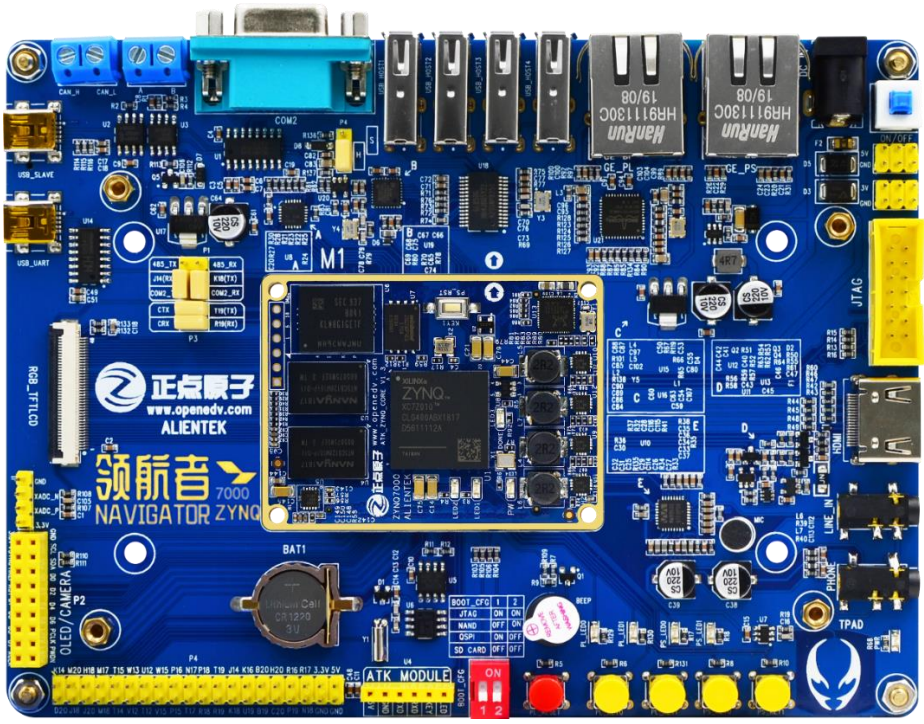
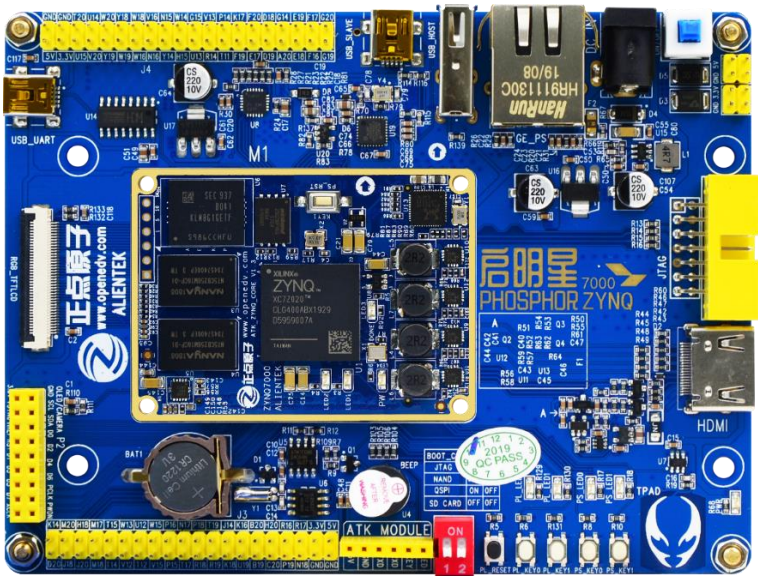
ZYNQ 双核 AMP

通信示例 V1.0

-正点原子 领航者&启明星 开发板教程





正点原子广州市星翼电子科技有限公司

淘宝店铺 1：http://eboard.taobao.com

淘宝店铺 2：http://openedv.taobao.com

技术支持论坛 (开源电子网) ：www.openedv.com

原子哥在线教学：www.yuanzige.com

官方网站：www.alientek.com

最新资料下载链接：http://www.openedv.com/posts/list/13912.htm

E-mail: 389063473@qq.com **QQ:** 389063473

咨询电话：020-38271790

传真号码：020-36773971

团队：正点原子团队

正点原子，做最全面、最优秀的嵌入式开发平台软硬件供应商。

友 情 提 示

如果您想及时免费获取“正点原子”最新资料，敬请关注正点原子微信公众平台，我们将及时给您发布最新消息和重要资料。

关注方法：

(1)微信“扫一扫”，扫描右侧二维码，添加关注

(2)微信→添加朋友→公众号→输入“正点原子”→关注

(3)微信→添加朋友→输入“alientek_stm32”→关注

文档更细说明

版本	版本更新说明	负责人	校审	发布日期
V1.0	第一版、最初版本	正点原子	正点原子	2020.06.08

目录

第一章 ZYNQ 双核 AMP 通信简介

多核处理器从多核的结构上是否一致，分为两种基本架构：同构多核架构和异构多核架构。同构多核处理器是指系统中的处理器在结构上是相同的；而异构处理器是指系统中的处理器在结构上是不同的，这些处理器可以是通用处理器，也可以是解决某些特定应用的专用硬核。同构多核架构相比于异构多核架构，在硬件和软件设计上较为简单，通用性较高，但在某些特定应用场合下，如异构多核架构专用的硬件加速硬核，异构多核架构的性能会更高。

Xilinx 的 ZYNQ SOC 融合了这两种架构，ZYNQ SOC 芯片包含两个独立的 Cortex-A9 内核，这两个核在结构上是相同的，同时又包括了可编程的逻辑单元（PL），使得 ZYNQ 整体系统成为了一个异构多核系统，如同时使其具有较高的通用性和性能。

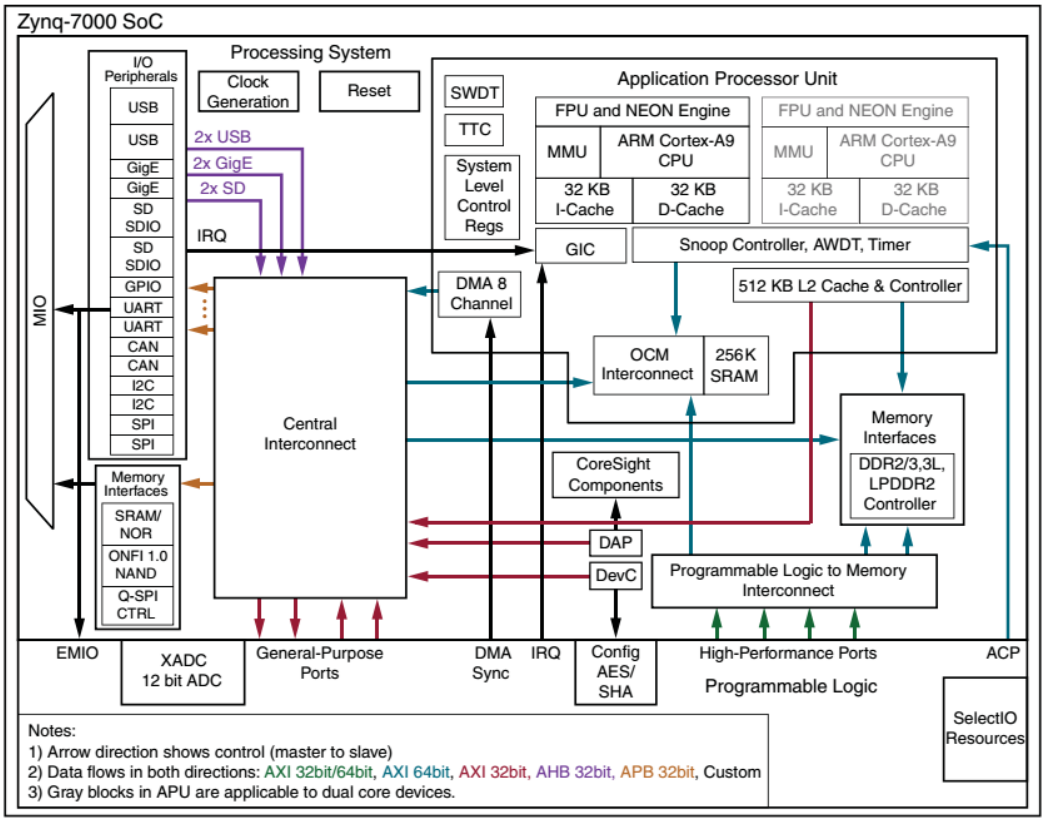


图 2.1.1 ZYNQ SoC 结构框图

从软件的角度看，多核处理器的运行模式有 AMP（非对称多处理）、SMP（对称多处理）和 BMP（受约束多处理）三种运行模式。

- AMP：该运行模式指多个内核相对独立的运行不同的任务，每个内核相互隔离，可以运行不同的操作系统（OS）或裸机应用程序。
- SMP：该运行模式指多个处理器运行一个操作系统，这个操作系统同等的管理多个内核，如 PC 电脑。
- BMP：该运行模式与 SMP 类似，但开发者可以指定将某个任务仅在某个指定核上执行。

一般来说，SMP 为较高级的应用提供统一的 OS 平台，开发者在 OS 之上构建应用时，无需考虑两个核之间的资源共享和进程间通信。除此之外，对 SMP 而言存在性能开销，这会对实时性要求较高的应用造成较大影响。如 PC 机电脑的多核处理器一般运行在 SMP 模式，实现的功能较为复杂，但对实时性的要求不高。

而 AMP 运行模式基本没有开销问题，在运行裸机应用程序时，甚至完全没有开销，比较适合实时性要求较高的应用，但需要考虑到处理器资源共享和处理器间通信等问题。如电力控制保护设备通常需要与人机接口实现复杂的通信和高实时性的计算能力，一般采用 AMP 运行模式，一个处理器运行 Linux 操作系统，另一个处理器运行裸机应用程序，从而兼顾了电力系统控制设备需要的复杂功能和实时性。

AMP 和 SMP 运行模式的框图如所示：

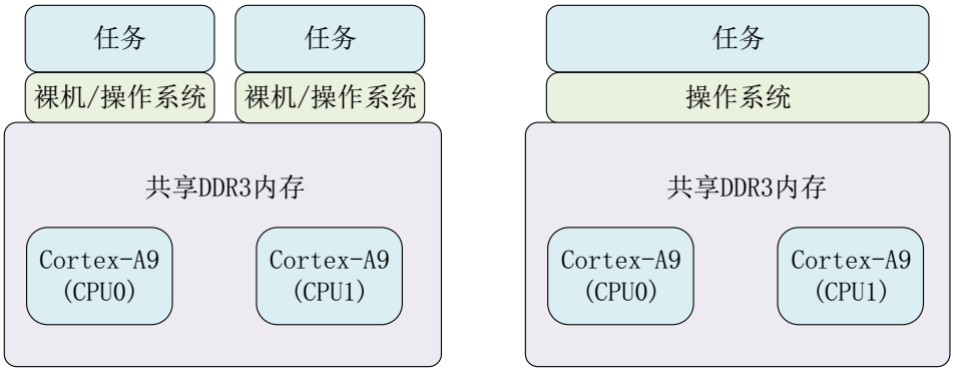


图 2.1.2 AMP 与 SMP 比较

在 AMP 运行模式下，虽然两个 Cortex-A 核可以独立运行，但往往大多数情况下，我们的软件设计要求两个核之间能够进行数据交互，也就是能够实现核间通信，以访问共享资源时避免冲突，如果没有考虑到这些问题，则可能会导致系统运行出现问题。所以本篇文档笔者将向大家介绍在 AMP 运行模式下，ZYNQ 双核之间如何进行通信以及双核通信测试。

双核通信可以使用哪些手段？

1、共享内存

我觉得对于大家来说最容易想到的应该是共享内存，我们可以设置一块系统内存区域，这块内存作为两个或多个核之间的共享数据区域，也就是说所有的核都可对该内存进行读写操作，当然这里需要对它们施加一定的限制，什么情况下可以去访问这片内存，什么情况下不能访问，这个东西只能是在具体的设计当中去实现了；其实共享内存是一个非常经典的数据交互的方式，并不限于双核（或多核）通信当中，比如在多线程、多进程环境下也非常合适，具体更多的示例就不给大家进行列举了。

2、SGI 中断

与硬件中断方式不同，SGI（软件生成中断（software generated interrupts）可以通过软件方式触发中断，可以中断自身、可以中断另一个 CPU（核）或多个 CPU，例如将某一个 SGI 中断信号绑定到 cpu1，并且绑定了相应的中断处理函数，当 cpu0 需要与 cpu1 进行通信时就可以触发这个 SGI 中断信号，这样就会执行到 cpu1 的中断处理函数；事实上，SGI 中断本身就是为了核间通信而设计的，所以它是最适合用于核间通信的一种方法，但往往多数情况下只使用 SGI 中断是很难实现比较复杂的核间通信的，还得需要共享内存的配合。

4、其他方式

其它的方式这里就不给大家列举了！

第二章 双裸机通信测试

本章我们将进行双裸机 AMP 通信测试，也就是让 ZYNQ 两个 Cortex-A9 核独立运行不同的 SDK 裸机程序的情况下进行双核通信测试，这也是最简单的一种双核通信测试环境，本章主要包含如下几个小节：

1.双核（多核）同时工作所需的注意事项；

2.ZYNQ CPU1 启动方式

3.SDK 软件设计

4.运行测试

2.1 双核同时工作所需的注意事项

在 AMP 运行环境下，必须要小心以防止两个 CPU 争夺这些共享资源，在 SoC 硬件系统当中，有一些资源是每个 CPU 私有的，而有一些资源则是公用的；一些 CPU 私有资源如下所示：

- L1 cache（一级缓存）；
- CPU 私有外设中断（PPI）；
- 内存管理单元（MMU）；
- CPU 私有定时器。

CPU 共享资源如下所示：

- 中断控制分配器（ICD）；
- DDR 内存；
- 片内存储器（OCM）；
- 全局定时器（Global Timer）；
- SUC 和 L2 cache（二级缓存）。
- 片上外设，例如 GPIO、SD、QSPI、UART、I2C、SPI（共享外设中断）、USB、CAN、SPI 等等。

私有资源可以不用管，但是对于双核公共资源来说，必须要小心处理，避免两个 CPU 同时使用这些资源的情况下导致混乱、数据错误，例如同时控制某一个 GPIO、同时使用串口等。

2.2 ZYNQ CPU1 启动方式

对于 ZYNQ 来说，当硬件上电启动之后，BootROM 是第一个运行的代码（BootROM 代码被固化在 ZYNQ SoC 片内 RoM 存储器中），并且运行于 CPU0 上；BootROM 代码会从启动介质中读取 BotROM 头信息，BootROM 头信息就存储在 BOOT.BIN 文件中，BootROM 头信息记录了 bitstream、用户代码等数据存放在 BOOT.BIN 文件中的位置偏移量以及对应在内存中加载地址，根据这些信息来加载 PL 端 bitstream 流文件以及在 PS 端启动用户代码。

BootROM 启动之后会使 CPU1 进入等待事件模式（WFE），没有对其启用任何功能，也就是说此时 CPU1 处于一个等待指令的状态；而这个指令由 CPU0 给它，说白了就是必须由 CPU0 来启动 CPU1，解除它的等待指令状态，使其运行我们的用户代码。

CPU0 在 CPU1 上启动应用程序需要少量协议，当 CPU1 接收到 CPU0 给到的 SEV 指令后，它会被唤醒，然后立即读取 0xFFFFFFF0 这个内存地址中存放的数据，并把这个读取到的数据作为应用程序的入口地址，接着跳转到该地址启动应用程序。所以由此可知，如果在执行 SEV 指令的时候，地址 0xFFFFFFF0 中存放的是一个无效的应用程序入口地址，则结果是不可预测的。CPU0 启动 CPU1 的示例代码如下所示：

示例代码 2.2.1 CPU0 启动 CPU1 示例代码

```
#define sev() __asm__ __volatile__ ("sev")        // C 语言内嵌汇编写法
#define    CPU1_RUN_ADDR          0x30000000
#define    CPU1_COPY_ADDR         0xFFFFFFF0

static void StartCpu1(void)
{
    Xil_Out32(CPU1_COPY_ADDR, CPU1_RUN_ADDR);   // 将 0x30000000 拷贝到 0xFFFFFFF0 地址处
    dmb();    // 等待内存写入完成（同步）
    sev();    // 执行 sev 指令唤醒 CPU1
}
```

```
/* CPU0 应用程序的 main 函数 */
int main(void)
{
......
    StartCpu1();        // 启动 CPU1
......
    return 0;
}
```

关于 ZYNQ CPU1 的启动方式就先讲到这里，如果大家对此还有什么不明白的地方，可以参考 xilinx 官方参考手册 UG585。

2.3 SDK 软件设计

为了节约搭建 vivado 工程的时间，笔者直接使用了开发板出厂时烧录的系统镜像所对应的 vivado 工程，在我们的开发板资料包中都已经给大家提供了，路径为**[启明星|领航者]ZYNQ 开发板资料盘(A盘)\4_SourceCode\ZYNQ_7020\3_Embedded_Linux\vivado_pro\Navigator_7020.zip**，大家根据自己的开发板型号选择。

笔者以领航者 7020 为例，将 Navigator_7020.zip 在 Windows 下解压开来，解压之后得到 vivado 工程，然后使用 vivado 软件将其打开，打开之后如下所示：

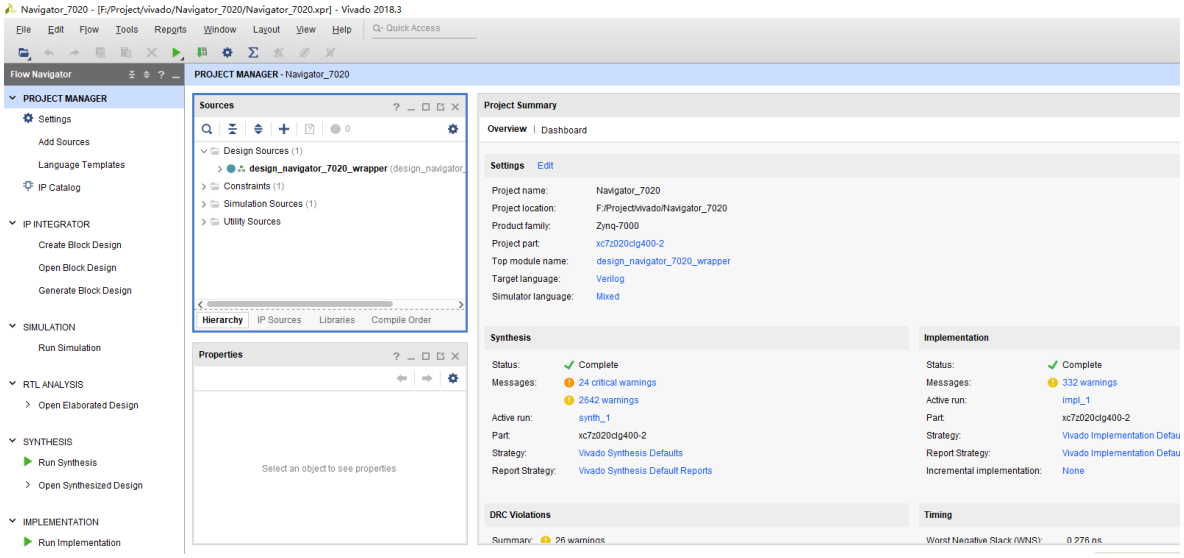


图 2.3.1 打开 vivado 工程

接下来点击 File-->Launch SDK 打开 SDK 软件，如下：

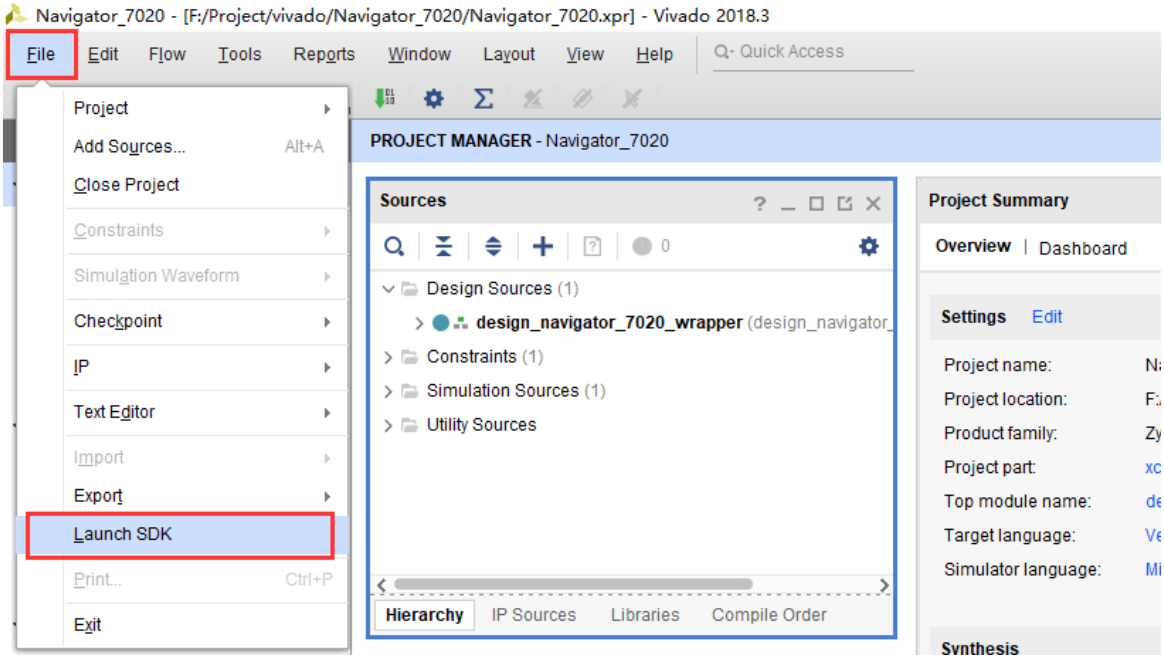


图 2.3.2 打开 SDK 软件

在弹出来的界面中直接点击 OK。

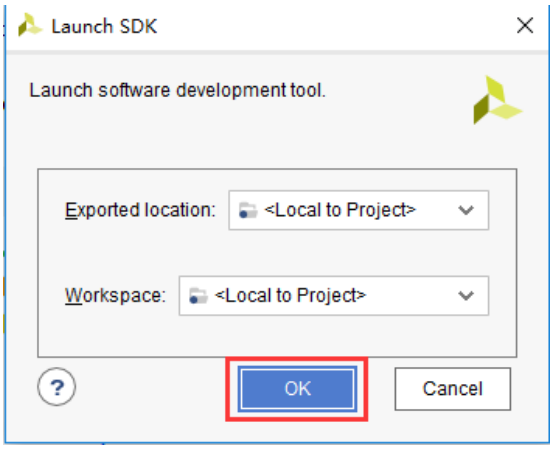


图 2.3.3 点击 OK

SDK 打开之后，接下来我们需要新建两个工程，分别为 CPU0 对应的裸机应用程序和 CPU1 对应的裸机应用程序。

2.3.1 CPU0 应用程序设计

1、创建 CPU0 工程

点击 File-->New-->Application Project 新建应用工程。

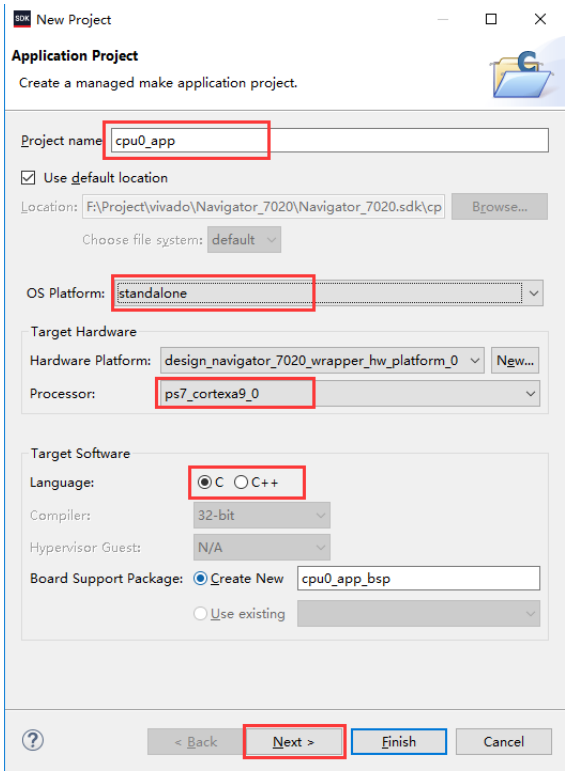


图 2.3.4 工程配置界面

注意 Processor 栏选择“ps7_cortexa9_0”，点击 Next 进入到下一步。

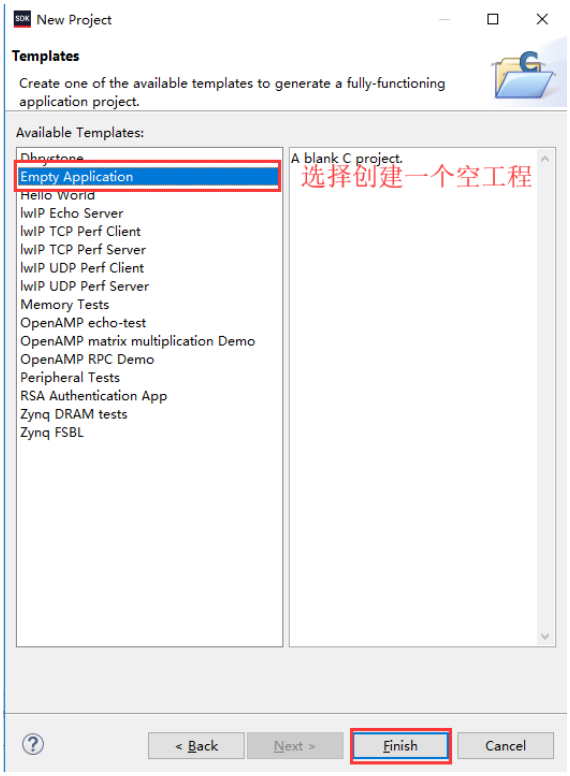


图 2.3.5 创建空工程

工程创建完成之后如下所示：

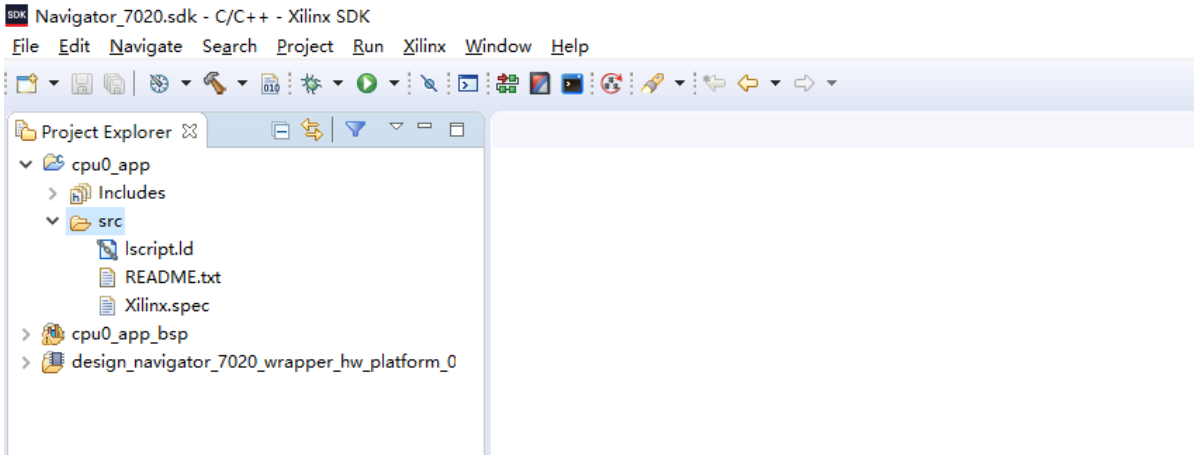


图 2.3.6 工程创建完成

2、配置应用程序内存空间的基地址和大小

我们需要为 CPU0 应用程序和 CPU1 应用程序分配各自运行所需的内存空间，注意它们使用的内存空间不能有重叠的情况，必须是独立的内存空间；双击 cpu0_app 工程的链接脚本文件 lscript.ld，这里笔者将 CPU0 应用程序内存空间的基地址设置为 0x100000、大小设置为 0x4000000，如下所示：

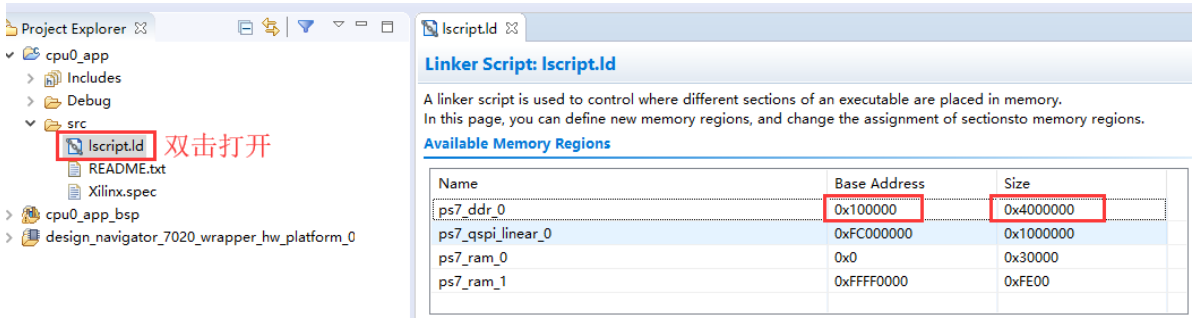


图 2.3.7 配置 CPU0 应用程序内存空间基地址和大小

修改完成之后按住“Ctrl+S”保存文件即可！

3、编写 CPU0 应用程序

接下来右键 src 文件夹，在弹出来的列表中选择 New-->Source File 新建一个名为 main.c 的源文件。

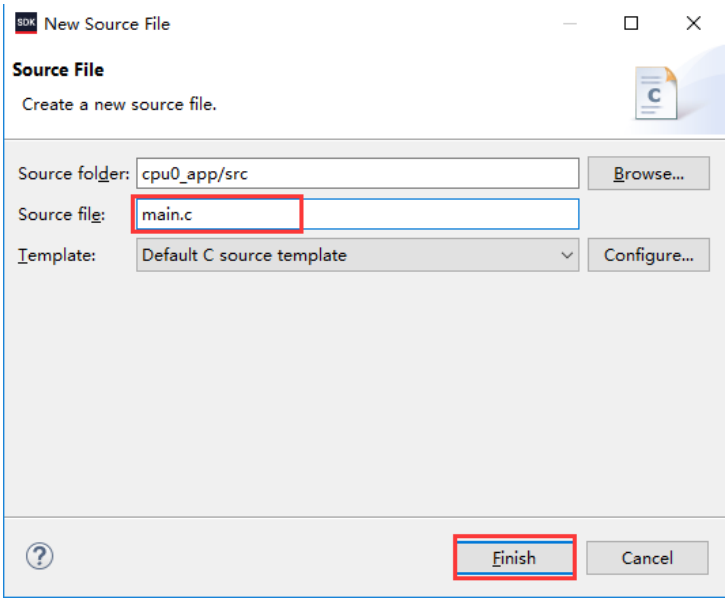


图 2.3.8 新建 main.c 源文件

点击 Finish 按钮确认。创建完成之后就可以在 main.c 文件中编写代码了，在开始编写代码之前，我们需要为两个 CPU 分配不同的任务、设计不同的角色，如下所示：

- ✓ CPU0 作为命令发送者，CPU1 作为命令接收者并执行相应的命令；
- ✓ 如果 CPU1 命令执行完毕之后将会通知 CPU0；
- ✓ 在本程序设计中我们会使用串口打印出信息，CPU0 和 CPU1 都会使用同一个串口，但为了避免同时使用串口，我们需要采取一定的措施来避免发生这种情况。

CPU0 应用程序接收来自串口的输入信息，比如输入不同的数字 1、2、3、4 等，不同的数字表示不同的频率等级，这个频率是 LED 的闪烁频率，LED 由 CPU1 应用程序控制，CPU1 接收来自 CPU0 的命令对 LED 频率进行调整；整个程序的大体设计思想就是这样，接下来我们开始编写 CPU0 应用程序代码，在 main.c 文件中编写代码，如下所示：

示例代码 2.3.1 cpu0 应用程序代码

```
#include <stdio.h>
#include <xil_printf.h>
#include <xscugic.h>
#include <sleep.h>
#include <xil_mmu.h>

#define sev()          __asm__ ("sev")    // C 语言内嵌汇编写法
#define CPU1_RUN_ADDR              0x10000000U
#define CPU1_COPY_ADDR             0xFFFFFFF0U

#define CPU0_SGI_ID                14          // 软中断号 14，范围为 0~15
#define CPU1_SGI_ID                15          // 软中断号 15，范围为 0~15
#define CPU0_ID                    0x1         // CPU0 的 id 号
#define CPU1_ID                    0x2         // CPU1 的 id 号
#define SHARE_MEM_ADDR     0x15000000U

```

```
17 static int work_finish_flag = 1;
18
19 static void StartCpu1(void)
20 {
21         Xil_Out32(CPU1_COPY_ADDR, CPU1_RUN_ADDR);
22         dmb();    // 等待内存写入完成（同步）
23         sev();    // 执行 sev 指令唤醒 CPU1
24 }
25
26 static void Cpu0SoftIntrHandler(void *data)
27 {
28         xil_printf("CPU0: Received SGI interrupt signal from CPU1\r\n");
29         xil_printf("\r\n");
30         work_finish_flag = 1;
31 }
32
33 int main (void)
34 {
35         XScuGic_Config *gic_cfg;
36         XScuGic gic;
37         int freq_level;
38         int ret;
39
40       /*
41         * 禁止共享内存 Cache，保持数据的一致性
42         * S=b1 TEX=b100 AP=b11, Domain=b1111, C=b0, B=b0
43         */
44         Xil_SetTlbAttributes(SHARE_MEM_ADDR, 0x14de2);
45
46       /*
47         * 禁止 0xFFFFFFF0 地址 Cache 属性
48         * S=b1 TEX=b100 AP=b11, Domain=b1111, C=b0, B=b0
49         */
50         Xil_SetTlbAttributes(CPU1_COPY_ADDR, 0x14de2);
51
52         /* 启动 CPU1 */
53         StartCpu1();
54
55         /* GIC 初始化 */
56         gic_cfg = XScuGic_LookupConfig(XPAR_SCUGIC_0_DEVICE_ID);
57         if (NULL == gic_cfg)
58               return XST_FAILURE;
```

```
59
60      ret = XScuGic_CfgInitialize(&gic, gic_cfg, gic_cfg->CpuBaseAddress);
61      if (XST_SUCCESS != ret)
62          return ret;
63
64      ret = XScuGic_Connect(&gic, CPU0_SGI_ID, Cpu0SoftIntrHandler, NULL);
65      if (XST_SUCCESS != ret)
66          return ret;
67
68      XScuGic_Enable(&gic, CPU0_SGI_ID);
69
70      /* IRQ 异常使能  */
71      Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
72                      (Xil_ExceptionHandler)XScuGic_InterruptHandler, &gic);
73      Xil_ExceptionEnable();
74
75      for ( ; ; ) {
76
77          if (work_finish_flag) {
78              xil_printf("CPU0: Please enter the number to "
79                          "change the LED blinking frequency:\r\n");
80              scanf("%d",&freq_level);
81
82              if (1 <= freq_level && 5 >= freq_level) {
83                  xil_printf("CPU0: You input number is %d\r\n", freq_level);
84                  Xil_Out32(SHARE_MEM_ADDR, freq_level);
85                  XScuGic_SoftwareIntr(&gic, CPU1_SGI_ID, CPU1_ID);
86                  work_finish_flag = 0;
87              }
88              else {
89                  xil_printf("CPU0: Error, The number range is 1~5\r\n");
90                  xil_printf("\r\n");
91              }
92          }
93
94          usleep(10 * 1000);
95      }
96
97      return 0;
98 }
```

第 7 行，宏定义 sev()，宏定义对应的内容为“__asm__ ("sev")”，这是 C 语言内嵌汇编代码的一种写法，关于 C 语言内嵌汇编的内容这里就不给大家做过多的介绍，想了解的朋友可以自己百度，都是很简单

地知识点；双引号当中的内容（sev）表示 ARM 的汇编指令 sev，前面给大家讲过，CPU0 可以通过 sev 指令来唤醒 CPU1。

第 8~9 行，定义了两个宏 CPU1_RUN_ADDR 和 CPU1_COPY_ADDR，CPU1_RUN_ADDR 用来表示 CPU1 应用程序的入口地址，这里对应的是 0x10000000，所以后面配置 CPU1 应用程序入口地址的时候也必须将其配置为 0x10000000；CPU1_COPY_ADDR 对应的是 0xFFFFFFF0，这个地址的意义前面给大家介绍过了，所以这里不再讲。

第 11~14 行，定义了 4 个宏 CPU0_SGI_ID、CPU1_SGI_ID、CPU0_ID、CPU1_ID。本程序设计当中，CPU0 和 CPU1 之间通过 SGI 软中断通知对方，软中断一共有 16 个，分别为 0~15；这里将 14 号软中断绑定到 CPU0，将 15 号中断绑定到 CPU1。CPU0_ID 和 CPU1_ID 分别对应 CPU0 和 CPU1 的 id 编号，这是硬件决定的，不可以修改！

第 15 行，宏定义 SHARE_MEM_ADDR 用于表示 CPU0 和 CPU1 之间的共享内存区，CPU0 将读取用户传入的参数（LED 闪烁频率等级），并将其写入到共享内存地址中，CPU1 再去读取共享内存地址得到该参数，并以此参数来设置 LED 的频率；这里笔者将 0x15000000 内存地址作为两个核的共享内存，该地址既不在 CPU0 应用程序地址空间，也不在 CPU1 应用程序地址空间中。

第 19~24 行，自定义函数 StartCpu1，用于启动 CPU1，步骤很简单，先将 CPU1 应用程序的入口地址写入到 0xFFFFFFF0 内存地址中，调用 dmb 指令进行同步确定数据写入到了 0xFFFFFFF0 地址中，然后调用 sev 唤醒 CPU1 即可！

第 26~31 行，CPU0 对应的软中断处理函数 Cpu0SoftIntrHandler，在本设计中，当 CPU1 完成工作之后会通过触发软中断通知 CPU0，所以当触发中断的时候会执行到 Cpu0SoftIntrHandler 函数。

接下来便是 main 函数中的内容，第 44 行，调用 Xil_SetTlbAttributes 函数来禁止共享内存的 Cache 缓存，以维护两个 CPU 访问该内存地址的数据一致性；第一个参数表示共享内存的基地址，第二个参数表示属性值，上面的注释当中说明了这些数字的由来，由于这个函数涉及到了比较底层的操作，例如 MMU、Cache、协处理器等等，就不给大家讲了，这东西讲起来又是一本书；初学者就不用去管了，知道有这么回事就行了。

第 50 行，同样调用了 Xil_SetTlbAttributes 函数来禁止 0xFFFFFFF0 地址的 Cache 属性，因为该地址存放了 CPU1 应用程序的入口地址，在启动 CPU1 的过程中也属于 CPU0 和 CPU1 的共享内存，大家理解吧。

第 53 行，调用 StartCpu1 函数启动 CPU1。

第 55~68 行，ARM GIC 模块初始化相关代码，第 64 行，调用 XScuGic_Connect 函数注册 CPU0 软中断（14）对应的中断服务函数 Cpu0SoftIntrHandler；第 68 行，使能该软中断。

第 70~73 行，使能 CPU0 的 IRQ 异常。

第 75~95 行，一个 for 死循环，在循环当中，通过 scanf 函数读取从串口终端接收到的用户数据，将接收到的数据写入到共享内存中，然后再通过 XScuGic_SoftwareIntr 函数触发 CPU1 对应的软中断（15）。

代码编写完成，按“Ctrl+S”保存并进行编译。

2.3.2 CPU1 应用程序设计

1、创建 CPU1 工程

点击 File-->New-->Application Project 新建应用工程。

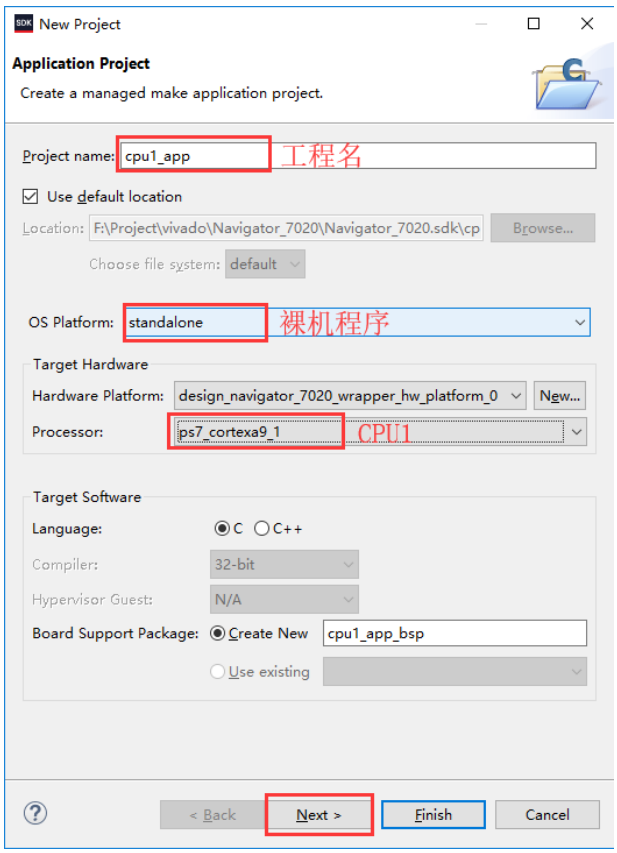


图 2.3.9 工程配置界面

注意在 Processor 栏选择“ps7_cortexa9_1”，点击 Next 按钮进入工程模板选择界面，同样这里我们也是创建一个空工程，与图 2.3.5 中选择一样，工程创建完成之后，如下所示：

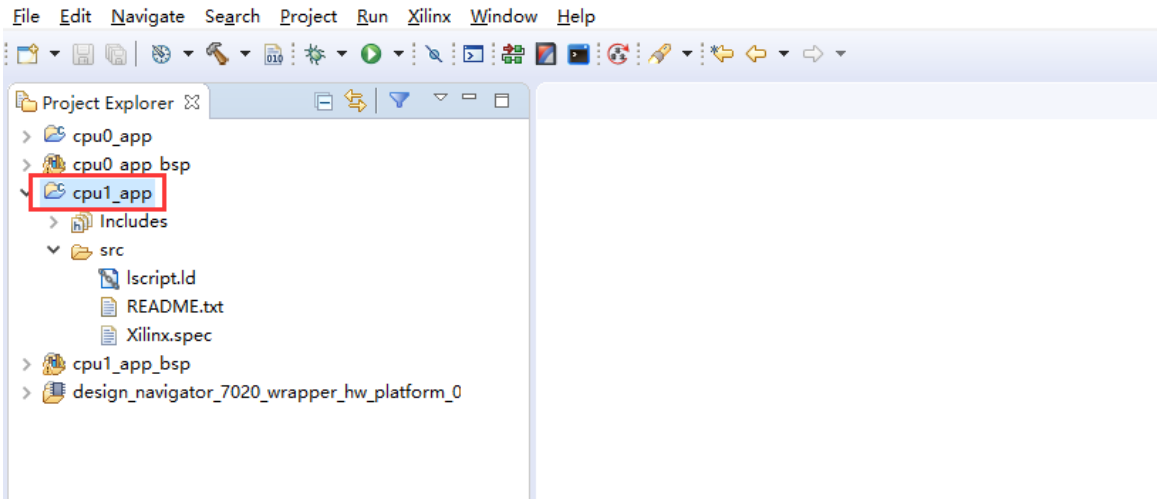


图 2.3.10 工程创建完成

2、配置应用程序内存空间的基地址和大小

首先双击 CPU1 工程的链接脚本文件 lscript.ld 打开它，修改 CPU1 应用程序内存空间的基地址和大小，这里笔者将基地址设置为 0x10000000。大小设置为 0x4000000，如下所示：

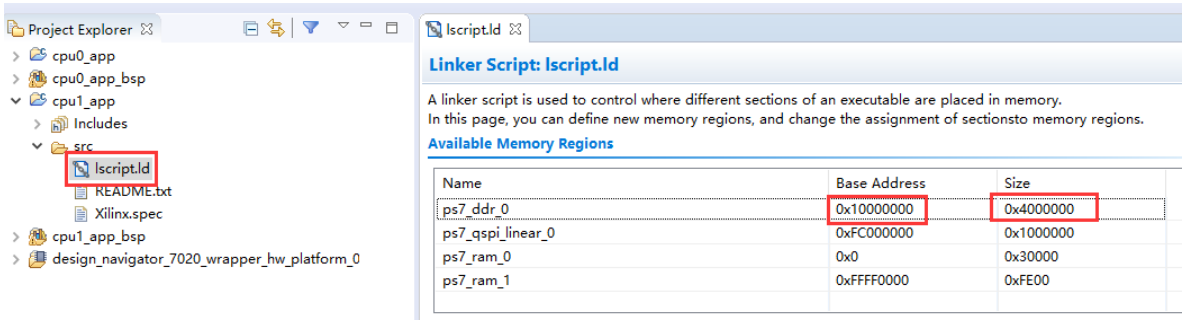


图 2.3.11 配置 CPU1 应用程序内存空间的基地址和大小

3、添加 USE_AMP 宏定义

此外我们还需要对 CPU1 工程的板级支持包做额外的设置，如下所示：

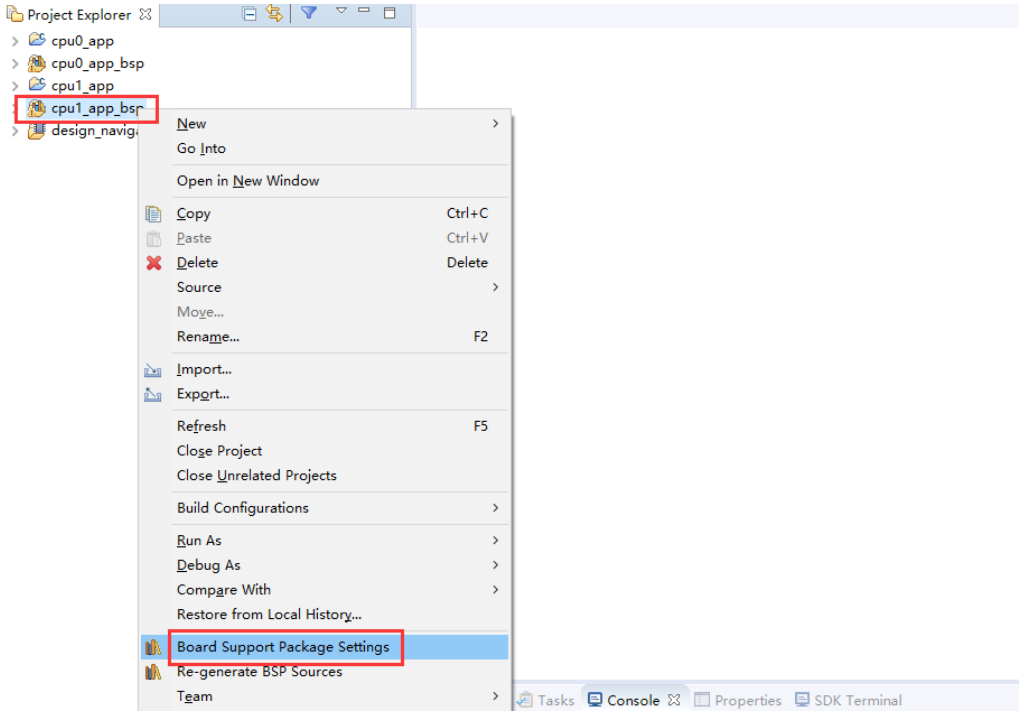


图 2.3.12 CPU1 工程板级支持包设置

在配置界面当中，选择 drivers-->ps7_cortexa9_1，在 extra_compiler_flags 栏添加“-DUSE_AMP=1”，如下所示：

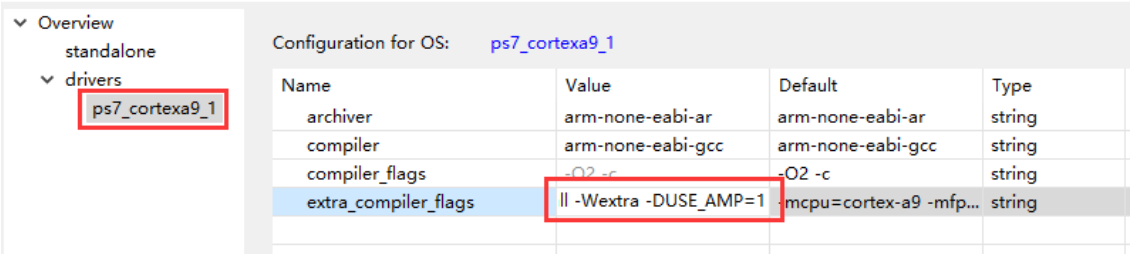


图 2.3.13 添加-DUSE_AMP=1

该页面用于配置工程的交叉编译工具，例如 compiler 指定了编译工程源码时所使用的交叉编译工具 arm-none-eabi-gcc，而 compiler_flags 和 extra_compiler_flags 则是执行 arm-none-eabi-gcc 命令时所携带的选项，“-D”选项的作用则是定义一个宏，所以我们这里添加“-DUSE_AMP=1”其实就类似于定义了一个宏“#define

USE_AMP 1”，这里定义的宏与普通.c、.h 文件中定义宏有所不同，这个宏在整个工程源码中都是有效的，你不需要包含任何的头文件都能使用它。

那问题来了？我们为什么需要定义这个宏，主要是跟 L2 Cache 有关系，因为 L2 Cache 属于 CPU0 和 CPU1 的共享资源，都可以使用，这对于 AMP 运行模式来说是一个麻烦，所以这里直接让 CPU1 禁止使用 L2 Cache，避免出现问题。当定义了这个宏之后，CPU1 就不可以使用 L2 Cache 了，大家可以在 SDK 中搜索下“USE_AMP”就知道了，例如：

示例代码 2.3.2 boot.S 汇编文件某段代码

```
/* Invalidate L2 Cache and enable L2 Cache*/
/* For AMP, assume running on CPU1. Don't initialize L2 Cache (up to Linux) */
#if USE_AMP!=1
    ldr r0,=L2CCCrtl        /* Load L2CC base address base + control register */
    mov r1, #0              /* force the disable bit */
    str r1, [r0]            /* disable the L2 Caches */

    ldr r0,=L2CCAuxCrtl     /* Load L2CC base address base + Aux control register */
    ldr r1,[r0]             /* read the register */
    ldr r2,=L2CCAuxControl      /* set the default bits */
    orr r1,r1,r2
    str r1, [r0]                /* store the Aux Control Register */

    ldr r0,=L2CCTAGLatReg   /* Load L2CC base address base + TAG Latency address */
    ldr r1,=L2CCTAGLatency  /* set the latencies for the TAG*/
    str r1, [r0]            /* store the TAG Latency register Register */
......
#endif
```

再比如下面这段代码：

示例代码 2.3.3 xil_cache.c 文件中的某段代码

```
void Xil_DCacheEnable(void)
{
    Xil_L1DCacheEnable();
#ifndef USE_AMP
    Xil_L2CacheEnable();
#endif
}
```

从示例代码中可以知道，当定义了这个宏 L2 Cache 就不会被使能；关于“USE_AMP”宏就给大家讲到这里。

4、编写 CPU1 应用程序

接下来右键 cpu1_app 工程目录下的 src 文件夹，在弹出来的列表中选择 New-->Source File 新建一个名为 main.c 的源文件。

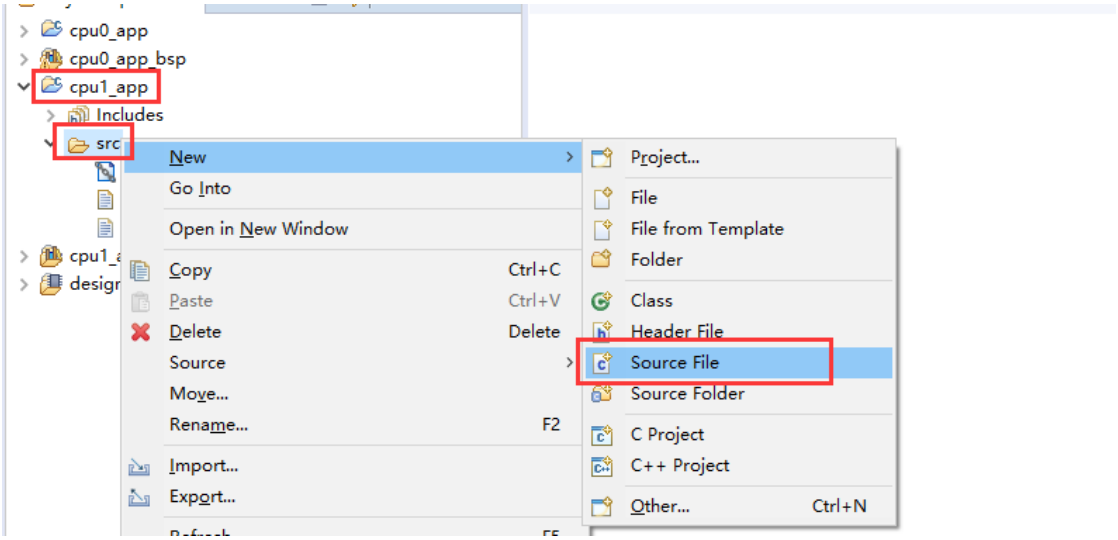


图 2.3.14 新建源文件

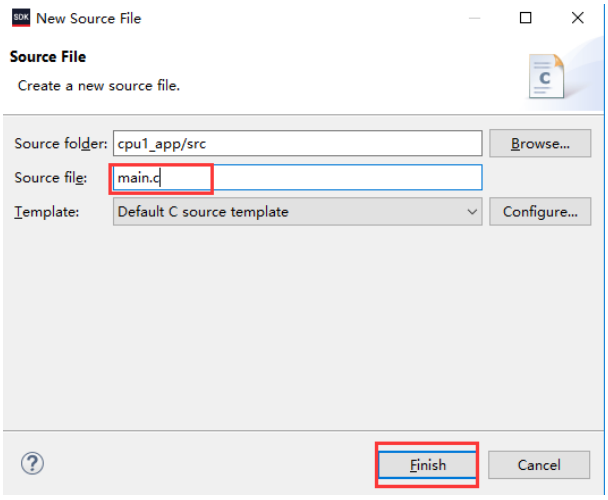


图 2.3.15 新建 main.c 源文件

接下来我们就可以 CPU1 工程的 main.c 文件中编写代码了，如下所示：

示例代码 2.3.4 cpu1 应用程序代码

```
#include <stdio.h>
#include <xil_printf.h>
#include <xscugic.h>
#include <sleep.h>
#include <xil_mmu.h>
#include <xgpiops.h>
#include <xscutimer.h>

#define CPU0_SGI_ID         14          // 软中断号 14，范围为 0~15
#define CPU1_SGI_ID         15          // 软中断号 15，范围为 0~15
#define CPU0_ID             0x1         // CPU0 的 id 号
#define CPU1_ID             0x2         // CPU1 的 id 号
#define SHARE_MEM_ADDR      0x15000000U // 共享内存地址
```

```
14
15 static XGpioPs gpio;
16 static XScuTimer timer;        // CPU 私有定时器
17
18 /* LED 不同闪烁频率对应的定时器计数值  */
19 enum Freq_Level {
20         LEVEL_1 = 0x16D933DFU,
21         LEVEL_2 = 0xB6C9944U,
22         LEVEL_3 = 0x491D72CU,
23         LEVEL_4 = 0x248EB96U,
24         LEVEL_5 = 0x12475CBU,
25 };
26
27 static void Cpu1SoftIntrHandler(void *data)
28 {
29         XScuGic *gic = (XScuGic *)data;
30         u32 level;
31         int val;
32
33         xil_printf("CPU1: Received SGI interrupt signal from CPU0\r\n");
34         val = Xil_In32(SHARE_MEM_ADDR);
35         xil_printf("CUP1: Received data is %d\r\n", val);
36
37         switch (val) {
38         case 1:
39                 level = LEVEL_1;
40                 break;
41
42         case 2:
43                 level = LEVEL_2;
44                 break;
45
46         case 3:
47                 level = LEVEL_3;
48                 break;
49
50         case 4:
51                 level = LEVEL_4;
52                 break;
53
54         case 5:
55                 level = LEVEL_5;
```

```
56              break;
57
58          default:
59              break;
60          }
61
62          XScuTimer_Stop(&timer);                 // 停止定时器
63          XScuTimer_LoadTimer(&timer, level);     // 加载计数值
64          XScuTimer_Start(&timer);                // 启动定时器
65
66          XScuGic_SoftwareIntr(gic, CPU0_SGI_ID, CPU0_ID);    // 触发 CPU0 对应的软中断
67  }
68
69  static void TimerIntrHandler(void *data)
70  {
71          static int val = 0;
72          XGpioPs_WritePin(&gpio, 0, val);
73          val = !val;
74          XScuTimer_ClearInterruptStatus(&timer); // 清除中断标志位
75  }
76
77  int main (void)
78  {
79          XScuGic_Config *gic_cfg;
80          XGpioPs_Config *gpio_cfg;
81          XScuTimer_Config *timer_cfg;
82          XScuGic gic;
83          int ret;
84
85         /*
86          * 禁止共享内存 Cache，保持数据的一致性
87          * S=b1 TEX=b100 AP=b11, Domain=b1111, C=b0, B=b0
88          */
89          Xil_SetTlbAttributes(SHARE_MEM_ADDR, 0x14de2);
90
91          /* 初始化 LED */
92          gpio_cfg = XGpioPs_LookupConfig(XPAR_XGPIOPS_0_DEVICE_ID);
93          if (NULL == gpio_cfg)
94              return XST_FAILURE;
95
96          ret = XGpioPs_CfgInitialize(&gpio, gpio_cfg, gpio_cfg->BaseAddr);
97          if (XST_SUCCESS != ret)
```

```
98          return ret;
99
100     XGpioPs_SetDirectionPin(&gpio, 0, 1);
101     XGpioPs_SetOutputEnablePin(&gpio, 0, 1);
102     XGpioPs_WritePin(&gpio, 0, 1);
103
104     /* GIC 初始化 */
105     gic_cfg = XScuGic_LookupConfig(XPAR_SCUGIC_0_DEVICE_ID);
106     if (NULL == gic_cfg)
107         return XST_FAILURE;
108
109     ret = XScuGic_CfgInitialize(&gic, gic_cfg, gic_cfg->CpuBaseAddress);
110     if (XST_SUCCESS != ret)
111         return ret;
112
113     ret = XScuGic_Connect(&gic, CPU1_SGI_ID, Cpu1SoftIntrHandler, &gic);
114     if (XST_SUCCESS != ret)
115         return ret;
116
117     XScuGic_Enable(&gic, CPU1_SGI_ID);
118
119     ret = XScuGic_Connect(&gic, XPAR_SCUTIMER_INTR, TimerIntrHandler, &timer);
120     if (XST_SUCCESS != ret)
121         return ret;
122
123     XScuGic_Enable(&gic, XPAR_SCUTIMER_INTR);
124
125     /* 初始化 CPU 私有定时器 */
126     timer_cfg = XScuTimer_LookupConfig(XPAR_XSCUTIMER_0_DEVICE_ID);
127     if (NULL == timer_cfg)
128         return XST_FAILURE;
129
130     ret = XScuTimer_CfgInitialize(&timer, timer_cfg, timer_cfg->BaseAddr);
131     if (XST_SUCCESS != ret)
132         return ret;
133
134     XScuTimer_EnableAutoReload(&timer);
135     XScuTimer_EnableInterrupt(&timer);
136     XScuTimer_Stop(&timer);
137
138     /* IRQ 异常使能 */
139     Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
```

```
140                     (Xil_ExceptionHandler)XScuGic_InterruptHandler, &gic);
141         Xil_ExceptionEnable();
142
143         for ( ; ; ) {
144
145             /* 死循环 */
146             usleep(20 * 1000);
147         }
148
149         return 0;
150 }
```

第 19~25 行，这里定义了一个枚举类型，用于表示 LED 五个不同的闪烁频率所对应的定时器计数值。

第 27~67 行，CPU1 对应的软中断处理函数 Cpu1SoftIntrHandler，在中断处理函数中，会读取 CPU0 写入到共享内存中的数据，并以此数据来调整 LED 的闪烁频率；在事情完成之后调用 XScuGic_SoftwareIntr 函数触发 CPU0 的软中断告知 CPU0“自己”已经把工作做完了。

第 69~75 行，CPU1 私有定时器的中断处理函数 TimerIntrHandler，在中断处理函数中，调用 XGpioPs_WritePin 周期性改变 LED 的亮灭状态实现 LED 闪烁，记得在后面添加调用 XScuTimer_ClearInterruptStatus 函数清除中断标志位。

第 77~150 行，CPU1 应用程序的 main 函数，第 89 行，同样调用 Xil_SetTlbAttributes 函数来禁止共享内存的 Cache 属性，保持数据一致性；第 92~102 行，GPIO 初始化相关的代码，因为程序中用到了 LED，对应于核心板上的 LED2；第 105~123 行，ARM GIC 模块相关初始化代码，第 113 行调用 XScuGic_Connect 函数注册 CPU1 软中断（15）对应的中断服务函数 Cpu1SoftIntrHandler，第 117 行调用 XScuGic_Enable 使能 CPU1 软中断；第 119 行注册 CPU 私有定时器中断服务函数并使能中断；第 126~136 行，CPU 私有定时器相关初始化代码。

第 143~147 行，for 死循环。

代码编写完成之后，按“Ctrl+S”保存并进行编译。

2.4 运行测试

我们已经将 CPU0 和 CPU1 所对应的应用程序代码编写完成并编译成功，接下就可以进行测试了！

2.4.1 JTAG 下载测试

首先我们将下载器与领航者（或启明星）底板上的 JTAG 接口相连，下载器另外一端与电脑连接。然后使用 Mini USB 连接线将 USB_UART 接口与电脑连接，用于串口通信；最后连接开发板的电源，将开发板启动方式设置为 JTAG 方式（通过底板的拨码开关进行配置）并打开电源开关。

右键 CPU0 工程文件夹 cpu0_app，选择 Run As-->Run Configurations。

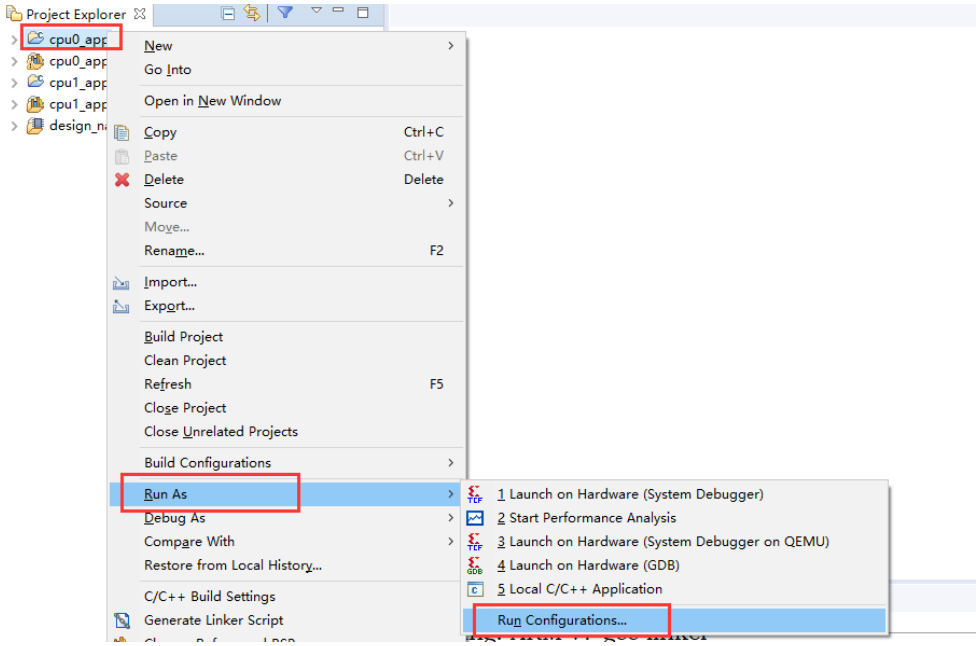


图 2.4.1 运行配置

在弹出的界面中双击 Xilinx C/C++ application（System Debug），对下载选项进行配置。在 Target Setup 配置页中选中“Reset entire system”，此时“Program FPGA”也会自动选中，如下所示：

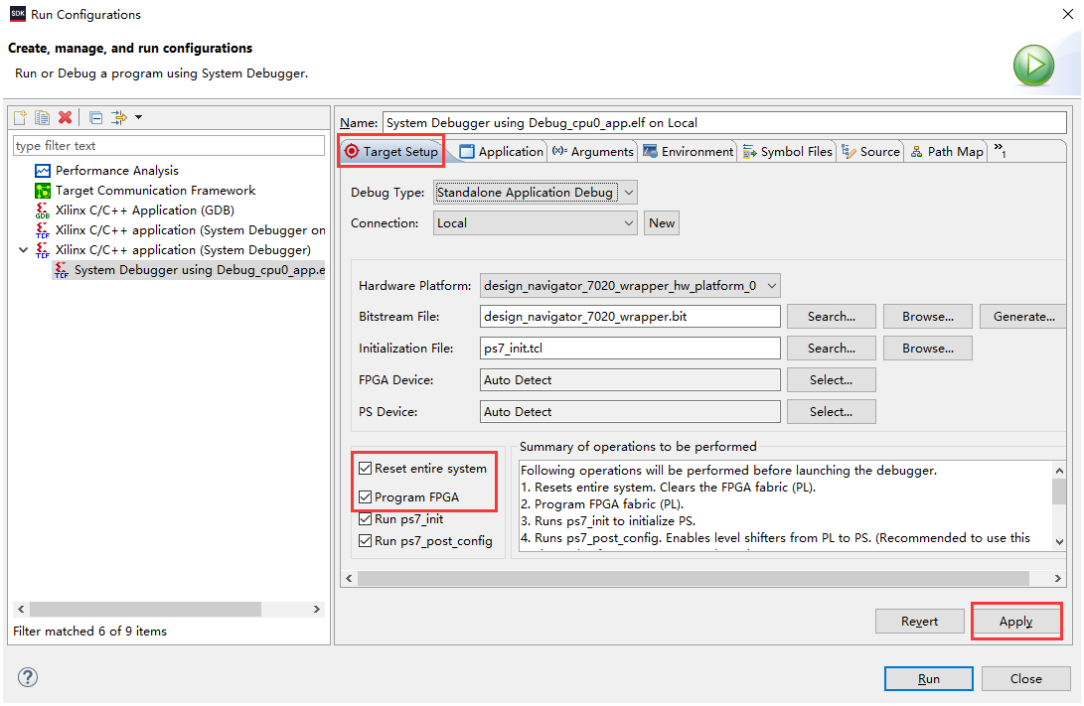


图 2.4.2 Target Setup 配置界面

将配置页面切换至“Application”，把“ps_cortexa9_1”也选中，如下图所示：

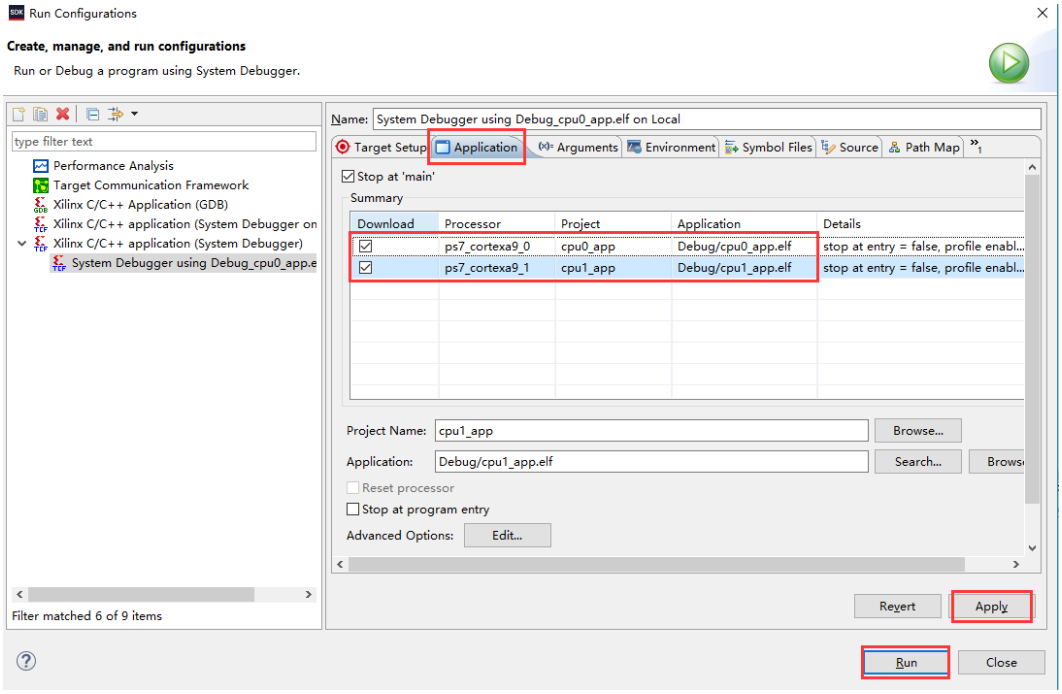


图 2.4.3 Application 配置界面

设置完成之后，点击“Apply”保存配置，之后点击“Run”按钮下载程序并运行，下载完成并运行之后我们会发现核心板上的 LED2 亮了。此时切换到串口终端，会出现打印信息，提示我们输入数字来改变 LED 灯的闪烁频率，如下所示：

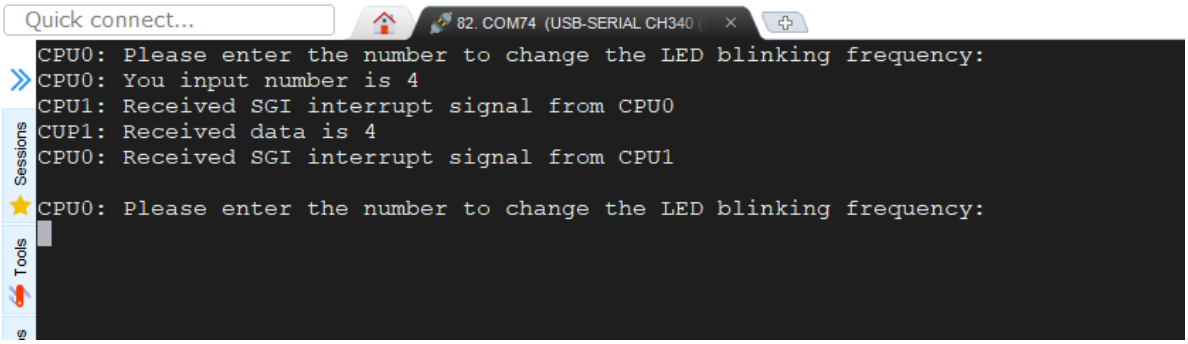


图 2.4.4 串口终端打印信息

图 2.4.4 中笔者输入了 4 按回车之后，LED 灯在闪烁，输入的数字越大闪烁频率越快，并打印这些信息，字符串前缀“CPU0:”和“CPU1:”表示该打印信息分别是由 CPU0 和 CPU1 打印出来的，首先我们输入数字之后按回车，CPU0 接收到了我们输入的数字并将其打印出来“CPU0: You input number is 4”，CPU1 接收到 CPU0 发送过来的软中断信号之后会打印出字符串“CPU1: Received SGI interrupt signal from CPU0”，此时 CPU1 读取共享内存中的数据并将其打印出来“CUP1: Received data is 4”，设置 LED 的闪烁频率；完成任务之后发送软中断信号通知 CPU0，CPU0 接收到中断信号之后会打印字符串“CPU0: Received SGI interrupt signal from CPU1”；这就是整个的一个工作流程。

2.4.2 制作 BOOT.BIN

上一小节中我们使用了 JTAG 下载的方式验证了双核 AMP 实验的运行情况，如果需要将可执行文件存放在 QSPI 或者是 SD 卡中启动运行，那么我们需要制作一个 BOOT.BIN 文件。

首先需要新建一个 FSBL 工程，然后创建 BOOT.BIN 文件，操作方法跟《领航者|启明星 ZYNQ 之嵌入式开发指南》中讲解的是一样的，只是在 Create Boot Image 时，在最后把 CPU1 应用程序对应的 elf 文件添加上去即可，如下所示：

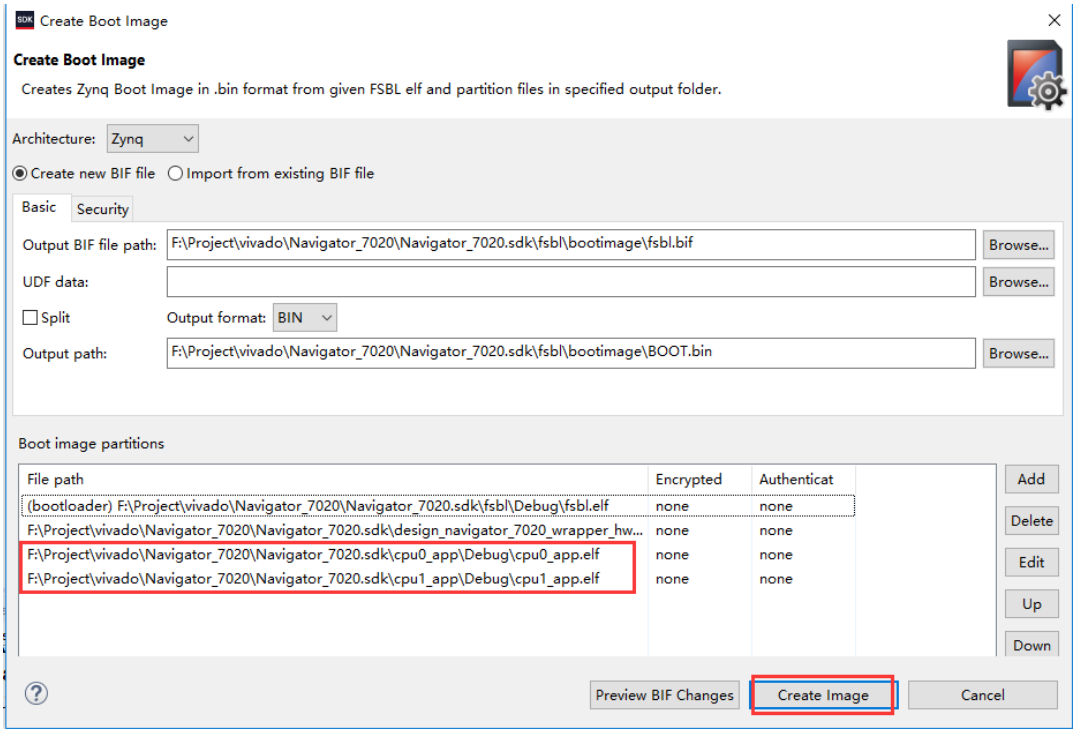


图 2.4.5 创建 Boot Image

这里需要注意，cpu0_app.elf 和 cpu1_app.elf 的顺序不能搞错了，必须将 CPU0 对应的 elf 文件放在前面，这个跟 FSBL 代码对 BOOT.BIN 的解析有关系，这里就不跟大家细说了，总之 FSBL 最终会执行第一个检索到的 elf 文件，有兴趣自己看看 FSBL 代码，很简单。

在本测试程序中，bitstream 文件可以不用放，我们根本没用到 PL 端的资源，最后点击“Create Image”按钮创建 BOOT.BIN 启动文件。这里就不给大家演示 SD 卡或 QSPI 启动测试了，正常情况下，代码运行的效果跟上一小节是一样的。

第三章 裸机和 Linux 通信测试

上一章中我们在双核都跑裸机应用程序的情况下进行了双核通信测试，那么本章我们将让 CPU0 跑 Linux 系统，而 CPU1 继续运行裸机应用程序，在这样的 AMP 运行环境下再来测试双核通信。

3.1 编写 CPU1 应用程序

在编写程序之前，我们需要先设计程序，为两个 CPU 分配不同的任务、设计不同的角色：

- CPU1 将数据写入到共享内存中，并触发软中断通知 CPU0；
- CPU0 接收到软中断信号之后读取共享内存中的数据，并打印出来，之后通过触发软中断通知 CPU1。

为了简化代码，我们在 CPU1 裸机程序中周期性给 CPU0 发送软中断信号，本程序代码我们可以直接在上一章 CPU1 工程的 main.c 文件中进行修改，如下所示：

示例代码 3.1.1 CPU1 应用程序代码

```
#include <xil_printf.h>
#include <xscugic.h>
#include <sleep.h>
#include <xil_mmu.h>

#define CPU0_SGI_ID             14          // 软中断号 14，范围为 0~15
#define CPU1_SGI_ID             15          // 软中断号 15，范围为 0~15
#define CPU0_ID                 0x1         // CPU0 的 id 号
#define CPU1_ID                 0x2         // CPU1 的 id 号
#define SHARE_MEM_ADDR      0x25000000U       // 共享内存地址

static void Cpu1SoftIntrHandler(void *data)
{
        xil_printf("CPU1: Received SGI interrupt signal from CPU0\r\n");
        xil_printf("\r\n");
}

int main (void)
{
        XScuGic_Config *gic_cfg;
        XScuGic gic;
        u32 count = 1;
        int ret;

        xil_printf("CPU1: Start successful!\r\n");

       /*
        * 禁止共享内存 Cache，保持数据的一致性
        * S=b1 TEX=b100 AP=b11, Domain=b1111, C=b0, B=b0
        */
        Xil_SetTlbAttributes(SHARE_MEM_ADDR, 0x14de2);

        /* GIC 初始化 */
        gic_cfg = XScuGic_LookupConfig(XPAR_SCUGIC_0_DEVICE_ID);
```

```
35      if (NULL == gic_cfg)
36          return XST_FAILURE;
37
38      ret = XScuGic_CfgInitialize(&gic, gic_cfg, gic_cfg->CpuBaseAddress);
39      if (XST_SUCCESS != ret)
40          return ret;
41
42      ret = XScuGic_Connect(&gic, CPU1_SGI_ID, Cpu1SoftIntrHandler, &gic);
43      if (XST_SUCCESS != ret)
44          return ret;
45
46      XScuGic_InterruptMaptoCpu(&gic, CPU1_ID, CPU1_SGI_ID);
47      XScuGic_Enable(&gic, CPU1_SGI_ID);
48
49      /* IRQ 异常使能 */
50      Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
51                                   (Xil_ExceptionHandler)XScuGic_InterruptHandler, &gic);
52      Xil_ExceptionEnable();
53
54      for ( ; ; ) {
55
56          /* 死循环 */
57          if (5 < count)
58              count = 0;
59
60          Xil_Out32(SHARE_MEM_ADDR, count);
61          XScuGic_SoftwareIntr(&gic, CPU0_SGI_ID, CPU0_ID);
62          count++;
63          sleep(2);
64      }
65
66      return 0;
67 }
```

在本次实验当中，我们将 CPU0 与 14 号软中断绑定在一起，将 CPU1 与 15 号软中断绑定在一起；当触发 14 号软中断的时候会由 CPU0 执行相应的中断处理函数，当触发 15 号软中断的时候会由 CPU1 执行相应的中断处理函数。

第 10 行，我们将共享内存地址设置为 0x25000000，同样这个地址既不在 CPU0 Linux 系统内存空间中、也不在 CPU1 裸机应用程序内存空间中。

第 54~64 行，在 for 死循环中，我们间隔 2 秒的时间周期性给 CPU0 Linux 系统发送软中断信号。

整个代码非常简单，该讲的东西前面都已经给大家讲过了，接下来我们需要调整下 CPU1 裸机应用程序的基地址和内存空间大小，双击 lscript.ld 链接脚本文件打开它，配置如下：

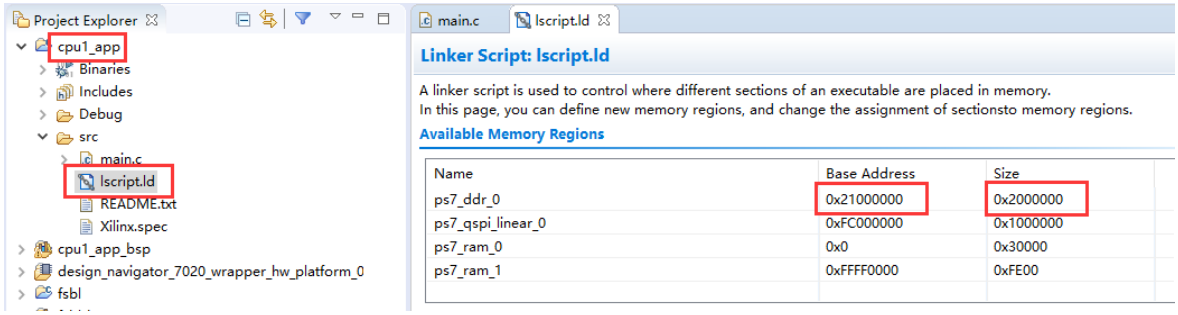


图 3.1.1 重新配置基地址和内存空间大小

笔者以 7020 核心板为例，7020 核心板 DDR 大小为 1GB，笔者将前 512MB 分配给 CPU0 Linux 系统，所以这里需要修改 CPU1 裸机应用程序内存空间的基地址和大小。如果是 7010 核心板，那就不能这样分配了，具体怎么分配自己根据需求去做，但是两者的内存空间不要发生重叠，必须要独立。

3.2 制作 BOOT.BIN 启动开发板

在上一章中，我们创建了一个 FSBL 工程，接下来直接利用这个已经创建好的 FSBL 工程来创建、生成 BOOT.BIN 启动文件。在生成 BOOT.BIN 启动文件之前，我们需要对 fsbl 工程的 main.c 源文件进行简单地添加与修改。

在上一章实验当中，我们在 CPU0 应用程序当中去启动 CPU1；同样在本章实验当中也是通过 CPU0 去启动 CPU1，因为 FSBL 代码运行在 CPU0 上，所以我们可以在 fsbl 中去启动 CPU1，双击打开 fsbl 工程中的 main.c 源文件，在 main 函数前添加如下代码：

示例代码 3.2.1 CPU1 启动相关代码

```
#define sev() __asm__ ("sev")                    // C 语言内嵌汇编写法
#define   CPU1_RUN_ADDR          0x21000000U
#define   CPU1_COPY_ADDR         0xFFFFFFF0U

static void StartCpu1(void)
{
    Xil_Out32(CPU1_COPY_ADDR, CPU1_RUN_ADDR);    // 将 0x10000000 拷贝到 0xFFFFFFF0 地址处
    dmb();          // 等待内存写入完成（同步）
    sev();          // 执行 sev 指令唤醒 CPU1
}
```

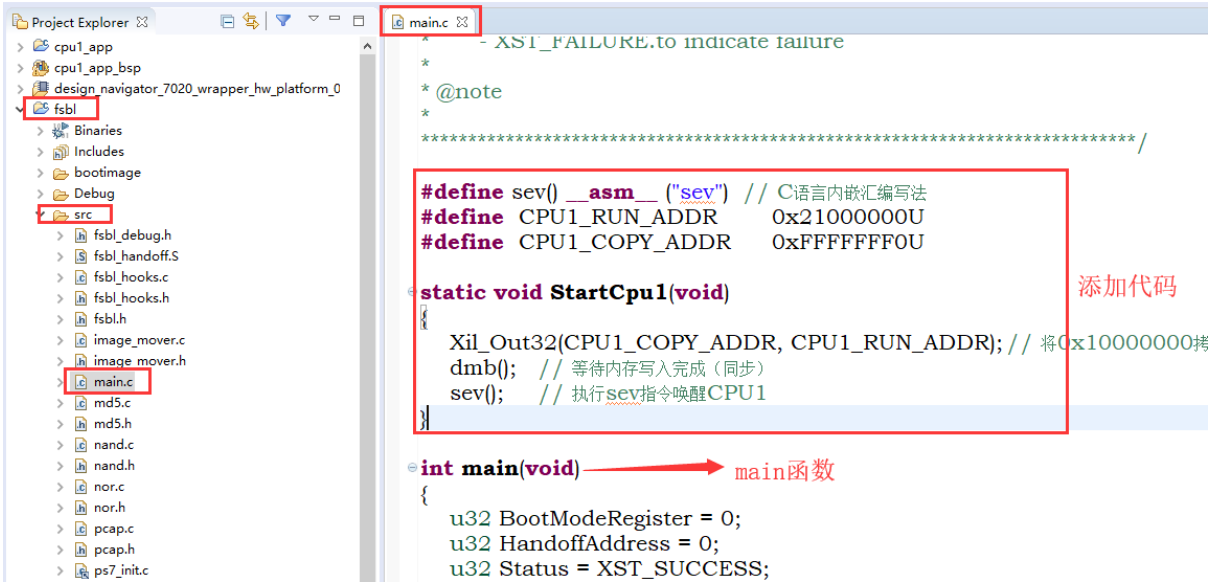


图 3.2.1 添加代码

添加完成之后，然后在“HandoffAddress = LoadBootImage()”代码后面调用上面定义的函数 StartCpu1，如下所示：

```
/*
 * Load boot image
 */
HandoffAddress = LoadBootImage();        在这条语句后面

/* 启动CPU1 */
StartCpu1();        调用我们定义的函数 启动CPU1

fsbl_printf(DEBUG_INFO,"Handoff Address: 0x%08lx\r\n",HandoffAddress);

/*
 * For Performance measurement
 */
```

图 3.2.2 调用 StartCpu1 函数

代码添加完成之后保存、编译 FSBL 工程得到 elf 格式的可执行文件 fsbl.elf。接下来我们需要制作 BOOT.BIN 文件，在制作 BOOT.BIN 启动文件之前，还得得到 u-boot 的镜像文件，将 u-boot 镜像文件和 fsbl 镜像文件合成为 BOOT.BIN 文件。

因为本章使用开发板出厂时对应的 vivado 工程、u-boot 源码以及 Linux 内核源码，笔者以 7020 开发板为例，上一小节说过，笔者需要将前 512MB 内存分配给 CPU0 所运行的 Linux 系统，所以这里需要修改 u-boot 对应的设备树文件，将分配给 Linux 系统的内存空间修改为 512MB，修改完成之后需要重新编译 u-boot 源码。

具体的方法大家参考文档《**领航者|启明星 ZYNQ 之 Linux 驱动开发指南**》中的《**附录 A3 编译出厂镜像章节**》，因为我们这里只需要修改 u-boot 设备树然后重新编译 u-boot 即可，所以大家只需要看 **A3.1.1** 小节和 **A3.2.2** 小节即可！

A3.2.2 小节介绍如何编译出厂的 u-boot 源码，在编译之前打开设备树文件，arch/arm/dts/atk-zynq-7020.dts（如果是 7010 核心板请选择 arch/arm/dts/atk-zynq-7010.dts），找到 memory 节点，如下所示：

```
        memory {
                device_type = "memory";
                reg = <0x0 0x40000000>;
        };
```

图 3.2.3 memory 节点

reg 属性描述了 u-boot 和 Linux 系统所能使用的内存空间，起始地址为 0x0，大小为 0x40000000，也就是 1GB；这里我们将 0x40000000 修改为 0x20000000，也就是该为 512MB，修改完成之后保存退出，然后编译 u-boot 源码得到 u-boot 镜像文件。

将得到的 u-boot 文件重命名为 u-boot.elf，然后将其拷贝到 Windows 系统的某个目录下，例如桌面：

图 3.2.4 u-boot.elf 文件

接下来制作 BOOT.BIN 文件，制作方法跟以前是一样的，在 SDK 软件中右键单击 FSBL 工程，选择“Create Boot Image”菜单，在弹出来的界面当中进行配置，如下所示：

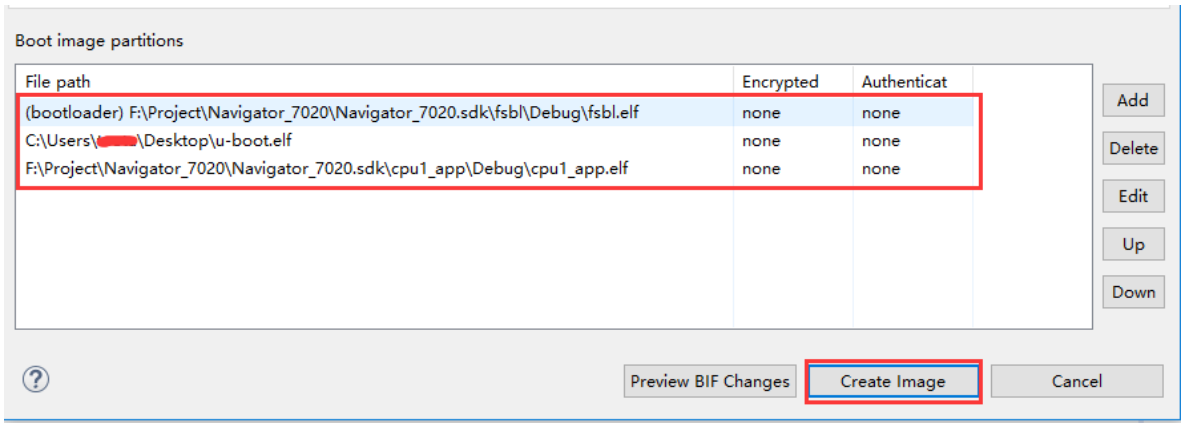


图 3.2.5 BOOT.BIN 配置

第一个文件是 ZYNQ fsbl 的 elf 格式文件 fsbl.elf。

第二个文件是 u-boot 的 elf 格式文件 u-boot.elf。

第三个文件是 CPU1 对应的应用程序的 elf 格式文件 cpu1_app.elf。

点击 Create Image 制作 BOOT.BIN 启动文件，此时我们可以使用 imageUSB 工具制作一张 SD 启动卡，然后将 SD 卡第一个分区中的 BOOT.BIN 文件替换成这里制作得到的 BOOT.BIN 文件即可！

3.3 启动开发板

BOOT.BIN 文件替换完成之后，将 SD 卡插入开发板启动，在启动内核之前，我们需要简单地修改 u-boot 的环境变量，目的是为了让 Linux 系统只能使用一个 CPU，也就是 CPU0，不要使用 CPU1，把 CPU1 独立出来运行一个裸机程序；进入 u-boot 命令行模式下，执行下面这条命令设置 bootargs 环境变量：

```
setenv bootargs 'console=ttyPS0,115200 maxcpus=1 earlyprintk root=/dev/mmcblk0p2 rw rootwait'
```

```
Zynq>
Zynq> setenv bootargs 'console=ttyPS0,115200 maxcpus=1 earlyprintk root=/dev/mmcblk0p2 rw rootwait'
Zynq>
Zynq> boot
```

图 3.3.1 设置 bootargs 环境变量

加入了 maxcpus=1 这条，这个其实就是告诉 Linux 系统，最多可以使用 1 个 CPU。设置完成之后，执行 boot 命令启动 Linux 系统。

3.4 编写驱动程序

接下来我们需要写一个驱动程序进行测试，驱动代码如下所示：

示例代码 3.4.1 AMP 驱动测试代码

```
1 #include <linux/module.h>
2 #include <linux/platform_device.h>
3 #include <asm/io.h>
4 #include <linux/irqchip/arm-gic.h>
5
6 #define SHARE_MEM_ADDR     0x25000000U        // 共享内存地址
7
8 static void __iomem *share_meme_addr;                // 共享内存虚拟地址
9
10 /* 软中断处理函数 */
11 static void zynq_ipi_handler(void)
12 {
13         u32 val;
14         val = readl(share_meme_addr);          // 读取共享内存中的数据
15         printk(KERN_INFO "CPU0: Received SGI interrupt signal from CPU1\n");     // 打印信息
16         printk(KERN_INFO "CPU0: Received data is %d\n", val);                    // 打印数据
17         gic_raise_softirq(cpumask_of(1), 15);                    // 触发软中断 15 通知 CPU1
18 }
19
20 static int zynq_amp_probe(struct platform_device *pdev)
21 {
22         share_meme_addr = ioremap(SHARE_MEM_ADDR, 4);// 将共享内存物理地址转换为虚拟地址
23         return set_ipi_handler(14, zynq_ipi_handler, "Zynq amp test"); // 注册软中断处理函数，将软中断 14 绑定到 zynq_ipi_handler 处理函数
24 }
25
26 static int zynq_amp_remove(struct platform_device *pdev)
27 {
28         clear_ipi_handler(14);
29         return 0;
30 }
31
32 static const struct of_device_id amp_of_match[] = {
33         { .compatible = "xlnx,zynq-amp" },
34         { /* Sentinel */ }
35 };
```

```
36
37 static struct platform_driver zynq_amp_test = {
38         .driver = {
39                 .name                   = "zynq_amp_test",
40                 .of_match_table = amp_of_match,
41         },
42         .probe                  = zynq_amp_probe,
43         .remove                  = zynq_amp_remove,
44 };
45
46 module_platform_driver(zynq_amp_test);
47
48 MODULE_AUTHOR("DengTao <773904075@qq.com>");
49 MODULE_DESCRIPTION("Xilinx ZYNQ Dual-Core Communication Test Based On SGI");
50 MODULE_LICENSE("GPL");
```

驱动代码非常简单，主要的内容就是在 platform 总线的 probe 函数中注册了一个软中断处理函数 zynq_ipi_handler，与 14 号软中断绑定；当 CPU1 应用程序触发 14 号软中断的时候会由 CPU0 执行这里注册的中断处理函数 zynq_ipi_handler。

第 17 行，在 zynq_ipi_handler 中断处理函数中，使用 gic_raise_softirq 函数触发软中断，这个函数有两个参数，第一个参数对应的就是 CPU 的 ID 号，在内核当中 CPU id 一般是从 0 开始；第二个参数也就是软中断号。

第 23 行，在 zynq_amp_probe 函数中调用 set_ipi_handler 注册软中断处理函数，这个函数提供了三个参数，第一个参数表示软中断号，第二个参数表示对应的中断处理函数，最后一个参数类似于一个描述的字符串。

整个程序就给大家讲到这里，本身非常简答没啥内容。需要注意，在 Linux 内核当中，有一些软中断号已经给内核使用，在 arch/arm/kernel/smp.c 源文件中有说明：

示例代码 3.4.2 内核已经使用的软中断

```
enum ipi_msg_type {
        IPI_WAKEUP,
        IPI_TIMER,
        IPI_RESCHEDULE,
        IPI_CALL_FUNC,
        IPI_CPU_STOP,
        IPI_IRQ_WORK,
        IPI_COMPLETION,
        IPI_CPU_BACKTRACE,
        /*
         * SGI8-15 can be reserved by secure firmware, and thus may
         * not be usable by the kernel. Please keep the above limited
         * to at most 8 entries.
         */
};
```

由此可知，内核使用了 0~7 一共 8 个软中断，所以这些是不能再被用户使用了，可以选择 8~15 这些中断号，所以本实验就选了 14 和 15。

驱动编写完成之后，还需要在内核设备树中添加对应的节点，例如：

示例代码 3.4.3 设备节点示例

```
zynq_amp_test {
    compatible = "xlnx,zynq-amp";
};
```

编译驱动、编译设备树，替换开发板 SD 启动卡中的设备树文件，重启开发板，进入到 Linux 系统，将驱动模块文件拷贝到开发板家目录下，如下：

```
root@ALIENTEK-ZYNQ:~#
root@ALIENTEK-ZYNQ:~# ls
zynq_amp_test.ko
root@ALIENTEK-ZYNQ:~#
root@ALIENTEK-ZYNQ:~#
```

图 3.4.1 驱动模块文件

3.5 测试

接下来加载 zynq_amp_test.ko 驱动模块文件：

```
insmod zynq_amp_test.ko
```

加载完成之后，出现了如下打印信息：

```
root@ALIENTEK-ZYNQ:~#
root@ALIENTEK-ZYNQ:~# insmod zynq_amp_test.ko
[   19.854501] zynq_amp_test: loading out-of-tree module taints kernel.
root@ALIENTEK-ZYNQ:~# [   20.347804] CPU0: Received SGI interrupt signal from CPU1
[   20.353199] CPU0: Received data is 5
CPU1: Received SGI interrupt signal from CPU0

[   22.347798] CPU0: Received SGI interrupt signal from CPU1
[   22.353192] CPU0: Received data is 0
CPU1: Received SGI interrupt signal from CPU0

[   24.347800] CPU0: Received SGI interrupt signal from CPU1
[   24.353195] CPU0: Received data is 1
CPU1: Received SGI interrupt signal from CPU0

[   26.347801] CPU0: Received SGI interrupt signal from CPU1
[   26.353197] CPU0: Received data is 2
CPU1: Received SGI interrupt signal from CPU0

[   28.347801] CPU0: Received SGI interrupt signal from CPU1
[   28.353198] CPU0: Received data is 3
CPU1: Received SGI interrupt signal from CPU0

[   30.347801] CPU0: Received SGI interrupt signal from CPU1
[   30.353192] CPU0: Received data is 4
CPU1: Received SGI interrupt signal from CPU0
```

图 3.5.1 加载驱动测试

打印信息会一直出现，因为在 CPU1 应用程序中，在 while 循环当中一直在触发 CPU0 对应的软中断，所以 CPU0 Linux 这边会一直接受到中断信号，然后执行中断处理函数。

打印信息就不给大家解释了，代码里边已经解释得非常清楚了！

3.6 总结

本章希望大家去理解、明白、并掌握裸机和 Linux 系统在 AMP 环境下的通信方法，不用照着步骤一步一步来，看懂就行了，明白每一个步骤的用意，然后你自己再去创建工程、一步一步去测试，自己去实操。

第四章 OpenAMP 测试

后续更新，敬请关注！