

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Mohammad Alizadeh Ghoulan 165521

EMBEDDED LINUX CUSTOMIZATION AND DRIVER DEVELOPMENT FOR SOC KIT

Master's thesis

Supervisor: Dr. Alar Kuusik
Senior Research
Scientist

Tallinn2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogiateaduskond

Mohammad Alizadeh Ghoulan 165521

**EMBEDDED LINUX-I KOHANDAMINE JA
DRAIVERITE ARENDAMINE SOC KIT
JAOKS**

Magistritöö

Juhendaja: Dr. Alar Kuusik
vanemteadur

Tallinn2019

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mohammad Alizadeh Ghoulan

01.05.2019

Abstract

Linux driver development for SoC FPGA which uses full-length/low-performance communication interface, and requires mastering in Linux programming, FPGA design and C coding, leads to complexity for developers. This article tries to provide a clear guidance of design flow by dividing the sequence in three different parts and supplying detailed information based on the author's experiences of Linux driver development for SoC Kit. XOR driver development using SoC FPGA Lightweight bridge interfacing presents simple and basic and at the same time comprehensive information for beginner developers. This work provides the most practical instruction for junior developers to verify hardware/software components and obtain a general overview of all creating special Linux distribution, driver development flow as well as FPGA design and C programming. This research is a part of Xiphera encrypting IP block project and has resulted in XOR driver for SoC Kit.

This thesis is written in English and is 54 pages long, including 3 chapters, 41 figures.

Annotatsioon

Embedded Linux-i Kohandamine ja dDraiverite ARrendamine SoC Kit jaoks

Linux'i draiverite arendamine SoC FPGA jaoks, mis kasutab täispika / madala jõudlusega kommunikatsiooniliidest ja vajab Linux'i programmeerimise, FPGA disaini ja C-kodeerimise rakendamist, viib arendajatele keerukuse. Käesolevas artiklis püütakse anda selgeid juhiseid disainivoogude kohta, jagades jada kolmeks erinevaks osaks ja esitades üksikasjaliku teabe, mis põhineb autori kogemustel Linux'i draiveri arendamisel SoC FPGA komplekti jaoks. XOR-i juhi arendamine SoC FPGA abil Kerge sillühendus pakub algajatele arendajatele lihtsat ja põhilist ning samal ajal terviklikku teavet. See töö annab noortele arendajatele kõige praktilisema juhendi riistvara / tarkvara komponentide kontrollimiseks ja üldise ülevaate saamiseks kõigist, mis loovad spetsiaalse Linux'i levitamise, draiveri arendamise voolu ning FPGA disaini ja C programmeerimise.

Selle projekti jaoks on vajaliku riistvarana kasutatud SoC FPGA komplekti. Juhatus on kombineeritud Altera Cyclone V FPGA ja ARM Cortex-9 Dual core-protssoriga. Kuigi turul on veel üks võistluslaud (Xilinx), kuid arvestades SoC FPGA komplekti konkurentsivõimelist hinda ja asjaolu, et Altera on nüüd osa Intelist, kes suudab pakkuda tehnilist tuge, alustas Xiphera oma projekti selle riistvaraplatvormi abil. See uuring on osa Xiphera IP krüpteerimisprojektist ning selle tulemuseks on XOR draiver SoC FPGA komplekti jaoks. See töö viib arendajatele üldise ettekujutuse sellest, kuidas Linux'i draiverit kujundada täispika liideseaga.

Lõputöö on kirjutatud [Inglise] keeles ning sisaldab teksti 54 leheküljel, 3 peatükki, 41 joonist.

List of abbreviations and terms

SoC	<i>System on Chip</i>
FPGA	<i>Field-Programmable Gate Array</i>
OS	<i>Operating System</i>
PC	<i>Personal Computer</i>
RTOS	<i>Real Time Operating System</i>
GNU	<i>GUN's Not Unix</i>
GCC	<i>GNU Compiler Collection</i>
API	<i>Application Program Interface</i>
CPU	<i>Central Processing Unit</i>
ARM	<i>Advanced RISC Machines</i>
RISC	<i>Reduced Instruction Set Computing</i>
U-Boot	<i>Universal Boot</i>
RAM	<i>Random access Memory</i>
ROM	<i>Read Only Memory</i>
SPL	<i>Secondary Program Loader</i>
I/O	<i>Input / Output</i>
IoT	<i>Internet of Things</i>
PROM	<i>Programmable Read Only Memory</i>
PLD	<i>Programmable Logic Device</i>
ASIC	<i>Application Specific Integrated Circuit</i>
ASSP	<i>Application Standard Parts</i>
FFTC	<i>Fast Fourier Transform Coprocessors</i>
DDUC	<i>Digital Down Converter-Up Converter</i>
VHSIC	<i>Very High-Speed Integrated Circuit</i>
VHDL	<i>VHSIC Hardwar Description Language</i>
FFT	<i>Fast Fourier Transform</i>
IFFT	<i>Inverse Fast Fourier Transform</i>
HPS	<i>Hard Processor System</i>
MPU	<i>Microprocessor Unit</i>
AXI	<i>Advanced eXtensible Interface</i>
MM	<i>Memory Mapped</i>
AMBA	<i>ARM Microcontroller Bus Architecture</i>
UART	<i>Universal Asynchronous Receiver Transmitter</i>

RBF	<i>Raw Binary File</i>
GHRD	<i>Golden Hardware Reference Design</i>
CD	<i>Compact disc</i>
SOF	<i>SRAM Object File</i>
SRAM	<i>Static RAM</i>
QPF	<i>Quartus Program File</i>
PIO	<i>Parallel Input / Output</i>
IP	<i>Intelligent Property</i>
TCL	<i>Tool Command Language</i>
DTS	<i>Digital Theater System</i>
DTB	<i>Device Tree Bulb</i>
BSP	<i>Board Support Package</i>
MTD	<i>Memory Technology Device</i>
RTS	<i>Remote Target System</i>
TUT	<i>Tallinn University of Technology</i>
VGA	<i>Video Graphics Array</i>

Table of contents

Author's declaration of originality	3
Abstract	4
Annotatsioon [Thesis title in Estonian]	5
List of abbreviations and terms	6
Table of contents.....	8
List of figures.....	9
1 Introduction	11
2 Customized Linux for Embedded systems	12
2.1 Why Linux?	12
2.2 Why Customization for Linux Is Required?.....	14
2.3 Linux for Embedded Systems.....	16
3 SoC FPGA's	22
3.1 SoC FPGA's Evaluation.....	23
3.2 SoC Kit	30
4 Driver Development for Embedded Linux.....	35
4.1 Software Requirement for Driver Development.....	35
4.2 FPGA Design Flow	39
4.3 Linux Distribution Development Steps	42
4.4 User Space Applications.....	50
4.5 XOR Driver Development	59
5 Summary.....	72

List of figures

Figure 1. Modular Structure of Linux	14
Figure 2. Running different applications in Kernel and User spaces	15
Figure 3. Five Elements of An Embedded Application	16
Figure 4. Generating an executable by Toolchain	18
Figure 5. Bootloader Initializing Flow.....	19
Figure 6. Generating PreLoader Sequences	21
Figure 7. Kernel Functionalities	21
Figure 8. SoC Kit	23
Figure 9. Advantages/disadvantages of FPGA and ASIC	26
Figure 10. FPGA, ASIC/ASSP and SoC comparison.....	27
Figure 11. SoC FPGA compare with ASIC	28
Figure 12. Fast Time to Market.....	29
Figure 13. SoC Kit Components.....	30
Figure 14. SoC Kit Block Diagram	31
Figure 15. Cyclone V SoC Bridges	32
Figure 16. Master and Slave Interfaces.....	33
Figure 17. AXI Master/Slave Interface.....	34
Figure 18. Quartus Prime 18.1 Lite Edition	36
Figure 19. Platform Designer Tool or Qsys	36
Figure 20. DS-5	38
Figure 21. DS-5 Debugger Tool.....	39
Figure 22. Customized IP.....	41
Figure 23. Avalon Interface.....	41
Figure 24. Device Tree Generating and U-Boot Compiling Steps.....	45
Figure 25. Kernel Configuration Window	47
Figure 26. Buildroot Configuration Window	48
Figure 27. Target Configuration of Buildroot	49
Figure 28. Toolchain Configuration of Buildroot.....	49
Figure 29. System Development Flow	50

Figure 30. Flash Device Connections with SoC FPG.....	51
Figure 31. Access to MTD from Terminal Window	52
Figure 32. MTD Partitions	52
Figure 33. Flash Memory Partitioning.....	53
Figure 34. Creating a New Connection from RTS	54
Figure 35. Configuration of RTS IP Address.....	54
Figure 36. DS-5 Debugger Menu	55
Figure 37. DS-5 Debugger Configuration Menu.....	56
Figure 38. DS-5 Debugger View	57
Figure 39. Full FIFO communication Qsys Design.....	58
Figure 40. FIFO Interface	59
Figure 41. Cyclone V SoC HPs Memory Map.....	63
Figure 42. Makefile compilation and transferring to the SD Card.....	70
Figure 43. XOR driver initializing.....	71
Figure 44. Communicating with driver and executing XOR operation.....	71

1 Introduction

Combination of FPGA and SoC into a single board, to utilize both parts' advantages, has optimized productivity and efficiency. Synchronically, Linux has been used to run these devices as it has been developers' favourite OS due to its flexibility and open source data base which provides freedom of design. Customization and driver development of embedded Linux for SoC Kit is an article focusing on Linux for embedded concept and its design flow. As the mentioned concept requires detailed knowledge and technical experience in Linux programming, FPGA design, VHDL/VeriLog coding and C programming, this research endeavours to provide a clear guidance by dividing design sequence into three different parts and step by step explanations. Driver development for SoC Kit requires Linux customization combining with Qsys design and HPS applications. These development steps are related to each other and must be done in a correct order otherwise the design process would be complicated and time consuming. This work tries to solve the complication of the design process and provide a clear guidance.

The final purpose of this project is to develop a XOR driver as a simple representation of the whole design sequence for SoC Kit. XOR driver verifies SoC Kit hardware/software facilities, such as GHRD and low-performance bridge communication between FPGA fabric and HPS sides using Avalon-MM interface. Full communication interface, which requires HPS to FPGA and FPGA bridges' interaction, has been described with a user space application. Full-length communication is used for Xiphera encrypting IP block development (which patented by Xiphera and is not a part of this work) and XOR driver development is fundamental validation of hardware/software components of SoC Kit. XOR driver provides Lightweight bridge communication by manipulation FPGA LEDs, which is a very good lead to obtain a general overview of full-length communication that can be used for more complicated projects.

This work consists of 3 sections. The first part investigates advantages of Linux customization and its necessity. The second section, SoC FPGA background, explains advantages and features of the SoC Kit, and finally, the third part engages in practicing and experimenting the real task/project. In this part, all three different stages of embedded Linux design flow, has been described separately. To provide the right materials for the mentioned sections, electronic and online sources have been also utilized.

2 Customized Linux for Embedded systems

Linux is a Unix-like operating system for computers and servers. It has been developed by Linus Torvalds, a computer science student at the University of Helsinki in 1991 [1]. The Unix system and the its hardware were both expensive and the Minix (a Unix version which was available for free) did not meet his needs. Therefore, he decided to develop a new Unix-like OS and shared his working result on the internet after six months of hard working (which had made a little progress toward general utility of the system) and found so many people who have the same desire. From 1991 Linux has been modified thousand times by different developers (as it is an open source OS) and has achieved a level of maturity that most of developers want.

In this chapter Linux properties and the reason that makes it the first option for the embedded systems development are discussed; while Windows is used widely (almost 75 %) by end user consumers or even giant companies [2].

2.1 Why Linux?

Referring to the market share statistics, most of computer users prefer to use Windows operating system while Linux has only 1.6 % of the whole market [2]. It is becoming more interesting, if the reality that Linux is free of charge and there is no need to pay for a license (except commercial distributions provided by vendors) is considered. The second important issue about Linux is its open source development property. So, there is freedom to develop new features and use whatever the project requires, and it is totally free of charge; but still people are using Windows OS incredibly more than Linux; but why?

After using Linux more actively in my professional life, the reason has been discovered. I must confess that I have learned computer working by Windows OS like the most people, but before my master I started to hear, learn and finally use Linux as the only OS every day at my job. Here are more evidences to compare both OS features. Linux features can be briefly listed as following:

- 1- Multiuser, multiprocessor and multiplatform: more than one user can be logged in to a single computer at the same time. Kernel (the core of Linux OS) multitasking property enables users to run multiple services on one computer; and finally, it has been developed for more than 24 hardware systems.
- 2- Flexibility: Linux can be configured for a variety of usage such as network host, router, web server, personal PC and many computing appliances that could be thought of.
- 3- Efficiency: the modular design of Linux enables to include only required components for running the desired service. Linux servers can work without any crash for even decades which makes it more reliable comparing to Windows.
- 4- Security: although Linux is open source which makes it risk-bearing, but it is highly secure OS because of its open source capability and a big group of volunteer developers who can easily identify attacking risks and modify Linux [3].

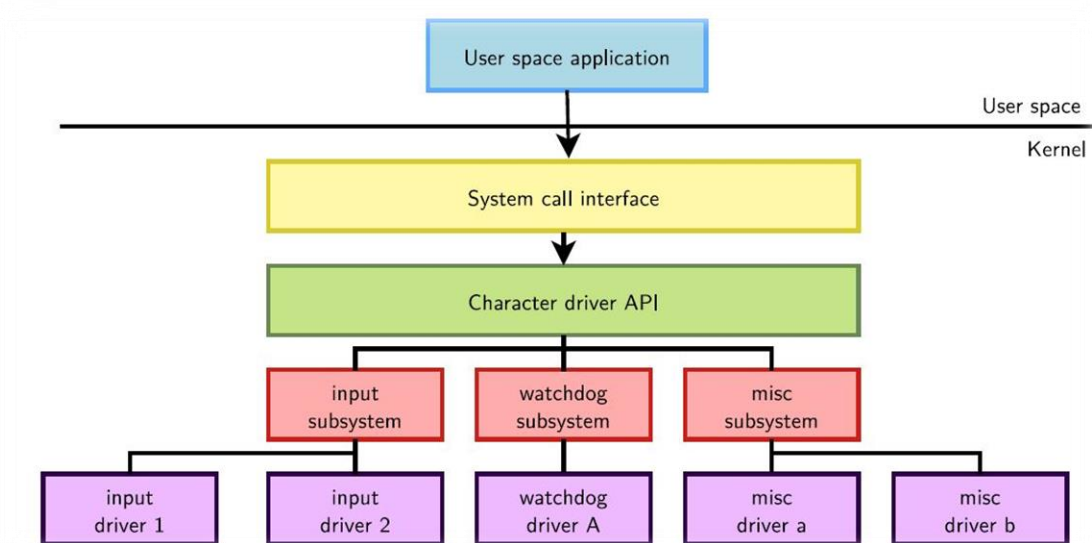
Considering above mentioned features, Linux is looking charming for developers who want to have a free-to-customization, secure, reliable and free OS. So why do amateur computer users prefer to use Window OS rather than Linux OS? The key words here are developers who are known as coder or programmer verses normal people who use computers as a device in daily life. Most people do not need to customize their OS, instead they want to have an already customized and ready to use device with all adjustments and configurations. Here the reason that Windows OS has taken the huge piece of market pie must be declared: ease of use! Windows OS has already all configurations done and needs only to be installed and run; moreover, it has a good support community which makes it attracted for end computer users. Windows has defined short cuts for different needs of users while in Linux it is needed to run a terminal window and execute a command (assuming these commands have been memorized after several time using) to have even a very basic feature.

On the contrary, developers prefer to use Linux because of its reliable and flexible features which enable them to configure OS for any desired requirement. More importantly, to develop a new project, which might need so many different adjustments

and configurations, the OS allows to play with different properties, use some of them or even develop new ones. To sum up, the customization capability of Linux in addition to its other features, are the main reasons that makes it interesting for developers.

2.2 Why Customization for Linux Is Required?

It was mentioned in the previous chapter that Linux modularity and open source features make it interesting for developers to easily program the desired services. A traditional monolithic OS uses one static-compiled image and runs in an all-or-nothing mode in which the entire OS needs to be restarted, if any element or application fails. While application and drivers in Linux have their own interface communicating with Kernel and interacting with the other applications independently. Looking at below figures describes the modular structure of Kernel:



^{1/2}

Figure 1. Modular Structure of Linux [4, p. 293] ³

¹ API: Application Program Interface, which interacts between Kernel and the applications run on it.

² Misc: Minimal Instruction Set computing, is a processor architecture with a very small number of basic operations [1].

³ Please note that all figures which have no citation in this project are screen shots from results of the original work by the author.

2.3 Linux for Embedded Systems

Linux is suitable for real time complex projects, especially when the connectivity among the applications and tasks is required. There may be a need for an up-to-date host machine (a laptop or personal PC) and a target board because Linux needs more resources compared to the traditional real time OS (RTOS). Linux is popular for embedded systems because of the same reasons were discussed in the previous chapter in general and mostly for PC's. Free software and open source community of Linux make it flexible for different variety of embedded systems projects, especially for limited investments with high returns. No need to mention again about Linux stability, reliability, network ability and multiuser/multitask capability features, which make it the favourite OS for embedded systems.

As it has been demonstrated in Figure 3, every project for embedded systems, which is running embedded Linux, needs below listed five elements to be obtained and configured properly. To begin:

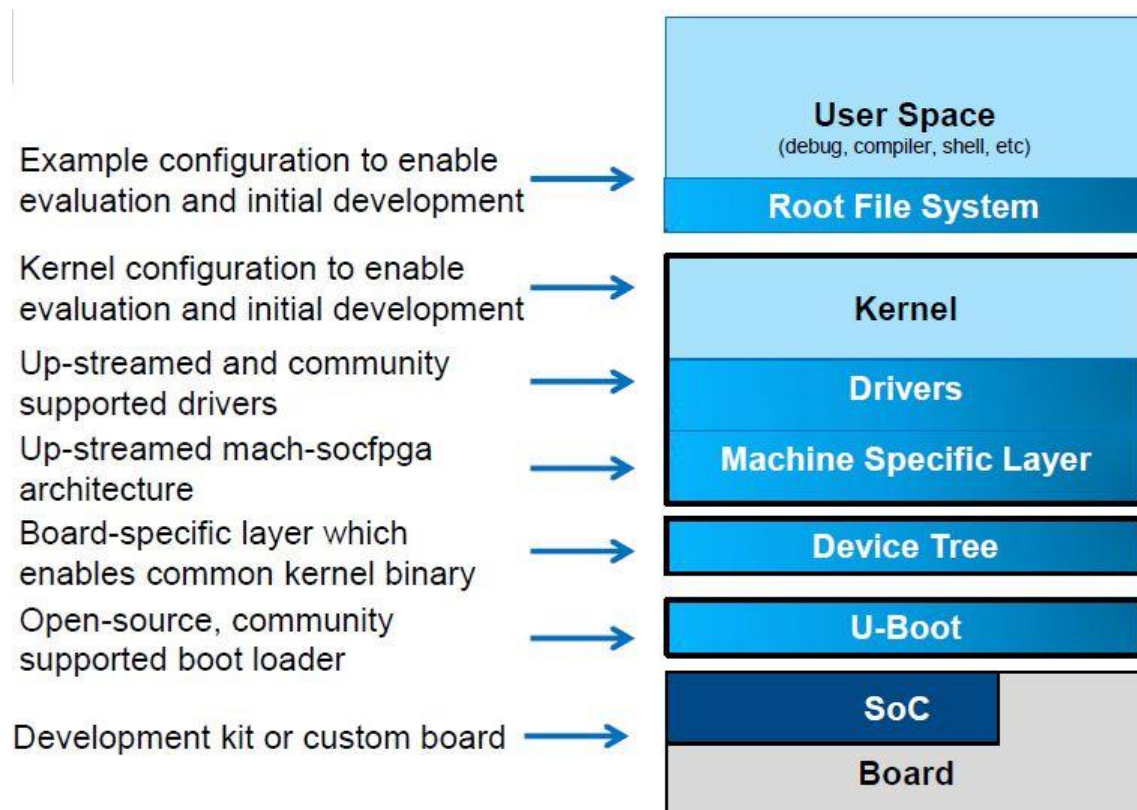


Figure 3. Five Elements of An Embedded Application [6, p. 76]

- 1- Toolchain: all steps for getting started a new project on an embedded board depend on Toolchain. Toolchain is the set of code generated by compiler and other tools for the target device. In Figure 3, Toolchain is not printed but it takes place in the bottom of all other steps. The reason it is not shown in this figure lies in the fact that Toolchain downloading, and configuration have been set by the host machine and the other steps, which depend on Toolchain, are being generated and put in a SD Card, which is used by the target board.
- 2- Bootloadr: after getting the Toolchain, the target board needs to be initialized; which Bootlaoder takes the responsibility here. Bootloader is shown in Figure 3 as U-Boot (Universal Boot) which itself is a step of booting.
- 3- Kernel: heart of the system which manages all resources and interfacing with the hardware. Kernel includes itself, drivers, machine specific layers and device tree in the figure. In fact, the device tree is kind of interacting between BootLoader and Kernel which defines hardware components of the embedded device. The device tree is loaded by BootLoader and passed to Kernel and Kernel image cannot be loaded without device tree compilation. A closer look at the device tree in the third chapter of the third part of this article is given.
- 4- Root filesystem: includes programs and libraries and runs after Kernel initialization [7].
- 5- Application: a collection of programs which you use for the project.

A closer look is taken at these five elements, as all these steps will be used later in order to develop a driver for own Linux distribution; so it is needed to have at least a general overview, (sometimes detailed expertise to solve problems is needed, as Linux is an open source OS and there is no guarantee for software developers that everything work well in all projects).

As it has been mentioned earlier, Toolchain is a set of tools that compiles required source codes into an executable file that can be run on the embedded board; and it is absolutely needed to be done before continuing the other steps of design sequence.

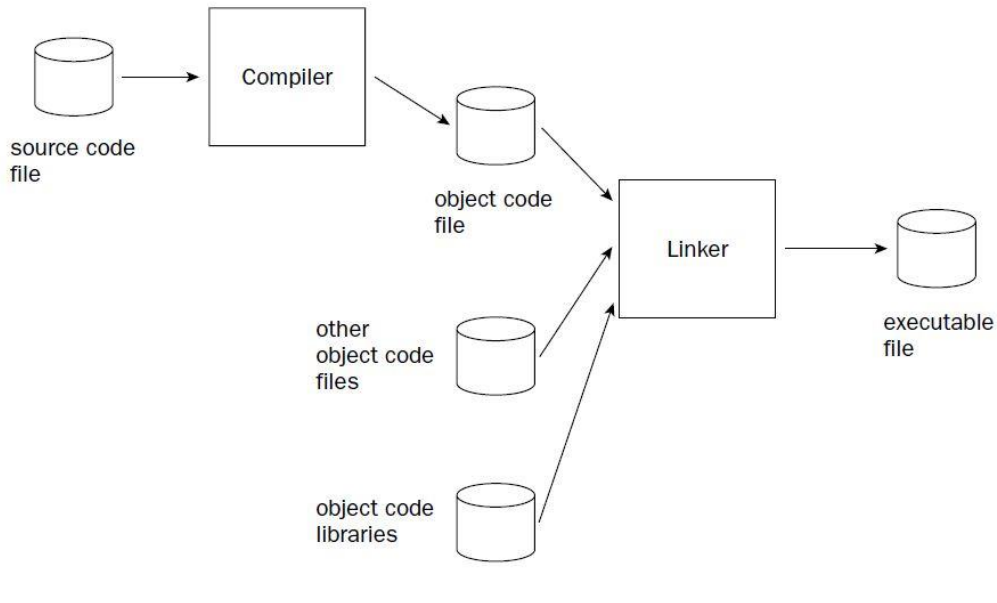


Figure 4. Generating an executable by Toolchain [8, p. 7]

Toolchain contains a compiler, linker and C libraries, (available Toolchain for Linux are mostly based on GNU which is notably written in C. There are different types of Toolchain that also include Assembly and C++ libraries). GNU is a Unix-like OS that provides components for Toolchain's using Linux. GNU is an acronym of "GNU's Not Unix" which has been developed and named by Richard Stallman, insisting that it is not a Unix system but Unix-like OS. He stated that after playing with words and being inspired from "The GNU" song has named his project GNU in 1983. This OS's compilers and linkers support Linux using Toolchain's and provide C libraries to start each embedded application. Every GNU Toolchain consists of three main components: [8]

- 1- Binutils: s set of binary utilities containing assembler and linker.
- 2- GNU Compiler Collection (GCC): compilers for C and other languages depends on the version of GCC.
- 3- C Library: there are different C libraries which are provided as Application Program Interface (API). This API is the main component for interfacing Kernel with applications.

It will be demonstrated how to use Toolchain to start booting the target device which depends on the CPU (Central Processing Unit) and in this case is ARM Cortex-9.¹ Choosing proper C libraries for the project will be also experienced. Altera's² open source database to download and set the Ttoolchain which supports Linux Ubuntu 16.04³ (the Linux version on the host machine) will be used. It is necessary to say that Ubuntu is an open source Linux distribution based on Debain which is a Unix-like OS. To sum up with the Toolchain matter, it is needed to download and configure it following the steps which will be provided in the relevant chapter.

The second step of the embedded application design flow is Bootloader which boots the embedded device and initialize it to get the Kernel. In fact, Bootloader prepares the target device to get the Kernel and ready to run it. Figure 5 describes Bootloader initializing flow:

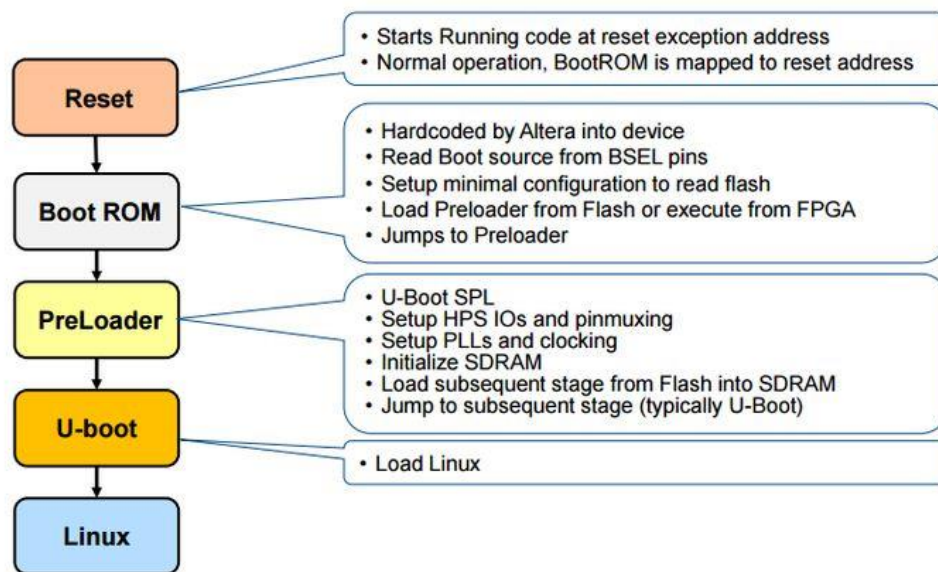


Figure 5. Bootloader Initializing Flow [6, p. 95]

¹ ARM, previously Advanced RISC Machines, and originally Acorn RISC Machines, is a family of computer processors which follows Reduced Instruction Set Computing (RISC) architectures.

² An American manufacturer of SoC FPGA's and other programmable processors. Altera produces Stratix, Aria and Cyclone V microprocessors series. It is now part of Intel.

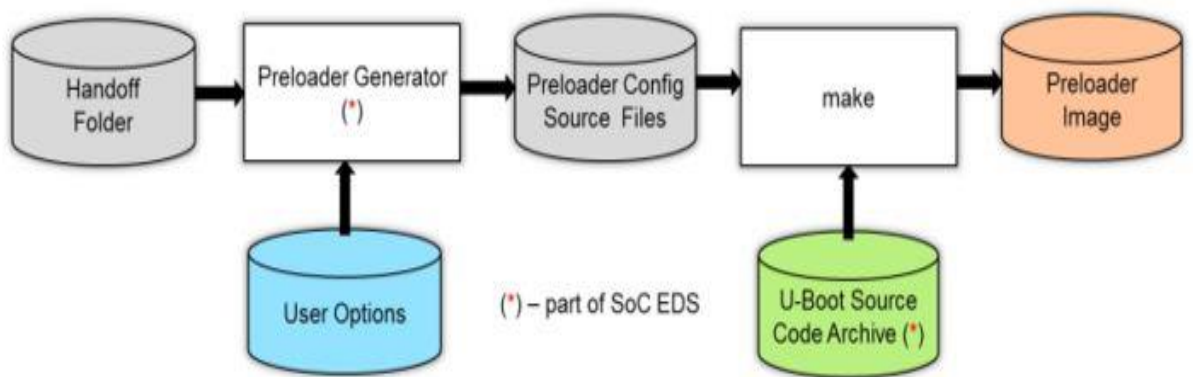
³ The Linux version can be seen by executing "cat/etc/os-release" in a terminal window for a Linux running on the host machine.

Bootloader prepares the CPU of the target device for the Kernel initialization. It briefly can be described as instruction code that is recognized by the target device's processor and it absolutely differs from different family of processors [8]. In this case, both SoC and FPGA sides are using ARM Cortex-9 processor. After some adjustment and cooperating with the CPU, Bootloader loads the Kernel into RAM and runs it. The Kernel starts to initialize hardware devices and its sub-systems. Here are some definitions of the concepts from the figure:

Boot ROM: is a very small piece of the ROM inside the CPU of the embedded device. It contains a very basic instruction code which is mentioned above and is executed when the device is powered on or reset.

BSEL pins: will be described further when the board will be explored; for now, it can be briefly said that they are configured to make embedded device CPU operates in the highest possible speed without modifying any software code.

PreLoader or SPL: Secondary Program Loader or PreLoader is a piece of software which is called from Boot ROM with the only purpose of preparing the system for actual BootLoader (U-Boot) [9]. Figure below shows the sequences for generating PreLoader image which is necessary to boot the device.



1

Figure 6. Generating PreLoader Sequences [10]

U-Boot: is a Universal Boot Loader and used to boot the Linux Kernel in ARM processor using devices [11].

The third step of the mentioned embedded device application design sequences is Kernel itself, which is responsible to manage all resources and interfacing with hardware components. Figure 7 can describe Kernel and its functionalities:

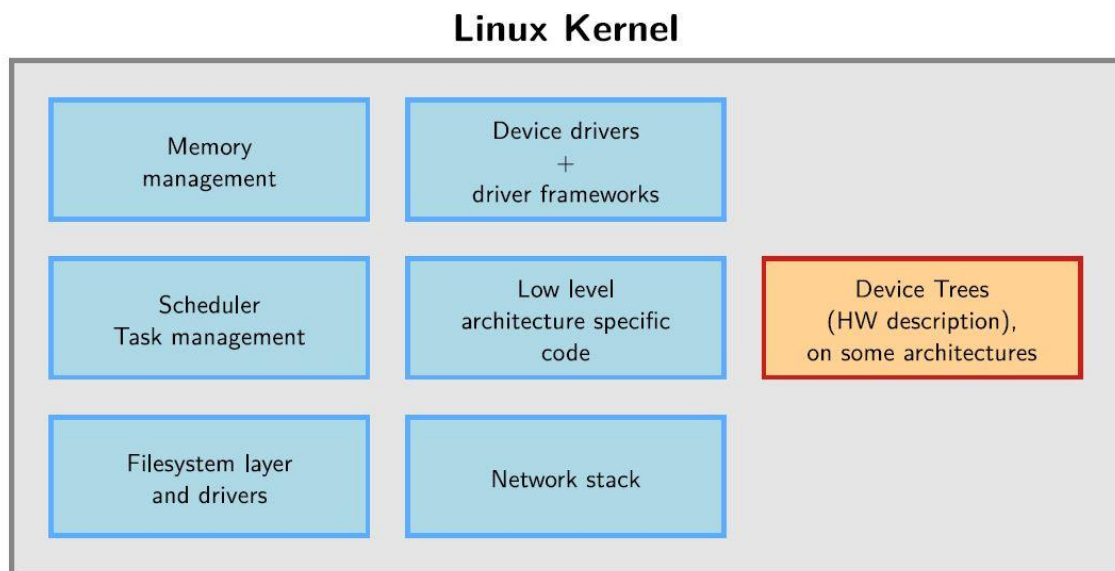


Figure 7. Kernel Functionalities [4, p. 24]

¹ Parts are indicated by red dot are generating by SoC EDS which is software platform for SoC board manipulations and configurations.

After booting by BootLoader, Kernel starts to handle all hardware resources such as CPU, memory and I/O and provides a set of hardware independent API architectures to allow user space applications and libraries to use the hardware components [12]. Kernel design will be more discussed further.

The fourth element of the design is Root Filesystem which includes all necessary files for initializing the system. Depending on the application, it can consist of

- 1- An initializing program, which is the first application running after Kernel booting.
- 2- A /dev directory to keep data, which is generated or required in user space.
- 3- Loadable modules, which are needed to be loaded during Kernel configuration [9].

All these steps will be described in the third chapter of the third part of this article when Linux Kernel distribution development is reviewed. Components of Root filesystem such as Busy Box, libraries and other utilities will be explained more.

The last element of an embedded application design are applications defined by developers for different projects with different purposes. This article is focusing on driver development for SoC Kit.

In this chapter Linux features and reasons that make it interesting for developers and specially embedded systems developers were discussed. Then, some basic information about embedded Linux design flow was provided, to have a general overview of the sequences of embedded design and other issues which will be concentrated during the rest of this article.

3 SoC FPGA's

The previous part was allocated to describe why developers need Linux customization for their various projects with different purposes. In the first chapter of this part, the hardware of the design which is SoC Kit will be discussed. A quick overview is given on processors' timelines, their evolution and progress to achieve nowadays' maturity and how development boards have been transformed to handle real time complex projects.

Afterwards, the second chapter will continue to discover the board for this project which is SoC Kit (has been shown in the figure 8), manufactured by Terasic, its main features, functionalities and hardware/software components requirements. By the end of this part, an exact overview is provided on hardware constituent of the project. Besides, it is tried to link embedded Linux explanation chapters with hardware clarification chapters in the last part of this article.

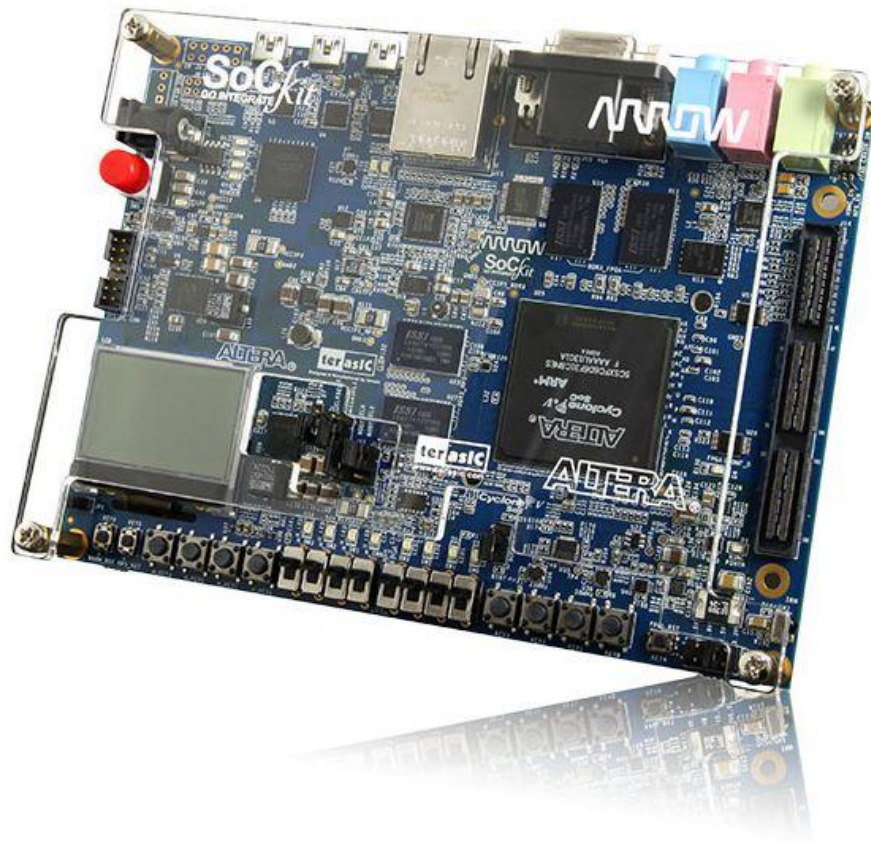


Figure 8. SoC Kit [13, p. 1]

3.1 SoC FPGA's Evaluation

In this chapter the focus is on the development boards' evolution and how they have been transformed to the nowadays' flexible and reliable boards. In order to have an exact idea about this transformation timeline, reasons and phases, first, it is required to get an

overview about developers and market expectations from these boards. Developers, students or any other interested groups of people to the embedded systems field, supposed to know that the market expectations determine hardware, software and design timelines. In the other word, companies, organizations, factories or even individuals, are looking for greater solutions day by day. If developers' solutions would not meet their anticipations, there will be no other options for this kind of solutions, either to be terminated or be enhanced in order to suit market qualifications.

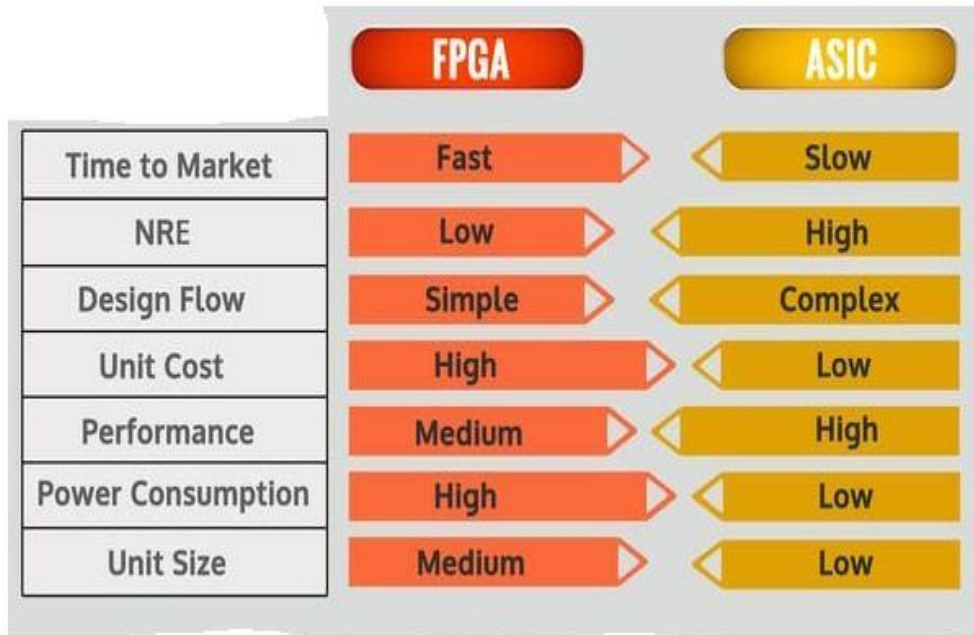
Therefore, market conditions and expectations from this industry are discussed. Certainly, the most important elements in the market are low cost, low power and high-performance expectation which are every body's desired parameters to evaluate any already existing or to be existed solution. From aerospace and defence industry to scientific research and medical industry and the other fields such as automotive, motor control, communication, image processing, high performance computing, data servers and security, which are some of main industry fields interested in FPGA, everybody is searching for the mentioned properties of FPGA design. Moreover, Internet of Things (IoT) which seems today's inevitable concept of design and is going to become most popular or even phenomenon of the forth industry revolution, is interested in FPGA design and applications. In brief, market expectation from FPGA industry can be listed below:

- 1- Low cost, low power, which is ideal for all industrial solutions and always take the first place of qualifications' list.
- 2- High performance, high content, a greater performance can be achieved by higher amount of hardware components and software elements. It is supposed to reach a level of very good balance among high content/performance and low cost/ power to idealize the final solution.
- 3- Well facility design and high integration property including peripherals and interfacing with memory, in order to handle complexity of high range computing.
- 4- Fast time to market, which follows previous conditions and if all would be close to what had been imagined, FPGA design flow would progress by desired schedule.

- 5- Connectivity, if application fields of FPGA are considered, a very high and reliable connectivity will be required and as it has been specified before, Linux behaves well with connection protocols and has high networking features. Therefore, a FPGA running Linux is needed to have high performance networking.

Meeting all these expectations, the development boards have been evolved since 1984 and have been transformed to the nowadays' SoC FPGA boards. A closer look at this evolution and its different phases, which is the best way to understand why SoC FPGA is used for the design, is beneficial. The FPGA industry began to develop since 1984 by integrating Programmable Read Only Memory (PROM) and Programmable Logic Device (PLD) which has been provided to the market by Altera. In 1985 Xilinx, one of nowadays' FPGA manufacturer, delivered the first FPGA to the market with limited functionalities. From the early ages, FPGA has been evolved in varied forms [1].

In a parallel way to FPGA timeline, there were other boards which have been used for almost similar purposes. Application Specific Integrated Circuits (ASIC's) are the most popular ones since their properties meet market qualifications. Figure 9 is demonstrating advantages/disadvantages of FPGA and ASIC to compare their reliability for complex projects:



1

Figure 9. Advantages/disadvantages of FPGA and ASIC [14]

Looking at this figure may create a little bit of confusion as both FPGA and ASIC have some advantages and some unwanted features which make developers to find a balance considering these features. At first, a project which requires hardware platform choosing should be considered. Moreover, depending on the budget and size of the project, time schedule and essential functionalities, a better view of the existing options could be achieved. If FPGA is chosen, for a high amount production, which has been scheduled properly for research and development period (there is no rush for time to market), then more financial resources is needed, as FPGA unit cost is higher compared to ASIC. FPGA might be selected because of its design flow and contents which are simple and well-integrated for this purpose. As it is clearly understood, FPGA has drawbacks in case they would be chosen for their advantages such as low NRE, simple design flow and fast time to market. So how these disadvantages could be compensated, or in better words, if these features would be enhanced, then FPGA might be the first option for embedded design.

¹ NRE: Non-Recurring Engineering, one-time expenditure for research, develop, design and test a new product. NRE unlike the production costs, which must be paid constantly, is paid once as it is being considered from fixed costs category [1].

Figure 10 provides a clear view on the options by analysing another type of ASIC's, System on Chip (SoC):

FPGA	ASIC/ASSP - SOC/non-SOC
Faster Time to Market - No layout, masks and manufacturing steps needed	Need longer design times to take care of all manufacturing steps
Field reprogrammability - Design changes can be absorbed even in field and FPGA reprogrammed	Once manufactured, need to spin again a new chip in case of bugs
More power consumption and may not be high performance because of programmable design and low clock speeds	Custom design for an application helps in designing for power/performance efficiencies
Good for prototyping and low volume designs as cost would be less	For larger volume of production, cost per unit will be much less for an ASIC
Generally not possible to have analog/mixed signal designs and limited to what vendor supports	Can support analog and mixed signal designs

1

Figure 10. FPGA, ASIC/ASSP and SoC comparison [15, p. 7]

SoC is more integrated with more components on ASIC's or ASSP's with single or more processor cores. (ASIC's and ASSP's can be SoC or non-SoC depending on if there is a processor mounted or not). So, what if desired features of both sides would be combined into a compact board to utilize all possible advantages. By this way a very good solution for the projects is found. Before going to deeper details, it is better first to have a specific definition of SoC FPGA by Altera (which has been merged into Intel since 2015):

“SoC FPGA's integrate both processor and FPGA architecture into a single device. Consequently, they provide higher integration, lower power consumption, smaller board size, and higher bandwidth communication between processor and FPGA. They also

¹ ASSP: Application Specific Standard Parts.

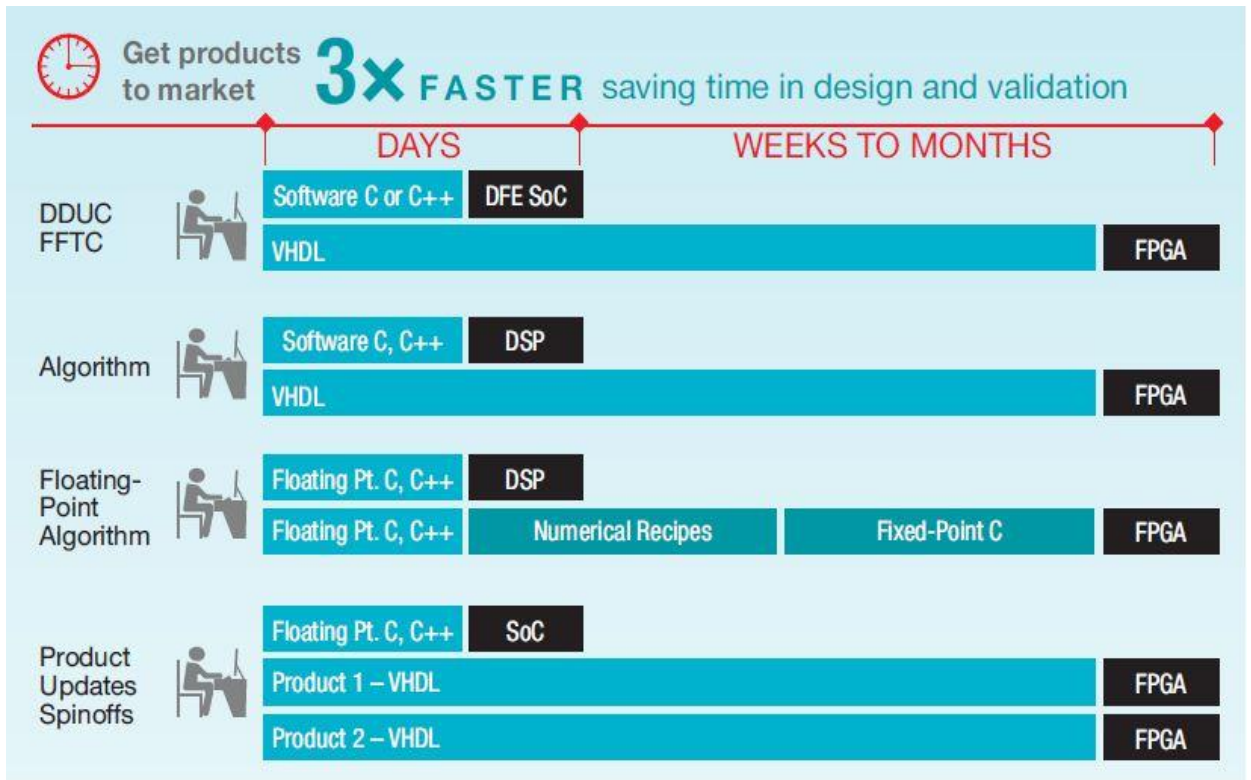
include a rich set of peripherals, an FPGA-style logic array and high-speed transceivers” [16].



Figure 11. SoC FPGA compare with ASIC [16, p. 3]

Figure 11 shows how SoC FPGA has been transformed from similar type boards. There are three commercially available SoC FPGA's which all are using ARM processors. Altera as one of the main competitors has Aria and Cyclone series from FPGA side (I am using Cyclone V integrated with ARM Cortex-9 dual-core processor).

Finally, a comprehensive overview about what were discussed is provided by figure 12; it is clearly demonstrated that if SoC FPGA's are used, products can get to the market more quickly [17].



12

Figure 12. Fast Time to Market [17, p. 4]

Floating point numbers and algorithms are an important data type in computation and represent real numbers with a fractional part. According to IEEE, “754 floating point standard” is the most common one used in the modern microprocessors, however, in 2008 it has been updated [18].

A clarification on exiting options, how and why to choose a more reliable hardware platform, SoC FPGA, for the design, have been provided.

¹ DDUC: Digital Down Converter-Up Converter; converts a band limited signal to a lower frequency with lower sampling rate to simplify the subsequent radio stage [1].

² FFTC: Fast Fourier Transform Coprocessors; an accelerator module that can be used for performing FFT and IFFT with higher floating-point rate.

3.2 SoC Kit

In this chapter the board SoC Kit is discovered. It “presents a robust hardware design platform built around Altera FPGA combines with Dual-Core ARM Cortex-9 processor that provides re-configurability paired with high-performance and low-power consumption” [13, p. 5]. Figure 13 contains description for some of critical components of the board and figure 14 is demonstrating a clear view of the board’s block diagram:

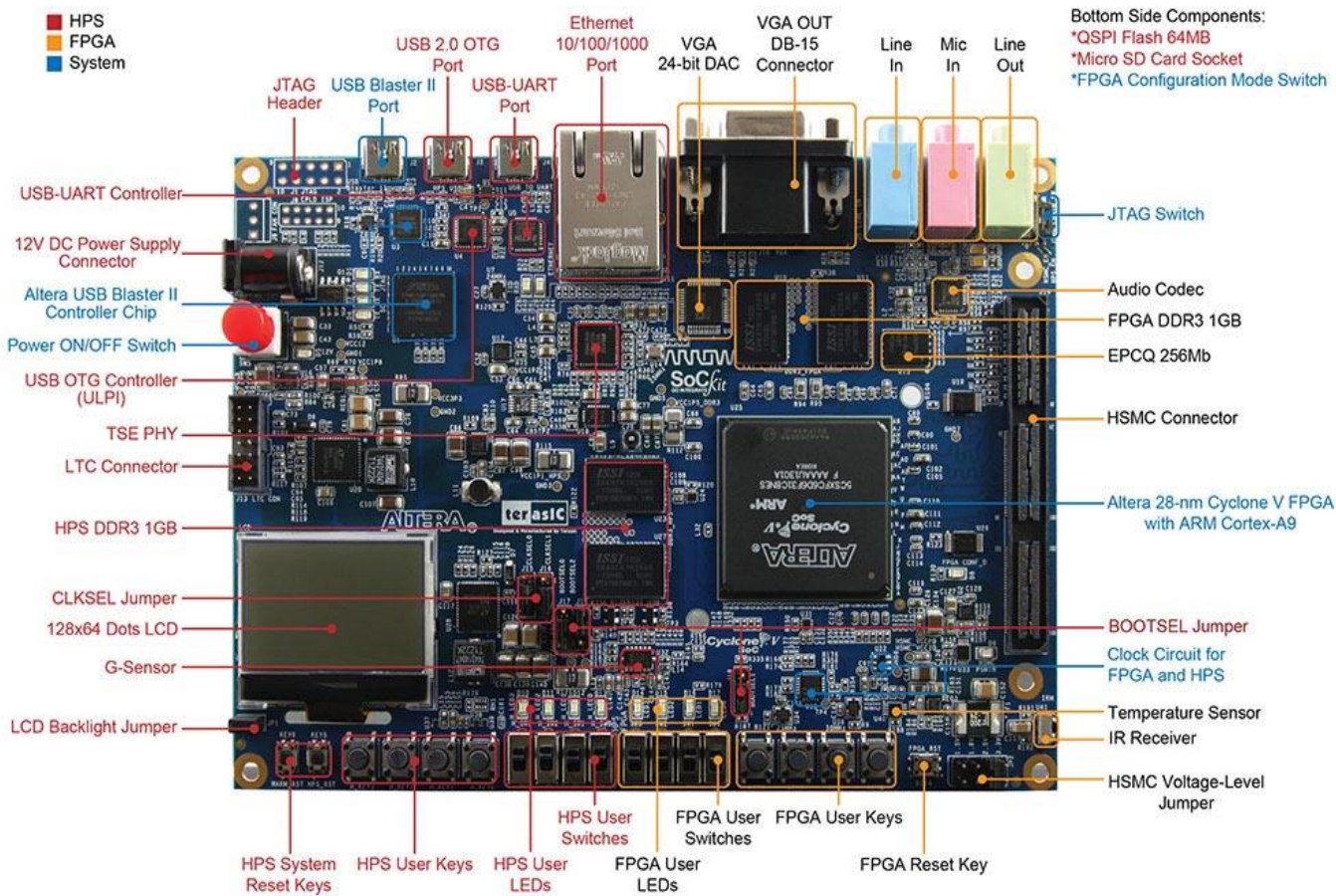


Figure 13. SoC Kit Components [13, p. 7]

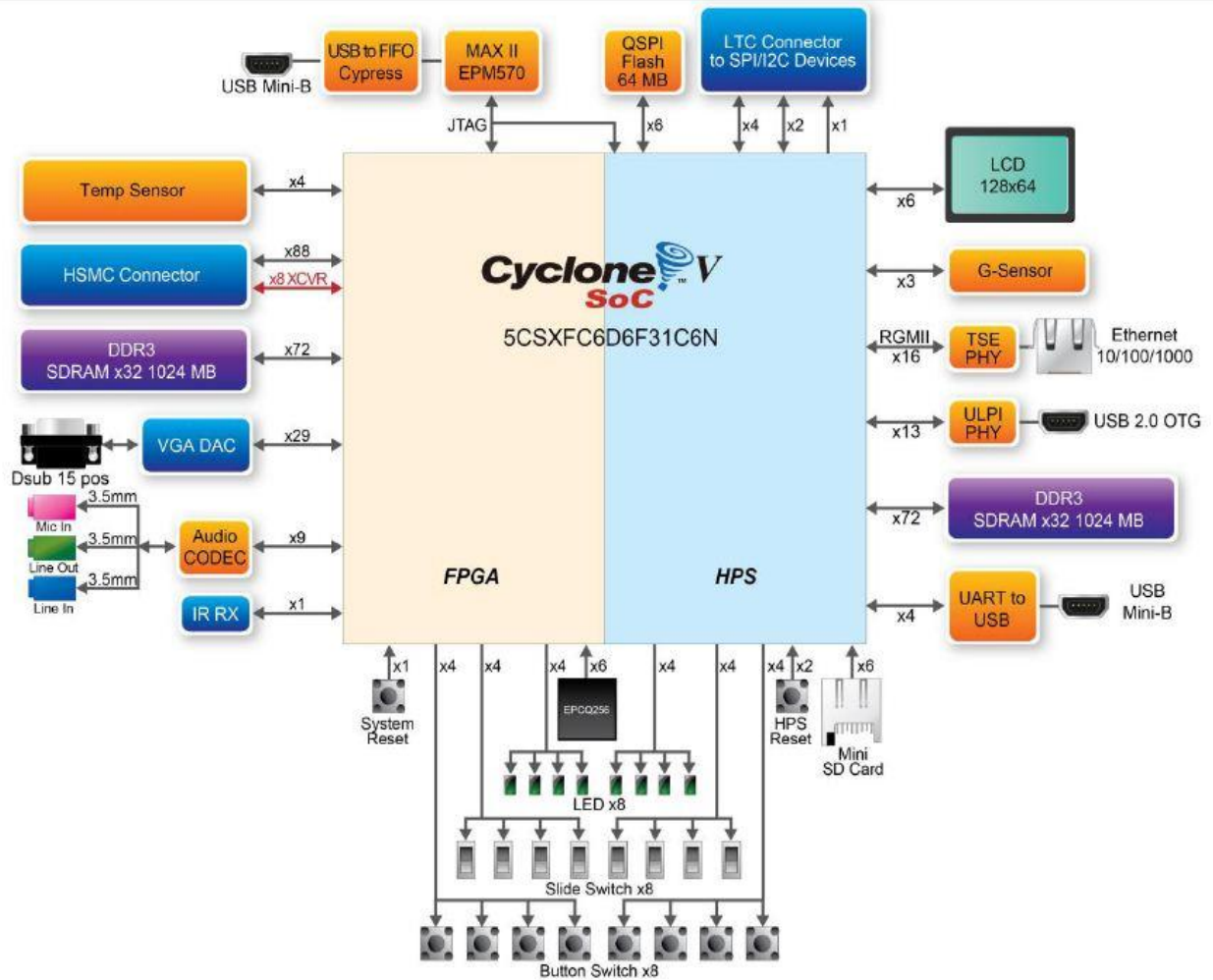


Figure 14. SoC Kit Block Diagram [13, p. 11]

As it has been clearly indicated in figure 14, the device brain consists of FPGA and Hard Processor System (HPS); Cyclone V is the general name of these series with FPGA Fabric and Dual-Core ARM Cortex-9 processor. The focus will be more on the board's functionalities to support the road map to design a driver for it; data sheet of the board can be studied for more technical and detailed information. The driver will manipulate four LEDs on FPGA side by giving the Exclusive OR (XOR) operation result to power on matched LEDs.

In order to have access to FPGA side LEDs, either FPGA direct design or HPS user space application, which give an ability to configure FPGA side LEDs via one of existing bridges, can be tried. There is another way which is in fact more reliable for real life projects and it is driver development which operates in the Kernel space. Now it is the

time for real work and having an experiment of how both sides of the board communicate with each other; Figure 15 is specifying three bridges and their connections with FPGA and HPS sides:

1

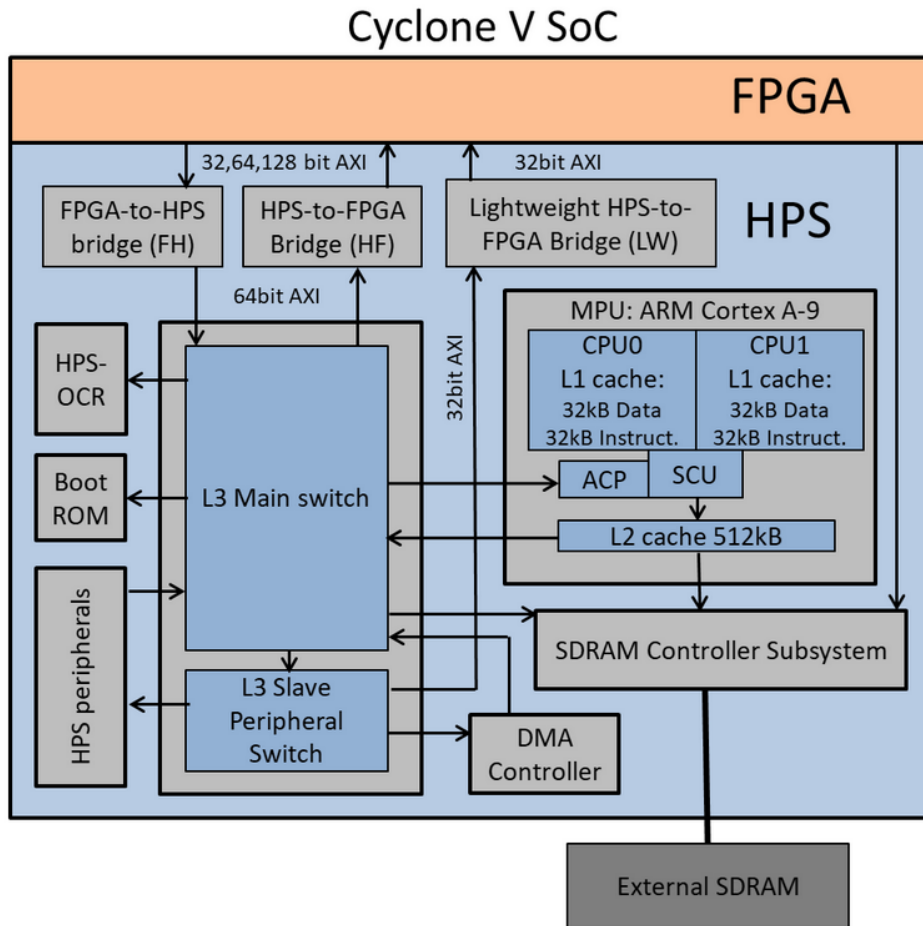


Figure 15. Cyclone V SoC Bridges [19]

HPS-FPGA bridges allow masters in FPGA fabric to communicate with slaves in HPS logic and vice versa. For instance, if a peripheral be implemented in FPGA side, HPS component such as Microprocessor Unit (MPU) can access it. In the same way, components implemented in FPGA fabric such as the driver, can access peripheral in HPS side. Each bridge consists of a master/slave pair with two interfaces which are exposed one to the FPGA and the other to HPS sides.

The FPGA to HPS bridge is supported by an Advanced eXtensible Interface (AXI) slave that can be connected to AXI master or Avalon-Memory Mapped interface in the FPGA

side. HPS to FPGA and Lightweight HPS to FPGA bridges expose an AXI master interface that can be connected to Avalon-MM slave interface in FPGA side [19]. Figure 16 has a detailed explanation of each master and slave interface with their data widths:

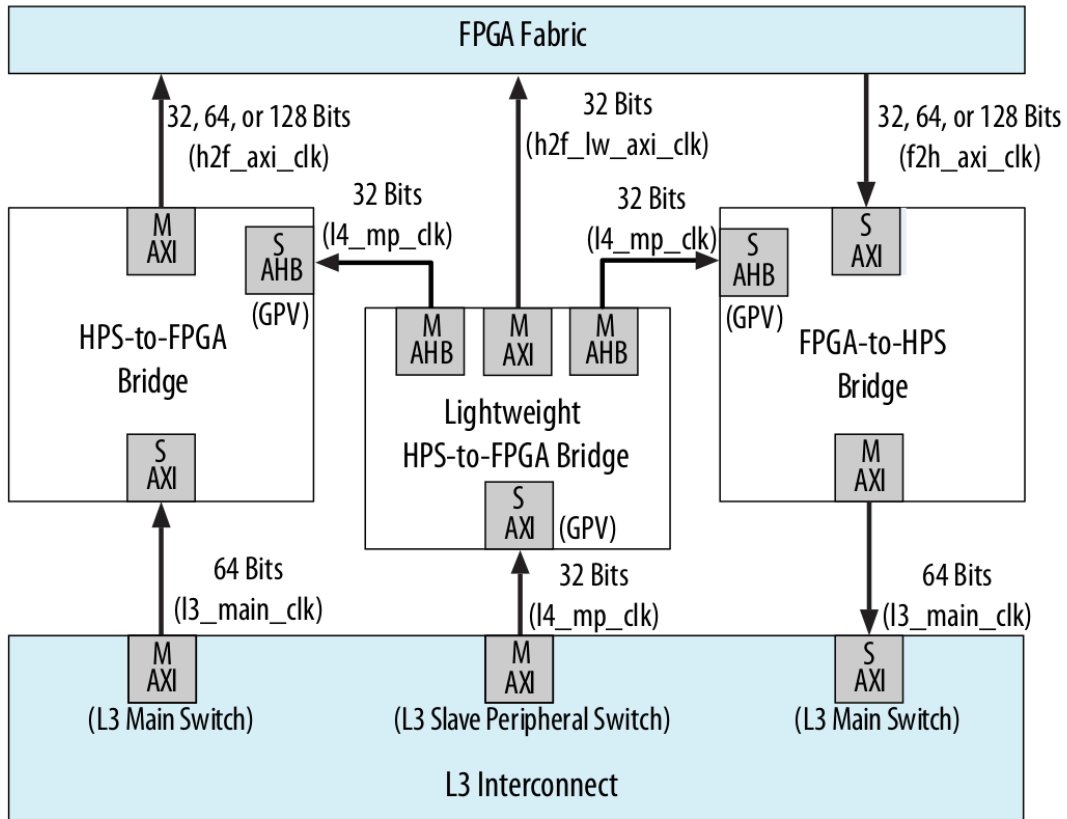


Figure 16. Master and Slave Interfaces [19]

As it is indicated in the figure 16, there are two bridges from HPS to FPGA which the Lightweight one provides lower performance interface with only 32 Bits band width. While, the other HPS to FPGA bridge as well as FPGA to HPS bridges perform with three options of band widths. The Lightweight HPS to FPGA bridge has limited access with 2 MB address spaces. The bridge is connected to control and status registers of soft peripherals in FPGA fabric.

It is needed to have a general idea about AXI and Avalon-MM interfaces because they will be used during FPGA and user space application design. As it has been mentioned previously, AXI is a set of specifications and a part of ARM Microcontroller Bus Architecture (AMBA) protocol. This protocol states that how different modules on the

system can communicate with each other, using a Handshake-Like¹ producer before all transmissions [11]. Figure 17 can describe more by illustrating master/slave interfaces:

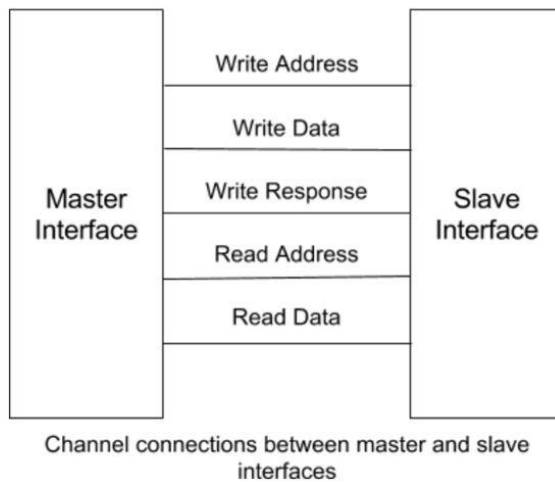


Figure 17. AXI Master/Slave Interface [11]

XOR project uses Write/Read data signals (shown in the above figure) for the FPGA design.

“Avalon interfaces simplify system design by allowing developer to easily connect components in Intel FPGA. These specifications define interface appropriate for high speed data streaming, reading/writing registers and the memory, and enable developers to incorporate custom components in Avalon interface to enhance interoperability of the design” [20]. I will use this interface in the custom component for FPGA’s side LEDs control. Avalon-MM interfaces can be used to implement read/write interface for master/slave components such as memories, UART²’s and Timers; and typically include only the signals required for the component logic.

To sum up, this part demonstrated hardware platform choosing, while getting an idea about the FPGA and more importantly SoC FPGA features. A detailed look at the SoC Kit and its bridges were provided, which will be used for the FPGA’s side design.

¹ Handshake is a process of communication that establishes all required protocols and links before the full communication begins [1].

² Universal Asynchronous Receiver Transmitter.

4 Driver Development for Embedded Linux

This part describes how the driver is built. The last two sections tried to give an overview on Linux for embedded, its design sequence, SoC FPGA evolution and the SoC Kit features. Obtaining a general and sometime a detailed idea about the background of the practical task was essential. This part of the article demonstrates that without having at least a general knowledge about embedded Linux design flow and the board specification, it is not possible to have any kind of manipulation on the design. A driver to run FPGA's side LEDs will be generated by the end of this part, and during design steps, the necessity of the background knowledge to manage sequence and handle probable problems is understood. This part consists of five chapters including software requirements for both FPGA and HPS sides, FPGA design explanation, obtaining a Linux distribution, user space application (HPS work) and finally XOR driver development.

4.1 Software Requirement for Driver Development

It has been mentioned before that SoC Kit's brain consists of FPGA and HPS and their relevant components are integrated in the board cooperating by high data speed interfaces. Although SoC Kit is a combination of efficiently integrated of FPGA and HPS components, these components can be categorized in two different categories: FPGA and HPS. What I am trying to insistently specify here is compacting two independent devices into a single board, while it can be considered and configured either as two different devices or as a single compact device. Thus, I will work with FPGA and have a component design for it and work on HPS separately and interestingly work on them together at the same time. This is really the handful property of the SoC FPGA's which is very useful for developers to have flexibility, efficiency of their design and fast time to market products.

At first comes discussion about FPGA side design which will be explained in the next chapter. Quartus as the platform, which enables developers to have FPGA design from scratch or to modify already existing designs, is needed. Quartus Prime 18.1 Lite Edition

can be free of charge obtained from Intel web page from software for FPGA section. Installation of Quartus is so simple and double clicking on the run file would be enough and as it has been illustrated in figure 18 and 19, it has Platform Design Tool (formerly known as Quartus System: Qsys) to create FPGA design and manipulate/modify hardware components.

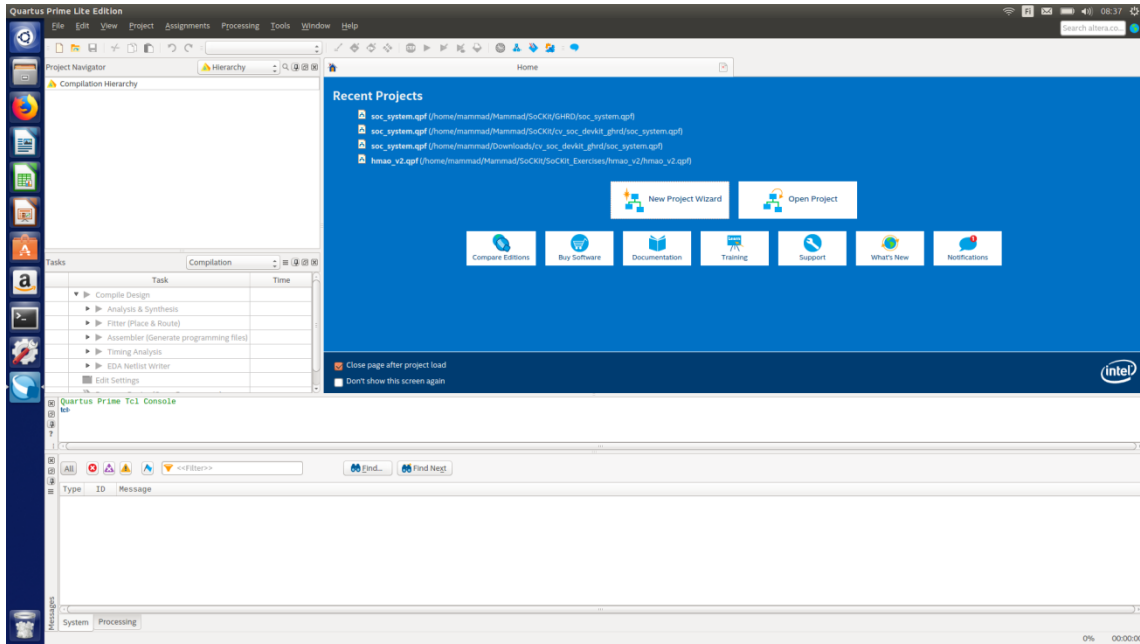


Figure 18. Quartus Prime 18.1 Lite Edition

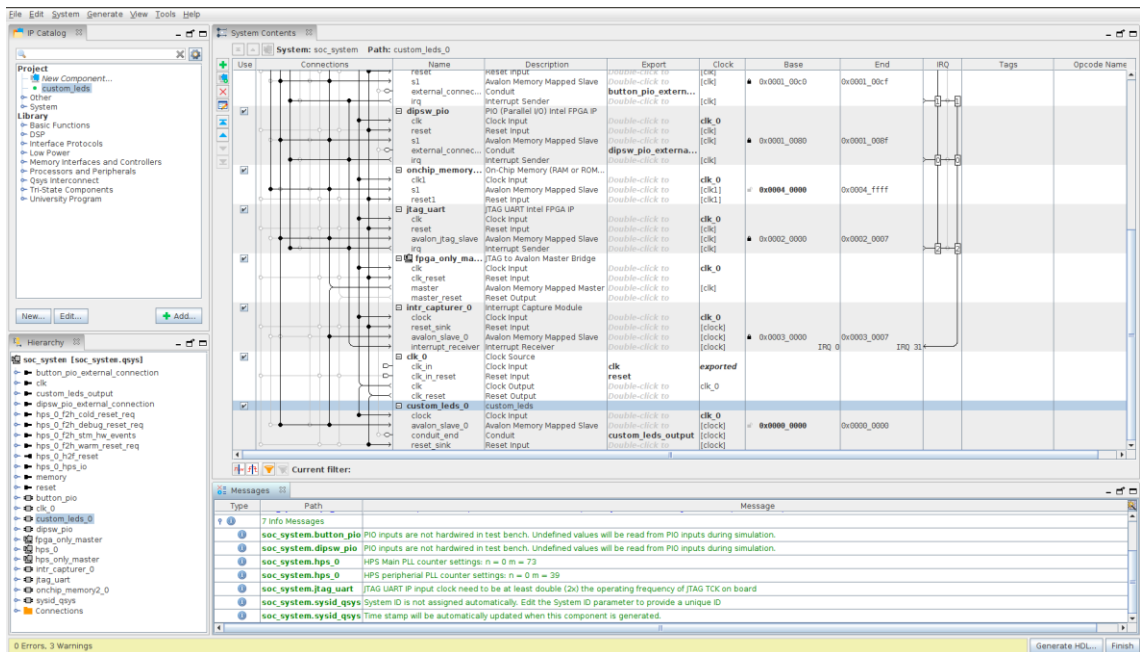


Figure 19. Platform Designer Tool or Qsys

The next chapter describes the FPGA design flow, which has been developed and verified by the author, using Qsys tool to generate Qsys file, and then using Quartus file converter to convert this Qsys file to Raw Binary File (.rbf), which FPGA needs to be configured and programmed by Linux. FPGA can be configured in two ways; using Quartus by adding the board from tools tab in menu bar and after recognizing the board by Quartus, adding or changing .sof¹ file which is generated at the end of Qsys operation, and finally running it. The result can be observed on FPGA LEDs. The second method is to copy .rbf file to a SD Card and use Linux to run this .rbf file and program the FPGA. Quartus creates a folder named “handoff folder”, describing hardware configurations, which is required to have device tree generating as an important step of the embedded design; because it includes hardware components’ configuration to introduce to the Kernel.

The HPS side of the board needs to be programmed using SoC EDS DS-5 platform which is Intel specific ARM product. It is named simply DS-5 as the original name would be confusing if it is considered the user interface of the platform which has a different name of Eclipse. This software platform also can be obtained from Intel website but its installation is slightly different for Linux running host machine compare to Quartus setting up. DS-5 installation can be explained as below:

- 1- Running DS-5 from a terminal window. (There may be a need for changing the mode using `sudo chmod 777 “file name”.run`).²
- 2- Not to install DS-5 yet, instead, starting Embedded Command Shell from the file directory which was obtained when DS-5 package has been downloaded.
(This can be done by executing `“./embedded_command_shell”` in a terminal window from correct directory).
- 3- Running DS-5 installation script from Embedded Command Shell and assigning correct installation directory.
- 4- After installation, there may be a need for adding PATH or editing that from .bashrc file which is hidden in the home directory. After adding the installation

¹ SRAM Object File which is machine generated code and be created by the end of Qsys operation and can be obtain with Quartus Programmer tool to run the FPGA. SRAM is static RAM which uses flip-flop method to store each bit.

² Changing mode in Linux is required because of hierarchic filing structure. “sudo” or super user enables developer to execute commands with privilege and 777 enables developer to execute the desired file as top-level hierarchic file.

- path to the PATH by editing .bashrc file, it is required to run following command line in a terminal window: “source ~/.bashrc” and by this installation is done.
- 5- To open Eclipse (user interface of DS-5), in Embedded Command Shell, it is required to go to Eclipse directory and execute the following: “sudo bash ./eclipse”.¹

Figures 20 and 21 are showing DS-5 and its Debugger tool which are similar to ARM Cortex software platform that is used in TUT Embedded courses:

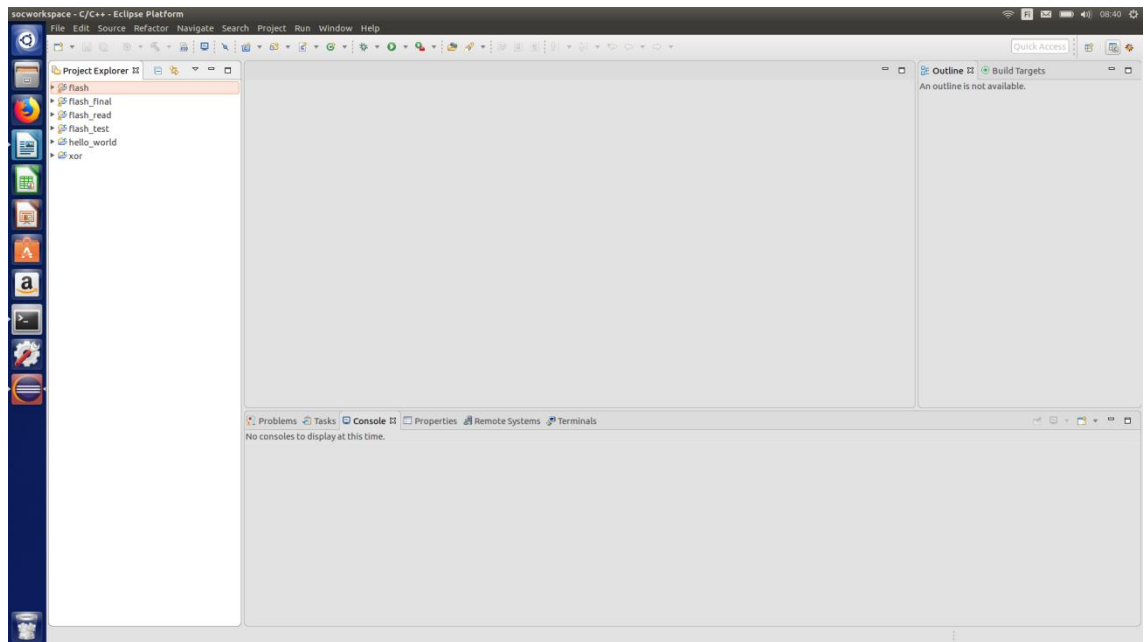


Figure 20. DS-5

¹ An appendix of useful Linux commands and their explanations will be provided.

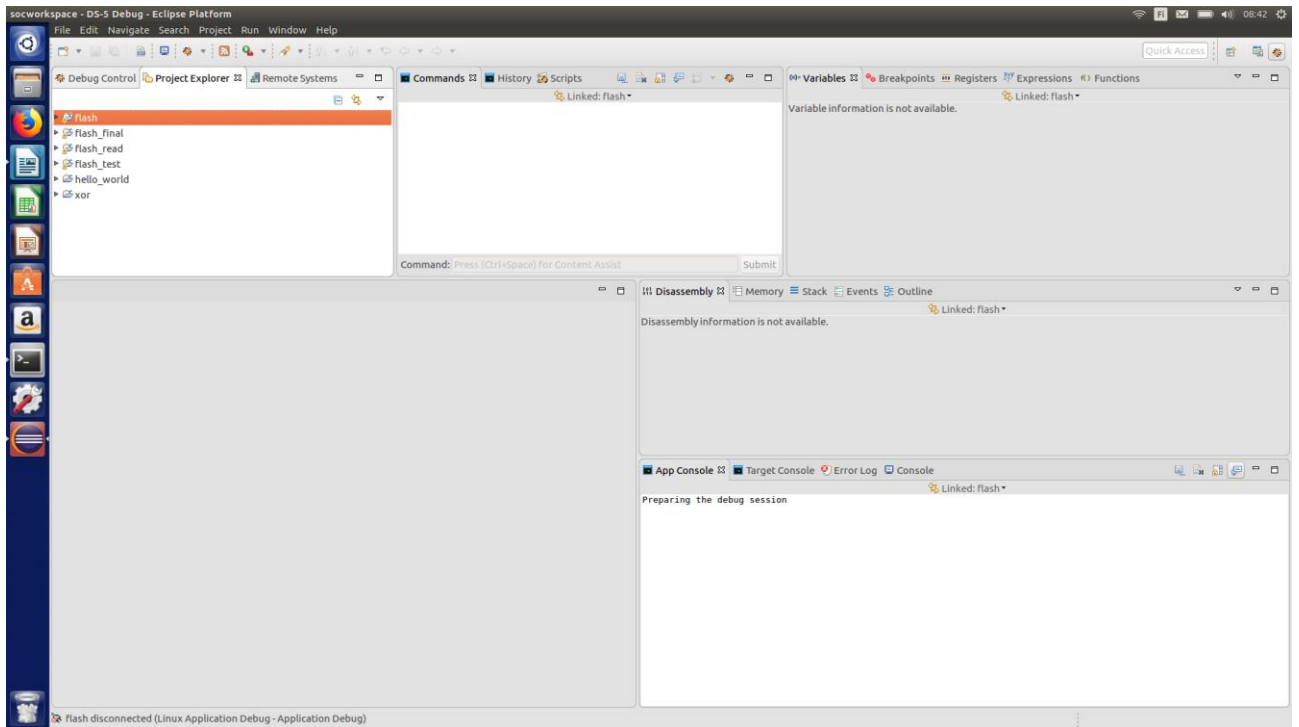


Figure 21. DS-5 Debugger Tool

4.2 FPGA Design Flow

After setting and running up the required software, now it is time to practice and execute the real task. The first and very important stage of the driver development journey is the Qsys design, because the driver, as software, will run this hardware component. It is important to remind that the author is trying to manipulate FPGA side LEDs by a driver which uses both FPGA and HPS utilities. Thus, it is required to analyse HPS facilities, bridge interfaces and FPGA configuration in order to avoid possible problems.

As it has been mentioned previously, there are two different designs: one is Qsys and the other is C programming in DS-5; which both require bridge interface manipulations as bridges are connected to FPGA fabric and HPS. Here is a clear view of the road map which consists of Qsys design with bridge interfaces and then C programming with essential bridge interface in HPs side. To begin the Qsys design, it is needed to start Quartus and from menu bar tab on Platform Designer tool to open it and then choosing the Golden Hardware Reference Design (GHRD) file from right directory of the host machine. GHRD file consists of all default hardware definitions and configurations which comes with the board user guide CD and differs from a board to another board.

Developers can generate all required files inside this folder by starting Qsys design from scratch which needs to have an absolute detailed knowledge of the board hardware components' specifications. In fact, Qsys design requires pin definitions, (in this project, SoC Kit has more than 220 different pins, which should be mapped and defined one by one). Moreover, facilities like display should be defined separately which is really hard work and needs sometimes weeks of analysing data sheets and VHDL programming. As this project has the GHRD folder ready, it can be started by uploading Quartus Project File (.qpf) and opening Qsys, then uploading already existing .qsys file.

LED_PIO (Parallel Input/Output) component, which is default design of the board's LEDs, is found. In order to have the custom component, it is required to remove this component and then add a new component instead, which needs to be designed. Here, the author added VHDL codes as two different files because one of these files consists of the IP block codes written to manipulate LEDs and the other is Avalon interface codes file. Here a piece of the IP block code which does the XOR operation, and a part of Avalon interface code, that includes required signals' names, are provided: “

```
begin

    process (clk)
    begin

        if rising_edge(clk) then
            if rst = '1' then
                temp <= (others => '0');
            elsif load = '1' then
                temp <= din(7 downto 4) xor din(3 downto 0);
            end if;
        end if;
    end process;

”

“

entity xiphera_test_block_avalon_interface is
port(
    clk          : in std_logic;
    rst          : in std_logic;
    read_ctrl    : in std_logic;
    write_ctrl   : in std_logic;
    writedata    : in std_logic_vector(7 downto 0);
    readdata     : out std_logic_vector(7 downto 0);
```



```

        led_export : out std_logic_vector(3 downto 0)
    );
end xiphera_test_block_avalon_interface;

```

Figures 22 and 23 are demonstrating the IP block general view and Avalon interface and its signals which have been mentioned in the second code, respectively:

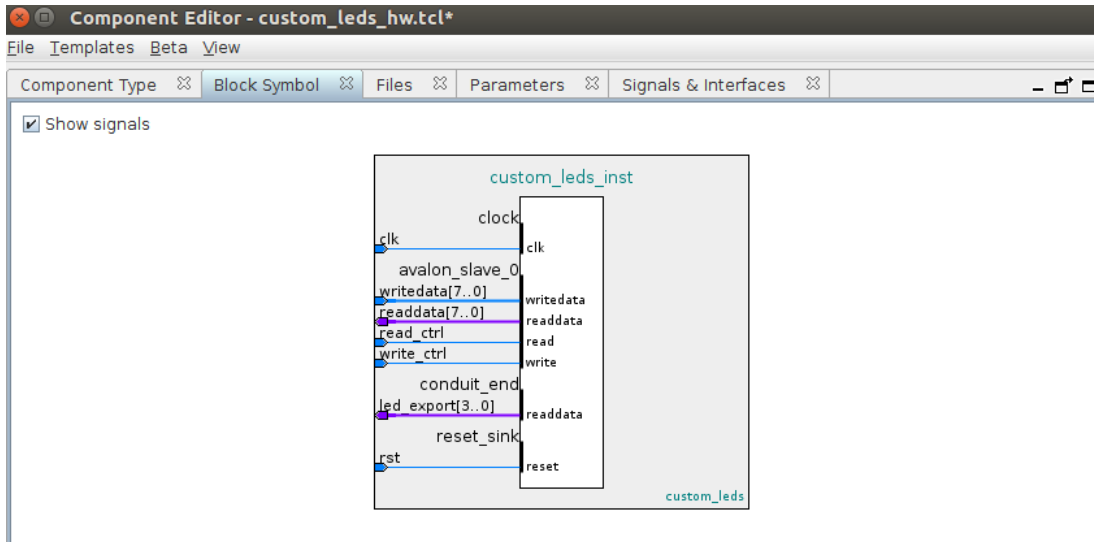


Figure 22. Customized IP

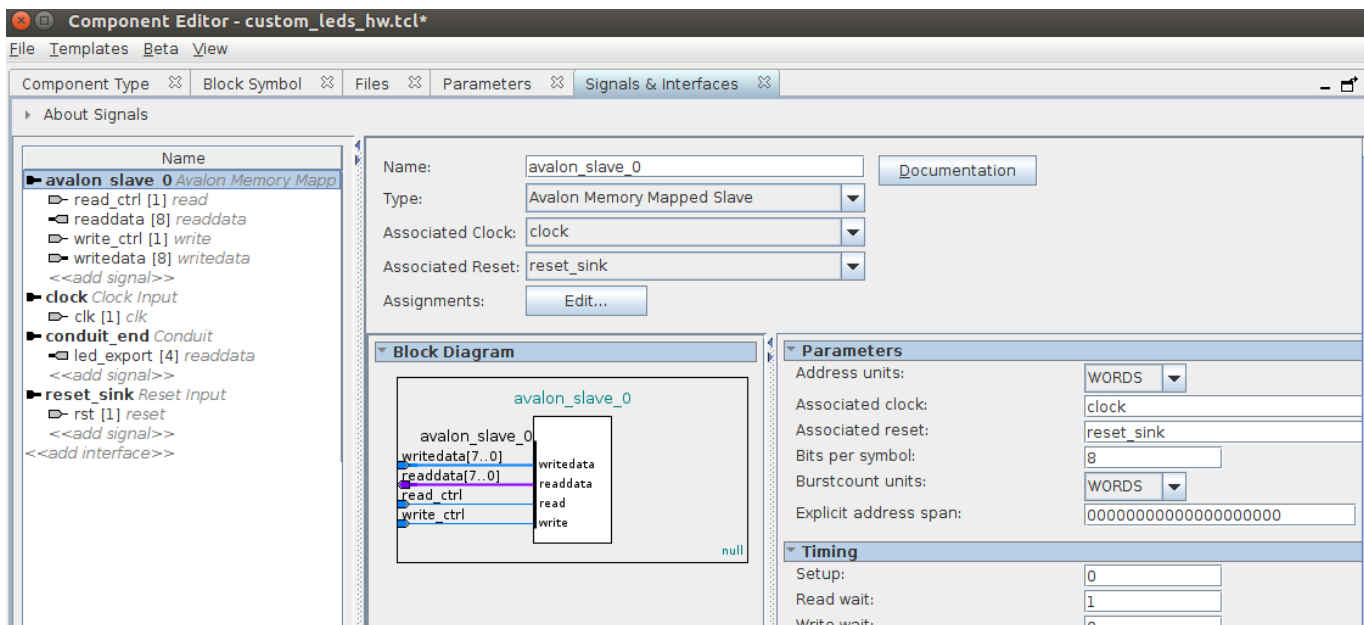


Figure 23. Avalon Interface

During Qsys design before generating VHDL code (which is an available option in Qsys tool), a very important issue, which will be used later in the device tree generating stage, needs to be considered. It is required to modify Tool Command Language (.tcl) file¹ which has been generated when the author started to Qsys design and has been used to define the customized IP to the device tree. The definition of the IP block was added in this file and it has been used to generate .dts file² and finally this .dts file has been used for .dtb (Device Tree Bulb) creating which is one of Quartus final results and is used by the device tree compiler. As it is clear from explanations, every step of the design depends on the next one, so it requires due caution and doing the process step by step.

Now VHDL code can be generated and the process continues to the Quartus design. After the same modification in VHDL code (inserting the IP block), it is time to start preparing for analyzing and synthesis which is the final stage of Quartus design to generate .sof file. Here, it is required to consider another important point that is running the .tcl script from Quartus by choosing it from right directory, before starting the full synthesis. Then full synthesis should be run and wait to have .sof file, which can be converted from Quartus converting program files tool, to .rbf file to use from Linux to configure FPGA. Finally, it is ready to try the result of the first step of design either by programming FPGA with .sof file and run LEDs using Quartus, or by running .rbf file from a terminal window and configure LEDs using Linux. This is the last operation before moving forward which gave us all required files for the rest of development. Now FPGA design could be left aside, and focus would be given on the Linux and HPS development stages which will be the next chapters.

4.3 Linux Distribution Development Steps

This chapter will describe all necessary steps to obtain and source the Kernel to manipulate its property and configure it as desired Linux distribution. As was mentioned,

¹ TCL (also pronounced as tickle) is an open source, general purpose and dynamic programming language [1].

² DTS: Digital Theater System file format which saves data in audio type.

elements of every embedded design have been listed as: Toolchain, Bootloader, Kernel, Root file system and User space application(s). It is clearly explained that obtaining the Toolchain is the first step of design, but before going further it seems necessary to clarify one issue. Toolchain is listed as the first step of every design but XOR project started with FPGA design; but why? The answer simply can be declared that FPGA design was a pre-request for the device tree which will be generated and added to the Kernel after U-Boot stage. By this way the author organized the process and would not need to come back to FPGA design in the middle of Linux development process. As it was insistently indicated before, everything else related to embedded Linux design depends on Toolchain and in this case, it is GCC which is installed when Linux distribution on the host computer was installed. Toolchain compiles object code files (which have been obtained from Altera open source data base) to its linker and then generates executable files which can be used by Linux. Here, no further action is needed, because Toolchain effects are seen when it is used to compile U-Boot and Kernel.

This part concentrates more on Bootloader step of the design. Booting sequences could be briefly reminded here. Every time that the power button be pushed, Linux booting happens by the following order: reset, boot ROM, pre-loader, U-Boot and finally Linux.¹ As it has been illustrated in the first part of the thesis, after reset, the system starts to read Boot ROM and check for some physical configuration (which must be done before any kind of design considering SoC Kit user manual guide), then tries to do a set of configurations to prepare flash. It is useful to specify the reason of storing some important information in the flash. The type of the flash is None-Volatile NOR flash which can reserve information permanently. By this way, significant information will be safe when the system powered off or reset and this substantial data can be used to repeat basic configuration of the board. After setting up the flash, Boot ROM tries to run Pre-Loader which must be generated before going on. Therefore, the second part of the practical task during this design will be pre-loader generating.

¹ For more see figure 5.

To begin, the author initially started with BSP¹ editor. Intel has Nios II BSP editor coming with the SoC EDS (Intel specific DS-5) installation package. I only started an Embedded Command Shell (the procedure has been declared in chapter 4.1) and ran the following command line “`bsp-editor &`”. This starts the editor window which asks for the file, and from handoff folder in GHRD directory the “`soc_system_hps_0`” file can be added. After some modification (which can be found from online sources), simply pressed generate and after a short while, closed the window and ran “`make`”. Note that all shell commands must be execute in Embedded Command Shell terminal window. That’s all about the pre-loader which is a pre-request element for Boot loader, then it is time to prepare for actual U-Boot.

After getting done two pre-request tasks (Qsys design for the device tree and pre-loader generating for U-Boot), now it is time to return to the original design flow with obtaining and compiling toolchain. As it has been mentioned before, toolchain comes with Linux installation, but it varies from one board to another, thus a new toolchain source for the new Linux distribution is needed. It can be downloaded from Linaro web page with a desired version.² Here, a very important matter, which is faced frequently, and is cross compile environmental issue, which must be set properly, should be notified. To do this, it is required to run the following from a terminal window “`export cross_compile Directory/arm-linux-gnueabihf-`” and then to check if it sets properly with “`printenv`”. Exporting the directory should be carefully done, because it can be source of many problems during U-Boot or Kernel compilation.

Next, the source code of U-Boot is obtainable from Altera open source data base (the desired version could be downloaded) and then U-Boot is compiled by:

```
“make socfpga_cyclone5_config
```

```
make”.
```

It will generate U-Boot.img (image) file. Another file named boot.script is required to complete U-Boot stage of the design. This file contains hardware information

¹ In embedded system a Board Support Package is a layer of software containing hardware specific drivers to allow RTOS operates in a particular hardware environment [1].

² versions 6.3.1 or 7.1.1 can be used as some other versions create problems during compilation, moreover, the hardware might not suit recently released versions.

of FPGA that U-Boot requires to load and pass to the Kernel. This file can be created and run to compile U-Boot and end the task here (its content can be found in Rocketboard [9] web page: embedded Linux beginners guide).

Now it is time to generate the device tree which previously has been mentioned several times. Before starting the process, an expert look at the device tree is required to realize what it is or why it is needed at all. Each embedded board has its own specifications that Kernel needs to know by obtaining its initialization code, which is provided by board manufacturer. Before the device tree, manufacturers had to provide maintenance service for Kernel every time that hardware configuration needed to be changed. Nowadays, the device tree takes care of hardware structure definition and is independent from U-Boot and the Kernel, which enables developers to modify only this file without concerning about other files that require more repairing time. Figure 24 shows a clear view of the device tree generating and U-Boot compiling steps:

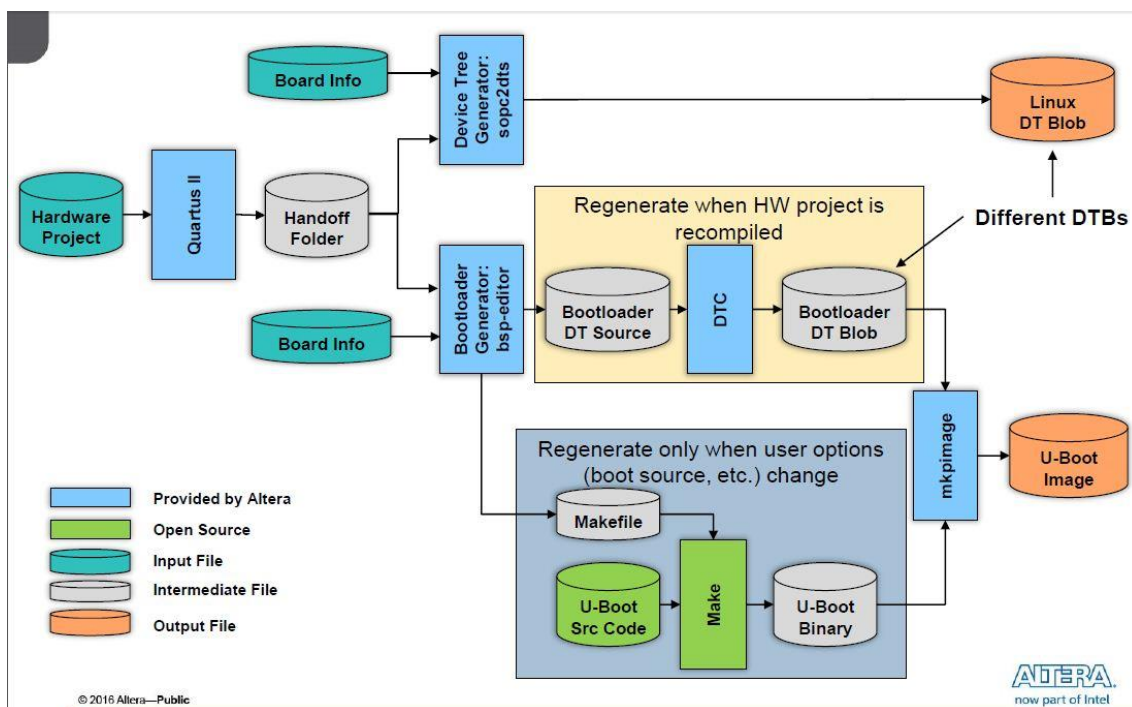


Figure 24. Device Tree Generating and U-Boot Compiling Steps [6, p. 50]

Board Info files shown on the figure 24, come with GHRD folder in .xml format and contain data for external peripherals. The FPGA should recognize the board peripherals using these .xml files. It was needed to run essential codes (input is soc_system_socpinfo file, and output is soc_system.dts; for the design only soc_system_board_info.xml file as

external peripheral is required) to generate .dts file and finally compile it to .dtb file and finish with the device tree. Note that it is needed to follow the instruction provided by Rocketboards web site and execute correct code lines in the correct directory from Embedded Command shell, in case of examining the process as a beginner.

The third and the most important stage, within the context of the Kernel design is the Toolchain obtaining as the heart of the entire system of the embedded design sequence. The Toolchain was obtained before, which is required to compile the Kernel. It can also be used in this stage, just to remind that in case the Shell window has been closed, cross compile command line could be ran again. The Kernel source needs to be acquired from Altera open source data base (in case a recent version would not be found, other online free resources could be searched). After downloading and unpacking the folder, its directory from the Shell window¹ should be navigated and the following be executed: “make ARCH=arm socfpga_defconfig” and then “make ARCH=arm menuconfig”. This will open a window which is illustrating in figure 25 and after required configuration it is possible to compile the Kernel.

¹ Navigating inside Linux file system can be done by “cd directory”.

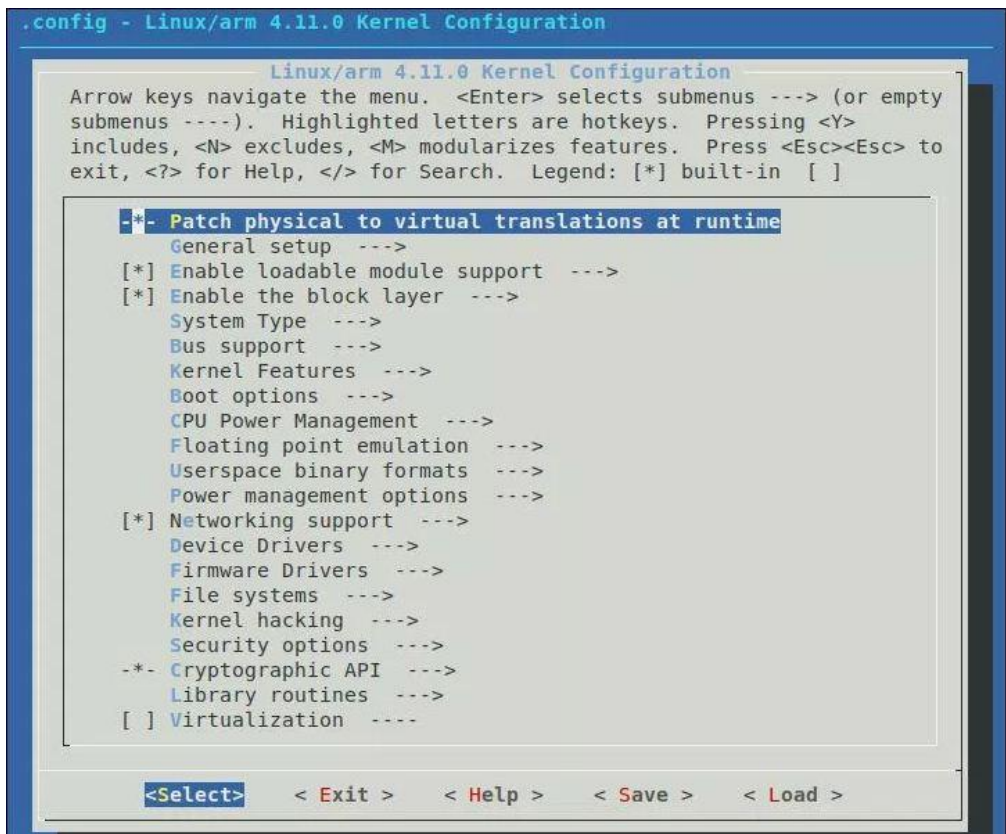


Figure 25. Kernel Configuration Window [9]

Here two options need to be changed: The first, “Automatically append version information of the version string” option which is in “General setup” tab and must be unchecked. When a Kernel module (driver) is loaded, the Kernel checks its version and in case of difference, the Kernel rejects loading. This feature of the Kernel could be disabled in order to be able to load various versions of drivers. The second option which needs to be changed is “Support for larger (2TB+) block devices and files” which can be found from “Enable the block layer” tab and it must be on. This option enables developers to mount ext4¹ file systems in write/read mode. (otherwise, ext4 can be mounted in read only mode). Now it is time for compiling the Kernel which is very crucial for the project, to do so “make ARCH=arm LOCALVERSION= zImage” should be ran. Once this operation is done (it can take a while), a zImage file will be generated that is a compressed version of the Linux distribution.

¹ Ext4 file system will be described in the SD Card creating part.

It is time now to take advantage of what have been done so far, which is creating own Linux distribution. It only requires root file system to be completed and utilized running on the target embedded board.

Finally, to create the own Linux distribution, a Root file system needs to be built and, as was mentioned previously, it contains essential files to boot the system up. It is required to configure a PATH to the Toolchain to compile Root file system:

```
“make -C buildroot ARCH=ARM  
BR2_TOOLCHAIN_EXTERNAL_PATH=$(pwd)/Toolchain directory_arm-linux-  
gnueabihf nconfig” which will open a configuration window shown in figure 26:
```

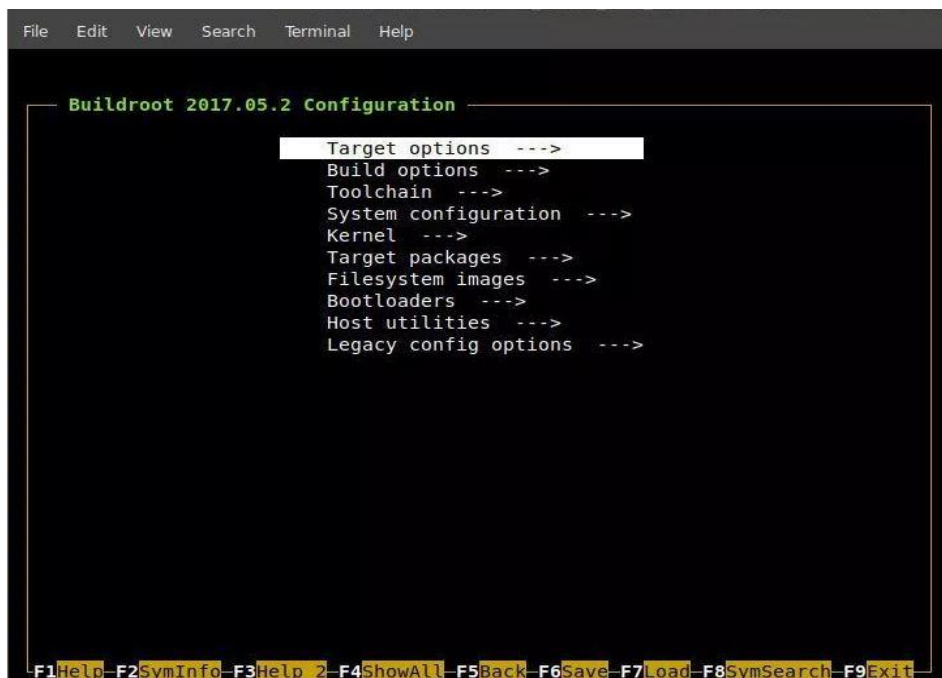


Figure 26. Buildroot Configuration Window [9]

From this window the author changed some properties in Target and Toolchain options on the base of the embedded Kernel features which is using ARM Cortex processor; so, the Buildroot should be configured accordingly. The other manipulation part is for Toolchain according to the compiled Toolchain which has been obtained from online sources. The configurations are demonstrating in below figures respectively:


```

Target options
-----
Target Architecture (ARM (little endian)) --->
Target Binary Format (ELF) --->
Target Architecture Variant (cortex-A9) --->
Target ABI (EABIhf) --->
[*] Enable NEON SIMD extension support
Floating point strategy (NEON) --->
ARM instruction set (ARM) --->

```

Figure 27. Target Configuration of Buildroot

```

Toolchain
-----
Toolchain type (External toolchain) --->
Toolchain (Linaro ARM 2014.09) --->
Toolchain origin (Pre-installed toolchain) --->
(/path/to/toolchain/usr) Toolchain path (NEW)
[*] Copy gdb server to the Target
[ ] Purge unwanted locales (NEW)
() Generate locale data (NEW)
[ ] Copy gconv libraries (NEW)
[*] Enable MMU support (NEW)
() Target Optimizations (NEW)
() Target linker options (NEW)
[ ] Register toolchain within Eclipse Buildroot plug-in (NEW)

```

Figure 28. Toolchain Configuration of Buildroot

After setting mentioned configuration, saving changes and exiting from the page (F6 + Enter and then F9), then it is time for moving to the next step of setting Busy box by running following piece of code: “make -C buildroot busybox-menuconfig”. it will open the configuration window and there is no need to do any change there, so just saving and exiting were needed. The only purpose of opening the Busy box without any manipulation is to inform the Kernel that all configurations are done. At the end the code that was used before to start the configurations of the Buildroot should be executed: “make -C buildroot ARCH=ARM BR2_TOOLCHAIN_EXTERNAL_PATH=\$(pwd)/Toolchain directory_arm-linux-gnueabihf all” and the Root file system will be generated in a few minutes, then it is time to think of the application which will be made as the Linux distribution is ready to upload them.

This chapter tried to have a revision of generating customized Linux distribution with desired FPGA design, Toolchain, U-Boot and Kernel source codes. I have used Terasic GHRD source and added the XOR component, obtained other required items mostly from Altera open source data base, and followed sequences to build the special Linux distribution, that could handle the user space applications and the Kernel modules (drivers). The next chapter demonstrates how to design a user space application by considering some example.

4.4 User Space Applications

It was indicated before that it is possible to operate in both user space and the Kernel space of the design. Their difference lies in their process running managements which proves that the memory dedicating for all processes is well organized in the Kernel. When developers run a process in the user space, only dedicated section of the memory, which has been defined by the application itself and specified by the Kernel, takes care of the process. Therefore, it is possible to run different application in the user space at the same time, moreover, drivers can be uploaded in the Kernel space. Thus, the visible difference of above-mentioned spaces can be specified as difference of applications' format, space for the process and memory indication. This chapter investigates user space application design flow, as figure 29 describes perfectly:

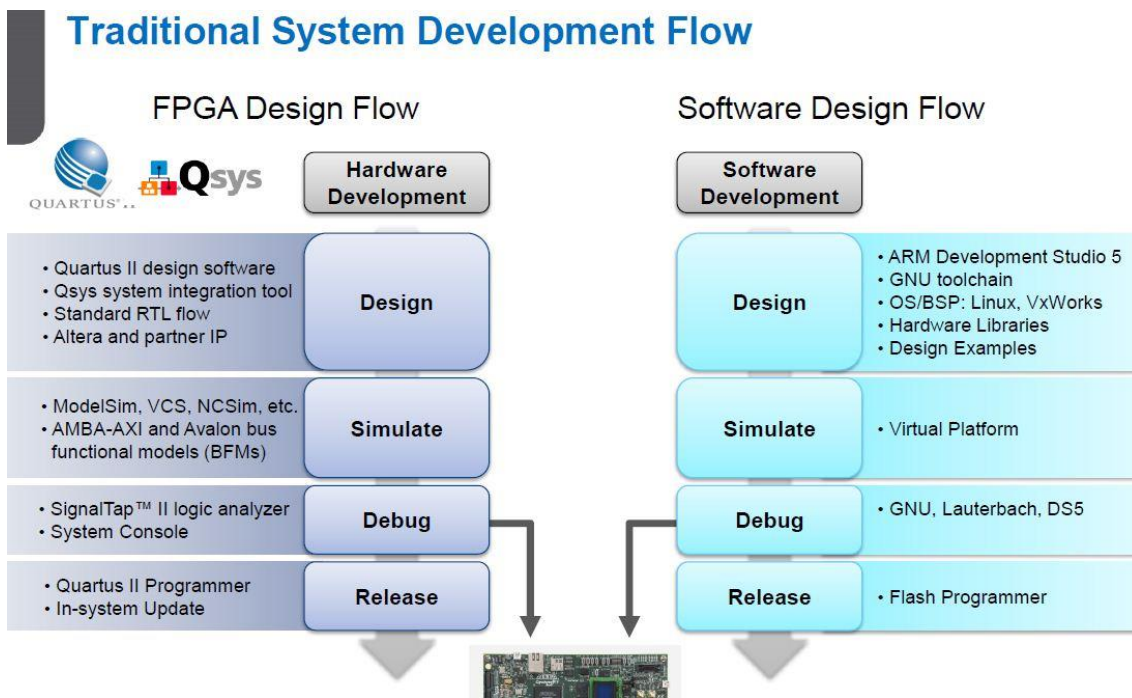


Figure 29. System Development Flow [21, p. 44]

As above figure illustrates, design flow for both user space applications and drivers (modules) in case of FPGA design is almost the same, except for the device tree generation for drivers which is not a part of user space application design sequence. Thus, it was needed to write VHDL/VeriLog code and design Qsys system if this project

requires to have physical access to FPGA. Otherwise, the project on the target embedded board using DS-5 (ARM Development Studio) could be tried, which is described in this chapter. In other words, if the project does not contain complexity (such as study projects or some professional ones that are being designed for simple purposes, for instance the Flash writing/reading application), the author could handle the situation only by writing code for DS-5 and examining the project on the board with Target Remote System if and if the project would not require any hardware design such as inserting, removing or any kind of components modification.

The author analyzed writing/reading from/to flash program as a user space application that does not require any Qsys manipulation, then investigated communicating via FIFO blocks, as an application that needs Qsys design, and inserted FIFO blocks as hardware components. Here is an overview on the board flash device: SoC Kit is equipped by a 512M-bit serial NOR flash device which is used for non-volatile HPS information such as Preloader image which is being used in U-Boot step of the design. Although the manufacturer has provided an access to the flash device using Quartus programmer, it is required to write C code in order to write to and read from the device. As it has been shown in figure 14, the flash device is connected to HPs side and “the HPS flash programmer sends file contents via USB Blaster II, to the HPS, and instructs the HPS to write the data to the flash memory” [13, p. 42].

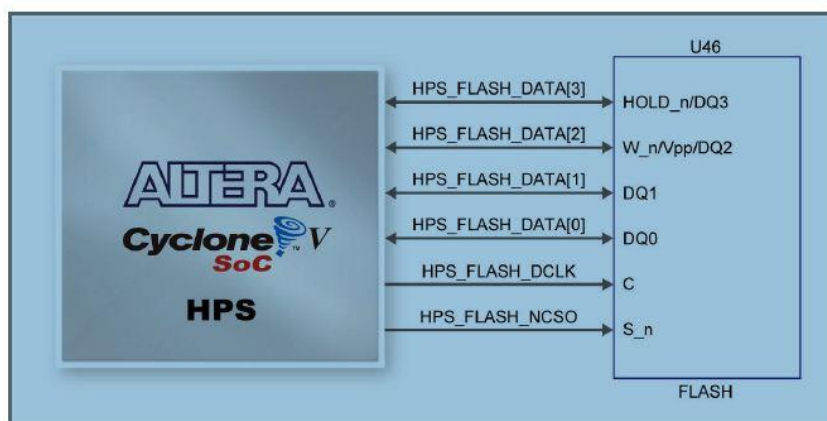


Figure 30. Flash Device Connections with SoC FPG [13, p. 42]

The reason that the author chose this flash program is to remind U-Boot process and emphasize on the importance of the non-volatile data structure type. Xiphera uses this

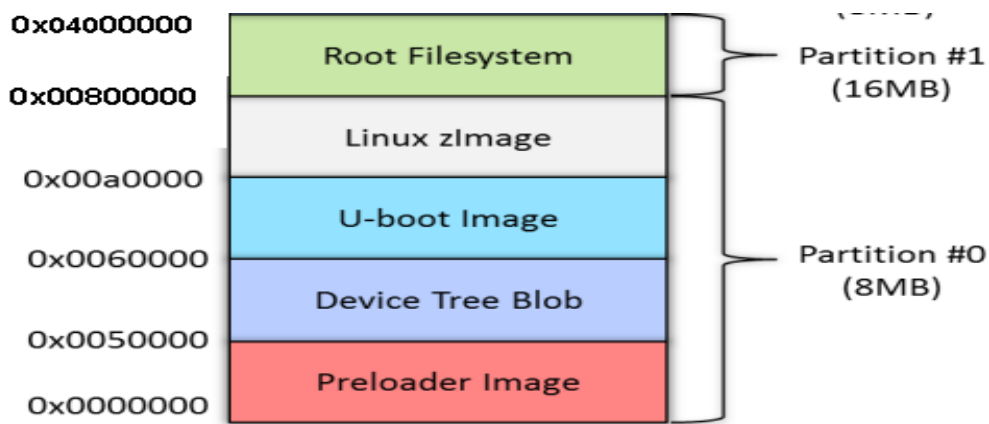


Figure 33. Flash Memory Partitioning

After demonstrating MTDs general structure, it is possible to focus on the design and C code to finalize the task about flash program. As it has been specified in this chapter, the data storage structure requires erasing before writing, which is considered here. The code general structure can be summarized as:

- 1- inserting an additional header file to control the MTD: `"# include <mtd/mtd-user.h>"`
- 2- Accessing the device from user space, as was operated there, and opening the device for reading and writing: `"mtd-info-t mtd-info"` , `"int fd = open("dev/mtd0" , ORDWR)"`
- 3- Data erase structure including getting device info, setting erase block size and erasing indicated block by following respectively: `"erase-info-t ei"` , `"ioctl (fd , MEMGETINFO , &mtd-info)"` , `"ei.length = mtd-info.erasesize"` and finally for the specified boundaries erasing operation: `"ioctl (fd , MEMERASE , &ei)"`
- 4- Reading from writing to the device: `"read (fd, read-buffer, sizeof(read-buf))"` , `"write (fd, data, sizeof(data))"`. Unsigned char characters `"read-buffer"` and `"data"`, which have been defined by the coder/author, include an empty sector and data, that requires to be written to the MTD accordingly.
- 5- Finally, the MTD needed to be closed by `"close (fd)"`

Then, the code should be tested to see if works as expected. To examine the code, it is needed to set DS-5 properly to have access to the SoC Kit. At the first chapter of this

section, it was described how to install DS-5 and open it from embedded Command Shell and run essential codes.

Now it is explained how to debug a written code by using Remote Target System:

- 1- Opening Eclipse and then from main menu window → show view → other → expand Remote Systems folder, choosing Remote System. After that, a new connection by clicking the New Connection button, shown on below picture, needs to be created.

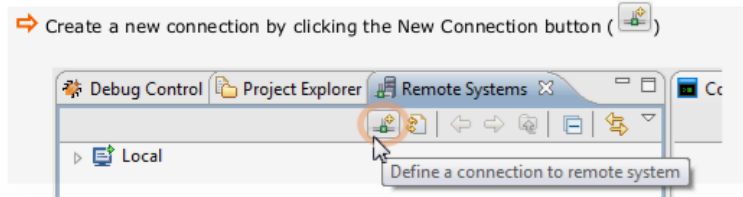


Figure 34. Creating a New Connection from RTS

Selecting SSH only → Next →, then putting the target's IP address¹ in the Host name field, while using SoCKit as the Connection name and clicking Next →

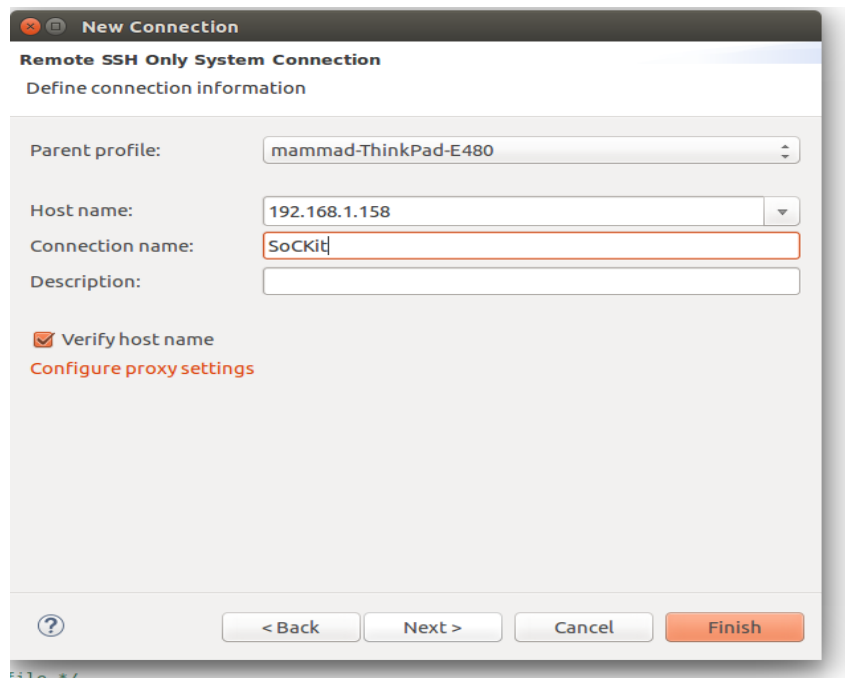


Figure 35. Configuration of RTS IP Address

¹ To get the IP address of the board: Boot the Linux on it by opening a communication window in a terminal window: “sudo screen /dev/ttyUSB0 115200” and switch on the board to boot Linux on it. It may ask for login password, give “root” to get in, then “ifconfig” to get the board IP address.

- 2- Checking “ssh.files” then clicking the “Finish” button. If “Next” is clicked instead of “Finish”, the rest of the default settings would be: “processes.shell.linux”, “ssh.shells”, and “ssh.terminals”.
- 3- Browsing the target's file system; Expanding “SoCKit” → “Sftp files” → “Root”. If the connection has “Files” instead of “Sftp Files” option, then the connection was not created correctly, and it is needed to disconnect, delete it and recreate it again.
- 4- Entering User ID=root, leaving the password blank; checking Save user ID and clicking the OK button. There will be a few authentication dialogs; accepting them all. A remote connection with the board is created.
- 5- To debug the project using newly established RTS, from Eclipse main menu, →debug configurations → DS-5 debugger should be run:

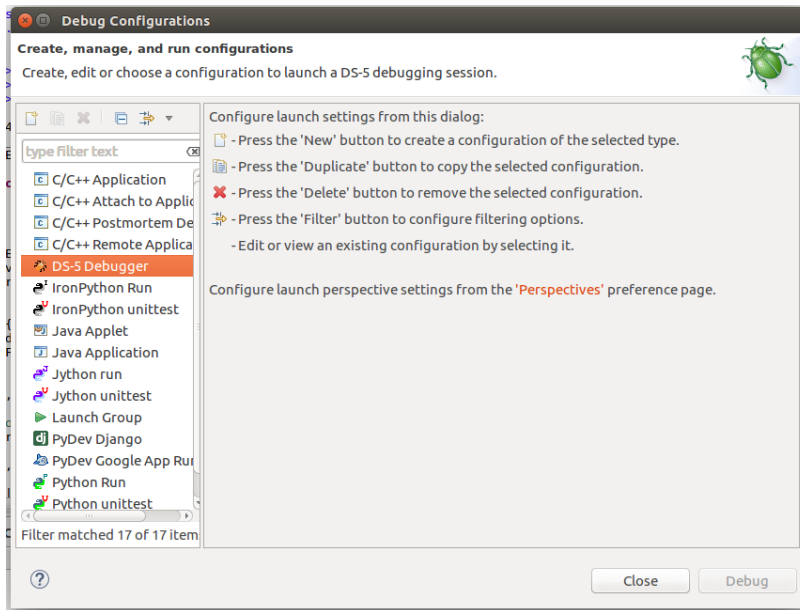


Figure 36. DS-5 Debugger Menu

- 6- Clicking on new (top-right corner of the menu) to open the debug configuration management window and giving the same name with the project and choosing

Download and debug application under Connections via gdbserver¹ in connection view:

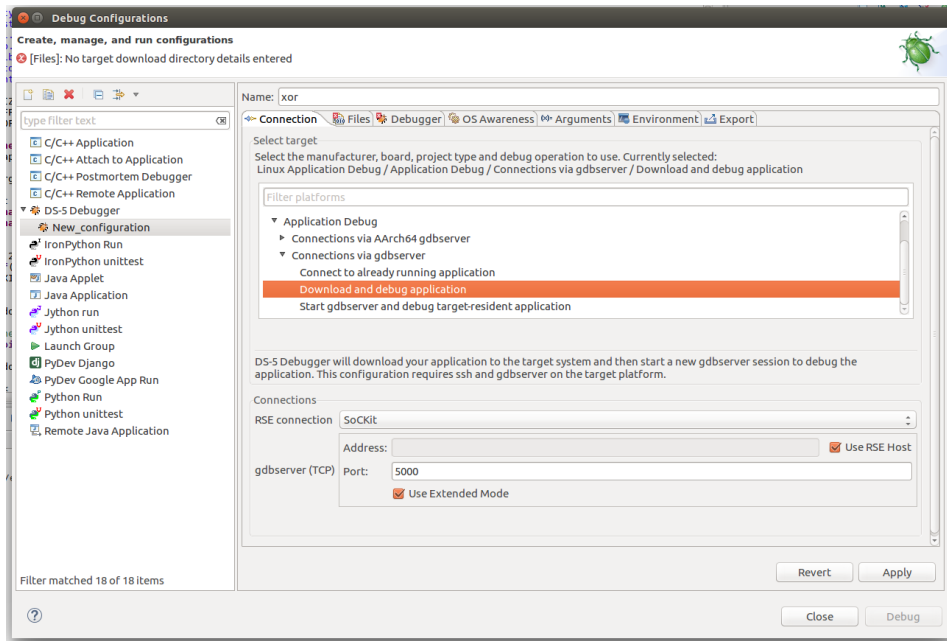


Figure 37. DS-5 Debugger Configuration Menu

In File view, assigning a directory for host machine to download (workspace is recommended →flash → debug →flash). “flash” is my program name.

- 7- In the same view, choosing target download and working directory (/home/root/flash for both of them in this case) and then selecting flash.o (object file) to load symbols from file option from workspace →flash→debug →flash.o and finally press apply and then debugging to open the DS-5 debugger view:

¹ gdbserver is a computer program that makes it possible to remotely debug other programs running on the same system as the program to be debugged and allows the GNU Debugger to connect from another system to the target board [1].

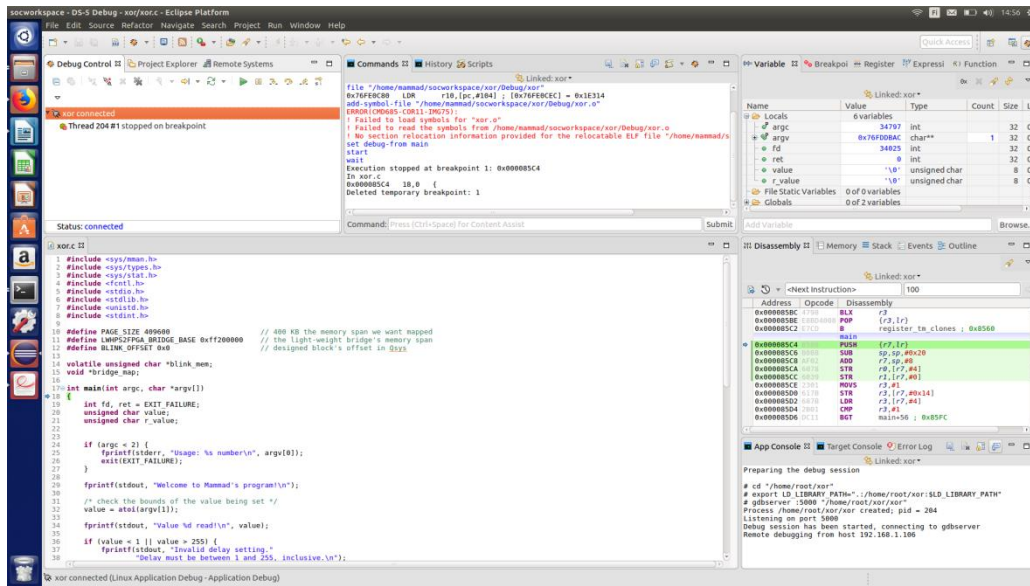


Figure 38. DS-5 Debugger View

After setting up RTS, it is easy to debug the code to find problems, trace them and solve. DS-5 debugger is similar to ARM Cortex one which is used in TUT embedded courses and has a very convenient user interface which needs to be explored by spending short time. It is required to switch off the board, turn it on again, boot Linux¹, enter the Linux and get the board's IP address to connect again to the host computer and finally read the content of the flash. If the read data is the same as it was written there a few minutes before, then it is possible to make sure that the duty is done and there is a program to write/read to/from the flash device.

It was explained that the SoC FPGA board's bridges' features before in chapter 3.2 and indicated that the Lightweight HPs to FPGA bridge data transmission is limited by 32 Bits, while other two bridges have 64 and 128 Bits interface options additionally. These mentioned features are considerably important, hence, they are analyzed and discussed in the next program. Depending on the application, only the LW bridge or all of them can be used. For instance, the XOR driver project, in FPGA design part, does not need so

¹ There is a default hardware setting with required files (GHRD) and Linux image generated on the base of these default configurations, which comes with the SoC development board from manufacturer. This Linux image contains all essential items for booting that can be copied on the SD Card and used to discover board's features by beginners. Own Linux distribution would be created by the end of this article and replace the initial one.

much data transmission between HPS and FPGA side as it is only LEDs manipulation project to examine the board property. However, sometimes a more complicated FPGA design is needed, which requires data transmission range more than LW bridge can provide. So, it is needed to utilize the other two bridges.

Different from the flash program, FIFO project requires Qsys design which includes creating one FIFO block for writing and another for reading, as the project could be called full communication between FPGA and HPS. In this example, HPS to FPGA FIFO block receives data which is waiting in FIFO block, then FIFO block reads this data to a buffer and sets a ready flag. Afterward, this data has been written to FPGA to HPS FIFO block and the ready flag can be cleared now. Figures 39 and 40 are showing Qsys design and FIFO block interface respectively:



Figure 39. Full FIFO communication Qsys Design

Bridges' interfaces have been shown before and in figure 40 only FIFO's interface is demonstrating. An Avalon-MM write master pushes data into FIFO and the read master pops it from FIFO's output port [23].

FIFO with Avalon-MM Input and Output Interfaces

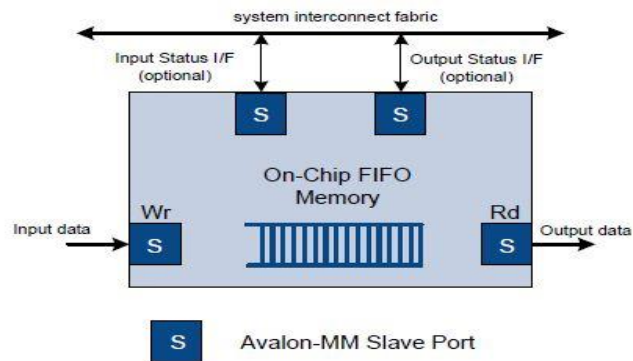


Figure 40. FIFO Interface [24, p. 2]

More detailed coding in XOR chapter will be shown and some points of the application just need to be mentioned here. It is needed to specify exact base address for both bridges and indicate the size of memory which will be occupied. Then opening the memory device, sending data to FIFO and reading them back. The general structure of the code is similar to the flash program code with only difference in code lines, as the target devices are different.

4.5 XOR Driver Development

XOR driver development is a part of Xiphera's IP block design process and tries to provide the most efficient representation of the whole design flow. As it has been described widely in previous chapters, driver development for embedded requires Linux customization combining with FPGA design and HPS user space applications. These development steps are related to each other and must be done in a correct order. Each step includes generating/obtaining source codes and organizing them to arrange correct configurations. The whole process is divided into 3 different sections, which require a team of experts. Xiphera as a start-up company planned to verify the whole process by the most efficient method to avoid excess financial and human recourses. The final purpose of this project has been defined as verifying hardware/software component in the most efficient way which is one of critical issues for start-up environments.

In the other words, developing more complicated drivers for SoC Kit (such as FIFO driver development to design a full-length communication between FPGA and HPS using all 3 bridges which is ongoing) requires a more complicated Qsys design, Xiphera's own definition for SoC Kit's pins assignment and Xiphera's specific device tree. Developing a pin map for SoC Kit without its peripherals (such as USB, UART and VGA connections and LCD) can be used for other FPGA (and not only SoC Kit). This pin map would be developer company's specific design and boost the company's reputation. However, developing such a pin map would take a longer period of time and requires a team work to divide definitions of pins to finalize the assignment in efficient way. Similarly, generating respective device tree can optimize FPGA design work time and developers can easily insert their new components to the device tree for different FPGAs. All mentioned items are only FPGA related issues and combining these designs with the Toolchain and Kernel source code must be considered before FPGA design. The Toolchain and the Kernel source codes either must be written by Xiphera or obtained from Altera open source data base. Most developers and start-up companies prefer to utilize free Toolchain and Kernel sources, however, this issue challenges their FPGA design; because all necessary versions of these sources are not available. Developing and obtaining these items take a long time and needs a greater team of experts, while the purpose of XOR project is verifying hardware/software component for Xiphera's Linux distribution in the fastest possible way. Therefore, a simple representation of the whole design flow is organized to observe the result as soon as possible. Different aspects of embedded Linux driver development and its sequences have been discussed in general. The purpose of the project and the reason to optimize existing resources to achieve to the final goal was explained. Now it is time to concentrate on actual purpose for this project which is XOR driver development flow.

Similar to all Linux embedded projects, XOR has three different design steps:

- 1- Qsys design for FPGA side which is at the same time a pre-request design for the device tree and Linux distribution generating. This part of design requires Quartus platform, our host machine and the target board.
- 2- Linux distribution obtaining and configuration which needs to be generated in the host computer and be tested on the board to observe if the recently created Linux boots without problem.

- 3- The last step of the XOR driver development is composing code for driver itself which can be done using C code and insert to the newly generated Linux from host machine and finally be examined on the board.

At the end of each part of the design I needed to test the obtained results in order to solve probable problems before continuing to the other part. All three parts have been considered in different chapters, and now I am going to combine all previous described information together and do a practical experiment. Please note that the purpose of this project is verifying the board functionalities such as booting, hardware component and LW bridge interface. This validation will be exposed on four LEDs in FPGA side which can be controlled from Quartus, HPS side using DS-5 as well as from Linux terminal window utilizing the produced driver. As it has been mentioned before, XOR uses LW bridge AXI interface and takes an input value in the range of 1 – 255, changes this value to a HEX number, executes XOR operation between lower and higher nibbles of this HEX integer and finally sends the result down to the FPGA side LEDs. For instance, if one gives the input value of 100 from either DS-5 debugger argument page or a terminal window, which is connected to the board with any screen program, XOR converts it to 64 HEX-decimal integer and then execute XOR operation between 4 (0100) and 6 (0110) = 0010 and send the result to the FPGA's LEDs and by the exposing result, the second LED will be turned on.

A general view of the project, by obtaining the purpose and functionality of it, has been provided. Now, it is beneficial to continue step by step. Qsys design flow and its necessity for device tree were explained in chapter 4.2. I inserted the customized component which is called xiphera_test_block to the already existing GHRD Qsys design and appended essential code line to .tcl file then generated VHDL code. Eventually, requiring connections were added to the recently inserted component and finished Qsys design. When a new component is meant to be added to the Qsys design, from configuration menu of the new component, it is needed to add VHDL/VeriLog code which in this case it was two VHDL files; one for xiphera IP block and the other one for Avalon interface. Before synthesis in Quartus, it is needed to add xiphera_test_block to the top level VHDL code instead of old component name which in this case it was LED_PIO not forgetting to run tcl script file before full synthesis. If directive is followed step by step,.sof file would be achieved which can be converted to .rbf file using Quartus file converter.

The second part of the design is C code developing to try the FPGA design using DS-5 platform. This part is a user space application project, but the necessity of this section is understood when the driver's code would be written. I needed to create a new project in DS-5 and added the C code then built the project to generate executable file and connected to the target board using RTS to debug it.

Now, the C code, which is a step toward the final C code of driver, needs to be analysed.

The code can be summarized as:

- 1- Header files, specifying memory space and the base address which has been illustrated in figure 41: “

```
#define PAGE_SIZE 409600
```

```
#define LWHPS2FPGA_BRIDGE_BASE 0xff200000
```

```
#define xiphera_test_block_OFFSET 0x0”, variables definitions, indicating input value boundaries (1 and 255)
```

- 2- Opening the memory device file with fd: “

```
fd = open (“/dev/mem”, ORDWR|O_SYNC);” and mapping LW HPS to FPGA bridge into process memory with mmap: “
```

```
bridge_map = mmap(NULL, PAGE_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd ,LWHPS2FPGA_BRIDGE_BASE);”
```

- 3- Getting the designed peripheral's (xiphera_test_block) base address which is 00 in my case: “

```
xiphera_test_block_mem = (unsigned char *) (bridge_map + BLINK_OFFSET)  
;”, write the input value into ” xiphera_test_block_mem” and close the file device by “close (fd)”.
```

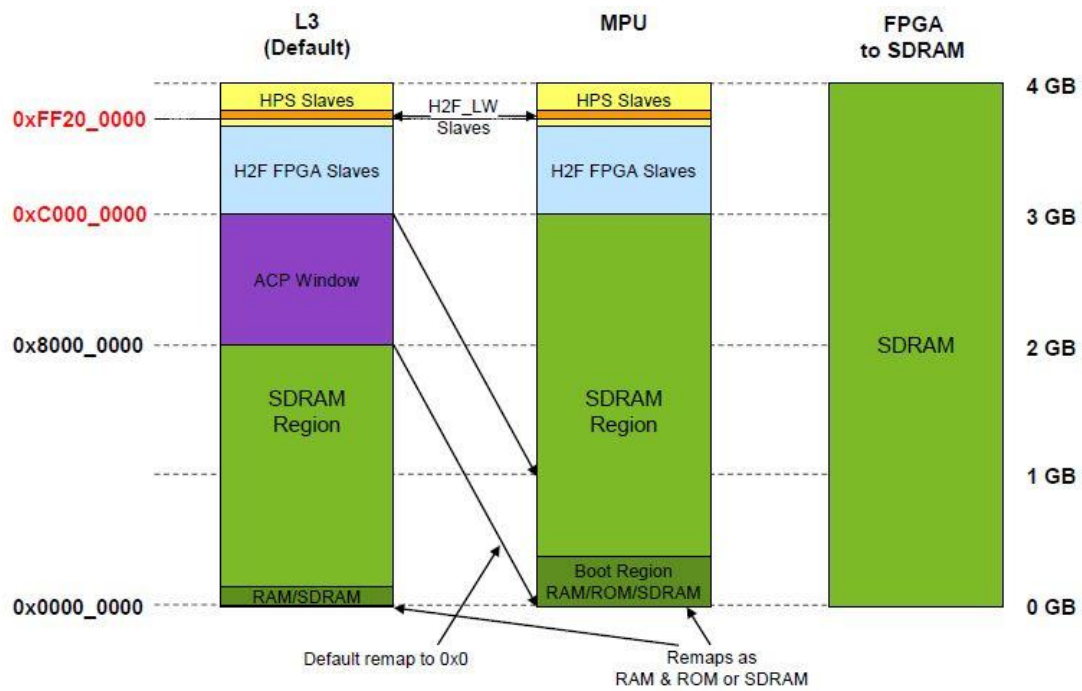


Figure 41. Cyclone V SoC HPs Memory Map [19, p. 55]

The code can be debugged to find solutions for possible problems, and if everything works properly, then it is possible to continue with Linux distribution development part, which has been already explained. To sum up, Quartus design, C code of user application and customized Linux on the base of the Qsys design and device tree manipulations, have been developed. Now, there remains only driver codes, Linux configuration to compile the driver and finally, gather everything together, transfer to a SD Card and try on the board.

The general structure of my driver C code can be summarized as:

- 1- Inserting header files: depending on the expected functionalities and aim of the driver, header files needed to be added. Developing of the algorithm of the driver design flow was one of the author's crucial tasks during this project. Having an algorithm design was helpful to find software requirements of the driver. XOR driver needs to take an input value from user space and map this value into the process memory and finally shows the value using FPGA's LEDs. As it is understood from the algorithm, XOR driver requires including "`<linux/uaccess.h>`" to access user space and copy the input value into the

allocated memory. XOR also needed inserting “<linux/ioport.h> and <linux/io.h>” to manipulate FPGA’s LEDs. Including “<linux/init.h>, <linux/module.h>, <linux/kernel.h> and <linux/device.h>”, provides essential prototypes for initializing/exiting, inserting/removing XOR driver to the Kernel and functions such as “*drv” (driver)/“*buf” (buffer). In order to define functions such as containing the input value, which is taken from user space, and writing this value into the memory, XOR uses “<linux/platform_device.h>”. Finally, XOR needs communicating between user and the Kernel spaces which should be handled in file system format as a requirement of Linux Kernel design. “<linux/kobjects.h> and <linux/sysfs.h>” header files include attributer functions such as “DRIVER_ATTR show and store”. These header files create a directory in “/sys/bus/platform/drivers/xiphera_test_block” of SoC Kit root and an entry point in this directory. This directory and its entry point are used to communicate with XOR driver which is in the Kernel space. Users can enter an input value between 1 – 255 from the terminal window, which is used to insert XOR driver to the Kernel. As it has been described initially, users operate in user space and need to send the input value to the entry point of the specified root directory of SoC Kit Kernel space.

- 2- Defining the inserted component address base which has been declared in SoC Kit data sheet and can be obtained from Qsys design. This memory space has been allocated on the base of LWbridge interface which has been used for this project. 400 KB of the memory span was defined for XOR to be mapped: “

```
#define xiphera_test_block_BASE 0xff200000

#define xiphera_test_block_SIZE PAGE_SIZE

#define PAGE_SIZE 409600”,
```

Informing the Kernel about the device tree which is being used by XOR driver and linking essential functions with XOR with following data structure: “

```
MODULE_DEVICE_TABLE(of, xiphera_test_block_dt_ids);

static struct platform_driver xiphera_test_block = {
```



```

.probe = xiphera_test_block_probe,

.driver = {

    .name = "xiphera_test_block",

    .owner = THIS_MODULE,

    .of_match_table = xiphera_test_block_dt_ids

}

};",

```

declaring “driver_attribute” which is the Kernel’s sysfile structure [25] and finally registering the driver, specifying license and inserting initialize and exit lines¹.

3- Creating directory and entry point of sysfs file and removing this configuration after exiting from driver: “

```

driver_create_file(&xiphera_test_block_driver,
&driver_attr_xiphera_test_block);
driver_remove_file(&xiphera_test_block_driver,
&driver_attr_xiphera_test_block);”

```

4- Accessing to I/O memory: “

```

res          =          request_mem_region(xiphera_test_block_BASE,
xiphera_test_block_SIZE, "xiphera_test_block");

if (res == NULL) {

driver_unregister(&xiphera_test_block_driver);

```

¹ It is highly recommended that driver writers refer to Linux Device Drivers book as a really helpful material in order to analyze module writing steps and techniques deeply and understand using of codes.

```
return -EBUSY; }”, re-mapping the component’s address into processor memory with “ioremap” function (as it has been done in XOR user space application code with “mmap( PROT_READ|PROT_WRITE)”): “
```

```
xiphera_test_block_mem=ioremap(xiphera_test_block_BASE,  
xiphera_test_block_SIZE);
```

```
if (xiphera_test_block_mem == NULL) {
```

```
release_mem_region(xiphera_test_block_BASE,  
xiphera_test_block_SIZE);
```

```
return -EFAULT;}” and finally write the input value which is a single bit into “xiphera_test_block_mem”: “iowrite8(value,  
xiphera_test_block_mem);”.
```

Then it is possible to setup required Kernel environment to compile XOR driver as it has been mentioned previously in Linux distribution development chapter. It is needed to run “export ARCH=arm” as this project aims to develop driver for ARM Cortex-9 processor of SoC Kit. In addition, I defined “export CROSS_COMPILE= ...” which is path the Toolchain directory in my host computer to compile and generate executable file using already obtained Toolchain source. Moreover, “export OUT_DIR=...” was executed to set the Kernel source directory in the host machine in order to use the Kernel source code. Above mentioned lines of code required to be compiled either from Command Shell window or using a Makefile from the host machine to set variables properly and compile recently designed driver to insert it to the Kernel. XOR driver projects followed the first method (compiling variable configurations from Command Shell terminal) to keep the Makefile as simple as possible.

The next step was creating the Makefile and a Kbuild file with below contents respectively:

```
“KDIR ?=OUT_DIR
```

```
default:
```

```
$(MAKE) -C $(KDIR) ARCH=arm M=$$PWD” and
```

```
“obj-m := xiphera_test_block.o”.
```

Compiling these two files generates “.ko” file which is containing all configuration and codes of XOR driver. This “.ko” file would be uploaded into the SD Card and inserted to the Kernel running on SoC Kit target. Then “make” command was executed from the Embedded Command Shell window to generate “.ko” file in the indicated output directory. Generating this file took a longer while than usual time in my case because there were some error messages in my code which came up during compilation. All error messages were written from terminal window and modified in “xiphera_test_block.c” file. So every time I needed to verify my code it was essential that the code would be debugged and probable error messages get corrected. This process happens during the driver code debugging before inserting the driver to the Kernel. In case the Makefile compiling step would be passed without any error and the generated “.ko” file inserted to the Kernel, the actual debugging starts. XOR debugging took a few days as I needed to examine it on the board and found out its problem, modified occurred errors and re-compile the Makefile again. The is a reasonable explanation for long while debugging period; As it has been mentioned previously, XOR project is a combination of Qsys design, Linux distribution development sequence and C code composing. These steps are linked together and must be followed in a correct order to obtain the final “.ko” file. When an error occurs, the developer has to debug all steps of background design flow. This background includes hardware configuration and verification which must be checked during development every step to avoid facing with possible problems. The hardware maintenance was simple part of the XOR driver project as SoC Kit was recently manufactured and tested. The most difficulty was software debugging process which contains different items. There were so many software related issues during this design period such as:

- MSEL pin configuration as there are different set of configurations for various modes (such as FPGA running from Quartus, programming from HPS, etc.). During Kernel booting into SoC Kit, incorrect MSEL pin configuration caused a problem and board hung. There were problems with USB cables which sometimes were stoping the design process. SD Card partitioning was another critical problem which required formatting and re-partitioning again.

- Qsys design related errors such as ignoring to run “.tcl” file before full synthesis in Quartus. This mistake causes problem with the device tree and prevent developer to add the pin assignment properly, consequently compile the device tree correctly. The other issue was adding customized hardware component to the device tree before generating VHDL during final step of Qsys design. If these issues would be ignored, driver developer has to start Qsys design from beginning which happened two times in XOR project.
- Linux related problems such the Kernel source code obtaining and its existing versions as well as the Toolchain code. In case other dominant items would be verified, there is no doubt except the available Kernel source code and its version if is matched with the Linux version on developer’s host machine.
- C code modification which is required to be done every time before compiling the Makefile.

The final step of XOR project was transferring generated files to a SD Card which required to be partitioned by following steps:

- 1- Inserting the SD Card (at least 1 GB memory space is required as the created Linux image file and other essential files occupy more than 512 MB) and mounting it¹.
- 2- Executing “`sudo fdisk /dev/sdb`” (/dev/sdb is the SD Card directory in my host computer and can be obtained by “`lsblk`”) gives instruction to specify partitions’ size and type.
- 3- Creating 3 partitions with 1, 254 and 256 MB sizes and unknown, Linux and FAT32 types respectively. (“n” for new partition, “t” for type specification and “w” to save configurations).
- 4- Running “`sudo partprobe /dev/sdb`” to aware the host Kernel about changes which have happened.
- 5- Creating file systems for the second and third partitions as the first one is a raw file type. “`sudo mkfs.ext4 /dev/sdbp2`” and “`sudo mkfs -t vfat`”

¹ “`sudo mount /dev/sdb`” sdb is my SD Card name which can be shown by running “`lsblk`”.

```
/dev/sdbp3” and then creating mount points for these partitions: “mkdir
sdbp2_mount
sudo mount /dev/sdbp2 sdbp2_mount/” and “
mkdir sdbp3_mount
sudo mount /dev/sdbp3 sdbp3_mount/”.
```

6- Copying “preloader-mkpimage.bin” file directly into the first partition (using “sudo dd if=... of=...” and then executing “sync” to do physical copy operation.

7- Running

```
“sudo cp ../u-boot.img ../u-boot.scr soc_system.dtb1 soc_system.rbf2
../zImage3 sdbp3_mount/” and then “sync” to copy booting files and the Kernel
image to the FAT partition of the SD Card.
```

8- Running “sudo tar -xvf ../rootfs.tar -C sdbp2_mount/” and then “sync” to extract required files into the second partition of the SD Card.

9- Copying “.ko” file to the second partition:

```
“sudo cp xiphera_test_block.ko sdbp2_mount/” and then “sync”.
```

10- Un-mounting the second and third partitions: “

```
sudo umount sdbp2_mount/
```

```
sudo umount sdbp2_mount/” and removing the SD Card and insert it to the
board.
```

Now it is time to boot Xiphera’s own Linux on the board. After booting XOR driver can be initialized by running “insmod xiphera_test_block.ko”. The driver can be checked by giving an input value:

“

¹ The device tree bulb, which is generated by device tree when the FPGA design is finished.

² The output file of Quartus file converter, which has been converted from .sof as Qsys design result.

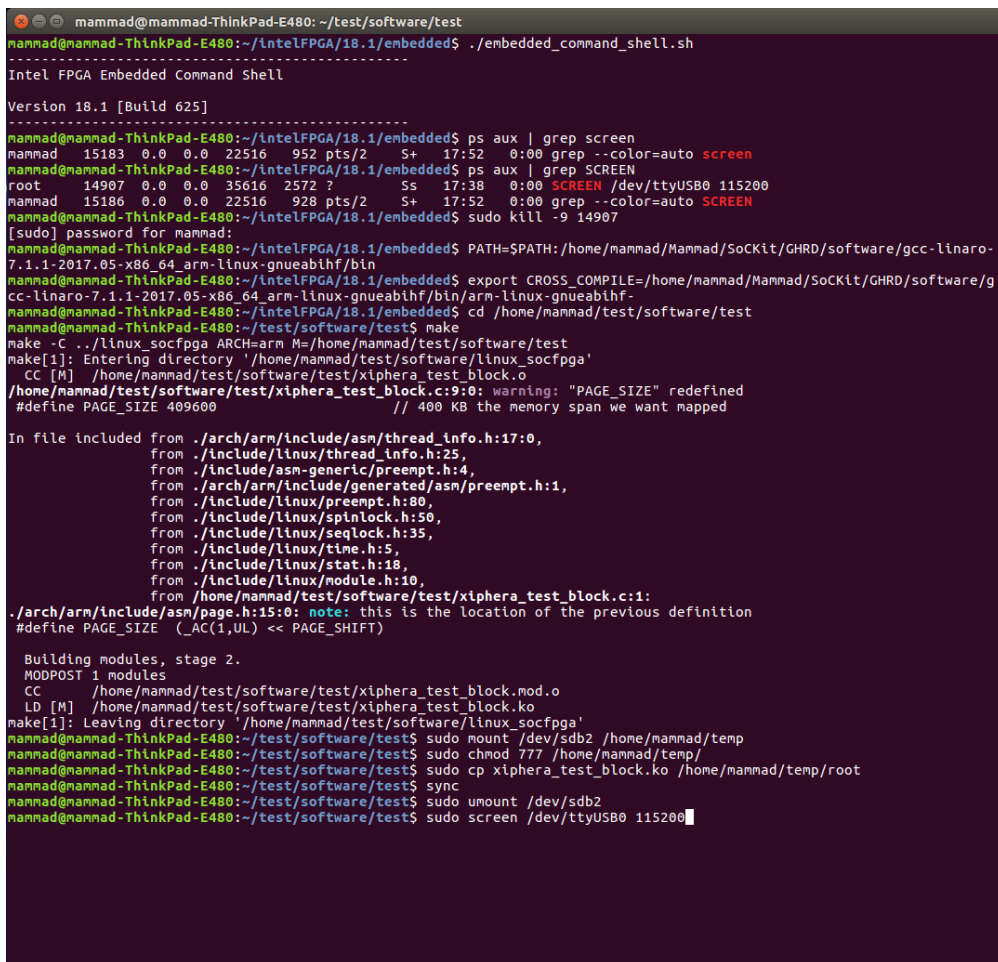
³ Linux image file, which has been generated in Linux distribution development section.

echo

“30”

/sys/bus/platform/drivers/xiphera_test_block/xiphera_test_block”

As it has been indicated before, the HEX representation of 30 is 1E with 0001 higher and 1110 lower nippers and XOR operation gives the 1111 result, which switches all FPGA’s LEDs on for SoC Kit. (“rmmod xiphera_test_block.ko” is using to unload the driver). Figures 42, 43 and 44 are demonstrating the Makefile compilation and copying “xiphera_test_block.ko” file to the SD Card, “xiphera_test_block” driver initializing and verifying the XOR operation using this driver respectively:



```
mammad@mammad-ThinkPad-E480: ~/test/software/test
mammad@mammad-ThinkPad-E480: ~/intelFPGA/18.1/embedded$ ./embedded_command_shell.sh
-----
Intel FPGA Embedded Command Shell
Version 18.1 [Build 625]
-----
mammad@mammad-ThinkPad-E480:~/intelFPGA/18.1/embedded$ ps aux | grep screen
mammad 15183 0.0 0.0 22516 952 pts/2 S+ 17:52 0:00 grep --color=auto screen
mammad@mammad-ThinkPad-E480:~/intelFPGA/18.1/embedded$ ps aux | grep SCREEN
root 14907 0.0 0.0 35616 2572 ? Ss 17:38 0:00 SCREEN /dev/ttyUSB0 115200
mammad 15186 0.0 0.0 22516 928 pts/2 S+ 17:52 0:00 grep --color=auto SCREEN
mammad@mammad-ThinkPad-E480:~/intelFPGA/18.1/embedded$ sudo kill -9 14907
[sudo] password for mammad:
mammad@mammad-ThinkPad-E480:~/intelFPGA/18.1/embedded$ PATH=$PATH:/home/mammad/Mammad/SoCKit/GHRD/software/gcc-linaro-7.1.1-2017.05-x86_64_arm-linux-gnueabi/btn
mammad@mammad-ThinkPad-E480:~/intelFPGA/18.1/embedded$ export CROSS_COMPILE=/home/mammad/Mammad/SoCKit/GHRD/software/gcc-linaro-7.1.1-2017.05-x86_64_arm-linux-gnueabi/btn/arm-linux-gnueabi/
mammad@mammad-ThinkPad-E480:~/intelFPGA/18.1/embedded$ cd /home/mammad/test/software/test
mammad@mammad-ThinkPad-E480:~/test/software/test$ make
make -C ../linux_socfpga ARCH=arm M=/home/mammad/test/software/test
make[1]: Entering directory '/home/mammad/test/software/linux_socfpga'
CC [M] /home/mammad/test/software/test/xiphera_test_block.o
/home/mammad/test/software/test/xiphera_test_block.c:9:0: warning: "PAGE_SIZE" redefined
#define PAGE_SIZE 409600 // 400 KB the memory span we want mapped
In file included from ./arch/arm/include/asm/thread_info.h:17:0,
from ./include/linux/thread_info.h:25,
from ./include/asm-generic/preempt.h:4,
from ./arch/arm/include/generated/asm/preempt.h:1,
from ./include/linux/preempt.h:80,
from ./include/linux/spinlock.h:50,
from ./include/linux/seqlock.h:35,
from ./include/linux/tine.h:5,
from ./include/linux/stat.h:18,
from ./include/linux/module.h:10,
from /home/mammad/test/software/test/xiphera_test_block.c:1:
./arch/arm/include/asm/page.h:15:0: note: this is the location of the previous definition
#define PAGE_SIZE (_AC(1,UL) << PAGE_SHIFT)

Building modules, stage 2.
MODPOST 1 modules
CC /home/mammad/test/software/test/xiphera_test_block.mod.o
LD [M] /home/mammad/test/software/test/xiphera_test_block.ko
make[1]: Leaving directory '/home/mammad/test/software/linux_socfpga'
mammad@mammad-ThinkPad-E480:~/test/software/test$ sudo mount /dev/sdb2 /home/mammad/temp
mammad@mammad-ThinkPad-E480:~/test/software/test$ sudo chmod 777 /home/mammad/temp/
mammad@mammad-ThinkPad-E480:~/test/software/test$ sudo cp xiphera_test_block.ko /home/mammad/temp/root
mammad@mammad-ThinkPad-E480:~/test/software/test$ sync
mammad@mammad-ThinkPad-E480:~/test/software/test$ sudo umount /dev/sdb2
mammad@mammad-ThinkPad-E480:~/test/software/test$ sudo screen /dev/ttyUSB0 115200
```

Figure 42. Makefile compilation and transferring to the SD Card

```

mammad@mammad-ThinkPad-E480: ~
response 0x900, card status 0x0
[ 10.894874] mmcblk0: retrying using single block read
Starting syslogd: OK
Starting klogd: OK
Initializing random number generator... done.
Starting network: OK

Welcome to Buildroot
buildroot login: root
# insmod xiphera_test_block.ko
[ 24.354870] mmcblk0: error -110 transferring data, sector 29442, nr 228, cmd
response 0x900, card status 0x0
[ 24.394812] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 400000H
z, actual 400000Hz div = 250)
[ 24.461285] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 2500000
0Hz, actual 250000000Hz div = 4)
[ 24.684870] mmcblk0: error -110 transferring data, sector 29442, nr 228, cmd
response 0x900, card status 0x0
[ 24.694827] mmcblk0: retrying using single block read
[ 24.915421] random: crng init done
[ 24.950499] xiphera_test_block: loading out-of-tree module taints kernel.
[ 24.957726] Initializing the Xiphera Xor_Test module
[ 24.962820] Xiphera Xor_Test module successfully initialized!
#

```

Figure 43. XOR driver initializing

```

mammad@mammad-ThinkPad-E480: ~
5.934069] mmcblk0: error -110 transferring data, sector 21948, nr 256, cmd response 0x900, card status 0x0
5.944029] mmcblk0: retrying using single block read
6.284067] mmcblk0: error -110 transferring data, sector 22716, nr 256, cmd response 0x900, card status 0x0
6.324003] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 400000Hz, actual 400000Hz div = 250)
6.390486] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 250000000Hz, actual 250000000Hz div = 4)
6.614972] mmcblk0: error -110 transferring data, sector 22716, nr 256, cmd response 0x900, card status 0x0
6.624032] mmcblk0: retrying using single block read
6.964076] mmcblk0: error -110 transferring data, sector 22428, nr 256, cmd response 0x900, card status 0x0
7.004003] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 400000Hz, actual 400000Hz div = 250)
7.070490] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 250000000Hz, actual 250000000Hz div = 4)
7.294069] mmcblk0: error -110 transferring data, sector 22428, nr 256, cmd response 0x900, card status 0x0
7.304027] mmcblk0: retrying using single block read
7.644065] mmcblk0: error -110 transferring data, sector 282914, nr 256, cmd response 0x900, card status 0x0
7.684002] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 400000Hz, actual 400000Hz div = 250)
7.750493] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 250000000Hz, actual 250000000Hz div = 4)
7.974070] mmcblk0: error -110 transferring data, sector 282914, nr 256, cmd response 0x900, card status 0x0
7.984202] mmcblk0: retrying using single block read
8.454077] mmcblk0: error -110 transferring data, sector 283530, nr 256, cmd response 0x900, card status 0x0
8.494003] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 400000Hz, actual 400000Hz div = 250)
8.560491] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 250000000Hz, actual 250000000Hz div = 4)
8.784071] mmcblk0: error -110 transferring data, sector 283530, nr 256, cmd response 0x900, card status 0x0
8.794119] mmcblk0: retrying using single block read
9.134096] mmcblk0: error -110 transferring data, sector 22204, nr 200, cmd response 0x900, card status 0x0
9.174003] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 400000Hz, actual 400000Hz div = 250)
9.240491] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 250000000Hz, actual 250000000Hz div = 4)
9.464072] mmcblk0: error -110 transferring data, sector 22204, nr 200, cmd response 0x900, card status 0x0
9.474003] mmcblk0: retrying using single block read
9.920072] EXT4-fs (mmcblk0p2): re-mounted. Opts: data=ordered
10.034079] mmcblk0: error -110 transferring data, sector 283314, nr 168, cmd response 0x900, card status 0x0
10.074007] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 400000Hz, actual 400000Hz div = 250)
10.140492] mmc_host mmc0: Bus speed (slot 0) = 200000000Hz (slot req 250000000Hz, actual 250000000Hz div = 4)
10.364077] mmcblk0: error -110 transferring data, sector 283314, nr 168, cmd response 0x900, card status 0x0
10.374079] mmcblk0: retrying using single block read
Starting logging: OK
Initializing random number generator... done.
Starting network: OK

Welcome to Buildroot
buildroot login: root
# rmdir xiphera_test_block.ko
# insmod xiphera_test_block.ko
# echo "200" > /sys/bus/platform/drivers/xiphera_test_block/xiphera_test_block
# echo "100" > /sys/bus/platform/drivers/xiphera_test_block/xiphera_test_block
# echo "128" > /sys/bus/platform/drivers/xiphera_test_block/xiphera_test_block
# echo "228" > /sys/bus/platform/drivers/xiphera_test_block/xiphera_test_block
# echo "78" > /sys/bus/platform/drivers/xiphera_test_block/xiphera_test_block
# echo "79" > /sys/bus/platform/drivers/xiphera_test_block/xiphera_test_block
# echo "9" > /sys/bus/platform/drivers/xiphera_test_block/xiphera_test_block
# echo "29" > /sys/bus/platform/drivers/xiphera_test_block/xiphera_test_block
#

```

Figure 44. Communicating with driver and executing XOR operation

The design flow was finalized, as XOR driver has been checked. The development and the obtained result were confirmed, which was the purpose of this project. The same result had been observed previously in Quartus design and user application processes.

5 Summary

Linux driver development for SoC FPGA requires deep knowledge of Linux programming, driver development flow and SoC FPGA design sequence. Mastering in these different fields needs years of experience while beginner developers who try to run their desired Linux driver need a simple and at the same time a comprehensive guide of whole process. This research pursued the fact of simplicity and practical guidance to provide a clear instruction of Linux driver development. It started with importance of customized Linux and its freedom and flexibility. In the second part, hardware component has been described and finally at the last section, Linux distribution obtaining has been explained.

All embedded designs require five elements of Toolchain, Bootloader, Kernel, Root filesystem and applications which have been analysed step by step. FPGA Qsys design which is pre-request of the device tree generation and the second item of mentioned sequence, has been described separately. DS-5 platform installation and user space applications has been considered with examples. Finally, I described how to design and compile XOR driver which is a part of Xiphera encrypting Ip block project. The purpose of XOR driver development project was verifying hardware/software component for SoC Kit in efficient way, which required a simple driver representation of whole design process. The aim of the project has been realized as the author could design and check XOR driver on the board successfully.

References

- [1] "Wikipedia," [Online]. Available: <https://www.wikipedia.org/> [Please note that in this research Wikipedia has been used only for superficial information such as year of productions, nationality of the developers and explanation of abbreviations].
- [2] StatCounter, "Global market share held by operating systems," 2013 - 2019.
- [3] R. & L. D.-A. Blum, *Linux for Dummies*, 9th Edition, Hoboken, NJ: Wiley Publishing, Inc., 2009.
- [4] Bootlin, "Linux Kernel and Driver Development Training," Bootlin, 2004-2019.
- [5] Altera, "SoC Devices Workshop 3: Developing Drivers for Altera SoC," Altera-Public, Austin, USA, 2016.
- [6] Altera, "SoC Devices Workshop 2: Altera SoC Linux Introduction," Altera, Austin, USA, 2016.
- [7] c. Simonds, *Mastering Embedded Linux Programming*, 2.edition, Birmingham, UK: Packt Publishing Ltd., 2017.
- [8] R. Blum, *Professional Assembly Language*, Indianapolis, IN: Wiley Publishing, Inc., 2005.
- [9] "RocketBoards," [Online]. Available: <https://rocketboards.org/foswiki/Documentation/EmbeddedLinuxBeginnerSGuide>.
- [10] Rocketboards, "SoC Kit Tutorials," 2015.
- [11] "ARM Community," [Online]. Available: <https://community.arm.com/developer/tools-software/oss-platforms/w/docs/293/u-boot>.
- [12] Bootlin, "Linux Kernel and Driver Development Training," 2004-2019.
- [13] A. & T. & Altera, "SoCKit User Manual," 2003-2014.
- [14] "Anysilicon," 30 January 2016. [Online]. Available: <https://anysilicon.com/fpga-vs-asic-choose/>. [Accessed 2019].
- [15] Ramdas, "ASIC vs SOC FPGA," *Verification Excellence*, 2016.
- [16] Altera, "Architecture Brief of SoC FPGA," Altera, San Jose, CA, 2014.
- [17] T. Instruments, "Multicore SoCs: stay a step ahead of SoC FPGAs," Texas, Instruments, Dallas, Texas, 2016.
- [18] V. Rajaraman, "IEEE Standard for Floating Point Numbers," Bengaluru, India, 2016.
- [19] Altera, "Cyclone V hard Processor System Technical Reference Manual," Altera, San Jose, Ca, 2018.
- [20] Intel, "Avalon Interface Specifications," 2018.
- [21] Altera, "SoC Devices Workshop 1: Altera SoC SW Development Overview," Altera, Austin, USA, 2016.
- [22] M. L. Jangir, "OpenSource," 2012. [Online]. Available: <https://opensourceforu.com/2012/01/working-with-mtd-devices/>.
- [23] Altera, "Embedded Peripheral IP User Guide," San Jose, CA, 2016.
- [24] Altera, "On-Chip FIFO Memory Core," Altera, San Jose, CA, 2009.

[25] J. & R. A. & K.-H. G. Corbet, Linux Device Drivers, 3.edition, Sebastopol, CA:
O'Reilly Media, Inc., 2005.