AC501_SoC FPGA 开发板黄金参考设计说明

小梅哥编写，未经许可严谨用于任何商业用途

2018 年 7 月 4 日星期三

什么是 GHRD

GHRD 是 Golden Hardware Reference Design 的简写，中文意思为黄金硬件参考设计，一般由 DEMO 板硬件厂家提供。在该硬件设计工程中，已经针对 DEMO 板上的各种硬件外设添加好了对应的控制电路，并分配好了引脚。使用时，设计者只需在该参考设计中添加或修改自己所需的工作即可。

SoC FPGA 的开发涉及到 HPS 中各种参数配置，IO 交错配置， DDR3 存储器参数配置以及 FPGA 工程信号连接和引脚分配。对于刚接触的设计者来说，从 0 开始搭建基于 HPS 的应用系统，不仅工作量巨大，而且很容易出错。因此，在基于 SoC FPGA 的板卡进行设计开发时，推荐使用板卡厂家提供的 GHRD 进行修改增删，以完成自己的系统设计。

GHRD FOR AC501-SoC

AC501-SoC 开发板提供了一个基本的 GHRD 工程，名为 AC501_SoC_GHRD，在该工程中，使用 Platform Designer 创建了一个包含 HPS 的 qsys 设计文件，在该系统中添加了 HPS，和一些常必备的 FPGA 侧软 IP，并对其各个工作参数，包括 HPS 主时钟、外设时钟、外设中断、外设引脚、DDR3 工作时钟等众多参数进行了合理设置。接下来首先介绍该 qsys 设计文件中内容。

打开和查看 GHRD

使用 Quartus 17.1 软件打开 AC501_SoC_GHRD 工程，在菜单栏中依次点击 Tools > Platform Designer 或者直接点击 Platform Designer 图标  以打开 Platform Designer 系统设计工具，在弹出的打开窗口中，选择 soc_system.qsys 文件并打开，如图 xxx 所示：

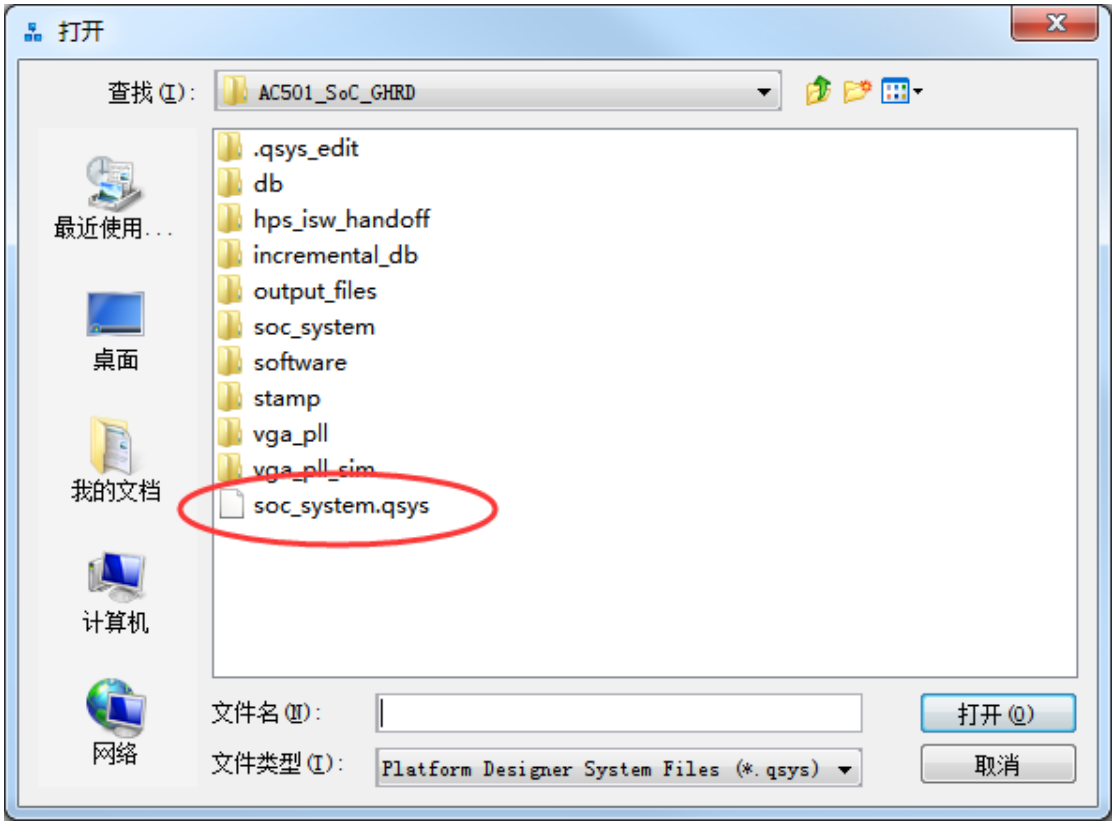


图 xxx 打开 qsys 设计文件

文件打开后可以看到，该设计文件中已经加入了众多的组件，包括时钟输入源（clk_0）、HPS（hps_0）、系统 ID（systemid_qsys）、led 控制引脚（led_pio，仅输出型 PIO 软核）、按键控制引脚（button_pio 仅输入型 PIO 软核）、异步串口（uart_0）、spi 控制器（spi_0）、I2C 控制器（i2c_0），图像帧读取控制器 Frame Reader（alt_vip_vfr_tft）、VGA 控制器（alt_vip_itc_0）。如图 xxx 所示：

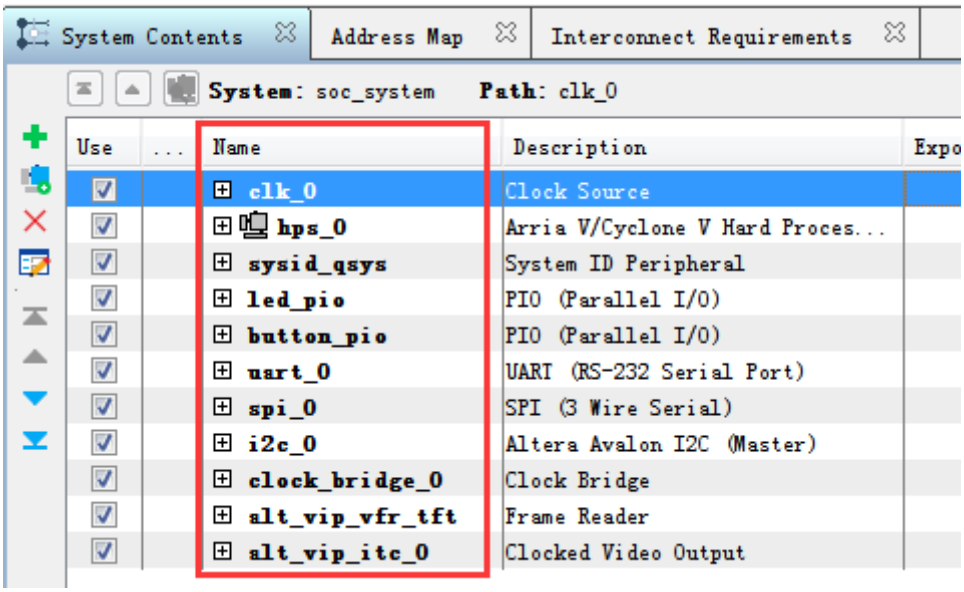


图 xxx Qsys 设计文件包含的组件

为了展示方便，在贴图前将所有 IP 都进行了折叠，不现实各种信号连接，在折叠的情况下，用户可以点击 IP 前面的+符号以展开 IP，从而查看其各种信号连接情况。如图 xxx 所示：

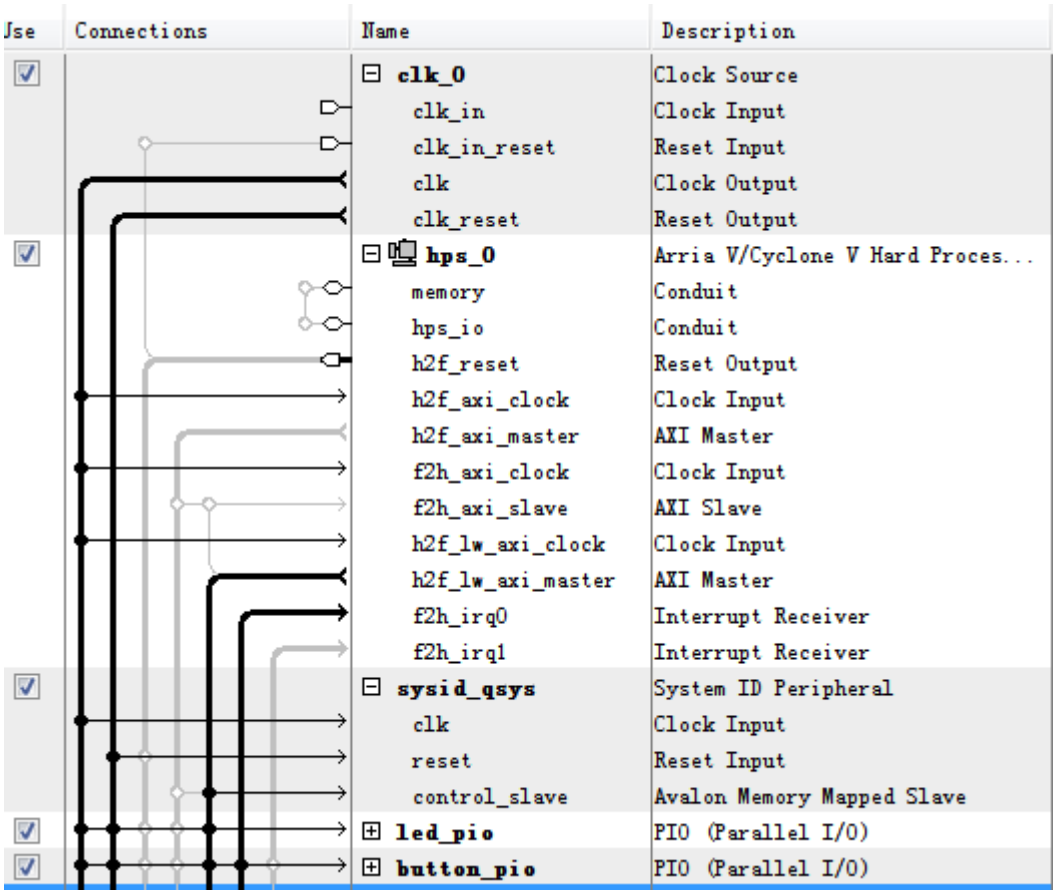


图 xxx 展开和折叠每个组件信号

对于每个组件，其都有一个或多个参数需要配置。我们可以通过选中该组件，然后右击选择 Edit 来查看和编辑其参数。以下对每个组件的功能作进一步介绍。

clk_0

时钟源 IP，实现从系统外部输入一个时钟和复位信号到系统内部的功能，该 IP 在基于 NIOS II 的 SOPC 系统设计中也是一个必用 IP，该 IP 核有一个参数可以设置，即外部输入时钟的频率，如果输入频率是已知的，勾选上 Clock frequency is known，并在上方的 Clock frequency 一栏中输入实际的时钟频率，例如在本系统中，该 IP 接到了板载 50MHz 有源晶振上，因此 Clock frequency 值设置为 50_000_000Hz，如下图所示：

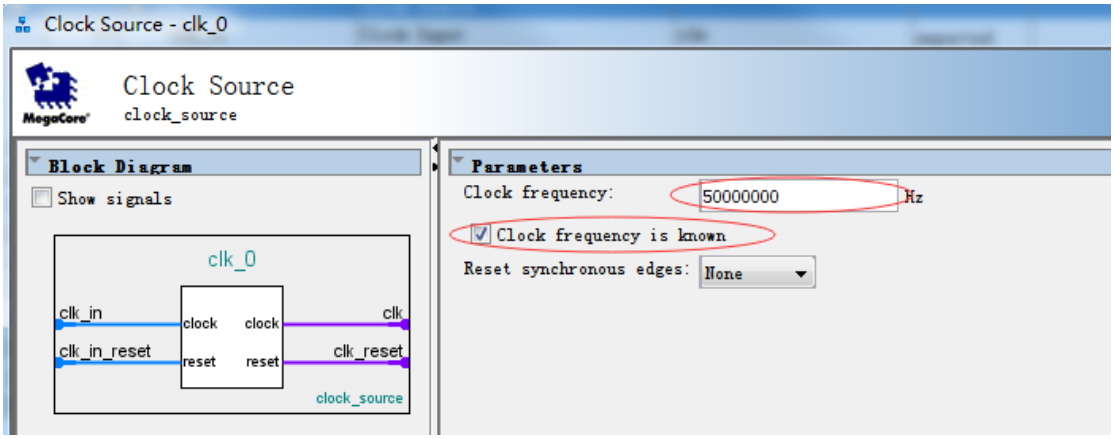


图 clk_0 IP 参数设置

sysid_qsys

系统 ID，用来在软件编程时识别系统，通过唯一 ID，来区分不同的硬件设计工程，只有软件和硬件匹配时才能够正常运行。该 ID 需要手动设置，且仅支持 16 进制格式的数字，我们可以根据自己的意愿，手工设置一个 ID 值。如图 xxx 所示：

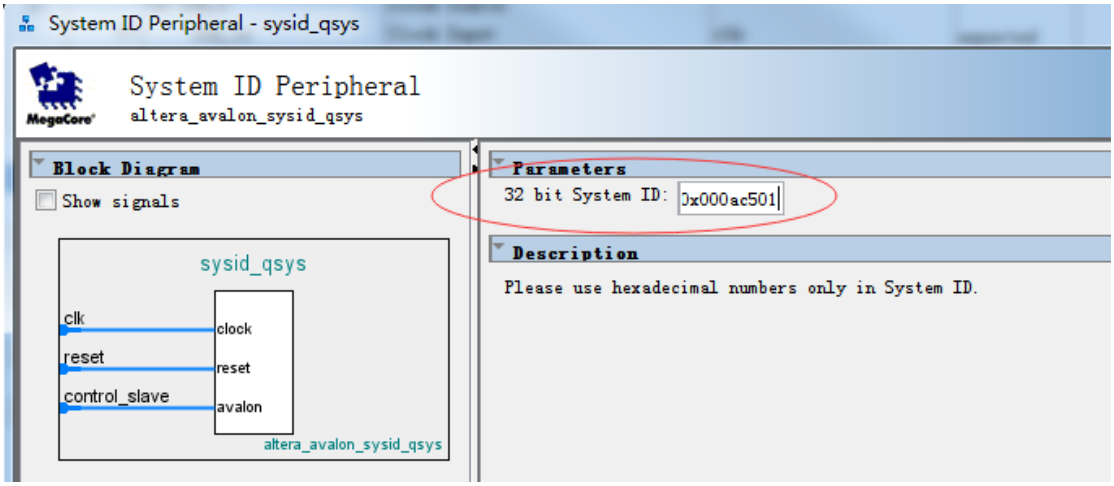


图 xxx sysid 参数设置

led_pio

PIO 软 IP，使用 IP Catalog 中的 PIO（Parallel I/O）组件，设置为 2 位的仅输出型 PIO，同时使能了独立位操作功能，用来驱动 FPGA 侧的两个 LED 灯，如图 xxx 所示：

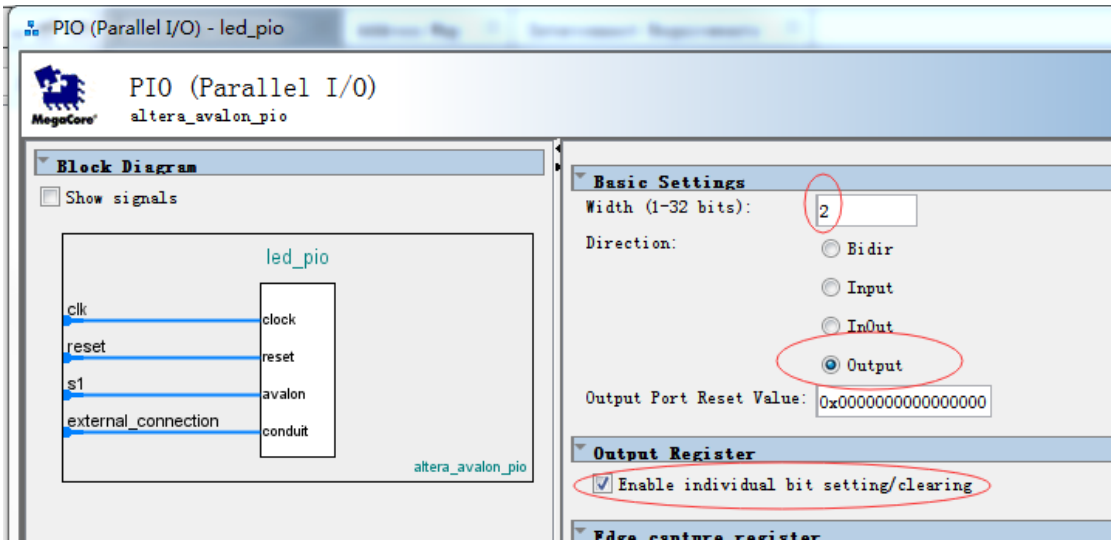


图 xxx led_pio 参数设置

button_pio

PIO 软 IP，使用 IP Catalog 中的 PIO（Parallel I/O）组件，设置为 2 位的仅输入型 PIO，同时使能了下降沿捕获和边沿中断触发功能，用来连接 FPGA 侧的两个轻触按键，如图 xxx 所示：

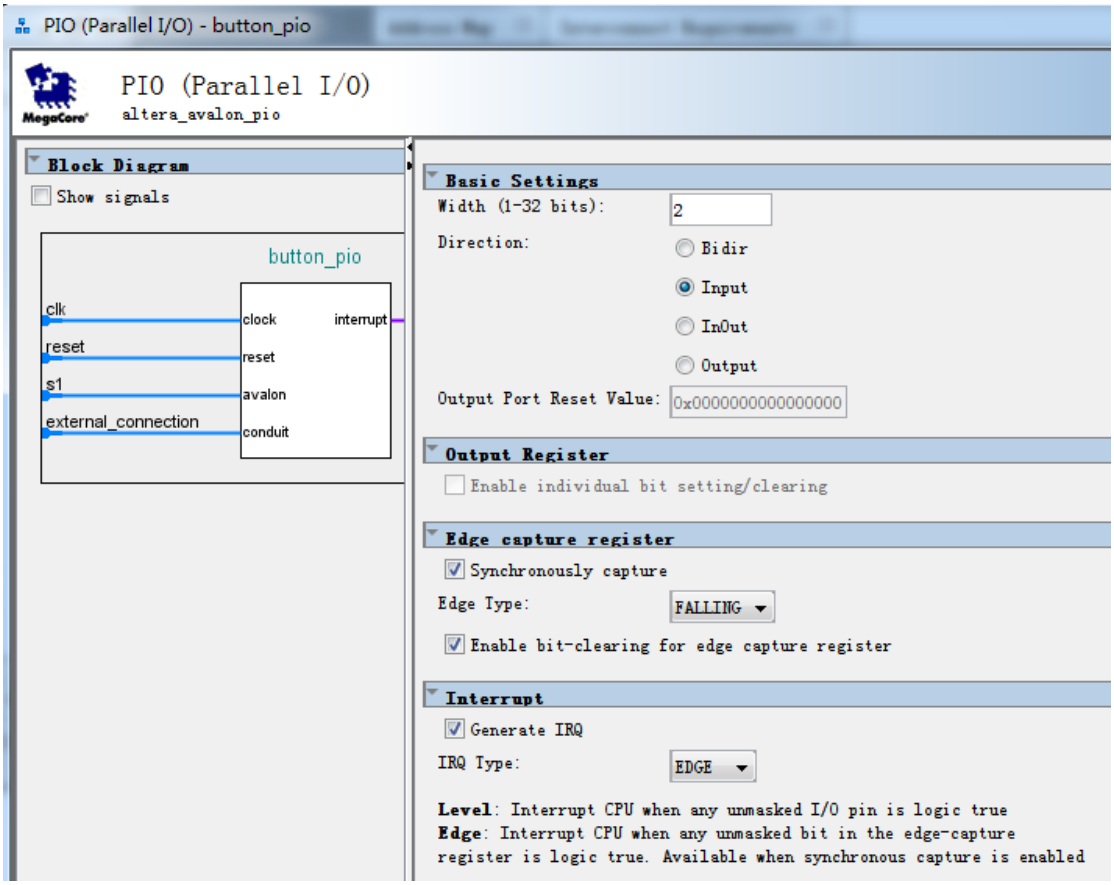


图 xxx button_pio 参数设置

uart_0

异步串口控制器，使用 IP Catalog 中的 UART （RS-232 Serial port）组件，设置默认波特率为 115200，如图 xxx 所示。由于 HPS 中只有两个 UART 硬 IP，而通过该组件可以为 HPS 添加一个或多个串口控制器，这在需要多串口的应用场合显得十分方便有用。该 IP 在 Linux 内核中已有现成驱动支持，可以直接被 Linux 内核识别并加载驱动，然后像使用标准的 Linux TTY 一样方便的使用。

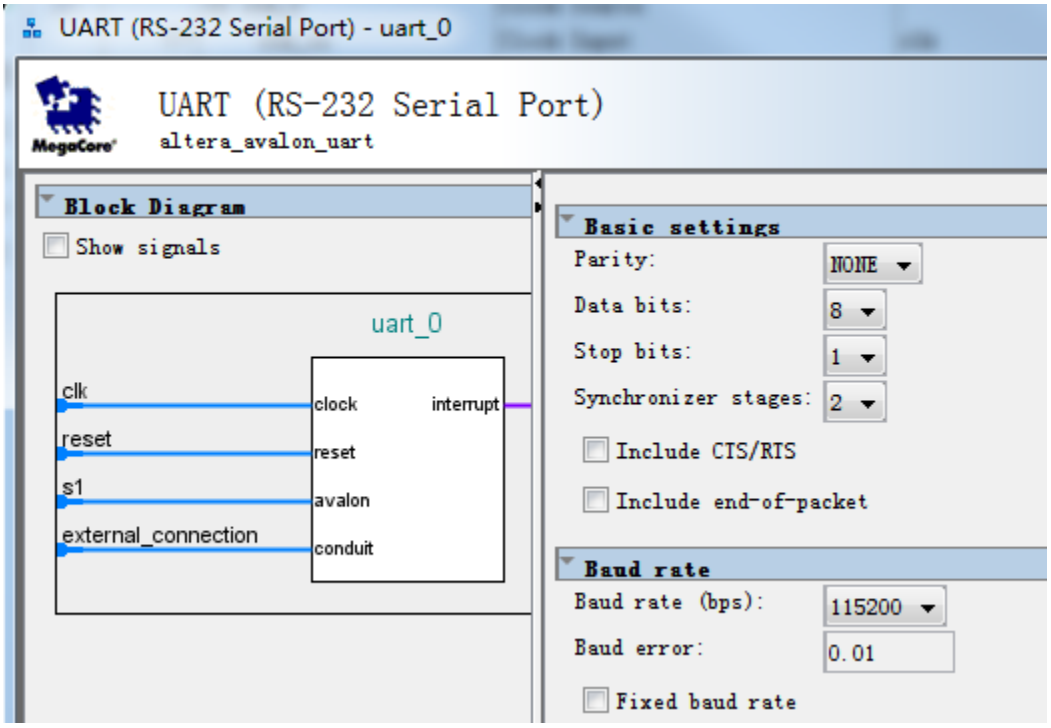


图 xxx uart_0 IP 参数设置

spi_0

SPI 总线控制器，使用 IP Catalog 中的 SPI（3 Wire Serial）组件。该 IP 可工作在主机或从机模式，本例中工作在主机模式，一个片选信号。SCLK 时钟速率为 2MHz（实际能够生产的有效工作时钟频率为 1923076Hz）。8 位宽数据模式。时钟相位和极性均为 0，如图 xxx 所示。由于 HPS 中只有两个 SPI 硬 IP，而通过该组件可以为 HPS 添加一个或多个 SPI 控制器，这在需要多 SPI 的应用场合显得十分方便有用。该 IP 在 Linux 内核中已有现成驱动支持，可以直接被

Linux 内核识别并加载驱动。不同于 UART 的是，SPI 属于总线，因此不能被直接识别为设备，因此需要进一步在该总线上添加设备描述信息才能够正常使用。该部分内容会在应用章节详细讲到。

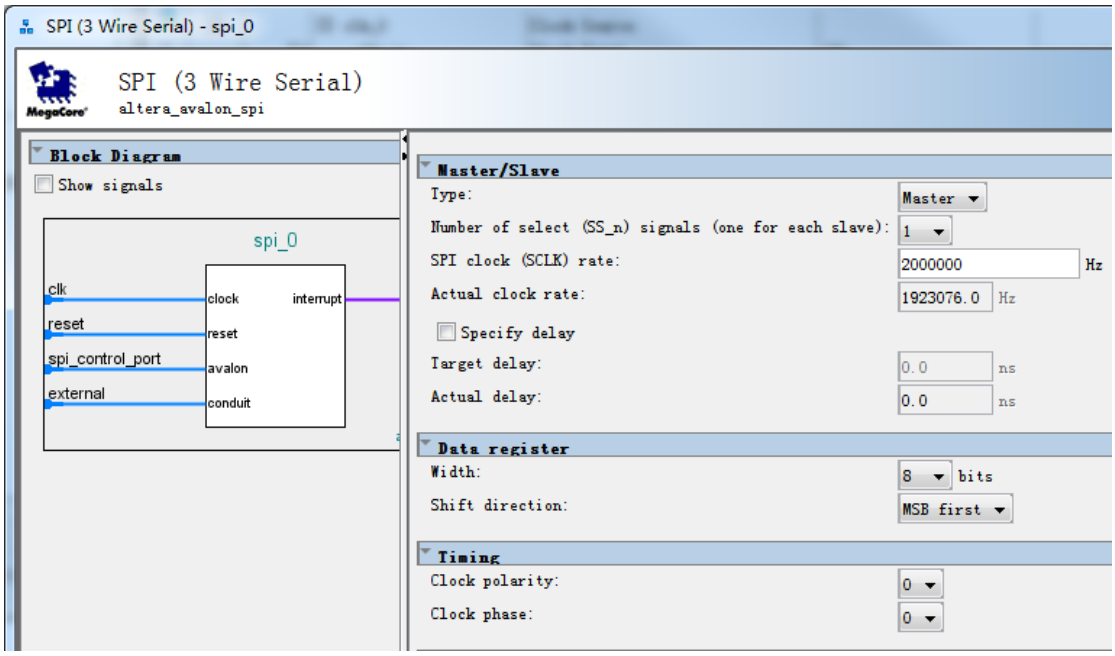


图 xxx spi_0 IP 参数设置

i2c_0

I2C 控制器，使用 IP Catalog 中的 Altera Avalon I2C（Master）组件。这是一个仅支持主机模式的 I2C 控制器，目前仅提供了 NIOS 下的驱动代码，在 Linux 系统中暂无对该 IP 的驱动支持，不过在使用时可以通过总线映射的方式将其虚拟地址映射到 Linux 内核空间，然后由软件设计者自行对其进行软件编程。

alt_vip_vfr_tft_0

图像帧缓存读取控制器，这是一个能够直接从 AXI 总线上以 DMA 的方式读取内存中的数据的 IP，通过在 Linux 系统中申请一块存储区域用来缓存一帧图像的数据，然后再由该 IP 从内存中读出，送往 VGA 控制器以驱动 VGA 显示器或 RGB TFT 显示屏，将读取出来的帧图像数据显示在显示屏上，从而实现图像显示的功能。该 IP 是从早期 Quartus 软件中沿用过来的，在 17.1 版本软件中，该 IP 已经合并到了 Frame Buffer II IP 中了，不过该 IP 依旧能够继续使用，而且使用起来比 Frame Buffer II IP 更加方便，因此这里还是继续沿用该 IP 进行设计。

该 IP 核可以设置读取的图像内容格式和大小，如图 xxx 所示。在其参数设置界面中，有多个比较重要的参数需要设置，现分别说明。

- **Bits per pixel per color plane**：每一种元色由多少位数据表示。对于 RGB 模式，每一个像素点颜色都由红（red）、绿（green）、蓝（blue）三种单色根据不同比例混合而成。pixel per color plane 则代表了每一种单元色用多少的数位表示。例如最常用的 RGB888 模式就是每种单元色用 8 位数据表示。
- **Number of color planes in parallel**：每一种颜色由多少个单元色组成，in parallel 的意思是将指定数量的单元色的数据合并到一起。例如，上面说到每个单元色用 8 位数据表示，这里每个像素有四个单元色，则实际的图像数据位宽就是 32。由于 RGB888 模式通常只有 3 种单元色，而这里设置为 4 种元色是为了在图像数据在内存中组织存储时能够与内存的 32 位数据存储方式对齐，增加寻址效率，实际在输出到 VGA 时，32 位数据的最高 8 位都是直接丢弃的。
- **Number of color planes in sequence**：按顺序传输时，每个像素颜色由多少个单元色组成，意思是一个像素点的数据，按顺序每个时钟周期传输一个单元色的数据，若干次传输的数据组合为一个完整像素图像。此种情形主要用于板件图像数据传输，例如在友晶的 DE2-115 开发板配套 7 寸触摸显示套件（tPad）中，就使用了此种传输方式，在 tPad 显示屏上使用了一个 MAX II CPLD 完成接收 DE2-115 发送的 in sequence 模式的图像数据，并将每 3 个时钟周期的数据的重新组合得到 24 位的 RGB888 数据，驱动 TFT 屏显示。
- **Maximum Image width**：最大图像宽度，即一行图像由多少个像素点组成，在 GHRD 中，默认是用来驱动 5 寸 800*480 分辨率的触摸显示屏的，因此该值为 800，如果需要驱动其他分辨率的显示器，则此处的值需要根据实际物理分辨率对应修改。
- **Maximum Image hight**：最大图像高度，即一幅图像由多少行组成，针对 800*480 分辨率的显示屏，该值应该为 480。

- **Master port width：**avalon mm 主机的数据位宽。avalon mm 主机负责快速的从 HPS 的内存中读取图像数据，该值应该与 f2h_axi_slave 桥的数据位宽一致，才能获得最高的读写效率。

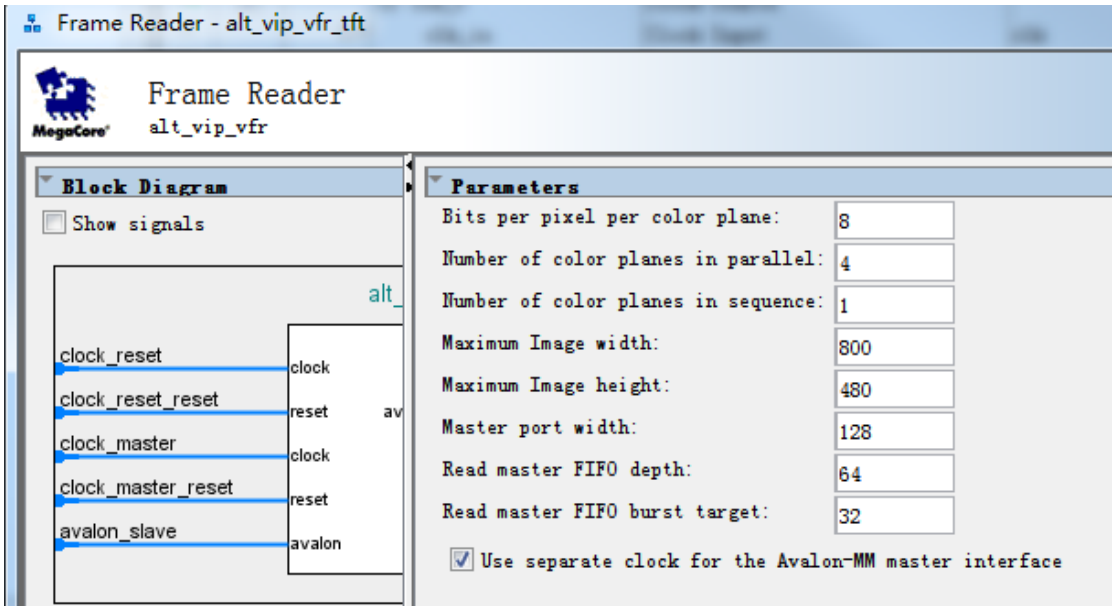


图 xxx alt_vip_vfr_tft_0 参数设置

由于 AC501-SoC 开发板配套的显示屏使用的是 RGB565 方式，因此一个像素点仅需 2 个字节数据即可，笔者曾经尝试修改 Frame Reader 的参数，以实现 RGB565 模式。因为相对于 RGB888 模式，RGB565 模式显示一幅图像需从 HPS 内存中读取的数据量仅为 RGB888 模式的一半，从而可以大幅降低对内存带宽的占用，预留更多带宽给其他应用使用，但是尝试失败，原因是 Linux 内核驱动中提供的默认该 Frame Reader 的驱动仅支持 32 位的 RGB888 模式，改为其他模式后需要修改驱动程序，因此暂时没有做修改，继续使用 RGB888 模式。

alt_vip_itc_0

实质就是一个 VGA 控制器，将 Frame Reader 读取的图像数据流按照 VGA 时序送往 VGA 显示器或者 RGB TFT 显示屏显示。支持设置多种分辨率，在 GHRD 工程中默认按照 800*480 分辨率的 TFT 显示屏进行设置。具体的参数修改，会在后续章节讲解修改 Frame Buffer 分辨率时详细介绍。

总结

本节介绍了黄金硬件参考设计（GHRD）的相关概念，并针对一个特定的SoC 板卡的 GHRD 工程进行了简单说明，讲述了工程中使用的各个 IP 组件的功能和一些重要的参数。关于 AC501-SoC 开发板的 GHRD 工程基本介绍就介绍到此，接下来将依次讲解如何通过简单的编程来完成中每个 IP 组件的使用，该部分工作主要基于 DS-5 软件编程实现。

使用 DS-5 编写和调试 SoC 的 Linux 应用程序

作者：小梅哥

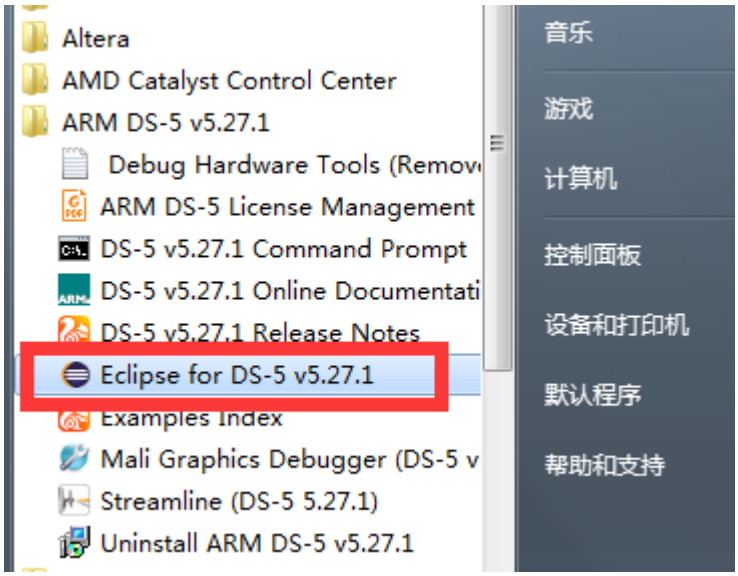
在上一节中，讲述了 AC501_SoC 开发板提供的黄金硬件参考设计文件中设计的相关组件的信息。本节将针对该黄金参考设计工程添加的各种 IP 组件，编写应用程序来实现对这些组件的操作控制。

虽说开发基于 Linux 操作系统的应用程序，一般都推荐在 Linux 开发环境如 Ubuntu 系统中进行。但是对于 Intel Cyclone V SoC FPGA 用户，如果仅仅开发应用程序，也可以在 Windows 环境下完成。Intel 针对其自家的 SoC FPGA 芯片提供了定制的 DS-5 软件，该软件为 ARM 公司针对其自家芯片研发，功能强大，尤其是其自带的编译器，编译出来的 ARM 可执行文件运行起来有更高的效率。软件不仅支持 ARM 裸机程序开发和调试，也支持 Linux 应用程序开发和调试。不过其自带的编译器和调试器是收费的，需要获得相应的 License 才能够使用。但同时该软件也集成了有 Linaro 公司针对 Cyclone V SoC FPGA 验证通过的专用 GCC 编译工具，名字叫 gcc-linaro-arm-linux-gnueabihf，使用该工具编译和调试 Linux 应用程序无需获取 License。

本节将介绍如何使用 Intel 版的 DS-5 软件，创建和编译，并调试一个最简单的 Linux 应用程序。

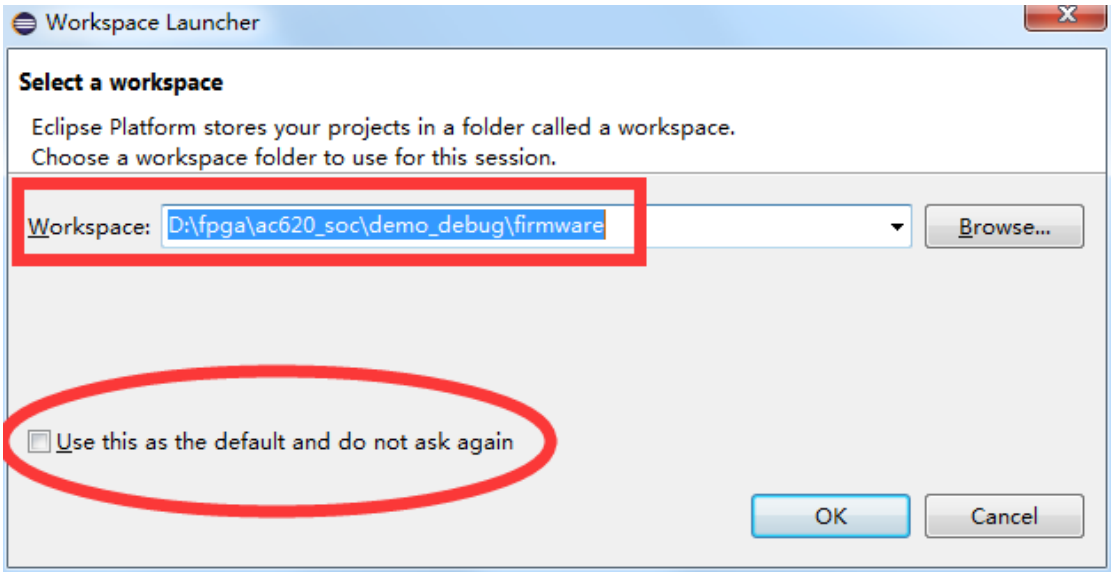
1、启动 DS-5

从开始菜单栏中找到 ARM DS-5 v5.27.1 目录，选择 Eclipse for DS-5 v5.27.1 打开。



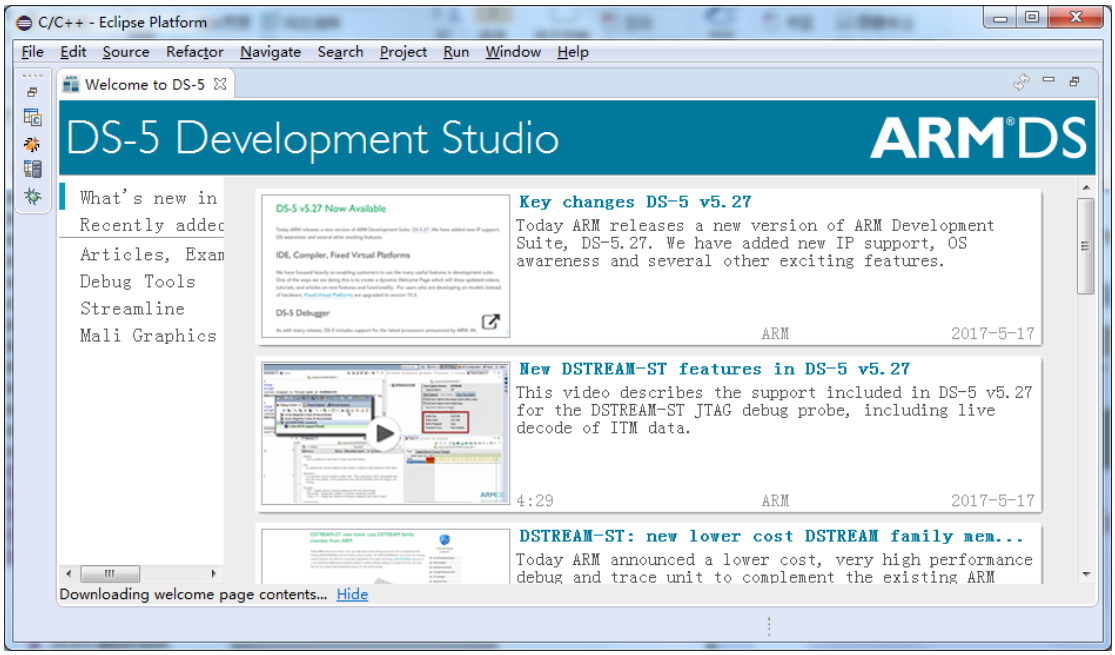
当然，如果已经熟悉了 SoC EDS Command Shell 工具的使用，也可以在 SoC EDS Command Shell 中直接输入“eclipse&”来打开，熟悉之后，推荐使用此种方式打开，因为使用此种方式打开能够通过 SoC EDS Command Shell 自动的设置好若干环境变量，在开发一些涉及到硬件外设的程序的时候有用到。

第一次启动会要求设置 workspace，如图下图所示，如果不希望每次都出现该提示框，可勾选界面中的复选框。



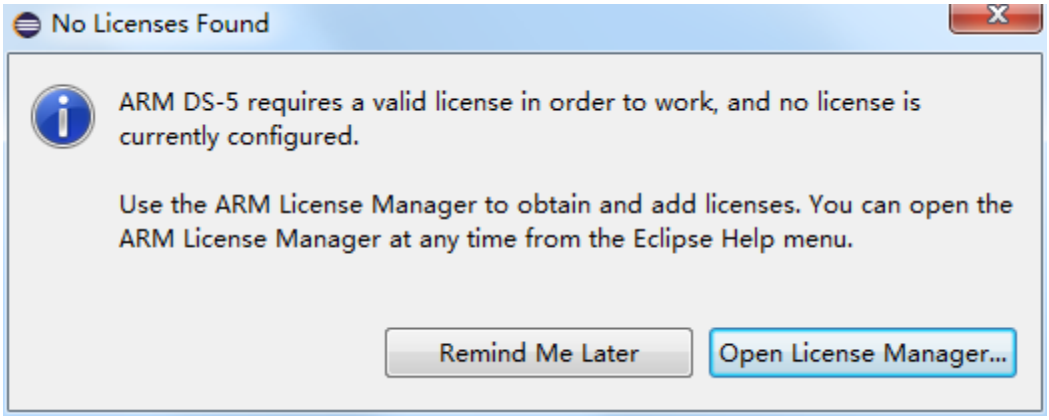
这里选择一个靠谱的位置作为工作空间路径，例如我们在编写资料时，选择的 D:\fpga\ac620_soc\demo_debug\firmware。

点击 OK，进入 Eclipse 的欢迎界面，如图 10.105 所示

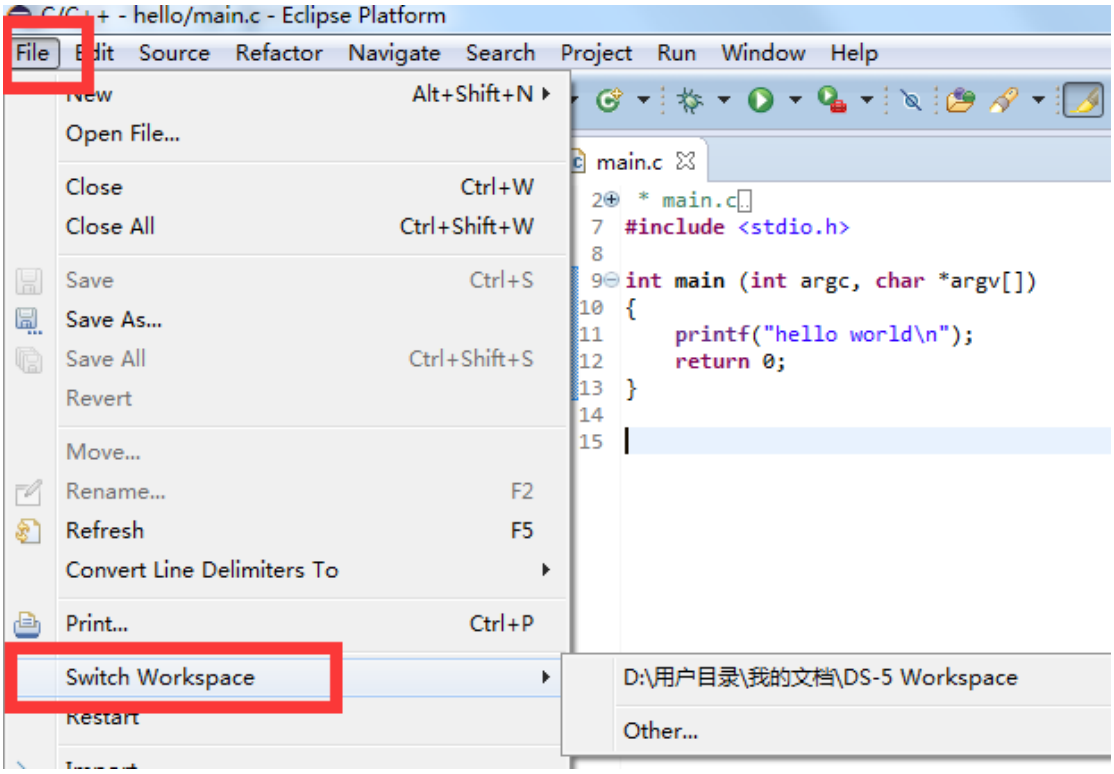


点击 Welcome to DS-5 页面的小叉即可关闭欢迎页面，进入软件的主界面。

如果我们没有为 DS-5 软件添加 License，那么软件每次次启动会默认弹出下列对话框，由于仅仅开发 Linux 应用程序无需 License，因此直接关闭该窗口即可。

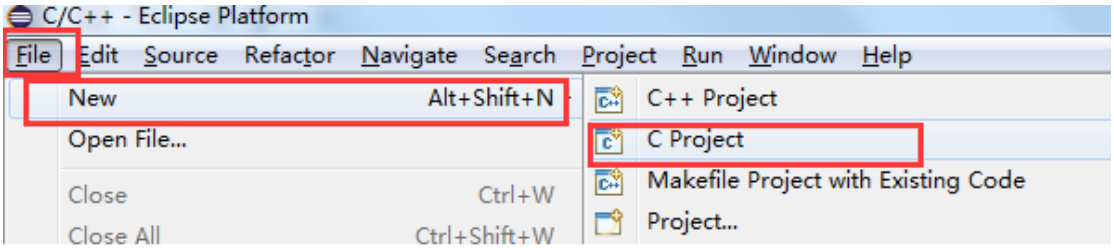


说明：勾选上界面中的复选框之后，下次打开 DS-5，软件会直接进入主界面，不会再弹出路径选择对话框，如果以后我们希望切换路径，可以在软件打开之后，依次点击菜单栏中的 File—> Switch Workspace->other 来重新打开这个窗口

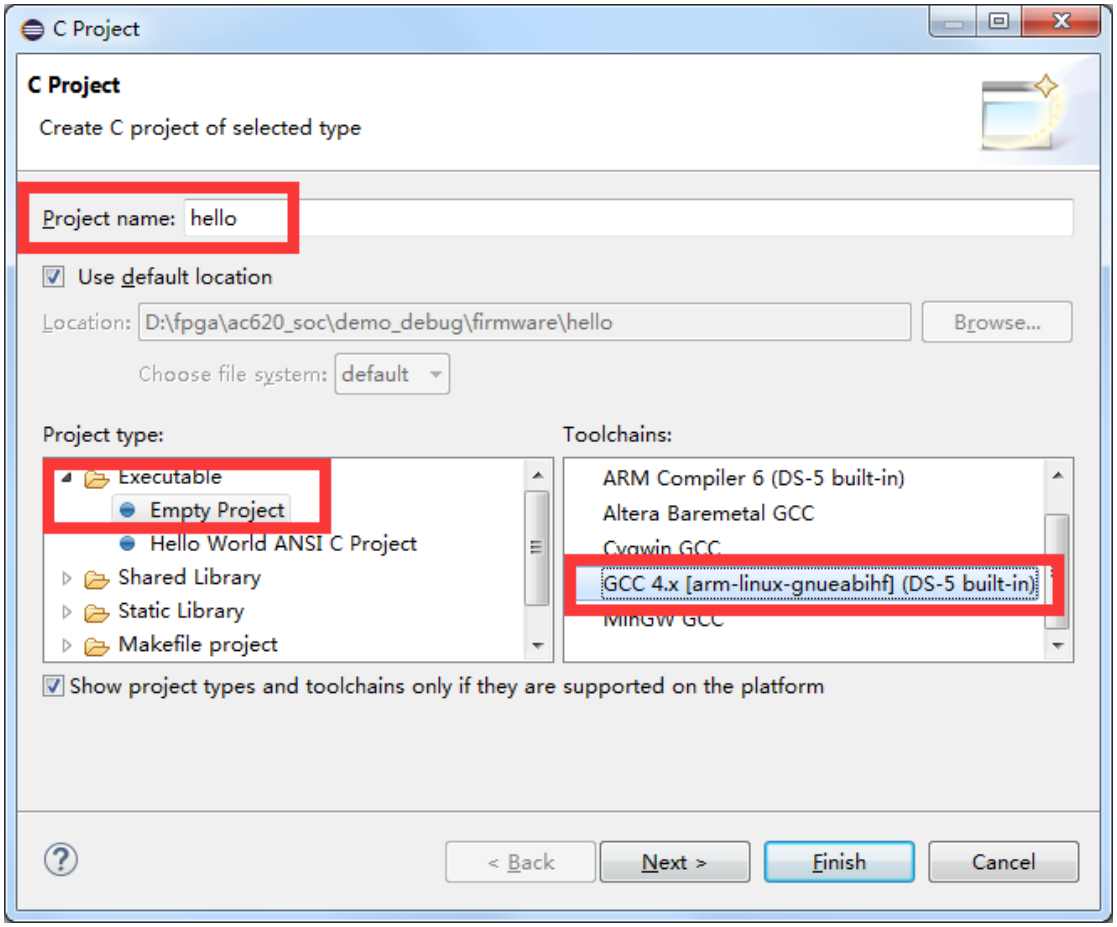


2、创建 C 工程

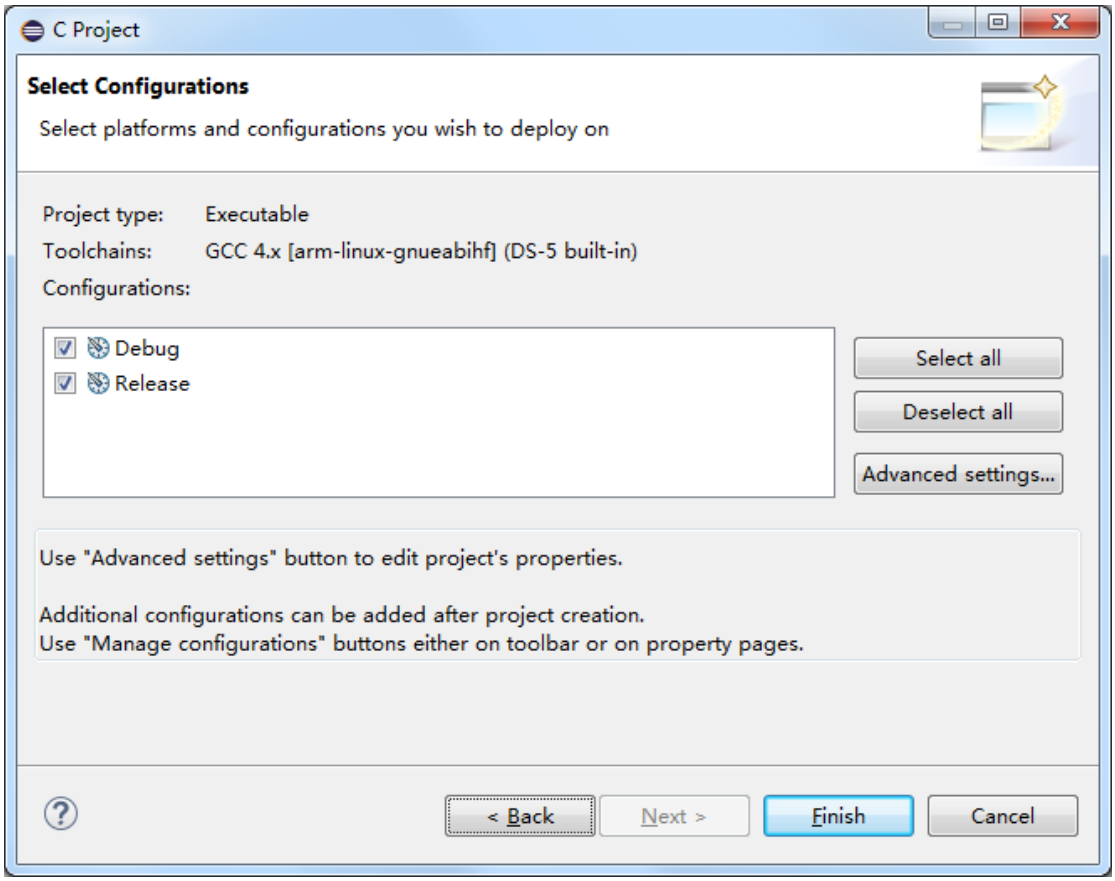
点击主界面 File→New→C Project，如图 10.107 所示，选择创建 C 工程。



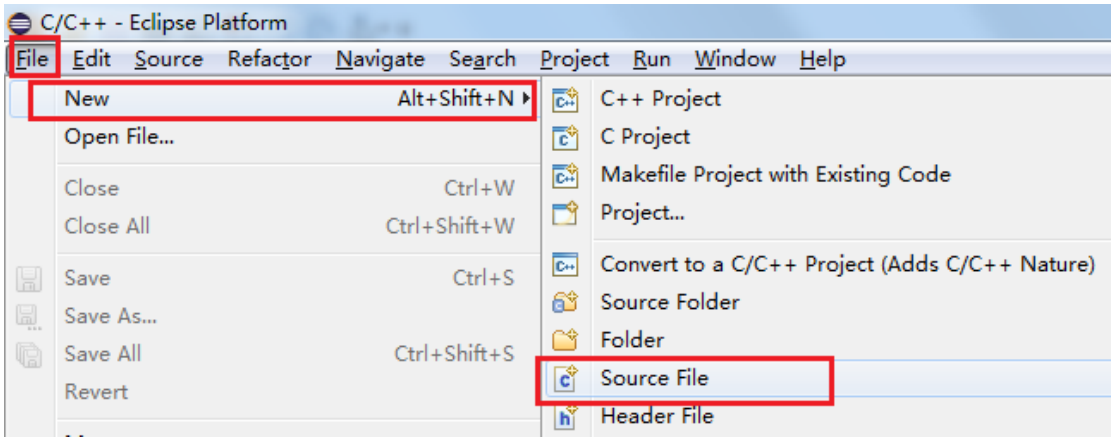
在弹出的工程创建界面，设置工程名称为“ hello”， 工程类型为“ Empty Project”， Toolchain 选择使用“GCC 4.x[arm-linux-gnueabihf](DS-5 built in)”，如下图所示。



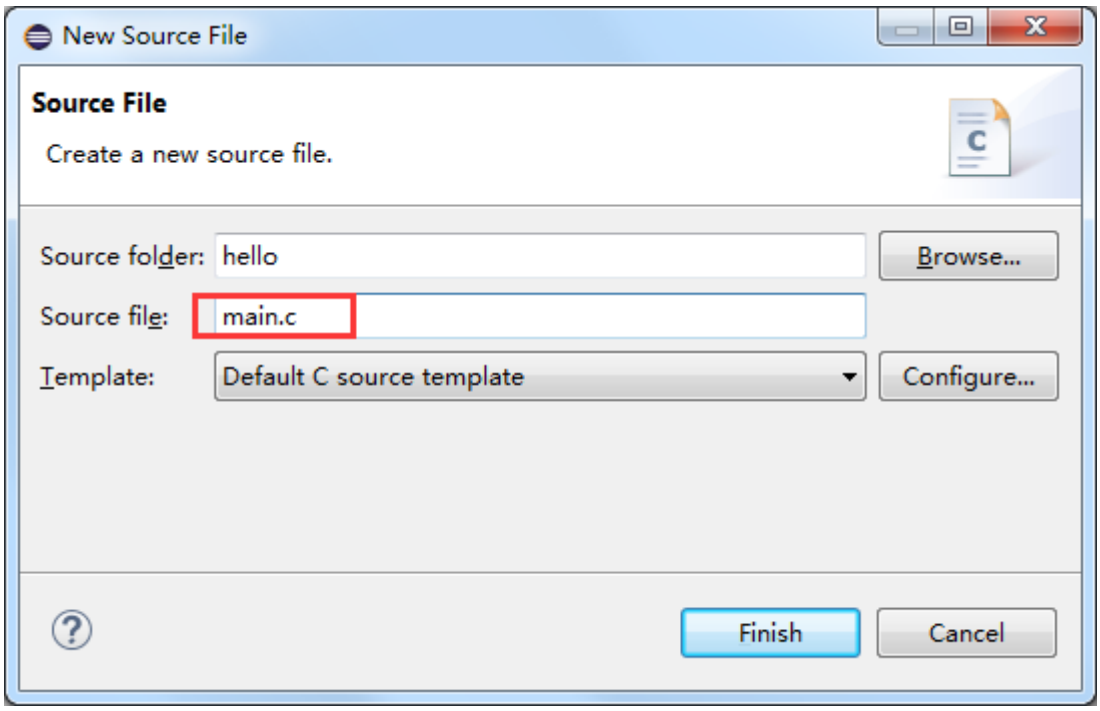
设置好后，点击 Next，在下图所示的配置界面选中“ Debug”和“ Release”两个编译目标。



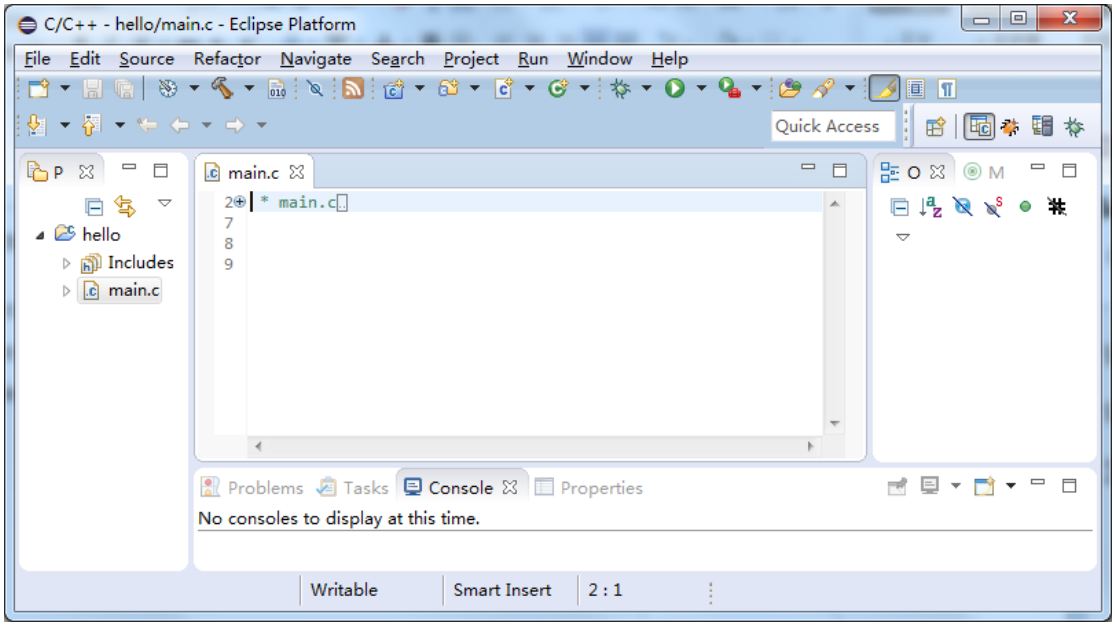
然后点击 Finish 按钮，完成工程创建，得到一个空工程，接下来将添加一个 C 程序文件到该工程，并编写 C 代码。点击 File→New→Source File，选择创建源文件，如下图所示。



在 Source File 界面，设置文件名为“main.c”，使用默认 C 模板，如下图所示：



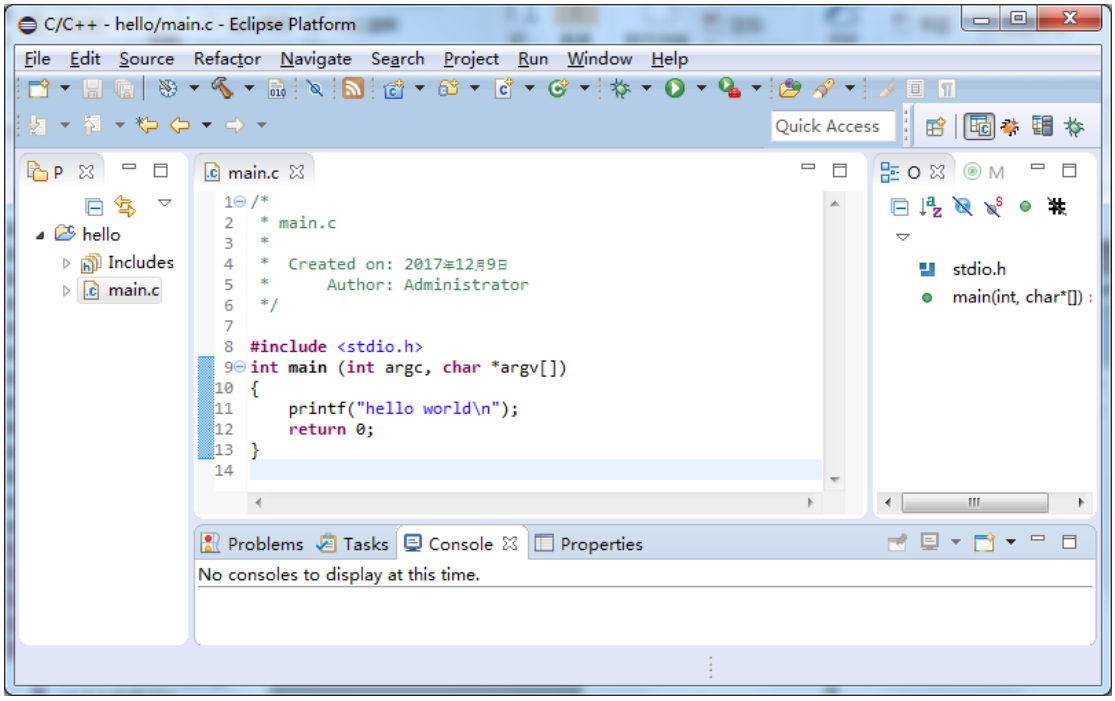
点击 Finish，得到添加了 main.c 的工程，如下图所示。



此时，main.c 文件还是空的，在 main.c 中添加如下代码并保存：

```
#include <stdio.h>

int main (int argc, char *argv[])

{

    printf("hello world\n");

    return 0;

}
```

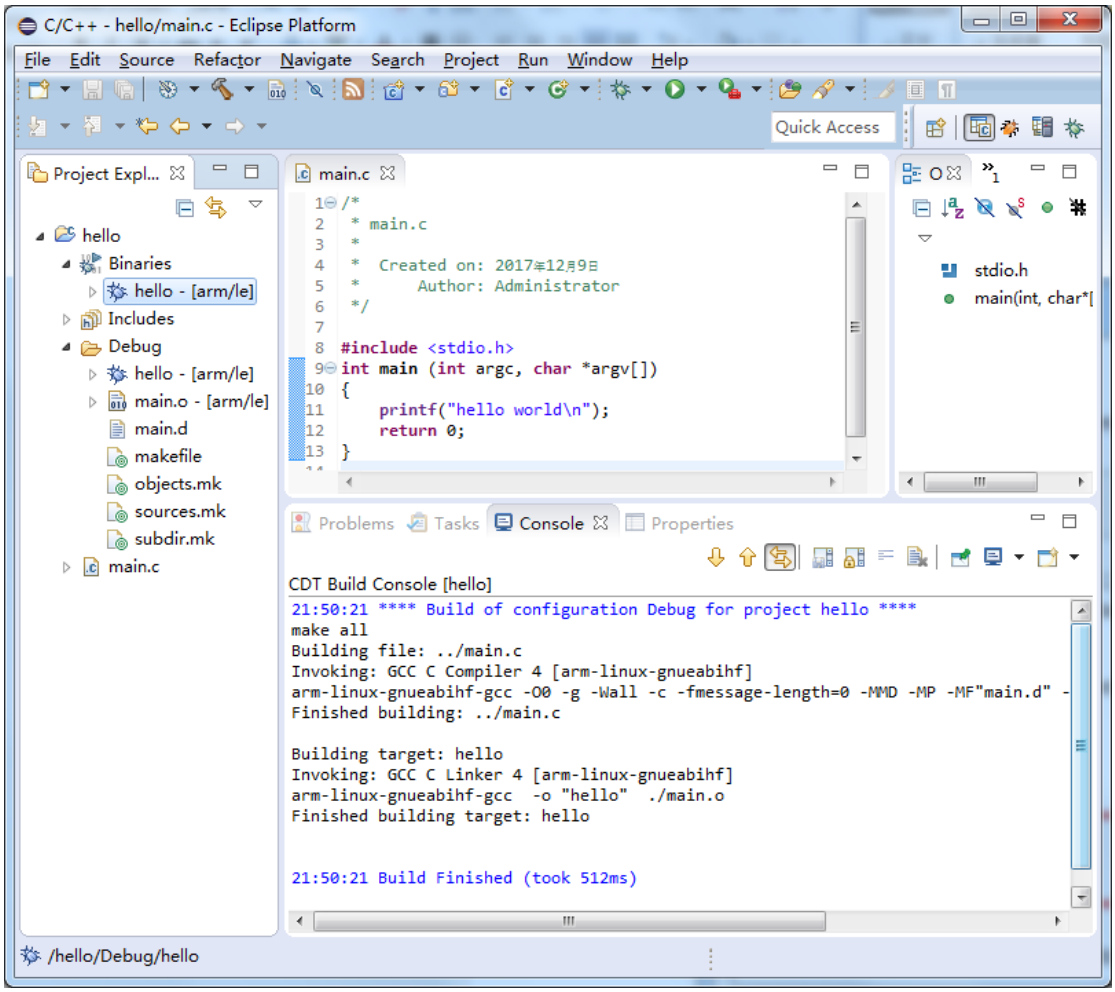
添加了 C 代码的工程如下图所示。



3、编译工程

代码编写完成后，就可以进行编译了，这里的编译实质是交叉编译，即编译得到能够在 ARM 平台上运行的 Linux 应用程序。编译的操作非常简单，按下键盘组合键 Ctrl+B 即可执行编译。

编译完成后左侧工程向导里会生成两个文件夹，分别为 Binaries 和 Debug，Binaries 文件夹下存放的是编译生成的 2 进制可执行文件，Debug 下包含的是生成结果，即通过编译生成了哪些文件，包括 2 进制可执行文件、makefile 和一些编译过程中间文件。对我们来说，当前关心和需要的是 2 进制可执行文件，即 hello。

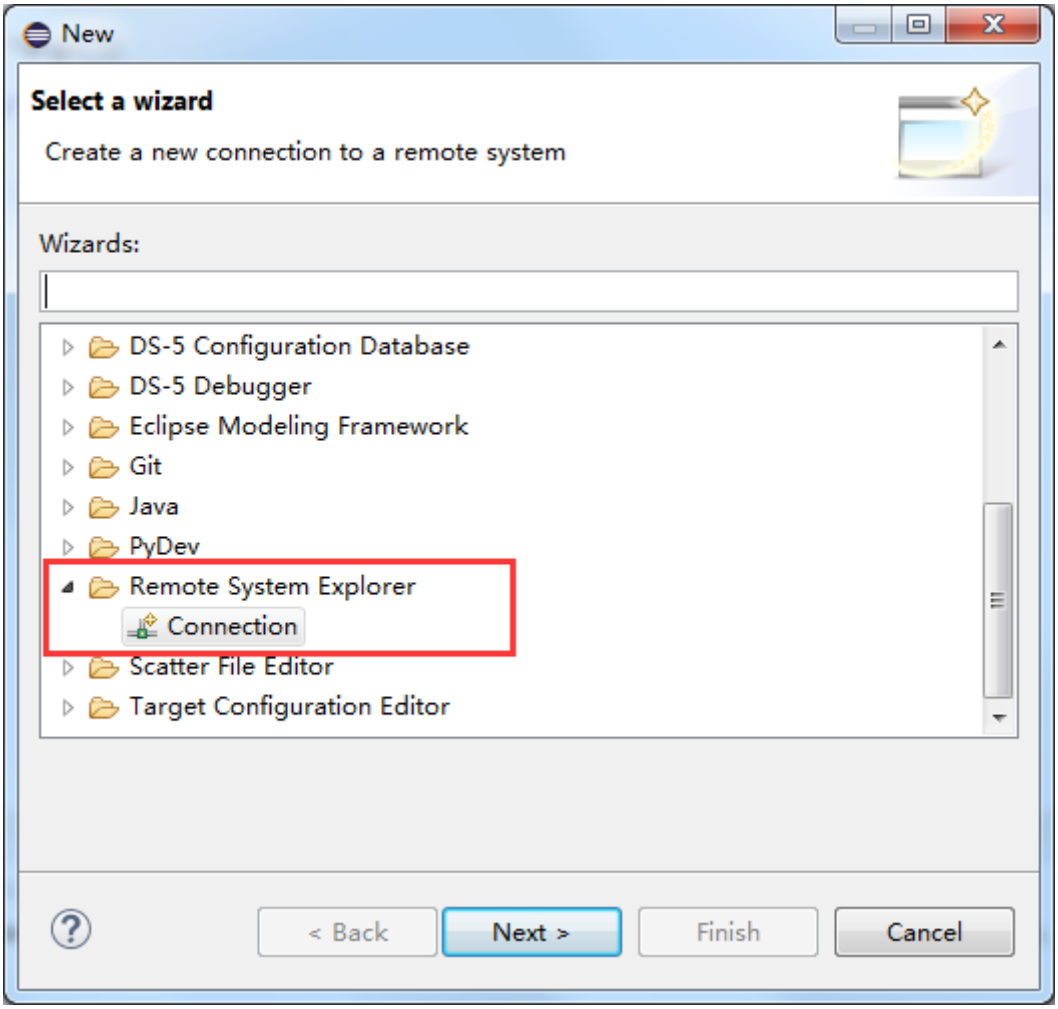


4、建立 SSH 远程连接

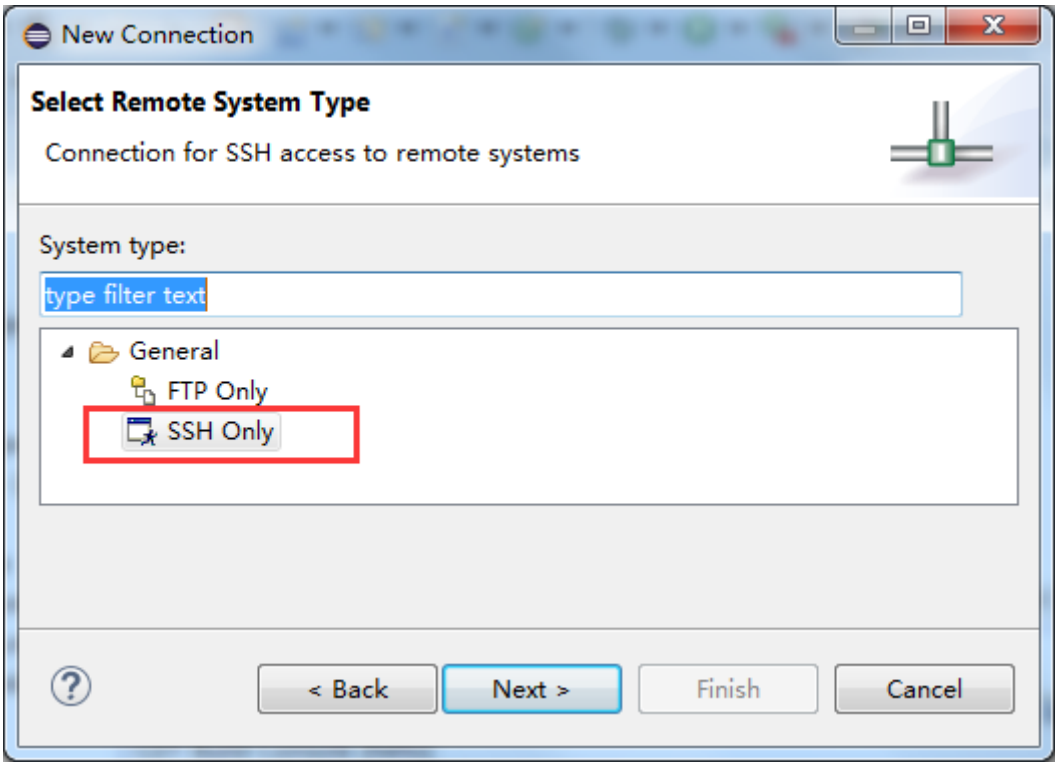
在 Windows 下用 Eclipse 调试应用程序，部署的目标板须能提供通过远程将可执行文件传输到目标板的功能，如 SSH、 FTP 等。这里以 SSH 为例进行介绍。本文首先介绍在 DS-5 软件中利用自带的 SSH 功能完成程序文件的传输。然后会补充介绍一个独立的 SSH 文件传输软件 WinSCP 的配置和使用。

1. 创建远程连接

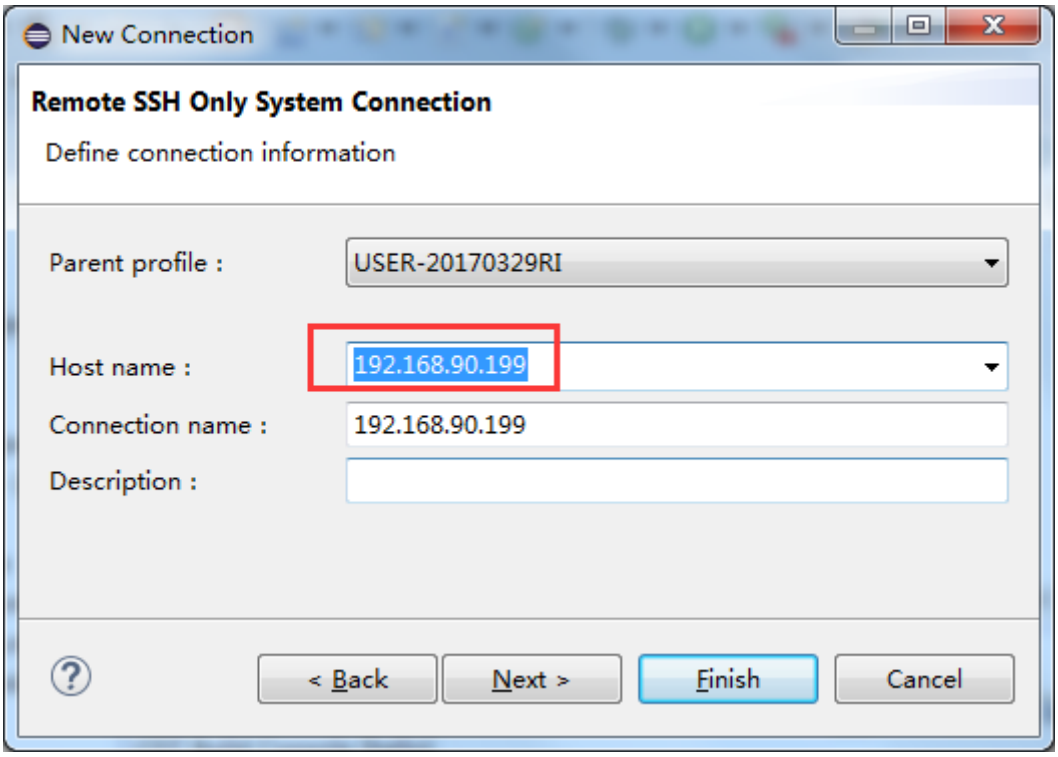
在 Eclipse 主界面，点击 File→New→Other，在如下图所示的窗口选择建立“ Remote System Explorer”，选择 Connection 后点击 Next。



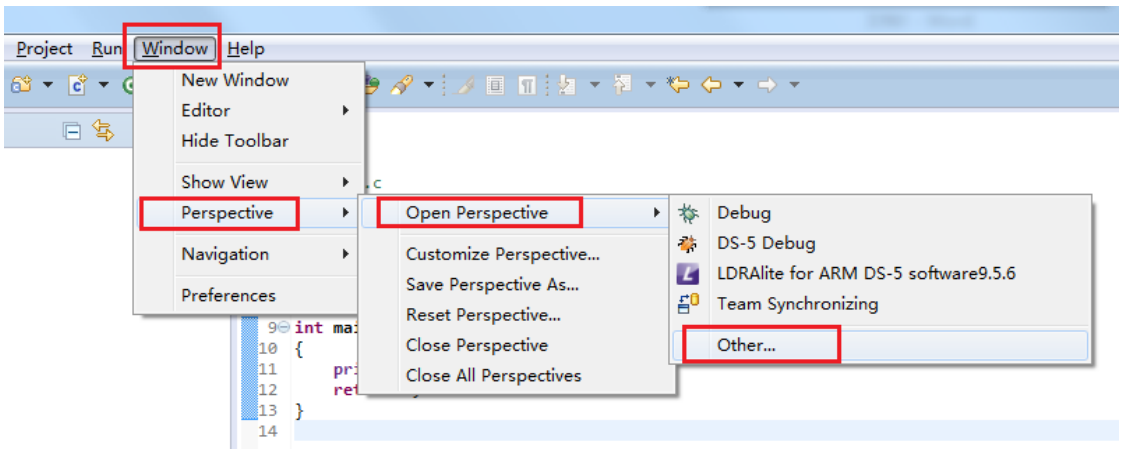
在下图所示的界面中，选择“ SSH Only”，选择建立 SSH 连接。



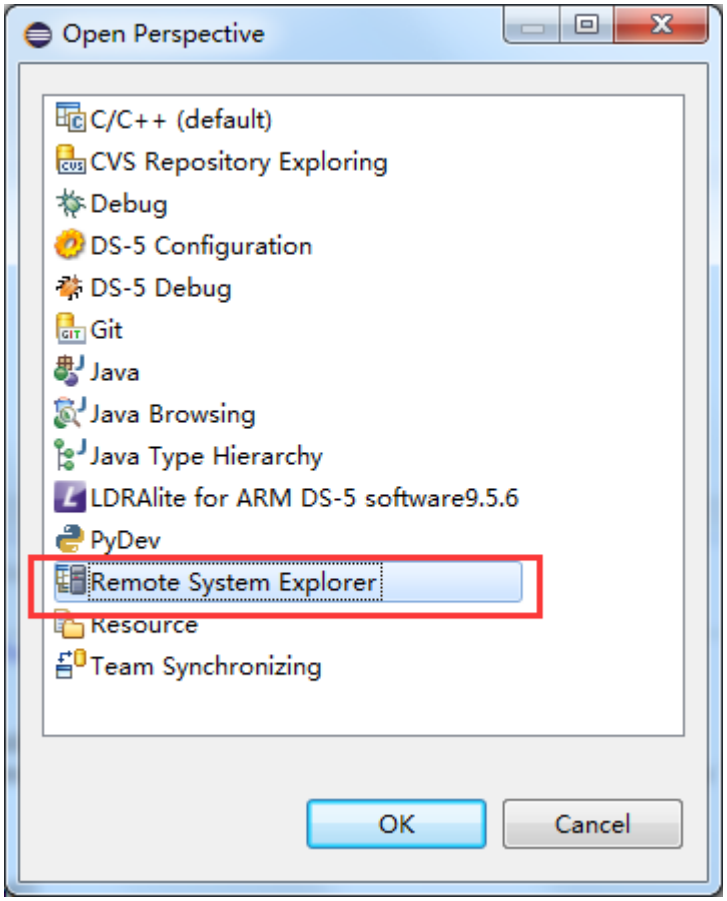
出现如下图所示的 SSH 连接设置界面，在 Host name 栏填写目标板的 IP 地址，如 192.168.90.199， Connection name 栏会自动填写为目标板 IP 地址，也可以修改。在开发本例时，AC501-SoC 开发板的 IP 地址设置为了 192.168.90.199，如果用户自己的板卡上 IP 地址不一样，需要输入实际的 IP 地址，关于如何查看和设置目标板的 IP 地址，参见本书的开发板基本操作章节。点击 Finish 完成设置。



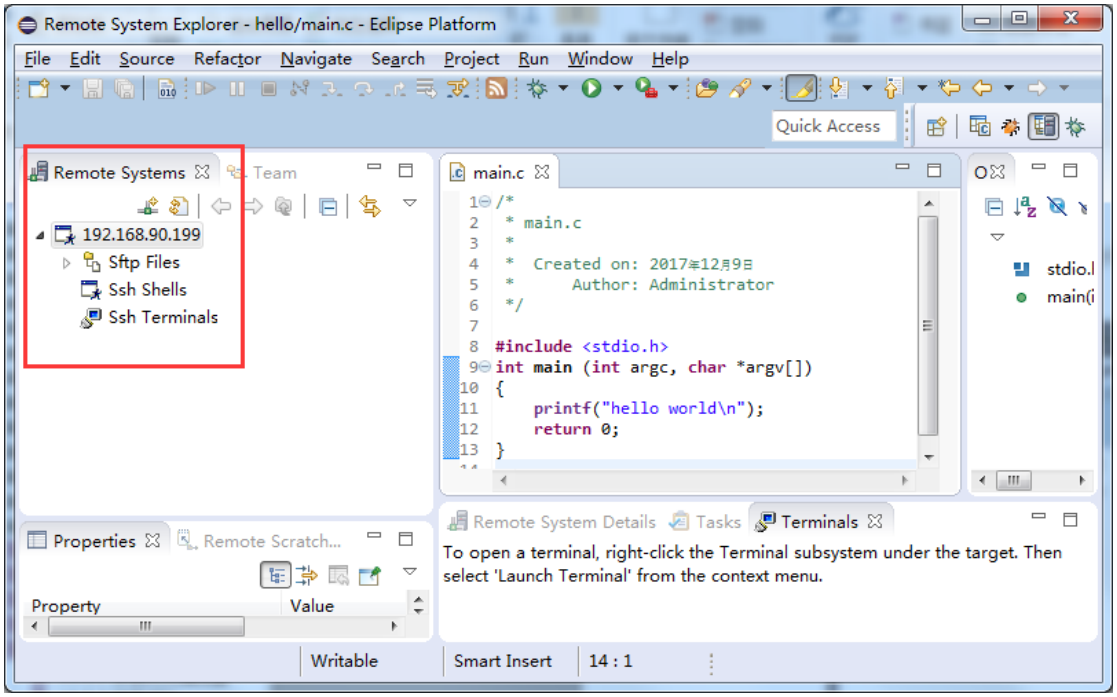
建立完成，依然是 C/C++程序界面视图。点击 Window→Perspective→Open Perspective→Other，如下图所示。



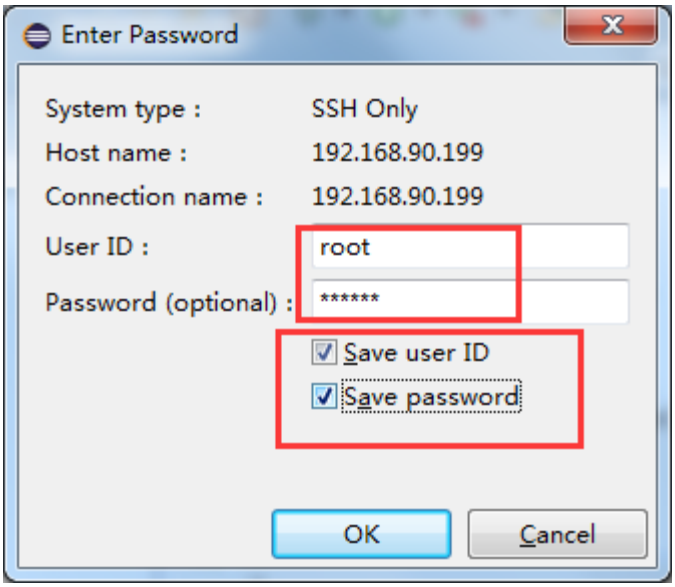
在如下图所示的界面中选择“ Remote System Explorer”，然后点击 OK。



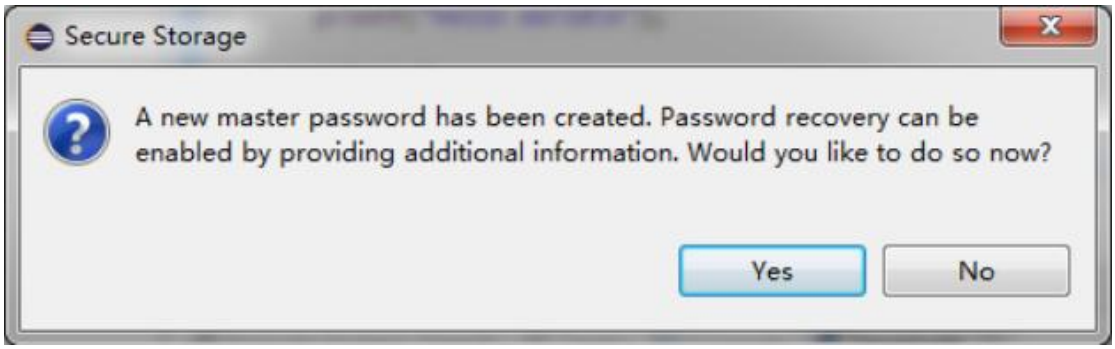
Eclipse 将会切换到远程系统视图，如下图所示，可以看到连接名称为“192.168.90.199”的远程系统。



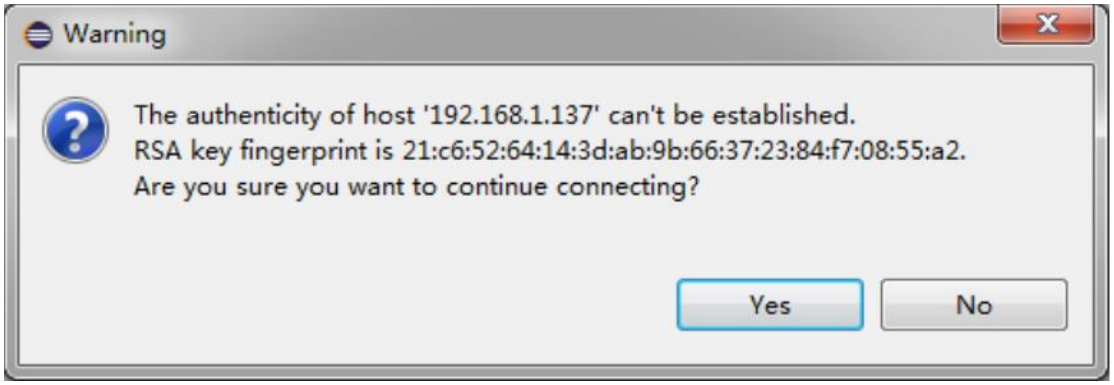
在开始连接前，需要先确保 SoC 板卡已经和 PC 连接在了同一个网段的路由器上，或者 SoC 板卡直接通过网线直连了 PC 的网口。然后右键单击连接名称，选择 Connect，出现 SSH 连接登录设置界面，在其中填写登录名和密码，如下图所示。为了方便以后连接，可选择保存密码。使用 ssh 登录需要有目标主机（这里就是运行 Linux 系统的 SoC 开发板）的用户名和密码，如果目标主机 root 账户没有设置密码，需要先给 SoC 板卡上的 root 账户设定密码。设定密码的方式可参见本书的开发板基本操作章节。



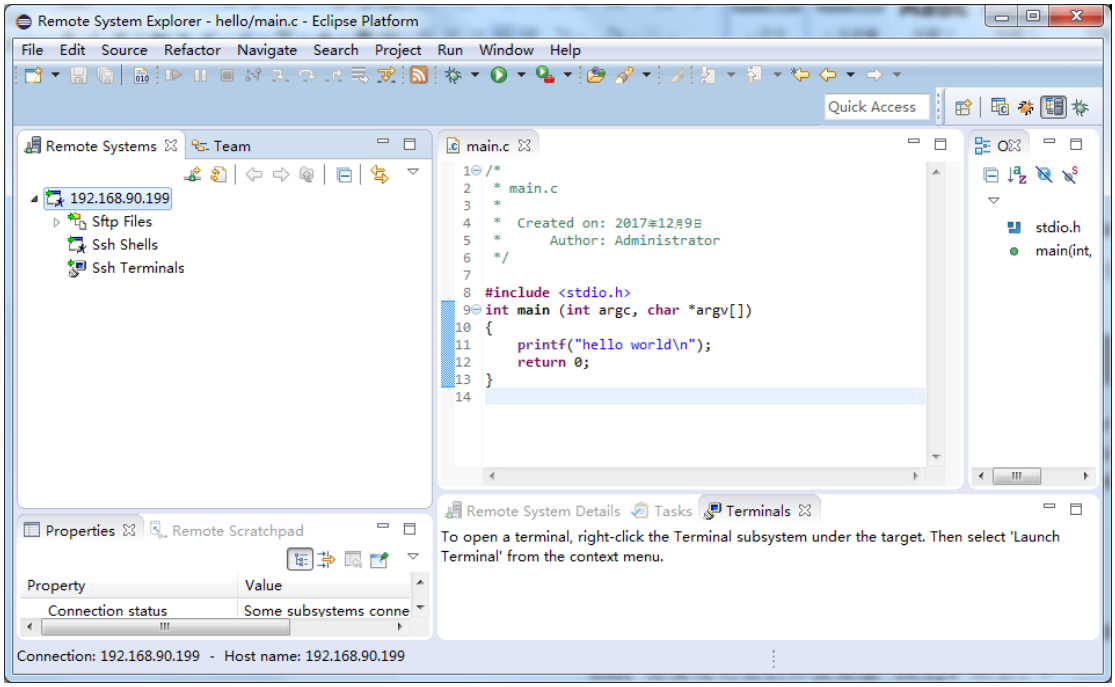
选择保存密码可能会出现如图 10.122 所示的安全提示，点击 No 不填写额外信息。



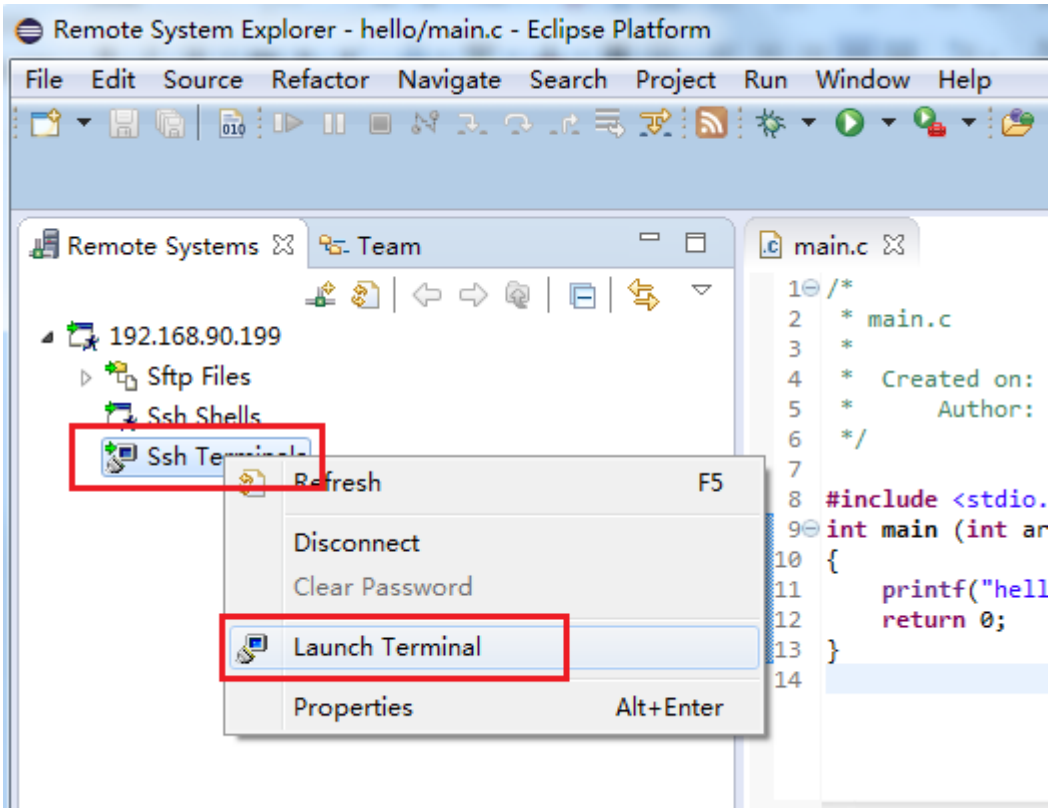
如果 Eclipse 是第一次进行 SSH 连接，可能会出现如下图所示的警告，点击 Yes 并进行操作即可。



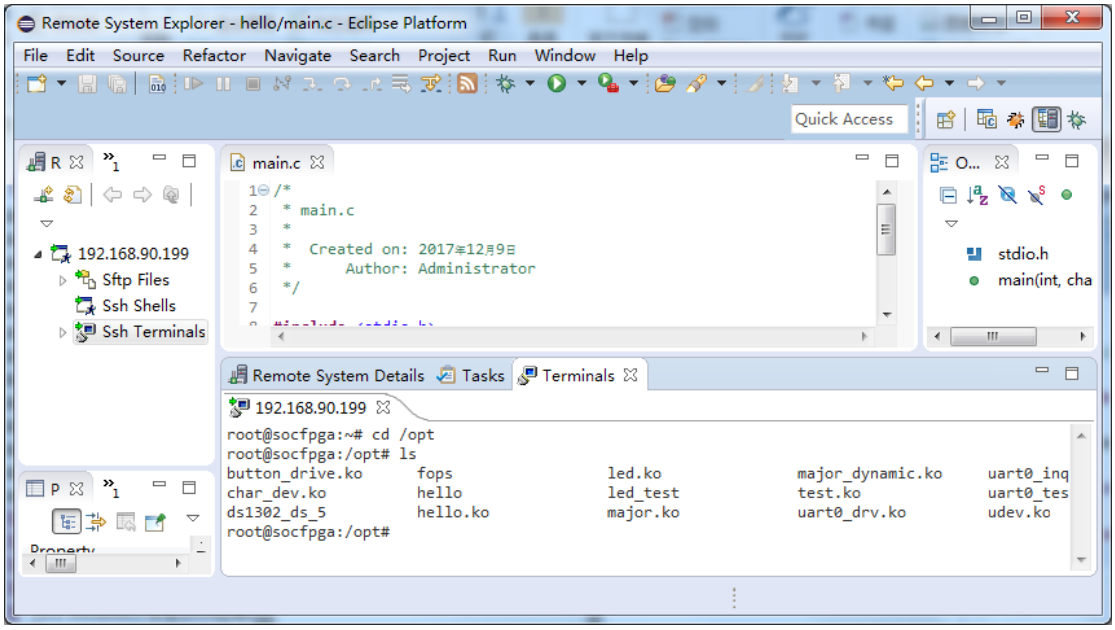
SSH 连接成功后的界面如下图所示，可以看到连接上出现了绿色的标记，表明连接已经建立。



右键单击连接的“ Ssh Terminals”，选择“ Launch Terminal”，如下图所示，选择打开一个远程终端。



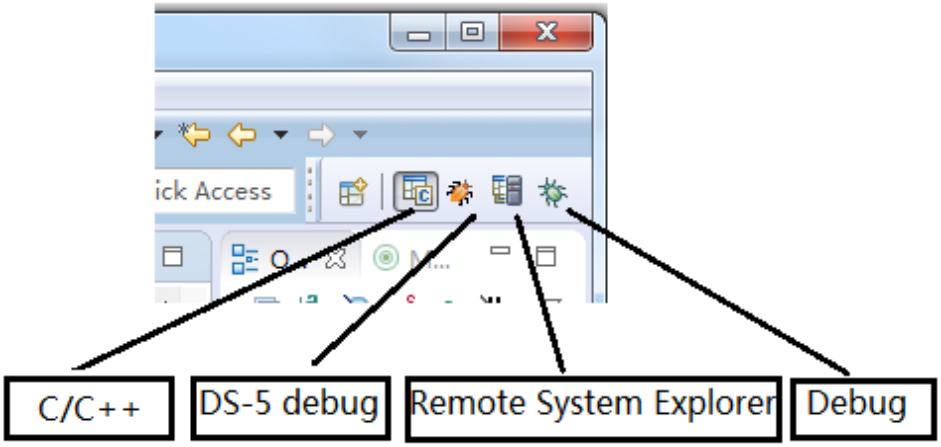
在 Eclipse 界面右下方，将出现一个远程终端，在其中可以输入 Linux 命令进行操作，如下图所示。



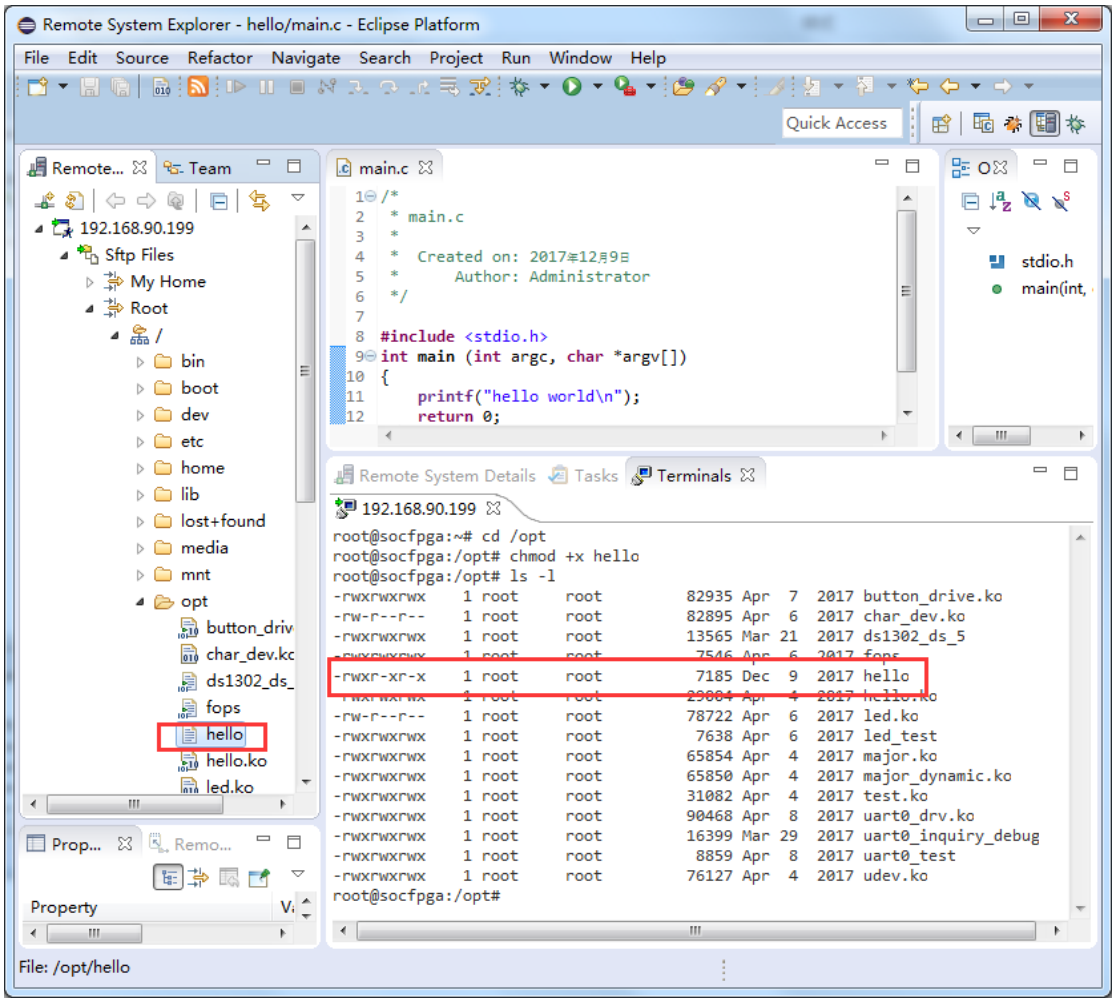
2. 复制文件到目标板

点击右上角监视窗口的 C/C++标签，切换到 C/C++视图，如下图所示。右键点击 hello 工程 Binaries 下的“ hello - [arm/le]”，选择 Copy，复制 hello 文件。

在点击监视窗口的“ Remote System Explorer”标签，切换到远程系统视图，点击展开“ /root”，找到 opt 文件夹，在右键菜单选择 Paste，将已复制的 hello 文件粘贴到目标系统的“ /opt”目录下。



然后在 Terminal 窗口，进入“ /opt”目录，用 chmod 命令为 hello 文件增加可执行权限，输入”chmod 777 hello”或者”chmod +x hello”命令即可实现。操作完成的结果如下图所示。

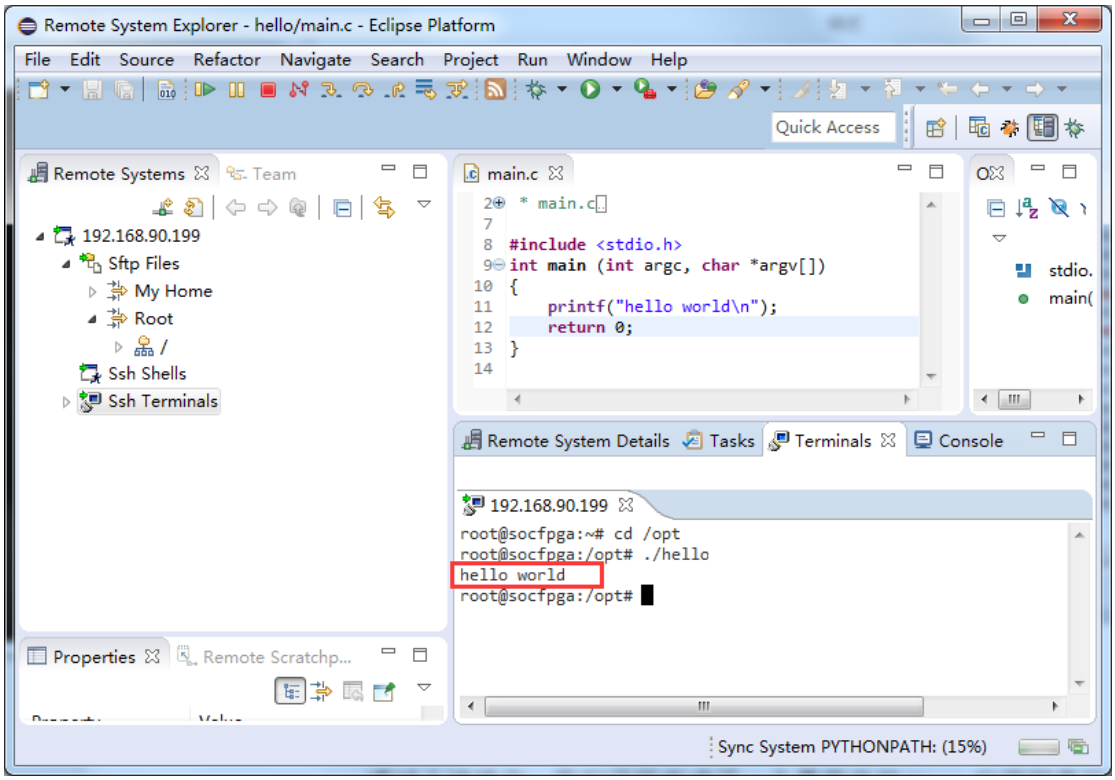


3. 运行应用程序

通过上述操作，我们已经完成了一个最简单的 Linux 应用程序的创建、编译和安装到目标板上，接下来就可以在目标板上执行了，执行的方法非常简单，只需要接着在 Terminal 窗口，输入下述命令即可执行该应用程序。

```
./hello
```

操作完成的结果如下图所示。



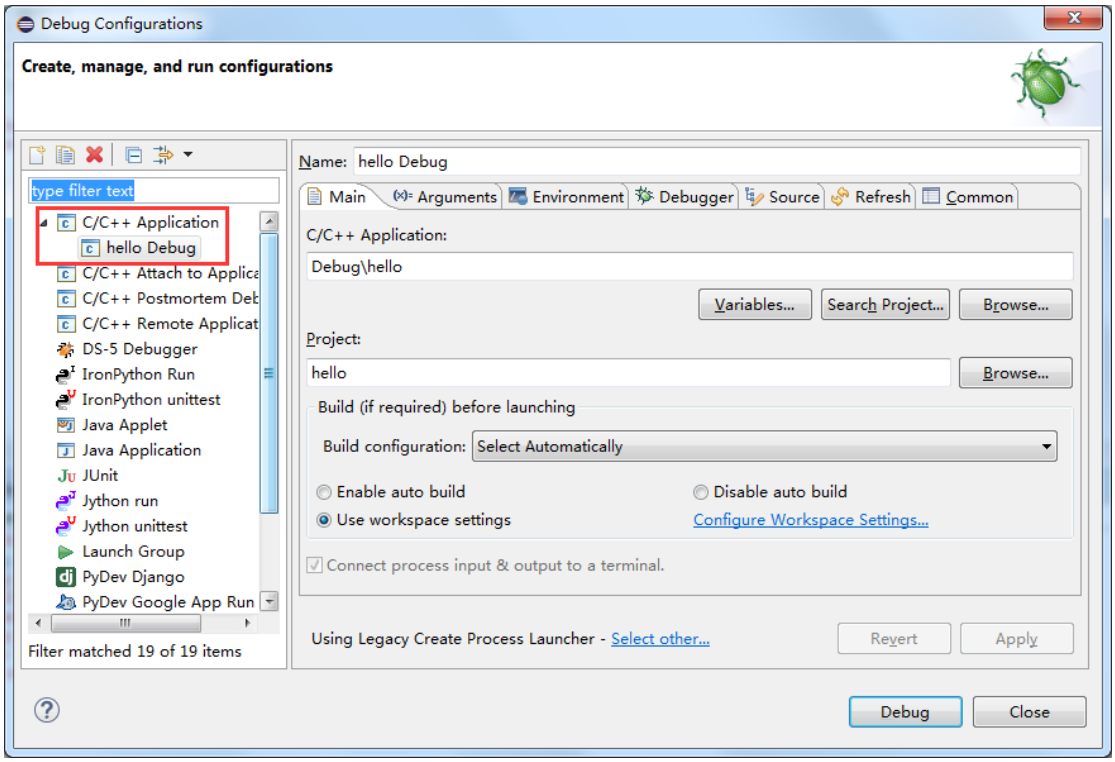
注意，该命令需要在可执行文件所在目录输入才能执行。否则请先使用 cd 命令将路径切换到 opt 目录下。

5、远程调试

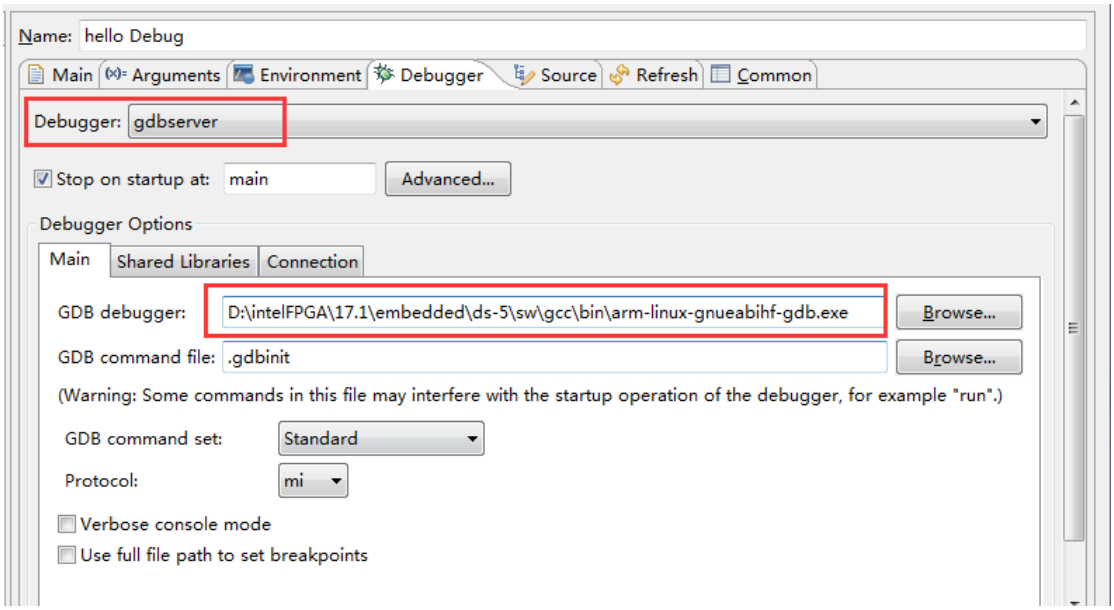
对于编写的软件应用程序，往往由于设计者设计时考虑问题不全面或者其他原因，需要通过查看程序的每一个执行步骤来查找编程中存在的 bug。在开发单片机的程序时，一般使用专用的 JTAG 调试硬件通过 JTAG 口进行调试。而对于基于 Linux 操作系统的应用程序，在调试时，可以通过网络使用 gdb 工具进行调试。接下来将介绍如何使用 gdb 工具对上述编写的 hello 工程进行调试。

1. GDB 设置

打开 Run→Debug Configrations，在调试配置界面，双击“ C/C++ Application”栏，将会生成“ hello Debug”调试目标，如下图所示。



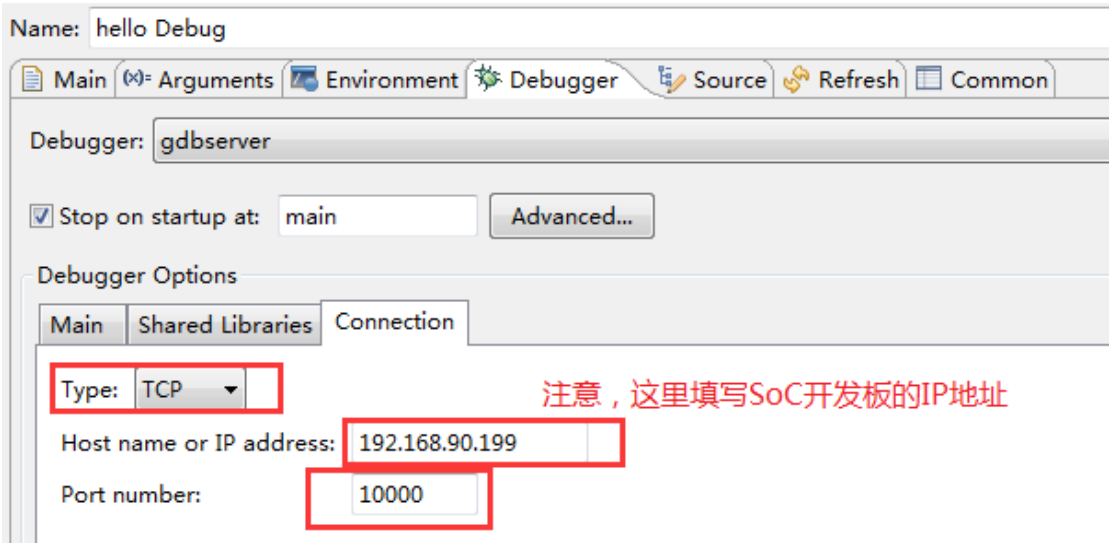
点击 Debugger 选项卡，设置 Debugger 为 gdbserver，并在 GDB Debugger 中浏览到交叉编译器目录下的 arm-linux-gnueabihf-gdb.exe，并设置 GDB command set 为 Standard，如下图所示。



特别说明：DS-5 软件的安装包下默认提供了 gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux 工具链，包括编译工具，但是将用于调试的 arm-linux-gnueabihf-gdb.exe 这个工具给去除了，导致软件默认安装完成后，D:\intelFPGA\17.1\embedded\ds-5\sw\gcc\bin\目录下是没有这个程序的，为了实现

对应用程序的调试，读者可以选择手动设置调试工具。具体方法为从网络上下载 gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux 工具链并将其中包含的 arm-linux-gnueabihf-gdb.exe 文件拷贝到 D:\intelFPGA\17.1\embedded\ds-5\sw\gcc\bin\目录下，然后就可以正常使用了。

点击 Debugger Options 的 Connection 选项卡，设置连接类型为 TCP，在 Host name 栏填写目标板的 IP 地址，在 Port number 栏设置 TCP 连接端口号，如下图所示：



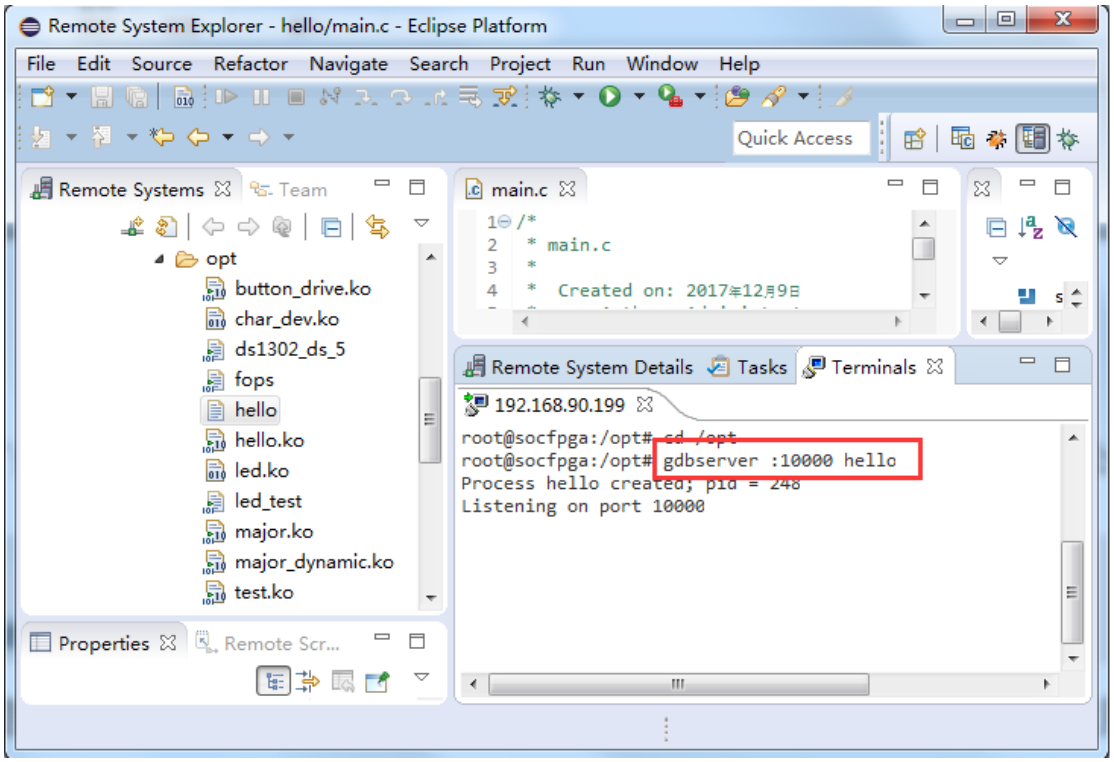
设置完毕后，点击 Apply 按钮，使设置生效，然后点击 Close 关闭窗口

2. GDB 连接和调试

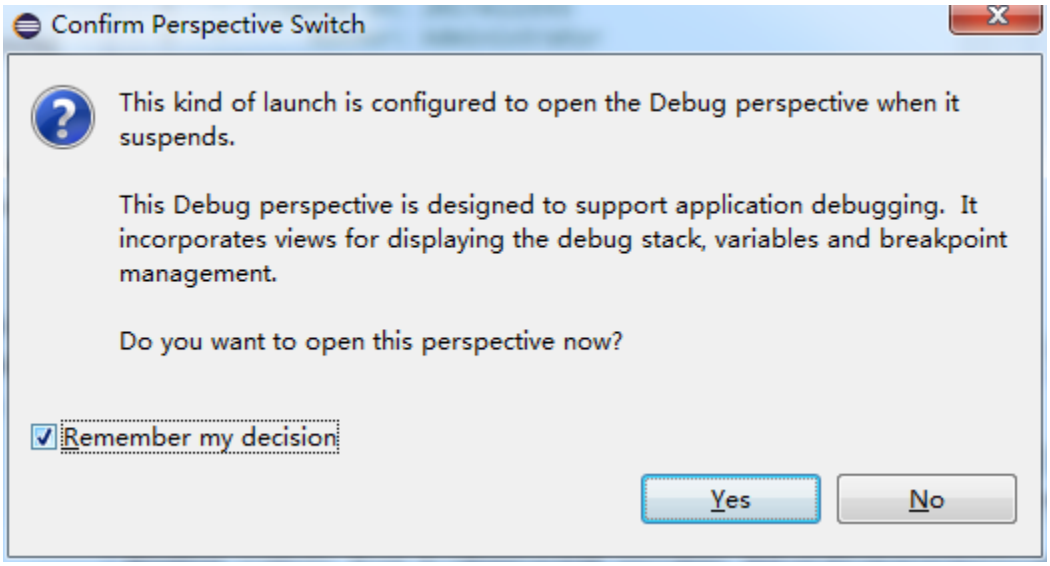
在 Eclipse 主界面，点击监视窗口的 Remote System Explorer，切换到远程系统视图，在终端输入下列命令启动 gdbserver：

```
# cd /opt

# gdbserver :10000 hello
```

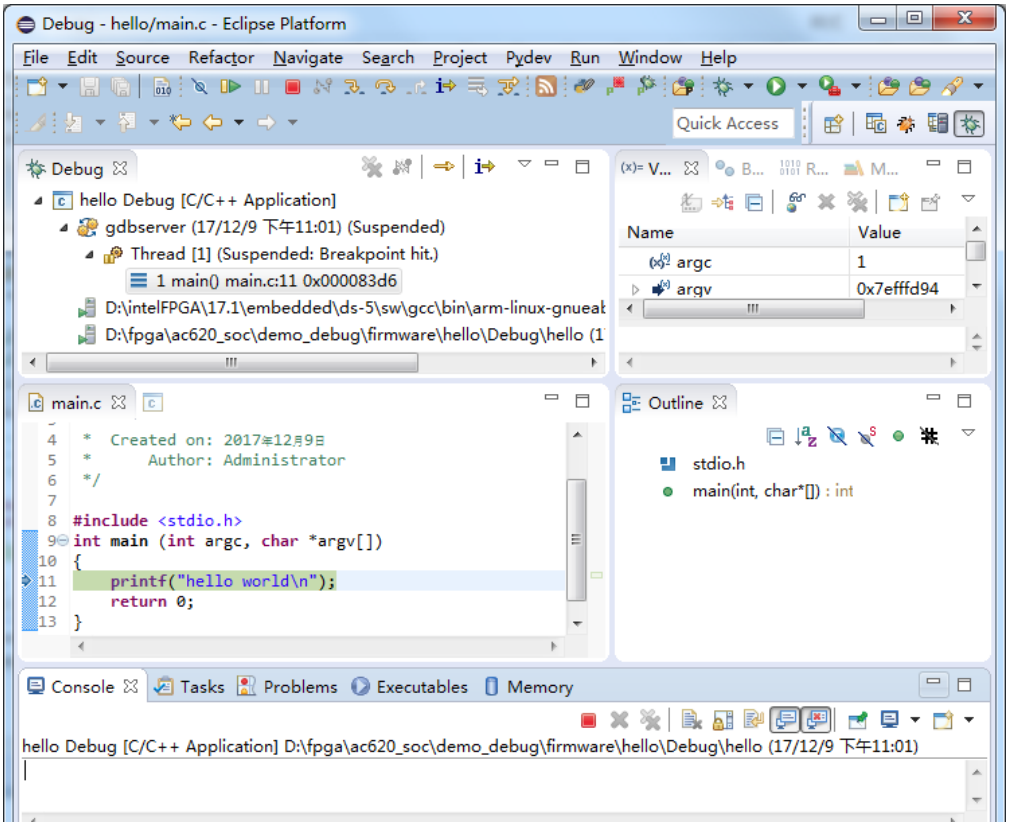
注意 TCP 连接端口必须与 Eclipse 所设定的一致。实际操作截图如下图所示



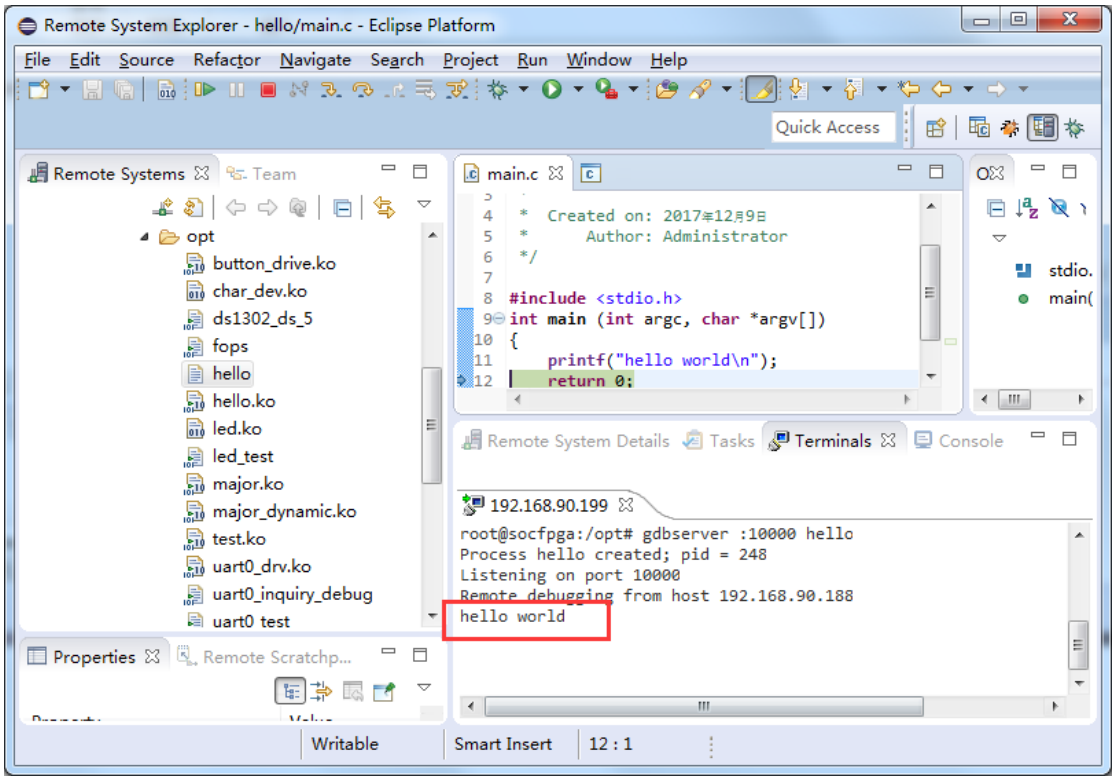
点击 Run→Debug，开始 GDB 远程连接，出现如下图所示窗口，点击 Yes 即可。



最终进入如下图所示的调试界面。



点击 Run→Step Over，或者直接点击 Step Over 图标，单步运行程序，切换到远程系统视图，可以看到终端输出字符串“ hello world”，如下图所示。



调试完毕，点击 Run→Terminate，或终止的快捷图标以停止调试。

6、 总结

至此，就完成了在 SoC 目标板上调试和运行第一个应用程序的操作。当然，本实验仅仅实现了 hello world 信息的打印，并未涉及到任何的硬件外设的操作。但是，通过本实验，读者可以掌握如何在 DS-5 软件中建立基于 Linux 操作系统的应用工程，编译并使用 gdb 工具进行调试，为后续开发复杂的应用程序做好了铺垫。

使用 WinSCP 在 Windows 和 Linux 系统之间传输文件

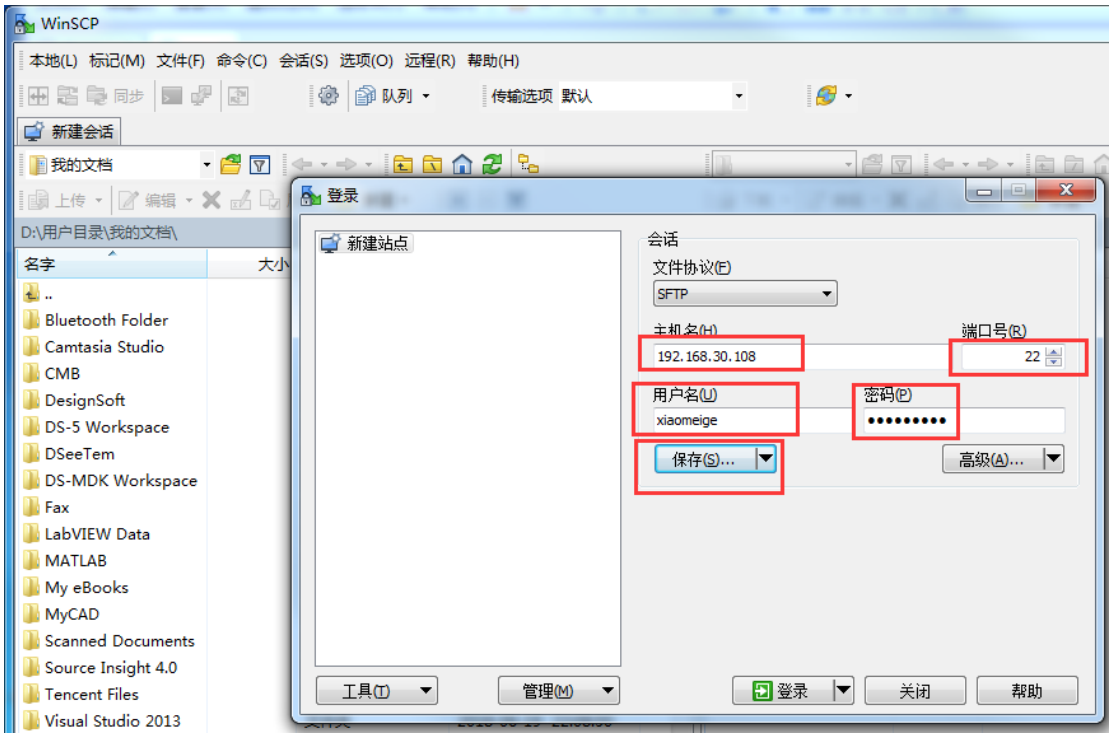
在前一节内容中，讲解了如何在 DS-5 软件中开发和调试 Linux 应用程序，其中讲到了使用 DS-5 自带的 SSH 工具传输文件和命令。该种方案已经很方便了，但是在进行文件拷贝时，需要在几个界面中来回切换，就笔者自己感觉来说，还是比较繁琐，因此在这里介绍另外一款专门用来进行文件传输的工具——WinSCP。

在日常 SoC 开发中，我们经常需要在 Windows 和 Linux 系统之间传输文件，例如在 Windows 系统上的 DS-5 集成开发环境中编写好的 Linux 应用程序需要传递到 Linux 嵌入式开发板中（例如 SoC FPGA 开发板），或者需要将 Linux 系统中的文件拷贝到 Windows 上进一步操作处理，就涉及到两者之间的文件传输。实现上述场景中文件传输的一种比较便捷的方式，是使用 SCP 方式。在 Windows 系统中，可以通过安装 WinSCP 软件来实现上述功能。

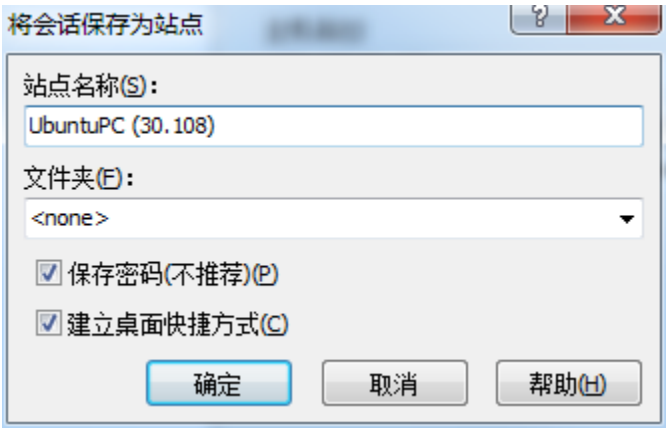
该软件可以在 https://winscp.net/eng/download.php 网址下载得到，光盘资料中提供了下载好的离线安装包 WinSCP-5.13.3-Setup.exe，直接双击即可运行安装。安装过程没有什么需要注意的，一律默认即可。

使用时，如果远程主机没有固定的 IP 和端口映射，则需要 Windows 主机和远程主机处于同一网段，例如连接在同一个路由器上，或者通过网线直连，并设置 IP 在同一网段，否则无法实现连接。

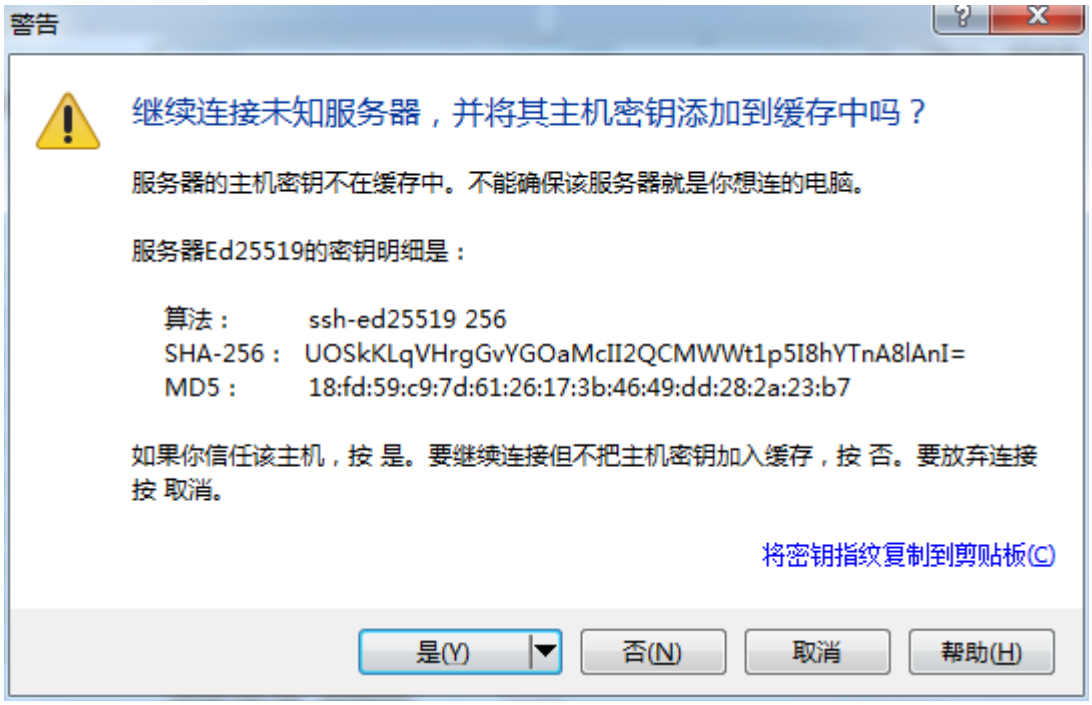
安装完成后运行。首次使用会自动弹出登录界面，在主机名处输入希望连接的主机的 IP 地址，端口号默认 22，用户名和密码输入远程系统的用户名和密码即可。



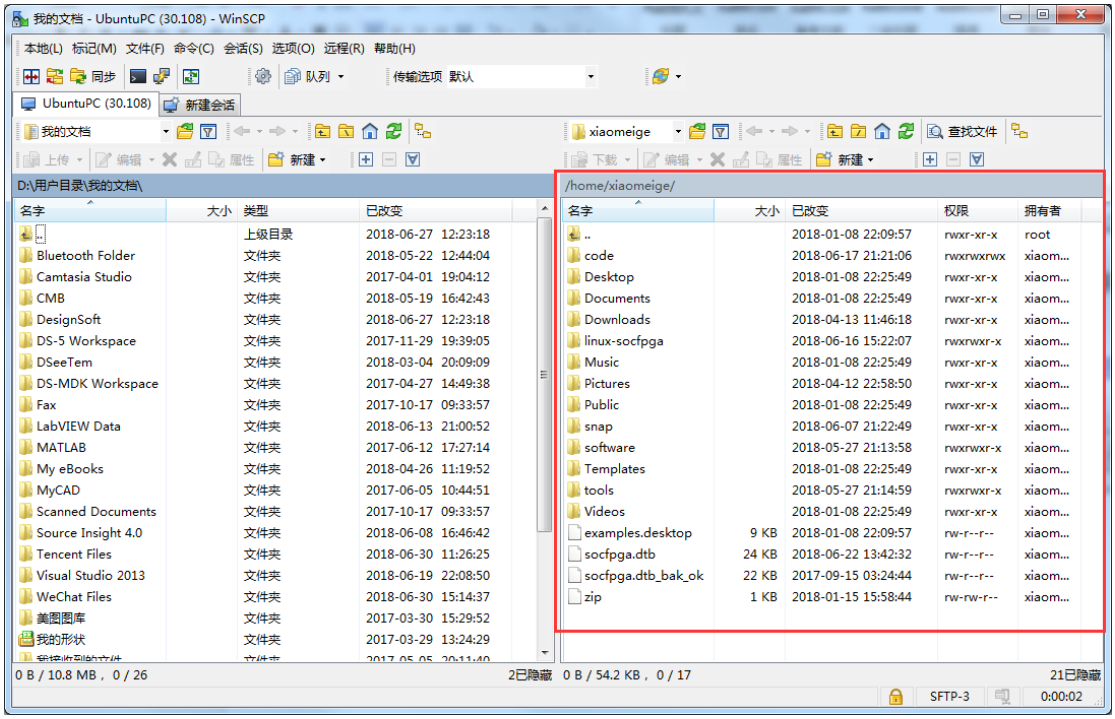
为了下次使用方便，可以点击保存，将该站点保存为常用站点，下次打开时就能快速打开该站点了。如果是在自己的实验电脑上做开发用，不涉及到数据保密安全问题。可以选择保存密码，方便下次快速登录。同时可以勾选建立快捷方式到桌面，这样下次想登录该主机时，直接双击该快捷图标就可以了。



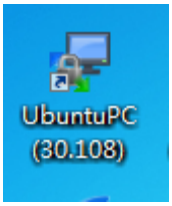
配置完成后，点击登录即可开始连接到远程主机。首次登录一个新主机时，会弹出下述对话框，选择是即可。

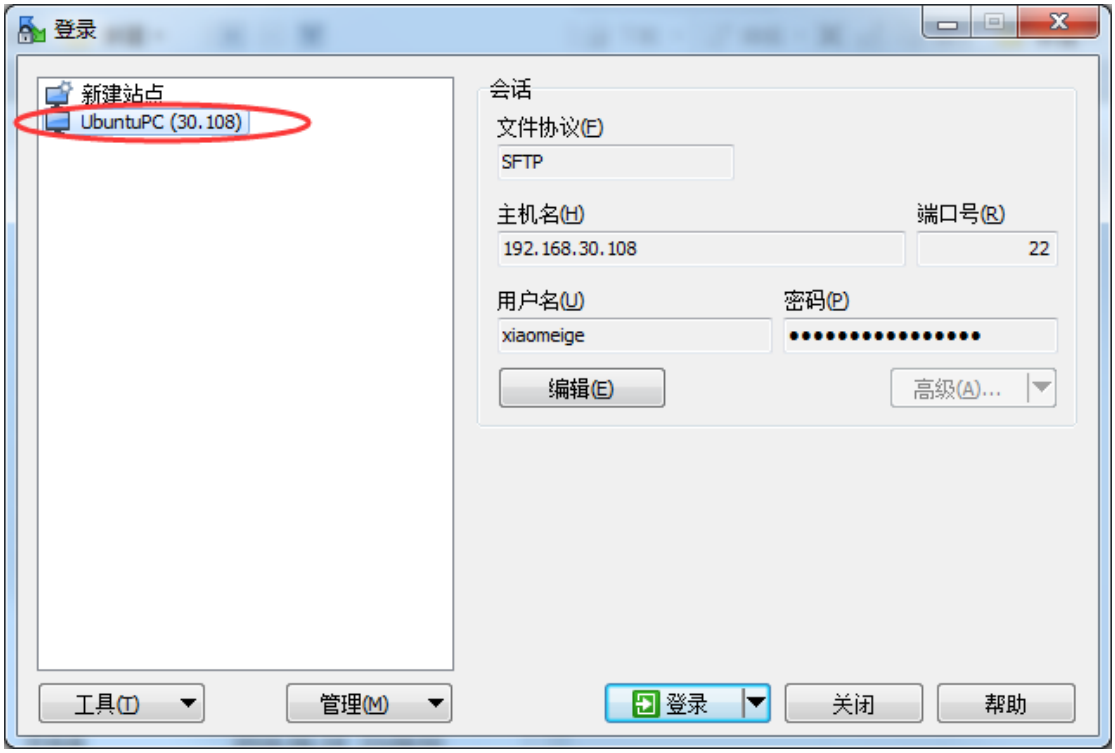


连接完成后，即可在文件浏览窗口的右侧浏览远程主机的文件系统了，左侧是 Windows 系统的资源管理器，在这个浏览器里，可以很方便的通过拖拽的方式将 Windows 中的文件拖动到远程 Linux 主机中，也可以直接从 Linux 主机中将文件或文件夹拖动到 Windows 系统中。使用完毕，直接关闭软件即可自动退出。



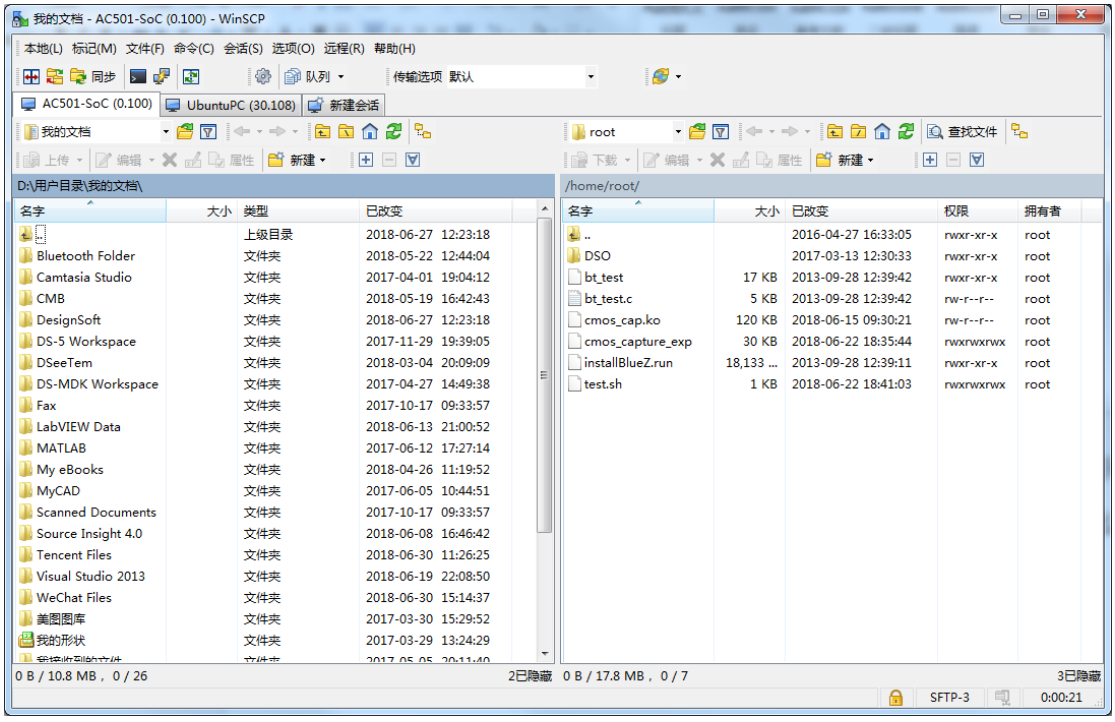
下次要使用时，可以直接在桌面双击保存的快捷方式以快速自动登录，也可以打开 WinSCP 软件，在弹出的对话框中选择已经保存的站点直接登录。





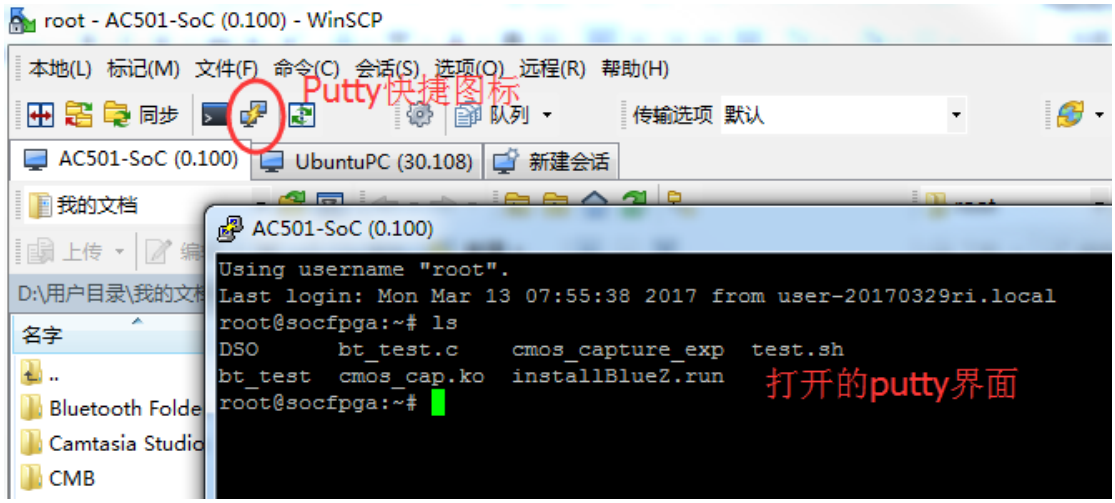
另外，WinSCP 软件可以同时登录多个远程主机，例如，在开发 SoC 时，建立两个远程连接，一个连接到 SOC 开发板的 Linux 系统，一个连接到电脑上的 Ubuntu 虚拟机，这样就可以通过网络分别在多个主机之间互传数据了。

建立多个远程连接时，点击新建站点，输入另一个远程站点的 IP 和用户名以及密码，就可以登录了。下图为同时使用 WinSCP 登录两个远程主机的截图。由于 Ubuntu 主机使用无线网卡联网，Windows 系统也使用无线网卡联网，虚拟机和 Windows 主机网卡使用桥接模式，因此处于同一网段，通过无线网卡能够直接连通。另外，PC 的有线网卡通过网线直接连接到了 SoC 板卡的网口上，通过手动设置两者处于同一网段（PC 的 IP 为 192.168.0.3、SoC 板卡 IP 为 192.168.0.100），则两者也能够顺利通信。



通过 WinSCP 工具，在以后的开发中，需要在虚拟机，Windows 系统、SoC 开发板中互相传输文件就非常方便了。无需设置 NFS 挂载，也无需使用 U 盘作为中间传输介质。

另外，该软件还可以调用 PuTTY 以实现 Shell 终端连接，执行各种命令。该功能需要用户的电脑 C:\Program Files (x86)\PuTTY\路径下存在 putty.exe 软件，如果没有的话，自己建立该路径，将 putty 软件放置进去即可。putty 准备好之后，只需要选中希望连接 shell 的远程主机，然后点击 putty 快捷图标即可。如下图所示：



实际使用时，笔者更倾向于使用 WinSCP 软件来完成各个系统之间的数据传输，毕竟其更加的方便，还可以独立工作，不用打开 DS-5 就能完成多个系统之间的文件传输。

基于虚拟地址映射的 Linux 硬件编程

在 AC501_SoC_GHRD 工程中，我们在 Platform Designer 集成开发工具中为 HPS 添加了若干个外设，包括 PIO、UART、SPI、IIC、Frame Buffer 等。这些 IP 外设，通过 Lw_H2F_AXI bridge 或 F2H_AXI bridge 连接到了 HPS 的内部总线上，能够被总线上的主机（HPS、MPU、DMA）直接寻址。因此，从处理器操作外设的基本原理来看，就是通过寻址的方式，让总线上的主机（HPS、MPU、DMA）能够直接读写指定地址存储的数据。

在基于嵌入式 Linux 的系统开发中，对于各种硬件外设的操作一般通过编写标准的 Linux 驱动程序来完成。在驱动程序中对外设的寄存器进行读写操作，然后提供给用户应用层标准的 UNIX 编程接口。但是此种方式对开发人员的要求较高，即要求开发人员具备 Linux 驱动开发的能力，对于使用 SoC FPGA 开发板的用户，一般多对 FPGA 开发了解的多一点，而对 ARM Linux 开发知识掌握的并不深，直接上手开发 Linux 驱动程序有较大的难度。针对这一现象，本书介绍了两种解决方案。第一种方案为使用虚拟地址映射的方式将外设寄存器映射到 Linux 用户空间，第二种为配置 Linux 内核，打开对添加的外设 IP 的驱动支持，即使用系统自带的驱动程序直接驱动外设 IP。

什么是虚拟地址映射

SoC FPGA 上的 HPS 组件是一个集成了双核的 ARM Cortex-A9 处理器、MPU 存储管理单元、DDR3 控制器的高性能硬件处理系统。其与普通的单片机、DSP、NIOS II 处理器对总线的寻址有较大的区别。对于普通的单片机、DSP、NIOS II 处理器，都是没有 MPU 的，所有的地址都由 CPU 直接指定，例如对于 STM32F103RBT6 单片机，USART1 外设挂载在外设总线的 0x3800 地址，而外设总线又是从内存总线的 0x40000000 地址上开始的，因此，如果 CPU 想直接操

作 USART1 外设的第 N 个寄存器，只需要对 0x40000000+0x3800+n 这个地址进行读写操作即可。

对于 HPS，如果使能了 MPU，那么总线是挂在 MPU 上的，CPU 想对总线上的某个地址进行寻址，需要先知道对应总线在 MPU 上的起始地址，然后再计算外设在该总线上的偏移地址，这样得到的地址被称为虚拟地址，该虚拟地址就是 CPU 能够直接读写的地址，CPU 直接读写该虚拟地址，就能够访问到外设的寄存器了。

即相较于普通的不含 MPU 的处理器，在使能了 MPU 的情况下，HPS 要想直接访问到某外设的寄存器，必须先进行虚拟地址映射，先将 MPU 映射到用户空间，得到 MPU 的虚拟地址。总线上每个外设的地址，是无法在程序运行前通过计算得到的，只能在程序运行后，在 MPU 虚拟地址上加上总线相对于 MPU 的偏移和外设基地址相对于总线的偏移才能得到。而一旦完成了虚拟地址映射，CPU 访问外设寄存器就像直接访问内存总线上的某个地址一样方便了。这样在编写 Linux 应用程序的时候通过简单的操作完成虚拟地址映射，就能够非常方便的去操作这些外设 IP 了，无需再编写 Linux 内核驱动程序，降低了开发难度。

虚拟地址映射的实现

在 Linux 系统中实现虚拟地址映射主要包含三个步骤，第一为得到存储器管理单元（MPU）的虚拟地址，第二为通过添加 h2f_lw_axi_master 桥在 MPU 上的偏移地址来得到 h2f_lw_axi_master 桥的虚拟地址，第三则是添加 FPGA 侧具体外设在 h2f_lw_axi_master 桥上的偏移地址来得到该外设在 Linux 系统中的虚拟地址。

在 Linux 系统中，通过 mmap()函数来实现虚拟地址的映射。函数原型为：

```
void *mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

该函数会在调用程序的虚拟地址空间 addr 处创建设备文件 fd，偏移地址 offset 的一个长度为 length 的映射，映射内核地址空间一段内存到用户进程地址空间。

要想将 MPU 映射到用户空间，需要先打开 MPU 以得到其描述符，该操作可以通过 open 函数实现，如下所示：

```
if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
    printf( "ERROR: could not open \"/dev/mem\"...\n" );
    return( 1 );
}
```

即当 Linux 系统正常工作时，会在 dev 目录下存在一个名为 mem 的设备，该设备就是 MPU 了，通过 open 函数打开该设备，得到其文件描述符，然后再使用该文件描述符来完成 h2f_lw_axi_master 总线的虚拟地址映射。

```
    void *periph_virtual_base;
    periph_virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ |
PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );
```

程序中，首先定义了一个 void 类型的指针，名为 periph_virtual_base，即 periph_virtual_base，外设虚拟地址的意思，然后使用 mmap 函数将 MPU 上偏移地址为 HW_REGS_BASE 的地址映射为虚拟地址，并赋值给 periph_virtual_base 指针。映射的地址空间长度为 HW_REGS_SPAN 大小，映射方式为可读可写。其中，HW_REGS_BASE 和 HW_REGS_SPAN 为宏定义，是与 HPS 中硬件外设相关的两个信息。

在 altera 给出的官方参考设计中，是这样来定义 HW_REGS_BASE 和 HW_REGS_SPAN 的：

```
#define HW_REGS_BASE (ALT_STM_OFST )
#define HW_REGS_SPAN (0x04000000 )
```

可以看到HW_REGS_BASE又被指向了一个名为ALT_STM_OFST的定义，该定义在hps.h中有定义：

```
#define ALT_STM_OFST 0xfc000000
```

即 HW_REGS_BASE 实际是被指定了一个确定的地址 0xfc000000。那么这个地址又具体是什么意义呢？这就要先从 SoC FPGA 中各个外设的地址分配说起，在 HPS 侧，有一段地址段被专门用作了外设地址段（peripherals region），该地址段是 HPS 统一内存空间最顶端的 64MB 空间，即从 0xfc000000~0xffffffff。在这段地址区间内，每一个地址段都被分配给了 HPS MPU 子系统的一个确定的外设，而在这个地址段中，第一个外设就是 STM 模块，详见表 xxx。即 STM 模块的基地址也就是整个外设区地址段的基地址，因此在这里将 HW_REGS_BASE 定义为 ALT_STM_OFST，实际上只是借用了 STM 的基地址来表达整个外设地址段的基地址，并不是说特指 STM 模块。

而 HW_REGS_SPAN 参数则被直接定义为了 0x04000000，也就是 64MB 的大小，刚好也就是整个外设地址段的地址空间大小。

所以，这段程序的意思就是将 MPU 上的整个外设地址段空间，共 64MB 的内容映射为虚拟地址，并赋值给定义的 periph_virtual_base 指针。

对于我们在 paltform designer 中添加的 IP，都是连接到了 lw_h2f_bridge 上，而 lw_h2f_bridge 本身就是处在这个外设地址段中的，表 xxx 中的 LWFPGASLAVES 就是这个桥，该桥的基地址为 0xff000000，共 2MB 的地址空间。而每一个连接在 lw_h2f_bridge 上的 FPGA 侧外设都有一个基地址，该地址可以在生成的 hps_0.h 的头文件中看到，所以，一个特定外设的地址在 Linux 用户空间的虚拟地址就是整个外设地址段映射得到的虚拟地址加上 lw_h2f_bridge 的偏移地址，再加上 lw_h2f_bridge 上该外设的基地址。

例如，对于 GHRD 工程中的 led_pio 核和 button_pio 核，其经过映射后的虚拟地址就应该为：

```
led_pio_virtual_base =
     periph_virtual_base + ( ( unsigned long  )( ALT_LWFPGASLVS_OFST +
           LED_PIO_BASE ) & ( unsigned long)( HW_REGS_MASK ) );

button_pio_virtual_base =
     periph_virtual_base + ( ( unsigned long  )( ALT_LWFPGASLVS_OFST +
          BUTTON_PIO_BASE ) & ( unsigned long)( HW_REGS_MASK ) );
```

其中LED_PIO_BASE和BUTTON_PIO_BASE 为led_pio和button_pio外设在lw_h2f_bridge上的基地址，这是两个宏定义，在hps_0.h文件中定义，关于hps_0.h文件，将在后文有介绍。而ALT_LWFPGASLVS_OFST即为lw_h2f_bridge在外设地址空间中的基地址。

外设从设备标识	说明	基地址	地址空间
STM	Space Trace Macrocell	0xFC000000	48 MB
DAP	Debug Access Port	0xFF000000	2 MB
LWFPGASLAVES	FPGA slaves accessed with lightweight HPS-to-FPGA bridge	0xFF200000	2 MB
LWHPS2FPGAREGS	Lightweight HPS-to-FPGA bridge global programmer's view (GPV) registers	0xFF400000	1 MB
HPS2FPGAREGS	HPS-to-FPGA bridge GPV registers	0xFF500000	1 MB
……	……	……	……

表 xxx 出自 Cyclone V Device Handbook 第 3 卷第一章的 HPS Address Map 节的 HPS Peripheral Region Address Map 小节。

基于虚拟地址映射的 PIO 编程应用

前面提到，在 AC501_SoC_GHRD 工程的 qsys 设计文件中，添加了两个 PIO 类型的外设，分别为 2 位的仅输出型 PIO，用以驱动 LED，和 2 位的仅输

入型 PIO，用以连接轻触按键。本实验，将针对添加的这两个外设，在 DS-5 中编写应用程序，首先完成虚拟地址映射，然后通过虚拟地址，读写 PIO 的寄存器，完成按键控制 LED 灯的功能。通过该实验，展示通过虚拟地址映射方式操作外设的基本方法。

PIO 外设的虚拟地址映射

完整的 led_pio 和 button_pio 外设的虚拟地址映射代码如下所示：

```
static volatile unsigned long *led_pio_virtual_base = NULL; //led_pio虚拟地址
static volatile unsigned long *button_pio_virtual_base = NULL;//button_pio虚拟地址

int fpga_init(long int *virtual_base) {
    int fd;
    void *periph_virtual_base;    //外设空间虚拟地址

    //打开MPU
    if ((fd = open("/dev/mem", ( O_RDWR | O_SYNC))) == -1) {
        printf("ERROR: could not open \"/dev/mem\"...\n");
        return (1);
    }

    //将外设地址段映射到用户空间
    periph_virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE),
            MAP_SHARED, fd, HW_REGS_BASE);
    if (periph_virtual_base == MAP_FAILED) {
        printf("ERROR: mmap() failed...\n");
        close(fd);
        return (1);
    }

    //映射得到led_pio外设虚拟地址
    led_pio_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + LED_PIO_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
    //映射得到button_pio外设虚拟地址
    button_pio_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + BUTTON_PIO_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
    *virtual_base = periph_virtual_base;  //将外设虚拟地址保存，用以释放时候使用
    return fd;
}
```

在 DS-5 中建立 PIO 应用工程

通过上述程序，完成了 led_pio 和 button_pio 外设的虚拟地址映射，接下来就可以使用该虚拟地址来操作这两个外设了。

首先参照上一章——使用 DS-5 编写和调试 SoC 的 Linux 应用程序的内容，建立好基本的工程。或者直接在 DS-5 中，选中已有的工程，然后右击，复制粘贴为新的工程，在粘贴时修改为新的工程名即可。如图 xxx 和图 xxx 所示：

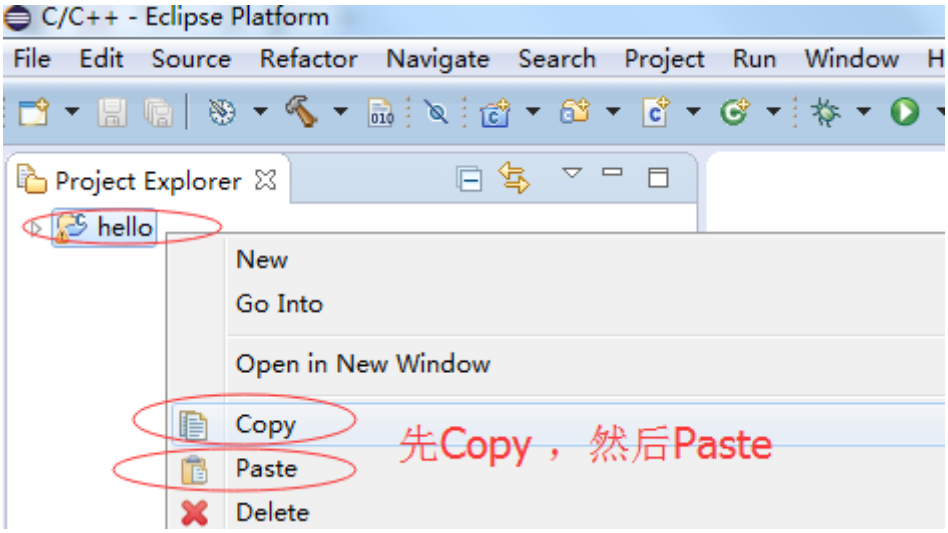


图 xxx 复制粘贴已有工程

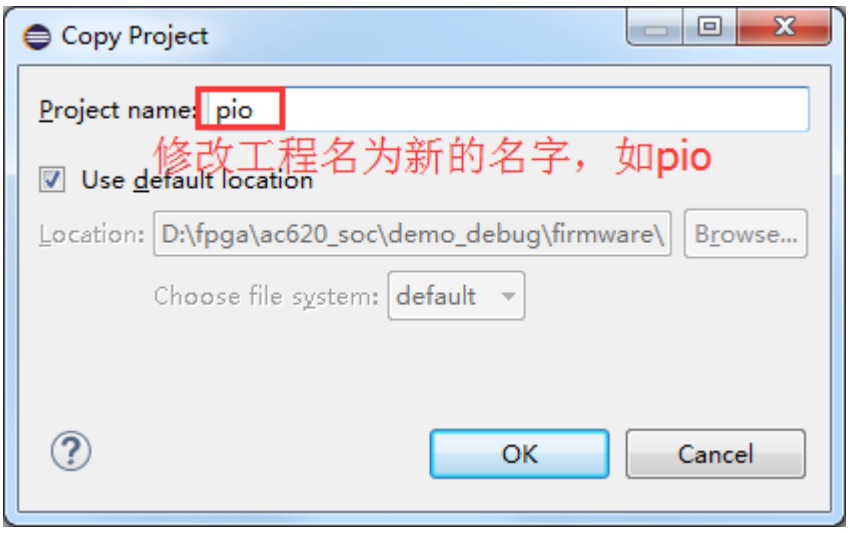


图 xxx 修改工程名为新的工程名

复制完成之后，先将新工程下的 Debug 目录删除，因为这些文件是之前工程编译出来的，在新工程中既不会被使用，也不会被自动删除，因此需要手动删除。

添加和包含 HPS 库文件

由于第一个实验只是简单的使用 Console 打印了一个“hello world”，是一个最简单的应用程序，与特定外设硬件没有任何关系，因此在创建该工程时，并未添加任何额外的包含文件。而在本例操作 led_pio 和 button_pio 时，就涉及到了 SoC FPGA 中 HPS 的专用外设了，因此需要添加与基本的 HPS 硬件信息相关的头文件。需要包含的基本头文件主要有 3 个，分别为：

```
#include "hwlib.h"
#include "socal/socal.h"
#include "socal/hps.h"
```

由于这些库是 SoC EDS 软件提供的，DS-5 中默认并没有包含该库，所以如果直接在程序中包含这些文件，DS-5 会提示找不到文件，因此需要在工程中设置头文件包含路径。

在 DS-5 中选中 pio 工程，鼠标右击选择 Properties，或者直接选中 pio 工程后，输入键盘组合键 Alt+Enter，打开工程属性对话框。点击 C/C++ General 下的 Paths and Symbols 选项，如图 xxx 所示：

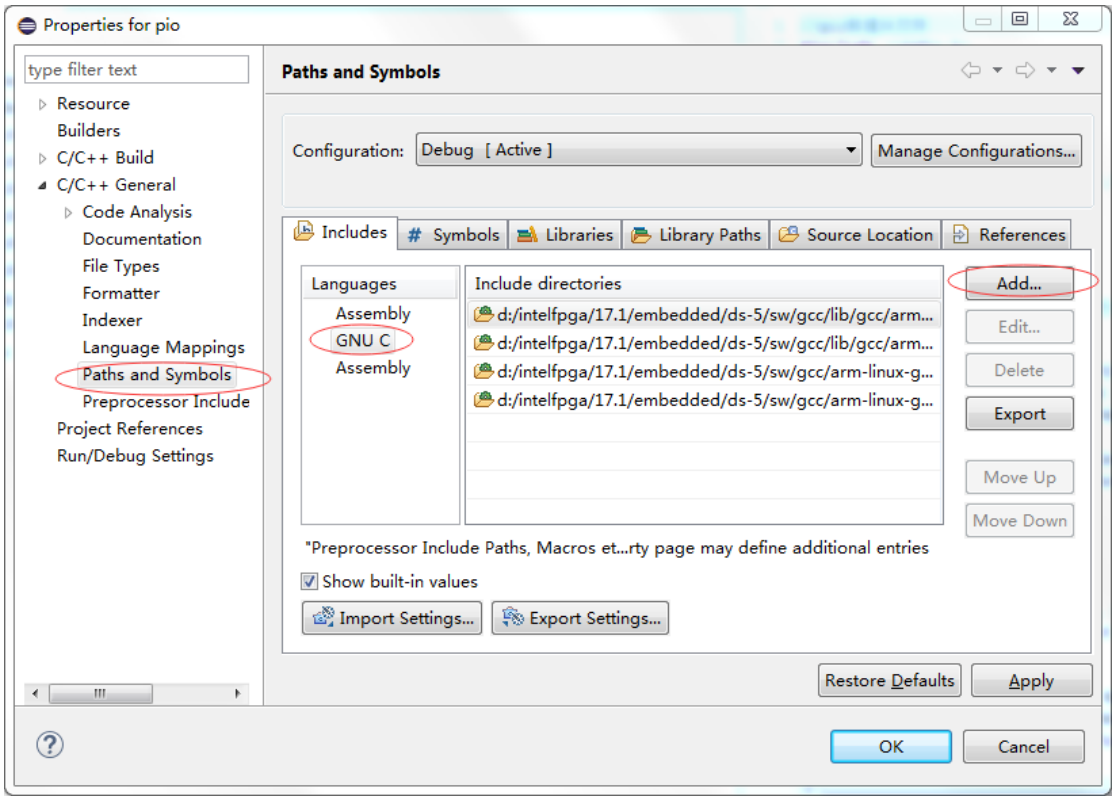


图 xxx 包含路径选项卡

在右侧的选项中，点击 GNU C 一栏，可以看到已经默认包含了很多的库文件。点击 Add 按钮，在弹出的对话框中输入用户电脑中 hwlib 库的头文件包含路径，例如笔者电脑上的位置为：D:\intelFPGA\17.1\embedded\ip\altera\hps\altera_hps\hwlib\include，勾选上 Add to all configurations 和 Add to all languages 选项，然后点击 OK 即可，如图 xxx 所示：

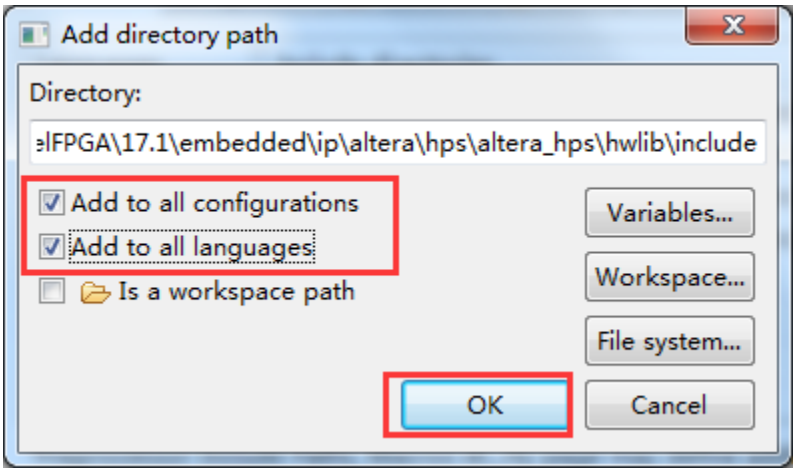


图 XXX 添加包含路径

使用通样的操作将 D:\intelFPGA\17.1\embedded\ip\altera\hps\altera_hps\hwlib\include\soc_cv_av 路径添加到包含路径中。然后点击 OK 即可。

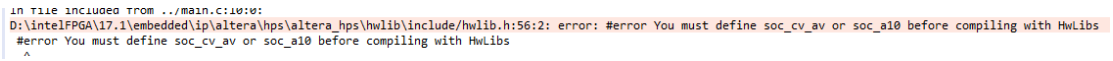
该源码包定义了大量的针对 HPS 硬件的宏定义和底层操作函数，在基于虚拟地址编程时，可以使用该源码包中的定义和函数来完成各种硬件外设的操作，避免了用户自行编写基于指针的操作函数。

hwlib.h：该文件中主要针对与 HPS 硬件编程相关的一些常量进行了定义，比如各种标志信号，实际在基于 Linux 的应用编程中用户直接使用该文件中的内容较少。但是该文件中有一个非常重要的条件编译选项需要关注，就是第 54 行的：

```
#if !defined(soc_cv_av) && !defined(soc_a10)
#error You must define soc_cv_av or soc_a10 before compiling with HwLibs
#endif
```

即该文件同时支持 Intel 的 Cyclone V、Arria V、Arria 10 SoC FPGA，针对不同的硬件平台，部分底层硬件定义会有差别，因此需要根据程序选择的器件

来选择底层定义。所以，一般在 main 函数所在文件中包含该头文件，用以检查是否制定了特定的硬件平台，如果没有定义，会在编译信息中提示。



指定专用硬件平台的方法最简单的就是在包含该头文件之前先定义硬件平台，例如在 mian 文件中，按照如下顺序来包含该头文件，就能够成功指定硬件平台了：

```
//hps 厂家提供的底层定义头文件
#define soc_cv_av  //定义使用soc_cv_av硬件平台

#include "hwlib.h"
#include "socal/socal.h"
#include "socal/hps.h"
```

socal.h：该文件中为一些基本的底层操作函数，如位、字节、半字、字的读写等。

hps.h：该文件中对 HPS 中各种外设地址信息进行了定义，如上述提到的 ALT_STM_OFST。该文件中的定义在进行虚拟地址映射时有用到。

添加 FPGA 侧外设硬件信息

上述头文件仅仅是针对 HPS 的架构的，没有包含在 Platform Designer 中添加的各种 FPGA 侧外设 IP，当我们需要对这些 FPGA 侧添加的外设进行操作时，还需要知道这些外设的硬件信息，如该外设在 lw_h2f_bridge 上的偏移地址。这些信息需要根据 Qsys 文件信息在 SoC EDS Command Shell 中用命令脚本生成。生成方式很简单，从开始菜单中的 Quartus 17.1 安装列表下找到 SoC Embedded Design Suite（EDS）下的 SoC EDS Command Shell 选项并打开，如图 xxx 所示。会弹出如图 xxx 所示的命令行窗口。

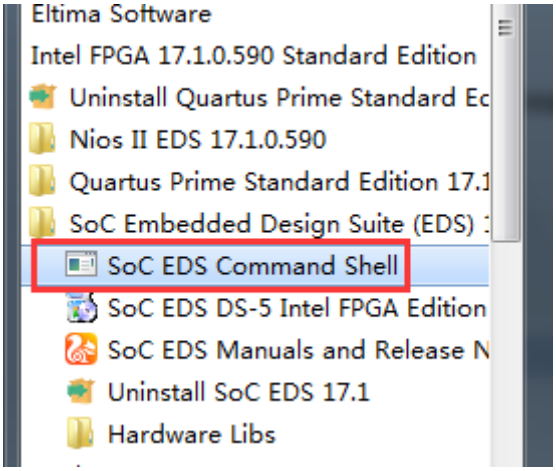


图 xxx 打开 SoC EDS Command Shell

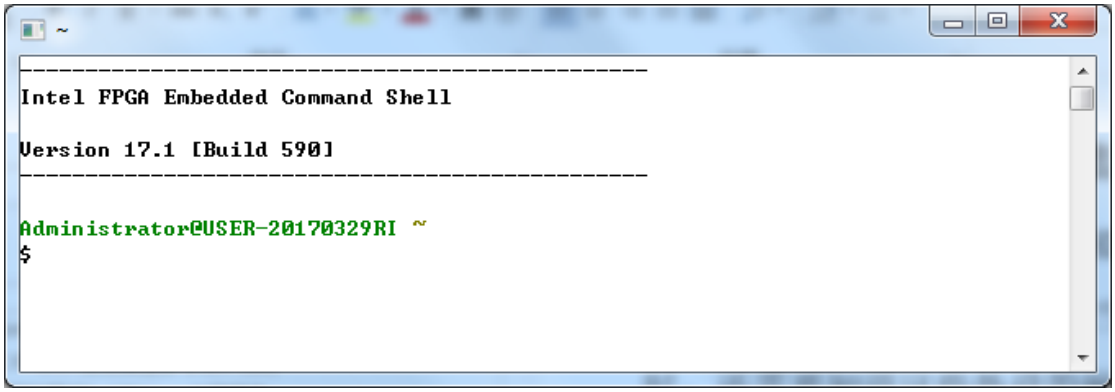
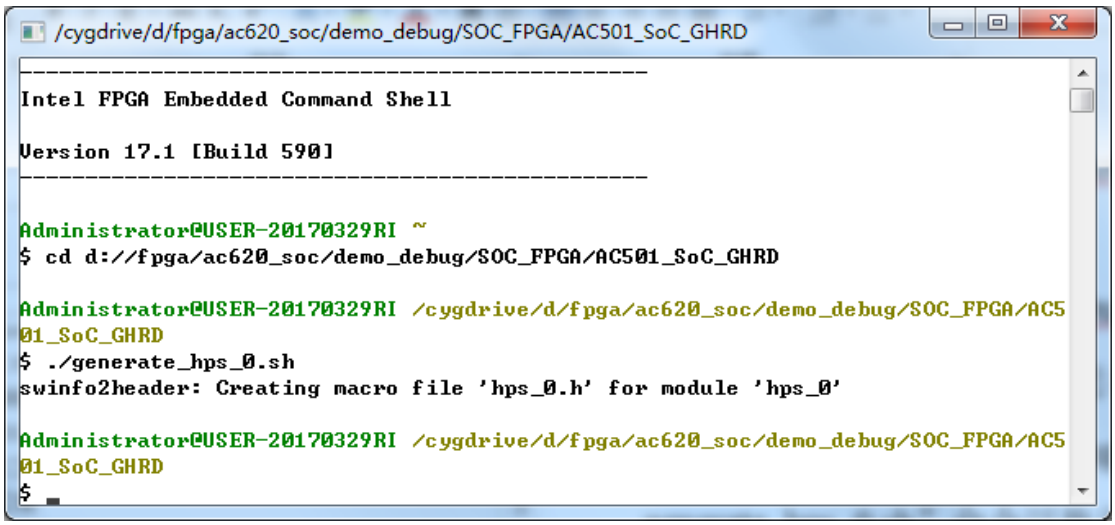


图 xxx SoC EDS Command Shell 主窗口

使用 cd 命令切换到对应的 Quartus 工程目录下，例如本例讲述的 AC501_SoC_GHRD 工程在笔者电脑上的位置为：D:\fpga\ac620_soc\demo_debug\SOC_FPGA\AC501_SoC_GHRD，然后输入“./generate_hps_0.sh”命令以执行 hps_0.h 文件生成脚本，即可在工程目录下生成或更新名为 hps_0.h 的头文件。如图 xxx 所示：



命令清单如下所示：

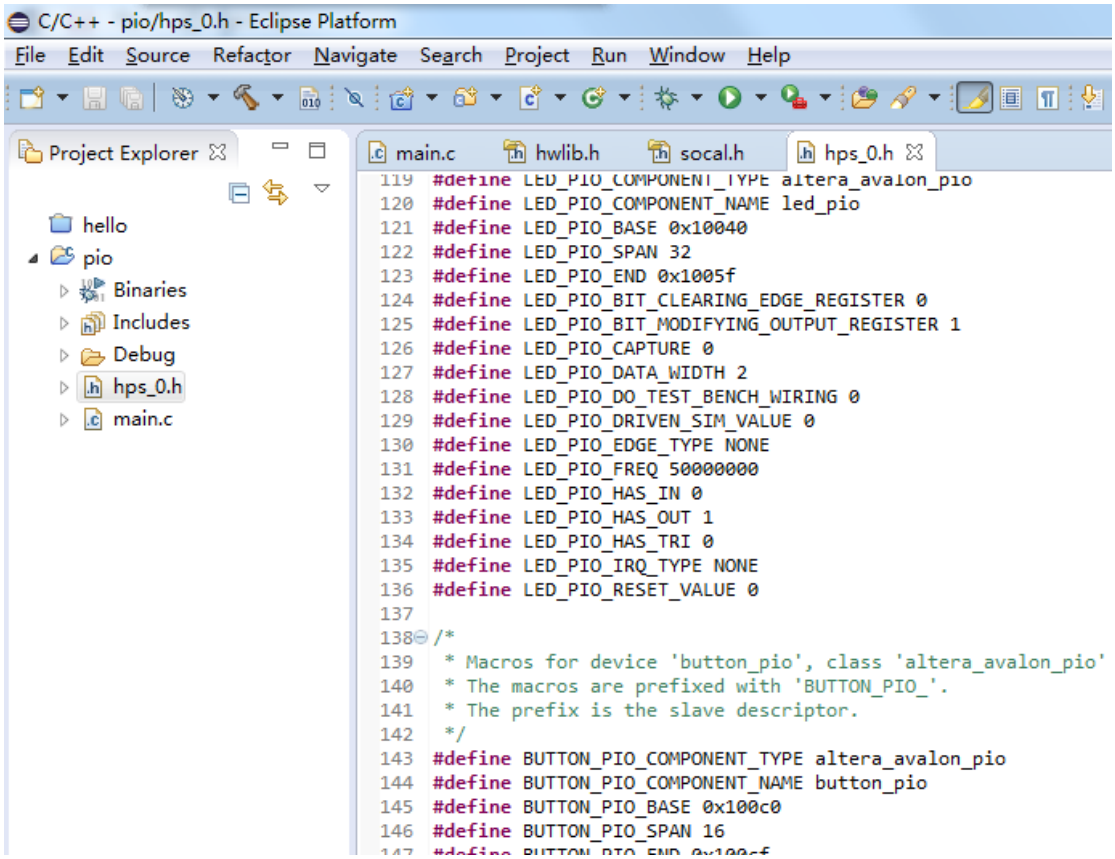
```
$ cd d://fpga/ac620_soc/demo_debug/SOC_FPGA/AC501_SoC_GHRD
$ ./generate_hps_0.sh
```

执行成功，会打印如下提示信息：

```
swinfo2header: Creating macro file 'hps_0.h' for module 'hps_0'
```

然后就能在 Quartus 工程根目录下找到生成好的 hps_0.h 文件了。

将该文件复制，然后粘贴到 DS-5 中 pio 工程下。双击打开该文件即可查看文件内容了，如图 xxx 所示：



可以看到，在 Platform Designer 中添加的外设 IP，在该文件中都有相应的硬件信息，例如 LED_PIO 外设的基地址为 0x10040，在 Patform Designer 中打开 soc_system.qsys 文件，切换到 Address Map 一栏，可以看到 led_pio.s1 在 hps_0.h2f_lw_axi_master 总线上的地址范围为 0x0001_0040~0x0001_005f，所以 led_pio 外设的基地址为 0x10040，结束地址为 0x1005f，地址空间为 32 字节，如图 xxx 所示。这与 hps_0.h 文件中 pio_led 外设的基地址（LED_PIO_BASE）、结束地址（LED_PIO_END）、地址空间（LED_PIO_SPAN）定义的值一致，因此，我们使用 hps_0.h 文件中的这些硬件信息，就能准确的操作 led_pio 外设了。例如在上述虚拟地址映射时提到的，在得到 led_pio 外设虚拟地址时，就使用了 LED_PIO_BASE 这个值。

```
led_pio_virtual_base = periph_virtual_base + ( ( unsigned
long  )( ALT_LWFPGASLVS_OFST + LED_PIO_BASE ) & ( unsigned
long)( HW_REGS_MASK ) );
```

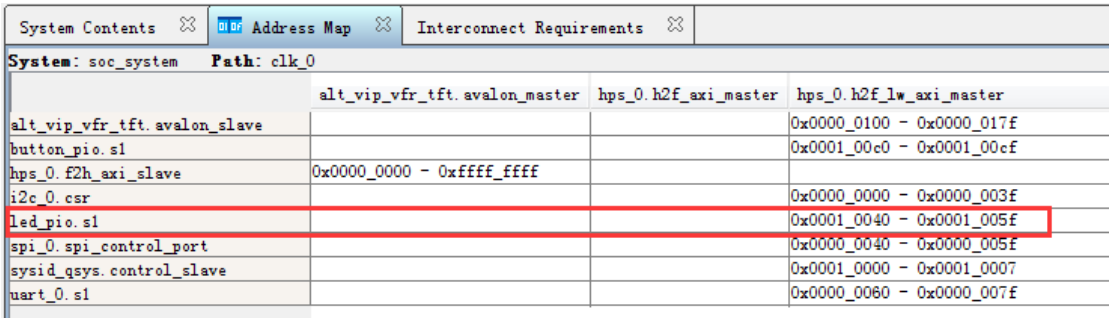


图 xxx Platform Designer 中外设地址信息

PIO IP 核介绍

通过上述操作，我们已经完成了 led_pio 和 button_pio 的虚拟地址映射，这些地址都是该外设的基地址，那么要如何才能够正确的操作该外设呢？例如，对于 led_pio，是用来驱动 led 灯的，而 AC501-SoC 开发板上的两个 FPGA 侧用户 LED 灯都是低电平点亮，低电平熄灭的。所以，我们需要通过设置 led_pio 的某个引脚为低电平来点亮该引脚连接的 led 灯，设置 led_pio 的某个引脚为高电平来熄灭该引脚连接的 led 灯。

Platform Designer 中提供的 IP 核都对应了有一个 IP 用户手册，名为《Embedded Peripherals IP User Guide》在该手册中可以查看该 IP 核的相关信息，包括 IP 功能描述、寄存器映射、寄存器功能等，PIO 核的信息在第 22 节，如图 xxx 所示。在该节中，包括了针对该 IP 核的简介（Core Overview）、功能描述（Function Description）、配置示例（Example Configurations）、配置说明（Configuration）、软件编程模型（Software Programming Model）。使用该 IP 前，需要先阅读该手册内容。

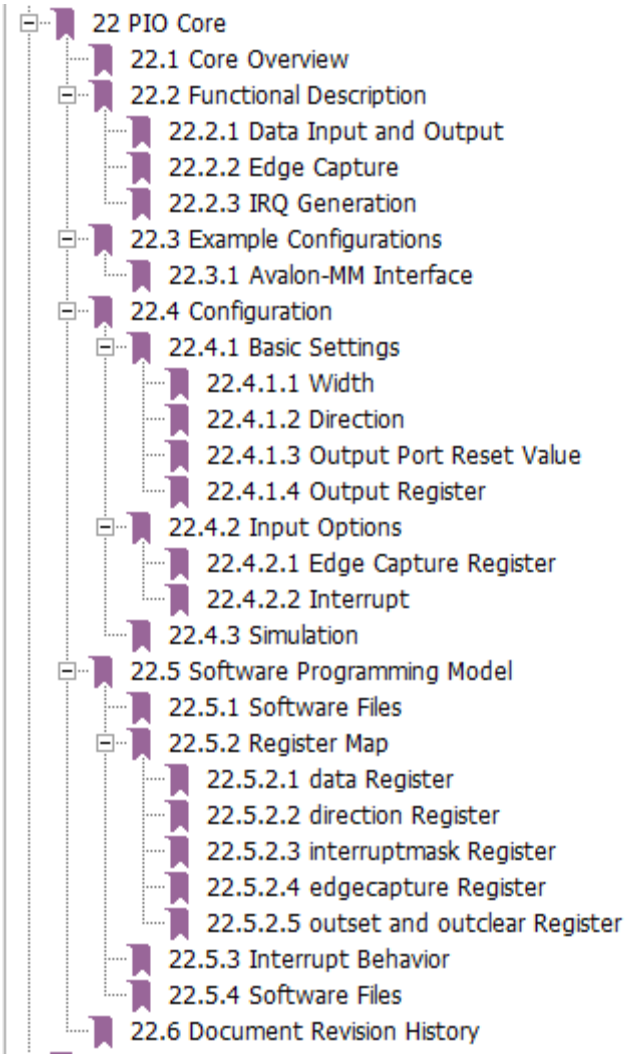


图 xxx PIO 核用户手册目录

作为 SoC FPGA 芯片中在 FPGA 侧使用可编程逻辑实现的 PIO 外设，其与 HPS 侧自带的外设在特性上有一定的差异，该差异主要表现在硬件可裁剪上。即对于一个完整功能的 GPIO 来说，一般支持配置为输入、输出、三态、边沿捕获、中断等，通过寄存器设置这些功能使能与否。而 Platform Designer 中的 PIO 核，则可以通过配置，选择是否具备这些功能。例如本例中，对于 led_pio 外设，只需要使用该引脚作为输出控制 LED 灯，无需使用输入功能，因此就可以将该 IP 核配置为仅输出功能，这样输入或三态功能逻辑就不会加入到 IP 核的逻辑代码中，编译时也就不会占用逻辑资源。而对于 button_pio 外设，只需作为输入功能，读取按键引脚电平。另外也可以捕获按键信号的边沿，并产生中断。所以选择使能边沿捕获功能，下降沿捕获，并产生边沿中断。由于功能

配置的差异，IP 核中的一些功能寄存器也会有差异，接下来介绍 PIO IP 核的寄存器映射和功能。

PIO 核寄存器映射

Avalon-MM 主机外设，例如 NIOS II CPU 或 HPS，可以通过 PIO 核提供的 4 个 32 位寄存器来对其实现控制和通信。

偏移	寄存器名		读写属性	n-1	……	2	1	0
0	data	读	仅读	当前 PIO 输入端口上的数据值				
		写	仅写	新的用来驱动 PIO 输出端口的值				
1	direction		读写	独立的方向控制寄存器，每一位对应一个 I/O 口的方向。为 0 设置该 I/O 为输入，为 1 设置该 I/O 为输出。				
2	interruptmask		读写	中断使能和禁止寄存器，每一位对应一个 I/O 口输入中断使能，为 1 使能该 I/O 输入中断，为禁止该 I/O 产生中断。				
3	edgecapture		读写	边沿捕获寄存器，每一位对应一个输入型 I/O 的边沿捕获状态				
4	outset		仅写	独立的输出端口置位控制，每一位为 1 时对应置位该输出端口的值。				
5	outclear		仅写	独立的输出端口清零控制，每一位为 1 时对应清零该输出端口的值。				

需要说明的是，direction、interruptmask、edgecapture、outset、outclear 寄存器可能并不存在。具体根据在 Platform Designer 中添加该 IP 时的设置决定。如果设置为非三态端口（Bidir），则 direction 寄存器不存在，如果设置为仅输出端口，或者设置为含输入功能的端口时没有使能中断，则 interruptmask 寄存器不存在，如果设置为仅输出端口，或者设置为含输入功能的端口时没有使能边沿捕获功能，则 edgecapture 寄存器不存在。如果没有勾选 Enable individual bit set/clear output register 选项，则 outset 和 outclear 寄存器将无效。

另外对于 edgecapture 寄存器，如果没有勾选 Enable bit-clearing for edge capture register 选项，那么往 edgecapture 寄存器写入任意值将清零所有位的捕获状态，否则，往指定位写入 1 将只清零对应位的值。

了解了该 IP 核的寄存器映射之后，就可以通过操作虚拟地址加对应寄存器偏移地址的方式来读写对应寄存器了。接下来示例讲解如何通过具体的 C 语言来检测按键，点亮 LED 灯。

编程时，可以直接使用指针的方式，对 PIO 外设的 data 寄存器对应位写入 0，来驱动对应的 I/O 输出低电平，从而点亮 LED 灯，例如设置 FPGA_LED0 点亮，FPGA_LED 1 熄灭，代码如下所示：

```
*(led_pio_virtual_base + 0) = 0x2; //LED0 亮，LED1 灭
```

或者使用数组下标的方式来指向 data 寄存器，代码如下所示：

```
led_pio_virtual_base[0] = 0x2; //LED0 亮，LED1 灭
```

0x2 的二进制值为 10b，即第 0 位值为 0，第 1 位值为 1，写入 PIO 核的数据寄存器之后，就会驱动 PIO 核对应的输出端口连接的 I/O 值电平为 0 或 1，从而实现点亮或熄灭 LED 灯的功能。

上述代码往数据寄存器中写入确定的值，一次性操作了所有位的状态。那么当我们仅希望对其中 1 位数据进行操作，不影响其他位的状态时，有两种方案可以选择，第一是首先读取 PIO 核的数据寄存器的值，到一个临时变量中，然后修改该变量中对应位的值为我们所希望的值，而不改变变量中其他位的值，然后再把修改好的临时变量的值重新写入数据寄存器。另一种方案就是利用 PIO 核里面的输出设置/输出清零寄存器，在配置 IP 时勾选了 Enable individual bit set/clear output register，则可以通过往 outset 寄存器的对应位写 1 来置位（置 1）输出端口的对应位，或是往 outclear 寄存器的对应位写 1 来清零（置 0）输出端口的对应位，而其他位不受任何改变。例如仅希望点亮 FPGA_LED1，而不影响 FPGA_LED 0 的状态，使用这两种方案的代码分别如下所示：

数据回写方式代码

```
    unsigned int data;//定义数据寄存器临时变量
    data = *(led_pio_virtual_base + 0);    //读取数据寄存器的值到data中
    data &= ~0x2;   //将data变量中的bit1设为0，其他位不变
    *(led_pio_virtual_base + 0) = data;    //将data值写回PIO核数据寄存器
```

使用清零寄存器方式代码

```
//向outclear寄存器的bit1写1，以清除输出端口中bit1的值
*(led_pio_virtual_base + 5) = 0x2;
```

对应的，如果要设置端口中 bit1 的输出状态为 1，则可以写 outset 寄存器的 bit1 的值为 1 来实现，代码如下所示：

```
//向outset寄存器的bit1写1，以置位输出端口中bit1的值
*(led_pio_virtual_base + 4) = 0x2;
```

而对于 button_pio 来说，是一个仅输入型 PIO，我们可以通过读取该 PIO 的数据寄存器的值来获知连接在该 PIO 端口上的每一个按键的输出电平，从而判断按键有没有被按下。使用指针方式读取 button_pio 数据寄存器的代码如下所示：

```
unsigned int button_data;//定义数据寄存器临时变量
button_data= *(button_pio_virtual_base +0);//读取PIO数据寄存器以获知按键状态
```

或者，也可以使用数组下标的方式来指定对应寄存器，代码如下所示：

```
button_data = button_pio_virtual_base[0];//读取PIO数据寄存器以获知按键状态
```

设计程序时，在使能了边沿捕获功能的情况下，也可以通过读取边沿捕获寄存器的值来判断按键有没有被按下过（按键按下过程中会产生下降沿，该下降沿会被边沿捕获功能捕获并存储在边沿捕获寄存器中）。使用指针方式读取 button_pio 边沿捕获寄存器的代码如下所示：

```
unsigned int button_edge;//定义边沿捕获寄存器临时变量
//读取PIO边沿捕获寄存器以获知是否有检测到设定的边沿
button_edge = *(button_pio_virtual_base + 3);
```

或者，也可以使用数组下标的方式来指定对应寄存器，代码如下所示：

```
unsigned int button_edge;//定义边沿捕获寄存器临时变量
//读取PIO边沿捕获寄存器以获知是否有检测到设定的边沿
button_edge = button_pio_virtual_base[3];
```

对于边沿捕获寄存器中的值，需要在程序中通过程序手动清楚，以保证下一次边沿事件能够被正确捕获。在勾选了 Enable bit-clearing for the edge capture register 选项的情况下，向该寄存器中某一位写 1 将清除对应位的值。如果没有使能该选项，则向该寄存器中写入任意值将清除所有位的值。使用指针方式清除 button_pio 边沿捕获寄存器中 bit0 的值的代码如下所示：

```
//清除边沿捕获寄存器的bit0的值
*(button_pio_virtual_base + 3) = 0x1;
```

对于中断信号，由于中断需要在 Linux 内核驱动程序中才能注册使用，因此在基于虚拟地址映射的应用程序开发中无法使用。所以，如果想使用外设的中断功能，则必须编写 Linux 内核驱动程序。不过在本例中，暂且不用考虑中断问题，直接使用直接使用读取数据寄存器的方式来判断按键是否有按下。

PIO IP 核应用实例

本节将通过一个具体的实例，来完成基于 PIO 核的按键控制 LED 功能。设计时，每当检测到 FPGA_KEY0 按下事件，就调整 FPGA_LED0 的闪烁频率，每当检测到 FPGA_KEY1 按下事件，就调整 FPGA_LED1 的闪烁频率，闪烁频率分别为 0.1Hz、0.5Hz、1Hz 三个档位，每按下一次按键，对应 LED 的闪烁频率就在这三个频率之间切换。以下为完整程序清单：

```
//gcc标准头文件
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

//hps 厂家提供的底层定义头文件
#define soc_cv_av     //定义使用soc_cv_av硬件平台

#include "hwlib.h"
#include "socal/socal.h"
#include "socal/hps.h"

//与用户具体HPS应用系统相关的硬件描述头文件
#include "hps_0.h"

#define HW_REGS_BASE (ALT_STM_OFST )  //HPS外设地址段基地址
#define HW_REGS_SPAN (0x04000000 )         //HPS外设地址段地址空间
#define HW_REGS_MASK (HW_REGS_SPAN - 1 )  //HPS外设地址段地址掩码

static volatile unsigned long *led_pio_virtual_base = NULL; //led_pio虚拟地址
static volatile unsigned long *button_pio_virtual_base = NULL;//button_pio虚拟地址

int fpga_init(long int *virtual_base) {
    int fd;
    void *periph_virtual_base;    //外设空间虚拟地址

    //打开MPU
```

```
    if ((fd = open("/dev/mem", ( O_RDWR | O_SYNC))) == -1) {
        printf("ERROR: could not open \"/dev/mem\"...\n");
        return (1);
    }

    //将外设地址段映射到用户空间
    periph_virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE),
            MAP_SHARED, fd, HW_REGS_BASE);
    if (periph_virtual_base == MAP_FAILED) {
        printf("ERROR: mmap() failed...\n");
        close(fd);
        return (1);
    }

    //映射得到led_pio外设虚拟地址
    led_pio_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + LED_PIO_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
    //映射得到button_pio外设虚拟地址
    button_pio_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + BUTTON_PIO_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
    *virtual_base = periph_virtual_base;  //将外设虚拟地址保存，用以释放时候使用
    return fd;
}

int main(int argc, char ** argv) {

    int fd;
    int virtual_base = 0;//虚拟基地址
    unsigned int button_edge; //定义边沿捕获寄存器临时变量

    bool led0 = true, led1 = true;

    //完成fpga侧外设虚拟地址映射
    fd = fpga_init(&virtual_base);

    //清除边沿捕获寄存器的bit0的值
    *(button_pio_virtual_base + 3) = 0x3;

    while (1) {
        //读取PIO边沿捕获寄存器以获知是否有检测到设定的边沿
        button_edge = *(button_pio_virtual_base + 3);
        switch (button_edge) {
```

```
        case 0x1:
            led0 = !led0; //对FPGA_LED 的bit0取反
            *(button_pio_virtual_base + 3) = 0x1; //清除边沿捕获寄存器的bit0位
            if (led0)
                *(led_pio_virtual_base + 4) = 0x1; //置位led_pio的bit0输出
            else
                *(led_pio_virtual_base + 5) = 0x1; //清零led_pio的bit0输出
            break;

        case 0x2:
            led1 = !led1; //对FPGA_LED 的bit1取反
            *(button_pio_virtual_base + 3) = 0x2; //清除边沿捕获寄存器的bit1位
            if (led1)
                *(led_pio_virtual_base + 4) = 0x2;     //置位led_pio的bit1输出
            else
                *(led_pio_virtual_base + 5) = 0x2;     //清零led_pio的bit1输出
            break;

        case 0x3:
            led0 = !led0; //对FPGA_LED 的bit0取反
            led1 = !led1; //对FPGA_LED 的bit1取反

            //清除边沿捕获寄存器的bit0和bit1位
            *(button_pio_virtual_base + 3) = 0x3;

            //将led0和led1的状态直接写入led_pio数据寄存器
            *(led_pio_virtual_base + 0) = led1 * 2 | led0;
            break;

        default:
            break;
        }
    }

    //程序退出前，取消虚拟地址映射
    if (munmap(virtual_base, HW_REGS_SPAN) != 0) {
        printf("ERROR: munmap() failed...\n");
        close(fd);
        return (1);
    }

    close(fd); //关闭MPU
    return 0;
}
```

程序中，通过读取 button_pio 外设的边沿捕获寄存器的值，判断有无按键被按下，如果有按键按下，则根据边沿捕获寄存器的值确定是哪个按键被按下，然后设置对应的 LED 的状态，如果是单个按键被按下，则通过写 led_pio 外设中单 bit 输出置位和清零寄存器的值来关闭或打开对应的 led 灯。如果是两个按键同时按下，则直接写 led_pio 外设的数据寄存器来完成 led 状态的更新。同时，在每次判断完毕后会及时的清除 button_pio 外设中边沿捕获寄存器的值。

该应用程序虽然功能简单，但是通过该简单的例子，向读者展示了如何使用 PIO 外设中的各个寄存器来实现相应的功能。方便读者根据该示例举一反三，设计自己的应用程序。

合理的程序退出机制

需要注意的是，该示例中，直接使用的 while(1)的形式来保证程序的持续运行，并未做合理的退出机制。如果要退出该程序，可在当前终端窗口中通过组合键 Ctrl+Z 来完成程序的强制退出。程序的强制退出可能会无法正常的释放打开的各种资源，因此，可以简单修改程序的功能，设置几个按键专用于控制程序的退出。例如，仅使用 button_pio 的第 0 位来控制 FPGA_LED0，而 button_pio 的第 1 位用来作为程序退出的检测标志，一旦 FPGA_KEY1 按下，则退出 while 循环。实例代码如下所示：

```
int main(int argc, char ** argv) {

    int fd;
    int virtual_base = 0;//虚拟基地址
    unsigned int button_edge; //定义边沿捕获寄存器临时变量

    bool led0 = true, led1 = true;

    //完成fpga侧外设虚拟地址映射
    fd = fpga_init(&virtual_base);

    //清除边沿捕获寄存器的bit0的值
    *(button_pio_virtual_base + 3) = 0x3;

    bool STOP = false;
    while(STOP == false)
    {
        //读取PIO边沿捕获寄存器以获知是否有检测到设定的边沿
```

```
        button_edge = *(button_pio_virtual_base + 3);
        switch (button_edge) {
        case 0x1:
            led0 = !led0; //对FPGA_LED 的bit0取反
            *(button_pio_virtual_base + 3) = 0x1; //清除边沿捕获寄存器的bit0位
            if (led0)
                *(led_pio_virtual_base + 4) = 0x1; //置位led_pio的bit0输出
            else
                *(led_pio_virtual_base + 5) = 0x1; //清零led_pio的bit0输出
            break;

        case 0x2:
            STOP = true;break;    //设置程序退出标志

        case 0x3:
            STOP = true;break;    //设置程序退出标志

        default:
            break;
        }
    }

    //程序退出前，取消虚拟地址映射
    if (munmap(virtual_base, HW_REGS_SPAN) != 0) {
        printf("ERROR: munmap() failed...\n");
        close(fd);
        return (1);
    }

    close(fd); //关闭MPU
    return 0;
}
```

在 while 循环中，每次按下 FPGA_KEY0 会正常的改变 FPGA_LED0 的状态，而按下 FPGA_KEY1，则会设置 STOP 的值为真，程序每次 while 循环都会检查 STOP 的状态，如果为假，则继续运行，如果为真，当程序再次运行到判断 while 循环条件时，就会因为条件不满足而跳出 while 循环，接着执行 while 后面的释放虚拟地址映射和关闭 MPU 内容，最后退出整个程序。

关于按键消抖

机械按键在按下和释放时都存在抖动，如果不对抖动进行合理的处理，就会影响软件程序的正确识别判定。在 MCU 中，一般都使用软件延时的方式来进行抖动滤除，此种方式会占用 CPU 软件资源。而在 FPGA 中，可以使用 Verilog 编写数字逻辑完成抖动的滤除得到纯净的按键按下和释放边沿信号，再将该纯净的边沿信号连接给 PIO 作为输入，从而降低软件编程时的复杂度。关于机械按键的抖动原理和使用 Verilog 编写抖动滤除逻辑的实现，可以参考笔者的《FPGA 自学笔记——设计与验证》一书中第三章 3.7 小节“独立按键消抖设计与验证”内容。

在 AC501_SoC_GHRD 工程中，就是使用了《FPGA 自学笔记——设计与验证》一书中独立按键消抖的代码，对输入的两个轻触按键的信号先进行了抖动滤除，然后再连接到 button_pio 的导出引脚上，实现按键输入的功能，如图 xxx 所示。因此在编程时，button_pio 的输入信号已经是不含抖动的按键信号了，无需在软件中进行抖动滤除功能，直接读取边沿捕获寄存器的值就能得到准确的按键按下事件。

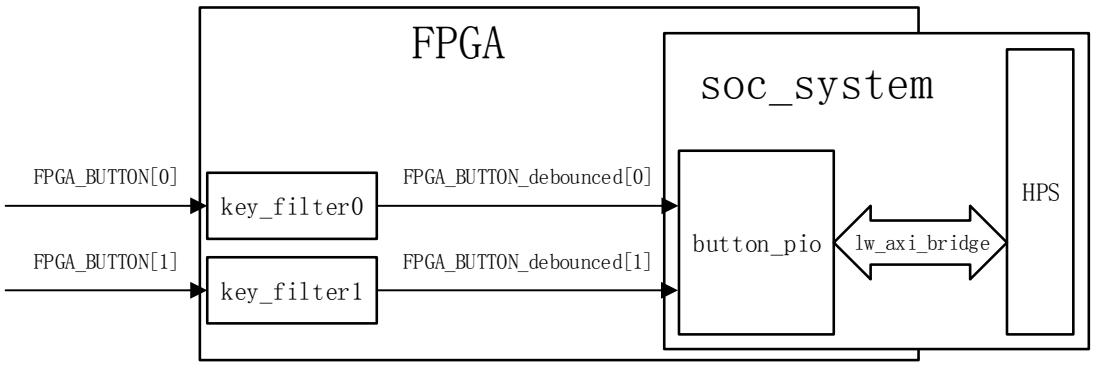


图 xxx 基于 FPGA 逻辑消抖的按键输入

基于虚拟地址映射的 UART 编程应用

上一节的实验，完成了虚拟地址的映射和基于虚拟地址的按键和 LED 指示灯的编程控制。本节将继续使用该种方法，完成对 AC501_SoC_GHRD 工程中添加的 uart_0 外设进行控制。

在 DS-5 中建立 UART 应用工程

在 DS-5 软件中选中上一节创建的 pio 工程，复制并粘贴为新的工程，命名为 fpga_uart，以建立好基本的工程。在复制完成之后，先将新工程下的 Debug 目录删除，因为这些文件是之前工程编译出来的，在新工程中既不会被使用，也不会被自动删除，因此需要手动删除。避免与新工程的生成文件混淆。

直接复制已有工程并重新命名以得到新工程最大的优点是可以直接使用已有工程的设置，而不用再新建工程后再进行一系列的硬件设置，例如添加和包含 HPS 库文件等。

UART （RS-232 Serial port）核介绍

UART （RS-232 Serial port）核是 Platform Designer 中提供的一个经典的字符型串行通信外设，使用该外设，能够方便的通 FPGA 片外的设备进行通信。该 IP 核实现了 RS-232 协议的时序，并提供可调整的波特率速度，校验位，停止位和数据位宽。这些参数都是可以配置的，在具体应用中，根据实际需要用到的功能，配置这些特性，在保证功能实现的前提下，降低对 FPGA 逻辑资源的占用。IP 核提供一个 Avalon Memory-Mapped (Avalon-MM) slave 接口来和 Avalon-MM master 外设，例如 Nios II CPU、HPS 进行通信。

该 IP 核还提供中断功能，支持以中断的方式来及时和主控进行通信，以获得更高的通信效率。

需要注意的是，该 IP 核内部不含数据 FIFO，不具备 16550 标准串口的一系列功能，每次接收到一个字符的数据，必须由处理器及时读取，否则数据将丢失，这为查询方式操作该 IP 核带来了一定的考验。另外，如果使能了接收和发送中断，则每接收或发送一个字符的数据，都会对 CPU 发起一次中断，因此频繁的数据收发会给 CPU 的中断处理造成较重的负担。

根据上述特性介绍，设计者在为 HPS 添加 UART IP 时，对于仅收发送少量命令和数据的的场合，由于该控制器结构简单，占用资源少，编程简单，使用该控制器是一个不错的选择。而需要考虑实时性和对 CPU 的中断资源开销问题的情况下，不推荐在高速、频繁的数据通信场合使用本 UART 控制器。对于性能有要求的场合，可以使用 Platform Designer 中提供的 Altera 16550 Compatible

UART 核，该 IP 核兼容标准的 16550 串口，提供与 16550 相同的功能和性能。不过使用该 IP 需要取得相应的授权文件（License），或是针对自己的应用编写增强型 UART 控制器。笔者就曾针对 ModBus 通信协议，编写过自带 CRC 校验、256 字节深度接收 fifo、自动帧结束判定的增强型 UART 控制器，并应用于工业通信设备上。

UART （RS-232 Serial port）寄存器映射

这个 UART IP 核提供给一个 Avalon-MM slave 接口来与 Avalon-MM master 接口的主机通信。IP 核内部设有 6 个 16 位的寄存器，包括控制寄存器（control）、状态寄存器（status）、接收数据寄存器（rxdata）、发送数据寄存器（txdata）、波特率分频寄存器（divisor）和数据包结束寄存器（endofpacket）。Avalon-MM master 接口的主机通过读写这些寄存器就能完数据的收发。

关于该控制器的详细功能介绍，不作为本书的重点，需要读者自行阅读《Embedded Peripherals IP User Guide》手册中第 8 节的内容。这里仅对其几个与编程重点相关的寄存器进行说明。

偏移	名称	读写属性	寄存器描述
0	rxdata	只读	接收数据寄存器，接收器每接收到一个字符的数据就会自动存入该寄存器。根据数据位的配置的不同，该寄存器的有效位宽为 7、8 或 9 位
1	txdata	只写	发送数据寄存器，将数据写入该寄存器，发送器会自动发送。根据数据位的配置的不同，该寄存器的有效位宽为 7、8 或 9 位
2	status	可读可写	状态寄存器，存储了 IP 工作中的各种状态
3	control	可读可写	控制寄存器，可以设置各种条件的中断使能
4	divisor	可读可写	波特率分频寄存器，在使能了可编程波特率的情况下，通过修改该寄存器的值，可以修改通信波特率
5	endofpacket	可读可写	在使能了流控功能的情况下，设置流结尾的数据值

rxdata：对于接收数据寄存器，根据在添加控制器时配置的数据位宽不同，该寄存器的有效位数为 7、8、9 三种情况。UART 接收模块每接收到一个字符的数据就会自动存入该寄存器。需要注意的是，存入该寄存器的数据必须由 Avalon MM 主机及时读取，否则当新接收完一个字符的数据后，该数据会被覆盖。

txdata：对于发送数据寄存器，根据在添加控制器时配置的数据位宽不同，该寄存器的有效位数为 7、8、9 三种情况。当该寄存器为空时，Avalon MM 主机向该寄存器中写入一个字符的数据，则该数据会被传入 UART 发送模块并发送出去。当该寄存器中不为空时，向该寄存器写入数据会导致前一个数据的丢失。

status：状态寄存器，该寄存器存储了 UART 控制器运行时的各种状态，每一位代表了一个特殊状态的值，在设计该控制器的应用程序时，比较重要的两个状态位为 bit7 的 rrdy 状态和 bit6 的 trdy 状态。

rrdy 状态位指示了 rxdata 寄存器的当前状态，当 rxdata 寄存器为空时，表明还没有接收到有效数据，则此时 Avalon MM 主机不能去读取 rxdata 寄存器中的值，rrdy 位的值为 0，当 rxdata 寄存器中存储了有效的接收数据时，该位自动置一。因此 Avalon MM 主机可以通过读取该位的值，来判断 rxdata 寄存器中是否有有效数据可读，如果 rrdy 位为 1，则 Avalon MM 主机应尽快将 rxdata 中的值读取掉。

trdy 状态位指示了发送寄存器 txdata 的状态，当发送寄存器为空时，Avalon MM 主机可以向该寄存器中写入新的需要发送的数据，此时 trdy 位的值为 1。而一旦 txdata 寄存器中被写入了新的数据，则该位变为 0。因此当 Avalon MM 主机检测到该位为 0 时，不能向 txdata 寄存器中写入新的值，只有当该位为 1 时，才能写入新的要发送的值。

control：控制寄存器，该寄存器随名为控制寄存器，但更像一个中断屏蔽寄存器，在 bit0~bit12 中，除了 bit11 是 RTS 信号控制位以外，其他每一位都对应了状态寄存器中一位状态的中断使能信号。一旦本寄存器中某一位被设置为了 1，那么当 status 寄存器中对应位变为 1 时，就会向 cpu 发出中断。

divisor：波特率分频寄存器，该寄存器是否存在，与 Platform Designer 中添加 IP 核时是否勾选固定波特率选项有关，如果勾选了固定波特率选项，则该寄存

器不存在，只有在没勾选固定波特率选项时，Avalon MM 主机才能通过写该寄存器的值来修改 UART 收发的波特率。波特率值的计算关系为：

$$\text{baud rate} = (\text{clock frequency}) / (\text{divisor} + 1)$$

其中 baud rate 为期望设定的波特率的值，clock frequency 为 UART 串口模块的输入时钟频率。由此可以得出 divisor 更准确的计算公式为：

$$\text{divisor} = \text{int}\left(\frac{\text{clock frequency}}{\text{baud rate}} + 0.5\right)$$

UART IP 核应用实例

通过对上述寄存器的解读，了解了每个寄存器及其中各个位的功能意义。然后就可以据此来编写相关的数据收发代码了。

虚拟地址映射

第一步还是完成虚拟地址映射。映射方式和 led_pio 外设一致，可以直接在 PIO 实验中以有代码的基础上添加 uart_0 的部分即可，代码如下所示：

```
static volatile unsigned long *led_pio_virtual_base = NULL;//led_pio虚拟地址
static volatile unsigned long *button_pio_virtual_base = NULL;  //button_pio虚拟地址
static volatile unsigned long *uart_0_virtual_base = NULL;  //uart_0虚拟地址

int fpga_init(long int *virtual_base) {
    int fd;
    void *periph_virtual_base;    //外设空间虚拟地址

    //打开MPU
    if ((fd = open("/dev/mem", ( O_RDWR | O_SYNC))) == -1) {
        printf("ERROR: could not open \"/dev/mem\"...\n");
        return (1);
    }

    //将外设地址段映射到用户空间
    periph_virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE),
            MAP_SHARED, fd, HW_REGS_BASE);
    if (periph_virtual_base == MAP_FAILED) {
        printf("ERROR: mmap() failed...\n");
        close(fd);
        return (1);
```

```
    }

    //映射得到led_pio外设虚拟地址
    led_pio_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + LED_PIO_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
    //映射得到button_pio外设虚拟地址
    button_pio_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + BUTTON_PIO_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
    //映射得到uart_0外设虚拟地址
    uart_0_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + UART_0_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
    *virtual_base = periph_virtual_base;  //将外设虚拟地址保存，用以释放时候使用
    return fd;
}
```

可以看到，代码中仅仅是在 PIO 核实验的代码基础上，新增定义了一个 uart_0_virtual_base 指针，并在进行外设虚拟地址映射时增加了 uart_0_virtual_base 一项的计算赋值。所以，通过这两个实验中虚拟地址映射的对比可以知道，当程序中需要得到多个外设的虚拟地址时，仅需先打开 MPU，然后得到外设地址空间的基地址 periph_virtual_base，然后再依次将所需用到的外设的虚拟地址通过与 periph_virtual_base 执行相应运算得到。无需对每个外设重新执行打开 MPU 和映射外设虚拟地址的操作。

设置波特率

在完成了 uart_0 的虚拟地址映射后，要使用 UART IP 核进行正确的数据收发，首先需要设置相应的收发波特率，当添加 UART IP 核时没有选择固定波特率选项时，就可以通过写 divisor 的值来实现。例如，设置波特率为 9600bps，就可以使用下面的代码来实现：

```
    //设置uart_0的波特率为9600bps
    *(uart_0_virtual_base + 4) = (int)(UART_0_FREQ/9600 + 0.5);
```

其中 UART_0_FREQ 是从外设信息头文件 hps_0.h 中得到的，这是 UART 控制器的 Avalon MM 总线所使用的时钟频率。

由于基于虚拟地址映射的操作是在用户空间完成对各种外设的操作，而用户空间是无法进行中断的注册和使用的，因此本实验中不使用中断功能，直接使用查询状态寄存器的形式完成数据的收发。所以对于 control 寄存器，无需进行任何设置，使用默认的全 0 值即可。

字符发送

串口发送数据是以基本的字符为单位的，最底层的操作一般就是 putc 函数来发送一个字符，然后上层再循环调用该函数来完成数据串的发送。使用 UART 控制器发送一个字符，基本思路是循环读取状态寄存器的值，一旦检测到状态寄存器中的 trdy 值为 1，表明 txdata 寄存器可以写入新的待发送数据了，就将需要发送的新的数据写入 txdata 寄存器即可。循环读取状态寄存器并发送数据的代码如下所示：

```
void uart_putc(char c)
{
    unsigned short uart_status;   //状态寄存器值
    do{
        uart_status = *(uart_0_virtual_base + 2);//读取状态寄存器
    }while(!(uart_status & 0x40));    //等待状态寄存器bit6（trdy）为1

    *(uart_0_virtual_base + 1) = c;   //发送一个字符
}
```

代码中使用了一个 do{}while()的循环结构来循环读取 uart_0 的状态寄存器，并检测其中 trdy 位（bit6）的值，一旦该位为 1，表明 txdata 寄存器可以被写入新的数据了，然后跳出循环，使用指针的形式将一个数据写入 UART 控制器的 txdata 寄存器。

字符串发送

有了基本的字符发送函数，就可以实现字符串发送了。基本的字符串发送函数设计思路很简单，只需要持续检测待发送的数据是否为空字符，如果为空字符表明一串字符串发送结束，退出函数。如果非空字符，则发送当前指针指向的字符，然后指针递增 1。简单的字符串发送函数如下所示：

```
void uart_printf(char *str)
{
    while(*str != '\0')  //检测当前指针指向的数是否为空字符
```

```
    {
        uart_putc(*str); //发送一个字符
        str++;   //字符串指针+1
    }
}
```

使用该函数发送字符串就很简单了，只需调用该函数并将需要发送的字符串作为参数传入即可，例如使用该发送函数发送“Hello World!”的代码如下所示：

```
    uart_printf("Hello World!\n"); //打印hello world字符
```

字符接收

串口接收数据也是以字符为基本单位的，当需要从串口接收一个字符的数据时，最底层的函数一般就是 getc 函数，然后上层再循环调用该函数来完成数据串的接收。使用 UART 控制器接收一个字符，基本思路是循环读取状态寄存器的值，一旦检测到状态寄存器中的 rrdy 值为 1，表明 rxdata 寄存器中有新的数据可以读取了，就将 rxdata 寄存器中的数据读取出来，返回给上层函数。循环读取状态寄存器并读取数据的代码如下所示：

```
int uart_getc(void)
{
    unsigned short uart_status;   //状态寄存器值
    do{
        uart_status = *(uart_0_virtual_base + 2);//读取状态寄存器
    }while(!(uart_status & 0x80));    //等待状态寄存器bit7（rrdy）为1

    return *(uart_0_virtual_base + 0);    //读取一个字符并作为函数返回值返回
}
```

代码中使用了一个 do{}while()的循环结构来循环读取 uart_0 的状态寄存器，并检测其中 rrdy 位（bit7）的值，一旦该位为 1，表明 rxdata 寄存器中有新的数据可以读取了，然后跳出循环，使用指针的形式读取 rxdata 寄存器中的值并将该值作为函数返回值返回给上层函数。

字符串接收

有了基本的字符接收函数，就可以实现字符串接收了。基本的字符串接收函数设计思路很简单，使用 uart_getc()函数从串口读取一个字符存入接收缓存指针指向的地址，并检测新接受到的数据是否为换行符“\n”，如果为换行符表明一行字符串接收结束，退出函数，返回当前接收到的字符个数。如果换行符，则继续读取新的数据，而且接收缓存指针递增 1。简单的字符串接收函数如下所示：

```
int uart_scanf(char *p)
{
    int cnt = 0; //接收个数计数器
    while(1)
    {
        *p = uart_getc();//读取一个字符的数据
        cnt++;
        if(*p == '\n')   //判断数据是否为换行
            return cnt;  //换行则停止计数,返回当前接收的字符个数
        else
            p++;//接收指针增1
    }
}
```

使用该函数接收字符串非常简单，只需调用该函数并将接收缓存的指针作为参数传入即可，例如使用该接收函数接收一行数据并存入 rx_buf 数组的代码如下所示：

```
    char rx_buf[128]={0};  //定义一个128字节的接收数组
    memset(rx_buf,0,128);  //清除数组中内容
    uart_scanf(&rx_buf);   //接收一行数据到rx_buf中
    printf(rx_buf);//打印当前接收到的字符串内容
```

程序首先定义了一个 128 字节的数组，然后使用 memset()函数清除了数组中的值，接着调用 uart_scanf()函数读取一行字符串到 rx_buf 中。最后使用系统 printf()函数将 rx_buf 中的字符串内容打印出来。注意，使用了 memset()函数，需要包含头文件 string.h。

以下为整个应用程序的程序清单：

```
//gcc标准头文件
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
```

```c
#include <sys/mman.h>
#include <string.h>

//hps 厂家提供的底层定义头文件
#define soc_cv_av     //定义使用soc_cv_av硬件平台

#include "hwlib.h"
#include "socal/socal.h"
#include "socal/hps.h"

//与用户具体HPS应用系统相关的硬件描述头文件
#include "hps_0.h"

#define HW_REGS_BASE (ALT_STM_OFST )  //HPS外设地址段基地址
#define HW_REGS_SPAN (0x04000000 )         //HPS外设地址段地址空间
#define HW_REGS_MASK (HW_REGS_SPAN - 1 )  //HPS外设地址段地址掩码

static volatile unsigned long *led_pio_virtual_base = NULL; //led_pio虚拟地址
static volatile unsigned long *button_pio_virtual_base = NULL;//button_pio虚拟地址
static volatile unsigned long *uart_0_virtual_base = NULL;  //uart_0虚拟地址

int fpga_init(long int *virtual_base) {
    int fd;
    void *periph_virtual_base;    //外设空间虚拟地址

    //打开MPU
    if ((fd = open("/dev/mem", ( O_RDWR | O_SYNC))) == -1) {
        printf("ERROR: could not open \"/dev/mem\"...\n");
        return (1);
    }

    //将外设地址段映射到用户空间
    periph_virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE),
    MAP_SHARED, fd, HW_REGS_BASE);
    if (periph_virtual_base == MAP_FAILED) {
        printf("ERROR: mmap() failed...\n");
        close(fd);
        return (1);
    }

    //映射得到led_pio外设虚拟地址
    led_pio_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + LED_PIO_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
```

```
    //映射得到button_pio外设虚拟地址
    button_pio_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + BUTTON_PIO_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
    //映射得到uart_0外设虚拟地址
    uart_0_virtual_base = periph_virtual_base
            + ((unsigned long) ( ALT_LWFPGASLVS_OFST + UART_0_BASE)
                    & (unsigned long) ( HW_REGS_MASK));
    *virtual_base = periph_virtual_base;  //将外设虚拟地址保存，用以释放时候使用
    return fd;
}

//串口字符发送函数
void uart_putc(char c) {
    unsigned short uart_status;   //状态寄存器值
    do {
        uart_status = *(uart_0_virtual_base + 2); //读取状态寄存器
    } while (!(uart_status & 0x40));  //等待状态寄存器bit6（trdy）为1

    *(uart_0_virtual_base + 1) = c;   //发送一个字符
}

//串口字符串发送函数
void uart_printf(char *str) {
    while (*str != '\0') //检测当前指针指向的数是否为空字符
    {
        uart_putc(*str); //发送一个字符
        str++;   //字符串指针+1
    }
}

//串口字符接收函数
int uart_getc(void) {
    unsigned short uart_status;   //状态寄存器值
    do {
        uart_status = *(uart_0_virtual_base + 2); //读取状态寄存器
    } while (!(uart_status & 0x80));  //等待状态寄存器bit7（rrdy）为1

    return *(uart_0_virtual_base + 0);    //读取一个字符并作为函数返回值返回
}

//串口字符串接收函数
int uart_scanf(char *p) {
    int cnt = 0; //接收个数计数器
```

```
    while (1) {
        *p = uart_getc();     //读取一个字符的数据
        cnt++;
        if (*p == '\n')  //判断数据是否为换行
            return cnt;  //换行则停止计数,返回当前接收的字符个数
        else
            p++;//接收指针增1
    }
}

int main(int argc, char ** argv) {

    int fd;
    int virtual_base = 0;//虚拟基地址

    //完成fpga侧外设虚拟地址映射
    fd = fpga_init(&virtual_base);

    //设置uart_0的波特率为9600bps
    *(uart_0_virtual_base + 4) = (int) (UART_0_FREQ / 9600 + 0.5);

    uart_printf("Hello World!\n");    //打印hello world!字符串
    uart_printf("Hello SoC FPGA!\n"); //打印Hello SoC FPGA!字符串
    uart_printf("www.corecourse.cn\n");   //打印www.corecourse.cn字符串

    char rx_buf[128] = { 0 }; //定义一个128字节的接收数组
    memset(rx_buf, 0, 128);  //清除数组中内容
    uart_scanf(&rx_buf); //接收一行数据到rx_buf中
    printf(rx_buf);  //打印当前接收到的字符串内容

    //程序退出前，取消虚拟地址映射
    if (munmap(virtual_base, HW_REGS_SPAN) != 0) {
        printf("ERROR: munmap() failed...\n");
        close(fd);
        return (1);
    }

    close(fd); //关闭MPU
    return 0;
}
```

UART IP 核板级调试

在进行试验时，由于 AC501_SoC_GHRD 中 uart_0 外设的引脚是分配到了 GPIO0 上的 FPGA_GPIO0_D6 和 FPGA_GPIO0_D7，为了能够进行串口调试看到对应的现象，需要使用串口调试模块来配合调试，例如常见的 USB 转 TTL 串口模块。例如对于一个常见的基于 CH340 的 USB 转 TTL 串口模块，按照如图 xxx 所示的连接方式进行连接：

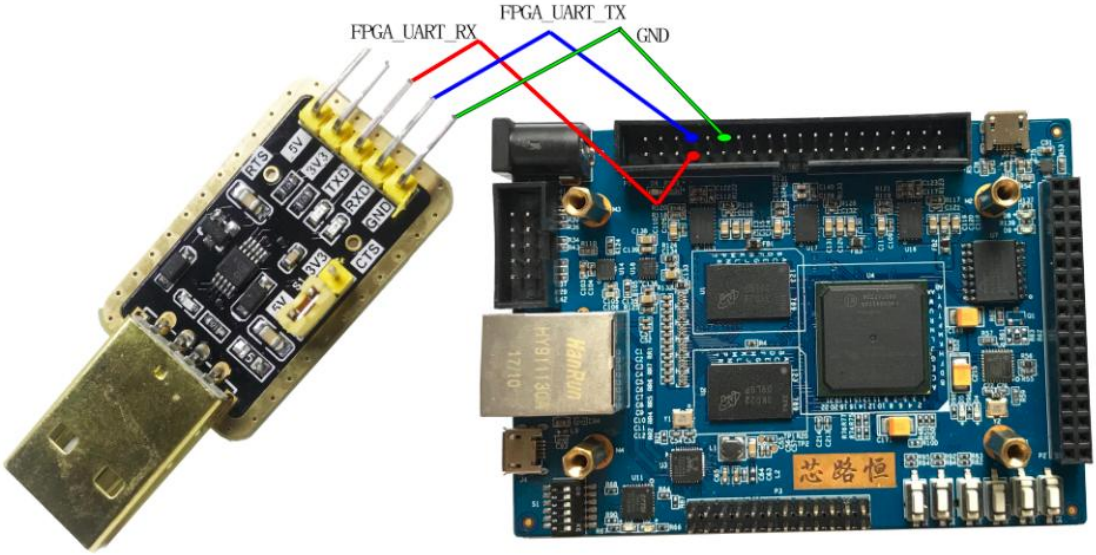


图 xxx 连接 USB 转串口模块和 AC501-SoC 开发板

实验时，将该 USB 转串口模块连接到 PC 的 USB 端口，通过设备管理器查看到准确的串口号之后，打开串口调试助手，设置发送和接收均为 ASCII 格式，波特率为 9600，然后打开该端口号。将 DS-5 中编译好的 fpga_uart 可执行文件使用 DS-5 中的 SSH 工具或者 winSCP 工具将其拷贝到开发板中，使用“chmod +x fpga_uart”命令为其添加可执行权限后，在开发板的终端窗口中输入“./fpga_uart”命令运行该程序，就可以在串口调试软件中看到打印的以下内容了

```
Hello World!
Hello SoC FPGA!

www.corecourse.cn
```

在串口调试助手的发送窗口中输入一串字符串并发送，例如输入“Hello，AC501-SoC”，按下回车键后点击发送。在开发板的终端窗口中就可以看到系统打印的接收到的数据内容了，也为“Hello，AC501-SoC”，如图 xxx 所示。需要注意的是，输入完字符串后一定得加入换行后再发送，不然程序将无法正确正确识别字符串的结束。

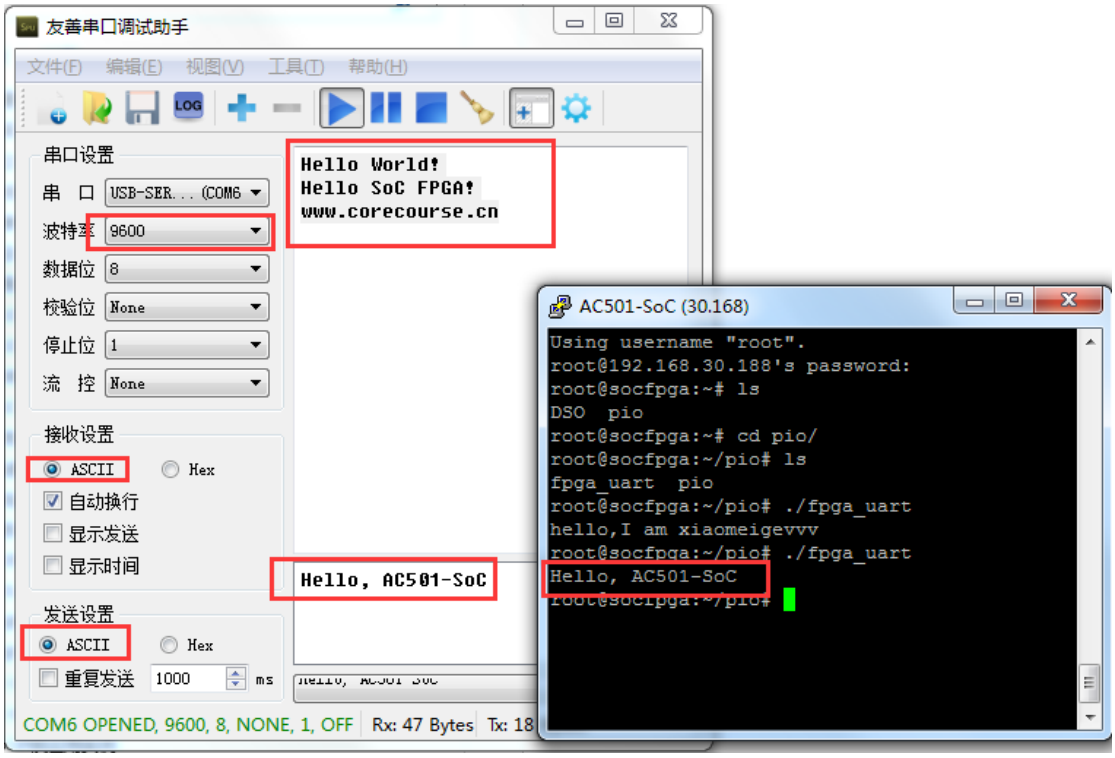


图 xxx UART 收发实验结果

总结

本节实验通过 uart_0 外设的编程实验，进一步复习了基于虚拟地址方式操作外设寄存器的方法，同时给出了简单的 uart 串口编程实例。由于无法使用中断，因此在同时兼顾发送和接收上存在一定的难点，本节内容并未针对该为内容做探讨。本节实验中的代码适合点对点一主一从式的简单数据收发，如需完整的串口应用功能，建议使用 Linux 提供的该 IP 核的内核驱动来实现。使用内核驱动程序来控制该串口的相关内容，将在本书后续章节讲到。

基于虚拟地址映射的 SPI 编程应用