

Building Xilinx ZynqMP System

ELECTGON
www.electgon.com
contact@electgon.com

25.04.2024



Contents

1	ZynqMP Description	1
1.1	APU	3
1.2	RPU	3
1.3	Programmable Logic	3
1.3.1	Connections	3
1.3.1.1	AXI Interfaces	4
1.3.1.2	Interrupts Interfaces	4
1.3.1.3	Clocks	4
1.3.1.4	EMIO	4
1.3.1.5	Dedicated Streams	4
1.3.1.6	PMU	5
1.3.1.7	DMA	5
1.3.2	Logic	5
1.4	Memory	5
1.4.1	Memory Components	5
1.4.1.1	On-Chip Memory (OCM)	5
1.4.1.2	Tightly-Coupled-Memory (TCM)	6
1.4.1.3	PL Memories	6
1.4.2	PS Memory Controllers	6
1.4.2.1	General Purpose DMA Controllers	7
1.4.2.2	Peripheral DMA	7
1.4.3	External Memory Interfaces	7
1.5	Peripherals	7
1.6	Interconnects	9
1.7	Interrupts	9
1.8	Bootng Process	9
1.8.1	Non-Secure Bootng	10
1.8.2	Secure Bootng	10
1.8.3	Bootngs Devices	10
1.9	Isolation and Protection	10
1.9.1	Trustzone	11
1.9.2	Exception Levels	11
1.10	Power Management	11

1.10.1	Power Domains	11
2	Hardware	12
2.1	Processor Configuration	13
2.2	HDF and Bitstream Generation	14
3	Firmware	15
3.1	Preparations	15
3.1.1	Cross Compiler	15
3.2	First Stage Boot Loader	15
3.2.1	Input Files	15
3.2.2	Output Files	15
3.2.3	Tools	16
3.2.4	Procedure Using HSI	16
3.2.5	Procedure Using XSCT	16
3.2.6	Procedure Using SDK	16
3.3	PMU	19
3.3.1	PMU ROM	20
3.3.1.1	Functions at Power On	20
3.3.2	PMU RAM	20
3.3.3	Input Files	20
3.3.4	Output Files	20
3.3.5	Procedure Using XSCT	20
3.3.6	Procedure Using SDK	20
3.4	ATF	21
3.4.1	Procedure in Linux	22
3.4.2	Procedure in Windows	22
3.4.3	Procedure Using SDK (GUI)	23
3.5	Device Tree	25
3.5.1	Device Tree Important Parameters	27
3.5.1.1	Label and Name	27
3.5.1.2	Binding	27
3.5.1.3	Addressing	27
3.5.1.4	Interrupt	28
3.5.1.5	Custom Arbitrary Parameters	29
3.5.1.6	Overwriting	29
3.5.2	DTS Organization in Xilinx	30
3.5.3	Other User DTS Configuration	33
3.5.4	Using SDK to build DTS file	33
3.5.4.1	Procedure Using SDK GUI	33
3.5.4.2	Procedure Using XSCT Command Line	36
3.5.5	Generating DTB Binary	36
3.6	UBoot	36

3.6.1	Quick Build	37
3.6.2	U-Boot Configuration	37
3.6.2.1	System Configuration	38
3.6.2.2	Device Tree Configuration	38
3.6.2.3	Board Configuration	38
3.6.3	Debugging U-boot Settings	39
3.6.4	Adding Customized Configurations	40
3.6.5	Bootting from Correct SD Partition	40
3.6.6	Bootting from TFTP Server	42
3.6.7	Bootting From NFS	42
3.6.8	Scripting Boot Procedure	43
3.6.8.1	Boot Script Sample	43
3.6.8.2	Boot Script Format	43
3.6.8.3	Boot Script Loading	44
3.7	FileSystem	45
3.7.1	Definition	45
3.7.2	Building File System	45
3.7.2.1	Build Using Busybox	46
3.7.2.2	Build Using Buildroot	46
3.7.3	Buildroot Quick Guides	48
3.7.3.1	Add Overlays	48
3.7.3.2	Modify Using Post Script	48
3.7.3.3	Modify Using Patches	49
3.7.3.4	Rebuild Packages	50
3.7.3.5	Get Package from User Specific Location	51
3.7.3.6	User Control During Build Process	51
3.7.3.7	Format of Zipped Output	52
3.7.4	File System Location	52
3.8	Linux Kernel	53
3.8.1	Kernel Image Forms	53
3.8.2	Flattened Image Tree	54
3.9	Boot Image	57
3.9.1	Input Files	57
3.9.2	Output Files	57
3.9.3	Tools	57
3.9.4	Procedure Using Command Line	57
3.9.4.1	Create BIF file	57
3.9.5	Procedure Using SDK	58
3.10	Boot Media	60
3.10.1	Boot Through TFTP Server	61
3.10.2	Boot Through SD Card	61
3.10.2.1	SD Card Partitioning	61

3.10.3	Bootting Through QSPI	62
3.10.3.1	QSPI Partitioning	63
3.10.3.2	Writing on QSPI	64
3.10.4	Bootting During Development Phases	66
3.10.4.1	Replacing Kernel, DTB, Filesystem Images	66
3.10.4.2	Replacing Boot File	66
3.10.5	Loading Bitstream File	66
4	Software	68
4.1	Linux Drivers	68
4.1.1	Build and Run Linux Kernel	68
4.1.2	Create Module	69
4.1.3	Build Module	69
4.1.4	Running Module	70
4.2	Linux Drivers Chain	70
4.2.1	Platform Hardware Link	71
4.2.2	Kernel Platform Link	73
4.2.3	Major Number	74
4.2.4	User Driver	75
4.3	Driver Integration	77
4.3.1	Integration with Kernel Image	77
4.4	GPIO Driver	78
4.4.1	Using sysfs	78
4.4.2	Using GPIOLIB	81
4.4.2.1	Prepare Device Tree file	81
4.4.2.2	Recognize GPIO Pins in the Platform Driver	81
4.4.2.3	Writing to the GPIO	82
4.4.3	Using Generic UIO	82
4.4.3.1	Define Driver in Hardware Node	82
4.4.3.2	Enable UIO Driver in Kernel Configuration	82
4.4.3.3	Set Driver Identification in Kernel	83
4.4.3.4	Writing to UIO	83

Abstract

This book is a practical guide for building an integrated system for Xilinx FPGA - Zynq family. What is meant by integrated system is the complete flow starting from the hardware logic up to a running application on an Operating System. Zynq device from Xilinx is an FPGA that consists of a Programmable Logic part (PL) and a built-in Processing System (PS). This means that this FPGA can be used to build a customized hardware that can be attached to the Processing System as a peripheral. Controlling the customized hardware is done then through an Operating System which is usually a Linux build that can be obtained from a Xilinx repository. Therefore this book is showing important steps needed to build both the hardware and Operating System stuff.

Chapter 1

ZynqMP Description

Understanding the target hardware platform shall be first step to be achieved in order to build needed binaries and bitstream easily as well as to know capabilities of this platform for best use. This book is discussing build procedures based on Xilinx ZynqMP Ultrascale+. Xilinx has already very beneficial documentations and support platform that makes explaining the ZynqMP here is not an added-value. A summarized overview of this platform can be sufficient as a starting point but for real explanations it is advised always to refer to available Xilinx User Guides and tutorials.

ZynqMP is partitioned basically into Processing System (PS) and Programmable Logic (PL). PS is a hard-wired ARM processors that are implemented into the chip, PL is the configurable hardware part of the FPGA. The chip can be classified also in terms of power domain consumed by its components. So it can be classified into

- Full Power Domain (FPD).
- Low Power Domain (LPD).
- PL Power Domain (PPD).
- Battery Power Domain (BPD).

with a component called Platform Management Unit (PMU), power management and isolation between these domains are controlled. Figure 1.1 shows other main components of ZynqMP.

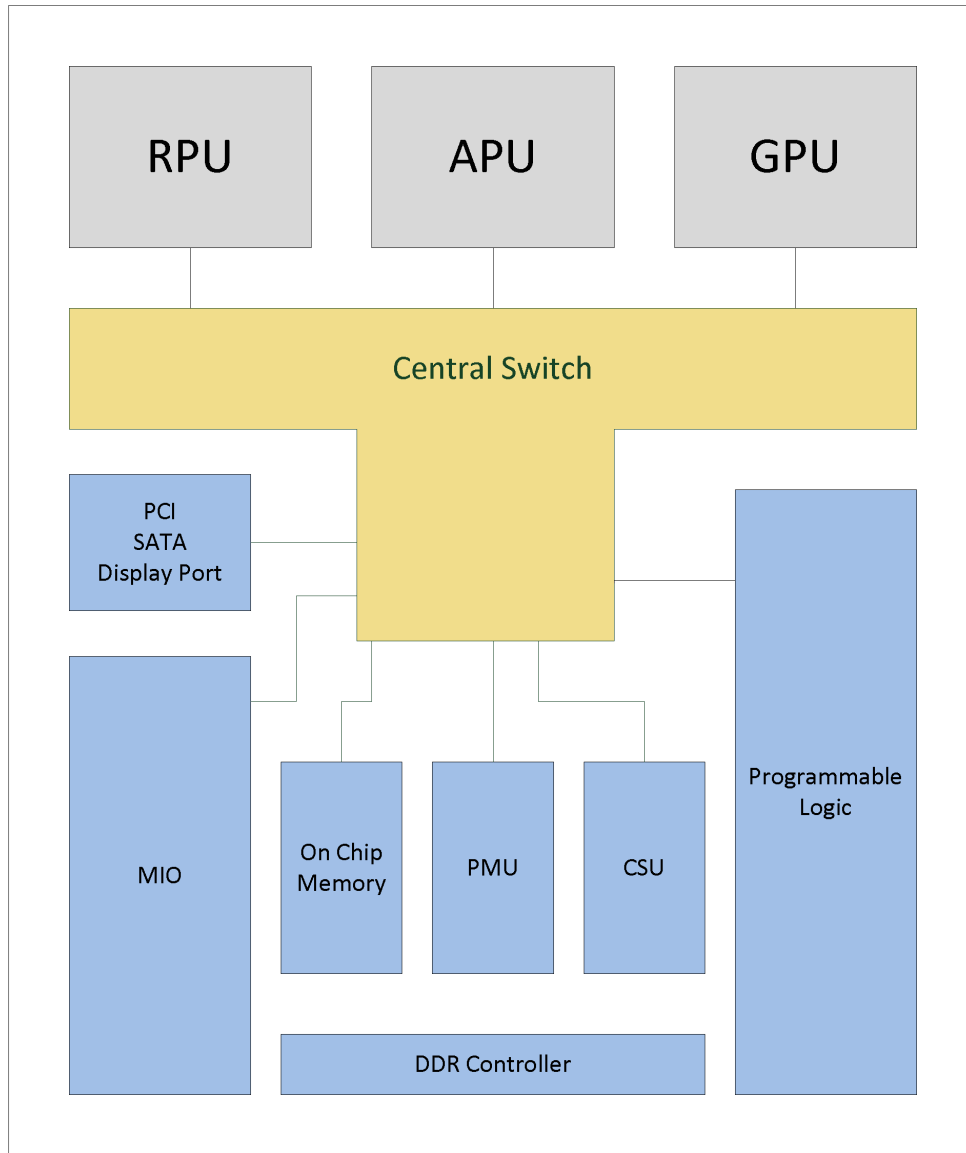


Figure 1.1: Power Domains in ZynqMP

Or it can be summed up according to the power domains as in table 1.1

	FPD	LPD	PPD	BPD
Processing	- APU - GPU	- RPU - PMU - CSU	-	-
Peripheral/Accelerator	- PCI / SATA /Display Port	- OCM - MIO Peripherals	- PL - PL Peripherals	-

Table 1.1: Main Components in Power Domains

Table 1.1 includes also an indication if the component is a processing system or a peripheral that serves a special function. This means that some components have running processors inside. The fact is each processing system can run independently. That is called they are Asymmetric processing systems. For example APU has four cores, they can run with different Operating Systems and they don't share workload. RPU has 2 cores they

can run independently. So the user has the option to choose which processor to work with (APU, RPU, etc) and how to employ cores of the target processor.

1.1 APU

Application Processing Unit consists of mainly a Quad-Core A53 ARM processor. Some chips have only a Dual-Core. A53 processor. This A53 processor is based on ARMv8-A architecture, 64 bit., can operate up to 1.5 GHz. It has independent Memory Management Unit (MMU) and dedicated L1 cache.

The four cores of A53 can run symmetrically (they can share workload) and also they can run independently (Asymmetric MultiProcessing); in this later case they can be unsupervised (no arbiter between AMP blocks) or Supervised (there is a hypervisor coordinating AMP blocks).

1.2 RPU

Realtime Processing Unit to be used for realtime applications. It consists of mainly a Dual-Core ARM R5 processor which is built on ARMv7 architecture, 32 bit, can operate up to 600 MHz. It has also 128 KB of tightly coupled Memory and a dedicated L1 cache.

The cortex-R5 can operate in two different modes: split mode or lock-step mode.

Step mode:

- This is the default mode.
- each core operates independently.
- one core may be running OS while the other is running bare-metal or both are running different OS.

Lock-Step mode:

- The dual core will act as single CPU.
 - Both cores are running same instructions in parallel delayed by 1.5 cycle.
-

1.3 Programmable Logic

1.3.1 Connections

There are several connections to/from the PL

1.3.1.1 AXI Interfaces

	Bi-directional	From PL	To PL
Coherent	S_AXI_ACE_FPD	S_AXI_HPC0_FPD S_AXI_HPC1_FPD	—
Belongs to LPD	—	S_AXI_LPD	M_AXI_HPM0_LPD
Performance	—	S_AXI_ACP_FPD	—
General Purpose, non-coherent	—	S_AXI_HP0_FPD S_AXI_HP1_FPD S_AXI_HP2_FPD S_AXI_HP3_FPD	M_AXI_HPM0_FPD M_AXI_HPM0_FPD

Table 1.2: PL AXI Interfaces

1.3.1.2 Interrupts Interfaces

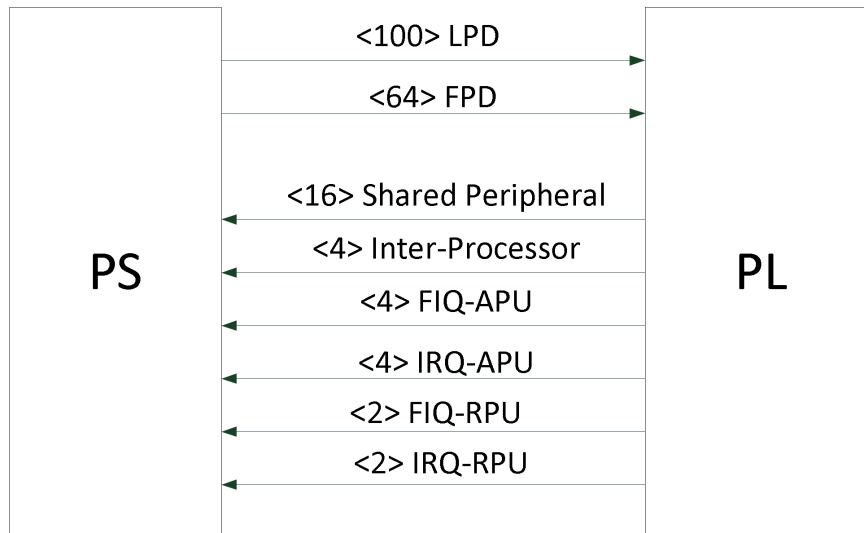


Figure 1.2: Interrupts Interfaces

1.3.1.3 Clocks

There are four clock signals that can be provided from PS to PL.

1.3.1.4 EMIO

ZynqMP has ready GPIO peripherals that can be configured by the user. These GPIO peripherals can be routed directly outside the chip through MIO or it can be routed into the PL part through EMIO interface.

1.3.1.5 Dedicated Streams

The PL part has also interface to high speed modules like Gigabit Ethernet and Display port peripherals.

1.3.1.6 PMU

PL can communicate with the PMU, there are 32 bit signals for data exchange in addition to some other control signals.

1.3.1.7 DMA

Interfaces to the Memory units is also available to the PL part.

1.3.2 Logic

- Any implemented PL logic should have AXI interface to communicate with the PS for data transfer.
 - If there are more than one logic, the designer should instantiate AXI Interconnect IP to allow multiple masters to talk to multiple slaves.
 - The PL includes integrated blocks for PCI Express, Interlaken, 100 G-Ethernet, System Monitor and video codec unit.
 - PL part is configured by the FSBL at startup.
 - It is possible to partially load the bitstream file. That is it, FSBL can load part of the bitstream and the rest can be loaded dynamically while the initial loaded logic continues to operate without interruption. To accomplish this, you must implement multiple configurations that result in full bitstream. This procedure is called Partial Reconfiguration and is discussed in Xilinx UserGuide UG947.
-

1.4 Memory

Zynq Ultrascale+MPSoC devices include:

1. Memory components
2. Memory access controllers
3. External memory interfaces

1.4.1 Memory Components

1.4.1.1 On-Chip Memory (OCM)

- 256 KB.
 - has interfaces to PS and PL as well.
 - can't be accessed through DDR Controller.
 - FSBL loads the code for booting the PS into the OCM.
 - used to program non volatile flash memory through JTAG boot mode.
-

1.4.1.2 Tightly-Coupled-Memory (TCM)

- Included in cortex-R5 (each core of R5 has TCM).
- 128 KB each.
- each TCM is formed by 2 banks ATCM and BTCM. So in total there are 4 banks in RPU.
- each of ATCM or BTCM is 64 KB - 64 bit wide.
- when RPU is operating in split mode, each RPU core can access two 64KB TCM memory.
- when RPU is operating in Lock-step mode, the RPU can access a block of 256 KB TCM memory.
- TCM is also mapped into the global system memory so that it can be accessed by the APU or PL part. It can be accessed then as two blocks of 128 KB or one block of 256 KB. The APU access it directly without memory arbitrators.

1.4.1.3 PL Memories

PL has three types of memory blocks which are faster and consume less power: BRAM - UltraRAM - LUTRAM.

BRAM	UltraRAM	LUTRAM
<ul style="list-style-type: none"> • These are traditional FPGA RAM blocks. • each block is 36 KB. • each block can be used as 2x18 KB or 1x36 KB (both can be used as double or single port). • ZynqMP has in total 35 MB. 	<ul style="list-style-type: none"> • These are additional in ZynqMP. • each block is 288 KB. • ZynqMP has in total 128 MB 	<ul style="list-style-type: none"> • aka Distributed RAM. • it is built out of LUT primitives.

1.4.2 PS Memory Controllers

ZynqMP has two general purpose DMA controllers and several peripheral specific DMA engines.

1.4.2.1 General Purpose DMA Controllers

- supports memory to memory, memory to I/O, I/O to memory, I/O to I/O transfers.
- one is located in FPD, the other is located in LPD.
- both manage 8 independent DMA channels.
- The FPD-DMA is connected to 128 bit AXI bus, uses 4K buffer. The LPD-DMA is connected to 64 bit AXI bus, uses 2K buffer.

1.4.2.2 Peripheral DMA

- High speed peripherals on ZynqMP (USB 3.0 - PCI Express - Gigabit Ethernet - SATA - SDIO - Display Port - QSPI) come with their own DMA controllers.

1.4.3 External Memory Interfaces

- Nevertheless, external memory can be connected or attached to ZynqMP efficiently. ZynqMP provides IP block called Memory Interface Generator (MIG) to handle attached external memory interfaces (DDR, QDR, RDRAM).
 - ZynqMP can support external DDR memory with capacity up to 32 GB.
 - The external memory will be connected to the system through AXI interconnect using 6 AXI interfaces.
 - The interface supports multiple memory standards (DDR3, DDR3L, LPDDR3, DDR4, LPDDR4, UDIMM, RDIMM).
-

1.5 Peripherals

As any system block of ZynqMP, peripherals are distributed over four power domains. Figure 1.3 locates each peripheral in its operation power domain along with the interface to which it has a way.

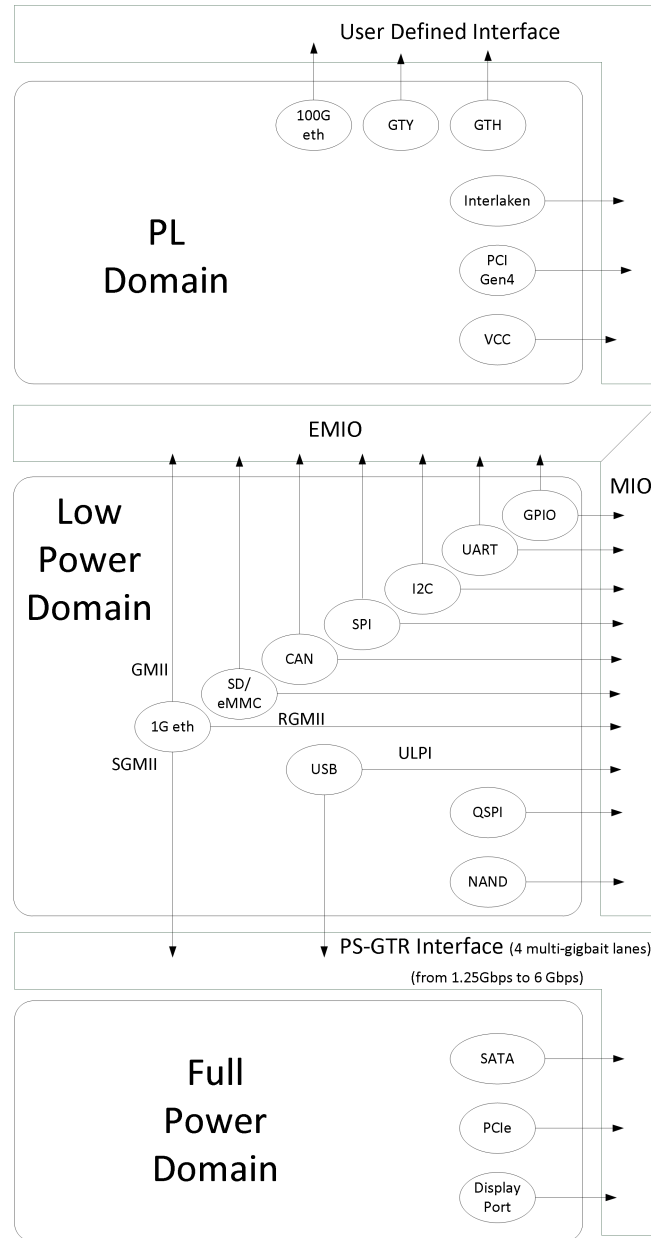


Figure 1.3: ZynqMP Peripherals

GPIO: Although 78 pins of MIO are shared between LPD peripherals, in case we don't use other peripherals, GPIO can control the entire 78 pins. It can also use pins of EMIO (96 pins as input, 192 pins as output).

SPI: SPI controllers don't support direct memory access (DMA), hence the communication is done through register reads and writes to the controller AXI interface.

UART: UART in ZynqMP devices supports a wide variety of parameters; programmable baud rate; 6 or 7 or 8 data bits; 1 or 1.5 or 2 stop bits; odd or even or space or mark or no parity.

UART can operate either in normal mode or 3 different loopback modes.

QSPI: The Quad SPI controller of ZynqMP includes two types of controllers: the Generic

QSPI and the legacy QSPI..

Ethernet: 4 Gigabit Ethernet Controllers. Implemented as MAC. Supporting 10/100/1000 Mb/s. Supporting GMII (through PL pins using EMIO), RGMII (through MIO), SGMII (through PS-GTR).

1.6 Interconnects

- It links together all of the processing blocks and enables them to interface with the outside world through access peripherals, devices and memory.
 - It is based on AXI.
 - AXI interface consists of five different channels: Read Address channel, Read Data channel, Write Address channel, Write Data channel, Write Response channel.
 - Because there are several components in Zynq, there almost no direct connection between Master and Slave of the AXI bus.
-

1.7 Interrupts

In Zynq there are two interrupt controllers

GIC-V1: It is implemented for RPU. Each cortex-r5 will have two interrupt lines (normal interrupt nIRQ, normal high priority nFIQ).

GIC-V2: This is same as GIC-V1 but enables interrupt virtualization and it is implemented for APU. Each cortex-a53 will have four interrupt lines (normal interrupt nIRQ, normal high priority nFIQ, virtual interrupt vIRQ, virtual high priority vFIQ).

Interrupt Sources:

- Software Generated Interrupts (SGI).
 - Private Peripheral Interrupts (PPI).
 - Shared Peripheral Interrupts (SPI).
 - Interprocessor Interrupts (IPI); In ZynqMP there are 11 IPI channels four, of them are reserved for communication with PMU.
-

1.8 Booting Process

Booting ZynqMP can be done in two modes

1.8.1 Non-Secure Booting

It can be declared as shown in figure1.4.

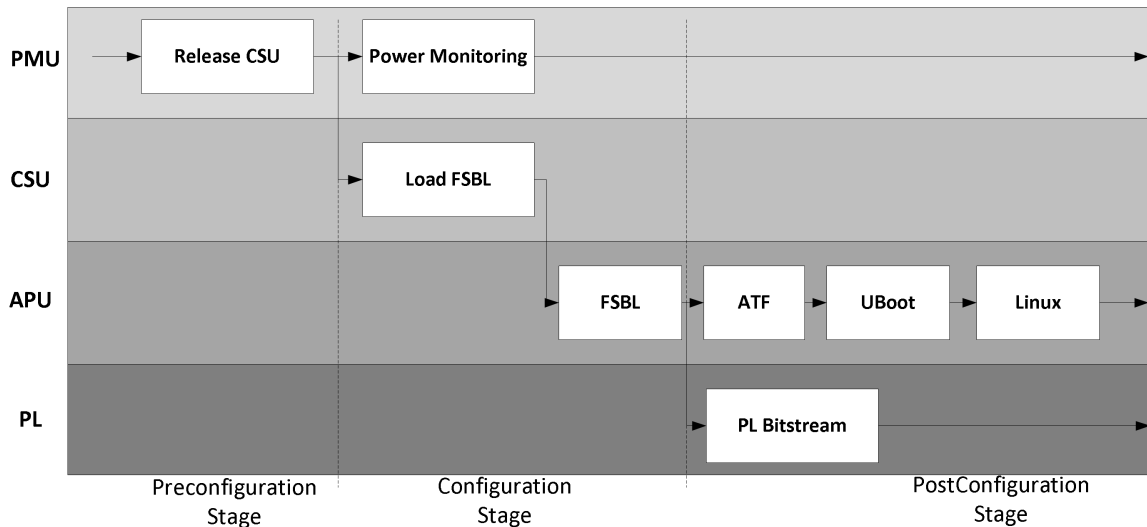


Figure 1.4: ZynqMP Boot Sequence

Firmware of the CSU is stored by the manufacturer into a ROM so it is not customizable. This firmware when it starts it loads FSBL firmware into OCM to be accessed by the APU (or RPU).

1.8.2 Secure Booting

It is same as Non-Secure booting but with addition that the loaded image is encrypted. In this case the CSU performs decryption and loads FSBL into OCM.

1.8.3 Bootings Devices

CSU supports booting from QSPI, NAND, SD, eMMC, JTAG. CSU doesn't support NOR, SATA, Ethernet, PCI Express.

1.9 Isolation and Protection

Xilinx ZynqMP provides some mechanisms to protect the device resource from tampering. These mechanisms are mainly filtering traffic passing to the resource based on its ID.

ZynqMP resources can be mainly described as: Processing Units, Memory Units, Peripherals Units. Therefore each of these units resources is protected by SMMU (System Memory Management Unit), XMPU(Xilinx Memory Protection Unit), XPPU (Xilinx Peripheral Protection Unit).

1.9.1 Trustzone

The APU provides further additional mechanism (Trustzone) for isolating and partitioning software running on it.

Trustzone mechanism makes partitioning of software to secure or non-secure. This partitioning is communicated across AXI interactions.

This mechanism adds more security level beneath OS level. Traditional processing systems give OS more privilege than the running software, which is known as user space and kernel space. Trustzone-capable ARM processor gives more privilege to this layer more than the OS. The OS still thinks as if it has the highest privilege.

1.9.2 Exception Levels

Trustzone as implemented on the APU (ARMv8-a53) defines four privilege levels or as it is called Exception Levels. Level 0 is lowest privilege, level 3 is the highest.

EL0: Applications SW are given this privilege.

EL1: OS are given this privilege (OS are called also supervisor).

EL2: Hypervisor is given this privilege (this is when running multicore processor).

EL3: ARM Trusted Firmware (ATF) is given this privilege as it is the security monitor.

1.10 Power Management

Five main techniques which we can use to tune system's power management on ZynqMP

1. Feature Disabling
2. Dynamic Power Management
3. Frequency Scaling
4. Clock Gating
5. Use of PL Acceleration

1.10.1 Power Domains

- Battery Power Mode: 180 nW ~ 3uW : used for realtime clock and battery backed RAM.
- Low Power Mode: 20 mW ~ 450 mW.
- Full Power Mode: 450 mW ~ 2.5 W.
- PL Power Mode: depends on implemented logic.

Chapter 2

Hardware

ZynqMP can't be used without its processing unit, i.e. developers can't utilize the PL part independently. That is because the PL is programmed through the FSBL firmware which is executed by the processing system. Therefore any hardware used in ZynqMP must be associated with a processing unit.

This leads us to conclude that Xilinx tools are needed in order to instantiate the processing unit into the hardware model, which means we have to use Vivado to build that and we have to make sure that we have appropriate license for the instantiated modules or IPs.

A typical system should look like what is shown in figure 2.1 which shows a simple system consists of the processing unit, AXI interconnect and a GPIO unit. The GPIO unit is a peripheral that is attached to the processing system through AXI interconnect instance, this is mandatory; i.e. we can't attach the peripheral directly to the processing system.

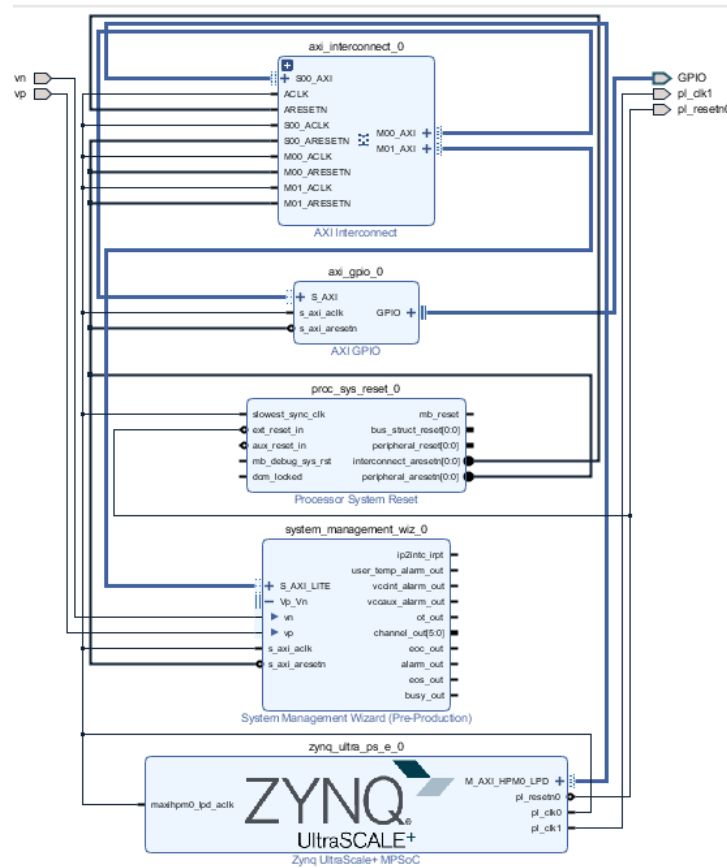


Figure 2.1: Sample Hardware Model [1]

In Vivado to build such a model, we have to create what is called Block Design. This Block Design contains important instances of the Hardware model such as the processing system and IP peripheral of Xilinx (GPIO, Gigabit transceivers, etc). It is important to note that, the Hardware model shall contain this block design at the second level. That means that the top design file shall contain a wrapper of a block design that contains ZynqMP processor. Without that, device tree model will not be generated for the design as putting the ZynqMP processor in any lower level will not allow to the processor to be seen or detected using SDK tool.

2.1 Processor Configuration

The processing system of ZynqMP can be configured according to the user needs. In Vivado GUI double click on the processing system instance in the block design. Figure 2.2 shows main processing system components, you can then choose which component to enable or to configure.

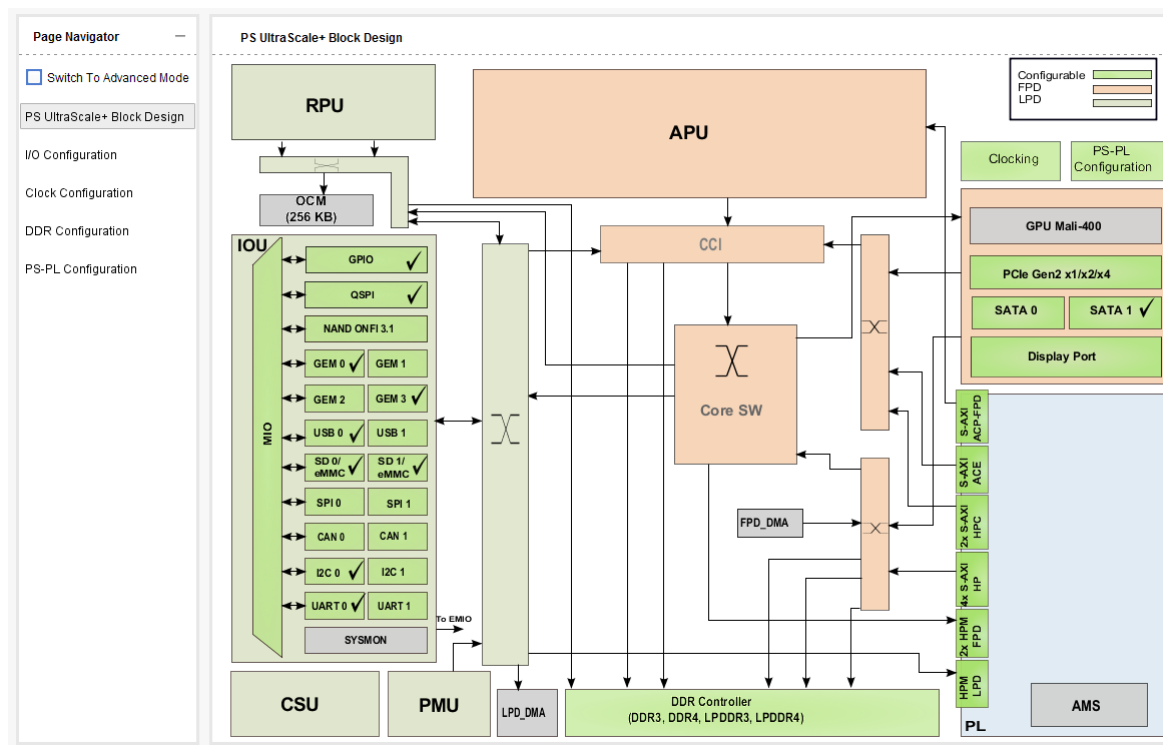


Figure 2.2: Processing System Configuration[1]

2.2 HDF and Bitstream Generation

After building the needed Hardware model, what is important to extract from it is to have HDF file and Bitstream File. HDF file is used to as a description of the Hardware and it is used in Software flow in order to generate Devicetree and other needed firmware. Bitstream file is optional to have and it is needed only if the hardware model is utilizing the PL part. To generate HDF file in Vivado GUI, choose from top menu File >> Export >> Export Hardware. Choose then where you want to store this HDF file.

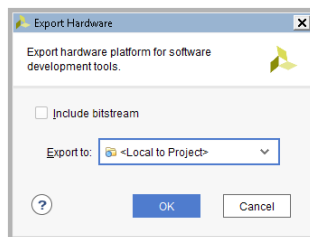


Figure 2.3: Export Hardware Window[1]

Alternatively, you can use the following Vivado TCL command.

```
write_hwdef -force -file path/to/preferred/filename.hdf
```

Generation of bitstream is similar to any PL flow; in Vivado GUI click on 'Generate Bitstream'.

Chapter 3

Firmware

3.1 Preparations

3.1.1 Cross Compiler

ZynqMP has ARM Cortex a53 and Cortex r5 processors. For applying any firmware or software that should run in these processors, we have to cross compile the firmware or software. By default, Xilinx tools (namely SDK) provides cross compiler toolchain for both processors. If you prefer to work with another cross compiler, it is fine.

Cross Compiler provided by SDK can be found in

“<INSTALLATION_DIR>/SDK/<version>/gnu/”

In this directory you can find cross compiler for ARM64 architecture (aarch64), and for R5 architecture (armr5). Inside aarch64, you can find cross compiler for Linux-based firmware (aarch64-linux) or for Bare-metal firmware (aarch64-none).

3.2 First Stage Boot Loader

After powering on ZynqMP (or any processing system in general), a built-in routine starts up and handles the booting process to next routine. In our case here, FSBL is first routine that initializes the ZynMP FPGA and make necessary setting. The following lines describes how to build FSBL for ZynqMP.

3.2.1 Input Files

hdf (or BMM) file, bit file is optional.

3.2.2 Output Files

elf file

3.2.3 Tools

SDK (GUI) or HSI (command line) or XSCT(command line).

3.2.4 Procedure Using HSI

HSI (Hardware Software Interface) is a framework by Xilinx used to perform some procedures needed for third party tools. Details about this tool is included in Xilinx UG1138. Some help can be displayed also by typing

```
$ hsi -h
```

Or for help about specific command you can type

```
$ hsi generate_app -help
```

For creating FSBL

```
$ hsi generate_app -hw $hwdsgn -os standalone -proc ps7_cortexa9_0 -app  
zynq_fsbl -compile -sw fsbl -dir <dir_for_new_app>
```

3.2.5 Procedure Using XSCT

XSCT is a Xilinx framework that is used to perform operations done in SDK but in command line mode. To generate an FSBL elf file the following routine can be used

```
setws .  
createhw -name hw0 -hwspec <<HW_HDF_FILE_HERE>>  
createapp -name fsbl1 -app {Zynq MP FSBL} -proc psu_cortexa53_0 -hwproject  
hw0 -os standalone -lang c -arch 64  
projects -build
```

you can get help about any of the above commands by typing

```
$ <command name> -help
```

3.2.6 Procedure Using SDK

Open SDK

In the top menu, File >> New >> Project

Choose from Xilinx >> Hardware Platform Specification

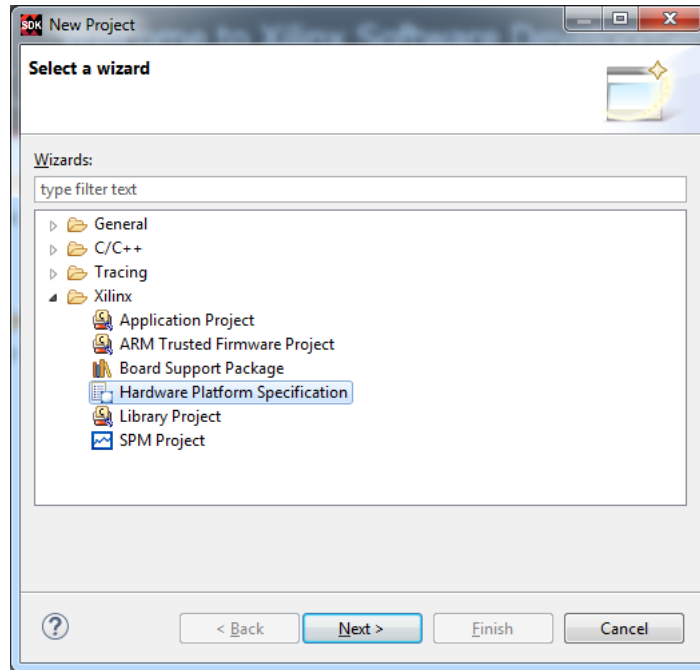


Figure 3.1: FSBL - Choose Hardware[1]

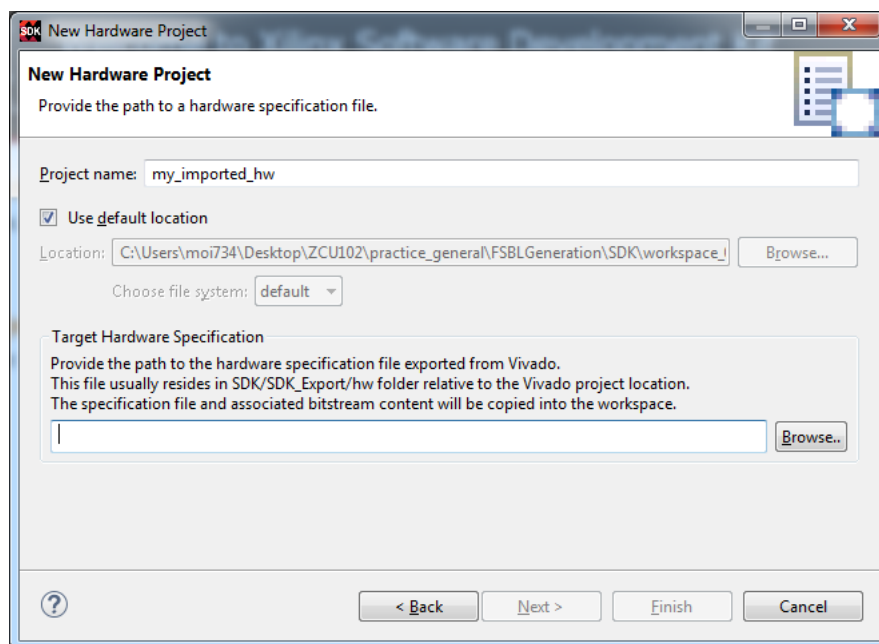


Figure 3.2: FSBL - Browse HDF File [1]

Browse to the generated Hardware (hdf file) then click Finish.

In the top menu, File >> New >> Application Project

Choose the shown options and make sure that Hardware Platform is what you have built previously.

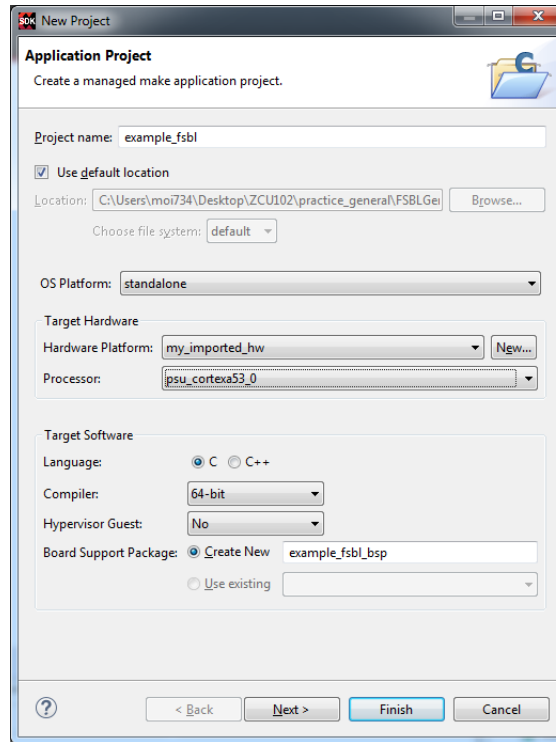


Figure 3.3: FSBL Settings [1]

Click Next, choose to create Zynq MP FSBL

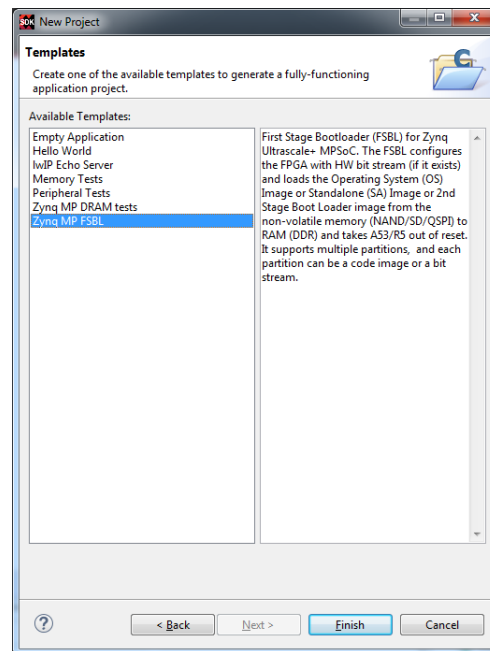


Figure 3.4: FSBL Template[1]

Click Finish to generate the FSBL

Right Click on the created project to 'Build Project'

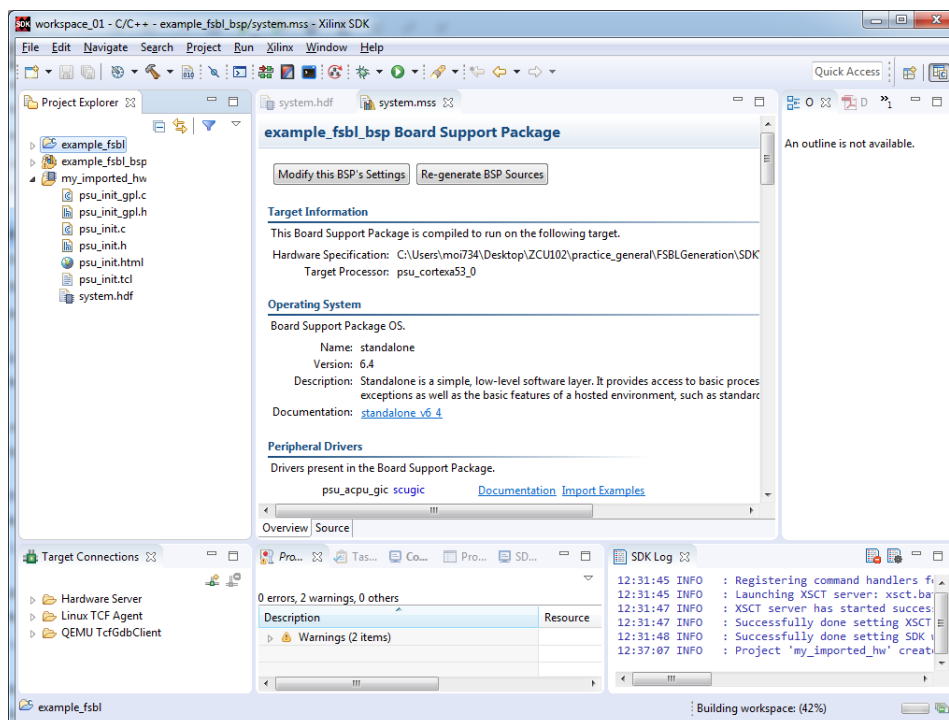


Figure 3.5: FSBL Board Support Package [1]

Generated FSBL elf file can be found under created FSBL project >> Debug >> example_fsbl.elf

3.3 PMU

Platform Management Unit is a special unit in the Zynq MP used to power up the system and power management (switching PS between different power modes). It also supports Inter-Process Interrupts (IPI) for communication between the system processors (to do some power actions).

This PMU has MicroBlaze processor which is connected to 128 kilobytes of RAM with error-correcting code (ECC) that is used for data and firmware as well as storage of the Xilinx provided framework code.

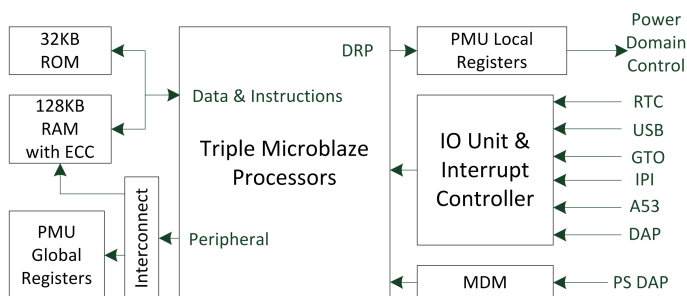


Figure 3.6: PMU Block Diagram

3.3.1 PMU ROM

Inside the ROM resides firmware that makes specific functions like:

3.3.1.1 Functions at Power On

After power on, the PMU performs the following sequence of events before handing off to the Configuration Setup Unit (CSU):

- Provides power integrity check using the system monitor (SysMon) assuring proper operation of the CSU and the rest of the LP domain.
- Initializes the PLLs
- Triggers and runs the Memory Built in Self Test (MBIST)
- Captures and signals errors which can be read through JTAG
- Powers down any Power Islands and other IP disabled via eFuse
- Releases Reset to CSU

3.3.2 PMU RAM

Inside the RAM resides Error Correction Code (ECC) and optional user firmware that can be tailored to manage and control power of the chip. This firmware can be generated using Xilinx tools (e.g. SDK) as it has ready template for that firmware.

3.3.3 Input Files

hdf (or BMM) file, bit file is optional.

3.3.4 Output Files

elf file

3.3.5 Procedure Using XSCT

You can use these commands to generate the elf file

```
createhw -name hw0 -hwspec $HDF_Source
createapp -name generated_pmu -app {ZynqMP PMU Firmware} -proc psu_pmu_0
        -hwproject hw0 -os standalone -lang c -arch 64
projects -build
closehw hw0
```

3.3.6 Procedure Using SDK

Same as FSBL generation except for processor type choose psu_pmu_0

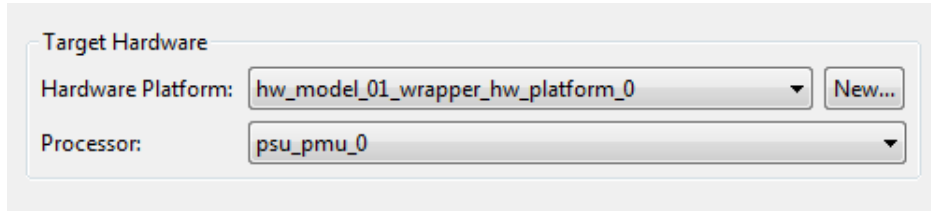


Figure 3.7: PMU Build Setting

3.4 ATF

ATF is responsible for preserving and restoring the non-secure context when switching to the secure context. It's also responsible for part of the power-management since the PMU, which is responsible for power management, is a secure AXI slave and will therefore not accept any commands issued by a hypervisor or non-secure OS running on the APU. Power-management requests made by non-secure OSes, such as Linux, are therefore provided through the ATF to interact with the PMU.

Xilinx software stacks running on the Zynq US+ MPSoC APU conform to the standard ARMv8 topology where Linux running at ARM EL1/0 has hardware-limited access to system or security-critical registers or devices. All interactions from Linux to those devices or registers are routed through ARM Trusted Firmware which runs at EL3.

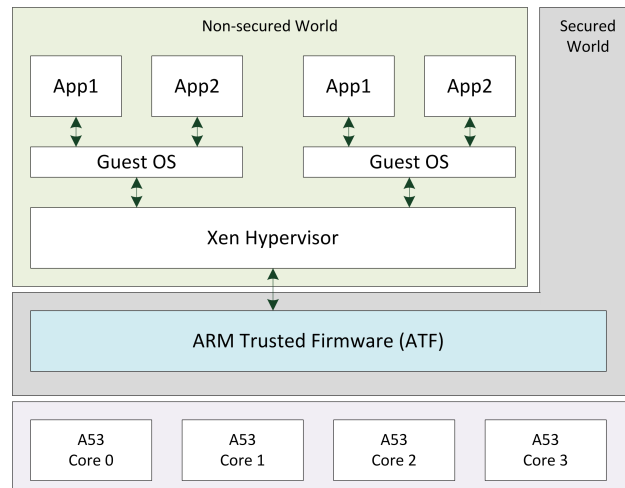


Figure 3.8: APU Software Stack

ARM Trusted Firmware provides a reference to secure software for ARMv8-A architecture and it provides implementations of various interface standards like PSCI(Power State Coordination Interface) and Secure monitor code for interfacing to Normal world software. Xilinx ARM trusted firmware is based on arm trusted firmware. Xilinx ARM Trusted Firmware implements the EL3 firmware layer for Xilinx Zynq UltraScale + MP-SoC. The platform only uses the runtime part of ATF(EL3 firmware) as ZynqMP already has a BootROM (BL1) and FSBL (BL2). [4]

3.4.1 Procedure in Linux

To build ATF in Linux, the following is needed

- Linux machine with the following packages are installed

```
$ sudo apt-get install build-essential gcc make git libssl-dev
```

- Cross Compiler tool: this can be obtained from Xilinx tools installation directory
Xilinx>>SDK>>2017.3>>gnu>>aarch64>>nt>>aarch64-linux

Note that if you want to build the ATF for another 32bit ARM, use cross compiler for that architecture (gnu>>aarch32>>...).

- Download ATF source code from:

```
$ git clone https://github.com/Xilinx/arm-trusted-firmware.git
```

or it can be downloaded from here (for the case of windows without git)

<https://github.com/Xilinx/arm-trusted-firmware/releases/>

- Define path to the cross compiler tool

```
$ export CROSS_COMPILE=<path-to-aarch64-gcc>/bin/aarch64-linux-gnu-
```

- Build the ATF by

```
$ make PLAT=zynqmp RESET_T0_BL31=1
```

After the build process completes the bl31.elf binary is created within the /build/zynqmp/release/bl31 directory.

3.4.2 Procedure in Windows

For running it in command line, XSCT shall be used. The following script can be used

```
set ATF_BuildDir ./workspace_02
set ATF_Source C:/Xilinx/SDK/2017.3/data/embeddedsw/lib/sw_apps/atf_sources
set MakeDir $ATF_BuildDir/atf_sources/src
set Script_Dir [pwd]

setws -switch $Script_Dir
after 1000
exec mkdir -p $ATF_BuildDir
file delete -force -- $ATF_BuildDir

exec mkdir -p $ATF_BuildDir
cd $ATF_BuildDir

setws -switch .
after 1000

exec cp -r $ATF_Source .

#_###NOTE: It is recommended to run the make step manually
#_###:so take the following line copy-paste in command window
#_#####
```

```
#exec -ignorestderr make -C $MakeDir CROSS_COMPILE=aarch64-none-elf-  
RESET_TO_BL31=1 PLAT=zynqmp bl31
```

3.4.3 Procedure Using SDK (GUI)

Open SDK

File>>New>>Project

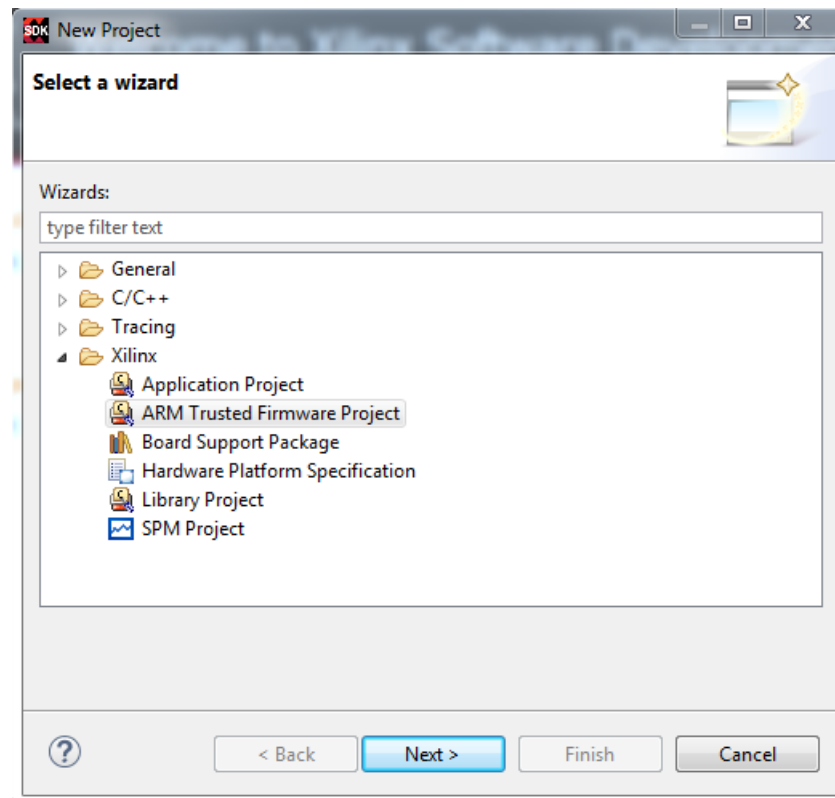


Figure 3.9: ATF Start Project [1]

Choose ARM Trusted Firmware Project then Click Next

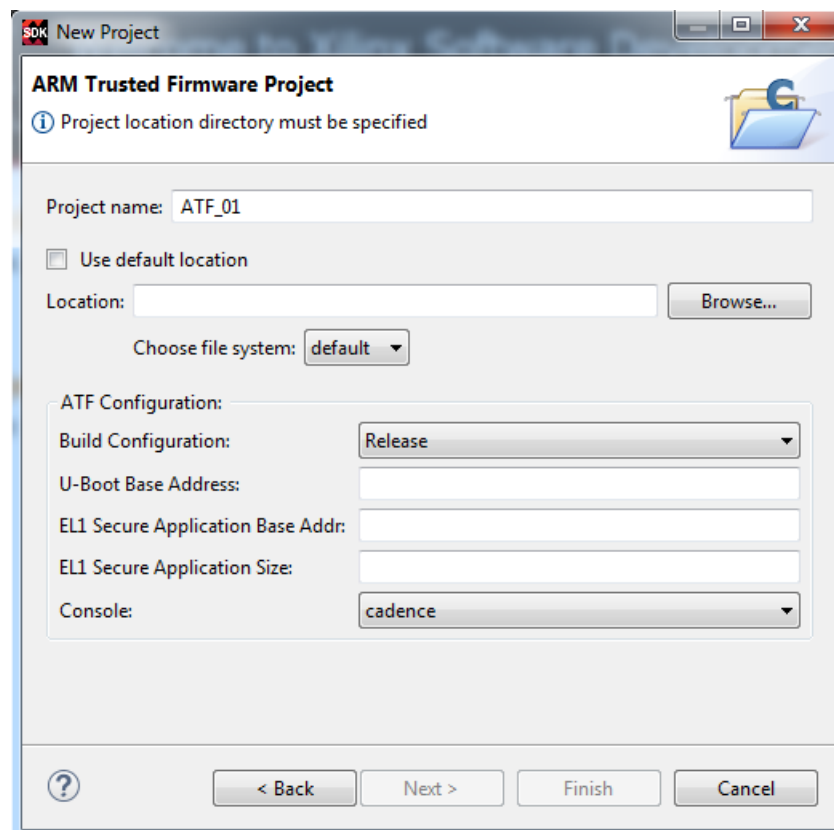


Figure 3.10: ATF Project Settings [1]

Provide Project Name and Location. Choose Build Configuration field as Release. Hit finish.

Output file will be built as shown below in the highlighted path

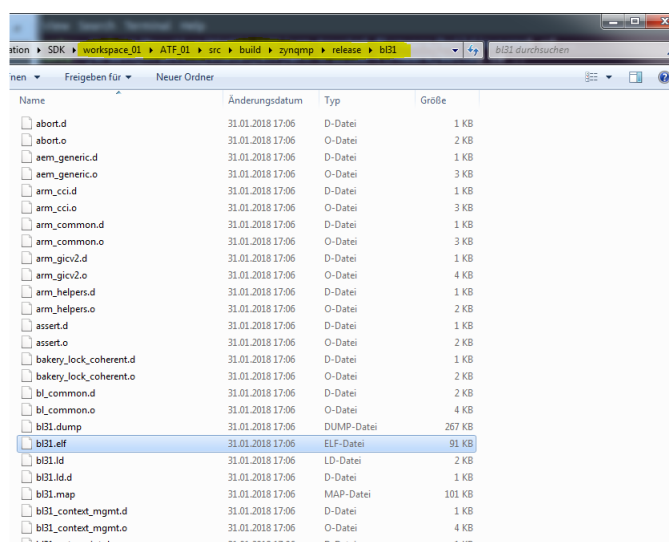


Figure 3.11: ATF Project Output [1]

Note that ATF source files are downloaded by default in SDK in the following path

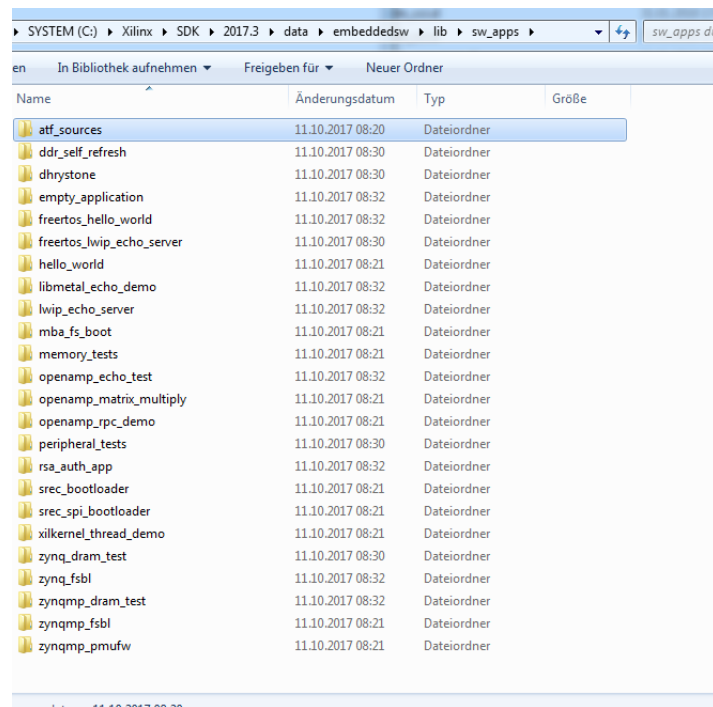


Figure 3.12: ATF Source Path [1]

3.5 Device Tree

In order for the Operating System to recognize what hardware or devices are built and attached to the processing system, a description of these devices shall be provided to the Operating System or specifically the kernel. For that purpose, a dts file (device tree source) is used to list all the attached devices. This dts file is compiled into dtb file (device tree blob) which is a binary file to be loaded during booting sequence into ram and read by the kernel (The boot loader copies that chunk of data into a known address in the RAM before jumping to the kernel's entry point).

Assume you have built a hardware in the Xilinx Zynq with the processing system and simple GPIO controller in the FPGA part as shown in figure 3.13

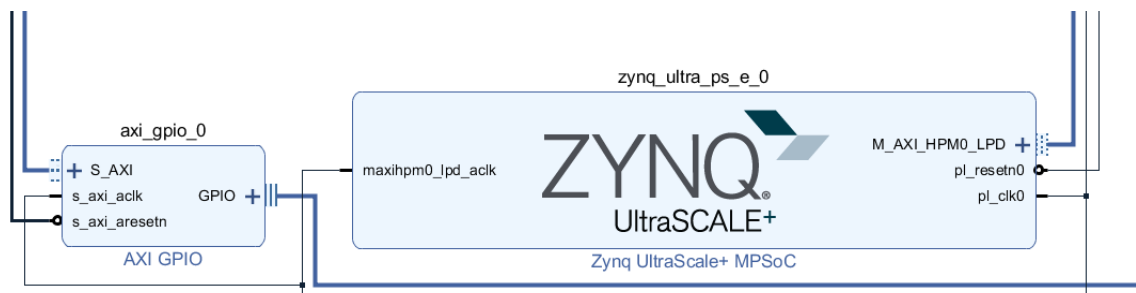


Figure 3.13: Simple Hardware in ZynqMP [1]

This GPIO controller is connected to the Zynq processor as a slave via AXI bus. To

build device tree for such a system Xilinx SDK tool can be used as discussed later. What can be shown here is the AXI bus is considered as a hardware node that is attached to the processor. The GPIO controller is considered as another hardware node that is attached to the AXI bus. So in the DTS file, the description of the GPIO controller will be inside description of the AXI bus. For the shown architecture, the DTS file should have

```

/ {
    amba_pl: amba_pl@0 {
        #address-cells = <2>;
        #size-cells = <2>;
        compatible = "simple-bus";
        ranges ;
        axi_gpio_0: gpio@80000000 {
            #gpio-cells = <2>;
            compatible = "xlnx,xps-gpio-1.00.a";
            gpio-controller ;
            reg = <0x0 0x80000000 0x0 0x1000>;
            xlnx,all-inputs = <0x1>;
            xlnx,all-inputs-2 = <0x0>;
            xlnx,all-outputs = <0x0>;
            xlnx,all-outputs-2 = <0x0>;
            xlnx,dout-default = <0x00000000>;
            xlnx,dout-default-2 = <0x00000000>;
            xlnx,gpio-width = <0x8>;
            xlnx,gpio2-width = <0x20>;
            xlnx,interrupt-present = <0x0>;
            xlnx,is-dual = <0x0>;
            xlnx,tri-default = <0xFFFFFFFF>;
            xlnx,tri-default-2 = <0xFFFFFFFF>;
        };
        psu_ctrl_ipi: PERIPHERAL@ff380000 {
            compatible = "xlnx,PERIPHERAL-1.0";
            reg = <0x0 0xff380000 0x0 0x80000>;
        };
        psu_message_buffers: PERIPHERAL@ff990000 {
            compatible = "xlnx,PERIPHERAL-1.0";
            reg = <0x0 0xff990000 0x0 0x10000>;
        };
    };
};

```

Figure 3.14: Device Tree Description

As you can see then, a DTS file contains tree structure of hardware nodes. Each node has some parameters and properties. A basic format of this structure can be considered as

```

/dts-v1/;

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        // hex is implied in byte arrays. no '0x' prefix is required
        a-byte-data-property = [01 23 34 56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
    };
};

```



```

        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};

```

Source : [2]

3.5.1 Device Tree Important Parameters

Back to the generated DTS of the GPIO controller, we need to get important parameters that a hardware node should contain.

3.5.1.1 Label and Name

In the gpio controller example, the label and name is *axi_gpio_0: gpio@80000000*. The label is an optional field. The name shall be in the shown format *<DeviceName>@<PhysicalAddress>*. Device name must not be unique, i.e. many devices can use the same name.

3.5.1.2 Binding

First and most important parameter is the “compatible” parameter. It should carry a unique name. This unique name will be used with the device SW driver in order to have a link between the SW driver and the HW device. When Linux kernel starts, it is interrogating the device tree file and kick off the corresponding kernel module according to the value specified in the parameter “compatible”. The developer is free to choose any name or string for his device but it should be the same name also used in the SW driver. As a convention, most vendors are defining their devices as in this form

“<manufacturer>, <model>”

So in our example here it is “xlnx,xps-gpio-1.00.a”. This name shall be used when building SW driver for the gpio controller. This driver will define it as follows in the driver source file

```

static struct of_device_id xillybus_of_match[] __devinitdata = {
    { .compatible = "xlnx,xillybus-1.00.a", },
    {}
};
MODULE_DEVICE_TABLE(of, xillybus_of_match);

```

More details about building the driver is discussed in SW drivers section later.

3.5.1.3 Addressing

In the attached node ‘reg’ parameter contain the assigned address of the hardware node. In its simplest form it should be like

```
reg = <0x80001000 0x1000>
```

which means that the device is assigned on physical address 0x80001000 and allocated with the size 0x1000 (4K). It can also have two addresses

```
reg = <0x101f3000 0x1000
      0x101f4000 0x0010>;
```

The parent node (amba_pl@0 in our example), will contain information about format of the 'reg' parameter. This is defined by #address-cells and #size-cells parameters. In the simplest form it can be defined as

```
#address-cells = <1>;
#size-cells = <1>;
```

In this case, it indicates that the child nodes will have 1 cell for the address and 1 cell for the size. Each cell is represented by 32 bit integer. So definition like reg = <0x80001000 0x1000> will be the right form.

In our example of GPIO controller address-cells and size cells are defined with value 2 both. This is because this system is built for 64 bit processor (Armv8). So the addressing of the GPIO controller was assigned as reg = <0x0 0x80000000 0x0 0x1000>; with 0x0 0x80000000 is considered a 64 bit address, 0x0 0x1000 is 64 bit size.

3.5.1.4 Interrupt

Interrupt is a signal between two devices to indicate special activity has taken place. In the DTS file, the device that receive interrupts is marked with the statement "interrupt-controller;" as shown in figure 3.15 taken from zynqmp.dtsi file

```
gic: interrupt-controller@f9010000 {
    compatible = "arm,gic-400", "arm,cortex-a15-gic";
    #interrupt-cells = <3>;
    reg = <0x0 0xf9010000 0x10000>,
        <0x0 0xf9020000 0x20000>,
        <0x0 0xf9040000 0x20000>,
        <0x0 0xf9060000 0x20000>;
    interrupt-controller;
    interrupt-parent = <&gic>;
    interrupts = <1 9 0xf04>;
};
```

Figure 3.15: Interrupt Parameter

Four properties are used to describe interrupt connections:

- Interrupt-controller: as shown in figure 3.15 this is the device that receives interrupt signal.
- Interrupt-parent: this parameter is declared at the device which sends interrupt signal to declare to whom this signal is sent. See figure 3.15 for example.
- #interrupt-cells: this is declared at the interrupt controller to specify how many values are used to describe the interrupts parameter.

- Interrupts: this is the parameter that describes when the interrupt is happening.

```
axi_timer_0: timer@80000000 {
    clock-frequency = <99990000>;
    clocks = <&misc_clk_0>;
    compatible = "xlnx,xps-timer-1.00.a";
    interrupt-parent = <&gic>;
    interrupts = <0 89 4>;
    reg = <0x0 0x80000000 0x0 0x10000>;
    xlnx,count-width = <0x20>;
    xlnx,gen0-assert = <0x1>;
    xlnx,gen1-assert = <0x1>;
    xlnx,one-timer-only = <0x1>;
    xlnx,trig0-assert = <0x1>;
    xlnx,trig1-assert = <0x1>;
};
```

Figure 3.16: Interrupt Source

In figure 3.16 the interrupt is described with the numbers `<0 89 4>`.

Xilinx is using 3 cells for describing the interrupt. In our example here first cell is 0 which means it is non SPI interrupt. Second cell is 89 is like ID for the interrupt. This ID can be found in Xilinx Zynq technical reference manual (UG 1085). In our example here the ID is mapped to IRQ number (89+32) to be at the end 121. From UG 1085 it is interrupt signal sent from PL to PS. Third cell is 4 which represents type of interrupt, it has three possible values

- 0 — Leave it as it was (power-up default or what the bootloader set it to, if it did)
- 1 — Rising edge
- 4 — Level sensitive, active high

3.5.1.5 Custom Arbitrary Parameters

Because each device is mainly different in functionality than other devices, developers can write their own custom information in the device tree. For example in the gpio controller it has parameter to define width of the output pins `"xlnx,gpio-width = <0x8>;`; "The `"xlnx,"` prefix protects against name collisions and it is arbitrary string. To get more information about convention that Xilinx is using for its DTS files, this can be found in [3].

There are some other properties that can be described in the DTS file like aliases and chosen. Which is best described in [2].

3.5.1.6 Overwriting

To overwrite a property, the node needs to be referenced using the ampersand character and the label. Later device tree entries overwrite earlier entries (the sequence order of entries is what matters, hence the include order matters). Typically the higher layers (e.g. carrier board device tree) overwrite the lower layers (e.g. SoC device tree) since the higher layers include the lower layers at the very beginning. e.g. for USB controllers which are

capable to be device or host (dual-role), one can overwrite the default mode explicitly using the `dr_mode` property:

```
&usbdev0 {
    dr_mode = "host";
};
```

3.5.2 DTS Organization in Xilinx

Since any hardware platform has a lot of attached devices and peripherals, describing node by node needs a lot of effort. Therefore it is better to use Xilinx tools for automating this process. Using SDK the following DTS can be generated for the system.

```
/dts-v1/;
/include/ "zynqmp.dtsi"
/include/ "zynqmp-clk-ccf.dtsi"
/include/ "pl.dtsi"
/include/ "pcw.dtsi"
/ {
    chosen {
        bootargs = "earlycon clk_ignore_unused";
        stdout-path = "serial0:115200n8";
    };
    aliases {
        ethernet0 = &gem3;
        serial0 = &uart0;
        serial1 = &uart1;
        spi0 = &qspi;
    };
    memory {
        device_type = "memory";
        reg = <0x0 0x0 0x0 0x80000000>, <0x00000008 0x00000000 0x0
            0x80000000>;
    };
};
```

which is organizing the hardware nodes into different files. `Zynqmp.dtsi` is a ready template to describe the processing system of Zynqmp. `Zynqmp-clk-ccf.dtsi` is describing clock configuration of each attached node. `Pl.dtsi` describes hardware nodes that are built in the programmable logic part.

`pcw.dtsi` is the configuration of the processor (processor configuration wizard). Since `zynqmp.dtsi` is a ready template, `pcw.dtsi` is used to configure or modify hardware nodes described in `zynqmp.dtsi`.

For example, by default all peripherals attached to the processing system are disabled as shown in figure 3.17.

```

ttc3: timer@ff140000 {
    compatible = "cdns,ttc";
    status = "disabled";
    interrupt-parent = <&gic>;
    interrupts = <0 45 4>, <0 46 4>, <0 47 4>;
    reg = <0x0 0xff140000 0x0 0x1000>;
    timer-width = <32>;
    power-domains = <&pd_ttc3>;
};

uart0: serial@ff000000 {
    u-boot,dm-pre-reloc;
    compatible = "cdns,uart-rlp12", "xlnx,xuartps";
    status = "disabled";
    interrupt-parent = <&gic>;
    interrupts = <0 21 4>;
    reg = <0x0 0xff000000 0x0 0x1000>;
    clock-names = "uart_clk", "pclk";
    power-domains = <&pd_uart0>;
};

uart1: serial@ff010000 {
    u-boot,dm-pre-reloc;
    compatible = "cdns,uart-rlp12", "xlnx,xuartps";
    status = "disabled";
    interrupt-parent = <&gic>;
    interrupts = <0 22 4>;
    reg = <0x0 0xff010000 0x0 0x1000>;
    clock-names = "uart_clk", "pclk";
    power-domains = <&pd_uart1>;
};

usb0: usb0@ff9d0000 {
    #address-cells = <2>;
    #size-cells = <2>;
    status = "disabled";
    compatible = "xlnx,zynqmp-dwc3";
    reg = <0x0 0xff9d0000 0x0 0x100>;
    clock-names = "bus_clk", "ref_clk";
    power-domains = <&pd_usb0>;
    ranges;
    nvmem-cells = <&soc_revision>;
    nvmem-cell-names = "soc_revision";
};

```

Figure 3.17: Disabled Peripherals

All shown peripherals here have status = “disabled”. But for a user who wants to activate UART1 peripheral, the pcw.dtsi will contain as shown in figure 3.18

```

&fpd_dma_chan7 {
    status = "okay";
};
&fpd_dma_chan8 {
    status = "okay";
};
&gpio {
    emio-gpio-width = <32>;
    gpio-mask-high = <0x0>;
    gpio-mask-low = <0x5600>;
    status = "okay";
};
&gpu {
    status = "okay";
};
&i2c0 {
    clock-frequency = <400000>;
    status = "okay";
};
&pinctrl0 {
    status = "okay";
};
&rtc {
    status = "okay";
};
&sdhci1 {
    clock-frequency = <199998000>;
    status = "okay";
    xlnx,mio_bank = <0x1>;
};
&serdes {
    status = "okay";
};
&uart1 {
    device_type = "serial";
    port-number = <0>;
    status = "okay";
    u-boot,dm-pre-reloc ;
};
&ams_ps {
    status = "okay";
};
&ams_pl {
    status = "okay";
};

```

Figure 3.18: Enabling Peripheral

Status of uart1 here is “okay”.

3.5.3 Other User DTS Configuration

There are other values that can be provided in Device Tree description which aids in building the system. For example the aliases values defines which node is targeted. Also there are three sources for the kernel boot command line in general:

- Those given as CONFIG_CMDLINE in the kernel configuration.
- Those passed on by the boot loader (U-boot on ARM processors).
- Those included in the device tree, under chosen/bootargs (see listing above).

Which one is used, depends on kernel configuration parameters.

3.5.4 Using SDK to build DTS file

3.5.4.1 Procedure Using SDK GUI

Open GUI interface of Xilinx SDK

From top menu open Xilinx>>Repositories.. The following window shall open.

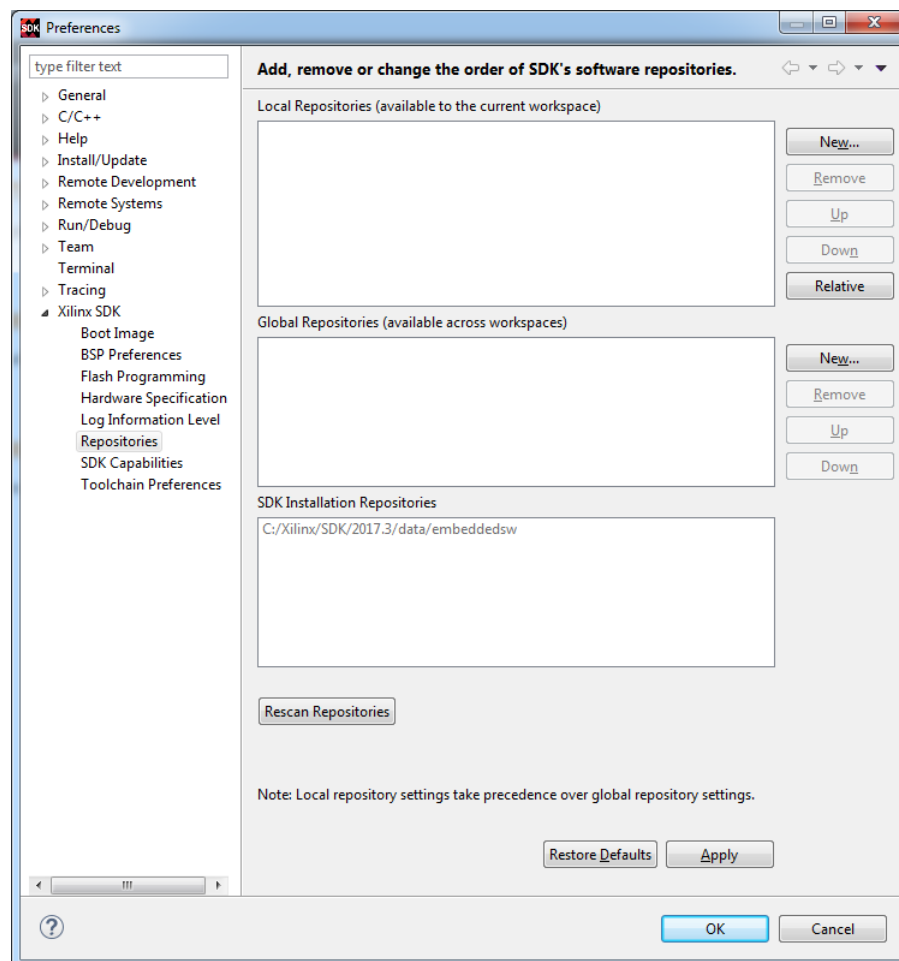


Figure 3.19: Repositories in SDK [1]

At this step we need to add repository of Xilinx Device Tree. This can be downloaded from this link

<https://github.com/Xilinx/device-tree-xlnx.git>

Add the local repository by hitting on ‘New’ button and browse to choose location of downloaded Xilinx Device Tree Repository.

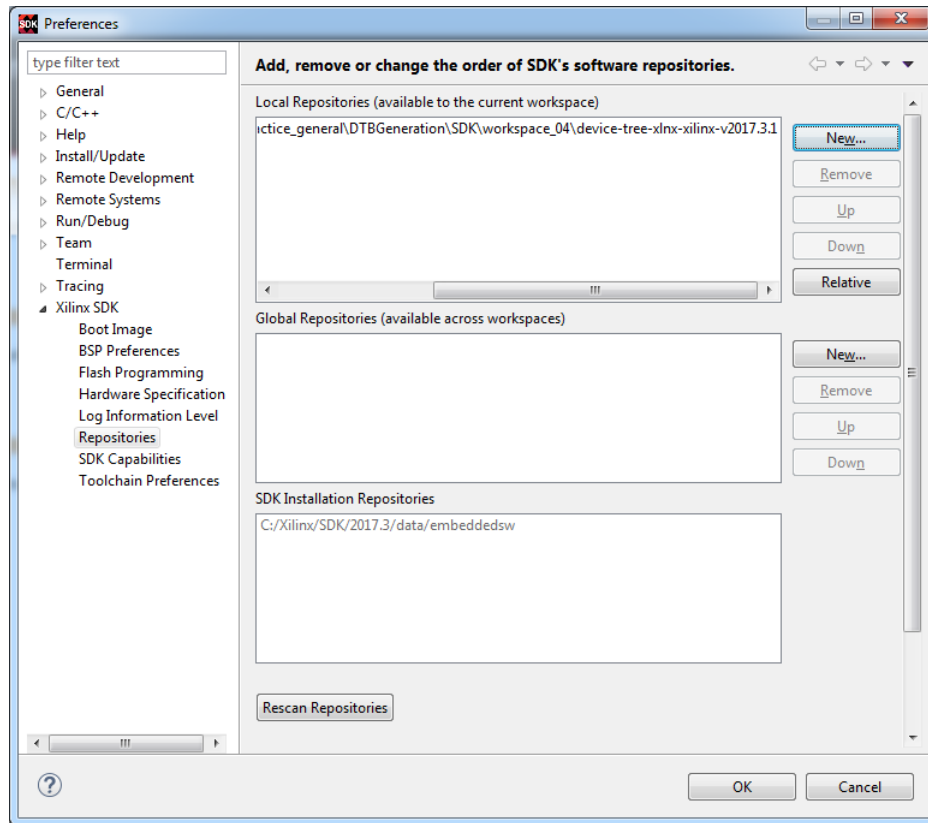


Figure 3.20: Add New Repository [1]

Click ‘OK’. Then in the Main SDK window, In the top menu, File >> New >> Project
 Choose from Xilinx >> Hardware Platform Specification
 Browse to add Target Hardware Specification is declared previously. Then click ‘Finish’.
 Then in SDK main window choose File>>New>>Board Support Package. The following wizard shall open.

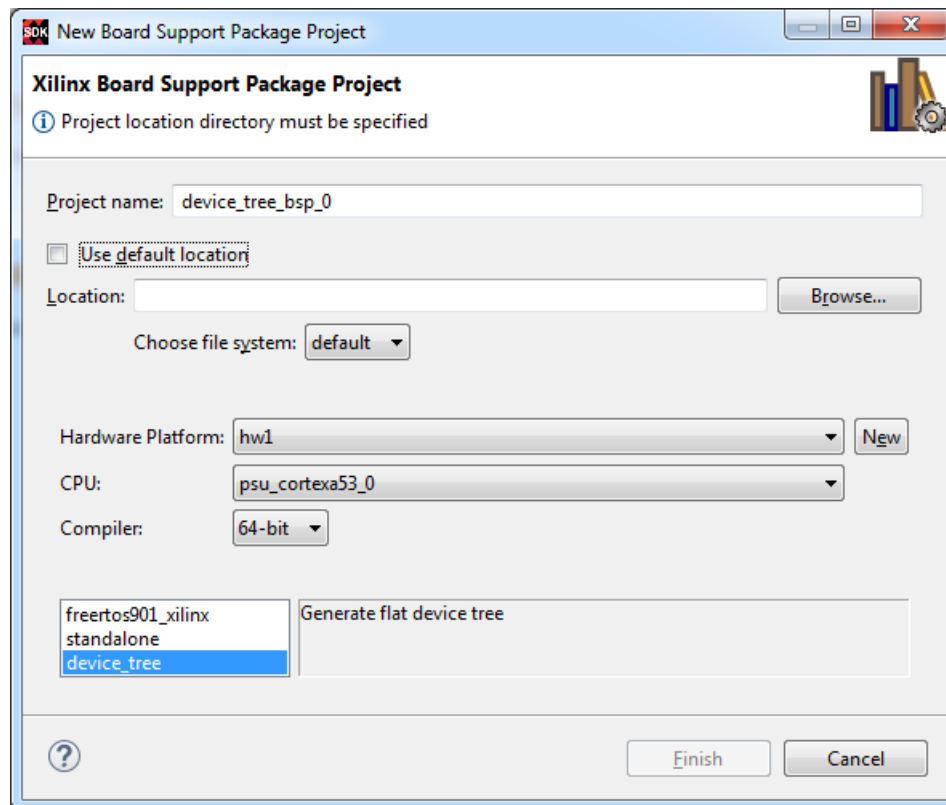


Figure 3.21: New Device Tree BSP [1]

Choose Location of the bsp project. Important to choose “device_tree” in the OS field. The following window will open to choose further configuration/setting for the generated dts file.

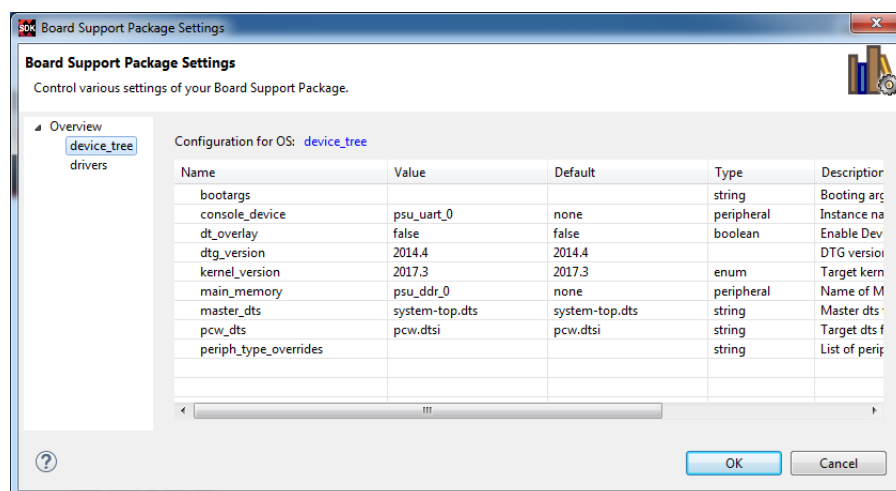


Figure 3.22: Device Tree BSP Configuration [1]

For example you can change name of the generated dts file or add boot arguments. Click ‘OK’. Generated dts files can be obtained from the created BSP directory.

3.5.4.2 Procedure Using XSCT Command Line

First we need to have local repository of Xilinx Device Tree which can be obtained by

```
git clone https://github.com/Xilinx/device-tree-xlnx.git
```

The following routine can be used to generate the dts files

```
createhw -name hw0 -hwspec $HDF_Source
repo -set $XDT_Dir
createbsp -name generated_DTS -hwproject hw0 -proc psu_cortexa53_0 -os
device_tree -arch 64

configbsp -bsp generated_DTS master_dts HW_System.dts
regenbsp -bsp generated_DTS

closehw hw0 closebsp generated_DTS
```

Note that in this routine, \$XDT refers to downloaded Xilinx Device Tree repository.

3.5.5 Generating DTB Binary

DTS file has to be converted eventually to DTB file to be understood by the kernel. Device Tree Compiler (DTC) can be obtained from Linux mainstream at:

```
$ git clone https://git.kernel.org/pub/scm/utils/dtc/dtc.git
```

then navigate to the downloaded directory and build it

```
$ cd dtc
$ make
```

Note that no cross compile flag is needed to build it as this build is to build the dtc compile itself on the host machine. So the resulted compiler is needed. So export it to the PATH variable

```
$ export PATH=/path/to/dtc:$PATH
```

The DTB file can be obtained then by

```
$ dtc -I dts -O dtb -o <devicetree name>.dtb <devicetree name>.dts
```

3.6 UBoot

Universal Bootloader is used for booting the kernel and rest of operating system components. It can be downloaded directly for Xilinx FPGAs from

```
$ git clone https://github.com/Xilinx/u-boot-xlnx.git
```

Then this can be configured according to the target board and built to generate binary file.

3.6.1 Quick Build

To build the uboot, a DTC compiler is needed. This can be downloaded also from here

```
$ git clone https://git.kernel.org/pub/scm/utils/dtc/dtc.git
```

This compiler should be built also. After building it add path of this compiler to the \$PATH environment variable

```
$ export PATH=/home/user/work/dtc:$PATH
```

also to build the UBoot, a cross compiler is needed

```
$ export CROSS_COMPILE=/path/to/bin/aarch64-linux-gnu-
```

now open the downloaded UBoot source

```
$ cd path/to/u-boot-xlnx
```

we need to load configuration of the target architecture (which is Xilinx Zynqmp)

```
$ make xilinx_zynqmp_zcu102_rev1_0_defconfig
```

note that this configuration file exists in ./configs directory. So make sure always of the right name of this file. You can add further your own configurations

```
$ make menuconfig
```

building is now ready

```
$ make
```

after make process is finished, output file u-boot.elf is needed in other steps.

3.6.2 U-Boot Configuration

Previous steps are straight forward steps to build a U-boot quickly. Occasionally, this is not the case as we need to configure U-boot to match our target hardware. U-boot is configured and built like a Linux manner. This means a .config file is loaded into u-boot directory in order to have final configuration of the system. As described before, this .config file is loaded from the target architecture configuration file that is located in ./configs directory. So eventually target_hw_defconfig is most important file as it contains all needed configurations. Among other system configuration, two important configurations we have to take care about: System Configuration and Device Tree Configuration, which are highlighted in figure 3.23

```

CONFIG_ARM=y
CONFIG_SYS_CONFIG_NAME="xilinx_zynqmp_zcu102"
CONFIG_ARCH_ZYNQMP=y
CONFIG_SYS_MALLOC_F_LEN=0x8000
CONFIG_SPL_SYS_MALLOC_SIMPLE=y
CONFIG_SPL_DM=y
CONFIG_DM_SPI_FLASH=y
CONFIG_DM_GPIO=y
CONFIG_ZYNQMP_QSPI=y
CONFIG_ZYNQMP_USB=y
CONFIG_SYS_TEXT_BASE=0x8000000
CONFIG_DEFAULT_DEVICE_TREE="zynqmp-zcu102"
CONFIG_SPL=y
CONFIG_FIT=y
CONFIG_FIT_VERBOSE=y

```

Figure 3.23: UBoot Configuration Sources

3.6.2.1 System Configuration

First configuration is called system configuration or as named as

“CONFIG_SYS_CONFIG_NAME”.

This parameter tells the compiler where to find other extension for the configuration. This extension of the configuration is not included in defconfig file as it is usually tailored according to the used system. For example, the user board is designed to boot from QSPI. So in the main configuration file (defconfig) we can enable booting from QSPI, but in the System Configuration file we can mention which partition of the QSPI we shall boot. In our example here the extended system configuration file is called Xilinx_zynqmp_zcu102.h and it can be found in ./include/configs.

3.6.2.2 Device Tree Configuration

The second highlighted configuration is the device tree configuration. As its name makes it clear, The running hardware has to be defined for the U-boot in order to enable U-boot to run into it. In our example here, the file zynqmp-zcu102.dts shall be provided in that directory: ./arch/arm/dts/

It worth mentioning here that special drivers are built in u-boot to handle then working with the attached devices that are described in device tree file.

3.6.2.3 Board Configuration

Nevertheless, you can configure how U-boot will detect the boot media from a board configuration file that is located in ./board directory. Here you can find some ready board configuration files for some supported device. For our case, Xilinx board configuration file is located in subdirectory of Xilinx. What is important in this file is the setting for the booting modes as shown in figure 3.24

```

ret = zynqmp_mmio_read((ulong)&crlapb_base->boot_mode, &reg);
if (ret)
    return -EINVAL;

if (reg >> BOOT_MODE_ALT_SHIFT)
    reg >>= BOOT_MODE_ALT_SHIFT;

bootmode = reg & BOOT_MODES_MASK;

puts("Bootmode: ");
switch (bootmode) {
case USB_MODE:
    puts("USB_MODE\n");
    mode = "usb";
    env_set("modeboot", "usb_dfu_spl");
    break;
case JTAG_MODE:
    puts("JTAG_MODE\n");
    mode = "pxe dhcp";
    env_set("modeboot", "jtagboot");
    break;
case QSPI_MODE_24BIT:
case QSPI_MODE_32BIT:
    mode = "qspi0";
    puts("QSPI_MODE\n");
    env_set("modeboot", "qspiboot");
    break;
case EMMC_MODE:
    puts("EMMC_MODE\n");
    mode = "mmc0";
    env_set("modeboot", "emmcboot");
    break;
case SD_MODE:
    puts("SD_MODE\n");
    mode = "mmc0";
    env_set("modeboot", "sdboot");
    break;
case SD1_LSHFT_MODE:
    puts("LVL_SHFT ");
    /* fall through */
case SD_MODE1:
    puts("SD_MODE1\n");
    mode = "mmc1";
    env_set("sdbootdev", "1");
    break;
case NAND_MODE:
    puts("NAND_MODE\n");
    mode = "nand0";
    env_set("modeboot", "nandboot");
    break;
default:
    mode = "";
    printf("Invalid Boot Mode:0x%x\n", bootmode);
    break;
}

```

Figure 3.24: UBoot Board Configuration

In this sample, U-boot configuration is reading specific register in the Xilinx FPGA in which it recognizes which medium is set for booting. Value of this register is set according to some jumpers settings in the host board. This in turn will set some values back in the extended system configuration file. i.e. System Configuration file knows boot medium according to values passed from board configuration file.

3.6.3 Debugging U-boot Settings

During development phase, developer may need to explore or understand some environment variable settings. Then he may need to stop auto booting of u-boot in order to tweak some settings. In this case we disable auto booting by setting boot delay to -1. This setting can be added using menuconfig in the main menuconfig window ">>" delay in seconds before automatically booting". This can be set to -1 or in the main configuration file we can set it using the parameters "CONFIG_BOOTDELAY=-1"

3.6.4 Adding Customized Configurations

In System Configuration file there is definition for the needed setting of booting process defined in the macro `CONFIG_EXTRA_ENV_SETTINGS`. For example here, in board configuration file, the “modeboot” variable is set to either `sdboot`, `nandboot`, `qspiboot`, etc. In System Configuration file, there is a description of how to boot using these values.

As a common technique, you can describe booting approach in external file called usually as `uEnv.txt`. This is not common in all u-boot files but mostly used. For example in Xilinx it is defined in the case of `sdboot` the following

```
sdboot=mmc dev $sdbootdev && mmcinfo && run uenvboot || run
sdroot$sdbootdev;
```

Which means that u-boot should switch to `mmc dev “sdbootdev”` then prints information about this sd device then runs a predefined command called `uenvboot` or run another predefined command `sdroot`. In `uenvboot` command it is checking whether there is a file called `uEnv.txt` or not, if so it should boot according to description mentioned in `uEnv.txt` a separate text file. For example the following can be defined in this text file

```
ddr=00:0a:35:00:01:22
kernel_image=uImage
devicetree_image=devicetree.dtb
sdboot=if mmcinfo; then echo UENV Copying Linux from SD to RAM... && load
mmc 0 0x3000000 ${kernel_image} && load mmc 0 0x2A00000 ${
devicetree_image} && bootm 0x3000000 - 0x2A00000; fi
```

what is needed then is to put this file with the content of the SD card with other booting files.

Importance of this approach appears when during development phases. Instead of compiling u-boot each time you modify the global environment variable, you can direct u-boot to simply to read these values from external file.

3.6.5 Booting from Correct SD Partition

Usually SD card is partitioned into two partitions: boot (partition 1) and root (partition 2). Pointing to the right partition is clarified in the environment variable

```
sdboot=mmc dev $sdbootdev && mmcinfo && run uenvboot || run
sdroot$sdbootdev;
```

The last command (`run sdroot$sdbootdev`) is running another environment variable after replacing “`$sdbootdev`” with its value. In our example here this “`$sdbootdev`” is expecting to have 0 (for `mmc0`) or 1 (for `mmc1`).

Both environment variables have the same arguments to boot from SD card except for partition number

```
"sdroot0=setenv bootargs $bootargs root=/dev/mmcblk0p2 rw rootwait\0" \
"sdroot1=setenv bootargs $bootargs root=/dev/mmcblk1p2 rw rootwait\0" \
```

What we can see then from these variables is they are using name of the partition (mmcblk0p2 or mmcblk1p2). One drawback can be faced from this definition is the name of the mmc device may change when linux kernel is loading. For example, in your device tree you may have

```
aliases {
    ethernet0 = &gem0;
    gpio0 = &gpio;
    i2c0 = &i2c0;
    i2c1 = &i2c1;
    mmc0 = &sdhci1;
    mmc1 = &sdhci0;
    rtc0 = &rtc;
    serial0 = &uart0;
    serial1 = &uart1;
    serial2 = &dcc;
    spi0 = &qspi;
    usb0 = &usb0;
};
```

Figure 3.25: MMC in Device Tree File

But when linux kernel is loaded the driver of sdhci devices will label it in reverse

```
[ 2.562611] sdhci: Secure Digital Host Controller Interface driver
[ 2.568711] sdhci: Copyright(c) Pierre Ossman
[ 2.573048] sdhci-pltfm: SDHCI platform and OF driver helper
[ 2.631082] mmc0: SDHCI controller on ff160000.sdhci [ff160000.sdhci] using ADMA 64-bit
[ 2.691081] mmc1: SDHCI controller on ff170000.sdhci [ff170000.sdhci] using ADMA 64-bit
[ 2.699235] ledtrig-cpu: registered to indicate activity on CPUs
[ 2.705266] usbcore: registered new interface driver usbhid
[ 2.710763] usbhid: USB HID core driver
[ 2.715077] fpga_manager fpga0: Xilinx ZynqMP FPGA Manager registered
[ 2.721916] pktgen: Packet Generator for packet performance testing. Version: 2.75
[ 2.730301] Netfilter messages via NETLINK v0.30.
[ 2.735031] ip_tables: (C) 2000-2006 Netfilter Core Team
[ 2.736972] mmc0: new high speed MMC card at address 0001
[ 2.737213] mmcblk0: mmc0:0001 W52516 14.3 GiB
[ 2.737303] mmcblk0boot0: mmc0:0001 W52516 partition 1 4.00 MiB
[ 2.737391] mmcblk0boot1: mmc0:0001 W52516 partition 2 4.00 MiB
[ 2.737473] mmcblk0rmb: mmc0:0001 W52516 partition 3 4.00 MiB
[ 2.740966] mmcblk0: p1
[ 2.751724] mmc1: new SDHC card at address 0001
[ 2.751906] mmcblk1: mmc1:0001 00000 14.9 GiB
[ 2.753878] mmcblk1: p1 p2
[ 2.782047] Initializing XFRM netlink socket
```

Figure 3.26: MMC in Boot Sequence

In this case, “sdroot0” and “sdroot1” shall be reversed in device tree. Or the better solution is to point to the target partition in u-boot using uuid. This can be achieved as follows

In the configuration file Xilinx_zynqmp.h, the following variable is defined for sdboot

```
sdboot=mmc dev $sdbootdev && mmcinfo && run uenvboot || run
sdroot$sdbootdev;
```

In which it runs “sdroot\$sdbootdev”. we can replace this last command with another command to direct u-boot to boot using uuid

```
sdboot=mmc dev $sdbootdev && mmcinfo && run uenvboot || run getuuid && run
  uuidroot;
```

In which we run two commands, first one to get uuid of the target partition (second partition of the SD card). Then we boot according to the obtained uuid. So these two commands have to be defined also in the loaded environment variable as follows

```
"getuuid=part uuid mmc $sdbootdev:2 myuuid\0" \
"uuidroot=setenv bootargs $bootargs root=PARTUUID=$myuuid rw rootwait\0" \
```

3.6.6 Booting from TFTP Server

In case of there is a need to boot the Linux Image from a remote machine, this can be done in U-boot using tftpboot command. This is needed especially during development phases of a project.

Dealing with tftpboot command is pretty simple:

```
tftpboot <load_addr> <target_image>
```

Using this command, we can then load Linux image, device tree file, compressed root file system. For example the following command can be used in environment variables declaration space

```
tftpboot 80000 Image && tftpboot $fdt_addr system.dtb && tftpboot 6000000
  rootfs.cpio.ub
```

In which we load the pointed files into the declared RAM address of the processor. Afterwards, all what we need is to boot from this loaded items. This can be done using the following command

```
booti 80000 6000000 $fdt_addr
```

But note that this tftpboot depends on initial declaration of the ip address of the tftp server. This can be easily defined also in u-boot environment variables space using the following sample

```
setenv ipaddr 192.168.1.10
setenv serverip 192.168.1.1
```

In which we set ip address for the target board and declare the ip address of the server.

At the server itself, a tftp server shall be running. In windows, free tool can be downloaded and run like tftpd64 (tftpd64.460.zip).

3.6.7 Booting From NFS

Using the same mechanism, we can use u-boot to enable loading root fs from a network in what is called NFS in which the root file system is hosted in a remote machine in the network without the need to transfer the whole file system to the target board. Linux

kernel when loaded will point to this remote root file system. This can be done in u-boot by passing the appropriate arguments to Linux kernel.

```
def_args=console=ttyPS0,115200 rw earlyprintk
nfs_args=setenv bootargs ${def_args} rootwait root=/dev/nfs nfsroot=${
serverip}:${serverpath} ip=dhcp
```

where serverpath is a variable that shall be defined to point out where the location of the NFS is. In the remote machine, an NFS server shall be installed. Then we can boot using the following command

```
bootm ${kernel_addr} - ${devicetree_addr}
```

3.6.8 Scripting Boot Procedure

So far we have seen many settings and configurations as well as different booting methods. As a developer, you might need to change all these configurations while building your system. There is one common approach is to write an initial script that U-boot can understand and behave according to instructions described in that file. The idea is somehow near to uEnv.txt mentioned previously with the difference that here in the script you can write U-boot commands.

3.6.8.1 Boot Script Sample

The following can be a sample of how a booting script can look like

```
setenv kernel_img_name "Image"
setenv kernel_loadaddr 0x6000000

setenv rootfs_img_name "rootfs.cpio.uboot"
setenv rootfs_loadaddr 0x10000000

setenv dtb_img_name "devicetree.dtb"
setenv dtb_loadaddr 0x6e00000

tftpboot $kernel_loadaddr $kernel_img_name;
tftpboot $rootfs_loadaddr $rootfs_img_name;
tftpboot $dtb_loadaddr $dtb_img_name;

booti $kernel_loadaddr $rootfs_loadaddr $dtb_loadaddr;
```

At the moment, let's assume that this script is saved in normal text file called sample_uboot_script.txt

3.6.8.2 Boot Script Format

In order to be able to run such a script within U-Boot environment, it should be prepared in special format. To prepare this format, there is an utility produced when compiling U-Boot. This utility is called mkimage and can be found after compiling U-Boot in the subdirectory "tools". So, you can add it to your PATH variable in Linux.

```
$ cd tools
$ export PATH=$PWD:$PATH
```

Then, to create the needed script format use the such like the following command

```
mkimage -T script -A arm64 -C none -n 'Our U-Boot script.' -d
sample_uboot_script.txt sample.scr
```

Where sample_uboot_script.txt is the text file contains your U-Boot script commands. Sample.scr is the output of this mkimage command. In this command we mentioned also which architecture the script will be running on using -A switch. -C is set to none to mean uncompressed image, -T to define type of the image which is a script in our case.

Note that you can review details about any image using “mkimage -I <image_name>” in the host machine. Or in the target machine you can run “iminfo” in U-Boot prompt.

3.6.8.3 Boot Script Loading

We can run quickly this script in U-Boot prompt. First we have to load it into the FPGA System RAM. To do that, you can load it over tftp

```
tftpboot 0x12000 sample.scr
```

(Assuming that you have already set serverip and ipaddr variables)

Or if you would like to load it from SD card you can use

```
load mmc 0:1 0x12000 sample.scr
```

(Assuming that your SD card is defined at dev 0 - partition 1). 0x12000 is any temp address.

To execute the loaded script then, simply use the following

```
source 0x12000
```

Further enhancement can be applied by automating loading of the script file. This can be done by assigning dynamic IP address for the chip using “dhcp” command. But first we need to define name of the script file by:

```
setenv bootfile sample.scr
saveenv
```

Then calling dhcp command will download the file automatically

```
dhcp
```

If you want to restore default environment values that you have already changed in last steps, you can restore it by:

```
env default -a
saveenv
```

Then reboot the system directly.

Another simple alternative that we can use without changing environment variables is to issue dhcp command with the file name that we need to fetch.

```
dhcp 0x12000 sample.scr
```

where 0x12000 is RAM address to which the file will be loaded.

3.7 FileSystem

3.7.1 Definition

Every Embedded Linux system needs a root file system which includes all of the files needed for the system to work. To boot a system, enough software and data must be present on the root partition to mount other file systems. This includes utilities, configuration, boot loader information, and other essential start-up data.

What is meant here by file system also is the entire hierarchy of directories that is used to organize files on a computer system. This shouldn't make confusion with type of file system storage on hard disks (FAT, ext2, ext3, ext4, etc).

When the bootloader loads and starts the kernel, the kernel will mount the target file system and starts the init application.

Linux Foundation released what is called Filesystem Hierarchy Standard (FHS) which defines directory structure and contents in Linux distributions. It can be summarized as in figure 3.27.

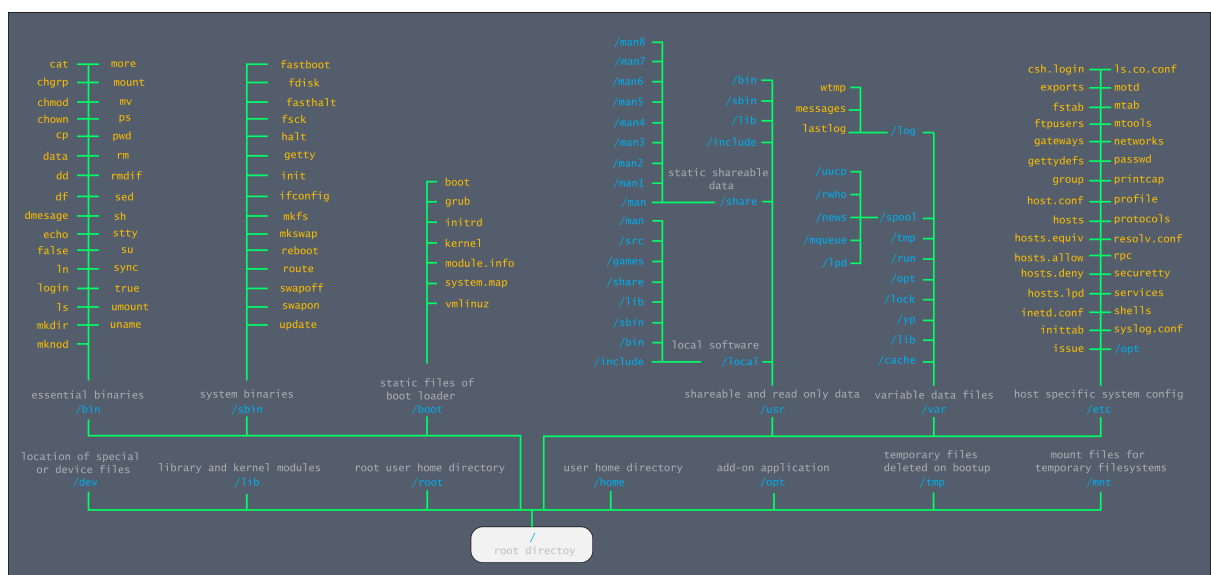


Figure 3.27: File System Structure

3.7.2 Building File System

This means to select files necessary for the system to run. There are minimum requirements for any file system, to allow the system to function properly:

Init application – shell – C library – Device Files – proc – sysfs.

Since these basic configurations are important and may differ from a system to another, a build tool can be used to create these basic components such as BusyBox.

3.7.2.1 Build Using Busybox

Busybox can be downloaded from

<https://busybox.net/downloads/>
or cloning using git by

```
$ git clone https://git.busybox.net/busybox
```

After downloading, navigate to the downloaded directory. Before starting the build operation, architecture of the target system and a cross compiler for it shall be defined.

```
$ export ARCH=arm
$ export CROSS_COMPILE=/path/to/your/CrossCompiler
$ make defconfig
$ make menuconfig
$ make
$ make install
```

This will result in a directory “_install” which contains the output of the build process

3.7.2.2 Build Using Buildroot

Although Busybox can help building minimal file system, however it will not result in the needed directories that are shown in previous figure 3.27. Busybox will generate only utilities that user chose contained in bin and sbin directories. The user still needs to add other directories manually.

As an alternative, Buildroot can be used to prepare the needed file system as well as needed utilities. In fact, Buildroot is using Busybox to generate needed utilities and binaries. Not only but also Buildroot can be used to generate the entire Linux system starting from bootloader, Linux kernel and the file system. Therefore build root is an efficient tool that can be used to build the file system.

To use Buildroot, first download it by

```
$ git clone https://git.buildroot.net/buildroot
```

then browse to the downloaded directory

```
$ cd buildroot
```

If you have ready configuration file that can be used for your target, you can load it using

```
$ make your_target_file_defconfig
```

You can add more changes to these configurations or if you don't have ready configuration file, you can use menuconfig to add your configuration

```
$ make menuconfig
```

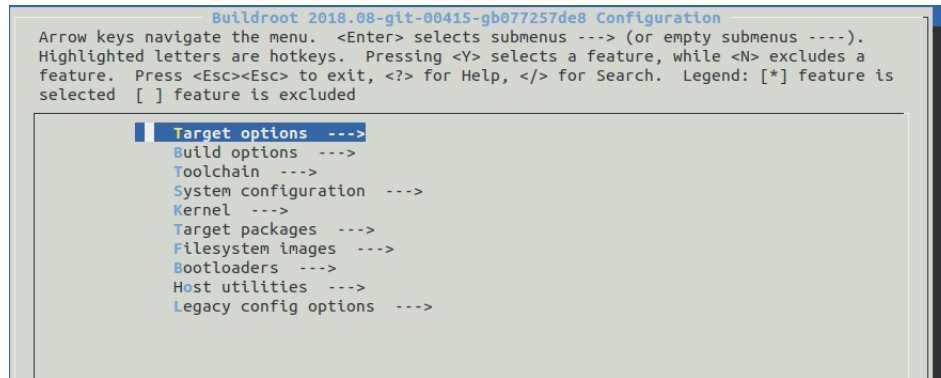


Figure 3.28: Buildroot Menuconfig

Important to make sure that you have the right Target options. Note also that for ARM64 architecture, glibc can only be used as other C libraries is not supporting this architecture [7].

Difference between these libraries is in its implementation. For example `malloc(0)` in glibc returns a valid pointer to something, while in uClibc calling `malloc(0)` returns a NULL [8].

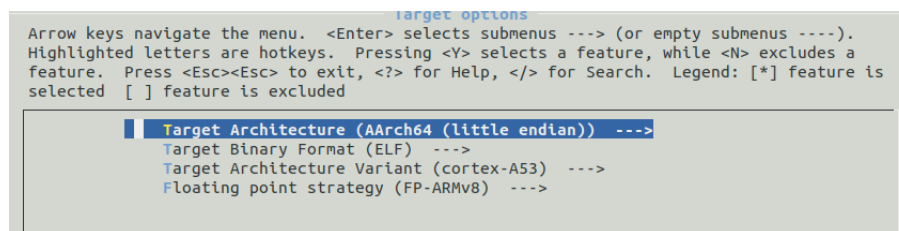


Figure 3.29: Buildroot Target Architecture

Then you can navigate through other options to choose other settings and utilities. Save your configurations then to start the build process. Cross Compiler is needed to be defined.

```
$ export CROSS_COMPILE=/path/to/your/CrossCompiler
$ make
```

This building process takes about 40 minutes to finish “According to your chosen preferences”.

When finished, the generated image can be found in `./output/images/rootfs.tar` which is root file systems compressed as tar file. To install it, mount the root partition of the SD card.

```
$ mkdir -p ./temp_mnt
$ sudo mount /dev/sdb2 temp_mnt
```

then extract the compressed file in the mounted location

```
$ sudo tar -xpf rootfs.tar -C /path/to/mounted/location
```

Root File System is now ready.

Note:

During building the rootfs, downloading of linux header was corrupted as the source of the linux (cdn.kernel.org) was not registered in locally verified URLs.

Configuration is set to use wget utility to download these linux headers. Luckily the error can be demoted then by using the switch `-no-check-certificate` which turns off checking if the link is secured or not. But where to add this switch then. This has to be done in configuration file. So after loading configuration file (xilinx_zynqmp_defconfig for instance), open the hidden file `‘.config’`. There is one parameter called `BRE2_WGET`. Modify this parameter to be

```
BR2_WGET = "wget -passive-ftp -no-check-certificate -nd -t 3"
```

Then you can build rootfs using make utility. Alternatively, you can use menuconfig to edit your preferences.

3.7.3 Buildroot Quick Guides

3.7.3.1 Add Overlays

User may need to add additional directory tree over the default created one. For example, User may need to create home directory for another user. This can be done using overlays. Simply you can create the additional needed directory tree and direct configuration of Buildroot to add this tree by mentioning its path. This setting can be done in configuration file as

```
BR2_ROOTFS_OVERLAY="/path/to/location/of/neededtree"
```

Or using menuconfig in section “System Configuration>Root filesystem overlay directories”.

3.7.3.2 Modify Using Post Script

Other modifications in the generated file system may be needed that cannot be done before building the system. For example assigning IP address for a network interface. This needs then to add some configuration in interfaces file. Such modifications can be scripted in an external file to be ready for execution before root file system is assembled.

For example, assume that we need to configure a network interface to acquire IP address dynamically. A file called `‘interfaces’` shall contain then the following lines

```
# interface file auto-generated by buildroot
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
```

In which we set interface eth0 to dynamically get IP address. Lets assume that this file is stored in some place like `“external_mod/interfaces”`.

We need then to create the script that will run to replace this ready interfaces file with the default one. Such a script could be like the following

```
#!/bin/sh -e

FS_DIR=$1

cp /path/to/external_mod/interfaces $FS_DIR/etc/network/interfaces
```

In this script and according to Buildroot manual, "The path to the target filesystem is passed as the first argument to each script". That is why we use "FS_DIR=\$1" as the first command. Note that "/path/to/external_mod" can be relative to Buildroot main directory.

In case you want to pass other arguments to these post scripts, write down these arguments in a config option called BR2_ROOTFS_POST_SCRIPT_ARGS. Note then that first argument by default is the path to target file system (i.e the image source directory that will be created). So any other argument defined in BR2_ROOTFS_POST_SCRIPT_ARGS will have the second order (i.e when referring to it the post scripts, use \$2, \$3,..etc).

Remaining now to direct Buildroot to use this script before creating the file system image. This can be done in configuration file using the following setting

```
BR2_ROOTFS_POST_BUILD_SCRIPT="../NetworkSettings.sh"
```

In which 'NetworkSettings' is the name of the script that contains copying commands. Alternatively, we can set it using menuconfig window in section "System Configuration > Custom scripts to run before creating filesystem images".

If you want to use more than one post script, simply enlist them with space separated in this field.

3.7.3.3 Modify Using Patches

More sophisticated modifications may be needed. For example, you may need to modify a package that going to be installed in the root file system. So waiting until the package is installed to change it is meaningless as you have to tweak it before it is installed. Such a situation can be handled by preparing a patch file that contains what you are going to change in the package settings files.

First you have to create a patch file with your changes. For example we will discuss here how to enable ssh connection for our target system (as ssh server) before deploying the root file system.

In 'sshd_config' file that is located in /etc/ssh, the following parameter should be set to yes to order to allow access over ssh port

```
PermitRootLogin yes
```

By default when building root file system it will be disabled

```
#PermitRootLogin prohibit-password
```

The following patch file was then created to make this change in 'sshd_config' file

```
--- openssh/orig_sshd_config 2018-07-13 07:16:48.154494887 -0400
+++ openssh/sshd_config      2018-07-13 06:40:58.000000000 -0400
```

```
@@ -29,7 +29,7 @@
# Authentication:

#LoginGraceTime 2m
-#PermitRootLogin prohibit-password
+PermitRootLogin yes
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10
```

This file was created using ‘diff’ command to get the patch file.

```
$ diff -u orig_sshd_config sshd_config > sshd.patch
```

Where orig_sshd_config was obtained actually during early development phases of the project.

Important to point out here is:

- The patch file should be name as filename.patch, as Buildroot will use this extension to apply the patch.
- Also to direct Buildroot to apply this patch file to our target package, we need to store this file in a directory with the same name of our package. So it should be maintained at “openssh/sshd.patch”.
- First two lines defines where exactly to find the target patched file in the package source directory.

Lets assume now that this directory is located in a directory called “external_mod”. Remaining now to direct Buildroot to apply external patch file. In Buildroot configuration file choose to enable ssh package

```
BR2_PACKAGE_OPENSSH=y
```

Or in menuconfig “Target packages > Networking applications > openssh”.

We need to set the following parameter to indicate where to find patch files

```
BR2_GLOBAL_PATCH_DIR="/path/to/external_mod"
```

Alternatively, we can use menuconfig in section “Build options > global patch directories”. The patch should be applied then. If we have more than one patch to be applied to the same package, we have to rename the patch files then to be in alphabetic order. For example 001_sshd.patch then 002_sshd.patch.

3.7.3.4 Rebuild Packages

For quick modifications in installed packages, developer may not need to rebuild the whole file system for these quick modifications. Instead, the target package binary can be only rebuilt in Buildroot. This can be done using the following command

```
$ make package_name-rebuild
```

Where package_name is the binary that needed to be changed. In this case only the resulted binary can replace the existing binary without the need to deploy the whole root file system again.

3.7.3.5 Get Package from User Specific Location

Normal operation of Buildroot is to download a package whenever it is specified to be included in the build. Some Developers may prefer to use their version of the package. So downloading the package is not entirely needed. In this case, we can use

`<pkg_name>_OVERRIDE_SRCDIR`

to direct Buildroot to use our package source. This configuration can be added in main configuration file of Buildroot and we have to assign path to the package source to it. Importance of this mechanism appears when we run `make clean`. The source will be untouched and can be integrated any time we rebuild the root file system.

3.7.3.6 User Control During Build Process

Building the root file system is based mainly on building target packages. Buildroot while building any package; is going through the following phases:

Download – Extract – Patch – Configure – Build – Install.

There are other phases that you can check in Buildroot user manual, but for simplicity not all of them are mentioned here. Fortunately, developers can add some control in between these phases. It is called hooks. In Buildroot there are two hooks for each phase, Pre & Post. Using these hooks developers can define some actions to be executed before or after the phase. To define at which package the hook shall be applied, Buildroot is using the following format for any hook definition

`<PACKAGENAME>_<PREorPOST>_<PHASE>_HOOKS`

The hook variable can be defined in the package mk file (for each package defined, it has `packagename.mk` file in which we declare some information about configurations and build options for the package). Prior to defining the hook variable, we have to define first (in mk file also) actions that shall be executed by at the hook point.

```
Define PACKAGENAME_DEVELOPER_ACTION_SET
    echo "User specific configurations"
    cp file1 file2
    # ... other developer's actions
Endef
```

Then we assign these actions to the hook variable, for example:

```
OURPACKAGE_PRE_CONFIGURE_HOOKS += PACKAGENAME_DEVELOPER_ACTION_SET
```

Buildroot allows hooks variables for the following phases

```
PACKAGENAME_PRE_DOWNLOAD_HOOKS
PACKAGENAME_POST_DOWNLOAD_HOOKS
PACKAGENAME_PRE_EXTRACT_HOOKS
PACKAGENAME_POST_EXTRACT_HOOKS
PACKAGENAME_PRE_RSYNC_HOOKS
PACKAGENAME_POST_RSYNC_HOOKS
PACKAGENAME_PRE_PATCH_HOOKS
PACKAGENAME_POST_PATCH_HOOKS
```

```

PACKAGENAME_PRE_CONFIGURE_HOOKS
PACKAGENAME_POST_CONFIGURE_HOOKS
PACKAGENAME_PRE_BUILD_HOOKS
PACKAGENAME_POST_BUILD_HOOKS
PACKAGENAME_PRE_INSTALL_HOOKS (for host packages only)
PACKAGENAME_POST_INSTALL_HOOKS (for host packages only)
PACKAGENAME_PRE_INSTALL_STAGING_HOOKS (for target packages only)
PACKAGENAME_POST_INSTALL_STAGING_HOOKS (for target packages only)
PACKAGENAME_PRE_INSTALL_TARGET_HOOKS (for target packages only)
PACKAGENAME_POST_INSTALL_TARGET_HOOKS (for target packages only)
PACKAGENAME_PRE_INSTALL_IMAGES_HOOKS
PACKAGENAME_POST_INSTALL_IMAGES_HOOKS
PACKAGENAME_PRE_LEGAL_INFO_HOOKS
PACKAGENAME_POST_LEGAL_INFO_HOOKS

```

3.7.3.7 Format of Zipped Output

When using the result output to be loaded into system RAM directly, a cpio format is needed. i.e. a tar output can't be extracted into the system RAM. More details about the reason is mentioned in this link. <https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt>.

3.7.4 File System Location

Many O/S distributions use a small RAM disk as the initial root filesystem and the init program will enumerate the devices and typically mount other filesystems.

- Removable media: SD Card, USB, HDD.
- On-board storage: Flash, RAM.
- Network (NFS).

Prior to getting more details about these different locations, we have to understand first that natively a Linux system (or an operating system) can (and indeed) run on the system RAM. In other words, Filesystems images that we can get from Buildroot or Busybox are loaded into the system RAM. This is possible because Buildroot or Busybox are generating minimal file system with the minimum needed binaries so that it can fit into the system RAM. This minimal Filesystem is called then Initial RAM Disk (initrd) or another type called tmpfs.

This note was important to mention to give the idea that a Linux system can successfully run based on this minimized Filesystem (initrd or tmpfs). What is done next is Linux is recognizing other attached devices and starts to mount it; the whole Filesystem will be extended then.

Mounting a Filesystem means to attach this Filesystem to the original Filesystem. It is simply can be thought like a tree; the original Filesystem (Root Filesystem) is already

started in system RAM then it will be connected to another Filesystem tree. The attached Filesystem shall be organized also in the Root Filesystem; therefore we shall define a location in the Root Filesystem tree. This location is called Mount Point through which the additional Filesystem will be attached. For example when you attach a CDROM to the system, one directory will be created in the Root Filesystem (usually `/mnt/cdrom`). This created directory will be the root of the new attached CDROM and in the same time it acts as part of the Root Filesystem. Mounting this CDROM doesn't mean that is loaded into the system RAM also, it is only connected through the created Mount Point (`/mnt/cdrom`).

In Linux to review all attached Filesystems, this can be done using command `df`. While in order to see how much space is occupied by the Root Filesystem in the RAM; this can be checked using `free` command.

One more point to mention here also is; since `initrd` or `tmpfs` is running or located in system RAM, it is lost after system is powered down. Whenever a Linux system is starting, it needs to know location of the Root Filesystem Image. User can provide Root Filesystem Image manually (using `tftp` methods as declared in U-Boot section). For stable or After-production systems, Location of the Root Filesystem is given to the kernel by defining the argument `"root=..."`.

Based on these notes, we can easily understand that Linux system can start while the final Filesystem is located in another media (remote or attached).

3.8 Linux Kernel

Preparing the Linux kernel is quite simple. Xilinx has ready configuration file that can be used to compile Linux kernel for Zynq devices. As quick declaration, the following commands can be used to get a Linux kernel for ZynqMP. Only a cross compiler is needed before running these commands.

```
$ git clone https://github.com/Xilinx/linux-xlnx.  
$ git cd linux-xlnx  
$ make ARCH=arm64 xilinx_zynqmp_defconfig  
$ make ARCH=arm64 -j4
```

The output will reside in `arch/arm64/boot/`

3.8.1 Kernel Image Forms

The resulting image from last step is uncompressed Linux Kernel Image. This is what is supported by default for ZynqMP, other development areas refer to the resulting uncompressed image as `vmlinux`. It is simply an elf file. Linux Kernel can be produced also as compressed image called `zImage`. In this case we have to generate using

```
$ make zImage
```

But note that this is not supported for ZynqMP.

It can be also produced as uImage which is the compressed version of zImage prepended by a U-boot header which is information for U-boot to recognize this uImage when booting.

To get uImage, we must have mkimage utility (obtained from U-Boot in its “tools” directory). We have to add path to mkimage to PATH env. Variable then we can run the following while compiling Linux image

```
$ make uImage LOADADDR=0x80008000
```

The other way is to use mkimage separately to convert any linux image to uImage

```
$ mkimage -A arm64 -O linux -T kernel -C none -a 0x80008000 -e 0x80008000
-n "Linux kernel" -d arch/arm64/boot/zImage uImage
```

It is important to note also that each form of these images is booted by dedicated boot command in U-Boot. Uncompressed Image is booted using booti, zImage is booted using zboot, uImage is booted using bootm.

3.8.2 Flattened Image Tree

Starting from last point where a Kernel image has been prepended with load address information (uImage), this image looks as shown in figure3.30.

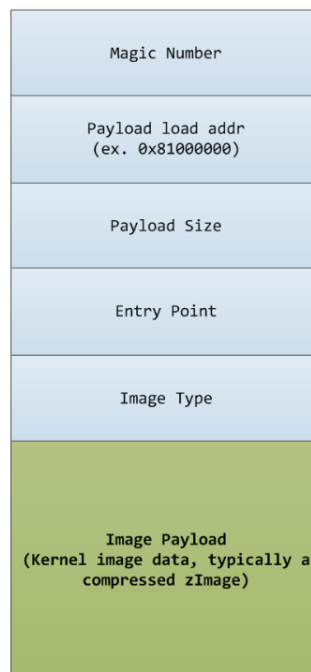


Figure 3.30: Kernel Image Structure [9]

In the same way, Device Tree file can be represented as an image, FileSystem can be compressed into an image, FPGA bit file, U-Boot scripts, etc. It is useful then if there is a way to include all the needed images in one image so that transferring and loading of the image can be faster.

Flattened Image Tree can be used to build such an image. Its approach is similar Device Tree file structure in which we describe each hardware as a node that contains other sub-hardware nodes. In FIT image we describe each image as node and provide information

needed for each node. As a sample, the following FIT file is provided in U-Boot as a sample of FIT image that includes a kernel and device tree

```
/*
 * Simple U-Boot uImage source file containing a single kernel and FDT blob
 */

/dts-v1/;

/ {
    description = "Simple image with single Linux kernel and FDT blob";
    #address-cells = <1>;

    images {
        kernel@1 {
            description = "Vanilla Linux kernel";
            data = /incbin/("./vmlinux.bin.gz");
            type = "kernel";
            arch = "ppc";
            os = "linux";
            compression = "gzip";
            load = <00000000>;
            entry = <00000000>;
            hash@1 {
                algo = "crc32";
            };
            hash@2 {
                algo = "sha1";
            };
        };
        fdt@1 {
            description = "Flattened Device Tree blob";
            data = /incbin/("./target.dtb");
            type = "flat_dt";
            arch = "ppc";
            compression = "none";
            hash@1 {
                algo = "crc32";
            };
            hash@2 {
                algo = "sha1";
            };
        };
    };

    configurations {
        default = "conf@1";
        conf@1 {
            description = "Boot Linux kernel with FDT blob";
            kernel = "kernel@1";
            fdt = "fdt@1";
        };
    };
};
```

```
};
```

So we can prepare images that we need in such a description, save it as `.its` file, create the fit image using `mkimage` utility

```
$ mkimage -f userImages.its all_images.itb
```

Then this `.itb` file can be transferred into system RAM to be loaded simply using “bootm” command. Recall from last section that `bootm` is used to boot images with U-Boot header. Note here that Linux kernel used to create this FIT image shall be `uImage`.

More details about working with FIT image can be found in U-Boot directory under `doc>>uImage.FIT`. What can be interesting to mention here is we can prepare two or more FIT images and transfer it to the system RAM, then we can pick a combination of first FIT kernel with second FIT device tree to boot. We can do that when calling `bootm`

```
bootm 20000:kernel@1 25000:filesystem@2 30000:fdt@3
```

Where 20000, 25000, 30000 are addresses of the FIT in the system RAM. We can also choose to boot using kernel and device tree (for example) as follows

```
bootm 20000:kernel@1 - 30000:fdt@3
```

Where the space for the file system has been specified “-” to tell U-Boot to discard it.

Important to note then when loading the `all_images.itb`, “bootargs” environment variable shall be correctly defined. Otherwise kernel will fail to load as no arguments have been passed to him. To set `bootargs` simply using the following

```
setenv bootargs <<user_args>>
```

Where `user_args` can be for example

```
“console=ttyPS0,115200 rw earlyprintk rootwait root=/dev/mmcblk1p2”
```

Another important point is to choose load and entry address of the kernel not to overwrite other loaded images. For resolving some issues that may face you when starting working with FIT image, you may need to load it, extract it manually to debug behavior. To extract it manually

```
imxtract 0x10000000 kernel@1 0x06000000
```

Where 0x10000000 is address where FIT image has been loaded, 0x06000000 is the address to which you want to load the kernel image. Same can be executed for `fdt`

```
imxtract 0x10000000 fdt@1 0x06e00000
```

then if successful, you can try to load the extracted images

```
bootm 0x06000000 - 0x06e00000
```

3.9 Boot Image

3.9.1 Input Files

All previously generated files (FSBL, ATF, PMU, Bitstream, Linux Image, UBoot, FileSystem)

3.9.2 Output Files

Boot.bin

3.9.3 Tools

SDK (GUI) or Bootgen (command line).

3.9.4 Procedure Using Command Line

3.9.4.1 Create BIF file

You need to create configuration file in Boot Image Format (*.BIF). The BIF file lists the input files to the boot image, along with optional attributes for addressing and optional encryption, authentication or checksums.

To write this file manually, you can add like the following lines

```

1 // Boot Image Format file created for Zynq
2
3 My_Image: {
4     // [fsbl_config]a53_x64
5     /*
6      Note that value "a53_x64" is deprecated, you can specify the hosting PS in the partition section
7      for more info about fsbl config type bootgen -bif_help fsbl_config
8      */
9     [bootloader, destination_cpu = a53-0]..\InFiles\fsbl_a53.elf
10    [pmufw_image]..\InFiles\pmu_fw.elf
11    [destination_cpu = a53-0, exception_level = el-3, trustzone]..\InFiles\bl31.elf
12    [destination_cpu = a53-0, exception_level = el-2]..\InFiles\u-boot.elf
13    [destination_cpu = a53-0]..\InFiles\system.dtb
14    [destination_cpu = a53-0]..\InFiles\Image
15 }
```

Figure 3.31: BIF File, Windows Path

Note that here in this Figure 3.31, paths to elf files are specified according to windows format (with backslash). In case of Linux Operation Systems, these path shall be modified with slash (/).

For more information about available attributes that can be used within BIF file you can type

```
exec bootgen -bif_help
```

to get detailed view



```

xsc3% exec bootgen -bif_help

COMMON BIF ATTRIBUTES
-----
Zynq / ZynqMP Architectures:
[init]           : Register Initialization File (.ini file)
[aeskeyfile]     : AES Encryption Key File (.nky file)
[ppkfile]        : Primary Public Key (.pub, .txt, .pk1)
[pskfile]        : Primary Secret Key (.pen, .txt, .pk1)
[spkfile]        : Secondary Public Key (.pub, .txt, .pk1)
[sskfile]        : Secondary Secret Key (.pen, .txt, .pk1)
[spksignature]   : SPK Signature File (.sig file)
[headersignature]: Header Signature File (.sig file)
[bootimage]      : Boot Image File (in Xilinx Boot Image format)
ZynqMP Architecture:
[pmufw_image]    : PMU Firmware Image (.elf file)
[auth_params]    : Authentication Parameters
[keysrc_encryption]: Encryption Key Source
[boot_device]    : Secondary Boot Device
[fsbl_config]    : FSBL Configuration Parameters
[split]          : Splits the image
[bhsignature]    : Boot Header Signature File (.sig file)

PARTITION BIF ATTRIBUTES
-----
Zynq / ZynqMP Architectures:
bootloader       : First Stage Boot Loader
encryption       : Encryption for the partition
authentication    : Authentication for the partition
checksum         : Checksum for the partition
presign          : Partition Signature (.sig file)
udf_data         : User Defined Data in the Authentication Certificate
xip_mode         : eXecute In Place
partition_owner  : Owner of the partition
alignment        : Alignment for the partition
offset           : Offset of the partition in the boot image
reserve         : Reserve space for the partition in the boot image
load             : Load address of the partition in memory
startup          : Executable address of the partition in memory
ZynqMP Architecture:
destination_cpu  : Destination CPU for the partition
destination_device: Destination Device for the partition
trustzone       : Enable or Disable Trust Zone
early_handoff   : Enable Early Handoff
hivec           : Enable HiveC
exception_level  : Exception Level of the partition
blocks          : Blocks for Key Rolling Encryption
Note           : For more info on bif parameters, use the command
                  bootgen -bif_help <bif parameter name>
Example        : bootgen -bif_help aeskeyfile

xsc3%

```

Figure 3.32: BIF File Options

After completing the BIF file, you can generate boot image using bootgen command

```
bootgen -image sd_boot.bif -arch zynqmp -w -o C:\BOOT.bin
```

3.9.5 Procedure Using SDK

Open SDK

Choose Workspace

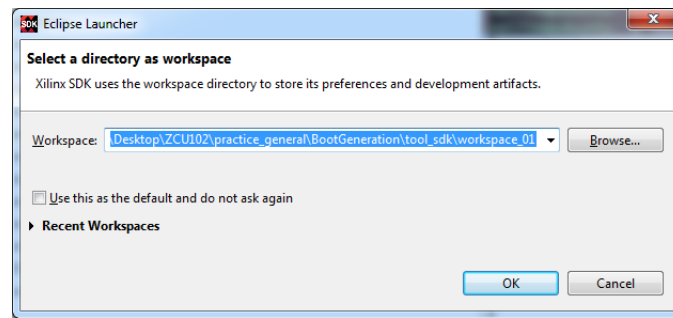


Figure 3.33: SDK Workspace [1]

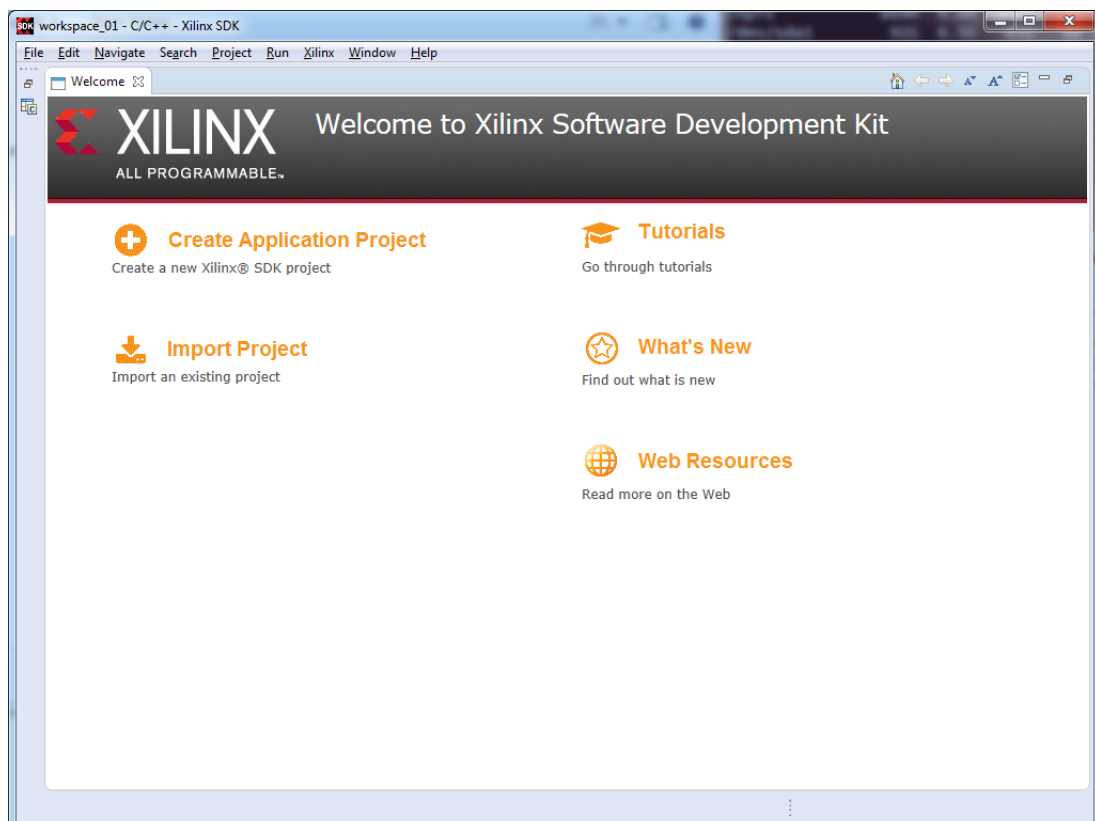


Figure 3.34: SDK Start Window [1]

Open Xilinx >> Create Boot Image
In Architecture, choose Zynq MP

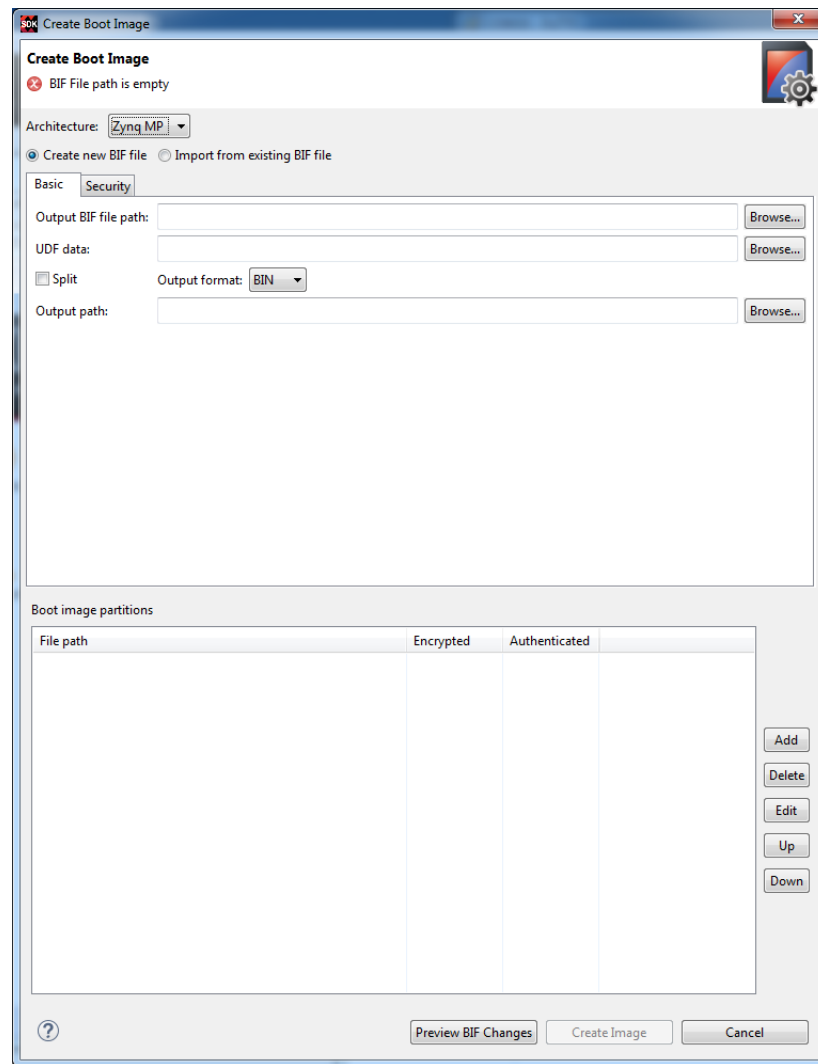


Figure 3.35: Create Boot Image [1]

In Output BIF File path browse to choose location of output files. Output path field will be updated accordingly. Then in Boot image partitions space click Add button to add your input files.

Click on Create Image button. BOOT.bin file shall be generated successfully in your output directory.

3.10 Boot Media

In order for the system to boot, created images (Kernel, File System, Device tree) shall be loaded into processing system RAM to start operation. To pass (or to load) these images into the RAM, the following approaches can be used.

3.10.1 Boot Through TFTP Server

This kind of connection allows transferring files between remote machine (tftp server) to a client machine. This method is then useful to have a centric booting files that can be deployed into cluster of FPGAs that need same booting files. Each FPGA will need to have its own first booting file stored internally, then this first booting file direct the boot sequence to get other booting files (Linux Kernel, Device tree, File System image) from the tftp server. Typical sequence can be like that

- Uboot is installed internally in the FPGA to start up after powering up the FPGA.
- Uboot is configured to read external boot script to follow instructions mention in that boot script.
- Booting script itself can be self-updated from the tftp server.
- Kernel, Device Tree, File System are described in the booting script how to be fetched as described in the following script example

```
setenv kernel_img_name "Image"
setenv kernel_loadaddr 0x6000000

setenv rootfs_img_name "rootfs.cpio.uboot"
setenv rootfs_loadaddr 0x10000000

setenv dtb_img_name "devicetree.dtb"
setenv dtb_loadaddr 0x6e00000

tftpboot $kernel_loadaddr $kernel_img_name;
tftpboot $rootfs_loadaddr $rootfs_img_name;
tftpboot $dtb_loadaddr $dtb_img_name;

booti $kernel_loadaddr $rootfs_loadaddr $dtb_loadaddr;
```

3.10.2 Boot Through SD Card

SD card can be used to contain the necessary booting files, these files are loaded eventually to processing system RAM. When booting from SD card is chosen, Boot.bin file is first loaded then it starts to search for other boot files (Kernel Image, Devicetree, Filesystem) in SD card in order to load it into system RAM.

3.10.2.1 SD Card Partitioning

If the SD card is going to be used as the boot medium, it has to be prepared to contain booting files and root file system. Booting files are files used to boot the system like boot.bin, Linux Image. Root file system is the directory structure of your linux system. Best way to do that is to make partitioning for the SD card so that it is formatted to have one partition for boot files and another partition for root file system.

In linux, partitioning the SD card can be performed using fdisk utility. The following script can be used for partitioning

```
#!/bin/sh -e
dd if=/dev/zero of=/dev/sdb bs=1024 count=1

(
echo x
echo h
echo 255
echo s
echo 63
echo c
echo
echo r
echo n
echo p
echo 1
echo 2048
echo +400M
echo n
echo p
echo 2
echo
echo a
echo 1
echo t
echo 1
echo c
echo t
echo 2
echo 83
echo p
echo "w"
) | fdisk /dev/sdb

sudo mkfs.vfat -F 32 -n boot /dev/sdb1
sudo mkfs.ext4 -L root /dev/sdb2
echo "please reboot....."
```

3.10.3 Booting Through QSPI

As a flash based storage system, this kind of memory can be used to store booting files. QSPI will be connected to the FPGA and configuration of the platform board will be set to load boot file from this QSPI.

Best practice to use QSPI, is to partition it to several partitions (five or six) allowing to store each boot file in a dedicated partition.

3.10.3.1 QSPI Partitioning

Partitions of the QSPI are set through the device tree file. U-Boot is referring to the device tree file when writing to the QSPI. For example, the following is defined in default device tree of Xilinx ZynqMP.

```
&qspi {
    status = "okay";
    is-dual = <1>;
    flash@0 {
        compatible = "m25p80"; /* 32MB */
        #address-cells = <1>;
        #size-cells = <1>;
        reg = <0x0>;
        spi-tx-bus-width = <1>;
        spi-rx-bus-width = <4>; /* FIXME also DUAL configuration possible */
        spi-max-frequency = <108000000>; /* Based on DC1 spec */
        partition@qspi-fsbl-uboot { /* for testing purpose */
            label = "qspi-fsbl-uboot";
            reg = <0x0 0x100000>;
        };
        partition@qspi-linux { /* for testing purpose */
            label = "qspi-linux";
            reg = <0x100000 0x500000>;
        };
        partition@qspi-device-tree { /* for testing purpose */
            label = "qspi-device-tree";
            reg = <0x600000 0x200000>;
        };
        partition@qspi-rootfs { /* for testing purpose */
            label = "qspi-rootfs";
            reg = <0x620000 0x5E0000>;
        };
    };
};
```

Figure 3.36: QSPI Partitions in Device Tree

Which depicts four partitions at addresses (0x0, 0x100000, 0x600000, 0x620000) and they are named (planned) for Uboot, Kernel, Devicetree and Filesystem respectively. The other value in reg parameter denotes the size of the partition.

Now in order to partition the QSPI, First step is specify the system RAM address where DTB file is stored (this means we need to load our device tree into the RAM first “our device tree means not the default U-Boot device tree”). In U-boot to specify the address of DTB file use the following command

```
fdt addr <DTB_addr>
```

If you want to change the partitions, you can use the following command

```
fdt set /amba/spi@ff0f0000/flash@0/partition@qspi-linux reg <0x00000000 ${bootimage_size}>
```

The general form of fdt set is

```
fdt set <path> <prop> [<val>]
```

[10]

So what we have done is, we have change the property “reg” of the linux partition defined as the shown path to a new value.

3.10.3.2 Writing on QSPI

For writing to the QSPI for the first time, you need external aid to boot from so that you can load first boot file into QSPI. Usually SD card is used for that and through U-boot prompt. To write into QSPI, when U-boot prompt is up type the following command to scan for the connected QSPI

```
sf probe
```

Boot files that are needed to be stored in the QSPI, shall be transferred from the system RAM. This is actually the available way in U-boot to write into the QSPI as it is not possible to write from the SD card. SD card is used only to bring up a working U-boot. This means that boot files have to be written first into the RAM, then write each boot file in its planned partition of the QSPI.

For loading boot files into system RAM, the usual way then is tftp. The following commands can be then issued in U-boot prompt to write to the QSPI

```
setenv autoload no
dhcp
setenv boot_img BOOT.BIN
setenv boot_img_addr 0x140000
setenv kernel_img_name "Image"
setenv kernel_loadaddr 0x6000000
setenv dtb_img_name "devicetree.dtb"
setenv dtb_loadaddr 0x6e00000

# mw is used to initialize the memory with '1'
# 0x2faf080 is estimation for the boot file size (50 MB)
# 0xe4e1c0 is estimation for the kernel file size (15 MB)
# 0xc350 is estimation for the device tree file size (50 KB)
mw.b $boot_img_addr 0xff 0x2faf080
mw.b $kernel_loadaddr 0xff 0xe4e1c0
mw.b $dtb_loadaddr 0xff 0xc350

tftpboot $boot_img_addr $boot_img
tftpboot $kernel_loadaddr $kernel_img_name
tftpboot $dtb_loadaddr $dtb_img_name
```

As just mentioned, next is to write each of the loaded file into its dedicated partition. Assume now the following partitions are initialized in the QSPI.

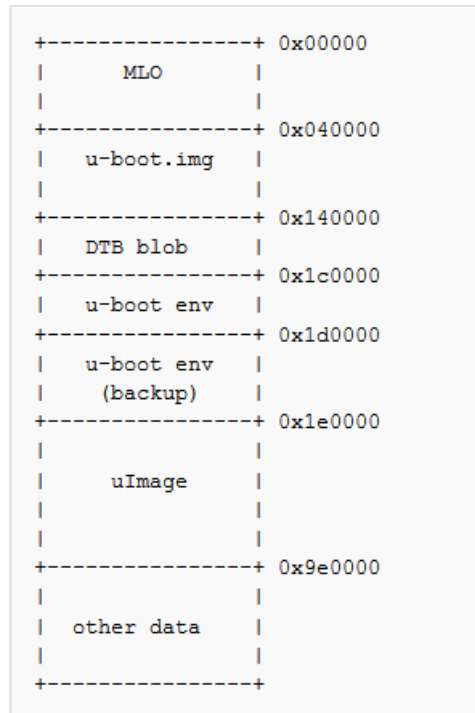


Figure 3.37: QSPI Memory Partitions

Start address of each partition is denoted in U-boot as offset. So for writing into the right partition we have to mention its offset in the command. The following commands can be used then

```
sf update $boot_img_addr 0x040000 0x2faf080
sf update $kernel_loadaddr 0x1e0000 0xe4e1c0
sf update $dtb_loadaddr 0x140000 0xc350
```

Previous commands were written in the shown order for declaration purposes. In fact, we should use it like the following

```
tftpboot $boot_img_addr $boot_img
sf update $boot_img_addr 0x040000 $filesize

tftpboot $kernel_loadaddr $kernel_img_name
sf update $kernel_loadaddr 0x1e0000 $filesize

tftpboot $dtb_loadaddr $dtb_img_name
sf update $dtb_loadaddr 0x140000 $filesize
```

The difference is in “filesize” variable which gets automatically the size of the loaded file, so that we can use it to write the exact size into the QSPI memory.

Booting from the QSPI then in subsequent operations (i.e. after QSPI has been written with boot files) can be using “sf read” command, for example

```
sf read $fdt_addr $fdt_offset $fdt_size
sf read $kernel_addr $kernel_offset $kernel_size
```

in which we read defined size from the defined offset of the QSPI into the defined address of the RAM.

3.10.4 Booting During Development Phases

As the system is not going to work from the first time, or at least many modifications may be applied, removing and inserting SD card to update booting files may not be practical approach. The more convenient approach is to have FPGA under development in connection with the development machine then transfer new updated files to replace old ones.

At this point, we have to distinguish between development files. For the case of applications running on the operating system, replacing the files while the system is running will be an easy task as the compiled binaries of the application will be only replaced (to some extent this is true).

3.10.4.1 Replacing Kernel, DTB, Filesystem Images

Other development files like Kernel Image, File System Image and Device tree still can be replaced but with the need of rebooting the system to let it boot with these new modified files. What is intended here is to keep system under connection with the development machine and find a way to update any of the booting files (like Kernel Image or File System Image or Device tree). This can be done by rebooting the system normally but halting it at u-boot stage. At this stage we can load modified files directly to the RAM using tftp.

To make this approach easier to use, it is recommended to configure u-boot to get loading instructions from external file. This external file is just a script written for U-Boot environment which is discussed previously in U-Boot section.

3.10.4.2 Replacing Boot File

U-boot and FSBL will be also modified frequently during development phases. Replacing the old files can't be performed directly. Instead, modified boot file will be stored temporarily in the system RAM, then we can replace and overwrite the old boot file from the temporarily location.

3.10.5 Loading Bitstream File

For designs that are using PL part of ZynqMP, a bitstream file is needed to be downloaded into the processor RAM which will program the PL part accordingly.

Normal or traditional way is to built-in the bitstream file into the boot file while creating it. Boot file is created using Xilinx tools (bootgen utility for instance) as described in previous section (Boot Image Generation) in which we list the bitstream file in BIF file to be included while creating boot.bin file. The bitstream file is then loaded into the FPGA when FSBL is loaded.

Another way is to load the bitstream file is through Linux operating system which is running already on the Processing System of ZynqMP. Xilinx provides one driver that can handle programming the PL part. This driver can be accessed through sysfs of Linux. Initial setting shall be performed first which is to represent the bitstream file (which is .bit file) into binary file (.bin file). Again Xilinx bootgen tool can be used for that as follows


```
bootgen -image bitstream.bif -arch zynqmp -process_bitstream bin
```

Where bitstream.bif is a BIF file which shall contain the following

```
all: {  
    [destination_device = pl] bitstream.bit  
}
```

Executing bootgen will result then in bistream.bit.bin file. After this setting, The generated bin file shall be stored exactly in a path like the following

/lib/firmware/bitstream.bit.bin

This Xilinx driver is using this path to identify the target bitstream files. Last step is to execute the following commands

```
$ echo 0 > /sys/class/fpga_manager/fpga0/flags  
$ echo bitstream.bit.bin > /sys/class/fpga_manager/fpga0/firmware
```

[11]

Another possible way of loading the bitstream file is to use U-boot for that. U-Boot has “fpga command” that can be used to program the attached PL. In this case the bitstream file has to be represented into binary file also as described previously. The command to use then in U-boot is as follows

```
fpga load 0 $fpga_loadaddr $filesize
```

where 0 is number of FPGA device.

Chapter 4

Software

Controlling processing system and its attached devices can be achieved mainly through Operating Systems or Bare-Metal applications. In both cases, the developer has to define some firmware that directs running software for how to work with and control the attached hardware device. This firmware is what we call a driver for the device and it is what we need to build in the path between hardware interface and end user interface.

4.1 Linux Drivers

In case of Linux system, the path between user and the hardware starts at the user side in which we need to have simple and easy to use program or application to work with. This application is communicating with device drivers that reside in the kernel space. Then the kernel layer comes afterward to provide mechanisms and firmware drivers of the hardware. End of this path is at the hardware interface which is seen by the kernel using the device tree file.

Since the hardware devices differ in functionality, properties and attributes, the driver shall be tailored to deal with this device. Device properties shall be communicated correctly between hardware developers and drivers developers. For the case of FPGA development, it can happen that hardware developer needs to build the hardware logic and its driver. The following guide may help then to clarify how to build driver in Linux kernel.

We can start by simple ‘hello world’ example to have an idea how a kernel module or driver can be built. This example is based on Xilinx ZynqMP chip.

4.1.1 Build and Run Linux Kernel

First, Linux kernel is needed in order to compile the module. Linux kernel for Xilinx can be downloaded by

```
$ git clone https://github.com/Xilinx/linux-xlnx.git
```

After downloading this kernel and building it, make sure that it can successfully run on your Xilinx Zynq.

4.1.2 Create Module

Simple ‘Hello World’ module can be built using the following C code

```
#include <linux/init.h>
#include <linux/module.h>

static int hello_init(void) {
    printk("Hello, ZynqMP world!\n");
    return 0;
}

static void hello_exit(void) {
    printk("Goodbye, ZynqMP world!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("FPGA Developer");
MODULE_DESCRIPTION("\"Hello, Zynq world!\" minimal module");
MODULE_VERSION("printk");
```

This module is based on basic entry and exit functions (hello_init & hello_exit) which are called by module_init & module_exit respectively. module_init & module_exit are invoked automatically when the module is installed in Linux kernel and they are defined in module.h header.

4.1.3 Build Module

To compile and build the module, a Make file will be used for that. First you have to take into account that this module is built for the Linux running on Xilinx Zynq chip. So kernel source of this Linux has to be pointed to. The following Makefile can be used to build the kernel

```
obj-m := hello_printk.o
PWD := $(shell pwd)
all:
    $(MAKE) -C $(KERNEL) M=$(PWD) modules
clean:
    $(MAKE) -C $(KERNEL) M=$(PWD) clean
```

When running this Makefile it should be executed as

```
$ make ARCH=arm64 KERNEL=/path/to/built/kernel CROSS_COMPILE=/path/to/
CrossCompiler
```

The following shell script can be easily used for doing everything

```
#!/bin/sh -e
export Current_Dir=$PWD
export CROSS_COMPILE=/path/to/CRS_COMP/aarch64-linux/bin/aarch64-linux-gnu-
```

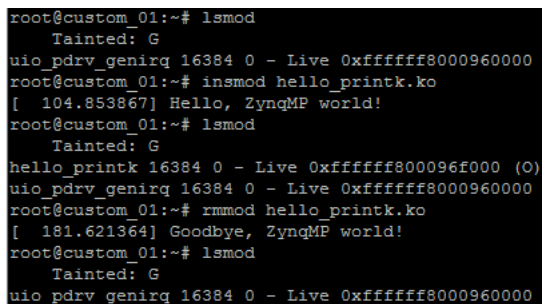
```
export KERNEL=/path/to/Linux/build
export Mod_Dir=/path/to/module/hello_example/

cd $Mod_Dir
make ARCH=arm64
cd $Current_Dir
```

4.1.4 Running Module

Build process will result in `hello_printk.ko` file. Copy this file into the SD card containing booting files and start the system.

To install the module use `insmod` command. To remove it use `rmmod`. To see list of available modules use `lsmod` command.



```
root@custom_01:~# lsmod
Tainted: G
uio_pdrv_genirq 16384 0 - Live 0xffffffff8000960000
root@custom_01:~# insmod hello_printk.ko
[ 104.853867] Hello, ZynqMP world!
root@custom_01:~# lsmod
Tainted: G
hello_printk 16384 0 - Live 0xffffffff800096f000 (O)
uio_pdrv_genirq 16384 0 - Live 0xffffffff8000960000
root@custom_01:~# rmmod hello_printk.ko
[ 181.621364] Goodbye, ZynqMP world!
root@custom_01:~# lsmod
Tainted: G
uio_pdrv_genirq 16384 0 - Live 0xffffffff8000960000
```

Figure 4.1: Installing Linux Module

If `printk` output is not displayed, try to use “`dmesg | grep world`” command.

4.2 Linux Drivers Chain

Previous steps are basic steps followed for having a kernel module. Because previous example has printed simple message for the user, it is not connected to any hardware device yet. Still we need to clarify the case of a module interacting with attached hardware devices. In this case, there is a chain of libraries and functions that are used to create the module. In order to organize and represent these libraries and functions, we will represent the kernel module in form of chain of devices and drivers as shown in figure 4.2 then we will elaborate more about each part.

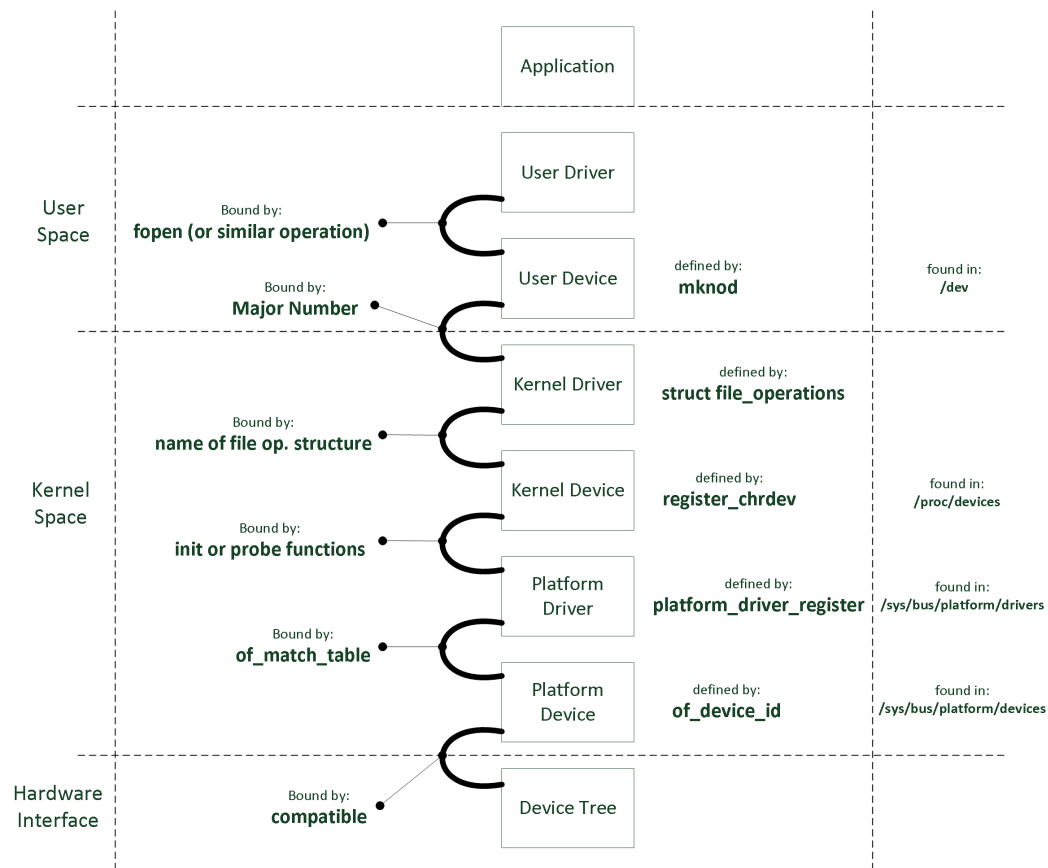


Figure 4.2: Linux Driver Chain

This figure is a trial for simplifying basic steps needed to build a Linux driver. In fact there are no references or resources mentioning a ‘Kernel Device’ and a ‘User Device’ for instance. But there is what is called ‘Platform Device’. So the author used this illustration to organize many drivers declarations.

What can be indicated in this figure is that there are User Space and Kernel Space in Linux system. As its name reflects, User Space is the place where is user can access his own functions. User is not allowed to access functions defined in Kernel Space. There are some specific functions that can be used to handle interactions between both spaces that will be declared later.

Figure 4.2 indicates also that a Linux driver is built as a chain of entities. Each entity is bound with the other with a keyword or attribute so user application can access the hardware device through this chain using specified keyword between each entity.

4.2.1 Platform Hardware Link

Device tree is the file describing attached hardware in the running system. Linux kernel is using this file to get some information about these attached hardware devices like registered address, width of input bus and some other parameters. One of these parameters is the “compatible” parameter. It is developer defined string to mark the hardware. Kernel developer is using this string also to define a Platform Device. Linux kernel provides `of_device_id` structure which can be used to store data about the hardware device. It is

defined in `/include/linux/mod_devicetable.h` as follows

```
/*
 * Struct used for matching a device
 */

struct of_device_id {
    char    name[32];
    char    type[32];
    char    compatible[128];
#ifdef __KERNEL__
    void    *data;
#else
    kernel_ulong_t data;
#endif
};
```

What is shown in this structure that it is designed to contain strings and data about the device. Usually, most of developers are using only “compatible” parameter to match it with the “compatible” parameter defined in the device tree file. Both “compatible” parameters shall have the same value, otherwise kernel will fail to match them. As a sample of how this `of_device_id` is defined, see the following example

```
Static const struct of_device_id User_IO_HW[] = {
{.compatible = "xlnx,user_io"},
{}
};

MODULE_DEVICE_TABLE(of, User_IO_HW);
```

The platform is defined here as array of type `of_device_id` without defining the size of the array. This is usual approach for defining platform device to allow developer to link as much hardware devices as he wants to this platform device. Eventually, this structure shall be terminated with a null member `{}`. One important function shall be used in the driver is “`MODULE_DEVICE_TABLE`” which is performing the matching between defined platform device and available hardware devices.

Once platform device is defined, it needs platform driver to drive it. Using `linux/platform_device.h` we can build this driver using structure `platform_driver` in which we can define what functions shall be executed when the device is detected (probe function), what shall be executed when device is removed (remove function) and some other details about the driver. The following is an example of the platform driver.

```
static struct platform_driver User_plfrm_drv = {
    .probe = gpio_probe,
    .remove = gpio_remove,
    .driver = {
        .name = "userA_driver",
        .owner = THIS_MODULE, //this macro is defined in module.h
        .of_match_table = id User_IO_HW,
    },
};
```

Name of the platform driver in this example is `userA_driver`, when linux runs this name can be found in `/sys/bus/platform/drivers`. It is optional to define the “owner” property but it is important to set value of “`of_match_table`” to the name of the platform device as this is how the platform driver is bound with the platform device. Platform drivers need to be registered after it have been defined. This can be done using `platform_driver_register`.

4.2.2 Kernel Platform Link

Upon detection of the hardware device and defining its corresponding platform device, this device needs to be registered in the kernel. It is like make an existence for the device in the kernel. This registration can be done using several functions or at different steps. For instance the device registration can be called immediately when the device is detected (define it in probe function), or it can be defined upon module installation (define it in init function). This is because this registration step is independent actually on the device. You can even register a device that doesn’t actually exist. This is possible as long as you perform valid operations on the registered device. In Linux, devices are classified as Block devices or Character devices. Block devices simply are devices built for storage purposes. Character devices are devices handling IO streaming. Therefore, there are many functions to register the device according to its type. For instance there are `alloc_chrdev`, `alloc_netdev`, `register_blkdev`, `register_chrdev`, etc. For example here we use function `register_chrdev` for illustration.

```
regdev_ret = register_chrdev(89, "Dev_Name", &gpio_fops);
```

This function takes three arguments; first one is called Major Number which will be discussed soon. In this example it is chosen 89. Second argument is a name for the device. Developer is free to choose any name since it is not bound to other objects in the module. Third argument is a name of a structure used for building the kernel driver. Upon success, this function will return 0.

Kernel Driver means here a set of functions used to perform necessary operations on the device. This set of functions is defined using structure `file_operations` type which is defined in `linux/fs.h`. For example the following structure defines basic operations that can be done on the device.

```
struct file_operations fops = {
    read: device_read,
    write: device_write,
    open: device_open,
    release: device_release
};
```

Which are open, read, write and close. This is because everything in Linux is defined as files, so using this structure we can tell the kernel what to do when he opens, reads, writes or closes the device. Developer has to define these functions and describe what shall be executed when these functions are called.

Needed setting and configuration in kernel space will be ready if previous entities were defined correctly. Remaining now to make some preparations in user space. The device has

to be instantiated also in user space. This is important as the final user space application should be in contact with a device. Devices in `/dev` directory are built for that purpose. We can instantiate our device then in user space using `mknod` command if the device has not been instantiated automatically by Linux.

So if the device has not been instantiated, display the file `/proc/devices` then find the Major Number that you have used while registering the kernel device. Beside this number you can find the name of the device as defined in the kernel device (lets assume it is `Dev_Name`). Use the following command to instantiate the device.

```
mknod /dev/Dev_Name -c 89 0
```

This `mknod` command will instantiate the device at `/dev` directory with the name `Dev_Name`. The switch `-c` directs the command to instantiate it as a character device. 89 is the Major Number assigned to the device in the kernel space. 0 is the Minor Number of the device.

4.2.3 Major Number

It is a number used by driver developer to identify which device is associated with the driver as shown in previous declaration. If the driver is managing different devices, there is what is called Minor Number which is used to differentiate between those different devices that are driven with one driver. It is not commonly used.

Now it comes the point, how to pick a Major Number to assign it to the registered device. There is one possibility when registering the device that we can register it dynamically. This means that we can let the kernel allocates this number based on available numbers. In this way we can avoid having conflict with another used Major Number. The dynamic allocation is performed when registering the device with Major Number 0. For example to be as follows

```
regdev_ret = register_chrdev(0, "Dev_Name", &gpio_fops);
```

The return value of this dynamic allocation will be the assigned Major Number. If you want to unregister the device in the driver code you can use the generate number.

```
unregister_chrdev(regdev_ret, "Dev_Name");
```

If you used dynamic allocation, you can't search for the registered device then in file `/proc/devices` with the Major Number as you don't know which number the kernel has generated. In this case you can search for it using the defined name ("`Dev_Name`" for instance). You can use a script like the following to get this number

```
#!/bin/sh -e

ModuleName="UserMod"

MajorNum=$(awk ' $2 ~ /"$ModuleName"/ {print $1}' /proc/devices)

mknod /dev/$ModuleName c $MajorNum 0
```


4.2.4 User Driver

So far, device has been recognized, registered and instantiated. Remaining now to build a User Driver that can handle user commands and transfer it through kernel driver to kernel space which will be executed then through platform driver in the platform device. User has no access to kernel space and functions, therefore there are special functions that can receive user commands and transfer it to kernel space. To use these functions, we have to be aware first that user can handle the instantiated device in Linux as a file. There is a direct mapping in Linux between file operations in user space and struct file_operations defined in the Kernel Driver. This means using if user opened the device file in user space, the open function of the struct file_operations will be called. Closing the file, the release function will be called. Writing to the file, the write function will be called. Reading from the file, the read function will be called.

In user space, to build the driver then we use standard fopen, fwrite, fread, fclose functions. The struct file_operations is defined as follows in Linux

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
        long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long
        , loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned
        long, loff_t *);
};
```

As shown, the write function for example shall be defined in this form

```
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
```

where first argument is pointer to user file that is opened and written to. Second argument is pointer to the string has be written in the file. Third argument is how many bytes shall be taken from the written string. Fourth argument is the offset to start take the bytes. The user in kernel driver has to define a write function that tells the kernel what to do when user space writes to the file. To clarify it more assume the following write function defined in Kernel Driver

```
ssize_t gpio_set
```

```

        (struct file *filep, const char *buf, size_t count, loff_t *f_pos)
        {
            printk("GPIO Kernel Driver: User has written some value \n"
                );
        }

```

In the Kernel Driver also the struct `file_operations` shall be defined in something like the following

```

struct file_operations gpio_fops = {
    read: gpio_get,
    write: gpio_set,
    open: gpio_open,
    release: gpio_close
};

```

As mentioned before, User has no access to kernel space. We have seen the mapping between file operations in user space and kernel space. Remaining now to understand how to get the value that user has written or send to user value recorded by the device. There are special functions that transfer needed values between user space and kernel space. These functions are defined in the library `asm/uaccess.h`. In this library you find some functions like `copy_from_user`, `copy_to_user`, `put_user`, `strncpy_from_user`, etc. Using these functions, we can transfer the content of the device file to/from user space. Assume the following example for the write function in Kernel Driver.

```

ssize_t gpio_set
    (struct file *filep, const char *buf, size_t count, loff_t *f_pos)
    {
        int user_value;
        printk("GPIO Kernel Driver: User has written some value \n"
            );
        copy_from_user(&user_value, buf, count);
        printk("GPIO Kernel Driver: user has written %c \n",
            user_value);
    }

```

If the user has defined the following User Driver

```

FILE * wr_d;
char Letter = 'A';
wr_d = fopen("/dev/Dev_Name", "w");
fwrite(&Letter, 1, 1, wr_d);
fclose(wr_d);

```

The result of the Kernel Driver then shall be:

```

GPIO Kernel Driver: User has written some value
GPIO Kernel Driver: user has written A

```

Figure 4.3 depicts the mapping between User Driver and Kernel Driver to clarify it more.

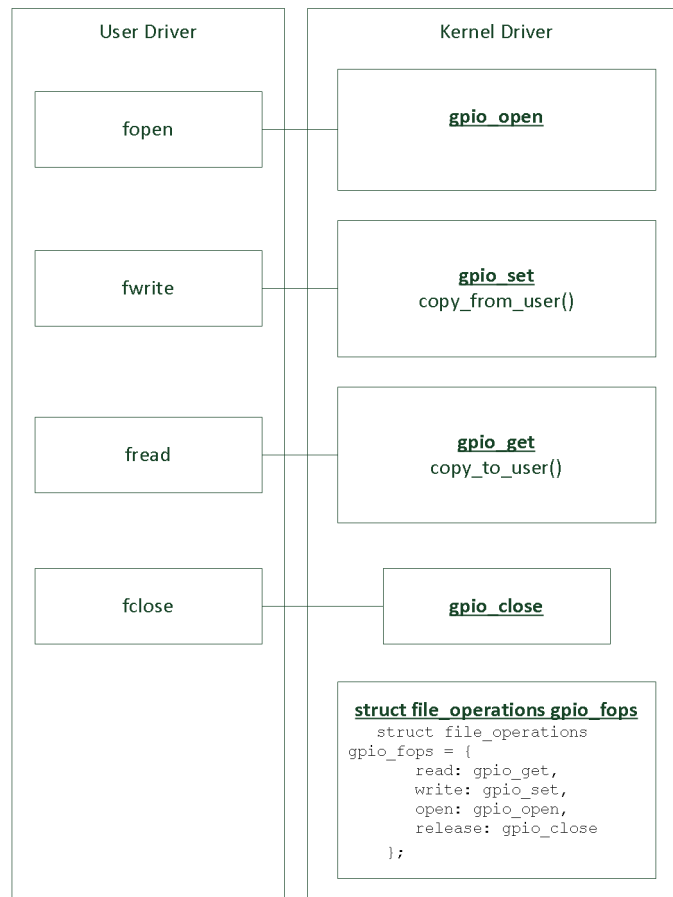


Figure 4.3: Kernel Driver Mapping

4.3 Driver Integration

Now that we have the driver is ready and has been tested manually (using insmod). Knowing how to include this driver in the final system is next step that we have to learn. To integrate your driver with the running Linux, there are two approaches; either integrating it with Linux Kernel source files or integrate it with Filesystem as Buildroot package.

4.3.1 Integration with Kernel Image

Linux Kernel source file has one directory for including drivers; it is “drivers”. All what we need is to prepare our driver along with its Makefile and Kconfig file, then indicate in the Makefile and Kconfig files of “drivers” directory to include our driver during Kernel compilation process. Makefile of the driver shall contain the following line

```
obj-$(CONFIG_OUR_GPIO) += <driver_file_name>.o
```

Kconfig of the driver shall contain the following

```
config OUR_GPIO
    tristate "Our driver for GPIO peripheral"
    help
```

```
my driver module.
```

Tristate means that the configuration field can be marked as y “yes”, n “no”, m “modularize”. Choosing no means driver will not be compiled, y means it will be compiled, m means it will be compiled but as a separate module. i.e. not included in the final kernel Image. As alternative, we can define the configuration field as Boolean which take only two values y or n

```
bool "enable our GPIO driver"
    default y
```

We can then define default value for the configuration as declared. These three files shall be included then in one directory for instance our_driver <Linux_Source_dir>/drivers/<our_driver>.

Next, in “drivers” directory we need to mention that there is new driver shall be ready for compilation. Open <Linux_Source_dir>/drivers/Makefile then add the following line at end of file

```
obj-$(CONFIG_OUR_GPIO) += our_driver /
```

Then open <Linux_Source_dir>/drivers/Kconfig and add the following line before “endmenu”

```
source "drivers/our_driver /Kconfig"
```

The driver is now ready for compilation. Before starting main kernel compilation, you can choose to include this driver with the kernel or not. This can be performed using menuconfig interface or insert “CONFIG_OUR_DRIVER=y” in the loaded configuration file (***_defconfig). Note that if the driver configuration file is defined as tristate, here you can set CONFIG_OUR_DRIVER to y or n or m.

4.4 GPIO Driver

There are many techniques to deal with the GPIO devices in Linux. Three of them are discussed here.

4.4.1 Using sysfs

Since everything in Linux is handled as a file, sysfs is a mechanism that is representing attached hardware devices as files for the user in order to control it. Sysfs can be considered as a file system that is mounted automatically (in most linux systems) by the kernel. This sysfs represents attached devices in form of files which means, it exports these devices from the binary representation in the kernel into the user interface in order to view and manipulate it. Figure 4.4 shows mainly the directory structure of sysfs. At the top level of the sysfs mount point are a number of directories. These directories represent the major subsystems that are registered with sysfs. These directories are created at system startup when the subsystems register themselves with the kobject core.

```

/sys/
|-- block
|-- bus
|-- class
|-- devices
|-- firmware
|-- module
`-- power

```

Figure 4.4: Sysfs Structure [12]

The block directory contains subdirectories for each block device that has been discovered in the system. The bus directory contains subdirectories for each physical bus type that has support registered in the kernel (either statically compiled or loaded via a module).

The class directory contains representations of every device class that is registered with the kernel. A device class describes a functional type of device. For example figure 4.5 shows content of class directory, you can find then that all registered devices are gathered here according to its functions (block, gpio, graphics, power_supply, tty, etc). Contents of this directory are symbolic link to its location in the devices directory.

```

root@xilinx-zcu102-2017_3:/sys/class# ls
ata_device  bsg          fpga_region  ieee80211    misc          power_supply  scsi_device  udc
ata_link    devcoredump  gpio         input        mmc_host      pps           scsi_disk    uio
ata_port    dma          graphics     iommu        mtd           ptp           scsi_host    vc
bdi         drm          hwmon       leds         net           regulator     sound        video4linux
block       fpga_bridge  i2c-adapter  mdio_bus     pci_bus       rfkill        spi_master   vtconsole
bluetooth   fpga_manager i2c-dev      mem          phy           rtc           tty          watchdog

```

Figure 4.5: Sysfs Class Directory

The Devices directory contains the global device hierarchy. This contains every physical device that has been discovered by the bus types registered with the kernel. There are two types of devices that are exceptions to this representation: platform devices and system devices. Platform devices are peripheral devices that are inherent to a particular platform. They usually have some I/O ports, or MMIO, that exists at a known, fixed location. Examples of platform devices are GPIO controllers, Serial controllers. System devices are non-peripheral devices that are integral components of the system. In many ways, they are nothing like any other device. Examples of system devices are CPUs, APICs, and timers.

To use sysfs for dealing with GPIO, the gpio device shall be exported i.e. represented to the user interface in form of file. In the `/sys/class/gpio` directory you can find hardware nodes of the registered devices in the device tree source file as shown in figure 4.6.

```

root@xilinx-zcu102-2017_3:/sys/class/gpio# ls -l
total 0
--W----- 1 root    root      4096 Mar  1 08:23 export
lrwxrwxrwx 1 root    root        0 Mar  1 08:23 gpiochip293 -> ../../devices/platform/amba/ff020000.i2c/i2c-1/1-0021/gpio/gpiochip293
lrwxrwxrwx 1 root    root        0 Mar  1 08:23 gpiochip309 -> ../../devices/platform/amba/ff020000.i2c/i2c-1/1-0020/gpio/gpiochip309
lrwxrwxrwx 1 root    root        0 Mar  1 08:23 gpiochip325 -> ../../devices/platform/amba/ff0a0000.gpio/gpio/gpiochip325
lrwxrwxrwx 1 root    root        0 Mar  1 08:23 gpiochip499 -> ../../devices/platform/amba_pl180/80020000.gpio/gpio/gpiochip499
lrwxrwxrwx 1 root    root        0 Mar  1 08:23 gpiochip507 -> ../../devices/platform/amba_pl180/80010000.gpio/gpio/gpiochip507
--W----- 1 root    root      4096 Mar  1 08:23 unexport

```

Figure 4.6: Sysfs GPIO Nodes

Using the ‘export’ file we can export the hardware in form of file system using the following First we have to know assigned base address in the kernel for our target GPIO

```
$ cat gpiochip499/base
```

In our example the result will be 499. Then we need to export this device

```
$ echo 499 > /sys/class/gpio/export
```

You can find then that the gpio device has been exported as file system.

```
root@xilinx-zcu102-2017_3:/sys/class/gpio# echo 499 > /sys/class/gpio/export
root@xilinx-zcu102-2017_3:/sys/class/gpio# ls -l
total 0
--w----- 1 root root 4096 Mar 1 08:37 export
lrwxrwxrwx 1 root root 0 Mar 1 08:37 gpio499 -> ../../devices/platform/amba_pl080/80020000.gpio/gpiochip1/gpio/gpio499
lrwxrwxrwx 1 root root 0 Mar 1 08:23 gpiochip293 -> ../../devices/platform/amba/ff020000.i2c/i2c-1/1-0021/gpio/gpiochip293
lrwxrwxrwx 1 root root 0 Mar 1 08:23 gpiochip309 -> ../../devices/platform/amba/ff020000.i2c/i2c-1/1-0020/gpio/gpiochip309
lrwxrwxrwx 1 root root 0 Mar 1 08:23 gpiochip325 -> ../../devices/platform/amba/ff0a0000.gpio/gpio/gpiochip325
lrwxrwxrwx 1 root root 0 Mar 1 08:23 gpiochip499 -> ../../devices/platform/amba_pl080/80020000.gpio/gpio/gpiochip499
lrwxrwxrwx 1 root root 0 Mar 1 08:23 gpiochip507 -> ../../devices/platform/amba_pl080/80010000.gpio/gpio/gpiochip507
--w----- 1 root root 4096 Mar 1 08:23 unexport
```

Figure 4.7: Exported GPIO Device

Note that you can use unexport file to revert back importing this device. Finally what we need then is to set direction of the pin either out or in (pin 499) in direction file. Then write which value we need to write in value file as shown in figure 4.8.

```
root@xilinx-zcu102-2017_3:/sys/class/gpio# echo out > /sys/class/gpio/gpio499/direction
root@xilinx-zcu102-2017_3:/sys/class/gpio# echo 1 > /sys/class/gpio/gpio499/value
```

Figure 4.8: Sysfs Writing to GPIO

Note that in previous steps we had imported pin no. 499 of the device gpiochip499. In order to know how many pins this device has, we can check that by the following

```
root@xilinx-zcu102-2017_3:/sys/class/gpio# cat /sys/class/gpio/gpiochip499/ngpio
8
```

Figure 4.9: Sysfs GPIO Pins

In our case here, the device has 8 pins. First pin has no. 499. If you need to control fourth pin for instance, then you have to export this pin and use the same sequence as shown in figure 4.10.

```
root@xilinx-zcu102-2017_3:/sys/class/gpio# echo 502 > /sys/class/gpio/export
root@xilinx-zcu102-2017_3:/sys/class/gpio# echo out > /sys/class/gpio/gpio502/direction
root@xilinx-zcu102-2017_3:/sys/class/gpio# echo 1 > /sys/class/gpio/gpio502/value
```

Figure 4.10: Sysfs Writing to Pin

To read a value from the pin same sequence can be done but with direction in

```
root@xilinx-zcu102-2017_3:/sys/class/gpio# echo 507 > /sys/class/gpio/export
root@xilinx-zcu102-2017_3:/sys/class/gpio# echo in > /sys/class/gpio/gpio507/direction
root@xilinx-zcu102-2017_3:/sys/class/gpio# cat /sys/class/gpio/gpio507/value
0
root@xilinx-zcu102-2017_3:/sys/class/gpio# cat /sys/class/gpio/gpio507/value
1
```

Figure 4.11: Sysfs Reading GPIO

4.4.2 Using GPIOLIB

In Linux, there is one library that can be used as c-based library to get and set values of the GPIO pins. To use these libraries, we have to know first that this library is designed to be used in kernel space. That is because it needs some information about the GPIO device that it is going to communicate with. This information will be caught from definition of the GPIO node in the device tree file. So to work with this library, device tree file shall be configured to provide needed information for the GPIOLIB. This library has sufficient information in its documentation [13].

What we can discuss here is showing usage of this library in basic steps.

4.4.2.1 Prepare Device Tree file

In the device tree file. Define the gpio pins as follows

```
gpio_node {
    compatible = "xlnx,org-gpio-1.00.a";
    status = "okay";
    org-gpios = <&axi_gpio_1 0 0>,
                <&axi_gpio_1 1 0>,
                <&axi_gpio_1 2 0>,
                <&axi_gpio_1 3 0>,
                <&axi_gpio_1 4 0>,
                <&axi_gpio_1 5 0>,
                <&axi_gpio_1 6 0>,
                <&axi_gpio_1 7 0>;
};
```

In this way, the GPIOLIB will consider group of pins (8 pins) that are attached to the device node `axi_gpio_1`. This means that the main device `axi_gpio_1` shall be defined also in the device tree file. This group of gpio pins will be named as `org`. So we have to stick to this format when defining name of the gpio pins: “`gpio_name-gpios`”.

4.4.2.2 Recognize GPIO Pins in the Platform Driver

When defining the probe function of the platform driver, we can define our gpio pins using `gpiod_get` functions. In case of group of pins we can use as in the following function

```
static struct gpio_descs *our_gpio;
static int gpio_probe(struct platform_device *pdev) {

    int gpio_width;
    struct device *dev = &pdev->dev;
    printk(KERN_ALERT "GPIO: Starting Module\n");
    our_gpio = gpiod_get_array(dev, "org", GPIOD_OUT_LOW);

    if (IS_ERR(our _gpio)) {
        printk("GPIO: failed to acquire device\n");
    } else {
        gpio_width = our_gpio->ndescs;
        printk("GPIO: found %d elements\n", gpio_width);
    }
}
```

```

    }
    return 0;
}

```

4.4.2.3 Writing to the GPIO

The following function can be used to write for the GPIO

```

gpiod_set_array_value(our_gpio->ndescs, moi_gpio->desc, value_arr);

```

4.4.3 Using Generic UIO

This is a general driver that can be used for many peripherals. Its idea is based on writing to the assigned memory address of the peripheral. According to data written to the peripheral registers, the peripheral shall react. The advantage is end user doesn't have to know the address of the peripheral to write into it, instead, he can write to a dedicated file to pass data to the peripheral. The generic UIO is importing for each peripheral using this driver a file to the user space so that he can use this file for passing the needed data. The following lines discuss how to use this generic User IO driver with the hardware peripheral.

4.4.3.1 Define Driver in Hardware Node

As any other driver, the driver ID shall be defined in the 'compatible' property of the peripheral in the device tree file. As a convention, the string "generic-uio" shall be used as the name of the driver.

```

gpio_node {
    compatible = "generic-uio";
    status = "okay";
};

```

4.4.3.2 Enable UIO Driver in Kernel Configuration

The driver shall be built with the kernel so that it can be used as part of the system. In kernel configuration file define the following attributes

```

CONFIG_UIO=y
CONFIG_UIO_PDRV_GENIRQ=y
CONFIG_UIO_DMEM_GENIRQ=y

```

Alternatively, you can choose these option through menuconfig before building the kernel

```

"Device Drivers" --->
|- "Userspace I/O drivers" --->
|- <*> Userspace I/O platform driver with generic IRQ handling
|- <*> Userspace I/O OF driver with generic IRQ handling

```


4.4.3.3 Set Driver Identification in Kernel

Although we have defined the driver while building the kernel, the string that we used in property “compatible” in device tree is not yet identified in the kernel driver. So we need to configure the UIO driver to use this string “generic-uio”. This step has to be done before booting the kernel. So we can pass it to the kernel as part of the bootargs. To do that, the following bootargs has to be used

```
bootargs = "earlycon clk_ignore_unused uio_pdrv_genirq.of_id=generic-uio";
```

in which we added “uio_pdrv_genirq.of_id=generic-uio”. This string will change the id of the driver to match the string that we defined in compatible property.

When the system boots, the UIO device is represented in the filesystem as “/dev/uioN” where N is an incrementing integer value for each separate UIO device. To know which uioN corresponds to which device, we can use

```
$cat /sys/class/uio/uio$N/name
```

note that \$N shall be replaced with the right assigned number.

Alternatively, we can use ‘lsuio’ utility to list all the devices that are binded to generic UIO driver.

```
$lsuio
```

4.4.3.4 Writing to UIO

In order to write files imported for each device in /dev directory, we can use one utility called ‘memtool’.

```
##To read
memtool md -s /dev/uio$N <offsetAddress>
##To write
memtool mw -d /dev/uio$N <offsetAddress> <data>
```

Note again that \$N represents the uio number assigned to the device.

Also another tool can be used like ‘devmem’ which also has similar syntax.

Bibliography

- [1] Xilinx Vivado and SDK Programm User Interface
- [2] https://elinux.org/Device_Tree_Usage
- [3] <https://www.kernel.org/doc/Documentation/devicetree/bindings/xilinx.txt>
- [4] <http://www.wiki.xilinx.com/Arm+Trusted+Firmware>
- [5] <https://emreboy.wordpress.com/2012/12/20/building-a-root-file-system-using-busybox/>
- [6] <http://xilinx.wikidot.com/zynq-rootfs>
- [7] http://www.etalabs.net/compare_libcs.html
- [8] https://mirrors.edge.kernel.org/pub/linux/libs/uclibc/Glibc_vs_uClibc_Differences.txt
- [9] https://elinux.org/images/f/f4/Elc2013_Fernandes.pdf
- [10] <http://www.wiki.xilinx.com/U-Boot+Flattened+Device+Tree>
- [11] <http://www.wiki.xilinx.com/Solution+ZynqMP+PL+Programming>
- [12] <https://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf>
- [13] <https://www.kernel.org/doc/Documentation/gpio/>