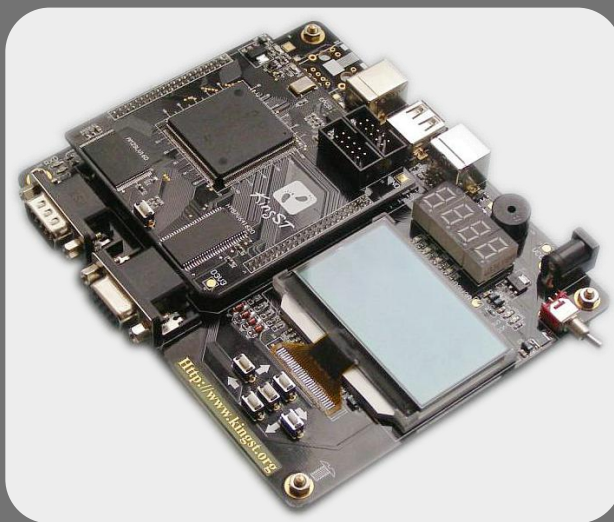


2010

NIOS II 那些事儿

--FPGA 黑金开发板配套教程

本套教程是 FPGA 黑金开发板配套教程，通过图文并茂的形式展现给读者，内容详细充实，由浅入深，逐步探索 NIOS II 技术，特别适合 NIOS II 的初学者阅读。



中国硬件开源网

[Http://www.oshcn.com](http://www.oshcn.com)



NIOS II 那些事儿

版本 V 5.0

软件版本：Quartus II 9.0 NIOS II 9.0 IDE

作者 将随时可能对本教程中的内容进行更改，这些改动不事先通知，但将会编入新版教程中，并上传到相关的网站上。

版权所有 **马瑞 (AVIC) QQ:984597569 Email : avic633@gmail.com**

技术博客：[Http://kingst.cnblogs.com](http://kingst.cnblogs.com)



《**NIOS II 的那些事儿**》by AVIC(马瑞) is licensed under a [Creative Commons 署名-相同方式共享 2.5 中国大陆 License](http://creativecommons.org/licenses/by-sa/2.5/).

Copyright © 2009-2010

目录

第一章 功能简介	5
一、 前言	7
二、 图片	7
三、 核心板配置	12
四、 下扩展板配置	12
第二章 硬件开发	15
一、 前言	17
二、 建立工程	17
三、 构建 NIOS II 软核	22
1. 构建 CPU 模块	24
2. 建立 SDRAM 模块	26
3. 建立 Avalon 三态桥	27
4. 建立 CFI 模块	28
5. 建立 SYSTEM ID	30
6. 建立 JTAG UART	31
7. 配置及编译 NIOS II	33
8. 分配管脚	38
四、 建立锁相环 PLL 模块	39
五、 调整 FLASH 引脚	44
六、 TCL 脚本文件	46
七、 配置工程	49
八、 下载程序	54
第三章 软件开发	56
一、 回顾	58
二、 摘要	58
三、 NIOS II IDE 简介	58
四、 建立软件工程	59
五、 编译	65
六、 运行	68
第四章 程序下载	72
一、 简介	74

二、	下载配置文件.....	74
三、	下载软件程序.....	79
第五章 编程规范		82
一、	规范参照标准.....	84
二、	格式	84
1.	缩进	84
2.	空格及空行.....	84
3.	大括号	85
三、	元素及命名规则.....	85
1.	文件	85
2.	宏、枚举体.....	85
3.	自定义类型.....	86
4.	函数声明及实体	86
5.	变量及初始化.....	87
6.	注释	88
四、	项目管理	89
五、	一些建议	90
1.	代码编辑器.....	90
2.	PC 端编译器及集成开发环境	90
3.	参考资源及网站	90
六、	示例代码	90
1.	C 文件	90
2.	h 文件.....	92
第六章 LED 实验		94
一、	简介	96
二、	硬件开发	96
三、	软件开发	103
第七章 中断实验		115
一、	简介	117
二、	硬件开发	119
三、	软件编程	124
四、	总结	129
第八章 串口实验		130
一、	简介	132
二、	硬件开发	132
三、	软件开发	134
第九章 RTC 实验		144

一、	简介	146
二、	硬件开发	146
三、	软件开发	148
第十章	SPI 实验	158
一、	简介	160
二、	硬件开发	160
三、	软件开发	165
第十一章	IIC 实验	172
一、	简介	174
二、	硬件开发	174
三、	软件开发	175
第十二章	定时器	183
一、	简介	185
二、	硬件开发	185
三、	软件开发	187
第十三章	SDRAM	196
一、	简介	198
二、	软件开发	198
第十四章	EPCS 下载	202
一、	简介	204
二、	硬件设置	204
三、	软件设置	205
第十五章	FLASH 编程	207
一、	简介	209
二、	软件开发	209
第十六章	AVALON	215
一、	简介	217
二、	DHL 模块设计	218
三、	硬件设计	223
四、	软件开发	230
第十七章	数码管	232
一、	简介	234
二、	例程	234
第十八章	USB (一)	237
一、	简介	239

二、	硬件开发	239
三、	软件开发	242
四、	上位机开发	251
第十九章	USB (二)	254
一、	简介	256
二、	软件设计	257
第二十章	附录	268
一、	NIOS II 下关于无法进行寄存器方式操作 PIO 的问题解析	270
二、	对寄存器结构体的详细解析	275
三、	TCL 脚本文件	279
四、	NIOS II 常见问题解答 (FAQ)	283
五、	黑金开发板印制板	289
1.	核心板	289
2.	扩展板	290

第一章 功能简介

功能简介

通过本章，您可以详细的了解黑金开发板的功能模块，以及核心板的功能。

本章分为以下几个部分：

- 一、前言
- 二、图片
- 三、核心板配置
- 四、下扩展板配置

一、 前言

真的不好意思，让大家等的太久了，升级版的黑金开发板终于可以跟大家见面了。借着发布《NIOSE 那些事儿》新版本的机会，简单介绍一下有关黑金开发板的各项功能。根据大家的需求，我们对黑金开发板进行了不断的修改，因此时间也是越拖越长，不过我们相信用这段时间的等待换取的升级版黑金开发板是值得的。熟话说，没有最好只有更好，对于这句话我非常认同，我们也是本着这个原则来开发黑金开发板的，我们会让黑金开发板不断的成长，不断的创新，满足大部分人的需求。下面我边上图片边给大家介绍升级版黑金开发板的资源配置及其他一些情况。

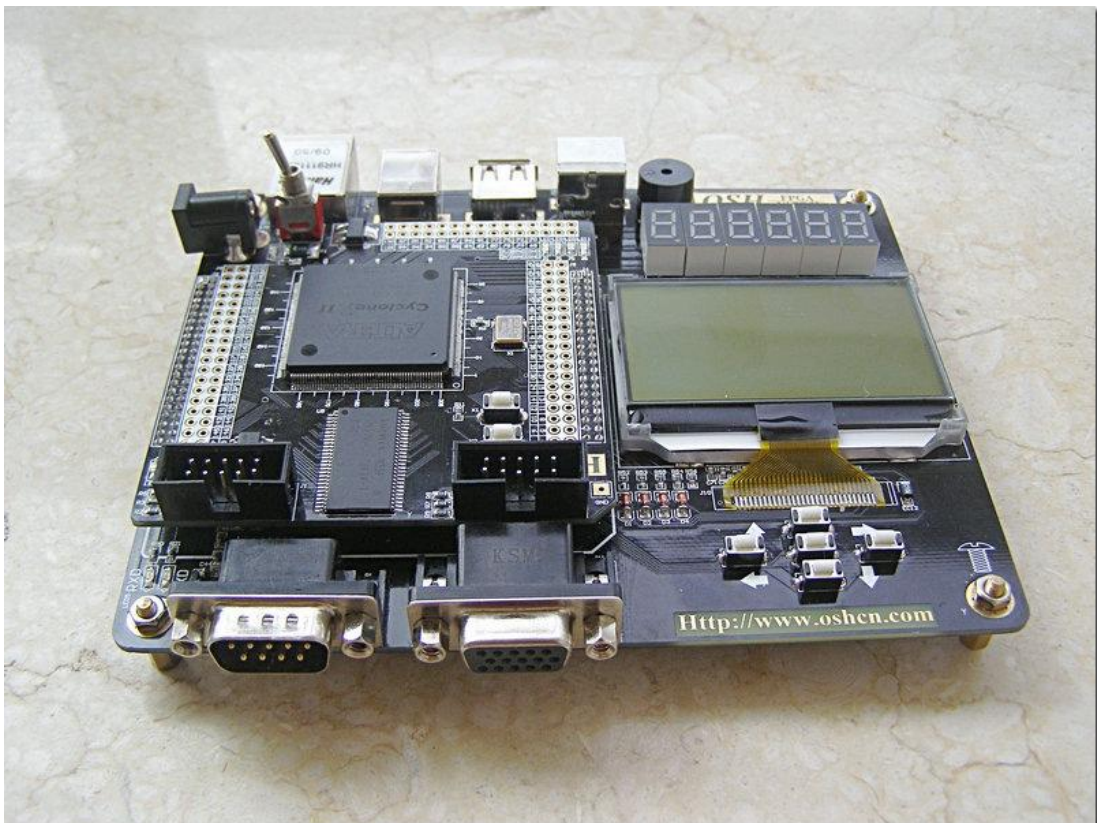
二、 图片

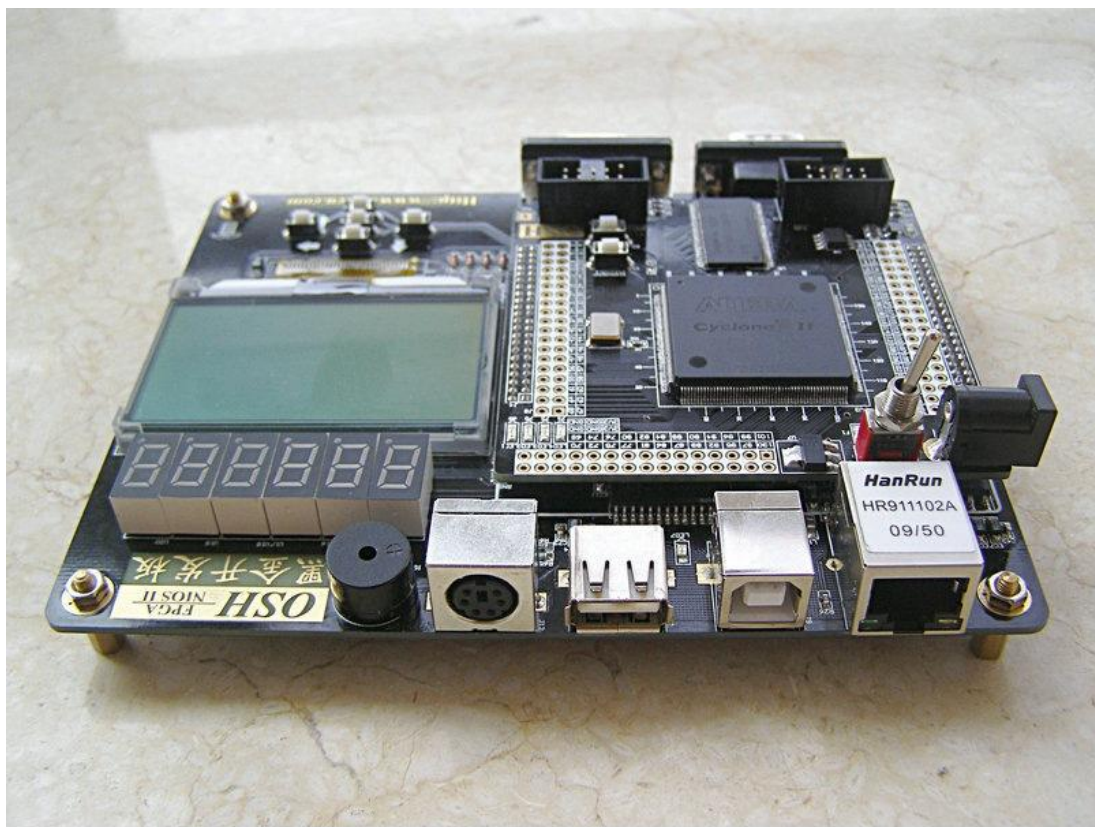
先上两个全身照，大家可以有个整体的概念



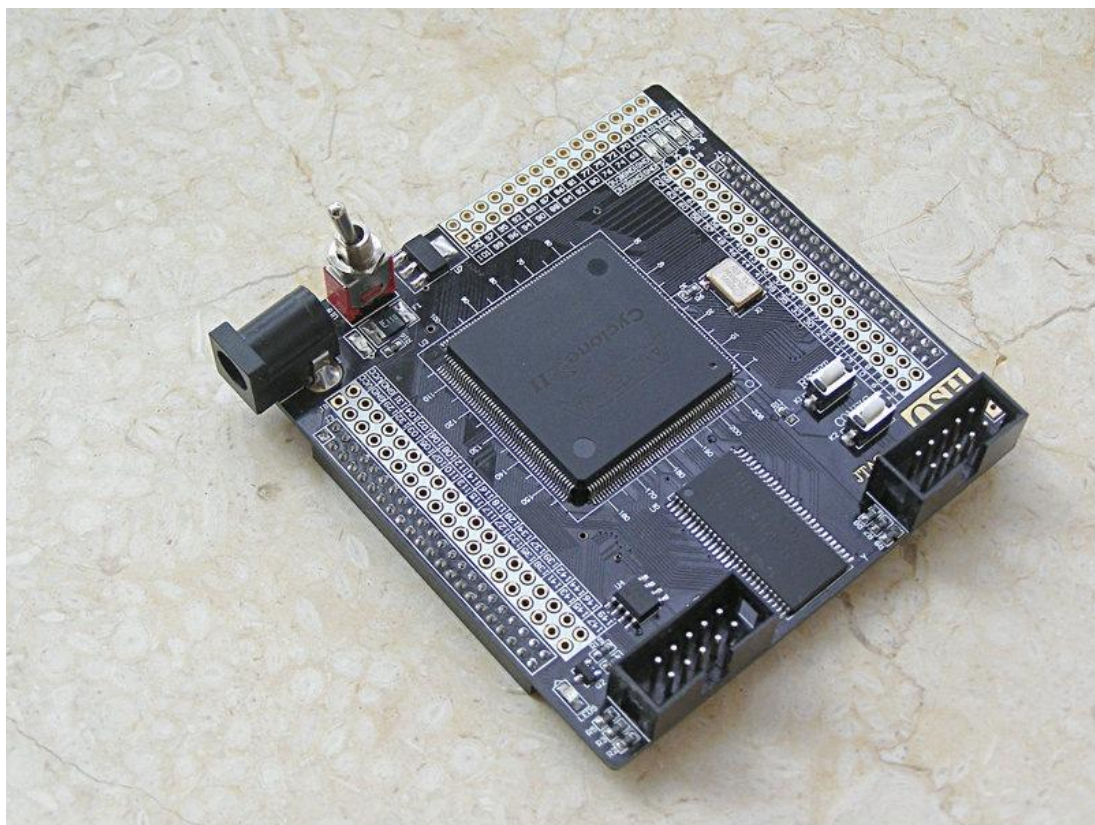


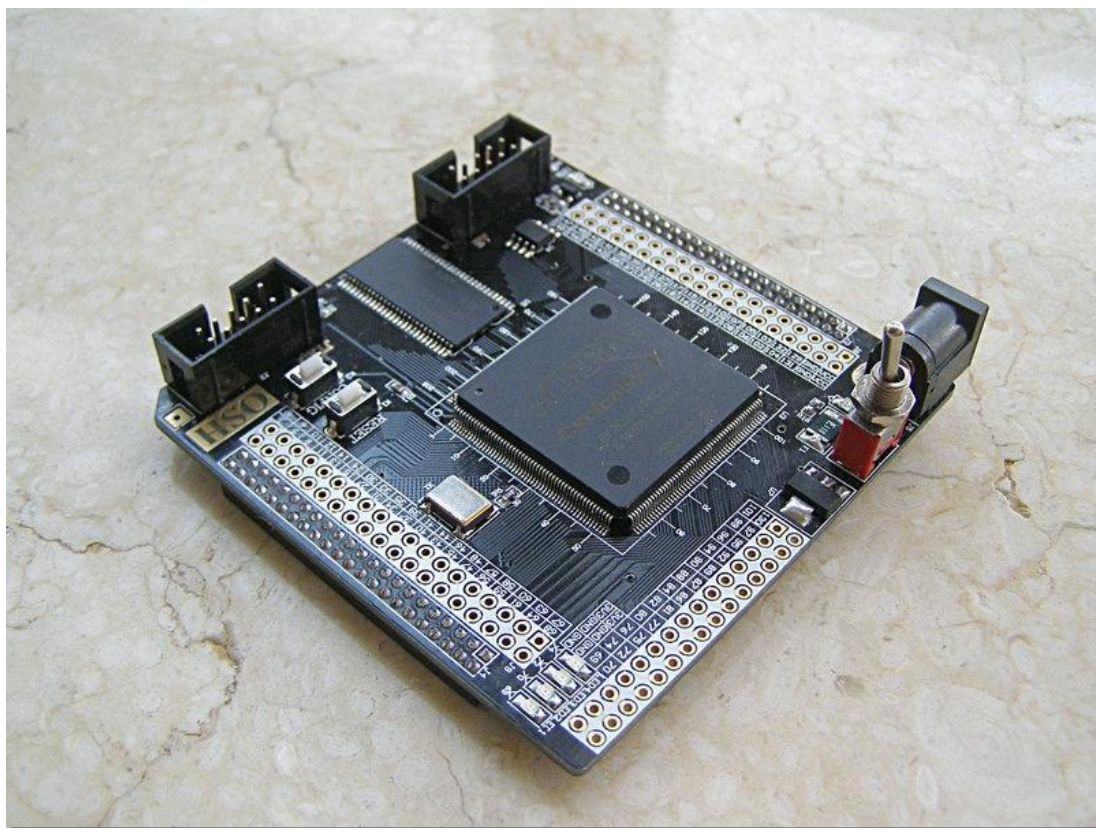
再来两张局部的，这样就能把接口都看到了





接下来，看看核心板吧，核心板尺寸是 7.6*7.6cm

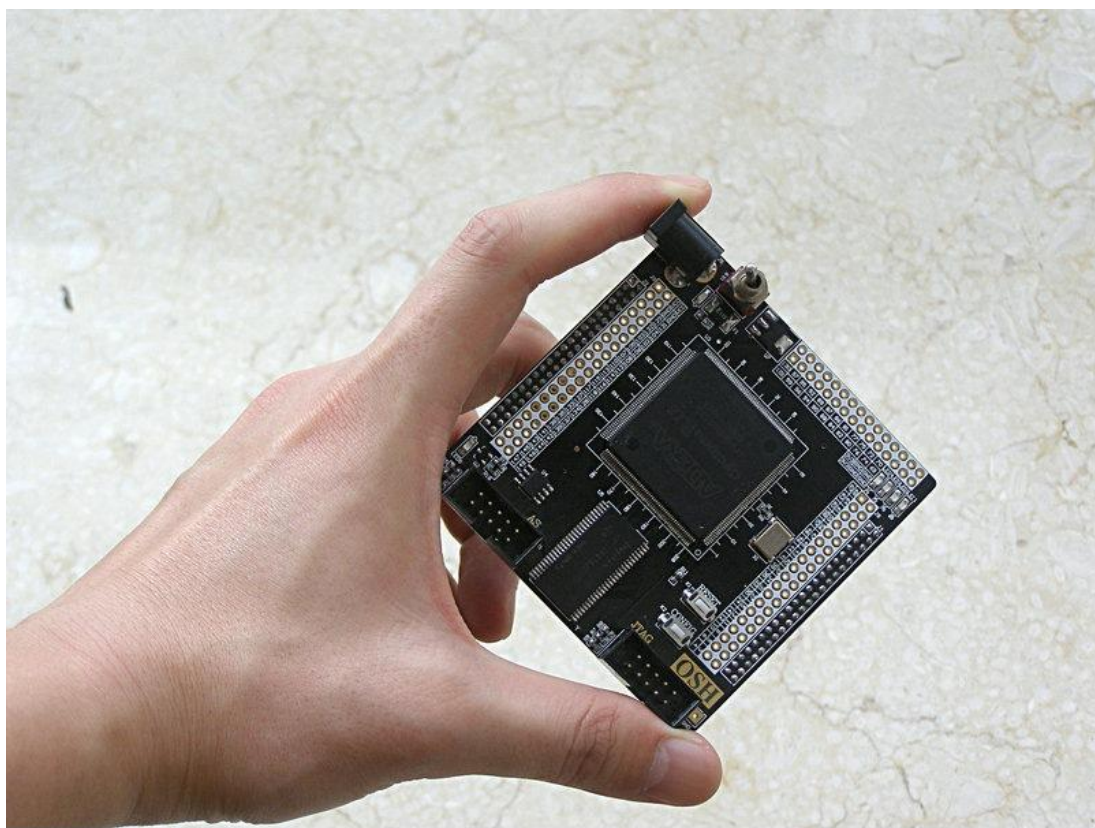
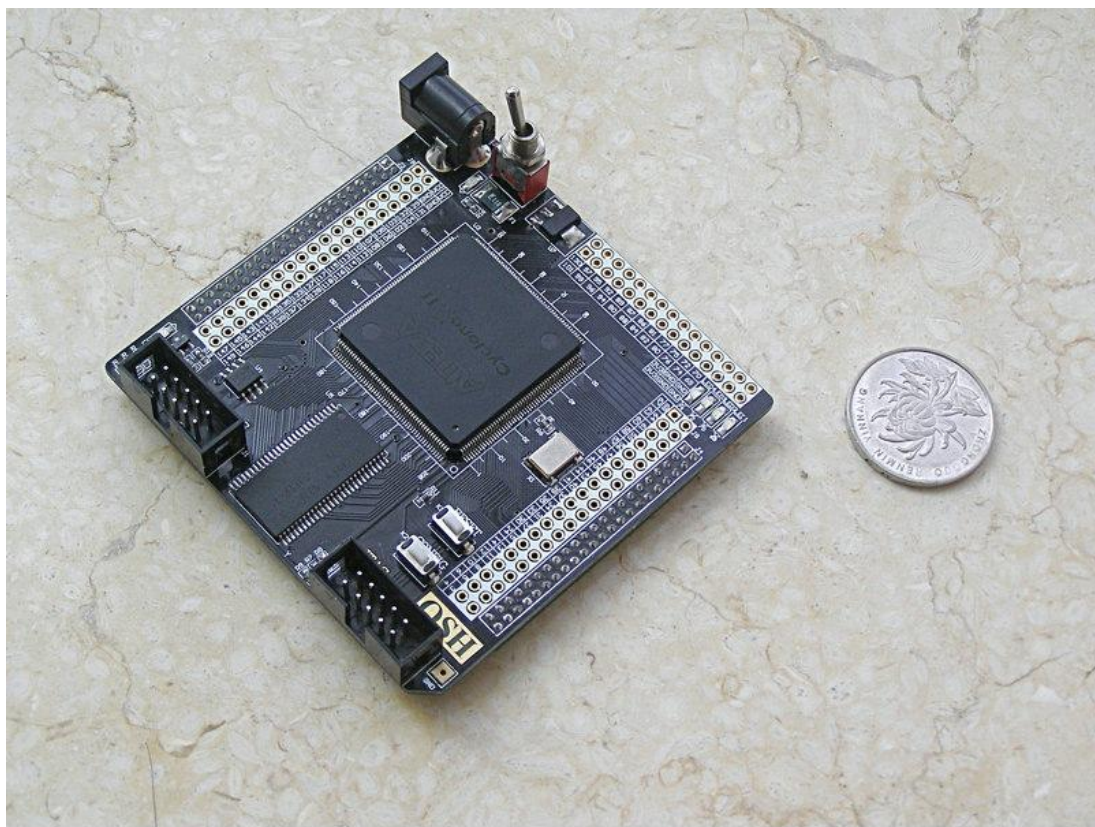




下面是底板，底板还是挺空的，还有很大的发展空间，哈哈



最后上两张核心板尺寸对比照片，还有只手呢，不要被吓到啊，哈哈



三、 核心板配置

- 核心板配置的 FPGA 芯片是 Cyclone II 系列的 **EP2C8Q208C** 具有 **8256** 个 LEs , **36** 个 M4K RAM blocks (4Kbits plus 512 parity bits) , 同时具有 **165,888bit** 的 RAM , 支持 **18** 个 Embedded multipliers 和 **2** 个 PLL , 资源配备十分丰富。实验证明,这款芯片在嵌入 NIOS II 软核将黑金开发板的所有外设全部跑起来,仅占全部资源的 70-80% ;
- 核心板同时配备了 **64Mbit** 的 SDRAM , 对于运行 NIOS 软核提供了有力的保障,这款芯片为时钟频率有 143MHz , 实验证明, NIOS II 软核主频可以平稳运行 120MHz , 速度还是相当快的 ;
- **16Mbit** 的配置芯片也为这款核心板增色不少,不仅可以存储配置信息,同时还可以实现 NIOS II 软件程序存储,你编写的程序再大也没有后顾之忧了。
- 20M 的有源晶振也是必不可少的,他是整个系统的时钟源泉 ; 4 个 LED 对于调试来说更是提供了很多方便 ; 复位按键,重新配置按键,配置指示灯一个也不能少 ; 同时支持 AS 模式和 JTAG 模式 ;
- 除此以外, **核心板一个更大的特点是它可以独立于底板单独运行**,为此配备了 5V 的电源接口,高质量的红色开关,为了安全还加入了自恢复保险丝。当然扩展口是不能少的,除了 SDRAM 占用的 38 个 IO 口外,其他 **100 个 IO 全部扩展出来**,为大家可以进行自我扩展实验做好了充分的准备。

四、 下扩展板配置

为了让 FPGA 发挥它的强大功能,黑金开发板为其设计一款资源丰富的下扩展板(之所以叫下扩展板,是因为我们后续还会有上扩展板)。下面我们就来简单介绍一下下扩展板的资源配置。

- **支持网络功能**,配置 ENC28J60 网口芯片。ENC28J60 是 Microchip Technology (美国微芯科技公司)推出的 28 引脚独立以太网控制器。目前市场上大部分以太网控制器的封装均超过 80 引脚,而符合 IEEE 802.3 协议的 ENC28J60 只有 28 引脚,既能提供相应的功能,又可以大大简化相关设计,减小空间 ;
- **支持 USB 功能**,配置 CH376 芯片。CH376 支持 USB 设备方式和 USB 主机方式,并且内置了 USB 通讯协议的基本固件,内置了处理 Mass-Storage 海量存储设备的专用通讯协议的固件,内置了 SD 卡的通讯接口固件,内置了 FAT16 和 FAT32 以及 FAT12 文件系统的管理固件,支持常用的 USB 存储设备(包括 U 盘 /USB 硬盘 /USB 闪存盘 /USB 读卡器)和 SD 卡(包括标准容量 SD 卡和高容量 HC-SD 卡以及协议兼容的 MMC 卡和 TF 卡) ;
- **支持板载 128*64 的点阵 LCD**。ST7565P 控制芯片,内置 DC/DC 电路,通过软件调节对比度。该芯片支持,并口和串口两种方式 ;

- **支持实时时钟 (RTC)**，配置 DS1302 芯片。DS1302 是美国 DALLAS 公司推出的一种高性能、低功耗、带 RAM 的实时时钟电路，它可以对年、月、日、周、时、分、秒进行计时，具有闰年补偿功能，工作电压为 2.5V ~ 5.5V。采用三线接口与 CPU 进行同步通信，并可采用突发方式一次传送多个字节的时钟信号或 RAM 数据。DS1302 内部有一个 31×8 的用于临时性存放数据的 RAM 寄存器。DS1302 是 DS1202 的升级产品，与 DS1202 兼容，但增加了主电源/后背电源双电源引脚，同时提供了对后背电源进行涓细电流充电的能力；
- **支持 EEPROM**，配置了 24LC04 芯片。24LC04 是 512*8bit 的 EEPROM，支持 IIC 接口；
- **支持 PS/2 接口**，可以实现 PS/2 接口的键盘和鼠标；
- **支持 RS232 串行接口**，可以实现串口数据的发送和接受；
- **支持 6 位共阳数码管**；
- **5 个独立按键**，可以与液晶配合，构造完美的人机界面；
- **支持 VGA 接口**；
- **支持蜂鸣器**。

基本就这么多，大家凑合看吧，呵呵。欢迎大家批评指点，您的评价就是黑金开发板的动力和源泉，也是我们对其改进的根本所在，在此谢谢大家了！

如果您对我们的黑金开发板感兴趣，可以通过以下方式跟我们联系：

Email : avic633@gmail.com

QQ : 984597569

Tel : 15026499986

同时，我们诚招校园代理，网络代理，以及全国实体代理，如果您对我们的产品感兴趣，就请跟我们联系吧。

第二章 硬件开发

硬件开发

通过本章，您可以详细的了解到 NIOS II 开发的硬件部分，通过 Quartus II 9.0 进行工程建立，NIOS II 软核构建，以及编译下载等，每一步都有图片配合指导，就算大家没有任何 NIOS II 开发基础，都可以顺利完成硬件开发部分内容。

本章分为以下几个部分：

- 一、前言
- 二、建立 quartus 工程
- 三、构建 NIOS II 软核
- 四、构建锁相环
- 五、下载

一、 前言

从今天开始，NIOS II 的学习征途正式拉开了。对于 NIOS 的学习爱好者，我相信这是一个福音，我将毫无保留的将我对 NIOS 的研究成果分享给大家。我之所以采用博客这种方式，就是想跟大家充分的交流，大家可以给我留言，也可以在 Ourdev 中提出问题，我将尽我的全力为大家解决问题。由于本人水平有限，如果有我解决不了的问题，还请高手们多多帮忙，我相信能为大家解决问题是一件很快乐的事情，你不会错过的。

废话少说，我们马上进入正题。今天是第一节，我首先说一下学习 NIOS 都需要哪些前提条件。听到这，初学者可能会有些害怕了，难道学习 NIOS 还要条件？是的，需要条件，不过这些条件并不是很高，只要大家努力，这些条件都不是问题。

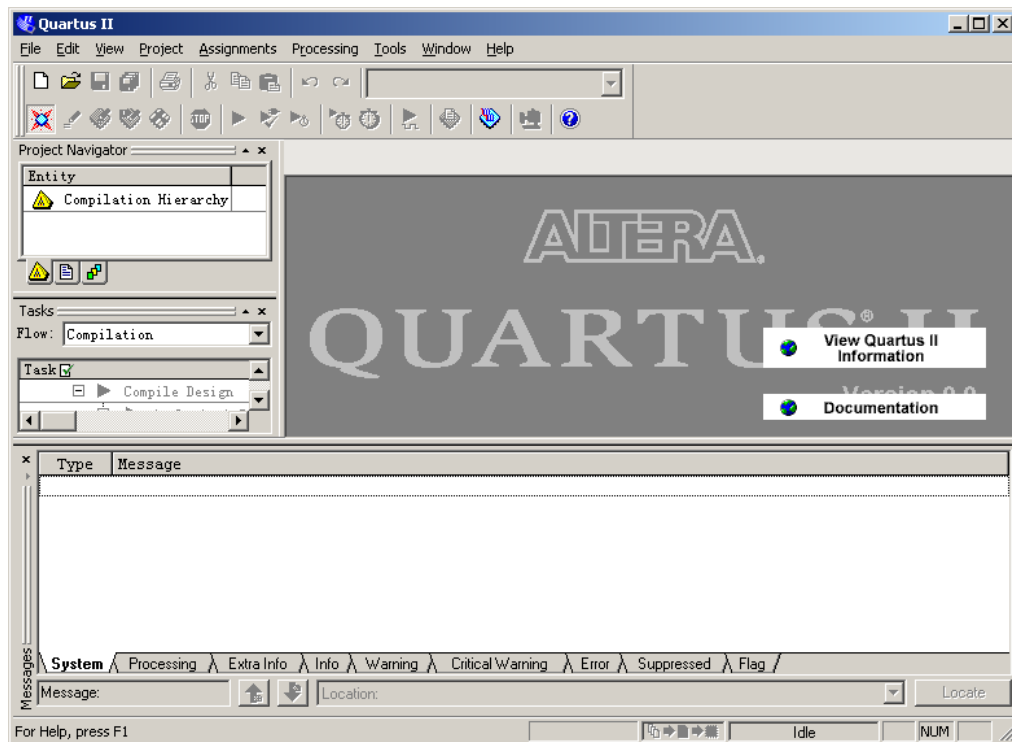
- 具有一定的单片机基础；
- 具有一定的 C 语言编程能力；
- 了解 Quartus II 的开发流程；
- 有一块开发板。

就这么多，大家觉得难么？首先说说第一条，具有一定的单片机基础，这个条件是要有的。单片机的基础在 NIOS II 学习中体现在它的寄存器操作方式上，这种操作方式是通用的，不管是 ARM，DSP，还是 51 都是一样的，你只要有一种单片机的实践经验就没问题了。再说第二条，这一条没什么可争议的，NIOS 的开发完全是用 C 语言的，如果你没有 C 语言的基础，我建议你还是先学习一下 C 语言再考虑学习 NIOS 吧。第三条呢，有最好，如果没有的话也可以，我在以后的文章中都会涉及到，大家跟着学就可以了。第四条也不是必须的，不过学习 NIOS 不像学习 Verilog HDL/VHDL，通过仿真看看也行，NIOS 的学习跟单片机很相似，最好是亲手操作硬件，这样对你的学习有更好的效率和效果。在这里推荐一下我的 FPGA 黑金开发板，不仅仅是广告哦，因为我以后的讲解都是以我的黑金板为基础的，大家学习起来也很方便的。

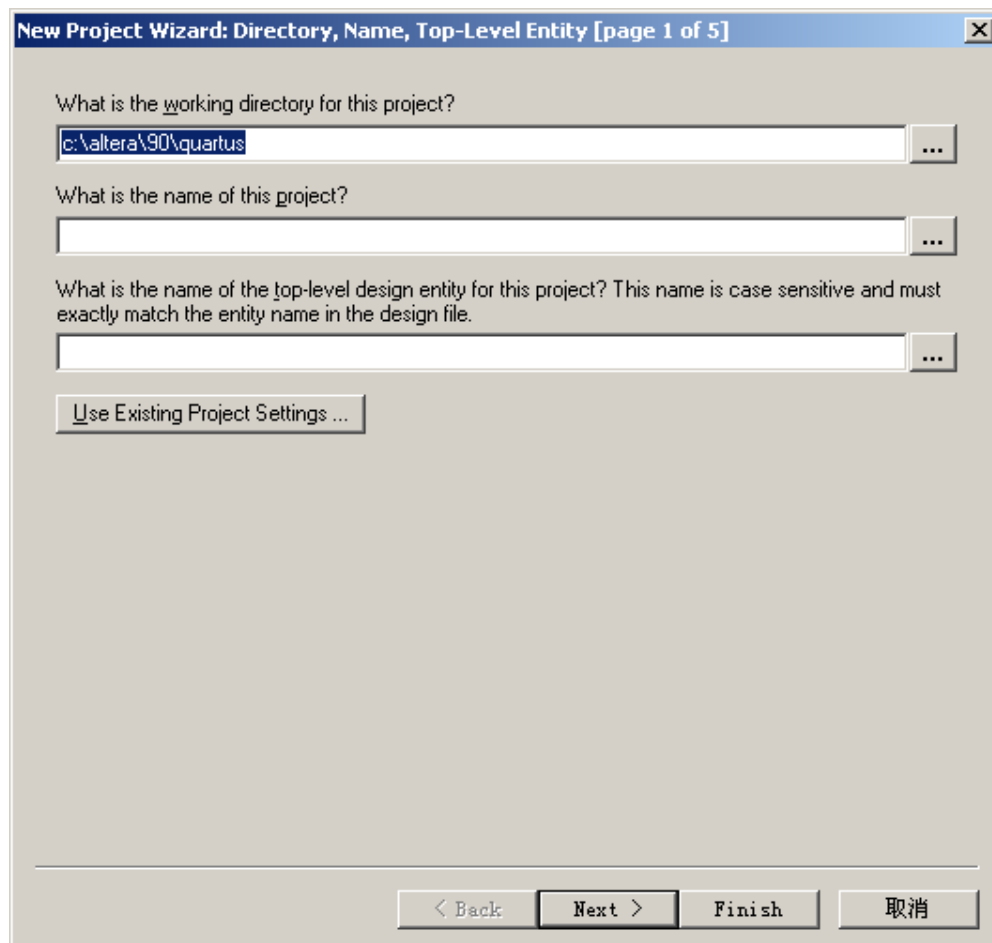
下面简单介绍一下 NIOS II 的内容，NIOS II 是一个用户可配置的通用 32 位 RISC 嵌入式处理器，它是 SOPC (System On a Programmable Chip, 片上可编程系统) 的核心。处理器以软核形式实现，具有高度的灵活性和可配置性。NIOS 的开发包括硬件开发和软件开发两部分。硬件开发是在 Quartus II 中实现的，而软件开发部分是在 NIOS IDE 软件中实现的。我们首先来介绍 NIOS 的硬件开发。所谓硬件开发就是用 Quartus II 和 SOPC builder 来建立自己需要的软核。

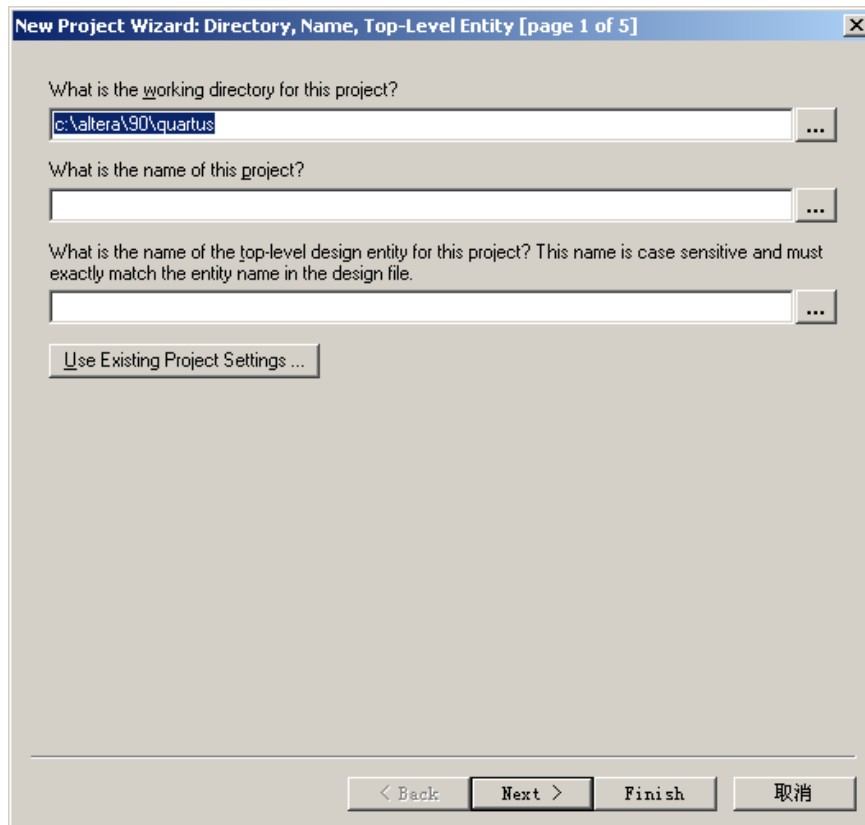
二、 建立工程

首先，打开 Quartus II 9.0 软件。

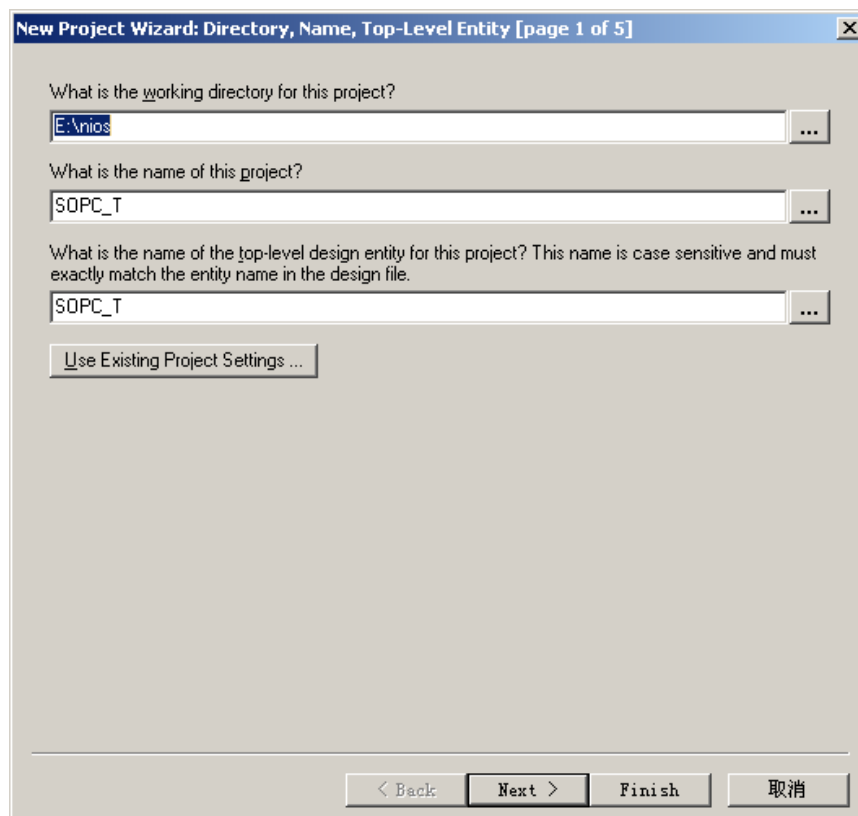


接下来，建立一个工程 File->New Project Wizard，

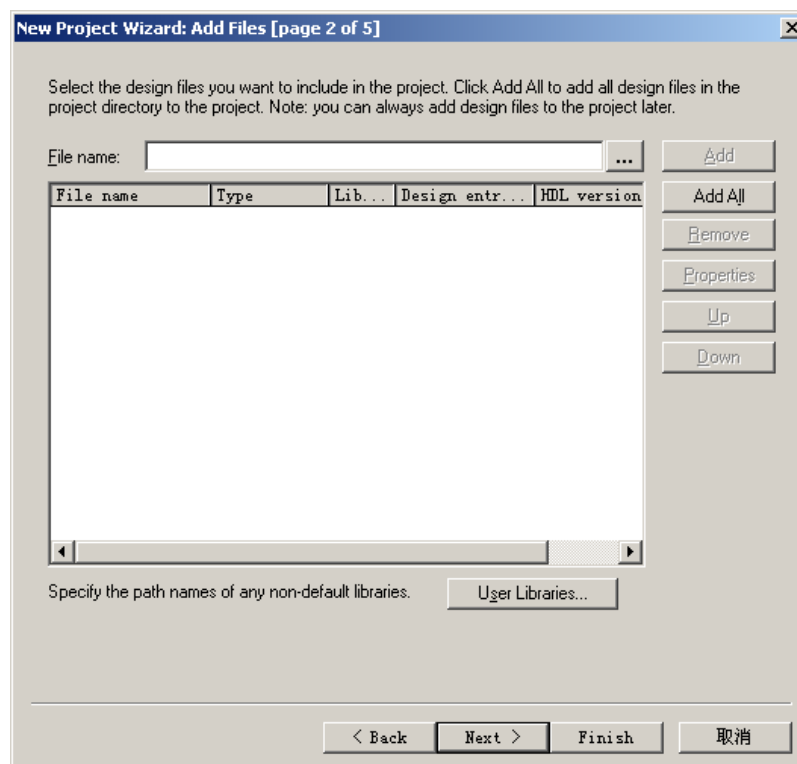




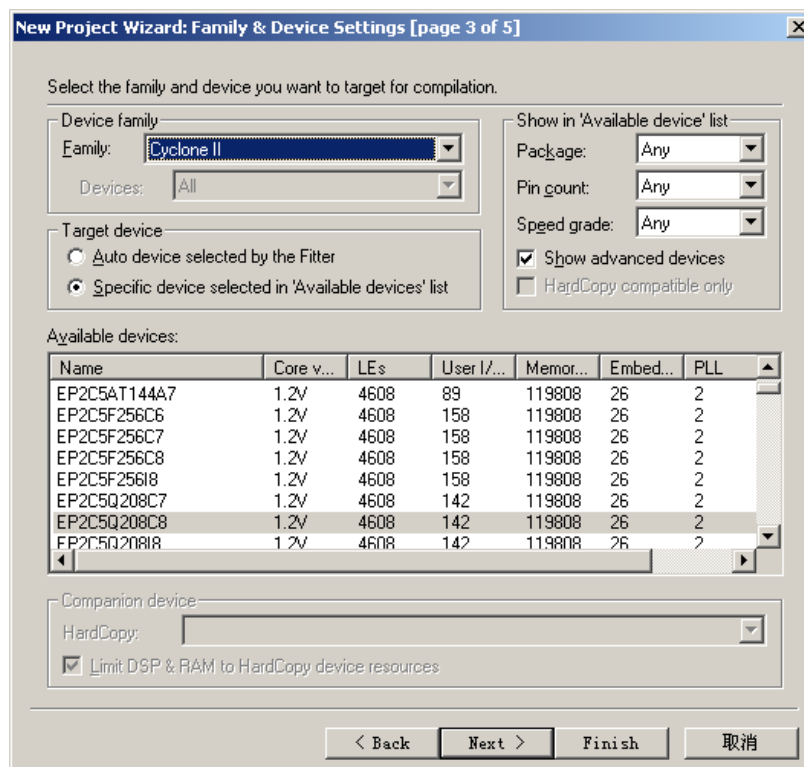
第一行是工程的路径，你选择你放置的路径即可，第二,第三行都是是工程名，写好以后如下，点击 Next，



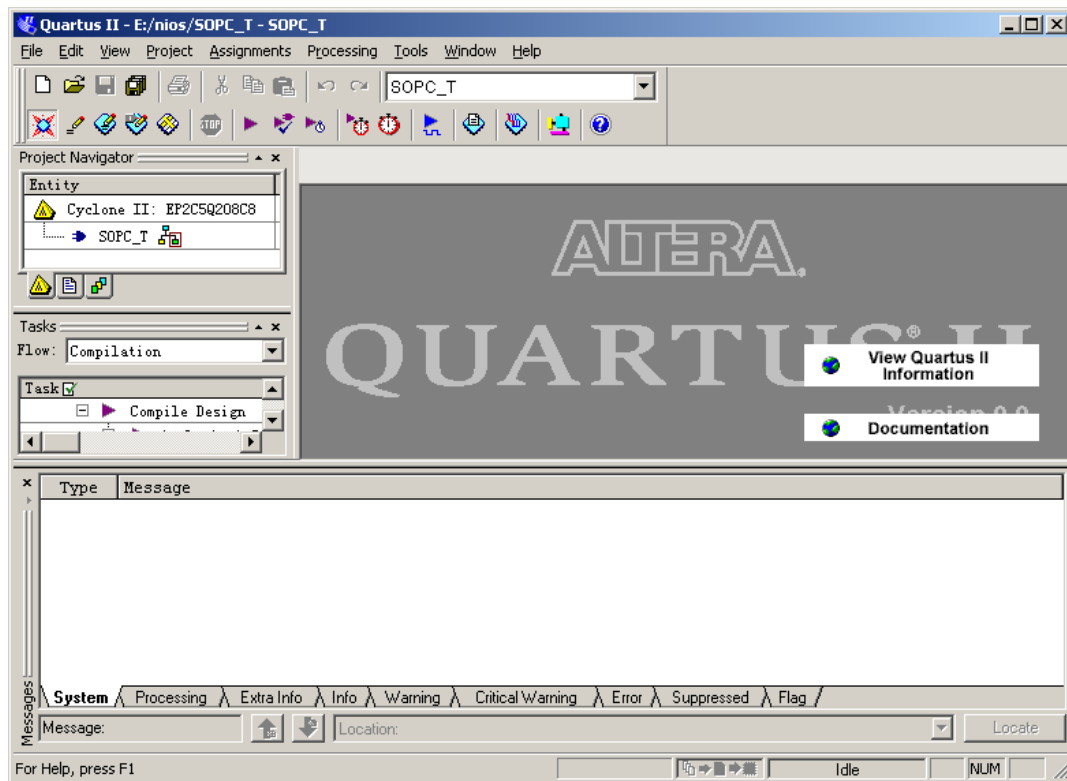
点击后，如下图所示，这个不需要改动，接着点击 Next



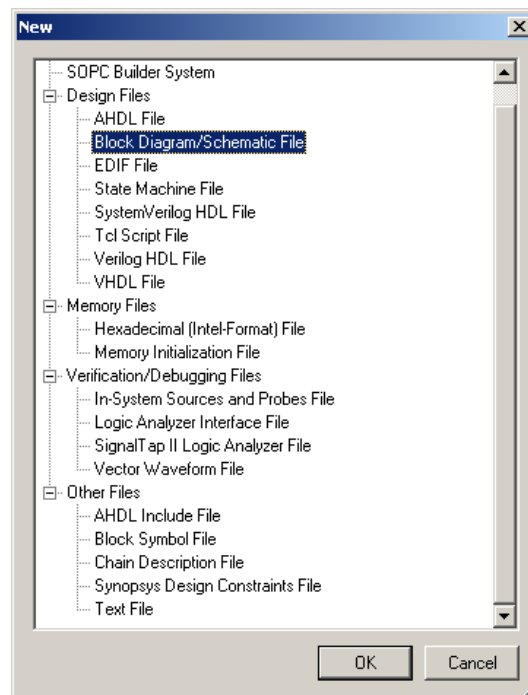
点击后，如上图所示，Family 里选择 Cyclone II，在 Available devices 中选择 EP2C5Q208C8(具体内容根据你的芯片所定)。



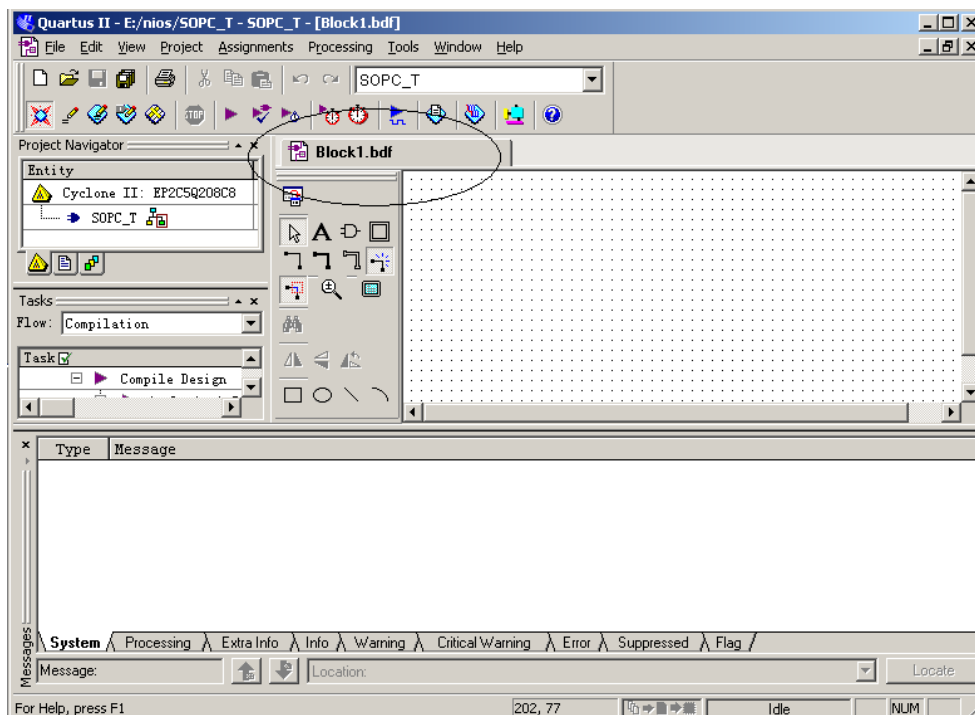
接着点击 Next，不需要修改，点击 Finish，显示如下图所示。



到此为止，工程已经建立完成。接下来，需要建立一个 Block Diagram/Schematic File，点击 File->New，如下图所示

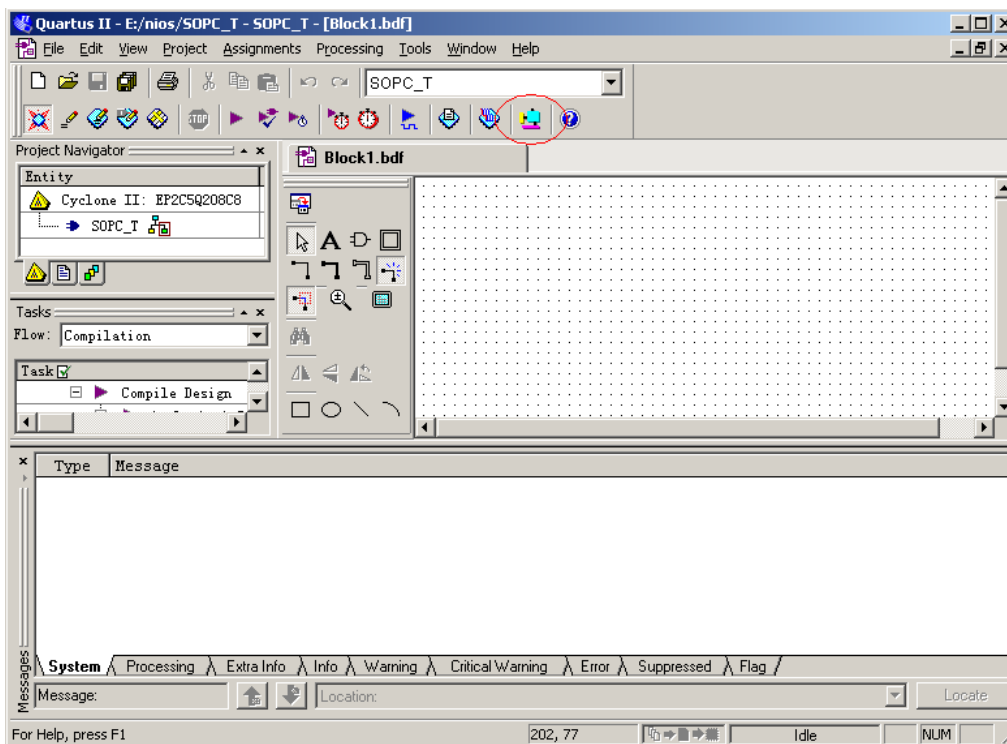


点击 OK，完成建立，工程中出现了一个 Block1.bdf 文件

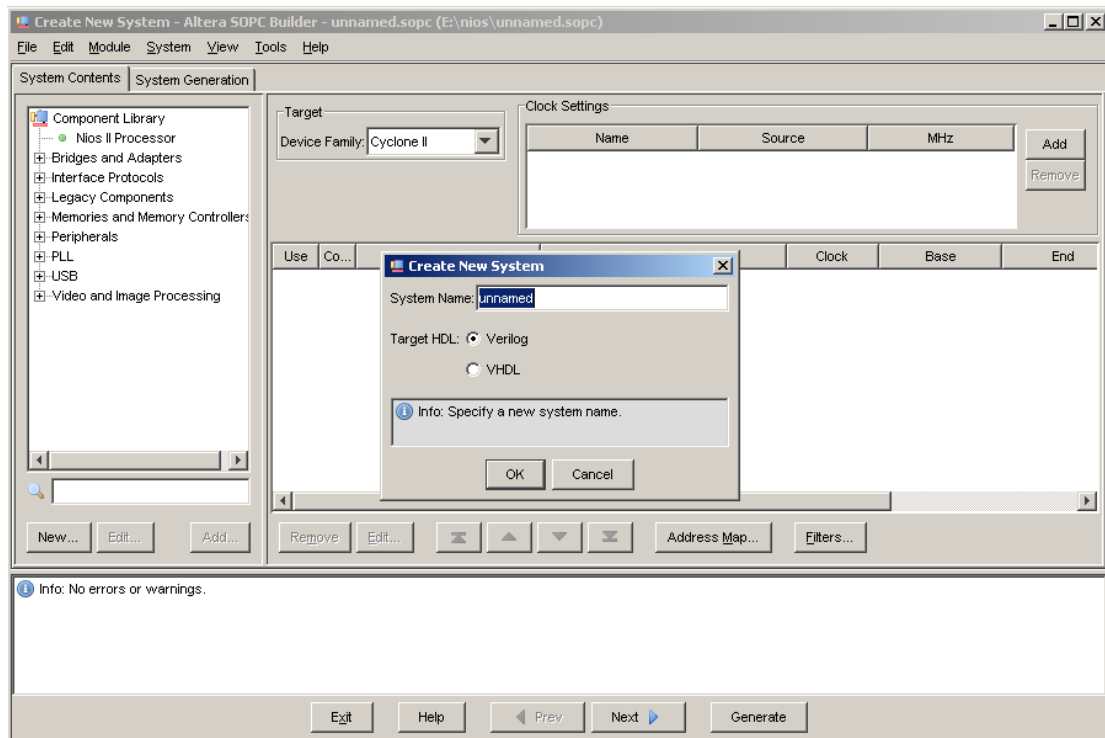


三、 构建 NIOS II 软核

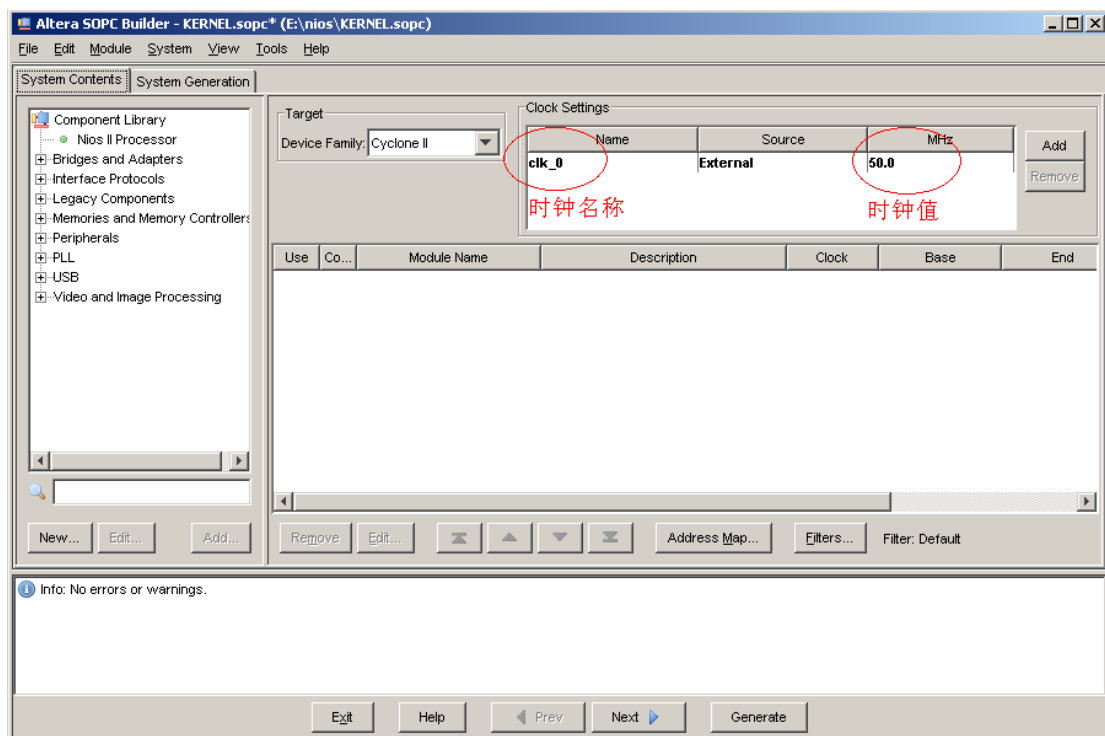
接下来，我们进入了构建软核环节，点击 Tools->SOPC Builder 或者下图红圈标示的图标



点击以后，SOPC Builder 运行，界面如下图示



System Name 中输入软核的名字：我将其命名为 KERNEL。点击 OK 后，如下图所示

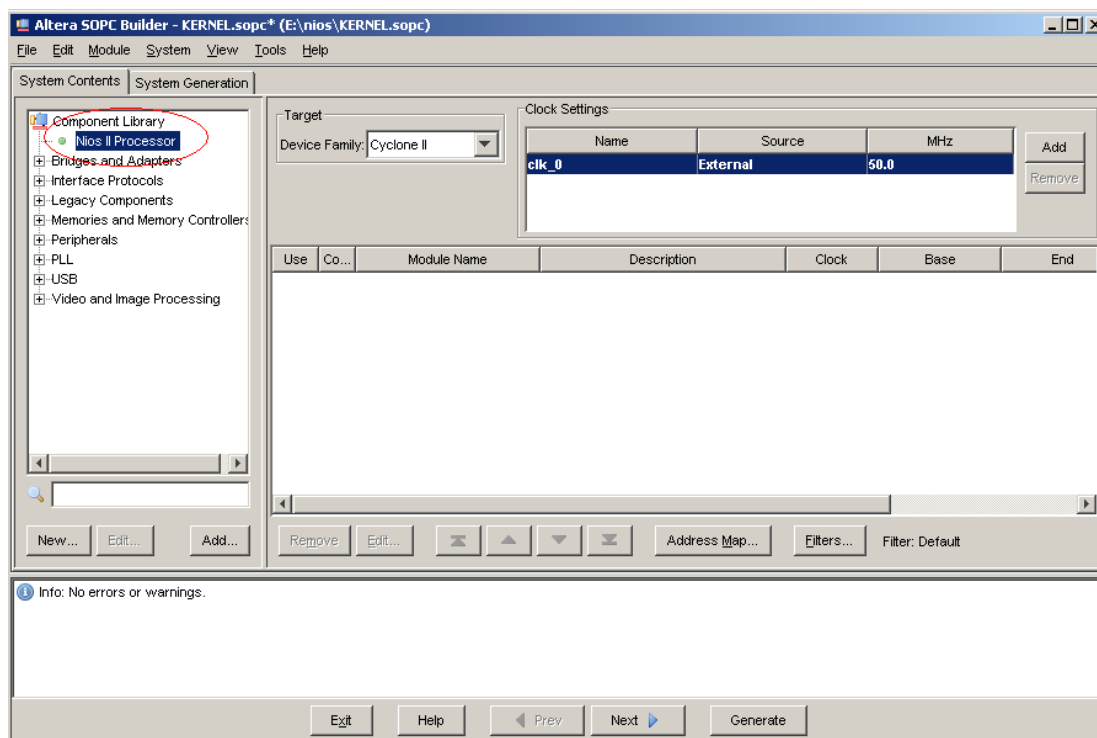


按图中标注的，clk_0 为时钟名称，50.0 为时钟值（单位为 MHz），我们可以对他们进行修改。用鼠标点击 50.0，将其改为 100.0。这时候，我们的软核时钟就为 100.0MHz

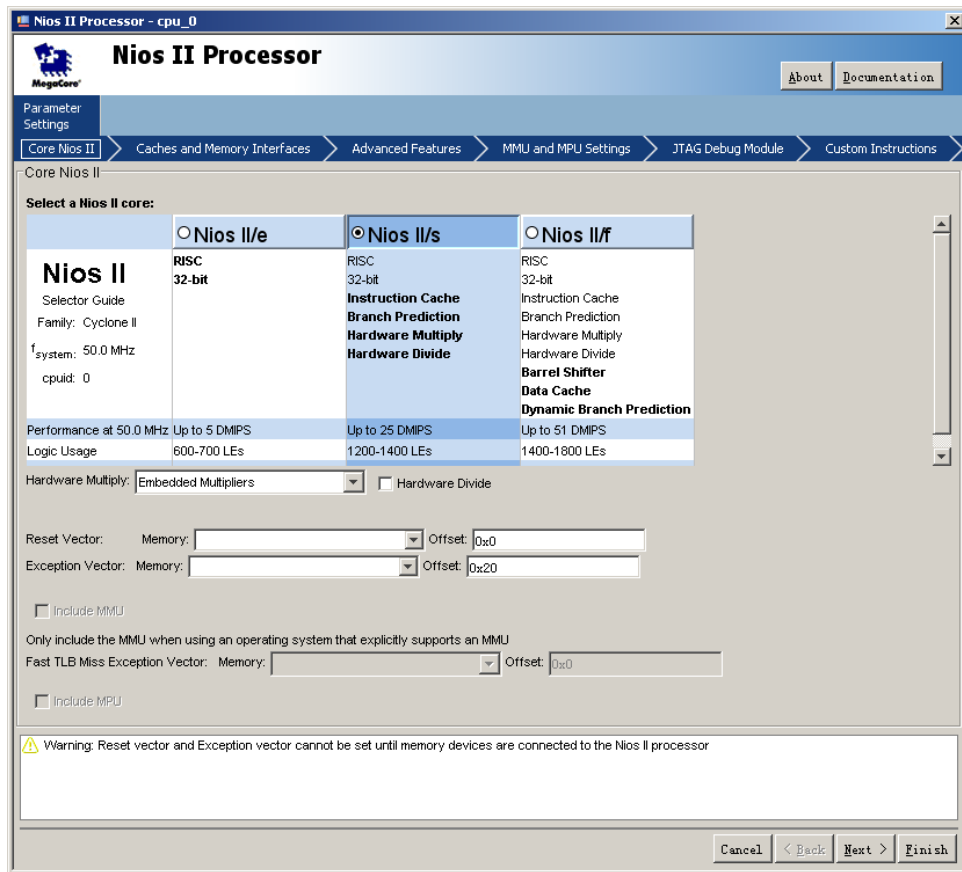
了。这是软核建立的第一步，接下来，我们要建立 Nios II Processor。

1. 构建 CPU 模块

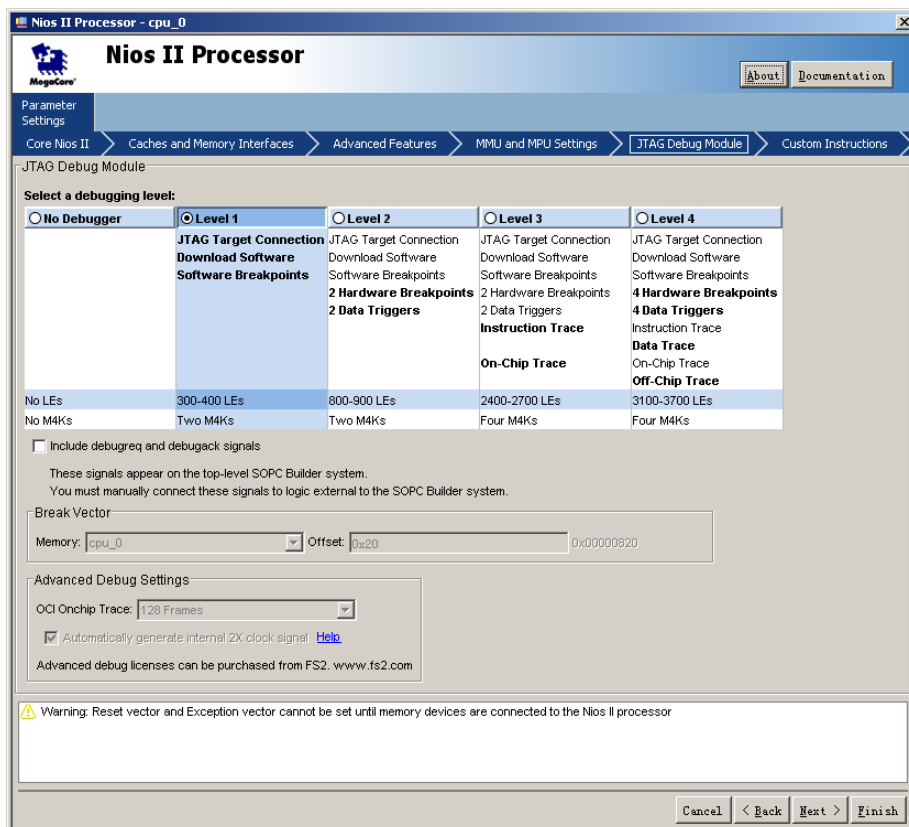
用鼠标点击左侧边框的红圈处 Nios II Processor，如下图示



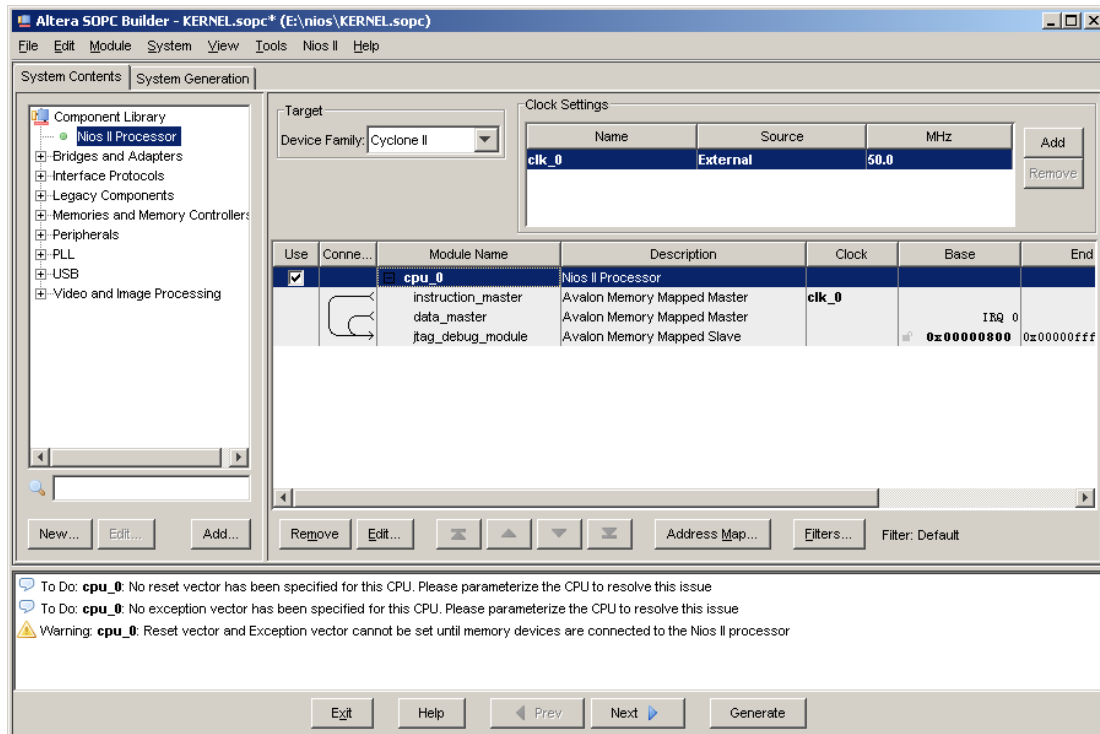
点击后 将出现下图 这一步我们来选择软核的类型。这里给我们提供了三种类型，Nios II/e 占用资源最少 600-800LEs，功能也最简单，速度最慢。Nios II/s 占资源比前者多一些，功能也多了，速度也快一些 Nios II/f 占资源最多，功能也最多，速度就快。选择的时候要根据你的需求和你的芯片资源来决定。在这里，我选择 Nios II/s，功能和速度都可以得到满足。下面的 Reset Vector 是复位后启动时的 Memory 类型和偏移量，Exception Vector 是异常情况时的 Memory 类型和偏移量。现在还不能配置，需要 SDRAM 和 FLASH 设置好以后才能修改这里，这两个地方很重要。



接下来连续点击 Next，一直到下图为止

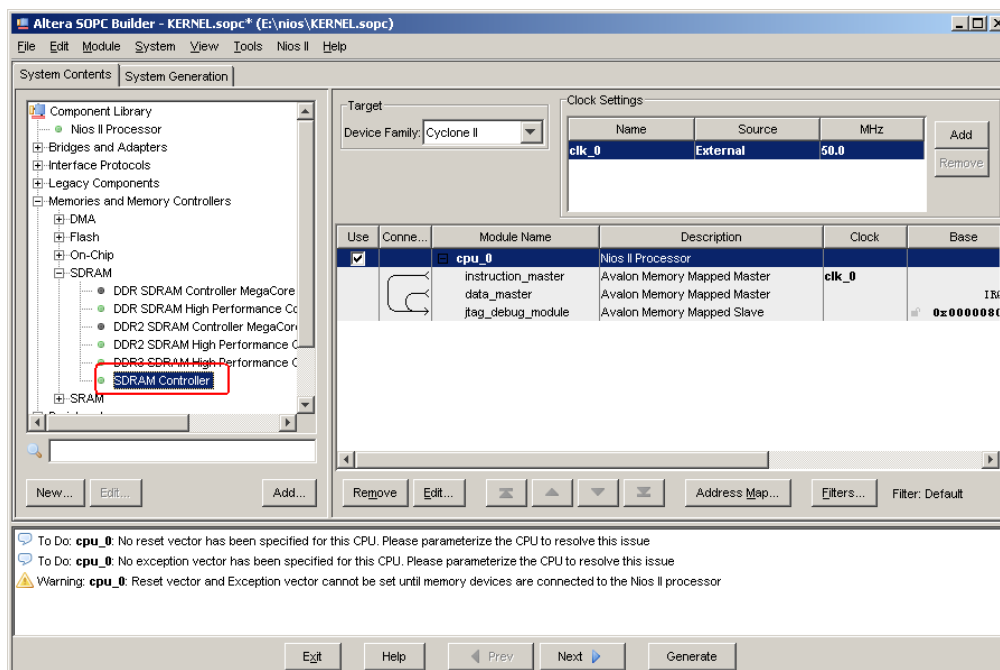


这里设置 JTAG Debug Module，即 JTAG 调试时所用到的功能模块。功能越多，需要的资源越多，这里，我们选择 Level 1 即可，不需要过多其他的功能。点击 Finish，结束 Nios II Processor 的建立后，如下图所示，

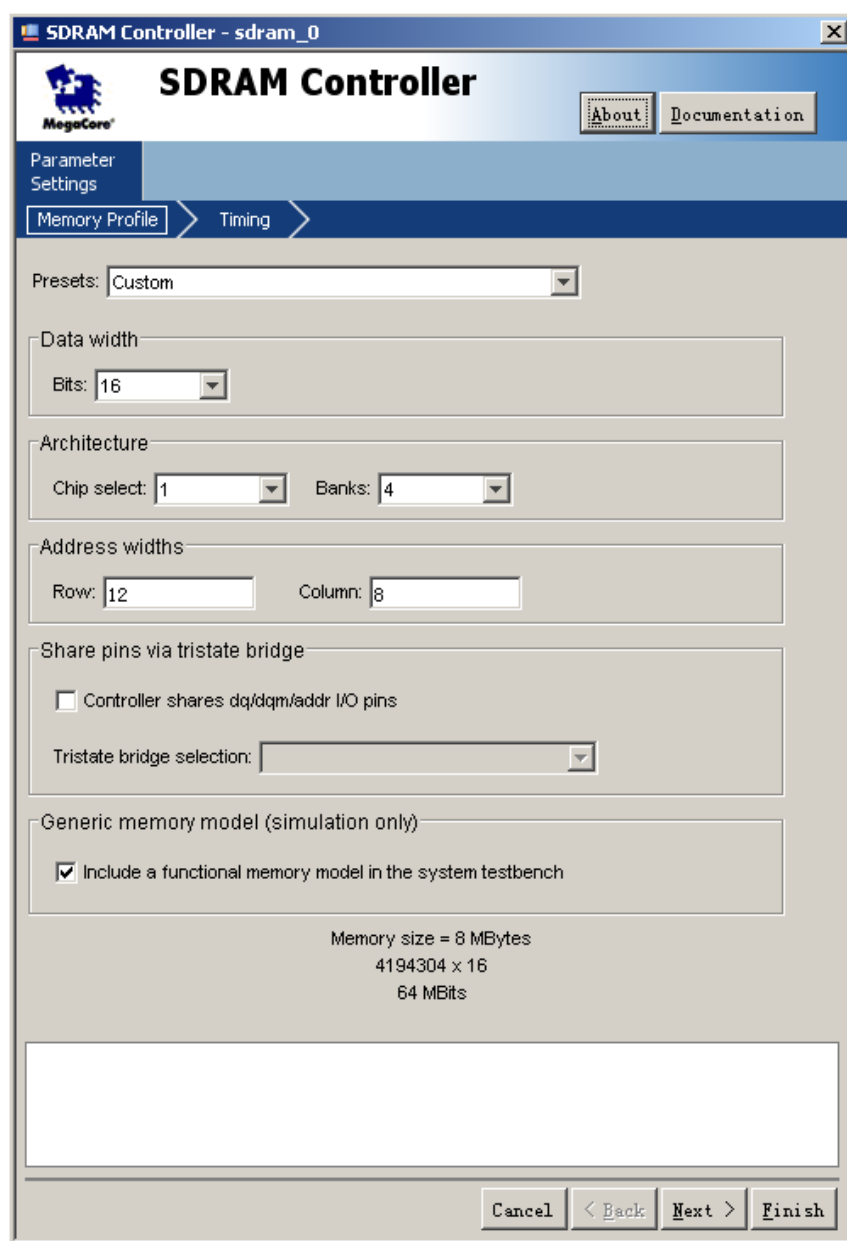


2. 建立 SDRAM 模块

接下来，我们要建立 SDRAM 控制器，点击下图红框所示的地方



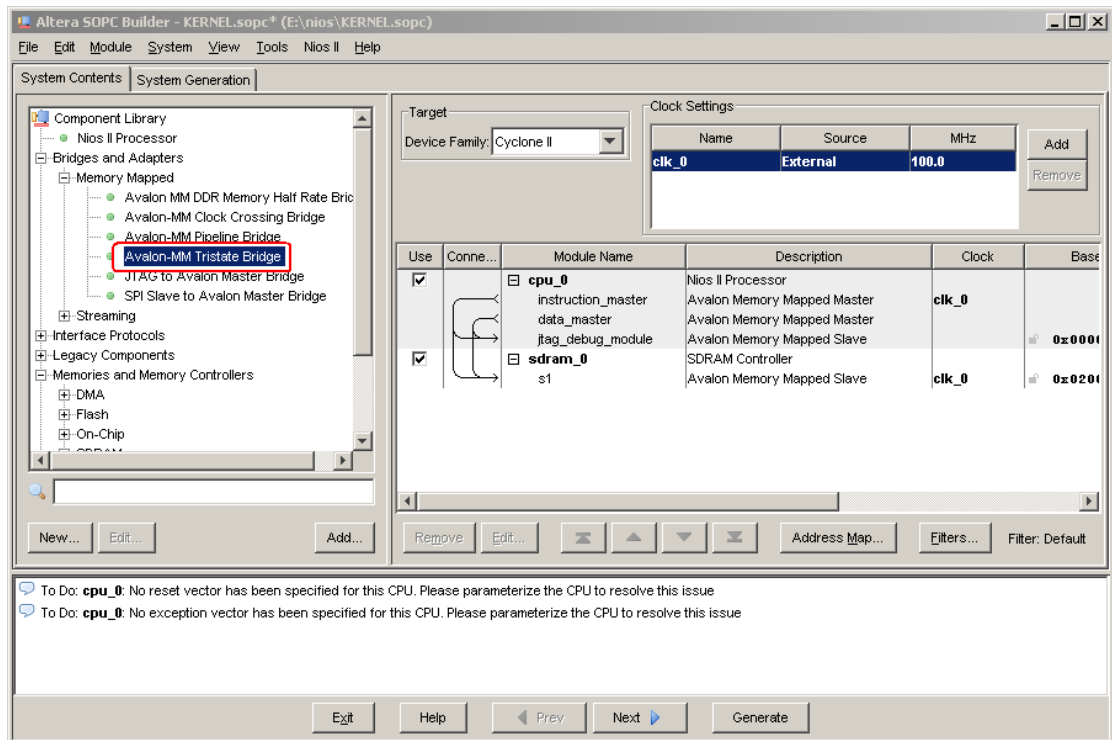
点击后，如下图所示



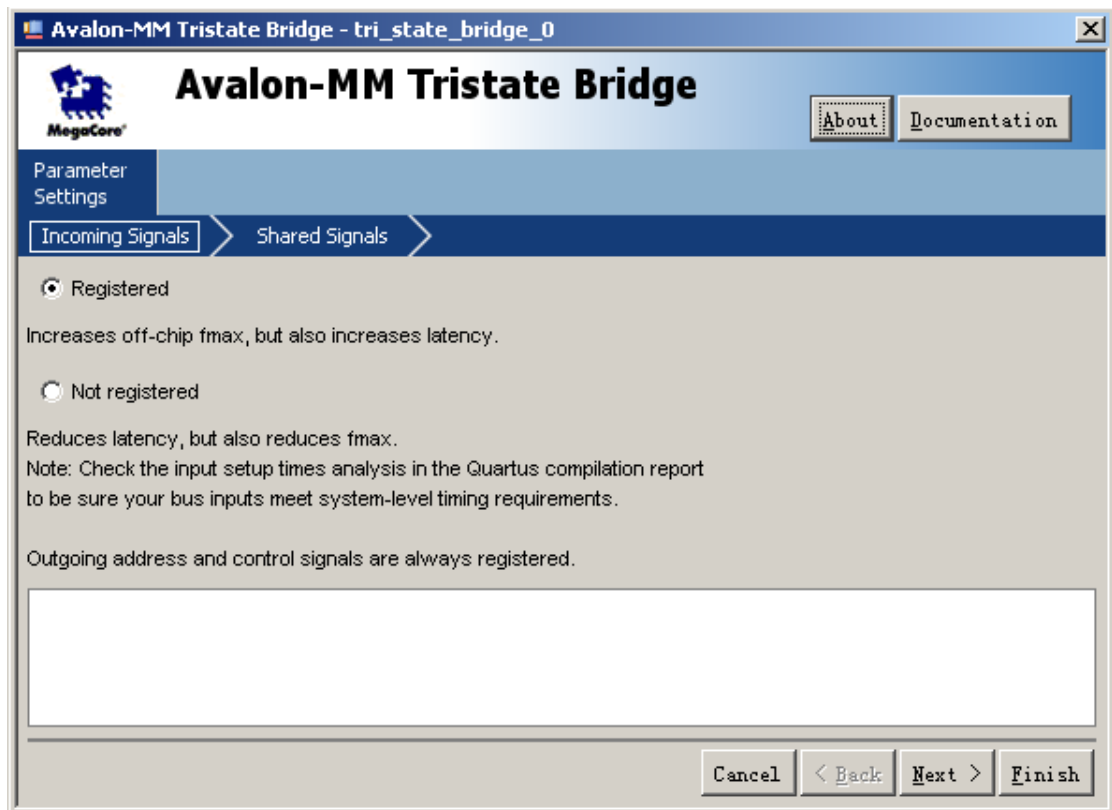
在 Presets 中选择第一项 Custom，在 Bits 中选择 16，其他项不动，点击 Next，点击 Finish，完成 SDRAM 控制器的设置。在这里之所以选择 16，是因为我们用的 SDRAM（HY57V641620）是 16 位的。

3. 建立 Avalon 三态桥

接下来，我们要建立一个 Avalon 三态桥，在 NIOS 系统中，要实现与 FPGA 片外存储器通信，就必须在 Avalon 总线和连接外部存储器的总线之间添加一个桥，这个桥就是 Avalon 三态桥。点击下图所示红圈处

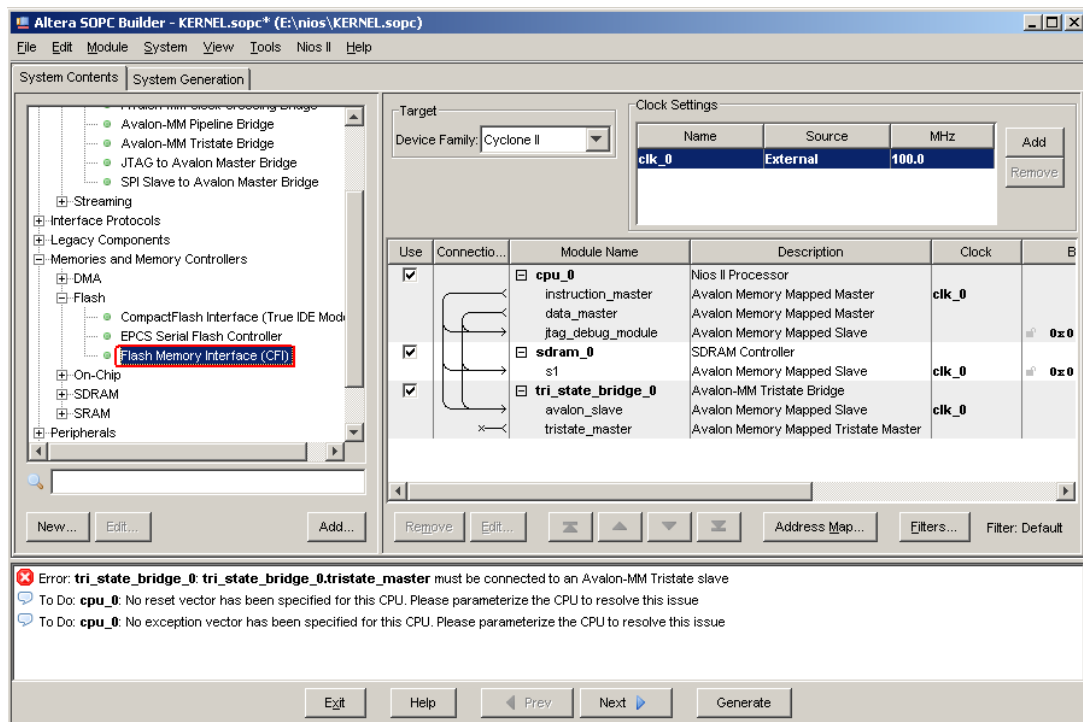


点击后，如下图所示，点击 Finish 后，完成三态桥的建立。

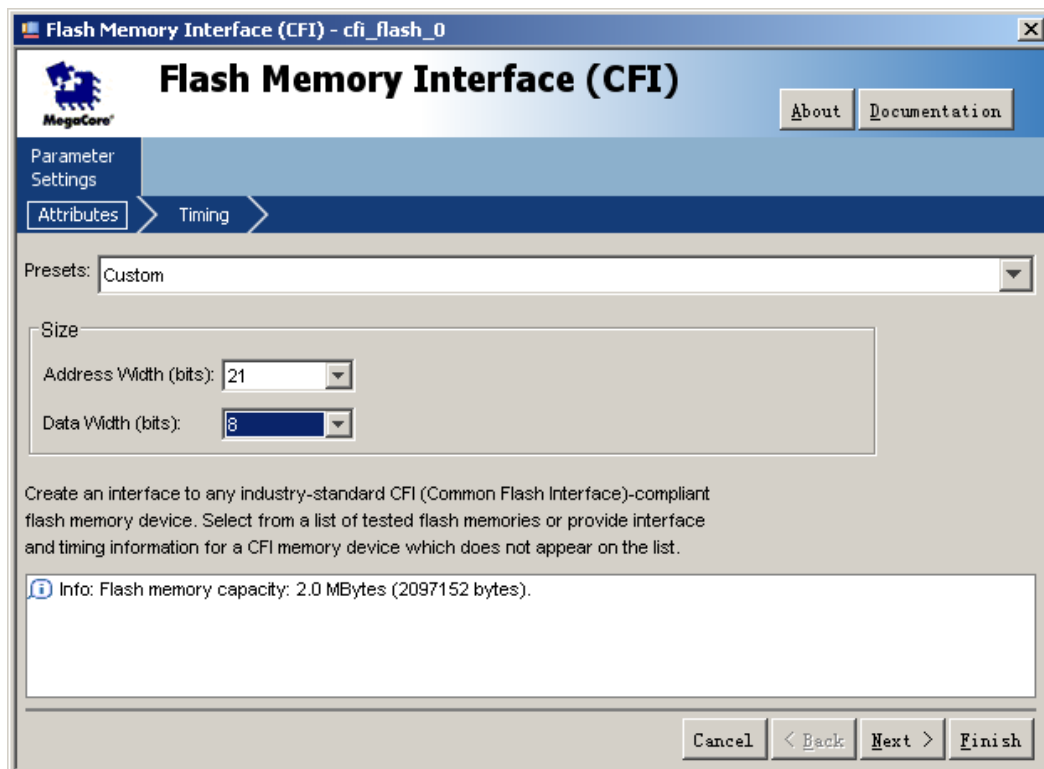


4. 建立 CFI 模块

接下来，我们就建立 Flash Memory Interface(CFI)模块，点击下图所示红圈处

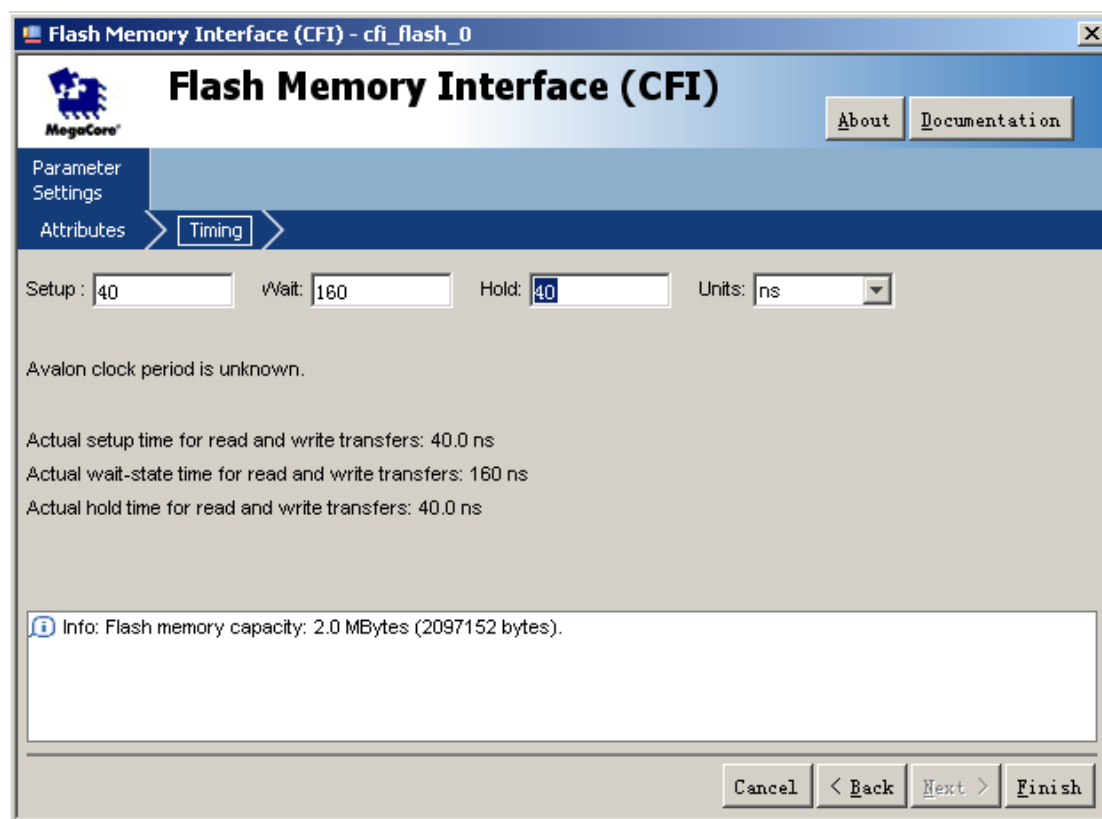


点击后，如下图所示



其中 Presets ,我们选择 Custom ,Address Width(bits)是地址线宽度 ,我们选择 21 ,

Data Width(bits)数据线宽度，我们选择 8 位。这些的选择都是根据芯片和电路设置决定的，我们的 FLASH (AM29LV1602) 可以选择为 8 位模式和 16 位模式，我在设计电路的时候将其配置为 8 位模式，所以在此数据线宽度选择为 6 位，地址线宽度选择为 21 位。设置好以后，点击 Next，出现下图所示

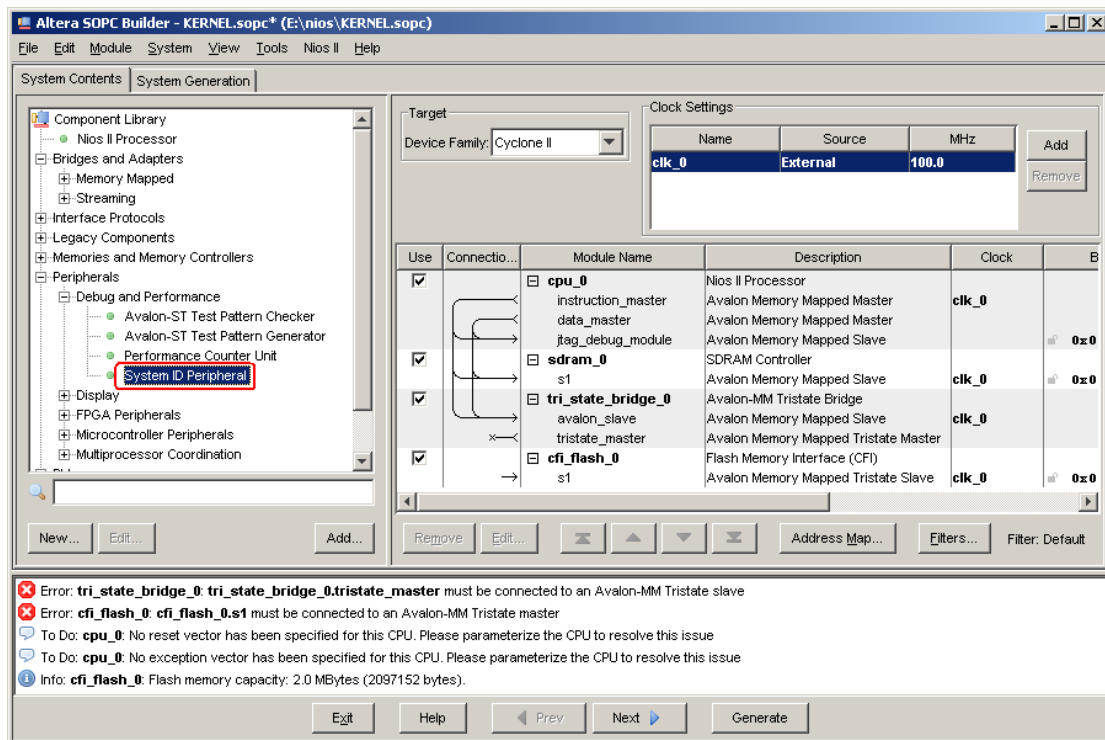


这里，我们需要设置三个量，Setup，Wait，Hold，我们分别将其设置为 40，160，40 如上图所示。这些量都是根据 FLASH 的芯片决定的，各个芯片都不一样，在此不具体讲了。点击 Finish 后，完成 Flash Memory Interface(CFI)的建立。

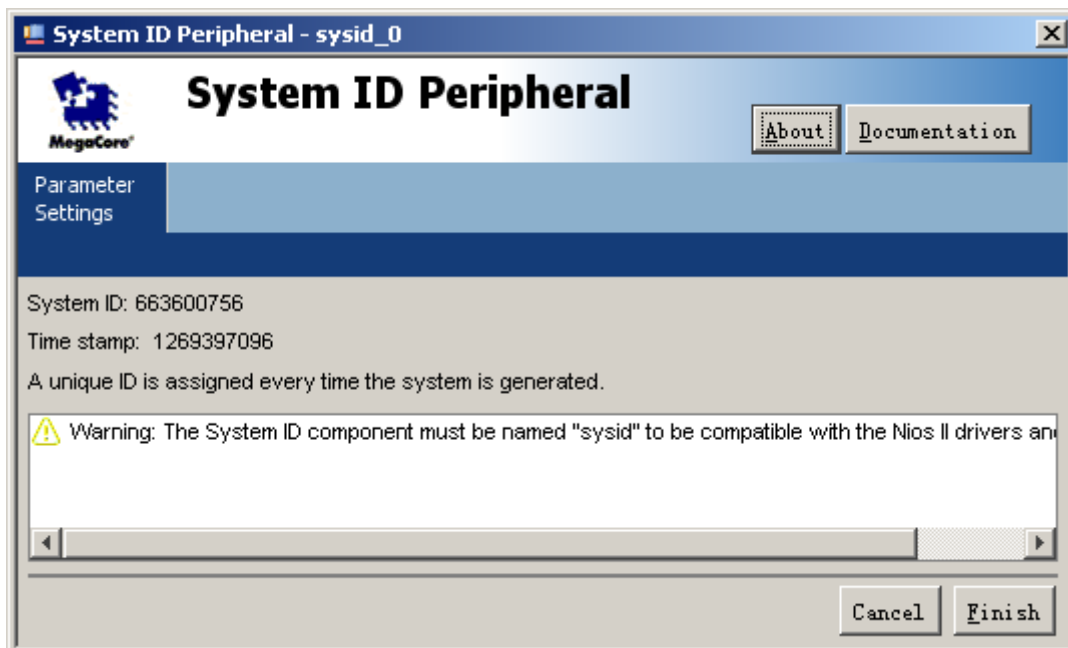
到此为止，我们已经建立了 CPU,SDRAM,FLASH 模块，接下来，我们还要添加一些必要的模块，System ID，JTAG UART。

5. 建立 SYSTEM ID

System ID 就是一种标示符，类似校验和的这么个东西，在你下载程序之前或者重启之后，都会对它进行检验，以防止错误发生，大家知道这个东西就可以了，没太大的用处，但需要有的。点击下图所示红圈处



点击后如下图所示

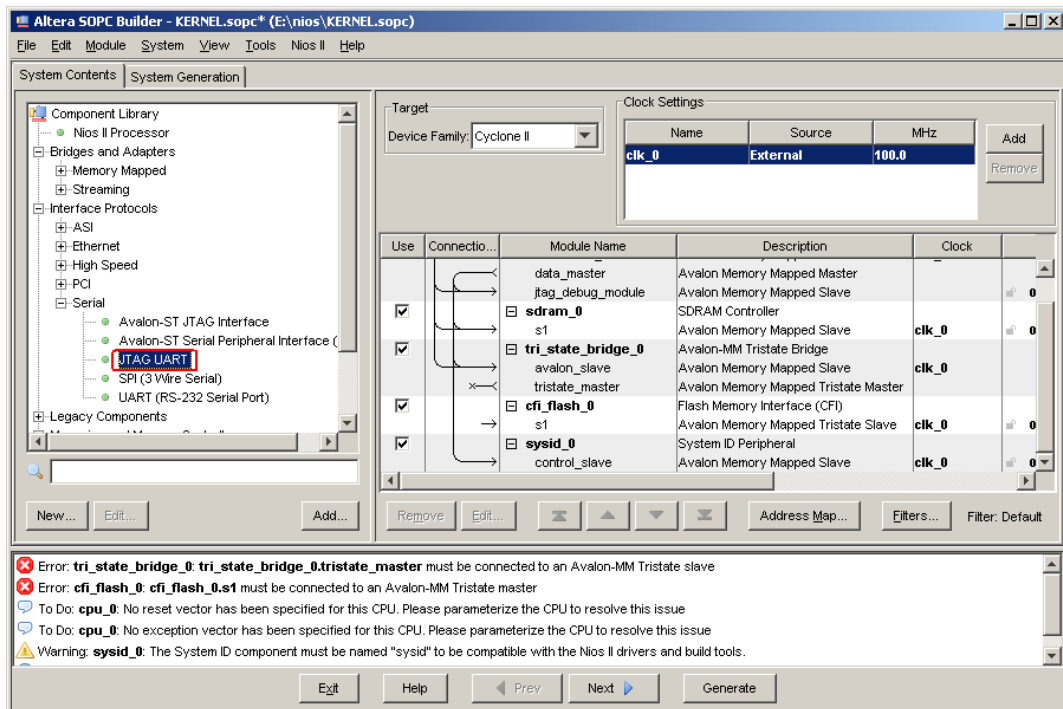


点击 Finish , 完成 System ID 的建立。

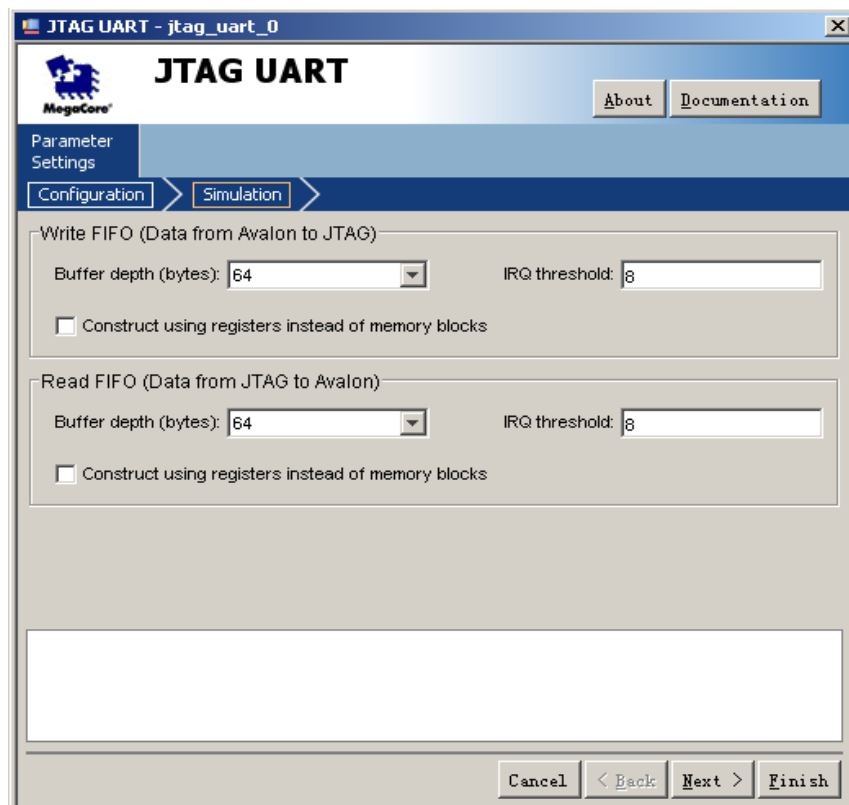
6. 建立 JTAG UART

JTAG UART 是实现 PC 和 Nios II 系统间的串行通信接口 ,它用于字符的输入输出 ,

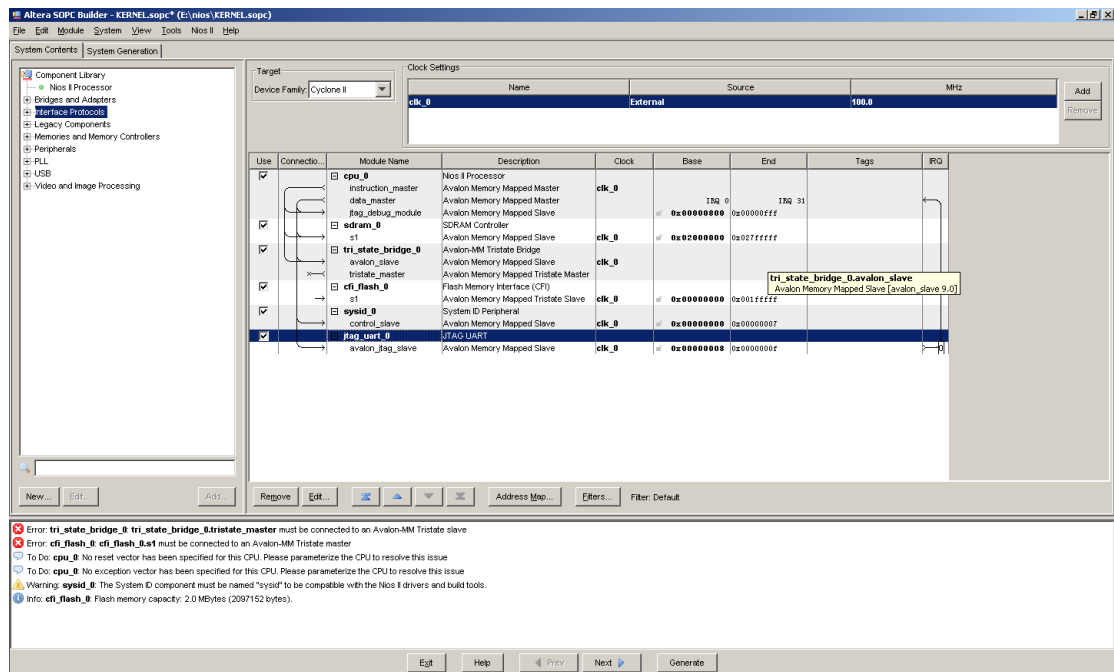
在 Nios II 的开发调试过程中扮演了重要的角色，接下来我们开始建立它的模块。点击下图所示红圈处，



点击后，如下图所示



什么都不用修改，直接点击 Next->Finish 完成 JTAG UART 模块的建立。到此为止，最基本的 NIOS 系统模块就建立完成了，如下图所示



7. 配置及编译 NIOS II

我建议大家将他们的名字都改一下，将_0 都去掉，看着别扭，鼠标右键点击相应名字，Rename 就可以修改了。我们离成功已经很近了，再进行几步处理以后，我们就可以编译了，大家加油吧。大家看下图红圈处

Use	Connecto...	Module Name	Description	Clock	Base	End	Tags	IRQ
☒		cpu_0	Nios II Processor	clk_0				
		instruction_master	Avalon Memory Mapped Master	clk_0		IRQ 0	IRQ 31	
		data_master	Avalon Memory Mapped Master	clk_0		0x00000000	0x00000fff	
☒		sdram_0	SDRAM Controller	clk_0		0x02000000	0x027fffff	
		s1	Avalon Memory Mapped Slave	clk_0		0x00000000	0x001fffff	
☒		tri_state_bridge_0	Avalon-MM Tristate Bridge	clk_0				
		avalon_slave	Avalon Memory Mapped Slave	clk_0				
		tristate_master	Avalon Memory Mapped Tristate Master	clk_0				
☒		cfi_flash_0	Flash Memory Interface (CFI)	clk_0		0x00000000	0x001fffff	
		s1	Avalon Memory Mapped Tristate Slave	clk_0		0x00000000	0x001fffff	
☒		sysid_0	System ID Peripheral	clk_0		0x00000000	0x00000007	
		control_slave	Avalon Memory Mapped Slave	clk_0		0x00000000	0x00000007	
☒		jtag_uart_0	JTAG UART	clk_0		0x00000000	0x0000000f	
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_0		0x00000000	0x0000000f	

cfi_flash 还没有跟 tristate_master 连接，当我们把鼠标移到红圈处位置时，就出现了下图所示的情况

Use	Connectio...	Module Name	Description	Clock	Base	End	Tags	IRQ
☒		☒ cpu	Nios II Processor					
		instruction_master	Avalon Memory Mapped Master	clk_0				
		data_master	Avalon Memory Mapped Master	clk_0				
		jtag_debug_module	Avalon Memory Mapped Slave	clk_0	0x00000800	0x00000fff		
☒		☒ sdr	SDRAM Controller					
		s1	Avalon Memory Mapped Slave	clk_0	0x02000000	0x027fffff		
☒		☒ tri_state_bridge	Avalon-MM Tristate Bridge					
		avalon_slave	Avalon Memory Mapped Slave	clk_0				
		tristate_master	Avalon Memory Mapped Tristate Master	clk_0				
☒		☒ cfi_flash	Flash Memory Interface (CFI)					
		s1	Avalon Memory Mapped Tristate Slave	clk_0	0x00000000	0x001fffff		
☒		☒ tri_state_bridge.tristate_master	System ID Peripheral					
		control_slave	Avalon Memory Mapped Slave	clk_0	0x00000000	0x00000007		
☒		☒ jtag_uart	JTAG UART					
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_0	0x00000008	0x0000000f		

这时候，我们用鼠标点击红圈处的空心圆，这时候奇迹出现了，呵呵，如下图所示

Use	Connectio...	Module Name	Description	Clock	Base	End	Tags	IRQ
☒		☒ cpu	Nios II Processor					
		instruction_master	Avalon Memory Mapped Master	clk_0				
		data_master	Avalon Memory Mapped Master	clk_0				
		jtag_debug_module	Avalon Memory Mapped Slave	clk_0	0x00000800	0x00000fff		
☒		☒ sdr	SDRAM Controller					
		s1	Avalon Memory Mapped Slave	clk_0	0x02000000	0x027fffff		
☒		☒ tri_state_bridge	Avalon-MM Tristate Bridge					
		avalon_slave	Avalon Memory Mapped Slave	clk_0				
		tristate_master	Avalon Memory Mapped Tristate Master	clk_0				
☒		☒ cfi_flash	Flash Memory Interface (CFI)					
		s1	Avalon Memory Mapped Tristate Slave	clk_0	0x00000000	0x001fffff		
☒		☒ sysid	System ID Peripheral					
		control_slave	Avalon Memory Mapped Slave	clk_0	0x00000000	0x00000007		
☒		☒ jtag_uart	JTAG UART					
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_0	0x00000008	0x0000000f		

空心圆变成了实心圆，这就完成了 cfi_flash 与三态桥的连接，大家亲手试试吧。

接下来我们需要对 cpu 进行设置一下，双击 cpu，Reset Vector 处的 Memeory 选择 cfi_flash，Exceptioni Vector 选择 sdr，其他不变，如下图所示

Nios II Processor

Parameter Settings

Core Nios II

Caches and Memory Interfaces

Advanced Features

MMU and MPU Settings

JTAG Debug Module

Custom Instructions

Core Nios II

Select a Nios II core:

Nios II/e

RISC 32-bit

Selector Guide

Family: Cyclone II

f_{system}: 100.0 MHz

cpu_{id}: 0

Performance at 100.0 MHz: Up to 9 DMIPS

Logic Usage: 600-700 LEs

Nios II/s

RISC 32-bit

Instruction Cache

Branch Prediction

Hardware Multiply

Hardware Divide

Up to 50 DMIPS

1200-1400 LEs

Nios II/f

RISC 32-bit

Instruction Cache

Branch Prediction

Hardware Multiply

Hardware Divide

Barrel Shifter

Data Cache

Dynamic Branch Prediction

Up to 101 DMIPS

1400-1800 LEs

Hardware Multiply: Embedded Multipliers ☐ Hardware Divide

Reset Vector: Memory: cfi_flash Offset: 0x0 0x00000000

Exception Vector: Memory: sdr Offset: 0x20 0x02000020

☐ Include MMU

Only include the MMU when using an operating system that explicitly supports an MMU

Fast TLB Miss Exception Vector: Memory: Offset: 0x0

☐ Include MPU

Cancel

< Back

Next >

Finish

点击 Finish，完成 cpu 设置。

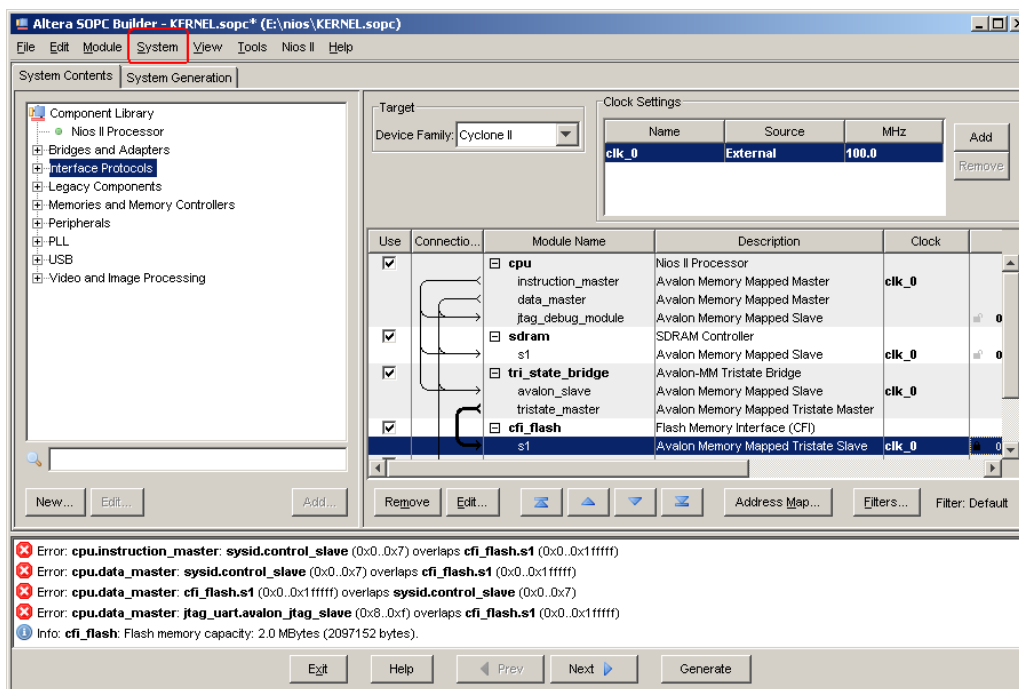
接下来，我们需要对 FLASH 地址进行锁定，保证 FLASH 的起始地址为 0x00000000，因为 FLASH 是系统重启后的起始位置，这样做的好处就是有利于我们操作，系统重启后从 0x00000000 开始也是我们的思维习惯。点击下图所示的红圈处

Use	Connectio...	Module Name	Description	Clock	Base	End	Tags	IRQ
☒		☐ cpu	Nios II Processor	clk_0				
		instruction_master	Avalon Memory Mapped Master					
		data_master	Avalon Memory Mapped Master					
		jtag_debug_module	Avalon Memory Mapped Slave					
☒		☐ sdram	SDRAM Controller					
		s1	Avalon Memory Mapped Slave	clk_0	0x02000000	0x027fffff		
☒		☐ tri_state_bridge	Avalon-MM Tristate Bridge					
		avalon_slave	Avalon Memory Mapped Slave	clk_0				
		tristate_master	Avalon Memory Mapped Tristate Master					
☒		☐ cfi_flash	Flash Memory Interface (CFI)					
		s1	Avalon Memory Mapped Tristate Slave	clk_0	0x00000000	0x001fffff		
☒		☐ sysid	System ID Peripheral					
		control_slave	Avalon Memory Mapped Slave	clk_0	0x00000000	0x00000007		
☒		☐ jtag_uart	JTAG UART					
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_0	0x00000008	0x0000000f		

点击后，奇迹又出现了，看看下图吧，开着的锁合上了，ALTERA 还真逗啊。

Use	Connectio...	Module Name	Description	Clock	Base	End	Tags	IRQ
☒		☐ cpu	Nios II Processor	clk_0				
		instruction_master	Avalon Memory Mapped Master					
		data_master	Avalon Memory Mapped Master					
		jtag_debug_module	Avalon Memory Mapped Slave					
☒		☐ sdram	SDRAM Controller					
		s1	Avalon Memory Mapped Slave	clk_0	0x02000000	0x027fffff		
☒		☐ tri_state_bridge	Avalon-MM Tristate Bridge					
		avalon_slave	Avalon Memory Mapped Slave	clk_0				
		tristate_master	Avalon Memory Mapped Tristate Master					
☒		☐ cfi_flash	Flash Memory Interface (CFI)					
		s1	Avalon Memory Mapped Tristate Slave	clk_0	0x00000000	0x001fffff		
☒		☐ sysid	System ID Peripheral					
		control_slave	Avalon Memory Mapped Slave	clk_0	0x00000000	0x00000007		
☒		☐ jtag_uart	JTAG UART					
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_0	0x00000008	0x0000000f		

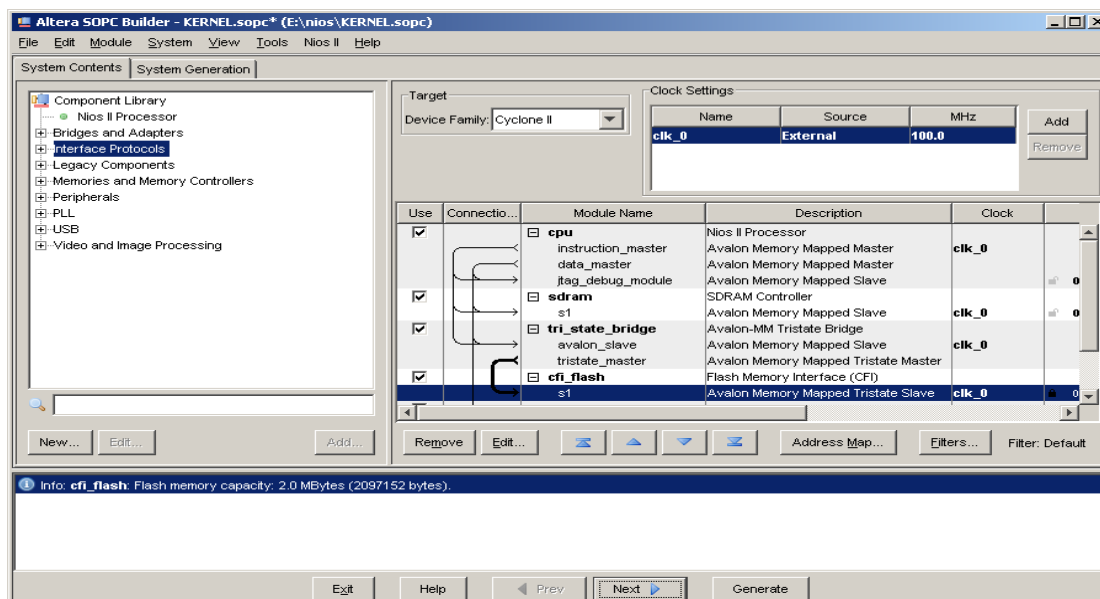
最后一步设置就是对地址自动分配，这样做是为了不浪费空间，如果有特殊要求的，也可以手动设置，点击相应的地址，就可以手动修改了，大家自己试试吧。地址自动分配操作如下，点击下图所示红圈处，SYSTEM->Auto-assign Basic Addresses



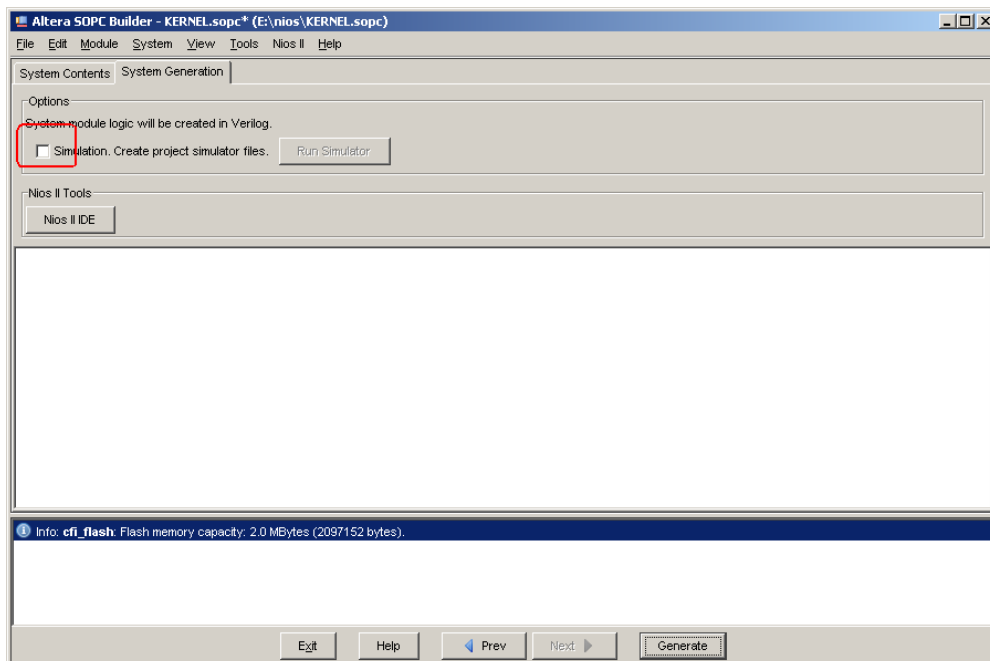
点击后，大家可以发现，各个模块的地址都有相应的变化，但 CFI_FLASH 基地址还是 0x00000000，锁定的还算成功啊，呵呵！

接下来是中断的自动分配，和地址自动分配一样，SYSTEM-> Auto-assign IRQs，如果有什么特殊需要，可以手动更改中断的优先级。之所以要自动分配一下，是因为这个软件还不够智能，当模块建好以后，有出现重复中断号的现象，编译的过程就会出现问題，自动分配了以后就不从上自下按顺序排列了。你还会发现，图片上出现的红叉也都消失了。

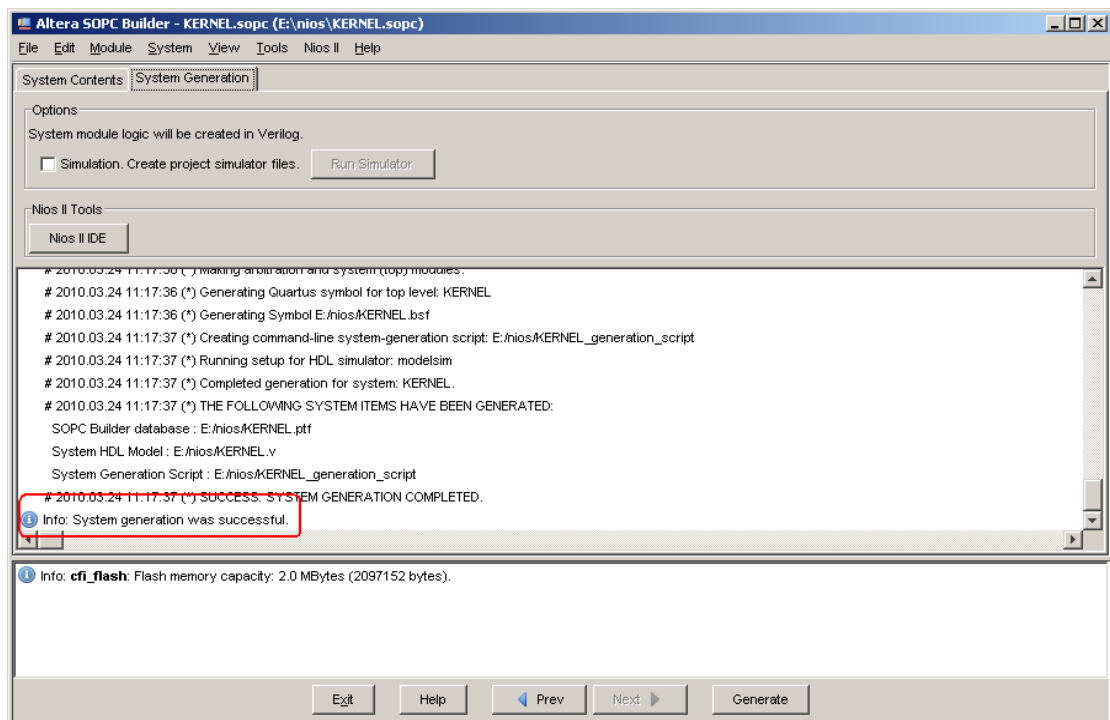
展现我们的劳动成果最后的样子，红叉没有了吧，这时候我们就可以编译了。



点击 Next , 出现下图 , 如果需要仿真的 , 点击红圈处 , 将其选中 , 我一般不进行仿真 , 此处就不选了。

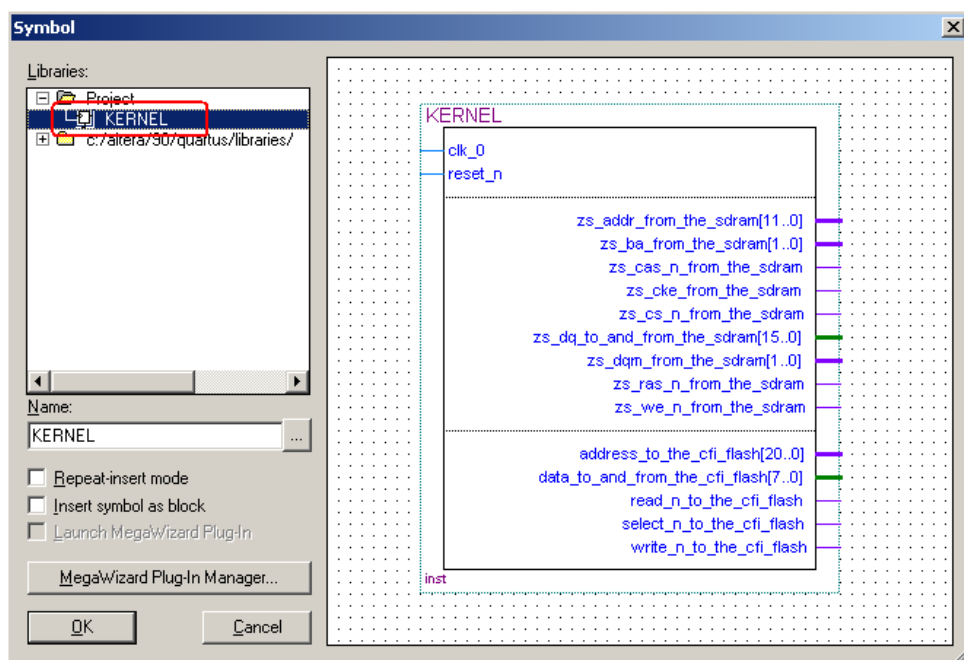


点击 Generate , 我们就开始编译了。想休息的赶紧休息一下 , 想去 WC 也快去吧 , 时间还充裕 , 呵呵.....经过漫长的等待以后 , 我们的程序终于编译好 , 请看下图红圈处 , 看到 successful 了么 ? 只要看到他就没问题了 , 软核已经好编译好了 , 点击 Exit , 回到了 Quartus 界面。

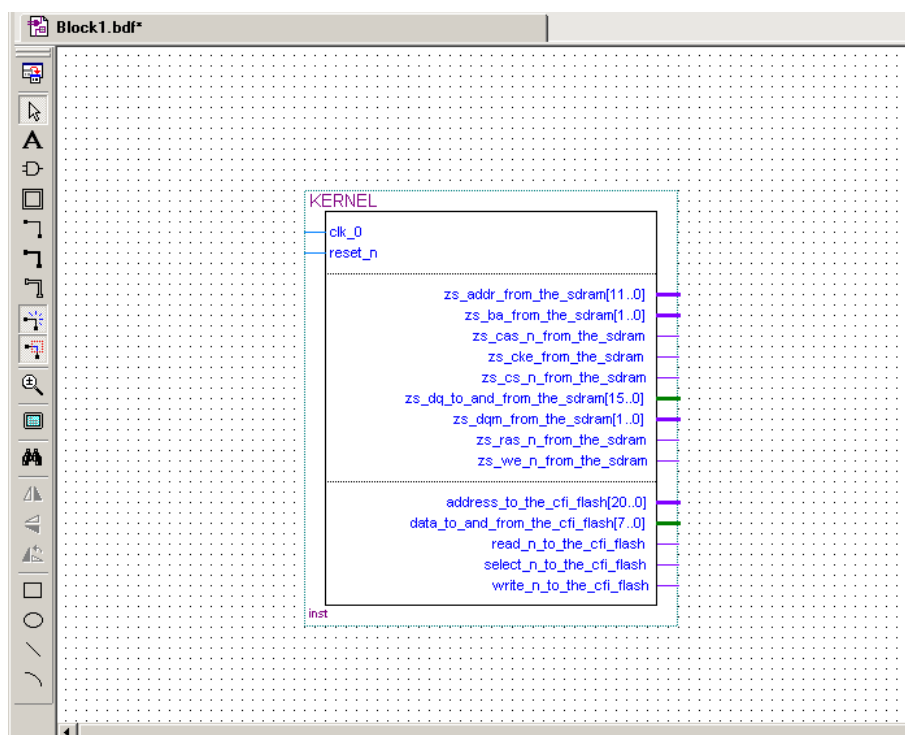


8. 分配管脚

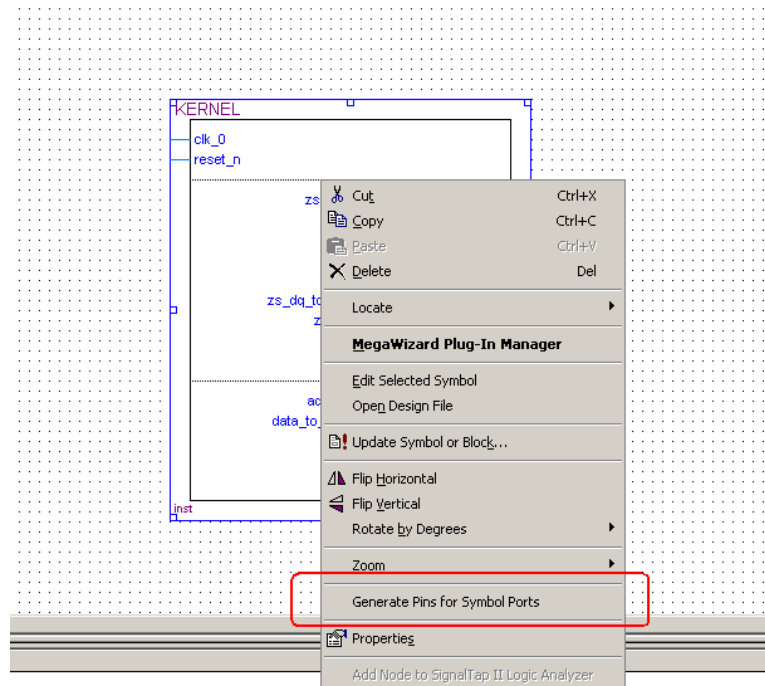
回到 Quartus 界面以后，在 Block1.bdf 界面里在空白处双击左键，会出现下图，看多了下图红圈处的地方了么，这个就是我们做好的 NIOS II 软核了，用鼠标双击它（或者选中以后点击 OK）后，将其放到 Block.bdf 的空白处。



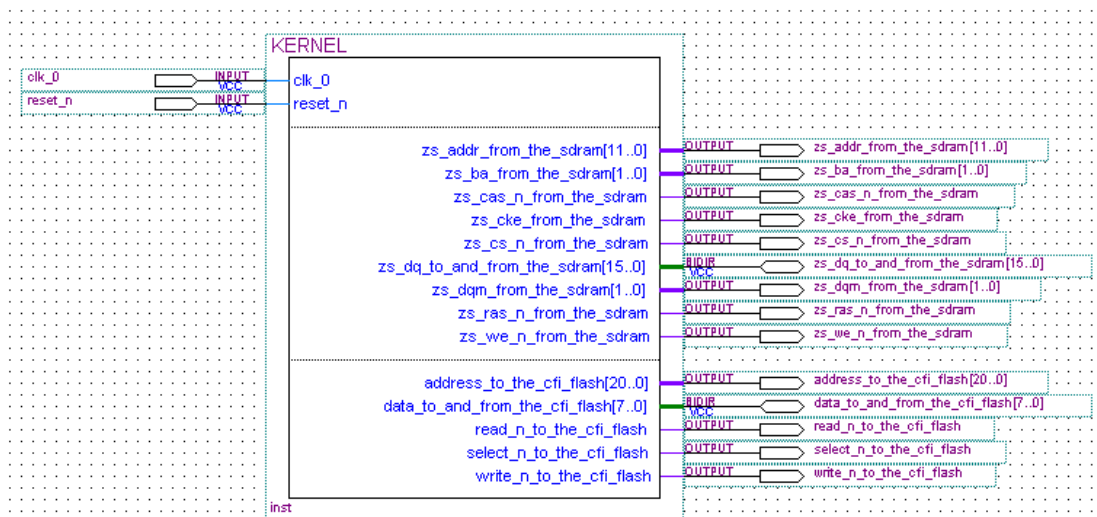
放好以后，如下图所示



在 NIOS 软核 KERNEL 上点击右键后点击 Generate Pins for symbol ports，这一步作用就是生成管脚，通过命名以后分配真是的引脚。



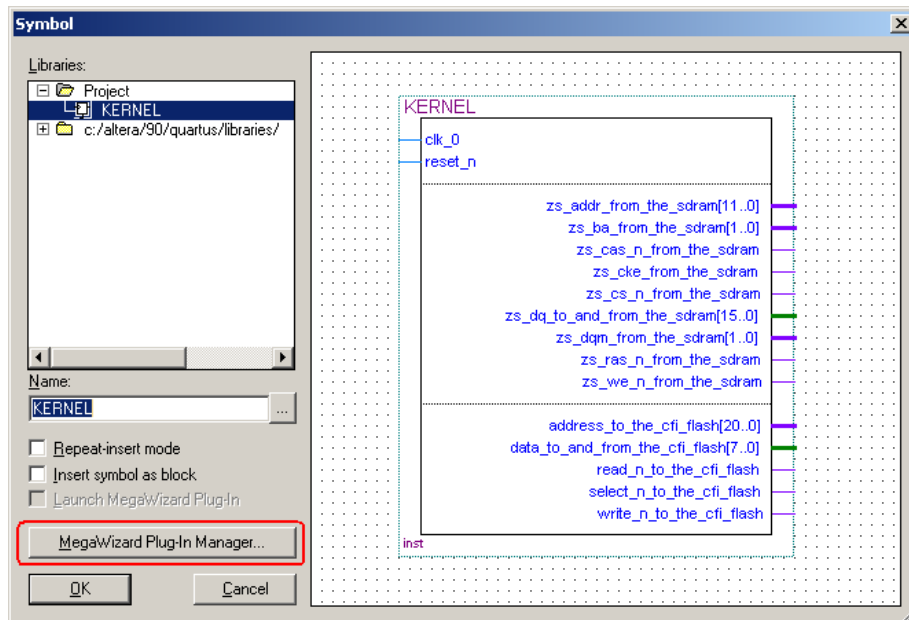
点击后，如下图所示，



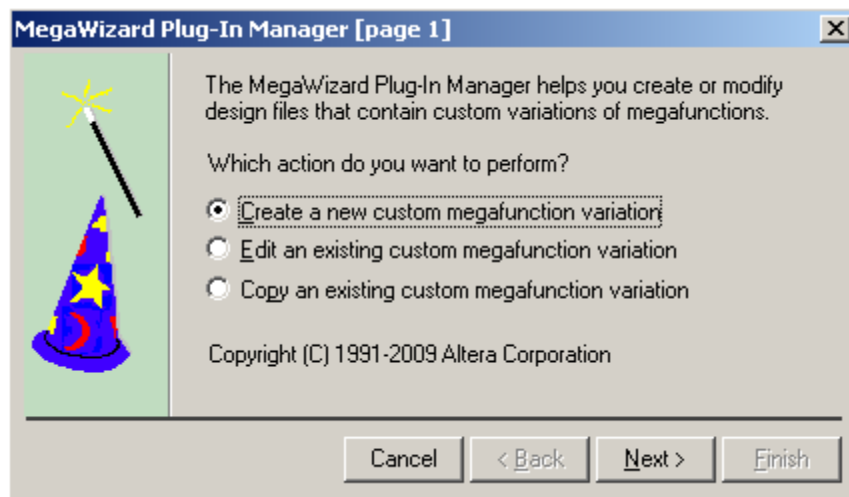
四、 建立锁相环 PLL 模块

接下来的工作我们还需要建立一个锁相环,对时钟进行倍频,我们板子上是 20MHz 的有源晶振,我们要将其倍频到 100MHz 满足我们之前设定的 NOIS 软核的时钟,还需要为 SDRAM 提供 100MHz 的时钟。下面我们就开始锁相环 PLL 模块。

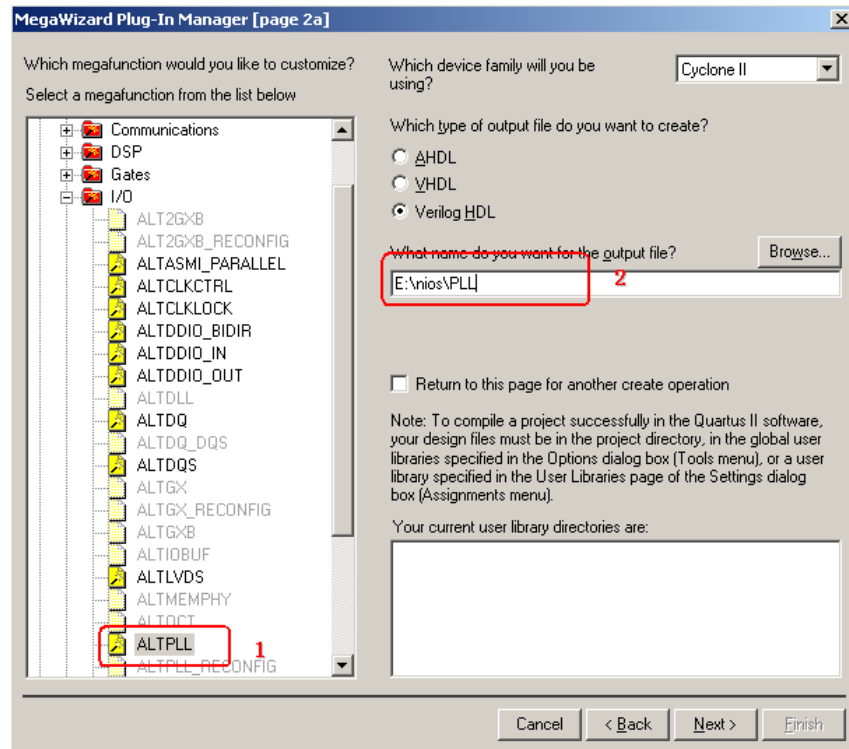
在 Block.bdf 的空白处双击鼠标, 点击下图所示的红圈处



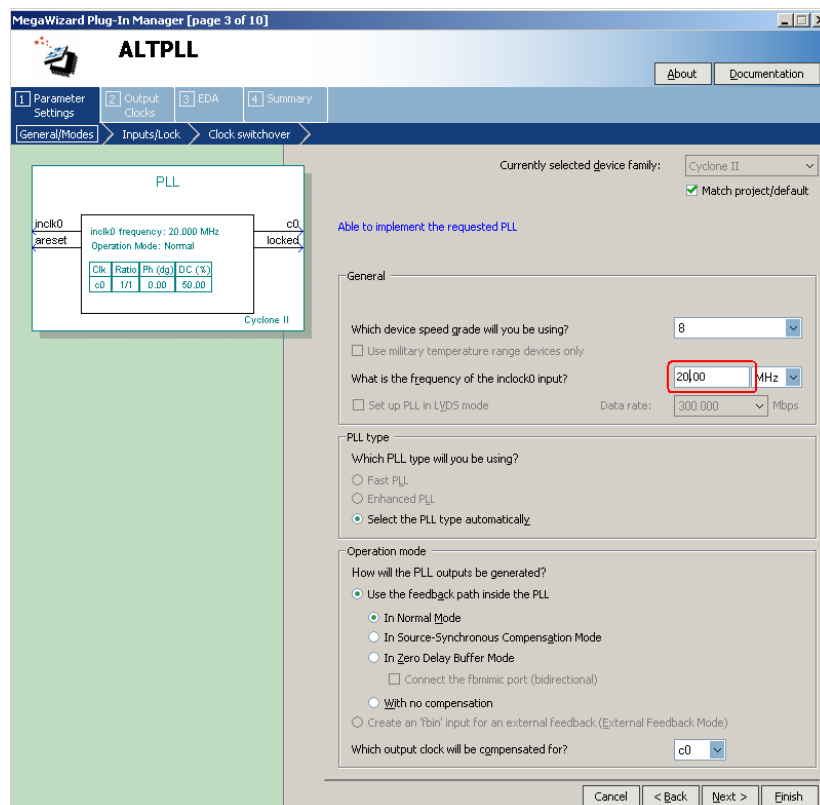
出现下图后点击 Next



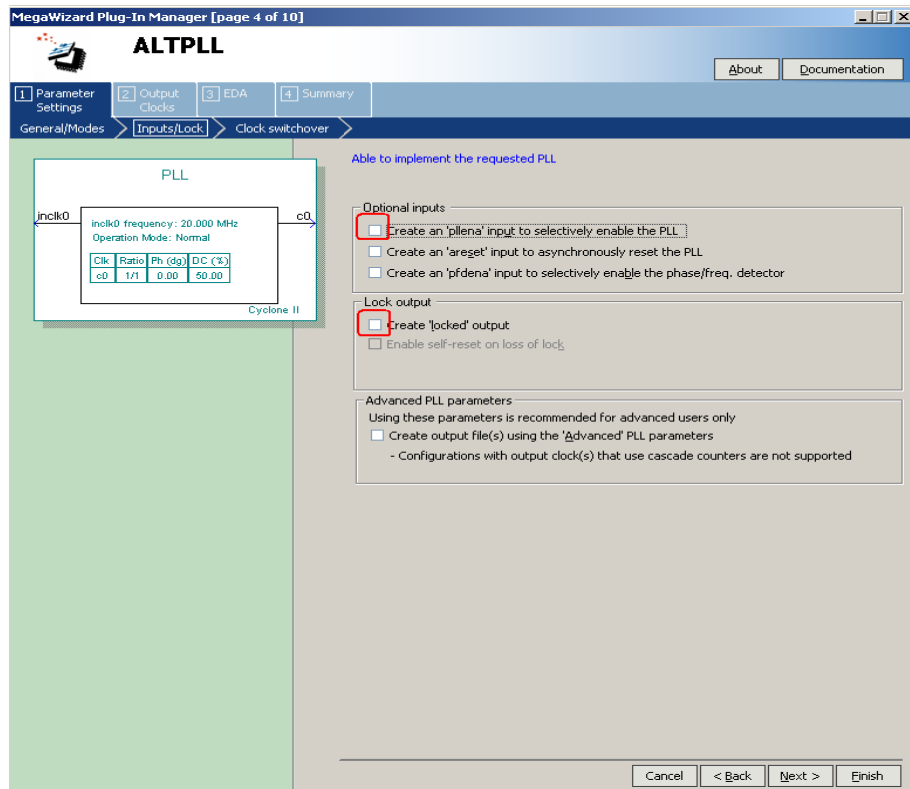
出现下图所示,用鼠标选中红圈 1 所在的位置 ALTPLL,在红圈 2 所在的位置处 加上 PLL , 目的是将我们所要建立的锁相环命名为 PLL。



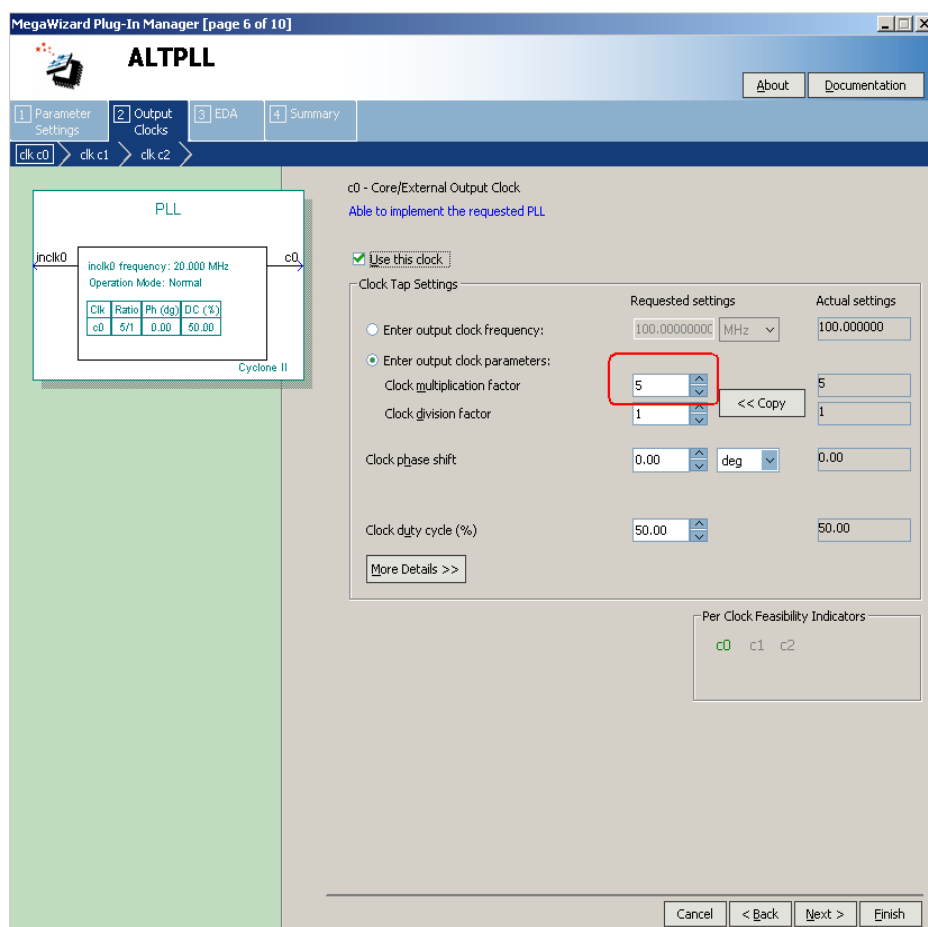
一切弄好以后，点击 Next，如下图所示，将红圈处设置为 20，即我们输入的晶振为 20MHz



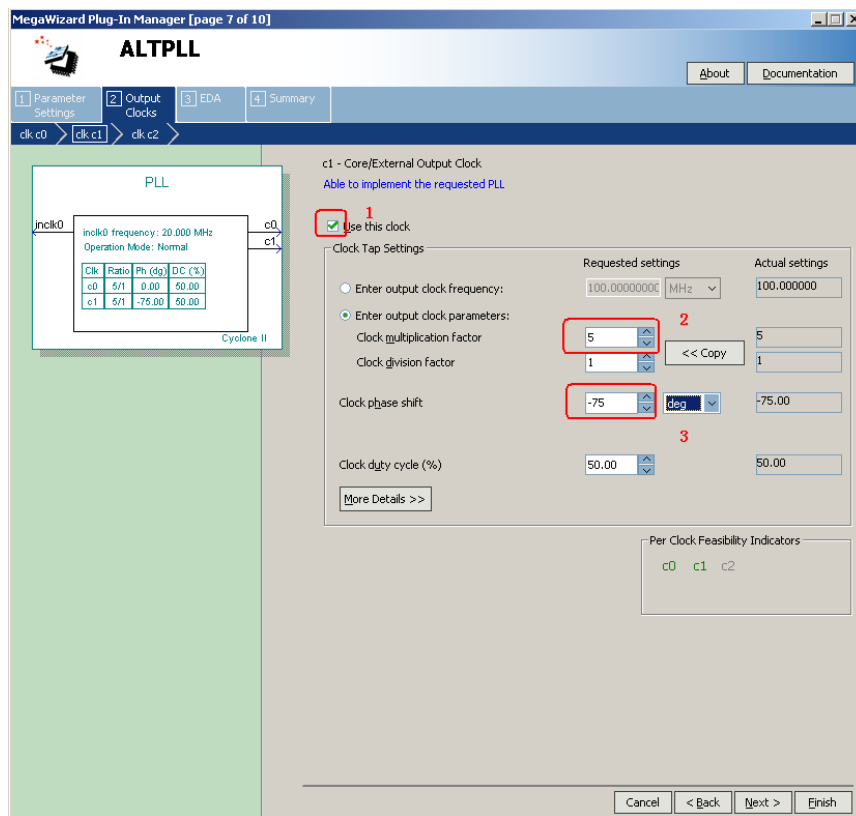
设置好以后，点击 Next，将下图中选中的地方去掉后，



连续点击 Next，等出现下图以后，将红圈处设置为 5，这是 c0 输出的频率就设置为了 100MHz，可以从 Actual settings 处看出来。

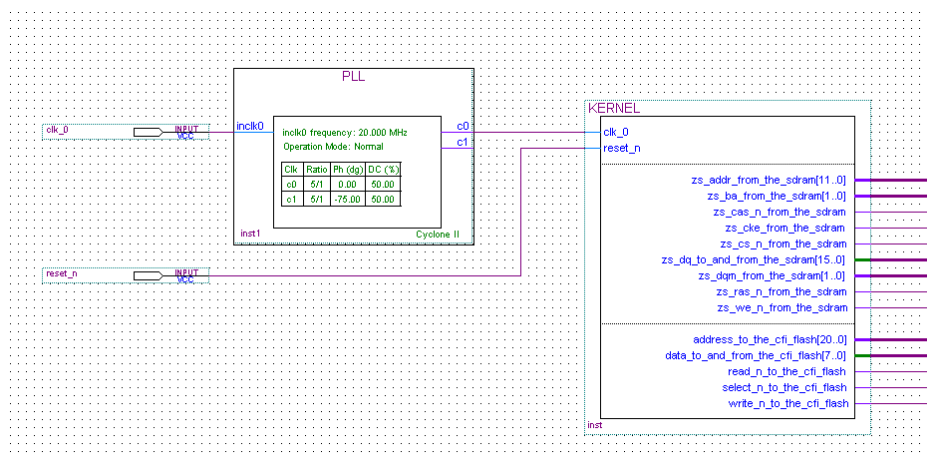


设置好以后点击 Next，出现下图，将红圈 1 处选中，红圈 2 处修改为 5，红圈 3 处修改为-75。这部分是给 SDRAM 提供时钟，我们利用锁相环 PLL 的 c1 来提供，时钟频率设置为 100MHz，偏移量为-75deg（这个地方影响到 SDRAM 是否正常运行，我们将在以后具体讲述他的设置，在这里不详述了）。

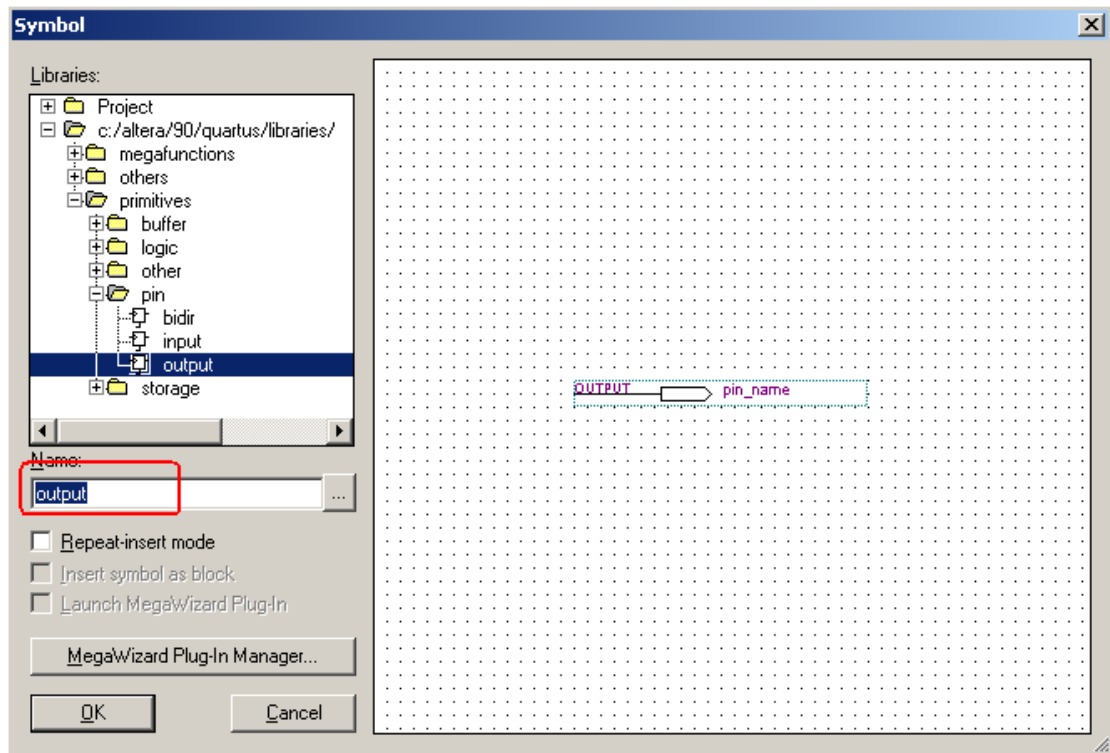


设置好以后，连续点击 Next，中间没有需要修改的，最后点击 Finish，完成 PLL 的建立，然后点击 OK，回到 Quartus 界面，将我们建好的 PLL 放到空白处。

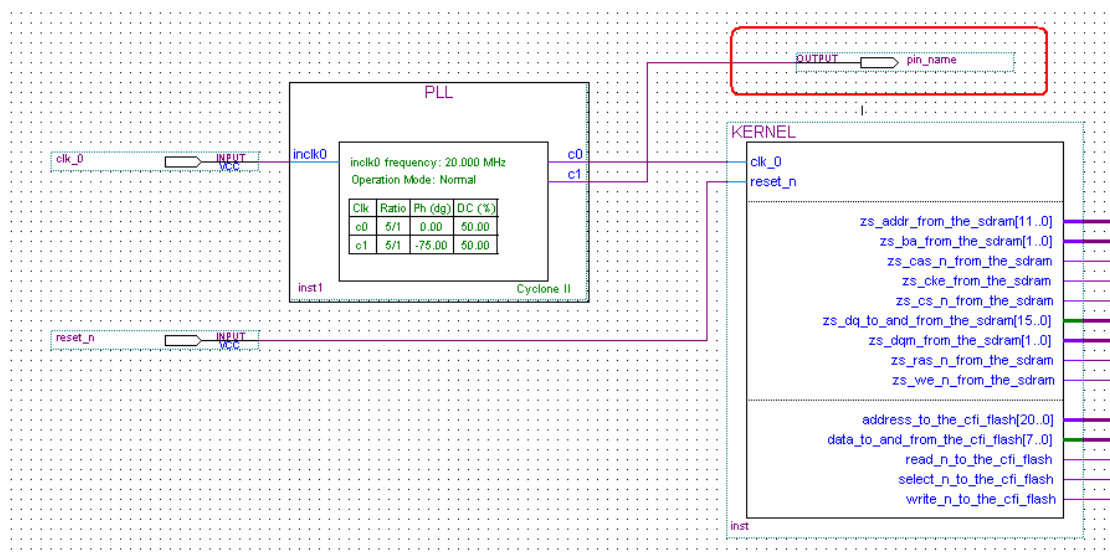
接下来的工作就是将 PLL 连接到 NIOS 软核上。如下图所示（连线方式很简单，大家自己研究一下，我就不具体说了）



接下来我们要给 SDRAM 的时钟分配引脚,这个就由 c1 完成了。首先是加一个 output 引脚,方法很简单,在空白处双击,出现下图,在 Name 处输入 output,一个回车,OK 了。假如你要加入 input,即输入引脚,你就在 Name 处输入 input,还有双向引脚的是 bidir,常用的就这几个,大家记住了就可以了,很简单。

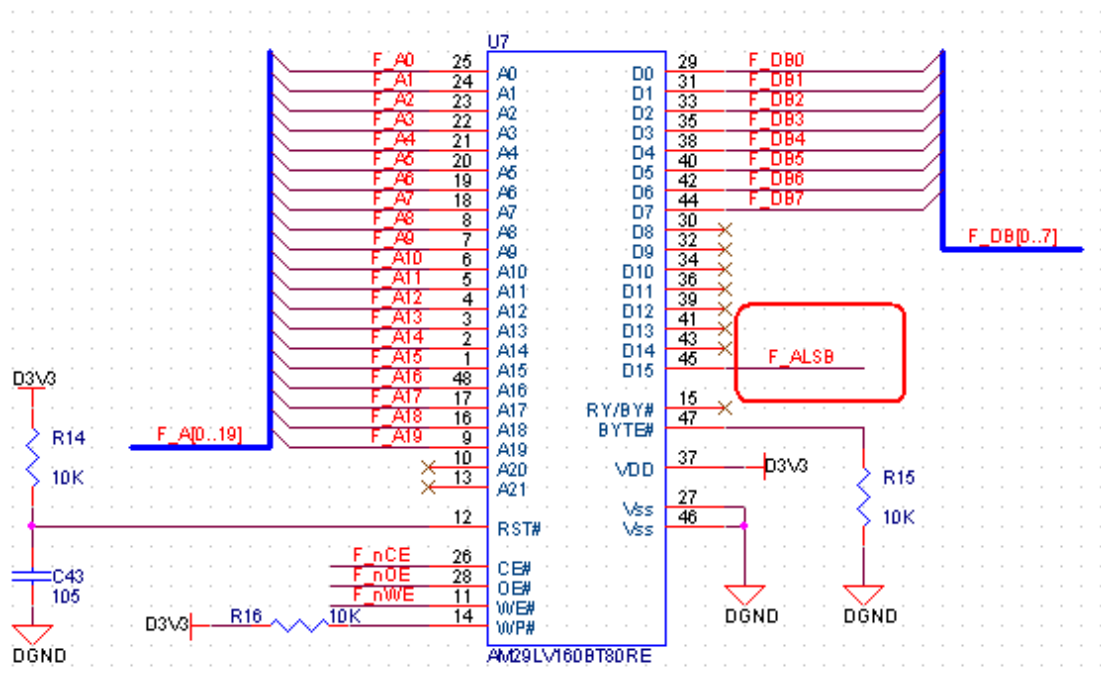


将其放到下图红框所示位置,连好线就 OK 了

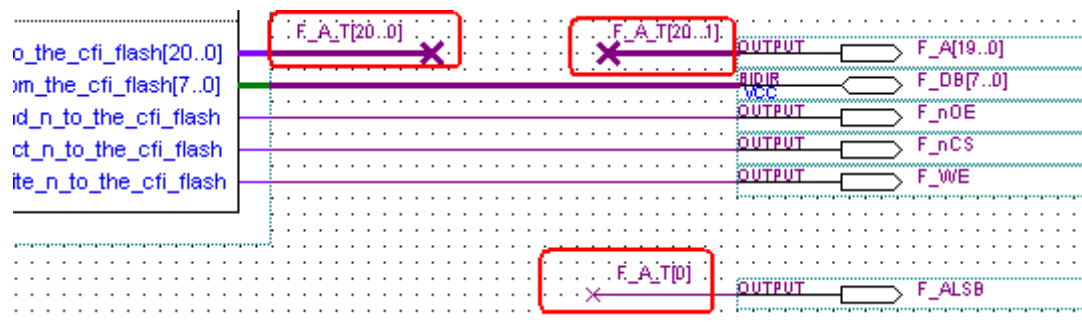


五、 调整 FLASH 引脚

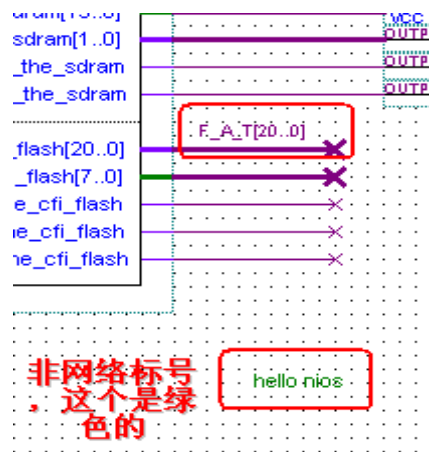
接下来，我们要对 Flash 连接做一下处理，因为是 8 位模式的解法，所以 FLASH 的地址线 F_A_T[0] 的最低位要跟 F_ALSB 相连，如下图所示红圈处，(下图为 FLASH 的原理图，8 位模式的时候，BYTE# 引脚下拉，16 位模式的时候则需要上拉，请大家注意)



那么，我们就来修改一下，修改结果如下图所示



这里，需要注意红圈处的改动，将软核输出命名为 F_A_T[20..0]，F_A[19..0] 连接的是 F_A_T[20..1]，F_ALSB 连接的是 F_A_T[0]，不知大家是否能看的懂。这个地方命名方式是这样的，点击想命名的线，当被选中后，直接输入你想输入的网络标号即可，大家自己试试吧，很简单。大家要注意一下，网络标号的颜色跟线的颜色是一样的，而文字的颜色是绿色的，大家要区分开，如下图所示



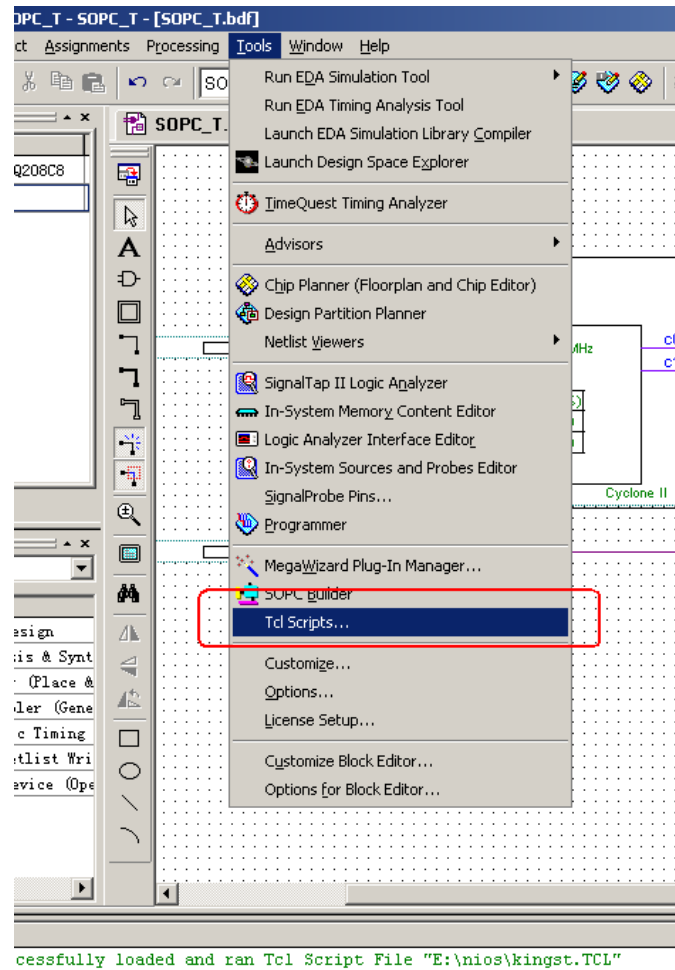
六、TCL 脚本文件

接下来就要开始分配引脚了，分配引脚有两种方式，在这里我只讲一种，也是我觉得比较好的一种，就是通过 TCL 脚本文件，这个文件有固定的写法，只要根据你的引脚情况修改一下就可以了，如下图所示

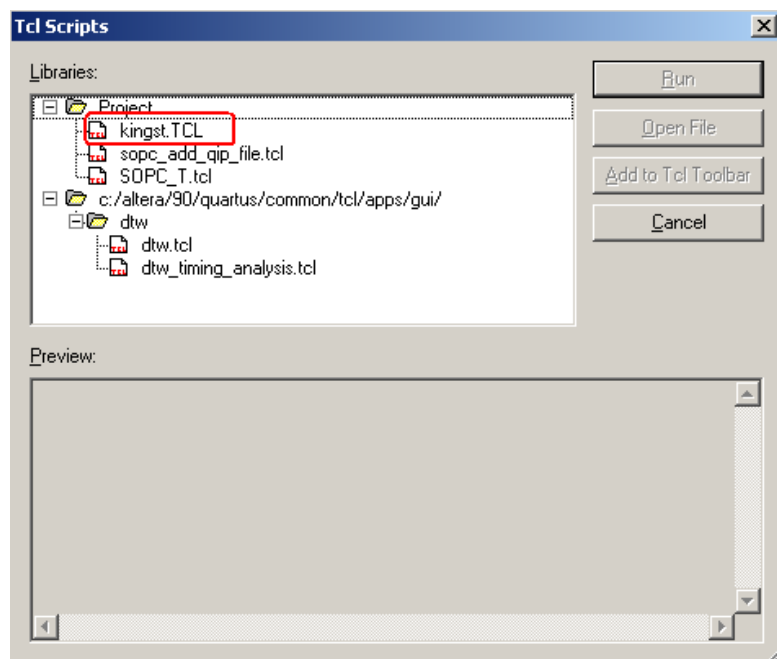
```
#-----SDRAM-----#
set_location_assignment PIN_162 -to S_DB[0]
set_location_assignment PIN_161 -to S_DB[1]
set_location_assignment PIN_160 -to S_DB[2]
set_location_assignment PIN_152 -to S_DB[3]
set_location_assignment PIN_151 -to S_DB[4]
set_location_assignment PIN_150 -to S_DB[5]
set_location_assignment PIN_149 -to S_DB[6]
set_location_assignment PIN_147 -to S_DB[7]
set_location_assignment PIN_179 -to S_DB[8]
set_location_assignment PIN_180 -to S_DB[9]
set_location_assignment PIN_181 -to S_DB[10]
set_location_assignment PIN_185 -to S_DB[11]
set_location_assignment PIN_187 -to S_DB[12]
set_location_assignment PIN_188 -to S_DB[13]
set_location_assignment PIN_189 -to S_DB[14]
set_location_assignment PIN_191 -to S_DB[15]
```

我们要命的名字就是后面那部分，PIN_*是 FPGA 硬件上的引脚，S_DB[*]就是对应的名字。下面我们举例说明一下吧

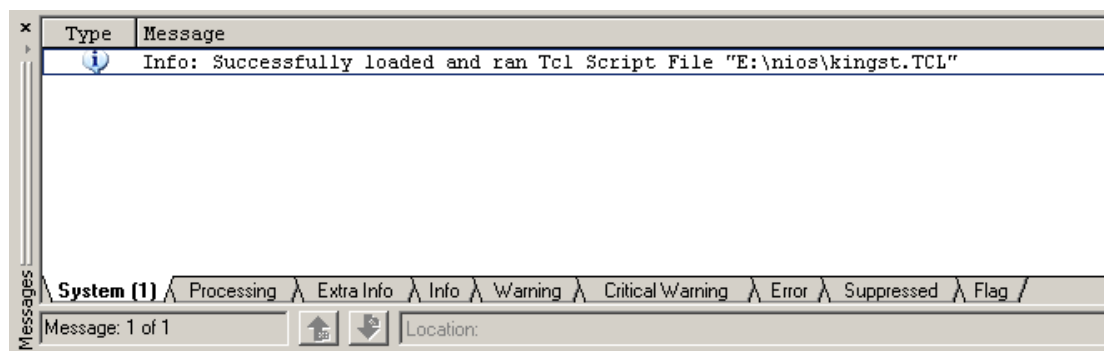
我现在修改时钟管脚，如下图所示，将其命名为 CLOCK，咱们再来看对应的 TCL 脚本文件



点击以后，出现下图，我们选择第一个（前提是你将 kingst.tcl 放到工程文件下）

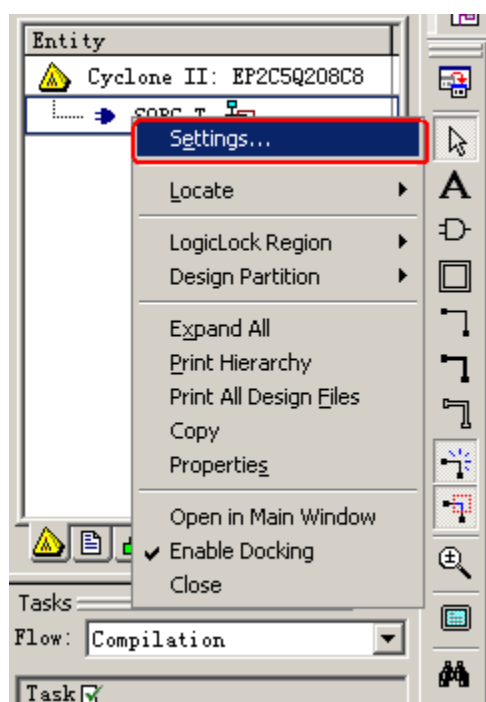


选中后点击 Run，如下所示，又看见 Successfully 了吧，这说明我们脚本文件运行成功了

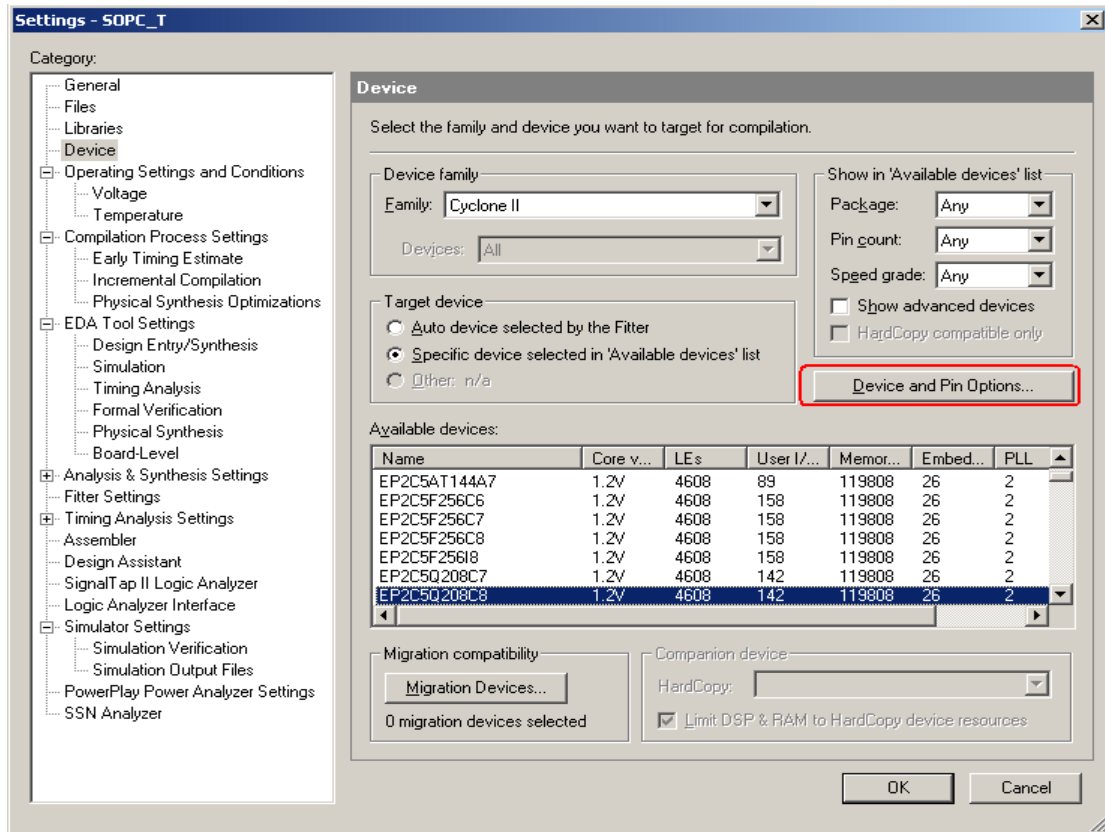


七、配置工程

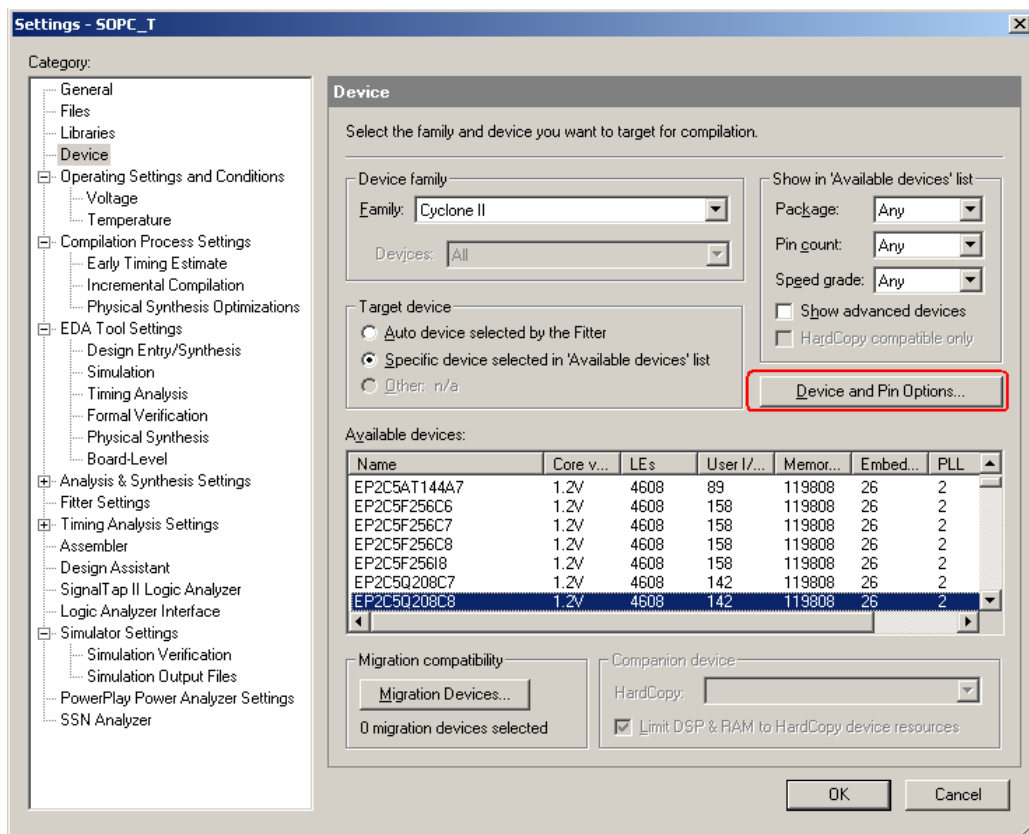
先保存一下吧，保存好以后，接下来，我们要对工程配置一下了，在左侧边框栏右键点击 SOPC_T，如下图所示



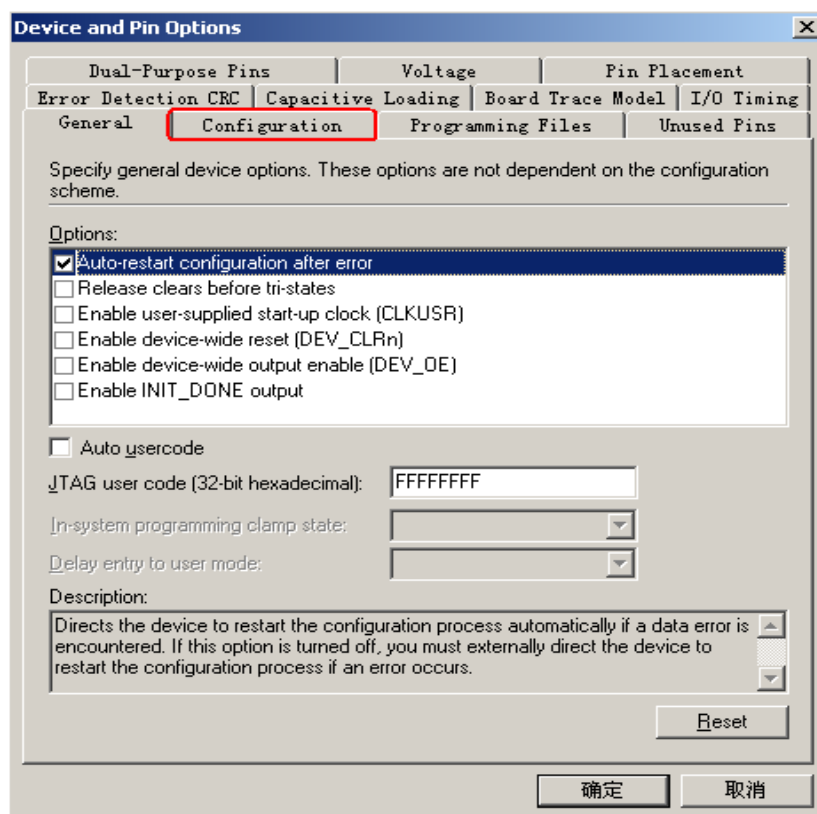
点击 setting 后，如下图所示，点击红圈处 Device and Pin Options



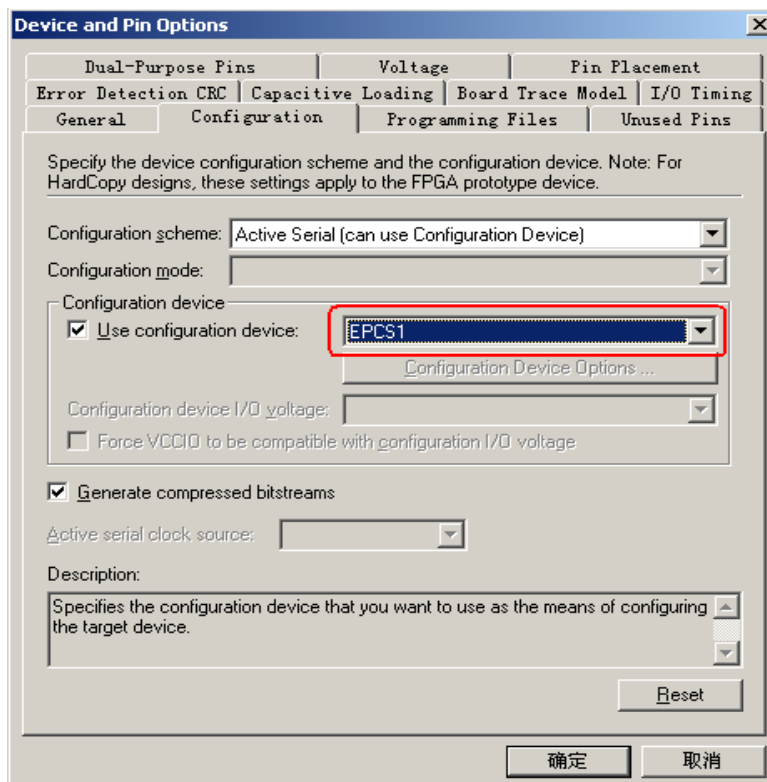
点击后，如下图所示



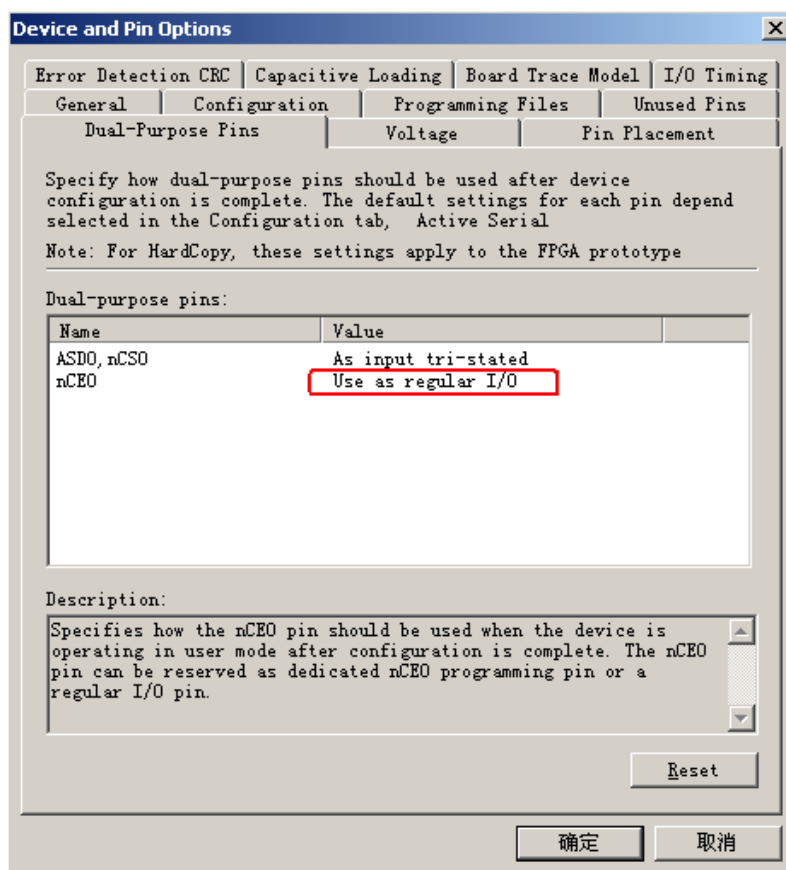
点击后如下图所示，点击红圈处 Configuration



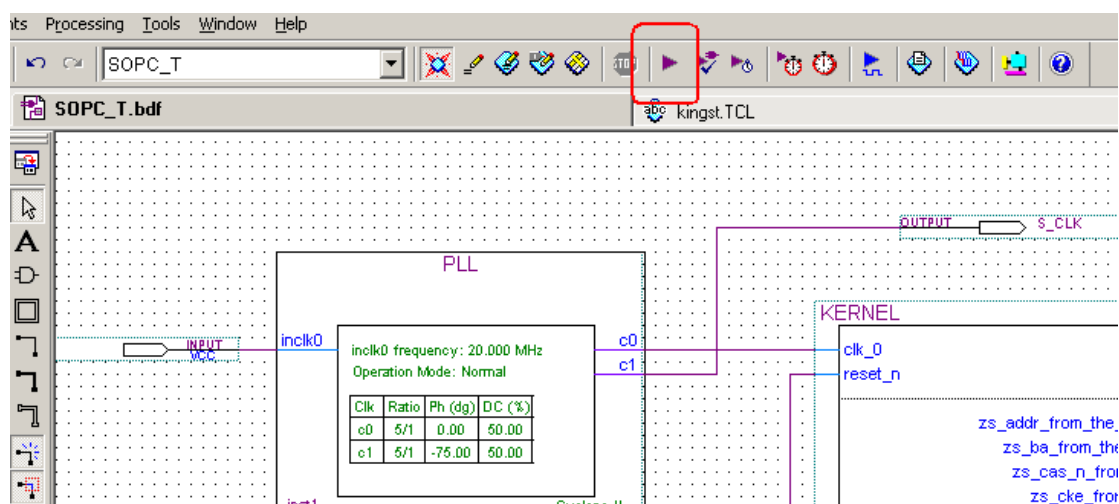
点击后，如下图所示，将红圈处改为 EPCS1，然后点击 Dual-Purpose Pins



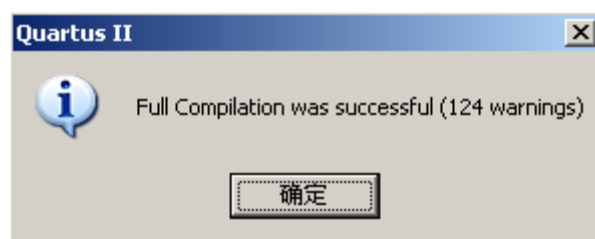
点击后，如下图所示，将红圈处改为 Use as regular I/O



都修改好以后，点击确定，点击 OK。接下来就开始了又一个漫长的编译过程了，大家又可以休息一会了。点击下图所示红圈处的按钮，开始编译。



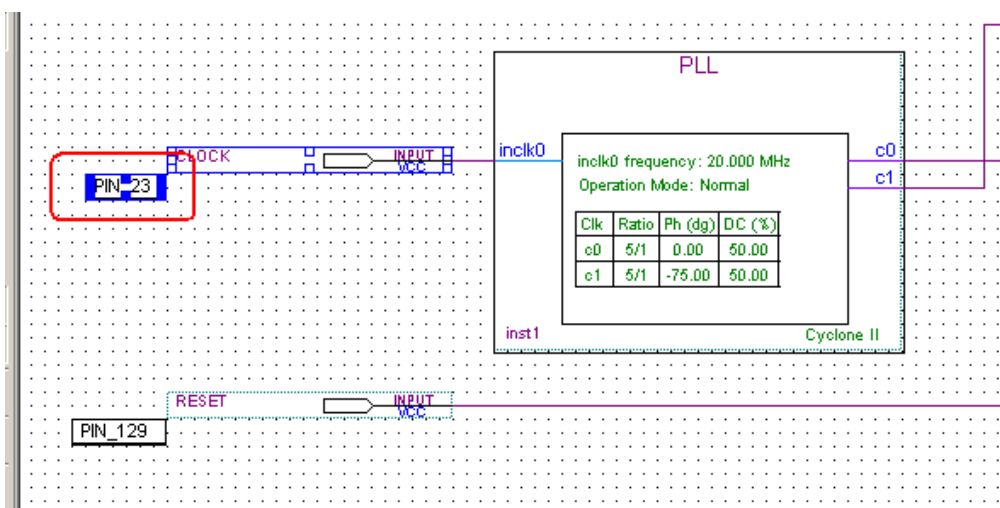
经过了漫长的编译过程，如果没有问题，编译成功后将出现下面的对话框



点击确定，编译过程全部结束。我们可以通过编译报告来看看我们用了多少资源，如下图所示，红圈处可以看出，我们这个系统，用了 66% 的 TLE。

Flow Status	Successful - Wed Mar 24 15:02:27 2010
Quartus II Version	9.0 Build 132 02/25/2009 SJ Full Version
Revision Name	SOPC_T
Top-level Entity Name	SOPC_T
Family	Cyclone II
Device	EP2C5Q208C8
Timing Models	Final
Met timing requirements	No
Total logic elements	3,032 / 4,608 (66 %)
Total combinational functions	2,638 / 4,608 (57 %)
Dedicated logic registers	1,700 / 4,608 (37 %)
Total registers	1816
Total pins	72 / 142 (51 %)
Total virtual pins	0
Total memory bits	48,720 / 119,808 (39 %)
Embedded Multiplier 9-bit elements	4 / 26 (15 %)
Total PLLs	1 / 2 (50 %)

最后 我们需要检查一下 是否每个引脚都已经分配好了 以免有个别没有分配的，影响后面的实验。回到 SOPC_T.bdf，看一看是不是每个引脚都有个小“尾巴”，如下图所示



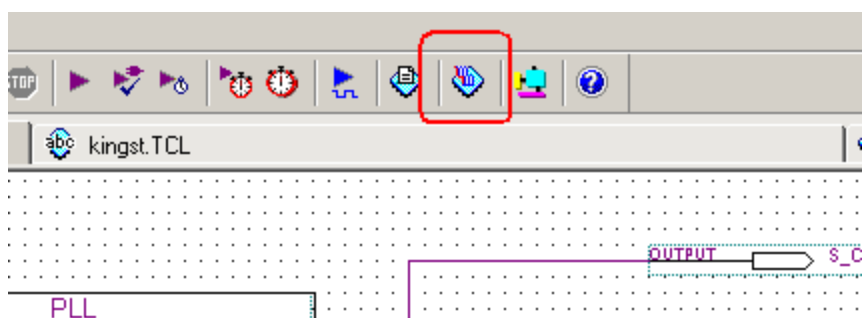
有的,说明这个引脚已经分配好了,仔细检查好每一个引脚,如果没有问题,就可以把程序下载到硬件里面了。

八、 下载程序

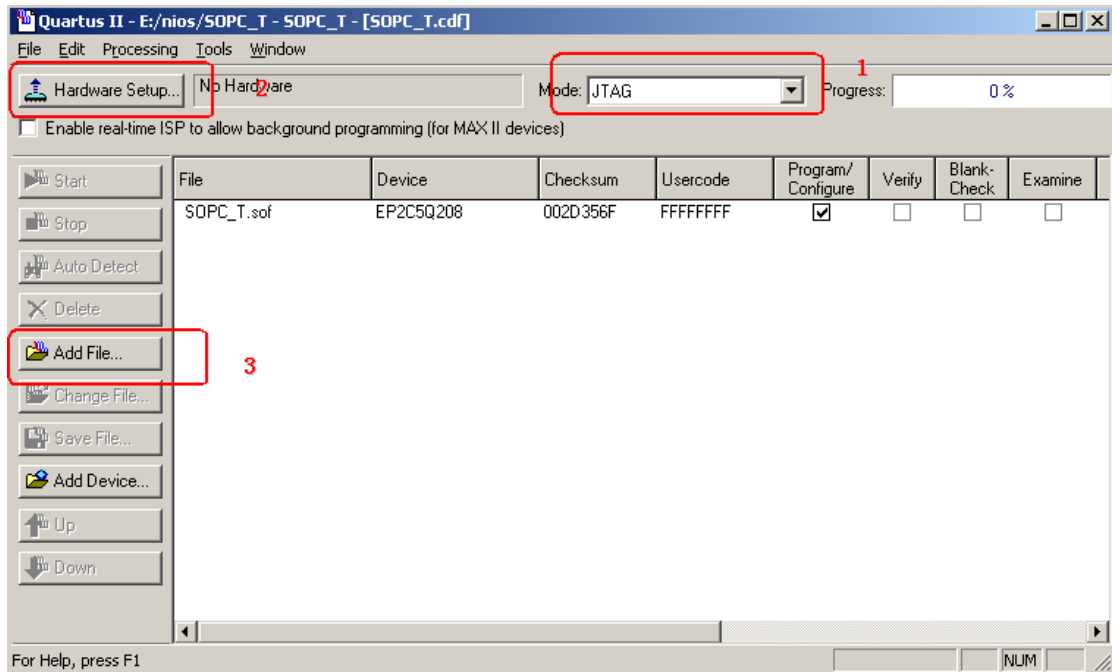
编译好以后的程序会生成两种格式的文件*.sof 和*.pof。*.sof 文件是给通过 JTAG 模式下载到 FPGA 内部的,掉电丢失。*.pof 文件是通过 AS 模式下载到 EPCS1 中的,掉电不会丢失。说到这,简单介绍一下 AS 模式和 JTAG 模式。

Cyclone 系列的 FPGA 使用 SRAM 单元来存储配置数据。SRAM 是易失性的,每次上电之前,配置数据必须重新下载到 FPGA 中。Cyclone FPGA 的配置方式包括:主动配置模式(AS),被动配置模式(PS)以及 JTAG 配置模式。

我的黑金板上配置了 AS 和 JTAG 两种模式,我们通过 AS 口可以将程序下载到 EPCS1 中,如下图操作方式,点击红圈处的图标



点击后,进入下图所示界面,红圈 1 是模式选择,红圈 2 是下载器的选择,红圈 3 加入相应的文件。如果你通过 JTAG 口下载程序,就要将下载线接到 JTAG 口,选择 JTAG 模式,将*.sof 文件加入到这里面。如果你通过 AS 口下载程序,你就要将下载线接到 AS 口,选择 AS 模式,将*.pof 文件加入到这里,大家自己试试吧。最后点击 Start,开始下载。现在市面上的下载线以 USB-BLASTER 为主,价格各有不同,贵的有 200 多的,便宜的 50 多块钱,我觉得可以用就行,贵的未必好,自己的看法啊,大家自行选择吧。



到这里，我们有关硬件开发的部分就高一段落了，如果对以上内容有问题，请大家留言，我将尽快给大家解决。

好了，这一节的内容就讲完了，有问题的可以给我的博客(kingst.cnblogs.com)留言，或者加入我的 qq : 984597569，或者加入最新的群 : 109711029 (107122106 已满)，当然也可以发邮件给我 : avic633@gmail.com，谢谢大家支持。

第三章 软件开发

软件开发

通过本章，您可以详细的了解到 NIOS II 开发的软件部分，通过 NIOS II 9.0 IDE 进行工程建立，配置，编译下载等，每一步都有图片配合指导，就算大家没有任何 NIOS II 开发基础，都可以顺利完成软件开发部分内容。

本章分为以下几个部分：

- 一、回顾
- 二、摘要
- 三、NIOS II IDE 简介
- 四、建立软件工程
- 五、编译
- 六、运行

这一节,我将给大家讲解 NIOS II 软件开发部分,这一部分是以第一节硬件开发部分为基础的,如果大家是从这一节开始看的,那我们就先回顾一下上一节我们所讲的内容。

一、 回顾

上一节,我们详细讲解了 NIOS II 硬件开发部分的全过程,其中包括了工程的建立, NIOS II 软核的构建,以及锁相环 PLL 倍频和如何下载等等。其中 NIOS II 软核的构建是重点内容,我们在软核中构建了 CPU,SDRAM,AVALON 三态桥,FLASH,SYSTEM ID,JTAG UART 6 个模块,时钟我们设计为 100MHz,FLASH 为 8 位模式。这一节我们就在其基础上详细讲解软件开发的全过程。

二、 摘要

首先,我先给大家简单介绍一下这一节的重要内容,这一节将详细讲述 NIOS II 的软件开发的整个流程,并简单介绍 NIOS II 9.0 IDE 的一些简单的使用方法,会让那些从来没有用过 NIOS II IDE 软件的人可以很轻松的上手使用。

三、 NIOS II IDE 简介

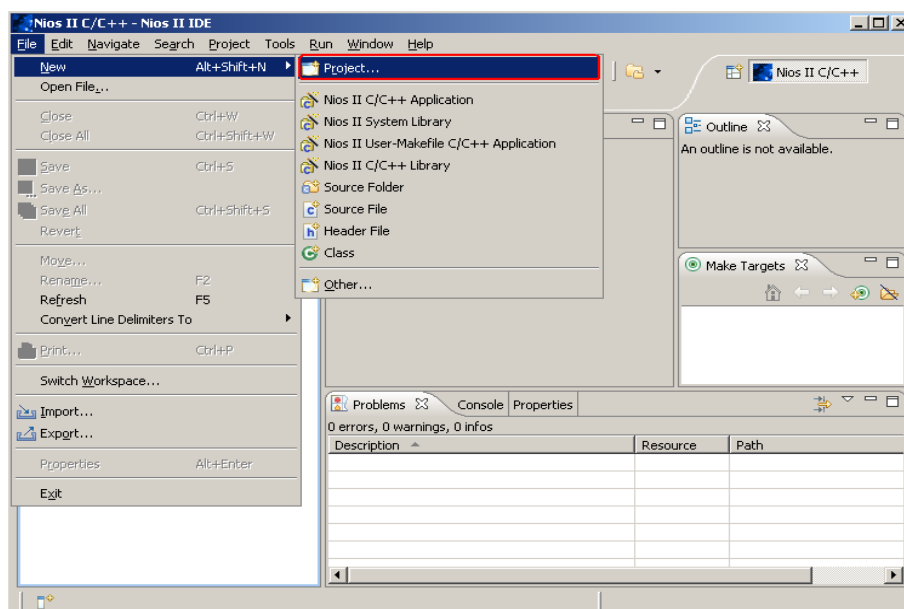
NIOS II IDE 是一个基于 Eclipse IDE 构架的集成开发环境,它包括了很多的功能,学过 JAVA 的人对其应该非常的熟悉,它的功能是非常之强大,下面我们就简单介绍一下 NIOS II IDE 的功能和特点:

- GNU 开发工具。熟悉 Linux 操作系统的人对它一定不陌生,它是一种开源的编译环境,包括标准的 GCC 编译器,连接器,汇编器以及 makefile 工具等。说到这大家可能头都大了,这些都是啥东西啊,如果你是一个 Linux 的忠实粉丝,这些你必须非常的清楚,但在 NIOS II 开发过程中你只要知道这些就 OK 了,没必要刨根问底的弄清楚它们,不过我在这里强烈推荐大家学习学习 linux 方面的东西,你一旦接触上了它,你就会发现它的魅力是太大了,里面的好东西让你一辈子都受益,呵呵,话说多了,我们继续我们的 NIOS II。
- 基于 GDB 的调试工具,包括仿真和硬件调试。这个东西也是在 Linux 平台下流行的调试工具,所以说 Linux 很强大吧。
- 集成了一个硬件抽象层 HAL(Hardware Abstraction Layer);
- 支持 MicroChip/OS II 和 LwTCP/IP 协议栈。
- 支持 Flash 下载 (Flash Programmer 和 Quartus II Programmer)。

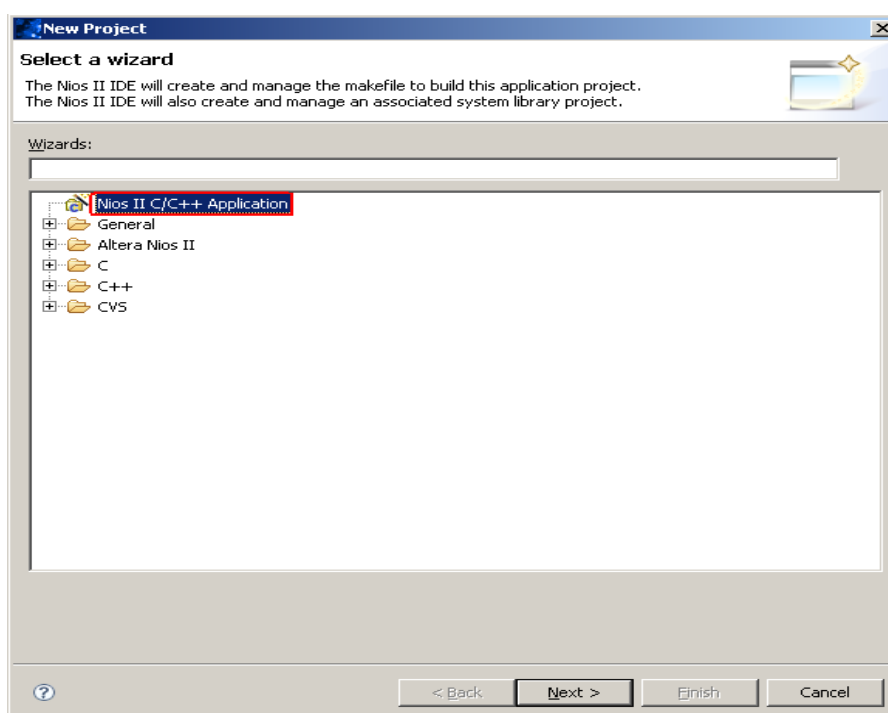
理论的就说这么多了,下面我们来点实际的。

四、 建立软件工程

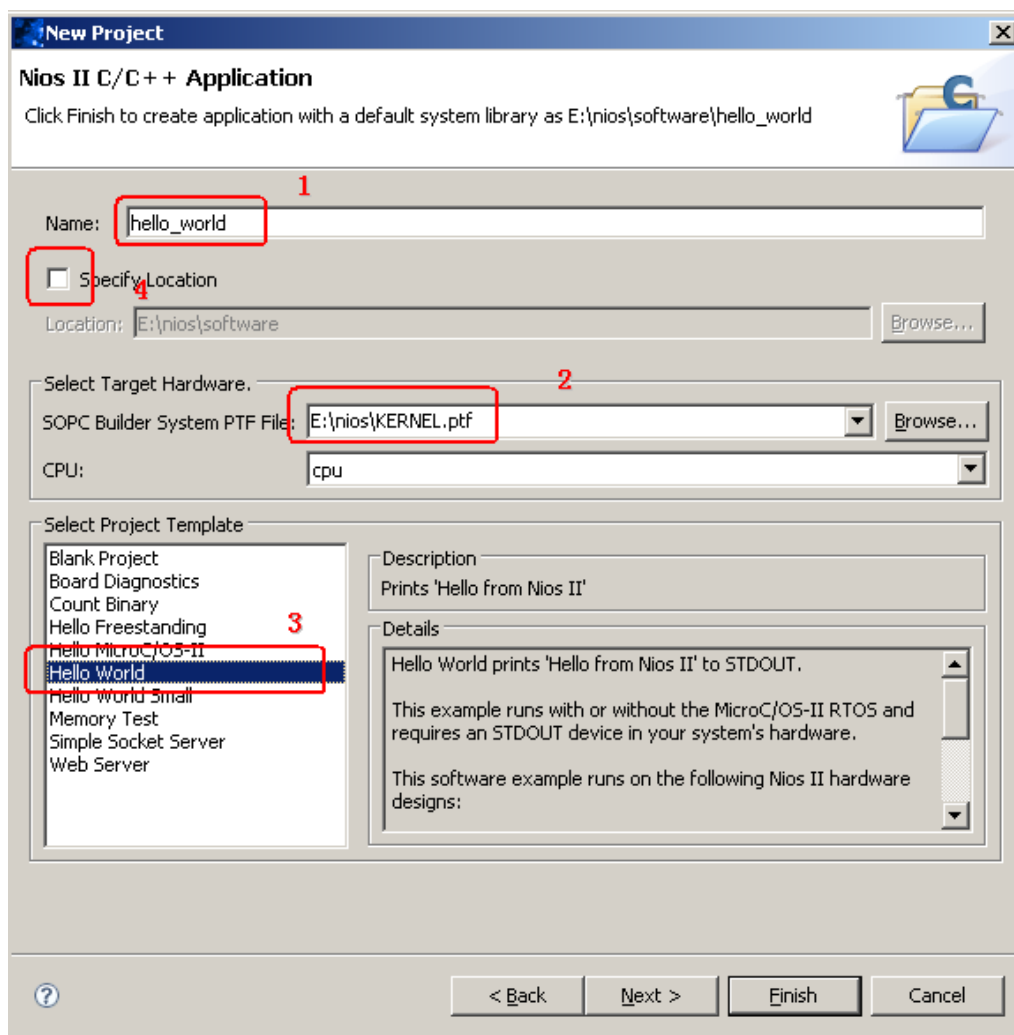
首先，将 NIOS II 9.0 IDE 软件打开，打开后 NIOS II IDE 的界面赫然显现在我们面前，界面很简单，跟其他的 IDE 没什么太大的区别，我们需要做的就是首先建立一个软件工程，操作方式如下图所示，File->New->Project



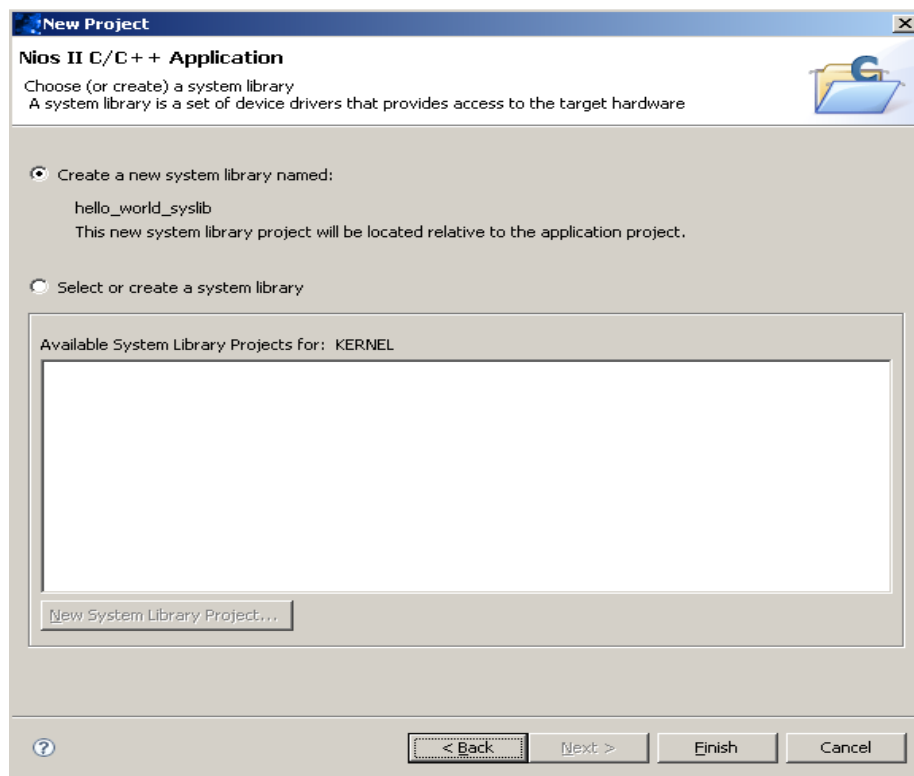
点击后，会出现工程向导界面，如下图所示，选中红圈处的内容，Nios II C/C++ Application，



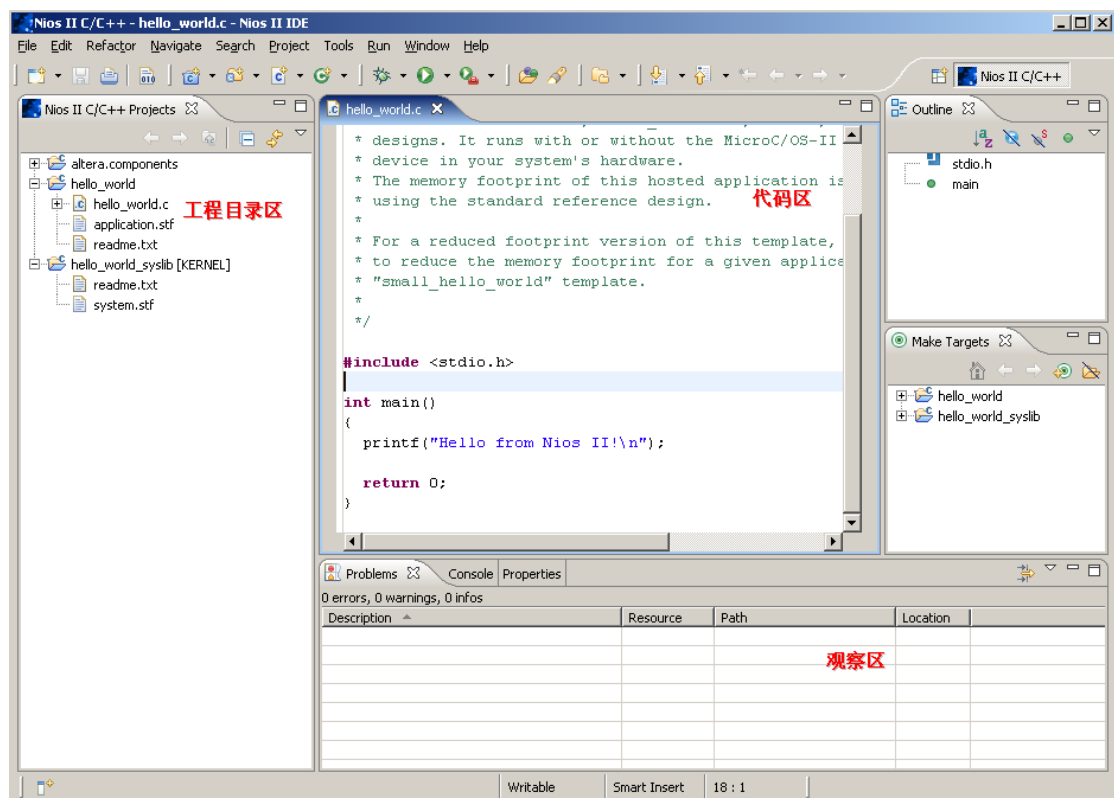
点击 Next 会出现下图所示内容 红圈 1 处是工程名 我将其修改为 hello_world , 红圈 2 处是目标硬件文件, 点击 Browse , 找到我们上一节生成的 NIOS 软核的位置, 这个文件是以 .ptf 为后缀的, 如果大家跟我的地址一样的话, 地址应该是在 E:\nios\KERNEL.ptf。在红圈 3 处选中 Hello World , 这个地方是工程模版。再说红圈 4 , 这个地方是改变工程所放位置的, 如果不修改, 软件工程的位置就在 Quartus 工程目录下的 software 下面。



点击 Next , 这里不用修改 , 点击 Finish , 完成工程向导。

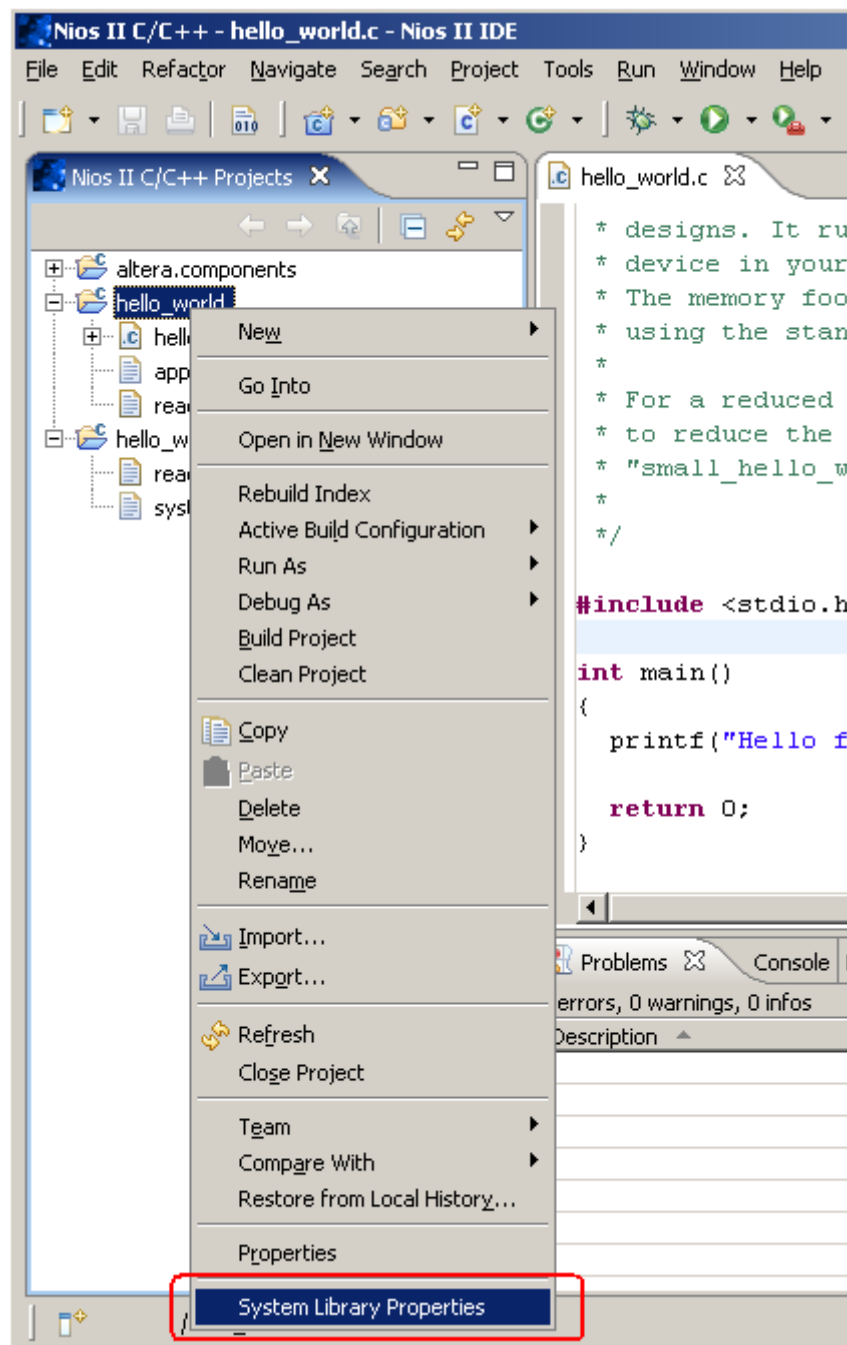


完成了上面的工程向导后，我们正式进入 NIOS II IDE 的界面了，如下图所示，我主要介绍三个部分，其他部分用处不大。我按功能将这三部分命名为代码区，工程目录区，和观察区（这几个名字很山寨吧，哥追求的就是通俗易懂，呵呵）

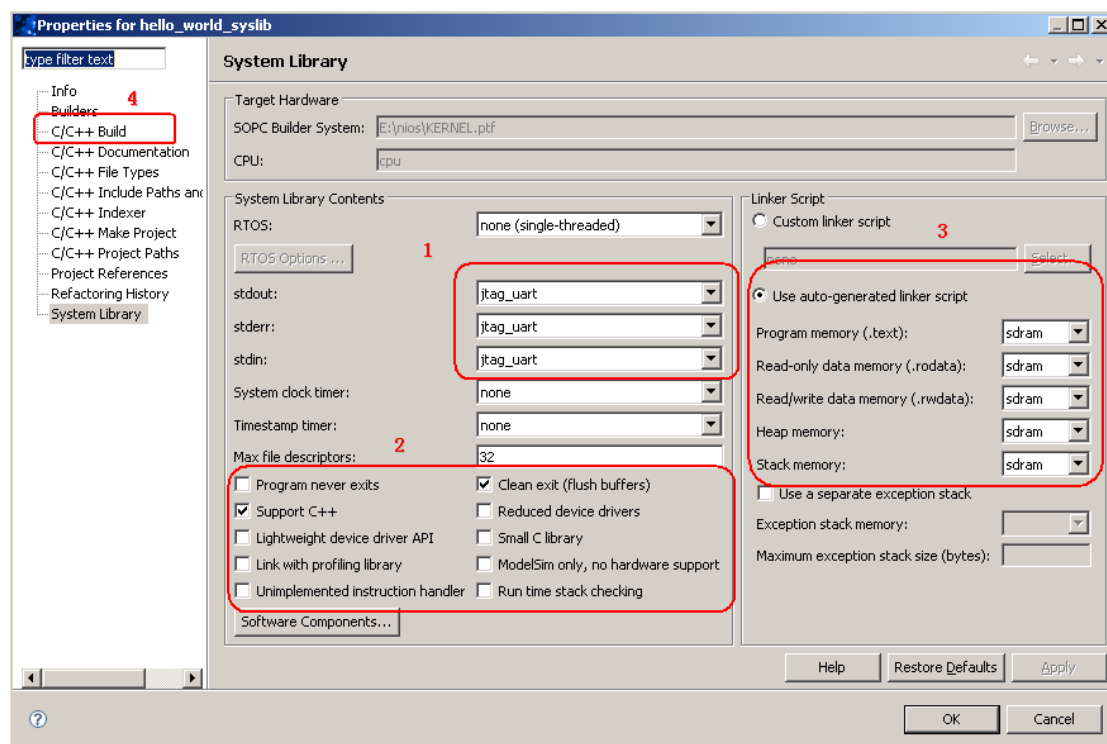


不用我多说，代码区就是显示代码的，工程目录区呢，显示所有与工程有关系的文件，跟我们有关系的.c 和.h 文件，还有一个非常非常之重要的 system.h 文件。观察区中有两个栏我们是会经常用到的，一个是 Console，一个是 Problems。第一个是编译信息显示区，一个是错误警告显示区。有了 JTAG UART 后，第一个 Console（控制台）栏多了一个用途，就是作为标准输出（stdout）的终端，这里不多了，一会我们就会用到。

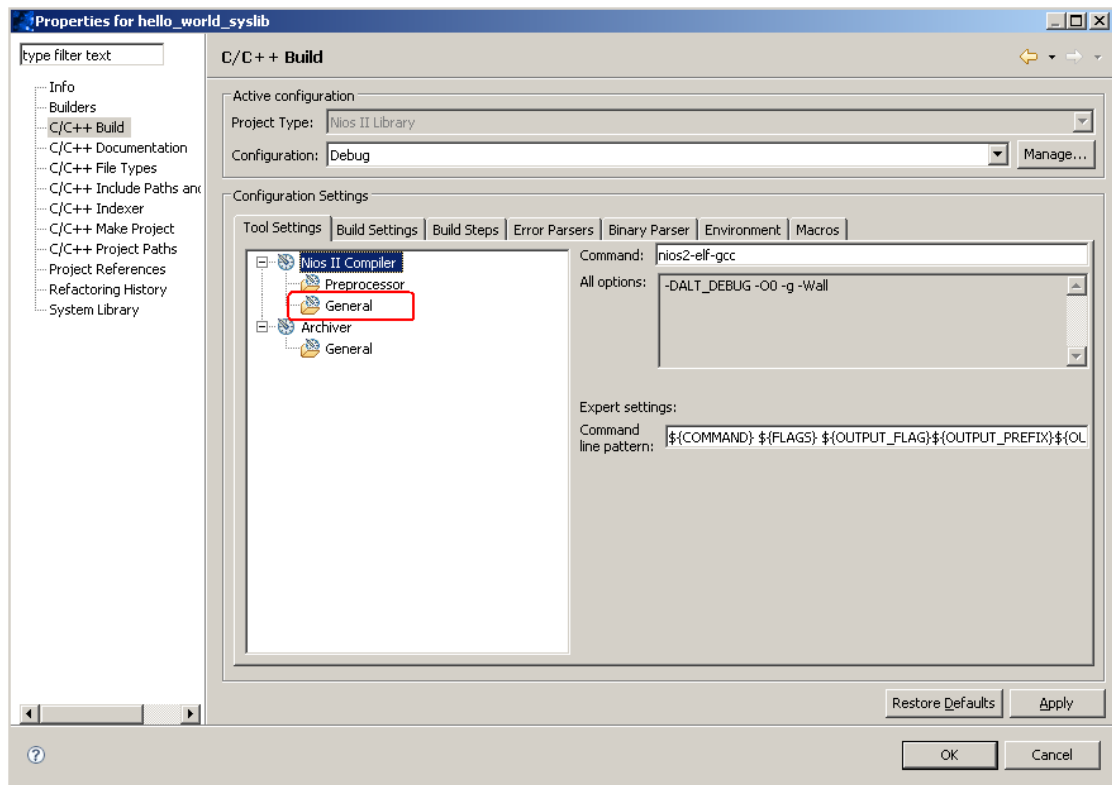
我们接下来的工作就是需要对工程配置一下，大家跟我来吧。在工程目录区中的 hello_world 项单击鼠标右键后，点击红圈处的位置 system library Properties



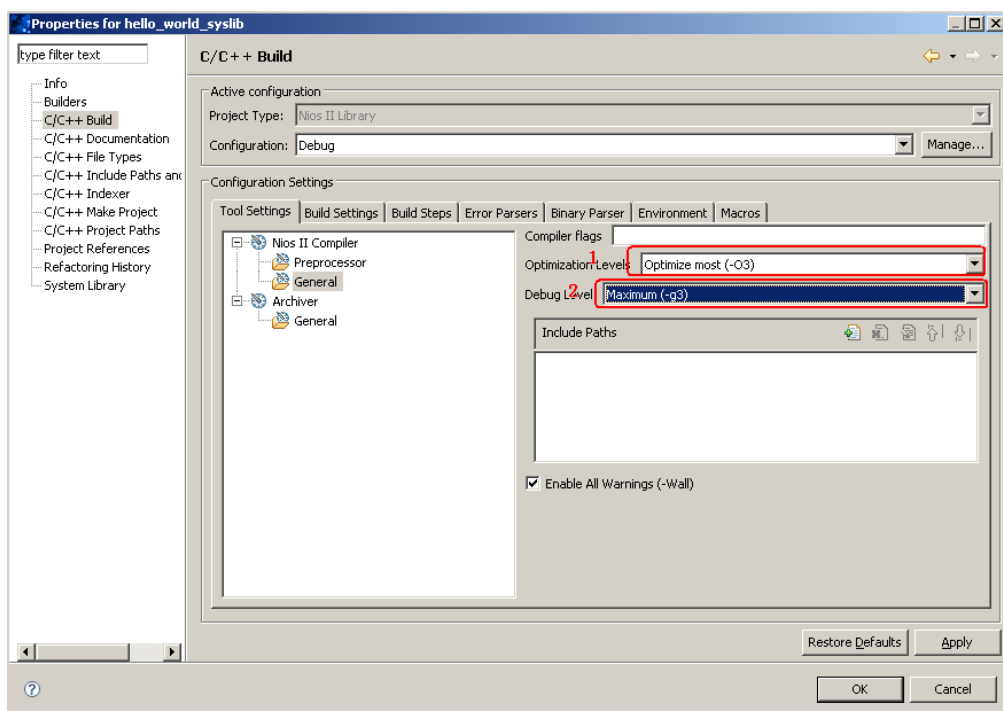
点击后，可看到下图所示界面，



按顺序来，红圈 1 处是标准输入（stdin）标准输出(stdout)、标准错误(stderr)的设置区，我们在软核中构建了 JTAG UART，在此出现效果了吧，如果我们没有构建 JTAG UART，那么，这个地方就不会出现 jtag_uart 选项了。在所红圈 2 处，这个地方也不需要修改，不过有一个地方需要注意，就是 Support C++，这个库相对 Small C library 要大，如果大家手中的板子没有 FLASH、SDRAM 这样大容量存储设备的话，选择 Small C library，用 FPGA 内部的 SRAM，也可以跑些小程序。再说红圈 3 处，这个是一些有关内存的选项，我们构建了 SDRAM 模块，这个地方也用到了，默认就可以，不用修改。该说红圈 4 了，点击红圈 4 后，出现下面界面，这个是对编译器就行配置的界面，大家可以自己观察一下，大部分都不需要修改，我们来看一下比较重要的地方，点击红圈处。



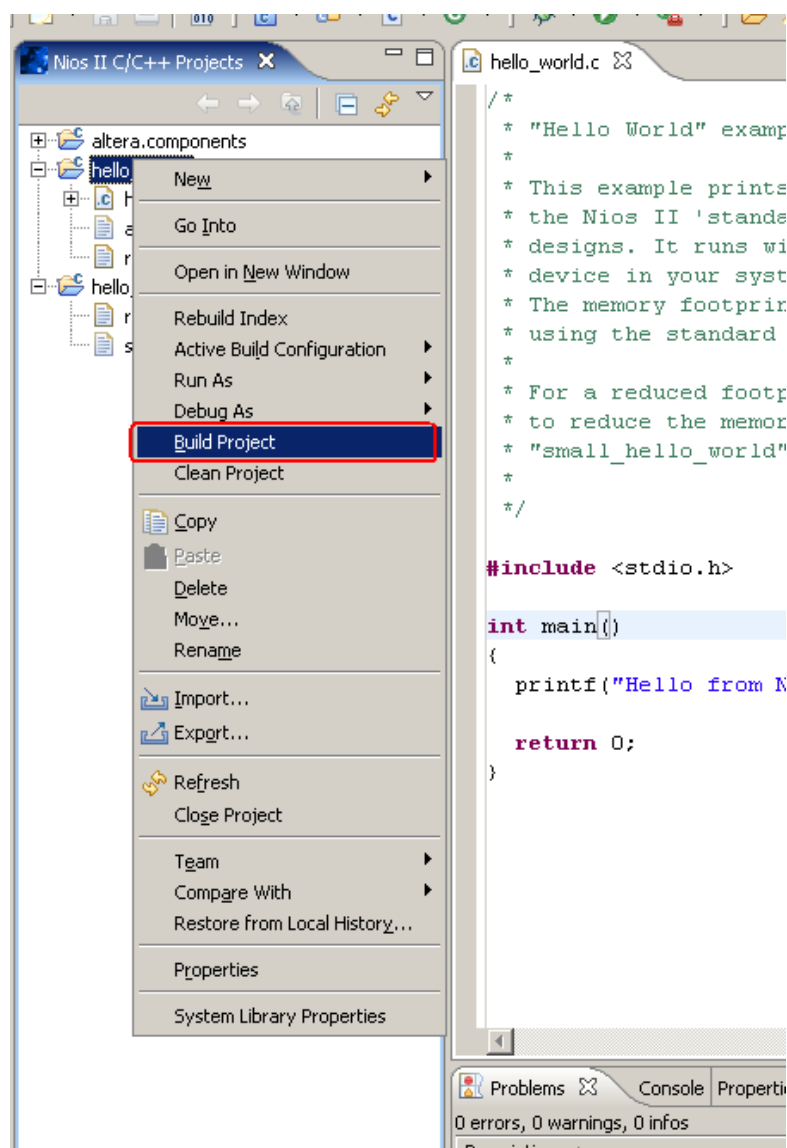
点击后，我们看看是什么样子，这里有两个关键点，一个是红圈 1 处，这个地方时配置编译器的优化级别，红圈 2 的地方是调试级别。编译器的优化级别会让你的生成的代码更小，当要求也很高，你的代码如果不严谨，有可能优化以后不好用了，大家要注意。调试级别是你在编译过程中显示编译内容多少，级别越高显示内容的越多，建议将调试级别调到最高。



将上面设置好以后，点击 Apply，然后点击 OK，回到主页面。

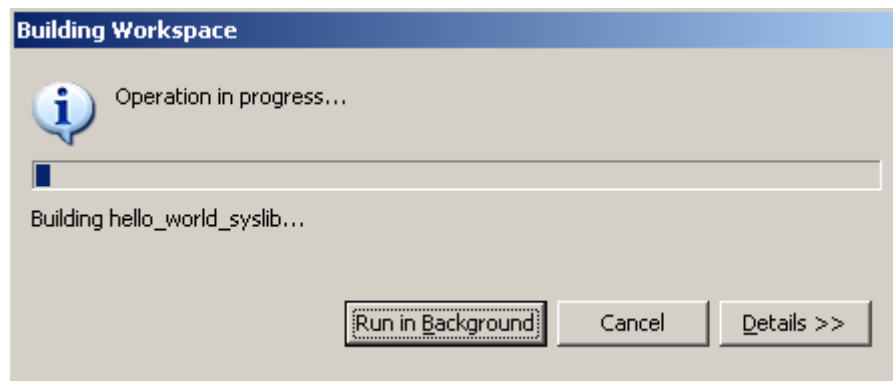
五、 编译

接下来我们就要开始编译了，第一次的编译时间比较长，因为编译过程中会生成一个我们之前所说的一个非常非常重要的文件 system.h，这个文件是根据我们构建的 NIOS II 软核产生的，也就是说，system.h 的内容与软核的模块——对应。一旦软核发生变化，就需要重新编译，重新产生 system.h 文件。现在就开始编译，如下图所示，在工程目录栏中单击右键后，点击红圈处 Build Project，或者直接按快捷键 Ctrl + b。

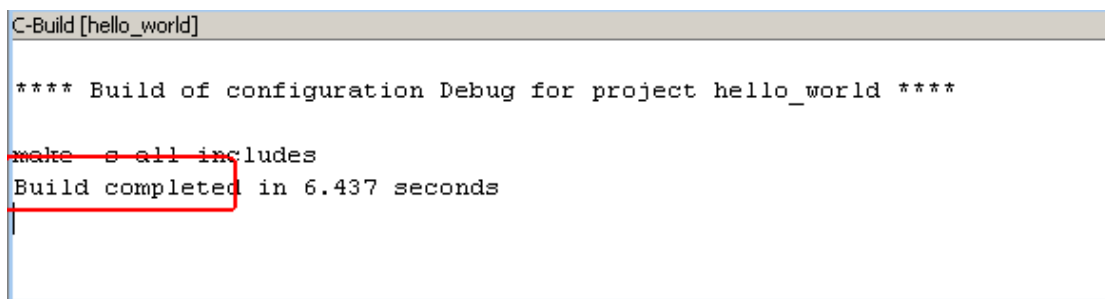


开始编译后，会出现下面的界面，进度条走的很慢吧，如果不想看它，点击 Run in Background，编译就在后台进行了。在编译构成中，大家可以观察观察栏中 Console 栏的内容，其中出现的内容就是编译器正在编译的东西，大家不妨好好看看编译过程都

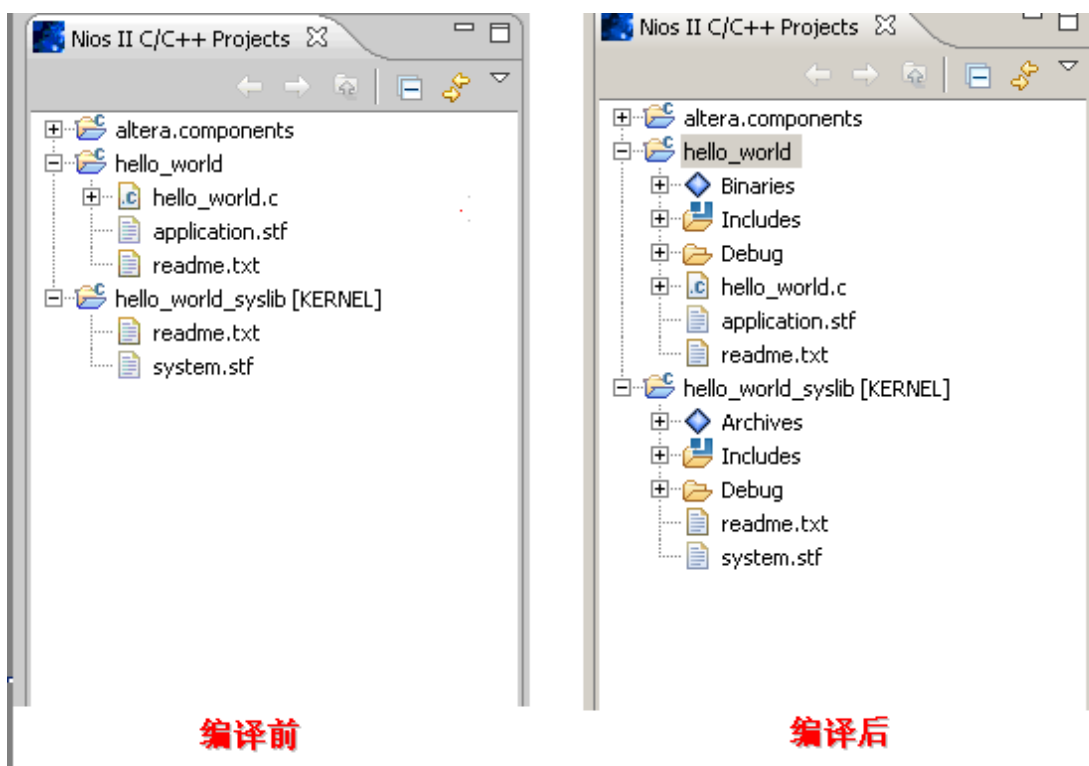
编译和生成了哪些东西。



编译好以后，大家可以看到下面界面，红圈处说明了，编译完成。

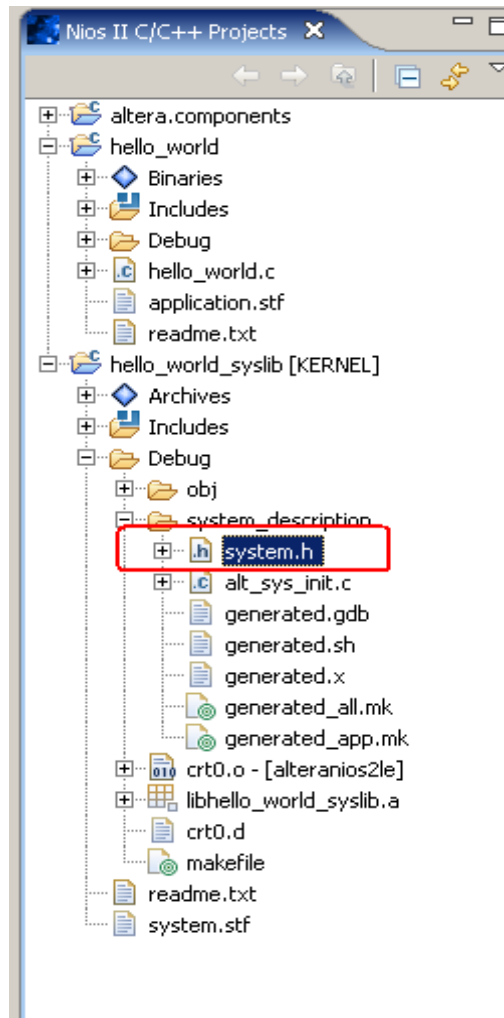


大家可以对比一下，编译前和编译后工程目录栏有哪些变化。



看来编译的成果还是很显著的，其他的我们没必要知道，关键的一个大家要知道，就是我们反复提到的 `system.h` 文件。下面我们就把它揪出来，它隐藏的还是很深的。

看到了吧，它在 `hello_world_syslib/Debug/system_description/system.h`，现在我们就看看里面有什么东西这么重要。



双击以后，在代码区大家就可以看到了，都是一些宏定义，我们找一个典型的作为例子，我给大家讲解一下，看下面的截图

```

/*
 * jtag_uart configuration
 *
 */

#define JTAG_UART_NAME "/dev/jtag_uart"
#define JTAG_UART_TYPE "altera_avalon_jtag_uart"
#define JTAG_UART_BASE 0x00201008
#define JTAG_UART_SPAN 8
#define JTAG_UART_IRQ 0
#define JTAG_UART_WRITE_DEPTH 64
#define JTAG_UART_READ_DEPTH 64
#define JTAG_UART_WRITE_THRESHOLD 8
#define JTAG_UART_READ_THRESHOLD 8
#define JTAG_UART_READ_CHAR_STREAM ""
#define JTAG_UART_SHOWASCII 1
#define JTAG_UART_READ_LE 0
#define JTAG_UART_WRITE_LE 0
#define JTAG_UART_ALTERA_SHOW_UNRELEASED_JTAG_UART_FEATURES 0
#define ALT_MODULE_CLASS_jtag_uart altera_avalon_jtag_uart

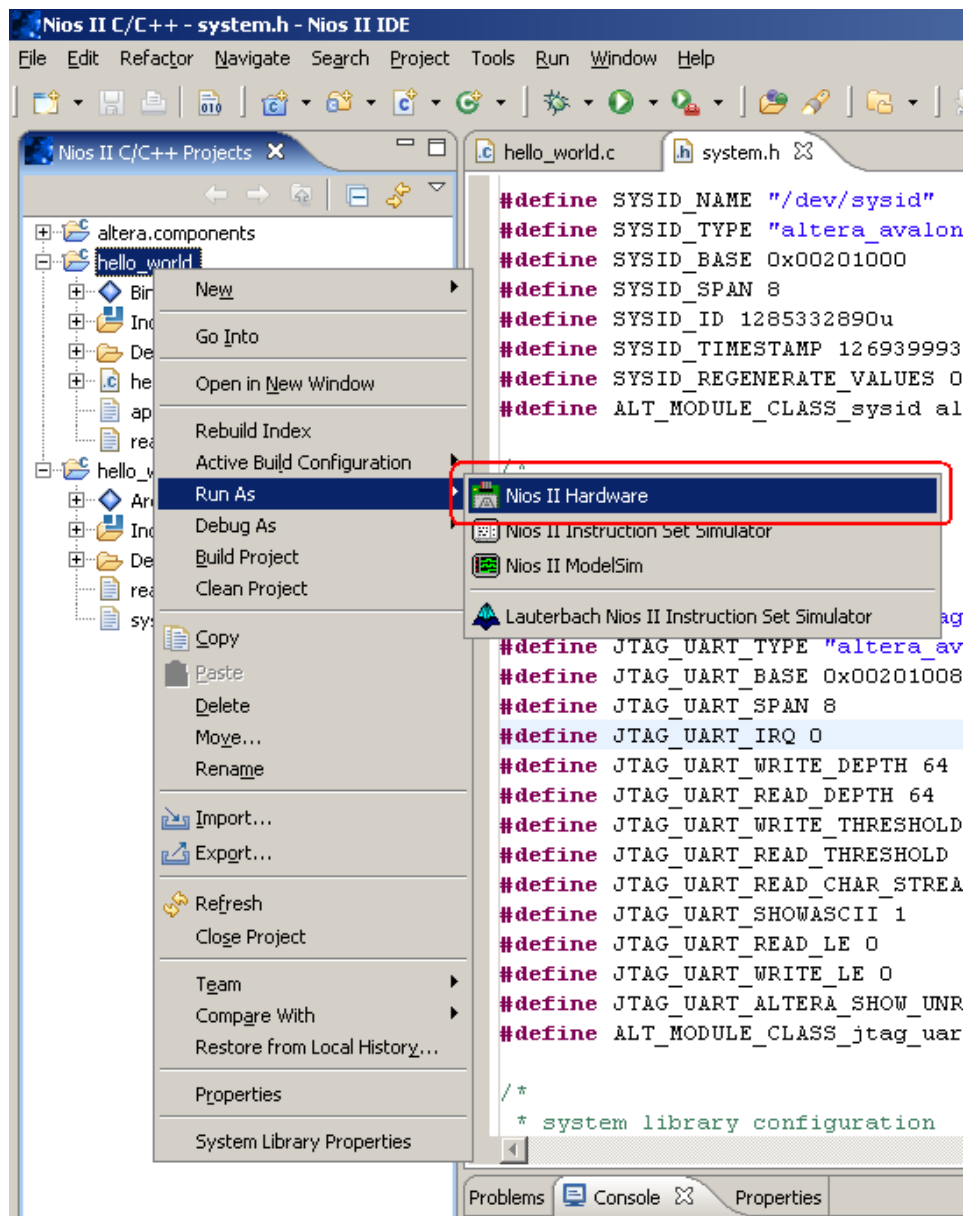
```

在这些信息中，对我们有用的是 JTAG_UART_BASE，还有 JTAG_UART_IRQ，JTAG_UART_BASE 是 JTAG_UART 的基地址，JTAG_UART_IRQ 是中断号，其他的是一些配置信息，我们先不关注。同理，SDRAM、FLASH 等都有相应的基地址，我们以后就要用到这些地址对 NIOS 软核进行寄存器操作，达到我们要实现的跟单片机一样的寄存器操作方式，在此我们就不详细讲述了，后面我们会单独讲解这一节。看了这个例子以后，大家在看看其他的，都大同小异，没有中断的就不会出现*_IRQ 这一项，不信大家自己看。NIOS 强大之处就在于此，根据大家的需求进行对软核的构建，然后产生相应的寄存器，整个构建都由设计者来掌控，缺什么建什么，不行的话还可以自己写底层的模块，你说强不强。

system.h 文件我们以后还会提到，暂时先讲到这，接下来我们要看看我们编译好的程序是不是跟我们想想的一样。

六、 运行

对于 NIOS IDE 提供了几种方法来验证，一种是直接硬件在线仿真，一种是软件仿真。我们先说第一种硬件在线仿真，很显然这种方式需要硬件配合，一块开发板，一个仿真器（仿真器就是大家用的 USB-BLASTER 或者 BYTE-BLASTER）。将仿真器与开发板的 JTAG 口相连（假设你的仿真器驱动已经装好了，如果有不知道怎么装仿真器驱动的请跟我联系）。安装好以后，我们进行下面的操作，点击红圈处 Nios II Hardware。

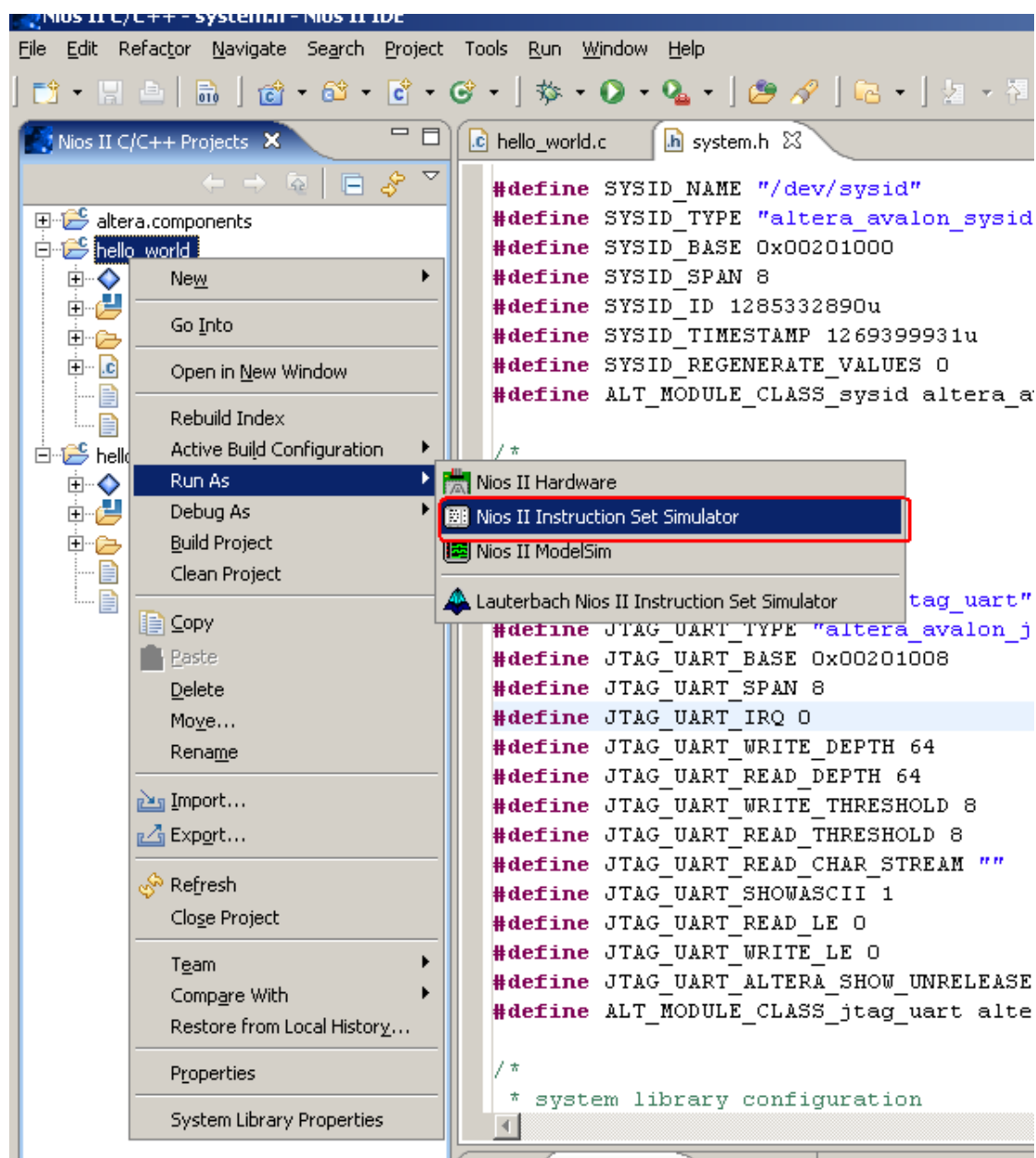


点击后，可以看见观察栏的控制台（ Console ），如果一切正常，我们将看到下面的结果出现。

```
Hello from Nios II!
```

看到了么？如果没看到，再好好检查一下，你的操作是否跟我说的一样，如果自己无法解决，请联系我，我将手把手帮你解决。

说完第一种硬件在线仿真以后，我们再说软件仿真。软件仿真不需要硬件，电脑单独运行即可，按下图所示操作，点击红圈处，Nios II Instruction Set Simulator。



点击后，还是看观察栏的控制台（Console），结果一样吧。我不建议大家用软件仿真，因为软件仿真在不涉及到硬件的情况下还好，如果有相关硬件操作了，效果就没有了。

到此为止，我们的 NIOS II 软件开发部分就结束了，如果想熟练掌握 NIOS II IDE，还要大家自己亲手去试试，光靠我的讲解是不行的，大家没事的时候可以研究一下它的每一个选项，都有什么功能，这样才能加深你对它的熟悉程度，更好的去掌握它，用好它。在此再多说一句，我不建议大家经常更新软件，对于 Quartus 和 NIOS 软件的升级，无非就是解决一些 BUG，多支持一些器件，编译的速度快了一点。在你还没有遇到非常严重的 BUG 之前，最好不要去更新它。更新以后带来的后果就是需要重新熟悉它的特性。NIOS II IDE 9.1 和 9.0 之间变化就很大，使用起来就不是那么顺手。建议大家不要轻易去更改。

好了，这一节的内容就到此结束吧，下一节我将给大家讲解“如何让 NIOS II 的开发像单片机一样简单，看透 NIOS II 系统的寄存器操作方式”。这一节内容是我教程中的核心部分，希望大家耐心等待，拜拜了！

第四章 程序下载

程序下载

通过本章，您可以详细的了解到 NIOS II 程序开发后如何将程序下载到开发板中。

本章分为以下几个部分：

- 一、简介
- 二、下载配置文件
- 三、下载软件程序

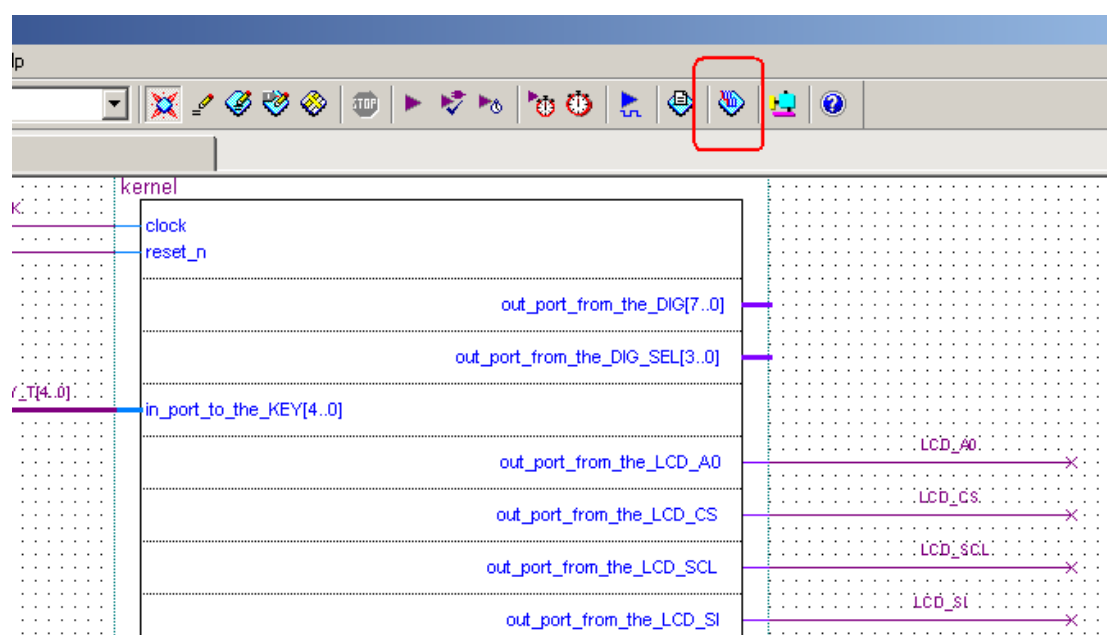
一、简介

这一节，我们来讲解一下如何将编译好的程序下载到开发板中。

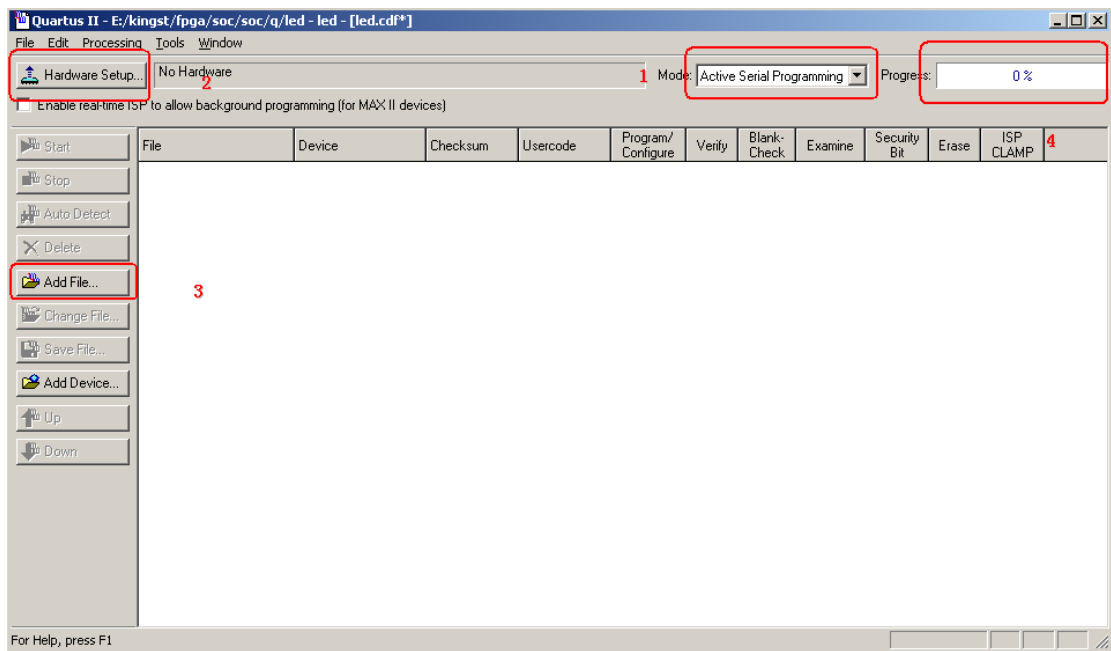
在开发 NIOS 过程中，需要下载两次程序。第一次是在 quartus 软件中，将我们的逻辑和软核生成的配置文件通过 AS 模式下载到 EPCS* (*为 1, 4, 8...) 中，或者通过 JTAG 模式下载到 FPGA 内部的 SRAM 中。第二次是在 NIOS 软件中，将程序下载到 FLASH 中。下面，我们就一个一个的给大家讲解。

二、下载配置文件

首先，需要将 usb blaster 与开发板连接，我们先将其与开发板的 AS 模式接口相连。然后我们打开 quartus 软件（假设我们已经将工程编译好了），打开后，点击下图所示红圈处

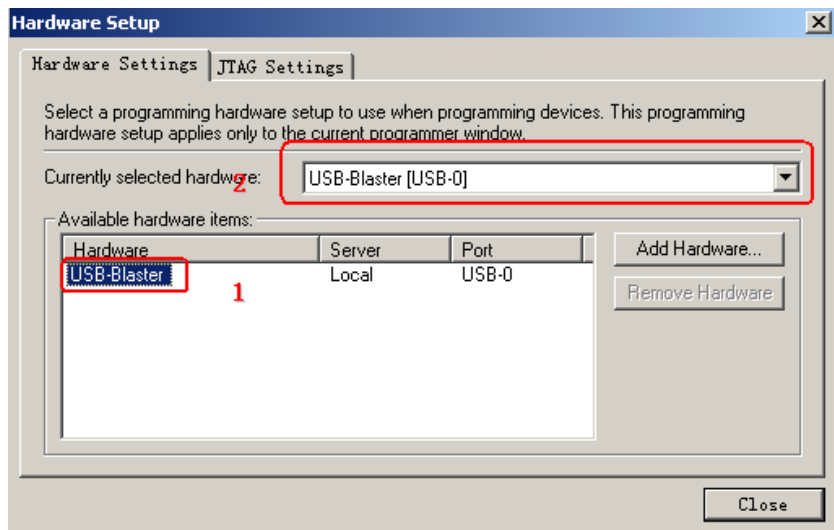


打开后，我们可以看到下图

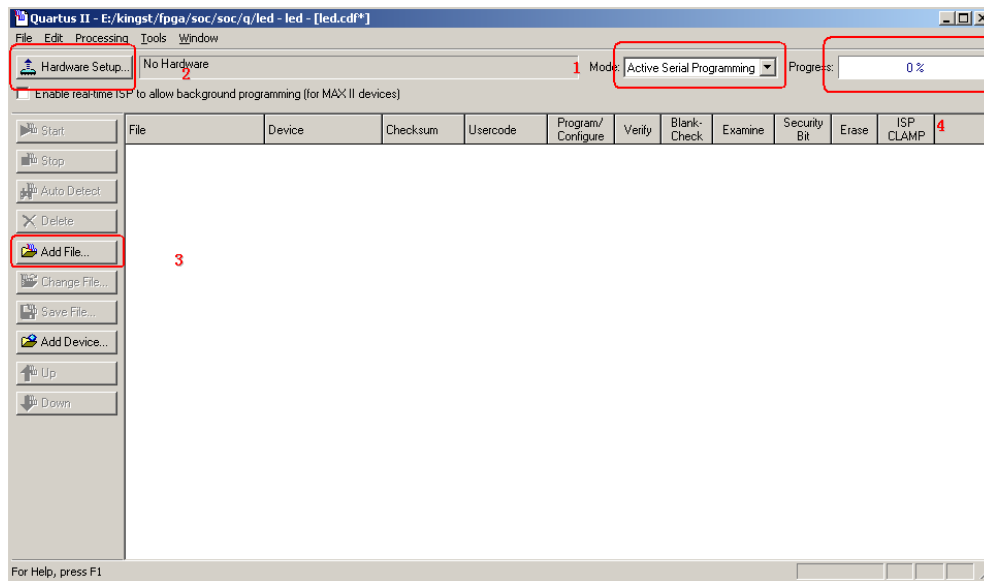


红圈 1 处是选择下载模式，我们选择 Active Serial Programming。如果你发现红圈 2 处旁边写着 No Hardware，你点击红圈 2 处。

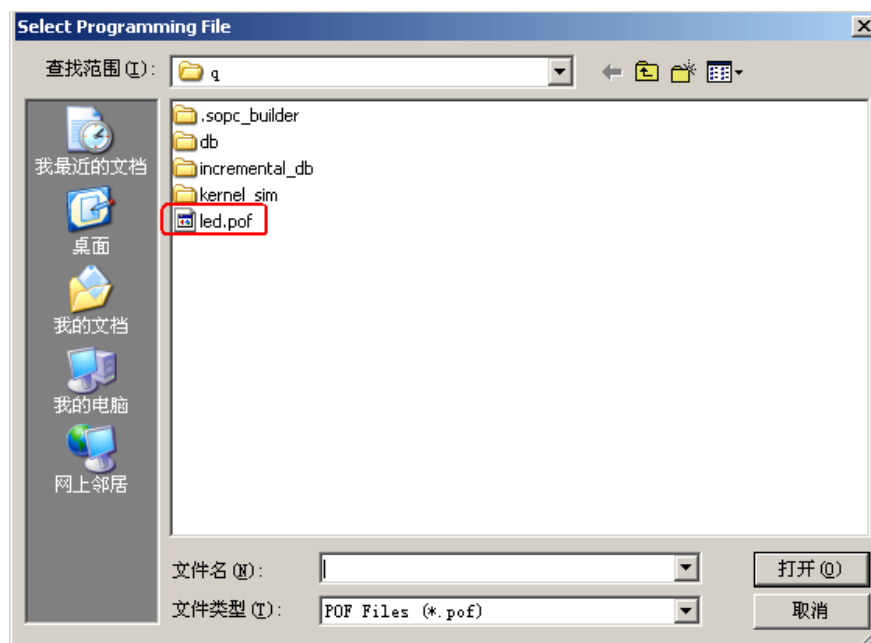
点击后，如下图所示，双击红圈 1 处，发现红圈 2 处为 USB-Blaster[USB-0]即可，点击 close。



点击后，我们回到下图界面，我们点击红圈 3 处（Add File...）将要下载的程序加入进来。

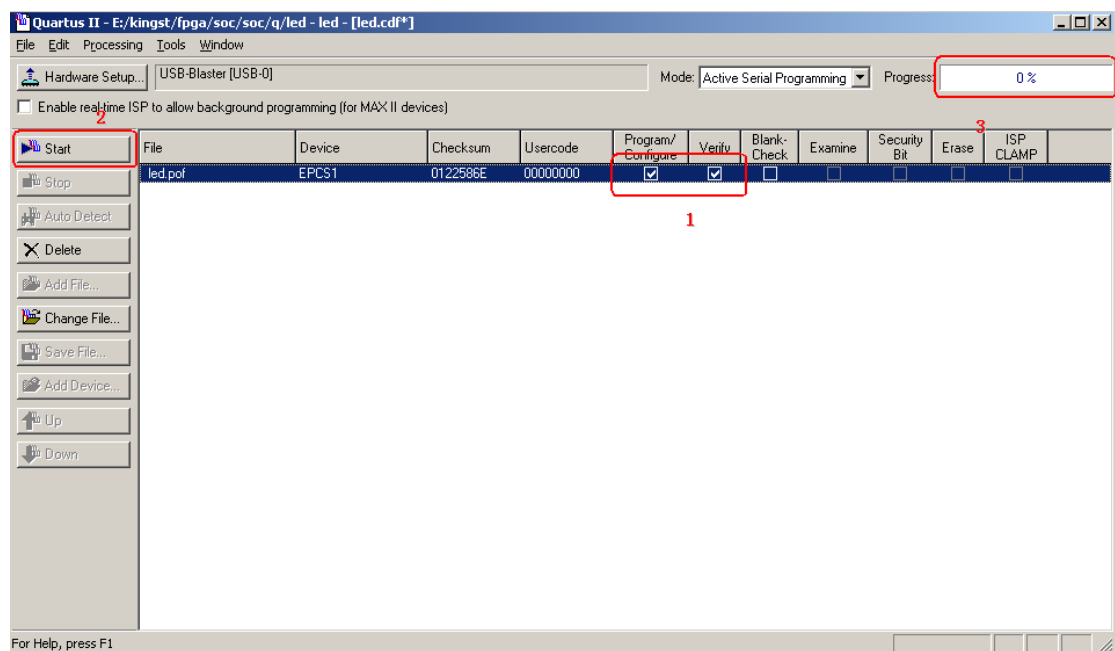


点击后，如下图所示，我们双击红圈处后，点击打开

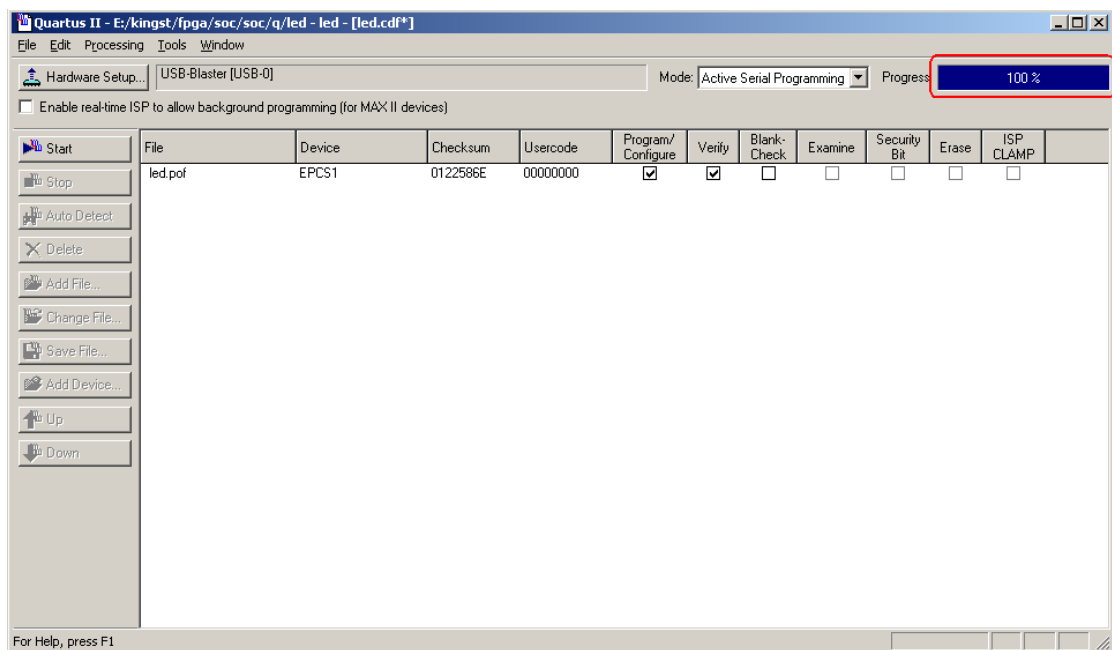


点击后，我们回到下图，我们还需要配置一下，将红圈 1 处的两个选项选中（Program 和 Verity），

然后点击红圈 2 处的 Start，点击后开始下载，红圈 3 处可以看到下载进度。



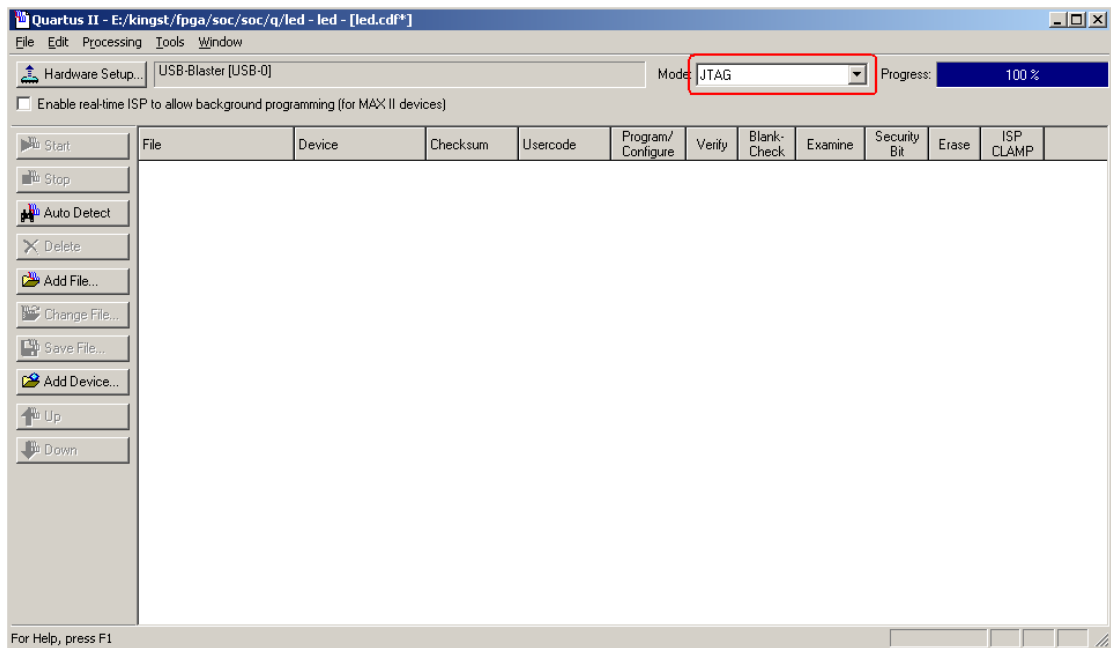
下载好以后，我们可以看到，红圈 3 处变成了下图样子，进度变为 100%，说明下载成功。



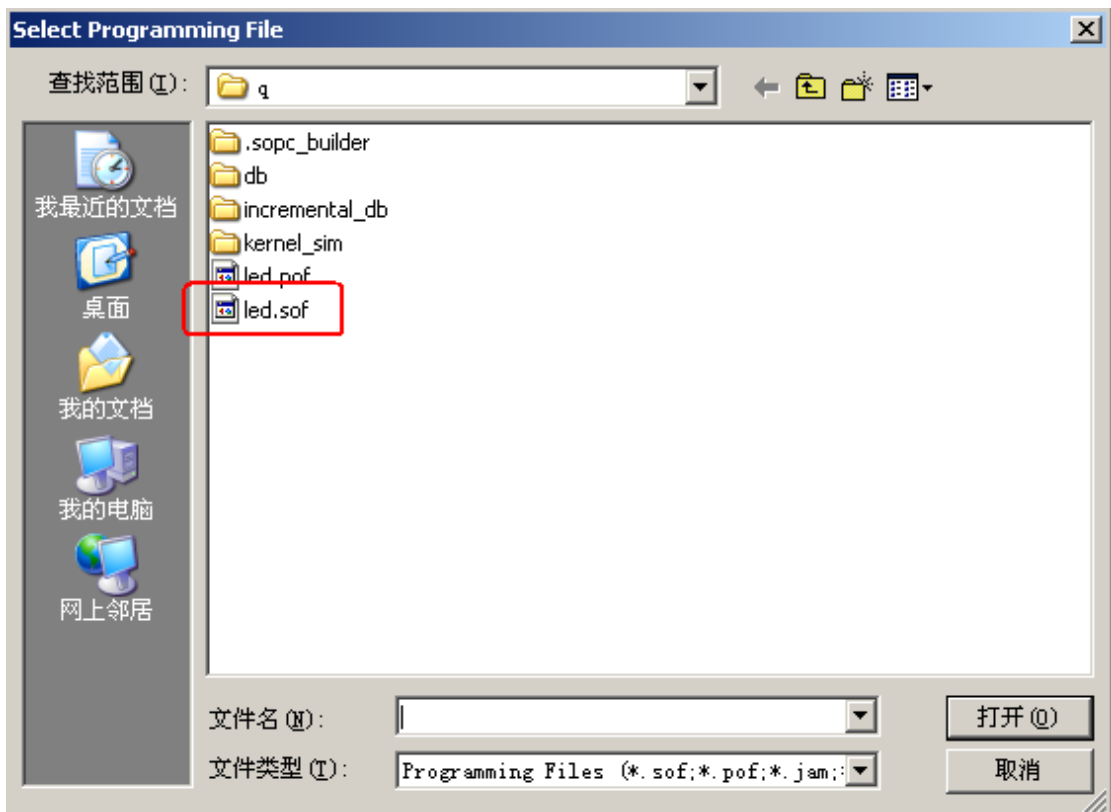
在 AS 模式下，下载成功后，需要把 USB Blaster 与开发板断开，重新启动开发板，下载的程序才能跑起来，这一点大家需要注意。用 AS 模式下载的程序是下载到 EPCS* 的，掉电以后可以保存，数据不会丢失。而用 JTAG 下载的程序时下载到 FPGA 内部的 SRAM 中的，掉电后数据会丢失。下面我们来看看如何实现 JTAG 口的程序下载。

方法跟 AS 模式相似，只是在模式选择的时候有所不同。如下图所示，红圈处选择

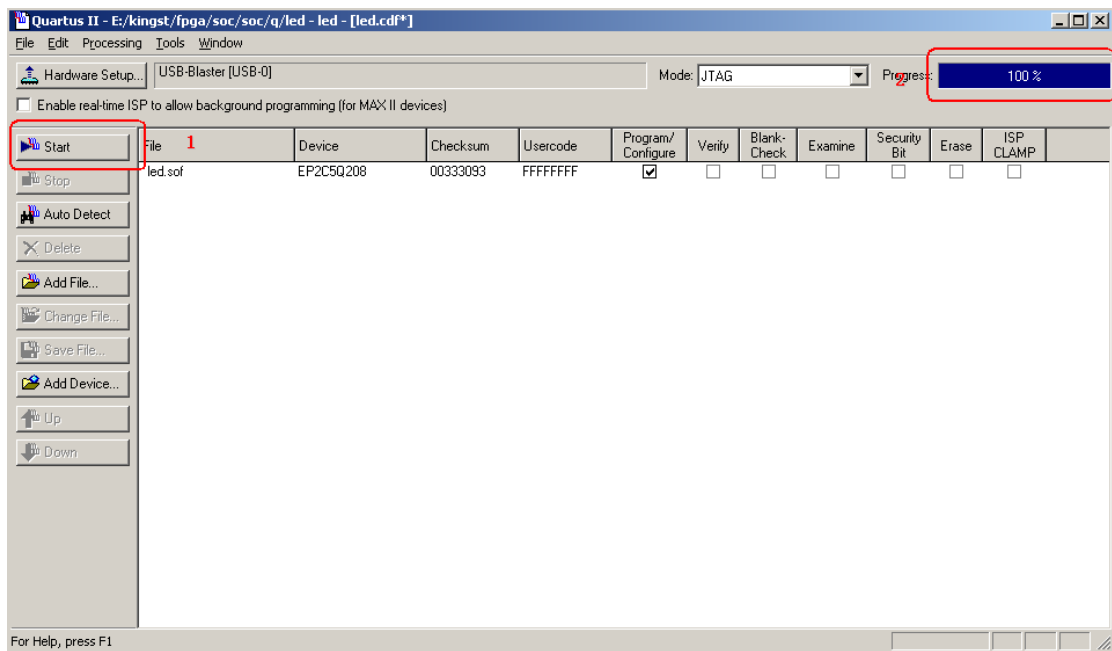
JTAG，然后点击 Add File，



点击后，如下图所示，选中红圈处的 led.sof



点击打开回到下图界面，点击红圈 1 出的 Start，点击后，红圈 2 处会看到进度，完成后进度显示 100%。

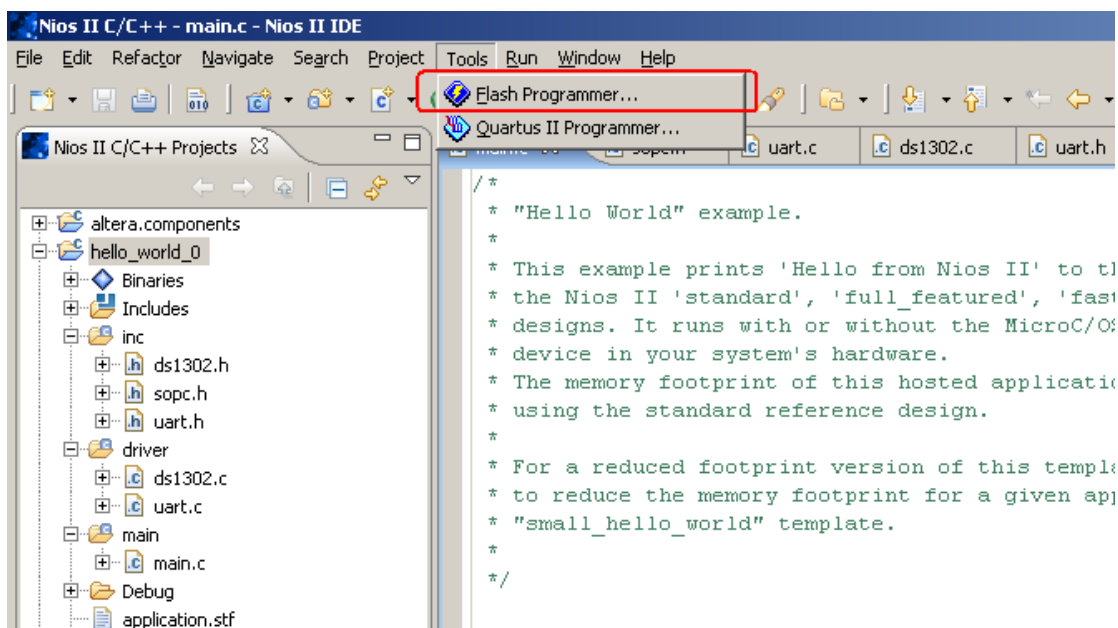


OK，在 quartus 下的程序下载就讲完了，下面我们来讲一讲在 NIOS 下如何进行程序下载到 FLASH 中。

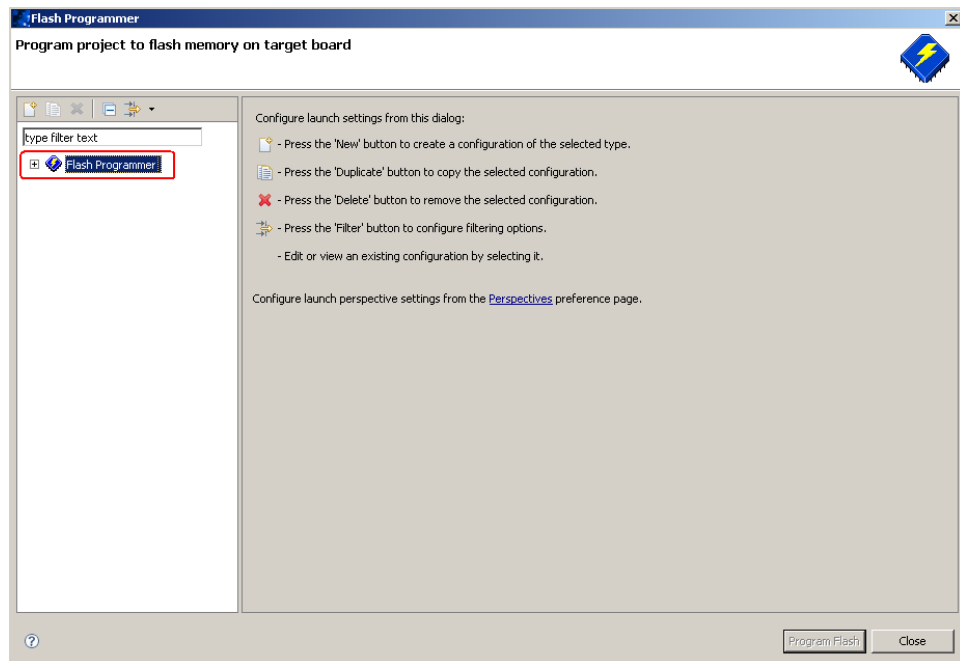
三、 下载软件程序

对于 NIOS 下程序下载，我们只需要 JTAG 口就可以了，我们首先将 USB-Blaster 与开发板的 JTAG 口连接，然后，打开 NIOS II IDE

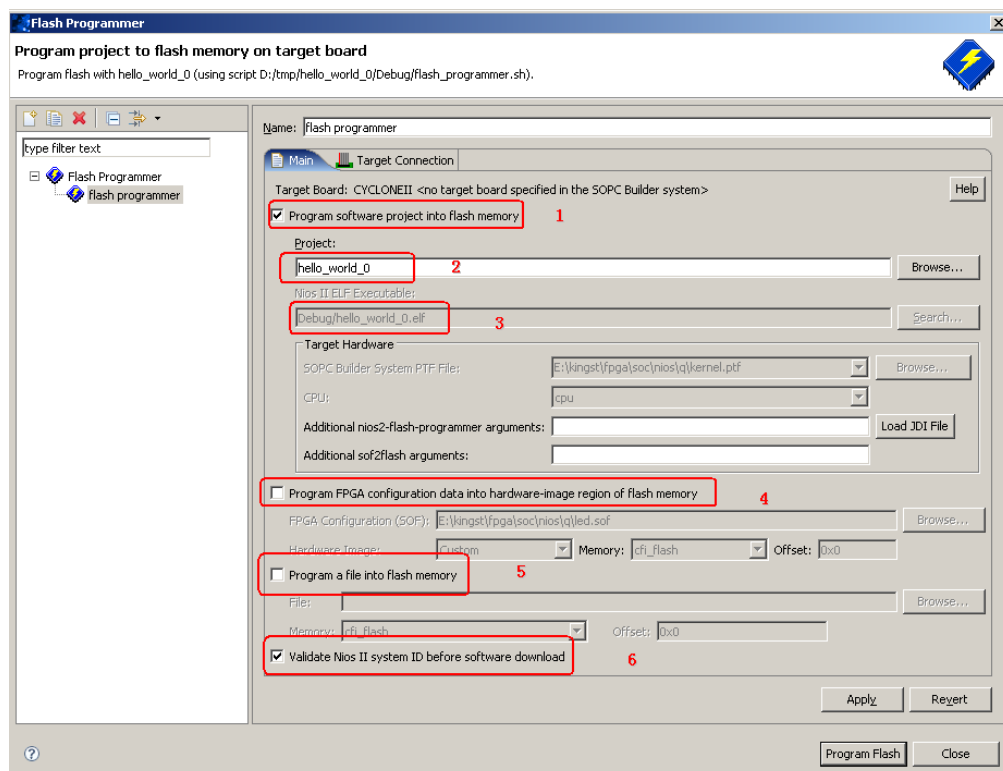
打开后，点击下图红圈处



点击后，如下图所示，双击红圈处



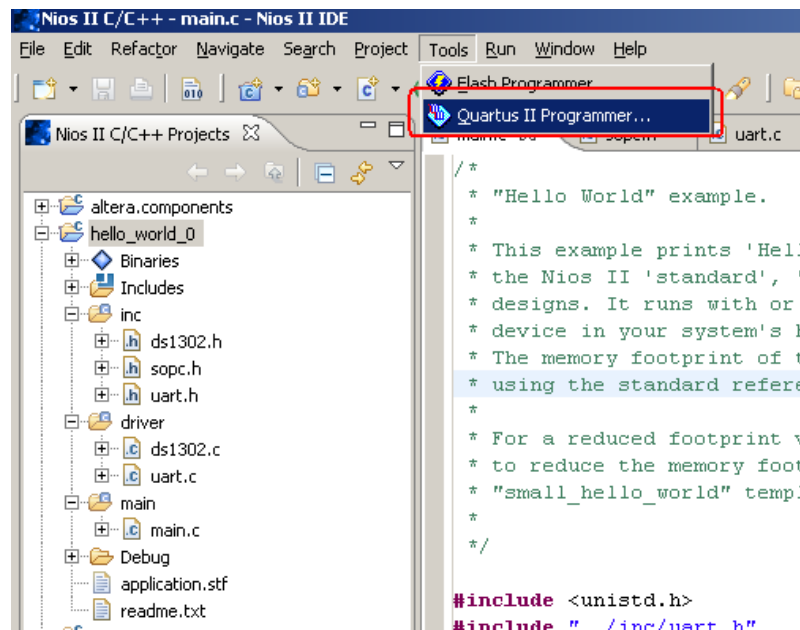
双击后，如下图所示，红圈 1 处选中，就为将 NIOS 程序烧到 Flash 中，红圈 2 为工程名，红圈 3 为烧到 flash 的文件名，以.elf 后缀结束。红圈 4 处是将配置文件烧到 Flash 中。红圈 5 处是将文件烧到 Flash 中，红圈 6 处在下载前检测 SYSTEM ID，这些根据大家的需求来修改。我们要将程序烧到 flash 中，默认就可以了，不需要改变。点击 Program Flash，开始烧写。



烧写过程中，Nios II IDE 的观察栏可以看到下载信息，下载好以后，如下图所示

```
00010000 (50%): Programming  
  
Programmed 68KB +60KB in 5.1s (25.0KB/s)  
Device contents checksummed OK  
Leaving target processor paused
```

烧写好以后，我们重新启动就可以了。其实，在 Nios II IDE 下也可以进入 Quartus II Programmer 下载配置信息，如下图所示



好了，这一节我们就讲到这，有问题的可以给我留言，我的 qq：984597569，或者加入我们的 NIOS 技术高级群：107122106。

第五章 编程规范

编程规范

通过本章,您可以知道一些你在大学 C 语言课堂学不到的东西,但这些东西非常有用,也很值得借鉴,那就是如何规范你的编程风格。本章节由 xiaomage 编写,版权归其所有。本章节为推荐性内容,虽然跟 NIOS 技术没有关系,但它的重要性远远超过了学习 NIOS 技术本身。

本章分为以下几个部分：

- 一、编程参照标准
- 二、格式
- 三、元素及命名规则
- 四、项目管理
- 五、一些建议
- 六、示例代码

一、 规范参照标准

良好的代码风格及编程规范，是书写优良代码的基础，也是工程师必备的技能。本规范遵循 C 语言的创始人 B.W.Kernighan 和 D.M.Ritchie (简称 K & R)所著的《The C Programming Language》一书的示例，并参照 Linux 内核代码风格。

本规范适用于有一定 c 语言基础的读者，对于需要入门的，建议熟读几遍《The C Programming Language》。另外本规范部分内容仅适用于基于单片机、arm 等嵌入式处理器的固件开发。

二、 格式

1. 缩进

函数体、if、for、while、switch case、do while 等都需要使用缩进。不管你用任何编辑器或者是集成开发环境，缩进均是基于“Tab”键的，而不是基于“空格”键。一般来说，我采用 8 个字符的缩进长度。例如：

```
int main(void)
{
    int i;

    for (i = 0; i < 100; i++) {
        printf("Hello world!\r\n");
    }

    return 0;
}
```

2. 空格及空行

空格和空行的出现，是以增强程序的可读性为目的。但是不要插入过多(两个以上)的空格和空行。函数开始局部变量声明后需加一个空行，函数内逻辑相对独立的部分，需加一个空行。文件结尾需加一个空行。

代码中加入空格是以程序逻辑清晰为目的。c 关键字后需加空格，例如：

```
if (a < 100 && a > 50)
    a++;
```

3. 大括号

函数的大括号位于函数体的第二行与末行，if、while、switch、do 的大括号位于关键字所在行的行尾和逻辑末行，末行的大括号与关键字上下对齐。例如：

```
while (i++ < 100) {
    .....
    .....
}
```

三、 元素及命名规则

1. 文件

C语言源文件主要包括 .c 文件和 .h 文件。文件命名要以体现其意义的名词为主，例如芯片 max525 的驱动程序，我们可以命名为 max525.c；实现 fat32 协议的驱动，我们可以命名为 fat32.c；忌出现 a.c、newfile.c、my.c、wang.c 等无意义文件名。

文件名用小写字母、下划线、数字的组合命名，不可出现空格等其他字符，更不允许出现汉字、日语、俄语等非 ascii 码的字符。

每个 .c 文件都要对应有一个 .h 文件来配合其对外资源声明。 .h 文件内可包含宏定义、类型定义、对外资源（全局变量、全局函数）声明。 .c 文件可以包含变量声明、函数原型、函数体。为了防止重复调用，.h 文件的逻辑开头需要加入开关控制，例如：

```
#ifndef __FAT32_H__
#define __FAT32_H__

.....

.....

#endif /* __FAT32_H__ */
```

2. 宏、枚举体

宏、枚举体均需用大写字母、数字及下划线的组合，宏与常量之间用“tab”隔离，同一类含义的宏定义在一起，并放于相关的头文件中。宏定义以能表达清楚含义为标准，除专业术语外，推荐用完整单词表示宏含义。不同含义的宏定义需用空行分割，部分需加注释。例如：

```
#define    WR0      0x00
#define    WR1      0x01
#define    WR2      0x02

#define    SYSTEM_CLK      40000000 //40MHz

#define    MAX_FREQUENCY    20000000 //20MHz
```

3. 自定义类型

我们可以用 c 关键字 typedef 进行自定义 c 语言中的数据类型。类型定义一般包括结构体、联合体类型定义及函数类型定义。ANSI C 包含的数据类型（如 unsigned char、unsigned short int、double 等），不建议重定义。

结构体、联合体类型定义推荐大写字母加 _T 的形式出现，例如：

```
typedef struct{
    char receive_buffer[BUFFER_SIZE];
    unsigned long int baudrate;

    int (* initialize)(int /* baudrate */);
    int (* send_string)(char * /* string */);
    int (* printf)(unsigned char *,...);
}UART_T;
```

4. 函数声明及实体

函数命名采用谓宾结构，中间用下划线隔开，函数必须使用小写、数字及下划线的组合。不管函数原型声明还是函数体，必须包含完整的函数类型及参数类型（包括 void 型亦不能省略）。文件的内部函数（不需要外部调用），需要在函数类型前加 static 关键字。

函数原型声明时，需用注释的方式，添加函数调用参数的意义，例如：

```
int write_data(int /* channel */, int /* data */);
```

大部分函数需返回函数执行状态,定义 0 为正常执行, -1 为一般错误, -1 ~ -999 为自定义错误。自定义错误可以通过宏或者变量实现。例如:

```
int write_data(int ch,int dat)
{
    if (ch > LTC2600_MAX_DATA)
        return ERROR_LTC2600_DATA_OVERFLOW;
    if (dat > LTC2600_DACH)
        return ERROR_LTC2600_CHAN_OVERFLOW;

    .....

    return 0;
}
```

从逻辑功能划分的角度来讲,函数需要简、短、精,函数实现的逻辑内容要跟函数名要——对应,不要超过函数名表达的范围,也不要只实现函数名所表达的部分功能。一般情况,尽量调用系统库来实现功能而不是自己去实现。

5. 变量及初始化

变量一律用小写字母、数字及下划线实现,全局变量要体现变量的意义,需用单词的全写; 由于局部变量作用的范围,一般都在视野范围你,所以可以用简写及单个字母,如 i、a 等。

全局变量尽量越少越好,并且需根据属性划分,以结构体形式体现为主。整个工程的全局变量需在头文件中用 extern 关键字对外声明,隶属于文件的全局变量(不属于整个工程的全局变量),需加 static 关键字修饰。

变量使用前必须初始化,初始化可以采用静态初始化和函数执行时初始化。结构体的初始化建议采用 c99 规范里的指定初始化。例如:

```
UART_T uart0 = {
    .baudrate = 9600,
    .initialize = uart0_init
};
```

数组维数最好用宏定义,数组用时必须初始化(赋值或者清零),对数组维数判断时,需用 sizeof 关键字,切忌直接用数字。

使用指针时，切忌指针越界及野指针的出现；指针也是个变量，用它之前也必须初始化。对 CPU 外设进行直接映射或者是中断内变量使用时，变量前需加 volatile 以防系统优化。

6. 注释

注释不宜过少，但也不宜过多。以表达清楚程序员的意图为最终目的，注释尽量不要用中文。函数体内注释，推荐采用 “//” 的注释方式。

每个文件头，均需加一个说明性的注释。例如：

```
/*
 * File           : ltc2600.c
 * Description    : This file is ltc2600 (8-channel 16bit DAC) driver.
 * Author        : XiaomaGee.
 * Copyright     : Harbin WAVETEK. Co.,Ltd.
 *
 * History
 * -----
 * Rev      : 0.00
 * Date    : 05/30/2009
 *
 * create.
 * -----
 * Rev      : 0.01
 * Date    : 06/01/2009
 *
 * Fix bugs.
 * -----
 *
 */
```

每个函数体的开始，均需加一个说明性的注释。例如：

```

/*
 * Name      : write_data
 * Description : Write data to ltc2600.
 * Author    : XiaomaGee.
 *
 * History
 * -----
 * Rev      : 0.00
 * Date     : 05/30/2009
 *
 * create.
 * -----
 */

```

四、 项目管理

1. 项目文件夹

每个 C 工程中，可以以功能为依据对源文件进行文件夹分类。文件夹不可以出现空格、除英文字母、数字、下划线外的字符；更不允许出现汉字、俄语等非 ASCII 码字符。例如某个工程可以划分为如下文件夹：

doc	说明文档
config	系统相关配置文件夹（系统参数配置、编译器、连接器配置文件等）
driver	硬件驱动文件夹（包括芯片驱动、cpu 外设驱动等）
font	字体驱动
Gui	图形界面驱动
main	主程序
include	头文件
obj	编译后的目标文件

2. 功能划分

功能划分应以逻辑清晰、层次关系明显为目的。一旦划分好后，不可越级调用系统资源（例如只有 driver 内文件内直接操作硬件资源，其他文件夹代码均不可调用最底层硬件资源）。也不要互相调用而使系统资源很快耗尽。

3. 文件管理

文件一旦建立，就需在文件头说明文件目的、版权及历史记录，每次修改后必须记录。工程完工或者是间歇性搁置时，需要对所有工程文件夹、文件进行只读属性设置及当前状态记录（进行到什么状态、存在什么 bug 等）。对源代码最好做到每天一备份，以防意外篡改及丢失，以备恢复之用。

五、 一些建议

1. 代码编辑器

有很多优秀的代码编辑器供大家选择，例如 Vim、Emacs、Souce-Insight、Edit-Plus 等。每个人可以根据自己的喜好，选择一种编辑器，切忌滥用。

2. PC 端编译器及集成开发环境

GCC (GNU Compiler Collection) 是一个非常优秀的编译器套装。他几乎在所有的操作系统下均有移植，并且有很多 CPU 的交叉编译器可供我们使用，例如 SDCC、arm-elf-gcc 等。MS-Windows 下推荐使用 Mingw32 移植版，相应的集成开发环境推荐 Dev-cpp 和 Code :: block。

3. 参考资源及网站

《The C Programming Language》 c 语言圣经；

《Advanced Programming in the UNIX Environment》 UNIX C 程序员圣经；

<http://www.gnu.org> GNU Operating System

<http://www.sf.net> Source Forge

<http://www.kernel.net> The Linux Kernel Archives

六、 示例代码

1. C 文件

```

1 /*
2  * File       : ltc2600.c
3  * Description : This file is ltc2600(8-channel 16bit DAC) driver.
4  * Author      : XiaomaGee.
5  * Copyright   : Harbin WAUETEK. Co.,Ltd.
6  *
7  * History
8  * -----
9  * Rev   : 0.00
10 * Date  : 05/30/2009
11 *
12 * create.
13 * -----
14 * Rev   : 0.01
15 * Date  : 06/01/2009
16 *
17 * Fix bugs.
18 * -----
19 */
20
21 //-----Include files-----//
22 #include    "..\include\spi.h"
23 #include    "..\include\ltc2600.h"
24
25 //-----Function Prototype-----//
26 static int initialize(void);
27 static int write_data(unsigned short int /* channel */, unsigned short int /* data */);
28
29 //-----Variable-----//
30 LTC2600_T    ltc2600 = {
31 .initialize = initialize,
32 .write_data = write_data
33 };
34

```

```

35 //-----Function-----//
36 /*
37  * Name       : initialize
38  * Description : Initialize ltc2600 hardware.
39  * Author      : XiaomaGee.
40  *
41  * History
42  * -----
43  * Rev   : 0.00
44  * Date  : 05/30/2009
45  *
46  * create.
47  * -----
48  */
49 static int initialize(void)
50 {
51     PINSEL2 &= ( ~ (1 << 6));
52     IO3DIR |= (1 << LTC2600CS);
53
54     LTC2600CS_ON;
55
56     return 0;
57 }
58

```

```

59 /*
60 * Name      : write_data
61 * Description : write data to ltc2600.
62 * Author    : XiaomaGee.
63 *
64 * History
65 * -----
66 * Rev   : 0.00
67 * Date  : 05/30/2009
68 *
69 * create.
70 * -----
71 */
72 static int write_data(unsigned short int ch,unsigned short int dat)
73 {
74     if (dat > LTC2600_MAX_DATA)
75         return ERROR_LTC2600_DATA_OVERFLOW;
76
77     if (ch > LTC2600_DACH)
78         return ERROR_LTC2600_CHAN_OVERFLOW;
79
80     LTC2600CS_OFF;
81
82     spi0.write_data(0x30 | ch); //Write ltc2600 channel
83
84     spi0.write_data(dat >> 8); //write 16bit data
85     spi0.write_data(dat & 0x00ff);
86
87     LTC2600CS_ON;
88
89     return 0;
90 }
91

```

2. h 文件

```

1 /*
2 * File      : ltc2600.h
3 * Description : This file is ltc2600(8-channel 16bit DAC) driver header.
4 * Author    : XiaomaGee.
5 * Copyright  : Harbin WAVETEK. Co.,Ltd.
6 *
7 * History
8 * -----
9 * Rev   : 0.00
10 * Date  : 05/30/2009
11 *
12 * create.
13 * -----
14 * Rev   : 0.01
15 * Date  : 06/01/2009
16 *
17 * Fix bugs.
18 * -----
19 */
20
21 #ifndef __LTC2600_H__
22 #define __ltc2600_h__
23
24 //-----Include files-----//
25 #include "..\include\hardware.h"
26
27 //-----Defines-----//
28 //cs
29 #define LTC2600CS_ON      IO3SET = (1 << LTC2600CS)
30 #define LTC2600CS_OFF    IO3CLR = (1 << LTC2600CS)

```

```

31
32 //channel
33 #define LTC2600_DACA      0
34 #define LTC2600_DACB      1
35 #define LTC2600_DACC      2
36 #define LTC2600_DACD      3
37 #define LTC2600_DACE      4
38 #define LTC2600_DACF      5
39 #define LTC2600_DACG      6
40 #define LTC2600_DACH      7
41
42 #define LTC2600_MAX_DATA      65535
43
44 //errors
45 #define ERROR_LTC2600_DATA_OVERFLOW      -101
46 #define ERROR_LTC2600_CHAN_OVERFLOW      -102
47
48 //-----Type define-----//
49
50 typedef struct {
51     int (* initialize)(void);
52     int (* write_data)(unsigned short int /* channel */, unsigned short int /* dat */);
53 }LTC2600_T;
54
55 //-----Extern-----//
56
57 extern LTC2600_T ltc2600;
58
59 #endif /* __LTC2600_H__ */
60

```

第六章 LED 实验

LED 实验

通过本章，您可以了解到如何通过寄存器形式对硬件进行控制，让大家可以更透彻的看清 NIOS II 开发过程。

本章分为以下几个部分：

- 一、简介
- 二、PIO 模块的构建
- 三、软件编程

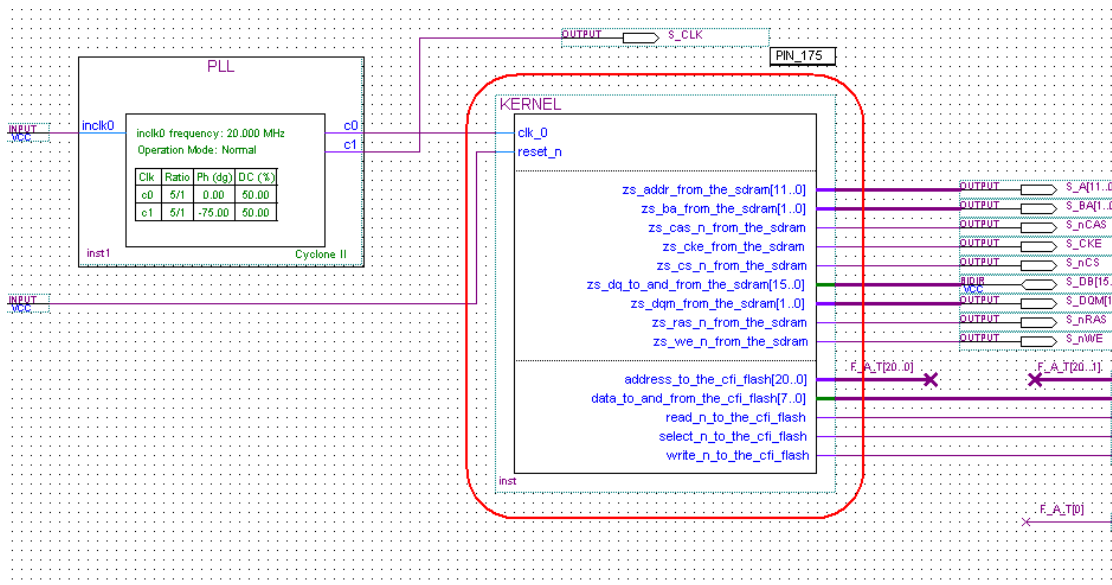
一、简介

这一节,我将给大家讲解第一个与硬件有关的程序,虽然内容简单,却极具代表性。我将采用一种寄存器的操作方式,让大家感受到开发 NIOS 跟单片机一样的简单,看透 NIOS II 开发的本质,尽量避免使用 NIOS II IDE 提供的 API,这样做有很多好处。首先,有单片机开发经验的人很熟悉这种操作方式,其次,可以了解到 NIOS 的本质,真正的去开发它,而不是仅仅用它的 API 来写一些应用程序。

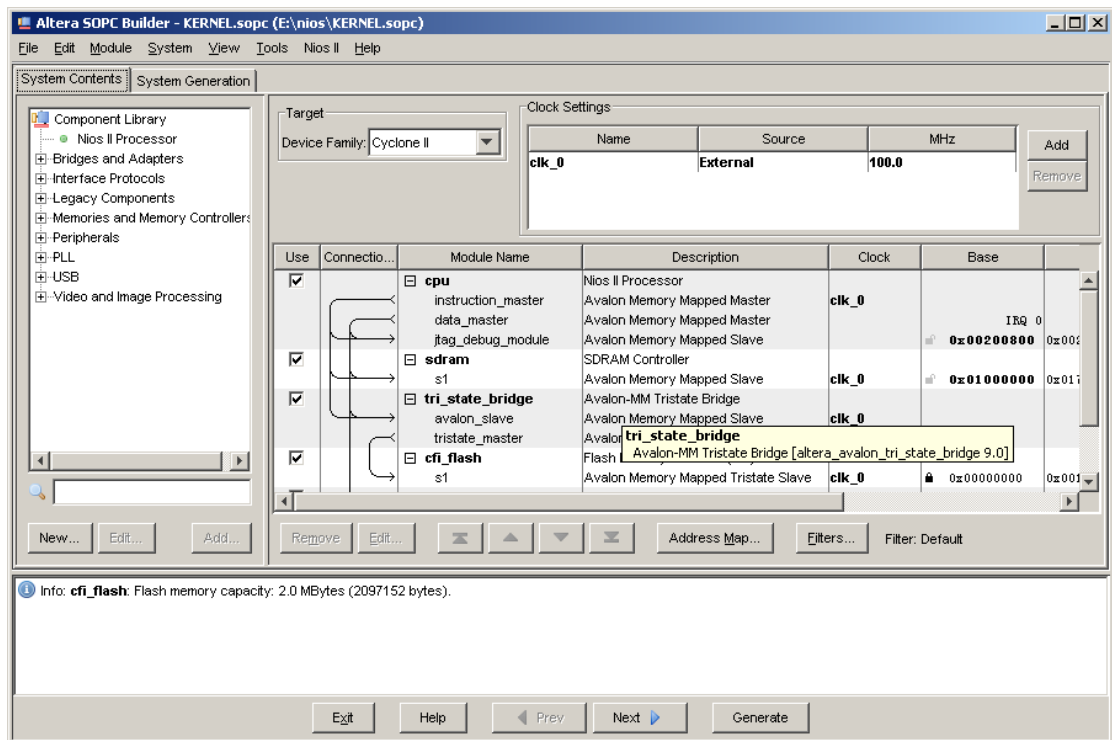
做过单片机试验的人一定对 LED 实验记忆犹新,因为它是硬件试验部分的第一课,通过这个简单的实验,可以让你对单片机的操作有一个感官上的了解,可以说意义不同寻常。这一节,我也通过 LED 实验来带大家进入 NIOS II 的开发世界,感受 NIOS 的魅力所在,下面我们开始吧。

二、硬件开发

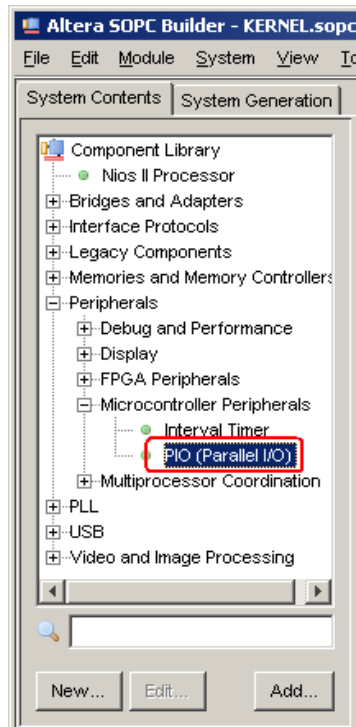
第一步,我们要在软核中加入 PIO 模块。打开我们第一节建的 Quartus 工程,然后双击 KERNEL,如下图红圈所示



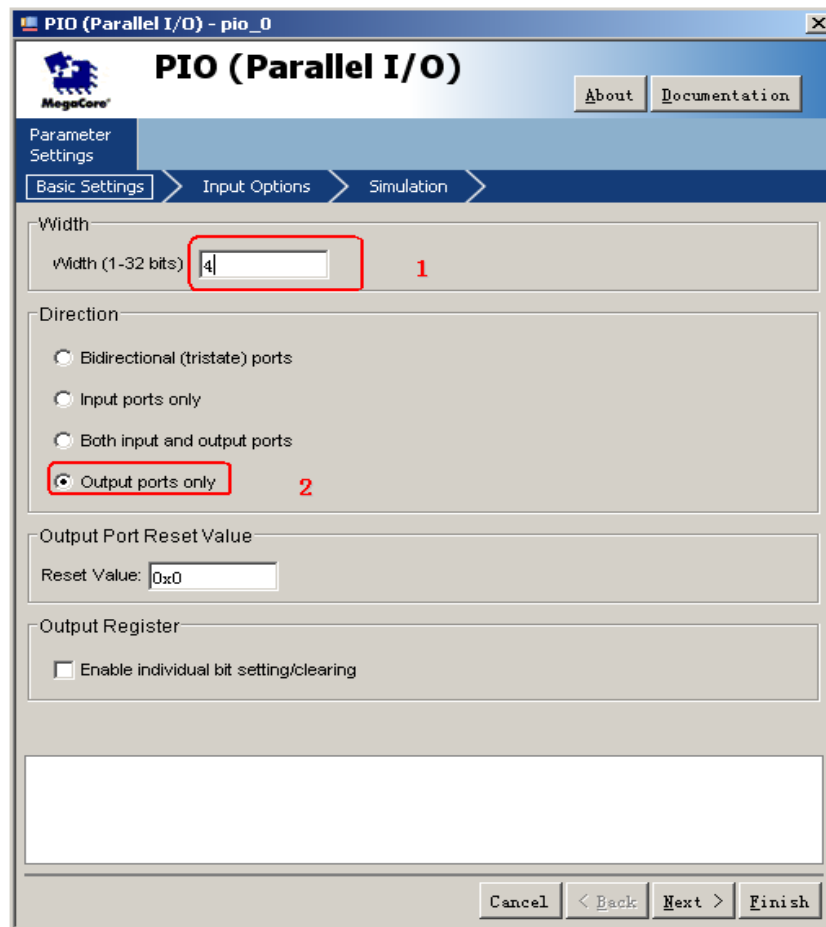
点击后进入了 SOPC BUILDER 界面,如下图所示



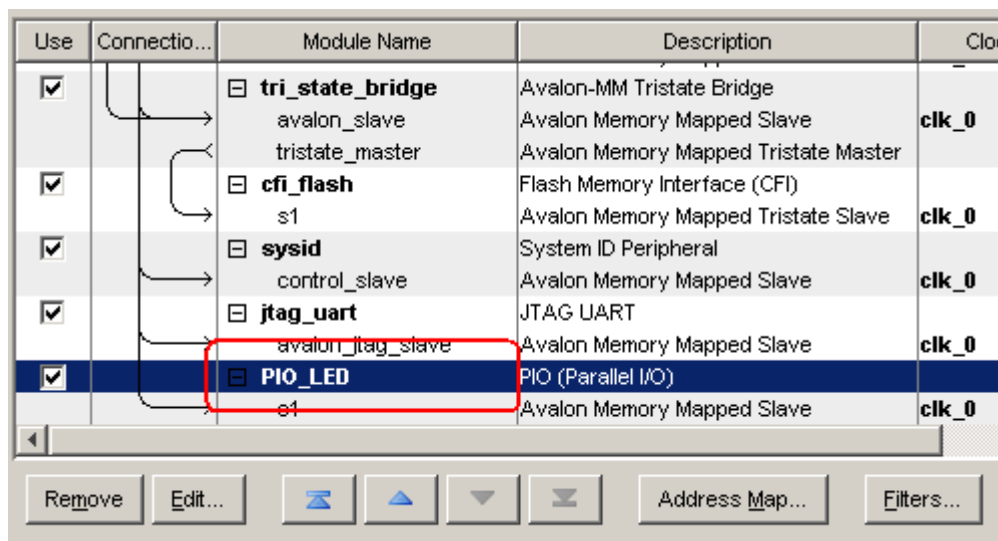
点击下图所示红圈处 PIO(Parallel I/O)



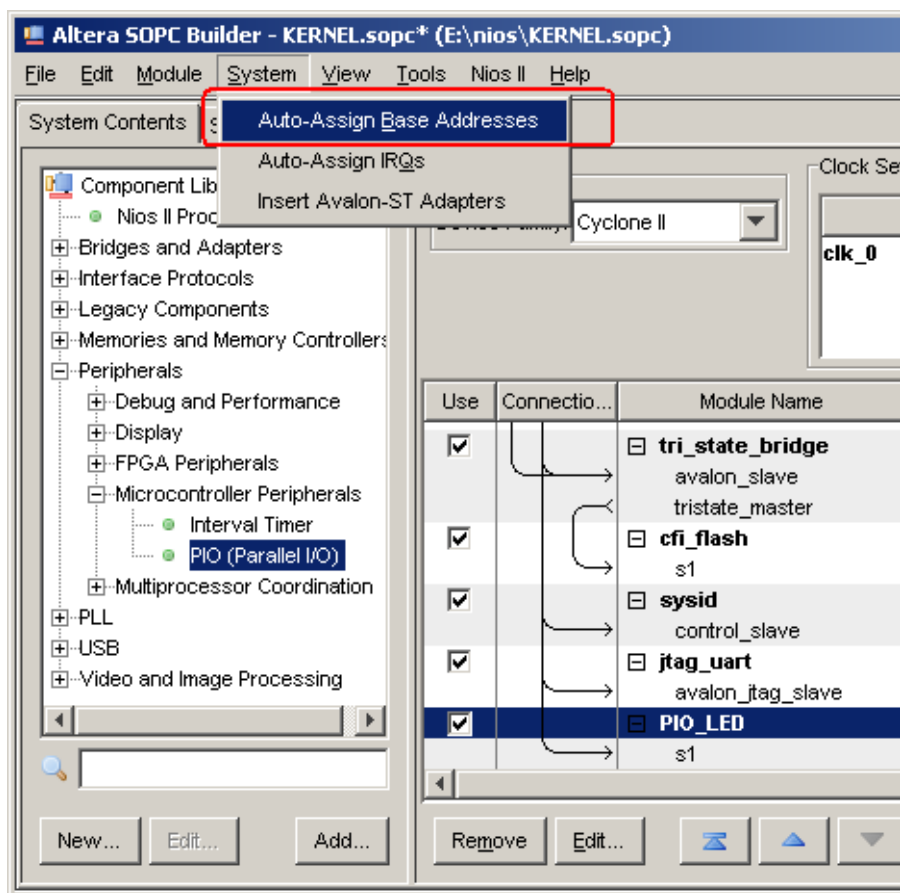
点击后，如下图所示，红圈 1 处是你需要的 PIO 口的宽度，即你需要几个 IO 口，这里面我设置为 4，即我要控制 4 个 LED 红圈 2 是选择输出方式，我选择为输出(Output)。



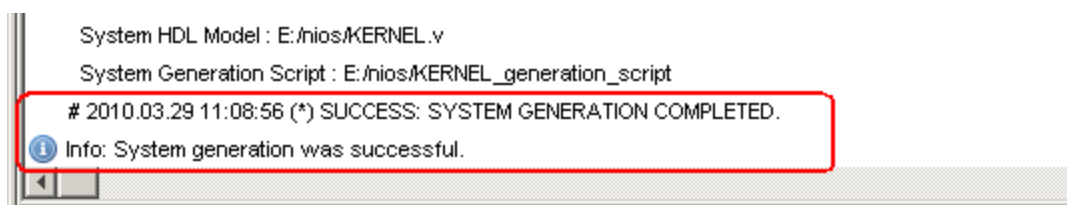
接下来，点击 Finish，完成 PIO 模块的构建，然后将其改名为 PIO_LED,如下图所示



接下来，需要自动分配一下地址，第一节我们已经讲过，如下图所示



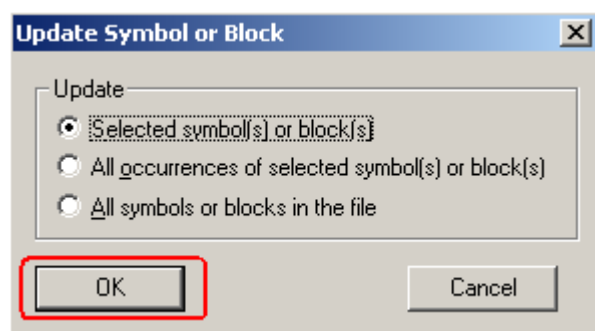
接下来,我们就要开始编译了,点击 Generate,需要保存一下,点击 save,开始编译。经过一段耐心的等待,编译成功,如下图所示



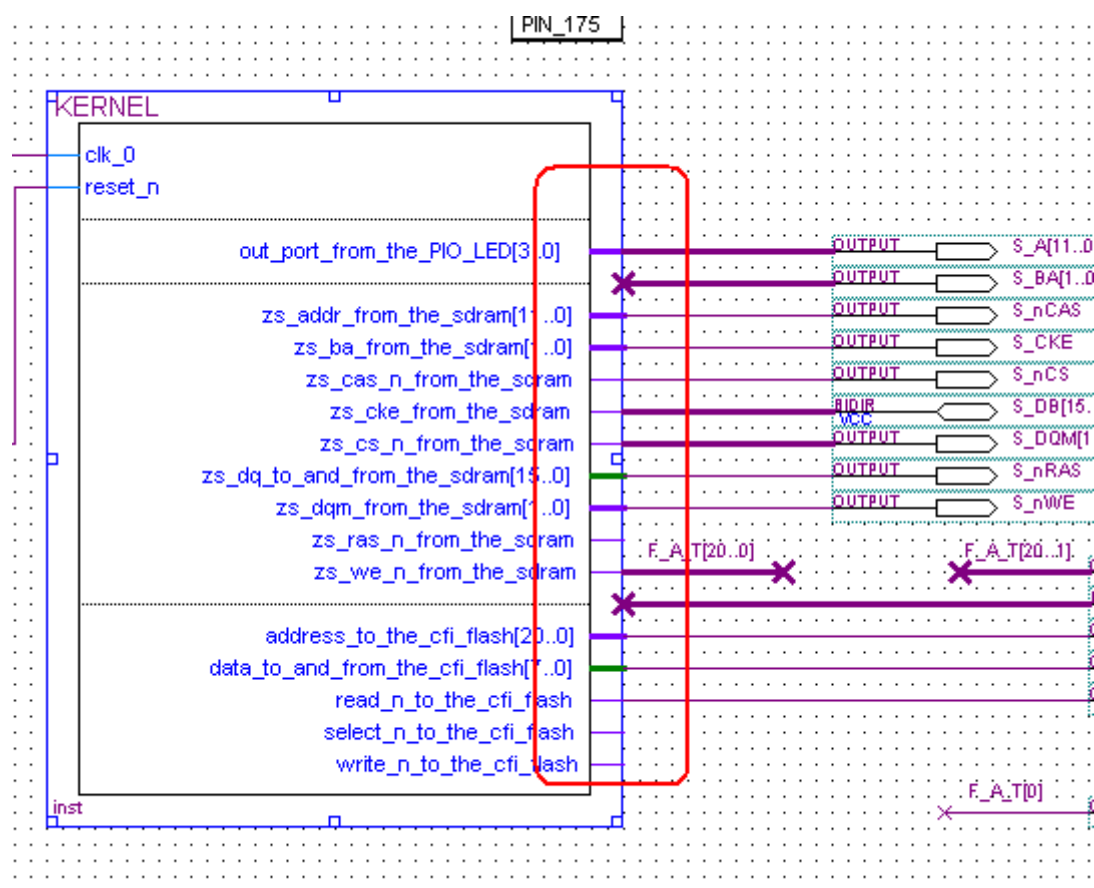
完成了上面的工作,点击 Exit,会出现下面的界面,询问是否需要更新 KERNEL,点击“是(Y)”。



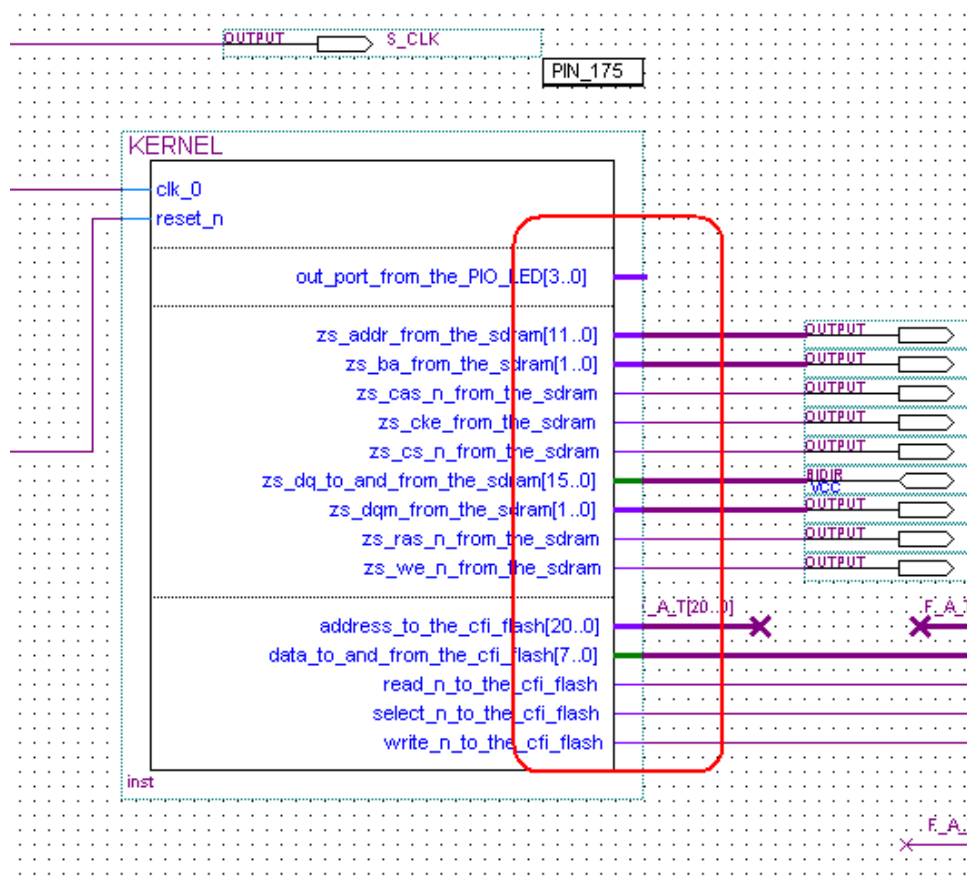
然后,会出现下面界面,点击 OK



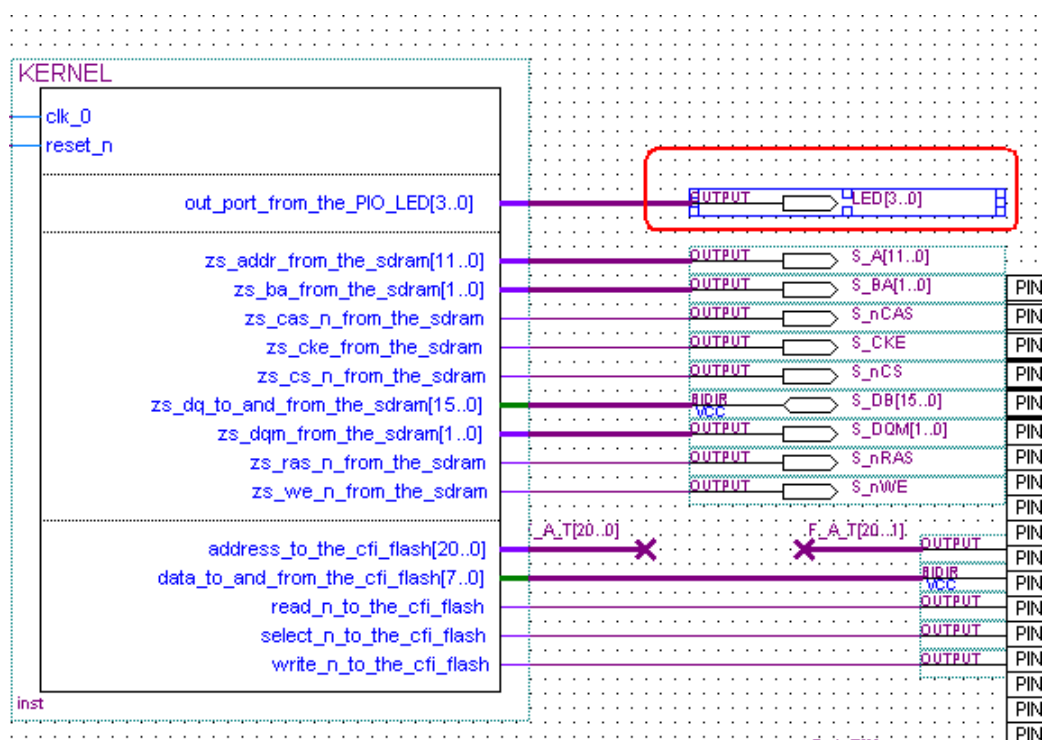
点击后，会出现下面情况，连线错位，这就需要我们手工来整理一下，将相应的管脚相连接。



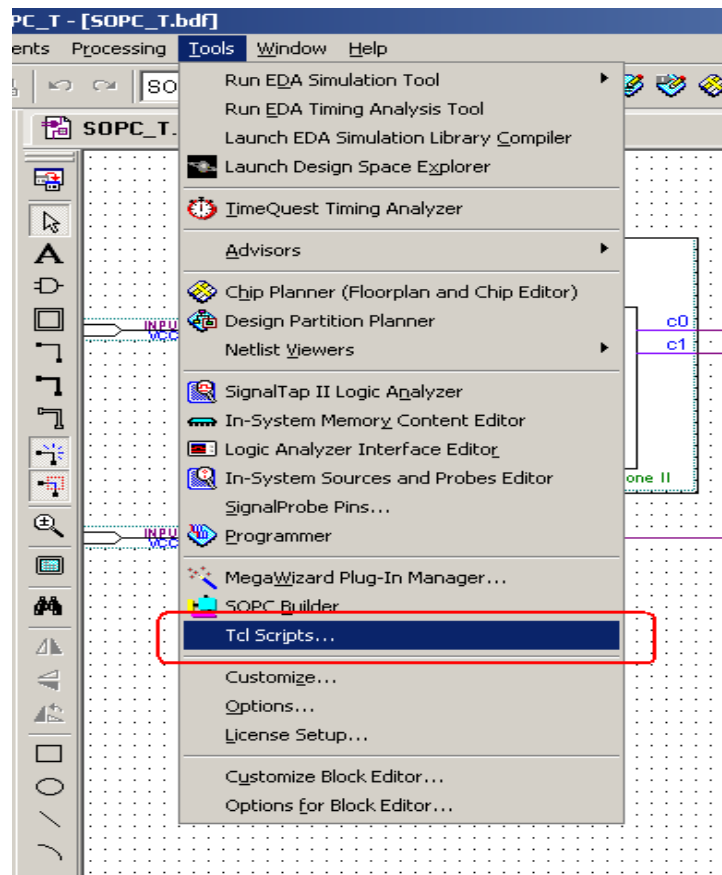
整理还以后，如下图所示，可以看出，还有 PIO_LED 没有管脚相连，我们来建立一个输出管脚,并重命名



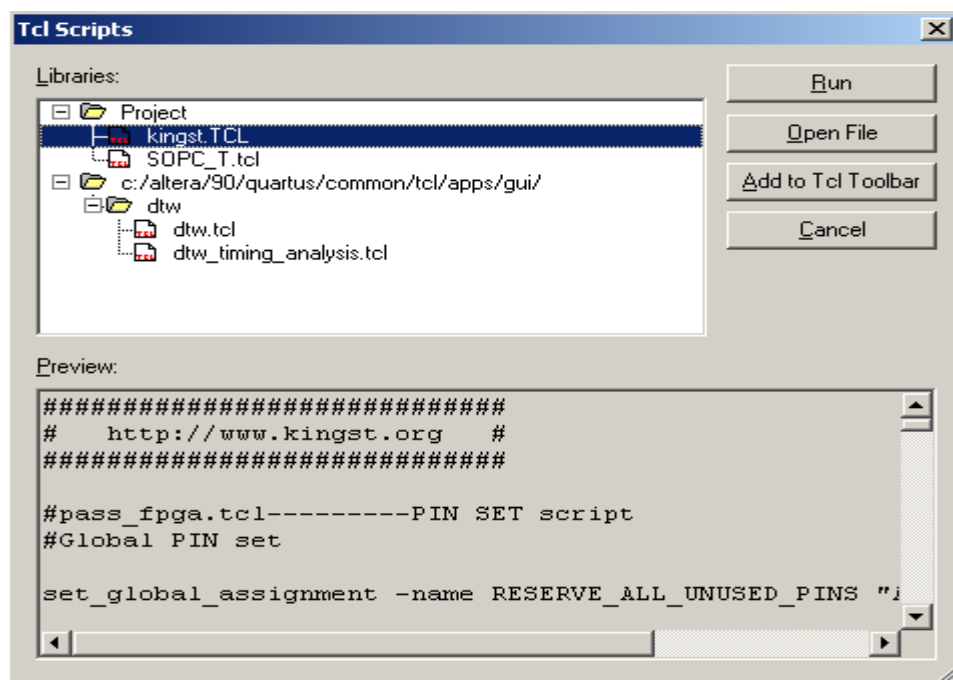
最后的样子大家可以看到，如下图所示，我们将将其命名为 LED[3..0]，这是 quartus 中总线的命名方式，大家要注意。



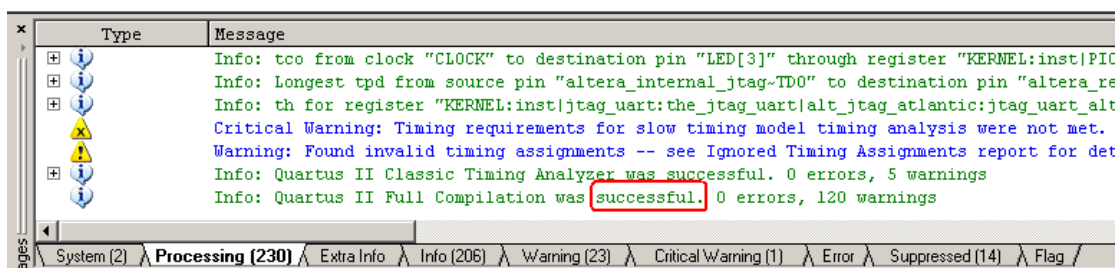
一切准备好了，我们需要给他进行引脚分配，按下图所示



点击 Tcl scripts,如下图所示，选中 kingst.tcl，点击 Run，完成管脚分配。



下面, 我们开始编译了, 又是一次漫长的等待后, 看到了下图红圈所示, 说明编译成功。



我们再来看看我们占用了多少资源, 如下图所示, 还是 66%, 说明一个 PIO 模块占用资源很少的。

Flow Status	Successful - Mon Mar 29 11:28:50 2010
Quartus II Version	9.0 Build 132 02/25/2009 SJ Full Version
Revision Name	SOPC_T
Top-level Entity Name	SOPC_T
Family	Cyclone II
Device	EP2C5Q208C8
Timing Models	Final
Met timing requirements	No
Total logic elements	3,023 / 4,608 (66 %)
Total combinational functions	2,630 / 4,608 (57 %)
Dedicated logic registers	1,704 / 4,608 (37 %)
Total registers	1820
Total pins	76 / 142 (54 %)
Total virtual pins	0
Total memory bits	46,720 / 119,808 (39 %)
Embedded Multiplier 9-bit elements	4 / 26 (15 %)
Total PLLs	1 / 2 (50 %)

到此为止, 我们在软核中添加 PIO 模块已经完成, 接下来, 我们进行的就是在 NIOS II IDE 中开发软件部分。

三、 软件开发

首先打开 NIOS II 9.0 IDE, 打开后, 第一件事就是重新编译整个工程, 按快捷键 Ctrl+b。又一次漫长的等待之后, 我们打开 system.h 文件, 大家能看到有什么变化么, 可能有人发现了, system.h 文件对比之前的多了下面内容, 这部分内容就是我们在软核中新构建的 PIO_LED 模块, 我们将要用到的是 PIO_LED_BASE, 它是 PIO_LED 所有寄存器的首地址。

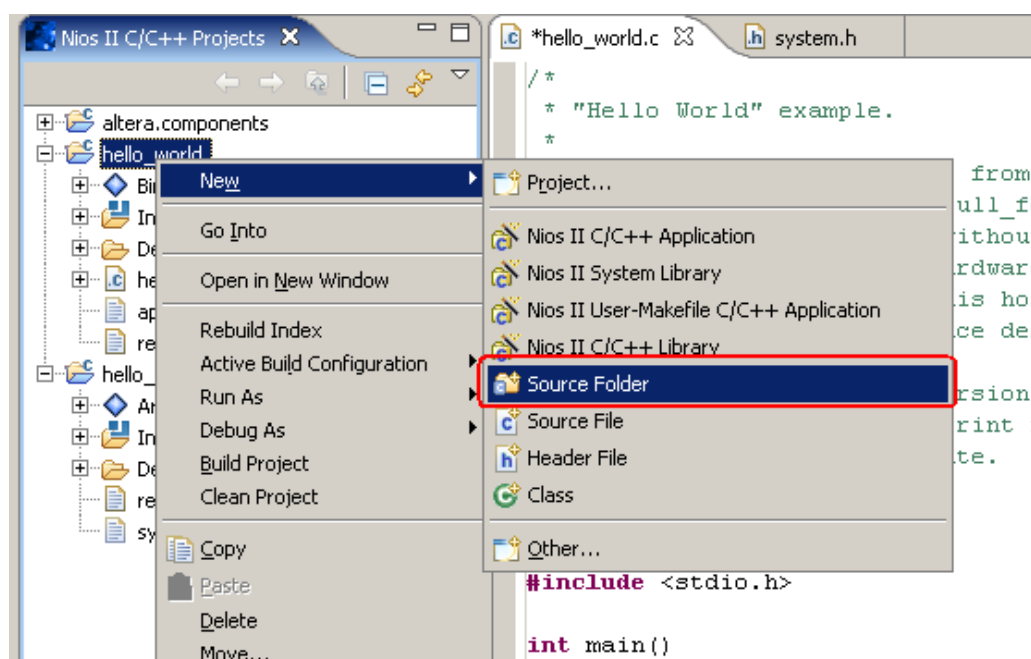
下面, 我们就来运用这个首地址来进行编程, 完成对 4 个 LED 的控制。

```

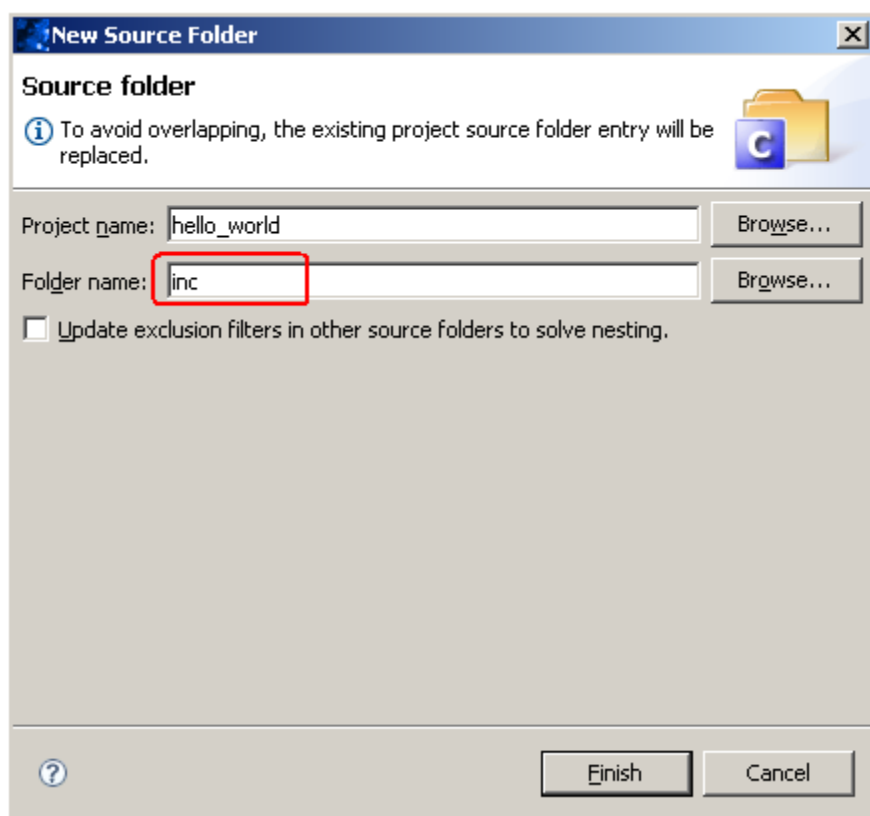
#define PIO_LED_NAME "/dev/PIO_LED"
#define PIO_LED_TYPE "altera_avalon_pio"
#define PIO_LED_BASE 0x00201000
#define PIO_LED_SPAN 16
#define PIO_LED_DO_TEST_BENCH_WIRING 0
#define PIO_LED_DRIVEN_SIM_VALUE 0
#define PIO_LED_HAS_TRI 0
#define PIO_LED_HAS_OUT 1
#define PIO_LED_HAS_IN 0
#define PIO_LED_CAPTURE 0
#define PIO_LED_DATA_WIDTH 4
#define PIO_LED_RESET_VALUE 0
#define PIO_LED_EDGE_TYPE "NONE"
#define PIO_LED_IRQ_TYPE "NONE"
#define PIO_LED_BIT_CLEARING_EDGE_REGISTER 0
#define PIO_LED_BIT_MODIFYING_OUTPUT_REGISTER 0
#define PIO_LED_FREQ 100000000
#define ALT_MODULE_CLASS_PIO_LED altera_avalon_pio

```

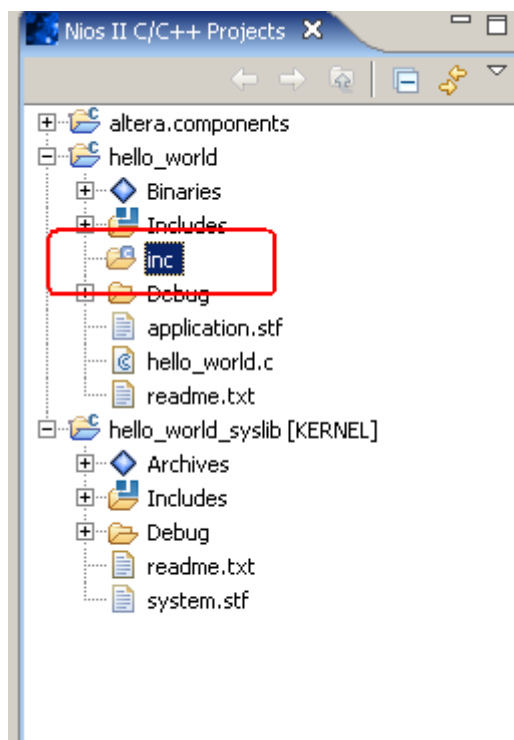
第一步，我需要建立一个放置头文件的文件夹，我将其命名为 inc，方法如下图所示，鼠标在 hello_world 上点击右键，然后点击 New 中的 Source Folder，这样就建立了一个文件夹，



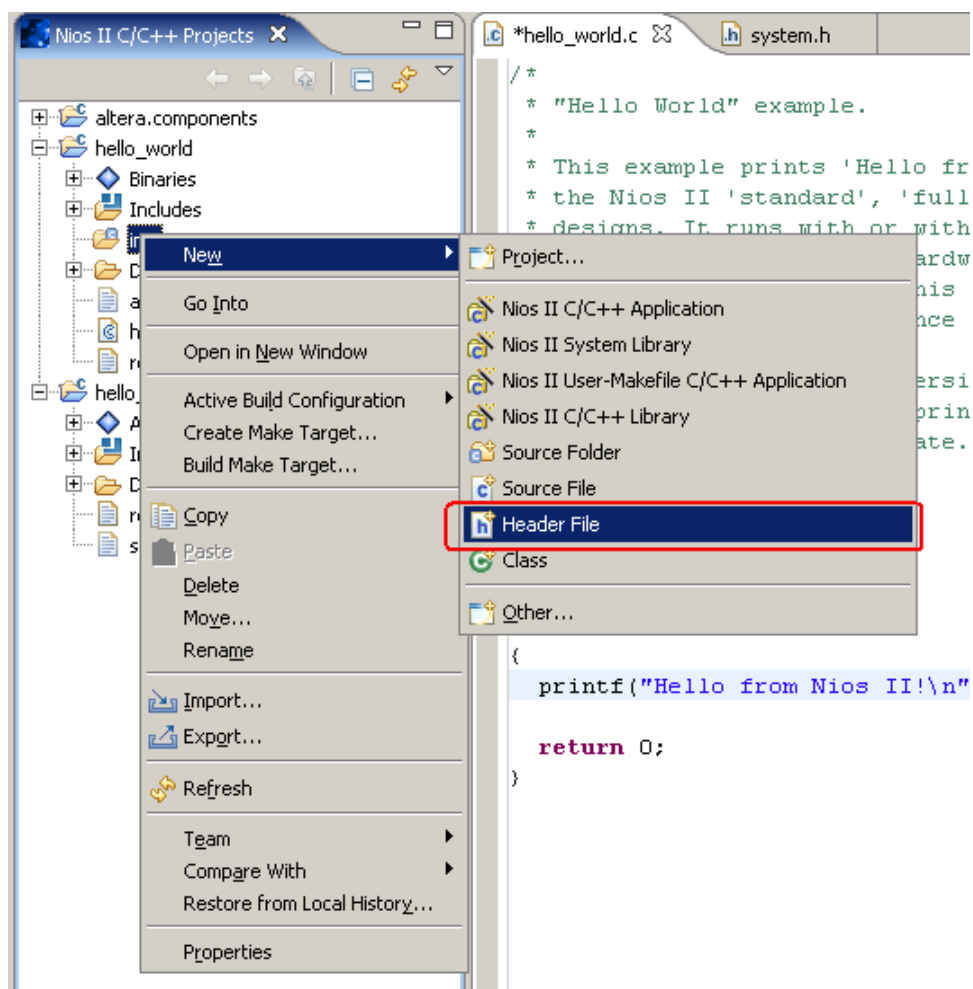
然后会出现下图，在红圈处输入 inc，点击 Finish，完成文件夹建立。



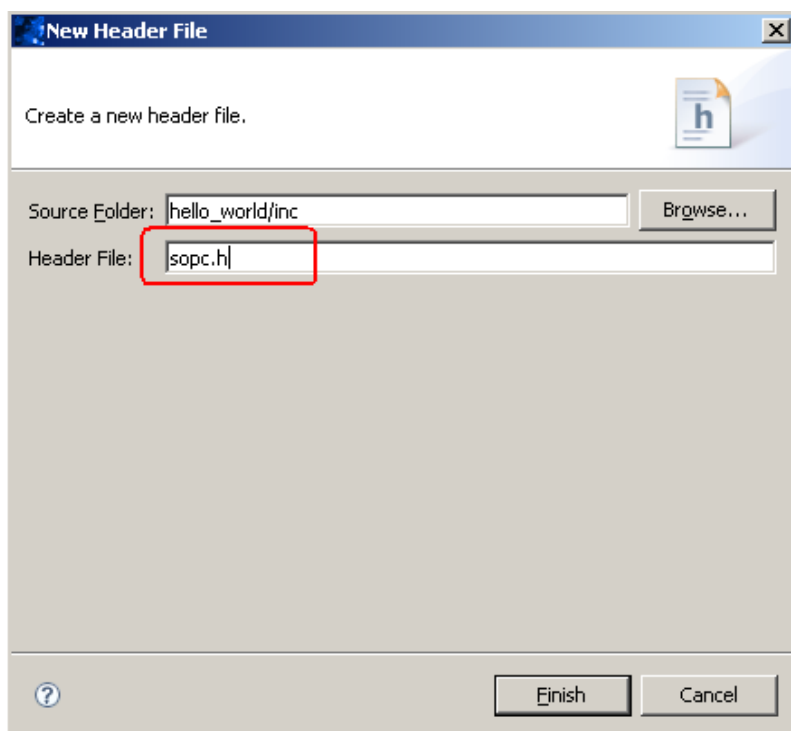
大家可以看出，在左边工程目录栏中多出了一个 inc 文件夹，如下图所示



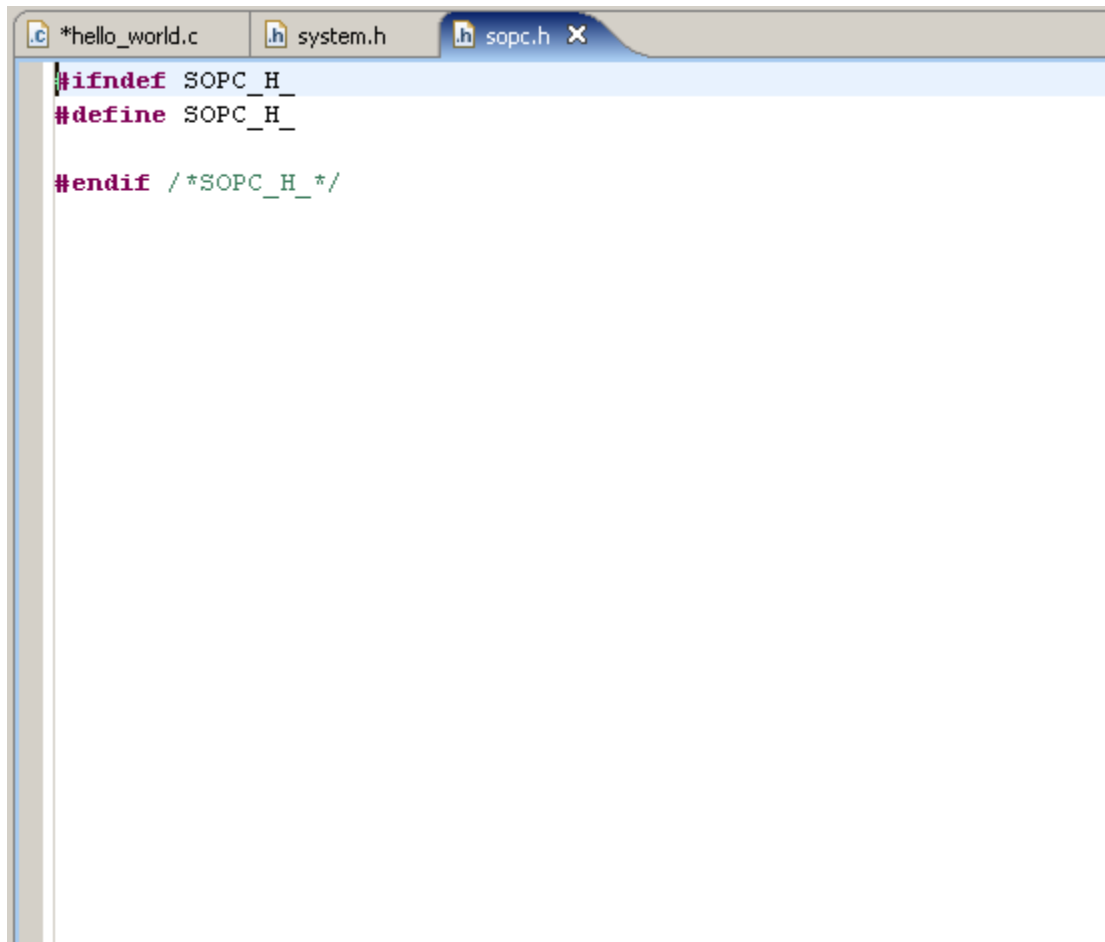
接下来，我要在 inc 中建立一个头文件，方法如下图所示，在 inc 上点击右键，然后点击 New 中的 Header File，红圈处所示



点击以后，出现下图，在红圈处输入 `sopc.h`，点击 Finish，完成头文件的构建。



完成上面的工作以后，代码栏处，可以看到下图，现在我们就对其进行编辑



The screenshot shows a code editor window with three tabs: *hello_world.c, system.h, and sopc.h. The sopc.h tab is active, displaying the following code:

```
#ifndef SOPC_H_
#define SOPC_H_

#endif /*SOPC_H_*/
```

修改后的代码如下图所示，

```
*hello_world.c  system.h  *sopc.h X
#ifndef SOPC_H_
#define SOPC_H_

#include "system.h"

#define _LED 1

typedef struct
{
    unsigned long int DATA;
    unsigned long int DIRECTION;
    unsigned long int INTERRUPT_MASK;
    unsigned long int EDGE_CAPTURE;
} PIO_STR;

#ifdef _LED
#define LED ((PIO_STR *)PIO_LED_BASE)
#endif

#endif /*SOPC_H_*/
```

大家可能看到上面的代码不明白什么意思，这很正常，第一次接触 NIOS 的人不是很了解这个东西，我现在就给大家详细的讲解一下。

首先我们看红圈 2 处的结构体，在这里，我们定义了一个结构体，将其命名为 PIO_STR，这个结构体的结构是有说法的，它是根据芯片手册来写的。我现在给大家截个图，这个图来自《n2cpu_Embedded Peripherals.pdf》，版本是下图所示

Quartus II Handbook Version 8.1

Volume 5: Embedded Peripherals

其中 9-5 页和 9-6 中也这么有一个表格，如下图所示

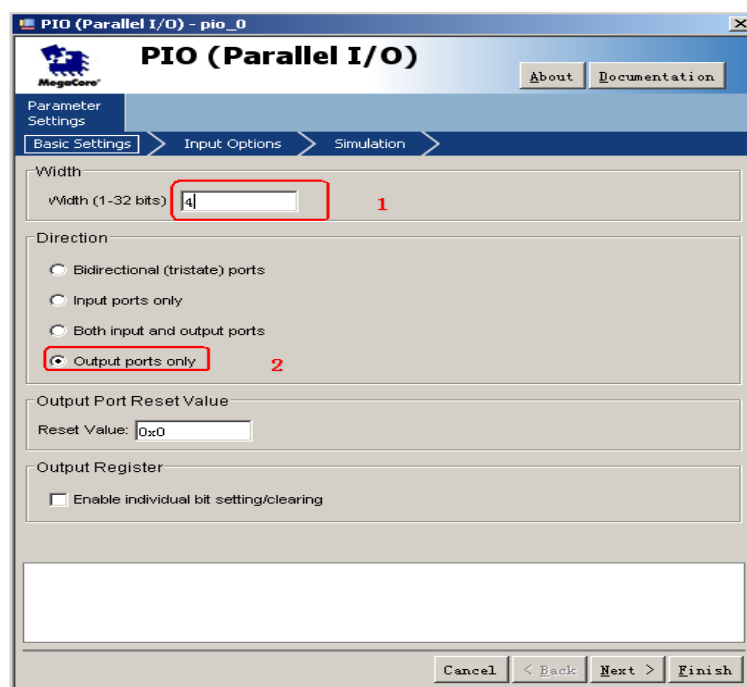
Table 9-2. Register Map for the PIO Core (Part 1 of 2)

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				

Table 9-2. Register Map for the PIO Core (Part 2 of 2)

Offset	Register Name		R/W	(n-1)	...	2	1	0
1	direction (1)		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	interruptmask (1)		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture (1), (2)		R/W	Edge detection for each input port.				

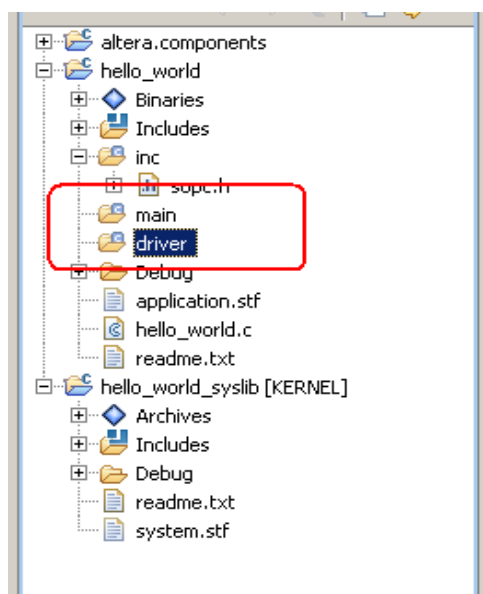
这个表格就是 PIO Core 寄存器映射，我的结构体就是根据它来写的，名字不重要，重要的是它的顺序，也就是偏移量（offset）。第一项是数据 data，第二项是 IO 口方向，第三项是中断控制位，第四项是边沿控制位。他们每一项都有 n 位，这个 n 就是我们在软核构建中的宽度（width），如下图所示的红框 1，我们之前设置为 4，那么 n 等于 4。



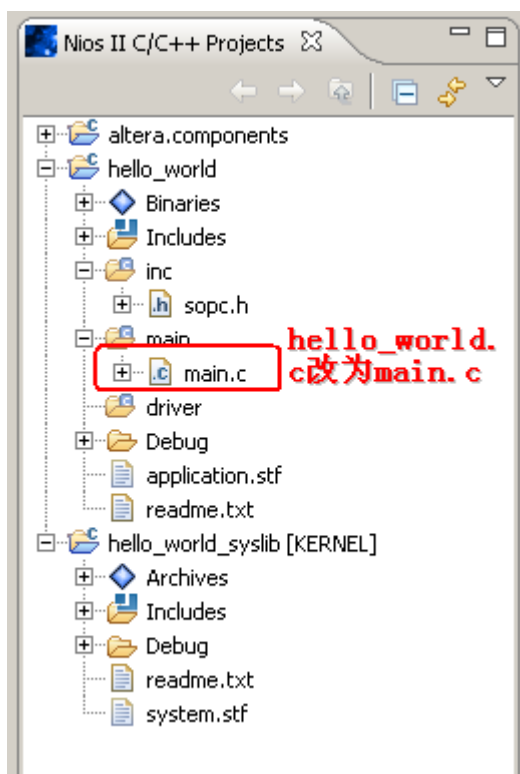
接下来，我们来讲红框 3 中的代码，这部分代码中你一定记得 `PIO_LED_BASE`，对，这就是我之前在 `system.h` 中特意强调的，`PIO_LED` 的基地址。那我红圈 3 中的代码意图就很明显了，定义了一个宏，命名为 `LED`，它是指向 `PIO_LED_BASE` 的结构体指针，这个结构体就是 `PIO_STR`（如果大家对结构体指针不理解，那你就得回去看 C 语言书了，我在这就不具体说了）。红圈 1 的代码也很简单，是为了控制红圈 3 的定义的，这样做是为了增强代码的严谨性和可控制性。当你没有定义 `PIO_LED_BASE` 时，你就可以将红圈 1 中的宏定义 `#define _LED` 去掉，不多说了，再多说就挨砖块了，呵呵。

大家理解了 `sopc.h` 代码以后，我们就可以进行 C 代码编程了。忘了说了，在 NIOS II IDE 中一定要记得修改以后保存，它很幼稚的，编译前不会提醒你去保存的，如果你忘记了保存，相当于你没有修改。好了，下面我们进行 C 代码编程了。

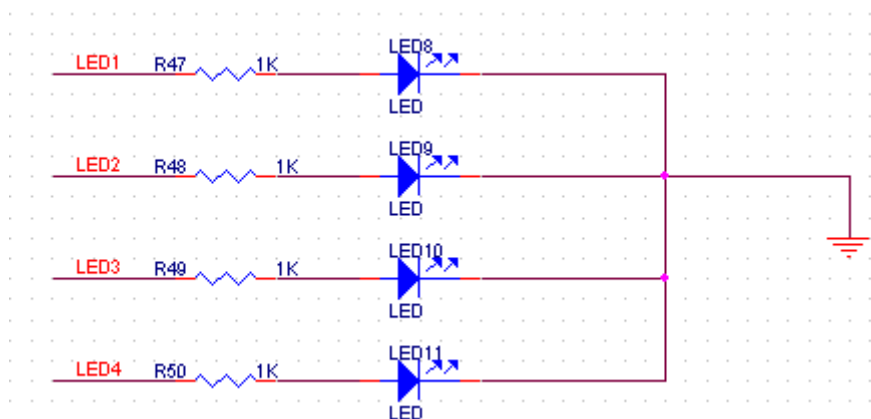
为了规范化程序，我需要对程序做一些调整，以便以后的讲解。首先建立两个文件夹，分别命名为 `driver`，`main`。建好以后，如下图所示，



然后将 `hello_world.c` 名字改为 `main.c`，并将它放到 `main` 文件夹中。改好以后，如下图所示

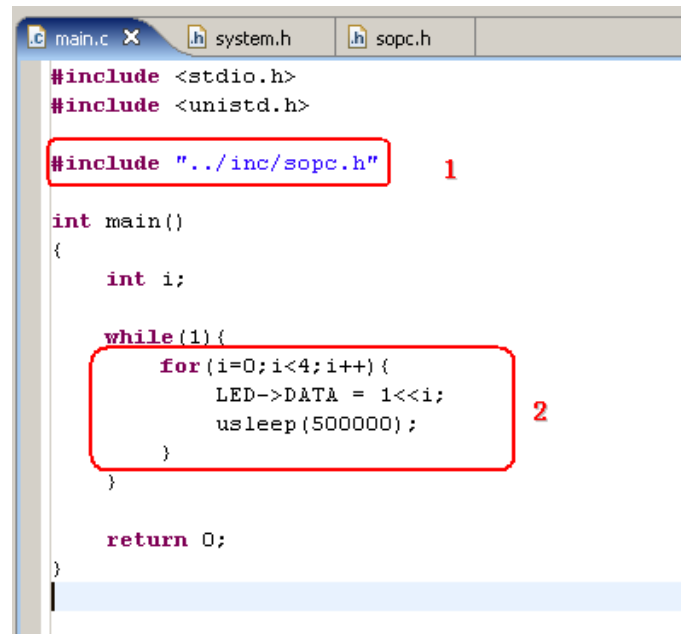


接下来，我们来修改 main.c 中的程序，我先介绍一下我这段代码的目的，这段代码就是控制我的黑金开发板的 4 个 LED，让他们按顺序闪烁，就是大家习惯说的流水灯。下图就是我们要控制的 4 个 LED，当 FPGA 将相应的管脚置 1 时，LED 就会亮，当置 0，LED 就不会亮了。



修改好的程序如下图所示，首先说红框 1，有些人可能没用过这种方式，这是一种相对路径的方式调用头文件，好处就是可移植性好，不管工程放到其他什么地方，这个头文件和 c 文件的相对位置都不会变，也就不需要对这个地方进行修改。再说红框 2 处的代码，关键是 `LED->DATA = 1<i;`；LED 大家应该记得，LED 是我们定义在 `sopc.h` 中的宏，是结构体指针，那么 `LED->DATA` 是什么意思大家就应该很清楚了，是它的结构体中 `DATA` 的内容（不是地址）。`LED->DATA = 1<i;`；就是对 `LED->DATA` 的四

个位循环置 1。所以，4 个 LED 就会按顺序闪烁了。usleep 是微妙级延时，我们在这里每次延时 0.5ms。



```
main.c x system.h sopc.h
#include <stdio.h>
#include <unistd.h>
#include "../inc/sopc.h" 1

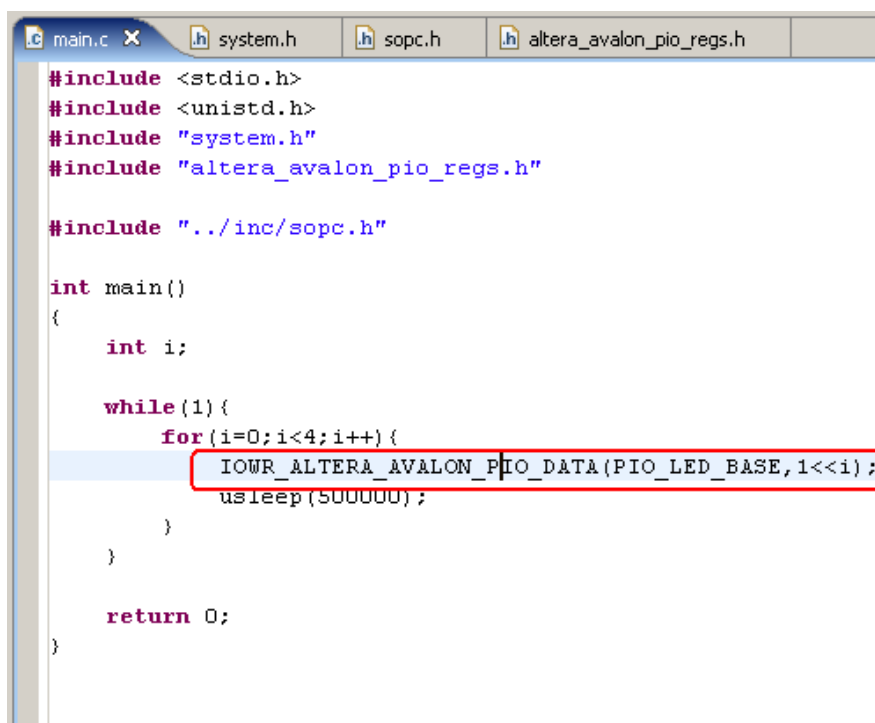
int main()
{
    int i;

    while(1){
        for(i=0;i<4;i++){
            LED->DATA = 1<<i;
            usleep(500000);
        }
    }

    return 0;
}
```

讲到这里，第一个程序就讲完了，下面需要做的就是编译，下载程序到 FPGA，验证是否正确，我就不详细说了。

接下来，我们总结一下这节内容。我们来对比一下寄存器操作方式与 API 之间有什么联系和不同，上面的程序，如果用 NIOS II IDE 提供的 API 来写，那么如下图所示



```
main.c x system.h sopc.h altera_avalon_pio_regs.h
#include <stdio.h>
#include <unistd.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "../inc/sopc.h"

int main()
{
    int i;

    while(1){
        for(i=0;i<4;i++){
            IOWR_ALTERA_AVALON_PIO_DATA(PIO_LED_BASE, 1<<i);
            usleep(500000);
        }
    }

    return 0;
}
```

IOWR_ALTERA_AVALON_PIO 是一个宏，在 altera_avalon_pio_regs.h 中，其定义如下（大家可以按住 ctrl 键后，用鼠标点击进入定义所在的位置）

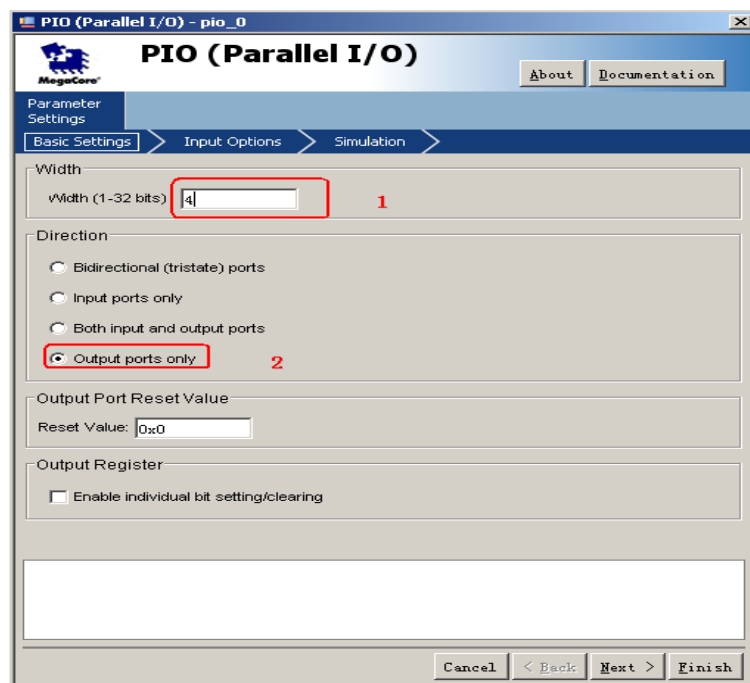
```
#ifndef __ALTERA_AVALON_PIO_REGS_H__
#define __ALTERA_AVALON_PIO_REGS_H__

#include <io.h>

#define IOADDR_ALTERA_AVALON_PIO_DATA(base)      __IO_CALC_ADDRESS_NATIVE(base, 0)
#define IORD_ALTERA_AVALON_PIO_DATA(base)        IORD(base, 0)
#define IOWR_ALTERA_AVALON_PIO_DATA(base, data)  IOWR(base, 0, data)
```

大家可以看到，它是一个 IOWR 的宏，而 IOWR 的具体写法我就在此不详细说了（大家感兴趣的可以去 NIOS 的源码），反正就是对硬件地址的控制。我的做法就是绕过这个大圈子，直接去控制它的寄存器。

大家可能有点纳闷，我们的机构体中定义了四个变量，但只用了 DATA 一个，在这说明一下原因，首先是 DIRECTION。这个是 IO 的方向，就是说是输入还是输出，或者是双向的。因为在我们构建 PIO 模块的构成中有了一个选项 如下图所示 红圈 2，我们选择了输出（Output ports only），也就是我们在底层就已经固定了它的方向，所以在软件中就不需要在定义了。还有两个变量是涉及到中断时才会用到，所以在这个程序中没有用到。



这一节是以后内容的基础，希望大家好好的理解，尤其是 sopc.h 中定义的那个结构体，大家一定要理解清楚。以后，还会涉及到 UART，SPI 等，都需要建立结构体，这个都是触类旁通，举一反三的东西，弄清楚一个，其他的都好理解了。

我要提醒大家一句，我虽然提倡大家用操作寄存器方式编程，但并不希望所有的程序都按这种方式来写，比如说对 flash 的操作，我们就可以使用 API 来写，因为 flash 的操作相对复杂，而利用 API 可以很简单的几个语句就能完成，没必要自己来写。

好了，今天我们就讲到这里，下一节我们将讲一下 NIOS 如何在线调试，以及调试中的一些小技巧，希望大家耐心等待.....

第七章 中断实验

中断实验

通过本章，您可以了解到 NIOS II 如何进行中断处理，以及调试的一些方法和技巧

本章分为以下几个部分：

- 一、简介
- 二、硬件开发
- 三、软件编程

一、简介

这一节，我们来讲解一下 NIOS II 的硬件中断的内容，同时借助这节内容也要介绍 NIOS II IDE 在线调试的方法和技巧。首先来点理论知识，介绍一下与硬件中断相关的内容，让大家对 NIOS II 的硬件中断有一个概括性的了解。

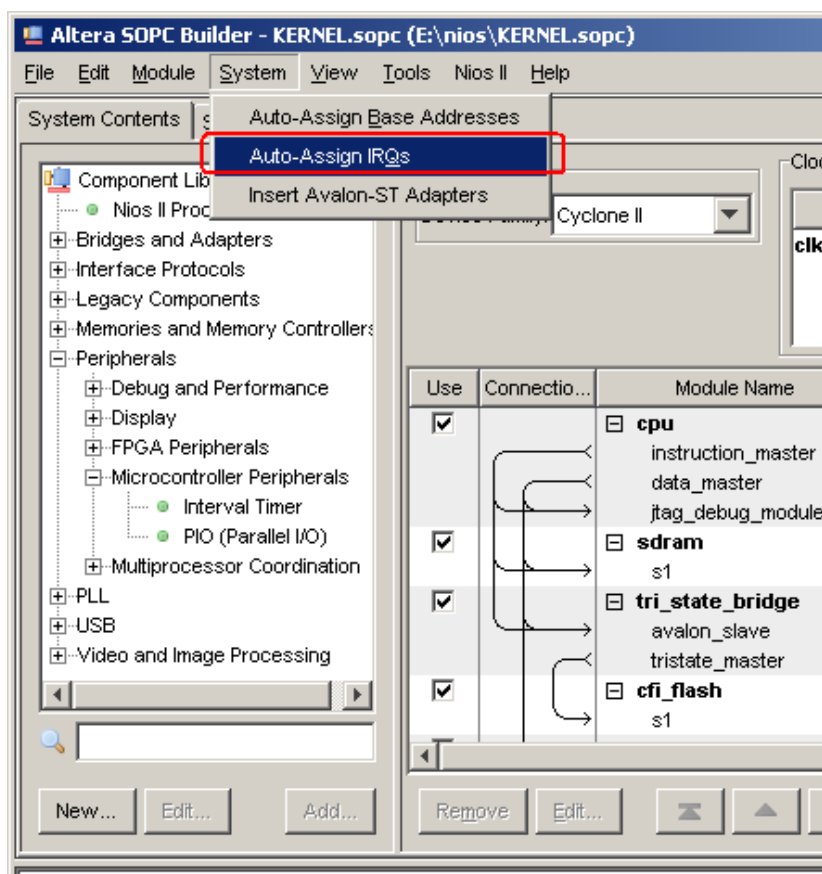
ISR(Interrupt Service Routine)中断服务函数是为硬件中断服务的子程序。NIOS II 处理器支持 32 个硬件中断，每一个使能了的硬件中断都应该有一个 ISR 与之对应。中断发生时，硬件中断处理器会根据检测到的有效中断级别，调用相应的 ISR 为其进行中断服务。

要完成硬件中断工作，我们需要做两件事：

第一，注册中断函数 ISR,它的函数原型如下所示：

```
Int alt_irq_register(alt_u32 id, void* context, void(*handler) (void*,alt_u32));
```

id：中断优先级，即所注册的 ISR 是为哪个中断优先级的中断服务的。中断优先级在 SOPC Builder 中分配的，在第一节中我们提起过，不知道大家是否记得，我们来回忆一下，如下图所示，通过这一步我们来完成中断的自动分配



分配好的 IRQ 如下图所示，当然，IRQ 可根据自己的特殊要求进行手工修改，只要不重复就没问题。

Use	Connectio...	Module Name	Description	Clock	Ba...	End	Tags	IRQ
☒		☐ cpu	Nios II Processor					
		instruction_master	Avalon Memory Mapped Master	clk_0				
		data_master	Avalon Memory Mapped Master		I...	I...		
		jtag_debug_module	Avalon Memory Mapped Slave			0...		
☒		☐ sdram	SDRAM Controller					
		s1	Avalon Memory Mapped Slave	clk_0		0...		
☒		☐ tri_state_bridge	Avalon-MM Tristate Bridge					
		avalon_slave	Avalon Memory Mapped Slave	clk_0				
		tristate_master	Avalon Memory Mapped Tristate Master					
☒		☐ cfi_flash	Flash Memory Interface (CFI)					
		s1	Avalon Memory Mapped Tristate Slave	clk_0		0...		
☒		☐ sysid	System ID Peripheral					
		control_slave	Avalon Memory Mapped Slave	clk_0		0...		
☒		☐ jtag_uart	JTAG UART					
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_0		0...		
☒		☐ PIO_LED	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0		0...		

而在 NIOS II IDE 软件中体现在 system.h 文件中，之前我们也提到过，如下图所示，看到了吧，JTAG_UART_IRQ 为 0，与上图的中断号正好相同，这可不是巧合。我已经说过了，system.h 文件是根据软核来生成的，所以 JTAG_UART_IRQ 等于 0 也是可想而知的了。

```

#define JTAG_UART_NAME "/dev/jtag_uart"
#define JTAG_UART_TYPE "altera_avalon_jtag_uart"
#define JTAG_UART_BASE 0x00201018
#define JTAG_UART_SPAN 8
#define JTAG_UART_IRQ 0
#define JTAG_UART_WRITE_DEPTH 64
#define JTAG_UART_READ_DEPTH 64
#define JTAG_UART_WRITE_THRESHOLD 8
#define JTAG_UART_READ_THRESHOLD 8
#define JTAG_UART_READ_CHAR_STREAM ""
#define JTAG_UART_SHOWASCII 1
#define JTAG_UART_READ_LE 0
#define JTAG_UART_WRITE_LE 0
#define JTAG_UART_ALTERA_SHOW_UNRELEASED_JTAG_UART_FEATURES 0
#define ALT_MODULE_CLASS_jtag_uart altera_avalon_jtag_uart

```

接下来我们继续说剩下两个参数，

Context，为所注册的 ISR 传递参数，可以是 NULL；

Handler，中断服务函数 ISR 的指针。

再来说返回值，返回值是 0 时，表示中断注册成功；返回为负数，表明中断注册失败。

这里面有一个需要注册的地方，如果 handler 不是 NULL，则该优先级中断在注册成功后将自动使能，也即是说，只要我们在 handler 处有相应的 ISR，我们就不需要再进行使能处理了。说完第一，我们来说说第二。

编写 ISR 函数，这个函数有我们自己来写，而不是 HAL 系统提供的。它跟一般的函数定义没什么区别，只是对 ISR 的函数原型有特定的要求：

```
void ISR_handler( void* context, alt_u32 id );
```

context: 传给 ISR 的形参，可以是 NULL;

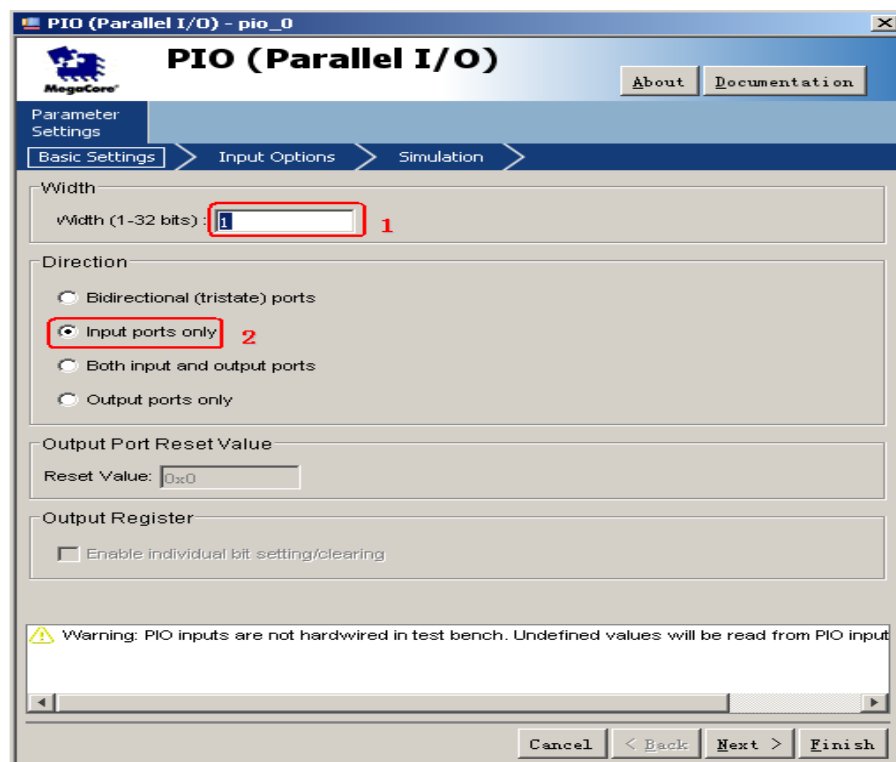
id: 中断优先级。

OK，只要这两步我们就可以完成中断函数的处理了。废话少说，我们来点实际的吧，跟我来。下面，我们就利用一个外部按键来验证一下中断函数的处理过程。

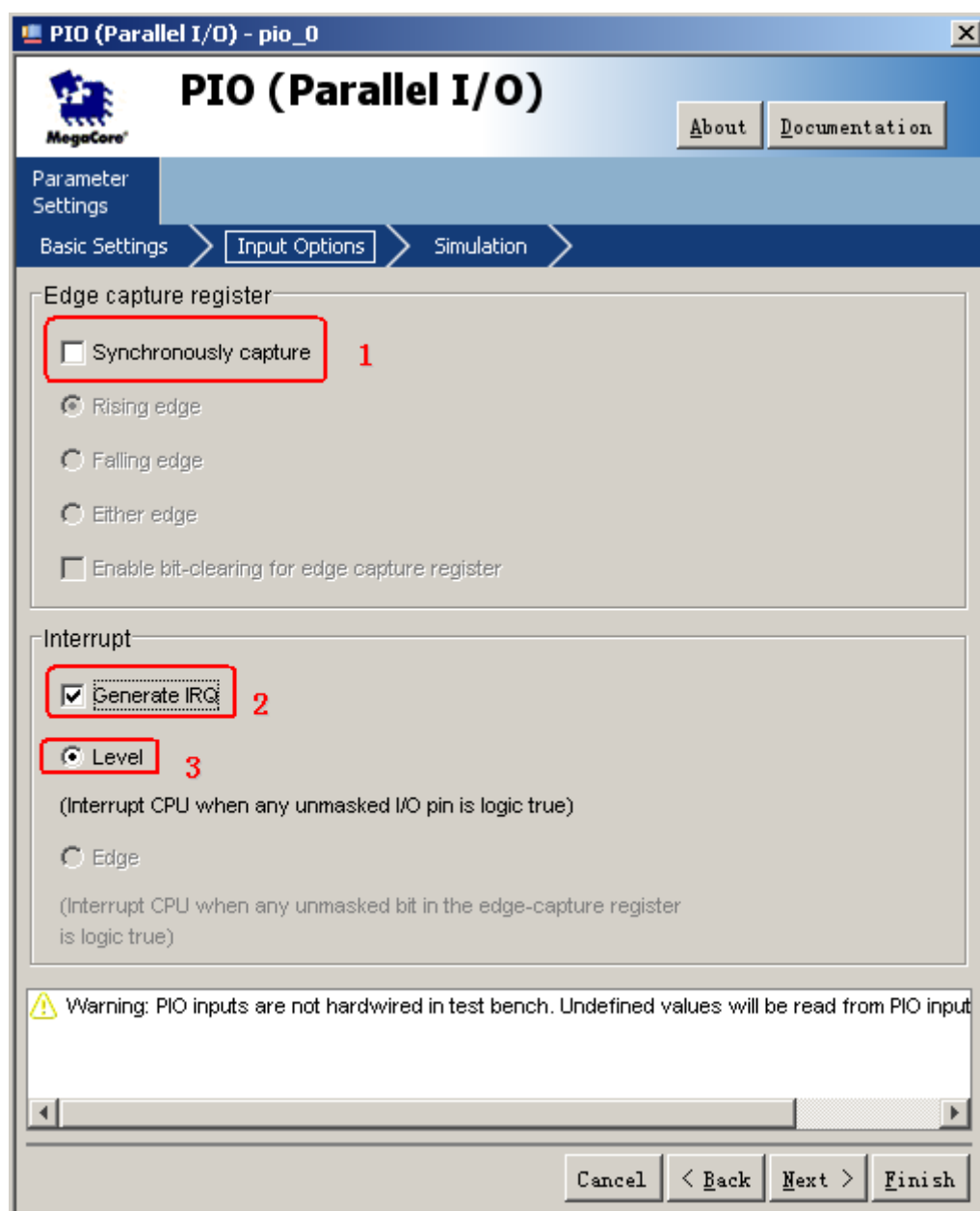
二、 硬件开发

首先，我们要构造一个给外部按键用的 PIO 模块。这部分内容之前已经详细的讲过了，在此简述略过吧。

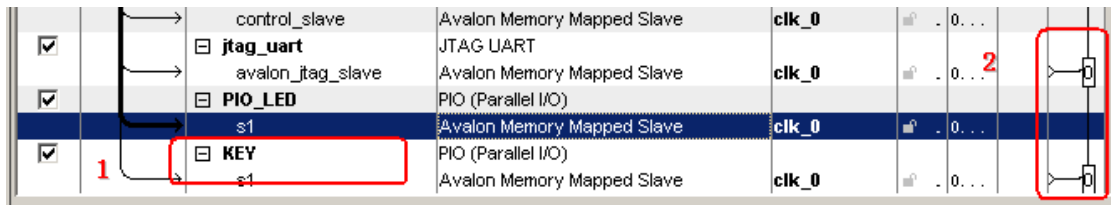
打开 Quartus II 软件，然后双击 KERNEL，进入 SOPC BUILDER。进入后，我们建立一个 PIO 模块，在建立过程中有一个地方有所不同，我们来看一下，如下图所示，红圈 1 处，我们输入 1，因为我们只需要一个按键（我们的黑金开发板中一共有 5 个按键）；红圈 2 处选择 Input ports only，也就是作为输入。完成后，点击 Next，



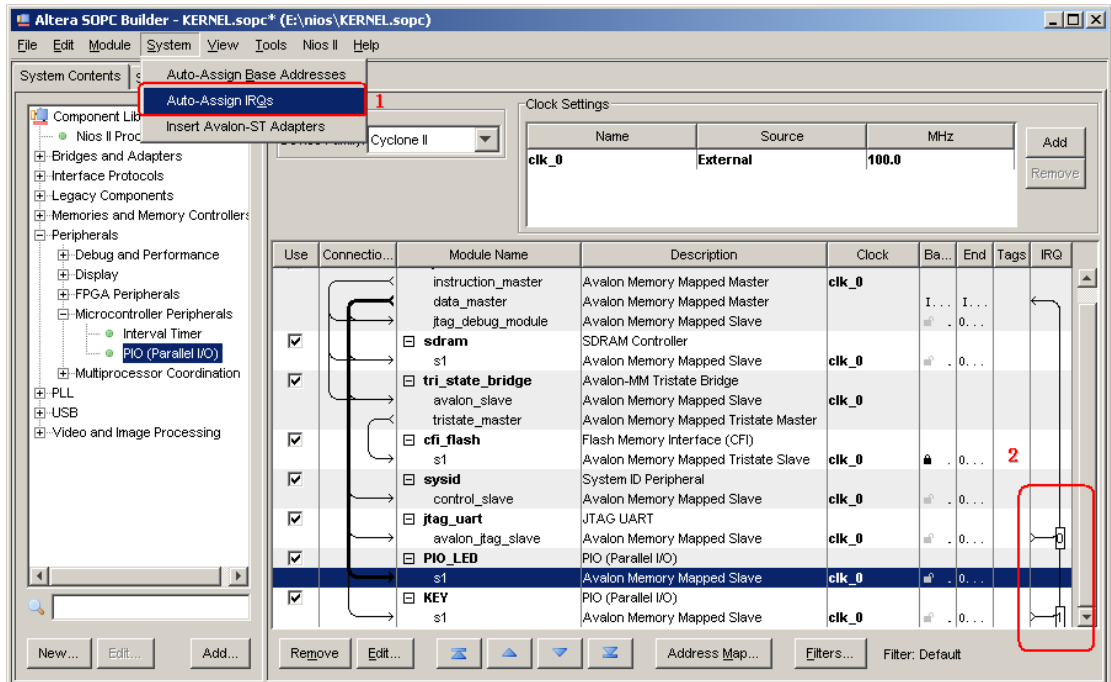
点击后，如下图所示，我们将红圈 2（General IRQ）处选中，其他不变。在这多说两句，这里的中断分为两种，一种是电平（Level）中断，也就是高电平/低电平中断，还有一种就是沿中断，包括上升沿、下降沿。做过单片机的人应该都很熟悉，如果你想要实现沿中断，需要把红圈 1（Synchronously capture）处选中，下面包括 3 种方式，大家可以根据自己的要求选择。完成上面工作以后，点击 Finish，完成 PIO 构建。



下面需要对名字需要进行修改，我将其命名为 KEY，如下图红圈 1 处所示。大家再看一下红圈 2 处，大家可以发现中断号有两个 0，一个是 jtag_uart 的，一个是我们新建立的 KEY 的，这回大家明白了，为什么我们需要自动分配中断号了吧，出现了相同的中断优先级了。

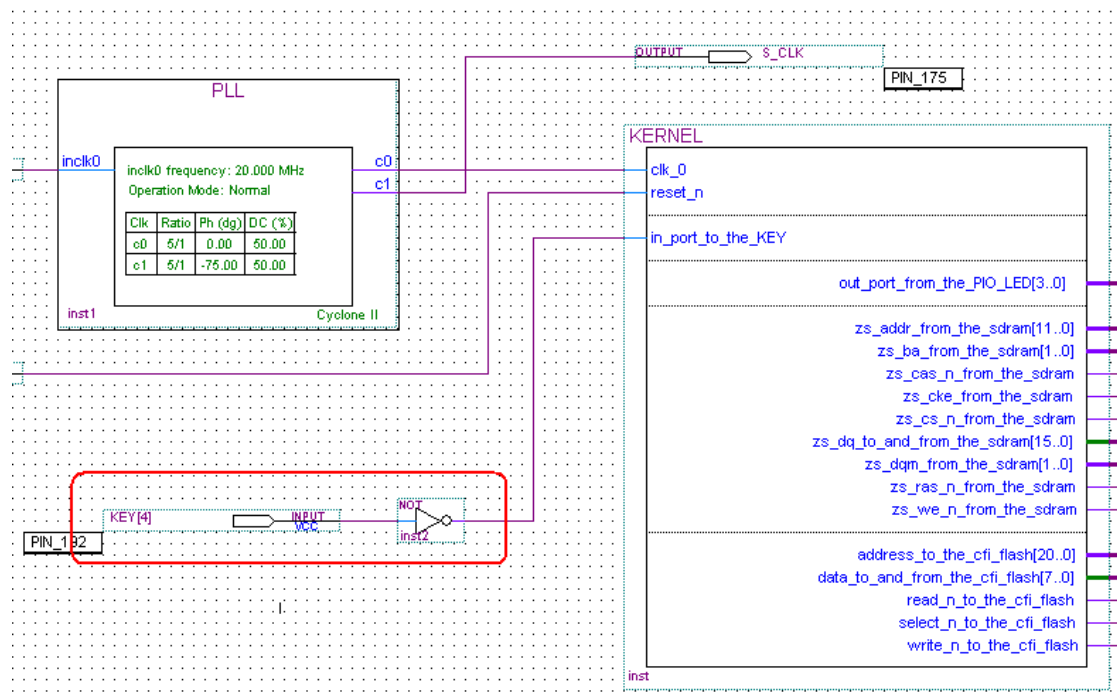


接下来，我们就对中断自动分配一下。如下图所示，大家可以看到红圈 2 处发生变化了吧，KEY 的中断级别变化了，变成了 1。自动中断分配是自上而下按顺序分配的。

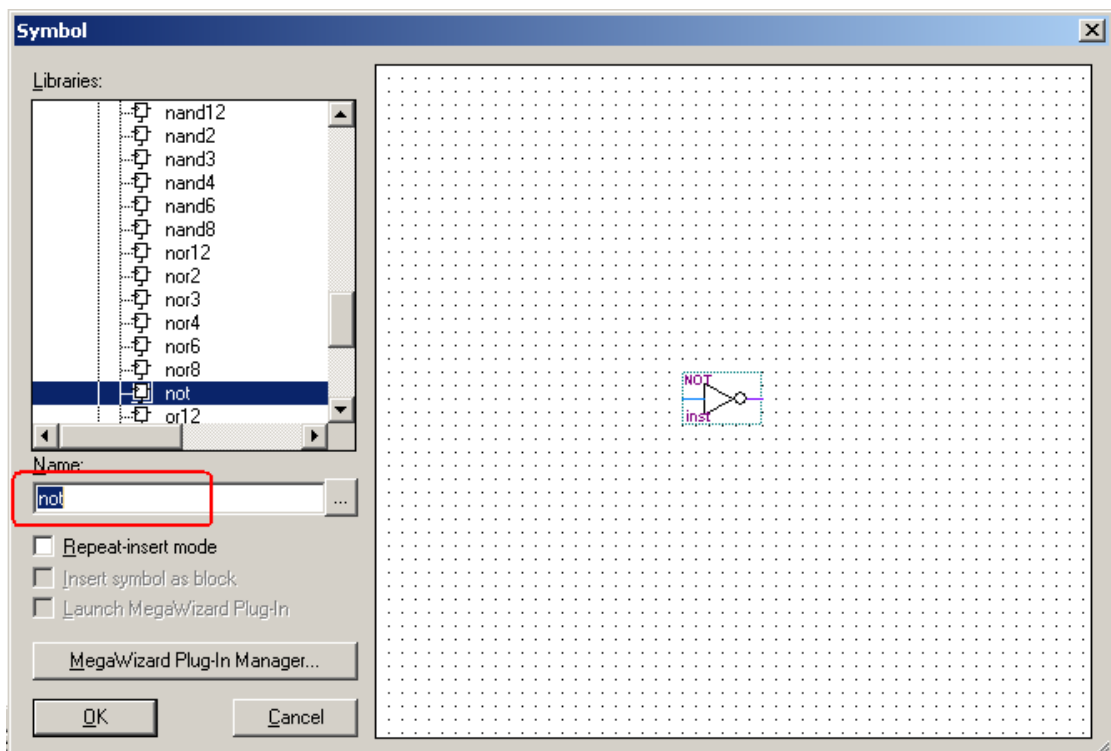


完成上面的构建以后开始编译了，又需要漫长的等待了.....（其实也不是很长，也就一两分钟吧，根据电脑配置而定）

编译好了以后，回到 Quartus 界面，需要进行整理，添加 KEY 的输入管脚，我这里就简要略过了。看一下整好以后的样子，如下图所示，我将其命名为 KEY[4]，这个对应的是我的黑金开发板的中间的那个按键。下图有一个地方需要注意，由于电平中断时，NIOS 只对高电平敏感，所以如果想实现低电平敏感需要加一个非门，



非门的加入方式跟加入 input 管脚是一样的，双击空白处出现下图，然后在红圈处输入 not，点击 OK 即可。



然后，分配引脚，编译，又是等待.....

编译好以后，咱们再来看看用了多少资源，如下图所示，还是 66%，

Flow Status	Successful - Tue Mar 30 14:28:32 2010
Quartus II Version	9.0 Build 132 02/25/2009 SJ Full Version
Revision Name	SOPC_T
Top-level Entity Name	SOPC_T
Family	Cyclone II
Device	EP2C5Q208C8
Timing Models	Final
Met timing requirements	No
Total logic elements	3,027 / 4,608 (66 %)
Total combinational functions	2,650 / 4,608 (58 %)
Dedicated logic registers	1,710 / 4,608 (37 %)
Total registers	1826
Total pins	77 / 142 (54 %)
Total virtual pins	0
Total memory bits	46,720 / 119,808 (39 %)
Embedded Multiplier 9-bit elements	4 / 26 (15 %)
Total PLLs	1 / 2 (50 %)

咱们进行对比一下，在上一节我们占用资源如下图所示

Flow Status	Successful - Mon Mar 29 11:28:50 2010
Quartus II Version	9.0 Build 132 02/25/2009 SJ Full Version
Revision Name	SOPC_T
Top-level Entity Name	SOPC_T
Family	Cyclone II
Device	EP2C5Q208C8
Timing Models	Final
Met timing requirements	No
Total logic elements	3,023 / 4,608 (66 %)
Total combinational functions	2,630 / 4,608 (57 %)
Dedicated logic registers	1,704 / 4,608 (37 %)
Total registers	1820
Total pins	76 / 142 (54 %)
Total virtual pins	0
Total memory bits	46,720 / 119,808 (39 %)
Embedded Multiplier 9-bit elements	4 / 26 (15 %)
Total PLLs	1 / 2 (50 %)

对比以后发现，PIO 模块占用资源是相当的少。

编译好以后，大家可以选择是通过 AS 模式或 JTAG 模式下载都可以，不过记得 JTAG 模式掉电会丢失，记得上电以后重新烧写。

硬件部分就讲完了，下面我们来讲软件编程部分的内容。

三、 软件编程

打开 NIOS II 9.0 IDE 软件，打开后，先编译一次，快捷键为 CTRL+b（本人比较喜欢快捷键，方便快捷么，呵呵）。编译好以后，我们来看看 system.h 有什么变化。大家可以看到，system.h 中多出来了下面内容，这部分内容就是有关 KEY 的，其中红圈 1 处是基地址，红圈 2 处是中断号，这些都是我们在下面要用到的，大家先有个印象。

```
#define KEY_NAME "/dev/KEY"
#define KEY_TYPE "altera avalon_pio"
#define KEY_BASE 0x00200000 1
#define KEY_SPAN 16
#define KEY_IRQ 1 2
#define KEY_DO_TEST_BENCH_WIRING 0
#define KEY_DRIVEN_SIM_VALUE 0
#define KEY_HAS_TRI 0
#define KEY_HAS_OUT 0
#define KEY_HAS_IN 1
#define KEY_CAPTURE 0
#define KEY_DATA_WIDTH 1
#define KEY_RESET_VALUE 0
#define KEY_EDGE_TYPE "NONE"
#define KEY_IRQ_TYPE "LEVEL"
#define KEY_BIT_CLEARING_EDGE_REGISTER 0
#define KEY_BIT_MODIFYING_OUTPUT_REGISTER 0
#define KEY_FREQ 100000000
#define ALT_MODULE_CLASS_KEY altera_avalon_pio
```

接下来，我们就要开始写程序了，首先需要在 socp.h 中添加内容，如下图所示，跟上一节的 LED 是一样的，在此就不做解释了。

```
#ifndef Sopc_H_
#define Sopc_H_

#include "system.h"

#define LED
#define _KEY 1

typedef struct
{
    unsigned long int DATA;
    unsigned long int DIRECTION;
    unsigned long int INTERRUPT_MASK;
    unsigned long int EDGE_CAPTURE;
} PIO_STR;

#ifdef _LED
#define LED ((PIO_STR *)PIO_LED_BASE)
#endif

#ifdef _KEY
#define KEY ((PIO_STR *)KEY_BASE) 2
#endif

#endif /*Sopc_H_*/
```

修改好以后，我们对 main.c 函数进行更改，整体程序如下图所示，这个程序没有对按键进行防抖处理，只是为了展示外部中断处理的操作过程，如想作为项目中应用必须加入按键防抖处理，在此不具体说明。这个函数通过外部按键来产生中断，因为我们设置的是低电平产生中断，所以当按键按下时，就会产生了一个低电平，这时就会进入中断函数。在中断函数中，我们对 key_flag 进行取反。而在主函数中，我们不断地进行查询，当 key_flag 为 1 时，LED->DATA 置 1，也就是让外部发光二极管亮，当 key_flag 为 0 时，LED->DATA 置 0，这时，发光二极管不亮。

接下来我们具体讲解一下每一个函数。

```
#include "../inc/sopc.h"
#include "system.h"
#include "sys/alt_irq.h"
#include <unistd.h>
#include <stdio.h>

int key_flag = 0;

void ISR_key(void * context,unsigned long id)
{
    key_flag = ~key_flag;
}

int init_key(void)
{
    KEY->INTERRUPT_MASK = 1;

    return alt_irq_register(KEY_IRQ,NULL,ISR_key);
}

int main()
{
    if(!init_key()){
        printf("register successfully!\n");
    }
    else{
        printf("Error: register failure!\n");
    }

    while(1){
        if(key_flag){
            LED->DATA = 1;
        }
        else{
            LED->DATA = 0;
        }
    }

    return 0;
}
```

首先编译一个初始化程序，如下图所示，一共有两条语句，我们先说红圈 1 处的语句，这个语句的目的是，使能中断位，上一节我已经讲过了，PIO 模块对应的结构体中的 INTERRUPT_MASK 是中断控制寄存器的内存映射，当该位置 1 时，允许中断，否则，禁止中断。再说红圈 2 处的语句，我们前面已经见过这个语句了，用它来完成中断的注册，KEY_IRQ 来自 system.h，ISR_key 是 ISR 函数。用 return 返回为了在主函数中判断注册是否成功，如果成功返回 0，非 0 表示注册失败。

```
int init_key(void)
{
    KEY->INTERRUPT_MASK = 1; 1
    return alt_irq_register(KEY_IRQ, NULL, ISR_key); 2
}
```

下面来看看 ISR_key 函数，如下图所示，只有一条语句，其中，key_flag 是一个全局变量，这条语句的意思，没进一次中断，就将 key_flag 取反。

```
void ISR_key(void * context, unsigned long id)
{
    key_flag = ~key_flag;
}
```

下面是主函数，红圈 1 处部分是判断初始化是否成功，如果 init_key() 函数返回值是 0，说明注册成功，打印 register successfully!\n，可以再观察栏中看到它，否则打印 Error: register failure!\n。红圈 2 处是判断标志位 key_flag，如果它为 1，则对 LED->DATA 置 1，也就是让对应的 LED 亮，否则 LED 不亮。

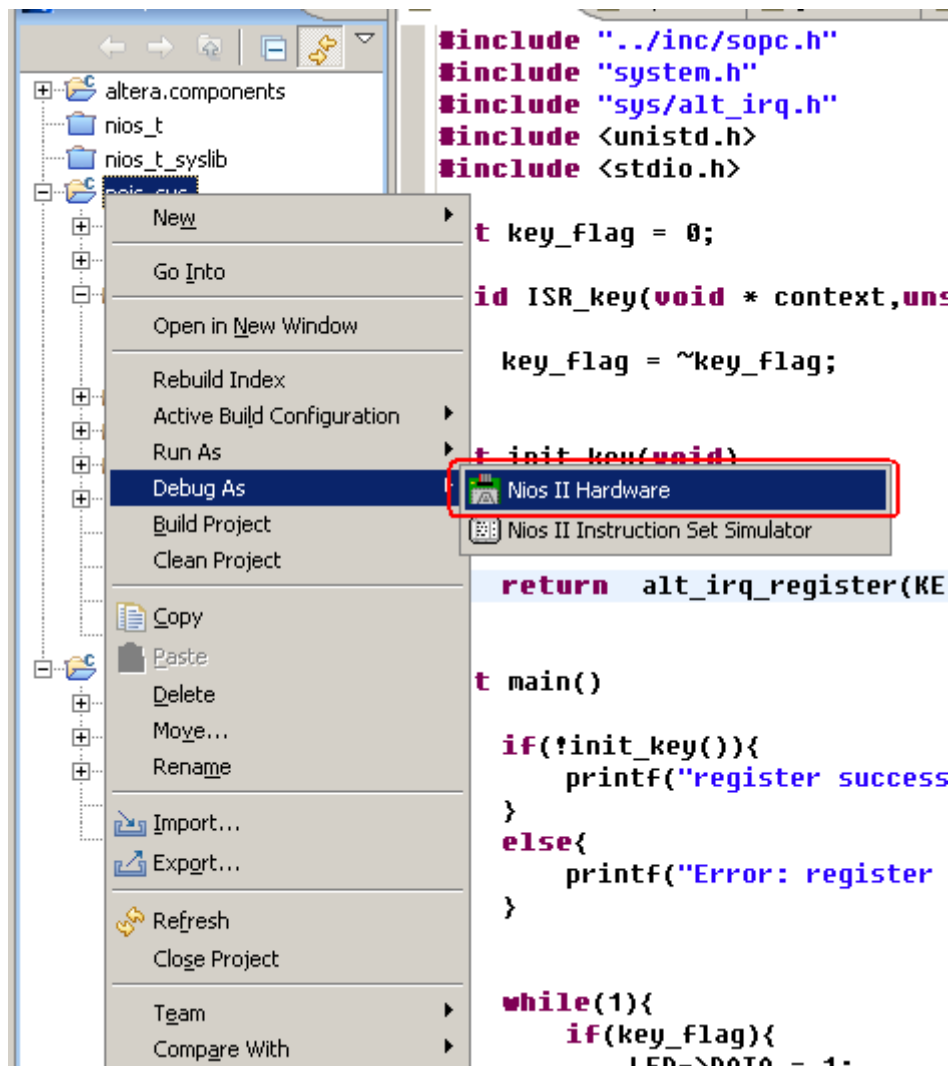
```
int main()
{
    if(!init_key()){ 1
        printf("register successfully!\n");
    }
    else{
        printf("Error: register failure!\n");
    }

    while(1){
        if(key_flag){
            LED->DATA = 1; 2
        }
        else{
            LED->DATA = 0;
        }
    }

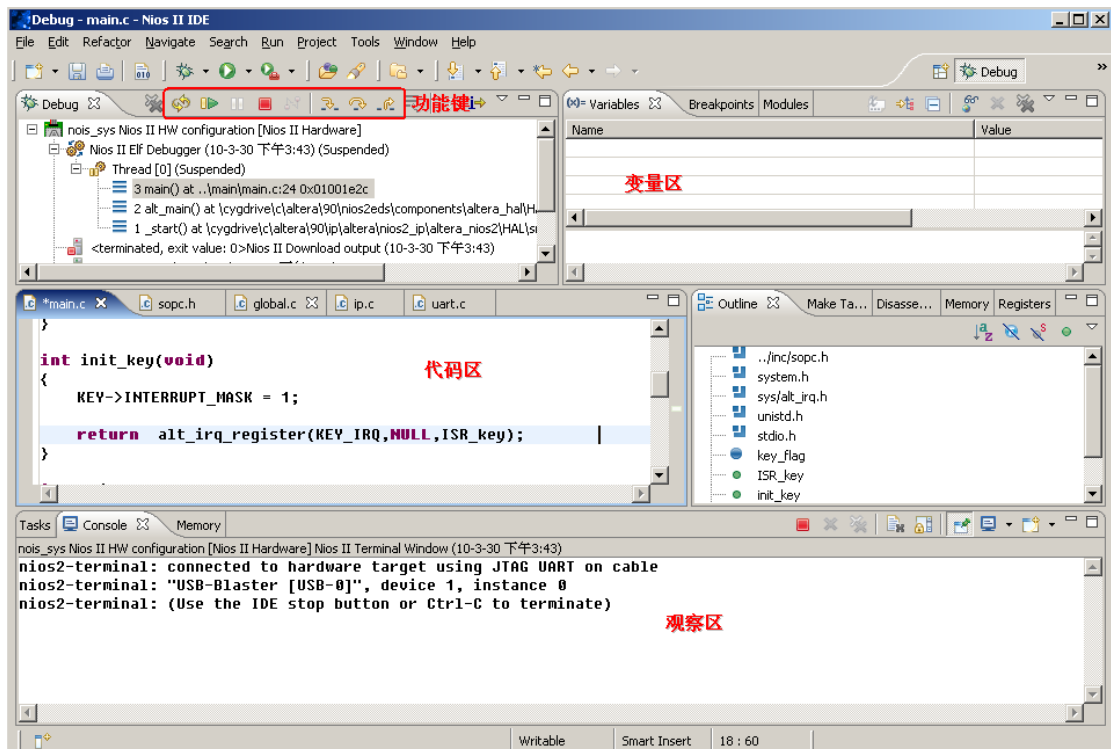
    return 0;
}
```

这个程序都很简单，有利于我们来了解中断的处理过程。

现在借着讲中断的机会，给大家介绍一下 NIOS II IDE 中如何进行在线调试 (DEBUG)。如下图示，在左边框的 hello_world 点击右键，选择 Debug As->Nios II Hardware，或者快捷键 F11（前提：已将仿真器与 JTAG 口相连，并且电源通电），我们就进入了 DEBUG 界面

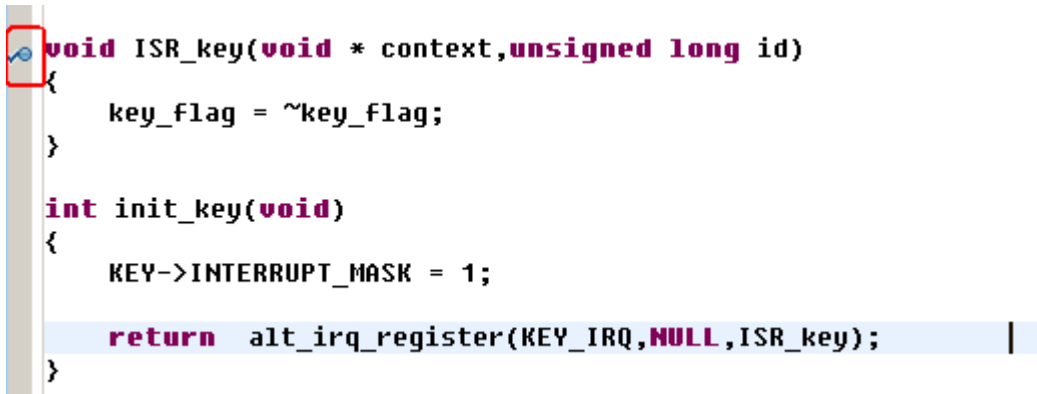


我们来简单介绍一下 DEBUG 界面下的几个功能栏，代码区不用说了，显示我们的源代码，观察区包括主要用到两个，一个是控制台 console，另一个是 Memory。Memory 在正常模式下是不能使用的，也就是说只有在 Debug 模式下，我们才能通过 Memory 来观察内存中数值的改变。变量区主要是用来观察变量值，它的存在有助于我们判断数值的变化是否与我们预想的一样。还有一部分使我们在调试过程中经常用的功能键。包括开始，停止，还有就是三种模式的单步调试功能键。其他区域也有一定的功能，大家可以自己研究一下，对大家调试程序有一定的益处。

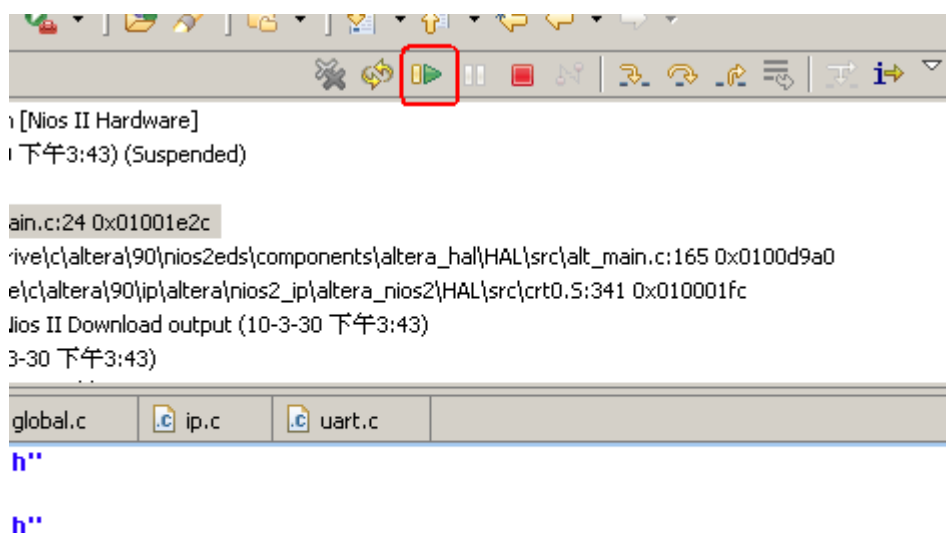


简单介绍了一些功能以后，我们来调试一下我们的中断程序。对于调试中断程序，如何判断中断是否进入，我们有一个小的技巧，简单而有效。

首先，我们在中断函数处设定一个断点，设置断点的方法是在你需要设置断点的地方，即红圈处，双击右键即可。



记下来，让程序全速跑起来，按红圈处的按钮。程序跑起来以后，按开发板上的的外部按键（中间的那个按键），如果中断没有问题，程序就会停在我们设置的中断处，这说明，中断进入成功。这个方法即使用有简单，对调试中断这类程序很有好处。



对于如何运用单步调试之类的功能，只要玩过单片机的，都应该会的，我就不多说了。

四、 总结

最后，再简单总结一下中断需要注意的地方，大家知道中断程序主要处理一些实时性比较强的操作，因此不能在中断程序中进行等待或者阻塞性的操作。所以，尽量保持中断服务程序精简，不要在中断程序中处理复杂的处理，比如浮点型操作，printf 函数等。

这一节就到此为止，由于匆忙，内容可能会出现有问题的地方，希望大家指出，我将进行修改。

第八章 串口实验

串口实验

通过本章，您可以了解到 NIOS II 如何进行串口操作，包括串口中断接收等发面。

本章分为以下几个部分：

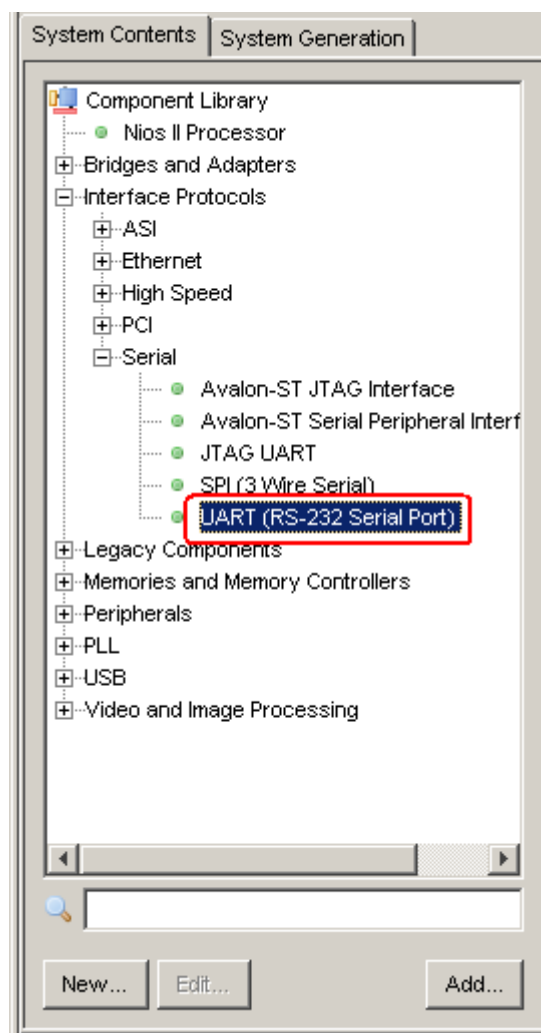
- 一、简介
- 二、硬件开发
- 三、软件编程

一、简介

这一节，我们来说说 RS232，俗称串口。大家对这东西应该很了解，没什么可说的。相对前面我们讲的内容，这一节比较复杂，我会尽力把它讲清楚。在这一节中，我不仅要给大家讲解如何去实现 RS232 功能，更重要的是要提出一种编程思想，如何让程序编写的更严谨，更专业，更有利于以后的维护和移植。

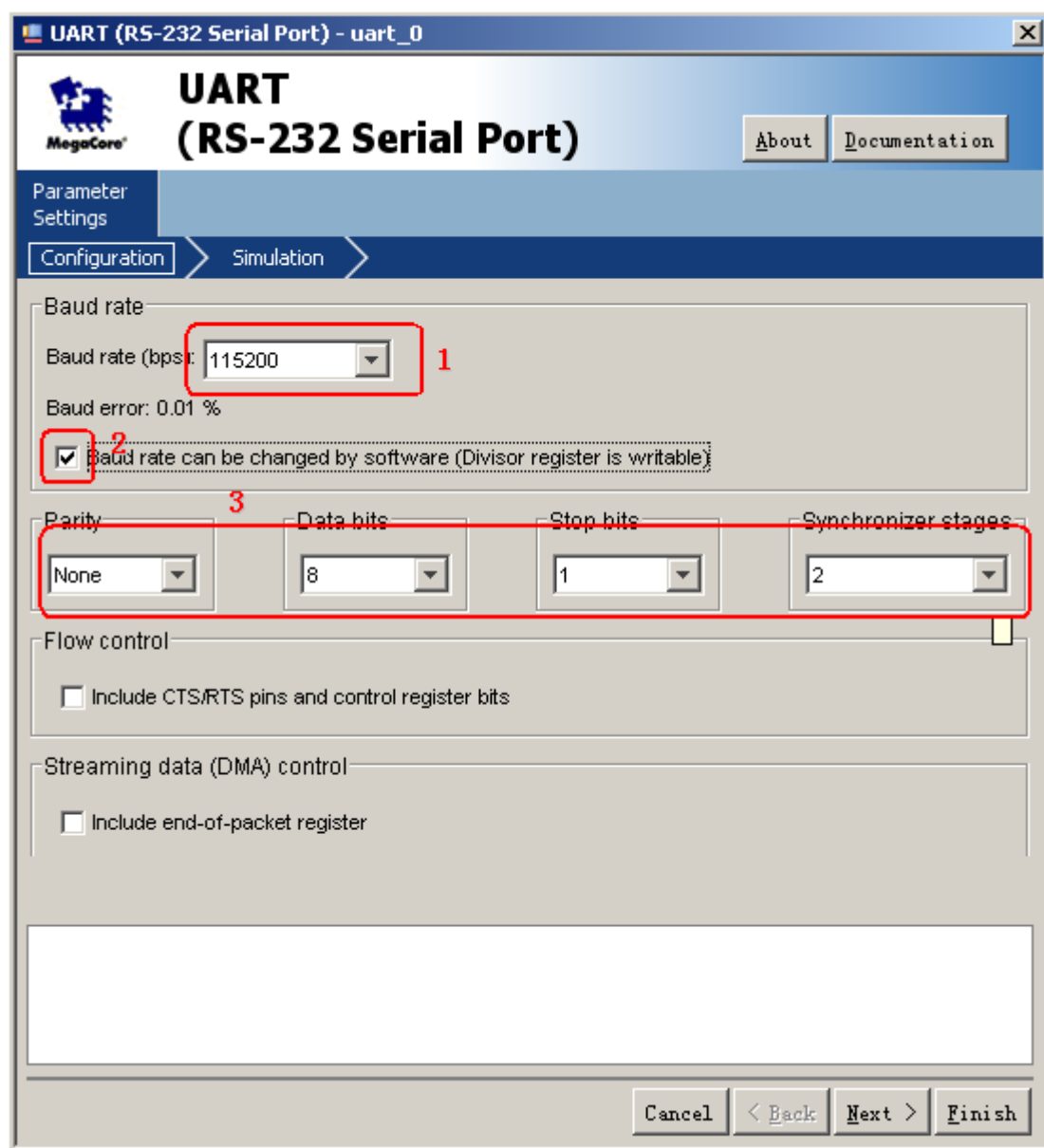
二、硬件开发

首先，我们要在 NIOS II 软核中构建 RS232 模块。打开 Quartus 软件，双击进入 SOPC BUILDER，然后点击下图所示红圈处，



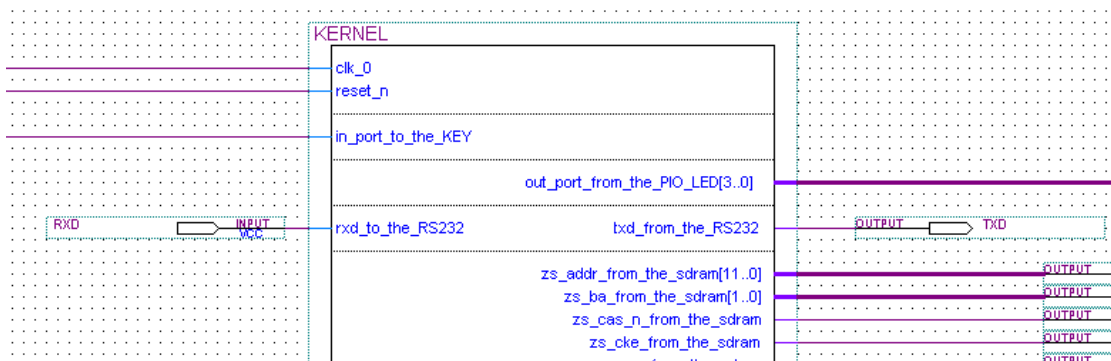
点击后，如下图所示，红圈 1 处为波特率，我们设置为 115200；红圈 2 处是是否允许通过软件改变波特率，我们选中，便是允许，这样我们就可以通过软件来随时更改波特率，如果软件不设置，默认值就是上面设置的 115200；红框 3 中是设置一些与串

口有关的参数，校验方式，数据位，停止位，后面那个基本不用，大家根据实际情况来修改。设置好以后，点击 Next，Finish，完成构建。



构建好以后，将其更名为 RS232，然后进行自动分配地址，自动分配中断号。一切就绪，点击 General，进行编译。

编译好以后退出，进入 Quartus 界面，给其分配引脚，如下图所示



然后运行 TCL 脚本，编译，等待.....

编译好以后，大家可以选择自己的方式将程序下载到 FPGA 中，AS 或 JTAG 都可以。

三、 软件开发

打开 NIOS II 9.0 IDE 后，按快捷键 Ctrl+b 编译程序，等待编译.....

编译好以后，我们再来看 system.h 文件。可以看到 rs232 部分的代码了，如下表所示，红圈处就是我们要用到的部分，大家已经熟悉了，一个是基地址，一个是中断号

```

/*
 * RS232 configuration
 *
 */

#define RS232_NAME "/dev/RS232"
#define RS232_TYPE "altera_avalon_uart"
#define RS232_BASE 0x00201000
#define RS232_SPAN 32
#define RS232_IRQ 2
#define RS232_BAUD 115200
#define RS232_DATA_BITS 8
#define RS232_FIXED_BAUD 0
#define RS232_PARITY 'N'
#define RS232_STOP_BITS 1
#define RS232_SYNC_REG_DEPTH 2
#define RS232_USE_CTS_RTS 0
#define RS232_USE_EOP_REGISTER 0
#define RS232_SIM_TRUE_BAUD 0
#define RS232_SIM_CHAR_STREAM ""

```

```
#define RS232_FREQ 100000000
#define ALT_MODULE_CLASS_RS232 altera_avalon_uart
```

下面，我们开始编写软件程序，首先是修改 sopc.h。如下表格所示

```
typedef struct
{
    //接收寄存器
    union{
        struct{
            volatile unsigned long int RECEIVE_DATA      :8;
            volatile unsigned long int NC                  :24;
        }BITS;
        volatile unsigned long int WORD;
    }RXDATA;
    //发送寄存器
    union{
        struct{
            volatile unsigned long int TRANSMIT_DATA      :8;
            volatile unsigned long int NC                  :24;
        }BITS;
        volatile unsigned long int WORD;
    }TXDATA;
    //状态寄存器
    union{
        struct{
            volatile unsigned long int PE                  :1;
            volatile unsigned long int FE                  :1;
            volatile unsigned long int BRK                  :1;
            volatile unsigned long int ROE                  :1;
            volatile unsigned long int TOE                  :1;
            volatile unsigned long int TMT                  :1;
            volatile unsigned long int TRDY                 :1;
            volatile unsigned long int RRDY                 :1;
            volatile unsigned long int E                    :1;
            volatile unsigned long int NC                    :1;
            volatile unsigned long int DCTS                 :1;
            volatile unsigned long int CTS                   :1;
            volatile unsigned long int EOP                  :1;
        }
    }
}
```

```

        volatile unsigned long int NC1                :19;
    } BITS;
    volatile unsigned long int WORD;
}STATUS;
//控制寄存器
union{
    struct{
        volatile unsigned long int IPE                :1;
        volatile unsigned long int IFE                :1;
        volatile unsigned long int IBRK               :1;
        volatile unsigned long int IROE               :1;
        volatile unsigned long int ITOE               :1;
        volatile unsigned long int ITMT               :1;
        volatile unsigned long int ITRDY               :1;
        volatile unsigned long int IRRDY               :1;
        volatile unsigned long int IE                 :1;
        volatile unsigned long int TRBK                :1;
        volatile unsigned long int IDCTS               :1;
        volatile unsigned long int RTS                 :1;
        volatile unsigned long int IEOP                :1;
        volatile unsigned long int NC                  :19;
    }BITS;
    volatile unsigned long int WORD;
}CONTROL;
//波特率分频器
union{
    struct{
        volatile unsigned long int BAUD_RATE_DIVISOR   :16;
        volatile unsigned long int NC                  :16;
    }BITS;
    volatile unsigned int WORD;
}DIVISOR;
}UART_STR;

```

这个结构体中包括 5 个共用体，这 5 个共用体对应 RS232 的 5 个寄存器，我们来看看这 5 个寄存器，下图所示，这个图来自《n2cpu_Embedded Peripherals.pdf》的第 6-11 页

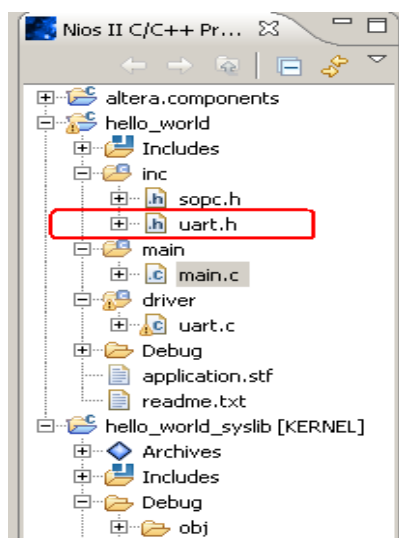
Offset	Register Name	R/W	Description/Register Bits													
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata	RO	Reserved					(1)	(1)	Receive Data						
1	txdata	WO	Reserved					(1)	(1)	Transmit Data						
2	status (2)	RW	Reserved	eop	cts	dcts	(1)	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	control	RW	Reserved	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	divisor (3)	RW	Baud Rate Divisor													
5	endof-packet (3)	RW	Reserved					(1)	(1)	End-of-Packet Value						

这个图中的 (1) 有一个说明，就是说第 7, 8 位根据设置的数据位有所改变，我们设置数据位 8 位，所以 7, 8 位与前 6 为性质相同。

与之前讲的 PIO 的结构体类似，这个结构体的内容是按上图的寄存器顺序来定义的，(因为 endofpacket 没用到，所以在结构中没有定义) 这样在操作过程中就可以实现相应的偏移量 (offset)。在这个结构体中，我们嵌套了 5 个共有体，在共用体中，我们又使用了结构体和位域。头一次看的一定很头晕。其实，我们这样做的目的就是想对寄存器的每一位进行单独的控制，同时也可以实现这个寄存器的整体控制。具体应用，我们在下面的程序中会应用到。有了上面来的结构体以后，我们需要定义一个宏，跟 PIO 的类似。

```
#define _UART
#ifdef _UART
#define UART ((UART_STR *) RS232_BASE)
#endif
```

不用解释了吧，在 PIO 部分已经解释过了，应该没什么问题了吧。接下来，我们要在 inc 下建立 uart.h 文件，如下图所示



建好以后，对 uart.h 进行编写，如下表所示

```
/*
 * =====
 *      Filename:  uart.h
 *      Description:  The head of uart device driver
 *      Version:
 *      Created:
 *      Revision:  none
 *      Compiler:  Nios II IDE
 *      Author:  AVIC
 *      Company:
 * =====
 */

#ifndef UART_H_
#define UART_H_
#include "../inc/sopc.h"
#define BUFFER_SIZE 200

/*-----
 *   Define
 *-----*/

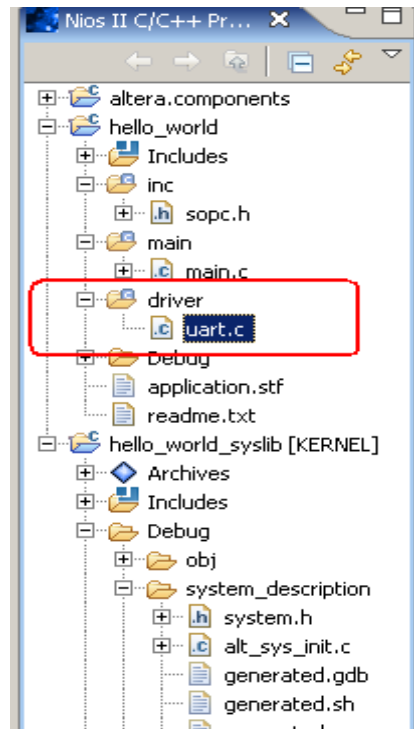
typedef struct{
    unsigned char mode_flag;    //xmodem 1;uart 0;
    unsigned int receive_flag;
    unsigned int receive_count;
    unsigned char receive_buffer[BUFFER_SIZE];
    int (* send_byte)(unsigned char data);
    void (* send_string)(unsigned int len, unsigned char *str);
    int (* init)(void);
    unsigned int (* baudrate)(unsigned int baudrate);
}UART_T;

extern UART_T uart;
#endif /*UART_H_*/
```

在上面的代码中，结构体 UART_T 很重要，它是模拟面向对象的一种编程思想，也是我之前说的一种很重要的编程方式。我们将与 UART 有关系的所有函数、变量都打包在一起，对其他函数来说，它们只能看到 uart 这个结构体，而里面的单独部分都是不可见的。希望大家可以好好体会其中的思想，对大家的编程一定会有很大的好处。

下面 我们要开始写 RS232 的驱动了，首先我们要在 driver 下面建立一个.c 文件，

命名为 uart.c , 如下图所示



建好以后, 我们来编写 uart.c 文件, 如下表所示

```
/*
 * =====
 *      Filename:  uart.c
 *      Description:  RS232 device driver
 *      Version:
 *      Created:
 *      Revision:  none
 *      Compiler:  Nios II IDE
 *      Author:  AVIC
 *      Company:
 *      =====
 */

/*-----
 *  Include
 *-----*/

#include "sys/alt_irq.h"
#include "../inc/sopc.h"
#include <stdlib.h>
```

```

#include <stdio.h>
#include "../inc/uart.h"

/*-----
 *   Function Prototype
 *-----*/
static int uart_send_byte(unsigned char data);
static void uart_send_string(unsigned int len, unsigned char *str);
static int uart_init(void);
static void uart_ISR(void);
static int set_baudrate(unsigned int baudrate);

//初始化 uart 结构体，大家注意结构体的初始化方式
UART_T uart={
    .mode_flag=0,
    .receive_flag=0,
    .receive_count=0,
    .send_byte=uart_send_byte,
    .send_string=uart_send_string,
    .init=uart_init,
    .baudrate=set_baudrate
};

/*
 * ===  FUNCTION  =====
 *      Name:  uart_send_byte
 *  Description:  发送一个字节数据
 * =====
 */
static int uart_send_byte(unsigned char data)
{
    //将发送的数据放到发送数据缓冲区内，等待状态寄存器 TRDY 置 1，当 TRDY 置 1，说明
    //接收完毕
    UART->TXDATA.BITS.TRANSMIT_DATA = data;
    while(!UART->STATUS.BITS.TRDY);

    return 0;
}

```

```

/*
 * === FUNCTION =====
 *      Name:  uart_send_string
 *  Description:  发送字符串数据
 * =====
 */
static void uart_send_string(unsigned int len, unsigned char *str)
{
    while(len--)
    {
        uart_send_byte(*str++);
    }
}

/*
 * === FUNCTION =====
 *      Name:  uart_init
 *  Description:  初始化程序
 * =====
 */
static int uart_init(void)
{
    //设置波特率为115200
    set_baudrate(115200);

    // 对控制寄存器的irrdy进行置1，表示当接收准备好后，中断使能
    UART->CONTROL.BITS.IRRDY=1;

    //清除状态寄存器，这就是处理整个寄存器的方式，大家要注意
    UART->STATUS.WORD=0;

    //注册uart中断，ISR为uart_ISR
    alt_irq_register(UART_IRQ, NULL, uart_ISR);

    return 0;
}

/*
 * === FUNCTION =====
 *      Name:  uart_ISR
 *  Description:  串口中断
 * =====

```

```

*/
static void uart_ISR(void)
{
    //等待状态寄存器的接收数据状态位rrdy，当rrdy位为1时，说明新接收的值传输到了接
    //收数据寄存器
    while(!(UART->STATUS.BITS.RRDY));

    //receive_buffer为我们通过栈的方式在内存中开设的内存块，将接受数据寄存器中的
    //数据放到这个内存块中
    uart.receive_buffer[uart.receive_count++] = UART->RXDATA.BITS.RECEIVE_DATA;

    //当接收数据的最后一位为\n（回车符）时，进入if语句，也就是说，\n作为了结束标志
    //符，每次发送数据后，要加一个回车符作为结束符
    if(uart.receive_buffer[uart.receive_count-1] == '\n'){
        uart.receive_buffer[uart.receive_count] = '\0';
        uart_send_string(uart.receive_count, uart.receive_buffer);
        uart.receive_count = 0;
        uart.receive_flag = 1;
    }
}
/*
* == FUNCTION =====
*      Name: set_baudrate
*  Description: 设置波特率
* =====
*/
static int set_baudrate(unsigned int baudrate)
{
    //设置波特率有一个公式的，波特率=时钟频率/(divisor+1),转换以后就是下面了。
    UART->DIVISOR.WORD = (unsigned int)(ALT_CPU_FREQ/ baudrate + 0.5);
    return 0;
}

```

编写好上面的函数以后，我们要修改 main.c，如下表所示

```

#include "../inc/sopc.h"
#include "system.h"
#include "sys/alt_irq.h"
#include <unistd.h>

```

```
#include <stdio.h>
#include "../inc/uart.h"

int main()
{
    unsigned char buffer[50]="Hello FPGA!\n";

    uart.init();

    while(1){
        uart.send_string(sizeof(buffer),buffer);
        usleep(500000);
    }

    return 0;
}
```

这一节到此就结束了，内容相对有些复杂，大家一定要好好体会，有问题可以给我留言，或者加入我们的 NIOS 技术群（107122106）提问，谢谢大家！

第九章 RTC 实验

RTC 实验

通过本章，您可以了解到实时时钟芯片 DS1302 在 NIOS II 下的实现方式。

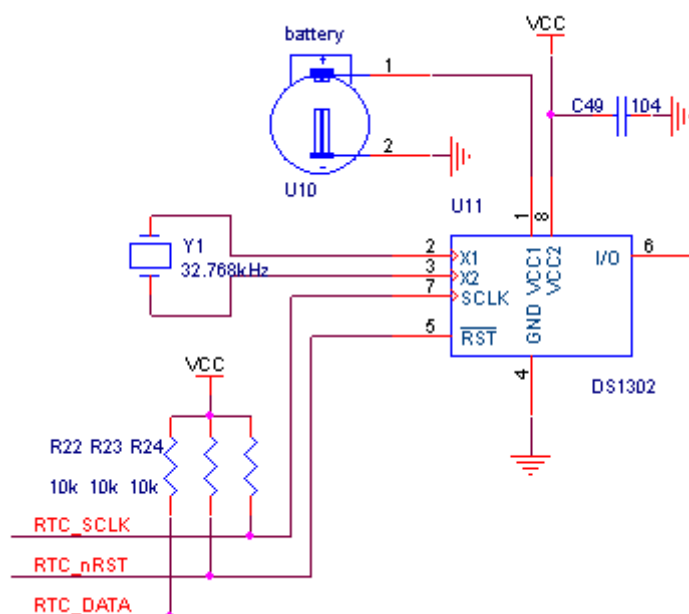
本章分为以下几个部分：

- 一、简介
- 二、硬件开发
- 三、软件编程

一、简介

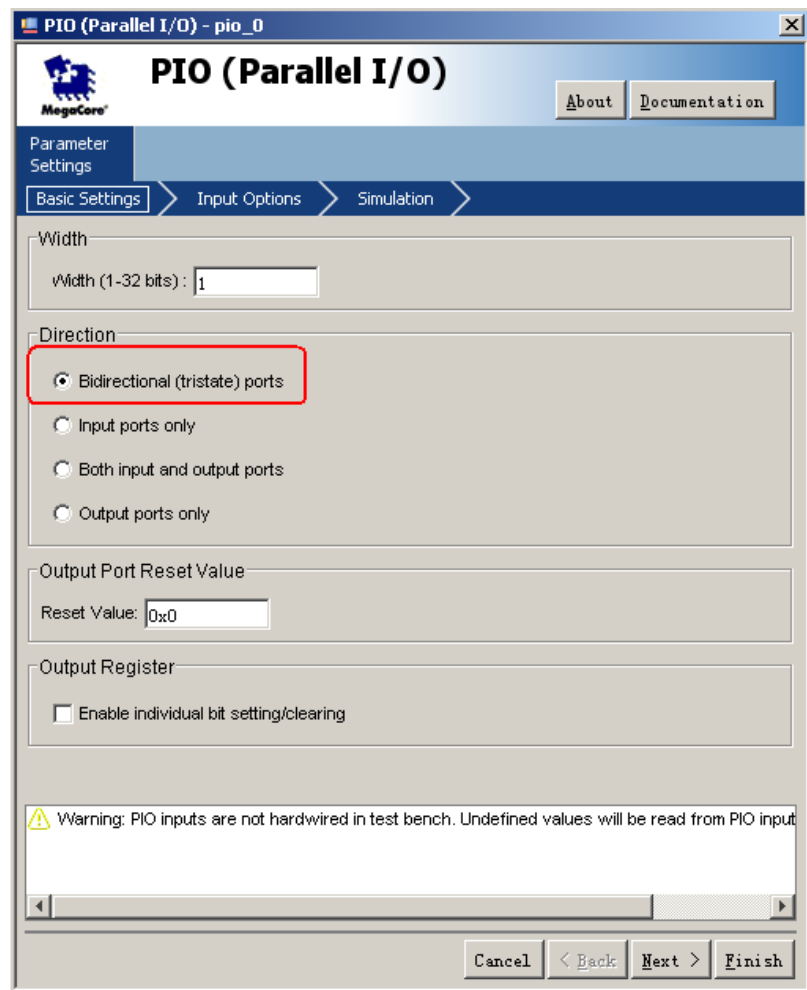
这一节,我将给大家讲解实时时钟部分的内容,我在黑金板上用的实时时钟芯片是 DS1302,这块芯片很常见,性价比也很高。我们主要来讲如何在 NIOS 当中实现其功能,所以功能介绍我简单概括一下,有问题的百度一下就都知道了。

DS1302 是 DALLAS 公司推出的涪流充电实时时钟芯片,内含一个实时时钟/日历和 31 字节静态 RAM,仅需要三根线:RES(复位),I/O(数据线),SCLK(串行时钟)。时钟/RAM 的读/写数据以一个字节或多达 31 个字节的字符组方式通信 DS1302 工作时功耗很低,保持数据和时钟信息时功率小于 1mW。下面看一下电路图吧,下图所示,很简单,三根线就可以搞定了,原理不清楚的去看芯片手册吧,我这里就不讲了



二、硬件开发

首先是,我们需要在软核中构建三个 PIO 模块,方法跟以前讲的一样。需要注意的是 RTC_DATA 这个 PIO,在构建的过程中,我们将其选择为双向的 IO 口,因为它是数据线,既要输入也需要输出,如下图所示,红圈处就是我们需要注意的地方,其他两个 IO 口设置为输出。

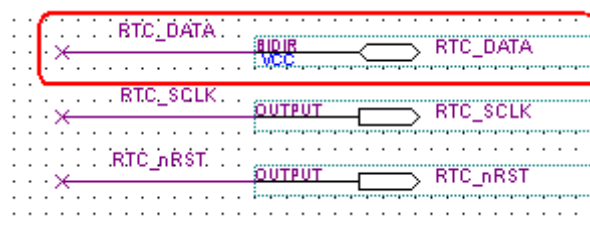


看看构建好以后的样子吧，如下图是所示

☒		☒	RTC_DATA s1	PIO (Parallel I/O) Avalon Memory Mapped Slave	clock		0x00201030	0x0020103f
☒		☒	RTC_SCLK s1	PIO (Parallel I/O) Avalon Memory Mapped Slave	clock		0x00201040	0x0020104f
☒		☒	RTC_nRST s1	PIO (Parallel I/O) Avalon Memory Mapped Slave	clock		0x00201050	0x0020105f

接下来就是自动分配地址，中断，然后开始编译，等待.....

回到 Quartus 后，分配引脚，还是需要注意数据线，也是双向的，分配引脚的时候，要构建双向引脚（bidir），如下图所示。



都设置好以后，我们运行 TCL 脚本文件，然后开始编译，又是等待.....

三、 软件开发

编译好后，我们打开 NIOS II IDE，首先，还是需要编译一下，CTRL+b，编译之后，我们看看 system.h 有什么变化。观察后可以看出，里面对了，RTC 部分的代码，如下表所示，

```
#define RTC_DATA_NAME      "/dev/RTC_DATA"
#define RTC_DATA_TYPE      "altera_avalon_pio"
#define RTC_DATA_BASE 0x00201030
.....

/*
 * RTC_SCLK configuration
 *
 */

#define RTC_SCLK_NAME      "/dev/RTC_SCLK"
#define RTC_SCLK_TYPE      "altera_avalon_pio"
#define RTC_SCLK_BASE 0x00201040
.....

/*
 * RTC_nRST configuration
 *
 */

#define RTC_NRST_NAME      "/dev/RTC_nRST"
#define RTC_NRST_TYPE      "altera_avalon_pio"
#define RTC_NRST_BASE 0x00201050
.....

/*
```

在这些代码中，我们需要用到的是以下部分

```
#define RTC_DATA_BASE 0x00201030
#define RTC_SCLK_BASE 0x00201040
#define RTC_NRST_BASE 0x00201050
```

好的，接下来，我们就开始写程序吧

第一步，修改 sopc.h 文件，加入以下代码到 sopc.h 中

```
#define _RTC

#ifndef _RTC
#define RTC_SCLK ((PIO_STR *) RTC_SCLK_BASE)
#define RTC_DATA ((PIO_STR *) RTC_DATA_BASE)
#define RTC_RST ((PIO_STR *) RTC_NRST_BASE)
#endif /* _RTC */
```

没什么可说的，接下来我们在 inc 文件夹下建立 ds1302.h，在其中加入以下内容，跟串口程序一样，里面也有个结构体，用这种方式整合所有的函数和变量。

```
/*
 * =====
 *
 *      Filename: ds1302.h
 *      Description:
 *      Version: 1.0
 *      Created:
 *      Revision: none
 *      Compiler: Nios II 9.0 IDE
 *      Author: AVIC
 *      Company:
 * =====
 */
#ifndef DS1302_H_
#define DS1302_H_

#include "../inc/sopc.h"

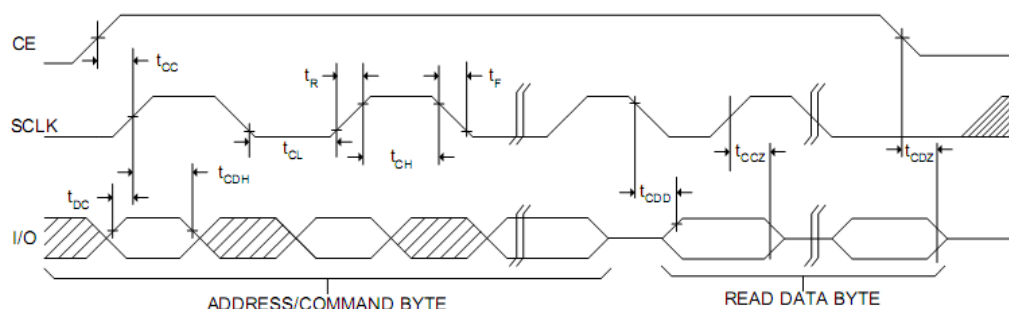
//对于双向的IO，操作的过程中要注意改变IO口的方向，置1为输出，置0为输入
#define RTC_DATA_OUT    RTC_DATA->DIRECTION = 1
#define RTC_DATA_IN     RTC_DATA->DIRECTION = 0

typedef struct{
    void (* set_time)(unsigned char *ti);
    void (* get_time)(char * ti);
}DS1302;
```

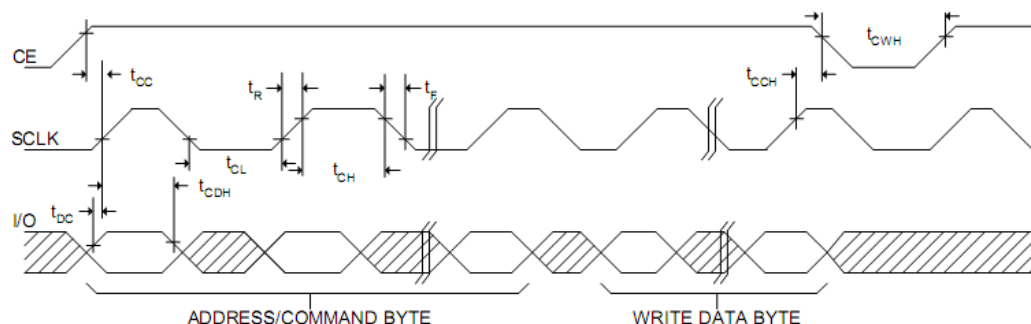
```
extern DS1302 ds1302;
```

```
#endif /*DS1302_H_*/
```

准备工作都做好以后，接下来我们要做的就是写 ds1302 的驱动了，根据 DS1302 的时序图来进行编写，首先我来给看看时序图吧，如下图所示，这个是读数据的时序图，



这个是写数据时序图



还有一个有关寄存器的表格，大家也要注意看一下，如下所示，前面两列是读和写的地址，每次操作时，都先写地址，再传数据。

READ	WRITE	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	RANGE	
81h	80h	CH	10 Seconds			Seconds				00–59	
83h	82h		10 Minutes			Minutes				00–59	
85h	84h	12/24	0	10 AM/PM	Hour	Hour				1–12/0–23	
87h	86h	0	0	10 Date		Date				1–31	
89h	88h	0	0	0	10 Month	Month				1–12	
8Bh	8Ah	0	0	0	0	0	Day			1–7	
8Dh	8Ch	10 Year				Year					00–99
8Fh	8Eh	WP	0	0	0	0	0	0	0	—	
91h	90h	TCS	TCS	TCS	TCS	DS	DS	RS	RS	—	

现在，我们就根据时序图来编写 ds1302 的驱动，在 driver 文件夹下建 ds1302.c 文件，然后添加以下内容，

```

/*
 * =====
 *      Filename:  ds1302.c
 *      Description:
 *      Version:  1.0
 *      Created:  2009-11-23
 *      Revision:  none
 *      Compiler:  Nios II 9.0 IDE
 *      Author:   AVIC
 *      Company:
 * =====
 */

#include "../inc/ds1302.h"

//函数声明
static void delay(unsigned int dly);
static void write_1byte_to_ds1302(unsigned char da);
static unsigned char read_1byte_from_ds1302(void);
static void write_data_to_ds1302(unsigned char addr, unsigned char da);
static unsigned char read_data_from_ds1302(unsigned char addr);
void set_time(unsigned char *ti);
void get_time(char *ti);

//对DS1302结构体进行初始化，注意结构体中函数指针的初始化方式
DS1302 ds1302={
    .set_time = set_time,
    .get_time = get_time
};

/*
 * ===  FUNCTION  =====
 *      Name:  delay
 *      Description:  延时函数
 * =====
 */
void delay(unsigned int dly)
{
    for(;dly>0;dly--);
}

```

```

/*
 * === FUNCTION =====
 *      Name: write_1byte_to_ds1302
 *      Description: 向ds1302写入1 byte数据
 * =====
 */
void write_1byte_to_ds1302(unsigned char da)
{
    unsigned int i;
    //写数据的时候，RTC_DATA为输出，先设置其为输出
    RTC_DATA_OUT;
    //以下步骤是处理串行数据的典型方法，一个位一个位的来判断
    for(i=8; i>0; i--)
    {
        if((da&0x01)!= 0)
            RTC_DATA->DATA = 1;
        else
            RTC_DATA->DATA = 0;
        //根据芯片手册，适当加些延时，不是精确延时
        delay(10);
        RTC_SCLK->DATA = 1;
        delay(20);
        RTC_SCLK->DATA = 0;
        delay(10);

        da >>= 1;
    }
}

/*
 * === FUNCTION =====
 *      Name: read_1byte_from_ds1302
 *      Description: 从ds1302读取1 byte数据
 * =====
 */
unsigned char read_1byte_from_ds1302(void)
{
    unsigned char i;
    unsigned char da = 0;
    //当读数据的时候，我们要将数据IO设置为输入

```

```

RTC_DATA_IN;
//以下是典型的读串行数据的方法
for(i=8; i>0; i--)
{
    delay(10);
    da >>= 1;
    if(RTC_DATA->DATA !=0 )
        da += 0x80;

    RTC_SCLK->DATA = 1;
    delay(20);
    RTC_SCLK->DATA = 0;
    delay(10);
}

RTC_DATA_OUT;

return(da);
}

/*
 * == FUNCTION =====
 *      Name: write_data_to_ds1302
 * Description: 向ds1302写入数据
 * =====
 */
void write_data_to_ds1302(unsigned char addr, unsigned char da)
{
    RTC_DATA_OUT;
    RTC_RST->DATA = 0;//复位，低电平有效
    RTC_SCLK->DATA = 0;
    delay(40);

    RTC_RST->DATA = 1;
    //先写地址，再写数据，每次写1字节
    write_1byte_to_ds1302(addr); // 地址，命令
    write_1byte_to_ds1302(da); // 写1Byte数据

    RTC_SCLK->DATA = 1;
    RTC_RST->DATA = 0;

```

```

        delay(40);
    }

    /*
    * === FUNCTION =====
    *      Name: read_data_from_ds1302
    * Description: 从ds1302读取数据
    * =====
    */
    unsigned char read_data_from_ds1302(unsigned char addr)
    {
        unsigned char da;

        RTC_RST->DATA = 0;
        RTC_SCLK->DATA = 0;

        delay(40);

        RTC_RST->DATA = 1;
        //先写地址，再读数据
        write_1byte_to_ds1302(addr);
        da = read_1byte_from_ds1302();

        RTC_SCLK->DATA = 1;

        RTC_RST->DATA = 0;

        delay(40);

        return(da);
    }

    /*
    * === FUNCTION =====
    *      Name: set_time
    * Description: 设置时间
    * =====
    */
    void set_time(unsigned char *ti)

```

```

{
    unsigned char i;
    unsigned char addr = 0x80;

    write_data_to_ds1302(0x8e,0x00); // 控制命令,WP=0,写操作

    for(i =7;i>0;i--)
    {
        write_data_to_ds1302(addr,*ti); // 秒 分 时 日 月 星期 年

        ti++;
        addr +=2;
    }

    write_data_to_ds1302(0x8e,0x80); // 控制命令,WP=1,写保护
}
/*
 * === FUNCTION =====
 *      Name: get_time
 * Description: 获取时间 ,读取的时间为BCD码 , 需要转换成十进制
 * =====
 */
void get_time(char *ti)
{
    unsigned char i;
    unsigned char addr = 0x81;
    char time;

    for (i=0;i<7;i++){
        time=read_data_from_ds1302(addr);//读取的时间为BCD码
        ti[i] = time/16*10+time%16;//格式为: 秒 分 时 日 月 星期 年
        addr += 2;
    }
}

```

OK,我们的驱动写好了 ,现在我们来写一个 main 函数来验证一下我们的驱动是否好用吧。

```

#include <unistd.h>
#include "../inc/uart.h"
#include "../inc/ds1302.h"

#include <stdio.h>

unsigned char time[7] = {0x00,0x19,0x14,0x17,0x03,0x17,0x10}; //格式为: 秒 分 时 日
月 星期 年

int main()
{
    unsigned char buffer[50]="\0";
    //设置时间
    ds1302.set_time(time);

    while(1){
        //获取时间
        ds1302.get_time(time);

        //将我们要的时间格式化一下, 如2010-4-4 15:25:00
        sprintf(buffer,"20%d-%d-%d %d:%d:%d\n",
            time[6],time[4],time[3],time[2],time[1],time[0]);

        //通过串口发送出去
        uart.send_string(sizeof(buffer),buffer);

        //延时1秒
        usleep(1000000);
    }

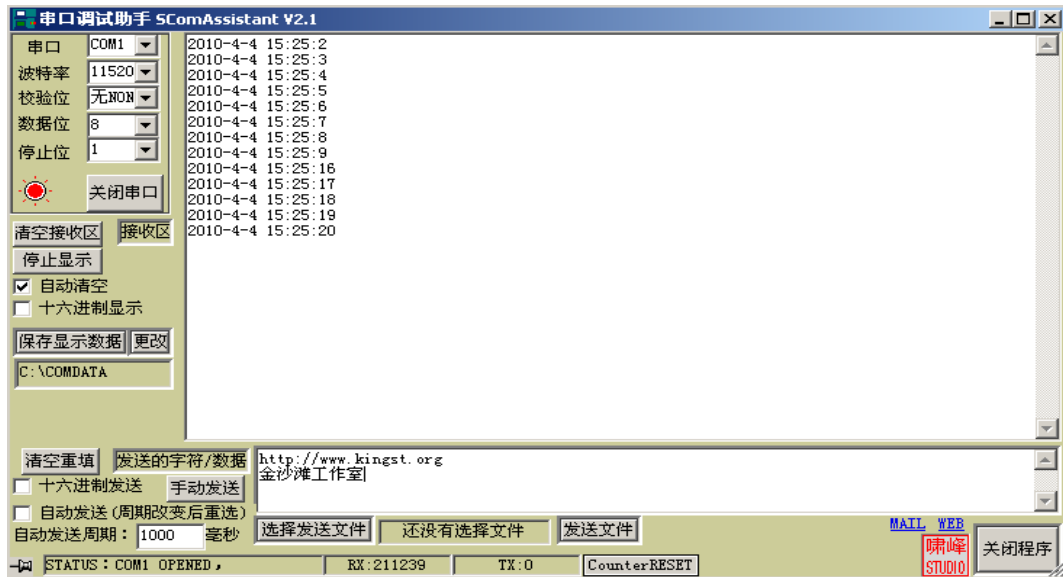
    return 0;
}

```

在上面的程序中,我们获取时间后通过串口发送到上位机,这样也复习了我们上一节讲的串口程序。当然,大家也可以直接通过 printf()打印出来。

在操作 ds1302 的时候有一点需要注意, ds1302 的输入和输出都是 8421BCD 码进行的,所以我们需要对其进行转换。不过,输入的时候我是直接输入 16 进制,比如,我们设置分钟为 10 的话,我直接输入十六进制的 0x10,这样就不需要转化了。而在输出的时候是必须要转化的,大家在写程序的时候注意这一点。

好了，我们来看看我们的劳动果实，看看串口传出的数据吧。



这一节就讲到这吧，如果有问题请给我留言，或者加入我们的 NIOS 技术群：107122106，让我们共同讨论解决，谢谢大家！

第十章 SPI 实验

SPI 实验

通过本章，您可以了解到 NIOS 系统中 SPI 总线的用法。

本章分为以下几个部分：

- 一、简介
- 二、硬件开发
- 三、软件编程

一、 简介

这一节，我们来讲讲 NIOS II 中的 SPI 总线的用法。首先，我们来简单介绍一下 SPI 总线吧，SPI 是英文 Serial Peripheral Interface 的缩写，中文意思是串行外围设备接口，是 Motorola 公司推出的一种同步串行通讯方式，是一种四线同步总线，因其硬件功能很强，与 SPI 有关的软件就相当简单，使 CPU 有更多的时间处理其他事务。

SPI 的通信原理很简单，它以主从方式工作，这种模式通常有一个主设备和一个或多个从设备，需要至少 4 根线，事实上 3 根也可以（用于单向传输时，也就是半双工方式）。也是所有基于 SPI 的设备共有的，它们是 MISO（主入从出），MOSI（主出从入），SCK（时钟），CS（片选）。

- MISO – 主设备数据输出，从设备数据输入；
- MOSI – 主设备数据输入，从设备数据输出；
- SCK – 时钟信号，由主设备产生；
- CS – 从设备使能信号，由主设备控制；

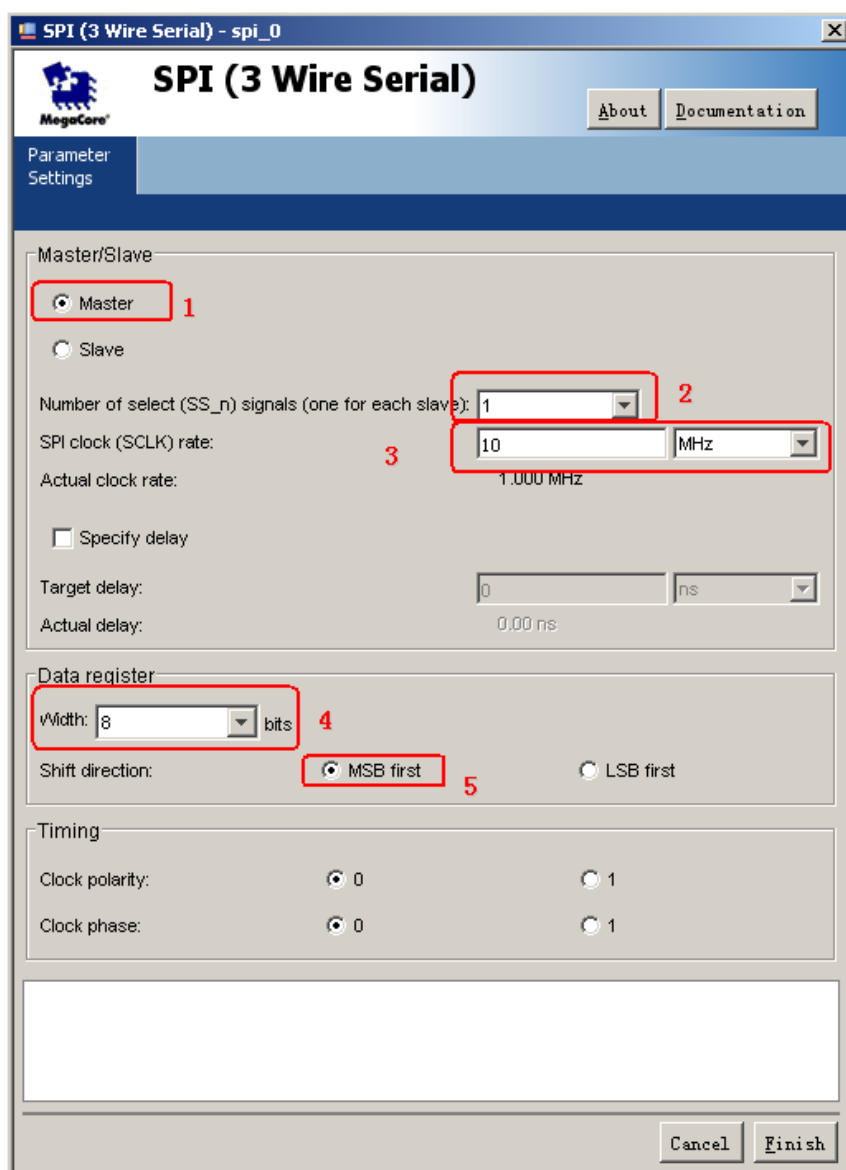
其中 CS 是控制芯片是否被选中的，也就是说只有片选信号为预先规定的使能信号时（高电位或低电位），对此芯片的操作才有效。这就允许在同一总线上连接多个 SPI 设备成为可能。

SPI 总线的理论知识就介绍这么多，想要看具体点的去网上百度一下吧。下面我们就开始 SPI 总线的开发旅程吧。

二、 硬件开发

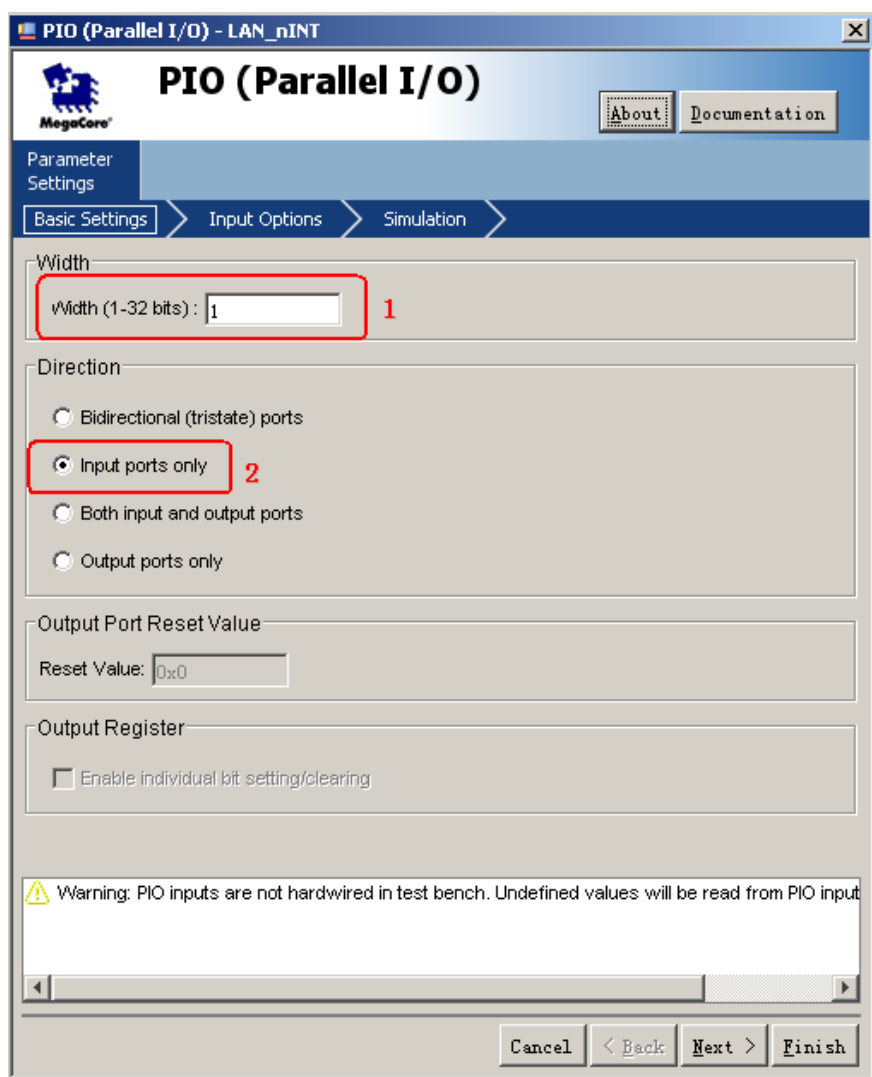
在我们开发板中网口部分是用 SPI 总线实现的，网络芯片是 MICROCHIP 公司的 ENC28J60，我们先看一下这部分的电路，如下图所示，其中与 SPI 总线相关的有，LAN_MISO，LAN_MOSI，LAN_SCK 这三个根线，其余的都是通过 PIO 模块实现的。而且有些线还用不到，比如 LAN_nWOL。

的频率有时候是不一致的（下面显示的是实际频率），例如，我们输入 50MHz，实际的频率只有 25MHz。红圈 4 处是数据的位数，我们选择 8；红圈 5 处是移位的方向，就是说串行数据过来时，是最高位先来还是最低位先来，我们选择 MSB first。

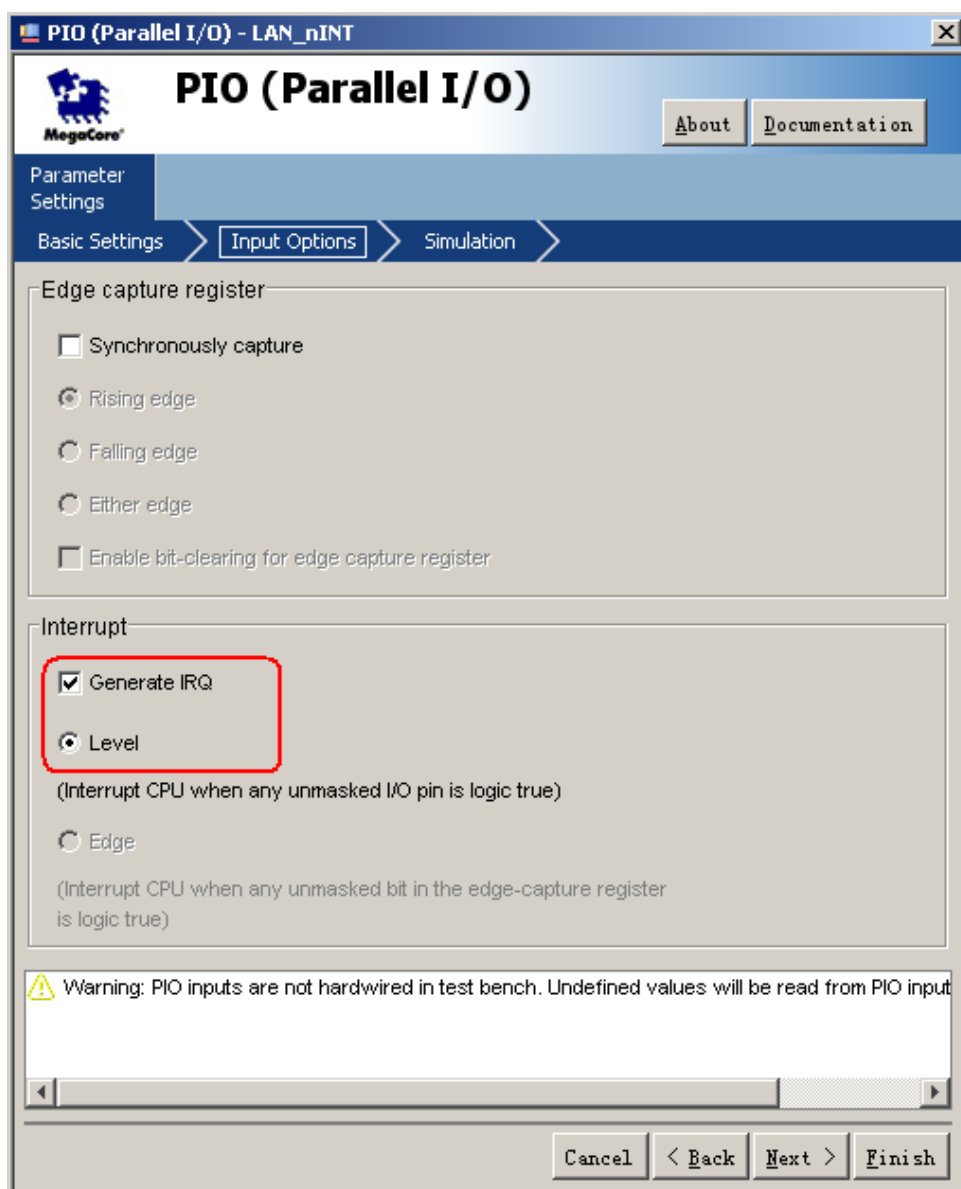


处理好这些以后，点击 Finish，完成构建。接下来，我们还要构建两个 PIO 模块，一个用作 CS 信号控制，一个用作中断信号。之所以没有用 SPI 总线本身的 CS，是由程序处理本身决定的。

中断信号的 PIO 模块，构建过程需要注意一下内容，首先作为中断信号，是输入信号，所以在选择过程中，如下图所示，红圈 1 处选择为 1，红圈 2 处选择 Input ports only，仅作为输入端口，点击 Next，进行下一步



点击后，进入下一步，如下图所示，外部中断要求电平触发，所以按红圈处选择方式。然后，我们点击 Finish，完成构建。

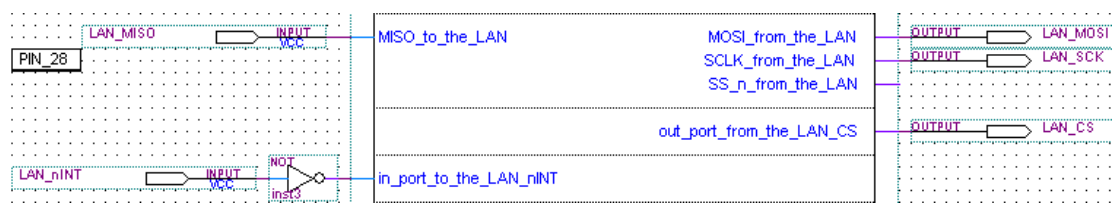


完成上述内容以后，我们需要对模块进行改名，如下图所示，

☐ LAN	s1	SPI (3 Wire Serial)
☐ LAN_CS	s1	Avalon Memory Mapped Slave
☐ LAN_nINT	s1	PIO (Parallel I/O)
		Avalon Memory Mapped Slave

一切就绪，别忘了自动地址分配和中断分配。哦了，我们开始编译吧，等待.....

编译好以后，我们回到 Quartus 界面，根据 TCL 脚本文件进行管脚分配，如下图所示



接下来我们运行脚本文件，进行编译，又一次漫长的等待.....

编译成功以后，我们开始进行软件部分的开发。

三、 软件开发

打开 NIOS II 9.0 IDE，然后进行编译，快捷键 Ctrl+b，等待编译成功后，我们来看看 system.h 中多了些什么，如下表所示，

```
/*
 * LAN configuration
 *
 */
#define LAN_NAME "/dev/LAN"
#define LAN_TYPE "altera_avalon_spi"
#define LAN_BASE 0x00201020
.....
/*
 * LAN_CS configuration
 *
 */
#define LAN_CS_NAME "/dev/LAN_CS"
#define LAN_CS_TYPE "altera_avalon_pio"
#define LAN_CS_BASE 0x00201060
.....
/*
 * LAN_nINT configuration
 *
 */
#define LAN_NINT_NAME "/dev/LAN_nINT"
#define LAN_NINT_TYPE "altera_avalon_pio"
#define LAN_NINT_BASE 0x00201070
.....
```

我们需要以下内容

```

#define LAN_BASE          0x00201020
#define LAN_CS_BASE       0x00201060
#define LAN_NINT_BASE     0x00201070

```

接下来，我们需要对 sopc.h 进行修改，在其中加入以下代码

```

typedef struct{
    volatile unsigned long int RXDATA;
    volatile unsigned long int TXDATA;
    union{
        struct{
            volatile unsigned long int NC          :3;
            volatile unsigned long int ROE          :1;
            volatile unsigned long int TOE          :1;
            volatile unsigned long int TMT          :1;
            volatile unsigned long int TRDY         :1;
            volatile unsigned long int RRDY         :1;
            volatile unsigned long int E            :1;
            volatile unsigned long int NC1          :23;
        }BITS;
        volatile unsigned long int WORD;
    }STATUS;

    union{
        struct{
            volatile unsigned long int NC          :3;
            volatile unsigned long int IROE         :1;
            volatile unsigned long int ITOE         :1;
            volatile unsigned long int NC1          :1;
            volatile unsigned long int ITRDY        :1;
            volatile unsigned long int IRRDY        :1;
            volatile unsigned long int IE           :1;
            volatile unsigned long int NC2          :1;
            volatile unsigned long int SSO          :21;
        }BITS;
        volatile unsigned long int CONTROL;
    }CONTROL;
    unsigned long int RESERVED0;
    unsigned long int SLAVE_SELECT;
}SPI_STR;

```

这部分代码是根据《n2cpu_Embedded Peripherals.pdf》的第 7-10 页，如下表所示，结构体的顺序是根据下表的排列顺序进行设计的，与串口中结构体的道理相同。

Internal Address	Register Name	32..11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata (1)	RXDATA (n-1..0)											
1	txdata (1)	TXDATA (n-1..0)											
2	status (2)				E	RRDY	TRDY	TMT	TOE	ROE			
3	control		SSO (3)		I E	IRRDY	ITRDY		ITOE	IROE			
4	Reserved												
5	slaveselct (3)	Slave Select Mask											

除了上述结构体以外，我们还要在 soc.h 中加入以下代码

```
#ifndef _LAN
#define LAN ((SPI_STR *) LAN_BASE)
#define LAN_CS ((PIO_STR *) LAN_CS_BASE)
#endif /* _LAN */
```

修改好 soc.h 以后，我们需要在 inc 文件夹下建立一个 enc28j60.h，在其中加入以下内容，(这只是 enc28j60.h 文件中的一部分，还有很大一部分宏定义没有写出)。

```
/*-----
 * Data Struct
 *-----*/
typedef const struct{
    unsigned char (* read_control_register)(unsigned char address);
    void (* initialize)(void);
    void (* packet_send)(unsigned short len,unsigned char * packet);
    unsigned int (* packet_receive)(unsigned short maxlen,unsigned char * packet);
}ENC28J60;

/*-----
 * external variable
 *-----*/
extern ENC28J60 enc28j60;
```

大家可以看出，在我们的程序中，这样的结构体随处可见，在之前的串口程序，还是这个 SPI 程序 我们都在用。它的好处就在于，可以将零散的函数和变量整合在一起，通过结构体的形式来处理，大大提高了程序的可读性，也增强了程序的可维护性和可移植性。

处理好上述内容后，我们开始编写 enc28j60 的驱动程序，内容很多，我们截取其中一部分有关 SPI 的内容来进行讲解

```
/*=====
 *      Filename:  enc28j60.c
 *      Description:  enc28j60 device driver
 *      Version:  1.0.0
 *      Created:  2009-8-7 13:05:54
 *      Revision:  none
 *      Compiler:  Nios II IDE
 *      Author:  AVIC
 *      Company:
 * =====
 */

/*-----
 *  Include
 *-----*/
#include "../inc/enc28j60.h"
#include "../inc/sopc.h"
#include <stdio.h>

/*-----
 *  Function Prototype
 *-----*/
static unsigned char enc28j60_read_control_register(
unsigned char address);
static void enc28j60_initialize(void);
static void enc28j60_packet_send(unsigned short len,
unsigned char * packet);
static unsigned int enc28j60_packet_receive(
unsigned short maxlen,unsigned char * packet);

/*-----
 *  Variable
 *-----*/
//结构体初始化，注意初始化的写法
ENC28J60 enc28j60={
    .read_control_register = enc28j60_read_control_register,
```

```

.initialize      = enc28j60_initialize,
.packet_send     = enc28j60_packet_send,
.packet_receive  = enc28j60_packet_receive
};

static unsigned char enc28j60_bank = 1;
static unsigned short next_packet_pointer;

/*
 * === FUNCTION=====
 *      Name:  set_cs
 *  Description:
 * =====
 */
static void set_cs(unsigned char level)
{
    if(level)
        LAN_CS->DATA = 1;
    else
        LAN_CS->DATA = 0;
}

/*
 * === FUNCTION =====
 *      Name:  enc28j60_write_operation
 *  Description:  ENC28J60的写操作
 * =====
 */
static void enc28j60_write_operation(unsigned char op,
    unsigned char address, unsigned char data)
{
    //首先将CS置低，CS低电平有效
    set_cs(0);

    //首先是写命令，等待状态寄存器的TMT位，当该位为0，说明正在发送数据，当该位
    //为1时，说明发送完毕，此时寄存器为空
    LAN->TXDATA = (op | (address & 0x1F));
    while(!(LAN->STATUS.BITS.TMT));

    //写数据，等待状态寄存器的TMT位，当该位为0，说明正在发送数据，当该位
    //为1时，说明发送完毕，此时寄存器为空

```

```

LAN->TXDATA = data;
while(!(LAN->STATUS.BITS.TMT));
//发送完毕以后，将CS置高
set_cs(1);
}

/*
 * == FUNCTION ==
 *      Name: enc28j60_read_operation
 *      Description: ENC28J60的读操作
 * ==
 */
static unsigned char enc28j60_read_operation(unsigned char op,
                                             unsigned char address)
{
    unsigned char data;

    //首先将CS置低，CS低电平有效
    set_cs(0);

    //首先是写命令，等待状态寄存器的TMT位，当该位为0，说明正在发送数据，当该位
    //为1时，说明发送完毕，此时寄存器为空
    LAN->TXDATA = op|(address&0x1f);
    while(!(LAN->STATUS.BITS.TMT));

    //写数据，发送0x00，0x00是个随机数，为了使能时钟，发送的数据与硬件有关系
    LAN->TXDATA = 0x00; //0x00 is random number ,to enable clock
    while(!(LAN->STATUS.BITS.TMT));

    //MAC和MII寄存器读的第一个字节是无效的，所以他们需要写两次
    if(address&0x80){
        LAN->TXDATA = 0x00;
        while(!(LAN->STATUS.BITS.TMT));
    }
    //开始读数据
    data = LAN->RXDATA;
    //读完以后，将CS置高
    set_cs(1);
    return data;
}

```

对于网口这部分程序，需要结合 TCP/IP 协议才能进行通信，所以，这部分主函数就暂时不写了，等讲到 TCP/IP 协议那部分的时候，我们再进行讲解。

我们这一节主要是讲解 SPI 总线的使用方法，对于数据的收发都有所涉及，希望大家能够充分的理解其中的编程方法。有关 ENC28J60 的完整驱动，我将以附件形式提供给大家，这一节到此结束了，如果大家对这部分内容有疑问，可以加入我们的 NIOS 技术群，或者通过邮件形式与我沟通，谢谢大家。

第十一章 IIC 实验

IIC 实验

通过本章，您可以了解到 NIOS 系统中 IIC 总线的用法。

本章分为以下几个部分：

- 一、简介
- 二、硬件开发
- 三、软件编程

一、简介

这一节，我们来讲一讲有关 IIC 总线的实验，在硬件中，我们实用了 24LC04，一个 512 字节的 EEPROM。在 NIOS II 中，没有集成 IIC 接口，为了实现这一功能，我们有两种途径，一种就是自己写 IP 核或者移植别人的 IP 核，另一种方法就是通过 IO 口模拟 IIC 总线协议。我们这一节采用的方法是后者，通过 IO 口模拟 IIC 总线协议，以达到对 24LC04 控制读写的目的。

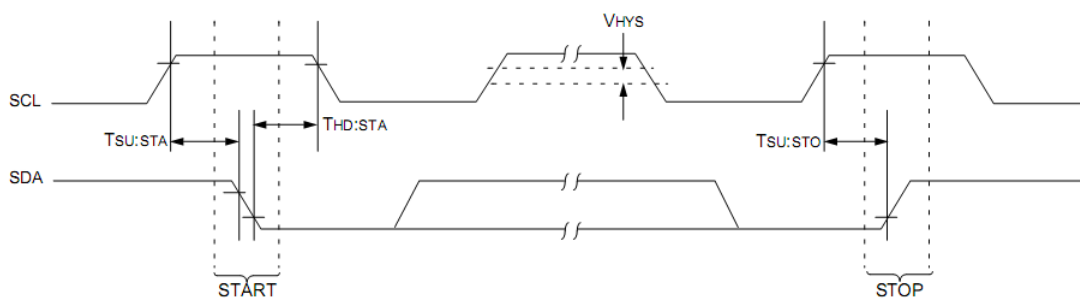
首先，我简单介绍一下 IIC 总线的原理，大家稍微了解一下。IIC(Inter - Integrated Circuit)总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。它是由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据。在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送，最高传送速率 100kbps。它在传送数据过程中共有三种类型信号，它们分别是：开始信号、结束信号和应答信号。

开始信号：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。

结束信号：SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。

应答信号：接收数据的 IC 在接收到 8bit 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

这些信号中，起始信号是必需的，结束信号和应答信号，都可以不要。下图就为 IIC 总线的时序图。



简单介绍之后，我们就要开始实践一下了，开始吧

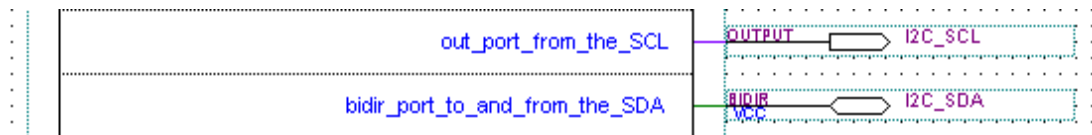
二、硬件开发

首先，需要在软核中添加两个 IO 模块，并将其命名为 SCL 和 SDA，其中，SCL 为 output 建 ports only（仅输出），SDA 为 Bidirection (tristate) port(双向)，建好以后，如下图所示

	SCL s1	PIO (Parallel I/O) Avalon Memory Mapped Slave	clock		0x00200000	0x0020000f
	SDA s1	PIO (Parallel I/O) Avalon Memory Mapped Slave	clock		0x00200010	0x0020001f

接下来，我们自动分配一下地址，编译。

完成后，我们回到 Quartus 界面。然后我们来分配引脚，如下图所示



分配好管脚以后，我们运行 TCL 脚本文件，开始编译(Ctrl+L).....

编译完成后，我们的硬件部分就结束了，接下来，就是我们的软件开发部分了。

三、 软件开发

首先，我们打开 NIOS II 9.0 IDE，然后进行编译 (Ctrl+B)。

编译好以后，我们看一下 system.h 文件，看是否多出了 SCL 和 SDA 部分代码。跟我们预期的一样，system.h 文件中出现了 SCL 和 SDA 部分代码，如下表所示

```

/*
 * SCL configuration
 *
 */
#define SCL_NAME "/dev/SCL"
#define SCL_TYPE "altera_avalon_pio"
#define SCL_BASE 0x00201060
.....
/*
 * SDA configuration
 *
 */

#define SDA_NAME "/dev/SDA"
#define SDA_TYPE "altera_avalon_pio"
#define SDA_BASE 0x00201070
.....

```

在跟大家讨论过程中，我了解到很多人都想知道有关 NIOS II 自带的 API 的用法，所以，今天我就用这种方式来实现我们的程序。不过我还是推荐大家用我之前的方式编写程序，道理我已经说过了，在此不再重复。

下面我们在 inc 目录下建立一个 iic.h 文件，如下表所示

```
#ifndef IIC_H_
#define IIC_H_

#define OUT 1
#define IN 0

typedef struct{
    void (* write_byte)(unsigned short addr, unsigned char dat);
    unsigned char (* read_byte)(unsigned short addr);
}IIC;

extern IIC iic;

#endif /*IIC_H_*/
```

接下来，我们需要在 driver 下建立 iic.c 文件，如下表所示

```
#include <stdio.h>
#include <sys/unistd.h>
#include <io.h>

#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"
#include "../inc/iic.h"

static alt_u8 read_byte(alt_u16 addr);
static void write_byte(alt_u16 addr, alt_u8 dat);

IIC iic = {
    .write_byte = write_byte,
    .read_byte = read_byte
};
```

```

/*
 * === FUNCTION =====
 *      Name:  start
 *      Description:  IIC启动
 * =====
 */
static void start(void)
{
    IOWR_ALTERA_AVALON_PIO_DIRECTION(SDA_BASE, OUT);
    IOWR_ALTERA_AVALON_PIO_DATA(SDA_BASE, 1);
    IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 1);
    usleep(10);
    IOWR_ALTERA_AVALON_PIO_DATA(SDA_BASE, 0);
    usleep(5);
}

/*
 * === FUNCTION =====
 *      Name:  uart_send_byte
 *      Description:  IIC停止
 * =====
 */
static void stop(void)
{
    IOWR_ALTERA_AVALON_PIO_DIRECTION(SDA_BASE, OUT);
    IOWR_ALTERA_AVALON_PIO_DATA(SDA_BASE, 0);
    IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 0);
    usleep(10);
    IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 1);
    usleep(5);
    IOWR_ALTERA_AVALON_PIO_DATA(SDA_BASE, 1);
    usleep(10);
}

/*
 * === FUNCTION =====
 *      Name:  ack
 *      Description:  IIC应答
 * =====
 */
static void ack(void)
{

```

```

alt_u8 tmp;

IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 0);
IOWR_ALTERA_AVALON_PIO_DIRECTION(SDA_BASE, IN);
usleep(10);
IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 1);
usleep(5);
tmp = IORD_ALTERA_AVALON_PIO_DATA(SDA_BASE);
usleep(5);
IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 0);
usleep(10);

while(tmp);
}
/*
 * === FUNCTION =====
 *      Name: iic_write
 * Description: IIC写一个字节
 * =====
 */
void iic_write(alt_u8 dat)
{
    alt_u8 i, tmp;

    IOWR_ALTERA_AVALON_PIO_DIRECTION(SDA_BASE, OUT);

    for(i=0; i<8; i++){
        IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 0);
        usleep(5);
        tmp = (dat & 0x80) ? 1 : 0;
        dat <<= 1;
        IOWR_ALTERA_AVALON_PIO_DATA(SDA_BASE, tmp);
        usleep(5);
        IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 1);
        usleep(10);
    }
}
/*
 * === FUNCTION =====
 *      Name: read

```

```

* Description: IIC读一个字节
* =====
*/
static alt_u8 iic_read(void)
{
    alt_u8 i, dat = 0;

    IOWR_ALTERA_AVALON_PIO_DIRECTION(SDA_BASE, IN);

    for(i=0; i<8; i++){
        IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 0);
        usleep(10);
        IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 1);
        usleep(5);
        dat <<= 1;
        dat |= IORD_ALTERA_AVALON_PIO_DATA(SDA_BASE);
        usleep(5);
    }

    usleep(5);
    IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 0);
    usleep(10);
    IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 1);
    usleep(10);
    IOWR_ALTERA_AVALON_PIO_DATA(SCL_BASE, 0);

    return dat;
}

/*
* === FUNCTION =====
*      Name: write_byte
* Description: 向EEPROM写一个字节
* =====
*/
static void write_byte(alt_u16 addr, alt_u8 dat)
{
    alt_u8 cmd;
    cmd = (0xa0 | (addr >> 7)) & 0xfe;

```

```

    start();
    iic_write(cmd);
    ack();
    iic_write(addr);
    ack();
    iic_write(dat);
    ack();
    stop();
}
/*
 * ===  FUNCTION  =====
 *      Name:  read_byte
 *  Description: 从EEPROM读一个字节
 * =====
 */
static alt_u8 read_byte(alt_u16 addr)
{
    alt_u8 cmd, dat;
    cmd = (0xa0 | (addr >> 7)) & 0xfe;

    start();
    iic_write(cmd);
    ack();
    iic_write(addr);
    ack();
    start();
    cmd |= 0x01;
    start();
    iic_write(cmd);
    ack();
    dat = iic_read();
    stop();

    return dat;
}

```

最后，我们来建立 main.c 函数

```

#include <unistd.h>
#include "../inc/iic.h"

```

```

#include <stdio.h>
#include "alt_types.h"

alt_u8 write_buffer[512], read_buffer[512];

int main()
{
    alt_u16 i, err;
    alt_u8 dat;

    printf("\nWriting data to EEPROM!\n");

    //写入512byte的数据，前256个数字为0到255，后256个数据为1
    for(i=0; i<512; i++){
        if(i<256)
            dat = i;
        else
            dat = 1;

        iic.write_byte(i, dat);
        write_buffer[i] = dat;
        printf("0x%02x ", dat);
        usleep(10000);
    }

    printf("\nReading data from EEPROM!\n");

    //将512byte数据读出来并打印
    for(i=0; i<512; i++){
        read_buffer[i] = iic.read_byte(i);
        printf("0x%02x ", read_buffer[i]);
        usleep(1000);
    }

    err = 0;
    printf("\nVerifying data!\n");

    //对比数据是否相同，如果有不同，说明读写过程有错误
    for(i=0; i<512; i++){
        if(read_buffer[i] != write_buffer[i])

```

```
        err ++;
    }

    if(err == 0)
        printf("\nData write and read successfully!\n");
    else
        printf("\nData write and read failed!--%d errors\n", err);

    return 0;
}
```

程序很简单，大家只要对 IIC 总线有一定的了解就会明白的。

好的，这节的内容就讲到这，谢谢大家对我的支持。如果有问题，可以给我留言或者直接加入我们的 NIOS 技术高级群：107122106，也可以加我的 qq：984597569

第十二章 定时器

定时器

通过本章，您可以了解到 NIOS 系统中定时器相关内容及用法。

本章分为以下几个部分：

- 一、简介
- 二、硬件开发
- 三、软件开发

一、简介

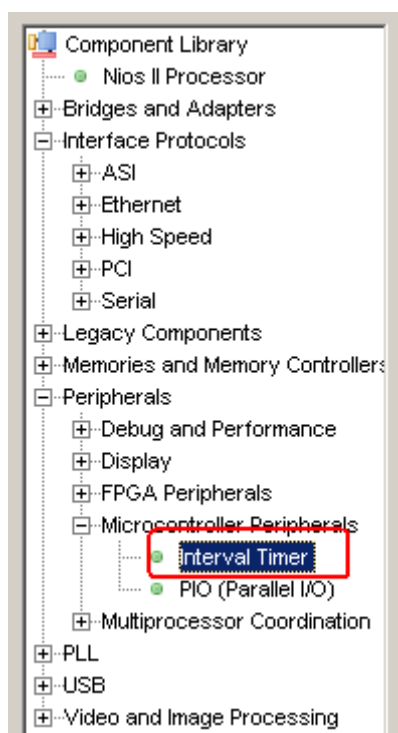
这一节，我们来讲讲有关定时器的内容。定时器，顾名思义，与时间有关系的一个器件，是用于对时钟周期进行计数并产生周期性中断信号的硬件外围设备。

用过单片机的人对定时器一定很熟悉，它主要用来处理周期性的事件，比如设置 AD 的采样频率，通过定时器产生周期性的定时器中等等。

我发现，在很多资料中，都是介绍如何实现系统时钟，Timestamp，或者是看门狗的功能，却没有真正的介绍如何去使用定时器本身具备的功能，这样很容易误导大家。有的人可能在学习这部分内容的时候遇到这样的问题，一个软核只能定义一个系统时钟，那如果我们想用到两个定时器怎么办呢？没办法，这种方式行不通，这就是系统时钟方式的弊端。为了解决这个问题，我们就要撇开系统时钟，真正的去了解 NIOS 本身定时器所具备的功能，它是可以像单片机的定时器一样来操作的。下面我们开始吧。

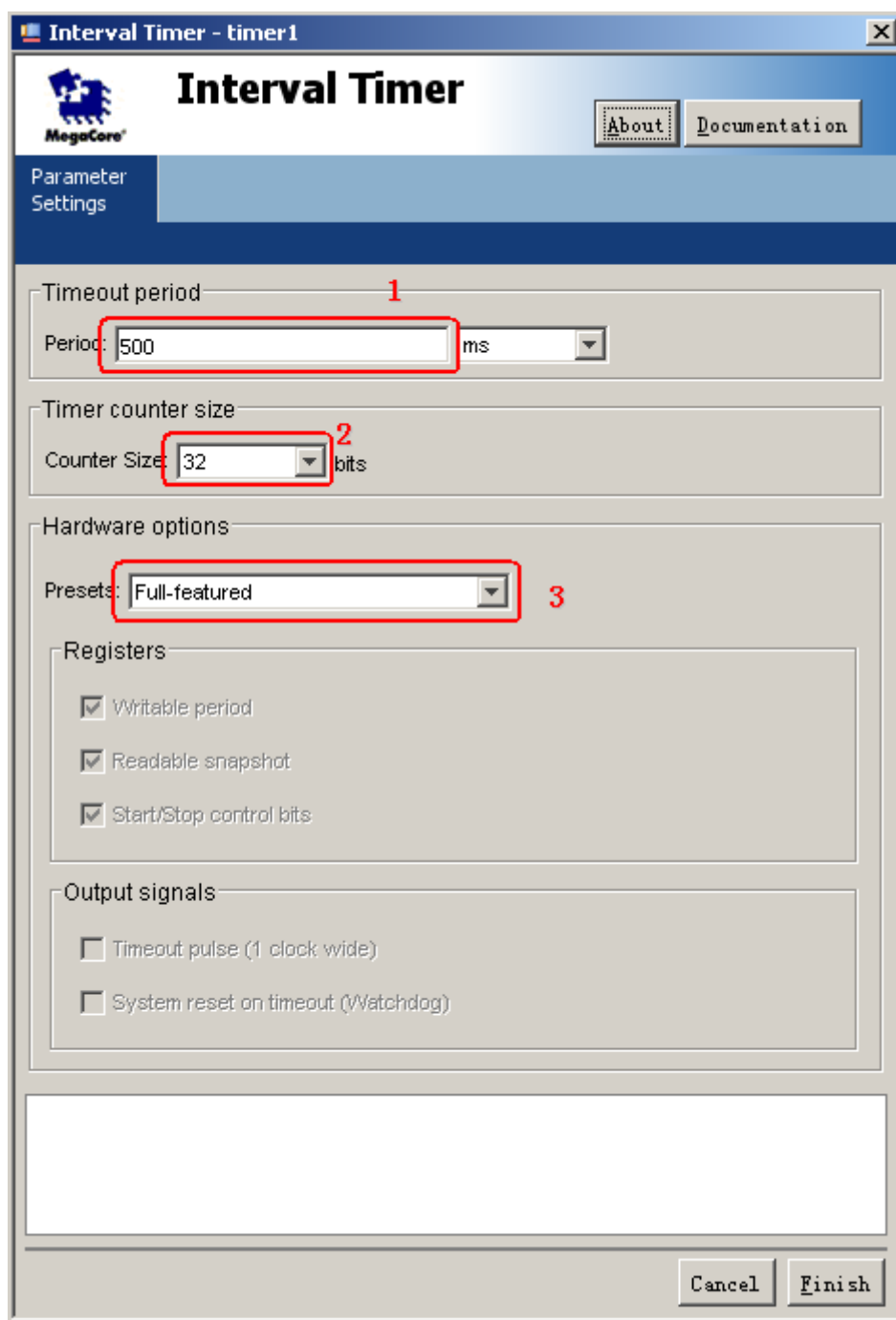
二、硬件开发

首先我们需要在 NIOS 软核中构建 timer 模块，如下图所示



点击进入后，如下图所示，红圈 1 处是用于预设硬件生成后的定时器周期，也就是说这是个初始值，我们可以通过软件来改变定时器的周期。红圈 2 处是定时器计数器的大小，它分为 32 位和 64 位两种，需要根据你的定时器的周期来选择，我们在这

里选择 32 位。红圈 3 处是定时器的几种预设方式，是为了实现不同的功能，我们在这里选择 Full-featured，就是全功能。选择了这个选项，我们就可以修改周期，可以控制停止开始位。选择好以后，点击 Finish 完成设置。



我们在这里建立两个定时器，另一个方法相同。建立好以后，如下图所示

	timer1	Interval Timer	
	s1	Avalon Memory Mapped Slave	clk
	timer2	Interval Timer	
	s1	Avalon Memory Mapped Slave	clk

我们这个定时器试验需要 4 个 LED 来配合，所以还需要建立一个 PIO 模块，这里不具体讲了，前面已经讲过了。接下来，自动配置地址，自动配置中断，跟以前的都一样，下来就是编译，等待，编译，等待...

硬件部分就 OK 了，接下来我们开始软件开发了。

三、 软件开发

打开 NIOS 9.0 IDE，首先编译一下，快捷键 Ctrl+b。编译完以后，我们可以去看看 system.h 文件，应该出现了下面内容

```
#define TIMER1_NAME "/dev/timer1"
#define TIMER1_TYPE "altera_avalon_timer"
#define TIMER1_BASE 0x00201000
.....
#define TIMER2_NAME "/dev/timer2"
#define TIMER2_TYPE "altera_avalon_timer"
#define TIMER2_BASE 0x00201000
.....
```

大家在开发 NIOS 软件的时候，要注意一定要先看 systetm.h 文件，首先确定硬件已经正确加载，而且名字跟你设定的是一样的，避免在这个地方出现问题。

下面，我们来看看定时器的寄存器，这面的表格来自《n2cpu_Embedded Peripherals.pdf》的第 24-6 页。

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	(1)					RUN	TO
1	control	RW	(1)			STOP	START	CONT	ITO
2	periodl	RW	Timeout Period - 1 (bits [15:0])						
3	periodh	RW	Timeout Period - 1 (bits [31:16])						
4	snapl	RW	Counter Snapshot (bits [15:0])						
5	snaph	RW	Counter Snapshot (bits [31:16])						

通过这个表格，我们可以看到，定时器有状态寄存器，控制寄存器，定时器周期的

高 16 位，低 16 位，还有 snap 的高 16 位，低 16 位，我们用到的是前 3 个寄存器，snap 寄存器还没有弄到。

我们下面的实验是通过定时器 1 来控制 4 个 LED 的闪烁，而定时器 2 是用来改变定时器 1 的定时器周期，这样，我们就可以看到，4 个 LED 的闪烁频率在不断的变化。

下面我们来看看源代码

```
/*
 * =====
 *      Filename:  main.c
 *      Description:  定时器试验
 *      Version:  1.0.0
 *      Created:  2010.4.16
 *      Revision:  none
 *      Compiler:  Nios II 9.0 IDE
 *      Author:  马瑞 (AVIC)
 *      Email:  avic633@gmail.com
 * =====
 */

/*-----
 *  Include
 *-----*/
#include <stdio.h>
#include <sys/unistd.h>
#include <io.h>
#include <string.h>

#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_timer_regs.h"
#include "alt_types.h"
#include "sys/alt_irq.h"
#include "../inc/sopc.h"

/*-----
 *  Variable
 *-----*/
static void timer_init(void);    //初始化中断
```

```

int i = 0,j = 0,flag;
alt_u32 timer_prd[4] = {5000000, 10000000, 50000000, 100000000};
//这四个是定时器的时钟数
//定时器的定时时间的计算方法是：定时器的时钟数/定时器的时钟周期
//我用的系统时钟是100MHz，所以，上面的四个的定时时间就为{0.05s, 0.1s, 0.5s, 1s}
/*
 * ===  FUNCTION
 *
 * Name:  main
 * Description:
 *
 *
 *
 *
 */
int main(void)
{
    //初始化Timer
    timer_init();

    while(1);

    return 0;
}
/*
 * ===  FUNCTION  =====
 *
 * Name:  ISR_timer
 * Description:
 *
 *
 *
 */
static void ISR_timer1(void *context, alt_u32 id)
{
    //控制流水灯闪烁，一共四个LED
    LED->DATA = 1<<i;

    i++;

    if(i == 4)
        i = 0;
}

```

```

//清除Timer中断标志寄存器
IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER1_BASE, 0x00);
}

/*
 * === FUNCTION =====
 *      Name:  ISR_timer2
 *      Description:  通过定时器2来改变定时器1的周期，改变后需要重新启动定时器
 * =====
 */
static void ISR_timer2(void *context, alt_u32 id)
{
    //改变定时器1的周期
    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER1_BASE, timer_prd[j]);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER1_BASE, timer_prd[j] >> 16);

    //重新启动定时器
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER1_BASE, 0x07);

    //闪烁频率先高后低然后又变高
    if(j == 0)
        flag = 0;
    if(j == 3)
        flag = 1;

    if(flag == 0){
        j++;
    }
    else{
        j--;
    }

    //清除中断标志位
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER2_BASE, 0);
}

/*
 * === FUNCTION =====
 *      Name:  timer_init
 *      Description:  定时器初始化

```

```

* =====
*/
void timer_init(void)    //初始化中断
{
    //清除Timer1中断标志寄存器
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER1_BASE, 0x00);
    //设置Timer1周期，这里输入的是时钟周期数
    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER1_BASE, 100000000);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER1_BASE, 100000000 >> 16);
    //允许Timer1中断
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER1_BASE, 0x07);
    //注册Timer1中断
    alt_irq_register(TIMER1_IRQ, (void *)TIMER1_BASE, ISR_timer1);

    //清除Timer2中断标志寄存器
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER2_BASE, 0x00);
    //设置Timer2周期
    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER2_BASE, 500000000);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER2_BASE, 500000000 >> 16);
    //允许Timer2中断
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER2_BASE, 0x07);
    //注册Timer2中断
    alt_irq_register(TIMER2_IRQ, (void *)TIMER2_BASE, ISR_timer2);
}

```

上面的方式是通过 HAL 提供的 API 实现的，当然我们也可以通过我以前提供的方法实现，建立一个结构体，直接对寄存器进行赋值，这样的方法我更喜欢，清晰而且结构完全自己控制。下面提供给大家一个这个结构体，实现方法大家自己实现吧，很简单，一句一句替换就可以了。建立结构体的内容根据下面的表格决定。

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	(1)						RUN TO
1	control	RW	(1)			STOP	START	CONT	ITO
2	periodl	RW	Timeout Period - 1 (bits [15:0])						
3	periodh	RW	Timeout Period - 1 (bits [31:16])						
4	snapl	RW	Counter Snapshot (bits [15:0])						
5	snaph	RW	Counter Snapshot (bits [31:16])						

```

typedef struct{
    union{
        struct{
            volatile unsigned long int TO    :1;
            volatile unsigned long int RUN   :1;
            volatile unsigned long int NC    :30;
        }BITS;
        volatile unsigned long int WORD;
    }STATUS;

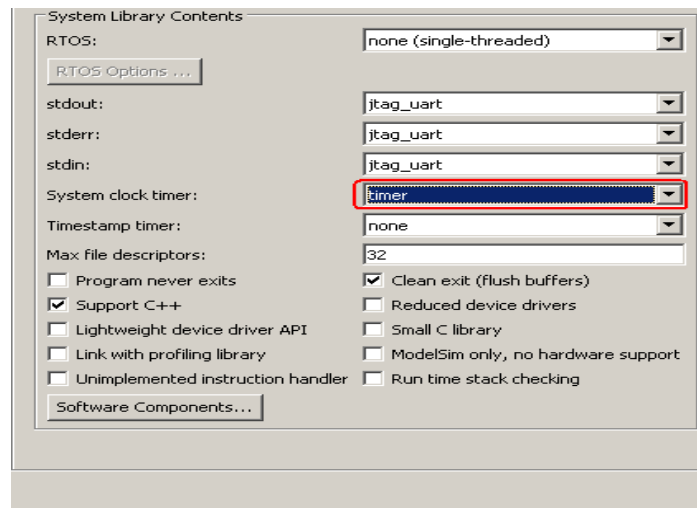
    union{
        struct{
            volatile unsigned long int ITO   :1;
            volatile unsigned long int CONT :1;
            volatile unsigned long int START:1;
            volatile unsigned long int STOP :1;
            volatile unsigned long int NC    :28;
        }BITS;
        volatile unsigned long int WORD;
    }CONTROL;

    volatile unsigned long int PERIODL;
    volatile unsigned long int PERIODH;
    volatile unsigned long int SNAPL;
    volatile unsigned long int SNAPH;
}TIMER;

```

有了这个结构体就可以按照我之前给大家讲的方法实现定时器功能了。到此，我说的定时器的方法就讲完了，同时用两个定时器的问题也能够得以解决了。

下面，我给大家一个有关定时器系统时钟的例程作为参考，但我不建议大家使用，原因我已经说过了。这个函数实现的是每隔一秒点亮一个 LED，一共 4 个 LED。首先需要软件设置一下，如下图所示，进入系统库属性，在 System clock timer 选项框中选择你要的定时器。选择好以后，需要重新编译才行。



```

/*
 * =====
 *      Filename:  main.c
 *      Description:
 *      Version:  1.0.0
 *      Created:  2010.4.16
 *      Revision:  none
 *      Compiler:  Nios II 9.0 IDE
 *      Author:   马瑞 (AVIC)
 *      Email:    avic633@gmail.com
 * =====
 */

/*-----
 *      Include
 *-----*/
#include <stdio.h>
#include <stdlib.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"
#include "sys/alt_irq.h"
#include "../inc/sopc.h"
#include "sys/alt_alarm.h"

/*-----
 *      Function prototypes
 *-----*/

```

```

alt_u32 my_alarm_callback(void *context);

/*-----
 * Variable
 *-----*/

unsigned int i = 0;
unsigned int alarm_flag;

/*-----
 * Define
 *-----*/

#define INTEVAL_TICK 1
#define DEBUG

/*
 * == FUNCTION =====
 *      Name:  main
 * Description:
 * =====
 */

int main(void)
{
    //调用API函数的变量
    alt_alarm alarm;
    //启动系统时钟服务
    if(alt_alarm_start(&alarm,INTEVAL_TICK,my_alarm_callback,NULL) < 0){
        #ifdef DEBUG
            printf("Error: No system clock available\n");
        #endif
        exit(0);
    }
    else{
        #ifdef DEBUG
            printf("Success: System clock available\n");
        #endif
    }

    while(1){
        if(alarm_flag != 0){
            LED->DATA = 1<<i;
            i++;

            if(i == 4)

```

```

        i = 0;

        alarm_flag = 0;
    }

};

return 0;
}
/* === FUNCTION =====
 *      Name:  my_alarm_callback
 *  Description:
 *  =====
 */
alt_u32 my_alarm_callback(void *context)
{
    alarm_flag = 1;
    //INTEVAL_TICK的值决定下次进入定时器中断函数的时间
    return INTEVAL_TICK;
}

```

第十三章 SDRAM

SDRAM

通过本章，您可以了解到 NIOS 系统中 SDRAM 相关内容及用法。

本章分为以下几个部分：

- 一、简介
- 二、软件开发

一、简介

这一节，我们来聊聊 SDRAM 吧。作为 NIOS 系统中最重要的一個外部器件，它担任着重要的角色，大家对它也应该很熟悉。每次上电的时候，FPGA 都会把 FLASH 中的程序送到 SDRAM 中运行，之所以这样做就是因为它的速度很快，但它掉电是要丢失数据的，所以要把数据存到 FLASH 中。

有关 SDRAM 的理论知识我在这里不说了，不知道的百度 google 一下都可以。其实在 NIOS II 开发过程中，就算你对 SDRAM 的理论知识不了解，也不耽误你对它的使用。SOPC builder 都已经完美的将它驱动起来，我们只要知道怎么使用它就可以了。下面，我们就来讲讲他的使用方法，其实真的很简单。

在我们讲第一节的时候，我们就已经讲了如何构建 SDRAM 的控制器了，我在这里不再重复了，假设你已经构建好了，我主要讲一下有关软件的部分。

二、软件开发

首先打开 NIOS II 9.0 IDE 软件，打开后，我们来看看 system.h 文件，确定一下 SDRAM 控制器模块是否已经加入进来。如果加入，有下面的内容出现

```
#define SDRAM_NAME "/dev/sdram"  
#define SDRAM_TYPE "altera_avalon_new_sdram_controller"  
#define SDRAM_BASE 0x01000000  
.....
```

接下来，我们开始编写有关 SDRAM 的软件代码，代码很简单，如下所示

```
/*  
 * =====  
 *      Filename:  main.c  
 *      Description: SDRAM读写试验  
 *      Version:   1.0.0  
 *      Created:   2010.4.16  
 *      Revision:  none  
 *      Compiler:  Nios II 9.0 IDE  
 *      Author:    马瑞 (AVIC)  
 *      Email:     avic633@gmail.com  
 * =====  
 */
```

```

/*-----
 * Include
 *-----*/
#include <stdio.h>
#include "../inc/sopc.h"
#include "system.h"
#include "string.h"

/*-----
 * Variable
 *-----*/
unsigned short * ram = (unsigned short *) (SDRAM_BASE+0x10000); //SDRAM地址
/*
 * == FUNCTION =====
 *      Name: main
 * Description: 函数主程序
 * =====
 */
int main(void)
{
    int i;

    memset(ram,0,100);
    //向ram中写数据，当ram写完以后，ram的地址已经变为(SDRAM_BASE+0x10100)
    for(i=0;i<100;i++){
        *(ram++) = i;
    }

    //逆向读取ram中的数据
    for(i=0;i<100;i++){
        printf("%d\n",*(--ram));
    }

    return 0;
}

```

程序很简单，就是向指定的 SDRAM 中赋值。在这个程序里面有几个地方需要说明一下。首先，我在程序前面定义了一个 unsigned short 类型的指针变量 ram，并将其指向 SDRAM+0x10000 这个位置。之所以设置为 unsigned short 数据类型，是因

为我们用的 SDRAM 是 16 位数据总线的。而将其指向 SDRAM+0x10000 是因为在 NIOS II 运行时用到 SDRAM 的部分空间 我们必须避开这部分空间 以免运行错误。0x10000 这个值不是固定的，只要避开 SDRAM 的那部分空间就可以了。除此之外还有一个地方需要注意，就是当我们对 sdram 赋值以后，指针就会向后移动，指向下一个地址空间，每加一次，地址都会向后面移动 16 位。假如我们现在是在 SDRAM+0X10000 这个位置，当指针向后移动一次以后，地址就变为了 SDRAM+0X10002，再加一次就变为了 SDRAM+0X10004，以此类推。这些都是内部自动处理的，不需要我们来参与，我们只要知道就可以了。

其实我讲这部分内容是想告诉大家，SDRAM 控制器一旦构建好以后，我们对 SDRAM 的处理就像对内部地址一样，我们可以随意的进行赋值和读取。对于开发板上的 64Mbit 的 SDRAM 其实有很少一部分用给 NIOS 系统，其余部分都在空闲，大家是不是觉得很浪费呢。其实在有些情况下，我们就可以利用起这部分资源，比如在某个系统中，我们需要将外设接收到的数据缓存一下，我们就可以用这部分空闲的 SDRAM 空间来处理。

在 C 语言中，如果我们要接收比较大的数据，还有另一种处理方法，那就是借助堆（heap）。可能有些人对堆和栈还分不清楚，我在这简单解释一下。栈（stack）由系统自动分配。例如，声明在函数中一个局部变量 int b，系统自动在栈中为 b 开辟空间。而堆（heap）需要程序员自己申请，并指明大小。有人用这样一个比喻来解释堆和栈的区别，非常形象贴切：使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大，这个人真是太有才了。下面我来写一个堆的代码，这部分代码是节选的，并不完全，功能是通过 xmodem 协议接收数据，并将其内容放到堆里面。如下所示

```
/*
 * == FUNCTION =====
 *      Name:  main
 *      Description: 主函数
 * =====
 */
int main()
{
    char *ram = (char *)malloc ( sizeof(char)* 500000 );

    if ( ram == NULL ) {
        fprintf ( stderr, "\ndynamic memory allocation failed\n" );
    }
}
```

```

        exit (EXIT_FAILURE);
    }

    uart.init();

    /*----- FLASH -----*/

    while(1){
        if(uart.mode_flag){
            xmodem.xmodem_rx();
            printf("ram_cnt:%d\n",ram_cnt);
            parse_srecord_buf(ram,ram_cnt);
            printf("successful!");
            ram_cnt = 0;
            uart.mode_flag = 0;
        }

    }

    free (ram);

    return 0;
}

```

看了上面的代码大家应该了解了堆的用法了吧，它是通过 malloc 向系统中申请空间的。申请成功以后，就会返回申请地址的首地址，失败则返回 NULL。到底申请在什么地方，我们可以通过打印指针来得知，但并没这个必要，因为这些都是系统来处理的，我们得到了首地址然后用就可以了。需要注意一点，我们申请完的地址用完以后需要释放，用 free 来释放就可以了。如果不释放，可能会出现内存泄露问题，到时候麻烦就大了，具体大到什么程度我也不知道，呵呵。

可能有人会问，为什么不用栈来处理这个问题呢？这就跟编译有关系了，在编译过程中，系统会将栈需要的空间加到代码中，也就是说如果你在代码中用栈来处理大的数据，那么你编译以后的代码会非常大，你下载到 flash 以后，加载到 sdram 中的时间也会非常之长。而堆这不会，系统对堆的处理方式是何时用合适分配，并不占代码空间。

说到这，有关 SDRAM 部分的内容讲完了。总结一下，使用 SDRAM 有两种方法，第一种是直接对 SDRAM 地址处理；第二种方法就是利用堆来处理。好了，这部分内容就讲到这吧，如果大家对此有疑问，或者发现我讲的内容有问题可以直接跟我联系，邮箱：avic633@gmail.com；qq:984597569，谢谢。

第十四章 EPCS 下载

EPCS 下载

通过本章，您可以了解到 NIOS 系统中如何将配置信息和软件烧写到 EPCSX 中。

本章分为以下几个部分：

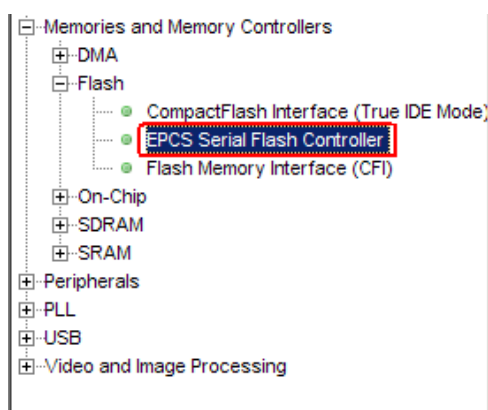
- 一、简介
- 二、硬件设置
- 三、软件设置

一、简介

这一节，应网友的要求，我们来讲解如何将 FPGA 配置文件和 NIOS 的程序下载到 EPCSx (x 为 1,4,16...) 里面。首先说几句，之所以我们要将程序下载到 EPCSX 中，而不下载到并行 FLASH 中，是因为我们可以将并行的 FLASH 去掉，这样就可以节省 32 根引脚，在布 PCB 的时候也可以节省空间，何乐而不为，我的改进的核心板就是这样做的。有人会问，并行的速度要快很多啊，其实我们存储在 FLASH 的代码，每次上电或复位，仅仅需要读取一次而已，在速度上没有太大的影响。除非你有特殊要求，如在程序中需要频繁的操作 FLASH，这样才需要并行的满足速度的要求。好了，下面我们来进行操作。

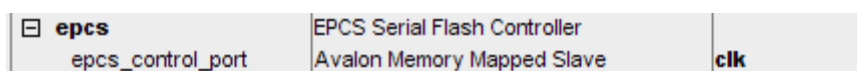
二、硬件设置

首先在软核中添加 EPCS Serial Flash controller，如下图所示红圈处

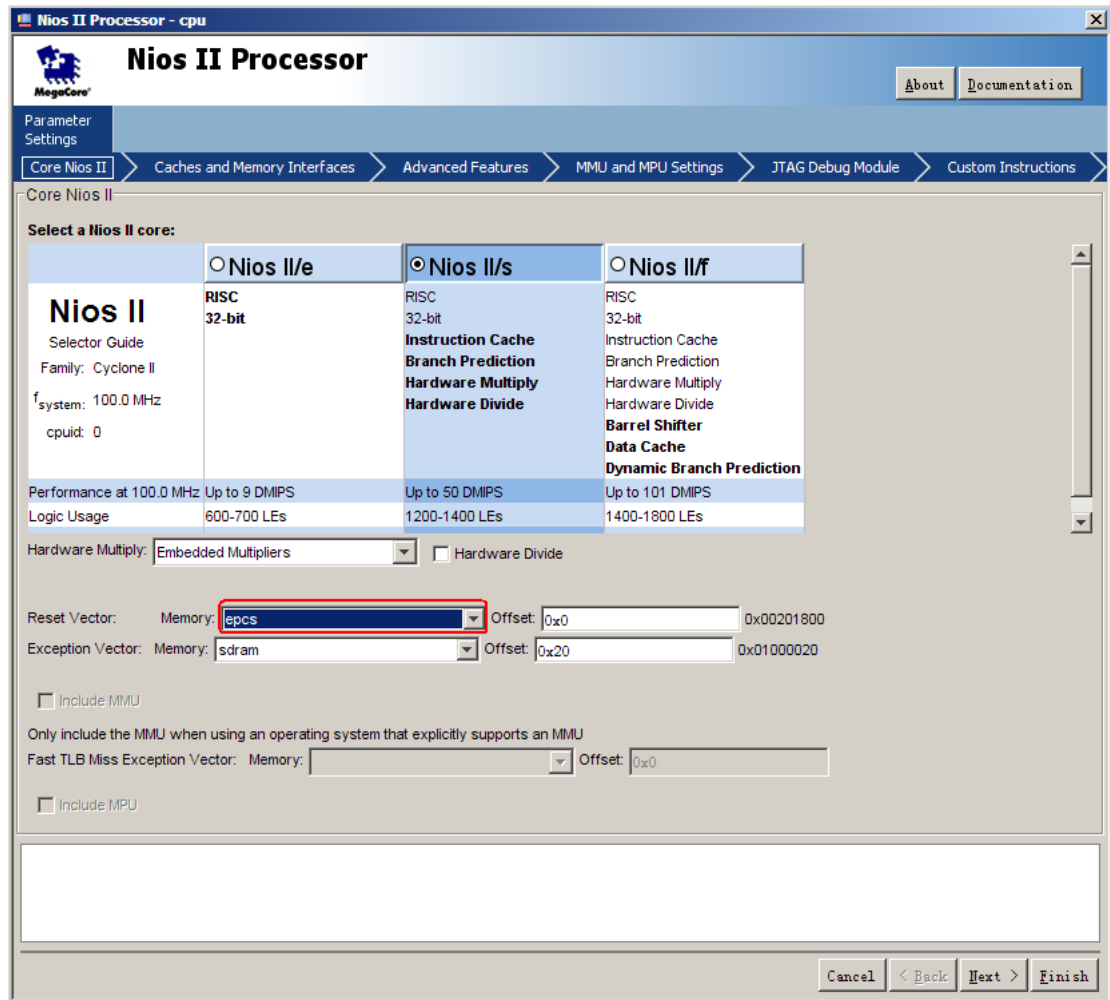


双击红圈处后，没什么需要修改的，直接点击 Finish 完成添加。

然后修改名字，完成后，如下图所示，其实修不修改都可以，只是为了看起来简单而已



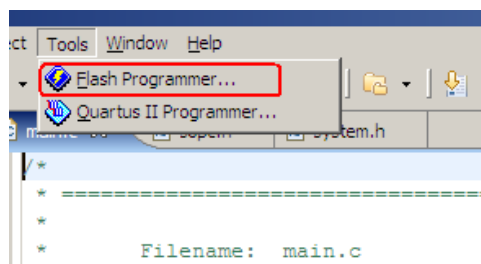
修改好以后，我们还有一步需要处理，双击 cpu，点击后，如下图所示红圈处，我们要 memory 选择为 epcs，也就是说，上电后，读取 epcs 中的数据。大家记得我在讲并行 FLASH 的时候讲的，这个地方应该选择 CFI_FLASH 吧，也就是说，上电或复位以后，你需要 NIOS 读取哪里的数据，就选择哪一个。因为我们要实现从 EPCS 中读取数据，所以我们这里选择 epcs。



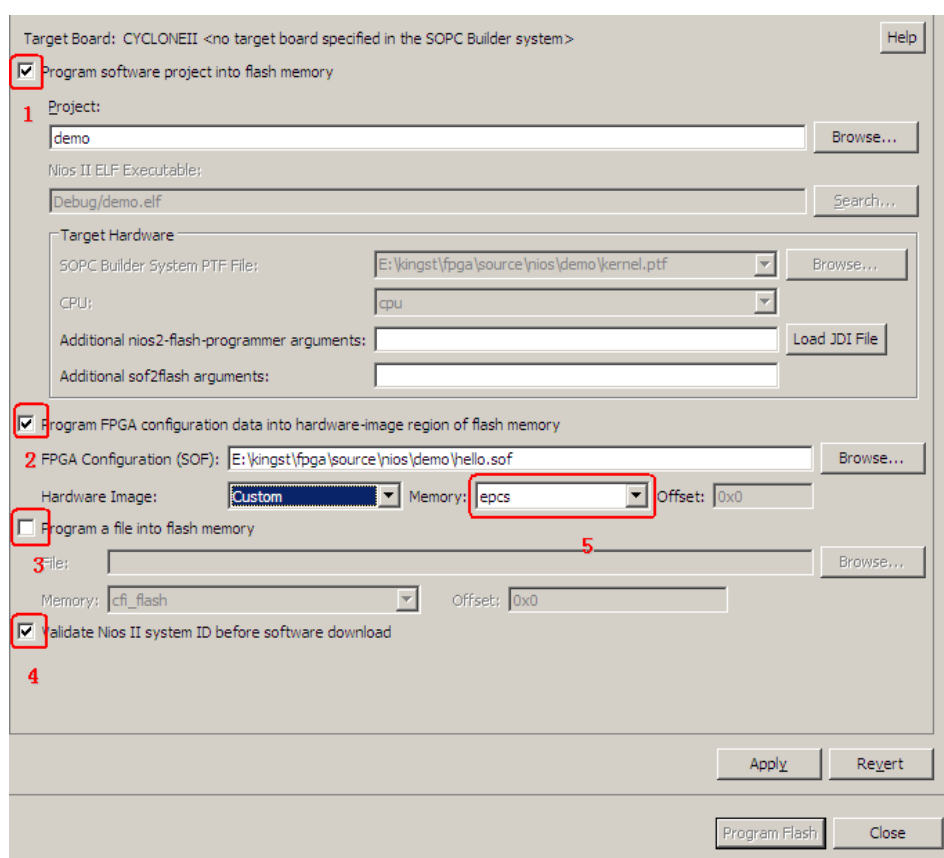
上面都完成了以后,接下来,就是自动配置地址,中断,编译,等待,编译,等待,硬件部分的工作就结束了。

三、 软件设置

下面我们来看看在 NIOS 9.0 IDE 中都需要进行哪些配置。在这里,我们假设我们需要下载的程序都已经编译成功。那么我们就开始下载程序到 EPCSx 中,点击下图所示红圈处,



点击后，如下图所示，其中需要将红圈 1、2、4 选中，红圈 3 不用选，选中红圈 1 是将 NIOS 软件程序写到 FLASH 中，而到底是写到哪里，是由前面我们讲过的 CPU 中 Reset Vector 的 memory 决定的。也就是说，我们之前选择了 epcs，那么我们就将代码下载到 EPCSx 中了。选中红圈 2 是将 FPGA 的配置文件下载到 FLASH 中，而这里到底下载到哪是由红圈 5 处决定的，我们在这里还是选择 epcs，就是将配置文件下载到 EPCSx 中。其实 EPCSx 实质就是一种串行的 FLASH。再说说红圈 3，红圈 3 是将文件下载到 flash 中，比如说字库文件啊，波形文件啊等等，将这些文件直接存储到 FLASH 中，只需要读取就可以了。不过这个选项跟我们下载配置文件和 NIOS 程序时没有关系的。所以这里不选择它。红圈 4 的作用我以前已经说过了，在这里不重复了。



上面选择好以后，点击 Apply，然后点击 Program Flash，就可以开始烧写 FLASH 了。

至此，如何将 FPGA 配置文件和 NIOS 的程序下载到 EPCSx (x 为 1,4,16...) 里面的过程就讲完了。很简单吧，大家只要稍加注意就可以了。

好了，这一节的内容就讲完了，有问题的可以给我直接留言，或者加入我的 qq：984597569，或者加入最新的群：109711029 (107122106 已满)，当然也可以发邮件给我：avic633@gmail.com，谢谢大家支持。

第十五章 FLASH 编程

FLASH 编程

通过本章，您可以了解到 NIOS 系统中 FLASH 编程的相关内容，其中包括并行 FLASH 和 EPCSX 系列。

本章分为以下几个部分：

- 一、简介
- 二、软件开发

一、 简介

这一节，我们来讲一讲如何使用 FLASH。在嵌入式系统中，Flash 是最常用组件之一。许多使用过 FLASH 的都知道，FLASH 的特点是“读来容易写来难”。通常情况，我们是可以直接读出 FLASH 中的内容的，但如果要写入数据，就要发送一长串命令，比如像：555，AA，2AA，55，555，A0，PA，PD 就表示对 PA 地址写入数据 PD。

而在 NIOS 开发过程中，FLASH 的这些繁琐的擦写过程 ALTERA 都帮我们做好了，它通过 API 给我们提供了完善的读写函数，我们只需要调用函数就可以完成读写 FLASH 的操作了。不过这些也是有前提条件的，那就是 FLASH 要满足 CFI 标准或者是 EPCSX 串行 FLASH(与 EPCSX 兼容的也是可以的)。也就是说，不满足满足这个条件的 FLASH 还需要您自己处理了，呵呵！下面我们来研究一下 NIOS 下的 FLASH 是如何操作的。

二、 软件开发

在前面，我们就已经详细讲述了 FLASH 和 EPCS 模块的构建过程，在这里，我们假设您已经构建好了 CFI FLASH 模块或者 EPCS 控制模块，由于 CFI FLASH 和 EPCS 的软件操作方式完全一致，所以我们以下内容不对他们加以区分。下面我们主要来讲 FLASH 的软件操作是如何实现的。

Altera 提供了两种类型的函数 提供给客户 :Simple Flash Access(简单的 Flash 访问)，以及 Fine-Grained Flash Access (细粒度 Flash 访问)。

一般情况下，我还是推荐大家使用 Fine-Grained Flash Access (细粒度 Flash 访问) 函数，比 Simple Flash Access 也复杂不了多少，但可以避免通常的跨块擦除问题。Flash 是按照 Block 组织起来的，通常一次擦除一整个块。例如，我们如果写 Flash 的时候跨越了两个块，而其后还跟随了其他数据，这样我们就有可能将其他数据同时擦除掉，导致数据的丢失，这样后果就很严重了。

在这里，我们介绍几个常用的函数，如果想了解的更详细，请参考 ALTERA 提供的文档资料。

首先介绍第一步：打开 Flash，就像 C 程序打开硬盘中的数据文件一样，使用之前要打开 FLASH。我们使用 `alt_flash_open_dev()` 打开 Flash，成功打开后返回一个句柄。比如，下面是使用这个函数的片断：

```
alt_flash_fd* fd;  
fd = alt_flash_open_dev(FLASH_NAME);
```

其中，FLASH_NAME 可以在 system.h 中找到它的定义，我的定义如下所示

```
CFI FLASH 定义为  
#define CFI_FLASH_NAME "/dev/cfi_flash"  
EPCSX 定义为  
#define EPCS_FLASH_NAME "/dev/epcs_flash"
```

读出 Flash 使用函数：alt_read_flash,原型如下：

```
int alt_read_flash( alt_flash_fd* fd,  
                   int offset,  
                   void* dest_addr,  
                   int length );
```

使用完以后需要将 FLASH 关掉，其原型如下：

```
void alt_flash_close_dev(alt_flash_fd* fd );
```

Fine-Grained Flash Access 机制还提供了如下几个函数：alt_get_flash_info(), alt_erase_flash_block(), alt_write_flash_block()。

alt_get_flash_info()可以提取 Flash 的信息，比如包含几个区域，每个区域有几个块，每个块的大小等等。它的原型如下：

```
int ret_code = 0;  
int number_of_regions=0;  
flash_region* regions;  
ret_code = alt_get_flash_info(fd, &regions, &number_of_regions);
```

这里涉及到一个结构：flash_region,原型如下：

```
typedef struct flash_region {  
    int offset; /* region相对FLASH首地址的偏移量 */  
    int region_size; /* region的大小*/  
    int number_of_blocks; /* 每个region中block的数量 */  
    int block_size; /* block的大小*/  
}flash_region;
```

擦除一个块使用函数：alt_flash_fd,函数原型如下：

```
int alt_erase_flash_block( alt_flash_fd* fd,int offset,int length) ;
```

写入一个块使用函数：alt_write_flash_block,函数原型如下：

```
int alt_write_flash_block( alt_flash_fd* fd,  
                           int block_offset,  
                           int data_offset,  
                           const void *data,  
                           int length) ;
```

几个常用的函数就介绍完了，下面我们就用这几个函数来操作 FLASH 吧。

```
#include "sys/alt_flash.h"  
#include <stdlib.h>  
#include <stdio.h>  
#include <unistd.h>  
  
#define BUF_SIZE 100  
#define FLASH_OFFSET 0x5000  
/*  
 * == FUNCTION =====  
 *      Name:  main  
 *      Description:  主函数，函数入口  
 * =====  
 */  
int main (void)  
{  
    int i;  
    alt_flash_fd* fd;  
    flash_region* regions;  
    int number_of_regions;  
    int ret_code = 0x0;  
    unsigned char source[BUF_SIZE];  
    unsigned char dest[BUF_SIZE];  
  
    for(i=0;i<100;i++){  
        source[i] = i;  
    }  
    fd = alt_flash_open_dev("/dev/cfi_flash");  
    //fd = alt_flash_open_dev( "/dev/epcs_flash" );//EPCSX  
    if (fd){
```

```

/*得到FLASH信息*/
ret_code = alt_get_flash_info(fd, &regions, &number_of_regions);
for(i=0;i<number_of_regions;i++){
    printf("Start 0x%8x End 0x%8x Number of Blocks %3d Block Size
0x%8x \n",

                                (regions+i)->offset,
                                (regions+i)->region_size+(regions+i)->offset,
                                (regions+i)->number_of_blocks,
                                (regions+i)->block_size);
}

alt_erase_flash_block(fd, FLASH_OFFSET, BUF_SIZE);
//读FLASH中的数据,成功后返回值为0
ret_code = alt_read_flash(fd,FLASH_OFFSET,dest,BUF_SIZE);

if(ret_code != 0){
    printf("Can't read flash device\n");
    exit(-1);
}
else{
    printf("Read Flash Device Successfully.\n");
}

//打印读取的数据
for(i=0;i<100;i++){
    printf("%d\n",dest[i]);
}

//向FLASH中写数据,成功后返回值为0
ret_code = alt_write_flash(fd,FLASH_OFFSET,source,BUF_SIZE);

if(ret_code != 0){
    printf("Can't write flash device\n");
    exit(-1);
}
else{
    printf("Write Flash Device Successfully.\n");
}

//读FLASH中的数据,成功后返回值为0
ret_code = alt_read_flash(fd,FLASH_OFFSET,dest,BUF_SIZE);

```

```

        if(ret_code != 0){
            printf("Can't read flash device\n");
            exit(-1);
        }
        else{
            printf("Read Flash Device Successfully.\n");
        }

        //打印读取的数据
        for(i=0;i<100;i++){
            printf("%d\n",dest[i]);
        }
    }

    alt_flash_close_dev(fd);
}

```

上述程序对于 CFI FLASH 和 EPCSX 是通用的，如果操作 EPCSX,则将 alt_flash_open_dev("/dev/cfi_flash")参数改为“ /dev/epcs_flash” 就可以了。

下面我们来看一看通过 alt_get_flash_info(fd, ®ions, &number_of_regions) 得到的一些 CFI FLASH 信息，

Start 0x0	End 0x4000	Number of Blocks	1	Block Size	0x4000
Start 0x4000	End 0x8000	Number of Blocks	2	Block Size	0x2000
Start 0x8000	End 0x10000	Number of Blocks	1	Block Size	0x8000
Start 0x10000	End 0x200000	Number of Blocks	31	Block Size	0x10000

通过上面的信息，我们可以看出，AM29LV160B 这款 FLASH 的包括了 1 个 16Kbyte ,2 个 8Kbyte ,1 个 32Kbyte 和 31 个 64Kbyte 字节的 block ,这个是 FLASH 处在 8 位模式下的情况，结构很奇怪吧。

我们再来看通过 alt_get_flash_info(fd, ®ions, &number_of_regions)得到的一些 EPCSX 的信息，

Start 0x0	End 0x80000	Number of Blocks	8	Block Size	0x10000
-----------	-------------	------------------	---	------------	---------

通过上面信息我们可以发现，我用的这款 EPCSX 是 EPCS4,包括了 8 个 128Kbit 的 block。这个结构就相对比较简单了。

好了，这一节内容就讲完了。

如果大家对此有疑问，或者发现我讲的内容有问题可以直接跟我联系，邮箱：
avic633@gmail.com；qq:984597569，同时欢迎大家加入我们的 NIOS 群：
109711029，大家共同探讨 NIOS 技术！

第十六章 AVALON

基于 AVALON 的 IP 定制

通过本章，您可以了解到 NIOS 系统中基于 AVALON IP 定制的详细过程。

本章分为以下几个部分：

- 一、简介
- 二、DHL 模块设计
- 三、硬件设计
- 四、软件开发

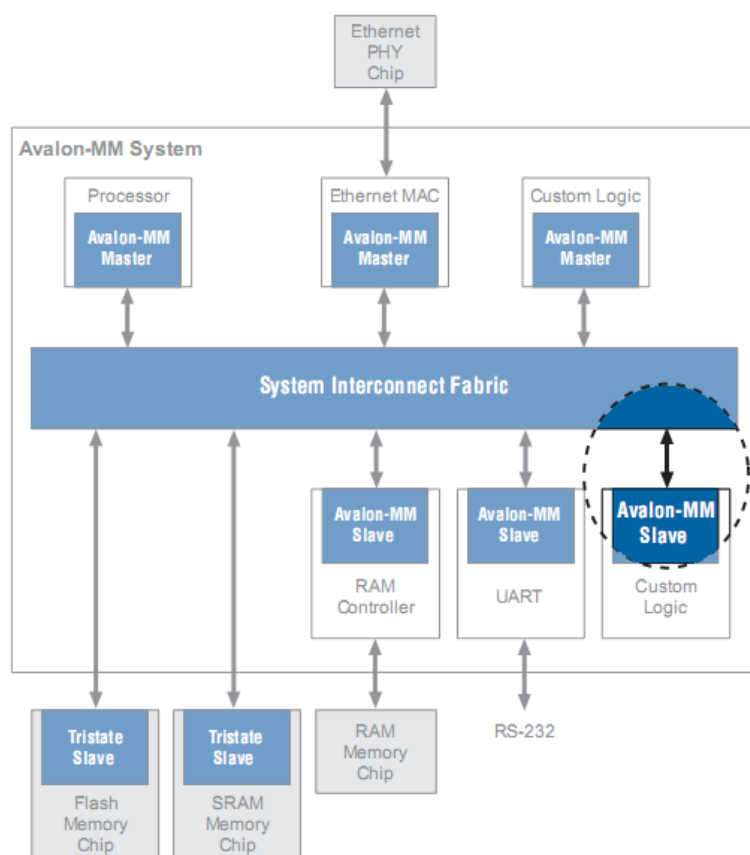
一、简介

NIOS II 是一个建立在 FPGA 上的嵌入式软核处理器，除了可以根据需要任意添加已经提供的外设外，用户还可以通过定制用户逻辑外设和定制用户指令来实现各种应用要求。这节我们就来研究如何定制基于 Avalon 总线的用户外设。

SOPC Builder 提供了一个元件编辑器，通过这个元件编辑器我们就可以将我们自己写的逻辑封装成一个 SOPC Builder 元件了。下面，我们就以 PWM 实验为例，详细介绍一下定制基于 Avalon 总线的用户外设的过程。

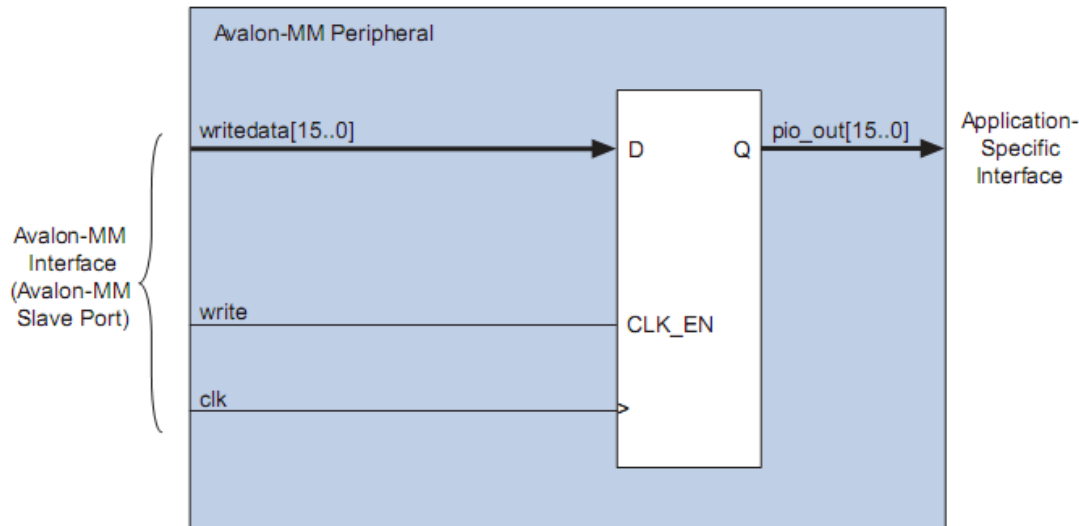
我们要将的 PWM 是基于 Avalon 总线中的 Avalon Memory Mapped Interface (Avalon-MM)，而 Avalon 总线还有其他类型的设备，比如 Avalon Streaming Interface (Avalon-ST)、Avalon Memory Mapped Tristate Interface 等等，在这里我就不详细叙述了，需要进一步了解的请参考 ALTERA 公司的《Avalon Interface Specifications》(mnl_avalon_spec.pdf)。

Avalon-MM 接口是内存映射系统下的用于主从设备之间的读写的接口，下图就是一个基于 Avalon-MM 的主从设备系统。而我们这节需要做的就是下图高亮部分。他的地位与 UART，RAM Controller 等模块并驾齐驱的。



Avalon-MM 接口有很多特点 ,其中最大的特点就是根据自己的需求自由选择信号线 ,不过里面还是有一些要求的。建议大家在看本文之前 ,先看一遍《Avalon Interface Specifications》, 这样就能对 Avalon-MM 接口有一个整体的了解。

下图为 Avalon-MM 外设的一个结构图 ,



理论的就说这些 ,下面我们举例来具体说明 ,让大家可以更好的理解。

二、 DHL 模块设计

我们这一节是 PWM 为例 ,所以首先 ,我们要构建一个符合 Avalon-MM Slave 接口规范的可以实现 PWM 功能的时序逻辑 ,在这里 ,我们利用 Verilog 语言来编写。在程序中会涉及到 Avalon 信号 ,在这里 ,我们说明一下这些信号 (其中 ,方向以从设备为基准)

HDL 中的信号	Avalon 信号类型	宽度	方向	描述
clk	clk	1	input	同步时钟信号
reset_n	reset_n	1	input	复位信号 , 低电平有效
chipselct	chipselct	1	input	片选信号
address	address	2	input	2 位地址 译码后确定寄存器 offset
write	write	1	input	写使能信号
writedata	writedata	32	input	32 位写数据值
read	read	1	input	读使能信号
byteenable	byteenable	1	input	字节使能信号
readdata	readdata	32	output	32 位读数据值

此外,程序中还包括一个 PWM_out 信号,这个信号是 PWM 输出,不属于 Avalon 接口信号。

PWM 内部还包括使能控制寄存器、周期设定寄存器以及占空比设置寄存器。设计中将各寄存器映射成 Avalon Slave 端口地址空间内一个单独的偏移地址。每个寄存器都可以进行读写访问,软件可以读回寄存器中的当前值。寄存器及偏移地址如下:

寄存器名	偏移量	访问属性	描述
clock_divide_reg	00	读/写	设定 PWM 输出周期的时钟数
duty_cycle_reg	01	读/写	设定一个周期内 PWM 输出低电平的始终个数
control_reg	10	读/写	使能和关闭 PWM 输出,为 1 时使能 PWM 输出

程序如下:

```

module PWM(
    clk,
    reset_n,
    chipselect,
    address,
    write,
    writedata,
    read,
    byteenable,
    readdata,
    PWM_out);

input clk;
input reset_n;
input chipselect;
input [1:0]address;
input write;
input [31:0] writedata;
input read;
input [3:0] byteenable;
output [31:0] readdata;
output PWM_out;

reg [31:0] clock_divide_reg;
reg [31:0] duty_cycle_reg;
reg control_reg;
reg clock_divide_reg_selected;
reg duty_cycle_reg_selected;

```

```

reg control_reg_selected;
reg [31:0] PWM_counter;
reg [31:0] readdata;
reg PWM_out;
wire pwm_enable;

//地址译码
always @ (address)
begin
    clock_divide_reg_selected<=0;
    duty_cycle_reg_selected<=0;
    control_reg_selected<=0;
    case (address)
        2'b00:clock_divide_reg_selected<=1;
        2'b01:duty_cycle_reg_selected<=1;
        2'b10:control_reg_selected<=1;
        default:
            begin
                clock_divide_reg_selected<=0;
                duty_cycle_reg_selected<=0;
                control_reg_selected<=0;
            end
    endcase
end

//写PWM输出周期的时钟数寄存器
always @ (posedge clk or negedge reset_n)
begin
    if(reset_n==1'b0)
        clock_divide_reg=0;
    else
        begin
            if(write & chipselect & clock_divide_reg_selected)
            begin
                if(byteenable[0])
                    clock_divide_reg[7:0]=writedata[7:0];
                if(byteenable[1])
                    clock_divide_reg[15:8]=writedata[15:8];
                if(byteenable[2])
                    clock_divide_reg[23:16]=writedata[23:16];
                if(byteenable[3])
                    clock_divide_reg[31:24]=writedata[31:24];
            end
        end
    end
end

```

```

end

//写PWM周期占空比寄存器
always @ (posedge clk or negedge reset_n)
begin
    if(reset_n==1'b0)
        duty_cycle_reg=0;
    else
        begin
            if(write & chipselect & duty_cycle_reg_selected)
                begin
                    if(byteenable[0])
                        duty_cycle_reg[7:0]=writedata[7:0];
                    if(byteenable[1])
                        duty_cycle_reg[15:8]=writedata[15:8];
                    if(byteenable[2])
                        duty_cycle_reg[23:16]=writedata[23:16];
                    if(byteenable[3])
                        duty_cycle_reg[31:24]=writedata[31:24];
                end
            end
        end
    end
end

//写控制寄存器
always @ (posedge clk or negedge reset_n)
begin
    if(reset_n==1'b0)
        control_reg=0;
    else
        begin
            if(write & chipselect & control_reg_selected)
                begin
                    if(byteenable[0])
                        control_reg=writedata[0];
                end
            end
        end
    end
end

//读寄存器
always @ (address or read or clock_divide_reg or duty_cycle_reg or
control_reg or chipselect)
begin
    if(read & chipselect)
        case (address)

```

```

        2'b00:readdata<=clock_divide_reg;
        2'b01:readdata<=duty_cycle_reg;
        2'b10:readdata<=control_reg;
        default:readdata=32'h8888;
    endcase
end

//控制寄存器
assign pwm_enable=control_reg;

//PWM功能部分
always @ (posedge clk or negedge reset_n)
begin
    if(reset_n==1'b0)
        PWM_counter=0;
    else
        begin
            if(pwm_enable)
            begin
                if(PWM_counter>=clock_divide_reg)
                    PWM_counter<=0;
                else
                    PWM_counter<=PWM_counter+1;
            end
            else
                PWM_counter<=0;
        end
    end
end

always @ (posedge clk or negedge reset_n)
begin
    if(reset_n==1'b0)
        PWM_out<=1'b0;
    else
        begin
            if(pwm_enable)
            begin
                if(PWM_counter<=duty_cycle_reg)
                    PWM_out<=1'b1;
                else
                    PWM_out<=1'b0;
            end
            else
                PWM_out<=1'b0;
        end
    end
end

```

```

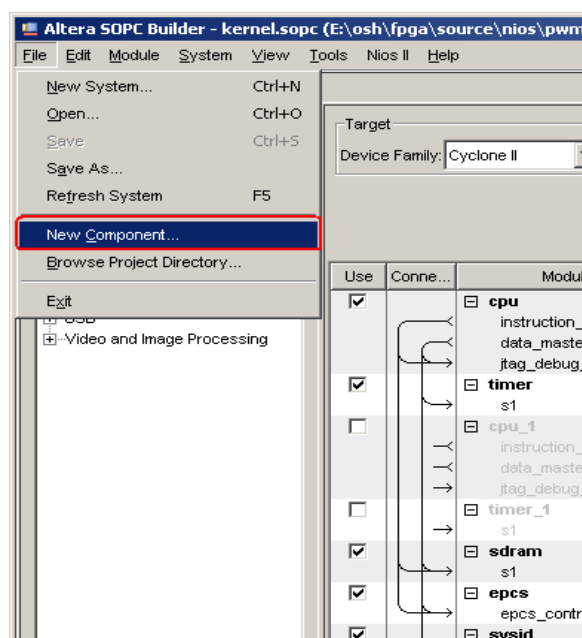
        end
    end
endmodule

```

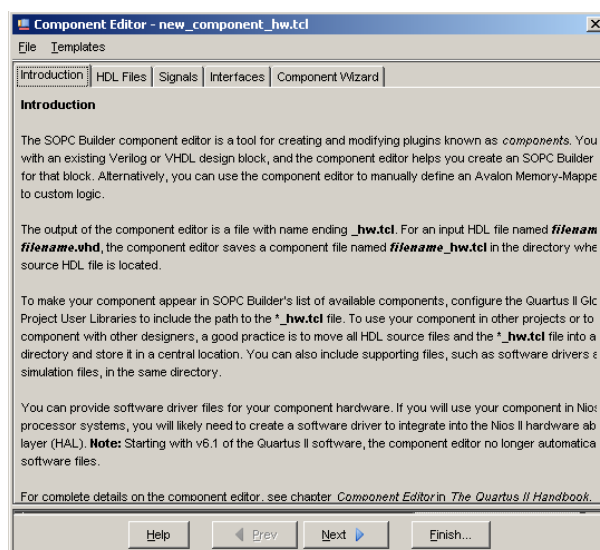
上面的程序保存好以后，命名为 PWM.c，并将其存放到工程目录下。

三、 硬件设计

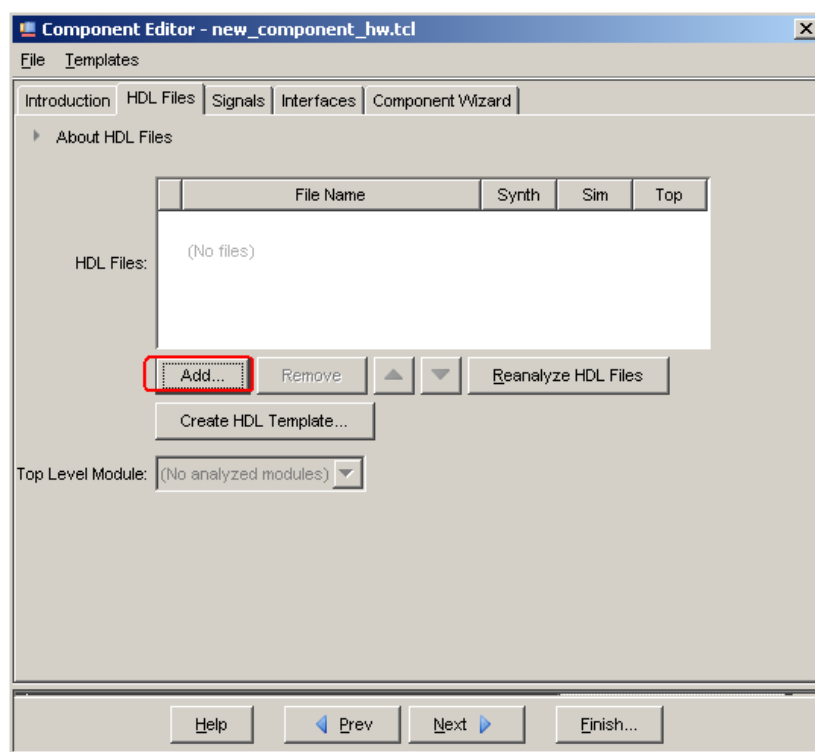
接下来，我们就通过 SOPC Builder，来建立 PWM 模块了。首先，打开 Quartus 软件，进入 SOPC Builder。进入后，点击下图红圈处



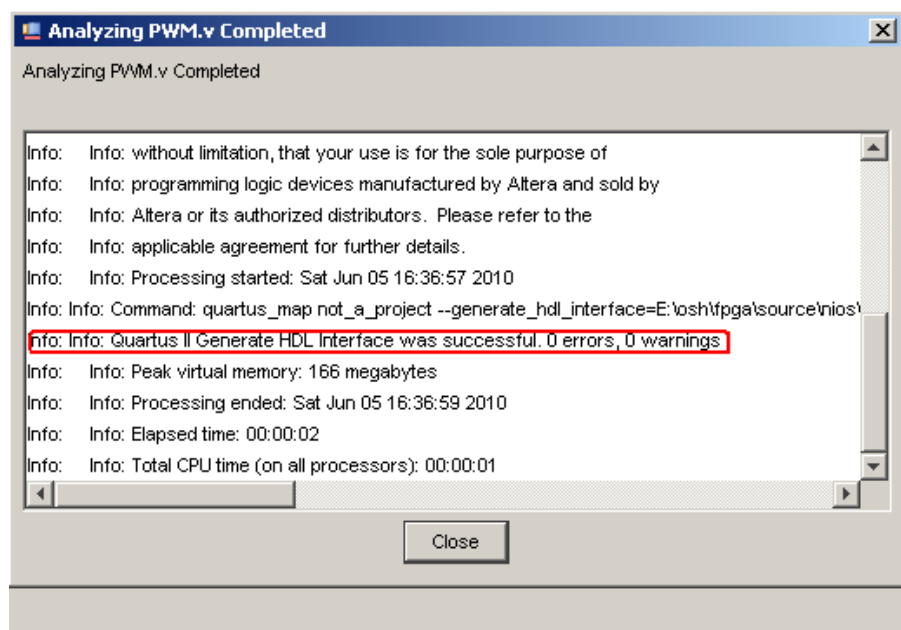
点击后，如下图所示，点击 Next，



点击后，如下图所示，点击下图红圈处，将我们刚才建立的 PWM.v 加进来。（我将 PWM.v 放到了工程目录下的 pwm 文件夹下）

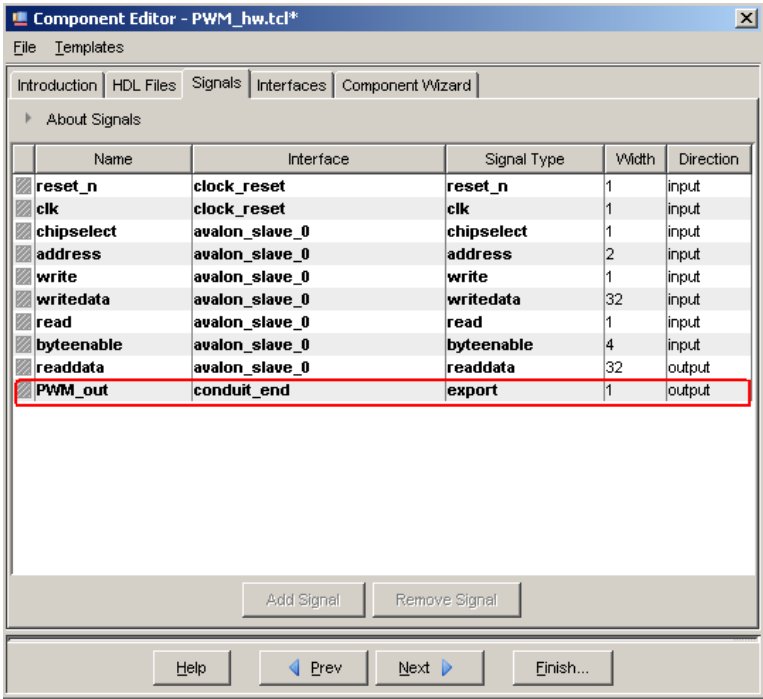


加入后，系统会对 PWM.v 文件进行分析，如下图所示，出现红圈处的文字，说明分析成功，点击 close，关闭对话框。

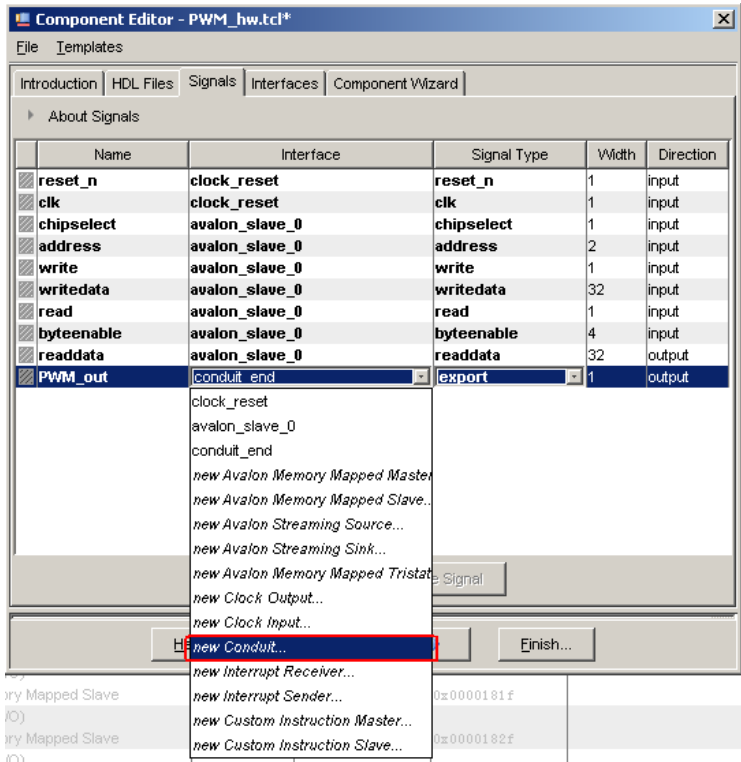


然后点击 Next，如下图所示，通过下图，我们可以看到，PWM.v 中的信号都出现在这里面了。我们可以根据我们的功能要求来配置这些信号，其中，Interface 是

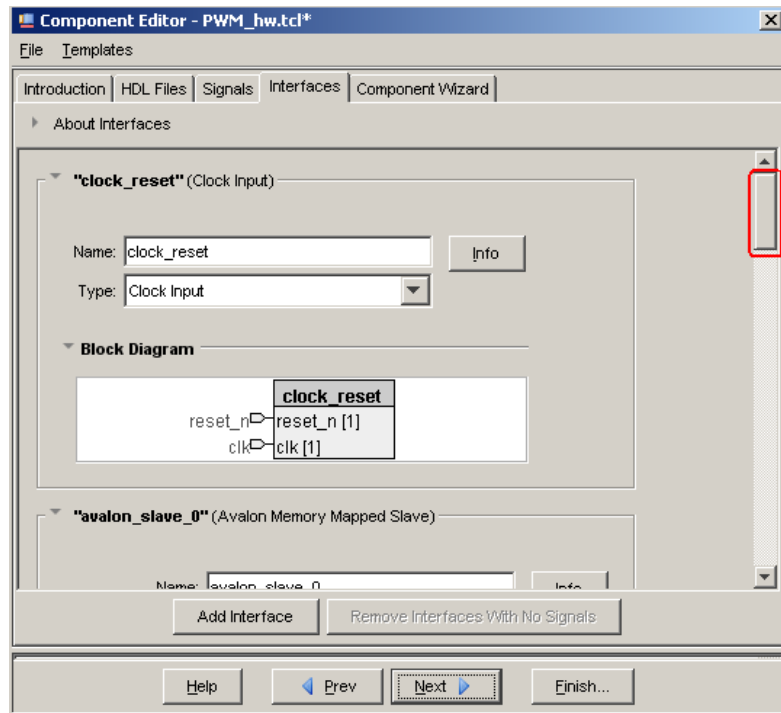
Avalon 接口类型，它包括 Avalon-MM、Avalon-ST、Avalon Memory Mapped Tristate Interface 等等。Signal Type 指的是各个 Avalon 接口类型下的信号类型。PWM.v 中的信号我们已经在前面都介绍过了，大家按照上面的要求设置就可以了。默认情况只有 PWM_out 需要改动，如下图示红圈处设置，



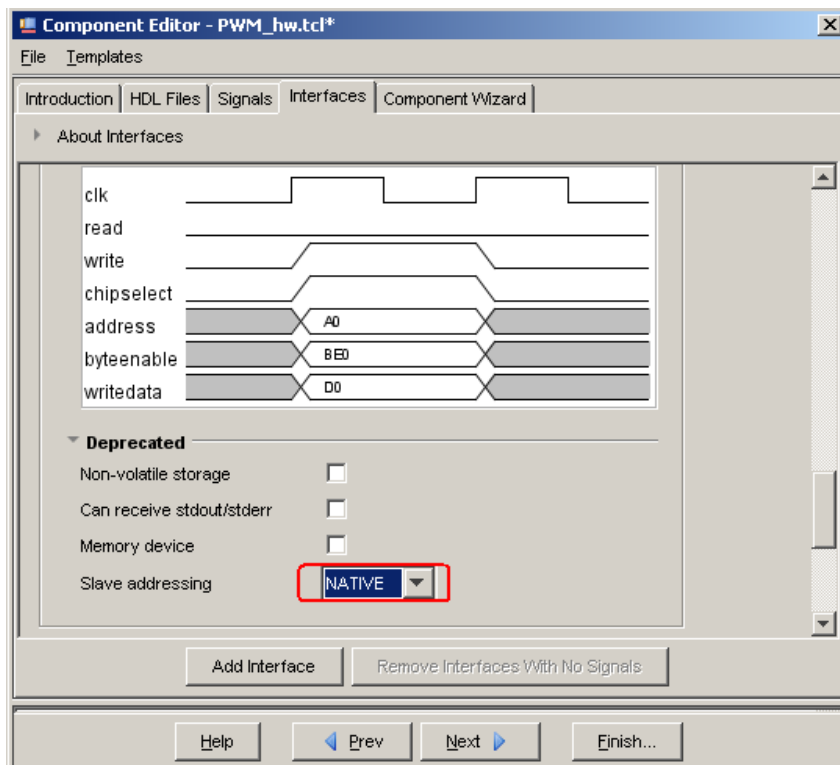
其中，Interface 在下拉菜单中选择下图红圈处所示的选项。



上面的选项都设置好以后，点击 Next，如下图所示，我们通过下图红圈处的下拉条向下拉

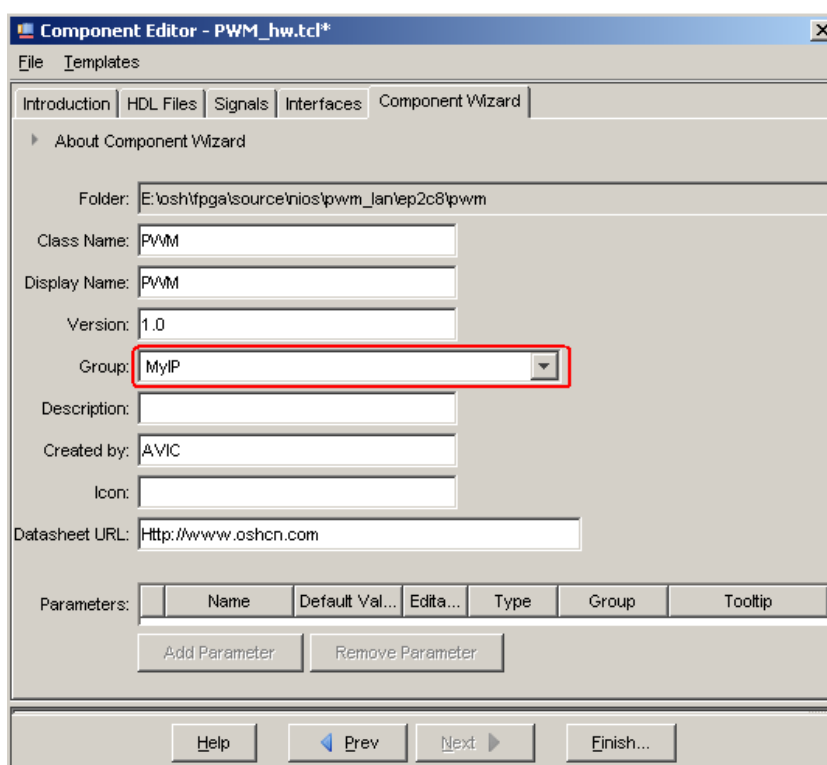


拉到下图所示位置停止，我们将红圈处的改选为 NATIVE，这个地方就是地址对齐的选项，我们选择为静态地址对齐。其他的地方都默认，不需要改动。

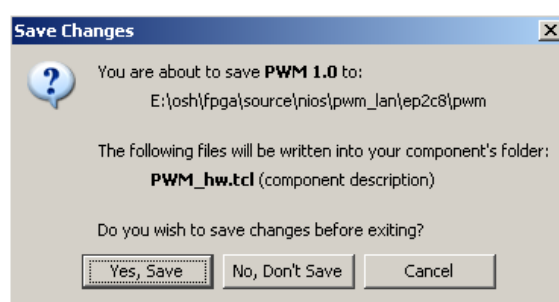


这里面还有很多选项，其中 Timing 部分需要说明一下，PWM 的 Avalon Slave 端口与 Avalon Slave 端口时钟信号同步，读/写时的建立保持时间为 0，因为读、写寄存器仅需要一个时钟周期，所以读/写时为 0 等待切不需要读延时。

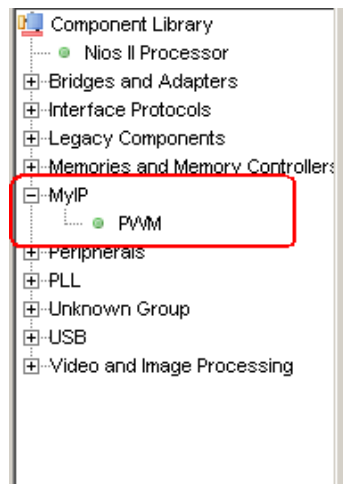
接着点击 Next，如下图所示，其中红圈处需要注意，这个地方需要可以建立新组，然后在 SOPC Builder 中体现出来。



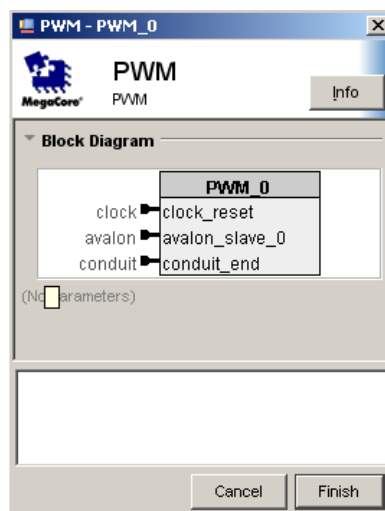
点击 Finish 后，会出现下面的对话框，点击 Yes，就会生成一个 PWM_hw.tcl 脚本文件，大家可以打开看一下，里面放置的是刚才我们配置 PWM 时候的配置信息。



上面都完成以后，我们回到了 SOPC Builder 界面，我们在左侧边栏中可以找到下图所示的红圈处

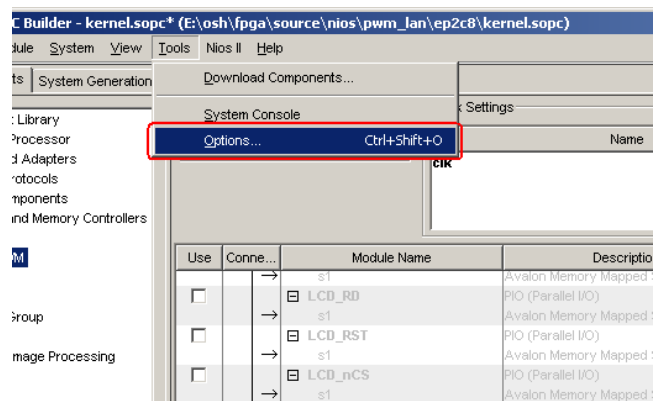


大家看到了吧，MyIP 就是我们刚才建立的 group。双击 PWM，我们建立 PWM 模块，如下图

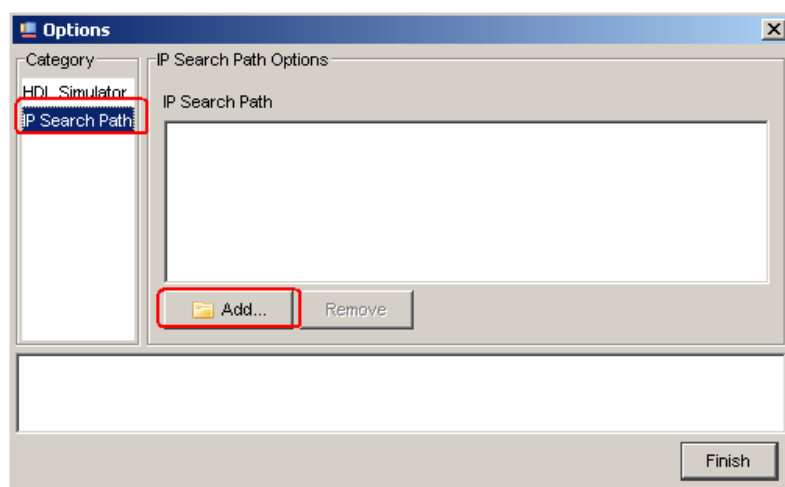


点击 Finish，完成建立。

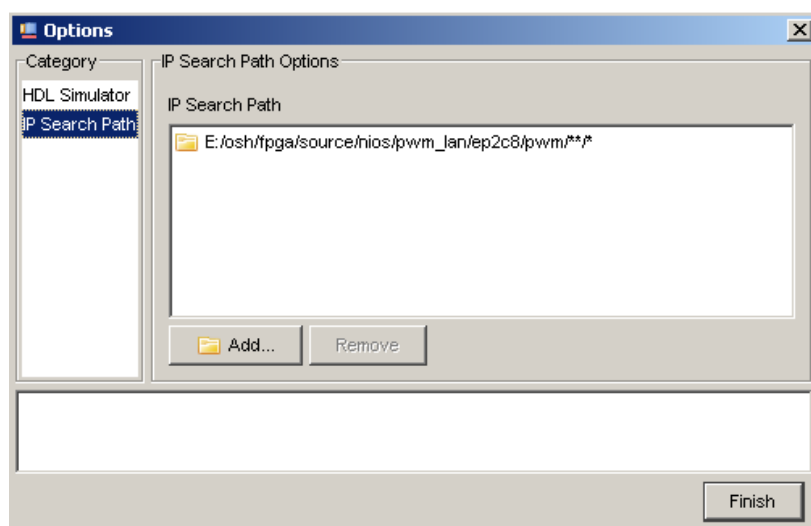
这里还需要设置一步，点击下图红圈处



点击后，如下图所示，点击 IP Search Path，然后点击 Add，添加 PWM.v 所在位置的路径



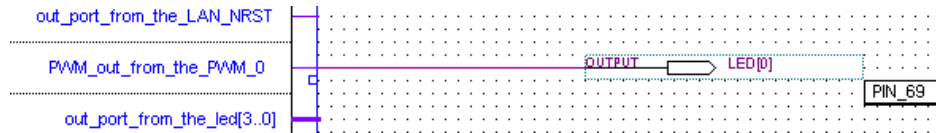
添加后，如下图所示



点击 Finish 完成。设置这个选项是为了让 SOPC Builder 可以找到 PWM.v 的位置。不然就会出现下次你进入 SOPC Builder 的时候 PWM 模块无效的问题。

接下来的工作就是自动分配地址，分配中断，编译，等待.....

编译好以后，我们回到 Quartus 软件界面，我们可以看到，PWM 出现了，我将它接到了一个 LED 上了，我们可以通过 PWM 改变 LED 的亮度，实现 LED 渐亮渐灭的过程。



接下来又是编译，等待.....

四、 软件开发

做好硬件部分工作以后，我们打开 NIOS IDE，开始软件编程部分。

首先对工程重新编译一次，Ctrl+B，等待.....

编译好以后，我们来看一下 system.h 的变化情况，我们可以发现，多出来 PWM 部分了。

```
#define PWM_0_NAME "/dev/PWM_0"
#define PWM_0_TYPE "PWM"
#define PWM_0_BASE 0x00001870
#define PWM_0_SPAN 16
#define PWM_0_TERMINATED_PORTS ""
#define ALT_MODULE_CLASS_PWM_0 PW
```

下面是 PWM 测试代码，

```
#include <unistd.h>
#include "system.h"

//根据寄存器的偏移量，我们定义一个结构体PWM
typedef struct{
    volatile unsigned int divi;
    volatile unsigned int duty;
    volatile unsigned int enable;
}PWM;

int main()
{
    int dir = 1;

    //将pwm指向PWM_0_BASE首地址
    PWM *pwm = (PWM *)PWM_0_BASE;
```

```

//对pwm进行初始化,divi最大值为 $2^{32}-1$ 。
pwm->divi = 1000;
pwm->duty = 0;
pwm->enable = 1;

//通过不断的改变duty值来改变LED一个周期亮灯的时间长短
while(1){
    if(dir > 0){
        if(pwm->duty < pwm->divi)
            pwm->duty += 100;
        else
            dir = 0;
    }
    else{
        if(pwm->duty > 0)
            pwm->duty -= 100;
        else
            dir = 1;
    }

    usleep(100000);
}

return 0;
}

```

到此，这一节的内容就讲完了，大家如果对此有疑问可以给我留言，也可以通过 qq : 984597569，或者 email : avic633@gmail.com 与我联系，谢谢！

第十七章 数码管

数码管

通过这一节，可以了解到动态数码管的操作方式，以及定时器的应用。

本章分为以下几个部分：

- 一、简介
- 二、例程

一、简介

最近写了个小程序 动态数码管扫描 ,算是定时器的应用吧 ,单片机上经常看得到 ,我移植过来了 ,充实一下内容。

首先在 SOPC 中添加一个定时器 ,两个 8 位的 PIO ,一个用于数据线 ,一个是作为开关 ,作为开关的可以根据数码管个数决定 PIO 的数量。

Interval Timer	
Interval Timer	
Avalon Memory Mapped Slave	clk
PIO (Parallel I/O)	
Avalon Memory Mapped Slave	clk
PIO (Parallel I/O)	
Avalon Memory Mapped Slave	clk

二、例程

编译完以后就可以写程序了 ,程序如下所示 ,利用的是动态余辉效果

```
/*
 *=====
 *      Filename:  main.c
 *      Description:
 *      Version:   1.0.0
 *      Created:   2010.6.17
 *      Revision:  none
 *      Compiler:  Nios II 9.0 IDE
 *      Author:    马瑞 (AVIC)
 *      Email:     avic633@gmail.com
 *=====
 */
#include <stdio.h>
#include <sys/unistd.h>
#include <io.h>
#include <string.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_timer_regs.h"
#include "alt_types.h"
#include "sys/alt_irq.h"
```

```

//0~9
unsigned char segtab[10]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90};
unsigned char led_buffer[8]={0};
unsigned char bittab[8]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf,0xbf,0x7f};
static unsigned char cnt=0;

static void timer_init(void);
/*
 * == FUNCTION =====
 *      Name:  main
 *      Description:  主函数
 * =====
 */
int main(void)
{
    unsigned char i=0,j=0;
    unsigned char buf[20];

    timer_init();

    while(1){
        sprintf(buf,"%08u",j++);

        for(i=0;i<8;i++){
            led_buffer[i] = buf[7-i]-'0';
        }
        usleep(500000);
    }

    return 0;
}
/*
 * == FUNCTION =====
 *      Name:  timer_irq
 *      Description:  定时器中断函数
 * =====
 */
static void timer_irq(void *context, alt_u32 id)
{
    IOWR_ALTERA_AVALON_PIO_DATA(SEG_SEL_BASE, 0xff);
    IOWR_ALTERA_AVALON_PIO_DATA(SEG_SEL_BASE, bittab[cnt]);
}

```

```

IOWR_ALTERA_AVALON_PIO_DATA(SEG_DAT_BASE, segtab[led_buffer[cnt]]);

cnt++;

if(cnt==8)
    cnt=0;

IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0);
}
/*
 * === FUNCTION =====
 *      Name:  timer_init
 *  Description:  定时器初始化
 * =====
 */
static void timer_init(void)
{
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0);

    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_BASE,200000);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_BASE,200000 >> 16);

    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0x07);

    alt_irq_register(TIMER_IRQ, NULL, timer_irq);
}

```

内容很简单，也不详细叙述了，说多了大家也嫌烦了，呵呵！

第十八章 USB (一)

USB(一)

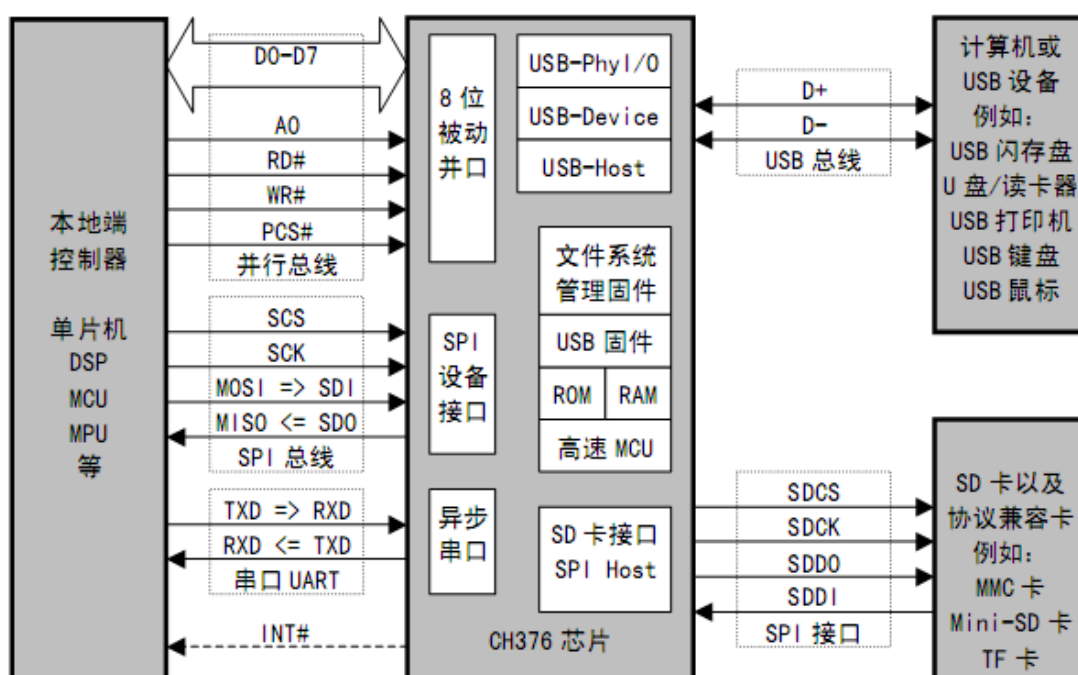
通过本章，您可以了解到 USB 设备模式的操作方式，以及有关上位机的相关内容。

本章分为以下几个部分：

- 一、简介
- 二、硬件开发
- 三、软件开发
- 四、上位机开发

一、简介

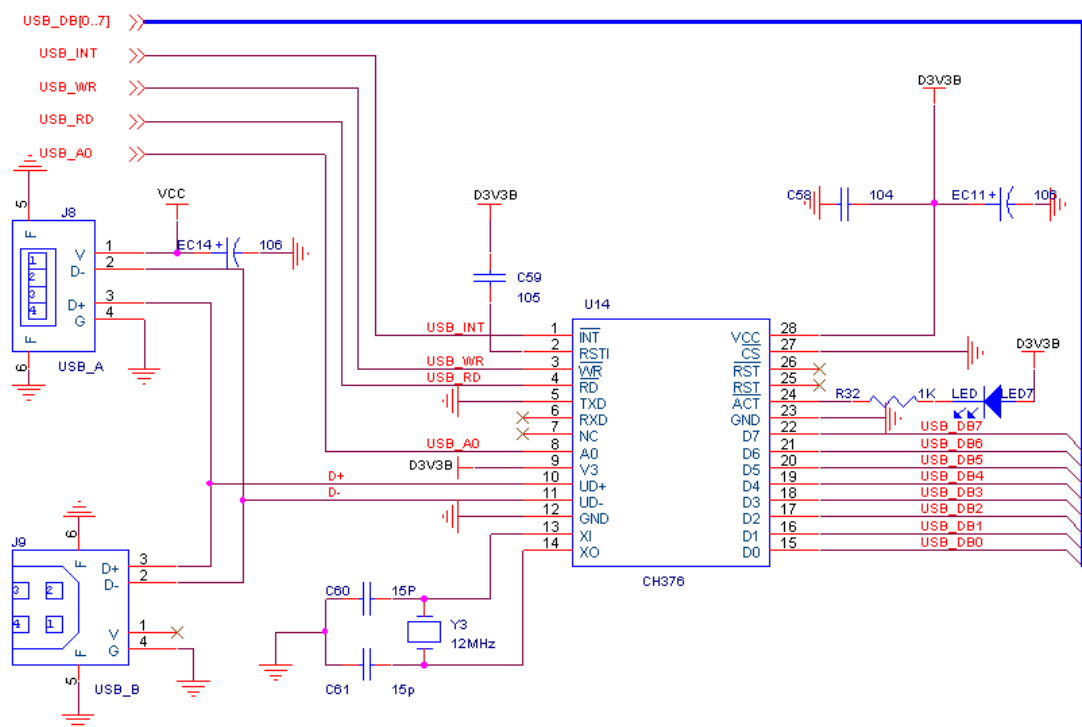
这一节，我们来讲讲黑金开发板 USB 部分的内容。黑金开发板上使用的 USB 芯片是南京沁恒公司的 CH376，它支持 USB 设备 (DEVICE) 方式和 USB(HOST) 主机方式，并且内置了 USB 通讯协议的基本固件，内置了处理 Mass-Storage 海量存储设备的专用通讯协议的固件，内置了 SD 卡的通讯接口固件，内置了 FAT16 和 FAT32 以及 FAT12 文件系统的管理固件，支持常用的 USB 存储设备 (包括 U 盘/USB 硬盘/USB 闪存盘/USB 读卡器) 和 SD 卡 (包括标准容量 SD 卡和高容量 HC-SD 卡以及协议兼容的 MMC 卡和 TF 卡)。



由于芯片内部集成了 USB 通讯协议的基本固件,因此，免去了我们自己编写 USB 通讯协议的麻烦了。不仅如此，它还集成了文件系统的管理固件,那么，我们不就可以直接读取 U 盘中的内容了？事实就是这样的，真的方便了很多哦。下面我们就来看看这款芯片到底有多好用吧。

二、硬件开发

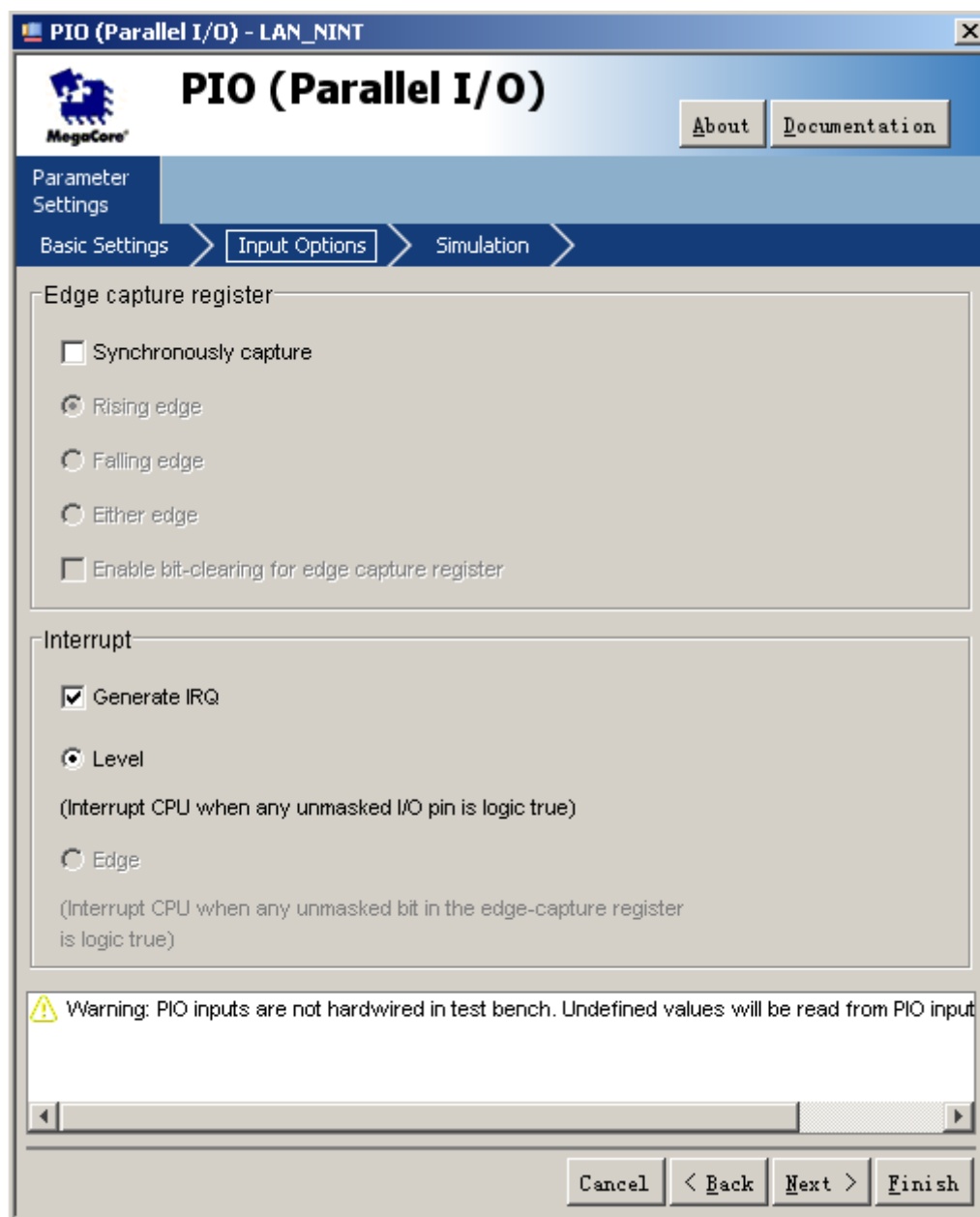
首先，我们看看这部分电路，如下图所示，我们采用的是 8 位总线模式，电路结构非常简单，与 FPGA 相连的一共有 12 根线，其中 8 根数据线，1 根中断线，3 根控制线。



下面，我们就在软核中添加 USB 部分的模块，其实都是通过 PIO 模块控制的。添加后如下图所示，其中 USB_DB 为 8 位输出 PIO；USB_WR，USB_RD，USB_A0 都是 1 位输出 PIO。

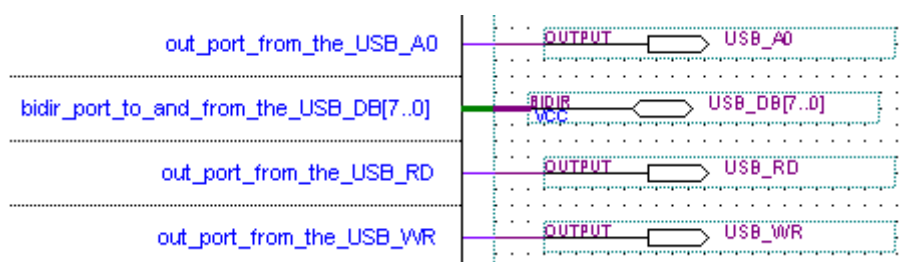
☐ USB_DB	PIO (Parallel I/O)			
s1	Avalon Memory Mapped Slave	clk	0x00001860	0x0000186f
☐ USB_nINT	PIO (Parallel I/O)			
s1	Avalon Memory Mapped Slave	clk	0x00001870	0x0000187f
☐ USB_WR	PIO (Parallel I/O)			
s1	Avalon Memory Mapped Slave	clk	0x00001880	0x0000188f
☐ USB_RD	PIO (Parallel I/O)			
s1	Avalon Memory Mapped Slave	clk	0x00001890	0x0000189f
☐ USB_A0	PIO (Parallel I/O)			
s1	Avalon Memory Mapped Slave	clk	0x000018a0	0x000018af

而 USB_nINT 为输入 PIO，而且电平中断，如下图设置，

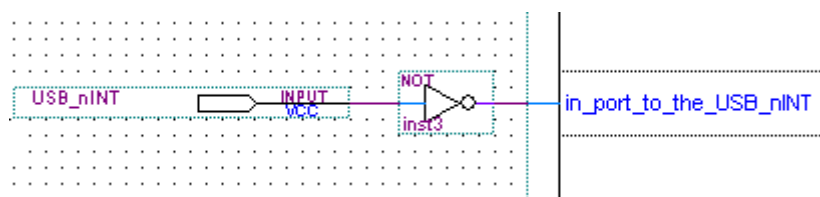


都设置好以后，自动分配地址，中断，接下来就可以编译了。

编译好以后，回到 Quartus 界面，整理好以后，如下图所示，要记得 USB_DB 数据线是双向的，因此，一定要用分配 bidir 双向引脚，而不要用 output。



下面是中断引脚部分，由于我们是低电平中断，而 NIOS 电平中断只支持高电平中断，所以我们需要加一个非门，如下图示所示



都设置好以后，我们就可以分配引脚了，TCL 脚本有关 USB 部分的代码如下

```
#-----USB-----#
set_location_assignment PIN_117 -to USB_DB[0]
set_location_assignment PIN_118 -to USB_DB[1]
set_location_assignment PIN_127 -to USB_DB[2]
set_location_assignment PIN_128 -to USB_DB[3]
set_location_assignment PIN_133 -to USB_DB[4]
set_location_assignment PIN_134 -to USB_DB[5]
set_location_assignment PIN_135 -to USB_DB[6]
set_location_assignment PIN_137 -to USB_DB[7]

set_location_assignment PIN_113 -to USB_A0
set_location_assignment PIN_115 -to USB_WR
set_location_assignment PIN_116 -to USB_nINT
set_location_assignment PIN_114 -to USB_RD
```

分配好引脚以后，大家就可以编译了。

三、 软件开发

USB 分主机模式和设备模式，这两种模式硬件部分是相同的，只是在软件编程方面有些不同。这一节，我们来讲设备模式，也就是开发板通过 USB 接口与主机（电脑）相连，实现开发板与电脑的数据通信。

我们首先打开 NIOS II 9.0 IDE 软件，还是老过程，首先编译一遍，Ctrl+b。编译成功以后，我们在 system.h 中会看到 USB 部分的代码，如下表所示

```
#define USB_DB_NAME "/dev/USB_DB"
#define USB_DB_TYPE "altera_avalon_pio"
#define USB_DB_BASE 0x00001840
.....
```

```

#define USB_NINT_NAME "/dev/USB_nINT"
#define USB_NINT_TYPE "altera_avalon_pio"
#define USB_NINT_BASE 0x00001850
.....
#define USB_WR_NAME "/dev/USB_WR"
#define USB_WR_TYPE "altera_avalon_pio"
#define USB_WR_BASE 0x00001860
.....
#define USB_RD_NAME "/dev/USB_RD"
#define USB_RD_TYPE "altera_avalon_pio"
#define USB_RD_BASE 0x00001870
.....
#define USB_A0_NAME "/dev/USB_A0"
#define USB_A0_TYPE "altera_avalon_pio"
#define USB_A0_BASE 0x00001880
.....

```

下面，我们来添加 USB 部分的代码。

首先，我们建立一个在 inc 下面建立一个 usb.h 文件，内容如下

```

#ifndef __usb_h__
#define __usb_h__
//-----Include files-----//
#include "system.h"
//----- CH375 DEFINE-----//
//下面部分是 USB 寄存器地址，这部分定义可以看 CH376 的芯片手册
#define USB_HOST    0X06
#define USB_DEVICE  0x02
#define USB_DISABLE 0X00

#define RESET_ALL    0X05
#define CHECK_EXIST 0X06
#define SET_USB_ID   0X12
#define SET_USB_MODE 0X15
#define GET_STATUS   0X22
#define UNLOCK_USB   0X23
#define RD_USB_DATA  0X28
#define WR_USB_DATA5  0X2A
#define WR_USB_DATA7  0X2B
#define GET_IC_VER    0X01

```

```

#define ENTER_SLEEP 0X03
#define CHK_SUSPEND 0X0B
#define RD_USB_DATA0    0X27

```

```

#define RET_SUCCESS 0X51
#define RET_ABORT   0X5B

```

```

#define INT_EP2_OUT 0x02
#define INT_EP2_IN  0x0a

```

```

//host

```

```

#define DISK_READ    0X54
#define DISK_RD_GO   0X55
#define DISK_READY   0X59
#define DISK_INIT    0X51

```

```

//status

```

```

#define USB_INT_CONNECT 0x15
#define USB_INT_DISCONNECT 0X16
#define USB_INT_SUCCESS 0X14
#define USB_INT_DISK_READ  0X1D

```

```

//-----bus define-----//

```

//下面是 USB 的接口部分定义，这次我没有像以往那样定义结构体，是为了让大家感受一下各种//形式的编程。大家要注意 PIO_USB_DB_DIR 的定义，通过以前的讲解，不知道大家是否理解，//它是 USB 数据线的方向控制寄存器的定义，知道为什么要+4 么，大家自己考虑吧，不明白就//看看附录中的有关 PIO 问题解析部分内容吧

```

#define PIO_USB_DB      *(volatile unsigned long int *)USB_DB_BASE
#define PIO_USB_WR      *(volatile unsigned long int *)USB_WR_BASE
#define PIO_USB_RD      *(volatile unsigned long int *)USB_RD_BASE
#define PIO_USB_A0      *(volatile unsigned long int *)USB_A0_BASE
#define PIO_USB_INT     *(volatile unsigned long int *)USB_INT_BASE
#define PIO_USB_DB_DIR  *(volatile unsigned long int *) (USB_DB_BASE+4)

```

```

#define VID 0X0FFE
#define PID 0X1000

```

```

typedef struct{
    char receive_buffer[200];
    int send_ok_flag;
    int receive_ok_flag;
}USB_T;

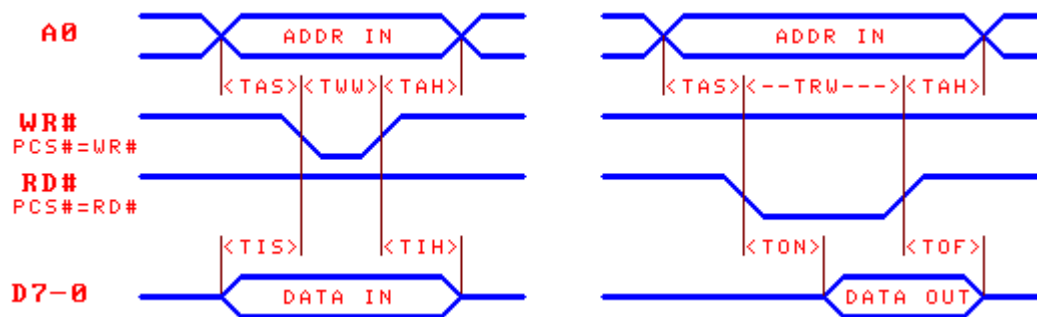
```

```
//-----Extern function-----//

extern USB_T usb;
extern int initialize_usb(void);
extern int set_usb_mode(unsigned char);
extern int send_string_to_usb(char *str,int str_len);
extern void write_command_to_usb(unsigned char command);
extern void write_data_to_usb(unsigned char data);

#endif //__usb_h__
```

接下来，我们看看 CH376 的时序图，如下图所示



接下来，我们就根据上面的时序图编写驱动部分，在 driver 中建立一个 usb.c 文件，内容如下表所示

```
/*
 * =====
 *      Filename:  usb.c
 *      Description:
 *      Version:   1.0.0
 *      Created:   2010.4.16
 *      Revision:  none
 *      Compiler:  Nios II 9.0 IDE
 *      Author:    马瑞 (AVIC)
 *      Email:     avic633@gmail.com
 * =====
 */

//-----Include files-----//
#include "../inc/usb.h"
#include "altera_avalon_pio_regs.h"
#include "sys/alt_irq.h"
#include <unistd.h>
#include <stdio.h>

//-----Function Prototype-----//
```

```

void write_command_to_usb(unsigned char command);
void write_data_to_usb(unsigned char data);
unsigned char read_data_from_usb(void);
void delay(void);
//-----Variable-----//

USB_T usb;
//-----Function-----//

/*
 * == FUNCTION =====
 *      Name:  irq_usb
 * Description:  中断函数
 * =====
 */
void irq_usb(void)
{
    unsigned int i;
    unsigned char interrupt_status,data_len;
    // static int times=0;

    write_command_to_usb(GET_STATUS);

    interrupt_status=read_data_from_usb();

    switch(interrupt_status){
        //Device
        case INT_EP2_OUT:
            write_command_to_usb(RD_USB_DATA);
            data_len=read_data_from_usb();

            for(i=0;i<data_len;i++)
                usb.receive_buffer[i]=read_data_from_usb();

            usb.receive_buffer[i]='\0';
            usb.receive_ok_flag=1;
            break;
        case INT_EP2_IN:
            write_command_to_usb(UNLOCK_USB);
            usb.send_ok_flag=1;
            break;
    }
}

```

```

        default :break;
    }
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(USB_NINT_BASE,0x00);
}

/*
 * ===  FUNCTION  =====
 *      Name:  send_string_to_usb
 *      Description:  发送字符串
 * =====
 */
int send_string_to_usb(char *str,int str_len)
{
    int i;

    write_command_to_usb(WR_USB_DATA7);
    write_data_to_usb(str_len);

    for(i=0;i<str_len;i++)write_data_to_usb(str[i]);

    return 0;
}

/*
 * ===  FUNCTION  =====
 *      Name:  initialize_usb
 *      Description:  初始化USB
 * =====
 */
int initialize_usb(void)
{
    PIO_USB_RD=1;
    PIO_USB_WR=1;
    PIO_USB_A0=1;
    usb.receive_ok_flag=0;

    // enable the io interrupt
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(USB_NINT_BASE,1);
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(USB_NINT_BASE,0);

    alt_irq_register(USB_NINT_IRQ,NULL,irq_usb);
}

```

```

set_usb_mode(USB_DEVICE);

    return 0;
}
/*
 * === FUNCTION =====
 *      Name:  usb_usb_mode
 *      Description:  设置USB模式
 * =====
 */
int set_usb_mode(unsigned char type)
{
    write_command_to_usb(SET_USB_MODE);
    write_data_to_usb(type);
    read_data_from_usb();

    if((read_data_from_usb())==0x51)return 0;
    else return -1;
}
/*
 * === FUNCTION =====
 *      Name:  write_command_to_usb
 *      Description:  写命令
 * =====
 */
void write_command_to_usb(unsigned char command)
{
    //A0
    PIO_USB_A0=1;

    //DB DIR output
    PIO_USB_DB_DIR=0xff;

    PIO_USB_DB=command;

    PIO_USB_WR=0;
    PIO_USB_WR=1;
}
/*
 *      === FUNCTION

```

```

=====
*          Name: delay
* Description: 延时
*
=====

=====
*/
void delay(void)
{
    int i;
    for(i=0;i<1000;i++);
}
/*
* === FUNCTION =====
*          Name: delay
* Description: 延时
* =====
*/
void write_data_to_usb(unsigned char data)
{
    //A0
    PIO_USB_A0=0;

    //DB DIR output
    PIO_USB_DB_DIR=0xff;

    PIO_USB_DB=data;

    usleep(20);
    PIO_USB_WR=0;
    delay();
    usleep(20);
    PIO_USB_WR=1
}
/*
* === FUNCTION =====
*          Name: write_data_to_usb
* Description: 写数据
* =====
*/

```

```

unsigned char read_data_from_usb(void)
{
    unsigned char data=0;

    //A0
    PIO_USB_A0=0;

    //DB DIR output
    PIO_USB_DB_DIR=0;

    PIO_USB_RD=0;
    delay();
    data=PIO_USB_DB;
    PIO_USB_RD=1;

    return data;
}

```

编写好驱动以后，我们需要编写主函数测试代码

```

#include <stdio.h>
#include <unistd.h>
#include "../inc/usb.h"
int main()
{
    unsigned char tmp[] = "Hello USB!\n";

    initialize_usb();

    while(1){
        if(usb.receive_ok_flag){
            printf("%s\n",usb.receive_buffer);
            usb.receive_ok_flag = 0;
        }

        send_string_to_usb(tmp,sizeof(tmp));
        usleep(100000);
    }
    return 0;
}

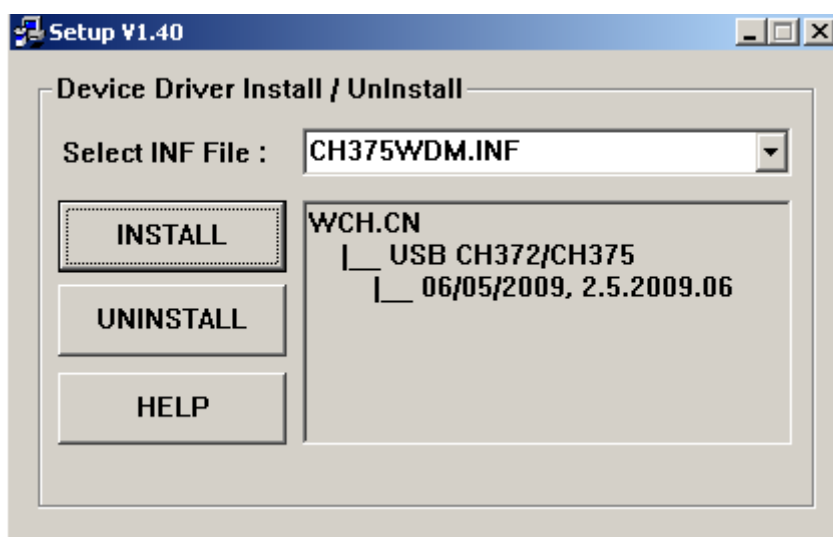
```

四、 上位机开发

程序都写完了，但工作还没有结束，如果要想调试，我们首先还需要在你的电脑上安装 CH376 的驱动。

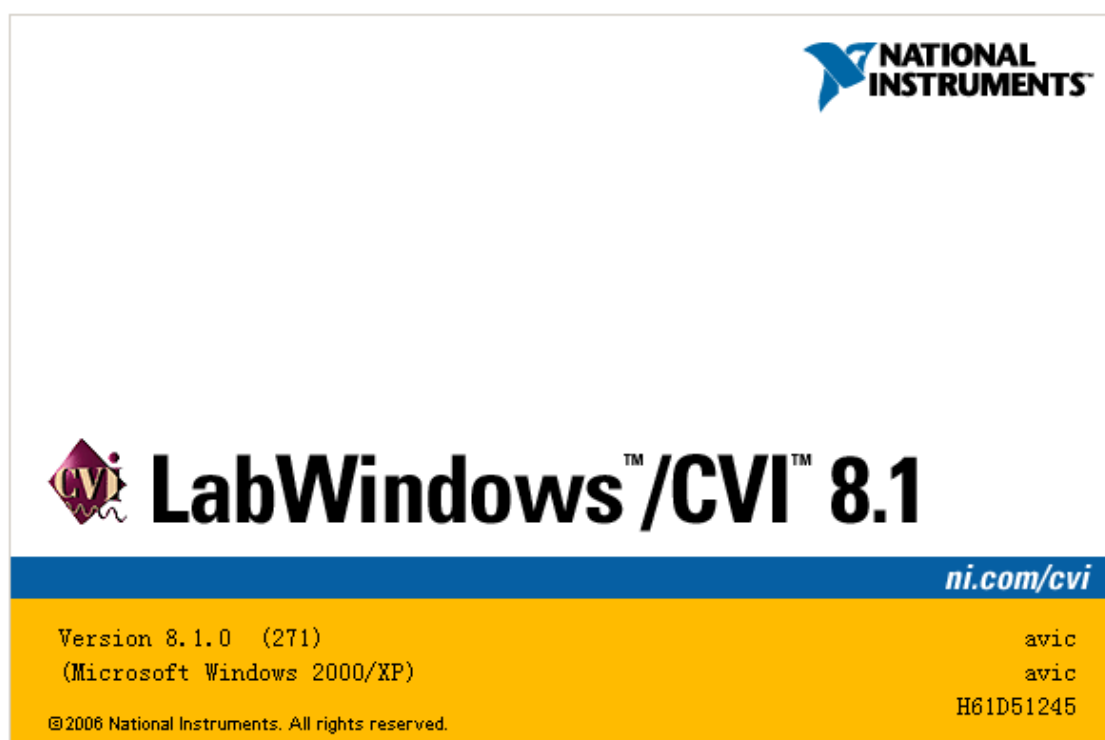
首先，去南京沁恒的网站下载驱动，下载地址是：
<http://www.wch.cn/download/list.asp?id=66>，CH376 的驱动跟 CH372,CH375 是一样的。

双击 CH372DRV.EXE，开始安装驱动，如下图所示，点击 INSTALL，直接安装就可以了。

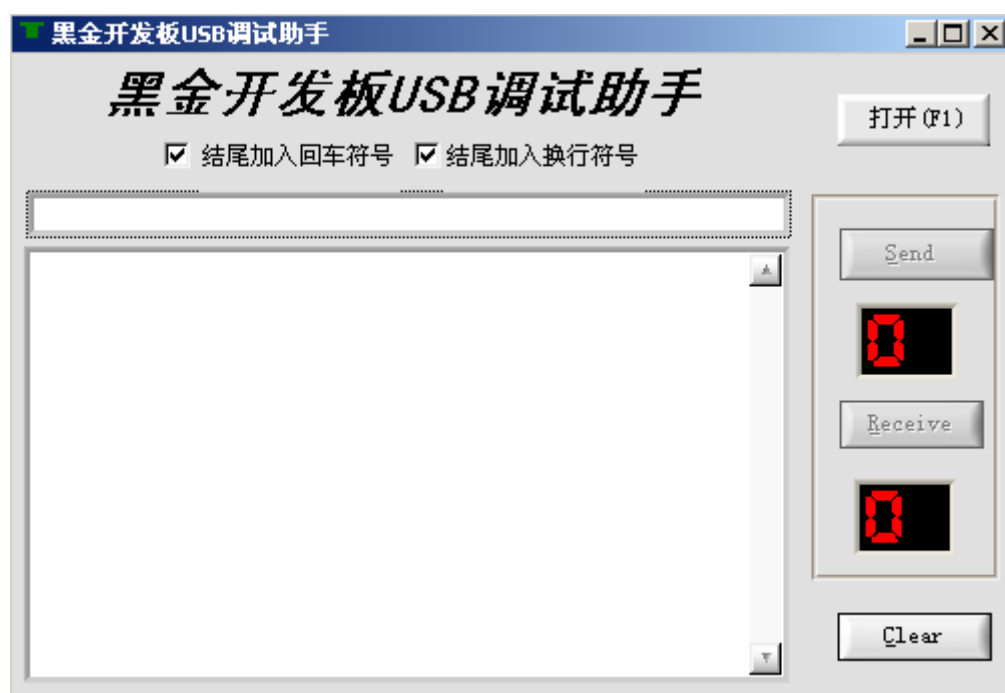


为了调试，我们还需要上位机的软件来配合，就像串口调试精灵的一个东西。这部分工作属于上位机部分的内容了。我在这里简单介绍一下吧。

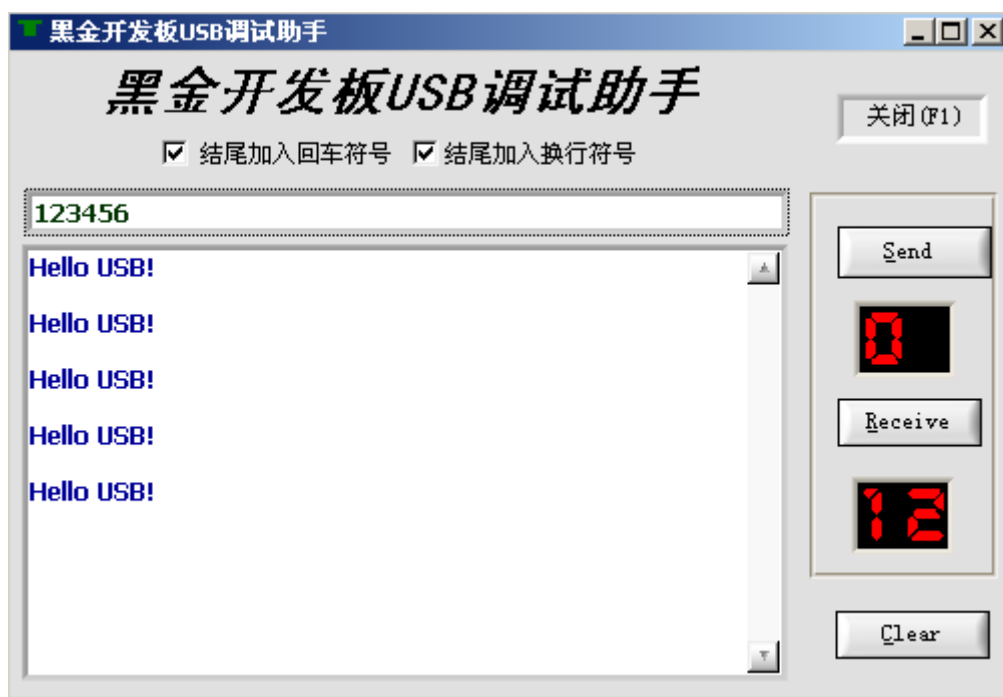
南京沁恒网站提供了上位机需要的静态库函数和头文件，下载地址是：
<http://www.wch.cn/download/list.asp?id=28>，我们可以利用他们构建自己的上位机。我使用的是 NI 公司的 Labwindows/CVI 8.1，当然大家也可以使用 VC 等软件开发。



我感觉这个软件还是蛮好用的,大家可以研究一下。写好的上位机面板如下图所示,



我们可以利用它进行简单的发送和接收,软件还不够完善。下面简单介绍一下使用方法,首先需要将 FPGA 运行起来,然后点击上位机的打开按钮。如果是接收的话,点击 Receive,每点一次接收一次。如果发送的话,将你发送的数据写到上面的输入框中,点击 Send,每点一次发送一次。如下图示



好了，到这里，有关 USB 的设备模式的内容就讲完了。下一节，我们将讲解有关 USB 主模式的内容，也就是如何读取 U 盘等相关内容。谢谢大家！

第十九章 USB (二)

USB(二)

通过本章，您可以了解到 USB 设备模式的操作方式，以及有关上位机的相关内容。

本章分为以下几个部分：

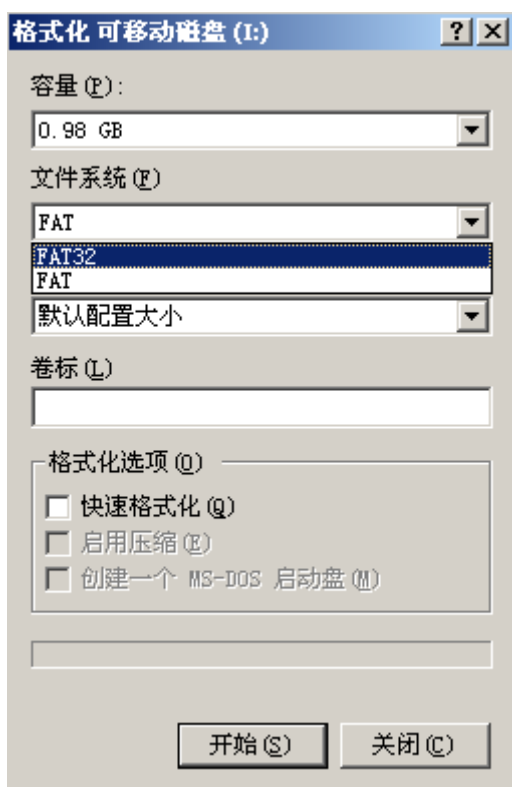
- 一、简介
- 二、硬件开发
- 三、软件开发
- 四、上位机开发

一、简介

上一节，我们讲了 USB 的设备模式，可以实现计算机与黑金开发板的数据通信。这一节，我们以上一节为基础，研究一下如何利用 CH376 芯片来实现 U 盘的读写。

大家都知道，不管是 U 盘，SD 卡，还是我们经常用到的电脑硬盘，都存在文件的建立，存取，修改等操作。从系统角度来说，负责管理这些工作的我们称之为文件管理系统，简称文件系统。经常用电脑的人对这个概念一定很熟悉，比如我们在装系统的时候，不可避免的会遇到选择何种文件系统(除非你用 GHOST ,呵呵) ,FAT32 还是 NTFS 等等，这个就是文件系统。文件系统是对文件存储器空间进行组织和分配，负责文件的存储并对存入的文件进行保护和检索的系统，具体地说，他负责为用户建立文件、存入、读出、修改、转储文件爱你，控制文件的存取，当用户不再使用时撤销文件等。

在 U 盘中，同样存在文件系统。大家熟知的，当我们想格式化 U 盘的时候，我们也需要选择文件系统类型，如下图所示



因此，如果要想通过 U 盘来实现开发板与电脑之间的数据交换，那么，在开发板上也也应该在 FAT 规范下通过文件的形式存取 U 盘中的数据。不过，CH376 已经把这个工作为我们做好了。

CH376 在芯片内部集成了 FAT 文件系统，支持 FAT32、FAT16 和 FAT12，

符合 WINDOWS 的文件系统格式。这种无微不至的“关怀”实在令我们感动啊，因为这样我们就不需要自己动手编写了文件系统了，为我们减少了很多的工作量，剩下的时间看看世界杯还是很不错哦，呵呵。

CH376 对 U 盘文件的读写方式分为两种：扇区模式和字节模式。

扇区模式下，以扇区（每扇区通常是 512 字节）为基本单位对 U 盘文件进行读写，所以读写速度略快，但是通常情况下需要额外的文件数据缓冲区，额外的文件数据缓冲区必须是扇区长度 512 的整数倍，所以适用于 RAM 多、数据量大、频繁读写数据的单片机系统。扇区读写的子程序主要有扇区读 CH376SecRead 和扇区写 CH376SecWrite。

字节模式下，以字节为基本单位对 U 盘文件进行读写，少则 1 字节，多则 65535 字节，读写速度略慢，但是不需要额外的文件数据缓冲区，使用方便，适用于 RAM 少（从几字节到几十 K 都可以）、数据量小或者数据零碎、不经常读写数据的单片机系统。但是，因为闪存只能进行有限次擦写，如果频繁地向 U 盘写入零碎的数据，可能会缩短 U 盘中闪存的使用寿命。

二、 软件设计

下面，我们就通过我们黑金开发板来实践一下，看如何使用 CH376 来实现 U 盘的读取的。

在南京沁恒的官方网站上，为我们提供了 CH376 相关的例程，不过都是基于单片机的，我们需要移植到 NIOS II 下实现。

首先，我们需要自己来写 USB 的底层驱动，这个跟上一节的 USB 驱动稍有不同。我们在 driver 下建立 hal.c 文件，如下所示

```
/*
 * =====
 *      Filename:  hal.c
 *      Description: ch376底层驱动
 *      Version:    1.0.0
 *      Created:    2010.4.16
 *      Revision:   none
 *      Compiler:   Nios II 9.0 IDE
 *
 *      Author:     马瑞 (AVIC)
 *      Email:      avic633@gmail.com
 */
```

```

*=====
*/
//-----Include files-----//
#include "../inc/hal.h"
#include "altera_avalon_pio_regs.h"
#include "sys/alt_irq.h"
#include <unistd.h>
#include <stdio.h>
#include "../inc/ch376inc.h"
#include "system.h"

//-----Function Prototype-----//
void xWriteCH376Cmd(unsigned char command);
void xWriteCH376Data(unsigned char data);
unsigned char xReadCH376Data(void);
unsigned int mInitCH376Host(void);
void delay(void);
int set_usb_mode(unsigned char type);
unsigned char Query376Interrupt(void);
void irq_usb(void * context, unsigned int id);
void mDelayMS(int ms);
//-----Variable-----//
USB_T usb;
/*
* ===  FUNCTION  =====
*      Name:  set_usb_mode
*      Description:  设置工作模式
* =====
*/
int set_usb_mode(unsigned char type)
{
    xWriteCH376Cmd(SET_USB_MODE);
    xWriteCH376Data(type);
    xReadCH376Data();

    if((xReadCH376Data()) == CMD_RET_SUCCESS)
        return 0;
    else
        return -1;
}

```

```

/*
 * ===  FUNCTION  =====
 *      Name: xWriteCH376Cmd
 *      Description: 写命令
 * =====
 */
void xWriteCH376Cmd(unsigned char command)
{
    //A0
    PIO_USB_A0=1;
    //DB DIR output
    PIO_USB_DB_DIR=0xff;

    PIO_USB_DB=command;

    PIO_USB_WR=0;
    PIO_USB_WR=1;
}

/*
 * ===  FUNCTION  =====
 *      Name: delay
 *      Description: 延时
 * =====
 */
void delay(void)
{
    int i;
    for(i=0;i<1000;i++);
}

/*
 * ===  FUNCTION  =====
 *      Name: xWriteCH376Data
 *      Description: 写数据
 * =====
 */
void xWriteCH376Data(unsigned char data)
{
    //A0
    PIO_USB_A0=0;

```

```

//DB DIR output
PIO_USB_DB_DIR=0xff;

PIO_USB_DB=data;
usleep(20);
PIO_USB_WR=0;
delay();
usleep(20);
PIO_USB_WR=1;
}
/*
 * ===  FUNCTION  =====
 *      Name:  xReadCH376Data
 *      Description:  读数据
 * =====
 */

unsigned char xReadCH376Data(void)
{
    unsigned char data=0;
    //A0
    PIO_USB_A0=0;

    //DB DIR output
    PIO_USB_DB_DIR=0;

    PIO_USB_RD=0;
    delay();
    data=PIO_USB_DB;
    PIO_USB_RD=1;

    return data;
}
/*
 * ===  FUNCTION  =====
 *      Name:  mInitCH376Host
 *      Description:  初始化
 * =====
 */
unsigned int mInitCH376Host(void)

```

```

{
    PIO_USB_RD=1;
    PIO_USB_WR=1;
    PIO_USB_A0=1;
    usb.receive_ok_flag=0;
    usb.send_ok_flag=0;
    //enable the io interrupt

    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(USB_NINT_BASE,0);
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(USB_NINT_BASE,0);

    set_usb_mode(USB_HOST);

    return USB_INT_SUCCESS;
}
/*
 * === FUNCTION =====
 *      Name: Query376Interrupt
 *      Description: 读取中断引脚
 * =====
 */
unsigned char Query376Interrupt( void )
{
    return(PIO_USB_NINT?TRUE:FALSE);
}
/*
 * === FUNCTION =====
 *      Name: mDelaymS
 *      Description: 延时
 * =====
 */
void mDelaymS(int ms)
{
    usleep(1000*ms);
}

```

接下来，我们需要在 inc 下建立 hal.h 文件

```

/*
 * =====
 *
 *      Filename:  hal.c
 *      Description:  ch376底层驱动
 *      Version:    1.0.0
 *      Created:    2010.4.16
 *      Revision:   none
 *      Compiler:   Nios II 9.0 IDE
 *      Author:    马瑞 (AVIC)
 *      Email:     avic633@gmail.com
 *
 * =====
 */

#ifndef __usb_h__
#define __usb_h__

//-----Include files-----//
#include "system.h"

//-----Data struct-----//

//----- CH375 DEFINE-----//

//common
#define USB_HOST      0X06
#define USB_DEVICE    0x02
#define USB_DISABLE   0X00

#define RESET_ALL     0X05
#define CHECK_EXIST   0X06
#define SET_USB_ID    0X12
#define SET_USB_MODE   0X15
#define GET_STATUS    0X22
#define UNLOCK_USB    0X23
#define RD_USB_DATA   0X28
#define WR_USB_DATA5   0X2A
#define WR_USB_DATA7   0X2B
#define GET_IC_VER    0X01
#define ENTER_SLEEP   0X03
#define CHK_SUSPEND   0X0B
#define RD_USB_DATA0   0X27

```

```

#define RET_SUCCESS 0X51
#define RET_ABORT 0X5B

#define INT_EP2_OUT 0x02
#define INT_EP2_IN 0x0a

//host
#define DISK_READ 0X54
#define DISK_RD_GO 0X55

#define DISK_READY 0X59
#define DISK_INIT 0X51

//status

#define USB_INT_CONNECT 0x15

//-----USB HOST-----//
#define CMD_RET_SUCCESS 0x51 /* 命令操作成功 */
#define CMD_RET_ABORT 0x5F /* 命令操作失败 */

//-----bus define-----//
//usb
#define PIO_USB_DB *(volatile unsigned long int *)USB_DB_BASE
#define PIO_USB_WR *(volatile unsigned long int *)USB_WR_BASE
#define PIO_USB_RD *(volatile unsigned long int *)USB_RD_BASE
#define PIO_USB_A0 *(volatile unsigned long int *)USB_A0_BASE
#define PIO_USB_NINT *(volatile unsigned long int *)USB_NINT_BASE

#define PIO_USB_DB_DIR *(volatile unsigned long int *) (USB_DB_BASE+4)

#define VID 0X0FFE
#define PID 0X1000

typedef struct{
    char receive_buffer[200];
    int send_ok_flag;
    int receive_ok_flag;
}USB_T;

//-----Extern function-----//

extern USB_T usb;
extern void xWriteCH376Cmd(unsigned char command);

```

```

extern void xWriteCH376Data(unsigned char data);
extern unsigned char xReadCH376Data(void);
extern unsigned int mInitCH376Host(void);
extern unsigned char Query376Interrupt(void);
extern void mDelaymS(int ms);

#endif // __usb_h__

```

上面属于底层驱动部分，如果想实现 U 盘的读取，还需要处理文件系统子程序 file_sys.c，官方已有提供，我们将其加入到 drvier 下。将头文件 file_sys.h 放到 inc 下。

将上述内容都设置好以后，我们需要编写一个测试程序。官方提供了很多的例程，大家可以讲我们编写的底层驱动替换下自带的就可以。

下面程序演示字节读写，文件枚举，用于将 U 盘中的/C51/CH376HFT.C 文件中的前 200 个字符显示出来，如果找不到原文件 CH376HFT.C，那么该程序将显示 C51 子目录下所有以 CH376 开头的文件名，如果找不到 C51 子目录，那么该程序将显示根目录下的所有文件名。

```

/*
 * =====
 *
 *      Filename:  hal.c
 *      Description:  ch376底层驱动
 *      Version:  1.0.0
 *      Created:  2010.4.16
 *      Revision:  none
 *      Compiler:  Nios II 9.0 IDE
 *      Author:  马瑞 (AVIC)
 *      Email:  avic633@gmail.com
 *
 * =====
 */
#include "../inc/ch376inc.h"
#include "../inc/hal.h"
#include "../inc/file_sys.h"
#include <unistd.h>
#include <string.h>
#include <stdio.h>

```

```

UINT8 buf[64];

void host(void) {
    UINT8    i, s;
    UINT8    TotalCount;
    UINT16   RealCount;
    P_FAT_DIR_INFO  pDir;

    s = mInitCH376Host( ); /* 初始化CH376 */

    while ( 1 ) {
        printf( "Wait Udisk/SD\n" );
        /* 检查U盘是否连接,等待U盘插入 */
        while (CH376DiskConnect() != USB_INT_SUCCESS) {
            usleep( 1000*100 ); /* 没必要频繁查询 */
            printf("USB FAILURE\n");
        }

        for ( i = 0; i < 100; i ++ ) { /* 最长等待时间,100*50mS */
            usleep( 1000*50 );
            printf( "Ready ?\n" );
            s = CH376DiskMount( ); /* 初始化磁盘并测试磁盘是否就绪 */
            if ( s == USB_INT_SUCCESS ) break; /* 准备好 */
            else if ( s == ERR_DISK_DISCON ) break; /* 检测到断开,重新检测并计时 */
            if ( CH376GetDiskStatus( ) >= DEF_DISK_MOUNTED && i >= 5 ) break; /*
有的U盘总是返回未准备好,不过可以忽略,只要其建立连接MOUNTED且尝试5*50mS */
        }
        if ( s == ERR_DISK_DISCON ) { /* 检测到断开,重新检测并计时 */
            printf( "Device gone\n" );
            continue;
        }
        if ( CH376GetDiskStatus( ) < DEF_DISK_MOUNTED ) { /* 未知USB设备,例如USB
键盘、打印机等 */
            printf( "Unknown device\n" );
            goto UnknownUsbDevice;
        }
        i = CH376ReadBlock( buf ); /* 如果需要,可以读取数据块
CH376_CMD_DATA.DiskMountInq,返回长度 */
        if ( i == sizeof( INQUIRY_DATA ) ) { /* U盘的厂商和产品信息 */
            buf[ i ] = 0;

```

```

        printf( "UdiskInfo: %s\n", ((P_INQUIRY_DATA)buf) -> VendorIdStr );
    }

/* 读取原文件 */
    printf( "Open\n" );
    strcpy( buf, "\\C51\\CH376HFT.C" ); /* 源文件名,多级目录下的文件名和路径名必须复制到RAM中再处理,而根目录或者当前目录下的文件名可以在RAM或者ROM中 */
    printf("buf:%s\n",buf);
    s = CH376FileOpenPath( buf ); /* 打开文件,该文件在C51子目录下 */
    if ( s == ERR_MISS_DIR || s == ERR_MISS_FILE ) { /* 没有找到目录或者没有找到文件 */
/* 列出文件,完整枚举可以参考EXAM13全盘枚举 */
        if ( s == ERR_MISS_DIR ) strcpy( buf, "\\*" ); /* C51子目录不存在则列出根目录下的文件 */
        else strcpy( buf, "\\C51\\CH376*" ); /* CH376HFT.C文件不存在则列出\C51子目录下的以CH376开头的文件 */
        printf( "List file %s\n", buf );
        s = CH376FileOpenPath( buf ); /* 枚举多级目录下的文件或者目录,输入缓冲区必须在RAM中 */
        while ( s == USB_INT_DISK_READ ) { /* 枚举到匹配的文件 */
            CH376ReadBlock( buf ); /* 读取枚举到的文件的FAT_DIR_INFO结构,返回长度总是sizeof( FAT_DIR_INFO ) */
            pDir = (P_FAT_DIR_INFO)buf; /* 当前文件目录信息 */
            if ( pDir -> DIR_Name[0] != '.' ) { /* 不是本级或者上级目录名则继续,否则必须丢弃不处理 */
                if ( pDir -> DIR_Name[0] == 0x05 ) pDir -> DIR_Name[0] = 0xE5;
/* 特殊字符替换 */
                pDir -> DIR_Attr = 0; /* 强制文件名字符串结束以便打印输出 */
                printf( "*** EnumName: %s\n", pDir -> DIR_Name ); /* 打印名称,原始8+3格式,未整理成含小数点分隔符 */
            }
            xWriteCH376Cmd( CMD0H_FILE_ENUM_GO ); /* 继续枚举文件和目录 */
/*
            s = Wait376Interrupt( );
        }
    }
    else { /* 找到文件或者出错 */

        TotalCount = 200; /* 准备读取总长度 */
        printf( "从文件中读出的前%d个字符是:\n", (UINT16)TotalCount );

```

```

        while ( TotalCount ) { /* 如果文件比较大,一次读不完,可以再调用
CH376ByteRead继续读取,文件指针自动向后移动 */
            if ( TotalCount > sizeof(buf) ) i = sizeof(buf); /* 剩余数据较多,限制单
次读写的长度不能超过缓冲区大小 */
            else i = TotalCount; /* 最后剩余的字节数 */

            s = CH376ByteRead( buf, i, &RealCount ); /* 以字节为单位读取数据块,
单次读写的长度不能超过缓冲区大小,第二次调用时接着刚才的向后读 */

            TotalCount -= (UINT8)RealCount; /* 计数,减去当前实际已经读出的字符
数 */

            for ( s=0; s!=RealCount; s++ ) printf( "%C", buf[s] ); /* 显示读出的字符
*/
            //          printf("%s",buf);
            if ( RealCount < i ) { /* 实际读出的字符数少于要求读出的字符数,说明已
经到文件的结尾 */
                printf( "\n" );
                printf( "文件已经结束\n" );
                break;
            }
        }
        printf( "Close\n" );
        s = CH376FileClose( FALSE ); /* 关闭文件 */
    }

UnknownUsbDevice:
    printf( "Take out\n" );
    while ( CH376DiskConnect() == USB_INT_SUCCESS ) { /* 检查U盘是否连接,等
待U盘拔出 */
        usleep( 1000*100 );
    }
    usleep( 1000*100 );
}
}

```

好了，这节内容就到此结束了。有关 FAT 文件系统部分内容，建议大家自己搜索相关资料，对它进一步了解，对程序的理解很有好处。谢谢大家！

第二十章 附录

附录

附录中包含了 NIOS II FAQ，还有一些零散的问题解决方式。

本章分为以下几个部分：

三、NIOS II 下关于无法进行寄存器方式操作 PIO 的问题解析

四、对寄存器结构体的详细解析

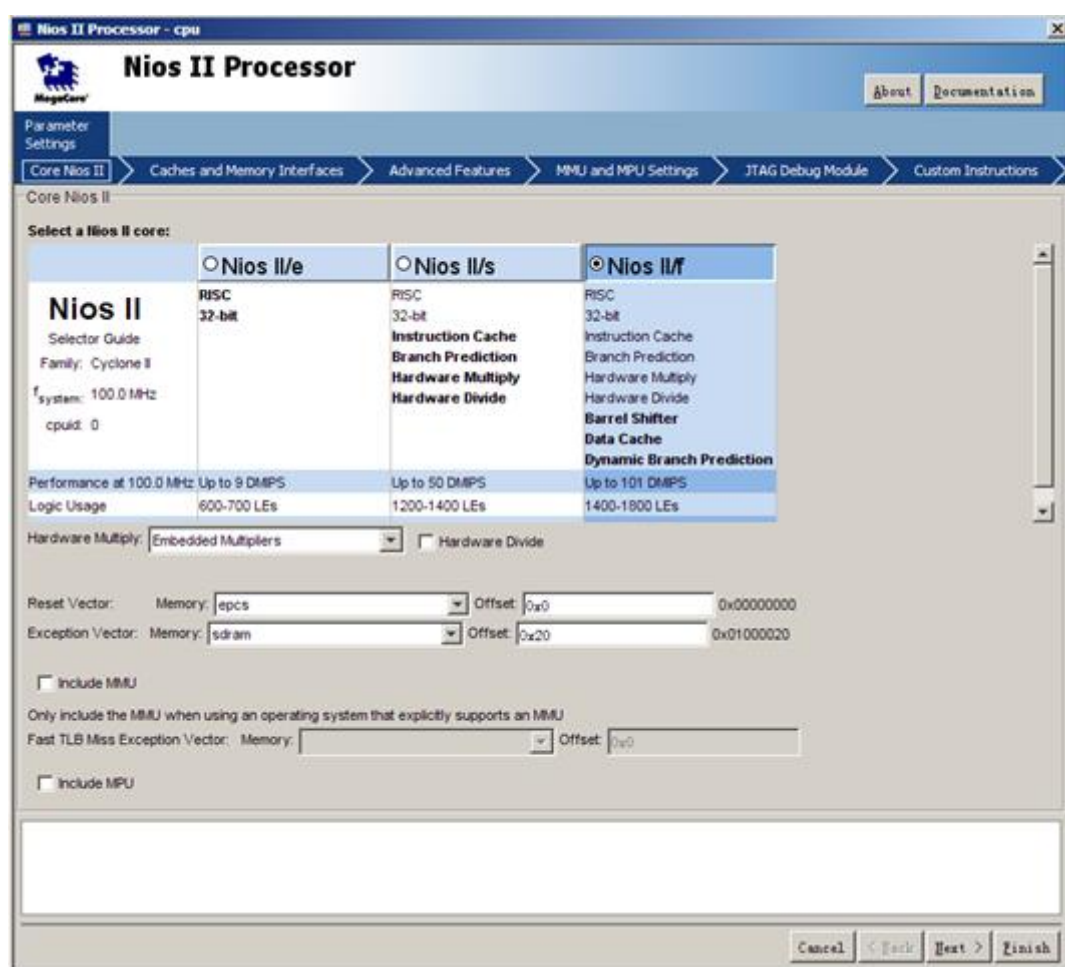
五、NIOS II 常见问题解答（FAQ）

六、黑金开发板印制板

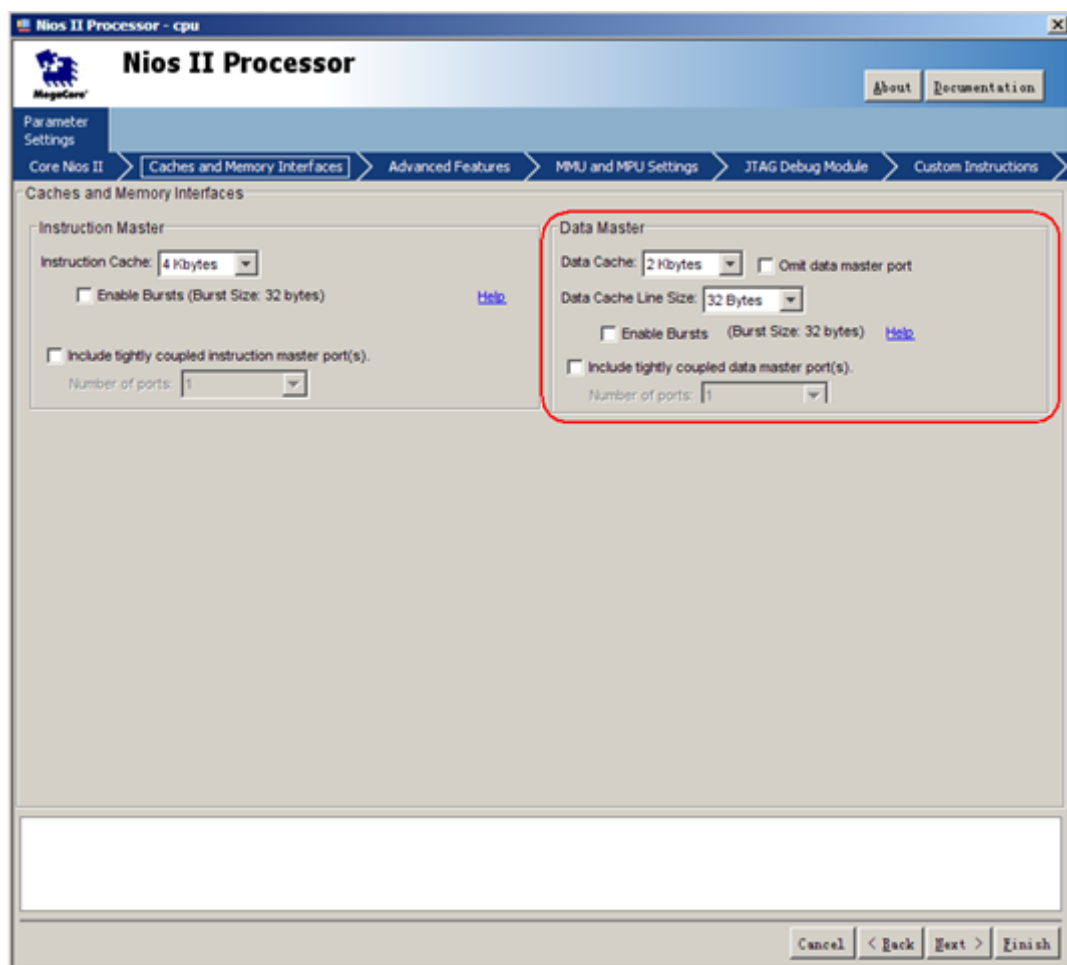
一、 NIOS II 下关于无法进行寄存器方式操作 PIO 的问题解析

最近得到大家的反馈 ,说用寄存器直接映射的方法不好用 ,点亮一个 LED 都不行。今天调试我的改版核心版 ,竟然也遇到了这样的情况。于是研究了一下 ,发现了问题的根源。在此给大家解释一下。

我的改版核心板用的是 EP2C8 ,资源比我以前的 EP2C5 要多一些。所以 ,我今天我用了 NIOS II/f ,这个软核资源占用的最多的 ,但速度也是最快的 ,如下图所示



这步没有任何问题 ,关键是下一步 ,下图的红圈处 ,在 NIOS II/f 中 ,DATA CACHE (数据高速缓存)被打开了 ,在其他两种 CPU 中这个地方时灰的不能用的 ,就是说只有 NIOS II/f 数据高级缓存才打开的。看来就是数据高速缓存惹的祸。



那 Data Cache 到底是什么呢，芯片手册上是这样介绍的，（引自《n2cpu_nii5v1.pdf》的 2-12 页）

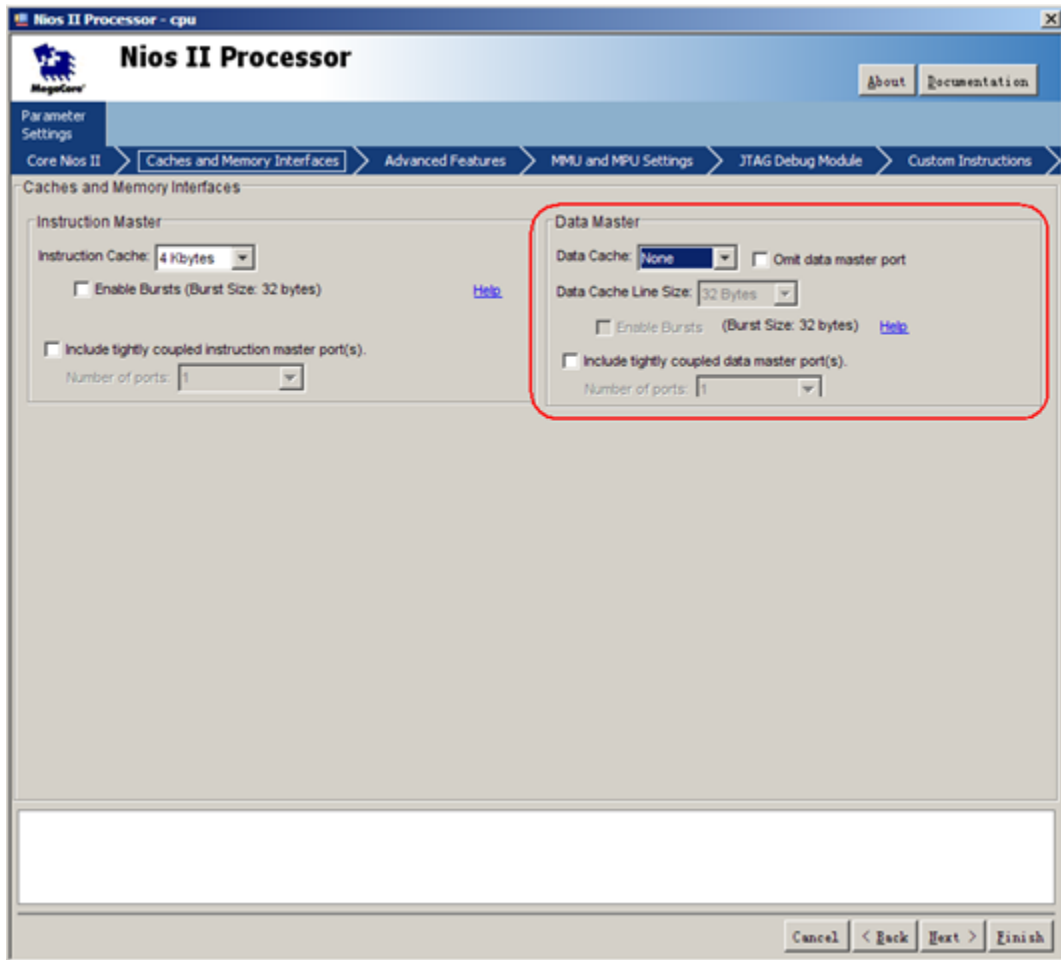
Cache memory resides on-chip as an integral part of the Nios II processor core. The cache memories can improve the average memory access time for Nios II processor systems that use slow off-chip memory such as SDRAM for program and data storage.

简单的说，Data Cache 是像电脑的一级缓存二级缓存一样的东东，就是为了提高系统速度的。可是有个这个高速缓冲区后，我们的代码执行时间却变得不可确定了，降低了程序的实时性。我们发出的数据，放到了高速缓冲区里而没有及时的去执行才导致代码不执行，没有显示效果。想要了解 Cache memory 相关内容的可以参考 http://www.altera.com/literature/hb/nios2/n2sw_nii52007.pdf。

既然知道原因了，那我们该如何解决呢。方法有两个：

第一种就是在建立 NIOS II/f 的时候，将 Data Cache 设置为 0，就是说关闭 Data Cache。如下图所示，这样就不存在 Dache Cache 了，这个办法彻底吧，这就叫斩草

除根，呵呵！



第二种是在开启了 Data Cache 前提下，通过软件来解决。
http://www.altera.com/literature/hb/nios2/n2sw_nii52007.pdf 这里面有所介绍，就是通过 31-bit Cache bypass。什么意思呢，很简单，NIO S II 将寄存器的第 31 位作为了 Cache 开启与否的控制位，如果此位为 1，则 Cache 关闭不启用，否则就开启。一般情况我们不会注意到这个的，所以才会出现无法控制 LED 的情况。我们把最简单的程序修改一下，如下所示

```
#include <unistd.h>
#include "system.h"

//这个地方要注意了，多了 (1<<31)，这就是bypass Cache
#define LED *(volatile unsigned long *) (LED_BASE | (1 << 31))

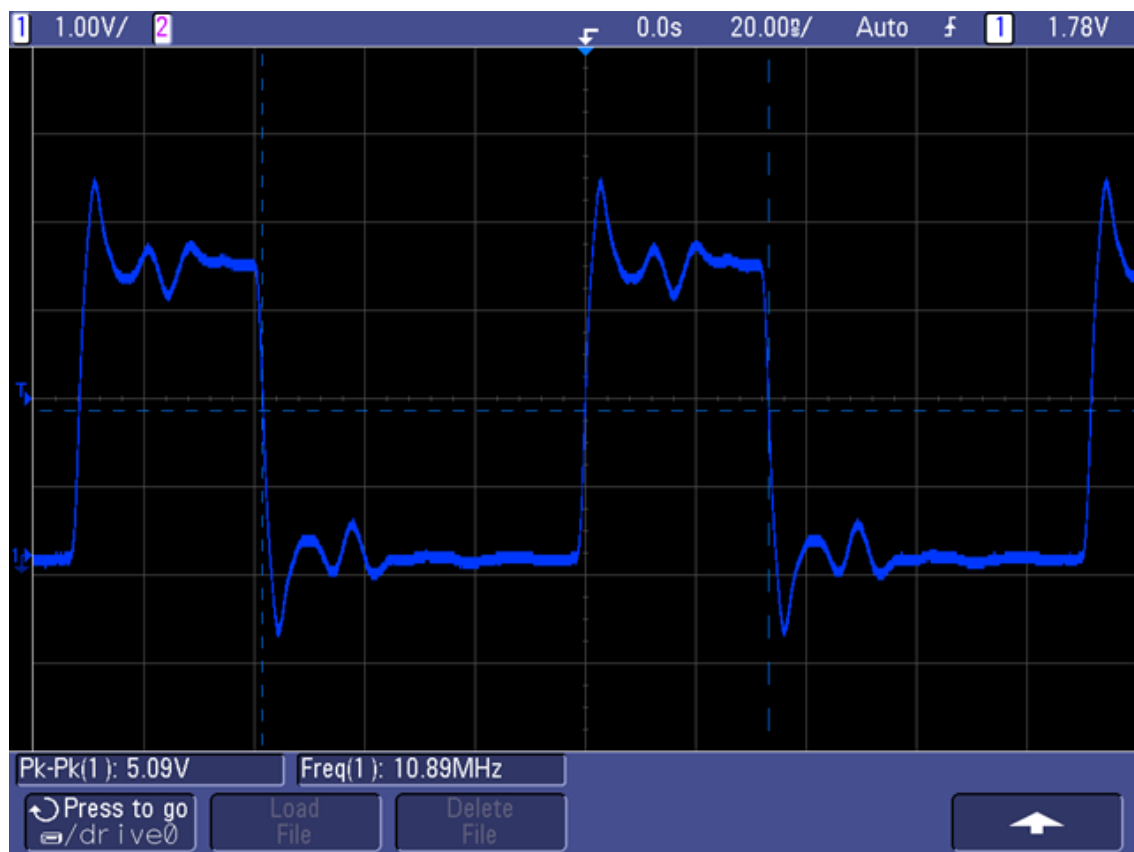
int main(void)
{
    while(1) {
        LED = 1;
        usleep(100000);
        LED = 0;
        usleep(100000);
    }
}
```

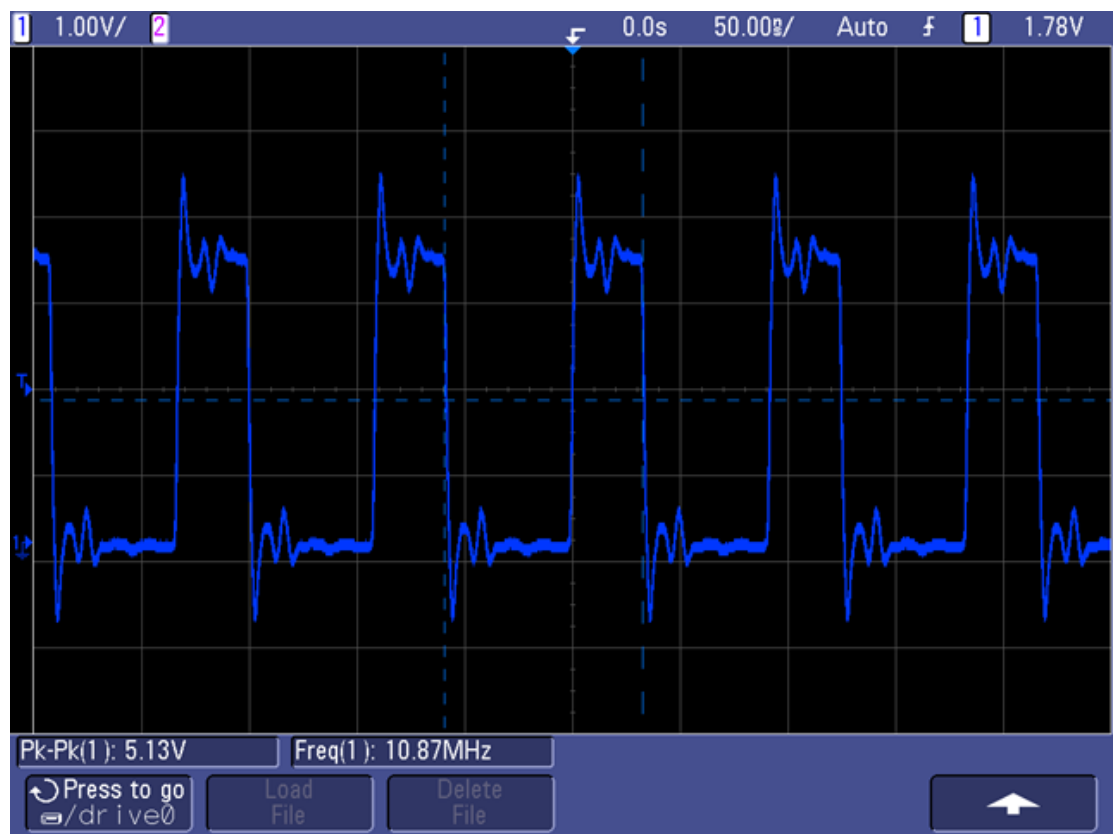
```
return 0;  
}
```

两种方法都讲完了，我更推荐第一种，斩草除根式。大家要注意，只有使用 NIOS II/f 这种类型的 CPU 才会出现这种问题。

新的核心板测好以后，顺便测了下 IO 速度，给大家看看吧，测下来 IO 频率不到 11MHz。

CPU : NIOS II/f
时钟 : 120MHz
示波器 : Agilent mso6032a 300MHz 2GSa/s





二、对寄存器结构体的详细解析

这一节，我们针对大家提出的有关定义寄存器结构体的问题进行解析。在 NIOS II 软件开发过程中，如果使用我们提出的寄存器操作方式的话，首先需要定义一个寄存器结构体，之所以这样做是为了在软件书写过程中操作方便，更是为了增强程序的可读性。我们就拿 UART 来举例说明。

首先，我们看一下 UART 的寄存器说明，如下表所示

Offset	Register Name	R/W	Description/Register Bits													
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata	RO	Reserved					(1)	(1)	Receive Data						
1	txdata	WO	Reserved					(1)	(1)	Transmit Data						
2	status (2)	RW	Reserved	eop	cts	dcts	(1)	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	control	RW	Reserved	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	divisor (3)	RW	Baud Rate Divisor													
5	endof-packet (3)	RW	Reserved					(1)	(1)	End-of-Packet Value						

我们通过上表可以看到，UART 包括 6 个寄存器（由于最后一个寄存器一般不用，所以建立的结构体中没有加入它），假设基地址为 0x00 的话，那么他们的地址分别为 0x00,0x01,0x02,0x03,0x04,0x05。也就是说，各个寄存器之间是存在顺序的。那么，在我们建立结构体过程中也要注意他们的顺序问题。建立的结构体如下所示

```
typedef struct{
    union{
        struct{
            volatile unsigned long int RECEIVE_DATA    :8;
            volatile unsigned long int NC                :24;
        } BITS;
        volatile unsigned long int WORD;
    } RXDATA;

    union{
        struct{
            volatile unsigned long int TRANSMIT_DATA    :8;
            volatile unsigned long int NC                :24;
        } BITS;
        volatile unsigned long int WORD;
    } TXDATA;
}
```

```

union{
    struct{
        volatile unsigned long int PE      :1;
        volatile unsigned long int FE      :1;
        volatile unsigned long int BRK     :1;
        volatile unsigned long int ROE     :1;
        volatile unsigned long int TOE     :1;
        volatile unsigned long int TMT     :1;
        volatile unsigned long int TRDY    :1;
        volatile unsigned long int RRDY    :1;
        volatile unsigned long int E       :1;
        volatile unsigned long int NC      :1;
        volatile unsigned long int DCTS    :1;
        volatile unsigned long int CTS     :1;
        volatile unsigned long int EOP     :1;
        volatile unsigned long int NC1     :19;
    } BITS;
    volatile unsigned long int WORD;
} STATUS;

```

```

union{
    struct{
        volatile unsigned long int IPE     :1;
        volatile unsigned long int IFE     :1;
        volatile unsigned long int IBRK    :1;
        volatile unsigned long int IROE    :1;
        volatile unsigned long int ITOE    :1;
        volatile unsigned long int ITMT    :1;
        volatile unsigned long int ITRDY   :1;
        volatile unsigned long int IRRDY   :1;
        volatile unsigned long int IE      :1;
        volatile unsigned long int TRBK    :1;
        volatile unsigned long int IDCTS   :1;
        volatile unsigned long int RTS     :1;
        volatile unsigned long int IEOP    :1;
        volatile unsigned long int NC      :19;
    } BITS;
    volatile unsigned long int WORD;
} CONTROL;

```

```

union{
    struct{
        volatile unsigned long int BAUD_RATE_DIVISOR    :16;
        volatile unsigned long int NC                    :16;
    } BITS;
    volatile unsigned long int WORD;
} DIVISOR;

}UART_STR;

```

对于这样一个大的结构体，我们来逐层分析一下：

第一，整个结构体由 5 个共用体组成，共用体的顺序是由寄存器的偏移量决定的，这一点前面已经有所叙述。

第二，每个共用体由一个结构体和一个 `volatile unsigned long int` 型的变量组成。

第二，共用体中的结构体由位域构成，位域中的内容也是存在顺序的，这个顺序是由寄存器的结构决定，而且是由低到高排列。其中，NC 表示该位为空位或保留，不能对其进行操作。

通过大家的反馈，除了语法问题以外，有两个问题需要说明一下：

1. 为什么里面的变量都定义成 `unsigned long int`？

首先需要说明一点，在 NIOS II 中，`volatile unsigned long int` 是 32 位，跟 `volatile int` 是一样的，`volatile long long int` 才是 64 位的。

有人会问，在寄存器的表格中，寄存器的位数是 0 到 15 的，也就是 16 位，那你为什么定义成 32 位的呢？其实，这个问题涉及到了 NIOS II 的地址对齐问题，它是属于硬件构架的范畴。

当系统中存在数据宽度不匹配的主从端口时就要考虑地址对齐的问题。地址对齐分为两类，一类是静态地址对齐，另一类就是动态地址对齐。一般来说存储器外设使用动态地址对齐，而寄存器外设使用静态地址对齐，之所以是这样，是由动态地址对齐和静态地址对齐的特点决定的，在静态地址对齐方式下，主端单次传输对应从端口的一次传输，而在动态地址对齐方式下，一个主端口读传输，则要引起多次从端口读传输。想要更加具体的了解他们特点的，大家自行查找吧，我在这里就不详细叙述了。

我们要将寄存器定义为 `volatile unsigned long int` 类型，就跟这个静态地址对齐有关系了。现在我们是 UART 端口 16 位，而 NIOS II 主端口 32 位的情况，在这种

情况下，NIO S II 主端口与 16 位 UART 端口进行数据传输时，只有 32 位的低 16 位有效，但是高 16 位也占用了地址空间，也就是说，UART 端口的 16 位实际上是占用了 32 位的。假设我们现在的基地址是 0X00，那么 6 个寄存器他们相对基地址的偏移分别为 0X00,0X01,0X02,0X03,0X04,0X05；那么，从主端口看，这 6 个寄存器的地址分别为 0X00,0X04,0X08,0X0C,0X10,0X14，而不是 0X00,0X01,0X02,0X03,0X04,0X05，也不是 0X00,0X02,0X06,0X08,0X0A,0X0C，这一点大家要特别注意。

2. 为什么要建立这样一个共用体呢，又有位域结构体又有一个 **volatile unsigned long int** WORD 变量，WORD 有啥用呢？

共用体的特点就是其中的成员占用同一个存储空间。也就说，由位域组成的结构体跟 WORD 是占用同一存储空间，而且他们都是 **volatile unsigned long int** 类型，那么，结构体中的每一个位域成员都对应 WORD 的一个位。当我们需要单独处理一个位的时候，我们就可以用位域，如下所示

```
RS232->CONTROL.BITS.IRRDY=1;//接收准备好中断使能
```

如果我们想要对状态寄存器整体清零呢，我们就可以用到 WORD 了，如下所示

```
RS232->STATUS.WORD=0;//清除状态寄存器
```

对于其他的寄存器都是一样的，在这里不再重复了。

这一节就讲到这里了，有问题请留言，或者这节与我联系。

三、 TCL 脚本文件

```
#####
#      URL:   http://www.oshcn.com
#      REV:   1.0
#  AUTHOR:   AVIC
#      DATE:   2010.6.19
#####

#-----GLOBAL-----#
set_global_assignment -name RESERVE_ALL_UNUSED_PINS "AS INPUT TRI-STATED"
set_global_assignment -name ENABLE_INIT_DONE_OUTPUT OFF

set_location_assignment PIN_23 -to RESET
set_location_assignment PIN_28 -to CLOCK
#-----SDRAM-----#
set_location_assignment PIN_175 -to S_DB[0]
set_location_assignment PIN_173 -to S_DB[1]
set_location_assignment PIN_171 -to S_DB[2]
set_location_assignment PIN_170 -to S_DB[3]
set_location_assignment PIN_169 -to S_DB[4]
set_location_assignment PIN_168 -to S_DB[5]
set_location_assignment PIN_165 -to S_DB[6]
set_location_assignment PIN_164 -to S_DB[7]
set_location_assignment PIN_205 -to S_DB[8]
set_location_assignment PIN_203 -to S_DB[9]
set_location_assignment PIN_201 -to S_DB[10]
set_location_assignment PIN_200 -to S_DB[11]
set_location_assignment PIN_199 -to S_DB[12]
set_location_assignment PIN_198 -to S_DB[13]
set_location_assignment PIN_197 -to S_DB[14]
set_location_assignment PIN_195 -to S_DB[15]

set_location_assignment PIN_179 -to S_A[0]
set_location_assignment PIN_180 -to S_A[1]
set_location_assignment PIN_181 -to S_A[2]
set_location_assignment PIN_182 -to S_A[3]
set_location_assignment PIN_185 -to S_A[4]
set_location_assignment PIN_187 -to S_A[5]
```

```

set_location_assignment PIN_188 -to S_A[6]
set_location_assignment PIN_189 -to S_A[7]
set_location_assignment PIN_191 -to S_A[8]
set_location_assignment PIN_192 -to S_A[9]
set_location_assignment PIN_176 -to S_A[10]
set_location_assignment PIN_193 -to S_A[11]

set_location_assignment PIN_207 -to S_CLK
set_location_assignment PIN_151 -to S_BA[0]
set_location_assignment PIN_150 -to S_BA[1]
set_location_assignment PIN_161 -to S_nCAS
set_location_assignment PIN_208 -to S_CKE
set_location_assignment PIN_160 -to S_nRAS
set_location_assignment PIN_162 -to S_nWE
set_location_assignment PIN_152 -to S_nCS
set_location_assignment PIN_206 -to S_DQM[1]
set_location_assignment PIN_163 -to S_DQM[0]
#-----USB-----#
set_location_assignment PIN_117 -to USB_DB[0]
set_location_assignment PIN_118 -to USB_DB[1]
set_location_assignment PIN_127 -to USB_DB[2]
set_location_assignment PIN_128 -to USB_DB[3]
set_location_assignment PIN_133 -to USB_DB[4]
set_location_assignment PIN_134 -to USB_DB[5]
set_location_assignment PIN_135 -to USB_DB[6]
set_location_assignment PIN_137 -to USB_DB[7]

set_location_assignment PIN_113 -to USB_A0
set_location_assignment PIN_115 -to USB_WR
set_location_assignment PIN_116 -to USB_nINT
set_location_assignment PIN_114 -to USB_RD
#-----LAN-----#
set_location_assignment PIN_129 -to LAN_nINT
set_location_assignment PIN_131 -to LAN_nWOL
set_location_assignment PIN_104 -to LAN_MOSI
set_location_assignment PIN_132 -to LAN_MISO
set_location_assignment PIN_103 -to LAN_SCK
set_location_assignment PIN_102 -to LAN_CS
set_location_assignment PIN_105 -to LAN_nRST
#-----VGA-----#

```

```

set_location_assignment PIN_142 -to VGA[0]
set_location_assignment PIN_143 -to VGA[1]
set_location_assignment PIN_144 -to VGA[2]
set_location_assignment PIN_146 -to VGA_HS
set_location_assignment PIN_145 -to VGA_VS
#-----LCD-----#
set_location_assignment PIN_8 -to LCD_CS
set_location_assignment PIN_12 -to LCD_A0
set_location_assignment PIN_11 -to LCD_SCL
set_location_assignment PIN_14 -to LCD_SI
#-----LED-----#
set_location_assignment PIN_69 -to LED[0]
set_location_assignment PIN_70 -to LED[1]
set_location_assignment PIN_72 -to LED[2]
set_location_assignment PIN_74 -to LED[3]
#-----KEY-----#
set_location_assignment PIN_3 -to KEY[0]
set_location_assignment PIN_5 -to KEY[1]
set_location_assignment PIN_4 -to KEY[2]
set_location_assignment PIN_10 -to KEY[3]
set_location_assignment PIN_6 -to KEY[4]
#-----UART-----#
set_location_assignment PIN_147 -to RXD
set_location_assignment PIN_149 -to TXD
#-----24LC04-----#
set_location_assignment PIN_112 -to I2C_SDA
set_location_assignment PIN_110 -to I2C_SCL
#-----PS2-----#
set_location_assignment PIN_139 -to PS2_DAT
set_location_assignment PIN_138 -to PS2_CLK
#-----DS1302-----#
set_location_assignment PIN_108 -to RTC_SCLK
set_location_assignment PIN_106 -to RTC_nRST
set_location_assignment PIN_107 -to RTC_DATA
#-----BUZZER-----#
set_location_assignment PIN_141 -to BUZZER
#-----DIG-----#
set_location_assignment PIN_44 -to DIG[0]
set_location_assignment PIN_43 -to DIG[1]
set_location_assignment PIN_46 -to DIG[2]

```

```
set_location_assignment PIN_56 -to DIG[3]
set_location_assignment PIN_57 -to DIG[4]
set_location_assignment PIN_48 -to DIG[5]
set_location_assignment PIN_47 -to DIG[6]
set_location_assignment PIN_45 -to DIG[7]
set_location_assignment PIN_58 -to SEL[5]
set_location_assignment PIN_59 -to SEL[4]
set_location_assignment PIN_60 -to SEL[3]
set_location_assignment PIN_61 -to SEL[2]
set_location_assignment PIN_63 -to SEL[1]
set_location_assignment PIN_64 -to SEL[0]
#-----END-----#
```

四、 NIOS II 常见问题解答 (FAQ)

1. Q: NIOS 能做浮点运算么？

A: NIOS 可以进行浮点运算,完全可以替代 MCU,时钟频率可以跑到 100Mhz,比 ARM7 还要快,ARM7 时钟频率一般为 72Mhz 左右。

2. Q: NIOS 是否可以不使用 SDRAM 和并行 FLASH？

A: 首先说明一下,SDRAM 是用来运行程序的,FLASH 是用来存储程序代码的 (SDRAM 掉电丢失,FLASH 则不会),每次上电的时候,都需要将 FLASH 中的程序代码放到 SDRAM 中,然后再运行。FPGA 内部的 memory (onchip memory) 比较小,跑比较大的程序就很难了,所以我们外扩了 SDRAM,以便比较大的程序运行。当然,我们也可以将并行的 FLASH 换成串行的 FLASH,而且这样可以节省很多引脚。其实 EPCS1(4,16...)就是串行 FLASH,所以我们可以利用它来存储代码,至于如何设置,我后面通过博客形式来给大家讲解。

3. Q:我的 Quartus II 编译硬件时没有错误,但是在下载程序的时候去出现以下错误的信息?

Info: Started Programmer operation at Thu May 06 01:39:46 2010

Error: Application Nios2 on 127.0.0.1 is using the target device

Error: Operation failed

Info: Ended Programmer operation at Thu May 06 01:39:46 2010

A: 这种情况出现在 JTAG 模式下,你在使用 NIOS 下的 JTAG 功能 (比如利用 BLASTER 进行在线仿真) ,同时你又想下载*.sof 文件 (就是在 JTAG 模式下下载程序) 。简单说就是你的 JTAG 已经被占用了。解决办法就是关闭你正在使用的 JTAG 功能,然后再下载*.sof 文件。初学者经常会犯这样的错误,一定要注意。

4. Q: 新建工程中遇到以下错误提示:

The software setting(STF)file associated with this project is damaged.This may be fixed by copying your source files into a new c/c++ application project.For more details see the error log.

A: 出现这种情况的原因是工程的存放地址不要有中文,也不要有空格,这个问题还是 NiosII IDE 对中文字符支持不好引起的,把目录名都改成英文而且不要有空格就没有问题了。

5. Q:运行时出现下面提示错误:

Using cable "USB-Blaster [USB-0]", device 1, instance 0x00

Pausing target processor: not responding.

Resetting and trying again: FAILED

Leaving target processor paused

A：遇到这种问题是因为忘记将 NIOS 软核的复位（RESET）添加引脚了，或者是因为你所添加的引脚不是对应按键也能导致这种问题的发生。

6. Q：下载时出现以下提示错误：

Error: Can't configure device. Expected JTAG ID code 0x020010DD for device 1, but found JTAG ID code 0x020B40DD.

A：SOPC 所选器件和开发板上的不一致。

7. Q：在 NOIS II 中 Bulid 例程 hello_world 都出现了错误，错误提示为：

gdrive/c/altera/kits/nios2/components/altera_nios2/HAL/src/alt_busy_sleep.c:68: error: parse error before '/' token 等错误全部由 alt_busy_sleep.c 引起。

A：这是 quartus 和 NIOS 安装版本不一致造成的，大家要注意他们的版本一定要相同，不能混装。

8. Q：在 SOPC 中 Generate 出现如下错误是怎么回事？

Error: Generator program for module 'epcs_controller' did NOT run successfully. 只要在 SOPC 中加入 epcs_controller 就会出现此错误，无法生成一个元件。

A：问题的原因是 Quartus 和 Nios 安装的版本不一致。

9. Q：在 Quartus II 中编译出现如下错误怎么办？

Error: Can't place pins assigned to pin location Pin_AE24 (IOC_X65_Y2_N2)

A：有两个管都分配在 PIN_AE24 了，找到其中一个删除掉就可以了。

10. Q：如何在 NIOS II IDE 下跟踪查看变量的定义或者函数的定义？

A：按住 CTRL 键，鼠标移动到变量或者函数名的地方，就可以发现这些地方高亮显示，单击就可以进入到变量或者函数定义的地方。

11. Q：在 SOPC 添加 Avalon Trisatate Bridge 时，提示有如下错误，该如何解决？

Tri state bridge/tristate master requires a slave of type Avalon tristate.Please add a slave of type Avalon tristate. Generate 按钮为灰色，无法 Generate

A：需要一个专门接三态桥的设备，把 flash 添加到 sopc 中就可以了。

12. Q：在做 count_binary 这个例子时，出现一个错误：

error: 'BUTTON_PIO_IRQ' undeclared (first use in this function) BUTTON_PIO_IRQ 的值如何给他定义？

A : 这个错误可能是在 sopc builder 中定制的 pio 端口名称是否与程序中用的不一致,要和程序里的一致,把 pio 组件的名称就改为 button_pio。

13. Q : 在 NIOS II 中编译时出现如下错误怎么解决? 错误是不是由 SOPC 中的 RAM 引起?

region ram is full (count_binary.elf section .text). Region needs to be 24672 bytes larger.address 0x80c1f8 of count_binary.elf section .rwd data is not within region ram.Unable to reach edge_capture (at 0x00800024) from the global pointer (at 0x0081419c) because the offset (-82296) is out of the allowed range, -32678 to 32767

A : 出现这种情况的原因是使用了 onchip memory。由于一般 FPGA 内部的 ram 有限,不足以让你跑起 NIOS 的程序,所以导致内存泄流。如果还是出现这个问题,加一个 SDRAM 试试。

14. Q : 在 NIOS II IDE 中工程的 System Library 选项中的这几个选项代表什么意思? .text .rodata .rwd data 与 reset .exception 这几个地址之间的关系是什么?

A : .text : 代码区 .rodata:只读数据区,一般存放静态全局变量 .rwd data:可读写变量数据区另外还有.bss:存放未被初始化的变量。

.text — the actual executable code

.rodata — any read only data used in the execution of the code

.rwd data — where read/write variables and pointers are stored

heap — where dynamically allocated memory is located

stack — where function call parameters and other temporary data is stored

15. Q : 在 NIOS II IDE 中调试,编译通过的软件时,出现了下面的提示,是什么原因?

Using cable "ByteBlasterII [LPT1]", device 1, instance 0x00

Processor is already paused

Downloading 00000000 (0%)

Downloaded 57KB in 1.2s (47.5KB/s)

Verifying 00000000 (0%)

Verify failed

Leaving target processor paused

A : 1. SDRAM 的时序不对,有时候不正确的 pll clock phase shift for sdram_clk_out 就会导致 SDRAM 不能正常工作:

2. SDRAM 的连线不对,物理板子的连线问题

3. 在调试的时候,程序下载的空间不是非易丢失存储器 (non-volatile memory) 或者存储器的空间不够也会导致这个错误

4. QuartusII 的默认设置导致的错误, QuartusII 默认将所有没有使用的 IO 口接

地,这种时候可能导致某些元器件工作不正常;最好将不用的 IO 口设置为三态

5. USB-blaster 坏了,或者 JTAG 通信的信号噪声太大, JTAG 的端口需要一个弱上拉电阻来抗干扰

6. 确保你的 sdram 既连接到 CPU 的指令总线也连接到 CPU 的数据总线。

16. Q :在 SOPC 中加了一个 200KB 的 onchip_memory ,为什么在 Quartus II 编译时出现这个错误 ?

Error: Selected device has 105 RAM location(s) of type M4K RAM. However, the current design needs more than 105 to successfully fit

A : SOPC 中的 onchip_memory 和 M4K RAM 根本就不是一个概念。Quartus II 中编译出现这个错误,是由于设计中用到了太多的 M4K。

17. Q : 关于 sopc-builder 中 reset address 的设置,一直搞得不是很明白。

A : SOPC 中的 reset address 指定的是最终全部软件程序代码下载到的地方,并且程序从 reset address 启动。SOPC 中的 exception address 指定的是系统异常处理代码存放的地方。如果 exception address 和 reset address 不一样,那么程序从 reset address 启动后将把放在 reset address 处的系统异常处理代码拷贝到 exception address 。NIOS II 软件中的 text address 指定的是程序运行的地方。如果 text address 和 reset address 不一样,那么程序从 reset address 启动后将把放在 reset address 处的普通只读程序代码拷贝到 text address 。NIOS II 软件中的 rodata address 指定的是只读数据的存放地方。如果 rodata address 和 reset address 不一样,那么程序从 reset address 启动后将把放在 reset address 处的只读数据拷贝到 rodata address 。

NIOS II 软件中的 rwdata address 指定的是可读写数据的存放地方。如果 rwdata address 和 reset address 不一样,那么程序从 reset address 启动后将初始化 rwdata address 处的可读写数据。

18. Q : 如何提高 NIOS II 系统的性能 ?

A : 主要可以从这几个方面入手 :

- 1、使用 fast CPU 类型。
- 2、提高系统主频。
- 3、优先在 SRAM 中运行程序,SDRAM 次之,最后选择 FLASH 中运行。
- 4、使用片内 RAM 作为数据缓冲,片外 SRAM 次之,最后选 SDRAM。
- 5、IO 数据传输尽可能采用 DMA。
- 6、对能并行处理的数据考虑使用多 CPU 协同处理。
- 7、典型算法做成用户指令,有 256 条可以做,足够你用的。
- 8、能用 HDL 模块来完成工作吗?能,就用 HDL 模块做成外设来完成吧
- 9、采用 C2H。

19. Q:这个问题经常会出现 :

Verifying 000xxxxx (0%)

Verify failed between address 0xxxxxxx and 0xxxxxxx

Leaving target processor paused

A : 1. 首先要根据 address 后面的两个地址判断出错的到底是什么器件。一般情况出现错误的大多是存储器。判断的方法是根据 sopc 中的地址，或者是 system.h 中的地址，查找相应出错的器件。

2. 检查硬件焊接是否正常。很多时候有些问题是硬件焊接造成的，这个主要针对的是自己焊接的板子，一旦地址数据总线有任何焊接问题，都会出现 verify failed 错误。

3. 检查 sopc 中的 componet 是否正常。如果是自己加入的接口逻辑，这个部分要确认其正常与否。

4. 检查 Quartus 中的设计：

- 检查引脚锁定是否正确，必须一一对应，不能有一个错误；
- 地址对齐问题：针对 8、16、32 位的外部存储器，对应地址最低位的应该是 0、1、2。也就是说如果用 16 位的外部存储器，那么它的最低位是 ADD[1]，而 ADD[0]是不用的，其他同理。
- 数据总线必须是双向 IO 口，这点很容易忽略。
- 如果是 SDRAM，需要计算并设定 PLL 的相移。

5. Nios IDE 中检查项目设计是否正确。

20. Q 在有 SDRAM 和 FLASH 的 SOPC 中 如果最后要下载程序到 FLASH 中，在 NIOS II 中用 FLASH PROGRAMER 下载时有

Programming flash with the project

```
"$SOPC_KIT_NIOS2/bin/nios2-flash-programmer"
```

```
--base=0x03000000
```

```
--cable='ByteBlasterII [LPT1]'
```

```
--sidp=0x04001038
```

```
--id=417604522
```

```
--timestamp=1257256685 "ext_flash.flash"
```

Empty flash content cannot be programmed or verified 的错误。提示说要给 flash 编程的数据是空的。

A : 原因是 RESET VECTOR 指向了 SDRAM，应该指向 FLASH。因为这里要下载的是一个工程，下载的是工程内容，reset vector 指向了 SDRAM 与 flash 就没有关系了，也就不会下载到 flash 中。

21. Q : Using cable "USB-Blaster [USB-0]", device 1, instance 0x00

Resetting and pausing target processor: OK

Reading System ID at address 0x00201040: verified

No CFI table found at address 0x00000000

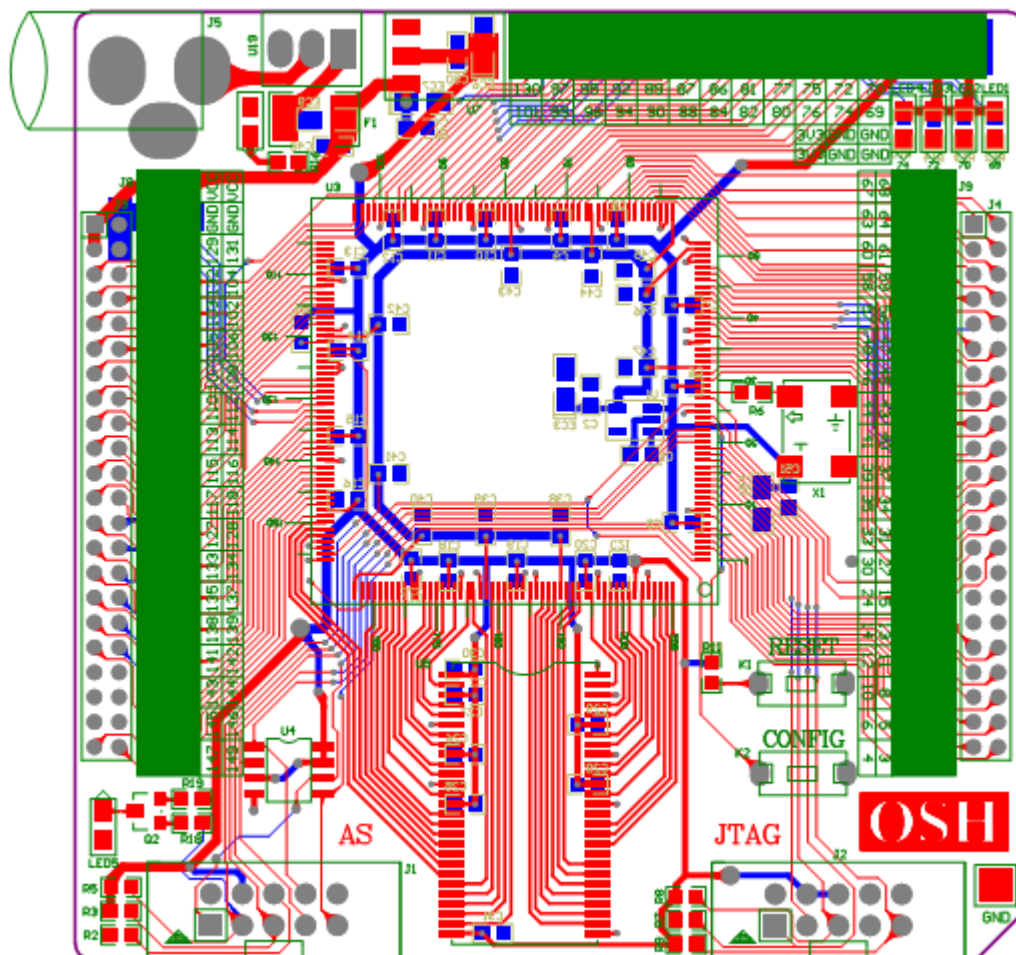
Leaving target processor paused 下载 FLASH 时出现这个问题？

A：这个是 FLASH 设置问题，需要几点需要注意：

1. 数据线双向；
2. 16 位和 8 位模式要分清楚；
3. 在 SOPC builder 中要根据自己的 FLASH 类型正确设置参数，并行 FLASH 要加三态桥。

五、 黑金开发板印制板

1. 核心板



2. 扩展板

