BeagleV-Fire

Table of contents

BeagleV®-Fire is a revolutionary SBC powered by the Microchip's PolarFire® MPFS025T RISC-V System on Chip (SoC) with FPGA fabric. BeagleV®-Fire opens up new horizons for developers, tinkerers, and the open-source community to explore the vast potential of RISC-V architecture and FPGA technology. It has the same P8 & P9 cape header pins as BeagleBone Black allowing you to stack your favorite BeagleBone cape on top to expand it's capability. Built around the powerful and energy-efficient RISC-V instruction set architecture (ISA) along with its versatile FPGA fabric, BeagleV®-Fire SBC offers unparalleled opportunities for developers, hobbyists, and researchers to explore and experiment with RISC-V technology.
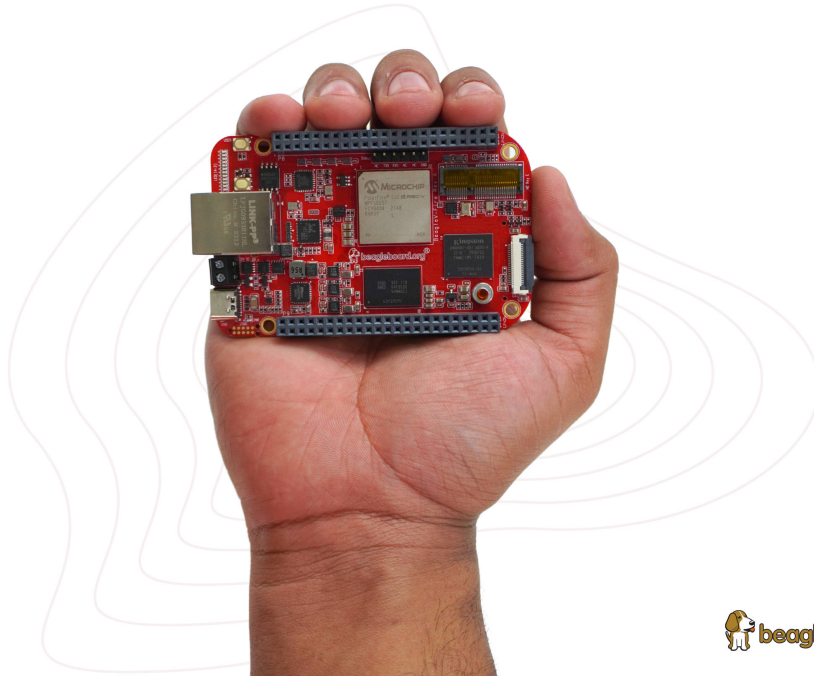
Chapter 1

Introduction

BeagleV®-Fire is a revolutionary SBC powered by the Microchip's PolarFire® MPFS025T System on Chip (SoC) with 4x RV64GC Application cores, 1x RV64IMAC monitor/boot core, and FPGA fabric. BeagleV®-Fire opens up new horizons for developers, tinkerers, and the open-source community to explore the vast potential of RISC-V architecture and FPGA technology. It has the same P8 & P9 cape header pins as BeagleBone Black allowing you to stack your favourite BeagleBone cape on top to expand it's capability. Built around the powerful and energy-efficient RISC-V instruction set architecture (ISA) along with its versatile FPGA fabric, BeagleV®-Fire SBC offers unparalleled opportunities for developers, hobbyists, and researchers to explore and experiment with RISC-V technology.
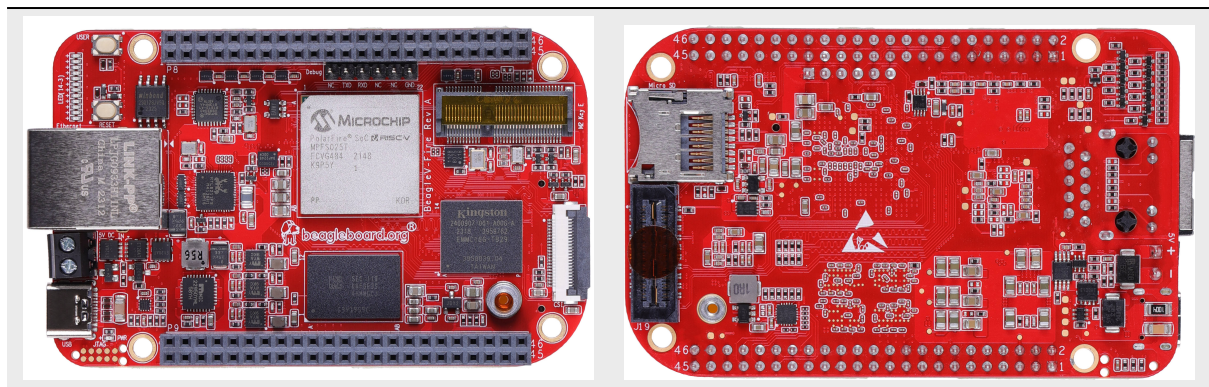
1.1   Pinout Diagrams

Choose the cape header to see respective pinout diagram.

P8 cape header

P9 cape header

Fig. 1.1: BeagleV-Fire P8 cape header pinout

Fig. 1.2: BeagleV-Fire P9 cape header pinout

1.2 Detailed overview

Table 1.1: BeagleV-Fire features

Feature	Description
Processor	MPFS025T-FCVG484E
Memory	2GB (1Gb x 16)- 1866MHz 3733Mbps, LPDDR4
Storage	Kingston 16GB eMMC
Wireless	1x M.2 Key E, support 2.4GHz/5GHz WiFi module
Ethernet	• PHY: Realtek RTL8211F-VD-CG Gigabit Ethernet phy • Connector: integrated magnetics RJ-45
USB C	• Connectivity: Flash/programming support • Power: Input: 5V @ 3A
Other connectors	• 1x SYZYGY High speed connector • microSD card slot • CSI connector compatible with BeagleBone AI-64, BeagleV-Ahead, Raspberry Pi Zero / CM4 (22-pin)

1.3 Board components location

This section describes the key components on the board, their location and function.

1.3.1 Front components location
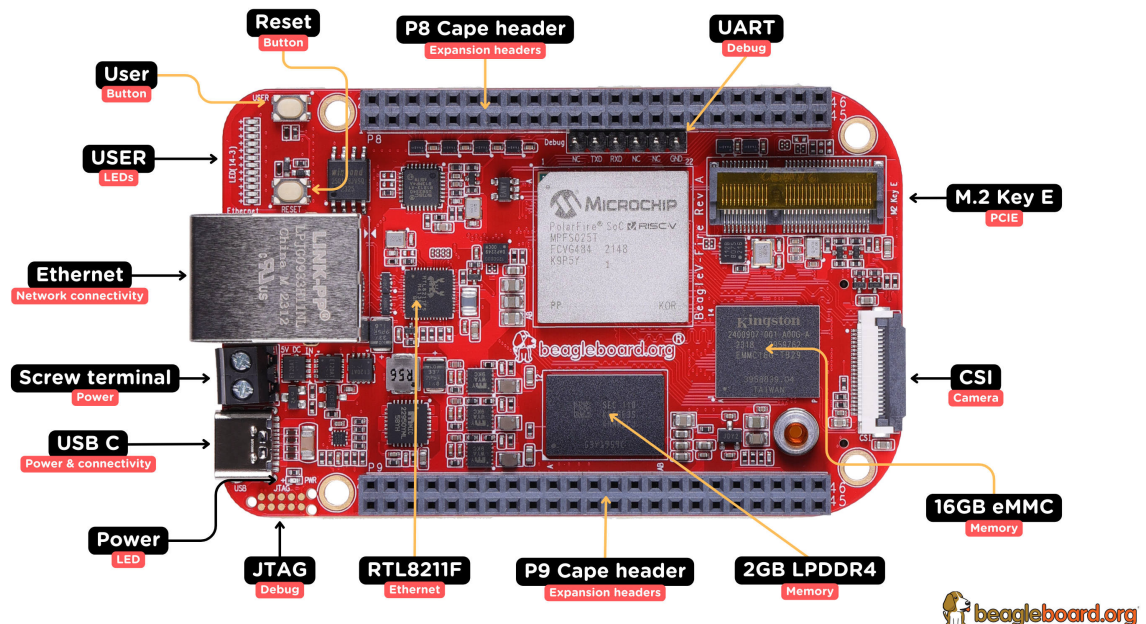
Fig. 1.3: BeagleV-Fire board front components location

Table 1.2: BeagleV-Fire board front components location

Feature	Description
Power LED	Power (Board ON) indicator
JTAG (MPFS025T)	MPFS025T SoC JTAG debug port
RTL8211F	Gigabit IEEE 802.11 Ethernet PHY
P8 & P9 cape header	Expansion headers for BeagleBone capes.
2GB RAM	2GB (1Gb x 16)- 1866MHz 3733Mbps, LPDDR4
16GB eMMC	Kingston 16GB eMMC Flash storage
CSI	22pin MIPI Camera connectors
M.2 Key E	PCIE M.2 Key E connector
UART debug header	6 pin UART debug header
Reset button	Press to reset BeagleV-Fire board (MPFS025T SoC)
User button	User defined (custom) action button
User LEDs	12x user programmabkle LEDs to show various board status during boot.
GigaBit Ethernet	1Gb/s Wired internet connectivity
Barrel jack	Power input
USB C	Power, connectivity, and board flashing.

1.3.2 Back components location
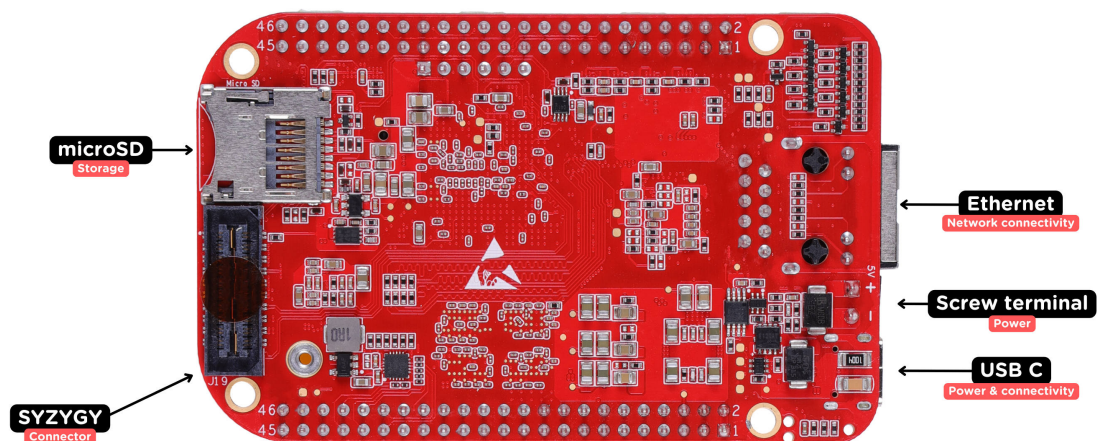
Fig. 1.4: BeagleV-Fire board back components location

Table 1.3: BeagleV-Fire board back components location

Feature	Description
microSD	microSD card slot
SYZYGY	SYZYGY High speed connector

Chapter 2

Quick Start

2.1 What's included in the box?

When you purchase a brand new BeagleV-Fire, In the box you'll get:

1. BeagleV-Fire board
2. Quick-start card

Tip: For board files, 3D model, and more, you can checkout BeagleV-Fire repository on OpenBeagle.

2.2 Unboxing

2.3 Tethering to PC

To connect BeagleV-Fire board to PC via USB Type C receptacle you need a USB type C cable. Connection guide for the same is shown below:

Tip: To get a USB type C cable you can checkout links below:

1. USB C cable 0.3m (mouser)
2. USB C cable 1.83m (digikey)

2.4 Flashing eMMC

2.4.1 Flash the latest image on eMMC

2.5 Access UART debug console

Note: Some tested devices that are working good includes:

1. Adafruit CP2102N Friend - USB to Serial Converter
2. Raspberry Pi Debug Probe Kit for Pico and RP2040

Fig. 2.1: https://youtu.be/5cylv1R-1mc
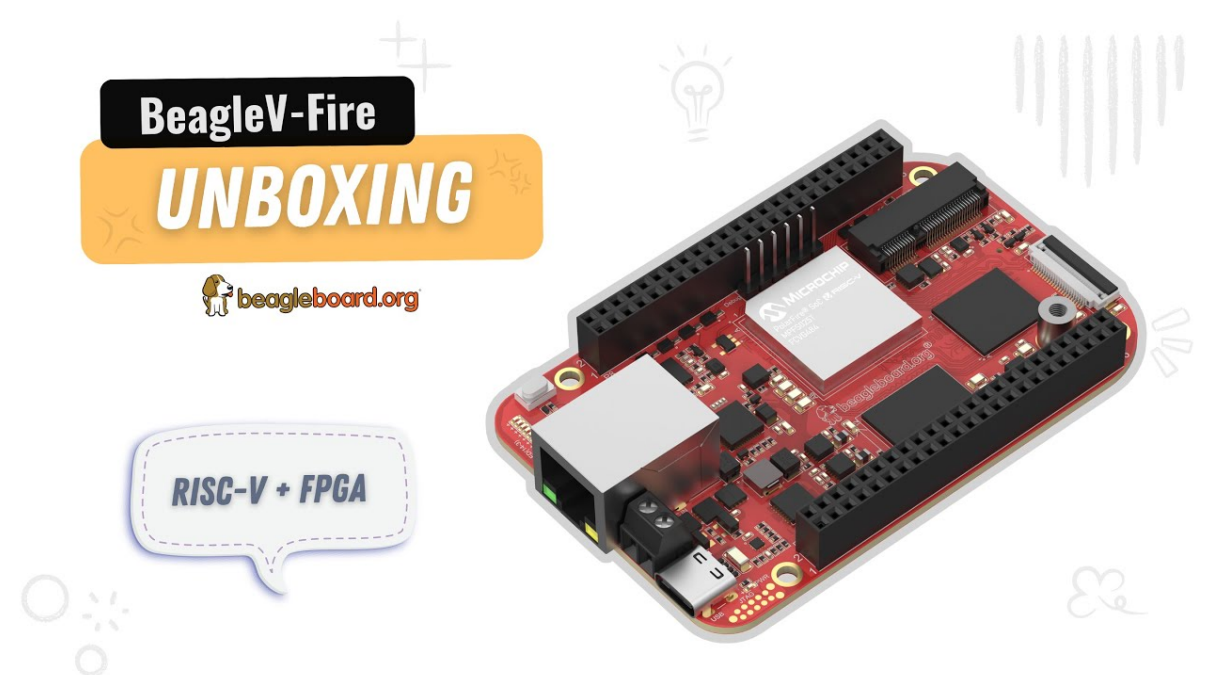
Fig. 2.2: BeagleV-Fire tethered connection

To access a BeagleV-Fire serial debug console you can connected a USB to UART to your board as shown below:
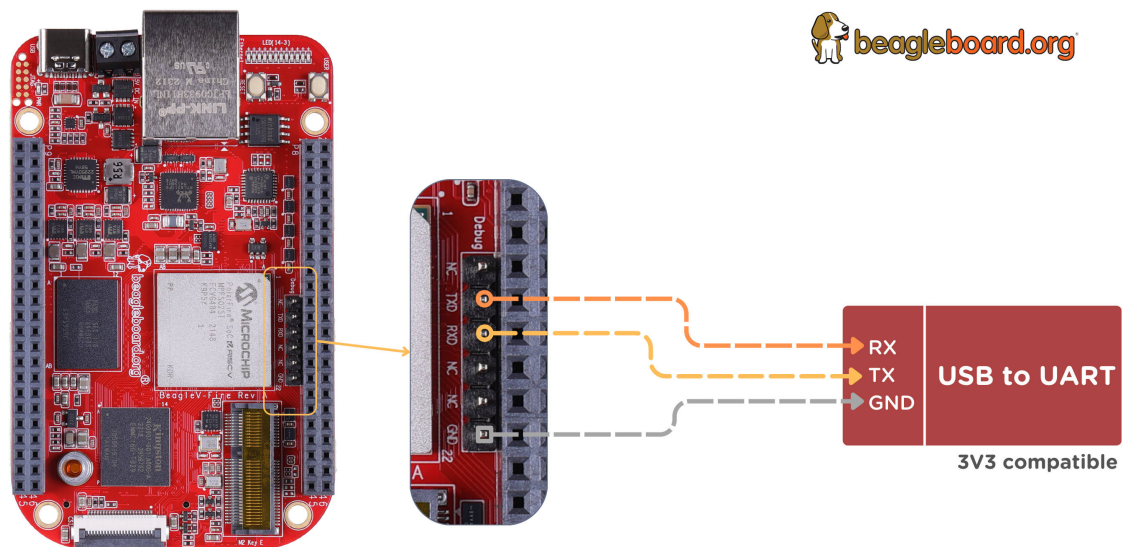
Fig. 2.3: BeagleV-Fire UART debug port connection

To see the board boot log and access your BeagleV-Fire's console you can use application like `tio` to access the console. If you are using Linux your USB to UART converter may appear as `/dev/ttyUSB`. It will be different for Mac and Windows operatig systems. To find serial port for your system you can checkout this guide.

```
[lorforlinux@fedora ~] $ tio /dev/ttyUSB0
tio v2.5
Press ctrl-t q to quit
Connected
```

2.6 Demos and Tutorials

- *How to retrieve BeagleV-Fire's gateware version*
- *Upgrade BeagleV-Fire Gateware*
- *Flashing gateware and Linux image*
- *Gateware Design Introduction*
- *Microchip FPGA Tools Installation Guide*
- *How to build the BeagleV-Fire Gateware on Windows*

Chapter 3

Design & specifications

If you want to know how BeagleV-Fire board is designed and what are it's high-level specifications then this chapter is for you. We are going to discuss each hardware design element in detail and provide high-level device specifications in a short and crisp form as well.

Tip: For hardware design files and schematic diagram you can checkout BeagleV-Fire GitLab repository: https://git.beagleboard.org/beaglev-fire/beaglev-fire
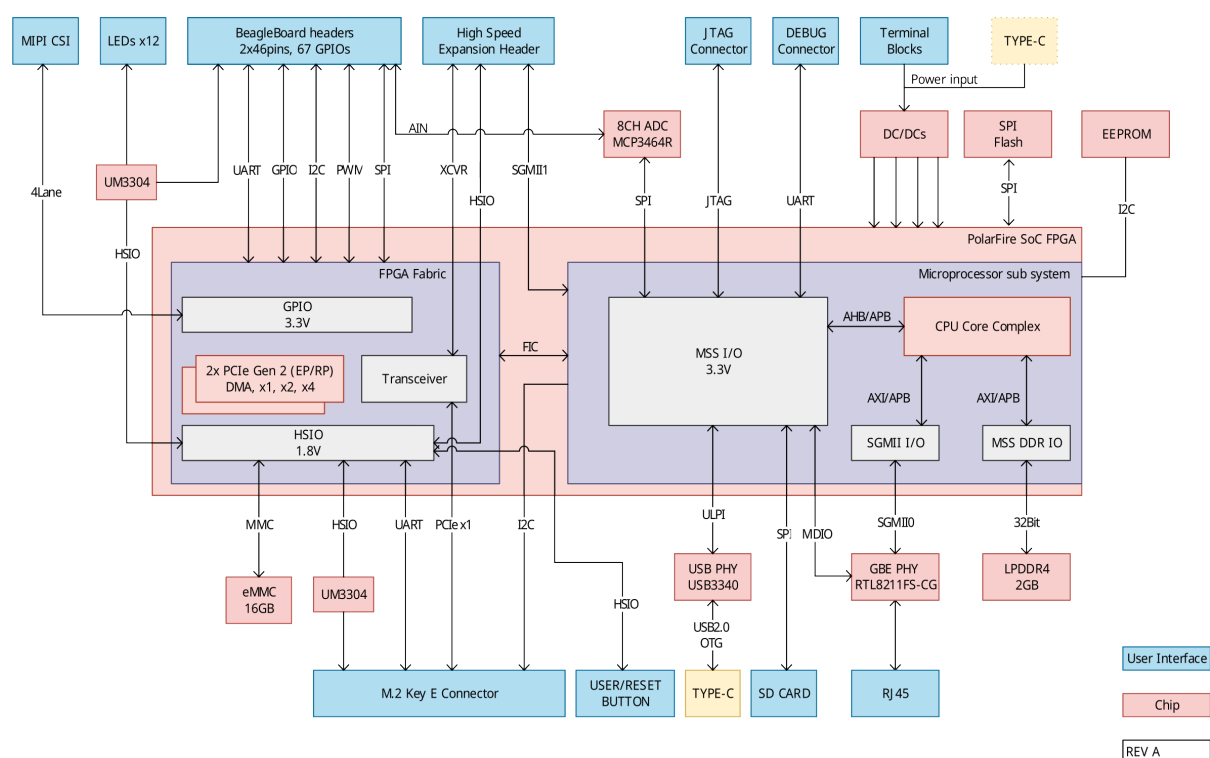
3.1 Block diagram

Fig. 3.1: System block diagram
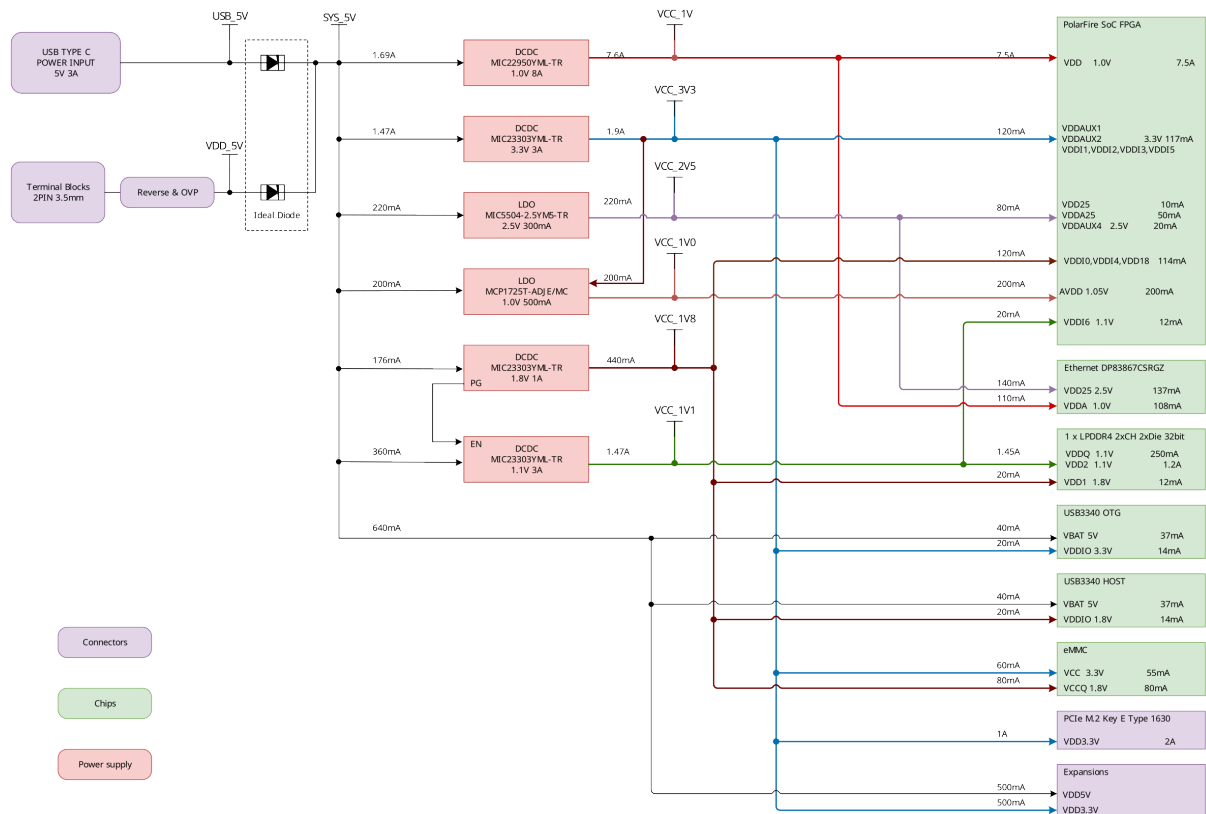
Fig. 3.2: Power tree diagram

3.2   System on Chip (SoC)

3.3   Power management

3.4   General Connectivity and Expansion

3.4.1   USB-C port

3.4.2   P8 & P9 cape header pins

3.4.3   ADC

3.5   Buttons and LEDs

3.5.1   User LEDs and Power LED

3.5.2   User and reset button

3.6   Connectivity

3.6.1   Ethernet

BANK2_I2C1_SCL

BANK2_I2C1_SDA

MPFS025T
FCVG484E

EEPROM
ADDR:0x50

M.2
Connector

CSI
Connector

High Speed
Connector

Fabric_I2C0_SCL

Fabric_I2C0_SDA

P9
Connector

Fig. 3.3: I2C tree diagram

BANK 0

U1A

MPFS025T-FCVG484E
m5s250_soc_fcv484

Fig. 3.4: SoC bank0

U1B

BANK 1

[17] B1_GPIO48N	C12	GPIO48NB1
[17] B1_GPIO48P	B12	GPIO48PB1/CLKIN_S_4
	D16	NC/GPIO10NB1
	E16	NC/GPIO10PB1/CCC_SE_PLL0_OUT1
[17] B1_GPIO1N	F16	GPIO1NB1
[17] B1_GPIO1P	F17	GPIO1PB1/CLKIN_S_9/CCC_SE_CLKIN_S_9
	D17	NC/GPIO12NB1
	C16	NC/GPIO12PB1
	A17	NC/GPIO13NB1
	A16	NC/GPIO13PB1
[17] B1_GPIO2N	D19	GPIO2NB1/DQS
[11] ADC_MCLK	E19	GPIO2PB1/DQS/CCC_SE_PLL1_OUT0
[17] B1_GPIO3N	C17	GPIO3NB1
[17] B1_GPIO3P	B17	GPIO3PB1/CCC_SE_CLKIN_S_10/CCC_SE_PLL1_OUT0
[17] B1_GPIO36N	D6	GPIO36NB1
[17] B1_GPIO36P	D7	GPIO36PB1/CCC_SW_CLKIN_S_0
[17] B1_GPIO37N	B4	GPIO37NB1
[17] B1_GPIO37P	C4	GPIO37PB1/CCC_SW_CLKIN_S_1
[17] B1_GPIO4N	E18	GPIO4NB1
[17] B1_GPIO4P	D18	GPIO4PB1/CCC_SE_PLL1_OUT0
[17] B1_GPIO38N	B5	GPIO38NB1/DQS
[17] B1_GPIO38P	C5	GPIO38PB1/DQS/CCC_SW_PLL1_OUT0
[7] SD_DET	C7	GPIO39NB1
[17] B1_GPIO39P	C6	GPIO39PB1/CLKIN_S_2/CCC_SW_CLKIN_S_2/CCC_SW_PLL1_OUT0
[17] B1_GPIO40N	A5	GPIO40NB1
[17] B1_GPIO40P	A6	GPIO40PB1/CCC_SW_PLL1_OUT1
[17] B1_GPIO41N	B7	GPIO41NB1
[17] B1_GPIO41P	A7	GPIO41PB1/CLKIN_S_3/CCC_SW_CLKIN_S_3
[17] B1_GPIO42N	D8	GPIO42NB1
[17] B1_GPIO42P	D9	GPIO42PB1
[17] B1_GPIO43N	B8	GPIO43NB1
[17] B1_GPIO43P	A8	GPIO43PB1
[17] B1_GPIO44N	C9	GPIO44NB1/DQS
[17] B1_GPIO44P	C10	GPIO44PB1/DQS/CCC_SW_PLL0_OUT0
[17] Fabric_I2C0_SDA	A11	GPIO45NB1
	A10	GPIO45PB1/CCC_SW_PLL0_OUT0
[17] B1_GPIO46N	D11	GPIO46NB1
[17] B1_GPIO46P	C11	GPIO46PB1/CCC_SW_PLL0_OUT1
[16,17] P8_PIN14	B9	GPIO47NB1
[16,17] P8_PIN13	B10	GPIO47PB1
[17] B1_GPIO5N	B18	GPIO5NB1
[17] B1_GPIO5P	A18	GPIO5PB1/CCC_SE_CLKIN_S_11
		DQ23

F10	GPIO180NB1/NC	
E10	GPIO180PB1/NC	
F11	GPIO181NB1/NC	
F12	GPIO181PB1/NC	
G12	GPIO182NB1/DQS/NC	
H12	GPIO182PB1/DQS/NC	
E11	GPIO183NB1/NC	
D12	GPIO183PB1/NC	
F13	GPIO184NB1/NC	
E13	GPIO184PB1/NC	
H13	GPIO185NB1/NC	
G13	GPIO185PB1/NC	
C20	GPIO18NB9/NC	
C19	GPIO18PB9/NC	
B19	GPIO19NB9/NC	
B20	GPIO19PB9/NC	
A12	GPIO49NB1	B1_GPIO49N [17]
A13	GPIO49PB1/CLKIN_S_5	B1_GPIO49P [17]
A21	GPIO20NB9/DQS/NC	
A20	GPIO20PB9/DQS/NC	
D20	GPIO21NB9/NC	
D21	GPIO21PB9/NC	
B22	GPIO22NB9/NC	
B21	GPIO22PB9/NC	
D22	GPIO23NB9/NC	
C22	GPIO23PB9/NC	
D14	GPIO50NB1/DQS	B1_GPIO50N [17]
D13	GPIO50PB1/DQS	B1_GPIO50P [17]
B14	GPIO51NB1	B1_GPIO51N [17]
B13	GPIO51PB1/CLKIN_S_6	B1_GPIO51P [17]
C15	GPIO52NB1/LPRB_B	B1_GPIO52N [17]
C14	GPIO52PB1/LPRB_A	B1_GPIO52P [17]
B15	GPIO53NB1	B1_GPIO53N [17]
A15	GPIO53PB1/CLKIN_S_7	B1_GPIO53P [17]
G14	GPIO6NB1/NC	
F15	GPIO6PB1/NC	
H15	GPIO7NB1/NC	
G15	GPIO7PB1/NC	
H17	GPIO8NB1/DQS/NC	
G17	GPIO8PB1/DQS/CCC_SE_PLL0_OUT0/NC	
E14	GPIO0NB1	B1_GPIO0N [17]
E15	GPIO0PB1/CLKIN_S_8/CCC_SE_CLKIN_S_8/CCC_SE_PLL0_OUT0	B1_GPIO0P [17]

VCC_3V3

R249 2.2k 1% R0402 R250 2.2k 1% R0402

MPFS025T-FCVG484E
m5s250_soc_fcv484

Fig. 3.5: SoC bank1

VCC_3V3

R108 2.2k 1% R0402 R109 2.2k 1% R0402

U1C

BANK 2

[10] USB0_CLK	G2	MSSIO14B2
[10] USB0_DIR	F1	MSSIO15B2
[10] USB0_NXT	G5	MSSIO16B2
[10] USB0_STP	G4	MSSIO17B2
[10] USB0_DATA0	F2	MSSIO18B2
[10] USB0_DATA1	E1	MSSIO19B2
[10] USB0_DATA2	G3	MSSIO20B2
[10] USB0_DATA3	F5	MSSIO21B2
[10] USB0_DATA4	D1	MSSIO22B2
[10] USB0_DATA5	D2	MSSIO23B2
[10] USB0_DATA6	F6	MSSIO24B2
[10] USB0_DATA7	F3	MSSIO25B2

MSSIO26B2	C1	I2C1_SCL [12,16]
MSSIO27B2	B1	I2C1_SDA [12,16]
MSSIO28B2	D3	UART0_RXD [16]
MSSIO29B2	C2	UART0_TXD [16]
MSSIO30B2	E5	ADC_SCK [7]
MSSIO31B2	E4	ADC_CSn [11]
MSSIO32B2	B2	ADC_MOSI [7,11]
MSSIO33B2	A2	ADC_MISO [7,11]
MSSIO34B2	B3	ADC_IROn [11]
MSSIO35B2	A3	PHY_MDC [9]
MSSIO36B2	E3	PHY_MDIO [9]
MSSIO37B2	D4	USB0_OCn [10]

MPFS025T-FCVG484E
m5s250_soc_fcv484

Fig. 3.6: SoC bank2

Fig. 3.7: SoC bank3

Fig. 3.8: SoC bank4

3.7 Memory, Media and Data storage

3.7.1 DDR memory

3.7.2 eMMC

3.7.3 microSD

3.7.4 EEPROM

3.7.5 SPI flash

3.8 Multimedia I/O

3.8.1 CSI

3.9 Debug

3.9.1 UART debug port

3.9.2 JTAG debug port

Fig. 3.9: SoC power

Fig. 3.10: DC 5V input
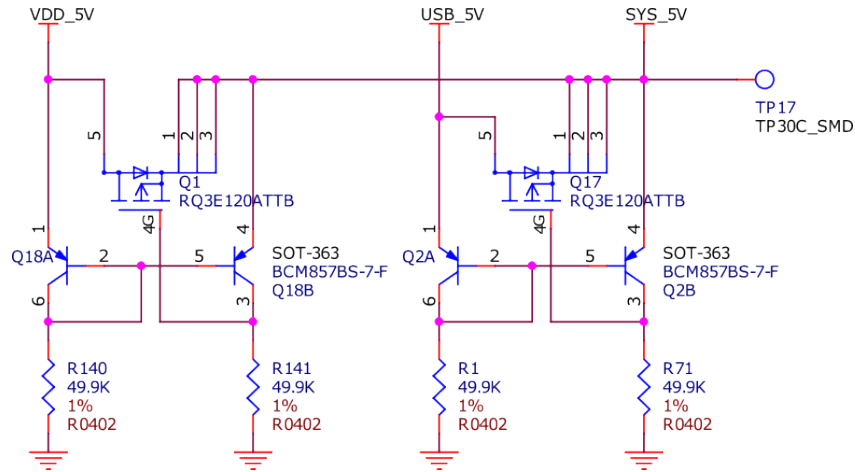
Fig. 3.11: Ideal diode
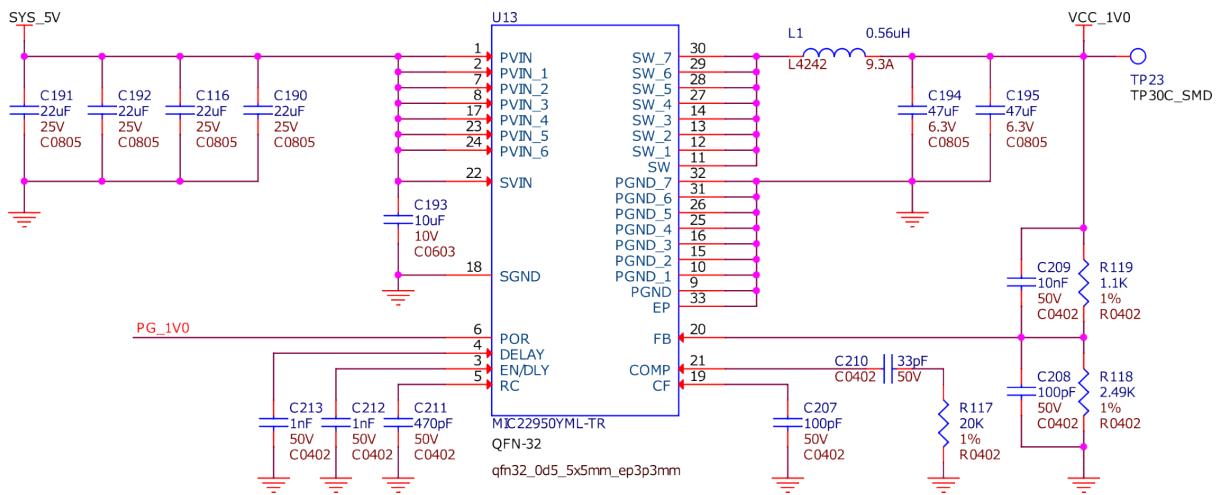
Fig. 3.12: VCC 1V0
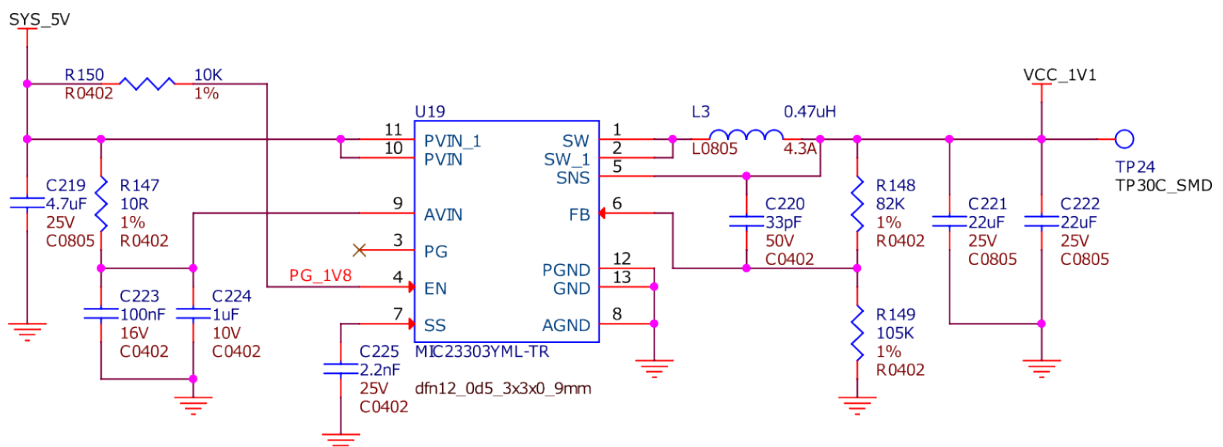
Fig. 3.13: VCC 1V1
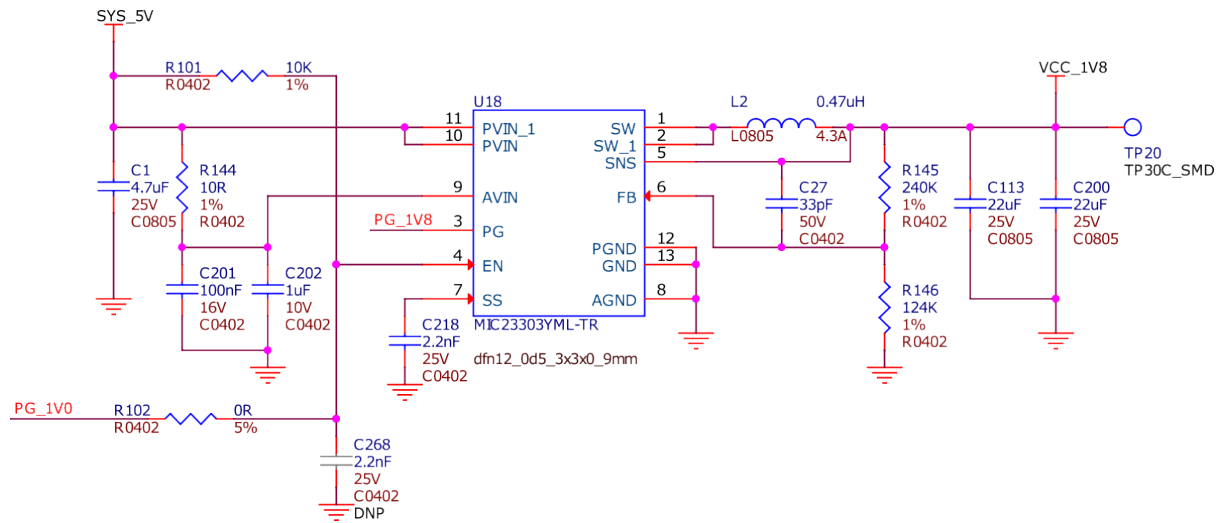
Fig. 3.14: VCC 1V8
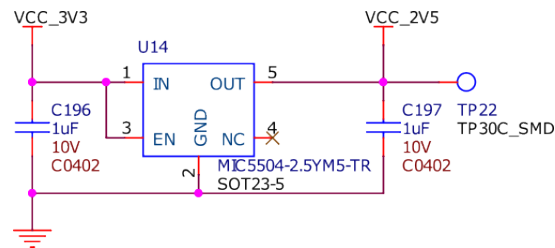
Fig. 3.15: VCC 2V5
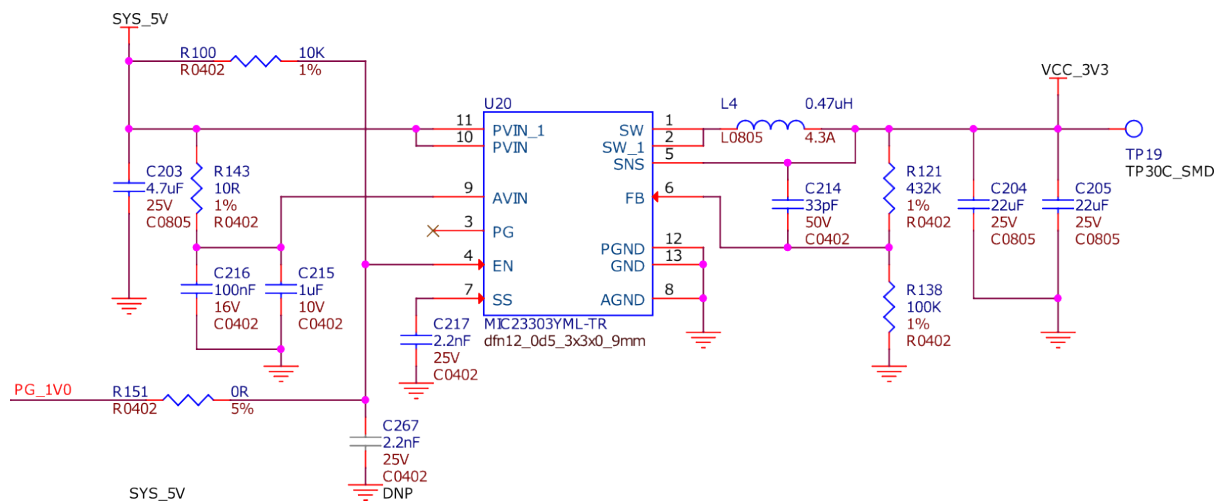
Fig. 3.16: VCC 3V3
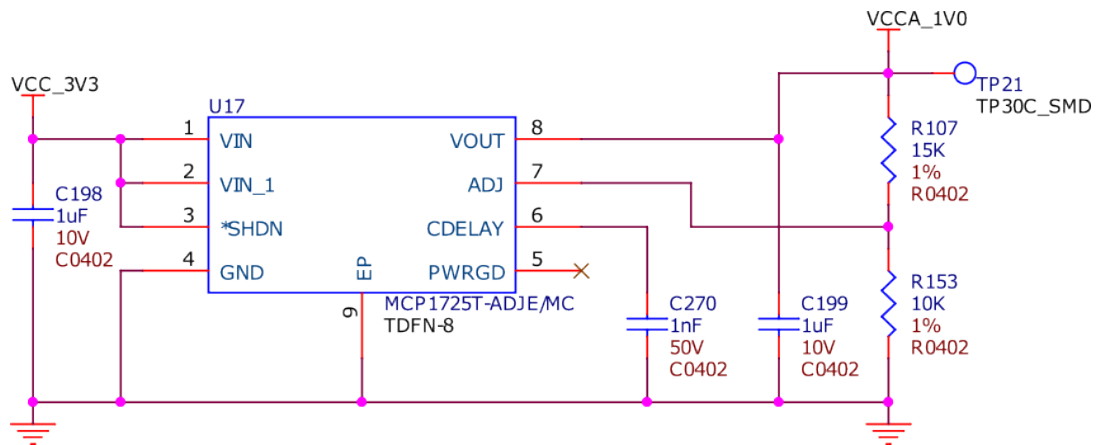
Fig. 3.17: VCCA 1V0
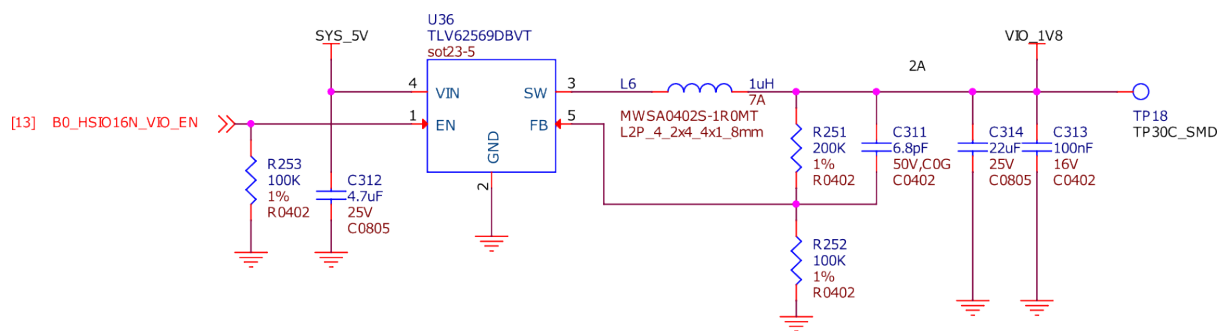
Fig. 3.18: VIO enable

3.10 Mechanical Specifications

Table 3.1: Dimensions & weight

Parameter	Values
Size	86.38 * 54.61 * 18.8 mm
Max heigh	18.8 mm
PCB Size	86.38 * 54.6 mm
PCB Layers	12 Layers
PCB Thickness	1.6 mm
RoHS compliant	Yes
Gross Weight	106 g
Net weight	45.8 g

Fig. 3.19: USB C

Fig. 3.20: P8 cape header

Fig. 3.21: P9 cape header

Fig. 3.22: Cape header voltage level translator

Fig. 3.23: 16bit Delta-Sigma ADC
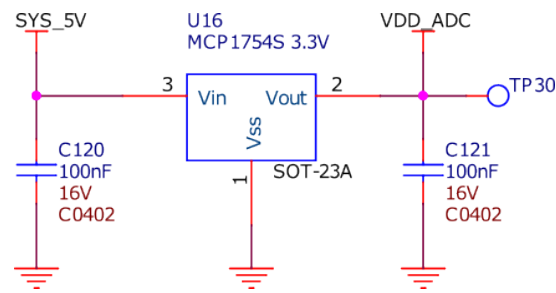
Fig. 3.24: ADC LDO power supply
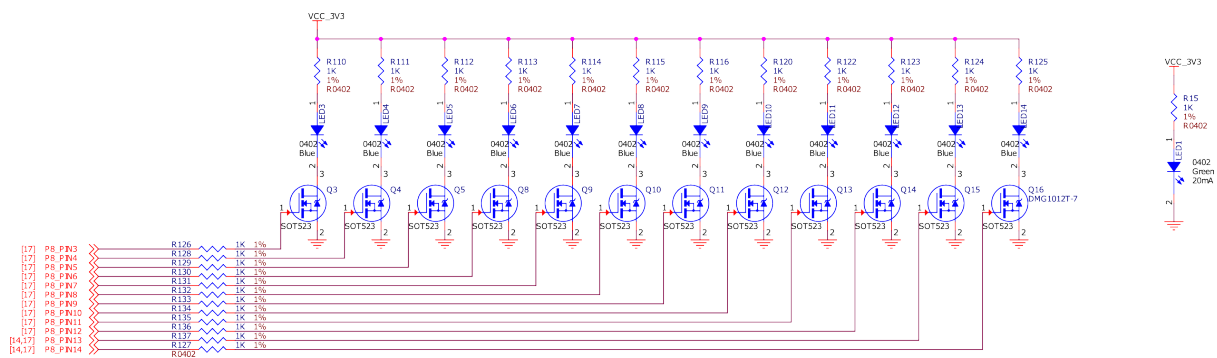
Fig. 3.25: User LEDs and power LED
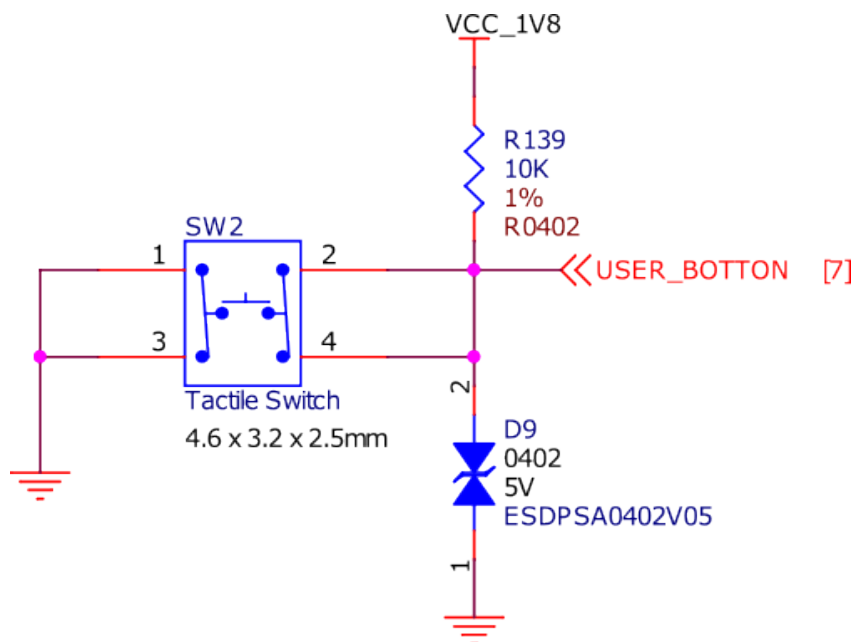
Fig. 3.26: User button
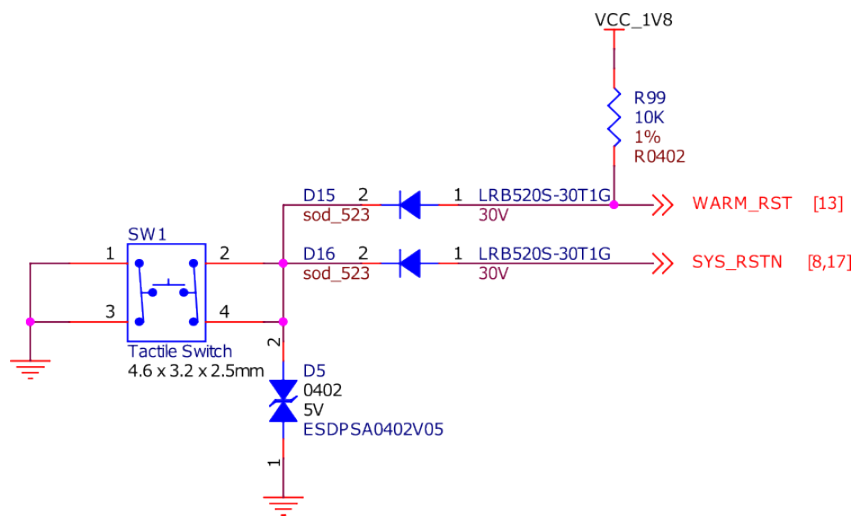
Fig. 3.27: Reset button
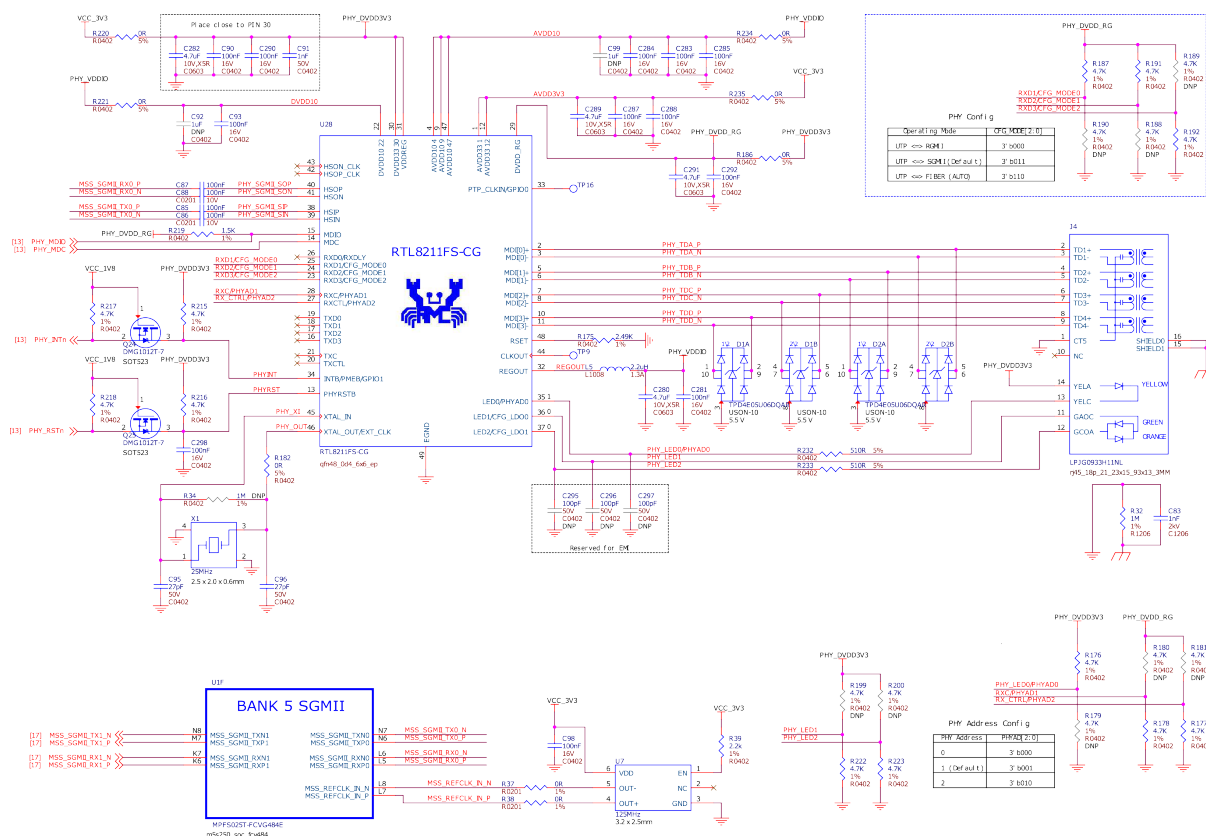
Fig. 3.28: Gigabit ethernet
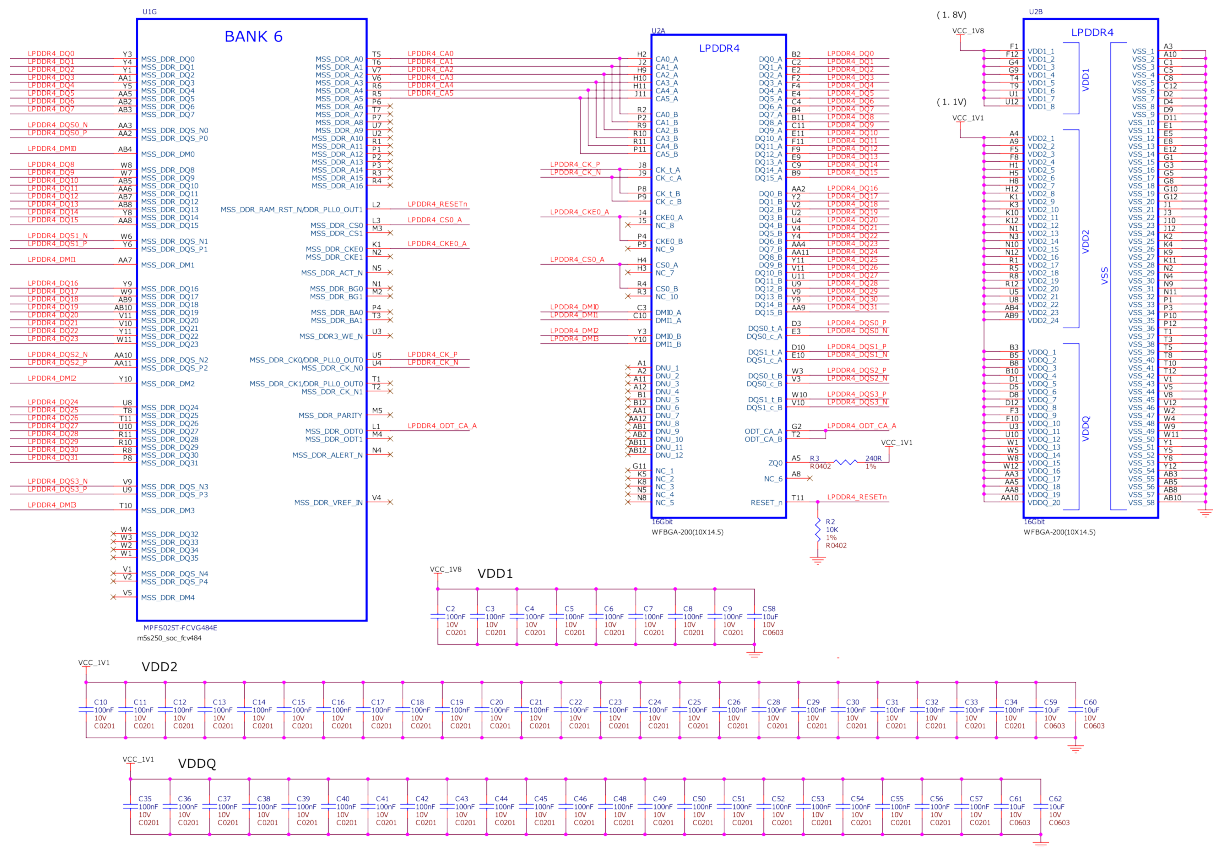
Fig. 3.29: LPDDR memory
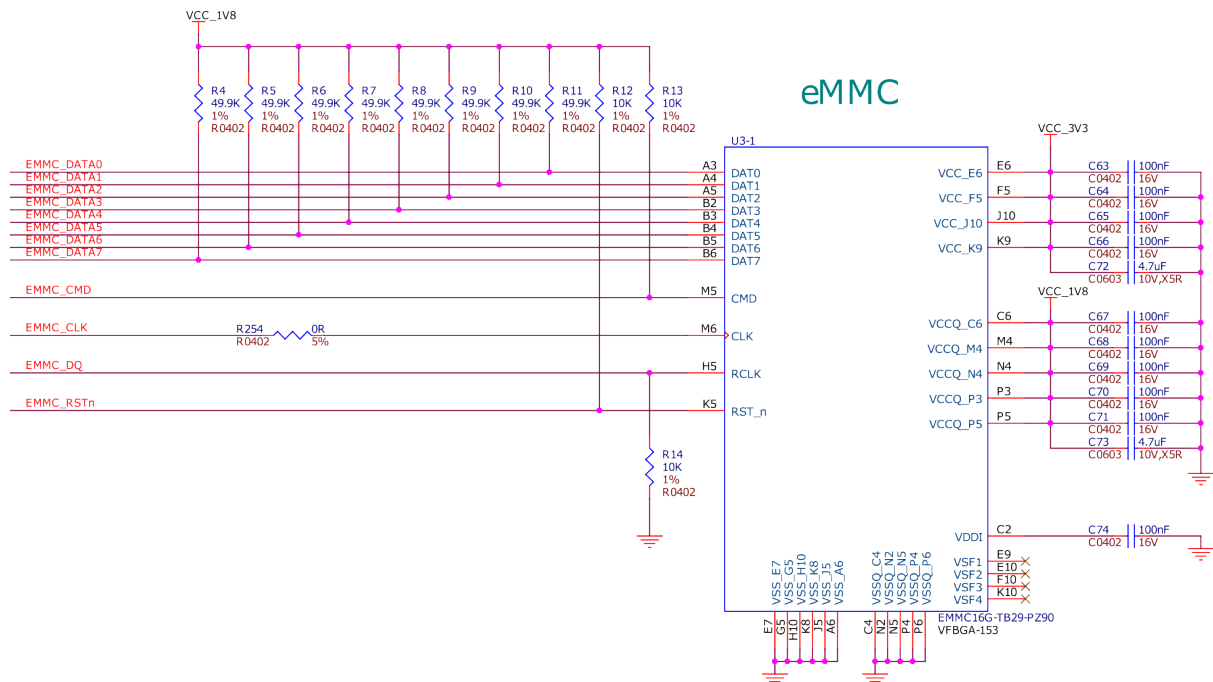
Fig. 3.30: EMMC flash storage

Fig. 3.31: SD Card socket

Fig. 3.32: EEPROM

Fig. 3.33: SPI Flash
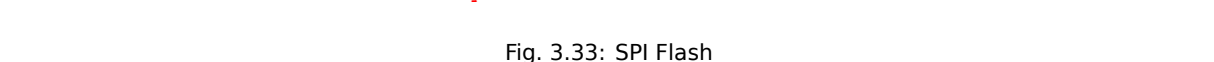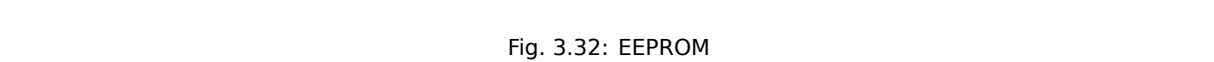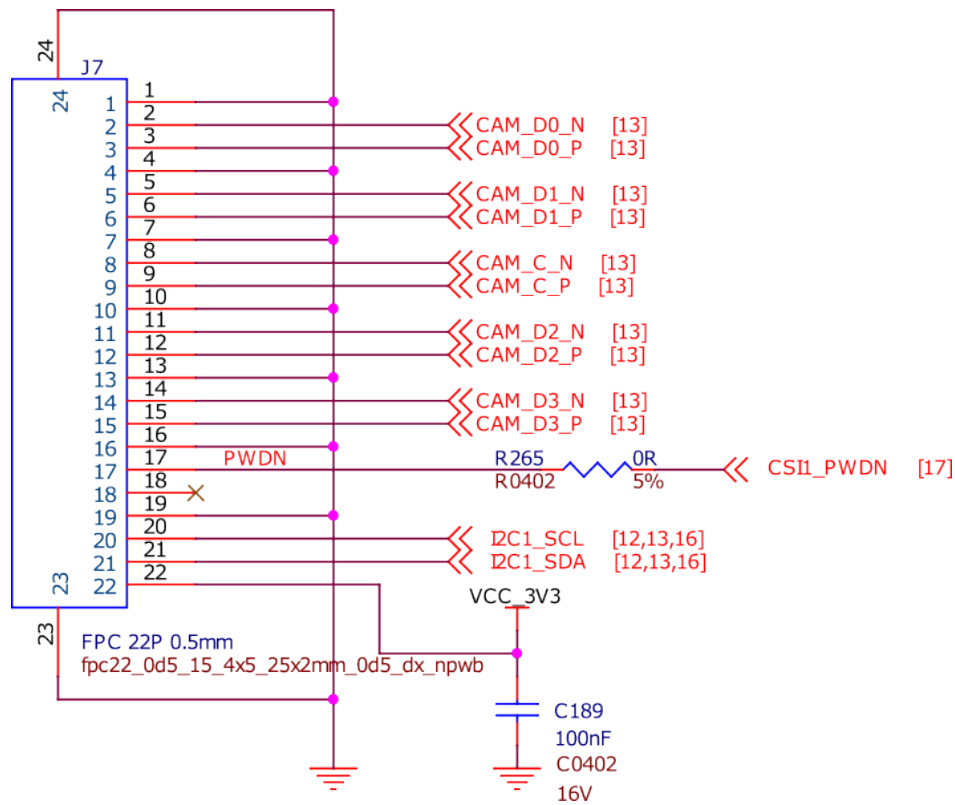
Fig. 3.34: CSI
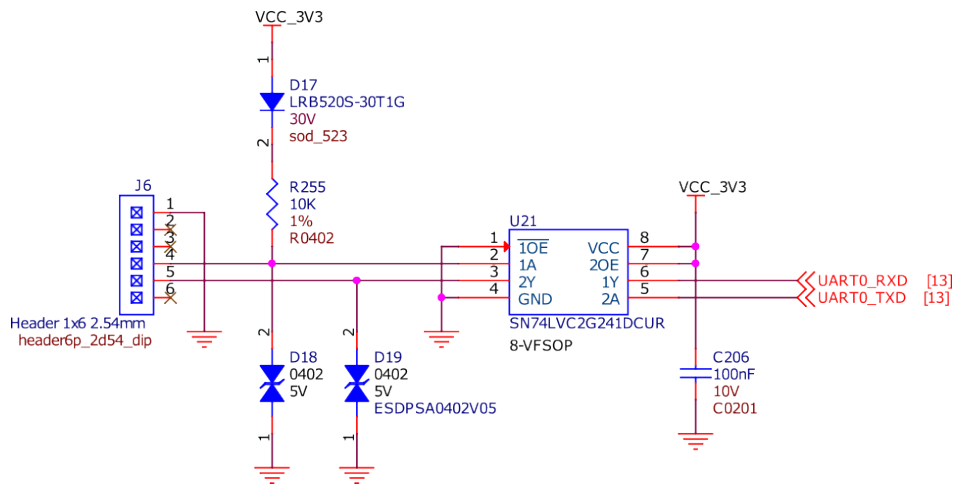
Fig. 3.35: UART debug header
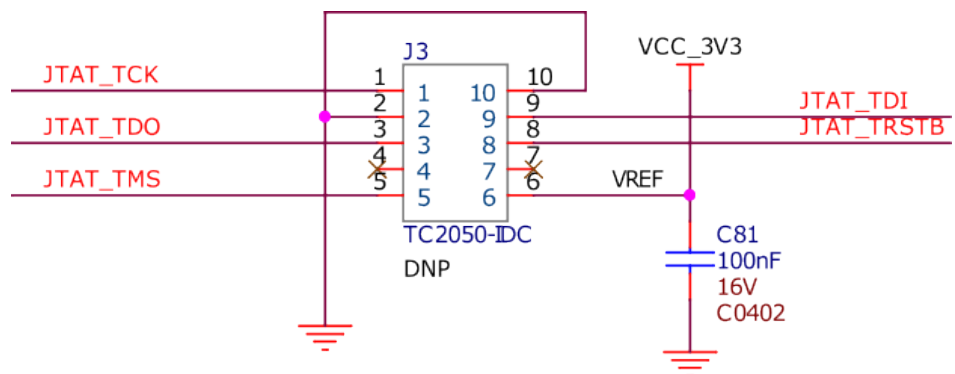
Fig. 3.36: JTAG debug header

Chapter 4

Expansion

4.1 Cape Headers

The expansion interface on the board is comprised of two headers P8 (46 pin) & P9 (46 pin). All signals on the expansion headers are **3.3V** unless otherwise indicated.

Note: Do not connect 5V logic level signals to these pins or the board will be damaged.

Note: DO NOT APPLY VOLTAGE TO ANY I/O PIN WHEN POWER IS NOT SUPPLIED TO THE BOARD. IT WILL DAMAGE THE PROCESSOR AND VOID THE WARRANTY.

NO PINS ARE TO BE DRIVEN UNTIL AFTER THE SYS_RESET LINE GOES HIGH.

4.1.1 Connector P8

The following tables show the pinout of the **P8** expansion header. The gateware image is responsible for setting the function of each pin. Refer to the processor documentation for more information on these pins and detailed descriptions of all of the pins listed. In some cases there may not be enough signals to complete a group of signals that may be required to implement a total interface.

The column heading is the pin number on the expansion header.

The **Name** row is the pin name on the processor.

The **BALL** row is the pin number on the processor.

The rows below **BALL** are the gateware setting for each pin.

NOTES:

DO NOT APPLY VOLTAGE TO ANY I/O PIN WHEN POWER IS NOT SUPPLIED TO THE BOARD. IT WILL DAMAGE THE PROCESSOR AND VOID THE WARRANTY.

NO PINS ARE TO BE DRIVEN UNTIL AFTER THE SYS_RESET LINE GOES HIGH.

P8.01-P8.02

P8.01	P8.02
GND	GND

P8.03-P8.05

Pin	P8.03	P8.04	P8.05
Name	HSIO6NB0	HSIO6PB0/CCC_NE_CLKIN_N_11	HSIO12NB0
BALL	V22	W22	V19
DEFAULT	MSS GPIO_2[0]	MSS GPIO_2[1]	MSS GPIO_2[2]
	User LED 0	User LED 1	User LED 2
GPIOS	MSS GPIO_2[0]	MSS GPIO_2[1]	MSS GPIO_2[2]
	User LED 0	User LED 1	User LED 2
ROBOTICS	MSS GPIO_2[0]	MSS GPIO_2[1]	MSS GPIO_2[2]
	User LED 0	User LED 1	User LED 2

P8.06-P8.09

Pin	P8.06	P8.07	P8.08	P8.09
Name	HSIO12PB0/CLKIN_N_9/CCC_NE_CLKIN_N_9	HSIO30NB0	HSIO30PB0/CLKIN_N_3/CCC_NW_CLKIN_N_3	HSIO8NB0
BALL	V20	V15	V14	V21
DEFAULT	MSS GPIO_2[3]	MSS GPIO_2[4]	MSS GPIO_2[5]	MSS GPIO_2[6]
	User LED 3	User LED 4	User LED 5	User LED 6
GPIOS	MSS GPIO_2[3]	MSS GPIO_2[4]	MSS GPIO_2[5]	MSS GPIO_2[6]
	User LED 3	User LED 4	User LED 5	User LED 6
ROBOTICS	MSS GPIO_2[3]	MSS GPIO_2[4]	MSS GPIO_2[5]	MSS GPIO_2[6]
	User LED 3	User LED 4	User LED 5	User LED 6

P8.10-P8.13

Pin	P8.10	P8.11	P8.12	P8.13	
Name	HSIO8PB0/CCC_NE_CLKIN_N_10/CCC_NE_PLL0_0	HSIO7NB0	HSIO7PB0/CCC_NE_PLL0_OUT	GPIO47PB1	
BALL	W21	Y21	Y20	B10	
DEFAULT	MSS GPIO_2[7]	MSS GPIO_2[8]	MSS GPIO_2[9]	core_pwm[1] @ 0x41500000	@
	User LED 7	User LED 8	User LED 9	PWM_2:1	
GPIOS	MSS GPIO_2[7]	MSS GPIO_2[8]	MSS GPIO_2[9]	MSS GPIO_2[10]	
	User LED 7	User LED 8	User LED 9	User LED 10	
ROBOTICS	MSS GPIO_2[7]	MSS GPIO_2[8]	MSS GPIO_2[9]	core_pwm[1] @ 0x41500000	@
	User LED 7	User LED 8	User LED 9	PWM_2:1	

P8.14-P8.16

Pin	P8.14	P8.15	P8.16
Name	GPIO47NB1	HSIO34NB0	HSIO34PB0/CCC_NW_CLKIN_N_1
BALL	B9	T12	U12
DEFAULT	MSS GPIO_2[11]	MSS GPIO_2[12]	MSS GPIO_2[13]
	User LED 11	GPIO	GPIO
GPIOS	MSS GPIO_2[11]	MSS GPIO_2[12]	MSS GPIO_2[13]
	User LED 11	GPIO	GPIO
ROBOTICS	MSS GPIO_2[11]	MSS GPIO_2[12]	MSS GPIO_2[13]
	User LED 11	GPIO	GPIO

P8.17-P8.19

Pin	P8.17	P8.18	P8.19
Name	HSIO29PB0	HSIO15PB0/DQS/CCC_NE_PLL1_OUT0	HSIO19NB0
BALL	W13	T16	W18
DEFAULT	MSS GPIO_2[14]	MSS GPIO_2[15]	core_pwm[0] @ 0x41500000
	GPIO	GPIO	PWM_2:0
GPIOS	MSS GPIO_2[14]	MSS GPIO_2[15]	MSS GPIO_2[16]
	GPIO	GPIO	GPIO
ROBOTICS	MSS GPIO_2[14]	MSS GPIO_2[15]	core_pwm[0] @ 0x41500000
	GPIO	GPIO	PWM_2:0

P8.20-P8.22

Pin	P8.20	P8.21	P8.22
Name	HSIO15NB0/DQS	HSIO9PB0/DQS/CCC_NE_PLL0_OUT0	HSIO9NB0/DQS
BALL	R16	AA21	AA22
DEFAULT	MSS GPIO_2[17]	MSS GPIO_2[18]	MSS GPIO_2[19]
	GPIO	GPIO	GPIO
GPIOS	MSS GPIO_2[17]	MSS GPIO_2[18]	MSS GPIO_2[19]
	GPIO	GPIO	GPIO
ROBOTICS	MSS GPIO_2[17]	MSS GPIO_2[18]	MSS GPIO_2[19]
	GPIO	GPIO	GPIO

P8.23-P8.26

Pin	P8.23	P8.24	P8.25	P8.26
Name	HSIO18PB0/CLKIN_N_7	HSIO18NB0	HSIO16PB0	GPIO49NB1
BALL	AB18	AA18	V17	A12
DEFAULT	MSS GPIO_2[20]	MSS GPIO_2[21]	MSS GPIO_2[22]	MSS GPIO_2[23]
	GPIO	GPIO	GPIO	GPIO
GOIOS	MSS GPIO_2[20]	MSS GPIO_2[21]	MSS GPIO_2[22]	MSS GPIO_2[23]
	GPIO	GPIO	GPIO	GPIO
ROBOTICS	MSS GPIO_2[20]	MSS GPIO_2[21]	MSS GPIO_2[22]	MSS GPIO_2[23]
	GPIO	GPIO	GPIO	GPIO

P8.27-P8.29

Pin	P8.27	P8.28	P8.29
Name	GPIO49PB1/CLKIN_S_5	GPIO51NB1	GPIO51PB1/CLKIN_S_6
BALL	A13	B14	B13
DEFAULT	MSS GPIO_2[24]	MSS GPIO_2[25]	MSS GPIO_2[26]
	GPIO	GPIO	GPIO
GPIOS	MSS GPIO_2[24]	MSS GPIO_2[25]	MSS GPIO_2[26]
	GPIO	GPIO	GPIO
ROBOTICS	MSS GPIO_2[24]	MSS GPIO_2[25]	MSS GPIO_2[26]
	GPIO	GPIO	GPIO

P8.30-P8.32

Pin	P8.30	P8.31	P8.32
Name	GPIO50NB1/DQS	GPIO50PB1/DQS	GPIO53NB1
BALL	D14	D13	B15
DEFAULT	MSS GPIO_2[27]	core_gpio[0] @ 0x41100000	core_gpio[1] @ 0x41100000
	GPIO	GPIO	GPIO
GPIOS	MSS GPIO_2[27]	core_gpio[0] @ 0x41100000	core_gpio[1] @ 0x41100000
	GPIO	GPIO	GPIO
ROBOTICS	MSS GPIO_2[27]	core_gpio[0] @ 0x41100000	core_gpio[1] @ 0x41100000
	GPIO	GPIO	GPIO

P8.33-P8.35

Pin	P8.33	P8.34	P8.35
Name	GPIO53PB1/CLKIN_S_7	GPIO52NB1/LPRB_B	GPIO52PB1/LPRB_A
BALL	A15	C15	C14
DEFAULT	core_gpio[2] @ 0x41100000	core_gpio[3] @ 0x41100000	core_gpio[4] @ 0x41100000
	GPIO	GPIO	GPIO
GPIOS	core_gpio[2] @ 0x41100000	core_gpio[3] @ 0x41100000	core_gpio[4] @ 0x41100000
	GPIO	GPIO	GPIO
ROBOTICS	core_gpio[2] @ 0x41100000	core_gpio[3] @ 0x41100000	core_gpio[4] @ 0x41100000
	GPIO	GPIO	GPIO

P8.36-P8.38

Pin	P8.36	P8.37	P8.38
Name	GPIO37NB1	GPIO37PB1/CCC_SW_CLKIN_S_1	GPIO3NB1
BALL	B4	C4	C17
DEFAULT	core_gpio[5] @ 0x41100000	core_gpio[6] @ 0x41100000	core_gpio[7] @ 0x41100000
	GPIO	GPIO	GPIO
GPIOS	core_gpio[5] @ 0x41100000	core_gpio[6] @ 0x41100000	core_gpio[7] @ 0x41100000
	GPIO	GPIO	GPIO
ROBOTICS	core_gpio[5] @ 0x41100000	core_gpio[6] @ 0x41100000	core_gpio[7] @ 0x41100000
	GPIO	GPIO	GPIO

P8.39-P8.41

Pin	P8.39	P8.40	P8.41
Name	GPIO3PB1/CCC_SE_CLKIN_S_10/CCC_SE_PLL1_OUT0	GPIO5NB1	GPIO5PB1/CCC_SE_CLKIN_S_11
BALL	B17	B18	A18
DEFAULT	core_gpio[8] @ 0x41100000	core_gpio[9] @ 0x41100000	core_gpio[10] @ 0x41100000
	GPIO	GPIO	GPIO
GPIOS	core_gpio[8] @ 0x41100000	core_gpio[9] @ 0x41100000	core_gpio[10] @ 0x41100000
	GPIO	GPIO	GPIO
ROBOTICS	core_gpio[8] @ 0x41100000	core_gpio[9] @ 0x41100000	core_gpio[10] @ 0x41100000
	GPIO	GPIO	GPIO

P8.42-P8.44

Pin	P8.42	P8.43	P8.44
Name	GPIO36NB1	GPIO36PB1/CCC_SW_CLKIN_S_0	GPIO42NB1
BALL	D6	D7	D8
DEFAULT	core_gpio[11] @ 0x41100000 GPIO	core_gpio[12] @ 0x41100000 GPIO	core_gpio[13] @ 0x41100000 GPIO
GPIOS	core_gpio[11] @ 0x41100000 GPIO	core_gpio[12] @ 0x41100000 GPIO	core_gpio[13] @ 0x41100000 GPIO
ROBOTICS	core_gpio[11] @ 0x41100000 GPIO	core_gpio[12] @ 0x41100000 GPIO	core_gpio[13] @ 0x41100000 GPIO

P8.45-P8.46

Pin	P8.45	P8.46
Name	GPIO42PB1	GPIO4PB1/CCC_SE_PLL1_OUT1
BALL	D9	D18
DEFAULT	core_gpio[14] @ 0x41100000 GPIO	core_gpio[15] @ 0x41100000 GPIO
GPIOS	core_gpio[14] @ 0x41100000 GPIO	core_gpio[15] @ 0x41100000 GPIO
ROBOTICS	core_gpio[14] @ 0x41100000 GPIO	core_gpio[15] @ 0x41100000 GPIO

4.1.2 Connector P9

The following tables show the pinout of the **P9** expansion header. The gateware image is responsible for setting the function of each pin. Refer to the processor documentation for more information on these pins and detailed descriptions of all of the pins listed. In some cases there may not be enough signals to complete a group of signals that may be required to implement a total interface.

The column heading is the pin number on the expansion header.

The **Name** row is the pin name on the processor.

The **BALL** row is the pin number on the processor.

The rows below **BALL** are the gateware setting for each pin.

NOTES:

DO NOT APPLY VOLTAGE TO ANY I/O PIN WHEN POWER IS NOT SUPPLIED TO THE BOARD. IT WILL DAMAGE THE PROCESSOR AND VOID THE WARRANTY.

NO PINS ARE TO BE DRIVEN UNTIL AFTER THE SYS_RESET LINE GOES HIGH.

P9.01-P9.05

P9.01	P9.02	P9.03	P9.04	P9.05
GND	GND	VCC_3V3	VCC_3V3	VDD_5V

P9.06-P9.10

P9.06	P9.07	P9.08	P9.10
VDD_5V	SYS_5V	SYS_5V	SYS_RSTN

Pin	P9.09
Name	HSIO19PB0
BALL	W19

P9.11-P9.13

Pin	P9.11	P9.12	P9.13
Name	GPIO38NB1/DQS	GPIO38PB1/DQS/CCC_SW_PLL1_OUT0	GPIO2NB1/DQS
BALL	B5	C5	D19
DEFAULT	MMUART4	core_gpio[1] @ 0x41200000	MMUART4
	UART4 RX	GPIO	UART4 TX
GPIOS	core_gpio[0] @ 0x41200000	core_gpio[1] @ 0x41200000	core_gpio[2] @ 0x41200000
	GPIO	GPIO	GPIO
ROBOTICS	~	core_gpio[0] @ 0x41200000	core_gpio[7] @ 0x41200000
	~	GPIO	GPIO

P9.14-P9.16

Pin	P9.14	P9.15		P9.16
Name	GPIO39PB1/CLKIN_S_2/CCC_SW_CLKIN_S_2/CCC_SW_PLL1_O	GPIO40NB1		GPIO40PB1/CCC_SW_PLL1_OUT1
BALL	C6	A5		A6
DEFAULT	core_pwm[0] @ 0x41400000	core_gpio[4] @ 0x41200000		core_pwm[1] @ 0x41400000
	PWM_1:0	GPIO		PWM_1:1
GOIOS	core_gpio[3] @ 0x41200000	core_gpio[4] @ 0x41200000		core_gpio[5] @ 0x41200000
	GPIO	GPIO		GPIO
ROBOTICS	core_pwm[0] @ 0x41400000	core_gpio[1] @ 0x41200000		core_pwm[1] @ 0x41400000
	PWM_1:0	GPIO		PWM_1:1

P9.17-P9.19

Pin	P9.17	P9.18	P9.19
Name	GPIO44NB1/DQS	GPIO44PB1/DQS/CCC_SW_PLL0_OUT0	GPIO45PB1/CCC_SW_PLL0_OUT0
BALL	C9	C10	A10
DEFAULT	~	~	MSS I2C0
	~	~	I2C0 SCL
GPIOS	core_gpio[6] @ 0x41200000	core_gpio[7] @ 0x41200000	MSS I2C0
	GPIO	GPIO	I2C0 SCL
ROBOTICS	~	~	MSS I2C0
	~	~	I2C0 SCL

P9.20-P9.22

Pin	P9.20	P9.21	P9.22
Name	GPIO45NB1	GPIO43NB1	GPIO43PB1
BALL	A11	B8	A8
DEFAULT	MSS I2C0	~	~
	I2C0 SDA	~	~
GPIOS	MSS I2C0	core_gpio[8] @ 0x41200000	core_gpio[8] @ 0x41200000
	I2C0 SDA	GPIO	GPIO
ROBOTICS	MSS I2C0	~	~
	I2C0 SDA	~	~

P9.23-P9.25

Pin	P9.23	P9.24	P9.25
Name	GPIO48NB1	GPIO48PB1/CLKIN_S_4	GPIO41NB1
BALL	C12	B12	B7
DEFAULT	core_gpio[10] @ 0x41200000	~	core_gpio[12] @ 0x41200000
	GPIO	~	GPIO
GPIOS	core_gpio[10] @ 0x41200000	core_gpio[11] @ 0x41200000	core_gpio[12] @ 0x41200000
	GPIO	GPIO	GPIO
ROBOTICS	core_gpio[2] @ 0x41200000	~	core_gpio[3] @ 0x41200000
	GPIO	~	GPIO

P9.26-P9.28

Pin	P9.26	P9.27	P9.28
Name	GPIO41PB1/CLKIN_S_3/CCC_SW_CLKIN_S_3	GPIO46NB1	GPIO46PB1/CCC_SW_PLL0_OUT1
BALL	A7	D11	C11
DEFAULT	~	core_gpio[14] @ 0x41200000	~
	~	GPIO	~
GPIOS	core_gpio[13] @ 0x41200000	core_gpio[14] @ 0x41200000	core_gpio[15] @ 0x41200000
	GPIO	GPIO	GPIO
ROBOTICS	~	~	~
	~	~	~

P9.29-P9.31

Pin	P9.29	P9.30	P9.31
Name	GPIO1PB1/CLKIN_S_9/CCC_SE_CLKIN_S_9	GPIO1NB1	GPIO4NB1
BALL	F17	F16	E18
DEFAULT	~	core_gpio[17] @ 0x41200000	~
	~	GPIO	~
GPIOS	core_gpio[16] @ 0x41200000	core_gpio[17] @ 0x41200000	core_gpio[18] @ 0x41200000
	GPIO	GPIO	GPIO
ROBOTICS	~	core_gpio[5] @ 0x41200000	~
	~	GPIO	~

P9.32-P9.40

P9.32	P9.34
VDD_ADC	GND

P9.33	P9.35	P9.36	P9.37	P9.38	P9.39	P9.40
AIN4	AIN6	AIN5	AIN2	AIN3	AIN0	AIN1

P9.41-P9.42

Pin	P9.41	P9.42
Name	GPIO0PB1/CLKIN_S_8/CCC_SE_CLKIN_S_8/CCC_SE_PLL0_OUT0	GPIO0NB1
BALL	E15	E14
DEFAULT	core_gpio[19] @ 0x41200000	core_pwm[0] @ 0x41000000
	GPIO	PWM_0:0
GPIOS	core_gpio[19] @ 0x41200000	core_gpio[20] @ 0x41200000
	GPIO	GPIO
ROBOTICS	core_gpio[19] @ 0x41200000	~
	GPIO	~

P9.43-P9.46

P9.43	P9.44	P9.45	P9.46
GND	GND	GND	GND

Chapter 5

Demos

5.1 Upgrade BeagleV-Fire Gateware

This document describes how to upgrade your BeagleV-Fire's gateware. This approach can be used out of the box using Linux commands executed on BeagleV-Fire

5.1.1 Required Equipment

- BeagleV-Fire board
- USB-C cable
- Ethernet cable

The USB-C cable provides power, a serial interface to BeagleV-Fire and allows connecting to BeagleV-Fire through a browser using IP address `192.168.7.2.`

The Ethernet cable connected to your local network (LAN) allows connecting to BeagleV-Fire using the SSH protocol. It also allows BeagleV-Fire to retrieve updated packages through your local network's Internet connection.

5.1.2 Connect to BeagleV-Fire Linux Command Line Interface

BeagleV-Fire boots Linux out of the box. Like all Beagleboard boards there are several methods to get BeagleV-Fire's Linux command prompt.

- Cockpit
- SSH
- Serial port

Cockpit

Enter the following URL in your web browser: `https://beaglev.localdomain:9090/`

On first use, click through the security warning. Login using `beagle/temppwd` as user/password. Click on Terminal in the left pane. You now have a Linux command prompt running on your BeagleV-Fire. Next step: enter the commands described in the Gateware Upgrade Linux Commands section of this document.

Note: You can connect to the Cockpit using the IP address dynamically assigned to your BeagleV-Fire in your local Ethernet network. One method of finding the value of that dynamically assigned IP address is to open

---

a serial terminal though the USB port and use the `ip address` Linux command. Please refer to the USB Serial Port section.

SSH

Like all Beagleboard boards, you can SSH to the board through the USB interface by using IP address 192.168.7.2.

Note: On Windows, this approach may require some drivers to be updated or installed. Use one of the other approaches if you are not immediately successful with this one. You can circle back later to adjust your Windows installation if required.

Serial Port

A serial port is available through the USB-C port. This serial port becomes available once Linux has booted on BeagleV-Fire. Please wait a couple of minutes after powering up the board before looking for additional serial ports reported by your host computer's operating system. You can then use your favorite serial port terminal tool such as Putty or Screen to access the BeagleV-Fire Linux command prompt.

For example on your Linux host computer:

```
screen /dev/ttyACM0 115200
```

Where `ttyACM0` is an additional serial port that appeared after BeagleV-Fire was connected to your Linux host computer. This serial port can be identified using the `dmesg | grep tty` Linux command which will show the most recent serial port added to the host computer.

On Windows, BeagleV-Fire's serial port number will show in the Windows Device Manager. Use that serial port number in Putty with a speed 115200 baud, no flow-control.

5.1.3 Gateware Upgrade Linux Commands

Note: BeagleV-Fire needs to be connected to the internet through your local network for the commands in this section to work. The connection can be through the Ethernet port or the Wi-Fi module.

Install bbb.io-gateware

You need to install the bbb.io-gateware package. This will allow retrieving the most up-to-date gateware.

```
sudo apt install bbb.io-gateware
```

Retrieve Available Updated Linux packages List

The list will include the latest BeagleV-Fire gateware packages.

```
sudo apt update
```

Upgrade Linux Packages

This will upgrade the BeagleV-Fire gateware Linux programming files located under `/usr/share/beagleboard/gateware`. Several directories are found in that location, each containing programming files for one individual gateware configuration.

```
sudo apt upgrade
```

Launch Reprogramming of BeagleV-Fire's FPGA

Change directory to `/usr/share/beagleboard/gateware`. This directory contains a script performing the gateware's reprogramming. It also contains one directory for each of the possible gateware configuration that can be programmed into your BeagleV-Fire. The name of one of these directories is passed as argument to the script to specify which gateware configuration you wish to program your BeagleV-Fire with.

```
cd /usr/share/beagleboard/gateware
. ./change-gateware.sh default
```

Important: Do not power-off BeagleV-Fire until it has rebooted by itself. The gateware reprogramming may take a couple of minutes.

The change-gateware script programs the selected gateware and its associated device tree overlays into the PolarFire SoC System Controllers SPI flash and triggers a software reboot. During the reboot, the Hart Software Services (HSS) will request the PolarFire SoC System Controller to reprogram the FPGA and eNVM. The PolarFire SoC System Controller will reprogram the FPGA if it finds it contains a different design version than the one in the SPI Flash. The board reboots on completion of the FPGA reprogramming.

5.2 Flashing gateware and Linux image

In this tutorial we are going to learn to flash the gateware image to FPGA and `sdcard.image` to eMMC storage.

Important: Additional hardware required:

1. FlashPro5/6 programmer

2. Tag connect TC2050-IDC-NL 10-Pin No-Legs Cable with Ribbon connector

3. TC2050-CLIP-3PACK Retaining CLIP board for TC2050-NL cables

---

5.2.1 Programming & Debug tools installation

To flash a gateware image to your BeagleV-Fire board you will require a FlashPro5/6 and FlashPro Express (FPExpress) tool which comes pre-installed as part of Libero SoC Design Suite. A standalone FlashPro Express tool is also available with MicroChip's Programming and Debug Tools package, which we are going to use for this tutorial. Below are the steps to install the software:

1. Download the zip for your operating system from Programming and Debug Tools page.

2. Unzip the file and in the unzipped folder you will find `launch_installer.sh` and `Program_Debug_v2023.1.bin`.

3. Execute the `launch_installer.sh` script to start the installation procedure.

```
[lorforlinux@fedora Program_Debug_v2023.1_lin] $ ./launch_installer.sh

No additional packages to install for installer usage

Requirement search complete.

See /tmp/check_req_installer608695.log for information.

Launch of installer
Preparing to install
Extracting the JRE from the installer archive...
Unpacking the JRE...
```

Note: It's recommended to install under *home/user/microchip* for linux users.

Enabling non-root user to access FlashPro

1. Download `60-openocd.rules`

2. Copy udev rule `sudo cp 60-openocd.rules /etc/udev/rules.d`

3. Trigger udevadm using `sudo udevadm trigger` or reboot the PC for the changes to take effect

5.2.2 Flashing gateware image

Note: content below is valid for beta testers only.

Launch FPExpress

1. Download `FlashProExpress.zip` file and unzip, it contains the `*.job` file which we need to create a new project in FPExpress.

2. Open up a terminal and go to `/home/user/microchip/Program_Debug_v202X.Y/ Program_Debug_Tool/bin` which includes FPExpress tool.

3. Execute `./FPExpress` in terminal to start FlashPro Express software.

Create new project

Important: Make sure you have your FlashPro5/6 connected before you create a new project.

Press `CTRL+N` to create a file and you will see a pop-up window like shown below,

Follow the steps below as annotated in the image above:

1. Click on browse (1) button to select the job file.

2. Click on browse (2) button to select the project location.

3. Click ok button to finish.

If your FlashPro5/6 is connected properly you'll see the window shown below:

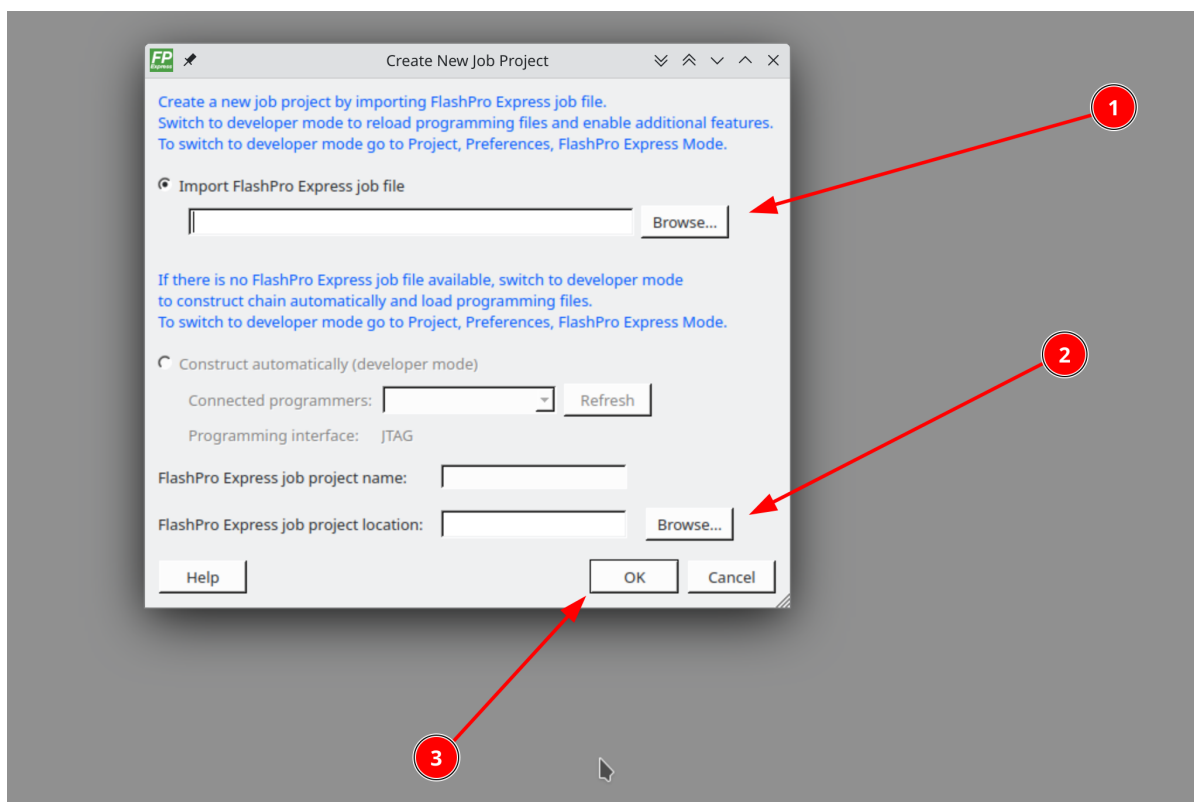Following the annotation in the image above:
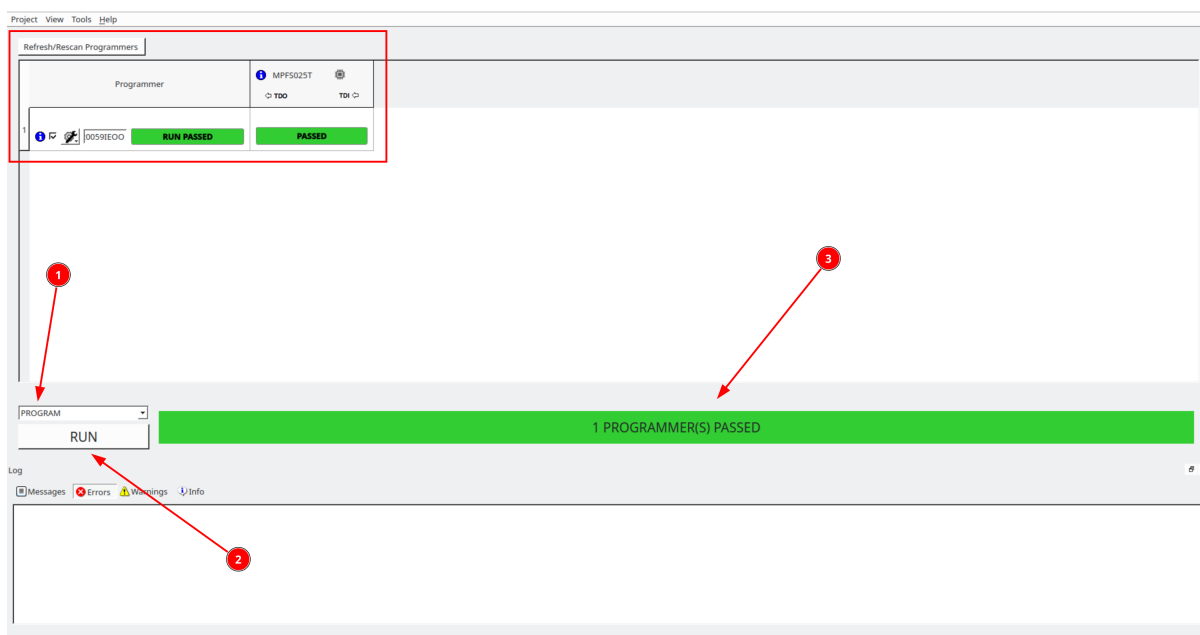
Fig. 5.1: FPExpress new project creation window

Fig. 5.2: FPExpress new project flash window

1. From drop-down select `Program` action

2. Click on `RUN` button

3. Shows the progress

If you see a lot of green color and the progress bar says `PASSED` then well done you have successfully flashed the gateware image on your BeagleV-Fire board.

5.2.3 Flashing eMMC

Connect to BeagleV-Fire UART debug port using a 3.3v USB to UART bridge.
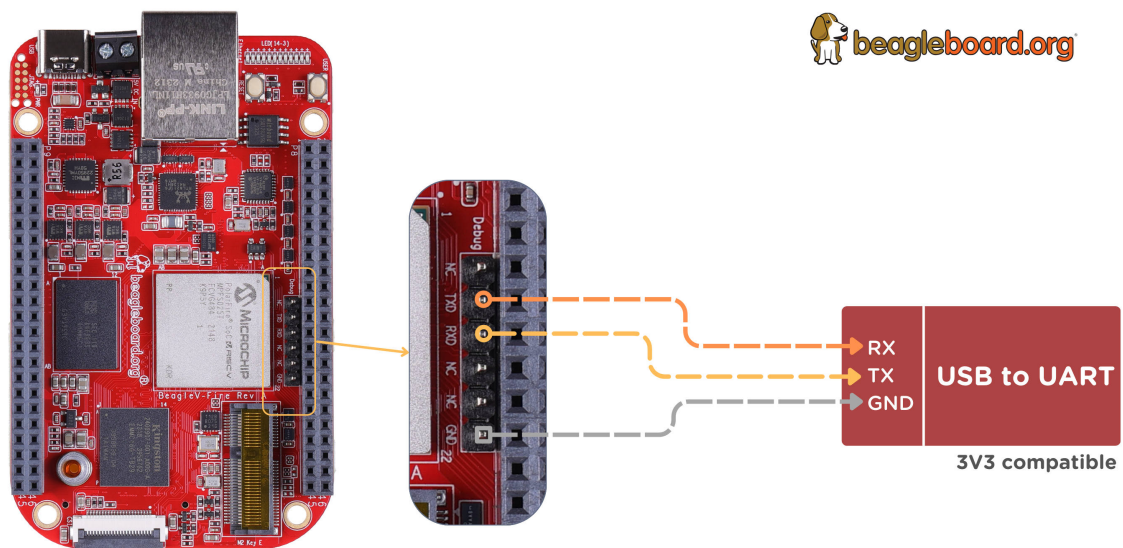
Fig. 5.3: UART debug connection

Now you can run `tio <port> -b 115200` in a terminal window to access the UART debug port connection. Once you are connected properly you can press the Reset button which will show you a progress bar like in the
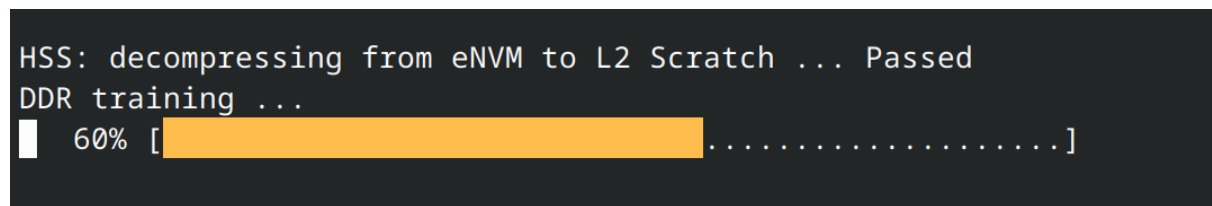
Fig. 5.4: BeagleV-Fire booting HSS with progress bar

Once you see that progress bar on your screen you can start pressing any button (0-9/a-z) which will stop the board from fully booting and you'll be able to access Hart Software Services (HSS) prompt. BeagleV-Fire's eMMC content is written by the Hart Software Services (HSS) using the `usbdmsc` command. The HSS `usbdmsc` command exposes the eMMC as a USB mass storage device USB type C connector.

Once you see `>>` you can execute the commands below:

1. `>> mmc`

```
HSS: decompressing from eNVM to L2 Scratch ... Passed
DDR training ... Passed ( 5306 ms)

-------------------------------
--       BeagleV-Fire        --
-------------------------------

[5.368717] PolarFire(R) SoC Hart Software Services (HSS) - version 0.99.36-BVF-250T-ES_0.0.1-229-g6d298cc
MPFS HAL version 2.2.104 / DDR Driver version 0.4.023 / Mi-V IHC version 0.1.1 / BOARD=bvf
(c) Copyright 2017-2022 Microchip FPGA Embedded Systems Solutions.

incorporating OpenSBI - version 1.2
(c) Copyright 2019-2022 Western Digital Corporation.

[5.401527] Build ID: cd5a966116e51ce2fe76385cf5d0d2d3531d7401
[5.408384] Built with the following tools:
 - riscv64-unknown-elf-gcc (xPack GNU RISC-V Embedded GCC (Microsemi SoftConsole build), 64-bit) 8.3.0
 - GNU ld (xPack GNU RISC-V Embedded GCC (Microsemi SoftConsole build), 64-bit) 2.32

[5.430258] NOTICE: Running from L2 Scratchpad

[5.435900] Serial Number:
72d51339518cdf7078fcde0ac038e8710000000000000000000000000000000000000000000000000000000000000000000
[5.448573] Segment Configuration:
        Cached: SEG0_0: offset 0x0080000000, physical DDR 0x00000000
        Cached: SEG0_1: offset 0x1000000000, physical DDR 0x00000000
    Non-cached: SEG1_2: offset 0x00c0000000, physical DDR 0x00000000
    Non-cached: SEG1_3: offset 0x1400000000, physical DDR 0x00000000
Non-cached WCB: SEG1_4: offset 0x00d0000000, physical DDR 0x00000000
Non-cached WCB: SEG1_5: offset 0x1800000000, physical DDR 0x00000000
[5.489456] L2 Cache Configuration:
    L2-Scratchpad:  4 ways (512 KiB)
        L2-Cache:  8 ways (1024 KiB)
          L2-LIM:  4 ways (512 KiB)
[5.503951] DDR-Lo size is   32 MiB
[5.508204] UART error
[5.511590] DDR-Hi size is 1888 MiB
[5.535425] Design Info:
    Design Name: DEV_BOARD_TESTS_6CECBF8DE4F8CE
    Design Version: 00.00.1
[5.545841] Attempting to select eMMC ... Passed
Press a key to enter CLI, ESC to skip
Timeout in 1 second
.[5.806250] Character 49 pressed

[5.810416] Type HELP for list of commands
[5.815277] >>
[9.427945] >>
[9.595339] >> █
```

Fig. 5.5: BeagleV-Fire boot messages with HSS prompt access

2. `>> usbdmsc`

```
[9.595339] >> mmc
[8647.667675] Selecting SDCARD/MMC (fallback) as boot source ...
[8647.674532] Attempting to select eMMC ... Passed
[8647.929915] >> usbdmsc
[8655.594694] initialize MMC
[8655.598426] Attempting to select eMMC ... Passed
[8655.853809] MMC - 512 byte pages, 512 byte blocks, 30621696 pages
Waiting for USB Host to connect... (CTRL-C to quit)
 . 0 bytes written, 0 bytes readUSB Host connected. Waiting for disconnect... (CTRL-C to quit)
 . 0 bytes written, 2300416 bytes read
```

Fig. 5.6: HSS commands to show eMMC as mass storage

After executing the commands above your BeagleV-Fire's eMMC will be exposed as a mass storage device like shown in the image below:
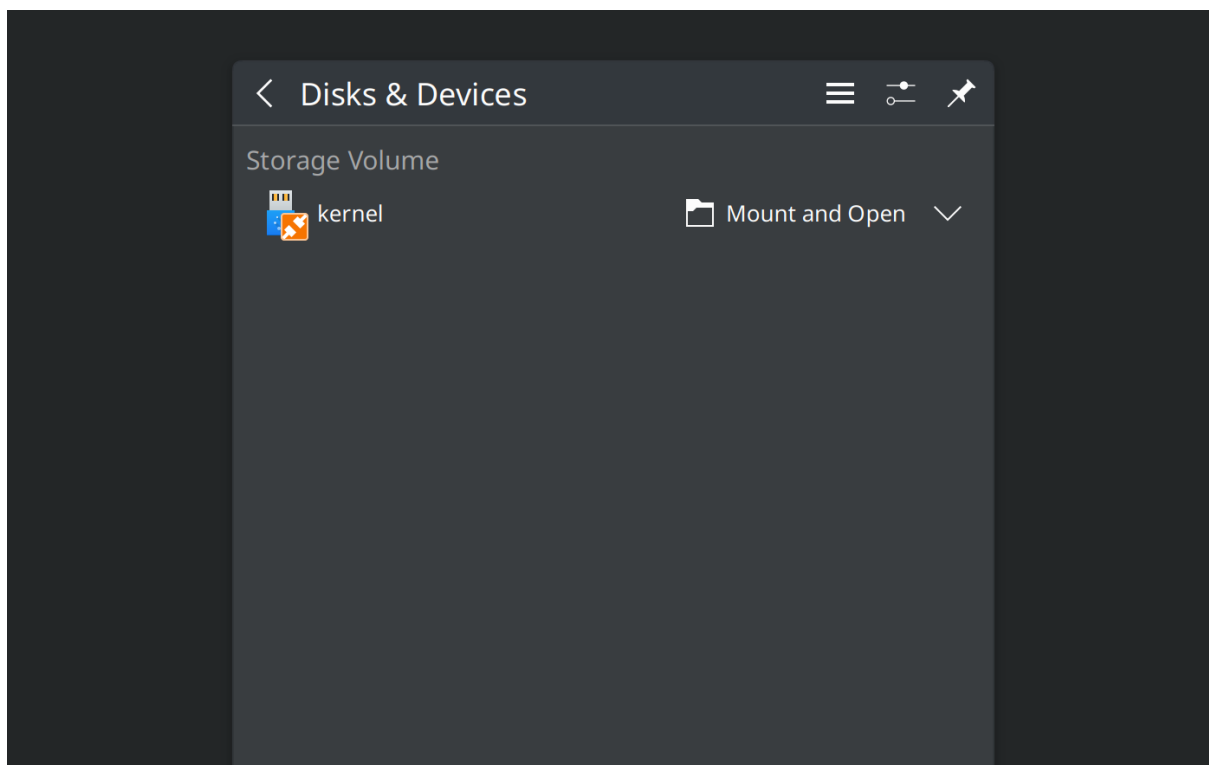

Fig. 5.7: BeagleV-Fire eMMC as mass storage

Once your board is exposed as a mass storage device, you can proceed to flash the `sdcard.img` on your BeagleV-Fire's eMMC.

This document outlines two methods to flash from your local machine to BeagleV Fire's eMMC.

1. First method is to use Balena Etcher software. This software can be used to flash image in either Windows or Linux operating system.

Select image

1. Select the `sdcard.img` file from your local drive storage.
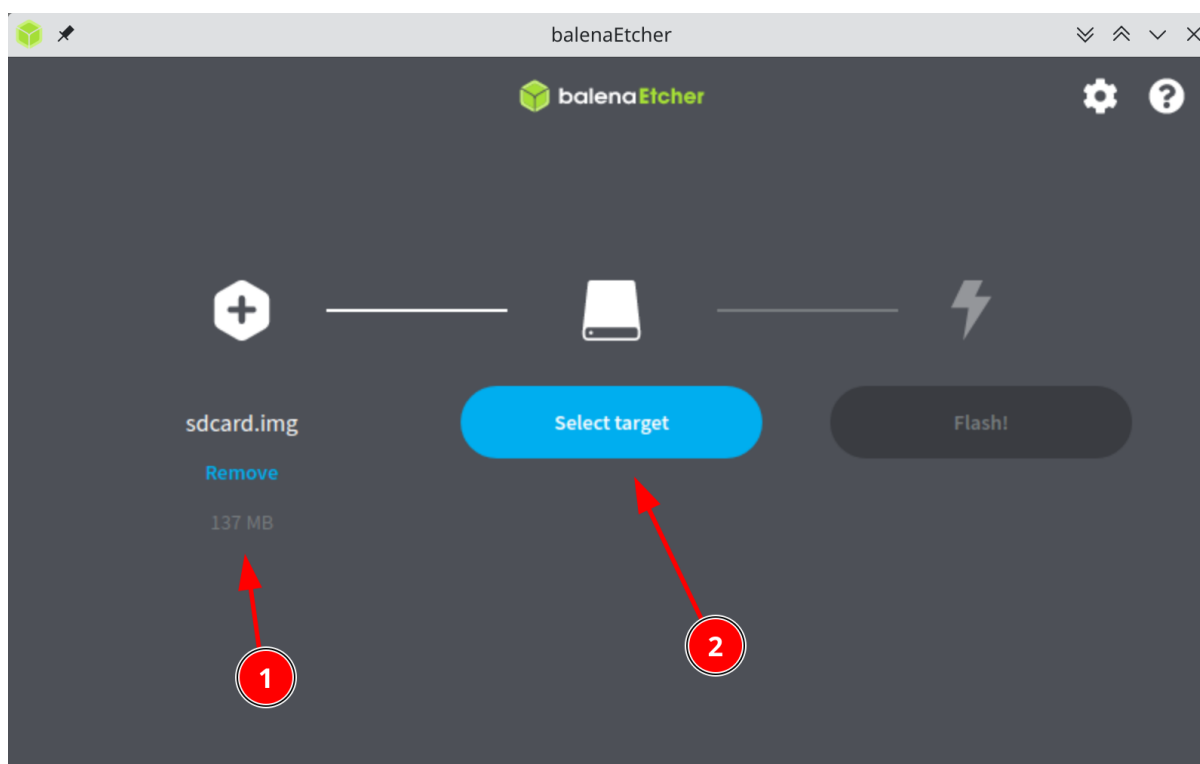
2. Click on select target.

Fig. 5.8: Balena Etcher selecting image

Select Target

1. Select `MCC PolarFireSoC_msd` as target.

2. Click `Select(1)` to proceed.

Flash image

1. Click on `Flash!` to flash the `sdcard.img` on BeagleV-Fire eMMC storage.

2. Second method is suitable for flashing the image using Linux machine via the command line.

```
sudo dd if=output/images/sdcard.img of=/dev/sdX bs=1M status=progress
```

Note: You need to replace /dev/sdX with the actual device name of your eMMC. Use tools like dmesg, lsblk, or GNOME Disks before and after exposing your device as a USB to identify the correct device name. Be very careful not to overwrite the wrong drive, as this action is irreversible.

- Once the transfer is complete, type `CTRL+C` to disconnect your device
- Finally boot the new Linux image by typing `boot` or reset your board

Congratulations! with that done you have fully updated BeagleV-Fire board with up to date gateware image on it's PolarFire SoC's FPGA Fabric and linux image on it's eMMC storage.

5.3 Microchip FPGA Tools Installation Guide

Instructions for installing the Microchip FPGA tools on a Ubuntu 20.04 or Ubuntu 22.04 desktop.
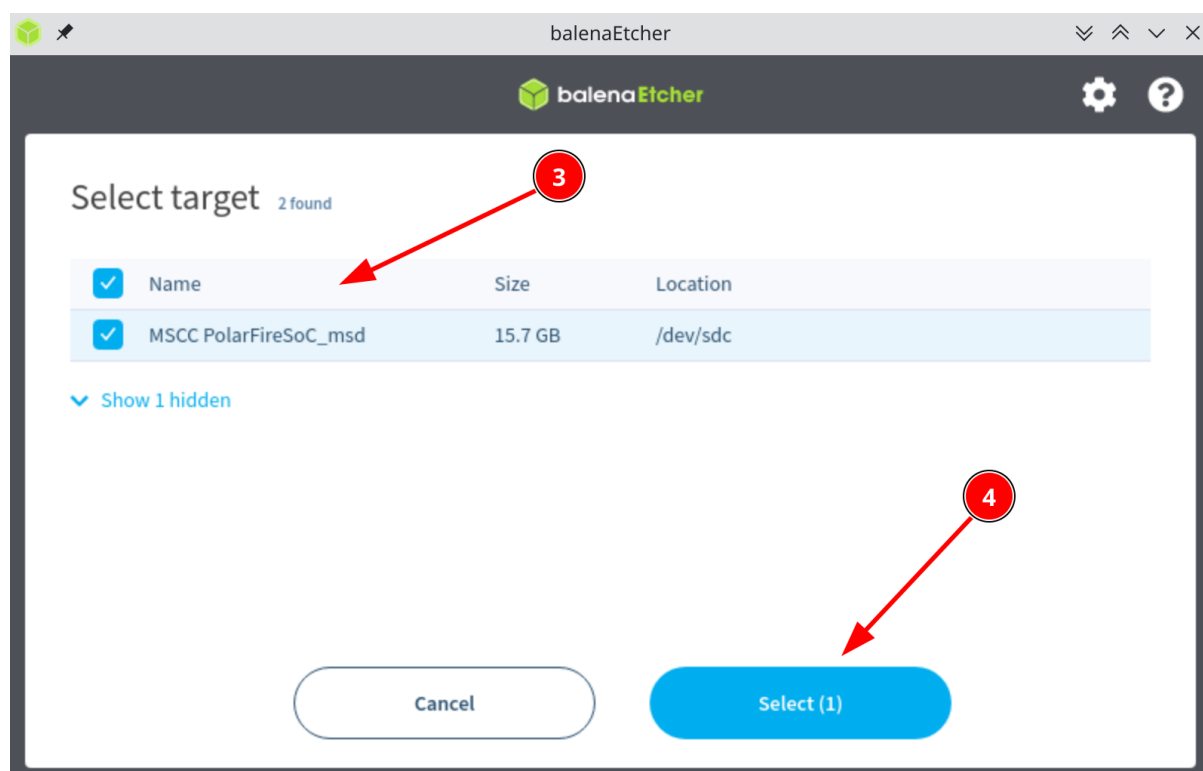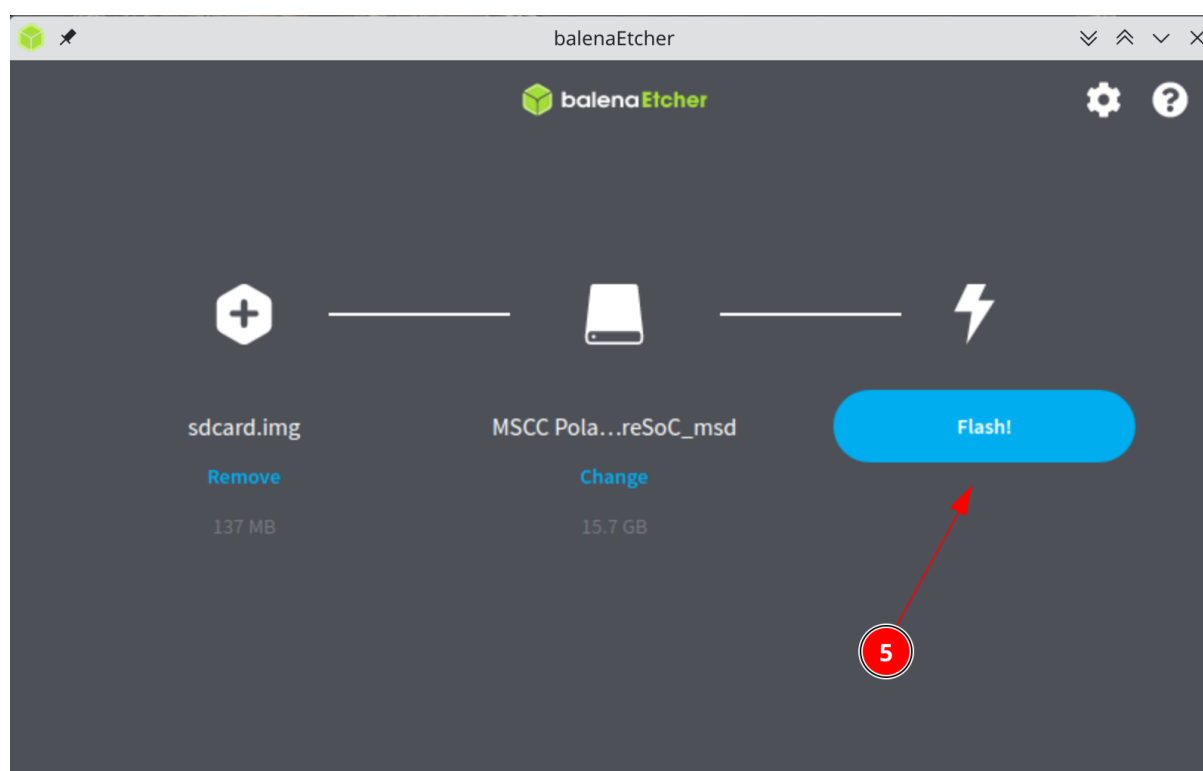
Fig. 5.9: Balena Etcher selecting target

Fig. 5.10: Balena Etcher flashing image

Important: We will be providing instances of Libero that you can run from git.beagleboard.org's gitlab-runners such that you do not need to install the tools on your local machine.

5.3.1 Install Libero

Note: Libero 2023.2, 2024.1 or 2024.2 should work. 2024.2 is used as an example.

Create a folder named Microchip in your /home folder

- Download installer from the Microchip's fpga and soc design tools section.
- Install Libero

```
unzip Libero_SoC_v2024.2_lin.zip

./launch_installer.sh
```

Important: Do not use the default location suggested by the Libero installer. Instead of /usr/local/Microchip/Libero_SoC_v2024.2 install into ~/Microchip/Libero_SoC_v2024.2

Run the post installation script which will install missing packages:

```
sudo /home/$USER/Microchip/Libero_SoC_v2024.2/Logs/req_to_install.sh
```

No need to run the FlashPro hardware installation scripts. This will be taken care of as part of the SoftConsole installation.

5.3.2 Install SoftConsole 2022.2

- Download installer from Microchip website.

```
sudo chmod +x Microchip-SoftConsole-v2022.2-RISC-V-747-linux-x64-installer.
↪run

./Microchip-SoftConsole-v2022.2-RISC-V-747-linux-x64-installer.run
```

Accept the license, Click Forward, Finish.

Perform the post installation steps as described in the html file opened when you click Finish.

Important: Please pay special attention to the "Enabling non-root user to access FlashPro" section of the post-installation instructions. This will actually allow you to program the board using Libero.

5.3.3 Install the Libero licensing daemon

Download the latest 64 bit Licensing Daemons from the Microchip's fpga and soc design tools section

- Linux_Licensing_Daemon_11.19.6.0_64-bit.tar.gz
- Windows_Licensing_Daemon_11.19.6.0_64-bit.zip

Older Daemon downloads can be found at Microchip's daemons section

- Linux_Licensing_Daemon_11.16.1_64-bit.tar.gz

- Windows_Licensing_Daemon_11.16.1_64-bit.zip

Copy the downloaded file to the Microchip directory within your home directory and untar it.

```
cd ~/Microchip

tar -xvf Linux_Licensing_Daemon_11.19.6.0_64-bit.tar.gz
```

Install the Linux Standard Base:

```
sudo apt-get update

sudo apt-get -y install lsb
```

5.3.4 Request a Libero Silver license

- Visit microchip's fpga software products page
- Choose "Libero Silver 1Yr Floating License for Windows/Linux Server" from the list.
- Enter your MAC address and click register.

Note: A MAC address looks something like 12:34:56::78:ab:cd when you use the "ip address" command to find out its value on your Linux machine. However, you need to enter it as 123456abcd in this dialog box.

You will get an email with a License.dat file. Copy it into the ~/Microchip/license directory. Replace *<put.hostname.here>* in *License.dat* with *localhost* and add Linux_Licensing_Daemon as the path to the daemons.

The top of your license file should look something like this after editing. Your daemon files should be located in the Linux_Licensing_Daemon folder inside the Microchip folder.

```
SERVER localhost 001584731680 1702
DAEMON actlmgrd  Linux_Licensing_Daemon/actlmgrd
# Starting Libero SOC v2024.2, customers are recommended ...
# DAEMON mgcld Linux_Licensing_Daemon/mgcld
DAEMON saltd Linux_Licensing_Daemon/saltd
VENDOR snpslmd  Linux_Licensing_Daemon/snpslmd
```

5.3.5 Execute tool setup script

Download the script:

Listing 5.1: Libero environment and license setup script

```bash
#!/bin/bash

# Gets Libero version

dir=$(pwd)
while [[ "$dir" != "/" ]]; do
  for subdir in "$dir"/*/; do
    if [[ $subdir =~ Libero_SoC_v([0-9]+\.[0-9]+)/ ]]; then
      Libero_ver="${BASH_REMATCH[1]}"
      break 2
    fi
  done
  dir=$(dirname "$dir")
done
```

(continues on next page)

```
#Set preferred Libero version here if needed
#Libero_ver=2023.2

echo "Using Libero version:" $Libero_ver

# Check if Libero_ver was set; if not, print an error and exit
if [[ -z "$Libero_ver" ]]; then
  echo "Error: No directory found matching the pattern 'Libero_SoC_vXXXX.Y/'"
  return 1
fi


 ↪#===============================================================================
# Edit the following section with the location where the following tools are
# installed if they aren't in the default location:
#    - SoftConsole (SC_INSTALL_DIR)
#    - Libero (LIBERO_INSTALL_DIR)
#    - Licensing daemon for Libero (LICENSE_DAEMON_DIR)

 ↪#===============================================================================

export SC_INSTALL_DIR=/home/$USER/Microchip/SoftConsole-v2022.2-RISC-V-747
export LIBERO_INSTALL_DIR=/home/$USER/Microchip/Libero_SoC_v$Libero_ver
export LICENSE_DAEMON_DIR=/home/$USER/Microchip/Linux_Licensing_Daemon
export LICENSE_FILE_DIR=/home/$USER/Microchip/license
export SMARTHLS_INSTALL_DIR=$LIBERO_INSTALL_DIR/SmartHLS-$Libero_ver/SmartHLS


 ↪#===============================================================================
# The following was tested on Ubuntu 20.04 with:
#    - Libero 2023.2 and 2024.1
#    - SoftConsole 2022.2
# It was also tested on Ubuntu 22.04 with:
#    - Libero 2024.2
#    - SoftConsole 2022.2

 ↪#===============================================================================

#
# SoftConsole
#
export PATH=$PATH:$SC_INSTALL_DIR/riscv-unknown-elf-gcc/bin
export FPGENPROG=$LIBERO_INSTALL_DIR/Libero/bin64/fpgenprog


#
# Libero
#
export PATH=$PATH:$LIBERO_INSTALL_DIR/Libero/bin:$LIBERO_INSTALL_DIR/Libero/
↪bin64
export PATH=$PATH:$LIBERO_INSTALL_DIR/SynplifyPro/bin
export PATH=$PATH:$LIBERO_INSTALL_DIR/ModelSimPro/modeltech/linuxacoem
export PATH=$PATH:$SMARTHLS_INSTALL_DIR/bin
export PATH=$PATH:$SMARTHLS_INSTALL_DIR/swtools/binutils/riscv-gnu-toolchain/
↪bin

export LOCALE=C
export LD_LIBRARY_PATH=/usr/lib/i386-linux-gnu:$LD_LIBRARY_PATH


#
```

```
# Libero License daemon
#
export LM_LICENSE_FILE=1702@localhost
export SNPSLMD_LICENSE_FILE=1702@localhost

$LICENSE_DAEMON_DIR/lmgrd -c $LICENSE_FILE_DIR/License.dat -l $LICENSE_FILE_
↪DIR/license.log
```

`setup-microchip-tools.sh`

Details:

You can create a folder named FPGA-Tools-Setup and store the file there, although this is not required, as long as it is inside the Microchip folder.

You shouldn't need to edit the script, as long as you have installed Libero inside a folder that follows the Libero_SoC_vXXXX.X format, or if you have multiple Libero versions installed and want to select a preferred one to use.

Source the script:

```
sudo chmod +x setup-microchip-tools.sh

. ./setup-microchip-tools.sh
```

Important: Do not forget the leading dot. It matters. You will need to run this every time you restart your machine.

Optionally, add this to the end of your *~/.bashrc* file to avoid running it each time on startup.

First, open *~/.bashrc*:

```
nano ~/.bashrc
```

Then, add the following lines at the end:

```
cd /home/$USER/Microchip/
. ./setup-microchip-tools.sh
```

You can then start Libero to open an existing Libero project.

```
libero
```

However you will more than likely want to use Libero to run a TCL script that will build a design for you.

```
libero SCRIPT:BUILD_A_DESIGN.tcl
```

5.4 Gateware Design Introduction

The PolarFire SoC device used on BeagleV-Fire is an SoC FPGA which includes a RISC-V processors subsystem and a PolarFire FPGA on the same die. The gateware configures the Microcprosessor subsystem's hardware and programs the FPGA with digital logic allowing customization of the use of BeagleV-Fire connectors.

5.4.1 Gateware Architecture

The diagram below is a simplified overview of the gateware's structure.
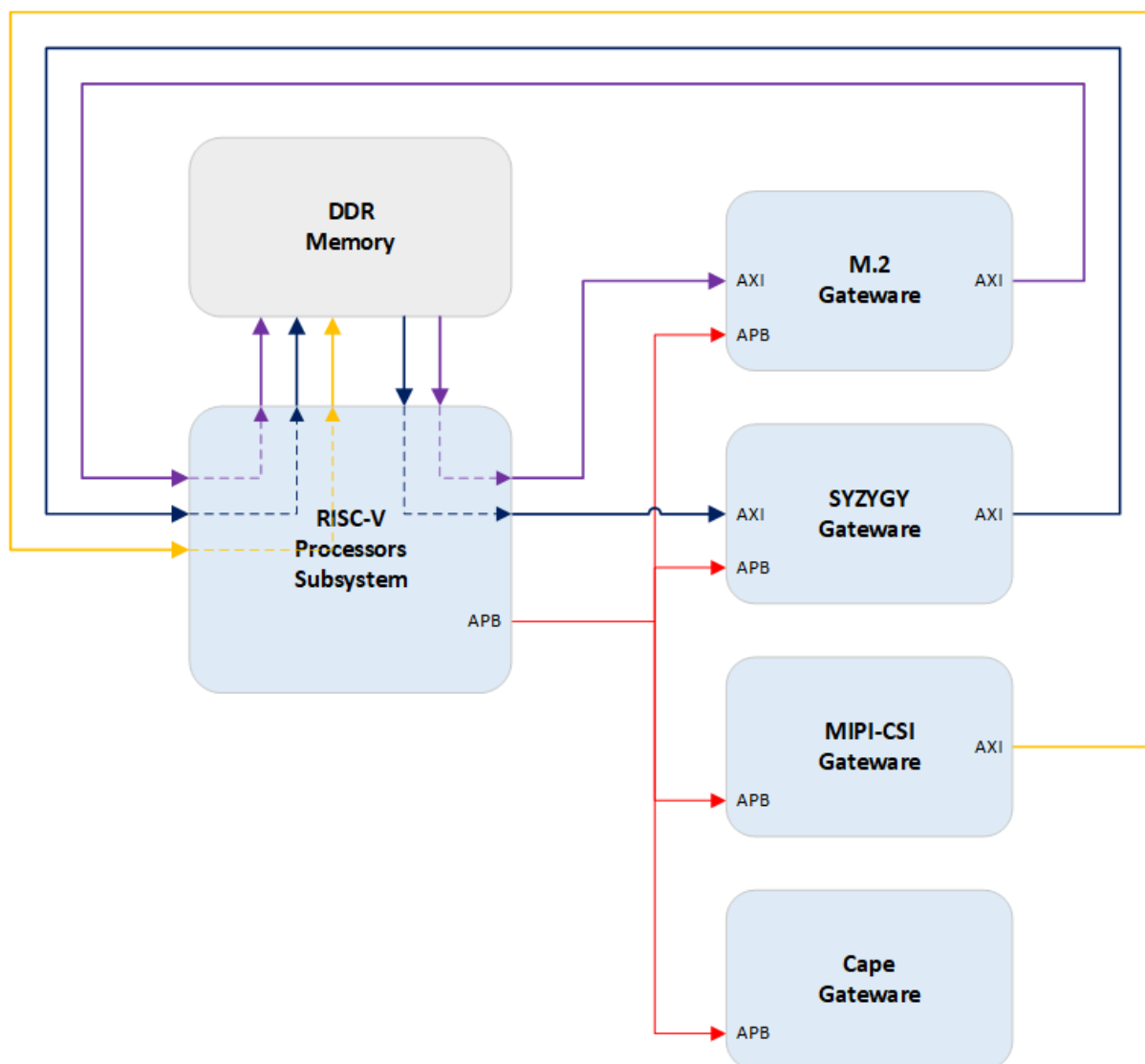
The overall gateware is made-up of several blocks, some of them interchangeable. These blocks are all clocked and reset by another "Clock and Resets" block not showed in the diagram for clarity. A 125MHz, and a 160MHz clock are provided for use by the gateware blocks.

Each gateware block is associated with one of BeagleV-Fire's connectors.

All gateware blocks have an AMBA APB target interface for software to access control and status registers defined by the gateware to operate digital logic defined by the gateware block. This is the software's control path into the gateware block.

Some gateware blocks also have an AMBA AXI target and/or source interfaces. The AXI interfaces are typically used to move high volume of data at high throughput in and out of DDR memory. For example, the M.2 gateware uses these interfaces to transfer data in and out of its PCIe root port.

Cape Gateware

The cape gateware handles the P8 and P9 connectors signals. This is where support for specific capes is implemented.

This is a very good place to start learning about FPGA and how to customize gateware.

SYZYGY Gateware

The SYZYGY gateware handles the high-speed connector signals. This connector includes:

- up to three transceivers capable of 12.7Gbps communications
- One SGMII interface
- 10 high-speed I/Os
- Clock inputs

There is a lot of fun that can be had with this interface given its high-speed capabilities.

Please note that only two tranceivers can be used when the M.2 interface is enabled.

MIPI-CSI Gateware

The MIPI gateware handles the signals coming from the camera interface.

Gateware for the MIPI-CSI interface is Work-In-Progress.

M.2 Gateware

The M.2 gateware implements the PCIe interface used for Wi-Fi modules. It connects the processor subsystem to the PCIe controller associated with the tranceivers bank.

There is limited fun to have here. You either include this block or not in your bitstream.

The M.2 gateware uses one of the four available 12.7 Gbps transceivers. Only two out of the three SYZYGY tranceivers can be used when the M.2 is included in the bitstream. This gateware needs to omited from the bitstream if you want to use all three 12.7Gbps transceivers on the SYZYGY high-speed connector.

RISC-V Processors subsystem

The RISC-V Processors Subsystem also includes some gateware mostly dealing with exposing AMBA bus interfaces for the other gateware blocks to attach to. It also handles immutable aspects of the gateware related to how some PolarFire-SoC signals are used to connect BeagleV-Fire peripherals such as the ADC and EEPROM. As such the RISC-V Processors Subsystem gateware is not intended to be customized.

5.5 How to retrieve BeagleV-Fire's gateware version

There are two methods to find out what gateware is programmed on a board.

5.5.1 Device Tree

The device tree overlays contains the list of gateware blocks included in the overall gateware design. You can retrieve that information using the following command:

```
tree /proc/device-tree/chosen/overlays/
```

This should give an output similar to the one below.

The gateware version can be retrieve by reading one of the overlay files. For example, the command:

```
cat /proc/device-tree/chosen/overlays/ROBOTICS-CAPE-GATEWARE
```

should result in:

where the result of a "git describe" command on the gateware repository is displayed. This provides the most recent tag on the gateware repository followed by information about additionanl commits if some exist. In the example above, the gateware was created from a gateware repository hash 3e0d338 which is 5 commits more recent than tag BVF-0.3.0.

5.5.2 Bootloader messages

The Hart Software Services display the gateware design name and design version retrieve from the FPGA at system start-up.

The design name is the name of the build option selected when using the bitstream-builder to generate the bitstream. The number at the end of the design name is the hash of the gateware repository used to build the bitstream.

The design version is specified as part of the bitstream-builder build configuration option.

Please note that design name "BVF_GATEWARE" indicates that the bitstream used to program the board was generated directly from the gateware repositories scripts and not the bitstream-builder. You might see this when customizing the gateware. Seeing "BVF_GATEWARE" as the design name should be a warning sign that there is a disconnect between the hardware and software on your board.

5.6 Gateware Full Build Flow

5.6.1 Introduction

BeagleV-Fire gateware is made up of several components:

- Digital design for the FPGA fabric.

- Microprocessor Subsystem (MSS) configuration containing MSS configuration register values.

- A zero stage bootloader (HSS).

- A set of device tree ovelays describing the content of the FPGA fabric.

The FPGA's digital design is a combination of:

- HDL/Verilog source code

- TCL scripts configuring IP blocks

- TCL scripts stiching IP blocks together

- Microprocessor Subsystem (MSS) configuration describing the MSS port list

- Pin, placement and timing constraints

The Hart Software Service (HSS) zero stage bootloader

- Configures the PolarFire SoC chip.

- Retrieves the next boot stage from eMMC and hand-over to the next boot stage (e.g. u-boot)

- Makes the board appear as a USB mass-storage for populating the eMMC with secondary boot-loader and operating system image.

The chip configuration applied by the HSS icludes the configuration of:

- Clocks

- Memory controllers

- IOs

- Transceivers

Of course all these components need to be in synch with each other for the system to work properly. This is the reason for using a gateware build system rather than building and tracking each component individually.

5.6.2 Programming artifacts

The gateware builder for BeagleV-Fire produces two programming artifacts:

- A bitstream containing the FPGA fabric and eNVM programming

- A device tree overlay describing the FPGA content.

These two artifacts are packaged differently depending on the programming method. They are merged into a single programming file for DirectC (.dat) and FlashPro Express (.job). They are kept separate for Linux programming (mpfs_bitstream.spi and mpfs_dtbo.spi).

5.6.3 Programming BeagleV-Fire with new gateware

There are several methods possible for programming the BeagleV-Fire with new gateware:

- Linux script executed on the BeagleV-Fire board.

- Running DirectC on anothe single board computer

- Using Microchip's FlashPro Express
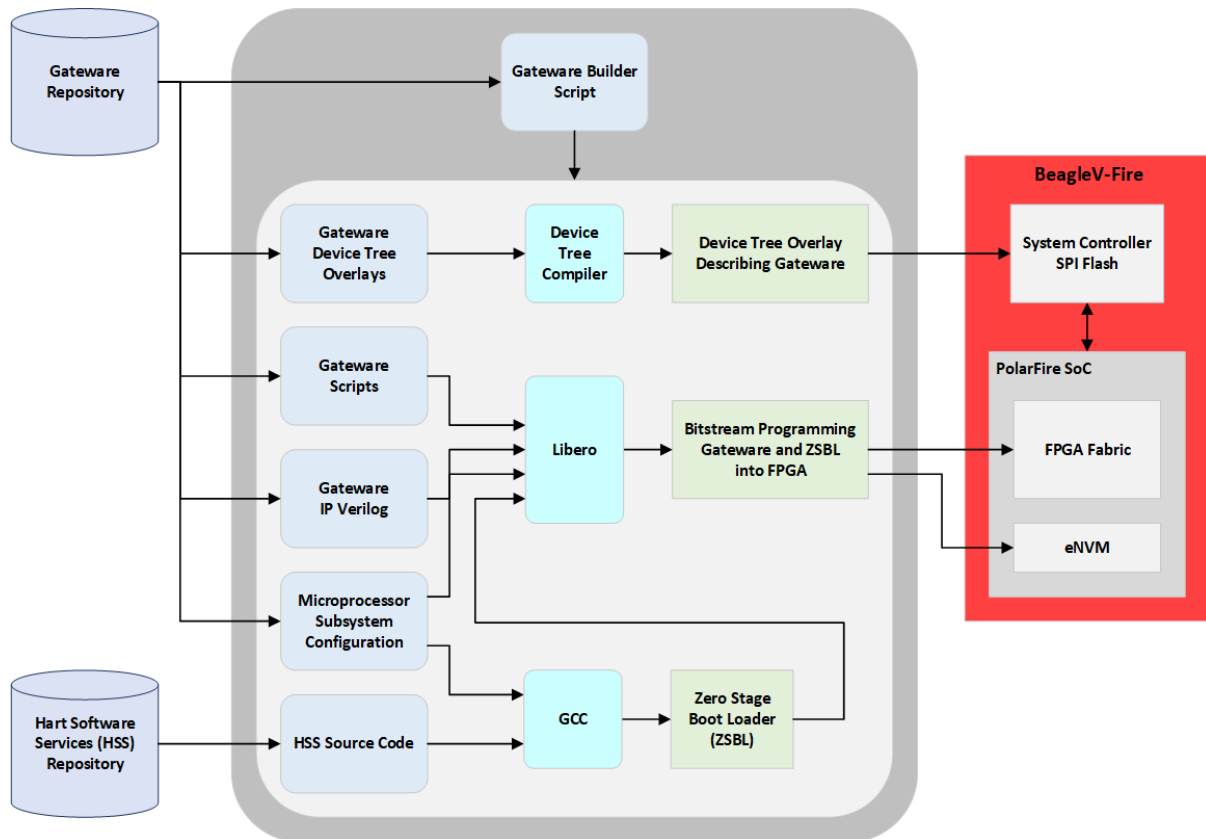
Linux script

This is the recommended approach. It does not require any additional hardware. Simply run the script located in /usr/share/beagleboard/gateware. You should use this methods unless you have soft-bricked your BeagleV-Fire.

DirectC

This approach uses a single board computer (SBC) connected to the BeagleV-Fire JTAG port. The SBC bit-bangs the FPGA programming protocol over GPIOs. This approach is only required for recovering a soft-bricked BeagleV-Fire.

FlashPro Express

This approach uses Microchip's FlashPro Express programming software and a FlashPro6 JTAG programmer. I would recommend using the Linux script even if you are an existing Microchip FPGA user with all the tools. This approach makes most sense when doing bare metal software development and already have a FlashPro programmer and don't care about device tree overlays.

5.7 Gateware TCL Scripts Structure

This document describes the structure of the gateware TCL scripts. It is of interest to understand how to extend or customize the gateware.
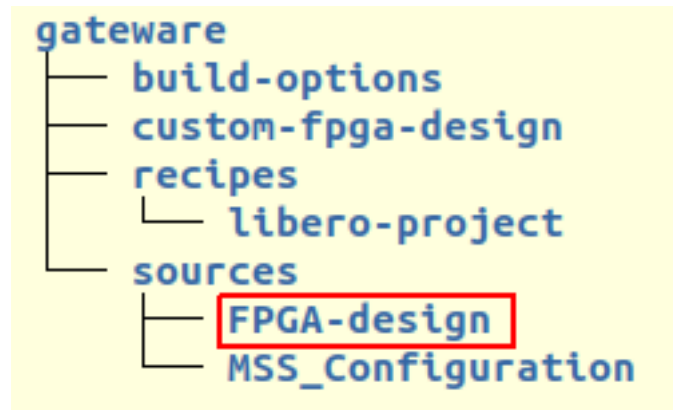
The Libero SoC TCL Command Reference Guide describes the TCL command used in the gateware scripts.

5.7.1 Gateware Project

The gateware project is made up of:

- TCL scripts

- HDL/Verilog source code

- IO pin constraints

- Placement constraints

- Device tree overlays

All these files are found in the FPGA-design directory.

5.7.2 Gateware Components
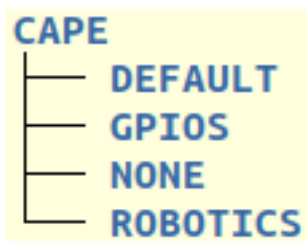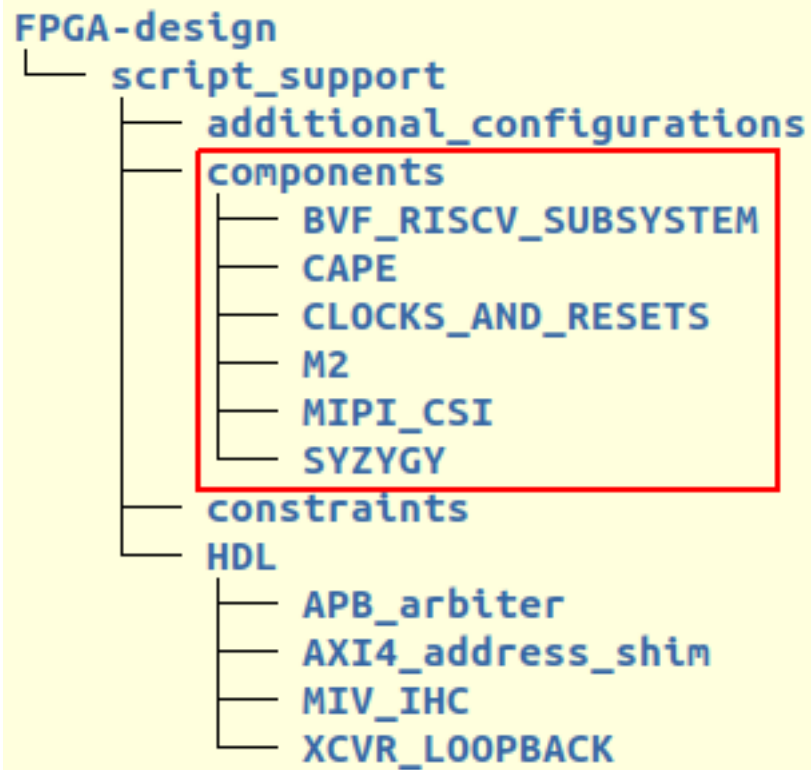
The gateware is organized into 6 components:

- Clocks and reset control

- A base RISC-V microprocessor subsystem

- Cape interface

- M.2 interface

- MIPI camera interface

- SYZYGY high speed interface

5.7.3 Gateware Build Options

Each interface component may have a number of build options. For example, which cape will be supported by the generated gateware.

The name of the directories within the component's directory are the option names passed to the top Libero BUILD_BVF_GATEWATE.tcl script. These directory names are the option name specified in the bitstream builder's build option YAML files.

The gateware is extended or customized by creating additional directories within the component directory of interest. For example, add a MY_CUSTOM_CAPE directory under the CAPE directory to add a gateware build option to support a custom cape.

```
FPGA-design
└── script_support
    ├── additional_configurations
    ├── components
    │   ├── BVF_RISCV_SUBSYSTEM
    │   ├── CAPE
    │   ├── CLOCKS_AND_RESETS
    │   ├── M2
    │   ├── MIPI_CSI
    │   └── SYZYGY
    ├── constraints
    └── HDL
        ├── APB_arbiter
        ├── AXI4_address_shim
        ├── MIV_IHC
        └── XCVR_LOOPBACK
```

```
CAPE
├── DEFAULT
├── GPIOS
├── NONE
└── ROBOTICS
```
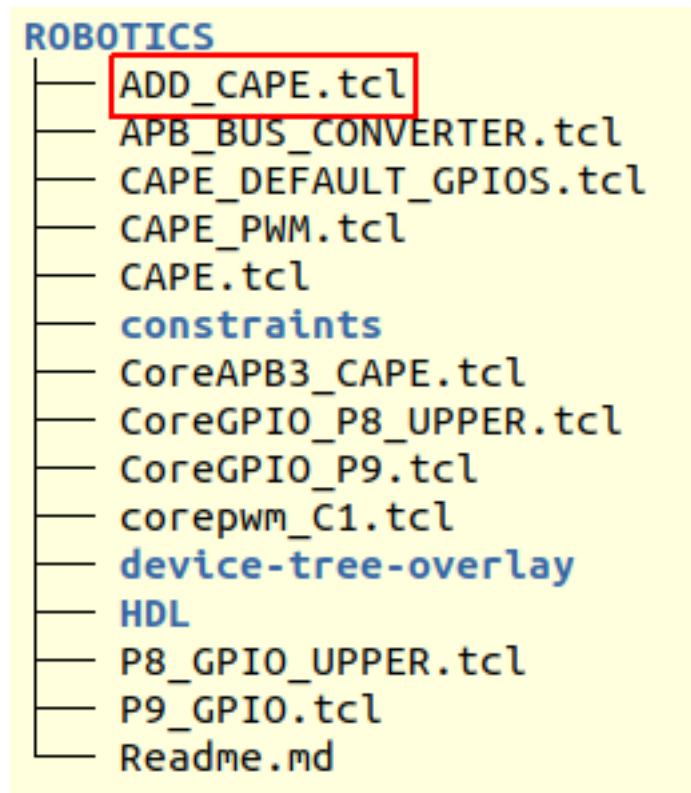
5.7.4 Gateware Component Directories

The component directory contains subdirectories for:

- Constraint files

- Device tree overlay

- Optional HDL/Verilog source code

```
ROBOTICS
├── constraints
├── device-tree-overlay
└── HDL
```
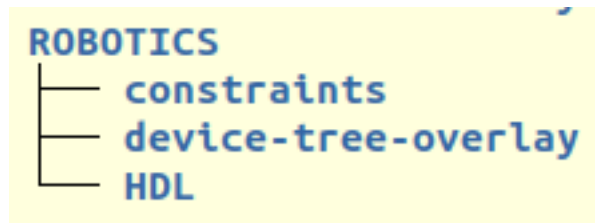
Gateware TCL Scripts

The component directory contains the TCL scripts executed by Libero to generate the gateware. The TCL script framework executes a hand-crafted ADD_<COMPONENT_NAME>.tcl script which instantiates the component and stiches it to the base RISC-V subsystem and top level IOs. The other TCL scripts are typically IP configuration scripts and SmartDesign stiching scripts.

```
ROBOTICS
├── ADD_CAPE.tcl
├── APB_BUS_CONVERTER.tcl
├── CAPE_DEFAULT_GPIOS.tcl
├── CAPE_PWM.tcl
├── CAPE.tcl
├── constraints
├── CoreAPB3_CAPE.tcl
├── CoreGPIO_P8_UPPER.tcl
├── CoreGPIO_P9.tcl
├── corepwm_C1.tcl
├── device-tree-overlay
├── HDL
├── P8_GPIO_UPPER.tcl
├── P9_GPIO.tcl
└── Readme.md
```

5.7.5 Opening the gateware as a libero project

It can be slightly difficult to explore the gateware design through the TCL files. To inspect the gateware design in detail easily, you can open the gateware as a Libero project. This is done by running the following command in the gateware directory:

58   **Chapter 5. Demos**

```
python build-bitstream.py ./build-options/default.yaml # build option
 →depending on the gateware
```

You will need to have all microchip tools installed and the environment variables set up correctly. Refer to the *Microchip tools installation guide* for information on how to install these tools.

5.8 Customize BeagleV-Fire Cape Gateware Using Verilog

This document describes how to customize gateware attached to BeagleV-Fire's cape interface using Verilog as primary language. The methodolgy described can also be applied when using other HDL languages.

It will describe:

- How to generate programming bitstreams without requiring the installation of the Libero FPGA toolchain on your development machine.
- How to use the cape Verilog template
- How to use the git.beagleboard.org CI infrastruture to generate programming bitstreams for your custom gateware

Steps:

1. Fork BeagleV-Fire gateware repository on git.beagleboard.org
2. Create a custom gateware build option
3. Rename a copy of the cape gateware Verilog template
4. Customize the cape's Verilog source code
5. Commit and push changes to your forked repository
6. Retrieve the forked repositories artifacts
7. Program BeagleV-Fire with your custom bitstream

5.8.1 Fork BeagleV-Fire Gateware Repository

Important: All new users need to be manually approved to protect from BOT spam. You will not be able to fork the Gateware Repository until you have been approved. A request to the forum may expedite the process.

Navigate to BeagleV-Fire's gateware source code repository.

Click on the `Forks` button on the top-right corner.

Fig. 5.11: BeagleV-Fire gateware repo fork button

On the Fork Project page, select your namespace and adjust the project name to help you manage multiple custom gateware (e.g. `my-lovely-gateware`). Click the `Fork project` button.

Fig. 5.12: Create gateware fork

Clone the forked repository

```
git clone git@git.beagleboard.org:<MY-NAMESPACE>/my-lovely-gateware.git
```

Where `<MY-NAMESPACE>` is your Gitlab username or namespace.

5.8.2 Create A Custom Gateware Build Option

BeagleV-Fire's gateware build system uses "build configuration" YAML files to describe the combination of gateware components options that will be used to build the gateware programming bitstream. You need to create one such file to describe to the gateware build system that you want your own custom gateware to be built. You need to have one such file describing your gateware in directory `custom-fpga-design`.

Let's modify the `./custom-fpga-design/my_custom_fpga_design.yaml` build configuration file to specify that your custom cape gateware should be included in the gateware bitstream. In this instance will call our custom cape gateware `MY_LOVELY_CAPE`.

```
HSS:
    type: git
    link: https://git.beagleboard.org/beaglev-fire/hart-software-services.git
    branch: develop-beaglev-fire
    board: bvf
gateware:
    type: sources
    build-args: "M2_OPTION:NONE CAPE_OPTION:MY_LOVELY_CAPE" # ①
    unique-design-version: 9.0.2
```

① On the gateware build-args line, replace VERILOG_TUTORIAL with MY_LOVELY_CAPE.

Note: The **custom-fpga-design** directory has a special meaning for the Beagleboard Gitlab CI system. Any build configuration found in this directory will be built by the CI system. This allows generating FPGA programming bitstreams without the requirement for having the Microchip FPGA toolchain installed on your computer.

5.8.3 Rename A Copy Of The Cape Gateware Verilog Template

Move to the cape gateware source code

```
cd my-lovely-gateware/sources/FPGA-design/script_support/components/CAPE
```

Create a directory that will contain your custom cape gateware source code

```
mkdir MY_LOVELY_CAPE
```

Copy the cape Verilog template

```
cp -r VERILOG_TEMPLATE/* ./MY_LOVELY_CAPE/
```

5.8.4 Customize The Cape's Verilog Source Code

Move to your custom gateware source directory

```
cd MY_LOVELY_CAPE
```

You will need to first edit the `ADD_CAPE.tcl` TCL script to use your source code within your custom gateware directory and not the Verilog template source code. In this example this means using source code within the `MY_LOVELY_CAPE` directory rather the VERILOG_TEMPLATE directory.

Edit ADD_CAPE.tcl

Replace `VERILOG_TEMPLATE` with `MY_LOVELY_CAPE` in file `ADD_CAPE.tcl`.

```
#----------------------------------------------------------------------------
↪---
# Import HDL source files
#----------------------------------------------------------------------------
↪---
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪apb_ctrl_status.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪P8_IOPADS.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪P9_11_18_IOPADS.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪P9_21_31_IOPADS.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪P9_41_42_IOPADS.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪CAPE.v}
```

Add the path to your additional Verilog source code files.

```
#----------------------------------------------------------------------------
↪---
# Import HDL source files
#----------------------------------------------------------------------------
↪---
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
```

```
↪blinky.v} // ①
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪apb_ctrl_status.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪P8_IOPADS.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪P9_11_18_IOPADS.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪P9_21_31_IOPADS.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪P9_41_42_IOPADS.v}
import_files -hdl_source {script_support/components/CAPE/MY_LOVELY_CAPE/HDL/
↪CAPE.v}
```

① In our case we will be adding a new Verilog source file called blinky.v.

You will only need to revisit the content of `ADD_CAPE.tcl` if you want to add more Verilog source files or want to modify how the cape interfaces with the rest of the gateware (RISC-V processor subsystem, clock and reset blocks).

Customize The Cape's Verilog source code

We will add a simple Verilog source file, `blinky.v`, in the `MY_LOVELY_CAPE` directory. Code below:

```verilog
`timescale 1ns/100ps
module blinky(
input    clk,
input    resetn,
output   blink
);


reg [22:0] counter;

assign blink = counter[22];

always@(posedge clk or negedge resetn)
begin
    if(~resetn)
        begin
            counter <= 23'h0;
        end
    else
        begin
            counter <= counter + 23'b1;
        end
end
endmodule
```

Let's connect the blinky Verilog module within the cape by editing the `CAPE.v` file.

Add the instantiation of the blinky module:

```verilog
//--------P9_41_42_IOPADS
P9_41_42_IOPADS P9_41_42_IOPADS_0(
        // Inputs
        .GPIO_OE  ( GPIO_OE_const_net_3 ),
        .GPIO_OUT ( GPIO_OUT_const_net_3 ),
        // Outputs
        .GPIO_IN  (  ),
        // Inouts
```

```
        .P9_41    ( P9_41 ),
        .P9_42    ( P9_42 )
        );

//--------blinky
blinky blinky_0(                   // ⬤
        .clk      ( PCLK ),        // ⬤
        .resetn   ( PRESETN ),     // ⬤
        .blink    ( BLINK )        // ⬤
        );

endmodule
```

① Create a blinky module instance called blinky_0.

② Connect the clock using the existing PCLK wire.

③ Connect the reset using the exisitng PRESETS wire.

④ Connect the blinky's blink output using the BLINK wire. This BLINK wire needs to be declared.

Add the BLINK wire:

```
wire              PCLK;
wire              PRESETN;
wire              BLINK;                      // ⬤
wire   [31:0]  APB_SLAVE_PRDATA_net_0;
wire   [27:0]  GPIO_IN_net_1;
```

① Create a wire called BLINK.

The BLINK wire will be used to connect the blinky module's output to a top level output connected to an LED. Do you see where this is going?

Now for the complicated part. We are going to change the wiring of the bi-directional buffers controlling the cape I/Os including the user LEDs.

The original code populates two 43 bits wide wires for controlling the output-enable and output values of the P8 cape connector I/Os. The bottom 28 bits being controlled by the microprocessor subsystem's GPIO block.

```
//-----------------------------------------------------------------
 ↪-
// Concatenation assignments
//-----------------------------------------------------------------
 ↪-
assign GPIO_OE_net_0  = { 16'h0000 , GPIO_OE };
assign GPIO_OUT_net_0 = { 16'h0000 , GPIO_OUT };
```

We are going to hijack the 6th I/O with our blinky's output:

```
//------------------------------------------------------------------
// Concatenation assignments
//------------------------------------------------------------------
assign GPIO_OE_net_0 = { 16'h0000, GPIO_OE[27:6], 1'b1, GPIO_OE[4:0] };      ↵
↪   // ⬤
assign GPIO_OUT_net_0 = { 16'h0000 , GPIO_OUT[27:6], BLINK, GPIO_OUT[4:0] }; ↵
↪   // ⬤
```

① Tie high the output-enable of the 6th bit to constantly enable that output.

② Control the 6th output from the blink module through the WIRE wire.

Edit The Cape's Device Tree Overlay

You should always have a device tree overlay associated with your gateware even if there is limited control from Linux. The device tree overlay is very useful to identify which gateware is currently programmed on your BeagleV-Fire.

```
/dts-v1/;
/plugin/;

&{/chosen} {
    overlays {
        MY-LOVELY-CAPE-GATEWARE = "GATEWARE_GIT_VERSION";   // ①
    };
};
```

① Replace VERILOG-CAPE-GATEWARE with MY-LOVELY-CAPE-GATEWARE.

This change will result in `MY-LOVELY-CAPE-GATEWARE` being visible in `/proc/device-tree/chosen/overlays` at run-time, allowing to check that my lovely gateware is successfully programmed on BeagleV-Fire.

5.8.5 Commit And Push Changes To Your Forked Repository

Move back up to the root directory of your gateware project. This is the my-lovely-gateware directory in our current example.

Add the `my-lovely-gateware/sources/FPGA-design/script_support/components/CAPE/MY_LOVELY_CAPE` directory content to your git repository.

```
git add sources/FPGA-design/script_support/components/CAPE/MY_LOVELY_CAPE/
```

Commit changes to `my-lovely-gateware/custom-fpga-design/my_custom_fpga_design.yaml`

```
git commit -m "Add my lovely gateware."
```

Push changes to your beagleboard Gitlab repository:

```
git push
```

5.8.6 Retrieve The Forked Repositories Artifacts

Navigate to your forked repository. Click Pipelines in the left pane then the Download Artifacts button on the right handside. Select `build-job:archive`. This will result in an `artifacts.zip` file being downloaded.
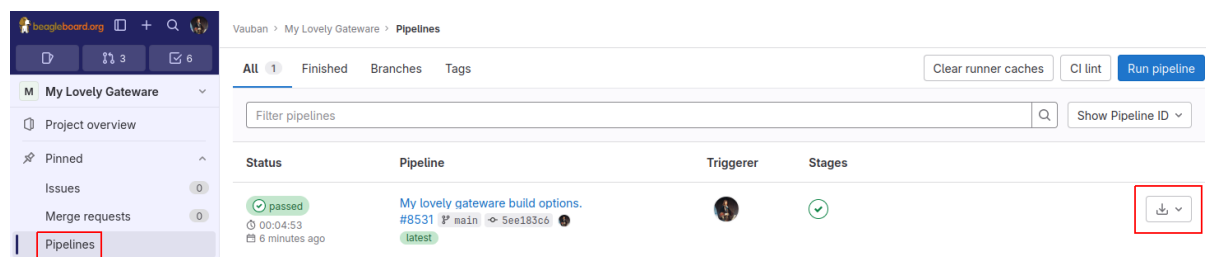

Fig. 5.13: gateware pipeline

5.8.7 Program BeagleV-Fire With Your Custom Bitstream

Unzip the downloaded `artifacts.zip` file. Go to the `gateware-builds-tester/artifacts/bitstreams` directory:

```
cd gateware-builds-tester/artifacts/bitstreams
```

On your Linux host development computer, use the scp command to copy the bitstream to BeagleV-Fire home directory, replacing <IP_ADDRESS> with the IP address of your BeagleV-Fire.

```
scp -r ./my_custom_fpga_design beagle@<IP_ADDRESS>:/home/beagle/
```

On BeagleV-Fire, initiate the reprogramming of the FPGA with your gateware bitstream:
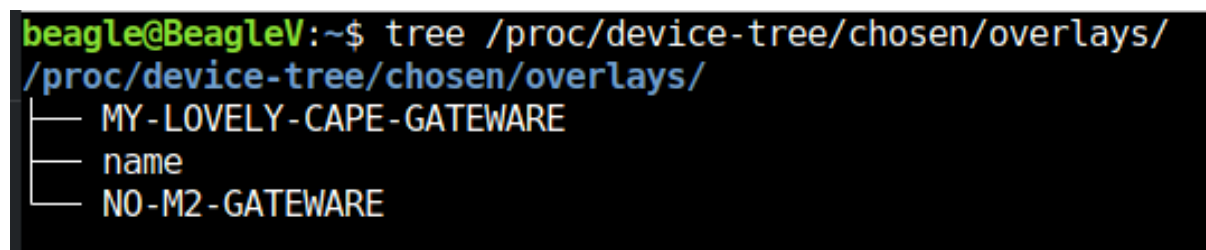
```
sudo /usr/share/beagleboard/gateware/change-gateware.sh ./my_custom_fpga_
→design
```

Wait for a couple of minutes for the BeagleV-Fire to reprogram itself.

You will see the 6th user LED flash once the board is reprogrammed. That's the Verilog you added blinking the LED.

On BeagleV-Fire, You can check that your gateware was loaded using the following command to see the device tree overlays:

```
tree /proc/device-tree/chosen/overlays/
```


Fig. 5.14: gateware lovely overlay

5.9 How to use PicoRV Softcore on BeagleV-Fire

5.9.1 Introduction

The PicoRV Softcore is a 32-bit RISC-V CPU subsystem on the BeagleV-Fire's FPGA fabric, functionally equivalent to the PRU subsystem on the BeagleBone Black. The core is designed for low-latency I/O operations and offloading smaller tasks like PWM, LEDS, etc .

5.9.2 Prerequisites

To be able to use the softcore component, you'll need to have following packages installed on your host computer:

- gcc-riscv64-unknown-elf
- picolibc-riscv64-unknown-elf

Note: These packages are used for compiling the code for the softcore.

For debian-based systems like ubuntu, you can use following commands:

```
sudo apt install gcc-riscv64-unknown-elf picolibc-riscv64-unknown-elf
```

If you are using any other linux distro these packages can be downloaded from respective package managers. But, it is highly recommended to use docker for accurate results.

Docker installation steps.

Once you have docker installed:

1. Start a container with ubuntu image.

2. Install the required toolchain using above mentioned command (remember not to use 'sudo' in docker).

```
Sudo docker run -it ubuntu:latest
```

After installing the necessary tools, you are all set for using the Softcore.

5.9.3 How to use the Softcore

1. Boot the softcore with firmware

Step 1 Navigate to `sources/FPGA-design/script_support/components/SOFTCORE/ PICO_RISCV/firmware` in the BeagleV-Fire repository.

Step 2 Edit the firmware.c file by adding your code in the main function. You can create your own custom file but remember to use firmware.h file for pre-defined functions to access gpios.

Step 3 Now run the `generatehex.sh` to compile the code. (remember to edit the file name in the script of you are using custom file)

```
. ./generatehex.sh
```

If you are using docker clone the repository on your docker container or use docker copy, then compile the program and use docker copy to obtain the hex file.

To copy file from local system to docker container:

```
docker cp ./path/to/file CONTAINER:/path/to/directory
```

To copy file from docker container to local system:

```
docker cp CONTAINER:/path/to/file /path/to/directory
```

To check the name of docker 'CONTAINER':

```
docker ps -a
```

This will list all the running and stopped containers To restart a stopped container, use:

```
sudo docker start name_of_container -i
```

Step 4 Select PicoRV build option for bitstream generation

Once the code is compiled, build the bitstream using the 'build-bitstream.py' script.

```
python build-bitstream.py ./build-options/picorv-softcore.yaml
```

Note: The Gitlab CI does not support the softcore code compilation yet, so the bitstream must be generated locally.

Step 5 Program BeagleV-Fire With Your Custom Bitstream.

2. Programming the softcore in run-time

Step 1

- Program the softcore with custom bitstream with picorv-softcore build-option.

- Follow the steps mentioned in previous section.

Step 2

- Edit and compile your new code in your local machine.(BeagleV-Fire does not support on-board compilation for softcore programs yet, this feature will be added soon.)

- Follow Steps 2 and 3 in previous section for compiling your new program.

Step 3 Copy the firmware directory from BeagleV-Fire repository to your board.

```
sudo scp -r path/to/firmware beagle@192.168.7.2:/home/beagle
```

Step 4

- Overwritting softcore's program memory.

- Compile the 'AXI_test.c' file using gcc and execute it.

```
gcc -o AXI_test AXI_test.c
sudo ./AXI_teste
```

- This will open a interactive menu with various option that you can try out and access the program memory of the softcore.

- Here we will select *option 2* to upload the hex file we just compiled to the softcore.

Note: This compilation in step 2 is to generate a hex file to program the Softcore, while the compilation in Step 3 is performed to command the Linux system on-board to access the program memory of the Softcore.

Note: The filename in **AXI_test.c** for the hex file is set as `firmware.hex`, so make sure the name of the hex file you compiled is the same else change the filename pointer in **AXI_test.c**.

Repository

PicoRV softcore builder.

5.10   How to build the BeagleV-Fire Gateware on Windows

5.10.1   Introduction

The BeagleV Fire gateware builder is a Python script that builds both the PolarFire SoC HSS bootloader and Libero FPGA project into a single programming bitstream. It uses a list of repositories/branches specifying the configuration of the BeagleV Fire to build.

5.10.2   Prerequisites

Tools

To be able to use the bitstream builder on Windows, you will need to install the following tools:

- Msys2-Mingw

- Make

- wsl

Please follow the installation instructions for Msys2 available at https://www.msys2.org/wiki/MSYS2-installation/

When installing *make* in your mysys2 terminal you're recommended to use the default command

```
pacman -S make
```

For those requiring a specific version of *make*, refer to the porting guide at https://www.msys2.org/wiki/Porting/

```
pacman -S <target>-make
```

Ensure that the *Msys2* bin path (e.g., *C:msys64usrbin*) is added to your system's environment variable PATH.

To enable and install WSL, follow these steps:

- Search for "Turn Windows features on or off" in the Windows start menu.

- Select "Windows Subsystem for Linux" and click OK.

- Open a command prompt as an administrator and execute:

```
wsl.exe --install
```

After installing the necessary tools, proceed to the repository and follow the instructions in the README to build the bitstream on Windows

Repository

Access the BeagleV-Fire gateware builder repository at https://openbeagle.org/cyril-jean/gateware-maintenance/

Note: If you encounter an end-of-line error (CRLF/LF) during the build process, change the local Git configuration *core.autocrlf* to false and clone the repository again

```
git config --global core.autocrlf false
```

Note:

- Should the build fail due to an unrecognized Python package, despite the package being installed, it may be due to multiple Python/pip versions. Reinstall the package using.

```
python -m pip install <package-name>
```

- Verify that the LM_LICENSE_FILE environment variable includes licenses for all required programs to avoid silent errors during the build process

5.11 Exploring Gateware Design with Libero

In this demonstration, we'll be exploring the BeagleV-Fire gateware in the Libero Design Suite, making changes to the default gateware. This demo will serve as an introduction to the design tool, an alternative method for developing gateware.

5.11.1 Prerequisites

The prerequisites required for creating the Libero project locally are:

1. Microchip design tools: Refer to the document here for *installation instructions* of microchip FPGA tools.

2. Python requirements for gateware build scripts:

```
pip3 install gitpython
pip3 install pyyaml
```

3. Build requirements:

```
sudo apt install device-tree-compiler
```

Tip: For convience, you can install a python command alias like so:

```
sudo apt install python-is-python3
```

This is optional, but remember to use `python3` in later command examples if you don't.

4. **Environment variables: The following environment variables are required for compilation:**
   - `SC_INSTALL_DIR`
   - `FPGENPROG`
   - `LIBERO_INSTALL_DIR`
   - `LM_LICENSE_FILE`

   A script is provided for setting up these variables in the *fpga tools installation* section.
   An example script for setting up the environment is available here.

5. It is highly recommended to go through the *Customize BeagleV-Fire Cape Gateware Using Verilog* tutorial to understand the basics of the gateware structure.

5.11.2 Cloning and Building the Gateware

First, we must source the environment to include the microchip tools.

```
source /path/to/microchip/fpga/tools/setup-microchip-tools.sh
```

Next, we'll clone the gateware repository to get a local copy of the project.

```
git clone https://openbeagle.org/beaglev-fire/gateware.git
cd gateware
```

We can then use the `build-bitstream.py` script to generate a Libero project for us,
where we can start making our changes.

Important:

Make sure to source the microchip setup script before starting the next command.
This is required every time you open a new terminal.

```
python build-bitstream.py ./build-options/default.yaml # exploring the
 ↪default gateware
```

This should start a big log stating the compilation of the project. First, the device tree overlays are compiled, which contain information for linux about the gateware.

Next, the Hart Software Services (HSS) is compiled. This acts as a zero-stage bootloader, configuring the Polarfire SoC and allowing services like loading the next stage bootloader and flashing the eMMC of the board.

Then the libero project generating is started. Here, TCL scripts inside the `sources` directory are executed, starting with the `BUILD_BVF_GATEWARE.tcl` script. This stitches each HDL module, IP, hardware configuration together in the gateware.

Once bitstream generation is completed, the Libero project is ready to be opened.
Start Libero on the same terminal in linux, like so:

```
libero &
```

or from the start menu in Windows, and open the project file by pressing `CTRL+O` and selecting the generated project as `gateware/work/libero/BVF_GATEWARE_025T.prjx` .

5.11.3 Exploring The Design

Let the IDE load everything, and then you're all set to browse around! You can go to the `Design Hierarchy` view to look at all Smart Design components. Here, all the gateware components are listed in block-like views. Double click the `DEFAULT_******` option in the hierarchy to have a look at the whole gateware. You should also be able to see the cape, M.2 interface and the RISC-V subsystem modules. These modules are explained in *Gateware Introduction*.

5.11.4 Adding Custom HDL

Once you're done exploring, we can start by adding our first HDL to the design.
Create a new HDL file through the menu bar, and name it `blinky`.
Once created, you can find the HDL file under the `User HDL Source Files`
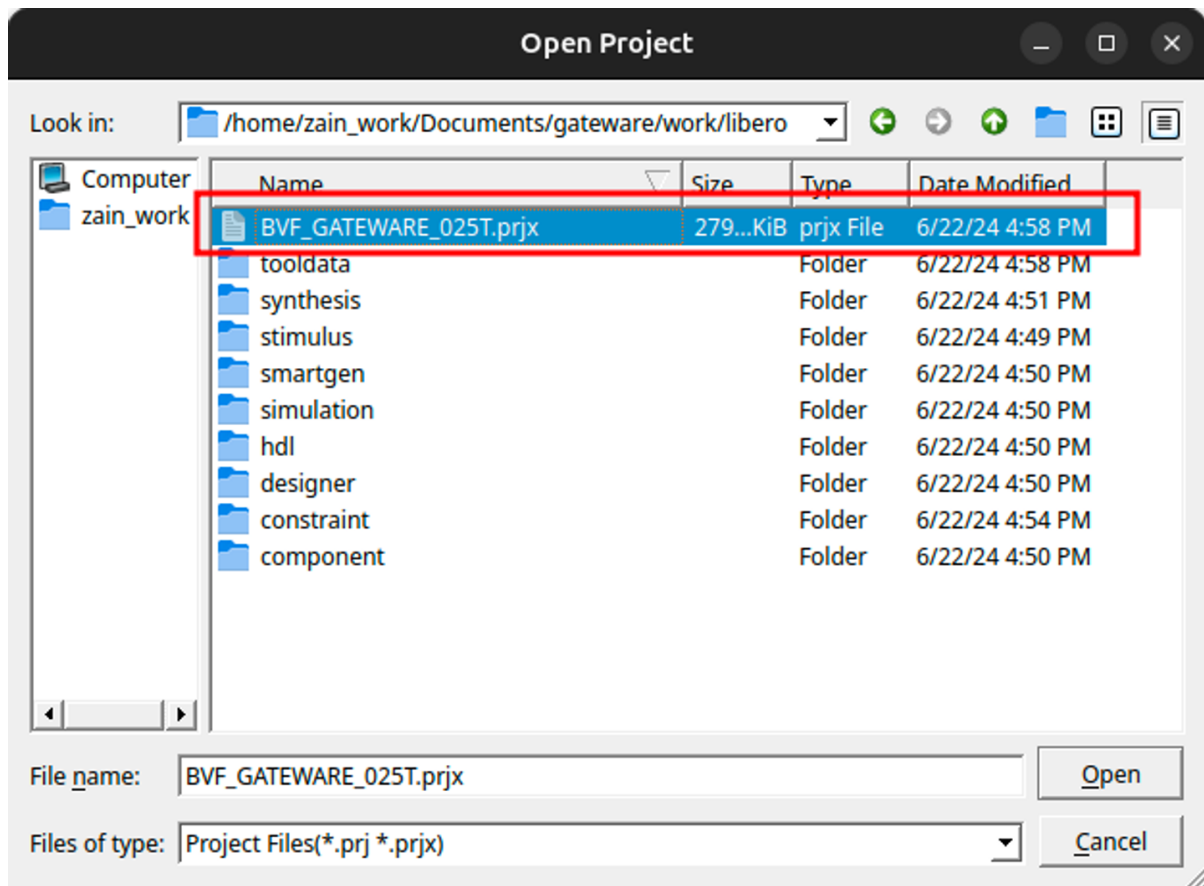heading in the Design Hierarchy.

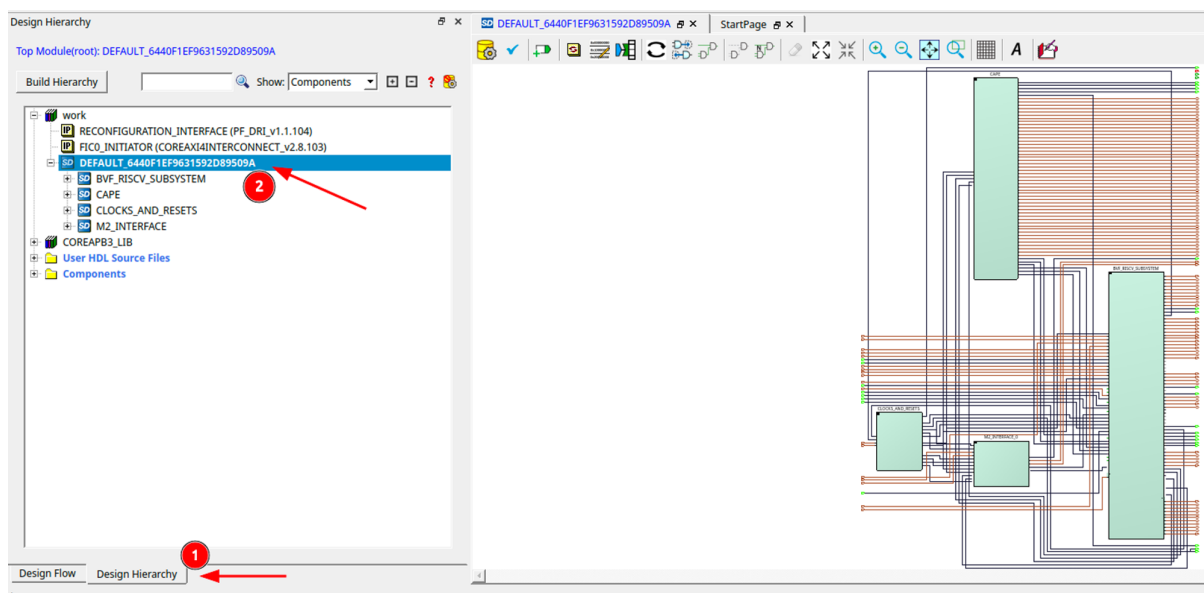Next, add this code to the file:

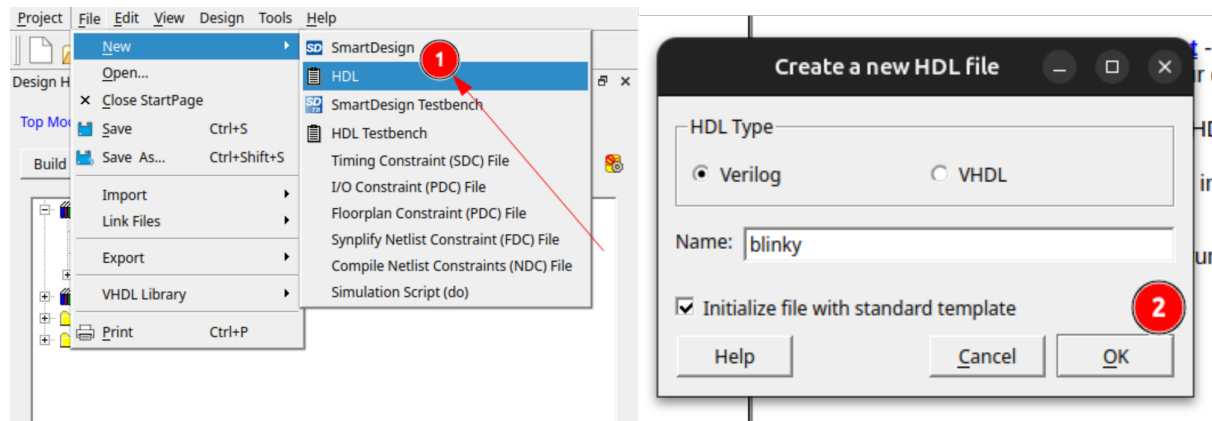Fig. 5.15: Libero project location

Fig. 5.16: Libero gateware overview

Fig. 5.17: Adding new HDL

```verilog
`timescale 1ns/100ps
module blinky(
input    clk,
input    resetn,
input [27:0] gpio_out,
input [27:0] gpio_enable,
output [27:0] modified_gpio,
output [27:0] modified_gpio_enable
);

reg [22:0] counter;
assign modified_gpio = {gpio_out[27:6], counter[22], gpio_out[4:0]};
assign modified_gpio_enable = {gpio_enable[27:6], 1'b1, gpio_enable[4:0]};

always@(posedge clk or negedge resetn)
begin
    if(~resetn)
        begin
            counter <= 23'h0;
        end
    else
        begin
            counter <= counter + 23'b1;
        end
end
endmodule
```

After saving it, press the `Build Hierarchy` button in the Design Hierarchy window to refresh it,
and bring the added HDL to the work directory.
Right click on it to select the "Create Core from HDL…." option.
Press `No` on the dialog that follows since we've described the ports completely in our HDL.

Now, double click the `CAPE` design under the `DEFAULT_****` smart design,
to have a look at what's in the cape.

Drag and drop the `blinky` file appearing in the work section into the cape design.
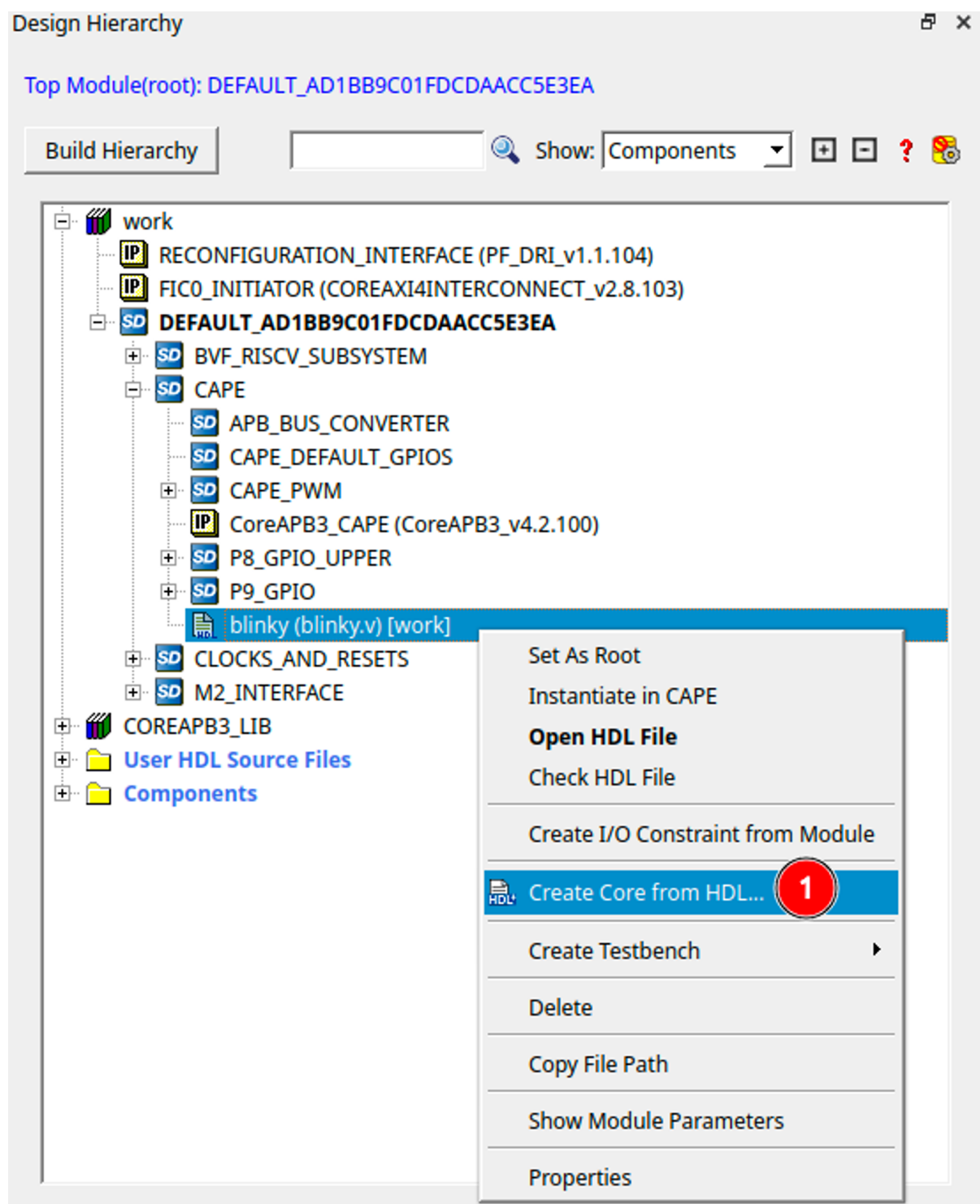
Fig. 5.18: Create core from HDL

You will have successfully instantiated the new verilog file into the cape smart design.
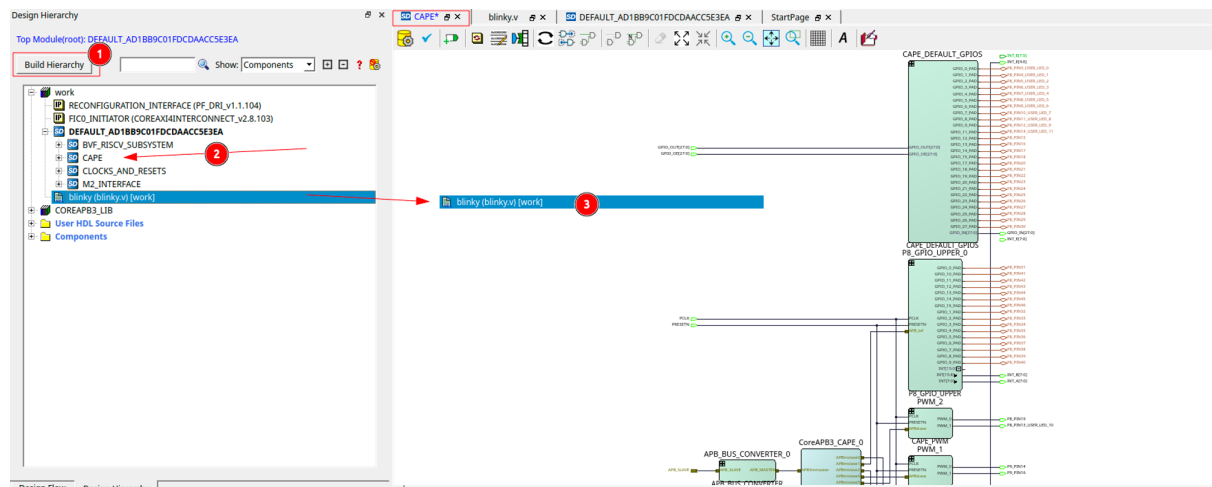

Fig. 5.19: Add blinky to cape

Making The Connections

You should see the blinky module within the CAPE design,
and it should be fairly obvious where we're going to be connecting
the module if you've gone through the previous demo.

First, delete the wires connecting the `GPIO_OUT` and `GPIO_OE` to the `CAPE_DEFAULT_GPIOS` module.
Then, simply connect the `GPIO_OUT` and the `GPIO_OE` terminals of the cape
to the `gpio_out` and the `gpio_enable` pins respectively.
Similarly connect the outputs of the blinky module to the `CAPE_DEFAULT_GPIOS` module.
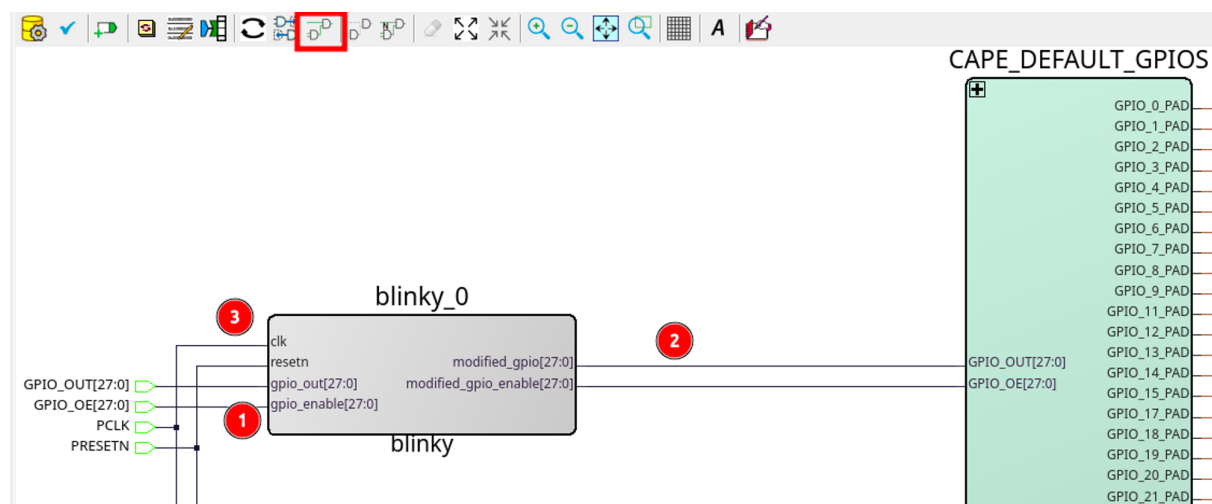

Fig. 5.20: Connect blinky to cape

Finally, connect the CLK and the RESET pins to the `PCLK` and the `PRESETN` pins below in the cape. You can use the **compress layout** button in the toolbar to make the design neat once you're done connecting the wires.

Go ahead and save the CAPE file.

You can also verify the design by pressing the checkmark icon in the editor toolbar.

Now, it's time to export our design back to the gateware repository.

5.11.5 Exporting The Design

Exporting the Cape

The SmartDesigns you have changed should show an "i" icon in front of them indicating that they need to be regenerated.

First, regenerate the designs by right clicking on them and selecting "Generate Component".

Rebuild the Hierarchy too as we've done before.

Next, right-click on the cape and select "Export Component Description (TCL)" to export it as a script which can be used in the gateware repository.

I suggest creating an export directory where you can temporarily store the exported gateware files before getting them into the repository.

Important: You **must** make sure your path exist, because Libero does not currently tell you if the export is successfull or not.

Now, simply copy it into the gateware at the following path.

```
cp ~/export/gateware/CAPE.tcl ~/gateware/sources/FPGA-design/script_support/
↪components/CAPE/DEFAULT/
```

Exporting The HDL

To add new HDL to the gateware repository, first we need to copy it to the HDL directory at *gateware/sources/FPGA-design/script_support/HDL*. You can do that by just creating a folder named blinky inside and copying the HDL to it.

```
mkdir ~/gateware/sources/FPGA-design/script_support/HDL/BLINKY
cp ~/gateware/work/libero/hdl/blinky.v ~/gateware/sources/FPGA-design/script_
↪support/HDL/BLINKY/
```

Now, to add the TCL script to import this design for the CAPE scripts, we can export the script by right-clicking on the HDL file in the Design Hierarchy and select `Export Component Description`.
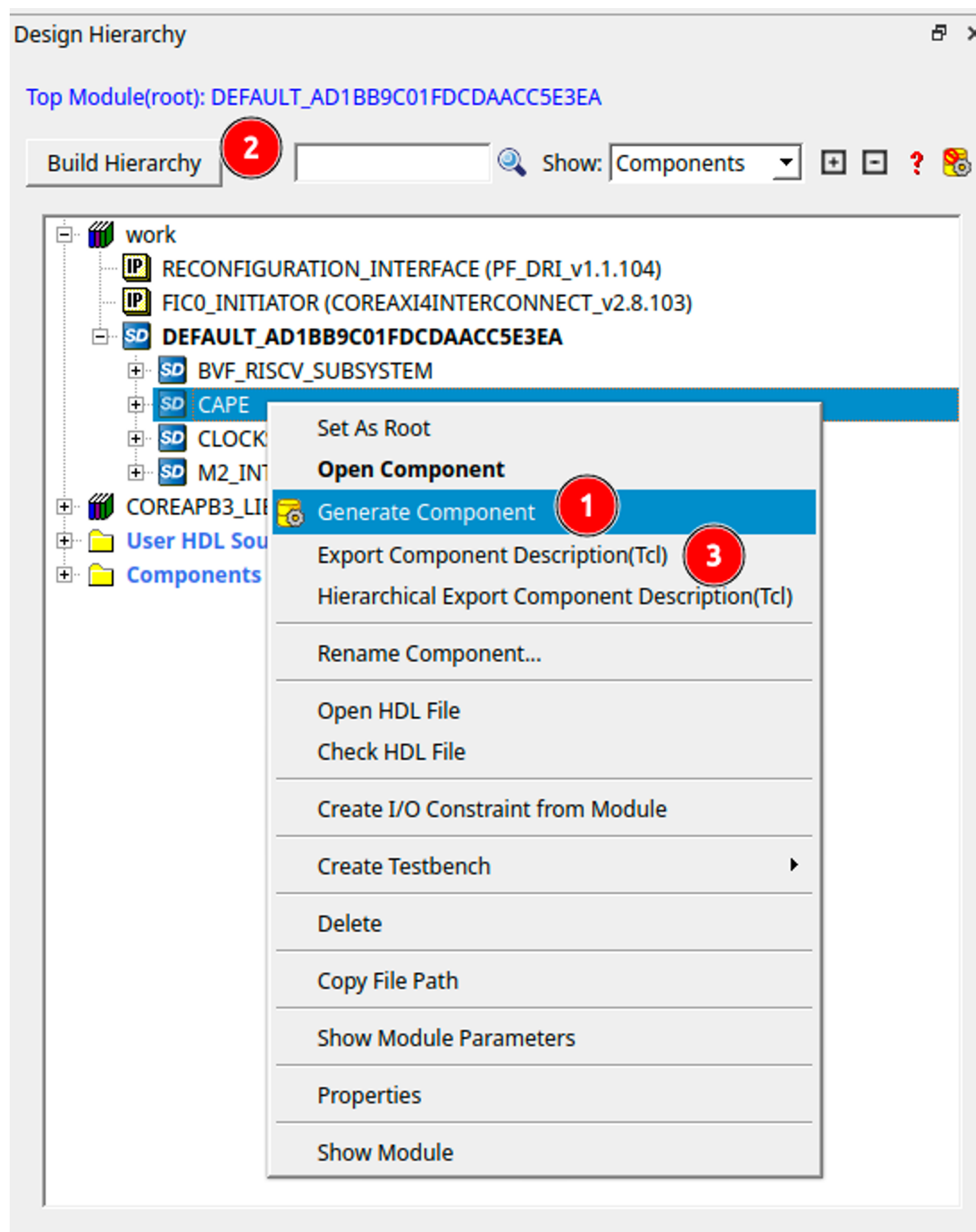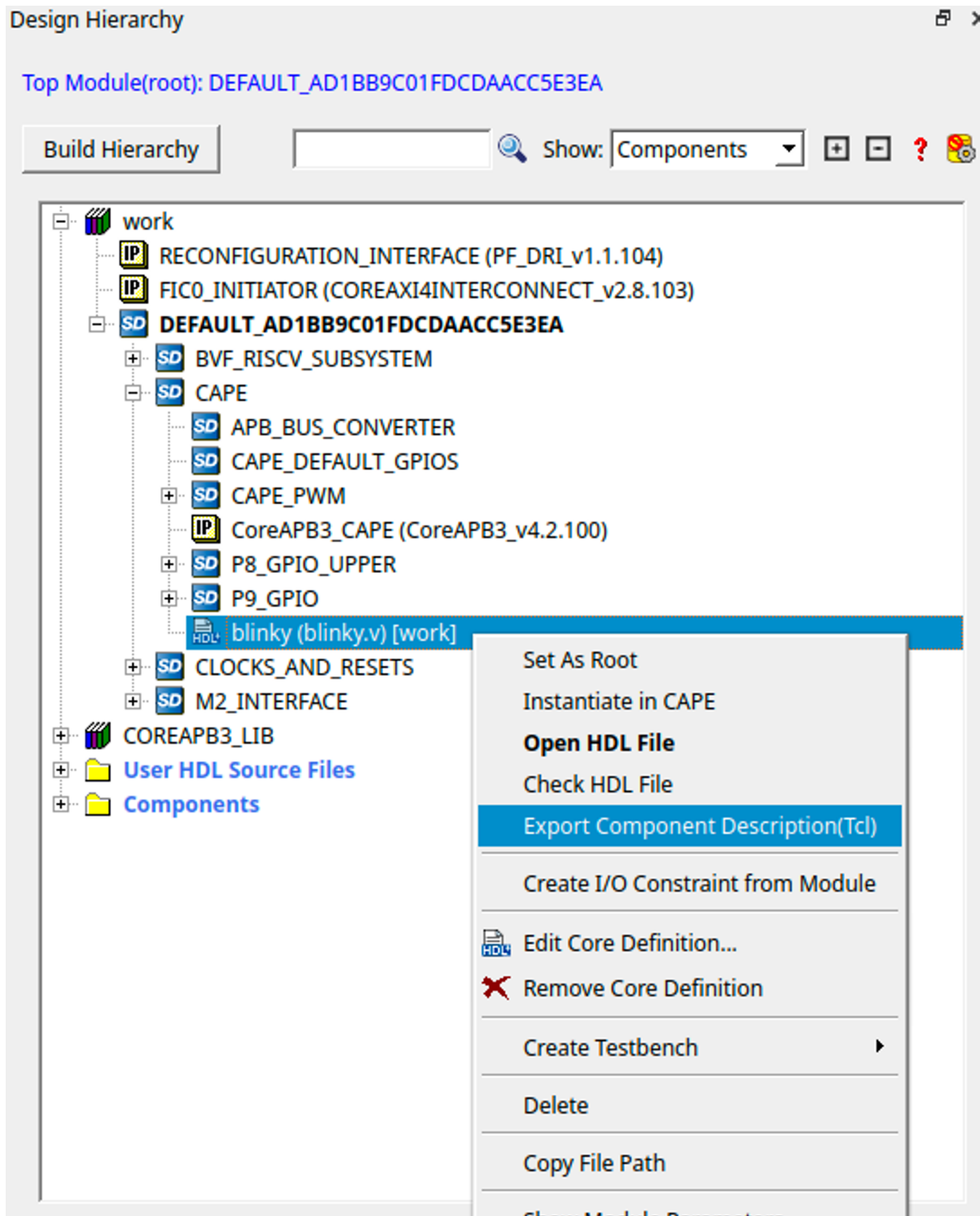
Fig. 5.21: Regenerate designs

Fig. 5.22: Export HDL

Now, concatenate the contents of this exported file to our gateware's HDL sourcing script at *gateware/sources/FPGA-design/script_support/hdl_source.tcl* like so:

```
cat blinky.tcl >> ~/gateware/sources/FPGA-design/script_support/hdl_source.
↪tcl
```

First, copy the contents of the exported TCL file to the bottom of the file.

Replace the `-file` argument in the line with `-file $project_dir/hdl/blinky.v`.

Finally, source the file by add a line below line no. 11 as:

```
    -hdl_source {script_support/HDL/AXI4_address_shim/AXI4_address_shim.v} \
    -hdl_source {script_support/HDL/BLINKY/blinky.v} # ⬅ Source the script␣
↪below line 11

#......
#...... towards the end of the file

hdl_core_assign_bif_signal -hdl_core_name {AXI_ADDRESS_SHIM} -bif_name {AXI4_
↪INITIATOR} -bif_signal_name {RREADY} -core_signal_name {INITIATOR_OUT_
↪RREADY}

create_hdl_core -file $project_dir/hdl/blinky.v -module {blinky} -library
↪{work} -package {}
# ⬅ Add the core at the end of the file
```

Feel free to cut any extra comment lines introduced when concatenating above. Verify your script as above, save it and now you're good to compile your project!

Important:

Make sure you close Libero at this point.

If you don't, `build-bitstream.py` **will** fail to properly checkout the required licenses.

Now is a good time to check in your changes to git:

```
cd ~/gateware
git add ./sources/FPGA-design/script_support/components/CAPE/DEFAULT/CAPE.tcl
git add ./sources/FPGA-design/script_support/hdl_source.tcl
git add ./sources/FPGA-design/script_support/HDL/BLINKY/blinky.v
git clean -df
```

5.11.6 Final Verification

Go ahead and run the python script to build the gateware and verify your changes:

```
python build-bitstream.py ./build-options/default.yaml
```

If at any point the compilation fails, you can debug the script at the mentioned line.

If it compiles successfully, it will mention it by saying:

```
The Execute Script command succeeded.
The BVF_GATEWARE_025T project was closed.
```

With a little luck, the script completes successfully and you can

now send your changes onto your gateware repository fork,

download the artifacts after compilation,

and program the gateware using the `change_gateware.sh` script.

Tip: For a more direct route you can copy the generated bitstream straight to your Beagle and try the result immidiately:

```
scp -r ./bitstream beagle@<ip or name here>:
```

On the beagle, use:

```
sudo /usr/share/beagleboard/gateware/change-gateware.sh ./bitstream
```

Have fun!

5.12 Simulating Gateware Design with Libero

In this demonstration, we will have a look at simulating the gateware design in Libero. Through simulations, one can verify the functionality of the design before implementing it on the hardware. The simulation is done using the ModelSim simulator, which is integrated with Libero. The gateware design that we will simulate is the blinky LED design, present in the `VERILOG_TUTORIAL` gateware option.

5.12.1 Prerequisites

1. Libero SoC 2022 or later. You can follow this guide to install Libero SoC: *Microchip FPGA Tools Installation Guide*.

2. A copy of the gateware repository.

3. The `setup_microchip_tools.sh` file, to start the license server and set the environment variables. The setup of this script is also covered in the installation guide.

5.12.2 Setting up ModelSim

Modelsim requires certain libraries to be present in the system, which might not be installed by default. You can check if ModelSim is working by:

```
source setup_microchip_tools.sh
$LIBERO_INSTALL_DIR/ModelSimPro/linuxacoem/vsim
```

If ModelSim is not working due to missing libraries, you might see an error message like:

```
vsim: error while loading shared libraries: libXft.so.2: cannot open shared
↪object file: No such file or directory
```

To fix this, you can install the required libraries using the following command:

```
sudo  apt-get install libxft2 libxft2:i386 lib32ncurses5
```

If you still cannot run ModelSim, you can try fixes from this guide.

5.12.3 Simulating the Blinky LED Design

To start with the simulation, we must first compile the gateware design. The blinky LED design is present in the `VERILOG_TUTORIAL` gateware option. To compile the design, follow the steps below:

1. First, compile the gateware design using the following command:

```
cd gateware
python build-bitstream.py custom-fpga-design/my_custom_fpga_design.yaml
```

1. Once the design is compiled, open the Libero SoC software and select the created project. You can find the project in the `work/libero` directory.

2. Once the project is opened, your window should look something like this. In front of you will be the overview of the gateware design, and on the left you will have multiple tabs showing the design hierarchy, design flow, etc.
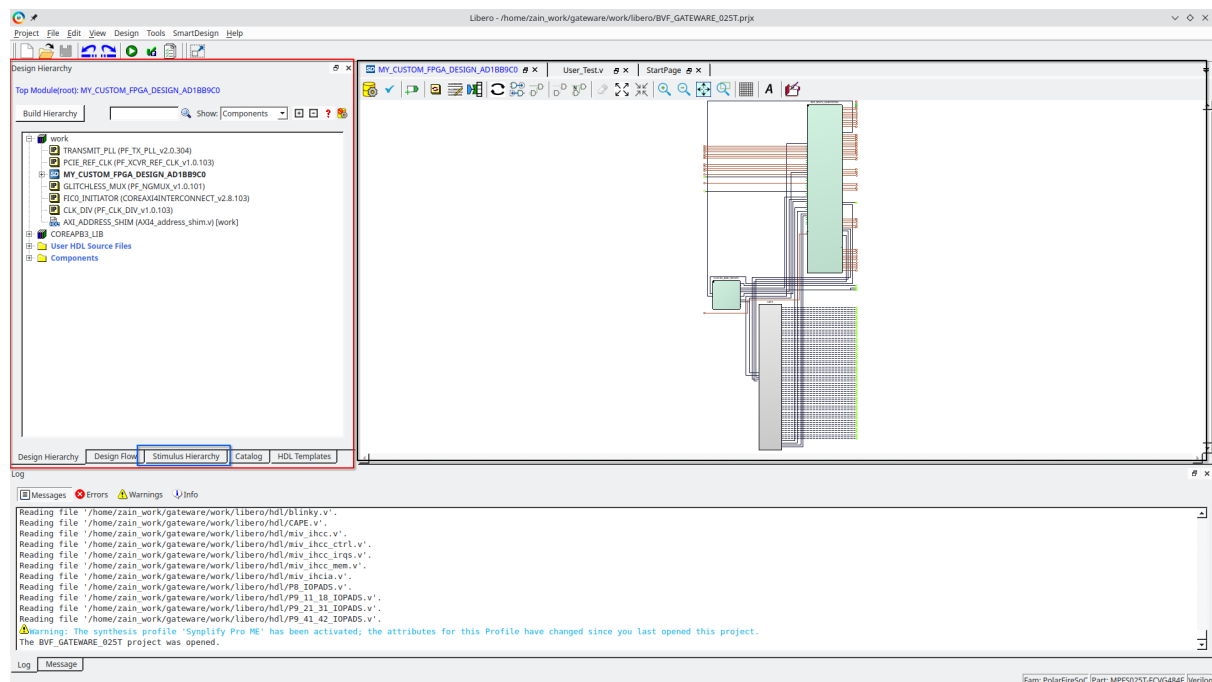

1. For the simulation, testbench files will need to be set up, which will be used to simulate the design. This can be done from the stimulus hierarchy. However, for this example we will be using the default testbench files provided by libero.

2. To set up the simulation, go to the `Design Flow` tab and right click on the `Simulate` button to select `Open Interactively`.

1. Before starting modelsim, Libero will ask you to add any additional files that you want to include in the simulation. For now, let's go with the ones that came with the design and it's IPs.

1. Once the simulation is started, you will see the ModelSim window open up.

5.12.4 Exploring ModelSim and Running the simulations

Looking at the modelsim window, there are four main sections to look at:

1. The top left section shows the design hierarchy. This is where you can see the design modules and their instances.

2. The section beside the design hierarchy is the object hierarchy. This shows the objects in the design, including the signals and variables.
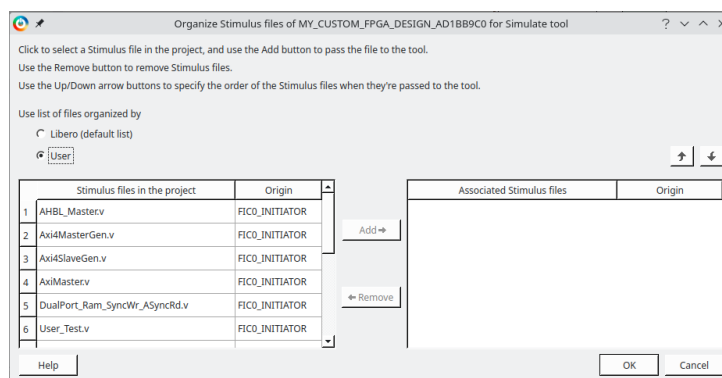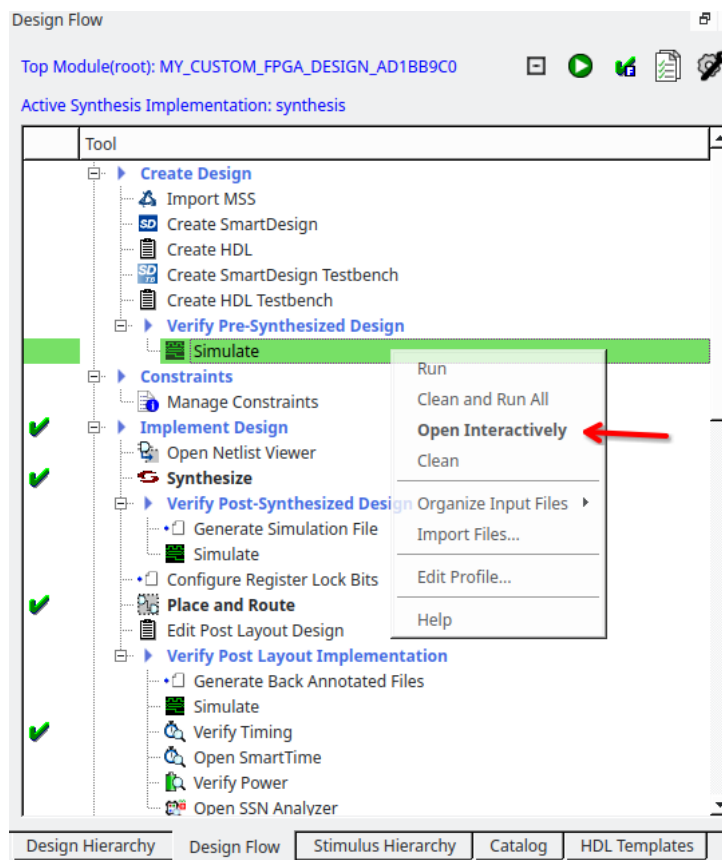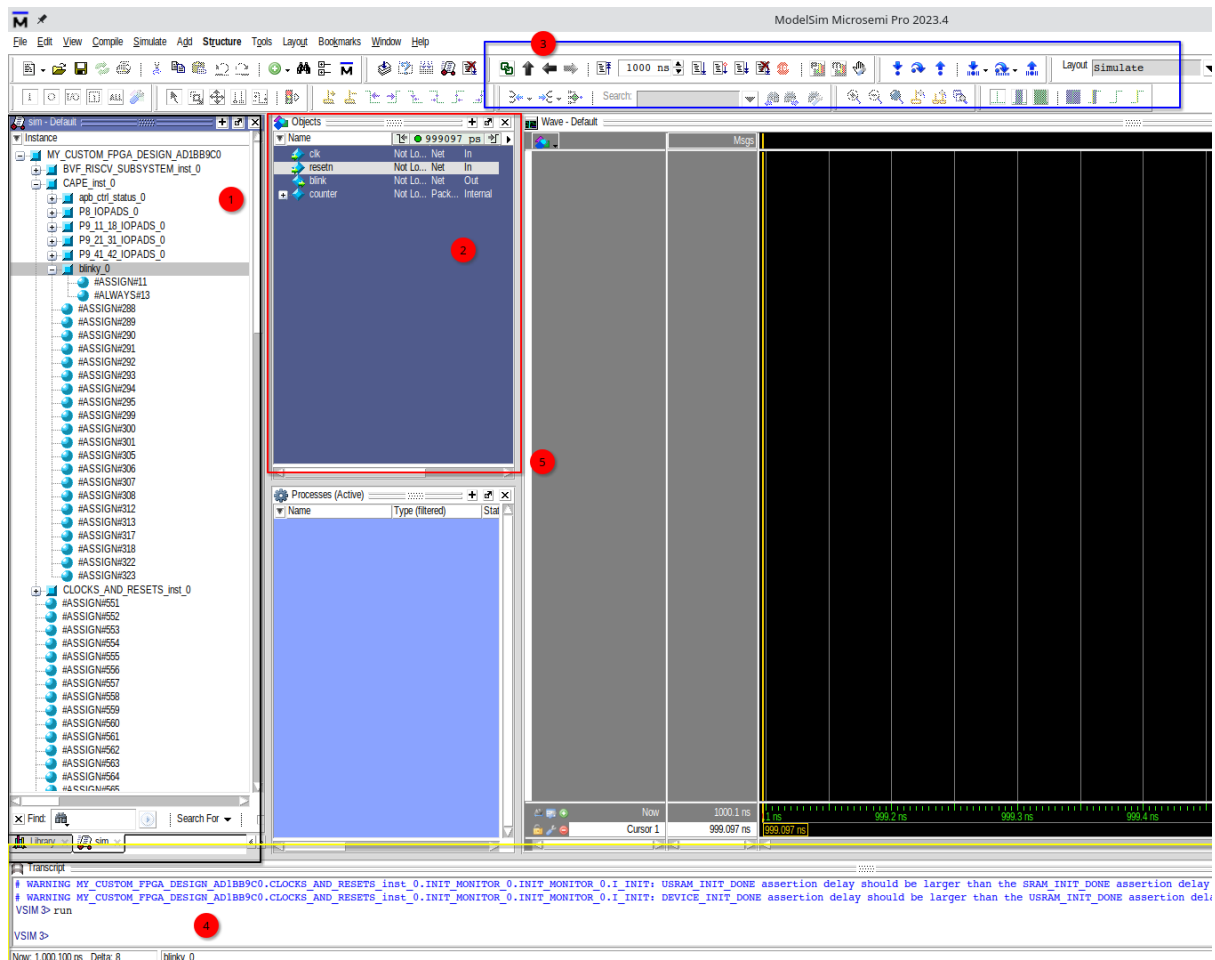
3. At the top, you should see the simulation toolbar. This is where you can run the simulations, add break-points, etc.

4. At the bottom, you should see the transcript window. This is where you can see the simulation logs. This also acts as a command line interface for ModelSim.

5. The far right section is the waveform window. This is where you can see the waveforms of the signals in the design.

You can add signals to the waveform window by right clicking on the signal in the object hierarchy and selecting `Add to Wave`. Once added, you can run the simulation by clicking on the `Run` button in the simulation toolbar. The simulation will run for a few nano seconds as specified in the toolbar beside the `Run` button.
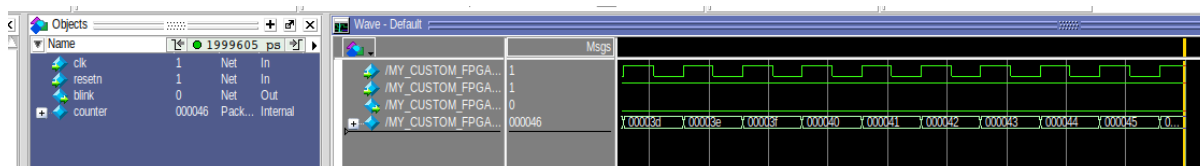
Once the simulation is complete, you can see the waveforms of the signals in the waveform window.

If you want to automate addition of signals to the waveform, you see the output of each GUI command in the transcript window. You can use these commands to automate the process. Just put the commands in a file with a `.do` extension and run the file using the `do` command in the transcript window.

An example of a `.do` file is shown below:

```
add wave -noupdate /tb_top/clk
add wave -noupdate /tb_top/rst
add wave -noupdate /tb_top/led

run 100 ns
```

A default `run.do` script is created at the following path - `work/libero/simulation` - when a simulation is run. You can use this file as a starter file for creating your own scripts as well as for understanding how the initial simulation is set-up.

Good luck with your simulations!

5.13 Comms Cape Gateware for BeagleV-Fire

The comms cape provides an array of communication protocols including

- one RS485,
- one CAN,
- two analog 4-20 mA current loops,
- two 3A 50V interfaces allowing the control of high current loads.

5.13.1 Cape schematics, layout, and mechanicals

For the schematics, layout, and mechanicals of the cape, please refer to industrial-comms-cape.

5.13.2 Usage

Firstly, the comms cape gateware must be compiled and updated on the beagleV-Fire. This can be done by using the *build-bitstream.py* script in the gateware repository with the *cape_comms.yaml* build option file.

```
python3 build-bitstream.py build-options/cape_comms.yaml
```

CAN

RS485

The RS485 interface is connected to UART4 on the BeagleV-Fire. It can be accessed using /dev/bone/ uart/4 in Linux.

Sink drivers

The sink drivers are connected to the P9_15 and P9_23 GPIOs. They can be controlled by writing to the GPIOs by:

```
echo 425 > /sys/class/gpio/export
echo 431 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio425/direction
echo out > /sys/class/gpio/gpio431/direction
echo 1 > /sys/class/gpio/gpio425/value
echo 1 > /sys/class/gpio/gpio431/value
```

Current loops

The current loops are connected to the ADC inputs of the BeagleV-Fire at pins P9_35 and P9_36. They can be accessed once QSPI is enabled in the comms cape device tree overlay. The current loops can be read by:

```
cat /sys/bus/iio/devices/iio:device0/in_voltage5_raw #Current Loop A
cat /sys/bus/iio/devices/iio:device0/in_voltage6_raw #Current Loop B
```

Voltage to current conversion will have to be done in software.

5.13.3 Pinout

The full pinout for the cape interface spec can be found here . You can also see the pinout below, refer to the last column for comms-cape specific pins.

P8 Header

Signal	Control	Irq #	Description
P8_1	n/a	n/a	GND
P8_2	n/a	n/a	GND
P8_3	MSS GPIO_2[0]	53	User LED 0
P8_4	MSS GPIO_2[1]	53	User LED 1
P8_5	MSS GPIO_2[2]	53	User LED 2
P8_6	MSS GPIO_2[3]	53	User LED 3
P8_7	MSS GPIO_2[4]	53	User LED 4
P8_8	MSS GPIO_2[5]	53	User LED 5
P8_9	MSS GPIO_2[6]	53	User LED 6
P8_10	MSS GPIO_2[7]	53	User LED 7
P8_11	MSS GPIO_2[8]	53	User LED 8
P8_12	MSS GPIO_2[9]	53	User LED 9

Table 5.1 – continued from previous page

Signal	Control	Irq #	Description
P8_13	core_pwm[1] @ 0x41500000	n/a	PWM_2:1
P8_14	MSS GPIO_2[11]	53	User LED 11
P8_15	MSS GPIO_2[12]	53	GPIO
P8_16	MSS GPIO_2[13]	53	GPIO
P8_17	MSS GPIO_2[14]	53	GPIO
P8_18	MSS GPIO_2[15]	53	GPIO
P8_19	core_pwm[0] @ 0x41500000	n/a	PWM_2:0
P8_20	MSS GPIO_2[17]	53	GPIO
P8_21	MSS GPIO_2[18]	53	GPIO
P8_22	MSS GPIO_2[19]	53	GPIO
P8_23	MSS GPIO_2[20]	53	GPIO
P8_24	MSS GPIO_2[21]	53	GPIO
P8_25	MSS GPIO_2[22]	53	GPIO
P8_26	MSS GPIO_2[23]	53	GPIO
P8_27	MSS GPIO_2[24]	53	GPIO
P8_28	MSS GPIO_2[25]	53	GPIO
P8_29	MSS GPIO_2[26]	53	GPIO
P8_30	MSS GPIO_2[27]	53	GPIO
P8_31	core_gpio[0] @ 0x41100000	126	GPIO
P8_32	core_gpio[1] @ 0x41100000	127	GPIO
P8_33	core_gpio[2] @ 0x41100000	128	GPIO
P8_34	core_gpio[3] @ 0x41100000	129	GPIO
P8_35	core_gpio[4] @ 0x41100000	130	GPIO
P8_36	core_gpio[5] @ 0x41100000	131	GPIO
P8_37	core_gpio[6] @ 0x41100000	132	GPIO
P8_38	core_gpio[7] @ 0x41100000	133	GPIO
P8_39	core_gpio[8] @ 0x41100000	134	GPIO
P8_40	core_gpio[9] @ 0x41100000	135	GPIO
P8_41	core_gpio[10] @ 0x41100000	136	GPIO
P8_42	core_gpio[11] @ 0x41100000	137	GPIO
P8_43	core_gpio[12] @ 0x41100000	138	GPIO
P8_44	core_gpio[13] @ 0x41100000	139	GPIO
P8_45	core_gpio[14] @ 0x41100000	140	GPIO
P8_46	core_gpio[15] @ 0x41100000	141	GPIO

P9 Header

Signal	Control	Irq #	Description
P9_1	n/a	n/a	GND
P9_2	n/a	n/a	GND
P9_3	n/a	n/a	VCC 3.3V
P9_4	n/a	n/a	VCC 3.3V
P9_5	n/a	n/a	VDD 5V
P9_6	n/a	n/a	VDD 5V
P9_7	n/a	n/a	SYS 5V
P9_8	n/a	n/a	SYS 5V
P9_9	n/a	n/a	NC
P9_10	n/a	n/a	SYS_RSTN
P9_11	MMUART4	94	UART4 RX <— For RS485
P9_12	core_gpio[1] @ 0x41200000	143	GPIO
P9_13	MMUART4	94	UART4 TX <— For RS485
P9_14	core_pwm[0] @ 0x41400000	n/a	PWM_1:0

Table 5.2 – continued from previous page

Signal	Control	Irq #	Description
P9_15	core_gpio[4] @ 0x41200000	146	GPIO
P9_16	core_pwm[1] @ 0x41400000	n/a	PWM_1:1
P9_17	MSS SPI0	54	SPI0 CS
P9_18	MSS SPI0	54	SPI0 MOSI
P9_19	MSS I2C0	58	I2C0 SCL
P9_20	MSS I2C0	58	I2C0 SDA
P9_21	MSS SPI0	54	SPI0 MISO
P9_22	MSS SPI0	54	SPI0 SCLK
P9_23	core_gpio[10] @ 0x41200000	152	GPIO
P9_24	CAN_1_RXBUS	n/a	CAN RX <— For CAN
P9_25	CAN_1_TX_EBL	154	CAN TX EBL
P9_26	CAN_1_TXBUS	n/a	CAN TX
P9_27	core_gpio[14] @ 0x41200000	156	GPIO
P9_28	MSS SPI1	55	SPI1 CS
P9_29	MSS SPI1	55	SPI1 MISO
P9_30	core_gpio[17] @ 0x41200000	159	GPIO
P9_31	MSS SPI1	55	SPI1 SCLK
P9_32	n/a	n/a	VDD ADC
P9_33	n/a	n/a	ADC input 4
P9_34	n/a	n/a	AGND
P9_35	n/a	n/a	ADC input 6
P9_36	n/a	n/a	ADC input 5
P9_37	n/a	n/a	ADC input 2
P9_38	n/a	n/a	ADC input 3
P9_39	n/a	n/a	ADC input 0
P9_40	n/a	n/a	ADC input 1
P9_41	core_gpio[19] @ 0x41200000	161	GPIO
P9_42	core_pwm[0] @ 0x41000000	n/a	PWM_0:0
P9_43	n/a	n/a	GND
P9_44	n/a	n/a	GND
P9_45	n/a	n/a	GND
P9_46	n/a	n/a	GND

5.14 Accessing APB and AXI Peripherals Through Linux

5.14.1 AXI

AXI is part of the ARM AMBA (Advanced Microcontroller Bus Architecture) protocol family.

It is designed for high-performance, high-frequency system-on-chip (SoC) designs.

AXI provides high-speed data transfer with minimal latency and is widely used in various applications, including high-end embedded systems and complex digital circuits.

5.14.2 APB

APB is also part of the ARM AMBA protocol family, designed for low-power and low-latency communication with peripheral devices.

It is simpler and lower performance compared to AXI, making it suitable for slower peripheral devices. An APB peripheral also consumes less resources on the FPGA fabric compared to an AXI peripheral.

5.14.3 Accessing AXI and APB Peripherals from Linux

To access AXI and APB peripherals from Linux, memory-mapped I/O (MMIO) is commonly used.

This involves mapping the physical addresses of the peripherals into the virtual address space of a user-space application.

The following sections demonstrate how to access APB peripherals using the Linux `/dev/mem` interface and AXI peripherals using the UIO (Userspace I/O) framework.

Note: The codes for accessing the interfaces are available in the snippets here: APB Interfaces and AXI Interfaces

---

APB Interfaces

The MSS includes fabric interfaces for interfacing FPGA fabric with the CPU Core Complex.

It provides one 32-bit APB master interface, FIC3, and can be connected to a slave in the fabric.

Design Details   For this example, you can try to write to the APB slave present in the Verilog Tutorial Cape gateware. Select the gateware by changing *custom-fpga-design/my_custom_fpga_design.yaml* to include `VERILOG_TUTORIAL` as the cape option.

The APB Slave has two registers, one read-only register at `0x00`, one read-write register at `0x10`, and a status register containing the last read value at `0x20`.

Having a look at the design, we can see that the APB slave is connected with a CoreAPB3 interconnect, which assigns it the `0xXX10_0000` address, the top two bits being ignored.

Tracing to the master connected with the CoreAPB3 device, we can see that another interconnect is present, which gives our slave the `0xX100_0000` address.

The polarfire technical manual shows that FIC3 peripherals can start from the `0x4000_0000` address.

Therefore, the final address of our APB slave becomes `0x4110_0000`.

Now, we shall access this address through a memory-mapped interface in Linux.

Important:

The following paragraphs will present to you several ways to test APB/AXI traffic.

Normally, this isn't harmful, but reading/writing to addresses

with no gateware behind it **will** lead to you stalling a CPU.

In rare cases, this stalling of a CPU *can* lead to loss of content on your eMMC,

so please make sure you have a known good backup!

For more information about the stalls, please read the section *on issues faced with the interfaces*.

Accessing the Interface   There are two ways to access such registers. One can use the *devmem2* utility or write a C program for accessing the memory region. The first method is quite simple.

1. To read from a register:

```
sudo devmem2 0x41100000 w
```

2. To write to a register:

```
sudo devmem2 0x41100010 w 0x1
```

In the second method, we can use the `/dev/mem` interface to access the registers inside the APB Slave. Here is an example C program which demonstrates this:

AXI Interfaces

The MSS includes three 64-bit AXI FICs out of which FIC0 is used for data transfers to/from the fabric. FIC0 is connected as both master and slave. For usage of AXI peripherals, an example is also provided by microchip in their Polarfire SoC Linux examples. The example here takes reference from the AXI LSRAM example.

Design Details

A simple design can be created by first connecting the FIC0 Initiator from the MSS to a CoreAXI4Interconnect. Now, you can connect an AXI slave to this interconnect. We will be using the Polarfire AXI LSRAM.

Both the CoreAXI4Interconnect and the PF AXI LSRAM will have to be configured.
The AXI ID Width of both the modules will have to be matched, as well as the address space of the only slave will have to be configured.
In this example, LSRAM gets an address of *0x6000_0000* to *0x6000_ffff*, and the AWID is kept at 9 bits.
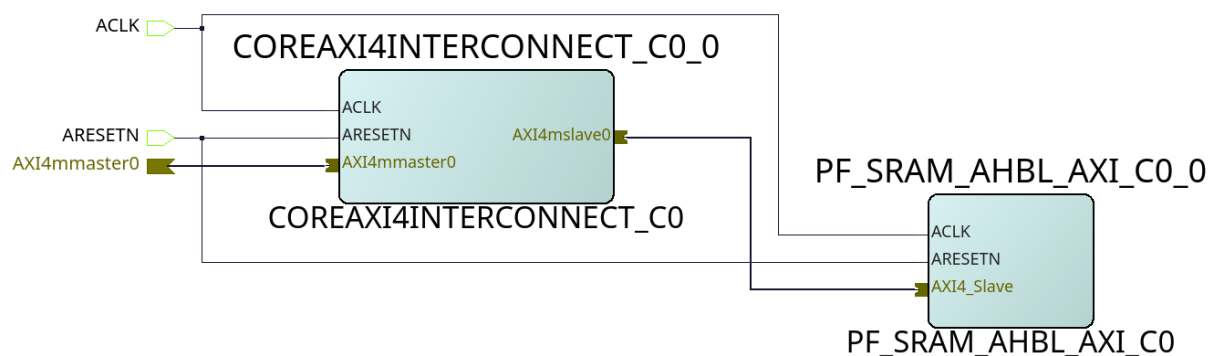

Fig. 5.23: AXI LSRAM slave (example design)

Finally, an entry will be added to the device tree to make a UIO device point to our LSRAM's memory region.

```
&{/} {
    fabric-bus@40000000 {
        fpgalsram: uio@60000000 {
            compatible = "generic-uio";
            linux,uio-name = "fpga_lsram"; // mandatory for program. If↳
→changed, please update program as well.
            reg = <0x0 0x60000000 0x0 0x1000>;
            status = "enabled";
        };
    };
};
```

Once the gateware is compiled, we can access the memory-mapped interface by the same methods, and by the UIO device as well.

1. Using devmem2:

```
sudo devmem2 0x60000000 w # for read
sudo devmem2 0x60000000 w 0x1 # for write
```

1. Using the UIO device:

Issues that can be faced when using an improperly configured AXI/APB interface

A CPU stall can be faced when accessing the FIC interfaces without any slaves connected to the memory region being accessed. Your BVF will stop responding if connected to SSH, and on serial you will see the following kernel messages:

```
[   24.110099] rcu: INFO: rcu_sched detected stalls on CPUs/tasks:
[   24.116041] rcu:     0-...0: (1 GPs behind) idle=e00c/0/0x1 softirq=40/41␣
→fqs=2626
[   24.123377]    (detected by 3, t=5255 jiffies, g=-1131, q=9 ncpus=4)
[   24.129573] Task dump for CPU 0:
[   24.132810] task:swapper/0      state:R  running task     stack:0    ␣
→pid:0     ppid:0      flags:0x00000008
[   24.142757] Call Trace:
[   24.145213] [<ffffffff80a67ba0>] __schedule+0x27c/0x834
```

If this happens, please double check your design. Specifically, check the address configured for the slaves, the AXI ID wire width and other AXI parameters.

In any case, this state is virtually impossible to recover from gracefully, so the **reset** button may be your last resort.

5.15 Building Linux for BeagleV-Fire using Buildroot

5.15.1 Introduction

Buildroot is a simple, efficient, and user-friendly tool for creating custom embedded Linux systems through cross-compilation.

This document provides a guide for building and flashing a Linux image on the BeagleV-Fire board with Buildroot. It outlines the process for compiling the image, writing image to the eMMC, and booting the new operating system.

Hardware requirements:

1. BeagleV-Fire board

2. USB-C cable

3. 3.3v USB to UART bridge

Connect BeagleV-Fire UART debug port using 3.3v USB to UART bridge.
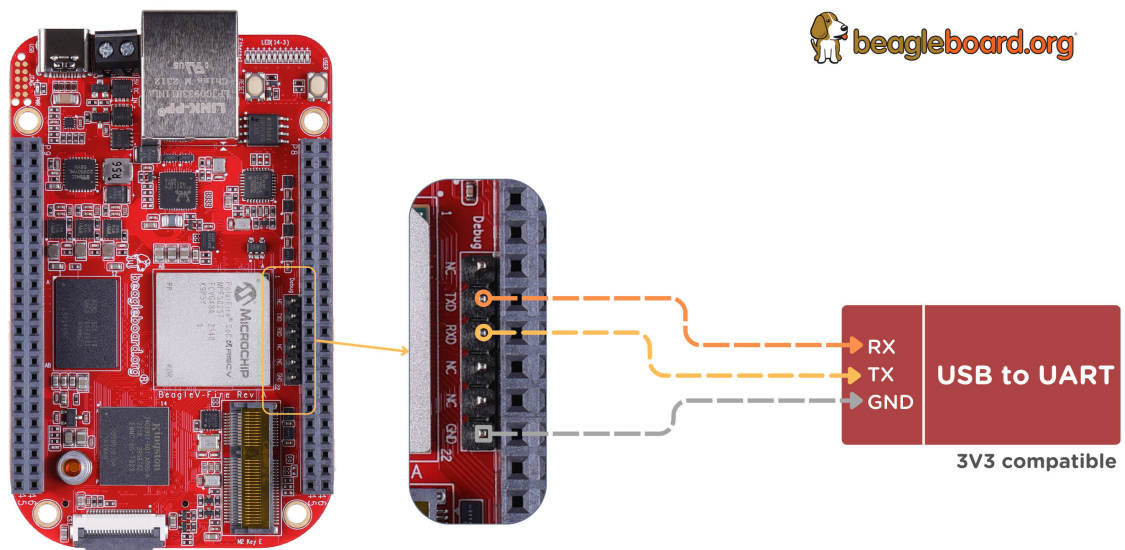
Software requirements:

Download Buildroot repository from GitHub.

5.15.2 Start Building

Note: The following steps are intended for a Linux operating system.

To build and flash Linux image using Buildroot,

Step 1. Navigate to the Buildroot directory

```
cd buildroot
```

Step 2. Configure the Build for BeagleV-Fire

Configure the build by selecting the default board configuration:

```
make beaglev_fire_defconfig
```

Step 3. Customise the build (Optional)

If you need to customize the build, use the following command:

```
make menuconfig
```

Step 4: Start the build process

```
make
```

Note: The build process can take 20-30 minutes for a clean build.

Step 5. Locate the build image

Once the build is complete, the Linux image will be saved as *sdcard.img* in the */output/images/* directory. The directory structure will look something like this:

```
$ ls output/images/
boot.scr          boot.vfat.bmap   dts/               Image.gz         mpfs_
↪icicle/     mpfs_icicle.its  rootfs.cpio       rootfs.tar        sdcard.img ↵
↪     sdcard.img.gz
```

(continues on next page)

```
boot.vfat         boot.vfat.gz       Image        microchip/       mpfs_
→icicle.itb  payload.bin     rootfs.cpio.gz   sdcard.bmap      sdcard.img.
→bmap   u-boot.bin
```

Step 6. Flash the Image to BeagleV-Fire's eMMC

- Restart the board and halt the HSS (Hart Software Services) by pressing any key

- In the HSS command line interface, type *usbdmsc* to expose the eMMC as a USB mass storage device using the USB-C connector.

- If successful, a message saying "USB Host connected" will be displayed

- Now, copy the image from local machine to BeagleV Fire's eMMC

```
sudo dd if=output/images/sdcard.img of=/dev/sdX bs=1M
```

Note: You need to replace /dev/sdX with the actual device name of your eMMC. Be very careful not to overwrite the wrong drive, as this action is irreversible.

- Once the transfer is complete, type *CTRL+C* to disconnect your device

- Finally boot the new Linux image by typing *boot* or reset your board

Detailed description of this step is mentioned in Flashing eMMC section.

Chapter 6

Support

All support for BeagleV Fire design is through BeagleBoard.org community at BeagleBoard.org forum.

6.1   Production board boot media

6.2   Certifications and export control

6.2.1   Export designations

- HS: 8471504090
- US HS: 8543708800
- EU HS: 8471707000

6.2.2   Size and weight

- Bare board dimensions: 86.38*54.61*18.8mm
- Bare board weight: 45.8g
- Full package dimensions: 140 x 100 x 40 mm
- Full package weight: 106g

6.3   Additional documentation

6.3.1   Hardware docs

For any hardware document like schematic diagram PDF, EDA files, issue tracker, and more you can checkout the BeagleV-Fire design repository.

6.3.2   Software docs

For BeagleV-Fire specific software projects you can checkout all the BeagleV-Fire project repositories group.

6.3.3 Support forum

For any additional support you can submit your queries on our forum, https://forum.beagleboard.org/tags/c/beaglev/15/fire

6.3.4 Pictures

6.4 Change History

Note: This section describes the change history of this document and board. Document changes are not always a result of a board change. A board change will always result in a document change.

6.4.1 Board Changes

For all changes, see https://git.beagleboard.org/beaglev-fire/beaglev-fire/. Versions released into production are noted below.

Table 6.1: BeagleV-Fire board change history

Rev	Changes	Date	By
A	Initial production version	2023-11-02	JK