terminal.
*/
static int proc_driver_show(struct seq_file *p, void *v)
{
    u32 driverValue;
    driverValue = adc_read(ADC_chanel);
    seq_printf(p, "ADC chanel %d = 0x%X", ADC_chanel, driverValue);
    printk(KERN_INFO DRIVER_NAME "ADC chanel %d = 0x%08X\n", ADC_chanel, driverValue);
    return SUCCESS;
}

/* Open function for /proc/driver
* ----------------------------------
*  When user want to read /proc/driver (i.e. cat /proc/driver), the open function
*  will be called first. In the open function, a seq_file will be prepared and the
*  status of driver will be filled into the seq_file by proc_driver_show function.
*/
static int driver_open(struct inode *inode, struct file *file)
{
    unsigned int size = 16;
    char * buf;
    struct seq_file * m;
    int rc = SUCCESS;

    buf = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
    if (buf != NULL)
    {
        rc = single_open(file, proc_driver_show, NULL);

        if (rc == SUCCESS)
        {
            m = file->private_data;
            m->buf = buf;
            m->size = size;
        }
        else
        {
            kfree(buf);
        }
    }
    else
    {
        printk(KERN_ALERT DRIVER_NAME "%s No memory resource\n", __FUNCTION__);
        rc = -ENOMEM;
    }

    return rc;
}

/* File Operations for /proc/driver */
static const struct file_operations proc_driver_operations =
```

```
{
    .open           = driver_open,
    .write          = proc_driver_write,
    .read           = proc_driver_read,
    .llseek         = seq_lseek,
    .release        = single_release
};

/* Shutdown function for driver
* ---------------------------------
*  Before driver shutdown, turn-off all the stuff
*/
static void driver_shutdown(struct platform_device *pdev)
{
    spi_set_slave_select(3);
}

/* Remove function for driver
* ---------------------------------
*  When driver module is removed, turn off all the stuff first,
*  release virtual address and the memory region requested.
*/
static int driver_remove(struct platform_device *pdev)
{
    driver_shutdown(pdev);

    /* Remove /proc/driver entry */
    remove_proc_entry(DRIVER_NAME, NULL);

    unregister_chrdev(major, DRIVER_NAME);

    /* Release mapped virtual address */
    iounmap(base_addr);

    /* Release the region */
    release_mem_region(res->start, remap_size);

    return SUCCESS;
}

/* Device Probe function for driver
* ------------------------------------
*  Get the resource structure from the information in device tree.
*  request the memory regioon needed for the controller, and map it into
*  kernel virtual memory space. Create an entry under /proc file system
*  and register file operations for that entry.
*/
static int driver_probe(struct platform_device *pdev)
{
    struct proc_dir_entry * driver_proc_entry;
    int rc = SUCCESS;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res)
    {
        dev_err(&pdev->dev, "No memory resource\n");
        rc = -ENODEV;
    }

    if (rc == SUCCESS)
    {
        remap_size = res->end - res->start + 1;
        if (!request_mem_region(res->start, remap_size, pdev->name))
        {
            dev_err(&pdev->dev, "Cannot request IO\n");
            rc = -ENXIO;
        }
    }

    if (rc == SUCCESS)
    {
        base_addr = ioremap(res->start, remap_size);
        if (base_addr == NULL)
        {
            dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n", (unsigned
long)res->start);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }
```

```
    if (rc == SUCCESS)
    {
        driver_proc_entry = proc_create(DRIVER_NAME, 666, NULL,
&proc_driver_operations);
        if (driver_proc_entry == NULL)
        {
            dev_err(&pdev->dev, "Couldn't create proc entry\n");
            iounmap(base_addr);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        printk(KERN_INFO DRIVER_NAME " Mapped at virtual address 0x%08lx\n", (unsigned
long) base_addr);

        major = register_chrdev(0, DRIVER_NAME, &proc_driver_operations);

        spi_initialize();
    }

    return rc;
}

/* device match table to match with device node in device tree */
static const struct of_device_id zynq_adc_match[] =
{
    {.compatible = "yarib-adc-1.00.a"},
    { },
};

MODULE_DEVICE_TABLE(of, zynq_adc_match);

/* platform driver structure for device driver */
static struct platform_driver zynq_adc_driver =
{
    .driver =
    {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = zynq_adc_match
    },
    .probe = driver_probe,
    .remove = driver_remove,
    .shutdown = driver_shutdown
};

/* Register device platform driver */
module_platform_driver(zynq_adc_driver);

/* Module Informations */
MODULE_AUTHOR("Yarib, Inc.");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION(DRIVER_NAME ": ADC module");
MODULE_ALIAS(DRIVER_NAME);
```

PWM

```
/*
 * AXI Timer v2.0 - PWM Linux driver
 *
 * Yarib Nevárez <yarib_007@hotmail.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

#include <linux/kernel.h>
#include <asm/uaccess.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <asm/uaccess.h>    /* Needed for copy_from_user */
#include <asm/io.h>         /* Needed for IO Read/Write Functions */
#include <linux/proc_fs.h>  /* Needed for Proc File System Functions */
#include <linux/seq_file.h> /* Needed for Sequence File Operations */
#include <linux/platform_device.h>  /* Needed for Platform Driver Functions */
#include <asm/uaccess.h>

#include <linux/ctype.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/fcntl.h>

#define KERNEL_MODULE
#include "../../app-workspace/base_framework/device/iodef.hpp"
#include "../../app-workspace/extension_framework/deviceid.hpp"

/* Define Driver Name */
#define DRIVER_NAME     "pwm_0"

static unsigned long   * base_addr  = NULL;
static struct resource * res        = NULL;
static unsigned long     remap_size = 0;
static int               major;     // major number we get from the kernel
static DeviceID          device_ID = 0;

#define SUCCESS 0

static void outport(u8 offset, u32 data)
{
    wmb();
    iowrite32(data, base_addr + offset);
}

static u32 inport(u8 offset)
{
    wmb();
    return ioread32(base_addr + offset);
}

// *** Register index
typedef enum
{
    TCSR0 = 0x0,    // Timer 0 Control and Status Register
    TLR0,           // Timer 0 Load Register
    TCR0,           // Timer 0 Counter Register
    RSVD0,          // Reserved
    TCSR1,          // Timer 1 Control and Status Register
    TLR1,           // Timer 1 Load Register
    TCR1,           // Timer 1 Counter Register
    RSVD1           // Reserved
} AXI_Timer_Register_Index;

// *** Control/Status Register 0 flags (TCSR0)
typedef enum
{
    MDT0   = 0x001, // Timer 0 Mode (generate/capture)
    UDT0   = 0x002, // Up/Down Count Timer 0
    GENT0  = 0x004, // Enable External Generate Signal Timer 0
```

```
    CAPT0  = 0x008, // Enable External Capture Trigger Timer 0
    ARHT0  = 0x010, // Auto Reload/Hold Timer 0
    LOAD0  = 0x020, // Load Timer 0
    ENIT0  = 0x040, // Enable Interrupt for Timer 0
    ENT0   = 0x080, // Enable Timer 0
    T0INT  = 0x100, // Timer 0 Interrupt
    PWMA0  = 0x200, // Enable Pulse Width Modulation for Timer 0
    ENALL0 = 0x400, // Enable All Timers
    CASC   = 0x800  // Enable cascade mode of timers
} TCR0_Flags;

// *** Control/Status Register 1 flags (TCSR1)
typedef enum
{
    MDT1   = 0x001, // Timer 1 Mode (generate/capture)
    UDT1   = 0x002, // Up/Down Count Timer1
    GENT1  = 0x004, // Enable External Generate Signal Timer1
    CAPT1  = 0x008, // Enable External Capture Trigger Timer1
    ARHT1  = 0x010, // Auto Reload/Hold Timer1
    LOAD1  = 0x020, // Load Timer1
    ENIT1  = 0x040, // Enable Interrupt for Timer1
    ENT1   = 0x080, // Enable Timer1
    T1INT  = 0x100, // Timer1 Interrupt
    PWMB0  = 0x200, // Enable Pulse Width Modulation for Timer1
    ENALL1 = 0x400  // Enable All Timers
} TCR1_Flags;

static void pwm_set_period(u32 period)
{
    outport(TLR0, period);
}

static u32  pwm_get_period(void)
{
    return inport(TLR0);
}

static void pwm_set_high_time(u32 high_time)
{
    outport(TLR1, high_time);
}

static u32  pwm_get_high_time(void)
{
    return inport(TLR1);
}

////////////////////////////////////////////////////////////////////////////
#define AXI_CLK_FREQUENCY_HZ  50000000 // <--- Define AXI clk frecuency !
#define PWM_FREQUENCY         1000     // <--- Define desired PWM frecuency !
////////////////////////////////////////////////////////////////////////////

#define CALCULATE_PERIOD_REGISTER(AXI_FREQ, PWM_FREQ)   (((AXI_FREQ) / (PWM_FREQ)) - 2)
#define AXI_PERIOD_REGISTER
CALCULATE_PERIOD_REGISTER(AXI_CLK_FREQUENCY_HZ, PWM_FREQUENCY)

static void pwm_initialize(void)
{
    outport(TCSR0, ENALL0 | PWMA0 | GENT0 | UDT0);
    outport(TCSR1, ENALL1 | PWMB0 | GENT1 | UDT1);

    pwm_set_period(AXI_PERIOD_REGISTER);
    pwm_set_high_time(0);
}


/* Write operation for /proc/driver
* ----------------------------------
*/
static ssize_t proc_driver_write(struct file *file,
                                 const char __user * buffer,
                                 size_t buffer_size,
                                 loff_t * position)
{
    ssize_t written_size = 0;

    if (buffer != NULL)
    {
        if (buffer_size == sizeof(device_ID))
        {
```

```
            written_size = buffer_size - copy_from_user((void *) &device_ID, buffer,
sizeof(device_ID));
        }
        else if (buffer_size == sizeof(IOPacket))
        {
            IOPacket packet;
            written_size = buffer_size - copy_from_user((void *) &packet, buffer,
sizeof(IOPacket));

            if (written_size == buffer_size )
            {
                device_ID = packet.device_ID;
                switch (packet.device_ID)
                {
                    case PWM_0:
                        pwm_set_high_time(packet.data);
                        break;
                    default:;
                }
                * position += written_size;
            }
        }
    }
    return written_size;
}

/* Read operation for /proc/driver
* ------------------------------------
*/
static ssize_t proc_driver_read(struct file *file,
                                char __user * buffer,
                                size_t buffer_size,
                                loff_t * position)
{
    ssize_t read_size = 0;

    if ((buffer != NULL) && (sizeof(IOPacket) <= buffer_size))
    {
        IOPacket packet;
        packet.device_ID = device_ID;
        packet.data = pwm_get_high_time();
        read_size = buffer_size - copy_to_user(buffer, (void *) &packet,
sizeof(IOPacket));
    }
    return read_size;
}


/* Callback function when opening file /proc/driver
* ----------------------------------------------------
*  Read the register value of driver file controller, print the value to
*  the sequence file struct seq_file *p. In file open operation for /proc/driver
*  this callback function will be called first to fill up the seq_file,
*  and seq_read function will print whatever in seq_file to the terminal.
*/
static int proc_driver_show(struct seq_file *p, void *v)
{
    seq_printf(p, " PWM_0, Period = 0x%08X, High time = 0x%08X\n", pwm_get_period(),
pwm_get_high_time());
    return SUCCESS;
}

/* Open function for /proc/driver
* ------------------------------------
*  When user want to read /proc/driver (i.e. cat /proc/driver), the open function
*  will be called first. In the open function, a seq_file will be prepared and the
*  status of driver will be filled into the seq_file by proc_driver_show function.
*/
static int driver_open(struct inode *inode, struct file *file)
{
    unsigned int size = 64;
    char * buf;
    struct seq_file * m;
    int rc = SUCCESS;

    buf = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
    if (buf != NULL)
    {
        rc = single_open(file, proc_driver_show, NULL);
```

```
        if (!rc)
        {
            m = file->private_data;
            m->buf = buf;
            m->size = size;
        }
        else
        {
            kfree(buf);
        }
    }
    else
    {
        printk(KERN_ALERT DRIVER_NAME "%s No memory resource\n", __FUNCTION__);
        rc = -ENOMEM;
    }

    return rc;
}

/* File Operations for /proc/driver */
static const struct file_operations proc_driver_operations =
{
    .open           = driver_open,
    .write          = proc_driver_write,
    .read           = proc_driver_read,
    .llseek         = seq_lseek,
    .release        = single_release
};

/* Shutdown function for driver
* ---------------------------------
*  Before driver shutdown, turn-off all the stuff
*/
static void driver_shutdown(struct platform_device *pdev)
{
    pwm_set_high_time(0x00000000);
}

/* Remove function for driver
* ---------------------------------
*  When driver module is removed, turn off all the stuff first,
*  release virtual address and the memory region requested.
*/
static int driver_remove(struct platform_device *pdev)
{
    driver_shutdown(pdev);

    /* Remove /proc/driver entry */
    remove_proc_entry(DRIVER_NAME, NULL);

    unregister_chrdev(major, DRIVER_NAME);

    /* Release mapped virtual address */
    iounmap(base_addr);

    /* Release the region */
    release_mem_region(res->start, remap_size);

    return SUCCESS;
}

/* Device Probe function for driver
* ---------------------------------
*  Get the resource structure from the information in device tree.
*  request the memory regioon needed for the controller, and map it into
*  kernel virtual memory space. Create an entry under /proc file system
*  and register file operations for that entry.
*/
static int driver_probe(struct platform_device *pdev)
{
    struct proc_dir_entry * driver_proc_entry;
    int rc = SUCCESS;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res)
    {
        dev_err(&pdev->dev, "No memory resource\n");
        rc = -ENODEV;
    }
```

```
    if (rc == SUCCESS)
    {
        remap_size = res->end - res->start + 1;
        if (!request_mem_region(res->start, remap_size, pdev->name))
        {
            dev_err(&pdev->dev, "Cannot request IO\n");
            rc = -ENXIO;
        }
    }

    if (rc == SUCCESS)
    {
        base_addr = ioremap(res->start, remap_size);
        if (base_addr == NULL)
        {
            dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n", (unsigned
long)res->start);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        driver_proc_entry = proc_create(DRIVER_NAME, 666, NULL,
&proc_driver_operations);
        if (driver_proc_entry == NULL)
        {
            dev_err(&pdev->dev, "Couldn't create proc entry\n");
            iounmap(base_addr);
            release_mem_region(res->start, remap_size);
            rc = -ENOMEM;
        }
    }

    if (rc == SUCCESS)
    {
        printk(KERN_INFO DRIVER_NAME " Mapped at virtual address 0x%08lx\n", (unsigned
long) base_addr);

        major = register_chrdev(0, DRIVER_NAME, &proc_driver_operations);

        pwm_initialize();
    }

    return rc;
}

/* device match table to match with device node in device tree */
static const struct of_device_id zynq_pwm_0_match[] =
{
    {.compatible = "yarib-pwm_0-1.00.a"},
    { },
};

MODULE_DEVICE_TABLE(of, zynq_pwm_0_match);

/* platform driver structure for device driver */
static struct platform_driver zynq_pwm_0_driver =
{
    .driver =
    {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = zynq_pwm_0_match
    },
    .probe = driver_probe,
    .remove = driver_remove,
    .shutdown = driver_shutdown
};

/* Register device platform driver */
module_platform_driver(zynq_pwm_0_driver);

/* Module Informations */
MODULE_AUTHOR("Yarib, Inc.");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION(DRIVER_NAME ": PWM module");
MODULE_ALIAS(DRIVER_NAME);
```