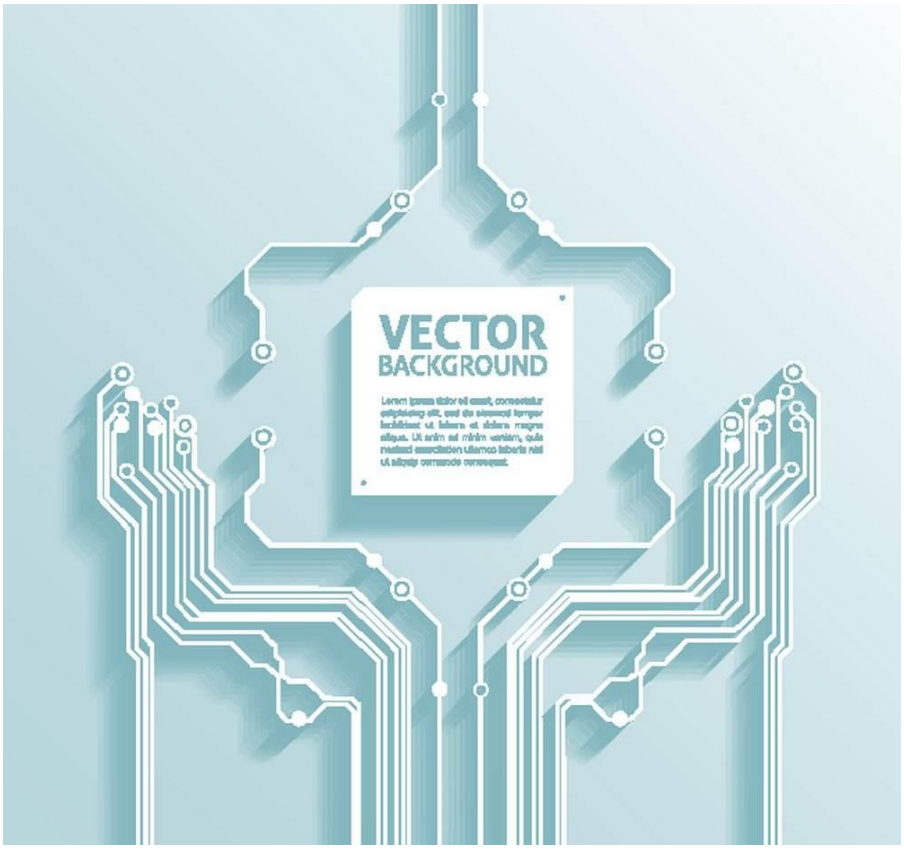
SadAngel 带你学硬综 FPGA

第一版



修改日期：2019 年 7 月 9 日

SadAngel

571102482@qq.com

Github:SadAngelF

前 言

硬件综合训练中的 FPGA 部分被大多数学长学姐吐槽太难，但是我想告诉大家其实难不难都是靠学的，FPGA 资料虽然少，但是还是可以找到的，这个文档是我们小组完成过程中遇到的问题和流程的归纳，希望能给大家一点启发，从而做出更好的作品出来，早日打败对面那些用树莓派和 STM32 的队伍。

这个手册不建议大家直接阅读，建议先学习一部分 verilog 语言并阅读完附带的康芯手册，之后再进行阅读本手册，进行理解的提升。手册主要介绍了整个流程和一些踩坑的点，希望大家可以作为参考。

在这里要感谢我的队友，孙铭洋，如果不是他，那么我肯定完成不了这个作品，同时他也提出了宝贵的建议和修改，封面就是出自孙铭洋之手，如果吐槽，请吐槽他。另外，我们的队友波波，因为其他的事情，所以没能和我们一起完成这个手册撰写，但他给我们的鼓励和支持是莫大的，嘻嘻嘻，而且腰很舒服（谁摸谁知道）。

另外，这个过程，感谢大连理工大学计算机硬件实验室和实验老师，徐刚老师帮助我们完成了很多硬件和经验上的分享，甚至帮助我们联系了官方的工作人员询问器件设计和使用问题，而最后的 PID 调节部分，董校老师也给出了很多宝贵的意见，感谢实验室的环境和老师的教导。

如果手册有什么改良意见或者其他不足，或者想寻求帮助，请联系我：冯湛搏，SadAngel，我的邮箱是 571102482@qq.com，QQ 也是这个，github 地址：https://github.com/SadAngelF，这里面也有一些其他的工程和算法，大家感兴趣也可以找我。

如果大家觉得有哪里需要修改或者问题需要补充，欢迎联系我，或者找我学习，嘻嘻嘻。

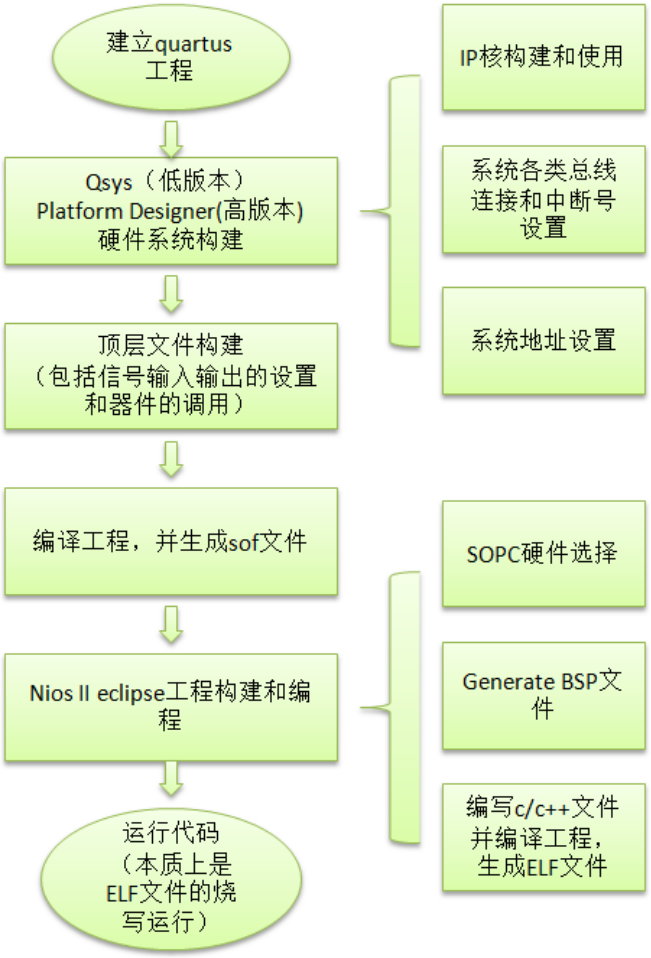
SadAngel

2019 年 7 月 9 日于实验室

目录

第一章 基于 Nios II 系统的 FPGA 开发流程

第一节 开发流程概述



整个开发过程如上图所示，这里介绍一下各个部分的大概关系：整个开发项目由一个工程构成，即 Quartus 工程，之后分成两个部分，一个部分是硬件：Qsys（Quartus 13 版本之前）或者 Platform Designer（Quartus 17 版本）,这个部分其实既可以用这两个工具来可视化完成（最后是一个.qip 文件），也可以直接写 VHDL 或者 Verilog 语言（最后是一个.v 文件）。另一个部分是软件，使用 Nios II eclipse 工具完成。

这两个部分的具体操作都可以在附录中康芯的实验讲义中获得，因此后面的部分以讲解三个子工程（顶层文件、Qsys/PlatformDesigner 和 Nios II eclipse 工程）的对应相关关系为主。

第二节 硬件开发讲解

具体而言，硬件开发分成两个部分，第一部分在于硬件器件的控制驱动书写（.v 文件），第二部分在于 Qsys/PlatformDesigner 布线和控制（最终会产生.qip 文件）。

首先，对于硬件器件的控制驱动，我会用两个 Demo 来讲解，利用数码管控制，来讲解信号量的声明规则和写时序设计；利用电机驱动，来讲解读写的完整过程。

（1） 数码管显示控制：

这部分讲解主要是为了讲解 FPGA 硬件驱动开发的一些变量定义的规范：

接口前缀	接口类型
asi	Avalon-ST 接收器（输入）
aso	Avalon-ST 源（输出）
avm	Avalon-MM 主端口
avs	Avalon-MM 从端口
axm	AXI 主端口
axs	AXI 从端口
coe	导管
csi	时钟接收器（输入）
cso	时钟源（输出）
inr	中断接收器
ins	中断发送器
ncm	Nios II 定制指令主端口
ncs	Nios II 定制指令从端口
rsi	复位接收器（输入）
rso	复位源（输出）
tcm	Avalon-TC 主端口
tcs	Avalon-TC 从端口

上图就是信号命名前缀的规范，当然，只看这个我们是无法获得任何信息的，你得查询这些接口类型的说明。最常用的三个信号前缀是 avs、csi 和 coe。

接下来看代码部分：

//模块声明，标注了各个量的属性，主要在于输入和输出，前缀表示变量的硬件属性，例如 avs 是需要通过 avalon 总线和 Nios 核心通信的信号量，而 csi 是供给整个系统的时钟信号，coe 则是负责输出的信号量

```
module dpy(
    input    csi_clk,
    input            csi_reset_n,
    input    avs_chipselect_n,
```

```
input             avs_address,
input   [3:0]   avs_byteenable_n,
input   avs_write_n,
input [31:0] avs_writedata,
input             avs_read_n,
output [31:0] avs_readdata,

output [7:0] coe_r_dig0,coe_r_dig1,coe_r_dig2,coe_r_dig3
);

reg [7:0] r_dig0,r_dig1,r_dig2,r_dig3;
```

//读写控制部分，对于每个器件而言，控制信号的获取和采集数据的收回都要通过 avalon 总线来完成，这个过程由 avalon 总线协议规定（感兴趣的同学可以在网上查到这个总线协议），例如下面这部分外部输入了两个信号量，一个是时钟信号，另一个是复位信号，复位信号发生时，会对寄存器初始化。下面的 if (~avs_chipselect_n & ~avs_write_n & avs_address == 1'b0)部分，则是属于读写控制，avalon 总线控制分为三个时序：片选 avs_chipselect_n（cpu 和器件通信控制）、写控制 avs_write_n（控制这次通信是否写入器件）、地址量 avs_address（读写的地址，表示你想写/读的寄存器地址），这里的三个信号量，对于片选和读写控制而言，高低/与非都无所谓，只要你将读和写区分开即可，这个时序是自己设计的，后面我会展示出数码管的设计部分，这个是没有读取的，所以我也没有设计读取部分，而是只设计了写的部分。

```
always@(posedge csi_clk or negedge csi_reset_n)
    if(~csi_reset_n)
    begin
            r_dig0 <= 7'b0;
            r_dig1 <= 7'b0;
            r_dig2 <= 7'b0;
            r_dig3 <= 7'b0;
    end
    else if (~avs_chipselect_n & ~avs_write_n & avs_address == 1'b0)
    begin
            if (~avs_byteenable_n[0])
                    r_dig0 <= avs_writedata[7:0];
            if (~avs_byteenable_n[1])
```

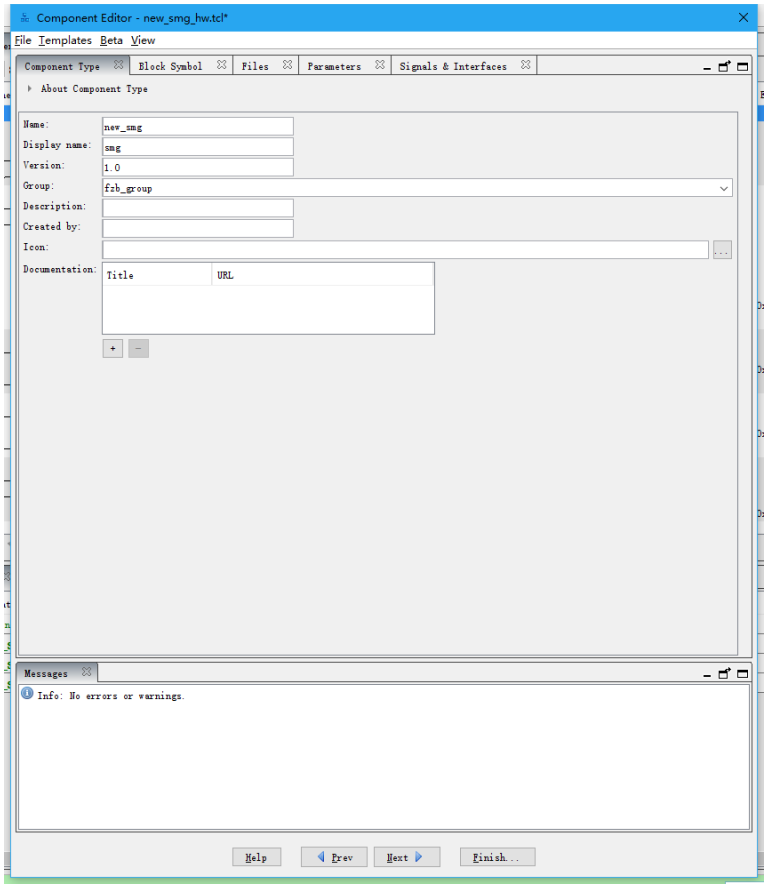
```
                r_dig1 <= avs_writedata[15:8];
        if (~avs_byteenable_n[2])
                r_dig2 <= avs_writedata[23:16];
        if (~avs_byteenable_n[3])
                r_dig3 <= avs_writedata[31:24];
    end
```

//verilog 语言中，不可以直接对输出管脚控制，因此这里我们通过 assign，将输出和寄存器连接，相当于在两个量上连接了导线，这样就可以实现输出。

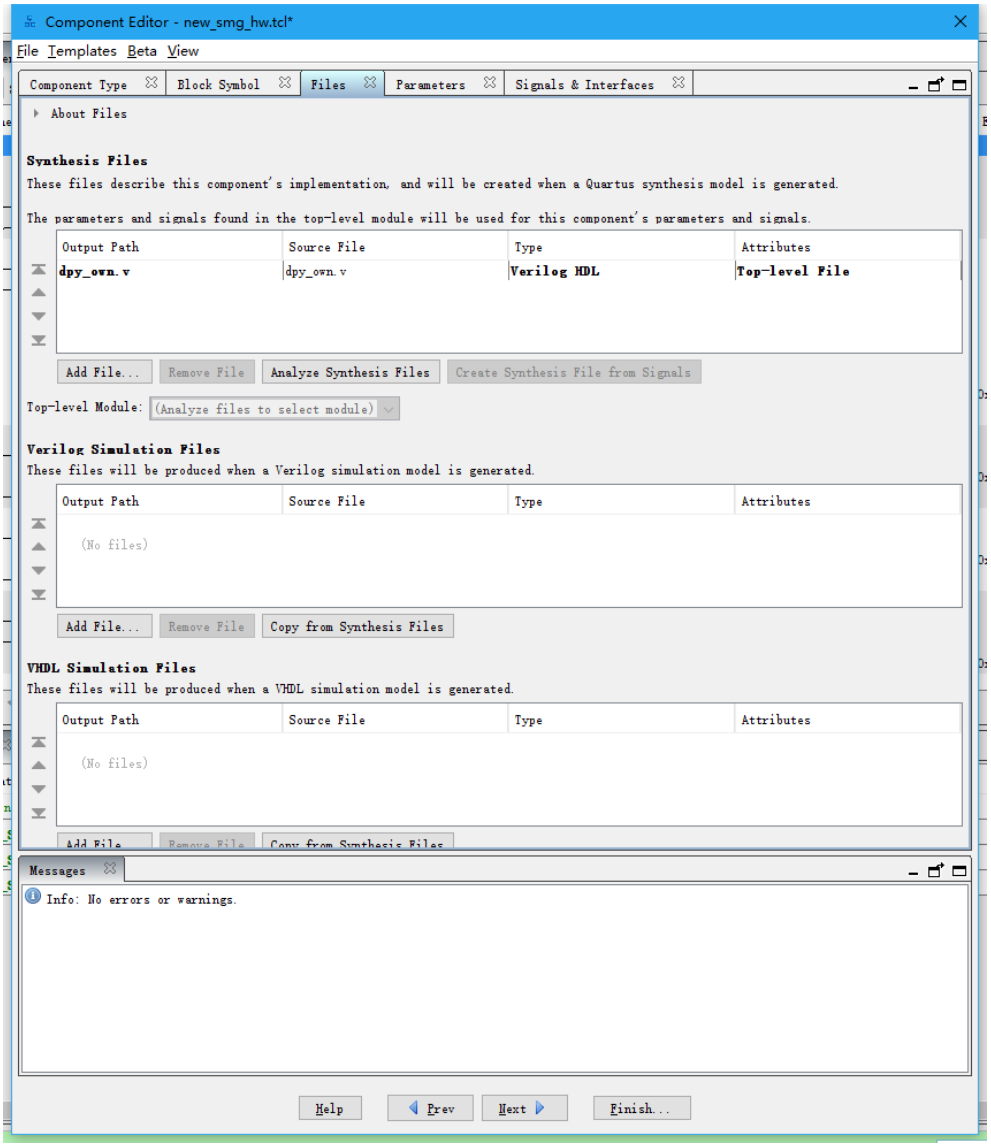
```
    assign coe_r_dig0 = r_dig0;
    assign coe_r_dig1 = r_dig1;
    assign coe_r_dig2 = r_dig2;
    assign coe_r_dig3 = r_dig3;

endmodule
```

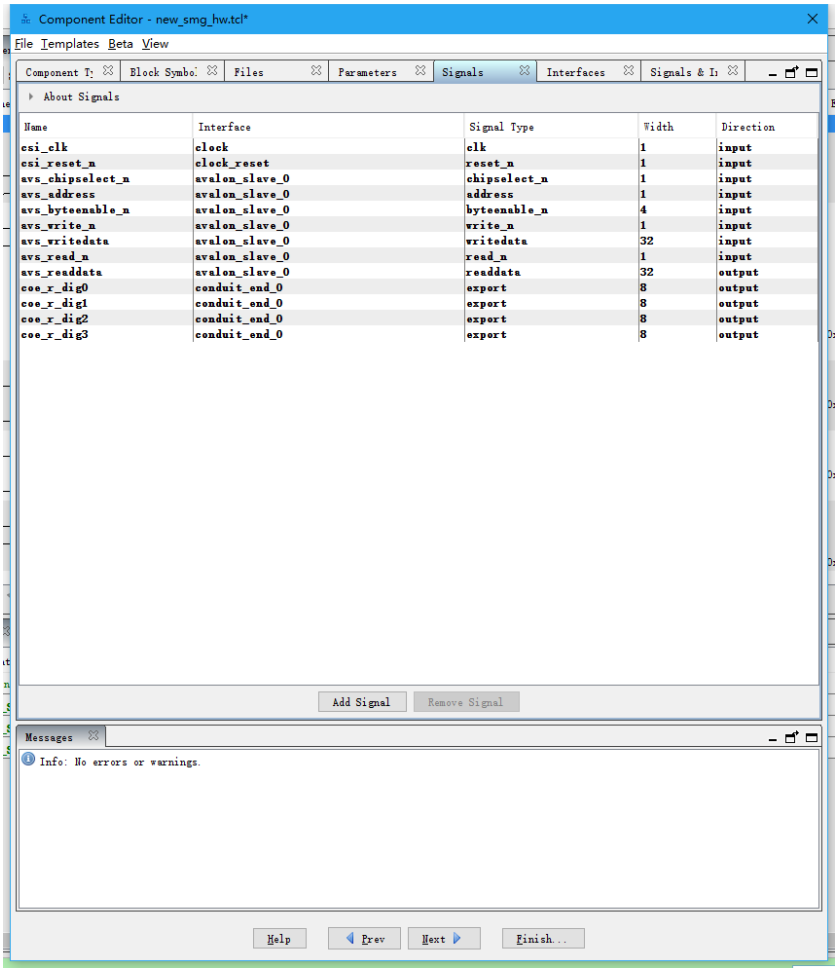
对于这个模块驱动，在 Qsys/PlatformDesigner 中生成 IP 核：



这个界面下，可以设计自己 IP 核心的名字和组别的划分。



文件分析和之前的旧版本没有明显区别，但是 13.1 版本没有对.v 文件的纠错，而 17 版本有了提示错误的功能，建议使用 17 版本来调试你的驱动文件。



这个页面下，是输入输出信号量属性控制，如果你在写.V 文件时遵守了前缀规范，这里就不需要额外改动，因为系统可以自动识别信号量的前缀，但是后面的分析，我们会发现并不是所有人都遵守这个规范，因此还是需要自己理解这部分属性：

Name	Interface	Signal Type	Width	Direction
csi_clk	clock	clk	1	input
csi_reset_n	clock_reset	reset_n	1	input
avs_chipselect_n	avalon_slave_0	chipselect_n	1	input
avs_address	avalon_slave_0	address	1	input
avs_byteenable_n	avalon_slave_0	byteenable_n	4	input
avs_write_n	avalon_slave_0	write_n	1	input
avs_writedata	avalon_slave_0	writedata	32	input
avs_read_n	avalon_slave_0	read_n	1	input
avs_readdata	avalon_slave_0	readdata	32	output
coe_r_dig0	conduit_end_0	export	8	output
coe_r_dig1	conduit_end_0	export	8	output
coe_r_dig2	conduit_end_0	export	8	output
coe_r_dig3	conduit_end_0	export	8	output

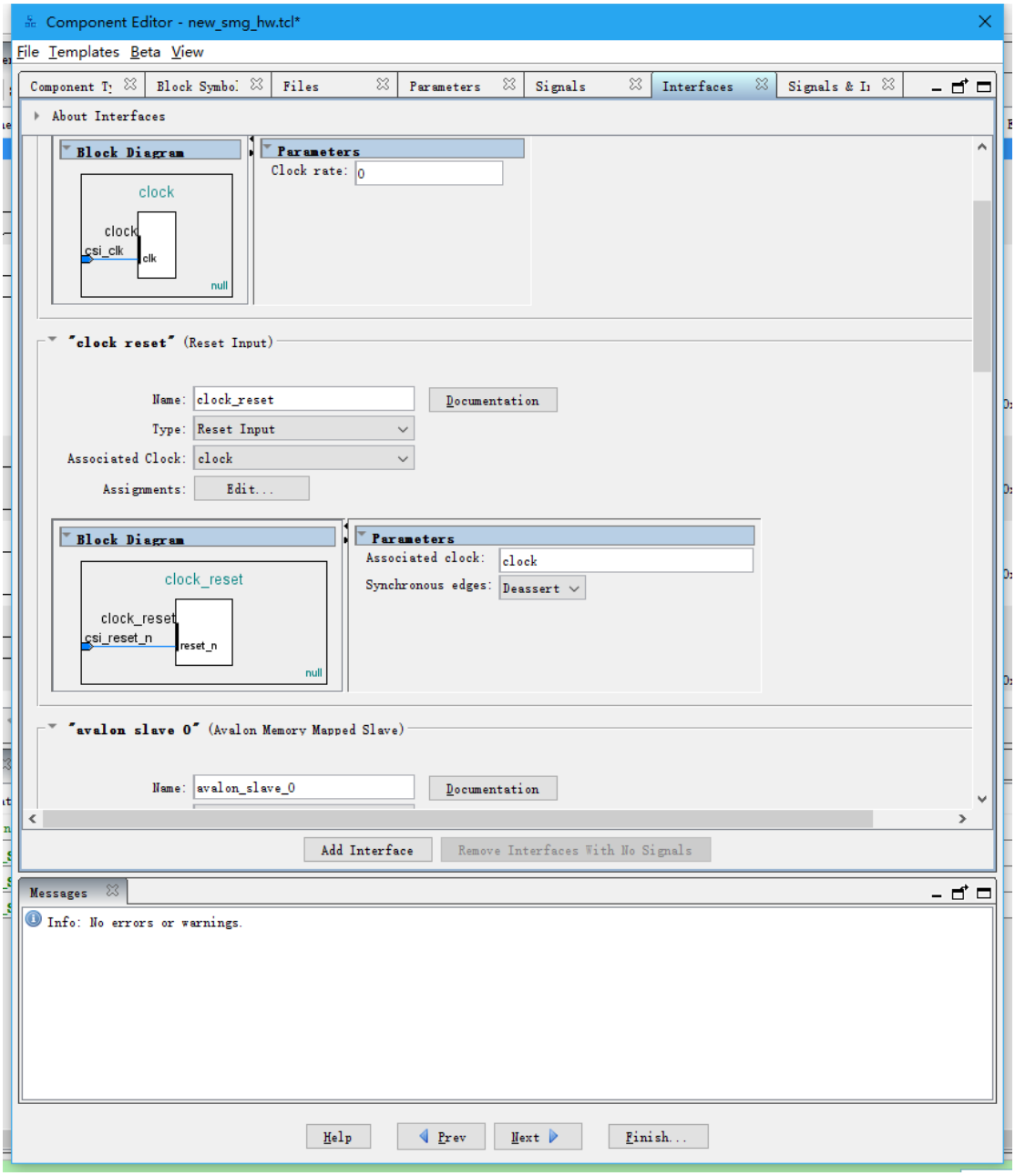
Name 这一列是你在硬件中声明的 input 和 output 的信号量的名字，这个可以在.v 文件中对应到相应的信号量。

Interface 这一列表示信号量的属性，基本上用到的也就上图中的四种，clock 代表时钟信号（连接外部的时钟信号），clock_reset 代表复位信号（连接外部复位信号），avalon_alave_0 代表 avalon_slave 总线（所有的片选、读

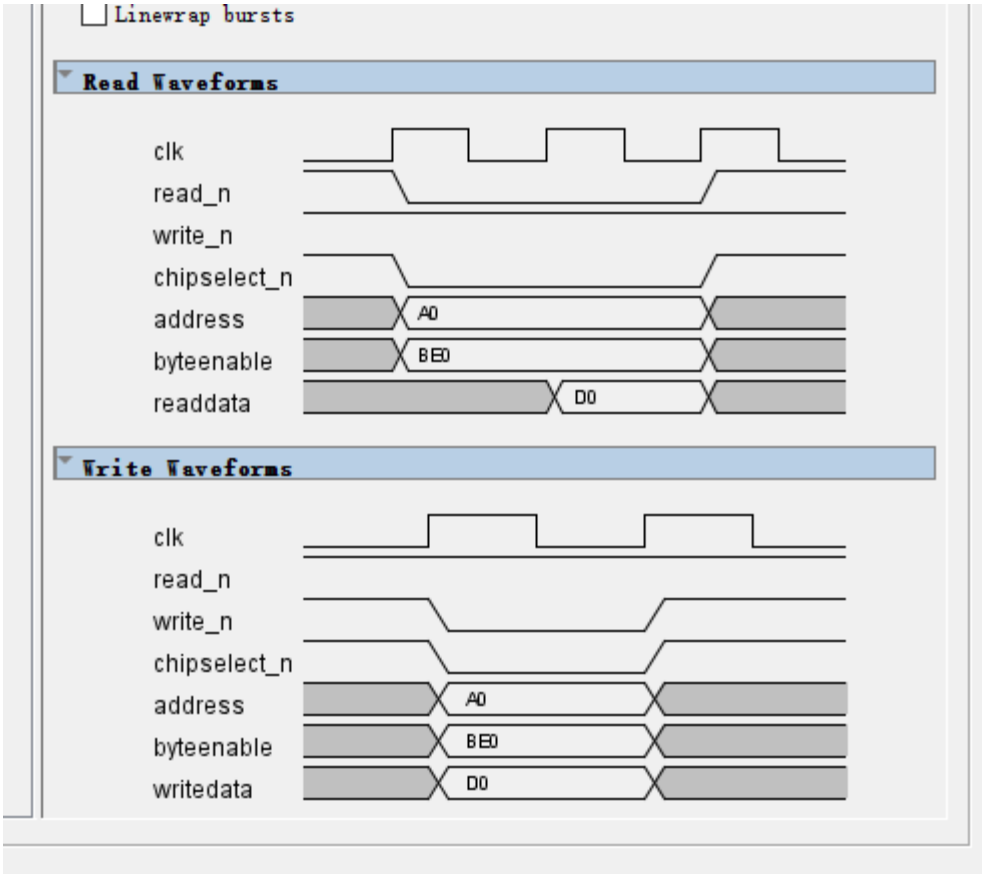
写和数据信号都是这个属性），conduit_end_0 代表输出（所有你要控制器件或者显示亮灯的信号都是这个属性）

Signal_Type 这一列代表了具体的属性值，这部分可以直接通过名字看出来是什么意思，比如片选和写入的数据，这里就和你当初设计的信号量相符即可。

后面的 Width 和 Direction 不再赘述。



这一部分是具体信号量和各部分相关关系控制，具体书写方法可以参考康芯手册，这里只说明一下时序的问题，可以下滑看到：



例如上图则是控制时序，在图中，我们可以看出，写时序，是在 write_n 信号低和片选为低时发生的写过程，这个时序和之前.v 文件中写的：

```
else if (~avs_chipselect_n & ~avs_write_n & avs_address == 1'b0)
begin
```

是相符的，因此可以设计自己的时序，参考器件手册来完成这部分驱动时序的设计。

至于写好之后的连线和命名部分，就不在这里解释了，操作看康芯手册，各个模块对应关系，我会在之后的软件部分中写出。

（2） 电机控制驱动：

众所周知，最简单的电机控制是通过输出不同占空比的方波来实现速度控制，占空比的大小决定了电机的转速，因此，这个.v 文件具体分成了几个部分，下面，我们来上代码，具体讲解（注意这个代码没有遵守前缀规范）：

```
//同样，模块的声明和信号属性的声明
module TERASIC_DC_MOTOR_PWM(
   input                                clk,
   input                                reset_n,
   //
   input                                s_cs,
   input          [1:0]                 s_address,
   input                                s_write,
   input          [31:0]                s_writedata,
   input                                s_read,
   output reg     [31:0]                s_readdata,

   //
   output reg                           PWM,
   output reg                           DC_MOTOR_IN1,
   output reg                           DC_MOTOR_IN2
);

   //宏定义部分，注意和 c/c++的不同在于使用前面有个`
`define REG_TOTAL_DUR       2'd0
`define REG_HIGH_DUR 2'd1
`define REG_CONTROL         2'd2


//////////////////////////////////////
// MM Port
reg         motor_go;
reg         motor_forward;
reg         motor_fast_decay;
//读写控制的部分，按照信号量的名称和状态来看何时读取数据，又何时输出数据。
always @(posedge clk or negedge reset_n)
begin
   if (~reset_n)
   begin
```

```
                // PWM
                high_dur <= 0;
                total_dur <= 0;

                // MOTOR
                motor_go <= 1'b0;
                motor_forward <= 1'b1;
                motor_fast_decay <= 1'b1;
        end
        else if (s_cs && (s_address == `REG_CONTROL))
        begin
                if (s_write)
                        {motor_fast_decay, motor_forward, motor_go} <=
s_writedata[2:0];
                else if (s_read)
                        s_readdata <= {29'b0, motor_fast_decay,
motor_forward, motor_go};
        end
        else if (s_cs & s_write)
        begin
                if (s_address == `REG_TOTAL_DUR)
                        total_dur <= s_writedata;
                else if (s_address == `REG_HIGH_DUR)
                        high_dur <= s_writedata;
        end
        else if (s_cs & s_read)
        begin
                if (s_address == `REG_TOTAL_DUR)
                        s_readdata <= total_dur;
                else if (s_address == `REG_HIGH_DUR)
                        s_readdata <= high_dur;
        end
    end



    ////////////////////////////////////////
```

```
    //电机的方向控制，这一部分应该很简单能看懂
    always @(*)
    begin
        if (motor_fast_decay)
        begin
                // fast decay
                if (motor_go)
                begin
                        if (motor_forward)
                                {DC_MOTOR_IN2, DC_MOTOR_IN1,PWM} <=
{1'b1, 1'b0,PWM_OUT}; // forward
                        else
                                {DC_MOTOR_IN2, DC_MOTOR_IN1,PWM} <=
{1'b0, 1'b1,PWM_OUT}; // reverse
                end
                else
                        {DC_MOTOR_IN2, DC_MOTOR_IN1,PWM} <= {1'b1,
1'b1,1'b0};
        end
        else
        begin
                // slow decay
                if (motor_go)
                begin
                        if (motor_forward)
                                {DC_MOTOR_IN2, DC_MOTOR_IN1,PWM} <=
{1'b1, 1'b0,PWM_OUT}; // forward
                        else
                                {DC_MOTOR_IN2, DC_MOTOR_IN1,PWM} <=
{1'b0, 1'b1,PWM_OUT}; // reverse
                end
                else
                        {DC_MOTOR_IN2, DC_MOTOR_IN1,PWM} <= {1'b0,
1'b0,1'b0};
        end
    end
```

```
//////////////////////////////////////
// PWM
//这就是 pwm 的控制，按照 total_dur 和 high_dur 来调节占空比，本质上是一个计数器
reg                     PWM_OUT;
reg     [31:0]  total_dur;
reg     [31:0]  high_dur;
reg     [31:0]  tick;

always @ (posedge clk or negedge reset_n)
begin
   if (~reset_n)
   begin
        tick <= 1;
   end
   else if (tick >= total_dur)
   begin
        tick <= 1;
   end
   else
        tick <= tick + 1;
end
//控制输出相应的高低电平
always @ (posedge clk)
begin
        PWM_OUT <= (tick <= high_dur)?1'b1:1'b0;
end

endmodule
```

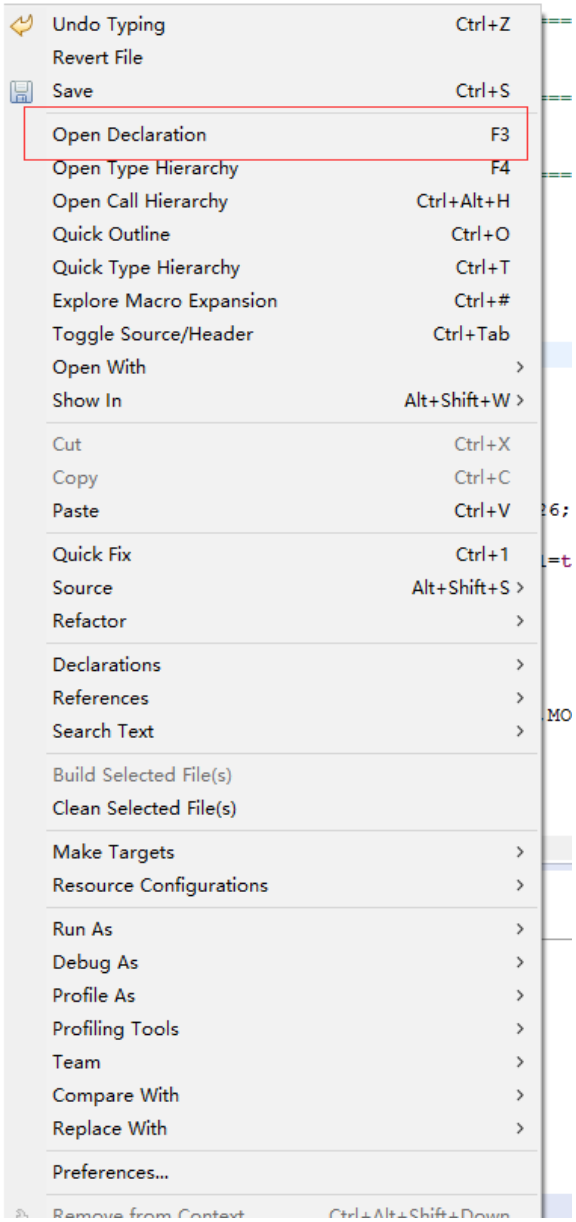
这个代码相对于数码管的代码来说，多了总线读取的部分（获得电机状态），而不只是输出信号控制，但是总体而言，时序并不复杂，应该能够阅读。

第三节 软件开发讲解

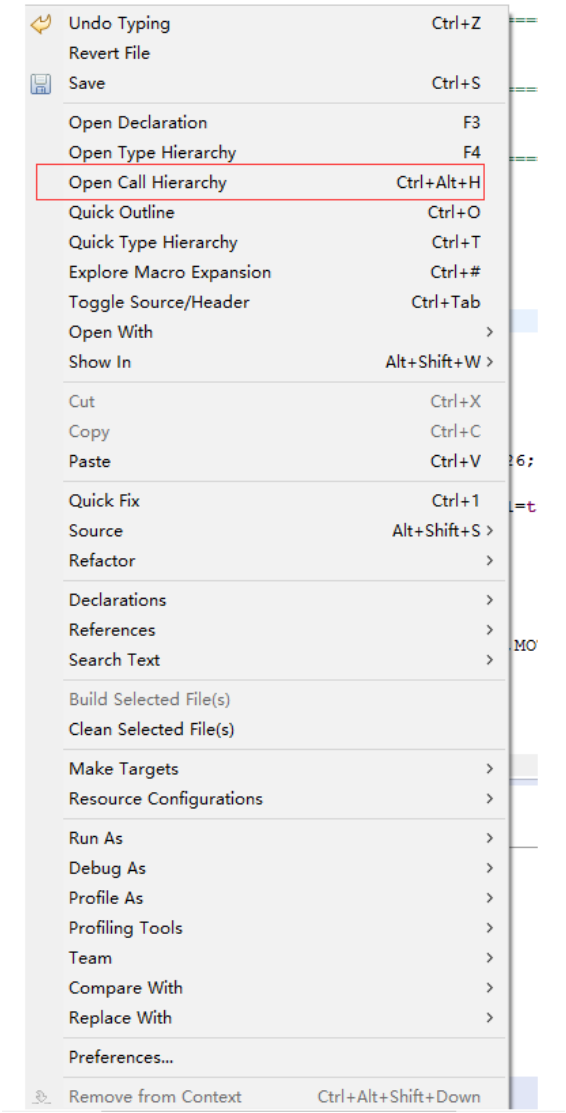
这个部分的软件开发，工具是 Quartus 中的 Nios II eclipse，首先声明，这个工具存在很多的 bug，所以需要耐心和对于源码的深刻理解，否则出现玄学问题，不好弄清根源。

这一部分因为是 c/c++的代码，因此不再根据源码来赘述，而是把我用到的一些好用而灵活的操作和注意事项告诉大家。

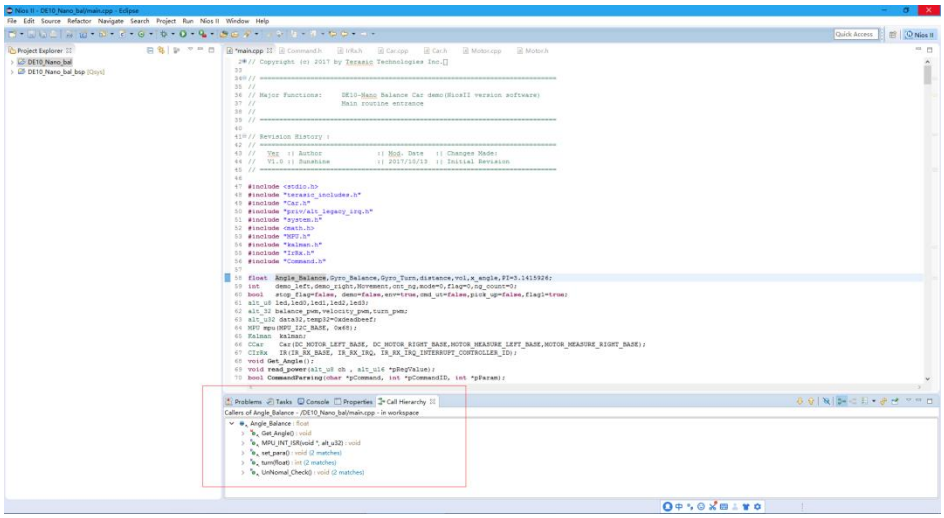
（1） 寻找函数/变量声明：点击右键之后，选择这个就行



（2） 寻找函数/变量的调用：点击右键之后，选择这个就行



之后可以在下图这个部分找到调用的地方：



（3） 其他的一些诸如定位 bug 或者定位出错的操作和正常的编译软件没有区别，因此也不再列出了。

（4） Nios II eclipse 工程和硬件的对应：

可能大家都比较奇怪，我这样写了 C/C++之后，怎么和硬件关联起来呢，这里就为大家讲一下这个关联过程。

首先，工程建立时，你会选择一个 SOPC 文件，这个文件时 Quartus 编译生成的一个硬件，因此你选择的这个硬件就是你代码对应的硬件。

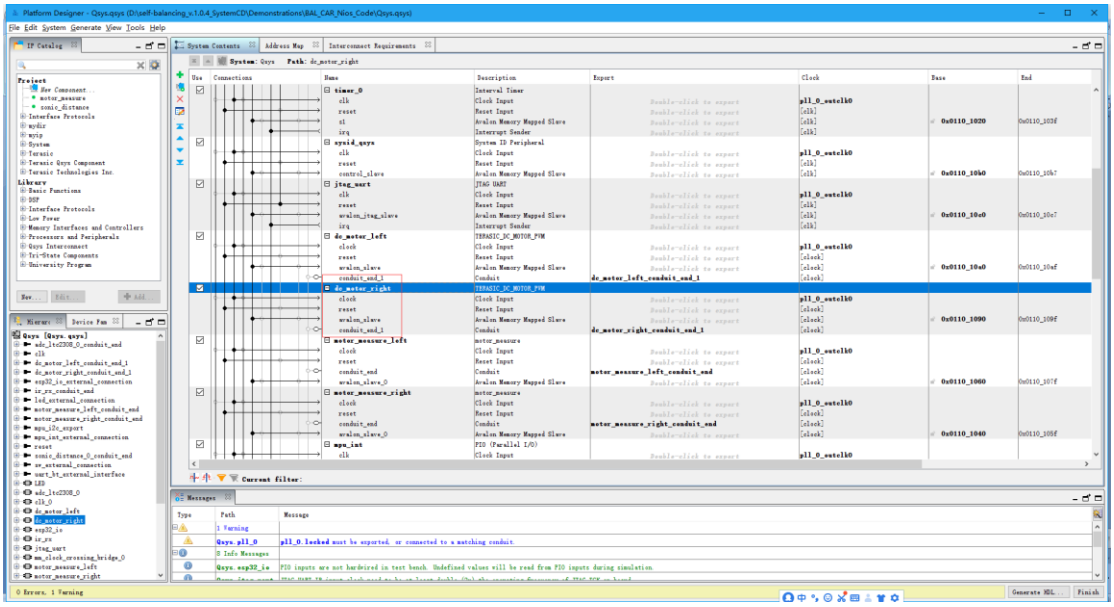
你在代码中引用的“system.h”就有这部分硬件的对应关系处理：

```
4
5  #define ALT_MODULE_CLASS_dc_motor_right TERASIC_DC_MOTOR_PWM
6  #define DC_MOTOR_RIGHT_BASE 0x1101090
7  #define DC_MOTOR_RIGHT_IRQ -1
8  #define DC_MOTOR_RIGHT_IRQ_INTERRUPT_CONTROLLER_ID -1
9  #define DC_MOTOR_RIGHT_NAME "/dev/dc_motor_right"
0  #define DC_MOTOR_RIGHT_SPAN 16
1  #define DC_MOTOR_RIGHT_TYPE "TERASIC_DC_MOTOR_PWM"
2
```

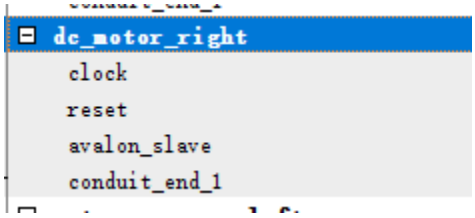
这部分是我从 system.h 文件截取的部分，一个宏定义的前缀代表了所对应器件的名称，而后缀对应了这个值的属性。

前缀：

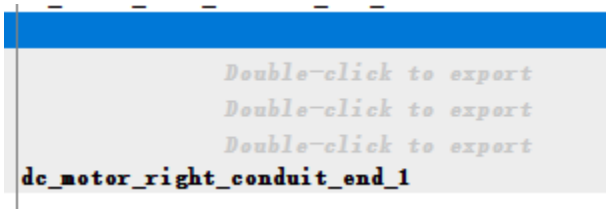
例如这里的 DC_MOTOR_RIGHT 是你在 Qsys/PlatformDesigner 设置的模块名称：



上图红色的部分放大：



这里的 dc_motor_right 就是你在 eclipse 工程中调用时，DC_MOTOR_RIGHT_BASE 的前缀，用来指明这个器件，而后缀 BASE 代表这个器件寄存器的起始地址，这就是两者的对应关系，即你在 eclipse 中操作对应器件，就是你在 Qsys/PlatformDesigner 设置的这个名称对应的器件。



而在你双击引出的管脚，这里的名称则是会在 PinPlaner 中出现，表明你要连接的管脚，同时这个管脚也会在之前的 IP 核设置中设置，如果在 IP 核中设置，名称就会是那里的名称。

后缀：

除了上面的 BASE 代表器件起始地址之外，另外一个常用的后缀是 IRQ，代表了中断信号量。

第二章 常见问题解决和处置办法

第一节 Qsys/PlatformDesigner 问题

（1） 地址冲突：

现象：界面下方报错，出现了几个器件地址重叠的问题。

解决办法：点击 System 后，点击 assign Base Addresses 通常可以解决这个问题，如果没有解决，那么吧其中一个器件地址前的锁点上，锁定地址后，重新自动分配地址。

第二节 Quartus 顶层文件编译问题

整个顶层文件既可以写成.V 文件，也可以利用图形化文件来设计，此处建议图形化设计（Block Diagram），除非你很熟悉 Verilog 语言。

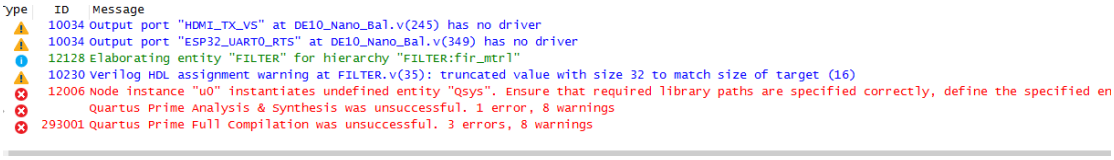
（1） Rom/ram 资源不够：

现象：编译时报错，无法为 rom/ram 分配相应大小

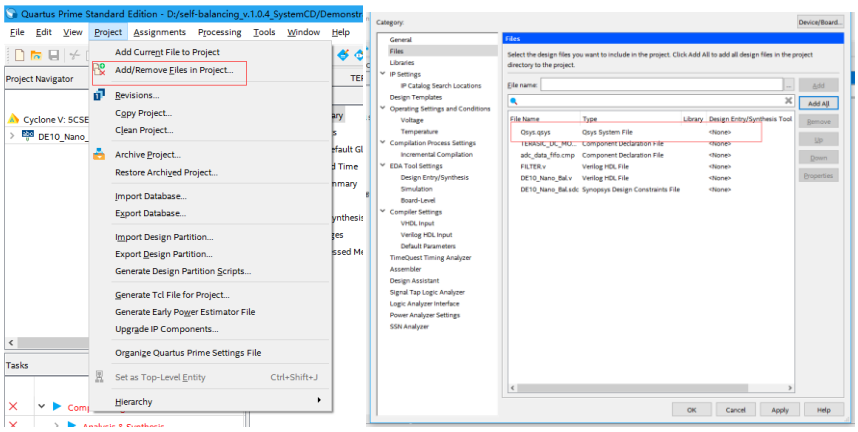
解决办法：重新打开 Qsys/PlatformDesigner，减小 ROM/RAM 的大小，重新编译整个工程。

（2） Qip 文件没有引用到工程：

现象：



解决办法：点击下图的这里，然后将 Qsys/PlatformDesigner 生成的.qip 文件加入工程，右图是加入工程之后的结果，重新编译，即可成功



（3） 如果其他问题报错，大家可以翻译错误信息，一般可以直接看懂，如果看不懂，那么就将信息部分填入百度，直接查询即可。

第三节 Nios II eclipse 问题

这个工具的 Bug 很多，吐槽也很多，但是只要能静下心来，还是可以调节明白的，操作在之前说明了，这里主要记录几个小问题：

如果你的问题没有在这个里面，那么请重新编译工程，如果还不行，那么重新建立一个新的工程。

（1） FPGA 初始化有问题：

现象：在主函数中写了 printf("HELLO");都没有在屏幕上显示

解决办法：初步怀疑是 FPGA 某些模块没有初始化，保留了之前的结果。可以通过稍微改变 Qsys/PlatformDesigner 中 ROM/RAM 大小，编译后重新写入 sof 文件来解决。

（2） C/C++调用问题

现象：有些头文件明明在当前目录下，但是当你#include 时，却报错了

解决办法：修改其中的 makefile，

```
39 APP_INCLUDE_DIRS :=
30 APP_LIBRARY_DIRS :=
31 APP_LIBRARY_NAMES :=
```

在这里加入相应的目录，当前目录通过“./”来表示。之后编译工程，解决问题。

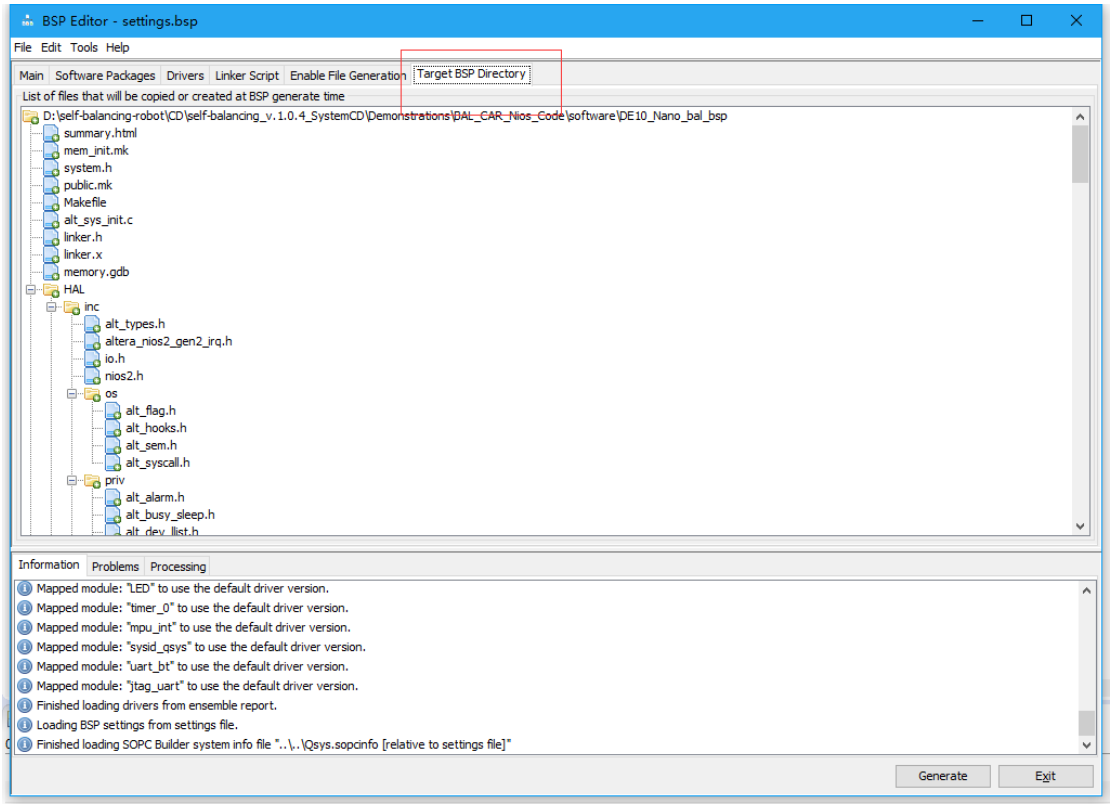
这里声明一点，如果是工程编译问题的话，最好了解 makefile 文件的操作再进行修改，不要直接改这个，多数情况下，不需要修改。

（3） 系统调用

现象：有些系统和软件中自有的头文件，比如之前说过的 system.h 和中断用的 io.h 也会出现报错的现象。

解决办法：重新启动工程或者重新编译，可以自动生成这些文件。

如果想看哪部分头文件是自带的，可以通过以下办法：



在 BSP editor 中的 target BSP Directory 下可以看到工程中有的头文件，引用时，直接引用即可，对于一些诸如 HAL 文件夹下面还包含了文件夹的头文件，

```
#include "priv/alt_legacy_irq.h"
```

需要指明这个文件的路径，但无需写出 HAL 这一层。

（4） 其他问题可以百度或者尝试重新新建工程解决！