

## NIOS II 开发实用手册

--小梅哥 NIOS II 开发手记

### 前言

从开始接触 Altera（现在应该叫 intel PSG 了）的 NIOS II 处理器，到现在，已经有 10 个年头了。从开始的 C 语言都不懂，到现在能使用 NIOS II 开发一些实用的东西，中间的过程也是非常的曲折。最开始的时候，完全是炼狱一般，走一步，十个坑，没人指导，填几天，再走一步，再填一个坑。到了后来对这个东西开始心生敬畏，敬畏不是因为它有多么多么强大，而是在学习和使用它的过程中，让我对 CPU 架构，单片机系统实现思路和编程方法有了较为底层的认识，也算是一个升华吧，虽然在这个过程中还是常常掉入坑里好久才能爬出来。到了现在，能够指导大家学习和使用 NIOS II 处理器结合 FPGA RTL 逻辑实现一些功能，自己也能做一些不大的小东西。这 6 年，感觉就像是按照指数函数的曲线进步的，最开始很慢，后面越来越快。想想自己能坚持到现在，可真不容易（偷笑）。

深知大家在自学这门技术的开始半年时间内有多么痛苦。我一早就想出一套 NIOS II 方面开发的实用性书籍文档，可一直一个人打理着各种事情，实在没有精力。我也深知当下讲解和使用 NIOS II 的开发已经有些不那么前沿了，毕竟现在嵌入硬核的 FPGA 应用已经较为成熟了，NIOS II 这个处理器处于中间这样一个尴尬的位置，实用性和性价比值得思量。但是，毕竟 NIOS II 和 Xilinx 的 MicroBlaze 处理器设计和开发思路异曲同工，MicroBlaze 和 Xilinx 当前非常受欢迎的 Zynq 硬核 FPGA 开发思路和过程很像，NIOS II 和 Intel（Altera）的 SOC FPGA 开发过程和思路很像，因此，学习 NIOS II 处理器，是一条经济轻巧的道路。真正掌握了 NIOS II 处理器的应用和开发，迁移到 Intel SOC FPGA 上，也就需要 3~5 周的时间，换到 Xilinx 的 MicroBlaze 或者 Zynq 平台上也只需要 5~8 周。所以，这个事情值得一做，毕竟，咱国家每年还有那么多高校学子需要一个合适的切入点来进入 SOPC/SOC 开发的大门。

当下定决心做这件事的时候，我却犯了难。到底，我该以一种怎样的方式来开启我的系列文档呢？是从 CPU 架构、指令集开讲，还是直接从 LED 点灯写起呢？每天在技术交流群里，看到大家学习时候遇到各种问题却不知道如何解决时，我突然觉得，其实，大家暂时不缺入门的教程，缺的，是继续学习下去的信心。那么信心从何而来？就从解决 NIOS II 开发中常见的各种问题着手开始吧。如果大家首先就有一份“捉虫子”手册放在旁边，遇到各种问题马上能从手册中找到解决方法或解决思路，那么大家的学习信心必然会与日俱增。所以，我选择我的这一份文档，从教大家捉虫子开始。

唠叨了这么多，下面开始切入正题。

大家在进行 NIOS II 系统开发时，往往有很多的疑问，第一个疑问就是大家经常提到的，NIOS II CPU 运行不稳定，NIOS II 开发 bug 太多。由于 FPGA 本身相对于单片机、ARM 处理器来说，应用市场要小的多。加上 NIOS II 仅仅是 FPGA 应用和开发的一个小的分支，用的就更加的少了。所以，关于 NIOS II 非常系统且科学的教程教材也是非常的少，导致大部分人在进行 NIOS II 的学习和开发的时候，都会遇到各种各样的问题，如 elf 文件下载失败，CPU 运行不起来，程序运行不正确，CPU 运行一段时间后停止，调试正常但是烧写到 EPCS 后无法运行等。那么今天，我就在这里将 NIOS II CPU 的各种问题做一

个总结分析，希望大家通过本总结分析，以后在开发 NIOS II 相关应用时候，能够手到擒来。

## 预备知识--良好的开发和编程习惯

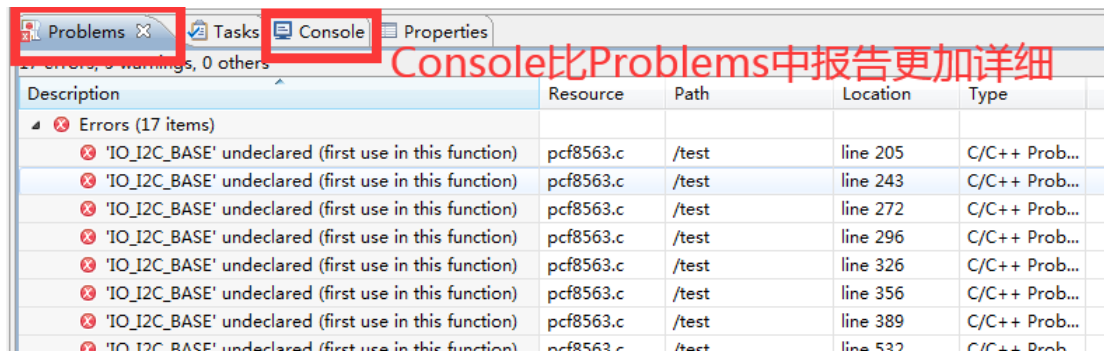
### 学会查看 NIOS II C 程序编译报错信息

NIOS II 的 C 程序开发时，其思路和调试方法与普通的 MCU 开发思路其实并无太大差别，而我们在开发 C 程序的过程中，免不了会因为各种原因出现一些错误，当出现这些错误之后，NIOS II 的 C 程序开发工具（基于 eclipse）会提供详细的报错信息，但是很多人都不知道如何分析这些报错信息，本节将有理有据的带领大家分析 NIOS II 的各种编译报错信息。

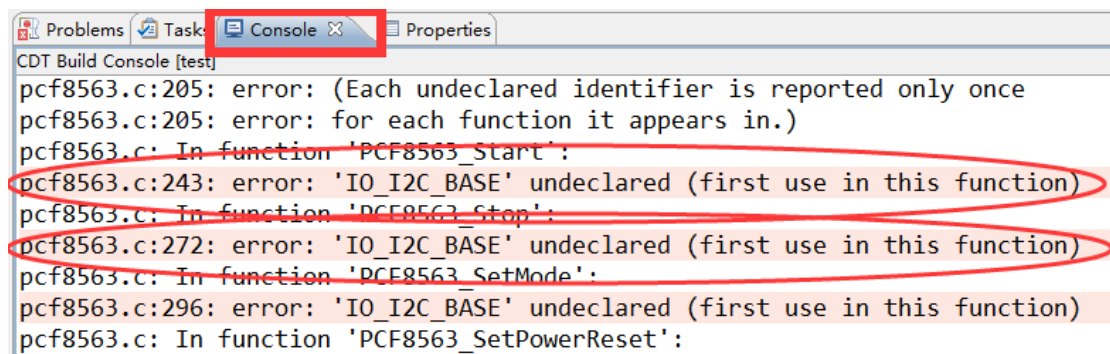
编译信息在 console 而不在 Problems 窗口!!!!

比分析报错并解决更为重要的是，如何查看编译报错信息。因为在实践中发现，有很多人根本都不知道该去哪儿查看编译报错信息，或者说他们认为的编译报错信息实际都是不对应的。

你知道去哪儿查看编译和报错信息吗？很多人只知道在 Problems 栏去查看报错信息，事实上，从实际使用来看，Problems 窗口往往只能给出总结性的报告，对于细节很多情况下报告的并不详细，真正详细的报错实际是在 Console 窗口，如下图所示。

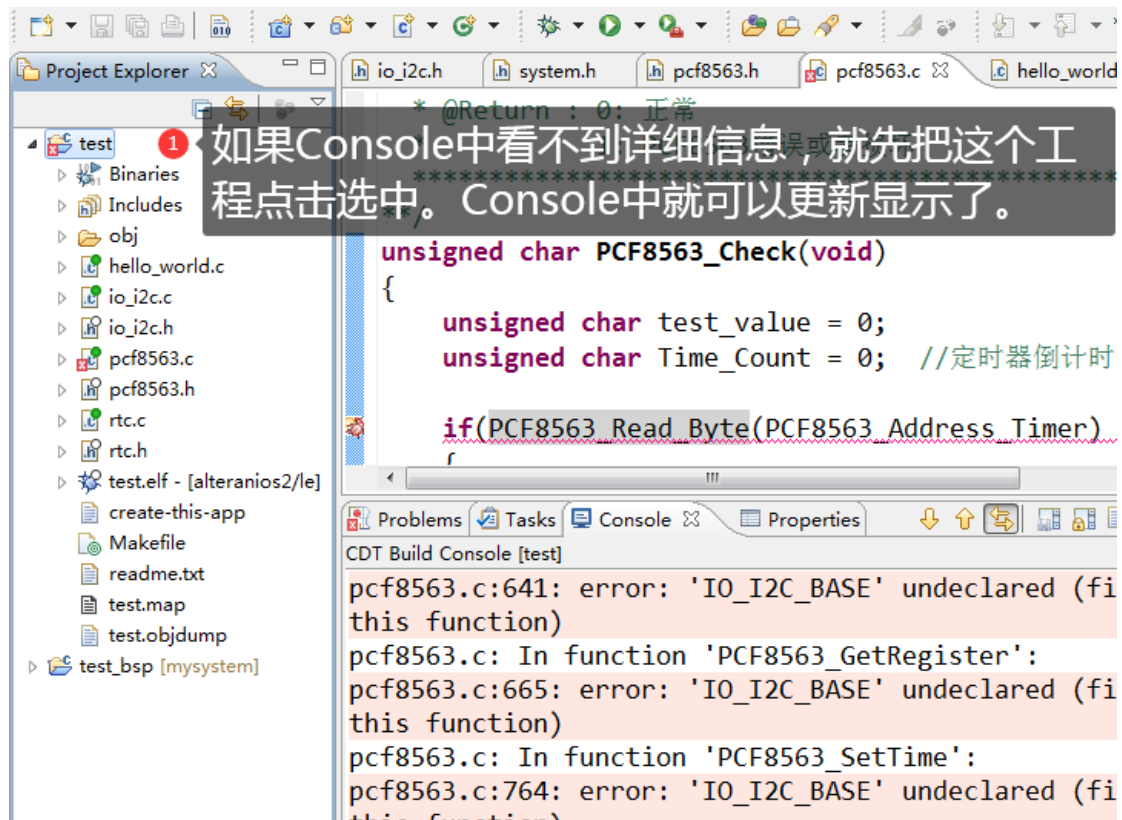


虽然上图中 Problems 中的报错信息已经差不多够我们定位问题了，但是如果切换到 Console 窗口，会发现更加细致的报错。



如果切换到 Console 窗口后没有看到详细的报错信息，可能与当前选择查看的公司是 BSP 工程而不是用户应用工程有关，此时，先在左侧点击一下你的应用工程，再看

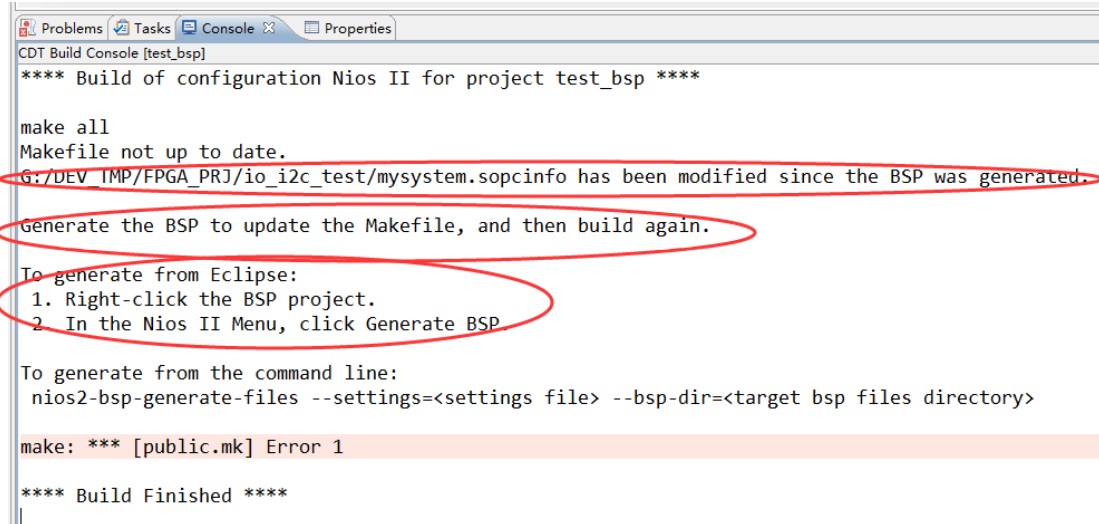
Console 中，就能看到相关的报告信息了。



再举一个例子，在下图中，Problems 窗口中几乎看不出很直接的提示信息，通过这个消息一般人很难知道问题出在哪里。



但是如果我们切换到 Console 窗口中呢？如下图所示。



```
CDT Build Console [test_bsp]
**** Build of configuration Nios II for project test_bsp ****

make all
Makefile not up to date.
G:/DEV_TMP/FPGA_PRJ/io_i2c_test/mysystem.sopcinfo has been modified since the BSP was generated.
Generate the BSP to update the Makefile, and then build again.

To generate from Eclipse:
1. Right-click the BSP project.
2. In the Nios II Menu, click Generate BSP.

To generate from the command line:
nios2-bsp-generate-files --settings=<settings file> --bsp-dir=<target bsp files directory>

make: *** [public.mk] Error 1

**** Build Finished ****
```

看看，图中用非常明确的信息说明了，在上次 BSPgenerate 之后，sopcinfo 文件被修改了，然后提供了解决的方法，既通过生成 bsp（generate bsp）的操作来自动更新 Makefile 文件，然后再重新编译即可。怕你不知道如何 generate bps，还列出了 generate bsp 操作的详细步骤，先右键单击 BSP 工程，然后再在弹出的窗口中，NIOS II 选项下，点击 Generate BSP 即可。

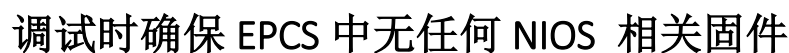
## 正确的打开和使用已有工程文件

当使用一个现成的工程，如我们提供的例程，需要注意使用正确的方法打开，具体的打开方法，请参见本文档的“解决 NIOS II 工程移动在磁盘上位置后 project 无法编译问题”一节的内容。

## 代码先保存再编译

改了代码先保存，再编译，先保存，再编译，由于 Eclipse 环境本身默认没打开编译前自动保存文件的功能，如果再加上用户没有好的习惯，不细心，就会出现你本身修改了代码，但是在执行编译的时候，由于没有将修改的代码文件执行保存，导致编译的时候实际编译的还是没有修改之前的内容，所以就会出现各种看着违反常理的报错。避免此问题要牢记编译前先对所有修改了的程序文件执行保存操作，养成良好的习惯。

当然，也可以在每次工程创建好之后，在软件中设置编译前自动保存选项，如下图所示。



保证 EPCS FLASH 中没有任何与 NIOS II 相关的固件，可以通过擦除 EPCS 存储器的方法实现。为啥要擦除，如果 EPCS 里面存储有与 NIOS II 相关的固件，就有可能导致调试时，Eclipse 将程序下载进板子后触发板子复位，而板子复位后受 EPCS 固件中的 NIOS II 启动代码影响，转而去执行 EPCS 中存储的程序代码。如果不懂原理，不要紧，按我说的做，准没问题。关于如何擦除 EPCS 内容的问题，可以参见本文档中的“擦除 Altera FPGA 的配置器件 EPCS 中内容的方法”一节的内容。

## 遵循固定的调试流程

每次调试时按照下述流程进行:

- 1、开发板断电
- 2、开发板上电
- 3、下载 sof 文件
- 4、在 eclipse 中执行 run 或 debug。

## 正确的提问求助方法

提问求助请一定准确的描述你的问题，提供尽可能准确的故障或报错信息，方便我们分析。只有准确合理的提问内容，才让解答者有解答的欲望。



## 新手易犯错误集合

### clk\_0 模块的 clk\_in 信号和 clk\_in\_reset 信号连接错误

正常这两个信号应该是导出到 Qsys 系统顶层，然后在 Quartus II 软件中连接的，部分网友在连接复位地址时习惯于使用菜单栏中的自动连接复位信号（Create Global Reset Network）操作连接复位信号，如图 1 所示。该操作实际并不是那么智能，会将 clk\_in\_reset 信号连接到所有的复位网络上，这就产生了错误。正确的是使用该操作自动连接复位信号后，再在 Export 栏将 clk\_in\_reset 信号双击以导出，如图 2 所示。否则生成系统时会报错。

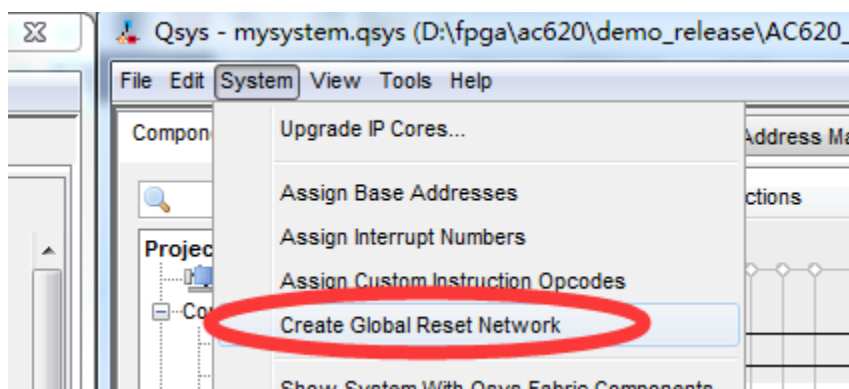


图 1 自动分配复位网络操作

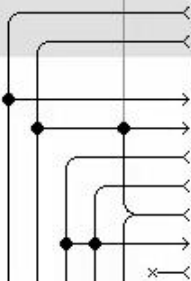
Connections	Name	Description	Export
	clk_0	Clock Source	
	clk_in	Clock Input	
	clk_in_reset	Reset Input	clk
	clk	Clock Output	
	clk_0.clk_in_reset	Output	
	Reset Input [reset_sink 13.0]	Processor	
	clk	Clock Input	Double-click to export
	reset_n	Reset Input	Double-click to export
	data_master	Avalon Memory Mapped Master	Double-click to export
	instruction_master	Avalon Memory Mapped Master	Double-click to export
	jtag_debug_module_re...	Reset Output	Double-click to export
	jtag_debug_module	Avalon Memory Mapped Slave	Double-click to export
	custom_instruction_m...	Custom Instruction Master	Double-click to export

图 2 正确的信号连接方式

### NIOS II EDS 创建工程时 sopcinfo 文件选择错误

由于 Eclipse 软件默认会记录上一次的各种操作，这就给我们本身创建和编辑新工程带来了隐患。当新建 NIOS II 的工程时，会要求指定 SOPC Information File name，点击浏览选项打开，如图 1 所示，默认会直接打开最近一次选择 sopcinfo 文件的路径，设计者必须准确核对路径，并重新切换到正确的路径下选择当前 FPGA 工程对应的 sopcinfo 文件，如图 2 所示。否则创建的软件工程和 FPGA 中的逻辑设计对不上，就会在下载 elf 程序时出

现程序无法下载，如图 3 所示。或者下载之后程序运行不正常的情况。

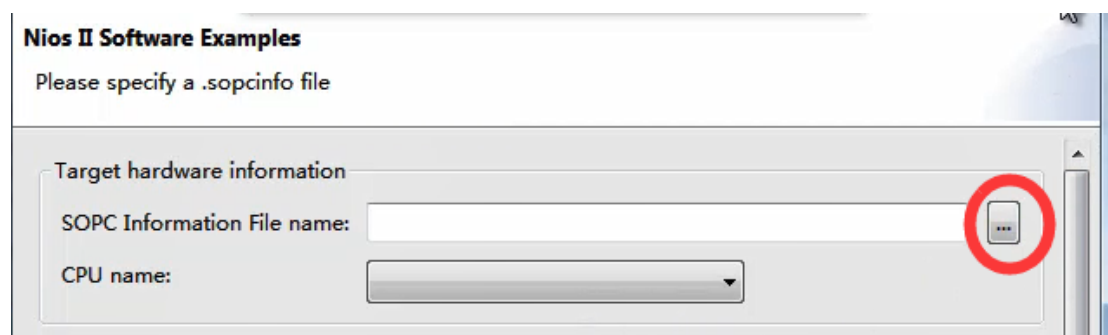


图 1 选择 socpinfo 文件

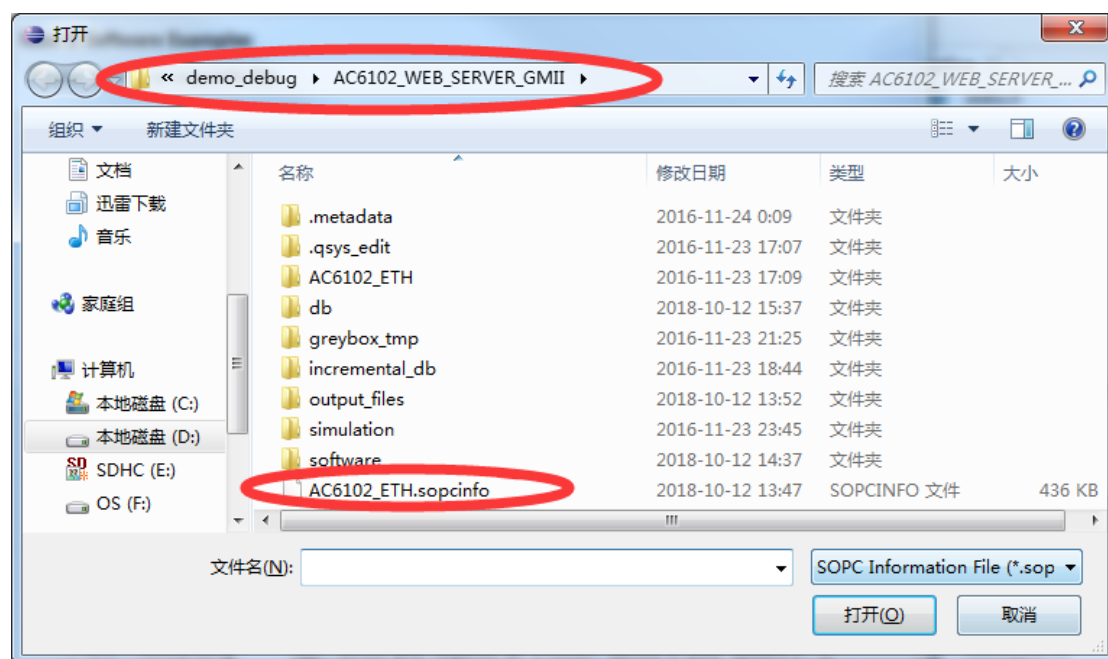


图 2 核对并切换到正确的文件路径下

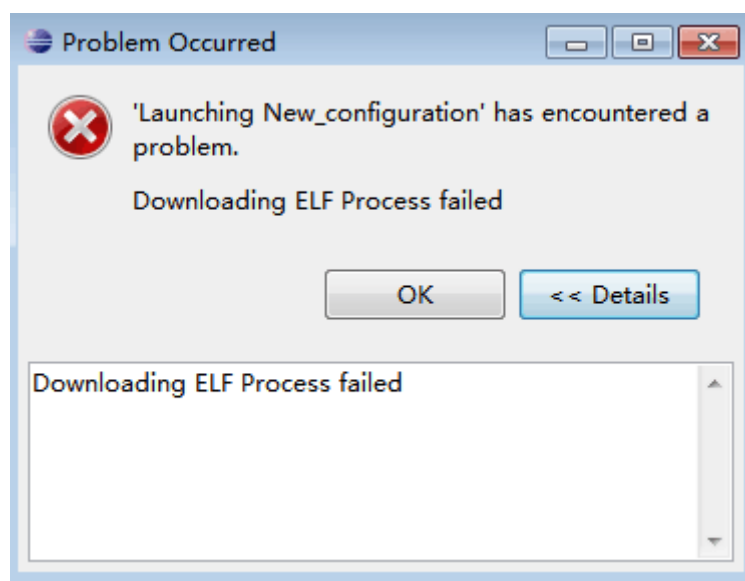




图 3 下载程序失败提示

## hello\_world 和 hello\_world\_small 模版工程差别

用户在创建最简单的应用工程时，一般可以通过创建模版工程，并在模版工程上进行修改来得到自己的设计，这种方法速度快，而且能够实现设置好各种编译和库参数，不用用户再自行指定，非常的方便。创建工程时有两个最常用的工程模版：`hello_world` 和 `hello_world_small`，如图 1 所示。

`Hello_world` 模版包含了非常全面的底层支持库，而且关闭了程序编译时候的优化选项。这些底层库能够非常方便的实现各种操作，但是，也因为包含了这么多库，导致生成的程序占用存储空间很大，一般使用 FPGA 片上 RAM 做 NIOS 内存的时候，是装不下这么大的程序的。因此使用该模版一般需要使用 SDRAM 或者 DDR SDRAM 作为内存时方可。

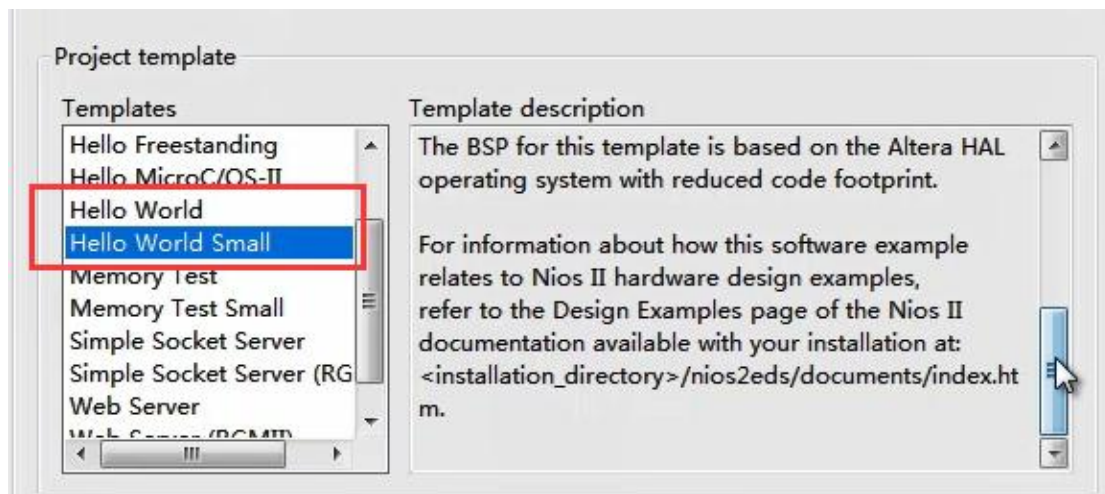


图 1 使用模版工程快速创建应用工程

`Hello_world_small` 模版精简了很多底层支持库，仅包含了一些必须，而且是轻量级的库。而且打开了程序编译时候的优化选项。该使用该模版工程，编译生成的程序占用存储很小，一般使用 FPGA 片上 RAM 做 NIOS 内存的时候，需要使用该模版创建工程。当然该模版创建的工程也可用于使用 SDRAM 等外部存储器作为内存的系统。不过受限于该模版提供的各种库的性能，一般不适合做各种全功能，高性能的应用。而且使用该模版工程时，设计者需要尤其关心程序设计时的潜在被优化可能，例如使用如下程序实现延时就会达不到效果：

```
void DelayNs(alt_u32 i)
{
    while (i--)
        ;
}
```

而使用下述方式，即可实现相应功能，即使用

```
void DelayNs(alt_u32 n)
```

```
{  
    volatile alt_u32 i=n;  
    while (i--)  
        ;  
}
```

## 修改或者编译 Quartus II 工程后编译 NIOS II 软件程序报错

当用户重新编译了对应的 Quartus II 工程，及时没有做任何修改，只是重新编译，然后再回到 NIOS II EDS 软件中编译程序，会弹出如图 1 所示报错，然后弹出如图 2 所示的解决方案。这是因为为了保证 NIOS II 软件工程和 Quartus II 工程时刻保持一致状态，避免出现软硬件配置不一致而引发各种程序执行错误，因此在编译 NIOS II 软件工程时会首先检查当前的 Quartus 工程版本信息（Quartus 工程每次编译都会更新 sopcinfo 文件中一个唯一标识 ID，每次编译后该 ID 都会发生变化），如果与当前软件工程所记录的版本信息不一致，就会提示图 1 所示的错误。解决方法很简单，按照软件提示的操作即可解决，如图所示，选中 bsp 工程，右击选择 NIOS II 下的 Generate BSP 选项，然后待生成完成后再编译，就不会报错了。

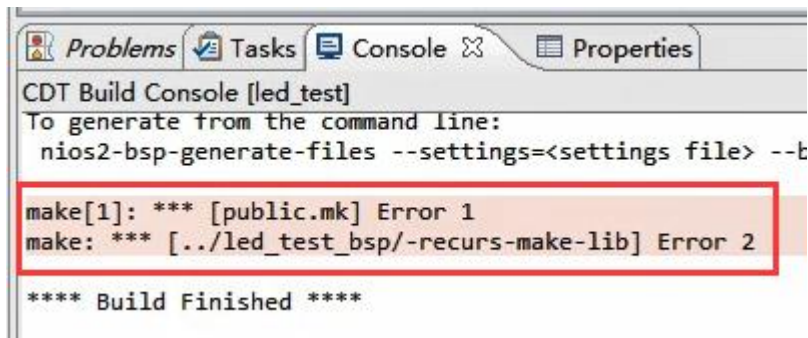


图 1 Quartus 重新编译后软件工程编译报错

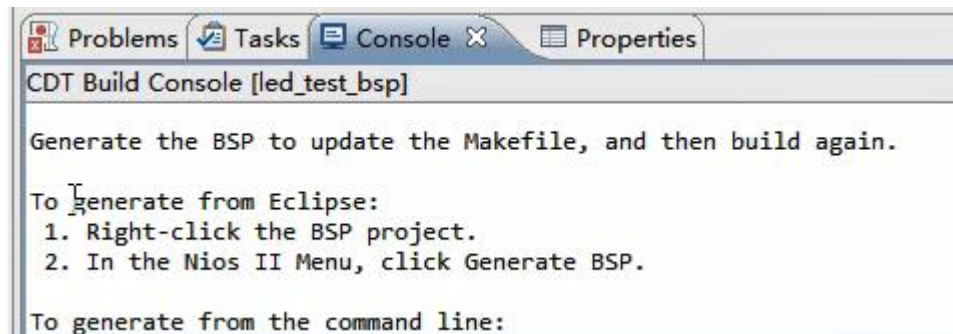


图 2 软件给出的解决方案提示

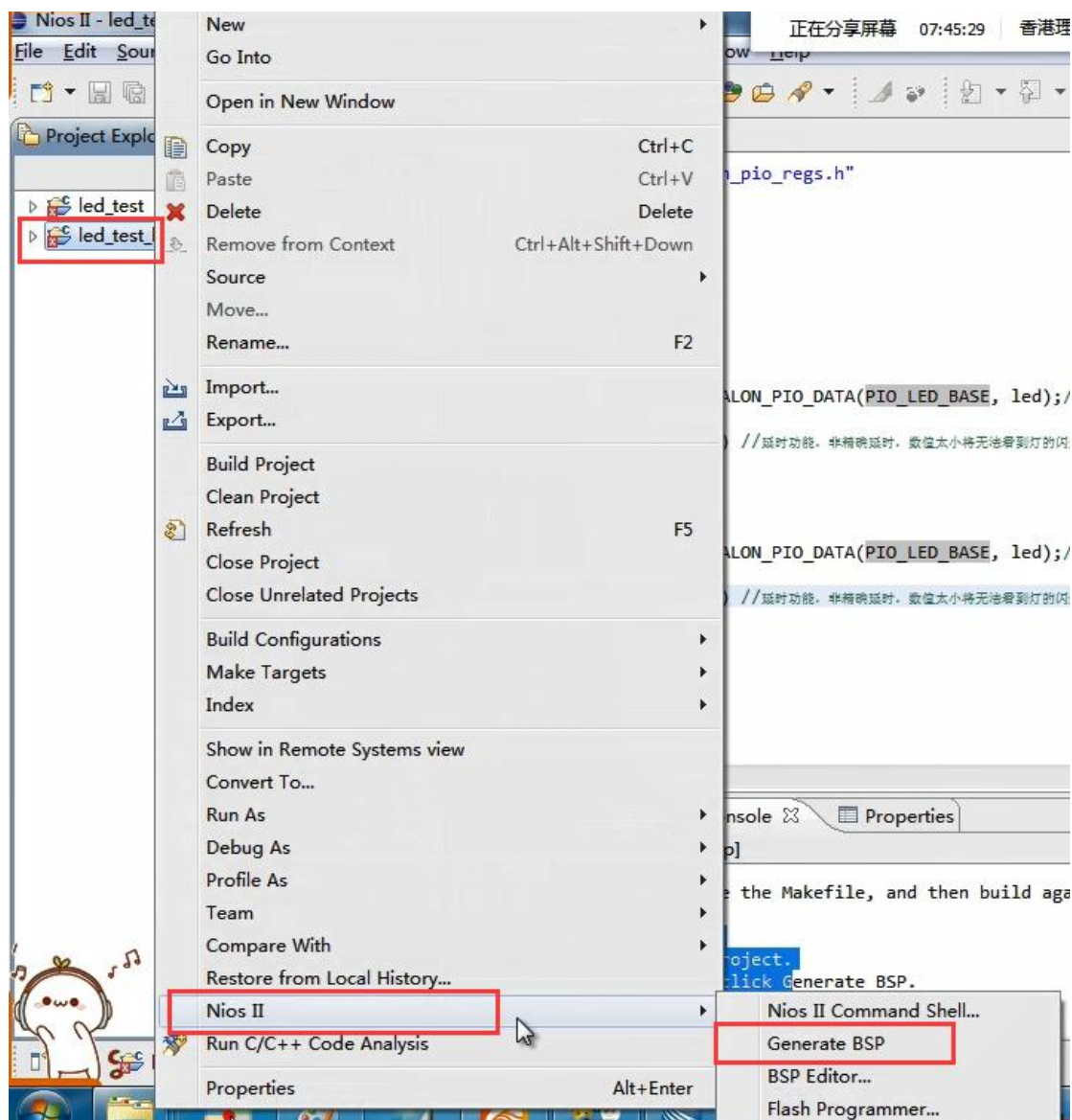


图 3 重新生成 BSP 工程操作

## C 程序软件与硬件设备基地址

很多同学在刚开始学习 SOPC 技术时，因为缺少经验积累，因此习惯对着例程直接照抄代码，这本身也是一种可行的学习方案，只是在抄代码的时候，因为对代码的理解不够透彻，导致代码中有些设计到硬件相关的参数，需要根据用户自己的工程中相关参数进行相应的修改替换。例如点亮 LED 例子中，使用到了一个名为 PIO\_LED\_BASE 的名称，如图 1 所示。该值实际上是我们在 Qsys 系统中添加的 pio\_led 外设的基地址，很多用户在 Qsys 中搭建系统时，命名可能与我们的教程例子不一致。例如用户添加的控制 LED 的 PIO 并不脚 PIO\_LED，可能叫 LED\_PIO，总之就是与例程不一致，这时候如果直接复制例程的 C 程序源码，编译就会提示 PIO\_LED\_BASE 找不到的错误。正确的操作方式应该是打开 system.h 文件，找到该文件中对该信息的定义，复制到 C 代码中替换 PIO\_LED\_BASE，然后就能正常编译了，如图 2 所示。（如果用户定义的驱动 LED 的管脚 LED\_PIO，那么 system.h 中该信息应该叫 LED\_PIO\_BASE）

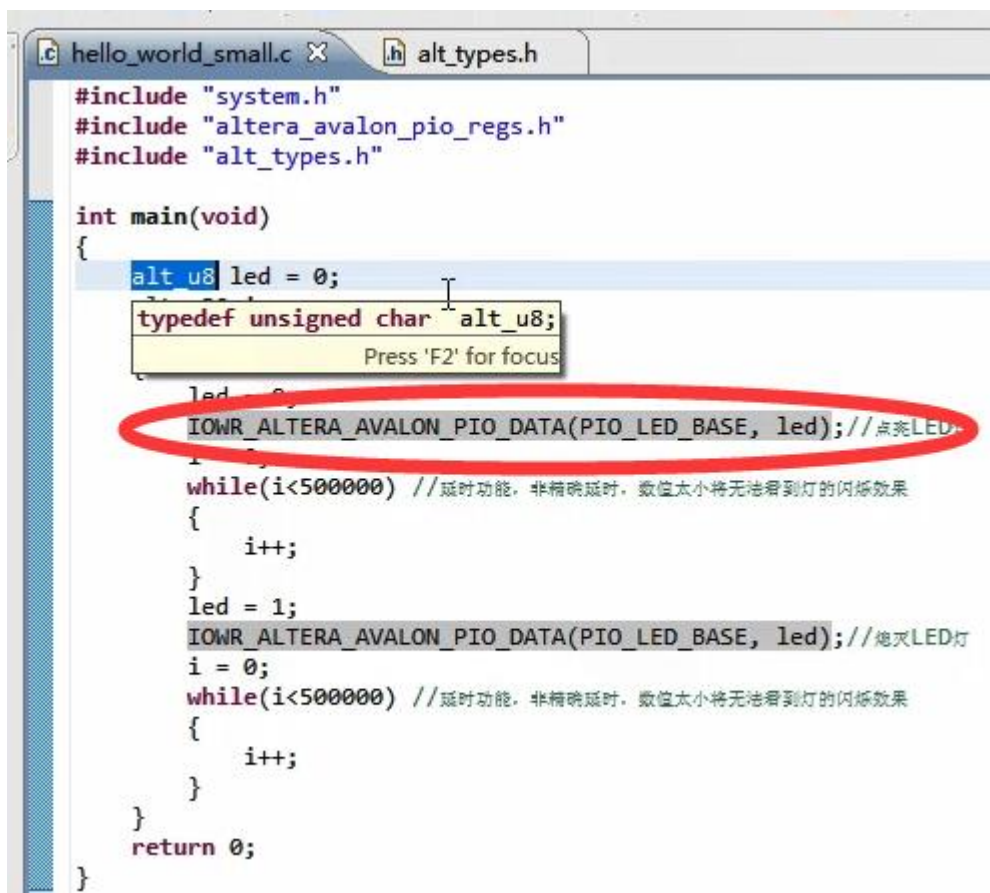


图 1 与硬件相关信息

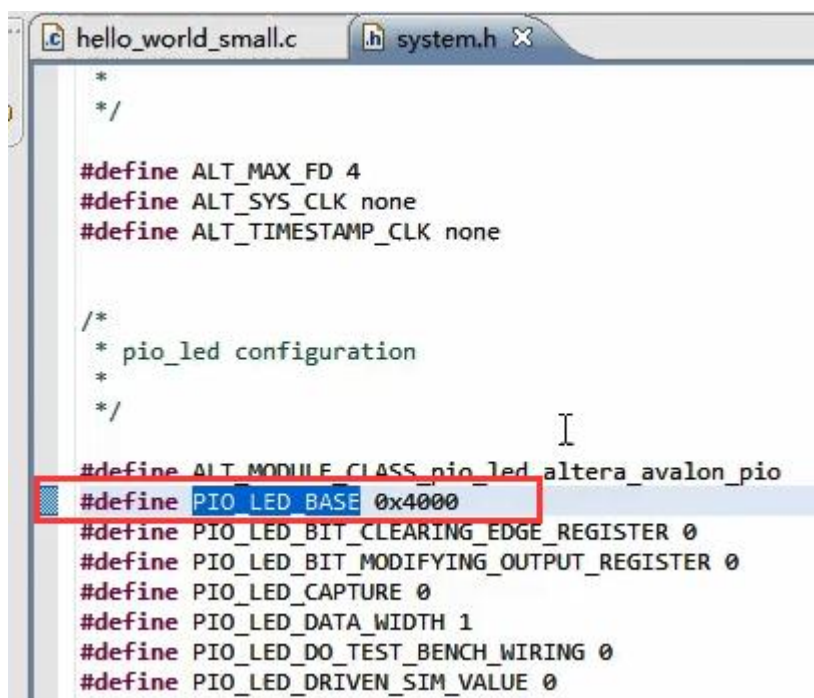


图 2 查看具体硬件信息



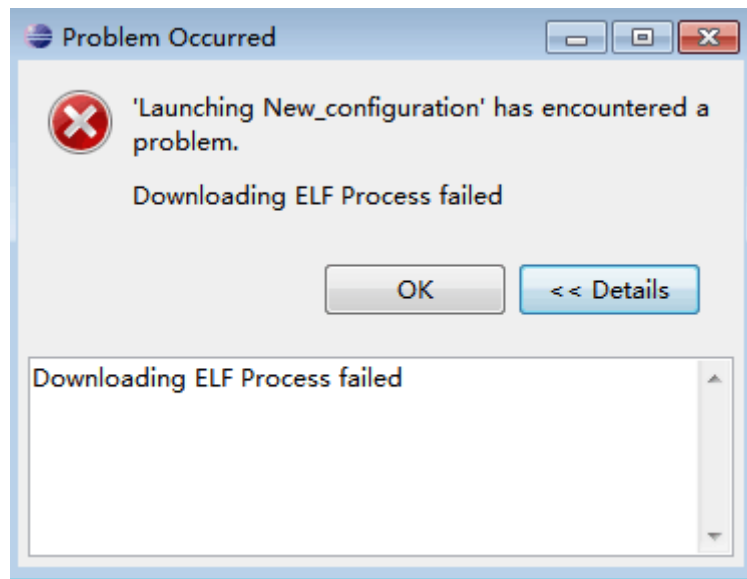


## 常见报错和异常问题

### elf 文件下载失败

当大家将 FPGA 的编程文件 SOF 文件下载到 FPGA 芯片中后，就可以下载 nios ii eds 中编译好的软件固件，该固件的尾缀为.elf。那么大家在下载的时候，会经常遇到下述问题：

Launching New\_configuration has encountered a problem. Downloading ELF Process failed。



原因分析：

个人目前总结的，出现该问题的原因主要有以下几点，

- 1) 目标板/开发板上的 NIOS II CPU 系统未能正常工作，例如，未下载对应的 SOF 文件到 FPGA 中，或者时钟、复位、存储器（SRAM、SDRAM、DDR2）引脚分配有误，当使用 SDRAM 存储器作为 NIOS II CPU 的内存时，还有可能是 SDRAM 的控制器时钟和接口时钟之间的相位差不合适。

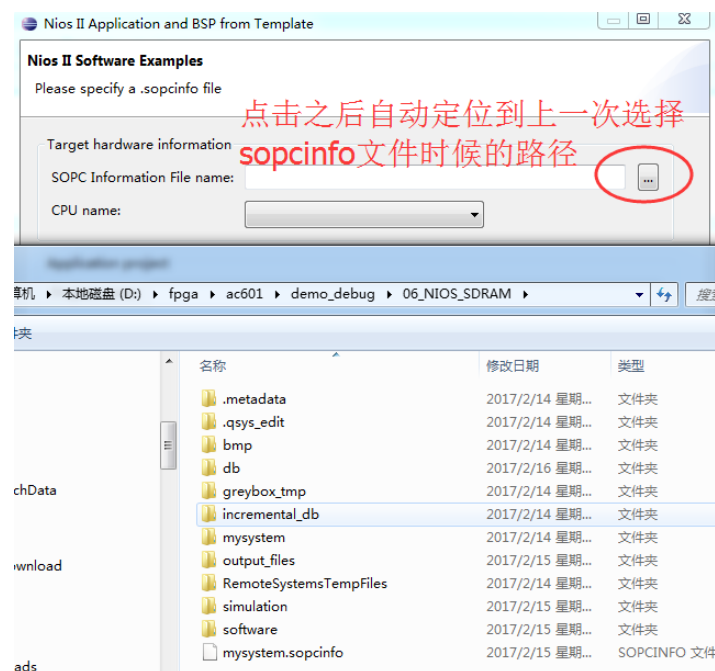
**解决方法：**重新下载 sof；检查核对引脚分配；查看 SDRAM 参数配置和时钟配置。

- 2) 创建 NIOS II 软件工程时候，选择的 sopcinfo 文件与对应的 FPGA 工程不一致。这一点我在每次公开课的时候都会强调，NIOS II 软件开发需要两个工程，一个板级支持包（BSP）和一个应用工程。每次创建 NIOS II BSP 工程的时候都需要选择一个 sopcinfo 文件，该文件就是实际我们在 Qsys 中搭建的希望使用的 NIOS II 系统的描述文件。NIOS II 软件开发环境根据该信息文件创建对应的硬件信息头文件“system.h”，但是，NIOS II 开发软件有一个比较不好的地方就是每次选择 sopcinfo 文件的时候，都会记录上一次选择 sopcinfo 文件的路径并直接自动定位到该路径，所以大家如果一旦粗心，一点击浏览文件，发现一个 sopcinfo 文件就直接选择的话，往往就会选择到上一次的工程的文件，而不是本次新的工程的文件。这样我们



创建软件工程时使用的 `sopcinfo` 文件就还是上一个工程的，而我们下载 `sof` 时又是下载的新的工程的，所以就出现了下载的 `sof` 文件与 `elf` 文件不是基于同一个工程的问题，导致无法下载成功。因此为了避免出现这个问题，新建工程时请时刻记住选择正确的 `sopcinfo` 文件。该问题一个更加奇妙的现象就是当前一个工程和这次的新工程两者之间差别不大的时候，`elf` 甚至可以正确下载，`NIOS II` 也能运行起来，但是就是现象与预期不一致，这一点可能往往也是让很多人误以为 `NIOS II` 不稳定的原因之一。如果你想知道你当前的工程的 `bsp` 文件是否正确，非常简单，打开 `bsp` 工程下的 `settings.bsp` 文件，查看第 9 行就可以啦。

**解决方法：**新建工程选择 `sopcinfo` 文件时务必选择正确的 `sopcinfo` 文件。

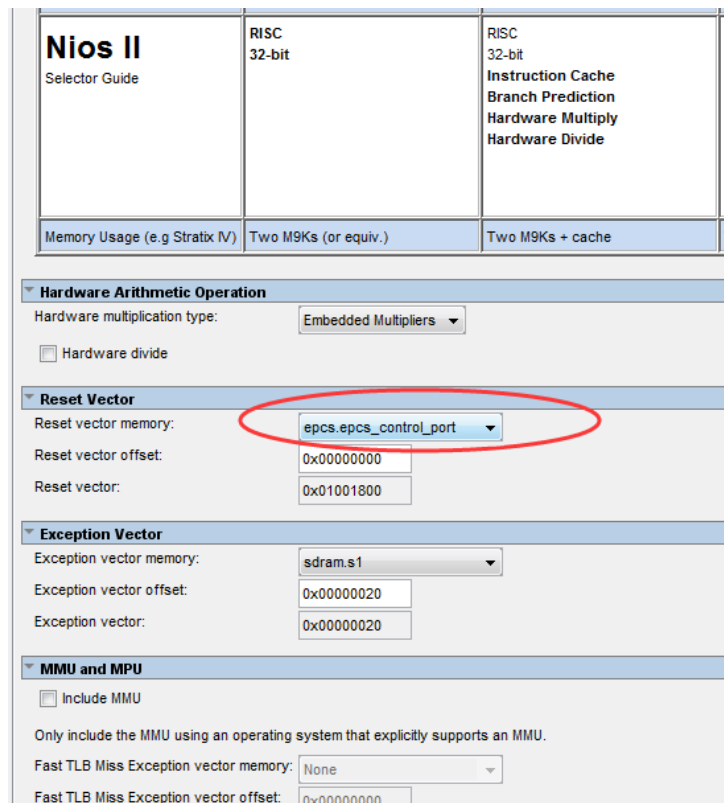


### 3) NIOS II 的启动地址错误。

`NIOS II` 是一个 CPU，其运行过程是受程序指令控制的，而程序有不同的存放位置。例如，程序可以直接存放在 `RAM` 中，然后 CPU 复位后直接从硬件定义的 `RAM` 中程序存放的初始位置开始执行。该种方式常见于我们在进行软件编写调试的过程中，这个过程，我们可能需要经常进行程序的调试，所以直接使用仿真器（`USB Blaster`）在线将程序下载到 CPU 的 `RAM` 中运行或者 `debug`。另一种情况就是项目发布的时候，我们做一个项目或产品，当产品功能都调试通过之后，需要将程序烧写在板卡上，这样板卡在上电之后，不需要 PC 端下载程序，就可以自动从非易失性存储器（`FLASH`、`EEPROM`）中加载程序并运行。此时我们需要 CPU 设置从非易失性存储器中开始启动。那么如果我们设置了 `NIOS II` 的启动地址为 `FLASH`（`EPCS`），而我们又下载了定义从 `RAM` 中启动的程序，那么程序会被下载到 `RAM` 中，CPU 启动时候会去 `FLASH` 中读取程序，由于我们的程序并没有下载到 `FLASH` 中去，因此 CPU 无法读取到正确的程序，就会无法正常运行，然后报此错误。同理，如果我们设置 CPU 从 `RAM` 中启动，而我们又将程序烧写在了 `FLASH` 中，那么 CPU 上电运行后，由于 `RAM` 中没有下载正确的程序，因此也无法运行。这一点实际上是我们上面提到的另一个现象，即调试正常但是烧写到 `EPCS` 后无法运行。

好了，饶了这么多口舌，该说大家最关心的问题，怎么设置 CPU 的复位地址

呢？其实有两个地方需要设置，而 90% 以上的人只知道一个地方，那就是 QSYS 中选择 CPU 的复位地址。当我们的系统中 FLASH 使用 EPCS 芯片剩余容量，那么如果我们要定义 CPU 硬件上从 FLASH 中启动，就需要定义 CPU 的 Reset Vector 为 EPCS，如果我们要定义 CPU 硬件上从 RAM（onchip\_ram/SDRAM/DDR2）中启动，那么就选择 Reset Vector 为 ram。当然，这没什么问题，但是我们任然可以在一开始就定义 CPU 的 Reset Vector 为 EPCS，然后在调试的时候，却从 RAM 中运行。为什么可以呢，这就是我说的 90% 人不知道的第二个地方，该设置在 NIOS II 软件开发环境中。

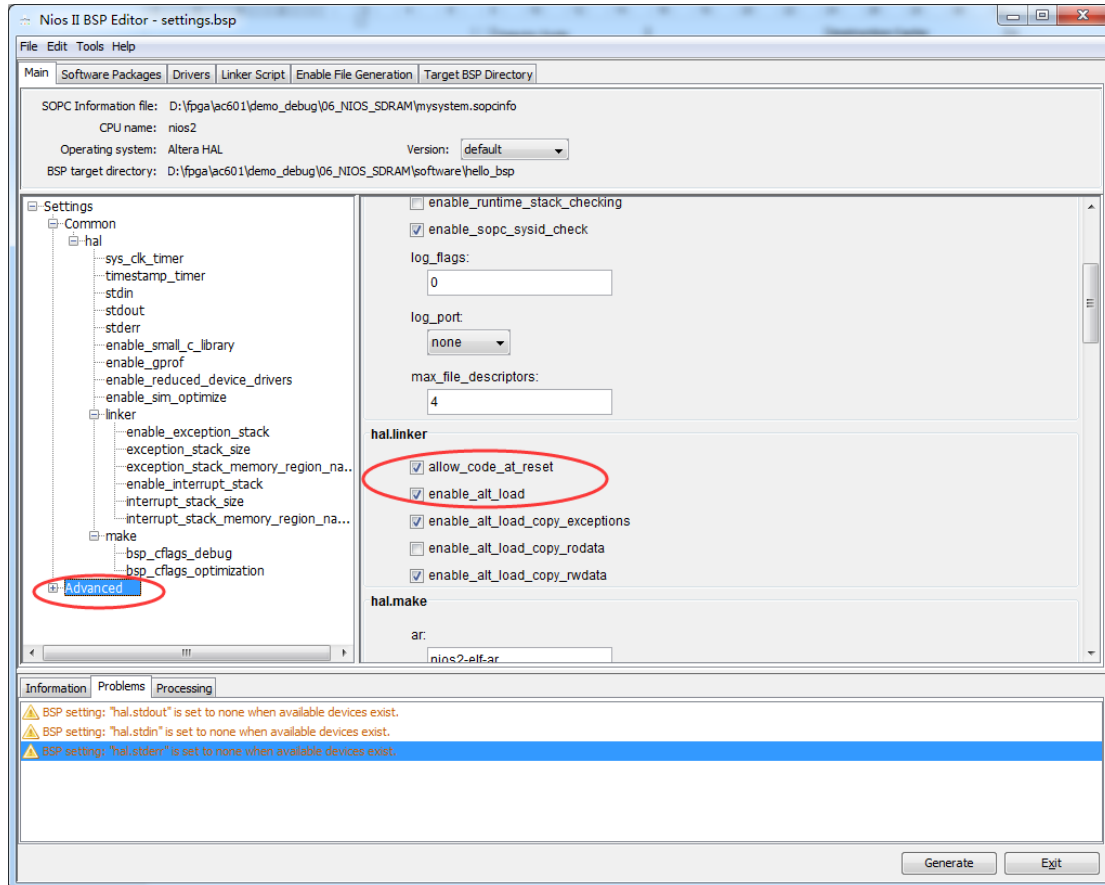


我们选择一个 bsp 工程，打开 bsp editor 页面，在第一个选项卡 main 中，找到下面的 Advanced 选项，然后右侧相关内容中有个 hal.linker 选项，在该选项中有 5 个可选项，其中第一个叫做 allow\_code\_at\_reset。这个选项是什么意思呢，就是允许代码存放在复位向量处，好了我们回头来想一下，上面我们说过，我们可以选择 CPU 的 Reset Vector 为 EPCS，而这里又使能了 allow\_code\_at\_reset 选项，所以程序编译的时候就会将启动程序部分编译在 EPCS 所在的地址段。这就是真真切切的将复位地址设置在了 EPCS 中。此种模式下我们无法在线调试的，如果强行调试一定会出现上述报错。若有人反驳说他的工程此种情况下也能下载成功，那一定是因为你之前已经将 EPCS 中烧写过一次本程序了，所以你的 CPU 能够启动，是因为设定的启动地址 EPCS 中有能够支持 CPU 启动的程序，但能启动不代表其他功能也能正常运行哦。一旦选择该选项，下面的 enable\_alt\_load 也要一并选上，选上之后 CPU 就能够正常从 EPCS 中加载了。部分用户在烧写 SOF 和 ELF 到 EPCS 中进行固化时不成功，原因也与没有勾选这两个选项有关。

那么如果我们希望先在 RAM 中调试怎么办呢，想必大家已经想到了，不使能 allow\_code\_at\_reset 和 enable\_alt\_load 就可以了。对的，当我们希望在 RAM 中调试的时候，将这两个勾选项取消，那么程序编译就会默认将启动代码编译到 RAM 中，然

后就可以从 RAM 中启动并调试啦。

**解决方法：**调试时不勾选 `allow_code_at_reset` 和 `enable_alt_load`。烧写时务必勾选这两个选项。



4、FPGA 逻辑时序不满足，CPU 无法运行在指定的频率下。

注意，一般用户在创建 NIOS II 的系统时候，系统频率都在 100M 左右，而这个频率下，如果不进行时序约束，Quartus II 软件默认布局布线综合出来的结果，是很难运行到这么高的频率的，往往只能跑到 40~80M 左右，因此，创建 NIOS II CPU 系统的时候，一定记得在 Quartus II 工程中添加 SDC 时序约束文件，约束也并不复杂，对输入时钟和 PLL 生成时钟进行下约束就可以了，具体实例可以参见小梅哥 SOPC 公开课《基于 SOPC 的 2.8 寸液晶显示屏应用》1 小时 36 分开始的内容，或者小梅哥 2017 培训班视频《0825\_01-(SOPC)LCD1602 与 NIOS 系统常见问题分析》从第 7 分 30 秒开始的内容。也可以参看《第 5 章 RT-Thread 操作系统在 NIOS II 上的移植》文档中相关说明。

**解决方案：**时序分析+时序约束

5、固件冲突

EPCS FLASH 存储器中已经有对应的 FPGA 固件和可以运行的软件程序，如果 CPU 复位地址设置在 EPCS 中。下载之后 CPU 会默认读取 EPCS 中的程序，实际运行的是 EPCS 中的存储的之前烧写过的 elf 程序，导致调试时新下载的 elf 无法运行。

**解决方法：**调试时不往 EPCS 中固化程序，如果已经固化，可以重新烧写一个不含 elf

程序内容的 jic 文件到 EPCS 中覆盖。推荐烧写正在调试的工程 sof 转换得到的纯 FPGA 硬件部分的 jic 文件。

## CPU 运行一段时间后停止

很多用户在调试或者运行 NIOS II CPU 程序的时候，可能最开始的时候程序能够正确的运行，但是过一会儿后 CPU 却停下来了，例如本来做的一个 LED 流水灯，结果流水了一两分钟后却不再动了。这里是不是又是 NIOS II CPU 运行不稳定的一个佐证呢？实际上，引发这个问题的原因，个人总结主要有两点：

第一点，系统使用了 JTAG UART 作为调试串口打印数据。用户在自己的软件代码中写了有 printf 的代码，例如每分钟打印一次“Hello”，同时让流水灯闪烁。但是用户可能在开发板运行的过程中，突然拔掉或者断开了 USB Blaster 下载器，或者关闭了 nios ii 软件开发工具。这就导致 JTAG UART 与 PC 间的链接断开了。由于 JTAG UART 是基于 JTAG 协议的一个模拟串口，实际上我们使用的 jtag uart 发送数据的时候，是先将数据写在 jtag uart 的 fifo 中，然后 pc 端的 nios ii 开发软件定时通过 jtag 协议读取 fifo 中的数据，以此模拟 jtag uart 的发送的。如果我们切断了 nios ii 开发软件与 jtag uart 之间的连接，或者关闭了 nios ii eds 软件，使之不再去读取 fifo 中的数据，那么 jtag uart 的 fifo 中的数据就会越积越多。当 fifo 满之后，数据就再也写不进去了。而 jtag uart 的软件驱动中，使用的是阻塞的方式发送数据的，即只有当所有的数据都写入 fifo 之后，程序才会执行下一步。可是，现在 fifo 中的数据一直是满的，没有被读走，那么软件就只能一直等在这个地方，等待 fifo 变成非满。然后，就导致整个 CPU 的运行被阻断在了这个地方。所以 NIOS 看起来就像是停止运行了。有的是拔掉下载器之后还能运行一会儿，这是因为 jtag uart 默认是 64 字节的 fifo，刚刚拔掉下载器，fifo 还没满，程序运行过程中间隔的写入数据到 fifo 中，过了一会写 fifo 满了，然后 CPU 就停下来了。

第二点，用户 C 水平不过关（不在少数），写的代码指针使用不合理，出现了指针越界行为，把正常的的数据给损毁了，导致 CPU 的运行数据在运行过程中被损毁，然后就无法继续运行下去了。

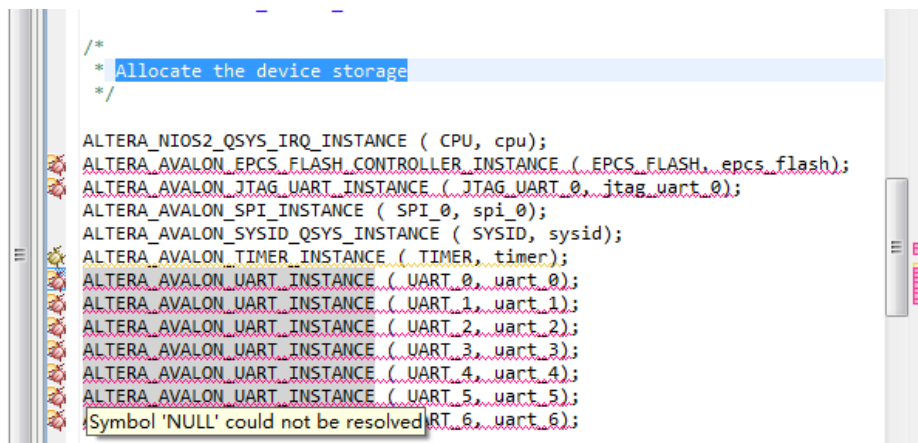
上述原因主要是用户通过反复分析，总是找不出程序问题的时候考虑的几点。至于本身程序就没写对导致的，那就还是乖乖回去深造 C 语言吧。合理利用 debug，帮助查找程序中存在的问题。

## Symbol 'NULL' could not be resolved

近期在评估使用 NIOS II 处理器进行项目的开发，我使用的软件是 Quartus II 13.0 的版本，一路下来，在 Qsys 系统中搭建 NIOS II 片上系统，在 Quartus II 中建立工程文件等等过程，没有太多的问题，这里暂且不表。只是在 NIOS II Software build tools for Eclipse 中进行软件开发时，一个非常让人不解的问题就是：我在工程向导中创建一个 Hello World 的模版工程，然后编译下载运行都没问题。然后关闭 NIOS II Software build tools for Eclipse 软件，再次打开时，结果就冒出一大堆错误，错误描述如下：

在 alt\_sys\_init.c 这个文件中，报错“Symbol 'NULL' could not be resolved”，此报错主要集中在分配设备存储（Allocate the device storage）这一部分，例如，我的系统中报错如下

所示：



```
/*
 * Allocate the device storage
 */

ALTERA_NIOS2_QSYS_IRQ_INSTANCE ( CPU, cpu);
ALTERA_AVALON_EPCS_FLASH_CONTROLLER_INSTANCE ( EPCS_FLASH, epcs_flash);
ALTERA_AVALON_JTAG_UART_INSTANCE ( JTAG_UART_0, jtag_uart_0);
ALTERA_AVALON_SPI_INSTANCE ( SPI_0, spi_0);
ALTERA_AVALON_SYSID_QSYS_INSTANCE ( SYSID, sysid);
ALTERA_AVALON_TIMER_INSTANCE ( TIMER, timer);
ALTERA_AVALON_UART_INSTANCE ( UART_0, uart_0);
ALTERA_AVALON_UART_INSTANCE ( UART_1, uart_1);
ALTERA_AVALON_UART_INSTANCE ( UART_2, uart_2);
ALTERA_AVALON_UART_INSTANCE ( UART_3, uart_3);
ALTERA_AVALON_UART_INSTANCE ( UART_4, uart_4);
ALTERA_AVALON_UART_INSTANCE ( UART_5, uart_5);
ALTERA_AVALON_UART_INSTANCE ( UART_6, uart_6);
```

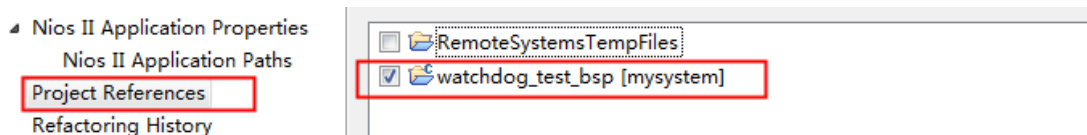
提示我 NULL 未定义。可是 NULL 明明是在 stddef.h 这样一个标准头文件中明确定义了的，怎么还会发生这种情况呢？

另外，由于本系统中使用到了 UART IP，所以系统在自动生成时也提供了对应的驱动，打开一个驱动程序“altera\_avalon\_uart\_write.c”，发现同样有报错的地方，这里主要提示：“Symbol 'O\_NONBLOCK' could not be resolved”，其它文件中也有类似的提示，这就让人很是纳闷了。O\_NONBLOCK 这个宏定义明明是在“sys/\_default\_fcntl.h”这个文件中定义了的，为什么总是说找不到呢？

以下从<第 7 章 给 NIOS II CPU 增加看门狗定时器并使用>摘选出解决方案。

我们选中【watchdog\_test】工程，按下键盘组合键“CTRL + Enter”键打开【Properties】设置界面，选择 Nios II Application Properties 下的 Nios II Application Paths，在右侧的 Application include directories 下，点击 Add 按钮，添加 hardware/inc 到包含路径中。然后在弹出的对话框中点击 Yes，即可将此路径添加为我们的头文件包含路径。如果用户之后自己有其他的头文件路径需要添加，也是按照这种方法进行。

接着我们查看下 Project References 中是否勾选了 watchdog\_test\_bsp 工程，如果没有勾选的话，当工程关闭了重新打开时，工程有可能会报各种无法理解的错误（当一个 workspace 中有多个应用工程时）。这里我们需要确认这个选项被勾选上了。



## 程序运行不正确

有很多用户在进行 NIOS II 开发的时候经常遇见程序下载到目标板后，运行不正确的情况，例如，中断不响应，printf 无法打印，期望的功能无法实现，这样的问题一般分为两类。

店铺：<https://xiaomeige.taobao.com>

技术博客：<http://www.cnblogs.com/xiaomeige/>

官方网站：[www.corecourse.cn](http://www.corecourse.cn)

技术群组：有点多，不列举



a、下载的 sof 与 elf 不属于同一个工程，这个原因其实在我们第 1 个问题的第二种可能原因的时候已经讲到，即由于前后两个 quartus 的工程差别不大，或者说最新编辑的工程是在前一个工程的基础上增删修改部分功能后得到，两个 QSYS 系统中相同外设的地址都是一样，这样，当我们创建 nios ii 软件工程的时候，选择了上一个工程的 sopcinfo 文件，而下载了新的工程的 sof 文件，就会出现，之前的功能是 OK 的，但是新增或者修改的功能总是无法实现。解决问题的方法就不说了，选择正确的 sopcinfo 文件就行。

#### b、NIO S II Cache 的影响

经常有人反映说自己写的代码无法正常工作，然后晒出自己的代码来。经过对用户代码的分析可以知道，他们都是参考了网络上流传的比较系统的两份资料的方式，采用直接地址映射，即用指针的方式来直接操作外设里面的寄存器。这样操作在不带 Cache 的 NIO S II 版本中能够很好的奏效，例如 NIO S II 的 e 版本和 s 版本，但是在 f 版本中，为了增强 NIO S II 处理器的性能，加入了数据 cache 和指令 cache。如果用户使用带 cache 的 CPU，却还是像之前那样，直接使用指针映射寄存器地址的方式，就会出现很多时候，我们希望写入的数据并没有直接写入到外设 IP 的寄存器中，而是写入到了 cache 中，即外设 IP 中并未立即写入我们希望写入的值，也就不会执行相应的操作了。（关于 CACHE 是个啥东西，这里不做专门讲解，有兴趣的请自行查阅相关资料）。

那么为什么明明是要写入到外设 IP 寄存器中的数据，却写入到了 cache 中呢？这是因为，NIO S II e 和 s 版有 31 位的地址线，f 版本有 32 位地址线，但是这第 32 位地址线恰恰就是用来选择 cache 的，当地 32 位地址线为 0 的时候，选择 cache，当第 32 位地址线为 1 的时候，就旁路/屏蔽 cache，也就是说，当我们还是继续使用指针映射的方式操作外设寄存器，假设寄存器的地址为 0x00000001，那么在带有 cache 的系统中，该地址实际是映射到了 cache 中，而真正的寄存器地址应该是 0x10000001，因为要想寻址到实际的寄存器地址，必须屏蔽 cache，即需要地址的最高位为 1。所以，如果我们还是用指针直接操作 0x00000001 这个地址，当然数据不会传入实际的寄存器中，所以无法生效。那么怎么解决呢，个人观点还是使用 Altera 官方提供的 HAL 库，如 io.h 这个文件里，提供了很多读写函数：

```
IORD_32DIRECT(BASE, OFFSET) //从某地址读出 32 位的数据
IORD_16DIRECT(BASE, OFFSET) //从某地址读出 16 位的数据
IORD_8DIRECT(BASE, OFFSET) //从某地址读出 8 位的数据

IOWR_32DIRECT(BASE, OFFSET, DATA) //向某地址写入 32 位的数据
IOWR_16DIRECT(BASE, OFFSET, DATA) //向某地址写入 16 位的数据
IOWR_8DIRECT(BASE, OFFSET, DATA) //向某地址写入 8 位的数据

IORD(BASE, REGNUM) //从某地址按照 CPU 数据位宽（32）读出数据
IOWR(BASE, REGNUM, DATA) //向某地址按照 CPU 数据位宽（32）写入的数据
```

这些函数在使用的时候，会自动屏蔽 CACHE，另外，针对每个特定的 IP，Altera 也都提供了相应的驱动文件，如 PIO 核，提供的驱动头文件名叫“altera\_avalon\_pio\_regs.h”，在这里面定义了对 PIO 外设 IP 进行读写和控制的所有



驱动函数，这些函数也都是自动屏蔽了 CACHE 的。而且，使用这个函数，能够方便的在各种 NIOS II 版本的 CPU 之间移植，而不用担心 CACHE 的问题。所以，个人强烈推荐使用官方库进行 IP 核的使用。如果用户执意要坚持自己的观点，使用指针直接映射，那么请在定义指针的时候，将地址最高位置为 1，例如，PIO\_LED 的地址为 0x00000001，那么用户定义该地址指针时，请用#define PIO\_LED (0x00000001 | 0x80000000)，或者#define PIO\_LED (PIO\_LED\_BASE | 0x80000000)，其中 PIO\_LED\_BASE 在 system.h 头文件中定义。这样再操作就不会有问题了。

c、自定义驱动与 NIOS II BSP 工程提供的 HAL 驱动冲突。

常见现象为外设不受控，如看门狗不受用户程序控制，定时器不受用户程序控制，串口数据发送/接收丢失数据或乱码等。这是因为，当我们的 QSYS 系统中添加了这些 IP 外设后，在 NIOS II EDS 软件中创建模版工程时候，如果选择标准工程，如 Hello World 模版工程，则会自动添加所有外设的驱动程序，并在 alt\_main() 函数中调用 alt\_sys\_init() 函数（位于 bsp 工程下 alt\_sys\_init.c 文件中）中将这此外设初始化，这些初始化就包括了注册外设驱动。因此，当我们在用户程序中再使用直接操作寄存器的方式来控制外设时候，则会出现用户程序和系统已经注册的标准驱动程序相冲突。

例如，对于串口接收，系统驱动已经默认将接受到的数据接收到了其缓存中（系统驱动使用中断方式接收，因此我们看不到直接的过程），可是这些是底层驱动实现的，我们并不知道，当我们再去读串口接收状态寄存器，就没法查到对应的接收成功状态寄存器是否有效，因为接收到数据后该状态已经被系统驱动函数给清零了，所以我们用户程序就出现了无法接收到串口收到的数据的现象。出现数据丢失。当然，这个情况会在接收数据一段时间后消失，用户程序又能读到状态和数据了，这是因为系统驱动函数将读到的数据先暂存在一个 fifo 数组中。读了一段数据后，fifo 中的数据由于没有被用户程序及时取走使用，因此 fifo 满了，驱动程序就不会再去主动接收串口数据，也就不会去及时清除状态，所以我们用户自己读取状态寄存器，就能读到正确的状态了。

那么怎么解决这个问题呢？很简单，两种方式，第一种是我们手动修改 alt\_sys\_init() 函数中的内容，将注册系统驱动函数这部分代码给屏蔽掉，例如默认的，该文件中的内容如下所示：

```
void alt_sys_init( void )
{
    ALTERA_AVALON_UART_INIT ( DEBUG_UART, debug_uart);
    ALTERA_AVALON_UART_INIT ( UART_RS232, uart_rs232);
    ALTERA_AVALON_UART_INIT ( UART_RS485A, uart_rs485a);
    ALTERA_AVALON_UART_INIT ( UART_RS485B, uart_rs485b);
}
```

我们可以看到，该函数将 4 个串口默认都初始化了，这个初始化函数中就有注册驱动的功能。因此如果我们对某个外设不希望使用系统提供的驱动，就可以直接将该外设的初始化函数注释掉。例如，我们不希望 UART\_RS485A 和 UART\_RS485B 使用系统提供的默认驱动，就可以将该函数中 ALTERA\_AVALON\_UART\_INIT

( UART\_RS485A, uart\_rs485a);和 ALTERA\_AVALON\_UART\_INIT  
( UART\_RS485B, uart\_rs485b);两个调用注释掉。修改完成后的该函数如下所示:

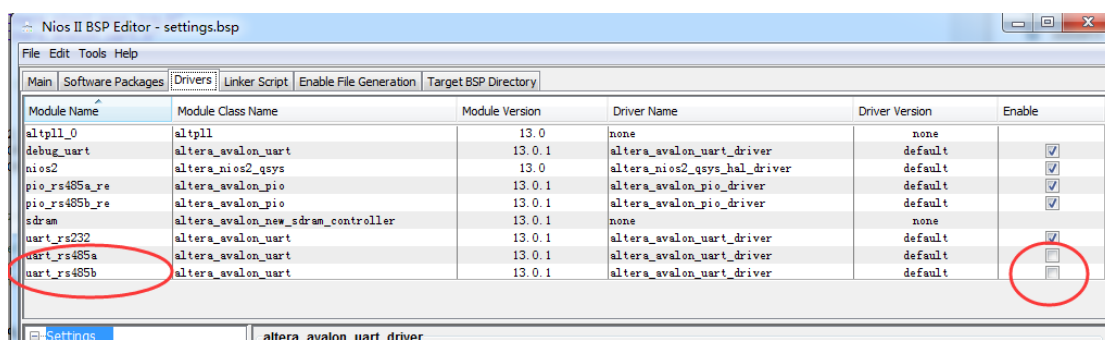
```
void alt_sys_init( void )  
{  
    ALTERA_AVALON_UART_INIT ( DEBUG_UART, debug_uart);  
    ALTERA_AVALON_UART_INIT ( UART_RS232, uart_rs232);  
    //  ALTERA_AVALON_UART_INIT ( UART_RS485A, uart_rs485a);  
    //  ALTERA_AVALON_UART_INIT ( UART_RS485B, uart_rs485b);  
}
```

这样, 这两个外设的驱动由于没有初始化, 因此不会工作, 这个时候我们就可以使用读写寄存器的方式操作该外设了, 如发送一个字节的的数据可以写为:

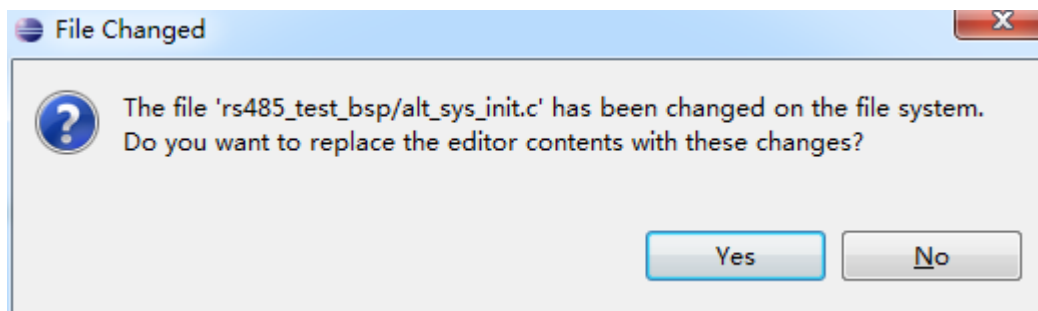
```
//等待发送寄存器可用  
while (!(ALTERA_AVALON_UART_STATUS_TRDY_MSK  
        & IORD_ALTERA_AVALON_UART_STATUS(UART_RS485A_BASE)))  
;  
//写入待发送的数据  
IOWR_ALTERA_AVALON_UART_TXDATA(UART_RS485A_BASE, 0x88);
```

此种方式在用户执行 generate bsp 操作后会失效, 因为我们改掉的代码会被重新覆盖为默认内容。该方法用个成语说叫做扬汤止沸, 无法根除问题, 因此不推荐大家使用。

第二种方式, 属于治根, 那就是在 BSP 工程中取消生成对应设备的驱动。由于 alt\_sys\_init.c 文件中的内容都是根据 bsp 设置中的相关选项对应生成的, 因此, 要想每次重新 generate bsp 后都不包含不希望初始化的设备代码, 可以在 bsp 中设置不使能生成该设备的驱动, 方法为, 在 BSP Editor 中, 切换到 Drivers 选项卡, 针对不希望生成驱动的设备, 在其后面的 Enable 栏中, 去掉勾项即可。



当设置完成后, alt\_sys\_init.c 文件会被自动更新, 如果我们当前本来就是在 alt\_sys\_init.c 文件打开的界面然后去设置 bsp 的, 那么当你设置完成, 回到这个文件, 系统会提示该文件已经被更改, 然后我们选择 yes, 以更新文件内容, 就可以发现, 之前的 RS485A 和 RS485B 两个设备的驱动初始化代码已经不存在了。



更新完成后，alt\_sys\_init 函数中原本对 RS485A 和 RS485B 两个端口的初始化代码已经不见了：

更新前

```
void alt_sys_init( void )
{
    ALTERA_AVALON_UART_INIT ( DEBUG_UART, debug_uart);
    ALTERA_AVALON_UART_INIT ( UART_RS232, uart_rs232);
    ALTERA_AVALON_UART_INIT ( UART_RS485A, uart_rs485a);
    ALTERA_AVALON_UART_INIT ( UART_RS485B, uart_rs485b);
}
```

更新后

```
void alt_sys_init( void )
{
    ALTERA_AVALON_UART_INIT ( DEBUG_UART, debug_uart);
    ALTERA_AVALON_UART_INIT ( UART_RS232, uart_rs232);
}
```

## 拔掉下载器或者关闭 NIOS II EDS 软件，NIOS II 停止运行

### 问题原因

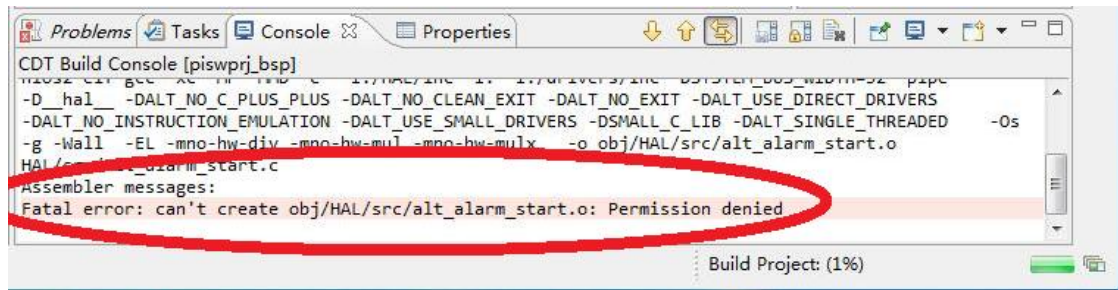
这是因为 NIOS II 系统中使用了 JTAG UART，而在 NIOS II 系统中，JTAG UART 的驱动是阻塞的，而且，该串口是使用 JTAG 模拟的，因此，NIOS II 通过 JTAG UART 发送数据实际是将数据送入了 JTAG 的 FIFO 中，需要电脑端使用软件（如 Eclipse 中的 Console）来读取数据，如果数据不被读走，则 NIOS II 中的程序会一直在这里等待数据被读走，才能发送下一个数据，因此卡在该处无法继续运行，就出现死机的现象。

### 解决建议

实际调试时建议使用 RS232 串口，不要使用 JTAG UART。

## 编译 NIOS 软件工程提示“Permission denied”

在编译一个由其他电脑创建的 NIOS II 软件工程时，提示：“Fatal error: can't create obj/././xxxxxxx.o, Permission denied”，如图所示：



### 问题原因

不同电脑对该文件夹的权限设置不一样，因此当文件被复制过来后，当前用户环境下没有权限对该文件进行编译，因此报错。

### 解决建议

首先关闭 NIOS II IDE 软件，然后将该文件夹取得管理员所有权，再打开软件，编译就不会有问题了。

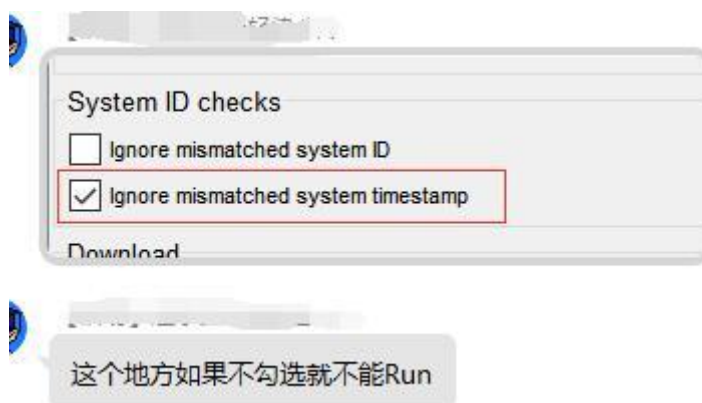
## 常用设置和操作

### systemid 和 system timestamp 功能意义作用介绍

有用户提问，在 qsys 系统中，systemid 加了，下载的时候 systemid 这一项校验也通过了，但是 system timestamp 值校验不通过。请问到底是怎么回事。用户还提供了截图，如下图所示。

```
Processor is already paused
Reading System ID at address 0x010040A0:
    ID value verified
    Timestamp value was not verified: value was not specified
Initializing CPU cache (if present)
```

而且他说，如果此时，在 run 的时候，勾选了忽略不匹配的系统时间戳（ignore mismatched system timestamp）选项，就能成功 run，如果不勾选，就不能 run，而且即使是勾选该选项后 run，run 的结果也有可能异常。



用户大概率会再次怀疑 NIOS II 不靠谱。

先做个总结说明，这并不是 NIOS II 不靠谱的表现，反而恰恰相反，这是 NIOS II CPU 为了降低大家使用过程中犯低级错误的概率，而设置的一套强制验证机制。

这里先对这两项的物理意义做个说明，相信说完大家就都懂问题到底在哪里了。

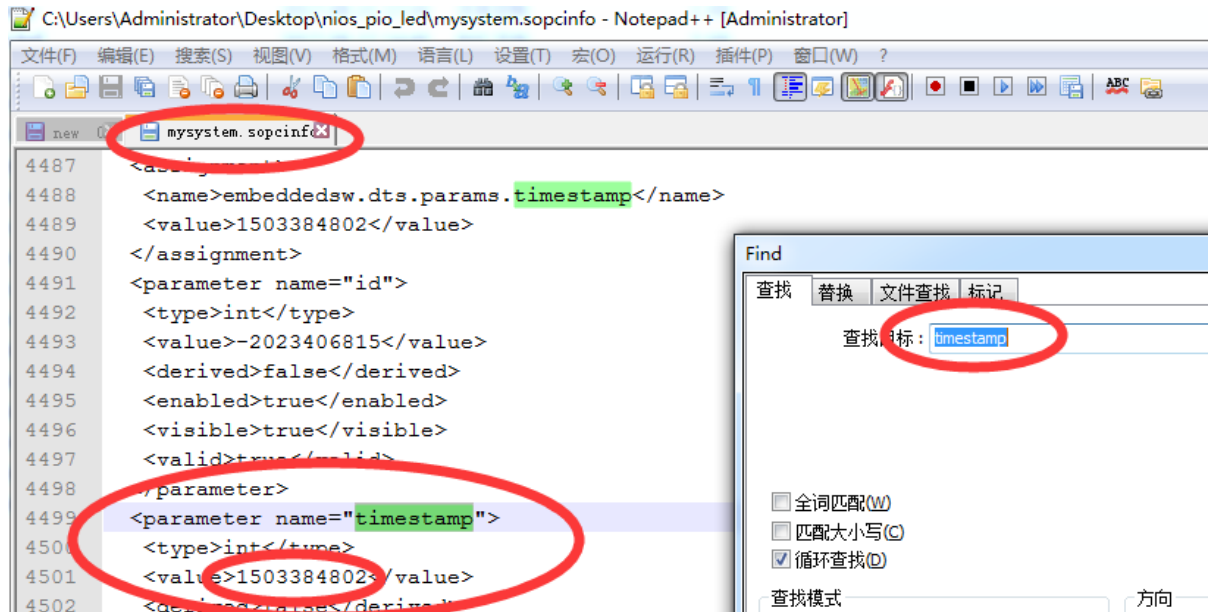
首先来说，用户截图的这个只能证明你当前下载的 sof 里面的 NIOS II 系统，使用的 system id 是你设置的 ID。

举个例子，你在 D 盘根目录下创建了个工程，里面加入了 sysid，设置的值为 0x12345678，然后调试了一段时间之后，你想加功能，所以你把这个工程拷贝到了 E 盘工程根目录下，然后在 E 盘下基于这个工程加入了一个 UART 串口外设再编译。这个时候，两个系统的 sysid 都是 0x12345678。此时，无论你下载 D 盘的还是 E 盘的 sof 到 fpga 芯片，在这个界面去调试的时候读到的 sysid 都是 0x12345678，那么你认为这两个工程一样吗？你下载 D 盘工程的 sof，能调试你针对 E 盘的工程写的 uart 的程序吗，所以，单就 sysid 校验正确，并不能确定这个工程就是对的，只要两个工程的 sysid 设置的相同，这一

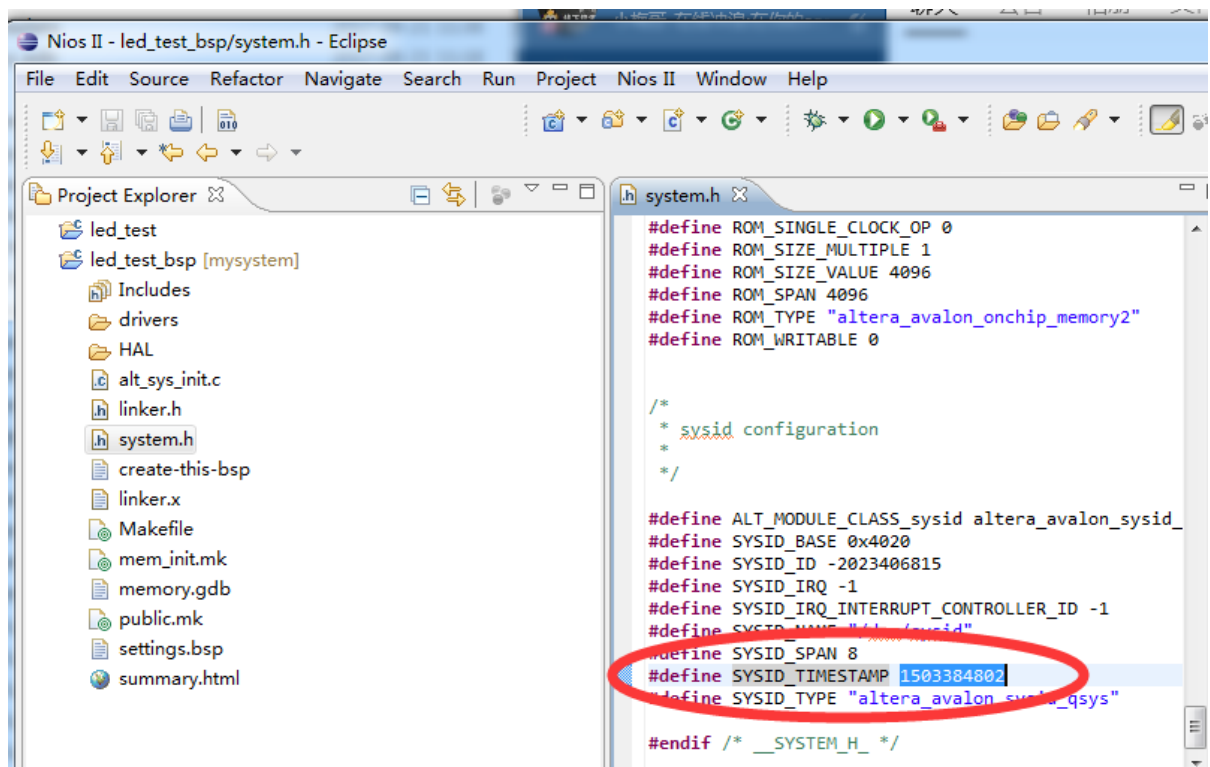


项就能通过。

也因如此，NISO 还有第二项机制来确保硬件（sof）和软件（elf）的匹配，这个就是 system timestamp。在你的 sopcinfo 文件中，有一个 timestamp 的参数，如下图所示。



这个参数，每次 Quartus 工程编译一次，注意，是 Quartus 工程编译一次，而不是 qsys 重新生成一次，即使两次编译你没有改任何 quartus 中任何一个字符，两次编译，这个 timestamp 的值都会变化一次，都是不一样的。然后，在 nios ii eds 中，每次你重新编译了 quartus 工程，都会提示你重新 generate bsp，这个重新 generate bsp，就会读取你这个新的 sopcinfo 文件，并在 system.h 文件中记录这个新的 timestamp 参数，如下图所示：



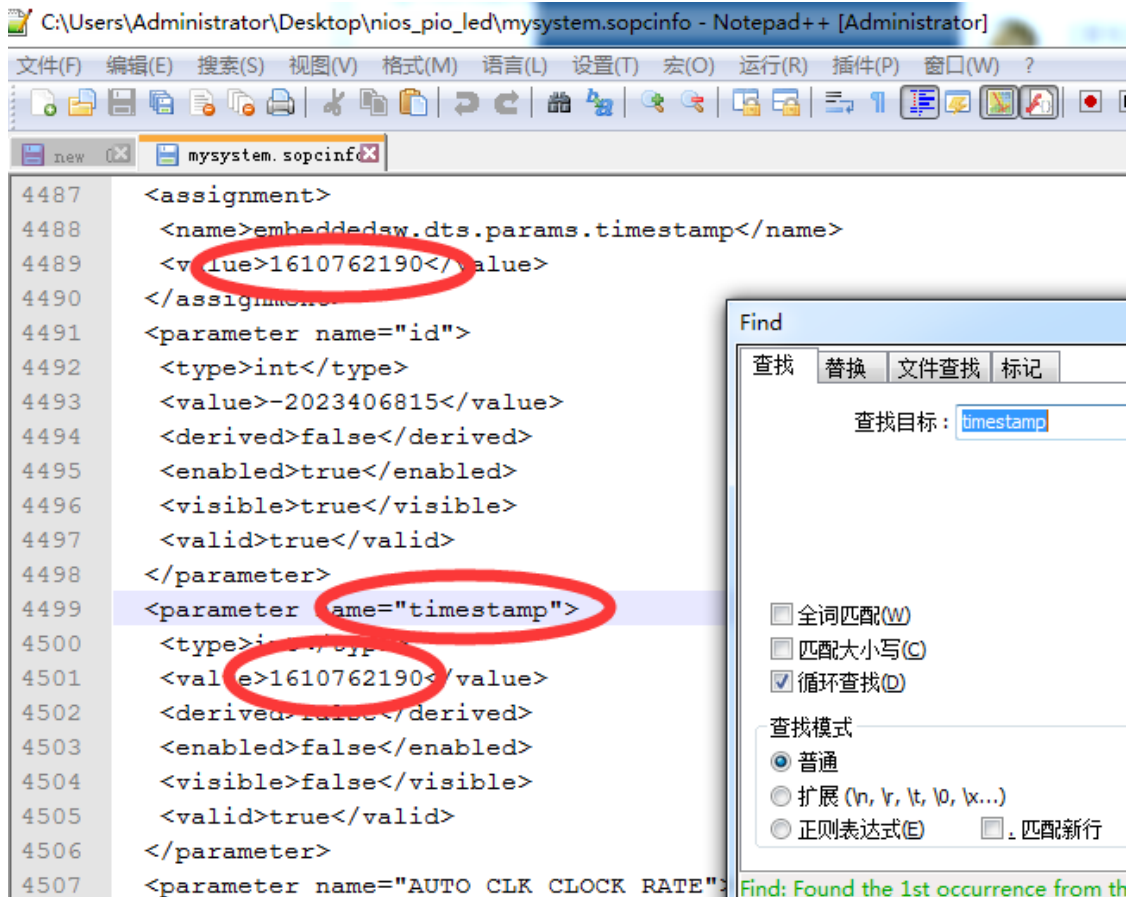


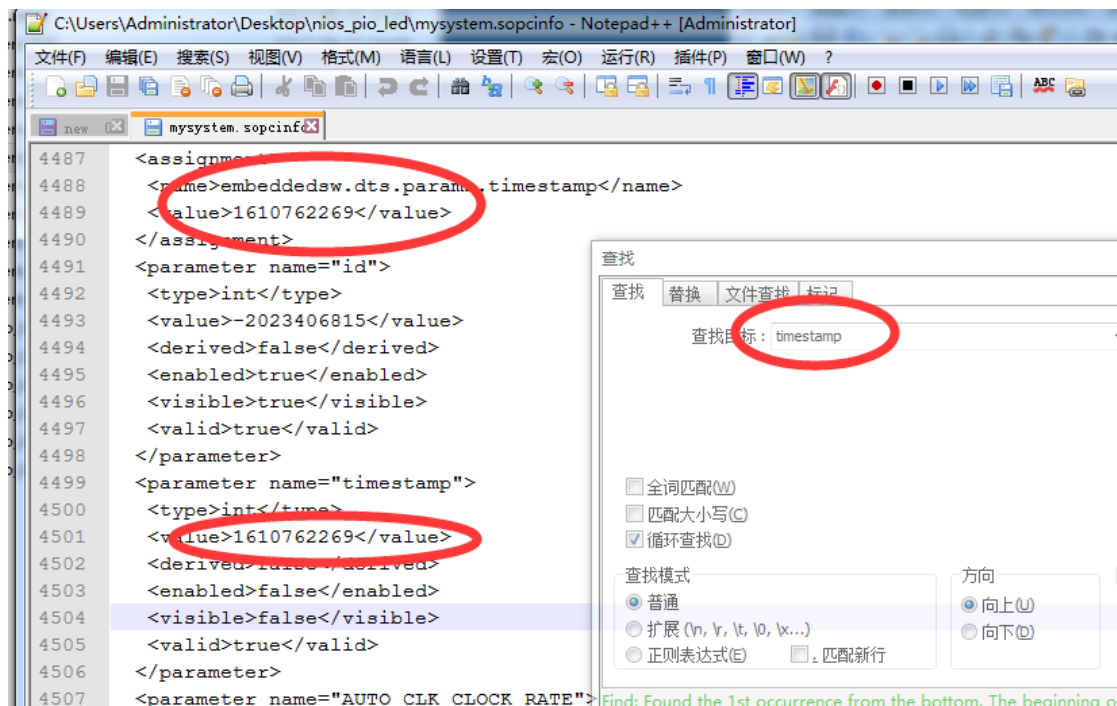
所以说，如果你的系统在校验的时候。timestamp 对不上，就一定百分之一亿的确  
定，你下载的 sof 和你当前的 nios ii 用的 bsp 对不上。

所以，以后，加上 sysid 是个好习惯，通过 sysid 和 timestamp 两个值来帮你辅助确认你  
用的两套内容到底是不是绝对对应的。说来惭愧，我本人一直没加 systemid 的这个习惯，  
但是因为我经验足够丰富，不会犯这种错误了，所以问题不大。新手建议还是都加上的  
好。

补充说明下：前文的 system timestamp 是系统时间戳的意思，不是在 qsys 中配置个  
timer 定时器来作为程序时间戳的意思。配置 timer 为 timestamp 功能，是用来测量 C 程序  
的运行时间用的。而这个 system timestamp 是用来确保整个工程的软硬件匹配用的。

以下再附图两张，证明下我前面说的 quartus 每次编译都会更改那个 system timestamp  
的值的情况。以下两个工程，两次编译时间间隔不超过 5 秒钟，可以看到，两次的 system  
timestamp 值确实是有了变化。





## 解决 NIOS II 工程移动在磁盘上位置后 project 无法编译问题

说明：本文档于 2017 年 3 月 4 日由小梅哥更新部分内容，主要是增加了讲解以 Quartus II13.0 为代表的经典版本和以 15.1 为代表的更新版本之间，解决问题的一些小的差异。

如果用户只是想快速解决问题，不想分析产生问题的原因并和我一起探寻解决问题的思路，可以直接跳到 6.4 节解决方案步骤总结。

针对目录改变时，Nios II project 无法编译的问题，网上有多种解决方法，不过都操作相对繁琐，这里，小梅哥进过探索，针对 11.0 及以后的版本，找到了一种简单可靠的解决办法，整个过程只需要简单的四步操作即可搞定，分别为：切换工作空间（workspace），移除旧版工程，修改 bsp 文件，重新导入（import）工程。

### 6.1 更改 NIOS II Project 目录原因

引用网上木易前辈的话，“我们常会有各种理由会改变原来 project 的目录名称或目录位置”例如：

1. 为了管理方便，可能将原来在 d:\project\的所有 project 移到 e:\project\下
2. 同事将 project 整个目录压缩给我，因为我并不知道该 project 放在同事计算机什么工作目录下，所以我将压缩文件解压缩到我自己的工作目录下

3. 从网络上下载整包范例程序的压缩文件后，因为我并不知道原本范例程序所存放的目录，所以我将压缩文件解压缩到我自己的工作目录下
4. 从书上光盘复制范例程序到硬盘，因为我并不知道原本范例程序所存放的目录，所以我将范例程序复制到我自己的工作目录下
5. 新的 project 与旧的 project 类似，想从旧的 project 去做修改即可，开了一个新的目录，将旧的 project 所有档案复制到新的目录下
6. 为了管理方便，想改变原本 project 的目录名称

## 6.2 更改 NIOS II Project 目录引发的问题

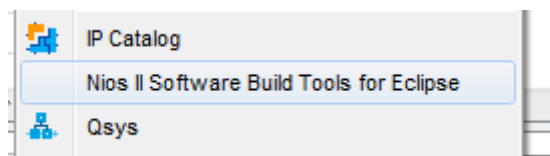
Quartus II 的工程在更改了路径后是不会有问题的，我们可以直接编译更改下载，而在 NIOS II EDS 中的工程却没那么简单，Nios II EDS 是用 Eclipse 去改的，用的是 Eclipse 的 workspace 概念，很类似 Visual Studio 的\*.sln 概念，但又不完全一样。Eclipse 允许你在一个 workspace 下，去管理多个 project，workspace 记住的是 project 的绝对路径，所以当你的 Nios II project 目录名称改变，或者目录位置改变，该 workspace 自然就找不到了。

或者，如果你的新工程是从老工程复制过来的，那么一切表面看起来不会出现任何问题，所有的软件工程仍然可以“正常打开”，我们依旧可以编译，重新生产 bsp 文件，下载。然而，这种方式存在更大的风险，因为这样，你修改的还是原来路径下的文件，因此，这样就有极大的风险使得你在希望更改新的软件工程的时候，把原本的工程给改了。很多朋友表示 NIOS II 开发中存在各种各样的问题，例如无法下载 elf 文件，下载后软件不能执行或者执行报错。经过这段时间的辅导答疑，发现他们出问题大部分也都是这个原因。

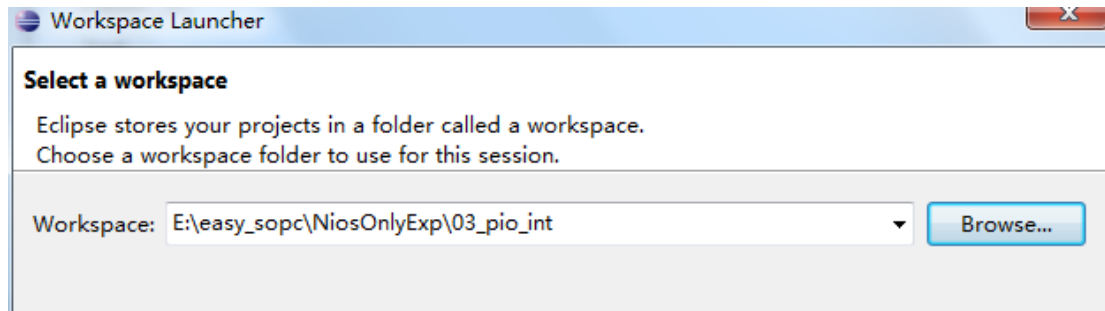
## 6.3 解决方案详解

因此，为了让各位 NIOS II 用户快速上手，避免遇到这个问题而耽误太多的时间，小梅哥这里介绍一种最简单粗暴的解决办法。整个过程只需要简单的四步操作即可搞定，分别为：切换工作空间（workspace），移除旧版工程，修改 bsp 文件，重新导入（import）工程。

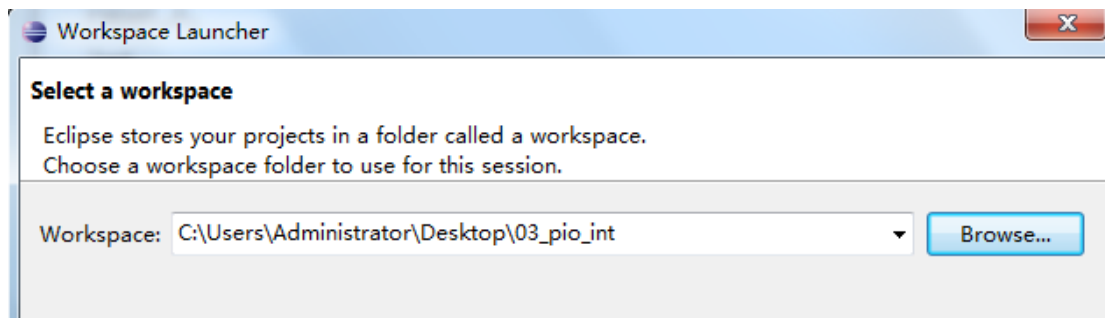
这里，我将我电脑中 E:\easy\_socp\NiosOnlyExp\03\_pio\_int 这个文件夹拷贝到桌面（C:\Users\Administrator\Desktop\03\_pio\_int）上，以符合更改路径这一前提，然后，打开其中的 Quartus II 工程“CoreCourse\_GHRD.qpf”，工程打开后，选择 tools->Nios II Software Build Tools For Eclipse。



然后在弹出的工作空间选择对话框中，可以看到，工作空间还是上次的工作空间路径。

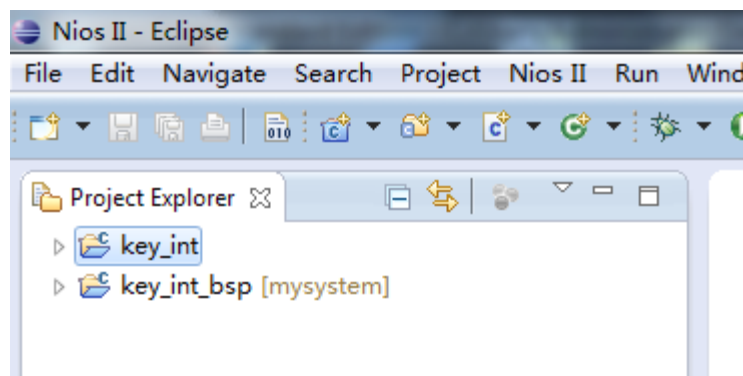


这里，我们将工作空间切换到 C:\Users\Administrator\Desktop\03\_pio\_int，如下图所示。

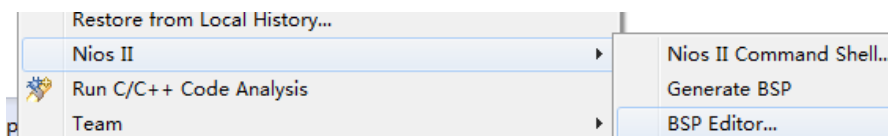


然后选择 OK 就会打开工程，打开后我们可以看到，软件自动加载了一个软件工程和一个 bsp 工程，该工程名字与复制前的工程

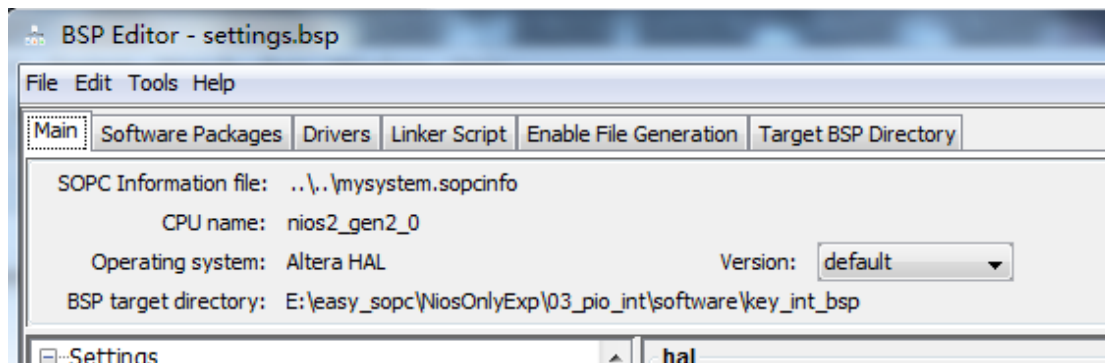
(E:\easy\_socp\NiosOnlyExp\03\_pio\_int\software) 中的软件工程名字一致（注意，如果用户磁盘上旧位置不存在该工程，例如该工程是从其他电脑拷贝过来的，工程将显示蓝色，无法打开，这个虽然会影响我们后续一步一步分析问题原因，但是不影响我们解决问题，如果各位自己电脑上无法打开，可以找一个原本存在的工程测试，或者跳过分析问题步骤，直接看解决方案总结）。



这时候，我们鼠标右键选中 key\_int\_bsp，选择 NIOS II -> BSP Editor

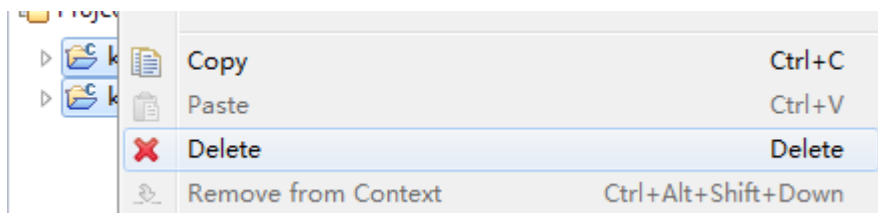


在弹出的 BSP 设置界面中我们可以看到，BSP target direction 还是 E:\easy\_socp\NiosOnlyExp\03\_pio\_int\software\key\_int\_bsp，即复制前的路径。



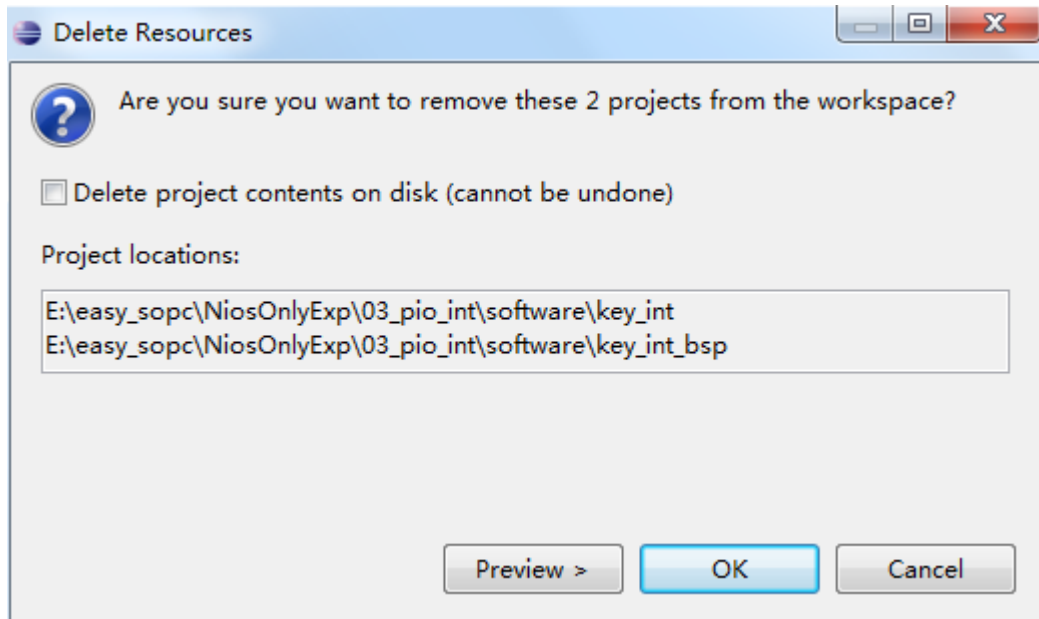
因此可知，如果此时我们在当前工程的 Qsys 中更改了 NIOS II 系统的架构或者增删了东西，重新生产 Qsys 文件，然后我们回到 Eclipse 中选择 generate BSP 时，软件会根据新的 Qsys 信息重新生产 bsp 文件，而这个 bsp 文件还是保存在之前的 E:\easy\_soc\NiosOnlyExp\03\_pio\_int\software\key\_int\_bsp 中，因此就导致原本的复制之前的工程内容被更改，即本来我们是想把整个工程复制到另一个地方进行单独修改的，然而却实质上把原路径下的工程文件给更改了，到最后导致原版和复制后的工程都被改变。

为了解决这个问题，接下来我们将新打开的工作空间中已经存在的两个工程移除。选中已经存在的工程和工程对应的 bsp，右键选择 Delete：

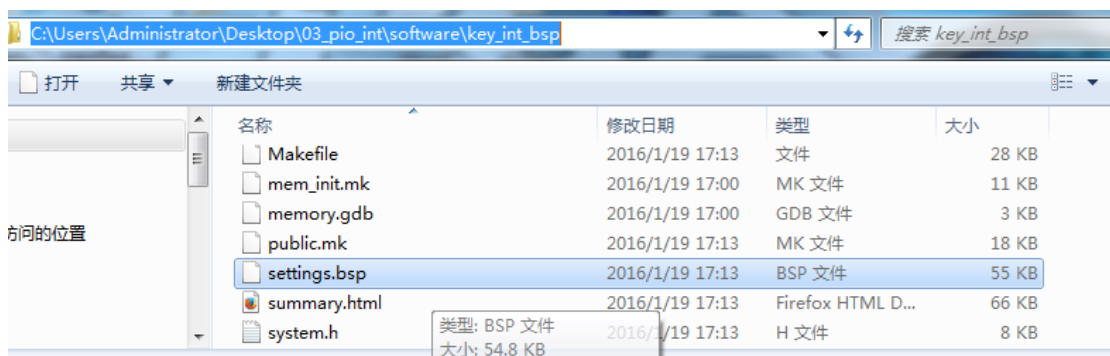


在弹出的对话框中，点击 OK。不过大家一定要注意的是，千万不要勾选上面的那个“Delete project contents on disk (cannot be undone)”，因为这个是删除软件工程的源文件，如果选择了这个，那么在删除时就会将原本没有复制之前的路径下的软件源工程给删除掉，那么原版工程就被彻底破坏掉了，我们所需要做的，只是把这两个工程从工作空间中移除，而不是将原工程删掉，这点大家要切记。



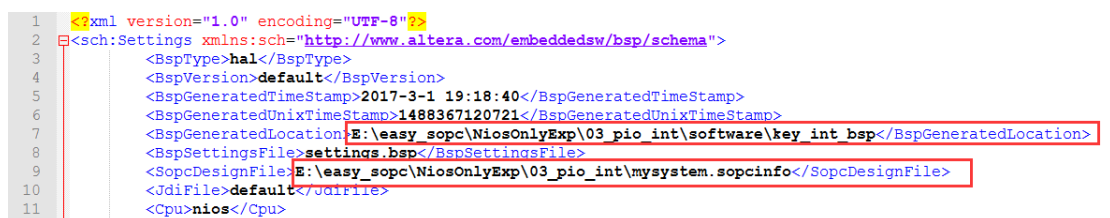


移除了工程之后，我们回到新复制的工程下的 key\_int\_bsp 文件夹下（C:\Users\Administrator\Desktop\03\_pio\_int\software\key\_int\_bsp），找到 settings.bsp 文件，使用任意一个文本编辑器打开：



注意，不同版本的 Quartus II 软件该文件稍有差别。对于 Quartus II 13.0 版本，BSP 文件中两个位置记录了工程绝对路径，而对于 15.1 或以上，则只有一个位置记录绝对路径，另一个位置已经改进为相对路径了（相对路径是相对当前工程，因此无需修改）（其他版本我暂时未检验）。

首先我们看一个 13.0 版本的工程



该文件的 7 和 9 两行都是使用的绝对位置记录的，因此在 Quartus II 13.0 版本的软件中需要修改这两个位置都为新的路径，修改后如下图所示：



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <sch:Settings xmlns:sch="http://www.altera.com/embeddedsw/bsp/schema">
3   <BspType>hal</BspType>
4   <BspVersion>default</BspVersion>
5   <BspGeneratedTimeStamp>2017-3-1 19:18:40</BspGeneratedTimeStamp>
6   <BspGeneratedUnixTimeStamp>1489367120721</BspGeneratedUnixTimeStamp>
7   <BspGeneratedLocation>C:\Users\Administrator\Desktop\03_pio_int\software\key_int_bsp</BspGeneratedLocation>
8   <BspSettingsFile>settings.bsp</BspSettingsFile>
9   <SopcDesignFile>C:\Users\Administrator\Desktop\03_pio_int\mysystem.sopcinfo</SopcDesignFile>
10  <JdiFile>default</JdiFile>
11  <Cpu>nios</Cpu>
12  <SchemaVersion>1.9</SchemaVersion>
13 </Setting>
```

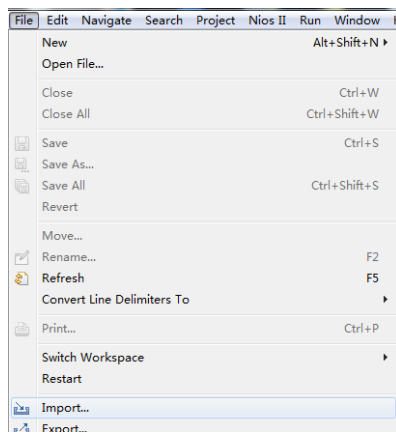
而对于 15.1 版本的工程，打开文件后我们可以看到，该文件下的第 7 行左右，BspGeneratedLocation 指定的还是复制前的路径，第 9 行已经改进为采用相对路径了：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <sch:Settings xmlns:sch="http://www.altera.com/embeddedsw/bsp/schema">
3   <BspType>hal</BspType>
4   <BspVersion>default</BspVersion>
5   <BspGeneratedTimeStamp>2016-1-19 17:13:04</BspGeneratedTimeStamp>
6   <BspGeneratedUnixTimeStamp>1453194784510</BspGeneratedUnixTimeStamp>
7   <BspGeneratedLocation>E:\easy_socp\NiosOnlyExp\03_pio_int\software\key_int_bsp</BspGeneratedLocation>
8   <BspSettingsFile>settings.bsp</BspSettingsFile>
9   <SopcDesignFile>..\..\mysystem.sopcinfo</SopcDesignFile>
10  <JdiFile>default</JdiFile>
11  <Cpu>nios2_gen2_0</Cpu>
12  <SchemaVersion>1.9</SchemaVersion>
```

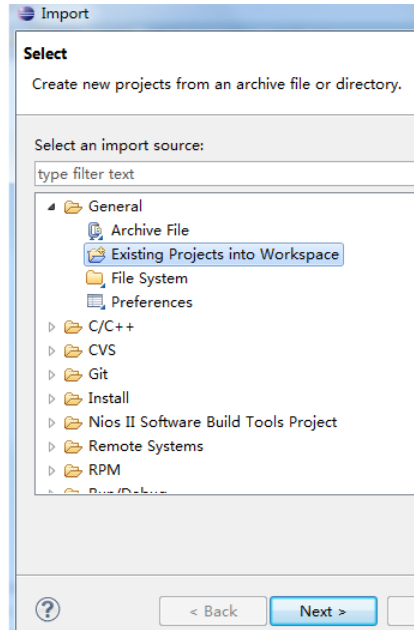
于是，这里只需将第 7 行这个地方更改为我们复制后的新路径 C:\Users\Administrator\Desktop\03\_pio\_int\software\key\_int\_bsp，第 9 行不变即可更改后的文件内容如下所示：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <sch:Settings xmlns:sch="http://www.altera.com/embeddedsw/bsp/schema">
3   <BspType>hal</BspType>
4   <BspVersion>default</BspVersion>
5   <BspGeneratedTimeStamp>2016-1-19 17:13:04</BspGeneratedTimeStamp>
6   <BspGeneratedUnixTimeStamp>1453194784510</BspGeneratedUnixTimeStamp>
7   <BspGeneratedLocation>C:\Users\Administrator\Desktop\03_pio_int\software\key_int_bsp</BspGeneratedLocation>
8   <BspSettingsFile>settings.bsp</BspSettingsFile>
9   <SopcDesignFile>..\..\mysystem.sopcinfo</SopcDesignFile>
10  <JdiFile>default</JdiFile>
11  <Cpu>nios2_gen2_0</Cpu>
```

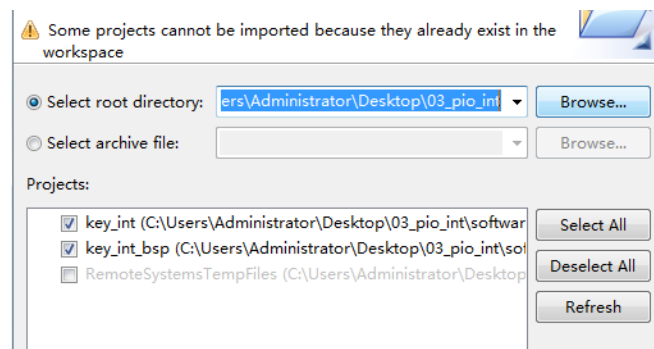
然后保存文件，回到 NIOS II Eclipse 中，选择 File -> Import



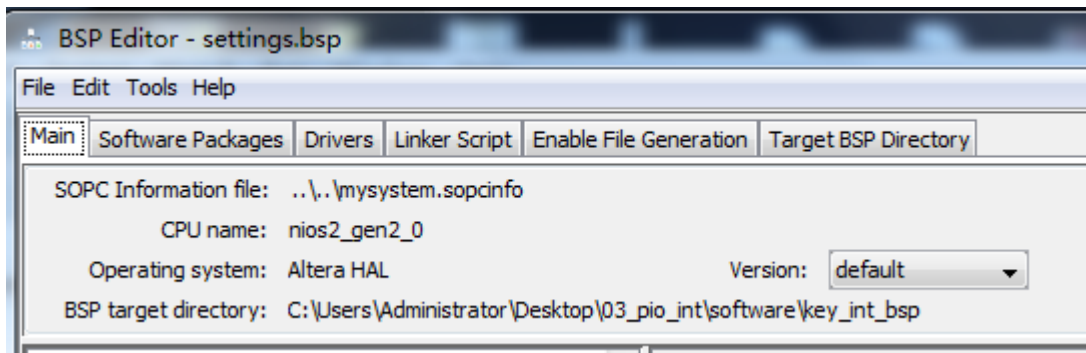
在弹出的窗口中，选择 General 下的 Existing Projects into Workspace:



在弹出的窗口中，选择 Select root directory，点击 Browse，定位到 Workspace 目录（这里也就是 Quartus II 工程目录）C:\Users\Administrator\Desktop\03\_pio\_int。



可以看到，软件会自动找到该目录下存在的软件工程，然后点击 finish 即可将工程导入进来。导入进来后，再次进入 BSP Editor，发现 BSP target direction 就已经是现在的新路径了。



为了区分开这样更改后是否还会对未复制前的工程造成影响，这里我将未复制前的工程剪切到了另一个地方放着，这样如果在重新编译的时候还是要对未复制前的工程进行操作，那么因为原工程已经不在，就会报警告或则报错。而事实上，当我再次编译时并没

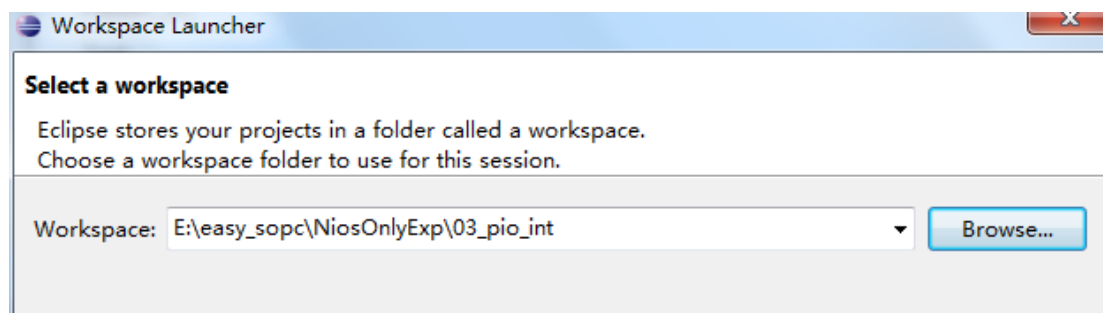
有报任何警告和错误。修改 C 代码后，重新编译生产 elf 文件，下载到芯航线 FPGA 开发板上，也能够正常的运行。打开原版工程（已经将原版工程拷贝回原路径了），还是之前的内容，编译下载也没问题。因此该问题得以完美解决。

## 6.4 解决方案步骤总结

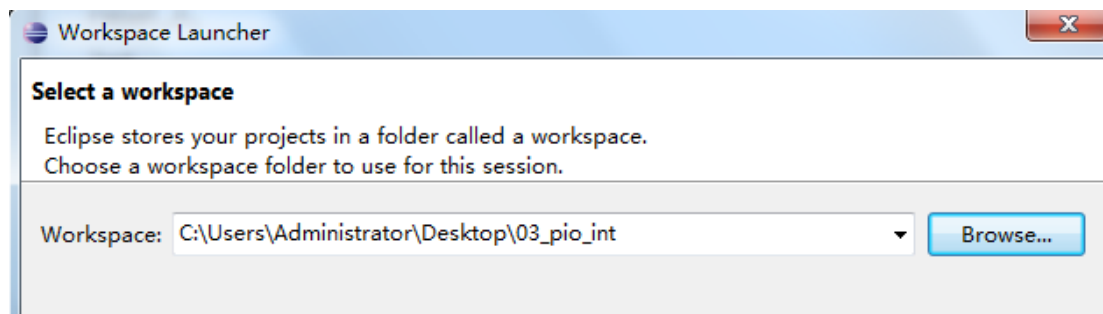
好了，说了这么多，各种推理和验证，导致真正有用的操作步骤被弱化了，不容易区分，这里，我再总结下：

### 6.4.1 切换工作空间（workspace）

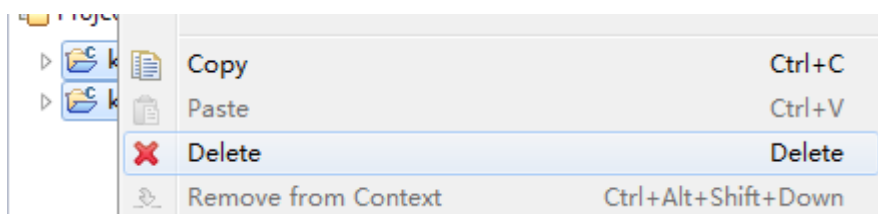
切换工作空间前：

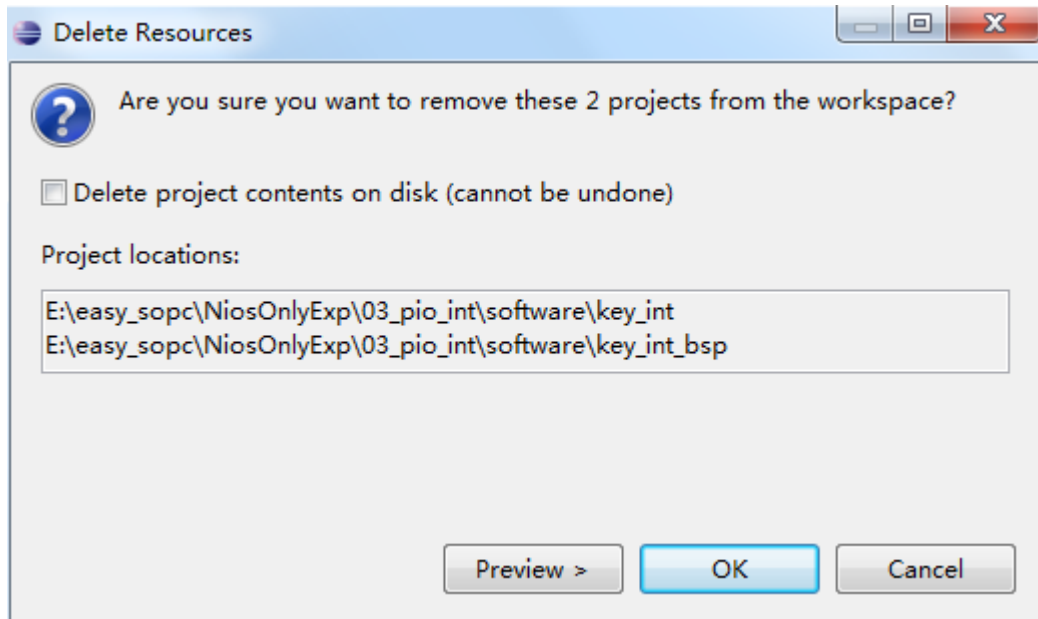


切换工作空间后：



### 6.4.2 移除旧版工程



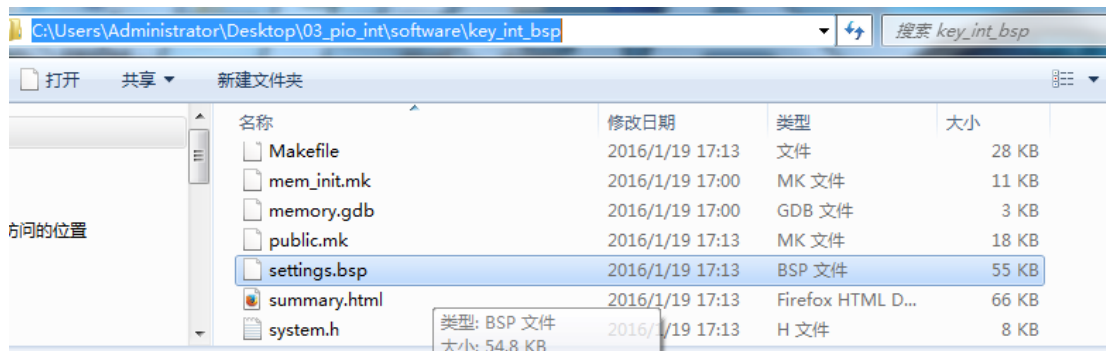


注意：千万不要勾选上面的那个“Delete project contents on disk（cannot be undone）”，

### 6.4.3 修改 bsp 文件

在新复制的工程下的 key\_int\_bsp 文件夹下

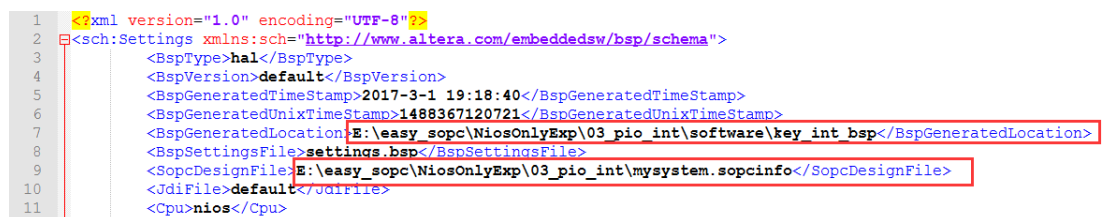
(C:\Users\Administrator\Desktop\03\_pio\_int\software\key\_int\_bsp)，找到 settings.bsp 文件，使用任意一个文本编辑器打开：



第 7 行左右，将 BspGeneratedLocation 指定路径由原版路径改为更改后的新路径：

## 13.0 版本

更改前：



更改后

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <sch:Settings xmlns:sch="http://www.altera.com/embeddedsw/bsp/schema">
3   <BspType>hal</BspType>
4   <BspVersion>default</BspVersion>
5   <BspGeneratedTimeStamp>2017-3-1 19:18:40</BspGeneratedTimeStamp>
6   <BspGeneratedUnixTimeStamp>1489367120721</BspGeneratedUnixTimeStamp>
7   <BspGeneratedLocation>C:\Users\Administrator\Desktop\03_pio_int\software\key_int_bsp</BspGeneratedLocation>
8   <BspSettingsFile>settings.bsp</BspSettingsFile>
9   <SopcDesignFile>C:\Users\Administrator\Desktop\03_pio_int\mysystem.sopcinfo</SopcDesignFile>
10  <JdiFile>default</JdiFile>
11  <Cpu>nios</Cpu>
12  <SchemaVersion>1.9</SchemaVersion>
13  </Setting>
```

## 15.1 及以上版本

更改前

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <sch:Settings xmlns:sch="http://www.altera.com/embeddedsw/bsp/schema">
3   <BspType>hal</BspType>
4   <BspVersion>default</BspVersion>
5   <BspGeneratedTimeStamp>2016-1-19 17:13:04</BspGeneratedTimeStamp>
6   <BspGeneratedUnixTimeStamp>1453194784510</BspGeneratedUnixTimeStamp>
7   <BspGeneratedLocation>E:\easy_socp\NiosOnlyExp\03_pio_int\software\key_int_bsp</BspGeneratedLocation>
8   <BspSettingsFile>settings.bsp</BspSettingsFile>
9   <SopcDesignFile>...\mysystem.sopcinfo</SopcDesignFile>
10  <JdiFile>default</JdiFile>
11  <Cpu>nios2_gen2_0</Cpu>
12  <SchemaVersion>1.9</SchemaVersion>
```

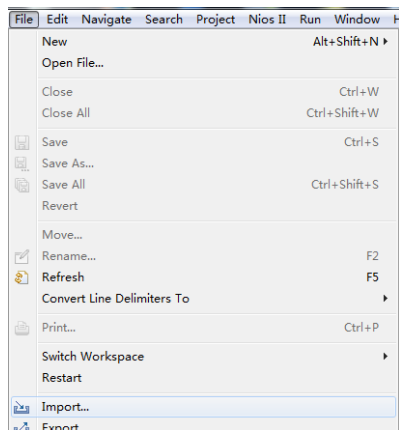
更改后

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <sch:Settings xmlns:sch="http://www.altera.com/embeddedsw/bsp/schema">
3   <BspType>hal</BspType>
4   <BspVersion>default</BspVersion>
5   <BspGeneratedTimeStamp>2016-1-19 17:13:04</BspGeneratedTimeStamp>
6   <BspGeneratedUnixTimeStamp>1453194784510</BspGeneratedUnixTimeStamp>
7   <BspGeneratedLocation>C:\Users\Administrator\Desktop\03_pio_int\software\key_int_bsp</BspGeneratedLocation>
8   <BspSettingsFile>settings.bsp</BspSettingsFile>
9   <SopcDesignFile>...\mysystem.sopcinfo</SopcDesignFile>
10  <JdiFile>default</JdiFile>
11  <Cpu>nios2_gen2_0</Cpu>
```

然后保存文件。

### 6.4.4 重新导入 (import) 工程

NIOS II Eclipse 中, 选择 File -&gt; Import

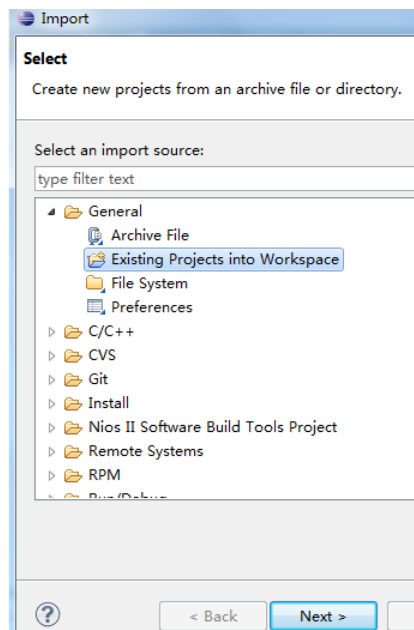


在弹出的窗口中, 选择 General 下的 Existing Projects into Workspace:

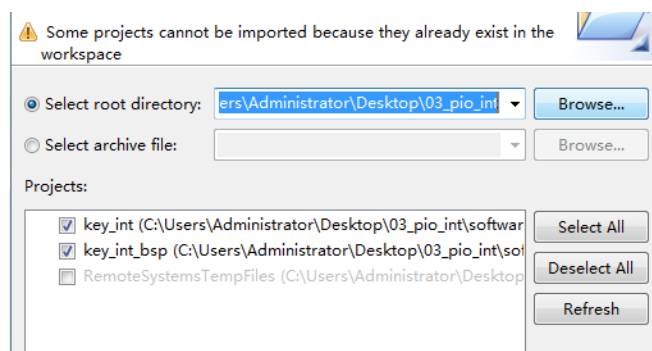
店铺: <https://xiaomeige.taobao.com>技术博客: <http://www.cnblogs.com/xiaomeige/>官方网站: [www.corecourse.cn](http://www.corecourse.cn)

技术群组: 有点多, 不列举





在弹出的窗口中，选择 Select root directory，点击 Browse，定位到 Workspace 目录（这里也就是 Quartus II 工程目录）C:\Users\Administrator\Desktop\03\_pio\_int



点击 finish 即可将工程导入进来。

接下来就可以放心的更改 Qsys 系统和软件工程啦。

## sof 与 NIOS II 的 elf 固件合并 jic 得到文件

### 7.1 为什么需要将 Sof 与 elf 合并得到 jic 文件

我们在学习和调试 NIOS II 工程的时候，一般都是先使用 Quartus II 软件中提供的 Quartus Programmer 来烧写 FPGA 配置文件（SOF），然后 NIOS II EDS 中提供的 Flash Programmer 工具来进行烧写 NIOS II 的。这对于开发者来说，并没有什么不便，反而因为这种方式的灵活，为开发带了的很大的便利。然而，当我们的产品已经设计完成并量产的时候，就需要将固件烧写到产品中。生产线上进行烧录时，总希望能够用最简单的方式实现。试想，如果生产线上在进行烧写时，还需要几个工具换来换去，等待很久，效率自然就下去了。因此这种 Quartus Programmer+Flash Programmer 的方式并不适合生产。

小梅哥在最近的工作中也遇到了这样的问题。我们新设计的一批开发板，在工厂生产完毕后，都要进行出厂测试。然而 SMT 厂家却并不熟悉我们的这种 Quartus Programmer+Flash Programmer 烧写方式。再说了，要使用这种方式还得安装 Quartus Programmer 和 NIOS II EDS 软件。厂家表示使用这种方式对他们来说有一定难度，而且效率也不高。所以我就根据 Altera 官方网站上的一个帖子，进行了转换，将 SOF 文件和 NIOS II 的 elf 固件合并并生成了一个 jic 文件，这样，厂家就只需要使用 Quartus Programmer 来烧写这个 jic 文件就能实现同时烧写 FPGA 配置文件和 NIOS II 固件的功能了，简化步骤，节省时间。

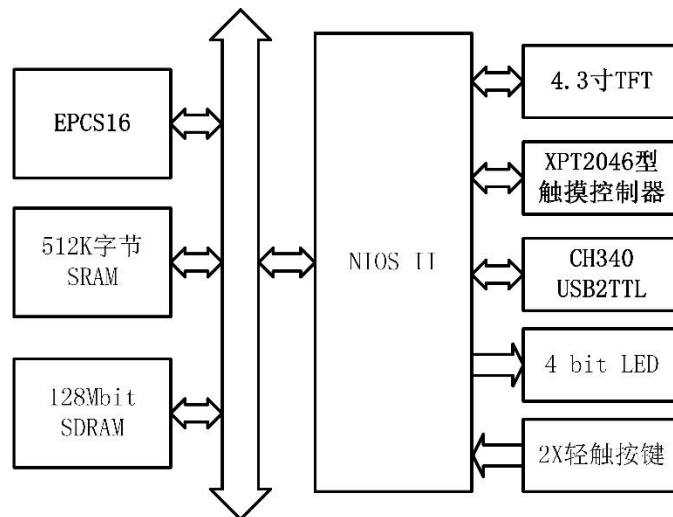
从 SOF 文件和 ELF 文件得到 JIC 文件的原帖地址如下：

[https://www.altera.com.cn/support/support-resources/knowledge-base/solutions/rd10132010\\_126.html](https://www.altera.com.cn/support/support-resources/knowledge-base/solutions/rd10132010_126.html)

## 7.2 本章示例介绍

因为有经验不足的朋友反映在看了这个教程后还是不知道怎么操作，总是不成功，因此这里小梅哥使用我们芯航线 FPGA 的开发板，一步一步演示这个实现过程，将整个过程具体化。

先说明下我这个设计工程的结构：



- EPCS16：用来存储 FPGA 配置文件和 NIOS 的固件，本例中最终转换得到的 JIC 文件也是烧写到该器件中。
- 512K 字节 SRAM：作为 NIOS II 运存或者 4.3 寸 TFT 显存，这里作为 TFT 显存。（PS：使用 SRAM 作为运存，相较于使用 SDRAM 作为运存，NIOS II 的性能会有较大的提升。）
- 128Mbit SDRAM：作为 NIOS II 运存或者 4.3 寸 TFT 显存，这里作为 NIOS II 运存。以运行较为复杂的程序或者 GUI。

- 4.3 寸 TFT：用来显示文字/图片等内容。
- XPT2046 触摸控制器：使用 SPI 接口，用来得到触摸屏信息，实现人机交互
- CH340 USB2TTL：将 UART 协议与 USB 协议进行互相转换。以实现调试的功能。
- 4bit LED：指示程序运行状态。
- 2X 轻触按键：输入控制信息

介绍完了这个系统，接下来就可以介绍整个转换过程了：

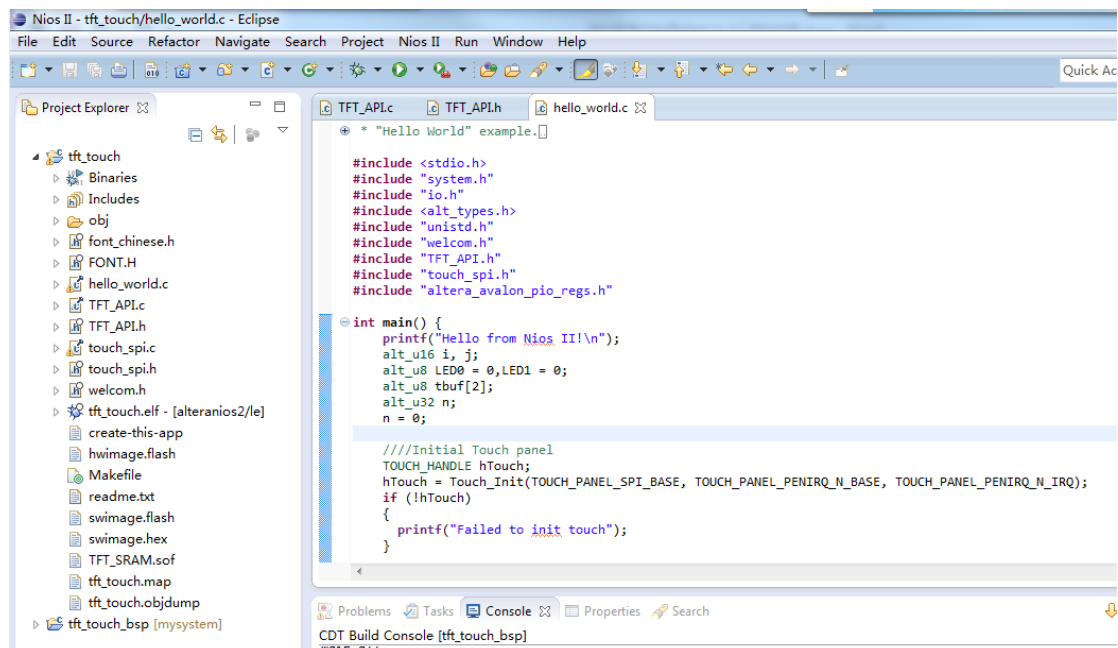
## 7.3 详细转换步骤

### 7.3.1 sof2flash:

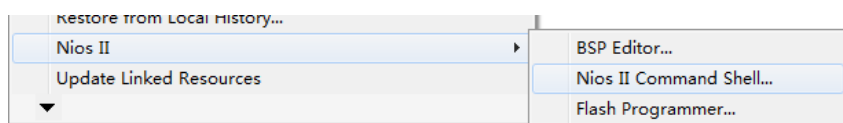
从一个.sof 文件生成一个 flash 文件：

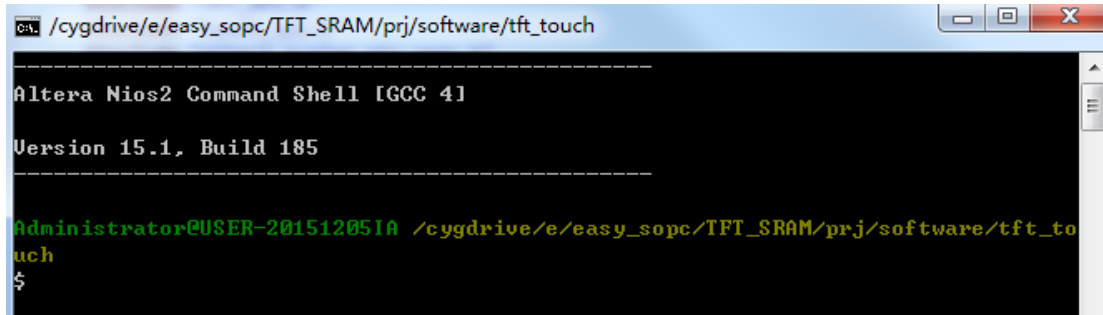
```
sof2flash --input=<hwimage>.sof --output=hwimage.flash --epcs -verbose
```

首先我们打开我们的 NIOS II 软件工程和对应板级支持包，这里名为 tft\_touch 和 tft\_touch\_bsp



然后选中 tft\_touch，单击右键选择 Nios II -> Nios II Command Shell

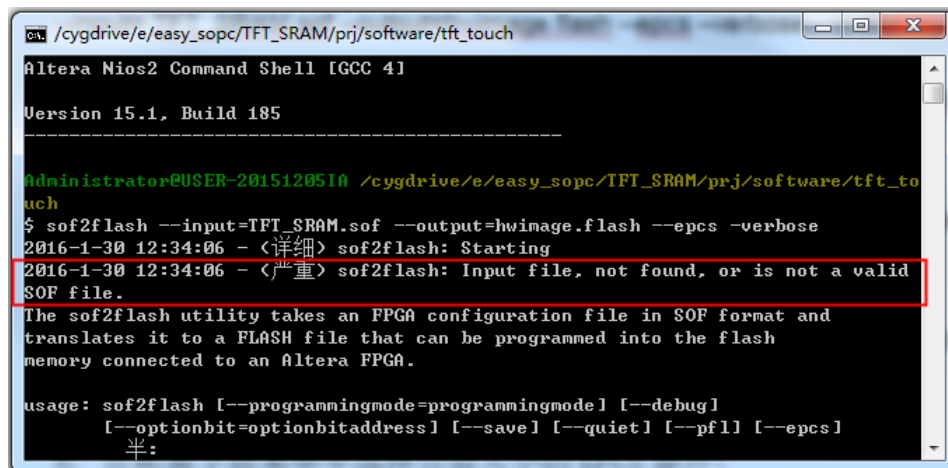




我们的 Quartus II 生产的 sof 文件名为”TFT\_SRAM.sof”，这个时候，如果我们直接输入

```
sof2flash --input=TFT_SRAM.sof --output=hwimage.flash --epcs -verbose
```

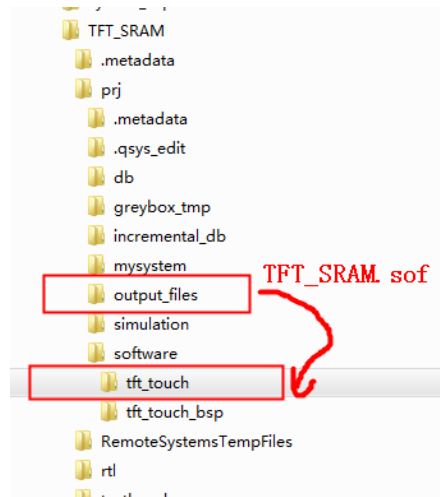
会提示找不到 input file 也就是找不到 TFT\_SRAM.sof 文件。



这是因为该命令是在当前目录下寻找 TFT\_SRAM.sof 文件，而我们的 TFT\_SRAM.sof 文件在 E:\easy\_soc\TFT\_SRAM\prj\output\_files 目录下，因此当然无法找到该文件了。解决这个问题有两种方法。

第一种，推荐方案。

因为很多不熟悉命令行的朋友在操作时速度慢而且容易出错，因此这里提供一种比较熟悉的方式。首先在 windows 中，将 TFT\_SRAM.sof 文件从 output\_files 文件夹中拷贝到 tft\_touch 文件夹中：



然后回到命令行窗口再次执行

`sof2flash --input= TFT_SRAM.sof --output=hwimage.flash --epcs -verbose` 命令

（提示：使用键盘上的向上方向键，可以快速切换到之前使用过的命令，这里在切换目录后，连接两次方向上键就应该能找到之前输入的 `sof2flash` 命令。）生成过程大约花费 10 秒钟。生成完成后的截图如下所示：

```
Administrator@USER-2015120510 /cygdrive/e/easy_soc/TFT_SRAM/prj/software/tft_to
uch
$ sof2flash --input=TFT_SRAM.sof --output=hwimage.flash --epcs -verbose
2016-1-30 13:04:18 - <详细> sof2flash: Starting
Info: Running Quartus Prime Convert_programming_file
Info: Command: quartus_cpf --no_banner --convert --device=EPCS128 --option=hwima
ge.opt TFT_SRAM.sof hwimage.pof
Info <210033>: Memory Map File hwimage.map contains memory usage information for
file hwimage.pof
Info: Quartus Prime Convert_programming_file was successful. 0 errors, 0 warning
s
Info: Peak virtual memory: 256 megabytes
Info: Processing ended: Sat Jan 30 13:04:24 2016
Info: Elapsed time: 00:00:01
Info: Total CPU time (on all processors): 00:00:01
Info: Running Quartus Prime Convert_programming_file
Info: Command: quartus_cpf --no_banner --convert hwimage.pof hwimage.rpd
Info: Quartus Prime Convert_programming_file was successful. 0 errors, 0 warning
s
Info: Peak virtual memory: 251 megabytes
Info: Processing ended: Sat Jan 30 13:04:31 2016
Info: Elapsed time: 00:00:03
Info: Total CPU time (on all processors): 00:00:03
2016-1-30 13:04:31 - <详细> sof2flash: Done
Administrator@USER-2015120510 /cygdrive/e/easy_soc/TFT_SRAM/prj/software/tft_to
uch
$
```

然后我们输入 `ls` 命令就能看到，确实生成了这样一个名为 `hwimage.flash` 的文件：

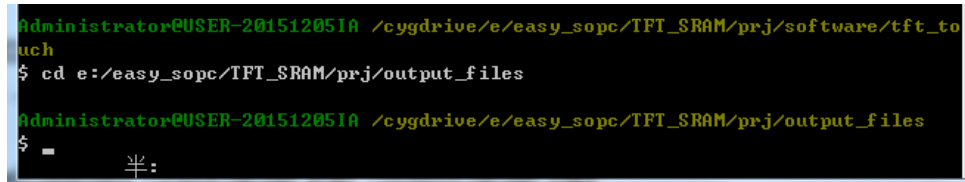
```
Administrator@USER-2015120510 /cygdrive/e/easy_soc/TFT_SRAM/prj/software/tft_to
uch
$ ls
FONT.H      TFT_SRAM.sof  hwimage.flash  tft_touch.map  welcon.h
Makefile    create-this-app obj            tft_touch.objdump
TFT_API.c   font_chinese.h readme.txt     touch_spi.c
TFT_API.h   hello_world.c tft_touch.elf  touch_spi.h

Administrator@USER-2015120510 /cygdrive/e/easy_soc/TFT_SRAM/prj/software/tft_to
uch
$
```



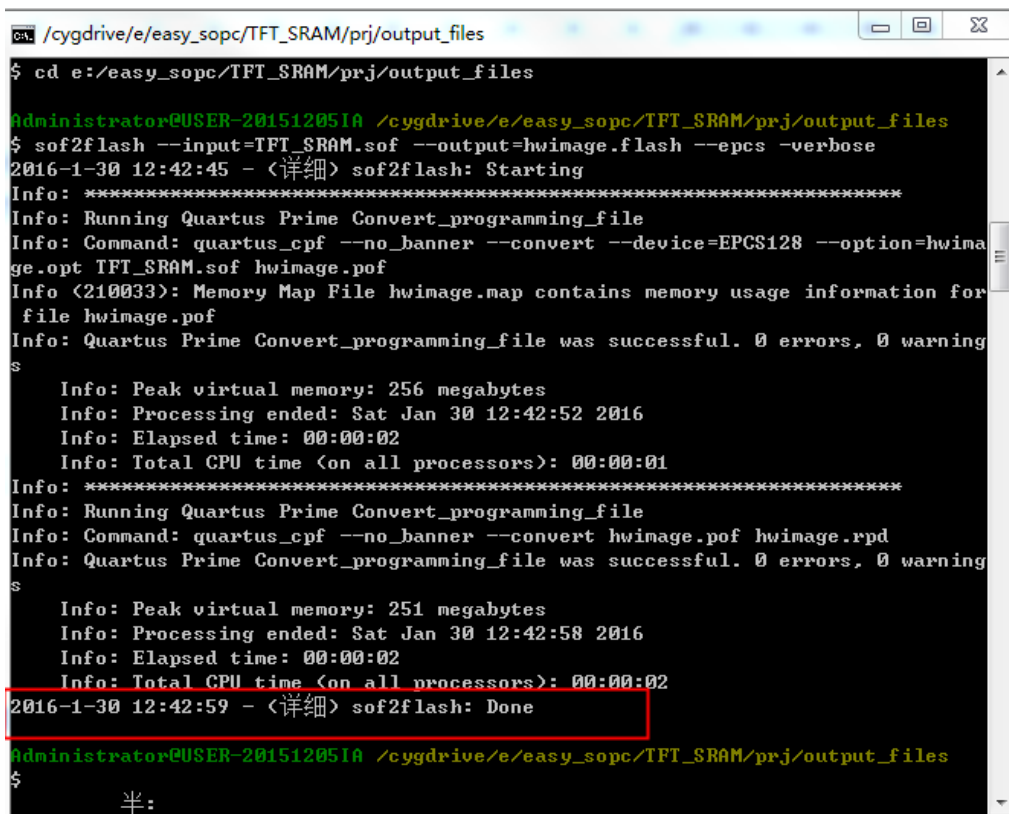
第二种方案：首先在 shell 中使用 cd 命令直接将目录切换到 sof 文件所在目录，也就是 E:\easy\_soc\TFT\_SRAM\prj\output\_files。相应命令为（注意斜线方向）：

```
cd e:/easy_soc/TFT_SRAM/prj/output_files
```



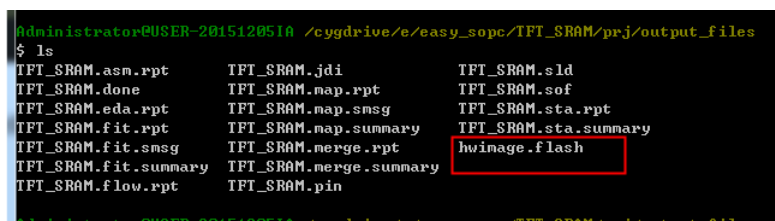
```
Administrator@USER-201512051A /cygdrive/e/easy_soc/TFT_SRAM/prj/software/tft_to
uch
$ cd e:/easy_soc/TFT_SRAM/prj/output_files
Administrator@USER-201512051A /cygdrive/e/easy_soc/TFT_SRAM/prj/output_files
$
```

然后再次执行 sof2flash 命令即可实现。生成完成后的截图如下所示：



```
Administrator@USER-201512051A /cygdrive/e/easy_soc/TFT_SRAM/prj/output_files
$ cd e:/easy_soc/TFT_SRAM/prj/output_files
Administrator@USER-201512051A /cygdrive/e/easy_soc/TFT_SRAM/prj/output_files
$ sof2flash --input=TFT_SRAM.sof --output=hwimage.flash --epcs -verbose
2016-1-30 12:42:45 - <详细> sof2flash: Starting
Info: *****
Info: Running Quartus Prime Convert_programming_file
Info: Command: quartus_cpf --no_banner --convert --device=EPCS128 --option=hwima
ge.opt TFT_SRAM.sof hwimage.pof
Info (210033): Memory Map File hwimage.map contains memory usage information for
file hwimage.pof
Info: Quartus Prime Convert_programming_file was successful. 0 errors, 0 warning
s
Info: Peak virtual memory: 256 megabytes
Info: Processing ended: Sat Jan 30 12:42:52 2016
Info: Elapsed time: 00:00:02
Info: Total CPU time (on all processors): 00:00:01
Info: *****
Info: Running Quartus Prime Convert_programming_file
Info: Command: quartus_cpf --no_banner --convert hwimage.pof hwimage.rpd
Info: Quartus Prime Convert_programming_file was successful. 0 errors, 0 warning
s
Info: Peak virtual memory: 251 megabytes
Info: Processing ended: Sat Jan 30 12:42:58 2016
Info: Elapsed time: 00:00:02
Info: Total CPU time (on all processors): 00:00:02
2016-1-30 12:42:59 - <详细> sof2flash: Done
Administrator@USER-201512051A /cygdrive/e/easy_soc/TFT_SRAM/prj/output_files
$
```

然后我们输入 ls 命令就能看到，确实生成了这样一个名为 hwimage.flash 的文件：



```
Administrator@USER-201512051A /cygdrive/e/easy_soc/TFT_SRAM/prj/output_files
$ ls
TFT_SRAM.asm.rpt      TFT_SRAM.jdi          TFT_SRAM.sld
TFT_SRAM.done         TFT_SRAM.map.rpt      TFT_SRAM.sof
TFT_SRAM.eda.rpt      TFT_SRAM.map.smsg     TFT_SRAM.sta.rpt
TFT_SRAM.fit.rpt      TFT_SRAM.map.summary  TFT_SRAM.sta.summary
TFT_SRAM.fit.smsg     TFT_SRAM.merge.rpt    hwimage.flash
TFT_SRAM.fit.summary  TFT_SRAM.merge.summary
TFT_SRAM.flow.rpt     TFT_SRAM.pin
```

一般推荐大家使用第一种方式，当然命令行高手除外。

### 7.3.2 elf2flash:

从一个 elf 生成一个 flash 文件：

店铺：<https://xiaomeige.taobao.com>

技术博客：<http://www.cnblogs.com/xiaomeige/>

官方网站：[www.corecourse.cn](http://www.corecourse.cn)

技术群组：有点多，不列举

```
elf2flash --input=<elf file>.elf --output=swimage.flash --epcs --after=hwimage.flash --verbose
```

因为推荐大家使用第一种方式操作，因此这里就按照第一种方式接着讲，相信有能力用命令行方式切换目录的朋友，也不会对其他操作存在问题。

这里我们就只需要直接输入 `elf2flash` 命令即可了，命令详细如下：

```
elf2flash --input=tft_touch.elf --output=swimage.flash --epcs --after=hwimage.flash --verbose
```

从命令中可以看到，第一步生成的 `hwimage.flash` 文件是作为参数的一部分的，所以这里必须保证 `hwimage.flash` 在当前目录下。（第一步中使用推荐的方式，则能够自动保证这一点）。命令执行结果如下：

```
Administrator@USER-201512051A /cygdrive/e/easy_soc/TFT_SRAM/prj/software/tft_touch
$ elf2flash --input=tft_touch.elf --output=swimage.flash --epcs --after=hwimage.flash --verbose
Expected integer value for option 'end'
Value was: 'rbosc'
Using default value: 4294967295
2016-1-30 13:17:03 - <信息> elf2flash: args = --input=tft_touch.elf --output=swimage.flash --epcs --after=hwimage.flash --verbose
2016-1-30 13:17:03 - <详细> elf2flash: Starting
2016-1-30 13:17:03 - <较详细> elf2flash: Program Record: 361216 bytes destined for 0x20
2016-1-30 13:17:03 - <较详细> elf2flash: Program Record: 32 bytes destined for 0x1001800
2016-1-30 13:17:03 - <较详细> elf2flash: Start Record: 1bc
2016-1-30 13:17:03 - <详细> elf2flash: Done
```

然后我们输入 `ls` 命令，可以看到，在当前文件夹下确实生成了一个名为 `swimage.flash` 的文件：

```
Administrator@USER-201512051A /cygdrive/e/easy_soc/TFT_SRAM/prj/software/tft_touch
$ ls
FONT.H      TFT_SRAM.sof  hwimage.flash  tft_touch.elf  touch_spi.h
Makefile    create-this-app obj            tft_touch.map  welcom.h
TFT_API.c   font_chinese.h readme.txt     tft_touch.objdump
TFT_API.h   hello_world.c swimage.flash  touch_spi.c
```

### 7.3.3 flash2hex:

转换 `.flash` 文件到 `.hex` 文件:直接输入以下命令（注意:altera 官网中原帖这个地方命令有误，前后对应不上,原帖为 `nios2-e..... mysw.flash mysw.hex`,应该讲 `mysw` 改为 `swimage`）：

```
nios2-elf-objcopy --input-target srec --output-target ihex swimage.flash swimage.hex
```

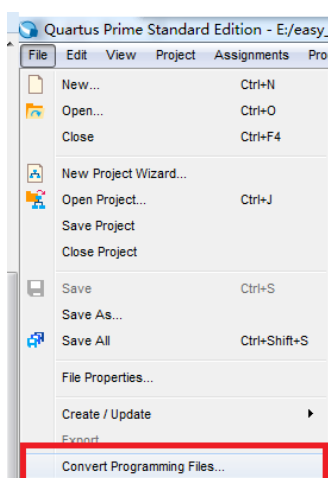
这个命令瞬间执行完成，我们 `ls` 下，就能看到当前文件夹下已经生成了一个 `swimage.hex` 的文件：

```
Administrator@USER-2015120510 /cygdrive/e/easy_soc/TFT_SRAM/prj/software/tft_to
uch
$ nios2-elf-objcopy --input-target src --output-target ihex swimage.flash swin
age.hex

Administrator@USER-2015120510 /cygdrive/e/easy_soc/TFT_SRAM/prj/software/tft_to
uch
$ ls
FONT.H      TFT_SRAM.sof  hwinage.flash  swimage.hex    touch_spi.c
Makefile    create-this-app  obj            tft_touch.elf  touch_spi.h
TFT_API.c   font_chinese.h  readme.txt     tft_touch.map  welcom.h
TFT_API.h   hello_world.c   swimage.flash  tft_touch.objdump
```

### 7.3.4 Convert Programming Files

在 Quartus® II 软件中，open File > Convert Programming Files > Set the programming file as JTAG Indirect Configuration File (.jic).



### 7.3.5 选择 EPCS

在配置下拉菜单中选择合适大小的 EPCS 器件（见 10 步图）

### 7.3.6 命名 jic

命名你的输出. jic 文件（见 10 步图）

### 7.3.7 Add Device

点击 Flash Loader 的下面，在右边选择 Add Device （见 10 步图）

### 7.3.8 选择 FPGA 器件

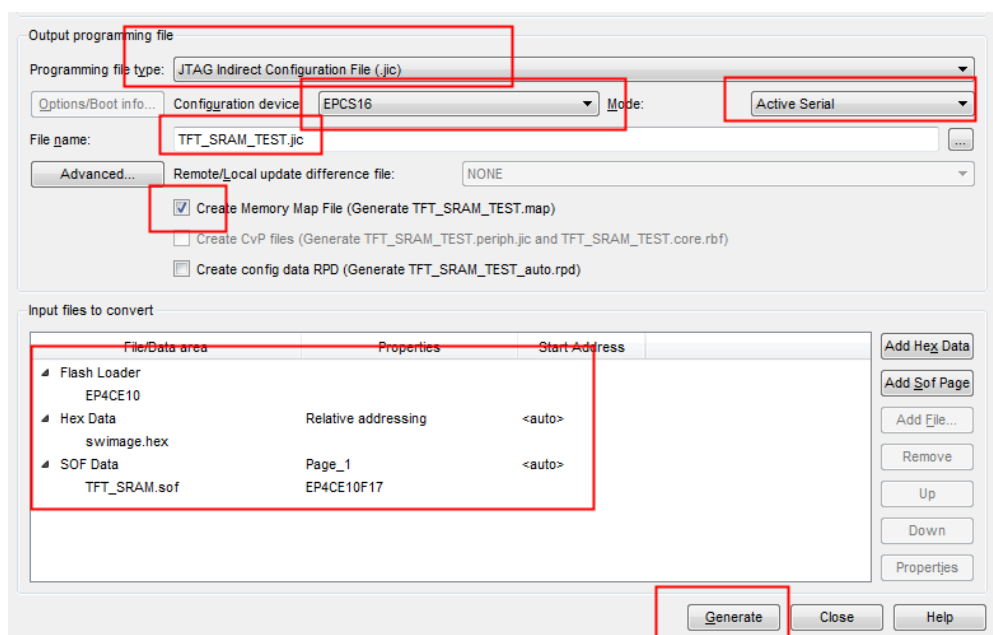
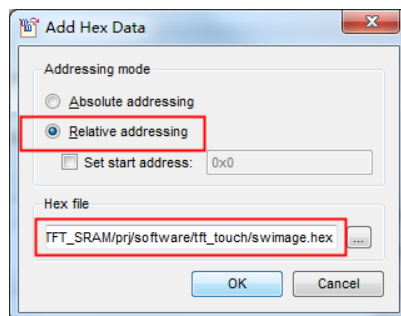
从列表中选择你的 FPGA 器件（见 10 步图）

### 7.3.9 Add SOF

点击 SOF Data, 选择 Add File, 选择加 .sof 文件 (见 10 步图)

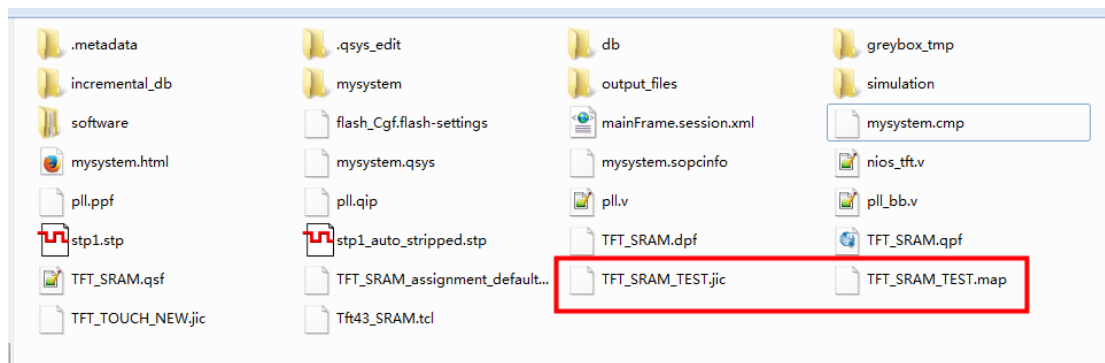
### 7.3.10 Add Hex data

点击 Add Hex data, 选择 Relative addressing, 选择上面生成的 .hex 文件



### 7.3.11 Generate

然后点击 Generate 生成。生成完成后检查生成的 .map 文件 (使用记事本打开) 有 Page\_0 在起始地址 0x0, .hex 文件在 Page\_0 结束地址后的起始地址 1



BLOCK	START ADDRESS	END ADDRESS
Page_1	0x00000000	0x00059D8A
swimage.hex	0x00059D8B	0x000B20C2

Configuration device: EP4CE10  
Configuration mode: Active Serial  
Quad-Serial configuration device dummy clock cycle: 8

Notes:

- Data checksum for this conversion is 0x1765EE65
- All the addresses in this file are byte addresses

### 7.3.12 烧写

现在在 Quartus II Programmer 中，选择 Add File，选择加.jic 文件。检查 Program 框，下一步.jic 文件，接着按 Start 即可。

### 7.3.13 测试效果

最后上一张测试图，女神镇楼。







## 擦除 EPCS 存储器中内容的方法

### 为啥要擦除？

在调试程序时候，EPCS 中原本存储的程序内容，上电会直接运行，然后我们在调试程序的时候，一般都是使用 jtag 在线下载程序，这里就存在一个问题：对于一些器件的配置工作可能发生冲突，例如使用 IIC 配置摄像头，芯片上电会从 EPCS 重读取固件，然后执行对摄像头的配置，然后我们在调试的时候，如果我们新下载的程序中没有完善的对摄像头的复位机制，就有可能导致 EPCS 中存储的固件对摄像头的一些寄存器的操作会影响到新的程序的运行。

尤其是在调试 NIOS II 的程序时，如果 EPCS 中已经烧录了有 NIOS II 的逻辑，甚至烧录了 NIOS II 的软件程序，那么在调试的时候，由于 NIOS II 启动和复位机制的问题，当 NIOS II 的软件程序下载进去之后，如果执行复位，有可能执行的并不是你刚刚下载的 elf 文件，而是之前存储在 EPCS 中的程序代码，就会出现各种稀奇古怪的问题，比如下载 elf 程序失败，运行效果和编写的 C 程序理论运行效果牛头不对马嘴等。

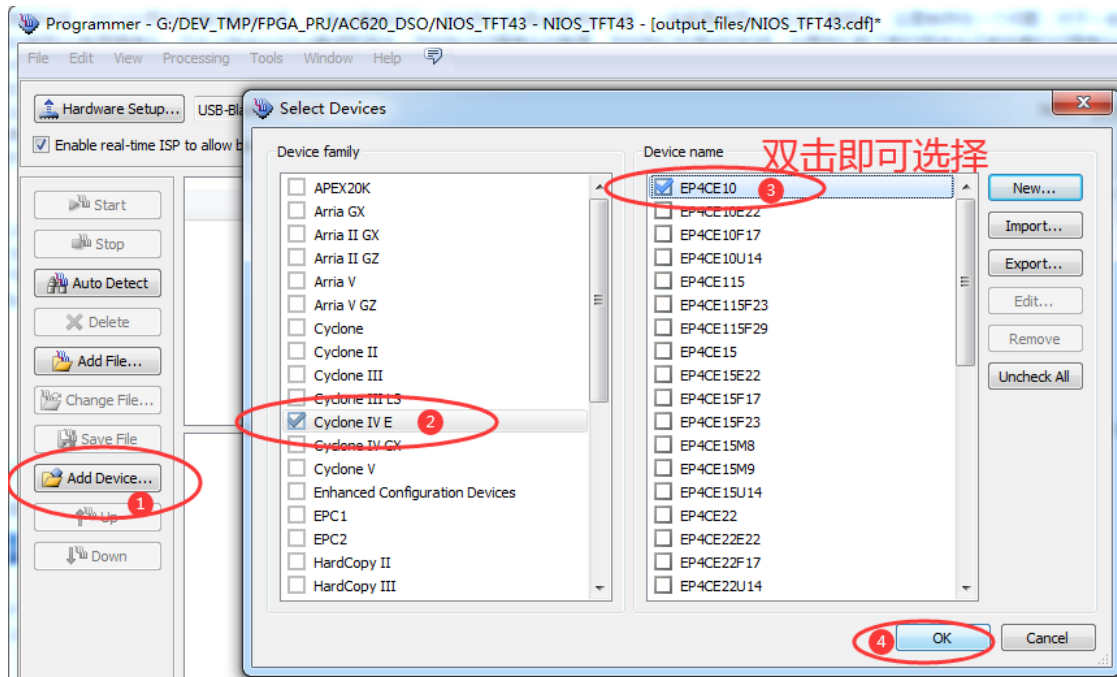
为了避开这些个影响，有必要保证 EPCS 中没有存放任何有可能影响到当前调试的逻辑和程序的数据。所以需要在调试之前擦除它。

### 怎么擦除？

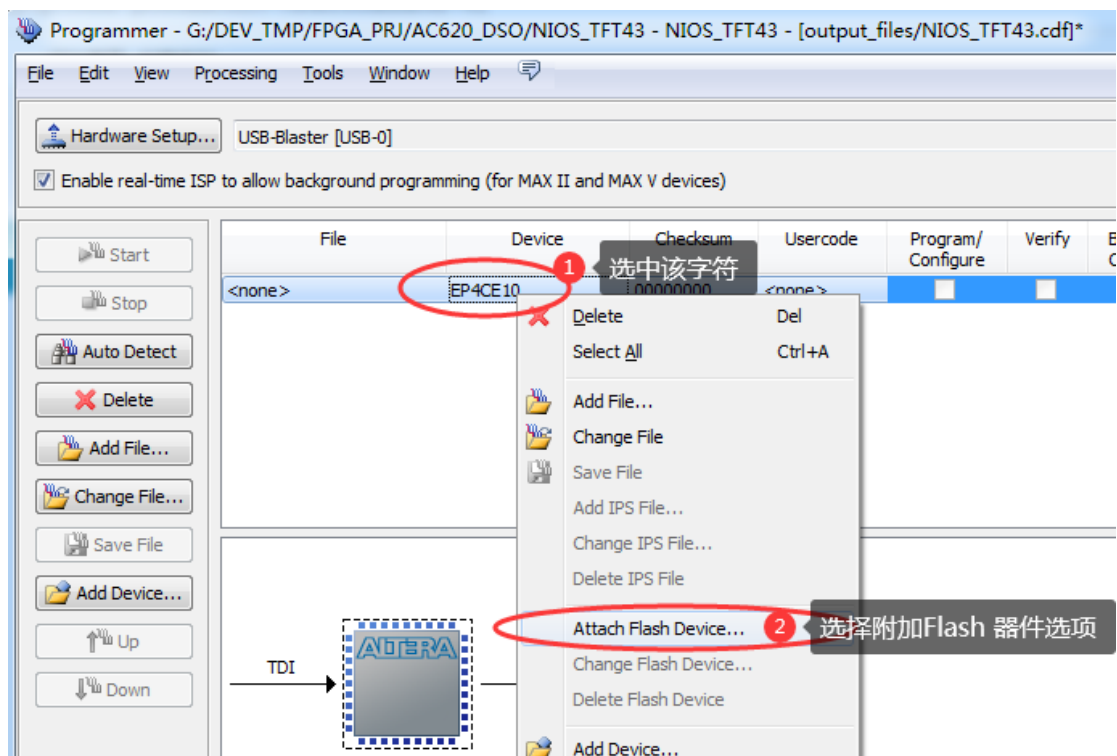
擦除方法很简单，使用下载器链接板子和电脑，打开下载界面经过简单的添加就能完成擦除工作，以下用图解的方式展示。

在下载界面中删除所有之前添加或者自动添加的文件。

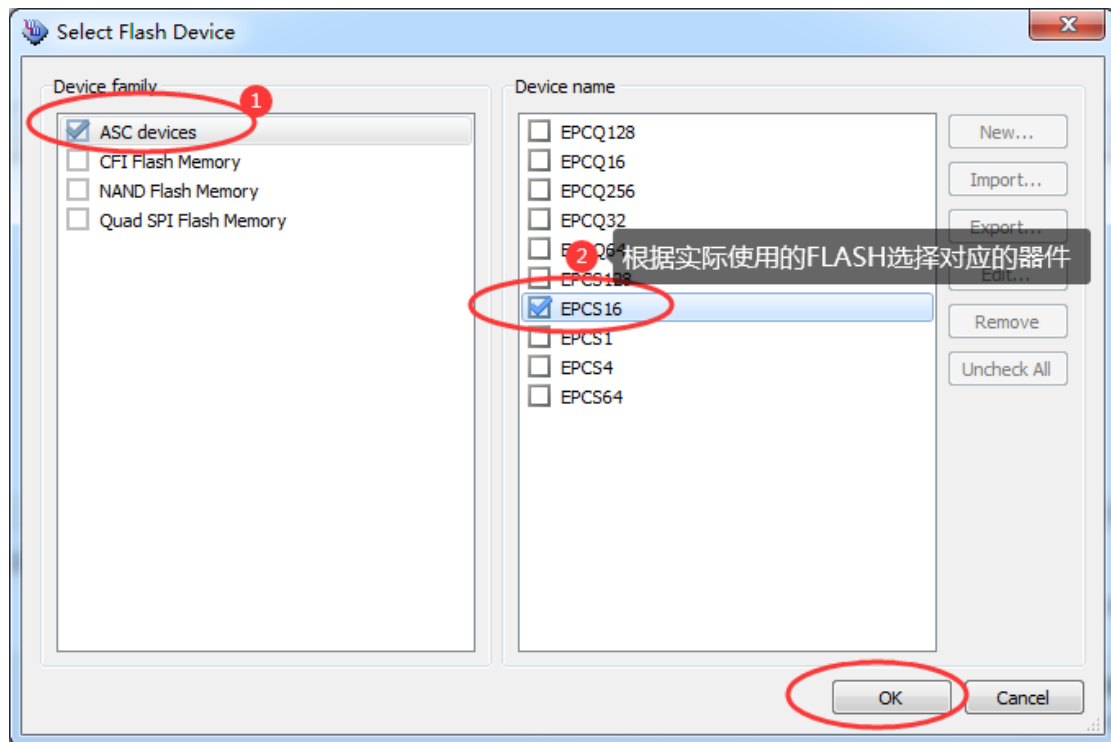
点击 Add Device 按钮，在弹出的对话框中找到你所使用的 FPGA 芯片的信号。例如对于 AC620，使用的是 EP4CE10F17C8，那么选择 Cyclone IV E 下的 EP4CE10 即可（双击 EP4CE10 即可选择）。然后按 OK 键退出。



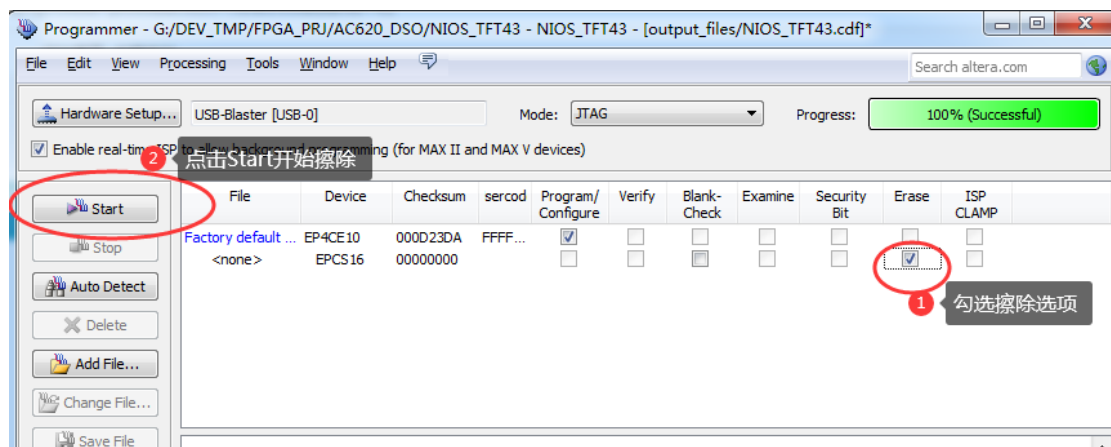
在下载界面中选中刚刚添加的 EP4CE10，然后鼠标右击，在弹出的对话框中选择 Attach Flash Device。



在弹出的界面中，选择 ASC device 下的对应型号的 Flash 器件，例如 AC620 开发板使用的是 EPCS16，因此选择 EPCS16，然后点击 OK 退出。



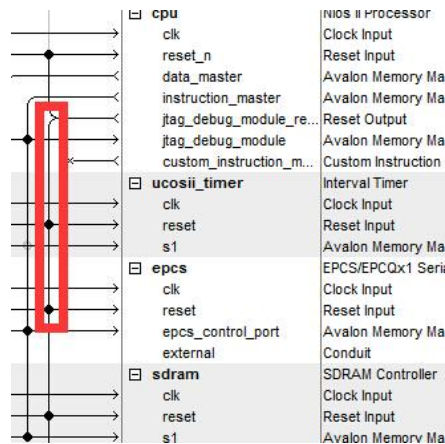
勾选 Erase 选项，然后点击 Start 按钮即可开始擦除 Flash。



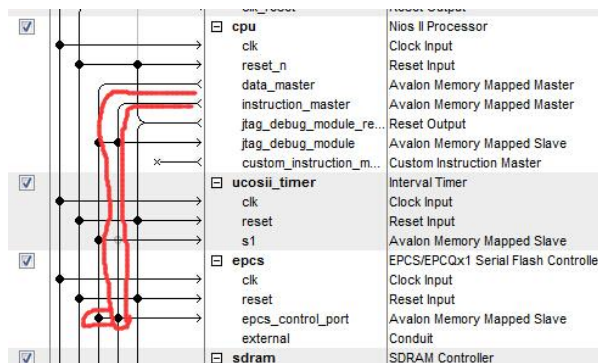
擦除完成后，给开发板重新上电，即可通过实际上电现象验证 EPCS 是否擦除完成。

## EPCS 控制器的添加和使用

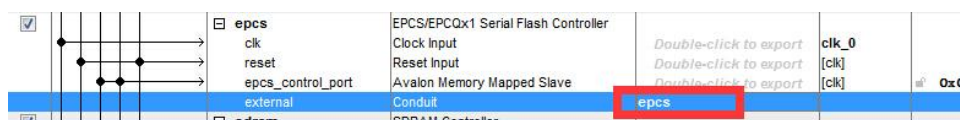
- 1、在需要最终将程序固化到 EPCS 的系统中，需要添加 EPCS/EPCQx1 Serial Flash Control。该控制器的 reset 信号一定要与 jtag debug module reset 信号（cpu 模块中）相连，另外，最好其他所有模块的 reset 信号都与 jtag debug module reset 信号连接上，否则，在最终通过 Flash Programmer 固化程序时，会出错。如下图所示：



2、epcs 的 Avalon Memory Mapped 端口需要与 CPU 的 data\_master 和 instruction\_master 均进行连接。如下图所示：



3、EPCS 的 external 信号需要导出到顶层（针对 Cyclone III 和 Cyclone IV 器件），以便进行引脚分配，如下图所示：



这里，导出到顶层后具体怎么分配引脚，在 Altera 的《Embedded Peripheral IP User Guide》中有相关介绍，

- On the **Dual-purpose pins** page (**Assignments > Devices > Device and Pin Options**), ensure that the following pins are assigned to the respective values:
  - Data[0] = **Use as regular I/O**
  - Data[1] = **Use as regular I/O**
  - DCLK = **Use as regular I/O**
  - FLASH\_nCE/nCS0 = **Use as regular I/O**
- Using the **Pin Planner** (**Assignments > Pins**), ensure that the following pins are assigned to the respective configuration functions on the device:
  - data0\_to\_the\_epcs\_controller = DATA0
  - sdo\_from\_the\_epcs\_controller = DATA1, ASDO
  - dclk\_from\_epcs\_controller = DCLK
  - sce\_from\_the\_epcs\_controller = FLASH\_nCE

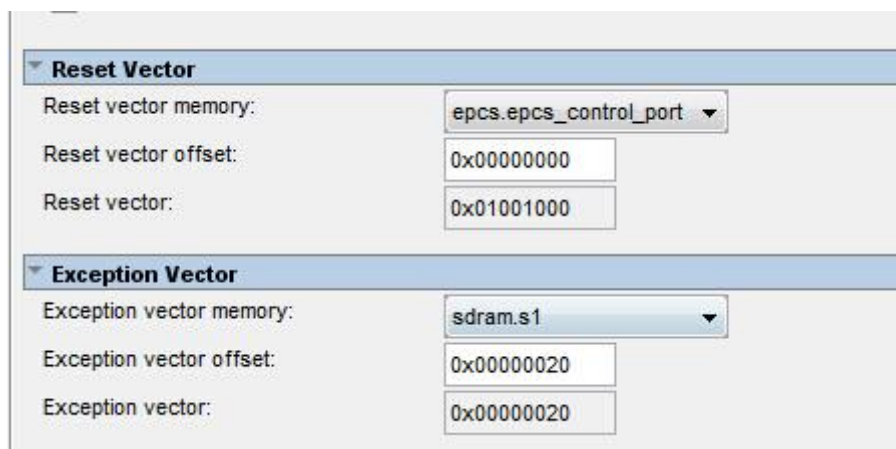
引脚分配完成后，需要在 Quartus II 中依次点击【Assignments】->【Device】，在弹出的界面中选择“Device and Pin Options”，在 Dual-Purpose Pins 中设置 DCLK、Data[0]、Data[1]/ASDO、FLASH\_nCE/nCS0 的 Value 为“Use as regular I/O”（双击值，然后在下拉



菜单中选择)。否则 Quartus II 进行全编译会报如下错误:

```
✖ 176310 Can't place multiple pins assigned to pin location Pin_C1 (IOPAD_X0_Y22_N21)
i 176311 Pin epcs_flash_sdo is assigned to pin location Pin_C1 (IOPAD_X0_Y22_N21)
i 176311 Pin ~ALTERA_ASDO_DATA1~ is assigned to pin location Pin_C1 (IOPAD_X0_Y22_N21)
✖ 176310 Can't place multiple pins assigned to pin location Pin_D2 (IOPAD_X0_Y21_N14)
i 176311 Pin epcs_flash_sce is assigned to pin location Pin_D2 (IOPAD_X0_Y21_N14)
i 176311 Pin ~ALTERA_FLASH_nCE_nCS0~ is assigned to pin location Pin_D2 (IOPAD_X0_Y21_N14)
✖ 176310 Can't place multiple pins assigned to pin location Pin_H1 (IOPAD_X0_Y17_N14)
i 176311 Pin epcs_flash_dclk is assigned to pin location Pin_H1 (IOPAD_X0_Y17_N14)
i 176311 Pin ~ALTERA_DCLK~ is assigned to pin location Pin_H1 (IOPAD_X0_Y17_N14)
✖ 176310 Can't place multiple pins assigned to pin location Pin_H2 (IOPAD_X0_Y17_N21)
i 176311 Pin epcs_flash_data0 is assigned to pin location Pin_H2 (IOPAD_X0_Y17_N21)
i 176311 Pin ~ALTERA_DATA0~ is assigned to pin location Pin_H2 (IOPAD_X0_Y17_N21)
```

4、CPU 的复位向量设置为 EPCS，CPU 的异常向量设置为内存 (on\_chip\_memory 或 SDRAM)，如下图所示：



## 使用非官方 EPCS 存储器固化 NIOS II 软件

Altera 器件有 EPCS 系列配置器件，其实，这些配置器件就是我们平时通用的 SPIFlash，据 AlteraFAE 描述：“EPCS 器件也是选用某家公司的 SPIFlash，只是中间经过 Altera 公司的严格测试，所以稳定性及耐用性都超过通用的 SPIFlash”。就本人看来，半导体的稳定性问题绝大部分都是由本身设计缺陷造成的，而成熟的制造工艺不会造成产品的不稳定；并且，现在 Altera 的器件在读入配置数据发生错误时，可以重新读取 SPIFlash 里面的数据，所以在工艺的稳定性以及设计的可靠性双重保证下，通过选用通用的 SPIFlash 来减少产品的成本压力。

假设我们正在使用一个普通 SPIFlash，打开 nios II command shell 窗口，使用 nios2-flash-programmer 命令下载\*\*\*.flash 文件时，会发生如下错误：

No EPCS layout data --- looking for section [EPCS-1C2017]

不同公司的 SPIFlash 有不同的 ID，并且不同大小的 Flash 的 Sector 大小及个数都不一样，所以需要新建一个文档去说明这些数据：

1. 首先在<nios2\_install>/bin 文件夹(如作者电脑上的路径为 D:\altera\13.0\nios2eds\bin)下面新建 nios2-flash-override.txt 文件；
2. 输入下述代码，下面描述的器件都是 Altera 的 EPCS 器件，sector\_size 表示 sector 大小，sector\_count 表示 sector 个数；



```
[EPCS-202011] # EPCS1N (lead-free)
```

```
sector_size = 32768
```

```
sector_count = 4
```

```
[EPCS-202013] # EPCS4N (lead-free)
```

```
sector_size = 65536
```

```
sector_count = 8
```

```
[EPCS-202015] # EPCS16N (lead-free)
```

```
sector_size = 65536
```

```
sector_count = 32
```

```
[EPCS-202017] # EPCS64N (lead-free)
```

```
sector_size = 65536
```

```
sector_count = 128
```

3. 在上述代码中添加自己选择的通用 SPIFlash，例如：

```
[EPCS-EF4015] # EPCS16N (lead-free)
```

```
sector_size = 65536
```

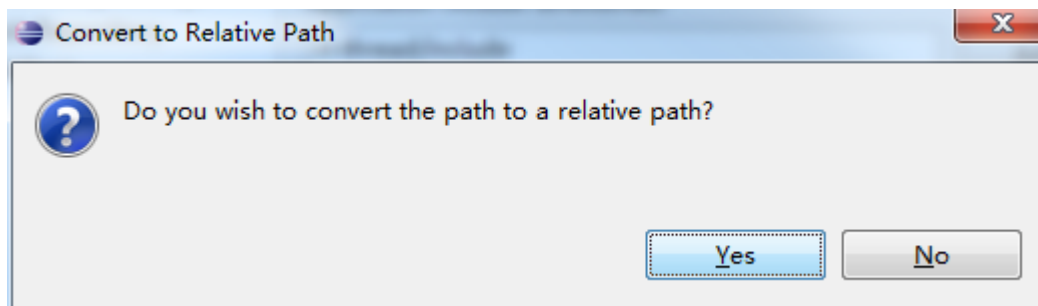
```
sector_count = 32
```

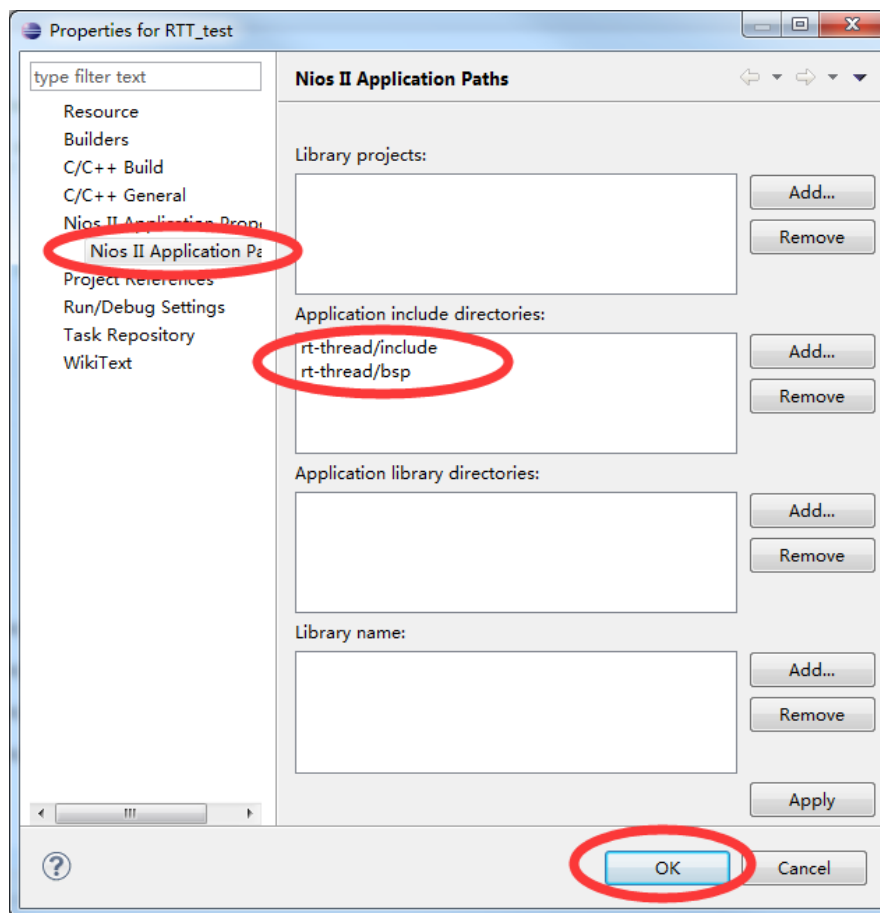
4. 然后再使用 `nios2-flash-programmer` 命令下载\*\*\*.flash 文件，就可以对 SPIFlash 进行下载了。

## 设置 NIOS II 软件工程头文件包含路径

本节内容节选自<第 5 章 RT-Thread 操作系统在 NIOS II 上的移植>

添加完所有的文件之后，我们还必须要在软件的设置中添加头文件搜索路径，选中 RTT\_test 工程，鼠标右击选择 Properties 选项，在弹出的对话框中选中 Nios II Application Paths 选项，添加 include 和 bsp 文件夹到头文件路径中，然后确认关闭。如果弹出下述相对路径转换提示，选择 Yes 即可。





至此，所有的运行 RT-Thread 操作系统的要求都已经满足了。

## 关闭 NIOS 开发环境中 IP 自带驱动的原因和方法

### 1、什么是 NIOS 开发环境中 IP 自带驱动

在开发基于 NIOS II 的应用程序时，常用到包括 SPI、定时器、UART 串口等 IP 核，只要在 Qsys 中添加好这些 IP，然后在 NIOS II 的开发环境（定制版 Eclipse）中编写驱动程序和应用程序即可完成该控制器的使用。当然，为了让这些 IP 用起来更加的方便，Intel（Altera）也是为大多数的 IP 提供了相应的驱动程序，例如最典型的使用串口时，我们可以直接使用 `printf` 来通过串口发送数据，通过 `scanf` 来从串口接收数据，这就是使用了 Intel 开发好的驱动程序和 HAL 库。同样的，对于 Qsys 中提供的定时器运行在不同的模式，也有相应的驱动程序。比如看门狗模式、系统心跳时钟模式等。

### 2、为啥要关闭？

出于某些原因，我们需要自己直接通过读写 IP 的寄存器来控制其实现相应功能，比如嫌弃软件自带驱动太过臃肿，增加了程序尺寸，比如想学习下如何通过读写 IP 寄存器实现

其控制，比如软件自带驱动无法很好的满足自己的需求等。总之，我们就是有理由在需要的时候想将其关闭。

### 3、不关闭会怎么样？

由于这些软件自带的驱动在 NIOS II 初始化的时候就会开始执行，如果我们在用户程序中通过读写直接寄存器的方式操作这些 IP，那么就会出现实际上有两个程序同时操作某个寄存器的情况，当一个程序操作了这个寄存器，另一个程序可能就无法操作该寄存器，或者操作该寄存器时无法得到正确的执行。举个简单的例子：

对于串口驱动，系统自带的驱动是使用了中断的方式来完成数据的收发，而我们自己写程序的时候有时候只想简单的通过查询寄存器状态的方式来读写数据。那么假设当串口接收到一个字节的數據后：

首先，由于系统驱动是使用的中断的方式响应，理论上当串口接收到数据后，系统驱动的中断首先响应该事件，并进入中断服务函数读取状态寄存器，根据状态寄存器来获知是接收到了新数据，再去读取接收到寄存器，把收到的数据读出并存起来。

可是，不要忘了，我们用户层也有一个接收的函数，该函数可能就是拼命的读取串口的状态寄存器，再根据状态寄存器的结果来决定是否收到了数据，如果收到了数据，就把数据寄存器中的数据读取出来并存起来。

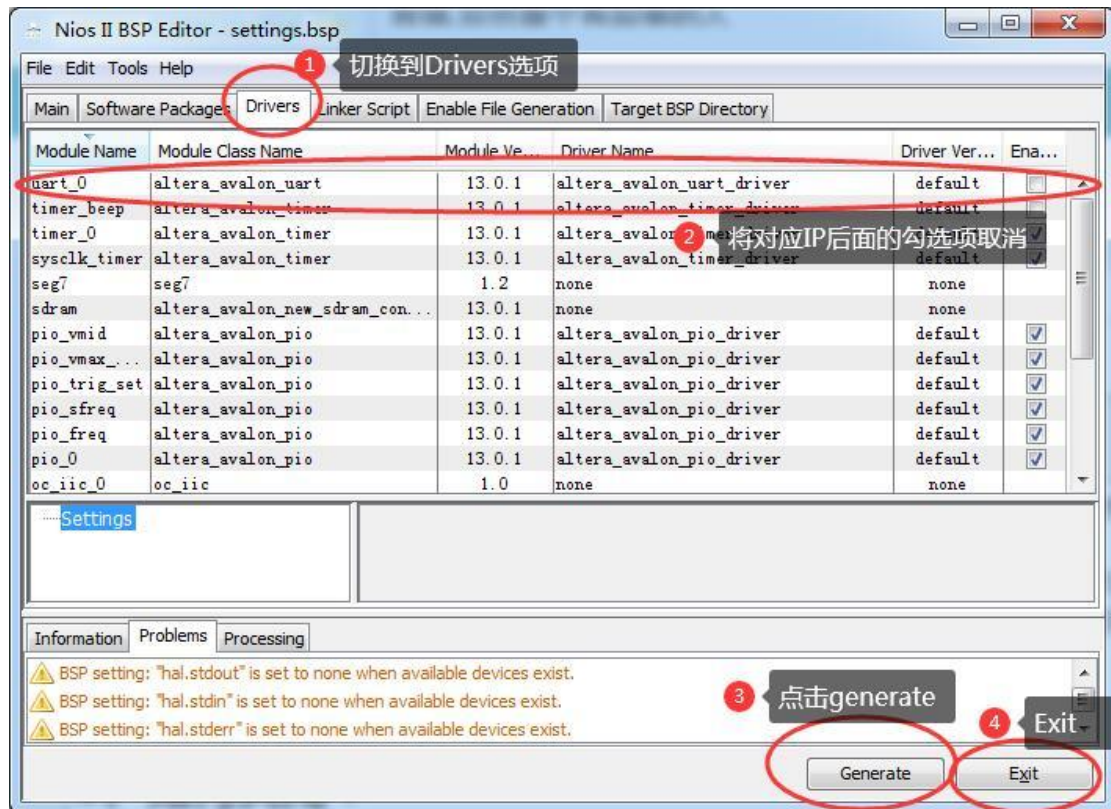
了解串口 IP 核的用户应该要知道，串口的状态寄存器中的位是自动置位，在读取过后自动清零的，也就是说，假设串口收到了一个数据，就会将状态寄存器中的对应位设置为 1，然后，当用户自己的程序或者系统自带的驱动只要有一个程序去读取了该状态寄存器的值，那么该寄存器中的一些位就被自动清零了。

好了，相信不用我再多说，大家也都知道问题出在哪里了，假设系统自带的驱动通过中断的方式抢先响应了接收到一个字节的數據这个事情，并读取了状态寄存器，那么我们自己用户编写的驱动再后面去读取状态寄存器时，就读不到有效的值，这样以来，这个接收到的数据就丢了，哎，丢了。

### 4、如何关闭？

关闭方法非常简单，鼠标左键选中 bsp 工程，注意，是 bsp 工程，不是 app 工程，什么是 bsp 工程，什么是 app 工程，简单点说，假设你按照我们的教程创建了一个名为 hello 的 nios 工程，那么软件会自动帮你创建一个名为 hello\_bsp 的工程，那么这个 hello 就是 app 工程（应用工程），hello\_bsp 就是 bsp 工程（板级支持包工程）。

关闭方法非常简单，鼠标左键选中 bsp 工程，然后在键盘上按下 Ctrl+9 组合键，就能进入到 bsp 设置界面，在该界面中切换到 drivers 选项中，然后在你希望关闭驱动的 IP 后面，将默认打开的勾选项取消即可。



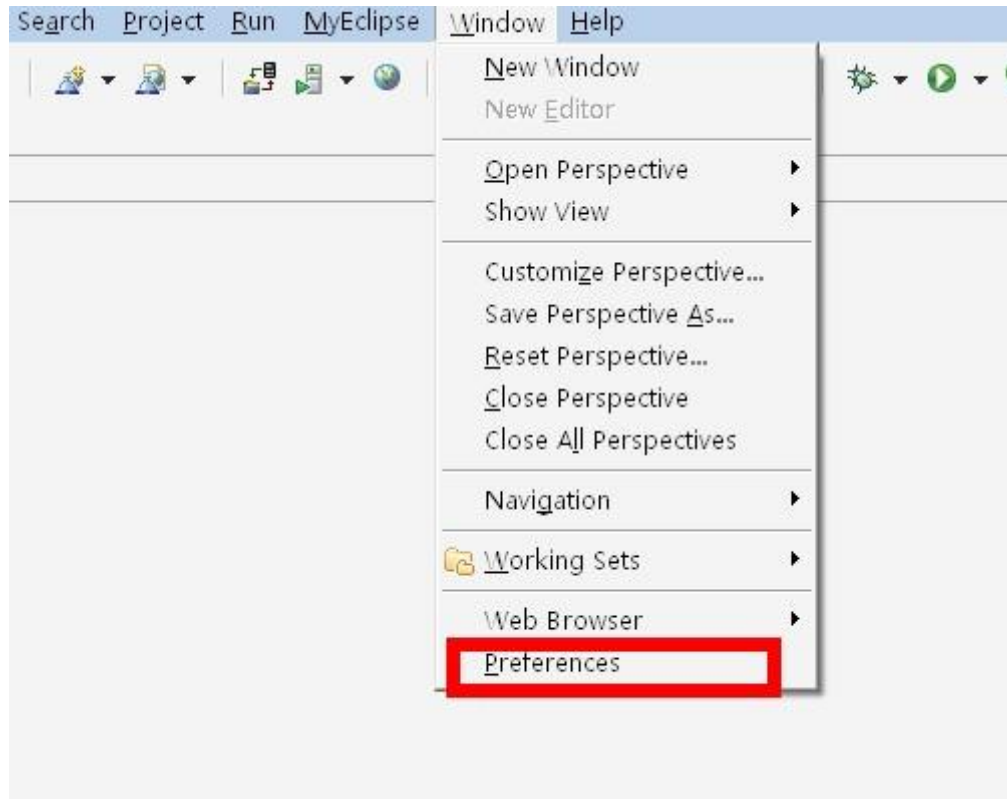
退出之后需要选中 bsp 工程，右键执行一次 NIOS II ->Generate BSP 操作，然后再编译工程（ctrl + B）。

## 设置 Eclipse 在编译(build) 前自动保存源代码文件

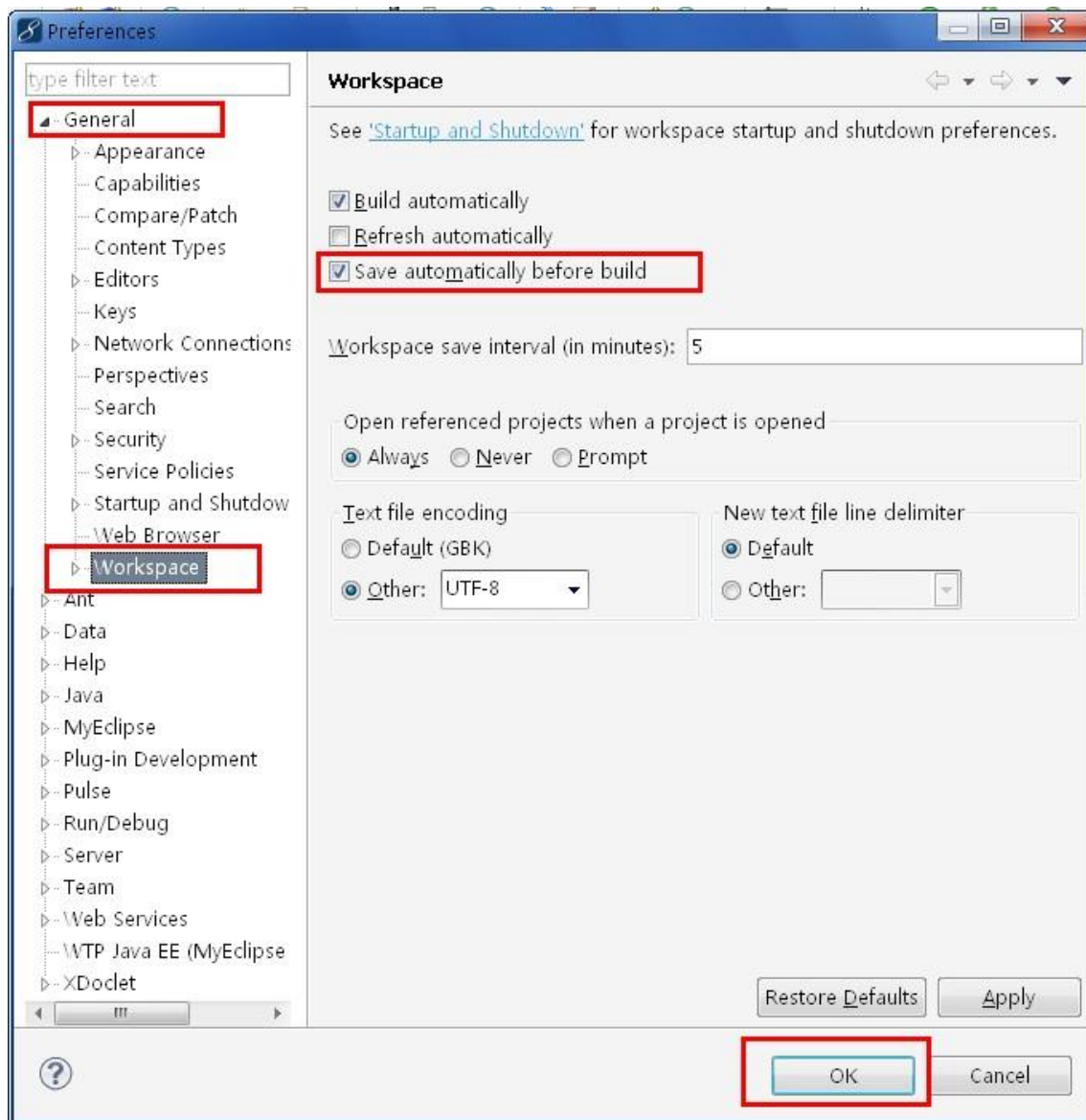
Eclipse 常用设置之让 Eclipse 在编译(build) 前自动保存源代码文件

### 一、让 Eclipse 在编译(build) 前自动保存源代码文件

这个操作很关键，如果编译前不保存。Eclipse 还是编译原来的文件。我经常性把代码改来改去，怎么编译都发现结果不对。结果仔细一看，气死了。修改后的源文件没有保存！







别小看这个设置哦！也许关键时刻能帮你一把！

## 切换 NIOS II CPU 的主内存后软中需要注意的几点设置

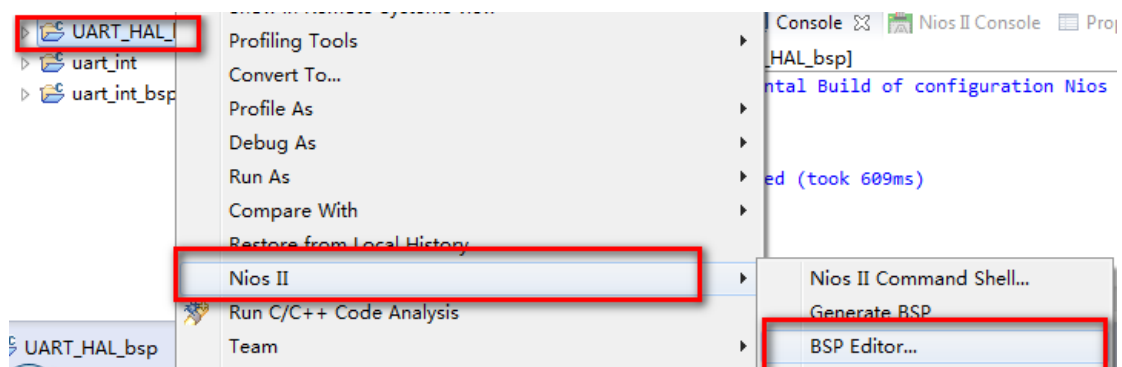
有时候，我们可能面对这样一种情况：

1. 我们创建一个 SOPC 系统，并在 QSYS 中设置 NIOS II 的复位地址和异常地址都指向 SRAM；
2. 我们创建了正确的 NIOS II 软件工程并能够正常运行
3. 由于某种需求（如 SRAM 内存不够用了，需要换成内存更大的 SDRAM），我们在面在 QSYS 中将 NIOS II CPU 的复位地址和异常地址修改为 SDRAM
4. 我们需要继续使用之前创建的 NIOS II 软件工程。

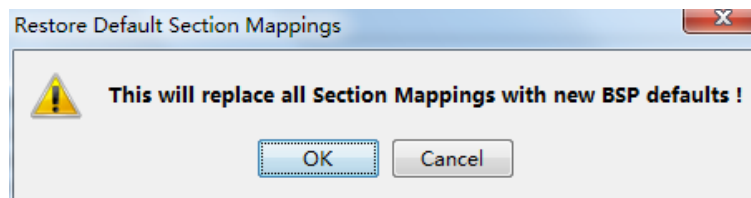
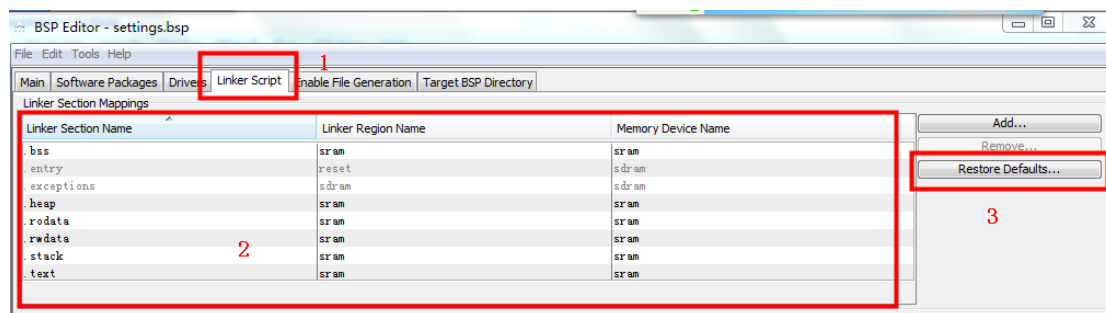
这里，如果我们更改后直接使用原来的软件工程，则编译会报错，因此我们需要在 BSP editor 中重新映射各个数据段的分配，然后重新 generate BSP，否则将导致固件无法烧写或者烧写后程序无法运行。

例如，在芯航线 FPGA 开发套件提供的 NIOS II 教程代码中，我们第 5 个实验和第 6 个实验分别是在 SRAM 和 SDRAM 中运行同样的软件工程（串口驱动设计实验），我们第 6 个实验的 QSYS 系统是直接在第 5 个实验的基础上增加了 SDRAM，并将 NIOS II CPU 的复位地址和异常地址由 SRAM 切换为了 SDRAM。这时候，如果我们直接打开之前的 NIOS II 软件工程，编译会报错。（这里默认用户已经知道并在 settings.bsp 文件中更改了 bsp 工程路径，如果不知道怎么更改的，请参看我的博文：）

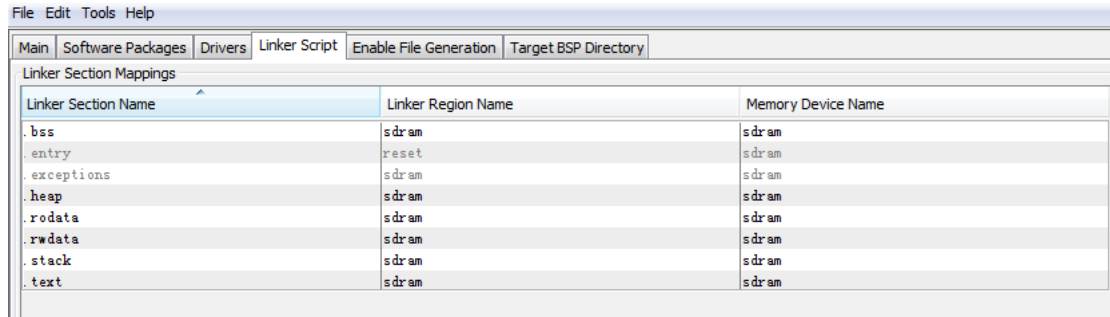
我们检查下这个时候的 bsp 设置：【选中 UART\_hal\_bsp 工程】，【单击右键选择 Nios II】，【在子菜单中选择 BSP Editor】，如下图所示：



在弹出的界面中，切换到【Linker Script】选项卡，可以看到，各个数据段都还是指向了 SRAM，很显然这是不对的，因此我们点击【Restore defaults】按钮。在弹出的对话框中点击 OK 即可。



然后这些数据段的位置就全部更新到 SDRAM 中了。如下图所示：



这个时候我们再点击右下角的 **generate** 按钮，就可以重新生成 bsp，然后再编译工程，下载就能够通过，而且正确运行了。