



FSL91030(M) chip

Test board and software manual

Version: **V1.6**

Wuhan Feisiling Microelectronics Technology Co., Ltd.

May 2023

1. Overview.....	3
1.1. Preface.....	3
1.2. Package Structure.....	3
2. BSP Introduction.....	4
2.1. Software Version.....	4
2.2. Startup Process.....	4
3. Test Board Usage.....	7
3.1. Startup.....	7
3.2. Login and Upgrade SDK.....	8
3.3. Port Numbering.....	9
4. Version Compilation & Flashing.....	12
4.1. Environment & Burning Tool Preparation.....	12
4.2. Bare metal SDK compilation & flashing.....	13
4.2.1. Unzip the bare metal SDK code and put it in the same directory as tools.....	13
4.2.2. Importing environment variables.....	13
4.2.3. Compile & Burn.....	13
4.3. Business SDK Compilation.....	15
4.3.1. Built-in CPU	15
4.3.2. External CPU	16
4.4. Linux SDK Compile & Burn.....	17
4.4.1. Hardware flash requirements.....	17
4.4.2. Partitions.....	17
4.4.3. Compilation preparation.....	18
4.4.4. Burn freeloader.elf via JTAG	19
4.4.5. ubifs version compilation & burning (default compilation and burning method for demo board).....	21
4.4.6. Compile and burn ramfs + jffs2 version.....	22
4.4.7. Compile and burn ramfs + ubifs versions.....	24
4.4.8. Commonly used commands under uboot.....	25
5. Configuration Software Usage.....	26
6. SDK Usage.....	28
7. Version History.....	29

1 Overview

1.1. Introduction

This document is the user manual of the FSL91030M chip test board and its supporting software package, which is used to introduce the FSL91030M chip software package. Porting, description and development guidance of functional components, and how to use the test board.

1.2. Package Structure

The structure diagram of the FSL91030M chip software package is as follows:

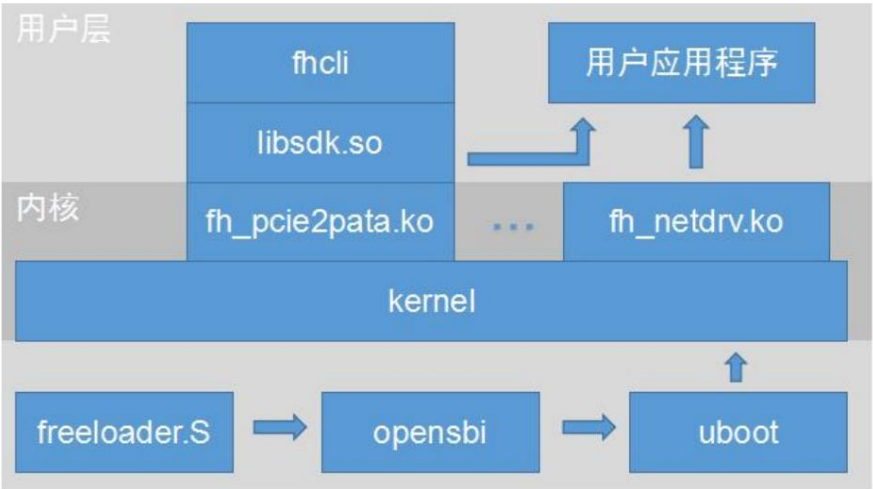


Figure 1-1 Structure diagram

The directory structure of the FSL91030M chip software package is shown in the following table:

Table 1-1 Software package directory structure

Package Name	Module Classification	Module Name	illustrate
linux_sdk.tgz tools.zip	BSP	freeloder.elf	It has opensbi and uboot integrated inside. After startup, it will boot into Enter uboot, uboot loads kernel and rootfs to start
		fdt.dtb	Device Tree
		kernel.bin	Kernel source code
		initrd.bin	Root file system
	SDK	Kernel	Kernel module driver
		User	User-mode code
		Thd_code	Use tool source code
	Compile script	Makefile	Unified compilation script
	Cross-compilation toolchain	riscv-nuclei-elf- gcc	Tool for compiling Linux system images
		riscv-nuclei-linu x-gnu-gcc	Compile other module mirroring tools of the software package

2. BSP Introduction

This chip integrates SOC subsystem, embedded RISC-V processor, and provides rich peripheral interfaces, including DDR, UART, GPIO, I2C, NOR flash, SPI, MDIO, Watchdog, etc.

2.1. Software Version

Software version:

Serial number	software name	Package Version
1	freeloder	-
2	opensbi	v 0.7
3	u-boot	2020.07-rc2
4	Linux Kernel	5.8.0
5	rootfs (root file system)	busybox version: 1.31.1 glibc version: 2.29

2.2. Startup process

The startup process after the chip is powered on is divided into the following 5 steps: freeloader -> opensbi -> uboot -> kernel -> rootfs, The following is an analysis from five stages:

The freeloader startup process is as follows:

Set the exception and interrupt jump address -> move opensbi, uboot, kernel, rootfs, fdt to the corresponding position of ddr -> jump
Go to opensbi

Opensbi

In the RISC-V architecture, there is an operating environment defined under the operating system. This operating environment will not only boot up RISC-V The operating system under it will also reside in the background and provide a series of binary interfaces for the operating system to obtain and operate hardware information. RISC-V provides a specification for such an environment and binary interface, called the "Operating System Binary Interface", or "SBI". Opensbi is a An implementation.

Opensbi not only provides guidance, but also provides the implementation of switching from M mode to S mode. At the same time, the kernel in S-Mode can Some M-Mode services can be accessed through this layer.

uboot

U-boot provides a command shell interface, similar to the command line of a Linux terminal. The supported commands mainly include: setting environment variables, network Network test command (ping), TFTP download command, NOR operation command (sp, etc.), memory operation command (md, mw, etc.), startup Kernel instructions, etc.

In addition, U-boot must provide system deployment function, transfer the image data into memory through network (tftp), and then burn the image data in DDR To Flash:

• Burning of the entire system (including dtb, U-boot, Kernel, Rootfs and other images);

• Upgrading the image in a single partition;

kernel

After the Linux kernel completes system initialization, it loads each module controller and then mounts a file system as the root file system.

System.

1) TABLE DMA

Supports configuration of switch chip table entries through TABLE DMA. The interface list is as follows:

Function name	describe
fslral_dma_mem_read	Read table entries via dma
fslral_dma_mem_write dma_alloc	Through dma configuration table items
	Allocate dma buffer
dma_free	Release dma cache
get_dma_info	Get the allocated dma buffer information

2) PACKET DMA

Supports sending and receiving messages between CPU and switch chip through PACKET DMA. The interface list is as follows:

Function name	describe
prepare_pkt	Configure the header of the packet to be sent
fsl_common_tx	Sending packets via pkt dma
fsl_common_rx_register	Register the packet receiving function
fsl_common_rx_unregister	Unregister the specified packet receiving processing function
fsl_common_rx_unregister_all fsl_common_rx_start	Unregister all packet receiving processing functions
fsl_common_rx_shutdown	Start receiving packets
	Stop receiving packets

3) I2C

Accessed through the /dev/i2c-x standard character device, the IOCTL interface is provided to allow applications to access and set I2C registers.

4) MDIO

Configure CPU temperature. The currently provided instructions are as follows:

Function name	describe
mdio_mode_cfg	Configure mdio to access internal or external
mdio_master_read	Read via mdio
mdio_master_write	Write via mdio

rootfs

The root file system is the storage area for files and data in a Linux system. It usually also includes system configuration files, applications such as Busybox, etc. Programs and libraries required by applications.

The current file system is in tmpfs format, and the changes made will be lost if the power is off. If you want to save the file permanently, you can mount the flash device first. Then put the file in the flash device. The folders included are as follows:

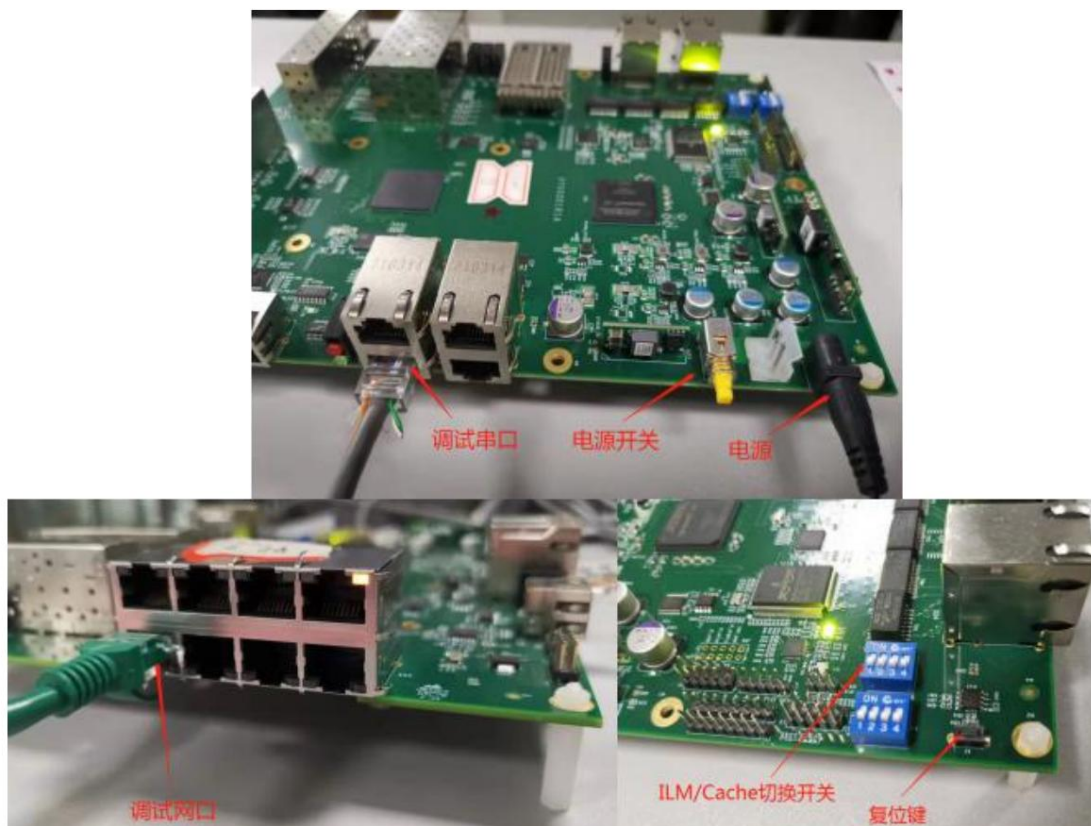
Table of contents	describe
/etc	System-related configuration files
/etc/init	Storage system startup related configuration
/sbin /bin	Storage system related tools, installed by Busybox
/usr/bin /usr/sbin	stores user tools, installed or customized by Busybox
/lib	A lot of library files are stored, and the application will load the required dynamic link library
/proc /	The mount point of the system's virtual file system, which contains virtual files and represents part of the system's status
sys	The mount point of the system's virtual file system, which fully represents the system status
/var	The mount point of the system's virtual file system, used to store some data
/dev	Store device files
/root	User Directory
/mnt	Mount the flash device

3. Test board use

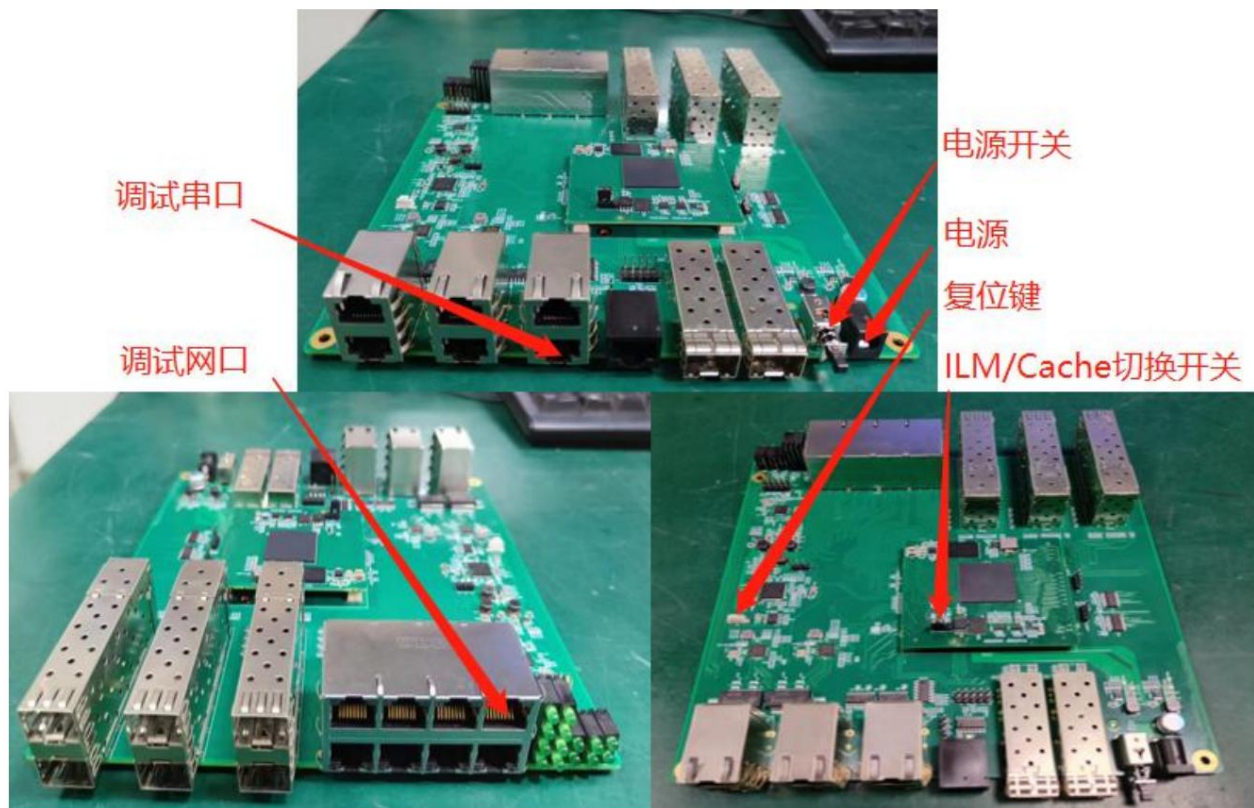
3.1. Startup

1) Assembly and connection

As shown in Figure 3-1 , connect the power cable, serial port cable (serial port baud rate 115200, 8N1) and network cable respectively. Turn off (select Cache mode); then turn on the power switch.



Connection diagram of test board without cover



Connection diagram of test board with cover

2) Log in to the Linux interface

After waiting for 30 seconds, the system automatically enters the Linux login interface.

```
Welcome to fsl xuanyuan System Technology
fsl login: █
```

3.2. Login and upgrade SDK

Login interface account: root, password: fsl.

After logging in, you will enter the "/"root" directory by default. The driver files (*.ko) in this directory are automatically installed when the system starts. Run "fhcli" command to enter the configuration interface. For related configuration operation information, please refer to Chapter 5.

Since the CPU network port is connected to port 0 (electrical port 0) of the switch chip, you need to enter the fhcli command line first and set the debug network port. The corresponding network device is set to eth0 to work properly.

The debugging network port is often used. The corresponding configuration operations are as follows:

```
quit //Exit fhcli
vi config.fhmc //Under root
cpu_port_enable=1 //cpu port enable
```



```
cpu_port=x //Logical port number of electrical port 0

wq sync after saving

Restart ./fhcli
```

You can use the ifconfig command to check that the network device corresponding to the debugging network port is eth0.

```
# ifconfig
[ 1268.940894] #enter get_stats
eth0      Link encap:Ethernet  HWaddr 00:02:AA:BB:CC:38
          inet addr:12.26.0.150  Bcast:12.26.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

After the network can be pinged, you can use the scp command or tftp command to upload and download files.

```
tftp -g -r libsdk.so 12.26.0.251

scp xxx@12.26.0.251:/home/libsdk.so (Note: Update the libsdk.so in this directory to upgrade the SDK)

chmod 777 libsdk.so
```

(Optional) You can modify mac and ip by running the following command:

```
ifconfig eth0 down

ifconfig eth0 hw ether 00:02:AA:BB:CC:13

ifconfig eth0 up

ifconfig eth0 12.26.0.122 netmask 255.255.0.0
```

3.3.Port Number

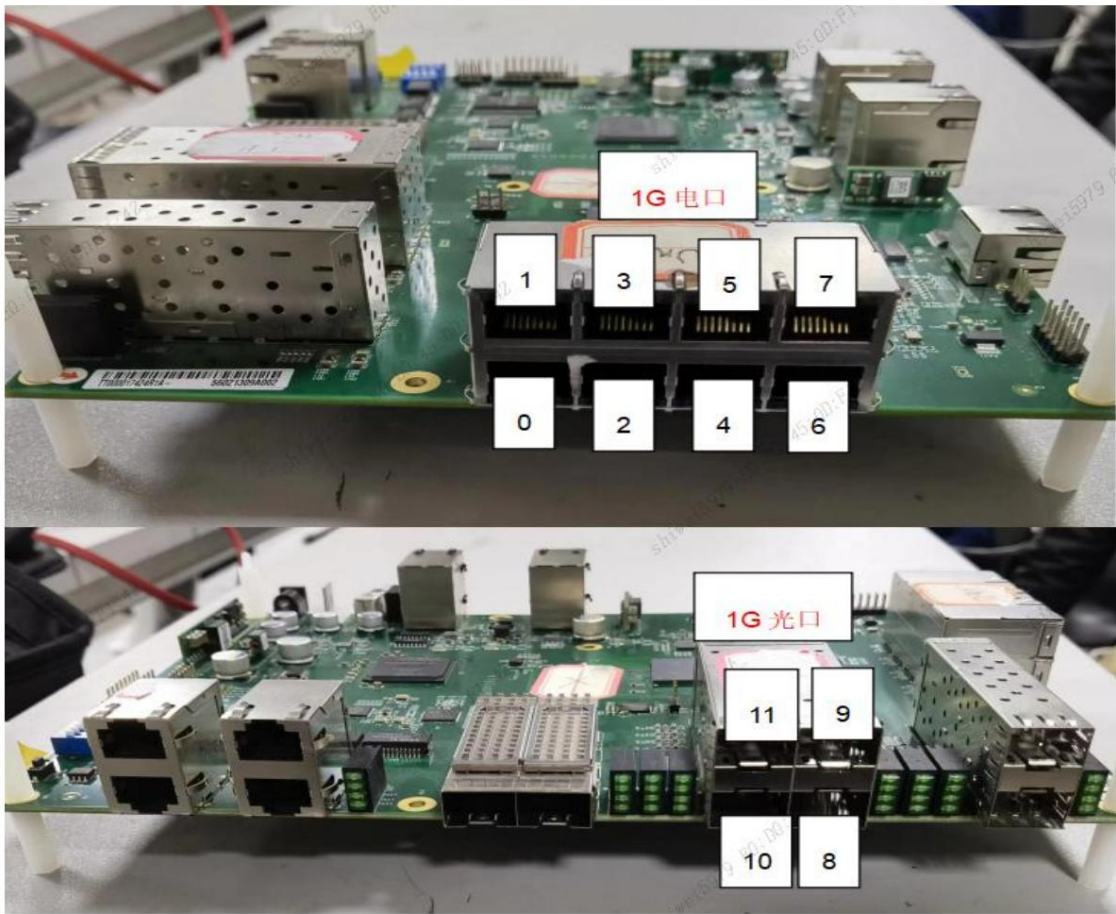
After the Demo board is powered on, log in through the serial port.

```
login:root
```

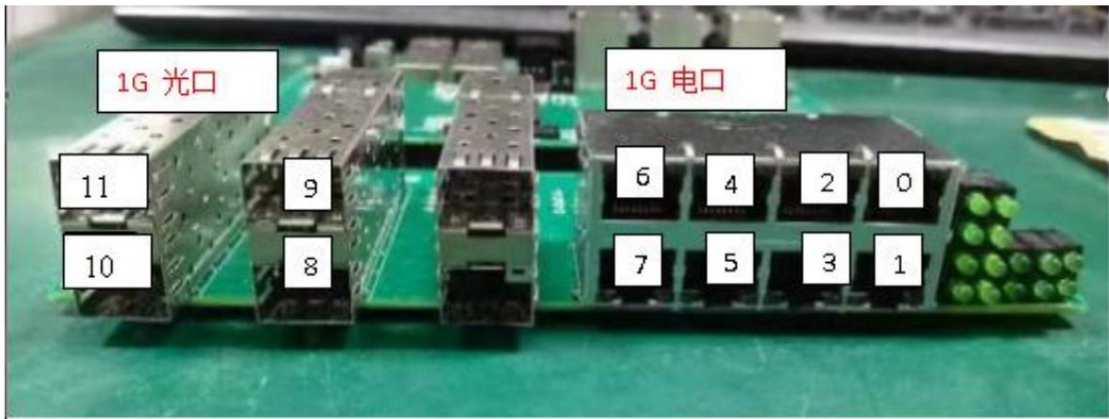
```
password:fsl
```

```
Run fhcli: ./fhcli
```

Switch the board port configuration to 8+4 mode, 8 electrical ports plus 4 1G optical ports (necessary for chip initialization):

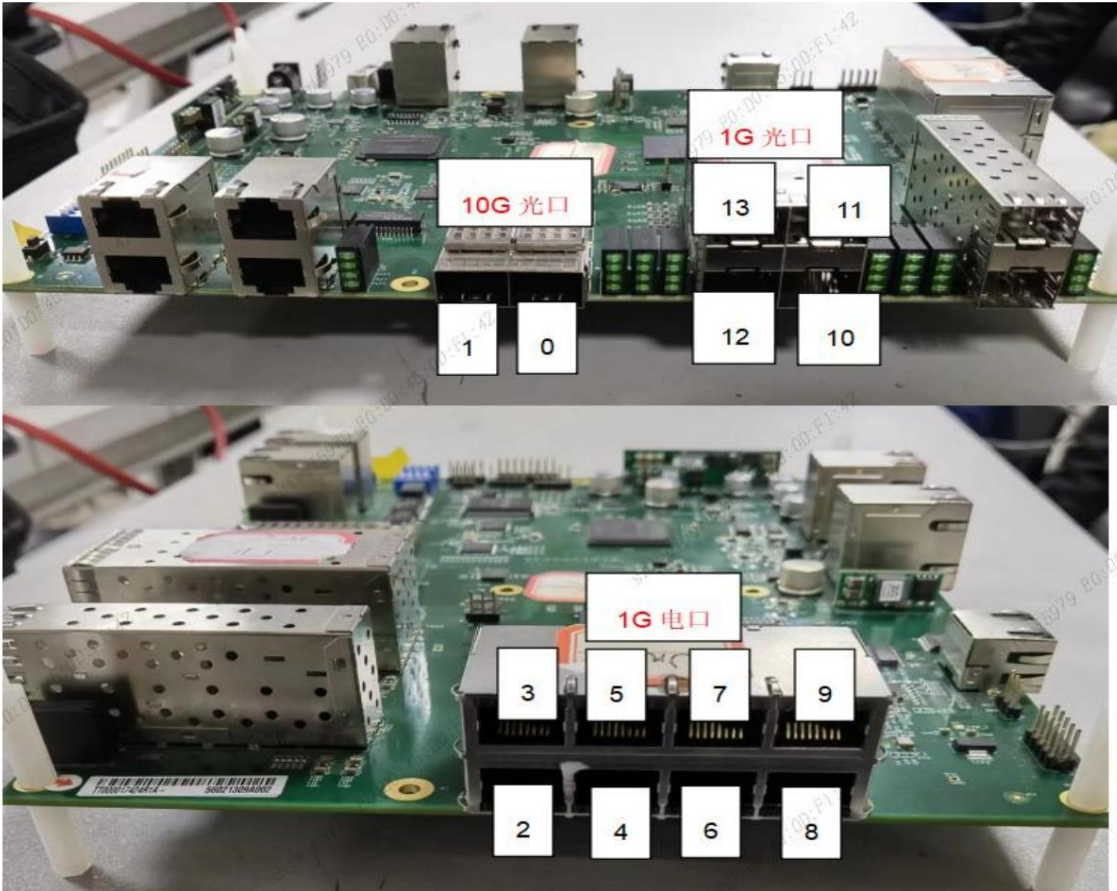


Connection diagram of test board without cover

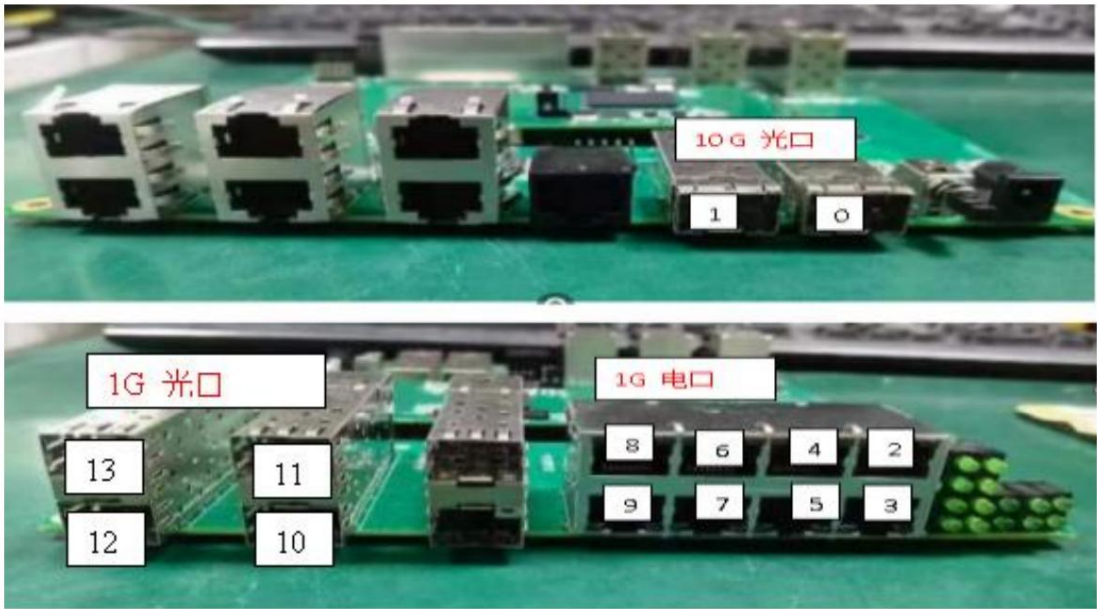


Connection diagram of test board with cover

Switch the demo board port configuration to 8+4+2 mode, 8 electrical ports plus 4 1G optical ports and 2 10G optical ports (chip initialization is necessary implement):



Connection diagram of test board without cover



Connection diagram of test board with cover











4. Version compilation & flashing

4.1. Environment & Flash Tool Preparation

Virtual machine: Ubuntu 20.04 and install the following tools

```
sudo apt-get install make
sudo apt install gcc
sudo apt install g++
sudo apt install git
sudo apt install bison
sudo apt install flex
sudo apt-get install device-tree-compiler
sudo apt-get install mtd-utils
```

Unzip the tools.zip tool package in the virtual machine (can be used for compiling and burning sdk, linux_sdk, and nuclei_sdk):

	gcc.tgz	2023/1/4 17:31	WinRAR 压缩文件	127,015 KB
	HBird_Driver.zip	2023/2/17 10:02	WinRAR ZIP 压缩...	6,576 KB
	linux_gcc.tgz	2023/1/4 17:33	WinRAR 压缩文件	379,691 KB
	openocd_nuclei.cfg	2022/11/1 15:41	CFG 文件	2 KB
	openocd_nuclei_ilm.cfg	2022/11/1 15:41	CFG 文件	2 KB
	openocd-2022.12-linux-x64.tgz	2023/1/4 19:02	WinRAR 压缩文件	3,674 KB
	openocd-2022.12-win32-x32.zip	2023/2/17 10:04	WinRAR ZIP 压缩...	7,186 KB
	sdktoolchain_cfg.sh	2022/11/29 16:44	SH 文件	1 KB
	setup.sh	2022/11/24 14:59	SH 文件	1 KB
	toolchain_cfg.sh	2022/11/24 14:59	SH 文件	1 KB

gcc.tgz and linux_gcc.tgz are the compilation tool chains and need to be decompressed.

setup.sh is used to import the toolchain before compiling nuclei_sdk. sdktoolchain_cfg.sh is used to import the toolchain before compiling business sdk.
toolchain_cfg.sh is used to import the toolchain before compiling linux_sdk.

Unzip HBird_Driver.zip and install it according to the internal "Installation Instructions.txt". Unzip openocd-2022.12-linux-x64.tgz
Or openocd-2022.12-win32-x32.zip. The following flashing instructions in this document take the Linux version of openocd as an example.

openocd_nuclei.cfg & openocd_nuclei_ilm.cfg are the openocd configuration files for flash and RAM respectively.

Establish a tftp server on the host computer or use tftp64.exe to download and update the kernel and root file system through tftp under uboot.

4.2. Bare metal SDK compilation & flashing

4.2.1. Unzip the bare metal SDK code and put it in the same directory as tools

```
tar -zxvf nuclei_sdk_v x_x_xx.tgz
```

4.2.2. Import environment variables

Modify the toolchain path in ~/tools/setup.sh

Source setup.sh in ~/tools/

```
NUCLEI_TOOL_ROOT=/home/sven/xy1030/Nuclei/gcc/bin
NMSIS_ROOT=../NMSIS

# Create your setup_config.sh
# and define NUCLEI_TOOL_ROOT like below
# NUCLEI_TOOL_ROOT=/home/develop/Software/Nuclei
# SETUP_CONFIG=setup_config.sh

[ -f $SETUP_CONFIG ] && source $SETUP_CONFIG

[ -f .ci/build_sdk.sh ] && source .ci/build_sdk.sh
[ -f .ci/build_applications.sh ] && source .ci/build_applications.sh

echo "Setup Nuclei SDK Tool Environment"
echo "NUCLEI_TOOL_ROOT=$NUCLEI_TOOL_ROOT"

export PATH=$NUCLEI_TOOL_ROOT:$PATH
```

4.2.3. Compile & Flash

In ~/nuclei-sdk-fh/nuclei_sdk_soc_test_cases/fh/RELEASE

```
make clean dasm/bin CORE=ux600f DOWNLOAD=flash HUBMODE_SW=0 BOARD_TYPE=1
LED_MODE=3 MANAGEMENT=1
```

HUBMODE_SW: 0(8+4), 1(8+2), 2(8+6), 3(8+4+2)

MANAGEMENT: 0(1030), 1(1030M)

BOARD_TYPE=1(COMMON) (FSL old demo boards(without pinch plate) Or user's boards)				
LED_MODE[1:0]=00	GEPHY [Link/Act]			
	1G	[Link/Act]		

	10G	[Link/Act]		
LED_MODE[1:0]=01	GEPHY [SPD1000/Link/Act]		[SPD100(10)/Link/Act]	
	1G	[SPD1000/Link/Act]	[Disable]	
	10G	[SPD1000/Link/Act]	[SPD10G/Link/Act]	
LED_MODE[1:0]=10	GEPHY [SPD1000/Link/Act]		[SPD100/Link/Act]	[SPD10/Link/Act]
	1G	[SPD1000/Link/Act]	[Disable]	[Disable]
	10G	[SPD1000/Link/Act]	[SPD10G/Link/Act]	[Disable]
LED_MODE[1:0]=11	GEPHY [Link/Act]		[SPD1000]	[SPD100]
	1G	[Link/Act]	[SPD1000]	[Disable]
	10G	[Link/Act]	[SPD1000]	[SPD10G]
BOARD_TYPE=0(OWNER) (FSL new demo boards(with pinch plate))				
LED_MODE[1:0]=00	GEPHY [Link/Act]			
	1G	[Act]		
	10G	[Act]		
LED_MODE[1:0]=01	GEPHY [SPD1000/Link/Act]		[SPD100(10)/Link/Act]	
	1G	[Act]	[Disable]	
	10G	[Act]	[Disable]	
LED_MODE[1:0]=10	GEPHY [Link]		[Act]	[Disable]

	1G	[Act]	[Disable]	[Disable]
	10G	[Act]	[Disable]	[Disable]
LED_MODE[1:0]=11	GEPHY [Link/Act]		[SPD1000]	[Disable]
	1G	[Act]	[Disable]	[Disable]
	10G	[Act]	[Disable]	[Disable]

For burning, refer to the burning steps of freeloader.elf in 4.4.4.

4.3. Business SDK compilation

4.3.1. Built-in CPU

1) Import environment variables

Modify the SDK_PATH in ~/sdk/build_config.cfg to the path of the current sdk/

Modify the toolchain path in ~/tools/sdktoolchain_cfg.sh

Source sdktoolchain_cfg.sh in ~/tools/

```
export PATH=/home/sven/xy1030/Nuclei/linux_gcc/gcc/bin:$PATH
export TOOLCHAIN_DIR=/home/sven/xy1030/Nuclei/linux_gcc/gcc
export CROSS_COMPILE=${TOOLCHAIN_DIR}/bin/riscv-nuclei-linux-gnu
export LD_LIBRARY_PATH=${TOOLCHAIN_DIR}/sysroot:${TOOLCHAIN_DIR}/sysroot/
lib64/ld64:${TOOLCHAIN_DIR}/sysroot/lib:${TOOLCHAIN_DIR}/sysroot/usr:
${TOOLCHAIN_DIR}/sysroot/usr/lib64/ld64
```

2) Compile

In the sdk directory:

make clean

make SDK_MODE=2 2>1.txt

3) Generate files

sdk/out/lib/libsdk.so

sdk/out/bin/fhcli

4.3.2. External CPU

1) Set environment variables

```
export PATH=
```

```
export TOOLCHAIN_DIR=
```

```
export CROSS_COMPILE=
```

```
export LD_LIBRARY_PATH=
```

The above path uses the path of your own compiled tool chain

2) Modify the compilation tool chain

Open the make.linux file in the make folder of the sdk directory.

```
26
27 ${info "ffffff RUN_MODE is ${RUN_MODE)}"
28 ifeq ($(R_SDK_MODE),B10)
29 ${info "vvvvvvvvv")
30 CC := ${TOOLCHAIN_DIR}/arm-wrs-linux-gnueabi-gcc
31 AR := ${TOOLCHAIN_DIR}/arm-wrs-linux-gnueabi-ar
32 RANLIB := ${TOOLCHAIN_DIR}/arm-wrs-linux-gnueabi-ranlib
33 LD:= ${TOOLCHAIN_DIR}/arm-wrs-linux-gnueabi-ld
34 CFLAGS += -march=armv7-a -mfloat-abi=hard -mfpu=neon -marm -mthumb-interwork -mtune=cortex-a7
--sysroot=/opt/windriver/wrlinux/8.0-fsl-ls10xx/sysroots/cortexa7hf-vfp-neon-wrs-linux-gnueabi
-std=c99 -Wextra -Wbad-function-cast -Wcast-align -Wcast-qual -Wchar-subscripts -Wmissing-proto
types -Wnested-externs -Wpointer-arith -Wredundant-decls -Wstrict-prototypes -Wparentheses -Wsw
itch -Wswitch-default -Wunused -Wuninitialized -Wunused-but-set-variable -Wno-unused-parameter
-Wno-missing-field-initializers -Wno-sign-compare -Wshadow -Wno-inline -MMD -MP -g -O2
35 endif
```

Change the toolchain highlighted in yellow to your own compilation toolchain. Line 34 CFLAGS may need to be adjusted depending on the toolchain.

3) Adapt access bus

Adapt external CPU to access bus, modify iic_fpga_func.c, mii_fpga_func.c in the corresponding src/core/ral/directory, spi_fpga_func.c, at least one access channel must be enabled.

4) Compile

In the sdk directory:

```
make clean
```

```
make SDK_MODE=1 2>1.txt
```

5) Generate files

```
sdk/out/lib/libsdk.so
```

```
sdk/out/bin/fhcli
```


4.4. Liunx SDK Compile & Flash

The following defaults to versions no lower than linux_sdk_v1_0_3. Versions lower than linux_sdk_v1_0_3 will be additionally commented.

4.4.1. Hardware flash requirements

The two SPIs are connected with NOR flash and NAND flash respectively.

nor flash

nor should be no less than 1M, used to place the boot loader uboot, device tree dtb, and uboot environment variable env.

nor If it is larger than 1M, it can also store the kernel kernel and the root file system rootfs (ramfs, jffs2).

nand flash

Nandflash is used to store the kernel and the root file system rootfs (ubifs).

The kernel and rootfs can be placed in nor or nand. You only need to modify the startup command bootcmd and load them from the corresponding place.

If NOR is large enough to store kernel and rootfs, NAND is not needed.

4.4.2. Partitioning

This linux sdk takes the hardware with 64MB norflash and 128MB nandflash as an example, and provides ramfs+jffs2, ramfs+ubifs, ubifs (default for demo board) to load the root file system.

ramfs+jffs2 only need nor flash, no need to use nandflash

ramfs+ubifs requires at least 1M of norflash and 40M of nandflash.

ubifs requires at least 1M of norflash and 64M of nandflash.

Users can modify the partition size according to actual needs. Please make sure that the partition information in linux_sdk/u-boot/include/configs/nuclei-hbird.h and linux_sdk/conf/nuclei_ux600fd.dts is correct.

At the same time, modify the relevant environment variables in linux_sdk/u-boot/include/configs/nuclei-hbird.h

Nor flash (64M) is partitioned as follows under uboot:

name	uboot	dtb	env	kernel	rootfs	rootfs_jffs2
Partition size (bytes)	896K	64K	64K	4M	7M	52M
Partition first address	0x0	0xE0000	0xF0000	0x100000	0x500000	0xc00000
effect	Freeloader device tree environment variables store kernel				Storing ramfs File system	Store and run jffs2 The root file system
uboot partition	freeloader			kernel_nor	ramfs_nor jffs2_nor	
Burning files	freeloader.elf / freeloader.bin			kernel.bin	initrd.bin	jffs2.img

Nand flash (128M) is partitioned as follows under uboot:

name	kernel	rootfs	rootfs_ubifs	reserved
Partition size (bytes):	4M	20M	16M / 40M	88M / 64M
Partition first address	0x0	0x400000	0x1800000	-
effect	Storing the kernel	Files storing ramfs system	Store and run ubifs The root file system	reserve
uboot partition	kernel_nand	ramfs_nand	ubifs_nand	-
Burning files	kernel.bin	initrd.bin	ubifs.img	-

4.4.3. Compilation preparation

Modify the toolchain path in ~/tools/toolchain_cfg.sh

Source toolchain_cfg.sh in ~/tools/

```
export PATH=$PATH:/home/sven/xy1030/Nuclei/gcc/bin:/home/sven/xy1030/Nuclei/
openocd/bin:/home/sven/xy1030/Nuclei/linux_gcc/gcc/bin
export TOOLCHAIN_DIR=/home/sven/xy1030/Nuclei/linux_gcc/gcc
export CROSS_COMPILE=${TOOLCHAIN_DIR}/bin/riscv-nuclei-linux-gnu
export LD_LIBRARY_PATH=${TOOLCHAIN_DIR}/sysroot:${TOOLCHAIN_DIR}/sysroot/
lib64/ld64:${TOOLCHAIN_DIR}/sysroot/lib:${TOOLCHAIN_DIR}/sysroot/usr:
${TOOLCHAIN_DIR}/sysroot/usr/lib64/ld64
```

Confirm linux_sdk under ln -s freeloader_rootfs freeloader

The Linux system development kit `linux_sdk.tgz` provides the source code and compilation environment for compiling the Linux system. System, you can unzip and enter the `linux_sdk` directory, and then execute the corresponding commands.

```
$make help //View the command

- buildroot_initramfs-menuconfig : run menuconfig for buildroot, configuration will be saved into conf/
- buildroot_initramfs_sysroot : generate rootfs directory using buildroot
- linux-menuconfig : run menuconfig for linux kernel, configuration will be saved into conf/
- buildroot_busybox-menuconfig : run menuconfig for busybox in buildroot
- uboot-menuconfig : run menuconfig for uboot
- initrd : generate initramfs cpio file
- bootimages : generate boot images for SDCard
- freeloader : generate freeloader(first stage loader) run in norflash
- freeloader4m : generate freeloader(first stage loader) 4MB version run in norflash
- upload_freeloader : upload freeloader into development board using openocd and gdb
- uboot : build uboot and generate uboot binary
- sim : run opensbi + linux payload in simulation using xl_spike
- clean : clean this full workspace
-cleanboot : clean generated boot images
-cleanlinux: clean linux workspace
-cleanbuildroot : clean buildroot workspace
-cleansysroot : clean buildroot sysroot files
-cleanuboot : clean u-boot workspace
- cleanfreeloader : clean freeloader generated objects
-cleanopensbi : clean opensbi workspace
- preboot : If you run sim target before, and want to change to bootimages target, run this to prepare environment
- presim : If you run bootimages target before, and want to change to sim target, run this to prepare environment
```

4.4.4. Burn freeloader.elf via JTAG

The `freeloader.elf` is burned through `jtag`. After the `freeloader.elf` is burned, it starts and runs to the `uboot` command line, and then burns the remaining files (since `freeloader.elf` already includes `fdt.dtb`, `fdt.dtb` does not need to be burned separately).

After ensuring that the device is powered off, turn the device DIP switch to download mode. Connect the test board to the computer via JTAG and power on the test board.

1> Linux version of `openocd` burns `freeloader.elf` (two methods, choose one)

If it is a Linux virtual machine, you can also see the devices shown in the figure below:



Open the terminal under `openocd_202212/openocd/bin`

```
sudo chmod 777 -R /dev/bus/usb //Set jtag executable permissions
./openocd -f openocd_nuclei.cfg
```

If the connection is successful, the following will be printed:

```
Info: Nuclei SPI controller version 0x00000001
Info: Found flash device 'micron mt25ql512' (ID 0x0020ba20)
cleared protection for sectors 0 through 1023 on flash bank 0
Info: Listening on port 6666 for tcl connections
Info: Listening on port 4444 for telnet connections
```

Keep the above terminal and open another terminal under `linux_sdk/`

```
source toolchain_cfg.sh
riscv-nuclei-elf-gdb
target remote localhost:3333
load ./freeloader_rootfs/freeloader.elf
```

2> Burn freeloader.elf with openocd for Windows (two methods, choose one)

Modify the paths of the first and second lines of `openocd.bat` to local paths, copy `openocd.bat` to the desktop, and double-click to run the first terminal to start openocd. If the connection is successful, the following will be printed:

```
Info: Nuclei SPI controller version 0x00000001
Info: Found flash device 'micron mt25ql512' (ID 0x0020ba20)
cleared protection for sectors 0 through 1023 on flash bank 0
Info: Listening on port 6666 for tcl connections
Info: Listening on port 4444 for telnet connections
```

Keep the above terminal and open another terminal under `linux_sdk/`

```
source toolchain_cfg.sh
riscv-nuclei-elf-gdb
target remote xx:xx:xx:xx:3333 //PC current IP address
load ./freeloader_rootfs/freeloader.elf
```

After the above two methods are successfully burned, it will be as shown below

```

Loading section .text, size 0xa0818 lma 0x20000000
Loading section .interp, size 0x21 lma 0x200a0818
Loading section .dynsym, size 0x18 lma 0x200a0840
Loading section .dynstr, size 0xb lma 0x200a0858
Loading section .hash, size 0x10 lma 0x200a0868
Loading section .gnu.hash, size 0x1c lma 0x200a0878
Loading section .dynamic, size 0x110 lma 0x200a0898
Loading section .got, size 0x8 lma 0x200a09a8
Start address 0x20000000, load size 657824
Transfer rate: 29 KB/sec, 13704 bytes/write.

```

4.4.5. ubifs version compilation & burning (default compilation and burning method for demo board)

Compile uboot, dtb, and env into a freeloader.elf, download it to NOR flash through jtag, and then use uboot to download kernel, ubifs is burned into nand flash. The system starts in ubifs mode, ubifs can be read and written, and the changes made will not be lost when the power is off.

This method is for the application scenario where the device has both NOR flash and NAND flash. ubifs saves the files that need to be modified.

1> Compile partition

Modify linux_sdk/u-boot/include/configs/nuclei-hbird.h: the code has been written, select for ubifs boot

```

//for ubifs boot

#define MTDPARTS_DEFAULT
"mtdparts=spi_nor:896K(uboot),64K(dtb),64K(env);spi-nand:4M(kernel_nand),20M(ramfs_nand),40M(ubifs_nand)"

```

Modify linux_sdk/conf/nuclei_ux600fd.dts: the code has been written, select for ubifs boot

```

//for ubifs boot

partition@01800000 {

    reg = <0x01800000 0x02800000>;

    label = "ubifs_nand";

};

```

2> Compile freeloader

```

make clean

make freeloader

```

Generate freeloader.elf, freeloader.bin, kernel.bin, initrd.bin in linux_sdk/freeloader_rootfs

3> Make ubifs.img

```

cd ~/linux_sdk/kernel/led_164                //Modify TOOLCHAIN_DIR in Makefile

make                                           //Generate led_164.ko

```

```
cd ~/linux_sdk/kernel/xy1000_net //Modify TOOLCHAIN_DIR in Makefile
make                               //Generate xy1000_net.ko
```

Copy led_164.ko, xy1000_net.ko and config.fhme, libsdk.so, fhcli compiled by business sdk to linux_sdk/driver\&lib/

Copy files to rootfs: Execute ./driver\&lib/cp.sh ubifs under linux_sdk

To add other files to ubifs, place them in the appropriate path under work/buildroot_initramfs_sysroot/, then Remake.

The mkfs.ubifs command is integrated into cp.sh. Please note that the -c parameter should be consistent with the actual one, otherwise the partition capacity will be incorrect.

4>Download

Build openocd and gdb environment, download freeloader.elf file to nor flash, refer to section 4.4.4.

5>Run uboot to burn (update) kernel and rootfs_ubifs

Connect any electrical port and serial port of the test board to the network port and USB port of the computer. After power on, press "asd" to enter uboot. When running, you need to burn kernel and ubifs to nand flash.

The uboot has integrated related burning instructions. Put the file with the specified name into the tftpboot folder of the server (you can also use Software tftpd32).

```
run updateos_nand                //Burn kernel.bin to kernel_nand partition of nand flash
run updateubifs_boot             //Burn ubifs.img to the ubifs_nand partition of nand flash

setenv bootcmd run bootcmd_ubifs_boot //Set the boot command

saveenv                          //Save environment variables

boot                             //Start Linux
```

***If the linux_sdk version is lower than linux_sdk_v1_0_3, use the following command

```
run updateos                    //Burn kernel.bin to kernel_nand partition of nand flash
run updateubifs                 //Burn ubifs.img to the ubifs_nand partition of nand flash

setenv bootcmd run bootcmd_ubifs //Set the startup command

saveenv                          //Save environment variables

boot                             //Start Linux
```

4.4.6. Compile and flash ramfs + jffs2 version

Compile uboot, dtb, and env into freeloader.elf (freeloader.bin), and download it to NOR flash through jtag.

Use uboot to burn kernel, ramfs, and jffs2 to nor flash. The system starts in ramfs mode, and then mounts jffs2 after startup.

Read-only, the changes made will be lost if the power is off; jffs2 is readable and writable, the changes made will not be lost if the power is off.

This method is for the application scenario where the device has only one nor flash and no nand flash. jffs2 saves files that need to be modified. Since the more files jffs2 contains, the longer the mounting time will be, which will affect the user experience, so it is used with ramfs. ramfs saves files that do not need to be modified, which can quickly start the system and reduce the size of jffs2 and its mounting time.

1> Compile partition

Modify linux_sdk/u-boot/include/configs/nuclei-hbird.h: the code has been written, select for ramfs + mount jffs2

```
//for ramfs + mount jffs2

define MTDPARTS_DEFAULT
"mtdparts=spi_nor:896K(uboot),64K(dtb),64K(env),4M(kernel_nor),20M(ramfs_nor),10M(jffs2_nor)"
```

2> Compile freeloader

```
make clean

make freeloader
```

Generate freeloader.elf, freeloader.bin, kernel.bin, initrd.bin in linux_sdk/freeloader_rootfs

3> Make jffs2.img

```
cd ~/linux_sdk/kernel/led_164 //Modify TOOLCHAIN_DIR in Makefile

make //Generate led_164.ko

cd ~/linux_sdk/kernel/xy1000_net //Modify TOOLCHAIN_DIR in Makefile

make //Generate xy1000_net.ko
```

Copy led_164.ko, xy1000_net.ko and config.fhme, libsdk.so, fhcli compiled by business sdk to linux_sdk/driver\&lib/

Copy files to rootfs: Execute ./driver\&lib/cp.sh ramfs under linux_sdk

ramfs is the initrd.bin file compiled in the freeloader folder

To add other files to ramfs, place the files in the corresponding path under work/buildroot_initramfs_sysroot/ and then rebuild.

Package jffs2.img: mkfs.jffs2 -r rootfs_jffs2 -o jffs2.img -l -n -s 0x100 -e 0x10000 --pad=0x00A00000

//Note: The parameters of -e --pad must be consistent with the actual ones, otherwise the mount will result in an error.

4>Download

Build openocd and gdb environment, download freeloader.elf file to nor flash, refer to section 4.4.4.

5>Run uboot to burn (update) kernel, rootfs, rootfs_jffs2

Connect any electrical port and serial port of the test board to the network port and USB port of the computer. After power on, press "asd" to enter uboot. When running, you need to burn kernel, ramfs, and jffs2 to nor flash.

The uboot has integrated related burning instructions. Put the file with the specified name into the tftpboot folder of the server (you can also use Software tftpd32).

```
run updateos_nor                //Burn kernel.bin to kernel_nor partition of nor flash
run updateramfs_nor             //Burn initrd.bin to ramfs_nor partition of nor flash
run updatejffs2_nor             //Burn jffs2.img to the jffs2_nor partition of the nor flash
setenv bootcmd run bootcmd_nor  //Set the startup command
saveenv                         //Save environment variables
boot                           //Start Linux
```

After startup, the system will automatically mount jffs2 through the script /etc/init.d/S30initconfig.sh.

4.4.7. Compile and flash ramfs + ubifs version

Compile uboot, dtb, and env into a freeloader.elf file and download it to NOR flash through jtag. Then use uboot to download kernel, ramfs, ubifs are burned to nand flash. The system starts in ramfs mode, and then mounts ubifs after startup. ramfs is read-only, so it is more

Changes will be lost if the power is off; ubifs is readable and writable, and the changes made will not be lost if the power is off.

This method is for the application scenario where the device has both nor flash and nand flash. ubifs saves the files that need to be modified. ramfs Save the files that do not need to be modified.

1> Compile partition

Modify linux_sdk/u-boot/include/configs/nuclei-hbird.h: the code has been written, select for ramfs + mount ubifs

```
//for ramfs + mount ubifs
#define MTDPARTS_DEFAULT
"mtdparts=spi_nor:896K(uboot),64K(dtb),64K(env);spi-nand:4M(kernel_nand),20M(ramfs_nand),16M(ubifs_nand)"
```

Modify linux_sdk/conf/nuclei_ux600fd.dts: The code has been written, select for ramfs + mount ubifs

```
//for ramfs + mount ubifs
partition@01800000 {
    reg = <0x01800000 0x01000000>;
    label = "ubifs_nand";
};
```

2> Compile freeloader

```
make clean
make freeloader
```

Generate freeloader.elf, freeloader.bin, kernel.bin in linux_sdk/freeloader_rootfs

3> Make ubifs.img


```
cd ~/linux_sdk/kernel/led_164           //Modify TOOLCHAIN_DIR in Makefile
make                                     //Generate led_164.ko

cd ~/linux_sdk/kernel/xy1000_net //Modify TOOLCHAIN_DIR in Makefile
make                                     //Generate xy1000_net.ko
```

Copy led_164.ko, xy1000_net.ko and config.fhmc, libsdk.so, fhcli compiled by business sdk to
linux_sdk/driver\&lib/

Copy files to rootfs: Execute ./driver\&lib/cp.sh ramfs under linux_sdk

ramfs is the initrd.bin file compiled in the freeloader folder

To add other files to ramfs, place them in the corresponding path under work/buildroot_initramfs_sysroot/ and then
Then remade.

Pack ubifs.img: mkfs.ubifs -F -m 2048 -e 124KiB -c 128 -r rootfs_ubifs ubifs.img

//Note: The -c parameter should be consistent with the actual value, otherwise the partition capacity will be incorrect.

4>Download

Build openocd and gdb environment, download freeloader.elf file to nor flash, refer to section 4.4.4.

5>Run uboot to burn (update) kernel, rootfs, rootfs_ubifs

Connect any electrical port and serial port of the test board to the network port and USB port of the computer. After power on, press "asd" to enter uboot.
When running, you need to burn kernel, ramfs, and ubifs to nand flash.

The uboot has integrated related burning instructions. Put the file with the specified name into the tftpboot folder of the server (you can also use
Software tftpd32).

```
run updateos_nand           //Burn kernel.bin to kernel_nand partition of nand flash
run updateramfs_nand        //Burn initrd.bin to the ramfs_nand partition of nand flash
run updateubifs_nand        //Burn ubifs.img to the ubifs_nand partition of nand flash
setenv bootcmd run bootcmd_nand //Set the startup command
saveenv                     //Save environment variables
boot                        //Start Linux
```

After startup, the system will automatically mount jffs2 through the script /etc/init.d/S30initconfig.sh

4.4.8. Commonly used commands under uboot

```
run updatedtb           //Burn fdt.dtb to the dtb partition of nor flash
```

```
run updatefreeloader //burn freeloader.bin to uboot and dtb partitions of nor flash, and erase env partition at the same time
```

5. Configure software usage

After the system starts, there is fhcli in the path /root. fhcli is the configuration software, which will call the functions provided by libsdk.so. Before starting fhcli, check whether the default configuration under /root/config.fhme is correct, especially the interface for accessing registers and the setting of the exchange mode. When Fhcli starts, it will perform initialization according to the configuration in config.fhme.

fhcli is a command line operation, and various functions can be completed by entering commands. After entering fhcli, the following figure is shown

```
FSLcli.0> [/root]
FSLcli.0> [/root] reglist TOP_CFG

mems below:
TOP_CFG_REG_AXI_CP_CFG
TOP_CFG_REG_CHIP_INFO_REG
TOP_CFG_REG_CHIP_INTR
TOP_CFG_REG_EFUSE_CSR
TOP_CFG_REG_INVISIBLE_REG
TOP_CFG_REG_PCS_SWITCH_MODE_CFG
TOP_CFG_REG_PLL_PD_CTRL
TOP_CFG_REG_PTP_PLL_DIVISOR
TOP_CFG_REG_RESET_GLOBAL
TOP_CFG_REG_RESET_MISC
TOP_CFG_REG_RESET_PCS_ADPT
TOP_CFG_REG_RESET_SERDES
TOP_CFG_REG_RESET_SERDES_PCS
TOP_CFG_REG_RESET_SWITCH_EPHY
TOP_CFG_REG_RGMII_ALM_CSR
TOP_CFG_REG_RGMII_CSR
TOP_CFG_REG_RGMII_DUPLEX
TOP_CFG_REG_SD1G_PLL_DIVISOR
TOP_CFG_REG_SDXG_PLL_DIVISOR
TOP_CFG_REG_SOC_PLL_DIVISOR
TOP_CFG_REG_SYNC_ETH_CFG
TOP_CFG_REG_SYS_PLL_DIVISOR
TOP_CFG_REG_TDC_CFG
TOP_CFG_REG_TI_MODE_CFG
TOP_CFG_REG_WORK_MODE_CFG
FSLcli.0> [/root] getreg TOP_CFG_REG_WORK_MODE_CFG
read addr 0x0
TOP_CFG_REG_WORK_MODE_CFG[0x0]=0x60fc2c: <SDXG_LED_MODE=0,SERIAL_CLK_EN=1,SERIAL_DATA_EN=1,CLOCK_CYCLE=0,BURST_CYCLE=0,BLINK_RATE=3,LED_ACTIVE_LOW=1,LED_MODE=7,REUSE_IND=0,SYNC_EN=0,SDXG1_EN=1,SDXG0_EN=3,GEPHY1_0FF=0,GEPHY0_0FF=0>
FSLcli.0> [/root] █
```

Enter fhcli:

```
./fhcli
```

Common commands:

1. Query the register name (enter an uppercase string to output all register names containing the string)

```
reglist [string]
```

```
reglist I_NET
```

2. Query table item name (enter an uppercase string to output all table item names containing the string)

```
memlist [string]
```

```
memlist I_NET
```

3. Read register

```
getreg [register name]
```

```
getreg I_NET_DEF_VLAN_CTL
```

4. Write register

```
modreg [register name] [domain name 1]=val1 < [domain name 2]=val2> ...
```

```
modreg I_NET_DEF_VLAN_CTL PORT_BMP=0
```

5. Read table items

```
dump [table item name] <[index]>
```

```
modify I_NET_PORT_SRM 0
```

6. Write table entries

```
modify [table item name] [index] 1 [domain name 1]=val1 <[domain name 2]=val2> ...
```

```
modify I_NET_PORT_SRM 0 1 STP_CHK_EN=0 BRG_EN=0
```

7. Configure the switch chip working mode

```
modeswitch switch mode=[mode] //mode is 1, 2, 3, ...
```

8. View the access interface of the switch chip register

```
ext_intf get
```

9. Set the access interface of the switch chip register

```
ext_intf set mode=3 addr=0x5c
```

6. SDK Usage

For details, please refer to "FSL91030(M) Chip SDK Interface Document_V1.4.pdf", "FSL91030(M) Chip SDK API Call Instructions Document" _V1.4.pdf", "FSL91030(M) chip SDK command line configuration business example_V1.32.pdf"

7. Version Record

Modification time	version number	Modification Record	Modification Record
2021.10.11	V1.0	initial version.	Shi Wei
2022.12.01	V1.4	Content optimization.	Shi Wei
2023.02.16	V1.5	Supplement the test board usage and sdk compilation; update the linux_sdk burning.	Shi Wei
2023.05.21	V1.6	Updates and Errata	Shi Wei