OpenFlash用户手册-v1.0

1. 概述

OpenFlash是一款基于FPGA的开放SSD敏捷开发与验证平台，给用户提供开放的固件和硬件逻辑编程接口，用户可以基于OpenFlash完成以下工作

（1）定制化开发SSD固件：实现自定义的地址映射、垃圾回收、缓存等策略；实现NVMe规范新特性，例如zoned namespace，flexilble data placement等

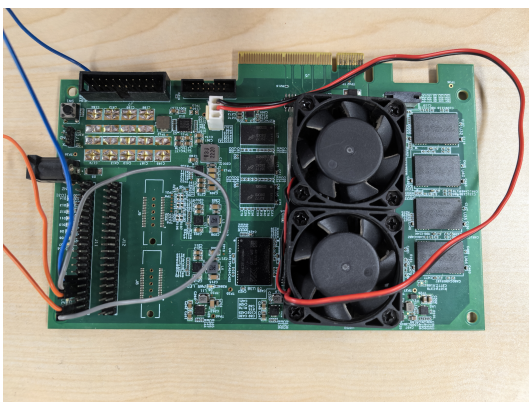
（2）开发近数据加速器

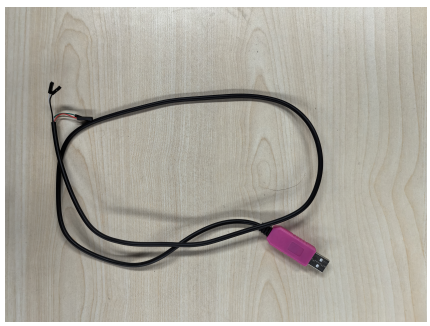
本手册主要介绍OpenFlash套件、部署使用流程和FTL固件框架。有任意疑问欢迎联系：jiangdejun@ict.ac.cn。

2. OpenFlash套件及规格

2.1 套件说明

表1 OpenFlash套件说明

序号	名称	说明	图片
1.	OpenFlash板卡	自制AIC板卡，包含FPGA芯片和NAND颗粒芯片	
2.	ARM烧写器	用于烧写OpenFlash的FTL固件，支持断点调试	

3.	Xilinx烧写器	用于烧写OpenFlash主控比特流	
4.	串口线	用于打印调试	

2.2 硬件规格

- SSD控制器
  - FPGA：Xilinx XCKU15P-2ffva1156e
  - ARM core (125MHz)
  - RISC-V core (62.5MHz)

    - Microcode for each ONFI channel
- 内存
    - SRAM：2MB
    - DDR4 DRAM：2GB
    - 1600MT/s
- NAND颗粒
    - ONFI 4 interface
    - 8 channel, 4 CE
    - 2TB (512GB per NAND chip)
- 主机接口
    - PCIe Gen3x4 End-Point (AIC)
    - Support for a subset of NVMe command set

2.3 性能参数

表2 OpenFlash性能参数（ARM核）

IO模式	带宽(MB/s)	IOPS(K)	延迟（QD=1 jobs=1）us
64KB顺序写（QD=32 jobs=4）	1818	27.7	43
64KB顺序读（QD=32 jobs=4）	2197	33.5	258
4KB随机写（QD=32 jobs=4）	1128	275	7
4KB随机读（QD=32 jobs=4）	1505	367	109

3. OpenFlash部署与启动

3.1 软硬件环境

（1）测试主板1块

- 用于安装OpenFlash板卡
- 参考配置：华硕B660M-PLUS D4的主板，CPU 13th Gen Intel(R)core(TM)i5-13400，内存DDR4 32G；操作系统 Ubuntu 22.04

（2）开发主机1台

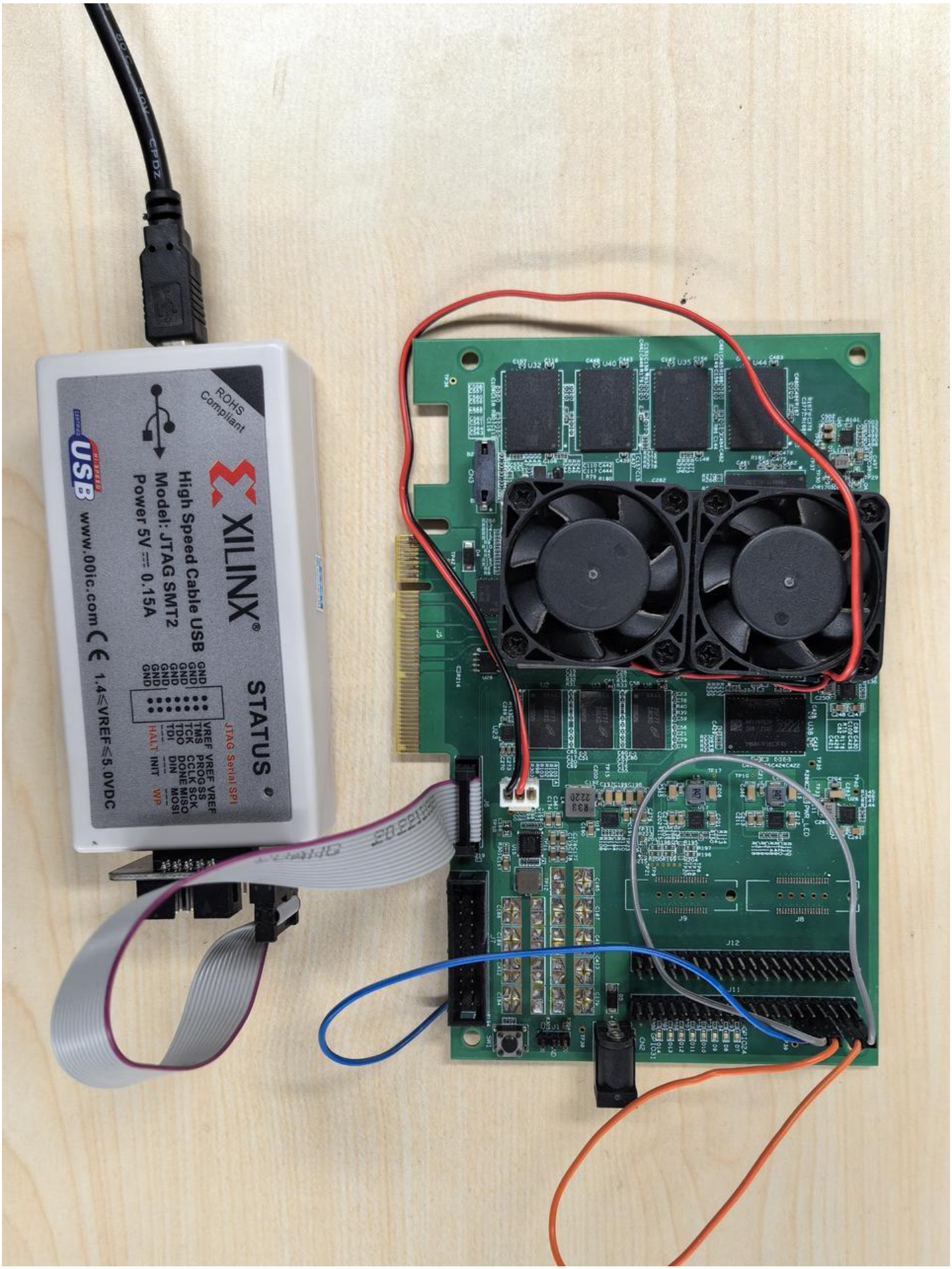
- 用于安装OpenFlash二次开发相关软件
- 参考配置：Windows主机

表3 开发主机配套软件

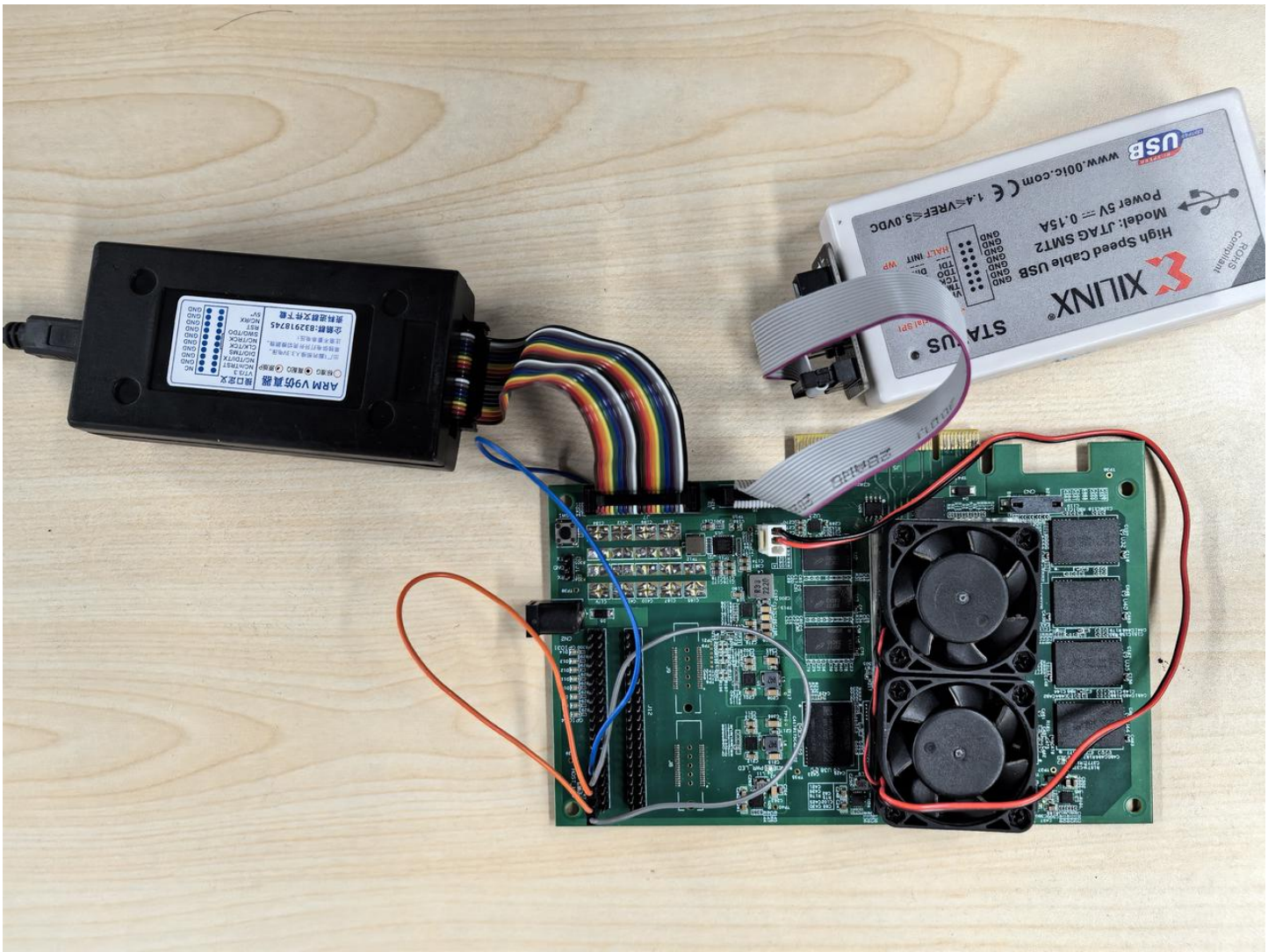
序号	名称	用途
1.	Vivado 2019.1	用于烧写比特流
2.	ARM DS-5 5.20.0	用于烧写固件和断掉调试
3.	MSYS	make工具链
4.	DS5-Jlink	调试驱动
5.	MobaXterm	用于连接OpenFlash串口和测试主板

3.2 板卡连线示意图

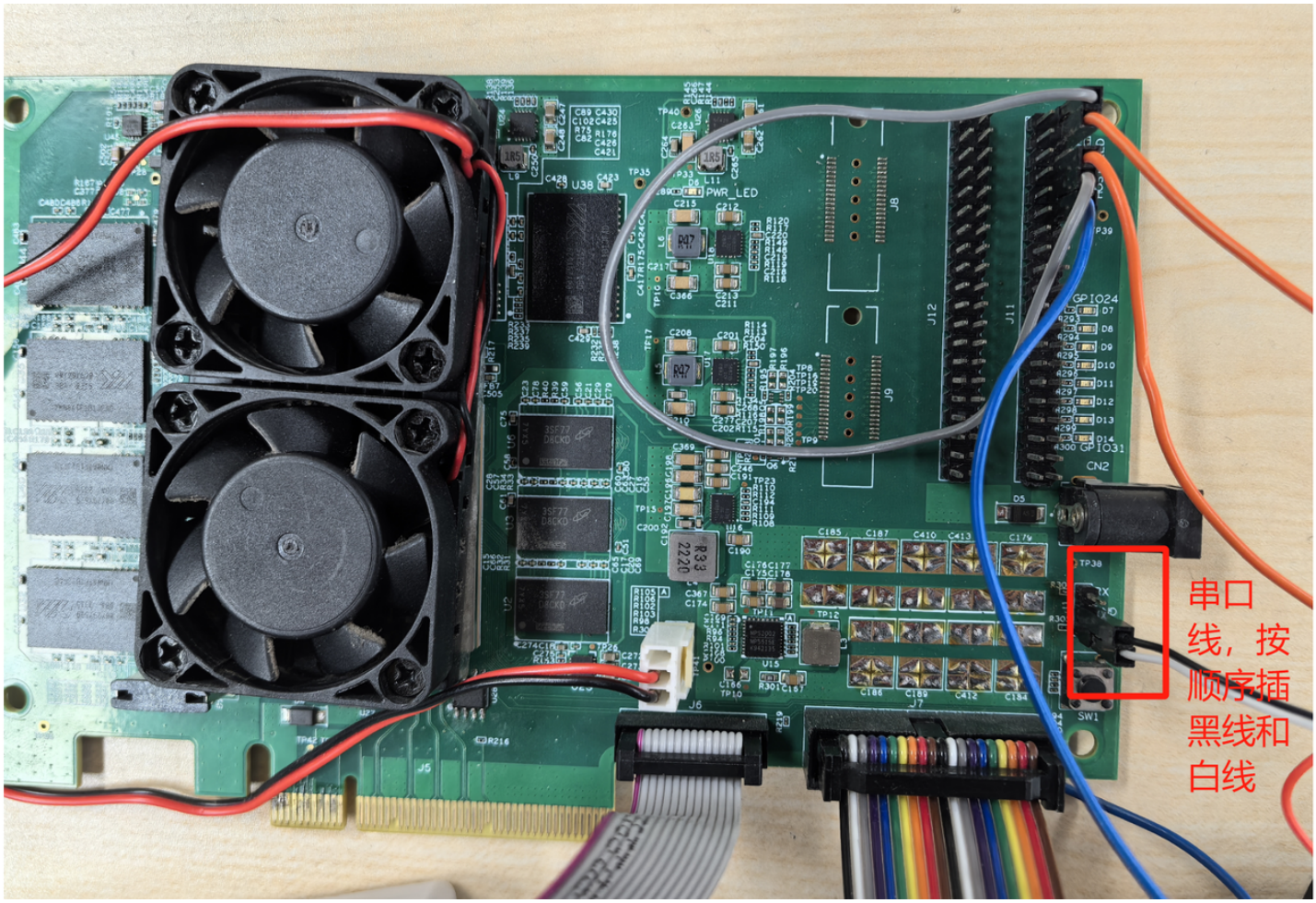
1. 在OpenFlash板卡上插上Xilinx烧写线，USB端插在开发主机上（需安装vivado）。



2. 在OpenFlash板卡上插上ARM烧写线，USB端插在开发主机上（需安装DS-5）。



3. 在OpenFlash板卡上插串口线，USB端插在开发主机上（需安装vivado)。



4. 把OpenFlash板卡插在测试主板上。

3.3 ARM核版本启动

3.3.1 代码结构介绍

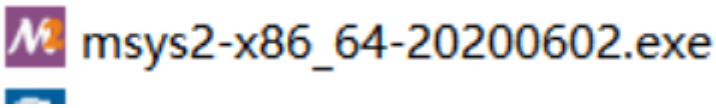
表4 OpenFlash代码目录

序号	文件夹名	说明
1.	bin	包含固件烧写文件SYS_S.axf
2.	drv	包含对nand操作的代码
3.	inc	包含头文件
4.	lib	编译工程中产生的依赖文件
5.	make	make文件夹
6.	rvct	包含上电启动的代码

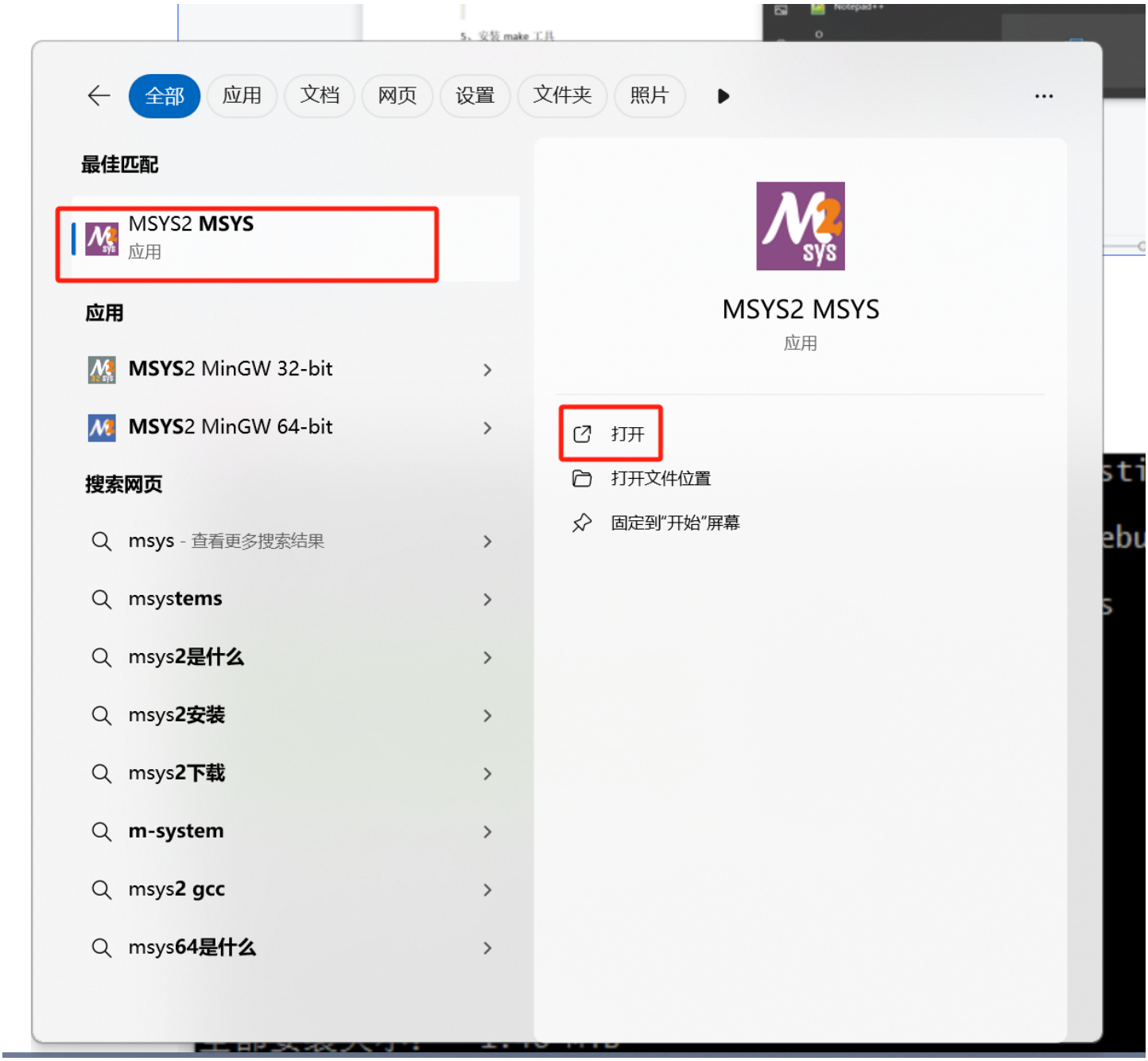
7.	sys	包含ftl代码

3.3.2 ARM核开发环境搭建

1. 自行安装Eclipse for DS-5，版本号是 5.20.0
2. 自行安装vivado 2019.1版本
3. 安装make工具



4. 打开msys


全部
应用
文档
网页
设置
文件夹
照片
最佳匹配
MSYS2 MSYS
应用
应用
MSYS2 MinGW 32-bit
MSYS2 MinGW 64-bit
搜索网页
msys - 查看更多搜索结果
msystems
msys2是什么
msys2安装
msys2下载
m-system
msys2 gcc
msys64是什么
MSYS2 MSYS
应用
打开
打开文件位置
固定到"开始"屏幕

5. msys界面

```
    Make sure you didn't emit any warnings while testing
msys/remake-git 4.1.2957.e3e34dd9-1
    Enhanced GNU Make - tracing, error reporting, debugging, pr
msys/sharutils 4.15.2-1
    Makes so-called shell archives out of many files

yangfangming162@DESKTOP-5MTIRTF MSYS ~
$

yangfangming162@DESKTOP-5MTIRTF MSYS ~
$ pacman -S msys/make
警告：make-4.3-1 已经为最新 -- 重新安装
正在解析依赖关系...
正在查找软件包冲突...

软件包 (1) make-4.3-1

全部安装大小：   1.48 MiB
净更新大小：   0.00 MiB

:: 进行安装吗？ [Y/n] n

yangfangming162@DESKTOP-5MTIRTF MSYS ~
$
```

6. 安装make

3.安装make编译器

```
#查询并找到msys/make
pacman -Ss make
#安装
pacman -S msys/make
```

注意，如果安装make时出现如下错误：

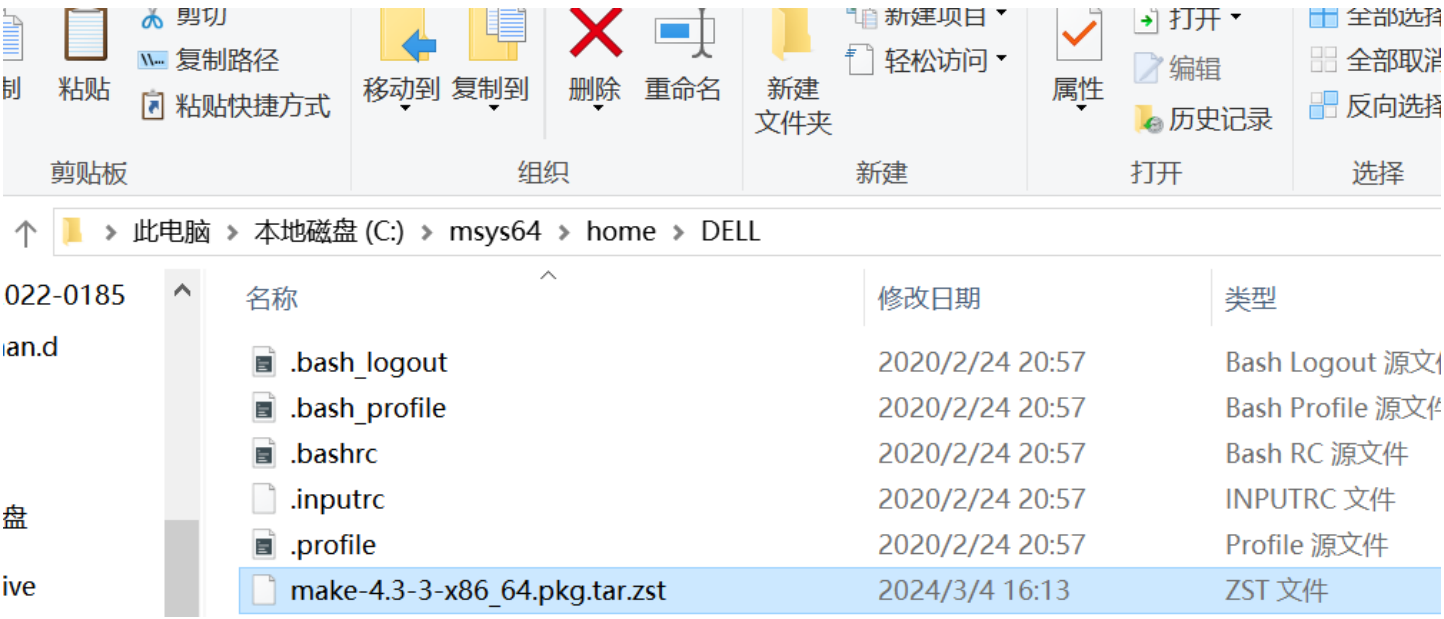
```
错误: 无法从 mirrors.ustc.edu.cn : The requested URL returned error: 404 获取文件 'mingw64.db'
错误: 无法从 mirror.bit.edu.cn : The requested URL returned error: 404 获取文件 'mingw64.db'
 mingw64 已经是最新版本
错误: 无法从 mirrors.ustc.edu.cn : The requested URL returned error: 404 获取文件 'msys.db'
```

先执行命令： pacman -Syu

```
# pacman -S msys/make
错误: mingw32: 密钥 "5F944B027F7FE2091985AA2EFA11531AA0AA7F57" 未知
:: Import PGP key 5F944B027F7FE2091985AA2EFA11531AA0AA7F57? [Y/n] n
错误: mingw64: 密钥 "5F944B027F7FE2091985AA2EFA11531AA0AA7F57" 未知
:: Import PGP key 5F944B027F7FE2091985AA2EFA11531AA0AA7F57? [Y/n] n
错误: msys: 密钥 "5F944B027F7FE2091985AA2EFA11531AA0AA7F57" 未知
:: Import PGP key 5F944B027F7FE2091985AA2EFA11531AA0AA7F57? [Y/n] n
错误: 数据库 'mingw32' 无效 (无效或已损坏的数据库 (PGP 签名))
错误: 数据库 'mingw64' 无效 (无效或已损坏的数据库 (PGP 签名))
错误: 数据库 'msys' 无效 (无效或已损坏的数据库 (PGP 签名))
```

下载 make-4.3-3-x86_64.pkg.tar.zst

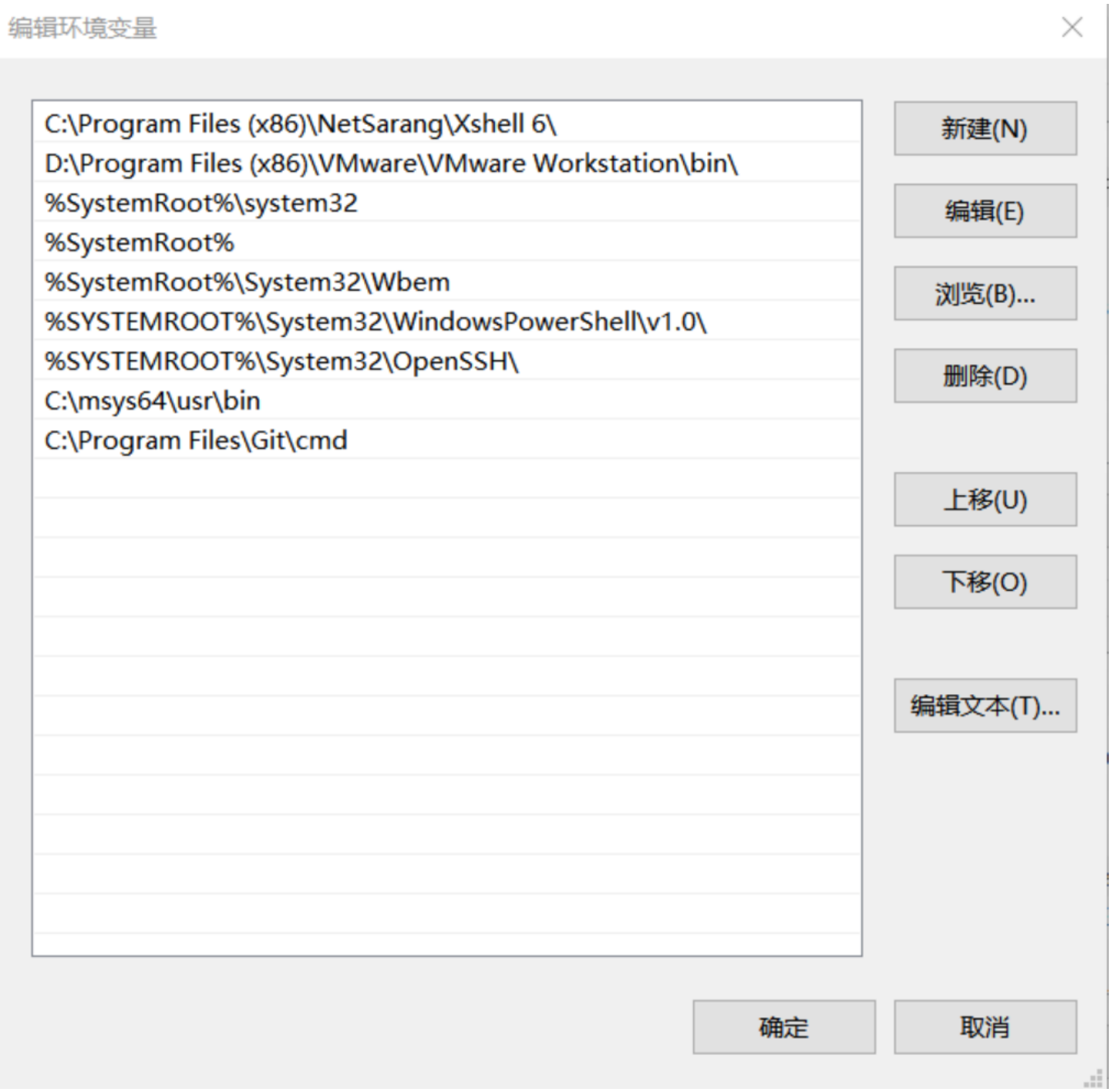
将make-4.3-3-x86_64.pkg.tar.zst放置到如下位置：



使用如下命令安装make

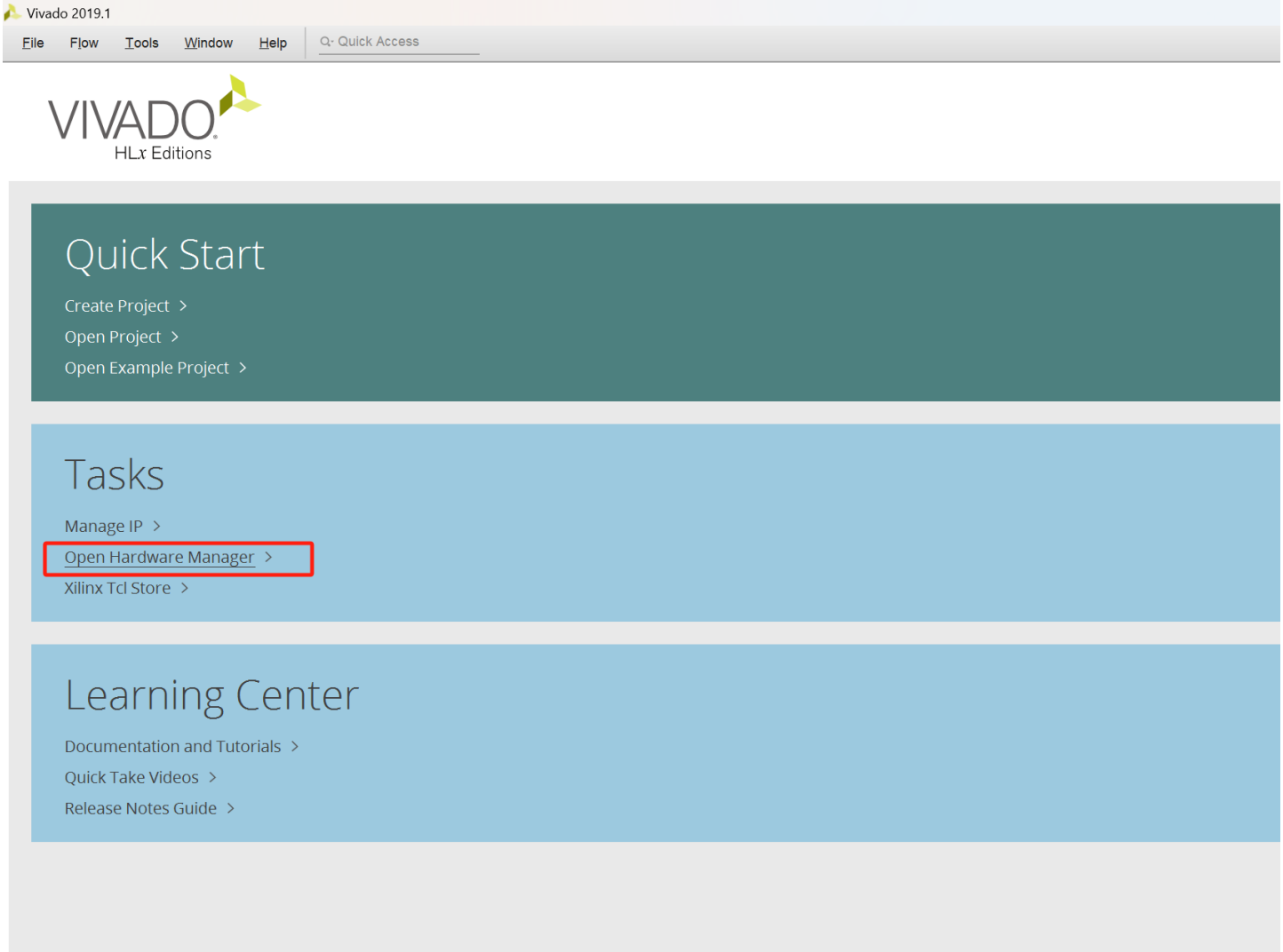
```
DELL@DESKTOP-J6NH68M MSYS ~
$ pacman -U make-4.3-3-x86_64.pkg.tar.zst
正在加载软件包...
正在解析依赖关系...
正在查找软件包冲突...

软件包 (1) make-4.3-3

全部安装大小:   1.48 MiB

:: 进行安装吗? [Y/n] y
(1/1) 正在检查密钥环里的密钥                          [#####################]
(1/1) 正在检查软件包完整性                            [#####################]
(1/1) 正在加载软件包文件                              [#####################]
(1/1) 正在检查文件冲突                                [#####################]
(1/1) 正在检查可用存储空间                            [#####################]
:: 正在处理软件包的变化...
(1/1) 正在安装 make                                   [#####################]
```

7. **将make的地址添加到环境变量（path里面），重启Eclipse make才会生效**

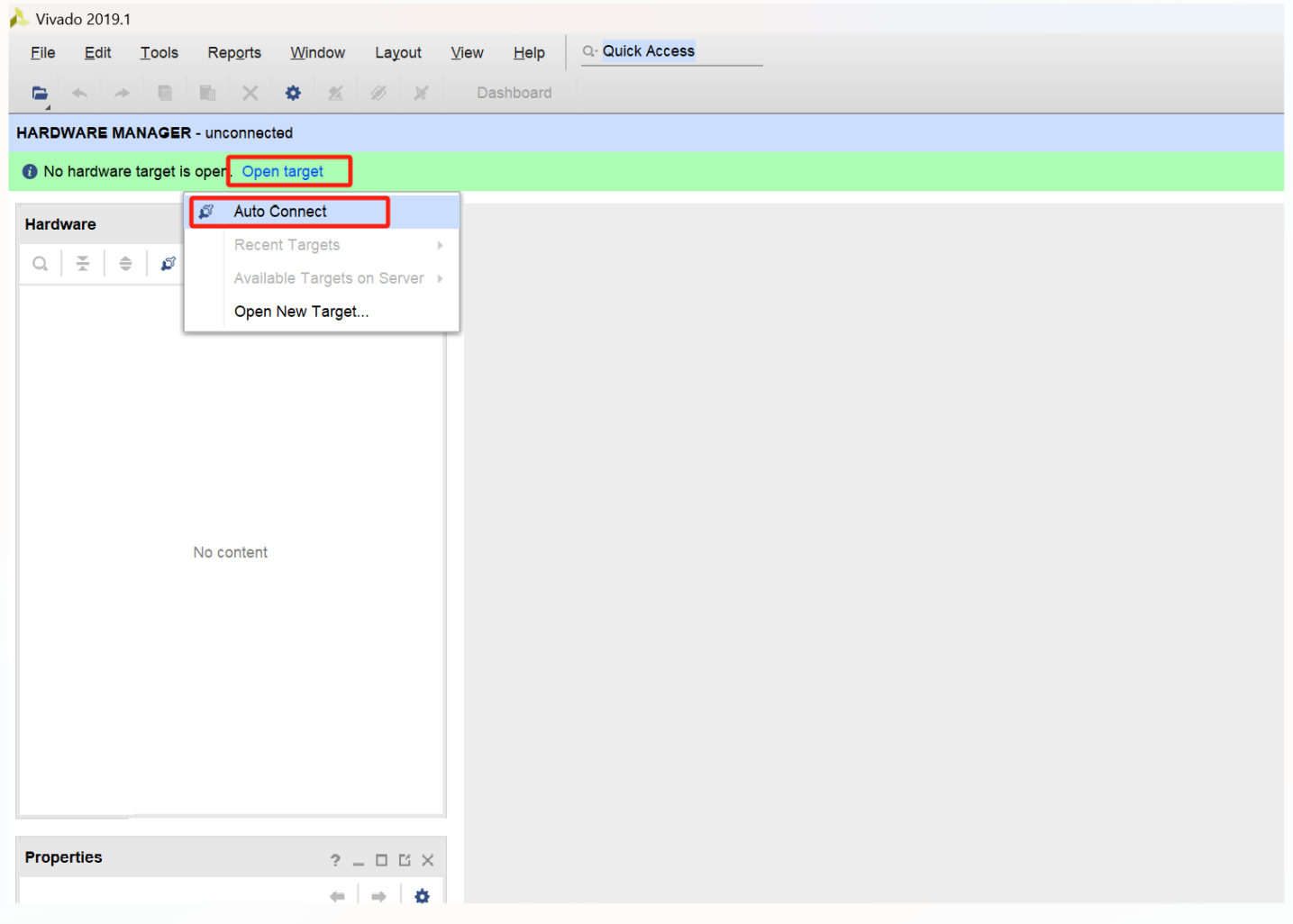


3.3.3 下载比特流

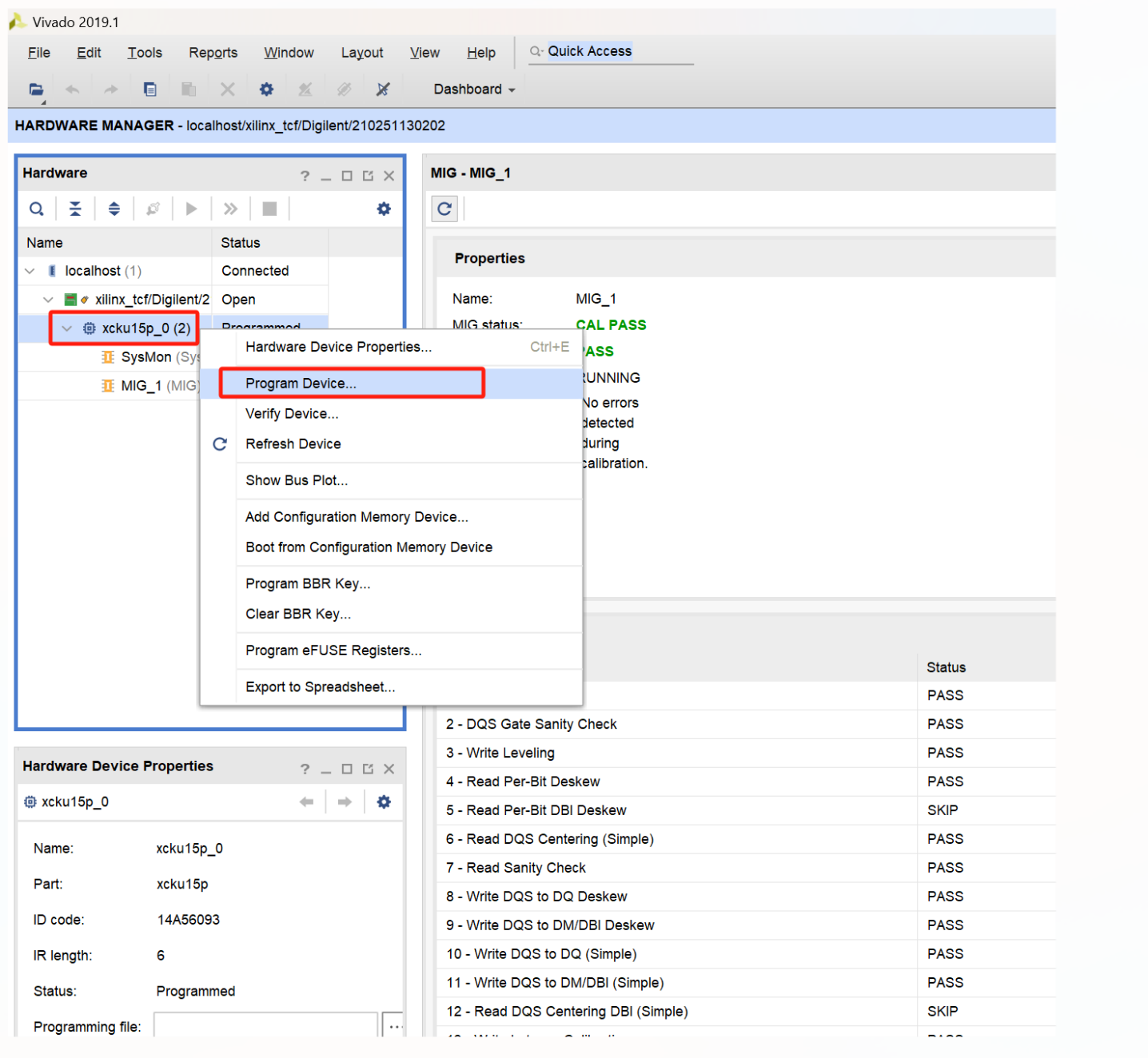
1. 按照板卡连线示意图插好下载线、调试线、串口线之后把OpenFlash板卡插在主板的PCIe口上，然后给主板上电，这样OpenFlash板卡可以通过PCIe口供电。
2. 打开vivado,点击Open Hardware Manager



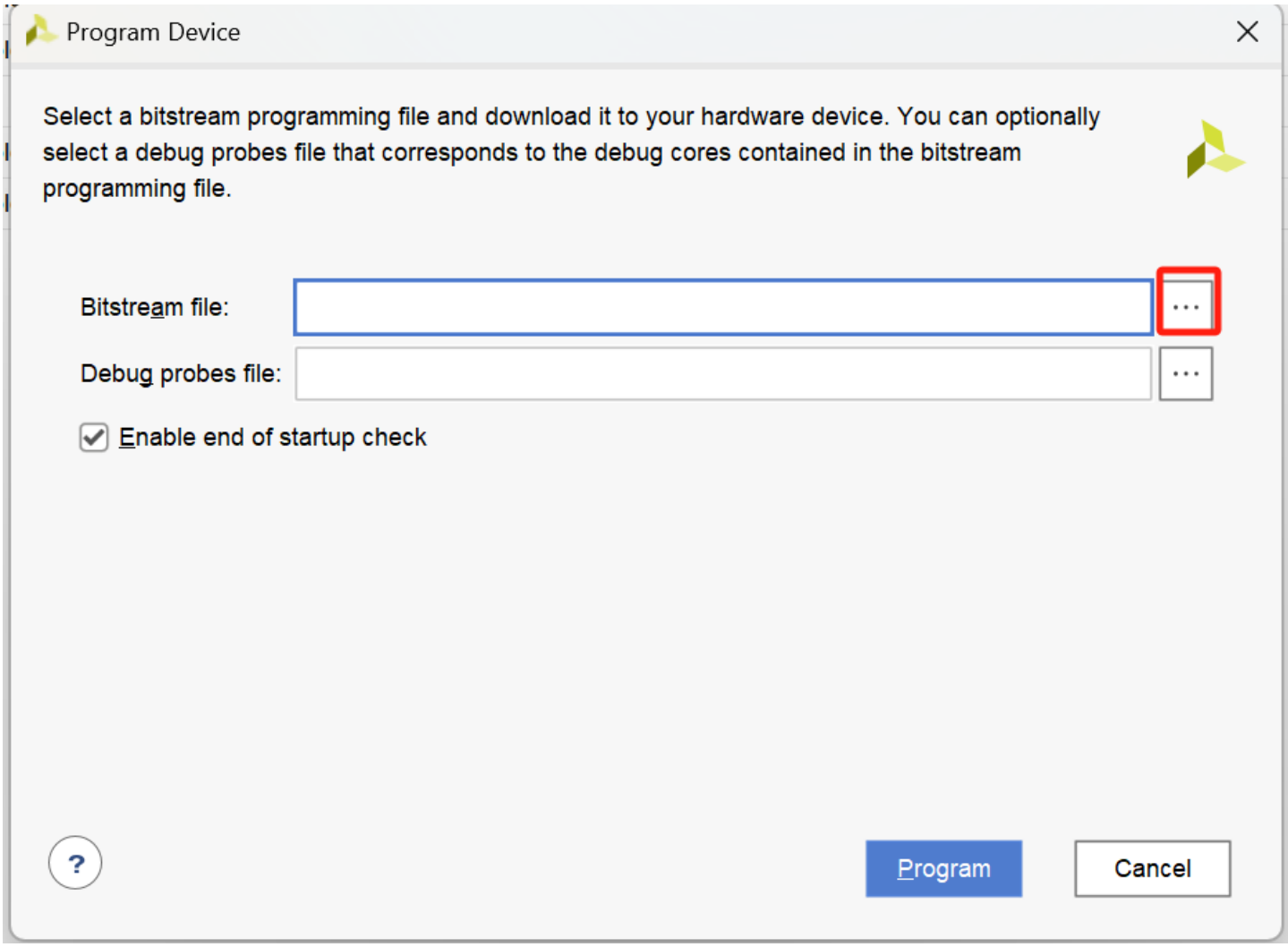
3. 点击Open target,然后 点击Auto Connect



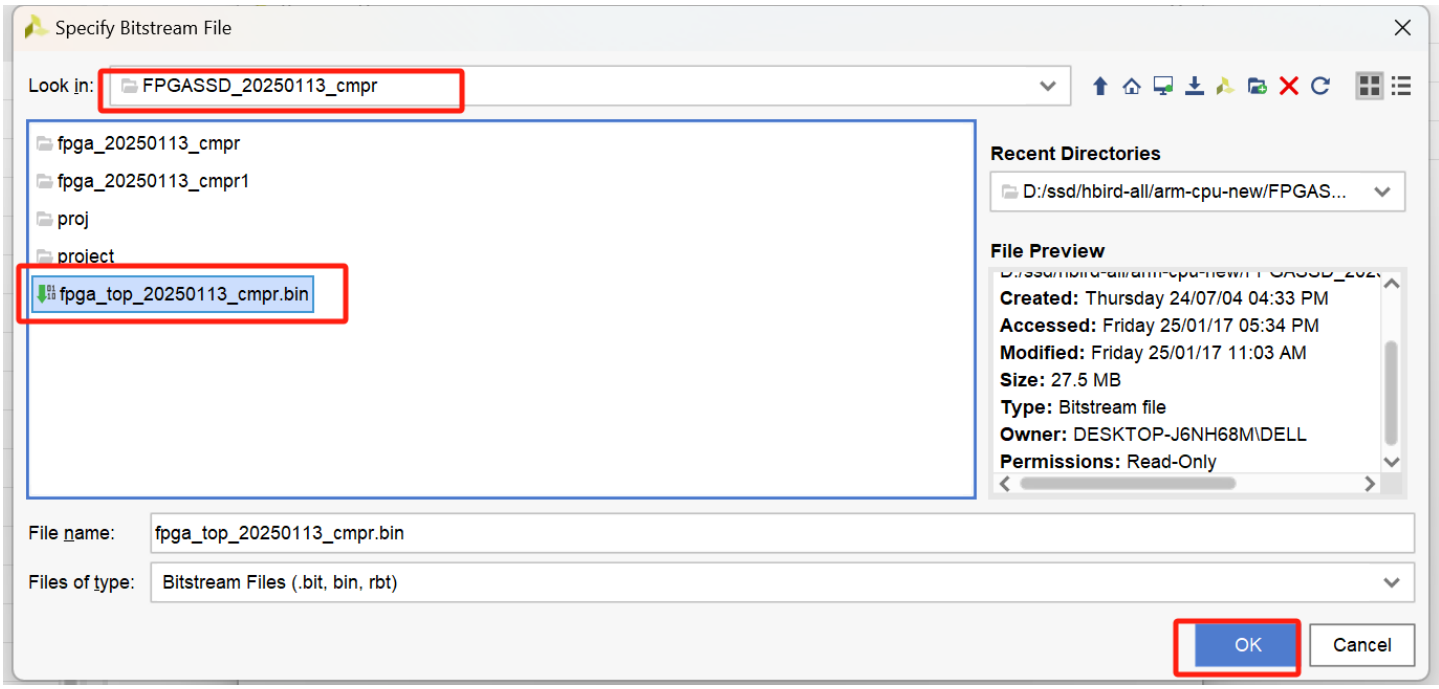
4. 鼠标右击xcku15p_0，点击Program Device


Vivado 2019.1
File
Edit
Tools
Reports
Window
Layout
View
Help
Quick Access
Dashboard
HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210251130202
Hardware
Name
Status
localhost (1)
Connected
xilinx_tcf/Digilent/2
Open
xcku15p_0 (2)
SysMon
MIG_1
Hardware Device Properties...
Ctrl+E
Program Device...
Verify Device...
Refresh Device
Show Bus Plot...
Add Configuration Memory Device...
Boot from Configuration Memory Device
Program BBR Key...
Clear BBR Key...
Program eFUSE Registers...
Export to Spreadsheet...
MIG - MIG_1
Properties
Name:
MIG_1
Hardware Device Properties
xcku15p_0
Name:
xcku15p_0
Part:
xcku15p
ID code:
14A56093
IR length:
6
Status:
Programmed
Programming file:
Status
PASS
2 - DQS Gate Sanity Check
PASS
3 - Write Leveling
PASS
4 - Read Per-Bit Deskew
PASS
5 - Read Per-Bit DBI Deskew
SKIP
6 - Read DQS Centering (Simple)
PASS
7 - Read Sanity Check
PASS
8 - Write DQS to DQ Deskew
PASS
9 - Write DQS to DM/DBI Deskew
PASS
10 - Write DQS to DQ (Simple)
PASS
11 - Write DQS to DM/DBI (Simple)
PASS
12 - Read DQS Centering DBI (Simple)
SKIP

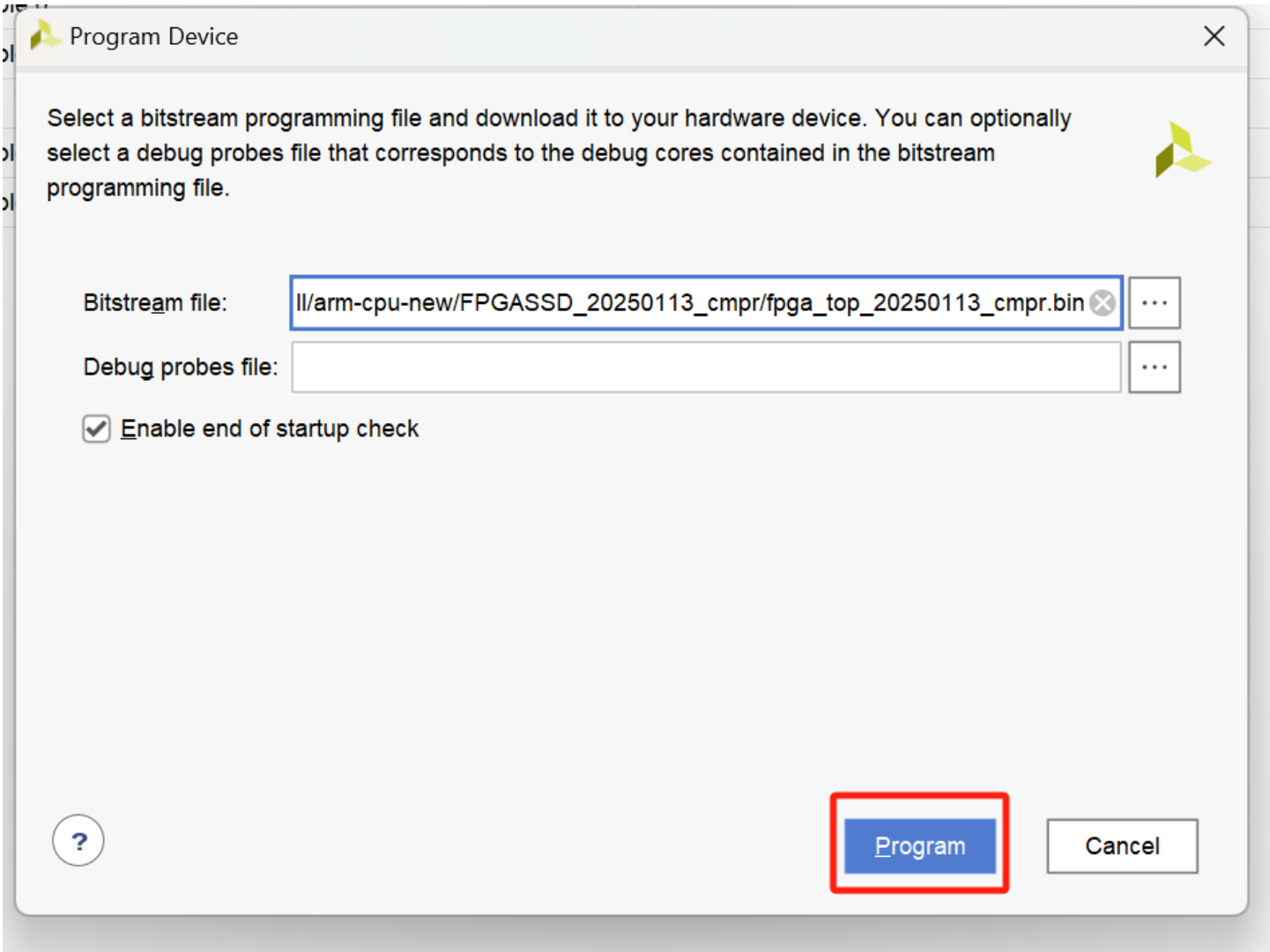
5. 点击...，弹出对话框



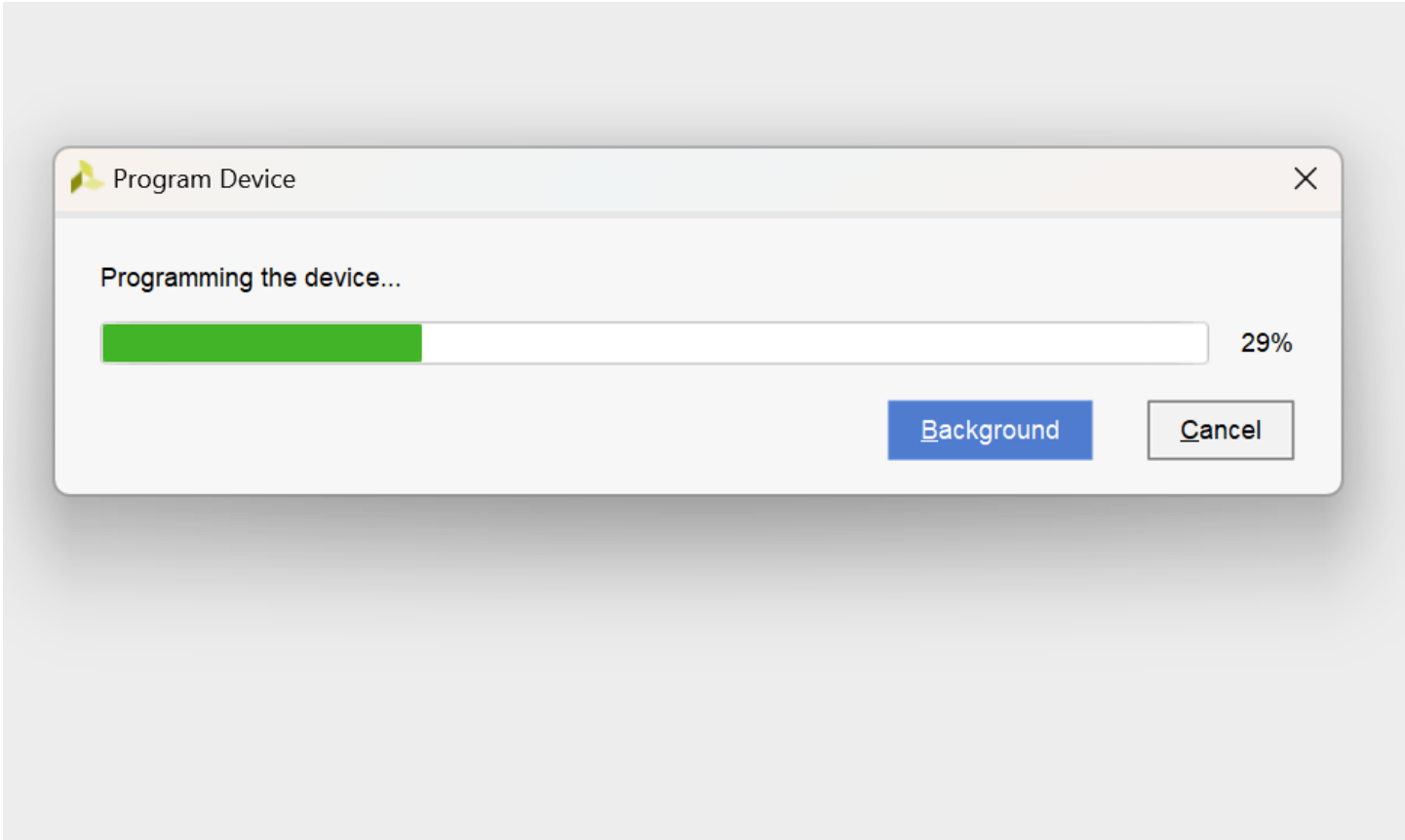
6. 选中提供的比特流文件fpga_top_20250113_cmpr.bin，点击ok



7. 点击Program进行比特流文件下载

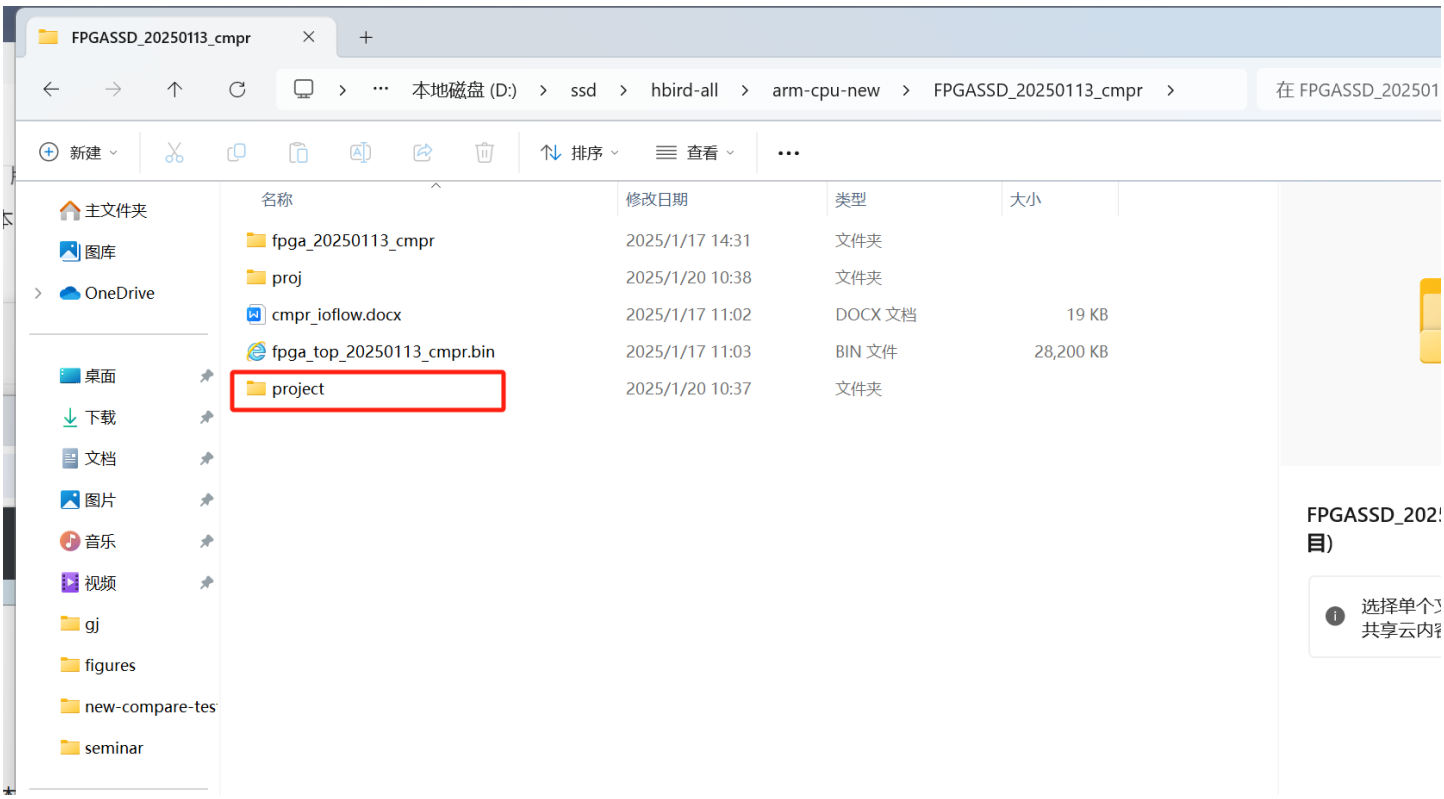

Program Device
Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.
Bitstream file:
ll/arm-cpu-new/FPGASSD_20250113_cmpr/fpga_top_20250113_cmpr.bin
Debug probes file:
Enable end of startup check
Program
Cancel

8. 等待下载完成

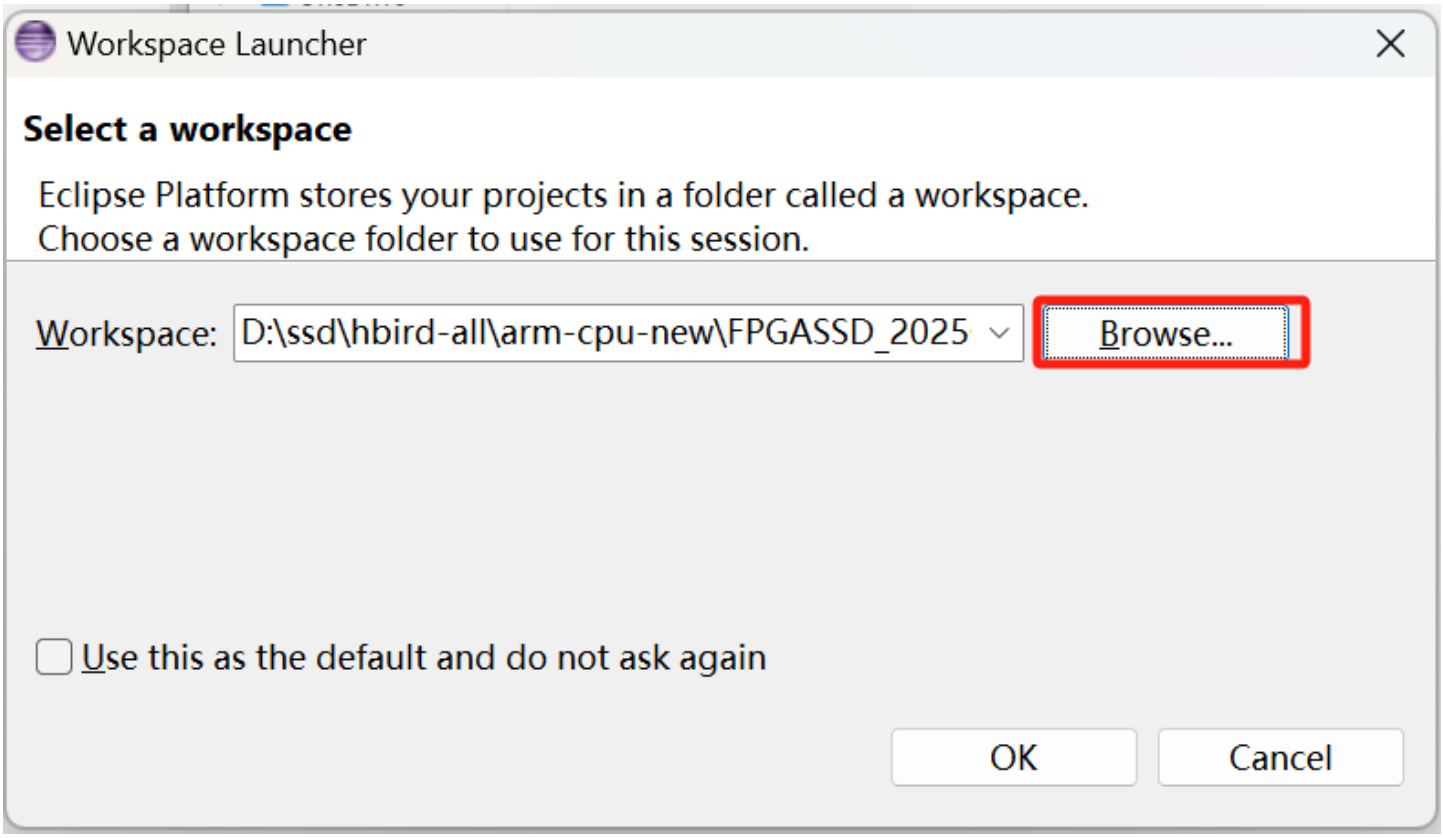

Program Device
Programming the device...
29%
Background
Cancel

3.3.4 创建工程和编译、下载

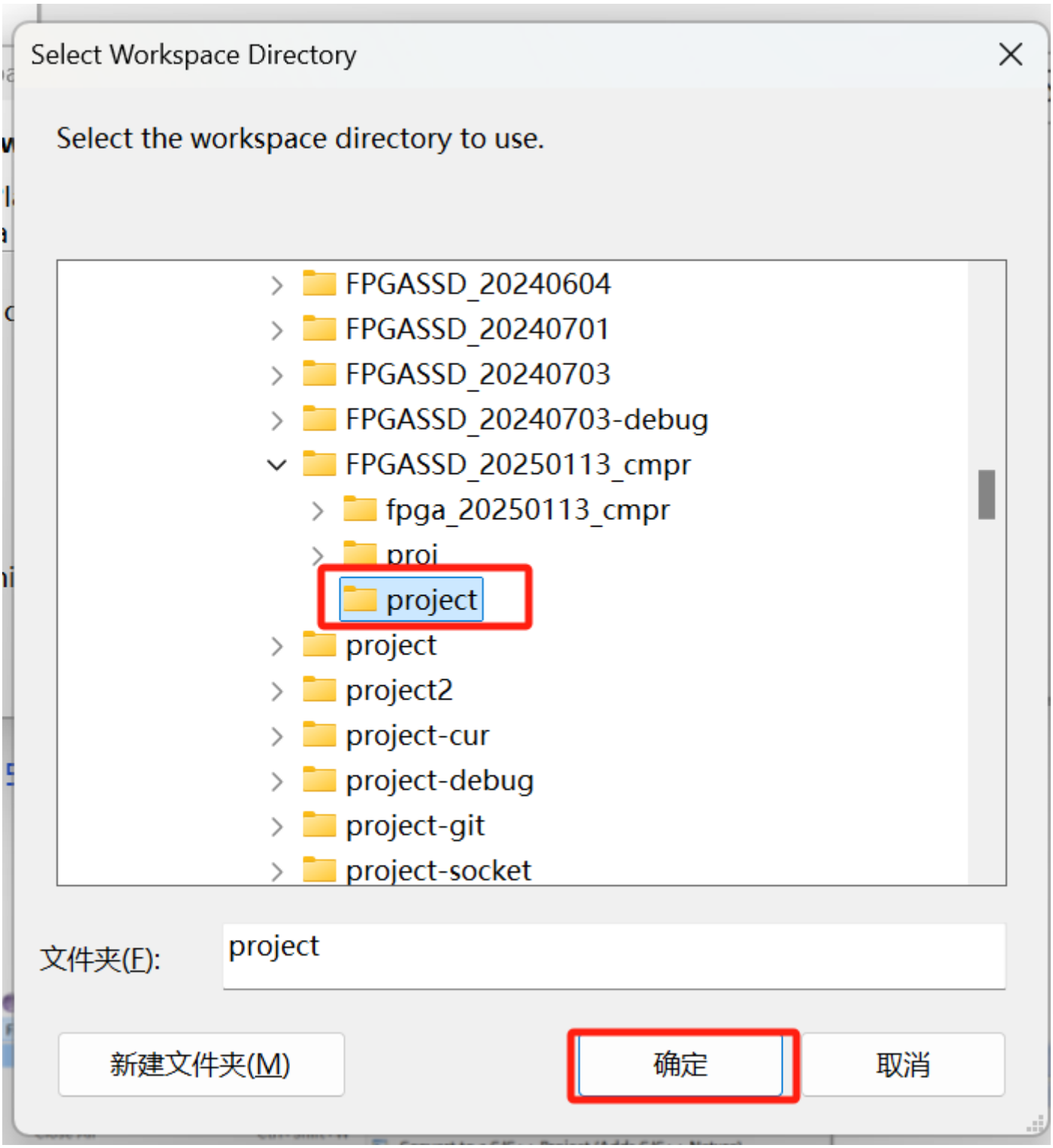
1. 创建工程文件夹project



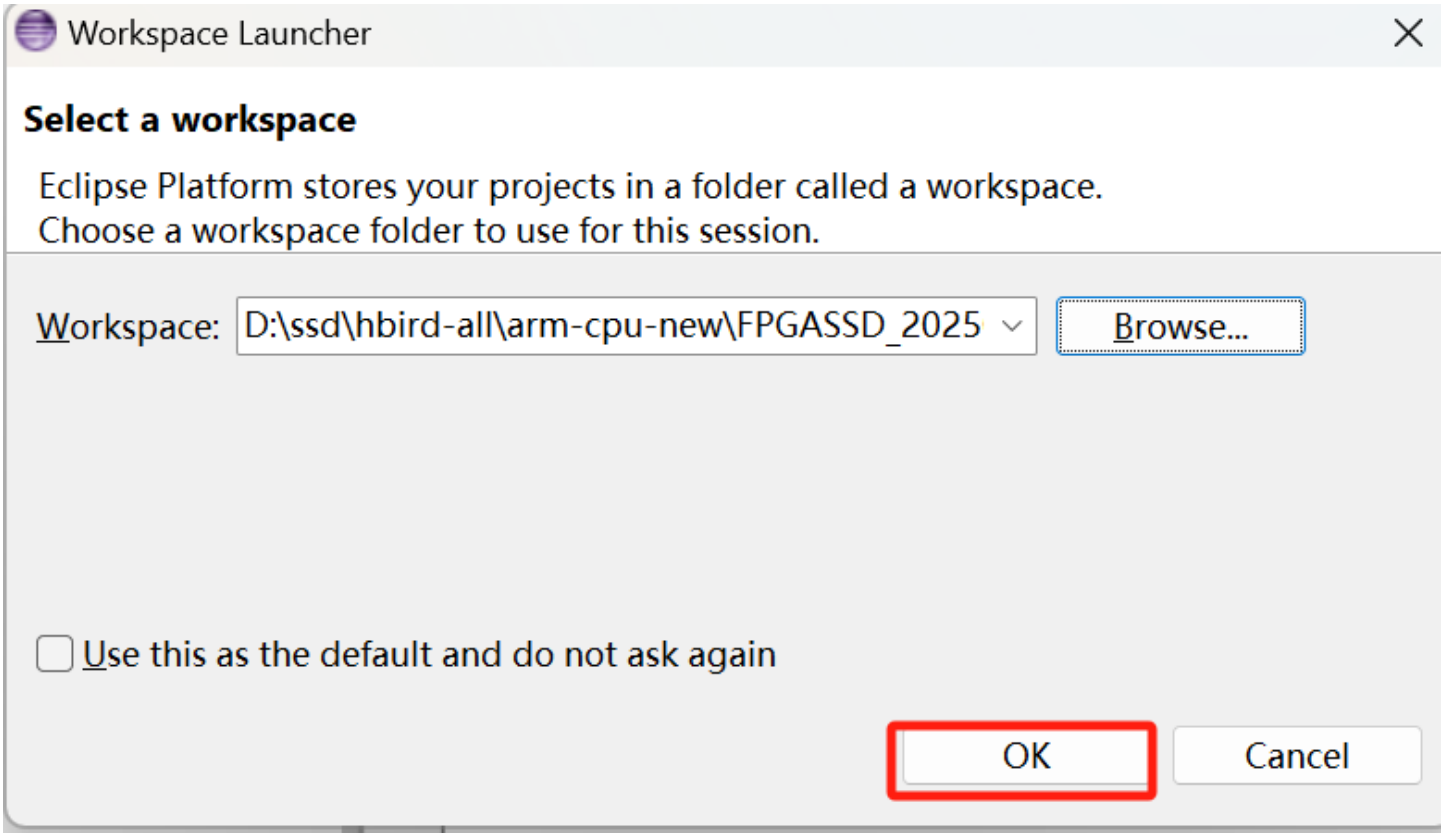
2. 打开DS-5软件， 点击Browse



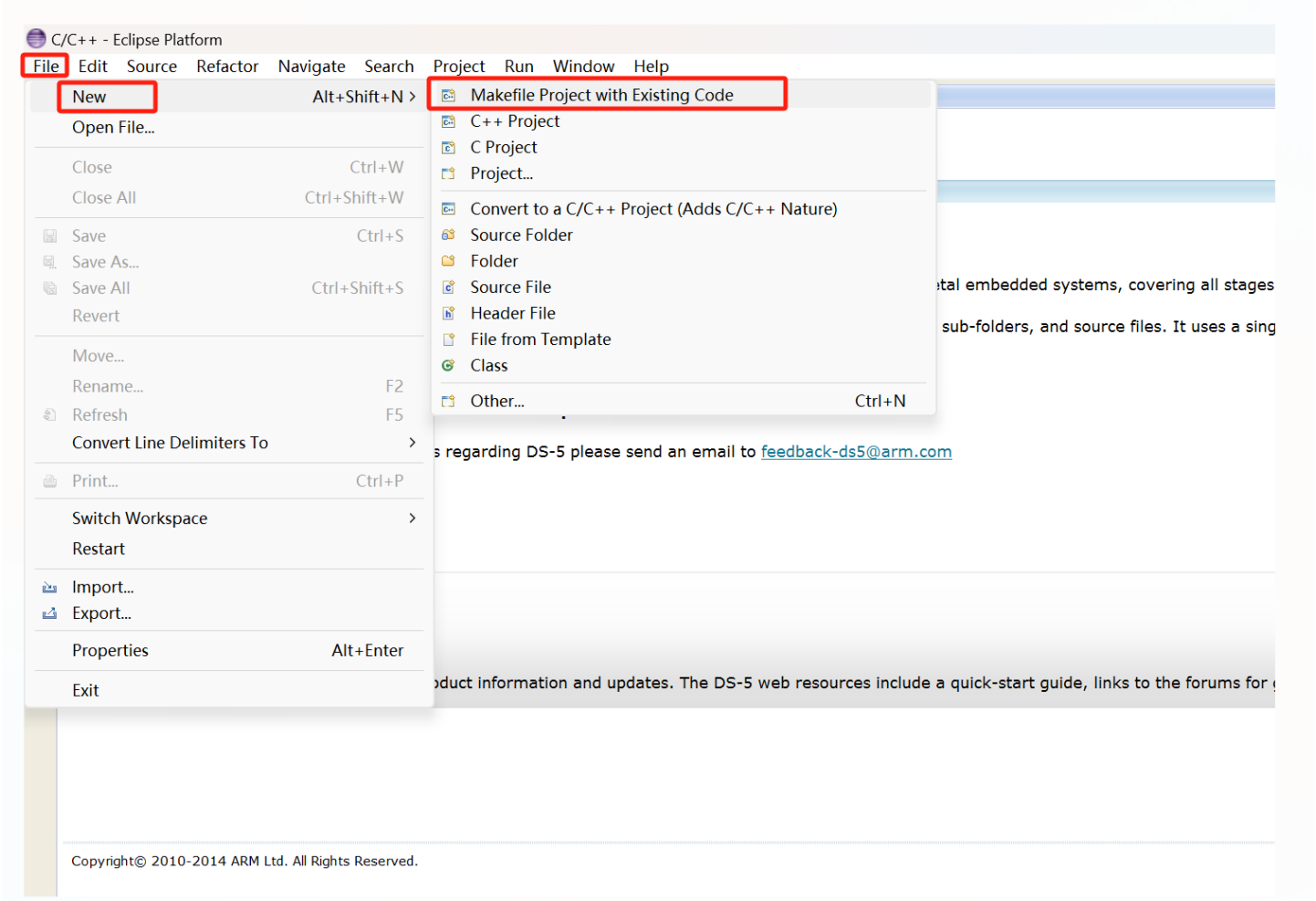
3. 选中之前创建的project文件夹，并点击确定。


Select Workspace Directory
Select the workspace directory to use.
FPGASSD_20240604
FPGASSD_20240701
FPGASSD_20240703
FPGASSD_20240703-debug
FPGASSD_20250113_cmpr
fpga_20250113_cmpr
proj
project
project
project2
project-cur
project-debug
project-git
project-socket
文件夹(F):
project
新建文件夹(M)
确定
取消

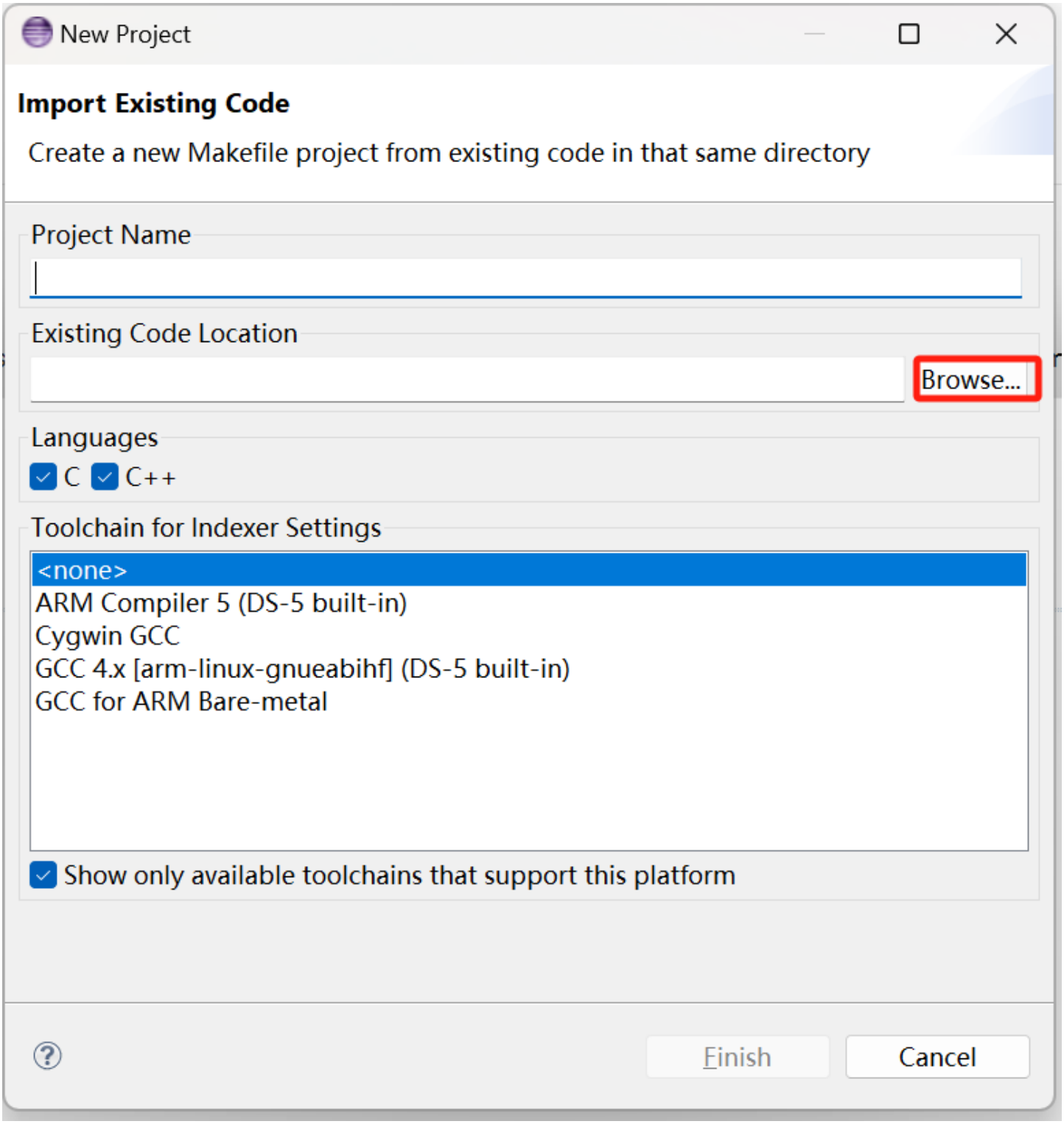
4. 点击ok



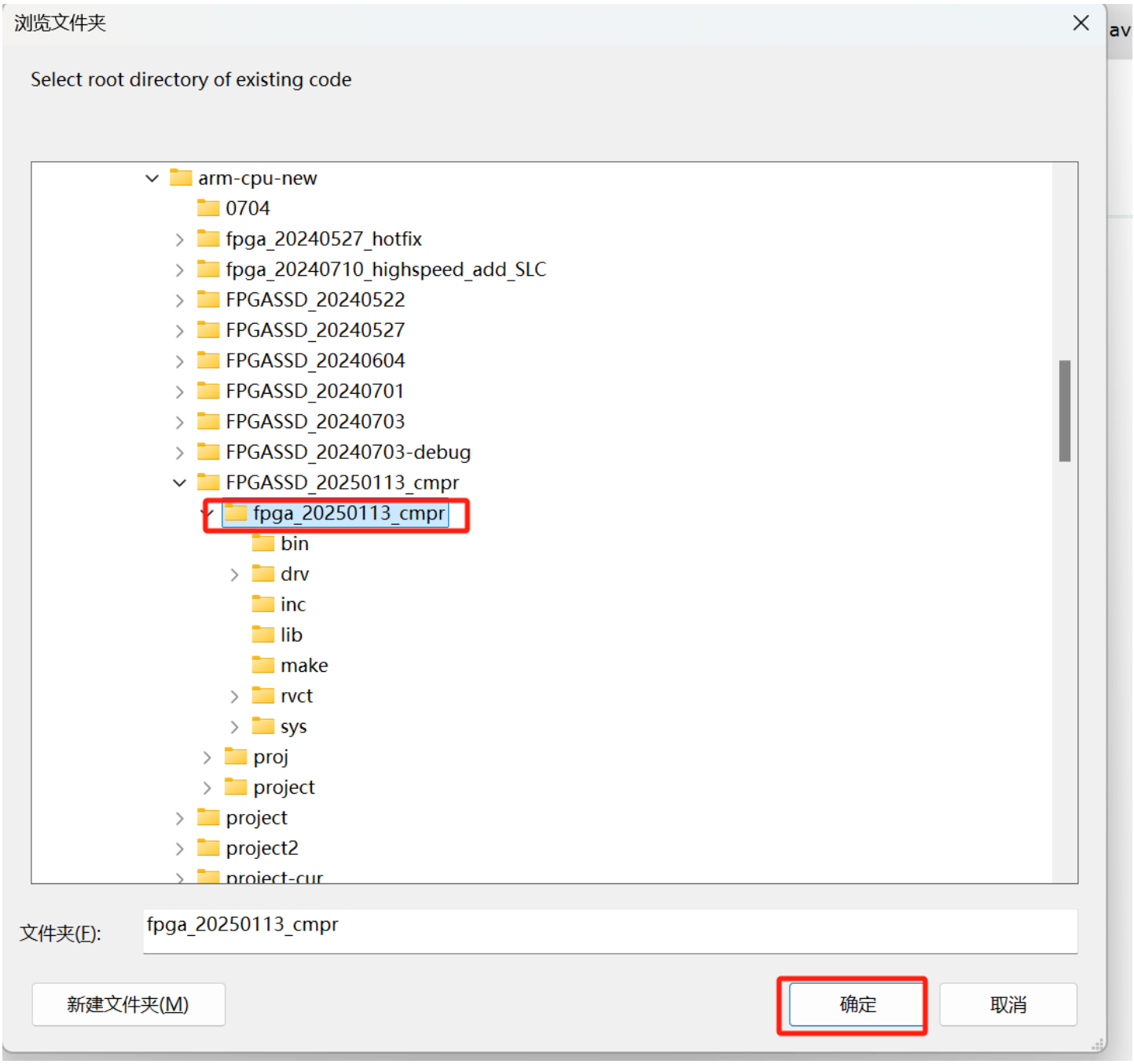
5. 进入DS-5界面，左键点击左上角File,再点击New,继续点击Makefile Project with Existing Code



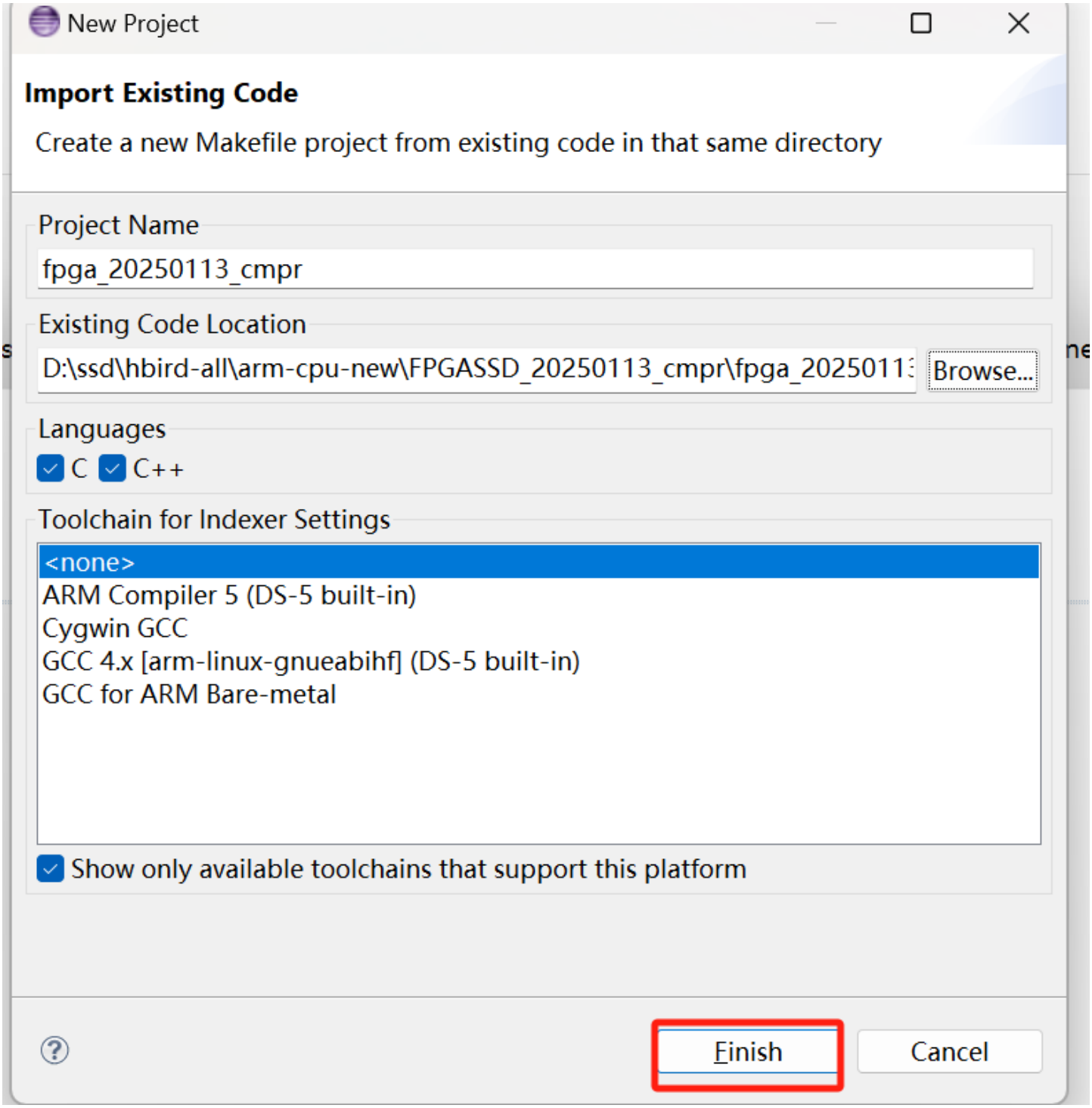
6. 点击Browse



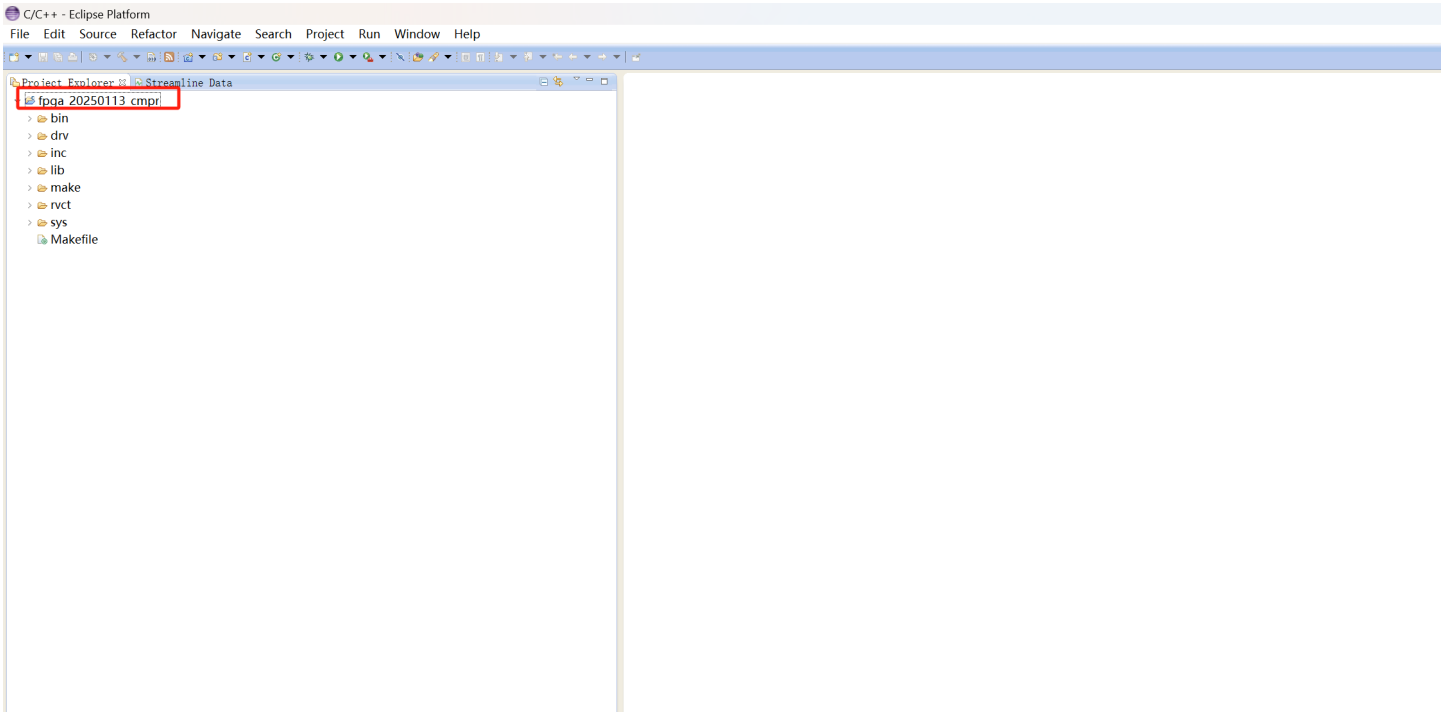
7. 选中代码所在的文件夹fpga_20250113_cmpr，点击确认


浏览文件夹
Select root directory of existing code
arm-cpu-new
0704
fpga_20240527_hotfix
fpga_20240710_highspeed_add_SLC
FPGASSD_20240522
FPGASSD_20240527
FPGASSD_20240604
FPGASSD_20240701
FPGASSD_20240703
FPGASSD_20240703-debug
FPGASSD_20250113_cmpr
fpga_20250113_cmpr
bin
drv
inc
lib
make
rvct
sys
proj
project
project
project2
文件夹(F):
fpga_20250113_cmpr
新建文件夹(M)
确定
取消

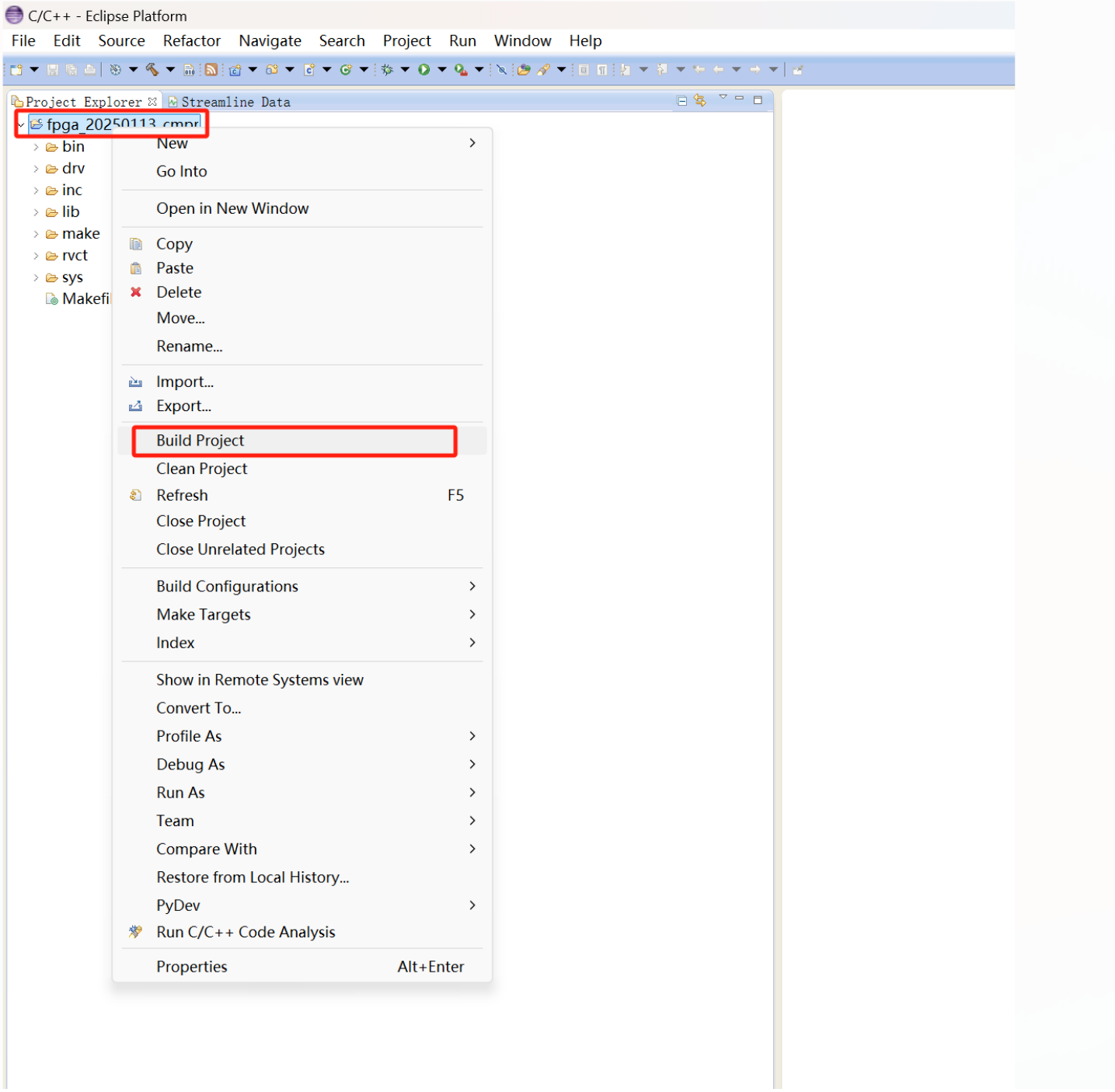
8. 点击Finish



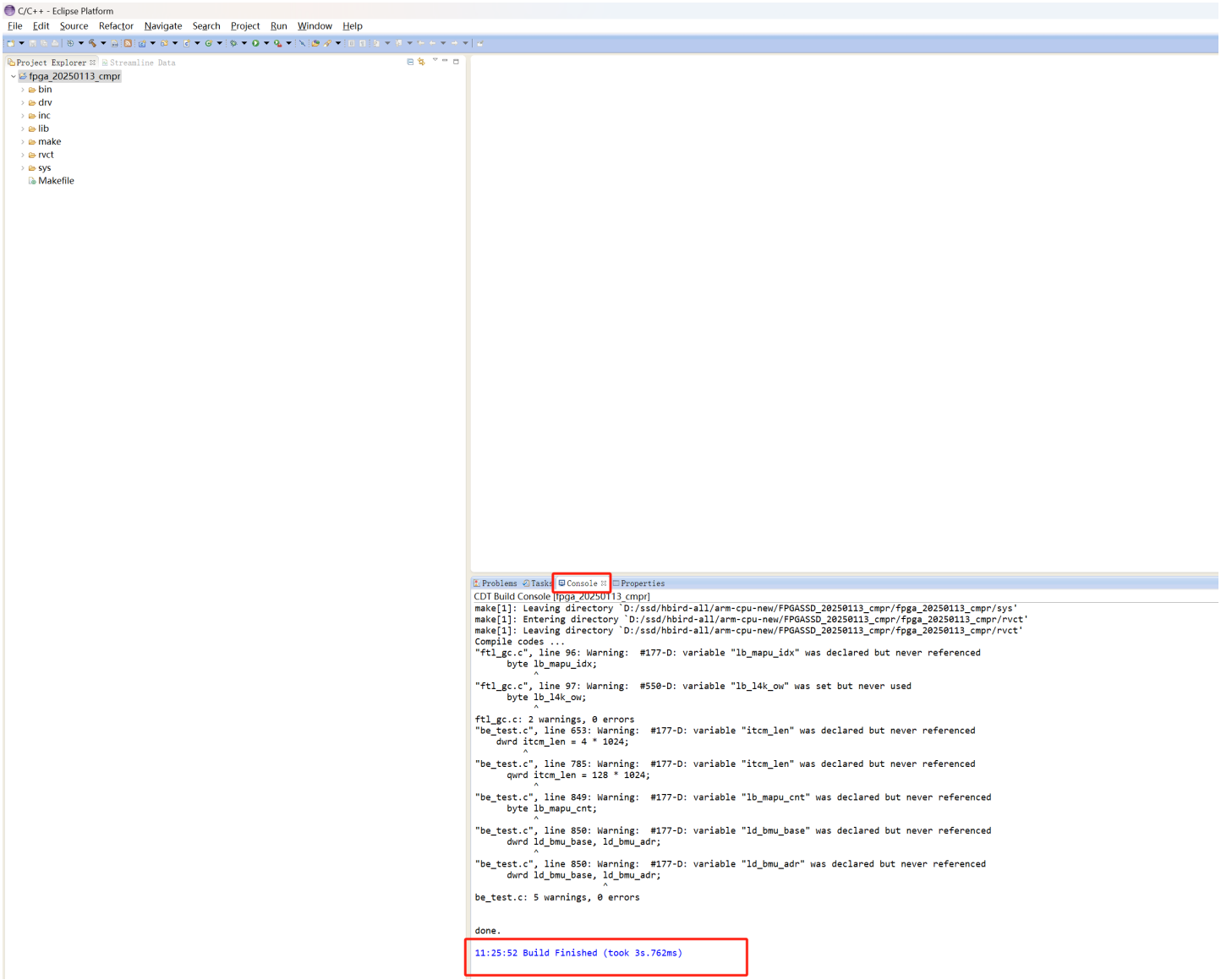
9. 关掉欢迎页面，可以看到代码已经导入到工程中



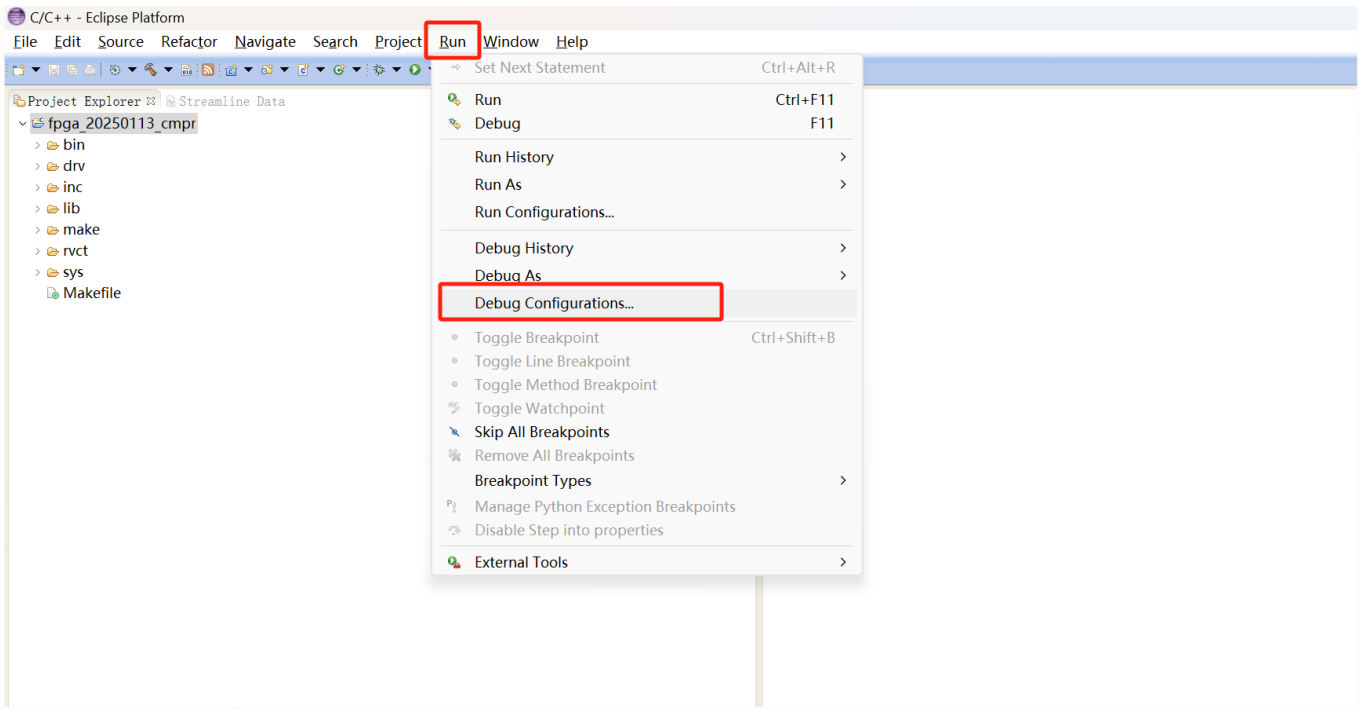
10. 右键点击fpga_20250113_compr,点击Build Project进行编译



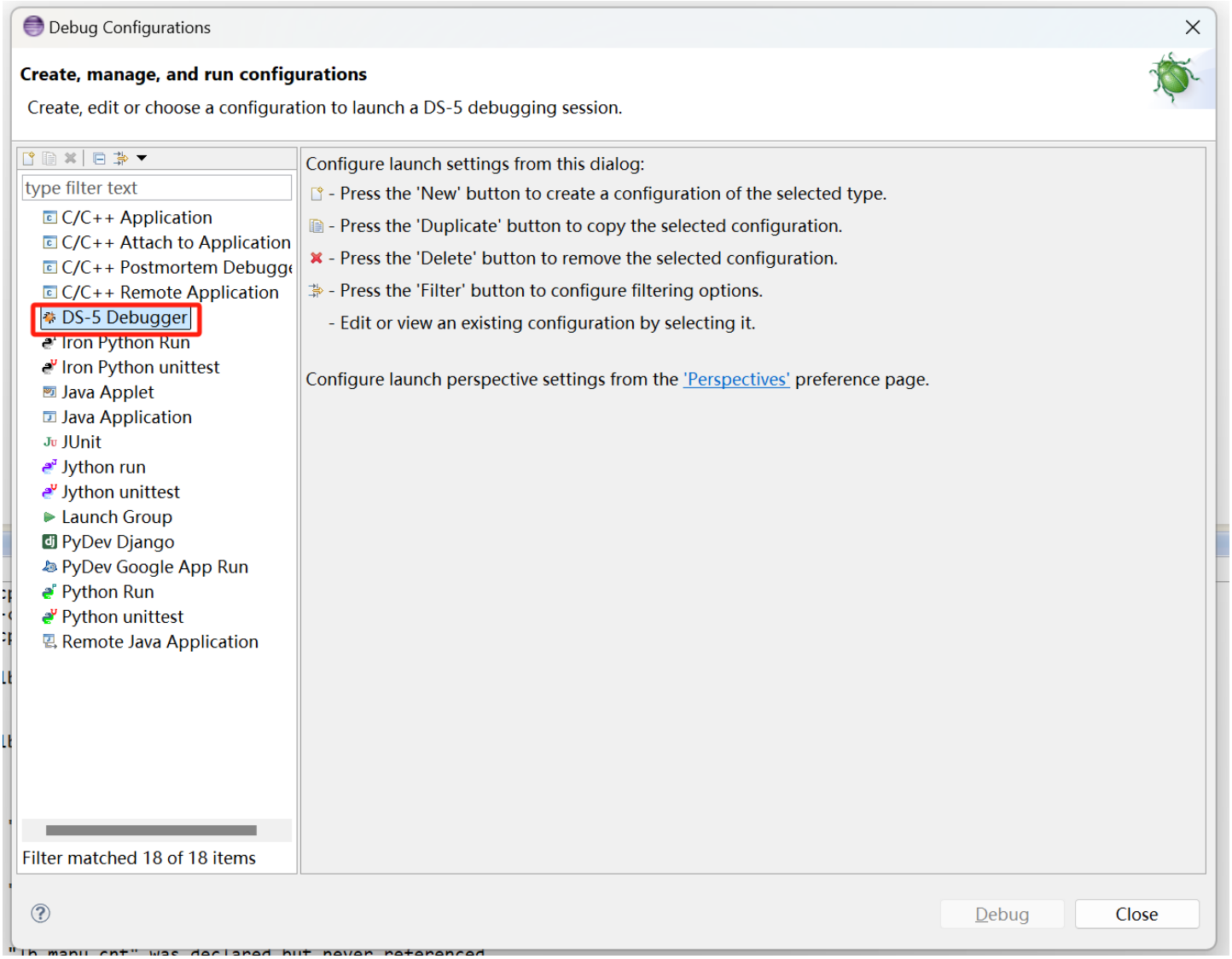
11. 编译完成后，可以点开Console，查看编译详情，如果编译不成功，会在此处显示错误信息，方便修改代码错误。



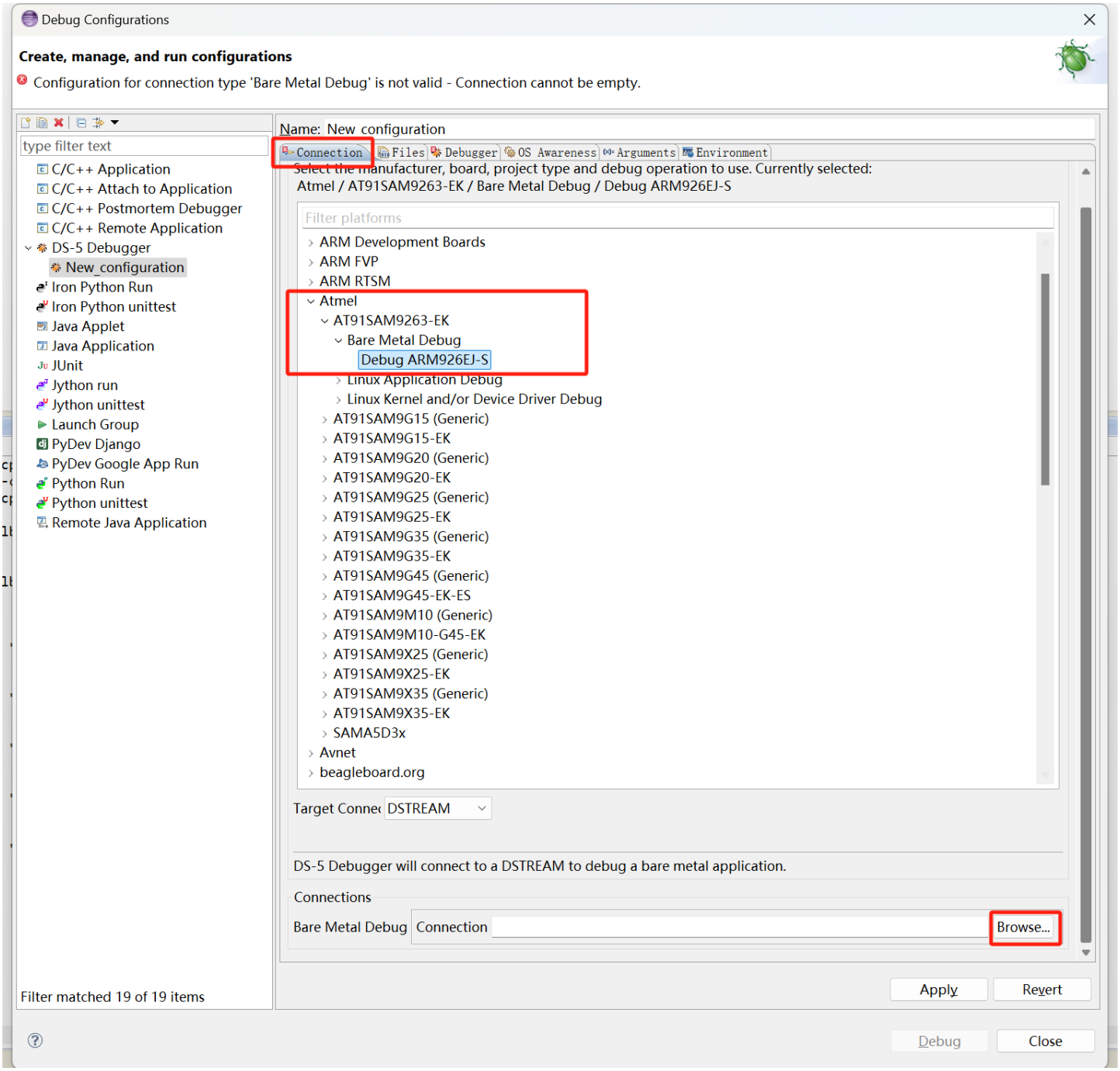
12. 编译成功之后，可以下载固件,先点击Run,然后点击Debug Configurations



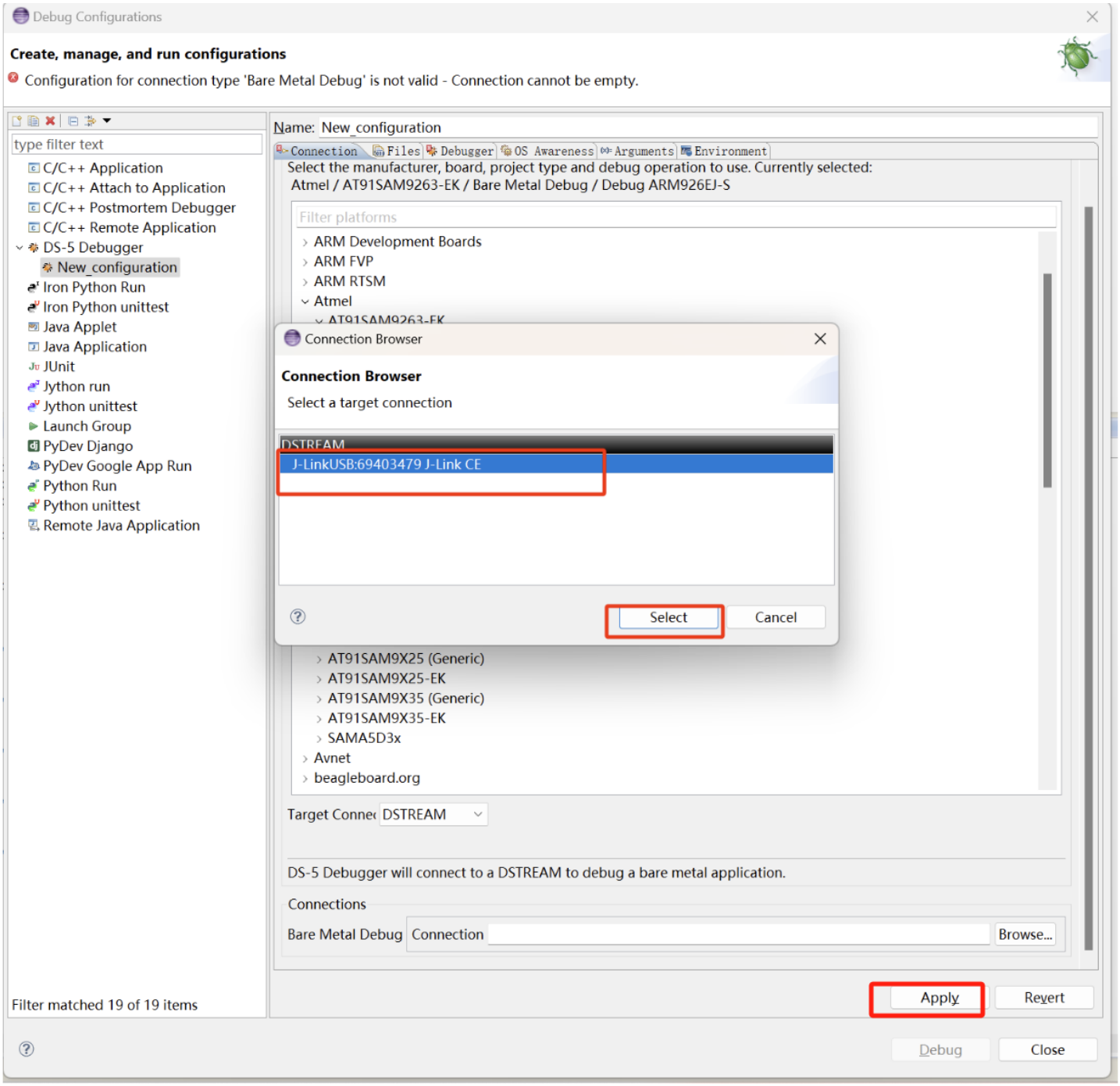
双击DS-5 Debugger



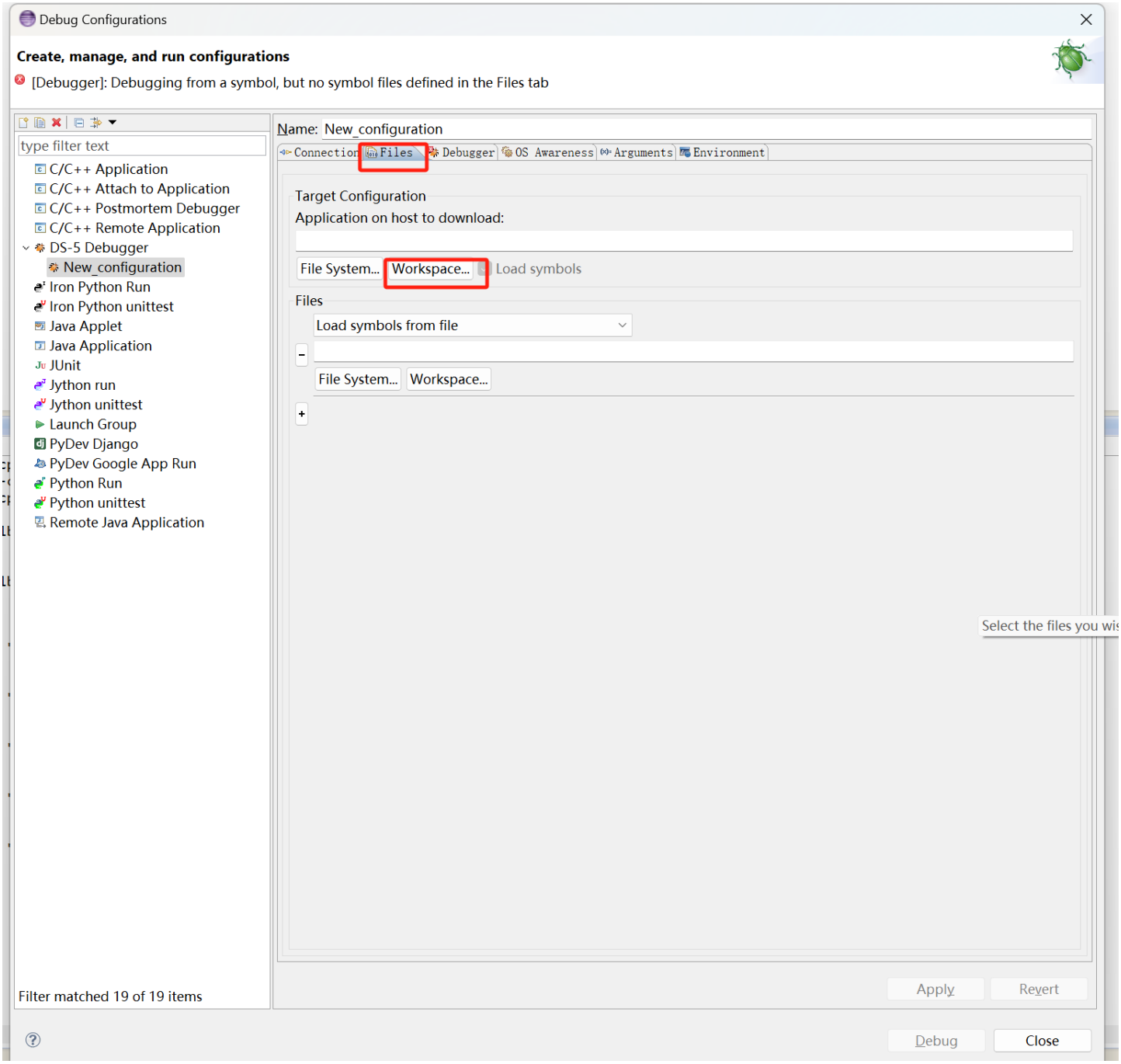
13. 在Connection中点击Atmel,选择Debug ARM926EJ-S,再点击Browse



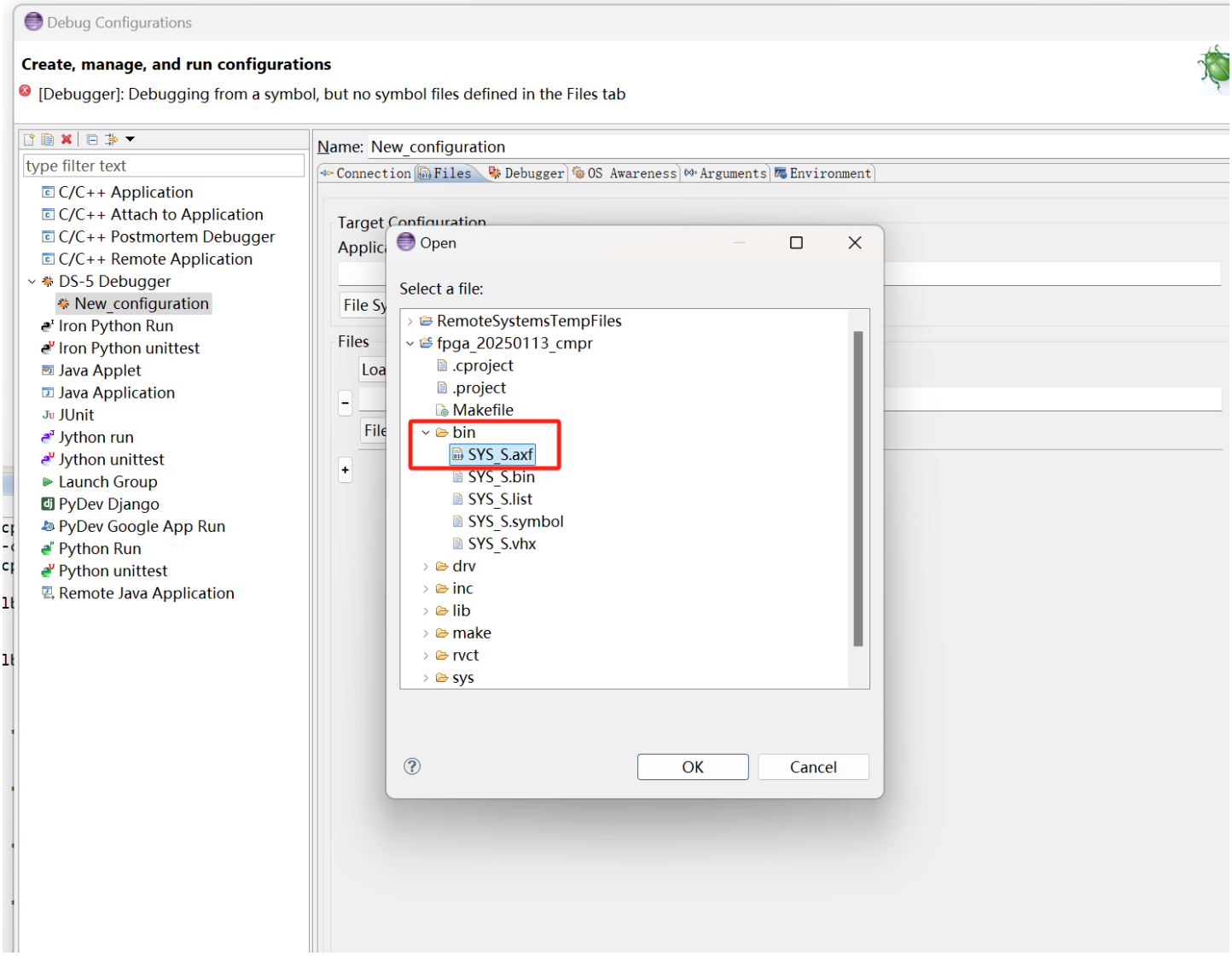
14. 依次点击J-LinkUSB，点击Select，再点击Apply



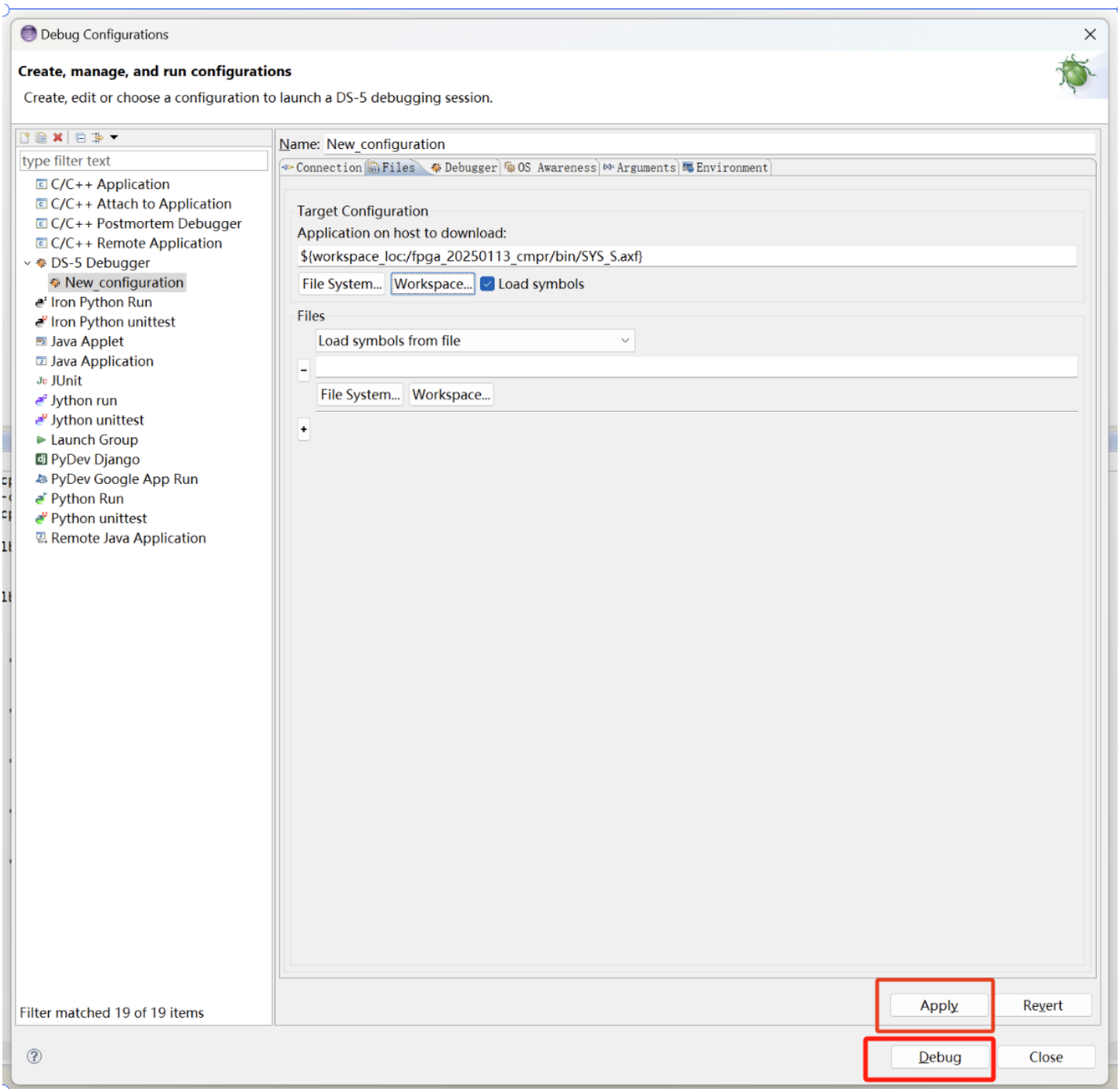
点击Files,再点击Workspace



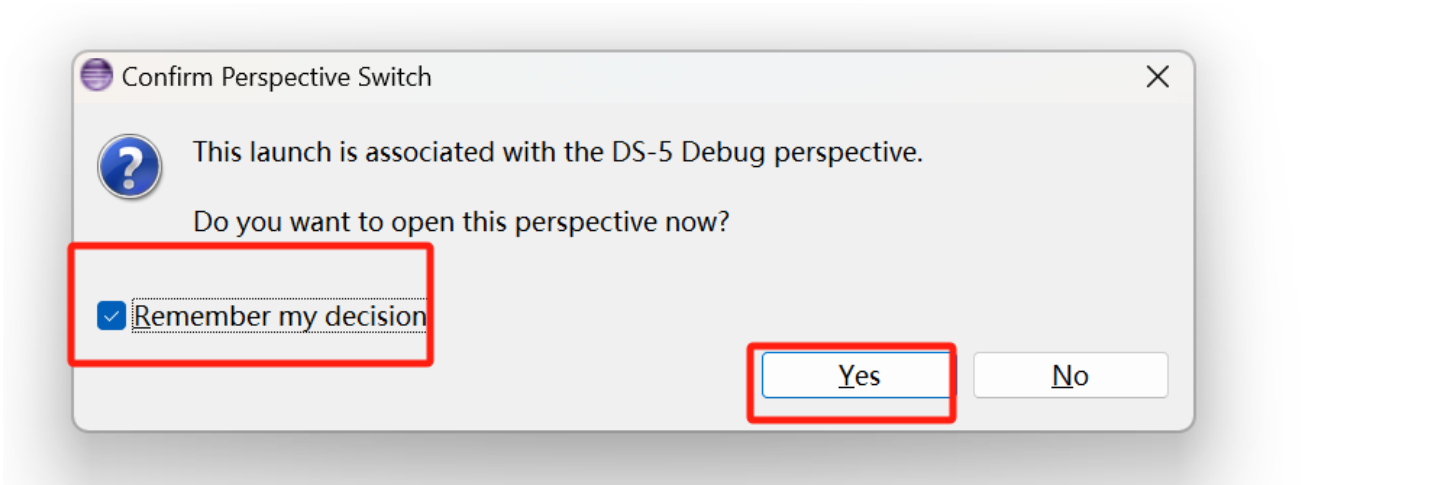
15. 点击bin文件夹中的SYS_S.axf文件，点击OK.



16. 点击Apply，再点击Debug

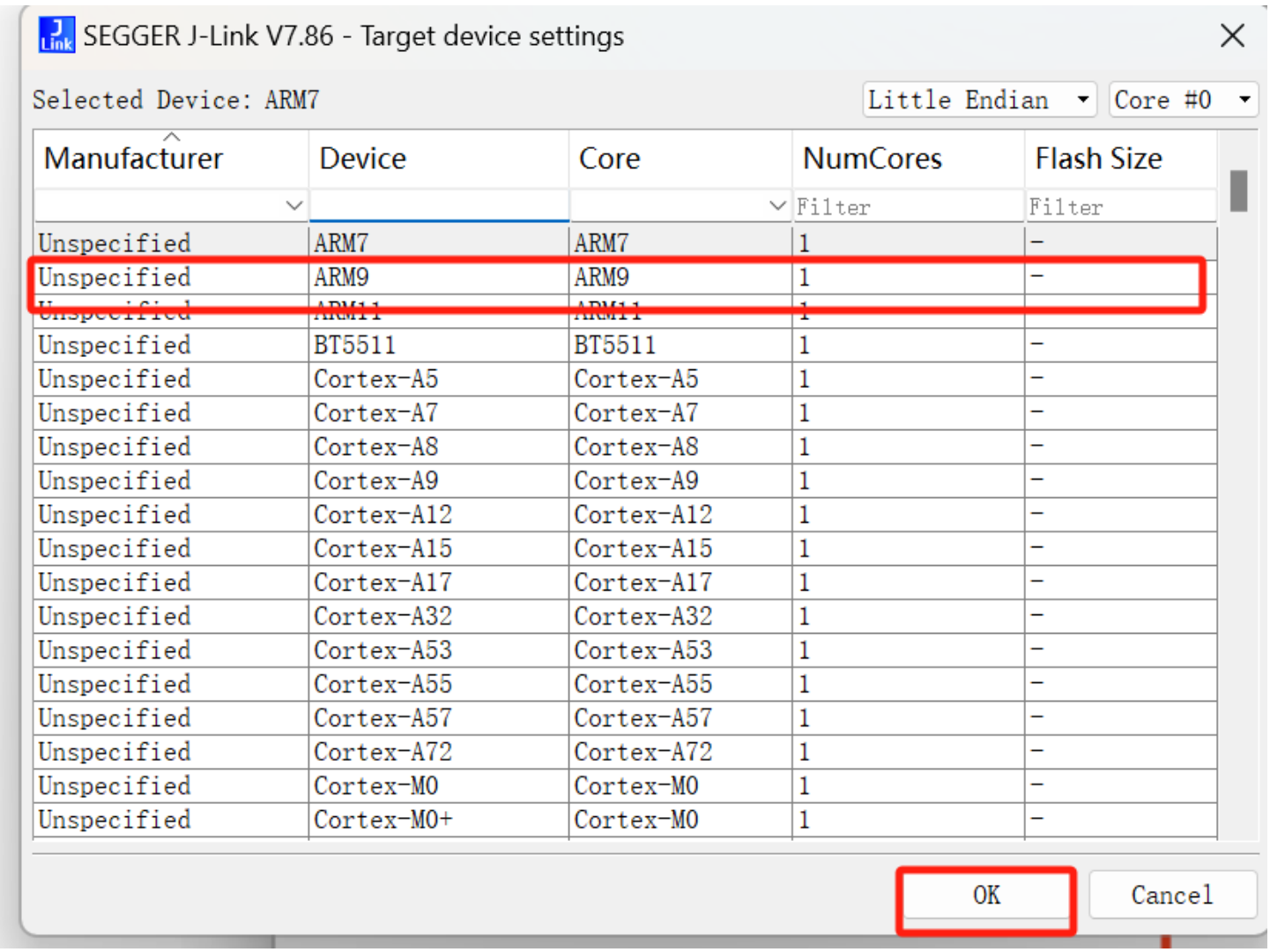

Debug Configurations
Create, manage, and run configurations
Create, edit or choose a configuration to launch a DS-5 debugging session.
type filter text
C/C++ Application
C/C++ Attach to Application
C/C++ Postmortem Debugger
C/C++ Remote Application
DS-5 Debugger
New_configuration
Iron Python Run
Iron Python unittest
Java Applet
Java Application
JUnit
Jython run
Jython unittest
Launch Group
PyDev Django
PyDev Google App Run
Python Run
Python unittest
Remote Java Application
Filter matched 19 of 19 items
Name: New_configuration
Connection
Files
Debugger
OS Awareness
Arguments
Environment
Target Configuration
Application on host to download:
${workspace_loc:/fpga_20250113_cmpr/bin/SYS_S.axf}
File System...
Workspace...
Load symbols
Files
Load symbols from file
File System...
Workspace...
Apply
Revert
Debug
Close

17. 点击Yes


Confirm Perspective Switch
This launch is associated with the DS-5 Debug perspective.
Do you want to open this perspective now?
Remember my decision
Yes
No

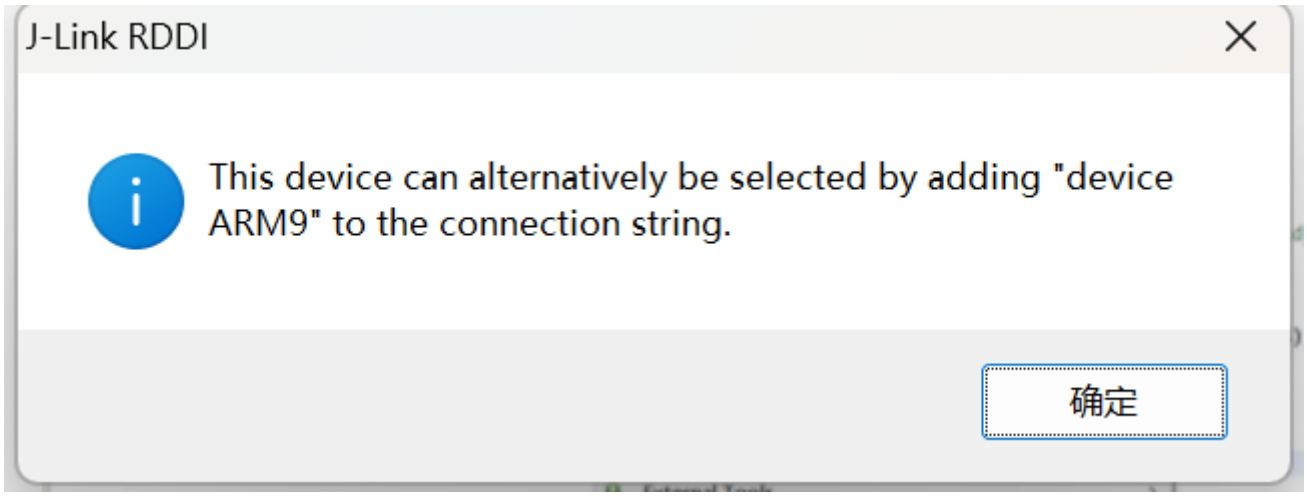
18. 点击是

19. 点击ARM9，再点击OK

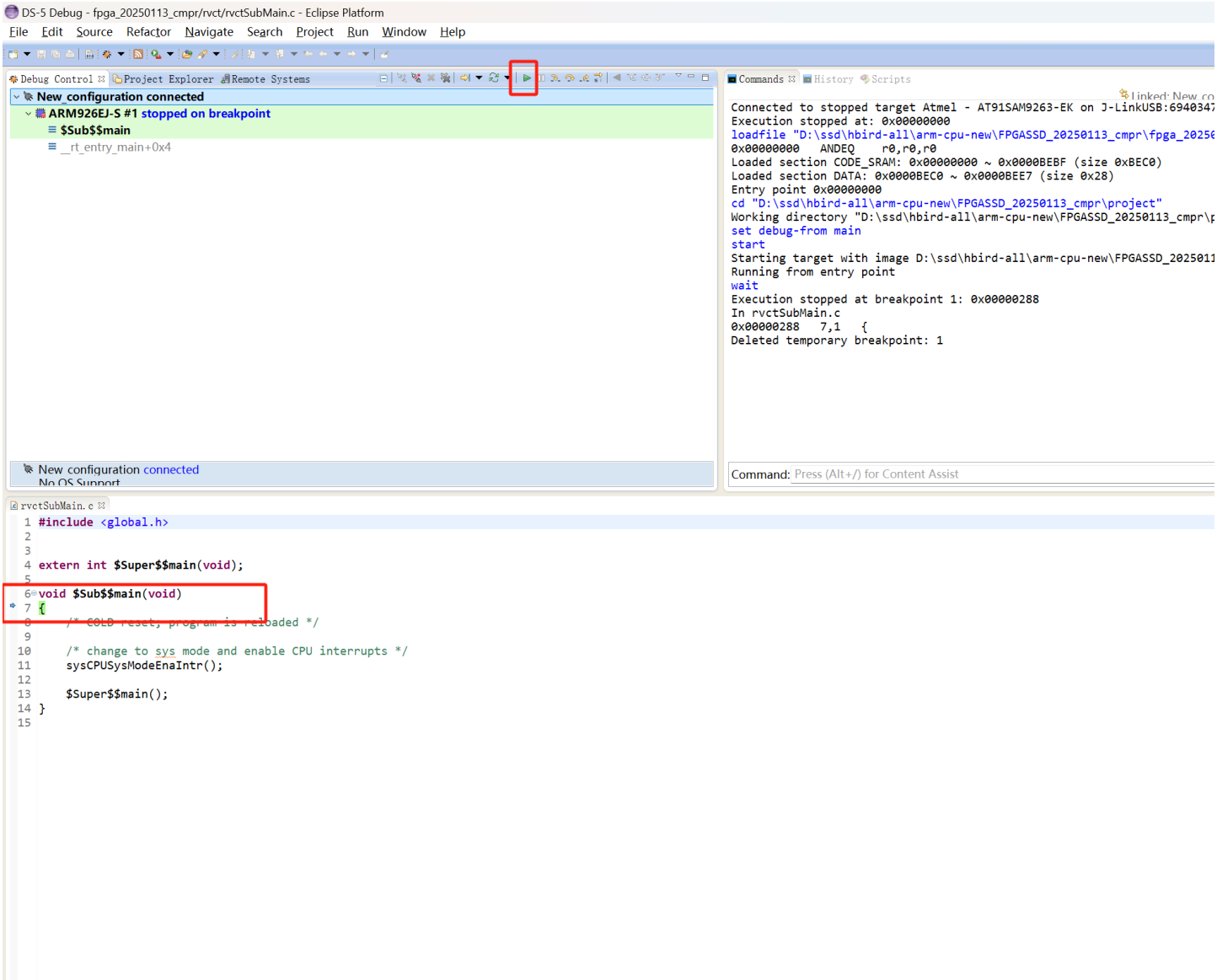


Manufacturer	Device	Core	NumCores	Flash Size
			Filter	Filter
Unspecified	ARM7	ARM7	1	-
Unspecified	ARM9	ARM9	1	-
Unspecified	ARM11	ARM11	1	
Unspecified	BT5511	BT5511	1	-
Unspecified	Cortex-A5	Cortex-A5	1	-
Unspecified	Cortex-A7	Cortex-A7	1	-
Unspecified	Cortex-A8	Cortex-A8	1	-
Unspecified	Cortex-A9	Cortex-A9	1	-
Unspecified	Cortex-A12	Cortex-A12	1	-
Unspecified	Cortex-A15	Cortex-A15	1	-
Unspecified	Cortex-A17	Cortex-A17	1	-
Unspecified	Cortex-A32	Cortex-A32	1	-
Unspecified	Cortex-A53	Cortex-A53	1	-
Unspecified	Cortex-A55	Cortex-A55	1	-
Unspecified	Cortex-A57	Cortex-A57	1	-
Unspecified	Cortex-A72	Cortex-A72	1	-
Unspecified	Cortex-M0	Cortex-M0	1	-
Unspecified	Cortex-M0+	Cortex-M0	1	-

20. 点击确定



21. 此时固件已成功下载到板卡中，并暂停在main函数内，此时需要先打开MobaXterm软件，用MobaXterm连接板卡的串口输出，然后点击DS-5的三角形运行图标运行固件。



22. 当看到串口打印如下所示，则表示固件运行正常，此时对linux主板输入sudo reboot命令，对主板进行重启。

```
bad block: ce:0, chan:4, blk_ptr:c5
bad block: ce:0, chan:1, blk_ptr:d2
bad block: ce:3, chan:5, blk_ptr:d6
bad block: ce:1, chan:4, blk_ptr:db
bad block: ce:3, chan:1, blk_ptr:dc
bad block: ce:3, chan:4, blk_ptr:e4
bad block: ce:2, chan:6, blk_ptr:e7
bad block: ce:2, chan:7, blk_ptr:e9
bad block: ce:3, chan:5, blk_ptr:e9
bad block: ce:3, chan:0, blk_ptr:ea
bad block: ce:3, chan:5, blk_ptr:ea
bad block: ce:0, chan:2, blk_ptr:ee
bad block: ce:0, chan:3, blk_ptr:f6
bad block: ce:1, chan:5, blk_ptr:fc
bad block: ce:3, chan:5, blk_ptr:104
bad block: ce:0, chan:4, blk_ptr:10d
bad block: ce:0, chan:1, blk_ptr:10f
bad block: ce:2, chan:5, blk_ptr:11b
bad block: ce:2, chan:4, blk_ptr:121
bad block: ce:1, chan:6, blk_ptr:123
bad block: ce:2, chan:0, blk_ptr:124
bad block: ce:1, chan:1, blk_ptr:126
bad block: ce:1, chan:1, blk_ptr:127
bad block: ce:2, chan:6, blk_ptr:127
bad block: ce:3, chan:5, blk_ptr:12d
bad block: ce:1, chan:0, blk_ptr:135
bad block: ce:2, chan:3, blk_ptr:13c
bad block: ce:3, chan:4, blk_ptr:144
bad block: ce:0, chan:5, blk_ptr:147
bad block: ce:0, chan:2, blk_ptr:14d
bad block: ce:2, chan:7, blk_ptr:154
bad block: ce:1, chan:1, blk_ptr:156
bad block: ce:3, chan:2, blk_ptr:159
bad block: ce:3, chan:1, blk_ptr:160
bad block: ce:0, chan:4, blk_ptr:162
bad block: ce:2, chan:3, blk_ptr:162
bad block: ce:1, chan:3, blk_ptr:169
free blk num:285
real cap 3743(GB),report cap 2338(GB), op 37%
```

23. 主板输入sudo reboot命令进行重启。注意，此时不能将主板断电重启

```
3. 10.156.152.92        4. COM3 (Silicon Labs CP210x USB t

                • MobaXterm Personal Edition v22.1 •
              (SSH client, X server and network tools)

 ➤ SSH session to xhk@10.156.152.92
   • Direct SSH      :  ✔
   • SSH compression :  ✔
   • SSH-browser     :  ✔
   • X11-forwarding  :  ✔  (remote display is forwarded through SSH)

 ➤ For more info, ctrl+click on help or visit our website.

Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.19.0-43-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

扩展安全维护（ESM）Applications 未启用。

139 更新可以立即应用。
要查看这些附加更新，请运行：apt list --upgradable

17 个额外的安全更新可以通过 ESM Apps 来获取安装。
可通过以下途径了解如何启用 ESM Apps：at https://ubuntu.com/esm

Last login: Fri Jan 17 17:41:30 2025 from 10.156.152.182
zsh: corrupt history file /home/xhk/.zsh_history
%
➜  ~
zsh: corrupt history file /home/xhk/.zsh_history
➜  ~ uname -a
Linux xhk-X10DRi 5.19.0-43-generic #44~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Mon May 22 13:39:36 UTC 2 x86_64 x86_64
➜  ~
➜  ~
➜  ~ sudo reboot
```

24. 当主板重启成功时串口打印如下：

```
bad block: ce:3, chan:5, blk_ptr:ea
bad block: ce:0, chan:2, blk_ptr:ee
bad block: ce:0, chan:3, blk_ptr:f6
bad block: ce:1, chan:5, blk_ptr:fc
bad block: ce:3, chan:5, blk_ptr:104
bad block: ce:0, chan:4, blk_ptr:10d
bad block: ce:0, chan:1, blk_ptr:10f
bad block: ce:2, chan:5, blk_ptr:11b
bad block: ce:2, chan:4, blk_ptr:121
bad block: ce:1, chan:6, blk_ptr:123
bad block: ce:2, chan:0, blk_ptr:124
bad block: ce:1, chan:1, blk_ptr:126
bad block: ce:1, chan:1, blk_ptr:127
bad block: ce:2, chan:6, blk_ptr:127
bad block: ce:3, chan:5, blk_ptr:12d
bad block: ce:1, chan:0, blk_ptr:135
bad block: ce:2, chan:3, blk_ptr:13c
bad block: ce:3, chan:4, blk_ptr:144
bad block: ce:0, chan:5, blk_ptr:147
bad block: ce:0, chan:2, blk_ptr:14d
bad block: ce:2, chan:7, blk_ptr:154
bad block: ce:1, chan:1, blk_ptr:156
bad block: ce:3, chan:2, blk_ptr:159
bad block: ce:3, chan:1, blk_ptr:160
bad block: ce:0, chan:4, blk_ptr:162
bad block: ce:2, chan:3, blk_ptr:162
bad block: ce:1, chan:3, blk_ptr:169
free blk num:285
real cap 3743(GB),report cap 2338(GB), op 37%
ADMIN_SET_FEATURES: ctag:0, fid:7
ADMIN_IDENTIFY: ctag:1, cns:1
ADMIN_IDENTIFY: ctag:2, cns:2
ADMIN_CREATE_IO_CQ: ctag:3
ADMIN_CREATE_IO_SQ: ctag:4
ADMIN_IDENTIFY: ctag:5, cns:0
ADMIN_GET_LOG_PAGE: ctag:6, lid:2
ADMIN_IDENTIFY: ctag:c, cns:1
ADMIN_GET_LOG_PAGE: ctag:d, lid:2
ADMIN_GET_LOG_PAGE: ctag:e, lid:2
```

25. 重连主板，输入lsblk命令，可以看到OpenFlash的nvme0n1设备，此时就可以把OpenFlash当做一块SSD盘去使用了。

3. 10.156.152.92 | 4. COM3 (Silicon Labs CP210x USB

```
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.19.0-43-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

扩展安全维护（ESM）Applications 未启用。

139 更新可以立即应用。
要查看这些附加更新，请运行：apt list --upgradable

17 个额外的安全更新可以通过 ESM Apps 来获取安装。
可通过以下途径了解如何启用 ESM Apps：at https://ubuntu.com/esm

New release '24.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Mon Jan 20 16:17:56 2025 from 10.156.152.182
zsh: corrupt history file /home/xhk/.zsh_history
➜  ~
zsh: corrupt history file /home/xhk/.zsh_history
➜  ~ lsblk
NAME    MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
loop0     7:0    0     4K  1 loop /snap/bare/5
loop1     7:1    0    64M  1 loop /snap/core20/2379
loop2     7:2    0  63.7M  1 loop /snap/core20/2434
loop3     7:3    0  73.9M  1 loop /snap/core22/1722
loop4     7:4    0  73.9M  1 loop /snap/core22/1663
loop5     7:5    0   273M  1 loop /snap/firefox/5273
loop6     7:6    0 274.4M  1 loop /snap/firefox/5600
loop7     7:7    0 349.7M  1 loop /snap/gnome-3-38-2004/140
loop8     7:8    0 349.7M  1 loop /snap/gnome-3-38-2004/143
loop9     7:9    0   497M  1 loop /snap/gnome-42-2204/141
loop10    7:10   0 505.1M  1 loop /snap/gnome-42-2204/176
loop11    7:11   0  91.7M  1 loop /snap/gtk-common-themes/1535
loop12    7:12   0  12.9M  1 loop /snap/snap-store/1113
loop13    7:13   0  12.2M  1 loop /snap/snap-store/1216
loop14    7:14   0  44.3M  1 loop /snap/snapd/23258
loop15    7:15   0  44.4M  1 loop /snap/snapd/23545
loop16    7:16   0   564K  1 loop /snap/snapd-desktop-integration/247
loop17    7:17   0   568K  1 loop /snap/snapd-desktop-integration/253
sda       8:0    0 931.5G  0 disk
├─sda1    8:1    0   512M  0 part /boot/efi
└─sda2    8:2    0   931G  0 part /var/snap/firefox/common/host-hunspell
                                   /
nvme0n1 259:0    0   2.1T  0 disk
➜  ~
```

26. 现在可以使用fio对OpenFlash进行测试。

3.3.5 ARM核上电后启动流程

1. 当前使用的ARM核上电时会有reset的中断发生，然后cpu会固定跳转到0x0的位置取值执行（跳转到0x0是硬件固定行为）。

2. fw中boot的文件参考arm_test/rvct 目录，上电boot会用到的文件有：

- rvctRstHdlr.s：包含指针、堆栈和memory的一些初始化设置等，使用arm汇编指令
- rvctScatter：linkfile，指定代码和data位置
- rvctSubMain.c： SubMain 函数，用来在main函数之前设置mode
- rvctVectors.s：包含中断向量表，我们会通过linkfile将中断向量表固定到0x0的位置，然后在0x0位置会是一条跳转指令，跳到我们的初始化文件rvctRstHdlr.s中执行初始化设置
- sysCPU_a.s 设置cpu mode

3. boot flow

- cpu上电有reset中断发生，cpu跳转到0x0地址，所以fw会在0x0地址放一条跳转指令，跳转到真正的初始化函数
- Linkfile中将rvctVectors.s的中断向量表固定分配到0x0地址，rvctVectors.s会定义vect的sector段，然后在linkfile中将其固定分配到0x0地址

```
AREA Vect, CODE, READONLY

  IMPORT  Reset_Handler
  IMPORT  Undefined_Handler
  IMPORT  SWI_Handler
  IMPORT  Prefetch_Handler
  IMPORT  Abort_Handler
  IMPORT  Reserved_Handler
  IMPORT  IRQ_Handler
  IMPORT  FIQ_Handler

  EXPORT  DbgCode


  ENTRY

  LDR PC, Reset_Addr
  LDR PC, Undefined_Addr
  LDR PC, SWI_Addr
  LDR PC, Prefetch_Addr
  LDR PC, Abort_Addr
  LDR PC, Reserved_Addr
  LDR PC, IRQ_Addr
  LDR PC, FIQ_Addr

Reset_Addr      DCD Reset_Handler
Undefined_Addr  DCD Undefined_Handler
SWI_Addr        DCD SWI_Handler
Prefetch_Addr   DCD Prefetch_Handler
Abort_Addr      DCD Abort_Handler
Reserved_Addr   DCD Reserved_Handler
IRQ_Addr        DCD IRQ_Handler
FIQ_Addr        DCD FIQ_Handler
```

```
rvctScatter
RAM_LOAD 0x0
{
    CODE_SRAM +0
    {
        rvctVectors.s.o (Vect, +First)
        * (+RO-CODE)
        * (+RO-DATA)
    }
}

DTCM 0x00400000 0x20000
{
    DATA +0
    {
        * (+RW,+ZI)
    }

    HEAP +0 UNINIT
    {
        rvctHeap.s.o (+ZI)
    }

    STACKS_SPACE 0x0041FFFC EMPTY - 0x1000
    {
    }

    STACKS 0x0041FFFC UNINIT
    {
        rvctStack.s.o (+ZI)
    }
}
```

- cpu从0x0地址取值执行，然后跳转到Reset_Handler（rvctRstHdlr.s文件中）
- cpu执行初始化设置，最后跳转到 Sub $main（rvctSubMain.c文件中）
- Sub $main是编译器提供的补丁功能，在进入main()之前，会先执行 Sub $main(void)函数中的内容，最后当执行到 $Super$ $main()函数时，程序才开始跳转到main（）函数，去执行main（）函数的内容。

rvctScatter | rvctVectors.s

```
4
5       IMPORT  Reset_Handler
6       IMPORT  Undefined_Handler
7       IMPORT  SWI_Handler
8       IMPORT  Prefetch_Handler
9       IMPORT  Abort_Handler
10      IMPORT  Reserved_Handler
11      IMPORT  IRQ_Handler
12      IMPORT  FIQ_Handler
13
14      EXPORT  DbgCode
15
16
17      ENTRY
18
19      LDR PC, Reset_Addr
20      LDR PC, Undefined_Addr
21      LDR PC, SWI_Addr
22      LDR PC, Prefetch_Addr
23      LDR PC, Abort_Addr
24      LDR PC, Reserved_Addr
25      LDR PC, IRQ_Addr
26      LDR PC, FIQ_Addr
27
28  Reset_Addr       DCD Reset_Handler
29  Undefined_Addr   DCD Undefined_Handler
30  SWI_Addr         DCD SWI_Handler
31  Prefetch_Addr    DCD Prefetch_Handler
32  Abort_Addr       DCD Abort_Handler
33  Reserved_Addr    DCD Reserved_Handler
34  IRQ_Addr         DCD IRQ_Handler
35  FIQ_Addr         DCD FIQ_Handler
36
37
38  DbgCode              DCD     0
39  Reserved1            DCD     0
40  Reserved2            DCD     0
41  Reserved3            DCD     0
```

0x0地址指令

rvctRstHdlr.s

```
37  SVC_Stack   EQU UND_Stack + UND_Stack_Size
38  USR_Stack   EQU SVC_Stack + SVC_Stack_Size
39
40
41  ;       ENTRY
42      EXPORT  Reset_Handler
43
44  Reset_Handler
45
46  ;
47  ; Initialize stack pointer registers
48  ;
49
50      IMPORT  top_of_stacks       ; defined in rvctStack.s and located
51
52      LDR r0, =top_of_stacks
53
54      MSR CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit ; No interrupts
55      SUB r1, r0, #FIQ_Stack
56      MOV sp, r1
57
58      MSR CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit ; No interrupts
59      SUB r1, r0, #IRQ_Stack
60      MOV sp, r1
61
62      MSR CPSR_c, #Mode_ABT:OR:I_Bit:OR:F_Bit ; No interrupts
63      SUB r1, r0, #ABT_Stack
64      MOV sp, r1
65
66      MSR CPSR_c, #Mode_UND:OR:I_Bit:OR:F_Bit ; No interrupts
67      SUB r1, r0, #UND_Stack
68      MOV sp, r1
```

```
rvctRstHdlr.s
103         MOV r6, #0x0
104         MOV r7, #0x0
105         MOV r8, #0x0
106         MOV r9, #0x0
107         MOV r10, #0x0
108         MOV r11, #0x0
109         MOV r12, #0x0
110
111         B   COLD_RESET  ;alwasys jump to COLD_RESET
112
113
114         IMPORT  __main
115
116     COLD_RESET
117         MOV r1, #0x0
118         MOV r2, #0x0
119         B   __main  ; note use B not BL, because an
120
121         END
122
```

- cpu跳转到main（）执行

```
rvctSubMain.c
1   #include <global.h>
2
3
4   extern int $Super$$main(void);
5
6   void $Sub$$main(void)
7   {
8       /* COLD reset; program is reloaded */
9
10      /* change to sys mode and enable CPU interrupts */
11      sysCPUSysModeEnaIntr();
12
13      $Super$$main();
14  }
15
```

4. FTL框架介绍

4.1 L2P Table

目前L2P Table是由Firmware去填写、更新，由Hardware去进行查L2P表。每个L2P entry的大小是4字节，这4个字节用来存储物理地址。L2P phy_addr的位域分布（高位->低位）：block(9bit)-page(13bit)-ce(2bit)-ch(3bit)-frag(5bit)。

frag:一个frag的大小是4KB，一个page是16KB，目前是multi-plane操作，所以一个multi-plane page的frag有24个，frag占用5bit。

- ch：有8个channel，占3bit
- ce：（bank）一个ch有4个ce,占2位。
- page:一个block有4176个page，占13bit
- block:一个plane有363个block，占9bit
- plane:一个bank有6个plane，可以由frag得到plane的值，所以不占位

```
inc/global.h
   #define PCE_SHIFT   2
   #define BANK_SHIFT  2
   #define PAGE_SHIFT  13
   #define BLK_SHIFT   9
   #define PLANE_SHIFT 3
   #define FRAG_QNTY   24
   #define CHAN_QNTY   8
   #define PCE_QNTY    4
   #define BANK_QNTY   4
   #define FRAG_SHIFT  5
   #define CHAN_SHIFT  3
   #define BANK_SHIFT  2
```

L2P TABLE是一个大数组，位于DRAM的起始位置（FTL_L2P_BASE 0x80000000），即L2P table数组从0x80000000开始。DRAM的大小是2GB.

目前L2Ptable在写操作时是固件去填写映射关系，在读操作时是硬件进行查询，如果想改成固件去查询映射表，可以注释LGET_EN宏。L2P table目前只存储在内存中，没有下刷到nand进行持久化。

4.2 P2L Table

P2L Table里面存储了逻辑地址，每个P2L table entry对应一个逻辑地址，一个逻辑地址对应4KB数据。一个P2L table entry占用4byte，所以P2L table最多可以索引4G个逻辑地址，可以管理4G*4KB=16TB空间的nand容量。

目前每写4K个4KB数据时，就会产生对应16KB大小的P2L映射表。

为了提高固件运行的速度，给P2L table设置了3层缓存：cpu cache->SRAM->DRAM。因为ECC是以16byte为单位进行的，所以当写入4个P2L entry时，将这4个P2L entry下刷到SRAM中。用2个4KB的SRAM来存储P2L entry，这两个4KB SRAM空间是乒乓结构，即同一时间只有一个4KB可以被写入P2L entry，当一个4KB SRAM写满之后（写入了1024个P2L entry），将这4KB数据从SRAM下刷到DRAM中，如果此时有新的P2L entry需要写入SRAM时，将这个P2L entry写入到另一个的4KB SRAM中。当写了4096个P2L entry之后，将DRAM中16KB大小的P2L table下刷到nand中。

由此，我们可以理解每当在nand里写了4K个4KB数据时，会有一个存储P2L table的page下刷到nand中。这个关系在进行GC时，可以推测读上来数据对应的P2L entry的存储在在哪个位置。目前每个

superblock最后的4个frag用来存放P2L。

4.3 BMU（Buffer Management Unit）

BMU对SRAM和DRAM中的buffer进行配置和管理。SRAM中buffer总大小为4MB，DRAM中buffer总大小为8MB，buffer单元大小4KB，由此计算出SRAM有1024个buffer，DRAM有2048个buffer，这些buffer根据使用需求进行分发。

buffer编号是0-3071，其中0-2047为DRAM的空间，2048-3071为SRAM的空间，具体将数据放入SRAM还是DRAM是可以进行配置的。

目前buff的管理主要是硬件实现的。硬件实现了如下功能：

- 给host读写命令分配buff；
- 在处理host读命令时，去查找数据是否缓存在buff中；
- 释放已经完成数据搬移的buff;

在Firmware给Flash controller发送写操作时，此时数据不一定搬到buf里了，硬件会进行控制，等到数据搬移到buf之后再对nand进行写操作。在进行读操作时，可以软件或者硬件分配buff。目前DRAM的大小是2GB，但是硬件管理的buff空间只有10MB DRAM+2MB SRAM，如果需要固件提前缓存一些热数据在DRAM中，可以在DRAM中划分部分区域进行extra read buf管理，在发送nand读命令前需要配置寄存器rb_cmd_qidx为QLNK_DMQ即可，并且对extra read buf用hash表进行管理。

4.4 GC

当满足GC的条件时，开始进行GC。首先挑选需要做gc的block，然后将这个block的P2L table读上来，每次读8个4KB到SRAM中（因为是multi-plane操作，所以nand支持一次读2个page数据）。

当P2L table读上来之后，使能硬件，让硬件去使用P2L table中存储的logical addr查找L2P table,获得physical addr。如果查表得到physical addr和正在进行GC的block地址相同，则认为这个physical addr存储的数据是有效的，在做GC时需要把这块数据读上来再写入到新的block中；如果查表得到physical addr和正在进行GC的block地址不同，则认为这个physical addr存储的数据是无效的，在做GC时就不需要把这块数据读上来，直接丢弃就行了。

当硬件完成P2L table中的地址对比之后，就可以将有效的数据块从nand读到DRAM中。然后给这块数据分配新的物理地址，更新L2P table，目前数据的metadata存储了这个数据块的物理地址和逻辑地址，因此还需要在OOB区域更新数据的metadata。在将数据写到新的block之前，firmware会让硬件去检查这个数据是否有host 的更新写，如果有，则将这个数据丢失，不写入新的block之中，如果没有host更新写，则将这块数据写入到新的物理地址中。

代码结构：

处理GC的函数是 `sys\ftl_gc.c` 中的 `fv_ftl_gc()` 函数，以下是对这个函数的解释说明。

```
    if((gb_gcblk_vld == 0) && (gb_gcblk_que < 2))
    {
        gd_min_vcnt = 0xffffffff;
        for(lw_cmd_blk = 1; lw_cmd_blk < RBLK_QNTY; lw_cmd_blk = lw_cmd_blk + 1)
        {...}
        ...
        gd_ht_tmp = gd_min_vcnt / (FRAG_QNTY * BKCH_QNTY) + 1;
        gd_gc_tmp = ((PAGE_QNTY - P2LP_NUM) > gd_ht_tmp) ? (PAGE_QNTY - P2LP_NUM - gd_ht_tmp) : 1;
        gb_gcblk_que++;
        if(gb_gcblk_que == 1)
        {
            gd_ht_weight = gd_ht_tmp;
            gd_gc_weight = gd_gc_tmp;
        }
        fv_uart_print("get gc source blk: blk_num:%x, vld_cnt:%d, gcblk_que:%d, ht_weight:%d, gc_weight:%d\r\n", gw_gcblk_num, gd_min_vcnt, gb_gcblk_que, gd_ht_tmp, gd_gc_tmp);
    }

```

- 查找valid count最小的block，将其作为GC的对象块。
- 计算 `gd_ht_weight` 和 `gd_gc_weight` ，包括后面提到的。 `gd_ht_token` 和 `gd_gc_token` ，这四个变量用来控制当前做GC还是处理host写命令，保证GC的速度可以跟上host写的速度。

```
    while(gb_gcblk_vld && !gb_p2lp_nrd[gb_p2lp_rds])
    {
        for(lb_mapu_ptr = 0; lb_mapu_ptr < 4; lb_mapu_ptr = lb_mapu_ptr + 1)
        {
            rb_cmd_qidx = QLNK_SMQ;
            ...
            rw_cmd_bptr = (SMDT_BASE << 9) | ((word)gb_p2lp_rds << 2) | (word)lb_mapu_ptr;
            rw_cmd_type = READ_CPTR | HIGH_PRI;
        }
        gb_p2lp_nrd[gb_p2lp_rds] = 1;
        gw_p2lp_blk[gb_p2lp_rds] = gw_gcblk_num;
        gb_p2lp_npg[gb_p2lp_rds] = gb_pmsg_npg[gw_pmsg_rdp];
        gb_p2lp_rds = !gb_p2lp_rds;

        if(gw_pmsg_rdp == PMSG_END)
        {
```

```
            gw_pmsg_rdp = 0;
            gb_gcblk_vld = 0;
            fv_uart_print("gc blk finish(read p2l): blk_num:%x, vld_cnt:%d\r\n", gw_gcblk_num, gd_vcnt_tbl[gw_gcblk_num]);
        }
        else
        {
            gw_pmsg_rdp++;
        }
    }
```

- 写入4K个4KB数据后，会写入一个p2l page，即16KB。
- `rb_cmd_qidx = QLNK_SMQ`：表示操作类型，`QLNK_SMQ`宏表示从NAND往SRAM或DRAM的buffer里面读取数据。这之后读取完成的信号将会送到`rb_smrd_vld`寄存器中。
- 采用ping-pong结构读取p2l page的内容，一次可以发出2个p2l page的read操作。
  - `gb_p2lp_rds`：控制ping-pong结构。
  - `gb_p2lp_nrd[2]`（nand read）：表示对应buffer是否被占用，若为1则被占用；若为0则被释放。此处代码会将其赋值为1，释放的代码在后面。
  - `gw_p2lp_blk[2]`：因为临界状况下，ping-pong结构读取的p2l page可能不是一个block下的，因此需要记录当前读取p2l page是哪个superblock的。
  - `rw_cmd_bptr`：控制此次存储p2l page的buffer。
    - Buffer 0-7由硬件默认预留，用来读取p2l page。
    - 当`gb_p2lp_rds`为0时，使用buffer 0-3；为1时使用buffer 4-7。
- `gw_pmsg_rdp`：统计一个block中读了多少次p2l page。
  - 对于一个superblock而言，容量确定的情况下，p2l page的个数也是确定的，从而可以根据读取次数确定一个block中的p2l page是否读完。

```
    while(rb_smrd_vld)
    {
        if(rw_smrd_bfp & 0x4)
        {
            gb_p2lp_cnt[1]++;
        }
        else
        {
            gb_p2lp_cnt[0]++;
        }
        rb_smrd_vld = SMRD_CLR;
    }
```

- 当NAND将数据返回到buffer时，硬件寄存器 `rb_smrd_vld` 会拉高。
  - valid拉高同时会通过寄存器返回一些相关信息，如表示读取地址的 `rd_smrd_pad` 、用于比对command的 `rd_smrd_cid` 等等（相关定义均在register.h中）。
  - `rb_smrd_vld` 寄存器是写1清零的。当清零后，硬件会立马将下一笔信息放入寄存器，因此读上述相关信息的操作应该是在将 `rb_smrd_vld` 清零前进行。
- `rw_smrd_bfp` 记录了当前command将数据返回到的buffer号。根据buffer号可以确定是读取p2l page时，ping-pong结构中哪一笔所发出的命令，并将对应的计数器加1。当对应的计数器加到4时，说明已经读完了一个p2l page。

```
if((rb_gcad_set == 0x0) && gb_gcad_exe)
{
    fv_uart_debug("clear p2l rd flag: gb_p2lp_fns:%d, gb_p2lp_nrd:%d\r\n", gb_p2lp_fns, gb_p2lp_nrd[gb_p2lp_fns]);

    gb_gcad_exe = 0;
    gb_p2lp_nrd[gb_p2lp_fns] = 0;
    gb_p2lp_fns = !gb_p2lp_fns;
}
```

- 当读取完p2l page后，**硬件**根据记录的logical addr查找l2p table，获得对应的physical addr，通过这个physical addr可以得到对应数据在当前block的位置。另外根据p2l page和在其中的logical addr可以反推出这个logical addr在当前正在做GC的block中的位置，又得到一个physical addr。通过比对这两个physical addr，可以判断出这个位置的数据还是否是有效的。
- 这里检查是否完成比较，向硬件发出启动比较的相关命令在下文。硬件比对的结果会放在一个队列中。满足此处if条件说明硬件已经完成比较。
- 此时存在一个并行结构控制
  - `gb_p2lp_nrd[gb_p2lp_fns] = 0`：完成比较后可以将对应buffer释放出来用于下一次p2l page的读。
  - 另一组buffer中有已经读上来的p2l page，可以启动下一次比较。

```
if(!gb_gcad_wat[gb_p2lp_cps] && !gb_gcad_exe && (gb_p2lp_cnt[gb_p2lp_cps] == 4) && (!gb_p2lp_npg[gb_p2lp_cps] || (((rb_ochk_map | rb_oclr_map) & (1 << gb_ochk_sel)) == 0)))
{
    ...
    rw_gcad_bfp(0) = gb_p2lp_cps ? 0x4 : 0x0;
    rw_gcad_bfp(1) = gb_p2lp_cps ? 0x5 : 0x1;
```

```
    rw_gcad_bfp(2) = gb_p2lp_cps ? 0x6 : 0x2;
    rw_gcad_bfp(3) = gb_p2lp_cps ? 0x7 : 0x3;

    if(rw_gcad_blk == gw_p2lp_blk[gb_p2lp_cps])
    {
        rd_gpcf_str = rd_gpcf_ptr;
        rb_gwln_str = rb_gwln_ptr;
        rb_pgrp_str = rb_pgrp_ptr;
    }
    else
    {
        rd_gpcf_str = 0;
        rb_gwln_str = 0;
        rb_pgrp_str = 0;
        rw_gcad_blk = gw_p2lp_blk[gb_p2lp_cps];
    }
    rb_gcad_set = GCAD_ACT;

    gb_gcad_exe = 1;
    gb_gcad_wat[gb_p2lp_cps] = 1;
    gw_gcad_blk[gb_p2lp_cps] = gw_p2lp_blk[gb_p2lp_cps];
    gb_p2lp_cnt[gb_p2lp_cps] = 0;
    gb_p2lp_cps = !gb_p2lp_cps;
}
```

- `rw_gcad_bfp`：通知硬件开始比较时，相应p2l page存在哪4个buffer中。
- 然后将p2l page的位置相关信息通知给硬件。
  - 若 `rw_gcad_blk` 与表示当前正在GC的block的变量 `gw_p2lp_blk[gb_p2lp_cps]` 相等，说明当前block还没有处理完。
    - `rd_gpcf_str`：wordline内部的偏移量。
    - `rb_gwln_str`：描述当前处理到了第几个wordline。
    - `rb_pgrp_str`：有的NAND中，不是每个wordline中都是3个page，起始的几个有可能是SLC或者MLC，需要通知给硬件。这个寄存器的值代表哪种配置由另一组寄存器来配置。
    - `rd_gpcf_ptr`、`rd_gpcf_ptr`、`rb_pgrp_ptr` 这几个寄存器存储的是硬件自己累加计算的结果，当不存在空洞时，可以直接将硬件计算的结果赋值给 `rd_gpcf_str`、`rb_gwln_str`、`rb_pgrp_str`。
  - 若 `rw_gcad_blk` 与表示当前正在GC的block的变量 `gw_p2lp_blk[gb_p2lp_cps]` 不相等，说明是对新的block进行处理。
    - 将 `rd_gpcf_str`、`rb_gwln_str`、`rb_pgrp_str` 赋0；
    - 更新 `rw_gcad_blk` 值。

- `rb_gcad_set = GCAD_ACT`：触发硬件开始进行对比。

```
while(rb_gcad_vld && (gw_gcrd_cnt < GC_RD_THR))
{
    if(!rb_gcad_nul)
    {...}

    if(rb_gcad_fin)
    {
        ...
        if(gb_pmsg_epg[gw_pmsg_gap])
        {...}

        if(gw_pmsg_gap == PMSG_END)
        {...}
        ...
    }
    ...
}
```

- `rb_gcad_vld`：表示已经比对完，可以去读取结果。
- `if(!rb_gcad_nul)`：表示存在valid page，需要进行读取操作。
- `if(rb_gcad_fin)`：表示当前p2l page中对应的有效数据已经读取完成。
  - `if(gb_pmsg_epg[gw_pmsg_gap])`：检查GC read发出去后，对应的数据有没有被host覆写过。这个是由硬件进行处理的。
- `if(gw_pmsg_gap == PMSG_END)`：整个block中p2l page对应的数据都已经读完。

```
while(rb_gcrd_vld)
{
    ...
    ld_buf_ptr = (dwrd)rw_gcrd_bfp;
    if((ld_buf_ptr >> 9) == SMDT_BASE)
    {
        memcpy((byte *)(ld_cah_ladr), (byte *)(&rq_smmt_dat(ld_buf_ptr, (HMETA_QSZ+0))), (CMPR_NUM*4));
    }
    else
    {
        memcpy((byte *)(ld_cah_ladr), (byte *)(&rq_dmmt_dat(ld_buf_ptr, (HMETA_QSZ+0))), (CMPR_NUM*4));
    }

```

```
    lb_olkp_act = (byte)rd_gcrd_cid(0);
    ...
    if(rb_olkp_hit) {lb_olkp_act = 0x0;}

    if(lb_olkp_act == 0x0)
    {...}
    else
    {
        ...
        gd_vcnt_tbl[lw_cmd_blk]--;
        ...
    }
    ...
}
```

- `rb_gcrd_vld`：当有一个GC读操作完成了，这个信号会拉高。
- `rw_gcrd_bfp`：所读取数据对应的buffer。
- `if((ld_buf_ptr >> 9) == SMDT_BASE)`：将对应buffer的meta data先取出来。
- 然后再检查对应数据有没有被覆写过。若 `rb_olkp_hit` 拉高表示对应数据被覆写过，对应buffer直接释放即可结束。
- 若没有覆写，则需要进行GC write，剩余部分的操作和host write的操作类似，可以参考写操作部分的解释说明。
    - 处理过程中，每次写操作将GC的valid count减1，而不是直接赋值为0，这样当处理完成后可以检查valid count是否为0，为0说明中间操作没有问题。

4.5 读操作

NVMe IO 读命令在硬件中进行解析，查找L2P table，获得logical addr对应的physical addr，并且查找数据是否在buff缓存中。将读命令封装成host_cmd给firmware。Firmware 通过读取寄存器rs_host_cmd来获得host_cmd。Firmware判断physical addr是否是有效地址，如果是无效地址，则把gw_null_ptr的内容传给host,gw_null_ptr存的全0数据。如果physical addr是有效地址，并且数据在buff中有缓存，则直接将缓存的数据发送给host。如果数据在buff中没有缓存，则将读命令发送给flash controller，之后硬件会给这个读命令分配buff id，当flash controller把数据从nand读取到buff之后，硬件会将数据传输给host，当数据传输完成之后，硬件会释放对这个buff的控制。

代码结构：

处理读和写命令的函数是 `sys\ftl_rw.c` 中的 `fv_ftl_hdl()` 函数，以下是对这个函数的解释说明。

```
if(rs_host_cmd.sb_cmd_type == HCMD_RD)
```

- 若提取到的命令类型是read命令,会跳转到 `drv\nand_rw.c` 中的 `fv_nand_read()` 函数进行处理，读命令会根据所访问的地址，将读命令分为3类：
  - 若所访问的地址在之前未被写过

```
if(rs_host_cmd.sd_cmd_padr == L2P_NULL)
{
    while(rd_cawf_set & CAWF_ACT);
    rd_cawf_cid(0) = rs_host_cmd.sd_cmd_cid;
    rd_cawf_set = CAWF_ACT | CAWF_KEP | CAWF_RFN | CAWF_WAT | CAWF_MSZ | ((dwrd)gw_null_ptr << 16);
}
```

    - 首先判断engine现在是否能用，检查上一次buffer传输中的数据是否已经被搬回host；
    - 然后将该NVMe命令对应的cid存于 `rd_cawf_cid(0)` 中；
    - 根据NVMe协议规定，这种情况返回的数据应是全0，对应的数据会存在 `gw_null_ptr` 对应的buffer中，该buffer由CPU提前申请；
    - `CAWF_KEP宏`：表示读取完数据后不会释放寄存器中的数据（因为null_ptr对应的寄存器是固有的）。
  - 若写过且所访问地址对应的数据已存于buffer中

```
else if(rs_host_cmd.sw_cmd_bfp != BPTR_INVLD)
{
    while(rd_cawf_set & CAWF_ACT);
    rd_cawf_cid(rs_host_cmd.sb_cmd_crl) = rs_host_cmd.sd_cmd_cid;
    rd_cawf_set = CAWF_ACT | ((byte)rs_host_cmd.sb_cmd_crd << 1) | CAWF_RLS | CAWF_RFB | CAWF_MSZ | ((dwrd)rs_host_cmd.sw_cmd_bfp << 16);
}
```

    - 首先判断engine现在是否能用，检查上一次buffer传输中的数据是否已经被搬回host；
    - 然后将该NVMe命令对应的cid存于 `rd_cawf_cid(rs_host_cmd.sb_cmd_crl)` 中，其中 `sb_cmd_crl` 对应的值目前只会为0；
    - 从对应的buffer中读取到所需数据；
    - `CAWF_RLS宏`：表示读取完数据后会释放寄存器中的数据。
  - 若所访问地址对应的数据未存于buffer中

```
else
{
    rd_cmd_cid(0) = rs_host_cmd.sd_cmd_cid;
    rd_cmd_padr = rs_host_cmd.sd_cmd_padr;
    fv_nque_chk(1);
    rw_cmd_bptr = BPTR_INVLD;
    rw_cmd_type = READ_CPTR | LOW_PRI;
}
```

- 设置nand read命令所需的cid信息 `rd_cmd_cid(0)` ，当表示读取完成的valid信号返回后，相同的cid会记录在 `rd_smrd_cid` 对应的硬件寄存器中，用于比对所读数据是哪笔命令需要的
- `fv_nque_chk(1)` 用来检查nand control向nand发命令的队列是否已满，但目前若队列满除了打印外，未做其余处理；
- 将 `rw_cmd_bptr` 设置为 `BPTR_INVLD` （0x3F），表示需要硬件自动去分配一个buffer。
  - 若此处改为其他的数，表示fw已经分配好了buffer，将对应buffer编号填进去。
- 处理完后返回 `fv_ftl_hdl()` 函数，向 `rb_host_get` 中写入 `HCMD_INV` ，将下一笔命令传递进来，清除前一笔命令的信息。

4.6 写操作

NVMe IO 写命令在硬件中进行解析，查找L2P table，获得logical addr对应的physical addr，并且给写命令分配一个buff id。将写命令封装成host_cmd给firmware。Firmware 通过读取寄存器rs_host_cmd来获得host_cmd。Firmware从寄存器rd_hp4k_loc中获取一个物理地址，这个物理地址就是硬件算出来的即将分配的物理地址。此时需要检查寄存器rb_head_stg，如果rb_head_stg的值为0，表示从rd_hp4k_loc中获取物理地址对应的block已经写满了，此时需要分配一个新的block去进行写操作，然后firmware给分配一个block给这个逻辑地址，物理地址的ch/page/frag沿用从rd_hp4k_loc中读到的地址，由此得到这个逻辑地址分配的物理地址，对block进行擦除操作。

之后更新L2P table,更新P2L table,更新meta data中的物理地址,把写命令发送给flash controller。当数据传输完成之后，硬件会释放对这个buff的控制。

每当写了4个frag数据时，将这4个P2L entry从CPU cache中下刷到SRAM；每当写1K个frag数据时，将这1K个P2L entry从SRAM中下刷到DRAM；每当写了4K个frag数据时，会将对应数据的P2L table从DRAM下刷到nand中。

代码结构：

处理读和写命令的函数是 `sys\ftl_rw.c` 中的 `fv_ftl_hdl()` 函数，以下是对这个函数的解释说明。

```
else if(rs_host_cmd.sb_cmd_type == HCMD_WR)
```

- 若提取到的命令类型是write命令，则在本函数中做相应处理。

```
rb_head_idx = rs_host_cmd.sb_cmd_strm;
```

- 首先对 `rb_head_idx` 赋值，该变量表示stream id。目前还没有做multi-stream，因此该变量值都是0。可以认为有8个计算physical addr的engine，在写命令和GC中各需要一个，此处为0，GC处的为1**(表述可能不够合理清楚)**。

```
do
{...} while(lb_p2lp_act);
```

- 这一部分处理host写命令， `lb_p2lp_act` 表示需要刷新p2l page
- **下面一部分是对do-while循环部分内的代码的解释**

```
lb_cmd_end = (rb_frag_loc == ((FRAG_QNTY - 1)) && (rb_gwln_loc == rb_gwln_end)) ? 1 : 0;
ld_cmd_padr = rd_hp4k_loc;
```

  - `lb_cmd_end`：由于目前的配置是TLC，由3个page组成一个wordline，这3个page要同时写，如果该信号拉高，则说明可以把前面的数据都写入
  - `rd_hp4k_loc` 寄存器中存储的是下一个可以用的物理地址，将其赋值给 `ld_cmd_padr`
  - **若 `lb_p2lp_act` 值为1，向NAND写的是p2l page：**

```
if(lb_p2lp_act) //p2l page
{
    ld_buf_ptr = (FTL_HTP2L_BASE >> 12) | (dwrd)lb_p2lp_ptr;    //DRAM address
    lb_p2lp_act = (lb_p2lp_ptr == 3) ? 0 : 1;
    lb_p2lp_ptr = (lb_p2lp_ptr == 3) ? 0 : (lb_p2lp_ptr + 1);

    rb_nddn_bit = 0xf; //clear transfer done flag
    while(rb_ddma_act);

    //send program cmd to HW
    fv_nand_htprg(lb_cmd_end, ld_buf_ptr, ld_cmd_padr);
}
```

- p2l的存储策略：SRAM里有4KB的buffer放p2l，当这4KB空间满了以后，会将数据搬至DRAM中。当在DRAM中凑够16KB数据后，再将数据搬至NAND中。
- 实际上SRAM中是有两个4K buffer，使用ping-pong策略。
  - 当一个buffer中更新满了之后，会把这个buffer中的数据搬至DRAM中，但当此时如果有新的host command，还需要去写新的p2l。
  - 因此SRAM中还有另一个buffer，保证前一个buffer在向DRAM搬移数据还没有释放出来时，另一个buffer可以继续写入新的p2l。
  - Buffer的切换通过软件实现。
- `ld_buf_ptr`：表示buffer的地址。`(FTL_HTP2L_BASE >> 12) | (dwrd)lb_p2lp_ptr` 表示DRAM中存储相关内容的地址
- `lb_p2lp_act`：表示是否需要往NAND刷p2l
- `lb_p2lp_ptr`：表示当前DRAM中存储p2l page的个数
- `while(rb_ddma_act)`：检查数据从SRAM搬至DRAM是否完成。

- **若 `lb_p2lp_act` 值为0，处理的是host写命令**

```
ld_buf_ptr = (dwrd)rs_host_cmd.sw_cmd_bfp;
```

- `ld_buf_ptr` 表示buffer的地址，当host command传递过来时，相关寄存器已经由BMU分配好

```
if(rs_host_cmd.sd_cmd_padr != L2P_NULL)
{
    ...
    gd_vcnt_tbl[lw_cmd_blk]--;
    ...

    if((gd_vcnt_tbl[lw_cmd_blk] == 0) && ((gb_blk_type[lw_cmd_blk] & HEAD_BLK) == 0x0))
    {
        ...
        gb_blk_type[lw_cmd_blk] &= ~VCNT_BLK;
        ...
    }
}
```

- 若 `rs_host_cmd.sd_cmd_padr` 不为0，表示对应物理地址之前已经写过，需要进行覆写并将对应的superblock的valid count减1
- `gd_vcnt_tbl[lw_cmd_blk]` 表示一个superblock中的valid fragment count
- 若减1后superblock的valid count为0，则表示该superblock中的数据已经全部被覆写，可以直接释放出来，并不需要等到GC的时候再进行
  - 存储superblock信息的数组 `gb_blk_type` 会将对应superblock的 `VCNT_BLK` 位会清零
  - 存储空闲superblock的列表是采用尾部插入的方式
  - 擦除的操作是在block被分配的时候才进行的

```
if(rs_host_cmd.sb_cmd_smap != BYTE_MASK)
{...}
```

- 没有4KB对齐的if判断目前不会成立，可以删除

```
if(rb_head_stg == 0x0)
{
    ...
    ld_cmd_padr = ld_cmd_padr | ((dwrd)lw_cmd_blk << (PAGE_SHIFT + BKCH_SHIFT + FRAG_SHIFT));
    ...
    fv_nand_erase(lw_cmd_blk);
    gd_ecnt_tbl[lw_cmd_blk]++;
}
```

- 若整个block都写满了，则需要分配新的block
- 其中 `ld_cmd_padr = ld_cmd_padr | ((dwrd)lw_cmd_blk << (PAGE_SHIFT + BKCH_SHIFT + FRAG_SHIFT));` 使用或操作的原因是满足这个if条件时，ld_cmd_padr的值是0，因此会被赋值为一个block开始的位置
- `fv_nand_erase(lw_cmd_blk);`：在使用block前将其擦除
- `gd_ecnt_tbl[lw_cmd_blk]` 表示这个block被擦除过的次数

```
gd_l2p_tbl[rs_host_cmd.sd_cmd_ladr] = ld_cmd_padr;
gd_vcnt_tbl[rw_hblk_loc]++;
```

- 然后会更新l2p table `gd_l2p_tbl` 以及要写入block的valid count `gd_vcnt_tbl`

```
rd_lupd_l4k = rs_host_cmd.sd_cmd_ladr;
rd_lupd_p4k = ld_cmd_padr;
rw_lupd_set = LUPD_ACT;
```

- 由于需要写的数据还在BMU中，需要告诉BMU这些数据的相关信息
- `rd_lupd_l4k` 和 `rd_lupd_p4k` 分别对应对应数据的逻辑地址和物理地址
- `rw_lupd_set` 表示需要更新l2p table，因为硬件内部还有一个存储l2p的cache（fw不可见），防止再次读的时候出错

```
rd_ochk_pat = rs_host_cmd.sd_cmd_padr;
```

- GC需要先将数据从NAND上读出再写将数据写入新的block。
- 当数据从NAND上读出但还没写回时，host发出了一个覆写的命令，此时host写的数据是更新的，但是GC并不知道此时不需要再将旧数据写回
- 因为存在时间差，通过此处的代码通知GC不需要再将旧数据写回

```
gd_htp2l_ladr[gw_htp2l_ofs & 0x3] = rs_host_cmd.sd_cmd_ladr;
```

- `gd_htp2l_ladr[4]` 相当于CPU中存储p2l的一级cache，此处先将 `sd_cmd_ladr` 存于这个数组中

```
if((ld_buf_ptr >> 9) == SMDT_BASE)
{
    rq_smmt_dat(ld_buf_ptr, (HMETA_QSZ+0)) = ((qwrd)ld_cmd_padr << DWRD_SHIFT) | (qwrd)rs_host_cmd.sd_cmd_ladr;
    memset((byte *)(&rq_smmt_dat(ld_buf_ptr, (HMETA_QSZ+1))), 0, (4*6));
}
else
{
    rq_dmmt_dat(ld_buf_ptr, (HMETA_QSZ+0)) = ((qwrd)ld_cmd_padr << DWRD_SHIFT) | (qwrd)rs_host_cmd.sd_cmd_ladr;
    memset((byte *)(&rq_dmmt_dat(ld_buf_ptr, (HMETA_QSZ+1))), 0, (4*6));
}
```

- 每个4k有一个meta data区域，写入fragment对应的OOB区域中

    - fw最多可写32byte，其中logical addr和physical addr均是4byte，其余24byte补0.
    - 这里是灵活控制的，也可以是16byte或8byte
    - 若32byte用完，会占用LDPC纠错的大小
- 由于0-2559为DRAM buffer的编号，2560-3071为SRAM buffer的buffer，通过这里可以判断该buffer是属于DRAM还是SRAM。此处若满足条件则为SRAM buffer，否则是DRAM buffer。
- `rq_smmt_dat()/rq_dmmt_dat()` 函数将physical addr和logical addr写入，共8byte
- `memset()` 函数将剩余24byte补0

```
if((gw_htp2l_ofs & 0x3) == 0x3)
{
    memcpy((byte *)(&rd_smdt_dat(gw_htp2l_bfp, (gw_htp2l_ofs & 0x3fc))), (byte *)(gd_htp2l_ladr), (4*4));
}
```

- 由于SRAM存在ECC纠错机制，纠错最小单位为16byte，因此往SRAM中传送数据要以16byte为单位
- 假如只写了少于16个byte的数据，是无法读出的
- 当凑够16byte后，在这里将p2l信息从CPU cache中，即前面提到的 `gd_htp2l_ladr[4]` ，搬到SRAM中

```
if((gw_htp2l_ofs & 0x3ff) == 0x3ff)
{
    while((rb_nddn_bit & 0xf) != 0xf);
    while(rb_ddma_act);
    rd_ddma_dad = ((FTL_HTP2L_BASE | ((gw_htp2l_ofs << 2) & 0x3000)) >> 4);
    rd_ddma_sad = ((dwrd)gw_htp2l_bfp << (13 - 4));
    rb_ddma_wen = 1;
    rb_ddma_act = 1;
    gw_htp2l_bfp = (gw_htp2l_bfp == 8) ? 9 : 8;

    //move p2l page from dram to nand
    if(gw_htp2l_ofs == 0xfff)
    {
        gw_htp2l_ofs = 0xffff;
        lb_p2lp_act = 1;
    }
}
```

- 0x3ff即1024，此处说明当凑够1024个p2l后，即4k数据后，需要将SRAM中的数据搬至DRAM
- `while((rb_nddn_bit & 0xf) != 0xf);` 检查最后一个p2l是否传递完毕
- `while(rb_ddma_act);` 检查上一次DMA传输是否完成
- `rb_ddma_wen` 为1说明数据是从SRAM搬至DRAM的，为0则说明数据从DRAM搬至SRAM
- `gw_htp2l_bfp = (gw_htp2l_bfp == 8) ? 9 : 8` 是处理ping-pong结构的，当上一个SRAM buffer填满后，我们使用另一个SRAM buffer记录p2l
- `if(gw_htp2l_ofs == 0xfff)`：若凑够4096个p2l，则需要将数据从DRAM搬至NAND

```
rb_hp4k_set = JUMP_ACT;
```

- 设置```rb_hp4k_set``寄存器，触发硬件计算下一个physical addr

```
if(rb_head_stg == 0x0)
{...}
```

  - 此处操作为冗余操作，与之前代码处理整个block写满时需要分配新的block的逻辑相差不大

```
if((rw_page_loc == (PAGE_QNTY - 1)) && (rb_cech_loc == (BKCH_QNTY - 1)) && (rb_frag_loc == (FRAG_QNTY - 4)))
{
    while(gw_htp2l_ofs < 0x1000)
    {
        rd_smdt_dat(gw_htp2l_bfp, (gw_htp2l_ofs & 0x3ff)) = P2L_NULL;
        ...
    }
    ...
 }
```

  - 每4K个4KB的数据需要有一个p2l page，每一个superblock的最后4个frag一定是p2l page
  - 当到达一个superblock的最后4个frag时，需要强制写p2l page
  - 整个写最后可能填不满p2l page，需要将p2l page刷满

- **以下为do-while循环部分外的代码的解释**

```
if(rs_host_cmd.sb_cmd_mued && (rb_frag_loc == 0x0))
{
    break;
}
```

- 当一直有host写命令传递进来时，不能一直进行写操作，达到条件时需要退出做GC