小梅哥 AC620 FPGA 开发板 SOPC 设计教程

全功能高性价比 FPGA 开发板 AC620 火热销售中，配套 800 页原创电子书。



购买链接：

https://item.taobao.com/item.htm?spm=a1z10.1-c-s.w4004-13687301132.6.8uPVJ6&id=544830995588

目录

Altera SOPC 嵌入式系统设计教程.....4
第 1 章　概述.....4
1.1 SOPC 及其技术.....4
1.2 NiosII 软核处理器.....6
第 2 章 SOPC 开发流程简介.....12
2.1 SOPC 开发流程.....12
2.2 简单 SOPC 实例开发任务及步骤.....17
2.3 分析系统需求.....17
2.4 使用 Quartus II 建立工程.....18
2.5 使用 Qsys 创建 Nios II 系统.....23
2.6 集成 NIOS II 系统到 Quartus II 工程.....41
2.7 编译系统.....44
2.8 下载硬件设计到目标 FPGA.....46
2.9 使用 NIOS II IDE 建立用户程序.....46
2.10 调试运行程序.....53
2.11 在目标板上调试程序.....55
2.12 固化系统和软件到目标板.....57
第 3 章 基于 NIOS II 处理器的 UC/OS II 操作系统设计实例.....58
第一步：建立 Quartus II 工程.....59
第二步：打开 Qsys 工具.....59
第三步：在 Qsys 系统中添加 NIOS II 处理器.....59
第四步：在 Qsys 中添加 UCOS 系统所需定时器.....60
第五步：在 Qsys 中添加 SDRAM.....60
第六步：在 Qsys 系统中添加 JTAG_UART.....62
第七步：在 Qsys 系统中添加 sysid.....62
第八步：连线和导出端口.....62
第九步：设定 sdram 基地址.....63
第十步：自动分配所有模块地址.....63
第十一步：设定 CPU 的复位向量地址和异常向量地址.....63
第十二步：保存设计.....63
第十三步：生成 Qsys 系统的 HDL 代码.....63
第十四步：在 Quartus II 系统中添加并例化 QSYS 系统.....64
第十五步：添加锁相环.....64
第十六步：添加 Qsys 系统到 Quartus II 工程中.....65
第十七步：完善 mysystem.v 模块.....65
第十八步：分析和综合；.....66
第十九步：分配引脚.....67
第二十步：全编译.....68
第二十一步：配置固件到 FPGA 芯片中.....68
第二十二步：打开 NIOS II EDS 软件.....68
第二十三步：建立 UC/OS II 模版工程.....68
第二十四步：编译软件工程.....70

第二十五步：下载并运行 70
第 4 章 Altera SOPC FrameBuffer 系统设计教程 72
第 5 章 RT-Thread 操作系统在 NIOS II 上的移植 87
RT-Thread 介绍 87
AC620 FPGA 开发板介绍 87
实验说明 88
操作步骤 88
第 7 章 给 NIOS II CPU 增加看门狗定时器并使用 100
第 8 章 给 NIOS II CPU 添加一颗澎湃的心——系统时钟的使用 107
NIOS II 开发注意点总结 116
elf 文件下载失败 117
CPU 运行一段时间后停止 121
程序运行不正确 122
解决 NIOS II 工程移动在磁盘上位置后 project 无法编译问题 126
6.1 更改 NIOS II Project 目录原因 126
6.2 更改 NIOS II Project 目录引发的问题 126
6.3 解决方案详解 127
6.4 解决方案步骤总结 132
sof 与 NIOS II 的 elf 固件合并 jic 得到文件 136
7.1 为什么需要将 Sof 与 elf 合并得到 jic 文件 136
7.2 本章示例介绍 136
7.3 详细转换步骤 137
Symbol 'NULL' could not be resolved 146
设置 NIOS II 软件工程头文件包含路径 147
EPCS 使用相关要点 148
使用非官方 EPCS 存储器固化 NIOS II 软件 149
拔掉下载器或者关闭 NIOS II EDS 软件，开发板上的系统即停止运行 150
问题原因 150
解决建议 150
编译 NIOS 软件工程提示” Permission denied” 150
问题原因 151
解决建议 151
设置 Eclipse 在编译(build）前自动保存源代码文件 151
切换 NIOS II CPU 的主内存后软中需要注意的几点设置 153

Altera SOPC 嵌入式系统设计教程

第 1 章　概述

SOPC(System On Programmable Chip，可编程的片上系统）是 Altera 公司提出来的一种灵活、高效的 SOC 解决方案。它将处理器、存储器、I/O 口、LVDS、CDR 等系统设计需要的功能模块集成到一个可编程器件上，构成一个可编程的片上系统。

1.1 SOPC 及其技术

现今，可将 SOPC 视为是基于 FPGA 解决方案的 SOC。与 ASIC 的 SOC 解决方案相比，SOPC 系统及其开发技术具有更多的特色，也有多种途径可构成 SOPC 方案。

1.1.1 基于 FPGA 嵌入 IP 硬核的 SOPC 系统

基于 FPGA 嵌入 IP 硬核的 SOPC 系统是指在 FPGA 中预先植入处理器。目前最常用的嵌入式处理器大多是采用含有 ARM32 位知识产权处理器核的器件。为了达到通用性，必须为常规的嵌入式处理器集成诸多通用和专用的接口，但这样无疑会增加芯片的成本和功耗。 如果将 ARM 或其他处理器核以硬核方式植入 FPGA 中，利用 FPGA 的可编程逻辑资源，按照系统功能需求来添加接口功能模块，既能实现目标系统功能，又能降低系统的成本和功耗。 这样就使得 FPGA 灵活的硬件设计与处理器的强大软件功能有机地结合在一起，高效地实现 SOPC 系统。

1.1.2 基于 FPGA 嵌入 IP 软核的 SOPC 系统

将 IP 硬核直接植入 FPGA 的解决方案存在以下不足：

(1)由于此类硬核多来自第三方公司，FPGA 厂商通常无法直接控制其知识产权费用，从而导致 FPGA 器件价格相对偏高。

(2)由于硬核是预先植入的，设计者无法根据实际需要改变处理器的结构，如总线规模、 接口方式、指令形式，更不可能将 FPGA 逻辑资源构成的硬件模块以指令的形式嵌入硬件加速模块（如 DSP)。

(3)无法根据实际设计需要在同一 FPGA 中集成多个处理器。

(4)无法根据需要裁剪处理器硬件资源以降低 FPGA 成本。

(5)只能在特定的 FGPA 中使用硬核嵌入式处理器。

如果利用软核处理器，就能有效地弥补上述不足。

目前最有代表性的软核处理器分别是 Altera 公司的 Nios II 核，以及 Xilinx 公司的 MicroBlaze 核。特别是 Nios II 核，能很好的解决上述五方面问题。

Altera 的 Nios II 核是用户可随意配置和构建的 32 位嵌入式处理器 IP 核，采用 Avalon 总线结构通信接口；包含由 First Silicon Solution(FS2)开发的基于 JTAG 的片内设备（OCI)内核（这为开发者提供了强大的调试功能)。在把 Nios II 植入 FPGA 前，用户可以根据设计要求，利用 Quartus II 和 Qsys，对 Nios II 及其外围设备进行构建，使该嵌入式系统在硬件结构、功能特点、资源占用等方面全面满足用户系统设计的要求。Nios II 核在同一 FPGA 中的植入数量没有限制，只要 FPGA 资源足够即可。

另外，在开发工具的完备性方面，Nios II 具有很大的优势。Altera 不仅提供了强大的 HAL 系统库支持，还提供了嵌入式操作系统和 TCP/IP 协议栈的支持。

就成本而言，由于 Nios II 是由 Altera 公司直接提供而非第三方厂商产品，故用户通常无须支付知识产权费用，Nios II 的使用费用仅仅是其占用的 FPGA 逻辑资源的费用。因此，用户选用的 FPGA 越便宜，则 Nios II 的使用费就越低。

特别值得一提的是，通过 Matlab 和 DSP Builder,用户可以为 Nios II 处理器设计各类硬件数字处理器，并以指令的形式加入 Nios II 的指令集。例如，用户可以根据设计项目的具体要求，随心所欲地构建自己的 DSP 处理器系统，而不必拘泥于其他 DSP 公司已上市的有限款式的 DSP 处理器。

指标	基于 ASIC 的 SOC	基于 FPGA 的软核 SOC
单片成本	低	较高
开发周期	长（>20 周）	短（<10 周）
开发成本	设计工程成本高 掩膜成本高 软件工具成本高	设计工程成本低 无掩膜成本 软件工具成本低
一次投片情况	一次投片成功率低，成本高，耗时长	可现场配置
集成技术	0.25um—14nm	0.25um—14nm

可重构性	不可重构	可重构

1.2 NiosII 软核处理器

1.2.1 Nios II 软核处理器简介

继第一代可配置嵌入式软核处理器 Nios 之后，2004 年 6 月，Altera 公司又推出了性能更好的 Nios II 嵌入式软核处理器。它与 2000 年上市的原产品 Nios 相比，最大处理性能提高了 3 倍，而 CPU 内核部分的面积最大可缩小 1/2。在 2013 年底，Altera 公司又推出了 NIOS II 嵌入式软核处理器 GEN2 版本，并将之前的 NIOS II 嵌入式软核处理器更名为 NIOS II Classic。GEN2 版本与 Classic 相比，有相似的尺寸和性能，用户只需要做很小的改动，就能够把 Classic 版本上的软件移植到 GEN2 版本上来。

Nios II 系列 32 位 RISC 嵌入式处理器具有超过 200 DMIP 的性能，在低成本 FPGA 中实现成本只有 35 美分。由于处理器是软核形式，具有很大的灵活性，可以在多种系统设置组合中进行选择，满足成本和功能要求。采用 Nios II 处理器进行设计，可以帮助用户将产品迅速推向市场，延长产品生命周期，防止出现处理器逐渐过时的情况。

Nios II 开发包含有一套通用外设和接口库，部分 IP 和接口如表 1.2 所列。

定时器/计数器	外接 SRAM 接口	CFI 接口 FLASH	SDRAM 接口	片内 RAM/ROM 接口
用户逻辑接口	FIFO 接口	并行 I/O（GPIO）	JTAG UART	系统 ID
EPCS 控制器	DMA	UART	SPI 接口	LCD 接口
外部三态桥	Avlaon ST 接口	Avalon 转 AHB 桥	AHB 转 Avalon 桥	SDI
PCI E	DDR SDRAM	CAN	RNG	USB
DDR2 SDRAM	DES	16550 UART	RAS	SHA-1
IIC	三速以太网 MAC	浮点单元	VIP	SGDMA

利用 Qsys 软件中的用户逻辑接口向导，用户还可以生成自己的定制外设，并将其集

成在 Nios II 处理器系统中。使用 Qsys，用户可以在 Altera FPGA 中，组合实现现有处理器无法达到的嵌入式处理器配置，得到用户所需的结果。

Nios II 系列嵌入式处理器使用 32 位的指令集结构（ISA)，建立在第一代 16 位 Nios 处理器的基础上，定位于广泛的嵌入式应用。Nios II Classic 处理器系列包括 3 种内核：快速的（Nios II/f)、经济的（Nios/e)和标准的（Nios II/s)内核。更新版的 Nios II GEN2 处理器系列则取消了标准型内核，保留了经济型和快速型。每一型号都针对价格和性能进行了优化。 使用 Altera 的 Quartus II 软件、Qsys 工具以及 Nios II 集成开发环境（IDE), 用户可以轻松地完成基于 Nios II 处理器的嵌入式系统开发。

表 1.3 列出了 Nios II 系统的 3 种内核及特性。

表 1.4 列出了 Nios II 处理器的特性。

内核	说明
NIOS II/f（快速）	性能最高，但占用的逻辑资源最多
NIOS II/e（经济）	占用的逻辑资源最少，但性能最低
NIOS II/s（标准）（GEN2 已不再支持）	平衡的性能和尺寸。

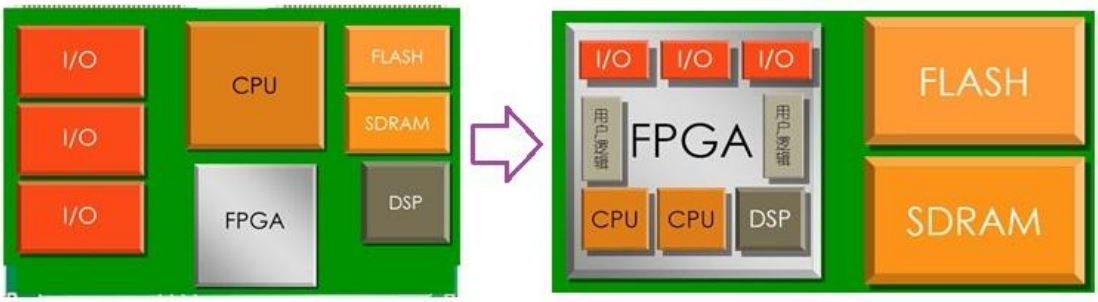
种类	特性	种类	特性
CPU 结构	32 位指令集	片内调试	基于边界扫描测试（JTAG）的调试逻辑，支持硬件断点，数据出发以及片内和片外的调试跟踪
	32 位数据位宽		
	32 个通用寄存器	定制指令	最多可达 256 个用户定义的 CPU 指令
	32 个外部中断源	软件开发工具	NIOS II 的集成开发环境（IDE）
	2GB 寻址空间		基于 GNU 的编译器
			硬件辅助的调试模块

1.2.2 可配置嵌入式软核处理器的优势

嵌入式处理器开发人员面对的一个最大挑战就是如何选择一个满足其应用要求的处理器。现在已有数百种嵌入式处理器，每种处理器都具备一组不同的外设、存储器、接口和性能特性，用户很难做出一个合理的选择：要么为了匹配实际应用所需的外设和

接口要求而不得不选择在某些性能上多余的处理器；要么为了保持成本的需求而达不到原先预计的理想方案。

采用 Nios II 处理器，用户将不会局限于预先制造的处理器技术，而是根据自己的要求定制处理器，按照需要选择合适的外设、存储器和接口。此外，用户还可以轻松集成自己专有的功能（如 DSP，用户逻辑），创建一款“完美”的处理器，如图 1.1 所示，使用户的设计具有独特的 竞争优势。



用户所需要的处理器，应该能够满足当前和今后的设计性能需求。由于今后发展具有不确定性，因此，设计人员必须能够更改其设计，加入多个 CPU，定制指令集及硬件加速器，以达到新的性能目标，而 Nios II 处理器能满足以上要求。

采用 Nios II 处理器，用户可以根据需要设置功能，在低成本 Altera 器件中实现。在单个 FPGA 中实现处理器、外设、存储器和 I/O 接口功能，可以降低用户的系统总体成本。

无论是外设、存储器接口、性能特性，还是成本。这些优势的体现都借助于在 Altera 的 FPGA 上创建一个定制的片上系统，或者更精确地说，创建一个可编程单芯片系统。

1.提供合理的性能组合

使用 Altera Nios 处理器和 FPGA，用户可以创建一个在处理器、外设、存储器和 1/0 接口方面的完美方案。选择如下：

- 3 种处理器内核——Nios II 开发人员可以选择一种或以下任意 3 种内核的组合：具备高性能的快速的内核（Nios II/f），具备低成本的经济的内核（Nios II/e），性能和尺寸折中的标准的内核（NiosII/s）。
- 超过 60 种 Qsys 配备的内核——用户可以创建一组适合自己应用的外设、存储器和 I/O 接口。

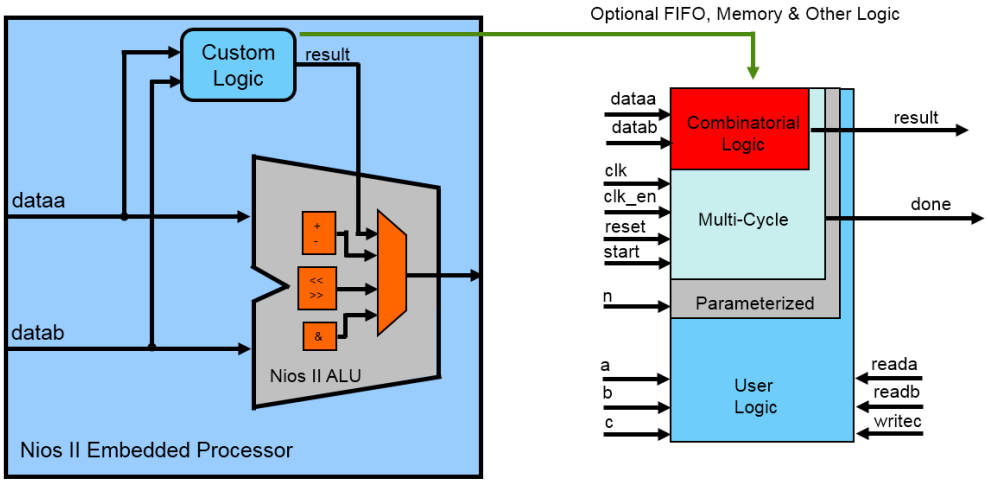
- 无限制的 DMA 通道组合——直接存储器存取（DMA)可以连接到任何外设，从而提高系统的性能。
- 可配置的硬件及软件调试特性——软件开发人员具有多个调试选择，包括基本的 JTAG 的运行控制（运行、停止、单步、存储器等）、硬件断点、数据触发、片内和片外跟踪、嵌入式逻辑分析仪。这些强大的工具可以在开发阶段使用，一旦调试通过后就可以去掉。

2.提升系统的性能

在嵌入式开发中，设计人员通常都会选择一个比实际所需的性能要高的处理器（意味着更高的成本），从而为设计保留一个性能上的安全余量。Nios II 系统的性能是可以根据应用来裁剪的，相比固定的处理器，在较低的时钟速率下，具备更高的性能。Nios II 的特性可以在以下几个方面来提升系统的性能：

- 多 CPU 内核一 Nios II 开发者可以选择最快的内核（Nios II/f)以获得高性能，还可以通过添加多个处理器来获得所需的系统性能。
- FPGA 系列—Nios II 处理器可以工作在 Altera 推出的所有 Cyclone 系列、MAX10 系列、Arria 系列和 Stratix 系列的 FPGA 上。尤其是在 Stratix V 器件上，Nios II/f 内核超过 350 DMIPS 的性能，仅占用 1800 个逻辑单元（LE)。可以说用户几乎是免费得到一个 200 DMIPS 性能的处理器。
- 多处理器系统——开发人员可以在 FPGA 内部实现多个处理器内核。通过将多个 Nios II/f 内核集成到单个器件内，可以获得极高的性能而不用重新设计印刷电路板。Nios II 的 IDE 支持这种多处理器在单一 FPGA 上的开发，或多个 FPGA 共享一条 JTAG 链。
- 定制指令—定制指令是一种扩展处理器指令的方法，最多可以定制 256 个用户指令 (见图 1.2)。定制指令是处理器处理复杂的算术运算和加速逻辑的最佳途径。例如， 将一个在 64K 字的缓冲区实现的循环冗余码校验（CRC)的逻辑块作为一个定制的指令，要比用软件实现快 27 倍。
- 硬件加速—通过将专用的硬件加速器添加到 FPGA 中作为 CPU 的协处理器，CPU 就可以并发地处理大块的数据。例如上面提到的 CRC 例子，通过硬件加速器处理一个 64K 字的缓冲区，比用软件快 530 倍。Qsys 设计工具中包含一个向导，

用户可以用此向导来将加速逻辑和 DMA 通道添加到系统中。



3.降低系统成本

嵌入式设计人员总是坚持不懈地寻找降低系统成本的方法。然而，选择一款处理器，总是存在性能、特性与成本的冲突，最终结果总是以增加系统成本为代价。利用 Nios II 处理器可以通过以下途径来降低成本：

- 更大规模的系统集成—将一个或更多的 Nios II 处理器组合，选择合适的一组外设、 存储器、I/O 接口，这些方法可降低电路板的成本、复杂程度以及功耗。
- 优化 FPGA/CPU 的选择—Cyclone 系列的 FPGA 提供可行的、低成本的嵌入式方案。经济型的内核（Nios II/e)只占用不到 35 美分的 Cyclone 器件资源，仅占用 600 个逻辑单元，给其他片外的器件保留了更多的逻辑资源。Nios II/e 内核定位于低成本、需要低处理性能的系统。占用很少的逻辑资源还使得在单个的 FPGA 芯片上嵌入多个处理器成为可能。
- 更好的库存管理—嵌入式系统通常包含了来自多个生产商的多种处理器，以应付多变的系统任务。当一种器件多余而另一种短缺的时候，管理这些处理器的库存便是个问题。但是使用标准化的 Nios II 软核处理器，库存的管理将会大大简化，因为通过将处理器实现在标准的 FPGA 器件上，减少了处理器种类的需求。

4.延长产品的生命周期

开发人员希望快速将产品推向市场并保持一个较长的产品生命周期，避免更新换代。基于 Nios II 的系统在以下几个方面可以帮助用户实现此目标：

- 缩短产品的上市时间—FPGA 可编程的特性使其具有最快的产品上市速度。许多的设计通过简单的修改都可以被快速地实现到 FPGA 设计上。Nios II 系统的灵活性和快速上市的特性源于 Altera 提供完整的开发套件、众多的参考设计、强大的硬件开发工具（Qsys)和软件开发工具（Nios II IDE)。用户可以借助 Nios II 开发套件所带的设计，在几个小时内就完成自己的设计原型。由于将 Nios II 处理器放置于 FPGA 内部就可以验证外部的存储器和 I/O 组件，电路板设计速度显著加快。
- 建立有竞争性的优势—维持一个基于通用硬件平台的产品的竞争优势是非常困难的。而包含一个或多个 Nios II 处理器，具备硬件加速、定制指令、定制的可裁剪的外设等的 SOPC 系统，具备了竞争的优势。
- 延长了产品的生存时间—使用 Nios II 处理器的 SOPC 产品带来的一个独特优势就是能够对硬件进行升级。即使产品已经交付给客户，仍可以定期升级。这些特性可以解决很多问题：
  - 延长产品的生存时间，随着时间的增加，可以不断将新的特性添加到硬件中。
  - 减小由于标准的制定和改变而带来的硬件上的风险。
  - 简化了硬件缺陷的修复和排除。
  - 避免处理器的过时。嵌入式处理器供应商通常提供一个很宽的配置选择范围以适应不同的客户群。不可避免的是，某个或多个处理器有可能会因为生产计划等原因而停止供应或很难寻找。Nios II 设计人员拥有在 Altera FPGA 上使用和配置基于 Nios II 设计的永久许可。一个基于 Nios II 的设计可以容易地被重新移植到新系列的 FPGA 器件中，从而保护了对应用软件的投资。
- 在产品产量增加的情况下降低成本—旦一个 FPGA 的设计被选定，并且打算进行大批量的生产，可以选择将它移植到 Altera 的 HardCopy(—种结构化的 ASIC 系列） 中，从而降低了成本并提升性能。Altera 还可以提供 Nios II 处理器的 ASIC 制造许可，用户可以将包含 Nios II 处理器、外设、Avalon 交换式总线的设

计移植到基于单元的 ASIC 中。

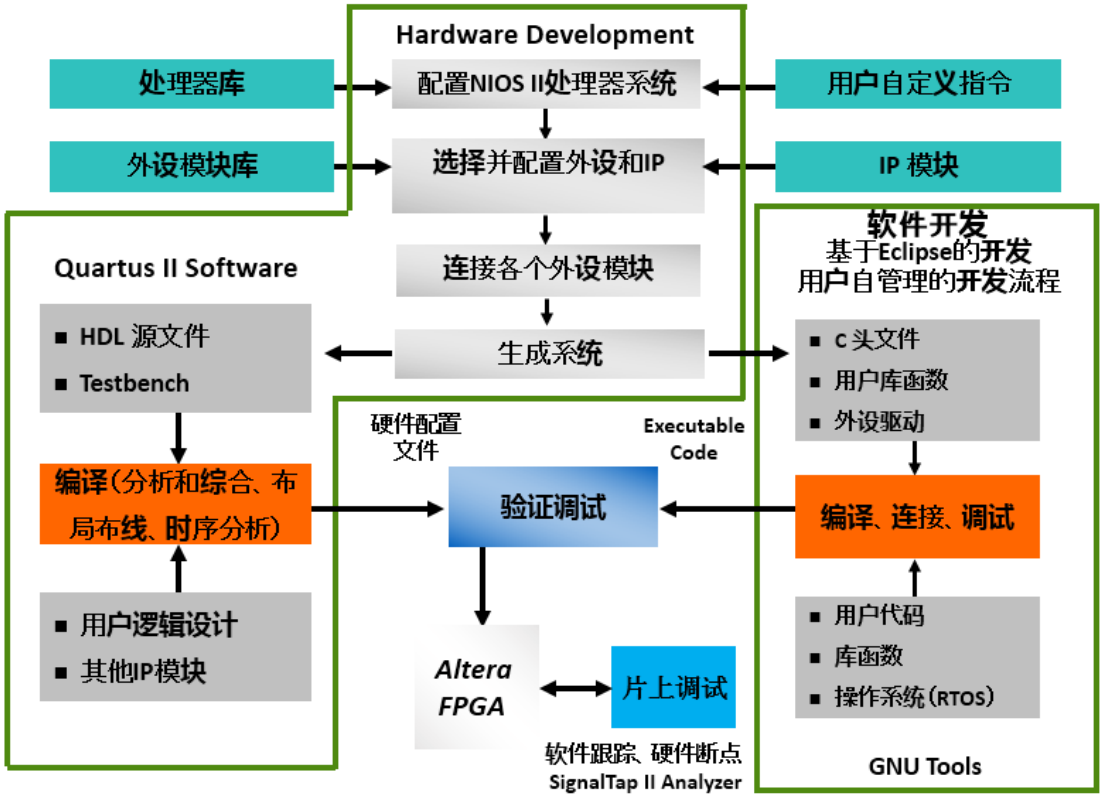
第 2 章 SOPC 开发流程简介

2.1 SOPC 开发流程

SOPC 设计包括以 Nios II 软核处理器为核心的嵌入式系统的硬件配置、硬件设计、硬件仿真、IDE 环境的软件设计、软件调试等。SOPC 系统设计的基本软件工具包括 Quartus II，用于完成 Nios II 系统的分析综合、硬件优化、适配、配置文件编程下载以及硬件系统测试等；Qsys,它是 Nios II 软核处理器的开发包，用于实现 Nios II 系统配置、生成以及与 Nios II 系统相关的监控和软件调试平台的生成；ModelSim，用于对 Qsys 生成的 NiosII 的 HDL 描述语言程序进行系统功能仿真；Matlab/DSP Builder,用于生成 Nios II 系统的硬件加速器，进而为 Nios II 系统定制新的指令；Nios II IDE,用于完成基于 Nios II 系统的软件开发和调试，并可借助其自带的 Flash 编程器完成对 Flash 以及 EPCS 的编程操作，此外 Nios II IDE 还包括一个指令集成模拟器、Microc/OS - II 实时操作系统、文件系统以及小型 TCP/IP 协议栈。

SOPC 的开发过程中要使用到 Quartus II、Qsys 以及 Nios II IDE，三者之间关系如图 2. 1 所示。

SOPC 的开发流程通常包括 2 个方面：基于 Quartus II、Qsys 的硬件设计，基于 Nios II IDE 的软件设计。对于比较简单的 Nios II 系统，一个人便可执行所有设计；对于比较复杂的系统，硬件和软件设计可以分开进行。



2.1.1 硬件开发

硬件开发使用 Quartus II 和 Qsys。硬件设计工作如下：

(1)用 Qsys 软件从 Nios II 处理器内核和 Nios II 开发套件提供的外设列表中选取合适的 CPU、存储器以及各外围器件（如片内存储器、PIO、定时器、UART、片外存储器接口等），并定制和配置它们的功能；分配外设地址及中断号；设定复位地址；最后生成系统。用户也可以添加用户自身定制指令逻辑到 Nios II 内核以加速 CPU 性能，或添加用户外设以减轻 CPU 的任务。

(2)使用 Qsys 生成 Nios II 系统后，将其集成到整个 Quartus II 工程中。可以在 Quartus II 工程中加入 Nios II 系统以外的逻辑，大多数的 SOPC 设计都包括 NiosII 系统以外的逻辑，这也是 SOPC 系统的优势所在。用户可以集成自身定制的硬件模块到 SOPC 设计，或集成从 Altera 或第 3 方 IP 供应商中得到的其它现成知识产权设计模块。

(3)使用 Quartus II 软件来选取具体的 Altera FPGA 器件型号；然后为 Nios II 系统上的各 I/O 口分配引脚，另外还要根据要求进行硬件编译选项或时序约束的设置；最后编译 Quartus II 工程，在编译过程中 Quartus II 将对 Qsys 生成系统的 HDL 设计文件进行布局布线，从 HDL 源文件综合生成一个适合目标器件的网表，生成 FPGA 配置文件(.sof)。

(4)使用 Quartus II 编程器和 Altera 下载电缆（如 USB Blaster)，将配置文件（用户定制的 Nios II 处理器系统的硬件设计)下载到目标板上。当校验完当前硬件设计后，可将新的配置文件下载到目标板上的非易失存储器里（如 EPCS 器件）。下载完硬件配置文件后，软件开发者就可以将此目标板作为软件开发的初期硬件平台进行软件功能的开发验证了。

2.1.2 软件开发

软件开发使用 Nios II IDE。它是一个基于 Eclipse IDE 构架的集成开发环境，包括：

- GNU 开发工具(标准 GCC 编译器、连接器、汇编器和 makefile 工具等)；
- 基于 GDB 的调试器,包括软件仿真和硬件调试；
- 提供用户一个硬件抽象层 HAL（Hardware Abstraction Layer）；
- 提供嵌入式操作系统 MicroC/OS-II 和 LwTCP/IP 协议栈的支持；
- 提供帮助用户快速入门的软件模板；
- 提供 Flash 下载支持（Flash Programmer 和 Quartus II Programmer)

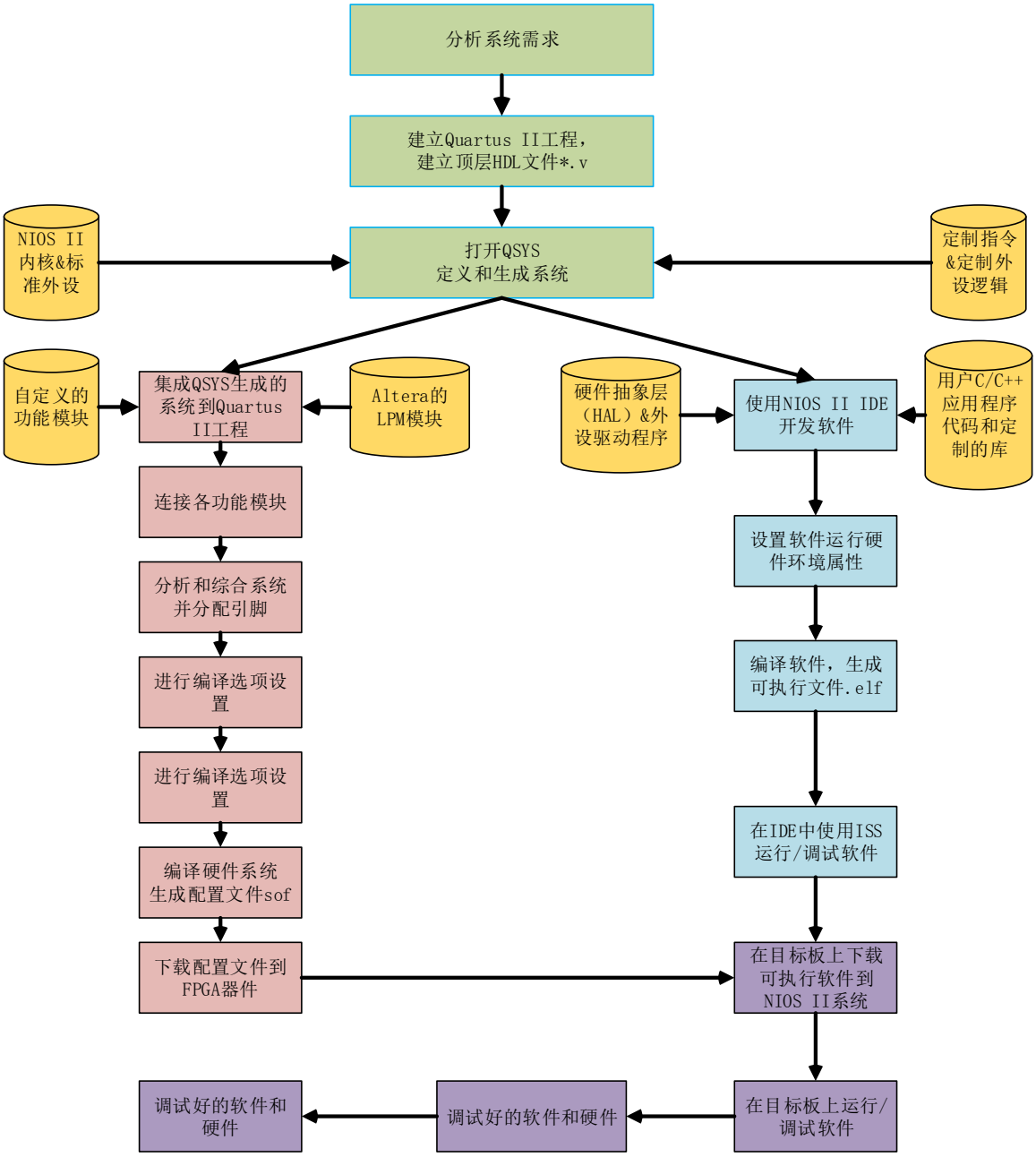
熟悉 Linux 下 GNU 开发工具的开发者基本上可以毫无困难地使用 Nios II IDE。

使用 Nios II IDE,可完成 Nios II 处理器系统的所有软件开发任务。使用 Qsys 生成系统后，可以直接使用 Nios II IDE 开始设计 C/C+ +应用程序代码。Altera 提供外设驱动程序和硬件抽象层（HAL)，使用户能够快速编写与低级硬件细节无关的 Nios II 程序。除了应用代码，用户还可以在 Nios II IDE 工程中设计和使用定制库。

如果没有软件开发的目标板，可以通过 Nios II 指令集仿真器（ISS)运行和调试代码；ISS 可仿真处理器、存储器和 stdin/stdout/stderr 流，使用户可以检验程序流和算法正确性。一旦有一个目标板，用户就可以使用下载电缆（如 USB Blaster)下载软件到目标板进行调试/ 运行。

2.1.3 SOPC 基本开发流程简介

在开始进行 SOPC 设计前，让我们先了解一下 SOPC 系统的基本开发流程。图 2.2 所示为基于 Nios II 系统的 SOPC 开发流程。



参照图 2.2 说明 SOPC 基本的开发流程如下：

1) 在进行 SOPC 开发之前，首先必须确定系统的需求，如应用系统需要的计算性能、带宽和吞吐量、接口类型以及是否需要多线程的软件等。完整的基于 Nios II 的 SOPC 系统是一个软硬件复合的系统。在开发时，可分为硬件和软件两个部分，但在实际设计过程中，有时候所需要的功能既可以用软件也可以用硬件来实现。具体采用什么方式，要根据系统设计要求来权衡。一般来说，用软件实现时，在设计上容易修改或者增删，查错也比较容易，又几乎不增加占用的逻辑资源，但执行速度较慢。所以，在设计规划时，在满足性能的前提下，优先考虑软件实现。

2) 每个开发过程开始时都应建立一个工程，Quartus II 是以工程的方式对设计过程进行管理的。在工程中建立顶层模块文件.bdf，其将整个工程的各个模块包容在里面，编译的时候就将这些模块整合在一起。这个顶层文件相当于传统电路设计中的电路板（PCB)。

3) 在 Qsys 中添加需要的功能模块(Nios II 及其标准外设模块)，完成后生成一个系统模块。如果需要，用户可以定制指令和外设逻辑。由 Qsys 最终生成的系统模块相当于传统的单片机或 ARM。

4) 如果需要，可以使用 Altera 公司提供的 LPM 功能模块。Quartus II 包含了大量常用功能模块（如计数器、累加器、比较器、译码器等），使用这些模块能大大加快开发的速度。一般来说，一个设计只有很少部分的模块需要自己从零设计。LPM 功能模块相当于传统设计中除处理器以外的逻辑芯片（如 74 系列）。

5) 如果需要，建立自己的功能模块。当设计中现有的模块功能不能满足设计要求时，要求用户自己设计。可以用硬件描述语言，也可以用原理图的输入方法来设计自己的功能模块。 设计自己的功能模块时，可以作为一个独立的工程来设计，并生成模块符号（Symbol)，然后将源文件（实现该模块的原理图或 HDL 文件）拷贝到顶层模块所在的工程目录下，这样在顶层模块中就可以使用这个自定义模块的符号。使用自定义功能模块，相当于在传统设计中，市面上买不到某种芯片而去定做芯片。

6) 在顶层模块中，分别将 Qsys 生成的系统模块、LPM 功能模块以及用户自定义功能模块添加到顶层模块中，然后将各个功能模块用连线连起来组成系统功能原理图。这个过程类似传统电路设计中，将所有要使用到的芯片焊接到电路板上，然后通过 PCB 上的连线将各个芯片连接起来，组成电路系统。

7) 为系统功能原理图选择芯片载体并为各个输入/输出信号分配芯片的引脚。

8) 设置编译选项，从而让编译器按照用户设定来进行编译。

9) 编译系统，生成硬件系统的配置文件.sof 和.pof。编译系统是一个非常复杂的过程，包括优化逻辑的组合、综合逻辑、适配 FPGA、布线以及时序分析等步骤。

10) 打开 Nios II IDE 软件进行用户程序开发。当然这 1 步也可以在 Qsys 生成系统模块后立即进行。这个过程与传统嵌入式系统的软件开发类似， 唯一的不同在于，软件所运行的嵌入式系统是自己定制的、裁剪过的，因此可能受硬件的局限会小一些。

11) 对用户软件进行编译，生成可执行文件.elf。

12) 可以在 IDE 的指令集仿真器(ISS)上仿真软件和运行/调试软件。

13) 将配置文件.sof 下载到 FPGA,将可执行文件.elf 下载到 RAM(外部或内部)。

14) 在目标板上调试软件,直到硬件和软件设计都达到设计要求。

15) 利用 IDE 的编程工具将配置文件烧写到 FPGA 的配置芯片或 Flash。将可执行文件 .elf 编程到 Flash。

2.2 简单 SOPC 实例开发任务及步骤

【任务】建立一个基于 NIOS II 处理器系统来控制一个 LED 灯闪烁。
【步骤】根据 2.1.3 小节，本节指导读者完成 SOPC 实例开发流程，主要步骤有：

(1) 分析系统需求；
(2) 启动 Quartus II 软件和新建实例设计工程；
(3) 启动新的 Qsys 系统；
(4) 在 Qsys 工具中定义并生成系统；
(5) 集成 Qsys 系统到 Quartus II 工程；
(6) 分析和综合系统并分配引脚；
(7) 设置编译选项并编译硬件系统；
(8) 下载硬件设计到目标 FPGA；
(9) 使用 NIOS II Software Build Tools For Eclipse 开发用户程序；
(10) 调试用户程序；
(11) 运行程序；

2.3 分析系统需求

每一个 SOPC 系统开发之前，都应该仔细分析系统需求。例如：对所设计系统的运行性能 有什么要求？系统要处理的带宽有多大？应用要求哪种类型的接口？应用要求多线程的软件吗？基于这些问题，用户可确定具体的系统要求，例如：
使用哪种 Nios II 处理器内核：比较小的还是比较快的？

- 设计要求什么外设及其数量是多少？
- 功耗有何要求？
- 使用哪种实时操作系统（RTOS)?
- 硬件加速逻辑在哪些方面可大大提高系统性能？例如：增加一个 DMA 外设能释放 CPU 在进行数据复制时所占用的资源吗？定制的指令能取代 DSP 算法吗？

本章实例的开发任务是：

- 展示可用于控制 LED 闪烁的简单 Nios II 处理器系统。
- 利用最通用和有效的方法来建立实际的定制 Nios II 系统。
- 整个系统仅使用片内资源，且不依赖于目标板。
- 析任务可以知道 FPGA 内硬件系统组成应包含：
- 带有 2 KB 指令高速缓存的 Nios II/s 处理器核；
- 4KB 片内 ROM 存储器(Onchip_ROM)用于存储程序代码以及程序运行空间；

- 4 KB 片内 RAM 存储器（Onchip_RAM)用于变董存储（R/W 数据）、Heap、stack;
- 1 位输出 I/O (PIO)引脚来控制 LED;
- 系统识别外围设备（SystemID)(关于 SystemID 的功能见 4.1.2 小节）。

说明：由于控制 LED 灯闪烁的用户程序代码很小，所以可将代码固化在片内 ROM 来执行，变量、堆栈 等空间使用片内 RAM,而不使用任何片外存储器。使用片内存储器能获得非常高的执行性能。

整个系统的框图如图 2. 3 所示。

图中，其他逻辑与 NiosII 系统一样可存在于 FPGA 中。事实上，大多数带有 Nios II 系统的 FPGA 设计也包括其他逻辑。Nios II 系统可与其他的片内逻辑相互作用，这取决于整个系统的需要。为了简单说明，本章的实例设计在 FPGA 内不包括其他逻辑。

假定读者具有以下的软件：

- Altera Quartus II 软件版本 13.0 或更高的版本——软件必须被安装在符合 Quartus II 最低要求的 Windows 计算机上。
- Nios II 开发套件版本 13. 0 或更高的版本。

无论是否拥有一块开发板，读者都可以按照本章所述来进行实例设计。如果没有目标板，

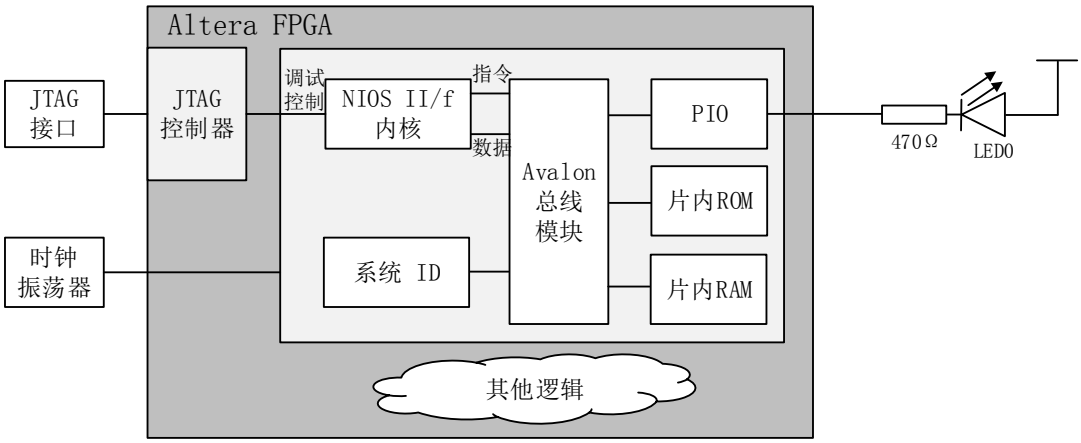


图 2.3 控制 LED 闪烁的系统框图

读者仍然可以使用本章所述内容，但不能目睹实例设计在硬件上运行。取而代之，用户将可以 在 Nios II 指令集仿真器（ISS)上仿真软件运行。如果读者拥有开发板以及 USB Blaster 下载电缆，用户将能目睹实例设计在硬件上运行及 LED 灯的闪烁。

2.4 使用 Quartus II 建立工程

每个开发过程开始时都应建立一个 Quartus II 工程，Quartus II 以工程的方式对设计过程进行管理，Quartus II 工程包括创建 FPGA 配置文件需要的所有设置和设计文件。

2.4.1 打开 Quartus II 软件并建立工程

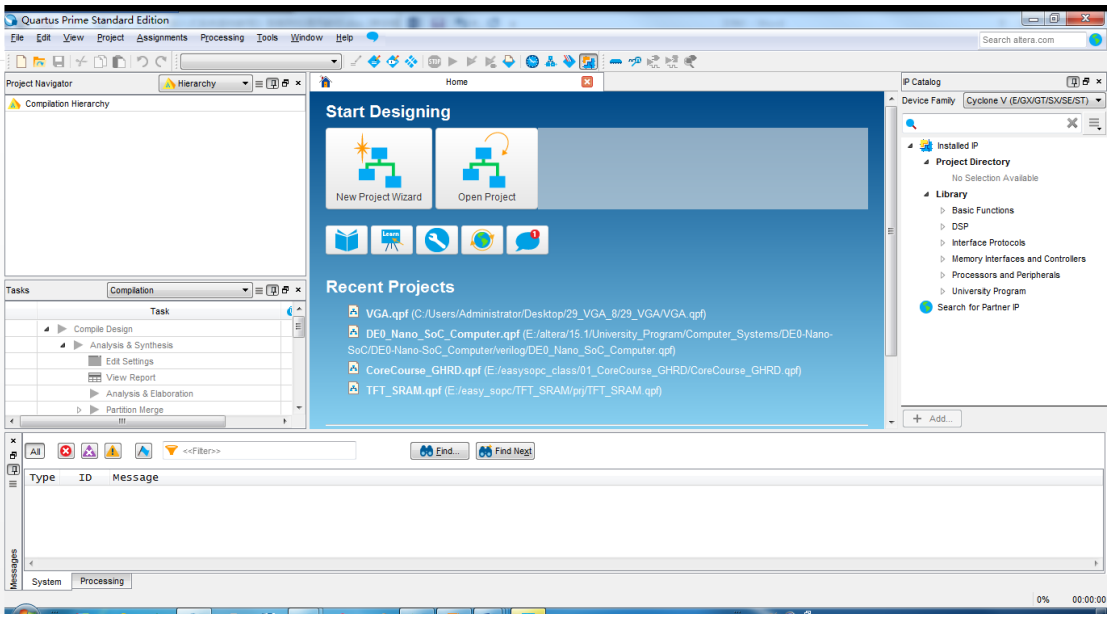
双击 Quartus II 图标打开 Quartus II 15.1 软件，如图 2.4 所示。

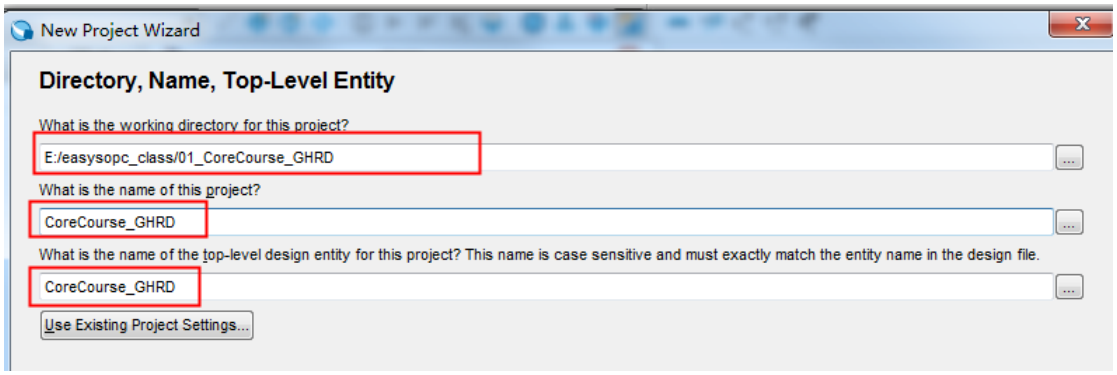
选择【file】【New】【New Quartus Prime Project】来新建一项工程。新建工程想到说明对话框如图 2.5 所示。

在新建工程向导说明对话框中可以了解在新建工程的过程中我们要完成哪些工作，这些

工作包括：

- 指定项目目录、名称和顶层实体；
- 指定项目设计文件；
- 指定该设计的 Altera 器件系列；
- 指定用于该项目的其他 EDA 工具，如仿真使用的 Modelsim 工具
- 项目信息报告。





任何一项设计都是一项工程(Project),必须首先为此工程建立一个放置与此工程相关的所有文件的文件夹，此文件夹将被 Quartus II 默认为工作库（Work Library)。通常，不同的设计项目最好放在不同的文件夹中，而同一工程的所有文件都必须放在同一文件夹中。

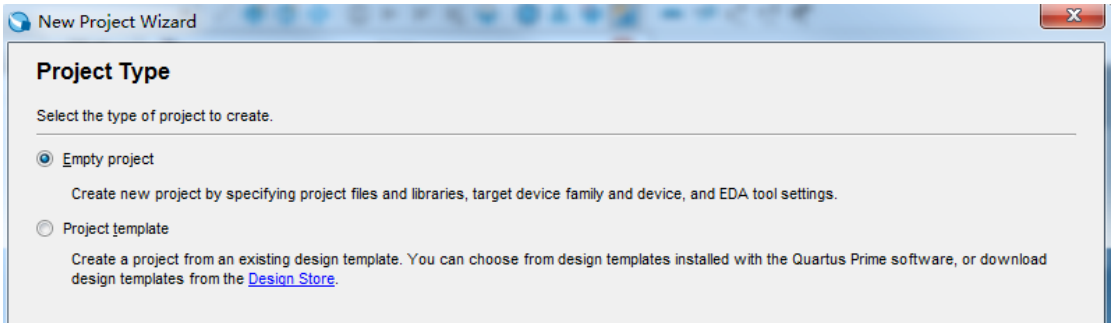
不要将文件夹设在计算机已有的安装目录中，更不要将工程文件直接放在安装目录中。文件夹所在路径名和文件夹名中不能用中文，不能用空格，不能用括号（），可用下划线“_”。

图 2.6 第一栏用于指定工程所在的工作库文件夹；第二栏用于指定工程名，工程名可以取 任何名字，也可以直接用顶层文件的实体名作为工程名（建议使用）. 第三栏用于指定顶层文件 的实体名。本例工程的路径为 E:\easysopc_class\01_CoreCourse_GHRD 文件夹，工程名与顶层文件的实体名同名为 CoreCourse_GHRD。

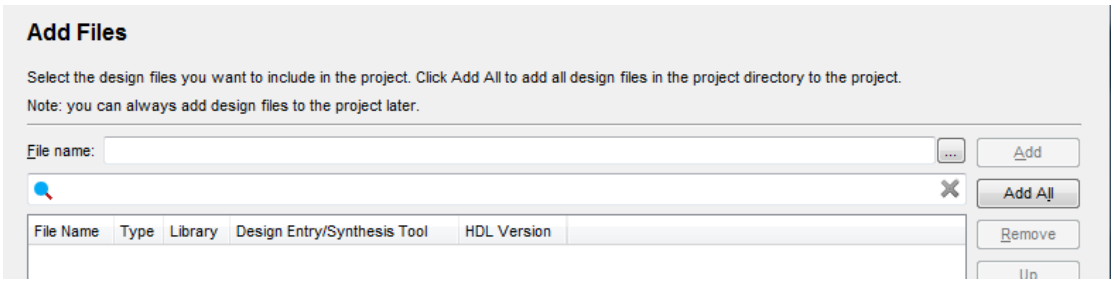
这里解释下 CoreCourse_GHRD 的意思，CoreCourse 是芯航线的意思，GHRD 叫做黄金设计顶层，GHRD 是 Altera 首先提出的一个基于 Qsys 的 HPS 硬件参考设计工程，一般由板商提供的最基本的 HPS 硬件工程，开发人员可以根据要求在此基础上添加或者删除 FPGA 端 IP

器件，也可以使能或者失能 HPS 的部分外设，配置 HPS DDR3 的时钟参数。这里我们将这个概念引入到我们的开发板中，作为我们板子的一个黄金设计顶层。当然，这一个实验中，这个黄金设计顶层只是一个框架，并没有晚上，我们将在后续的实验中逐步完善这个工程，当我们所有实验结束后，整个工程将是一个复杂且完整的，具有参考和实用价值的设计工程，成为真正意义上的黄金设计顶层。

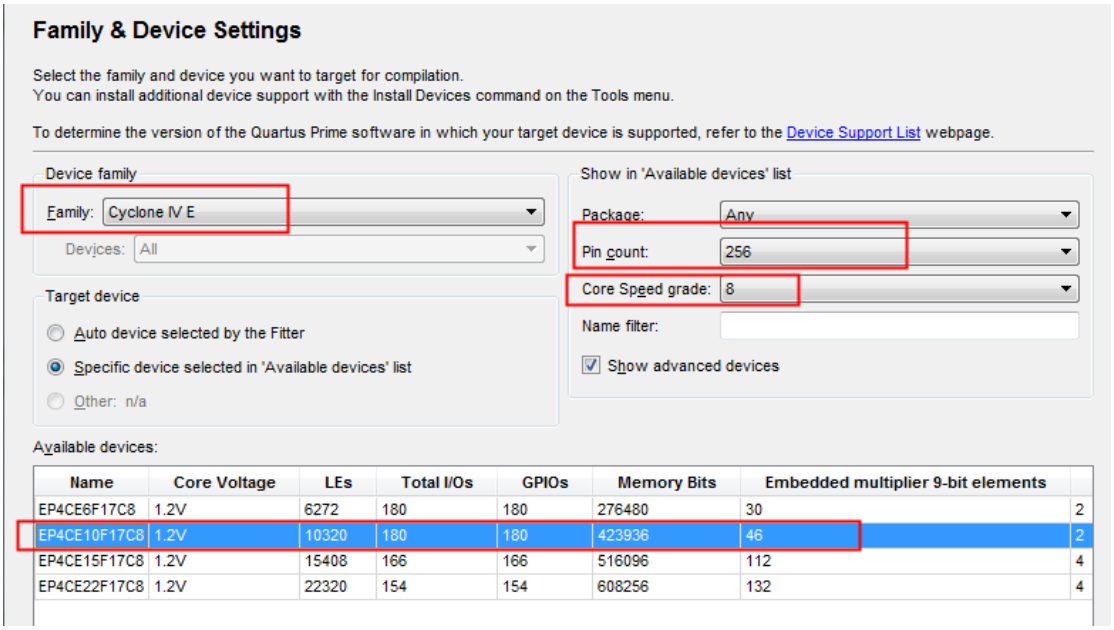
(4)单击 Next > 进入图 2.7 所示对话框。



这里为选择创建工程类型，可选项包括创建空工程（Empty Project）和使用工程模版（Project Template），由于工程模版需要在 Altera 官方网站下载，而且不一定适合我们，因此这里我们直接选择空工程即可。点击 Next > 进入下一步。



由于是新建工程，暂无输入文件，所以直接单击 Next >，进入图 2.8 所示对话框。要根据实际使用的 FPGA 来选择目标器件，这里指定为 Cyclone IV 系列的 EP4CE10F17C8。

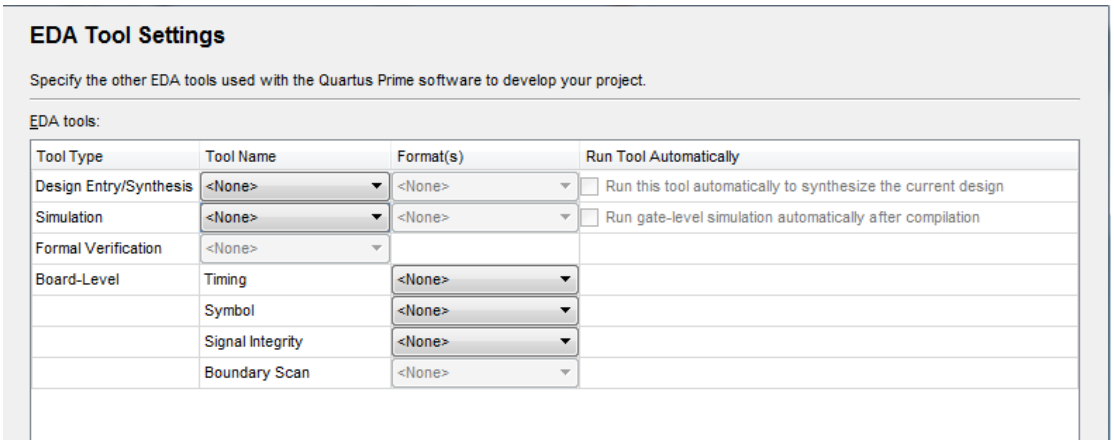


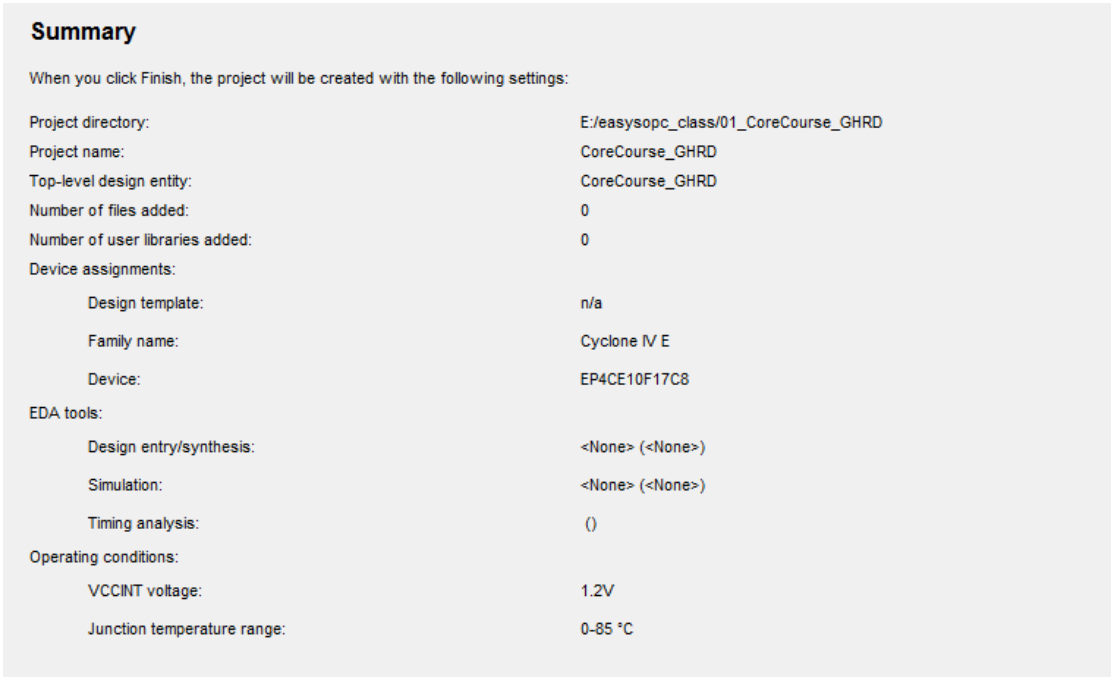
在图 2. 8 右边的过滤器选项区域中，设计者可以通过指定封装、引脚数以及器件速度等级来加快器件查找的速度。

指定完器件后，单击 Next > 进入图 2. 9 所示对话框，选择 EDA 工具。

本章所述实例是利用 Quartus II 的集成环境进行开发，不使用任何 EDA 工具，基于 Verilog 的数字逻辑教程中常用的设置仿真工具为 modelsim（altera），由于本实验暂不涉及到用户逻辑的开发，无需使用 modelsim 仿真，因此这里 不作任何设置和改动。

(7) 单击图 2. 9 中 Next > 进入图 2. 10 所示对话框。从图 2. 10 中，设计者可以看到工程文件配置信息报告。单击 Next > 按钮（图中未画出），完成新建工程的建立。





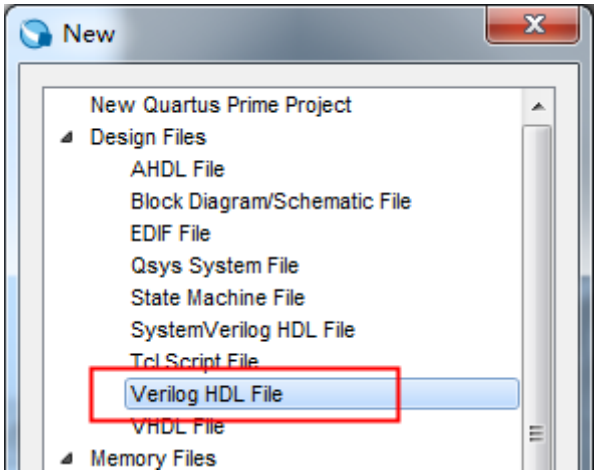
当然，工程建立完成后，也是可以根据设计中的实际情况，通过选择【Assignments】【Settings】来对工程进行重新设置。

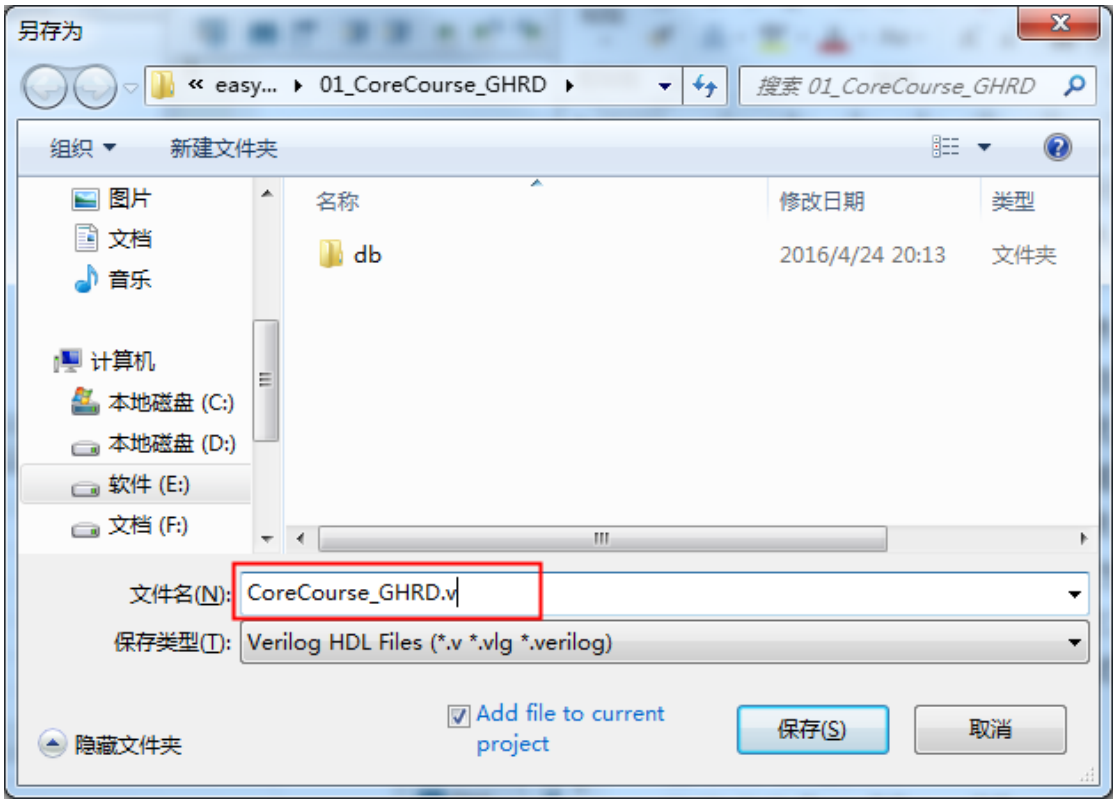
2.4.2 建立顶层模块

顶层模块用于将整个工程的各个模块包含在里面，Quartus II 编译时将这些模块整合在一起。顶层文件相当于传统电路设计中的电路板（PCB），用于将各种功能的芯片焊接在上面。

顶层文件的实现方式有两种，第一种是比较传统的使用原理图文件（.bdf），将所有的模块生成原理图符号，类似于一个一个的集成电路芯片，然后使用画线的方式将各个模块对应的端口连接起来，另一种是世界使用 Verilog 或者 VHDL 语言进行例化，将每个模块在一个 Verilog 文件中例化并将相关的信号名为相同的名字以实现连接。这两种方式各有优缺点。原理图方式进行顶层工程设计的优势是模块间关系一看便知，逻辑清晰，但是缺点是不易维护，尤其是在需要例化的模块越来越多的时候，这种方式无论是从实现效率和可移植性上，都有非常大的局限性，因此近些年来，使用的人已经越来越少。HDL 文件直接例化的方式，虽然在逻辑关系上看着不如原理图方式清晰，但是却拥有无可匹敌的可移植行和便捷的修改特性，因此近些年来，越来越多的工程师开始选择使用 HDL 文件来设计工程顶层。这里，我们也强烈推荐大家使用 HDL 的方式来设计顶层文件，尽量不要使用原理图的方式。

依次点击【File】【New】【Verilog HDL File】来创建一个新的 Verilog 文件，并点击【File】【Save As】将文件保存为 CoreCourse_GHRD.v





对话框，如图 2. 13 所示。

在图 2. 13 中接受默认的文件名，然后点击保存即可。至此，完成了顶层模块的建立。在 2.6.1 小节中将讲述怎样将 QSYS 生成的模块加入到顶层模块中。

2.5 使用 Qsys 创建 Nios II 系统

2.5.1 Qsys 简介

Qsys 包含在 Quartus II 软件中，它为建立 SOPC 设计提供了图形化环境，是早期 Quartus II 软件中 SOPC Builder 版本的升级，在保持基本功能不变的前提下，增加了系统建设的灵活性。 SOPC 由 CPU、存储器接口、标准外设和用户自定义的外设等组件组成。Qsys 允许选择和自定义系统模块的各个组件和接口。利用 Qsys，用户可以很方便地将处理 器、存储器和其它外设模块连接起来，形成一个完整的系统。Qsys 中已经包含了 Nios II 处理器以及一些常用的外设 IP 模块，用户也可以设计自己的外设 IP。用户在 Qsys 中定义 Nios II 系统的硬件特性，例如使用哪个 Nios II 内核，系统中包括什么外设。 Qsys 不定义软件操作，例如存储器中哪里存储指令或哪里发送 stderr 字符流，这些 工作在 Nios II HDE 中完成，见 2. 9. 2 小节。

Avalon 总线和外设的配置是在 Qsys 的图形用户界面（GUI)中指定的。用户 在 GUI 界面中指定各种参数和选项，这些参数和选项会存入一个系统 qsys 文件。qsys 文件是一个文本文件，它完整地定义了以下内容：

(1) Avalon 总线模块结构与功能参数；

(2) 每个外设结构与功能参数；

(3) 每个外设的主/从角色；
(4) 每个外设提供的端口信号；
(5) 每个可被多个主端口访问的从端口的仲裁机制。

从内部来看，Qsys 包含两个主要部分：图形用户界面（GUI)和系统生成程序。 图形用户界面提供管理 IP 模块、配置系统和报告错误等功能。用户通过图形用户界面设计系统时，所有的设置都保存在一个以系统命名的 qsys 文件中，所以图形用户界面实际上就是 qsys 文件的专用编辑器。用户通过图形用户界面完成设计之后，单击 Generate 将启动系统生成程序。系统生成程序完成大量的功能，创建了几乎所有的 Qsys 输出文件（HDL 逻 辑文件、C 程序的头文件和库文件、模拟仿真文件等）。

qsys 文件是图形用户界面和系统生成程序之间的唯一交互渠道。对大部分用户来说，仅仅了解 qsys 文件是由图形用户界面产生，并且系统生成程序要读取 qsys 文件的内容就足够了。高级用户可以使用文本编辑器来修改 qsys 文件编辑自己的设计，而不是使用图形用户界面。qsys 文件传递给 HDL 生成器，用来创建系统模块实际的寄存器传输级（RTL)描述。

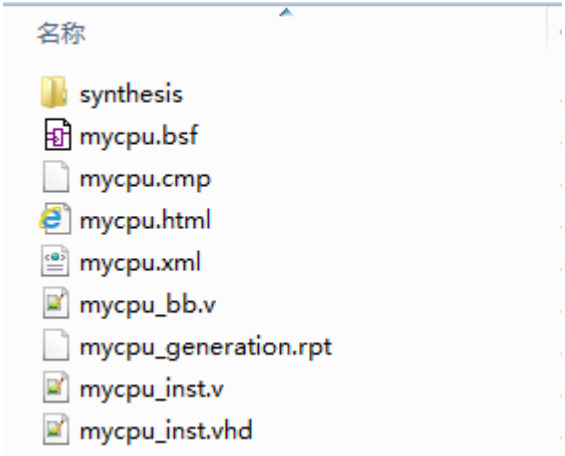
当用户使用 Qsys 创建一个新的系统时，Qsys 自动生成一个以系统名 命名的系统 qsys 文件，所有的设计信息都保存在该 qsys 文件中。当使用 Qsys 重新打开一个已有的系统时，qsys 文件是 Qsys 工具读取该系统具体设计信息的唯一来源。系统 qsys 文件的内容随着用户吐信用户界面对系统的编辑而改变。

当 Qsys 完成一个系统的搭建并生成了系统之后，会在工程文件夹下生成如下若干文件或文件夹。例如将 Qsys 系统保存为 mycpu.qsys，则工程文件夹下将会生成如下的文件或文件夹。

mycpu.qsys：Qsys 系统文件（.qsys)——用于描述系统的硬件结构。

mycpu.sopcinfo：Sopc 信息文件（.sopcinfo），Nios II IDE 使用.sopcinfo 文件来信息来为目标硬件编译软件程序。

mycpu 文件夹：mycpu 下包含了整个 Qsys 系统生成的各种详细文件，包括整个系统的例化文件(mycpu_inst.v)以及 Qsys 中使用到的所有组件的硬件描述语言(HDL)文件(在 synthesis 文件夹中)， QuartusII 符号模块文件（.bsf)——该文件中的符号（Symbol)用于在使用原理图设计顶层时添加到 Quartus II 工程顶层文件，使用 Verilog 设计顶层的方式不使用。



所有 Qsys 的菜单命令如表 2.1 所列。

在下一小节中，将以一个实例来讲述 Qsys 的使用，读者将执行下列步骤：

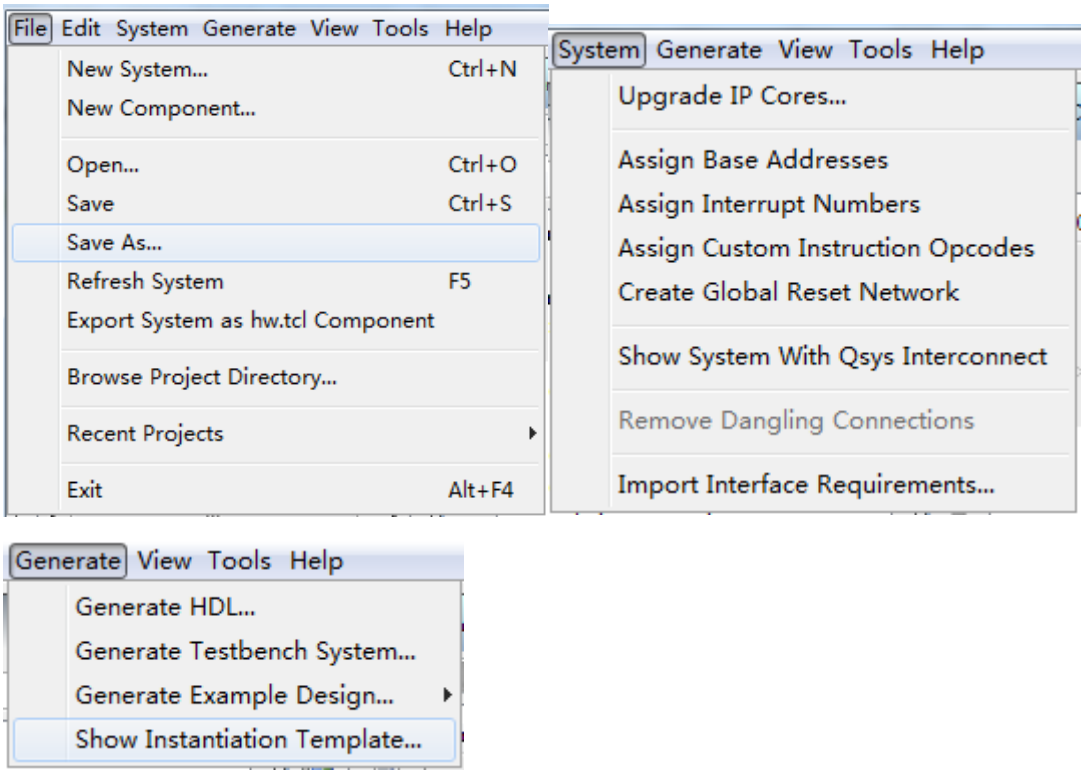
1) 启动 Qsys;
2) 指定目标 FPGA 和时钟设置;

3) 增加 Nios II 内核，片内存储器和其它外设。
4) 指定基地址和中断请求（IRQ)优先级。
5) 指定更多的 Nios II 设置。
6) 生成 Qsys 系统。

Qsys 设计过程不需要按固定的顺序进行，本节所述的设计步骤是常用的，便于读者理解。然而，读者可使用不同的顺序来执行 Qsys 设计步骤。

表 2.1 Qsys 菜单命令列表

菜单	命令	功能描述
File	New System	在当前项目目录新建一个 Qsys 系统项目文件
	New Component	将用户自定义的逻辑作为一个元件加入到 Qsys 中
	Open	打开一个 Qsys 系统项目文件：<系统名>.qsys
	Save	保存当前项目系统
	Save As	将当前项目系统另存为
	Refresh System	更新元件列表显示
	Export System as hw.tcl Component	将当前系统以 tcl 组件导出
	Browse Project Directory	浏览工程目录
	Recent Projects	最近打开的 Qsys 工程
	Exit	退出 Qsys 系统
System	Upgrade IP Cores	更新 IP 核（针对打开低版本软件的设计）
	Assign Base Addresses	自动给添加的个元件分配基地址
	Assign Interrupt Numbers	自动给添加的个元件分配中断优先级
	Assign Custom Instruction Opcodes	生成用户自定义指令的操作码（指令）
	Create Global Reset Network	创建全局复位网络
	Show System With Qsys Interconnet	使用 Qsys 内部连接查看工具查看系统
Generate	Generate HDL	生成系统的 HDL 设计文件
	Generate Testbench System	生成系统的 testbench 文件
	Generate Example Design	生成示例设计
	Show Instanttiation Template	生产 HDL 例化模版，用于添加到工程顶层



2.5.2 启动 Qsys

打开 Quartus II 软件。在 Quartus II 中选择【Tools】->【Qsys】选项来启动 Qsys，建议首先新建一个 Quartus II 工程，然后再打开 Qsys，如图 2. 14 所示。

Qsys 启动后，将自动创建好一个名为 unsaved.qsys 的工程，该项目中仅仅包含了一个时钟管理单元，如下图所示：

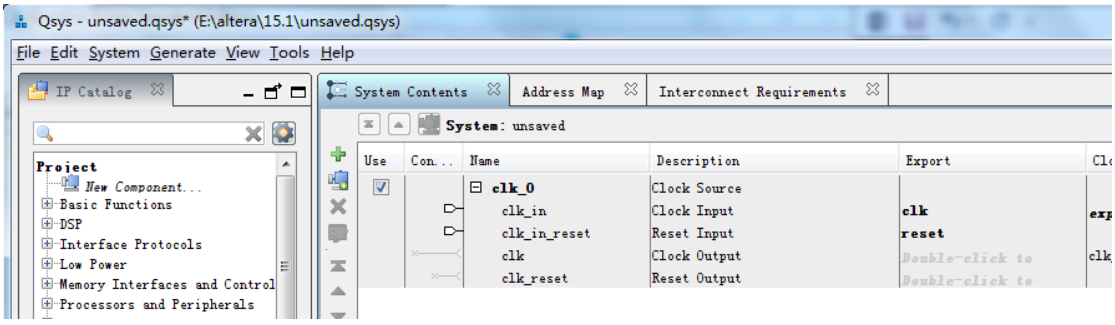


图 2. 14 启动 Qsys

这个时候我们点击【File】【Save】即可保存当前项目，这里我们将项目命名为 mysystem.qsys

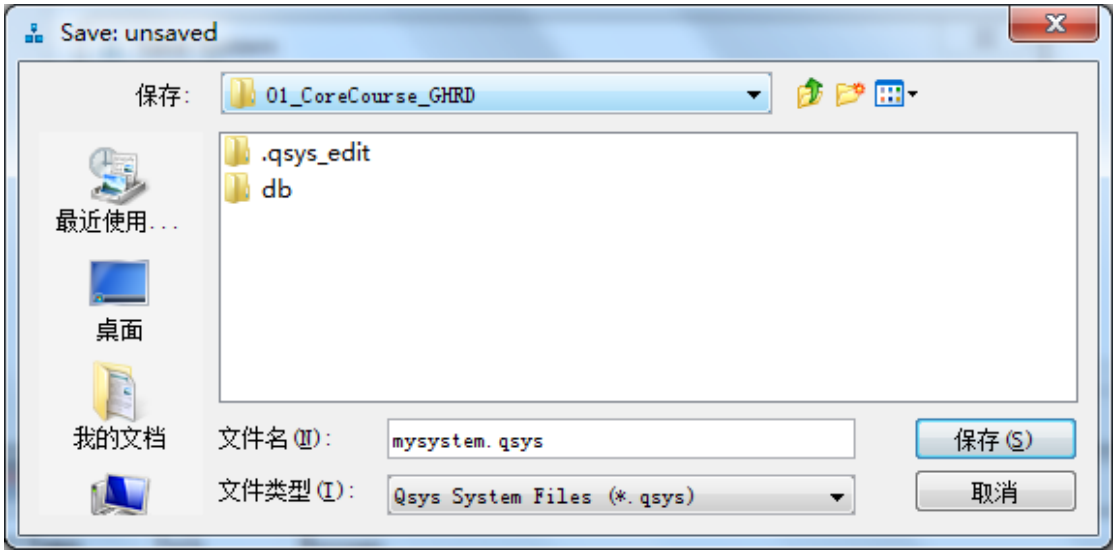


图 2.15 保存项目对话框

在图 2.15 中输入项目名称。在本例中项目名为 mysystem。

特别注意，项目名称不能与 2.4.2 小节中建立的顶层 HDL 文件名同名，否则在后期 Quartus II 编译该工程时将会报错。

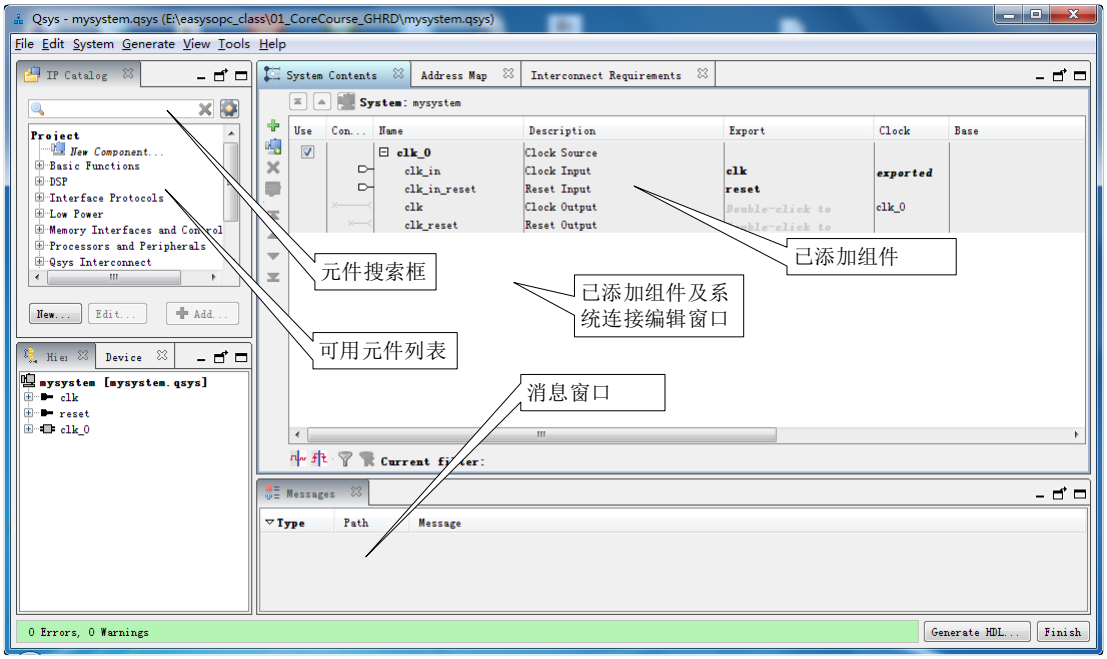


图 2.16 Qsys 界面

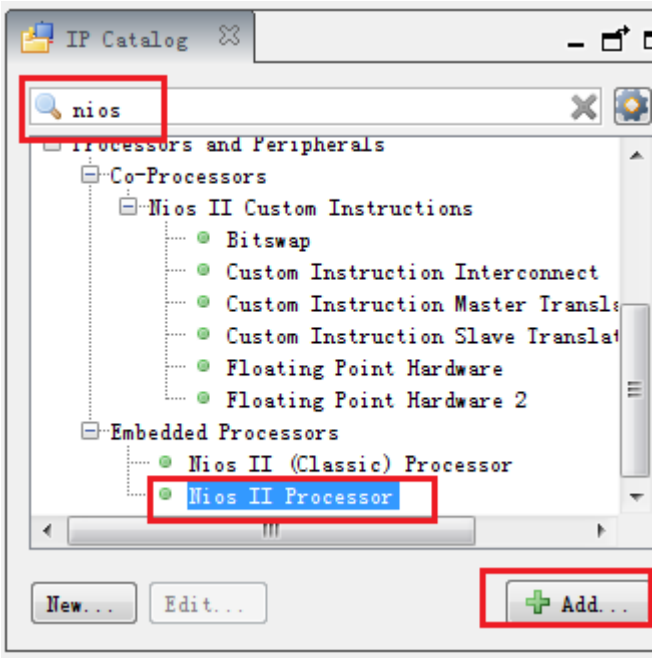
2.5.3 指定系统输入时钟频率

Qsys 工具打开后，clk_0 是自动添加进来的，该元件是用来对输入的时钟和复位信号进行处理的。我们双击该元件的名称（clk_0），将会在右侧弹出时钟设置对话框，这里我们可以设置输入时钟频率，因为系统默认为 50MHz，而我们芯航线 FPGA 开发板上使用的也是 50MHz 的晶振，即与系统默认一致，因此，这里我们不需要修改。当读者在使用自己的

目标板时，就需要根据自己的目标板实际情况进行设置。需要注意的是，所设置的频率一定要与系统实际运行的时钟频率相匹配。否则一些涉及到精确时序参数的外设（定时器、UART、SPI 等等）可能无法正常工作。

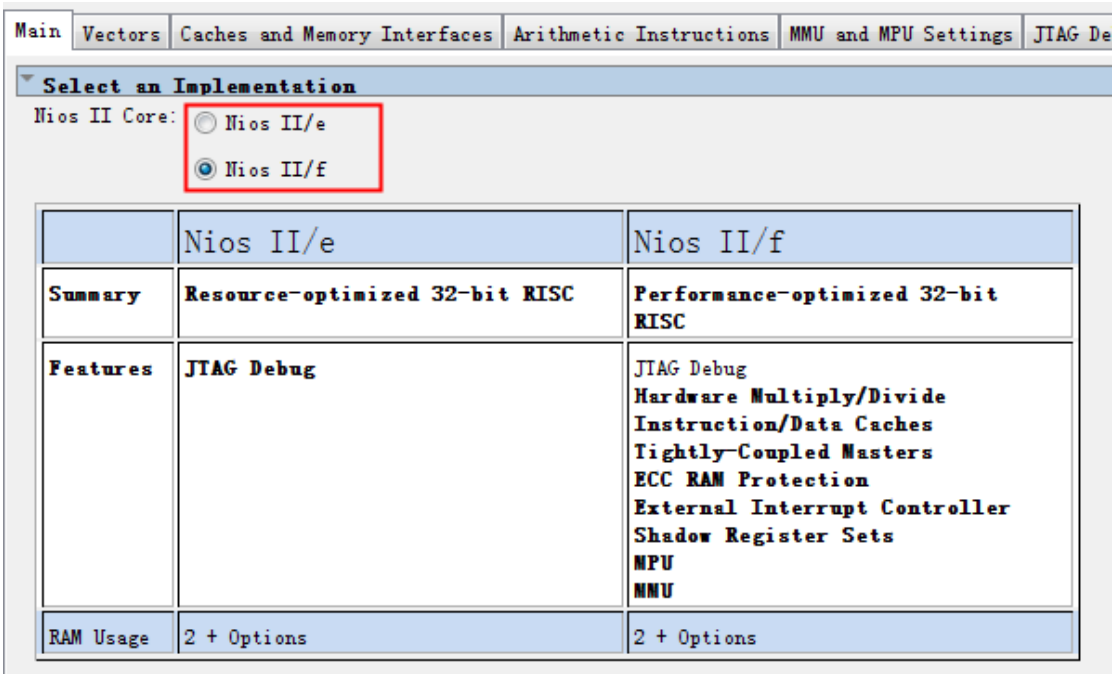
2.5.4 添加 Nios II 处理器内核

执行下面的步骤，添加 Nios II/s 内核到系统：



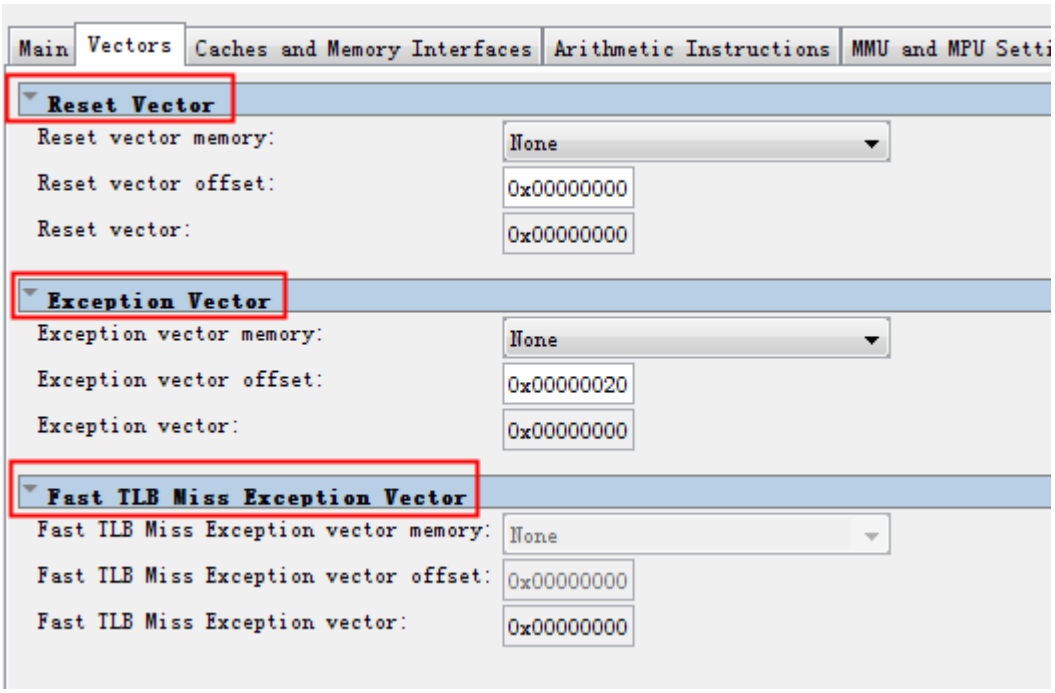
在图 2. 16 所示 IP Catlog 栏，搜索框中输入 nios，在搜索结果中选中 Nios II Processor，双击或者点击 Add 按钮打开 NiOS II 处理器配置窗口。这里分别对该设置窗口中的一些参数进行介绍。

main： 选择 NIOS II 处理器的内核类型。从 main 选项卡可以看到，Nios II 系列支持两种类型的 CPU，并给出了其支持的相关功能特性，详细内容见 3. 10 节所列。根据应用要求和目标 FPGA 中的器件约束（逻辑资源）条件，选择适合的 CPU 核。由于芯航线 FPGA 学习套件上使用的 FPGA 器件（EP4CE10）的逻辑资源比较充足（10K），因此本实例选择快速型 CPU 核 Nios II/f，如果用户使用的是逻辑资源有限的器件，如 EP4CE6，则使用快速型内核比较紧张，建议选择经济型内核。这也是我们选择 EP4CE10 这款器件的原因，为了给用户学习和使用 NIOS II 处理器留出相当的逻辑资源。



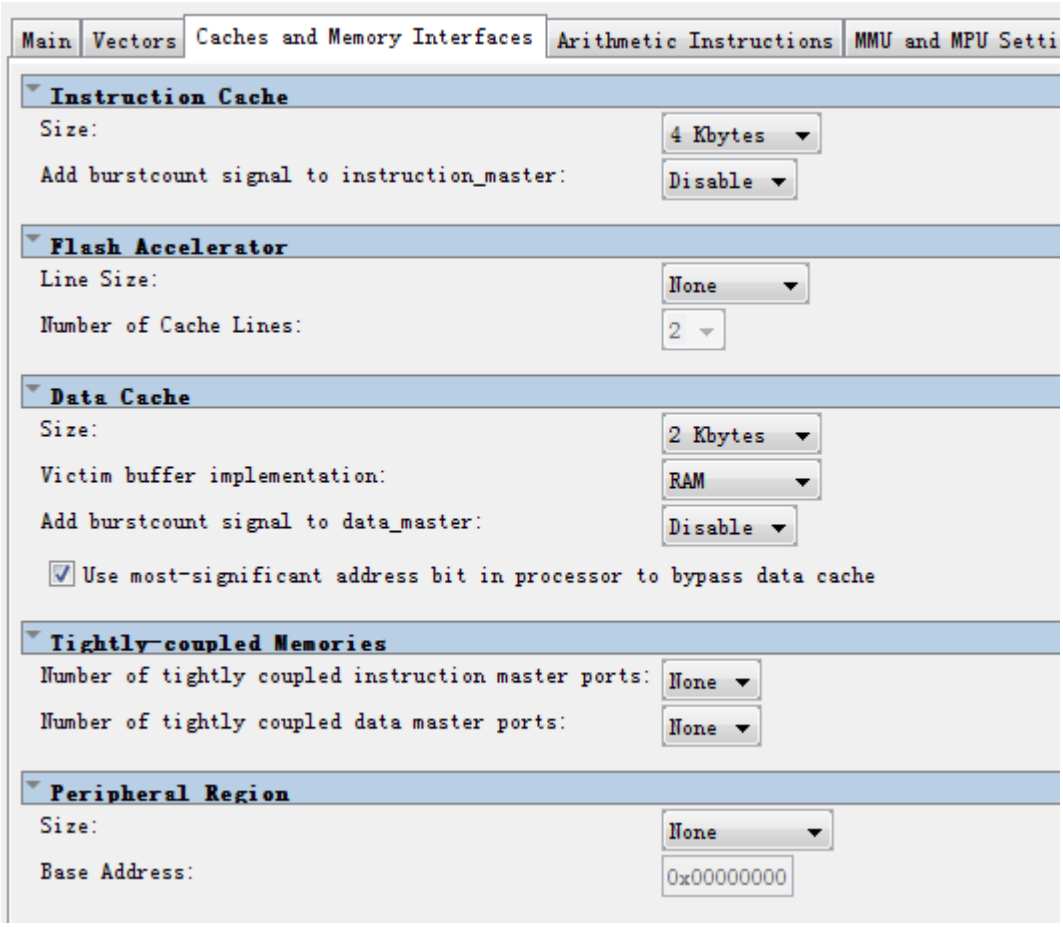
Vectors：设置 CPU 的复位向量（Reset Vevtor）、异常向量(Exception Vector)和快速 TLB 命中异常向量（Fast TLB Miss Exception Vector）。

- Reset Vevtor：复位向量用于设置 CPU 复位后从何处启动，例如我们在调试系统时，为了方便，常设置 CPU 的复位向量为 SDRAM 或 SRAM，而当我们的系统开发完成后，则将 CPU 的复位向量设置于 FLASH，例如 EPCS 存储器。这样系统即可上电后自动从 FLASH 中复制程序代码到内存中并运行。该选项只有当 CPU 的存储器系统就绪后方可配置，因此，当我们添加 NIOS II CPU 时，因为此时 CPU 的存储器组件并未就绪，因此此项无法配置，只有等我们后续为 NIOS II 添加好存储器组件后，再回头来进行修改。
- Exception Vector：异常向量用于存放 CPU 处理异常事件的代码，例如中断处理代码。在一个典型的系统中，该向量应该指向一个低延迟的存储器，例如 SDRAM、SRAM、或者片上存储器（On Chip RAM）。
- Fast TLB Miss Exception Vector：快速 TLB 命中异常向量主要与 MMU 协作，管理存储器，本书所讲解的知识赞不涉及到 MMU 方面的内容，因此此部分内容暂不做介绍。

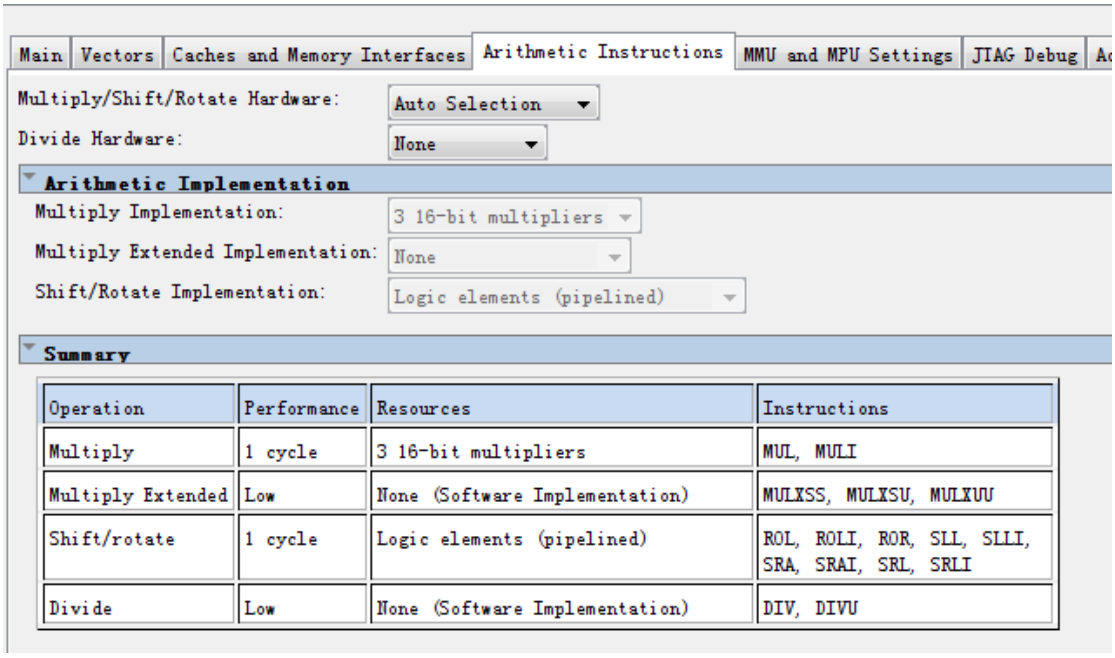


Caches and Memory Interfaces Tab： 本选项卡主要设置 CPU 的数据和指令缓存部件。

- Instruction cache：指令缓存用于缓存 NIOS II 处理器的各种指令，以提高 CPU 的运行效率。该缓存的大小从 512 字节到 64K 字节用户可自定义设置，，当然也可以不使用指令缓存。当选择不使用指令缓存时，NIOS II 处理器的指令总线将会被移除，因此，我们必须要包含一个紧耦合型指令存储器（tightly-coupled instruction memory）。关于此部分的深入应用，需要用户扎实掌握 NIOS II 处理器的知识后，再来深入研究，刚开始接触时，在芯片资源充足的情况下，使用默认设置即可。
- Data cache：数据缓存用于存放数据，Data cache 数据缓存选项用于指定缓存的大小。该缓存的大小从 512 字节到 64K 字节用户可自定义设置，当然也可以不使用指令缓存。关于此部分的深入应用，需要用户扎实掌握 NIOS II 处理器的知识后，再来深入研究，刚开始接触时，在芯片资源充足的情况下，使用默认设置即可。

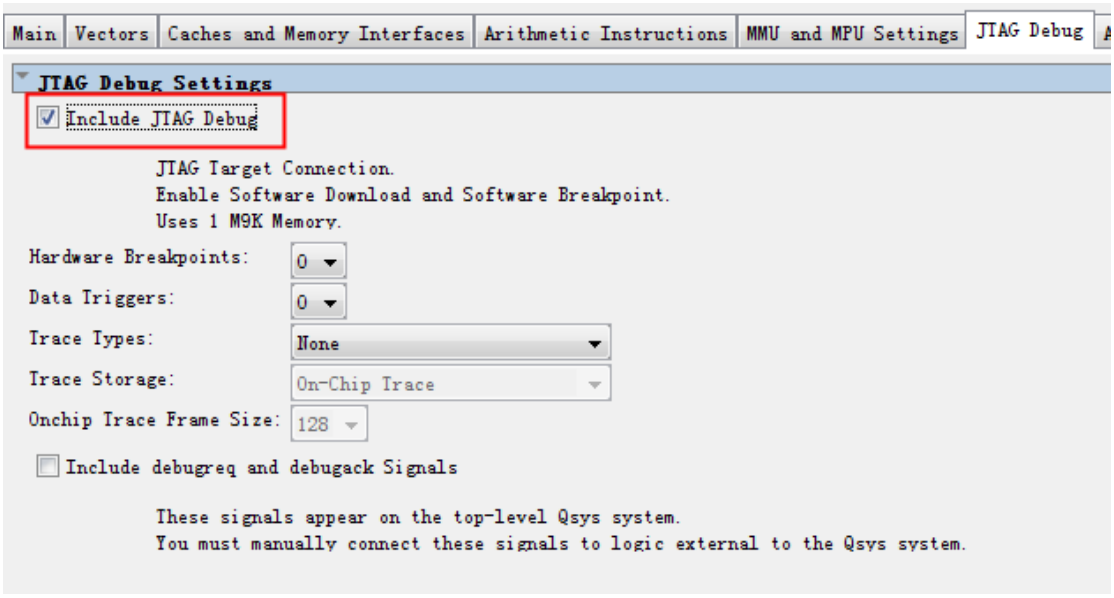


Arithmetic Instructions:该选项卡用来设置 NIOS II 处理器实现算术指令的方式，这里我们可以通过选择 Divide Hardware 来设置 NIOS II 处理器是否支持硬件除法运算。也可以选择乘法、移位等运算的实现方式。



JTAG Debug： 为了方便调试，为 CPU 加入 JTAG 调试模块。JTAG 调试模块要占用较多的逻

辑单元， 如果整个系统已经调试完毕了，可以选用取消勾选 Include JTAG Debug，以减少系统占用资源。JTAG 调试模块根据功能的不同可设置是否使用硬件断点，使用数据触发，跟踪类型和跟踪存储等。



其他选项为用户在实际的应用情况中根据自己的需求合理设置，这里不做过多的介绍。在本实例中，我们选择使用 Nios II/f 型内核，不使用硬件除法器，指令缓存默认设置为 4K 字节，数据缓存默认设置为 2K 字节。Jtag Debug 默认，其他设置不变。点击 Finish 按钮即可完成 NIOS II 处理器的添加。

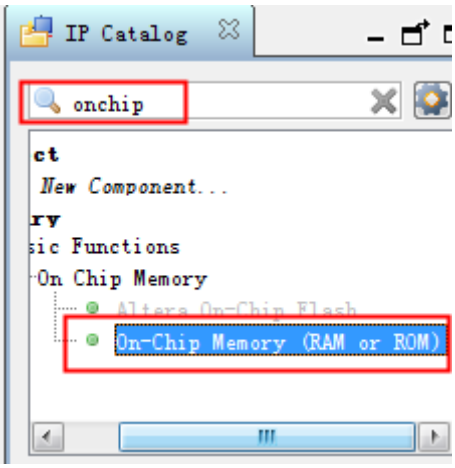
添加完成后，系统自动将该 NIOS II 处理器命名为 nios2_gen2_0，我们可以选中改名字，点击鼠标右键选择 rename 以对处理器进行重命名，这里我们不做改变，使用默认名。

✧ 注意，大家在给 Qsys 中创建的各个组件重命名时，名字最前面应该使用英文字母。能使用的字符只有英文字母、数字和“_”，不能连续使用“_”符号，在名字的最后也不能使用“_”。

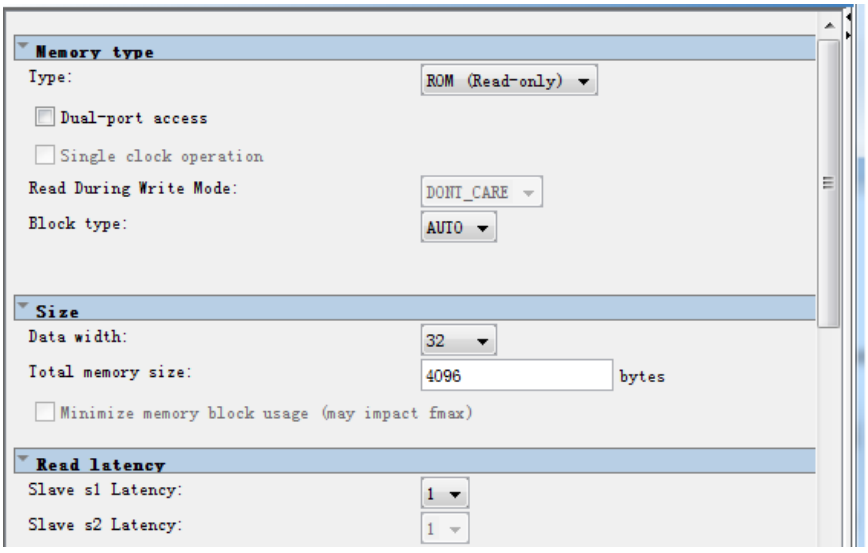
2.5.5 添加片内存储器

众所周知，处理器系统至少要有一个存储器用于数据和指令。本章实例设计一个 4KB 片内 ROM 存储器（Onchip_ROM），用于存储程序代码以及程序运行空间；4KB 片内 RAM 存 储器（onchip_RAM)用于变量存储（R/W 数据）、Heap、stack 等。执行下列步骤来添加存储器：

(1) 如图，在 IP Catalog 中，输入 onchip 来进行过滤，在搜索结果中选择并双击 On-Chip Memory (RAM or ROM),弹出片内存储器配置向导对话框，如 图 2. 21 所示。



(2) 在 Memory Type 选项区域中选中 ROM (read-only)，即指定为 ROM 型。FPGA 内部其实并没有专用的 ROM 硬件资源，实现 ROM 的思想是对 RAM 赋初值，并保持该初值，使其为只读的。ROM 的内容在对 FPGA 进行配置时，一起写入 FPGA。



(3) 在 Total Memory Size 文本框中输入 4096，即指定 4 KB 的存储容量。

(4) 不要改变其它任何默认设置。

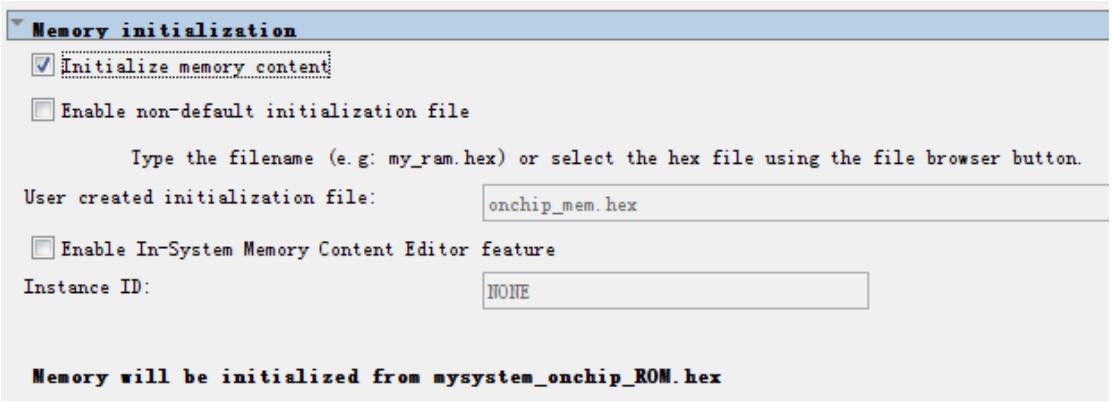
(5) 单击 Finish，在激活元件窗口中出现名称为 onchip_memory2_0 的片内存储器。

(6) 右击激活元件窗口中Onchip_memory_0,然后选择 Rename 将 onchip_memory_0 重命名为 onchip_ROM，如图 2. 22 所示。



(7) 更改了 ROM 的名称后，我们在此双击 onchip_ROM 以对其参数进行编辑或查看，在最底部可以看到 Memory will be initialized from mysystem_onchip_ROM.hex，即 onchip_ROM 存储器在上电时将使用 mysystem_onchip_ROM.hex 文件进行初始化。mysystem_onchip_ROM.hex 在 Quartus II 工程目录下，该文件可以由用户编辑生成，也可以由 IDE 编译生成，见 2. 9. 3 小节。在本实例中，将使用 onchip_rom 来存储用户程

序，mysystem_onchip_ROM. hex 将由 IDE 编译生成，文件的内容即用户程序。



使用与添加 onchip_RAM 相同的方法，添加一个 4KB 的 onchip_RAM。添加步骤中不同之处在于：

(1) 在第（2）步中，Memory Type 选项区中选中 RAM(writeable)，即指定为 RAM 型；

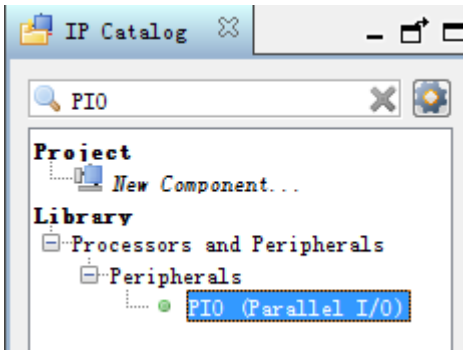
(2) 在第（3）步中，Total Memory Size 文本框中输入 4096，即指定 4 KB 的存储容量。

(3) 在第（6）步中，重命名为 onchip_RAM。

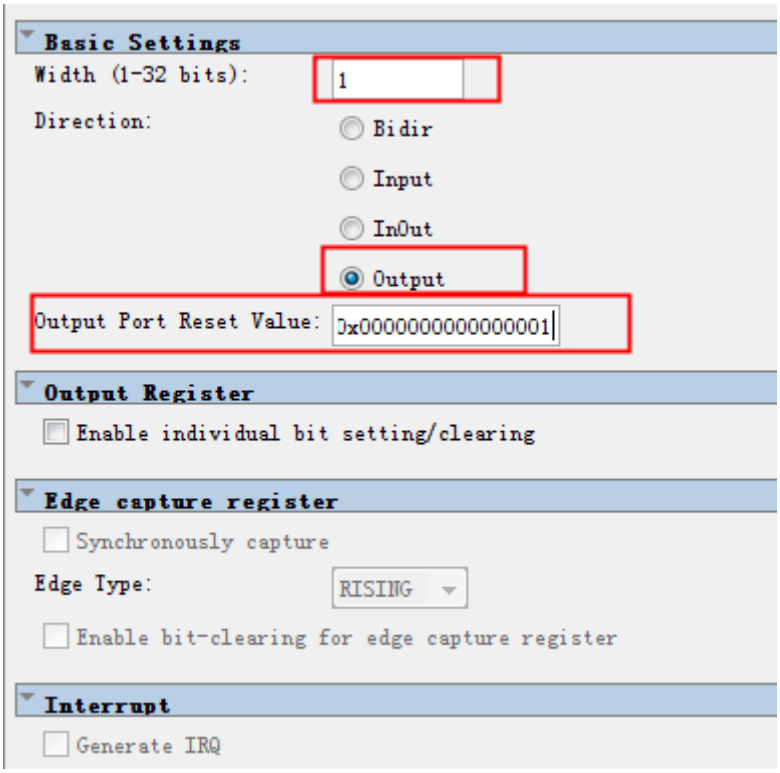
2.5.6 添加 PIO

PIO 为 Nios II 处理器系统接收输入信号以 及输出信号提供了一种简易的方法。本实例设计使用 1 个 PIO 来驱动 1 个 LED。执行下列步骤添加 PIO:

(1) 如图，在 IP Catalog 中，输入 PIO 来进行过滤，在搜索结果中选择 PIO （Parallel I/O）并双击弹出 PIO 配置向导对话框，如 图 2. 23 所示。



(2) 在 Width 选项区的文本框中输入 1，将 PIO 的宽度改为 1 位；在 Direction 选项区选择 Output，即仅输出模式。Output Port Reset Value 处修改值为 1，即设定 PIO 复位时为 1。这里设置为 1 主要是为了使系统复位后 LED 处于熄灭状态。



(3) 单击 Finish，在激活元件窗口中出现名称 为 pio_0 的 PIO。

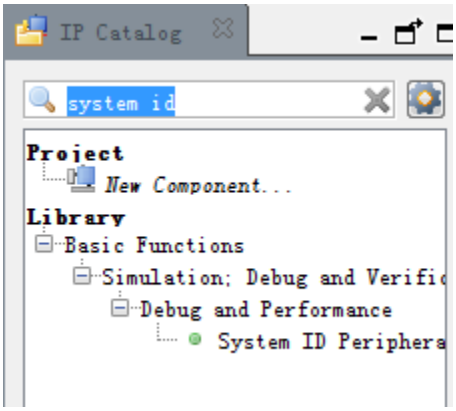
(4) 右击激活元件窗口中 pio_0,选择 Rename 将 pio_0 重命名为 pio_led。给出硬件外设的描述名称是一种良好的习惯。这样会方便我们在分配引脚和编写软件程序时对外设的理解。

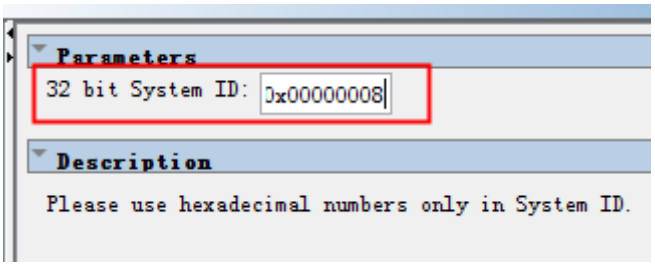
注意：在 Nios II IDE 的用户程序中，将使用 PIO_LED（NIOS II IDE 中不论组件名的大小写，统一使用大写字母，无需用户强制使用大写字母命名）这个名称来访问外设，所以在这里推荐将 PIO 命名为 pio_led 当然，用户也可以命名为其他的名称，只是要在 IDE 的用户程序中，使用相应的名称来访问 PIO 即可，由于考虑到是第一个实例，为了尽量减少学习者的出错几率，这里还是强烈建议大家直接使用 PIO_LED 或者 pio_led 来命名。

2.5.7 添加系统 ID 外设

如果系统包括系统 ID 外设，则当 Qsys 生成 Nios II 系统时，将为该 Nios II 系统生成一个标识符（ID 号）。该标识符会被写入 SYSTEM ID 寄存器中，供 IDE 编译器和用户辨别所运行的程序是否与目标系统匹配。在 IDE 中，如果用户程序不是基于对应的 Nios II 系统的，那么调试时，Nios II IDE 将阻止用户下载程序到 Nios II 系统，当然，我们也可以在调试时选择忽略系统 ID 的核对来跳过这个校验。

在 IP Catalog 中，输入”system id”来进行过滤，在搜索结果中选择 System ID Peripheral 并双击弹出 System ID 配置向导对话框，在这里，我们可以在“32 bit System ID”处手动的输入一个 ID 作为系统标识符，本实例我们设置 0x00000008。





单击 Finish，在激活元件窗口中出现名称 为 sysid_qsys_0 的 System ID Peripheral。

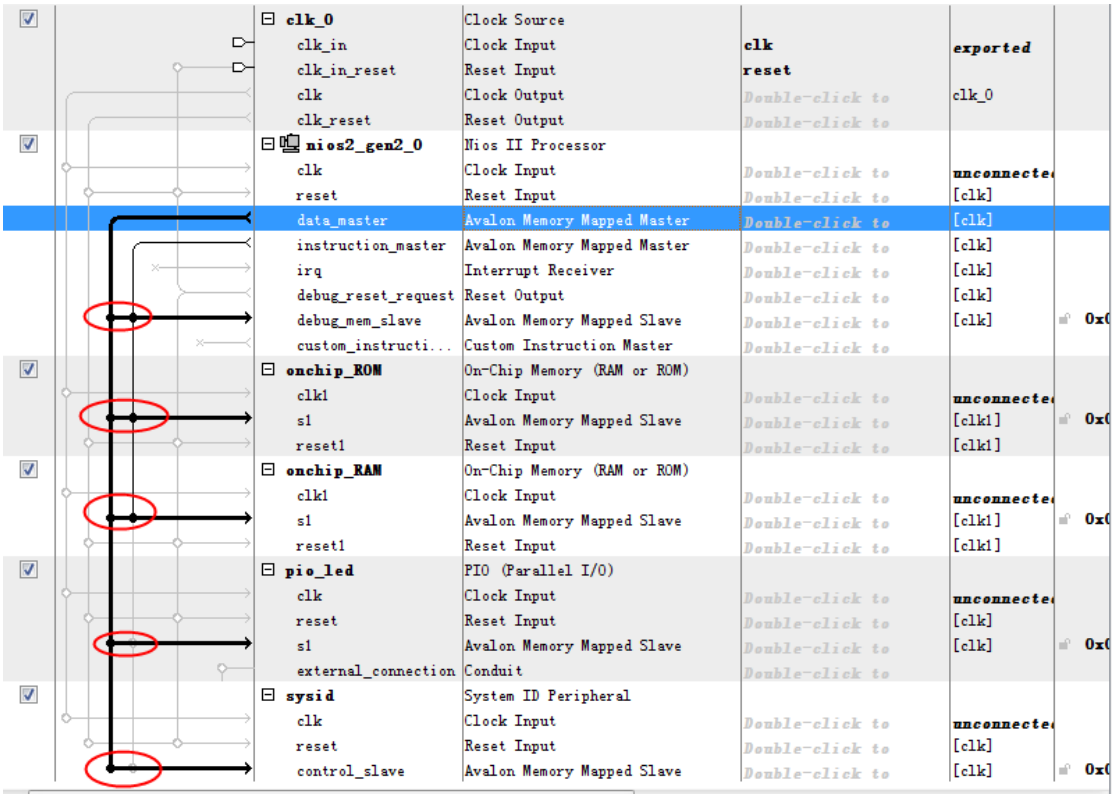
右击激活元件窗口中 sysid_qsys_0，选择 Rename 将 sysid_qsys_0 重命名为 sysid。

2.5.8 连接各外设到 Avalon MM 总线上

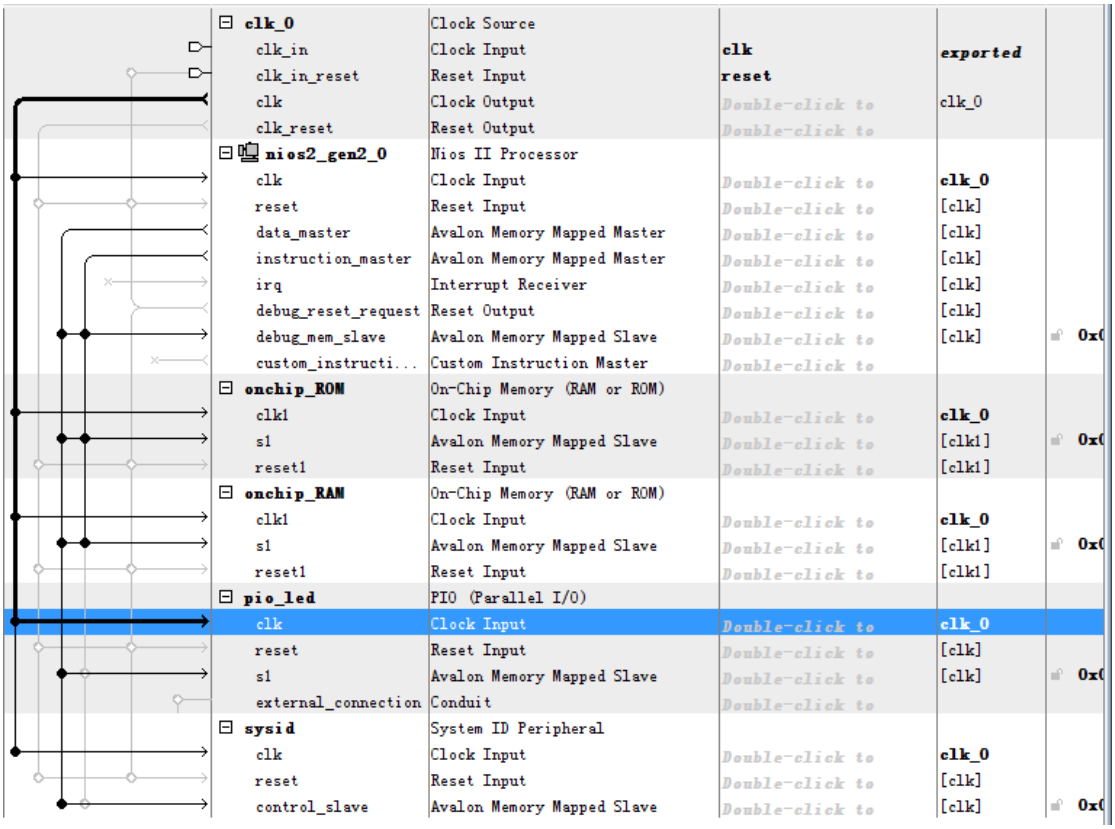
通过以上步骤，我们完成了添加实现点亮 LED 灯所需的所有外设组件，然而，这些组件当前都还是各自独立的，与系统核心 NIOS II CPU 并没有关联起来。因此我们需要使用 Avalon MM 总线或 Avalon ST 总线来将这些组件连接起来，因为本实例中不存在数据流组件，因此不涉及到 Avalon ST 总线的连接。

本系统中，总共涉及到 Avalon MM 总线的指令总线、数据总线、时钟网络、复位网络和中断网络。一个完整的系统由至少一个主设备和至少一个从设备构成，本实例中 NIOS II CPU 为主设备，sysid 和 PIO_LED 为从设备。onchip_ROM 和 onchip_RAM 为存储器设备。我们连接时，首先将存储器设备的 slave（s1）同时和 NIOS II CPU 的数据总线（data master）和指令（instruction master）总线连接，然后将 pio_led 和 sysid 的 slave（s1）与 NIOS II CPU 的数据总线（data master）相连，不要将 s1 与指令总线相连接。连接总线的方法很简单，将两个需要连接到一起的总线交叉处的空心圆圈点击一下，即可实现两个网络的连接。

关于什么时候需要连接指令总线，什么时候不需要，这里给出一个连接技巧：凡是不涉及到存储 CPU 指令的外设，如串口，定时器，SPI 等非存储器外设，只需要将 s1 连接到 CPU 的 data master 即可。而对于需要存储 CPU 程序指令的存储器，如 EPCS、onchip_ROM、SDRAM 等，需要将 s1 和 CPU 的 data master 和 instruction master 同时连接。当然，一些明确不会存储指令的存储器，也不需要连接 instruction master。连接完成的网络如下图所示：



以上完成了所有组件的数据总线和指令总线的连接，接下来还需要完成时钟网络和复位网络的连接。本实例中只有一个时钟网络，因此将所有组件的时钟网络连接到一起即可，连接时钟网络的系统如下图所示：

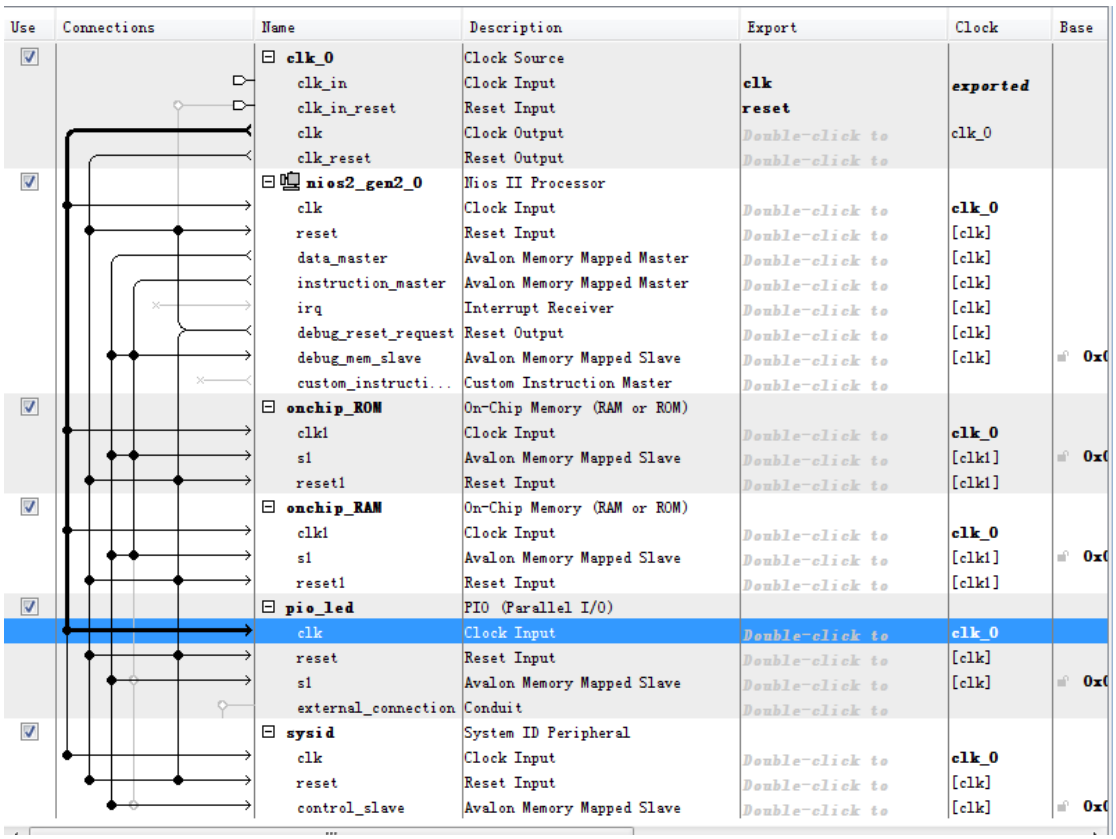


接下来我们需要连接全局复位网络，连接复位网络有两种方式，第一种方式和连接时钟

网络一样，手动一个一个的去点击交叉点连接。第二种方式就是直接点击菜单栏【System】【Create Global Reset Network】来连接复位网络，这里推荐初次接触 Qsys 的用户使用第二种方式，以减少出错的概率。

由于本系统中没有添加能够产生中断请求的组件，因此没有中断网络需要连接。关于中断网络的连接和中断号的分配，我们将在后续章节讲解中断的使用的时候进行讲解。

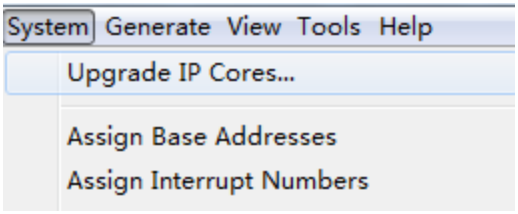
所有总线连接完成后的系统如下图所示。



2.5.9 指定基地址和中断请求优先级

在添加和完成组件间的总线网络连接后，我们就需要为每个外设分配基地址和中断请求优先级，由于本实例中所有外设都不含有中断，所以无需进行中断优先级分配。

Qsys 提供 Assign Base Addresses 和 Assign Interrupt Numbers 命令，如下图所示，这两个命令可以分别简单的实现分配外设基地址和中断优先级。



对于许多系统，包括本实例设计，Assign Base Addresses 能满足要求。当然，用户可以自己调整基地址和中断优先级来满足系统的需求。NIOS II 处理器的内核可寻址 31 位地址范围（2G），用户必须分配 0x00000000~0x7fffffff 之间的基地址。NIOS II 程序使用宏定义的符号

常量来访问外设，基地址的改变不会造成程序的修改。

由于 Qsys 只是搭建系统，并不处理软件操作，因此它不能做出关于最好的 IRQ 分配的有根据猜测，因此建议用户不采用自动分配的方式，而是自己根据系统结构手动分配中断优先级。

这里，我们要分配系统基地址，可以使用自动分配的方式，直接点击【System】【Assign Base Addresses】即可。

2.5.10 设置 NIOS II 复位和异常地址

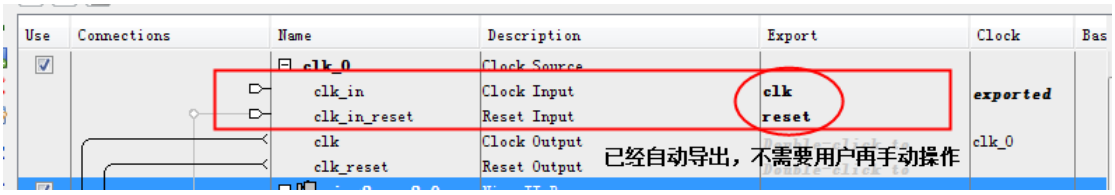
双击 nios2_gen2_0 组件，在打开的选项卡中，点击 Vectors 切换到向量设置选项卡。本实例，系统上电后，从内部 ROM 开始运行，所以 Reset Vector 的 Reset Vector memory 选择 onchip_ROM.s1,offset 为 0x00000000。异常向量表放在内部 RAM 里面，所以，Exception Address 选择 onchip_RAM.s1，Offset 地址为 0x00000020。

注意：Reset Address 和 Exception Address 的 Offset 只有在多处理器的系统中才进行设置，且其值必须为 0x20 的倍数，如果地址设置违反规则，信息窗口会给出错误提示。

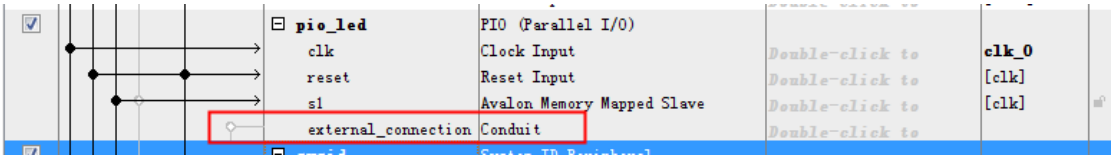
2.5.11 设置需要引出到 Qsys 系统顶层的信号

在最终形成的 FPGA 系统中，Qsys 所创建的组件是以一个子模块的形式添加到 Quartus II 工程中的，而作为子模块，就存在相应的输入输出，在 Qsys 系统中，当系统组件添加完成后，还需要手动的将系统中各个组件对应的信号导出到顶层，这样才能正常的加入到 Quartus II 工程中。

那么具体有哪些信号是需要导出到模块顶层的呢？在本例中 clk_0 模块的 clk_in 和 clk_in_reset 信号是需要导出到顶层的，但是这两个信号在系统创建之时就自动存在并导出到顶层了，因此不需要我们手动再去操作。

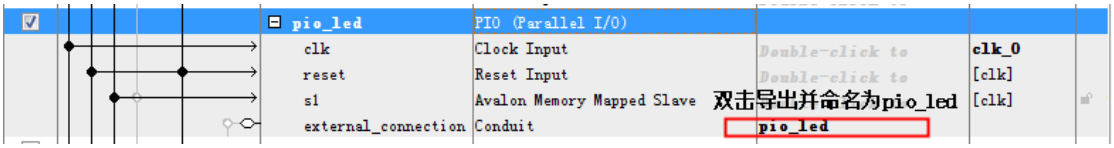


而 onchip_ROM、onchip_RAM、systemID 等外设仅与 CPU 进行交互，没有需要导出到系统顶层与其他模块或引脚连接的信号，因此不用连接。pio_led 作为一个输出引脚，最终是通过引脚连接到开发板上的 LED 灯的控制角，因此需要导出到顶层。每一个模块，如果有需要导出到系统顶层的信号，则有一个名为 external_connection 的总线名，其属性为 Conduit。因此，对于刚接触 Qsys 系统的用户，只需要认准这种属性的信号，即为需要导出到顶层的信号。



导出这个信号，只需要在该信号的 export 一栏双击，并输入你希望命名的信号名称即

可。例如，本实例中，我们双击 Double-click to 即可输入信号名，这里我们双击并经信号名修改为 pio_led 即可。如下图所示：



至此，我们整个系统就搭建完成了，接下来点击【File】->【Save】保存当前系统，然后即可开始生产该系统对应的 HDL 文件了。

2.5.12 生成 NIOS II 系统

使用 Qsys 生成 NIOS II 系统，执行以下步骤：

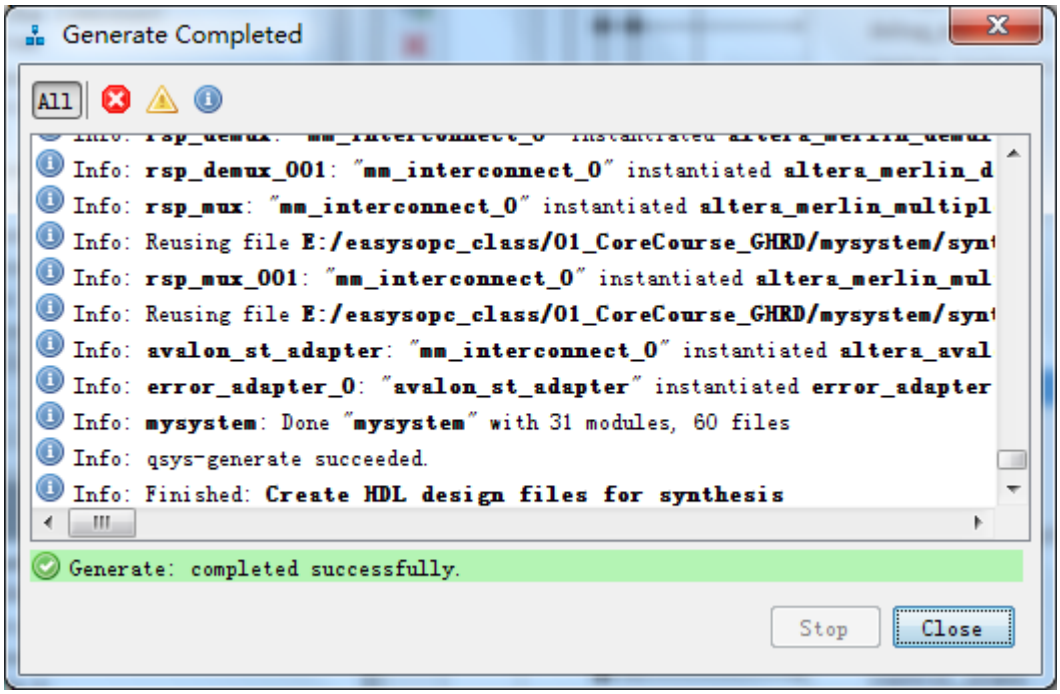
1、 点击【Generate】->【Generate HDL】打开 Generation 对话框。

2、 默认输出路径不变

3、 点击 Generate 按钮，系统生成开始。

在系统生成过程中，Qsys 会执行一系列操作，Qsys 会为添加的所有部件生成 Verilog HDL 源文件，并生成每个硬件部件以及连接部件的片内总线结构、仲裁和中断逻辑。Qsys 会为系统生成 NIOS II IDE 软件开发所需的硬件抽象层（HAL）、C 以及汇编头文件。这些头文件定义了存储器映射、中断优先级和每个外设寄存器空间的数据结构。这样的自动生成过程有助于软件设计者处理硬件潜在的变化性。如果硬件改变了，Qsys 会自动更新这些头文件。Qsys 也会为系统中现有的诶个外设生成定制的 C 和汇编函数库。如果添加了片内存储器，Qsys 还将为片内 ROM、RAM 生成其初始化所使用的 HEX 文件（空白文件）。在生成阶段的最后一步，Qsys 创建适合于系统部件的总线结构，并把所有部件连接在一起。

生成过程时间随计算机的性能而不同，一般需要几分钟。生成结束后，显示 Finished: Create HDL design files for synthesis，点击 Close 即可。然后关闭 Qsys 工具。



这时，在 Quartus II 工程中会弹出以下窗口提示，点击 OK 即可。此时，我们一个基于

NIOS II 处理器的系统就搭建完成了。

NIOS II 系统生成后将产生下列文件：

- ✓ **Qsys 系统文件**：mysystem.sopcinfo，它定义了 Qsys 生成完整系统必须的详细信息，该文件存储 NIOS II 系统的硬件内容，NIOS II IDE 需要使用该文件信息来为目标硬件编译软件程序。详细内容见 2.9.1 小节。
- ✓ **硬件描述语言（HDL）文件——**mysystem.v 以及各外设的 HDL 文件，这些文件是描述 NIOS II 系统硬件设计文件。Quartus II 软件将使用这些 HDL 文件来编译整个 FPGA 设计。这些文件位于 E:\easysopc_class\01_CoreCourse_GHRD\mysystem\synthesis 路径下。

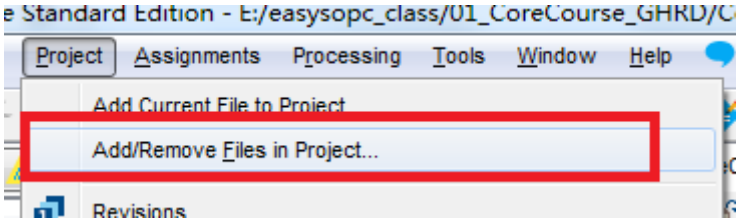
至此，已经完成 NIOS II 处理器系统的创建。生成系统结束后，要将系统集成到 Quartus II 硬件工程并使用 NIOS II IDE 来开发软件，可以先使用 NIOS II IDE 进行软件开发，也可以先将系统集成到 Quartus II。当然，如果是多人合作开发，两者可同时进行。

2.6 集成 NIOS II 系统到 Quartus II 工程

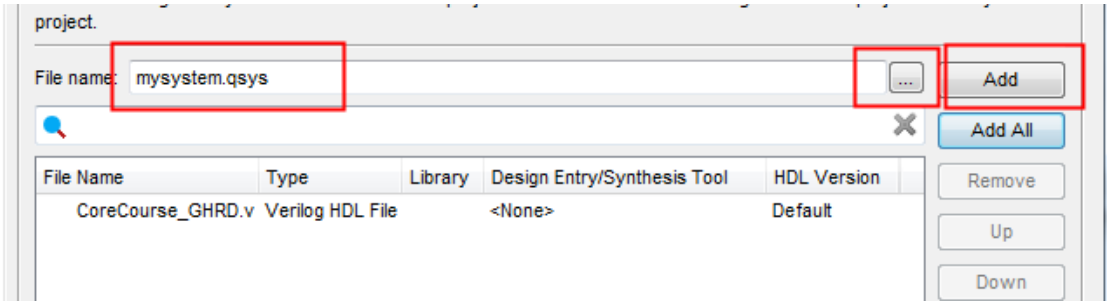
在 2.4.2 小节中已经建立好了 Quartus II 顶层模块，本节的内容是将刚生产的 NIOS II 系统添加到顶层模块中。

2.6.1 添加 NIOS II 系统模块到 Quartus II 顶层模块

添加 NIOS II 系统模块到 Quartus II 顶层模块，需要先将生成的.qsys 文件添加到工程中。点击【Project】->【Add/Remove Files in Project】。

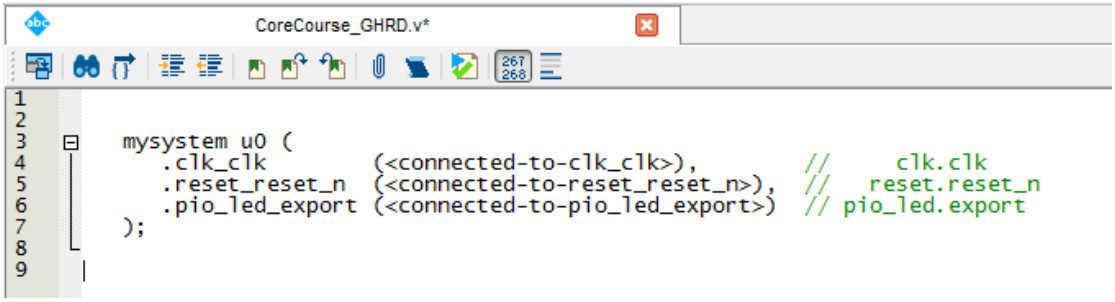


在打开的窗口中点击浏览文件按钮，选择 mysystem.qsys 文件，然后点击 Add 添加。最后点击【Apply】【OK】即可。



接着将 NIOS II 系统例化到工程顶层文件中，这里可以直接打开 E:\easysopc_class\01_CoreCourse_GHRD\mysystem 目录下的 mysystem_inst.v 文件，然后复制

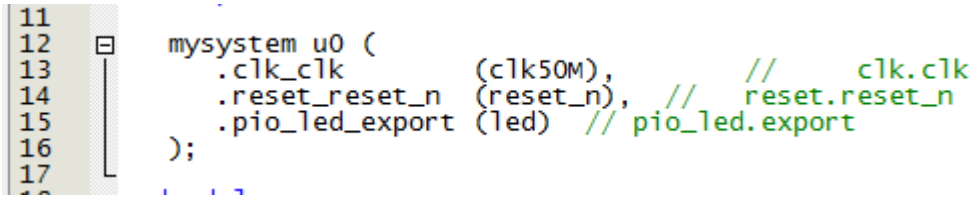
其中的内容，粘贴到顶层文件 CoreCourse_GHRD.v 中。



最后我们完成完整顶层文件的其他部分设计。首先编写端口列表，本实例系统比较简单，仅有时钟（clk）、复位（reset_n）两个输入以及一组 LED 输出。因此编写端口列表如下：

```
module CoreCourse_GHRD(
    clk50M,
    reset_n,
    led
);

    input clk50M;   //参考时钟输入，来自板载有源晶振，50M
    input reset_n;  //复位输入，外接板载清楚按键，默认高电平

    output [3:0]led;    //led，低电平点亮，高电平关闭
```

接着我们将 NIOS II 系统的各个信号连接到整个系统的输入输出管脚上。clk_clk 连接 clk50M，reset_reset_n 连接 reset_n，pio_led_export 连接到连接如下所示：



最终，整个顶层文件的代码如下所示：

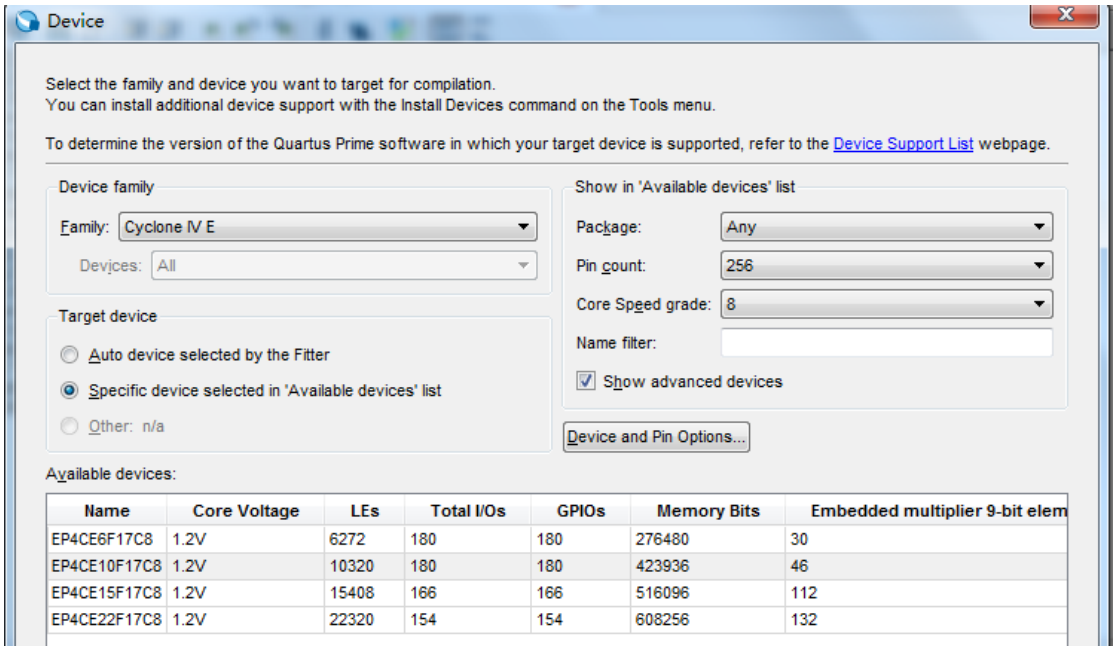
```
module CoreCourse_GHRD(
    clk50M,
    reset_n,
    led
);

    input clk50M;   //参考时钟输入，来自板载有源晶振，50M
    input reset_n;  //复位输入，外接板载清楚按键，默认高电平

    output [3:0]led;    //led，低电平点亮，高电平关闭

    mysystem u0 (
        .clk_clk        (clk50M),       //     clk.clk
```

```
        .reset_reset_n  (reset_n),  //   reset.reset_n
        .pio_led_export (led)  // pio_led.export
    );

endmodule
```

2.6.2 选择器件型号

每种型号的 FPGA 芯片的引脚可能都不相同，因此在进行引脚分配之前，都应该选择相应目标 FPGA 芯片新号。这一步其实在 2.4.1 小节中已经完成，这里只是让读者了解，在 Quartus II 工程创建好后仍然可以选择并修改器件型号。

在 Quartus II 菜单栏中依次点击【Assignments】->【Device】打开器件选择对话框，如下图所示。在该对话框中，指定所需的目标器件型号，本实例为选用芯航线 AC620 开发板上使用的 FPGA 芯片 EP4CE10F17C8。如果使用其他板卡，请选择板上对应的芯片型号。



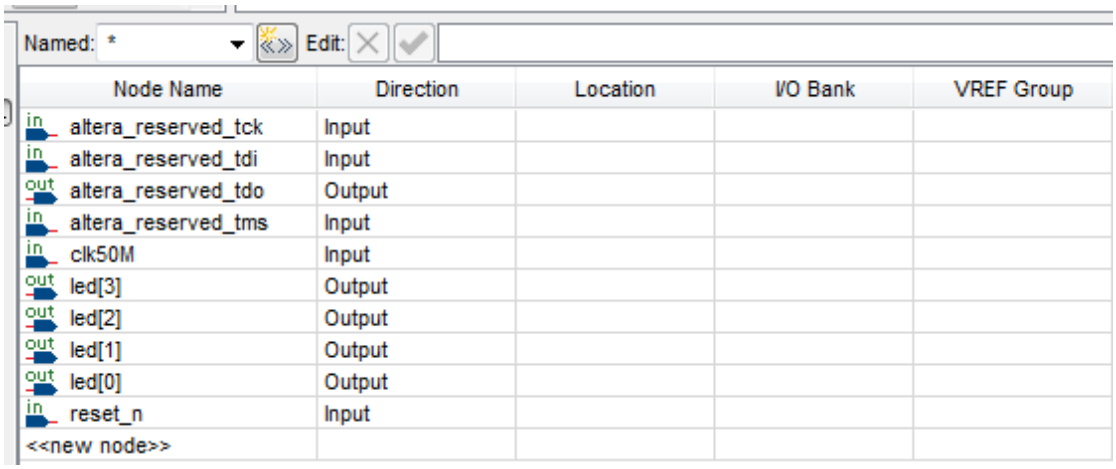
2.6.3 分配 FPGA 引脚

本实例中各引脚的命名以及与目标 FPGA 板卡上对应功能引脚标号如下表所示，注意，此处要根据不同的目标板进行调整。一般板卡商都会提供相应的引脚分配说明表。

分配 FPGA 引脚的位置，执行下面步骤：

要进行引脚分配，需要先对工程进行分析和综合，当分析和综合完成后，将会自动的分析得到工程中的各个端口信号。点击键盘组合键 Ctrl + K，对工程启动分析和综合。

分析和综合完成后，在 Quartus II 菜单栏中依次点击【Assignments】->【Pin Planner】打开引脚分配对话框，如下图所示。

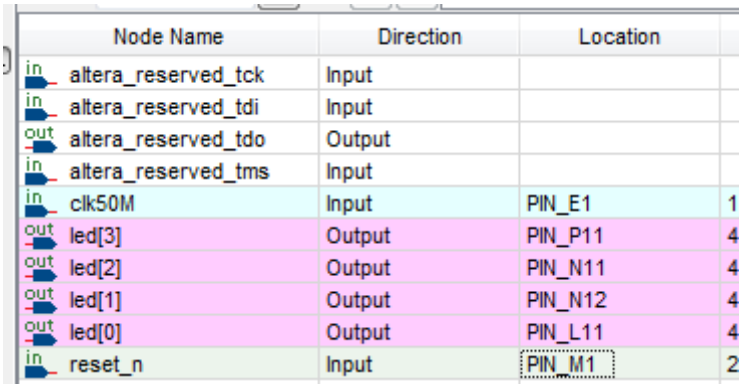


可以看到，引脚分配窗口中已经自动从我们的工程中提取出了需要分配引脚的信号。

根据使用的开发板，完成不同的引脚分配，下表列出了芯航线三款 FPGA 开发板的引脚信息，用户在使用时可以根据自己具体的目标板按照下表，在 Location 栏中输入对应的引脚名称即可。

板卡 / 信号	Starter	AC620	AC6102
clk50M	PIN_E1	PIN_E1	PIN_G1
led[3]	PIN_P11	PIN_A3	PIN_M20
led[2]	PIN_N11	PIN_A4	PIN_M19
led[1]	PIN_N12	PIN_B3	PIN_N20
led[0]	PIN_L11	PIN_A2	PIN_N19
reset_n	PIN_M1	PIN_E16	PIN_L8

最终引脚分配结果如下图所示：



其中前四个信号为 Jtag 接口相关信号，仅在针对 JTAG 接口有特殊应用时才需要进行引脚分配，一般情况无需分配。

分配完成后，直接关闭引脚分配对话框即可（关闭即可自动保存）。

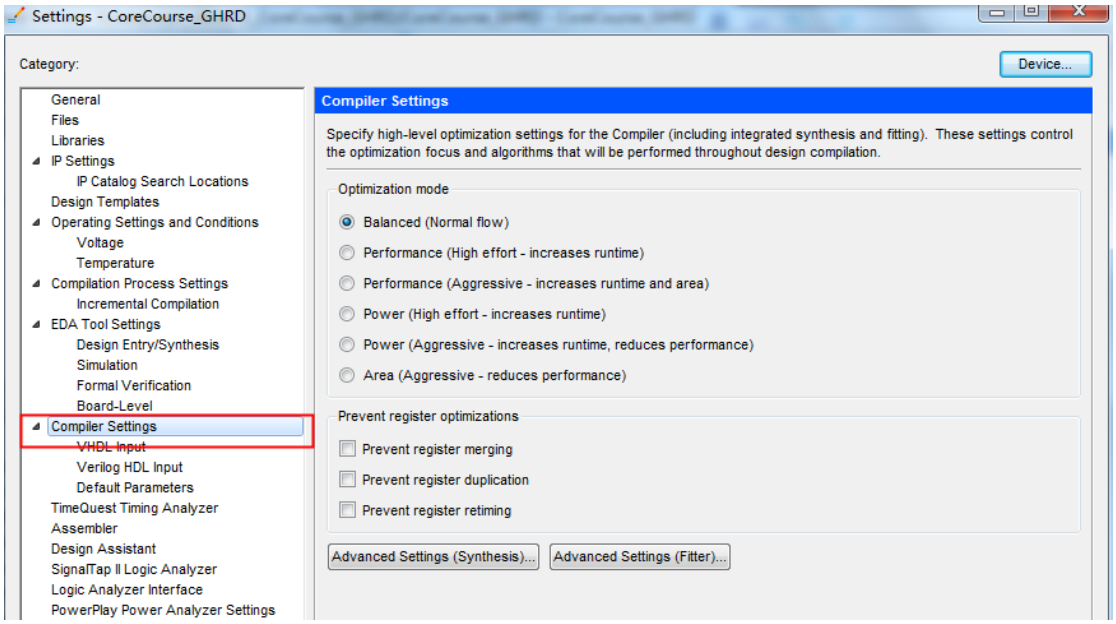
2.7 编译系统

通过 Quartus II 编译器，可以检查设计错误，综合逻辑，并且为仿真、时序分析及器件

配置生成输出文件。

2.7.1 设置编译选项

在下图中，可以通过选择相应的设置选项进行设置。来决定对当前工程的各种编译选项。本实例采用默认的编译设置，不进行任何修改。若要进行编译选项设置，请参考 Altera 的 Quartus II Handbook。



2.7.2 编译硬件系统

在编译过程中，编译器定位并处理所有工程文件，生成与编译相关的消息与报告，创建 SOF 文件及任何可选配置文件。

选择 Processing—>Start Compilation 进行全编译，也可以直接点击  按钮或者直接按键盘组合键 Crtl + L 来启动全编译。在编译系统是，状态窗口显示整个编译进程及每个编译阶段所用的时间。编译结果显示在 Compilation Report 窗口中，整个编译时间为几分钟到几十分钟，具体耗时取决于计算机性能以及编译选项设置。

在编译过程中，如果添加了内部存储器，会使用 onchip_ROM.hex 文件对 onchip_ROM 进行初始化。

在编译过程中，可能产生很多警告信息，但这些不会影响设计结果。

2.7.3 查看编译报告

编译结束后，在 Compilation Report 窗口显示编译结果，包括编译报告，综合报告，适配报告，时序分析报告等。由于本设计比较简单，器件适配，静态时序等情况很容易满足要求，所以在此不详细分析这些报告。

2.8 下载硬件设计到目标 FPGA

成功编译硬件系统后，将产生 CoreCourse_GHRD.sof 的 FPGA 配置文件输出。本节简单介绍将 SOF 文件下载到目标 FPGA 器件的步骤。

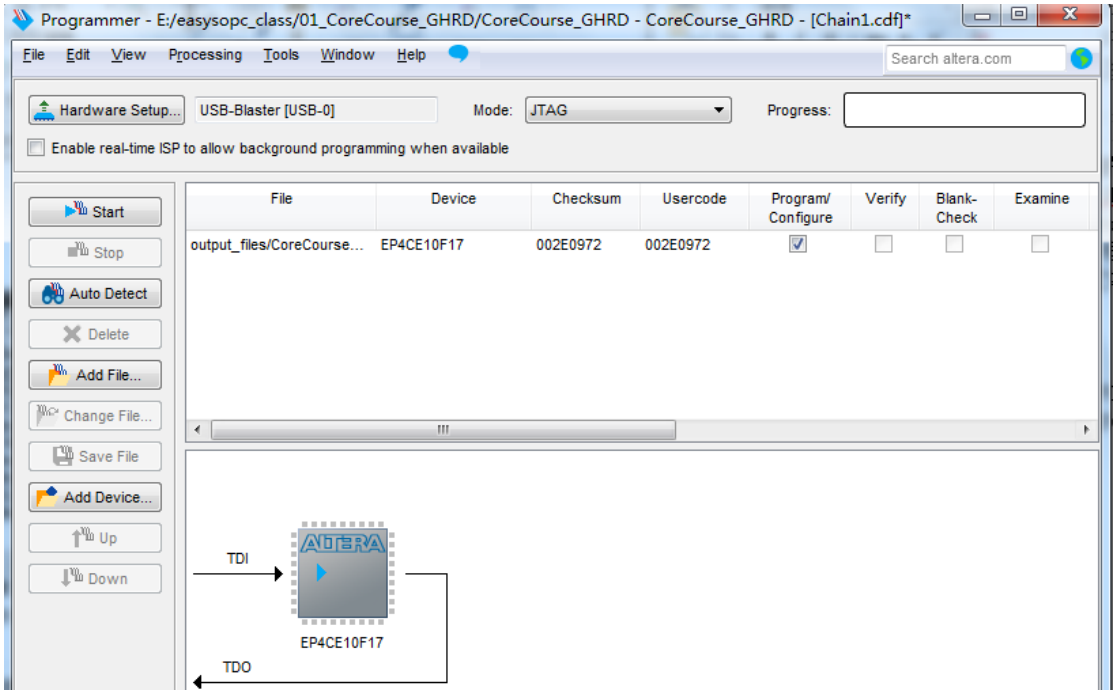
(1)通过 USB Blaster 下载电缆连接目标板和主计算机，接通目标板电源。

(2)在 Quartus II 软件中选择【Tools】->【Programmer】，打开编程器窗口并自动打开配置文件（CoreCourse_GHRD.sof），如下图所示。如果没有自动打开配置文件，则须添加需要编程的配置文件，点击 Add File，然后选择 output files 文件夹下的 CoreCourse_GHRD.sof 文件，选择 open 即可。

(3)确保编程器窗口的 Hardware Setup 栏中硬件已经安装。

(4)确保 Program/Configure 下的方框选中。

(5)单击 Start 按钮，开始使用配置文件对 FPGA 进行配置，Progress 栏显示配置进度



本节只讲述了将配置文件下载到 FPGA 中，掉电后 FPGA 中的配置数据将丢失，可以将配置文件写入掉电保持的 EPCS 器件。在上电时使用 EPCS 对 FPGA 进行配置，详细内容见《芯航线 FPGA 数字系统设计教程+实例解析》中第一部分“十一、芯航线 FPGA 学习套件配置文件固化教程”内容。

2.9 使用 NIOS II IDE 建立用户程序

本节中，将启动 Nios II 集成开发环境来建立和编译一个简单的 LED 灯闪烁的 C 语言程序。

针对 NIOS II 软核，要开发一个能够正常运行在目标板上的程序，需要同时具备用户应用工程（Application Project）和板级支持包工程（BSP Project）。

其中板级支持包工程（BSP Project）是一个与硬件系统紧密联系的工程，其使用我们创建 Qsys 系统时生成的.sopcinfo 文件。BSP 工程创建时会根据.sopcinfo 文件中定义的 NIOS II 系统的硬件信息，生成对应的设备驱动（drivers）、HAL 库（HAL）以及硬件信息头文件

（System.h）。

2.9.1 创建一个新的 C/C+ +应用工程

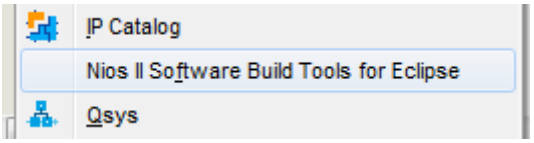
启动一个新的 C/C++应用工程时，Nios II IDE 需要使用 Qsys 系统文件 (.sopcinfo)。在目标硬件上运行和调试应用工程之前，软件设计者需要先使用 FPGA 配置文件(.sof)配置 FPGA。IDE 可产生下面列出的几个输出，但不是所有的工程都要求这些输出。

- ✧ System.h 文件——为系统中的硬件信息进行宏定义，帮助软件设计者处理硬件潜在的变化性，软件设计者可以使用这些宏定义而不是具体的硬件信息（如地址值、中断号 等）。该文件也可用于查阅系统中的硬件。创建一个新板级支持包（BSP）工程时，1DE 自动生成该文件。
- ✧ 可执行的连接文件（.elf)—是编译 C/C+ +应用工程的结果，可直接将它下载到 Nios II 处理器。
- ✧ 存储器初始化文件(.hex)——如果添加的片内存储器，片内存储器可在上电时预定义存储器的内容。IDE 可生成片内存储器的初始化文件，该存储器支持初始化的内容。
- ✧ Flash 编程数据——IDE 包括 Flash 编程器，利用 IDE 的 Flash 编程器可以写程序到 Flash 存储器。用户也可以使用 Flash 编程器来写任意二进制数据到 Flash 存储器。

本章仅讨论直接下载.elf 文件到 Nios II 系统。关于使用 IDE 中 Flash 编程器进行编程的详细内容请参考本书相关章节。

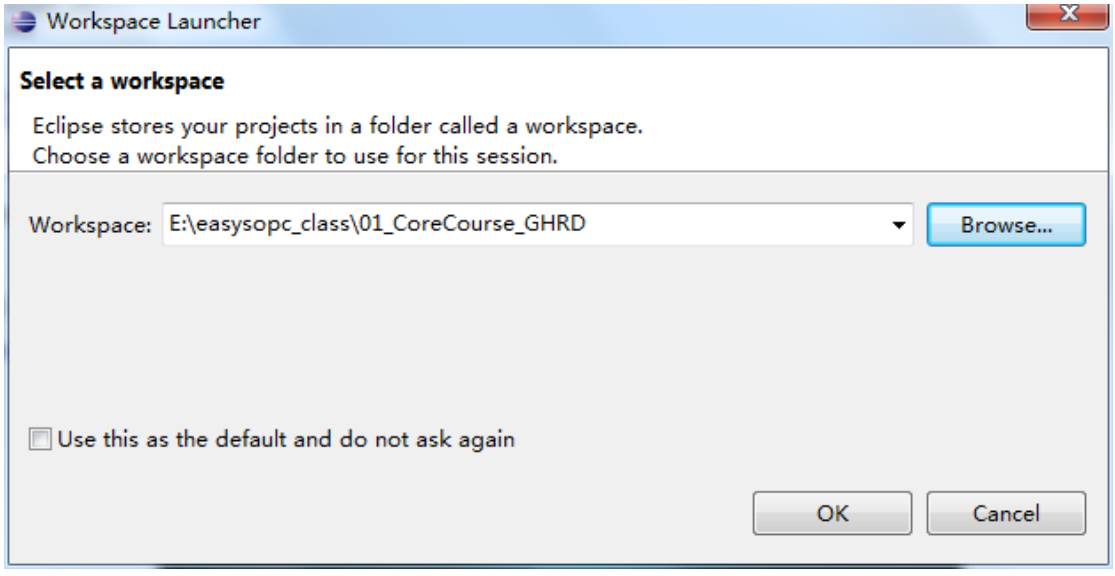
执行下面的步骤来创建一个新的 C/C+ +应用工程：

(1) 在 Quartus II 软件菜单栏中，依次点击【Tools】->【Nios II Software Build Tools for Eclipse】启动 Nios II IDE。



(2)在弹出的 Workspace Launcher 对话框中，单击

按钮，设置工作空间为 Quartus II 工程的文件夹(E:\easysopc_class\01_CoreCourse_GHRD)，如图 2.41 所示，这样便于管理。然后点击 OK 进入 Nios II IDE 编辑界面。



(3)在该界面中选择 File—New—Nios II Application and BSP from Template(图 2. 42)，打开

新建工程向导，如图 2.43 所示。

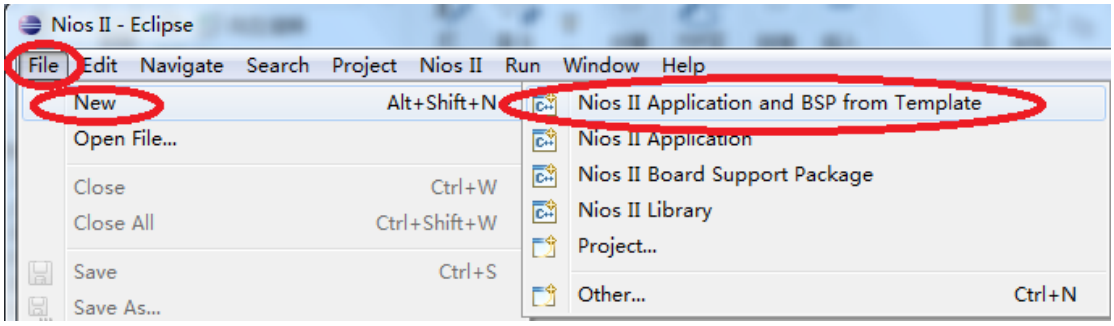


图 2. 42 打开新建 C/C+ +工程向导

(4)单击 SOPC Information File name 选项区中的 ... 按钮，打开选择文件对话框，选择当前工程目录下的 mysystem.sopcinfow 文件，（注意，一定要核对当前 sopcinfo 文件的路径是否正确，软件会默认记录上次打开的 sopcinfo 文件路径，如果不注意这一点，会经常容易出错）。选择完成后，软件会自动识别当前硬件系统中的 CPU 名称，如下图所示：

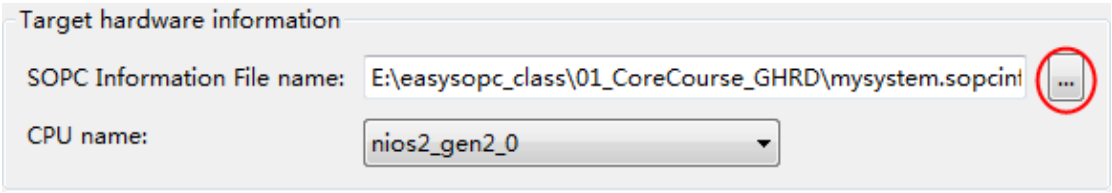


图 2.44 选择硬件目标文件

(5)选择 Project Template 列表框中的 Hello World。并在 Project Name 栏中输入 hello_led_0，选中 Use Dafault Location 复选框，单击 Finish 按钮，完成工程创建。

向导中的 Select Project Templates—栏中是已经设计好的软件工程，用户可以选择其中的一个，把它当作模板来创建自己的工程。当然也可以选择 Blank Project(空白工程)，完全由 用户写所有的代码。本例中选取了 Hello World 工程，然后在此基础上进行适当的修改，一般情况下这比空白工程更加容易，也更方便。

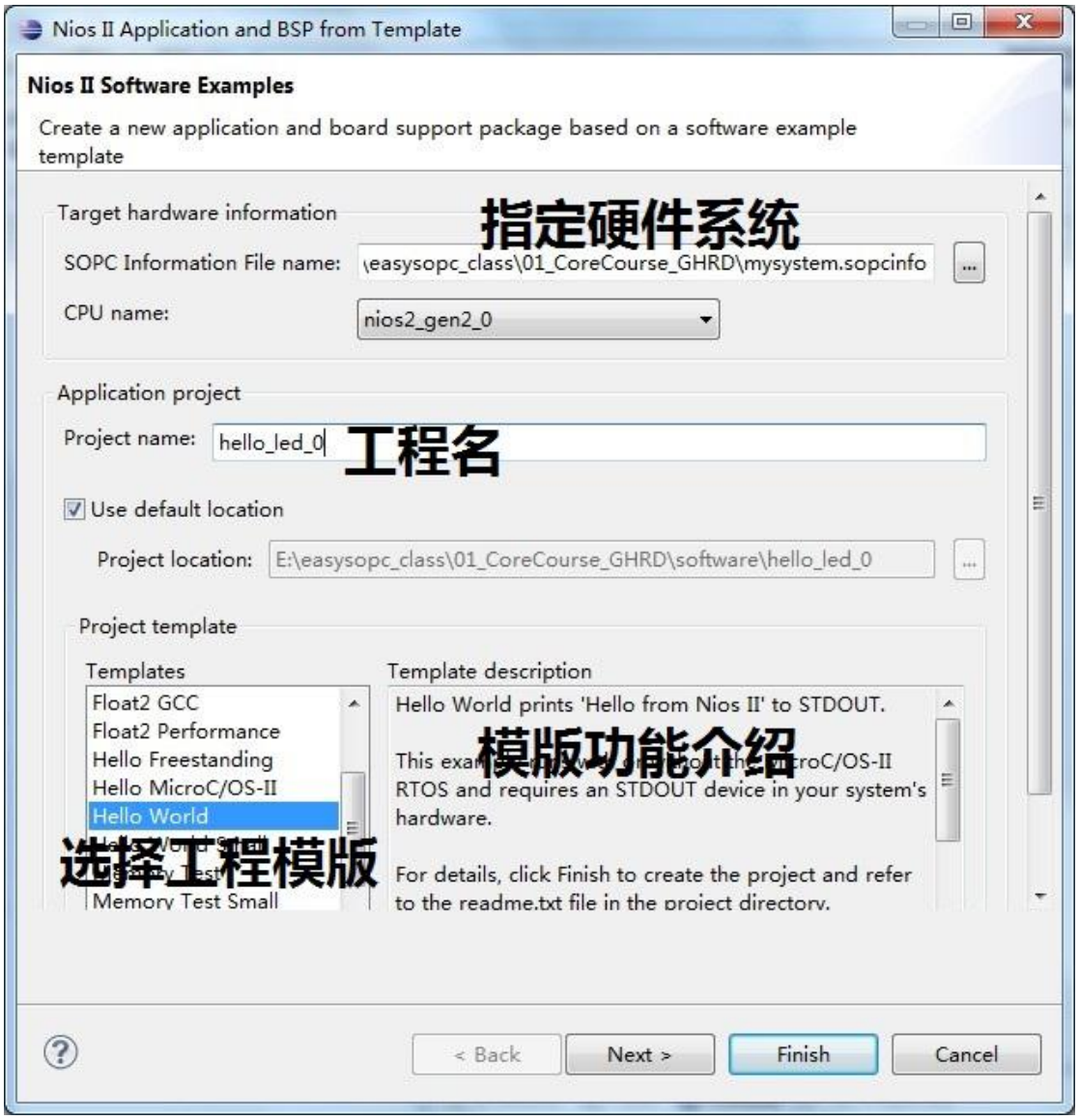
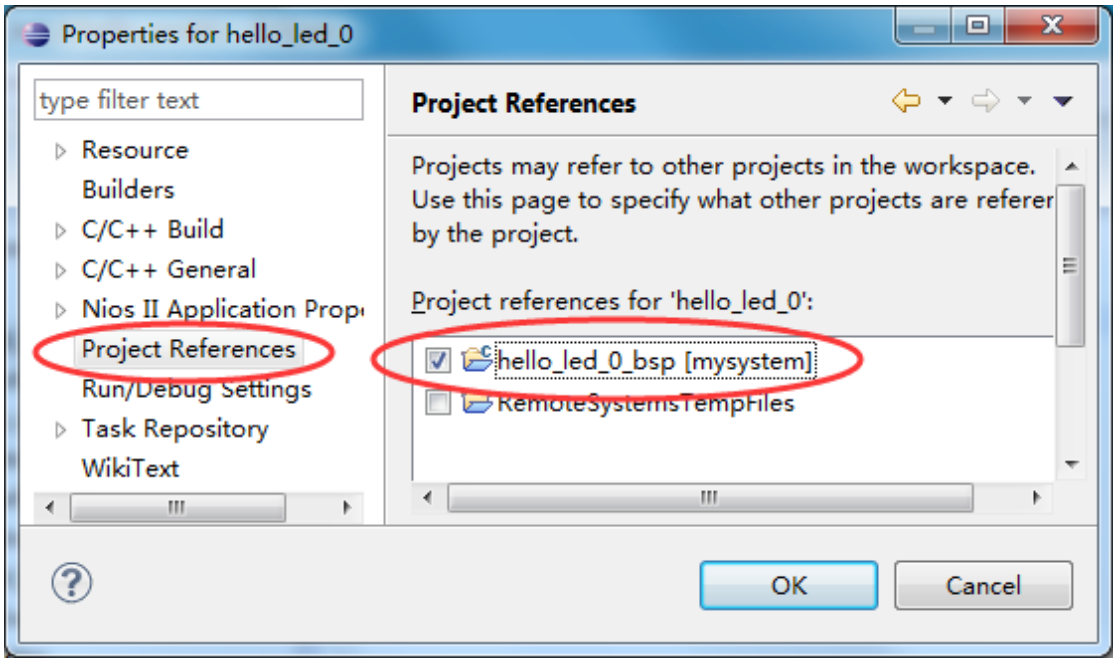


图 2.45 完成设置后的工程向导

创建工程后，在 Nios II IDE 主界面左侧的 C/C++ Projects 选项卡中显示两个新的工程:hello_led_0 和 hello_led_0_BSP。hello_led_0 是 C/C++应用工程，而 hello_led_0_BSP 是描述 CoreCourse_GHRD 系统硬件细节的系统库。应用工程的各种底层细节必须依赖对应的 bsp 工程。

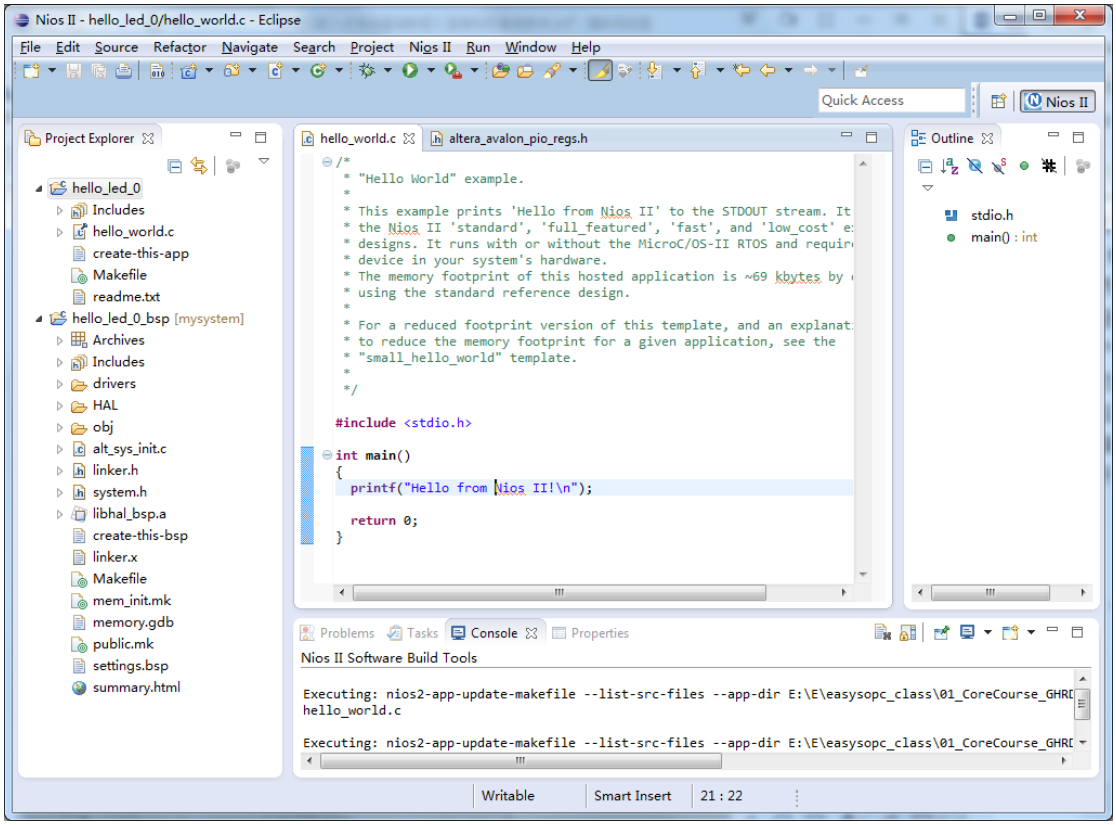
需要注意的是，Nios II IDE 默认是无法将一个应用工程和一个 bsp 工程对应起来的，那么所存在的问题就是，每当用户关闭该工程后重新打开时，两个工程并不能关联上，因此出现应用工程中各种报错找不到头文件和硬件地址定义。因此我们需要手动设置两个工程的关联性。鼠标选中 hello_led_0 工程，右击，选择“Properties”，或者鼠标选中 hello_led_0 工程后，按下键盘组合键“Alt+Enter”，在打开的属性设置窗口中，鼠标左击 Project References，在右侧的列表中，将 hello_led_0_bsp 前面的复选框勾选上，然后点击 OK 按钮以退出。这样应用工程和 BSP 工程就相互关联起来了，不会再存在找不到各种硬件相关头文件和硬件地址定义的情况，如下图所示。



Nios II IDE 中每个工作界面都包括一个或多个窗口，每个窗口都有其特定的功能。在工作界面中包括的主要窗口有编辑器窗口和一个或多个浏览器。编辑器可用于打开并编辑一个工程。浏览器用于对编辑器提供各种支持，可由用户根据需要进行选择。

用户可以同时打开多个编辑器，但在同一时刻只能有一个编辑器处于激活状态。在工作界面上的主菜单和工具条的各种操作只对处于激活状态的编辑器起作用。在编辑区中的各个标签上是当前被打开的文件名，带有“*”号的标签表示这个编辑器中的内容还没有被保存。在编译工程前，要养成先对各文件进行保存的好习惯。

打开 hello_led.c，可以看到模版工程默认设计了一个使用串口发送欢迎信息的程序，如下图所示。由于本例在 Qsys 系统中并未为 NIOS II CPU 添加串口相关的外设，因此该工程实际将是无法运行的。这里使用下述程序清单中的内容替换 hello_led.c 中的内容，然后保存。

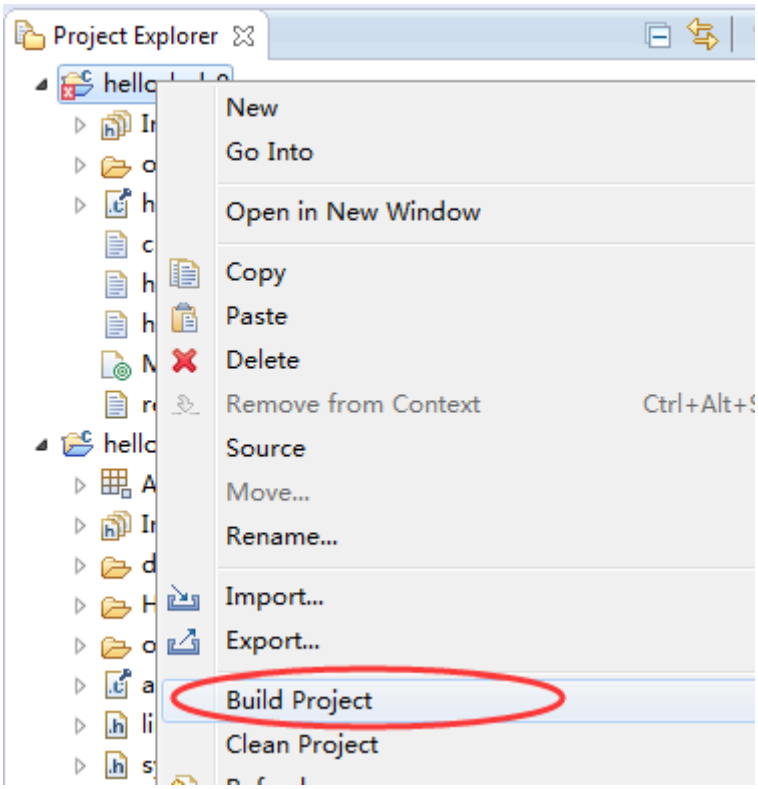



```c
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

int main(void)
{
    alt_u8 led = 0;
    alt_u32 i;
    while(1)
    {
        led = 0;
        IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, led);//点亮LED灯
        i = 0;
        while(i<500000)//延时功能，非精确延时，数值太小将无法看到灯的闪烁效果
        {
            i++;
        }
        led = 1;
        IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, led);//熄灭LED灯
        i = 0;
        while(i<500000)//延时功能，非精确延时，数值太小将无法看到灯的闪烁效果
        {
            i++;
```

```
        }
    }
    return 0;
}
```

替换以后，如果代码中有错误，请手动修正，一般编辑器会在出错的地方用显眼的红色波浪线提示，将光标放置在报错位置，也会提示更加具体的报错信息。

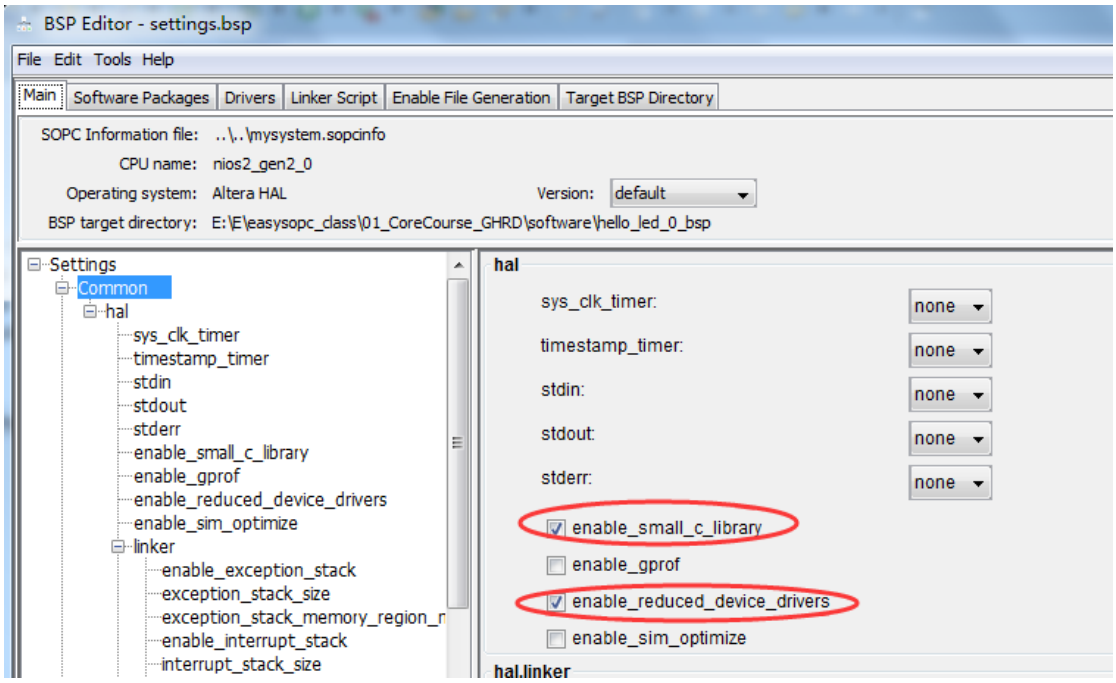
当完成替换且代码正确无误之后，就可以对整个软件工程进行编译了。可以选中应用工程 hello_led_0 然后鼠标右击，在弹出的选项中选择 Build Project。或者左击选中工程后，按键盘组合键“Ctrl + B”以开始编译。



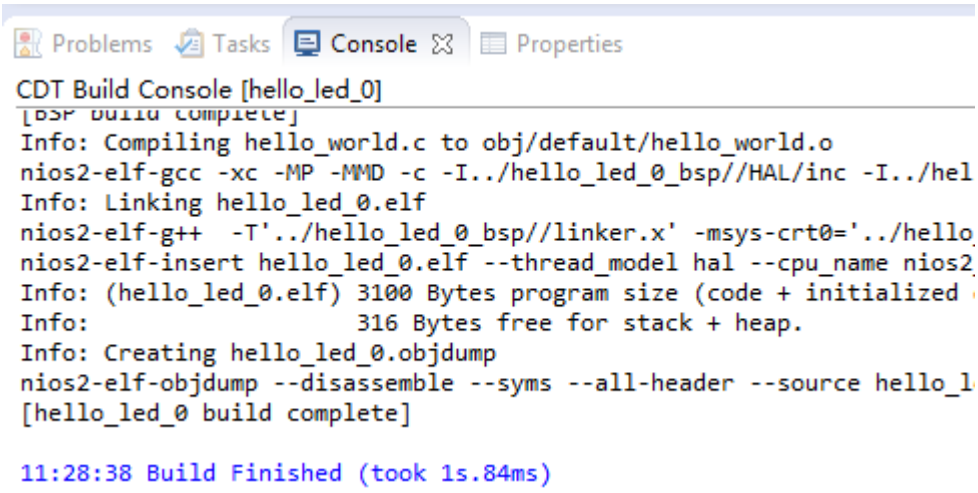
正常情况下，编译器会完成编译并生成 elf 格式的可执行文件，但是如果代码有问题或者系统设置有问题导致编译无法正常完成，将无法生成可执行文件。相应的，软件会在下方的 Console 窗口显示出错的相关信息。例如本软件工程当前就无法正常编译并得到可执行文件。在 Console 窗口中提示信息如下：

```
'ld.exe: address 0x3524 of hello_led_0.elf section `.bss' is not within region `onchip_RAM'
'ld.exe: address 0x3524 of hello_led_0.elf section `.bss' is not within region `onchip_RAM'
```

address 0x3524 of hello_led_0.elf section `.bss' is not within region `onchip_RAM'这句话的意思是说程序中使用到的一个地址 0x3524 已经超出了 onchip_RAM 的最大存储空间，因此无法正确存取。所以无法最终生成可执行文件。此时，我们可以通过精简一些不必要的外设驱动参与编译，以及使用小体积的 C 语言支持包来减小我们的代码体积。设置方法非常简单，鼠标选中 hello_led_0_bsp 工程，右击，在弹出的菜单中选择 Nios II，再在弹出的选项中选择 BSP Editor，打开 BSP 设置界面，在界面中，勾选上 enable_small_c_libray 和 enable_reduced_device_drivers 上即可。这两个选项中，enable_small_c_libray 是设置使用小体积的 C 语言库，而 enable_reduced_device_drivers 则是设置减小一些外设驱动的支持。设置完成后，点击 generate 按钮，再点击 Exit 按钮即可退出。

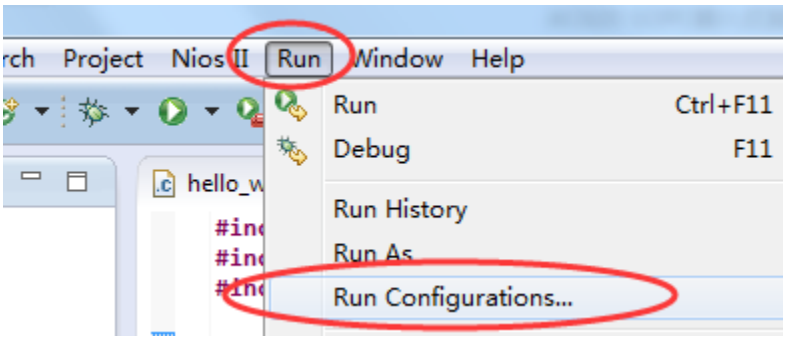


以上设置完成后，再次对工程进行编译，就能过编译通过了。编译结果如下图所示，可以看到，程序总共大小为 3100 字节，存储器还剩下 316 字节可用。

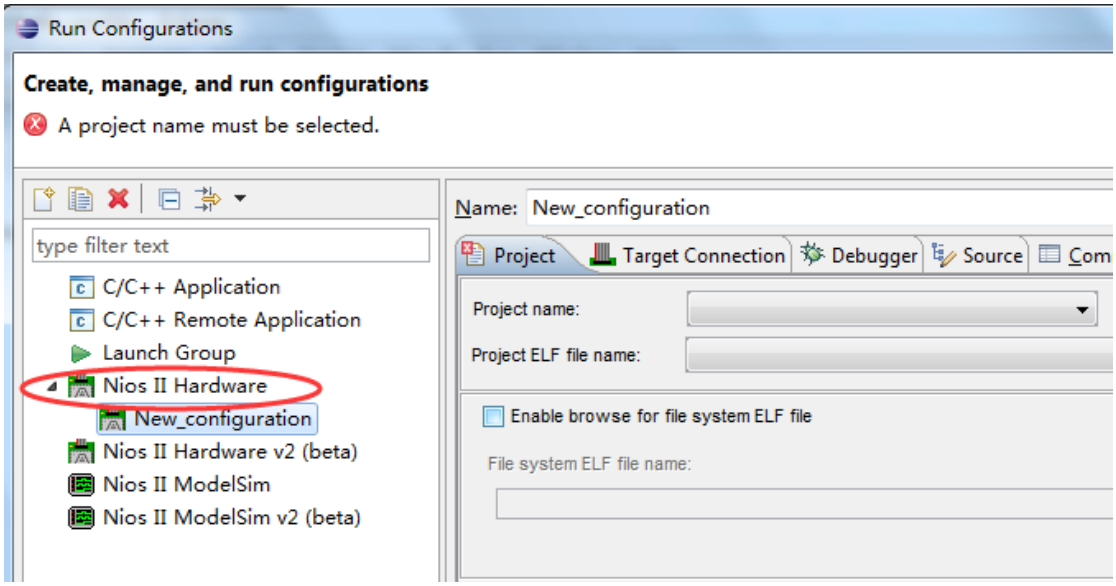


2.10 调试运行程序

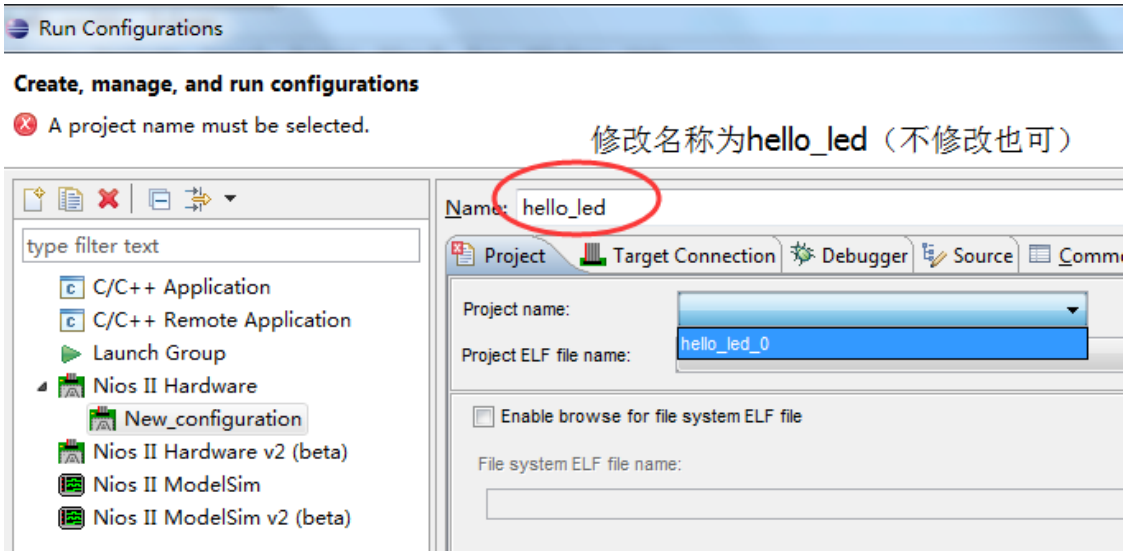
如下图所示，编译连接成功后，在软件菜单栏中依次点击【Run】—>【Run Configurations】，打开运行配置窗口。



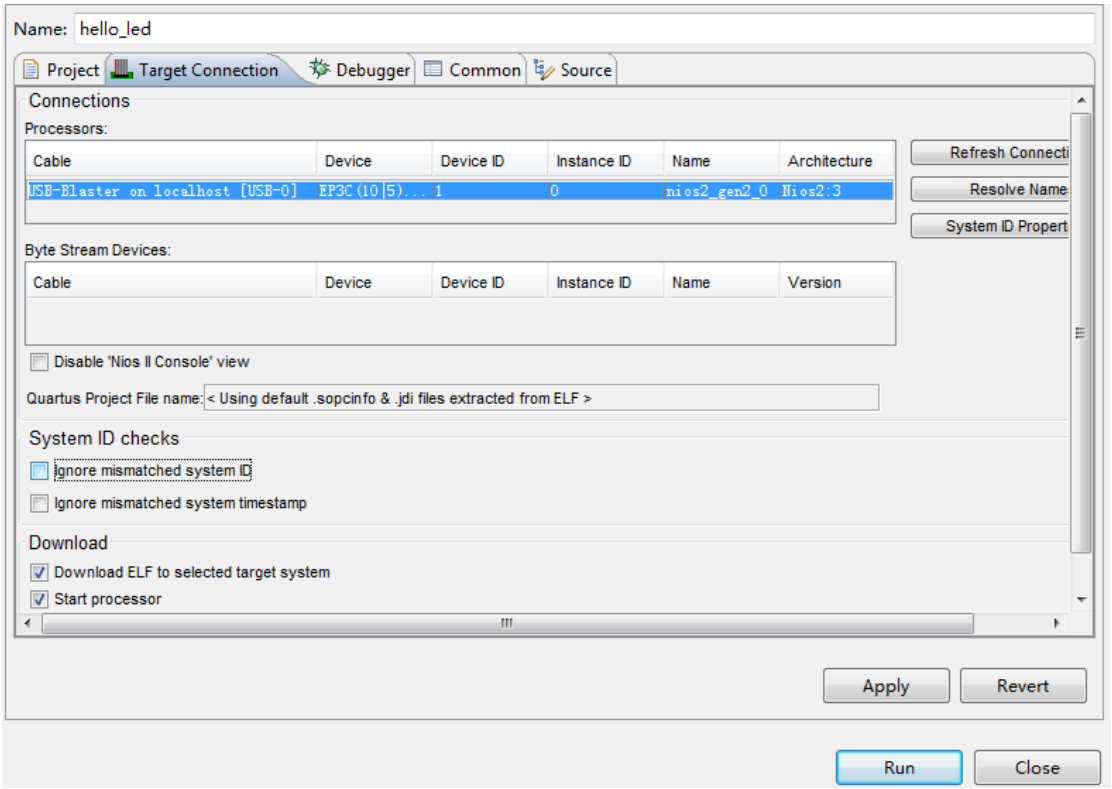
在配置窗口中，双击 Nios II Hardware，新建一个 New_configuration 选项，如下图所示。



在 Name 一栏，可以根据自己的习惯，修改名称为工程名称，当然，这里不修改也是可以的。然后在 Project name 中下拉选择我们希望运行的工程 hello_led_0。



然后切换到 Target Connection 选项卡，查看 Processors 栏中是否有 USB Blaster 选项，如果没有显示，可以点击右侧的 Refresh Connections 以刷新 Blaster 链，如果还是没有显示，请尝试重新下载 FPGA 的 sof 文件，然后再 Refresh Connections。

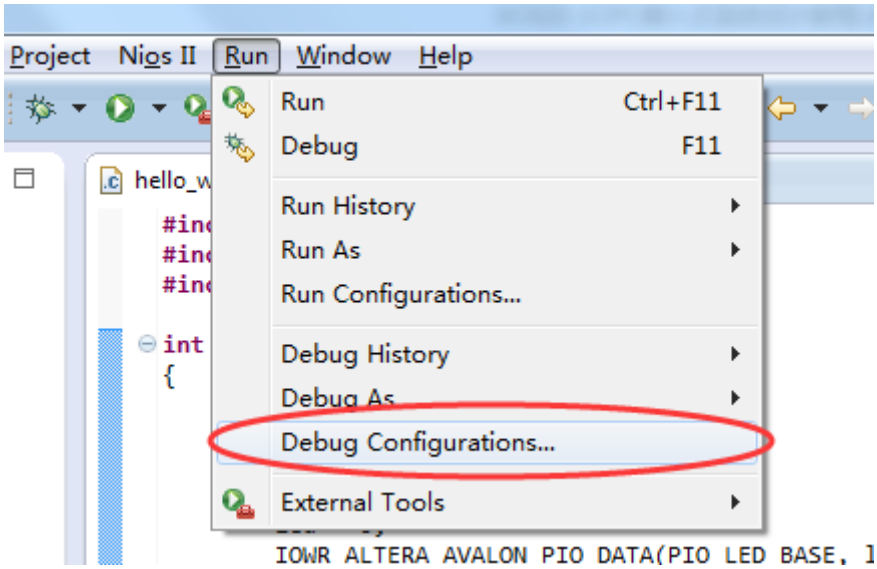


当以上条件都 OK 后，点击 Apply，然后就可以点击 Run 按钮进行下载程序了。程序下载结束后，就可以看到目标开发板上的 LED0 以大约 1 秒的时间亮灭闪烁了。

至此，整个设计过程已经结束。如果硬件配置成功后，板上的用户指示灯 LED1 并没有闪烁，说明用户程序存在问题，可以通过调试来解决。

2.11 在目标板上调试程序

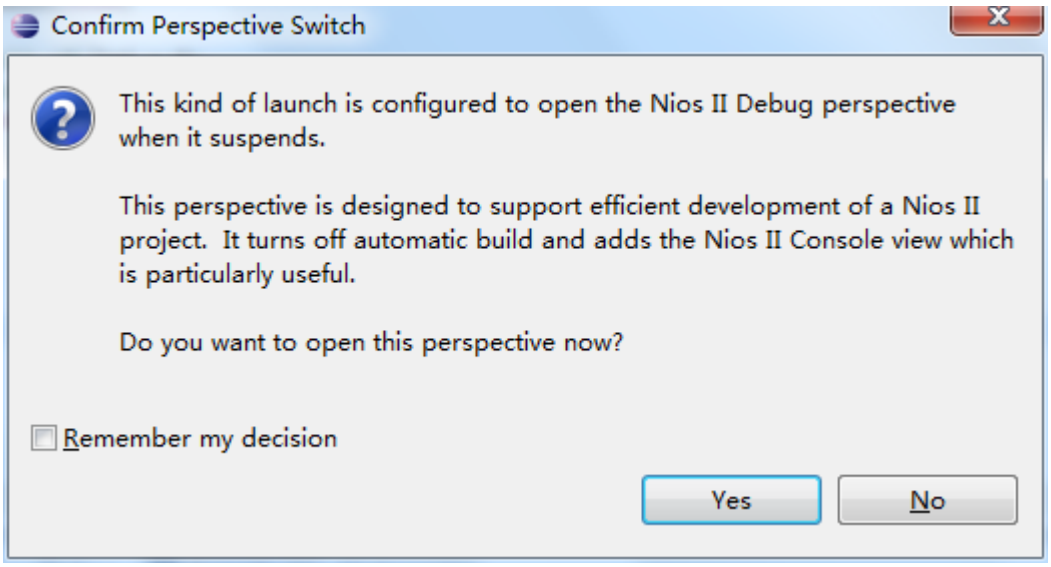
启动调试程序和启动运行程序类似如下图所示，编译链接成功后在菜单栏依次点击【Run】->【Debug Configurations】



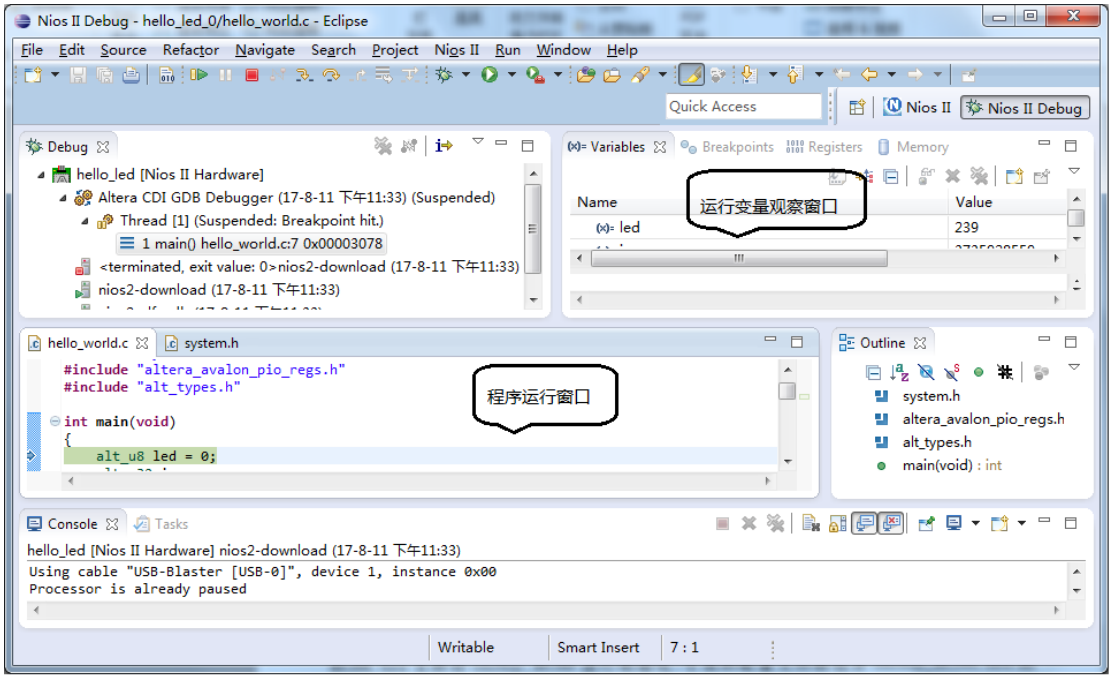
打开 Debug Configurations 配置界面后，选择之前创建好的 hello_led 选项，然后点击

Debug 按钮即可开始下载程序。

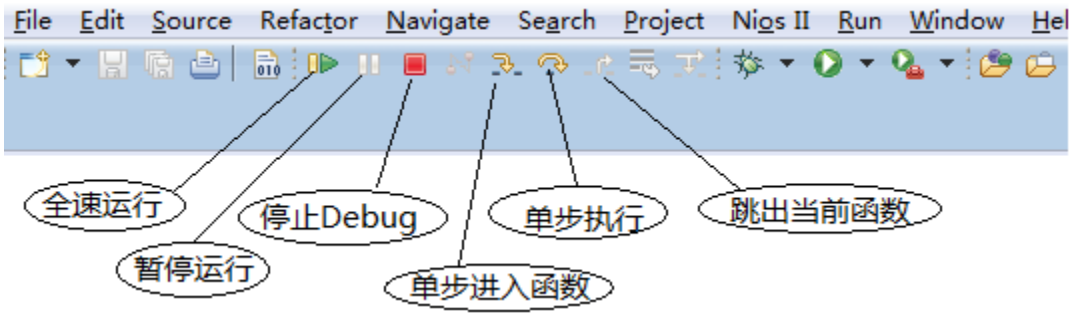
程序下载完成后，会弹出如下界面，点击 Yes 按钮即可进入调试界面，



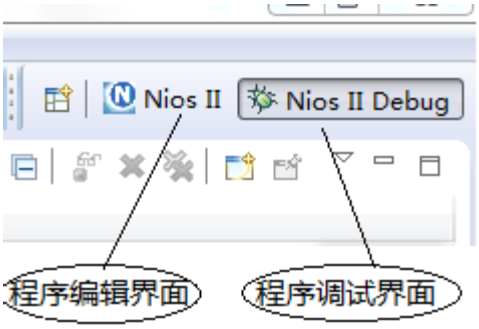
进入调试界面后，程序会自动运行到 main()函数的第一行停下来，等待用户开始进行调试。



用户可以选取下图所示的通用调试控制工具来跟踪程序的运行。在程序运行观察窗口中可以观察程序的运行情况；在运行变量观察窗口中可以观察变量的值。此外还能够查看寄存器、存储器中的值。并对程序中已经设置的断点使能和旁路。



当用户希望回到最开始的程序编辑界面时，点击软件右上角的 Nios II 图标即可切换回程序编辑界面，如下图所示。



2.12 固化系统和软件到目标板

当程序设计并调试完成，需要固化到目标板上长期运行时，需要将 FPGA 的配置文件（.sof）和 NIOS II 的软件代码同时烧写到 FPGA 的配置器件中。在本例中，使用了 FPGA 的片上存储器构建 RAM 和 ROM，因此需要将 NIOS II 的软件代码生成 RAM 和 ROM 的初始化文件（.hex），然后添加到 Quartus 工程中，由 Quartus 软件自动使用这两个 hex 文件初始化 RAM 和 ROM，从而实现 RAM 和 ROM 的初始化，即完成程序的植入。

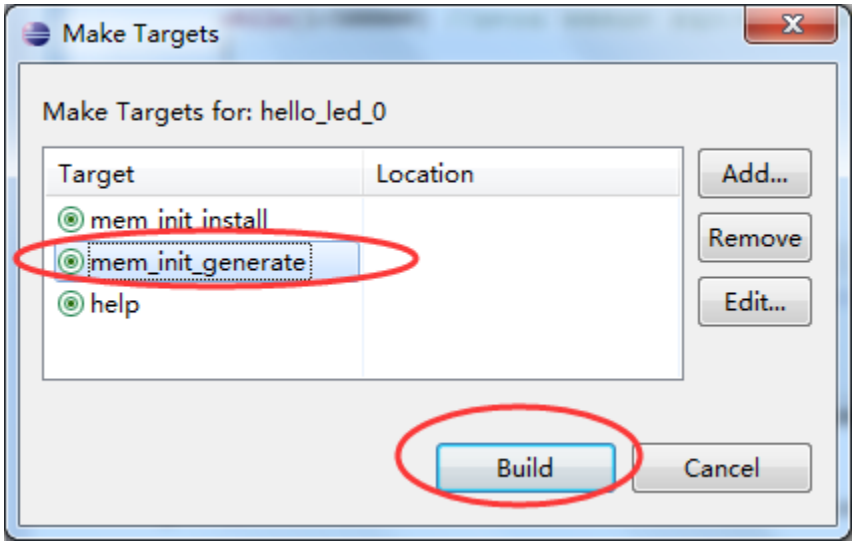
Nios II Software Build Tools for Eclipse 软件默认不会将软件程序生成 hex 文件，需要用户手动操作。操作方法如下：

1、回到程序编辑界面，选择 hello_led_0 工程文件夹

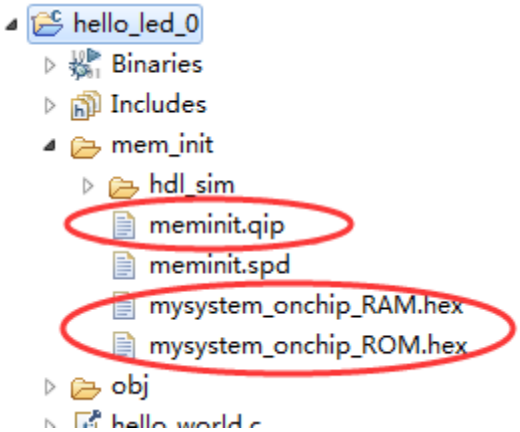
2、右击选择 Make Targets，然后在弹出的下级选项卡中，选择 Build，如下图 1 所示。

3、在打开的 Make Targets 选项卡中，选择 mem_init_generate，然后点击 Build，如下图 2 所示。





执行完以上三步后，在 hello_led_0 文件目录下会生成一个叫 mem_init 的文件夹，下面有一个名为 meminit.qip 和一个 meminit.spd 的文件，以及 mysystem_onchip_RAM.hex 和 mysystem_onchip_ROM.hex 两个文件。两个.hex 文件即分别为 RAM 和 ROM 的初始化文件，qip 文件包含了当前文件的路径信息，spd 文件保存了当前路径下需要添加到 Quartus 工程中文件的详细信息。



通过以上工作，就完成了程序文件到 RAM 和 ROM 初始化文件的转化，接下来，回到 Quartus 工程中，添加 meminit.qip 文件到 Quartus 工程中。然后全编译工程，得到 sof 文件，下载 sof 文件到目标板中，就可以看到目标板上的 LED0 开始以大约 1 秒的时间亮灭闪烁了。如果需要固化程序并掉电不丢失，可以在 Quartus 软件中将该 sof 文件转化为 jic 文件，烧写到 EPCS 存储器中，就可以让目标板上的 FPGA 上电自动从 EPCS 器件中读取 FPGA 配置数据（包括 NIOS II 软件程序），然后正确运行了。

第 3 章 基于 NIOS II 处理器的 UC/OS II 操作系统设计实例

温馨提示：本文可配合小梅哥 2017 培训班实录视频《0825_02-(SOPC)定时器与操作系统应用》内容加深理解。

这里以在小梅哥团队出品的 AC620 型 FPGA 开发板上搭建 NIOS II 软核并运行 UCOS II 操作系统为例介绍 SOPC 的开发流程。

第一步：建立 Quartus II 工程

建立 Quartus II 工程时需要注意以下几点

1、 器件选择为 EP4CE10F17C8N；

2、 工程路径中不得出现非法字符（空格和中文字符）；

3、 开发工具选择 Quartus II 11.0 及以上，这里我选择的版本为 Quartus II 15.1。

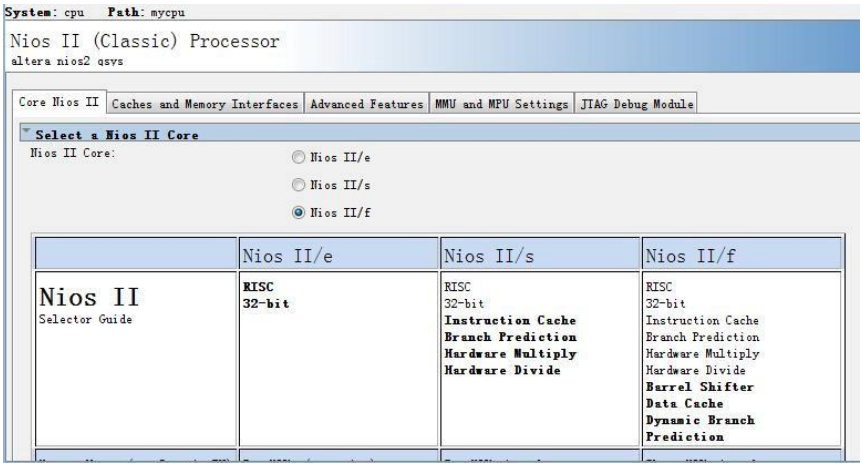
4、 由于我们最终的软件工程也是建立在 Quartus II 工程目录下的，而软件工程有一个最让人崩溃的地方就是，软件工程在建立时，很多包含文件是以据对路径存在的，如果你更改了工程路径，那么当你再次编译工程时，就会报错。所以这里我在建立工程时，直接在 E 盘根目录下建立了一个 mysystem 的文件夹，并将工程建立在此文件夹中。这样可以避免路径太深，以后切换电脑时，只需要将工程直接拷贝到新的电脑的 E 盘下即可。只要 QII 软件安装位置相同，再次编译软件工程就不会报错。

第二步：打开 Qsys 工具

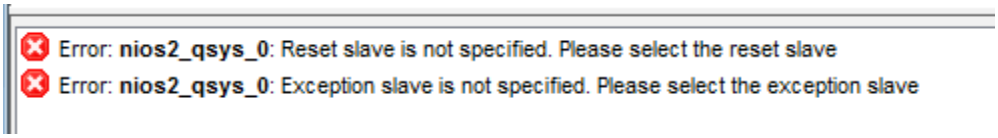
在 Quartus II 软件中，依次点击 tools/Qsys，以打开 Qsys 工具。

第三步：在 Qsys 系统中添加 NIOS II 处理器

在左侧的 IP Catalog 的搜索框中，输入 nios，在搜索结果中，选择并添加 NIOS II（Classic）Processor 到右侧的 System Contents 中来。添加时，选择 NIOS II/f 型 CPU。添加完成后，在 System Contents 中修改其名字为 mycpu。



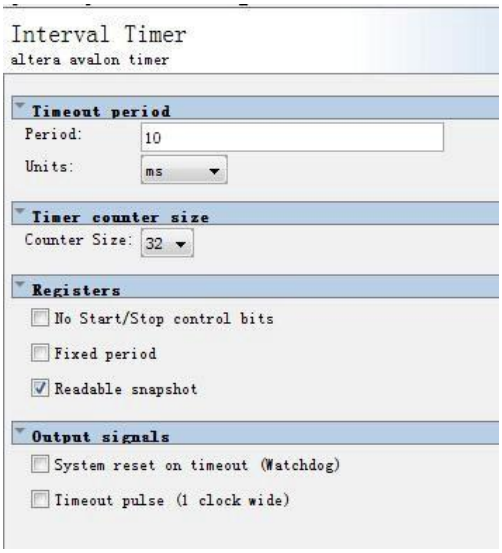
添加完成后，会报如下错误



这个是因为目前只添加了 CPU，还没有添加对应的存储器，因此会报错。这里不用理会，在后面的步骤中，只要按照操作来，这些错误最好都会消失的。后续操作中如果遇上其他错误也不要惊慌，最后只要系统搭建完成，错误都是会消失的。

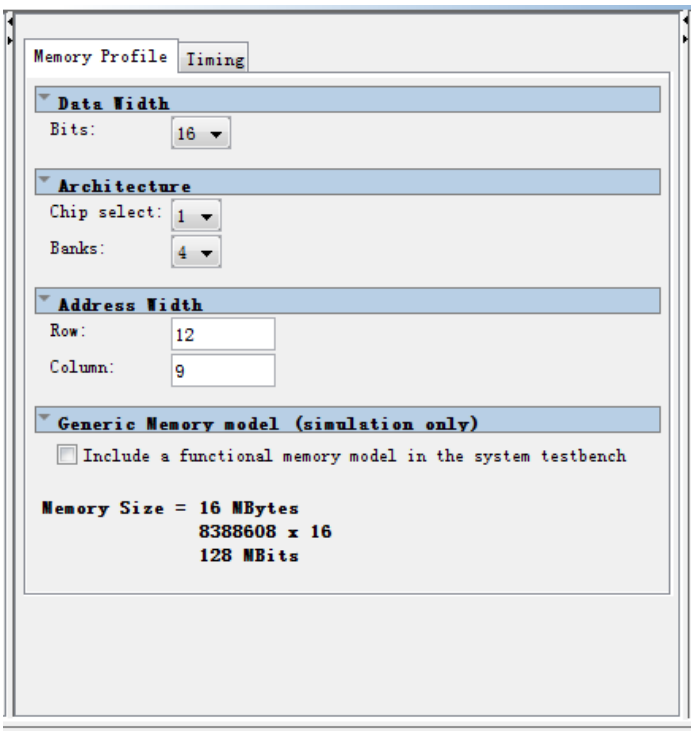
第四步：在 Qsys 中添加 UCOS 系统所需定时器

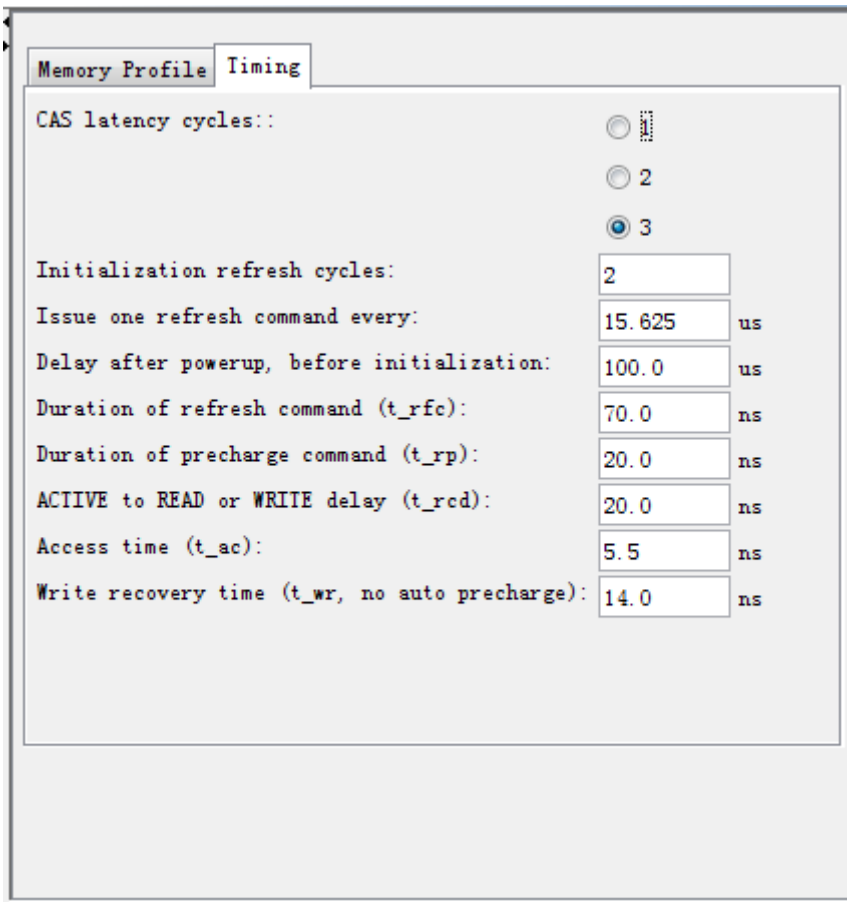
在左侧的 IP Catalog 的搜索框中，输入 timer，在搜索结果中，选择并添加 Interval timer 到右侧的 System Contents 中来。添加时，将 timeout period 设定为 10ms。添加完成后，修改其名字为 ucosii_timer。



第五步：在 Qsys 中添加 SDRAM

在左侧的 IP Catalog 的搜索框中，输入 sdram，在搜索结果中，选中并双击 SDRAM Control，在打开的参数配置选项卡中，按照如下图所示的配置进行设置：





添加完成后，修改其名字为 sdram。

第六步：在 Qsys 系统中添加 JTAG_UART

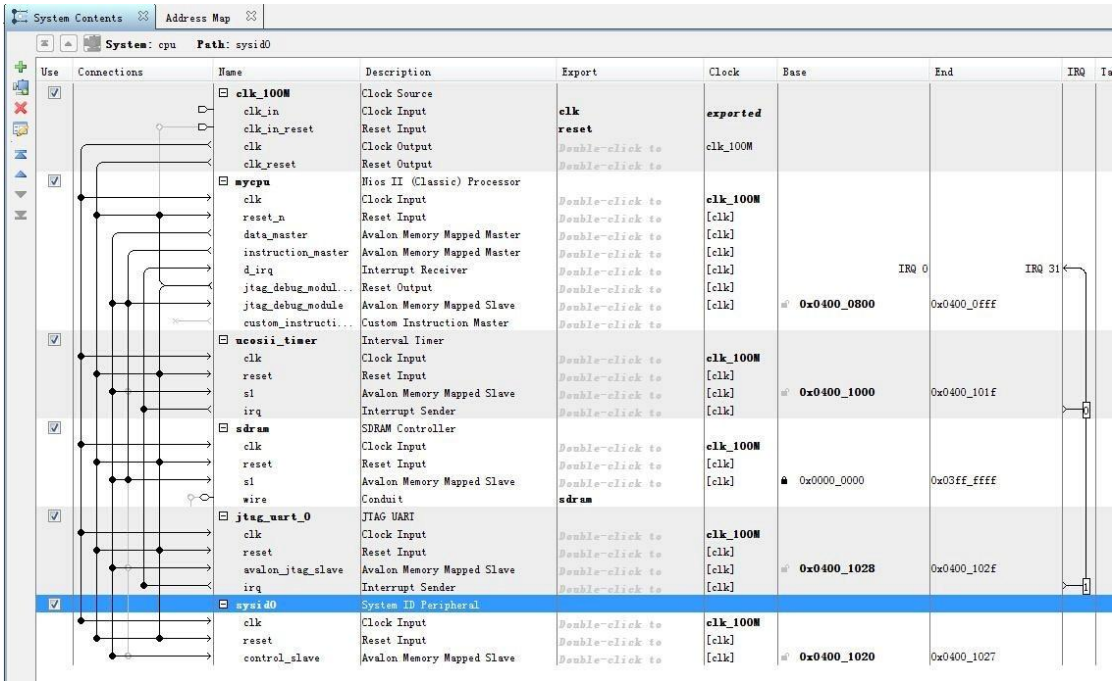
在左侧的 IP Catalog 的搜索框中，输入 uart，在搜索结果中，选择并添加 JTAG UART 到右侧的 System Contents 中来。添加时，所有参数默认即可。添加完成后，修改其名字为 JTAG_ UART_0。

第七步：在 Qsys 系统中添加 sysid

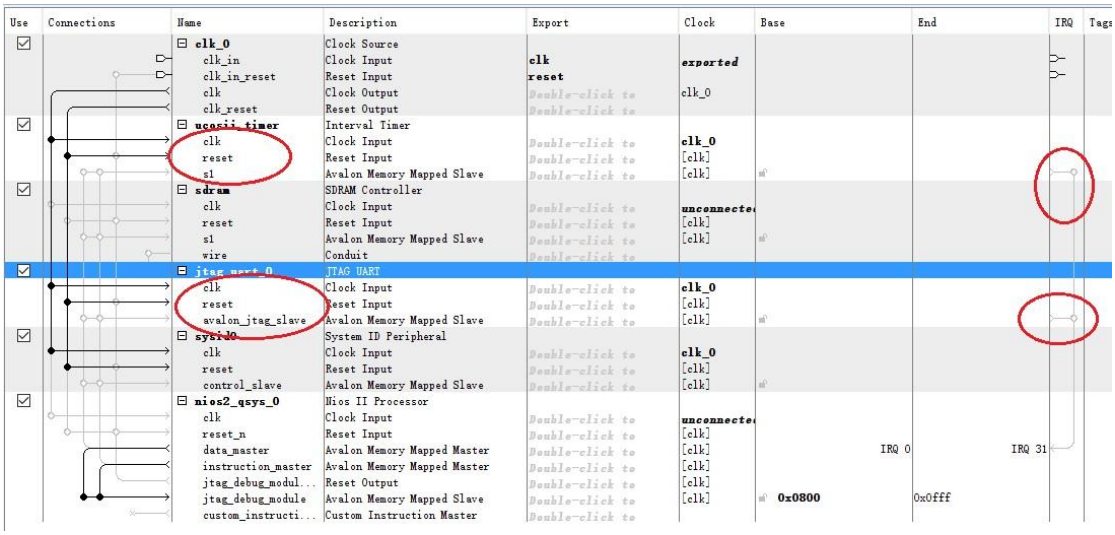
在左侧的 IP Catalog 的搜索框中，输入 sys，在搜索结果中，选择并添加 System ID Peripheral 到右侧的 System Contents 中来。添加时，设定 32bit system id 为自定义值，我这里设定为 0x00000002。添加完成后，修改其名字为 sysid0。

第八步：连线和导出端口

自此，所有的外设添加就已经完成了，接下来，需要将每个模块的端口连接到对应的总线上，这里不一一介绍，详情请参看下图：



其中，clk_100M 模块的 clk_in 和 clk_reset，以及 sdram 模块的 wire 需要 export。将 ucosii_timer 和 jtag_uart_0 的 irq 连接到 mycpu 的 d_irq 上，在右侧的 IRQ 一栏中，ucosii_timer 的中断号设置为 0，jtag_uart_0 的中断号设置为 1。 注意，在 Quartus II13.0 及以前的版本中，所有部件第二列 name 栏中并不会有 d_irq 或 irq 这样一个信号，这个无需在意，只是没显示而已。只需要直接点击右侧的 IRQ 一栏中的空心圈并输入序号即可完成中断的分配，如下图所示：



第九步：设定 sdram 基地址

在 Base 一栏中，将 sdram 模块的起始地址设置为 0，然后点击地址前面很小的那个锁的符号，使其变为锁定状态。

第十步：自动分配所有模块地址

在 Qsys 系统中，依次点击菜单栏中的 sysytem/Assign Base Address，系统即可自动为我们分配各个模块的基地址。由于 sdram 的基地址本身处于锁定状态，所以在此过程中，可能其他模块的基地址会因为自动修改而发生变化，但是 sdram 的基地址将保持不变。

第十一步：设定 CPU 的复位向量地址和异常向量地址

双击 mycpu 模块，在打开的参数设置中，将 Reset Vector 和 Exception Vector 都选择为 sdram.s1。

以上，我们就完成了完整 Qsys 系统的搭建，接下来，我们就可以生成 Qsys 系统的 RTL 代码了。

第十二步：保存设计

点击 file/save，将我们的系统保存，我这里命名为“cpu”。

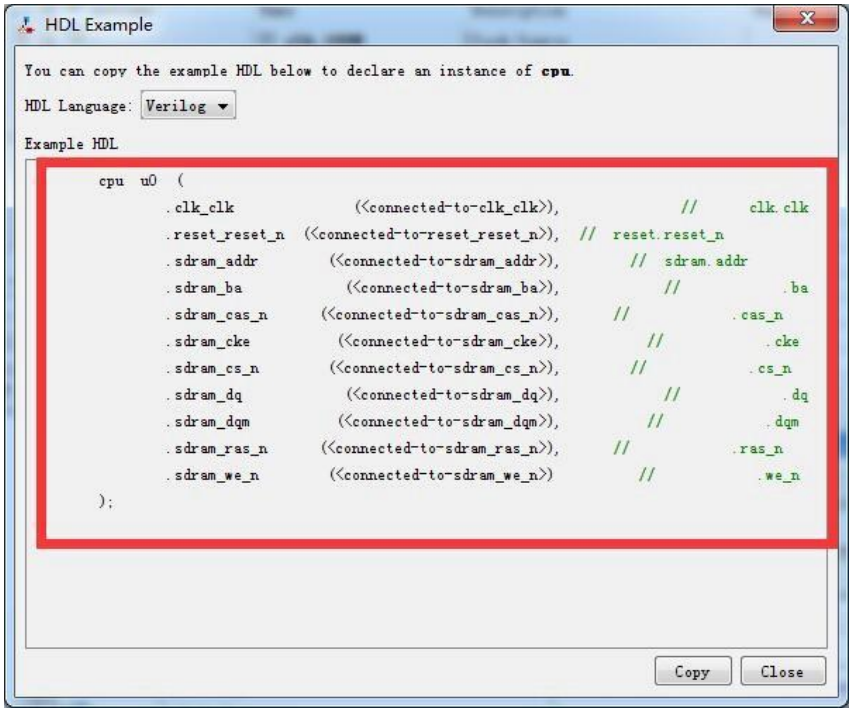
第十三步：生成 Qsys 系统的 HDL 代码

在 Qsys 系统中，依次点击 Generate/Generate HDL，在弹出的窗口中，点击 Generate

来完成 HDL 代码的生成。此过程根据电脑配置的不同，大约需要 3 到 5 分钟。

第十四步：在 Quartus II 系统中添加并例化 QSYS 系统

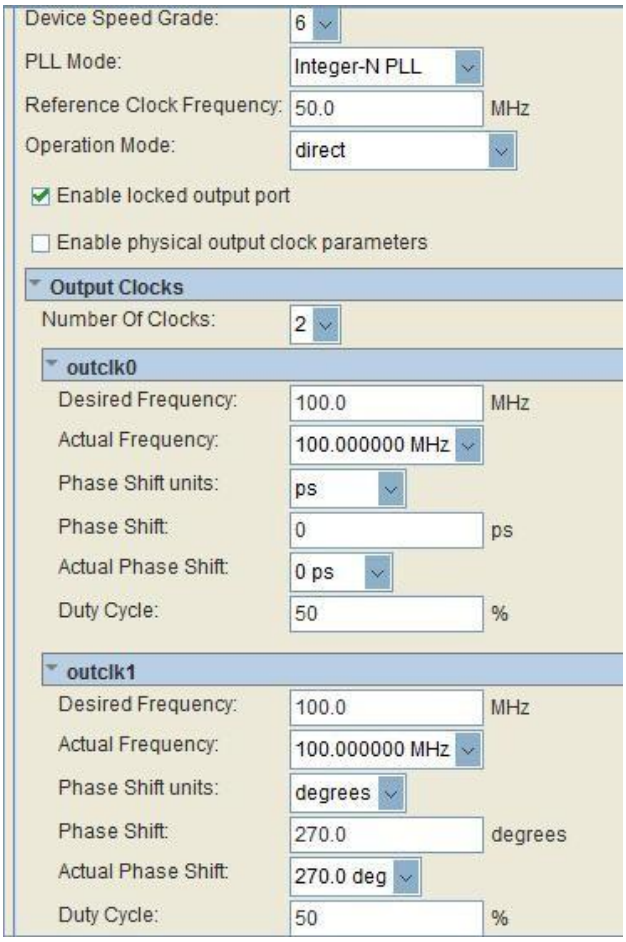
在 Qsys 系统中，依次点击 Generate/HDL Example，在打开的界面中，将 HDL Example 中的代码选中并复制，如下图中红线圈起来的内容：



回到 Quartus II 工程中，新建一个 Verilog HDL 文件，将刚刚复制的内容粘贴进去，将文件保存，命名为 mysystem。

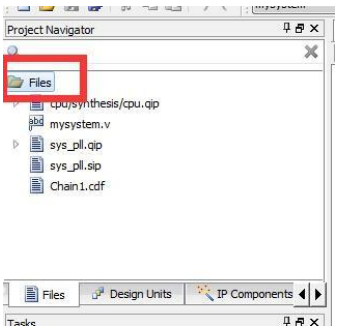
第十五步：添加锁相环

在 Quartus II 软件中，依次点击 tools/IP Catalog，在右侧弹出的窗口中，在搜索栏输入 pll，然后双击打开搜索结果中的 Altera PLL，在打开的窗口中，选择 Verilog 格式，并保存名为 sys_pll。然后点击 OK。等待大于 10 秒钟左右，会弹出 PLL 的配置窗口，在配置窗口中，按下图所示的内容进行配置：



第十六步：添加 Qsys 系统到 Quartus II 工程中

第十四步中，只是完成了 Qsys 系统的例化，实际并没有将 Qsys 系统添加到 Quartus II 工程中来。这里，我们双击 Quartus II 软件中 Project Navigator 窗口中 Files 选项卡中的 Files



在弹出的窗口中，找到 E:\mysystem\cpu\synthesis 路径下的 cpu.qip 文件，加入到工程中。

第十七步：完善 mysystem.v 模块

将 mysystem.v 中的内容完善如下：

```
module mysystem(
    refclk,
    rst n,
    sdram addr,
    sdram ba,
    sdram_cas_n,
    sdram cke,
    sdram cs n,
    sdram dq,
    sdram dqm,
    sdram ras n,
    sdram we n,
    sdram clk
);
    input refclk;
    input rst_n;
    output [12:0]sdram addr;
    output [1:0]sdram ba;
    output sdram cas n;
    output sdram cke;
    output sdram cs n;
    inout [15:0]sdram_dq;
    output [1:0]sdram dqm;
    output sdram ras n;
    output sdram we n;
    output sdram clk;

    wire cpu clk;
    wire sdram clk;
    wire cpu reset n;
    wire pll locked;

    assign cpu reset n = pll locked;

    cpu u0 (
        .clk clk      (cpu clk),      //  clk.clk
        .reset reset n (cpu reset n), // reset.reset n
        .sdram addr    (sdram addr),    // sdram.addr
        .sdram ba      (sdram ba),      //      .ba
        .sdram cas n   (sdram cas n),   //      .cas n
        .sdram cke     (sdram cke),     //      .cke
        .sdram cs n    (sdram cs n),    //      .cs n
        .sdram dq      (sdram dq),      //      .dq
        .sdram dqm     (sdram dqm),     //      .dqm
        .sdram ras n   (sdram ras n),   //      .ras n
        .sdram we n    (sdram we n)     //      .we n
    );

    sys pll 0002 sys pll inst (
        .refclk   (refclk),   //  refclk.clk
        .rst      (~rst n),     //  reset.reset
        .outclk 0 (cpu clk), // outclk0.clk
        .outclk 1 (sdram clk), // outclk1.clk
        .locked   (pll locked)    //  locked.export
    );

endmodule
```

第十八步：分析和综合；

按下键盘组合键 Ctrl + K 以对工程进行分析和综合。

第十九步：分配引脚

引脚分配如下表所示：

信号名	I/O	管脚映射
refclk	Input	PIN_E1
rst_n	Input	PIN_E16
sdram_we_n	output	PIN_T9
sdram_ras_n	output	PIN_N9
sdram_dqm[1]	output	PIN_R10
sdram_dqm[0]	output	PIN_T8
sdram_dq[15]	Inout	PIN_P9
sdram_dq[14]	Inout	PIN_N8
sdram_dq[13]	Inout	PIN_M8
sdram_dq[12]	Inout	PIN_L8
sdram_dq[11]	Inout	PIN_K8
sdram_dq[10]	Inout	PIN_L9
sdram_dq[9]	Inout	PIN_K9
sdram_dq[8]	Inout	PIN_R9
sdram_dq[7]	Inout	PIN_R8
sdram_dq[6]	Inout	PIN_R6
sdram_dq[5]	Inout	PIN_T5
sdram_dq[4]	Inout	PIN_R5
sdram_dq[3]	Inout	PIN_T4
sdram_dq[2]	Inout	PIN_R4
sdram_dq[1]	Inout	PIN_T3
sdram_dq[0]	Inout	PIN_R3
sdram_cs_n	output	PIN_R12
sdram_clk	output	PIN_T10
sdram_cke	output	PIN_T11
sdram_cas_n	output	PIN_R11
sdram_ba[1]	output	PIN_M9
sdram_ba[0]	output	PIN_T12
sdram_addr[12]	output	PIN_N11
sdram_addr[11]	output	PIN_R13
sdram_addr[10]	output	PIN_M10
sdram_addr[9]	output	PIN_T14
sdram_addr[8]	output	PIN_R14
sdram_addr[7]	output	PIN_T15
sdram_addr[6]	output	PIN_L11
sdram_addr[5]	output	PIN_M11
sdram_addr[4]	output	PIN_N12
sdram_addr[3]	output	PIN_T13

sdram_addr[2]	output	PIN_R5
sdram_addr[1]	output	PIN_T5
sdram_addr[0]	output	PIN_R6

第二十步：全编译

此过程大约需要 6 到 10 分钟左右。

第二十一步：配置固件到 FPGA 芯片中

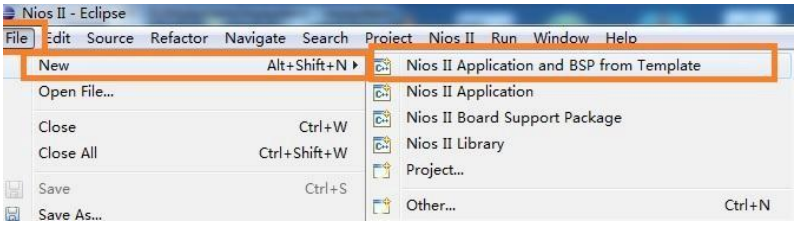
将生成的 SOF 文件配置到芯航线开发板中。

第二十二步：打开 NIOS II EDS 软件

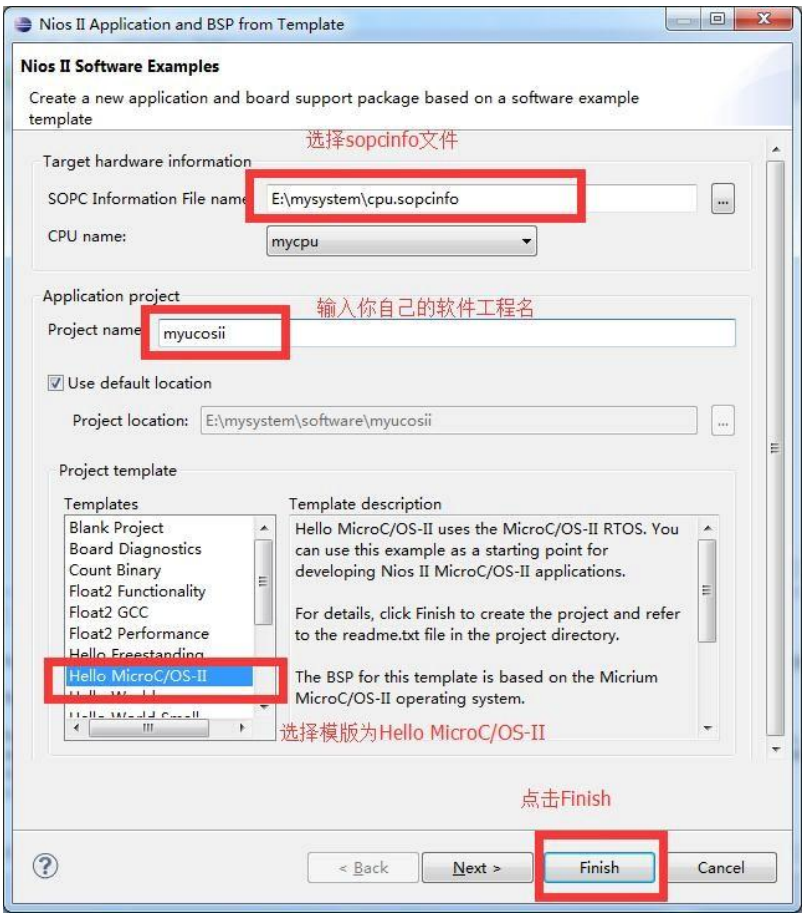
在 Quartus II 软件中，依次点击 tools/NIOS II software Buil tools for Eclipse，在打开的 Workspace Launcher 中，设定 Workspace 为 E:\mysystem，然后点击 OK。

第二十三步：建立 UC/OS II 模版工程

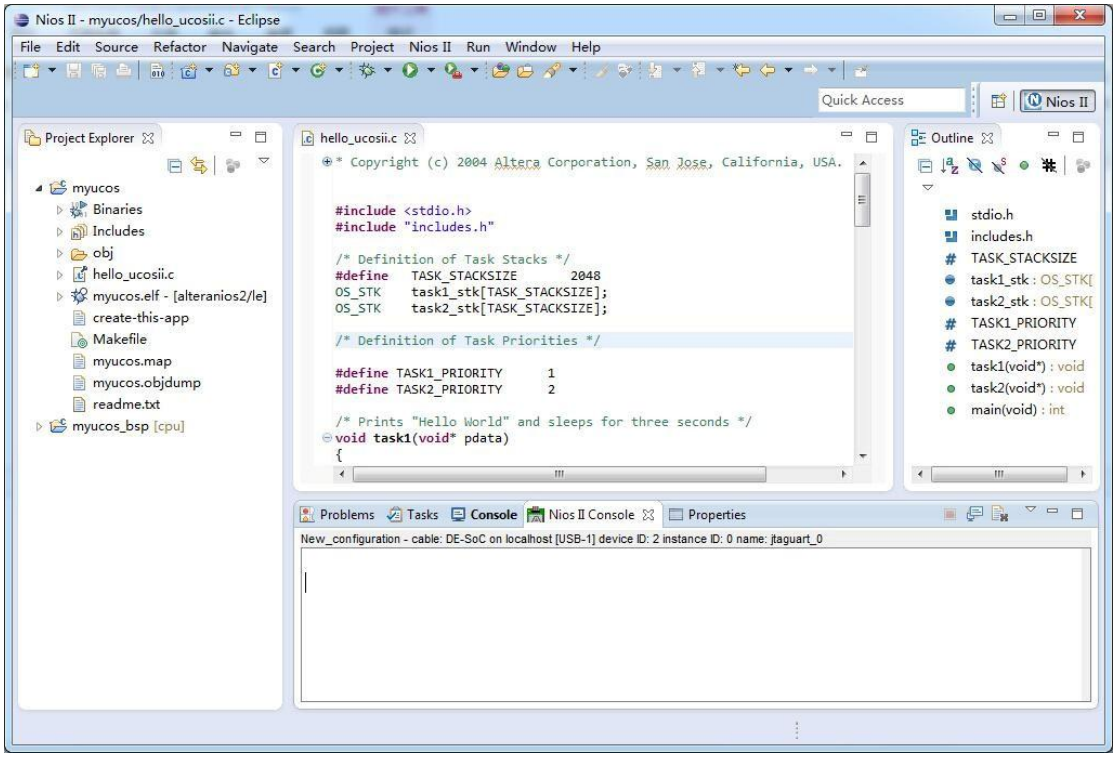
在打开的开发环境中，点击 File/New/Nios II Application and BSP from Template。



在打开的窗口中，按下图所示的内容进行选择：



然后系统便能为我们自动生成完整的系统，生成后的界面如下所示：

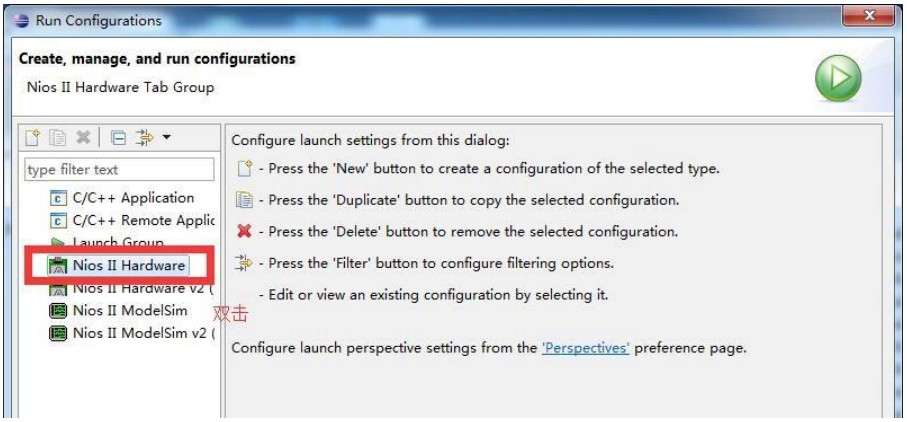


第二十四步：编译软件工程

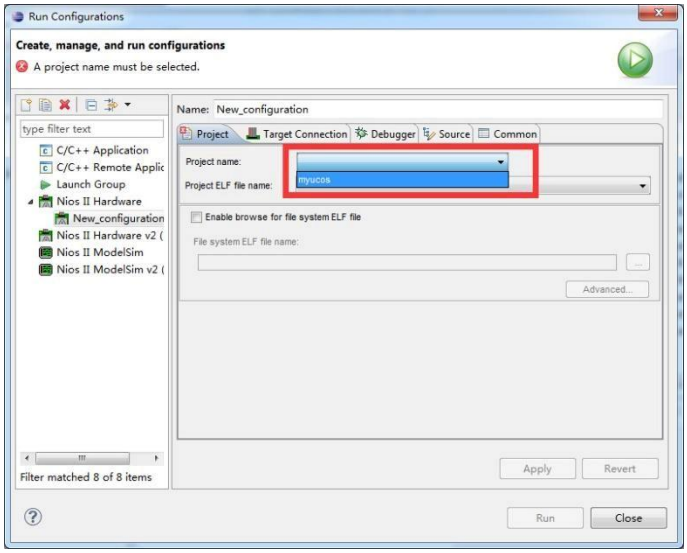
此时，我们按下键盘上的组合键“ctrl + B”，就能够对软件工程全编译。

第二十五步：下载并运行

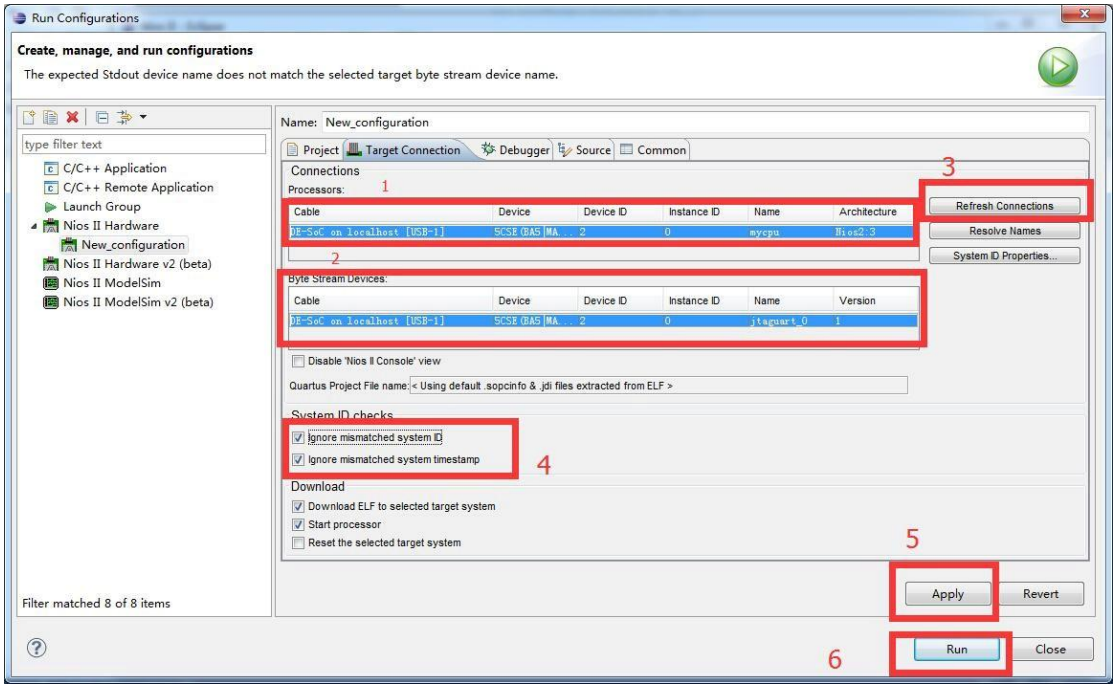
编译完成后，点击 Run/Run Configurations，在打开的窗口中，



双击 Nios II Hardware，在弹出的界面中，选择 Project Name 为 myucosii（注：图中我的工程名为 myucos，只是工程名字不同而已，无其他影响）。

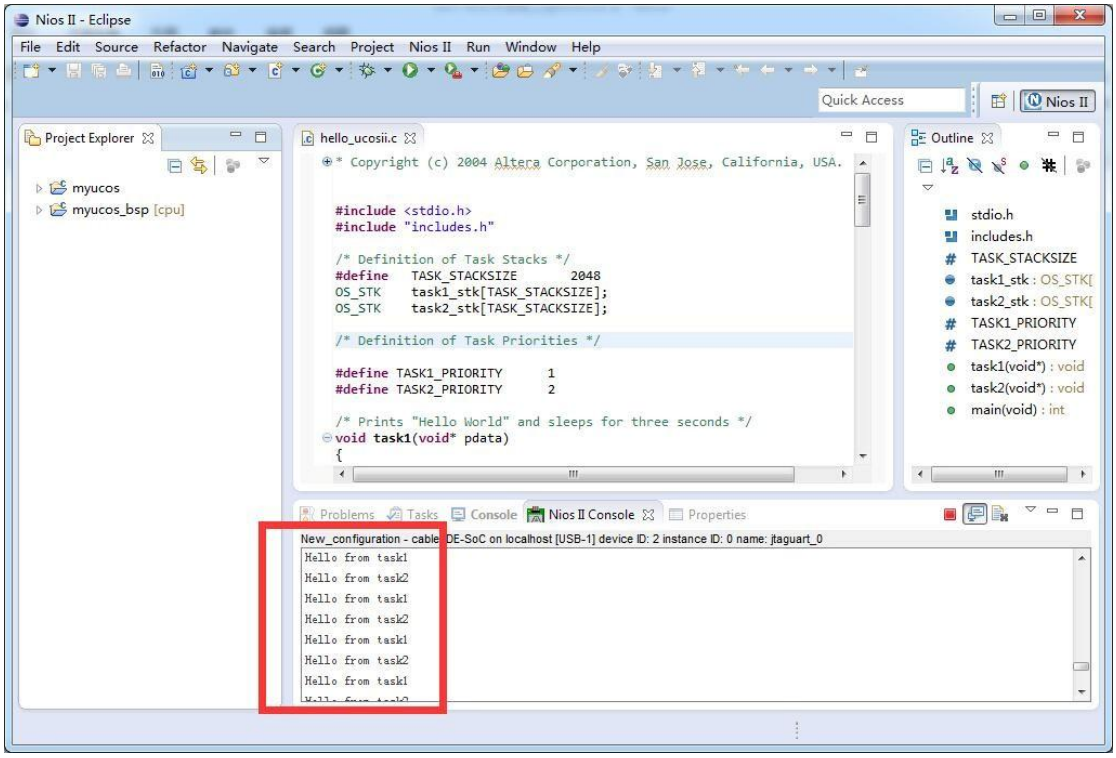


选择完成后，切换到 Target_Connection 选项卡中，如下图：



若 1 处和 2 处无内容，则点击 3 处的 Refresh Connections，直到 1 和 2 中出现我们的 Cable 和 Device 为止。选中 4 处的两个选项。然后点击 5 处的 apply，最后点击 6 处的 Run，则系统将启动软件烧写过程，并在烧写完成后启动 FPGA 中的 Nios II 处理器。

此时，我们在 Eclipse 的 Nios II Console 窗口中便能看到 Nios II 处理器通过串口发送到 PC 上的内容，如下图所示：



自此，整个在 AC620 FPGA 上搭建 NIOS II 处理器并运行 UC/OS-II 系统的实验告一段落。

总结

本文主要通过一个最简单的例子介绍了常见的 SOPC 简单开发流程，后续详细的 SOPC 开发教程敬请关注小梅哥后续视频教程。

第 4 章 Altera SOPC FrameBuffer 系统设计教程

小梅哥编写，未经授权，严禁转载或用于任何商业用途

在嵌入式系统中，LCD 屏作为最友好的人机交互方式，被大量的应用到了各个系统中。在基于 ARM 处理器的系统中，应用更是非常广泛。FPGA 作为广义嵌入式系统的一员，自然也有很多时候需要来驱动显示屏显示一些内容，例如经常有需求要用 FPGA 来做液晶测试架，做显示器驱动测试卡。很多学习了 FPGA 的朋友都知道，FPGA 驱动 VGA 显示器是比较轻松的，几乎每个板卡商提供的资料中都提供了有诸如显示彩条，显示图案，甚至显示简单的文字。但是，当希望显示更加复杂的内容的时候，往往就很难办到了。因为 FPGA 设计的是电路，是硬件，特点是头脑简单，四肢发达。而显示复杂的内容，恰恰需要一个头脑发达的控制器来执行这项任务。

在 FPGA 这些年的发展中，先后经历了简单逻辑扩展、复杂逻辑设计、SOPC 系统和 SOC 系统，逻辑设计部分就不多说了，对于 SOPC 系统设计，Xilinx 家有 MicroBlaze 软核、Altera 家有大名鼎鼎的 NIOS II 软核处理器。而到了 SOC 时代，Xilinx 家和 Altera（现已被因特尔收购，成为 Intel 的可编程事业部）都推出了内嵌双核 ARM Cortex-A9 的 SOC 芯片，Xilinx 家著名的 zynq7000 系列，以及 Altera 花了大力气在高校推广的 Cyclone V SOC。

虽然随着技术的发展以及时间的推移，SOC 的应用会越来越广泛，但是作为我们一般学习和使用来说，SOPC 技术也还是有其使用的价值的，毕竟内嵌硬核的 SOC 芯片，动辄几百块，而使用软核的方案，只要 40 元不到的 BOM 成本就能完成一个系统设计，优势还是很明显的。本例，我们就带领大家使用 Altera Cyclone IV 系列最低端的 FPGA 芯片 EP4CE6/EP4CE10 设计的能够驱动 640*480 分辨率显示屏动态显示复杂内容的系统。例如显示文字，显示图片

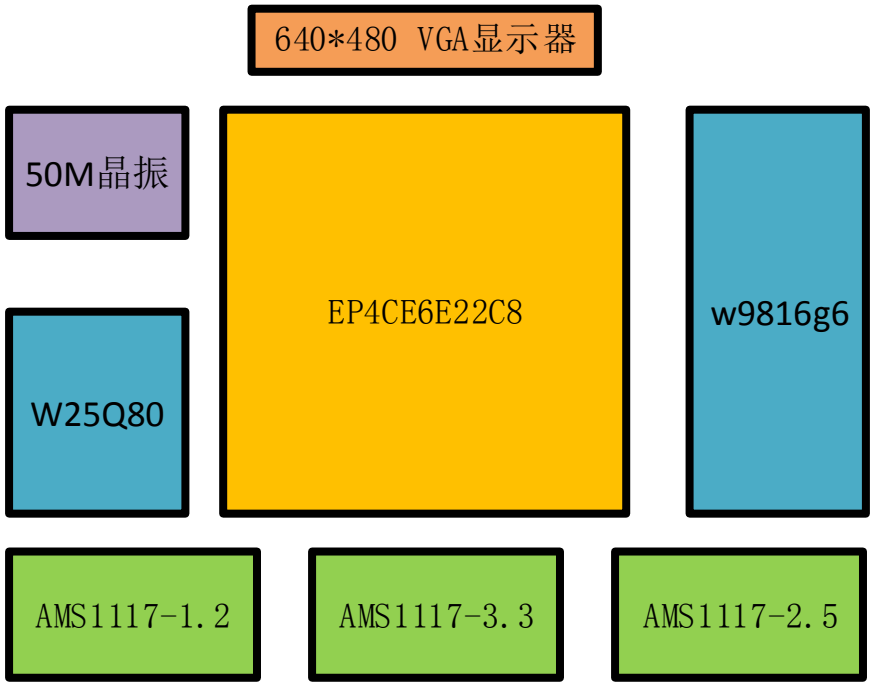
在之前我们讲解过 RGB 显示屏的驱动，RGB 显示屏驱动时序和 VGA 驱动时序几乎完全一样，只是不同的分辨率之间，其时序参数不同。说到这个驱动 480*272 分辨率显示屏动态显示复杂内容，相信很多朋友并不陌生，是的，在我们之前的 SOPC 公开课时候，就已经给大家讲解过实现这种系统的一种方式，即使用了一片独立的 SRAM 来作为显示屏的显存，显示驱动模块直接读取 SRAM 中的数据并实时刷新显示，而 NIOS II CPU 则只在需要刷新显示内容的时候，才执行对 SRAM 的写入操作。此种方式，需要一片独立的 SRAM 作为显存，以及一片 SDRAM 作为 NIOS II 的运行内存。除了 SRAM 的使用会增加系统的 BOM 成本外，SRAM 也会占用掉 FPGA 芯片较多的管脚，导致必须得使用 256 脚 BGA 方案的芯片才能够实现。因此，此种方案仅适合在对成本不太敏感高的场合使用。本节要介绍的方案，是对该系统改进优化得到的。

本节所描述的系统，仅需用到以下硬件资源

1. EP4CE6/EP4CE10 型 FPGA 芯片一片（30 元）
2. 16Mbit 或以上 SDRAM 芯片一片（3 元，如华邦 w9816g6）
3. 50M 有源晶振一片（3 元）

4. 4Mbit SPI FLASH 芯片（1.5 元，如 W25Q80）
5. 3.3V、2.5V、1.2V LDO 稳压电源（AMS117 系列，合计不到 1 元）

本节希望达到的目的：
在显示屏上显示一帧图像，并在指定位置显示字符内容

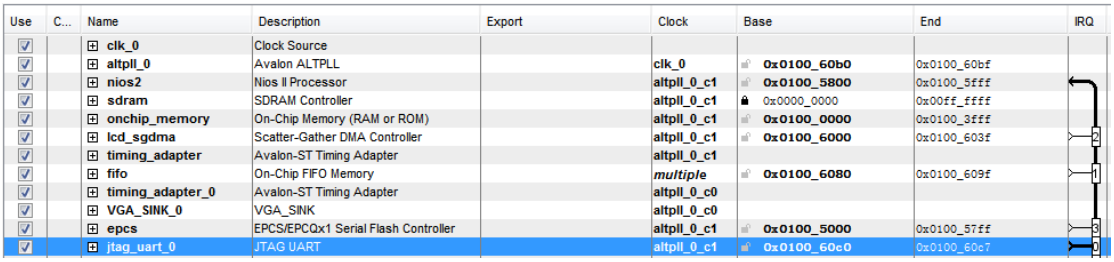


上述框架是我们经过分析，要实现本系统方案所需的最小硬件系统，本节我们将基于芯航线的 AC620 学习套件来完成 FPGA 的系统搭建和 NIOS II 软件设计。芯航线的 AC620 FPGA 开发板整体硬件配置高于上述分析，有助于我们进行原型验证。

FPGA 系统搭建
本系统主要在 Qsys 系统中完成，包含了以下 IP 核：

- clk_0：输入时钟管理单元，系统默认添加
- altpll_0：PLL 锁相环，对输入时钟进行分频和倍频，以得到两路频率为 100MHz，相位相差 180 度的时钟信号，分别提供给系统中逻辑电路工作（包含 NIOS II 软核处理器）和 SDRAM 芯片使用。以及 1 路 24M 的时钟信号，供 640*480 分辨率的 VGA 驱动电路使用。
- NIOS II CPU：实现系统的控制以及显示内容的处理。
- SDRAM：NIOS II CPU 运行内存和 TFT 显示图像帧缓存。
- onchip_memory：片上存储器，指定 SGDMA 要执行的数据传输，主要用作 SGDMA 的描述符存储器。
- lcd_sgdma：SGDMA IP，主要实现大量数据的高效搬运，支持流模式，效率比 Avalon MM 接口的 DMA 核高。
- timing_adapter：时序匹配 IP 核，主要用于流数据在两个不同模块间进行传递时，对部分信号加上一定延迟，使得数据和标志信号能够完全同步（说的简单点儿，就是用寄存器打拍的方式对一些信号进行延迟，使数据和控制信号对齐）。

- fifo：双时钟 fifo，主要完成数据流的传输速度转换。SGDMA 输出的数据流，是按照和存储器相同的时钟速度（本例中为 100M）进行的，而在数据使用端，即 VGA 显示部分，是按照 9M 的时钟取用数据的，因此使用双时钟 fifo，解决跨时钟域的问题。
- VGA_SINK：该 IP 为第三方开源 IP，由友晶科技提供，主要完成 Avalon ST 流数据转换成 VGA 行场时序并驱动 VGA 屏进行显示。
- EPCS：EPCS FLASH 存储器控制 IP，使用该 IP，使得 EPCS 芯片能够存储 FPGA 固件和 NIOS II 的程序。
- jtag_uart0：调试串口，主要用来打印一些调试信息，在设计的前期调试中很有用



Use	C...	Name	Description	Export	Clock	Base	End	IRQ
☑		clk_0	Clock Source					
☑		altpll_0	Avalon ALTPLL		clk_0	0x0100_60b0	0x0100_60bf	
☑		nios2	Nios II Processor		altpll_0_c1	0x0100_5800	0x0100_5fff	
☑		sdram	SDRAM Controller		altpll_0_c1	0x0000_0000	0x00ff_ffff	
☑		onchip_memory	On-Chip Memory (RAM or ROM)		altpll_0_c1	0x0100_0000	0x0100_3fff	
☑		lcd_sgdma	Scatter-Gather DMA Controller		altpll_0_c1	0x0100_6000	0x0100_603f	2
☑		timing_adapter	Avalon-ST Timing Adapter		altpll_0_c1			
☑		fifo	On-Chip FIFO Memory		*multiple*	0x0100_6080	0x0100_609f	1
☑		timing_adapter_0	Avalon-ST Timing Adapter		altpll_0_c0			
☑		VGA_SINK_0	VGA_SINK		altpll_0_c0			
☑		epcs	EPCS/EPCQx1 Serial Flash Controller		altpll_0_c1	0x0100_5000	0x0100_57ff	3
☑		jtag_uart_0	JTAG UART		altpll_0_c1	0x0100_60c0	0x0100_60c7	0

接下来，将详细讲解本系统的搭建过程。

1、 altpll_0

为了保证整个系统能够具有较高的性能，因此让其运行在 100MHz，所以需要使用 PLL 对外部晶振产生的 50MHz 输入的时钟信号进行倍频，得到两路频率为 100M，相位相差 90 度的时钟信号。另外，还需要提供一路 9MHz 时钟信号供 480*272 分辨率的 TFT 屏驱动使用。关于 PLL 的详细添加和参数修改方式这里不再多说，仅提供设置结果：

inclk0：50M

不需要 areset 信号和 locked 信号

c0：9MHz

c1：100MHz，相位为 0

c2：100MHz，相位为-180 度

2、 nios ii cpu

所有设置使用默认值，等后续添加完 sdram 和 EPCS 和，将复位向量（Reset Vector）设置为 epcs，将异常向量（Exception Vector）设置为 sdram。

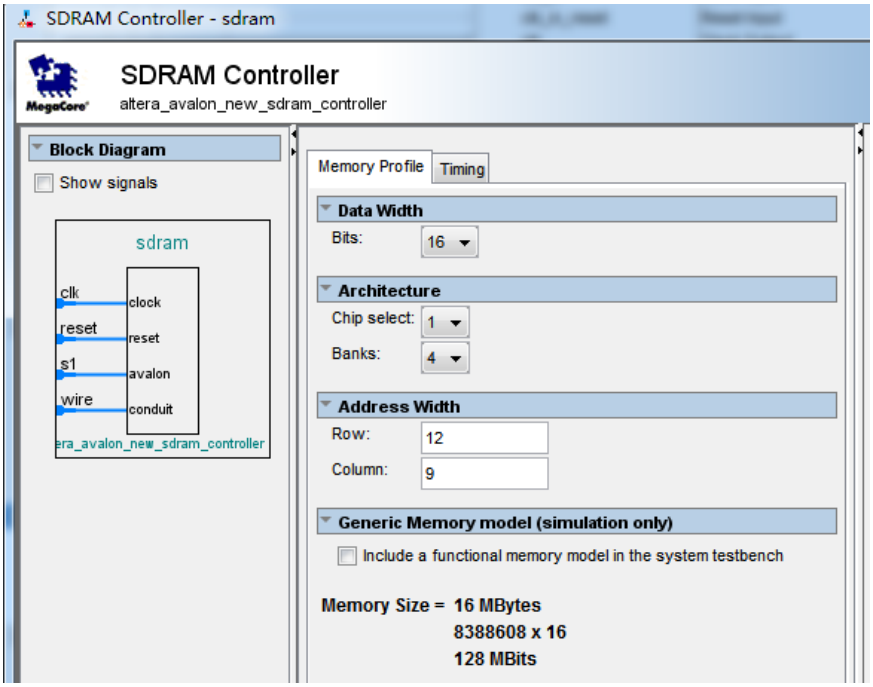
3、 sdram

数据位宽（Data Width）为 16bit。

结构（Architecture）为一个片选，4 个 bank。

地址宽度（Address Width）Row 地址为 12、Column 地址为 9。

其他默认即可。



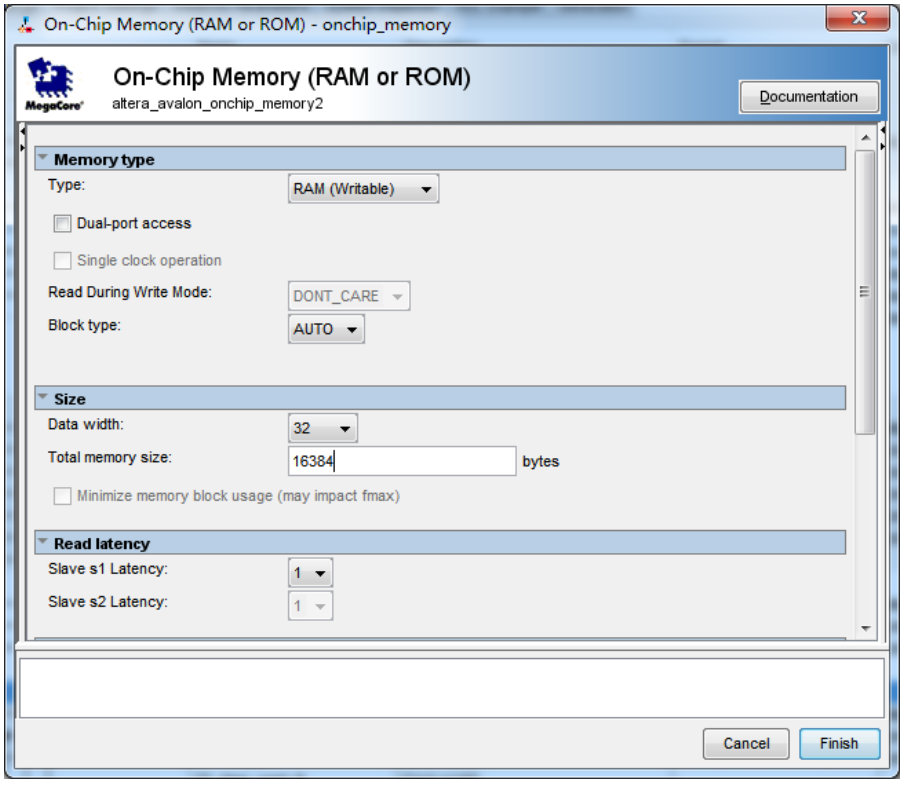
4、 onchip_memory

片上缓存 onchip_memory 作为 SGDMA 的描述符存储器，存储 SGDMA 的数据传输描述内容。

类型为 RAM；

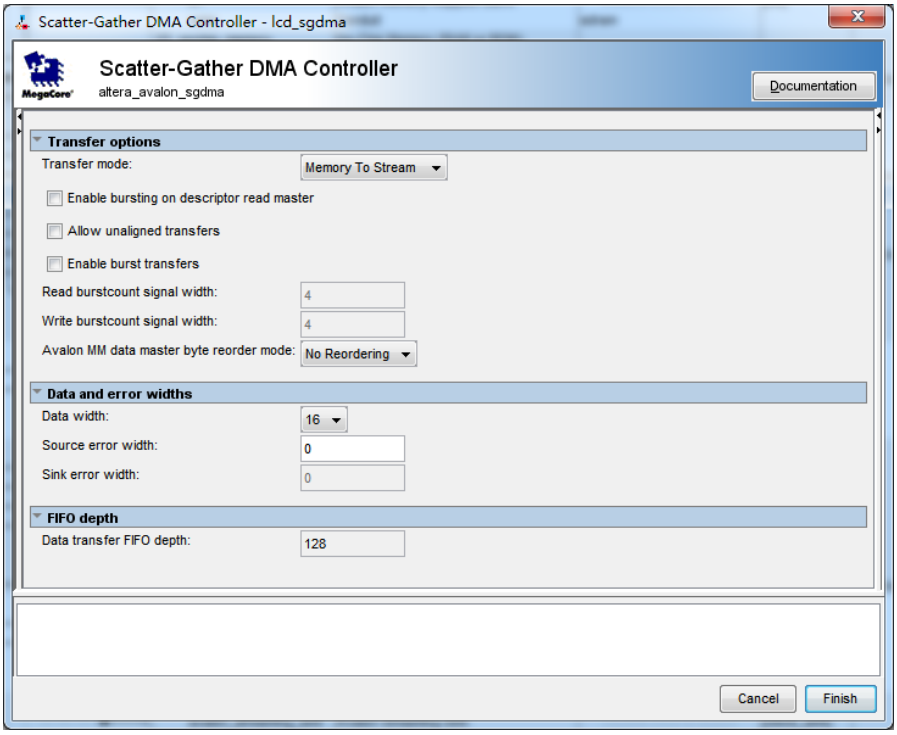
数据位宽选择 32 位；

存储器总量选择 16384Bytes（用户可以根据自己芯片的 RAM 容量适当增加或减小该值）；



5、 lcd_sgdma

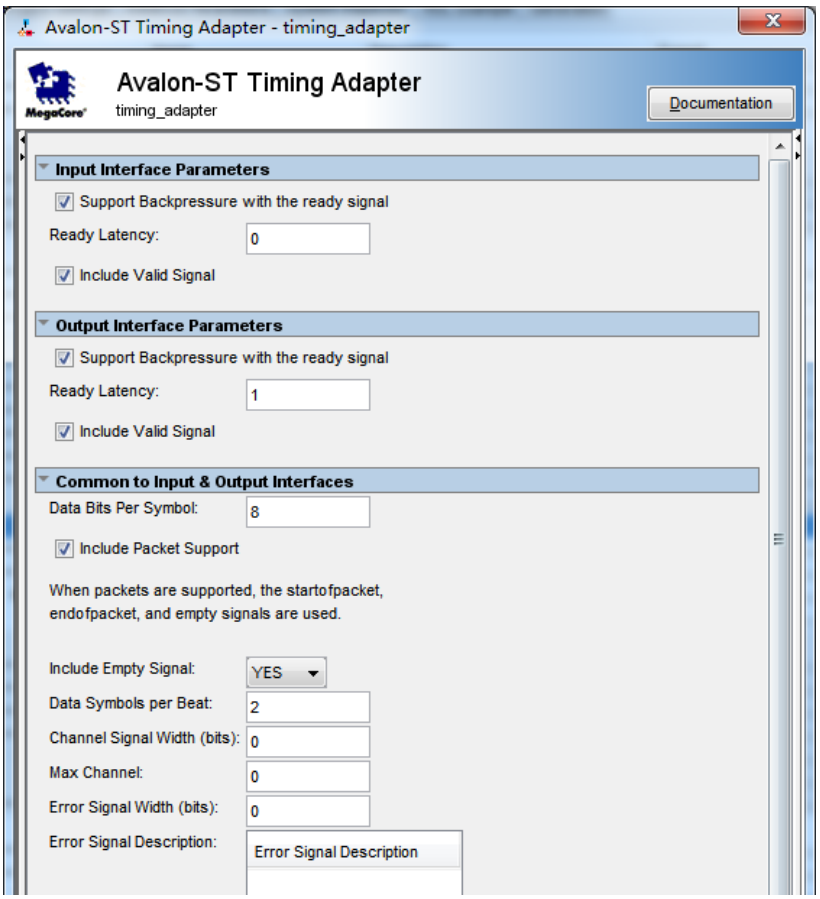
选择传输模式为存储器到数据流模式（Memory To Streram）；
数据宽度（Data width）为 16bit，该位宽与存储器（SDRAM）的 Avalon MM 总线宽度一致。
其他按照默认即可，如下图所示。



6、 timing_adapter

时序匹配 IP 核，主要用于流数据在两个不同模块间进行传递时，对部分信号加上一定延迟，使得数据和标志信号能够完全同步（说的简单点儿，就是用寄存器打拍的方式对一些信号进行延迟，使数据和控制信号对齐）。

如下图所示设置，所有勾选项都勾选上，输入 Ready Latency 设置为 0，输出 Ready Latency 设置为 1，每个数据总共有两个 symbols，每个 symbol 含 8 位数据。（根据 RGB 数据流的实际意义，这里应该是每个数据有 3 个 symbols，但是本设计使用的是 16 位色 RGB565 模式，即 RGB 总共才 16 位，所以在搭建系统的时候，假设只有 2 个 symbols，只是在最终数据输出到 VGA 的时候，将 16bit 数据拆分成 RGB565，分别送给 VGA 的 R、G、B 色通道）



以上各个模块，都是工作在 100MHz 的时钟频率下。主要完成将需要显示的数据从 SDRAM 中读取出来。而读取出来的数据最终是需要送到 TFT 显示屏上去显示的，TFT 显示部分，对于本系统，数据传输速率为 9M，因此就需要实现数据从 100M 时钟域到 9M 时钟域的跨时钟域传输。而实现数据流跨时钟域传输，较好的方式是使用双时钟 fifo。因此，接下来就需要添加一个双时钟 fifo，先将 100M 时钟域的数据缓存起来，然后再等待 9M 时钟域的读取逻辑来读取。

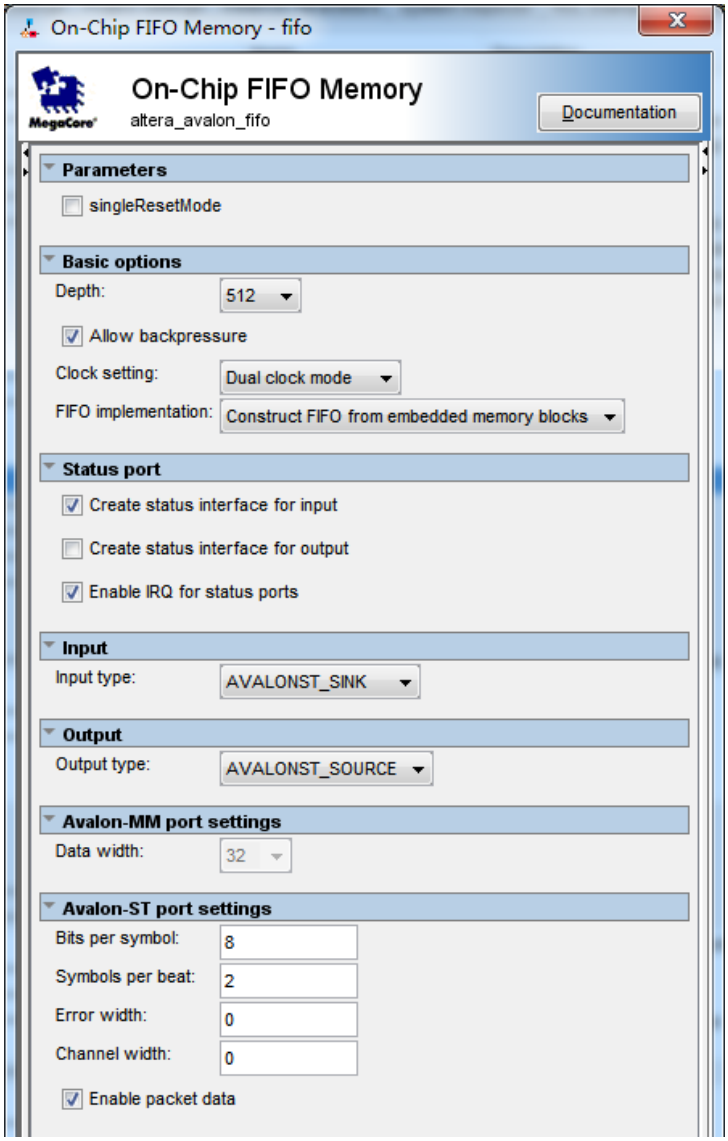
7、 fifo

设置 fifo 的深度为 512 字节，每个数据含 2 个 symbols，每个 symbols 由 8 位数据组成，即整个数据宽度为 16 位。时钟设置为双时钟模式（Dual Clock Mode），输入和输出都使用 Avalon ST 接口，使能数据包（Enable Packet Data）。这里顺带说一下，Cyclone IV E 器件中的嵌入式块 RAM 为 M9K 存储器，每个器件里都有若干个 M9K 存储器。所谓 M9K 存储器，就是一个块存储器中有 9Kb 个存储位，每个 M9K 存储器可以被配置为以下模式：

- 8192 × 1
- 4096 × 2
- 2048 × 4
- 1024 × 8
- 1024 × 9
- 512 × 16
- 512 × 18

- 256 × 32
- 256 × 36

由于本设计中，数据是 16 位的，因此，如果要想仅使用一个 M9K 存储器就实现本 Fifo，存储深度可在 1 到 512 之间任选。假如我们选择存储深度为 128，那么就只占用 1/4 个 M9K 的存储容量，但是，剩下的 3/4 的存储容量也依旧无法单独使用，在该设计中会永远的被浪费掉，因此，不如将存储深度直接设置为 512，还能保证缓存足够大，避免意外丢失数据。

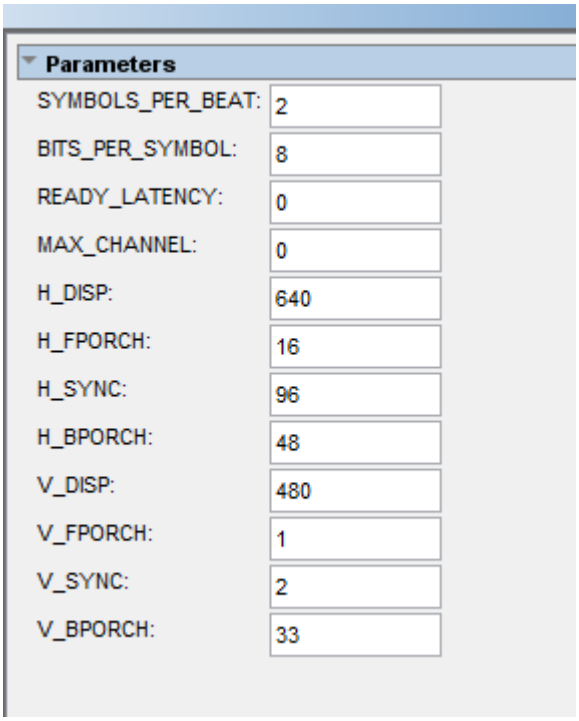


8、 VGA_SINK

VGA_SINK 模块是从友晶提供的设计中修改得到的，该 IP 原本仅支持 24 位色模式，而本系统，由于 SDRAM 为 16 位宽度，如果强行使用 24 位色模式，则必须得从 SDRAM 中读取两个数据才能拼接成一个像素，导致 SDRAM 的负载过大，从而使得系统带宽遇到瓶颈，无法正常工作，因此对该 IP 进行了修改，即降低为 16bit 的 RGB565 模式。（根据 RGB 数据流的实际意义，这里应该是每个数据有 3 个 symbols，但是本设计使用的是 16 位色 RGB565 模式，即 RGB 总共才 16 位，所以在搭建系统的时候，假设只有 2 个 symbols，只是在 VGA_SINK 模块内部，在最终数据输出到 VGA 的时候，进行了修改，将 16bit 数据拆分成 RGB565，分别送给 VGA 的 R、G、B 色通道）。这里参数分别如下设置：

H_DISP（行显示有效像素）：640
H_FPORCH：16 时钟周期
H_SYNC：96 时钟周期
H_BPROCH：48 时钟周期

V_DISP（行显示有效像素）：480
V_FPORCH：1 时钟周期
V_SYNC：2 时钟周期
V_BPROCH：33 时钟周期



这些参数都是从 VGA 标准中查到的。

9、EPCS

EPCS 的添加没有什么需要特别注意的，一切按照默认设置即可。

至此，我们已经完成了整个显示系统的组件添加工作，接下来就该进行组件间的总线连接了。

在总线连接之前，必须要先给大家讲解一下本系统的数据流向，因为本系统和我们之前在 SOPC 公开课时候讲过的系统架构还不太一样，本系统主要引入了 Avalon ST 总线。在之前公开课时候讲过的各种例子，都是基于 Avalon MM 总线的，所有外设 IP 都通过 Avalon MM 总线连接到 NIOS II CPU 上，因此总线连接是非常简单的。但是 Avalon MM 总线的数据搬运能力是比较弱的，无法支持高速大量的数据传输。而在基于 Avalon ST 总线的数据流中，NIOS II CPU 实质相当于被旁路了，只起到了状态管理的作用，所有的数据流转都是模块间通过 Avalon ST 总线进行交互的，不需要 NIOS II 去执行搬运工作，因此数据的搬运效率大大提高。

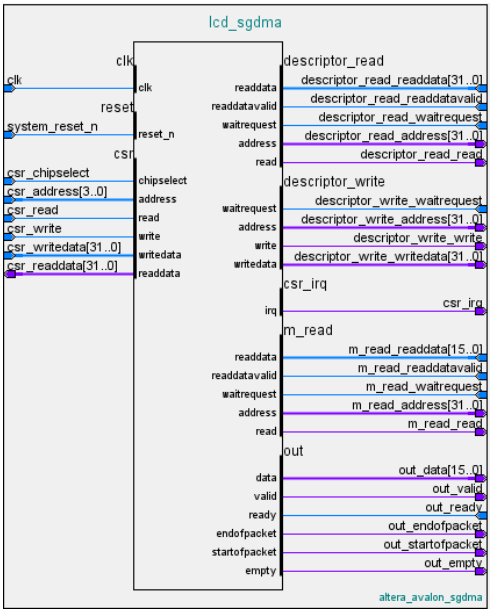
本节暂不对 Avalon ST 总线进行过多的讲解，总之，我们可以把 Avalon ST 总线想象成一

个水管，水从水管的一头可以很容易且快速的流到另一头，不需要第三方用水瓢一瓢一瓢的去舀过去。

Framebuffer 系统数据流

在整个 FrameBuffer 系统中，SGDMA 模块是最核心的部分，他实现了对 SDRAM 存储器的直接读取操作，并将读到的数据最终以数据流的形式输出。SGDMA 的使用相当的广泛，不仅是在本系统中，在 Altera 很多官方设计中也都使用了 SGDMA IP 核，比如 PCIE 示例、千兆以太网示例、VIP 示例等，关于 SGDMA 的详细介绍，可以参看 SGDMA 的用户手册，我们也希望能尽快推出 SGDMA 的专题讲解。这里仅对 SGDMA 的端口和功能进行下简述。

本系统中 SGDMA 被配置为存储映射模式到数据流模式，即数据的源头采用地址映射的方式组织，而数据的接收方则使用数据流的方式进行接收。在此种模式下，SGDMA 总共有 5 个端口，分别为 Avalon MM Slave 接口的控制总线，Avalon MM Master 接口源数据读取总线、Avalon MM Master 接口的描述符写入总线和 Avalon MM Master 接口的描述符读取总线，以及最终的 Avalon ST Source 数据输出总线。



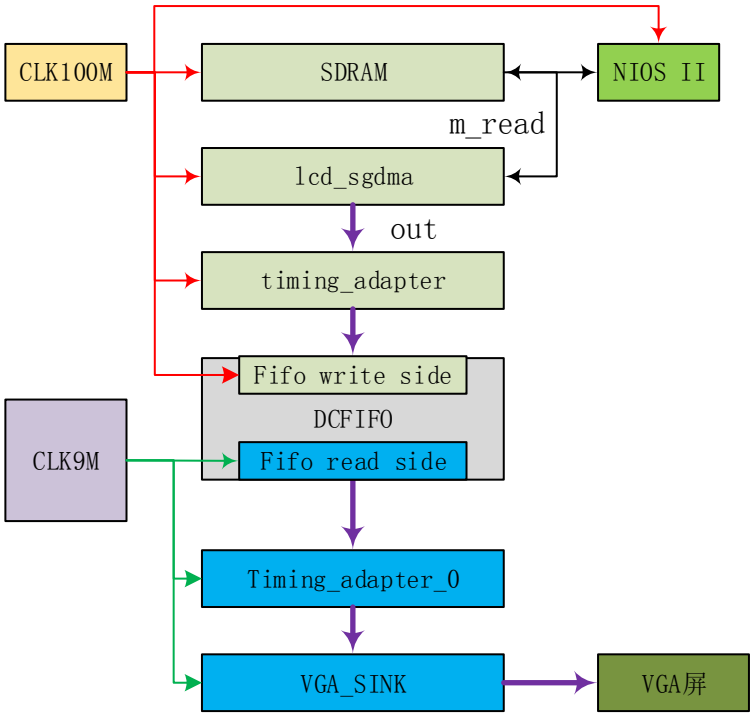
csr：Avalon MM Slave 接口的控制总线，与 NIOS II CPU 进行连接，NIOS II CPU 通过该接口读写 SGDMA 内部的控制和状态寄存器，以实现 DMA 的传输控制。这也是整个数据流传输唯一需要 NIOS II 参与的地方。NIOS II 实际就扮演了个老板的角色，安排 SGDMA 去做什么，SGDMA 接到任务后就下去干搬运的苦力活儿了，而 NIOS II 这个老板，则可以坐在办公室喝喝咖啡，看看报纸，等着 SGDMA 把苦力干完后回来报告（中断）或者隔一段时间打个电话问问工作状态和进度（查询）。

m_read：Avalon MM Master 接口的源数据读取总线，该接口实现对 Avalon MM 总线上挂载的存储需要传输的数据源存储器的读取，例如 SDRAM、DDR2 SDRAM。从这里，大家也可以看到，在 Qsys 搭建的系统中，不是只有 NIOS II 能够去读取 SDRAM、DDR2 等存储器，实际上，只要是 Avalon MM Master 接口的 IP，都可以去读取这些存储器，我们甚至可以自己编写一个 Avalon MM Master 接口的逻辑，来取代 NIOS II CPU 完成各种 IP 核的读写操作。

descriptor_write：描述符写端口，SGDMA 实现数据传输需要有一个描述符，SGDMA 的所有传输都是受描述符控制的，通过描述符存储 SGDMA 的实时传输状态。这里专门使用一个片上存储器来存储 SGDMA 的描述符，能够节省 FPGA 资源。否则，如果使用 FPGA 资源来实现描述符，将会带来非常大的资源消耗。

descriptor_read：描述符读端口，SGDMA 使用此端口读取描述符，以获得传输信息。
out：数据流输出端口，从 SDRAM 或 DDR2 中读取到的数据会经由此端口流出，以提供给数据的使用端使用。

对于我们分析数据流来说，可以暂时忽略 csr、descriptor_write 和 descriptor_read 端口，因为这些端口并不处于数据通路上，真正处于数据通路上的，是 m_read（数据流入）和 out（数据流出）端口。下图为本系统的数据流图。



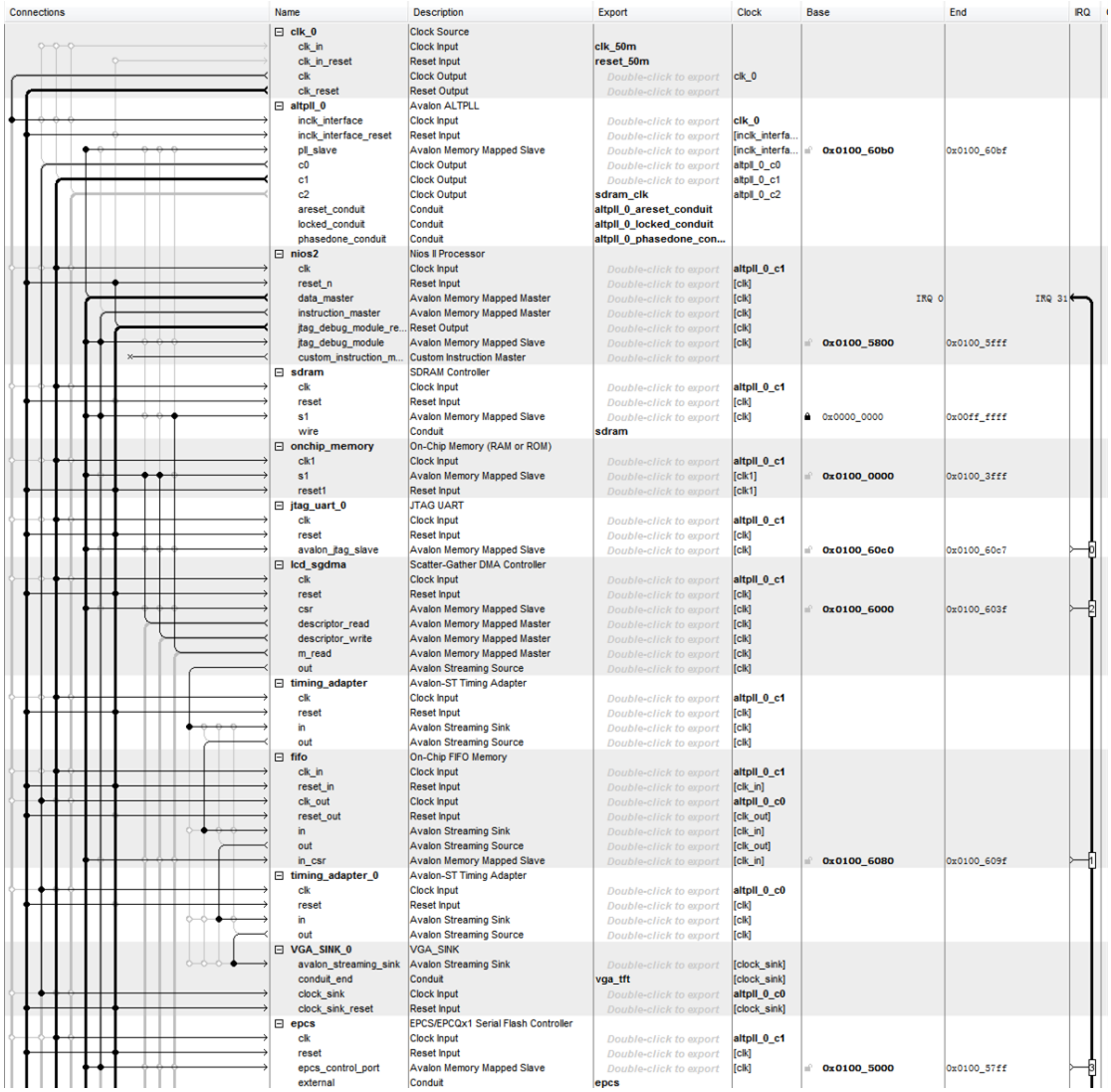
NIOS II CPU 使用 SDRAM 作为运行存储器，存储程序和数据，同时，SGDMA 从 SDRAM 中实时读取需要显示到 VGA 屏上的数据。NIOS II 和 SGDMA 之间对 SDRAM 的使用仲裁由 Avalon MM 总线的仲裁机制自动处理。

通过以上介绍，了解了系统的数据流向，就可以首先完成数据流总线的连接了。每个 st 流接口的模块都有 2 个流端口，一个是流输入端口（in: Avalon Streaming Sink），另一个是流输出端口（out: Avalon Streaming Source）。连接时，只需将上一级模块的流 out 连接到下一级模块的流 in 端口，即可实现数据流的自动衔接。在本例中每个模块端口连接如下表所示：

sdram	sdram.s1	nios2.data_master; nios2.instruction_master;	clk100m
lcd_sgdma	lcd_sgdma.csr	nios2.data_master;	
	lcd_sgdma.des_read	onchip_memory.s1	
	lcd_sgdma.des_write	onchip_memory.s1	
	lcd_sgdma.m_read	sdram.s1	
	lcd_sgdma.out	timing_adapter.in	
timing_adapter	timing_adapter.out	fifo.in	
fifo	fifo.in_csr	nios2.data_master;	
fifo	fifo.out	timing_adapter_0.in	clk9m
timing_adapter_0	timing_adapter_0.out	VGA_SINK_0.avalon_streaming_sink	

VGA_SINK_0	VGA_SINK_0.conduit_end	导出，连接到 VGA 驱动芯片引脚	

整体连接完成后，结果如下图所示：



当然，连接完成总线后，不要忘了连接复位网络、导出需要引出到系统顶层的信号以及中断网络。

- 修改 NIOS II 的复位向量为 EPCS、异常向量为 SDRAM。
- 自动分配基地址。
- 到 HDL Eaxmple 栏复制例化模版。
- 保存并开始 generate。

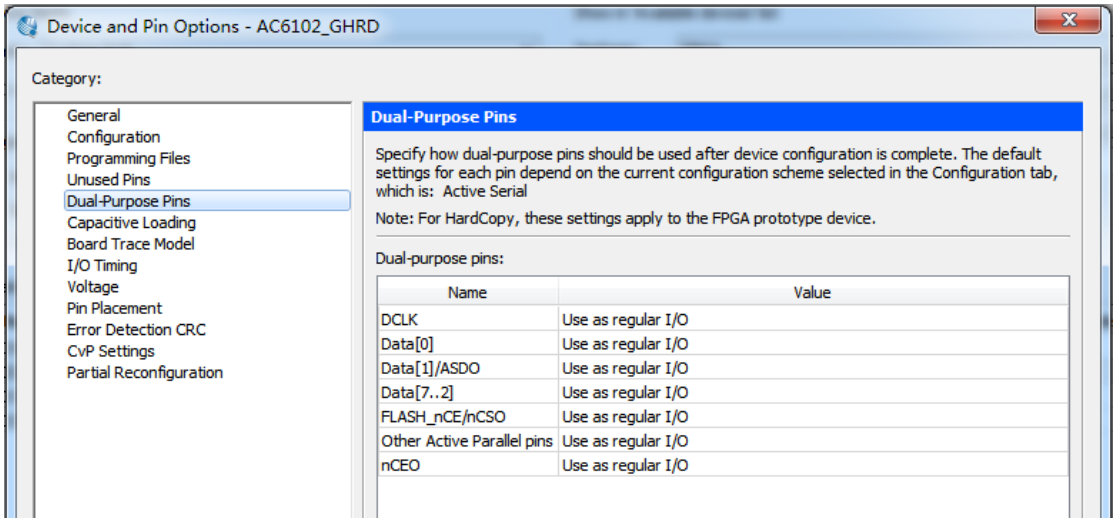
至此，整个系统就搭建完成了，下一步就是将系统加入到 Quartus II 工程中。首先是将 mysystem.qsys 文件添加到工程，然后完成工程顶层文件的编写。这些最基本的操作，这里就不在赘述，以下为工程顶层代码：

```
module Framebuffer VGA(
       input  wire        clk 50m,
       input  wire        reset n,
       output wire        vga clk,
       output wire        vga de,
```

```
        output wire [7:0]  vga r,
        output wire [7:0]  vga g,
        output wire [7:0]  vga b,
        output wire        vga hs,
        output wire        vga vs,
        output wire        vga bl,
        output wire        sdram clk,
        output wire [11:0] sdram addr,
        output wire [1:0]  sdram ba,
        output wire        sdram cas n,
        output wire        sdram cke,
        output wire        sdram cs n,
        inout  wire [15:0] sdram dq,
        output wire [1:0]  sdram dqm,
        output wire        sdram ras n,
        output wire        sdram we n,
        output wire        epcs dclk,
        output wire        epcs sce,
        output wire        epcs sdo,
        input  wire        epcs data0
);

    assign vga bl = 1;

    wire vga clk r;

    assign vga clk = ~vga clk r;    //对 VGA 时钟反相，以保证数据中心对齐

    mysystem u0 (
        .clk 50m clk                    (clk 50m),
        .reset 50m reset n               (reset n),
        .vga clk                        (vga clk r),
        .vga de                         (vga de),
        .vga r                          (vga r),
        .vga g                          (vga g),
        .vga b                          (vga b),
        .vga hs                         (vga hs),
        .vga vs                         (vga vs),
        .altpll 0 phasedone conduit export (),
        .altpll 0 locked conduit export    (),
        .altpll 0 areset conduit export    (),
        .epcs dclk                      (epcs dclk),
        .epcs sce                       (epcs sce),
        .epcs sdo                       (epcs sdo),
        .epcs data0                     (epcs data0),
        .sdram clk clk                   (sdram clk),
        .sdram addr                     (sdram addr),
        .sdram ba                       (sdram ba),
        .sdram cas n                    (sdram cas n),
        .sdram cke                      (sdram cke),
        .sdram cs n                     (sdram cs n),
        .sdram dq                       (sdram dq),
        .sdram dqm                      (sdram dqm),
        .sdram ras n                    (sdram ras n),
        .sdram we n                     (sdram we n)
    );

endmodule
```

然后分配引脚，本系统在 AC620 开发板上的引脚分配请参照 AC620 开发板引脚分配表。

分配引脚后，记得在 quartus ii 软件中设置所有双功能引脚为用户模式。

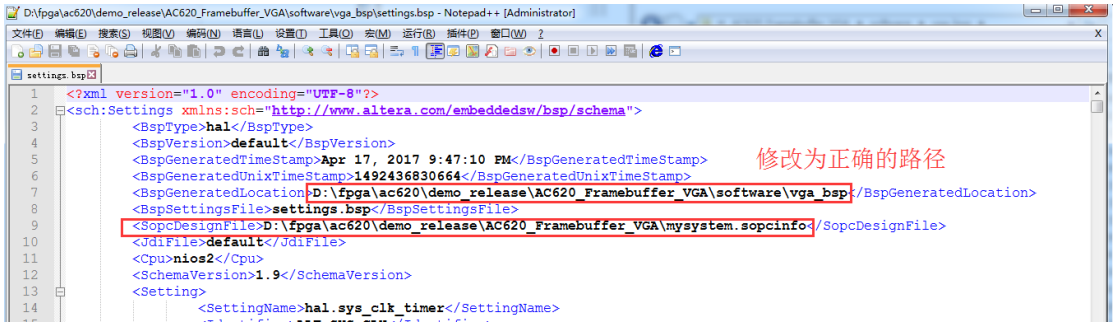


编译工程得到 sof 编程文件。

至此，整个 FrameBuffer 系统硬件设计就完成了，下一步，我们就可以编写相应的驱动程序和应用程序，在目标板上运行了。

创建 Eclipse 工程

打开 NIOS II EDS 软件，导入我们提供的 VGA 和 VGA_bsp 两个工程。修改 setting.bsp 文件中第 7 行和第 9 行两个位置为自己电脑上对应的位置。关闭文件，然后 build 工程。如果编译失败，提示无权限，请先关闭所有文件，然后选择整个文件夹右键获取管理员权限。



以下为 main 函数内容

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "sys/alt_alarm.h"
#include "altera_avalon_sgdma.h"
#include "altera_avalon_sgdma_descriptor.h"
#include "altera_avalon_sgdma_regs.h"
#include "alt_types.h"
#include "alt_video_display.h"
#include "unistd.h"
#include "pic1.h"
#include "pic2.h"
```

```
#include  "system.h"


#define  WIDTH 640
#define  HEIGHT 480
#define  NUM_FRAME 1


int main() {
    unsigned int d = 0;

    ////Initial LCD Display
    alt_video_display* display_global;
//  printf("Initializing LCD display controller\n  ");
    display_global = alt_video_display_init(LCD_SGDMA_NAME, // Name
of video controller
            WIDTH, // Width of display
            HEIGHT, // Height of display
            16, // Color depth (32 or 16)
            SDRAM_BASE + SDRAM_SPAN / 2, // Where we want our frame
buffers
            ONCHIP_MEMORY_BASE, // Where we want our descriptors
            NUM_FRAME);
//  if (display_global)
//      printf(" - LCD Initialization OK\n");
//  else
//      printf(" - LCD FAILED\n");
    alt_video_display_clear_screen(display_global, 0xff);
    show_pic(display_global, pic1);
    usleep(1000000);
    while(1){
    }
}
```

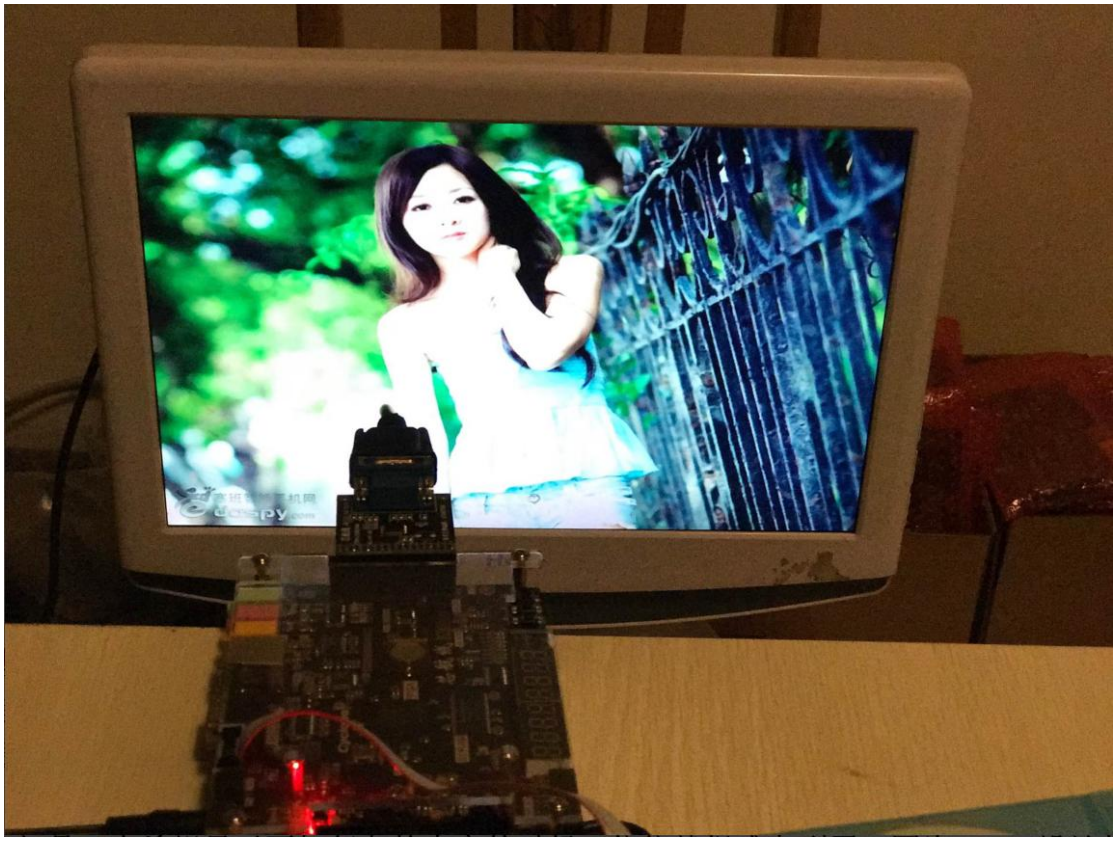
编译完成后下载 sof，并在 nios ii eds 中执行 run。连接 VGA 显示器，就能在屏幕上显示一副美女图，如下图所示：

如有其它问题，请参见小梅哥博客中关于 NIOS II 的相关内容。如“【2017 最新】解决 NIOS II 工程移动在磁盘上位置后 project 无法编译问题.pdf”和“*NIOS II 开发注意点总结.pdf*”

本文档对应设计工程名为：AC620_Framebuffer_VGA，请到 AC620 开发板网盘/光盘资料中获

取。





第 5 章 RT-Thread 操作系统在 NIOS II 上的移植

作者：小梅哥　个人博客：http://www.cnblogs.com/xiaomeige/

RT-Thread 介绍

RT-Thread 是一个集实时操作系统（RTOS）内核、中间件组件和开发者社区于一体的技术平台，由熊谱翔先生带领并集合开源社区力量开发而成，RT-Thread 也是一个组件完整丰富、高度可伸缩、简易开发、超低功耗、高安全性的物联网操作系统。RT-Thread 具备一个 IoT OS 平台所需的所有关键组件，例如 GUI、网络协议栈、安全传输、低功耗组件等等。经过 11 年的累积发展，RT-Thread 已经拥有一个国内最大的嵌入式开源社区，同时被广泛应用于能源、车载、医疗、消费电子等多个行业，累积装机量超过两千万台，成为国人自主开发、国内最成熟稳定和装机量最大的开源 RTOS。

RT-Thread 拥有良好的软件生态，支持市面上所有主流的编译工具如 GCC、Keil、IAR 等，工具链完善、友好，支持各类标准接口，如 POSIX、CMSIS、C++应用环境、Javascript 执行环境等，方便开发者移植各类应用程序。商用支持所有主流 MCU 架构，如 ARM Cortex-M/R/A, MIPS, X86, Xtensa, C-Sky, RISC-V，几乎支持市场上所有主流的 MCU 和 Wi-Fi 芯片。

AC620 FPGA 开发板介绍

AC620 FPGA 开发板是由小梅哥 FPGA 团队针对 FPGA 在基础应用领域的常见应用方面而精心设计的一款多功能 FPGA 开发板，FPGA 使用 Intel 公司的 Cyclone IV E 系列 EP4CE10f17c8n 芯片、板载 128M bit 的 SDRAM 存储器、16Mbit 的 FLASH 存储器，以及 12 位 AD/DA，CAN 通信接口、RS485 接口、RS232 接口、RTC，板载百兆以太网 PHY，也提供外扩的 DM9000 以太网 MAC 模块，非常适合作为 RT-Thread 操作系统的运行平台。



实验说明

本节以小梅哥 AC620 型 FPGA 开发板为例，说明了如何基于在 Qsys 中搭建支持 RT-Thread 操作系统的基本 NIOS II 处理器系统，并在此基础上实现创建 2 个 LED 闪烁的静态线程的示例。

RT-Thread 操作系统版本：3.0
软件版本：Quartus II 13.0
开发环境：Windows7 64 位旗舰版
硬件平台：小梅哥 AC620 FPGA 开发板
嵌入式处理器：NIOS II

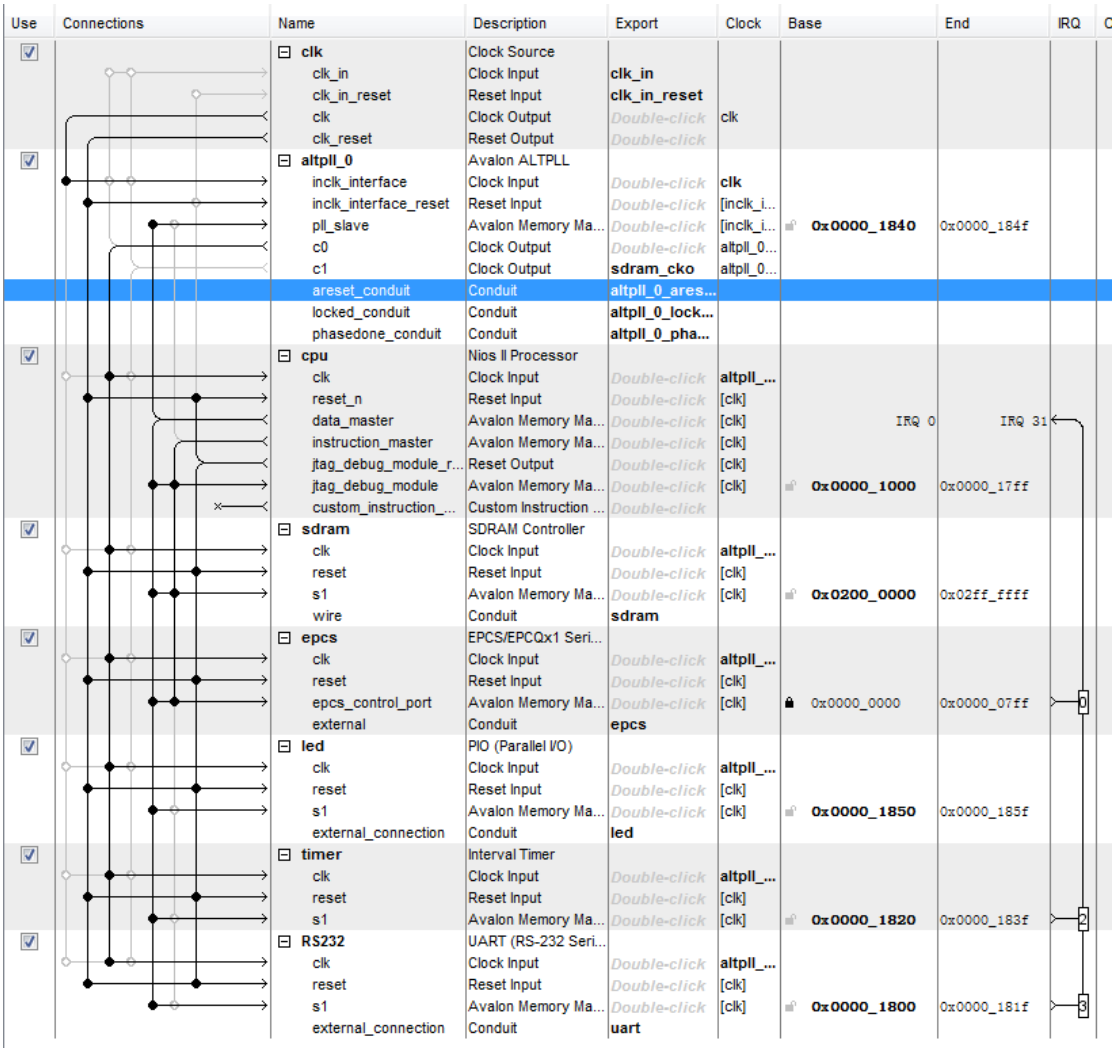
操作步骤

搭建支持 RT-Thread 操作系统的 NIOS II CPU 系统

在 Qsys 中搭建能够支持 RT-Thread 操作系统的 NIOS II 处理器系统，除了必要的 NIOS II 处理器、RAM 存储器（片上 RAM、SRAM、SDRAM、DDR2）、FLASH 存储器（EPCS）外，为了支持 RT-Thread 操作系统的运行，还需要添加 UART 串口外设、定时器外设、以及为了使系统能够工作在较高的频率，而添加 PLL 锁相环。同时，为了能够驱动 LED 灯，还需要根据实际硬件情况添加一定位宽的输出型 PIO 以驱动 LED 灯。本系统主要包含以下外设：

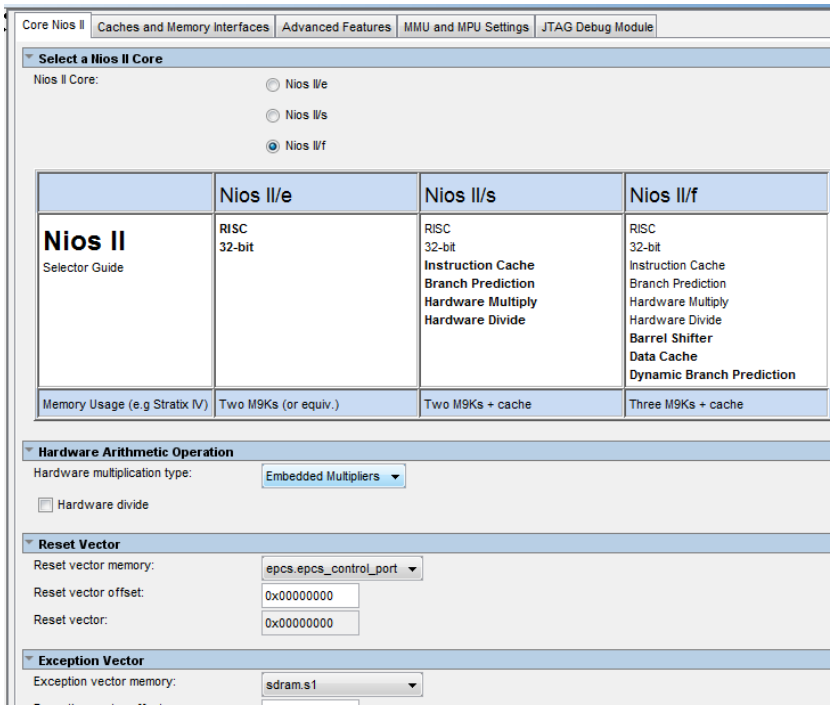
- PLL 时钟管理单元，输出 2 路相位相差 90° 的时钟，一路提供给系统内所有逻辑使用，一路输出到 SDRAM 芯片的 clk 管脚上。（相位为-90°）
- NIOS II 处理器 ，选择 f（快速）版本，其他默认。
- SDRAM 控制器，16 位数据位宽，12 位行地址，9 位列地址，用作 NIOS II 的数据和指令存储器
- EPCS 控制器，用以掉电存储 FPGA 和 NIOS II 的软硬件固件信息
- UART（RS232）控制器，用作系统标准输入输出设备，默认 115200 波特率
- Timer 定时器，设置默认 10 毫秒定时器，并可通过软件修改寄存器以修改定时时间。
- PIO，仅输出型 PIO，4 位，对应开发板上 4 个 LED 灯

下图为搭建好的最小 NIOS II 应用系统

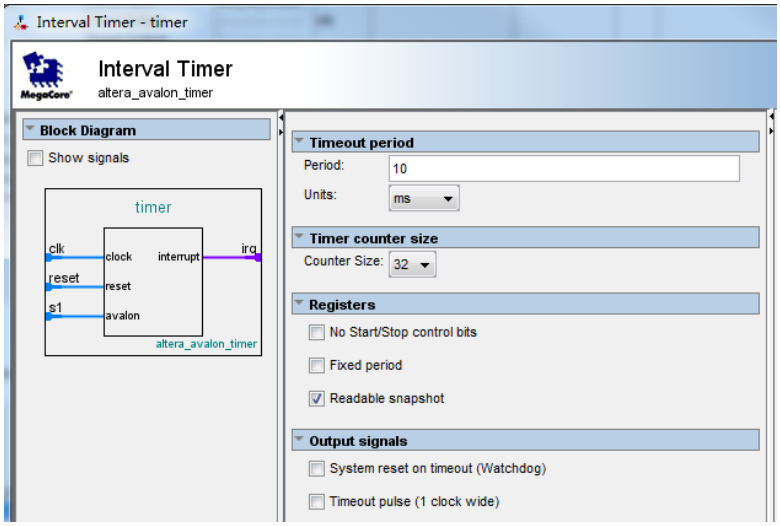


其中需要注意的是，驱动 LED 的 PIO 为 4 位，并直接命名为了 led，uart 串口被命名为了 RS232，这是为了与 RT-Thread 官方提供的 NIOS II 的 BSP 文件中相关代码相对应。

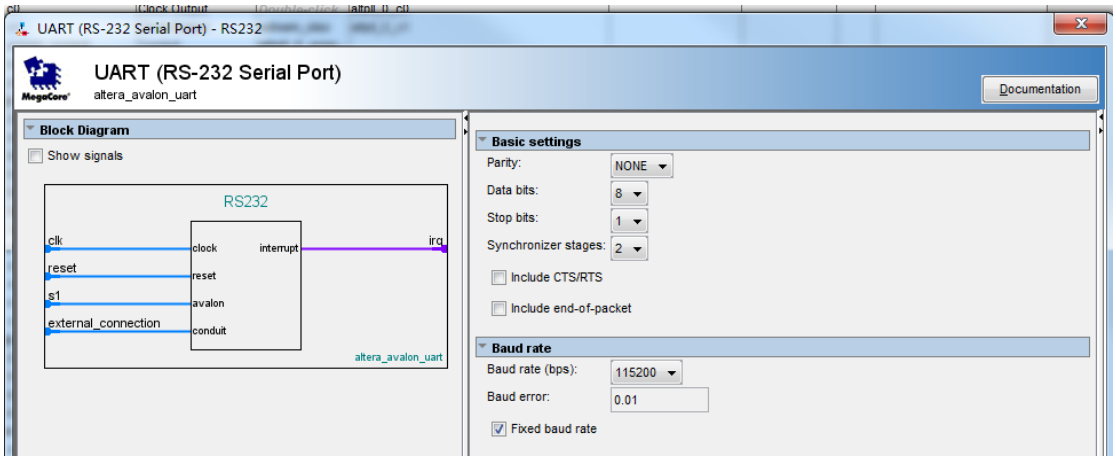
NIOS II CPU 相关设置



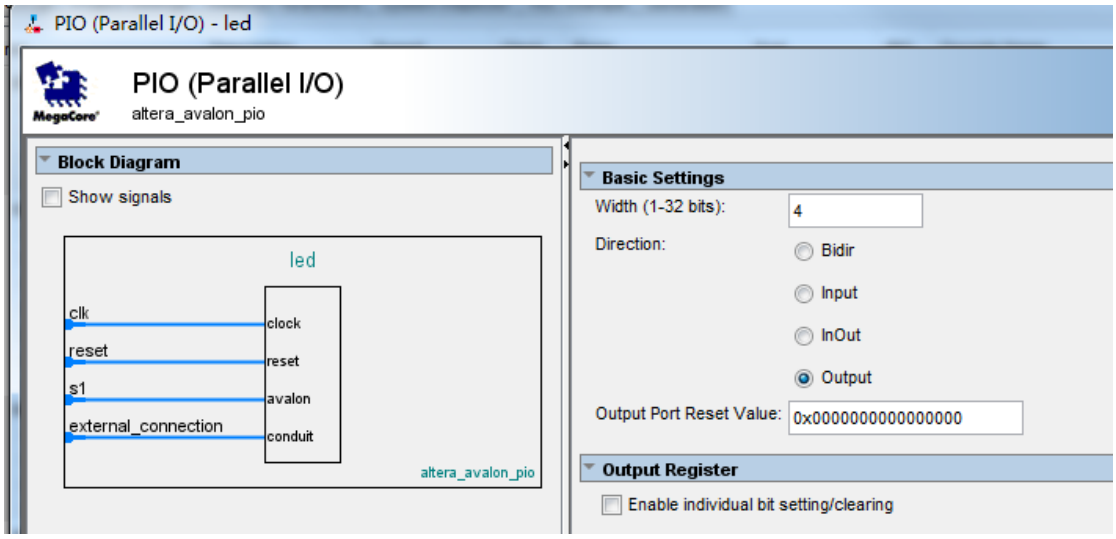
定时器参数设置



UART 控制器相关设置



LED PIO 相关设置



其他的相关细节操作就不在此多说，如果需要了解可以参见小梅哥 SOPC 公开课相关内容。

在 Quartus II 工程中例化 CPU 系统

设计顶层文件

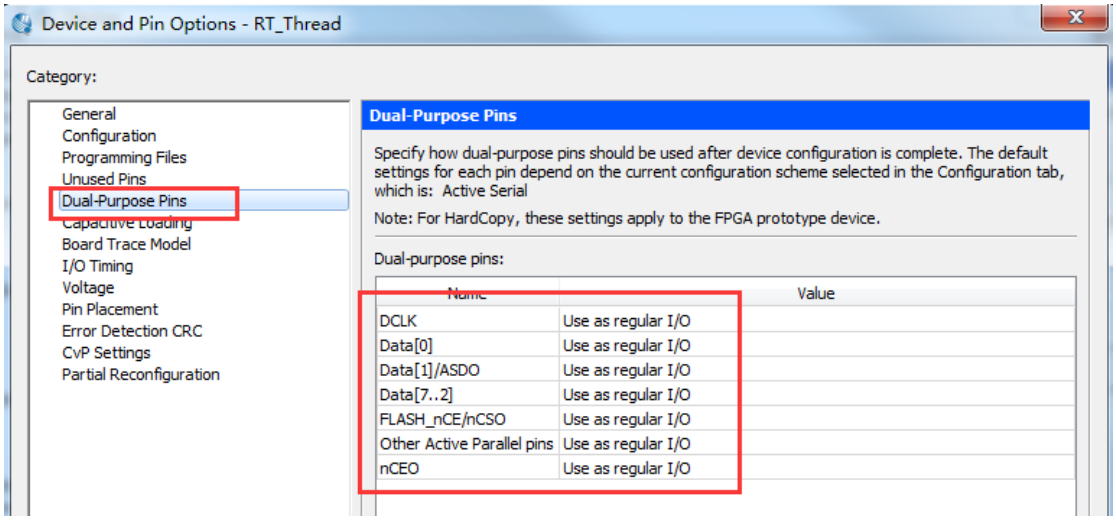
将 Qsys 中搭建的系统命名为 mysystem，然后 generate 得到 HDL 文件，在 Quartus II 软件中添加 mysystem.qsys 文件，新建 Verilog 文件，完善顶层例化信息，完善后的代码如下所示：

```
module ac620 ghrd(
       input  wire        reset n,
       input  wire        clk,
```

```
        output wire [11:0] sdram addr,
        output wire [1:0]  sdram ba,
        output wire        sdram cas n,
        output wire        sdram cke,
        output wire        sdram clk,
        output wire        sdram cs n,
        inout  wire [15:0] sdram dq,
        output wire [1:0]  sdram dqm,
        output wire        sdram ras n,
        output wire        sdram we n,
        output wire [3:0]  led,
        input  wire        uart rxd,
        output wire        uart txd,
        output wire        epcs dclk,
        output wire        epcs sce,
        output wire        epcs sdo,
        input  wire        epcs data0
);

    mysystem u0 (
        .clk in reset reset n          (reset n),
        .clk in clk                    (clk),
        .sdram addr                    (sdram addr),
        .sdram ba                      (sdram ba),
        .sdram cas n                   (sdram cas n),
        .sdram cke                     (sdram cke),
        .sdram cs n                    (sdram cs n),
        .sdram dq                      (sdram dq),
        .sdram dqm                     (sdram dqm),
        .sdram ras n                   (sdram ras n),
        .sdram we n                    (sdram we n),
        .led export                    (led),
        .uart rxd                      (uart rxd),
        .uart txd                      (uart txd),
        .sdram cko clk                 (sdram clk),
        .epcs dclk                     (epcs dclk),
        .epcs sce                      (epcs sce),
        .epcs sdo                      (epcs sdo),
        .epcs data0                    (epcs data0),
        .altpll 0 phasedone conduit export (),
        .altpll 0 locked conduit export    (),
        .altpll 0 areset conduit export    ()
    );

endmodule
```

设置 EPCS 管脚功能

根据 AC620 FPGA 开发板引脚分配表或者开发板背面的丝印对用到的引脚功能一一分配正确的管脚。注意设置所有的 IO 电平为 3.3LVTTL，以防止无法正确驱动 SDRAM 存储器。另外由于使用到了 EPCS，因此需要在 Quartus 中设置其功能为 regular IO，如下图所示：



添加 SDC 约束文件

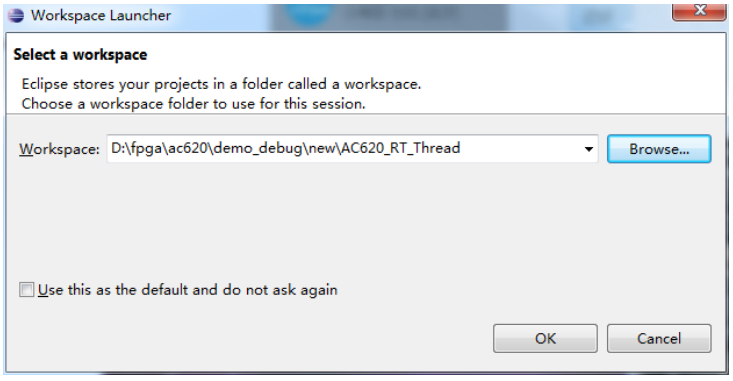
在做基于 NIOS II 的 SOPC 开发时，务必要添加 SDC 约束文件，对系统的时钟进行约束，以保证 Quartus II 软件能够按照运行的频率要求对系统进行布局布线，否则，在没有加约束的情况下，整个系统可能最高运行频率比 50MHz 还要低，就会出现系统无法正常运行，软件无法下载等各种故障。本实例中约束文件比较简单，大家只需要将下述内容新建一个 sdc 文件保存起来并添加到 Quartus II 工程中即可：

```
set_time_format -unit ns -decimal_places 3
create_clock -name {clk} -period 20.000 -waveform { 0.000 10.000 } [get_ports {clk}]
derive_pll_clocks
```

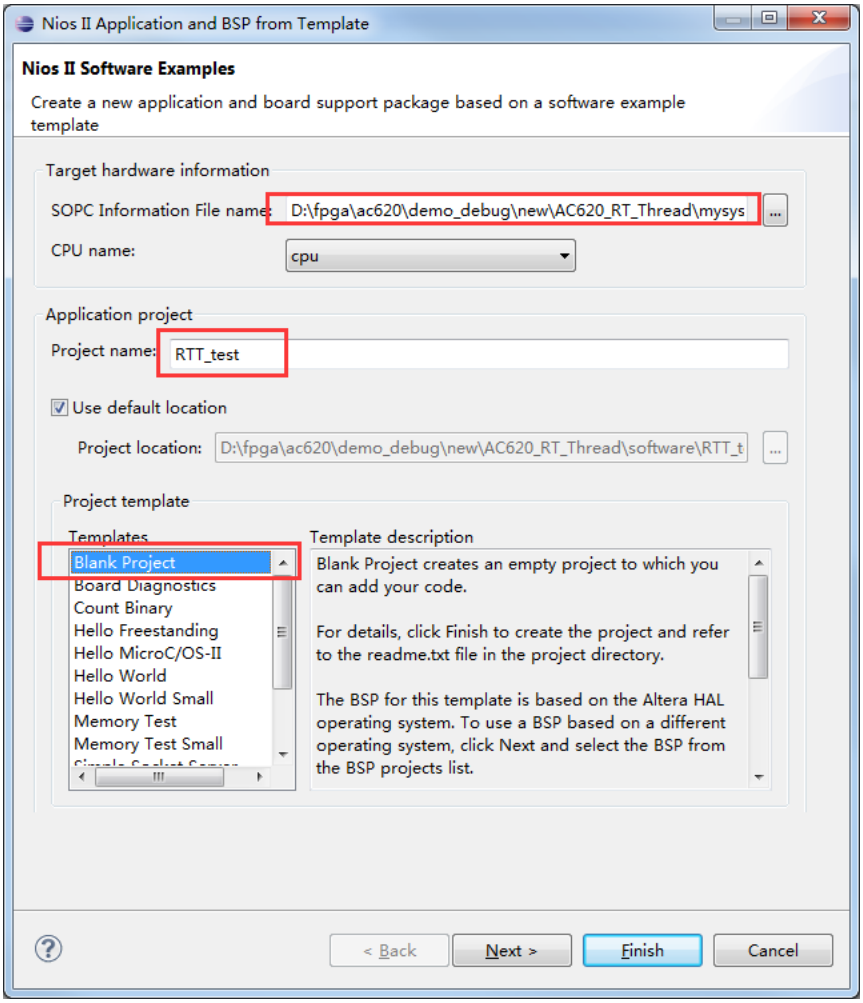
添加完成后编译整个工程，得到 sof 文件。

创建 NIOS II 应用工程模版

打开 Quartus II 集成的 NIOS II 软件开发工具（基于 Eclipse），然后切换工作空间到 Quartus II 工程目录下。



然后新建一个空白模版工程和 bsp 工程，如下所示。注意，sopcinfo 文件路径一定不能错。

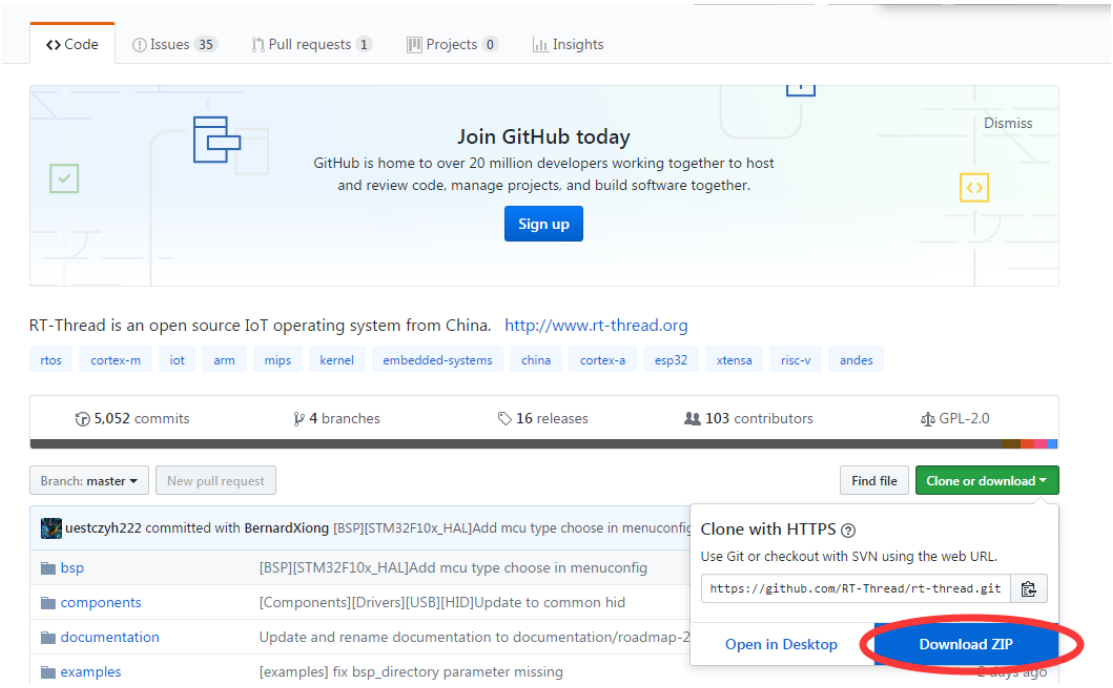


移植 RT-Thread 操作系统

下载 RT-Thread 操作系统源码

从 git 上下载 RT-Thread 操作系统的源文件，源码地址：https://github.com/RT-Thread/rt-thread

选择 master 分支，点击 Clone or download 按钮，选择 Download zip 来下载源文件，当然也可以从我们提供的本实例的压缩包中得到该文件。

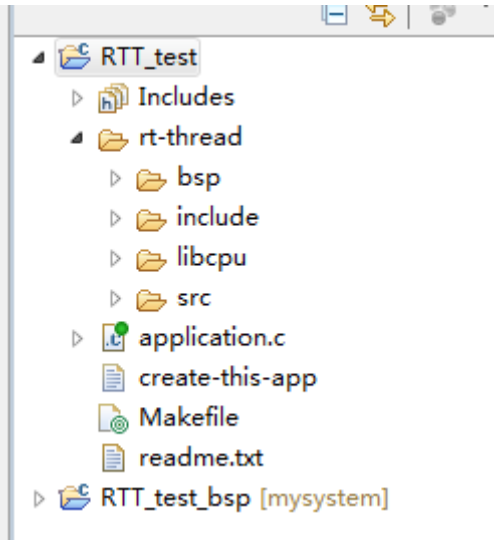


下载完成之后的文件名为：rt-thread-master.zip

移植 RT-Thread 操作系统源码

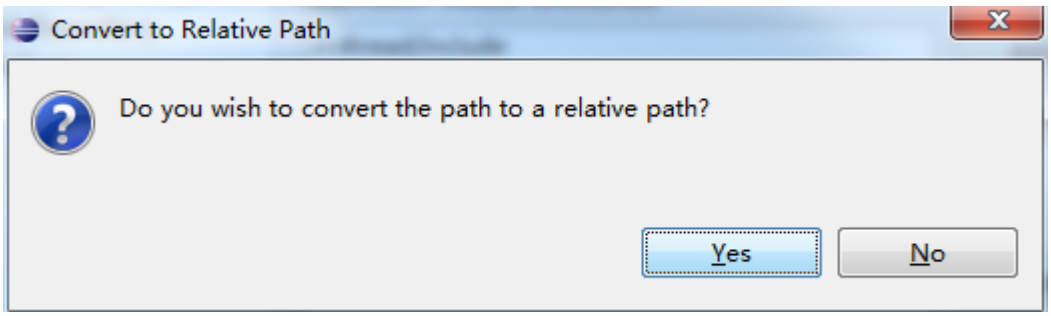
在 RTT_Test 工程下新建一个文件夹，命名为”rt-thread”，添加 src 文件夹，include 文件夹到 rt-thread 文件夹下，然后添加 libcpu 下 nios 部分内容到 rt-thread 文件夹下。注意，libcpu 下提供了对各种体系架构的 cpu 的支持，我们这里只需要 nios 部分即可，其他部分不需要添加到工程中。

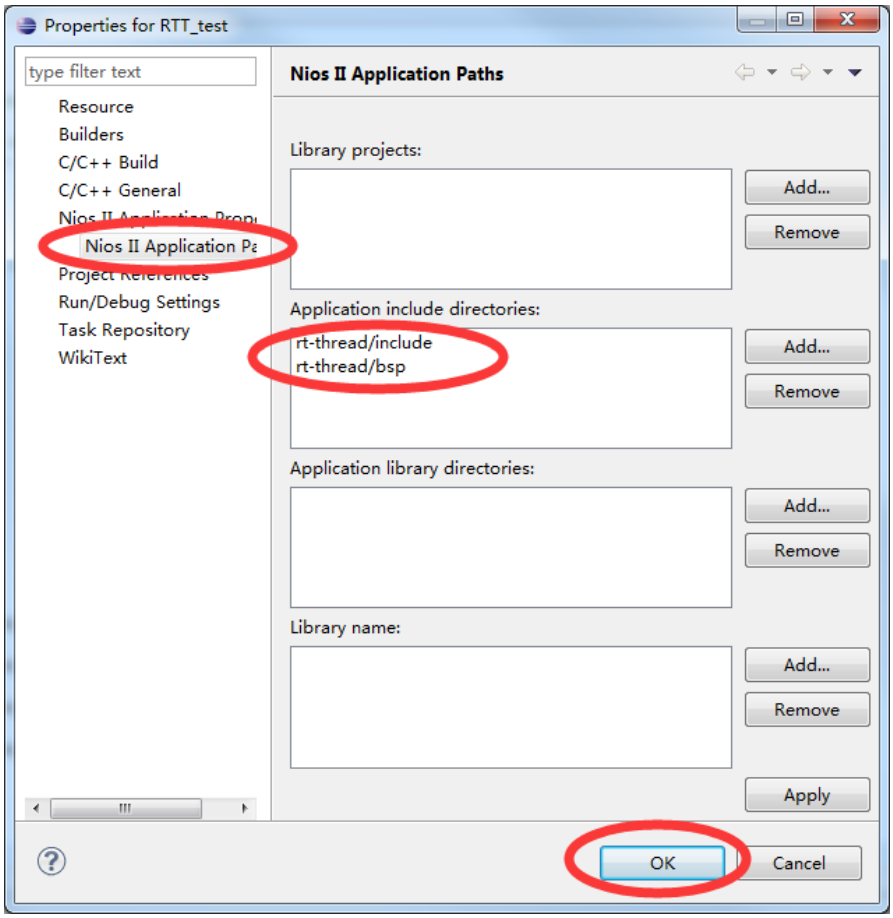
上述就完成了 RT-Thread 操作系统源码的添加。但是此时还并不完整，我们还需要添加对 NIOS II CPU 的 bsp 文件，该文件在 bsp 路径下的 nios ii 路径下，我们拷贝该路径下的所有.c 和.h 文件到工程中，注意，application.c 是整个系统的应用部分，也就是 main 函数所在文件，因此将该文件从 rt-thread 文件夹中移出到软件工程的根目录（拖动即可），添加完成后的工程如下所示（具体可以参考我们提供的实例工程文件）。



设置头文件搜索路径

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 文件夹道头文件路径中，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。

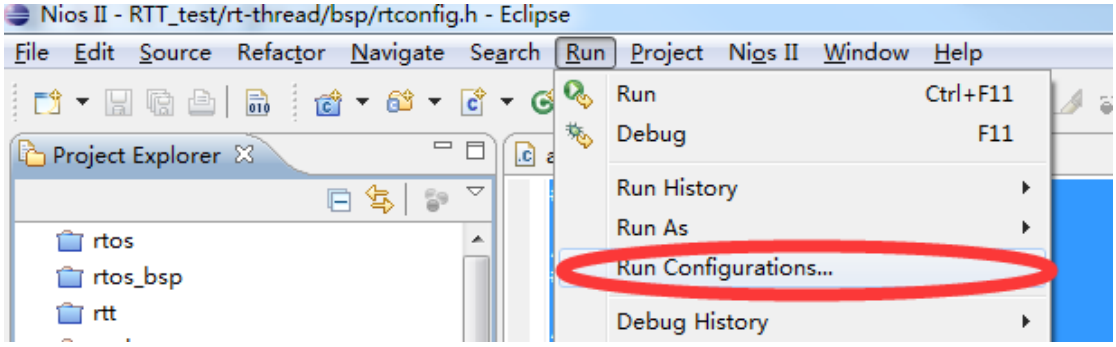
关闭 FINSH 支持功能

接下来，我们打开工程的 rt-thread -> bsp -> rtconfig.h 文件，定位到第 80 行左右，将 #define RT_USING_FINSH 这句话前面加上单行注释符“//”，以屏蔽该定义，即不使用 FINSH 功能。

运行 RT-Thread 操作系统

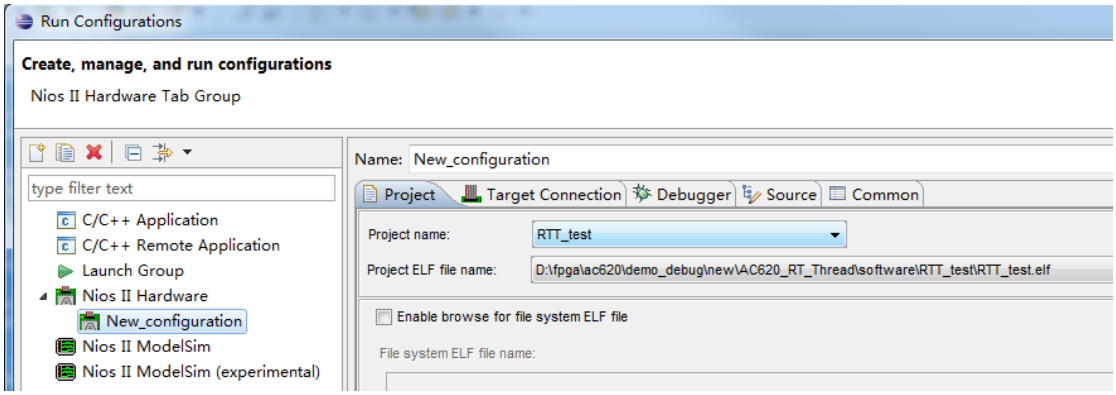
编译运行程序

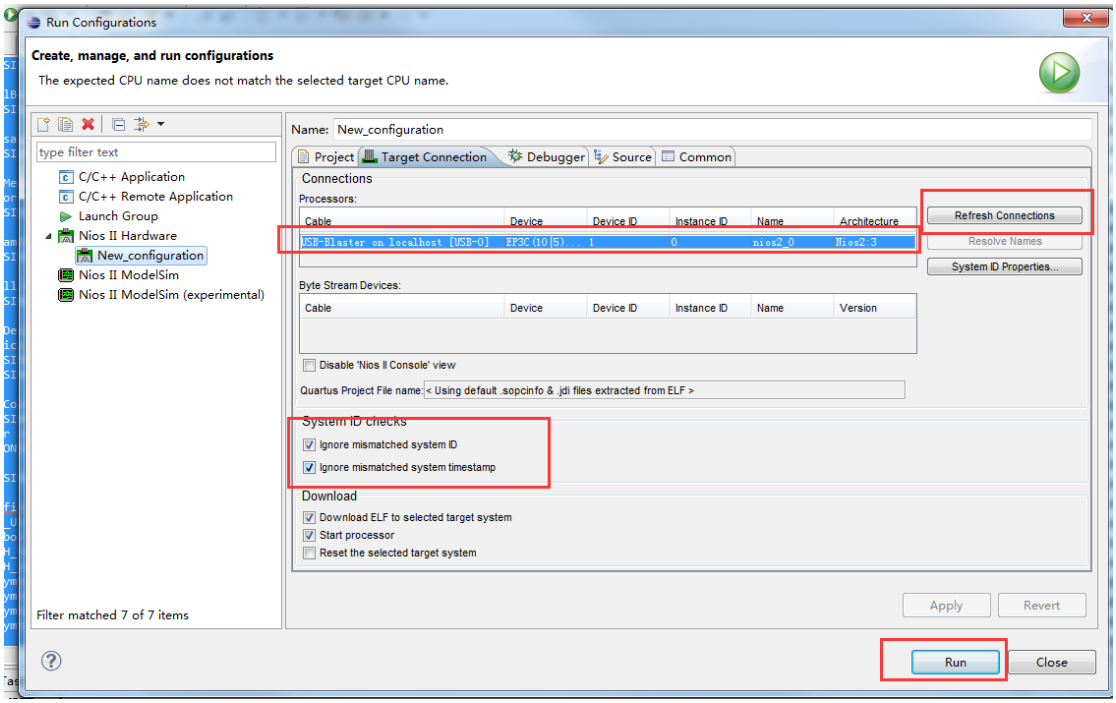
然后对整个工程进行全编译(快捷键为 Ctrl + B)，编译无误，就会得到可下载到 NIOS II CPU 中运行的 elf 文件，然后在菜单栏中依次点击 RUN -> Run Configurations 选项，打开下载运行页面。



新建一个硬件运行配置并选择 RTT_test 工程，然后切换到 Target Connection 选项卡，刷新连接以确认硬件 USB Blaster 已经找到 CPU。

注意，在此操作之前需要先将 Quartus II 软件编译生成的 sof 文件下载到开发板中，并使用 Micro USB 数据线连接开发板和电脑，并打开串口调试工具，以找到开发板对应串口，设置波特率 115200，ASCII 接收。



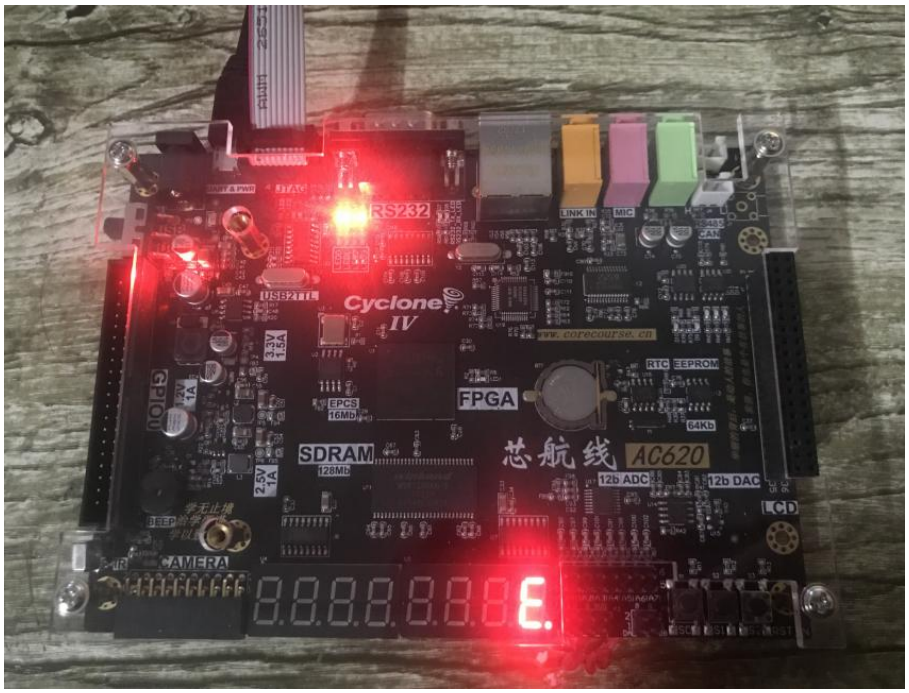


勾选忽略 system id 和时间戳选项，然后点击 Run。即可开始软件程序的下载。

实验结果

下载完成后，即可在串口调试助手上看到系统运行时打印的信息，同时，开发板上 4 个 LED 灯循环闪烁。





小梅哥

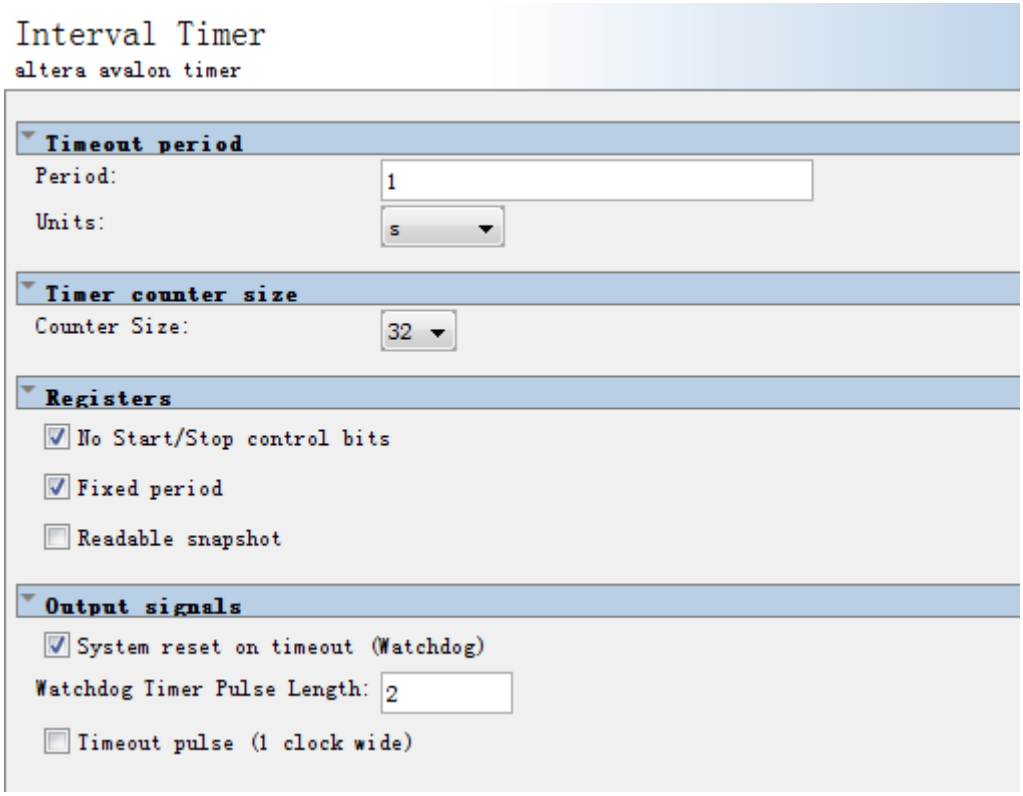
2017 年 11 月 17 日

武汉　光谷

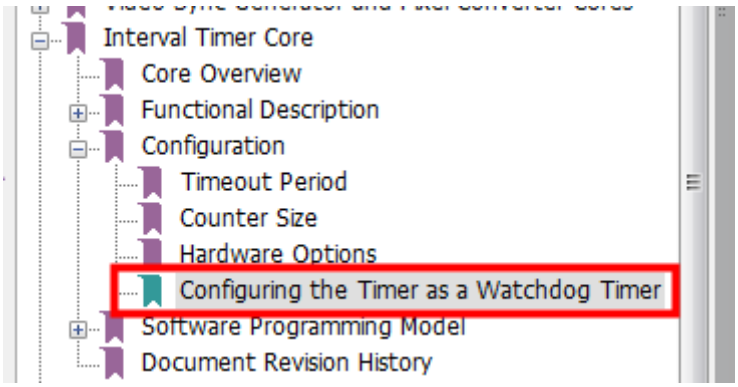
第 7 章 给 NIOS II CPU 增加看门狗定时器并使用

配置看门狗定时器：

1. 设置计时溢出时间为 1 秒
2. 计数器位宽为 32 位
3. 勾选 No Start/Stop control bits
4. 勾选 Fixed period
5. 不勾选 Readable snapshot
6. 勾选 System reset on timeout.(Watchdog)
7. 不勾选 Timeout Pulse （1 clock wide）



这部分配置可以参见“Embedded Peripheral IP User Guide.pdf”中 Interval Timer Core 章下的 Configuration 节下的 Configuring the Timer as a Watchdog Timer 小节相关内容。



Configuring the Timer as a Watchdog Timer

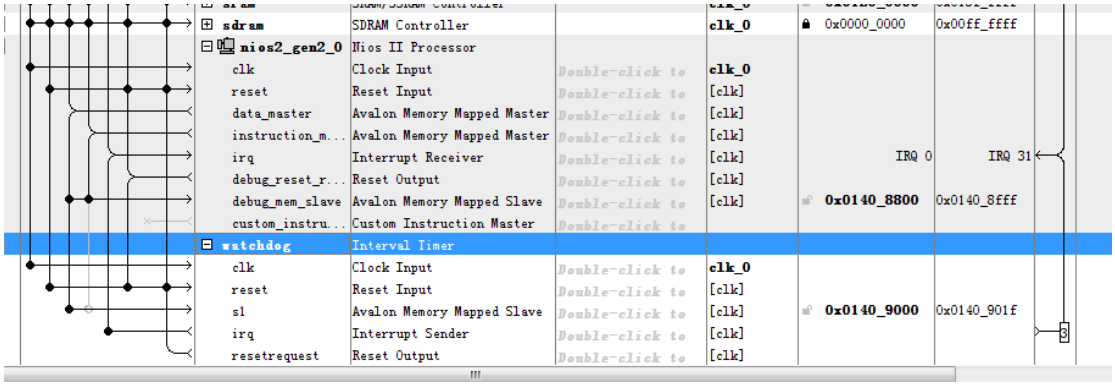
To configure the core for use as a watchdog, in the MegaWizard Interface select **Watchdog** in the **Preset Configurations** list, or choose the following settings:

- Set the **Timeout Period** to the desired "watchdog" period.
- Turn off **Writeable period**.
- Turn off **Readable snapshot**.
- Turn off **Start/Stop control bits**.
- Turn off **Timeout pulse**.
- Turn on **System reset on timeout (watchdog)**.

A watchdog timer wakes up (comes out of reset) stopped. A processor later starts the timer by writing a 1 to the `control` register's `START` bit. Once started, the timer can never be stopped. If the internal counter ever reaches zero, the watchdog timer resets the system by generating a pulse on its `resetrequest` output. The `resetrequest` pulse will last for two cycles before the incoming reset signal deasserts the pulse. To prevent an indefinite `resetrequest` pulse, you are required to connect the `resetrequest` signal back to the reset input of the timer.

To prevent the system from resetting, the processor must periodically reset the timer's count-down value by writing one of the period registers (the written value is ignored). If the processor fails to access the timer because, for example, software stopped executing normally, the watchdog timer resets the system and returns the system to a defined state.

添加完成后，将核名字改为 watchdog（当然也可以自己定义），然后将 clk 信号连接到系统 clk 上，reset 与所有 reset 网络信号相连（不包括 clk 核的 reset_in），将 s1 连接到 nios cpu 的 data_master 上，点击连接 irq，将 resetrequest 连接到所有核的 reset 输入上。

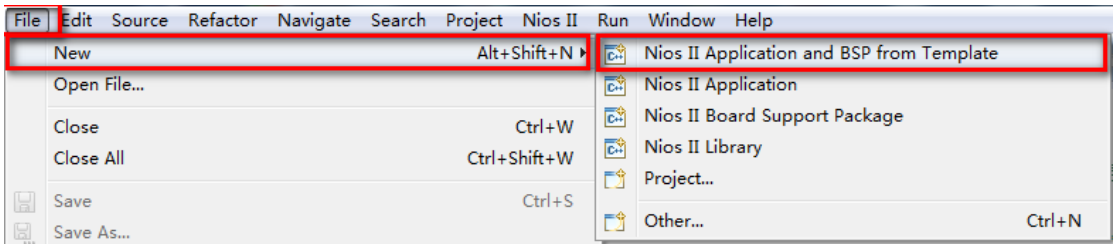


然后生成 QSYS 系统。

创建 Quartus II 工程并编写顶层文件。如果是在上一节实验的基础上增加的看门狗核，则不需要对 Quartus II 工程做任何更改，直接全编译即可。否则请按照我们提供的示例工程创建 Quartus II 工程并编写顶层文件。

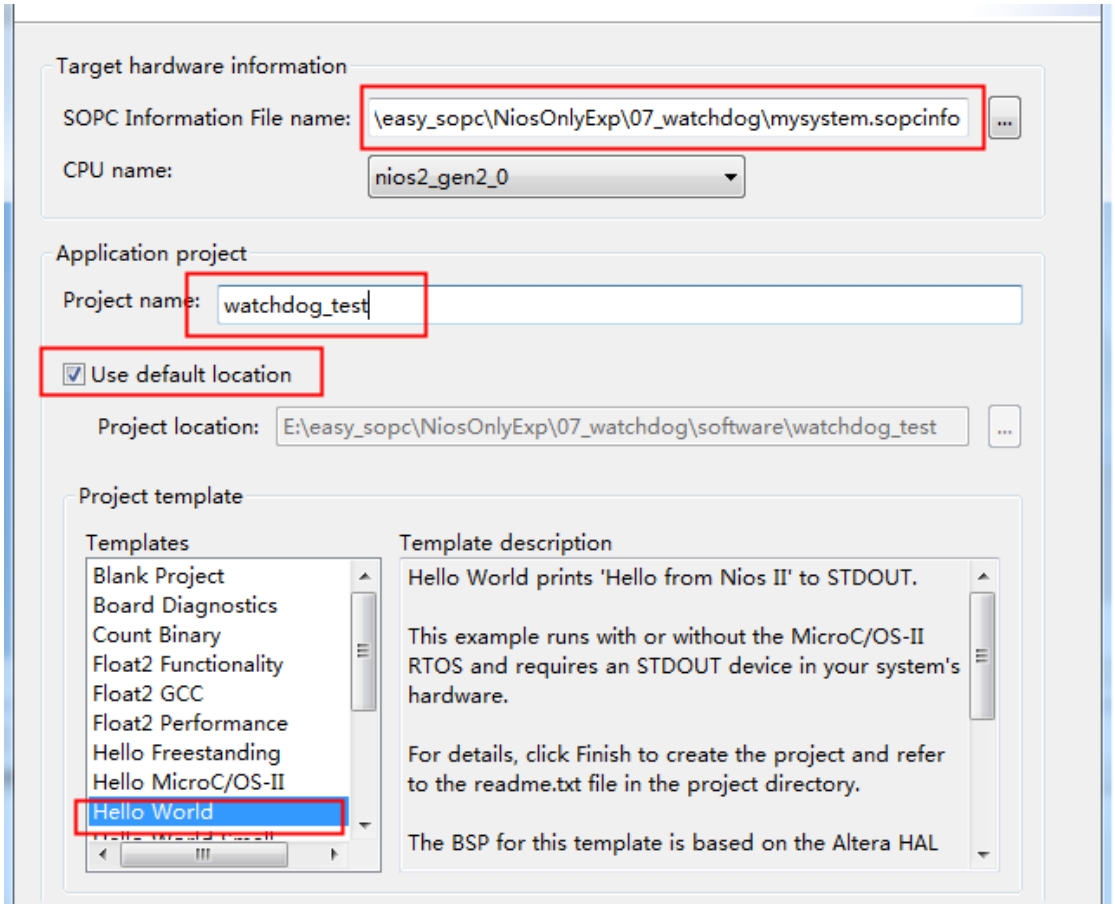
打开 NIOS II 开发软件，切换工作空间到当前工程目录，如果整个工程是直接复制上一个实验的工程得到的，则工作空间中还会有上一个实验的软件工程，这里需要首先删除工作空间中已经有的工程，注意，不要删除工程源文件，因为这时候工程源文件还是指向了上一个工程的源文件所在位置，直接删除源文件会破坏上一个实验的工程。然后在 software 目录下手动删除原有的工程文件。

接着点击【File】->【New】->【Nios II Application and BSP from Template】打开创建新工程和板级支持包向导。

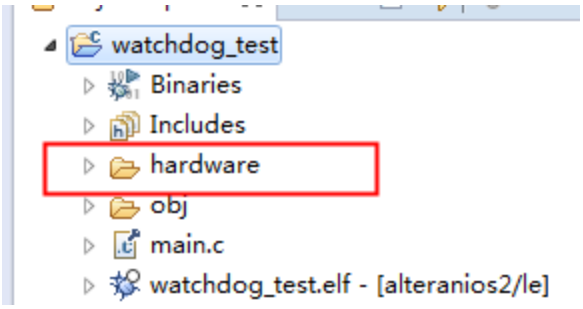


然后选择 sopcinfo 文件为本工程目录下的 sopcinfo 文件（注意，软件总是记住上一次选

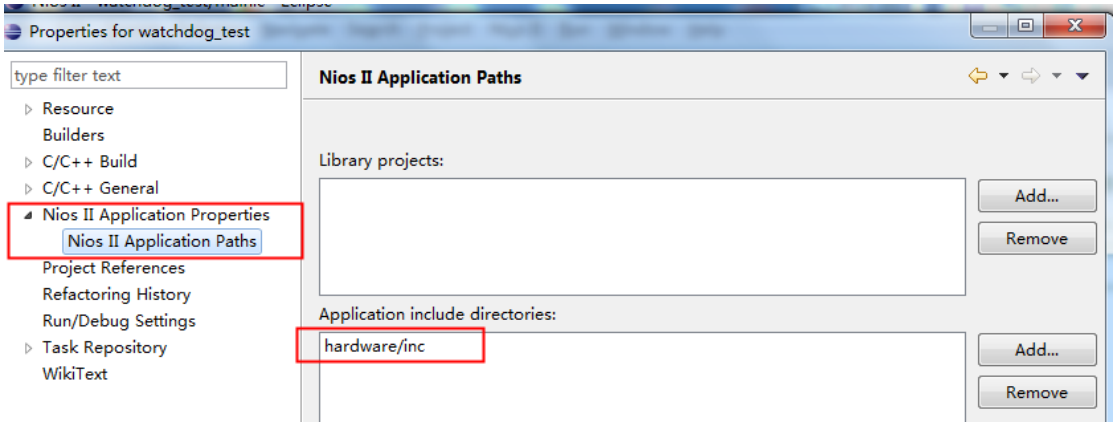
择 sopcinfo 文件时的路径，因此这里特别容易出错，我们需要手动重新定位目录，软件自动打开的那个目录下的 sopcinfo 文件往往都是上一个实验的，不能使用）创建 hello world 模版工程，命名工程为 watchdog_test。然后点击 finish 即可创建新工程。



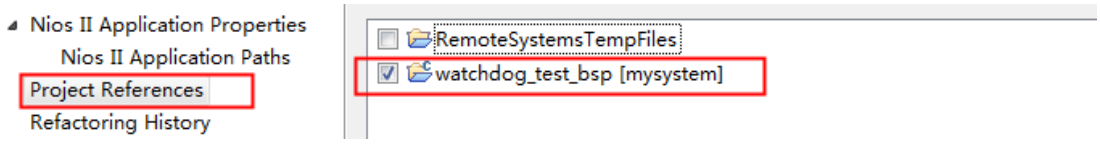
复制上一个工程中软件工程中的【hardware】文件夹到本工程下的【watchdog_test】文件夹下，最好直接在电脑的文件管理器中进行（对 Eclipse 使用非常熟练的，请尽情使用拖拽添加的方式）。复制完成后，选中【watchdog_test】工程，单击右键，选择【Refresh】，就可以在工程中看到【hardware】文件夹了。



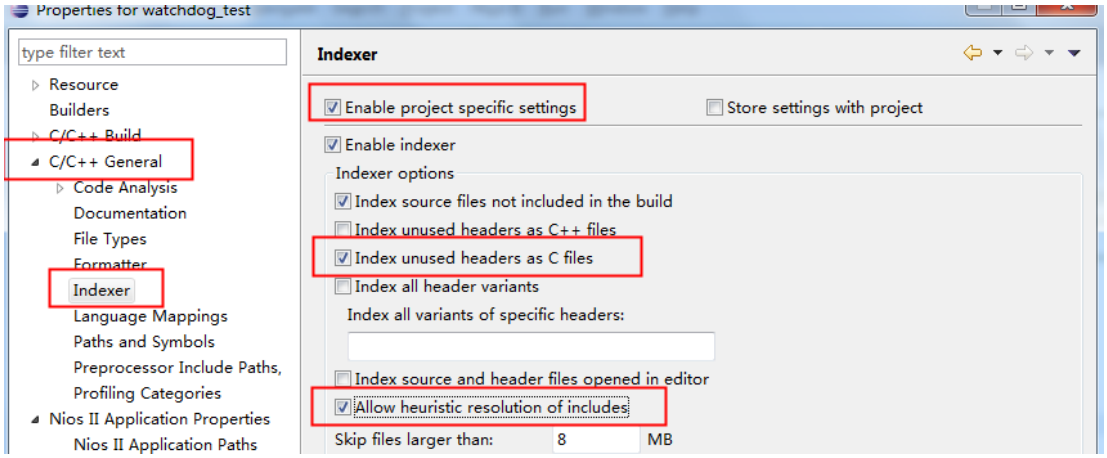
接着我们添加【hardware】下的【inc】路径到工程头文件包含路径中来。我们选中【watchdog_test】工程，按下键盘组合键“ATRL + Enter”键打开【Properties】设置界面，选择 Nios II Application Properties 下的 Nios II Application Paths，在右侧的 Application include directories 下，点击 Add 按钮，添加 hardware/inc 到包含路径中。然后在弹出的对话框中点击 Yes，即可将此路径添加为我们的头文件包含路径。如果用户之后自己有其他的头文件路径需要添加，也是按照这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



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。



然后我们再在 C/C++ General 下的 Indexer 下，勾选按照下图中所示进行勾选。通过这样勾选后，一般工程不管怎么折腾，都不会出现 xxxx 'xxxx' could not be resolved 的报错了。



设置完成后，我们点击 Apply，然后点击 OK 退出。

然后我们选中【watchdog_test_bsp】工程，单击右键选择【Nios II】->【BSP Editor】，设置 sys_clk_timer 为 none,（软件会默认搜索工程中存在的定时器外设并添加为 sys_clk_timer），因此我们这里可能看到打开时默认将 watchdog 设置为了 sys_clk_timer，所以我们要手工更改为 none，如果不更改，系统将不能正常运行。Timestame_timer 设置为 none。Stdin、stdout、stderr 都设置为 uart_0。（默认为 jtag_uart，但是 jtag_uart 容易导致 Eclipse 崩溃，因此我一般不用）。设置完毕后点击 generate，然后点击 exit 退出。

接着我们在 hardware/src 目录下创建一个“watchdog.c”文件，编写内容如下：

```
#include "watchdog.h"


/*****************************************
 * 驱动使用的看门狗定时器基地址，独立定义，然后初
 * 始化时由用户输入具体基地址名称，避免不同用户对看
```

```
 * 门狗定时器不同的命名带来的程序移植性问题。
*****************************************/
alt_u32 WTD_BASE;

/*****************************************************************
* 名    称：WatchDog_Init()
* 功    能：初始化 WatchDog,即开启看门狗定时器；
*         需要注意的是，看门狗定时器一旦开启，无法关闭。
* 入口参数：WATCHDOG_BASE  :看门狗定时器基地址
* 出口参数：无
*****************************************************************/
void  WatchDog_Init(void *MyWatchDog_BASE)
{
    //将系统的 watchdog 基地址传递给驱动使用的基地址
    WTD_BASE = MyWatchDog_BASE;
    /* 启动 WATCHDOG */
   IOWR_ALTERA_AVALON_TIMER_CONTROL(WTD_BASE,
          ALTERA_AVALON_TIMER_CONTROL_START_MSK);
}

/*****************************************************************
* 名    称：WatchDog_Feed()
* 功    能：看门狗喂狗操作，向看门狗寄存器写入任意值即可完成看门狗复位
* 入口参数：无
* 出口参数：无
*****************************************************************/
void  WatchDog_Feed(void)
{
    IOWR_ALTERA_AVALON_TIMER_PERIODL(WTD_BASE, 0x1234);
}
```

接着我们在 hardware/inc 目录下创建一个“watchdog.h”文件，编写内容如下：

```
#include "system.h"
#include "altera_avalon_timer_regs.h"
#include "alt_types.h" //Altera 定义的数据类型

#ifndef WATCHDOG_H_
#define WATCHDOG_H_

void  WatchDog_Init(void *MyWatchDog_BASE);
void  WatchDog_Feed(void);

#endif /* WATCHDOG_H_ */
```

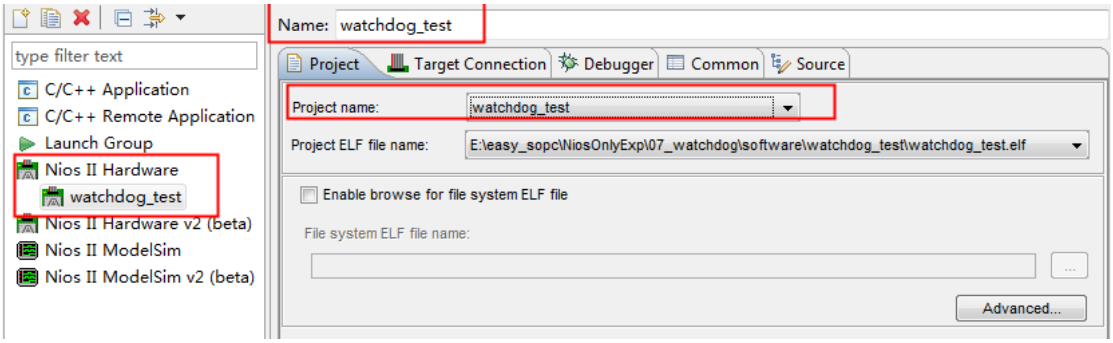
最后我们编写 main 文件函数如下所示：

```
/*************************************************************
* 文 件 名：main.c
* 功    能：运行 WDT，并控制 LED0--LED3 显示输出。程序开始先对 LED0--LED3 进行闪烁控制
*           和喂狗处理；然后只点亮 LED1，并进入死循环，等待 WDT 复位。
* 说    明：
*************************************************************/
#include <stdio.h>
#include "key.h"
#include "led.h"
#include "uart.h"
#include "string.h"
#include "stdlib.h"
#include "priv/alt_busy_sleep.h"
#include "watchdog.h"

//#define USE_WTD 1 //该宏定义决定系统中的看门狗是否被使用

/*************************************************************
* 名    称：DelayNS()
* 功    能：长软件延时，具有喂狗功能。
* 入口参数：dly    延时参数，值越大，延时越久
* 出口参数：无
*************************************************************/
void  DelayNS(alt_u32  dly)
{
    alt_u32  i;
    for(; dly>0; dly--)
    {
        for(i=0; i<1000; i++)
        {
            #ifdef USE_WTD
                WatchDog_Feed();    //喂狗
            #endif
        }
    }
}

/*************************************************************
* 名    称：main()
* 功    能：初始 I/O 口及 WDT，然后开始先对 LED0--LED3 闪烁控制，并进行喂狗处理；
*           然后只点亮 LED1，并进入死循环，等待 WDT 复位。
```

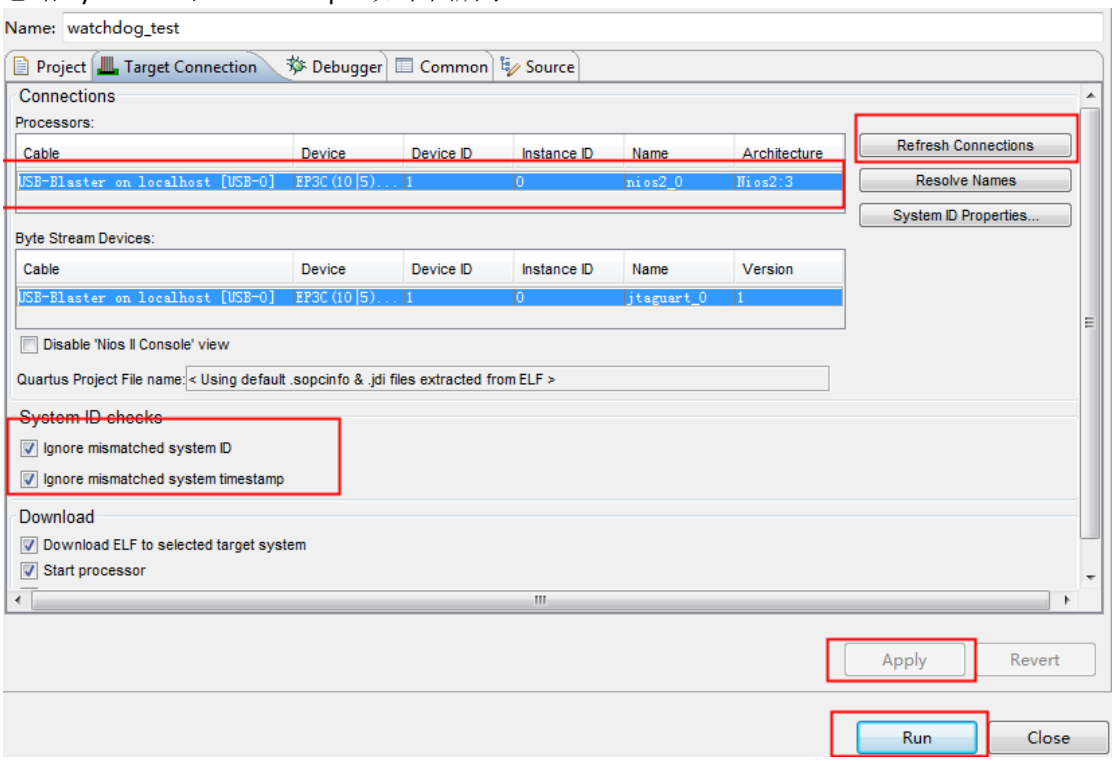
```
* 说    明：如果 main 函数中 InitWDT()被使用，则系统会自动复位，现象就是 led 永远不停的闪烁
*           如果 main 函数中 InitWDT()被屏蔽，则系统无法自动复位，现象就是 led 闪烁 8 次后
*           停止下来，led0 保持点亮
*************************************************************/
int main(void)
{
    alt_u8  i;

    LED_HANDLE hLED;
    hLED = LED_Init(PIO_LED_BASE);
    if (!hLED) {
        printf("Failed to init LED\n");
    }

#ifdef USE_WTD
    //屏蔽此句可以禁止使用看门狗，用来测试不使用看门狗时的系统运行情况
    WatchDog_Init(WATCHDOG_BASE);              // 初始化看门狗
    WatchDog_Feed();             // 进行喂狗操作
#endif

    for(i=0; i<8; i++)
    {
        //点亮 LED0 和 LED2，熄灭 LED1 和 LED3
        LED_WriteData(hLED, LED0 | LED2);
        DelayNS(300);

        //点亮 LED1 和 LED3，熄灭 LED0 和 LED2
        LED_WriteData(hLED, LED1 | LED3);
        DelayNS(300);
    }
    //关闭 LED0~3
    LED_Off(hLED, LED0 | LED1 | LED2 | LED3);
    LED_On(hLED, LED1); //打开 LED1
    while(1);
    return(0);
}
```

这样我们的软件就编写完成了，接着我们点击键盘的组合键“CTRL + B”（或者依次点击【Project】【Build All】）来编译整个工程。

接着我们在 Quartus II 中打开配置下载窗口将 sof 文件下载到 FPGA 中，然后在 Eclipse 中点击【Run】->【Run Configuration】，在弹出的界面中，双击左侧的【Nios II Hardware】新建一个下载设置，将 name 更改为 Watchdog_test，与工程名保持一致。右侧 Project Name 选择 Watchdog_test，如下图所示：



然后切换到 Target Connection 选项卡，查看如果 Connections 下能找到 USB Blaster 就行，如果不能找到，就点击 Refresh Connections，以让软件找到连接。在 System ID Checks 下勾选忽略 System ID 和 timestamp，如下图所示：



然后点击 Apply，再点击 Run 即可开始下载。下载完成后，我们看到开发板上的 4 个 LED 灯闪烁 8 次后停止下来，这是因为没有使用 watchdog 的缘故，程序正常运行到 while（1）处进入死循环。

接着，我们将“//#define USE_WTD 1”这一行前面的“//”注释标识符取消，然后重新编译工程，并运行下载。这里我们可以看到，开发板上的 4 个 LED 灯将持续不停的闪烁。这是因为当程序运行到 while(1)死循环中后，不再有喂狗操作，因此看门够计数溢出，然后复位系统，系统重新开始运行，则 LED 重新开始闪烁，如此持续不断的进行。

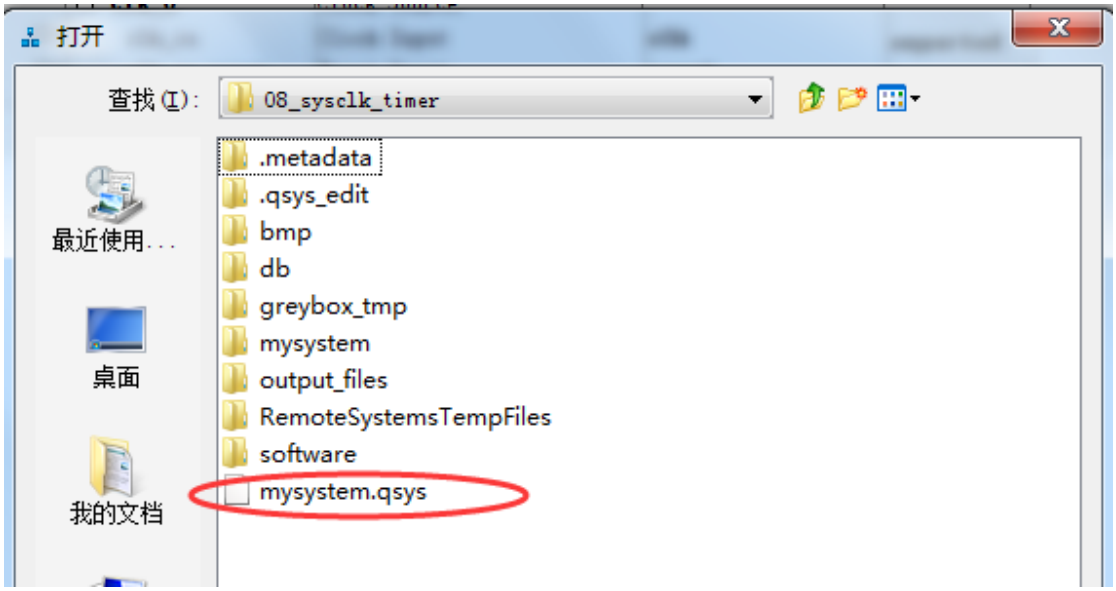
第 8 章　给 NIOS II CPU 添加一颗澎湃的心——系统时钟的使用

本实验介绍如何在 Qsys 中添加一个定时器作为 NIOS II 的心跳定时器，并在 NIOS II 中软件编程使用该定时器。

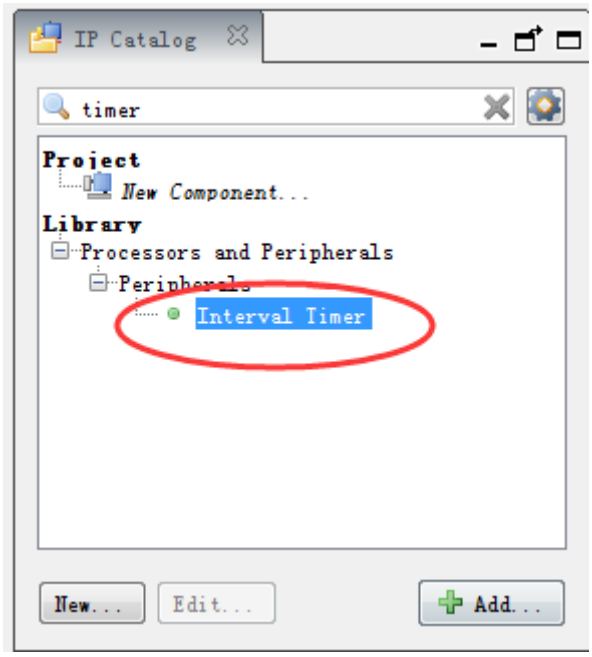
将上一个实验 watchdog 工程复制、粘贴并重命名为 08_sysclk_timer。在 Quartus II 中打开该工程，然后点击 Qsys 快捷图标打开 Qsys 组件，如下图所示：



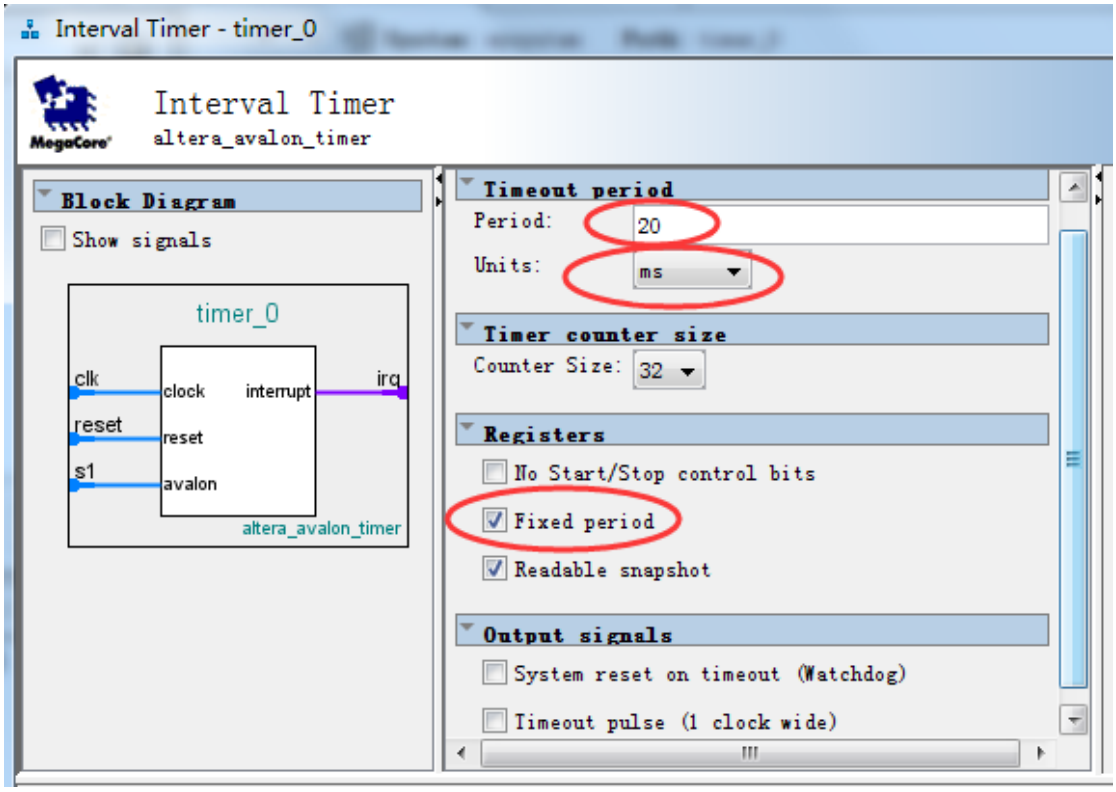
在 Qsys 中，打开工程目录（08_sysclk_timer）下的 mysystem.qsys 工程。



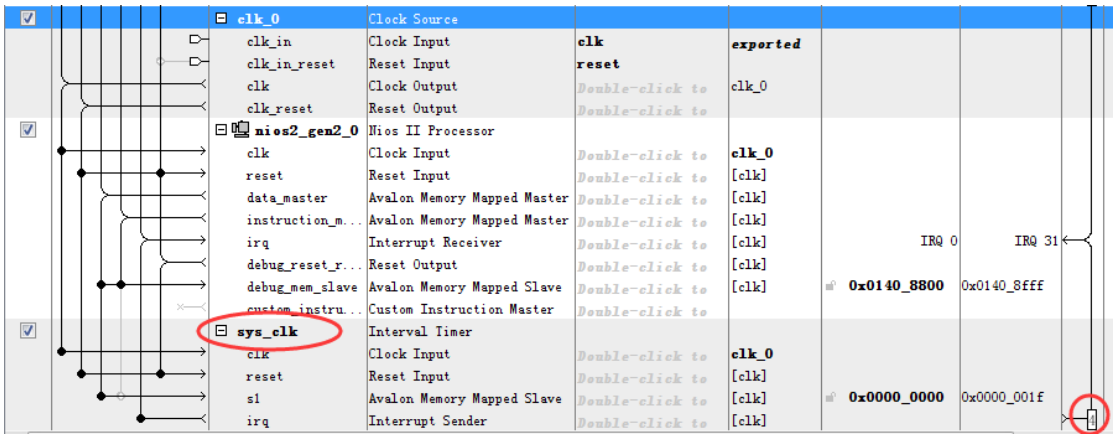
打开后，在左侧的 IP 列表中输入“timer”，然后双击 Interval Timer 打开参数配置界面:



设置 Period 为 20，单位（Units）为 ms，勾选上 Fixed period。其他保持不变，点击 finish，则 Timer 被加入到了我们之前的已有的系统中。

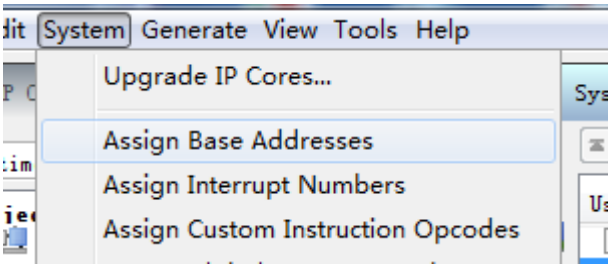


修改 timer 的名字为 sys_clk 按照上一节看门狗的连线内容将所有信号连接到对应总线上。



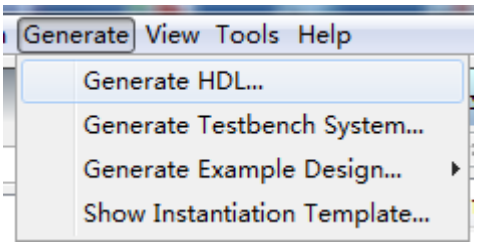
添加的过程中下面信息栏会有报错，暂时不用理会，等我们后面的操作完成后，报错会自动消失。

然后在菜单栏中点击【System】->【Assign Base Addresses】自动分配基地址。如下图所示：

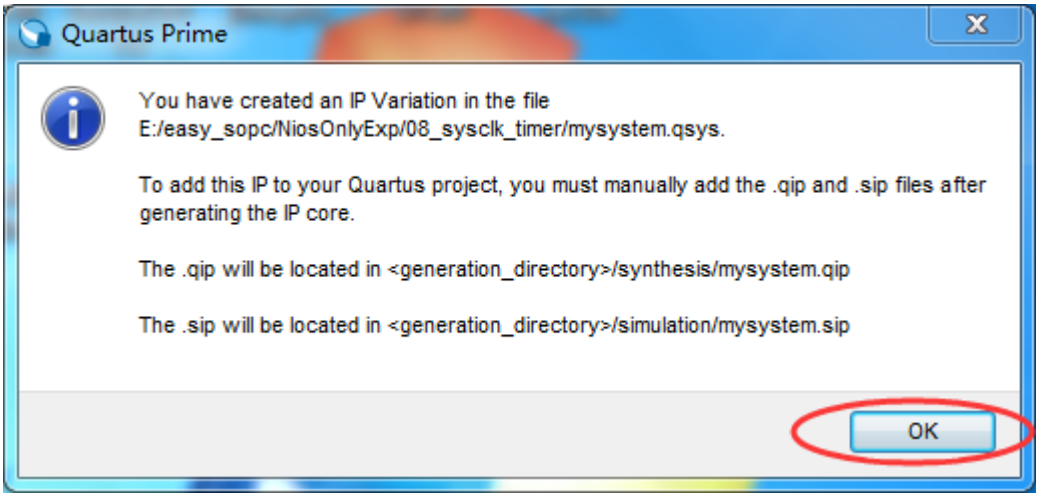


这个时候，应该所有错误都消失了。接着我们在菜单栏中点击【Generate】->【Generate HDL】

来生成 Quartus II 软件可用的设计文件，如下图所示：

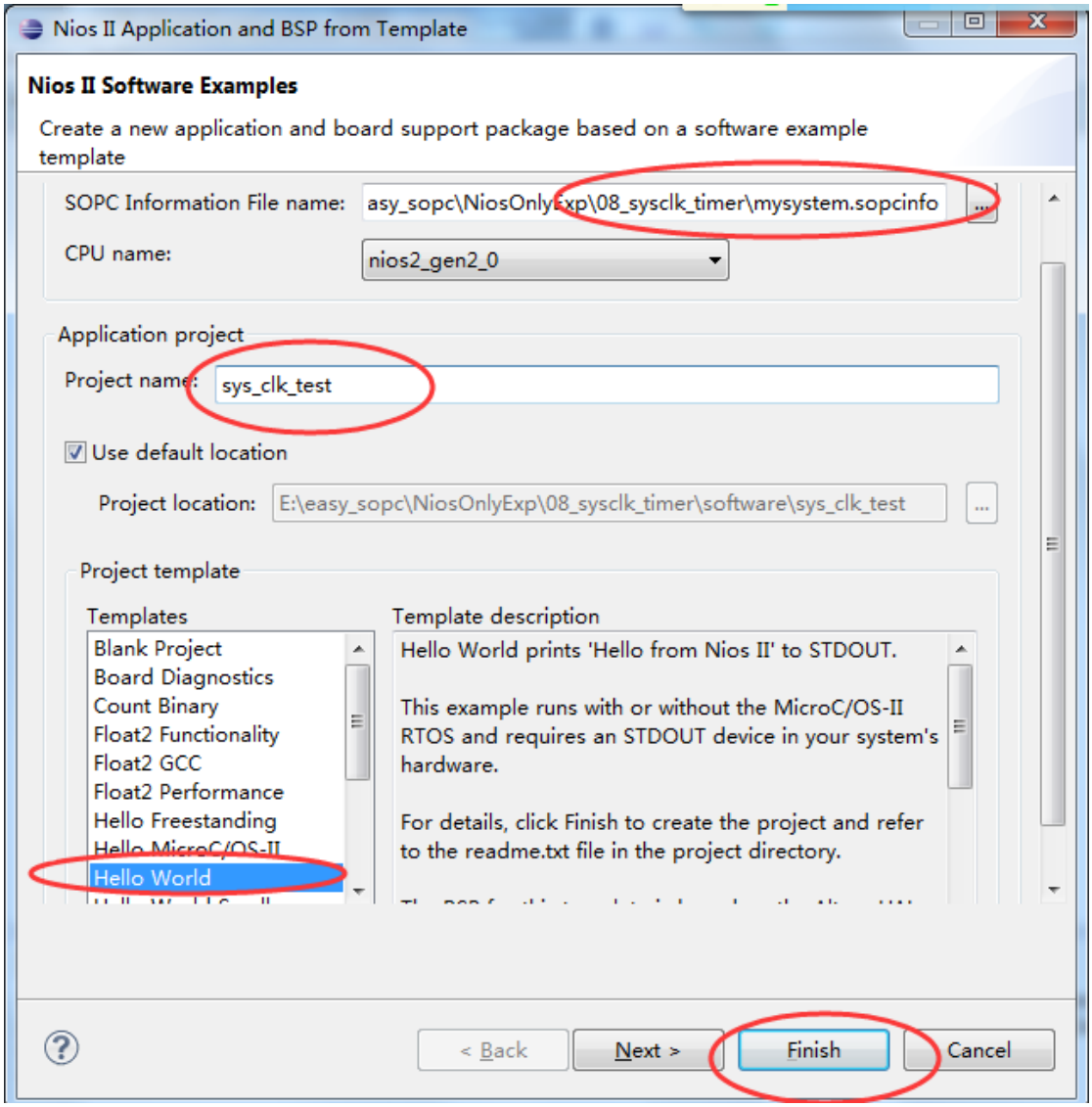


生成完成之后，我们点击 finish 即可关闭 Qsys 系统，同时 Quartus II 软件弹出提示对话框，点击 OK 以关闭。



因为本实验中我们依然没有添加需要引出到外部引脚的 IP，引出在 Quartus II 中对系统的例化文件不需要进行任何修改，直接点击“Start Compilation”（或者使用 Ctrl + L 组合键）对工程进行全编译即可。

打开 NIOS II IDE 开发软件。创建 NIOS II 软件工程和板级支持包工程，选择 sopcinfo 文件为“E:\easy_sopc\NiosOnlyExp\08_sysclk_timer\mysystem.sopcinfo”（自己根据具体路径修改，注意切换路径）。创建工程名字为 sys_clk_test，选择 Hello World 模版工程，然后点击 Finish。如下图所示：

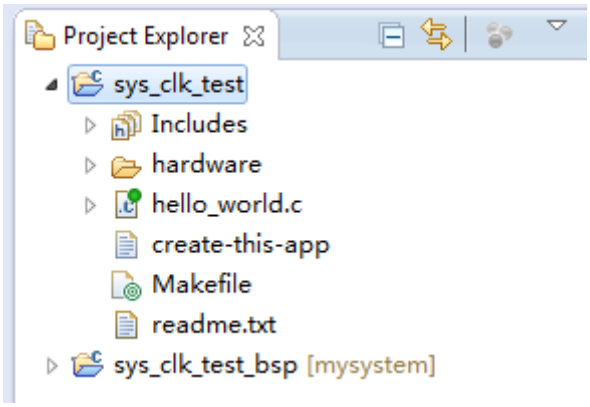


选中 bsp 工程（sys_clk_test_bsp），进入 BSP Editor。

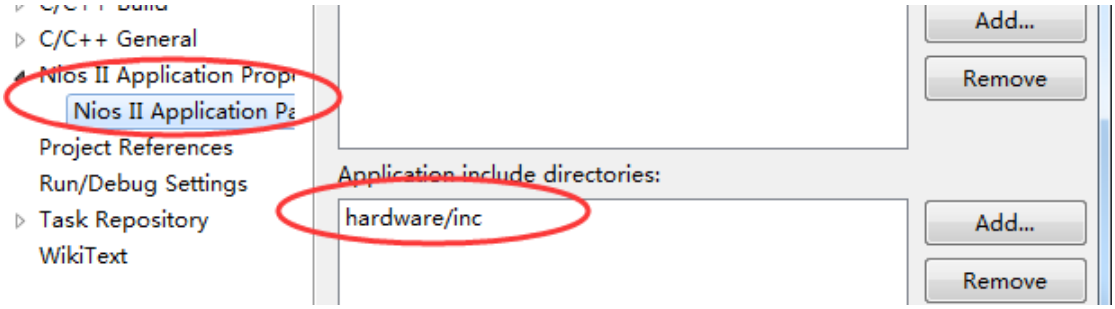
1. 设置 sys_clk_timer 为“sys_clk”，
2. 设置 timestamp_timer 为 none，
3. 设置 stdin、stdout、stderr、为 uart_0。

其他不变，点击 generate，然后点击 exit 退出。

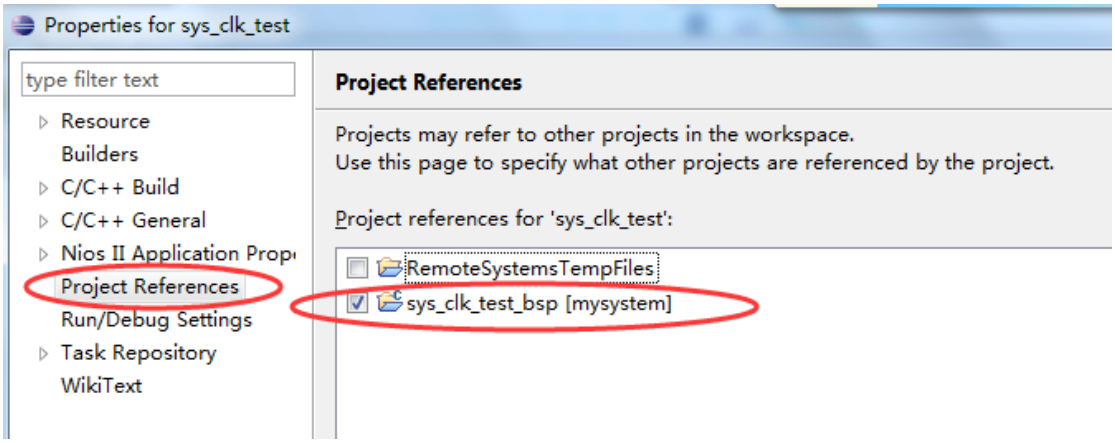
复制上一个工程中软件工程中的【hardware】文件夹到本工程下的【sys_clk_test】文件夹下，最好直接在电脑的文件管理器中进行（对 Eclipse 使用非常熟练的，请尽情使用拖拽添加的方式）。复制完成后，选中【sys_clk_test】工程，单击右键，选择【Refresh】，就可以在工程中看到【hardware】文件夹了。



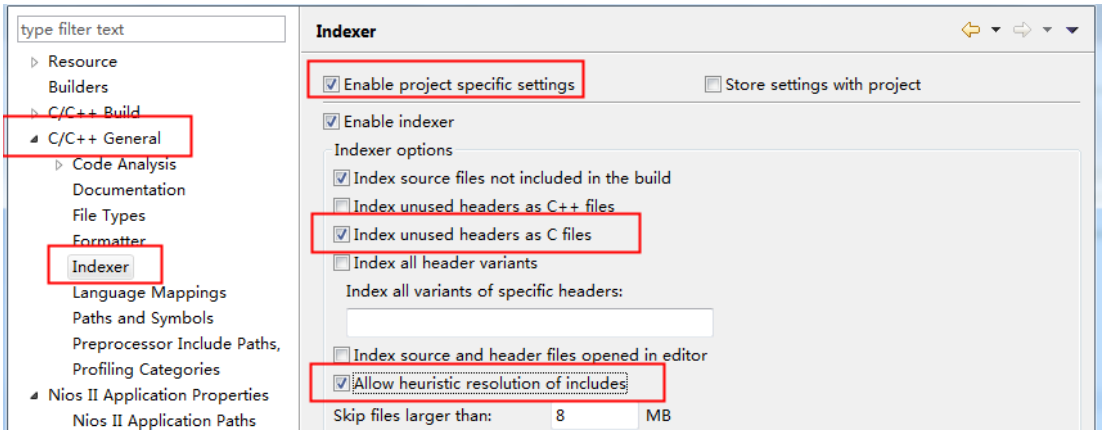
接着我们添加【hardware】下的【inc】路径到工程头文件包含路径中来。我们选中【sys_clk_test】工程，按下键盘组合键“ATRL + Enter”键打开【Properties】设置界面，选择 Nios II Application Properties 下的 Nios II Application Paths，在右侧的 Application include directories 下，点击 Add 按钮，添加 hardware/inc 到包含路径中。然后在弹出的对话框中点击 Yes，即可将此路径添加为我们的头文件包含路径。如果用户之后自己有其他的头文件路径需要添加，也是按照这种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



接着我们查看下 Project References 中是否勾选了 sys_clk_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。



然后我们再在 C/C++ General 下的 Indexer 下，勾选按照下图中所示进行勾选。通过这样勾选后，一般工程不管怎么折腾，都不会出现 xxxx 'xxxx' could not be resolved 的报错了。



我们可以选择将系统创建时默认的 main 函数所在文件 hello_world.c 文件改名为 main.c，修改时只需要选中 hello_world.c 文件，单击右键，选择 rename，然后改为 main.c 即可。

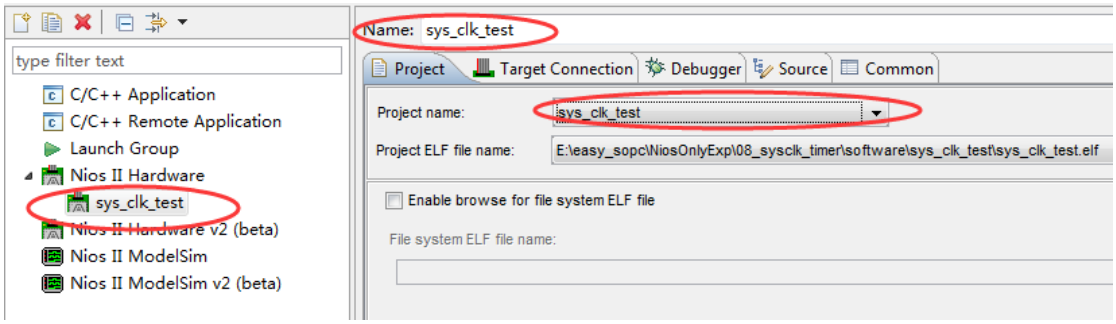
最后我们编写 main 文件函数如下所示：

```
/*************************************************************************
 * 文 件 名：main.c
 * 功    能：利用系统时钟服务产生 1s 的周期性事件，并借此控制 LED 闪烁，
 *
 * 说    明：观察 LED 闪烁效果。

**************************************************************************/
#include <stdio.h>
#include "system.h"                    //包含基本的硬件描述信息
#include  "led.h"
#include "sys/alt_alarm.h"              //系统时钟服务头文件
#include "altera_avalon_pio_regs.h"     //包含基本的 IO 寄存器信息
#include "alt_types.h"                 //Altera 定义的数据类型


static alt_alarm alarm;              //按调用 API 函数规定定义的变量

alt_u8 led_data;

LED_HANDLE hLED;

/*************************************************************************
 * 名    称：my_alarm_callback
 * 功    能：按调用规定系统时钟回调函数，在该函数中实现用户功能
 * 入口参数：context，系统传给回调函数的参数
 * 出口参数：返回下一次的系统时钟服务的周期值

**************************************************************************/
```

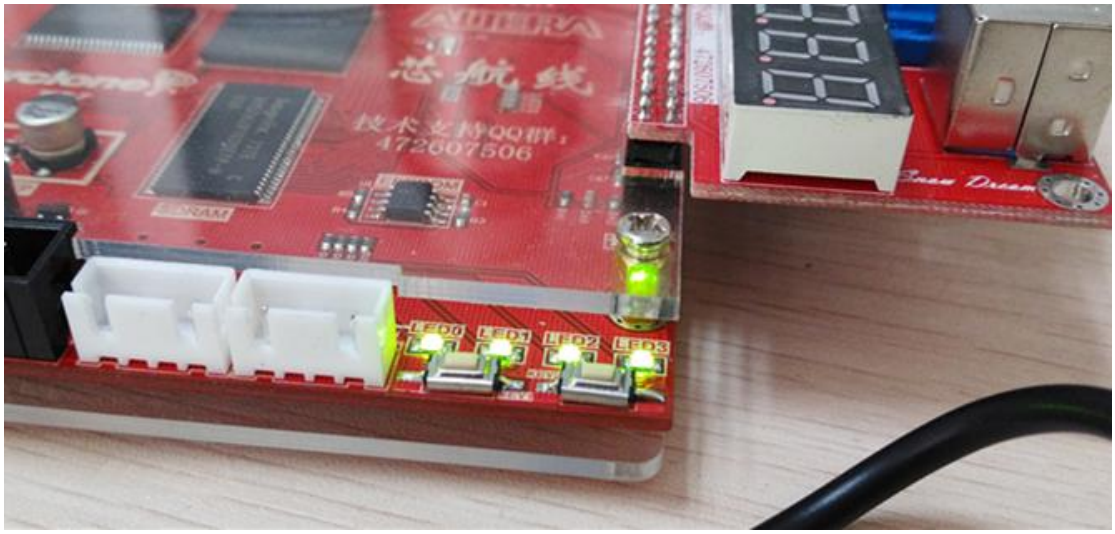
```
alt_u32 my_alarm_callback(void* context) {
    if (led_data == 0x0f) {
        led_data = 0x00;
    } else {
        led_data = 0x0f;
    }
    LED_WriteData(hLED, led_data); //一次性控制所有 LED 的亮灭
    return alt_ticks_per_second();
}

int main() {
    //初始化 LED
    hLED = LED_Init(PIO_LED_BASE);

    if (!hLED) {
        //    printf("Failed to init LED\n");
    }

    //熄灭所有 LED
    LED_Off(hLED, LED0 | LED1 | LED2 | LED3);

    printf("test alarm...\n");      //打印提示信息
    printf("alt_ticks_per_second() is %ld",
alt_ticks_per_second());

    //启动系统时钟服务
    if (alt_alarm_start(&alarm, alt_ticks_per_second()
            , my_alarm_callback, NULL) < 0) {
        printf("No system clock available\n");
    }
    while (1)
        ;  //等待时钟事件发生
    return 0;
}
```

这样我们的软件就编写完成了，接着我们点击键盘的组合键“CTRL + B”（或者依次点击【Project】【Build All】）来编译整个工程。

接着我们在 Quartus II 中打开配置下载窗口将 sof 文件下载到 FPGA 中，然后在 Eclipse 中点击【Run】->【Run Configuration】，在弹出的界面中，双击左侧的【Nios II Hardware】新建一个下载设置，将 name 更改为 sys_clk_test，与工程名保持一致。右侧 Project Name 选择 sys_clk_test，如下图所示：



运行后在芯航线 FPGA 开发板上可以看到，4 个 LED 每隔 1 秒闪烁一次。



补充阅读：

不常用操作系统的工程师很少碰到 alarm 这个概念，在他们心目中 alarm 实质上就是一个简单的定时器周期中断事件。在操作系统中，alarm 却是一个常用的概念，比如，当一个进程需要等待某个事件发生又不想永远等待下去时.该进程会设置一个超时(timeout)值。当这个超时值到达时，操作系统会向进程发送一个 alarm(警告信号)，提醒进程不要再等待。在 Nios II 中，alarm 也可以同样理解为是打断正在执行的流程，提醒系统不要等待的信号。

如何初始化系统时钟服务？按照一般思路，先初始化定时器，再编写定时器中断服务程序。但是，在 HAL 层上进行应用程序开发不必如此，因为与硬件细节相关的初始化过程都由系统替用户完成了。用户在使用某项服务前，只需要开启该项服务即可。

使用 HAL 提供的系统时钟服务大致分为 3 个步骤：

1) 调用 alt_alarm_start()开启系统时钟服务，该函数位于 BSP 工程下的 HAL\src 文件夹下的 alt_alarm_start. c 中。
2) 按照格式要求编写回调函数。该回调函数将实现用户的功能。
3) 调用 alt_alarm_stop()关闭系统时钟服务。该函数位于 BSP 工程下的 HAL\src 文件夹下的 alt_tick. c 中。

通过接口函数访问定时器对象中的变量，为了达到隐藏信息的目的，Altera 公司提 供了一个接口函数 alt_ticks_per_second()供用户获得一个设定 alarm 服务周期为 1 s 的变量值。

系统时钟服务 API 函数分析：

① 开启系统时钟服务函数：alt_alarm__start()

函数原型：

```
int alt_alarm_start (
                alt_alarm * alarm,
                alt_u32 nticks,
                alt_u32 (* callback) (void* context),
                void* context)
```

函数功能：启动系统时钟服务。

输入参数：

alarm——一个指向 alt_alarm 结构体类型的指针变量，用户需要为每个系统时钟服务创建一个 alt_alarm 类型变量。系统由该变量对系统时钟服务进行维护。例如，用户需要两个系统时钟服务，则需要定义 alt_alarm alarm_1 和 alt_alarm alarm_2。

nticks——指示每隔 nticks 执行一次回调函数。如果定时器预设的中断周期为 1 ms,则每个 tick 的时间间隔为 1 ms。如果用户希望获得 1 s一次的服务，则 nticks 的输入值为 1 s/1 ms=1000。

Callback——用户回调函数指针。

Context—-传给回调函数的参数，一般为寄存器的地址，也可以是其它参数。如果不需要传递参数则此处填入 NULL。

② 用户回调函数：my_alarm_callback()

函数原型：

```
alt_u32 my_alarm_callback (void * context)
```

输入参数：

context —— alt_alarm_start()函数传入的参数，函数名可以自己定义，但是函数原型一定要按上述格式书写。

注意：不要在回调函数中实现复杂的功能，因为回调函数实际是定时器中断服务函数的一部分。

③停止系统时钟服务函数 alt_alarm_stop()

函数原型：

```
void alt_alarm_stop ( alt_alarm * alarm )
```

输入参数：

alarm—— 一个指向 ah_alarm 结构体类型的指针变量，用户希望停止哪一个 系统时钟服务，即把该时钟服务对应的 alarm 变童传给 alt_alarm_stop()。

至此，通过 HAL API 函数使用系统时钟服务的方法已总结完毕，其它 API 函数本文不再叙述，读者可参阅 Altera 公司提供的 API 函数手册。

最后留一个问题供读者思考，如何开启 3 个系统时钟服务，使得 3 个 LED 分别以 2 Hz、1 Hz 和 0.5 Hz 的频率闪烁？

NIOS II 开发注意点总结

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 6 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。我一早就想出一点 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。

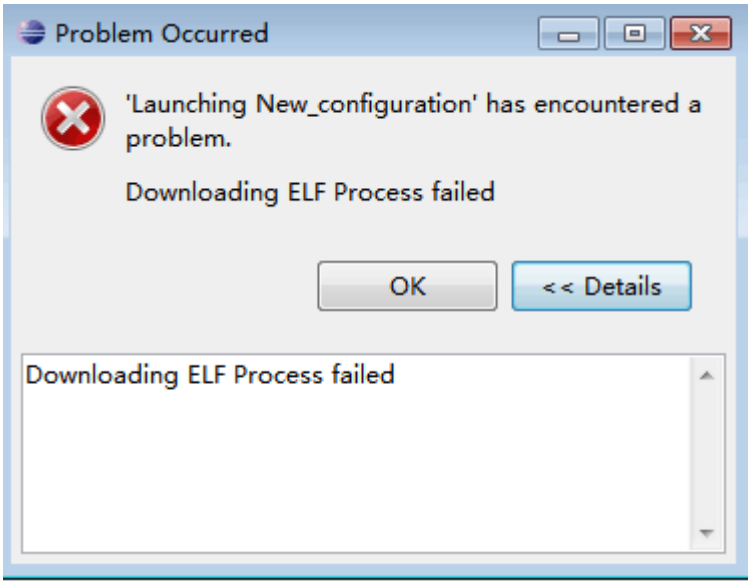
当下定决心做这件事的时候，我却犯了难。到底，我该以一种怎样的方式来开启我的系列文档呢？是从 CPU 架构、指令集开讲，还是直接从 LED 点灯写起呢？每天在技术交流群里，看到大家学习时候遇到各种问题并却不知道如何解决时，我突然觉得，其实，大家暂时不缺入门的教程，缺的，是继续学习下去的信心。那么信心从何而来？就从解决 NIOS II 开发中常见的各种问题着手开始吧。如果大家首先就有一份“捉虫子”手册放在旁边，遇到各种问题马上能从手册中找到解决方法或解决思路，那么大家的学习信心必然会与日俱增。所以，我选择我的这一份文档，从教大家捉虫子开始。

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。

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：

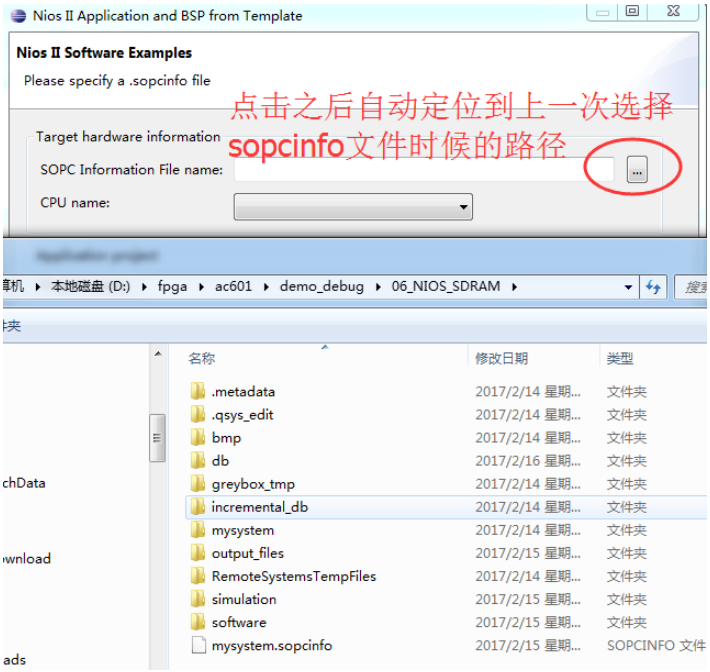
个人目前总结的，出现该问题的原因主要有以下几点，

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。

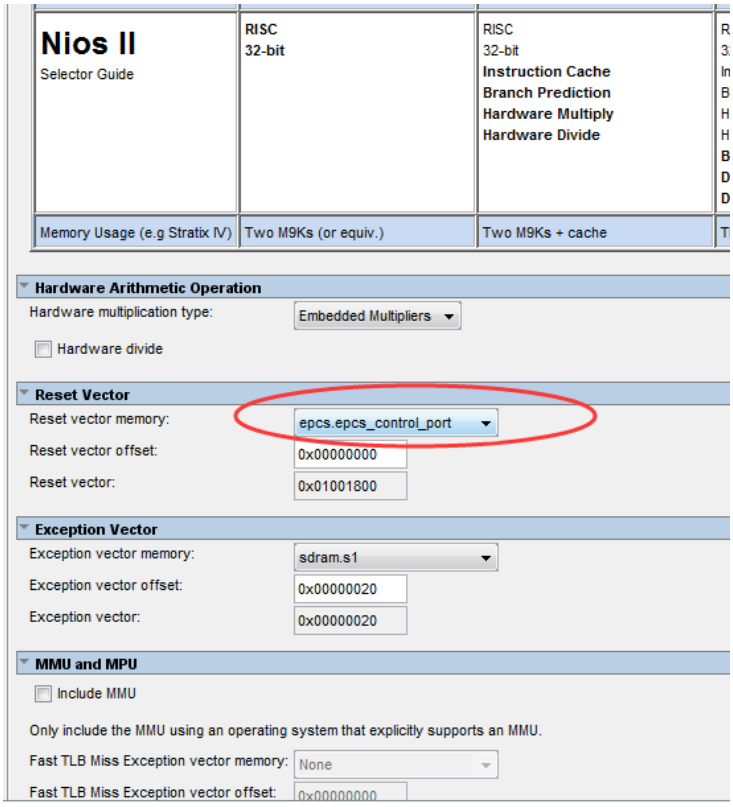
   解决方法：新建工程选择 sopcinfo 文件时务必选择正确的 sopcinfo 文件。



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。

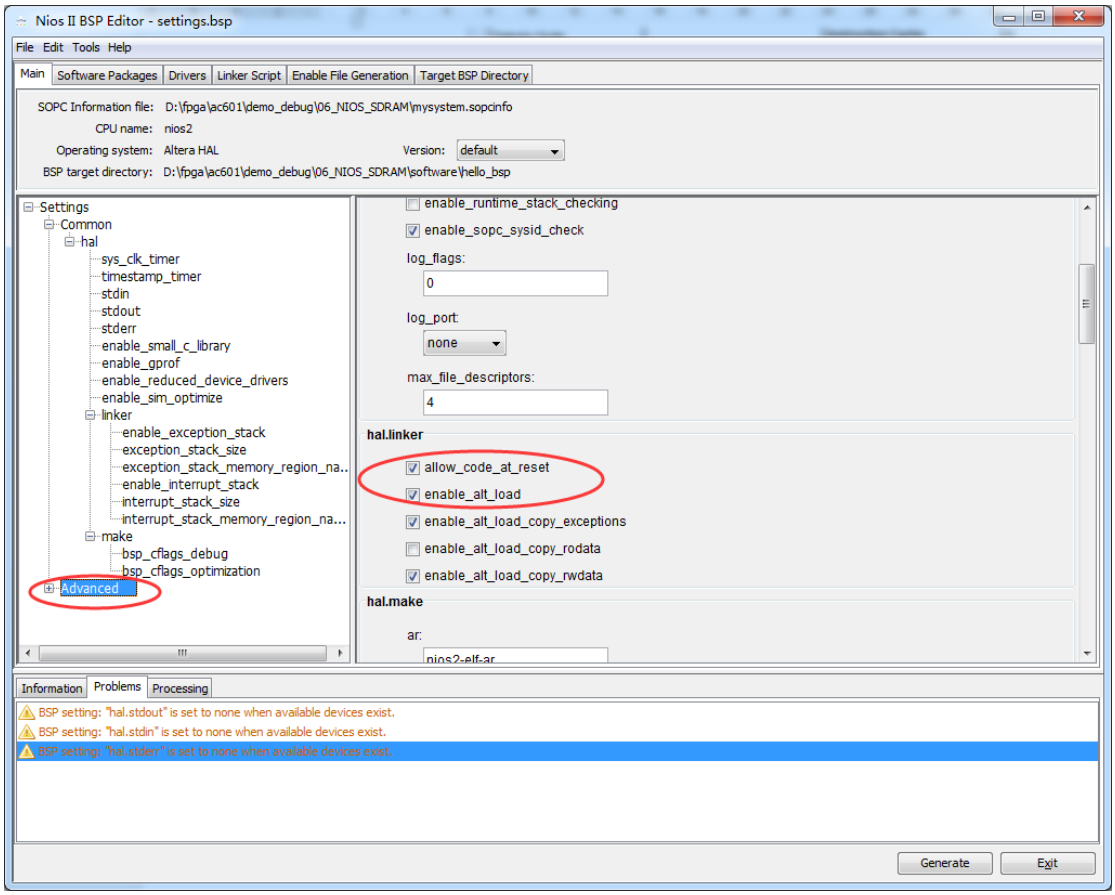
好了，饶了这么多口舌，该说大家最关心的问题了，怎么设置 CPU 的复位地址呢？其实有两个地方需要设置，而 90%以上的人只知道一个地方，那就是 QSYS 中选择 CPU 的复位地址。当我们的系统中 FLASH 使用 EPCS 芯片剩余容量，那么如果我们要定义 CPU 硬件上从 FLASH 中启动，就需要定义 CPU 的 Reset Vector 为 EPCS，如果我们要定义 CPU 硬件上从 RAM（onchip_ram/SDRAM/DDR2）中启动，那么就选择 Reset Vector 为 ram。当然，这没什么问题，但是我们任然可以在一开始就定义 CPU 的 Reset Vector 为 EPCS，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的 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。

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。

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、 NIOS 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 NIOS II 版本中能够很好的奏效，例如 NIOS II 的 e 版本和 s 版本，但是在 f 版本中，为了增强 NIOS 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，NIOS 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）中将这些外设初始化，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，对于串口接受，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：

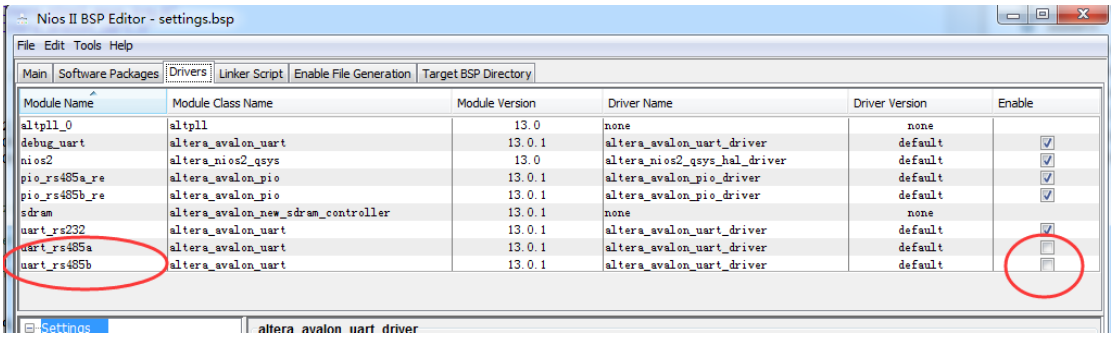
```
void alt_sys_init( void )
{
    ALTERA_AVALON_UART_INIT ( DEBUG_UART, debug_uart);
    ALTERA_AVALON_UART_INIT ( UART_RS232, uart_rs232);
//    ALTERA_AVALON_UART_INIT ( UART_RS485A, uart_rs485a);
//    ALTERA_AVALON_UART_INIT ( UART_RS485B, uart_rs485b);
}
```

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：

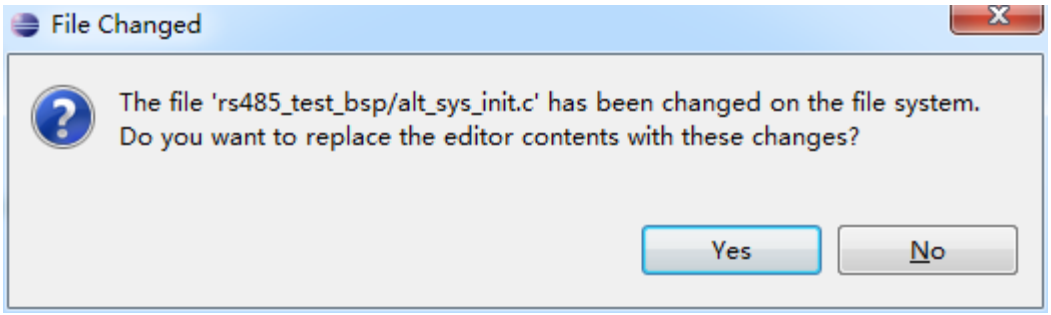
```
//等待发送寄存器可用
while (!(ALTERA_AVALON_UART_STATUS_TRDY_MSK
          & IORD_ALTERA_AVALON_UART_STATUS(UART_RS485A_BASE)))
       ;
//写入待发送的数据
    IOWR_ALTERA_AVALON_UART_TXDATA(UART_RS485A_BASE, 0x88);
```

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。该种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在了：

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);
}
```

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存在问题的，我们可以直接编译更改下载，而在 NIOS II EDS 中的工程却没这么简单，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 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。

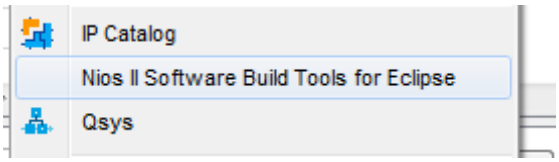
或者，如果你的新工程是从老工程复制过来的，那么一切表面看起来不会出现任何问题，所有的软件工程仍然可以“正常打开”，我们依旧可以编译，重新生产 bsp 文件，下载。然而，这种方式存在更大的风险，因为这样，你修改的还是原来路径下的文件，因此，这样就有极大的风险使得你在希望更改新的软件工程的时候，把原本的工程给改了。很多朋友表示 NIOS

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。

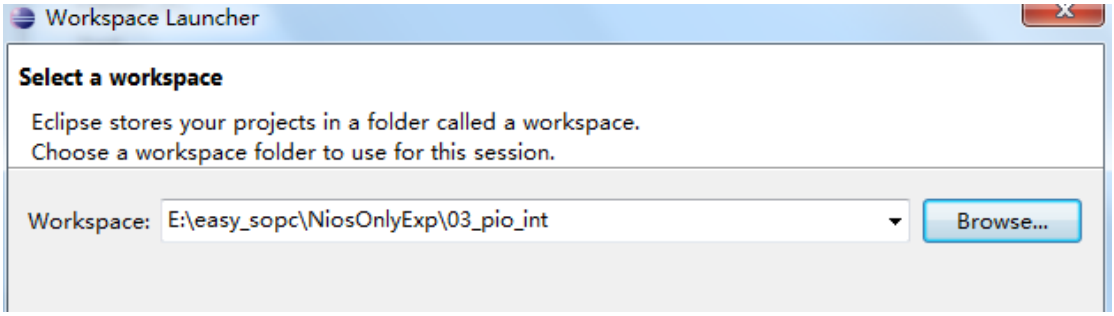
6.3 解决方案详解

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（workplace），移除旧版工程，修改 bsp 文件，重新导入（import）工程。

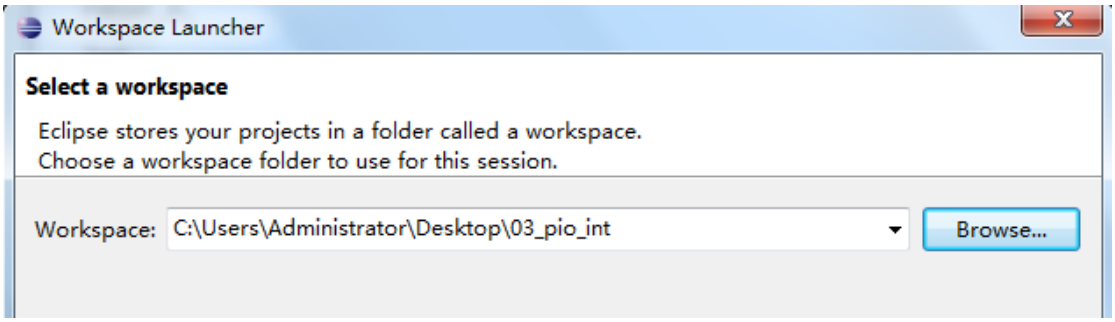
这里，我将我电脑中 E:\easy_sopc\NiosOnlyExp\03_pio_int 这个文件夹拷贝到桌面（C:\Users\Administrator\Desktop\03_pio_int）上，以符合更改路径这一前提，然后，打开其中的 Quartus II 工程“CoreCourse_GHRD.qpf”，工程打开后，选择 tools->Nios II Software Build Tools For Eclipse。



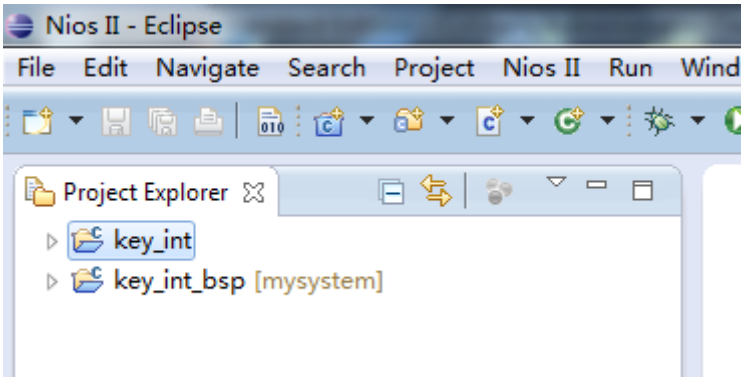
然后在弹出的工作空间选择对话框中，可以看到，工作空间还是上次的工作空间路径，



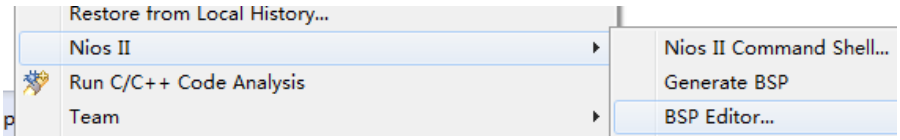
这里，我们将工作空间切换到 C:\Users\Administrator\Desktop\03_pio_int，如下图所示：



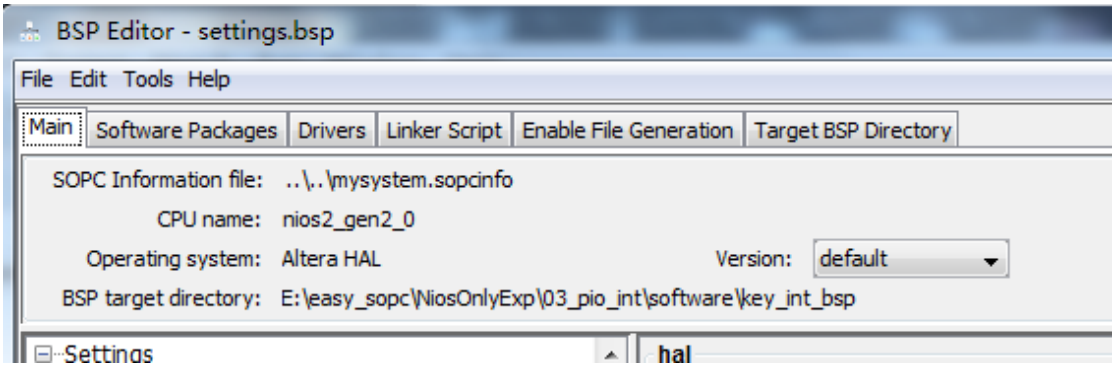
然后选择 OK 就会打开工程，打开后我们可以看到，软件自动加载了一个软件工程和一个 bsp 工程，该工程名字与复制前的工程（E:\easy_sopc\NiosOnlyExp\03_pio_int\software）中的软件工程名字一致（注意，如果用户磁盘上旧位置不存在该工程，例如该工程是从其他电脑拷贝过来的，工程将显示蓝色，无法打开，这个虽然会影响我们后续一步一步分析问题原因，但是不影响我们解决问题，如果各位自己电脑上无法打开，可以找一个原本存在的工程测试，或者跳过分析问题步骤，直接看解决方案总结）。



这时候，我们鼠标右键选中 key_int_bsp，选择 NIOS II -> BSP Editor

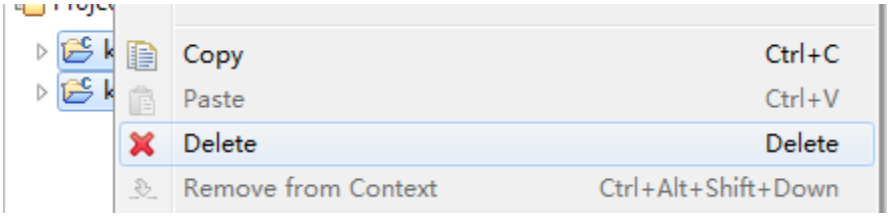


在弹出的 BSP 设置界面中我们可以看到，BSP target direction 还是 E:\easy_sopc\NiosOnlyExp\03_pio_int\software\key_int_bsp，即复制前的路径。



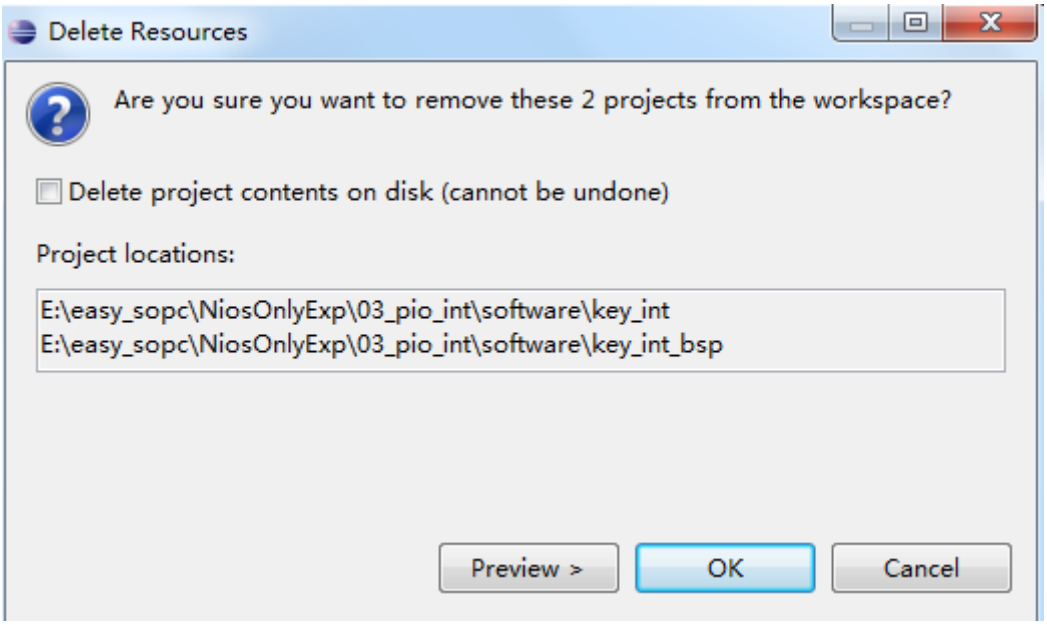
因此可知，如果此时我们在当前工程的 Qsys 中更改了 NIOS II 系统的架构或者增删了东西，重新生产 Qsys 文件，然后我们回到 Eclipse 中选择 generate BSP 时，软件会根据新的 Qsys 信息重新生产 bsp 文件，而这个 bsp 文件还是保存在之前的 E:\easy_sopc\NiosOnlyExp\03_pio_int\software\key_int_bsp 中，因此就导致原本的复制之前的工程内容被更改，即本来我们是想把整个工程复制到另一个地方进行单独修改的，然而却实质上把原路径下的工程文件给更改了，到最后导致原版和复制后的工程都被改变。

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：

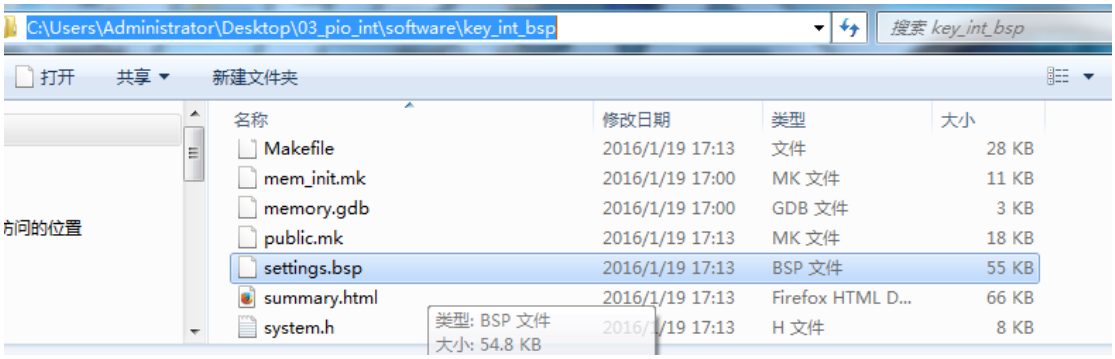


在弹出的对话框中，点击 OK。不过大家一定要注意的是，千万不要勾选上面的那个“Delete project contents on disk（cannot be undone）”，因为这个是删除软件工程的源文件，如果选择了这个，那么在删除时就会将原本没有复制之前的路径下的软件源工程给删除掉，

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 II13.0 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

```
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>1488367120721</BspGeneratedUnixTimeStamp>
7          <BspGeneratedLocation>E:\easy_sopc\NiosOnlyExp\03_pio_int\software\key_int_bsp</BspGeneratedLocation>
8          <BspSettingsFile>settings.bsp</BspSettingsFile>
9          <SopcDesignFile>E:\easy_sopc\NiosOnlyExp\03_pio_int\mysystem.sopcinfo</SopcDesignFile>
10         <JdiFile>default</JdiFile>
11         <Cpu>nios</Cpu>
```

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 II13.0 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>1488367120721</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>
```

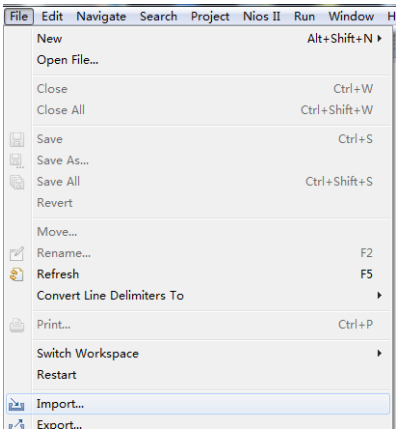
而对于 15.1 版本的工程，打开文件后我们可以看到，该文件下的第 7 行左右，BspGeneratedLocation 指定的还是复制前的路径，第 9 行已经改进为采用相对路径了：

```
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_sopc\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>
```

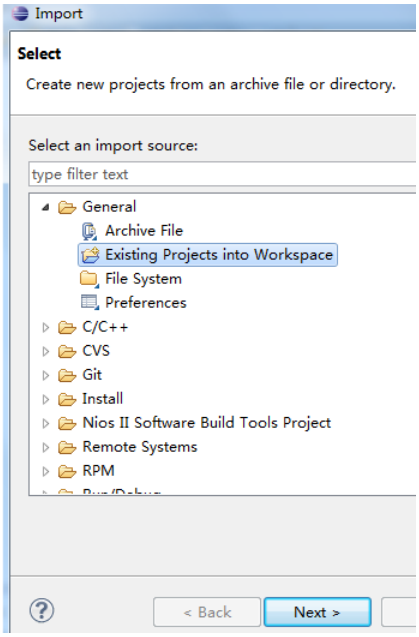
于是，这里只需将第 7 行这个地方更改为我们复制后的新路径 C:\Users\Administrator\Desktop\03_pio_int\software\key_int_bsp，第 9 行不变即可更改后的文件内容如下所示：

```
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>
```

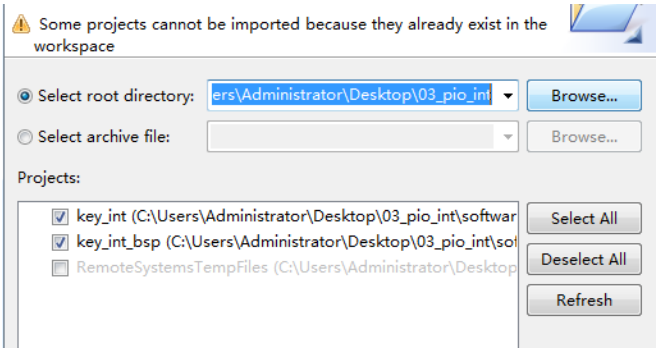
然后保存文件，回到 NIOS II Eclipse 中，选择 File -> Import



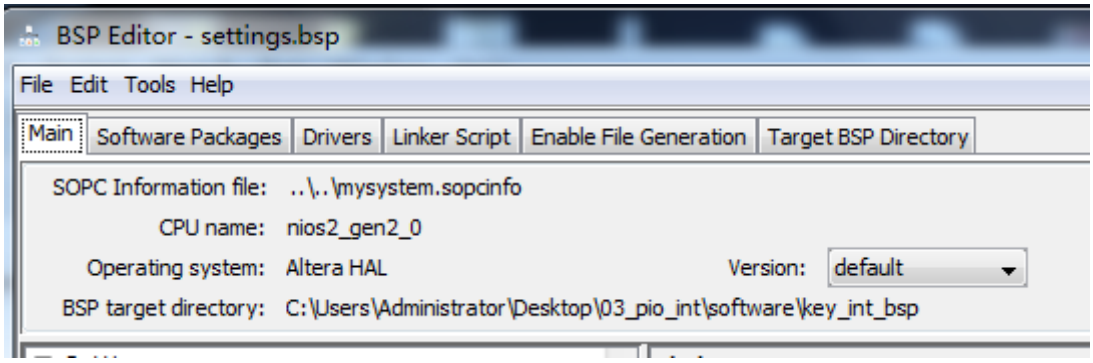
在弹出的窗口中，选择 General 下的 Existing Projects into Workspace:



在弹出的窗口中，选择 Select root directiory，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了，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

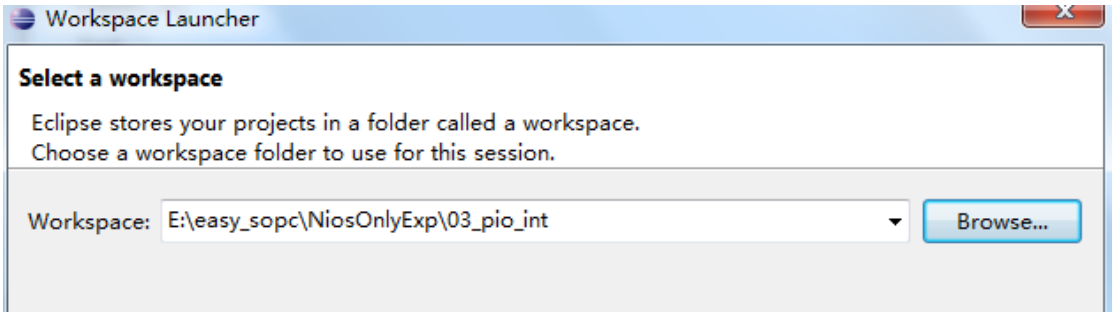
容，编译下载也没问题。因此该问题得以完美解决。

6.4 解决方案步骤总结

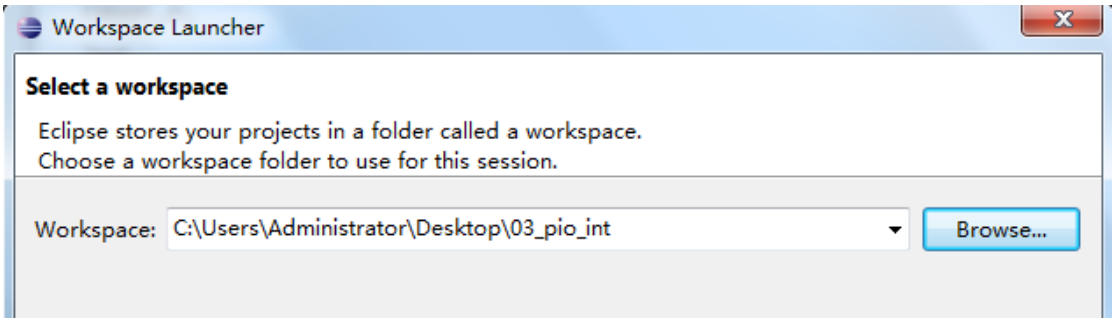
好了，说了这么多，各种推理和验证，导致真正有用的操作步骤被弱化了，不容易区分，这里，我再总结下：

6.4.1 切换工作空间（workspace）

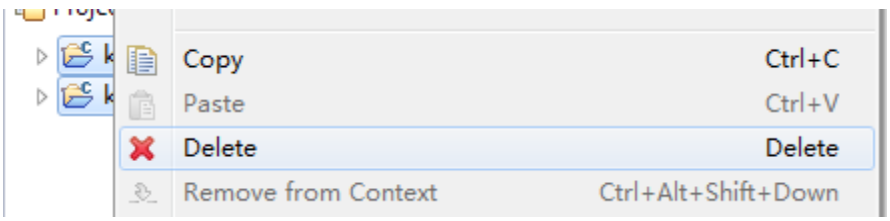
切换工作空间前：

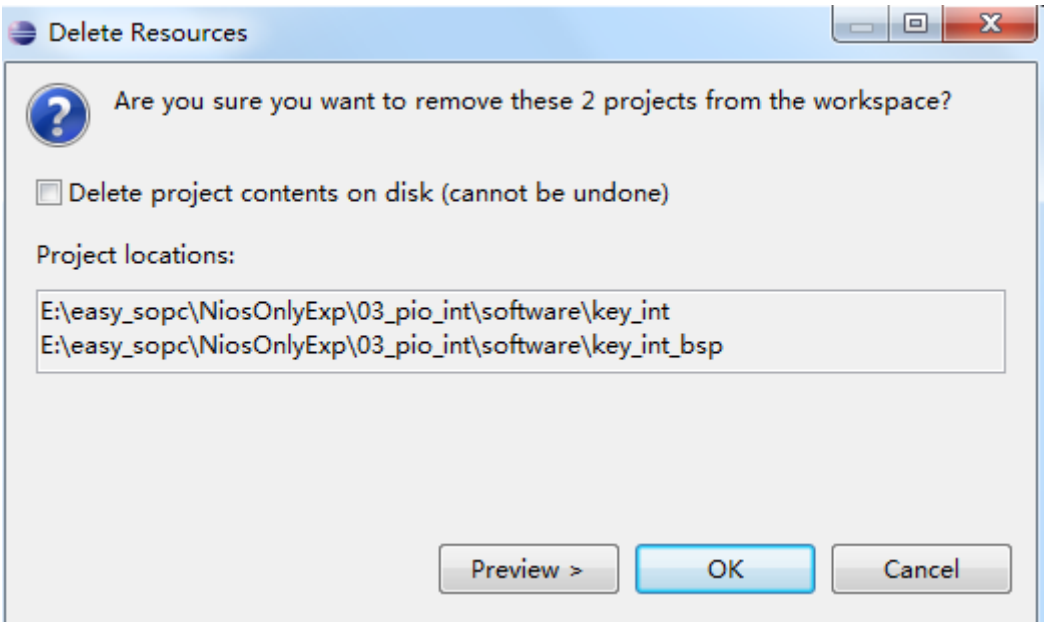


切换工作空间后：



6.4.2 移除旧版工程

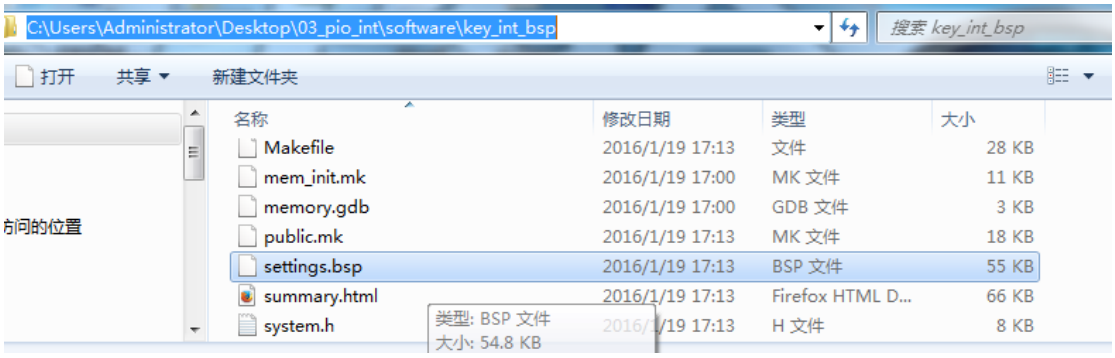




注意：千万不要勾选上面的那个“Delete project contents on disk（cannot be undone）”，

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：

```
<?xml version="1.0" encoding="UTF-8"?>
<sch:Settings xmlns:sch="http://www.altera.com/embeddedsw/bsp/schema">
        <BspType>hal</BspType>
        <BspVersion>default</BspVersion>
        <BspGeneratedTimeStamp>2017-3-1 19:18:40</BspGeneratedTimeStamp>
        <BspGeneratedUnixTimeStamp>1488367120721</BspGeneratedUnixTimeStamp>
        <BspGeneratedLocation>E:\easy_sopc\NiosOnlyExp\03_pio_int\software\key_int_bsp</BspGeneratedLocation>
        <BspSettingsFile>settings.bsp</BspSettingsFile>
        <SopcDesignFile>E:\easy_sopc\NiosOnlyExp\03_pio_int\mysystem.sopcinfo</SopcDesignFile>
        <JdiFile>default</JdiFile>
        <Cpu>nios</Cpu>
```

更改后

```
<?xml version="1.0" encoding="UTF-8"?>
<sch:Settings xmlns:sch="http://www.altera.com/embeddedsw/bsp/schema">
        <BspType>hal</BspType>
        <BspVersion>default</BspVersion>
        <BspGeneratedTimeStamp>2017-3-1 19:18:40</BspGeneratedTimeStamp>
        <BspGeneratedUnixTimeStamp>1488367120721</BspGeneratedUnixTimeStamp>
        <BspGeneratedLocation>C:\Users\Administrator\Desktop\03_pio_int\software\key_int_bsp</BspGeneratedLocation>
        <BspSettingsFile>settings.bsp</BspSettingsFile>
        <SopcDesignFile>C:\Users\Administrator\Desktop\03_pio_int\mysystem.sopcinfo</SopcDesignFile>
        <JdiFile>default</JdiFile>
        <Cpu>nios</Cpu>
        <SchemaVersion>1.9</SchemaVersion>
        <Setting>
```

15.1 及以上版本

更改前

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

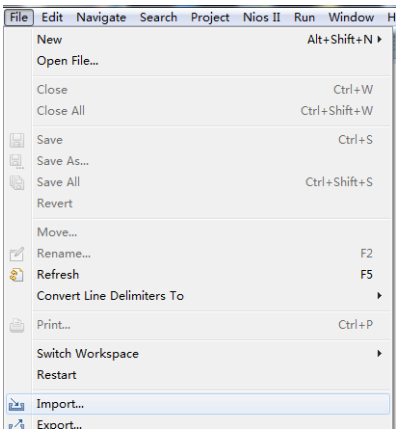
更改后

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

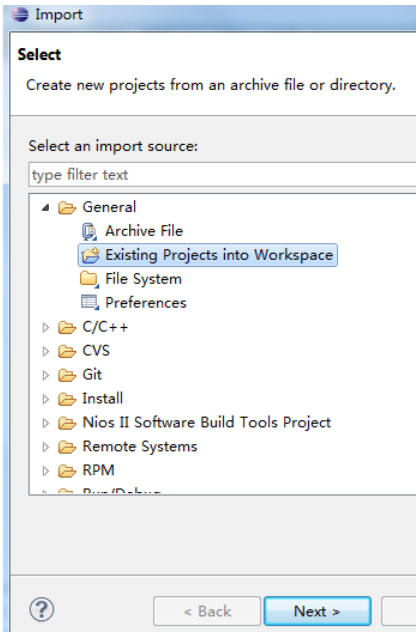
然后保存文件。

6.4.4　重新导入（import）工程

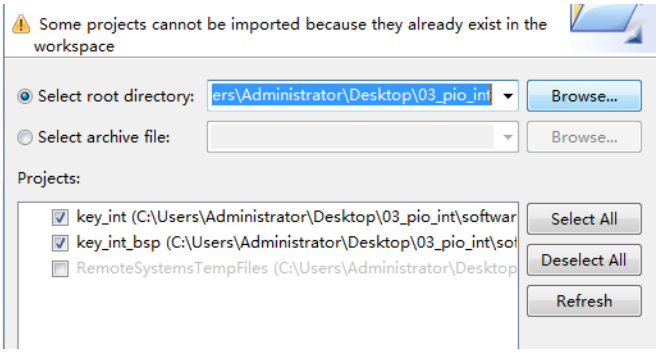
NIOS II Eclipse 中，选择 File -> Import



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 directiory，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。

小梅哥
芯航线电子工作室
2016 年 1 月 21 日星期四初创
2017 年 3 月 4 日星期六日更新

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，为开发带了了很大的便利。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。

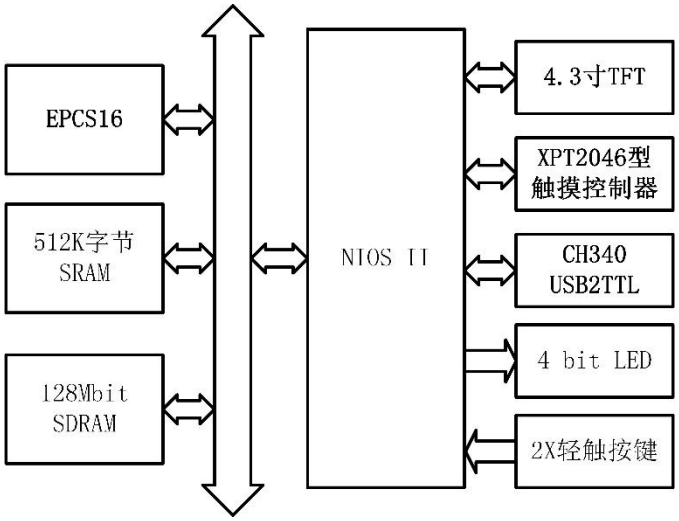
小梅哥在最近的工作中也遇到了这样的问题。我们新设计的一批开发板，在工厂生产完毕后，都要进行出厂测试。然而 SMT 厂家却并不熟悉我们的这种 Quartus Programmer+Flash Programmer 烧写方式。再说了，要使用这种方式还得安装 Quartus Programmer 和 NIOS II EDS 软件。厂家表示使用这种方式对他们来说有一定难度，而且效率也不高。所以我就根据 Altera 官方网站上的一个帖子，进行了转换，将 SOF 文件和 NIOS II 的 elf 固件合并并生成了一个 jic 文件，这样，厂家就只需要使用 Quartus Programmer 来烧写这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

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

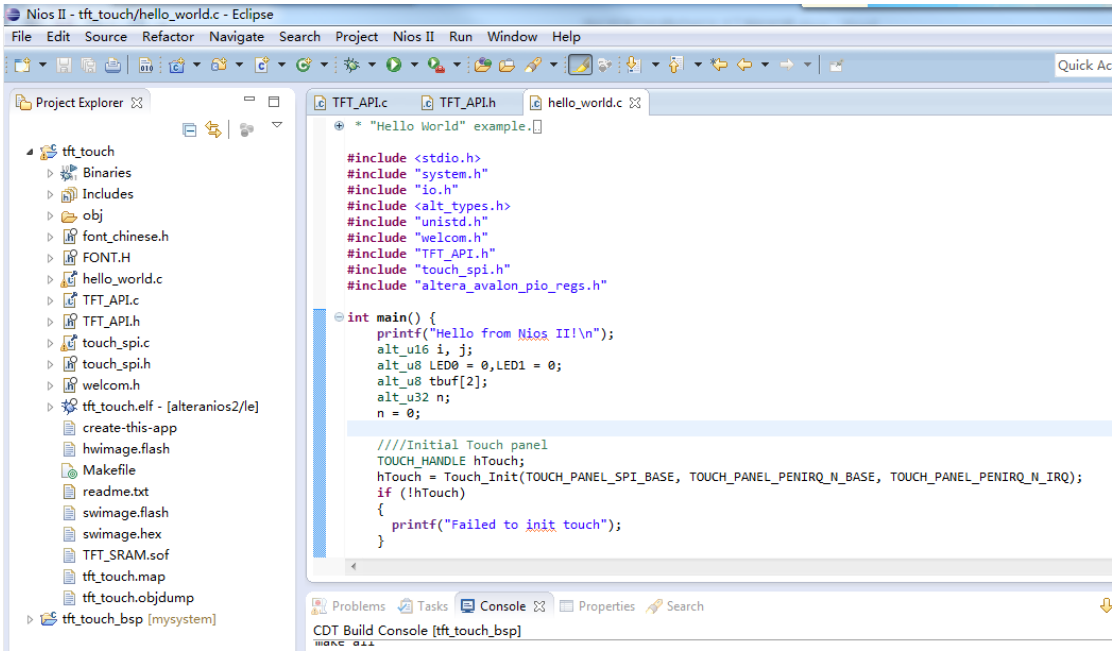
7.3.1　sof2flash：

从一个.sof 文件生成一个 flash 文件：

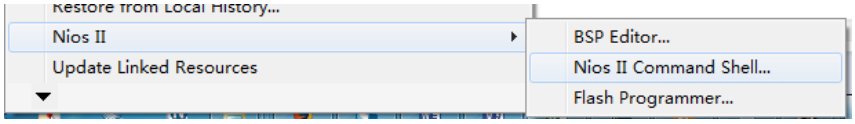
sof2flash --input=<hwimage>.sof --output=hwimage.flash --epcs –verbose

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

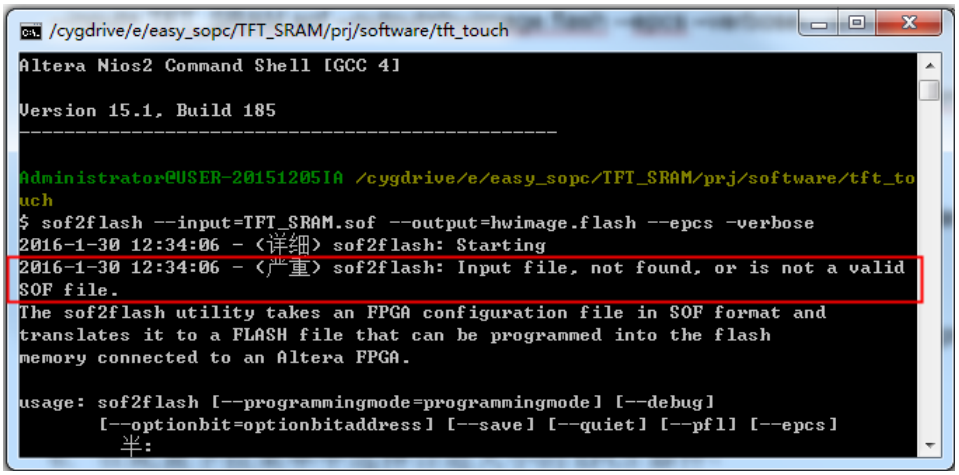


然后选中 tft_touch，单击右键选择 Nios II　->　Nios II Command Shell





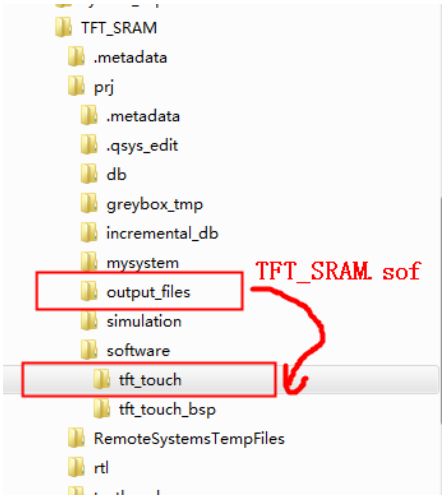
我们的 Quartus II 生产的 sof 文件名为”TFT_SRAM.sof”，这个时候，如果我们直接输入
sof2flash --input= TFT_SRAM.sof --output=hwimage.flash --epcs –verbose
会提示找不到 input file 也就是找不到 TFT_SRAM.sof 文件。



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_sopc\TFT_SRAM\prj\output_files 目录下，因此当然无法找到该文件了。解决这个问题的方法有两种。

第一种，推荐方案。

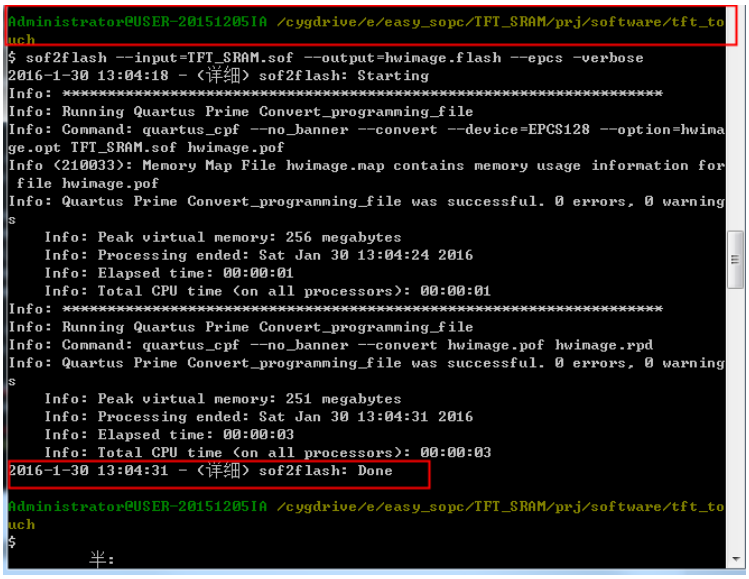
因为很多不熟悉命令行的朋友在操作时速度慢而且容易出错，因此这里提供一种比较熟悉的方式。首先在 windows 中，将 TFT_SRAM.sof 文件从 output_files 文件夹中拷贝到 tft_touch 文件夹中：



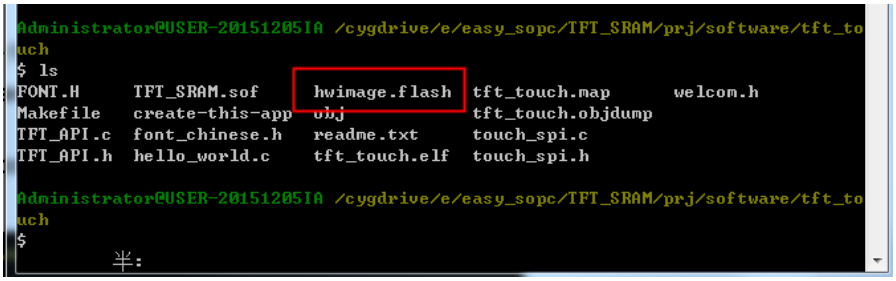
然后回到命令行窗口再次执行

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，连按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：

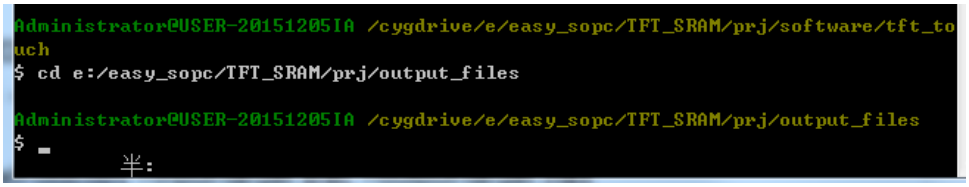


然后我们输入 ls 命令就能看到，确实生成了这样一个名为 hwimage.flash 的文件：

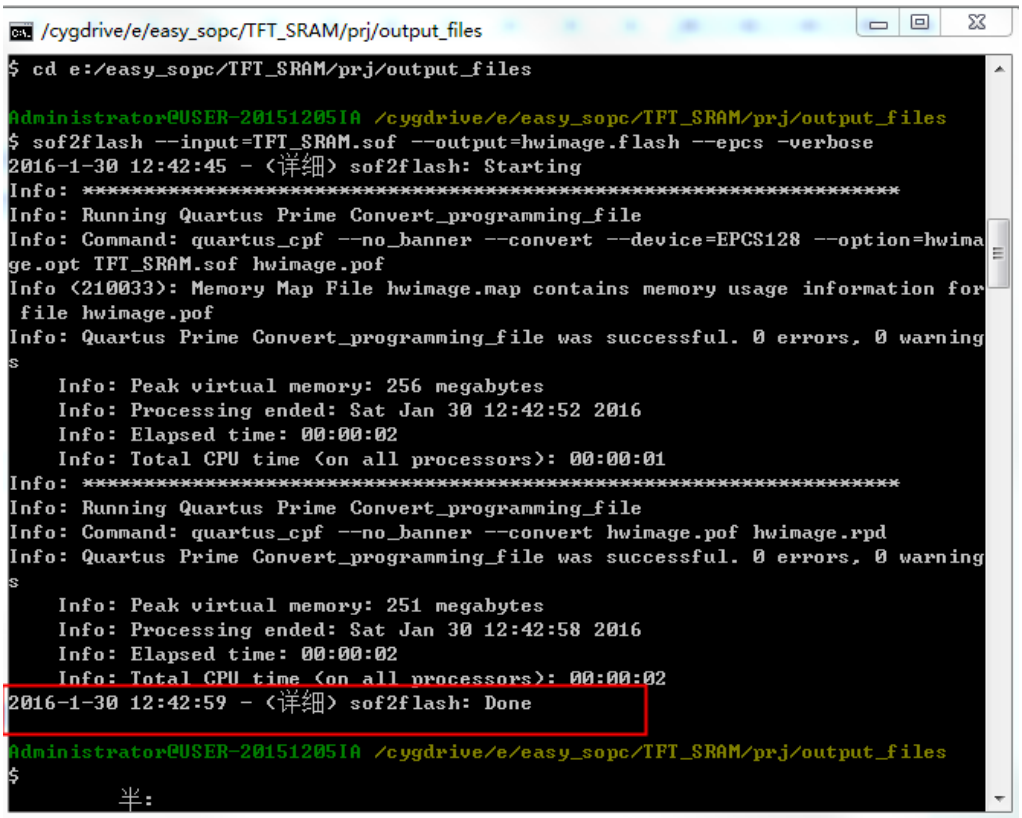


第二种方案：首先在 shell 中使用 cd 命令直接将目录切换到 sof 文件所在目录，也就是 E:\easy_sopc\TFT_SRAM\prj\output_files。相应命令为（注意斜线方向）：

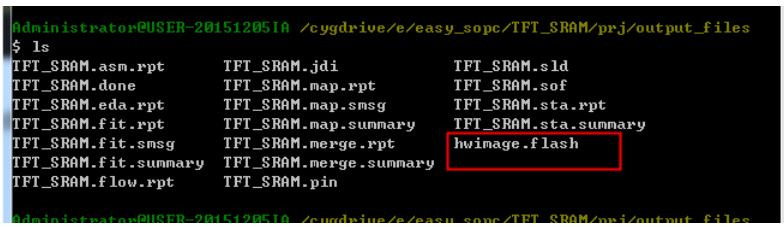
cd e:/easy_sopc/TFT_SRAM/prj/output_files



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：



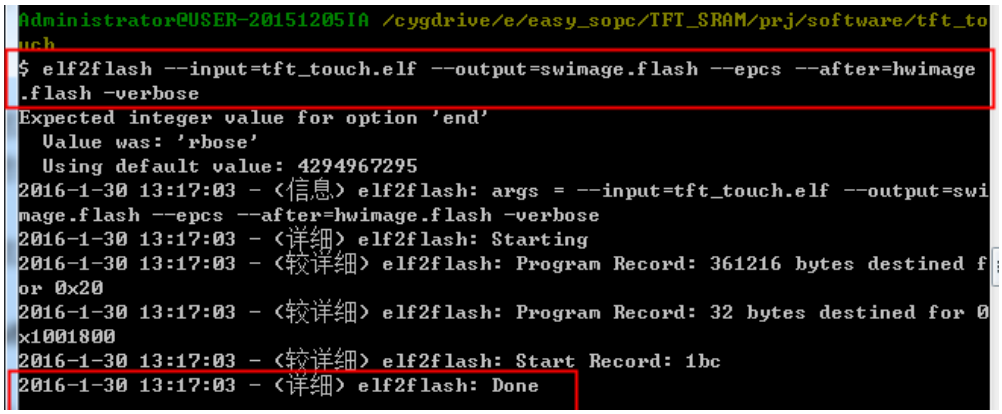
然后我们输入 ls 命令就能看到，确实生成了这样一个名为 hwimage.flash 的文件：



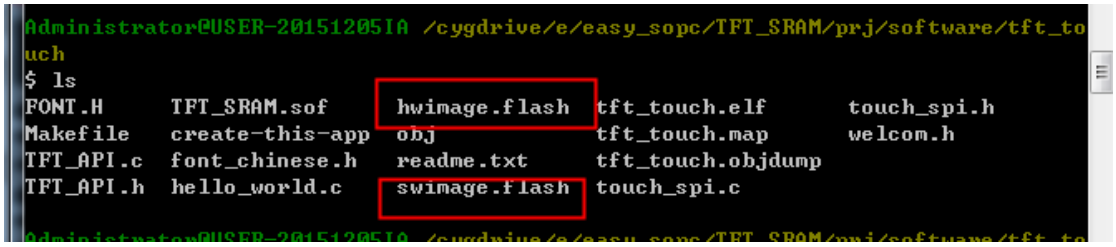
一般推荐大家使用第一种方式，当然命令行高手除外。

7.3.2　elf2flash：

从一个,elf 生成一个 flash 文件：
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 文件是作为了参数的一部分的，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：

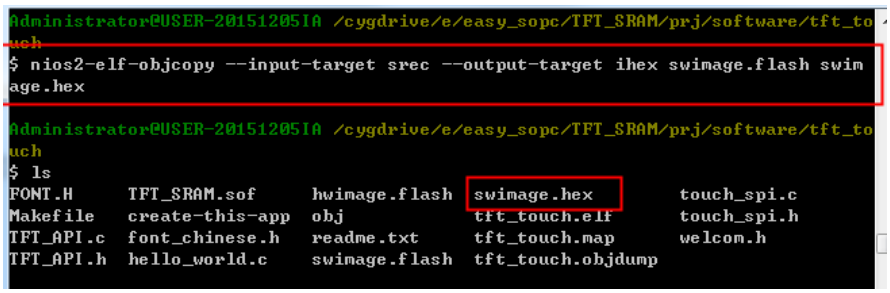


然后我们输入 ls 命令，可以看到，在当前文件夹下确实生成了一个名为 swimage.flash 的文件：



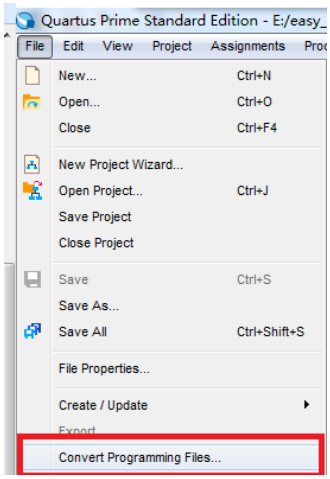
7.3.3　flash2hex:

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：



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

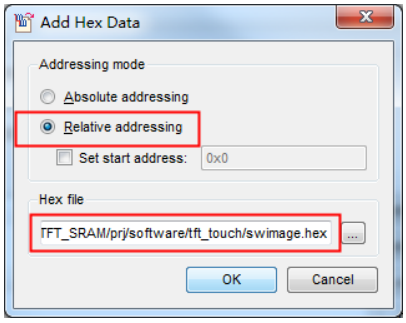
从列表中选择你的 FPGA 器件（见 10 步图）

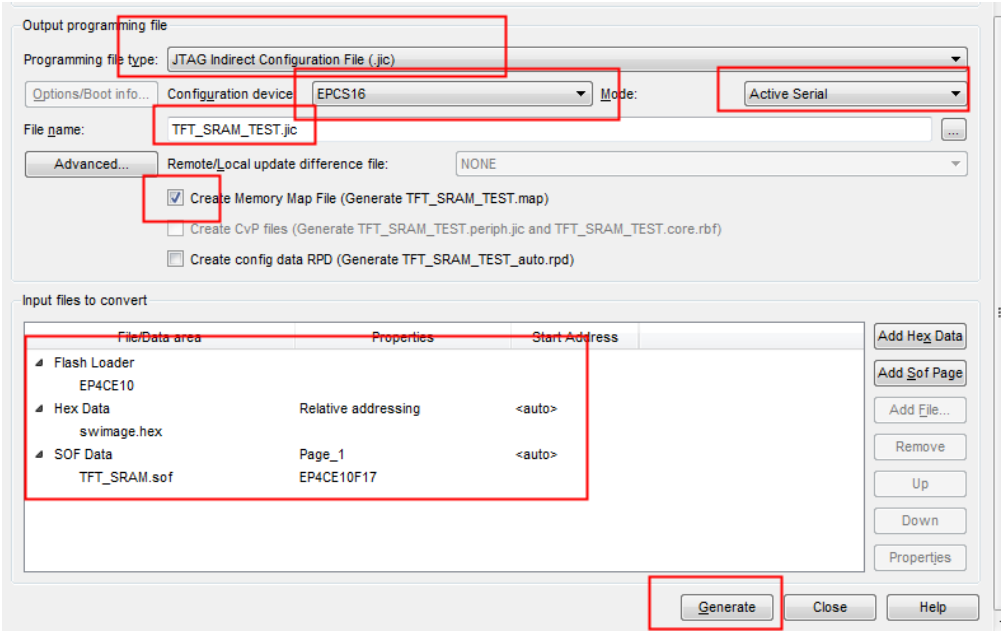
7.3.9　Add SOF

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

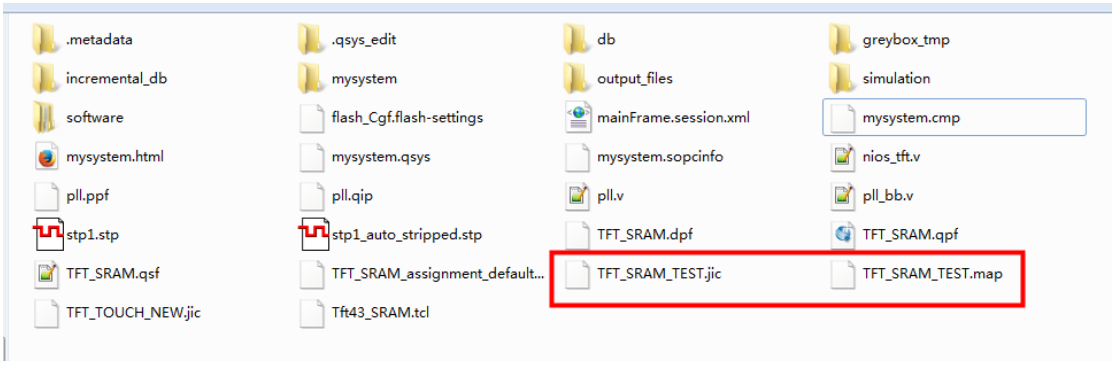




7.3.11　Generate

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（话说放这样一张图给客户，客户是不是会觉得特别开心呢）：





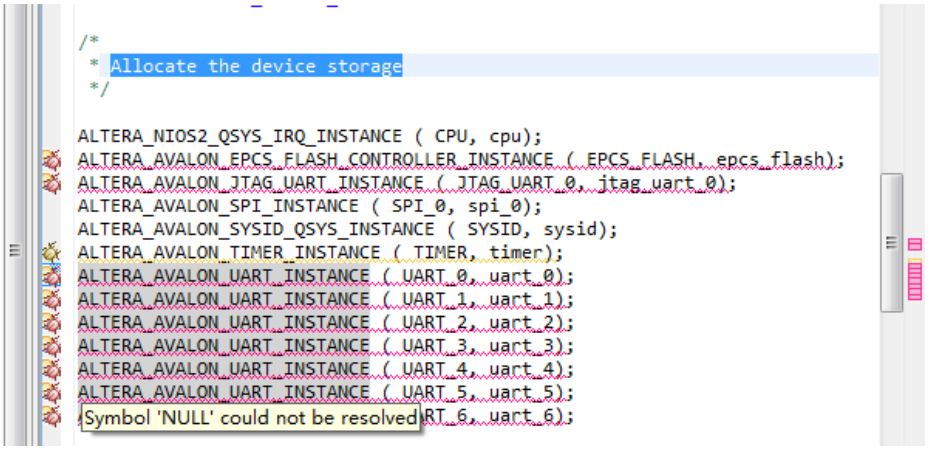
如有更多问题，欢迎加入芯航线 FPGA 技术支持群：472607506

小梅哥
芯航线电子工作室
2016 年 1 月 30 日星期六

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：



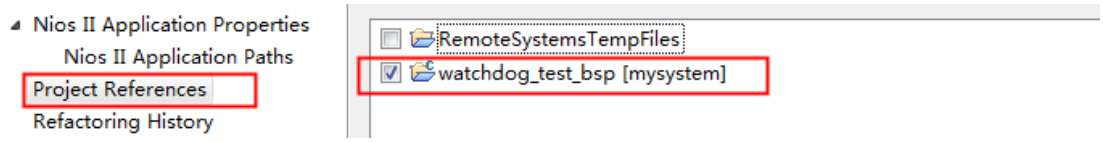
提示我 NULL 未定义。可是 NULL 明明是在 stddef.h 这样一个标准头文件中明确定义了的，怎么还会发生这种情况呢？

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“ATRL + 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。

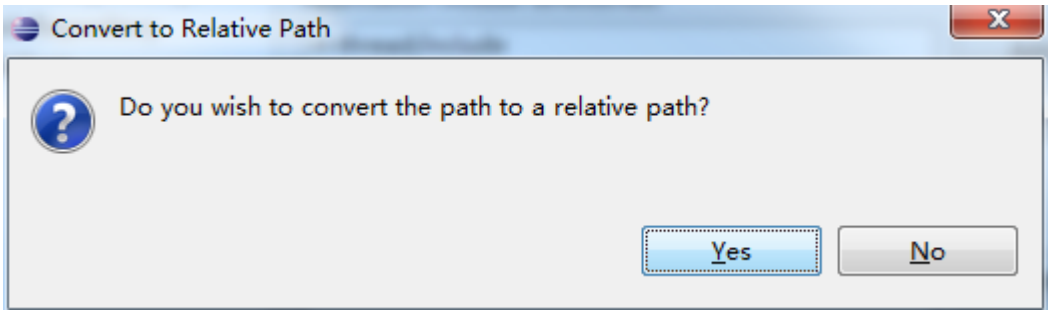
接着我们查看下 Project References 中是否勾选了 watchdog_test_bsp 工程，如果没有勾选的话，当工程关闭了重新打开时，工程有可能会报各种无法理解的错误（当一个 workspace 中有多个应用工程时）。这里我们需要确认这个选项被勾选上了。

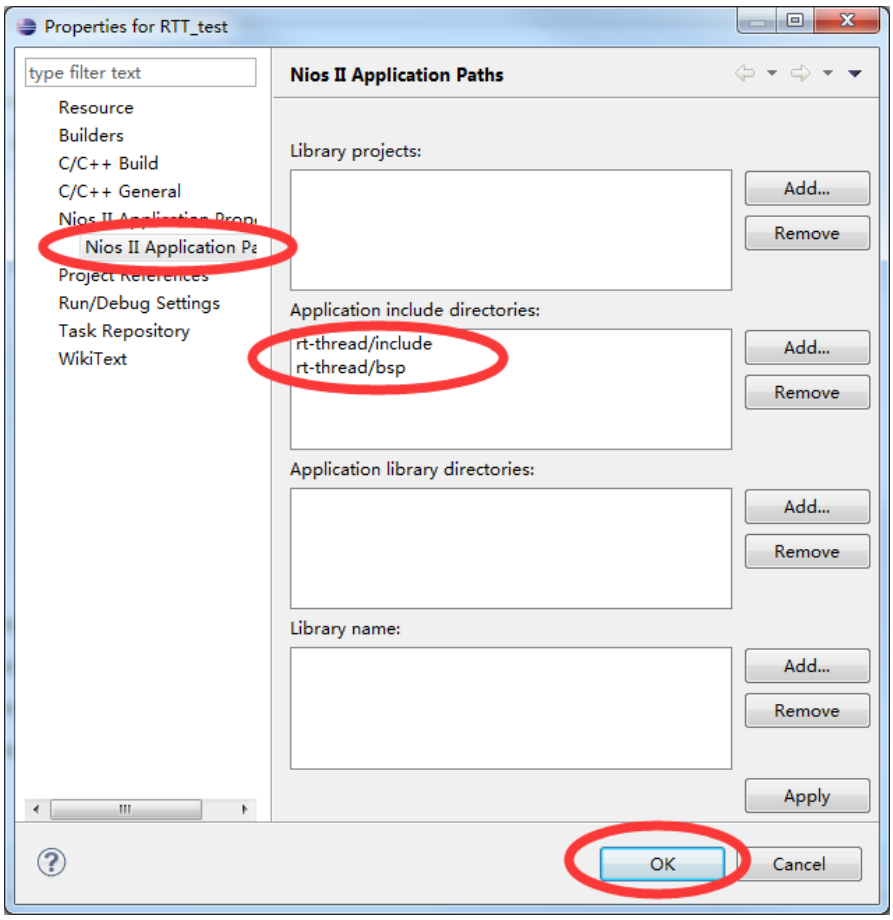


设置 NIOS II 软件工程头文件包含路径

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 文件夹道头文件路径中，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。

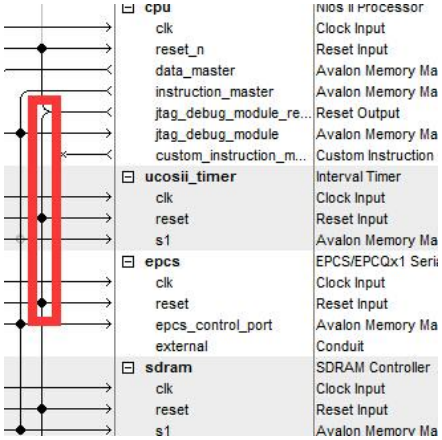




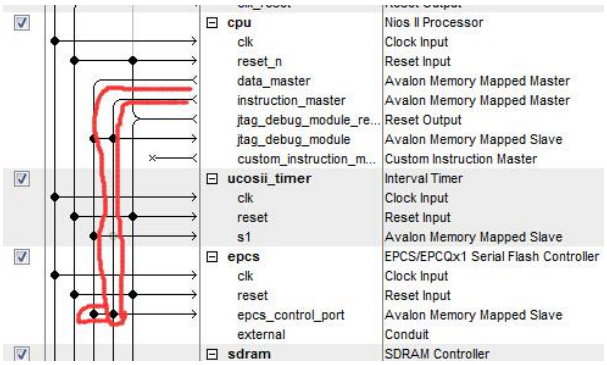
至此，所有的运行 RT-Thread 操作系统的要求都已经满足了。

EPCS 使用相关要点

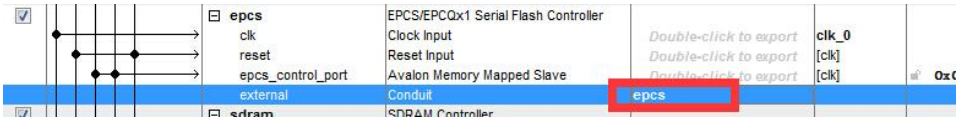
1、 在需要最终将程序固化到 EPCS 的系统中，需要添加 EPCS/EPCQx1 Serial Flash Control。该控制器的 reset 信号一定要与 jtag debug module reset 信号（cpu 模块中）相连，另外，最好其他所有模块的 reset 信号都与 jtag debug module reset 信号连接上，否则，在最终通过 Flash Programmer 固化程序时，会出错。如下图所示：



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：



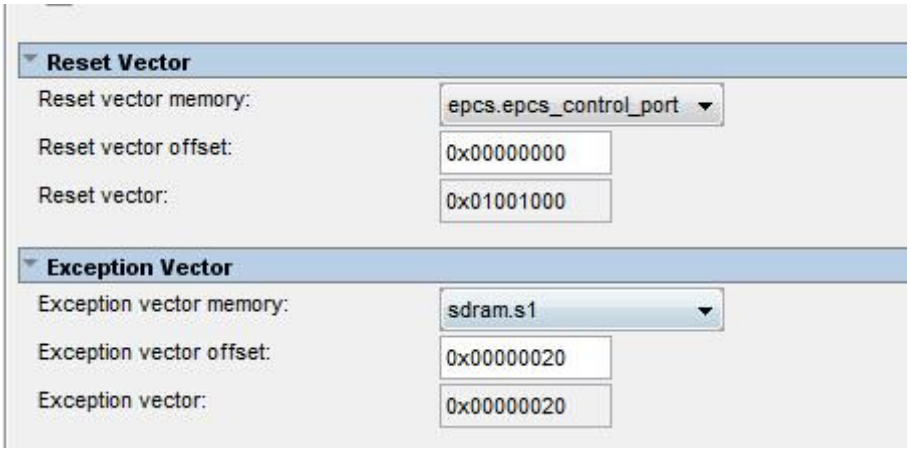
这里，导出到顶层后具体怎么分配引脚，在 Altera 的《Embedded Peripheral IP User Guide》中有相关介绍，

- 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 regularr I/O**
  - `DCLK` = **Use as regular I/O**
  - `FLASH_nCE/nCSO` = **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【Assigments】-> 【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/nCSO 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)
176311 Pin epcs_flash_sdo is assigned to pin location Pin_C1 (IOPAD_X0_Y22_N21)
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)
176311 Pin epcs_flash_sce is assigned to pin location Pin_D2 (IOPAD_X0_Y21_N14)
176311 Pin ~ALTERA_FLASH_nCE_nCSO~ 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)
176311 Pin epcs_flash_dclk is assigned to pin location Pin_H1 (IOPAD_X0_Y17_N14)
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)
176311 Pin epcs_flash_data0 is assigned to pin location Pin_H2 (IOPAD_X0_Y17_N21)
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。

拔掉下载器或者关闭 NIOS II EDS 软件，开发板上的系统即停止运行

问题原因

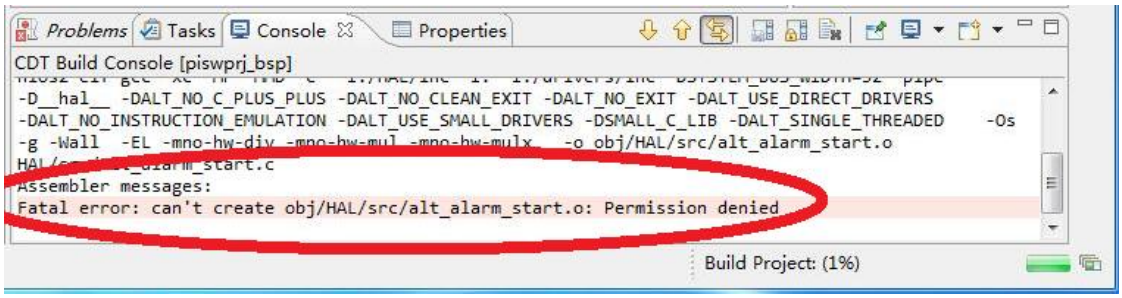
这是因为 NIOS II 系统中使用了 JTAG UART，而在 NIOS II 系统中，JTAG UART 的驱动是阻塞的，而且，该串口是使用 JTAG 模拟的，因此，NIOS II 通过 JTAG UART 发送数据实际是将数据送入了 JTAG 的 FIFO 中，需要电脑端使用软件（如 Eclipse 中的 Console）来读取数据，如果数据不被读走，则 NIOS II 中的程序会一直在这里等待数据被读走，才能发送下一个数据，因此卡在该处无法继续运行，就出现死机的现象。

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

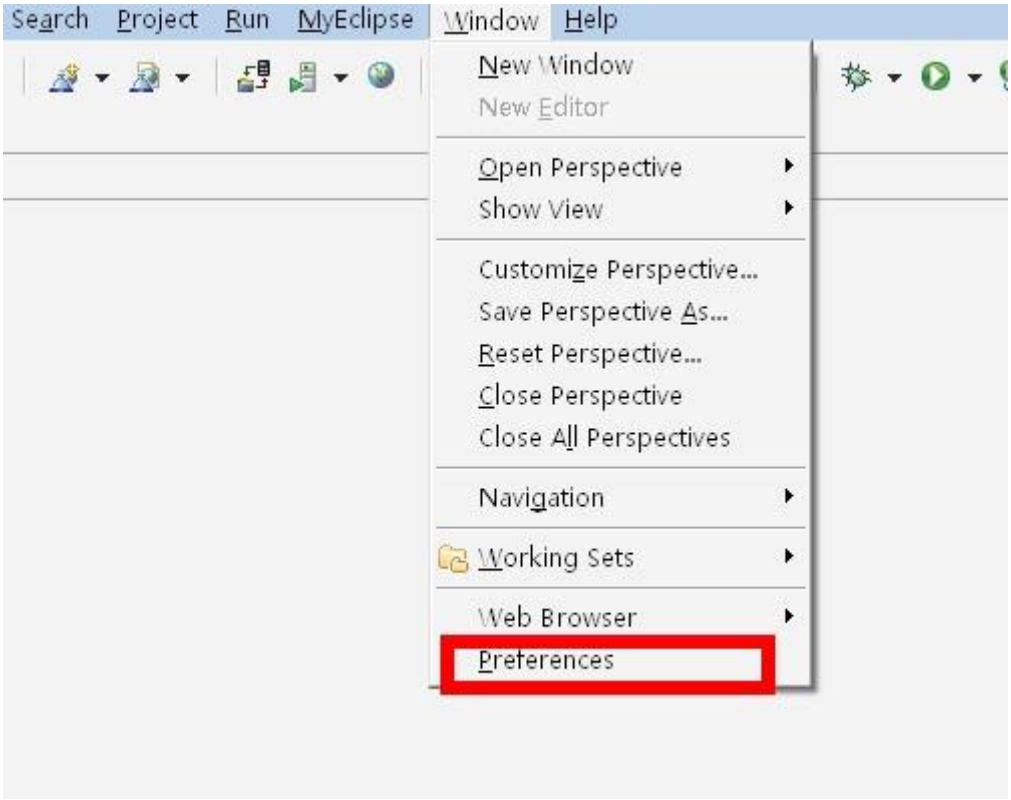
首先关闭 NIOS II IDE 软件，然后将该文件夹取得管理员所有权，再打开软件，编译就不会有问题了。

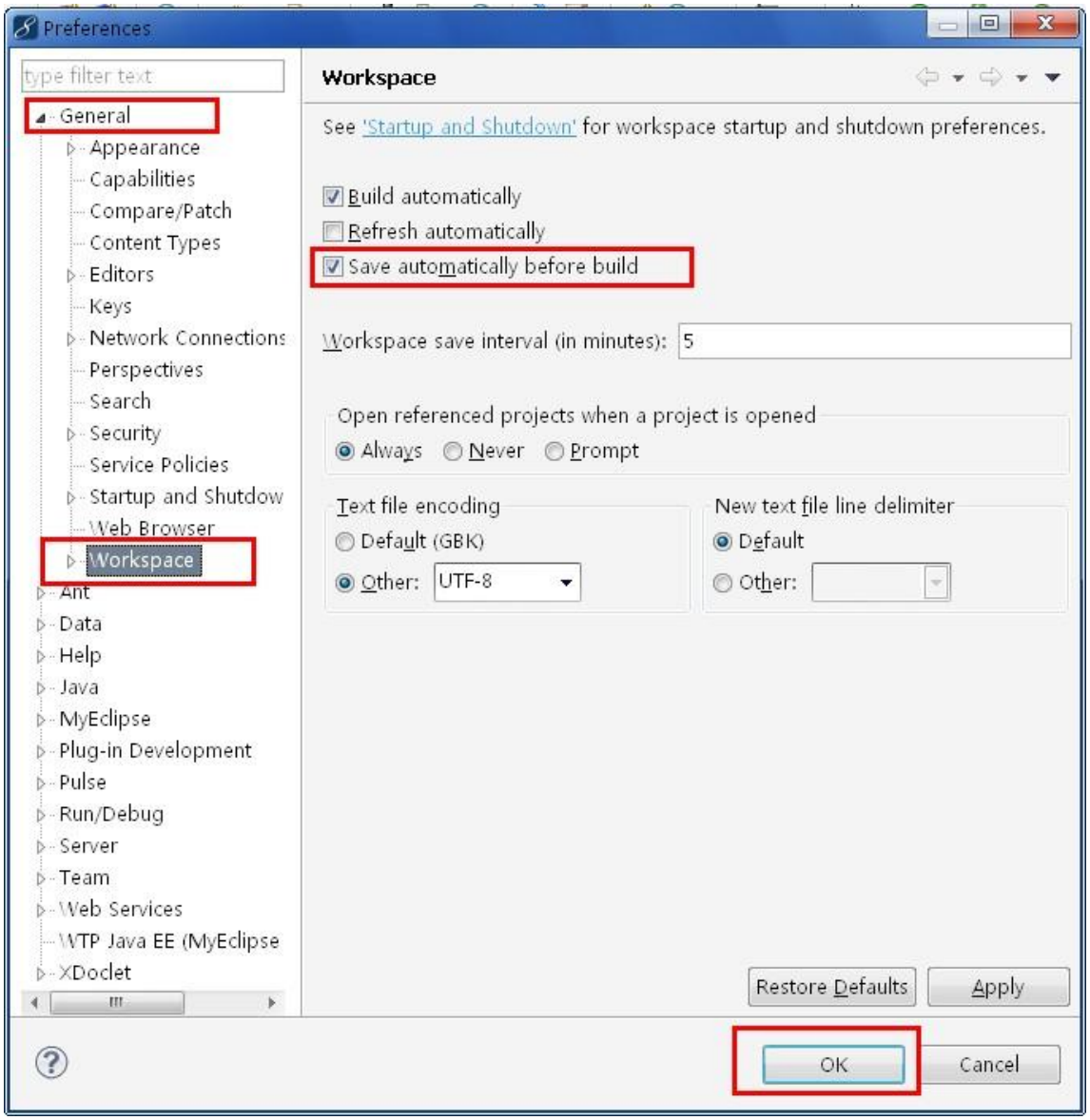
设置 Eclipse 在编译(build）前自动保存源代码文件

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！


Search
Project
Run
MyEclipse
Window
Help
New Window
New Editor
Open Perspective
Show View
Customize Perspective...
Save Perspective As...
Reset Perspective...
Close Perspective
Close All Perspectives
Navigation
Working Sets
Web Browser
Preferences



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

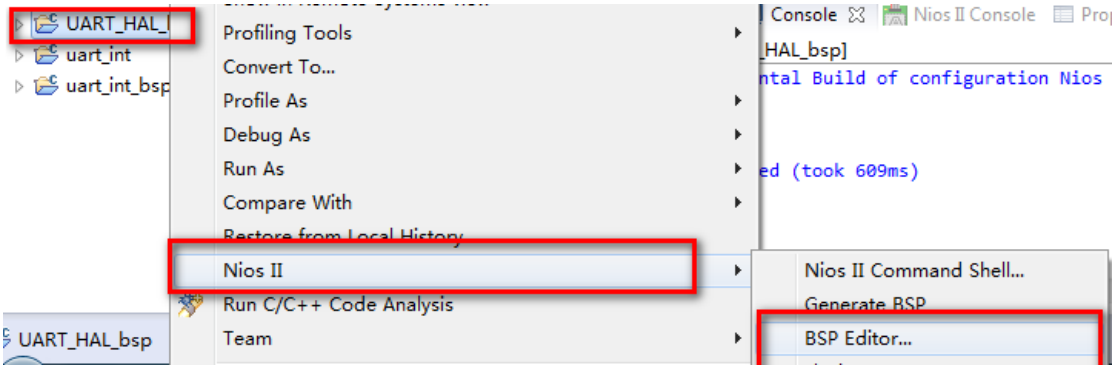
在面在 QSYS 中将 NIOS II CPU 的复位地址和异常地址修改为 SDRAM

4. 我们需要继续使用之前创建的 NIOS II 软件工程。

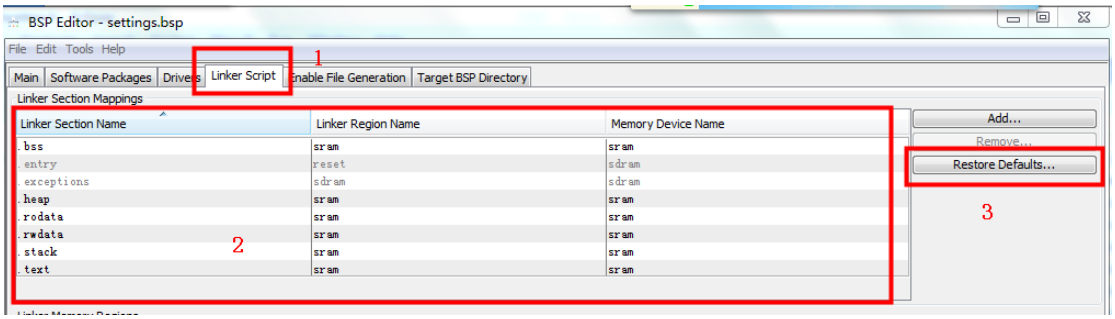
这里，如果我们更改后直接使用原来的软件工程，则编译会报错，因此我们需要在 BSP editor 中重新映射各个数据段的分配，然后重新 generate BSP，否则将导致固件无法烧写或者烧写后程序无法运行。

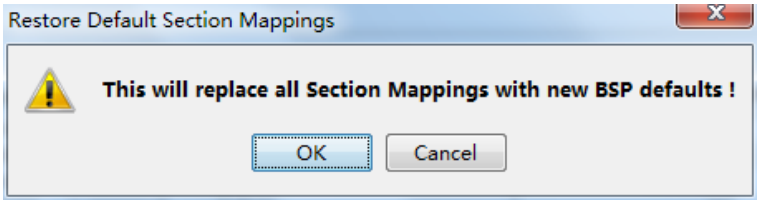
例如，在芯航线 FPGA 开发套件提供的 NIOS II 教程代码中，我们第 5 个实验和第 6 个实验分别是在 SRAM 和 SDRAM 中运行同样的软件工程（串口驱动设计实验），我们第 6 个实验的 QSYS 系统是直接在第 5 个实验的基础上增加了 SDRAM，并将 NIOS II CPU 的复位地址和异常地址由 SRAM 切换为了 SDRAM。这时候，如果我们直接打开之前的 NIOS II 软件工程，编译会报错。（这里默认用户已经知道并在 settings.bsp 文件中更改了 bsp 工程路径，如果不知道怎么更改的，请参看我的博文：）

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：

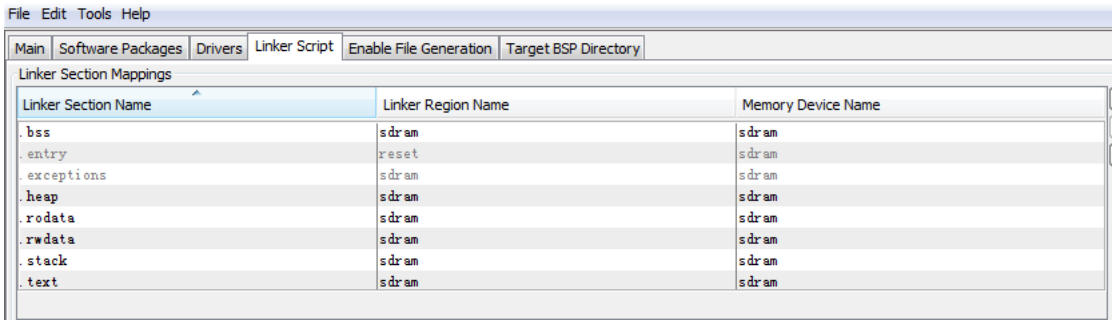


在弹出的界面中，切换到【Linker Script】选项卡，可以看到，各个数据段都还是指向了 SRAM，很显然这是不对的，因此我们点击【Restore defaults】按钮。在弹出的对话框中点击 OK 即可。





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。