小眼睛科技

提供一站式 FPGA&嵌入式解决方案

Kosmo 之 PU 使用指导手册

Kosmo 系列（PG2K100-6IMBG400）

开发板实验教程

紫光同创 Kosmo 系列开发平台

文档版本修订记录

版本号	发布日期	修订记录
V1.0	2026/5/20	初始版本

公司名称：深圳市小眼睛科技有限公司

地址：深圳市宝安区西乡街道 F518 时尚创意园

官方网址：www.meyesemi.com

官方淘宝店铺：小眼睛半导体

B 站：小眼睛半导体（视频教程免费学）

* 加入 FPGA 开发者技术交流与 5000+FPGA 开发者实时沟通

QQ2 群： 442106123QQ3 群：882634519)

*配套资料下载、技术答疑请登录逻辑矩阵技术论坛

逻辑矩阵技术论坛欢迎各位发烧友加入
让我们共建开源生态，持续赋能行业发展
https://www.szlogicmatrix.com/

*扫码注册开源技术论坛

*扫一扫关注官微

* 官方旗舰店

目录

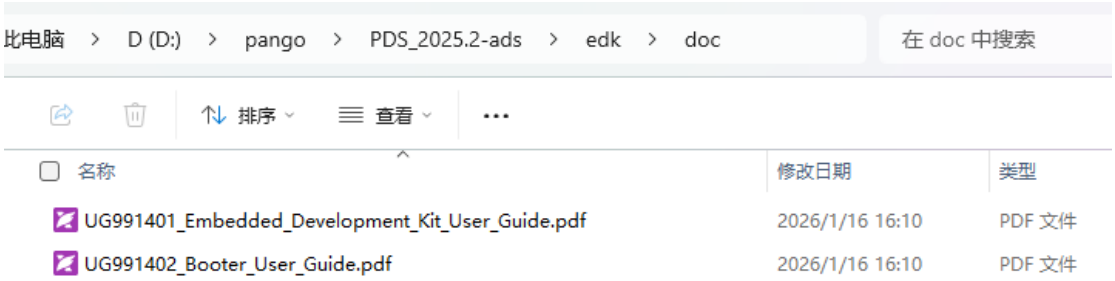
1.EDK 基本介绍

EDK 是一款基于 Eclipse 的软件开发工具，旨在为使用 Pango Kosmo 系列器件的开发者提供一个便捷的开发环境。 EDK 支持 C/C++语言和汇编语言的编写，还集成了许多常用的工具，如镜像生成工具，编译器，烧录器和调试器等，使开发者可以方便地进行全面的软件开发，调试和验证。

1.1. 软件环境

软件版本：PDS_2025.2-ads

安装好 PDS 后，在安装目录有两个文件，用户可先行阅读，里面主要是讲了 EDK 工具的使用步骤以及合成工具 Booter 的使用，如下图所示：



1.2.实验手册说明

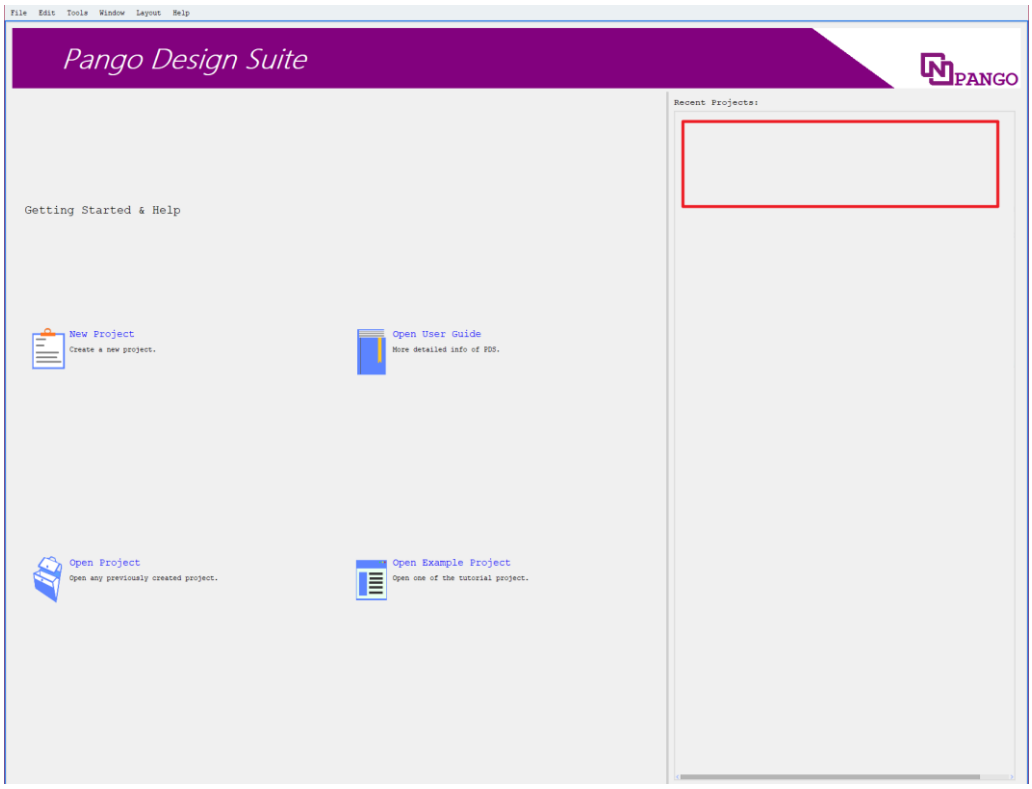
实验手册主要介绍通过 PDS 配置 Kosmo 核、PU 端外围接口应用，例如：MIO、EMIO、定时器等、以及添加一些 PA 端的外设等基本操作流程。希望能够通过这些实验能够进一步了解 Kosmo 系列器件的使用，为自己应用开发打下基础，更进阶更详细的使用方法请参阅安装 EDK 路径下的《UG991401_Embedded_Development_Kit_User_Guide》以及《UG991402_Booter_User_Guide》。

1.3.缩略语清单

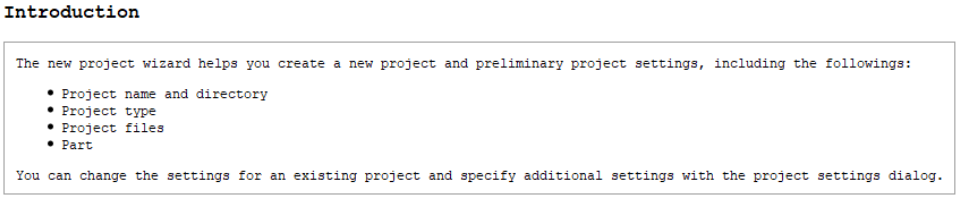
缩略语	英文全拼	中文解释
BSP	Board Support Package	板级支持包
HDD	Hardware Definition Description file	硬件定义描述
PSS	Processor Software Specification file	微处理器软件说明
UDI	User Design Interface	用户设计接口
HPC	Hardware Platform Configuration file	硬件平台配置文件
CFG	Configuration file	驱动的配置文件
OpenOCD	Open On-Chip Debugger	开源的芯片调试器

2.Hardware 工程创建

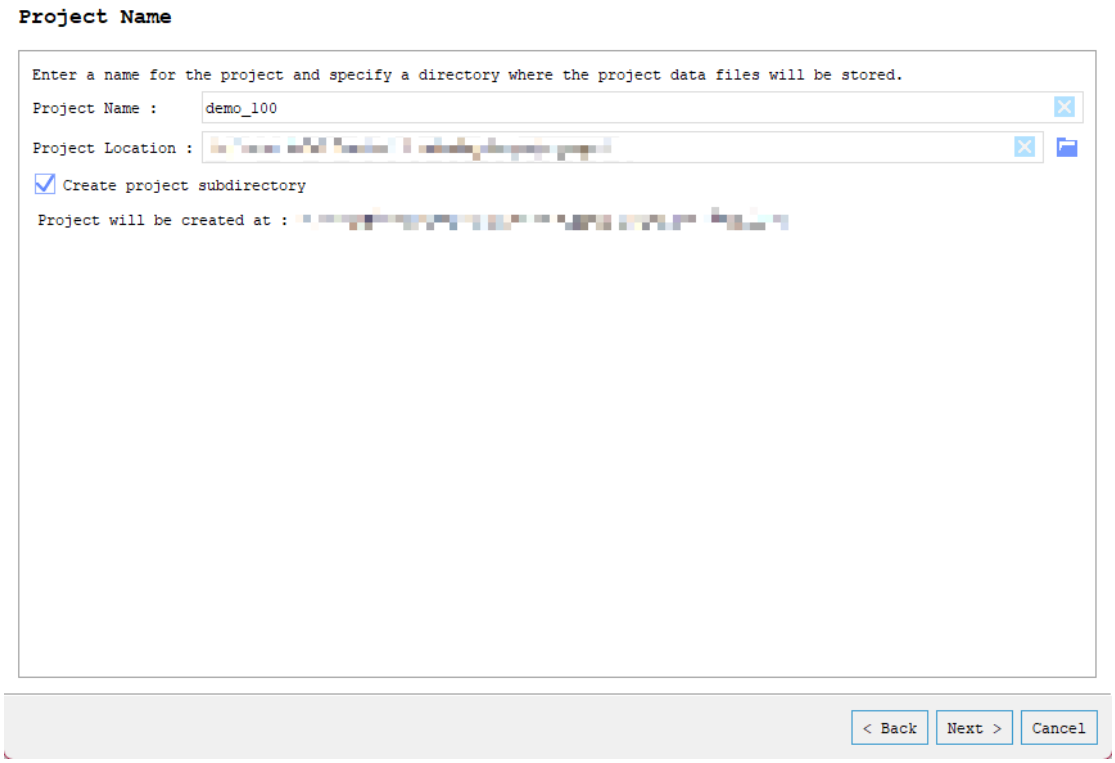
启动 PDS 软件工具，启动后的工作界面如下，右侧会显示已经创建的工程，双击可以直接打开该工程。若未创建过工程，下图红色框部分不会显示工程信息。



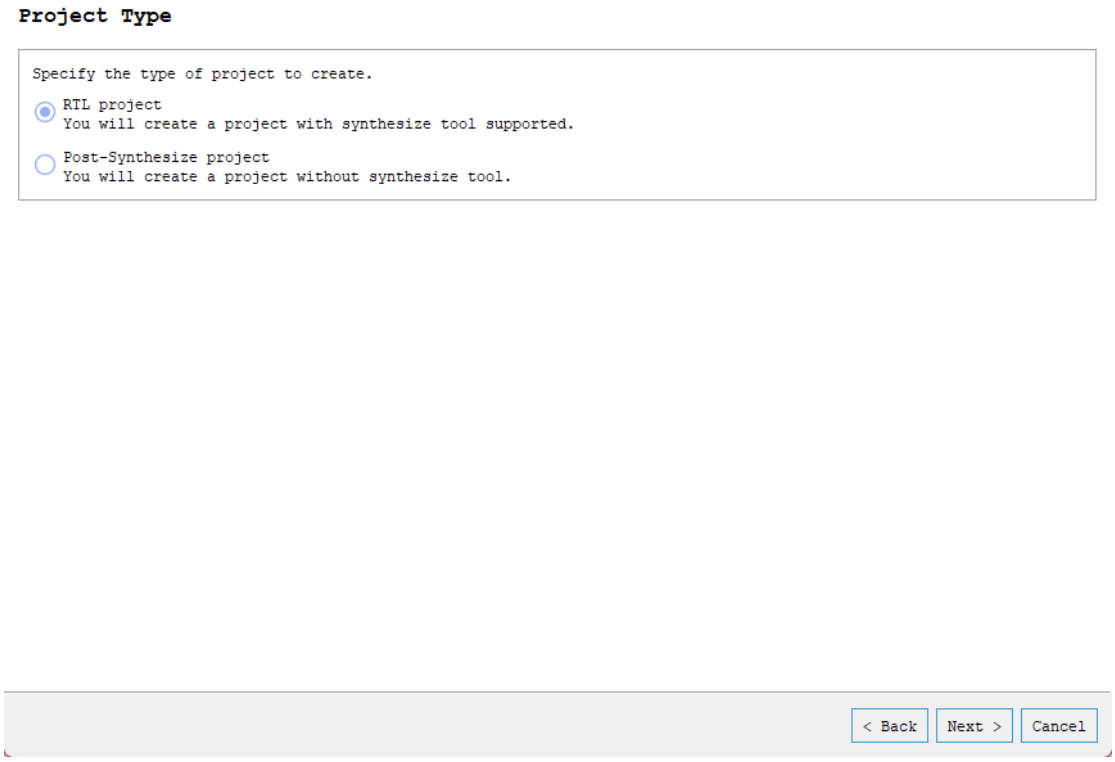
创建项目，依次点击菜单[File-> New Project]，界面显示如下：



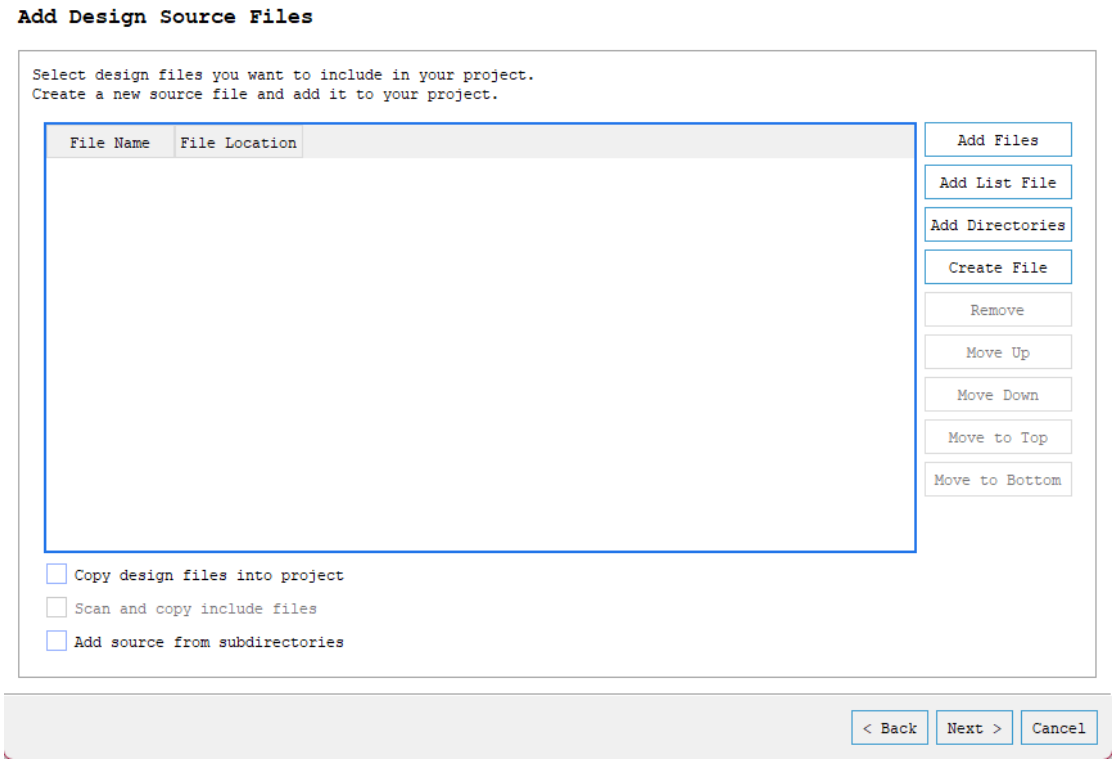
点击上图的“Next”按钮， 进入下图界面。



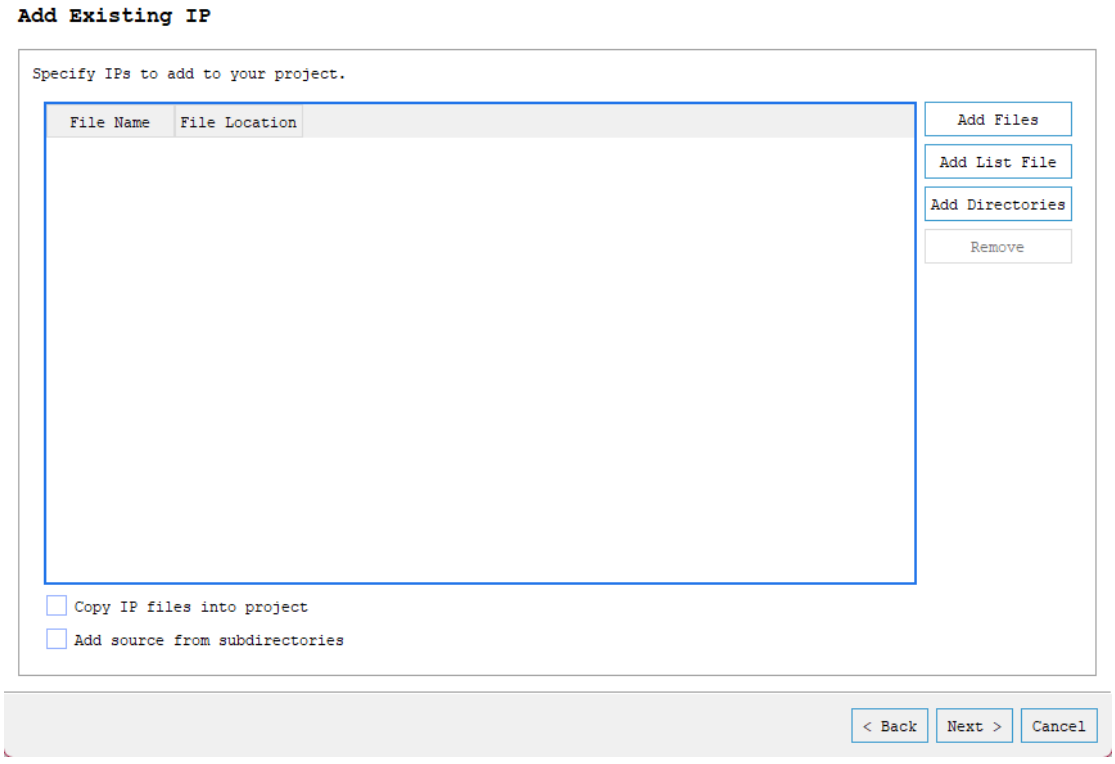
输入上图中的工程名并点击“Next”按钮，进入下图界面。



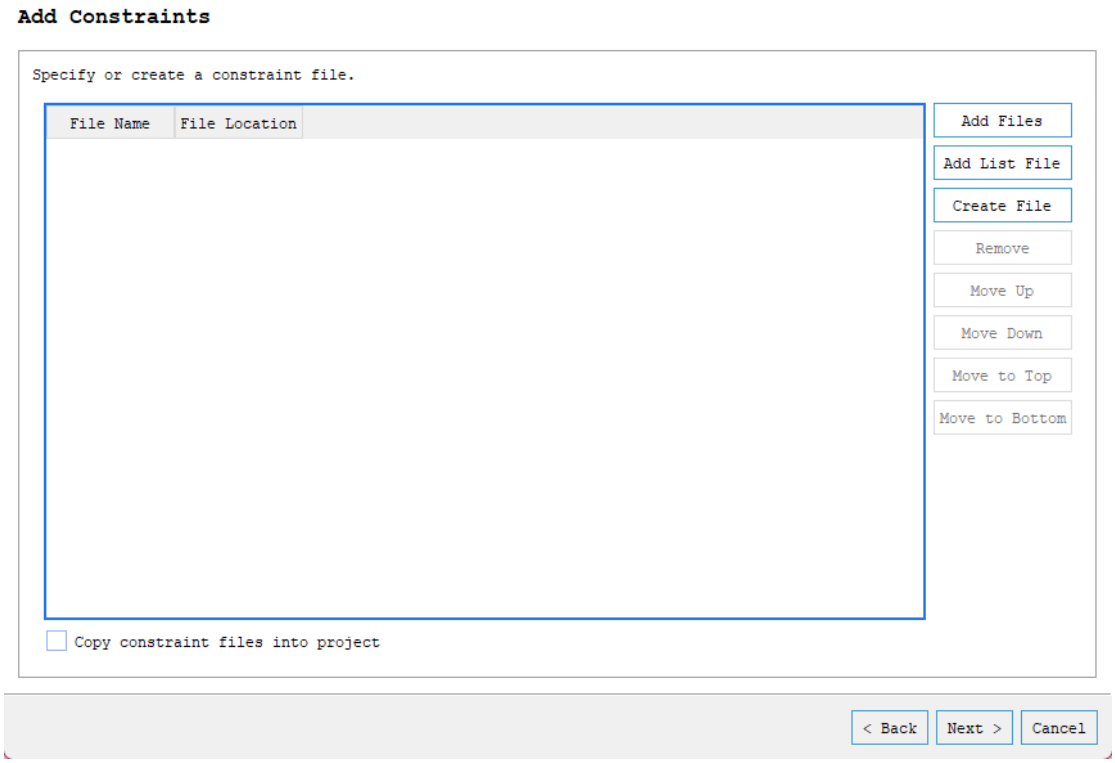
点击上图的“Next”按钮，进入下图界面。



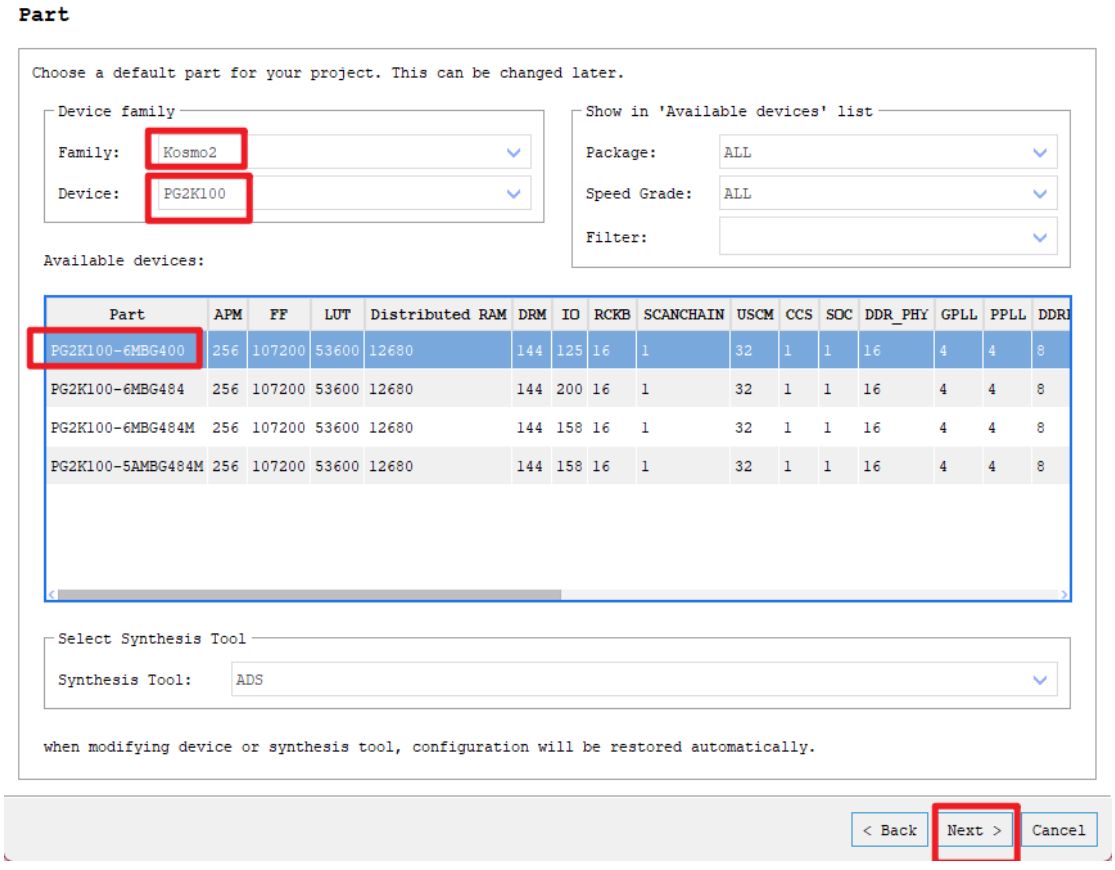
点击上图的“Next”按钮， 进入下图界面。



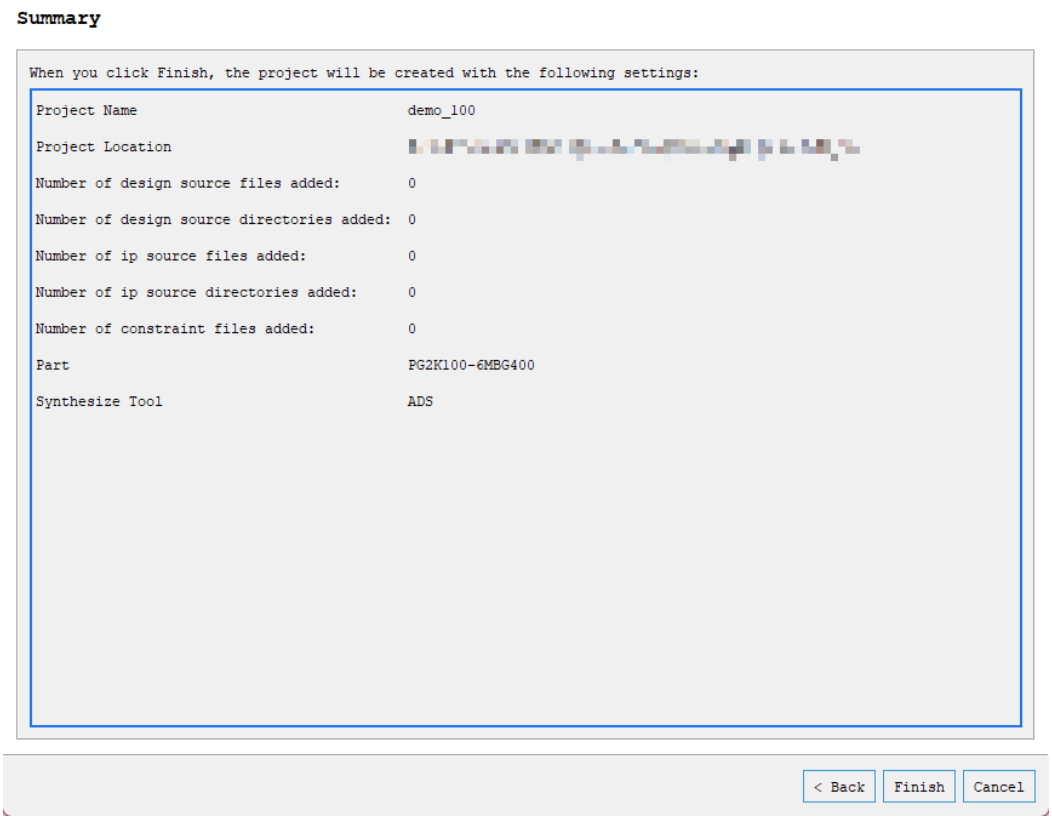
点击上图的“Next”按钮， 进入下图界面。



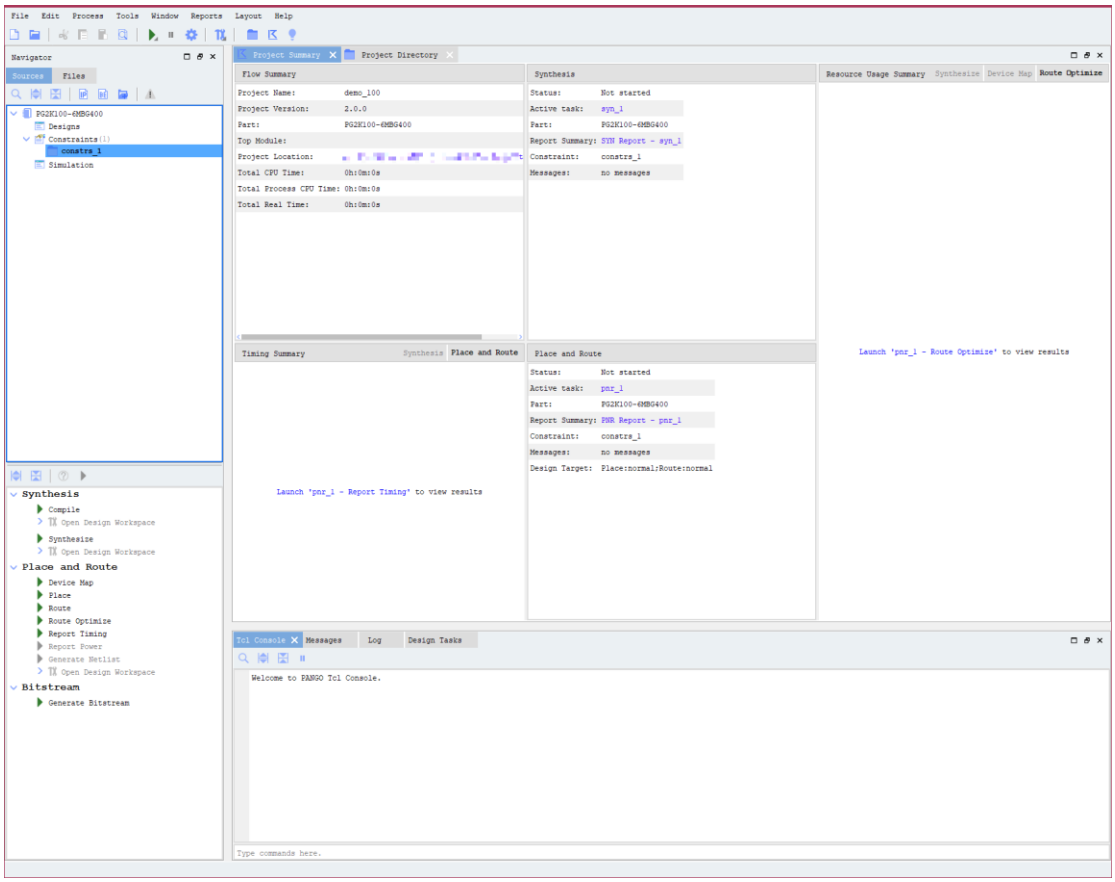
点击上图的“Next”按钮， 进入下图界面。 请将界面修改为下图所示。



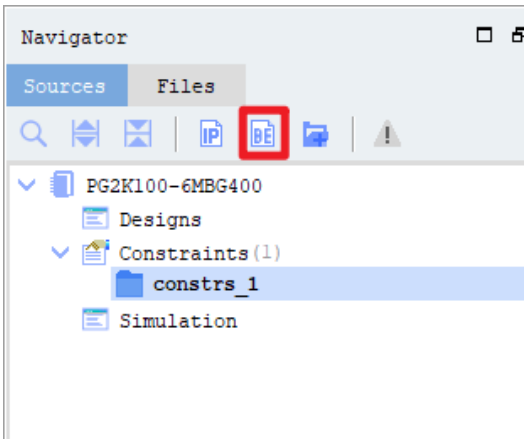
点击上图的“Next”按钮， 进入下图界面。



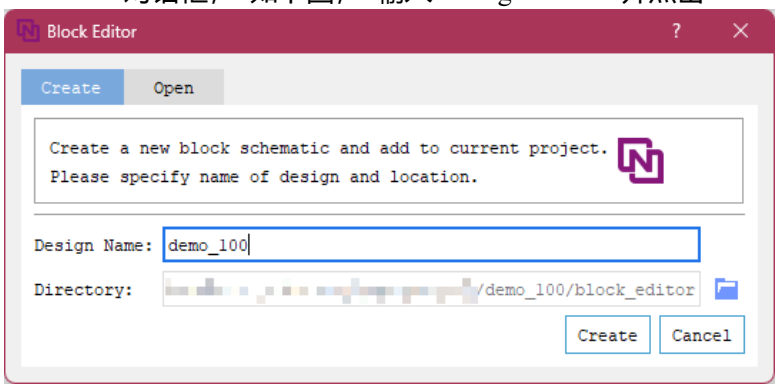
点击上图中的“Finish”按钮。 进入主工作界面， 如下图所示。



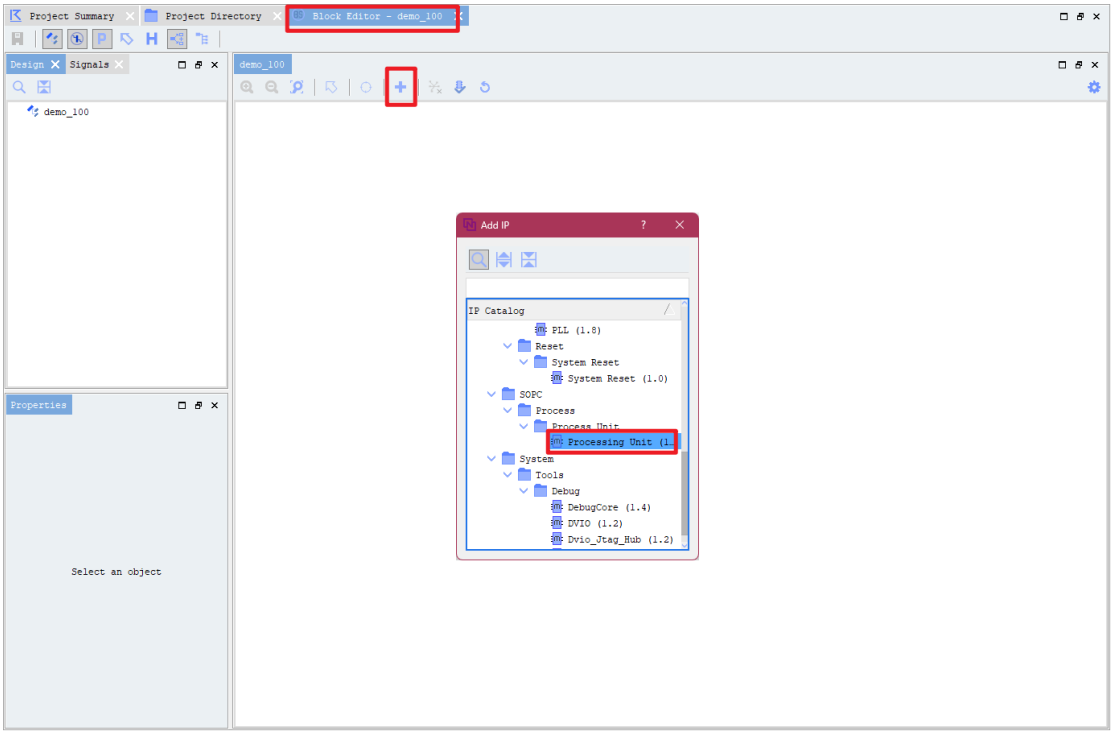
点击下图中的 Navigator/Sources 中的 BE 图标， 如下图中红色圈起的图标。

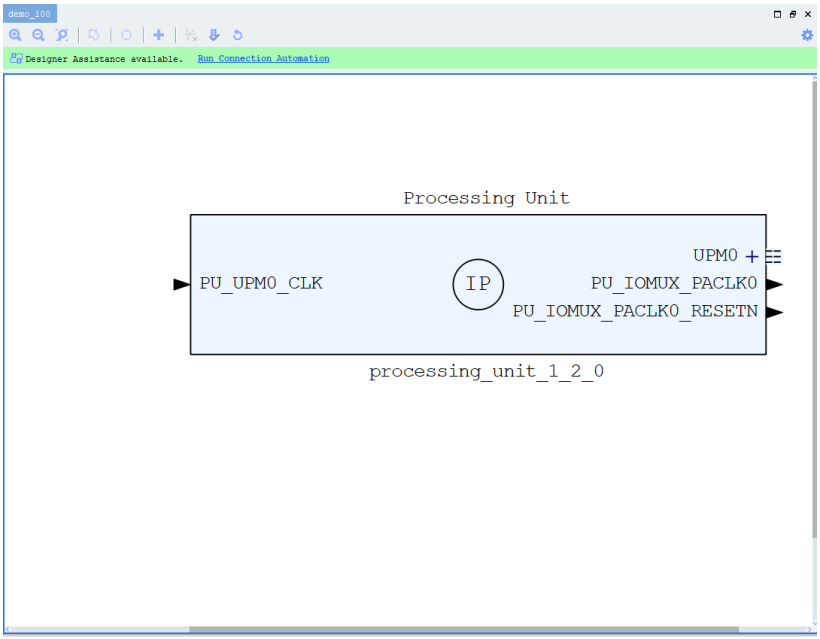


弹出“Block Editor”对话框， 如下图， 输入 Design Name 并点击“Create”按钮。

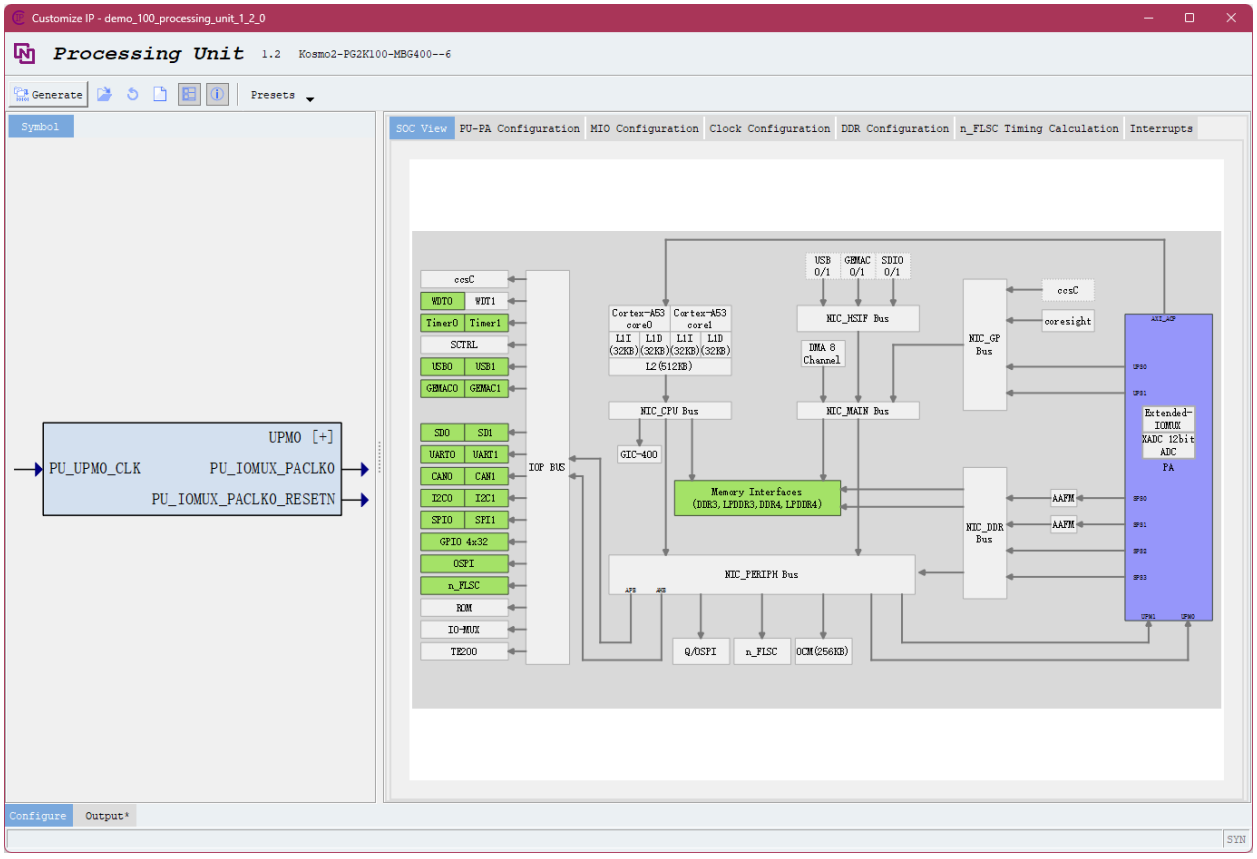


弹出“Block Design”界面， 如下图， 点击下图红框中的“+”添加 IP， 两者任选其一。弹出 “Add IP” 对话框， 如下图， 按照图中所示， 选择 SOPC 的 IP， 然后 “双击” ， 界面生成 IP。

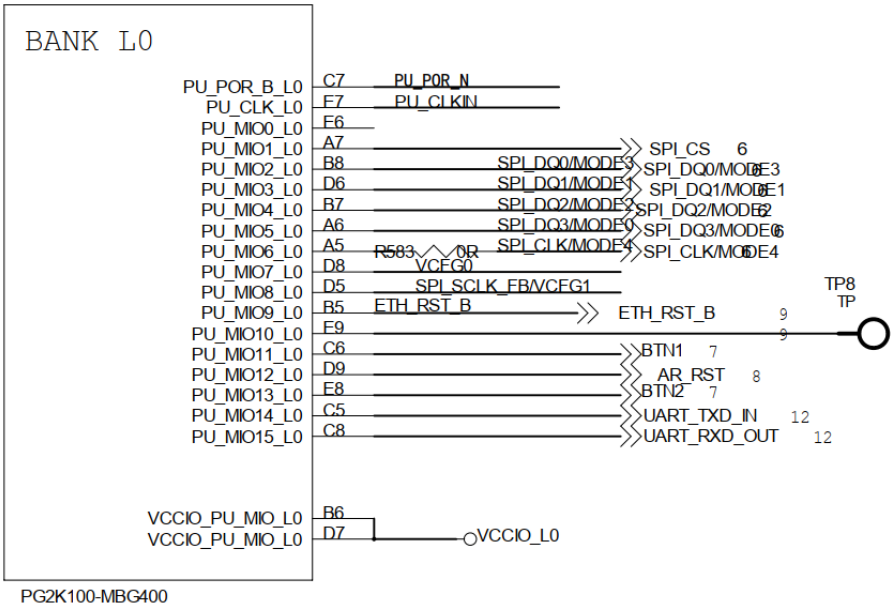


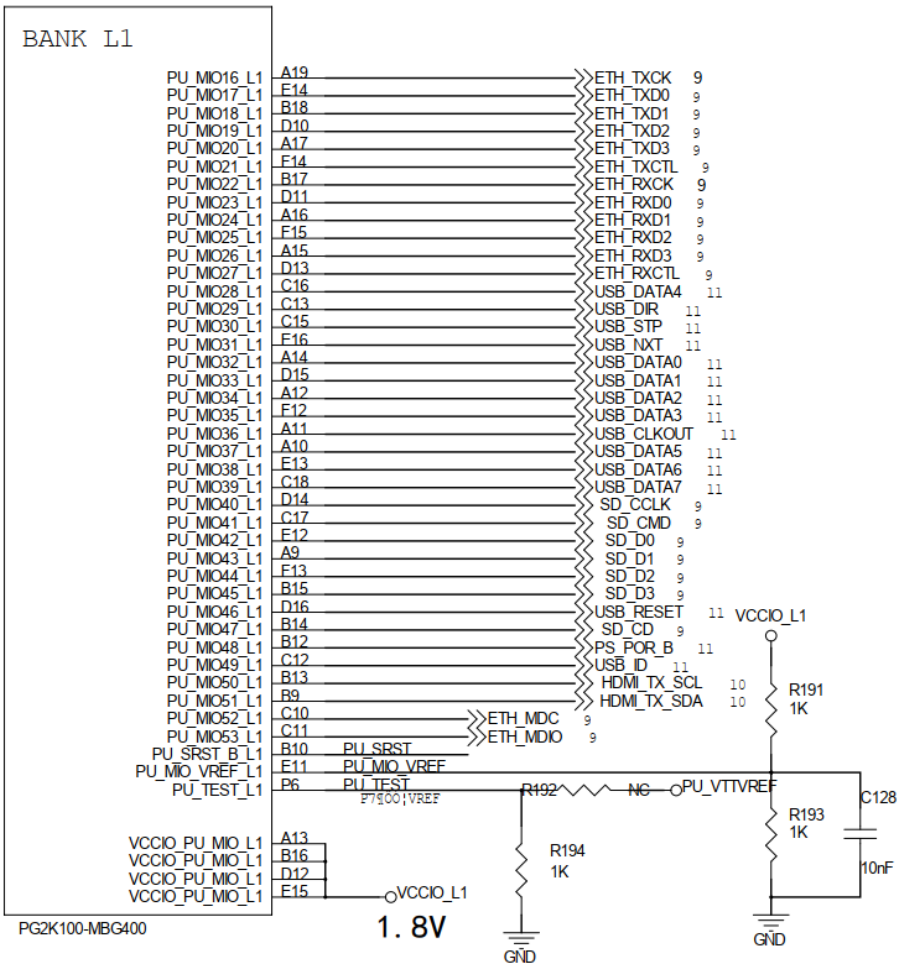


双击上图所选择的 IP， 将启动 IP 定制界面， 如下图所示。

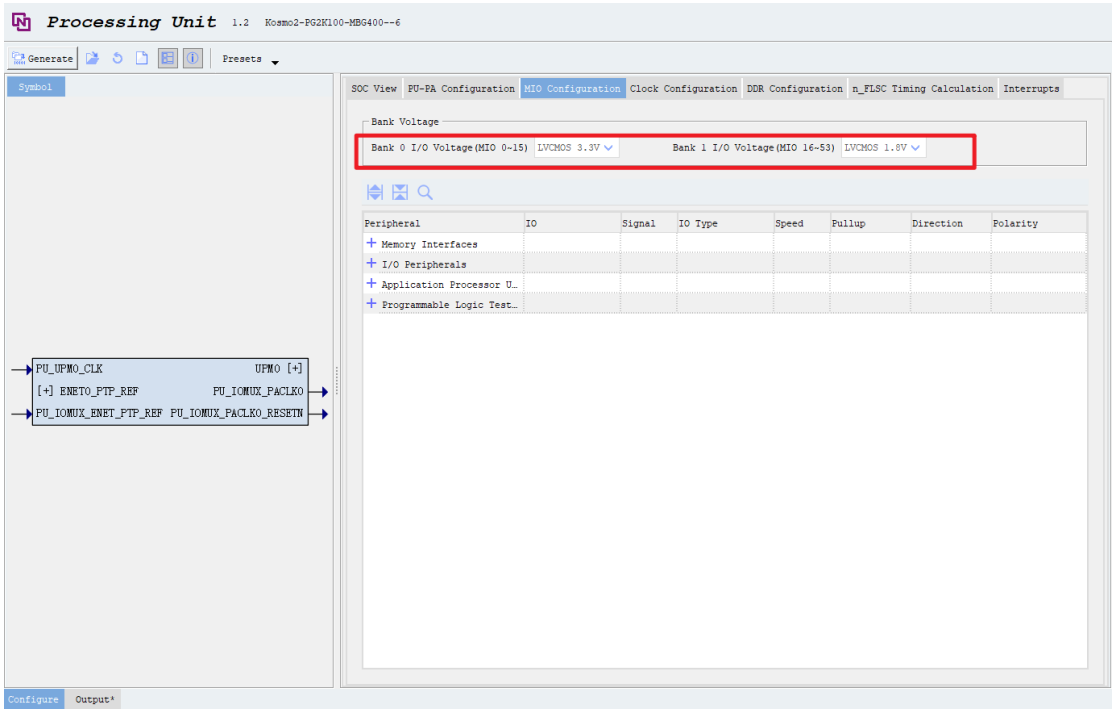


PU 端外设原理图， 以下分别为 BANK L0、 BANK L1

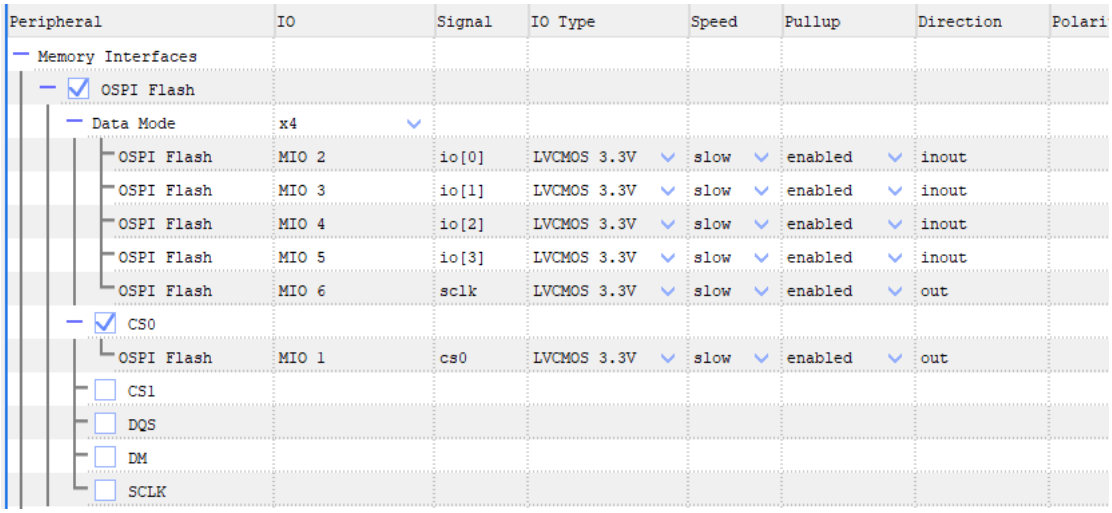




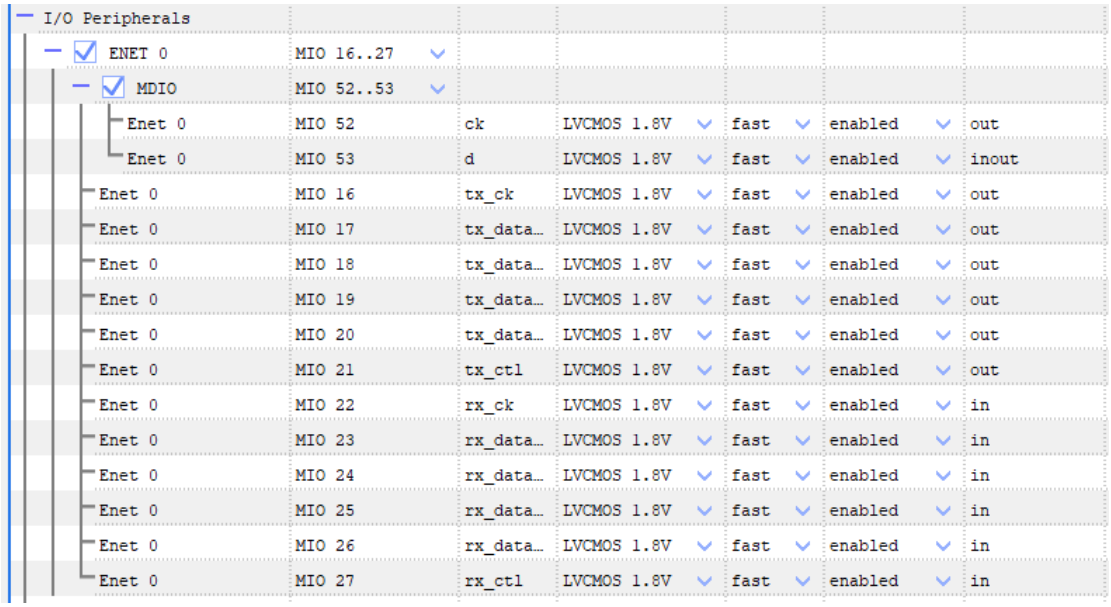
点击 MIO Configuration 选项卡，PU 端 MIO 分为两个 Bank，对应原理图 BANK L0 和 BANK L1，电压均选择 3.3V 和 1.8V，如下图所示进行配置；



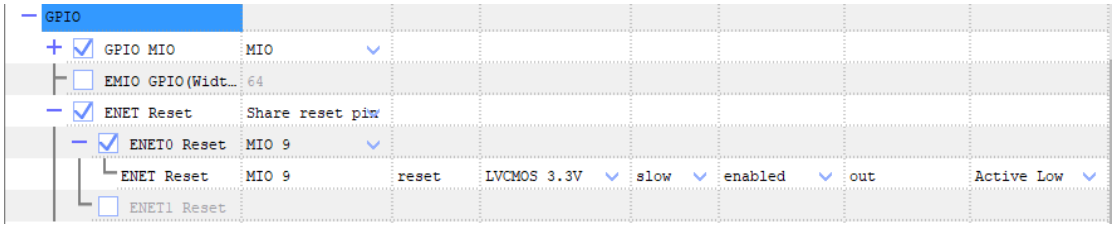
配置 QSPI，根据原理图找到 QSPI Flash，配置为如下；



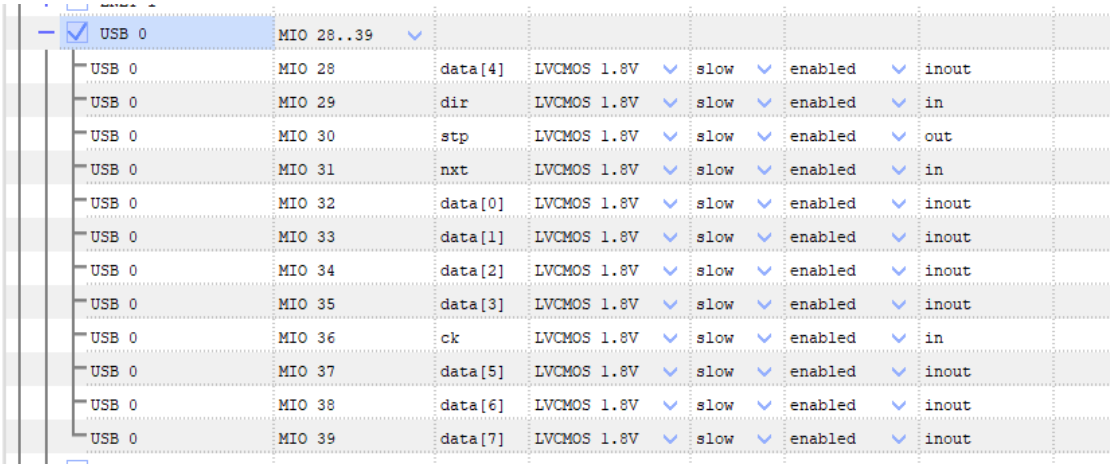
配置以太网根据原理图找到以太网，配置为如下；



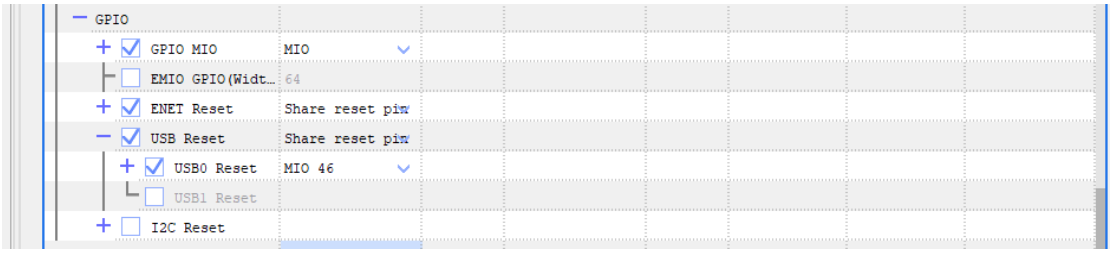
ENET Reset 对应原理图 PHY_RESET，勾选 ENET Reset，



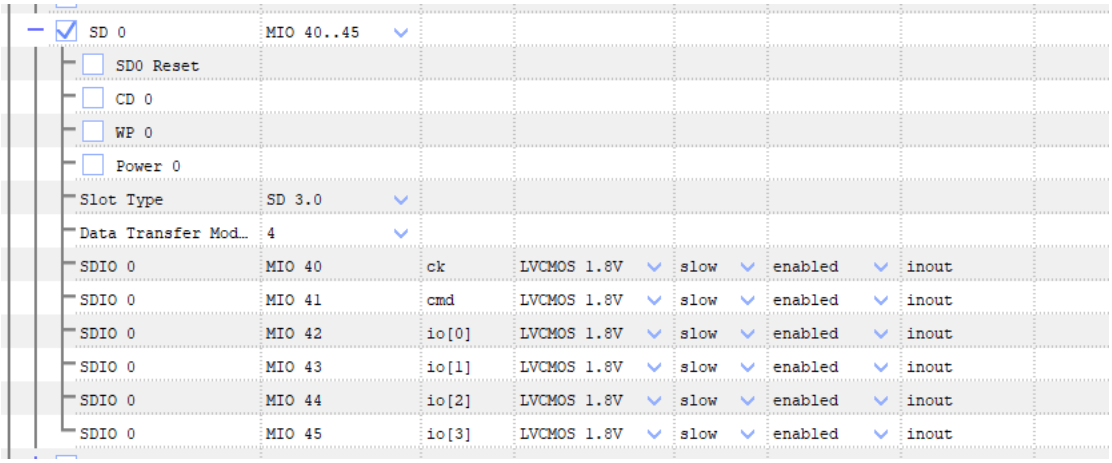
配置 USB0，根据原理图找到 PU_USB，勾选 USB0，



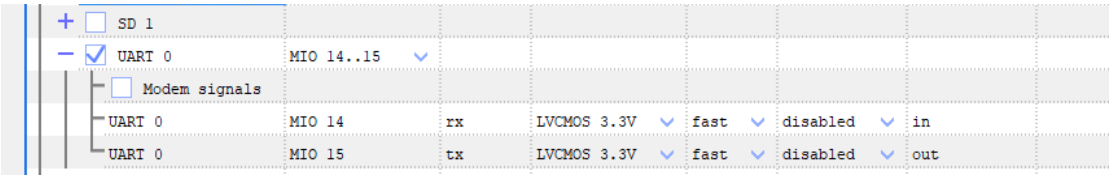
USB Reset 对应原理图 USB_RESET，勾选 USB Reset，



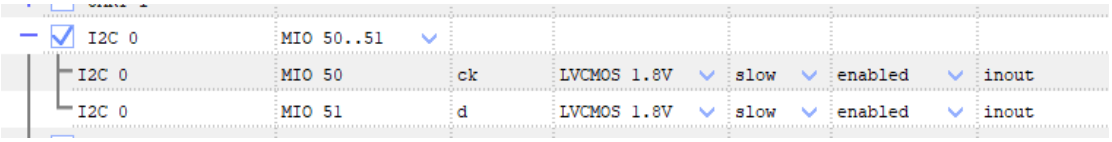
除了 QSPI 启动，还有 SD 卡模式启动，选择 SD0 对应原理图



配置 UART0，根据原理图找到 UART

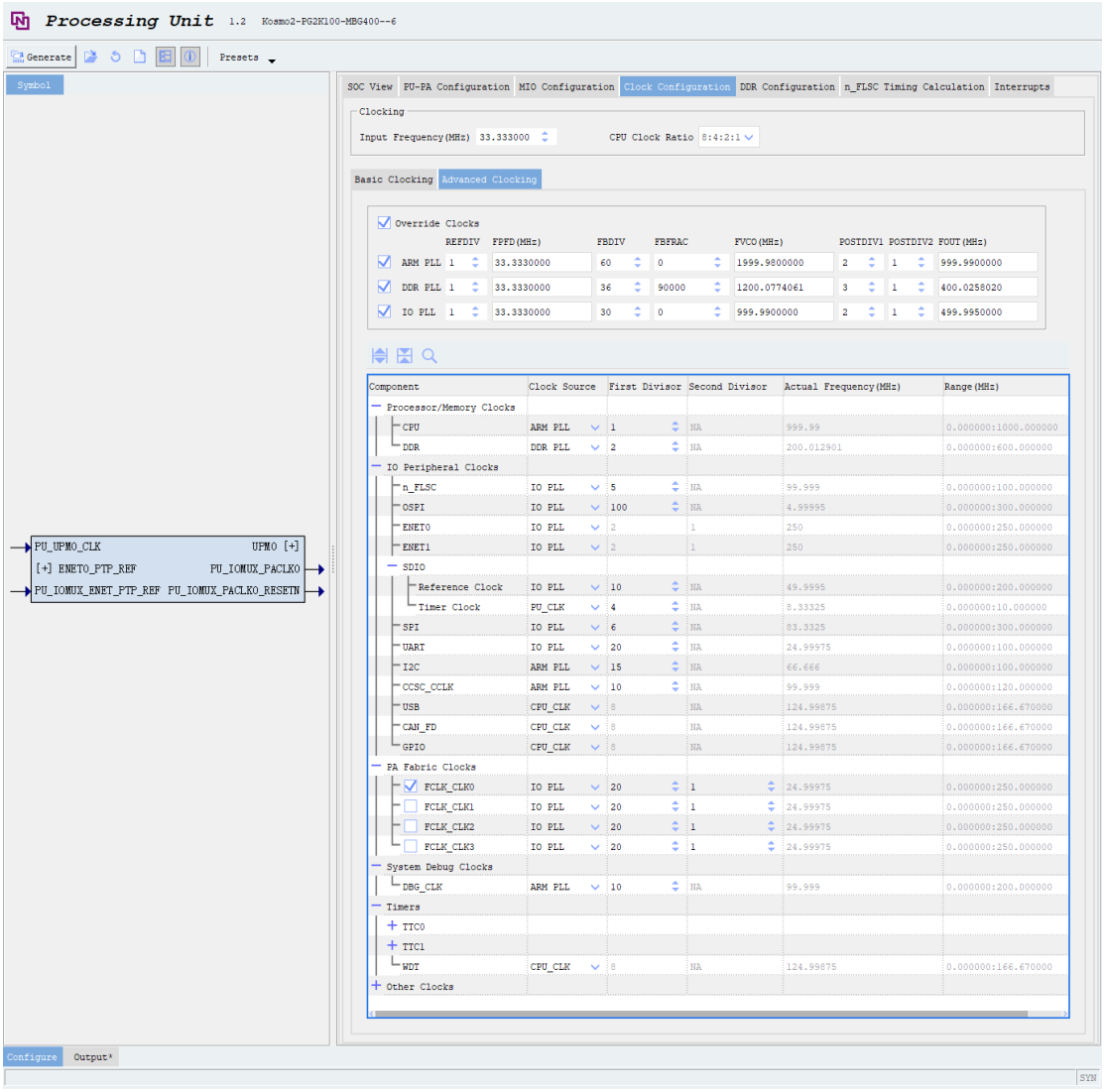


配置 IIC0，根据原理图找到连接在 PU 管脚

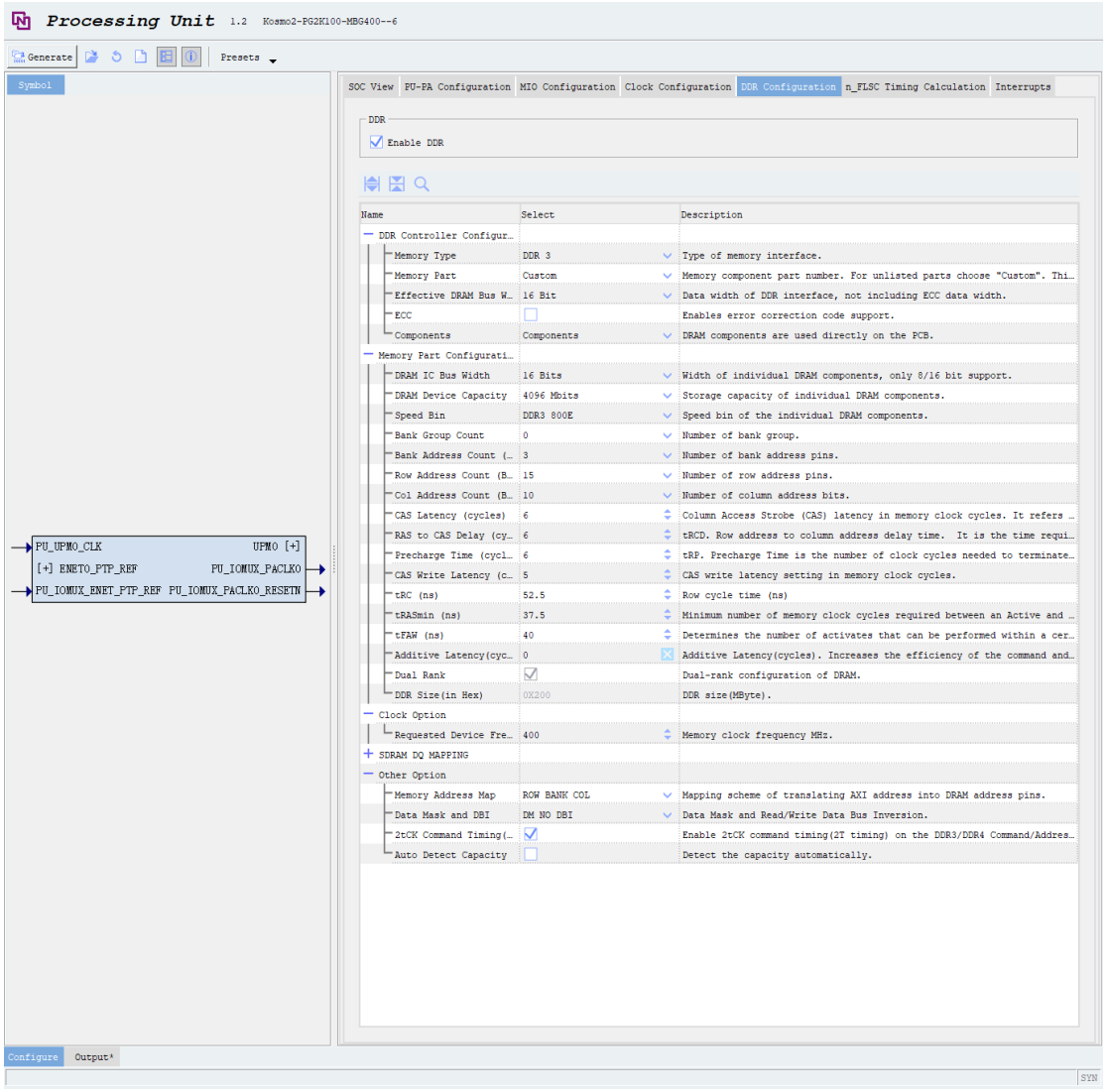


配置 PU 端剩余未分配的 MIO，勾选 GPIO MIO，便可以控制剩余未分配的 MIO 用作 GPIO

点击 Clock Configuration 选项卡， 如下图所示进行配置



点击 DDR Configuration 选项卡， 如下图所示进行配置。



1)点击选项栏中的“DDR Configuration”选项，会出现对应的 DDR 配置页面。

2)勾选 DDR 使能选项“Enable DDR”，颗粒参数不支持调整。

3)DDR 类型选择，可选择“DDR3”或者 DDR4。

4)DDR 的型号选择，由第三步中选择类型之后需要选择对应的型号，DDR3 与 DDR4 暂时只支持一种型号。

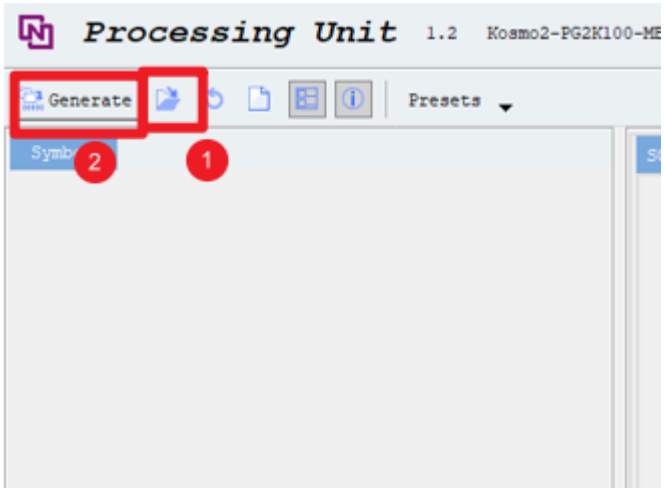
5)DDR 控制器侧频率选择，DDR3 支持 933MHz 和 400MHz，DDR4 支持 800MHz 和 1066MHz。

6)PHY 侧的速率，DDR3 所支持的为 DDR3_800F 和 DDR3_1866F，DDR4 所支持的为 DDR4_1600F 和 DDR4_2133F，其与第五步中的对应，是第五步中的两倍。而且会与第五步所设置的自适应（修改第六步之后第五步会自动修改，修改第五步之后第六步也会自动修改）

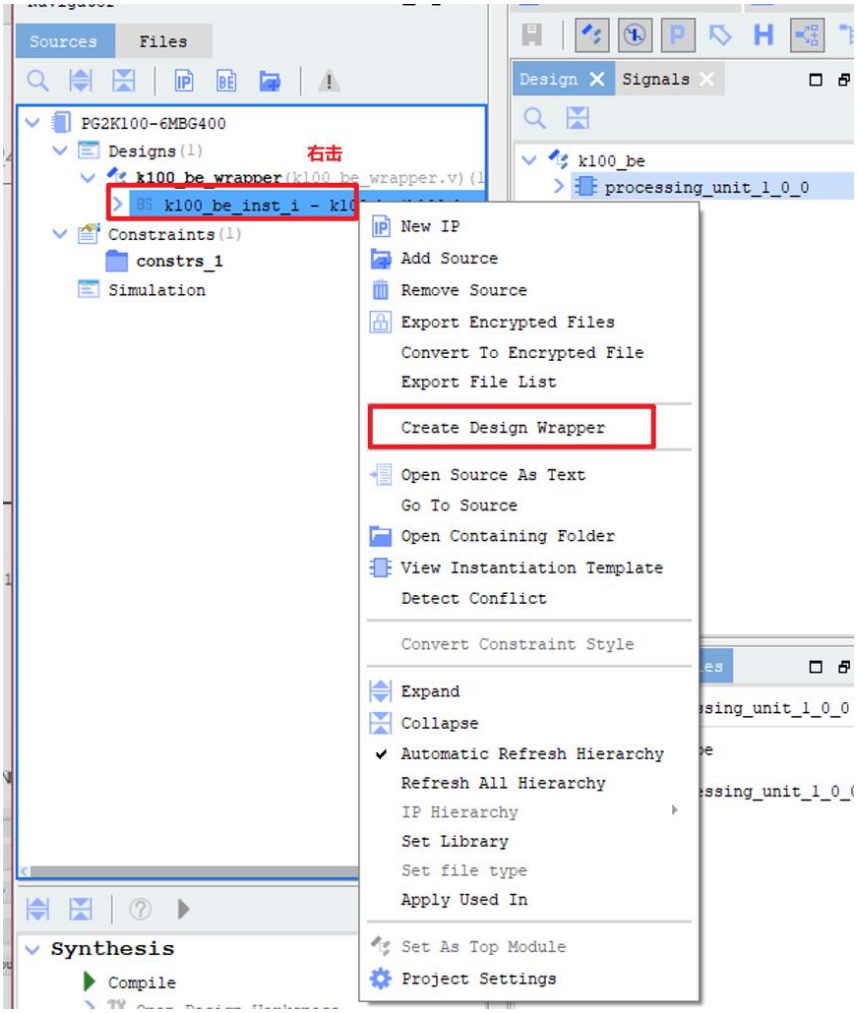
注意：这里设置的频率之后需要修改 DDR 的时钟频率，这里设置的频率是 Clock Configuration 中所设置频率（FOUT）的两倍。 （例如：若将这里的频率设置为 400MHz，

需要通过倍频和分频将 Clock Configuration 的 DDR 的频率（FOUT）设置为 200MHz）。用新版本的工具打开旧版本工程时，需要重配 DDR 参数。

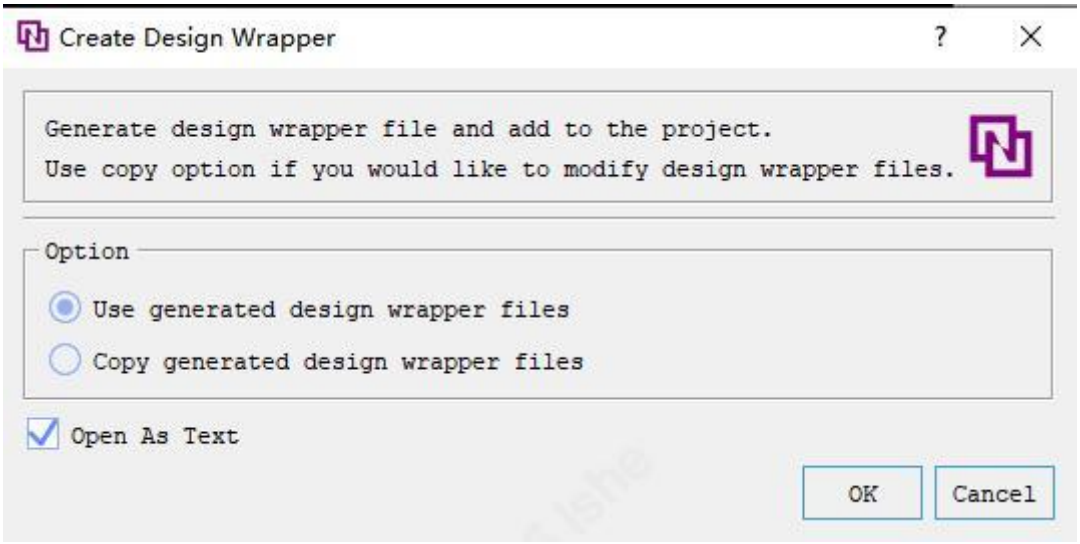
先点击保存按钮，然后点击 Generate 按钮，最后关闭 IP 定制对话框。



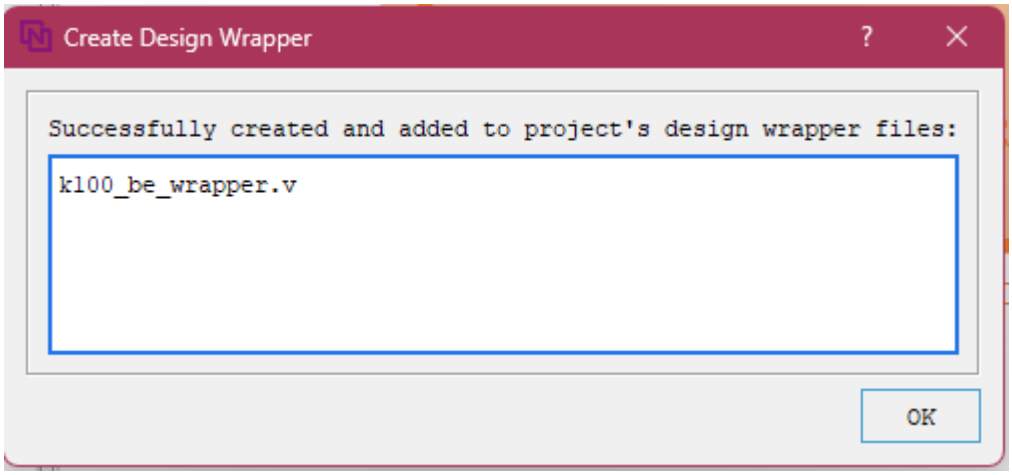
生成顶层文件，右键.pbs 文件， 点击 Creat Design Wrapper 选项。



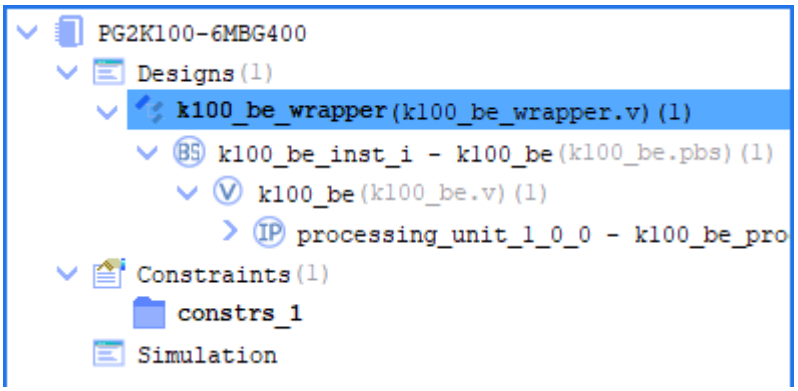
生成如下图的界面，点击“OK”按钮。



生成如下图界面，点击“OK”按钮。

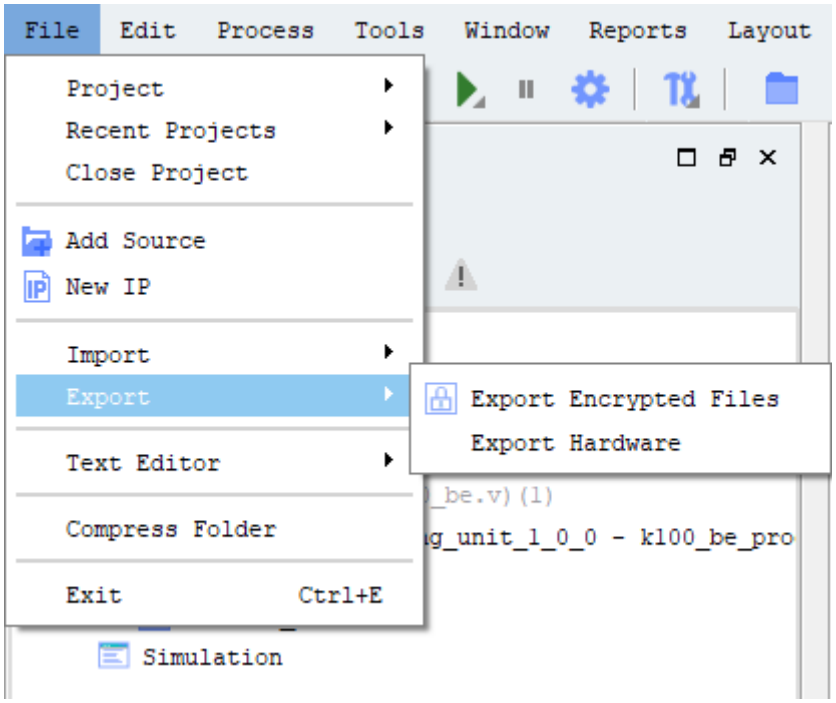


在左侧工程栏中会生成顶层文件

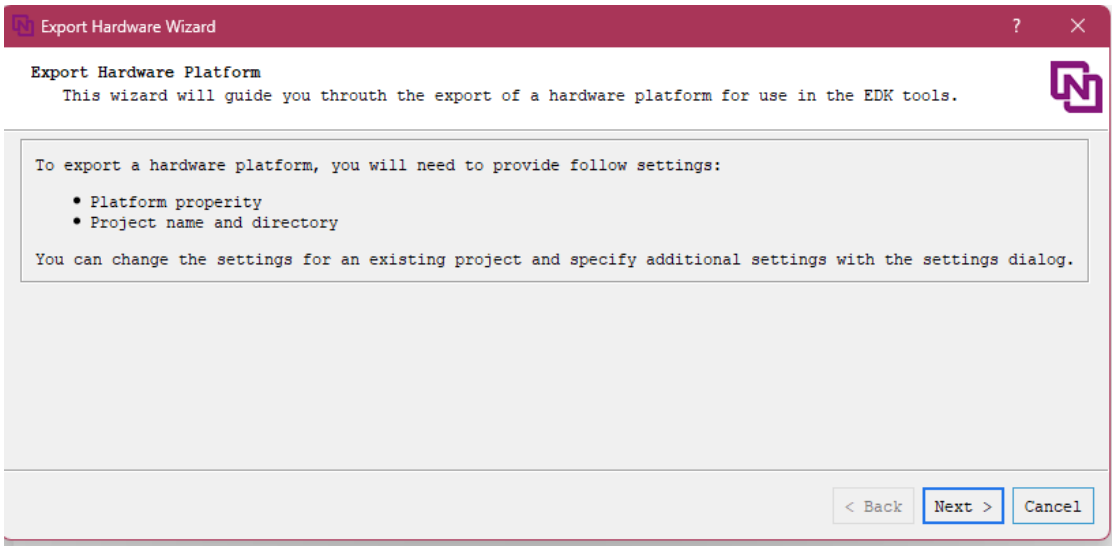


2.1.导出 Hardware

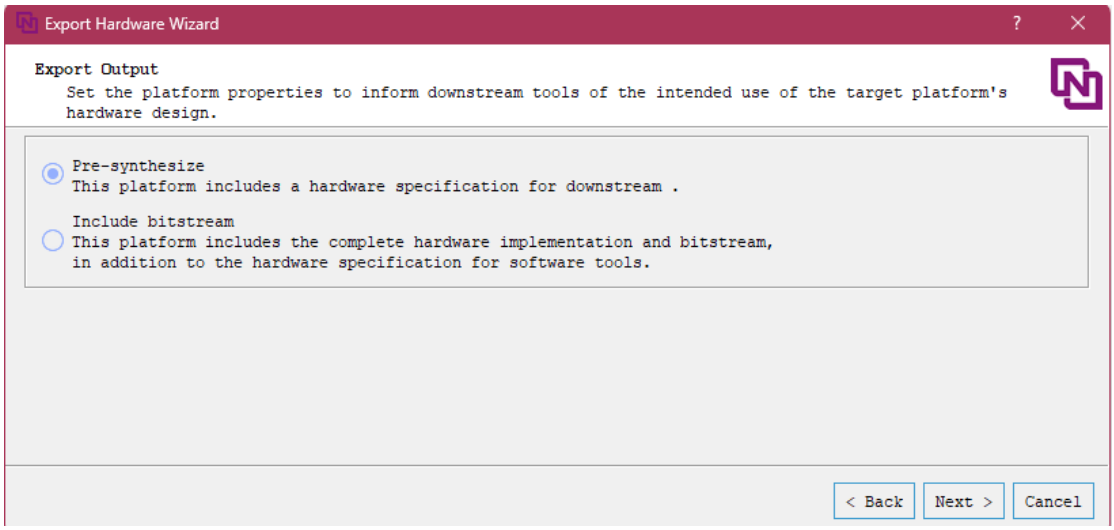
1.点击菜单[File -> Export -> Export Hardware]。



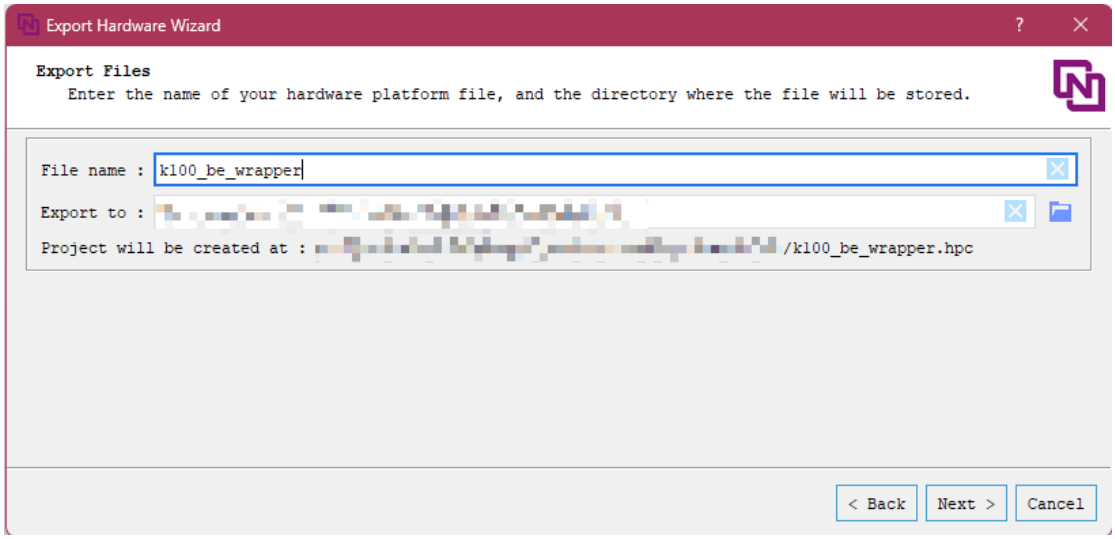
2.生成如下图所示的 hardware 导出界面。



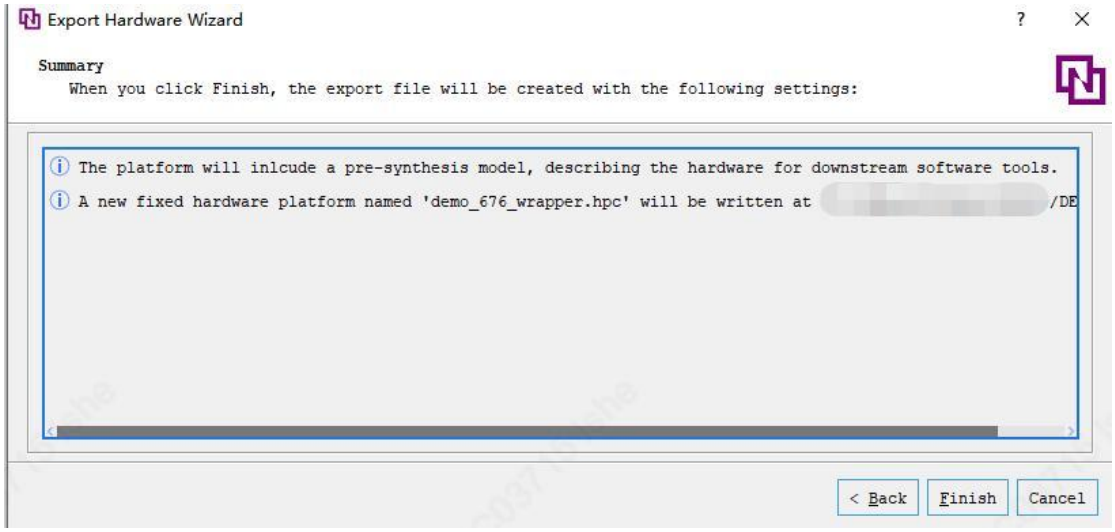
3. 点击 Next。



4.点击 Next。



5.输入 File name 并点击 Next。



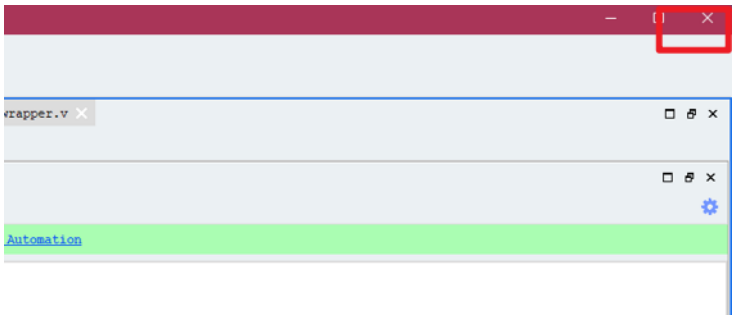
6.点击 Finish，对话框将关闭，回到主界面。

2.2.操作注意点

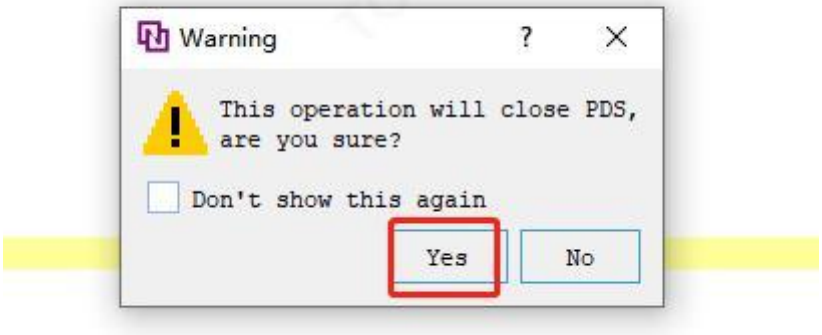
关闭工程时需要手动点保存

第一次关闭 PDS 工具时需要对工程进行保存（会弹出对话框提示保存，而不是自己主动通过 ctrl+s 来保存），按照如下图所示进行操作。（若未按照下图所示进行操作，则下次直接打开工程时，该工程所配置的文件不会被加载需手动重新加载）。

1.按照如下图所示点击右上角红框中的“x”关闭软件。



2.选择“Yes”按钮。

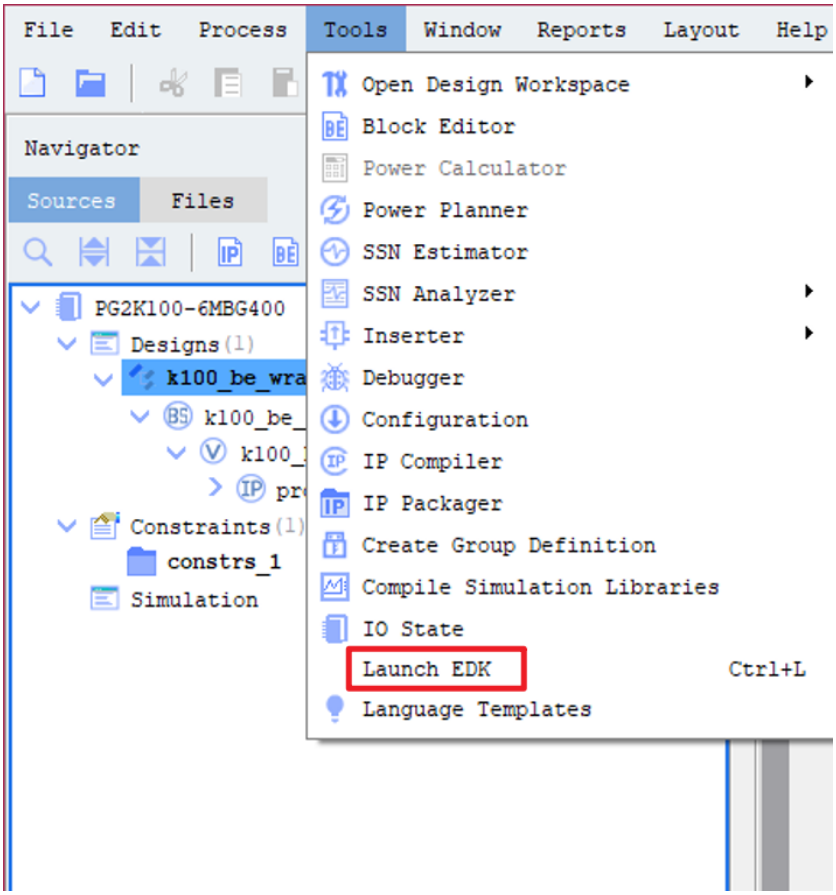


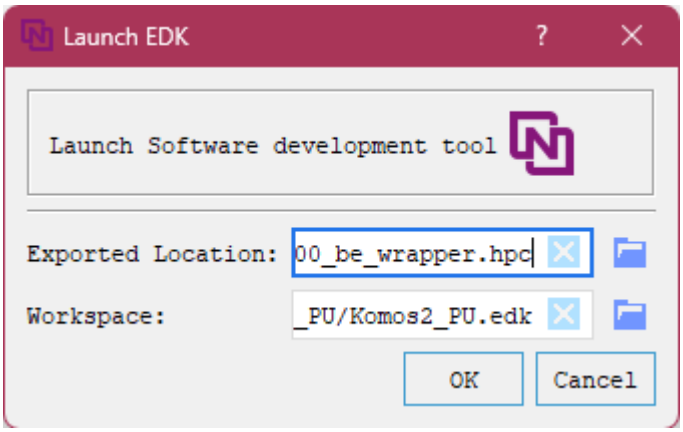
3.选择“Yes”按钮，退出 PDS 界面。

3.EDK 工程

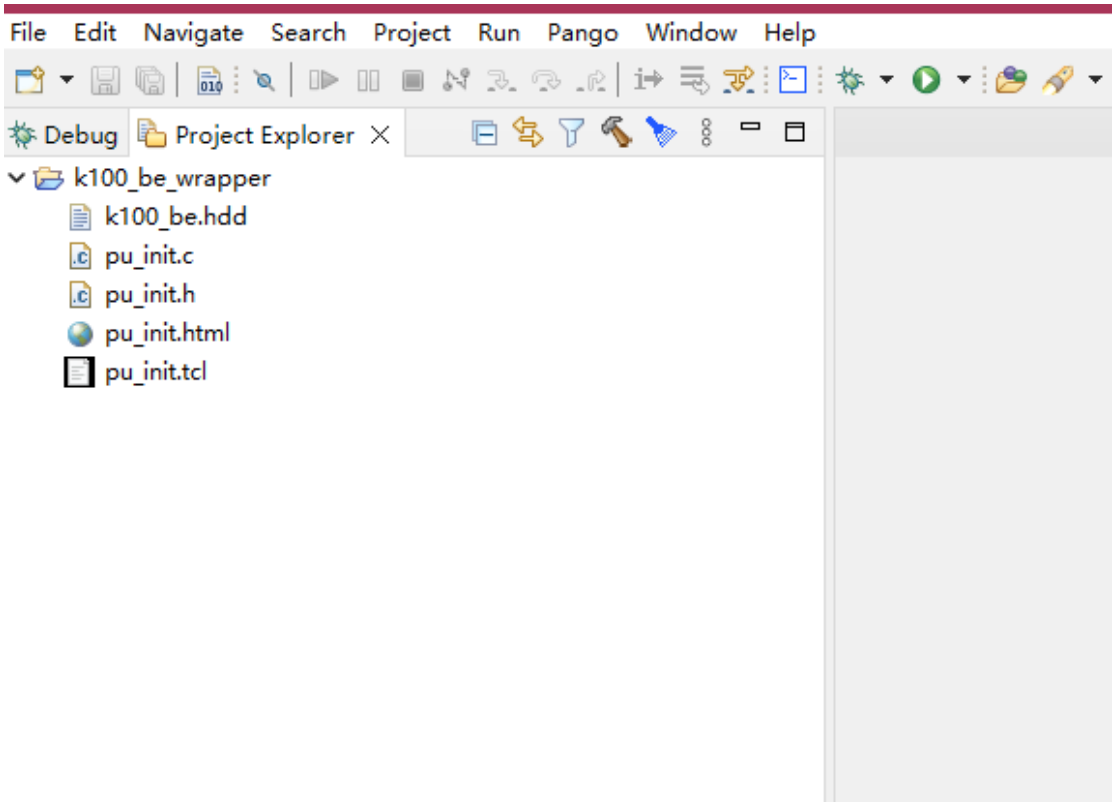
3.1.启动 EDK

1. 在主界面，依次点击 PDS 菜单[Tools-> Launch EDK]。

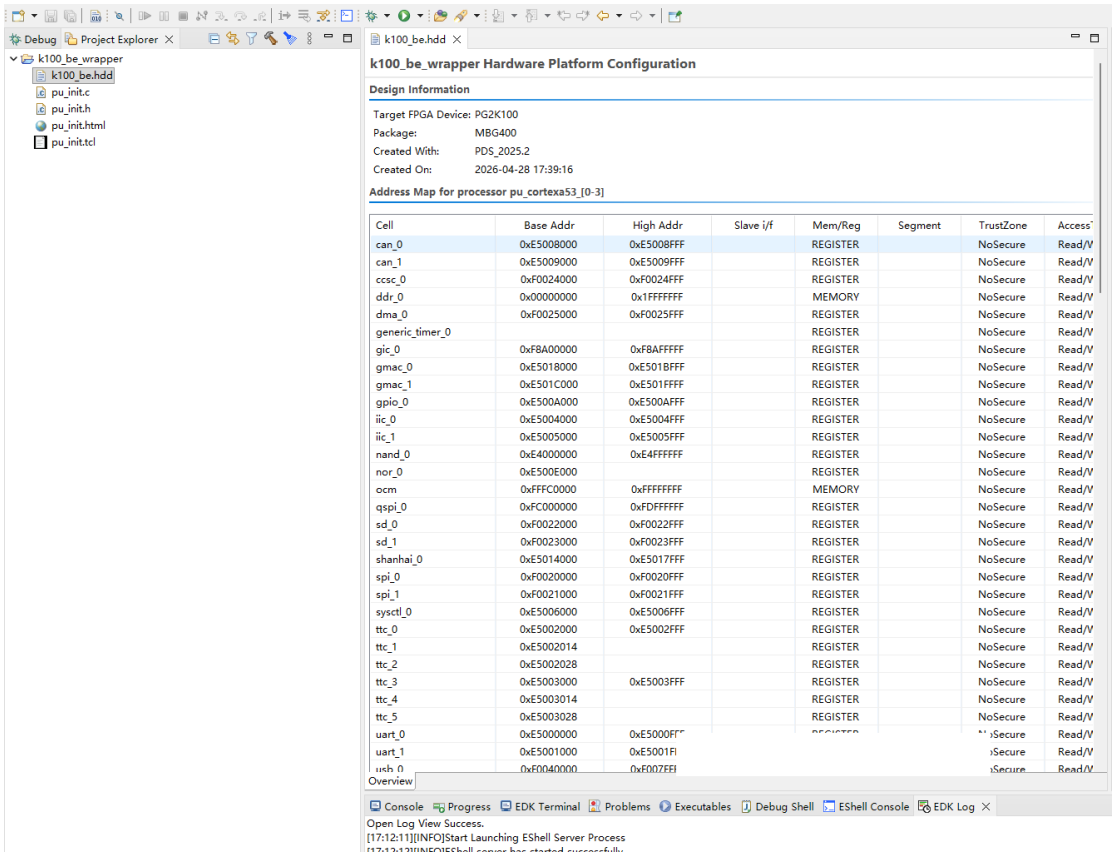




2. 点击上图的“OK”按钮， 会启动 EDK， 启动后的 EDK 如下。

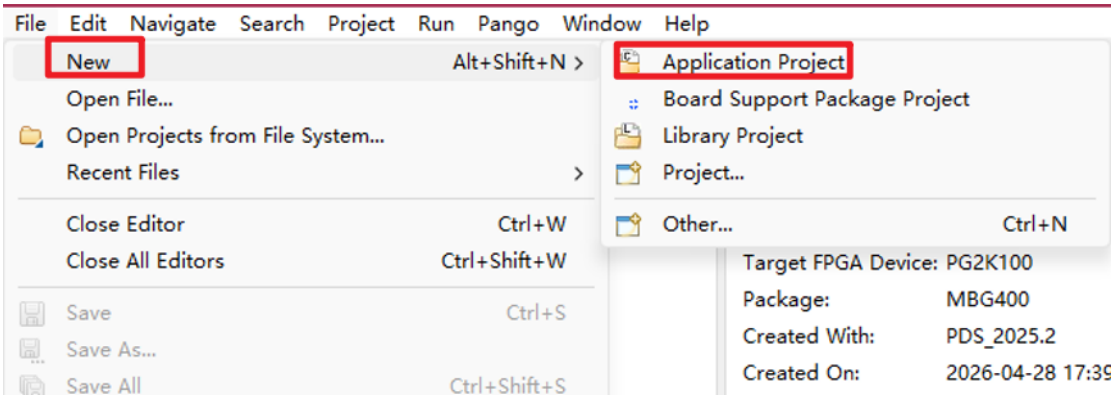


3.启动 EDK 之后会生成平台相关的文件夹，其中有一个“demo_100.hdd”（名字与 PDS 中的 Block Editor 工程名相关），这个文件包含了 PDS 硬件设计的信息，有利于软件开发使用。

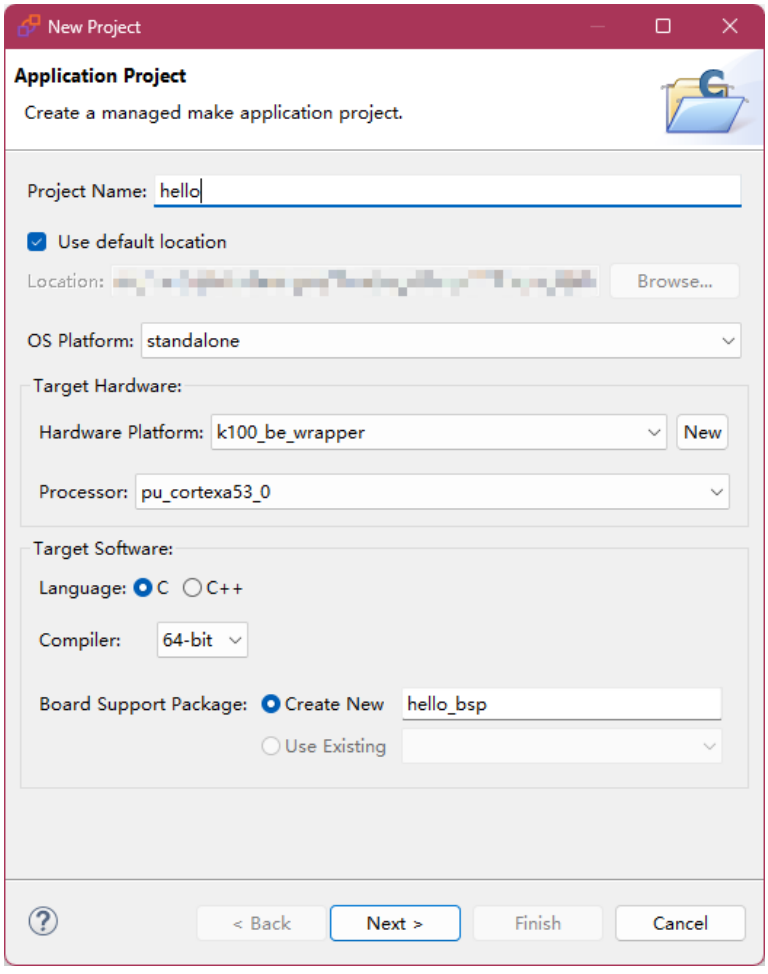


3.2.创建 project

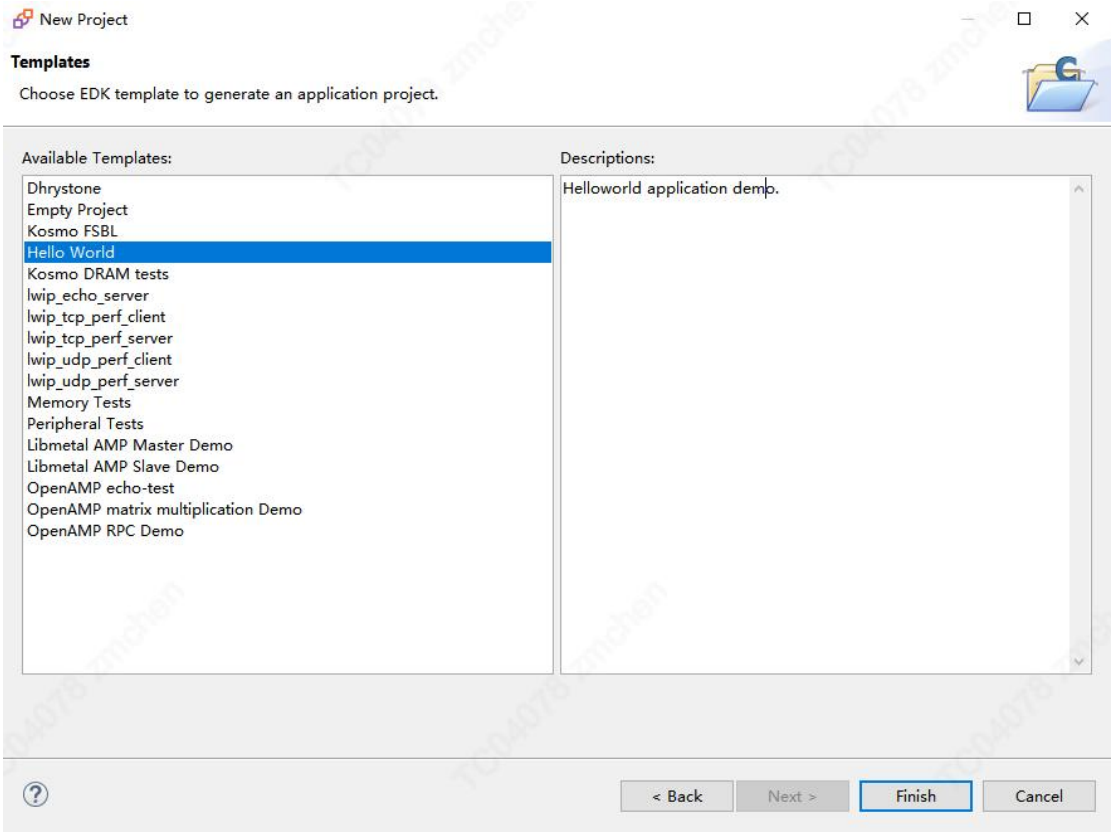
1.依次点击菜单[File -> New -> Application Project]。



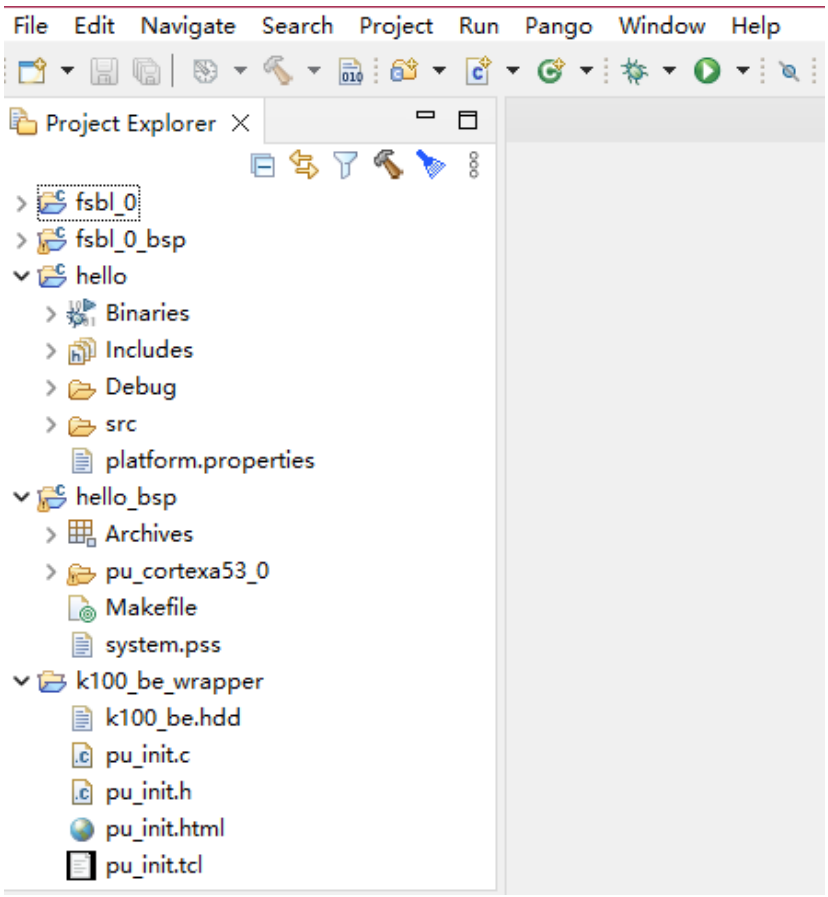
2.弹出 New Project 界面，按下图进行配置，然后点击“Next”按钮。



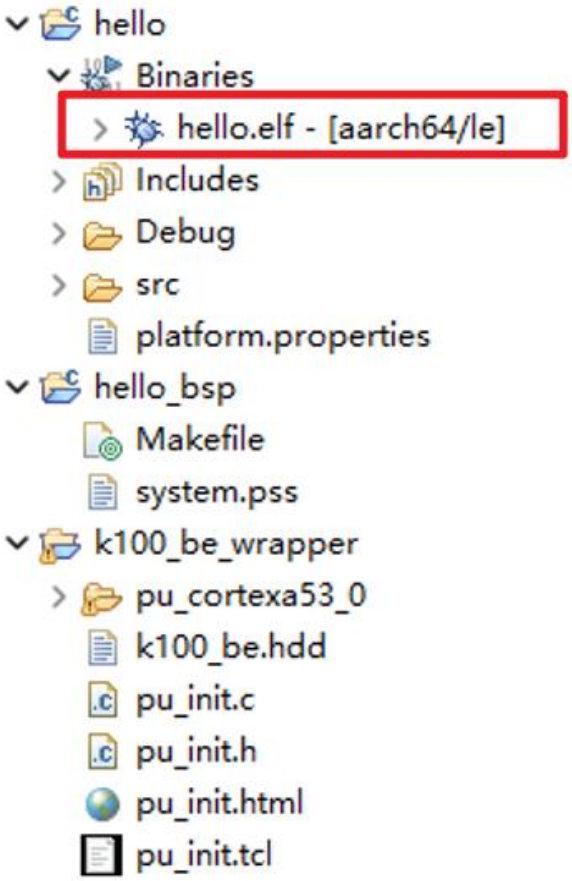
3.弹出模板工程选择界面，选择“HelloWorld”模板工程，然后点击“Finish”按钮。



4.等到窗口自己关闭，会回到主界面。



5.创建 helloworld 工程之后，第一次程序会自动编译，等待编译结束，如下图所示生成的可执行.elf 文件。

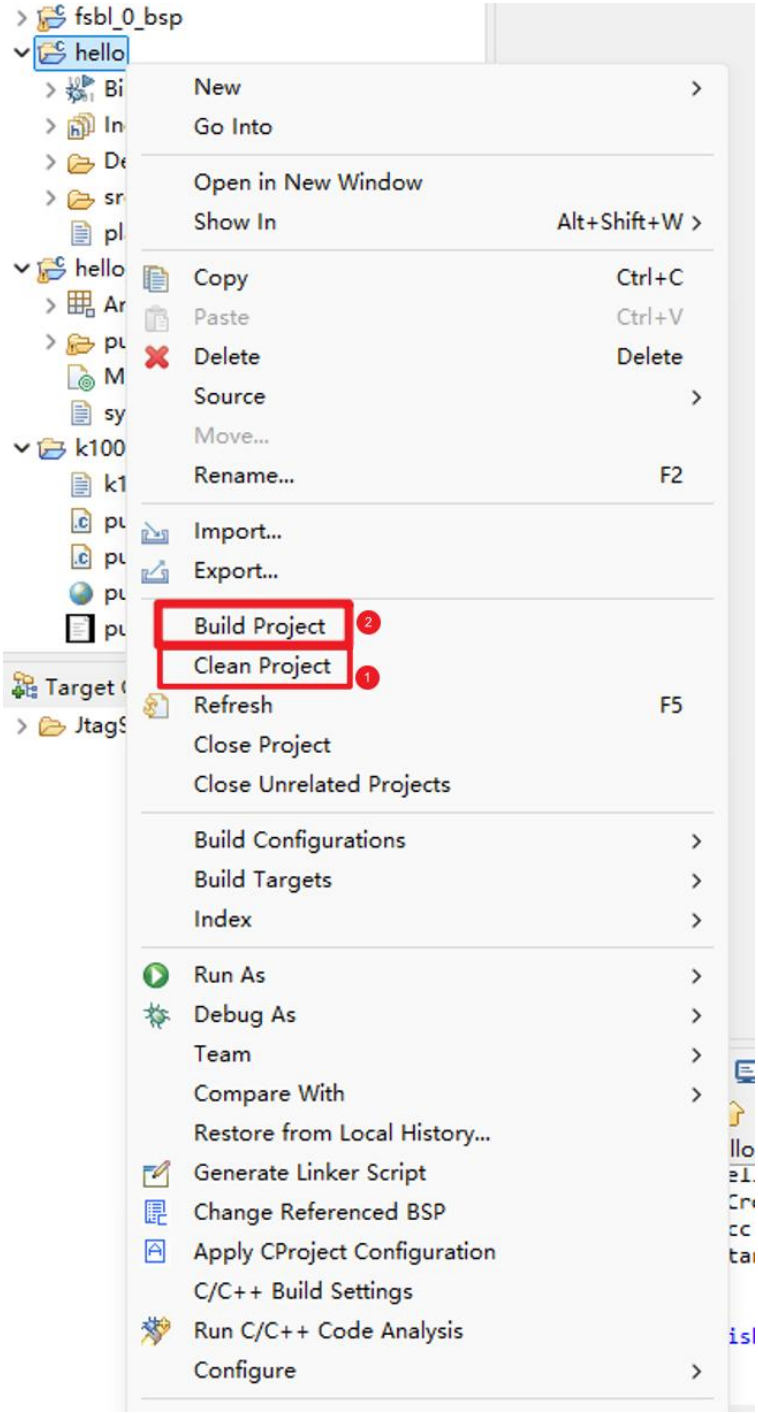


3.3.编译 project

创建工程之后，第一次程序会自动编译，如果修改代码之后手动重新编译需要先清除再编译，如下图=顺序执行：

选中工程在菜单栏使用下快捷图标

或右击工程

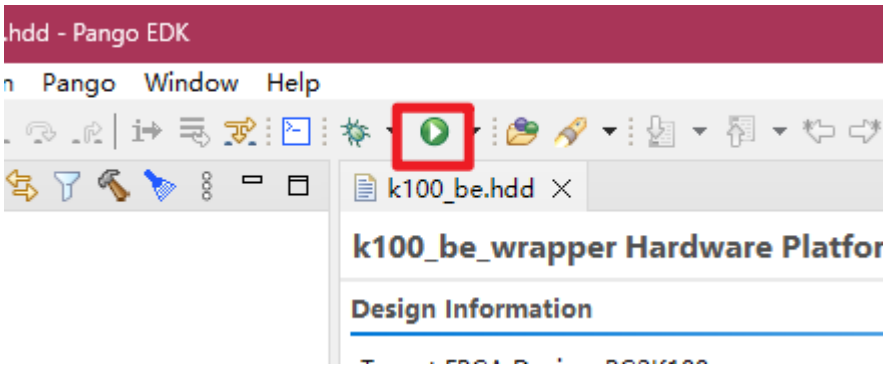


成功编译后信息栏打印成功信息。

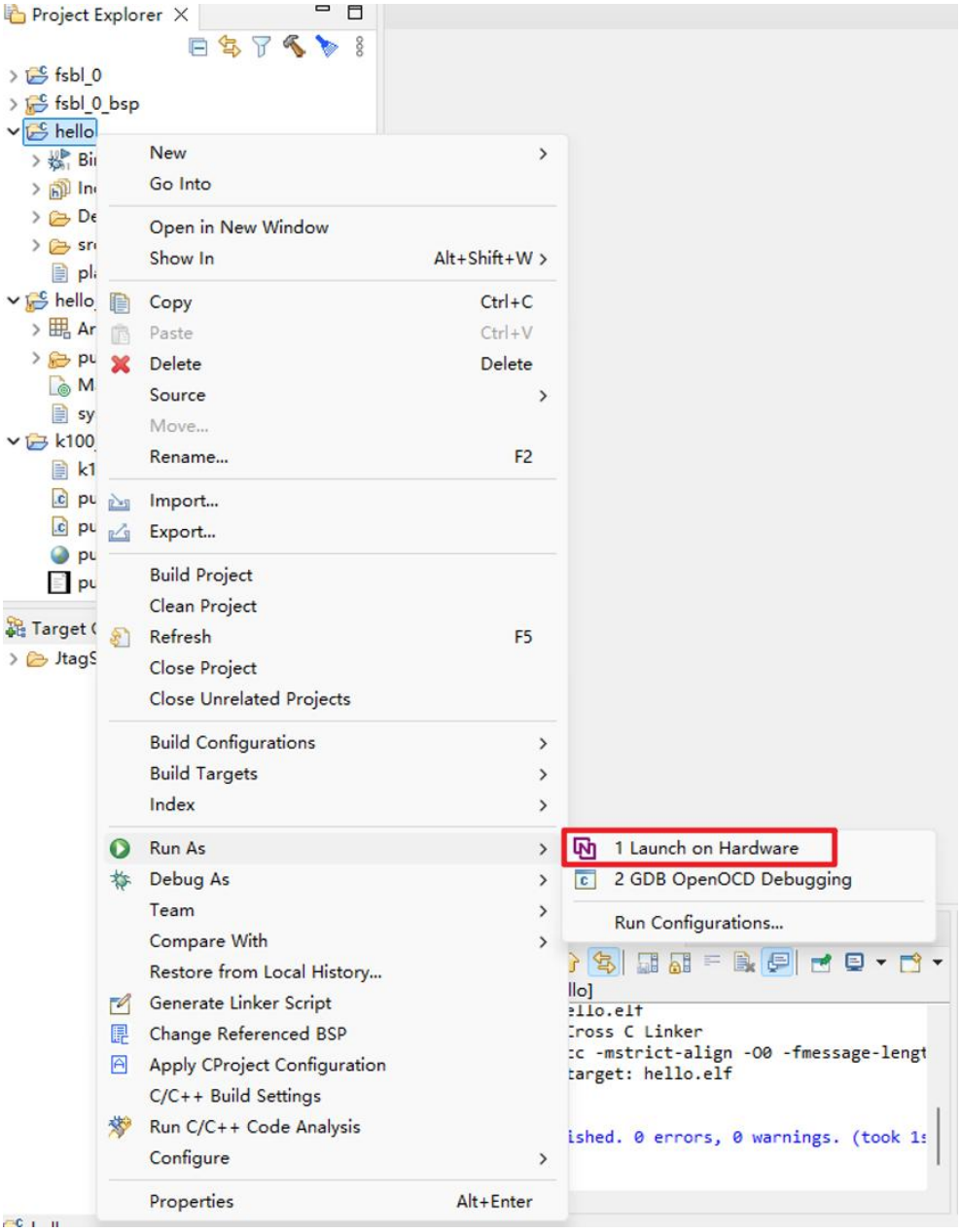
```
aarch64-none-elf-gcc -mstrict-align -O0 -fmessage-length=0 -ffunction-sections -fdata-sections -Wall  -g3 -DARMA53_64 -D__aarch64__ -I
Finished building: ../src/helloworld.c

Building file: ../src/platform.c
Invoking: GNU Arm Cross C Compiler
aarch64-none-elf-gcc -mstrict-align -O0 -fmessage-length=0 -ffunction-sections -fdata-sections -Wall  -g3 -DARMA53_64 -D__aarch64__ -I
Finished building: ../src/platform.c

Building target: hello.elf
Invoking: GNU Arm Cross C Linker
aarch64-none-elf-gcc -mstrict-align -O0 -fmessage-length=0 -ffunction-sections -fdata-sections -Wall  -g3 -T "D:\Users\lukn\Desktop\0.
Finished building target: hello.elf


17:49:01 Build Finished. 0 errors, 0 warnings. (took 1s.500ms)
```

3.4.下载

选中工程在菜单栏使用下快捷图标



或者右击工程



3.5.固化程序

3.5.1.启动流程

PG2K100 启动一般最少包含 Stage0 和 Stage1，一般在跑系统(Linux)的情况下才会使用到 stage2，其主要的作用介绍如下：

Stage0：在上电复位或者热复位之后，处理器首先执行 BootRom 里的代码，这一步是最初始启动设置 BootRom 是在非 JTAG 模式下才会被执行，主要存放了一段用户不可更改的代码。其主要作用有：

一、包含了最基本的 NAND，NOR，Quad-SPI，SD 和 CCSC 的驱动；

二、把 fsbl 的代码搬运到 ocm 之中（即 stage1 的代码），空间限制为 92KB。

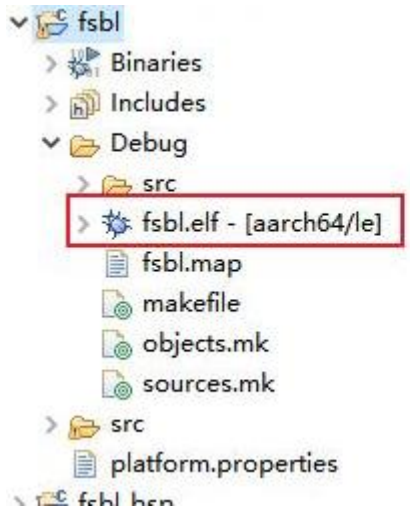
Stage1：当 BootRom 搬运 FSBL 到 OCM 后，处理器开始执行 FSBL 代码，FSBL 主要有以下作用：

- 初始化 Pu 端的配置，主要包含 pds 中相关的配置。包括初始化 DDR，MIO，SCLR，时钟等。主要执行 pu_init.c，其中 pu_init.tcl 的执行效果和 pu_init.c 是一样的
- 若存在 PA 端程序，加载 PA 端 bitstream
- 加载 Stage2 或者 bare-metal 应用程序到 DDR
- 跳转到 Stage2 或者 bare-metal 应用程序

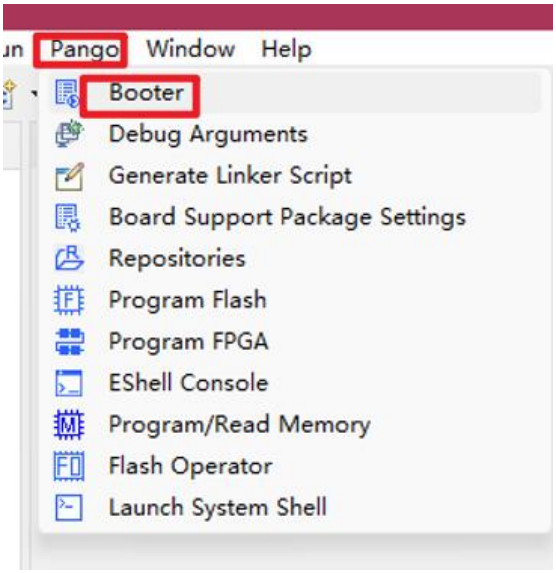
Stage2：Second stage bootloader 是可选项，一般在跑操作系统的情况下使用，比如 Linux 系统的 uboot。

3.5.2.创建 BOOT_PG 文件

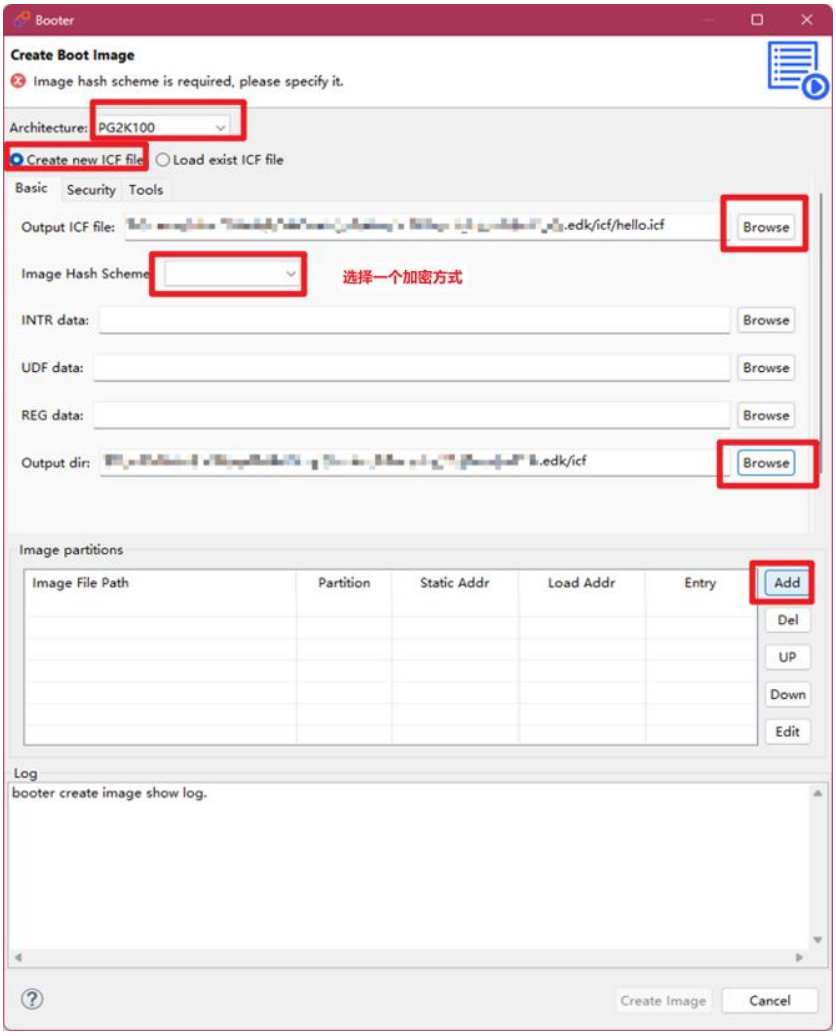
1.参考之前创建 project 章节中按照其流程可以得到 fsbl 的可执行程序（这里为 fsbl.elf），如下图。



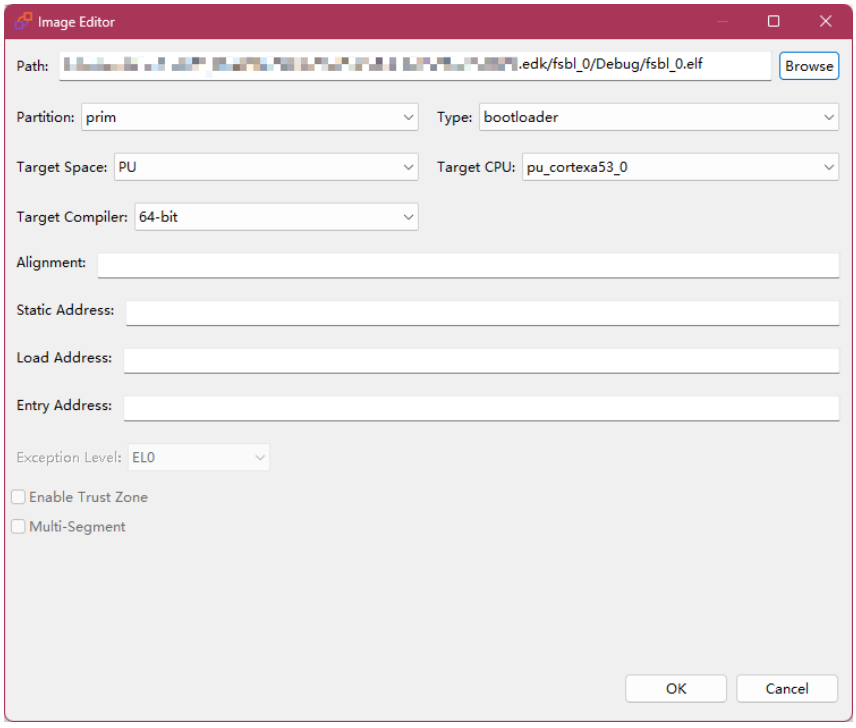
2. 依次点击菜单[Pango -> Booter]，



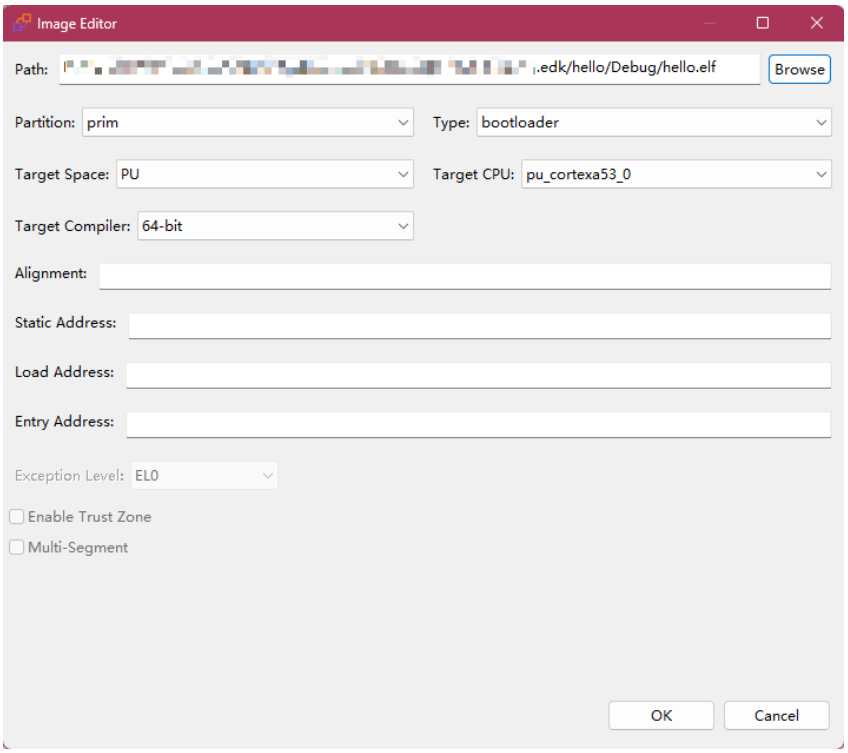
按照下图所示进行配置，其中注意 Output ICF file，这里为新创建 ICF file，所以 Output ICF file 中所填的 icf 文件是自己新创建后生成的文件名并不是原本该文件夹中存在的 icf 文件（若选中已存在的 icf 文件，会覆盖掉该文件）



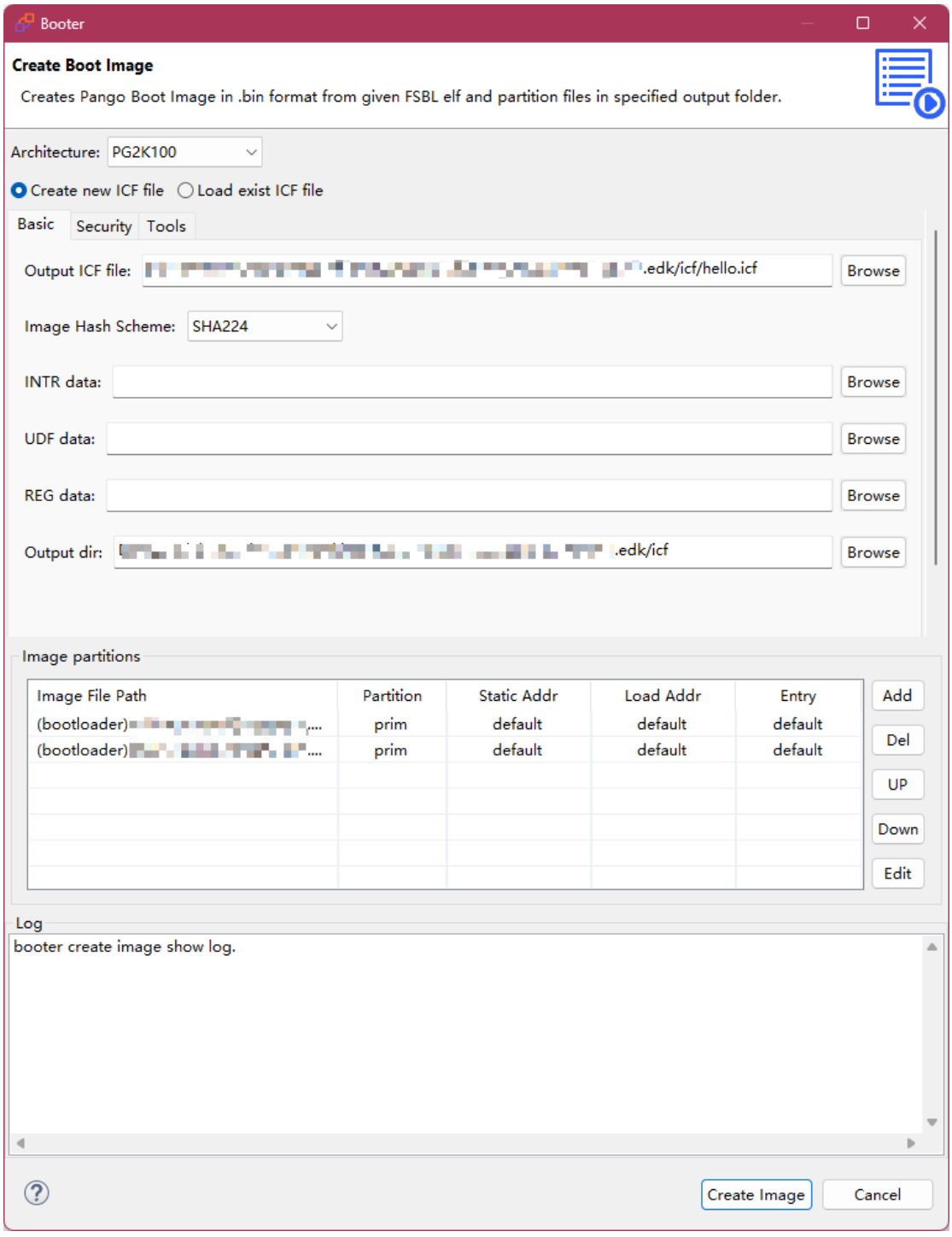
3. 在上图的 Image partitions 区域，右侧点击 Add 按钮



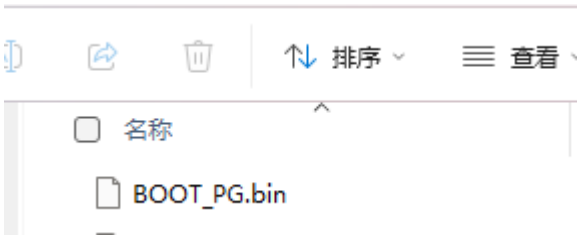
4.上图的 Path，就是步骤 1 中所说的 fsbl.elf 文件,添加文件后继续添加程序 elf 文件。



5. 如上图选择需要固化的程序文件这里为 hello.elf 点击上图中的“OK”按钮。



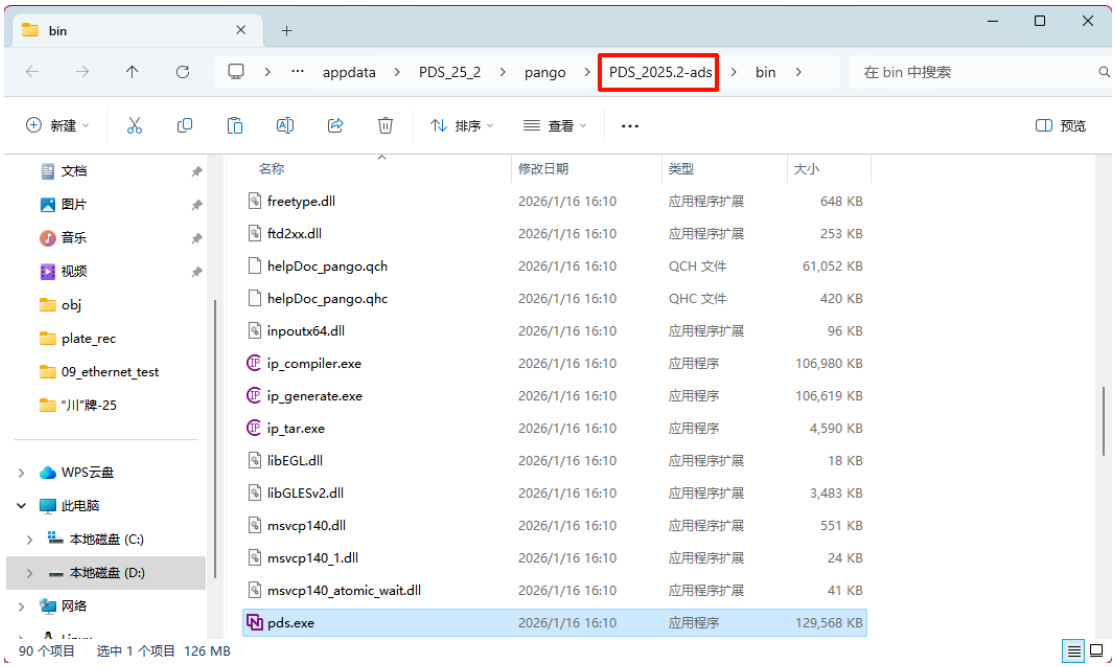
6.点击上图中的 Create Image 按钮。等待生成 BOOT_PG.bin 进度条结束，在上图中的 Output dir 对应的目录中就可以看到 BOOT_PG.bin 文件。



在将程序烧录到 Flash 之前还需要替换 PDS 安装目录下的 flash_device.json 文件，具体操作如下：

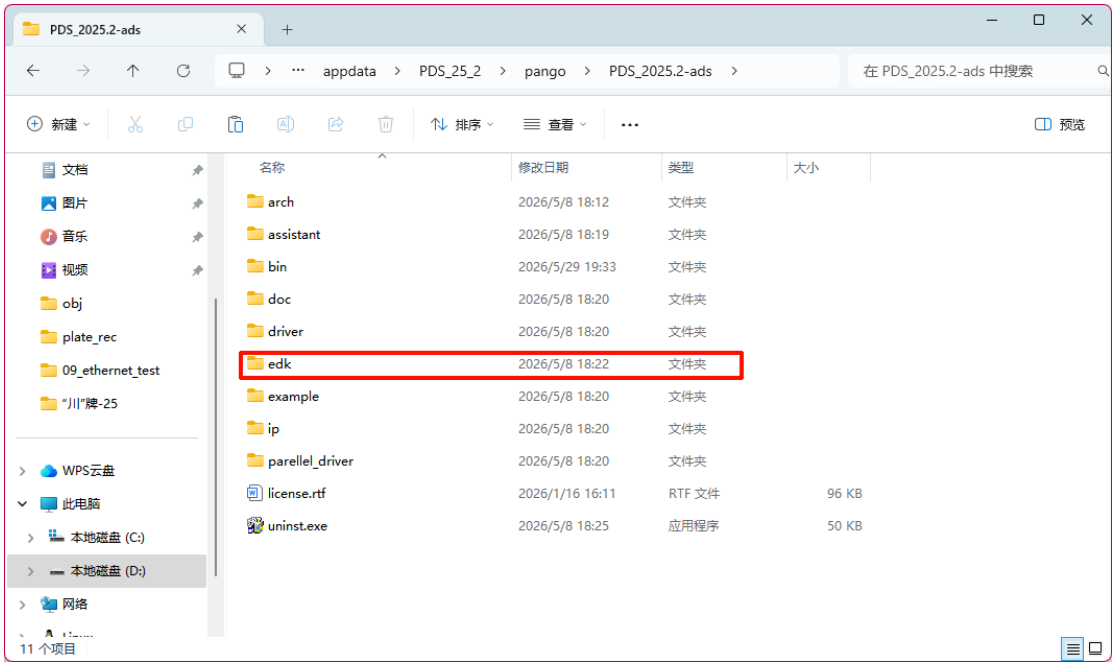
首先右键打开 PDS 的安装位置

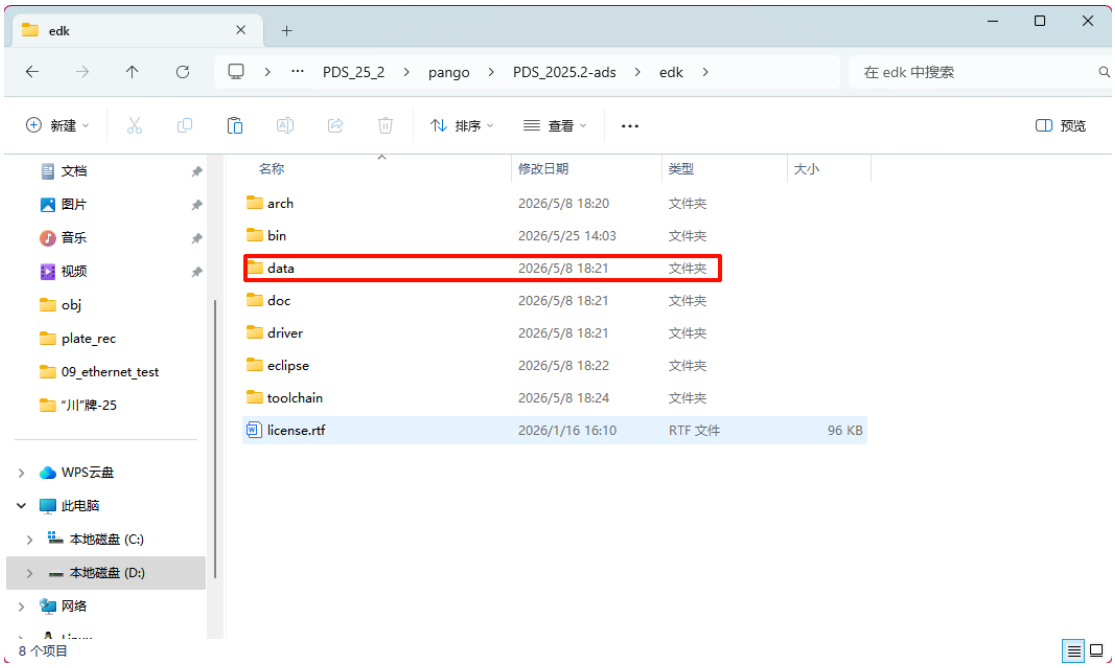
打开后界面如下操作：

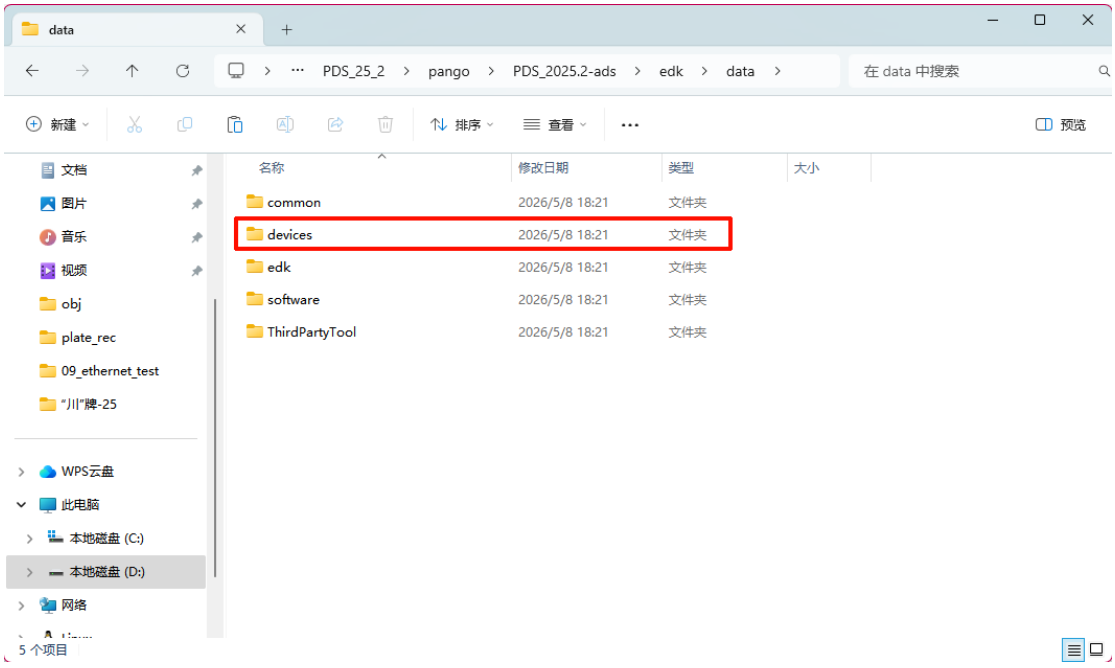


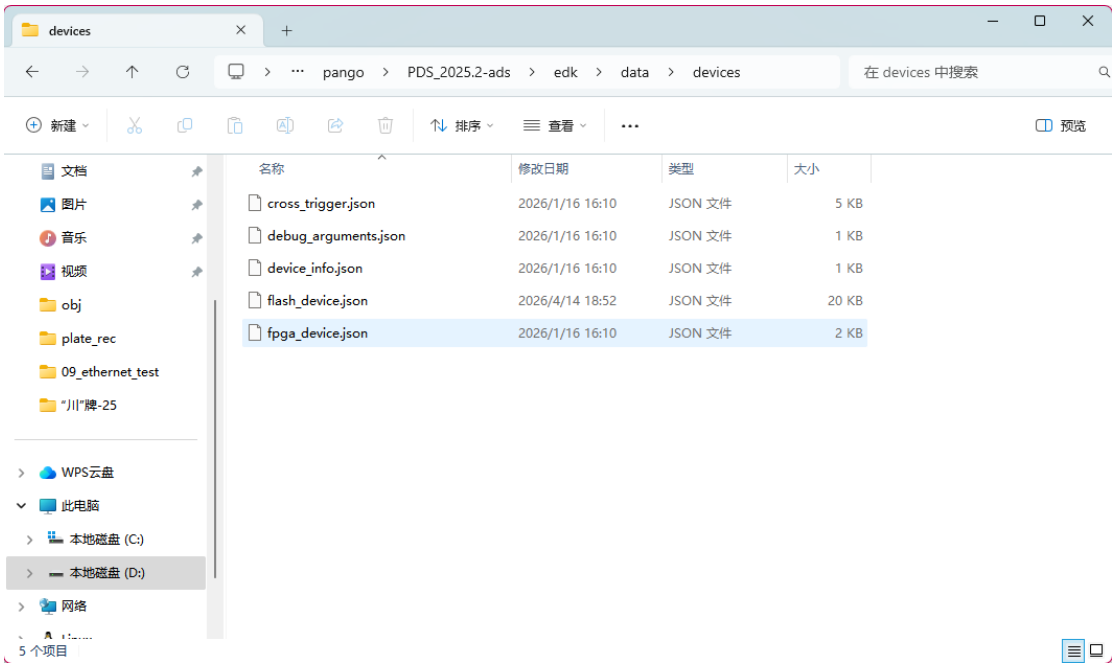
进入 PDS_2025.2-ads 的安装目录

接着进入 edk 工具的目录，并在该目录中依次点击 data/device

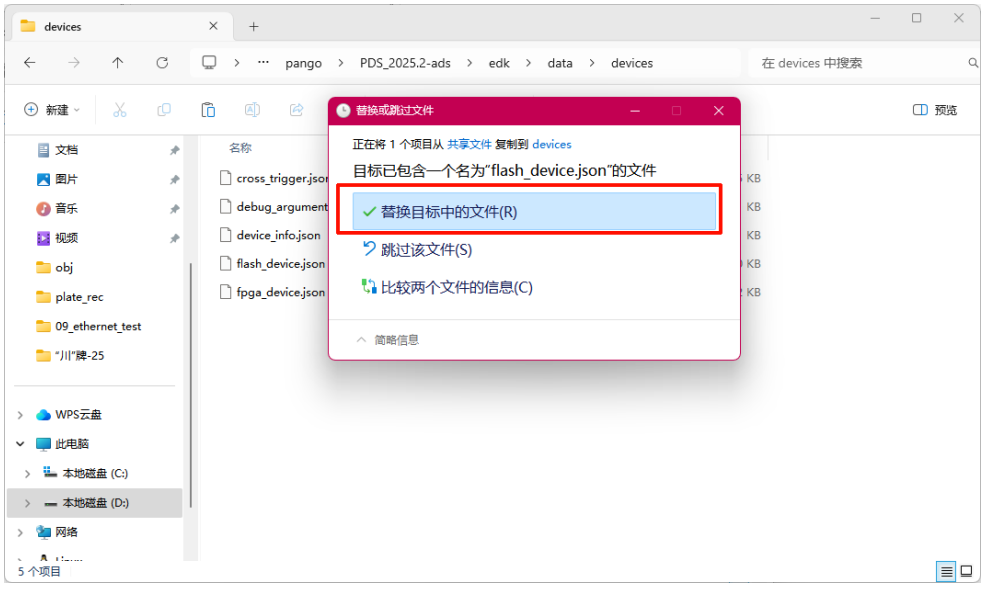



edk
PDS_25_2 › pango › PDS_2025.2-ads › edk ›
在 edk 中搜索
新建
排序
查看
预览
文档
图片
音乐
视频
obj
plate_rec
09_ethernet_test
"川"牌-25
WPS云盘
此电脑
本地磁盘 (C:)
本地磁盘 (D:)
网络
名称
修改日期
类型
大小
arch 2026/5/8 18:20 文件夹
bin 2026/5/25 14:03 文件夹
data 2026/5/8 18:21 文件夹
doc 2026/5/8 18:21 文件夹
driver 2026/5/8 18:21 文件夹
eclipse 2026/5/8 18:22 文件夹
toolchain 2026/5/8 18:24 文件夹
license.rtf 2026/1/16 16:10 RTF 文件 96 KB
8 个项目


data
PDS_25_2 › pango › PDS_2025.2-ads › edk › data ›
在 data 中搜索
新建
排序
查看
预览
文档
图片
音乐
视频
obj
plate_rec
09_ethernet_test
"川"牌-25
WPS云盘
此电脑
本地磁盘 (C:)
本地磁盘 (D:)
网络
名称
修改日期
类型
大小
common 2026/5/8 18:21 文件夹
devices 2026/5/8 18:21 文件夹
edk 2026/5/8 18:21 文件夹
software 2026/5/8 18:21 文件夹
ThirdPartyTool 2026/5/8 18:21 文件夹
5 个项目



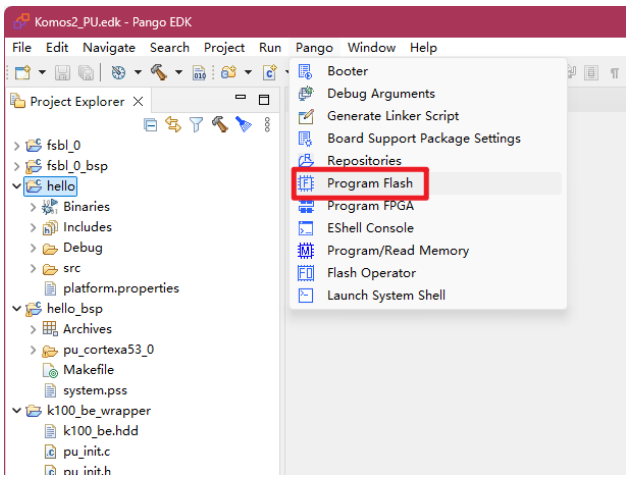
在该目录下将资料包中的 6.EDK_doc 中提供的 flash_device.json 复制粘贴到该目录下



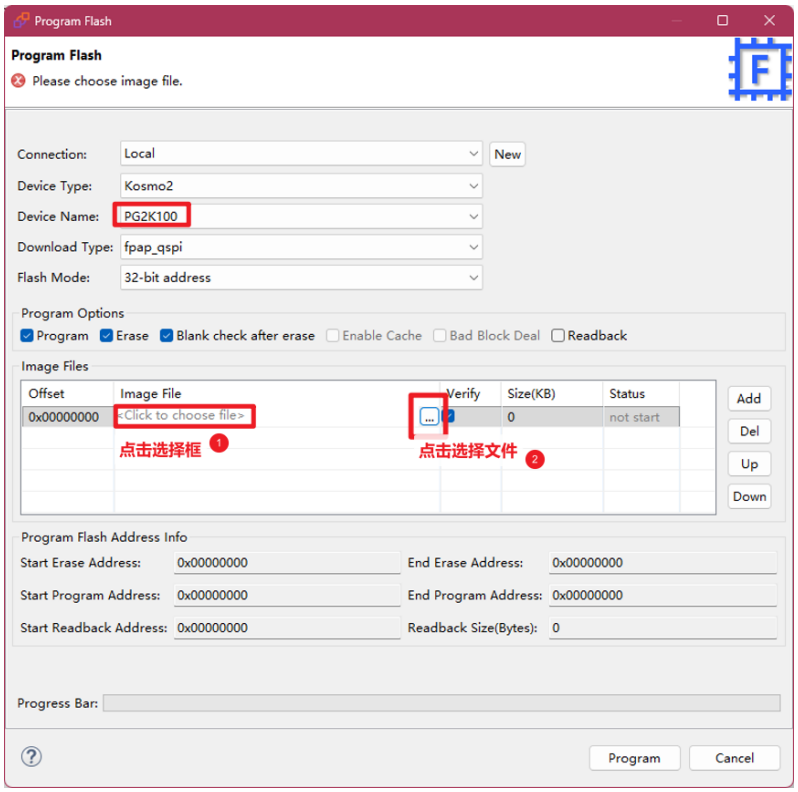
完成文档替换后，重启 PDS 工具即可正常使用 Program Flash 的功能

3.5.3.BOOT 下载到 QSPI flash

1. 依次点击菜单栏的[Pango -> Program Flash]。

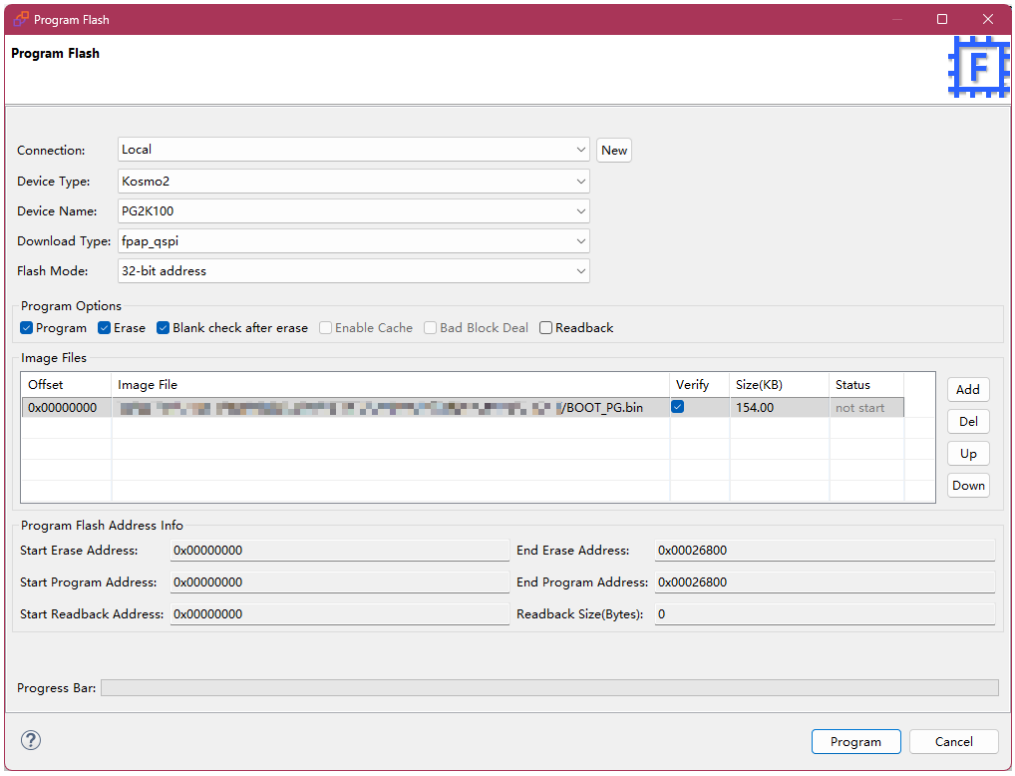


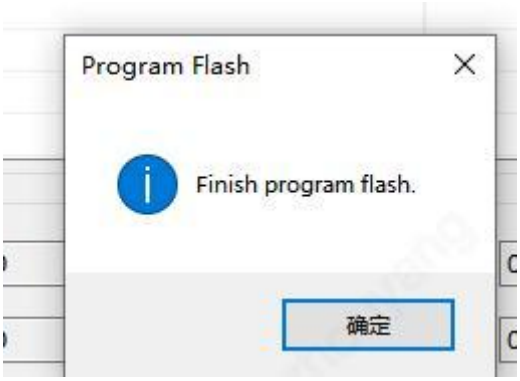
2. 添加 image file。



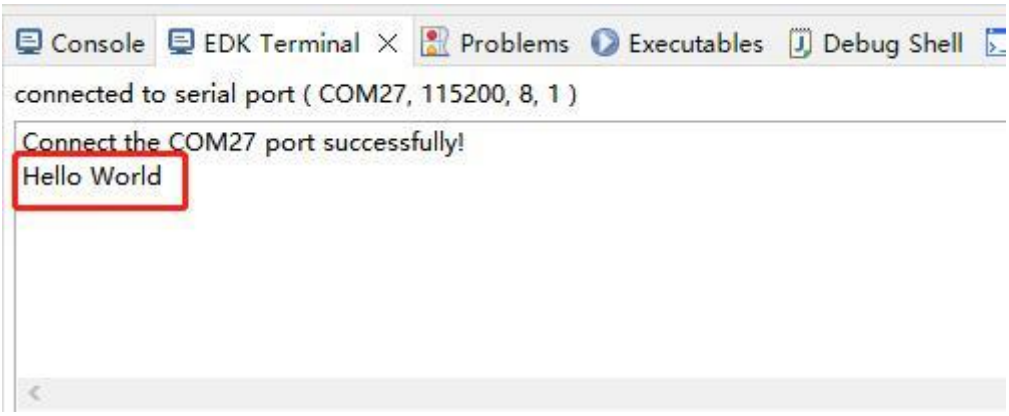
3. 选择制作的 BOOT_PG.bin 文件

4. 点击“Program”按钮并等待下载结束。





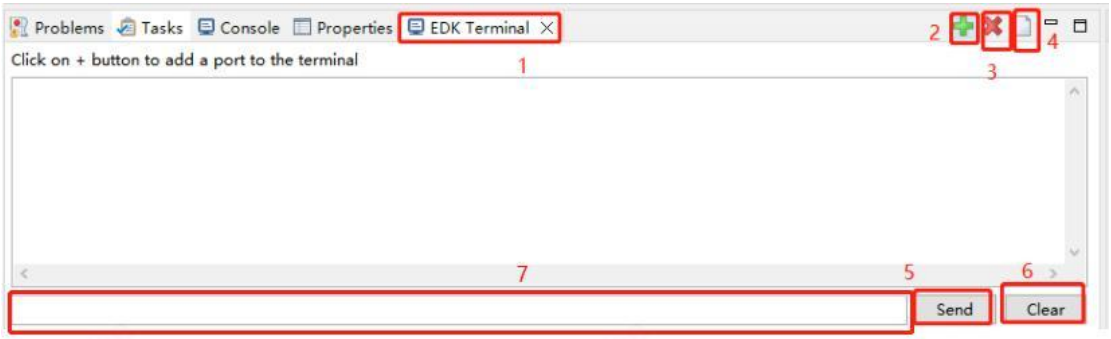
5. 更改启动方式，修改成 QSPI 启动。通过 EDK 串口工具可以看到程序结果



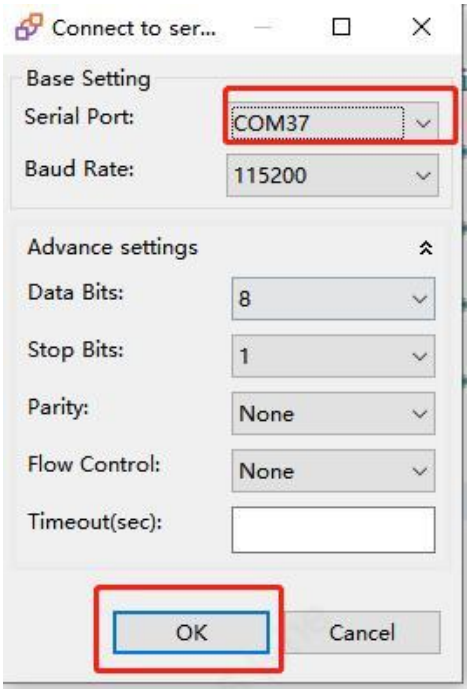
3.6.EDK 串口工具的使用

EDK Terminal 串口调试助手，旨在使用串口打印工程调试信息， 辅助用户调试。

EDK Terminal 如下图所示。



点击上图标 1 处会出现串口相关的界面；点击图中标 2 处进入串口连接配置界面，配置对应的端口号，然后点击“OK”按钮；



点击图中标 3 处断开串口的连接；

点击图中标 4 处清除串口打印页面的信息；

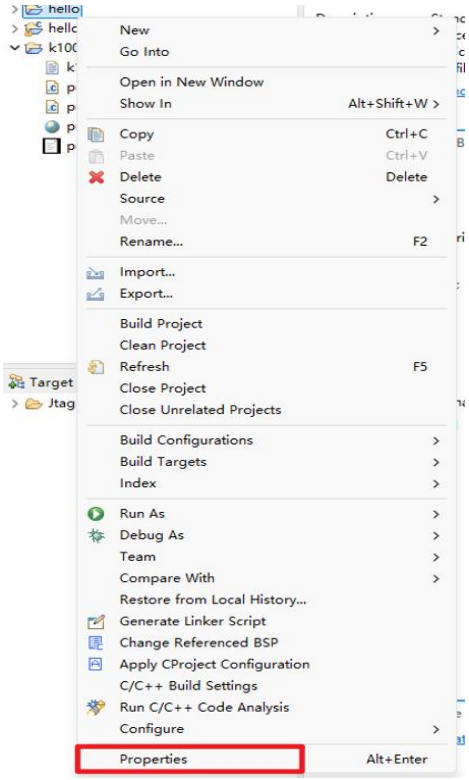
点击图中标 5 处，会将 7 处输入栏中输入的信息发送给串口；

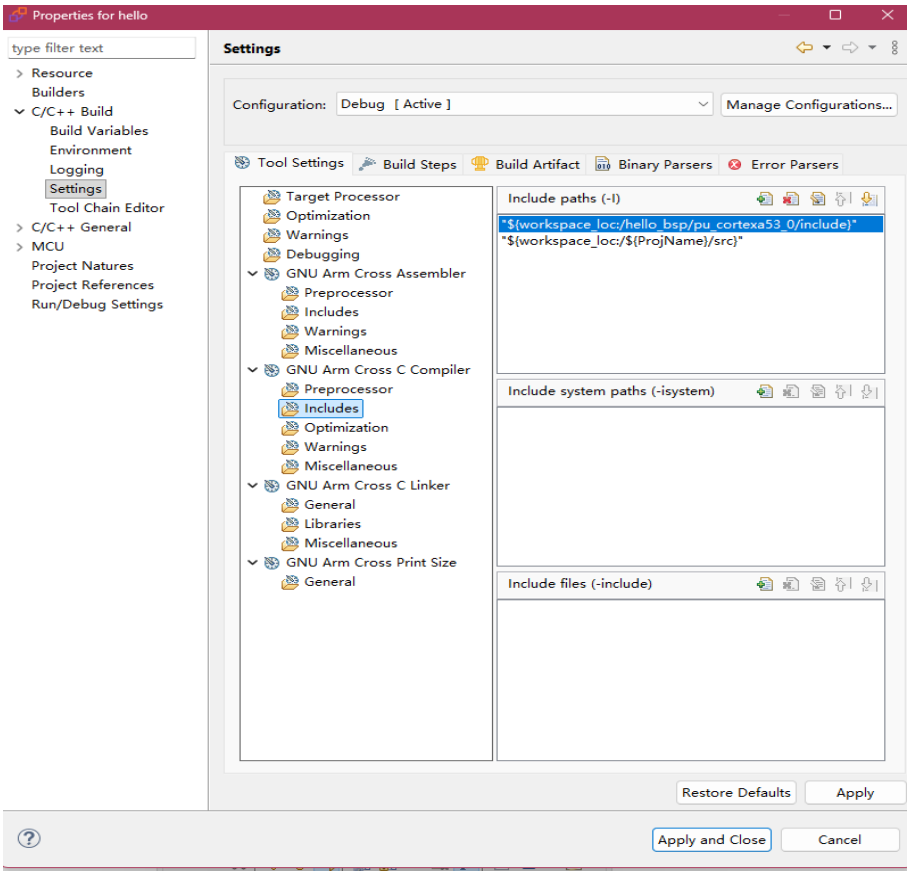
点击图中标 6 处可以清除 7 处输入栏中输入的信息；

图中标 7 处为输入栏，可以输入发送信息。

4.DEMO 工程说明

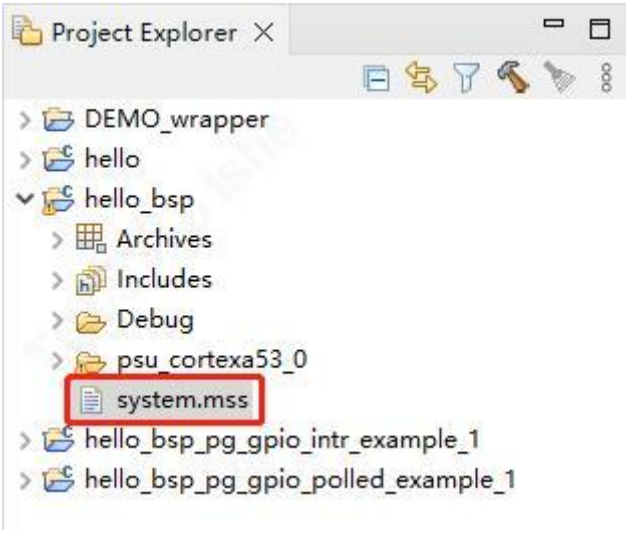
给出来的 demo 工程， 默认只包含两个头文件目录， 分别为该工程中的 src 目录和 bsp 中的 include 目录， 可通过如下图所示的操作查看默认包含的头文件目录。



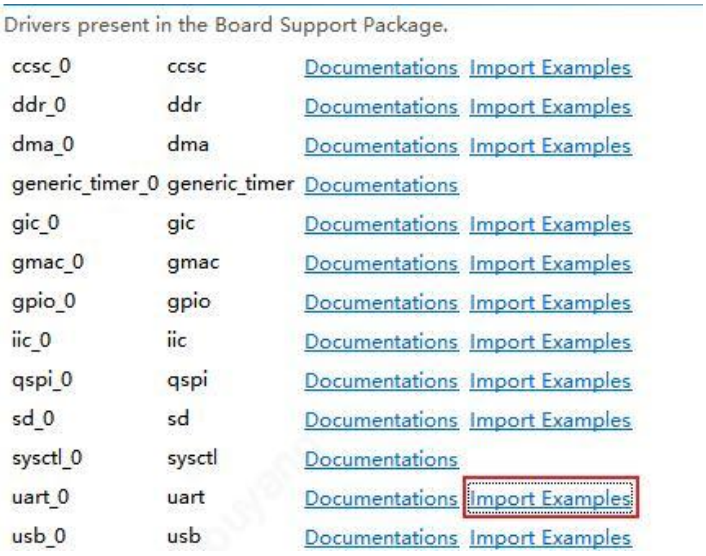


4.1.UART DEMO 工程

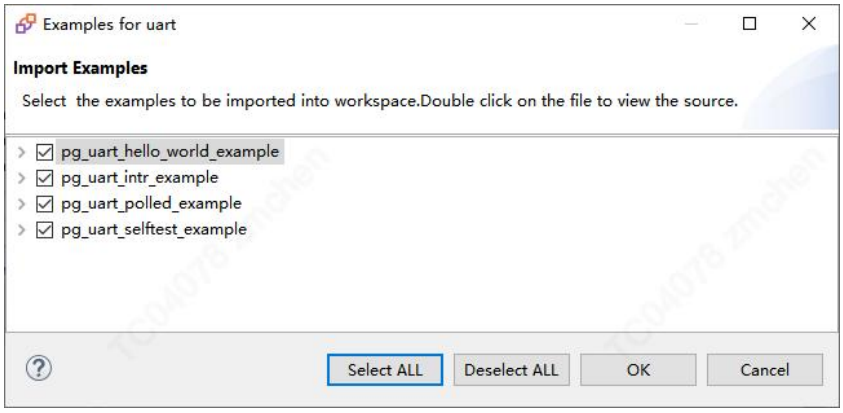
1.点击 BSP 中的 mss 文件。



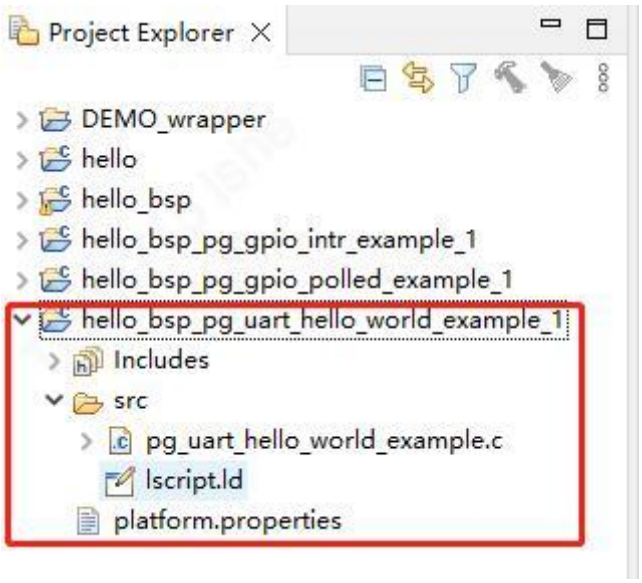
点击 uart_0 的 Import_Examples。



3.勾选如下图所示，并点击 OK 按键。



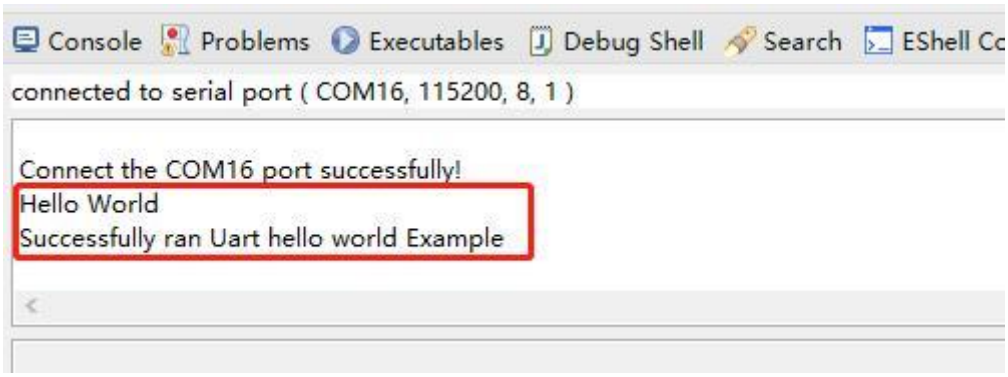
4. Project Explorer 中生成如下图红框中的文件



4.1.1.pg_uart_hello_world_example

1.编译运行。

2.运行结果，串口输出如下图所示。



4.1.2.pg_uart_intr_example

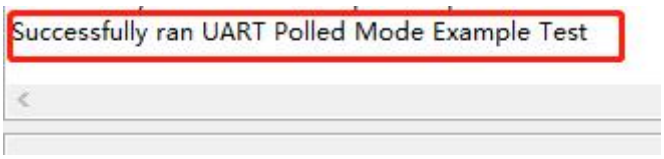
1.编译运行。

2.运行结果，串口将输出如下图所示结果。

4.1.3.pg_uart_polled_example

1.编译运行。

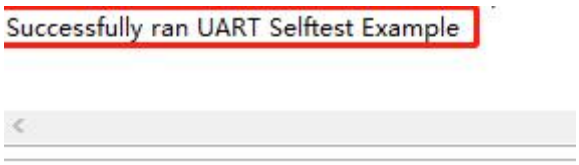
2.运行结果，串口将输出如下图所示结果。



4.1.4.pg_uart_selftest_example

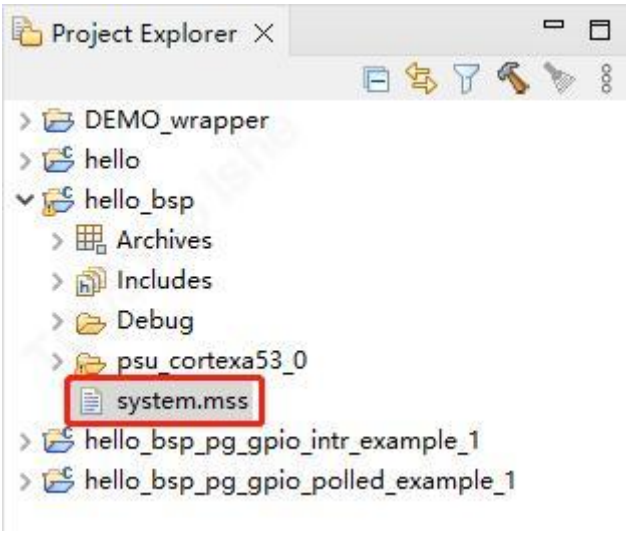
1.编译运行。

2.运行结果，串口将输出如下图所示结果。

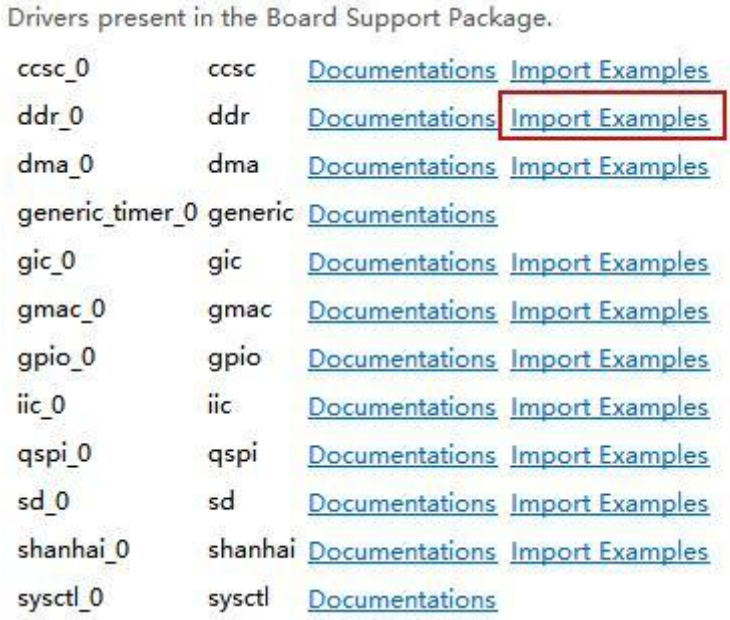


4.2.DDR DEMO 工程

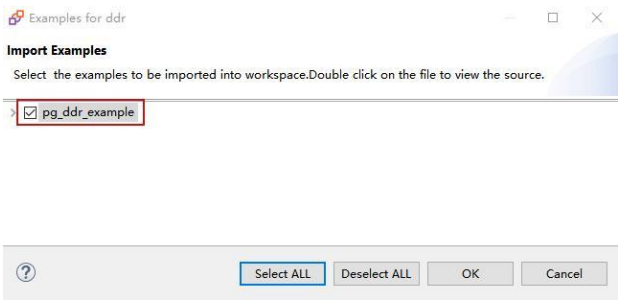
点击 BSP 中的 mss 文件；



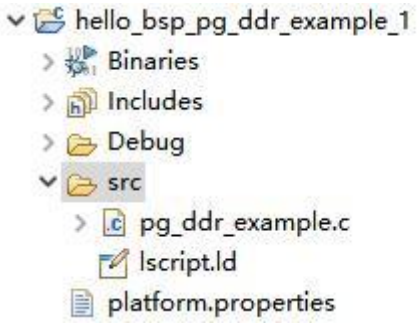
点击 ddr_0 的 Import_Examples；



勾选如下图所示，并点击 OK 按键；

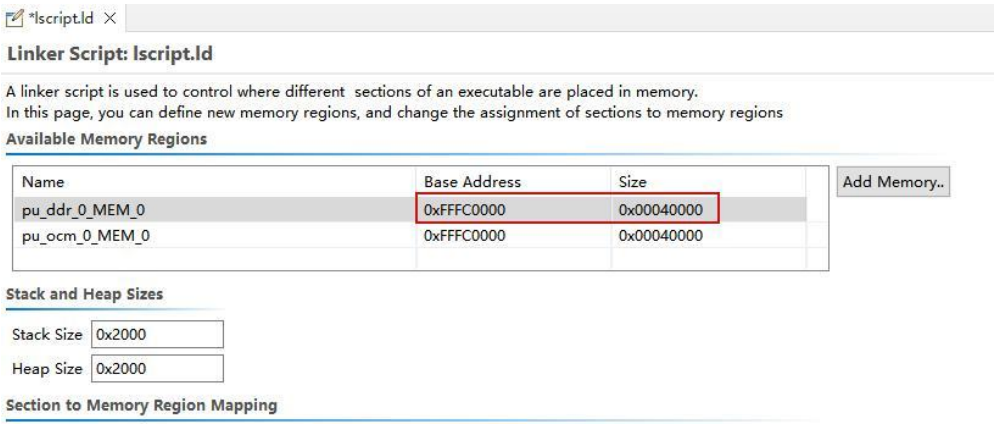


Project Explorer 中生成如下图红框中的文件；



4.2.1.pg_ddr_example

1.选中 src 目录下的 lscript.ld 文件，修改 pu_ddr_0_MEM_0 地址段起始为 0xFFFC0000，size 为 0x00040000，如下图所示；

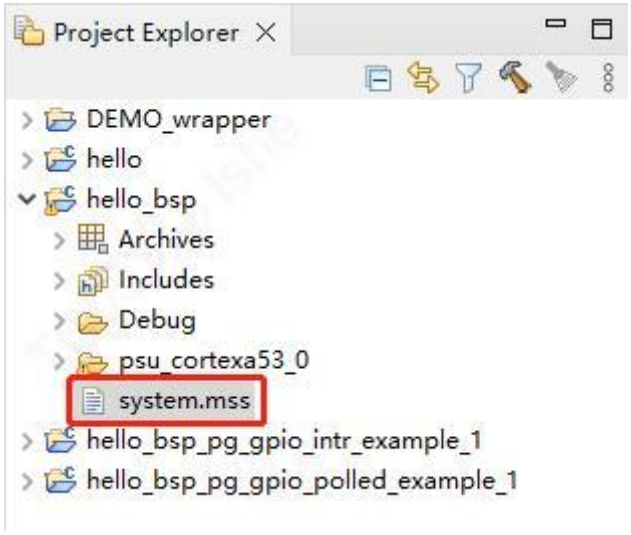


2.编译运行；

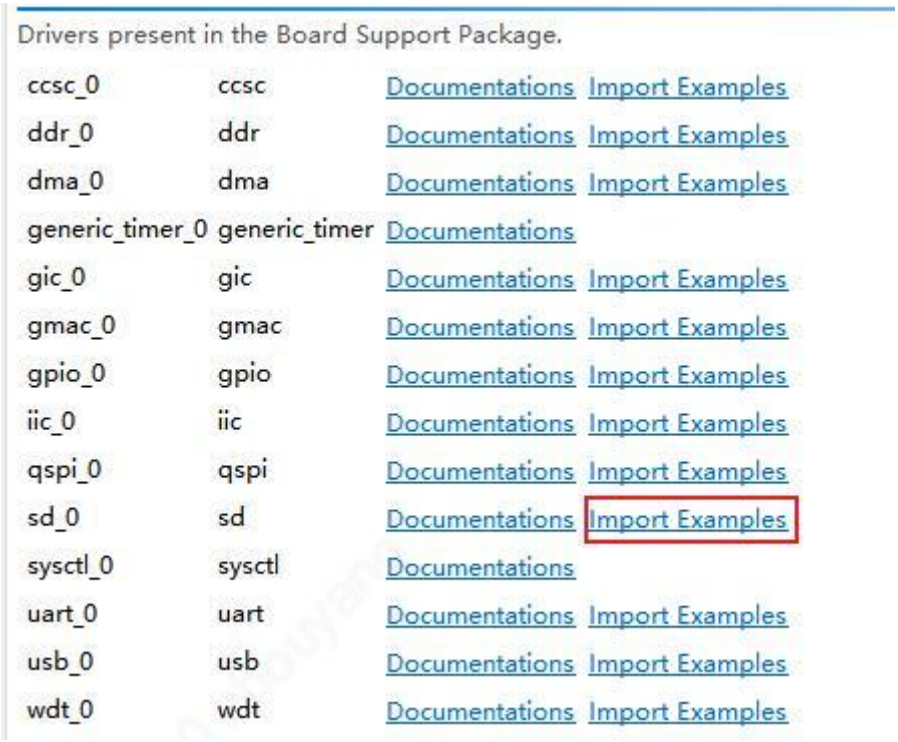
3.运行结果，串口将输出如下图所示的结果；

4.3.SD DEMO 工程

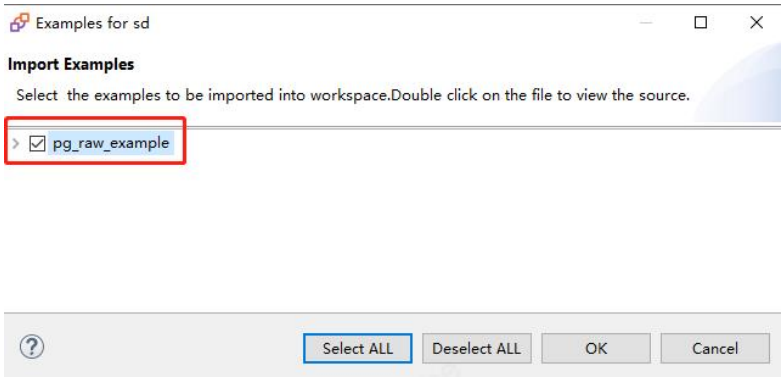
点击 BSP 中的 mss 文件。



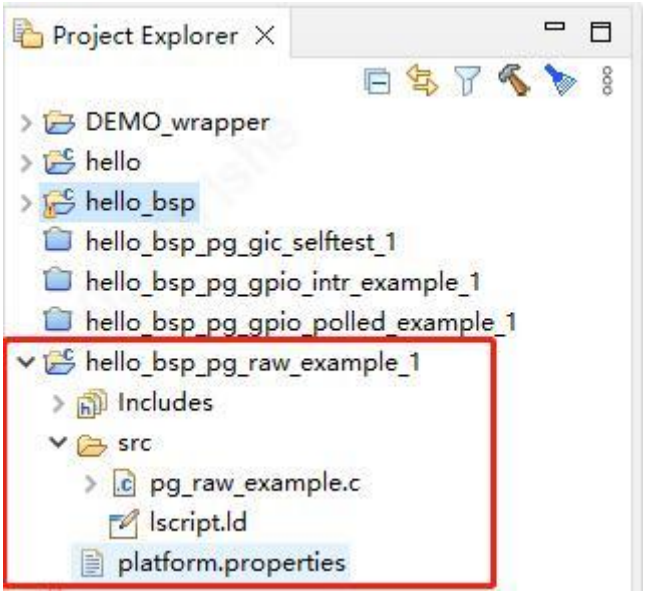
点击 sd_0 的 Import_Examples。



勾选如下图所示， 并点击 OK 按键。



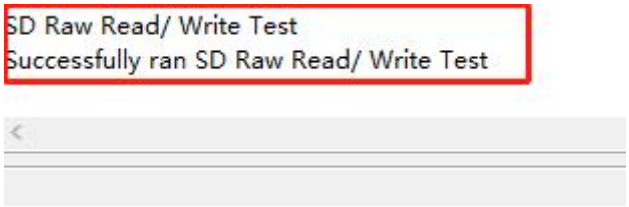
Project Explorer 中生成如下图红框中的文件。



4.3.1.pg_raw_example

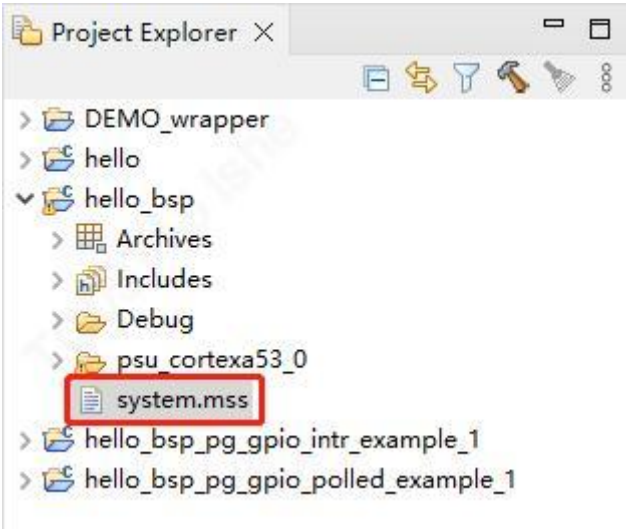
1.编译运行。

2.运行结果，串口将输出如下图所示的结果。

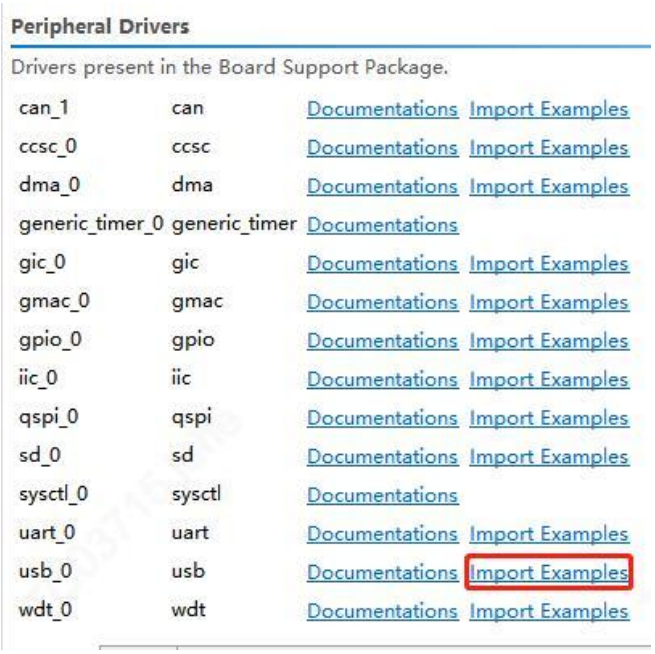


4.4.USB DEMO 工程

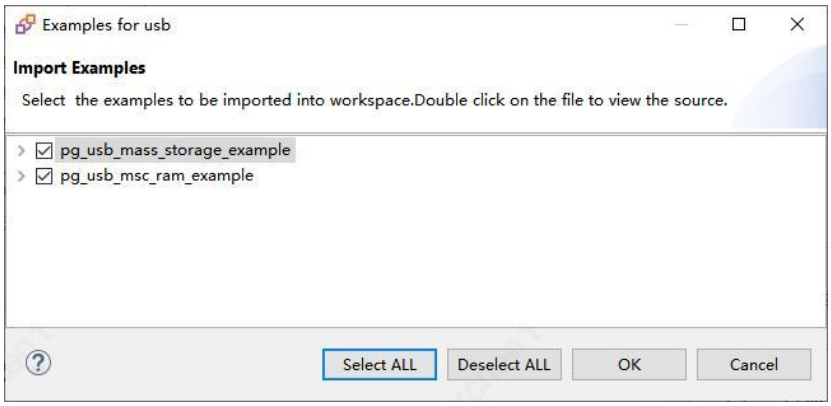
点击 BSP 中的 mss 文件。



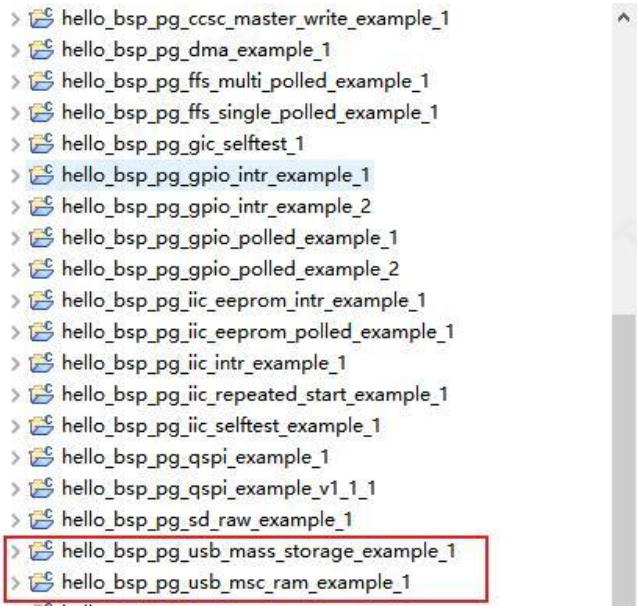
点击 usb_0 的 Import_Examples。



勾选如下图所示， 并点击 OK 按键。



Project Explorer 中生成如下图红框中的文件。



4.4.1.pg_usb_mass_storage_example

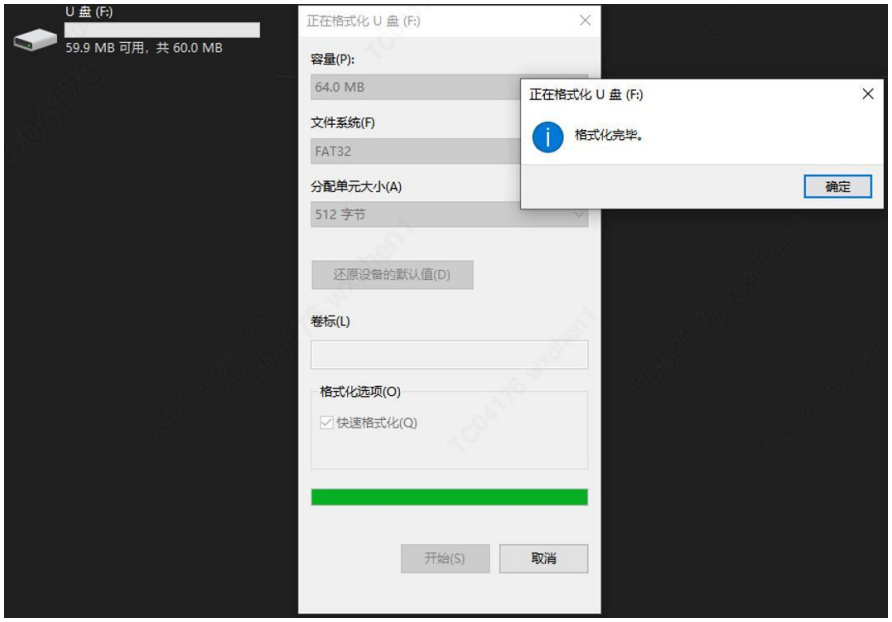
在 demo 板上插上 U 盘，然后选中 Project Explorer 页面中的 pg_usb_mass_storage_example 示例工程。调试工程，编译运行。串口的输出结果，如下图所示。

```
the usb host start to detect the USB device
get_device_descriptor
set_device_address
get_device_descriptor
bnumConfigurations:1
get_configration_descriptor
get_configration_descriptor
configuration buffer 0: 9
configuration buffer 1: 2
configuration buffer 2: 20
configuration buffer 3: 0
configuration buffer 4: 1
configuration buffer 5: 1
configuration buffer 6: 0
configuration buffer 7: 80
configuration buffer 8: 32
configuration buffer 9: 9
configuration buffer 10: 4
configuration buffer 11: 0
configuration buffer 12: 0
configuration buffer 13: 2
configuration buffer 14: 8
configuration buffer 15: 6
configuration buffer 16: 50
configuration buffer 17: 0
configuration buffer 18: 7
configuration buffer 19: 5
configuration buffer 20: 1
configuration buffer 21: 2
configuration buffer 22: 0
configuration buffer 23: 2
configuration buffer 24: 0
configuration buffer 25: 7
configuration buffer 26: 5
configuration buffer 27: 81
configuration buffer 28: 2
configuration buffer 29: 0
configuration buffer 30: 2
configuration buffer 31: 0
set_configration
bnumInterfaces:1
Interfaces buffer 0:9
Interfaces buffer 1:2
Interfaces buffer 2:20
Interfaces buffer 3:0
Interfaces buffer 4:1
Interfaces buffer 5:1
Interfaces buffer 6:0
Interfaces buffer 7:80
Interfaces buffer 8:32
Interfaces buffer 9:9
Interfaces buffer 10:4
Interfaces buffer 11:0
Interfaces buffer 12:0
Interfaces buffer 13:2
Interfaces buffer 14:8
Interfaces buffer 15:6
Interfaces buffer 16:50
Interfaces buffer 17:0
Interfaces buffer 18:7
Interfaces buffer 19:5
Interfaces buffer 20:1
Interfaces buffer 21:2
Interfaces buffer 22:0
Interfaces buffer 23:2
Interfaces buffer 24:0
Interfaces buffer 25:7
Interfaces buffer 26:5
Interfaces buffer 27:81
Interfaces buffer 28:2
Interfaces buffer 29:0
Interfaces buffer 30:2
Interfaces buffer 31:0
bnumEndpoints:2
Endpoints buffer 0:9
Endpoints buffer 1:2
Endpoints buffer 2:20
Endpoints buffer 3:0
Endpoints buffer 4:1
Endpoints buffer 5:1
Endpoints buffer 6:0
Endpoints buffer 7:80
Endpoints buffer 8:32
Endpoints buffer 9:9
Endpoints buffer 10:4
Endpoints buffer 11:0
Endpoints buffer 12:0
Endpoints buffer 13:2
Endpoints buffer 14:8
Endpoints buffer 15:6
Endpoints buffer 16:50
Endpoints buffer 17:0
Endpoints buffer 18:7
Endpoints buffer 19:5
Endpoints buffer 20:1
Endpoints buffer 21:2
Endpoints buffer 22:0
Endpoints buffer 23:2
Endpoints buffer 24:0
Endpoints buffer 25:7
Endpoints buffer 26:5
Endpoints buffer 27:81
Endpoints buffer 28:2
Endpoints buffer 29:0
Endpoints buffer 30:2
Endpoints buffer 31:0
pg_usb test success
```

4.4.2.pg_usb_msc_ram_example

将 demo 板上的 USB 接口连接到可读写的计算机的 USB 接口，然后选中 Project Explorer 页面中的 pg_usb_msc_ram_example 示例工程。调试工程，编译运行。

例程运行后，在计算机的文件管理器会出现一个 U 盘的盘符，并且会有是否需要格式化该盘符的弹窗。右击该盘符，格式化该盘符的文件系统为“FAT32”（如下图所示），其他选项默认，等待格式化完成。



可以向新增的盘符添加文件（如下图所示）或删除文件。

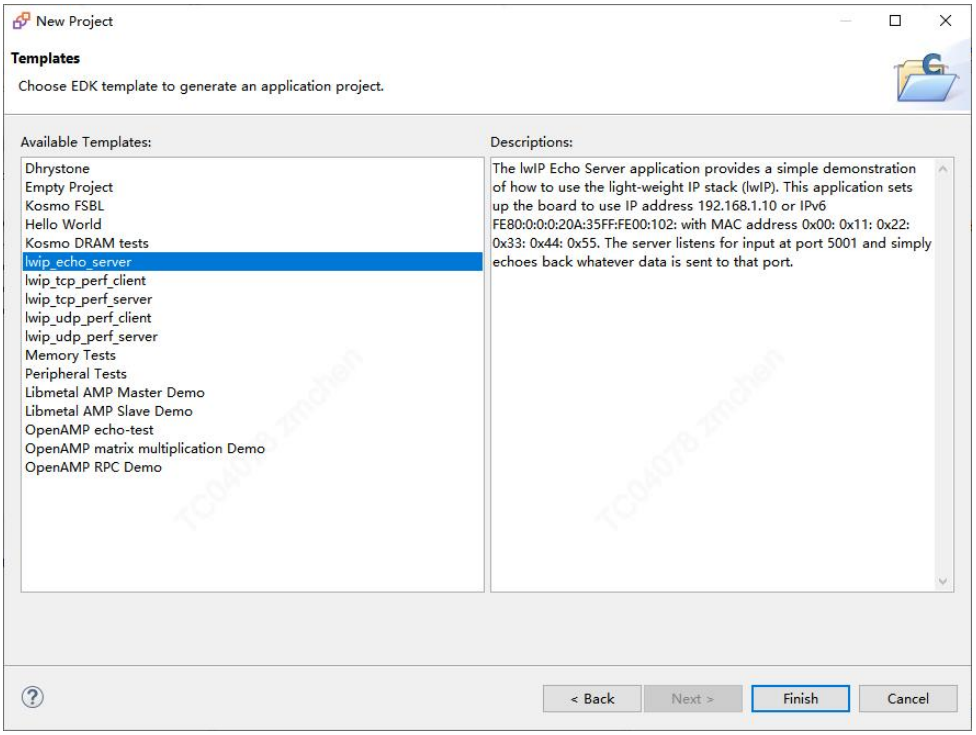
串口的输出结果，如下图所示；

```
□ [32m[I/USB] ========== dwc2 udc params ==========
□ [0m□ [32m[I/USB] CID:00000000
□ [0m□ [32m[I/USB] GSNPSID:4F54400A
□ [0m□ [32m[I/USB] GHWCFG1:00000000
□ [0m□ [32m[I/USB] GHWCFG2:2288F090
□ [0m□ [32m[I/USB] GHWCFG3:039804E8
□ [0m□ [32m[I/USB] GHWCFG4:EE002020
□ [0m□ [32m[I/USB] fsphy type : Not supported
□ [0m□ [32m[I/USB] hsphy type : ULPI
□ [0m□ [32m[I/USB] dma support: Internal DMA
□ [0m□ [32m[I/USB] endpoints  : 13, default config: 12
□ [0m□ [32m[I/USB] dfifo depth: 920 (32-bit words)
□ [0m□ [32m[I/USB] ==================================
□ [0m□ [32m[I/USB] fifo0 size:0010, offset:0100
□ [0m□ [32m[I/USB] fifo1 size:0080, offset:0110
□ [0m□ [32m[I/USB] fifo2 size:0010, offset:0190
□ [0m□ [32m[I/USB] fifo3 size:0010, offset:01A0
□ [0m□ [32m[I/USB] fifo4 size:0010, offset:01B0
□ [0m□ [32m[I/USB] fifo5 size:0010, offset:01C0
□ [0m□ [32m[I/USB] fifo6 size:0000, offset:01D0
□ [0m□ [32m[I/USB] fifo7 size:0000, offset:01D0
□ [0m□ [32m[I/USB] fifo8 size:0000, offset:01D0
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0100, wIndex 0x0000, wLength 0x0040
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x00, bRequest 0x05, wValue 0x0024, wIndex 0x0000, wLength 0x0000
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0100, wIndex 0x0000, wLength 0x0012
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0200, wIndex 0x0000, wLength 0x00FF
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0303, wIndex 0x0409, wLength 0x00FF
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0300, wIndex 0x0000, wLength 0x00FF
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0302, wIndex 0x0409, wLength 0x00FF
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0100, wIndex 0x0000, wLength 0x0012
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0200, wIndex 0x0000, wLength 0x0009
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0200, wIndex 0x0000, wLength 0x0020
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0300, wIndex 0x0000, wLength 0x0002
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0300, wIndex 0x0000, wLength 0x0004
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0303, wIndex 0x0409, wLength 0x0002
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x80, bRequest 0x06, wValue 0x0303, wIndex 0x0409, wLength 0x0016
□ [0m□ [32m[I/USB] Setup: bmRequestType 0x00, bRequest 0x09, wValue 0x0001, wIndex 0x0000, wLength 0x0000
```

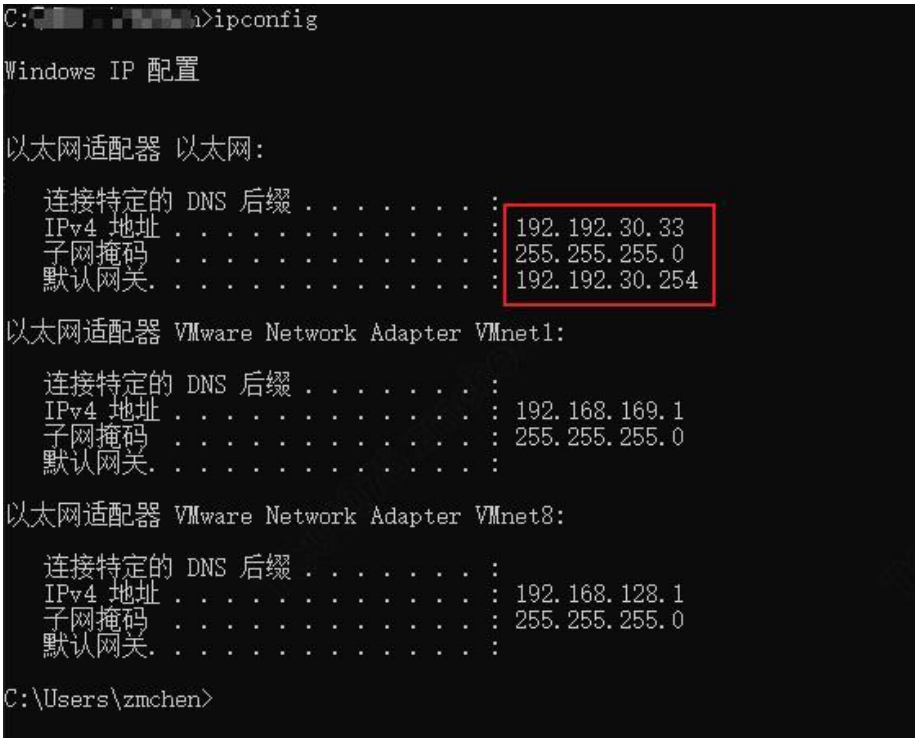
4.5.Lwip DEMO 工程

4.5.1.Lwip_echo_server

1.创建基于 LWIP 模板的 APP，选择 lwip_echo_server；



2.将板子网口和主机端网口相连，查看主机的网段信息，如下所示；



3.修改工程代码，设置板子的网段信息和主机一致，如下所示；

```
160
161     while(((echo_netif->ip_addr.addr) == 0) && (dhcp_timoutcntr > 0))
162         pg_emacif_input(echo_netif);
163
164     if (dhcp_timoutcntr <= 0) {
165         if ((echo_netif->ip_addr.addr) == 0) {
166             pg_printf("DHCP Timeout\r\n");
167             pg_printf("Configuring default IP of 192.192.30.10\r\n");
168             IP4_ADDR(&(echo_netif->ip_addr), 192, 192,  30, 10);
169             IP4_ADDR(&(echo_netif->netmask), 255, 255, 255,  0);
170             IP4_ADDR(&(echo_netif->gw),      192, 168,  128,  1);
171         }
172     }
173
```

4.调试工程，编译运行。

5.运行程序，可以看到串口打印出一些信息，可以看到自动获取到地址为“192.192.30.10”，tcp 端口为 5001；

```
-----lwIP TCP echo server ------
TCP packets sent to port 5001 will be echoed back
pg_gmac_type_gmac
                  Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
phy device 1000, fullduplex
link speed for phy address 2: 1000
set 1000M
close phy loopback
open duplex
set 1000M
close phy loopback
open duplex
init dma success
start.........
DHCP Timeout
Configuring default IP of 192.192.30.10
Board IP: 192.192.30.10
Netmask : 255.255.255.0
Gateway : 192.168.128.1
TCP echo server started @ port 5001
```

打开网络调试工具 NetAssist，设置协议类型为 TCP Client，远程主机地址为获取到的 Board IP“192.192.30.10”，远程主机端口为获取到的 5001，打开调试助手；

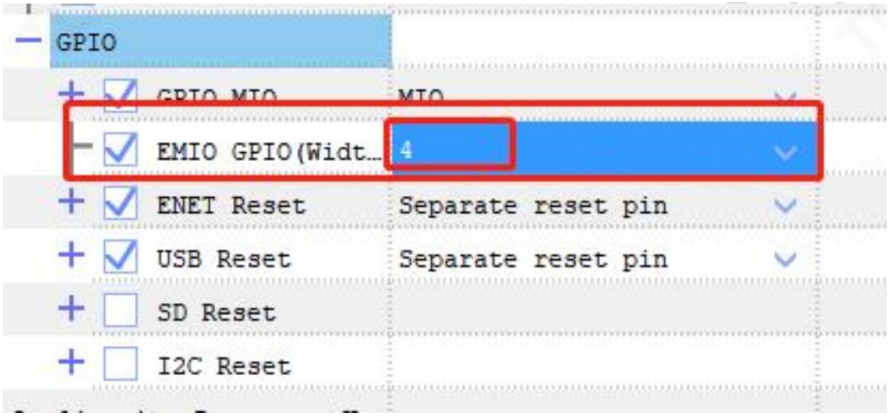
7.点击数据发送，看到 SEND ASCII 和 RECV ACSII 数据均一致；

5.PU 端 EMIO 的使用

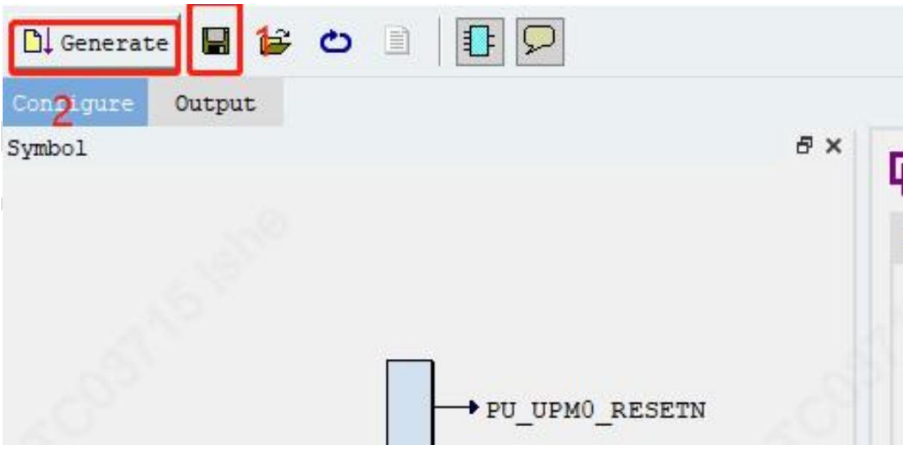
本章节主要介绍采用 EMIO 控制 PA 端的 LED 和 KEY，通过开发板来搭建环境以及做相关的操作。打开 2.1 章节所创建的 PDS 工程，继续修改工程。

5.1.PDS 工程建立

在 IP 配置界面，勾选上 EMIO GPIO（这里使能 EMIO 一定要确保 GPIO MIO 勾上），选择 PA 侧的两个 KEY 和两个 LED，所以 EMIO 的位宽设置为 4。

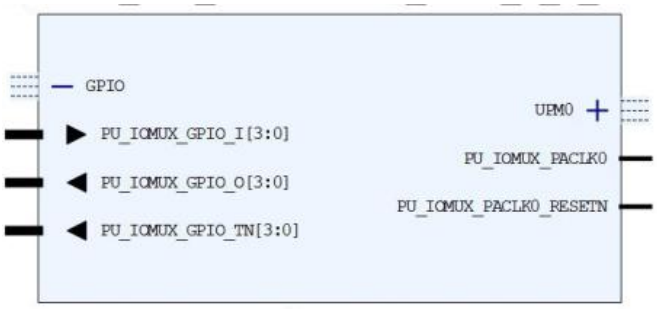


点击保存和生成按钮。

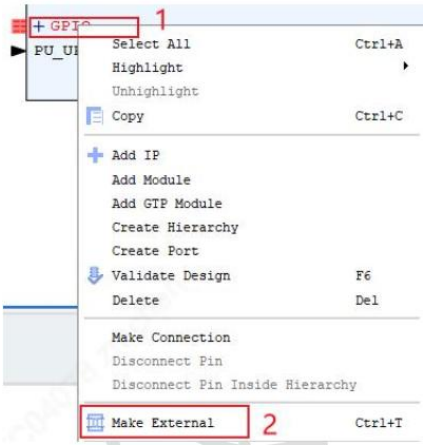



```
Create directory 'edk\' ...
Copy 'edk\pu_init.h.xml' ...
Copy 'edk\pu_init.c.xml' ...
Copy 'edk\pu_init.tcl.xml' ...
Copy 'sopc_wrap.v.xml' ...
Run file 'edk\common_functions.tcl' ...
Run file 'edk\pu_pa_functions.tcl' ...
Run file 'edk\clock_functions.tcl' ...
Run file 'edk\mio_functions.tcl' ...
Run file 'edk\peripherals_functions.tcl' ...
Run file 'edk\pll_functions.tcl' ...
Run file 'edk\ddr_functions.tcl' ...
Run file 'edk\post_config_functions.tcl' ...
Run file 'edk\interrupts_functions.tcl' ...
Compile file 'edk\pu_init.h.xml' to 'edk\pu_init.h' ...
Compile file 'edk\pu_init.c.xml' to 'edk\pu_init.c' ...
Compile file 'edk\pu_init.tcl.xml' to 'edk\pu_init.tcl' ...
Compile file 'sopc_wrap.v.xml' to 'demo_900p_Processing_Unit_1_0_0.v' ...
Found top module 'demo_900p_Processing_Unit_1_0_0' in file 'demo_900p_Processing_Unit_1_0_0.v'.
Create template file 'demo_900p_Processing_Unit_1_0_0_tmpl.v' ...
Create template file 'demo_900p_Processing_Unit_1_0_0_tmpl.vhdl' ...
There is 1 source file to synthesize.
Synthesis is disabled.
Done: 0 error(s), 0 warning(s)
```

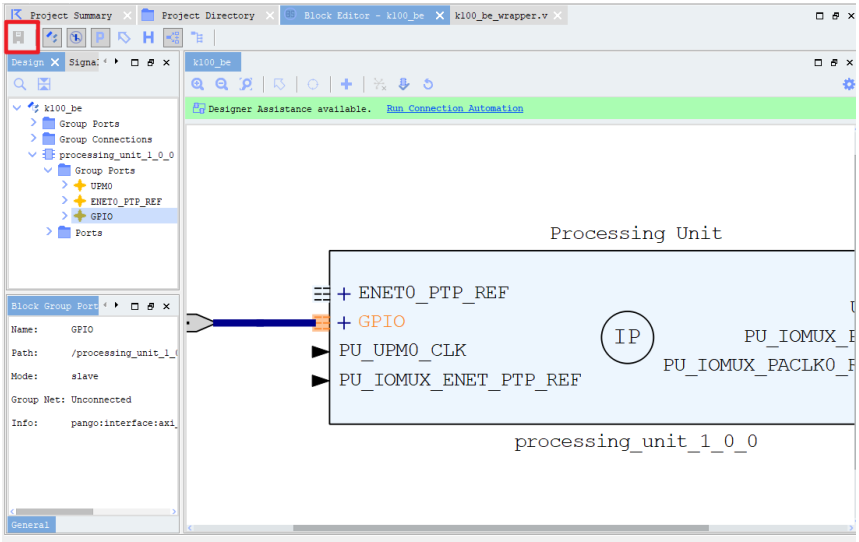
回到主界面， .pbs 文件点击 GPIO 旁边的 + 号， 展开可以看到新增的 emio 信号。

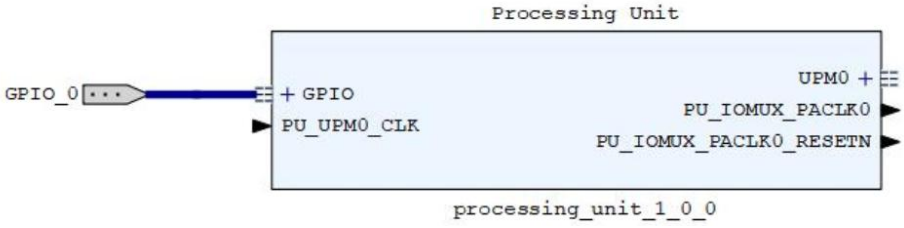


通过右键该信号然后选择“Make External”，新增的三个信号都需要做这样的操作。

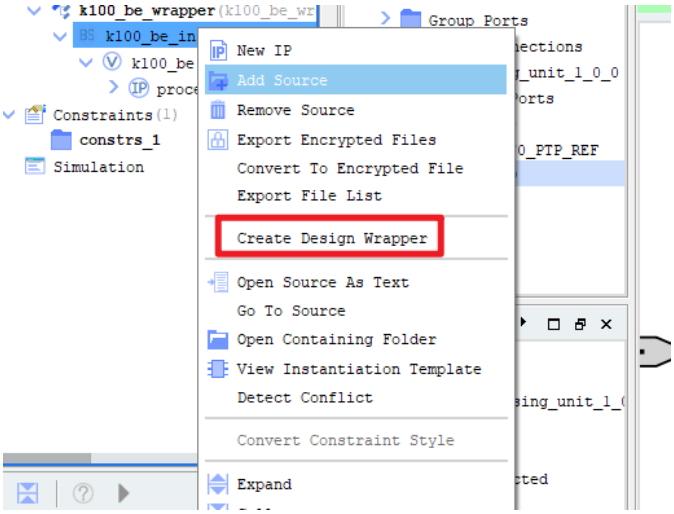


点击保存按钮，保存当前的配置。



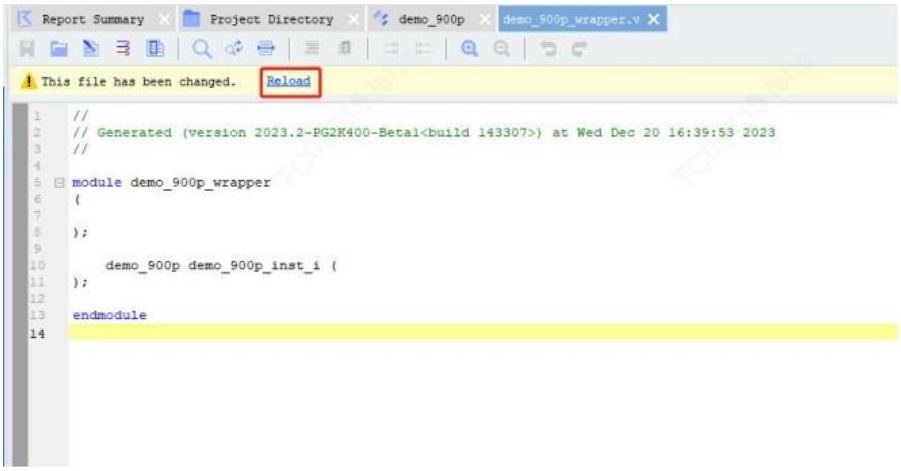


重新生成 pbs 的顶层文件



生成之后如下图所示，

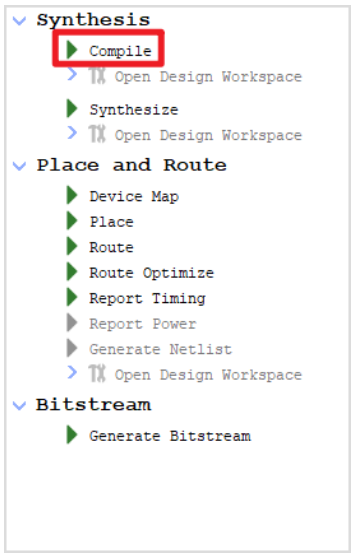
点击“reload”按钮。



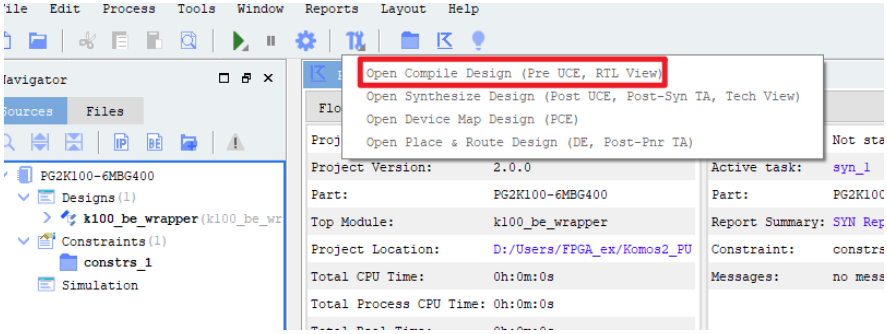
新生成的顶层文件，如下图所示。

```
//
// Generated (version 2024.2<build 176434>) at Mon Jan  6 11:25:24 2025
//

module emio_wrapper
(
    inout [3:0] PU_IOMUX_GPIO_IO
);

    wire [3:0] PU_IOMUX_GPIO_I;
    wire [3:0] PU_IOMUX_GPIO_O;
    wire [3:0] PU_IOMUX_GPIO_TN;


    GTP_IOBUF PU_IOMUX_GPIO_IOBUF_0 (
            .I(PU_IOMUX_GPIO_O[0]),
            .O(PU_IOMUX_GPIO_I[0]),
            .IO(PU_IOMUX_GPIO_IO[0]),
            .T(PU_IOMUX_GPIO_TN[0]));

    GTP_IOBUF PU_IOMUX_GPIO_IOBUF_1 (
            .I(PU_IOMUX_GPIO_O[1]),
            .O(PU_IOMUX_GPIO_I[1]),
            .IO(PU_IOMUX_GPIO_IO[1]),
            .T(PU_IOMUX_GPIO_TN[1]));

    GTP_IOBUF PU_IOMUX_GPIO_IOBUF_2 (
            .I(PU_IOMUX_GPIO_O[2]),
            .O(PU_IOMUX_GPIO_I[2]),
            .IO(PU_IOMUX_GPIO_IO[2]),
            .T(PU_IOMUX_GPIO_TN[2]));

    GTP_IOBUF PU_IOMUX_GPIO_IOBUF_3 (
            .I(PU_IOMUX_GPIO_O[3]),
            .O(PU_IOMUX_GPIO_I[3]),
            .IO(PU_IOMUX_GPIO_IO[3]),
            .T(PU_IOMUX_GPIO_TN[3]));
```

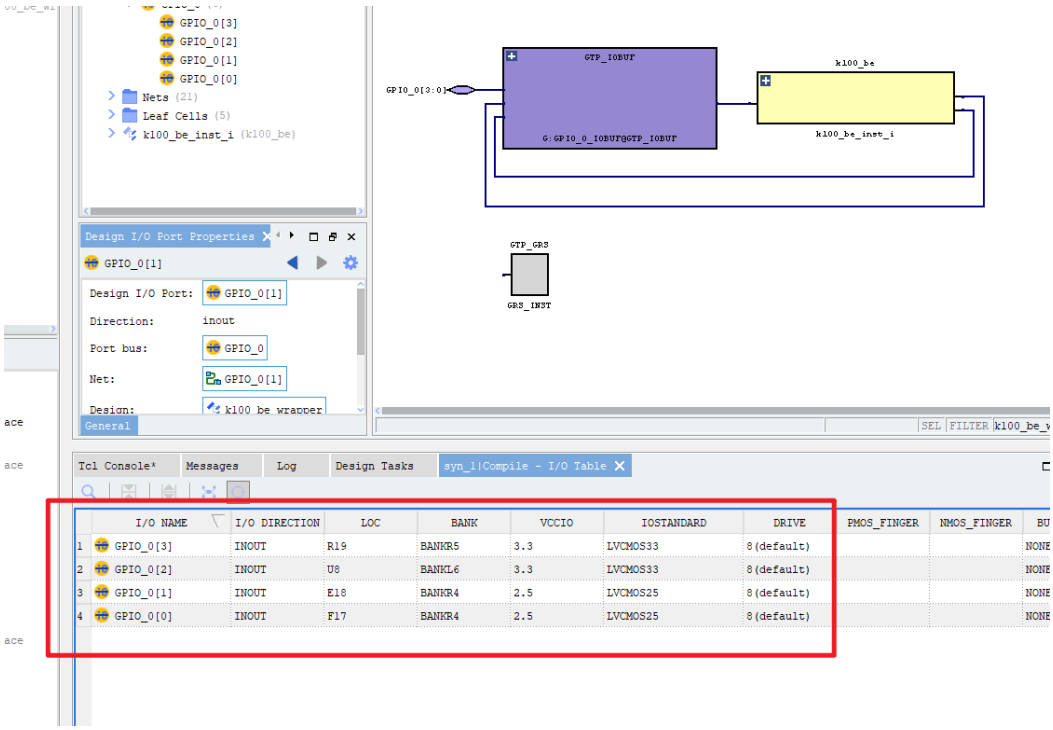
compile 选项被呈现出来，点击“compile”按钮，并等待其结束。



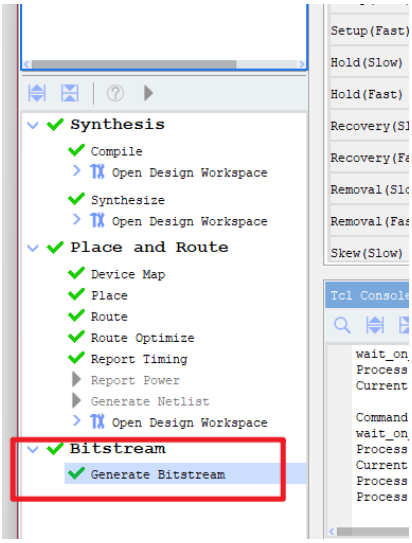
选择如下图工具栏中红框圈中的的按钮并点击“Pre Synthesize UCE”按钮。



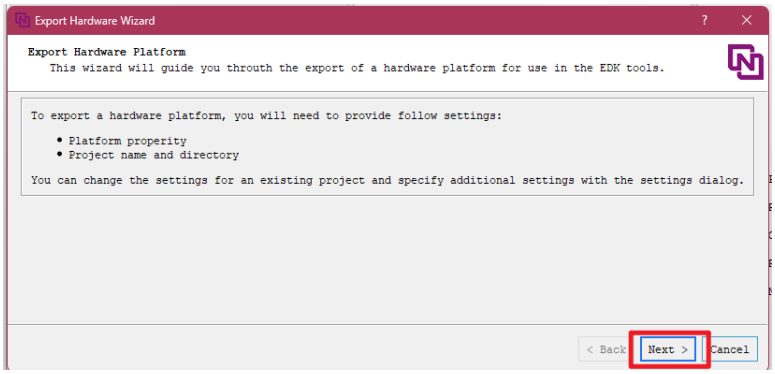
等待 UCE 界面打开。根据电路图配置约束如下图所示，这里的 emio[0-1]对应的为 PA 侧的 LED，emio[2-3]对应的为 PA 侧的 KEY。

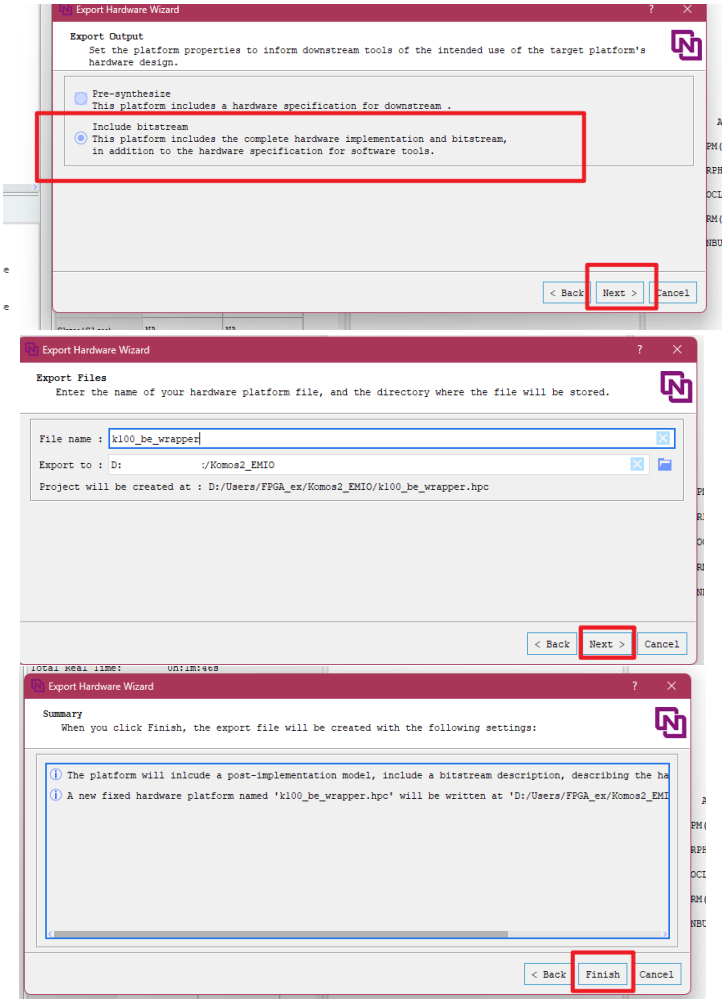


点击如下图所示的“Generate Bitstream”选项，等待其结束，



导出硬件，这里和之前类似，只是额外的选择上了位流选项，首先按照如下图依次点击[File->Export->Export Hardware]。



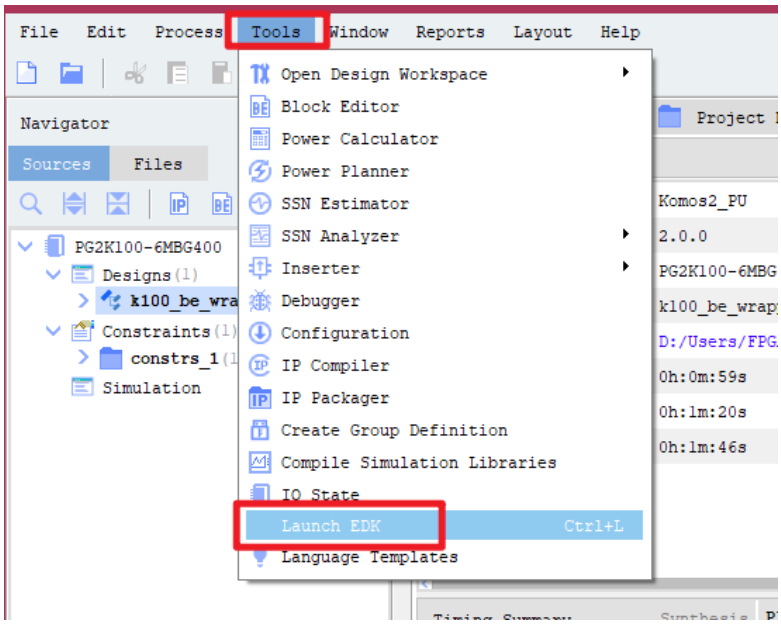


5.2.EDK 程序编写

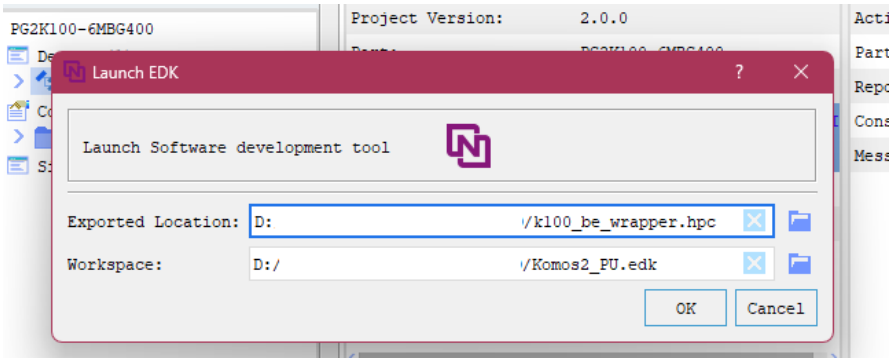
对于 emio 的程序编写，主要是依赖于 PU 侧相关的 example，这里使用的是 emio gpio，所以主要依赖于 gpio 相关的 example，在其基础上添加相关的 IO。

注意 PU 侧的 gpio IP 对应的中断相关的寄存器只能处理 Port A 的 32 个 gpio，而 EMIO 属于 Port C 和 Port D，所以 EMIO 暂不支持使用中断。

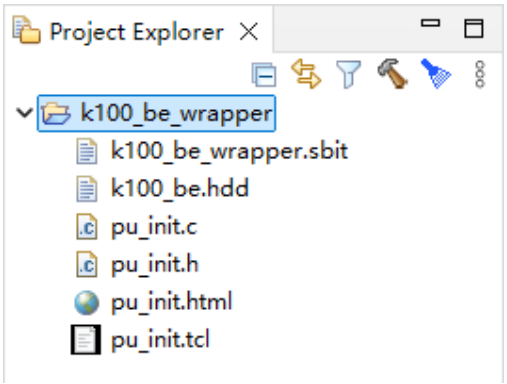
启动 EDK。



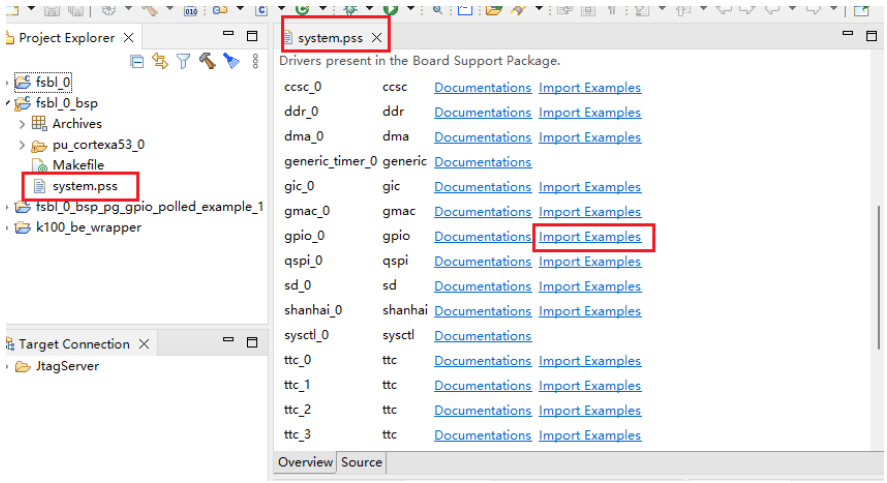
注意点是保持默认这里选择的是新生成的顶层文件。



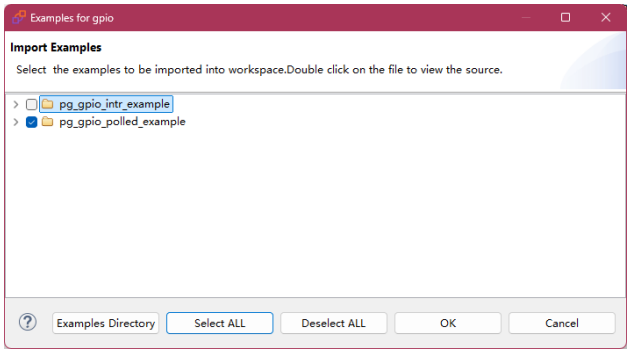
在 EDK 的 Project Explorer 界面会显示新生成平台文件。



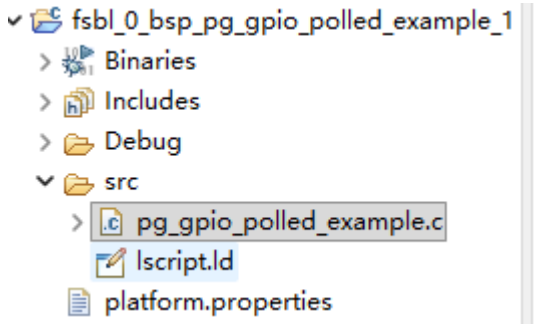
接下来要创建对应的工程文件。



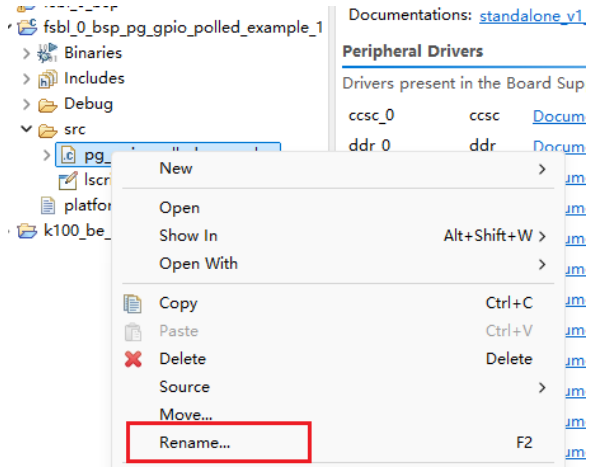
点击弹出窗口，勾选工程点击 ok



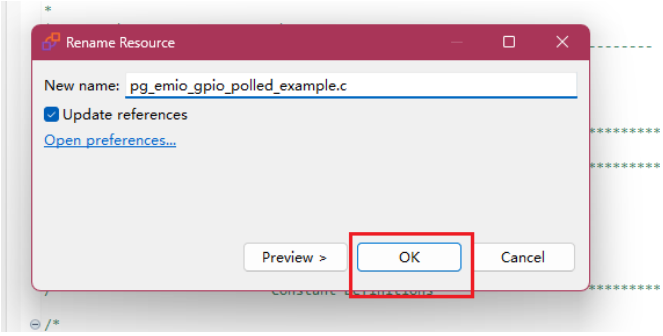
创建对应的一个工程文件， 创建出来如下图所示。



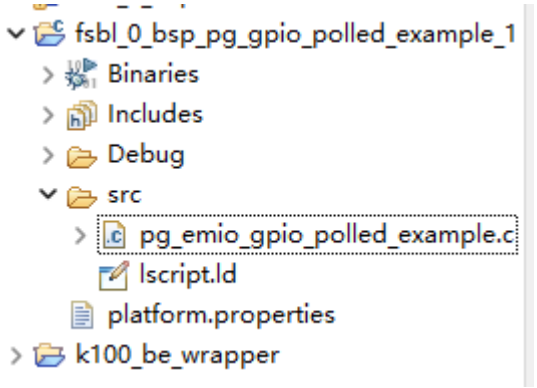
修改主函数的名字，选中主函数，点击右键，然后点击“Rename”



输入修改之后的名字，并点击“OK”按钮。



Project Explorer 界面如下图所示。



代码修改，这里修改的方式主要是将新增的 emio 添加进去。

注意：因为 io 口 0-53 对应的是 MIO，所以 EMIO 是从 54 开始的，约束按照下图所示进行的宏定义

修改这部分宏定义为：

```
#define LED_DELAY 50000000
#define EMIO_KEY0 56
#define EMIO_KEY1 57
#define EMIO_LED0 54
#define EMIO_LED1 55
```

```
/************************ Constant Definitions ***************************/

/*
 * The following constant is used to wait after an LED is turned on to make
 * sure that it is visible to the human eye.  This constant might need to be
 * tuned for faster or slower processor speeds.
 */
#define LED_DELAY 50000000
#define EMIO_KEY0 56
#define EMIO_KEY1 57
#define EMIO_LED0 54
#define EMIO_LED1 55
#define LED_MAX_BLINK 0x10 /* number of times the LED Blinks */

/************************** Type Definitions *****************************/
```

同时添加静态全局变量

static volatile u32 g_input_pin; /* Switch button */
static volatile u32 g_input_pin1;
static volatile u32 g_output_pin; /* LED button */
static volatile u32 g_output_pin1;

```
/************************** Variable Definitions *************************/
static volatile u32 g_input_pin; /* Switch button */
static volatile u32 g_input_pin1;
static volatile u32 g_output_pin; /* LED button */
static volatile u32 g_output_pin1;

/*
 * The following are declared globally so they are zeroed and can be
 * easily accessible from a debugger.
```

对于 EMIO 相关的操作， 由于通过 PU 侧的控制器控制 PA 侧的外设， 所以在导入了 example 之后首先需要将移位寄存器使能。在 mian 函数中添加：

REG32(0xE5006198) = 0xF;

```
#ifndef PG_TESTAPP_CREATE
s32 main(void)
{
    s32 status;
    u32 input_data;
    REG32(0xE5006198) = 0xF;
    pg_printf("GPIO Polled Mode Example Test \r\n");

    status = pg_gpio_polled_example(PG_PAR_GPIO_0_DEVICE_ID, &input_data);
    if (status != PG_SUCCESS) {
        pg_printf("GPIO Polled Mode Example Test Failed\r\n");
        return PG_FAILURE;
    }
```

在 pg_gpio_polled_example 函数中添加

g_input_pin = EMIO_KEY0;
g_input_pin1 = EMIO_KEY1;
g_output_pin = EMIO_LED0;
g_output_pin1 = EMIO_LED1;

（需要删除 g_input_pin = 10; g_output_pin = 0;这部分）

```
s32 pg_gpio_polled_example(u16 device_id, u32 *data_read)
{
    s32 status;
    pg_gpio_config *config_ptr = NULL;

    /* Initialize the GPIO driver. */
    config_ptr = pg_gpio_lookup_config(device_id);

    g_input_pin = EMIO_KEY0;
    g_input_pin1 = EMIO_KEY1;
    g_output_pin = EMIO_LED0;
    g_output_pin1 = EMIO_LED1;

    status = pg_gpio_cfg_initialize(&g_gpio, config_ptr, config_ptr->base_addr);
    if (status != PG_SUCCESS) {
        return PG_FAILURE;
    }
    /* Run the output Example. */
    status = gpio_output_example();
    if (status != PG_SUCCESS) {
        return PG_FAILURE;
    }
```

修改 **gpio_output_example** 函数为下图所示：

```
static s32 gpio_output_example(void)
{
    volatile s32 delay;
    u32 led_loop;
    pg_gpio_set_direction_pin(&g_gpio, g_output_pin, 1);
    pg_gpio_set_direction_pin(&g_gpio, g_output_pin1, 1);
    /* set the GPIO output to be low. */
    pg_gpio_write_pin(&g_gpio, g_output_pin, 0x0);
    pg_gpio_write_pin(&g_gpio, g_output_pin1, 0x0);

    for (led_loop = 0; led_loop < LED_MAX_BLINK; led_loop++) {
        for (delay = 0; delay < LED_DELAY; delay++) {
        }
        pg_gpio_write_pin(&g_gpio, g_output_pin, 0x1);
        /* Wait a small amount of time so the LED is visible. */
        for (delay = 0; delay < LED_DELAY; delay++) {
        }
        /* Clear the GPIO output. */
        pg_gpio_write_pin(&g_gpio, g_output_pin, 0x0);
        /* Wait a small amount of time so the LED is visible. */
        for (delay = 0; delay < LED_DELAY; delay++) {
        }
        /* set the GPIO output to High. */
        pg_gpio_write_pin(&g_gpio, g_output_pin1, 0x1);

        /* Wait a small amount of time so the LED is visible. */
        for (delay = 0; delay < LED_DELAY; delay++) {
        }
        /* Clear the GPIO output. */
        pg_gpio_write_pin(&g_gpio, g_output_pin1, 0x0);
    }
    return PG_SUCCESS;
```

```
}
```

```
static s32 gpio_output_example(void)
{
    volatile s32 delay;
    u32 led_loop;

    /*
     * set the direction for the pin to be output and
     * Enable the output enable for the LED Pin.
     */
    pg_gpio_set_direction_pin(&g_gpio, g_output_pin, 1);
    pg_gpio_set_direction_pin(&g_gpio, g_output_pin1, 1);
    /* set the GPIO output to be low. */
    pg_gpio_write_pin(&g_gpio, g_output_pin, 0x0);
    pg_gpio_write_pin(&g_gpio, g_output_pin1, 0x0);

    for (led_loop = 0; led_loop < LED_MAX_BLINK; led_loop++) {
        /* Wait a small amount of time so the LED is visible. */
        for (delay = 0; delay < LED_DELAY; delay++) {
        }

        /* set the GPIO output to High. */
        pg_gpio_write_pin(&g_gpio, g_output_pin, 0x1);

        /* Wait a small amount of time so the LED is visible. */
        for (delay = 0; delay < LED_DELAY; delay++) {
        }

        /* Clear the GPIO output. */
        pg_gpio_write_pin(&g_gpio, g_output_pin, 0x0);

        /* Wait a small amount of time so the LED is visible. */
        for (delay = 0; delay < LED_DELAY; delay++) {
        }

        /* set the GPIO output to High. */
        pg_gpio_write_pin(&g_gpio, g_output_pin1, 0x1);

        /* Wait a small amount of time so the LED is visible. */
        for (delay = 0; delay < LED_DELAY; delay++) {
        }

        /* Clear the GPIO output. */
        pg_gpio_write_pin(&g_gpio, g_output_pin1, 0x0);
    }

    return PG_SUCCESS;
}
```

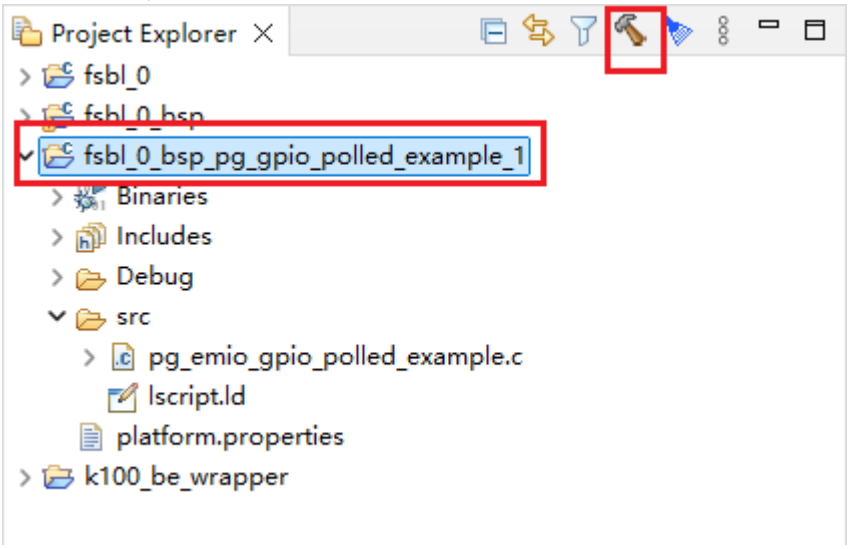
最后在 gpio_input_example 函数中修改为：

```
static s32 gpio_input_example(u32 *data_read)
{
    if (NULL == data_read) {
        return PG_FAILURE;
    }
    /* set the direction for the specified pin to be input. */
    pg_gpio_set_direction_pin(&g_gpio, g_input_pin, 0x0);
    pg_gpio_set_direction_pin(&g_gpio, g_input_pin1, 0x0);
    /* Read the state of the data so that it can be verified. */
    pg_printf("please press board KEY3!  \r\n");
    *data_read = pg_gpio_read_pin(&g_gpio, g_input_pin);
    while (*data_read != 0) {
    *data_read = pg_gpio_read_pin(&g_gpio, g_input_pin);
    }
    pg_printf("KEY3 pressed, key3 read %08x\r\n", *data_read);
    pg_printf("please press board KEY4!  \r\n");
    *data_read = pg_gpio_read_pin(&g_gpio, g_input_pin1);
    while (*data_read != 0)
    {
```

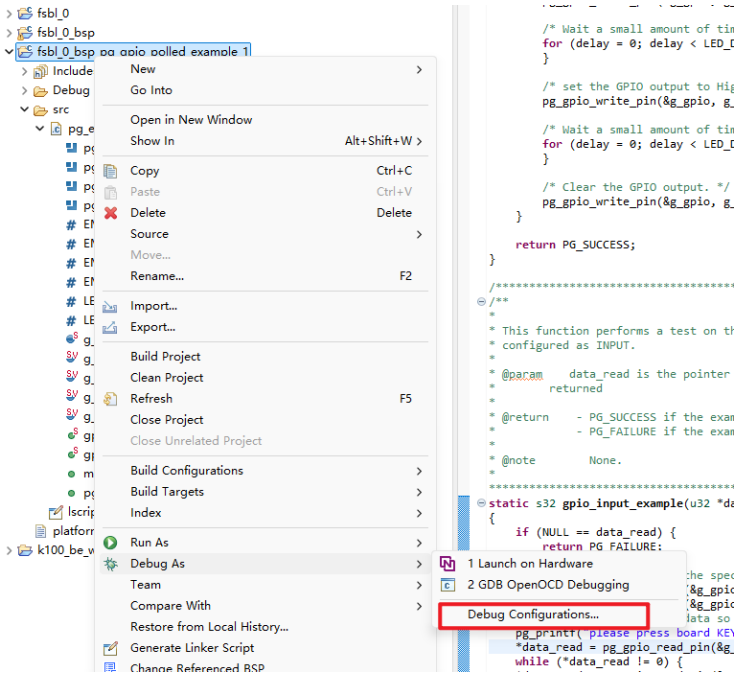
```
    *data_read = pg_gpio_read_pin(&g_gpio, g_input_pin1);
    }
    pg_printf("KEY4 pressed, key4 read %08x\r\n", *data_read);

    return PG_SUCCESS;
}
```

```
static s32 gpio_input_example(u32 *data_read)
{
    if (NULL == data_read) {
        return PG_FAILURE;
    }
    /* set the direction for the specified pin to be input. */
    pg_gpio_set_direction_pin(&g_gpio, g_input_pin, 0x0);
    pg_gpio_set_direction_pin(&g_gpio, g_input_pin1, 0x0);
    /* Read the state of the data so that it can be verified. */
    pg_printf("please press board KEY3: \r\n");
    *data_read = pg_gpio_read_pin(&g_gpio, g_input_pin);
    while (*data_read != 0) {
    *data_read = pg_gpio_read_pin(&g_gpio, g_input_pin);
    }
    pg_printf("KEY3 pressed, key3 read %08x\r\n", *data_read);
    pg_printf("please press board KEY4: \r\n");
    *data_read = pg_gpio_read_pin(&g_gpio, g_input_pin1);
    while (*data_read != 0)
    {
        *data_read = pg_gpio_read_pin(&g_gpio, g_input_pin1);
    }
    pg_printf("KEY4 pressed, key4 read %08x\r\n", *data_read);

    return PG_SUCCESS;
}
```

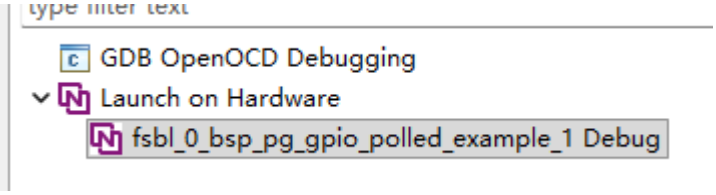
修改完成，构建 Project。



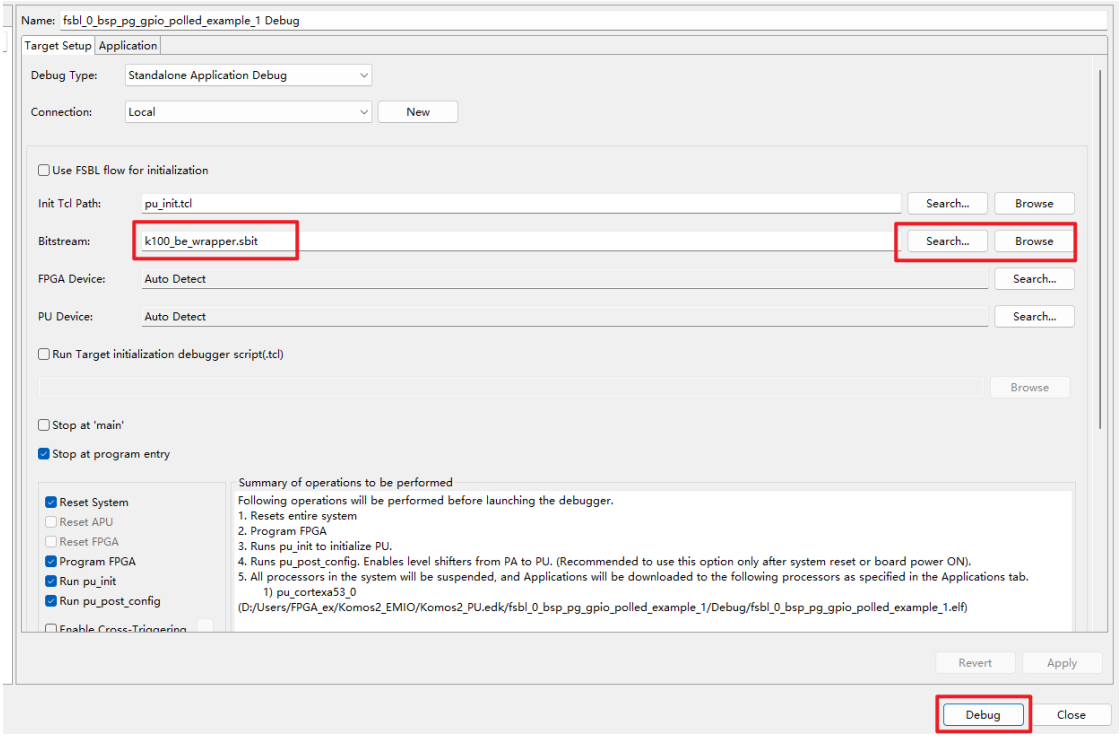
调试 project，因为这里需要下载位流，所以需要“debug configure”选项，按照如下图所示进入这个工程的“debug configure”页面。



双击“Launch on Hardware”。



生成如下图界面，点击位流选项中的“Browse”按钮，找到 FPGA 生成的位流文件，存放的位置为平台文件夹下面。



运行结果，程序运行之后两个 LED 会接续的闪烁， 串口会有打印信息， 当 LED 熄灭时需要按下 PA_KEY2 程序才会接着运行， 之后需要按下 PA_KEY1 程序才会接着运行， 最终串口的打印信息如下图所示。

```
GPIO Polled Mode Example Test
please press board KEY3!
KEY3 pressed, key3 read 00000000
please press board KEY4!
KEY4 pressed, key4 read 00000000
data read from GPIO input is  0x0
Successfully ran GPIO Polled Mode Example Test
```

6.PU 端 Linux 启动使用

PG2K100 启动一般最少包含 Stage0 和 Stage1， 一般在跑系统的情况下才会使用到 Stage2，其主要的作用介绍如下：

Stage0：

在上电复位或者热复位之后， 处理器首先执行 BootRom 里的代码， 这一步是最初始启动设置。 BootRom 是在非 JTAG 模式下才会被执行， 主要存放了一段用户不可更改的代码。 其主要作用有：

1. 包含了最基本的 NAND， Quad-SPI， SD 驱动；
2. 把 FSBL 的代码搬运到 OCM 之中（即 stage 1 的代码） ， 空间限制为 192K B。

Stage1：

当 BootRom 搬运 FSBL 到 OCM 后， 处理器开始执行 FSBL 代码， FSBL 主要有以下作用：

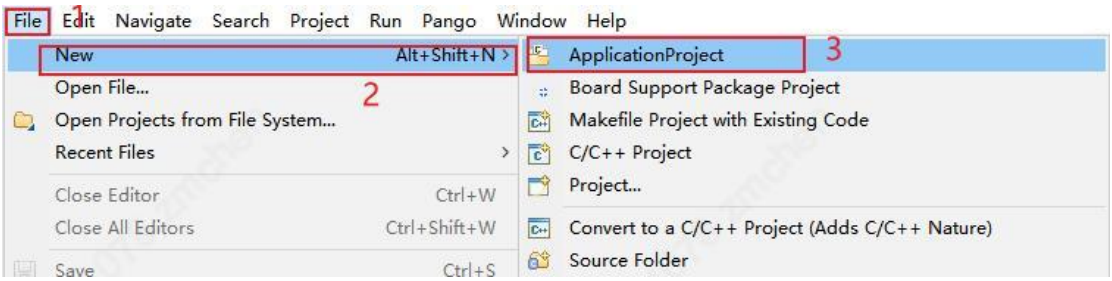
- 初始化 PU 端的配置， 主要包含 PDS 中相关的配置。 包括初始化 DDR， MIO， SCLR， 时钟等。 主要执行 pu_init.c， 其中 pu_init.tcl 文件一般用作于在线调试时候配置硬件环境， 其执行效果和 pu_init.c 是一样的；
- 若存在 PA 端程序， 加载 PA 端 bitstream；
- 加载 Stage2 或者 bare-metal 应用程序到 DDR；
- 跳转到 Stage2 或者 bare-metal 应用程序。

Stage2：

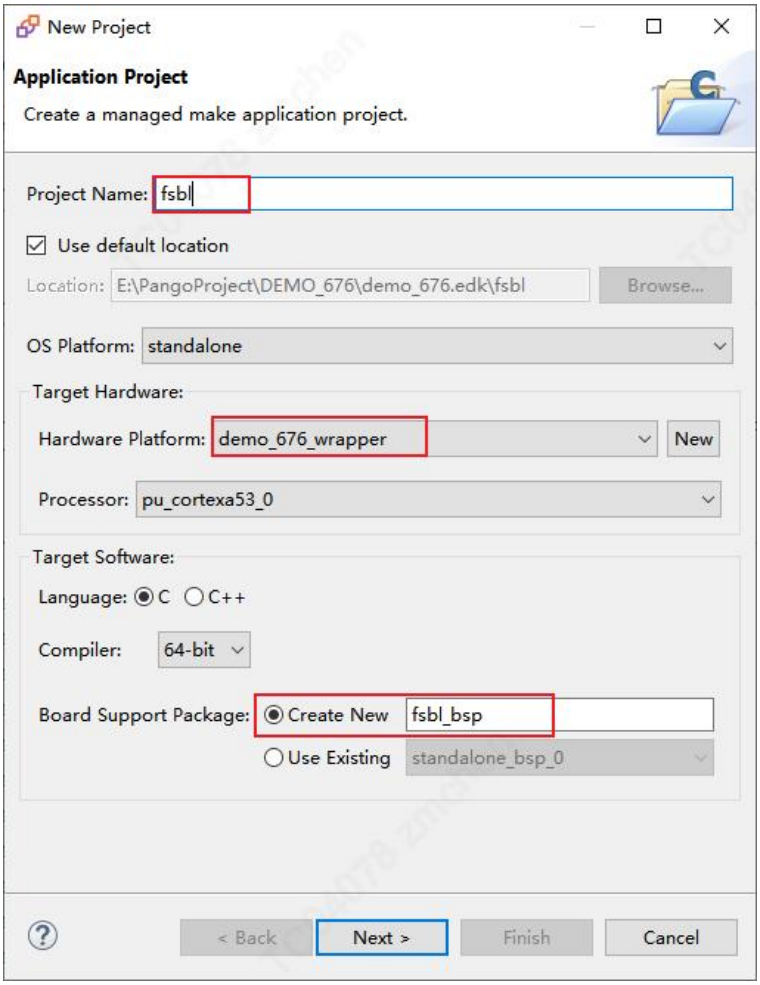
Second stage bootloader 是可选项， 一般在跑操作系统的情况下使用， 比如 Linux 系统的 uboot。

6.1.创建 FSBL 工程

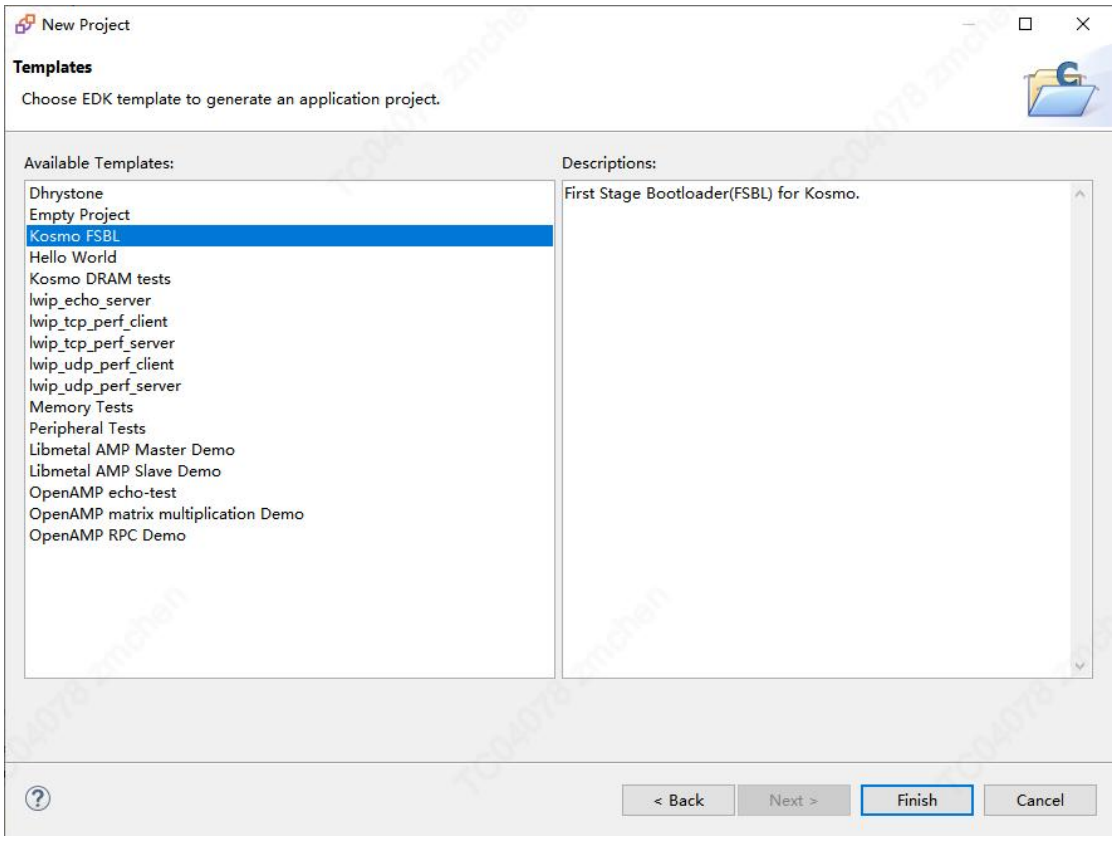
1. 依次点击菜单栏的[File -> New -> Application Project]。



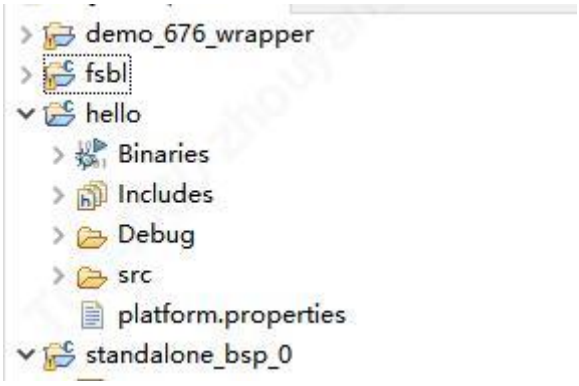
2. 弹出“New Project”界面， 按照下图所示进行配置， 并点击“Next”按钮。



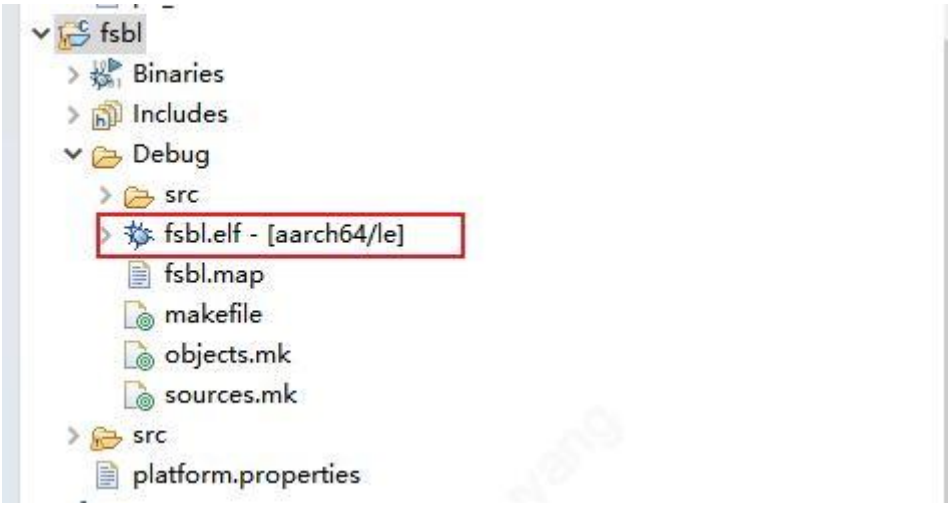
3. 进入模板工程选择界面， 选择“Kosmo FSBL”工程， 并点击“Finish”按钮。



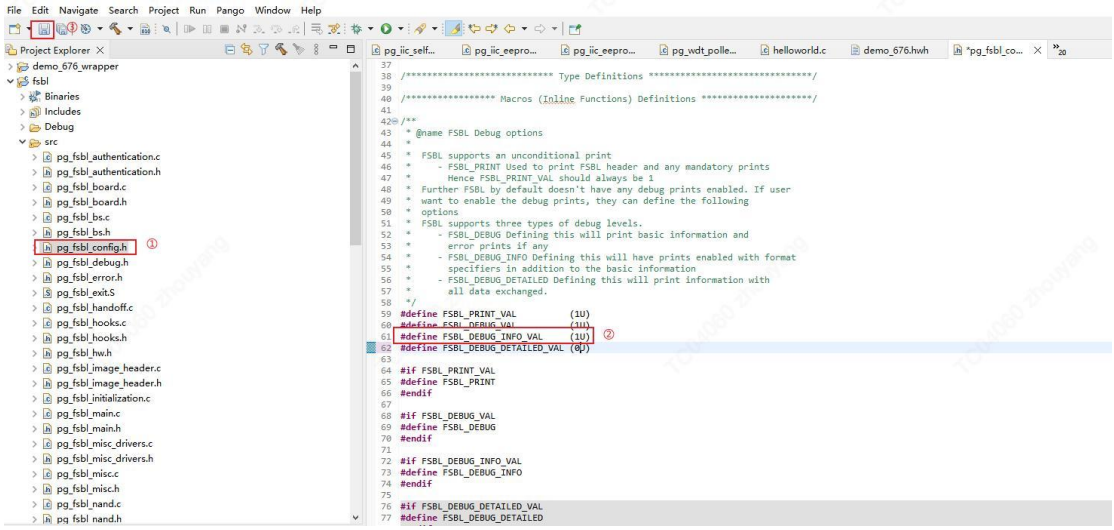
等待窗口自己关闭， 会回到主界面。



5. 创建 fsbl 工程后，第一次会自动编译，编译结束生成如下图所示 fsbl 的可执行.elf 文件。



6. 选择 pg_fsbl_config.h 文件， 修改 FSBL_DEBUG_INFO_VAL 为 1， 保存修改。

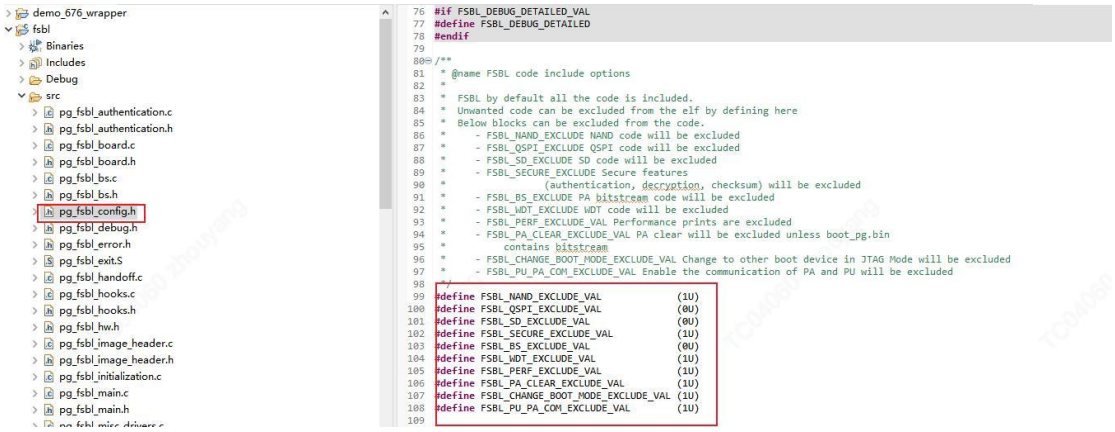


7. fsbl 调试打印信息增多， 有利于调试， 但会导致启动时间变长。

8. 其他配置可参考下表：

宏	功能描述	值
FSBL_PRINT_VAL	Fsbl 基础头信息以及强制信息的打印开关	0U(关闭) 1U(开启)
FSBL_DEBUG_VAL	基本信息以及报错后错误信息的打印开关	0U(关闭) 1U(开启)
FSBL_DEBUG_INFO_VAL	除基本信息外的格式说明信息的打印开关	0U(关闭) 1U(开启)
FSBL_DEBUG_DETAILED_VAL	所有涉及数据交换信息的打印开关	0U(关闭)1U(开启)

9. 确认代码删减的配置选项。



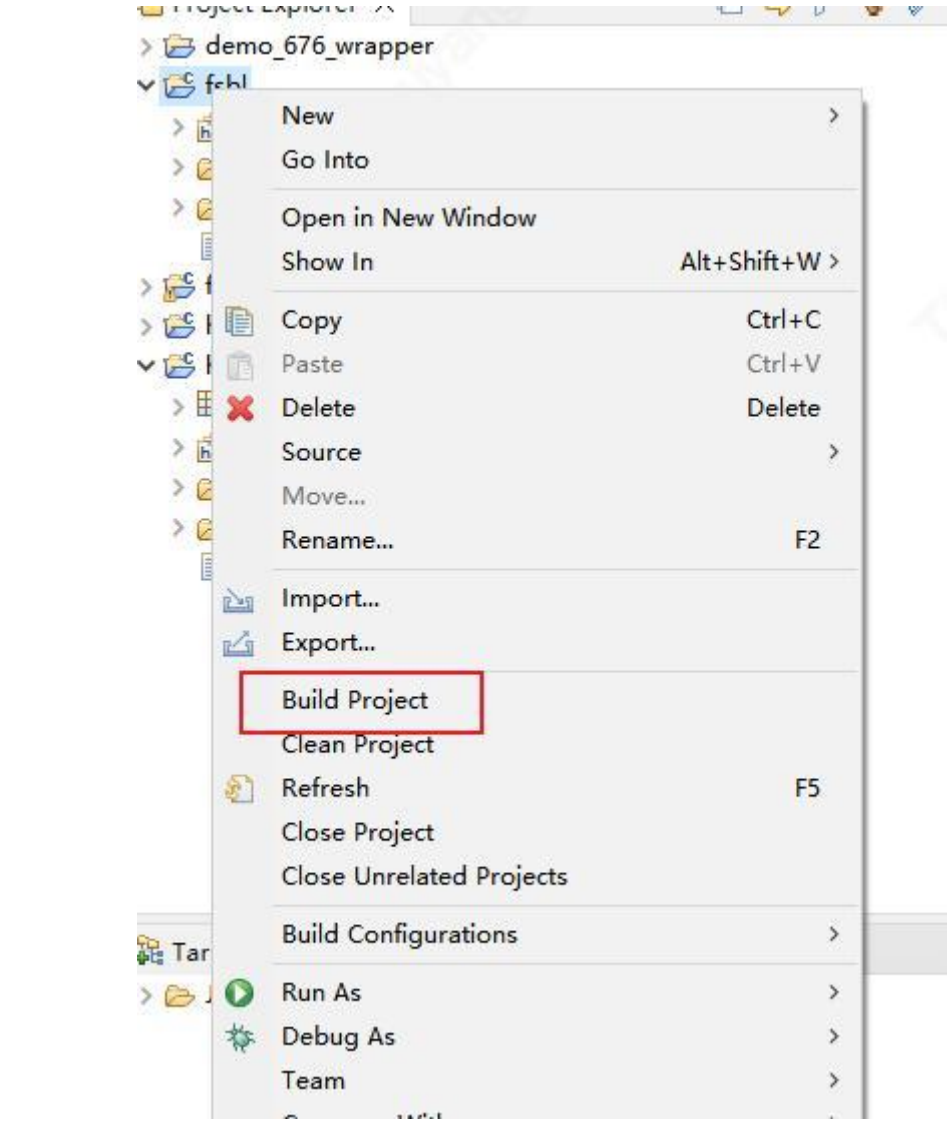
10. 如果配置 FSBL_QSPI_EXCLUDE_VAL 为 0，FSBL_SD_EXCLUDE_VAL 为 1。 步骤 5 里的.elf 文件去除了 SD 卡启动的代码， 保留了 QSPI 启动的代码， size 减小。 且该配置下支持 QSPI 启动不支持 SD 卡启动

11. 其他配置可以参考下表：

宏	功能描述	值
FSBL_NAND_EXCLUDE_VAL	NAND 部分代码的删减控制	0U(保留) 1U(删除)
FSBL_QSPI_EXCLUDE_VAL	QSPI 部分代码的删减控制	0U(保留) 1U(删除)
FSBL_SD_EXCLUDE_VAL	SD 部分代码的删减控制	0U(保留) 1U(删除)
FSBL_SECURE_EXCLUDE_VAL	安全功能(安全认证、 解密、 校验)部分代码的删减控制	0U(保留) 1U(删除)
FSBL_BS_EXCLUDE_VAL	PA bitstream 代码部分的删减控制	0U(保留) 1U(删除)
FSBL_WDT_EXCLUDE_VAL	WDT 部分代码的删减控制	0U(保留) 1U(删除)
FSBL_PERF_EXCLUDE_VAL	性能信息打印部分代码的删减控制	0U(保留) 1U(删除)
FSBL_PA_CLEAR_EXCLUDE_VAL	PA clear 部分代码的删减控制	0U(保留) 1U(删除)
FSBL_CHANGE_BOOT_MODE_EXCLUDE_VAL	Jtag 模式切换到其他启动设备部分代码的删减控制	0U(保留) 1U(删除)
FSBL_PU_PA_COM_EXCLUDE_VAL	PA 和 PU 通信部分代码的删减控制	0U(保留) 1U(删除)
FSBL_PA_RST_EXCLUDE_VAL	当位流加载时， 是否需要复位 PA	0U(保留) 1U(删除)

6.2.构建 Project

1. 在 fsbl 工程上， 右键后， 选择 Build Project 进行编译， 或者选择 fsbl 工程之后直接选择下红框中的锤子按钮也可以进行编译。



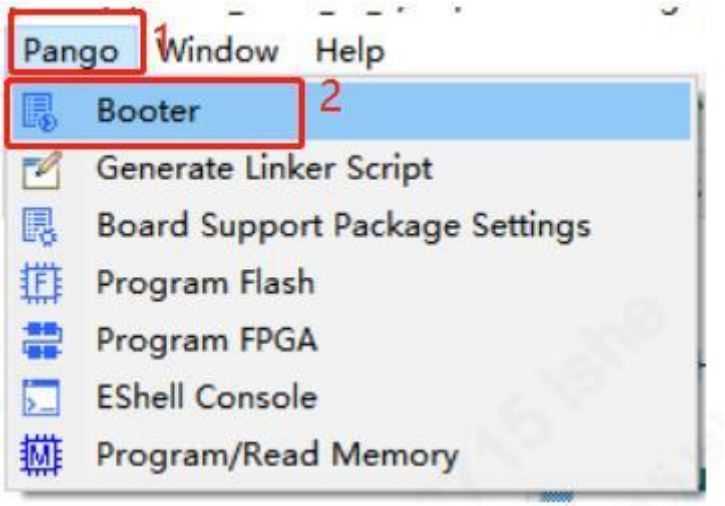
看到如下结果说明编译成功。

```
Building target: fsbl.elf
Invoking: GNU Arm Cross C Linker
aarch64-none-elf-gcc -mstrict-align -O0 -fmessage-length=0 -ffunctio
Finished building target: fsbl.elf

Invoking: GNU Arm Cross Print Size
aarch64-none-elf-size --format=berkeley "fsbl.elf"
   text    data     bss     dec     hex filename
  44436   42640   13680  100756   18994 fsbl.elf
Finished building: fsbl.siz
```

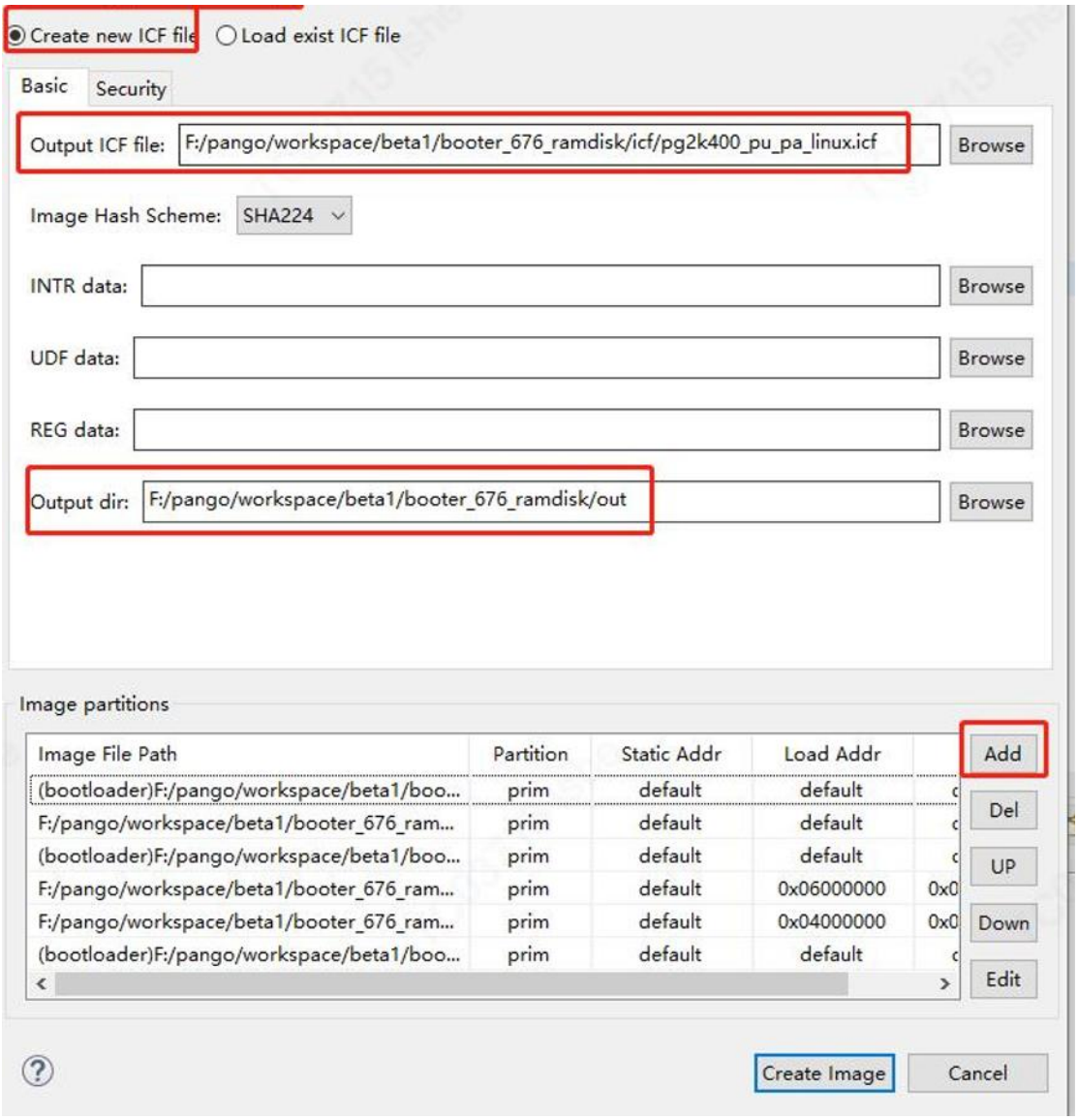
6.3.制作含有 Linux 的 Booter

制作 Linux 的 BOOT_PG.bin 的方式有 Linux 环境制作和 EDK 制作两种方式，本小节主要介绍制作含有 Linux 镜像和设备树文件的 booter。

依次点击菜单栏[Pango -> Booter]。

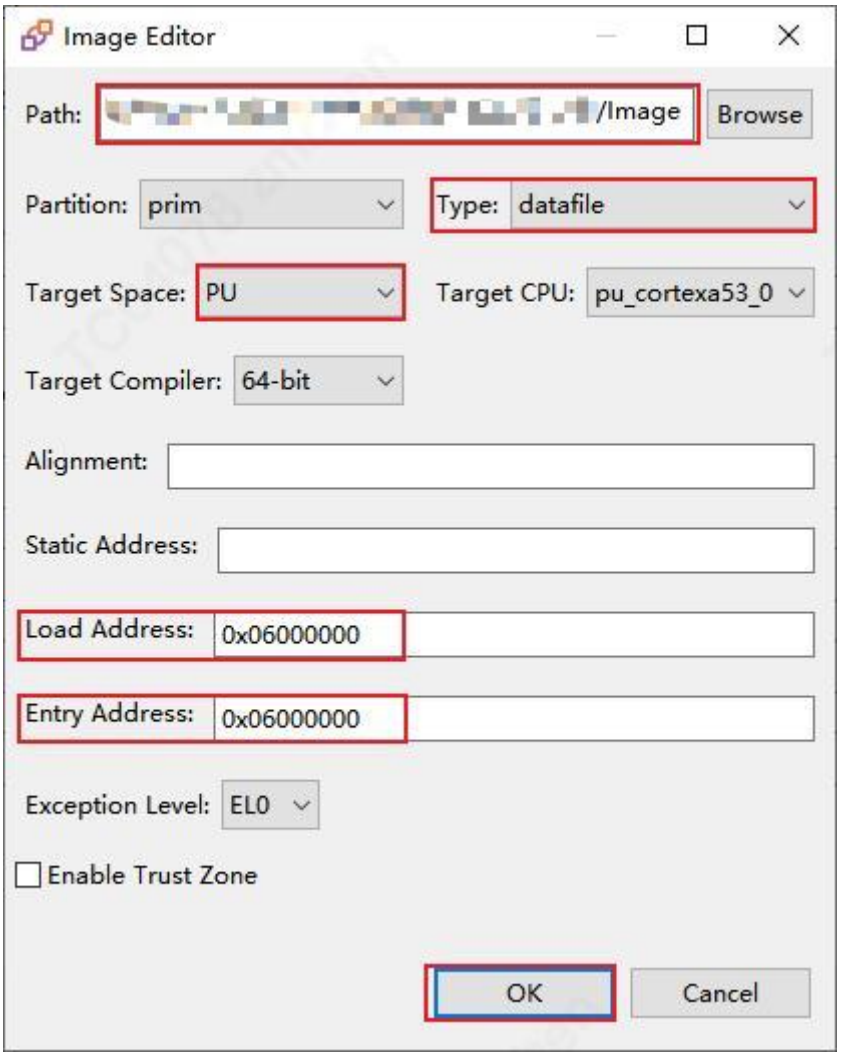


相关配置选择如下图所示， Output ICF File 以及 Output dir 根据自己所创建的目录进行选择。

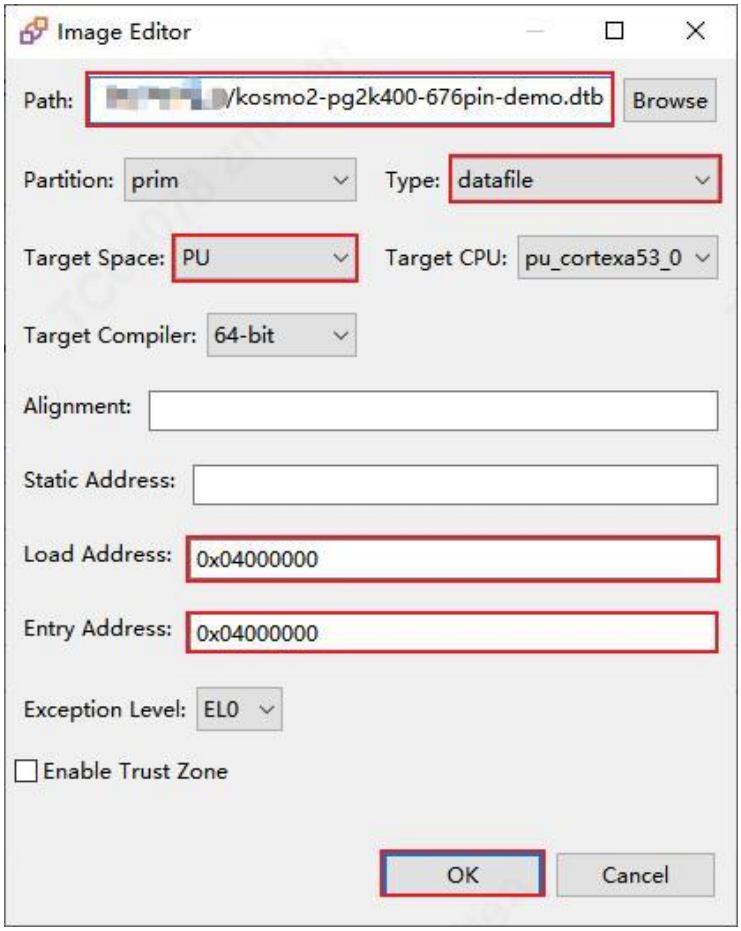


点击 Add 按钮添加 Image partitions，这里主要介绍添加的 Linux 镜像以及设备树。

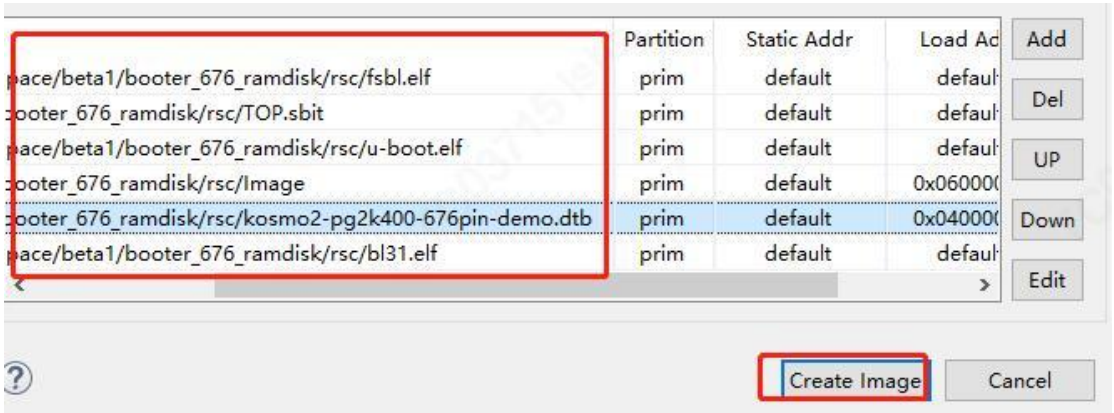
1) Linux 镜像的添加，按照如下图所示的配置，选择 Linux 镜像，Type 选择为 datafile，Target Space 选择为 PU，配置 Load Address 和 Entry Address 配置其地址为 0x06000000， 然后点击“OK”。



2) 设备树的添加， 按照如下图所示的配置，选择设备树，Type 选择为 datafile，TargetSpace 选择为 PU，配置 Load Address 和 Entry Address 配置其地址为 0x04000000，然后点击“OK”。



通过“UP”和“Down”按钮调整 Image File 的顺序，确保 fsbl 在首位，bl31.elf 在末尾，其中 Top.sbit 为可选加入的比特流文件。

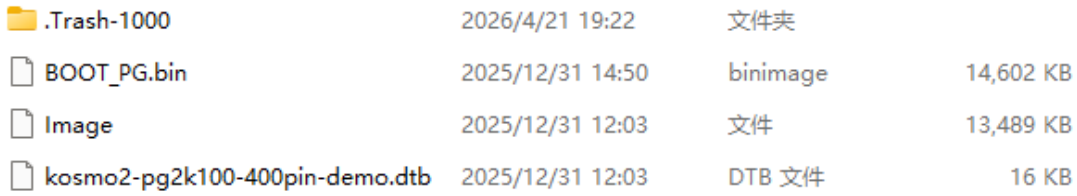


点击“Create Image”按钮，等待生成 image 结束，在设置的 Output dir 目录下会生成对应 BOOT_PG.bin 文件。

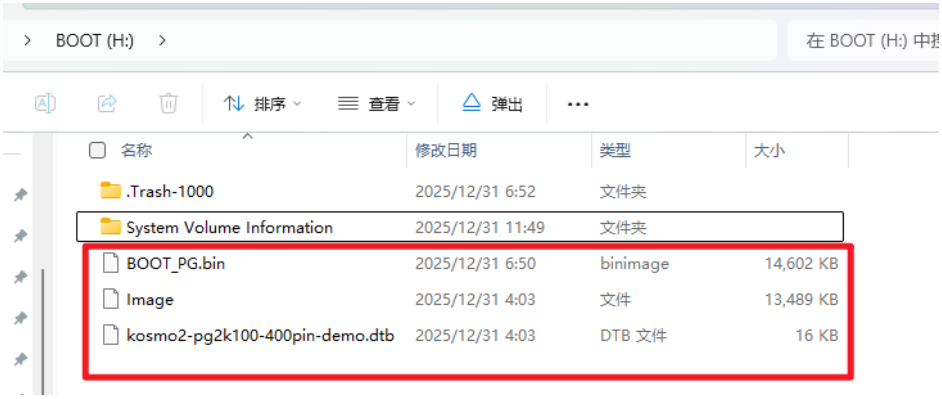
注意：image partitions 的添加顺序（第一个必须为 fsbl，最后一个必须为 bl31.elf，中间其它文件顺序可以改变），其中顺序为 fsbl.elf、TOP.sbit、（hello.elf）uboot.elf、bl31.elf，其中 stage2 和应用程序一般情况不会被同时添加。若需要启动 linux 则添加 uboot.elf、bl31.elf，若需加载 helloworld 则添加 hello.elf。TOP.sbit 为位流文件其类型为 datafile，如下图所示。若需改变 image file 的先后顺序可使用右侧的 UP 按钮和 Down 按钮来改变。

6.4.SD 卡启动引导 Linux

1. 制作的 BOOT_PG.bin 包含的可执行文件为 fsbl.elf、u-boot.elf 以及 bl31.elf。也可以直接使用提供的 Linux 相关的 BOOT_PG.bin。



3. 将 BOOT_PG.bin、Image 和 dtb 拷贝到 SD 卡 BOOT 分区（FAT32 格式）根目录下，文件在 SD 卡启动的压缩包中。



3.将 SD 卡插入开发板的 SD 卡槽。

4.跳冒进行选择设置。

5.开发板上电后，可以在 EDK Terminal 串口看到如下日志。

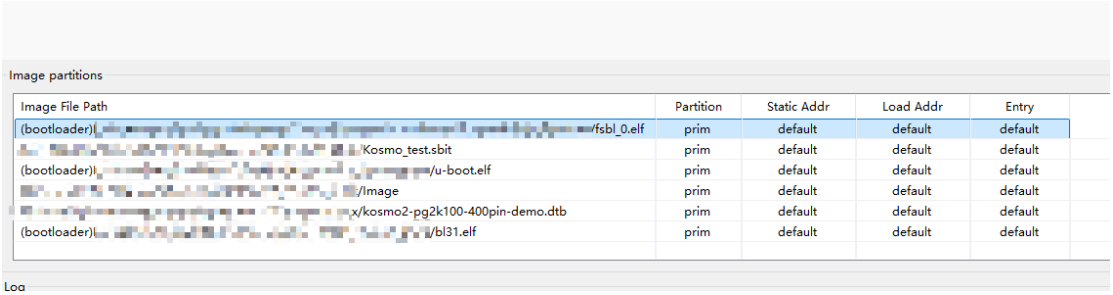
```
FSBL:Jump to 0x00100000 in mode 64



NOTICE:  BL31: v2.4(release):V001R001C001SPC003-2024-12-23-38-g65d85f7-dirty

NOTICE:  BL31: Built : 15:00:39, Jan 10 2025

INFO:    ARM GICv2 driver initialized

INFO:    BL31: Initializing runtime services

INFO:    BL31: PM Service Init Complete: API v1.1

INFO:    BL31: Preparing for EL3 exit to normal world

INFO:    Entry point address = 0x2a0000

INFO:    SPSR = 0x3c9


Debug uart enabled
Use the pango cfg serial console configuration

U-Boot 2021.01 (Jan 10 2025 - 15:00:40 +0800)

Model: Kosmo2 PG2K400 RevA
Board: PG2K400
DRAM:  1 GiB
ICACHE: Enable ICACHE
DCACHE: Enable DCACHE and MMU
EL Level: EL2
NAND:  0 MiB
MMC:   mmc@f0022000: 0
***fun:env_load;Loading Environment from FAT...
*** Warning - bad CRC, using default environment

Load OK
In:    serial
Out:   serial
Err:   serial
Bootmode: SD_MODE

OK
Starting telnetd: OK
Starting NFS statd: OK
Starting NFS services: OK
Starting linuxptp daemon: OK
Starting linuxptp system clock synchronization: OK
Starting vsftpd: OK


Welcome to pango kosmo rootfs, Release Version:1.0.0

buildroot login:
```

6.5.PU 加载 PA 的 SD 卡启动测试

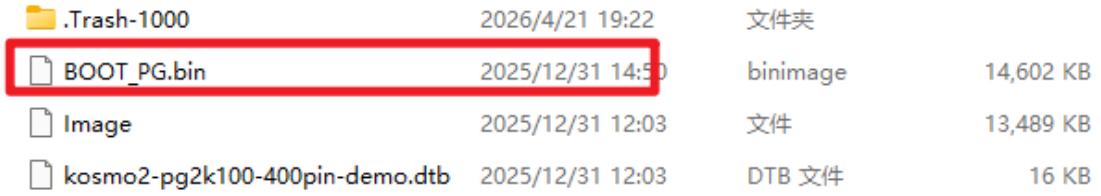
通过在 Booter 中制作包含的可执行文件 fsbl.elf、Top.sbit、hello.elf 的 BOOT_PG.bin 可以在 SD 启动的同时加载 sbit 流以及 hello.elf。Booter 中第一个必须为 fsbl，最后一个必须为 bl31.elf，中间其它文件顺序可以改变，其中顺序为 fsbl.elf、TOP.sbit、（hello.elf）uboot.elf、bl31.elf，其中 uboot.elf 和应用程序（hello.elf）不会被同时添加。若需要启动 linux 则添加 uboot.elf、bl31.elf，若需加载 helloworld 则添加 hello.elf。

6.5.1.SD 卡启动带 PA 的 linux

下图为带 PA 的 linux 启动 booter 制作 BOOT_PG.bin 文件内容，其中.sbit 文件为 PA 比特流。



将制作过程中包含 Top.sbit 的 BOOT_PG.bin 文件拷贝到 SD 卡 0 分区根目录下，替换掉 BOOT_PG.bin，结果如下。

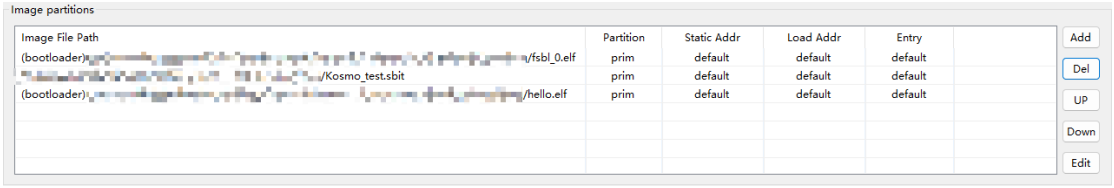


3. 将 SD 卡插入开发板的 SD 卡槽

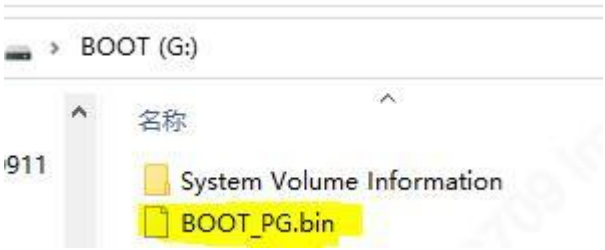
4.开发板上电后， 可以在开发板上看到 PA 数据流加载通过 EDK Terminal 串口可以看到 Linux 启动。

6.5.2.SD 卡启动的带 PA 的 PU 程序

下图为带 PA 和应用程序的 BOOT_PG.bin 文件制作内容，其中.sbit 文件为 PA 比特流。



将制作过程中包含 Top.sbit 的 BOOT_PG.bin 文件拷贝到 SD 卡 0 分区根目录下，替换掉 BOOT_PG.bin，结果如下。



3. 将 SD 卡插入开发板的 SD 卡槽

4.开发板上电后， 可以在 EDK Terminal 串口看到如下日志。

```
FSBL Release 2024.4 (Jan 13 2025 - 20:08:34)
[IPSS-M]Configure CE HW...
FSBL:Jump to 0x00100000 in mode 64
Hello World
```