

XQTyer 评估板例程使用手册

版本记录

版本	版本说明	作者	日期
V1.0	初始版本	Martin	2022-11-1

目录

XQTyer 评估板例程使用手册	1
1 目的	4
2 软件版本说明	4
3 ZYNQ 与 DSP 之间通信例程	5
3.1 ZYNQ 与 DSP 之间 SRIO 通信	5
3.1.1 例程位置	5
3.1.2 功能简介	5
3.1.3 例程使用	5
3.2 ZYNQ 与 DSP 之间 EMIF16 通信	20
3.2.1 例程位置	20
3.2.2 功能简介	20
3.2.3 例程使用	21
3.3 ZYNQ 与 DSP 之间 uPP 通信	27
3.3.1 例程位置	27
3.3.2 功能简介	27
3.3.3 例程使用	28
3.4 ZYNQ 与 DSP 之间 GPIO 通信	32
3.4.1 例程位置	32
3.4.2 功能简介	32
3.4.3 例程使用	33
4 DSP 单独例程	38
4.1 DSP 以太网通信	38
4.1.1 例程位置	38
4.1.2 功能简介	38
4.1.3 例程使用	39
4.2 DSP UART0 串口通信	45
4.2.1 例程位置	45
4.2.2 功能简介	45
4.2.3 例程使用	45
4.3 DSP GPIO LED 示例	50
4.3.1 例程位置	50
4.3.2 功能简介	50
4.3.3 例程使用	51
4.4 DSP DDR3 内存读写测试	54

4.4.1	例程位置	54
4.4.2	功能简介	54
4.4.3	例程使用	55
4.5	DSP DDR3 初始化以及内存读写测试	59
4.5.1	例程位置	59
4.5.2	功能简介	59
4.5.3	例程使用	61
4.6	DSP IPC 核间通信之 NOTIFY	65
4.6.1	例程位置	65
4.6.2	功能简介	65
4.6.3	例程使用	68
4.7	DSP IPC 核间通信之 MessageQ	74
4.7.1	例程位置	74
4.7.2	功能简介	74
4.7.3	例程使用	77
5	ZYNQ PL 单独例程	82
5.1	ZYNQ PL Cameralink 回环例程	82
5.1.1	例程位置	82
5.1.2	功能简介	82
5.1.3	Cameralink 接口时序说明	83
5.1.4	管脚约束	85
5.1.5	例程使用	85
5.2	ZYNQ PL SFP 光口通信例程	89
5.2.1	例程位置	89
5.2.2	功能简介	89
5.2.3	管脚约束	91
5.2.4	例程使用	92
5.3	ZYNQ PL SFP 光口 IBERT 链路误码测试	97
5.3.1	例程位置	97
5.3.2	功能简介	97
5.3.3	例程使用	98
5.4	ZYNQ PL M.2 接口验证例程	106
5.4.1	例程位置	106
5.4.2	功能简介	106
5.4.3	管脚约束	108
5.4.4	例程使用	109
5.5	ZYNQ PL M.2 接口之 NVMe Host IP 例程说明	113
5.5.1	例程声明	113
5.5.2	设计目的	113
5.5.3	NVMe Host FPGA IP 核简介	114
5.5.4	资源消耗	115
5.5.5	NVMe Host FPGA IP 测试截图和说明	116
5.6	ZYNQ PL FMC 之 6 路 ADC 采集例程	123
5.6.1	例程位置	123



5.6.2	功能简介	123
5.6.3	管脚约束	124
5.6.4	例程使用	126
6	ZYNQ PS 单独例程	130
6.1	ZYNQ PS GPIO LED 驱动实验	130
6.1.1	例程位置	130
6.1.2	功能简介	130
6.1.3	例程使用	130
6.2	ZYNQ PS 串口中断实验	136
6.2.1	例程位置	136
6.2.2	功能简介	137
6.2.3	例程使用	137
6.3	ZYNQ PS DDR 内存读写实验	146
6.3.1	例程位置	146
6.3.2	功能简介	146
6.3.3	例程使用	146

广州星嵌电子科技有限公司

1 目的

本手册主要介绍说明 xQTyer 评估板实验平台各个例程的功能、使用步骤以及各个例程的运行效果。同时，也对例程所使用的软件版本、软件工具存放路径以及软件安装参考手册存放位置进行了必要说明。

2 软件版本说明

开发对象	基于软件版本	软件安装工具存放路径	软件安装参考手册
ZYNQ PL	Vivado 2018.3	Tools\ZYNQ\VIVADO2018.3 (注: 提供的是 Vivado 网络安装器, 需要联网安装)	文档 \ZYNQ 开发工具 安装说明.docx
ZYNQ PS	Xilinx SDK 2018.3	说明: 安装 Vivado 工具时, 附带一起进行安装	说明.docx
DSP	CCS 7.4	Tools\DSP\CCS7.4.0.00015_win32.zip	文档 \CCS7.4 软件安装说明.docx
	编译器 TI C6000 CGT 8.3.12	Tools\DSP\CCS 组件安装包 \ti_cgt_c6000_8.3.12_windows-x64_installer.exe	文档 \CCS 编译器和 依赖组件 安装说明.docx
	bios_mcsdk_02_01_02_06	Tools\DSP\CCS 组件安装包 \BIOSMCSDK-C66X	

3 ZYNQ 与 DSP 之间通信例程

3.1 ZYNQ 与 DSP 之间 SRIO 通信

3.1.1 例程位置

ZYNQ 例程保存在资料盘中的 Demo\ZYNQ\PL\srio_gen2_0_ex 文件夹下。

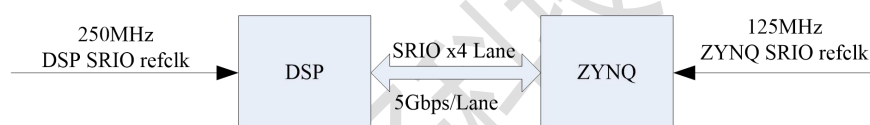
DSP 例程保存在资料盘中的 Demo\DSP\XQ_SRIO_x4LANE_5Gbps 文件夹下。

3.1.2 功能简介

实现 DSP 与 ZYNQ 之间 SRIO 接口传输功能。

DSP 与 ZYNQ 之间 SRIO 通道宽度为 4，每个 SRIO 通道速率 5Gbps。DSP SRIO 参考时钟频率为 250MHz，ZYNQ SRIO 参考时钟频率为 125MHz。

DSP 与 ZYNQ 之间 SRIO 接口相关信号连接示意图如下图所示：



DSP 作为 Initiator 发起 NWrite 数据写事务，将数据写入 ZYNQ PL 端的 RAM 空间（最大 2KB）；接着，DSP 发起 NRead 数据读事务，从 ZYNQ PL 端的 RAM 空间读取数据；DSP 完成数据写、读事务后，对读写数据进行比对，以检测 SRIO 传输是否有数据错误。

3.1.3 例程使用

特别提示：例程使用，请参考下面章节顺序执行。确保 ZYNQ PL 程序要先运行，然后才能运行 DSP 程序。

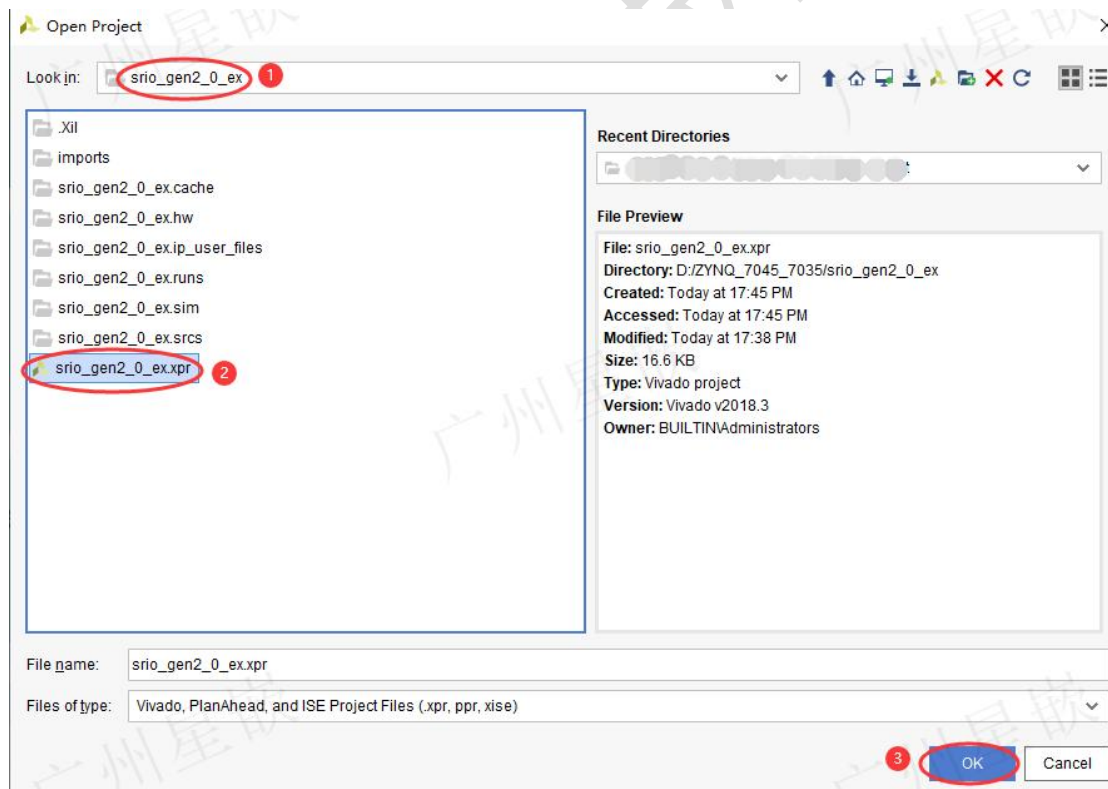
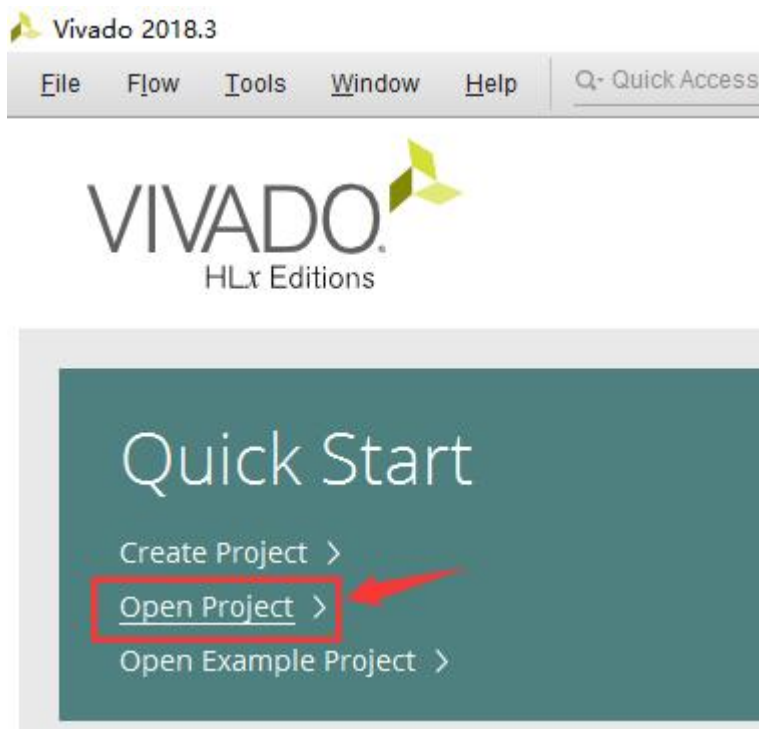
3.1.3.1 加载运行 ZYNQ 程序

3.1.3.1.1 打开 Vivado 工程

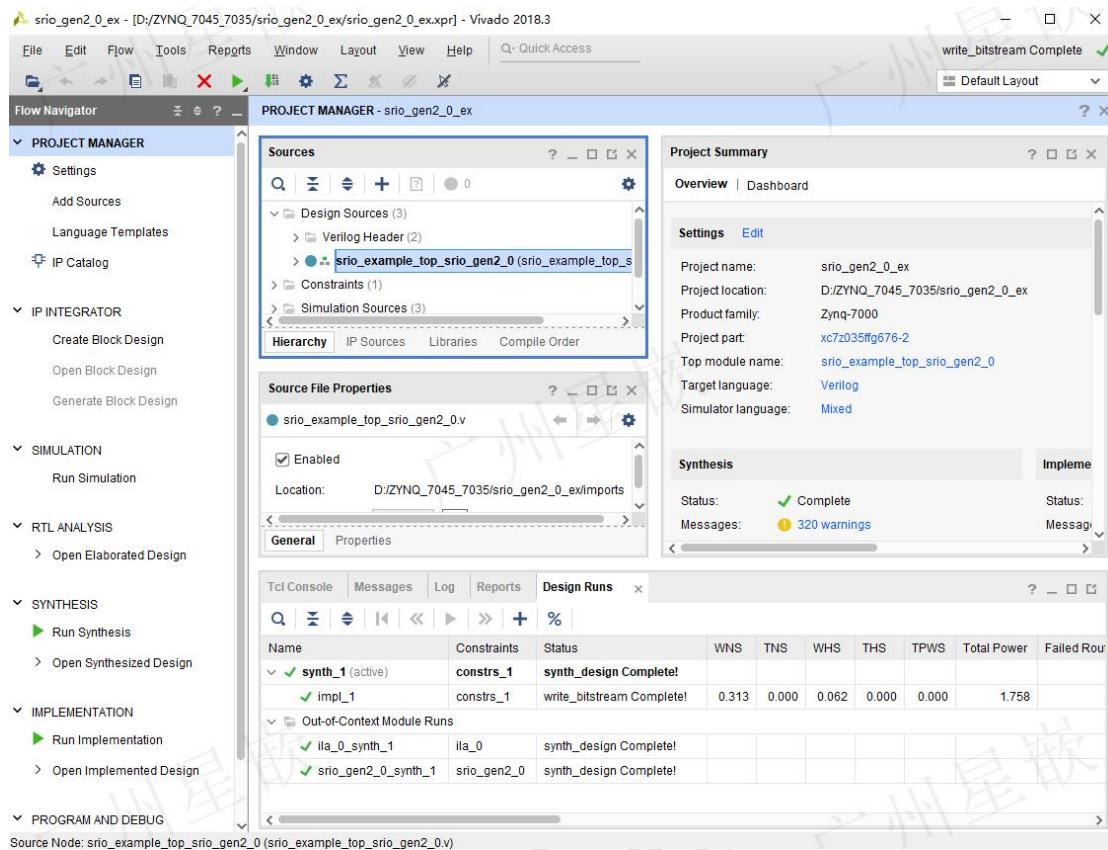
双击桌面 Vivado 图标，打开 Vivado 工具：



点击 Open Project，打开工程（**注意：确保例程路径为非中文路径**）：

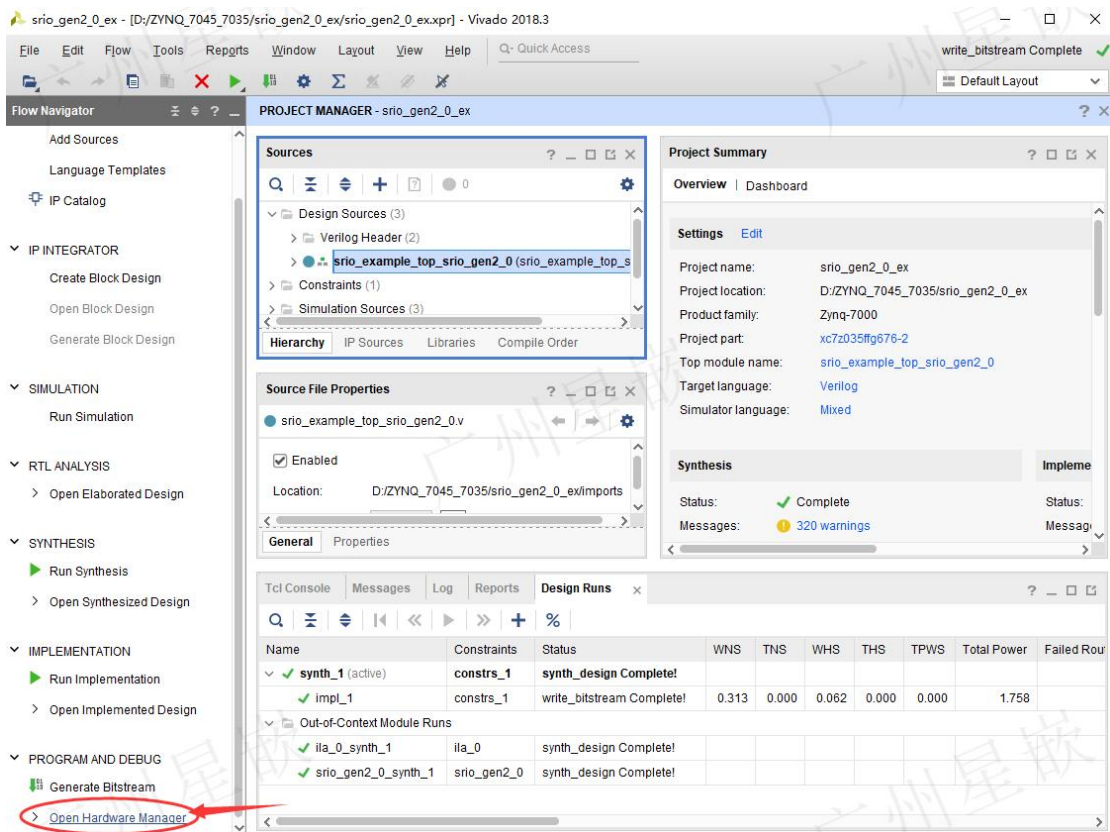


Vivado 工程打开后界面如下图所示：

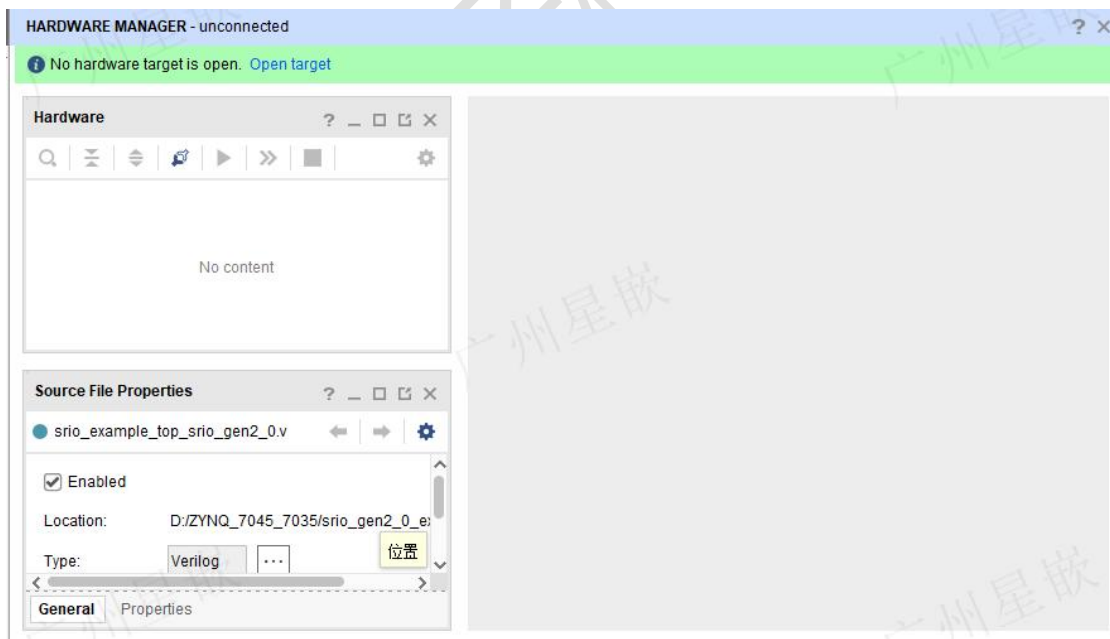


3.1.3.1.2 下载 ZYNQ PL 程序

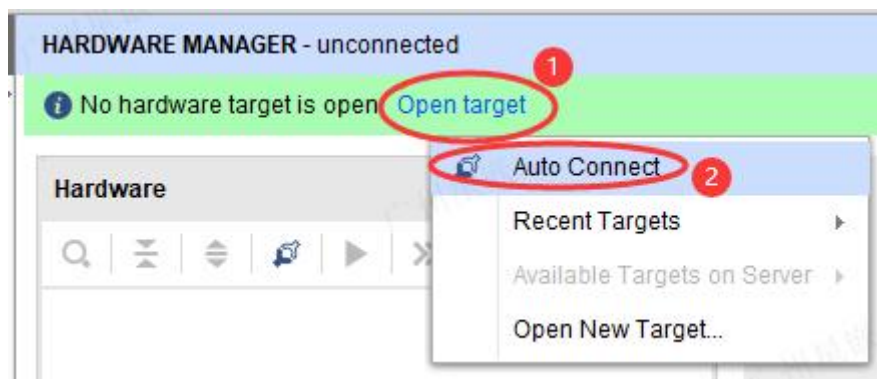
点击 Open Hardware Manager:



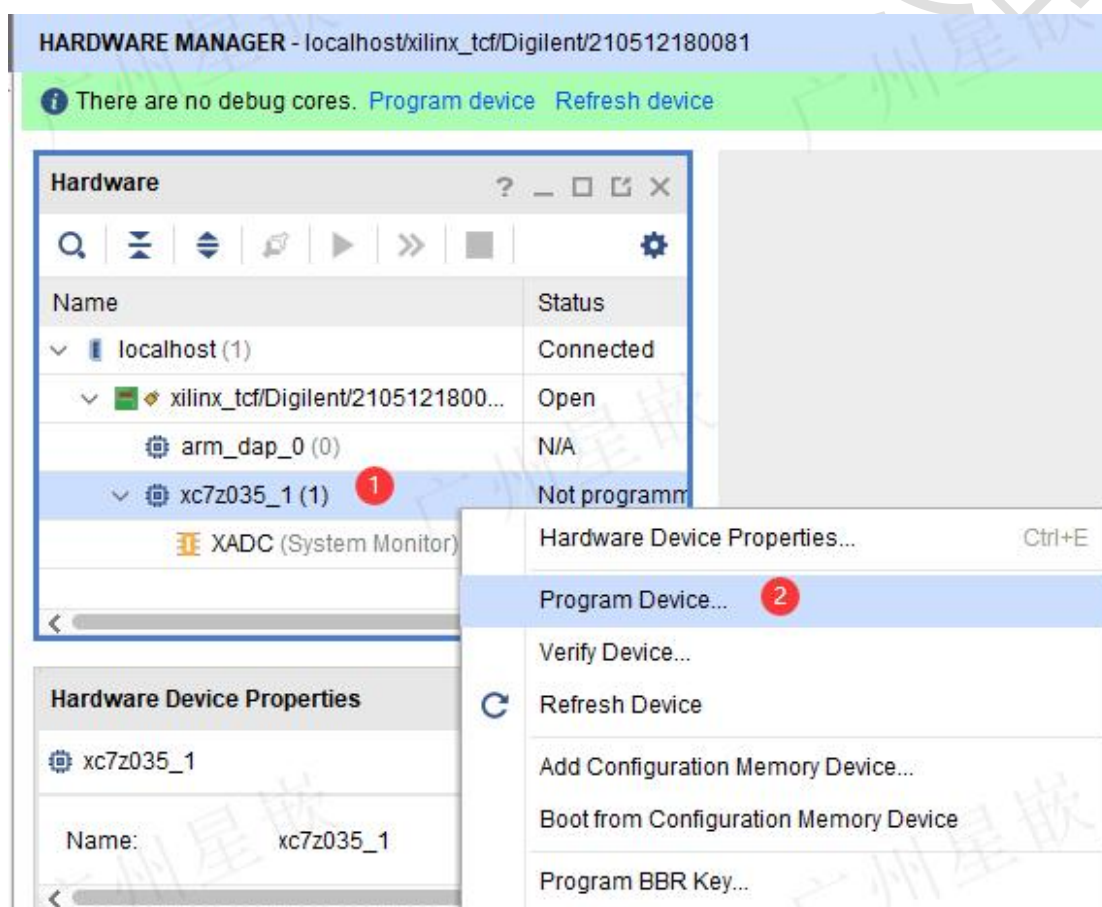
打开 Hardware Manager 的界面如下图所示:



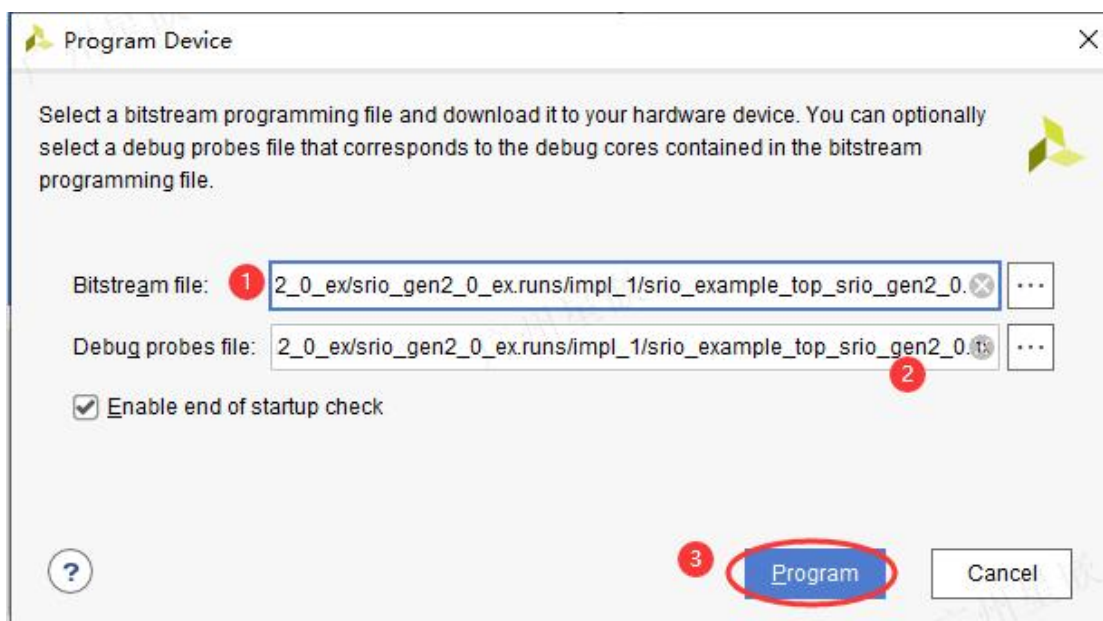
此后，确保 FPGA JTAG 仿真器已连至接板卡和电脑，并且板卡处于上电状态。
点击 Hardware Manager 界面上的 Open target，并在弹出的菜单中单击 Auto Connect:



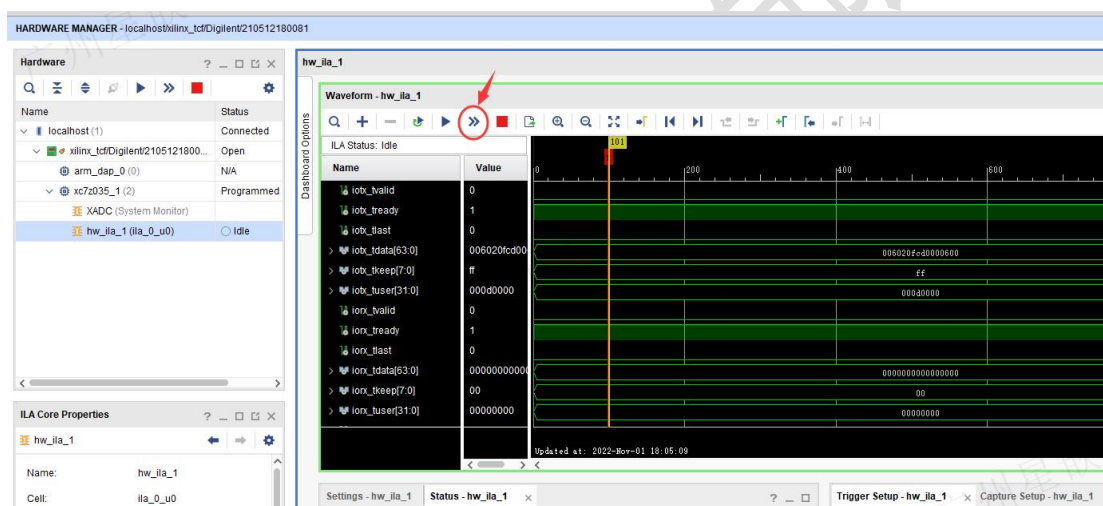
仿真器连接成功后，在找到的 xc7z035_1 器件上右击，并在弹出的菜单中点击 Program Device...:



一般来说，Vivado 下载工具会自动找到本工程下的程序 bit 流下载文件和调试文件，如果没有自动找到，则需要用户通过旁边的浏览按钮去自行选取。确保程序下载文件没问题后，点击 Program 下载程序：



程序下载完成后，点击界面中的“>>”按钮，可实时抓取查看 ZYNQ PL 端信号运行波形：



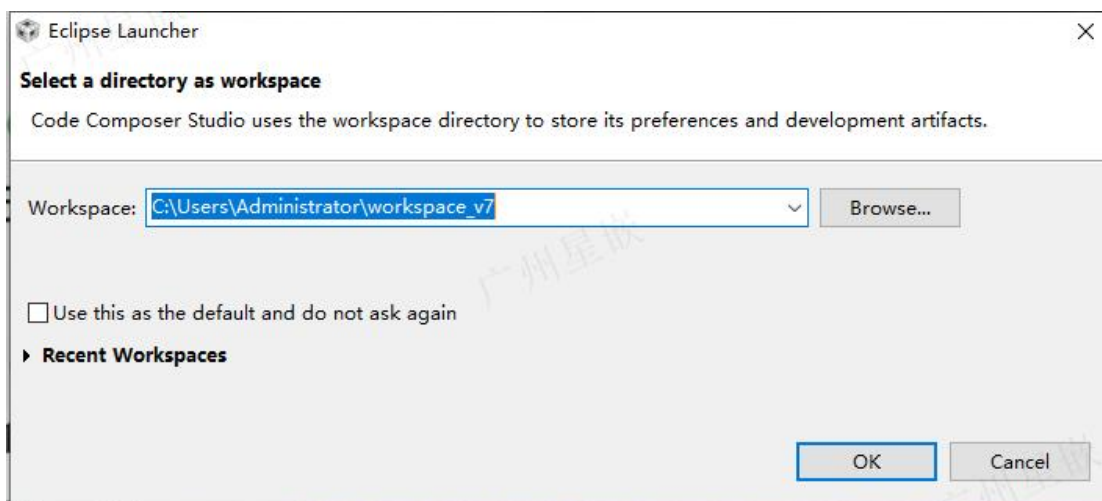
3.1.3.2 加载运行 DSP 程序

3.1.3.2.1 CCS 导入例程

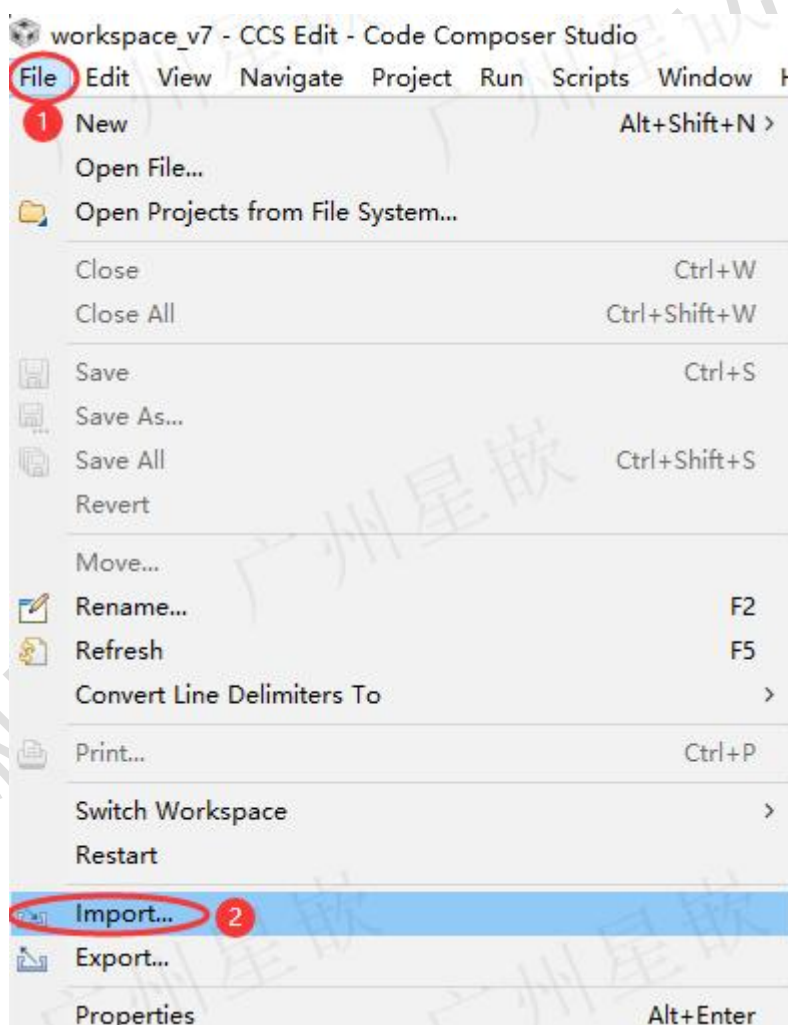
双击桌面 CCS 快捷图标，打开 CCS 软件：



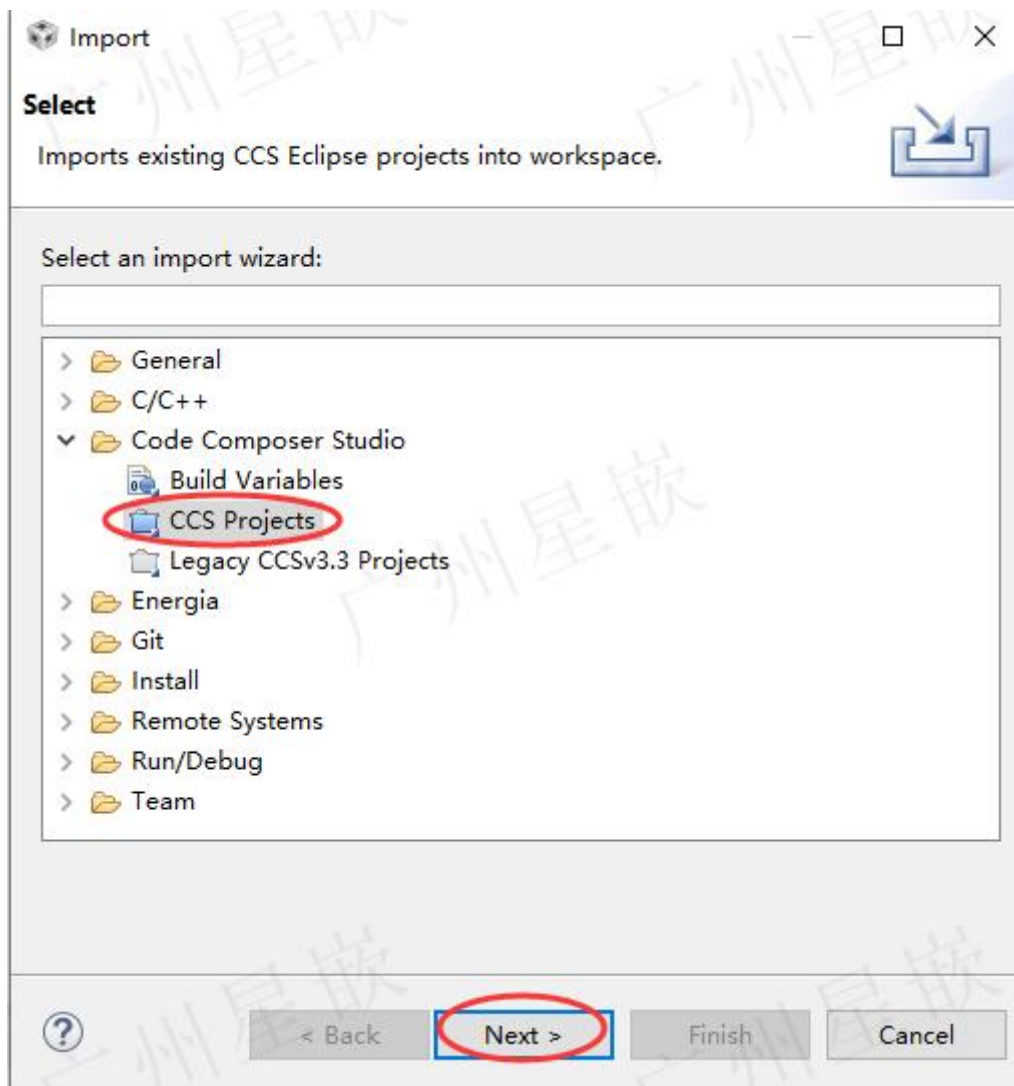
CCS 设置工作空间时，选择默认即可：



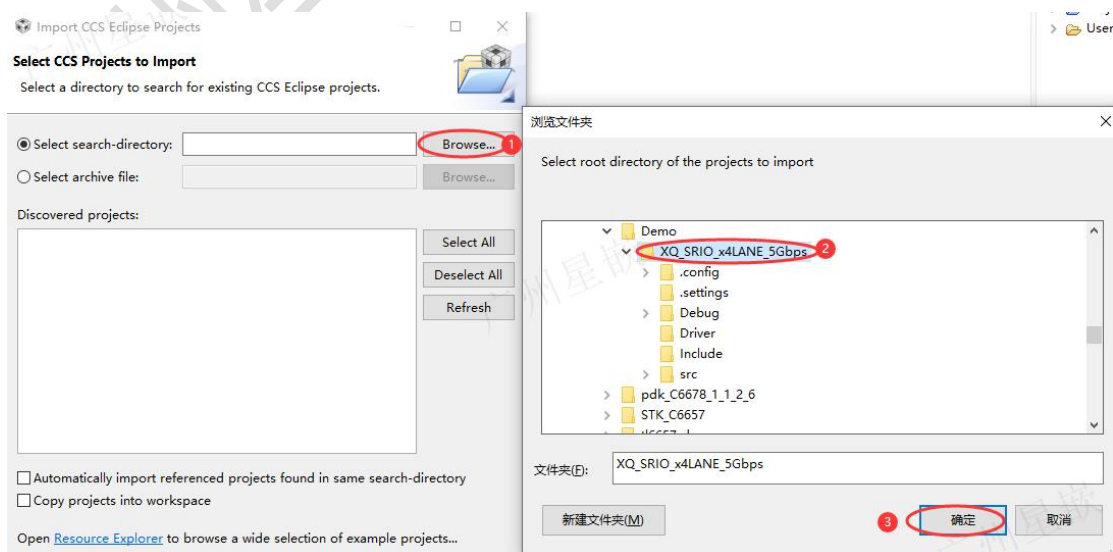
通过菜单 File->Import...导入 CCS 工程:



导入项目选择 CCS Projects:

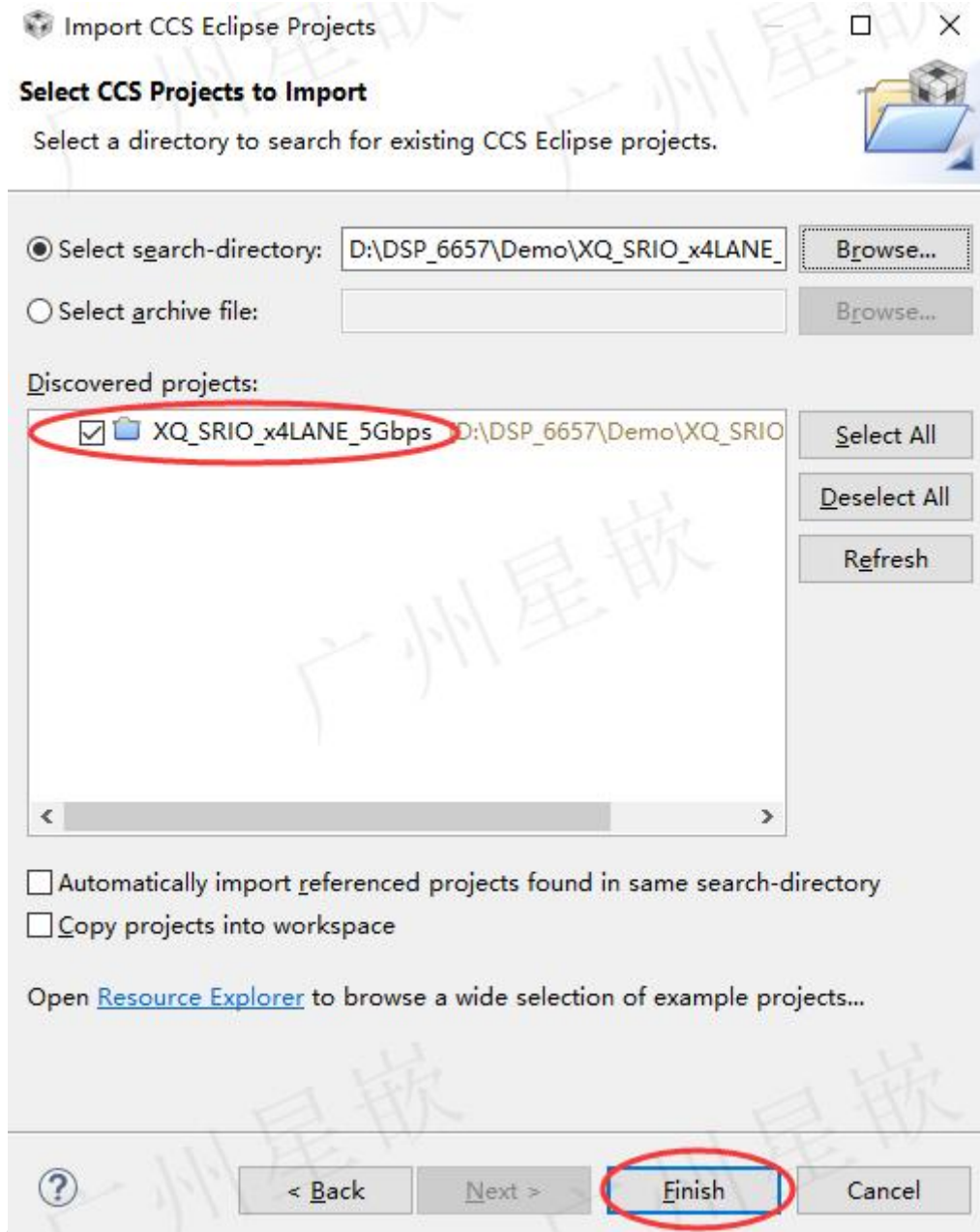


点击 **Browse...**，浏览找到例程所在位置（**注意：确保例程路径为非中文路径**），选中例程所在目录，并点击“确定”：

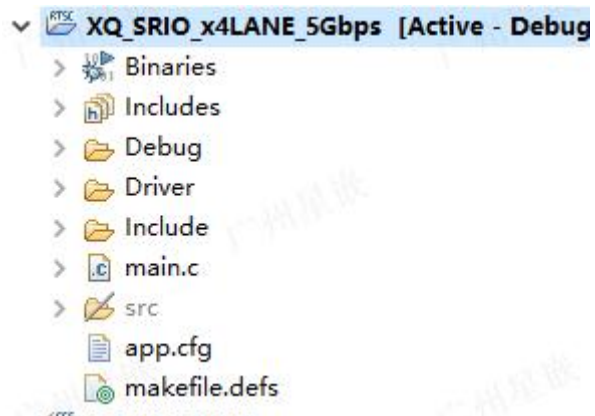




CCS 软件将识别到的例程显示在 Discovered projects 一栏，最后点击 Finish:



例程导入后界面如下图所示:

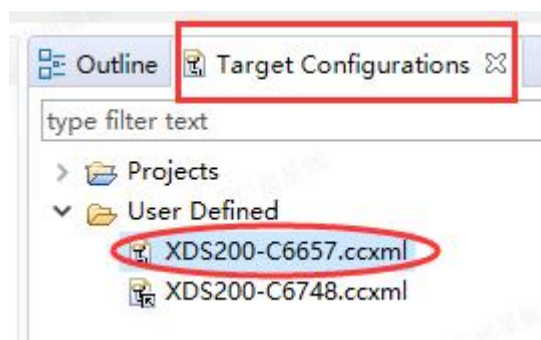


3.1.3.2.2 下载 CCS 程序

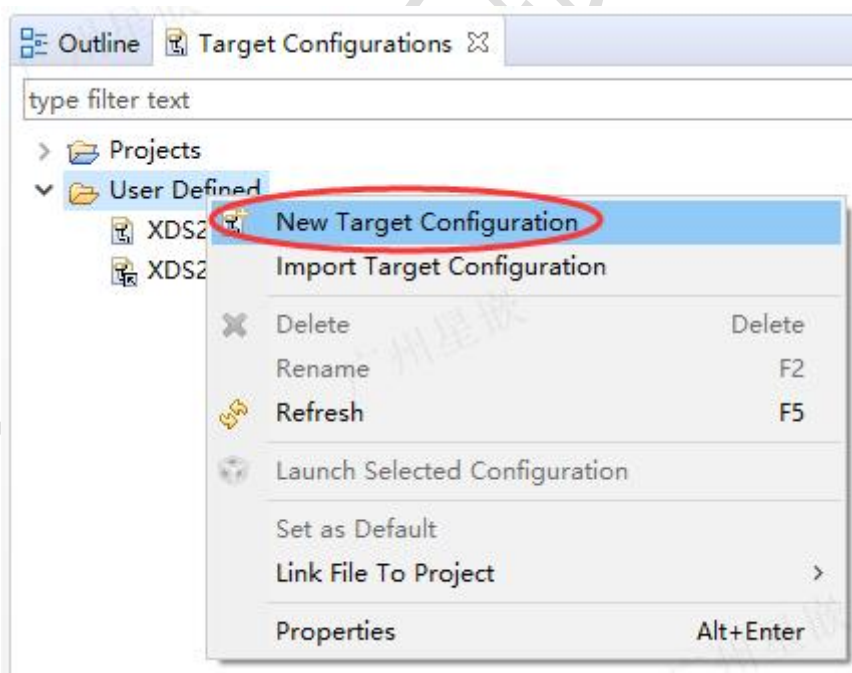
3.1.3.2.2.1 目标配置文件设置

特别提示：目标配置文件设置这一步骤可以只进行一次，后面例程可以反复使用，不用重复创建或设置。

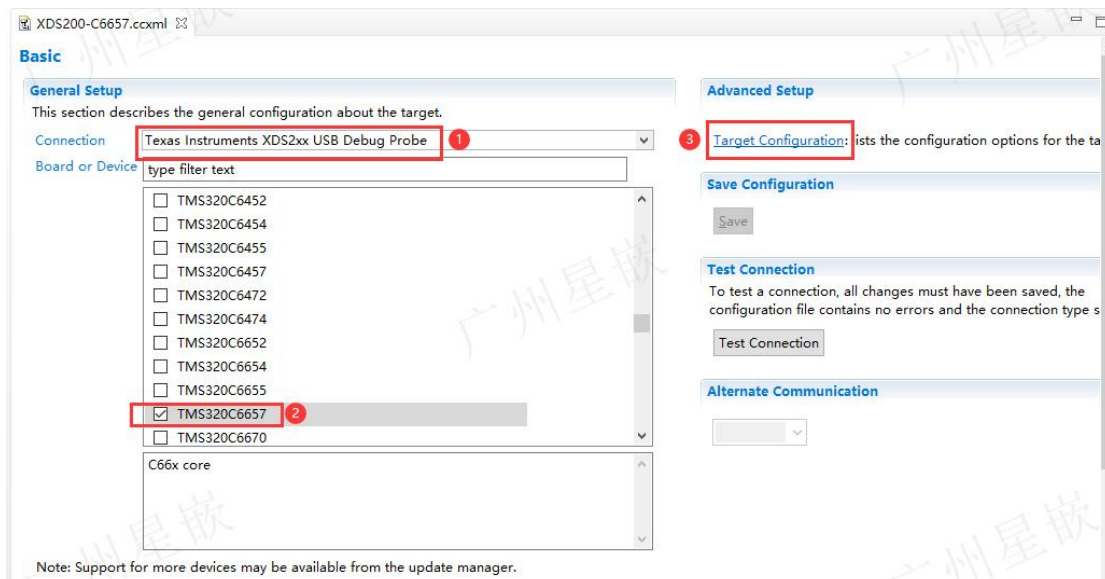
在右边的 Target Configuration 窗口，双击打开之前创建好的目标配置文件，如下图的 XDS200-C6657.ccxml：



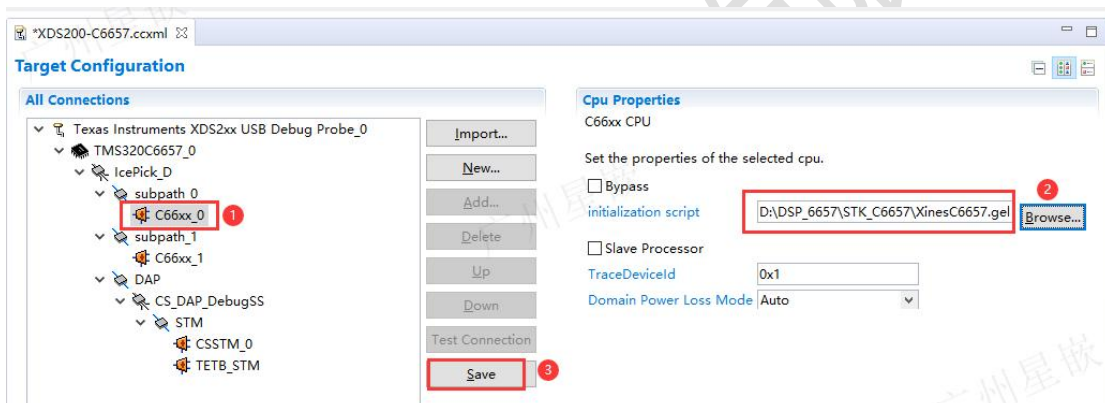
如果还没有目标配置文件，则在 Target Configuration 窗口的空白处或 User Defined 文件夹处右击，并在弹出的菜单中点击“New Target Configuration”新建目标配置文件：



在打开的目标配置文件中，需要配置仿真器类型、器件型号，我们实验用的仿真器为 XQ-XDS200U，选中仿真器类型 XDS2xx USB Debug Probe 即可，器件型号勾选上 TMS320C6657，如下图所示，然后点击高级设置项 Target Configuration，准备 Gel 文件设置：

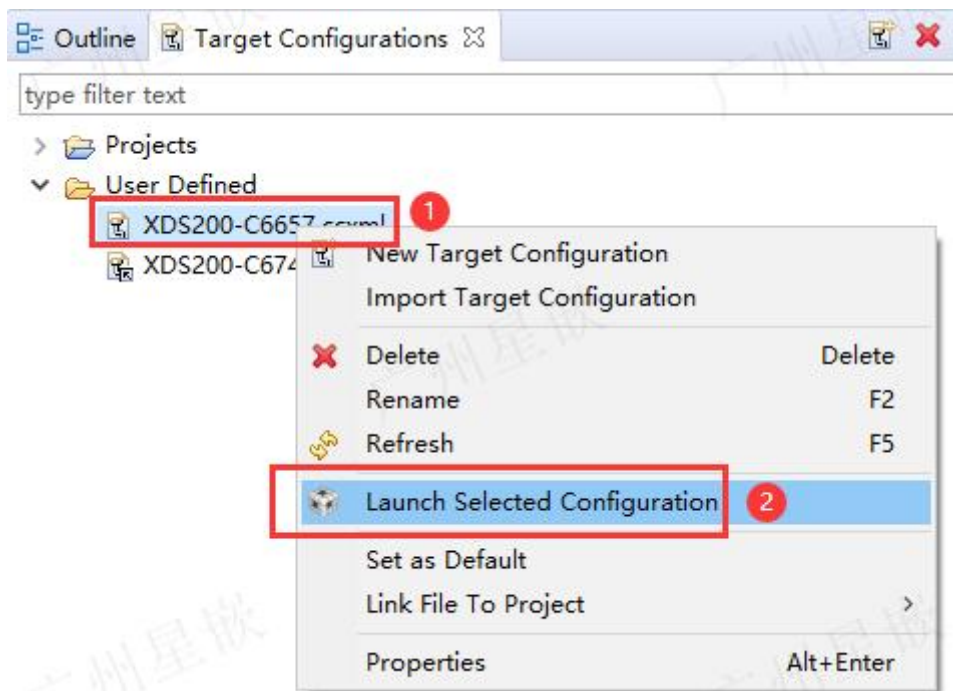


在高级项设置窗口中，点击 C66xx_0 核心，然后在右侧的初始化脚本栏中，点击 Browse，找到我们提供的 Gel 文件，即 XinesC6657.gel。设置完 Gel 文件后，点击 Save：



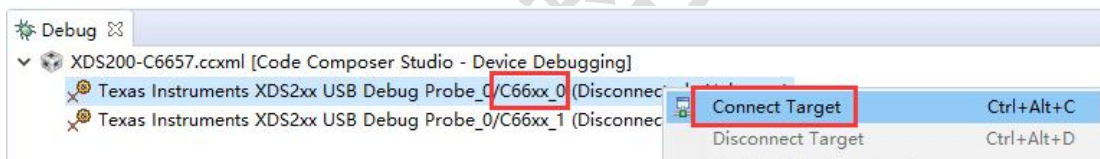
3.1.3.2.2.2 启动目标配置文件

在已经创建并设置好的目标配置文件处右击，并在弹出的菜单中点击 Launch Selected Configuration，打开调试窗口：



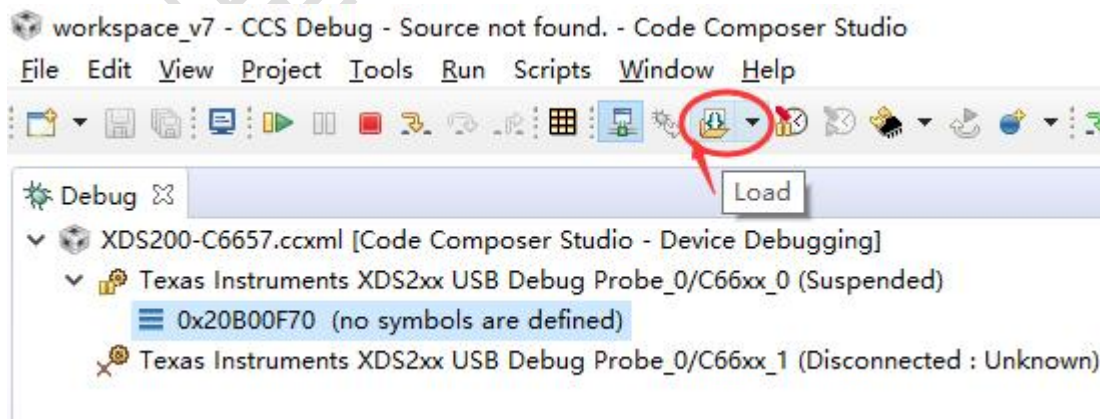
3.1.3.2.2.3 仿真器连接目标器件

调试窗口打开后，右键单击 C66xx_0 核心 0，并在弹出的菜单中点击 Connect Target:



3.1.3.2.2.4 加载 DSP 程序

点击 Load 图标，加载 DSP 程序:

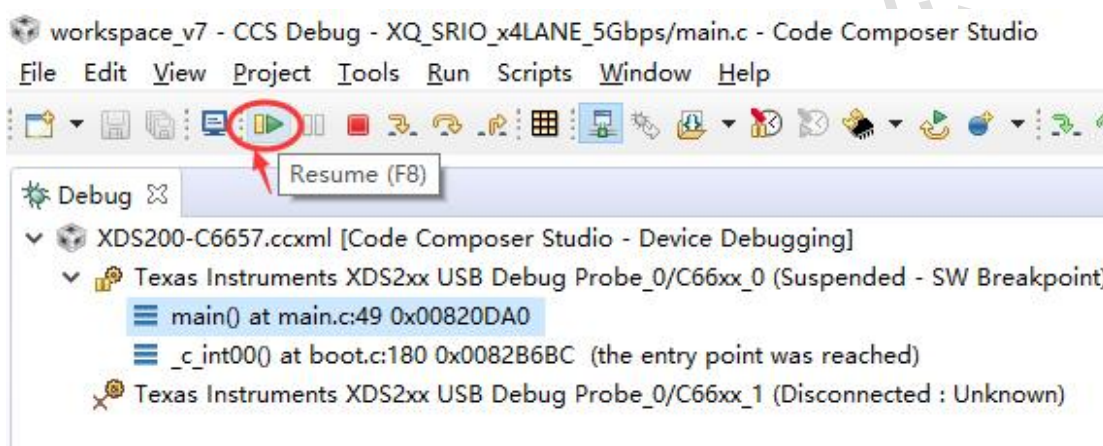


点击 Browse...或 Browse project..., 找到 DSP 程序的可执行文件（以.out 为后缀），然后点击 OK，如下图所示:



3.1.3.2.5 DSP 程序运行

点击 Resume 运行图标，运行 DSP 程序，如下图所示：



3.1.3.3 运行结果说明

3.1.3.3.1 DSP 程序运行结果

CCS 软件的 Console 控制台窗口打印 SRIO 调试信息。

DSP 通过 SRIO 接口先发起 NWrite 写事务，数据长度为 2048 字节；接着 DSP 发起 NRead 事务，数据长度为 2048 字节，然后比对读写事务对应的数据。

如果 SRIO 传输异常，存在数据错误，则程序里面错误计数器累加，并输出打印当前错误个数。每当完成 100 次 NWrite 和 NRead SRIO 读写事务，则输出打印一次“DSP <-> FPGA 204800 bytes OK!”字样，如下图所示：

```

Console X
XDS200-C6657.ccxml:CIO
[C66xx_0] Check SRIO Link...OK.
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
DSP <-> FPGA 204800 bytes OK!
    
```

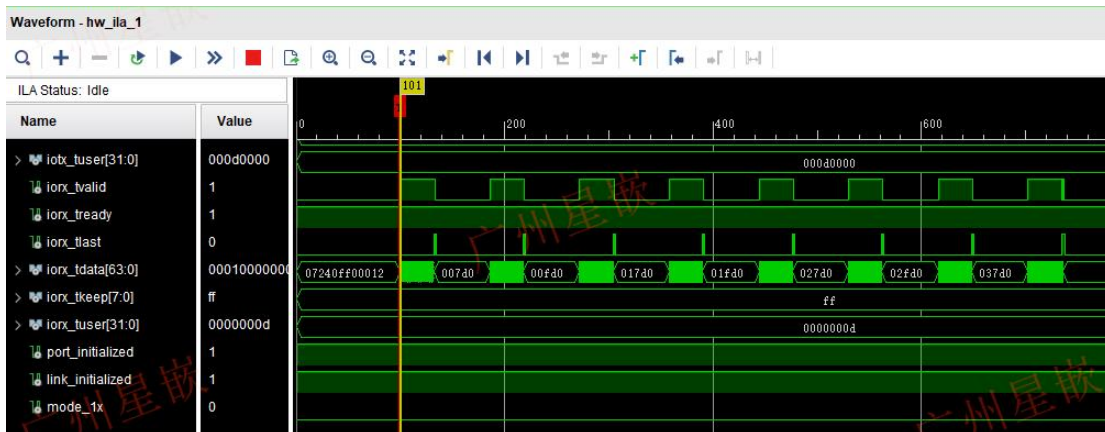
3.1.3.3.2 ZYNQ PL 程序运行结果

ZYNQ PL 端提供的 ILA 调试窗口，可以实时抓取采集 SRIO 本地总线信号时序波形。SRIO 本地总线信号说明如下（详细定义请参考数据手册 Xilinx 文档 pg007_srio_gen2.pdf）：

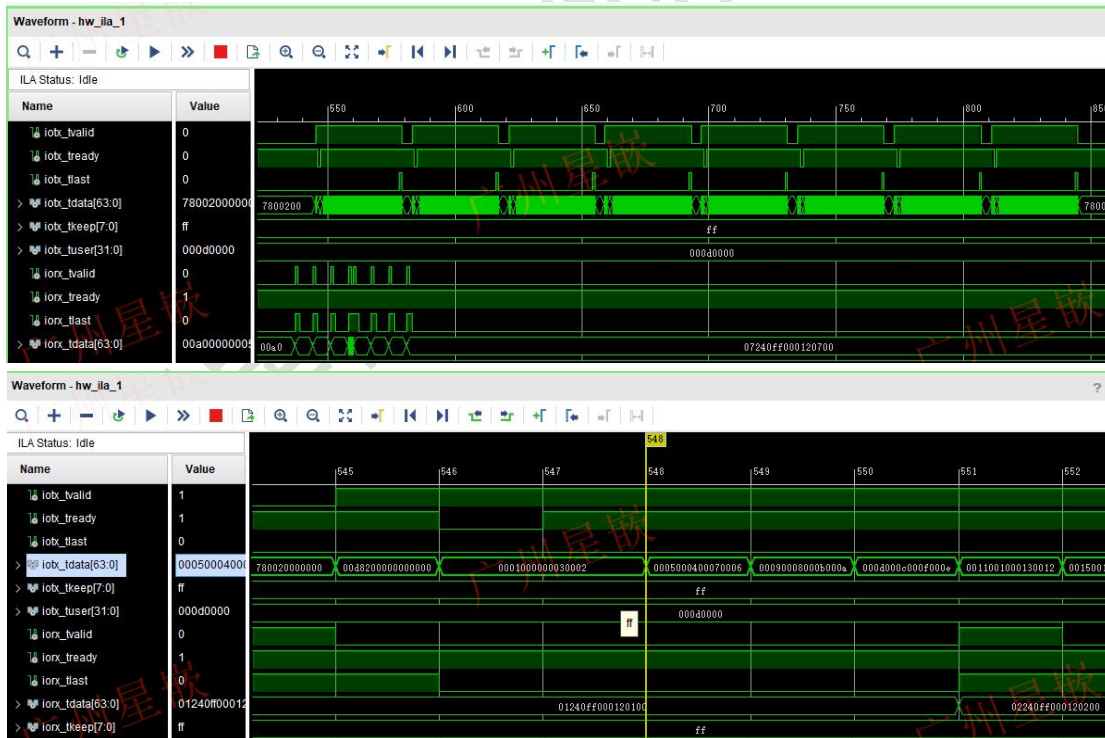
SRIO 本地发送总线信号	
iotx_tvalid	发送数据有效标志位
iotx_tready	发送准备好标志位
iotx_tlast	最后一个发送数据标志位
iotx_tdata	发送数据
iotx_tkeep	发送数据字节控制位
iotx_tuser	发送控制数据，主要内容是源 ID 和目的 ID
SRIO 本地接收总线信号	
iorx_tvalid	接收数据有效标志位
iorx_tready	接收准备好标志位
iorx_tlast	最后一个接收数据标志位
iorx_tdata	接收数据
iorx_tkeep	接收数据字节控制位
iorx_tuser	接收控制数据，主要内容是源 ID 和目的 ID
状态信号	
port_initialized	SRIO 端口初始化完成标志位 1: SRIO 端口初始化完成; 0: SRIO 端口初始化未完成。
link_initialized	SRIO 链路初始化完成标志位 1: SRIO 链路初始化完成; 0: SRIO 链路初始化未完成。

<p>mode_1x</p>	<p>SRIO 运行模式</p> <p>1: SRIO 运行在降速模式, 即 4 个通道减速到 1 个通道运行;</p> <p>0: SRIO 运行在全速模式, 即 4 个通道全部运行。</p>
----------------	--

ZYNQ 端 SRIO 接收抓取示例如下图所示 (对应 DSP 端发起 NWrite 事务):

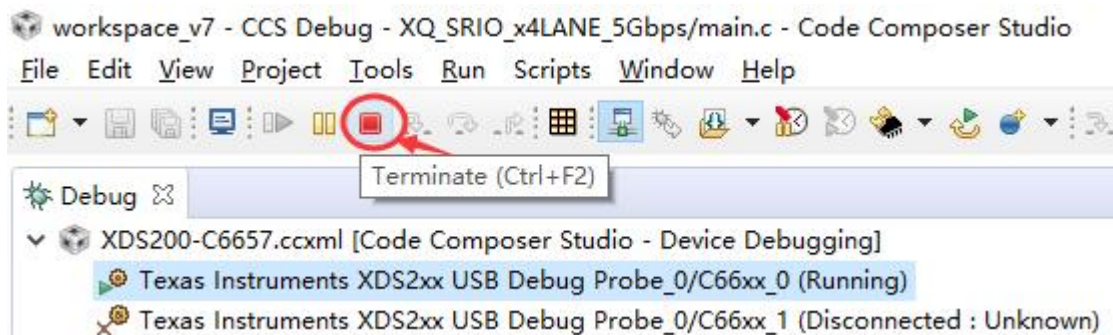


ZYNQ SRIO 发送抓取示例如下图所示 (对应 DSP 端发起 NRead 事务):

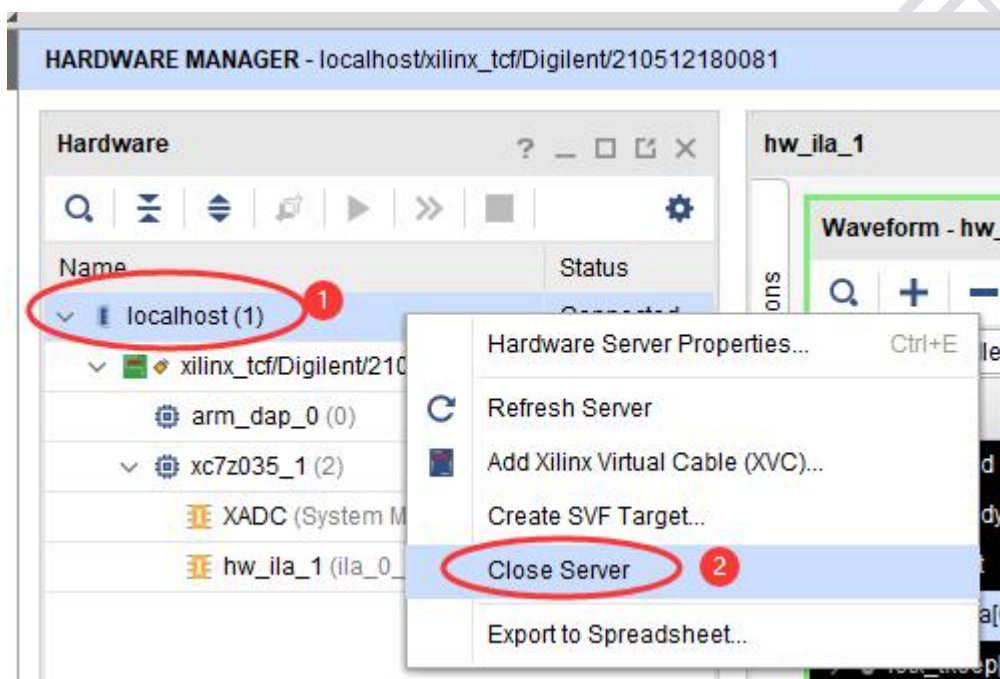


3.1.3.4 退出实验

CCS 软件窗口上, 点击 Terminate 断开 DSP 仿真器与板卡的连接:



Vivado 调试界面 Hardware Manager 窗口，右键单击 localhost(1)，在弹出的菜单中点击 Close Server，断开 ZYNQ JTAG 仿真器与板卡的连接：



最后，关闭板卡电源，实验结束。

3.2 ZYNQ 与 DSP 之间 EMIF16 通信

3.2.1 例程位置

ZYNQ 例程保存在资料盘中的 Demo\ZYNQ\PL\FPGA_DSP_EMIF 文件夹下。

DSP 例程保存在资料盘中的 Demo\DSP\XQ_EMIF16 文件夹下。

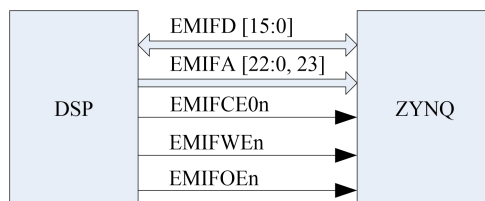
3.2.2 功能简介

实现 DSP 与 ZYNQ PL 端之间 EMIF16 接口传输功能。

DSP 首先通过 EMIF16 接口往 ZYNQ PL 端发送 4096 字节数据，然后再读回来，并检测数据是否有错，数据发送、读回以及错误情况实时打印。

ZYNQ PL 端开辟了一块 RAM 空间，用于存放 DSP 通过 EMIF16 接口写入的数据，同时用作 DSP 通过 EMIF16 接口读数据时的数据源。

DSP 与 ZYNQ PL 端之间 EMIF16 接口连接示意图如下图所示：



EMIF16 接口信号定义说明如下表格所示：

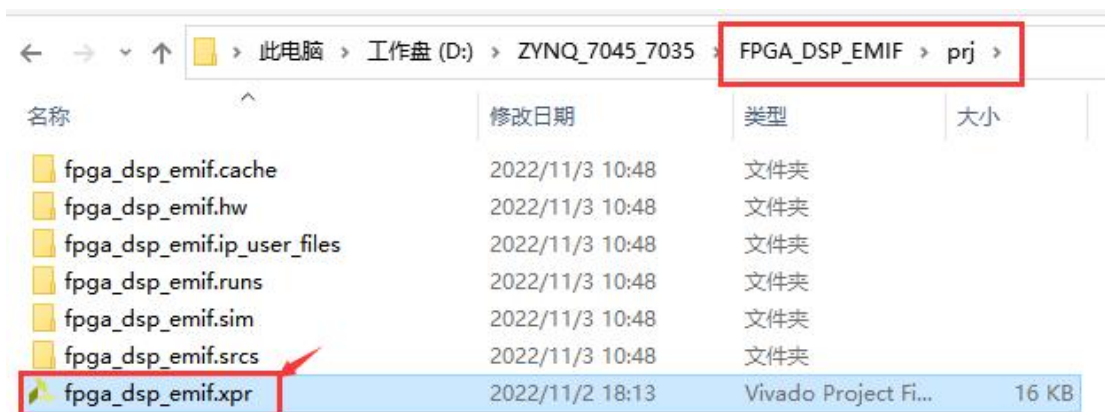
信号名	功能描述
EMIFD [15:0]	双向数据总线。 DSP 读数据时，对 DSP 而言是输入接口； DSP 写数据时，对 DSP 而言是输出接口。
EMIFA [23:0]	地址总线，注意实际使用时，23bit 作为最低位。ZYNQ PL 端例程只使用了其中的 12bits 作为地址总线作为示例。 DSP 端：为输出接口； ZYNQ PL 端：为输入接口。
EMIFCE0n	片选信号，低电平有效。 DSP 端：为输出接口； ZYNQ PL 端：为输入接口。
EMIFWEEn	写使能信号，低电平有效。 DSP 端：为输出接口； ZYNQ PL 端：为输入接口。
EMIFOEn	输出使能信号，即 DSP 读使能，低电平有效。 DSP 端：为输出接口； ZYNQ PL 端：为输入接口。

3.2.3 例程使用

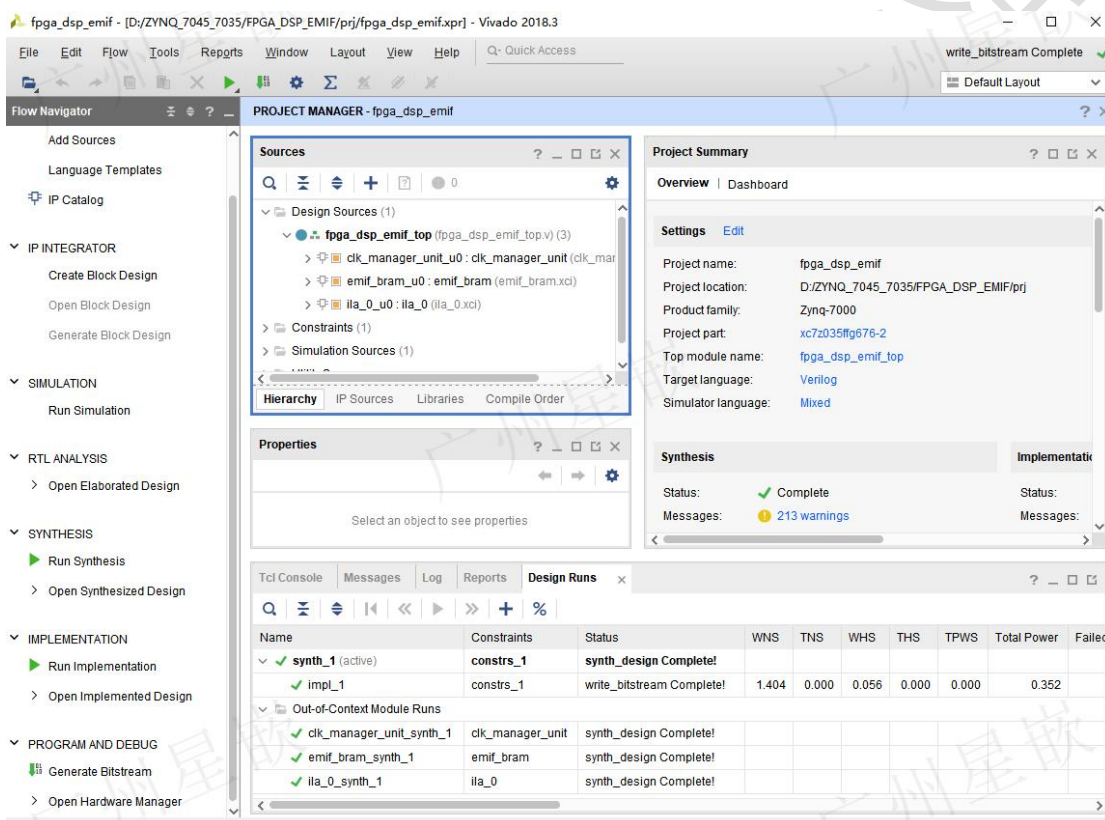
3.2.3.1 加载运行 ZYNQ 程序

3.2.3.1.1 打开 Vivado 工程

打开 Vivado 示例工程：

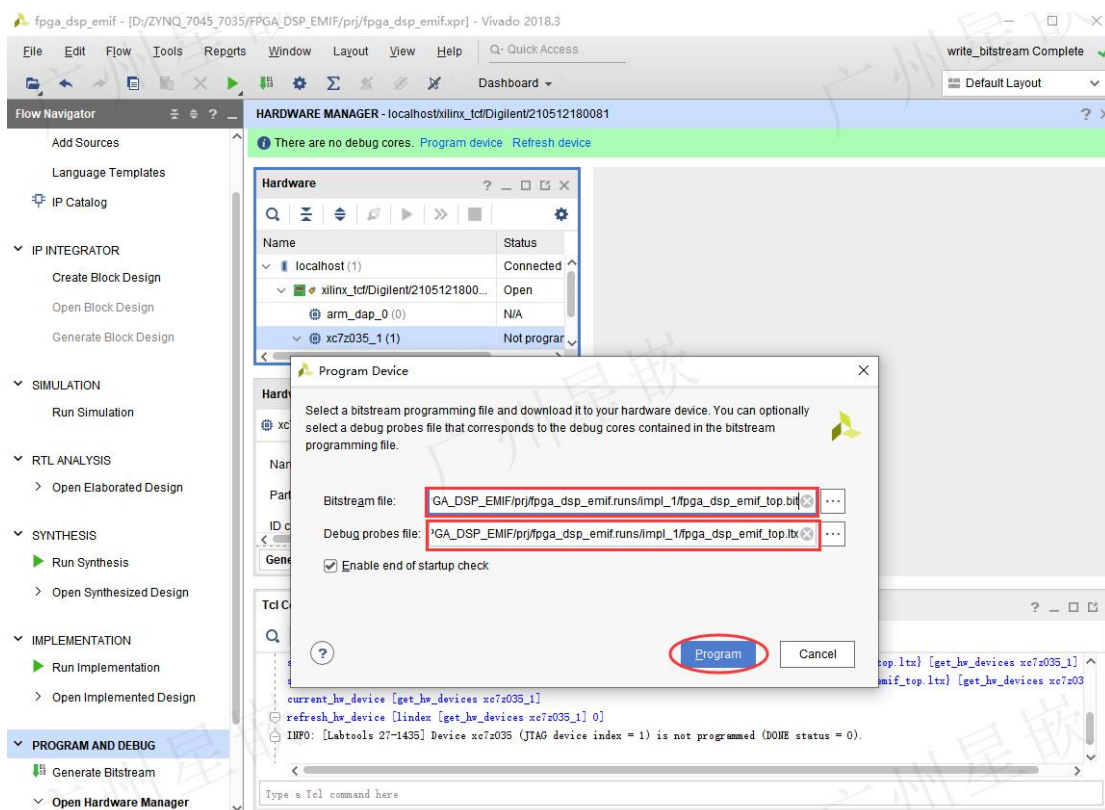


工程打开后界面如下图所示：



3.2.3.1.2 下载 ZYNQ PL 程序

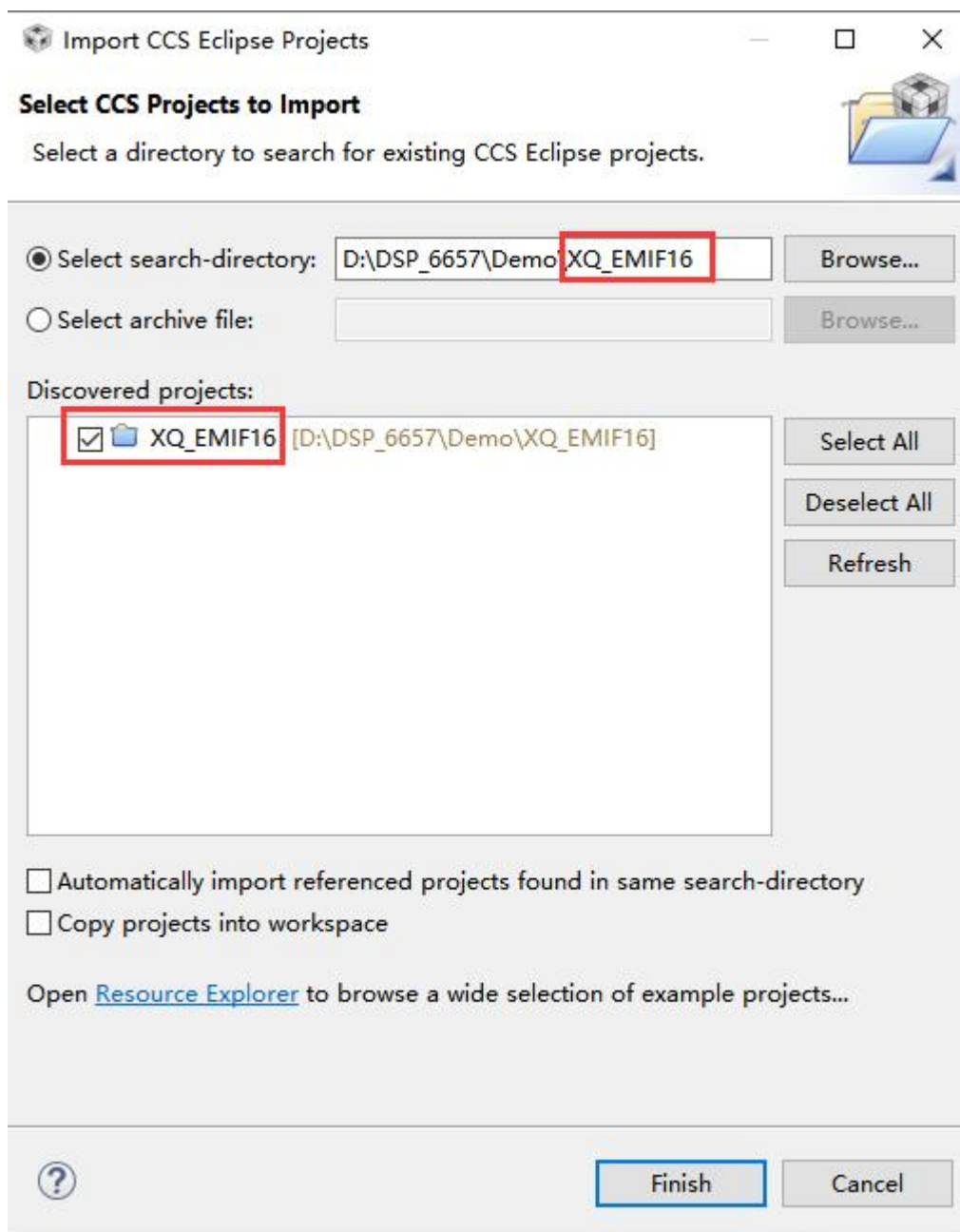
下载 bit 流文件 fpga_dsp_emif_top.bit，并且配套 fpga_dsp_emif_top.ltx 调试文件，如下图所示下载界面所示：



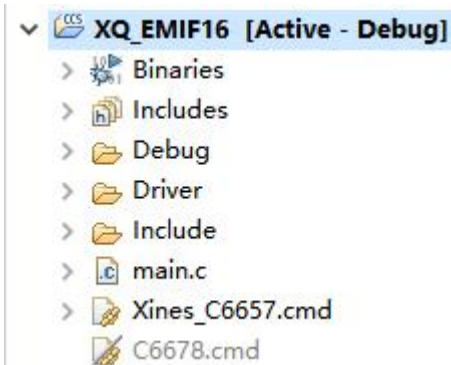
3.2.3.2 加载运行 DSP 程序

3.2.3.2.1 CCS 导入例程

CCS 软件导入 EMIF16 示例工程 XQ_EMIF16，如下图所示：

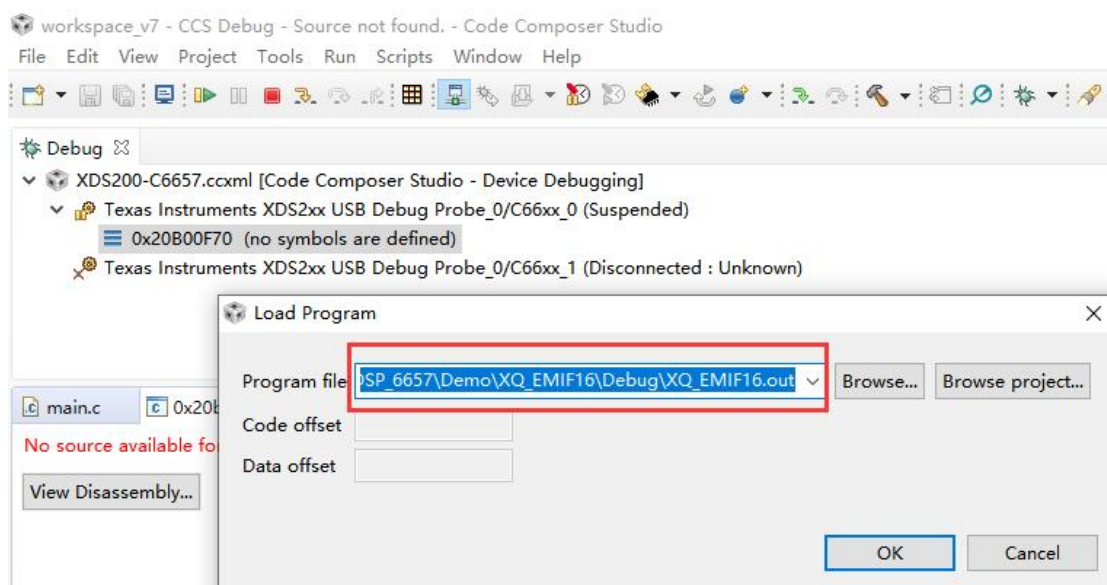


CCS 示例工程导入后界面如下图所示：

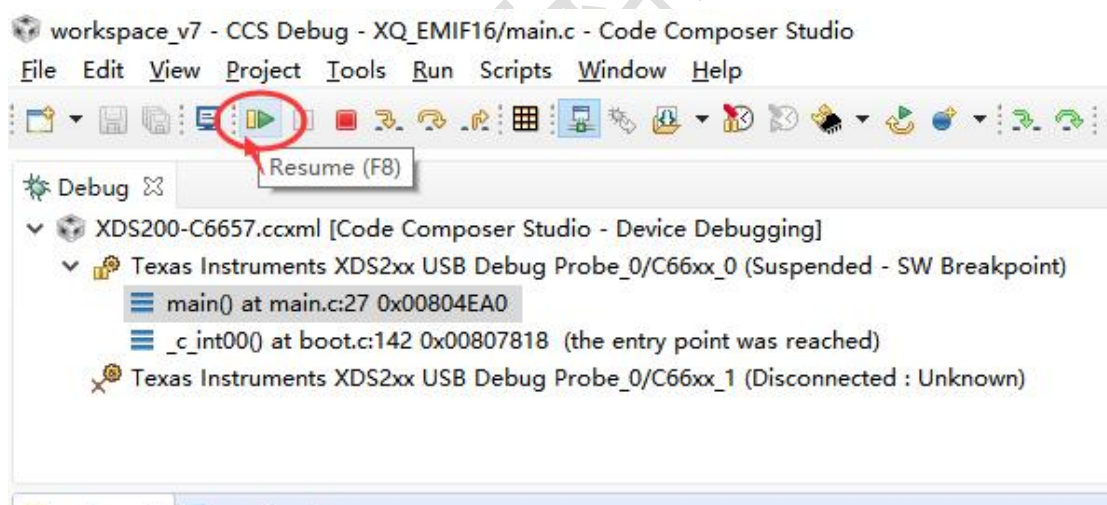


3.2.3.2.2 下载运行 CCS 程序

下载 DSP 可执行文件 XQ_EMIF16.out:



点击 Resume 运行 DSP 程序:



3.2.3.3 运行结果说明

3.2.3.3.1 DSP 程序运行结果

DSP 首先通过 EMIF16 接口往 ZYNQ PL 端发送 4096 字节数据, 然后再读回来, 并检测数据是否有错, 数据发送、读回以及错误情况实时打印, 如下图所示:

```

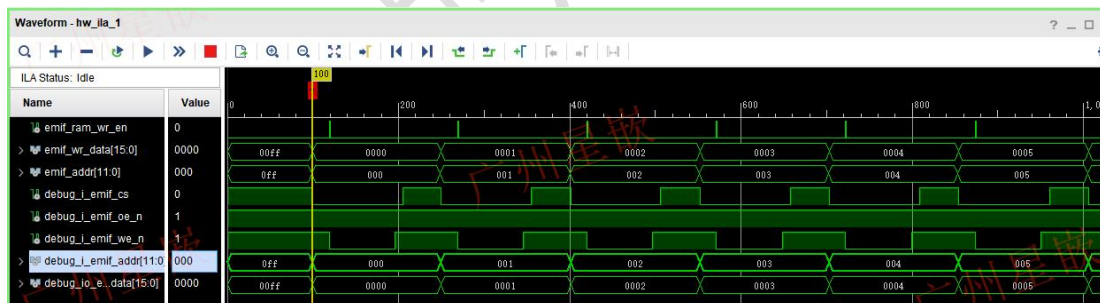
Console X
XDS200-C6657.ccxml:CIO
=====

DSP Write to ZYNQ PL 4096 Bytes
DSP Read from ZYNQ PL 4096 Bytes
=====
EMIFA_error_count = 0 Bytes
=====

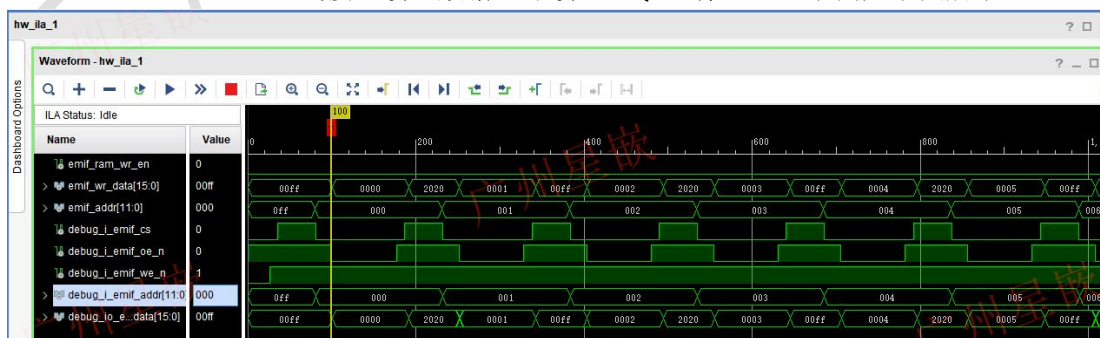
DSP Write to ZYNQ PL 4096 Bytes
DSP Read from ZYNQ PL 4096 Bytes
=====
EMIFA_error_count = 0 Bytes
=====
    
```

3.2.3.3.2 ZYNQ PL 程序运行结果

ZYNQ PL 端提供的 ILA 调试窗口，可以实时抓取采集 EMIF16 总线信号时序波形。DSP 通过 EMIF16 总线接口发送数据（即写 ZYNQ PL 端 RAM）示例如下图所示：



DSP 通过 EMIF16 总线接口读回数据（即读 ZYNQ PL 端 RAM）示例如下图所示：



3.2.3.4 退出实验

CCS 软件窗口上，点击 Terminate 断开 DSP 仿真器与板卡的连接。

Vivado 调试界面 Hardware Manager 窗口，右键单击 localhost(1)，在弹出的菜单中点击 Close Server，断开 ZYNQ JTAG 仿真器与板卡的连接。

最后，关闭板卡电源，实验结束。

3.3 ZYNQ 与 DSP 之间 uPP 通信

3.3.1 例程位置

ZYNQ 例程保存在资料盘中的 Demo\ZYNQ\PL\FPGA_DSP_uPP 文件夹下。

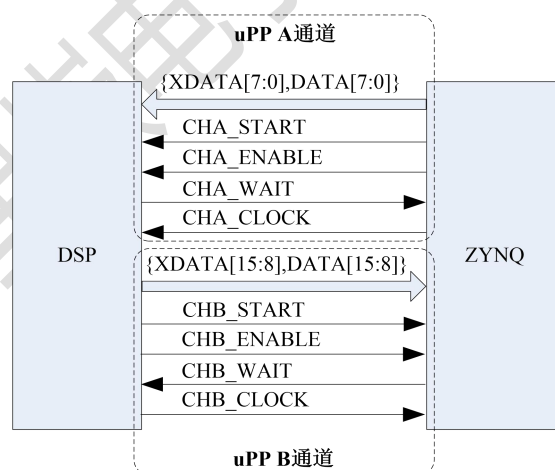
DSP 例程保存在资料盘中的 Demo\DSP\XQ_uPP 文件夹下。

3.3.2 功能简介

实现 DSP 与 ZYNQ PL 端之间 uPP 接口传输功能。

DSP 通过 uPP B 通道往 ZYNQ PL 端发送 204800 字节数据，ZYNQ PL 端收到 uPP B 通道数据后直接交给 uPP A 通道送回至 DSP。DSP 程序比对 uPP A 通道接收到的数据和 uPP B 通道发送出去的数据，检测数据是否有错，uPP A/B 两个通道的数据收发以及错误情况实时打印。

DSP 与 ZYNQ PL 端之间 uPP A/B 两个通道接口的连接示意图如下图所示：

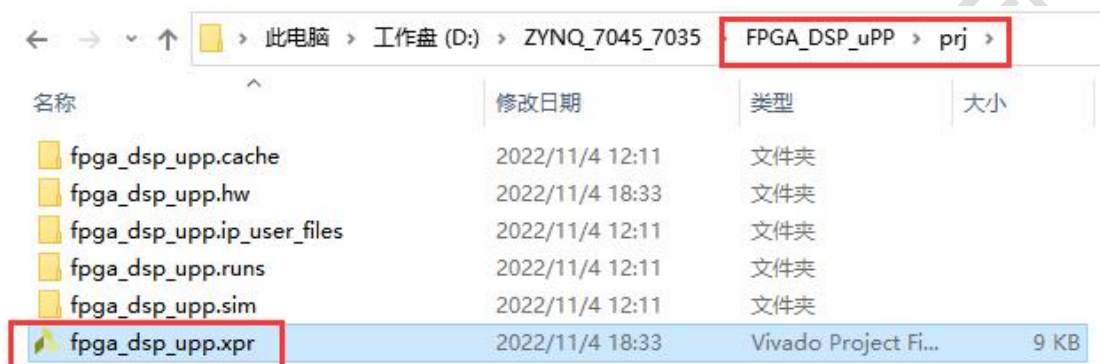


3.3.3 例程使用

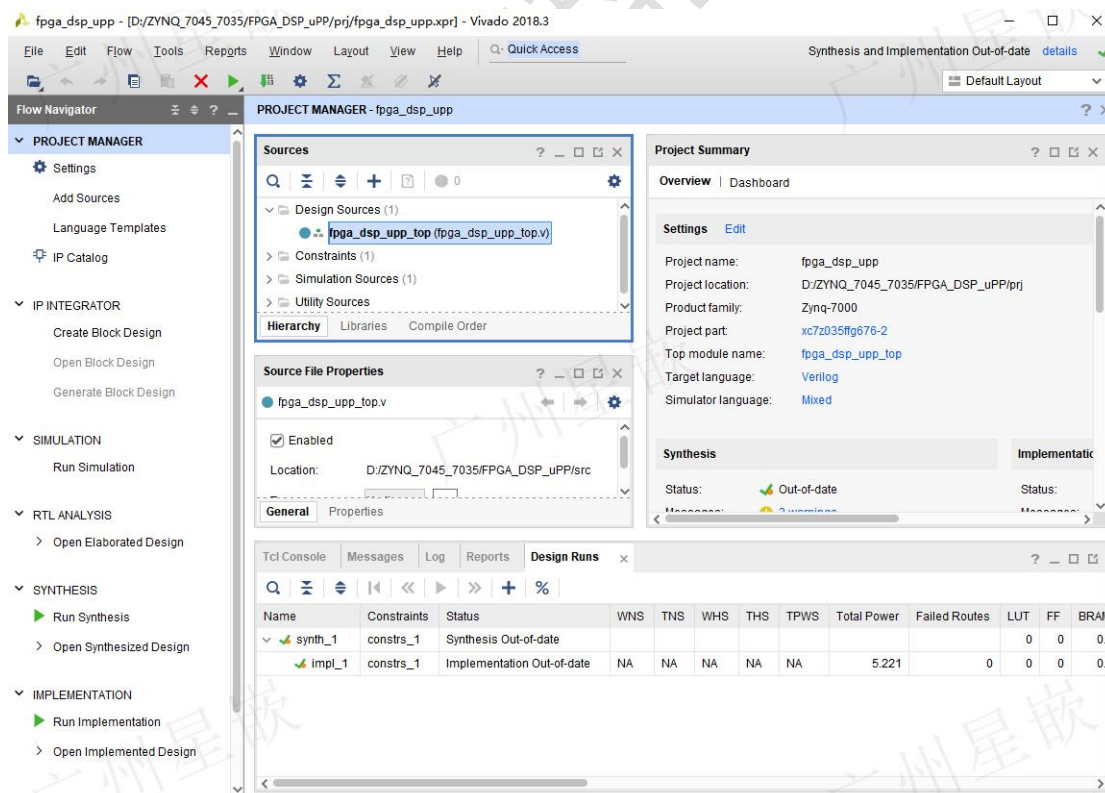
3.3.3.1 加载运行 ZYNQ 程序

3.3.3.1.1 打开 Vivado 工程

打开 Vivado 示例工程：

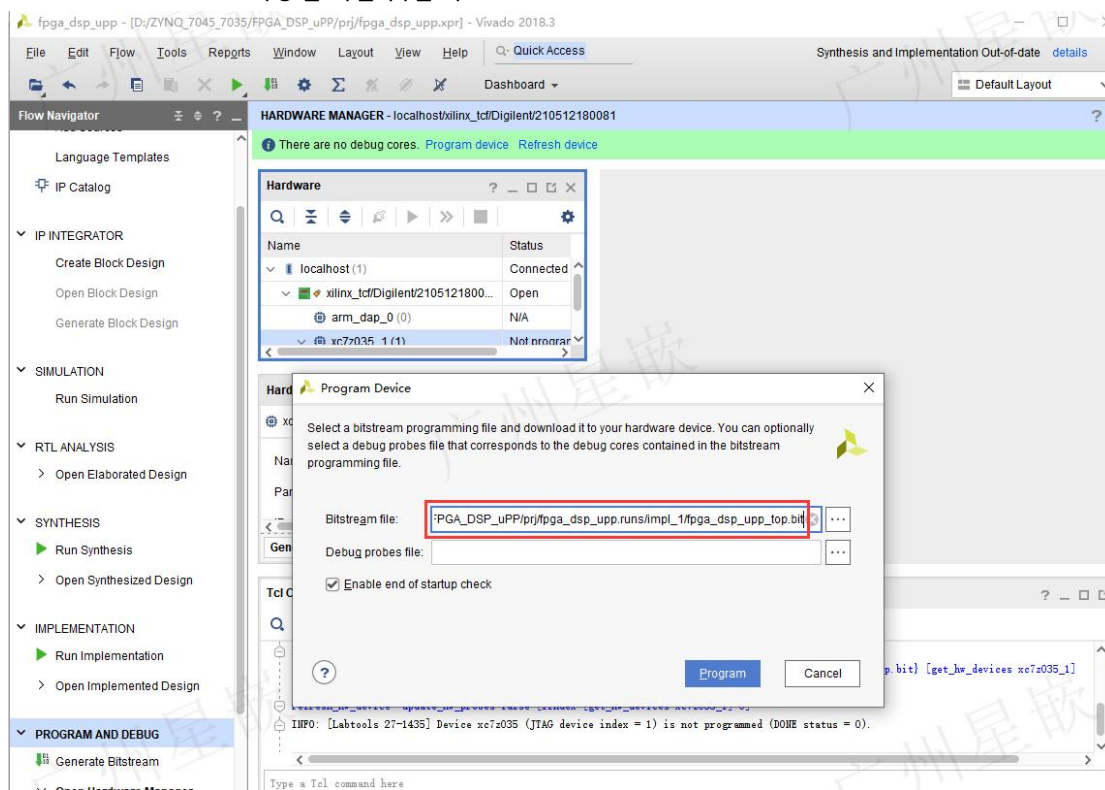


工程打开后界面如下图所示：



3.3.3.1.2 下载 ZYNQ PL 程序

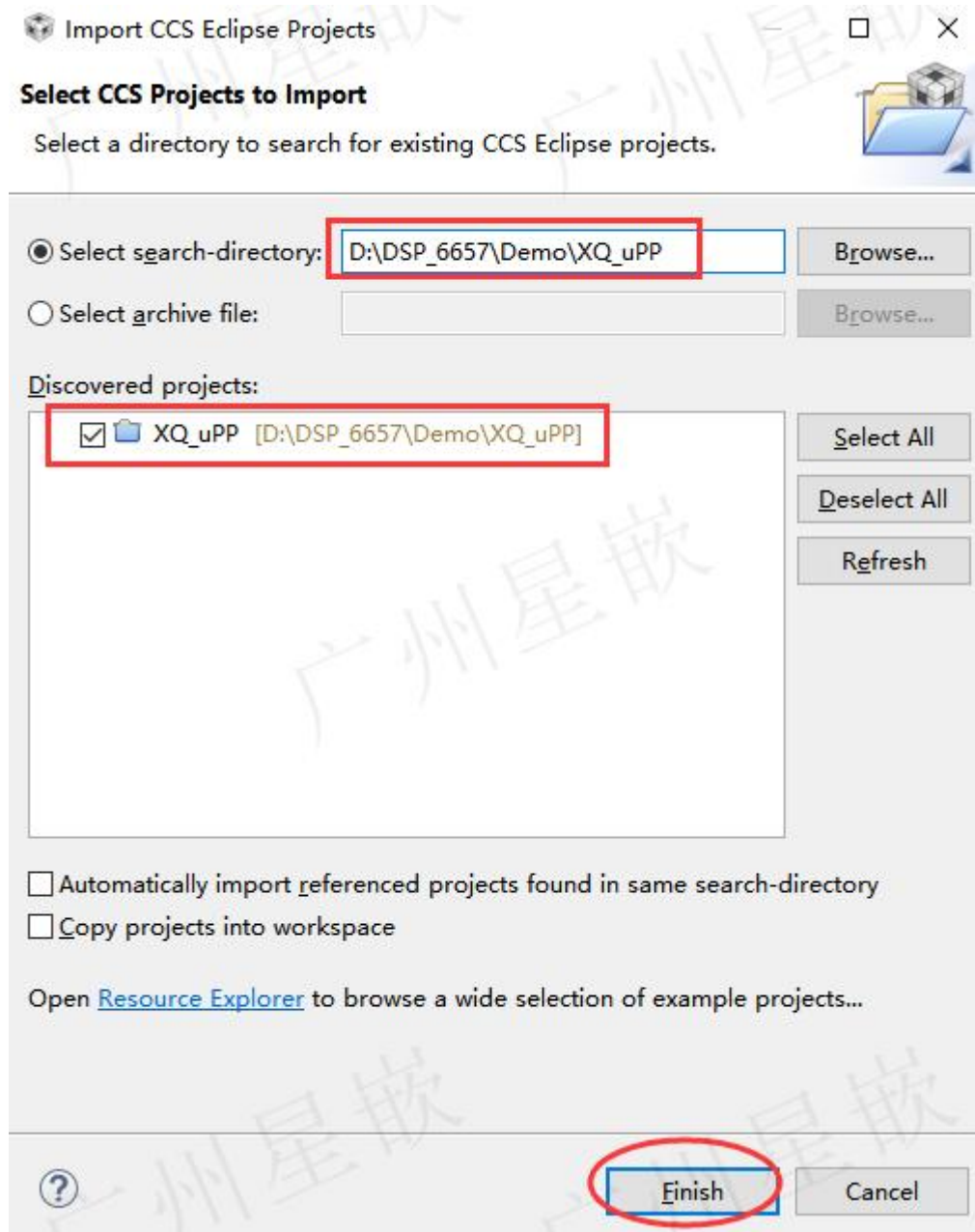
下载 bit 流文件 fpga_dsp_upp_top.bit，如下图下载界面所示：



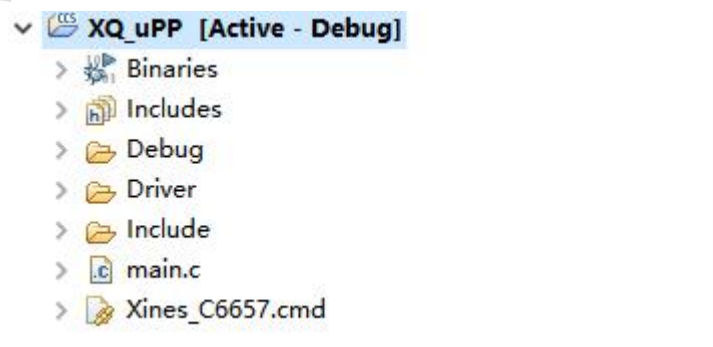
3.3.3.2 加载运行 DSP 程序

3.3.3.2.1 CCS 导入例程

CCS 软件导入 uPP 示例工程 XQ_uPP，如下图所示：

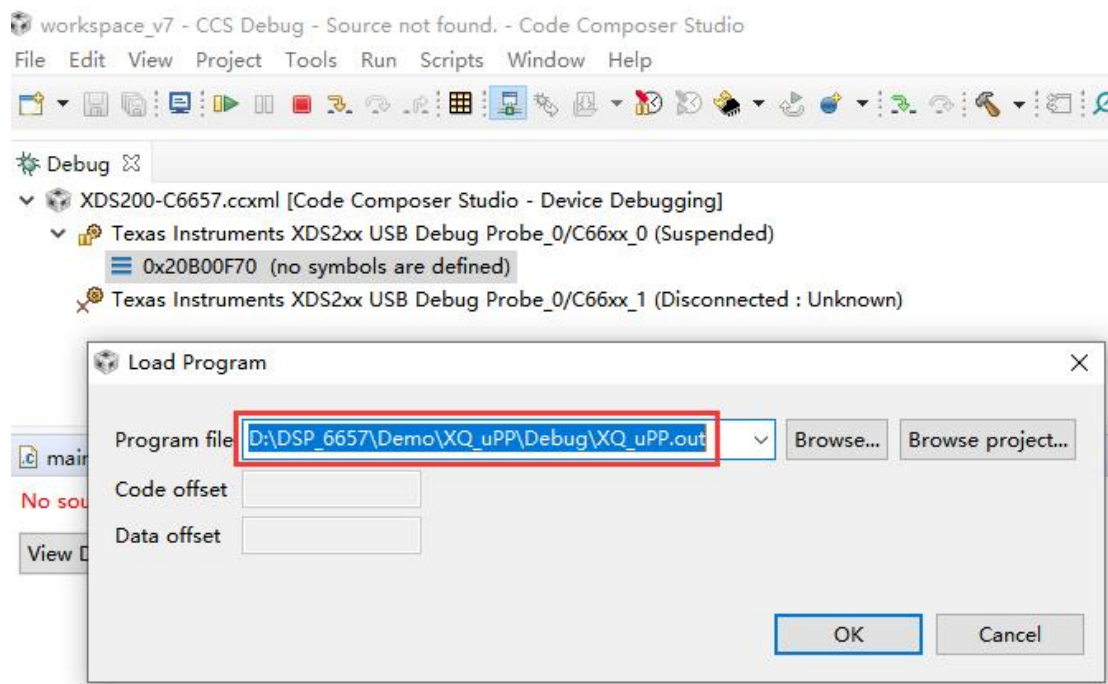


CCS 示例工程导入后界面如下图所示：

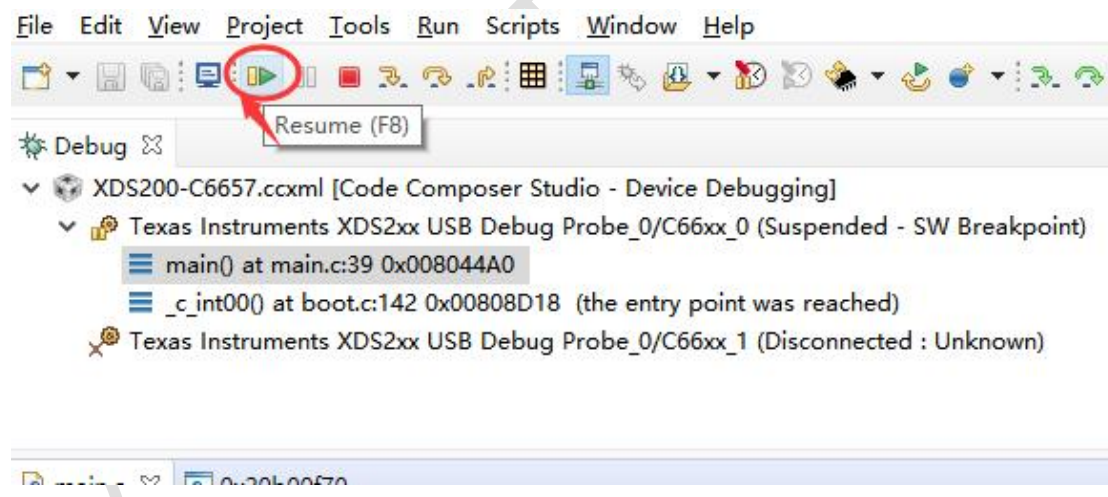


3.3.3.2 下载运行 CCS 程序

下载 DSP 可执行文件 XQ_uPP.out:



点击 Resume 运行 DSP 程序:



3.3.3.3 运行结果说明

3.3.3.3.1 DSP 程序运行结果

DSP 每完成一次 uPP 收、发传输后 (uPP B 通道发送、uPP A 通道接收), 程序对 uPP A/B 两个通道的收发数据进行比对, 如果传输数据正常, 则打印完成且无错误信息, 否则打印此



次 uPP 传输存在错误的提示信息，如下图所示为正确传输打印信息：

```

Console X
XDS200-C6657.cxml:CIO
    uPP Transmission finished with 204800 Bytes data been sent and received!
    Verifying the results...
    PASSED : uPP transmission completed with no errors !

=====
=== uPP Transmission Test : Start...
=====
    Initializing uPP buffers...
    Starting uPP transfers...
    uPP Transmission finished with 204800 Bytes data been sent and received!
    Verifying the results...
    PASSED : uPP transmission completed with no errors !
  
```

3.3.3.4 退出实验

CCS 软件窗口上，点击 Terminate 断开 DSP 仿真器与板卡的连接。

Vivado 调试界面 Hardware Manager 窗口，右键单击 localhost(1)，在弹出的菜单中点击 Close Server，断开 ZYNQ JTAG 仿真器与板卡的连接。

最后，关闭板卡电源，实验结束。

3.4 ZYNQ 与 DSP 之间 GPIO 通信

3.4.1 例程位置

ZYNQ 例程保存在资料盘中的 Demo\ZYNQ\PL\FPGA_DSP_GPIO 文件夹下。

DSP 例程保存在资料盘中的 Demo\DSP\XQ_GPIO_FPGA 文件夹下。

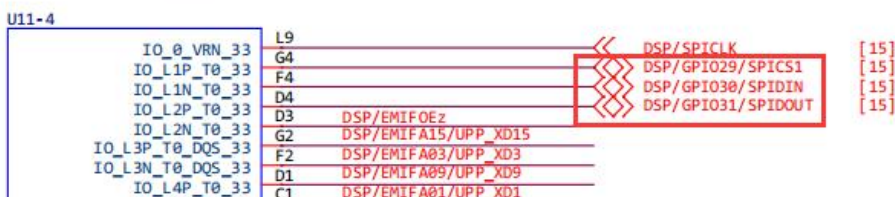
3.4.2 功能简介

实现 DSP 与 ZYNQ PL 端之间 GPIO 接口传输功能。

DSP 与 ZYNQ PL 端之间有 3 根 GPIO 信号相连，如下原理图标注所示：

ZYNQ PL CONNECTED TO DSP

BANK33



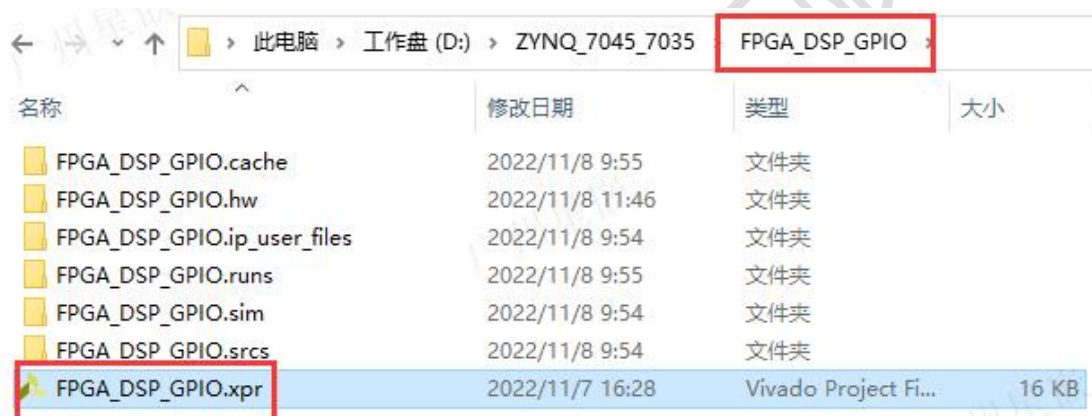
DSP 示例通信程序将 GPIO29、GPIO30 两个 GPIO 设置为输出（对 ZYNQ 而言就是输入），GPIO31 设置为输入（对 ZYNQ 而言就是输出）。DSP 在 GPIO29、GPIO30 两个 GPIO 上产生方波信号，ZYNQ 可通过 ILA 软逻辑分析仪抓取波形查看；ZYNQ 通过 VIO 虚拟 IO 往 GPIO31 上输出高、低电平，DSP 示例程序里面检测 GPIO31 下降沿中断，一旦检测到 GPIO31 管脚下降沿中断信号，则打印测试信息。

3.4.3 例程使用

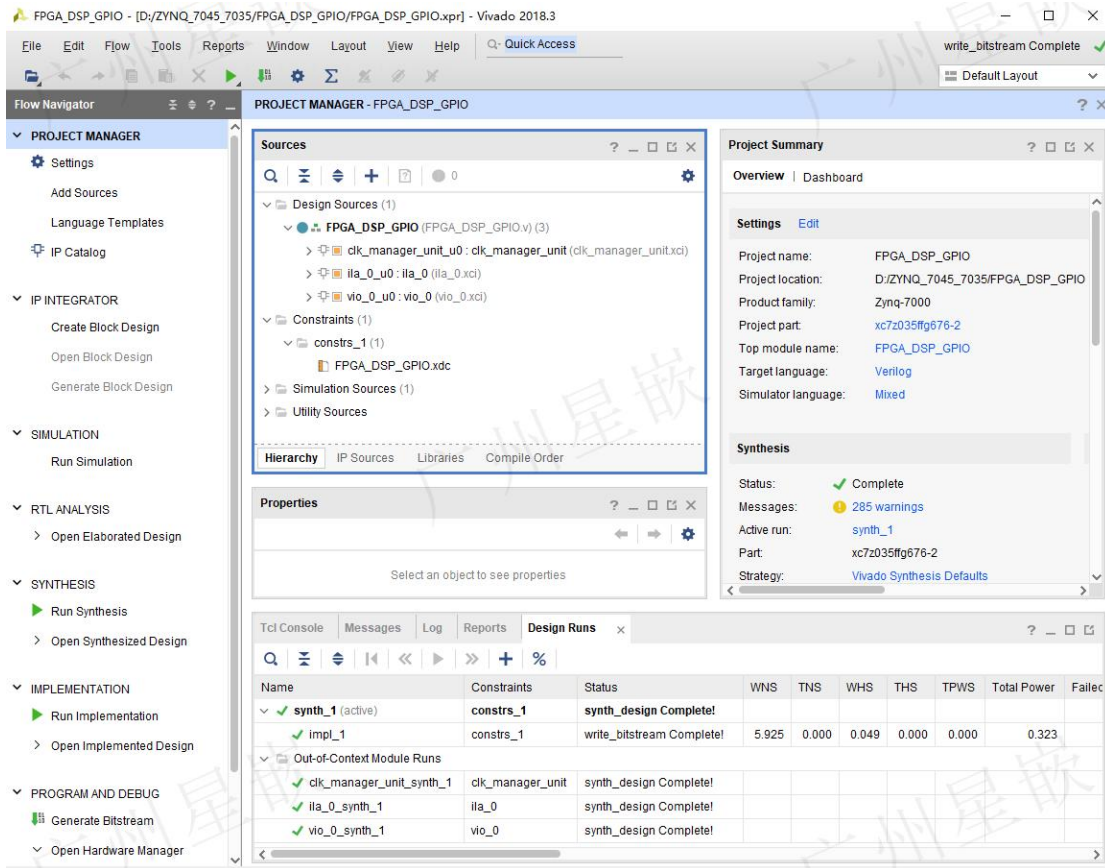
3.4.3.1 加载运行 ZYNQ 程序

3.4.3.1.1 打开 Vivado 工程

打开 Vivado 示例工程：

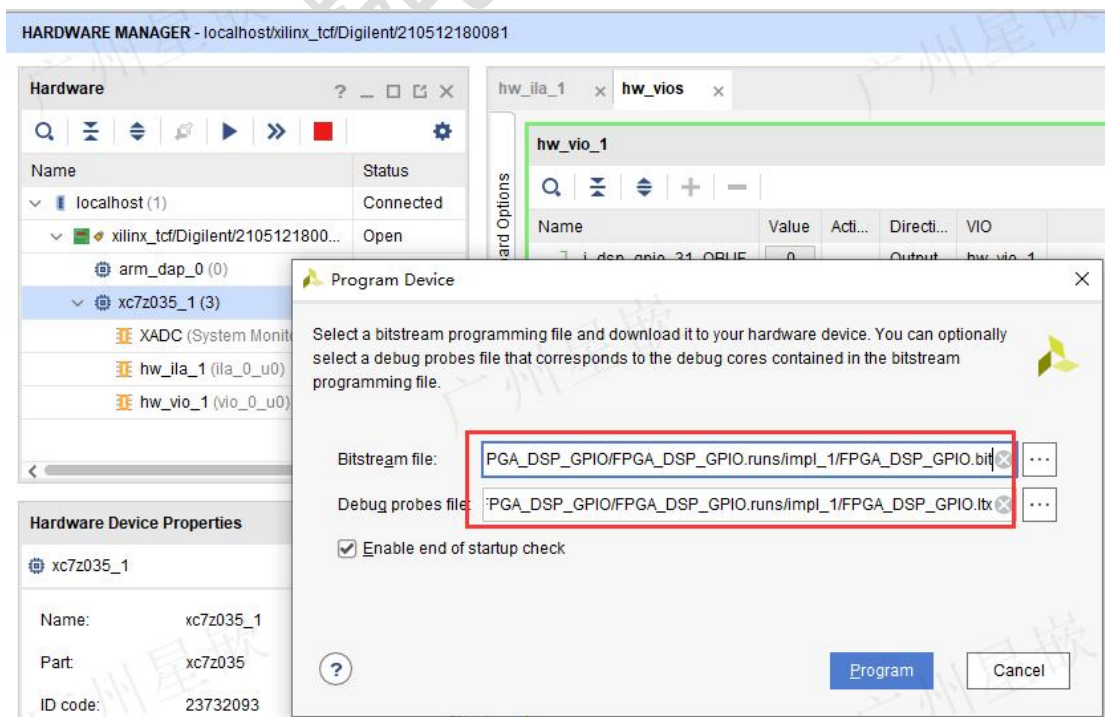


工程打开后界面如下图所示：



3.4.3.1.2 下载 ZYNQ PL 程序

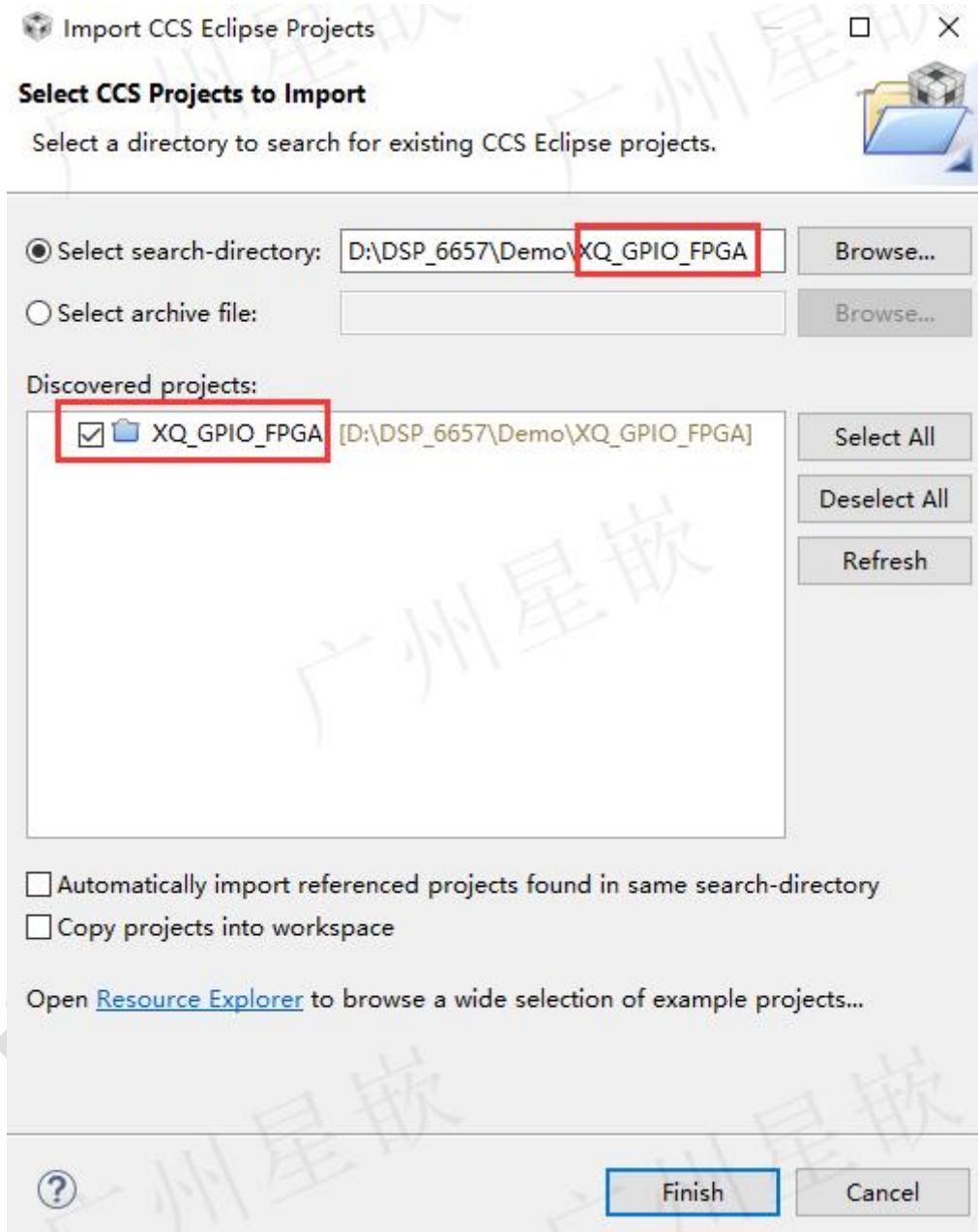
下载 bit 流文件 FPGA_DSP_GPIO.bit，如下图下载界面所示：



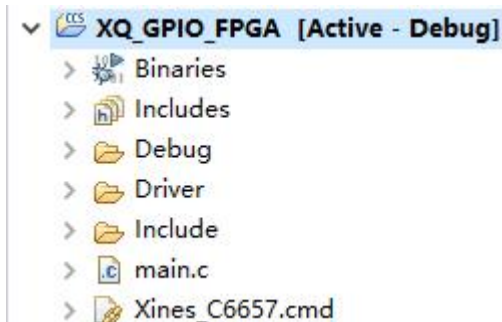
3.4.3.2 加载运行 DSP 程序

3.4.3.2.1 CCS 导入例程

CCS 软件导入示例工程 XQ_GPIO_FPGA，如下图所示：

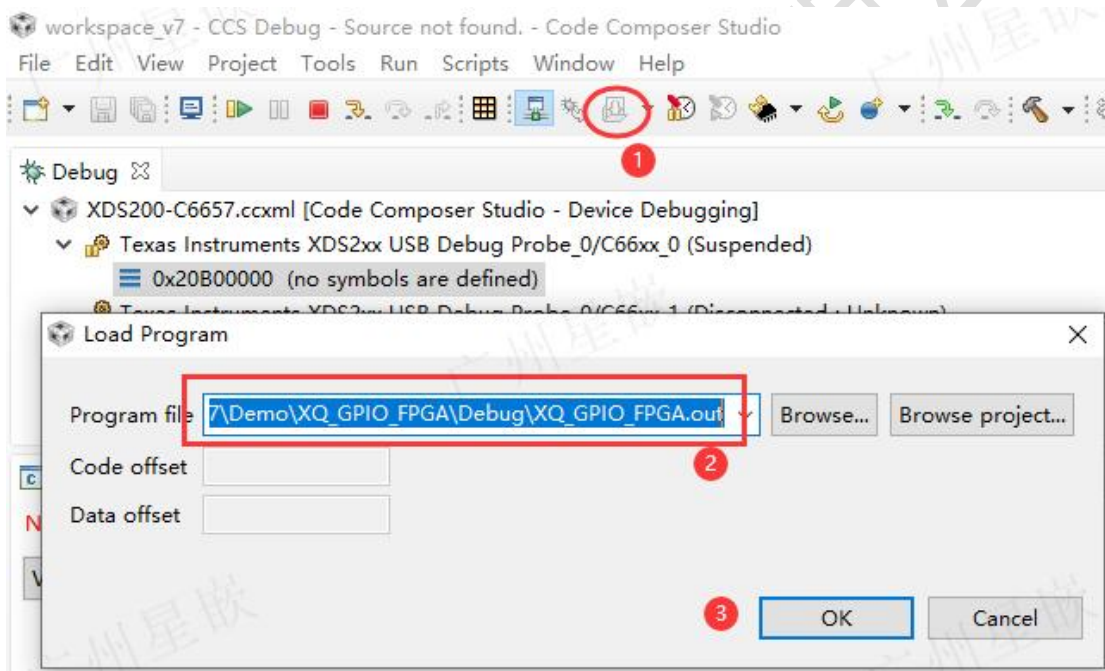


CCS 示例工程导入后界面如下图所示：

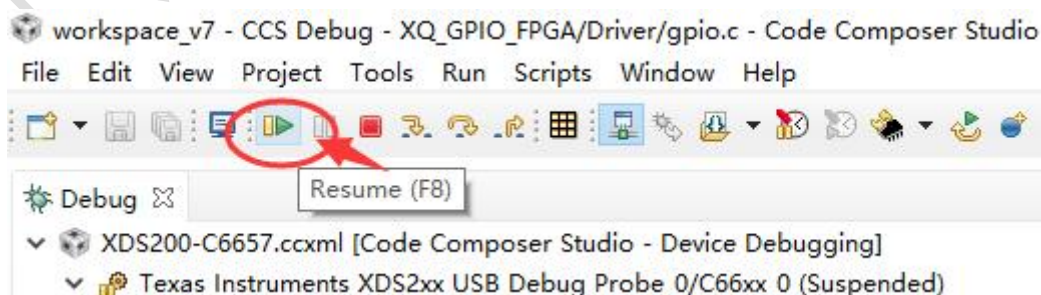


3.4.3.2.2 下载运行 CCS 程序

下载 DSP 可执行文件 XQ_GPIO_FPGA.out:



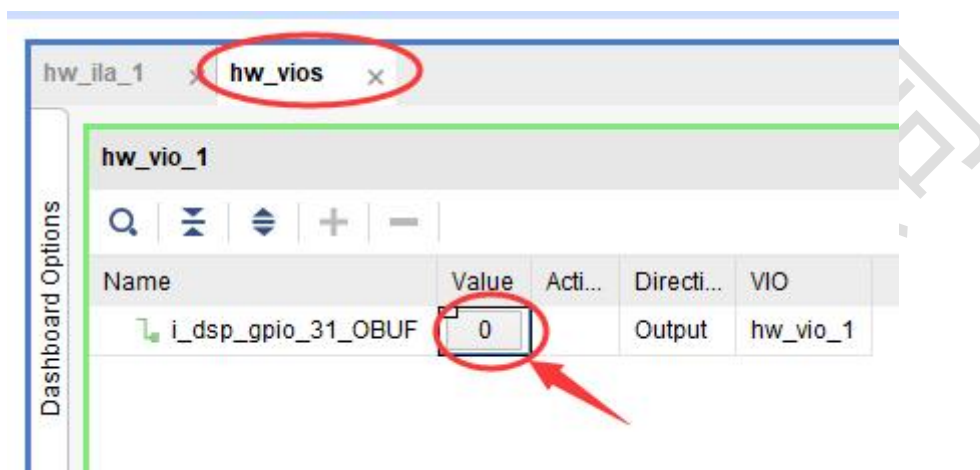
点击 Resume 运行 DSP 程序:



3.4.3.3 运行结果说明

3.4.3.3.1 DSP 程序运行结果

点击 ZYNQ PL 调试界面上 hw_vios 窗口中 Value 下面的数字 0 或 1，从而在 GPIO31 上产生高低电平信号：

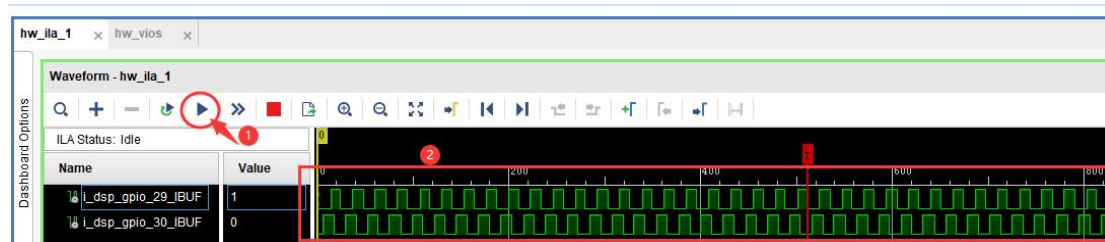


每当 GPIO31 上产生高低电平下降沿信号，DSP 程序进入中断服务函数，打印如下图所示的信息：

```
Console XDS200-C6657.ccxml:CIO
[C66xx_0] ZYNQ PL GPIO falling edge detected!
ZYNQ PL GPIO falling edge detected!
ZYNQ PL GPIO falling edge detected!
ZYNQ PL GPIO falling edge detected!
ZYNQ PL GPIO falling edge detected!
ZYNQ PL GPIO falling edge detected!
ZYNQ PL GPIO falling edge detected!
```

3.4.3.3.2 ZYNQ 程序运行结果

点击 hw_ila_1 窗口上三角符号的采集触发按钮，如下图①处标识的按钮，可查看到 DSP 通过 GPIO29、GPIO30 两个 GPIO 管脚发过来的方波信号：



3.4.3.4 退出实验

CCS 软件窗口上，点击 **Terminate** 断开 DSP 仿真器与板卡的连接。

Vivado 调试界面 **Hardware Manager** 窗口，右键单击 **localhost(1)**，在弹出的菜单中点击 **Close Server**，断开 ZYNQ JTAG 仿真器与板卡的连接。

最后，关闭板卡电源，实验结束。

4 DSP 单独例程

4.1 DSP 以太网通信

4.1.1 例程位置

DSP 例程保存在资料盘中的 `Demo\DSP\XQ_NDK_TCP_UDP_ECHO` 文件夹下。

4.1.2 功能简介

实现 DSP 千兆以太网通信测试功能，实现 TCP、UDP 数据回显功能。

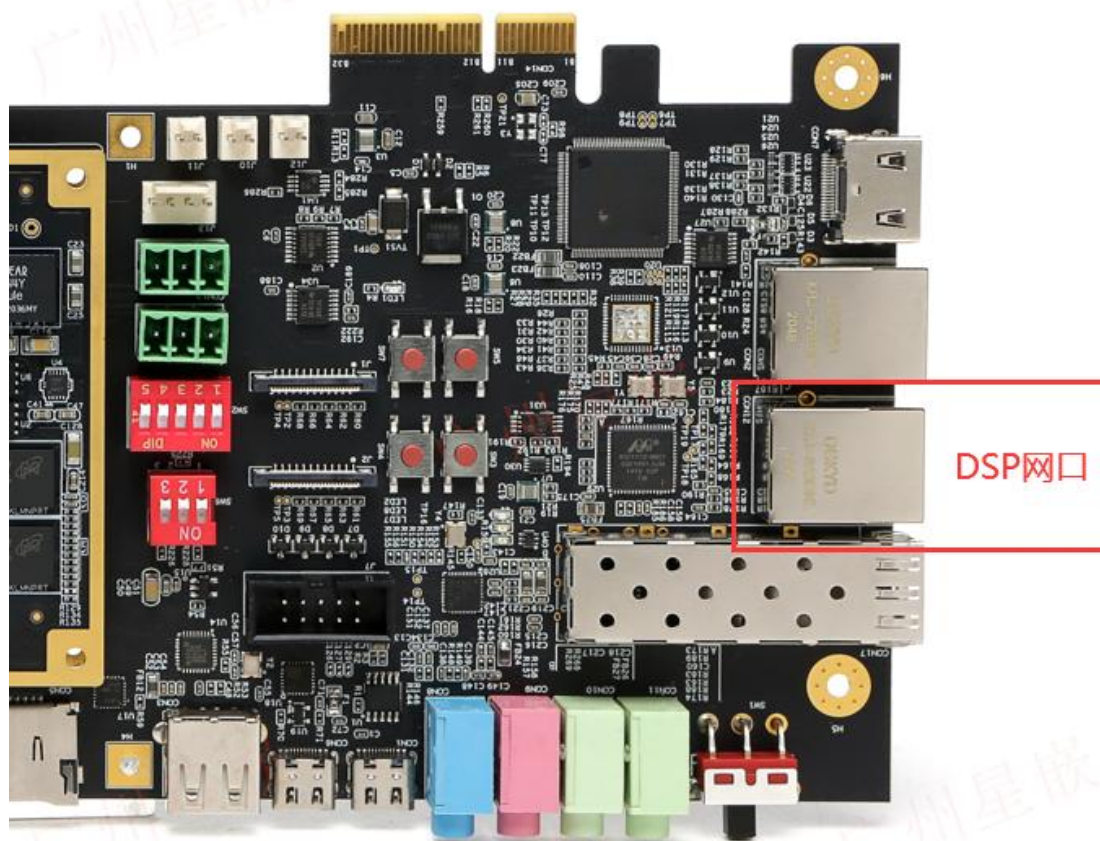
`main.c` 文件中的 `NetworkOpen` 函数里面创建了两个守护进程，端口号 7 对应任务 `dtask_udp_echo`，响应 TCP 数据接收，并将 TCP 数据回传；端口号 8 对应任务 `dtask_tcp_echo`，响应 UDP 数据接收，并将 UDP 数据回传：

```
main.c
322 static HANDLE hTcpEcho=0;
323 //
324 // NetworkOpen
325 //
326 // This function is called after the configuration has booted
327 //
328 static void NetworkOpen()
329 {
330     // Create our local udp echo server
331     hUdpEcho = DaemonNew( SOCK_DGRAM, 0, 7, dtask_udp_echo,
332                          OS_TASKPRINORM, OS_TASKSTKNORM, 0, 1 );
333
334     // Create our local tcp echo server
335     hTcpEcho = DaemonNew( SOCK_STREAMNC, 0, 8, dtask_tcp_echo,
336                          OS_TASKPRINORM, OS_TASKSTKNORM, 0, 1 );
337 }
```

4.1.3 例程使用

4.1.3.1 网线连接

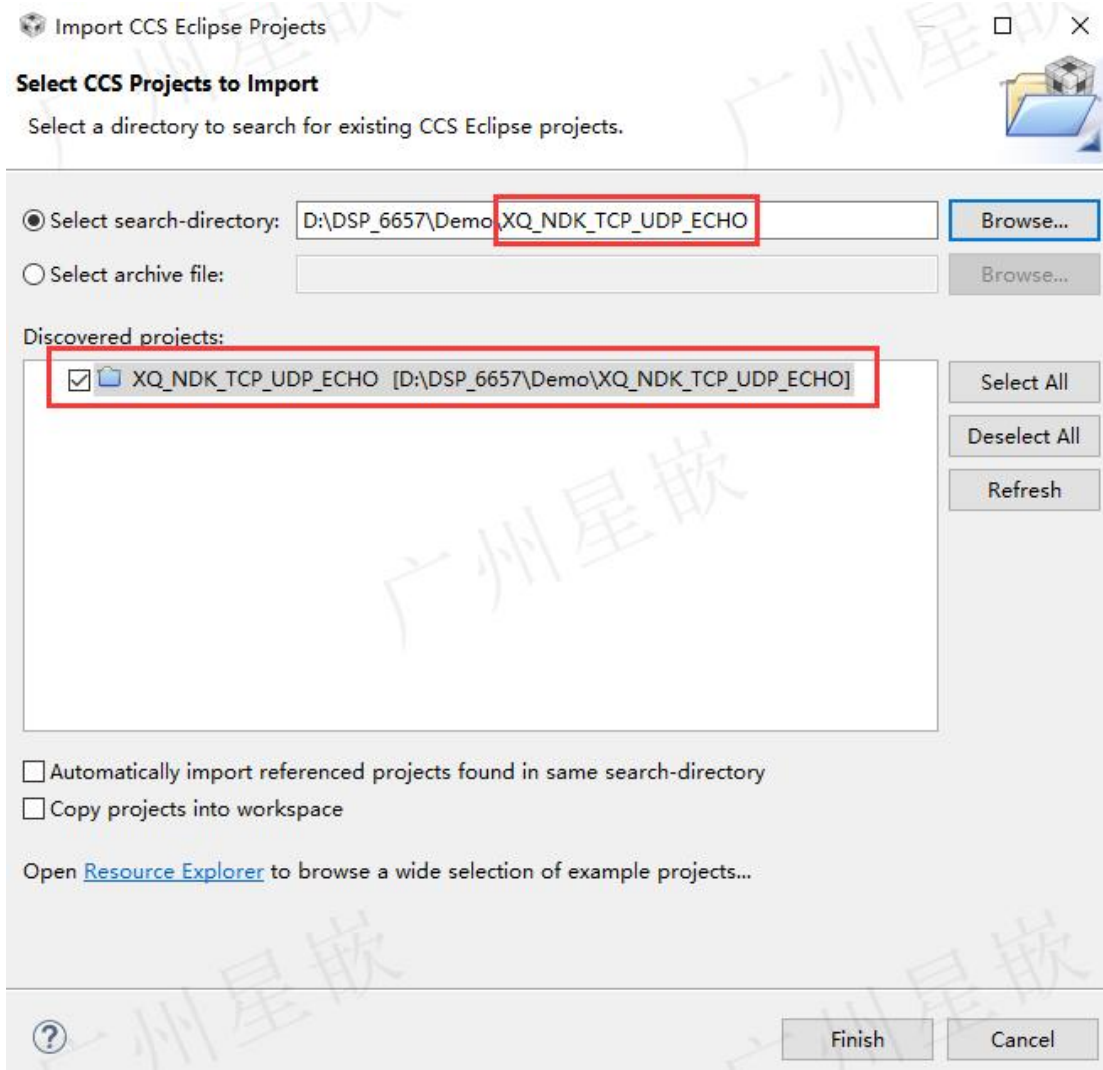
用网线连接板卡上 DSP 网口至交换机，要求和测试电脑在同一个网段：



4.1.3.2 加载运行 DSP 程序

4.1.3.2.1 CCS 导入例程

CCS 软件导入示例工程 XQ_NDK_TCP_UDP_ECHO，如下图所示：

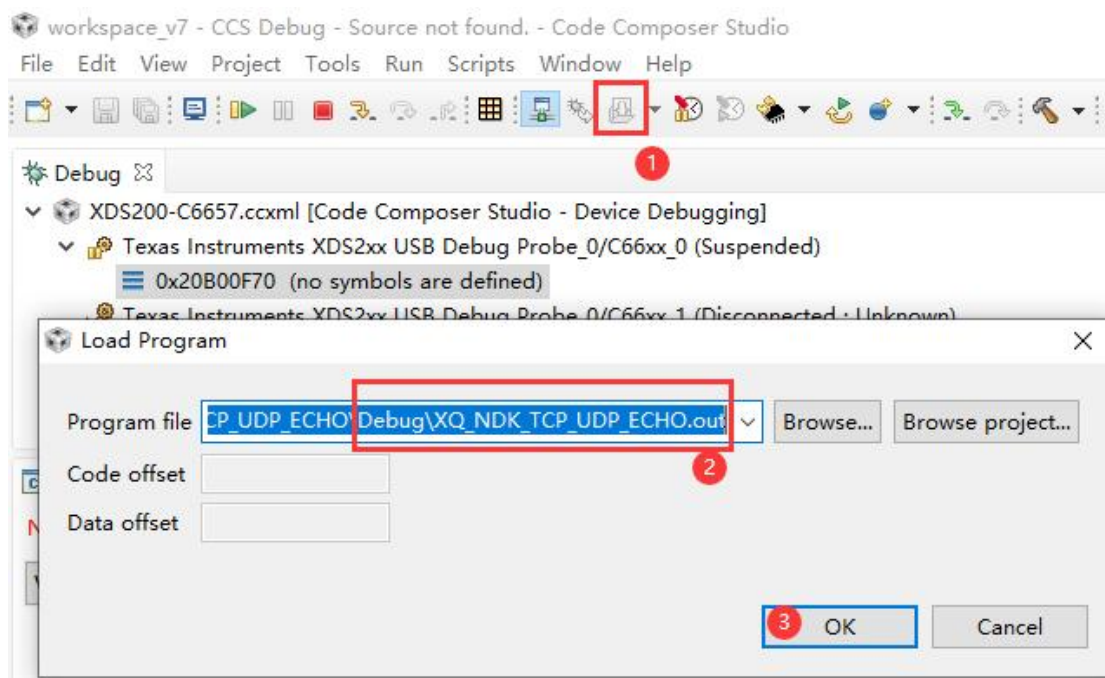


CCS 示例工程导入后界面如下图所示：

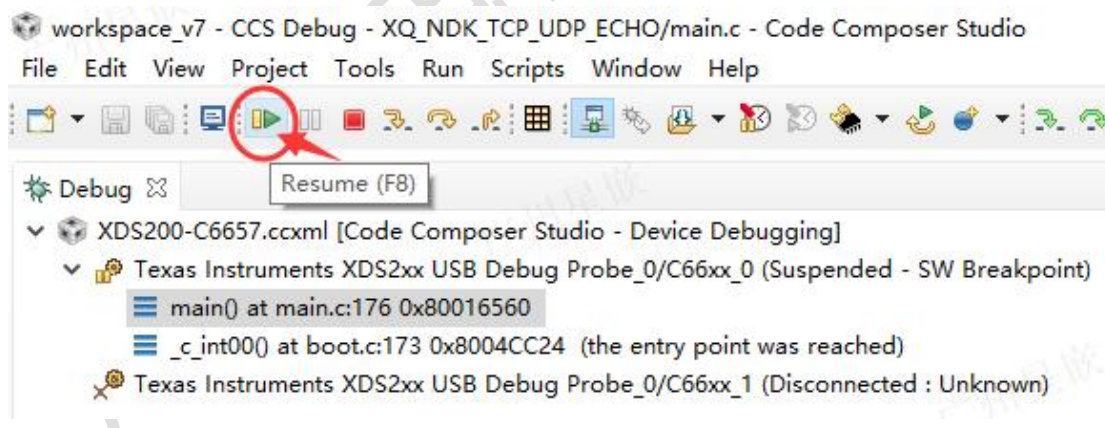


4.1.3.2.2 下载运行 CCS 程序

下载 DSP 可执行文件 XQ_NDK_TCP_UDP_ECHO.out:



点击 Resume 运行 DSP 程序:



4.1.3.3 运行结果说明

4.1.3.3.1 DSP 程序运行结果

DSP 程序运行后，CCS 控制台打印如下图所示的信息：

```
Console XDS200-C6657.ccxml:CIO
[C66xx_0]
Xines TCP/IP Stack 'TCP/UDP Echo' Application

emac_init: core 0, port 0, total number of channels/MAC addresses: 1/1
MAC addresses configured for channel 0:
00-35-FF-AF-BF-90
SGMII reset successful.....
SGMII config successful.....
emac_open core 0 port 0 successfully
Registration of the EMAC Successful, waiting for link up ..
Service Status: DHCP      : Enabled   :          : 000
Service Status: DHCP      : Enabled   : Running  : 000
Network Added: If-1:192.168.1.102
Service Status: DHCP      : Enabled   : Running  : 017
```

上图打印信息显示，DSP 网络链接成功，DSP 网络 MAC 地址为 00-35-FF-AF-BF-90，分配到的 IP 地址为 192.168.1.102。

4.1.3.3.2 Ping 测试

打开测试电脑命令行运行窗口，并输入 `ping 192.168.1.102` 命令，然后回车，测试电脑 ping DSP 网络的效果：

```
管理员: 命令提示符
Microsoft Windows [版本 10.0.19044.2130]
(c) Microsoft Corporation. 保留所有权利。

C:\Users\Administrator>ping 192.168.1.102

正在 Ping 192.168.1.102 具有 32 字节的数据:
来自 192.168.1.102 的回复: 字节=32 时间<1ms TTL=255
来自 192.168.1.102 的回复: 字节=32 时间<1ms TTL=255
来自 192.168.1.102 的回复: 字节=32 时间<1ms TTL=255
来自 192.168.1.102 的回复: 字节=32 时间=1ms TTL=255

192.168.1.102 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间(以毫秒为单位):
        最短 = 0ms, 最长 = 1ms, 平均 = 0ms

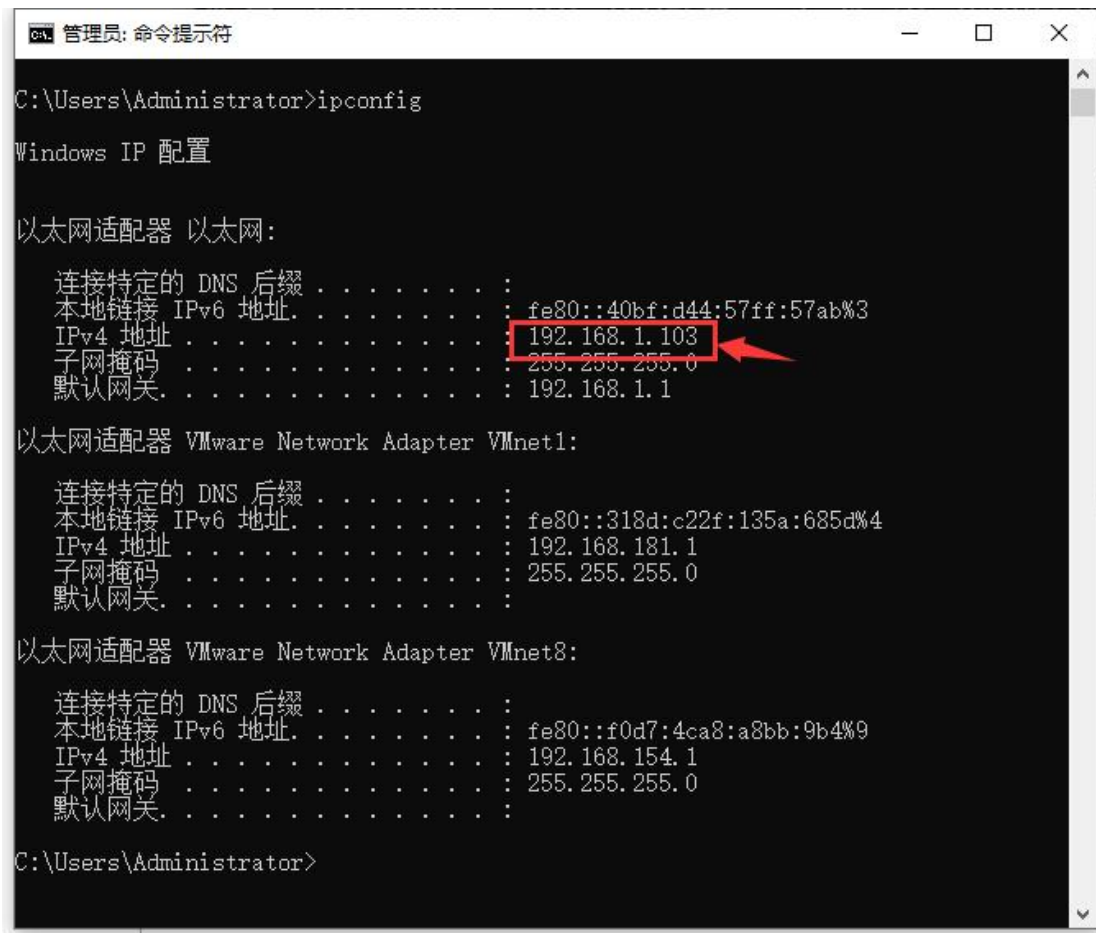
C:\Users\Administrator>
```

上图显示，电脑能正常 ping 通 DSP 网络。

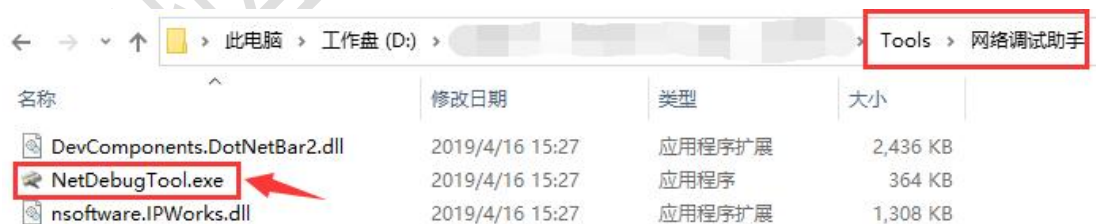
4.1.3.3.3 网络调试助手测试

使用网络调试助手进行 TCP、UDP 通信测试。

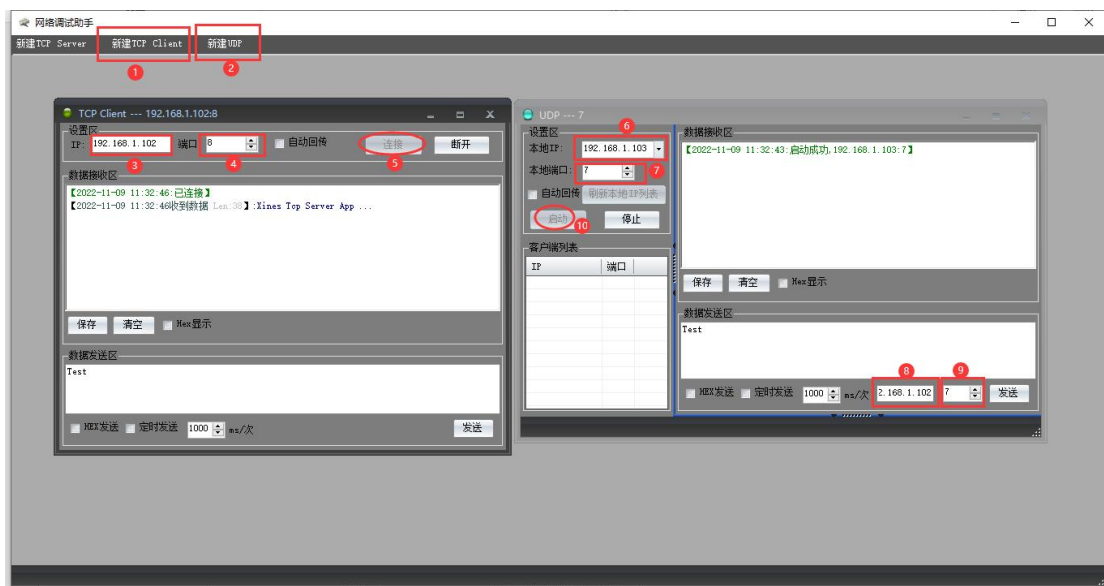
首先，在电脑命令行窗口使用 ipconfig 查看电脑的 IP 地址：



双击打开网络调试助手工具：

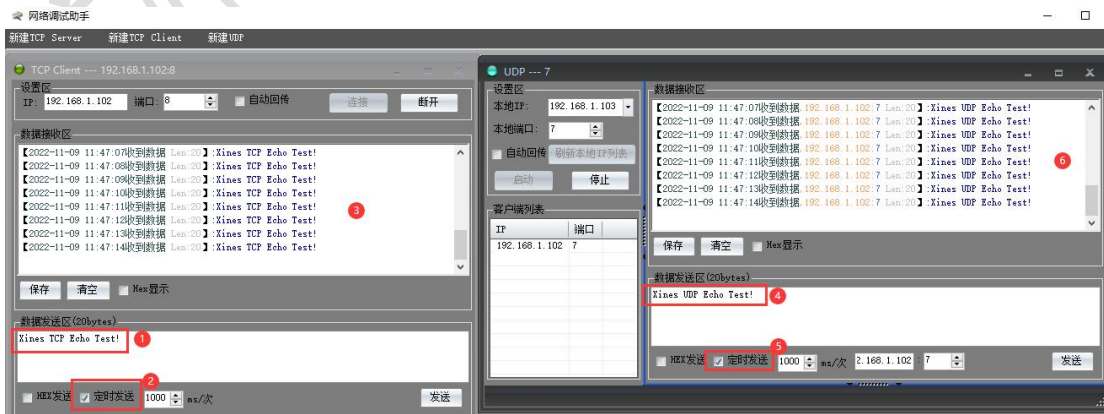


点击“新建 TCP Client “、”新建 UDP “，然后按照下面①~⑩顺序进行参数设置：



- ①：新建 TCP Client，新建 TCP 客户端窗口；
- ②：新建 UDP，新建 UDP 客户端窗口；
- ③：设置 TCP 主机 IP 地址，即 DSP 网络 IP 地址，通过 CCS 控制台窗口打印信息那里获取；
- ④：设置 TCP 主机 TCP 端口号，即 DSP 网络 TCP 端口号，DSP 程序里面固定设置为 8，因此这里需填写 8；
- ⑤：设置好 TCP 参数后，点击“连接”，连接 TCP 主机。如果长时间没有 TCP 通信，那么 TCP 客户端会自动断开与 TCP 主机的连接，那么需要用户再次点击“连接”按钮以便继续 TCP 通信；
- ⑥：设置电脑 IP 地址，通过在电脑命令行窗口输入 ipconfig 命令获取；
- ⑦：设置电脑 UDP 端口号，随意设置，只要窗口不弹出错误就行；
- ⑧：设置 DSP 网络 IP 地址，通过 CCS 控制台窗口打印信息那里获取；
- ⑨：设置 DSP 网络 UDP 端口号，DSP 程序里面固定设置为 7，因此这里需填写 7；
- ⑩：设置好 UDP 参数后，点击“启动”。

TCP、UDP 测试参数设置好后，下面开始 TCP、UDP 数据收发测试。在 TCP、UDP 测试窗口的数据发送区输入电脑网络发送数据，然后勾选上“定时发送”。如果 DSP 程序正常运行的话，那么会在数据接收区接收到 DSP 回显信息，即 DSP 将电脑发给它的网络数据原本返回，如下图所示：



4.1.3.4 退出实验

CCS 软件窗口上，点击 **Terminate** 断开 DSP 仿真器与板卡的连接。

关闭网络调试助手。

最后，关闭板卡电源，实验结束。

4.2 DSP UART0 串口通信

4.2.1 例程位置

DSP 例程保存在资料盘中的 Demo\DSP\XQ_UART_INT 文件夹下。

4.2.2 功能简介

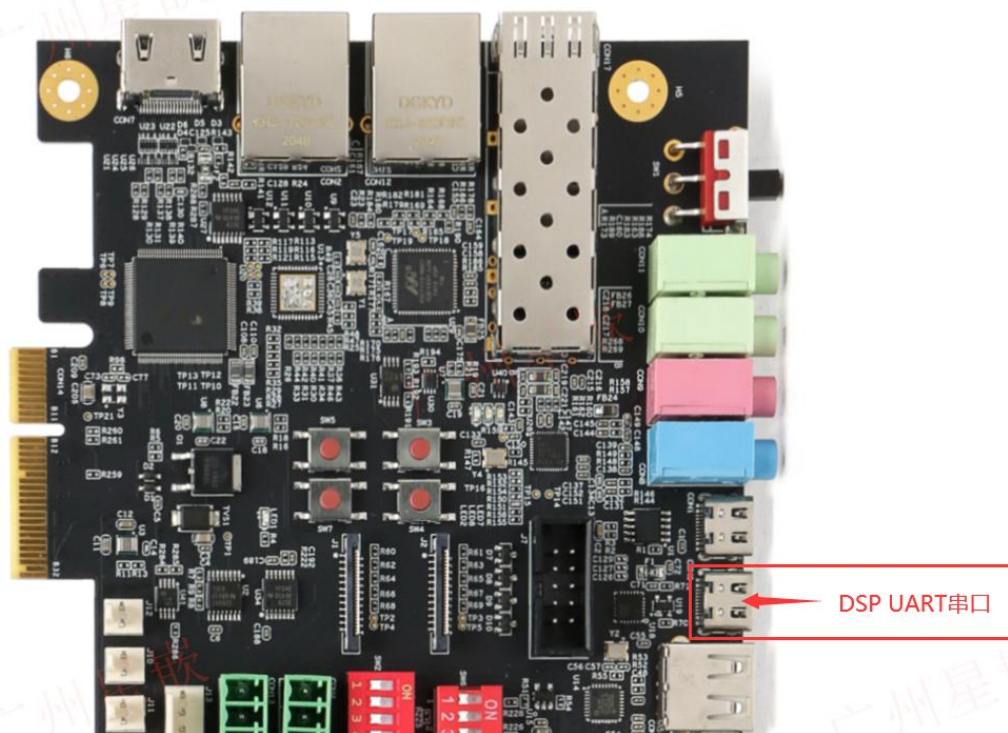
实现 DSP 串口通信测试功能。

硬件设计上，使用的是 UART0 串口。程序设计基于 UART 串口中断方式，实现串口数据回显功能。

4.2.3 例程使用

4.2.3.1 串口线缆连接

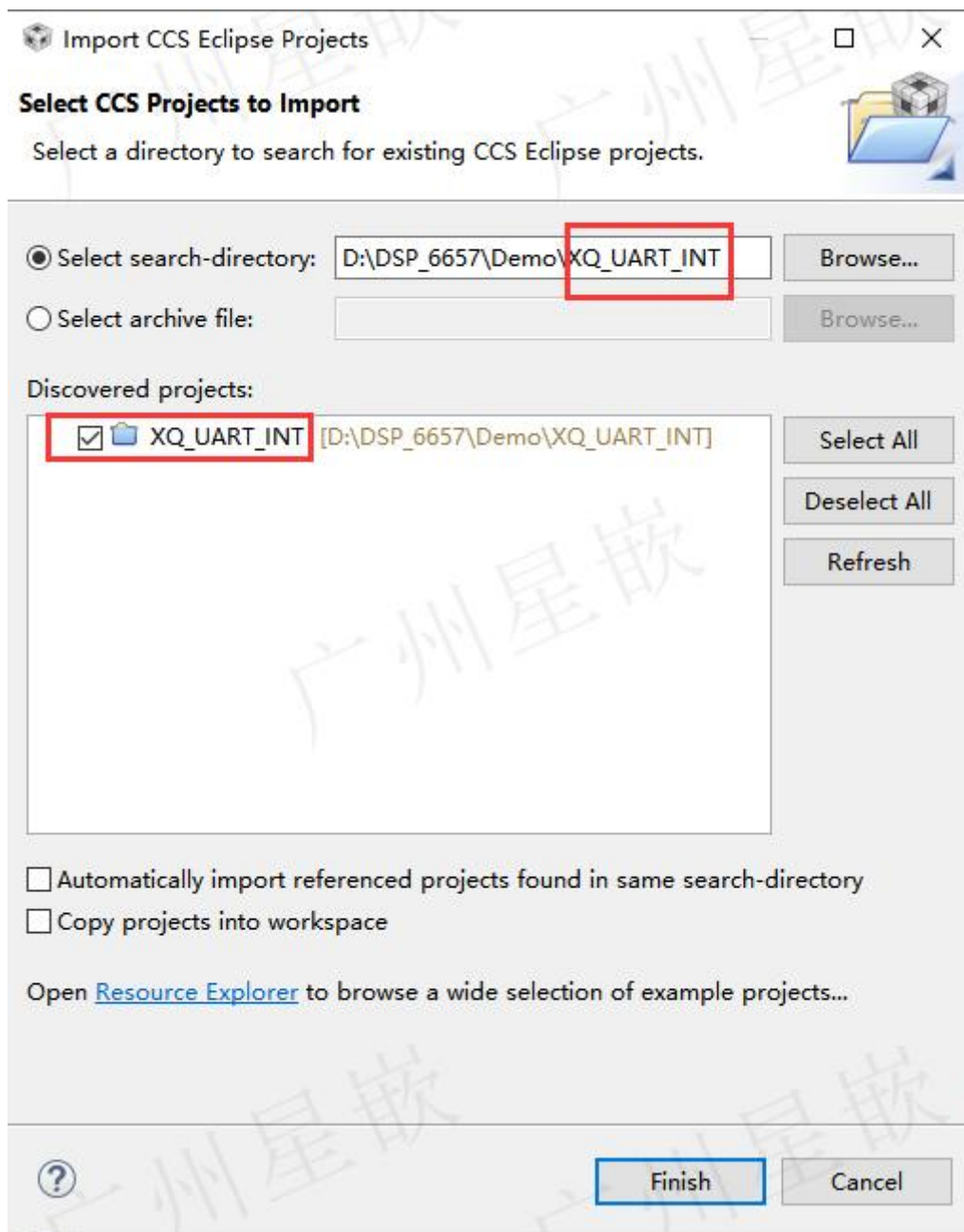
用 USB Type-C 线缆连接板卡上 DSP 串口至测试电脑：



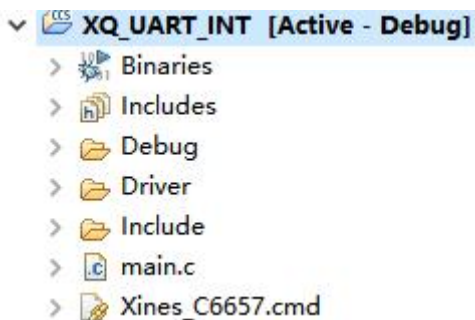
4.2.3.2 加载运行 DSP 程序

4.2.3.2.1 CCS 导入例程

CCS 软件导入示例工程 XQ_UART_INT，如下图所示：

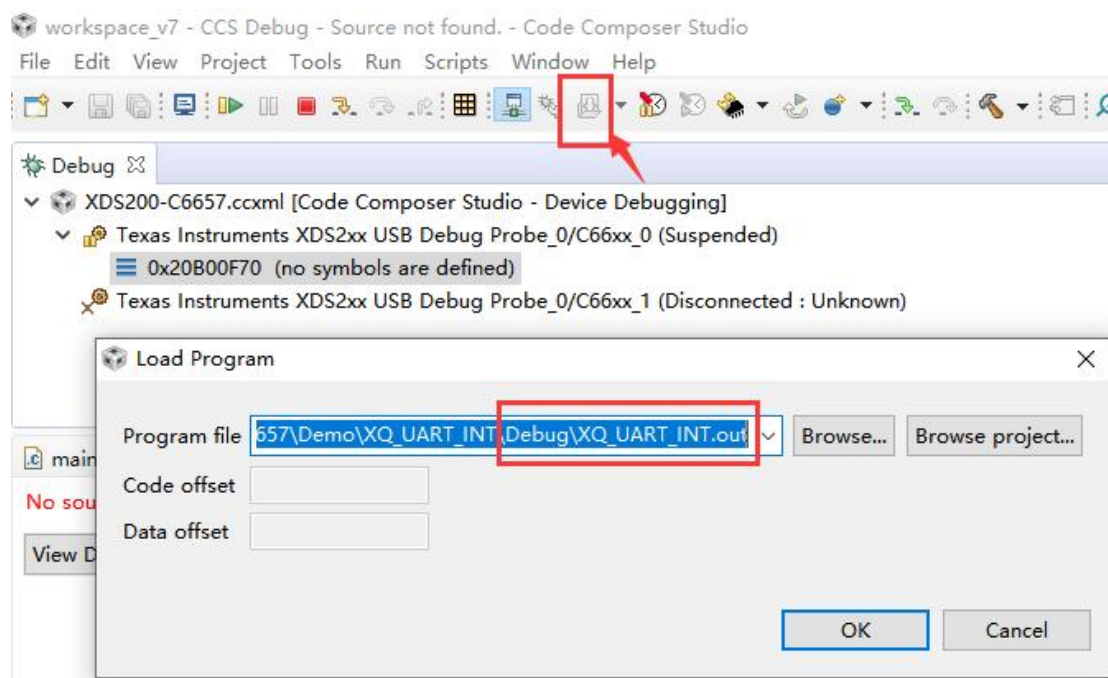


CCS 示例工程导入后界面如下图所示：

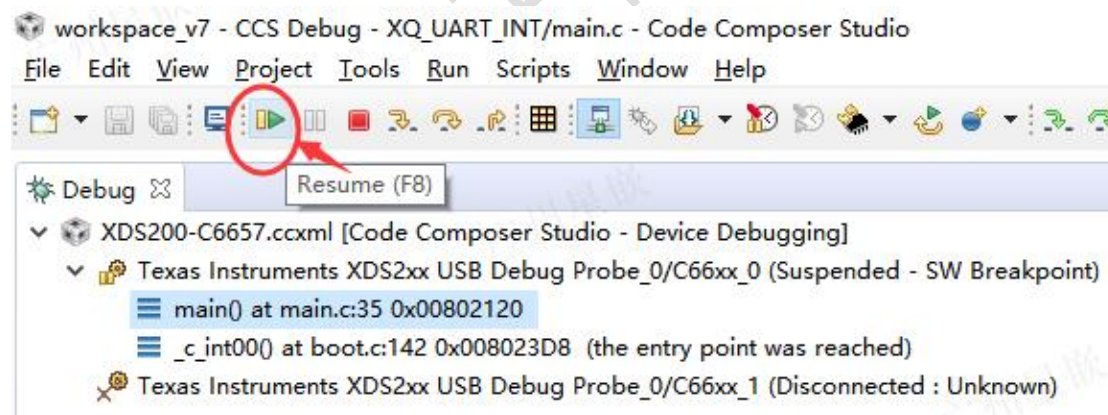


4.2.3.2 下载运行 CCS 程序

下载 DSP 可执行文件 XQ_UART_INT.out:

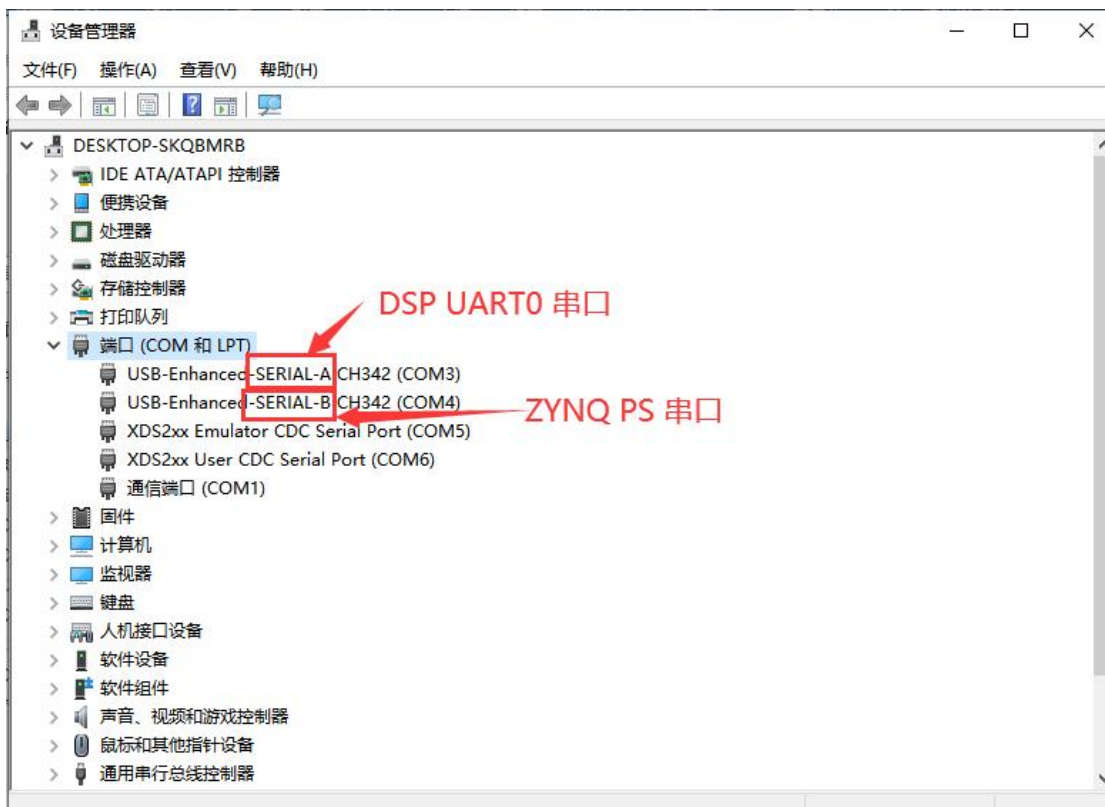


点击 Resume 运行 DSP 程序:

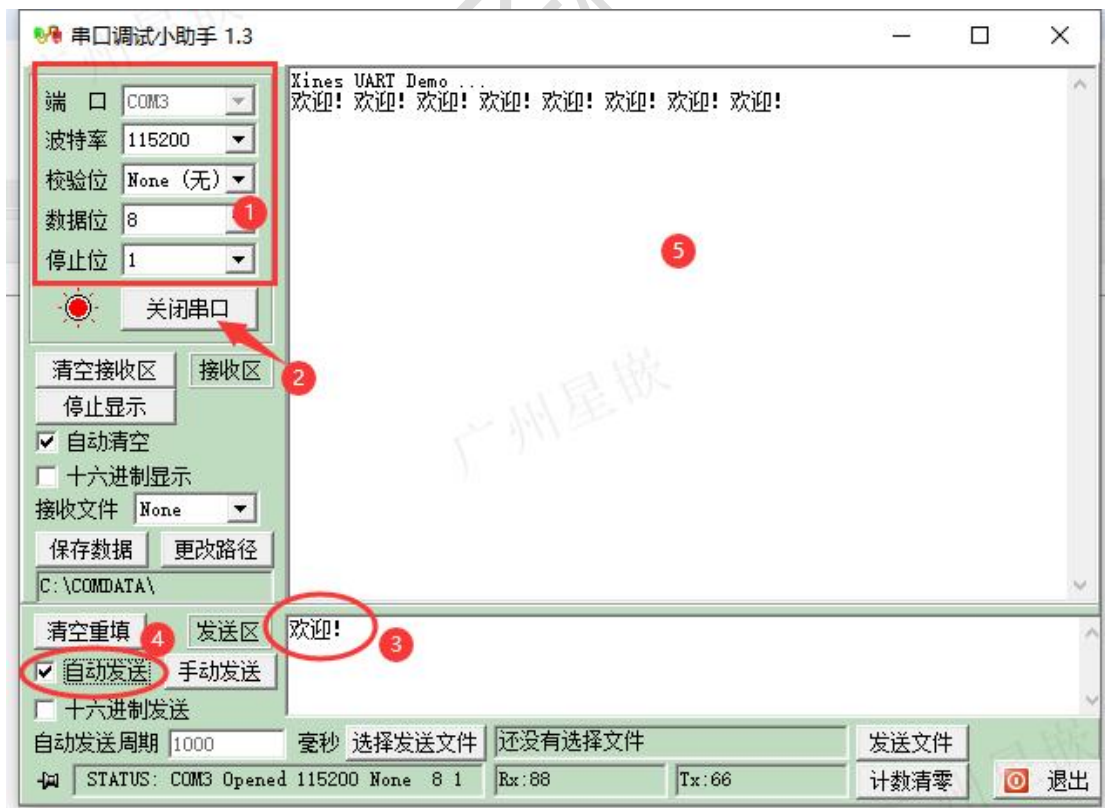


4.2.3.3 运行结果说明

板上电,并通过设备管理器查看串口号。带 SERIAL-A 字样的为 DSP 端串口,带 SERIAL-B 字样的为 ZYNQ PS 串口,记住后面的 COM 编号,这里 DSP 串口编号为 COM3:



在 DSP 程序运行之前，打开串口调试助手，并设置串口参数，具体设置如下图①区域标识所示：



设置完串口参数后，在②位置点击“打开串口”。接着，在③位置输入串口发送数据，并勾选上④位置处的“自动发送”，然后可以在⑤位置查看串口接收数据。串口调试助手将

串口数据发给 DSP，DSP 将接收到的串口数据原本返回给串口调试助手。

4.2.3.4 退出实验

- CCS 软件窗口上，点击 Terminate 断开 DSP 仿真器与板卡的连接。
- 关闭串口调试助手。
- 最后，关闭板卡电源，实验结束。

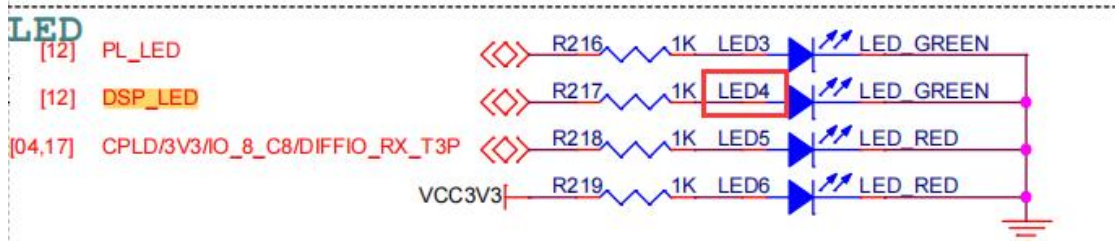
4.3 DSP GPIO LED 示例

4.3.1 例程位置

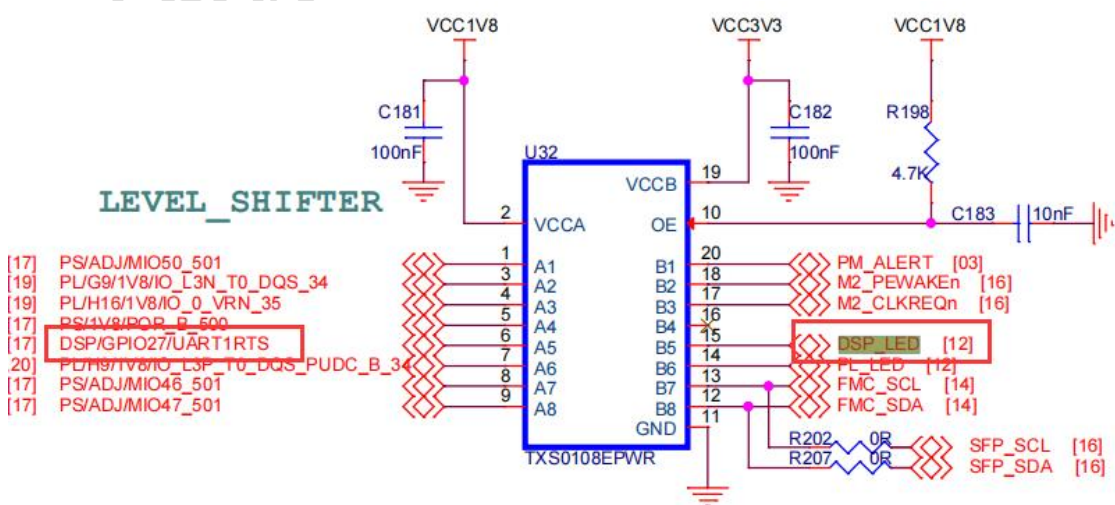
DSP 例程保存在资料盘中的 Demo\DSP\XQ_GPIO_LED 文件夹下。

4.3.2 功能简介

DSP 通过 GPIO 控制 LED 灯的亮、灭。具体控制的是底板上的 LED4 灯，原理图设计如下所示：



LED4 灯的控制信号 DSP_LED 通过一颗电平转换芯片连接到 DSP 的 GPIO27 上，原理图部分如下图所示：



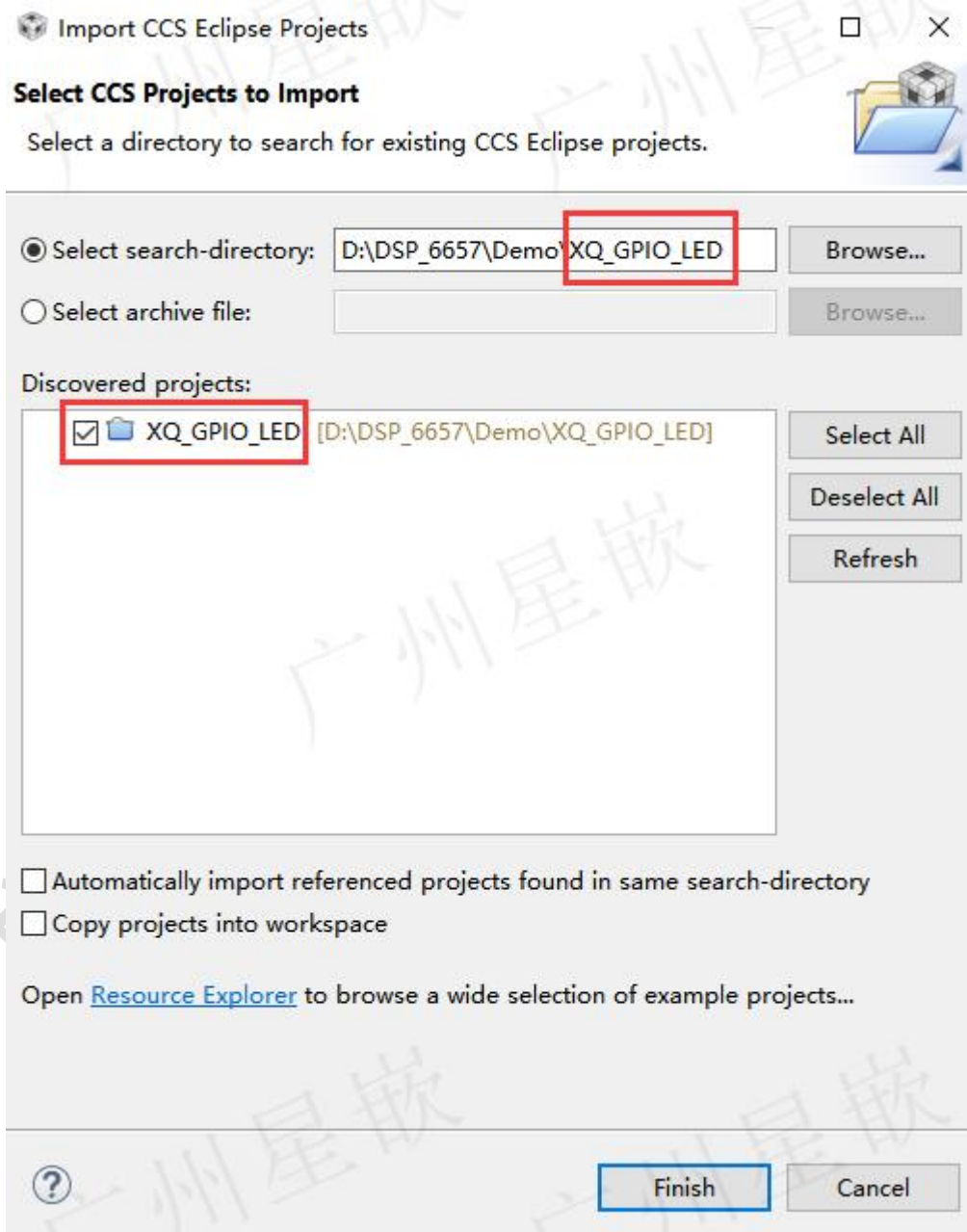
所以，DSP 程序设计时，通过控制 GPIO27 来实现对底板上 LED4 灯的亮、灭操作。

4.3.3 例程使用

4.3.3.1 加载运行 DSP 程序

4.3.3.1.1 CCS 导入例程

CCS 软件导入示例工程 XQ_GPIO_LED，如下图所示：

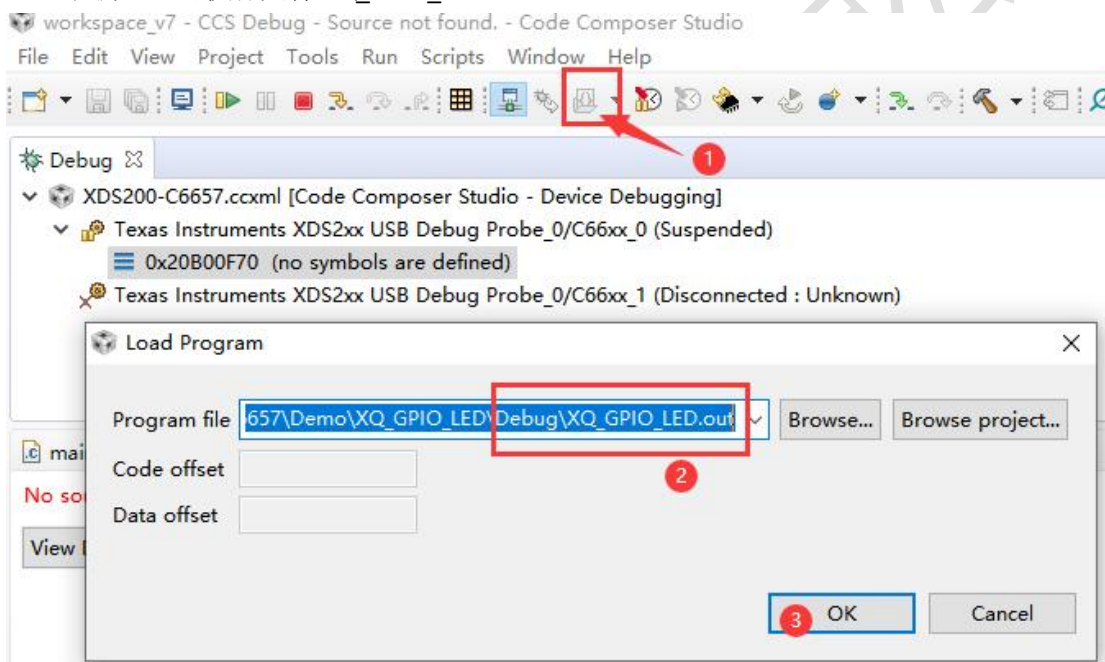


CCS 示例工程导入后界面如下图所示：

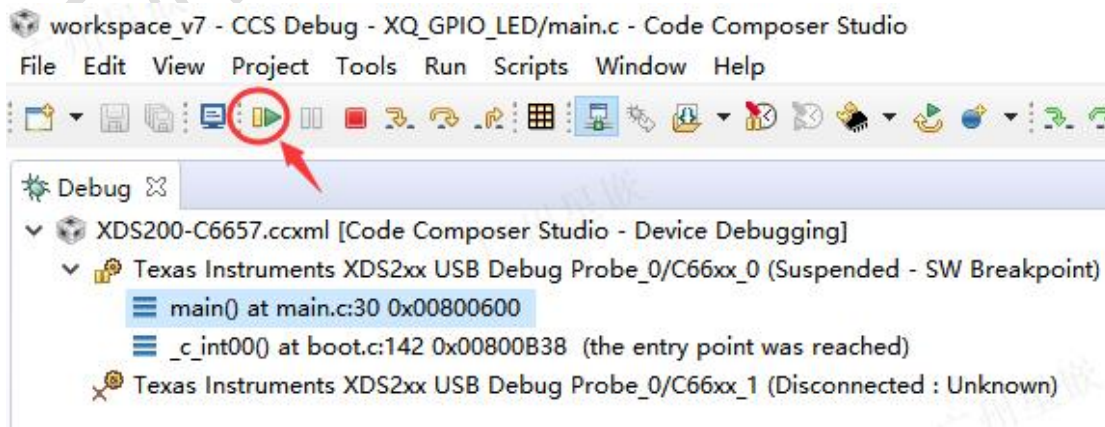
- ▼ CCS XQ_GPIO_LED [Active - Debug]
 - > Binaries
 - > Includes
 - > Debug
 - > Driver
 - > Include
 - > main.c
 - > Xines_C6657.cmd

4.3.3.1.2 下载运行 CCS 程序

下载 DSP 可执行文件 XQ_GPIO_LED.out:



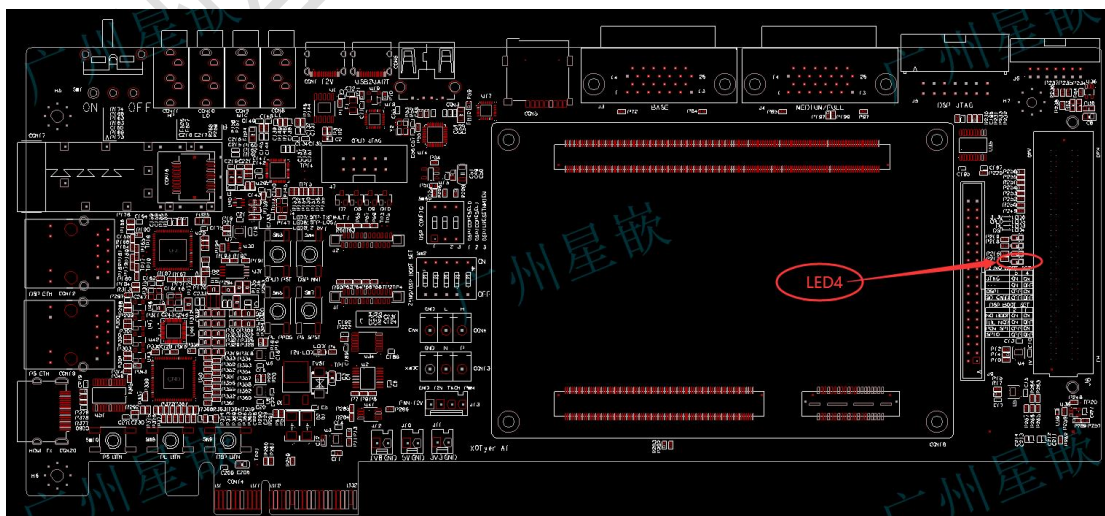
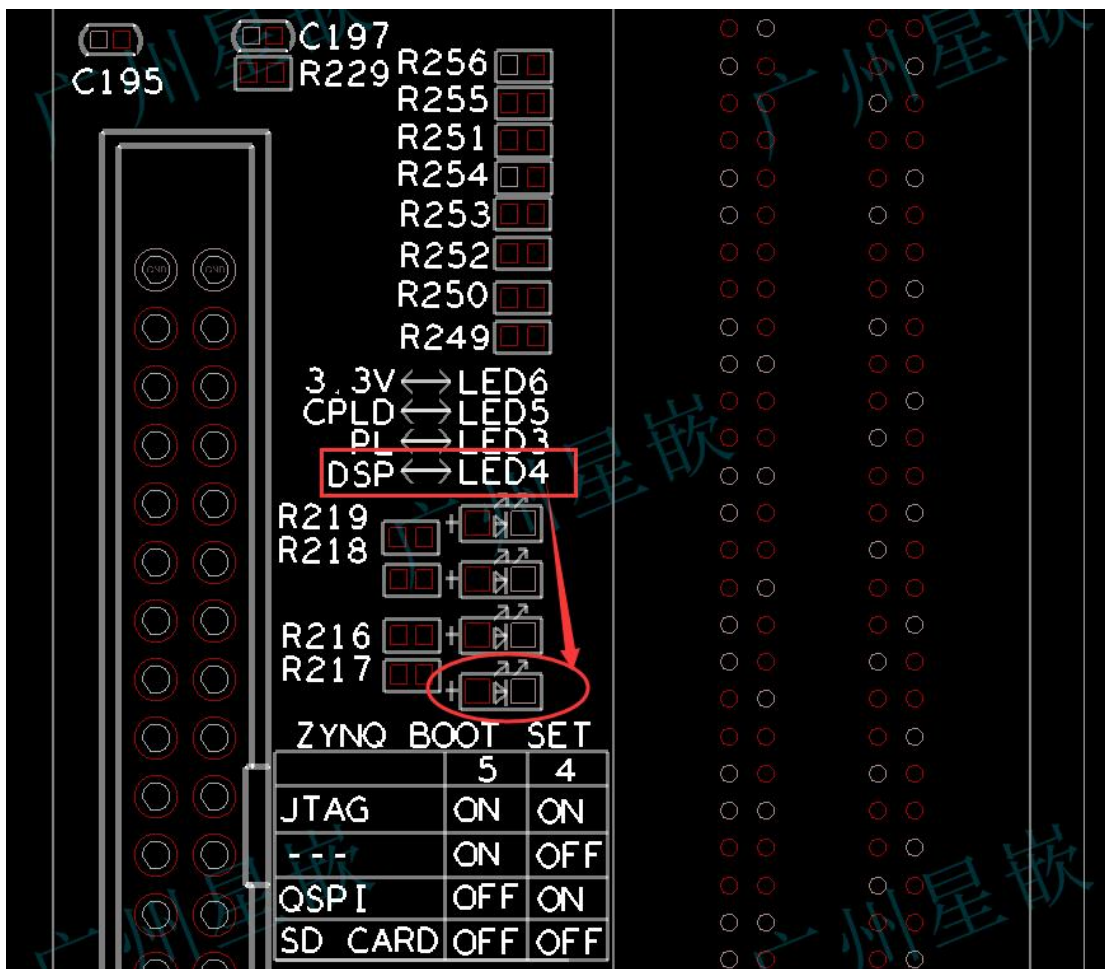
点击 Resume 运行 DSP 程序:



4.3.3.2 运行结果说明

DSP 程序运行起来后，如果程序正常运行的话，则可看到底板 LED4 灯会一亮一灭，快速闪烁。

LED4 灯在底板上的具体位置如下面两张图所示：



4.3.3.3 退出实验

CCS 软件窗口上，点击 Terminate 断开 DSP 仿真器与板卡的连接。
最后，关闭板卡电源，实验结束。

4.4 DSP DDR3 内存读写测试

4.4.1 例程位置

DSP 例程保存在资料盘中的 Demo\DSP\XQ_DDR3_Test 文件夹下。

4.4.2 功能简介

DSP 外挂 2 颗 DDR3 颗粒，总容量 1Gbytes，总位宽 32bits。

本实验通过两种方式测试 DDR3 的读写，分别是：EDMA 方式和 DSP 核心直接读写方式，两种方式的 DDR3 读写测试如下图程序代码段标注位置所示：

```
/* main函数 */
void main()
{
    int i;

    // TSC初始化，用于时间测量
    TSC_init();

    // EDMA初始化
    EDMA_init();

    /*make DDR cacheable and prefetchable*/
    for(i= 0; i< (DDREndAddress-DDRStartAddress)/16/1024/1024; i++)
        gpCGEM_regs->MAR[(DDRStartAddress/16/1024/1024)+i]=1 |
            (1<<CSL_CGEM_MAR0_PFX_SHIFT);

    // 设置缓存
    CACHE_setL1PSize(CACHE_L1_32KCACHE);
    CACHE_setL1DSize(CACHE_L1_32KCACHE);
    CACHE_setL2Size(CACHE_0KCACHE);

    tscl= TSCL;
    tsch= TSCH;
    printf("DDR3 Memory Test Start at %lld cycle\n", _itoll(tsch, tscl));

    // 以EDMA方式对DDR3内存进行读写测试
    KeyStone_memory_EDMA_test(DDRStartAddress, DDREndAddress, 1, "DDR3");

    // DSP核心直接对DDR3内存进行读写测试
    KeyStone_memory_test(DDRStartAddress, DDRStartAddress+512*1024, 1, "DDR3");

    tscl= TSCL;
    tsch= TSCH;
    printf("DDR3 Memory test complete at %lld cycle\n", _itoll(tsch, tscl));
}
```

EDMA 方式测试时，对整个 DDR3 地址空间（1GB）进行读写测试，EDMA 测试 DDR3 地

址空间设置代码如下代码段所示：

```
1 // DDR3测试起始地址和结束地址，地址空间1GB  
2 unsigned int DDRStartAddress = 0x80000000;  
3 unsigned int DDREndAddress= 0xC0000000;
```

DSP 核心直接方式读写 DDR3 测试时，只测试其中的 512KB 空间：

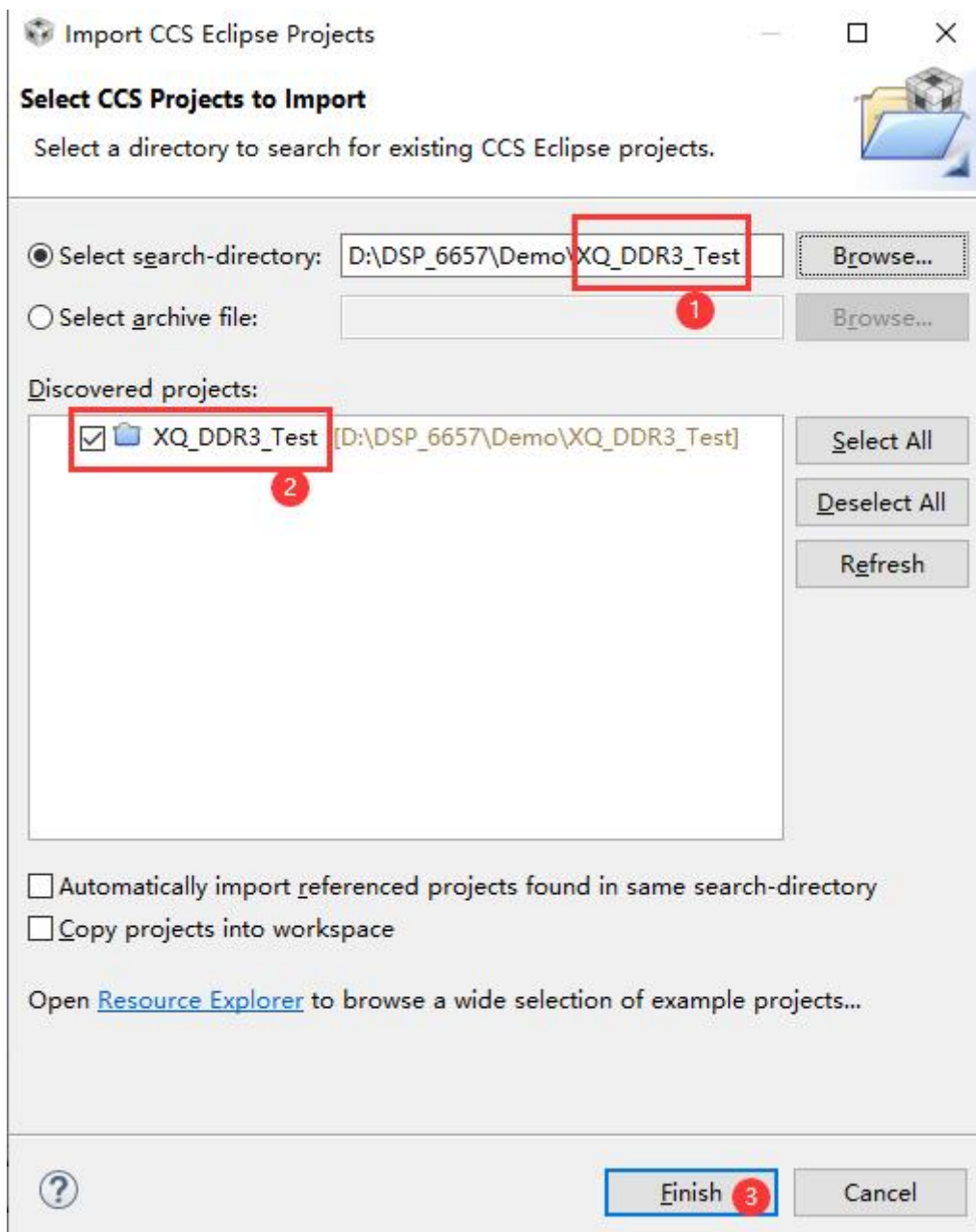
```
73 // DSP核心直接对DDR3内存进行读写测试  
74 Keystone_memory_test(DDRStartAddress, DDRStartAddress+512*1024, 1, "DDR3");  
75
```

4.4.3 例程使用

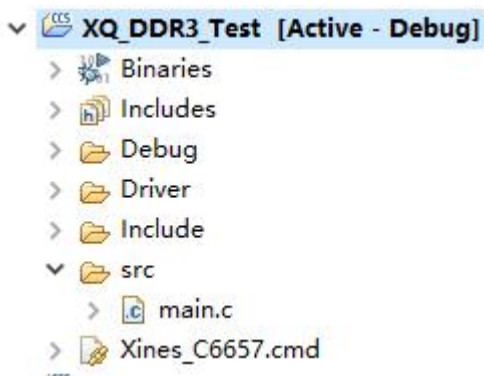
4.4.3.1 加载运行 DSP 程序

4.4.3.1.1 CCS 导入例程

CCS 软件导入示例工程 XQ_DDR3_Test，如下图所示：

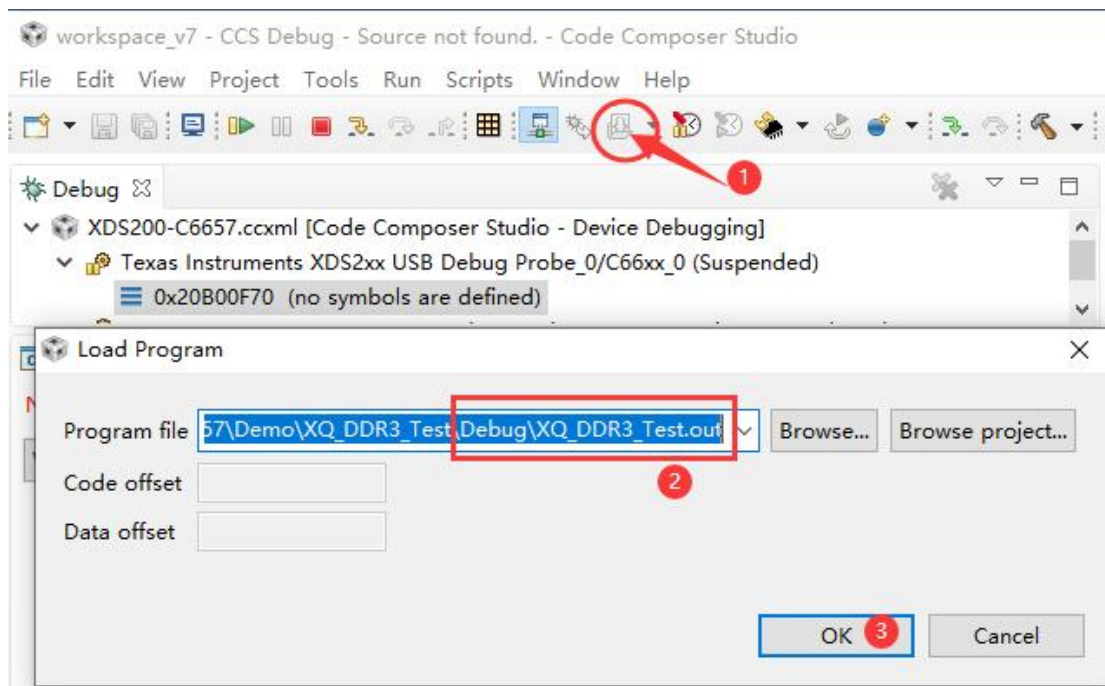


CCS 示例工程导入后界面如下图所示：

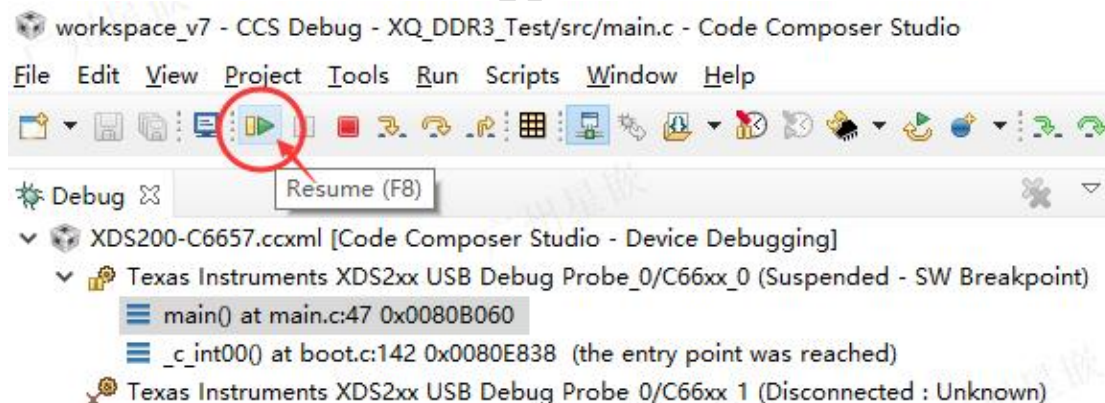


4.4.3.1.2 下载运行 CCS 程序

下载 DSP 可执行文件 XQ_DDR3_Test.out:



点击 Resume 运行 DSP 程序:



4.4.3.2 运行结果说明

首先以 EDMA 方式读写 DDR3 内存，往 DDR3 写测试数据，并读出对比。如果读写数据正确，则打印 Passed 字样，否则打印 Failed 字样，EDMA 方式测试通过后的打印信息如下图所示：

打印信息如下图所示：

```
DDR3 memory test at 503315890088 cycle
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x      0
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0xffffffffffffff
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0xaaaaaaaaaaaaaa
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x5555555555555555
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0xcccccccccccccccc
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x3333333333333333
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0xf0f0f0f0f0f0f0f0
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x f0f0f0f0f0f0f0f0
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0xff00ff00ff00ff00
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x ff00ff00ff00ff
Passed Memory Address Test from 0x80000000 to 0x80080000
Passed Memory Bit Walking from 0x80000000 to 0x80080000
Passed Times:      2      Failed Times:      0
DDR3 Memory test complete at 503755549709 cycle
```

4.4.3.3 退出实验

CCS 软件窗口上，点击 Terminate 断开 DSP 仿真器与板卡的连接。
最后，关闭板卡电源，实验结束。

4.5 DSP DDR3 初始化以及内存读写测试

4.5.1 例程位置

DSP 例程保存在资料盘中的 Demo\DSP\XQ_DDR3_INIT_TEST 文件夹下。

4.5.2 功能简介

DSP 外挂 2 颗 DDR3 颗粒，总容量 1Gbytes，总位宽 32bits。

本实验不需要使用 Gel 文件去对 PLL 和 DDR3 外设进行初始化配置，而是在用户代码中实现对 PLL 和 DDR3 外设的初始化配置工作。初始化完成后，通过两种方式测试 DDR3 内存的读写，分别是：EDMA 方式和 DSP 核心直接读写方式。

主函数代码如下图所示：

```

main.c
49 int main(void)
50 {
51     int i;
52
53     // TSC初始化, 用于时间测量
54     TSC_init();
55
56     // DSP core speed: 100*10/1=1000MHz
57     Keystone_main_PLL_init(100, 10, 1); ← PLL初始化
58
59     // XQTyer C6657 with 100MHz input for DDR
60     Keystone_DDR_init(100, 64, 6, NULL); ← DDR3控制器外设初始化
61
62     // EDMA初始化
63     EDMA_init();
64
65     /*make DDR cacheable and prefetchable*/
66     for(i= 0; i< (DDREndAddress-DDRStartAddress)/16/1024/1024; i++)
67         gpCGEM_regs->MAR[(DDRStartAddress/16/1024/1024)+i]=1|
68             (1<<CSL_CGEM_MAR0_PFX_SHIFT);
69
70     // 设置缓存
71     CACHE_setL1PSize(CACHE_L1_32KCACHE);
72     CACHE_setL1DSize(CACHE_L1_32KCACHE);
73     CACHE_setL2Size(CACHE_0KCACHE);
74
75     tsc1= TSCL;
76     tsch= TSCH;
77     printf("DDR3 Memory Test Start at %lld cycle\n", _itoll(tsch, tsc1));
78
79     // 以EDMA方式对DDR3内存进行读写测试          DDR3内存读写测试
80     Keystone_memory_EDMA_test(DDRStartAddress, DDREndAddress, 1, "DDR3");
81
82     // DSP核心直接对DDR3内存进行读写测试
83     Keystone_memory_test(DDRStartAddress, DDRStartAddress+512*1024, 1, "DDR3");
84
85     tsc1= TSCL;

```

EDMA 方式测试时, 对整个 DDR3 地址空间 (1GB) 进行读写测试, EDMA 测试 DDR3 地址空间设置代码如下代码段所示:

```

// DDR3测试起始地址和结束地址, 地址空间1GB
unsigned int DDRStartAddress = 0x80000000;
unsigned int DDREndAddress= 0xC0000000;

```

DSP 核心直接方式读写 DDR3 测试时, 只测试其中的 512KB 空间:

```

73 // DSP核心直接对DDR3内存进行读写测试
74 Keystone_memory_test(DDRStartAddress, DDRStartAddress+512*1024, 1, "DDR3");
75

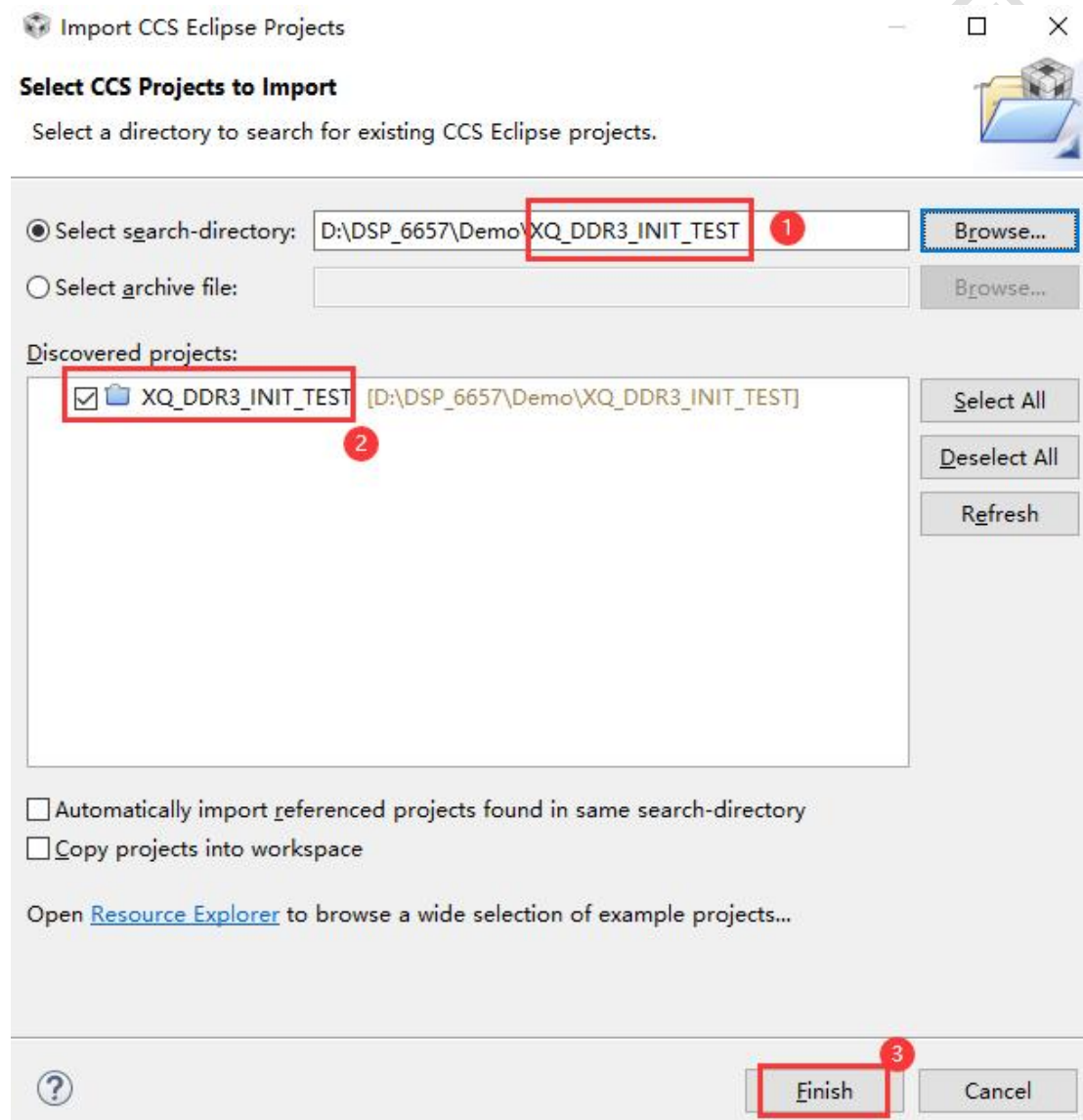
```

4.5.3 例程使用

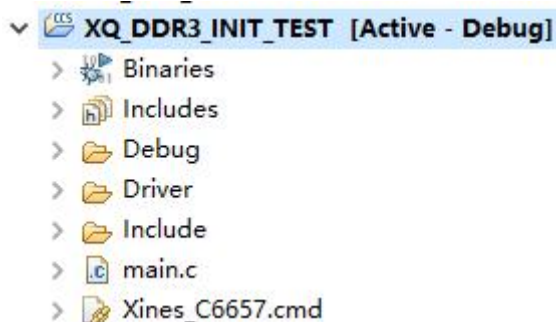
4.5.3.1 加载运行 DSP 程序

4.5.3.1.1 CCS 导入例程

CCS 软件导入示例工程 XQ_DDR3_INIT_TEST，如下图所示：

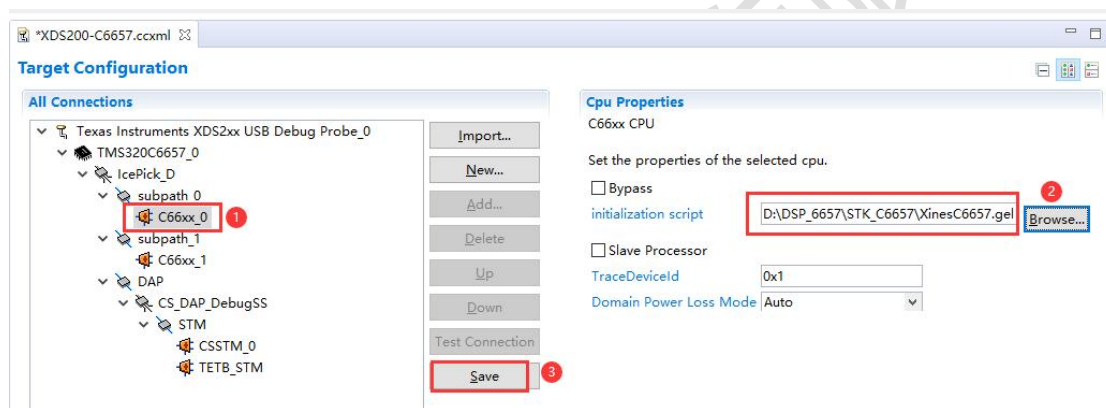


CCS 示例工程导入后界面如下图所示：

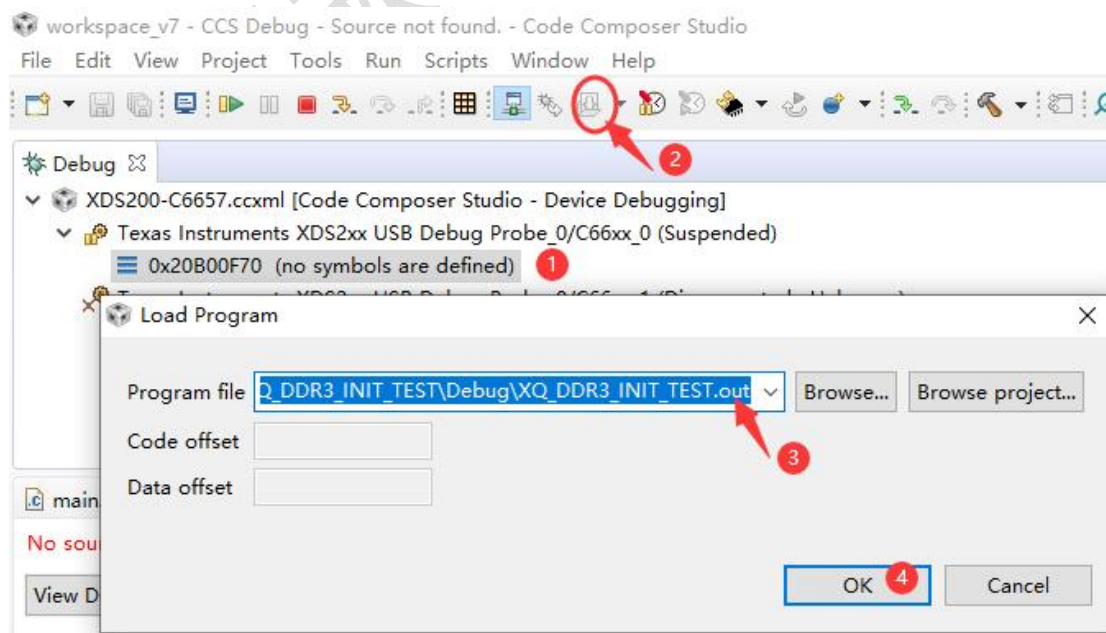


4.5.3.1.2 下载运行 CCS 程序

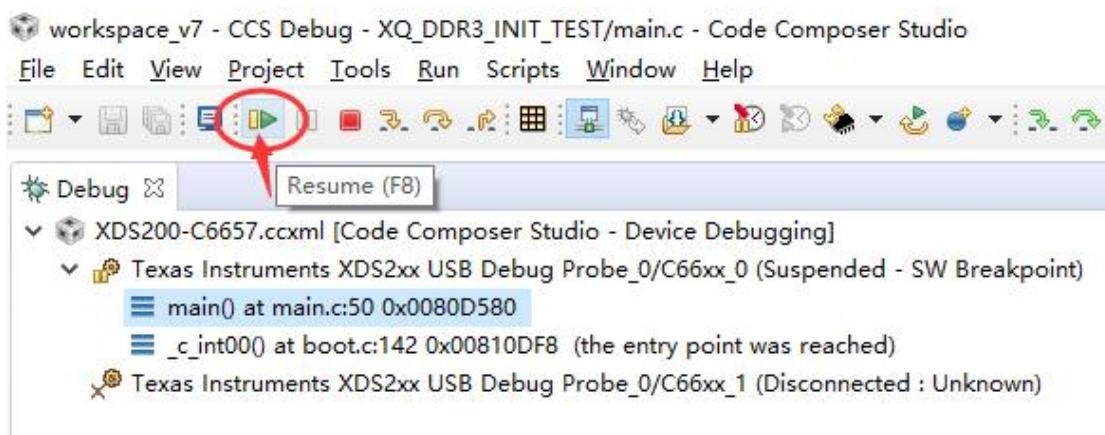
特别说明：本实验可以使用 Gel 文件，也可以不使用 Gel 文件。Gel 文件的设置在 .ccxml 目标配置文件中，如下图②位置所示的 .ccxml 目标配置文件中 Gel 文件设置，可以为空，也可以指定 Gel 文件位置：



下载 DSP 可执行文件 XQ_DDR3_INIT_TEST.out:



点击 Resume 运行 DSP 程序:



4.5.3.2 运行结果说明

首先以 EDMA 方式读写 DDR3 内存，往 DDR3 写测试数据，并读出对比。如果读写数据正确，则打印 Passed 字样，否则打印 Failed 字样，EDMA 方式测试通过后的打印信息如下图所示：

```

Console
XDS200-C6657.ccxml:CI0
[C66xx_0] Initialize DDR speed = 100.00MHzx64/6 = 1066.667MTS
DDR3 Memory Test Start at 11724702 cycle

DDR3 memory test with EDMA at 11734709 cycle
Passed Memory Fill Test from 0x80000000 to 0xc0000000 with pattern 0x          0 with EDMA CC0 TC0
Passed Memory Fill Test from 0x80000000 to 0xc0000000 with pattern 0xfffffffffffffff with EDMA CC0 TC1
Passed Memory Fill Test from 0x80000000 to 0xc0000000 with pattern 0xaaaaaaaaaaaaaaaa with EDMA CC0 TC2
Passed Memory Fill Test from 0x80000000 to 0xc0000000 with pattern 0x5555555555555555 with EDMA CC0 TC3
Passed Memory Address Test from 0x80000000 to 0xc0000000 with EDMA CC0 TC0
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x          100000001
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xfffffffffffffffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x          200000002
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xfffffffffffffffff
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x          400000004
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xffffffffbfffffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x          800000008
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xffffffff7fffffff
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x          100000010
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xfffffffffffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x          200000020
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xffffffffdfffffff
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x          400000040
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xffffffffbfffffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x          800000080
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xffffffff7fffffff
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x          1000000100
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xffffefffffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x          2000000200
    
```

```

Console X
XDS200-C6657.ccxml:CIO
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xffffeffffffffeffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x 2000000020000
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xffffdffffffdfdfdf
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x 4000000040000
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xffffbfffffffbfffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x 8000000080000
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xffff7ffffff7fffff
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x 10000000100000
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xffeffffffffeffffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x 20000000200000
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xffdfdfdfdfdfdfdf
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x 40000000400000
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xfffbfffffffbfffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x 80000000800000
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xff77ffffff7fffff
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x 100000001000000
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xfefffffffeffffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x 200000002000000
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xfdffffffdfdfdfdf
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x 400000004000000
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xfbfffffffbfffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0xf77ffffff7fffff
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0x1000000010000000
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0xefffffffeffffff
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0x2000000020000000
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0xdffffffdfdfdfdf
Memory Bit Walking with EDMA CC0 TC1 at 0x80000000 with pattern 0x4000000040000000
Memory Bit Walking with EDMA CC0 TC2 at 0x80000000 with pattern 0xbfffffffbfffff
Memory Bit Walking with EDMA CC0 TC3 at 0x80000000 with pattern 0x8000000080000000
Memory Bit Walking with EDMA CC0 TC0 at 0x80000000 with pattern 0x7ffffff7fffff
Passed Memory Bit Walking from 0x80000000 to 0xc0000000 with DMA
Passed Times: 1 Failed Times: 0
  
```

接着，以 DSP 核心直接方式读写 DDR3 内存，往 DDR3 写测试数据，并读出对比。如果读写数据正确，则打印 Passed 字样，否则打印 Failed 字样，DSP 核心直接方式测试通过后的打印信息如下图所示：

```

DDR3 memory test at 470786837771 cycle
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x 0
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0xffffeffffffffeffff
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0xaaaaaaaaaaaaaaaa
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x5555555555555555
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x31cccccccccccc
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x3333333330363630
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x 4f0f0f0f0
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x 81bbc900000000
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0xffff00ff00ff00
Passed Memory Fill Test from 0x80000000 to 0x80080000 with pattern 0x 80f3a000000001
Passed Memory Address Test from 0x80000000 to 0x80080000
Passed Memory Bit Walking from 0x80000000 to 0x80080000
Passed Times: 2 Failed Times: 0
DDR3 Memory test complete at 471225759895 cycle
  
```

4.5.3.3 退出实验

CCS 软件窗口上，点击 Terminate 断开 DSP 仿真器与板卡的连接。
最后，关闭板卡电源，实验结束。

4.6 DSP IPC 核间通信之 NOTIFY

4.6.1 例程位置

DSP 例程保存在资料盘中的 Demo\DSP\XQ_IPC_NOTIFY 文件夹下。

4.6.2 功能简介

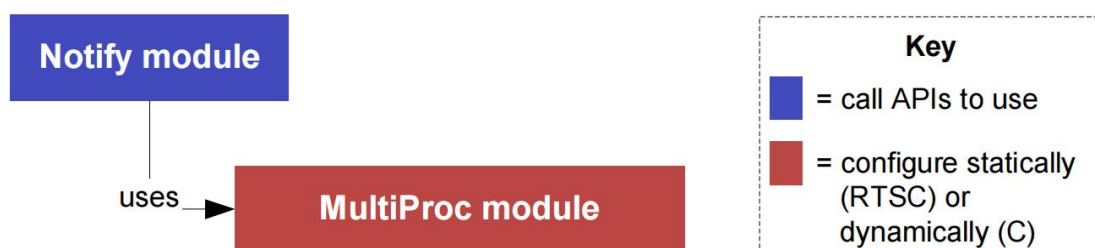
IPC 组件可用于处理器多核之间的核间通信，通信内容可包含消息传递、数据流以及链表等，IPC 组件的运行是建立在 SYS/BIOS 基础之上的。

根据难易程度，IPC 有如下几种使用场景：

- 1) IPC 最简化使用场景：该场景下，处理器核心之间只运行通知机制，即 Notify 通知。Notify 通知可携带 32bits 的用户数据。Notify 通知事件可用于多个处理器核心之间的简单同步。
- 2) 数据传输使用场景：相比于 IPC 最简化使用场景，该场景增加了处理器核心之间链表元素传递功能。数据传输场景下，可通过共享内存或门来辅助实现处理器核心之间的同步。
- 3) 动态链表分配使用场景：该场景下，可从 heap 中动态添加或删除链表元素。
- 4) MessageQ 消息传递使用场景：使用 MessageQ 模块在处理器核心之间传递消息。

本节实验内容使用 Notify 模块实现两个 DSP 核心之间的简单通知和 32bits 数据传递功能。

Notify 通知需要用到两个模块：Notify module 和 MultiProc module，其中 Notify module 提供 API 供用户调用，Notify module 需要使用 MultiProc module 中配置的属性，MultiProc module 可静态配置或动态配置：



本实验两个 DSP 核心使用同一个程序镜像，通过核心号区分哪些代码运行在核心 0，哪些代码运行在核心 1。首先，需要调用 `ipc_start()` 函数，与其他核心进行同步；然后注册 Notify 通知事件，Notify 事件注册时需要关联核心号（即本核心需要响应哪个核心发过来的 Notify 事件）、中断线、事件 ID 以及 Notify 事件回调函数等参数，具体实现如下 main 函数里面的代码段所示：

```

Int main(Int argc, Char* argv[])
{
    Int status;
    UInt numProcs = MultiProc_getNumProcessors();

    /*
     * Determine which processors Notify will communicate with based on the
     * local MultiProc id. Also, create a processor-specific Task.
     */
    srcProc = ((MultiProc_self() - 1 + numProcs) % numProcs);
    dstProc = ((MultiProc_self() + 1) % numProcs);

    System_printf("main: MultiProc id = %d\n", MultiProc_self());
    System_printf("main: MultiProc name = %s\n",
        MultiProc_getName(MultiProc_self()));

    /*
     * Ipc_start() calls Ipc_attach() to synchronize all remote processors
     * because 'Ipc.procSync' is set to 'Ipc.ProcSync_ALL' in *.cfg
     */
    status = Ipc_start(); ① 与远程处理器核心进行同步
    if (status < 0) {
        System_abort("Ipc_start failed\n");
    }

    /*
     * Register call back with Notify. It will be called when the processor
     * with id = srcProc sends event number EVENTID to this processor.
     */ ② 注册Notify事件
    status = Notify_registerEvent(srcProc, INTERRUPT_LINE, EVENTID,
        (Notify_FnNotifyCbck)cbFxn, NULL);
    if (status < 0) {
        System_abort("Notify_registerEvent failed\n");
    }

    BIOS_start();
}

```

本核心需要响应的远程处理器核心编号
 Notify中断信号线编号
 Notify事件ID号
 Notify事件回调函数

Notify 中断信号线、Notify 事件 ID 号、远程处理器核心编号三者共同决定 Notify 事件的响应。如果远程处理器核心发给本地处理器核心的 Notify 事件中的 Notify 中断信号线、Notify 事件 ID 号、远程处理器核心编号均与本地注册的 Notify 事件参数匹配，那么本地核心处理器就会响应远程发过来的 Notify 事件，调用 Notify 事件回调函数执行。

本实验的 Notify 事件回调函数代码实现如下：

```

Void cbFxn(UInt16 procId, UInt16 lineId,
           UInt32 eventId, UArg arg, UInt32 payload)
{
    /* The payload is a sequence number. */
    recvProcId = procId;
    recvPayload = payload;
    Semaphore_post(semHandle);
}

```

Notify 事件到来后，如果远程处理器核心编号、Notify 中断信号线、Notify 事件 ID 号三个参数与本地注册的 Notify 事件参数匹配（对应回调函数中 procId、lineId、eventId 三个形参），那么 Notify 事件回调函数开始执行，记录远程处理器核心编号和 32bits 的附带数据，然后 post 信号量 semHandle。

程序里面静态创建了一个任务 tsk0，对应的任务函数为 tsk0_func，tsk0_func 中核心 0 代码实现部分如下图所示：

```
Void tsk0_func(UArg arg0, UArg arg1)
{
    Int i = 1;
    Int status;

    if (MultiProc_self() == 0) { // 核心0运行程序
        while (i < NUMLOOPS) {
            /* Send an event to the next processor */
            // ipc notify发送至ID号为dspProc的处理器, ipc notify附带发送的payload数据为i
            status = Notify_sendEvent(dstProc, INTERRUPT_LINE, EVENTID, i,
                                     TRUE);

            /* Continue until remote side is up */
            if (status < 0) {
                continue;
            }

            System_printf("Sent ipc notify to %s with payload = %d\n",
                          MultiProc_getName(dstProc), i);

            /* Wait to be released by the cbFxn posting the semaphore */
            // 等待核心1 post 信号量
            Semaphore_pend(semHandle, BIOS_WAIT_FOREVER);

            System_printf("Received ipc notify from %s with payload = %d\n",
                          MultiProc_getName(recvProcId), recvPayload);

            /* increment for next iteration */
            i++;
        }
    }
    else { // 核心1运行程序
```

核心 0 调用 `Notify_sendEvent` 函数，往核心 1 发送 `Notify` 事件，附带 32bits 数据，数据内容为 `i`，即循环次数。然后，核心 0 等待核心 1 发布信号量 `semHandle`，一旦等到核心 1 发布信号量 `semHandle`，则进入下一次循环，如此循环执行，直到指定循环次数结束。

`tsk0_func` 中核心 1 代码实现部分如下图所示：

```
else { // 核心1运行程序
    while (i < NUMLOOPS) {
        /* wait forever on a semaphore, semaphore is posted in callback */
        // 等待核心0 post 信号量
        Semaphore_pend(semHandle, BIOS_WAIT_FOREVER);

        System_printf("Received ipc notify from %s with payload = %d\n",
            MultiProc_getName(recvProcId), recvPayload);

        /* Send an event to the next processor */
        // ipc notify发送至ID号为dspProc的处理器, ipc notify附带发送的payload数据为i+100
        status = Notify_sendEvent(dstProc, INTERRUPT_LINE, EVENTID, i+100,
            TRUE);
        if (status < 0) {
            System_abort("sendEvent failed\n");
        }

        System_printf("Sent ipc notify to %s with payload = %d\n",
            MultiProc_getName(dstProc), i+100);

        /* increment for next iteration */
        i++;
    }
}

System_printf("Test completed\n");
BIOS_exit(0);
}
```

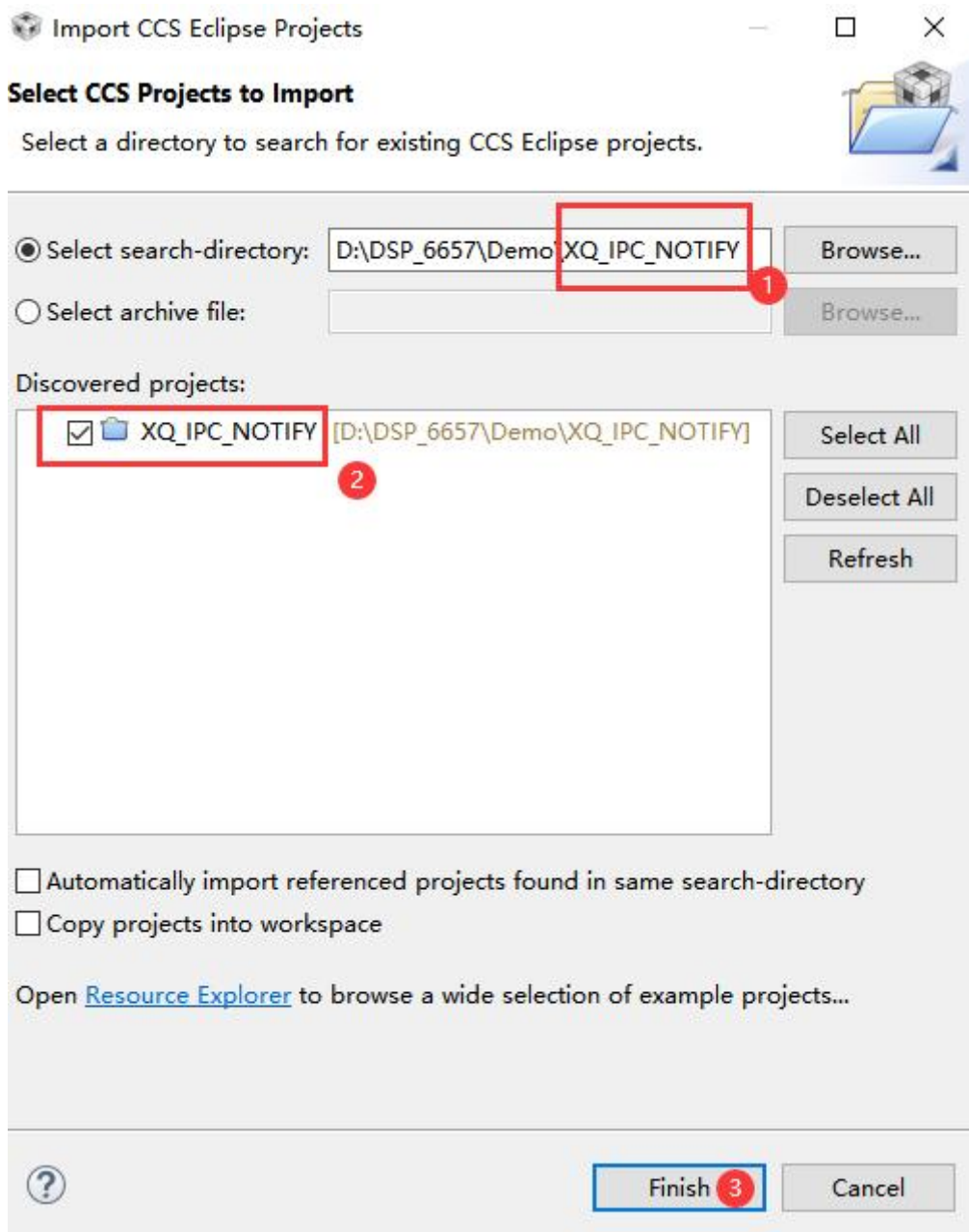
首先,核心 1 等待核心 0 发布信号量 semHandle,一旦等到核心 0 发布信号量 semHandle,核心 1 调用 Notify_sendEvent 函数,往核心 0 发送 Notify 事件,附带 32bits 数据,数据内容为 i+100,即循环次数+100。然后,进入下一次循环,如此循环执行,直到指定循环次数结束。

4.6.3 例程使用

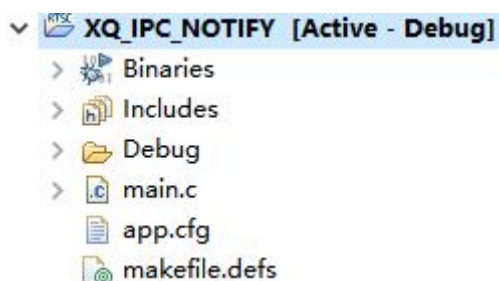
4.6.3.1 加载运行 DSP 程序

4.6.3.1.1 CCS 导入例程

CCS 软件导入示例工程 XQ_IPC_NOTIFY, 如下图所示:



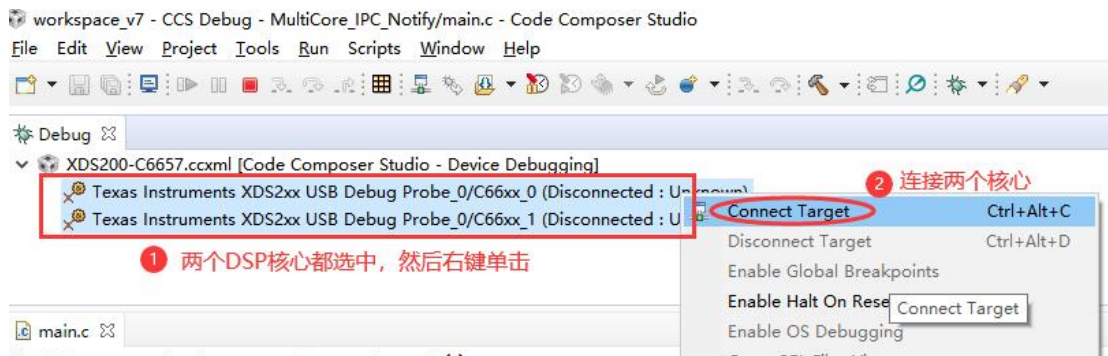
CCS 示例工程导入后界面如下图所示：



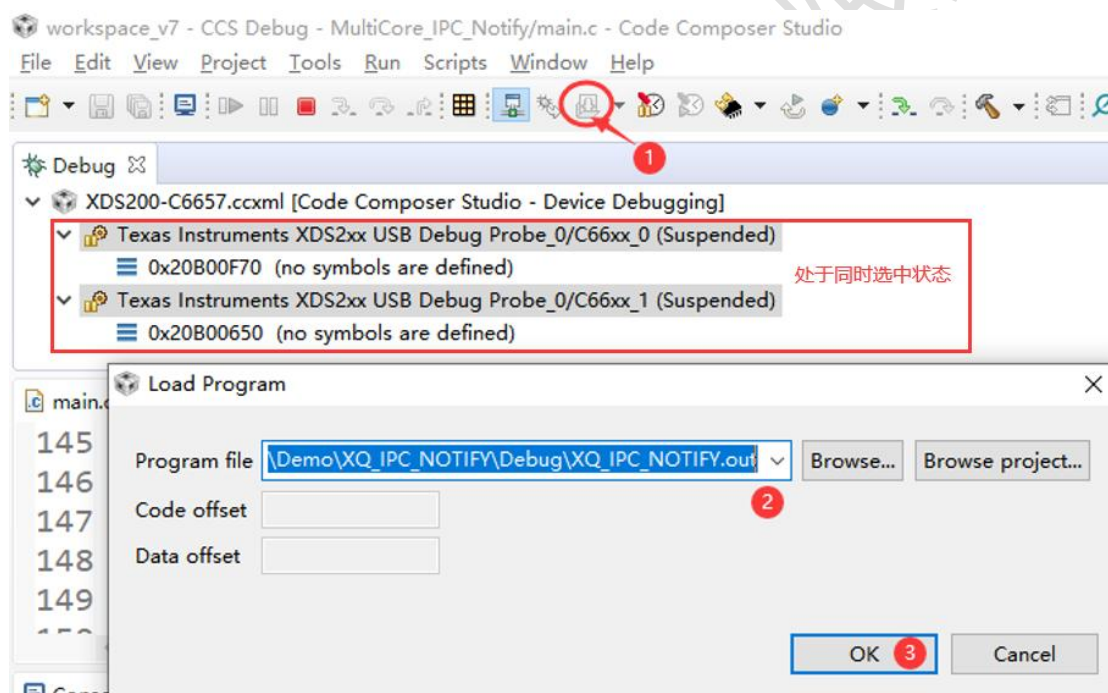
4.6.3.1.2 下载运行 CCS 程序

此实验需要连接 DSP 的两个核心，并给这两个 DSP 核心下载同一个镜像文件。

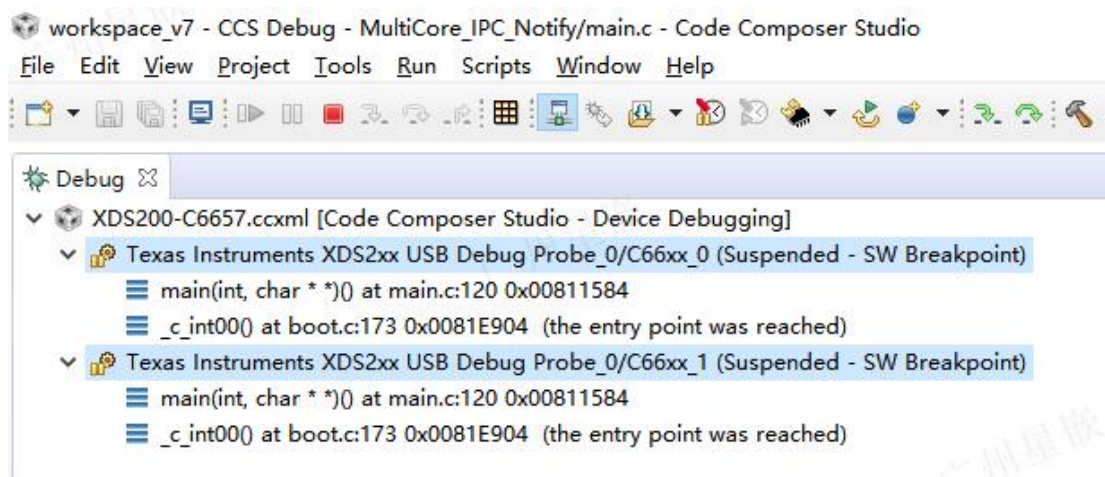
首先，连接两个 DSP 核心：



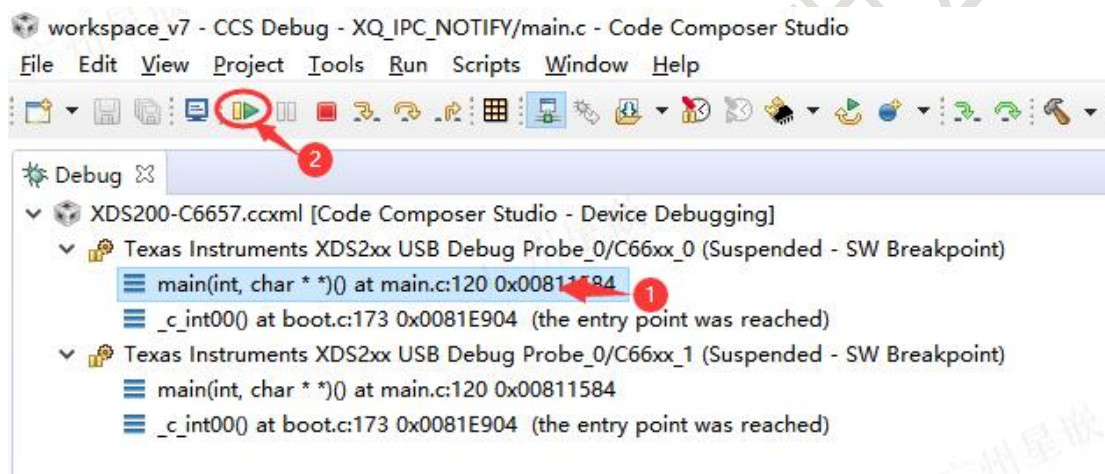
其次，两个 DSP 核心连接上后，给两个 DSP 核心下载同一个镜像文件 XQ_IPC_NOTIFY.out：



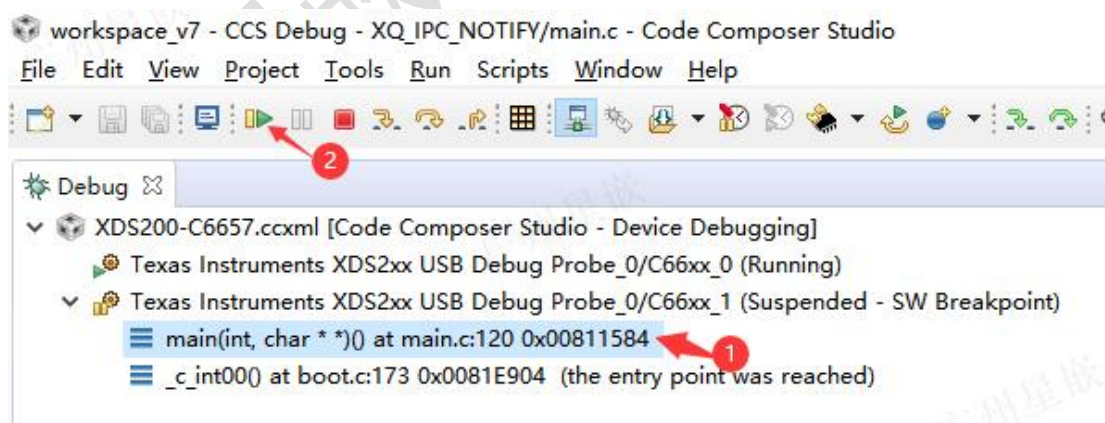
两个 DSP 核心均下载完 DSP 可执行镜像文件 XQ_IPC_NOTIFY.out 后的界面如下图所示：



选中核心 0 的 main 函数位置，点击 Resume 运行 DSP 核心 0 的程序：



选中核心 1 的 main 函数位置，点击 Resume 运行 DSP 核心 1 的程序：



4.6.3.2 运行结果说明

CCS Console 窗口打印程序运行结果。下面，首先对部分打印信息进行解读，然后附带给出程序运行产生的所有打印信息。

部分打印信息解读:

核心 0 给核心 1 发送 Notify 通知事件, 通知事件附带用户数据 1, Console 打印信息如下示例所示:

```
[C66xx_0] Sent ipc notify to CORE1 with payload = 1
```

核心 1 收到核心 0 发过来的 Notify 通知事件后, 将收到的附带数据以及谁发过来的这些信息也打印出来, 如下示例所示:

```
[C66xx_1] main: MultiProc id = 1  
main: MultiProc name = CORE1  
Received ipc notify from CORE0 with payload = 1  
Sent ipc notify to CORE0 with payload = 101
```

核心 1 给核心 0 发送 Notify 通知事件, 通知事件附带用户数据 101, Console 打印信息如下示例所示:

```
[C66xx_1] main: MultiProc id = 1  
main: MultiProc name = CORE1  
Received ipc notify from CORE0 with payload = 1  
Sent ipc notify to CORE0 with payload = 101  
[C66xx_0] Sent ipc notify to CORE1 with payload = 1
```

核心 0 收到核心 1 发过来的 Notify 通知事件后, 将收到的附带数据以及谁发过来的这些信息也打印出来, 如下示例所示:

```
[C66xx_0] Sent ipc notify to CORE1 with payload = 1  
Received ipc notify from CORE1 with payload = 101  
Sent ipc notify to CORE1 with payload = 2
```

程序运行产生的所有打印信息:

IPC 核间 Notify 程序运行结果的全部打印信息如下图所示:

Console

XDS200-C6657.ccxml:CIO

```
[C66xx_0] main: MultiProc id = 0
main: MultiProc name = CORE0
[C66xx_1] main: MultiProc id = 1
main: MultiProc name = CORE1
Received ipc notify from CORE0 with payload = 1
Sent ipc notify to CORE0 with payload = 101
[C66xx_0] Sent ipc notify to CORE1 with payload = 1
Received ipc notify from CORE1 with payload = 101
Sent ipc notify to CORE1 with payload = 2
[C66xx_1] Received ipc notify from CORE0 with payload = 2
Sent ipc notify to CORE0 with payload = 102
[C66xx_0] Received ipc notify from CORE1 with payload = 102
Sent ipc notify to CORE1 with payload = 3
[C66xx_1] Received ipc notify from CORE0 with payload = 3
Sent ipc notify to CORE0 with payload = 103
[C66xx_0] Received ipc notify from CORE1 with payload = 103
Sent ipc notify to CORE1 with payload = 4
[C66xx_1] Received ipc notify from CORE0 with payload = 4
Sent ipc notify to CORE0 with payload = 104
[C66xx_0] Received ipc notify from CORE1 with payload = 104
Sent ipc notify to CORE1 with payload = 5
[C66xx_1] Received ipc notify from CORE0 with payload = 5
Sent ipc notify to CORE0 with payload = 105
[C66xx_0] Received ipc notify from CORE1 with payload = 105
Sent ipc notify to CORE1 with payload = 6
[C66xx_1] Received ipc notify from CORE0 with payload = 6
Sent ipc notify to CORE0 with payload = 106
[C66xx_0] Received ipc notify from CORE1 with payload = 106
Sent ipc notify to CORE1 with payload = 7
[C66xx_1] Received ipc notify from CORE0 with payload = 7
Sent ipc notify to CORE0 with payload = 107
[C66xx_0] Received ipc notify from CORE1 with payload = 107
Sent ipc notify to CORE1 with payload = 8
[C66xx_1] Received ipc notify from CORE0 with payload = 8
Sent ipc notify to CORE0 with payload = 108

[C66xx_0] Received ipc notify from CORE1 with payload = 108
Sent ipc notify to CORE1 with payload = 9
[C66xx_1] Received ipc notify from CORE0 with payload = 9
Sent ipc notify to CORE0 with payload = 109
Test completed
[C66xx_0] Received ipc notify from CORE1 with payload = 109
Test completed
```

4.6.3.3 退出实验

CCS 软件窗口上，点击 **Terminate** 断开 DSP 仿真器与板卡的连接。
最后，关闭板卡电源，实验结束。

4.7 DSP IPC 核间通信之 MessageQ

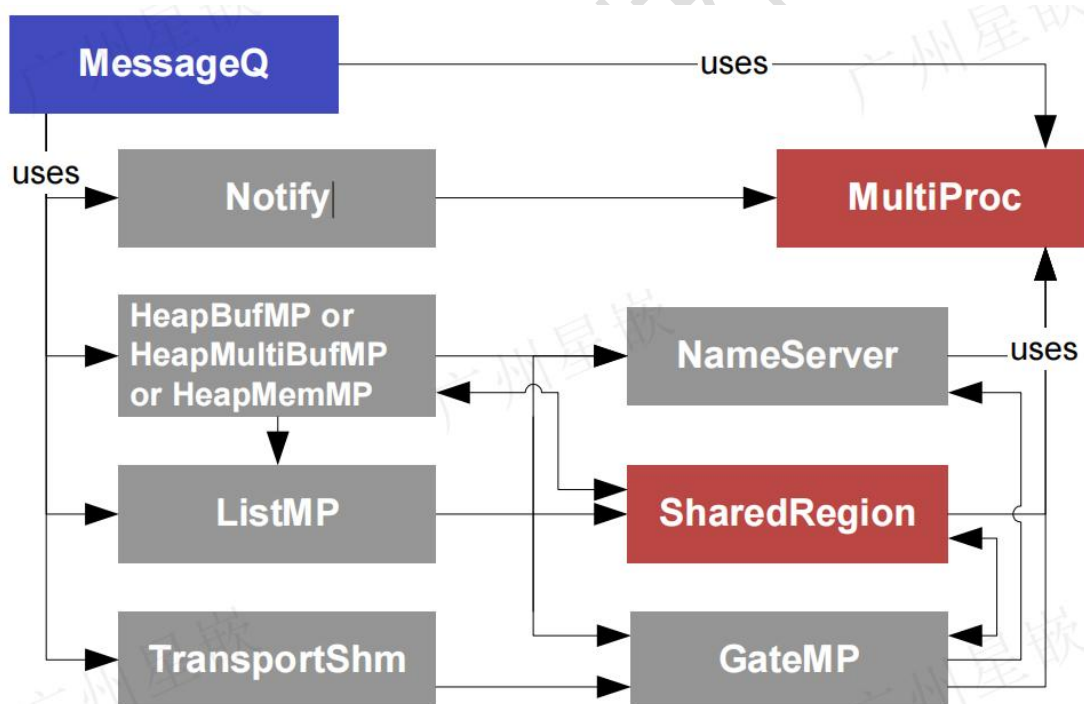
4.7.1 例程位置

DSP 例程保存在资料盘中的 Demo\DSP\XQ_IPC_MESSAGEQ 文件夹下。

4.7.2 功能简介

本实验使用 IPC MessageQ 模块实现 DSP 核心之间的相互通信。

IPC MessageQ 通信场景下，所用到的 IPC 模块以及模块之间调用关系如下图所示：



用户应用程序只需配置 **MultiProc** 和 **SharedRegion** 模块，然后调用 **MessageQ** 模块提供的 API 函数即可。像上图所示的灰色模块不需用户去手动配置，而是由 **ipc_start()** API 函数自动配置。

本实验 **SharedRegion** 模块配置使用静态方式配置，即在 **.cfg** 配置文件中进行 **SharedRegion** 模块配置。**SharedRegion** 模块配置共享内存区域查找表项，以便各处理器利用此查找表项对共享内存区域进行访问。配置内容包括：查找表序号、共享内存区域基地址 (base)/大小 (len)、负责共享内存区域管理的处理器核心号 (ownerProcid)、该共享内存区域是否有效 (isValid) 以及该共享内存区域名称 (name)，具体实现如下图所示：

```

main.c app.cfg
63 /* BIOS/XDC modules */
64 var BIOS          = xdc.useModule('ti.sysbios.BIOS');
65 BIOS.heapSize     = 0x8000;
66 var Task          = xdc.useModule('ti.sysbios.knl.Task');
67
68 var tsk0 = Task.create('&tsk0_func');
69 tsk0.instance.name = "tsk0";
70
71 /* Synchronize all processors (this will be done in Ipc_start) */
72 Ipc.procSync = Ipc.ProcSync_ALL;
73
74 /* Shared Memory base address and length */
75 var SHAREDMEM          = 0x0C000000;
76 var SHARDEMEMSIZE     = 0x00100000;
77
78 /*
79 * Need to define the shared region. The IPC modules use this
80 * to make portable pointers. All processors need to add this
81 * call with their base address of the shared memory region.
82 * If the processor cannot access the memory, do not add it.
83 */
84 var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');
85 SharedRegion.setEntryMeta(0,
86   { base: SHAREDMEM,
87     len: SHARDEMEMSIZE,
88       ownerProcId: 0,
89       isValid: true,
90       name: "DDR3 RAM",
91     });
92

```

MessageQ 实验示例程序 main 函数里面只调用了 Ipc_start() 函数，实现各处理器之间的同步：

```

Int main(Int argc, Char* argv[])
{
    Int status;

    nextProcId = (MultiProc_self() + 1) % MultiProc_getNumProcessors();

    /* Generate queue names based on own proc ID and total number of procs */
    System_sprintf(localQueueName, "%s", MultiProc_getName(MultiProc_self()));
    System_sprintf(nextQueueName, "%s", MultiProc_getName(nextProcId));

    /*
    * Ipc_start() calls Ipc_attach() to synchronize all remote processors
    * because 'Ipc.procSync' is set to 'Ipc.ProcSync_ALL' in *.cfg
    */
    status = Ipc_start();
    if (status < 0) {
        System_abort("Ipc_start failed\n");
    }

    BIOS_start();

    return (0);
}

```

其余 IPC 任务操作放在了一个任务函数里面实现：

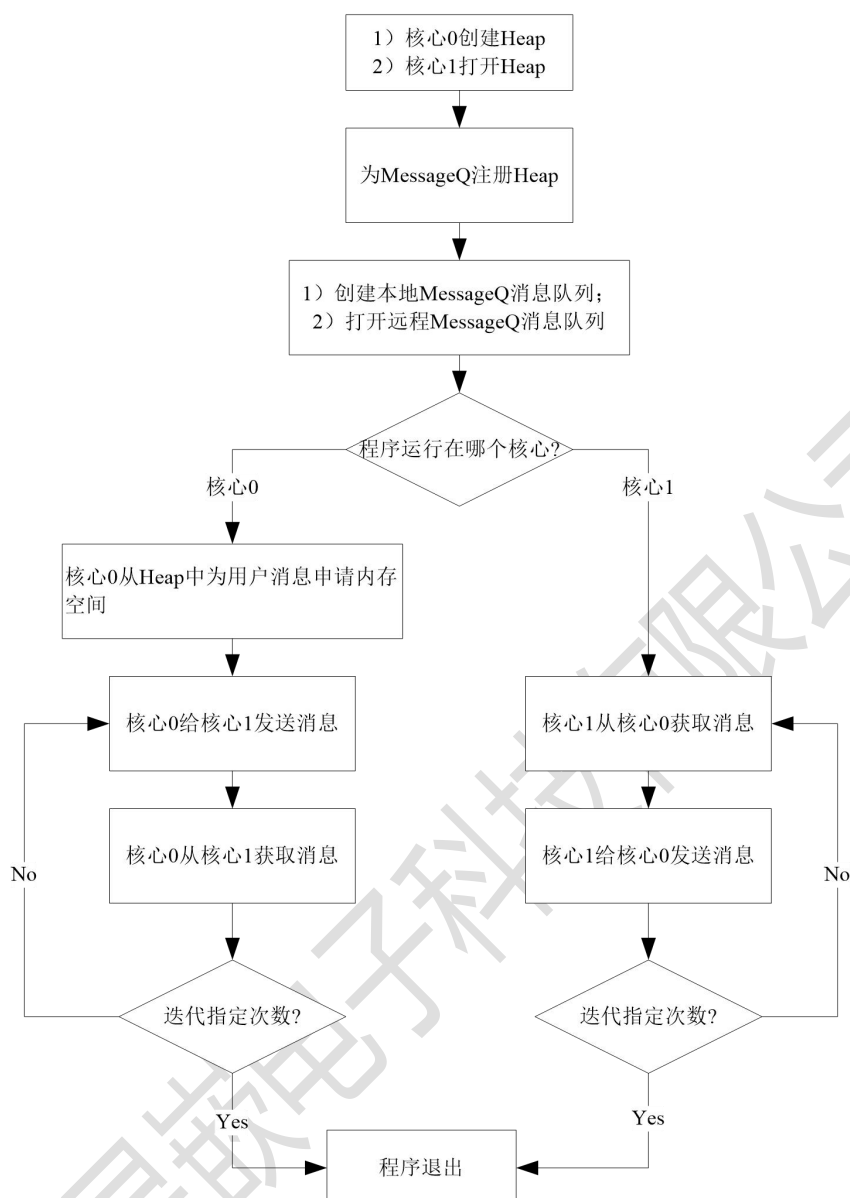
```

* ===== tsk0_tunc =====
* Allocates a message and ping-pongs the message around the processors.
* A local message queue is created and a remote message queue is opened.
* Messages are sent to the remote message queue and retrieved from the
* local MessageQ.
*/
Void tsk0_func(UArg arg0, UArg arg1)
{
    pMessageQ_UserMsg msg;
    MessageQ_Handle messageQ;
    MessageQ_QueueId remoteQueueId;
    Int status;
    UInt16 msgId = 0;
    HeapBufMP_Handle heapHandle;
    HeapBufMP_Params heapBufParams;

    if (MultiProc_self() == 0) { /* 核心0 运行代码*/
        /*
        * Create the heap that will be used to allocate messages.
        */
        HeapBufMP_Params_init(&heapBufParams);
        heapBufParams.regionId = 0;
        heapBufParams.name = HEAP_NAME;
        heapBufParams.numBlocks = 1;
        heapBufParams.blockSize = sizeof(MessageQ_MsgHeader);
        heapHandle = HeapBufMP_create(&heapBufParams);
        if (heapHandle == NULL) {
            System_abort("HeapBufMP_create failed\n" );
        }
    }
}

```

tsk0_func 任务函数实现的程序流程图如下图所示:

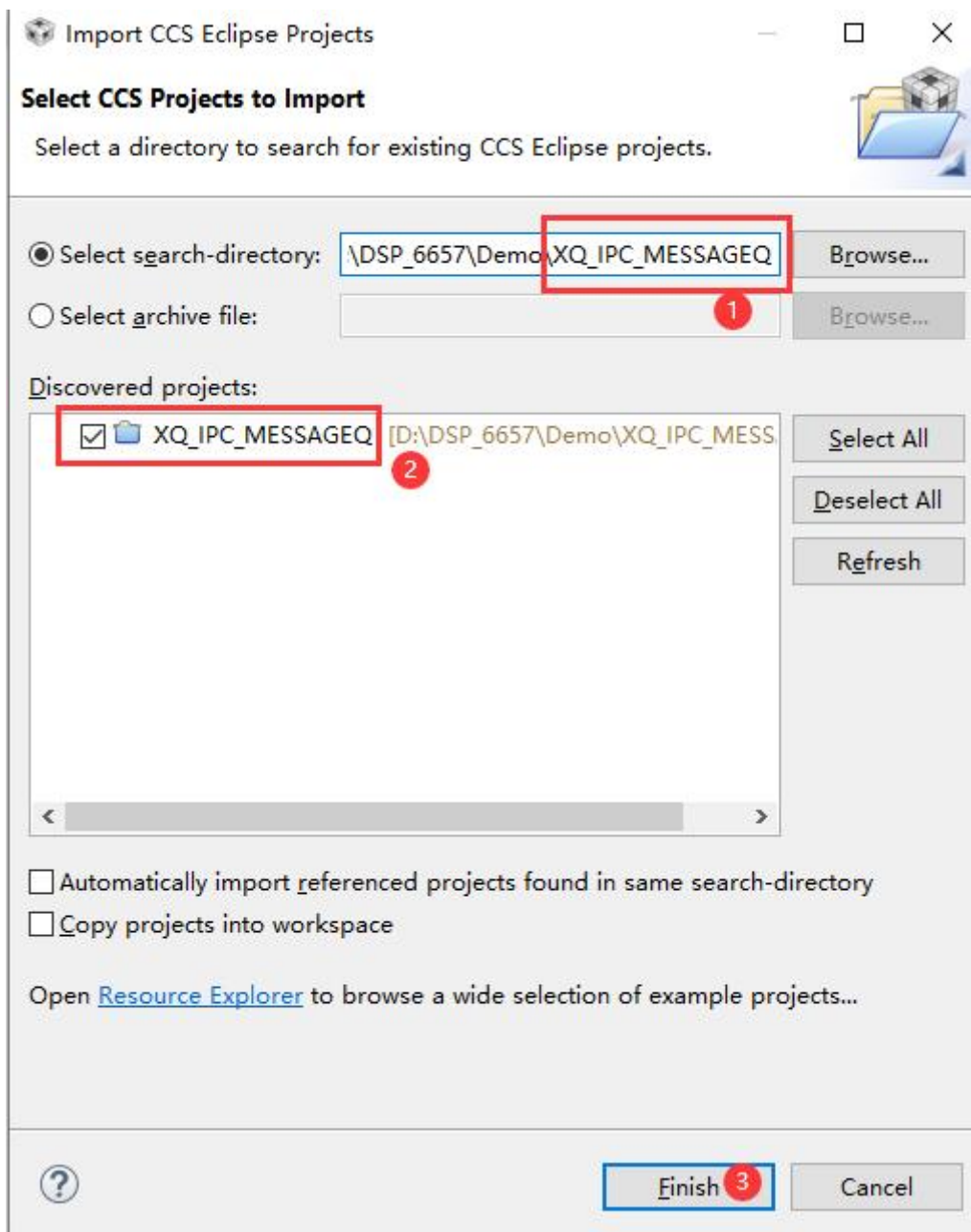


4.7.3 例程使用

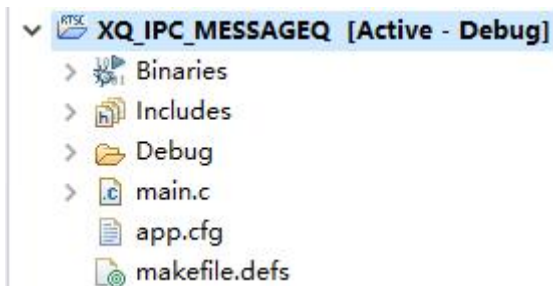
4.7.3.1 加载运行 DSP 程序

4.7.3.1.1 CCS 导入例程

CCS 软件导入示例工程 XQ_IPC_MESSAGEQ，如下图所示：



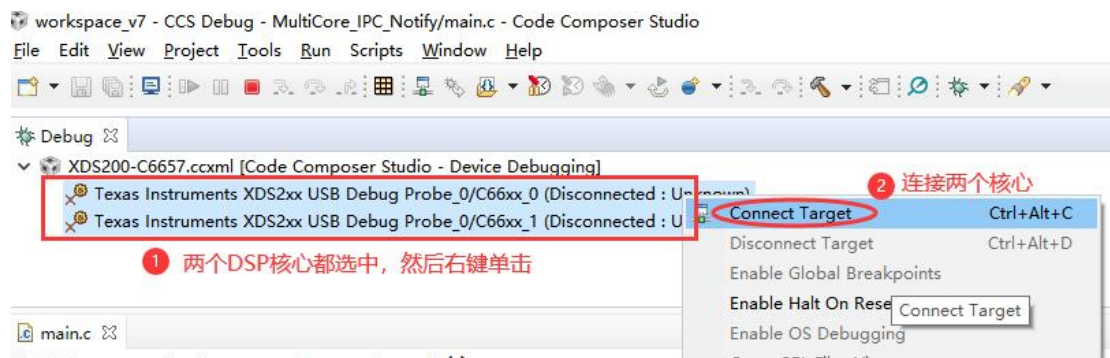
CCS 示例工程导入后界面如下图所示：



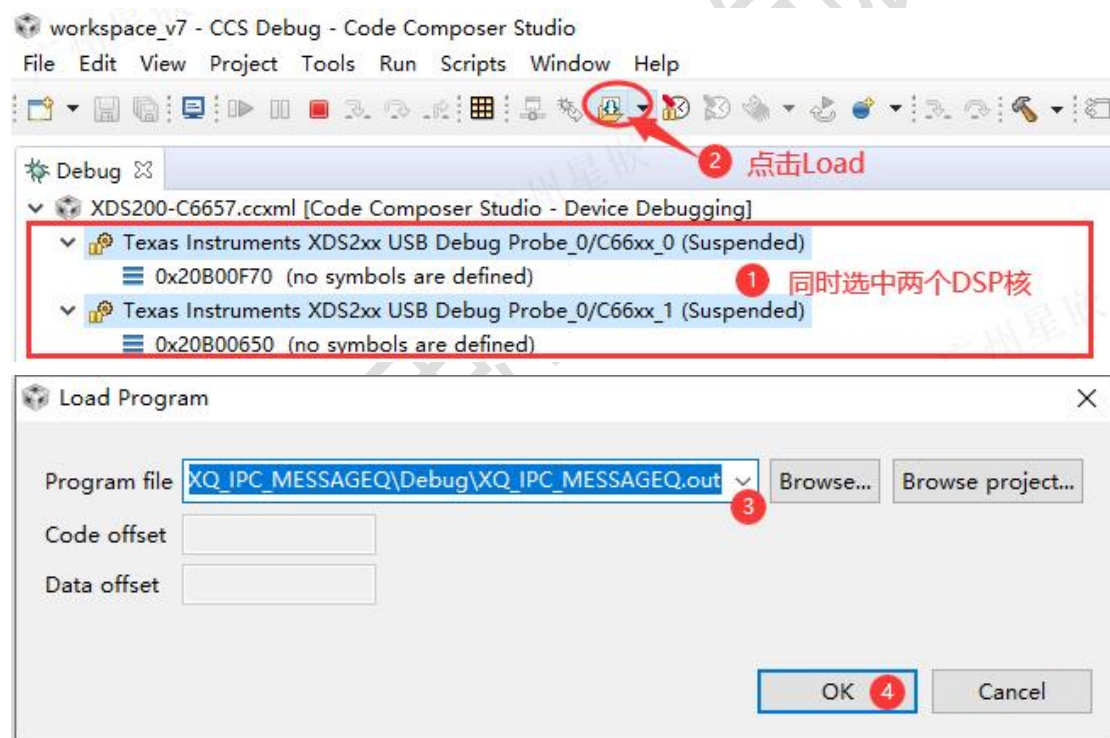
4.7.3.1.2 下载运行 CCS 程序

此实验需要连接 DSP 的两个核心，并给这两个 DSP 核心下载同一个镜像文件。

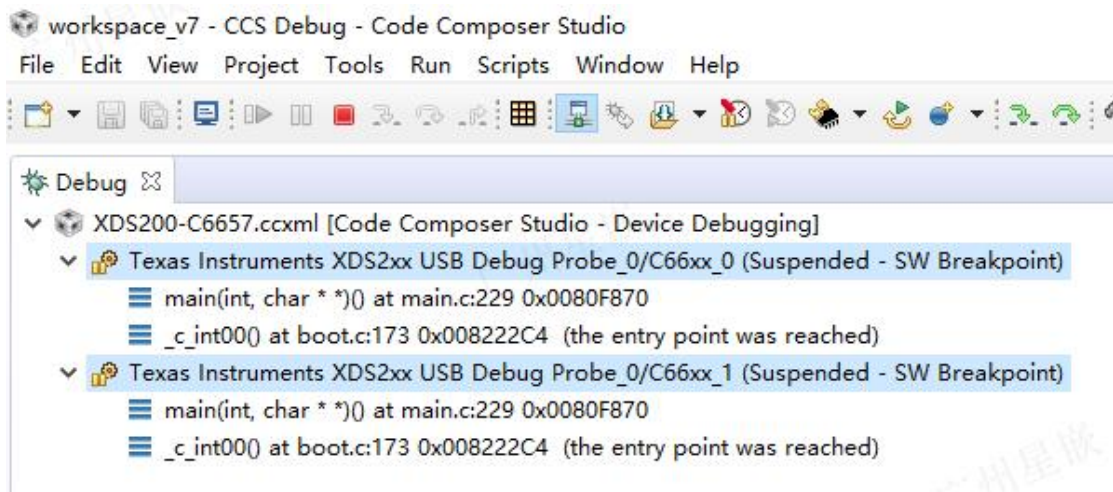
首先，连接两个 DSP 核心：



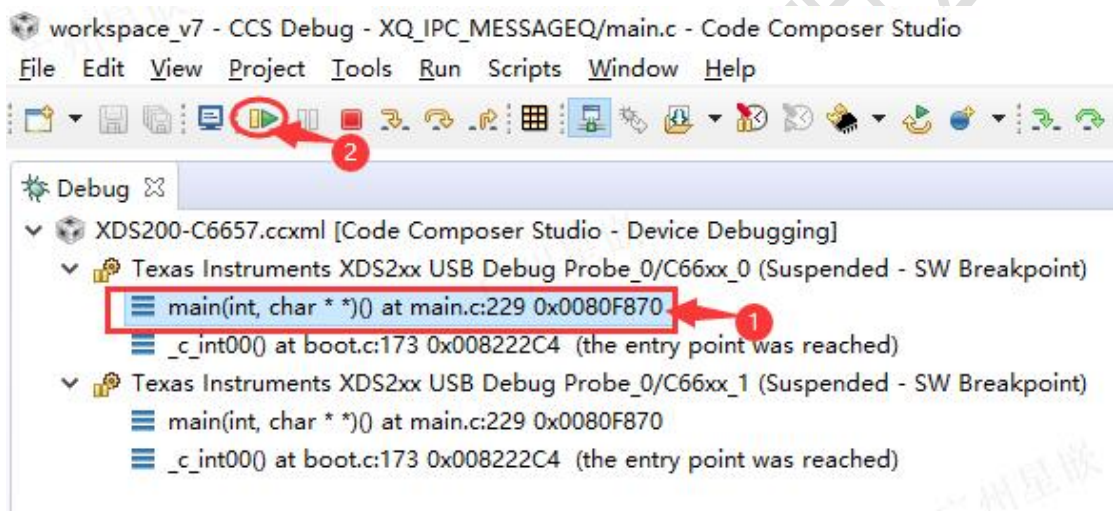
其次，两个 DSP 核心连接上后，给两个 DSP 核心下载同一个镜像文件 XQ_IPC_MESSAGEQ.out：



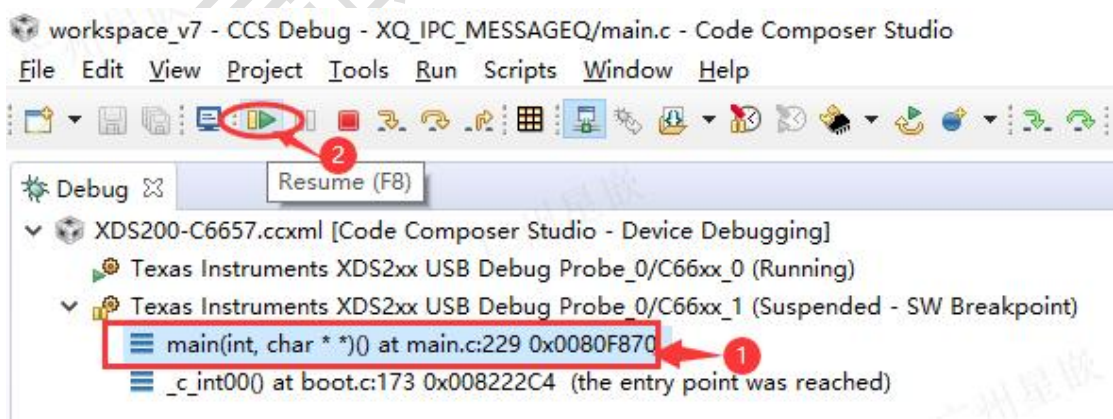
两个 DSP 核心均下载完 DSP 可执行镜像文件 XQ_IPC_MESSAGEQ.out 后的界面如下图所示：



选中核心 0 的 main 函数位置，点击 Resume 运行 DSP 核心 0 的程序：



选中核心 1 的 main 函数位置，点击 Resume 运行 DSP 核心 1 的程序：



4.7.3.2 运行结果说明

CCS Console 窗口打印程序运行结果。下面，首先对部分打印信息进行解读，然后附带

给出程序运行产生的所有打印信息。

部分打印信息解读：

核心 0 从核心 1 那获得消息，消息 ID 为 1，消息数据为 201；核心 0 给核心 1 发送消息，消息 ID 为 2，消息数据为 102，Console 打印信息如下示例所示：

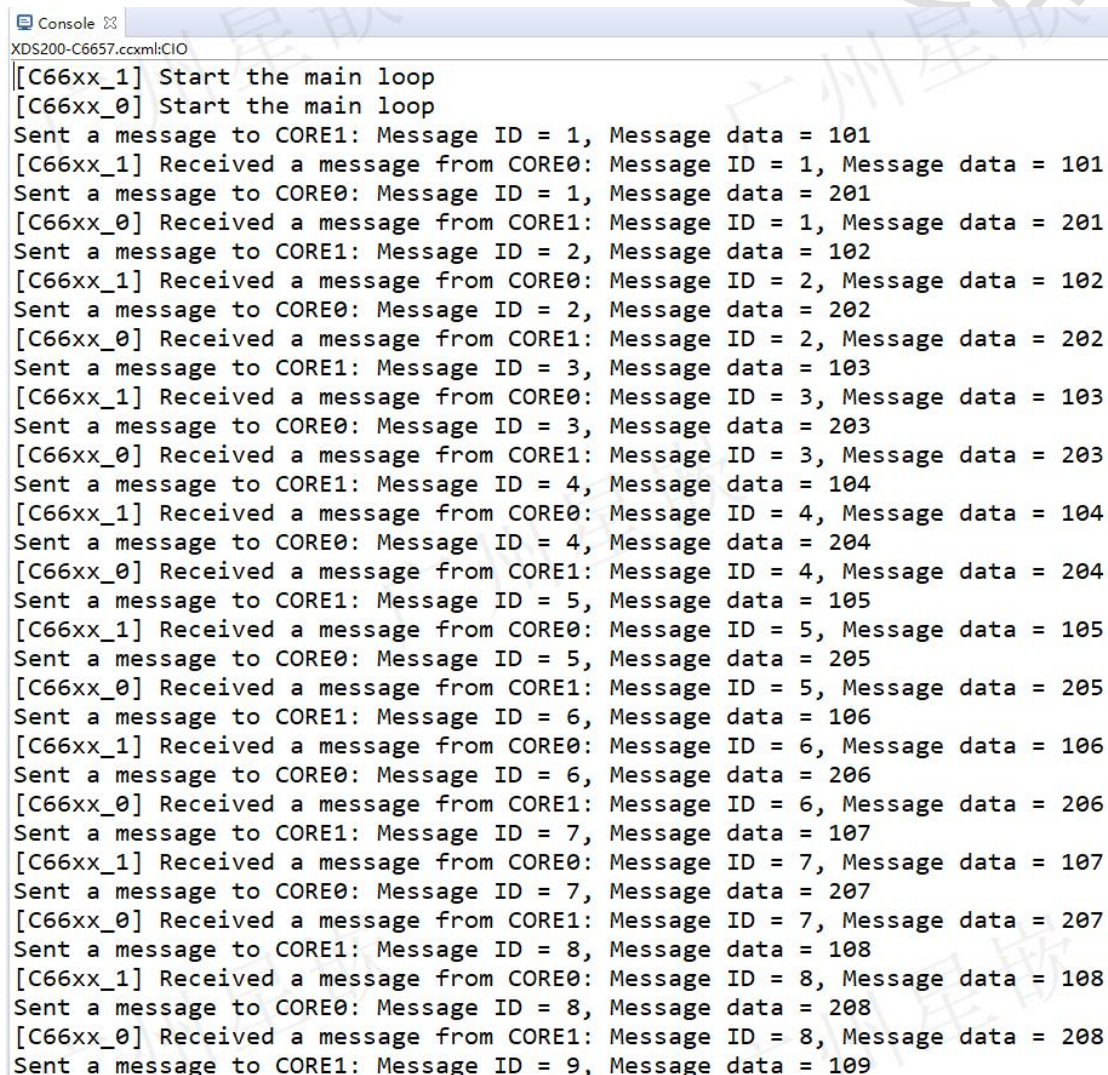
```
[C66xx_0] Received a message from CORE1: Message ID = 1, Message data = 201  
Sent a message to CORE1: Message ID = 2, Message data = 102
```

核心 1 从核心 0 那获得消息，消息 ID 为 2，消息数据为 102；核心 1 给核心 0 发送消息，消息 ID 为 2，消息数据为 202，Console 打印信息如下示例所示：

```
[C66xx_1] Received a message from CORE0: Message ID = 2, Message data = 102  
Sent a message to CORE0: Message ID = 2, Message data = 202
```

程序运行产生的所有打印信息：

IPC 核间 MessageQ 程序运行结果的全部打印信息如下图所示：



```
Console XDS200-C6657.cxml:CI0  
[C66xx_1] Start the main loop  
[C66xx_0] Start the main loop  
Sent a message to CORE1: Message ID = 1, Message data = 101  
[C66xx_1] Received a message from CORE0: Message ID = 1, Message data = 101  
Sent a message to CORE0: Message ID = 1, Message data = 201  
[C66xx_0] Received a message from CORE1: Message ID = 1, Message data = 201  
Sent a message to CORE1: Message ID = 2, Message data = 102  
[C66xx_1] Received a message from CORE0: Message ID = 2, Message data = 102  
Sent a message to CORE0: Message ID = 2, Message data = 202  
[C66xx_0] Received a message from CORE1: Message ID = 2, Message data = 202  
Sent a message to CORE1: Message ID = 3, Message data = 103  
[C66xx_1] Received a message from CORE0: Message ID = 3, Message data = 103  
Sent a message to CORE0: Message ID = 3, Message data = 203  
[C66xx_0] Received a message from CORE1: Message ID = 3, Message data = 203  
Sent a message to CORE1: Message ID = 4, Message data = 104  
[C66xx_1] Received a message from CORE0: Message ID = 4, Message data = 104  
Sent a message to CORE0: Message ID = 4, Message data = 204  
[C66xx_0] Received a message from CORE1: Message ID = 4, Message data = 204  
Sent a message to CORE1: Message ID = 5, Message data = 105  
[C66xx_1] Received a message from CORE0: Message ID = 5, Message data = 105  
Sent a message to CORE0: Message ID = 5, Message data = 205  
[C66xx_0] Received a message from CORE1: Message ID = 5, Message data = 205  
Sent a message to CORE1: Message ID = 6, Message data = 106  
[C66xx_1] Received a message from CORE0: Message ID = 6, Message data = 106  
Sent a message to CORE0: Message ID = 6, Message data = 206  
[C66xx_0] Received a message from CORE1: Message ID = 6, Message data = 206  
Sent a message to CORE1: Message ID = 7, Message data = 107  
[C66xx_1] Received a message from CORE0: Message ID = 7, Message data = 107  
Sent a message to CORE0: Message ID = 7, Message data = 207  
[C66xx_0] Received a message from CORE1: Message ID = 7, Message data = 207  
Sent a message to CORE1: Message ID = 8, Message data = 108  
[C66xx_1] Received a message from CORE0: Message ID = 8, Message data = 108  
Sent a message to CORE0: Message ID = 8, Message data = 208  
[C66xx_0] Received a message from CORE1: Message ID = 8, Message data = 208  
Sent a message to CORE1: Message ID = 9, Message data = 109
```

```
[C66xx_1] Received a message from CORE0: Message ID = 9, Message data = 109
Sent a message to CORE0: Message ID = 9, Message data = 209
[C66xx_0] Received a message from CORE1: Message ID = 9, Message data = 209
Sent a message to CORE1: Message ID = 10, Message data = 110
[C66xx_1] Received a message from CORE0: Message ID = 10, Message data = 110
Sent a message to CORE0: Message ID = 10, Message data = 210
The test is complete
[C66xx_0] Received a message from CORE1: Message ID = 10, Message data = 210
The test is complete
```

4.7.3.3 退出实验

CCS 软件窗口上，点击 Terminate 断开 DSP 仿真器与板卡的连接。
最后，关闭板卡电源，实验结束。

5 ZYNQ PL 单独例程

5.1 ZYNQ PL Cameralink 回环例程

5.1.1 例程位置

ZYNQ 例程保存在资料盘中的 Demo\ZYNQ\PL\base_cameralink_loop\prj 文件夹下。

5.1.2 功能简介

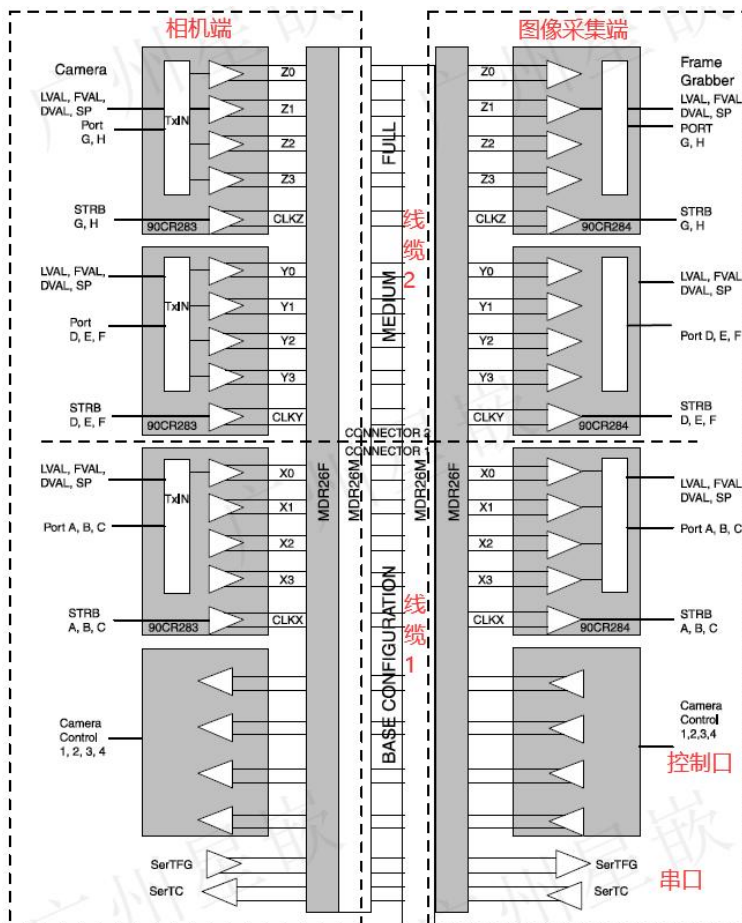
Cameralink 回环例程将 J3、J4 当作两个独立的 Base Cameralink 接口使用，一个接收，另一个发送。

Cameralink 接收端，利用 Xilinx ISERDESE2 原语进行串/并转换，将 LVDS 串行数据转换成 28bit 的 cameralink 并行数据。解串后的并行数据通过 ila 进行在线分析和查看，并实时检测并行数据是否有误码。

Cameralink 发送端，利用 Xilinx OSERDESE2 原语进行并/串转换，将本地 28bit cameralink 并行数据串行化为 LVDS 数据发送出去。

5.1.3 Cameralink 接口时序说明

5.1.3.1 Cameralink 三种配置模式



Base 模式: 只需一根 Cameralink 线缆；4 对差分数据、1 对差分时钟；

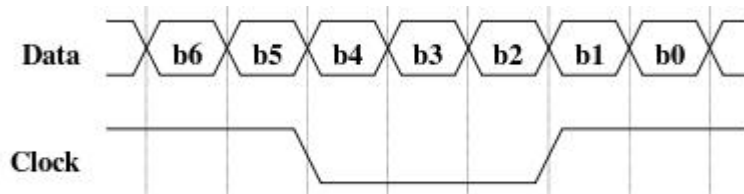
Medium 模式: 需要两根 Cameralink 线缆；8 对差分数据、2 对差分时钟；

Full 模式: 需要两根 Cameralink 线缆；12 对差分数据、3 对差分时钟。

各种模式下，统一都包含一组控制口和一组串口。控制口有 4 根信号，用于图像采集端对相机的 IO 控制；串口用于图像采集端对相机参数的配置。

5.1.3.2 单路差分数据与时钟之间时序关系

单路 Cameralink 差分数据与随路的差分像素时钟之间的时序关系如下图所示：



一个时钟周期内传输 7bits 串行数据，首先传输串行数据的最高位，最后传输串行数据的最低位。7bits 数据起始于像素时钟高电平的中间位置，即数据的最高位在 Clock 高电平的中间时刻开始传输。

Clock 高电平时间比 Clock 低电平时间多一个 bit 位。

5.1.3.3 通道传输数据与图像数据映射关系

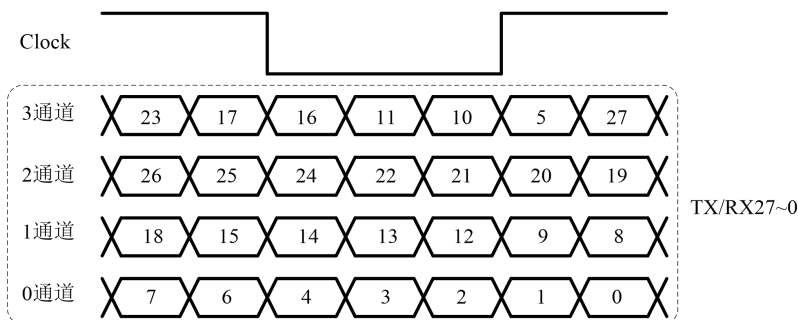
1 路差分数据通道上，一个 Clock 像素时钟周期传输 7bits 串行数据，那么 4 路差分数据通道总共就是 $4 \times 7\text{bits} = 28\text{bits}$ ，我们称这 28bits 数据为并行数据，为了方便描述，这 28bits 数据记为 TX/RX27~0。Cameralink Base 模式下，这 28bits 数据与图像行/场同步/数据有效标记、图像数据的映射关系如下图所示：

TX/RX24		LVAL
TX/RX25		FVAL
TX/RX26		DVAL
TX/RX23		Spare
TX/RX0		PortA0
TX/RX1		PortA1
TX/RX2		PortA2
TX/RX3		PortA3
TX/RX4	28bits	PortA4
TX/RX6	并行数据	PortA5
TX/RX27		PortA6
TX/RX5		PortA7
TX/RX7		PortB0
TX/RX8		PortB1
TX/RX9		PortB2
TX/RX12		PortB3
TX/RX13		PortB4
TX/RX14		PortB5
TX/RX10		PortB6
TX/RX11		PortB7
TX/RX15		PortC0
TX/RX18		PortC1
TX/RX19		PortC2
TX/RX20		PortC3
TX/RX21		PortC4
TX/RX22		PortC5
TX/RX16		PortC6
TX/RX17		PortC7

TX/RX24 映射为行同步标记 LVAL，TX/RX25 映射为场同步标记 FVAL，TX/RX26 映射为图像数据有效标记 DVAL，TX/RX23 未使用，其余位对应图像数据。

5.1.3.4 28 位并行数据与 4 路差分数据通道之间的映射关系

上述 28 位并行数据是如何通过 4 路差分数据通道进行传输的呢？28 位并行数据映射到 4 路差分数据通道各个时刻点的位置关系如下图所示：



5.1.4 管脚约束

ZYNQ PL 工程管脚约束如下图所示：

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	IN_T
refclk_n	IN		J4	✓	33	LVDS*					NONE	NONE	
i_lvds_ncdk_n	IN		G7	✓	34	LVDS*	1.800				NONE	NONE	
o_lvds_bcdk_n	OUT		H7	✓	34	LVDS*	1.800				NONE	FD_100	
i_lvds_nrd_n	IN			✓	34	LVDS*	1.800				NONE	NONE	
i_lvds_nrd_n[3]	IN		C4	✓	34	LVDS*	1.800				NONE	NONE	
i_lvds_nrd_n[2]	IN		B5	✓	34	LVDS*	1.800				NONE	NONE	
i_lvds_nrd_n[1]	IN		B6	✓	34	LVDS*	1.800				NONE	NONE	
i_lvds_nrd_n[0]	IN		A4	✓	34	LVDS*	1.800				NONE	NONE	
o_lvds_brd_n	OUT			✓	34	LVDS*	1.800				NONE	FD_100	
o_lvds_brd_n[3]	OUT		J10	✓	34	LVDS*	1.800				NONE	FD_100	
o_lvds_brd_n[2]	OUT		B2	✓	34	LVDS*	1.800				NONE	FD_100	
o_lvds_brd_n[1]	OUT		G6	✓	34	LVDS*	1.800				NONE	FD_100	
o_lvds_brd_n[0]	OUT		J11	✓	34	LVDS*	1.800				NONE	FD_100	

5.1.5 例程使用

5.1.5.1 连接 Cameralink 线缆

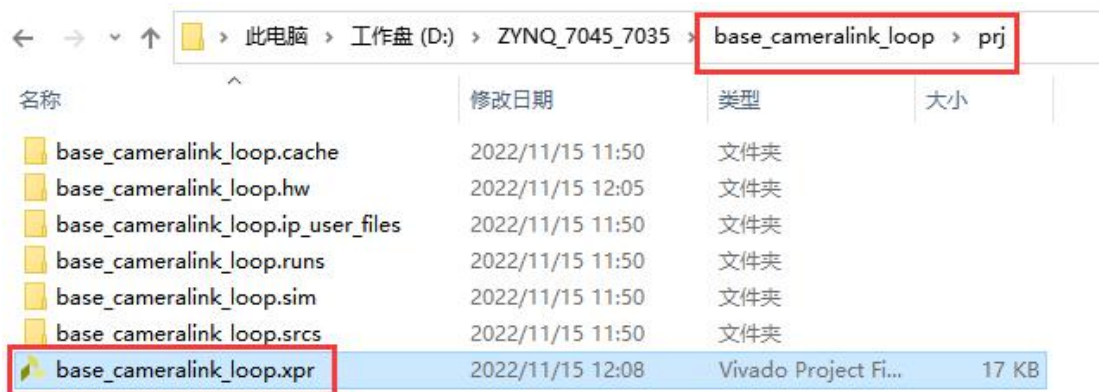
使用 Cameralink 线缆将 J3、J4 两个接口连接在一起：



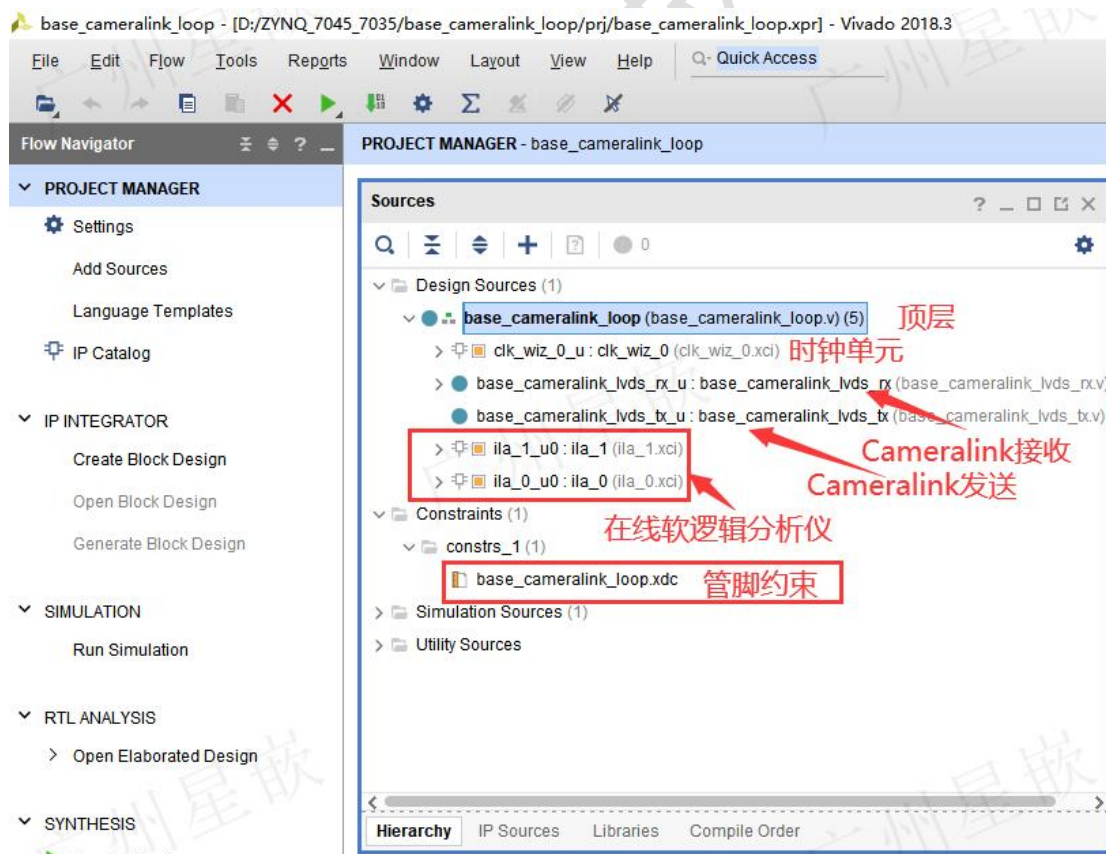
5.1.5.2 加载运行 ZYNQ 程序

5.1.5.2.1 打开 Vivado 工程

打开 Vivado 示例工程：

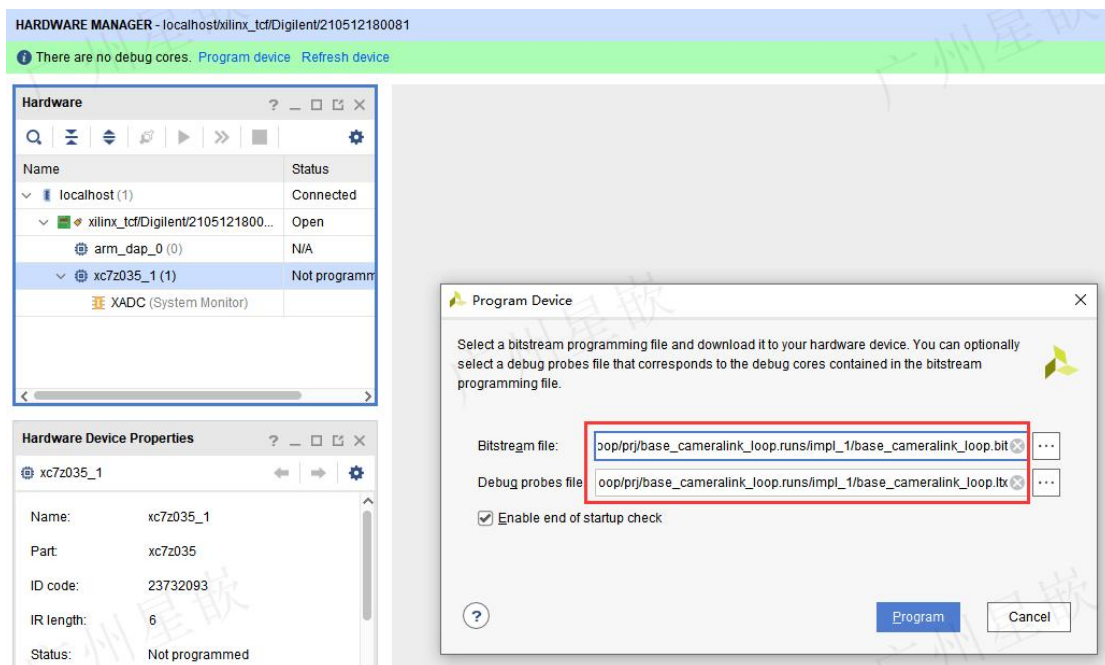


工程打开后界面如下图所示：



5.1.5.2 下载 ZYNQ PL 程序

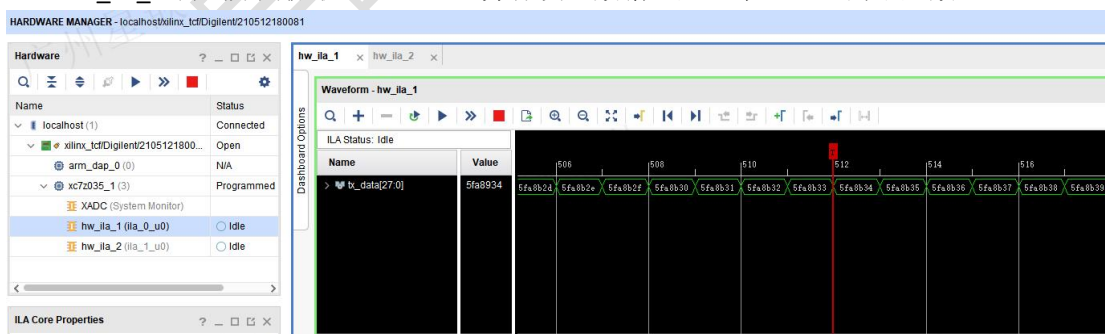
下载 bit 流文件 base_cameralink_loop.bit, 并且配套 base_cameralink_loop.ltx 调试文件, 如下图下载界面所示:



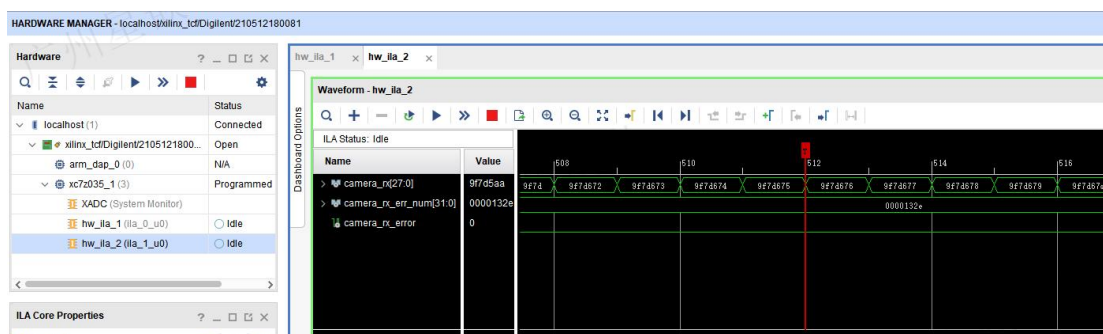
5.1.5.3 运行结果说明

ZYNQ PL 端提供的 ILA 调试窗口, 可以实时抓取采集 Cameralink 并行信号以及错误检测信号的时序波形。

hw_ila_1 调试界面抓取 Cameralink 并行发送数据, 是一个 28bits 的累加数:



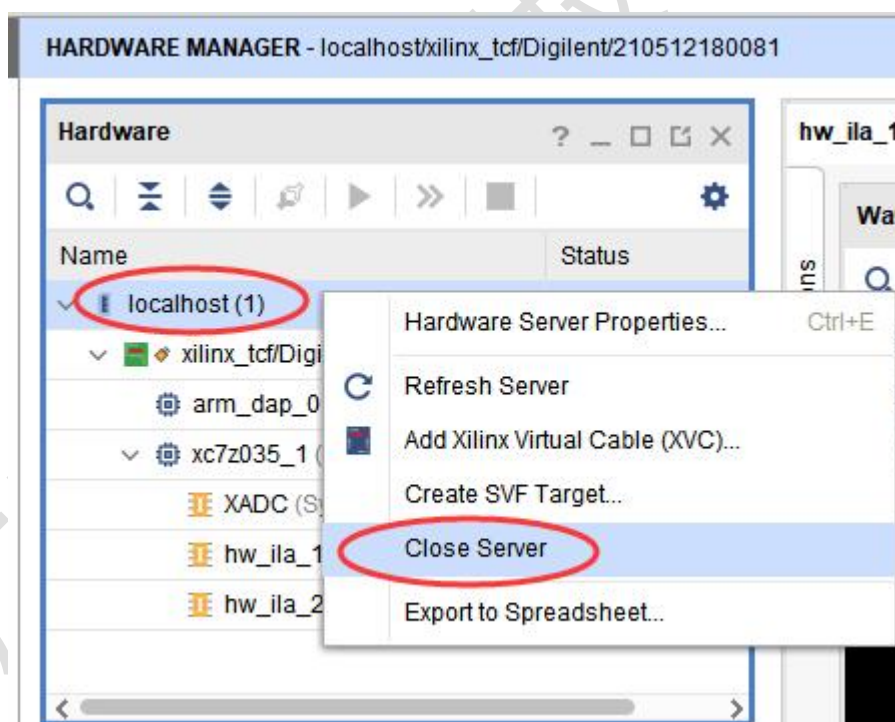
hw_ila_2 调试界面抓取 Cameralink 并行接收数据、接收误码统计以及接收误码实时标识信号, 如下图所示:



cameralink_rx_err_num 显示有数值，则说明 Cameralink 接收过程中存在误码。可能在开始通信初始化期间存在误码现象，导致 cameralink_rx_err_num 误码统计累加。待程序下载完毕后，如果 Cameralink 通信正常的话，cameralink_rx_err_num 误码统计应该不会再累加。如果 cameralink_rx_err_num 误码统计继续不断累加，则通过触发 camera_rx_error 信号可以捕捉到误码具体发生时刻。

5.1.5.4 退出实验

Vivado 调试界面 Hardware Manager 窗口，右键单击 localhost(1)，在弹出的菜单中点击 Close Server，断开 ZYNQ JTAG 仿真器与板卡的连接：



最后，关闭板卡电源，实验结束。

5.2 ZYNQPL SFP 光口通信例程

5.2.1 例程位置

ZYNQ 例程保存在资料盘中的 Demo\ZYNQ\PL\aurora_8b10b_0_ex 文件夹下。

5.2.2 功能简介

使用 Aurora 8B/10B IP 核生成后带的例子工程，稍作修改。

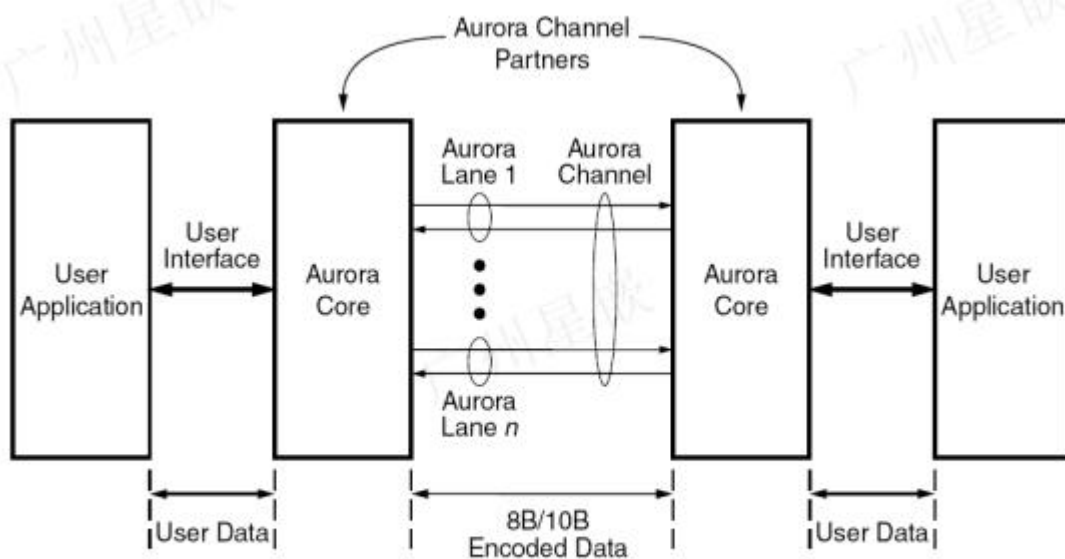


Figure 1-1: Aurora 8B/10B Channel Overview

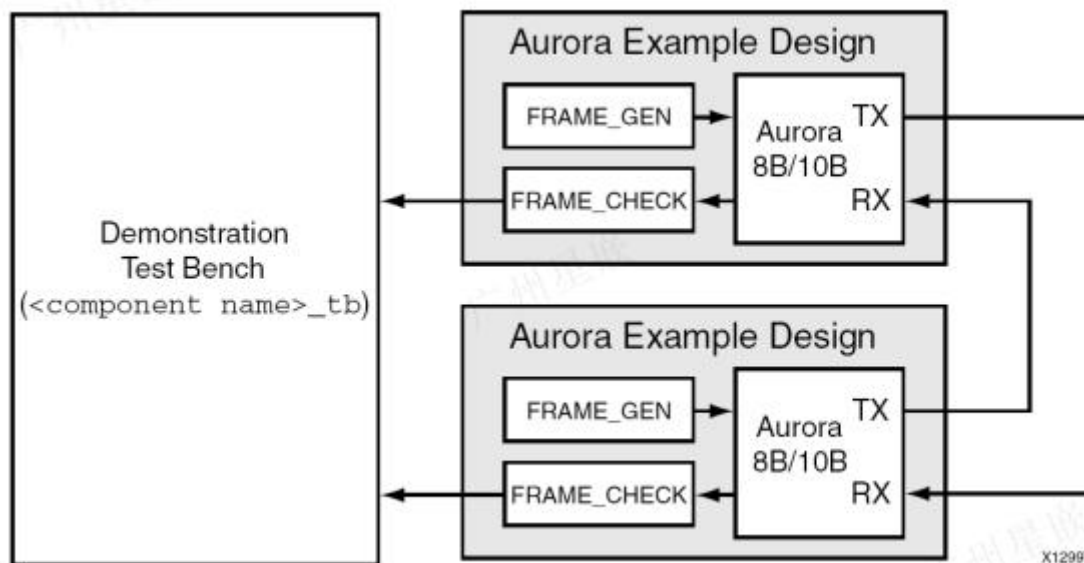


图 Aurora 8B/10B 例子工程

FRAME_GEN: 本地并行数据发送模块

功能: 本地产生 GTX 并行发送数据

接口说明:

```
// User Interface
output [0:15] TX_D; //发送数据
output TX_REM; //最后一个 16bits 数据的高低字节有效标识, 0 表示,
表示 TX_D[0:7]有效; 1 表示 TX_D[0:15]有效。
output TX_SOF_N; //发送开始标识, 低电平有效
output TX_EOF_N; //发送结束标识, 低电平有效
output TX_SRC_RDY_N; //发送数据源端准备好标志, 低有效
input TX_DST_RDY_N; //发送数据目的端准备好标, 为 0 时才允许发送数
据

// System Interface
input USER_CLK; //用户时钟, 由 Aurora IP 核提供, 数据发送模块用此时
钟作为同步时钟
input RESET; //复位, 高有效
input CHANNEL_UP; //GTX 通道初始化完成标志, 为 1 时表示完成
```

数据发送模块只有在 RESET=0、CHANNEL_UP=1 和 TX_DST_RDY_N=0 时, 才允许发送数据。

FRAME_CHECK: 本地并行数据接收检测模块

功能: 本地接收 GTX 并行数据, 并检测数据是否存在误码

接口说明:

```
// User Interface
input [0:15] RX_D; //接收数据
output RX_REM; //最后一个 16bits 数据的高低字节有效标识, 0 表示,
表示 RX_D[0:7]有效; 1 表示 RX_D[0:15]有效。
output RX_SOF_N; //接收开始标识, 低电平有效
output RX_EOF_N; //接收结束标识, 低电平有效
input RX_SRC_RDY_N; //接收数据有效, 低电平有效

// System Interface
input USER_CLK; //用户时钟, 由 Aurora IP 核提供, 数据发送模块用此时
钟作为同步时钟
input RESET; //复位, 高有效
input CHANNEL_UP; //GTX 通道初始化完成标志, 为 1 时表示完成
output [0:7] ERR_COUNT; //接收数据错误个数
```

Aurora 8B10B IP 核参数设置如下图所示:

Aurora 8B10B (11.1)

Documentation IP Location Switch to Defaults

Show disabled ports

Component Name: `aurora_8b10b_0`

Core Options GT Selections Shared Logic

Physical Layer

- Lane Width (Bytes): 2
- Line Rate (Gbps): 5
- GT Refclk (MHz): 100.000
- INIT clk (MHz): 100
- DRP Clk (in MHz): 100

Link Layer

- Dataflow Mode: Duplex
- Interface: Framing
- Flow Control: None
- Back Channel: Sidebands
- Scrambler/Descrambler Little Endian Support

Error Detection

- CRC

Debug and Control

- Vivado Lab Tools
- Additional transceiver control and status ports

Port list:

- + USER_DATA_S_AXI_TX
- + GT0_DRP_IF
- + CORE_CONTROL
- + GT_SERIAL_RX
- + QPLL_CONTROL_IN
- USER_DATA_M_AXI_RX
- + CORE_STATUS
- + GT_SERIAL_TX
- reset
- gt_reset
- drpclk_in
- gt_qpllclk_quad4_in
- gt_qpllrefclk_quad4_in
- init_clk_in
- user_clk
- sync_clk
- gt_refclk1
- link_reset_out
- gt0_qpllreset_out
- tx_out_clk
- sys_reset_out

Aurora 8B10B IP 核显示最高只支持 6.6Gbps，这里我们将线速率设置为 5Gbps，参考时钟设置为 100MHz。

5.2.3 管脚约束

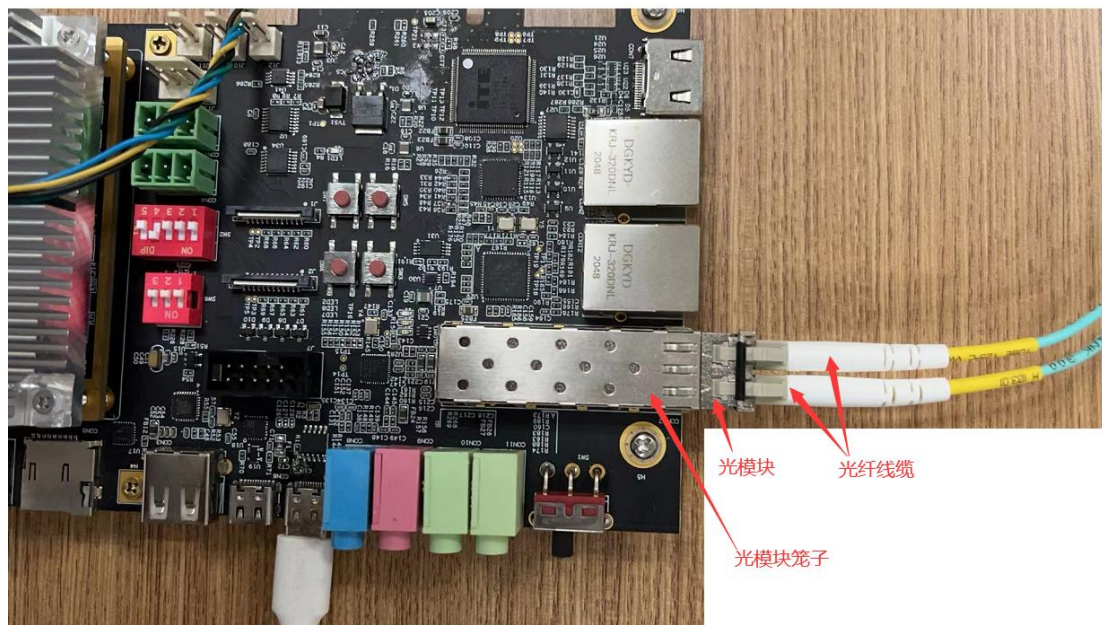
ZYNQ PL 工程管脚约束如下图所示：

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcc0	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination
IN	GTXQ3_N		R6	✓	112							
IN	INIT_CLK_N		J4	✓	33	DIFF_HSTL_IL18					NONE	NONE
IN	RXN		V4	✓	112							
OUT	TXN		U2	✓	112							

5.2.4 例程使用

5.2.4.1 连接光纤模块

将光模块插入光模块笼子，并使用光纤线缆将光模块的收、发端口自环对接：



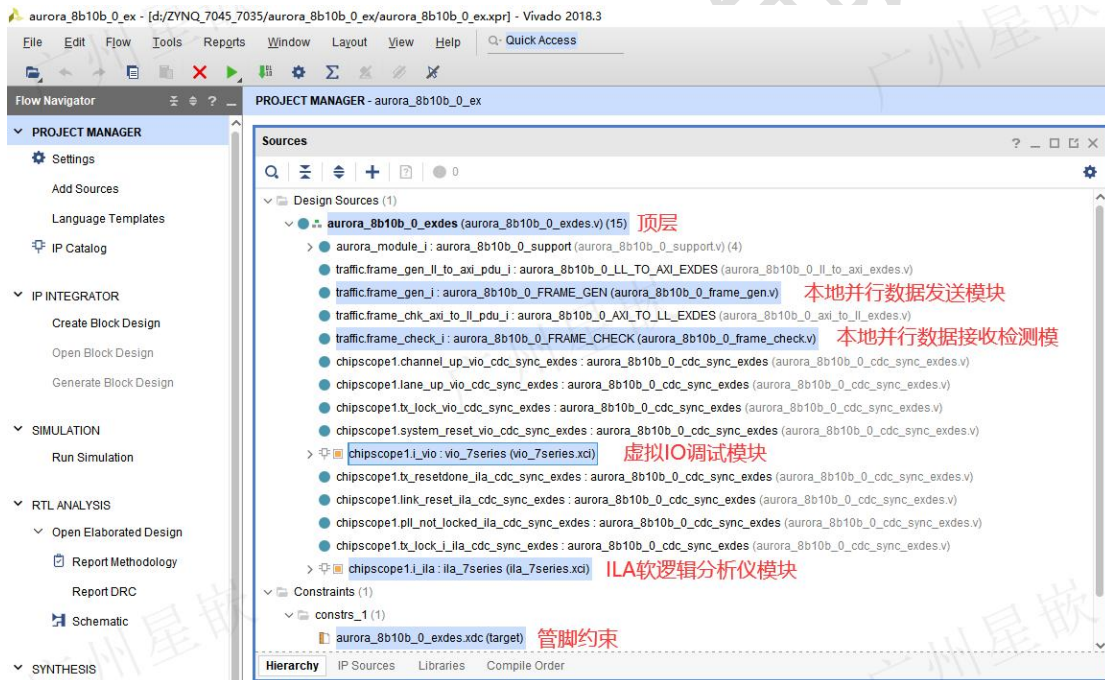
5.2.4.2 加载运行 ZYNQ 程序

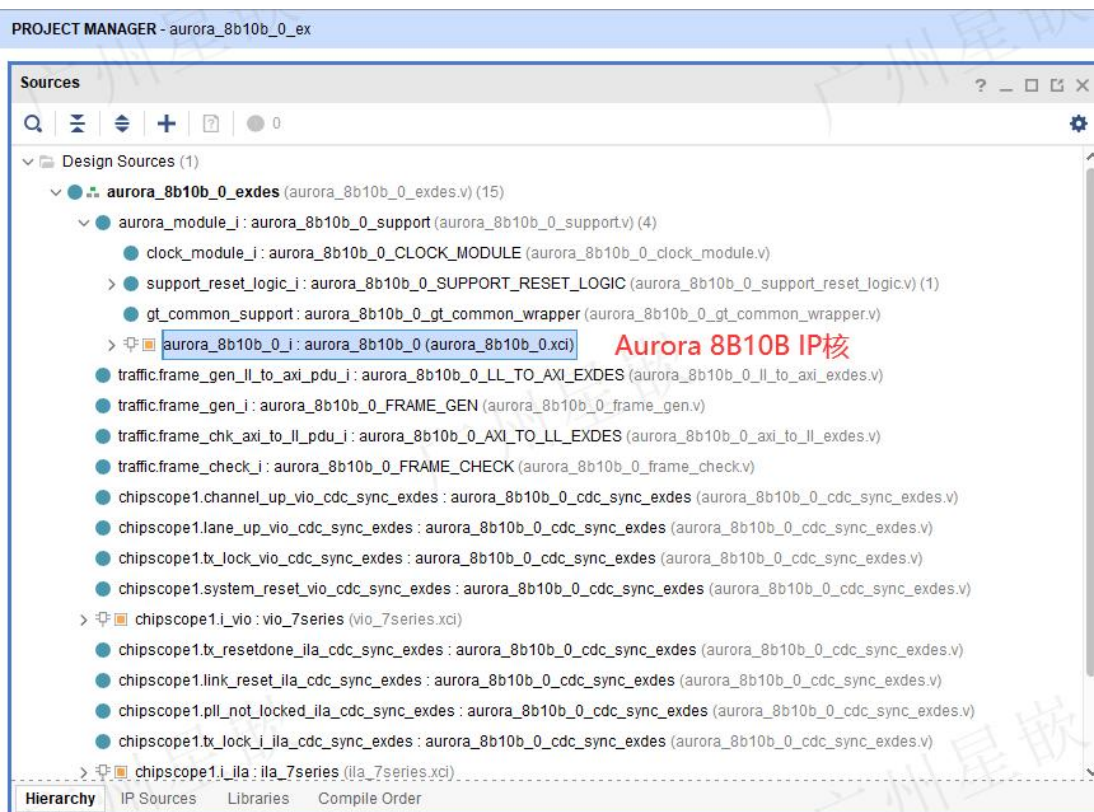
5.2.4.2.1 打开 Vivado 工程

打开 Vivado 示例工程：



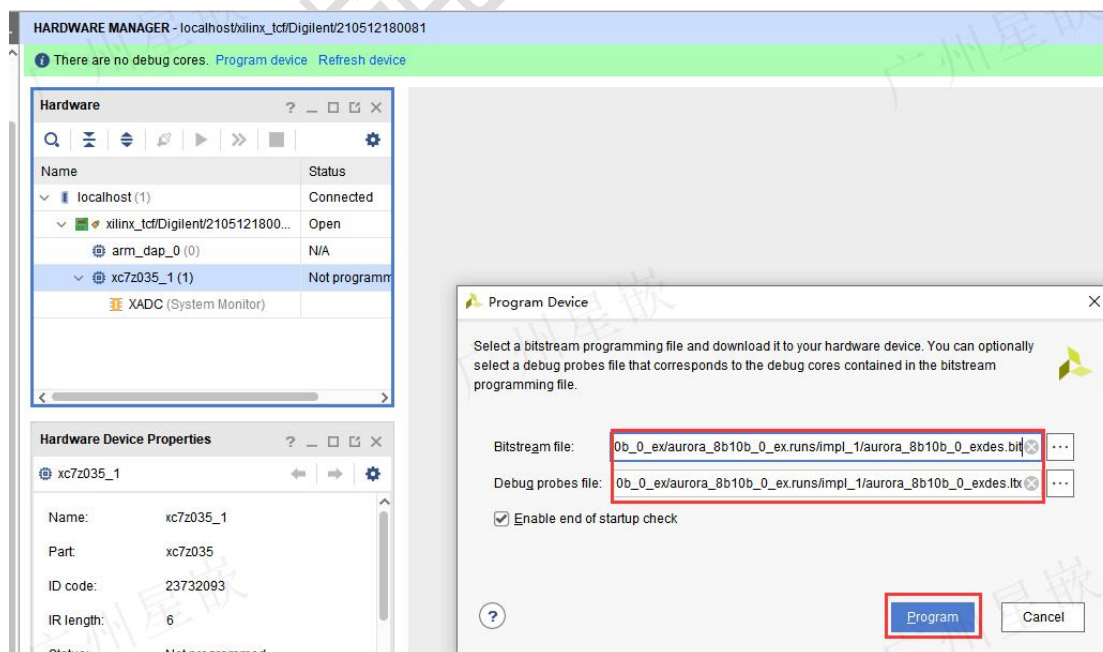
工程打开后界面及工程主要模块说明如下图所示：





5.2.4.2.2 下载 ZYNQ PL 程序

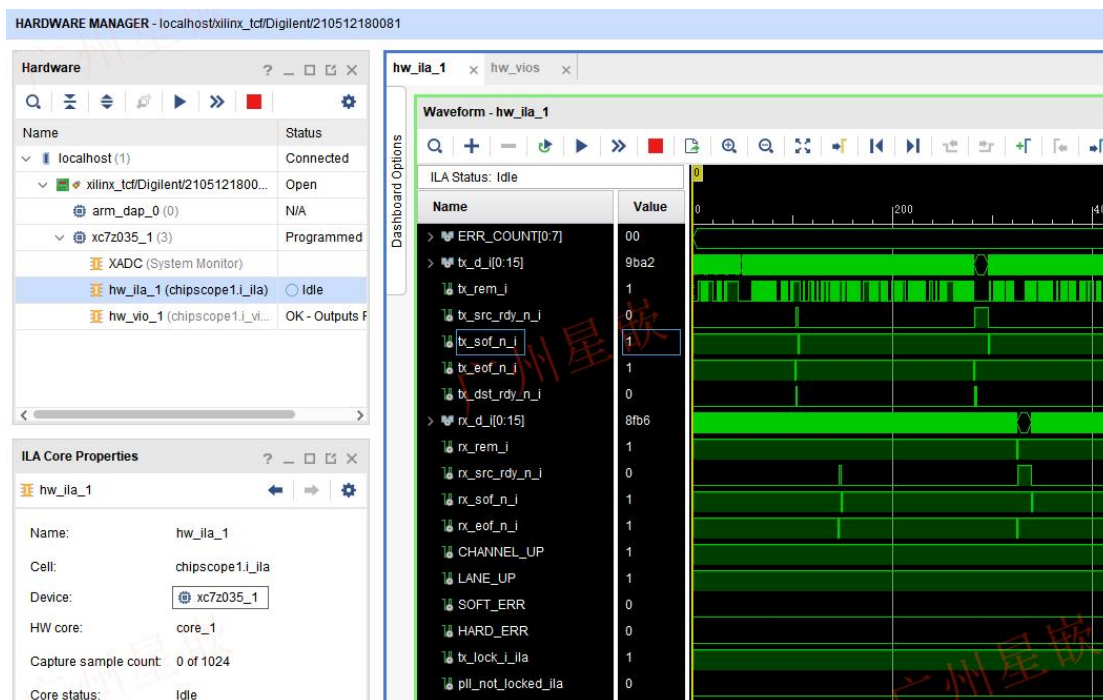
下载 bit 流文件 aurora_8b10b_0_exdes.bit, 并且配套 aurora_8b10b_0_exdes.ltx 调试文件, 如下图下载界面所示:



5.2.4.3 运行结果说明

ZYNQ PL 端提供的 ILA 调试窗口，可以实时抓取采集 GTX 收发本地并行信号以及错误检测信号的时序波形。

ILA 抓取波形如下图所示：



ILA 抓取信号说明如下：

ERR_COUNT[0:7]：接收数据错误个数，接收模块分析接收数据是否正确；

tx_d_i[0:15]：发送数据；

tx_rem_i：最后一个发送数据的高低字节有效标识，0表示，表示tx_d_i[0:7]有效，1表示tx_d_i[0:15]有效；

tx_src_rdy_n_i：发送数据源端准备好标志，结合tx_dst_rdy_n_i使用，都为0时表示可以发送数据；

tx_sof_n_i：发送开始标识，低电平有效；

tx_eof_n_i：发送结束标识，低电平有效；

tx_dst_rdy_n_i：发送数据目的端准备好标志；

rx_d_i[0:15]：接收数据

rx_rem_i：最后一个接收数据的高低字节有效标识，0表示，表示rx_d_i[0:7]有效，1表示rx_d_i[0:15]有效；

rx_src_rdy_n_i：接收数据源端准备好标志；

rx_sof_n_i：接收开始标识，低电平有效；

rx_eof_n_i: 接收结束标识, 低电平有效;

CHANNEL_UP: 为1表示GTX通道完成正常初始化;

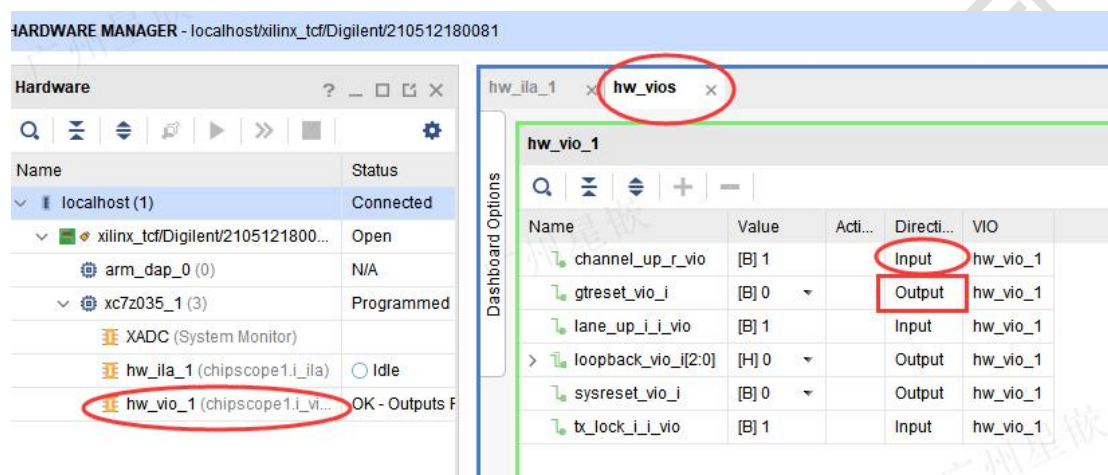
LANE_UP: 指示GTX每个lane是否正常初始化成功, 这里只有1个Lane;

SOFT_ERR、HARD_ERR: 软、硬件错误指示, 正常情况应该为0

tx_lock_i_ila: GTX时钟锁定指示, 正常情况应该为1

pll_not_locked_ila: GTX时钟失锁指示, 正常情况应该为0

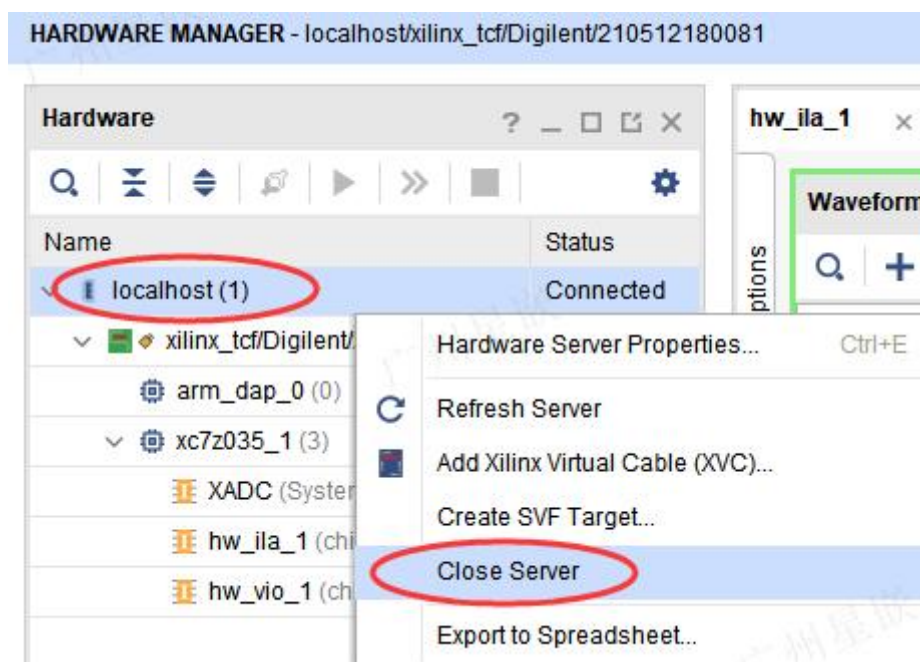
VIO 虚拟 IO 界面如下图所示:



VIO 界面上标识 Input 的为采集信号, 用户只能查看对应信号当前的逻辑电平值, 1 表示高电平, 0 表示低电平; VIO 界面上标识 Output 的为用户控制信号, 用于控制用户逻辑的, 用户可以在 Value 一栏输入 0/1 电平值, 从而达到控制用户逻辑的目的。VIO 界面主要用于复位用户逻辑, 以及查看通道是否链接成功, VIO 界面可以不用操作。

5.2.4.4 退出实验

Vivado 调试界面 Hardware Manager 窗口, 右键单击 localhost(1), 在弹出的菜单中点击 Close Server, 断开 ZYNQ JTAG 仿真器与板卡的连接:



最后，关闭板卡电源，实验结束。

5.3 ZYNQ PL SFP 光口 IBERT 链路误码测试

5.3.1 例程位置

ZYNQ IBERT 链路误码测试例程有两个，分别用于光口运行在 5Gbps 和 10Gbps 两种线路速率情形下的误码统计和眼图测试，IBERT 测试例程保存在资料盘中的位置如下：

(1) 5Gbps IBERT 例程存放位置：

Demo\ZYNQ\PL\ibert_test\ibert_7series_gtx_0_ex_5gbps 文件夹下；

(2) 10Gbps IBERT 例程存放位置：

Demo\ZYNQ\PL\ibert_test\ibert_7series_gtx_0_ex_10gbps 文件夹下。

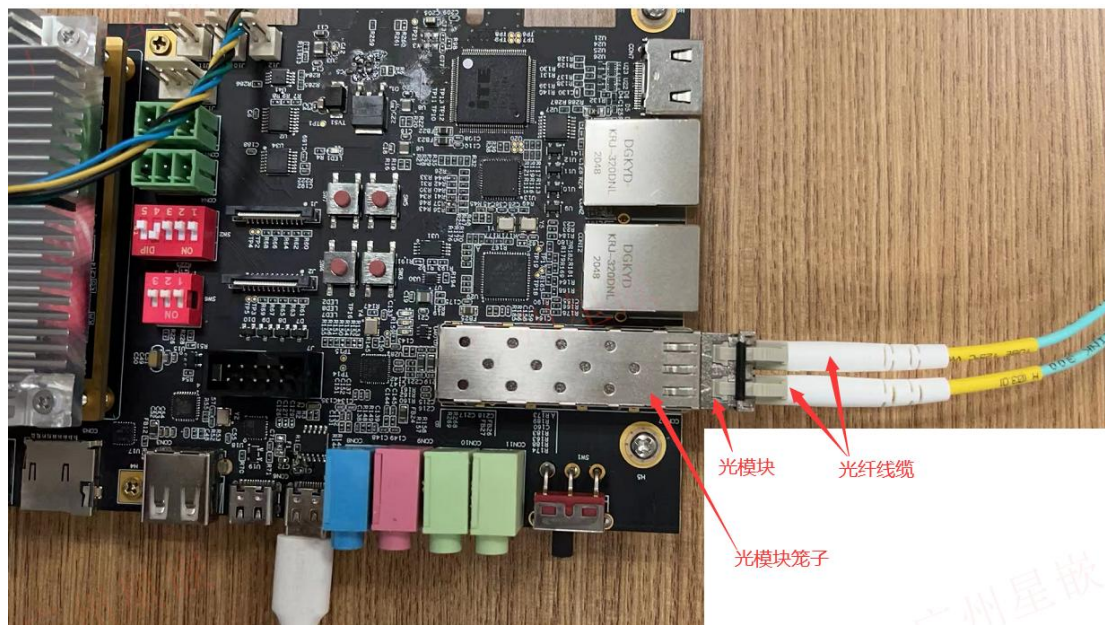
5.3.2 功能简介

LogiCORE IBERT IP 核是 Xilinx 提供的集成式误码率测试 IP 核，该 IP 核产生测试样式，由发送端发出测试样式，经接收端接收测试样式并进行误码检测、分析，以检测 Xilinx 器件内部高速串行收发器的收发性能。由 IBERT IP 生成的测试工程会提供一个图形化测试界面，方便用户直观控制和检测高速串行收发器的参数指标。

5.3.3 例程使用

5.3.3.1 连接光纤模块

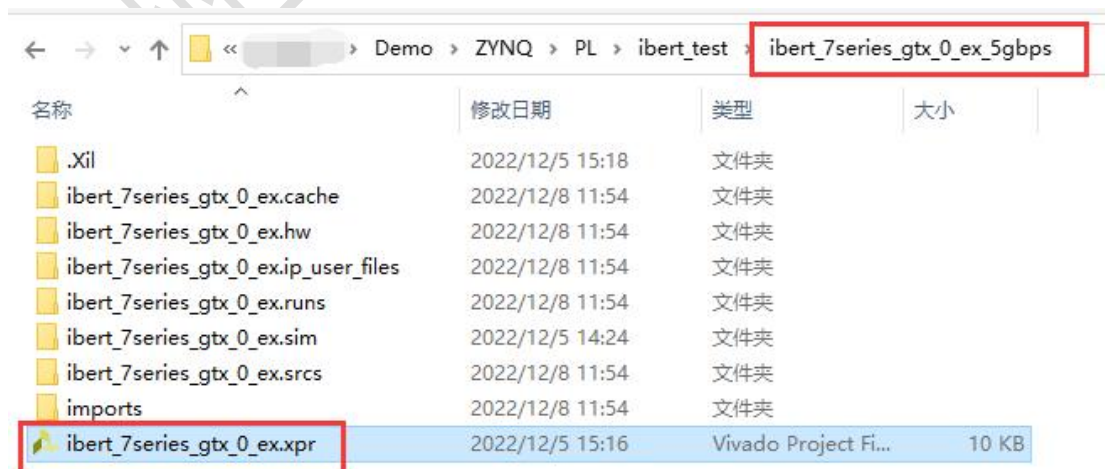
将光模块插入光模块笼子，并使用光纤线缆将光模块的收、发端口自环对接：



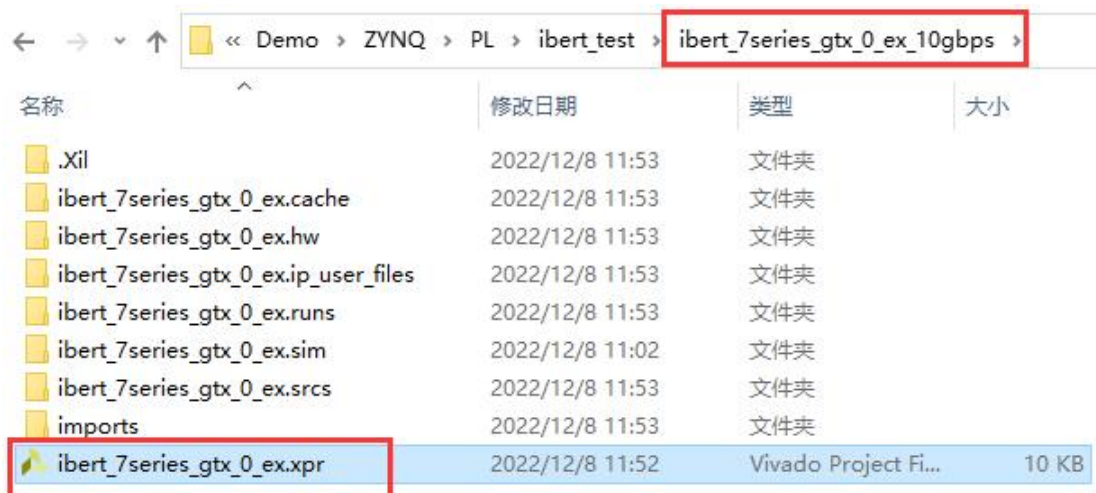
5.3.3.2 加载运行 ZYNQ 程序

5.3.3.2.1 打开 Vivado 工程

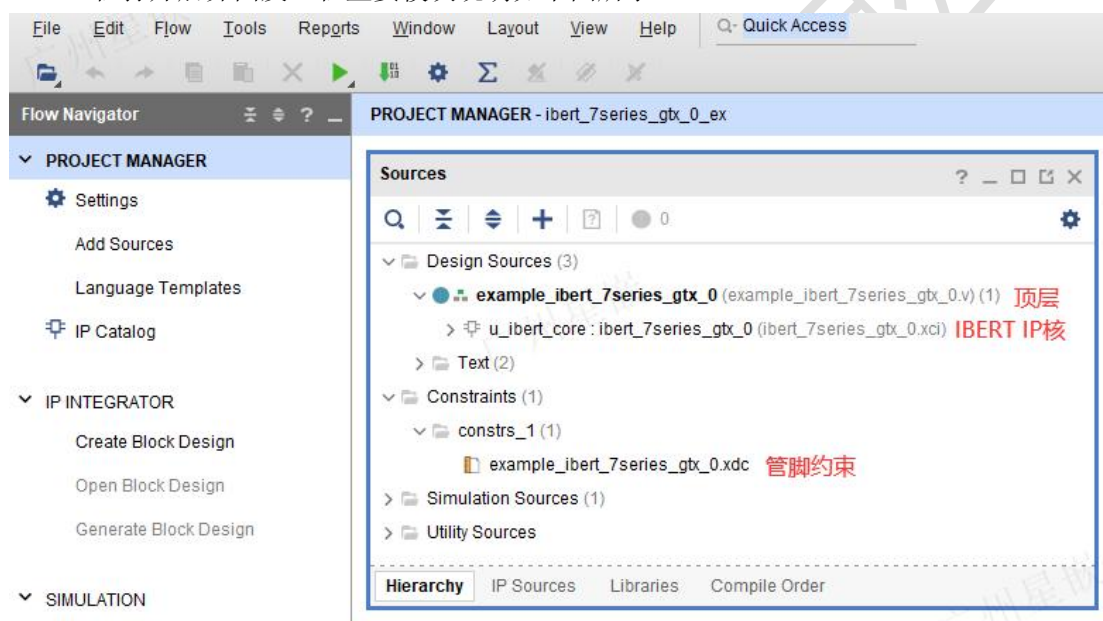
打开 Vivado 示例工程，打开 5Gbps IBERT 例程或 10Gbps IBERT 例程：



或

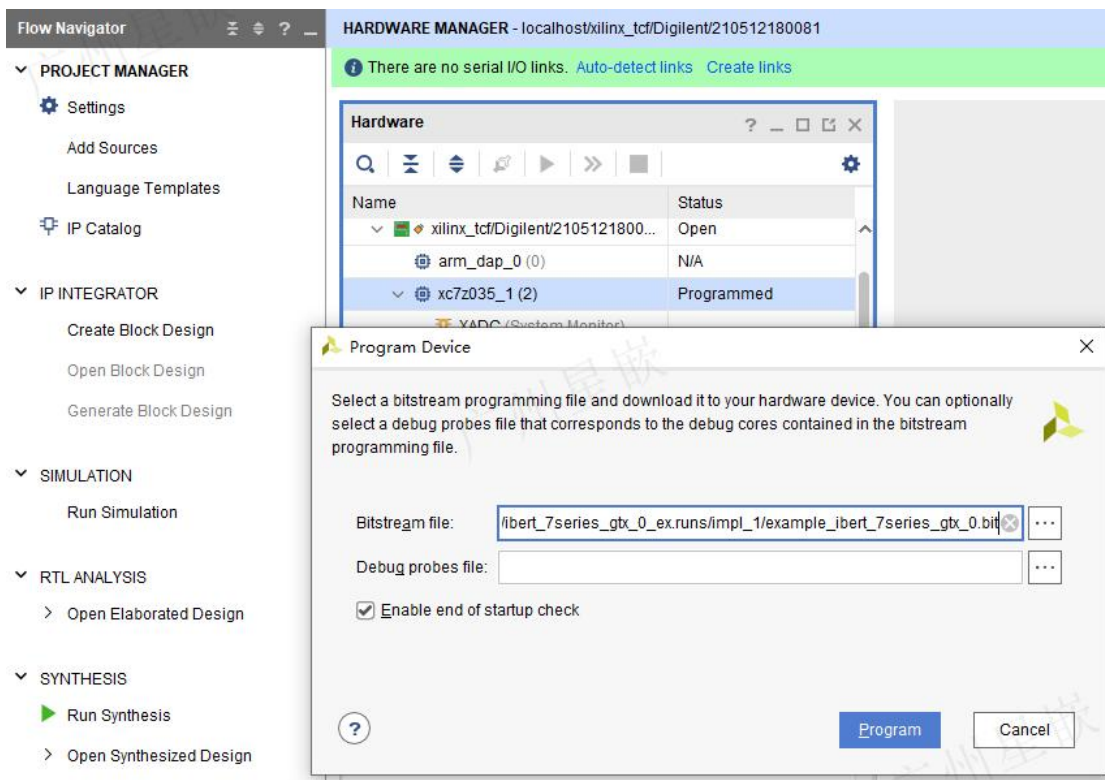


工程打开后界面及工程主要模块说明如下图所示：



5.3.3.2.2 下载 ZYNQ PL 程序

下载 bit 流文件 example_ibert_7series_gtx_0.bit，如下图下载界面所示：

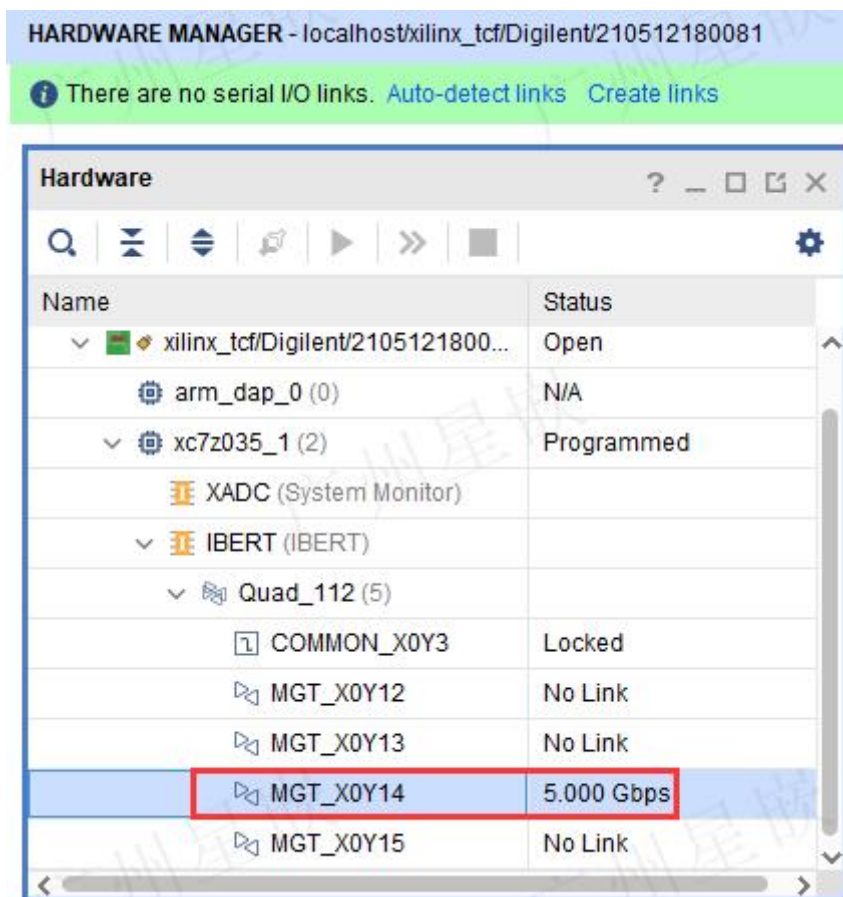


5.3.3.3 运行结果说明

5.3.3.3.1 查看链路状态

5.3.3.3.1.1 5Gbps 速率测试时

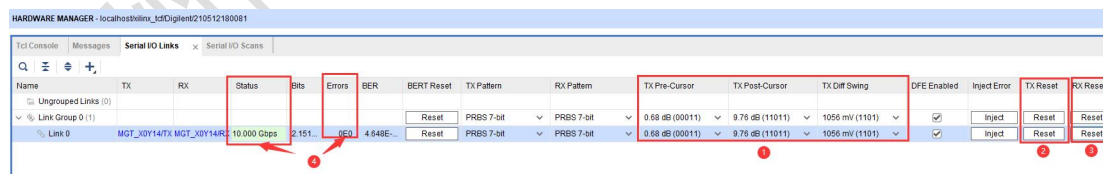
可查看到 MGT_X0Y14 链路锁定在了 5Gbps 线路速率，通信链路已经建立起来：

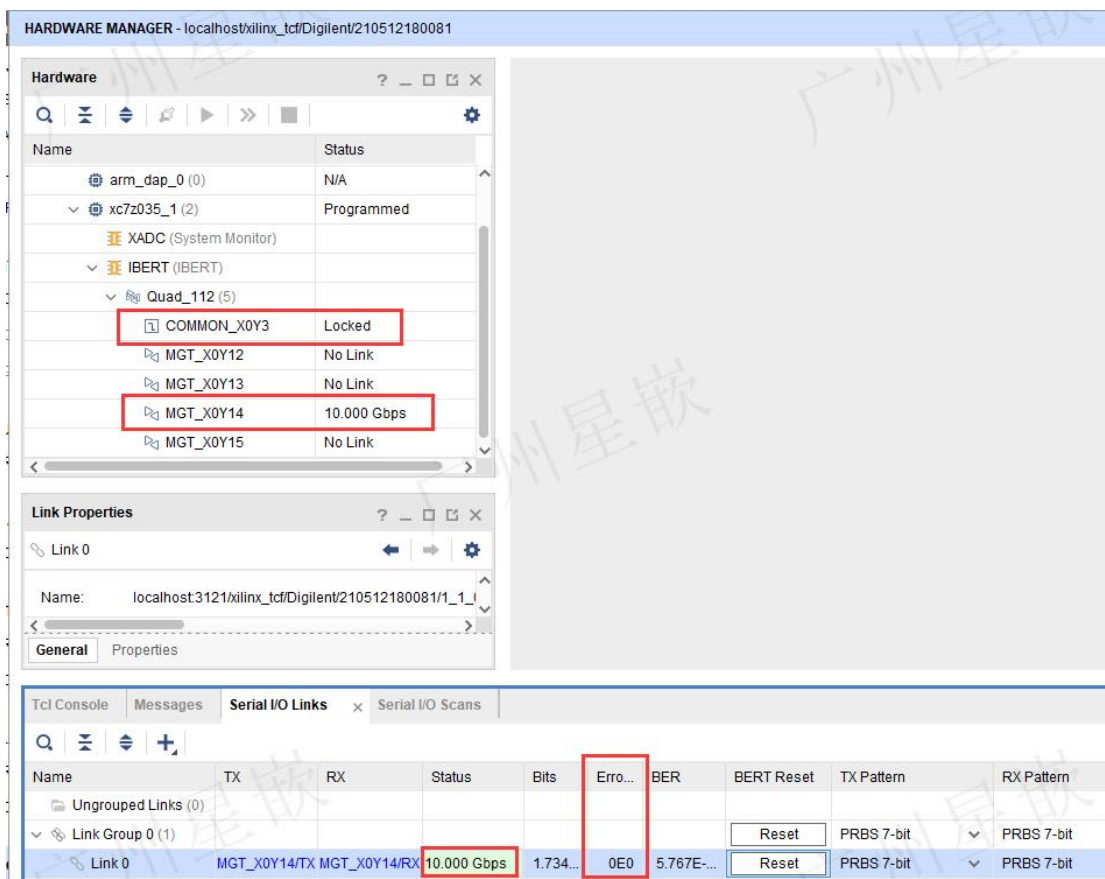


5.3.3.3.1.2 10Gbps 速率测试时

10Gbps 速率测试时，需要用户根据板卡实际情况调节链路参数。在调试窗口的下方，有一个 Serial I/O Links 窗口，打开此窗口去完成链路参数调节任务。

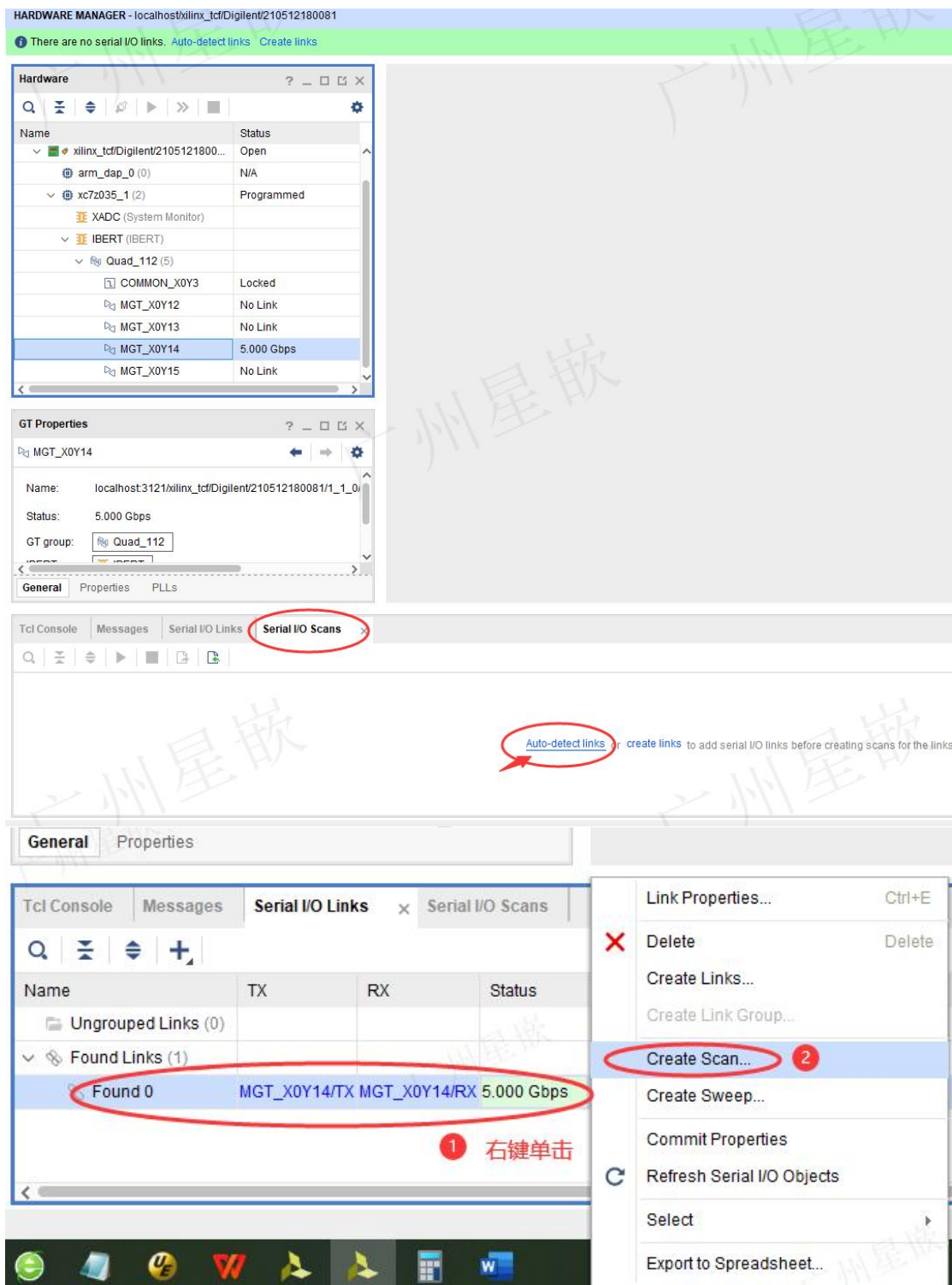
用户调节 Tx Pre-Cursor、Tx Post-Cursor、Tx Diff Swing 这三个链路参数，参数调节完毕后，首先点击 TX Reset，然后点击 RX Reset，最后再查看 Status 和 Errors 状态，直到 Status 显示出预期的线路运行速率，比如 10Gbps，且 Errors=0E0，即误码，则表示链路目前运行正常：





5.3.3.3.2 查看眼图

在 Serial I/O Scans 窗口，点击创建链路，然后再针对链路创建眼图扫描：



设置眼图扫描参数:



Create Scan ×

Set the description and other properties to create and optionally run a scan on the selected link.

Link: Found 0 (MGT_X0Y14/TX, MGT_X0Y14/RX)

Description: ×

Scan Properties

Scan type:	<input type="text" value="2D Full Eyescan"/> ▼
Horizontal increment:	<input type="text" value="8"/> ▼
Horizontal range:	<input type="text" value="-0.500 UI to 0.500 UI"/> ▼
Vertical increment:	<input type="text" value="8"/> ▼
Vertical range:	<input type="text" value="100%"/> ▼

Dwell

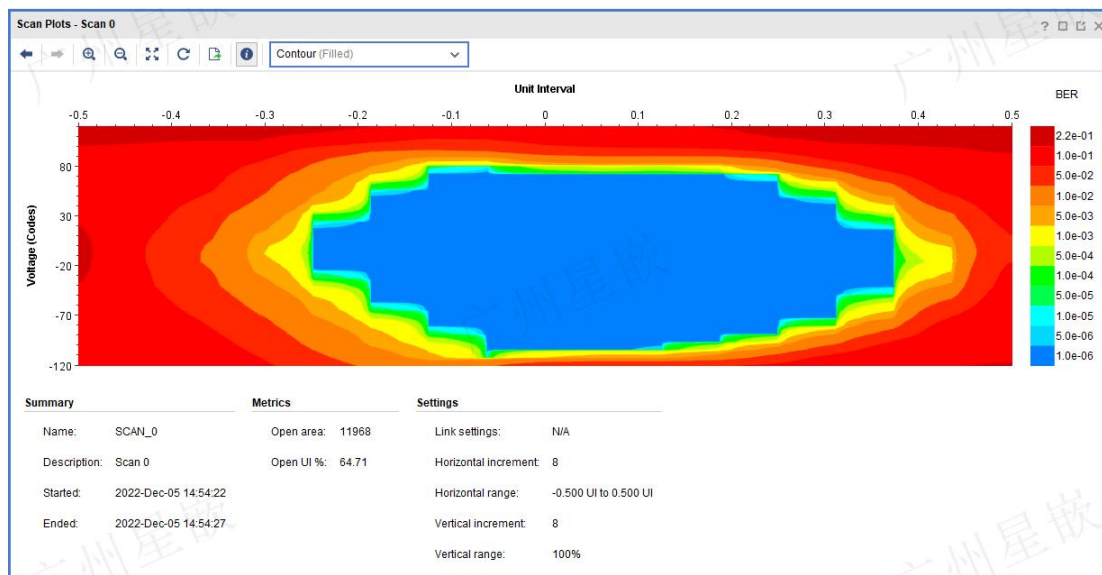
BER: ▼

Time: ▲▼

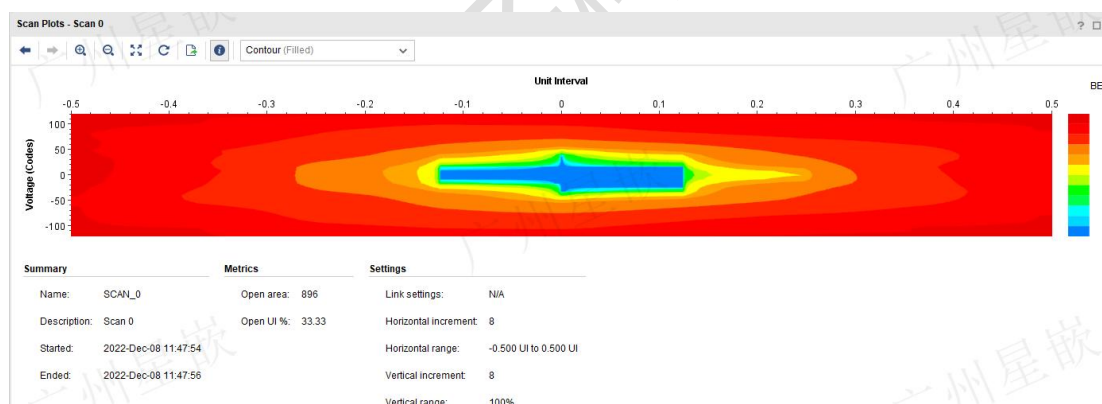
Run scan

5.3.3.3.2.1 眼图测试结果

5.3.3.3.2.1.1 15Gbps 速率测试时



5.3.3.3.2.1.2 10Gbps 速率测试时



5.3.3.4 退出实验

Vivado 调试界面 Hardware Manager 窗口，右键单击 localhost(1)，在弹出的菜单中点击 Close Server，断开 ZYNQ JTAG 仿真器与板卡的连接。

最后，关闭板卡电源，实验结束。

5.4 ZYNQPL M.2 接口验证例程

5.4.1 例程位置

ZYNQ 例程保存在资料盘中的 Demo\ZYNQ\PL\pcie_7x_0_ex 文件夹下。

5.4.2 功能简介

使用 7 Series Integrated Block for PCI Express IP 核 PCIe RC (PCIe 根复合体) 例子工程, 即 PCIe 配置示例工程, 该示例工程执行 PCIe RC 端配置事务, 枚举和配置 PCIe EP 端的配置空间, 同时也可产生用户数据包交互。

PCIe 配置示例工程数据流和实现框图如下图所示, PCIe 配置示例工程实现的是图中 Configurator Example Design 部分, PIO Master 为用户数据包产生逻辑, 可由用户自行修改, 以便产生所需的 PCIe TLP 事务包:

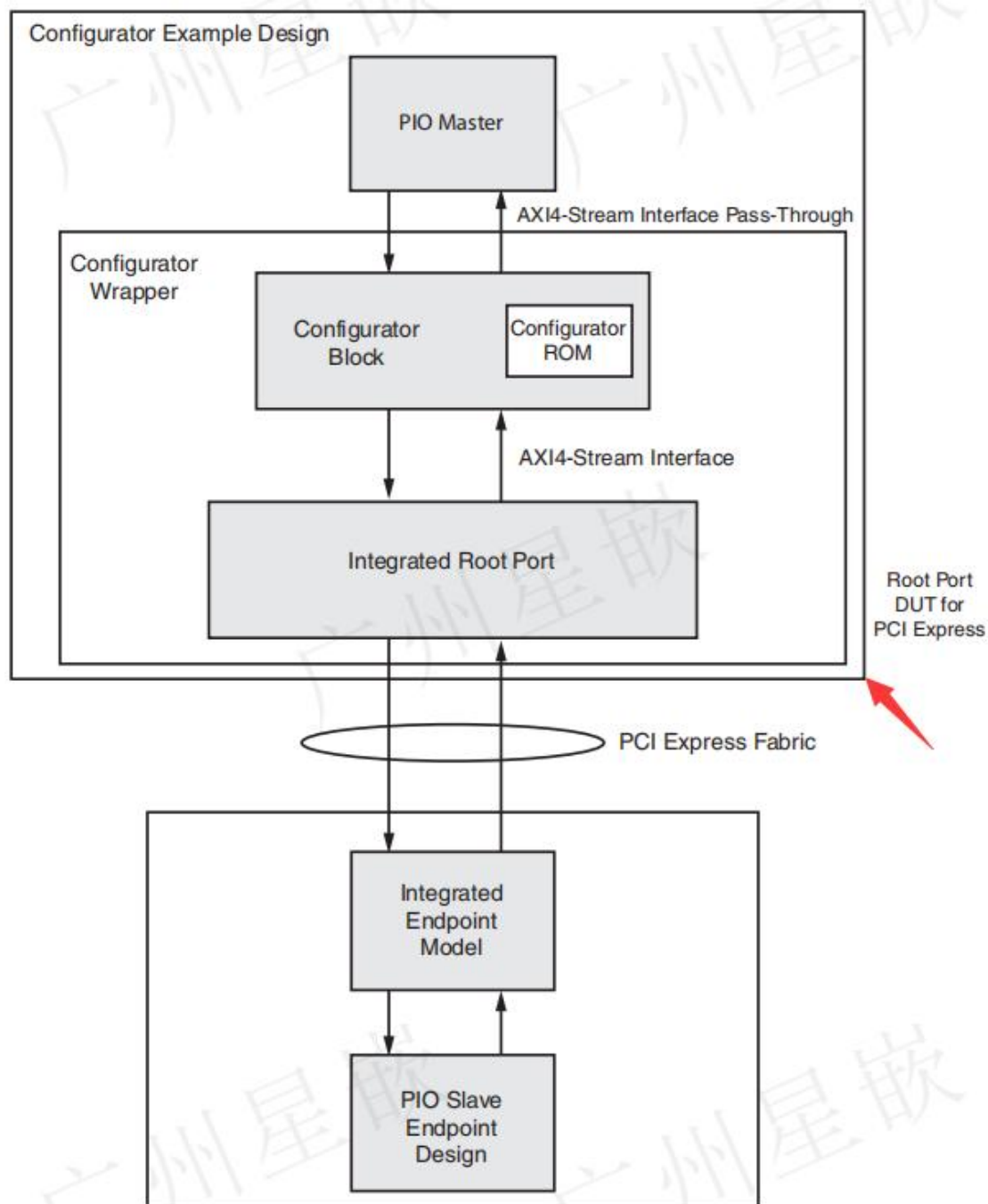


图 PCIe RC 配置示例工程数据流和框图

7 Series Integrated Block for PCI Express IP 核参数设置如下图所示:

7 Series Integrated Block for PCI Express (3.3)

Documentation IP Location Switch to Defaults

Show disabled ports

- + s_axis_tx
- + pipe_clock
- + drp
- + pcie2_cfg_err
- + pcie2_cfg_interrupt
- + pcie2_cfg_control
- + pcie2_pl
- sys_clk
- sys_rst_n
- pcie_drp_clk
- m_axis_rx
- pcie_7x_mgt
- pcie2_cfg_status
- pcie_cfg_fc
- pcie2_msg_rcvcd
- user_clk_out
- user_reset_out
- user_lnk_up
- user_app_rdy

Component Name: pcie_7x_0

Basic | IDs | BARS | Core Capabilities | Interrupts

Mode: Basic

Device Port Type: Root Port of PCI Express Root Complex

PCIE Block Location: X0Y0

Number of Lanes: Lane Width: X2

Maximum Link Speed: 5.0 GT/s

AXI Interface Frequency: 125 MHz

AXI Interface Width: 64 bit

Reference Clock Frequency (MHz): 100 MHz

Tandem Configuration: None

PIPE Mode Simulations: None

其他参数默认即可。

5.4.3 管脚约束

ZYNQ PL 工程管脚约束如下图所示：

ELABORATED DESIGN - xc7z035ffg676-2 (active)

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std
All ports (12)						
pcie_7x_mgt_23453 (8) (Multiple)						
RXP (4)						
RXP[1]	IN	RXN	Y4	✓	112	
RXP[0]	IN	RXN[0]	AB4	✓	112	
TXP (4)						
TXP[1]	OUT	TXN	W2	✓	112	
TXP[0]	OUT	TXN[0]	AA2	✓	112	
Scalar ports (0)						
Scalar ports (4)						
clk_in1_p	IN	clk_in1_n	J4	✓	33	DIFF_HSTL_IL_18
sys_clk_p	IN	sys_clk_n	R6	✓	112	

5.4.4 例程使用

5.4.4.1 插入 NVMe 硬盘

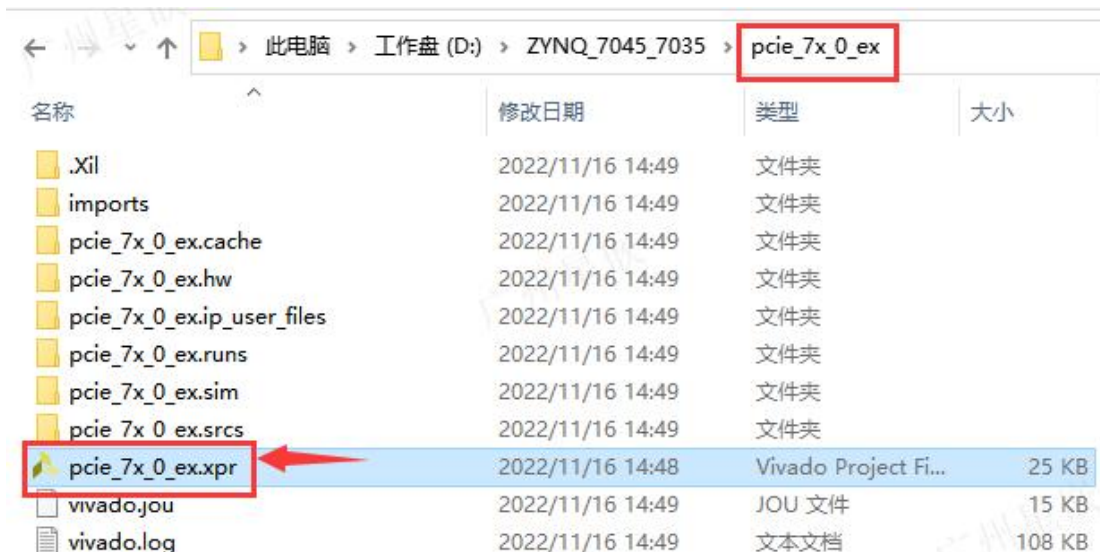
将一块 NVMe 硬盘插入底板背面的 M.2 接口插槽上，并拧上螺丝固定硬盘：



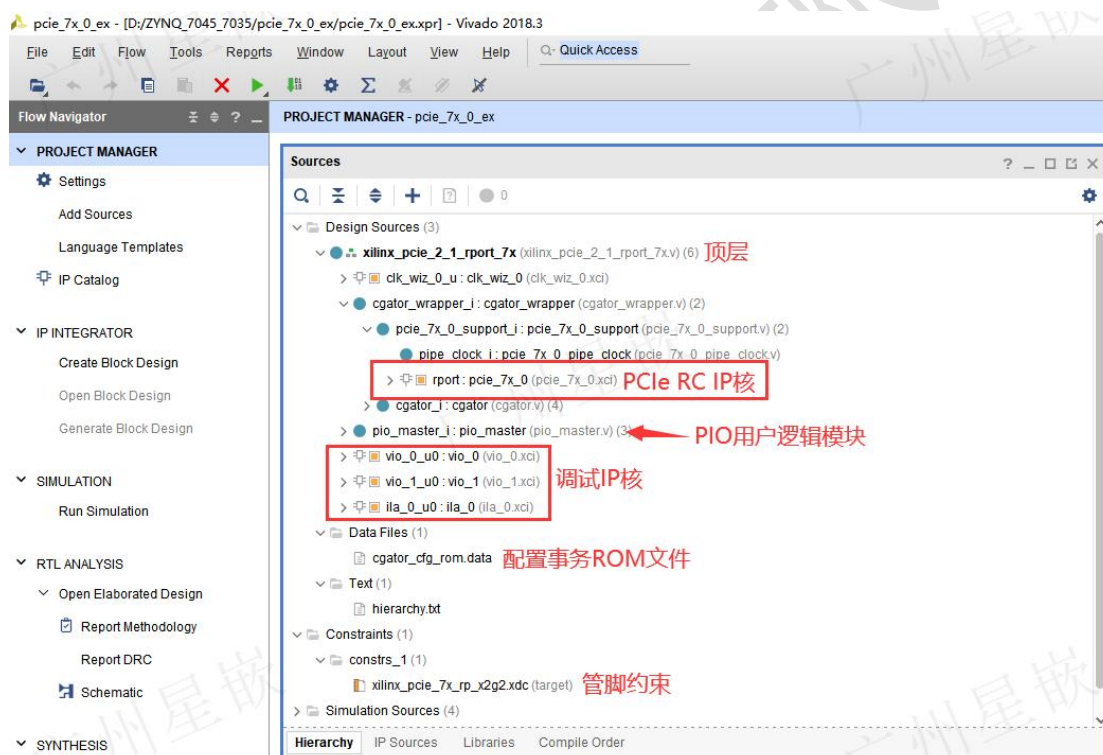
5.4.4.2 加载运行 ZYNQ 程序

5.4.4.2.1 打开 Vivado 工程

打开 Vivado 示例工程：

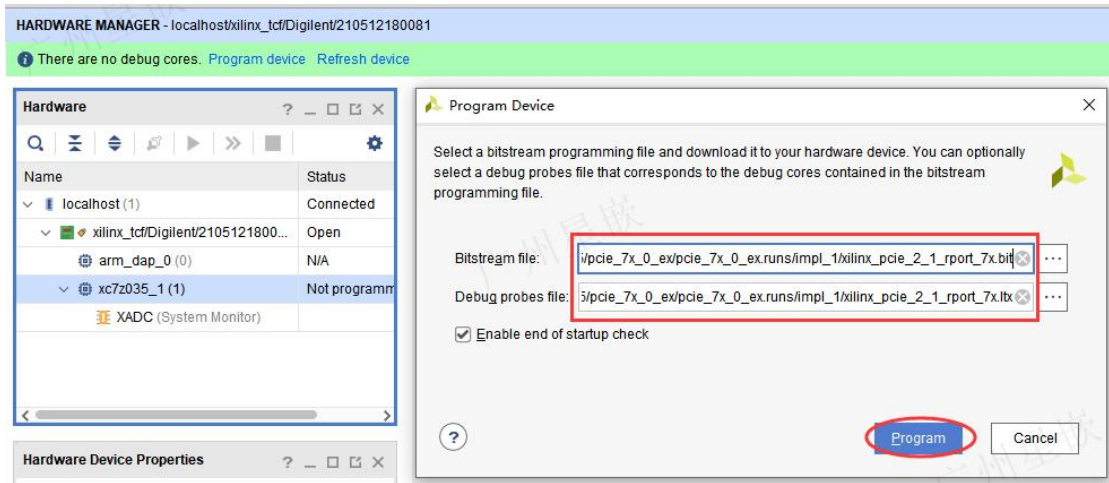


工程打开后界面及工程主要模块说明如下图所示：



5.4.4.2.2 下载 ZYNQ PL 程序

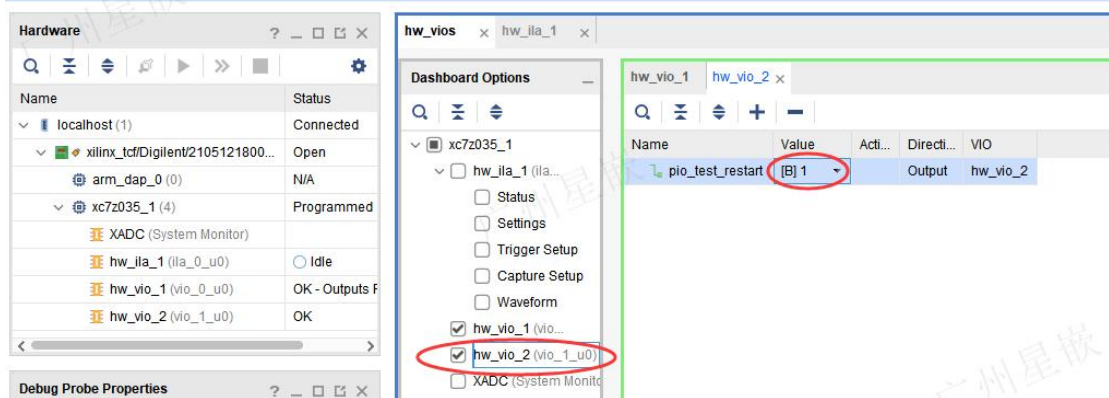
下载 bit 流文件 xilinx_pcie_2_1_rport_7x.bit，并且配套 xilinx_pcie_2_1_rport_7x.ltx 调试文件，如下图下载界面所示：



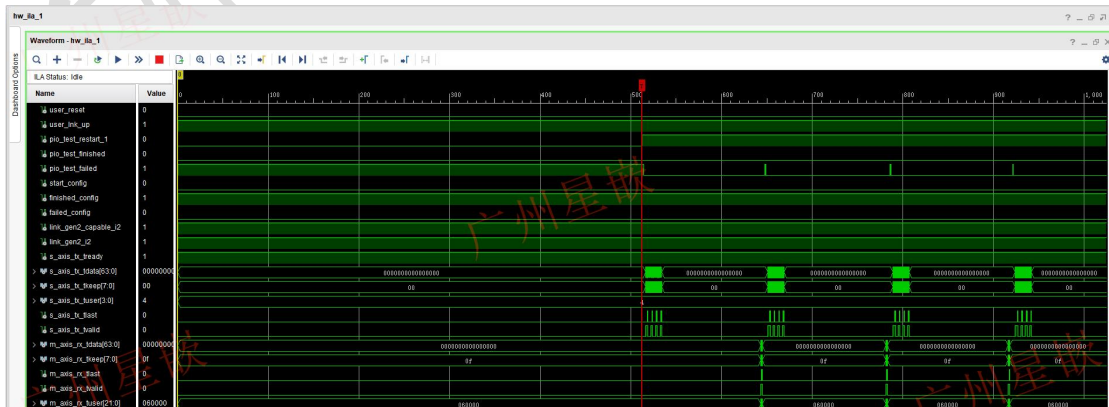
5.4.4.3 运行结果说明

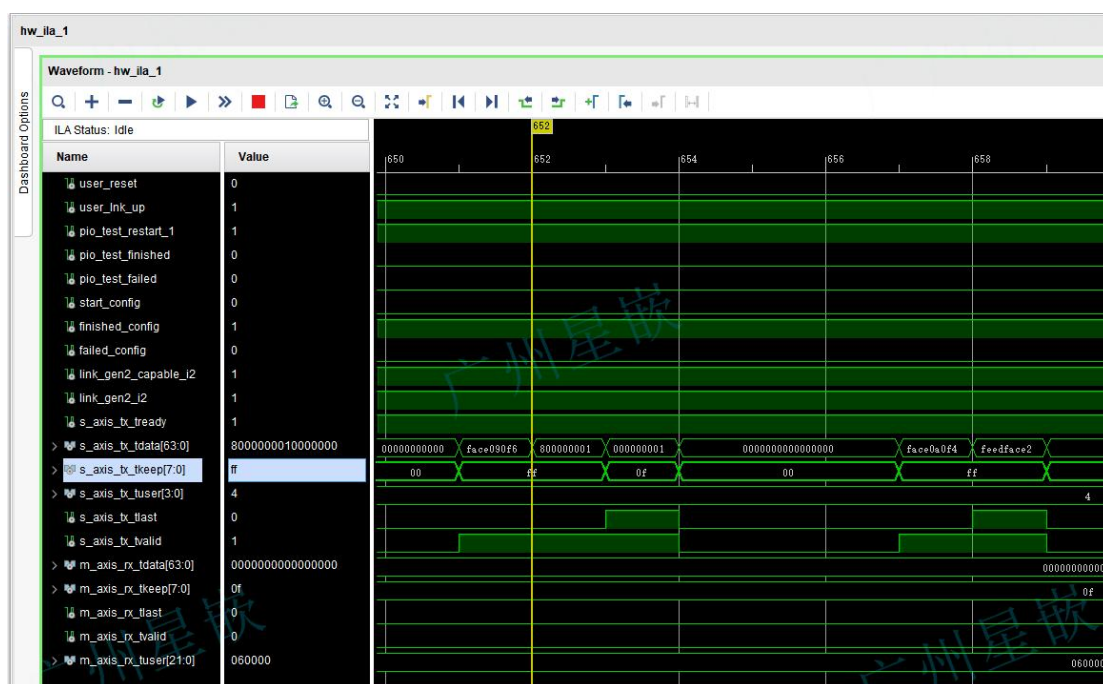
ZYNQ PL 端提供的 ILA 和 VIO 调试窗口，ILA 可以实时抓取采集 PIO 用户逻辑信号的时序波形，VIO 可以控制 PIO 事务的重启。

hw_vio_2 里面的 `pio_test_restart` 控制信号用来控制 PIO 事务重启操作，先设置为 0 值，再设置为 1 值，可以启动一次 PIO 事务：



ILA 抓取波形如下图所示：





ILA 抓取的关键信号说明如下：

user_reset: 事务层用户复位信号，正常为0；

user_lnk_up: 事务层Link up信号，正常为1；

pio_test_restart_1: PIO事务测试启动信号，0到1启动一次PIO事务测试，主要是读写PCIe EP端的用户BAR存储空间，如果挂载NVMe硬盘测试的话，PIO事务测试将无意义，因为NVMe硬盘上可能并未开辟对应用户BAR存储空间；

pio_test_finished: PIO事务测试成功完成标识，正常为1，只有对应的PCIe EP端开辟了对应的用户BAR存储空间才有意义；

pio_test_failed: PIO事务测试失败标识，正常为0，只有对应的PCIe EP端开辟了对应的用户BAR存储空间才有意义；

start_config: PCIe配置启动信号，1表示启动PCIe配置事务过程，0表示结束配置或不在配置阶段；

finished_config: PCIe配置成功完成标识，正常为1；

failed_config: PCIe配置失败标识，正常为0；

link_gen2_capable_i2: 速率支持指示，1表示支持PCIe Gen2协议速率，即5Gbps，0表示不支持PCIe Gen2协议速率；

link_gen2_i2: 当前PCIe链路速率，0表示当前PCIe链路速率为2.5 Gb/s，1表示当前PCIe链路速率为5.0 Gb/s；

s_axis_tx_xxx: TLP事务用户发送数据和控制接口；

m_axis_rx_xxx: TLP事务用户接收数据和控制接口。

5.4.4.4 退出实验

Vivado 调试界面 Hardware Manager 窗口，右键单击 localhost(1)，在弹出的菜单中点击 Close Server，断开 ZYNQ JTAG 仿真器与板卡的连接。

最后，关闭板卡电源，实验结束。

5.5 ZYNQPL M.2 接口之 NVMe Host IP 例程说明

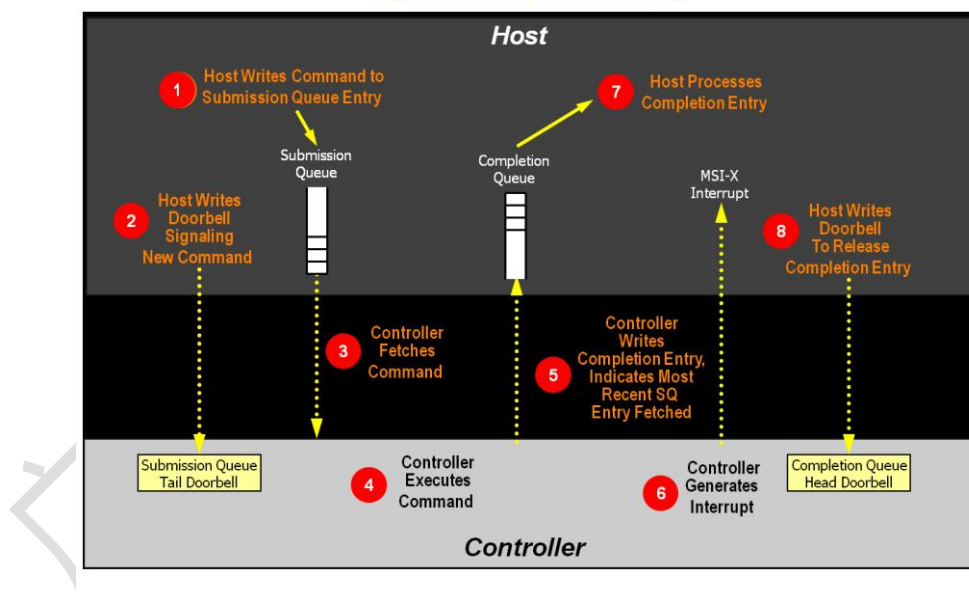
5.5.1 例程声明

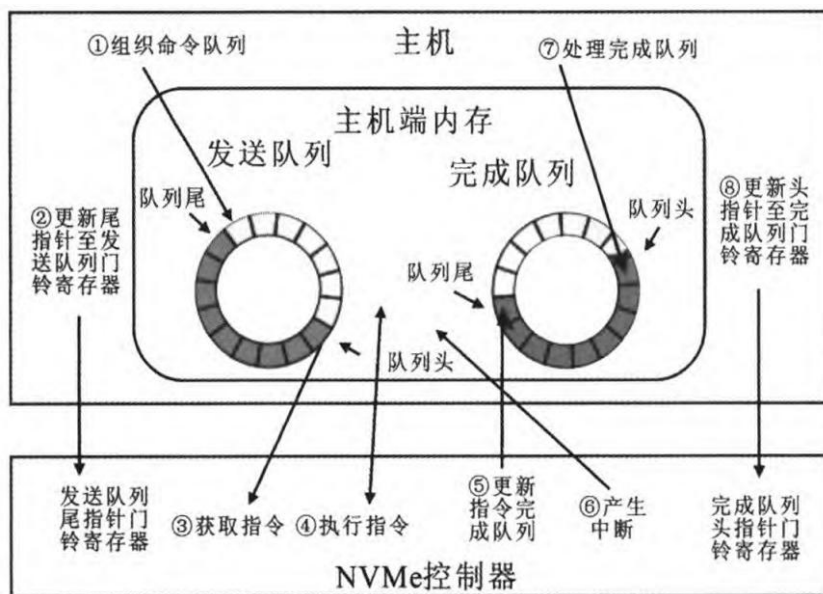
此例程涉及到收费 IP 核，因此只对例程进行介绍性说明，如需要详细例程和对应 IP 核资料，请联系市场洽谈。

5.5.2 设计目的

使用 FPGA 实现标准 NVMe 1.3 协议的 Host 端，即纯逻辑实现 NVMe Host IP。

Figure 245: Command Processing





用户可以根据自身需求，定制 NVMe Host FPGA IP，还可以在此基础上定制 FPGA 纯逻辑来实现文件系统 ExFAT。

NVMe Host FPGA IP 核使用 VHDL 硬件描述语言的纯逻辑方式来实现，NVMe 物理层使用 Xilinx 7 系列 PCIe 核，无需 CPU 参与。目前已在广州星嵌电子 ZYNQ+DSP 平台上实现并经充分测试验证：PCIe 2.0 x2 Lane 平台，连续读速率：685 MB/s，连续写速率：531 MB/s，日后升级可能还有提升空间。后续广州星嵌电子会提供 PCIe 2.0 x4/x8 Lane 平台，以获取更高的硬盘读写带宽。

5.5.3 NVMe Host FPGA IP 核简介

NVMe Host FPGA IP 访问接口简单，用户可将此 IP 当作双端口 RAM 来使用，只是相对普通双端口 RAM 而言多增加了一些读、写命令握手接口信号而已。当然，用户还可将 IP 定制为自身所需要的接口形式。

NVMe Host FPGA IP 对外接口图如下：

NVMe_Host_Component	
NVMe_Host_Reset_N	
NVMe_SSD_Access_Go	
NVMe_SSD_Access_NLB[15:0]	Index_NVMe_SSD_Read_Data[11:0]
NVMe_SSD_Access_SLBA[63:0]	NVMe_Host_Ready
NVMe_SSD_Access_Type[1:0]	NVMe_Host_User_CLK
NVMe_SSD_Read_Clock	NVMe_SSD_Access_Area
NVMe_SSD_Read_Configure_Start[11:0]	NVMe_SSD_Access_Done
NVMe_SSD_Read_Configure_Stop[11:0]	NVMe_SSD_Access_Status[5:0]
NVMe_SSD_Read_Go	NVMe_SSD_Read_Area
NVMe_SSD_Write_Address[11:0]	NVMe_SSD_Read_Data[31:0]
NVMe_SSD_Write_Clock	NVMe_SSD_Read_Data_Valid
NVMe_SSD_Write_Data[31:0]	NVMe_SSD_Read_Done
NVMe_SSD_Write_Enable	PCIE_TXN[3:0]
PCIE_RXN[3:0]	PCIE_TXP[3:0]
PCIE_RXP[3:0]	sl_oport0[0:16]
PCie_CLK	sl_oport0_1[0:16]

名词和概念解释:

SLBA (Starting LBA): 逻辑块基地址, 数据在 SSD 固态硬盘上的起始地址。

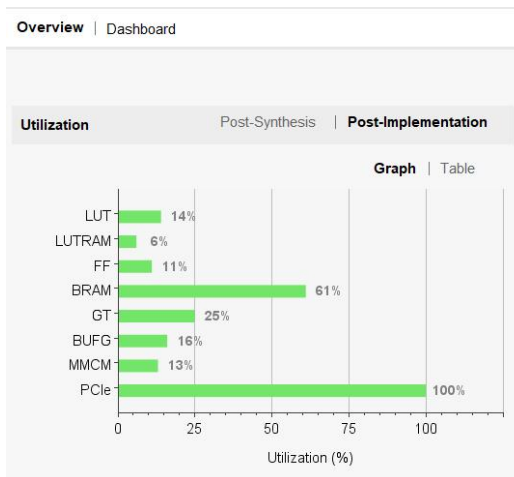
NLB (Number of Logical Blocks): 逻辑块数量, 指定数据传输大小, 需要指出的是这是一个 0 基数值, 即以 0 为初始值, 所以最终传的是(NLB+1)个逻辑块。

NVMe Host FPGA IP 支持最大队列数 64 对, 最大队列深度 16383, 用户可以根据需要进行定制。NVMe Host FPGA IP 核源码实现了参数化, 可根据用户功能及性能 (比如队列数与队列深度) 对 IP 参数进行修改设置。

NVMe Host FPGA IP 使用的 FPGA 软件开发平台为 Vivado 2018.3, 用户可以定制为自身所需的 FPGA 软件开发平台上。

5.5.4 资源消耗

广州星嵌电子 ZYNQ+DSP 平台上使用 ZYNQ 7035 PL 端作为 FPGA 开发平台, ZYNQ 芯片型号为 XC7Z035FFG676-2。NVMe Host FPGA IP 在 XC7Z035FFG676-2 芯片上的资源消耗报表如下:

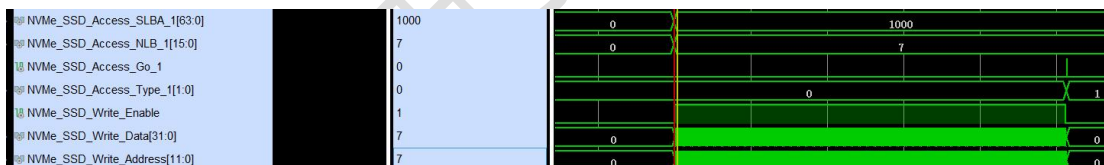


(注：此为第一版 IP 资源消耗图表，还有优化空间)

5.5.5 NVMe Host FPGA IP 测试截图和说明

借助 NVMe Host FPGA IP，往 NVMe SSD 固态硬盘上写入测试数据（例程使用的是累加数），然后读出，并在 FPGA 上使用逻辑进行比对，并给出比对结果，以验证 NVMe 硬盘读写数据是否一致。

5.5.5.1 单次写 8 个扇区

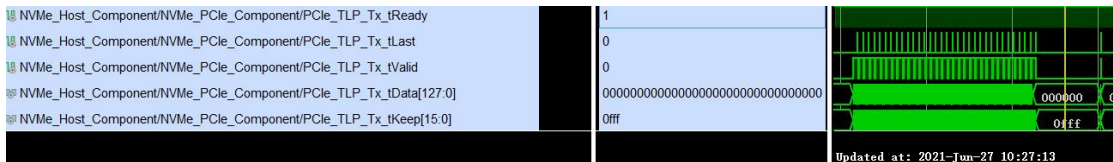


注：NLB = 7，即逻辑块数量 8。

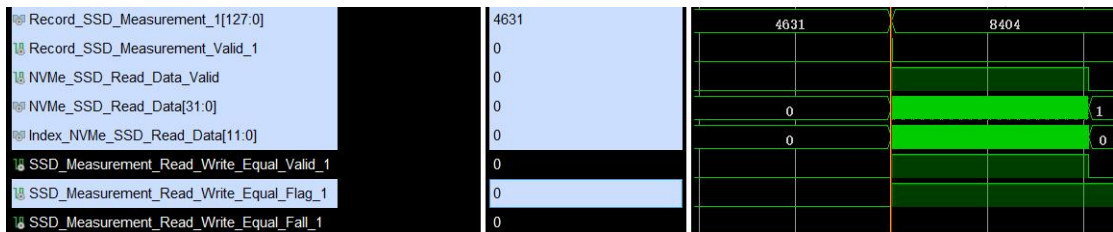
数据波形放大查看，可以看到写数据与写地址相同，写数据为累加数：



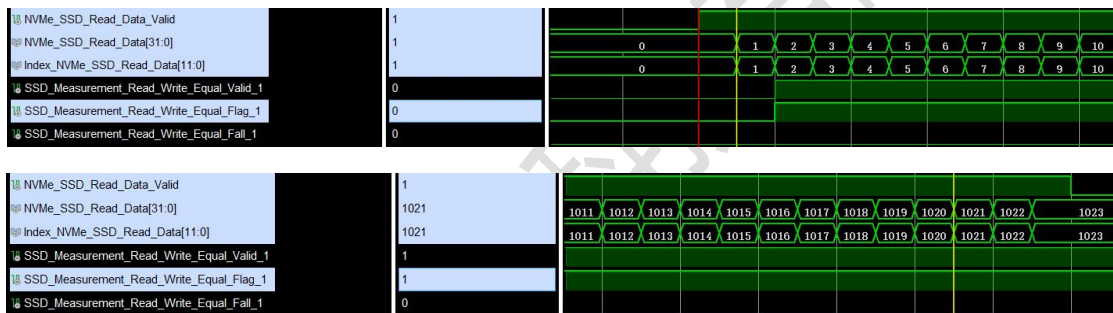
单次写 8 个扇区对应的 PCIe 底层时序波形如下图所示：



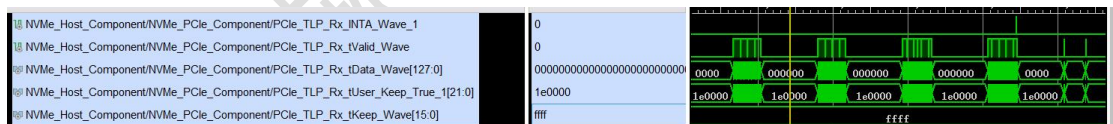
5.5.5.2 单次读 8 个扇区



对上面数据波形放大，可发现读数据与读地址相同，读数据为累加数：

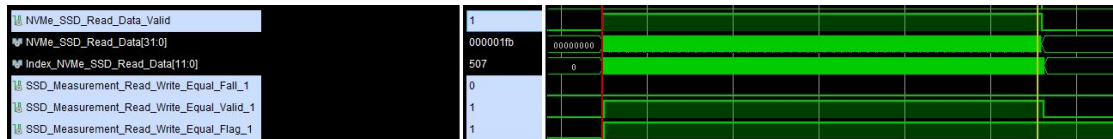


单次读 8 个扇区对应的 PCIe 底层时序波形如下图所示：



5.5.5.3 扇区读写比对测试

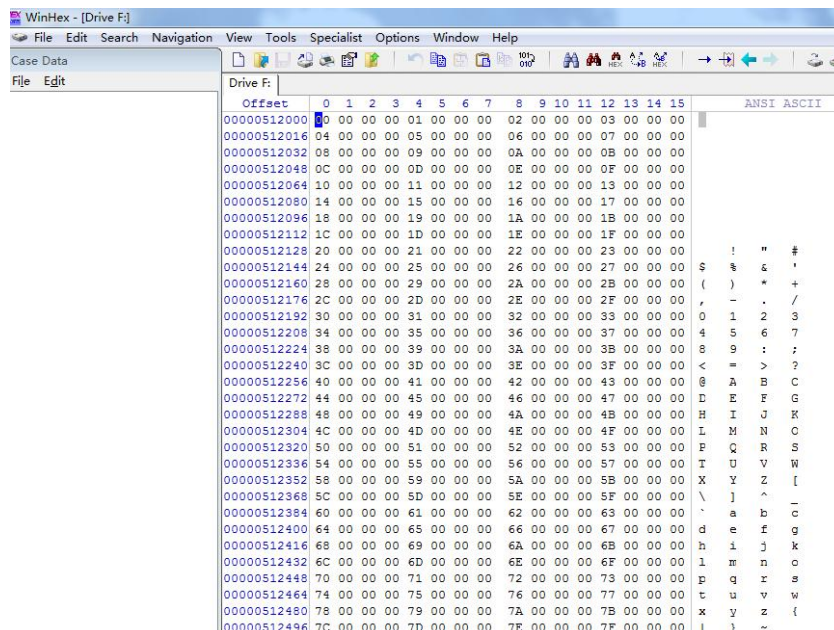
比对结果如下图高亮信号所示：



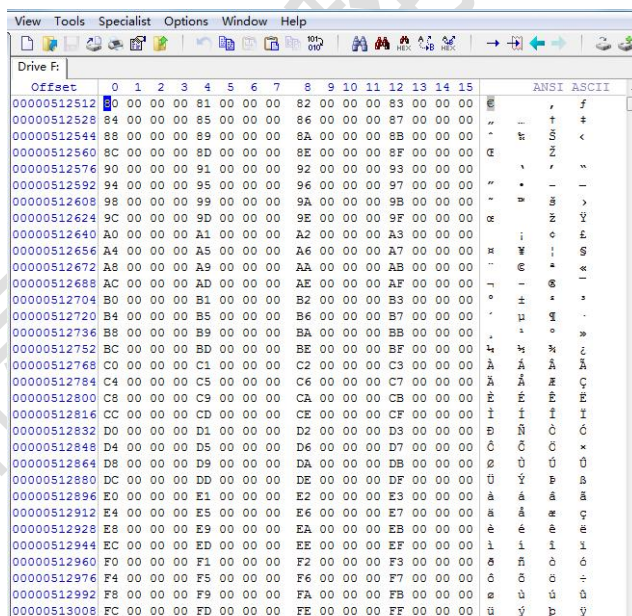
- Equal_Fail: SSD 硬盘读写数据不一致标识，1 表示读写数据不一致；
- Equal_Valid: 读写数据比对结果有效标识，1 表示读写对比结果有效；
- Equal_Flag: SSD 硬盘读写数据一致标识，1 表示读写数据一致。

NVMe SSD 硬盘读写测试完后，从测试平台上取下 NVMe SSD 固态硬盘，并将 SSD 硬盘连接至 PC 电脑，使用 WinHex 软件工具分析 NVMe SSD 固态硬盘的写入数据。

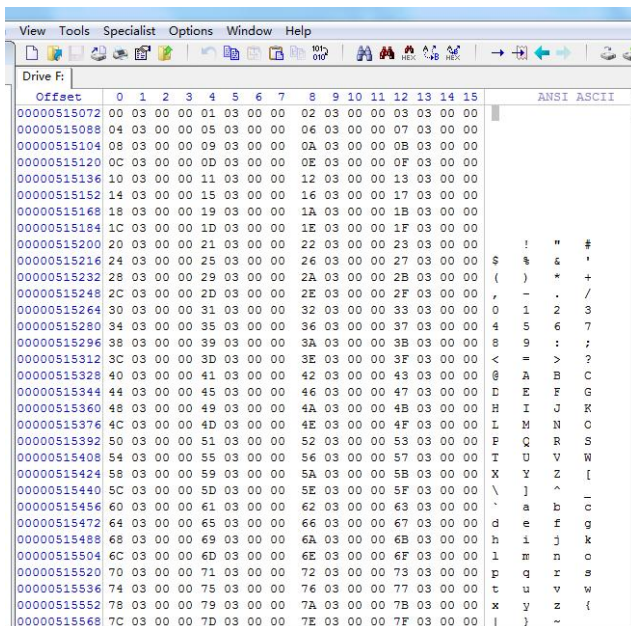
扇区 1 写入数据：



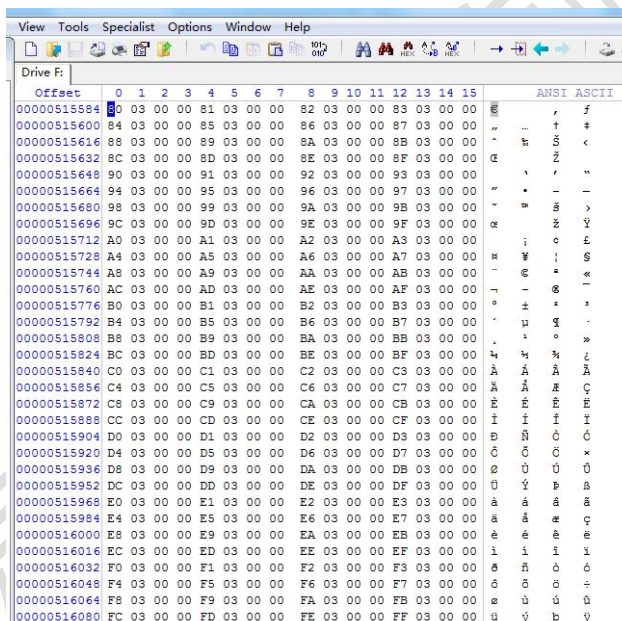
扇区 2 写入数据：



扇区 3 写入数据：



扇区 8 写入数据：



5.5.5.4 指定扇区某位置写入指定值

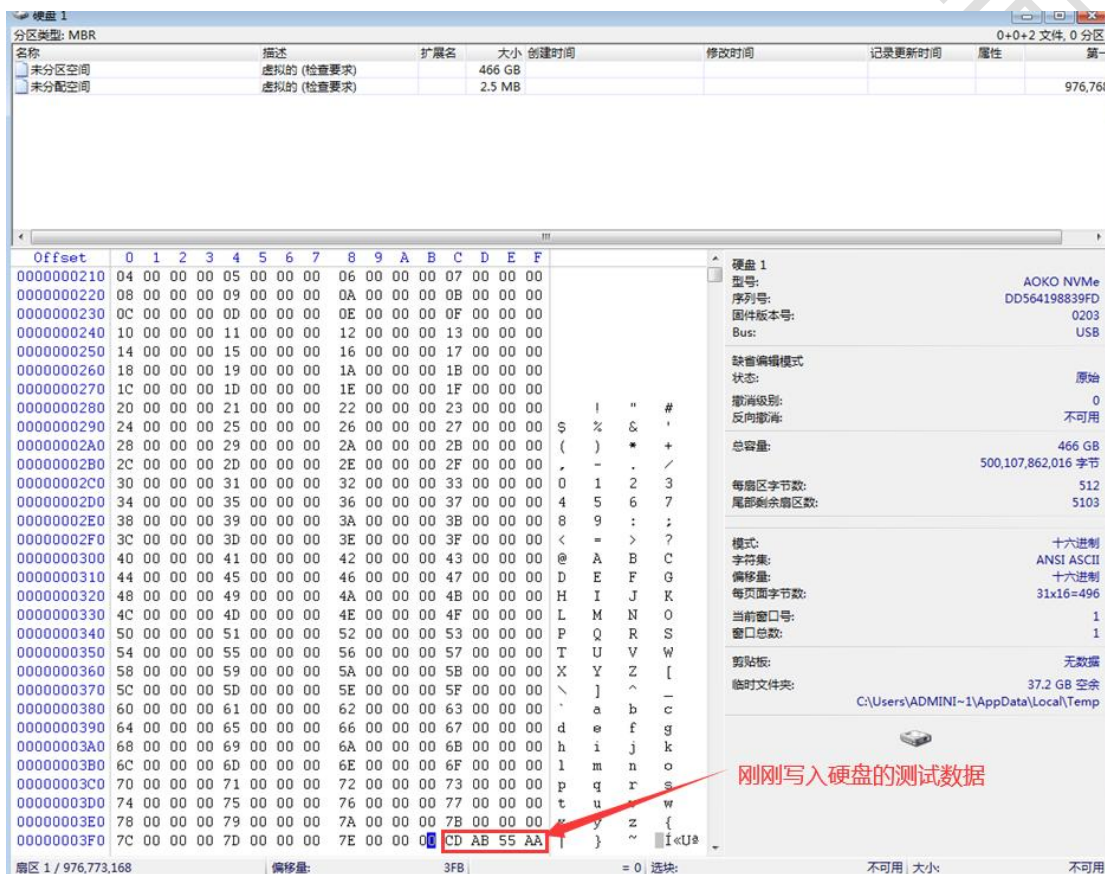
指定往 SSD 硬盘的第 1 扇区偏移地址 127 中写入数据 0xAA55_ABCD，FPGA VIO 设置如下图所示：

NVMe_SSD_Access_SLBA_VIO[63:0]	[H] 0000_0000_0000_0001	Output
NVMe_SSD_Access_NLB_VIO[15:0]	[H] 0001	Output
NVMe_SSD_Access_Command_VIO	[B] 0	Output
NVMe_SSD_Access_Type_VIO[1:0]	[H] 1	Output
SSD_Write_VIO_Enable	[B] 1	Output
SSD_Write_Command_VIO	[B] 0	Output
SSD_Write_Address_VIO[11:0]	[U] 127	Output
SSD_Write_Data_VIO[31:0]	[H] AA55_ABCD	Output

指定从 SSD 硬盘的第 1 扇区偏移地址 127 中读出数据，读数据为 0xAA55_ABCD，与写入数据一致，读时序和读写比对结果如下图所示：

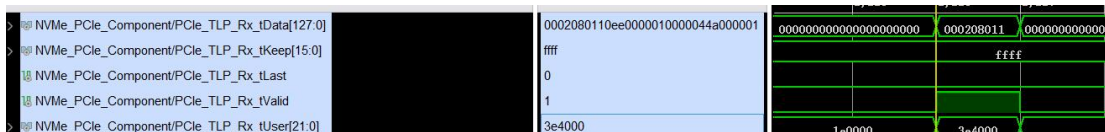


从测试平台上取下 NVMe SSD 固态硬盘，并将硬盘连接至 PC 电脑，使用 WinHex 软件工具分析刚刚写入硬盘的测试数据，如下图所示，也可发现确实已经将 0xAA55_ABCD 用户数据写入了硬盘：

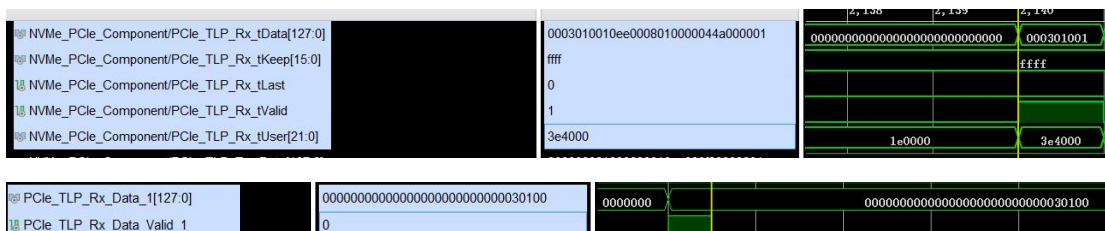


5.5.5.5 NVMe 协议相关寄存器读取

NVMe 协议码：0x010802，对应的 PCIe 底层读时序波形如下图所示：



NVMe 协议版本 1.3：0x00010300，对应的 PCIe 底层读时序波形如下图所示：



5.6 ZYNQPL FMC 之 6 路 ADC 采集例程

此例程需要配合 6 路 ADC FMC 子卡使用。

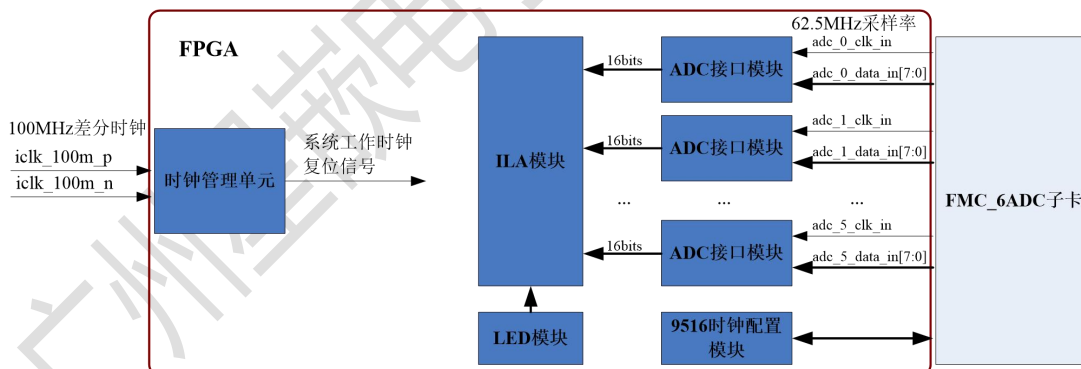
5.6.1 例程位置

ZYNQ 例程保存在资料盘中的 Demo\ZYNQ\PL\fmc6adc_prj\prj 文件夹下。

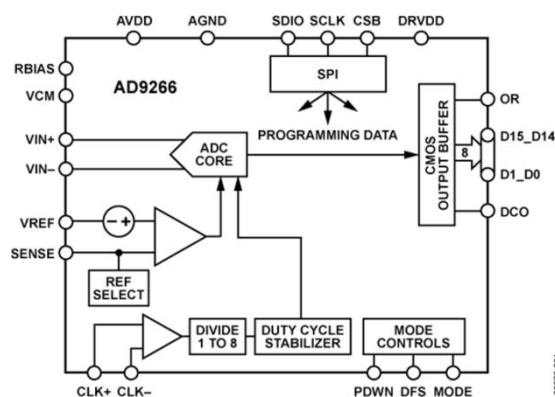
5.6.2 功能简介

6 路 ADC FMC 子卡挂载在底板 FMC/LPC 接口上，ZYNQ PL 采集 6 路 ADC 输入数据，并通过 ILA 显示采样波形数据。

例程设计框图如下：



FMC_6ADC 子卡：该子卡提供 6 路 ADC，ADC 芯片采用 ADI 公司的 AD9266。AD9266 是一款单芯片、单通道、16 位、20 MSPS/40 MSPS/65 MSPS/80 MSPS 模数转换器(ADC)，采用 1.8 V 电源供电。



iclk_100m_p/n: 为例程提供时钟源，使用板上 100MHz 差分时钟；

时钟管理单元: 为系统提供工作时钟和复位信号；

ADC 接口模块: 从 ADC 接收的数据为 DDR 模式，此接口模块将 DDR 双沿数据转换为 SDR 单沿数据，输出实际的 16bits 位宽 ADC 数据；

9516 时钟配置模块: 通过 SPI 总线配置 FMC 子卡上的时钟芯片 AD9516，为各路 ADC 提供采样时钟；

ILA 模块: 抓取信号波形，这里主要抓取 ADC 信号波形；

LED 模块: 用户自定义 LED 跳变模块。

注意: 提供的例程中，ADC 采样率为 62.5MSPS，用户可以自行修改程序，调整采样率，最高支持 80MSPS 采样率。

5.6.3 管脚约束

ZYNQ PL 工程管脚约束如下图所示：



I/O Ports							
Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	
▼ All ports (63)							
▼ BRAM_PORTA_BRAM_PORTB_32893 (1)	IN			✓	12	LVCMOS25*	
▼ Scalar ports (1)							
adc_5_clk_in	IN		AC14	✓	12	LVCMOS25*	
▼ adc_0_data_in (8)	IN			✓	13	LVCMOS25*	
adc_0_data_in[7]	IN		AE26	✓	13	LVCMOS25*	
adc_0_data_in[6]	IN		AE23	✓	13	LVCMOS25*	
adc_0_data_in[5]	IN		AE25	✓	13	LVCMOS25*	
adc_0_data_in[4]	IN		AD26	✓	13	LVCMOS25*	
adc_0_data_in[3]	IN		AC26	✓	13	LVCMOS25*	
adc_0_data_in[2]	IN		AD25	✓	13	LVCMOS25*	
adc_0_data_in[1]	IN		AB26	✓	13	LVCMOS25*	
adc_0_data_in[0]	IN		AD24	✓	13	LVCMOS25*	
▼ adc_1_data_in (8)	IN			✓	13	LVCMOS25*	
adc_1_data_in[7]	IN		AA20	✓	13	LVCMOS25*	
adc_1_data_in[6]	IN		AF20	✓	13	LVCMOS25*	
adc_1_data_in[5]	IN		AF19	✓	13	LVCMOS25*	
adc_1_data_in[4]	IN		AE21	✓	13	LVCMOS25*	
adc_1_data_in[3]	IN		AE20	✓	13	LVCMOS25*	
adc_1_data_in[2]	IN		AF22	✓	13	LVCMOS25*	
adc_1_data_in[1]	IN		AE22	✓	13	LVCMOS25*	
adc_1_data_in[0]	IN		AF23	✓	13	LVCMOS25*	
▼ adc_2_data_in (8)	IN			✓	13	LVCMOS25*	
adc_2_data_in[7]	IN		AD18	✓	13	LVCMOS25*	
adc_2_data_in[6]	IN		AB20	✓	13	LVCMOS25*	
adc_2_data_in[5]	IN		AB22	✓	13	LVCMOS25*	
adc_2_data_in[4]	IN		AD19	✓	13	LVCMOS25*	
adc_2_data_in[3]	IN		AF25	✓	13	LVCMOS25*	
adc_2_data_in[2]	IN		AB21	✓	13	LVCMOS25*	
adc_2_data_in[1]	IN		AF24	✓	13	LVCMOS25*	
adc_2_data_in[0]	IN		AC22	✓	13	LVCMOS25*	
▼ adc_3_data_in (8)	IN			✓	(Multiple)	LVCMOS25*	
adc_3_data_in[7]	IN		AF10	✓	12	LVCMOS25*	
adc_3_data_in[6]	IN		AF17	✓	12	LVCMOS25*	
adc_3_data_in[5]	IN		AE11	✓	12	LVCMOS25*	
adc_3_data_in[4]	IN		AF13	✓	12	LVCMOS25*	
adc_3_data_in[3]	IN		AA17	✓	12	LVCMOS25*	

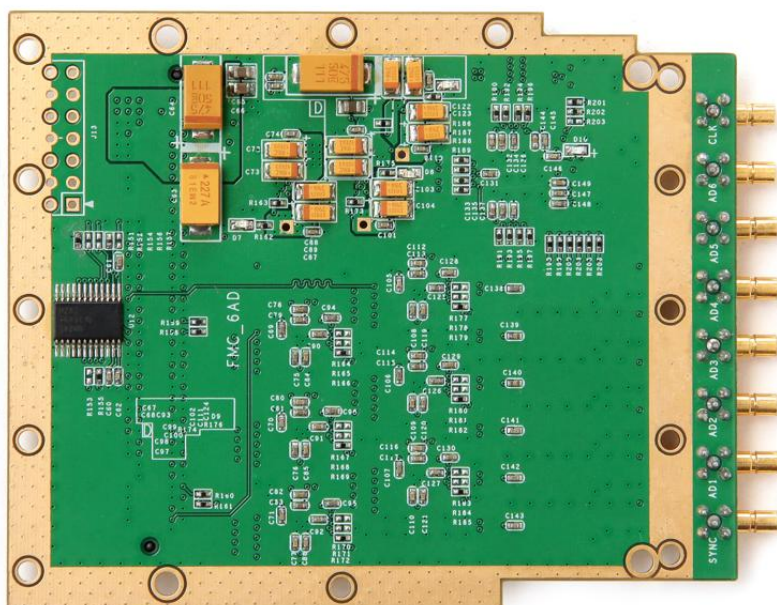
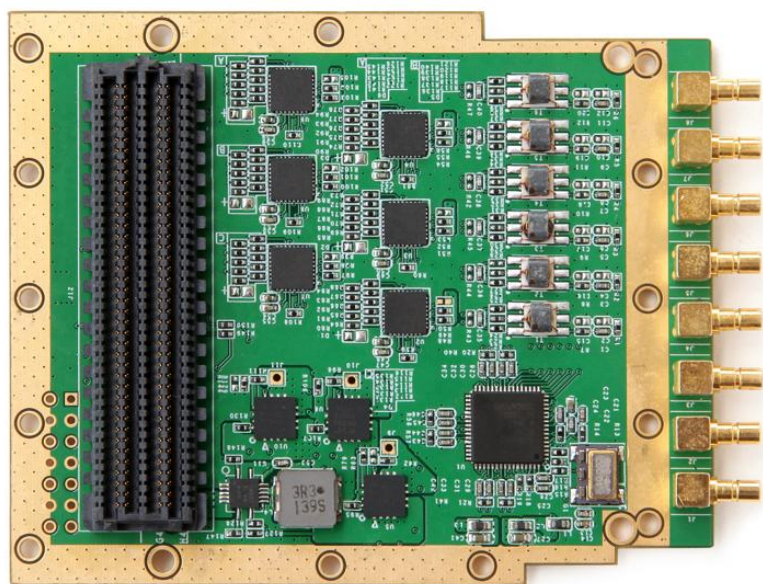
adc_3_data_in[2]	IN		AC19	▼	✓	13	LVC MOS25*	▼	
adc_3_data_in[1]	IN		AC18	▼	✓	13	LVC MOS25*	▼	
adc_3_data_in[0]	IN		AD14	▼	✓	12	LVC MOS25*	▼	
adc_4_data_in (8)	IN				✓	(Multiple)	LVC MOS25*	▼	
adc_4_data_in[7]	IN		AD15	▼	✓	12	LVC MOS25*	▼	
adc_4_data_in[6]	IN		Y17	▼	✓	12	LVC MOS25*	▼	
adc_4_data_in[5]	IN		AE13	▼	✓	12	LVC MOS25*	▼	
adc_4_data_in[4]	IN		AE17	▼	✓	12	LVC MOS25*	▼	
adc_4_data_in[3]	IN		AD16	▼	✓	12	LVC MOS25*	▼	
adc_4_data_in[2]	IN		AB19	▼	✓	13	LVC MOS25*	▼	
adc_4_data_in[1]	IN		AA19	▼	✓	13	LVC MOS25*	▼	
adc_4_data_in[0]	IN		AD13	▼	✓	12	LVC MOS25*	▼	
adc_5_data_in (8)	IN				✓	(Multiple)	LVC MOS25*	▼	
adc_5_data_in[7]	IN		AB25	▼	✓	13	LVC MOS25*	▼	
adc_5_data_in[6]	IN		AA25	▼	✓	13	LVC MOS25*	▼	
adc_5_data_in[5]	IN		Y15	▼	✓	12	LVC MOS25*	▼	
adc_5_data_in[4]	IN		Y16	▼	✓	12	LVC MOS25*	▼	
adc_5_data_in[3]	IN		AC16	▼	✓	12	LVC MOS25*	▼	
adc_5_data_in[2]	IN		AC17	▼	✓	12	LVC MOS25*	▼	
adc_5_data_in[1]	IN		AE15	▼	✓	12	LVC MOS25*	▼	
adc_5_data_in[0]	IN		AE16	▼	✓	12	LVC MOS25*	▼	
Scalar ports (14)									
ad9516_cs_n_out	OUT		AA24	▼	✓	13	LVC MOS25*	▼	
ad9516_id_in	IN		Y20	▼	✓	13	LVC MOS25*	▼	
ad9516_refsel_out	OUT		AA23	▼	✓	13	LVC MOS25*	▼	
ad9516_rstn_out	OUT		AF18	▼	✓	13	LVC MOS25*	▼	
ad9516_sclk_out	OUT		AB24	▼	✓	13	LVC MOS25*	▼	
ad9516_sdata_out	OUT		AE18	▼	✓	13	LVC MOS25*	▼	
ad9516_syn_n_out	OUT		AA22	▼	✓	13	LVC MOS25*	▼	
adc_0_clk_in	IN		AD20	▼	✓	13	LVC MOS25*	▼	
adc_1_clk_in	IN		AD23	▼	✓	13	LVC MOS25*	▼	
adc_2_clk_in	IN		AC21	▼	✓	13	LVC MOS25*	▼	
adc_3_clk_in	IN		AC13	▼	✓	12	LVC MOS25*	▼	
adc_4_clk_in	IN		AC23	▼	✓	13	LVC MOS25*	▼	
iclk_100m_p	IN	iclk_100m_n	J4	▼	✓	33	DIFF_HST...	▼	

5.6.4 例程使用

5.6.4.1 硬件准备

将一块 6 路 ADC FMC 子卡插入到底板上的 FMC/LPC 接口槽位上。

6 路 ADC FMC 子卡正、反面实物图如下图所示：

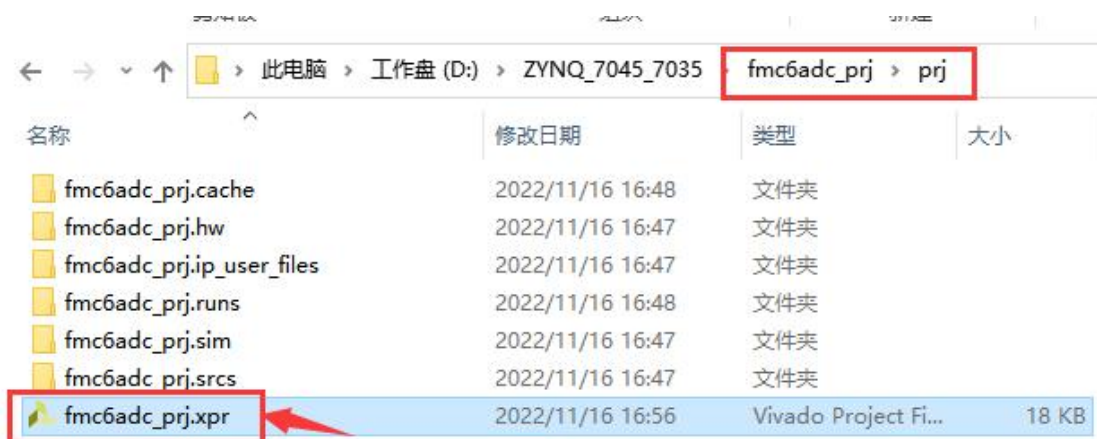


然后通过射频线缆连接信号源和 6 路 ADC FMC 子卡上的 AD1~AD6 输入接口，由信号源为 ADC 提供模拟输入信号。

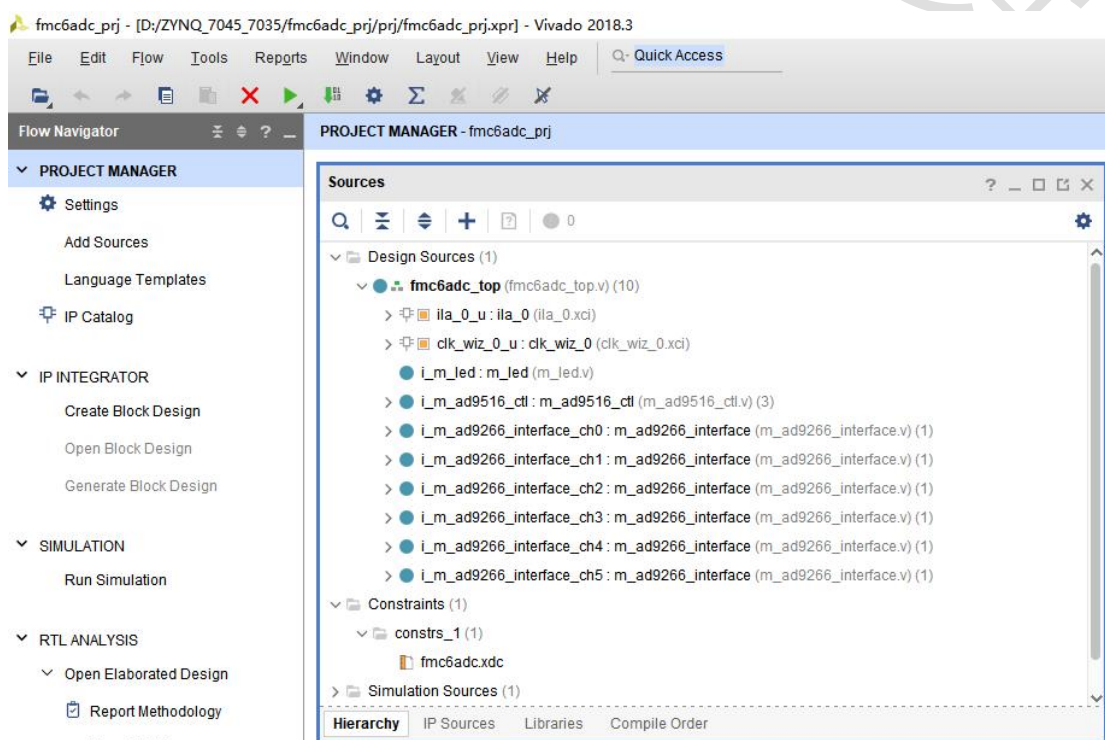
5.6.4.2 加载运行 ZYNQ 程序

5.6.4.2.1 打开 Vivado 工程

打开 Vivado 示例工程：

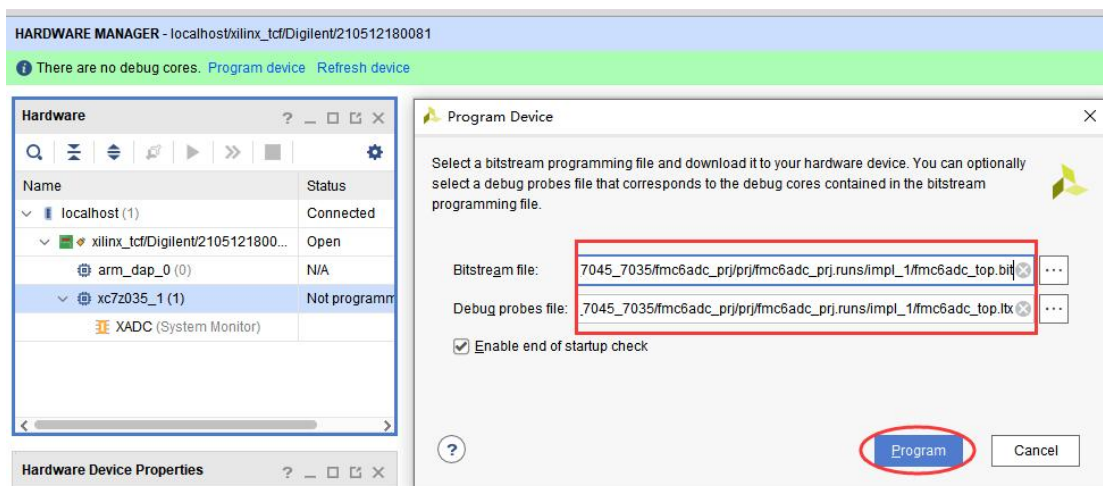


工程打开后界面如下图所示：



5.6.4.2.2 下载 ZYNQ PL 程序

下载 bit 流文件 `fm6adc_top.bit`，并且配套 `fm6adc_top.ltx` 调试文件，如下图所示：



5.6.4.3 运行结果说明

1) 信号源设置

AD0 (对应 FMC 子卡的 AD1 通道): 2MHz 正弦波输入, 2Vpp 峰峰值, 直流偏置 0V

AD1 (对应 FMC 子卡的 AD2 通道): 1MHz 正弦波输入, 2Vpp 峰峰值, 直流偏置 0V

2) ILA 波形抓取

ILA 里面抓取了 6 路 ADC 采集得到的数字信号, 抓取示例结果如下图所示:



注: ILA 中为了方便查看时域波形, 将信号显示方式设置为模拟显示。

5.6.4.4 退出实验

Vivado 调试界面 Hardware Manager 窗口, 右键单击 localhost(1), 在弹出的菜单中点击 Close Server, 断开 ZYNQ JTAG 仿真器与板卡的连接。

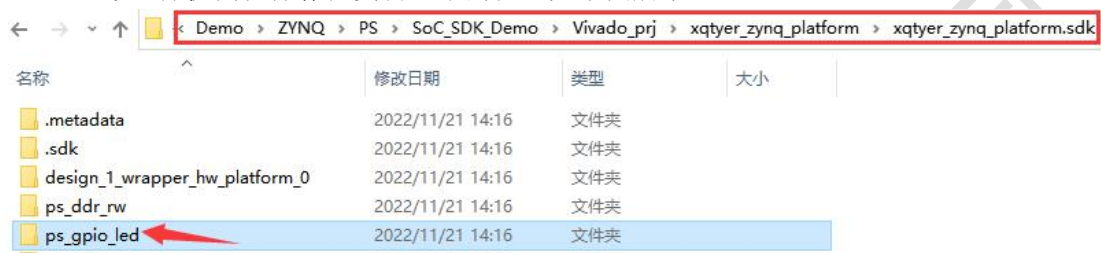
最后, 关闭板卡电源, 实验结束。

6 ZYNQ PS 单独例程

6.1 ZYNQ PS GPIO LED 驱动实验

6.1.1 例程位置

ZYNQ PS 裸机例程保存在资料盘中的位置如下图所示：



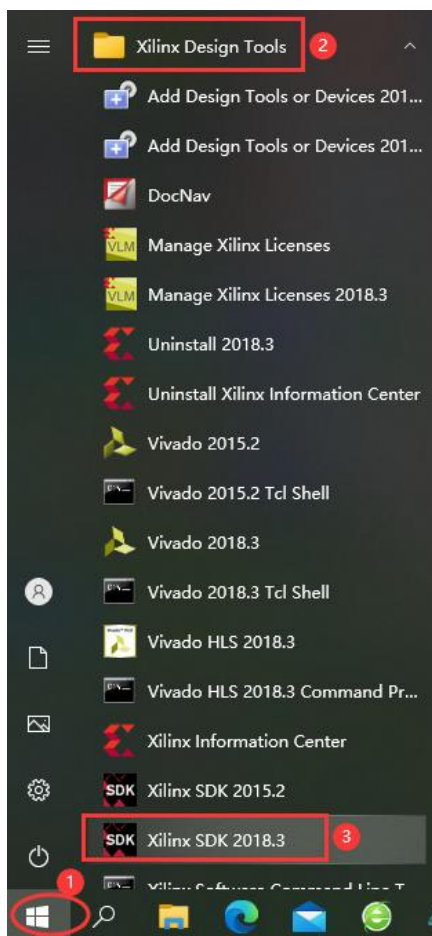
6.1.2 功能简介

ZYNQ PS 端通过 EMIO 接口实现对底板 LED 灯的驱动控制。

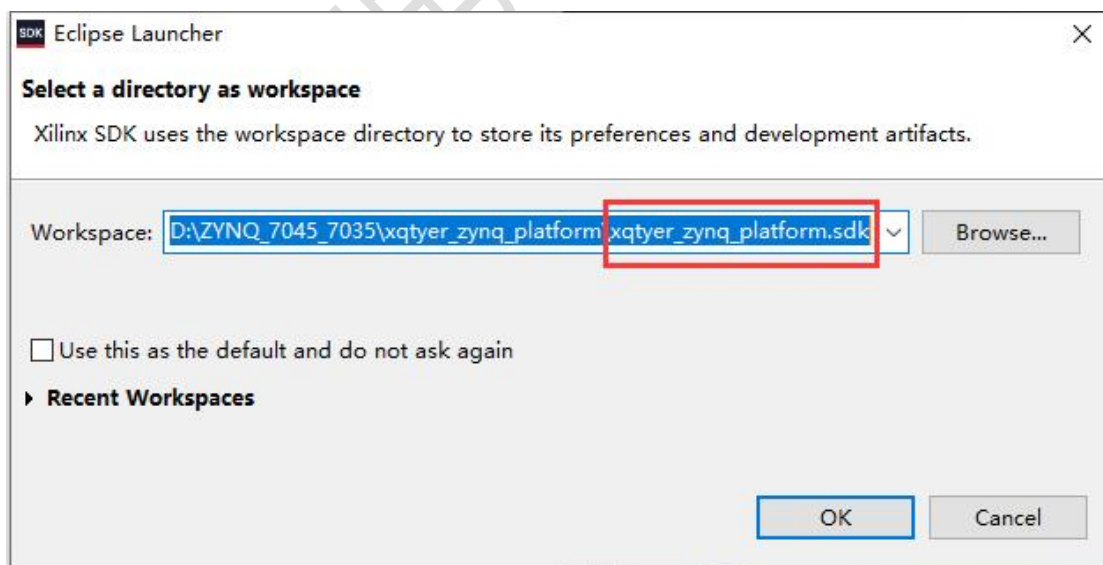
6.1.3 例程使用

6.1.3.1 打开 Xilinx SDK 软件例程

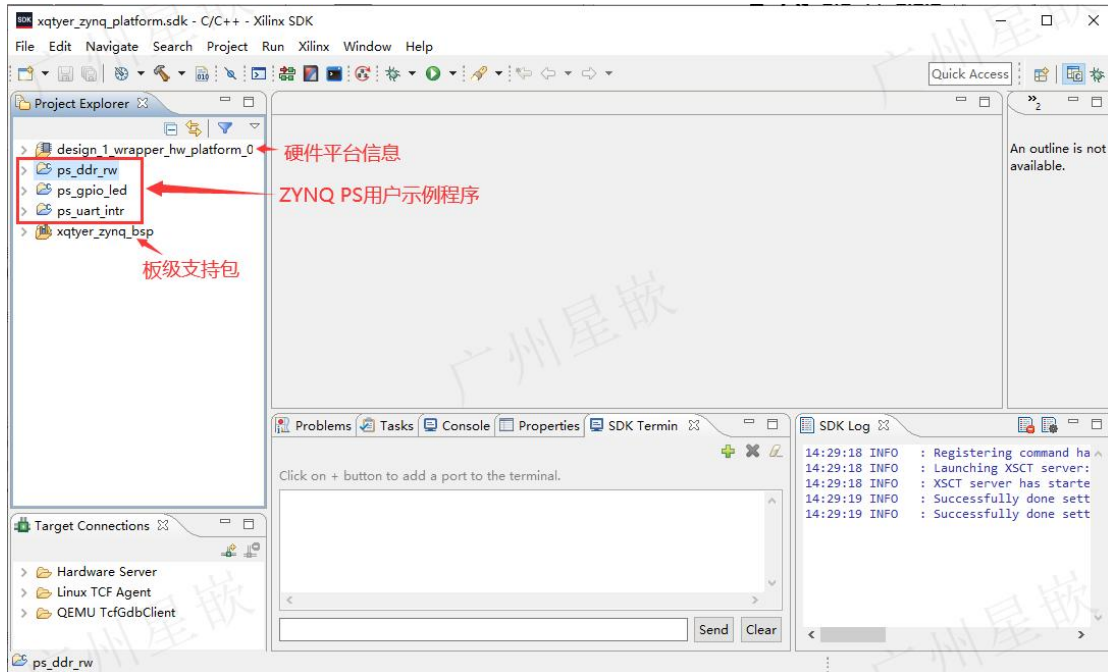
通过开始菜单，打开 Xilinx SDK 软件，如下图所示：



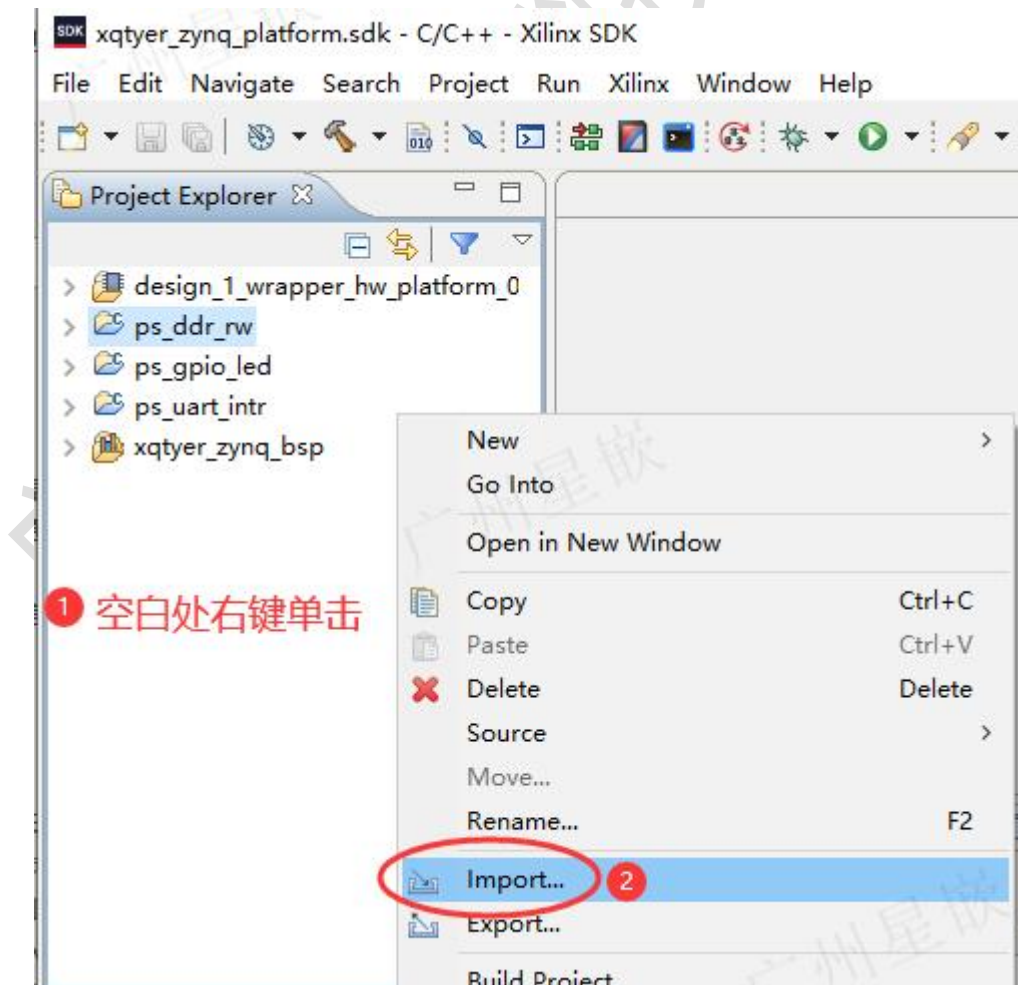
设置 Xilinx SDK 软件工作空间：



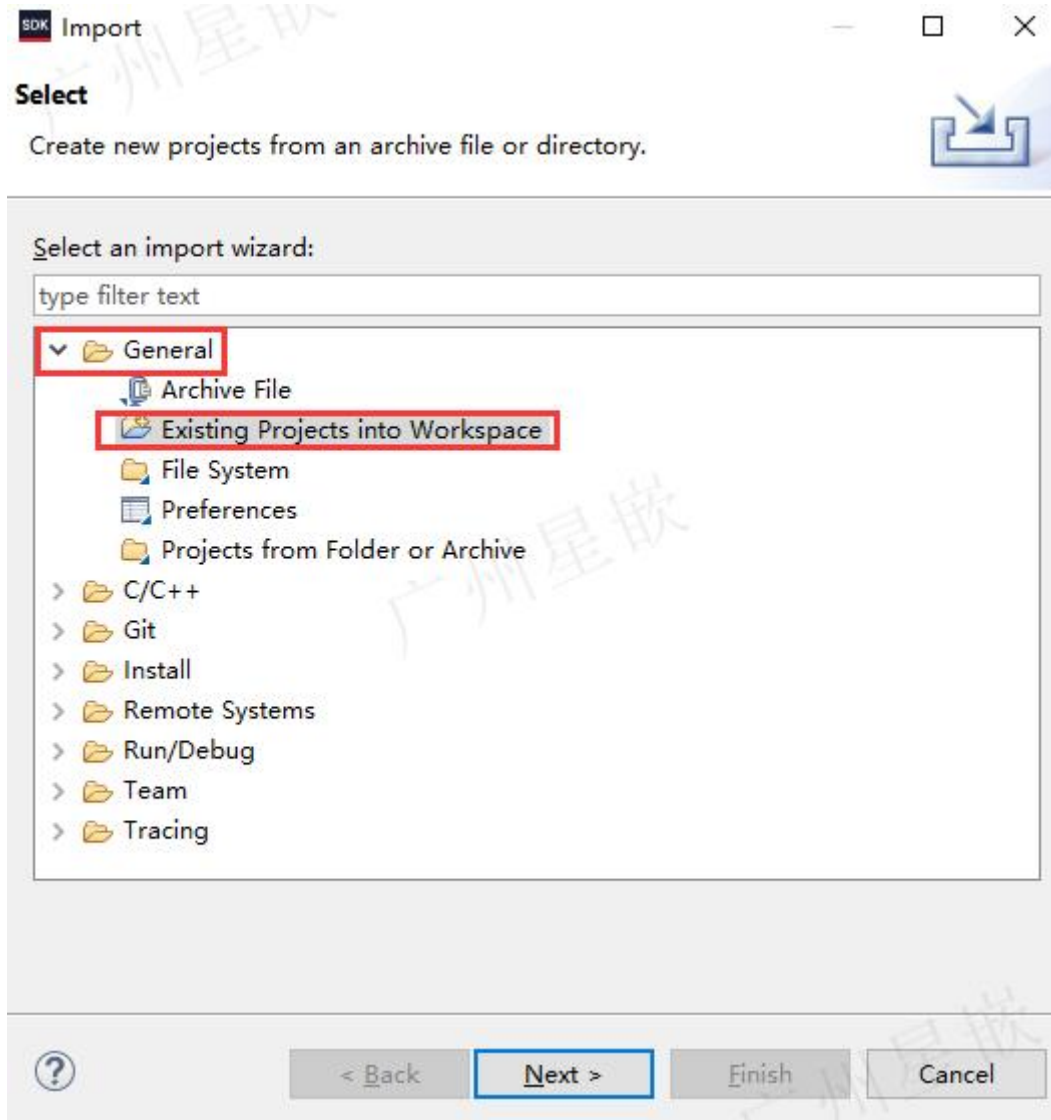
Xilinx SDK 软件打开后界面如下图所示，在 Project Explorer 窗口会显示已导入到工作区内的工程：



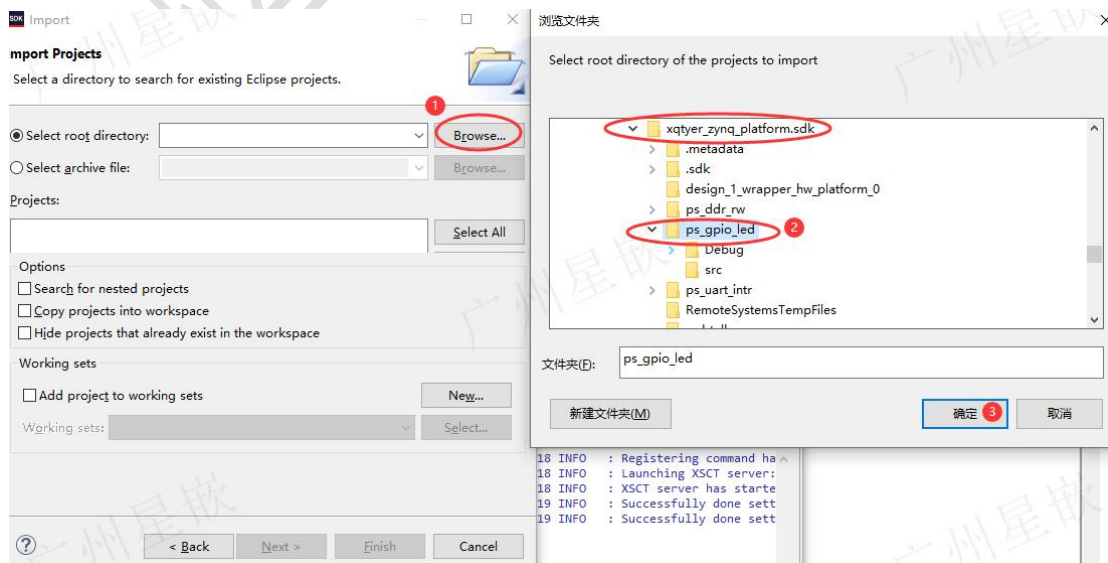
请确认 Project Explorer 窗口已经存在 ps_gpio_led 例程，如果已经存在，则忽略后面的操作，直接进入下一节“程序运行测试”；如果示例工程不存在，则在 Project Explorer 窗口空白处右键，然后点击 Import...导入：



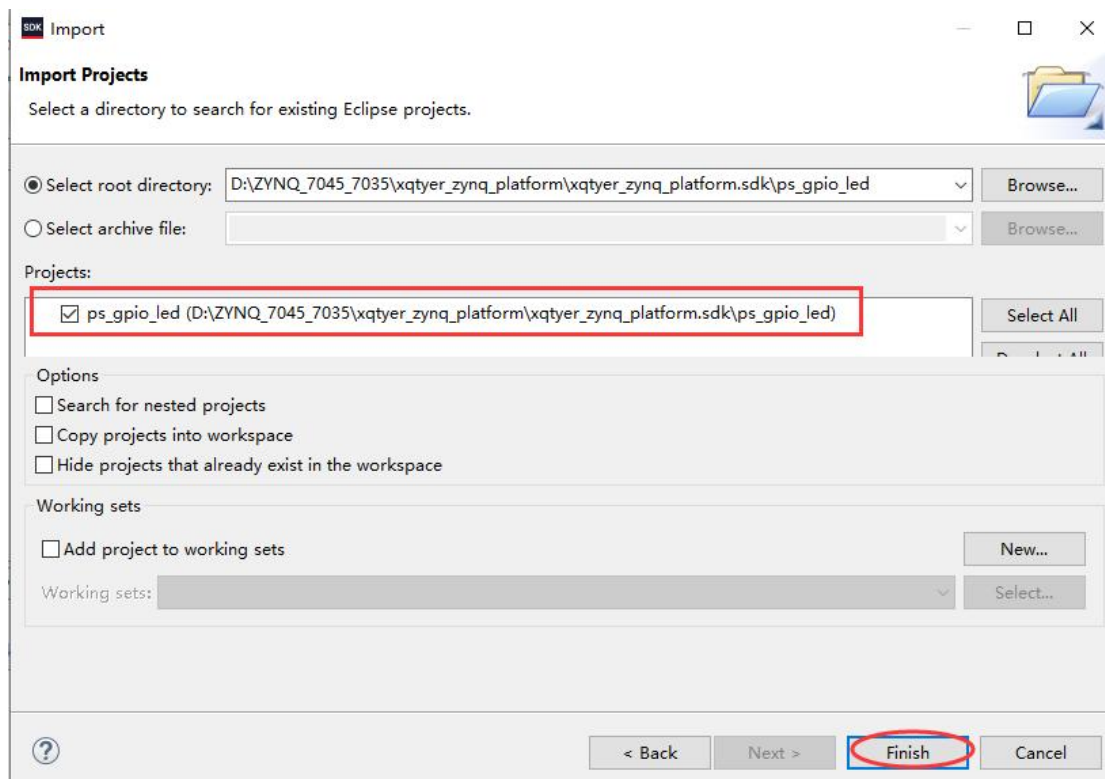
导入时，选中导入已有工程：



然后浏览找到 Xilinx SDK 示例工程，如下图所示的 ps_gpio_led 示例工程：



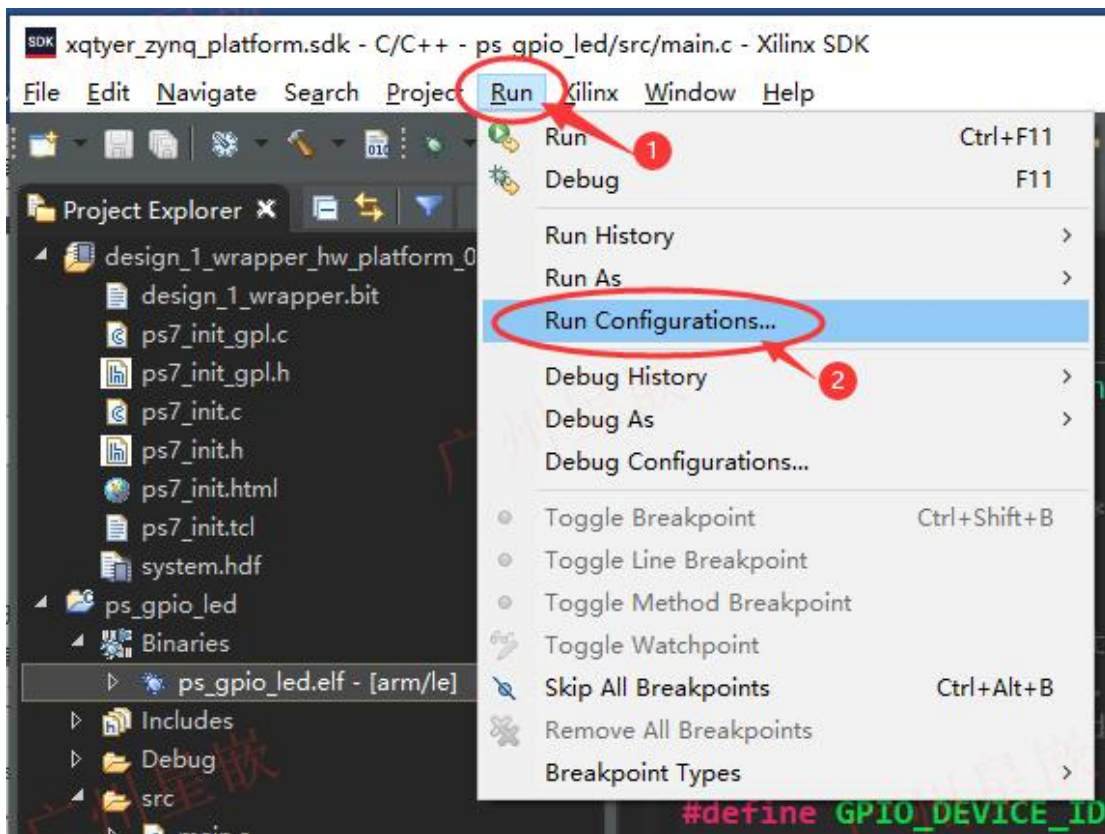
在 Projects 中列出了在所浏览的目录中发现的 Xilinx SDK 工程，点击勾选上前面的方框，然后点击 Finish 完成 Xilinx SDK 工程导入（**注意：如果 Xilinx SDK 工作区内已经存在此工程，则 Projects 中列出的工程项目为灰色条目，用户无法继续导入操作**）：



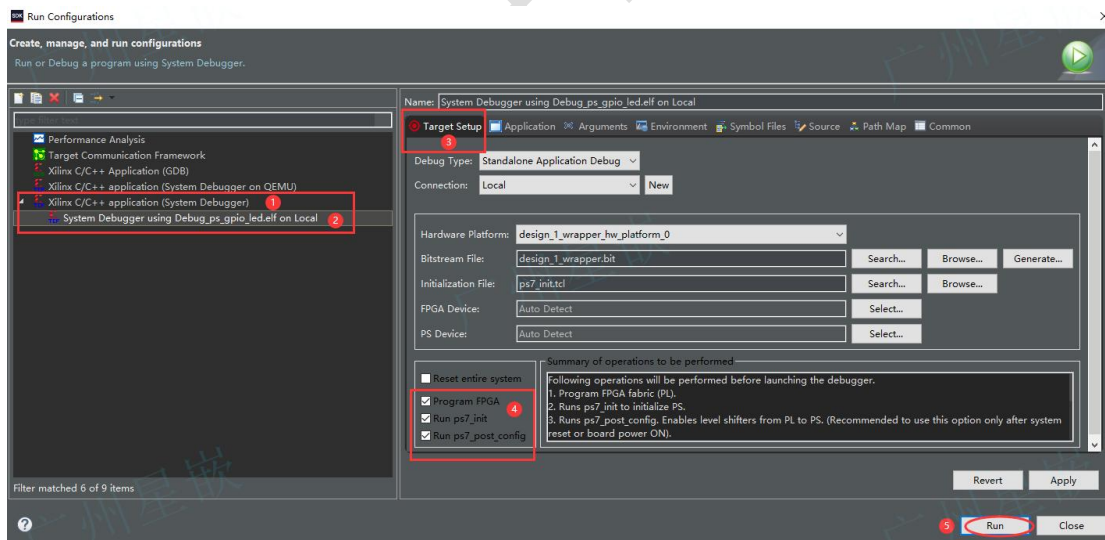
6.1.3.2 程序运行测试

板卡连上 JTAG 仿真器，拨码开关 SW2 将 ZYNQ 启动模式设置为 JTAG 启动，然后给板卡上电。

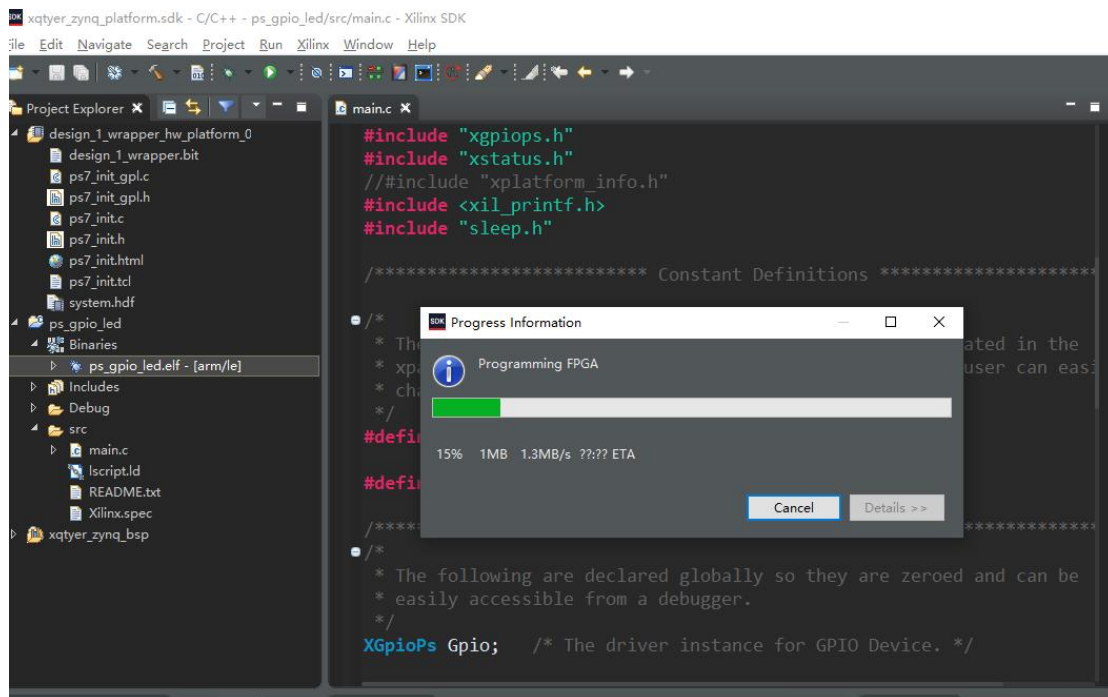
在 Xilinx SDK 软件界面，点击 Run->Run Configurations...:



双击 Xilinx C/C++ application(System Debugger)，选中 Xilinx C/C++ application(System Debugger)下面的 System Debugger using Debug_ps_gpio_led.elf on Local，然后点击在右侧界面的 Target Setup 一栏，并勾选上下图所示的④号红框内的三个选项，最后点击 Run：



点击 Run 运行后，首先将 bit 流文件下载到 ZYNQ PL，下载过程如下图所示：



ZYNQ PL 端 bit 流文件下载完成后，则自动开始运行用户应用程序，本实验就是 GPIO LED 灯的驱动程序。此时，可以观察底板上的 LED3 是否在闪烁。如果程序运行正常，则 LED3 会亮 0.5 秒，然后灭 0.5 秒，如此反复，形成 1 秒钟闪烁 1 次的效果。

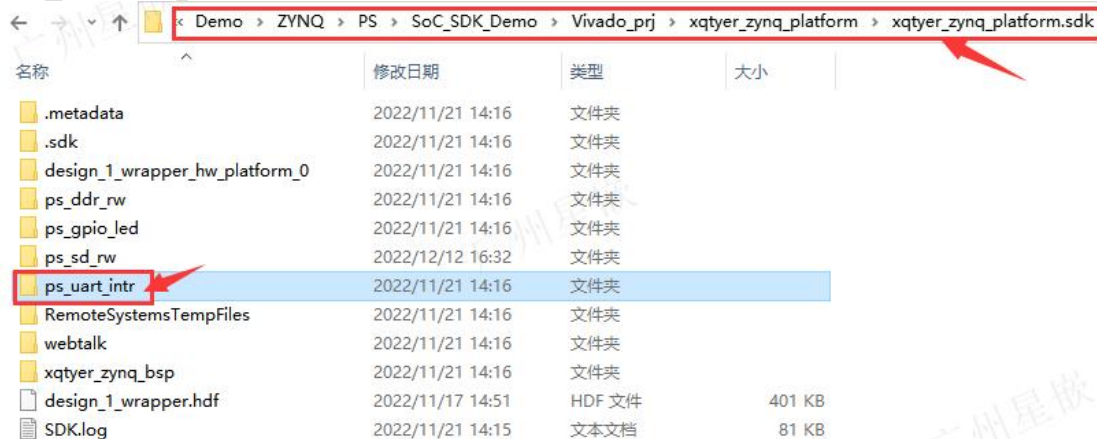
6.1.3.3 结束实验

直接关闭板卡电源，结束本次实验。

6.2 ZYNQ PS 串口中断实验

6.2.1 例程位置

ZYNQ PS 裸机例程保存在资料盘中的位置如下图所示：



6.2.2 功能简介

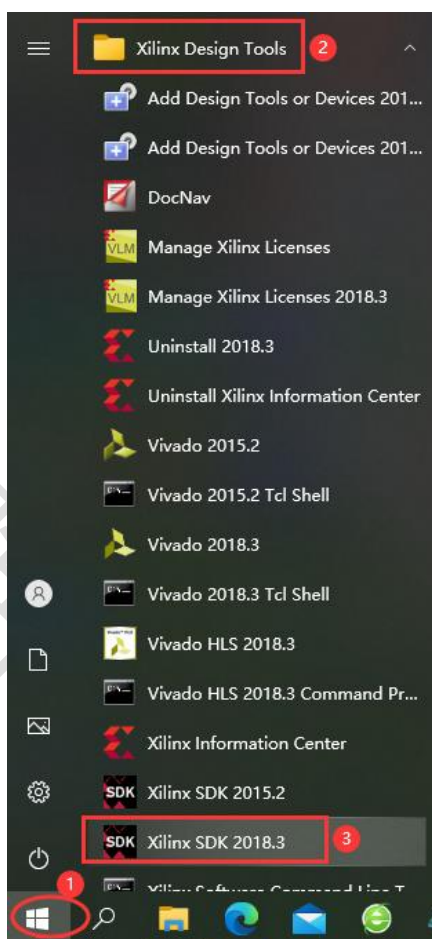
以中断方式，实现 ZYNQ PS 端 UART 串口数据收发。

ZYNQ PS 包含 UART0 和 UART1 两个串口模块，我们硬件设计只使用了 UART1，对应到 MIO48 和 MIO49 管脚。

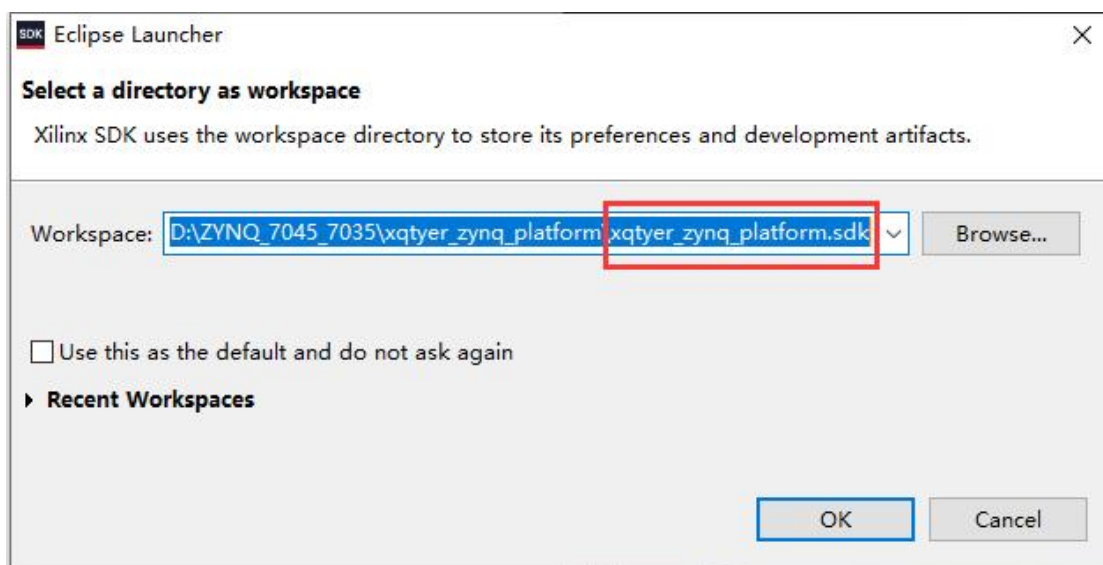
6.2.3 例程使用

6.2.3.1 打开 Xilinx SDK 软件例程

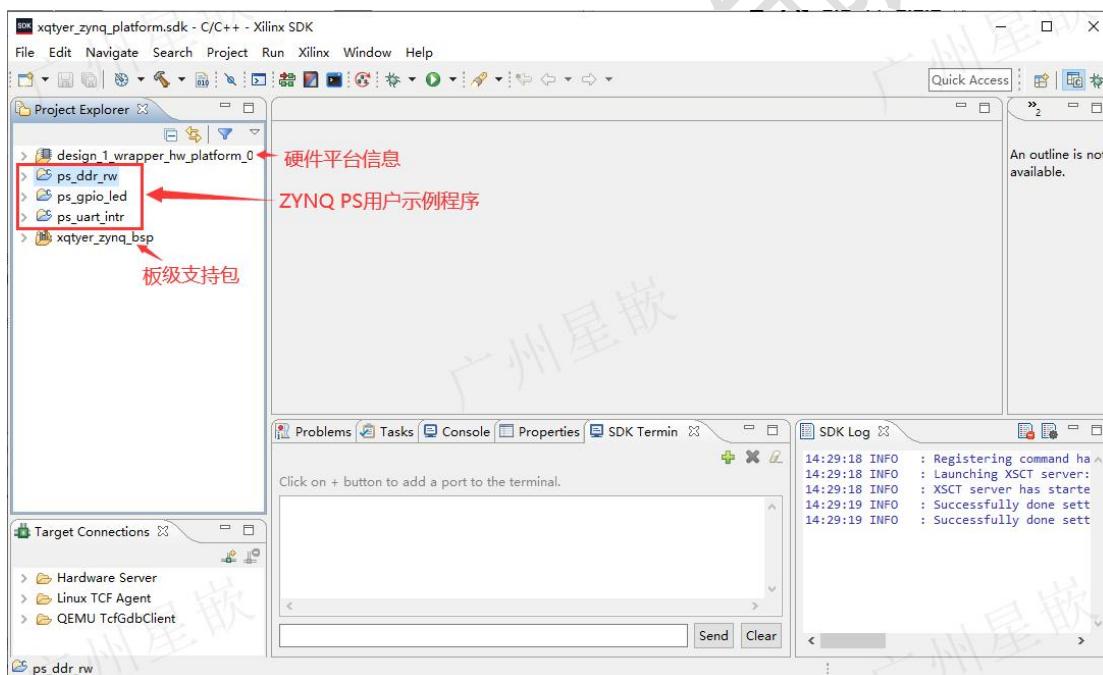
通过开始菜单，打开 Xilinx SDK 软件，如下图所示：



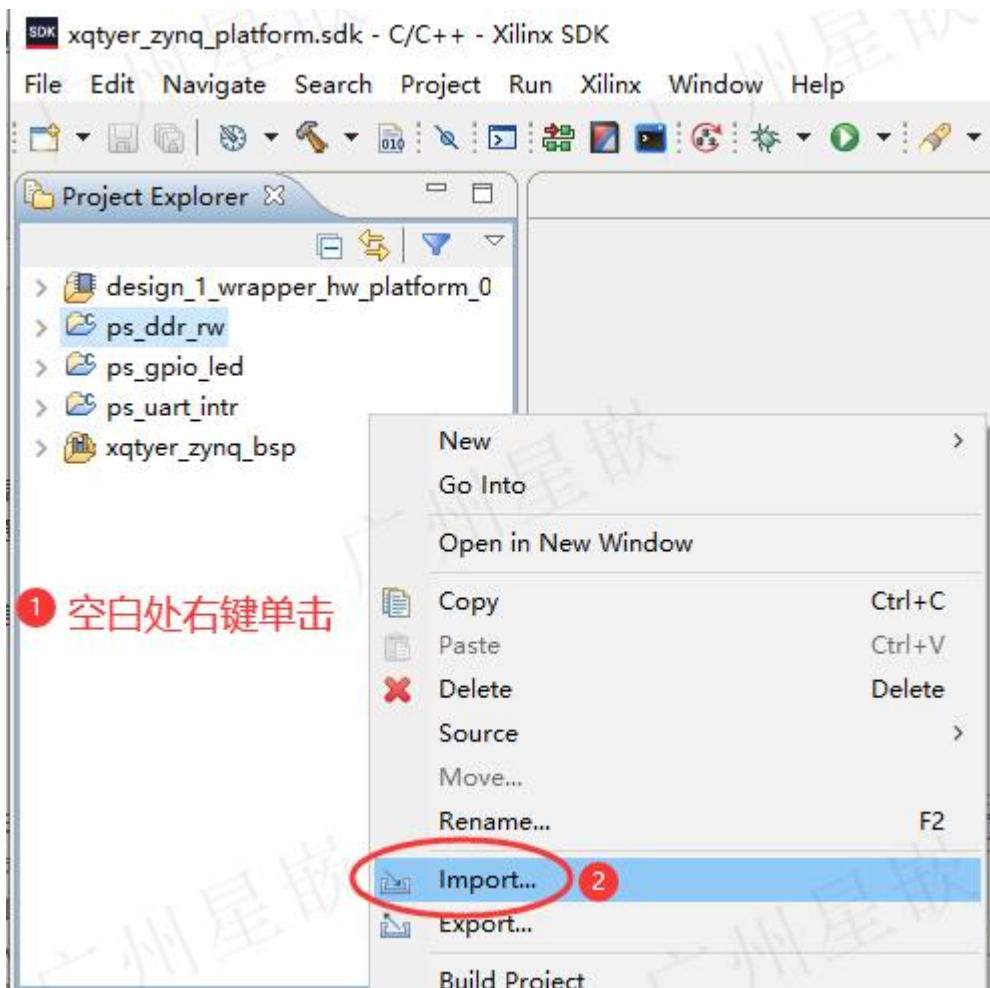
设置 Xilinx SDK 软件工作空间：



Xilinx SDK 软件打开后界面如下图所示，在 Project Explorer 窗口会显示已导入到工作区内的工程：

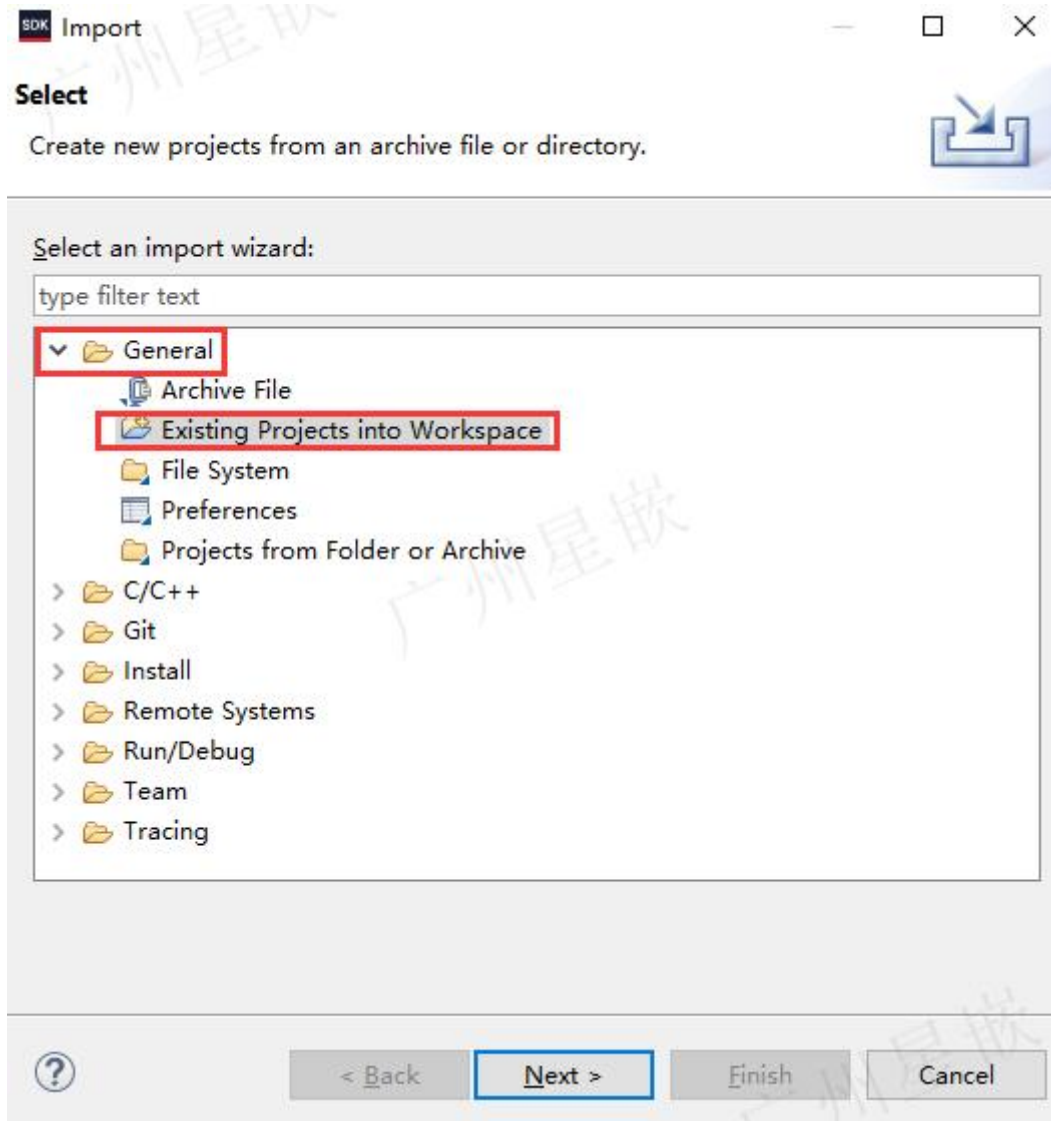


请确认 Project Explorer 窗口已经存在 ps_uart_intr 例程，如果已经存在，则忽略后面的操作，直接进入下一节“程序运行测试”；如果示例工程不存在，则在 Project Explorer 窗口空白处右键，然后点击 Import... 导入：

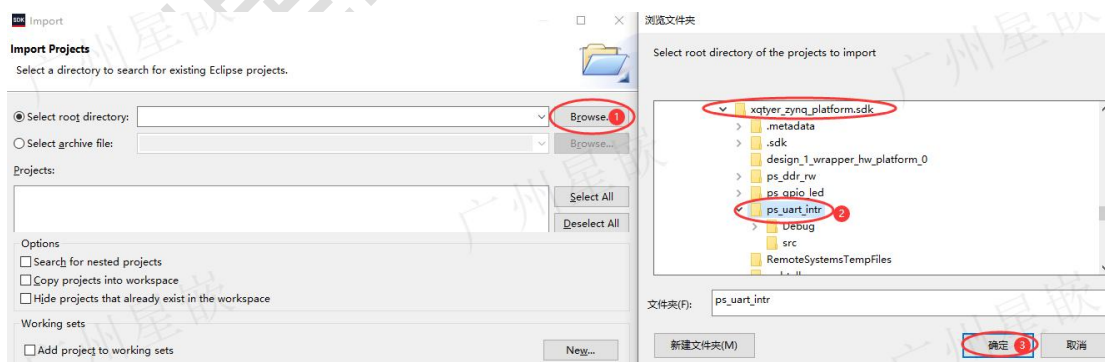


1 空白处右键单击

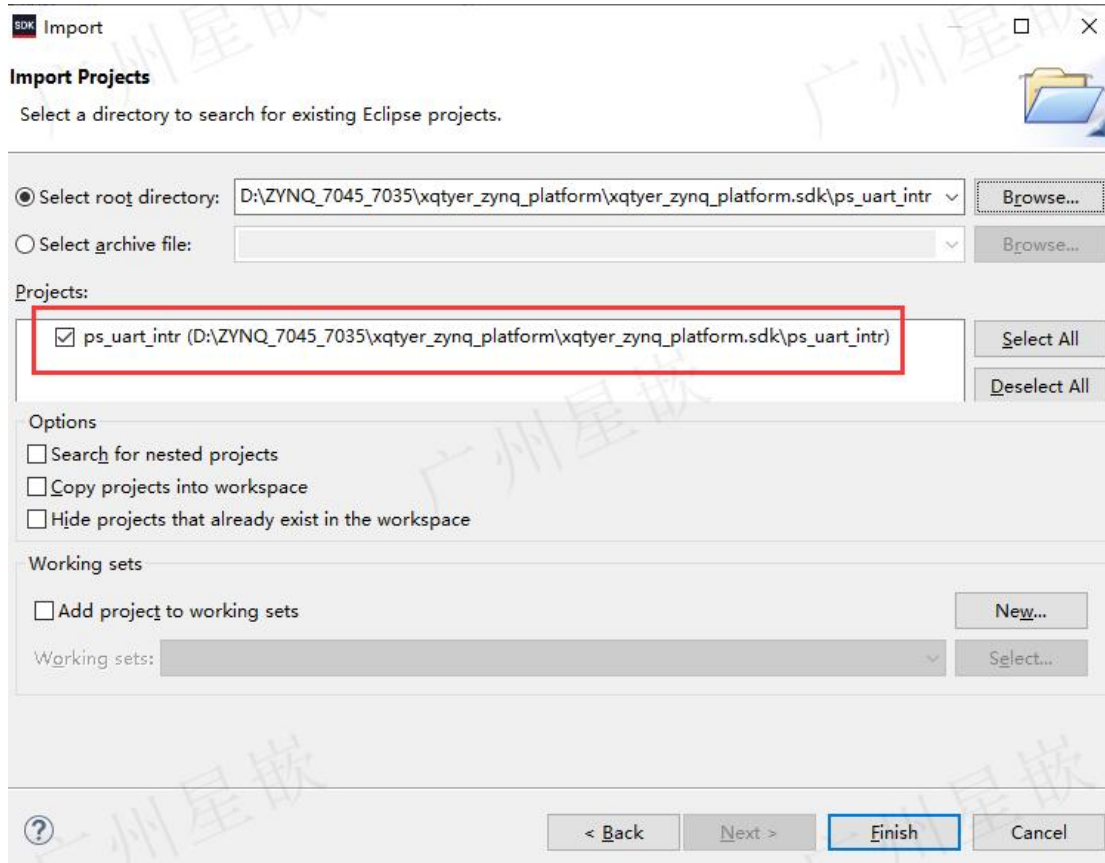
导入时，选中导入已有工程：



然后浏览找到 Xilinx SDK 示例工程，如下图所示的 ps_uart_intr 示例工程：

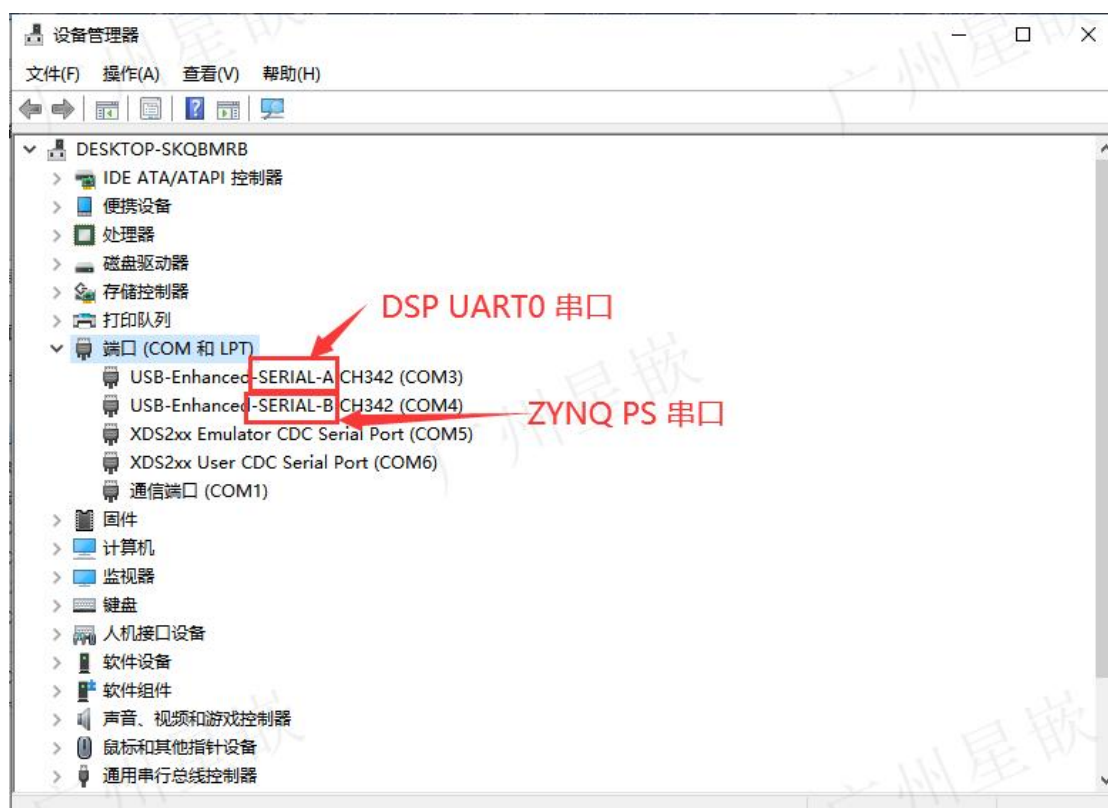


在 Projects 中列出了在所浏览的目录中发现的 Xilinx SDK 工程，点击勾选上前面的方框，然后点击 Finish 完成 Xilinx SDK 工程导入（**注意：如果 Xilinx SDK 工作区内已经存在此工程，则 Projects 中列出的工程项目为灰色条目，用户无法继续导入操作**）：

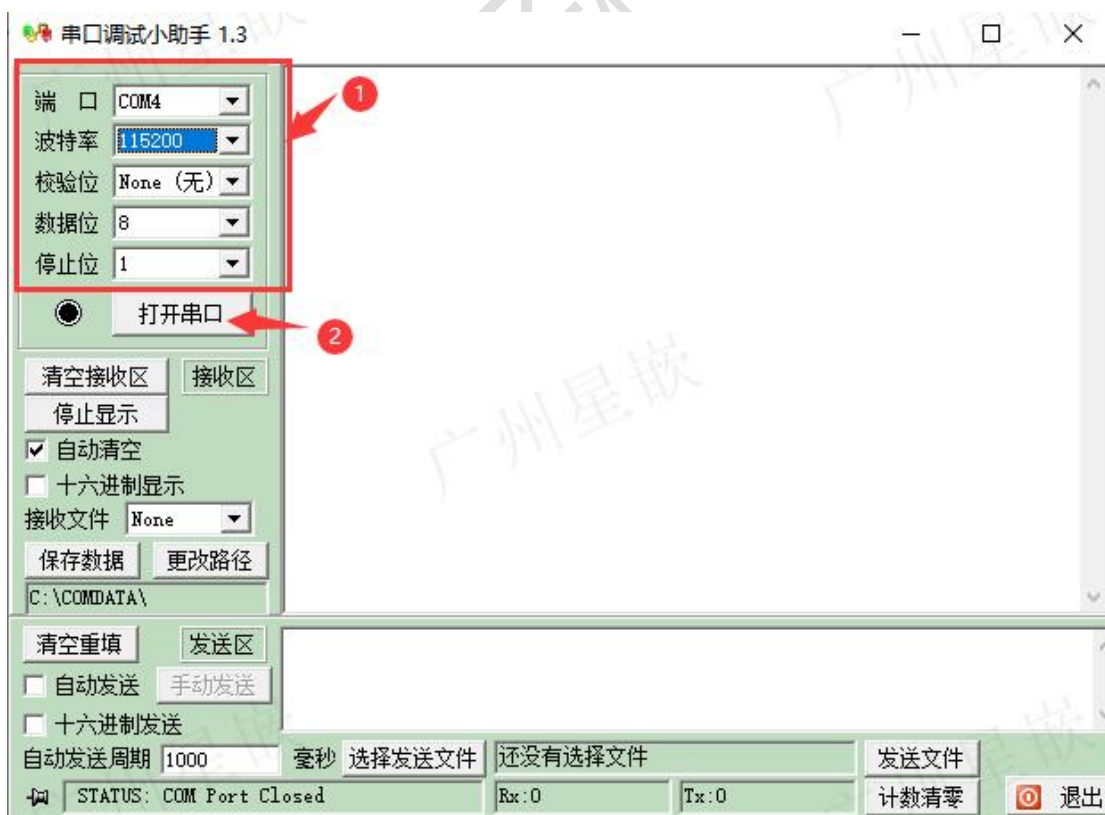


6.2.3.2 程序运行测试

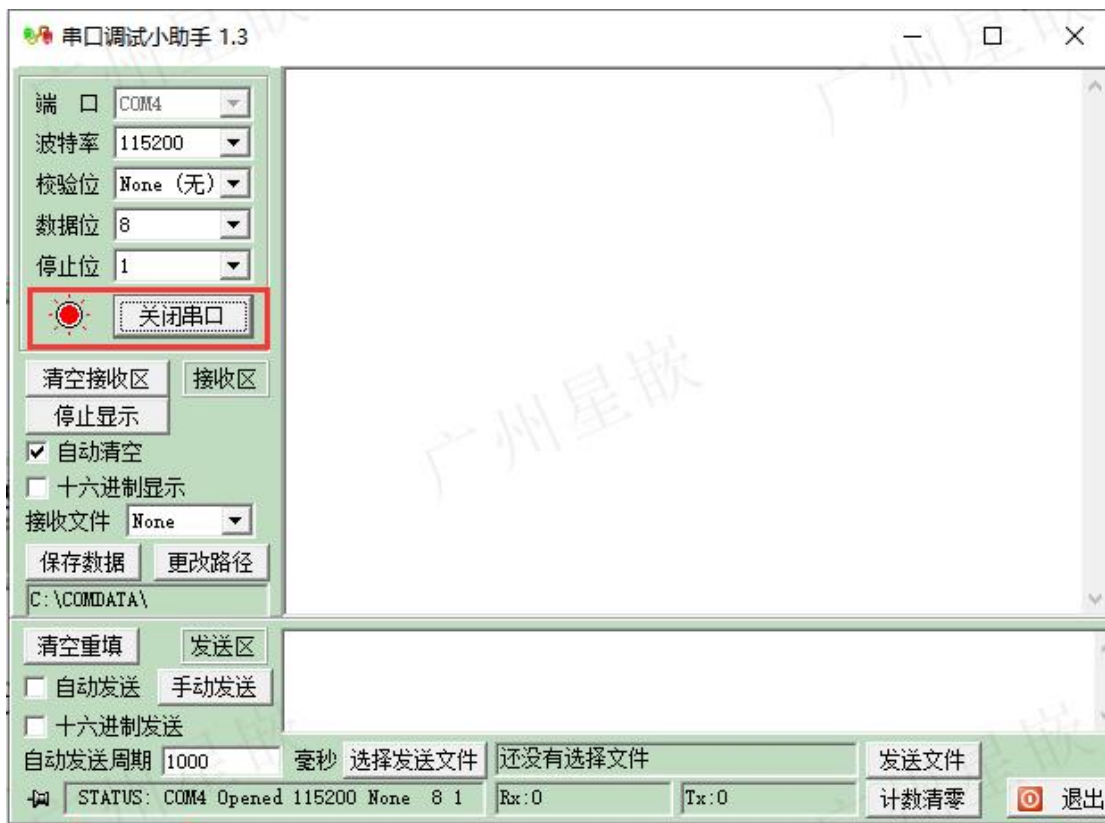
- (1) 板卡连上 JTAG 仿真器，拨码开关 SW2 将 ZYNQ 启动模式设置为 JTAG 启动，然后给板卡上电。
- (2) 电脑端打开设备管理器查看串口号。带 SERIAL-A 字样的为 DSP 端串口，带 SERIAL-B 字样的为 ZYNQ PS 串口，记住后面的 COM 编号，这里 ZYNQ PS 端的串口编号为 COM4:



- (3) 打开串口调试助手，按照前面查看到的 COM 号设置串口端口号，波特率设置为 115200，无校验位，数据位为 8 位，停止位为 1 位，设置好后点击“打开串口”：

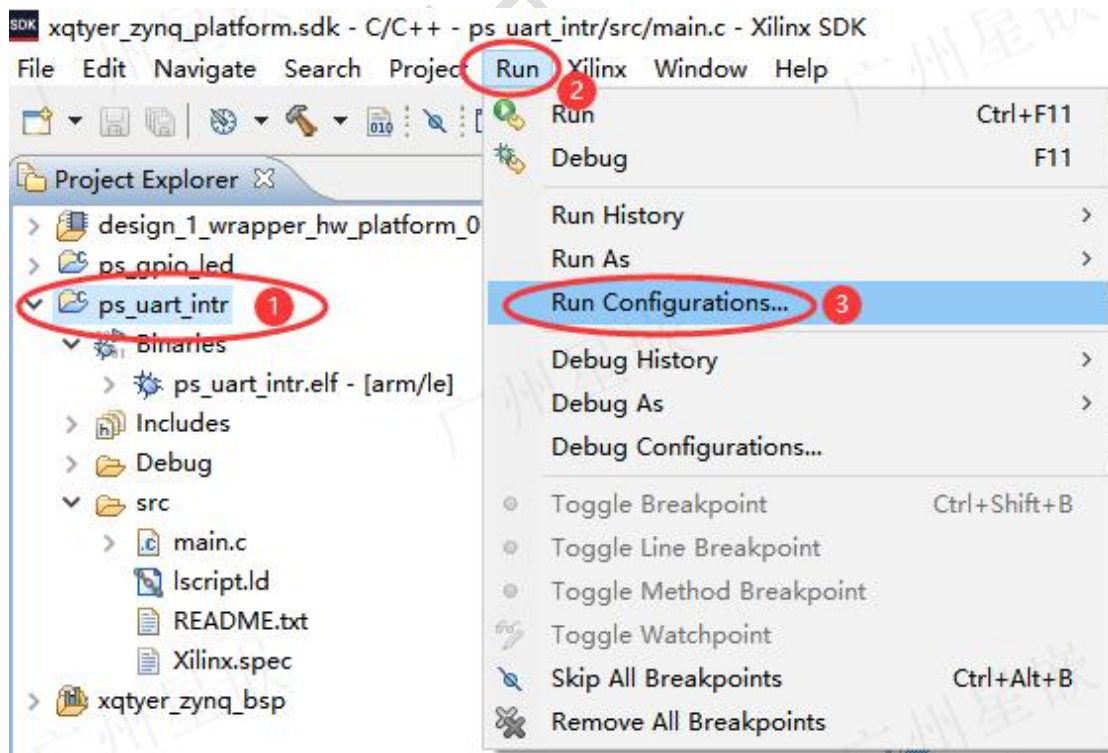


串口调试助手将串口打开后的界面如下图所示：



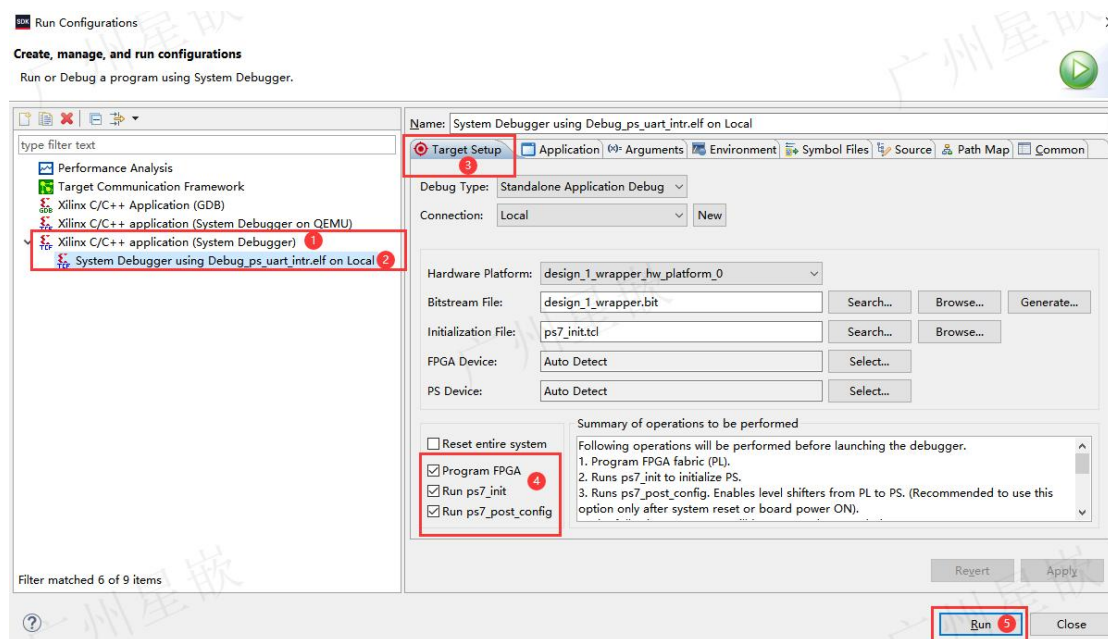
(4) 下载和运行应用程序

在 Xilinx SDK 软件界面，点击选中应用工程，然后点击 Run->Run Configurations...:

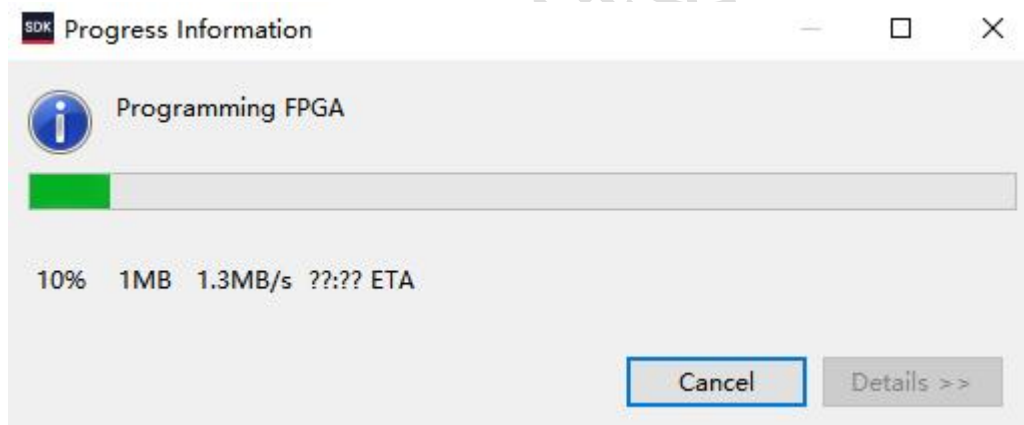


双击 Xilinx C/C++ application(System Debugger)，选中 Xilinx C/C++ application(System Debugger)下面的 System Debugger using Debug_ps_uart_intr.elf on Local，然后点击在右侧界

面的 Target Setup 一栏，并勾选上下图所示的④号红框内的三个选项，最后点击 Run：

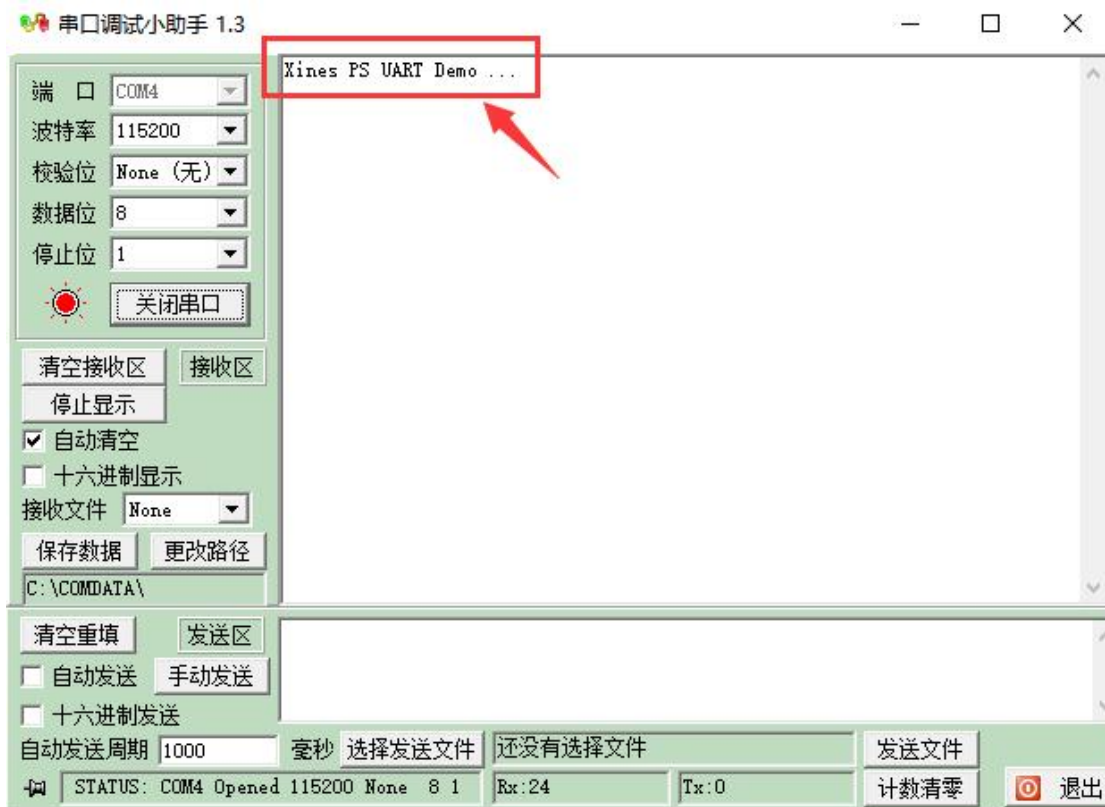


点击 Run 运行后，首先将 bit 流文件下载到 ZYNQ PL，下载过程如下图所示：

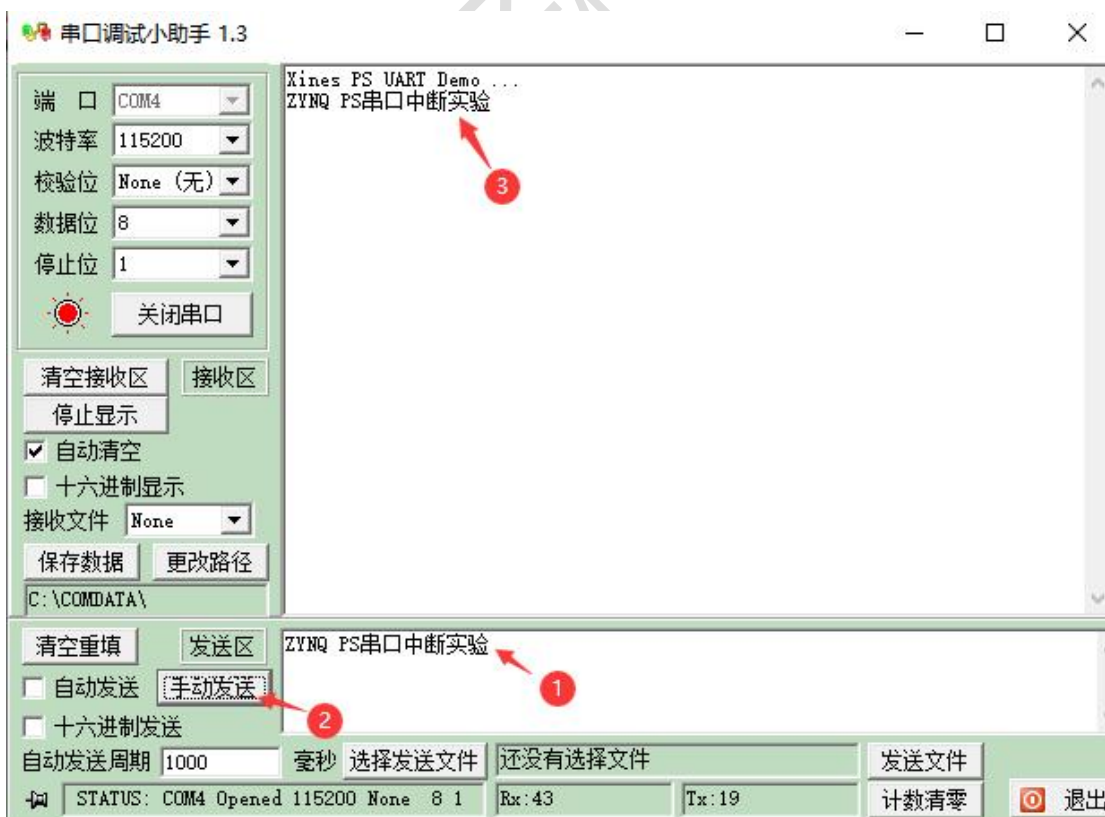


ZYNQ PL 端 bit 流文件下载完成后，则自动开始运行用户应用程序。

- (5) 用户应用程序下载运行后，可查看到串口调试助手打印程序运行起来的信息：



- (6) 在串口调试助手发送区输入信息，然后点击“手动发送”，ZYNQ PS 端串口收到信息后原样再返回给串口调试助手，并在串口调试助手的接收区显示，示例如下图所示：



6.2.3.3 结束实验

关闭串口调试助手。

最后，关闭板卡电源，实验结束。

6.3 ZYNQ PS DDR 内存读写实验

6.3.1 例程位置

ZYNQ PS 裸机例程保存在资料盘中的位置如下图所示：



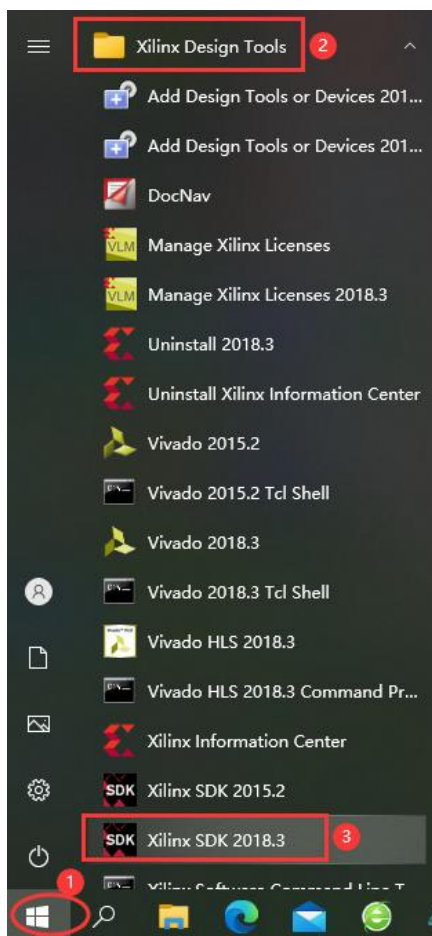
6.3.2 功能简介

对 ZYNQ PS 端挂载的 DDR3 内存进行读写测试。该实验同时也对 ZYNQ PS OCM 片上共享内存进行了读写测试。

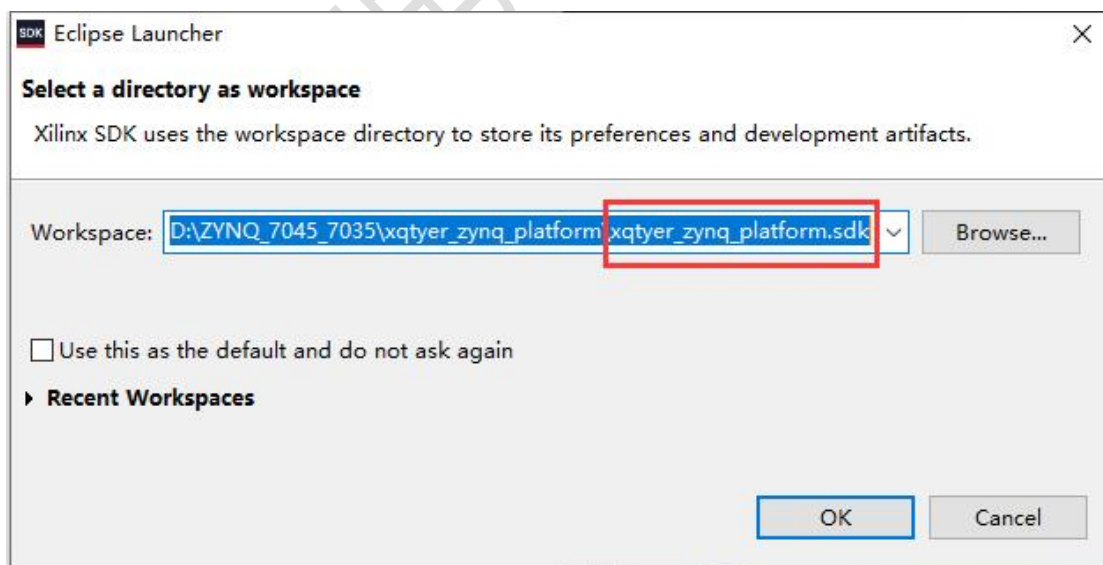
6.3.3 例程使用

6.3.3.1 打开 Xilinx SDK 软件例程

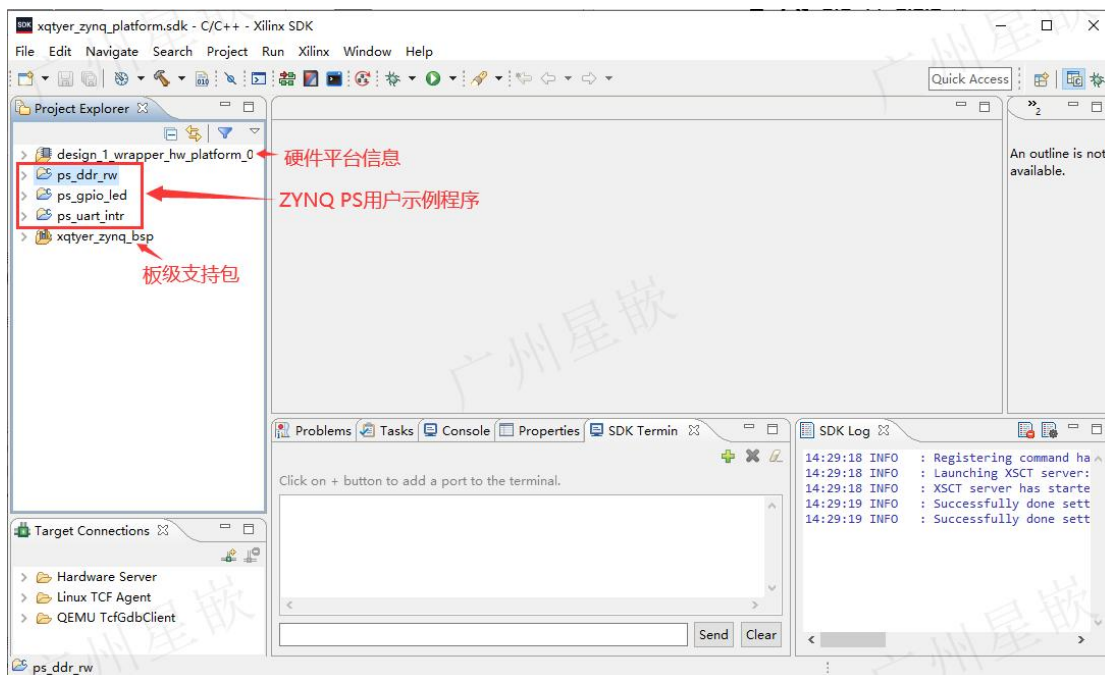
通过开始菜单，打开 Xilinx SDK 软件，如下图所示：



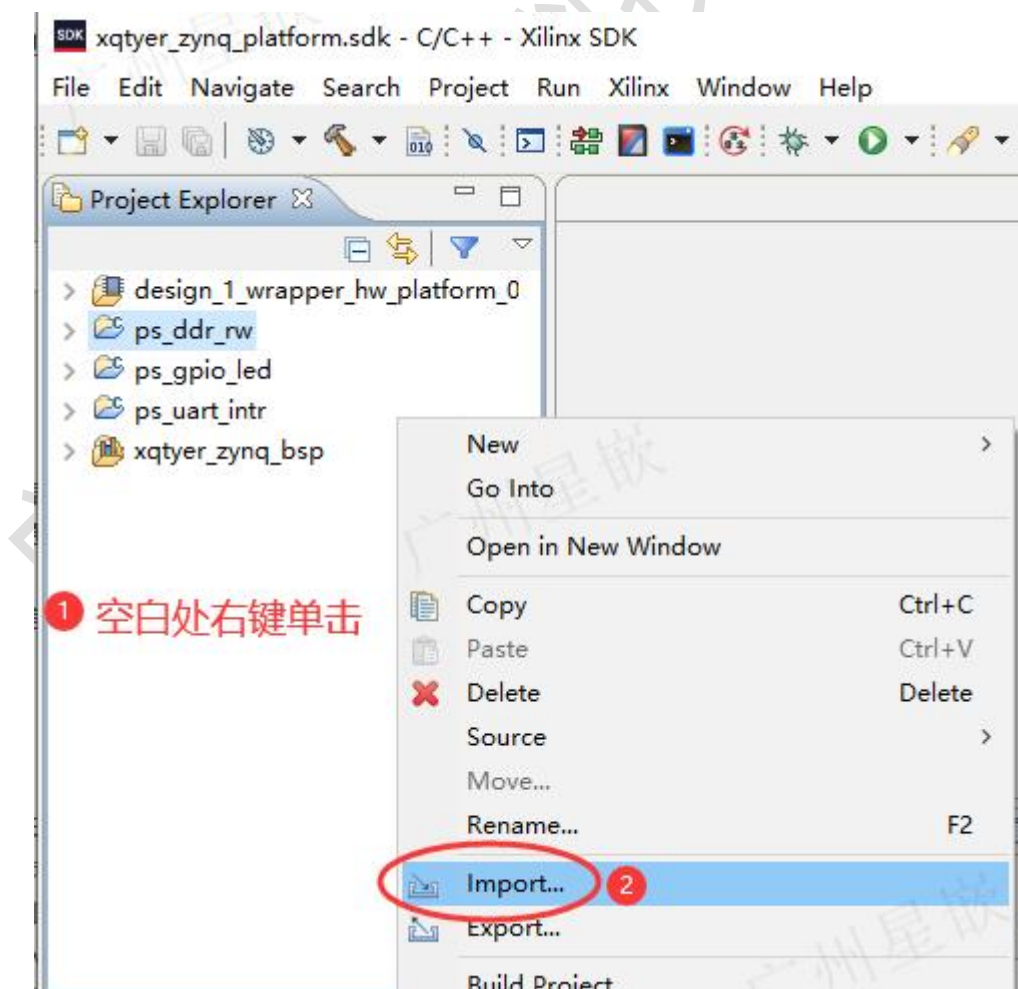
设置 Xilinx SDK 软件工作空间：



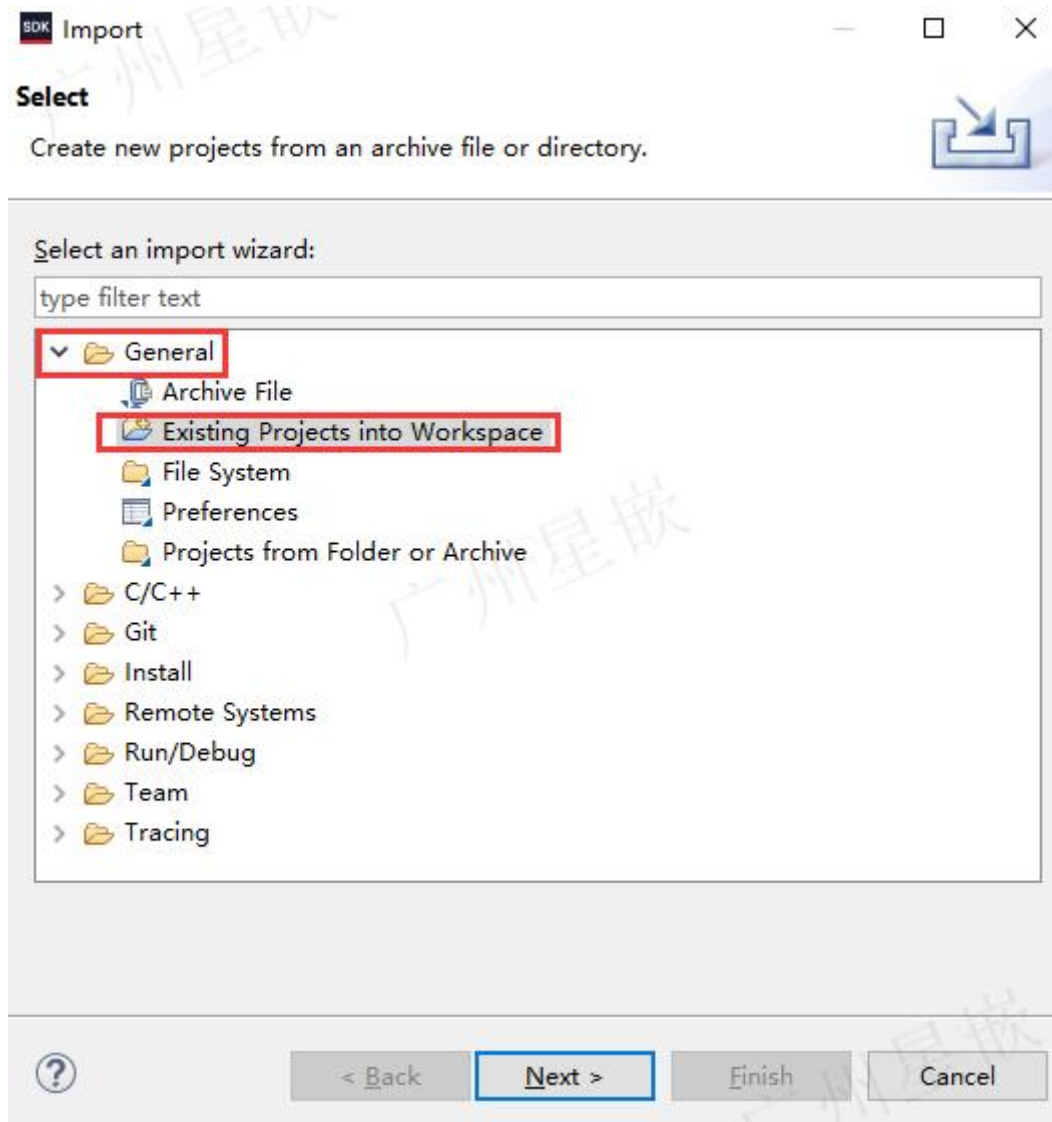
Xilinx SDK 软件打开后界面如下图所示，在 Project Explorer 窗口会显示已导入到工作区内的工程：



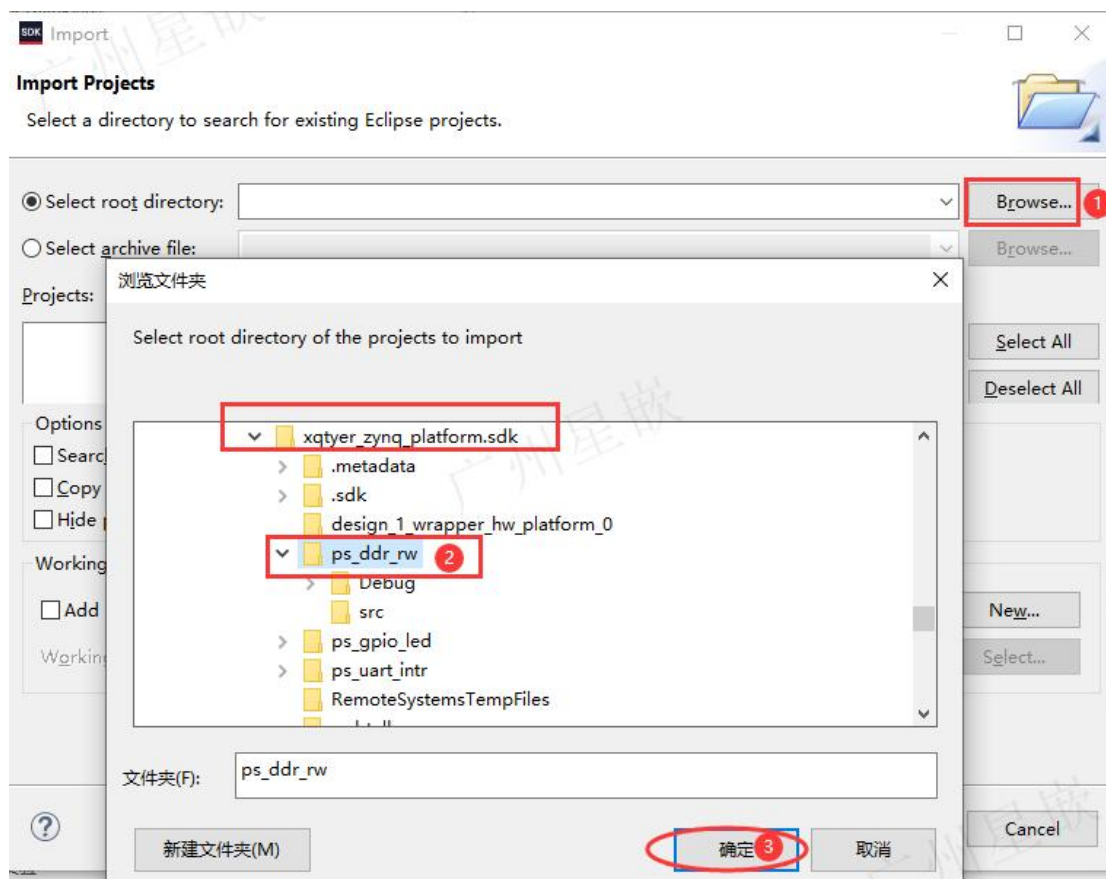
请确认 Project Explorer 窗口已经存在 ps_ddr_rw 例程，如果已经存在，则忽略后面的操作，直接进入下一节“程序运行测试”；如果示例工程不存在，则在 Project Explorer 窗口空白处右键，然后点击 Import...导入：



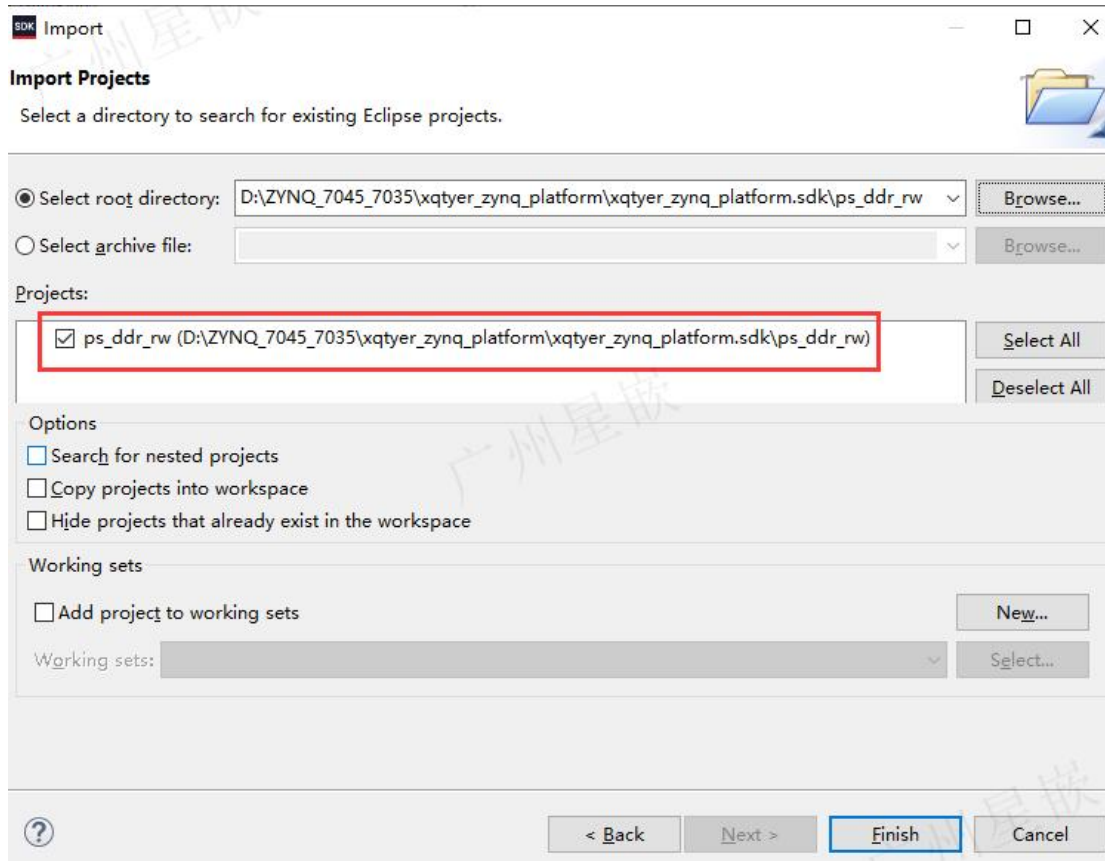
导入时，选中导入已有工程：



然后浏览找到 Xilinx SDK 示例工程，如下图所示的 ps_dds_rw 示例工程：

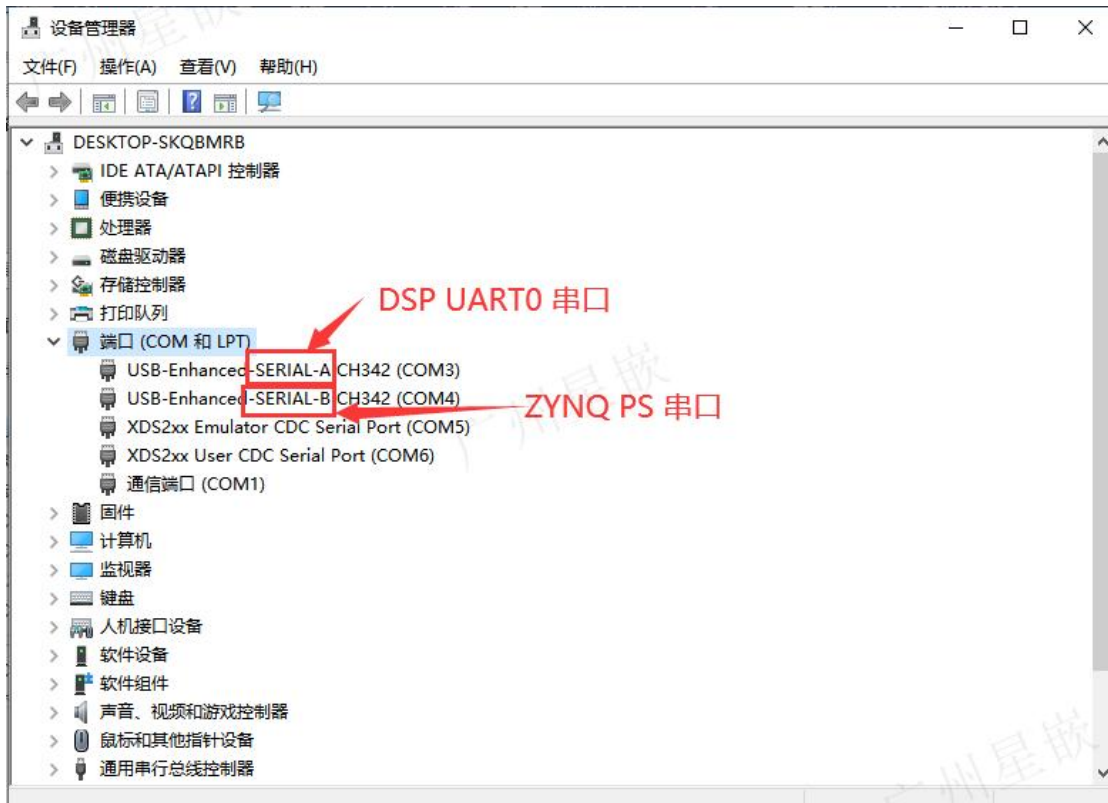


在 Projects 中列出了在所浏览的目录中发现的 Xilinx SDK 工程，点击勾选上前面的方框，然后点击 Finish 完成 Xilinx SDK 工程导入（**注意：如果 Xilinx SDK 工作区内已经存在此工程，则 Projects 中列出的工程项目为灰色条目，用户无法继续导入操作**）：

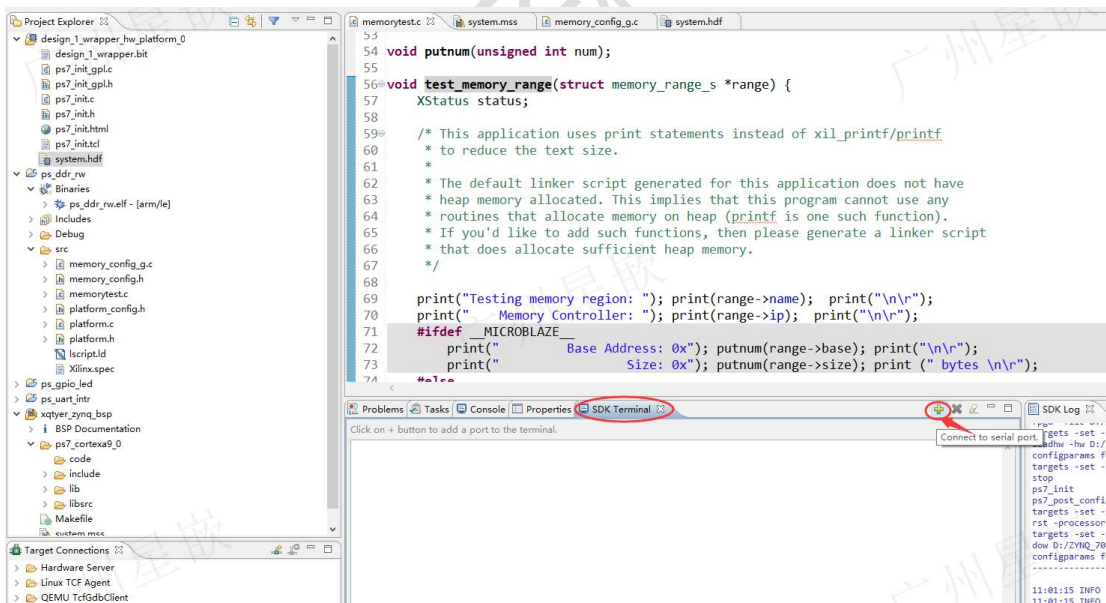


6.3.3.2 程序运行测试

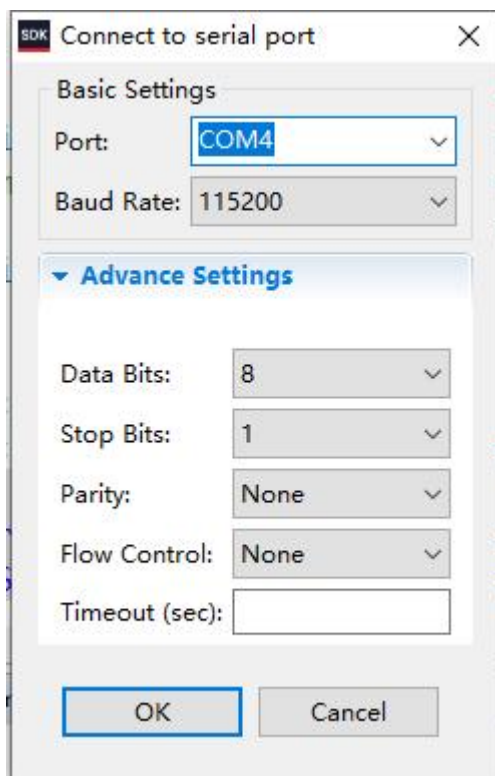
- (1) 板卡连上 JTAG 仿真器，拨码开关 SW2 将 ZYNQ 启动模式设置为 JTAG 启动，然后给板卡上电。
- (2) 电脑端打开设备管理器查看串口号。带 SERIAL-A 字样的为 DSP 端串口，带 SERIAL-B 字样的为 ZYNQ PS 串口，记住后面的 COM 编号，这里 ZYNQ PS 端的串口编号为 COM4:



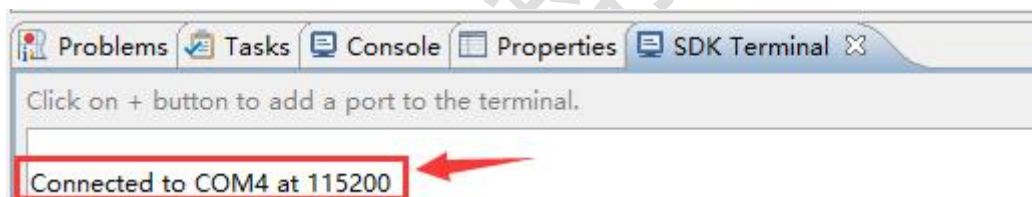
- (3) 打开 Xilinx SDK 自带的串口终端，如下图所示的 SDK Terminal，点击 SDK Terminal 终端界面的加号 $+$ ，连接串口：



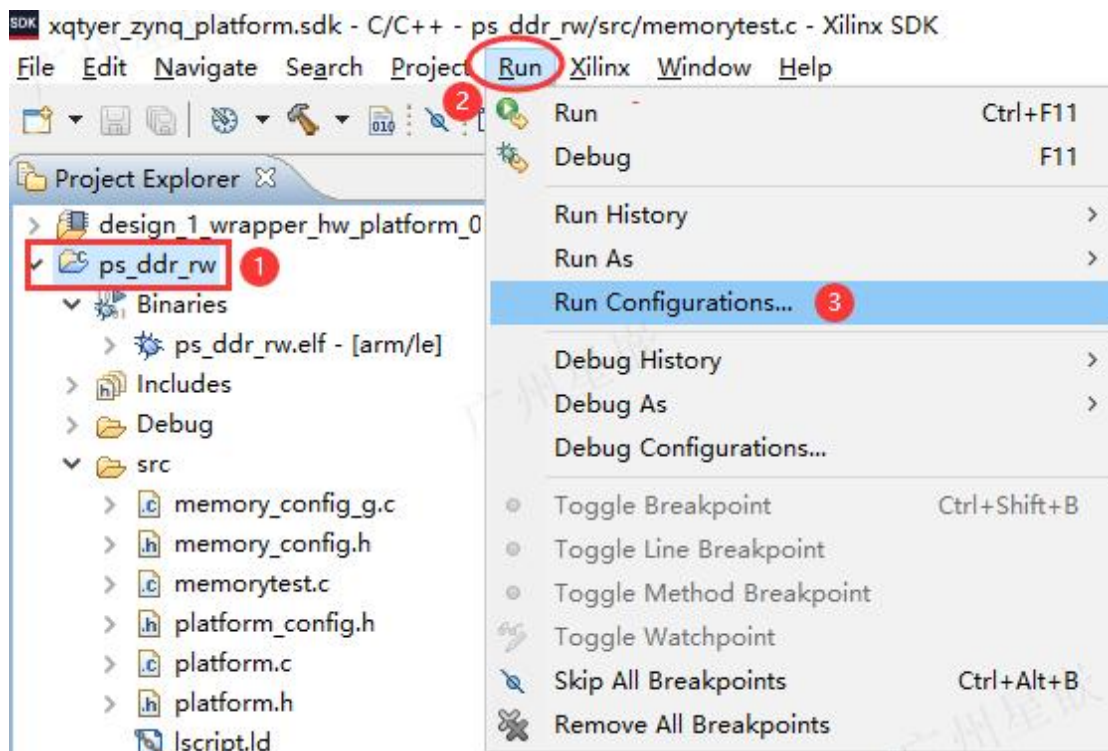
按照前面查看到的 COM 号设置串口端口号，波特率设置为 115200，无校验位，数据位为 8 位，停止位为 1 位，设置好后点击“OK”：



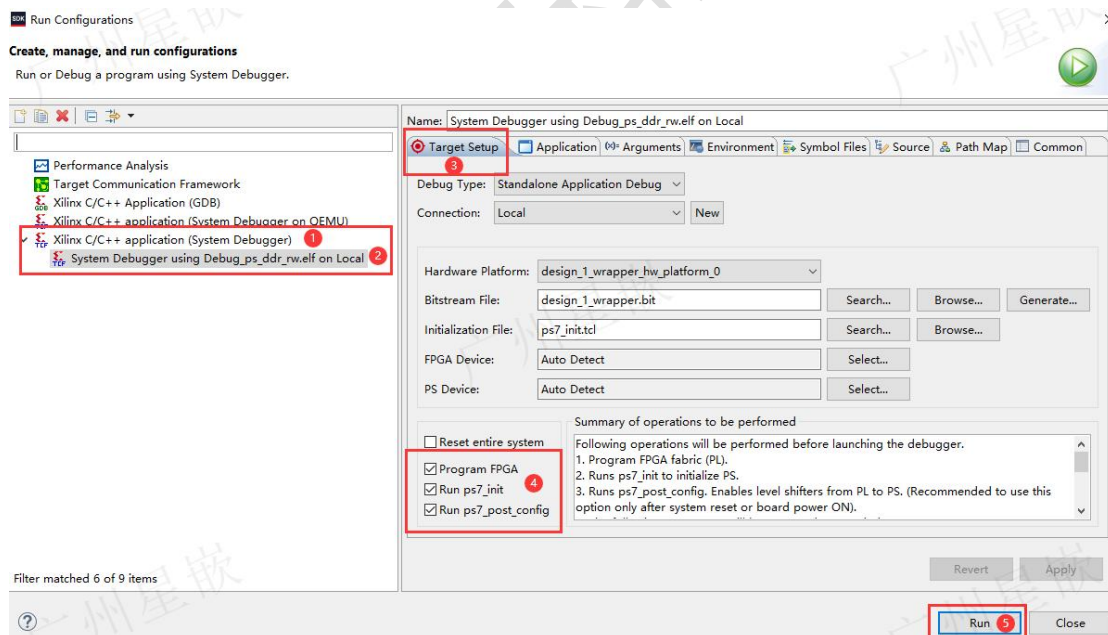
串口连接成功后，显示如下图所示：



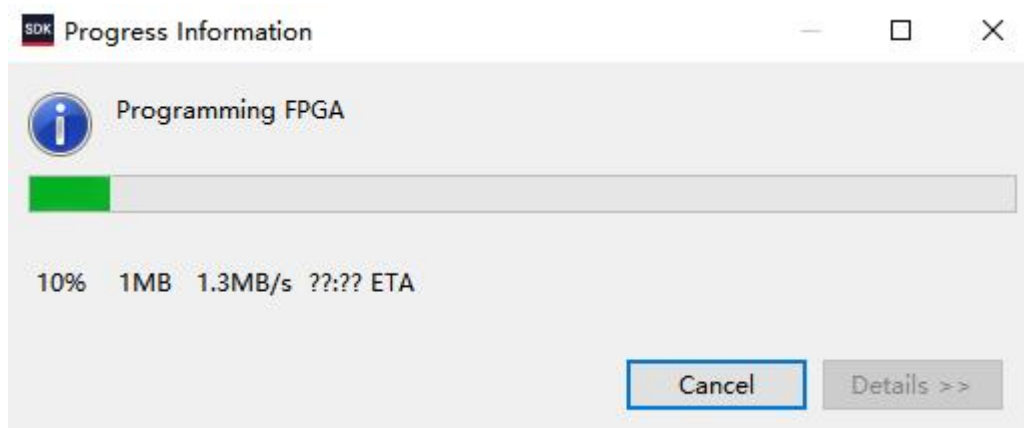
- (4) 下载和运行应用程序
在 Xilinx SDK 软件界面，点击选中应用工程，然后点击 Run->Run Configurations...:



双击 Xilinx C/C++ application(System Debugger) ，选中 Xilinx C/C++ application(System Debugger)下面的 System Debugger using Debug_ps_ddr_rw.elf on Local，然后点击在右侧界面的 Target Setup 一栏，并勾选上下图所示的④号红框内的三个选项，最后点击 Run:

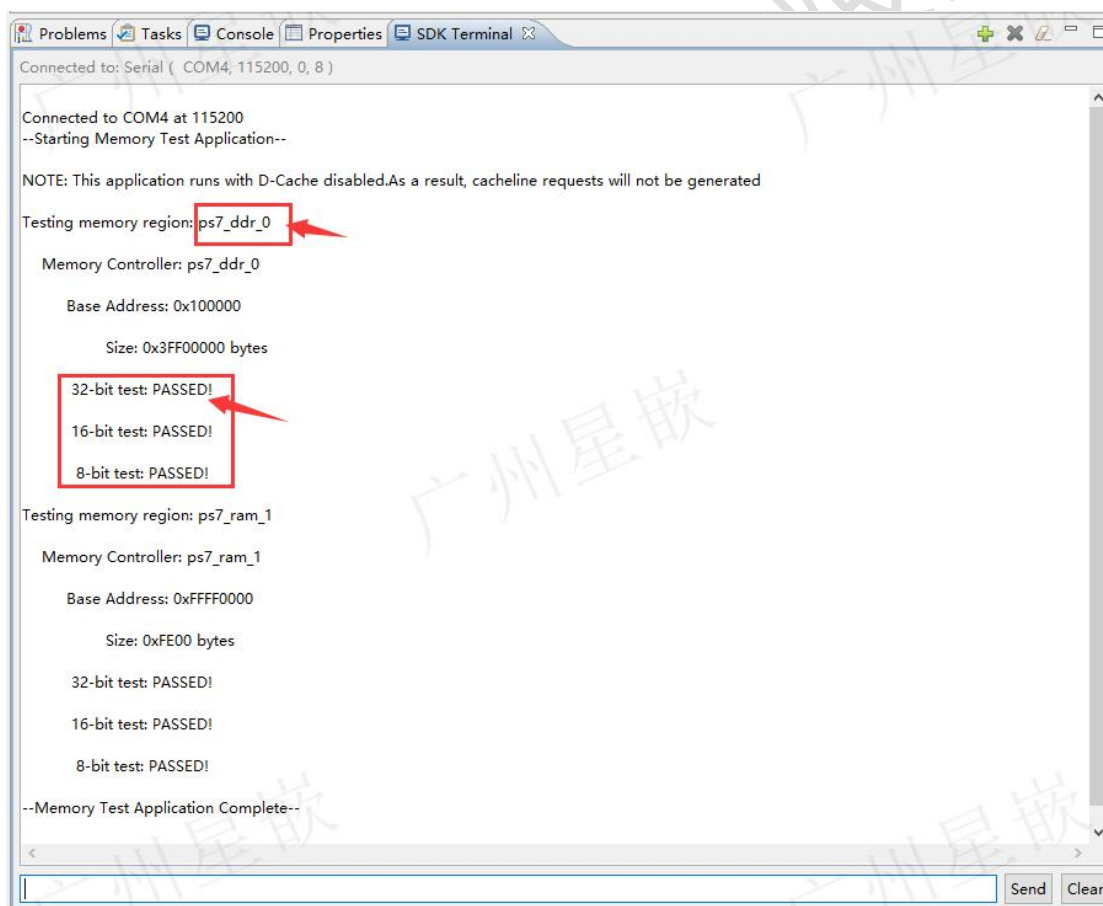


点击 Run 运行后，首先将 bit 流文件下载到 ZYNQ PL，下载过程如下图所示:



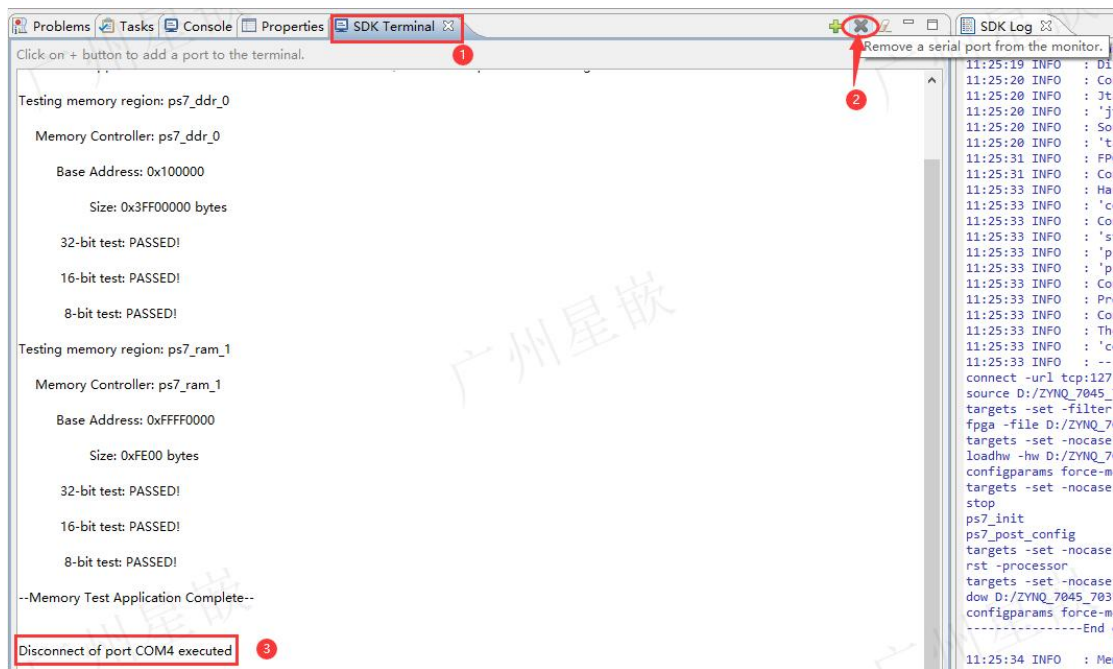
ZYNQ PL 端 bit 流文件下载完成后，则自动开始运行用户应用程序。

- (5) 用户应用程序下载运行后，可通过查看串口终端打印信息来判断内存读写测试情况，从下图打印信息看，DDR3 内存读写测试成功：



6.3.3.3 结束实验

点击 SDK Terminal 界面上的叉号 **X**，断开串口连接。串口断开连接后：SDK Terminal 窗口打印信息 Disconnect of port COM4 executed:



最后，关闭板卡电源，实验结束。