JH7110 Ethernet Developing and Porting Guide

VisionFive 2

Version: 1.0

Date: 2022/12/30

Doc ID: JH7110-PGEN-001

Legal Statements

Important legal notice before reading this documentation.

PROPRIETARY NOTICE

Contact Us

Address: Room 502, Building 2, No. 61 Shengxia Rd., China (Shanghai) Pilot Free Trade Zone, Shanghai, 201203, China

Website: http://www.starfivetech.com

Email:

- Sales: sales@starfivetech.com
- Support: support@starfivetech.com

Preface

About this guide and technical support information.

About this document

This document mainly provides the SDK developers with the developing and porting instructions for the Ethernet module of the StarFive next generation SoC platform - JH7110.

Audience

This document mainly serves the Ethernet relevant driver developers. If you are developing and porting other modules, place a request to your sales or support consultant for our complete documentation set on JH7110.

Revision History

Table 0-1 Revision History

Version	Released	Revision
1.0		First official release.

Notes and notices

The following notes and notices might appear in this guide:

- **Tip:**
  Suggests how to apply the information in a topic or step.

- **Note:**
  Explains a special case or expands on an important point.

- **Important:**
  Points out critical information concerning a topic or step.

- **CAUTION:**
  Indicates that an action or step can cause loss of data, security problems, or performance issues.

- **Warning:**
  Indicates that an action or step can result in physical harm or cause damage to hardware.

Contents

List of Tables

List of Figures

1. Introduction

Like all other SoCs in the Linux operating system, U-Boot and Ethernet are the first two modules to develop applications and design porting strategies on.

This document primarily introduces the procedures of porting the JH7110 U-Boot and the YT8531 PHY to a new development board. You can use the information included as a reference for porting any other Ethernet PHY.

The code sources referenced in this document are based on the following conditions:

- SDK version: 3.0
- U-Boot version: 3.0
- Linux Kernel version: 5.15

Note:
For different U-Boot or Linux Kernel versions, these references may be slightly different, consult your StarFive sales consultant or technical support before the porting practices.

1.1. Device Tree Overview

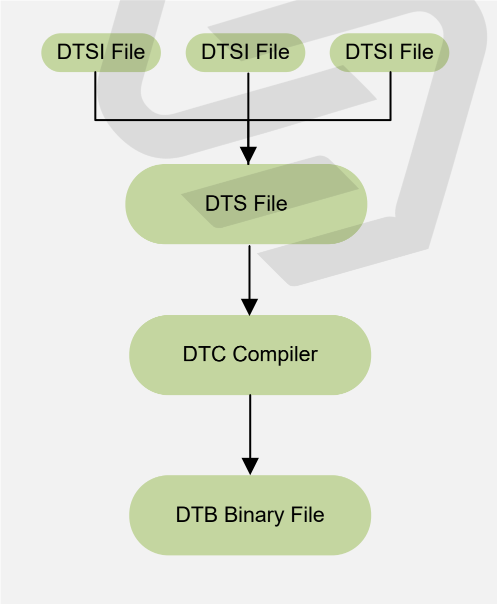
Since Linux 3.x, device tree is introduced as a data structure and language to describe hardware configuration. It is a system-readable description of hardware settings so that the operating system doesn't have to hard code details of the machine.

A device tree is primarily represented in the following forms.

- *Device Tree Compiler (DTC)*: The tool used to compile device tree into system-readable binaries.
- *Device Tree Source (DTS)*: The human-readable device tree description file. You can locate the target parameters and modify hardware configuration in this file.
- *Device Tree Source Information (DTSI)*: The human-readable header file which you can include in device tree description. You can locate the target parameters and modify hardware configuration in this file.
- *Device Tree Blob (DTB)*: The system-readable device tree binary blob files which is burned in system for execution.

The following diagram shows the relationship (workflow) of the above forms.

Figure 1-1 Device Tree Workflow



1.2. Device Tree Source Code

Overview Structure

The device tree source code of JH7110 is listed as follows:

```
linux
├── arch
|   ├── riscv
|   |   ├── boot
|   |   |   ├── dts
|   |   |   |   └── starfive
|   |   |   |       ├── codecs
|   |   |   |       |   ├── sf_pdm.dtsi
|   |   |   |       |   ├── sf_pwmdac.dtsi
|   |   |   |       |   ├── sf_spdif.dtsi
|   |   |   |       |   ├── sf_tdm.dtsi
|   |   |   |       |   └── sf_wm8960.dtsi
|   |   |   |       ├── evb-overlay
|   |   |   |       |   ├── jh7110-evb-overlay-can.dts
|   |   |   |       |   ├── jh7110-evb-overlay-rgb2hdmi.dts
|   |   |   |       |   ├── jh7110-evb-overlay-sdio.dts
|   |   |   |       |   ├── jh7110-evb-overlay-spi.dts
|   |   |   |       |   ├── jh7110-evb-overlay-uart4-emmc.dts
|   |   |   |       |   ├── jh7110-evb-overlay-uart5-pwm.dts
|   |   |   |       |   └── Makefile
|   |   |   |       ├── jh7110-clk.dtsi
|   |   |   |       ├── jh7110-common.dtsi
|   |   |   |       ├── jh7110.dtsi
|   |   |   |       ├── jh7110-evb-can-pdm-pwmdac.dts
|   |   |   |       ├── jh7110-evb.dts
|   |   |   |       ├── jh7110-evb.dtsi
|   |   |   |       ├── jh7110-evb-dvp-rgb2hdmi.dts
|   |   |   |       ├── jh7110-evb-pcie-i2s-sd.dts
|   |   |   |       ├── jh7110-evb-pinctrl.dtsi
|   |   |   |       ├── jh7110-evb-spi-uart2.dts
|   |   |   |       ├── jh7110-evb-uart1-rgb2hdmi.dts
|   |   |   |       ├── jh7110-evb-uart4-emmc-spdif.dts
|   |   |   |       ├── jh7110-evb-uart5-pwm-i2c-tdm.dts
|   |   |   |       ├── jh7110-fpga.dts
|   |   |   |       ├── jh7110-visionfive-v2.dts
|   |   |   |       ├── Makefile
|   |   |   |       └── vf2-overlay
|   |   |   |           ├── Makefile
|   |   |   |           └── vf2-overlay-uart3-i2c.dts
```

SoC Platform

The device tree source code of the JH7110 SoC platform is in the following path:

```
freelight-u-sdk/linux/arch/riscv/boot/dts/starfive/jh7110.dtsi
```

VisionFive 2

The device tree source code of the VisionFive 2 *Single Board Computer (SBC)* is in the following path:

```
freelight-u-sdk/linux/arch/riscv/boot/dts/starfive/jh7110-visionfive-v2.dts
-- freelight-u-sdk/linux/arch/riscv/boot/dts/starfive/jh7110-common.dtsi
-- freelight-u-sdk/linux/arch/riscv/boot/dts/starfive/jh7110.dtsi
```

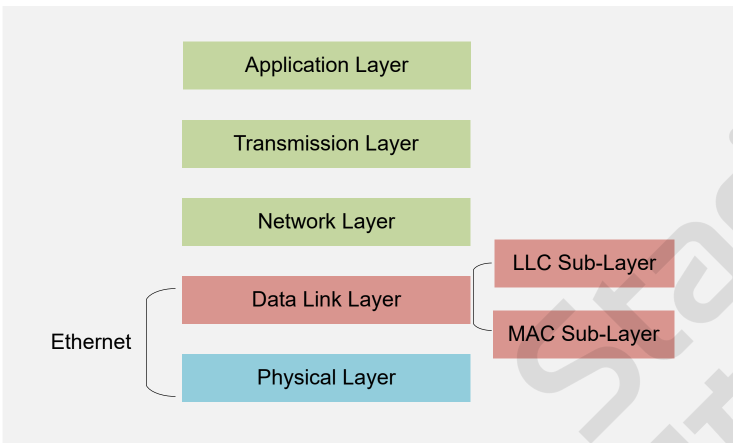
2. Ethernet Introduction

This chapter introduces how to configure an existing Ethernet driver.

2.1. About Ethernet

Ethernet is a *Local Area Network (LAN)*-based network communication technology. Ethernet follows the IEEE802.3 protocol standards, and includes the Ethernet speed ranges of 10 M, 100 M and 1,000 M. In the TCP/IP protocols, Ethernet is located in the following layers.

Figure 2-1 Ethernet Relevant Layers



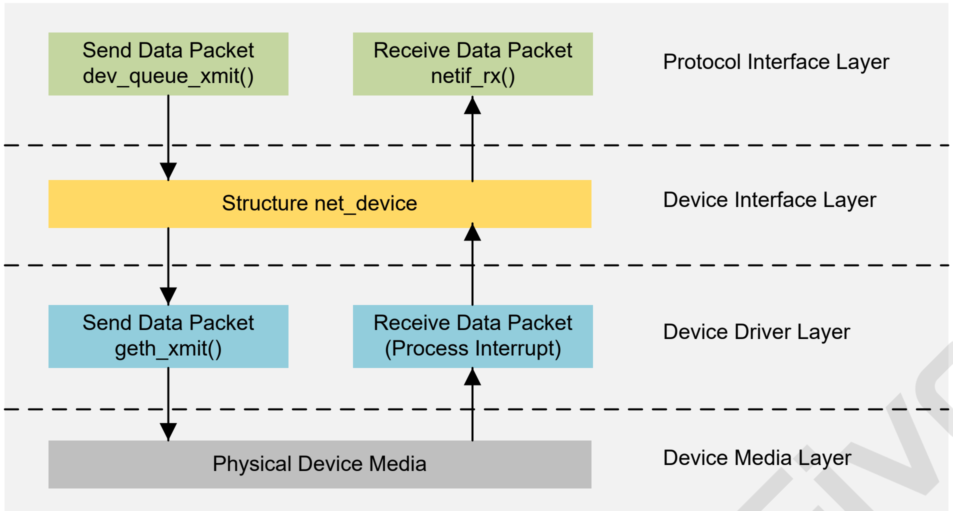
Ethernet is relevant to the physical layer (L1) and the data link layer (L2) in the TCP/IP layers. The data link layer contains the Logic Link Control (LLC) sub-layer and the Multimedia Access Control (MAC) sub-layer.

2.2. Ethernet Device Framework

The following diagram shows the network device framework in the Linux kernel.

The framework has the following layers.

Figure 2-2 Ethernet Device Framework



- **Protocol Interface Layer**: The layer provides unified data send and receive interfaces. The interface `dev_queue_xmit()` is used for sending data and `netif_rx()` is used for receiving data.
- **Device Interface Layer**: The layer provides the unified structure of net_device which is used to describe network device attributes and operation details. The structure works as a container for all functions in the device driver layer.
- **Device Driver Layer**: The layer realizes the functional operation pointers of defined in the structure of **net_device**. And then the operations are handed over to hardware drivers for execution.
- **Device Media Layer**: The layer contains as the physical elements which completes the data packet sending and receiving tasks, including the network transmission adapter and the media used for transmission.

2.3. GMAC Source Code Structure

The source code of GMAC is located in the following path:

```
Drivers/net/ethernet/stmicro/stmmac
```

The following code block provides an example of the GMAC source code.

```
Drivers/net/ethernet/stmicro/stmmac

├── stmmac.h
├── dwmac-starfive-plat.c
├── stmmac_main.c
```

Table 2-1 GMAC Source Code Structure

File	Explanation
`stmmac.h`	GMAC driver header file of the DWMAC platform. In this file, some macros, data structures and internal interfaces are defined.
`dwmac-starfive-plat.c`	GMAC driver specific configuration options of the StarFive DWMAC platform
`stmmac_main.c`	GMAC driver public interface on the DWMAC platform

2.4. Configuration

2.4.1. Kernel Menu Configuration

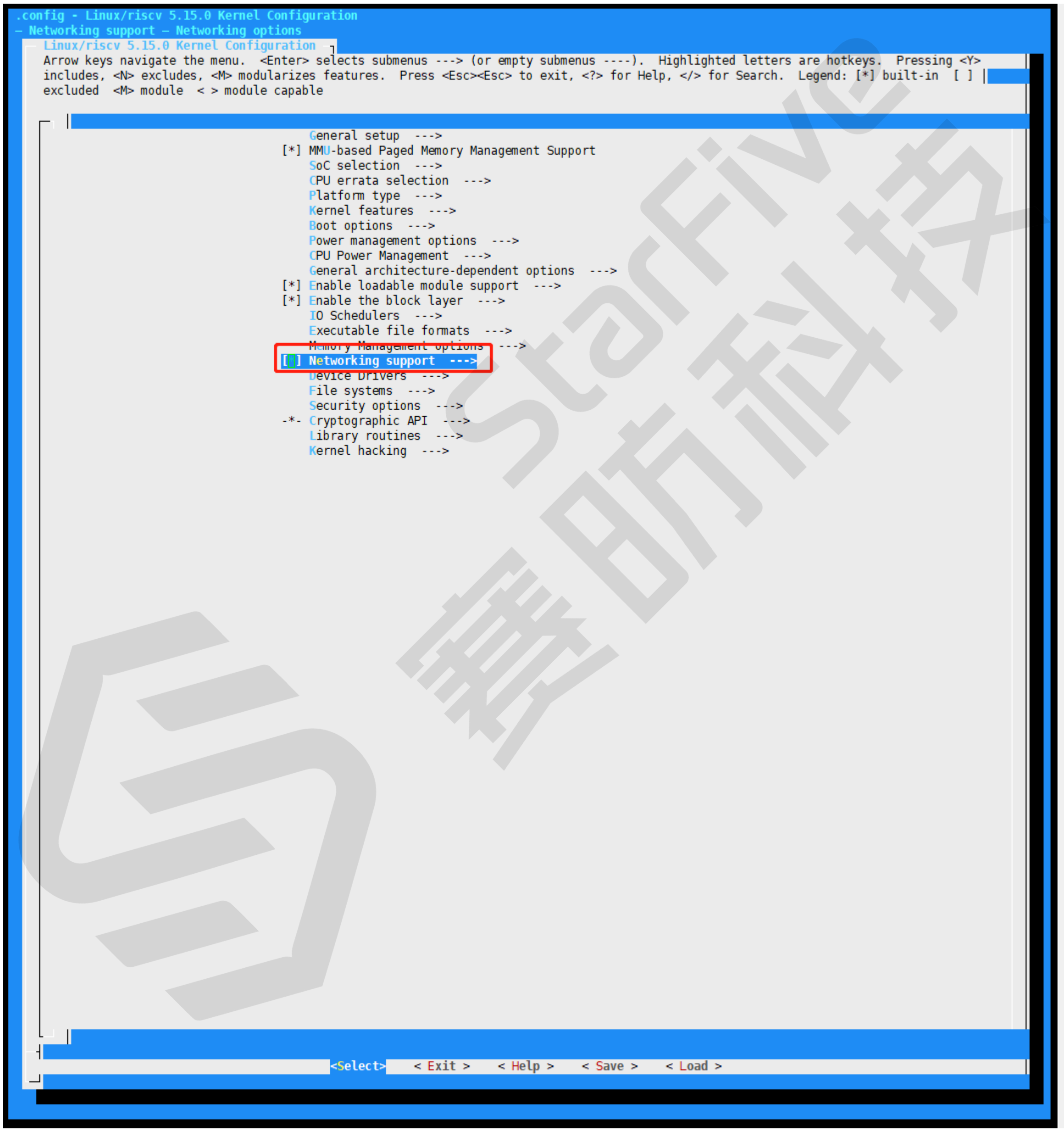
Follow the steps below to enable GMAC support in the kernel menu dialog.

1. Under the root directory of `freelight-u-sdk`, type the following command to enter the kernel menu configuration GUI.

```
make linux-menuconfig
```

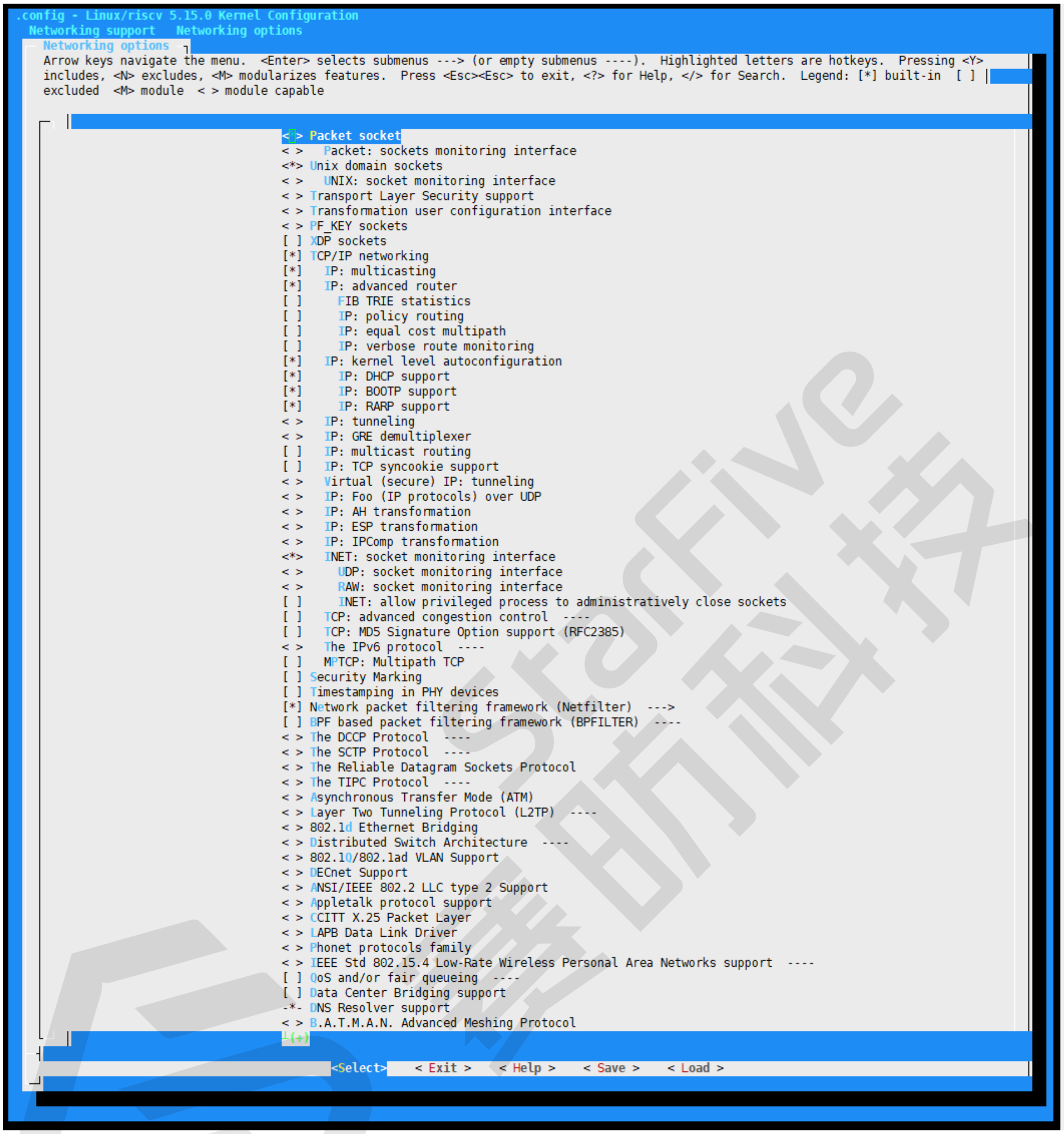
2. Enter the **Networking support** menu.

Figure 2-3 Networking Support



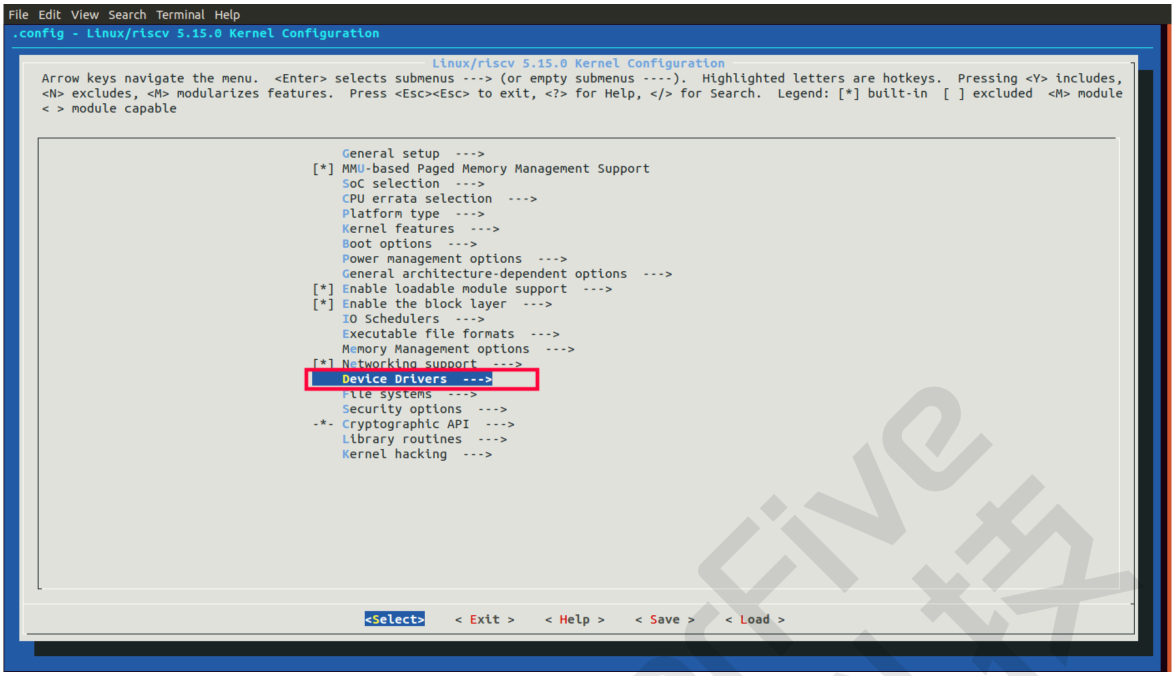
3. Ether the **Networking options** menu, and in the menu, select the **supported network protocols**.

Figure 2-4 Networking Options



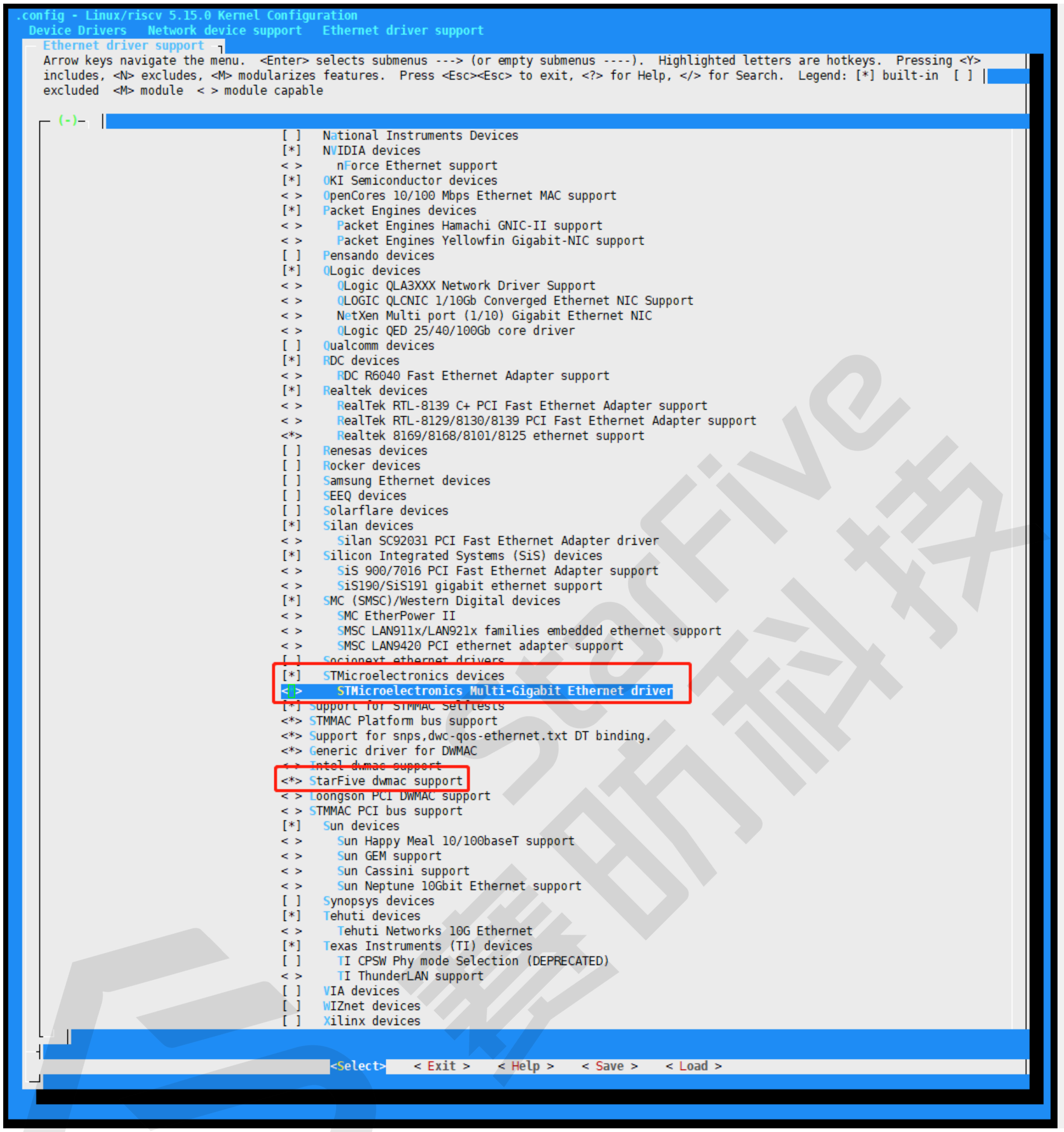
4. Enter the **Device Drivers** menu.

Figure 2-5 Device Drivers



5. Enter the **Network device support > Ethernet drivers support** menu and select the GMAC drivers you expect the system to support.

Figure 2-6 Ethernet Driver Support



6. Save your change before you exit the kernel configuration dialog.

2.4.2. Device Driver Configuration

A DTS/DTSI file is used to store all the device tree configuration.

The device tree of Ethernet is stored in the following path:

```
linux-5.10/arch/riscv/boot/dts/starfive/
```

The following code block shows the DTS file structure for Ethernet.

```
linux-5.15.0
└── arch
└── | -- riscv
| -- | -- | -- boot
| -- | -- | -- | -- dts
| -- | -- | -- | -- | -- starfive
| -- | -- | -- | -- | -- | -- jh7110-common.dtsi
| -- | -- | -- | -- | -- | -- jh7110.dts
```

The following code block shows an example of the device tree source code of the "**gmac0**" in the file `jh7110.dts`.

```
gmac0: ethernet@16030000 {
 compatible = "starfive,dwmac","snps,dwmac-5.10a";
 reg = <0x0 0x16030000 0x0 0x10000>;
 clock-names = "gtx",
  "tx",
  "ptp_ref",
  "stmmaceth",
  "pclk",
  "gtxc",
  "rmii_rtx";
 clocks = <&clkgen JH7110_GMAC0_GTXCLK>,
   <&clkgen JH7110_U0_GMAC5_CLK_TX>,
   <&clkgen JH7110_GMAC0_PTP>,
   <&clkgen JH7110_U0_GMAC5_CLK_AHB>,
   <&clkgen JH7110_U0_GMAC5_CLK_AXI>,
   <&clkgen JH7110_GMAC0_GTXC>,
   <&clkgen JH7110_GMAC0_RMII_RTX>;
 resets = <&rstgen RSTN_U0_DW_GMAC5_AXI64_AHB>,
   <&rstgen RSTN_U0_DW_GMAC5_AXI64_AXI>;
 reset-names = "ahb", "stmmaceth";
 interrupts = <7>, <6>, <5> ;
 interrupt-names = "macirq", "eth_wake_irq", "eth_lpi";
 max-frame-size = <9000>;
 phy-mode = "rgmii-id";
 snps,multicast-filter-bins = <64>;
 snps,perfect-filter-entries = <128>;
 rx-fifo-depth = <2048>;
 tx-fifo-depth = <2048>;
 snps,fixed-burst;
 snps,no-pbl-x8;
 snps,force_thresh_dma_mode;
 snps,axi-config = <&stmmac_axi_setup>;
 snps,tso;
 snps,en-tx-lpi-clockgating;
 snps,en-lpi;
 snps,write-requests = <4>;
 snps,read-requests = <4>;
 snps,burst-map = <0x7>;
 snps,txpbl = <16>;
 snps,rxpbl = <16>;
 status = "disabled";
};
```

The following list provides explanations for the parameters included in the above code block.

- **compatible**: Compatibility information, used to associate the driver and its target device.
- **reg**: Register base address "`0x16030000`" and range "`0x10000`".
- **clocks**: The clocks used by the Ethernet module.
- **clock-names**: The names of the above clocks.
- **resets**: The reset signals used by the Ethernet module.
- **reset-names**: The names of the above reset signals.
- **interrupts**: Hardware interrupt ID.
- **interrupt-names**: The names of the above interrupts.
- **phy-mode**: The Ethernet PHY mode, for example, "`rgmii`" or "`rmii`".
- **snps**: See Synopsis documentation for PHY specific parameters.
- **status**: The work status of the Ethernet, "enabled" or "disabled".

The following code block shows an example of the device tree source code of the "**gmac0**" in the file `jh7110-common.dtsi`:

```
&gmac0 {
 status = "okay";
```

```
 #address-cells = <1>;
 #size-cells = <0>;
 phy0: ethernet-phy@0 {
  rxc_dly_en = <1>;
  rx_delay_sel = <0>;
  tx_delay_sel_fe = <5>;
  tx_delay_sel = <0xa>;
  tx_inverted_10 = <0x1>;
  tx_inverted_100 = <0x1>;
  tx_inverted_1000 = <0x1>;
 };
};

&gmac1 {
 #address-cells = <1>;
 #size-cells = <0>;
 status = "okay";
 phy1: ethernet-phy@1 {
  tx_delay_sel_fe = <5>;
  tx_delay_sel = <0>;
  rxc_dly_en = <0>;
  rx_delay_sel = <0>;
  tx_inverted_10 = <0x1>;
  tx_inverted_100 = <0x1>;
  tx_inverted_1000 = <0x0>;
 };
};};22 };
```

The following list provides an explanation of the parameters in the above code block.

- **rxc_dly_en**: This field is used to set whether to enable the 2ns time delay of the receiver in RGMII mode. `1`: Enable. `0`: Disable.
- **rx_delay_sel**: This field is used to configure the receiver clock time delay, 150 ps per step width, accepted range: `0x0` - `0xf`.
- **tx_delay_sel_fe**: This field is used to configure the transmitter clock time delay in 10 M/100 M mode, 150 ps per step width, accepted range: `0x0` - `0xf`.
- **tx_delay_sel**: This field is used to configure the transmitter clock time delay in 1,000 M mode, 150 ps per step width, accepted range: `0x0` - `0xf`.
- **tx_inverted_10**: This field is used to set whether to enable the transmitter clock inversion in 10 M mode. `1`: Enable. `0`: Disable.
- **tx_inverted_100**: This field is used to set whether to enable the transmitter clock inversion in 100 M mode. `1`: Enable. `0`: Disable.
- **tx_inverted_1000**: This field is used to set whether to enable the transmitter clock inversion in 1,000 M mode. `1`: Enable. `0`: Disable.

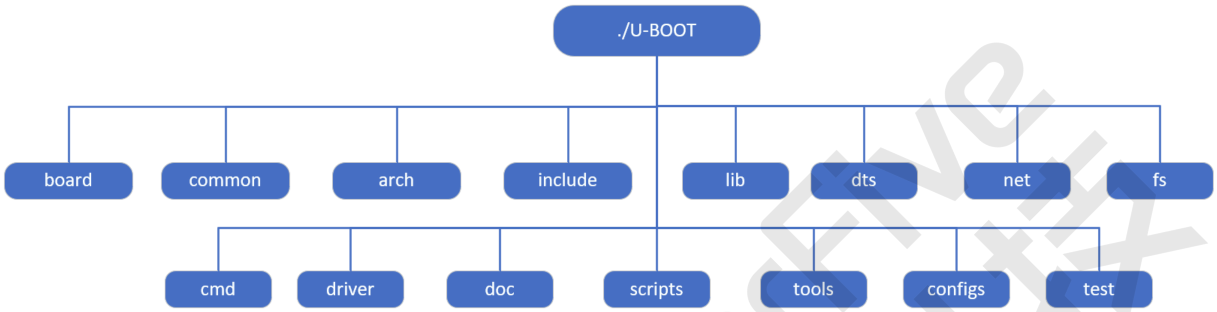
3. U-Boot Initialization

This chapter introduces how to initialize U-Boot as a preparation for adding a new device driver.

3.1. U-Boot Source Code Structure

The following image shows the U-Boot source code file directory for JH7110.

Figure 3-1 U-Boot Source Code Structure



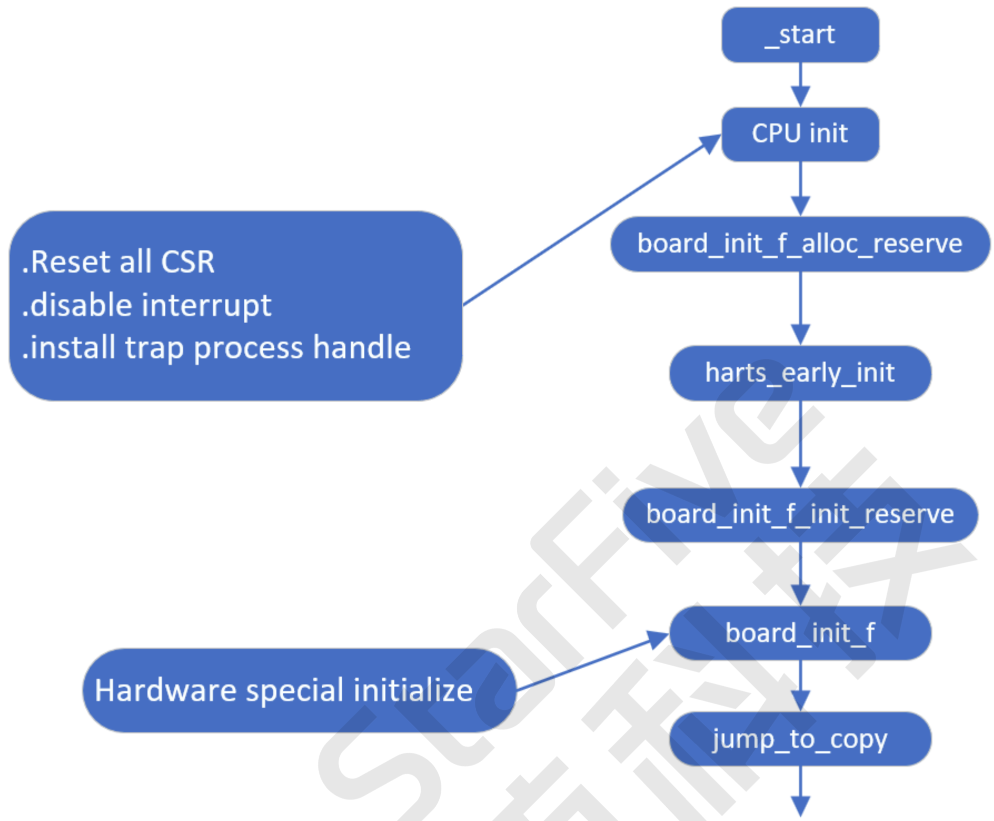
The following list provides an introduction for some of the above folders.

- **board**: The board folder contains all the board-specific files, including the files for StarFive JH7110 and the files for VisionFive 2, etc.
- **arch**: The core-specific folder which contains all the core initialization files. The files are not board-independent; thus, you don't need to modify anything in this folder.
- **driver**: The folder includes all the drivers supported by U-Boot, including the Ethernet driver, the PHY driver, the USB driver, and so on.
- **net**: The folder contains all the upper-layer protocols support in U-Boot, including the `ping`, the `tftp`, the `icmp`, and other protocols.
- **cmd**: The folder includes all the commands supported by U-Boot.
- **configs**: The folder includes all the deconfiguration files, each file related to a special board.
- **scripts**: The folder includes the rule files which used for compilation.

3.2. U-Boot Boot-up Process

The following diagrams show the U-Boot boot-up process.

Figure 3-2 U-Boot Boot-up Process 1



The following list provides a description of each procedure mentioned in the above diagram.

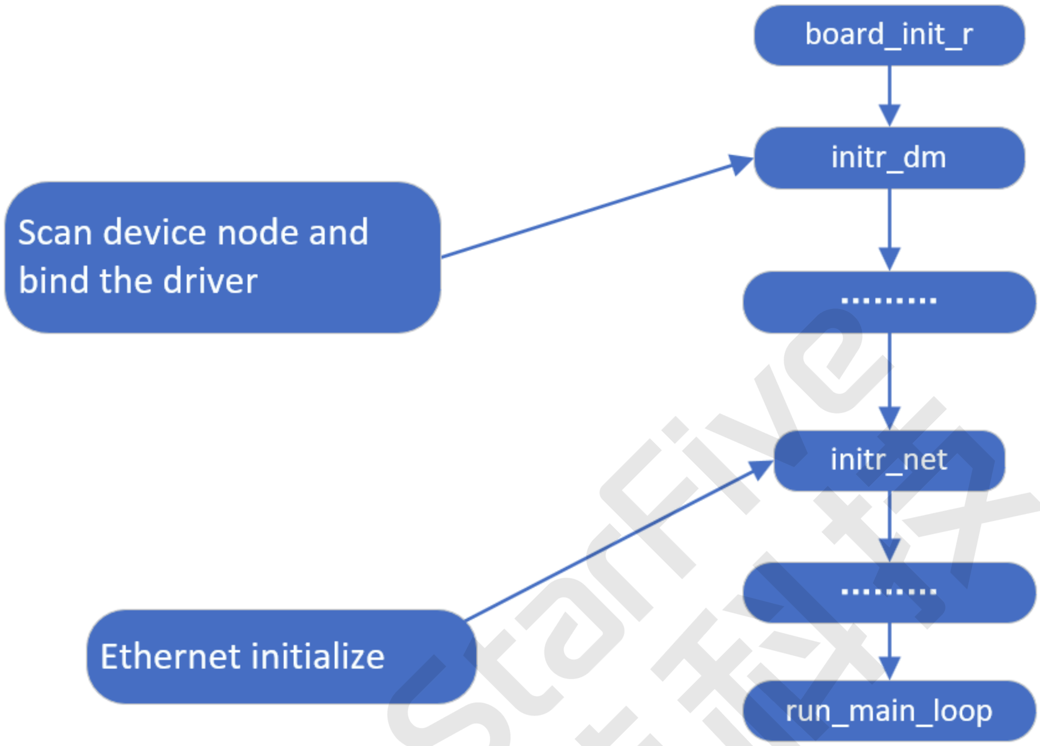
- **_start**: Each board with the same arch has the same `start.s` file. The file is located under the arch directory. The **_start** as the first instruction the system will use when the core powers on.
- **CPU init**: The CPU initialization step, which will set up all the CPU-related and specific registers. The step will also set up the RISC-V core-specific registers as illustrated in the above figure.

Note:
U-Boot will only use one core for boot-up, all the other cores are set to idle mode. Most of the time U-boot does not use a secondary core until the Linux is up.

- **board_init_f_alloc_reserve**: Reserve early **malloc** arena and global data **struct** arena.
- **harts_early_init**: Configure proprietary settings and customized CSRs of harts.
- **board_init_f_init_reserve**: Initialize reserved space.
- **board_init_f**: Initialize the basic hardware and running environment before relocate symbols, such as CPU, timer, console, device tree etc.
- **jump_to_copy**: Copy the global data **struct** to high address space and relocate the monitor code.

After the relocate symbols and the monitor code, the system will start the following boot-up process.

Figure 3-3 U-Boot Boot-up Process 2



The following list provides a description of each procedure mentioned in the above diagram.

- **board_init_r**: The board initialization file. All the board-related initialization processes as illustrated in the above diagram will be performed one by one.
- **init_dm**: Scan the device nodes and keep associated with the appropriate driver.
- **initr_net**: The Ethernet initialization file. The file will initialize all the Ethernet interfaces you expect to include on your board.
- **main_loop**: The final initialization step before the U-Boot pops up on your screen.

Result: After all of the entire processes, U-boot is up and ready for use.

In the **initr_net** process, a function named **phy_init** which is located under the `drivers/net/phy/phy.c` folder will be called to initialize the Ethernet PHY. See PHY Device Initialization *(on page 23)* for more information.

4. Adding a New Ethernet Driver

If the Ethernet PHY used is not already supported within U-Boot, you can follow the procedures below to add the PHY driver code for your new device.

4.1. Ethernet Driver Structure

The following code block shows the Ethernet PHY structure on the high-level overview.

```
phy_yutai_init(void)
{
    phy_register(&YT8512_driver);
    phy_register(&YT8521_driver);
    phy_register(&YT8531_driver);

    return 0;
}
```

The above file contains all the Ethernet PHY supported (self-adaptive) by default in U-Boot.

Exactly as described in this file, the system will initialize the PHY mentioned one by one.

Note:
If you find the boot-up process spends too much time, by examining each PHY use, you may remove some unused PHY and leave only the required ones.

The following image shows a specific U-Boot PHY structure as an example.

Figure 4-1 U-Boot PHY Structure Example

```
static struct phy_driver YT8521_driver = {
    .name = "YuTai YT8521",
    .uid = 0x0000011a,
    .mask = 0x00000fff,
    .features = PHY_GBIT_FEATURES,
    .config = &yt8521_config,
    .startup = &ytphy_startup,
    .shutdown = &genphy_shutdown,
};
```

The following list provides descriptions for the above parameters.

- **.name**: The name of the Ethernet PHY that you want to support, and you can input a random name, but it is recommended to input a device-specific name for future maintenance.
- **.uid**: The manufacturer ID as well as the device ID of the Ethernet PHY which can be found in the manual from the PHY manufacturer.
- **.mask**: The mask of the Ethernet PHY, in the example, "`0x00000fff`", the position of digit of "`f`" is the UID number. In practice, this digit can be omitted to simplify the input.
- .**feature**: The Gigabit feature of the PHY. For example, whether the Ethernet PHY is a Gigabit PHY or not.
- **.config**: The function call which introduces how to initialize the Ethernet PHY. For most PHY, the configuration is not needed. For complex PHY with QSGMI and RMII, the configuration is required to specify the role of the PHY.

4.2. Adding a New PHY

For example, if you wish to add a new PHY named YT8531 from Motorcomm, you need to locate the file `drivers/net/phy/motorcomm.c`, and perform the following operations.

- Create a new structure following the existing data structure. The data structure in the file is defined by U-Boot, to add your new PHY support, you must follow the data structure and format exactly.
- Reuse the existing start-up and shut-down functions. Modify them only when your device has special requirements.
- Ensure you have registered the new PHY by adding a function call of **phy_register()** as a new entry, for example:

```
phy_register(&YT8531_driver)
```

Note:

If you are adding a PHY from other vendors, ensure you find the right document written in C for PHY registration, for example, for Broadcom PHY, use the file `broadcom.c`.

4.3. Enable PHY on U-Boot

Follow the steps below to enable the new PHY on U-Boot.

1. To enable your new PHY for the U-Boot, first you need to define the macro definition in the board specific header file.

   The following code block provides an example of adding the YT8531 PHY in the VisionFive 2 header file `include/configs/starfive-visionfive.h.h`.

   ```
   #define DWC_NET_PHYADDR
   ```

   Note:
   Make sure the PHY address you defined in the header file is correct, otherwise, the system has to enumerate all the PHY address available.

2. Then you need to add the defined macro definition in the configuration file.

   The following image shows an example of adding the YT8531 PHY in the configuration file.

Figure 4-2 Add PHY in Configuration File

```
int phy_init(void)
{

#ifdef CONFIG_B53_SWITCH
	phy_b53_init();
#endif
#ifdef CONFIG_MV88E61XX_SWITCH
	phy_mv88e61xx_init();
#endif
#ifdef CONFIG_PHY_AQUANTIA
	phy_aquantia_init();
#endif
#ifdef CONFIG_PHY_ATHEROS
	phy_atheros_init();
#endif
.....
.....

#ifdef CONFIG_PHY_NCSI
	phy_ncsi_init();
#endif
#ifdef CONFIG_PHY_XILINX_GMII2RGMII
	phy_xilinx_gmii2rgmii_init();
#endif
#ifdef CONFIG_PHY_YUTAI
	phy_yutai_init();
#endif
	genphy_init();

	return 0;
}
```

3. Then you can add a new entry for PHY device initialization.

   The following image provides an example of adding the YT8531 PHY in the file `drivers/net/phy/motorcomm.c`.

Figure 4-3 Add PHY in Device Initialization

```
int phy_yutai_init(void)
{
	phy_register(&YT8512_driver);
	phy_register(&YT8521_driver);
	phy_register(&YT8531_driver);

	return 0;
}
```

4. Then you need to define the driver structure.

The following image provides an example of defining the data structure of the YT8531 PHY in the file `drivers/net/phy/motorcomm.c`.

Figure 4-4 Define PHY Data Structure

```
static struct phy_driver YT8531_driver = {
    .name           = "YT8531 Gigabit Ethernet",
    .uid            = PHY_ID_YT8531,
    .mask           = MOTORCOMM_PHY_ID_MASK,
    .features       = PHY_GBIT_FEATURES,
    .config         = &yt8531_config,
    .startup        = &ytphy_startup,
    .shutdown       = &genphy_shutdown,
};
```

4.4. PHY Device Initialization

The following image shows an example of the YT8521 PHY device initialization code.

Figure 4-5 YT8521 PHY Initialization

```
static int yt8521_config(struct phy_device *phydev)
{
    int ret, val;

    ret = 0;
    genphy_config_aneg(phydev);

    /* disable auto sleep */
    val = ytphy_read_ext(phydev, EXTREG_SLEEP_CONTROL);
    if (val < 0)
        return val;

    val &= ~(1 << YT8521_EN_SLEEP_SW_BIT);
    ret = ytphy_write_ext(phydev, EXTREG_SLEEP_CONTROL, val);
    if (ret < 0)
        return ret;

    /*set delay config*/
    ret = ytphy_of_config(phydev);
    if (ret < 0)
        return ret;

    val = ytphy_read_ext(phydev, YT8521_EXT_CLK_GATE);
    if (val < 0)
        return val;

    val &= ~(1 << 12);
    ret = ytphy_write_ext(phydev, YT8521_EXT_CLK_GATE, val);
    if (ret < 0)
        return ret;

    return 0;
}
```

The following images show an example of the YT8531 PHY device initialization code.

Figure 4-6 YT8531 PHY Initialization 1

```
static int yt8531_config(struct phy_device *phydev)
{
    int ret;

    ret = 0;
    genphy_config_aneg(phydev);

    /* set delay config */
    ret = ytphy_of_config(phydev);
    if (ret < 0)
        return ret;
    return 0;
}
```

Figure 4-7 YT8531 PHY Initialization 2

```
static int ytphy_of_config(struct phy_device *phydev)
{
	ofnode node;
	u32 val;
	u32 cfg;
	int i;

	node = phydev->node;
	if (!ofnode_valid(node)) {
		/* Look for a PHY node under the Ethernet node */
		node = dev_read_subnode(phydev->dev, "ethernet-phy");
	}

	if (!ofnode_valid(node)) /* No node found*/
		return 0;

	/*read rxc_dly_en config*/
	cfg = ofnode_read_u32_default(node, ytphy_rxden_grp[0].name, ~0);
	if (cfg != -1) {

		val = ytphy_read_ext(phydev, YTPHY_EXTREG_CHIP_CONFIG);

		/*check the cfg overflow or not*/
		cfg = (cfg > ((1 << ytphy_rxden_grp[0].size) - 1)) ?
			((1 << ytphy_rxden_grp[0].size) - 1) : cfg;

		val = bitfield_replace(val, ytphy_rxden_grp[0].off,
			ytphy_rxden_grp[0].size, cfg);
		ytphy_write_ext(phydev, YTPHY_EXTREG_CHIP_CONFIG, val);
	}

	/* set drive strenght of rxd/rx_ctl rgmii pad */
	val = ytphy_read_ext(phydev, YTPHY_PAD_DRIVES_STRENGTH_CFG);
	val |= YTPHY_RGMII_SW_DR_MASK;
	ytphy_write_ext(phydev, YTPHY_PAD_DRIVES_STRENGTH_CFG, val);

	val = ytphy_read_ext(phydev, YTPHY_EXTREG_RGMII_CONFIG1);
	for (i = 0; i < ARRAY_SIZE(ytphy_rxtxd_grp); i++) {

		cfg = ofnode_read_u32_default(node,
			ytphy_rxtxd_grp[i].name, ~0);
		cfg = (cfg != -1) ? cfg : ytphy_rxtxd_grp[i].dflt;

		/*check the cfg overflow or not*/
		cfg = (cfg > ((1 << ytphy_rxtxd_grp[i].size) - 1)) ?
			((1 << ytphy_rxtxd_grp[i].size) - 1) : cfg;

		val = bitfield_replace(val, ytphy_rxtxd_grp[i].off,
				ytphy_rxtxd_grp[i].size, cfg);
	}

	return ytphy_write_ext(phydev, YTPHY_EXTREG_RGMII_CONFIG1, val);
}
```

The above function calls specify how to initialize the Ethernet PHY. You have to use the MDIO bus to access the PHY control registers. Thus, you need to make sure your MDIO interface is configured properly before the configuration.

5. Driver Verification

5.1. Verification Environment

Before you start to verify the new Ethernet driver, you need to define the environment variables for the following items.

- U-Boot
- Board IP address (by setting the variable of **ipaddr**)
- Active Ethernet Interface (by setting the variable of **ethact**)
- Interface MAC address (by setting the variable of **ethaddr**)

As a single-process operating system, Linux can only operate one Ethernet driver (interface) at a time. Thus you need to specify in the above parameters to inform U-Boot which interface is active before use.

The following code block provides an example.

```
===>print
baudrate=115200
boottargs=console=ttyS0,115200 debug rootwait earlycon=sbi
bootcmd=run load_vf2_env;run importbootenv;run boot2;run distro_bootcmd
bootcmd_mmc0=devnum=0; run mmc_boot
bootde|ay=2
bootdir=/boot
eth0addr=6c:cf:39:7c:4e:22
ethladdr=6c:cf:39:7c:3e:53
ethact=ethernet@16030000
ethaddr=6c:cf:39:7c:4e:22
ipaddr=192.168.120.230
netmask=255.255.255.0
stderr=serial@10000000
stdin=serial@10000000
stdout=serial@10000000
```

5.2. New Driver Verification

After you have added the new Ethernet driver, when you access U-Boot in the second time, you will see the following screen.

Figure 5-1 Ethernet Driver Verification

```
U-Boot 2021.10-dirty (Nov 23 2022 - 15:24:46 +0800)

CPU:   rv64imacu
Model: StarFive VisionFive V2
DRAM:  8 GiB
MMC:   sdio0@16010000: 0, sdio1@16020000: 1
Loading Environment from SPIFlash... SF: Detected gd25lq128 with page size 256 Bytes, erase size 4 KiB, total 16 MiB
*** Warning - bad CRC, using default environment

StarFive EEPROM format v2

--------EEPROM INFO--------
Vendor : StarFive Technology Co., Ltd.
Product full SN: VF7110B1-2228-D008E032-00000001
data version: 0x2
PCB revision: 0x1
BOM revision: B
Ethernet MAC0 address: 6c:cf:39:7c:4e:22
Ethernet MAC1 address: 6c:cf:39:7c:3e:53
--------EEPROM INFO--------

In:    serial@10000000
Out:   serial@10000000
Err:   serial@10000000
Model: StarFive VisionFive V2
Net:   eth0: ethernet@16030000, eth1: ethernet@16040000
switch to partitions #0, OK
mmc1 is current device
found device 1
bootmode flash device 1
Failed to load 'uEnv.txt'
Can't set block device
Hit any key to stop autoboot:  0
StarFive #
StarFive #
```

The above highlighted information shows that the SoC support for the interface is ready for use, however, we still need to verify the data communication in case the data is blocked in the PHY.

5.3. Access PHY via MIDO Command

You need to access the Ethernet PHY using the MDIO command.

The following image shows a list of the Ethernet PHYs, each with a corresponding command to access the PHY.

Figure 5-2 MIDO Commands

```
StarFive #
StarFive # mdio list
ethernet@16030000:
ethernet@16040000:
StarFive #
StarFive #
```

You can use the above command to examine whether the PHY is ready on your board for data communication.

5.4. PING - Digital Loopback

After you have confirmed the access to the PHY is ready, you can use the PING command to initiate a digital loopback for send and receive ping data packages.

To initiate the test, run the command `ping $ipaddr`.

The following figure shows an example return of executing the command.

Figure 5-3 Ping Command

```
StarFive #
StarFive # ping 192.168.120.72
ethernet@16030000 Waiting for PHY auto negotiation to complete.... done
Using ethernet@16030000 device
host 192.168.120.72 is alive
StarFive # ping 192.168.120.72
Using ethernet@16030000 device
host 192.168.120.72 is alive
StarFive #
```

6. Debug Methods

6.1. General Debug Commands

The following list provides examples for the commands generally used for debugging Ethernet connections.

- Check Ethernet device information:
  - Check adapter status:

    ```
    ifconfig eth0
    ```

  - Check data package send and receive statistics:

    ```
    cat /proc/net/dev
    ```

  - Check the current speed:

    ```
    cat /sys/class/net/eth0/speed
    ```

- Enable or disable an Ethernet device.
  - Enable:

    ```
    ifconfig eth0 up
    ```

  - Disable:

    ```
    ifconfig eth0 down
    ```

- Configure an Ethernet device.
  - Configure static IP address:

    ```
    ifconfig eth0 192.168.1.101
    ```

  - Configure MAC address:

    ```
    ifconfig eth0 hw ether 00:11:22:aa:bb:cc
    ```

  - Automatically obtain the IP address:

    ```
    udhcpc -i eth0
    ```

  - Set PHY mode: (Set the speed of 100 M, enable full duplex and auto negotiation.)

    ```
    ethtool -s eth0 speed 100 duplex full autoneg on
    ```

- General test commands:
  - Connection test:

    ```
    ping 192.168.1.101
    ```

  - Throughput test:

    Note:
    Make sure you have enabled the **iperf** tool in the kernel menu before performing the test.

    - TCP throughput test:

      Server side:

      ```
      iperf3 -s -i 1
      ```

Client side:

```
iperf3 -c 192.168.1.101 -i 1 -t 60 -P 4
```

- UDP throughput test:

  Server side:

```
iperf3 -s -u -i 1
```

  Client side:

```
iperf3 -c 192.168.1.101 -u -b 100M -i 1 -t 60 -P 4
```

6.2. General Troubleshooting Procedures

The topic introduces some general troubleshooting steps.

Software Troubleshooting

The following list shows the general troubleshooting steps for software problems.

1. Verify whether the PHY mode is configured correctly.
2. Verify whether the clock settings are configured correctly.
3. Verify whether the GPIO settings are configured correctly, for example, IO MUX (multiplexing) functions, drive strength, and pull-up/pull-down settings, etc.
4. Verify whether the PHY reset settings are configured correctly.
5. Use the following command to verify the status of sending and receiving data packets on "**eth0**".

```
cat /proc/net/dev
```

Hardware Troubleshooting

The following list shows the general troubleshooting steps for hardware problems.

1. Verify whether the PHY power supply **vcc-ephy** is working properly.
2. Verify whether the clock waveform looks good.

7. Known Issue

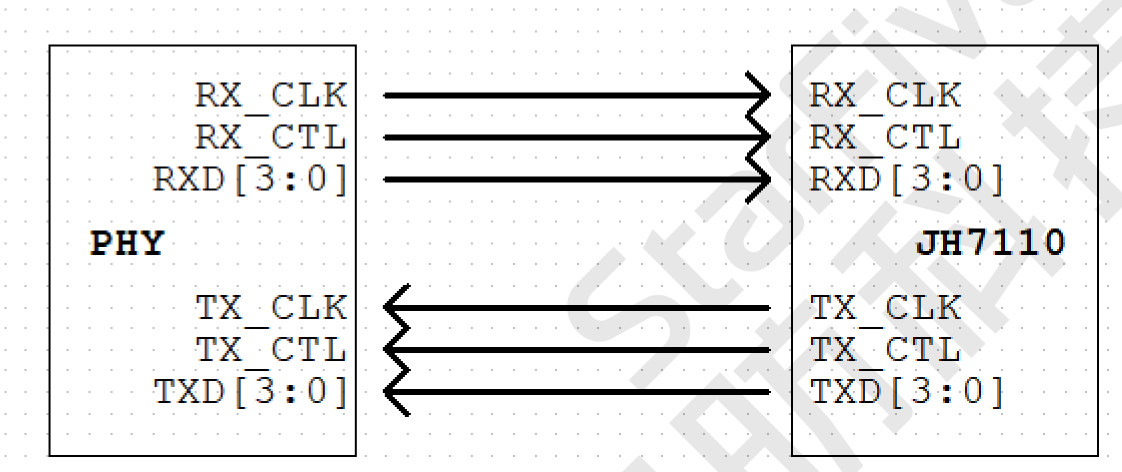
7.1. Ethernet GMAC Supports RGMII Only

JH7110 only supports RGMII mode for Ethernet GMAC connections. Due to this limitation, JH7110 has the following layout requirements.

7.1.1. 1,000 M Only

If you only need to support 1,000 M mode, you can design the layout following the requirements below.

Figure 7-1 GMAC 1,000 M Only



Layout requirements.

- The RX/TX trace length cannot exceed 6,000 mil.
- Match the RXD[3:0] signal group and the RX_CTL and RX_CLK signals with trace length to within 100 mil. Match the TXD[3:0] signal group and the TX_CTL and TX_CLK group trace length to within 100 mil.
- The routing of data and clock lanes should keep a complete reference plane.

7.1.2. Auto-Negotiation

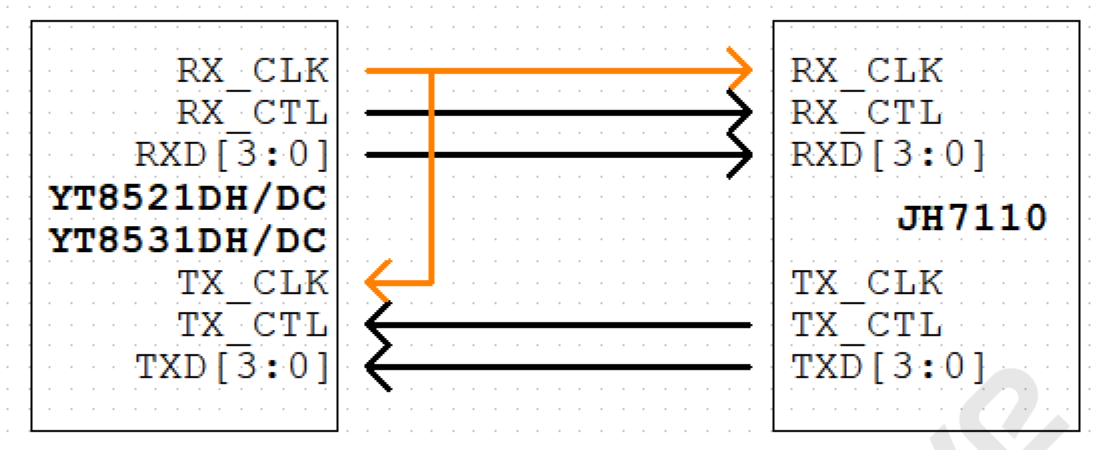
If you need to support 10/100/1,000 M mode auto-negotiation, you need to know the following limitations, and then you can design the layout following the requirements below.

Important:
For auto-negotiation mode, only the following PHY models are supported.

- YT8521DH/DC
- YT8531DH/DC

Plus, you need to connect the RX_CLK of the PHY to its TX_CLK as shown by the orange lines in the following diagram.

Figure 7-2 GMAC 10 M/100 M/1,000 M Auto-Negotiation



Layout requirements for GMAC0.

- The trace length from TX_CLK to RX_CLK cannot exceed 500 mil.
- The RX and TX trace length cannot exceed 4,300 mil.
- Match the RXD[3:0] signal group and the RX_CTL and RX_CLK signals with trace length to within 100 mil.
- Match the TXD[3:0] signal group and the TX_CTL and RX_CLK signals with trace length to within 100 mil.
- The routing of data and clock lanes should keep a complete reference plane.

Layout requirements for GMAC1.

- The trace length from TX_CLK to RX_CLK cannot exceed 500 mil.
- The RX_CLK trace length cannot exceed 4,000 mil. Match the RXD[3:0] signal group and the RX_CTL and RX_CLK signals with trace length to within 100 mil.
- The TX_CLK trace length is 2,000 mil longer than that of the RX_CLK. Match the TXD[3:0] signal group and the TX_CTL and RX_CLK signals with trace length to within 100 mil.
- The routing of data and clock lanes should keep a complete reference plane.