Fraunhofer IIS

Universidad de Navarra

MASTER THESIS

IMPLEMENTING A DAB RECEIVER ON XILINX ZYNQ FPGA PROTOTYPING PLATFORM

by

Iñigo Cortés Vidal

Supervisor in the University of Navarra:	Dr.-Ing. Juan Francisco Sevillano
Supervisor in Fraunhofer IIS:	Dipl.-Ing. Martin Speitel
Supervisor in Fraunhofer IIS:	Dipl.-Inform. Thomas Dettbarn

April 12, 2018

Contents

List of Figures

List of Tables

Abbreviations

ADC Analog to Digital Converter

AM Analog Modulation

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

CIF Common Interleaved Frame

CU Capacity Unit

D-QPSK Differential QPSK

DMB Digital Multimedia Broadcast

EEP Equal Error Protection

FIB Fast Information Blocks

FIC Fast Information Channel

FM Frequency Modulation

FPGA Field Programmable Gate Array

FSBL First Stage Bootloader

GPIO General Purpose Input/Output

HE-AAC High-Efficiency Advanced Audio Coding

ILA Integrated Logic Analyzer

IPL Inner Physical Layer

IQ In-phase/Quadrature

LLR Logarithmmic Likelihood Ratio

MPEG Moving Pictures Experts Group

MSC Main Service Channel

MSPS Megasamples per second

OFDM Orthogonal Frequency Division Multiplexing

OPL Output Physical Layer

OS Operating System

QPSK Quadrature Phase Shift Keying

ROOTFS Root File system

RTL Register Transfer Level

SDK Software Development Kit

SDR Software Defined Radio

SoC System on Chip

SSB Second Stage Bootloader

U-boot Universal Bootloader

UART Universal Asynchronous Receiver Transmitter

UEP Unequal Error Protection

VHDL Very High Speed Integrated Circuit Hardware Description Language

Zero-IF Zero Intermediate Frequency

SWORN STATEMENT

Signature

_____

Name Surname

Chapter 1

Introduction

Analog radio standards like AM and FM are the most common and traditional used methods of audio signal transmission. In this kind of systems, the content is very sensitive to radio frequency impairments happened in physical transmission (such as noise, fading channels, etc.), giving rise to a poor fidelity. As an alternative, digital radio standards are used not only to improve fidelity, but also power consumption and spectral efficiency. Digital Audio Broadcast (DAB) is a digital radio standard ramping up quickly in many European countries and the number of available DAB receivers for home entertainment and car radios is strongly increasing. Contrary to AM or FM, the content is encoded and, therefore, more robust against radio frequency impairments.

DAB receivers are usually implemented in a Software Defined Radio (SDR), a radio communication system which is software implemented in an embedded system. The digital signal processing, data analysis and decoding of the received signal is performed and this process must be done without losing any portion of the received signal. For example, taking into account that the DAB received signal bandwidth is 1.5MHz and the sampling frequency of the ADC in the receiver is 2.048MSPS, if the clock frequency of the microprocessor is 900MHz, for each sample there are at most 440 cycles to perform the corresponding processing. The higher frequency the microprocessor is working, the easier to perform all the signal processing of the sampled received data in real time. However, one disadvantage of DAB software receivers is the huge load of work that suffers the microprocessor, having less resources to do other tasks.

Thus, one solution to reduce significantly the workload of the microprocessor is the design of a hardware module for the DAB receiver. In Fraunhofer IIS, a DAB receiver hardware module has been already implemented on a big prototyping platform composed by 6 FPGAs and a Tensilica microprocessor. The DAB receiver is implemented on the FPGAs with a clock frequency of 16MHz. In this case, although there are at most 8 cycles per received sample to perform the processing in the FPGA, the microprocessors have more capacity to perform other tasks. The goal of this master thesis is to implement that hardware solution in a smaller, cheaper and less power consuming platform: the Xilinx ZYNQ-7000 All Programmable System on Chip (SoC) ZC706 Evaluation Kit. It is a Xilinx Evaluation Board that contains a SoC called ZYNQ-7000 composed by a single FPGA and a dual-core ARM Cortex ^{TM}-A9. The two ARMs with a Linux Operating System are configured to achieve communication with the DAB hardware module.

Moreover, hardware modules are developed to synchronize the data transfers between the FPGA and the microprocessors and control the DAB hardware module. To have a better understanding about the hardware modules and verify the correct functionality, different tests are performed. The DAB receiver is completed to ensure a correct functionality while listening to music in real time.

This master thesis is organized in ascending order as follows. Firstly, the DAB standard, the state of the art of actual DAB receivers and the proposed objectives are presented. Then, the configuration of the processing system in the ZYNQ and the implementation of the DMA is explained. Based on the latter, in the following chapter the implementation of the DAB receiver hardware module on the ZYNQ is described. Afterwards, a complete DAB receiver design implemented on the ZYNQ is explained. These two last chapters include tests to validate the correct operation of the designs. Finally, the possible improvements, future work and conclusions are drawn.

Chapter 2

State of the art

DAB is a digital radio standard where audio and multimedia services are encoded on a dedicated device before being broadcasted and it is transmitted in Band III (174-230 MHz). By adding algorithms for forward error correction to the radio and post processing filters, users can have the best possible listening experience, even in poor transmission scenarios. DAB was created using MPEG Layer II audio coding. Later, the DAB+ standard was developed using MPEG-4 HE-AAC v2 audio coding due to its high efficiency and performance at lower bit rate.

2.1 DAB standard

DAB defines four distinct transmission modes, each one intended for a different set of scenarios. They are numbered in roman numerals from mode I to mode IV [1]. These modes have in common a main structure (figure 2.1) composed by three main channels: the synchronization channel, the fast information channel (FIC) and the main service channel (MSC). The synchronization channel is used for basic demodulator functions (transmission frame synchronization, automatic frequency control...). The data in FIC is always encoded in the same format and it is divided by fast information blocks (FIB). Among other things, it contains information about how the data in the MSC is encoded. The MSC carries the audio and data service components. This channel is composed by common interleaved frames (CIF) where each one is divided by 864 Capacity Units (CU).

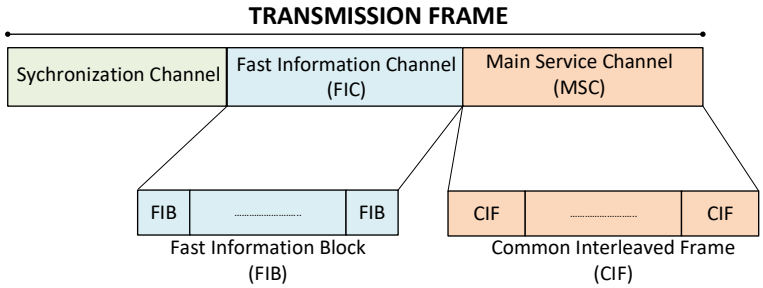

Figure 2.1: Transmission frame structure

In the DAB transmitter, the FIC and the CIF data of different content providers is structured depending on the specified transmission mode and then modulated in QPSK.

[1] The difference between these modes resides in the number of carriers, the carrier spacing, the symbol duration and the guard interval. See clause 14.2 of [1].

These QPSK symbols are frequency interleaved to mitigate frequency selective fading. After that, a $\pi/4$ - shift D-QPSK modulation is applied. Finally, the synchronization channel, the FIC and the MSC are joined in a OFDM signal generator and the DAB signal with a frequency bandwidth of 1.5 MHz is sent. Content providers have to choose if they want to transmit the signal at a higher bit rate or with a higher redundancy to achieve a more robust reception. In DAB standard eight Equal Error Protection (EEP) levels and 64 Unequal Error Protection (UEP) profiles have been defined. It is possible to change the error protection depending on the desired robustness of the reception that the content providers want.

The mode I DAB transmission frame can be observed in figure 2.2. The duration of the transmission frame is 96 ms and taking into account that the frequency sample is 2,048MSPS, the total number of samples is 196608. One OFDM symbol has 2552 samples: the first 504 samples corresponds to the guard interval and the remainder samples to the data. There is a total of 76 OFDM symbols: 1 for the synchronization channel, 3 for the FIC and 72 for the MSC. The Null symbol takes the remaining 2656 samples. The FIC is divided in 4 blocks that contains 3 FIBs each one and the MSC is also divided in 4 blocks. Each FIC block describes one of the MSC blocks.

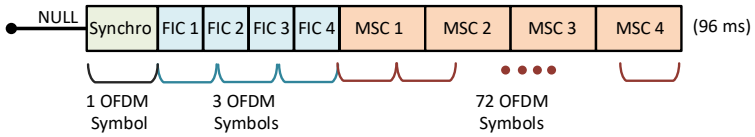

Figure 2.2: Frame structure in DAB transmission mode I.

Content providers book a number of those CUs from a network operator. The network operator is transmitting those contents in parallel on a given frequency band (called ensemble). Figure 2.3 shows an example of the DAB service structure. An ensembler called "Bayern" has two services: Bayern 1 OBB and Bayern 1 SCHW. Each service has services components such as service information or audio. The amount of booked CUs of a service component is called a sub-channel. The service information is usually carried in the FIC but, if the service information is very extensive, it is carried in a separate sub-channel (see the service information of Bayern 1 SCHW). In addition, the audio service component is carried in a sub-channel and both services can have a second audio component which shares the same sub-channel. In DAB it is possible to have up to 64 sub-channels within a single ensemble. Sometimes, the content providers can make reconfigurations booking less or more CUs. The FIC will contain information regarding to the name of the ensemble and the corresponding services. One service can have one or more sub-channels.

Not only DAB uses this main structure, but DAB+ and Digital Multimedia Broadcasting

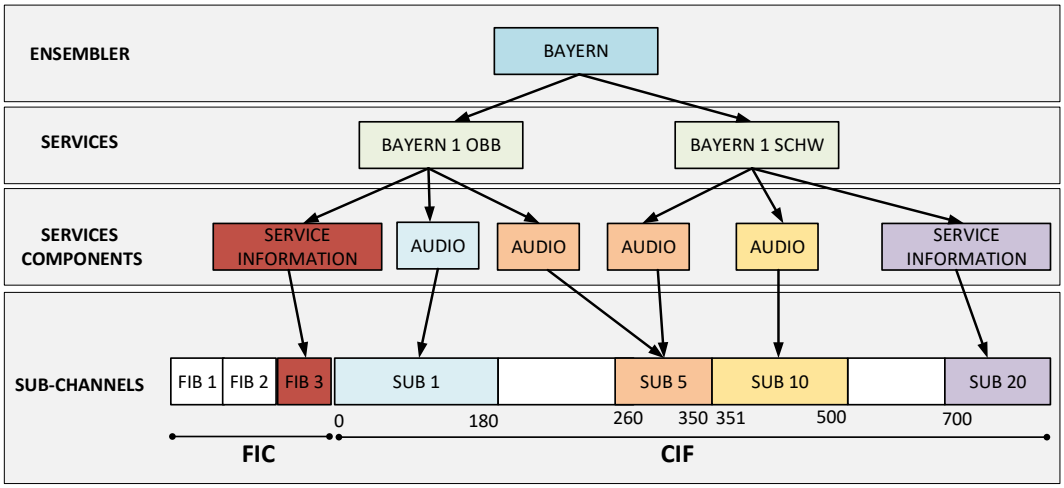

Figure 2.3: DAB service structure: The FIC will store the SI of the services but, if SI is too large, it is carried in a subchannel. This CIF has 4 sub-channels.

(DMB) also has the same physical layer for transmitting their data. The difference between them lies in the format of the encoded data. As they use the same infrastructure, they can co-exist and thus enable a wide range of possible multiplex scenarios.

2.2 DAB receiver work-flow

Figure 2.4 shows the general block diagram of the DAB receiver. The tuner receives the desired DAB signal located in a certain frequency band. Among of other operations, it performs the quadrature IQ Zero-IF demodulation and the ADC will sample the incoming I- and Q-channel with a 16 bit resolution. The sampling frequency is 2.048MSPS being able to sample the incoming signal of 1.5 MHz bandwidth without aliasing (figure 2.5). The 16-bit I and 16-bit Q samples are stored in an input buffer and the digital signal processing begins.

The DAB receiver performs a timing synchronization with the Null symbol of the synchronization channel finding the beginning of a DAB transmission frame. Once the null symbol is detected, the phase reference symbol is next. This reference symbol is used to calculate the frequency and phase offset. This information is sent to the inner physical layer (IPL) module, that performs the frequency offset and phase offset compensation (equalization). After that, the data is frequency deinterleaved and differential demodulated. The last block of the IPL is the soft demapper performing the logarithmic likelihood ratio (LLR) to estimate the most probable incoming bit. The deinterleaver will store the received LLR MSC data from the IPL and the LLR FIC data will be directly sent to the Output Physical Layer (OPL). In the OPL, the LLR data is depunctured and error corrected by the Viterbi block. Finally, the data is descrambled by the Energy Dispersal block and stored in a buffer. The received MSC data can be taken out to perform the

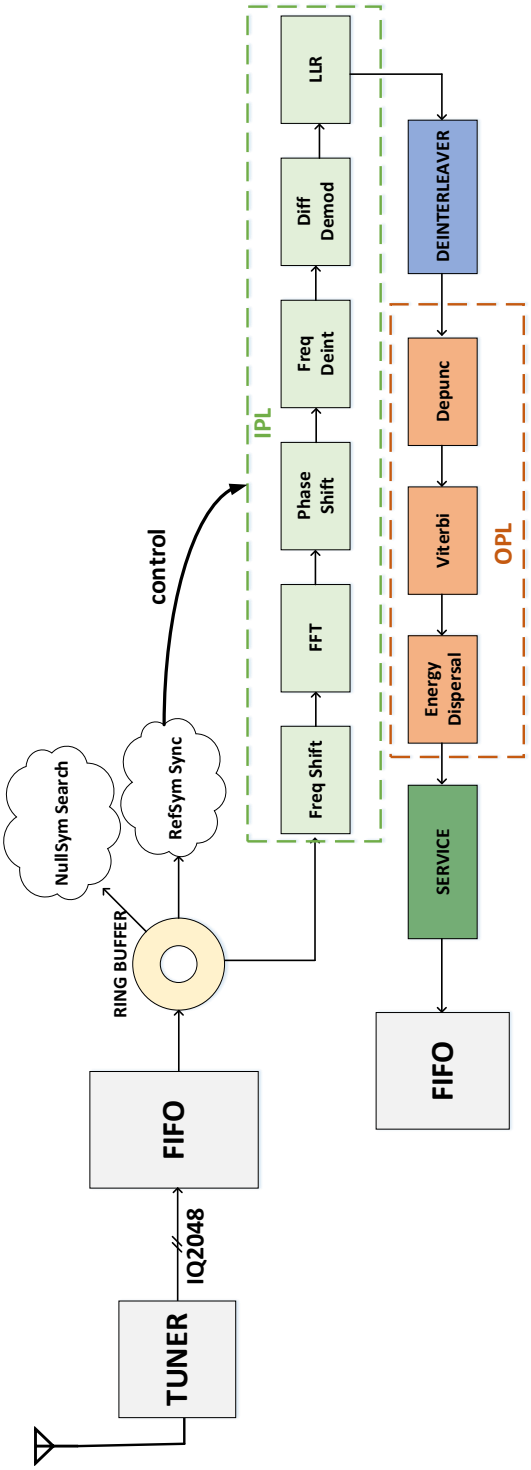

Figure 2.4: DAB receiver block diagram: IQ data sampled at 2.048MSPS (IQ2048) is the input of the DAB baseband digital signal processing, that is composed by two main blocks: IPL and OPL.

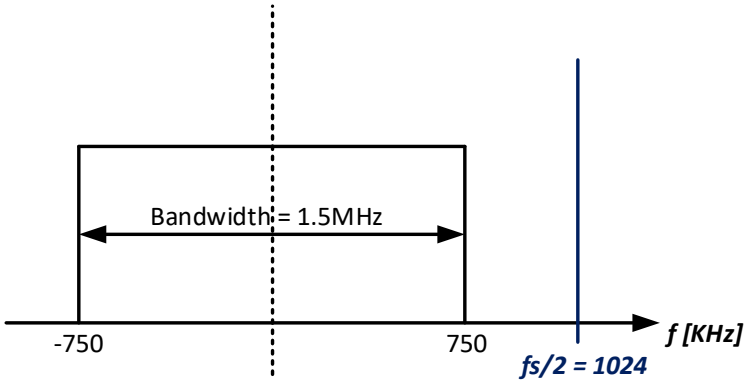

Figure 2.5: Incoming OFDM signal in the frequency domain. The frequency bandwidth is 1.5MHz and the sampling frequency (f_s) is 2.048MSPS. There is no aliasing because $f_s/2 > 750$KHz.

audio decoding and extract the expected audio signal.

In the DAB hardware module, besides performing the digital signal processing, the user can configure each of its sub-modules writing or reading data in the corresponding bank of registers through Inter-Integrated Circuit (I2C). Figure 2.6 shows the bank of registers for each module. For example, the IPL registers are located in bank 1 (address 0x100 to 0x1FF), the deinterleaver in bank 2 and so on. Then, if the data located in address 0x200 is written, the IPL ignores the request but the deinterleaver accepts it and performs the corresponding read/write operation.

2.3 Actual DAB solutions

A SDR platform for DAB to the automotive industry is being implemented using a processor of Texas Instrument called Jacinto. This Jacinto-based platform is used also for multiple standards around the world, and finally that implies a huge last of work for the microprocessor.

This software workload is reduced implementing a DAB-interface on a first prototyping platform. This first prototype is big, expensive and power consuming. Size, cost and power consumption must be minimized as much as possible in order to come to the market successfully.

Xilinx offers an innovative family of products called *ZYNQ-7000 All Programmable SoCs* that integrates a dual-core or single-core ARM Cortex ^{TM}-A9 based processing system (PS) and a Xilinx programmable logic (PL) in a single device. The PS also includes a rich set of peripheral connectivity interfaces. Depending on the needed requirements (number of processor cores, maximum frequency of the PS, number of programmable logic cells in the PL, etc.) this family is classified in different device names[2]. Figure 2.7 shows

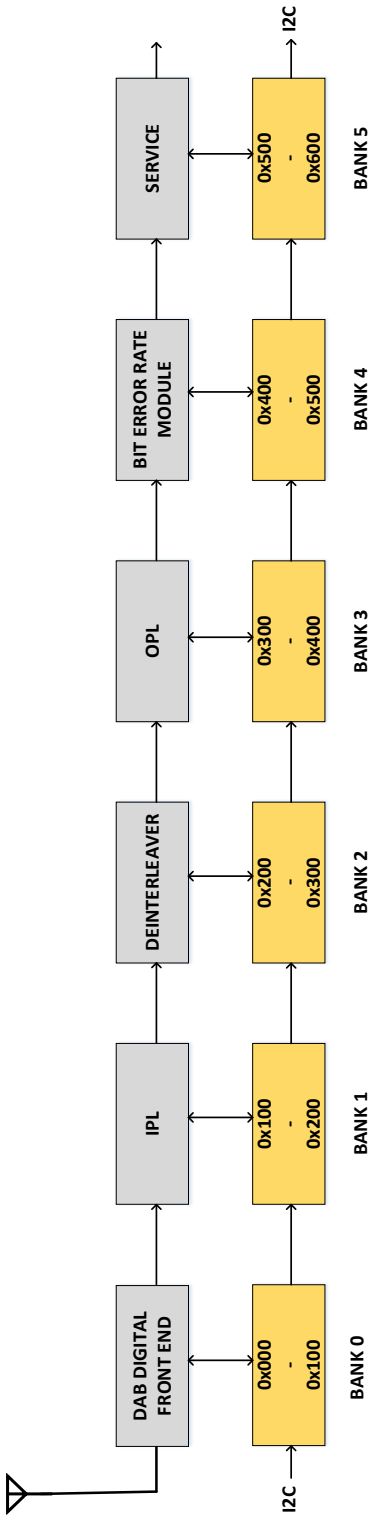

Figure 2.6: Block diagram of the DAB hardware module implemented on the first prototyping platform. The register map is subdivided into register banks of 256 addresses each. Through I2C those registers can be read or written to configure the DAB Module.

Figure 2.7: ZYNQ-7000 All Programmable SoC ZC706 Evaluation Kit.

the ZC706 Evaluation Kit, whose device name is Z-7045. The features given by the ZC706 Evaluation Kit are very attractive to integrate the DAB receiver[3][4].

Chapter 3

Objectives

DAB receivers implemented on SDR platforms presents a huge load of work for the processor. In order to avoid this problem, a hardware module working as a coprocessor for DAB is designed, removing significantly the software workload. The hardware DAB receiver module is already created and implemented on a first prototyping platform. The goal of this master thesis is to implement it on a cheaper, smaller and less power consuming prototyping platform: the mentioned ZYNQ-7000 ZC706 Evaluation Kit.

In order to achieve the above main objective, the following milestones have been defined:

- **Configuration of the PS**: This first milestone is to install the Linux Operating System on both ARMs and boot the ZYNQ correctly from a SD card. Knowledge about the structure of a Petalinux project and the boot process of the ZYNQ is required. This milestone is completed verifying the successfully boot of the ZYNQ.

- **Direct Memory Access implementation on ZYNQ**: The second milestone is to create a simple hardware demo design using a Xilinx provided DMA module in the PL. The PS is also configured adding a modified Xilinx provided device driver to control the DMA hardware module from the PS. In addition to this, an user application is created. The user sends a IQ data file from the PS to the PL and the resultant processed data is be sent back to the PS and stored in a file. This milestone is completed when this stored file is correctly processed. The DMA behavior is analyzed with some real hardware waveforms using the Integrated Logic Analyzer (ILA). Another important observation is the bit rate of the complete design. The achieved bit rate of the design must be high enough to continue with the following milestones. To undertake this milestone knowledge of VHDL and C language is necessary.

- **OPL implementation on ZYNQ**: Taking into account the previous hardware design in the PL using the DMA, one more hardware design is created in this third milestone to ensure the correct functionality of the OPL module. Simulations made by elaborated test-benches are shown to understand the modules. Moreover, two user interface applications are created: One application consists of sending the output processed FIC data of the IPL from the PS to the PL, the PL performs the

corresponding processing of that incoming data and it sends back the resultant data to the PS. The second application is used to control the OPL through the register bank of the OPL hardware module. Once the processed data is verified that is correct and the control is correctly performed, this milestone is completed.

- **DAB receiver implementation on ZYNQ**: This fourth milestone is to implement the complete DAB receiver on the ZYNQ. Elaborated test-benches are performed to understand the behavior of the hardware design. The same user application as in the third milestone is used to configure the DAB sub-modules through their corresponding register banks. Moreover, the ZYNQ platform is connected by USB to a configurable tuner that is connected to an antenna. The tuner can receive AM, FM or DAB signals and the gain and the frequency can be selected to receive the desired signal. Another user application is created to configure the tuner with the ZYNQ, so that real time IQ data in DAB mode can be received. Then, the real time processing of the received IQ data is performed and a sub-channel of the incoming data is selected. This final milestone is completed when music is listened from the ZYNQ.

Chapter 4

Configuration of the Processing System

This first milestone is based on the configuration of the PS in the ZYNQ-7000 prototyping platform (figure 4.1). A Symmetric Multiprocessing (SMP) model with an operating system (OS) based on the Linux kernel is installed in both of the ARM Cortex-A9 using the Petalinux tools. A simple Petalinux project is created and the steps to boot the ZYNQ from the SD card with the required files are shown. The fulfillment of this milestone is achieved when the correct installation of the Linux OS is verified accessing to the ZYNQ-7000 through UART.

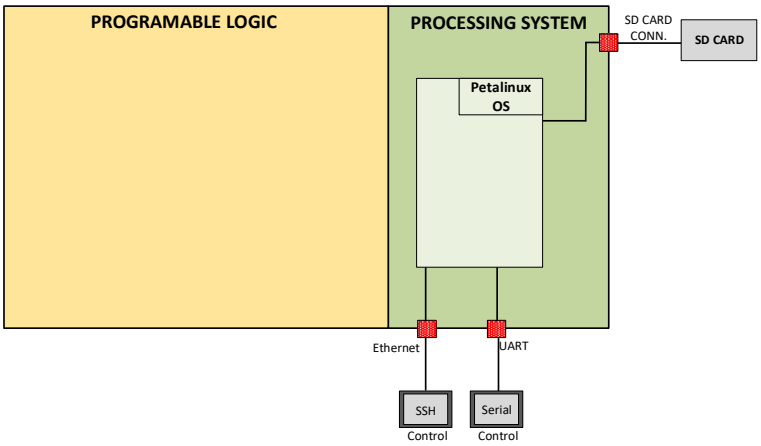

Figure 4.1: Block diagram of first milestone: Boot the ZYNQ from the SD card and install Linux OS on ARMs.

4.1 Petalinux Tools

Linux is open source and a popular choice among developers. Although Baremetal (Standalone) applications are faster than Linux applications (because they do not need any system calls between kernel and user space), Linux is used because services like networking and multimedia (among others) are required in this project.

Petalinux Tools offers Embedded Linux Solutions on Xilinx Processing Systems. New Linux developers use this tool because of the easiness creating a Linux structure for an embedded system. When a Petalinux project is created, the developers don't start from scratch, but a default structure is already done and they can configure the project the way they want (modifying the Linux kernel, creating Linux applications ...). The structure of Petalinux is changing when new versions are being released. In this project the version of Petalinux is 2017.2 [5]. This version in particular, unlike older ones, has a Yocto based

structure[1].

4.2 Xilinx Vivado Design Tool

Vivado is a design environment for FPGA products from Xilinx. HDL designs can be synthesized and analyzed performing simulations. Vivado also provides a large Xilinx IP library that is very useful for the created designs during this project[6]. Its version is the same as the Petalinux Tools version, 2017.2 (it is recommendable to use same versions). First of all, when a Vivado project is created, the used SoC or board must be specified. In this case, the board ZC706 of ZYNQ-7000 family is selected. Next, a design block is created and the PS of the ZYNQ SoC, with dual ARM Cortex-A9 inside, is added and configured with the ZC706 preset configuration (Figure 4.2). After that, Xilinx default hardware modules and created Register Transfer Level (RTL) designs are added to the design. Then, the design is synthesized and implemented and a *bitstream* file is generated. Finally, the *bitstream* of the project is exported and the Xilinx Software Development Kit (SDK) application is launched [7]. SDK creates a folder called *design_1_wrapper_hardware_platform_X* that is needed by the Petalinux Tools in order to generate the BOOT image.

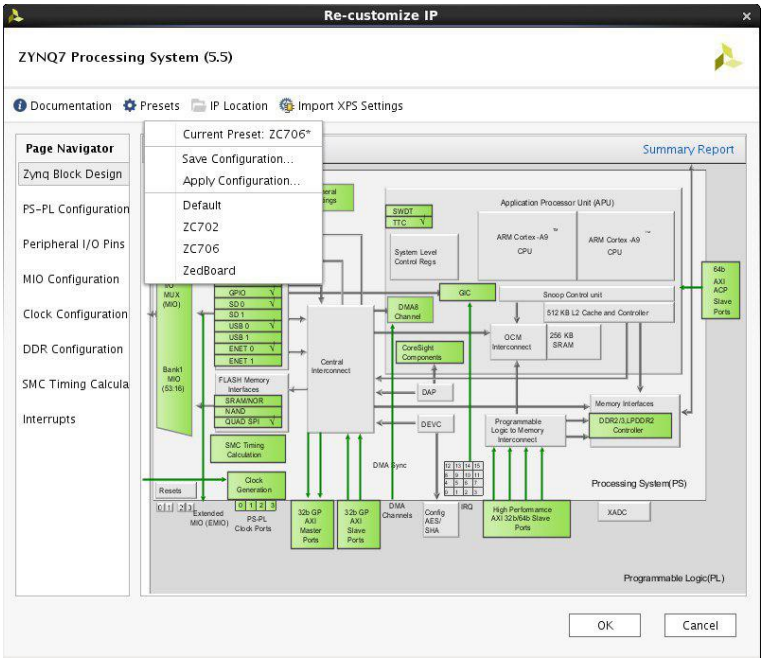

Figure 4.2: Preset configuration of PS in ZC706 Evaluation Kit.

A default hardware design provided by the Petalinux Tools is applied for the created Petalinux project and the Vivado application is not used for this milestone.

[1] Yocto is an open source collaboration project that provides templates, tools and methods to help you create custom Linux-based systems for embedded products regardless of the hardware architecture.

4.3 ZYNQ SoC boot sequence

The boot process of the ZYNQ is more complex than a traditional System on Chip (SoC). In Figure 4.3, when the ZYNQ SoC is powered on, the First Stage Bootloader (FSBL) is the one that handles the booting process. The FSBL initializes first the PS block as it was configured in Vivado (*ps7_init* file). After that, the FSBL for the PL is initialized and the hardware design (*bitstream*) is exported to the FPGA. Due to the fact that a linux OS is going to be installed, a second stage bootloader (SSB) called *Universal Bootloader* (U-boot) is executed. U-Boot is an open source bootloader used by the ZYNQ-7000 family. This file points to three important files: the device tree, the Linux kernel image and the Root File System. The next step is the execution of the Linux kernel and the Linux Root File System (called also *ROOTFS*). Finally, Linux kernel device drivers and user level applications are added.

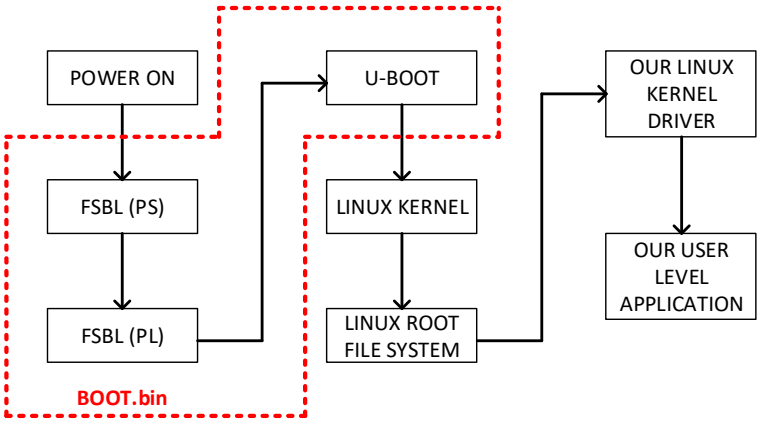

Figure 4.3: Boot sequence of the ZYNQ SoC.

The hardware modules in the PL that are connected directly to the PS are called *platform devices* and they are not automatically detected by the system. That is why the device tree, a structured file, informs to the kernel the location of those platform devices. Being more specifically, it indicates which platform driver in the kernel is responsible and where the platform device is located in memory. The platform driver is an extension of the kernel (device driver) that provides information related to the communication with a platform device[8].

In a Petalinux project, the device tree is divided in a hierarchical way and when the project is built, a single combined device tree is achieved. On boot time this file is compiled and given to the kernel. During this project, one platform driver provided by Xilinx is used and it is described in the next chapter. All the user device drivers, Linux user applications and pre-built packages are stored in the Root File System.

4.4 Layer model structure

Figure 4.4 shows the layer model structure of the ZYNQ SoC using a Linux OS. There are three main layers: the Physical layer, the Kernel space and the User space. The physical layer is related to the designed hardware in the PL, where the hardware modules are located in specific physical addresses. Those physical addresses are mapped to virtual addresses in the kernel space. This second layer communicates with the first one with the help of the previously mentioned devices drivers through bus infrastructures such as the Advanced eXtensible Interface (AXI). The user memory and the kernel memory are independent and implemented in separate address spaces. The Linux kernel framework allows the device drivers to expose the hardware features to user space applications and the system call interface performs the link between both spaces. In the user space there is a *virtual file system* that is a mechanism for the kernel to export operating details to the user space. In the following chapters, the user applications requests and sends data to the implemented device driver.

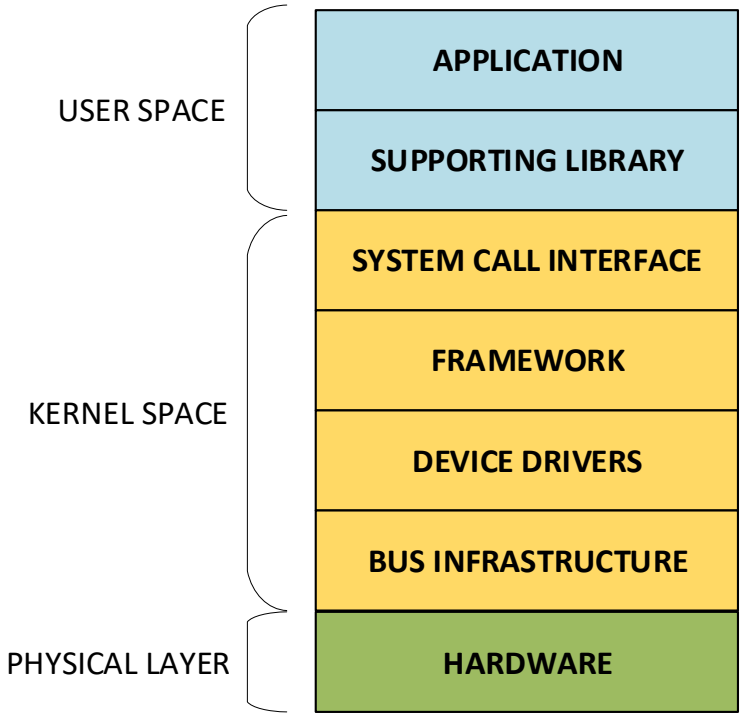

Figure 4.4: Layer model structure of the ZYNQ SoC under Linux.

4.5 SD card configuration

Figure 4.5 indicates the files that must be imported to the SD card in order to achieve a successful boot in ZYNQ SoC. The SD card must be divided in two partitions. The format

of the main partition is FAT32 (2 GBytes is the recommended size) and it is composed by Linux kernel image (image.ub), the device tree of the system (system.dtb) and the boot image (BOOT.bin). The second partition (the remainder space of the SD card) contains the Root File System and the format is EXT4.

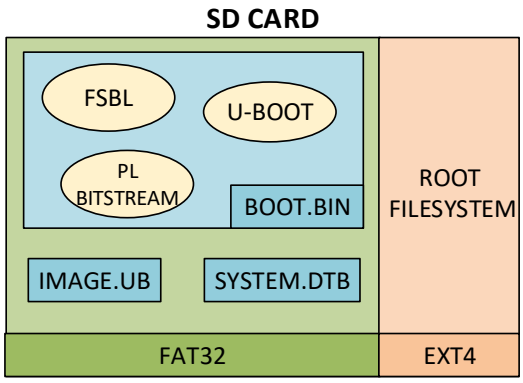

Figure 4.5: SD card partitions and required files for a correct boot process.

4.6 Results

Appendix A shows a bash script file that makes all the configuration steps of a Petalinux project. Appendix B is also a bash script that mounts the necessary files of the Petalinux project into the SD card. Appendix C shows the installation requirements of the Petalinux Tools application. Taking into account the previous sections and Appendix A, B and C, a Petalinux project has been created using a default Xilinx provided hardware design. Through the UART port of the ZYNQ platform, the successfully boot of the Linux OS can be observed in the computer using a serial connection with PuTTY application (Appendix D).

Chapter 5

Direct Memory Access implementation

The DMA is an essential hardware module for the final design. It manages the transfer of data between the PL and PS and vice versa. In this milestone a demo design is created in the ZYNQ to test the transfer data rate of a Xilinx provided AXI DMA hardware module [9] (figure 5.1). First, the transfer procedure of the AXI DMA is described. Then, the implemented hardware design using a Xilinx provided AXI DMA module is shown. After that, a Xilinx provided platform driver for this hardware module is presented and the created user application is described. Finally, the achieved results and possible improvements are discussed.

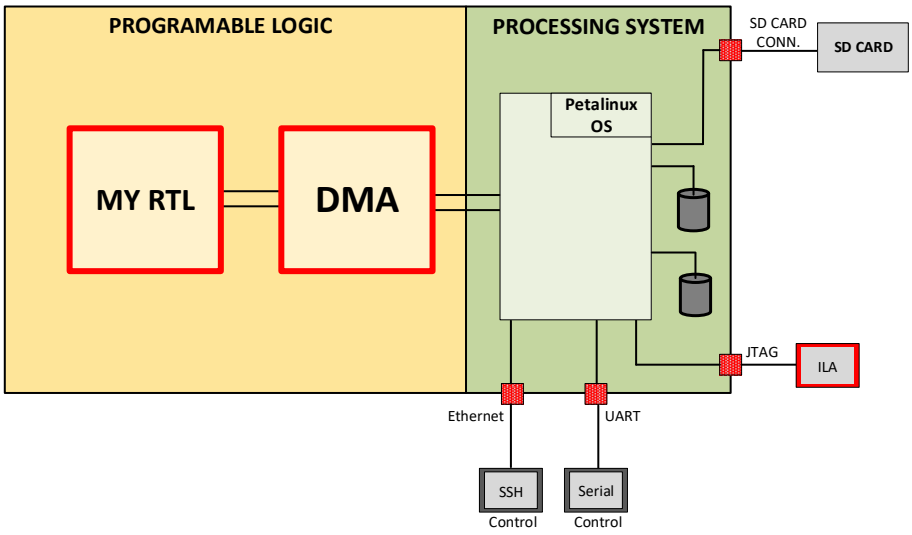

Figure 5.1: Block diagram of second milestone: A hardware demo design using the AXI DMA module is created. An extension of the Kernel is made for the AXI DMA and one user application is created.

5.1 AXI DMA transfer procedure

The AXI DMA is controlled by the PS using descriptors. A descriptor is an indicator of amount of data that should be transferred from a source to a destination. In other words, a descriptor is a transfer task defined by the PS. Once the AXI DMA receives a descriptor, it performs the transfer and as soon as the transfer is finished, the DMA generates an interrupt to the PS and then the PS sends another descriptor to the AXI DMA. This is the simplest transfer mode called *Direct mode* and is the one being used in this project. There is a second transfer mode called *Scatter/Gather*. In this mode, the PS defines a set of transfer tasks and sends it to a memory. After that, the PS instructs

the DMA to begin the operation and provides the address of the first and the last defined transfer task located in that memory. When the DMA finishes with all the tasks, it sends to the PS an interrupt and the PS defines a new set of transfer tasks in the memory. This mode gives a kind of independence to the PS because it defines at one time slot a set of descriptors and while the AXI DMA is performing the data transfer, the PS is available for other operations. However, the direct mode is used because it is enough for the final design.

5.2 DMA demo design procedure

Figure 5.2 shows the block diagram of the implemented hardware design for this milestone. The ZYNQ PS controls the three Xilinx provided modules: AXI GPIO 0, AXI GPIO 1 and AXI DMA. These three modules are connected to a created module *MY RTL*, which performs an addition between the incoming data of AXI DMA (S_AXIS_TDATA) and AXI GPIO 0 (A). The resultant data is sent back to the AXI DMA (M_AXIS_TDATA) and it is accumulated and sent to AXI GPIO 1 (B).

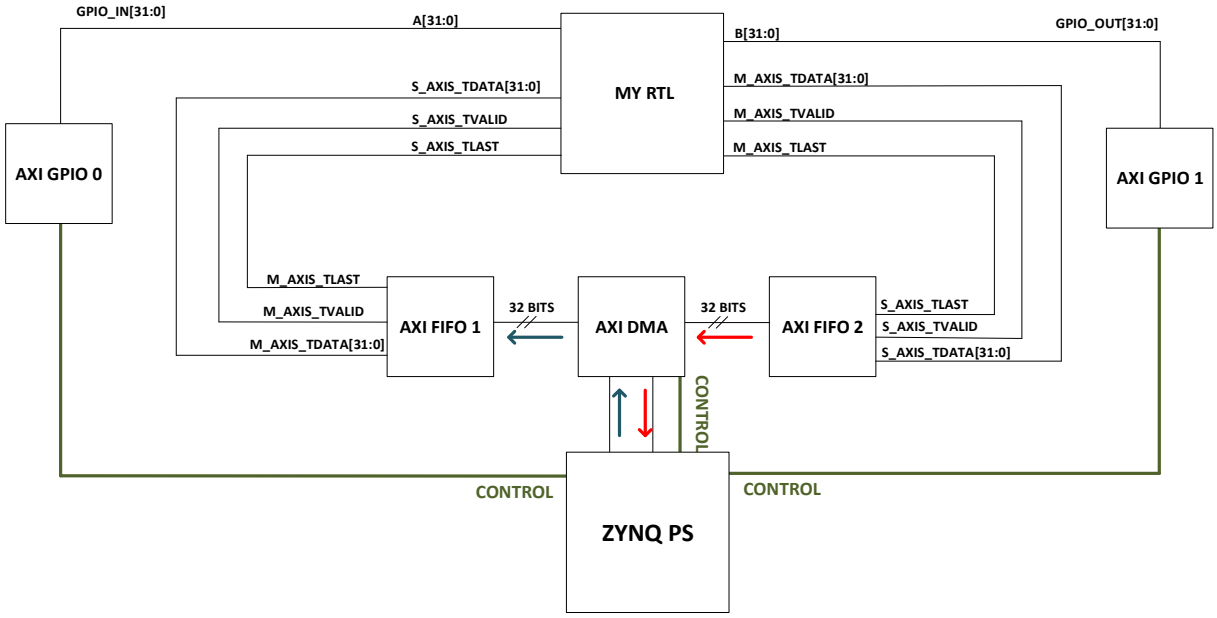

Figure 5.2: DMA demo design.

The main idea of this design is to transfer 82.58 MBytes of IQ data [1] (corresponding to 10 seconds DAB signal) from the PS to the PL, send back the IQ processed data to the PS and store it in a file. To perform this transfer the Xilinx provided AXI DMA is used and configured in direct mode. The AXI DMA hardware module has a small buffer and its clock frequency is 50MHz (the same as the rest of the hardware modules in this

[1] As an example, a total transfer size of 82.58 MBytes is performed. The total size of the transfer is needed to calculate later the bit rate of the entire design.

design). On the other hand, the clock speed of the PS is around 600 MHz, much higher than the AXI DMA's clock frequency. If 82.58 MBytes are sent continuously from the PS to the AXI DMA at 600MHz, the buffer of the AXI DMA collapses and the design does not work. That is why the amount of data that is sent must be partitioned. In this case, the amount of data that is defined by the descriptor is 32 KBytes [2]. Then, the PS sends 32KBytes of memory-mapped data (with a certain address) in chunks of 4 Bytes to the AXI DMA. The AXI DMA sends 4 Bytes of stream data (without address) to *MY RTL*, that performs a simple addition. The result is sent back to the AXI DMA and also accumulated in register B.The addition is performed between the incoming 4 Bytes data and another 4 Bytes (A) defined by the PS through one AXI GPIO[10] (AXI GPIO 0). The accumulated result is sent to the PS through a second AXI GPIO (AXI GPIO 1). The *tlast* signal of the AXI communication received by the DMA from *MY RTL* indicates the last chunk of 4 Bytes and when it is set, the AXI DMA is ready to send the interrupt once that last chunk is sent to the PS. Then, the PS sends another descriptor to receive the 32 KBytes processed data. Once received the processed data, another interrupt is sent from the AXI DMA to the PS and the same process repeats until sending the 82.58 MBytes IQ data.

5.3 Linux DMA

Figure 5.3 shows the structure of the Linux kernel and user space to achieve a communication with the AXI DMA platform device in the Physical Layer. A Xilinx provided device driver is used. In order to control the AXI DMA in a consistent and more abstract manner, the DMA Engine Framework is added (also from Xilinx). In addition, a DMA proxy device driver template that uses the functions provided by the DMA Engine Framework has been customized and a DMA proxy test application is created in the User space.

Once the design in Vivado is finished, the *design_1_wrapper_hardware_platform_X* folder created in SDK is imported to the Petalinux project and the Petalinux project is configured taking into account the previous chapter's considerations. A module called *dmaproxy* is created and it contains the DMA proxy device driver that is linked to the DMA Engine Framework, which, in turn, is linked to the AXI DMA device driver. The device tree is modified to inform to the PS that two DMA transfer channels is used: One channel to transfer the data from PS to AXI DMA hardware module and other one doing the contrary. Moreover, an application called *dmatest* is created and it contains the DMA proxy test application.

[2] After performing some test, this is the best amount of data to achieve a good transfer data rate without collapsing the AXI DMA.

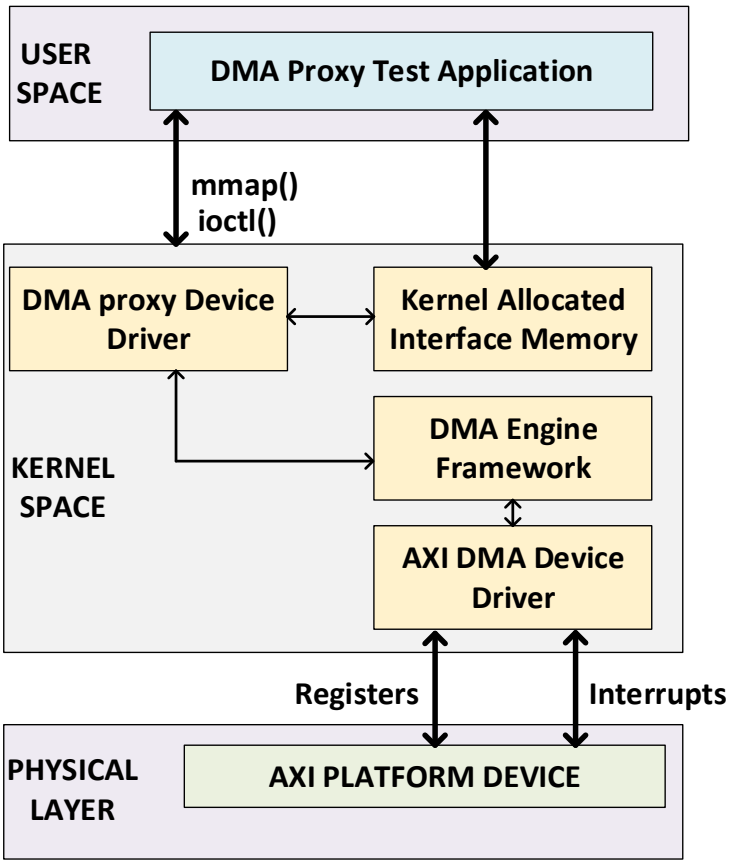

Figure 5.3: Linux DMA structure.

Once the Petalinux project is built, the necessary files are exported to the SD card. Then, the SD card is inserted in the ZYNQ platform and the ZYNQ turns on. First of all, the *dmaproxy* module located in the root file system is mounted with the command *insmod* (Figure 5.4). Then, the two channels are created and set up as character devices to allow user space control. After that, the user application *dmatest* can be used to make the IQ data transfer (figure 5.7).

```
root@xilinx-zc706-2017_2:~# insmod /lib/modules/4.9.0-xilinx-v2017.2/extra/dmaproxy.ko
dmaproxy: loading out-of-tree module taints kernel.
dma_proxy module initialized
TX channel.
channel achieved.
CLASS done.
device driver done.
Allocating uncached memory at 0xF0A7C000
RX channel.
channel achieved.
CLASS done.
device driver done.
Allocating uncached memory at 0xF0A91000
```

Figure 5.4: Inserting the DMA proxy device driver.

Application *dmatest* has the following arguments:

- -i : Initialize the GPIOs. Through the virtual file system *sysfs* the pins of each AXI GPIOs (32 pins or bits each one) are configured setting the directions (input or output).

- -s "path of the IQ data file": The direction of the IQ data source file is specified.

- -d "path of the IQ processed data file": The destination of the processed IQ file.

- -b "on|off": enable (on) or disable (off) the created hardware module (MY_RTL) in PL.

- -a "value": if we want to give a value to the A register, in order to make the addition between the incoming IQ data and this register. If the hardware module is disabled, this addition is not going to be performed.

- -k : it shows the value of the accumulator.

In the DMA proxy test application a loop is defined to transfer 82.58 MBytes of IQ data in partitions of 32 KBytes. That is, in each loop 32 KBytes of IQ data is sent from the PS to the PL and another 32 KBytes of processed data is sent back to the PS. The kernel space shares an amount of memory with the user space for each created DMA subchannel (Kernel Allocated Interface Memory). The DMA proxy test application is able to send and receive 32KBytes from that shared memory using the mmap() function. The function ioctl is used to start a transfer in the specified channel and it waits until the transfer is done (blocking function), that is, until the AXI DMA hardware module sends an interrupt to the PS.

Before creating a shared memory for the user and kernel space, another alternative without shared memory was tried to implement using the functions *copy_from_user()* (copy that from the user space to the kernel space) and *copy_to_user()* (from kernel space to user space). It was seen that this functions give a higher latency than the shared memory structure and that is why this alternative was not chosen.

5.4 Results

Figures 5.5 shows the first IQ data values and figure 5.6 the IQ processed data results after implementing the *dmatest* application of figure 5.7. The addition between the A register (value of 10) and IQ data chunks of 4 Bytes is successfully performed (4 Bytes of data correspond to 8 hexadecimal values). Figure 5.8 shows the hardware signals through the Integrated Logic Analyzer (ILA). The AXI DMA receives 4 Bytes IQ data from AXI FIFO 1 (FIFO1_TDATA_MASTER) and the resultant addition is sent to AXI FIFO 2 (FIFO2_TDATA_SLAVE). For example, if 0xFEEFF6E5 is sent, the output resultant data (considering A equals to 10) is 0xFEEFF6EF. The correct operation of the design is verified.

```
root@xilinx-zc706-2017_2:~# hexdump -C /iq_datei/DRN_NDS_justagc.iq2048 |head
00000000  e5 f6 ef fe ab fe 9f fd  80 fb 21 07 5c f5 a3 07  |..........!.\...|
00000010  ba f5 6c f5 61 ec 4f f3  86 f0 2c f3 eb 00 7a ea  |..l.a.O...,...z.|
00000020  fc fe 91 fa 33 f9 dc 03  61 02 f2 f6 d4 13 7d ff  |....3...a.....}.|
00000030  40 0e 53 09 f4 f4 5d 00  d0 fa 4f fd b4 03 3f fc  |@.S...]...O...?.|
00000040  5e f5 89 fd 4d 03 2f 02  4b fb 78 fe bc 01 47 02  |^...M./.K.x...G.|
00000050  cb 03 4d fa 43 04 5b f8  e4 fb 0b 01 70 00 c5 02  |..M.C.[.....p...|
00000060  33 10 67 07 90 fe 4a 09  1e eb 05 fe b0 f7 f7 fb  |3.g...J.........|
00000070  df 01 cf 02 88 06 50 fd  f3 0b 31 f7 af 06 35 05  |......P...1...5.|
00000080  9e 03 53 0f 5f 08 e5 fe  85 0a 4e f9 84 07 6b 06  |..S._.....N...k.|
00000090  6f 05 da ff b5 03 1e f2  ea fa 90 ee 1f fc b9 f1  |o...............|
```

Figure 5.5: IQ data file transferred from PS to PL.

```
root@xilinx-zc706-2017_2:~# hexdump -C /tmp/output.iq2048 |head
00000000  ef f6 ef fe b5 fe 9f fd  8a fb 21 07 66 f5 a3 07  |..........!.f...|
00000010  c4 f5 6c f5 6b ec 4f f3  90 f0 2c f3 f5 00 7a ea  |..l.k.O...,...z.|
00000020  06 ff 91 fa 3d f9 dc 03  6b 02 f2 f6 de 13 7d ff  |....=...k.....}.|
00000030  4a 0e 53 09 fe f4 5d 00  da fa 4f fd be 03 3f fc  |J.S...]...O...?.|
00000040  68 f5 89 fd 57 03 2f 02  55 fb 78 fe c6 01 47 02  |h...W./.U.x...G.|
00000050  d5 03 4d fa 4d 04 5b f8  ee fb 0b 01 7a 00 c5 02  |..M.M.[.....z...|
00000060  3d 10 67 07 9a fe 4a 09  28 eb 05 fe ba f7 f7 fb  |=.g...J.(.......|
00000070  e9 01 cf 02 92 06 50 fd  fd 0b 31 f7 b9 06 35 05  |......P...1...5.|
00000080  a8 03 53 0f 69 08 e5 fe  8f 0a 4e f9 8e 07 6b 06  |..S.i.....N...k.|
00000090  79 05 da ff bf 03 1e f2  f4 fa 90 ee 29 fc b9 f1  |y...........)...|
```

Figure 5.6: IQ processed data file transferred from PL to PS.

The achieved data rate can be calculated taking into account the time duration of the application that is shown in figure 5.7. 82.58 MBytes corresponds to 10 seconds of signal and the transfer through the application lasts 3 seconds. The achieved data rate is around 26MBps while the minimum necessary data rate for the next designs is approximately 8MBps (2.048MSPS*4symbols/sample). In conclusion, this milestone is successfully completed having enough transfer data rate.

```
root@xilinx-zc706-2017_2:~# time dmatest -i -s /ig_datei/DRN_NDS_justage.ig2048 -d /tmp/output.ig2048 -a 10 -k -b on
GPIOs exported successfully
GPIOs initialized
Block activated
Value set up

Welcome to the application demo!

  Transfering data...
  Transfer done!
Accumulator's value = 259965041
FINISHED, See you!

real    0m3.001s
user    0m0.560s
sys     0m1.370s
```

Figure 5.7: *Dmatest* application: 82.58 MBytes are transferred from PS to PL, processed in PL and finally transferred from PL to PS in 3 seconds.

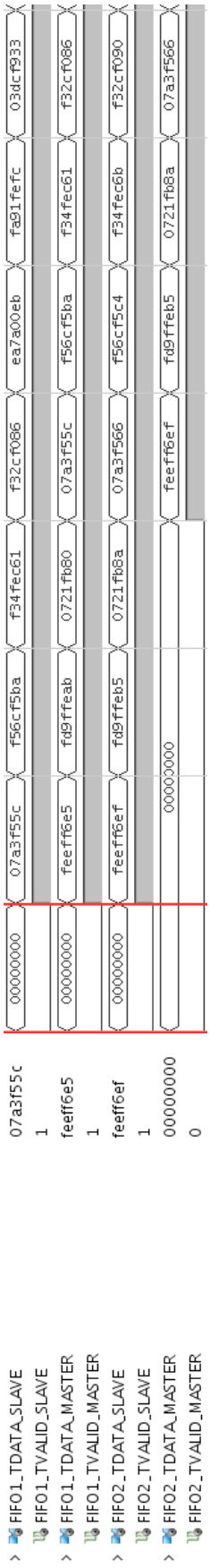

Figure 5.8: Signal debugging with the Integrated Logic Analyzer (ILA).

Chapter 6

Output Physical Layer integration

In this milestone one of the sub-modules of the DAB receiver is implemented on ZYNQ: the OPL (figure 6.1). The AXI DMA hardware module and its corresponding device drivers resulting from the second milestone are also used in this design. Furthermore, the already created OPL hardware module is implemented on the PL of the ZYNQ. The OPL hardware module is slightly modified in order to be compatible with the ZYNQ platform. Moreover, 4 RTL modules are created to synchronize the OPL with the implemented AXI DMA and two AXI GPIOs. In addition, two user applications are created: The first application configures the OPL through the register bank of the OPL and the second application performs the transfer of 2304 LLR (that corresponds to the output data of the IPL when one FIC block is processed) from the PS to the PL and the resultant processed data is sent back and stored in the PS . This milestone is completed when the correct behavior of the OPL is verified, that is, the processed data of the latter application is correct.

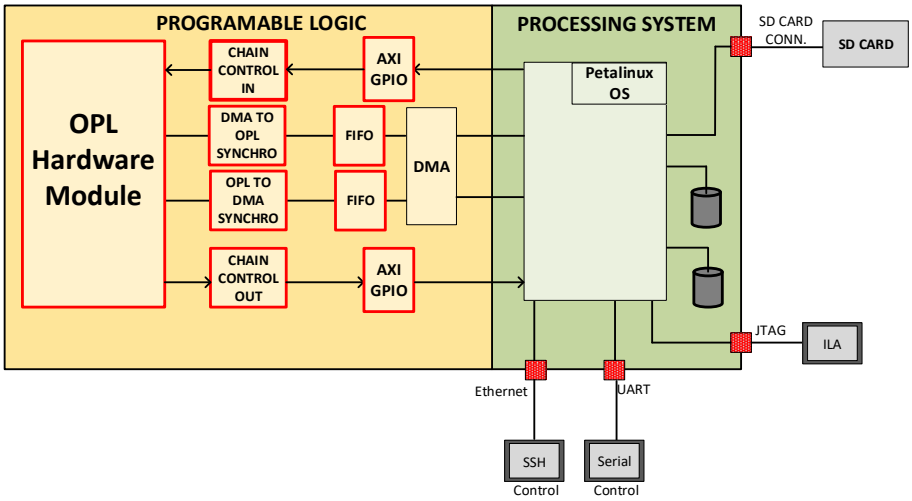

Figure 6.1: Block diagram of third milestone: In the PL, the OPL is implemented and 4 RTL modules are created to synchronize the OPL with the Xilinx provided modules (AXI DMA and AXI FIFOs). Two user applications are created.

6.1 Required background

Figure 6.2 shows an example of a DAB IQ incoming signal in mode I, meaning that the duration of the transmission frame is 96 ms. This real case IQ data example is used to test the DAB hardware implemented module. The first step of the DAB receiver is the search of the Null symbol. Once the null symbol is found, the frequency and phase offset

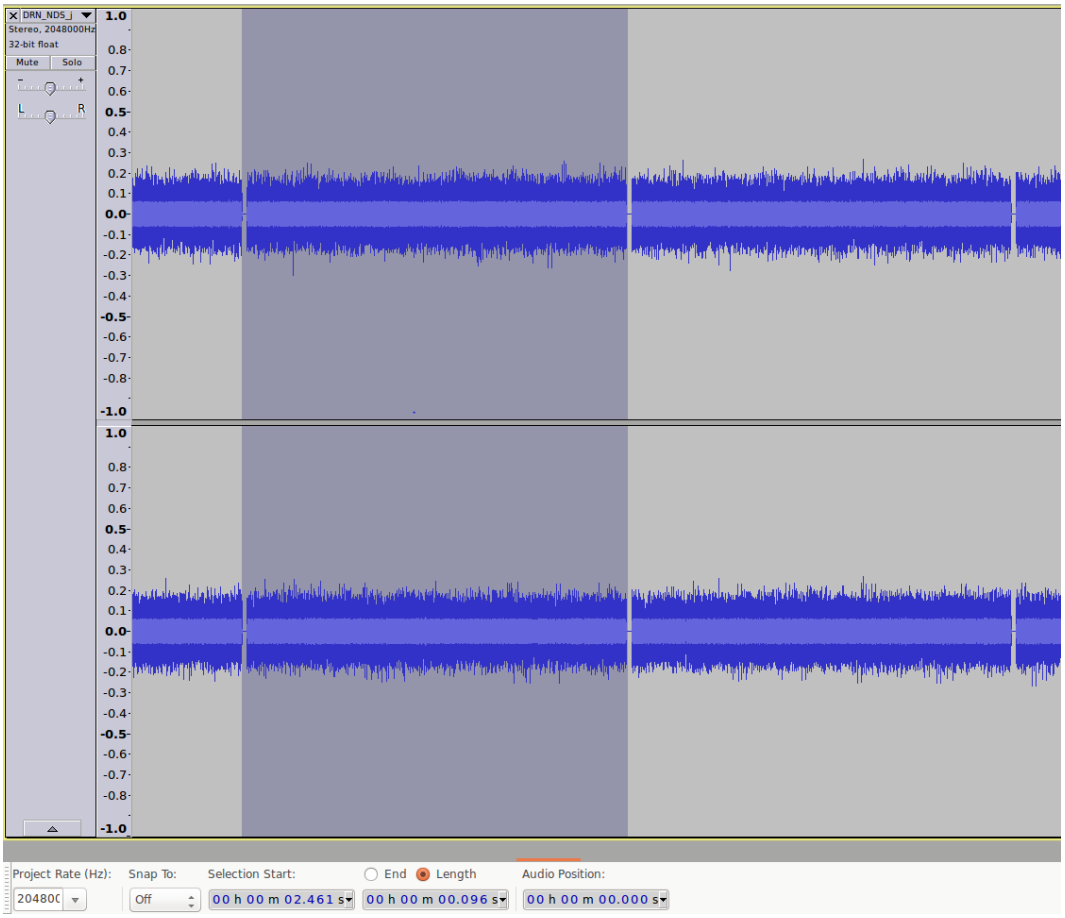

Figure 6.2: DAB incoming IQ signal.

is calculated by the reference synchronization symbol. After that, FIC and MSC symbols are sent to the IPL and the frequency compensation in each symbol is performed. Next, 1536 cells are defined performing the FFT of each symbol and then the phase offset is corrected. Afterwards, those cells are re-ordered in frequency domain and differentially demodulated. For each cell, composed by 4 Bytes (16 bit I, 16 bit Q), two LLR values of 1 Byte size each one is estimated. One LLR represent one encoded bit that was obtained by performing the convolutional encoding of the real content in the transmitter. Then, with 1536 cells 3072 LLRs are obtained. If the LLR data belongs to the FIC, it goes directly to the OPL. On the other hand, if the data is MSC data, it is stored in the deinterleaver block.

The OPL block contains three main sub-blocks: the depuncturer, the Viterbi decoder and the Energy Dispersal descrambler block. Depending on how was punctured the encoded data in the transmitter, the depuncturer takes between 1 and 4 LLR data and send it to the Viterbi Decoder, whose inputs are 4 LLRs. For instance, if the configuration of the depuncturer is to take 3 LLR data, it sends those 3 LLRs and an added Null Byte to the Viterbi. The first incoming LLR data belongs to the FIC and it is known how the encoded data was punctured. The FICs give information about the puncturing of the MSCs, since it is not known and varies. The Viterbi decoder performs the data error

correction and the output bits are sent to the energy dispersal descrambler. This block performs the XOR bit operation between the input data and a pseudorandom number sequence of length 511. After that, a shift register is implemented to send 1 byte of data.

Apart from the processing of the incoming DAB IQ signals, there is also an implemented register map that the user can write and read data from those registers. This control is used to configure the DAB blocks (i.e. enable the IPL, OPL...) or have information of important registers (i.e. the sub-channel, the amount of received CIFs...).

6.2 OPL implementation

The first FIC symbol of the DAB IQ signal is used to test the OPL hardware module. After performing the FFT of this symbol, 1536 cells are taken and 2304 of 3072 LLRs represent the first FIC block (FIC1) that describes the first MSC block (MSC1). These 2304 LLRs are sent to the OPL. As it is FIC data, it is known how the depuncturing must be done [1]. The output of the OPL is 768 decoded bits of information data.

The OPL hardware module, as the DAB receiver hardware module, is created to be used in different target devices and a new target device for the ZC706 Evaluation Kit is added. Depending on the specified target different memory IP core instances are used. For the OPL, two Block RAM IP cores are defined and used as buffers for the Viterbi and the Depuncturer sub-modules.

Two RTL synchronization modules are created to connect the OPL hardware module with the DMA (figure 6.3). 2304 LLRs (the size of 1 LLR is 1 byte) are sent from the PS to the DMA in chunks of 4 Bytes. The DMA sends the 4 Bytes data to a FIFO buffer provided by Xilinx. Because the input port *LLR_DATA* of the OPL has a size of 1 Byte, a FIFO data width converter from 4 Bytes to 1 Byte is used. *LLR_DATA* is delayed one cycle with the *LLR_DFF* RTL module because *LLR_VALID* must be sent one cycle before. Moreover, a RTL module called *OPL_DMA_SYNCH* is created in the output of the OPL module and it counts the number of valid output bytes. Then, the bus width converter from 1 Byte to 4 Bytes is used to send 4 chunks of data to the AXI DMA and the AXI DMA sends the processed data to the PS. If 96 Bytes have been sent, *OPL_DMA_SYNCH* RTL sets high the *tlast* signal during one clock cycle, meaning that it is the last processed byte. The reason of using a 4 to 1 Byte data width converter and vice versa, is because the AXI DMA input and output data ports have at least a size of 4 Bytes (It would be better, if the size of the input output data ports were 1 Byte, avoiding the data width converters).

There is also a hardware module called *chain_control* used to configure the OPL. Two

[1] See page 130 of [1] to know how the puncturing procedure works.

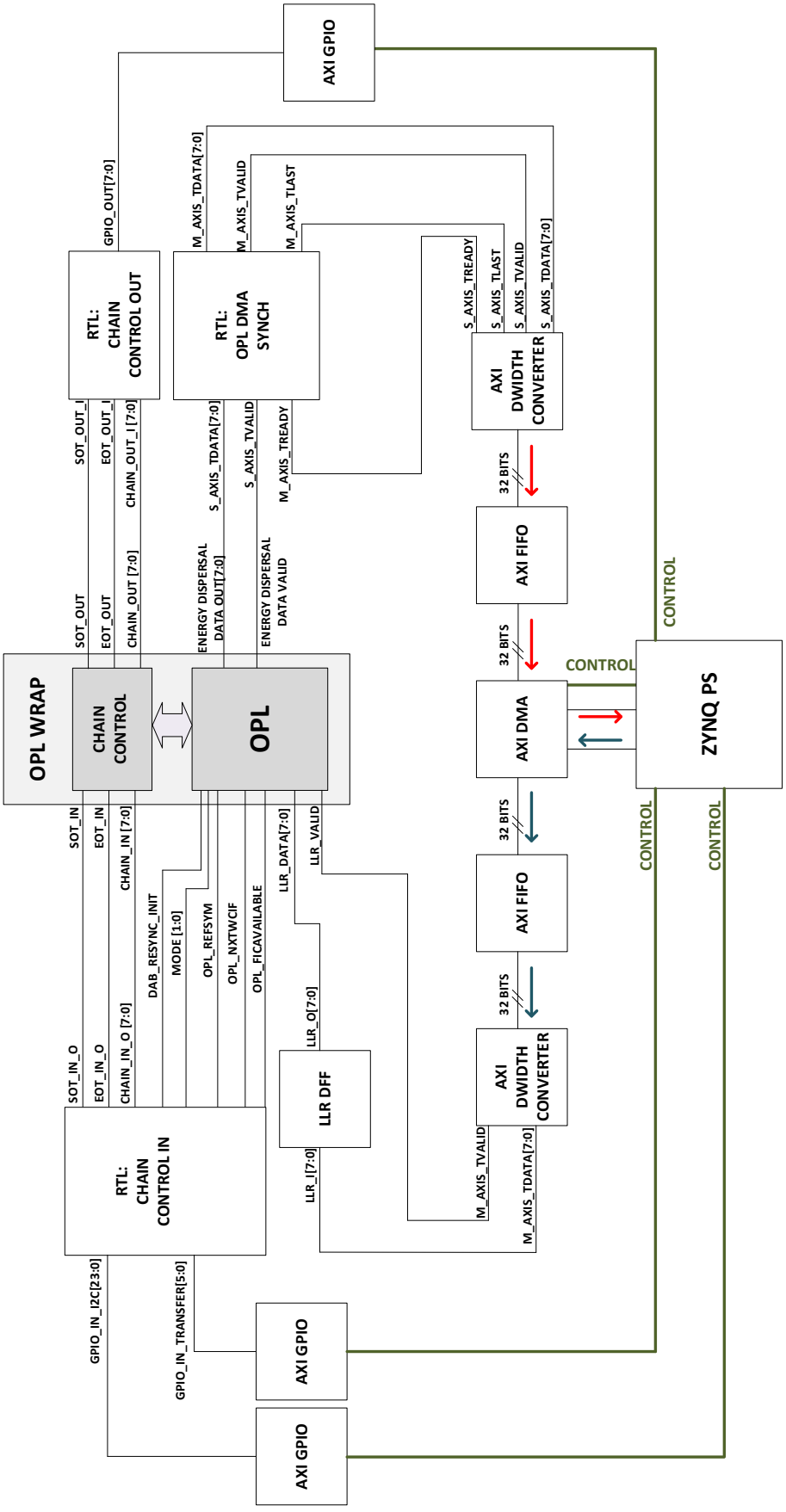

Figure 6.3: OPL design detailed block diagram. The significant input/output ports are shown.

RTL hardware modules and one user application (called *chaincontrol*) are created to configure the OPL from the PS. *Chain_control_in* RTL module takes data from two AXI GPIOS. One is used to control the *chain_control* module from the PS and figure 6.4 shows how the register is divided. The other one represents the outputs of previous hardware modules of the OPL (deinterleaver and IPL in particular) and figure 6.5 shows the register structure. On the other hand, *chain_control_out* RTL module is used to read the data stored in a register of the OPL. If the output port *eot_out* of the OPL is set to one, the data of *chain_out* is read and sent to the PS.

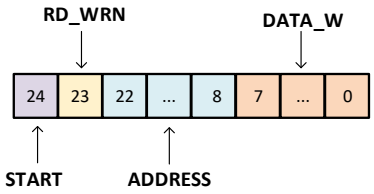

Figure 6.4: *Gpio_in_i2c* register.

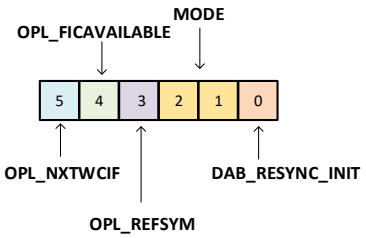

Figure 6.5: *Gpio_in_transfer* register.

The finite state machine of *chain_control_in* is shown in figure 6.6 and is specifically used to achieve a correct communication between the *chain_control* module and the PS. There are five states and when reset is enabled, it starts in IDLE_1. Once disabled, when the start flag of the register *gpio_in_i2c* is set to one, the RTL goes to state ADDR_HIGH . In this state, the last 7 bits of the address and the rd_wrn flag are sent to the *chain_control* module through the *chain_in_o* port and the port *sot_in_o* is set to one during one cycle, meaning that the start of a read or write operation is being performed. The next cycle, it goes directly to the next state, ADDR_LOW. This next state sends the remaining 8 bits of the address through *chain_in_o* and the following state depends on the rd_wrn flag. If this flag is enable, it means that a read operation is made and the next state is IDLE_2, that goes to IDLE_1 if the start flag is set to 0. If the write operation is enabled (the rd_wrn flag set to null), the next state is DATA_WR and the data to be written in the register located in the specific address is sent to the *chain_control* module through *chain_in_o*. After that, the following state is IDLE_2 and it waits until the start flag is set to null to go to IDLE_1.

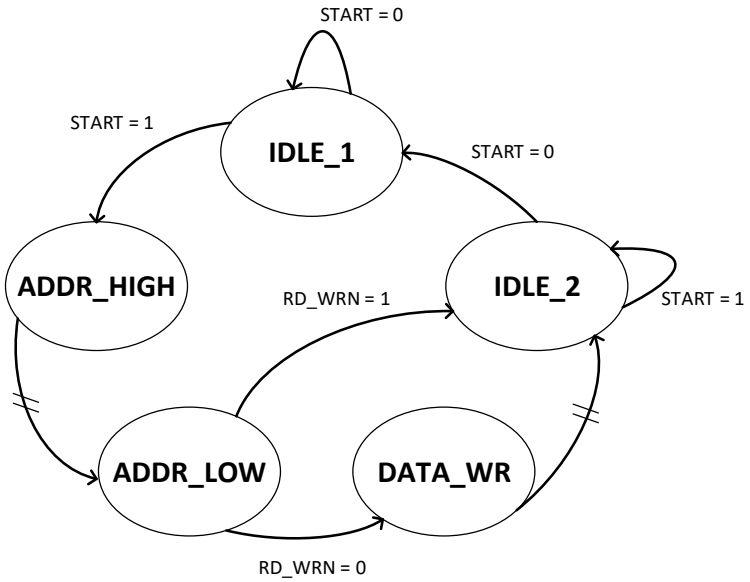

Figure 6.6: Finite State Machine of *chain_control_in* RTL module.

6.3 Test-bench analysis

One test-bench is created to ensure in a simulation the correct behavior of the OPL module. Figures 6.7 and 6.8 show the behavior of the chain control.

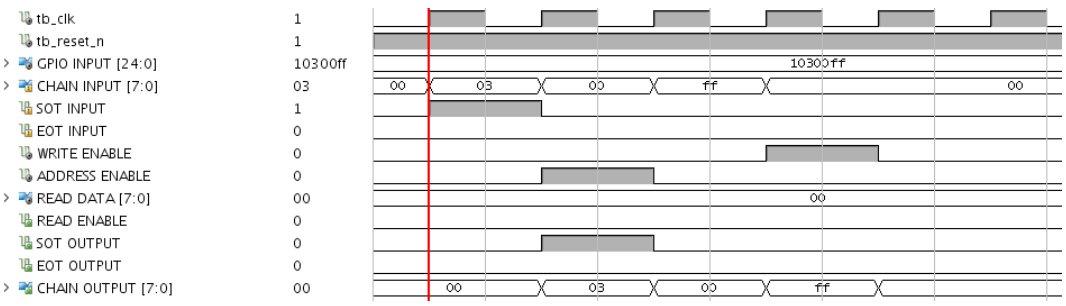

Figure 6.7: Testbench: Chain control in write operation. 0xFF is written in the register located in address 0x0300. This register enables or disables the sub-blocks of the OPL. In this case, all the OPL sub-blocks are being enabled.

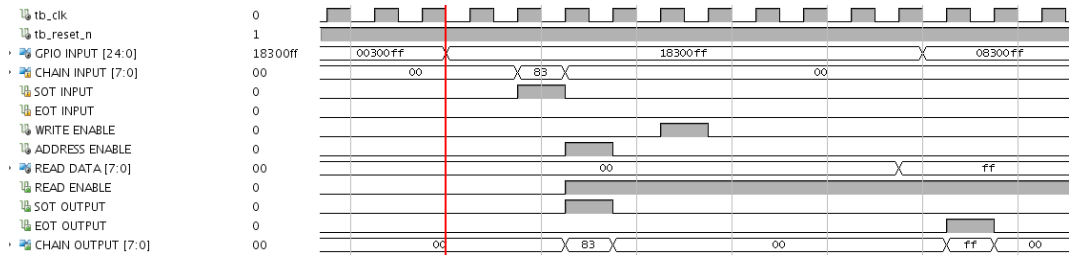

Figure 6.8: Test-bench: Chain control in read operation. The data of the register located in address 0x0300 is read. The read value is 0xFF and is sent to the *chain_control_out* RTL module setting to one the *eot_out* signal during one cycle.

Figures 6.9 shows the behavior of the OPL hardware module and how the *tlast* signal of the *OPL_DMA_SYNCH* is set to one when 96 Bytes of output data is sent.

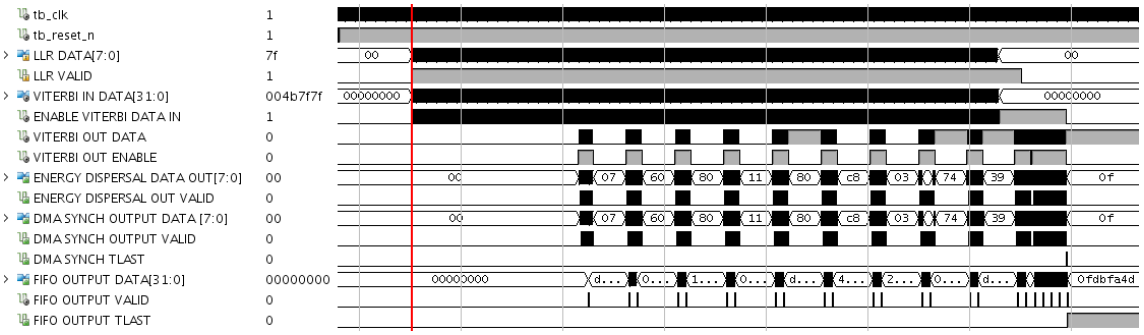

Figure 6.9: Test-bench: The first FIC symbol is sent (2304 LLRs) and the output is 96 bytes. See how the *tlast* is set to one when the last byte is sent.

Figures 6.10 is a closer look of the inner OPL signals. The *VITERBI IN DATA* signal is the output of the depuncturer. In this case, the depuncturer sends 3 LLR data and a added Null Byte to the Viterbi decoder. In addition, this test-bench shows how the Viterbi output data is accumulated in the energy dispersal through a shift register (each 8 cycles, 1 valid byte of *ENERGY DISPERSAL DATA OUT* is sent).

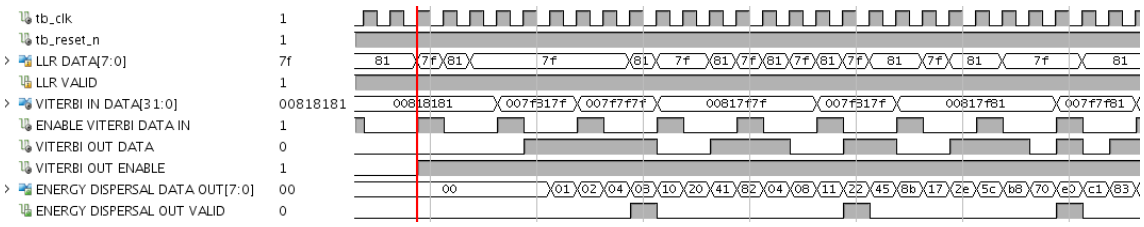

Figure 6.10: Test-bench: The depuncturing can be observed sending 3 LLRs and a null symbol to the Viterbi module. When *VITERBI OUT ENABLE* is set to one, each 8 cycles the signal *ENERGY DISPERSAL OUT VALID* sets to one, meaning that the data output of the energy dispersal is valid.

6.4 User applications: fictransfer and chaincontrol

An application called *chaincontrol* is defined to configure the OPL. It is also used in the complete DAB hardware module implementation. In figure 6.11 and 6.12 two examples are showed and the application has the following arguments:

- -a "hexadecimal value": Address of the registers to be written or read.
- -w "hexadecimal value" : Value to be written in a specific address.
- -r : Command to read the value of a specific address.

```
root@xilinx-zc706-2017_2:~# chaincontrol -a 0300 -w ff
OPL GPIO initialized
The value has been written!
FINISHED, See you!
```

Figure 6.11: Application: Chain control in write operation. 0xFF is written in the register located in address 0x0300.

```
root@xilinx-zc706-2017_2:~# chaincontrol -a 0300 -r
OPL GPIO initialized
Reading value...
Value= ff
FINISHED, See you!
```

Figure 6.12: Application: Chain control in read operation.The data of the register located in address 0x0300 is read.

Another application called *fictransfer* is created send the LLR data from a file and store the output in another file (figure 6.13). The application has the following arguments:

- -m "on|off": Enable or disable the sub-blocks of the OPL.
- -s ".llr" : Input LLR file.
- -d ".service" : Output file.

```
root@xilinx-zc706-2017_2:~# time fictransfer -s /llr_data/again.llr -d /tmp/output.service -m on
OPL activated


Welcome to the OPL demo!

 Transfering data...
 Transfer done!
FINISHED, See you!

real    0m3.010s
user    0m0.000s
sys     0m0.000s
```

Figure 6.13: Application: Transfer of a FIC symbol from the user space in PS to the PL. The processed data is stored in a file in the PS.

6.5 Results

The obtained output file (figure 6.14) of the latter application is correct because is the same output as the test-bench of figure 6.10. In addition,service labels as NDR 2 and NDR Data Service can be observed. This milestone is then completed due to the validation of the correct functionality of the OPL design.

```
root@xilinx-zc706-2017_2:~# hexdump /tmp/output.service -C
00000000  08 22 e0 d1 10 03 01 c0  42 10 03 00 60 3c 18 01  |."......B...`<..|
00000010  00 70 05 1c 01 01 20 3c  30 03 ff 00 00 00 04 5c  |.p.... <0......\|
00000020  36 04 01 d3 82 20 20 20  20 20 4e 44 52 20 32 20  |6....     NDR 2 |
00000030  20 20 20 20 20 07 c0 ff  00 00 00 00 00 00 1d 0a  |     ............|
00000040  37 05 e0 d1 10 03 4e 44  52 20 44 61 74 61 20 53  |7.....NDR Data S|
00000050  65 72 76 69 d6 e0 d0 75  e2 85 1e bf 4d fa db 0e  |ervi...u....M...|
00000060  00 00 00 00                                       |....|
00000064
```

Figure 6.14: Result of the processed data after sending a FIC symbol from PS to PL.

Chapter 7

Digital Audio Broadcast integration

7.1 DAB hardware module implementation

At this point, the Linux OS is correctly installed in the PS, the AXI DMA implemented on the FPGA of the ZYNQ works at an enough data rate and the OPL hardware module is correctly implemented on the ZYNQ and works successfully. The fourth and last milestone consist of the implementation of the entire DAB Hardware module on ZYNQ (figure 7.1). The hardware design is similar to the one in the third milestone. Instead of implementing the OPL module, the entire DAB receiver hardware module is implemented and it is slightly modified in order to be compatible with the ZYNQ. The two RTL modules used to configure the OPL in the third milestone is also used in this design to configure not only the OPL, but also the rest of sub-modules of the DAB. The remainder two RTL modules are different from the OPL design, but the task is the same: synchronization between DAB and AXI DMA and vice versa. On the other hand, two applications are created in the PS. The first application is the *chaincontrol*, the same one as in the third milestone. The second application performs the transfer of IQ data from the PS to the PL. The IQ data is taken through USB from the tuner that is connected to an antenna. The processed data in the PL is sent back to the PS and stored in a file. The application selects the desired sub-channel of the processed data to be decoded. The decoded data goes to a speaker that is connected through USB to the PS and the music starts. This final milestone is completed when music in real time is listened correctly.

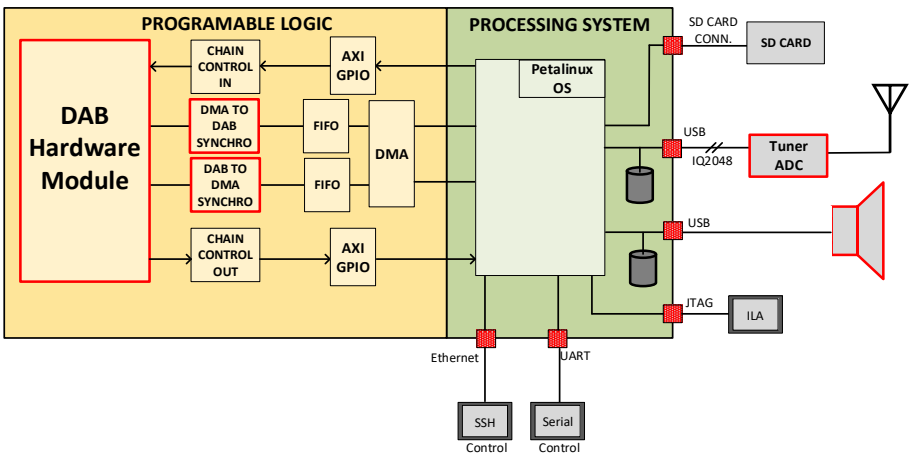

Figure 7.1: Block diagram of final milestone: The entire DAB receiver implemented on the ZC706 Evaluation Kit. The blocks marked in red are the additions comparing to the previous design.

Figure 7.2 shows a detailed block diagram of this design. The structure is similar as previous designs, using an AXI DMA hardware module to transfer data from PS to PL and vice versa. The DAB hardware module is called *DAB TOP* and inside it, there are five sub-modules: IPL, Deinterleaver, OPL, Bit Error Rate module and Service module. The first three modules were already described before. The Bit Error Rate module, as the name says, calculates the bit error rate of the incoming data. The Service module is the last sub-module of the DAB module and it structures the output data of the OPL in an ordered way. Figure 7.3 shows the output data structure when synchronization is established with the incoming IQ data and figure 7.4 is the resultant data when synchronization is not achieved. These data structures was designed so that is easy to implement in hardware. In synchronization, every 24 ms a new block is being produced. If the DAB transmission mode is I, the processed data of one incoming transmission frame is obtained by four of these blocks.

Each sub-module has a bank of registers with different address spaces. For instance, if the data located in address 0x0200 is written, the sub-module that contain this address in its register bank is configured (in this case, this register enables or disables the Deinterleaver sub-module). As in the previous design, the same two RTL modules are used to control the chain control modules of the DAB module.

Instead of transferring LLR data, IQ data is transferred to the FPGA through the AXI DMA. The real IQ data file of 82,58 MBytes used in chapter 5 is sent to the design in transfers of 30 KBytes that are divided in chunks of 4 Bytes. The *DMA DAB SYNCH* RTL module performs a delay of 8 cycles each time 4 Bytes are sent to the DAB module (8 cycles means 500 ns, the necessary time to perform all the processing of the incoming 4 Bytes in order to have a real time design). The *DAB DMA SYNCH* RTL module waits for the valid output data of the DAB module and a Finite State Machine is done to take the data considering if there is synchronization or not and their correspondent magic numbers.

The diagram flow of *DAB DMA SYNCH* RTL module is shown in figure 7.5. There are three main states: IDLE, NO_SYNCHRO and SYNCHRO. When reset is enabled, it starts in IDLE. If *S_AXIS_TVALID* and *M_AXIS_TREADY* are set to one, the RTL goes to the next state: READ_NO_SYNCHRO. Being *S_AXIS_TVALID* set to one means that the data in *S_AXIS_TDATA* is valid and *M_AXIS_TREADY* indicates if the connected slave module (AXI FIFO) is ready (set to one) or not (set to null) to accept data. In READ_NO_SYNCHRO state valid data in no synchronization mode is sent to the AXI DMA. the same condition as before is applied to take valid data from *DAB_TOP* module. A counter is created to set *M_AXIS_TLAST* to one each 26800 valid Bytes. If the last magic number in no synchronization and the first magic number in synchronization are found, *M_AXIS_TLAST* is set to one and the RTL goes to the fol-

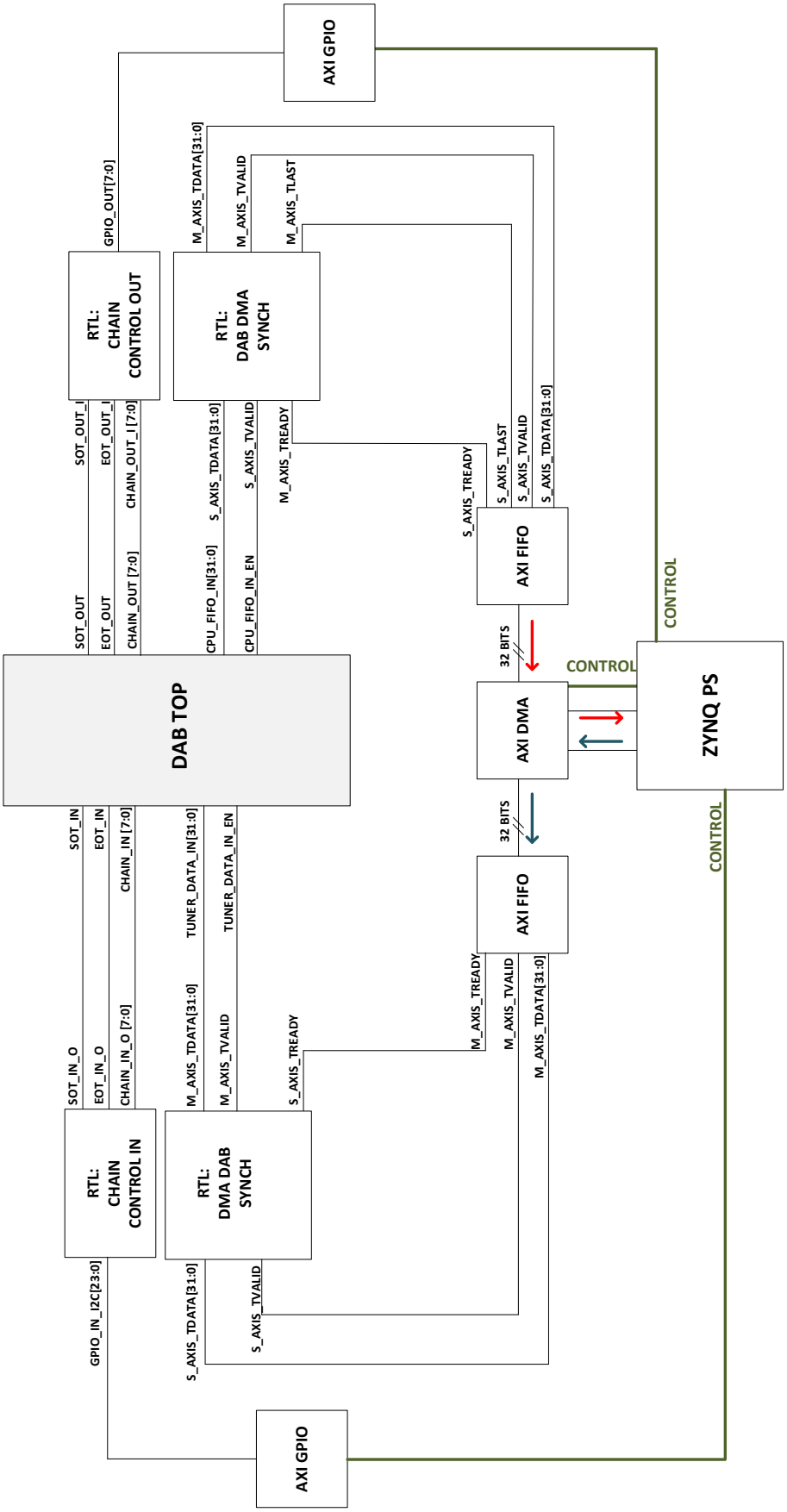

Figure 7.2: DAB design block diagram. The significant input/output ports are shown.

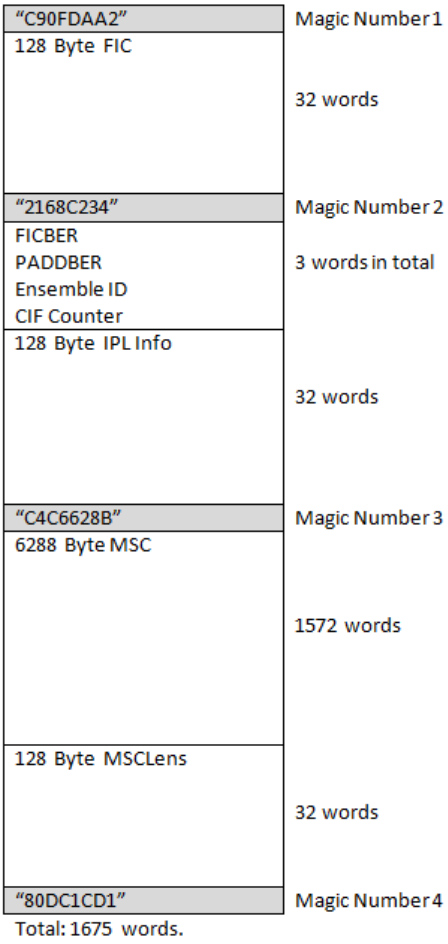

Figure 7.3: Output structure of the DAB module, once it is synchronized. Each word are 4 Bytes and the magic numbers are $\pi \times 2^{126}$ written as hexadecimal numbers.

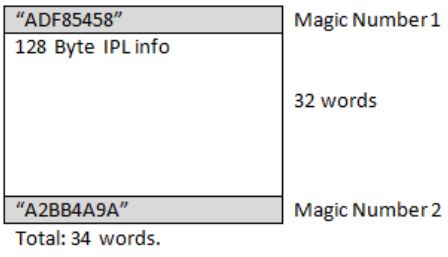

Figure 7.4: Output structure of the DAB module, once it is not synchronized. Each word are 4 Bytes and the magic numbers are $e \times 2^{62}$ written as hexadecimal numbers.

lowing state: READ_SYNCHRO. This last state reads the synchronized data that sends the *DAB_TOP* module. Using a counter, *M_AXIS_TLAST* is set to one each 26800 valid Bytes. The reason of sending 26800 valid Bytes is because the size of a processed transmission frame in mode I is taken, being four times 6700 Bytes (the size of 24 ms DAB processed signal).

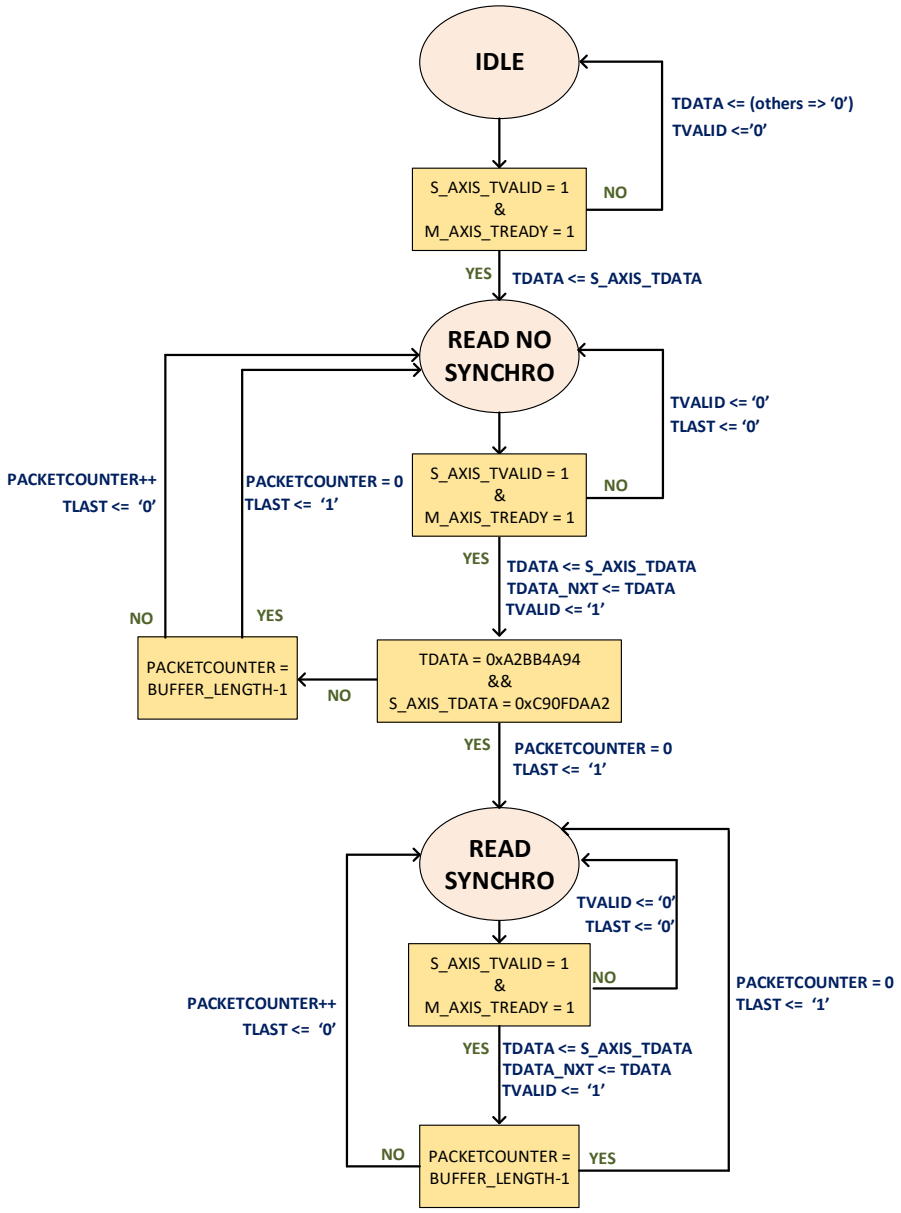

Figure 7.5: Diagram flow of the *DAB DMA SYNCH* RTL module.

7.2 Testbench analysis

In figure 7.6 the initialization of the *DAB_TOP* module through the chain control can be observed.

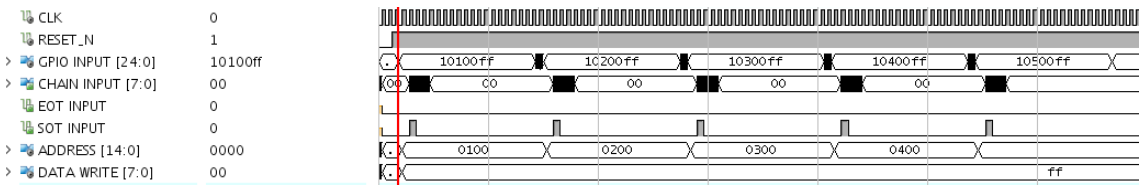

Figure 7.6: Testbench : Chain control in write mode. Initialization of the *DAB_TOP* module

The *DMA_DAB_SYNCH* rtl module delays the incoming data from the AXI DMA.

Figure 7.7 shows how the delay of 8 cycles is performed.

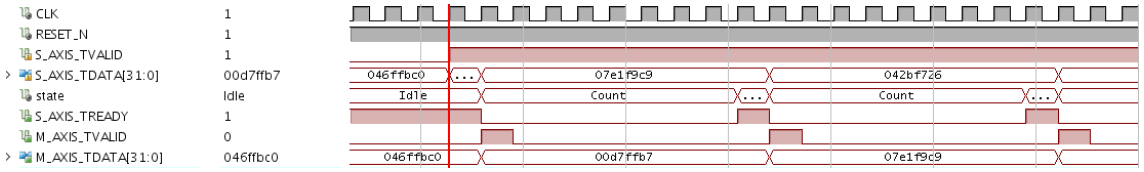

Figure 7.7: Testbench: *DMA_DAB_SYNCH* rtl module.

The IPL module of *DAB_TOP* performs the phase and frequency compensation, the frequency deinterleaving, the differential demodulation and the LLR estimation of the incoming symbols. The signals in figure 7.8 are divided in different colors. The purple signals represents the FFT process of the incoming data and the blue ones the estimated LLR data to send to the deinterleaver or to the OPL. It can be observed the real and imaginary part of the LLR (1 Byte size each signal).

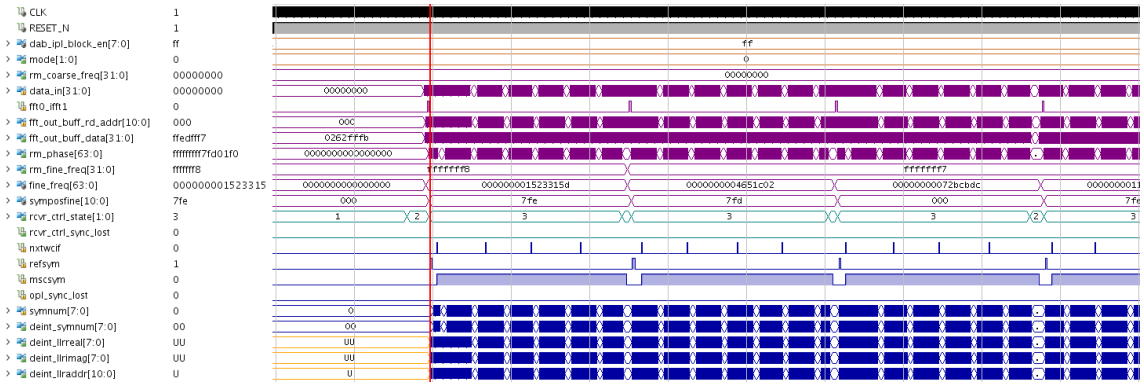

Figure 7.8: Testbench: IPL module in DAB.

The OPL module of *DAB_TOP* performs the depuncturing, the Viterbi decoding and the Energy Dispersal descrambling. Figure 7.9 shows the behavior of the OPL. See how the CIF number and the *fic0_msc1* signal is changing. If the latter signal is null the sent Byte belongs to FIC and if it is 1, the output Byte belongs to MSC.

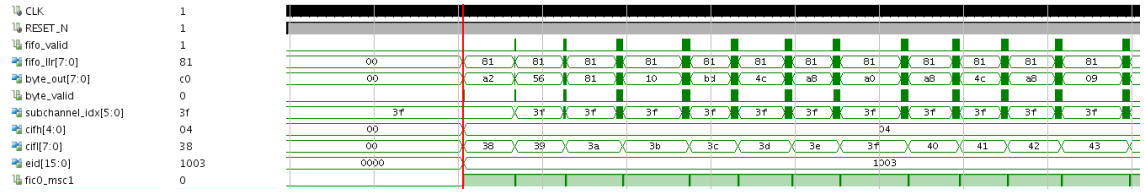

Figure 7.9: Testbench: OPL module in DAB.

The *DMA_DAB_SYNCH* rtl module sends to the AXI DMA the no synchronized and synchronized blocks in a structured way. *PacketCounter* signal counts the number of sent

chunks of 4 Bytes. If *PacketCounter* is equal to 6700, the *tlast* signal is set up. However, see that in READ_NO_SYNCHRO state the *tlast* is set up early because the first magic word in synchronization mode after the last magic word in no synchronization mode is found.

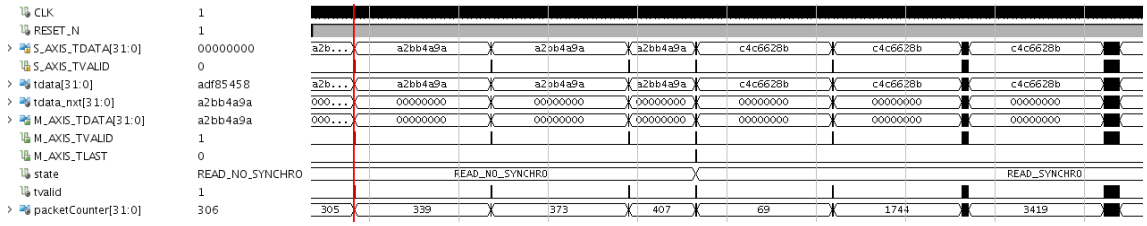

Figure 7.10: Testbench: *DAB_DMA_SYNCH* rtl module..

7.3 User applications

First of all, an user application is created to test the processing of real IQ data content in the DAB hardware module. This application is not included in the Petalinux project, but in Xilinx SDK. An Ethernet connection between the Xilinx SDK and the ZYNQ through TCF agent is performed and the application can be built from the SDK. Then, it is not necessary to do all the process of building the Petalinux project and exporting the important files to the SD card each time the application is changed. Another possibility is to cross compile the application and send to the ZYNQ through *scp* the built application. In this last alternative, a Makefile is created to perform an ordered cross compilation including the necessary libraries, headers and *C* files.

Figure 7.11 shows the final used user applications. The *chaincontrol* application is the same as the previous created one in the third milestone, used to configure and read important information of the sub-modules through their corresponding register banks. Furthermore, a new application is created called *dabaudio*. This application is connected through TCP to the *rtltcp* application in order to receive real time IQ data from the antenna and the tuner. *Dabaudio* performs the processing of the IQ data, stores the resultant data and sends the data corresponding to an user desired sub-channel to the *decoder* application. This latter application decodes the data to a pcm file and the *aplay* Linux function is run to play the pcm file in some speakers that are connected by USB.

Figure 7.12 describes more detailed the work-flow of *dabaudio*. It is composed by two threads: *tx_thread* and *rx_thread*. On the one hand, *tx_thread* asks for IQ samples to *rtltcp* application. *Rtltcp* takes a variable amount of samples from the tuner and sends the samples to *dabaudio*. *Tx_thread* asks for samples to *rtltcp* until the amount of received samples is bigger than the DMA transfer data size: 30KBytes. Then, transfers of

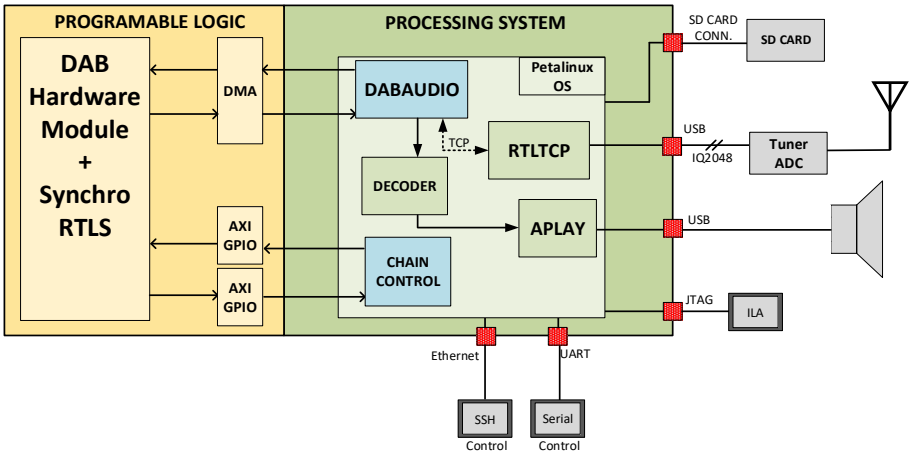

Figure 7.11: User applications of DAB receiver design: *dabaudio* and *chaincontrol* are created applications and *decoder*, *rtltcp* and *aplay* were already created ones.

30KBytes to the AXI DMA in the PL are performed. *Tx_thread* waits when each transfer is done receiving an interrupt from the AXI DMA. Once the interrupt is received and the remainder received data is less than the DMA transfer data size, this thread starts again to the beginning asking again for new samples.

In the meantime, *rx_thread* waits for the processed data performed by the PL. Once processed data is available, PL sends to PS through the AXI DMA 26,8KBytes of data, that means the same size of the processing of an incoming DAB transmission frame. When *rx_thread* receives the interrupt from the AXI DMA, the received data is stored in a *.dabhw* file and the MSC data corresponding to an user specified sub-channel is sent to the decoder. Then, that data is decoded and the audio is played through the speakers with the *aplay* function. Once that data is decoded, the *rx_thread* starts again waiting for processed data.

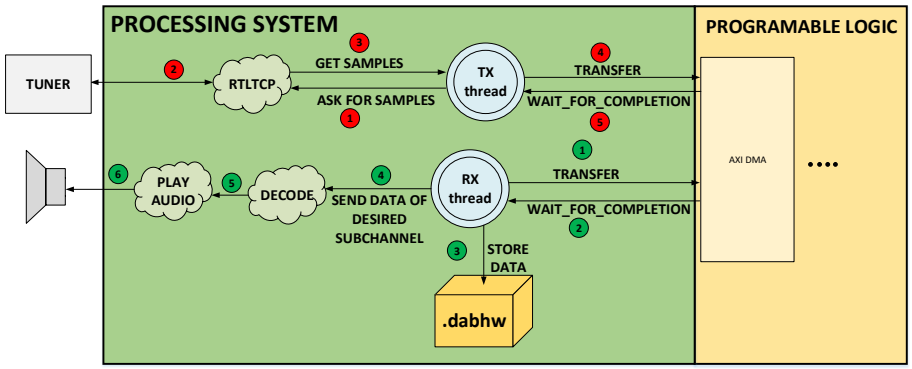

Figure 7.12: Work-flow of *dabaudio* application.

At first, to run *dabaudio* correctly, the DMA driver is initialized and the DAB sub-modules are enabled through the *chaincontrol* application. Moreover, the IP address and the port

of the *rtltcp* application must be specified to establish a TCP connection. Then, before running the application, the *rtltcp* is run first. The arguments of this application are shown next:

1. -a "0.0.0.0" : Specified address for *rtltcp*.

2. -p "1234" : Defined port for *rtltcp*.

3. -d "0": Defined direction for *rtltcp*.

An example of *rtltcp* is shown next:

```
#rtltcp -a 0.0.0.0 -p 1234 -d 0
```

The arguments of the *dabaudio* user application are set in the following order:

1. IP address of the *rtltcp* application.

2. Port where the *rtltcp* is located.

3. Desired frequency band.

4. Gain of the tuner.

5. Desired sub-channel.

The following *dabaudio* example decodes the data corresponding to the sub-channel 2 in the frequency band located at 1,81936 MHz:

```
#dabaudio 0.0.0.0 1234 181936000 12 2
```

In order to play the music of that decoded data, the application *aplay* is run in parallel to *dabaudio*:

```
#dabaudio 0.0.0.0 1234 181936000 12 2 | aplay -r 48000 -c 2
-f S16_LE -D "Device name"
```

7.4 Results

The successfully completion of this last milestone is verified due to the correct real time functionality of the *dabaudio* application. The user is able to select the frequency band, the gain and the desired sub-channel. When the application is run, the possible sub-channels in that frequency band are shown and, if the user selects non-existing sub-channels, it would be not possible to listen to them. Figure 7.13 shows the final physical structure of the DAB receiver design.

Figure 7.13: Final structure of the DAB receiver implementation on the ZC706 Evaluation Kit.

Chapter 8

Conclusion

The main objective of this master thesis is the implementation of the DAB receiver Hardware Module on the ZYNQ platform, in particular, on the ZC706 Evaluation Kit. First, the transfer data rate of the Xilinx provided AXI DMA in direct mode is tested and verified that is enough for the entire DAB Receiver design. Secondly, one of the sub-modules of the DAB Hardware Module is implemented on the PL of the ZYNQ platform and a complete design in the ZYNQ platform was created to verify the correct functionality of this sub-module. A test-bench is created to simulate the hardware design and a successfully real implementation with two user applications is achieved. One of these applications is used also in the final design to configure the sub-modules of the DAB Hardware Module. Finally, the entire DAB Receiver is implemented on the ZYNQ platform. The DAB Receiver Hardware Module is implemented on the PL and a test-bench is created to ensure the correct functionality. A complete user application is created to listen to music corresponding to a certain sub-channel of a certain frequency band in real time.

Table 8.1 shows the utilization of the final hardware design in the ZYNQ Z-7045 device. Due to the low percentage of used resources, the integration from the ZYNQ device Z-7045 to a smaller ZYNQ device can be performed such as the Z-7035 device.

	Utilization	Total	%
LUTs	57911	218600	26.5
Flip Flops	16926	437200	6.28
Block RAMs (32Kb)	142	545	29.08
DSPs	122	900	13.56

Table 8.1: Utilization in programmable logic of the ZYNQ Z-7045 device for the final design.

During this master thesis there have been challenges, which have been overcame. First of all, Petalinux Tools 2017.2 application was different to previous versions and time was required to understand the new structure with scarce resources. Moreover, while implementing the DMA on the ZYNQ, a lot of time was required to understand and develop the Xilinx provided device drivers in PS in order to achieve a correct communication with the AXI DMA in the PL. There was a small but essential spelling failure in the Xilinx provided *dmaproxy* device driver. Furthermore, issues were found related to the created block RAMs in the DAB receiver hardware module implementation on the ZYNQ device.

When single or dual port block RAMs are created, is very important set the operating mode to READ FIRST[1]. One last issue was the structure of the IPL. Because of the use of Verilog files that were created with C files by the high-level synthesis tool *Catapult C*, it was difficult to understand this sub-module and to implement it correctly on the PL.

In conclusion, the main objective of this master thesis is not only successfully completed, but also a complete DAB receiver is created and is possible to integrate the hardware design into a smaller ZYNQ device. From this point on, new objectives can be established to achieve a final hardware module implemented on an Application-Specific Integrated Circuit (ASIC) being attractive for the market.

[1] In particular, it is very important to set this operating mode to the created dual port Block RAMs, in order to be more robust against collision behaviors[11].

Chapter 9

Future work

A DAB receiver has been implemented on the ZC706 Evaluation Kit in this master thesis. In this chapter some ideas to improve the performed implementation are marked and the future work is drawn.

Regarding to the DAB receiver implementation, the Xilinx provided AXI DMA can be set in *Scatter/Gather* mode in order to reduce the workload of the processing system. In this mode, the PS sends a set of descriptors to the AXI DMA and while the AXI DMA performs the transfers operations, the PS can perform other tasks. In addition, the utilization of the DAB receiver hardware module implemented on the PL can be reduced. Design space can be saved if the number of Block RAMs of 32Kb is reduced in the deinterleaver module. The minimum number of 32Kb Block RAMs can be calculated improving then design efficiency. The number of LUTs can be also reduced replacing the Verilog files created by Catapult C into VHDL improved files. These files are located in the IPL module and represent the equalization process of the incoming OFDM symbols. If the number of LUTs is reduced to 47160, the implemented hardware design can be integrated in Z-7030 device without any risk (using the 60% of the total LUTs). On the other hand, if the number of LUTs is reduced even more and the number of 32Kb Block RAMs is reduced to 140, it would be possible to integrate in Z-7020.

Finally, if these improvements are performed and the best DAB hardware module solution is achieved, it is possible to think about the final main objective: Implement the DAB hardware module on an ASIC in order to come on to the market successfully.

Chapter 10

Economy Study

10.1 Workforce

The workforce is calculated depending on the type of professionals and the time devoted for each of them in the project development. A supervising person has dedicated a time period of 3 hours per week to guide and provide the steps of this diploma thesis. The cost of the supervisor is 112 €for a PhD professional according to the German government fees. The time dedicated to this project was 40 hours per week during 26 weeks, costing 23 €per hour according to the fees applied to a diploma thesis student by Fraunhofer IIS. The total workforce cost results in:

Project management: 26 weeks x 3h x 112 €	8.736 €
Project delivery: total amount of 1.040 h x 23 €	23.920€
Total Workforce:	**32.656€**

10.2 Equipment

The cost of the required equipment is summarized in the following list, taking into account the depreciation of them:

Licenses	459,33 €
Depreciation of personal computer	233,56€
Depreciation of external services	187,78€
Depreciation of operating systems	68,42€
Office supplies	150,00 €
Total Equipment:	**1.099,09€**

10.3 Summary

The total cost of the project is summarized in the next list:

Workforce	32.656,00 €
Equipment	1.099,09 €
Total:	**33.755,09€**

Appendix A

Create project in Petalinux

```
# Script that creates a Petalinux Project for ZC706 Evaluation Kit
#!/bin/bash
# Create project
read -p 'Project name: ' PROJECT_NAME
petalinux-create -t project -n $PROJECT_NAME -s bsp/xilinx-zc706-v2017
    .2-final.bsp
# Go to project
cd $PROJECT_NAME
# Configure project (SD configuration manually)
read -p 'Design Wrapper path: ' DESIGN_WRAPPER_PATH
petalinux-config --get-hw-description=$DESIGN_WRAPPER_PATH
# Build project
petalinux-build
# Bitstream Path
BIT_PATH_PET=components/plnx-workspace/device-tree-generation/design*
# Recipes Path
RECIPES_PATH=project-spec/meta-user
# Device Tree Blod Path
DTB_USER=project-spec/meta-user/recipes-bsp/device-tree/files
# First Stage Bootloader path
FSBL_PATH_PET=images/linux/zynq_fsbl.elf
# Generate BOOT.BIN file
petalinux-package --boot --fsbl $FSBL_PATH_PET --fpga $BIT_PATH_PET --
    uboot --force
# Package everything to a prebuilt folder for convenience
#petalinux-package --prebuilt --fpga $BIT_PATH_PET --force
# Configure rootfs (Fylesystem packages manually)
petalinux-config -c rootfs
# Create modules and apps
petalinux-create -t modules -n dmaproxy --enable
# Edit device driver of DMA (copy default dmaproxy file)
gedit $RECIPES_PATH/recipes-modules/dmaproxy/files/dma* &
# Edit Makefile of DMA (copy default Makefile of dmaproxy file)
gedit $RECIPES_PATH/recipes-modules/dmaproxy/files/Make* &
# Create and enable dabaudio application
petalinux-create -t apps -n dabaudio --enable
# Create and enable chaincontrol application
petalinux-create -t apps -n chaincontrol --enable
# Create and enable rtltcp application
petalinux-create -t apps -n rtltcp --enable
# Edit dabaudio application(copy default dabaudio file)
```

```
gedit $RECIPES_PATH/recipes-apps/dabaudio/files/dabaudio &
# Edit chaincontrol application(copy default chaincontrol file)
gedit $RECIPES_PATH/recipes-apps/chaincontrol/files/chaincontrol &
# Edit rtltcp application(copy default rtltcp file)
gedit $RECIPES_PATH/recipes-apps/rtltcp/files/rtltcp &
# Edit system_user.dtb device tree(copy default system_user file)
gedit $DTB_USER/system* &
# Configure rootfs
petalinux-config -c rootfs
# Configure kernel
petalinux-config -c kernel
# After editing correctly all the files and configuring
# the rootfs and the kernel, build again the project.
```

Appendix B

Mount Petalinux project in SD card

```
# A simple script: We are going to mount ext3 file of the petalinux
   project to /mnt and after that we copy it to rootfs partition of SD
   card
#!/bin/bash
#read -p 'Select ext: ' PATH_EXT
#read -p 'Path of SD rootfs partition: ' PATH_ROOTFS
#read -p 'Path of SD BOOT partition: ' PATH_BOOT
PATH_ROOTFS=/media/inigo/r*
PATH_BOOT=/media/inigo/B*
sudo cp BOOT.BIN system.dtb image.ub $PATH_BOOT
sudo mount rootfs.ext4 /mnt1 -o loop
cd $PATH_ROOTFS
sudo rm -rf bin boot dev etc home lib media mnt proc run sbin sys tmp
   usr var
sudo cp -r /mnt1/* $PATH_ROOTFS
sudo ls $PATH_ROOTFS
```

Appendix C

Petalinux Installation Requirements

```
-------------------------------------------------
##     PETALINUX 2017.2 INSTALLATION  ##
-------------------------------------------------

Petalinux 2017.2 Documentation:
In this link we can see the installation requirements:
 https://www.xilinx.com/support/documentation/sw_manuals/
 xilinx2017_1/ug1144-petalinux-tools-reference-guide.pdf



Download the following files in (https://www.xilinx.com/
support/download/index.html/content/xilinx/en/downloadNav/
embedded-design-tools/2017-2.html):

Intallation package: petalinux-v2017.2-final-installer.run
Board support package of the ZC706: xilinx-zc706-v2017.2-final.bsp
SDK 2017.2

Install petalinux 2017.2:
#./petalinux-v2017.2-final-installer.run installation_path
```

Appendix D

Running the Zynq ZC706

```
-----------------------------------------
##    RUNNING THE ZYNQ ZC706  ##
-----------------------------------------

1. Insert the SD card in the ZC706

2. Connect the UART port to the USB port of the PC and power on the ZYNQ.

3. Download the necessary drivers to detect the ZYNQ and install them.
   The next link provides the necessary drivers:

http://www.wiki.xilinx.com/Setup+a+Serial+Console

4. Open PuTTY and start the serial connection:

username: root
password: root

5. Run the device driver you have been created in Petalinux:

insmod  /lib/modules/4.9.0-xilinx-v2017.2/extra/dmaproxy.ko
```

Acknowledgments

I would first like to thank my thesis supervisors Juan Francisco Sevillano at the university of Navarra and Martin Speitel and Thomas Dettbarn at Fraunhofer IIS (Erlangen). During my stay in Germany I have always been treated very well in Fraunhofer IIS and I am very grateful for that. I could specialize in Linux, hardware design development and theoretical concepts about the DAB receiver and Martin Speitel and Thomas Dettbarn were always available whenever I ran into a trouble spot or had a question about my research or writing. Following the advice of my supervisor Juan Francisco Sevillano, a successfully completed and ordered thesis has been achieved.

I would also like to thank a worker of Fraunhofer IIS and, at the end, a very good friend who was involved in the validation survey for this research project: Xabier Abancens. Without his help and participation applied in all the workers, the validation survey could not have been successfully achieved.

Finally, I must express my very profound gratitude to my parents, to my brother Andoni and to my sisters Aintzane and Ainhoa for providing me a continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Bibliography

[1] *Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers.* http://www.etsi.org/deliver/etsi_en/300400_300499/300401/02.01.01_20/en_300401v020101a.pdf. Version: October 2016

[2] *Zynq-7000 All Programmable SoC Family Product Tables and Product Selection Guide.* https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf. Version: 2014-2017

[3] *Zynq-7000 All Programmable SoC ZC706 Evaluation Kit.* https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf. Version: January 2015

[4] Louise H. Crockett, Martin A. E. Ross A. Elliot E. Ross A. Elliot ; Stewart, Robert W.: *The Zynq Book.* https://is.muni.cz/el/1433/jaro2015/PV191/um/The_Zynq_Book_ebook.pdf. Version: 2014

[5] *PetaLinux Tools Documentation.* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1144-petalinux-tools-reference-guide.pdf. Version: June 2017

[6] *Vivado Tutorial.* https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2015x/VHDL/docs-pdf/Vivado_Tutorial.pdf. Version: 2013

[7] *Xilinx Software Development Kit (SDK) User Guide.* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug1145-sdk-system-performance.pdf. Version: November 2014

[8] Jonathan Corbet, Alessandro R. ; Kroah-Hartman, Greg: *Linux Device Drivers.* https://lwn.net/Kernel/LDD3/. Version: June 2017

[9] *AXI DMA v7.1.* https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf. Version: January 2018

[10] *AXI GPIO v2.0.* https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf. Version: October 2016

[11] *Block Memory Generator v8.3.* https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf. Version: April 2017