

版本信息：

版本

REV2018

时间

12/16/2017

ZYNQ 修炼秘籍

基于米联客系列开发板

第四季 基于 ZYNQ 硬件的 LINUX 开发

电子版自学资料

常州一二三电子科技有限公司
溧阳米联电子科技有限公司
版权所有

米联客学院 03QQ 群： 543731097

米联客学院 02QQ 群： 86730608

米联客学院 01QQ 群： 34215299

版本	时间	描述
Rev2016	2015-07-25	第一版初稿，大部分采用 zedboard 资料
Rev2017	2017-01-31	做了重大改进，自己编写里批处理命令，方便移植
Rev2018	2017-12-16	对 2017 版本改进，修改教程 bug 同时增加更多学习课程

感谢您使用米联客开发板团队开发的 ZYNQ 开发板，以及配套教程。本教程将对之前编写的《ZYNQ 修炼秘籍》-LINUX 部分内容做出改进，并且增加新的课程内容。本教程不仅仅适合用于米联客开发板，而且可以用于其他的 ZYNQ 开发。

软件版本：VIVADO2015.4 (linux 部分安装主要用到里面的交叉编译环境)

软件版本：VIVADO2016.4 (首期代码用 2016.4,读者可以自行升级到高版本)

软件版本：VIVADO2017.4 (2017.4 预计在 2018 年 1 月官方发布软件)

版权声明：

本手册版权归常州一二三电子科技有限公司/溧阳联电子科技有限公司所有，并保留一切权利，未经我司书面授权，擅自摘录或者修改本手册部分或者全部内容，我司有权追究其法律责任。

技术支持：

版主大神们都等着大家去提问--电子资源论坛 www.osrc.cn

微信公众平台：电子资源论坛



目录

ZYNQ 修炼秘籍	1
目录	4
第四季 基于 ZYNQ 的 LINUX 系统开发开发	6
S04_CH01_搭建工程移植 LINUX/测试 EMMC/HDMI	7
1.1 概述:	7
1.2 LINUX 开发环境搭建	8
1.2.1 虚拟机环境配置（提供下载虚拟机已经完成）	8
1.2.2 下载资源	8
1.3 VIVADO 工程的搭建	9
1.3.1 VIVADO 硬件工程构架	9
1.3.2 时钟设置	9
1.4 PS 设置	12
1.4.1 PS SDK 测试显示器输出	12
1.4.2 测试效果	15
1.4.3 新建 FSBL 工程	15
1.4.4 产生设备树	16
1.5 编译 u-boot、kernel、设备树和文件系统	17
1.5.1 批处理文件	17
1.5.2 修改设备树	19
1.5.3 添加 framebuffer 驱动	22
1.5.4 执行 mk_kernel.sh 编译内核	24
1.5.5 执行 mk_bootloader.sh 编译 uboot	25
1.5.6 制作 UBOOT.BIN	25
1.6 EMMC 8GB 内存测试(MZ701Amini 不支持)	25
1.7 测试 framebuffer	27
1.8 小结	29
S04_CH02_工程移植 ubuntu 并一键制作启动盘	30
2.1 概述	30
2.2 搭建硬件系统	30
2.3 一键制作	30
S04_CH03_QSPI 烧写 LINUX 系统	31
3.1 概述	31
3.2 搭建硬件系统	31
3.3 修改内核文件	31
3.3 编译内核及 uboot	34
3.4 制作 qspi 镜像	34
3.5 安装 screen	35
3.6 一件烧写 QSPI FLASH 1	36

3.7 烧写 QSPI FLASH 2	37
S04_CH04_自动挂载 8GB EMMC 板载内存	40
4.1 概述	40
4.2 执行 source setup_env.sh	40
4.3 修改 zynq-7000.dtsi 文件	40
4.4 设置 mount_emmc.sh 批处理命令的开机启动	42
4.5 烧写程序到 QSPI FLASH	44
4.6 验证测试	44
4.7 思考为什么	45
S04_CH05_在线升级 QSPI 镜像(U 盘方式)	47
5.1 概述	47
5.2 执行 source setup_env.sh	47
5.3 烧写程序到 QSPI FLASH	47
5.4 查看系统根目录	47
5.5 基于 U 盘在线升级	48
S04_CH06_hello_linux	51
6.1 概述	51
6.2 执行 source setup_env.sh	51
6.3 SD 卡手动运行 hello 程序	51
6.4 EMMC 卡手动运行 hello 程序	53
S04_CH07_Hello_Qt 在开发板上的运行	54
7.1 概述	54
7.2 搭建交叉编译环境	54
7.2.1 使用批处理命令搭建交叉编译环境	54
7.2.2 setup_env.sh 批处理文件源码	59
7.2.3 get_qt_sources.sh 批处理文件源码	61
7.2.4 mk_qt_img.sh 批处理文件源码	62
7.2.4 init.sh 文件	65
7.2.5 测试结果	65
7.3 在 PC 端 LINUX 安装 qt5.8.0	66
7.4 QtE LINUX PC 端创建工程	71
7.5 对 QtE 设置交叉编译	78
7.6 测试结果	83

第四季 基于 ZYNQ 的 LINUX 系统开发

第四季课程共计 16 课时，主要讲解 LINUX 开发环境搭建，LINUX 如何移植，如何修改设备树，批处理文件的使用和理解。EMMC 测试、编译 QSPI FLASH UBOOT.BIN 文件，烧写 UBOOT.BIN 到 QSPI FLASH。通过 U 盘烧写 UBOOT.BIN。

然后会讲解 LINUX 驱动入门，QT 变成入门、OPENCV 移植入门等教程。

由于目前本季课程还在更新，请大家耐心等待，并且关注我们 VIP QQ 群的最新发布。

S04_CH01_搭建工程移植 LINUX/测试 EMMC/HDMI

1.1 概述：

本章内容是在已经提供安装了 VIVADO2015.4 的 ubuntu 系统下，进行。大家可以下周我们已经提供的虚拟机镜像，我们提供的虚拟机镜像是安装了 VIVADO 的 ubuntu 系统，系统版本是 ubuntu14.04。

主要完成的内容如下：

- 1)、利用 VIVADO 搭建 VDMA Framebuffer 工程 修改 VTG IP 模块 支持 1024X600 分辨率 (主要考虑支持 7 寸 HDMI 液晶显示器)
- 2)、产生 FSBL 文件
- 3)、环境变量的批量设置
- 4)、bootloader 和 kernel 部分修改设备树(修改官方的设备树，可以简化很多开发过程)
- 5)、修改 kernel 其他文件
- 6)、通过 menuconfig 向导配置 framebuffer 驱动
- 7)、编译 kernel、编译 u-boot
- 8)、测试显示器输出和串口打印信息
- 9)、测试读 EMMC 内存、写 EMMC 内存、读写 EMMC 内存
- 10)、测试 framebuffer 应用程序
- 11)、提供的配套开发板 ubuntu 环境全部搭建好了，读者如果不想学习修改的过程，可以跳过所有修改过程，直接输入如下指令直接编译。

以上步骤比较繁琐，如果读者只是在我们的开发板上移植系统可以跳过以上步骤快速执行以下执行完成。

对于 MZ701A/MZ702A/MZ702B/MZ702N 开发板 cfg_bootloader.sh 文件需要选用

make -C \${UBOOT_DIR} zynq_zybo_defconfig

对于 MIZ7035 开发板或者 MIZ702N 开发板选用

make -C \${UBOOT_DIR} zynq_zed_defconfig

sudo su

root

cd /mnt/workspace/linux/scripts

source setup_env.sh

cfg_bootloader.sh

cfg_kernel.sh

mk_bootloader.sh

mk_kernel.sh

mk_sd_image.sh

之后复制 image 文件夹下的内容到 TF 卡，之后插入开发板 TF 卡插口就可以启动了。下面给出了修改的详细过程。

1.2 LINUX 开发环境搭建

1.2.1 虚拟机环境配置（提供下载虚拟机已经完成）

Step1:

本例程的工作环境（包括 FPGA 及嵌入式 Linux 的开发）是在 ubuntu14.04 操作系统下完成，对于其它 Linux 操作系统可能需要解决相关包的依赖问题。

Step2:

例子放在 /mnt/workspace/linux 目录下，读者可以在该目录下正确编译、运行。而 /mnt/workspace/linux 目录是为读者实验准备的。

Step3:

单击桌面上的控制台或者 (ctrl+alt+T) 即可打开命令行，然后输入 su，根据提示输入 root 密码即可切换到 root 用户。

Step4:

对于新安装的 ubuntu 操作系统需要命令行下运行一下 scripts 目录下的 fix_xilinx_deps.sh 脚本，该脚本主要是解决编译 u-boot、kernel 源码等所需要的包依赖。而提供的虚拟机已经解决了这些问题。
#/mnt/workspace/linux/scripts/fix_xilinx_deps.sh

注意：开机的时候可能系统会提示正在检查更新，此时可以打开图示的有个勾的那个图标，然后点击 Install Updates，待其完成更新后再运行该脚本（如下图所示）。当然，在平时的开发过程中，可以在打开虚拟机之前，禁止网络功能，提供的虚拟机已经禁止了。笔者建议不要更新系统，以免造成一些兼容性问题。

Step5:

对于 Vivado 开发工具的安装，将下载的压缩包解压后，从命令行进入该目录，执行 xsetup 即可像在 Windows 一样安装。注意：为减少虚拟机所占用硬盘空间，虚拟机里提供的开发套件是直接将本人 PC 中安装好的 Vivado 和 SDK 等复制到 /mnt/workspace/toolchains 目录中的，这里不提供全新安装 Vivado 的步骤，若需要帮助的话，可以通过邮件联系我。

Step6:

本开发使用的是 Vivado 开发套件里提供的交叉编译器，无须再安装其它交叉编译器。

Step7:

整个开发过程主要使用脚本进行操作，故在每次开发前，需要执行如下图所示操作来设置好环境变量。

1.2.2 下载资源

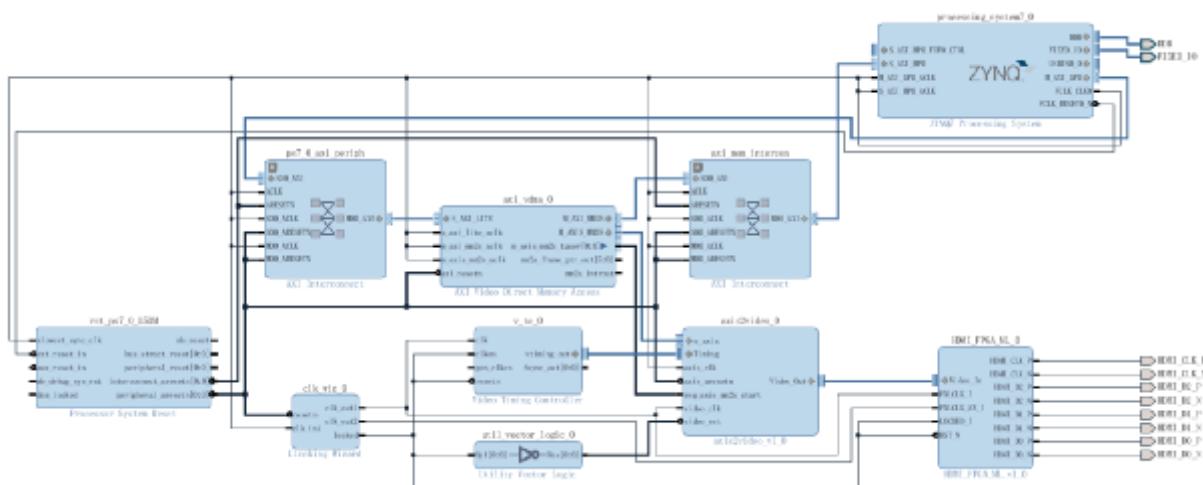
使用 get_xilinx_sources.sh 脚本下载 uboot、kernel、device tree 等源码及 ramdisk 到 packages 目录中，并解压 uboot 和 kernel 等源码。若需要更改源码的版本，则打开 get_xilinx_sources.sh 文件后，修改相应的设置即可。当网络不是很好时，可以直接压缩包目录中的包复制到 packages 目录下即可。

注意：提供的 ubuntu 系统已经完成以上任务，当然读者可以自己再做一遍。

1.3 VIVADO 工程的搭建

设计 FPGA 这部分相信读者已经相当熟悉了，这里只是对工程里的一些关键地方进行说明。笔者提供的 ubuntu 系统里面已经安装的是 vivado2015.4（主要是安装包小），而目前新的代码可能是 2016.4 和 2017.4。所以建议初学者 WINDOWS 下使用 VIVADO 工程。

1.3.1 VIVADO 硬件工程构架



1.3.2 时钟设置

Step1：双击 ZYNQ CPU IP 进行如下步骤设置：

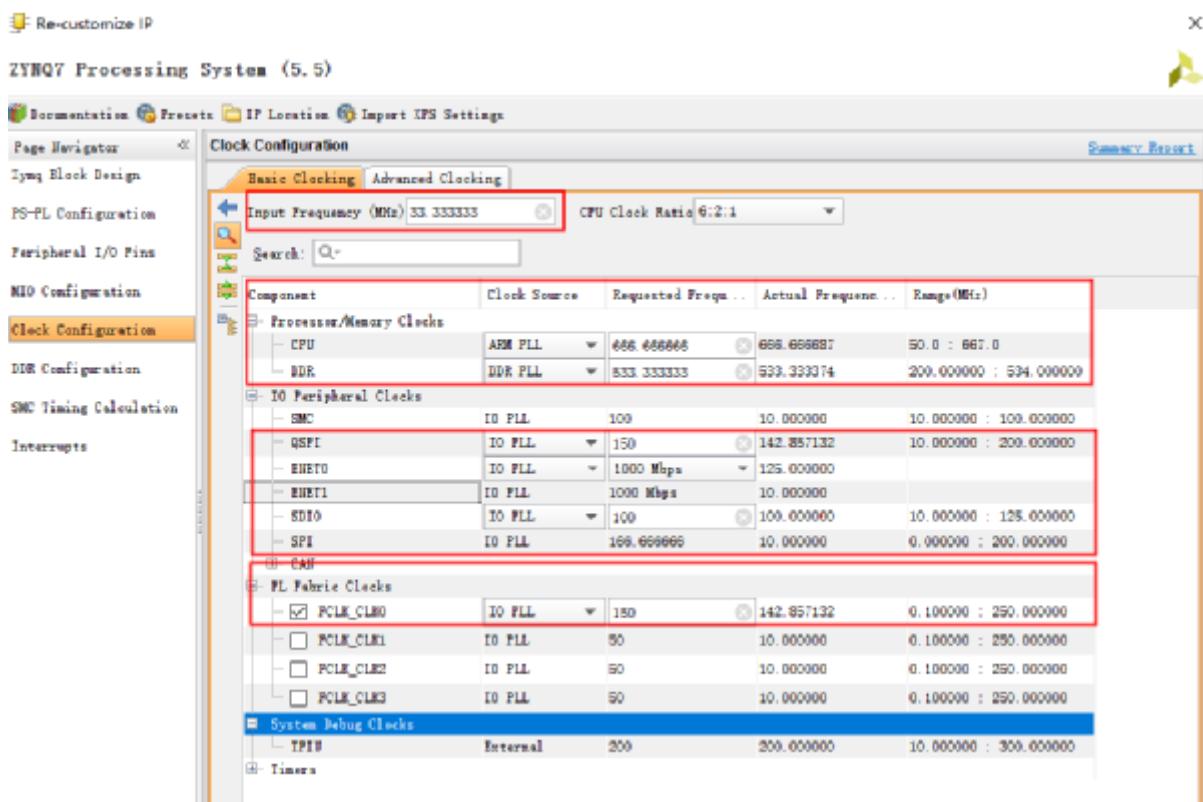
对于 MZ701mini/MZ701A/MZ702A/MZ702B/MZ702N/MZ7035

A)、输入时钟是 333.333333MHz

B)、对于速度等级-1 的 7010 或者 7020 芯片 CPU 主频最高设置到 667M，越高速度等级的芯片可以设置最高的频率就越高。笔者这里暂且设置 667M 这样所有芯片都可以支持。读者可以自行设置频率，挑战下芯片的性能。

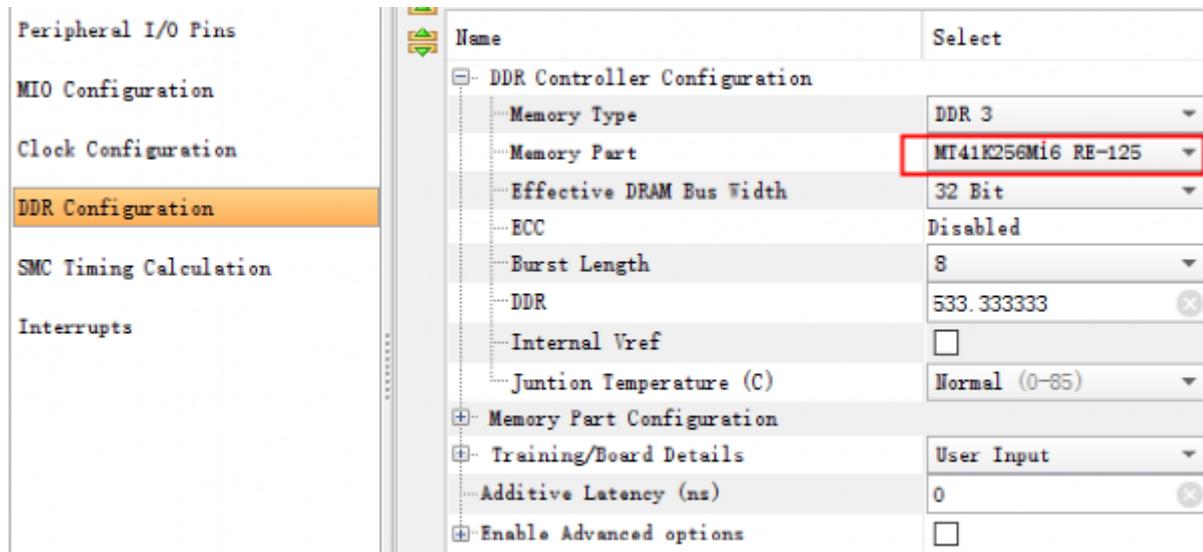
C)、QSPI 的时钟设置为 150M 对于速度等级-1 的芯片设置太高了，可能无法从 QSPI 启动。对于速度等级 2 的芯片笔者测试可以设置到 200M 正常运行

D)、FCLK_CLK0 提供给 PL 的时钟频率改为 150M, 这个时钟用于 AXI 总线的通信时钟，如果速度太高了，导致系统不稳定，可以把这个频率降低一些。笔者这里设置 150M 测试可以稳定运行。

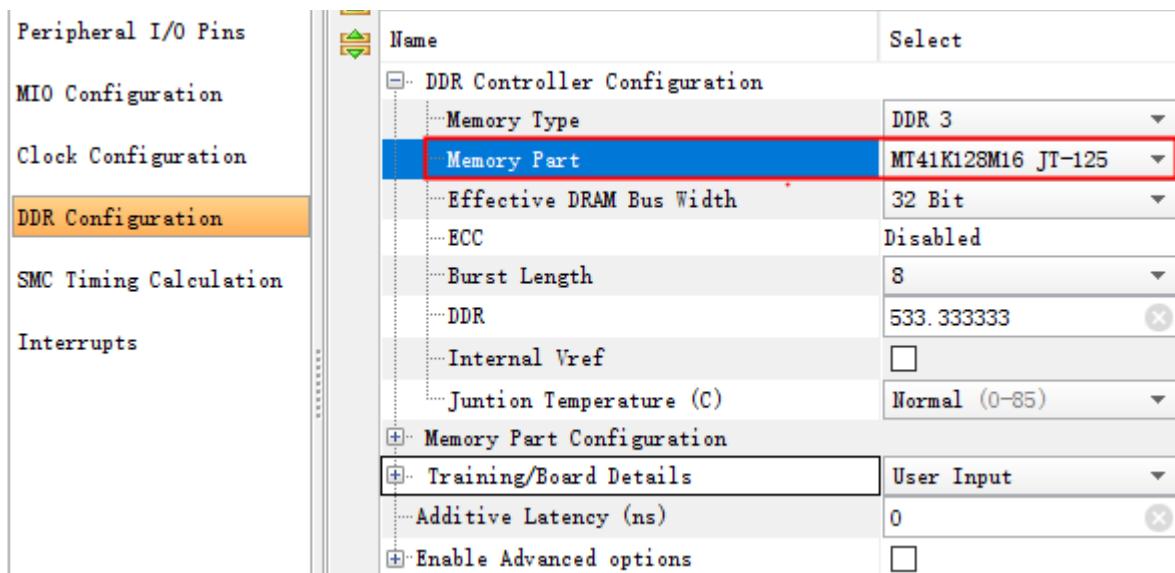


Step2: 设置内存型号:

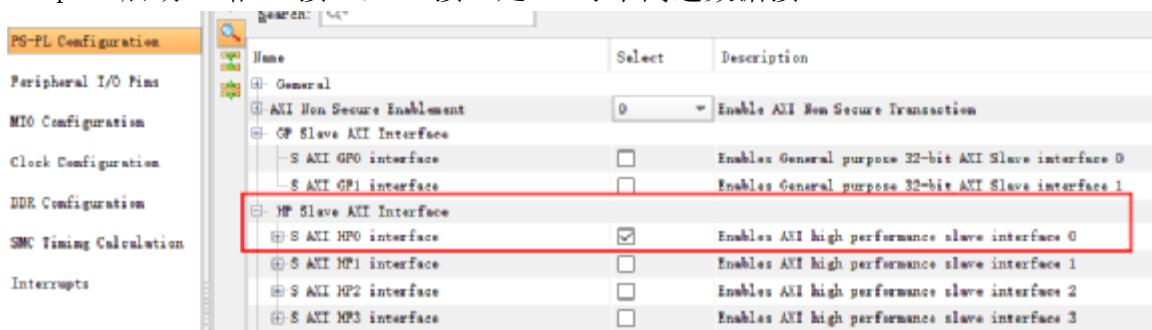
A)、MZ701A/MZ702A/MZ702B/MZ702N/MZ7035 设置为单片 512MB 的 MT41K256M16 RE-125



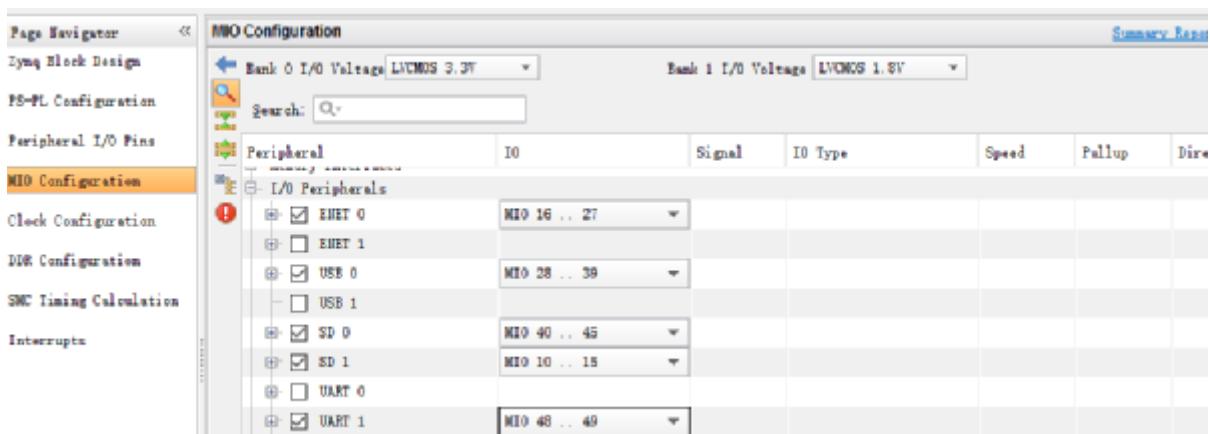
B)、对于 MZ701Amini 的设置 单片 256MB 的 MT41K128M16JI-125



Step3: 启动 1 路 HP 接口，HP 接口是 ZYNQ 个高速数据接口

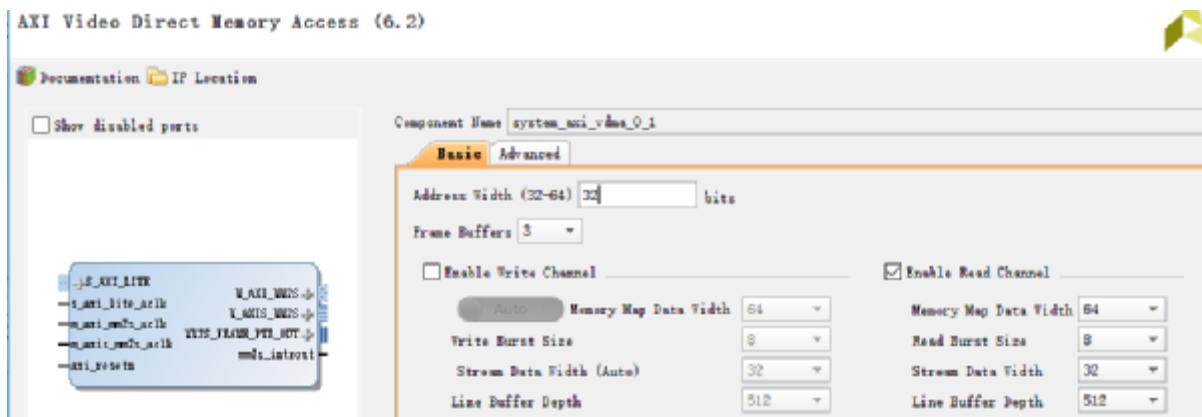


Step4: 增加必要的外设，包括 ENET0 以太网接口、USB_0 USB 接口、SD0 TF 卡接口、SD1 EMMC 接口、UART1 串口。(MZ701Amini/MIZ702 没有焊接 EMMC 所以无需勾选 SD1)。读者可以参考我们工程进行配置。



Step5: 设置完成后单击 OK

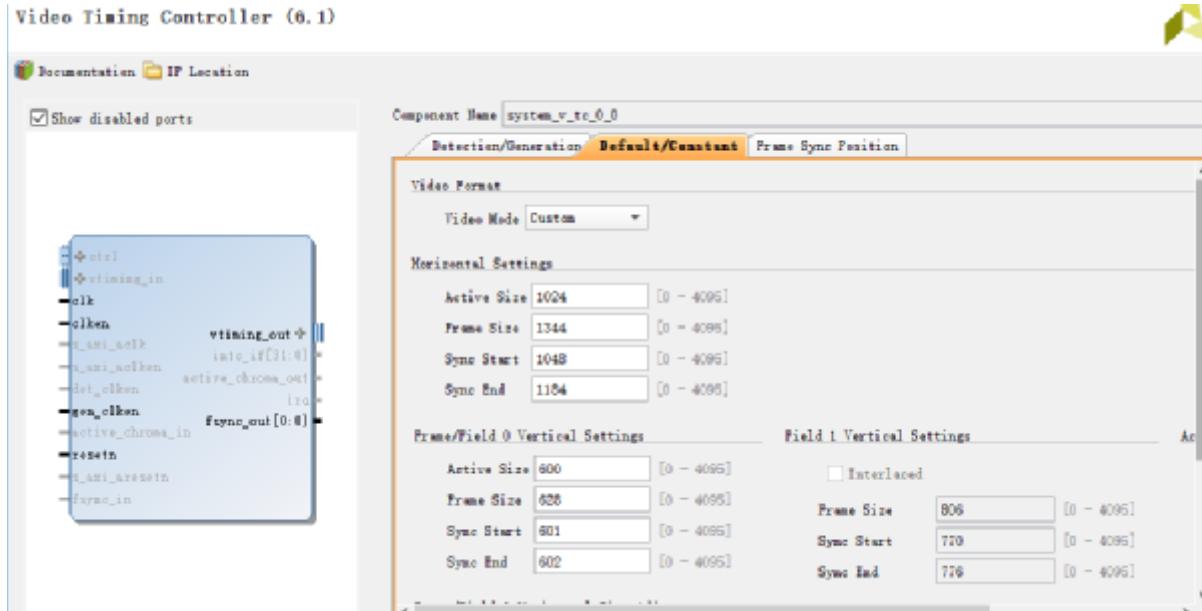
Step6: 双击 VDMA IP 由于只使用了 VDMA 读通道设置如下：



Step7: 双击 PLL 时钟 IP 设置 HDMI 输出时钟，不同的分辨率需要修改不同的时钟

The phase is calculated relative to the active input clock.									
Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drivers	Matched Routing
		Requested	Actual	Requested	Actual	Requested	Actual		
<input checked="" type="checkbox"/> clk_sut1	clk_out1	50.04	50.638	0.000	0.000	50.000	50.0	BURG	+
<input checked="" type="checkbox"/> clk_sut2	clk_out2	253.2	253.189	0.000	0.000	50.000	50.0	BURG	+
<input type="checkbox"/> clk_sut3	clk_out3	100.000	100.000	0.000	0.000	50.000	100.0	BURG	+
<input type="checkbox"/> clk_sut4	clk_out4	100.000	100.000	0.000	0.000	50.000	100.0	BURG	+
<input type="checkbox"/> clk_sut5	clk_out5	100.000	100.000	0.000	0.000	50.000	100.0	BURG	+
<input type="checkbox"/> clk_sut6	clk_out6	100.000	100.000	0.000	0.000	50.000	100.0	BURG	+
<input type="checkbox"/> clk_sut7	clk_out7	100.000	100.000	0.000	0.000	50.000	100.0	BURG	+

Step8: 修改 VTC 显示时序发生 IP 参数符合 1024X600 分辨率



1.4 PS 设置

1.4.1 PS SDK 测试显示器输出

新建 Display_VMDA_Test 空的工程，为了测试在裸机下图形系统显示正确，编写 SDK 测试代码 main.c 函数以及其他必要函数。

```
/*
 *南京米联电子科技有限公司
 *www.milinker.com
 *www.osrc.cn
 *test display
 Copyright (c) 2009-2012 Xilinx, Inc. All rights reserved.
 */

#include "xaxivdma.h"
#include "xaxivdma_i.h"
#include "sleep.h"

#define DDR_BASEADDR          0x00000000
#define VDMA_BASEADDR         XPAR_AXI_VDMA_0_BASEADDR
#define H_STRIDE               1024
#define H_ACTIVE                1024
#define V_ACTIVE                600
#define pi                      3.14159265358
#define COUNTS_PER_SECOND      (XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ)/64

#define VIDEO_LENGTH (H_STRIDE*V_ACTIVE)
#define VIDEO_BASEADDR0 DDR_BASEADDR + 0x2000000
#define VIDEO_BASEADDR1 DDR_BASEADDR + 0x3000000
#define VIDEO_BASEADDR2 DDR_BASEADDR + 0x4000000

u32 *BufferPtr[3];

unsigned int srcBuffer = (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x1000000);
int run_triple_frame_buffer(XAxiVdma* InstancePtr, int DeviceId, int hsize,
    int vsize, int buf_base_addr, int number_frame_count,
    int enable_frm_cnt_intr);

//函数声明
void Xil_DCacheFlush(void);
// 所有数据格式 为 RGBA, 低位的透明度暂不起作用
extern const unsigned char gImage_beauty[1729536];
extern const unsigned char gImage_mm[1228800];
extern const unsigned char gImage_miz702[600000];
extern const unsigned char gImage_miz702_rgba[600000];

void show_img(u32 x, u32 y, u32 disp_base_addr, const unsigned char * addr, u32 size_x, u32 size_y)
{
    //计算图片 左上角坐标
```

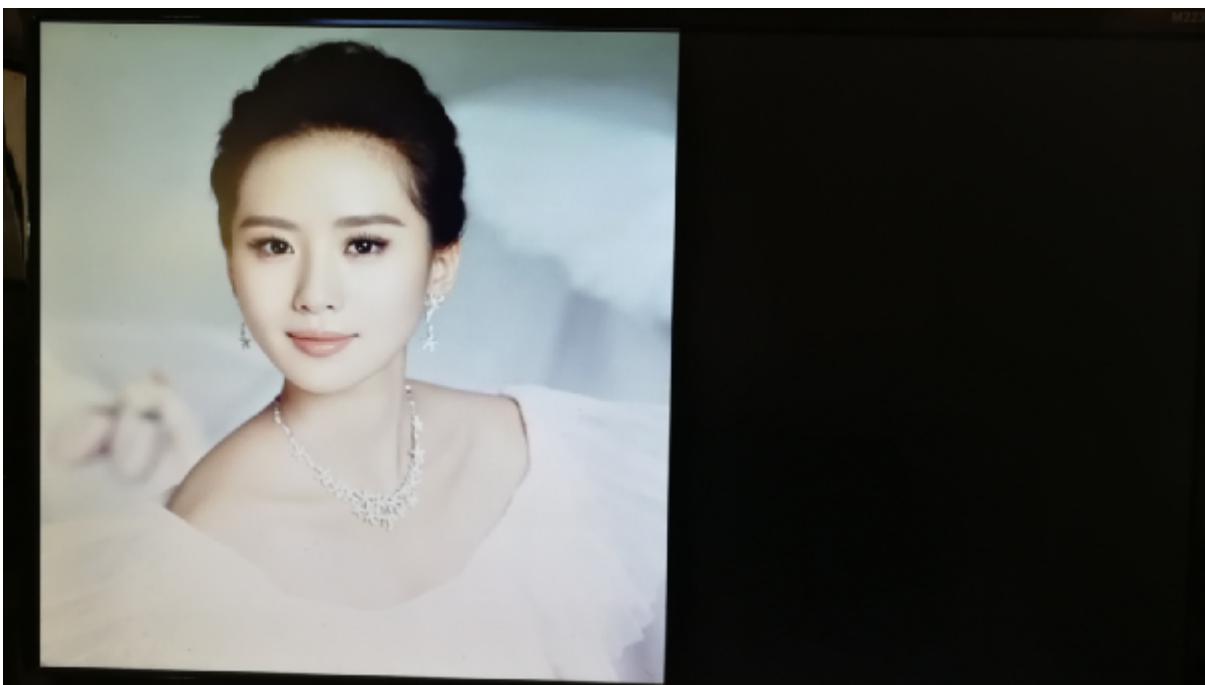
```
u32 i=0;
u32 j=0;
u32 r,g,b;
u32 start_addr=disp_base_addr;
start_addr = disp_base_addr + 4*x + y*4*H_STRIDE;
for(j=0;j<size_y;j++)
{
    for(i=0;i<size_x;i++)
    {
        b = *(addr+(i+j*size_x)*4+1);
        g = *(addr+(i+j*size_x)*4+2);
        r = *(addr+(i+j*size_x)*4+3);
        Xil_Out32((start_addr+(i+j*H_STRIDE)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
    }
}
Xil_DCacheFlush();
}

int main(void)
{
    u32 i;
    //Xil_DCacheFlush();
    xil_printf("Starting the first VDMA \n\r");
    //VDMA configureAXI VDMA0
    /*****往 DDR 写数据设置*******/
    //Xil_Out32((VDMA_BASEADDR + 0x030), 0x00000003); // enable circular mode
    //Xil_Out32((VDMA_BASEADDR + 0x0AC), VIDEO_BASEADDR0); // start address
    //Xil_Out32((VDMA_BASEADDR + 0x0B0), VIDEO_BASEADDR1); // start address
    //Xil_Out32((VDMA_BASEADDR + 0x0B4), VIDEO_BASEADDR2); // start address
    //Xil_Out32((VDMA_BASEADDR + 0x0A8), (H_STRIDE*4)); // h offset (640 * 4) bytes
    //Xil_Out32((VDMA_BASEADDR + 0x0A4), (H_ACTIVE*4)); // h size (640 * 4) bytes
    //Xil_Out32((VDMA_BASEADDR + 0x0A0), V_ACTIVE); // v size (480)
    /*****从 DDR 读数据设置*******/
    Xil_Out32((VDMA_BASEADDR + 0x000), 0x00000003); // enable circular mode
    Xil_Out32((VDMA_BASEADDR + 0x05c), VIDEO_BASEADDR0); // start address
    Xil_Out32((VDMA_BASEADDR + 0x060), VIDEO_BASEADDR0); // start address
    Xil_Out32((VDMA_BASEADDR + 0x064), VIDEO_BASEADDR0); // start address
    Xil_Out32((VDMA_BASEADDR + 0x058), (H_STRIDE*4)); // h offset (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x054), (H_ACTIVE*4)); // h size (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x050), V_ACTIVE); // v size (480)
    for(i=0;i<614400;i++)
    {
```

```
Xil_Out32(VIDEO_BASEADDR0+i,0);
}
while(1)
{
    show_img(0,0,VIDEO_BASEADDR0,&gImage_beauty[0],563,600);
    sleep(5);
    show_img(0,0,VIDEO_BASEADDR0,&gImage_miz702_rgba[0],375,400);
    sleep(5);
}

return 0;
}
```

1.4.2 测试效果



1.4.3 新建 FSBL 工程

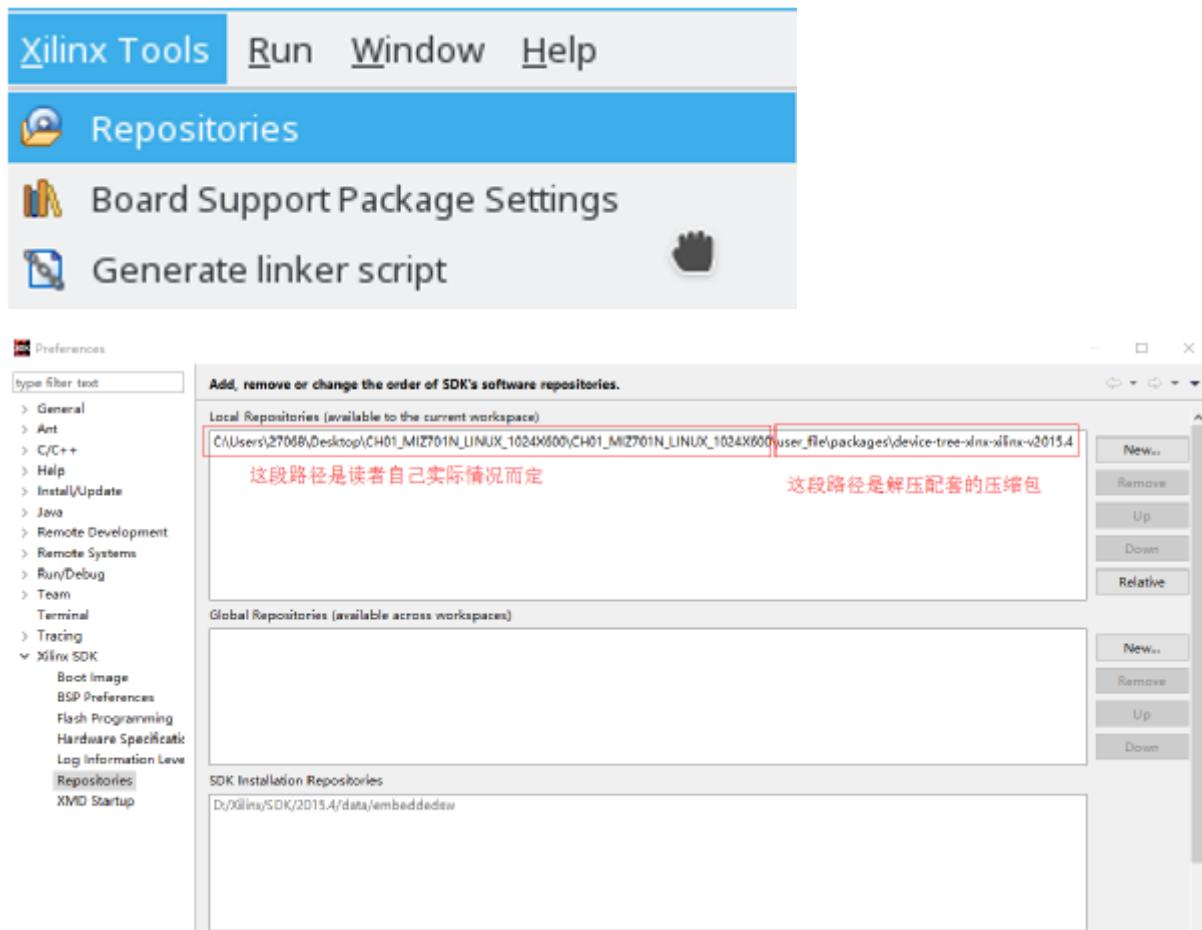
产生的 fsbl.elf 后面将用于参数 BOOT.BIN 文件

- > fsbl
- > fsbl_bsp
- > Linux_Test
- > Linux_Test_bsp
- > system_wrapper_hw_platform_0

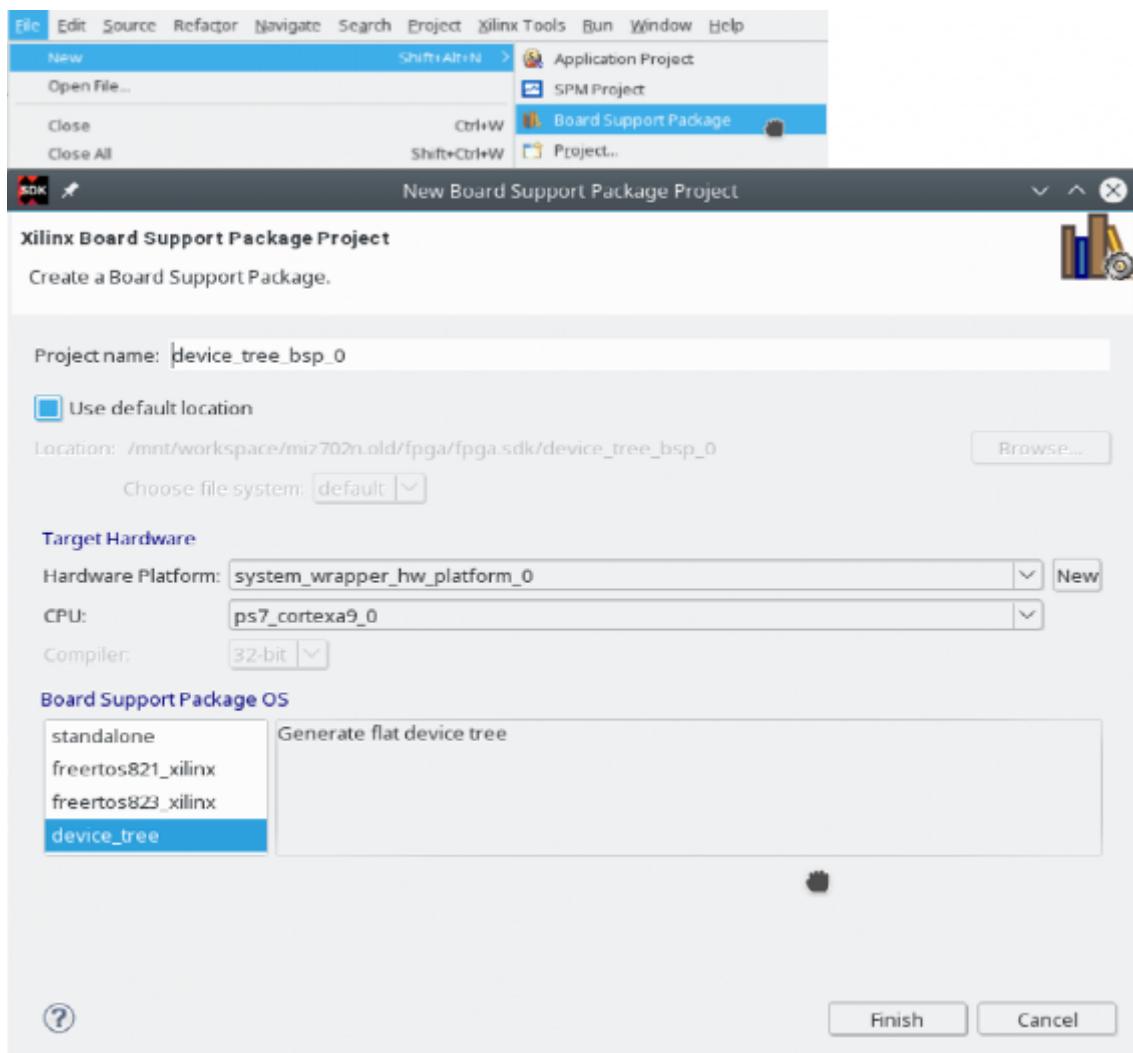
1.4.4 产生设备树

本教程可以省去 1.4.4 所有内容

Step1:首先解压 device-tree-xlnx-xilinx-v2015.4.tar.gz, 然后打开 Xilinx Tools→Repositories 将刚才解压的目录包含进来。



Step2:打开 File→New→Board Support Package 创建,其它弹出窗口按默认设置即可



Step3: 在该工程中，zynq-7000.dtsi 文件是 Xilinx 提供给所有 zynq-7000 开发板使用的，只所有的 status 都 设置为” disabled”，而 system.dts 里根据板子上的具体实现，修改设备树使其可以正常工作。pl.dtsi 文件 是 FPGA 里使用的 IP 对应的设备树。

1.5 编译 u-boot、kernel、设备树和文件系统

1.5.1 批处理文件

这里使用 xilinx 在 github 上提供的 u-boot 和 kernel 源码，在 Wiki 上提供的文件系统（当然，也可以直 接使用 buildroot 自己匹配根文件系统，该方法简单快捷，不用再去解决什么依赖问题，但好像国内很 少有人这么干，他们都把 busybox 等模块一个个搭建起来）。1、配置 uboot 和编译 uboot 自带的工具，为编译 kernel 提供 mkimage 工具的支持。注意，由于编译新 版本的 uboot 需要 openssl 和 dtc 的支持，这里呢，在系统里已经安装好了 openssl，而 dtc 则利用 kernel 里自带的，所以这一步做完，我们将直接编译内核，等编译完内核后再来编译 uboot。注意：cfg_bootloader.sh 可以在任意目录下使用，而 make tools 只能在 bootloader 目录下

使用，bootloader 就是我们存放 uboot 源码的地方。（注意：这部分仅在恢复配置文件至 Xilinx 提供的配置时用，在自己修改完配置之后，除了需要恢复配置文件外，请谨慎使用这两条命令）。

Step1:运行 setup_env.sh 批处理文件(管理员模式下的密码为 root)

```
osrc@osrc-virtual-machine:~$ sudo su
[sudo] password for osrc:
root@osrc-virtual-machine:/home/osrc# cd /mnt/workspace/linux/scripts
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# ls
cfg_bootloader.sh  get_xilinx_sources.sh  mk_bootloader.sh  setup_env.sh
cfg_kernel.sh      install_linaro_image.sh  mk_kernel.sh    vivado.jou
echo_color.sh       install_sd_image.sh    mk_sd_image.sh  vivado.log
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# source setup_env.sh
```

Step2:运行 cfg_bootloader.sh 批处理文件

对于 MZ701A/MZ702A/MZ702B/MZ702N 开发板 cfg_bootloader.sh 文件需要选用

make -C \${UBOOT_DIR} zynq_zybo_defconfig

对于 MIZ7035 开发板或者 MIZ702N 开发板选用

make -C \${UBOOT_DIR} zynq_zed_defconfig

```
root@osrc-virtual-machine:/mnt/workspace/linux# cd ./bootloader
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader# cfg_bootloader.sh
Configure U-Boot on the /mnt/workspace/linux/bootloader
make: Entering directory '/mnt/workspace/linux/bootloader'
#
# configuration written to .config
#
make: Leaving directory '/mnt/workspace/linux/bootloader'
U-Boot - configure Done.
```

Step2:运行 make tools 处理命令（执行的是清理工作，重置配置时候使用）

```
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader# make tools
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK  include/config.h
  GEN  include/autoconf.mk
  GEN  include/autoconf.mk.dep
  GEN  spl/include/autoconf.mk
  CHK  include/config/uboot.release
  CHK  include/generated/version autogenerated.h
  CHK  include/generated/timestamp autogenerated.h
  UPD  include/generated/timestamp autogenerated.h
  CHK  include/generated/generic-asm-offsets.h
  CHK  include/generated/asm-offsets.h
  HOSTCC tools/mkenvimage.o
  HOSTLD tools/mkenvimage
  HOSTCC tools/fit_image.o
  HOSTCC tools/image-host.o
  HOSTCC tools/dumpimage.o
  HOSTLD tools/dumpimage
  HOSTCC tools/mkimage.o
  HOSTLD tools/mkimage
  HOSTLD tools/fit_info
  HOSTLD tools/fit_check_sign
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader#
```

Step3:配置内核(注意：这部分仅在恢复配置文件至 Xilinx 提供的配置时用，在自己修改完配置之后，除了需要恢复配置文件外，请谨慎使用这条命令)

```
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader# cfg_kernel.sh
Configure Linux kernel on the /mnt/workspace/linux/kernel
make: Entering directory '/mnt/workspace/linux/kernel'
#
# configuration written to .config
#
make: Leaving directory '/mnt/workspace/linux/kernel'
Linux Kernel - configure Done.
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader#
```

1.5.2 修改设备树

Step1:bootloader 的修改

A)、MZ701Amini/MZ701A/MZ702A/MZ702B/MZ702N 使用的网口芯片是 RTL8211 正好 zybo 的网口芯片是采用的 RTL8211 所以可以使用 zybo 配置部分。这里修改。路径 /mnt/workspace/linux/bootloader/include/configs/zynq_zybo.h 文件如下。

```
/* Define ZYBO PS Clock Frequency to 33.33333MHz */
#define CONFIG_ZYNQ_PS_CLK_FREQ 3333333UL

#include <configs/zynq-common.h>

#endif /* __CONFIG_ZYNQ_ZYBO_H */
```

并且修改路径/mnt/workspace/linux/bootloader/arch/arm/dts/zynq_zybo.dts

zybo 的设备数中的时钟改为 33.33333MHZ,笔者测试中发现,如果 bootloader 设备数中加载网口会导致网口芯片工作不正常,所以这里还要删除 zynq_zybo.dts 中的网口部分设备树。

```
aliases {
    serial0 = &uart1;
    spi0 = &gspi;
    mmc0 = &sdhci0;
};

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

chosen {
    bootargs = "";
    stdout-path = "serial0:115200n8";
};

usb_phy0: phy0 {
    compatible = "usb-nop-xceiv";
    #phy-cells = <0>;
    reset-gpios = <&gpio0 46 1>;
};
};

soclk {
    ps-clk-frequency = <33333333>;
};

gspi {
    u-boot,dm-pre-reloc;
    status = "okay";
};
```

D)、MIZ7035 采用的 88E1518 由于网络部分参考了，zedbaord 的设计方案，为了简化移植可以使用 zedboard 的驱动部分。这里什么都不改。

Step2:kernel 的修改

由于各种开发板可以参考 zedboard 的配置设备参数，因此我们可以修改已经存在的相关文件，来满足我们的自定义需求打开 /mnt/workspace/linux/kernel/arch/arm/boot/dts/zynq-zed.dts

修改 bootargs 信息 启动参数 bootargs 是传递给 kernel 的参数。Console 是一个输出系统管理信息的文本输出设备，这些信息来自于内核，系统启动和系统用户，其中 console=ttyPS0,115200 用来设置串口作为输出终端设备，是这些信息可以通过串口在远程的终端上显示。而 console=tty0 则是设置显示器作为输出终端。

如果想屏幕永不休眠，则在启动参数 bootargs 中增加 consoleblank=0，网络上大部分是讲修改内核源码，但我觉得这种方法才是最方便，且符合一个内核适合多种产品的思想。其它参数读者应该也知道是什么意思，这里也就不多讲了。

```
chosen {
    /* bootargs = ""; */
    bootargs = "console=ttyPS0,115200 console=tty0 consoleblank=0 root=/dev/ram rw earlyprintk";
    stdout-path = "serial0:115200n8";
};
```

Step3:添加 vdma 和 framebuffer 设备树节点，注意：这里是添加到根节点。笔者觉得，arm linux 引进设备树，比以前版本的做法好很多，只是驱动里需要设备树提供哪些参数，在 Linux 文档里没有很好的说明，或者是驱动里所需要的参数改变了，而文档又没有及时更新，这部分是 linux 的不足之处。所以，开发 Linux 过程中，最好还是以源码为中心。设备树这部分是参考之前在 SDK 里产生的设备树和内核里相关文档，在阅读 vdma 等驱动源码后，修改而来的，并不是说 SDK 里产生的设备树直接搬来就可以正常使用。

```

/ {
    amba_pl: amba_pl {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";
        ranges;
        /* 液晶显示屏 */
        /* framebuffer架构驱动 */
        axi_vdma_0: dma@43000000 {
            compatible = "xlnx,axi-vdma-1.00.a";
            #dma-cells = <1>;
            reg = <0x43000000 0x10000>;
            /* dma-ranges = <0x00000000 0x00000000 0x40000000>; */
            interrupt-parent = <&intc>;
            /* interrupts = <0 33 4>; */
            xlnx,num-fstores = <0x1>;
            xlnx,flush-fsync = <0x2>;
            xlnx,addrwidth = <0x20>;
            clocks = <&clkc 0>;
            clock-names = "s_axi_lite_aclk";
            dma-channel@43000000 {
                compatible = "xlnx,axi-vdma-mm2s-channel";
                interrupts = <0 33 4>;
                xlnx,datawidth = <0x20>;
                xlnx,device-id = <0x0>;
            } ;
        } ;
        axi_vdma_lcd {
            compatible = "topic,vdma-fb";
            dmas = <&axi_vdma_0 0>;
            dma-names = "axivdma";
        } ;
    } ;
}

```

Step4:修改 SD 卡和 emmc 设备树节点

这部分直接将之前在 SDK 里产生的设备树复制过来就可以正常使用。总的来说，对于 PS 的外设，其 硬件是固定的，驱动也基本不会有太大的变化，故基本上是可以直接搬来就可以用，但也有例外，如果 USB，SDK 里并不知道你要当 host 或者 otg 来使用，所以需要做些小修改。而对于 PL 中使用的 IP，其 硬件和驱动经常有变化，SDK 里产生的设备树不能直接拿来使用。对于 MZ701Amini 没有 EMMC 所以不要配置 sdhci1.

```

&sdhci0 {
    status = "okay";
    xlnx,has-cd = <0x1>;
    xlnx,has-power = <0x0>;
    xlnx,has-wp = <0x1>;
};

&sdhci1 {
    status = "okay";
    xlnx,has-cd = <0x1>;
    xlnx,has-power = <0x0>;
    xlnx,has-wp = <0x1>;
};

```

1.5.3 添加 framebuffer 驱动

提供的虚拟机下面的步骤都已经做完了，读者可以跳过，或者再做一遍。

Step1: 把 vdmafb.c 文件复制到 /mnt/workspace/linux/kernel/drivers/video/fbdev 目录下，对于驱动这部分，主要还是要看源码，注意，该驱动仅支持 640x480 分辨率，若需要支持其它分辨率，需要修改和调试相关源码。

Step2: 打开 /mnt/workspace/linux/kernel/drivers/video/fbdev 目录下的 Makefile 文件，当然也可以用其它文本编辑器。

找到 CONFIG_FB_XILINX，在其下方添加 obj-\$(CONFIG_FB_VDMA) += vdmafb.o。读者可以直接复制笔者提供的开发包中的修改好的文件。

```
obj-$(CONFIG_FB_XILINX)          += xilinxfb.o
obj-$(CONFIG_FB_VDMA)           += vdmafb.o
```

Step3: 打开 /mnt/workspace/linux/kernel/drivers/video/fbdev 目录下的 Kconfig 文件：

找到 FB_XILINX，在其下方添加以下信息。读者可以直接复制笔者提供的开发包中的修改好的文件。

```
config FB_XILINX
    tristate "Xilinx frame buffer support"
    depends on FB && (XILINX_VIRTEX || MICROBLAZE || ARCH_ZYNQ || ARCH_ZYNQMP)
    select FB_CFB_FILLRECT
    select FB_CFB_COPYAREA
    select FB_CFB_IMAGEBLIT
    ---help---
        Include support for the Xilinx ML300/ML403 reference design
        framebuffer. ML300 carries a 640*480 LCD display on the board,
        ML403 uses a standard DB15 VGA connector.

config FB_VDMA
    tristate "VDMA frame buffer support"
    depends on FB && ARCH_ZYNQ
    select FB_CFB_FILLRECT
    select FB_CFB_COPYAREA
    select FB_CFB_IMAGEBLIT
    select XILINX_AXIVDMA
    ---help---
        Include support for the VDMA LCD reference design framebuffer.
```

Step4: 打开 /mnt/workspace/linux/kernel/drivers 目录下的 Makefile 文件，找到 obj-y += video/ 读者可以直接复制笔者提供的开发包中的修改好的文件。

```
# GPIO must come after pinctrl as gpios may need to mux pins etc
obj-$(CONFIG_PINCTRL)          += pinctrl/
obj-y                         += gpio/
obj-y                         += pwm/
obj-$(CONFIG_PCI)              += pci/
obj-$(CONFIG_PARISC)            += parisc/
obj-$(CONFIG_RAPIDIO)           += rapidio/
obj-y                         += video/
obj-y                         += idle/
```

把它剪切并粘贴到

```
* iommu/ comes before gpu as gpu are using iommu controllers
obj-$(CONFIG_IOMMU_SUPPORT)      += iommu/
  

* gpu/ comes after char for AGP vs DRM startup and after iommu
obj-y                           += gpu/
  

obj-$(CONFIG_CONNECTOR)          += connector/
  

obj-y                           += video/
```

Step5:回到 kernel 目录下, 执行 make menuconfig, 配置 kernel。

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# make menuconfig
scripts/kconfig/mconf Kconfig
```

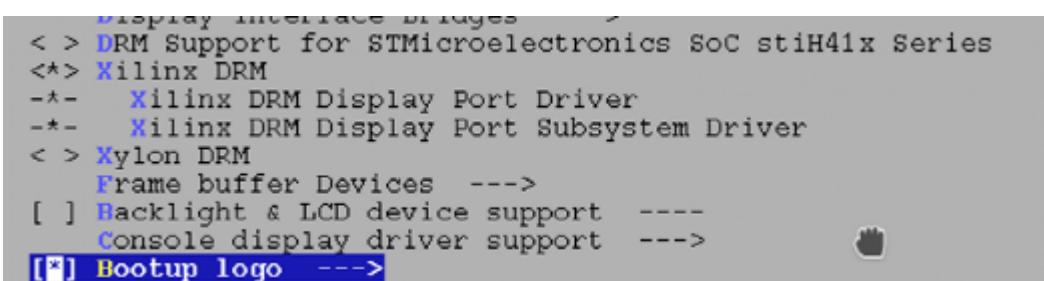
按向下方向键找到 Device Drivers, 按回车键进入

```
-- Patch physical to virtual translations at runtime
  General setup --->
  [*] Enable loadable module support --->
  [*] Enable the block layer --->
    System Type --->
    Bus support --->
    Kernel Features --->
    Boot options --->
    CPU Power Management --->
    Floating point emulation --->
    Userspace binary formats --->
    Power management options --->
  [*] Networking support --->
    Device Drivers --->
      Firmware Drivers --->
      File systems --->
      Kernel hacking --->
      Security options --->
  -- Cryptographic API --->
    Library routines --->
  -- Virtualization --->
```

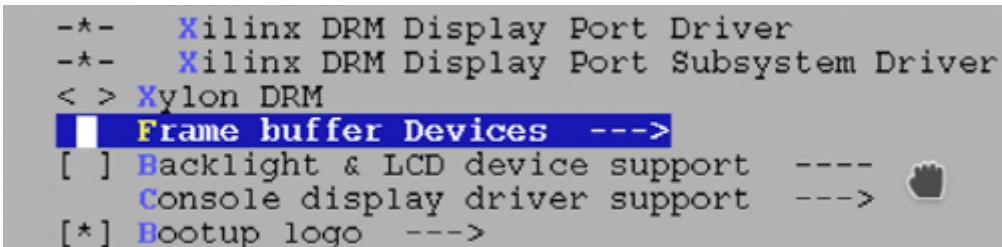
按向下方向键找到 Graphics support, 按回车键进入

```
-- Board level reset or power off --->
  [ ] Adaptive Voltage Scaling class support ----
  <*> Hardware Monitoring support --->
  <*> Generic Thermal sysfs driver --->
  [*] Watchdog Timer Support --->
    Sonics Silicon Backplane --->
    Broadcom specific AMBA --->
    Multifunction device drivers --->
  [*] Voltage and Current Regulator Support --->
  <*> Multimedia support --->
  Graphics support --->
    Sound card support --->
    HID support --->
  [*] USB support --->
  < > Ultra Wideband devices ----
```

按向下方向键找到 Bootup logo, 按空格键选择



按向上方向键找到 Frame buffer Devices，按回车键选择



至此已经完成 linux kernel 的配置，按向左方向键至<Exit>，再按回车键返回上一级菜单



直到看到以下窗口，按<Yes>保存。



1.5.4 执行 mk_kernel.sh 编译内核

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_kernel.sh
Build kernel and drivers on the /mnt/workspace/linux/kernel
make: Entering directory '/mnt/workspace/linux/kernel'
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK  include/config/kernel.release
  CHK  include/generated/uapi/linux/version.h
  CHK  include/generated/utsrelease.h
make[1]: 'include/generated/mach-types.h' is up to date.
  CHK  include/generated/timeconst.h
  CHK  include/generated/bounds.h
  CHK  include/generated/asm-offsets.h
  CALL scripts/checksyscalls.sh
  CHK  include/generated/compile.h
  GZIP kernel/config_data.gz
  CHK  kernel/config_data.h
  UPD  kernel/config_data.h
```

1.5.5 执行 mk_bootloader.sh 编译 uboot

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_bootloader.sh
Build U-Boot on the /mnt/workspace/linux/bootloader
make: Entering directory '/mnt/workspace/linux/bootloader'
  CHK      include/config/uboot.release
  CHK      include/generated/timestamp autogenerated.h
  UPD      include/generated/timestamp autogenerated.h
  CHK      include/generated/version autogenerated.h
  CHK      include/generated/asm-offsets.h
  CHK      include/generated/generic-asm-offsets.h
 HOSTCC  tools/mkenvimage.o
 HOSTCC  tools/image-host.o
 HOSTCC  tools/fit_image.o
 HOSTCC  tools/dumpimage.o
 HOSTCC  tools/mkimage.o
 HOSTLD  tools/mkenvimage
 HOSTLD  tools/fit_info
```

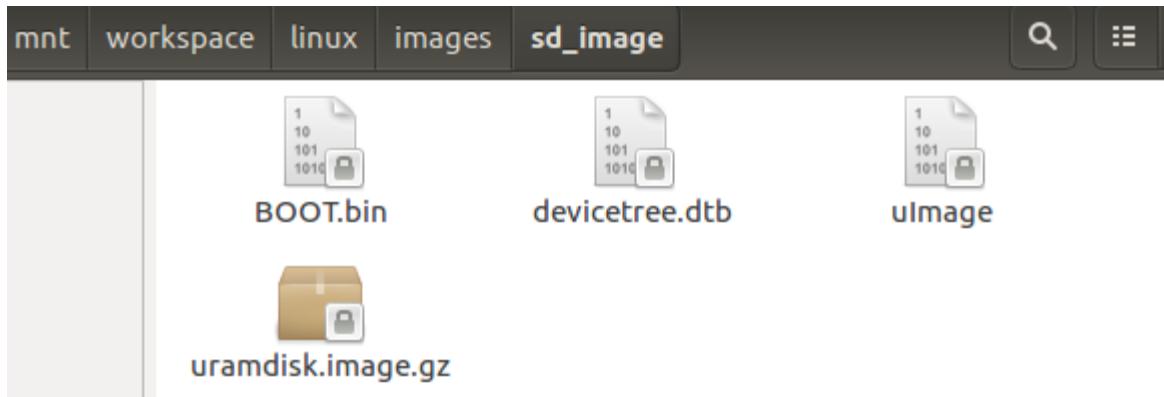
1.5.6 制作 UBOOT.BIN

Step1: 把 system_wrapper.bit 和 fsbl.elf 复制到 output/target/ 目录下。

Step2: 执行 mk_sd_image.sh 打包从 SD 启动所需要的文件

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_sd_image.sh
Remove older files...
```

Step3: 至此系统部分已经完成, 复制 linux/image/sd_images 文件夹下文件到 TF 卡测试移植好的 LINUX 系统。



1.6 EMMC 8GB 内存测试(MZ701Amini 不支持)

Step1: 从系统的启动信息可以看到, 系统已经发现 mmc0 (即 SD 卡) 和 mmc1 (即 emmc), 而且列出了 SD 卡为 7.41GB, 而 emmc 为 7.20GB

```

sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhci-ptfm: SDHCI platform and OF driver helper
mmc0: SDHCI controller on e0100000.sdhci [e0100000.sdhci] using DMA
mmc1: SDHCI controller on e0101000.sdhci [e0101000.sdhci] using DMA
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
mmc0: new high speed SDHC card at address 59b4
NET: Registered protocol family 10
mmcblk0: mmc0:59b4 USDQ0 7.41 GiB
sit: IPv6 over IPv4 tunneling driver
mmcblk0: p1
NET: Registered protocol family 17
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
Registering SWP/SWPB emulation handler
hctosys: unable to open rtc device (rtc0)
ALSA device list:
    No soundcards found.
RAMDISK: gzip image found at block 0
mmc1: new high speed MMC card at address 0001
mmcblk1: mmc1:0001 P1XXXX 7.20 GiB
mmcblk1boot0: mmc1:0001 P1XXXX partition 1 2.00 MiB
mmcblk1boot1: mmc1:0001 P1XXXX partition 2 2.00 MiB
mmcblk1rpmb: mmc1:0001 P1XXXX partition 3 128 KiB

```

Step2:给 eMMC 分区 当然，如果只分为一个分区的话，其它也可以不分区。在命令行下运行 fdisk /dev/mmcblk1 来对 emmc 进行分区。这里需要注意，确认有没有 SD 卡插入，也就是说确认当前 eMMC 是/dev/mmcblk1 还是 /dev/mmcblk0，还有对于有多个分区的，可能存在 /dev/mmcblk0p1、/dev/mmcblk0p2 等等。

```

zynq> fdisk /dev/mmcblk1
The number of cylinders for this disk is set to 236032.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): m
Command Action
a   toggle a bootable flag
b   edit bsd disklabel
c   toggle the dos compatibility flag
d   delete a partition
l   list known partition types
n   add a new partition
o   create a new empty DOS partition table
p   print the partition table
q   quit without saving changes
s   create a new empty Sun disklabel
t   change a partition's system id
u   change display/entry units
v   verify the partition table
w   write table to disk and exit
x   extra functionality (experts only)

Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-236032, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-236032, default 236032): Using default value 236032

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table
mmcblk1: p1
zynq>

```

Step3: 将分区格式化为 ext2 格式 能格式化为什么格式, 如 ext2、ext3、nfts、fat32, 这些都跟系统的定制有关

```
zynq> mkfs.ext2 /dev/mmcblk1p1
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
472352 inodes, 1888254 blocks
94412 blocks (5%) reserved for the super user
First data block=0
Maximum filesystem blocks=4194304
58 block groups
32768 blocks per group, 32768 fragments per group
3144 inodes per group
superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632
random: nonblocking pool is initialized
zynq> ls /mnt/
zynq> mount /dev/mmcblk1p1 /mnt/
EXT4-fs (mmcblk1p1): couldn't mount as ext3 due to feature incompatibilities
EXT4-fs (mmcblk1p1): filesystem too large to mount safely on this system
zynq> ls /mnt/
lost+found
zynq>
```

Step4: 测试 emmc 的性能

在 Linux 下, 既可以使用 dd 命令来低格 U 盘等, 也可以用来复制文件 (类似于 windows 下的 ghost 功能), 当然也可以用来测试硬盘等的性能, 虽然没有专业软件的测试得准确, 但用来对比性能已经足够了。对于/dev/zero 和/dev/null 两个设备的说明, 可以百度一下

Step4.1 写性能

下面使用 dd 命令将从/dev/zero 设备中产生一个 1GB 的文件写入到 emmc:

```
zynq> dd if=/dev/zero of=/mnt/test.img bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.0GB) copied, 75.173231 seconds, 13.6MB/s
```

Step4.2 读性能

下面使用 dd 命令将 1GB 的文件写入到/dev/null 设备中:

```
zynq> dd if=/mnt/test.img of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.0GB) copied, 52.177487 seconds, 19.6MB/s
```

Step4.3 读写性能

下面使用 dd 命令将 emmc 中的一个 1GB 的文件写入到 emmc 的另外一个文件

```
zynq> dd if=/mnt/test.img of=/mnt/test.copy.img
2097152+0 records in
2097152+0 records out
1073741824 bytes (1.0GB) copied, 130.697851 seconds, 7.8MB/s
```

1.7 测试 framebuffer

Step1: 将虚拟机/mnt/workspace/linux/framebuffer 目录下的测试程序(源代码也

在该目录里) 复制到 SD 卡上, 然后给开发板上电。在 Windows 下打开串口终端 putty 软件。

Step2: 切换到 sd 卡目录下(比如 mount /dev/mmcblk0p1 /mnt, cd /mnt), 然后运行 ./framebuffer, 在串口终端 会显示 以下信息。

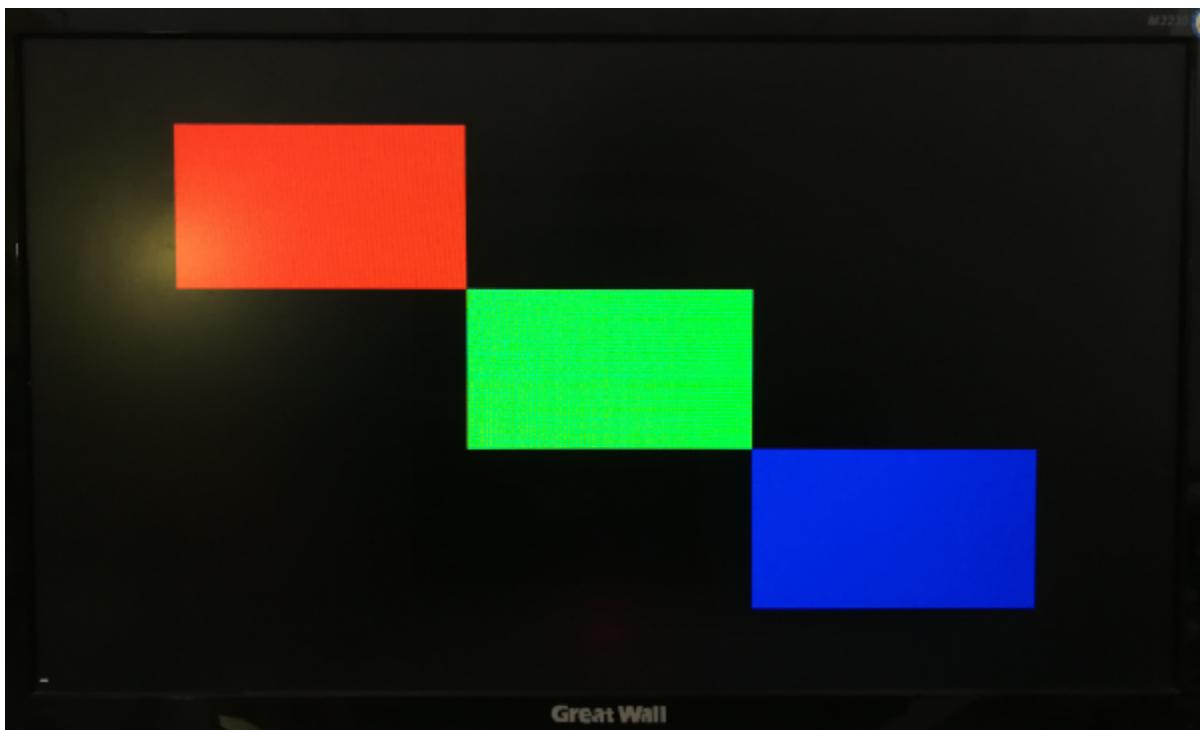
```
zynq> ./framebuffer
Fixed screen info:
  id: vdma-fb
  smem_start: 0x1f100000
  smem_len: 1228800
  type: 0
  type_aux: 0
  visual: 2
  xpanstep: 0
  ypanstep: 0
  ywrapstep: 0
  line_length: 2560
  mmio_start: 0x0
  mmio_len: 0
  accel: 0

Variable screen info:
  xres: 640
  yres: 480
  xres_virtual: 640
  yres_virtual: 480
  yoffset: 0
  xoffset: 0
  bits_per_pixel: 32
  grayscale: 0
  red: offset: 16, length: 8, msb_right: 0
  green: offset: 8, length: 8, msb_right: 0
  blue: offset: 0, length: 8, msb_right: 0
  transp: offset: 24, length: 8, msb_right: 0
  nonstd: 0
  activate: 0
  height: 0
  width: 0
  accel_flags: 0x0
  pixclock: 39721
  left_margin: 0
  right_margin: 0
  upper_margin: 0
  lower_margin: 0
  hsync_len: 0
  vsync_len: 0
  sync: 0
  vmode: 0

Frame Buffer Performance test...
  Average: 2186 usecs
  Bandwidth: 536.082 MByte/Sec
  Max. FPS: 457.457 fps

Will draw 3 rectangles on the screen,
they should be colored red, green and blue (in that order).
  Done.
```

Step3: 在屏幕上会显示以下信息



1.8 小结

如果编译不过，或者编译后程序还有问题，可能是残余编译文件导致，需要在 bootloader\kernel 路径下执行 make distclean 命令之后重新执行编译操作，一般就可以解决问题。

S04_CH02_工程移植 ubuntu 并一键制作启动盘

2.1 概述

2.2 搭建硬件系统

本章硬件工程还是使用《S04_CH01_搭建工程移植 LINUX/测试 EMMC/VGA》所搭建的VIVADO 工程。由于使用批处理命令，本章操作起来十分简单，但是批处理命令的源码大家可以好好分析。笔者会在后期的课程中专门把所有用到的常用批处理命令总结分析一下。

2.3 一键制作

Step1:输入 su 切换到 root 用户

```
osrc@osrc-virtual-machine:~$ sudo su
[sudo] password for osrc:
root@osrc-virtual-machine:/home/osrc#
```

Step2:输入 cd /mnt/workspace/linux/scripts 切换到/mnt/workspace/linux/scripts 目录下，执行 source setup_env.sh（注意：source 和 “.” 是相同的效 果）初始化环境变量

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# source setup_env.sh
```

Step3:执行 mk_kernel.sh 编译内核

Step4:执行 mk_sd_image.sh 生成基于 ramdisk 文件系统的 SD 镜像，当然，这里主要使用 linaro 文件系统，只会使用 到其中的部分文件而已

Step5:插入 SD

Step6:执行 install_linaro_image.sh 即可删除 SD 卡的所有分区，然后重新分区、格式化，并烧录 linaro 系统，执行完毕 后，拔下 SD 卡至 开发板上，设置启动模式为从 SD 启动即可。

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# install_linaro_image.sh
硬盘分区信息
Disk /dev/sda: 20 GiB, 21474836480 bytes, 41943040 sectors
Disk /dev/sdb: 40 GiB, 42949672960 bytes, 83886080 sectors
请输入需要分区的磁盘{hdX|sdX}或者输入q退出此脚本: sdc
即将格式化并分区磁盘sdc，请输入(y/n)确认?y
df: /mnt/hgfs: Protocol error
1+0 records in
1+0 records out
512 bytes copied, 0.00555828 s, 92.1 kB/s
已经清除磁盘sdc上的所有分区
磁盘sdc分区成功
df: /mnt/hgfs: Protocol error
正在格式化启动分区...
正在格式化根文件系统...
完成磁盘sdc的分区操作
挂载/dev/sdc1至/mnt/workspace/linux/output/linaro/boot目录下
挂载/dev/sdc2至/mnt/workspace/linux/output/linaro/rootfs目录下
正在制作boot...
正在制作文件系统...
已经烧录Linaro系统至SD卡...
root@osrc-virtual-machine:/mnt/workspace/linux/scripts#
```

2.4 运行结果

S04_CH03_QSPI 烧写 LINUX 系统

3.1 概述

3.2 搭建硬件系统

本章硬件工程还是使用《S04_CH01_搭建工程移植 LINUX/测试 EMMC/VGA》所搭建的 VIVADO 工程。

3.3 修改内核文件

Step1:切换到管理员模式

```
osrc@osrc-virtual-machine:~$ sudo su  
[sudo] password for osrc:  
root@osrc-virtual-machine:/home/osrc#
```

Step2:切换到 scripts 目录下，执行 source setup_env.sh（注意 source 和 “.” 是一致的），并将 scripts.tar.gz 中的两个脚本放到 scripts 目录下，通过这两个脚本可以打包 QSPI 镜像和将 QSPI 镜像烧录至 QSPI 中。本项目是基于上一个项目工程。

```
osrc@osrc-virtual-machine:~$ sudo su  
[sudo] password for osrc:  
root@osrc-virtual-machine:/home/osrc# cd /mnt/workspace/linux/scripts  
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# . setup_env.sh  
root@osrc-virtual-machine:/mnt/workspace/linux/scripts#
```

Step4:切换到 bootloader 源码目录，打开 include/configs/zynq-common.h 文件

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# cd ..  
root@osrc-virtual-machine:/mnt/workspace/linux# cd bootloader  
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader# cd include/configs  
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader/include/configs# vi zynq-common.h  
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader/include/configs#
```

Step5:修改以下内核、设备树及内存文件系统在 QSPI 的起始位置如 Step6 所示：

```

/* Default environment */
#ifndef CONFIG_EXTRA_ENV_SETTINGS
#define CONFIG_EXTRA_ENV_SETTINGS \
    "ethaddr=00:0a:35:00:01:22\0" \
    "kernel_image=uImage\0" \
    "kernel_load_address=0x2080000\0" \
    "ramdisk_image=uramdisk.image.gz\0" \
    "ramdisk_load_address=0x4000000\0" \
    "devicetree_image=devicetree.dtb\0" \
    "devicetree_load_address=0x2000000\0" \
    "bitstream_image=system.bit.bin\0" \
    "boot_image=800T.bin\0" \
    "loadbit_addr=0x100000\0" \
    "loadbootenv_addr=0x2000000\0" \
    "kernel_size@0x500000\0" \
    "devicetree_size=0x20000\0" \
    "ramdisk_size=0x5E0000\0" \
    "boot_size=0xF00000\0" \
    "fdt_high=0x20000000\0" \
    "initrd_high=0x20000000\0" \
    "bootenv=uEnv.txt\0" \
    "loadbootenv=load mmc 0 ${loadbootenv_addr} ${bootenv}\0" \
    "importbootenv=echo Importing environment from SD ...; " \
        "env import -t ${loadbootenv_addr} ${filesize}\0" \
    "sd_uEnvtxt_existence_test=test -e mmc 0 /uEnv.txt\0" \
    "preboot;if test $modeboot = sdboot && env run sd_uEnvtxt_existence_test; " \
        "then if env run loadbootenv; " \
            "then env run importbootenv; " \
                "fi; " \
            "fi; \0" \
    "mmc_loadbit#echo Loading bitstream from SD/MMC/eHMC to RAM.. && " \
        "mmcinfo && " \
        "load mmc 0 ${loadbit_addr} ${bitstream_image} && " \
        "fpga load 0 ${loadbit_addr} ${filesize}\0" \
    "norboot#echo Copying Linux from NOR flash to RAM... && " \
        "cp.b 0xE2100000 ${kernel_load_address} ${kernel_size} && " \
        "cp.b 0xE2600000 ${devicetree_load_address} ${devicetree_size} && " \
        "echo Copying ramdisk... && " \
        "cp.b 0xE2620000 ${ramdisk_load_address} ${ramdisk_size} && " \
        "bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}\0" \
    "qspiboot#echo Copying Linux from QSPI flash to RAM... && " \
        "sf probe 0 0 0 && " \
        "sf read ${kernel_load_address} 0x100000 ${kernel_size} && " \
        "sf read ${devicetree_load_address} 0x600000 ${devicetree_size} && " \
        "echo Copying ramdisk... && " \
        "sf read ${ramdisk_load_address} 0x620000 ${ramdisk_size} && " \
        "boot ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}\0" \
    "uenvboot;" \
        "if run loadbootenv; then " \

```

Step6:QSPI 的起始位置放置 FSBL、FPGA 比特流和 uboot，文件大小大概是 5MB，所以内核的存放位置从 0x500000 开始，而这里同样给内核留 5MB，当然，如果你的内核增加更多配置或减少配置，可以适当修改，这样的话，设备树就需要从 0xA00000 位置开始存放，然后留给设备树的空间大概是 2K，即从 0xA20000 开始存储文件系统，这样 ramdisk 的大小就应该是 0x15E0000，当然，内核默认支持 8MB

```

/* Default environment */
#ifndef CONFIG_EXTRA_ENV_SETTINGS
#define CONFIG_EXTRA_ENV_SETTINGS
    "ethaddr=00:0a:35:00:01:22\0" \
    "kernel_image=uImage\0" \
    "kernel_load_address=0x2080000\0" \
    "ramdisk_image=uramdisk.image.gz\0" \
    "ramdisk_load_address=0x4000000\0" \
    "devicetree_image=devicetree.dtb\0" \
    "devicetree_load_address=0x2000000\0" \
    "bitstream_image=system.bit.bin\0" \
    "boot_image=BOOT.bin\0" \
    "loadbit_addr=0x100000\0" \
    "loadbootenv_addr=0x2000000\0" \
    "kernel_size=0x500000\0" \
    "devicetree_size=0x20000\0" \
    "ramdisk_size=0x15E0000\0" \
    "boot_size=0xF00000\0" \
    "fdt_high=0x20000000\0" \
    "initrd_high=0x20000000\0" \
    "bootenv=uEnv.txt\0" \
    "loadbootenv=load mmc 0 ${loadbootenv_addr} ${bootenv}\0" \
    "importbootenv=echo Importing environment from SD ...;" \
        "env import -t ${loadbootenv_addr} ${filesize}\0" \
    "sd_uEnvtxt_existence_test=test -e mmc 0 /uEnv.txt\0" \
    "preboot=if test $modeboot = sdboot && env run sd_uEnvtxt_existence_test;" \
        "then if env run loadbootenv;" \
            "then env run importbootenv;" \
                "fi;" \
        "fi; \0" \
    "mmc_loadbit=echo Loading bitstream from SD/MMC/eMMC to RAM.. && " \
        "mmcinfo && " \
        "load mmc 0 ${loadbit_addr} ${bitstream_image} && " \
        "fpga load 0 ${loadbit_addr} ${filesize}\0" \
    "norboot=echo Copying Linux from NOR Flash to RAM... && " \
        "cp.b 0xE2100000 ${kernel_load_address} ${kernel_size} && \0" \
        "cp.b 0xE2600000 ${devicetree_load_address} ${devicetree_size} && \0" \
        "echo Copying ramdisk... && " \
        "cp.b 0xE2620000 ${ramdisk_load_address} ${ramdisk_size} && \0" \
        "bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}\0" \
    "qspiboot=echo Copying Linux from QSPI flash to RAM... && " \
        "sf probe 0 0 0 && " \
        "sf read ${kernel_load_address} 0x500000 ${kernel_size} && " \
        "sf read ${devicetree_load_address} 0xA00000 ${devicetree_size} && " \
        "echo Copying ramdisk... && " \
        "sf read ${ramdisk_load_address} 0xA20000 ${ramdisk_size} && " \
        "bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}\0" \
    "uenvboot=" \
        "if run loadbootenv; then " \

```

Step7:打开 zynq-zed.dts, 找到 qspi 节点, 这里把原有的分区删掉, 当然, 你也可以根据刚才对 QSPI 的分区, 做对应的修改。可以出从 SD 启动或从 QSPI 启动 Linux, 然后在系统里更新 QSPI 镜像。

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# vt arch/arm/boot/dts/zynq-zed.dts
```

```

@qspi {
    status = "okay";
    is-dual = <0>;
    num-cs = <1>;
    flash@0 {
        compatible = "n25qi28a11";
        reg = <0x0>;
        spi-tx-bus-width = <1>;
        spi-rx-bus-width = <4>;
        spi-max-frequency = <500000000>;
        #address-cells = <1>;
        #size-cells = <1>;
        partition@qspi-fsbl-uboot {
            label = "qspi-fsbl-uboot";
            reg = <0x0 0x100000>;
        };
        partition@qspi-linux {
            label = "qspi-linux";
            reg = <0x100000 0x5000000>;
        };
        partition@qspi-device-tree {
            label = "qspi-device-tree";
            reg = <0x600000 0x200000>;
        };
        partition@qspi-rootfs {
            label = "qspi-rootfs";
            reg = <0x620000 0x5E0000>;
        };
        partition@qspi-bitstream {
            label = "qspi-bitstream";
            reg = <0xC00000 0x400000>;
        };
    };
};

@qspi [
    status = "okay";
    is-dual = <0>;
    num-cs = <1>;
    flash@0 {
        compatible = "n25qi28a11";
        reg = <0x0>;
        spi-tx-bus-width = <1>;
        spi-rx-bus-width = <4>;
        spi-max-frequency = <500000000>;
        #address-cells = <1>;
        #size-cells = <1>;
        /* */
        partition@qspi-fsbl-uboot {
            label = "qspi-fsbl-uboot";
            reg = <0x0 0x100000>;
        };
        partition@qspi-linux {
            label = "qspi-linux";
            reg = <0x100000 0x5000000>;
        };
        partition@qspi-device-tree {
            label = "qspi-device-tree";
            reg = <0x600000 0x200000>;
        };
        partition@qspi-rootfs {
            label = "qspi-rootfs";
            reg = <0x620000 0x5E0000>;
        };
        partition@qspi-bitstream {
            label = "qspi-bitstream";
            reg = <0xC00000 0x400000>;
        };
    };
];

```

3.3 编译内核及 u-boot

Step1: 执行 mk_bootloader.sh 编译 bootloader 源码

```

root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_bootloader.sh
Build U-Boot on the /mnt/workspace/linux/bootloader
make: Entering directory '/mnt/workspace/linux/bootloader'
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK      include/config.h
  UPD      include/config.h
  GEN      include/autoconf.mk.dep
  GEN      include/autoconf.mk
  GEN      scripts/include/autoconf.mk

```

Step2: 执行 mk_kernel.sh 编译 kernel 源码

```

root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_kernel.sh
Build kernel and drivers on the /mnt/workspace/linux/kernel
make: Entering directory '/mnt/workspace/linux/kernel'
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK      include/config/kernel.release
  CHK      include/generated/uapi/linux/version.h
  CHK      include/generated/utsrelease.h
  HOSTCC  scripts/genksyms/lex.lex.o
  HOSTCC  scripts/dtc/dtc.o

```

3.4 制作 qspi 镜像

Step3: 制作 qspi 镜像，制作完毕后，可以在 images/qspi_image 目录下看到 qspi_image.bin 镜像

```

root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_qspi_image.sh
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# 

```

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# cd /mnt/workspace/linux/images/qspi_image/
root@osrc-virtual-machine:/mnt/workspace/linux/images/qspi_image# ls
qspi_image.bin
root@osrc-virtual-machine:/mnt/workspace/linux/images/qspi_image#
```

3.5 安装 screen

Step1:安装 screen, 可以直接在 Ubuntu 系统下查看串口调试信息, 当然, 你也可以在 Windows 下使用 putty 之类的

```
root@osrc-virtual-machine:~# sudo apt-get install screen
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  linux-headers-4.4.0-59 linux-headers-4.4.0-59-generic linux-headers-4.4.0-62
  linux-headers-4.4.0-62-generic linux-image-4.4.0-59-generic linux-image-4.4.0-62-generic
  linux-image-extra-4.4.0-59-generic linux-image-extra-4.4.0-62-generic
Use 'sudo apt autoremove' to remove them.
Suggested packages:
  lselect | screen | byobu ncurses-term
The following NEW packages will be installed:
  screen
0 upgraded, 1 newly installed, 0 to remove and 334 not upgraded.
Need to get 560 kB of archives.
After this operation, 972 kB of additional disk space will be used.
Get:1 http://cn.archive.ubuntu.com/ubuntu xenial/main amd64 screen amd64 4.3.1-2build1 [560 kB]
Fetched 560 kB in 0s (918 kB/s)
Selecting previously unselected package screen.
(Reading database ... 274503 files and directories currently installed.)
Preparing to unpack .../screen_4.3.1-2build1_amd64.deb ...
Unpacking screen (4.3.1-2build1) ...
Processing triggers for systemd (229-4ubuntu10) ...
Processing triggers for ureadahead (0.100.0-19) ...
Processing triggers for install-info (6.1.0.dfsg.1-5) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up screen (4.3.1-2build1) ...
Processing triggers for systemd (229-4ubuntu10) ...
Processing triggers for ureadahead (0.100.0-19) ...
root@osrc-virtual-machine:~#
```

Step2:输入 /dev/ttUSB0 正是串口

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# ls /dev/
agpgart      loop0      ram2      tty10     tty4      tty51    uhid
autofs       loop1      ram3      tty11     tty40     tty510   uinput
block        loop2      ram4      tty12     tty41     tty511   urandom
bsg         loop3      ram5      tty13     tty42     tty512   userio
btrfs-control loop4      ram6      tty14     tty43     tty513   vcs
bus          loop5      ram7      tty15     tty44     tty514   vcs1
cdrom        loop6      ram8      tty16     tty45     tty515   vcs2
cdrw        loop7      ram9      tty17     tty46     tty516   vcs3
char        loop-control random   tty18     tty47     tty517   vcs4
console      mapper    rfkill   tty19     tty48     tty518   vcs5
core         mcelog    rtc      tty2      tty49     tty519   vcs6
cpu          mem       rtc0    tty20     tty5      tty52    vcs7
cpu_dma_latency memory_bandwidth sda      tty21     tty50     tty520   vcsa
cuse         midi      sda1    tty22     tty51     tty521   vcsa1
disk         nqueue   sda2      tty23     tty52     tty522   vcsa2
dm-midi      net       sda5    tty24     tty53     tty523   vcsa3
dri         network_latency sdb      tty25     tty54     tty524   vcsa4
dvd         network_throughput sdb1    tty26     tty55     tty525   vcsa5
ecryptfs     null      serial   tty27     tty56     tty526   vcsa6
fb0          port      sg0      tty28     tty57     tty527   vcsa7
fd           ppp       sg1      tty29     tty58     tty528   vfio
full         psaux    sg2      tty3      tty59     tty529   vga_arbiter
fuse         ptmx      sg3      tty30     tty6      tty53    vhci
hidraw0     pts       snapshot  tty31     tty60     tty530   vhost-net
hpet         ram0      snd      tty32     tty61     tty531   vmcl
hugepages    ram1      sr0      tty33     tty62     tty54    vsock
hw RNG      ram10     stderr   tty34     tty63     tty55    zero
initctl     ram11     stdin    tty35     tty7      tty56
input        ram12     stdout   tty36     tty8      tty57
kmsg         ram13     tty      tty37     tty9      tty58
lightnvm    ram14     tty0     tty38     ttyprintk  tty59
log          ram15     tty1     tty39     tty50     ttvUSB0
```

3.6 一件烧写 QSPI FLASH 1

接下 JTAG 并打开开发板电源，执行 `program_qspi_flash.sh` 将 QSPI 镜像烧录到 QSPI 芯片上，或许会出现以下错误，没关系，此时因为驱动还没有加载，重新执行一次。

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# ./program_qspi_flash.sh

***** Xilinx Program Flash
***** Program Flash v2015.4 (64-bit)
**** SW Build 1412921 on Wed Nov 18 09:44:32 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Connecting to hw_server @ TCP:127.0.0.1:3121

WARNING: Failed to connect to hw_server at TCP:127.0.0.1:3121
Attempting to launch hw_server at TCP:127.0.0.1:3121

Connected to hw_server @ TCP:127.0.0.1:3121
Available targets and devices:
Target 0 : jsn-JTAG-HS1-210249855195
Device 0: jsn-JTAG-HS1-210249855195-4ba00477-0

JTAG chain configuration
-----
Device ID Code      IR Length Part Name
1     4ba00477       4        arm_dap
2     13722093        6        xc7z010

-----
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".
```

```
CortexA9 Processor Configuration
-----
Version..... 0x00000003
User ID..... 0x00000000
No of PC Breakpoints..... 6
No of Addr/Data Watchpoints..... 4
Waiting for Bootrom to re-enable DAP after reset
Processor Reset .... DONE
SF: Detected S25FL256S_64K with page size 256 Bytes, erase size 64 KiB, total 32 MiB
Performing Erase Operation...
Erase Operation successful.
INFO: [Xicom 50-44] Elapsed time = 29 sec.
Performing Program Operation...
0%.....10%.
.....20%.
.....30%.
.....40%.
.....50%.
.....60%.
.....70%.
.....80%.
.....90%.
.....100%.
.....Program Operation successful.
INFO: [Xicom 50-44] Elapsed time = 234 sec.

Flash Operation successful
root@osrc-virtual-machine:/mnt/workspace/linux/kernel#
```

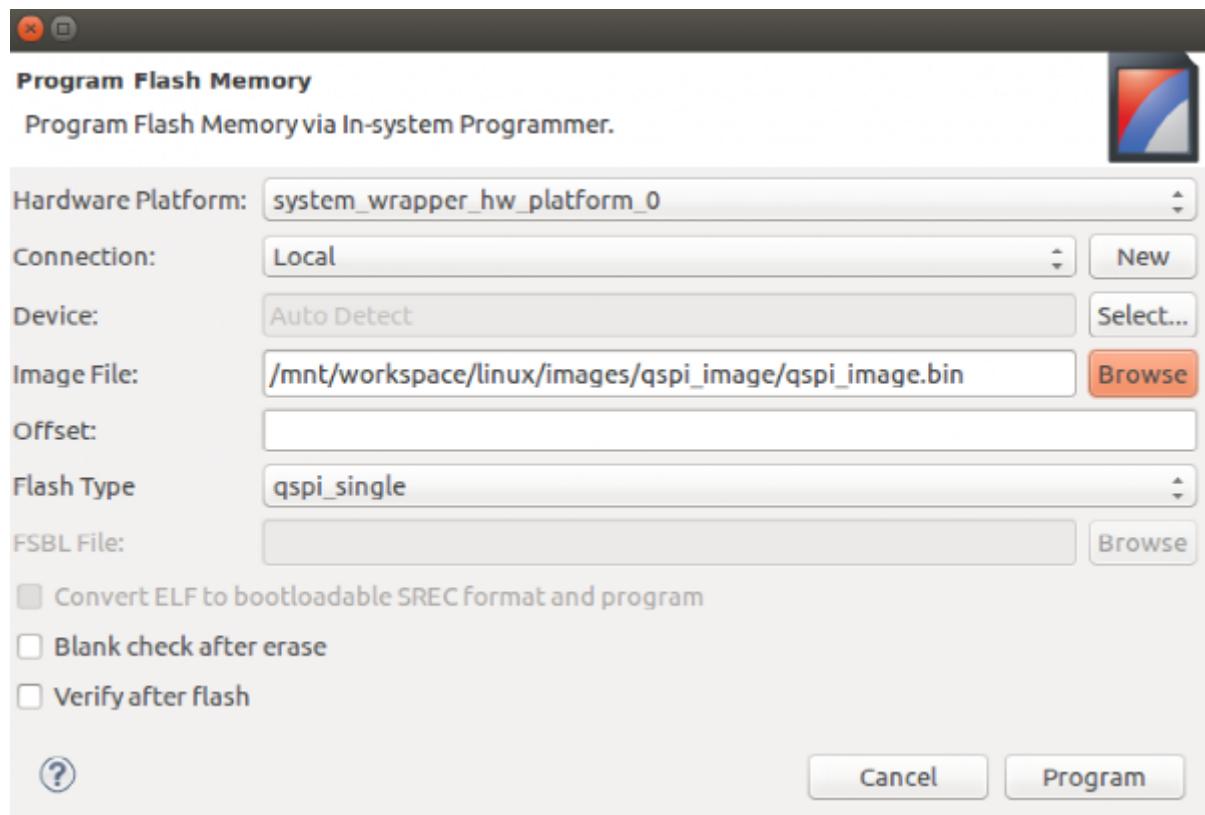
3.7 烧写 QSPI FLASH 2

如果确实不行，那么打开 SDK 来烧录吧

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# cd /mnt/workspace/miz701n  
root@osrc-virtual-machine:/mnt/workspace/miz701n# cd Miz_sys  
root@osrc-virtual-machine:/mnt/workspace/miz701n/Miz_sys# vivado Miz sys.xpr
```

如果是切换到 su 的话，可能会提示没有找到 license，那么把/home/osrc/.Xilinx 目录复制到/root/目录下

```
root@osrc-virtual-machine:/home/osrc# cp .Xilinx/ /root/ -r  
root@osrc-virtual-machine:/home/osrc#
```



烧录完毕后，重启开发板，当然，请先确保调整启动模式为从 QSPI 启动，在 Ubuntu 下打开串口终端

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# screen /dev/ttyUSB0 115200
```

即可以看到以下信息

```
[PERF] transceiver vendor:, product id 0x0131, 0x150.
Found TI TUSB1210 ULPI transceiver.
ULPI integrity check: passed.
ci_hdrc ci_hdrc.0: EHCI Host Controller
ci_hdrc ci_hdrc.0: new USB bus registered, assigned bus number 1
ci_hdrc ci_hdrc.0: USB 2.0 started, EHCI 1.00
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
mousedev: PS/2 mouse device common for all mice
i2c /dev entries driver
EDAC MC: ECC not enabled
Xilinx Zynq CpuIdle Driver started
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhci-pltfm: SDHCI platform and OF driver helper
mmc0: SDHCI controller on e0100000.sdhci [e0100000.sdhci] using DMA
mmc1: SDHCI controller on e0101000.sdhci [e0101000.sdhci] using DMA
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
NET: Registered protocol family 10
mmc0: new high speed SDHC card at address 0001
sit: IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
mmcblk0: mmc0:0001 00000 7.32 GiB
  mmcblk0: p1 p2
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
Registering SWP/SWPB emulation handler
hctosys: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
RAMDISK: gzip image found at block 0
usb 1-1: new full-speed USB device number 2 using ci_hdrc
mmc1: new high speed MMC card at address 0001
mmcblk1: mmc1:0001 P1XXXX 7.20 GiB
mmcblk1boot0: mmc1:0001 P1XXXX partition 1 2.00 MiB
mmcblk1boot1: mmc1:0001 P1XXXX partition 2 2.00 MiB
mmcblk1rpmb: mmc1:0001 P1XXXX partition 3 128 KiB
  mmcblk1: p1
EXT4-fs (ram0): couldn't mount as ext3 due to feature incompatibilities
EXT4-fs (ram0): mounted filesystem without journal. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 1024K (c0800000 - c0900000)
random: sshd urandom read with 5 bits of entropy available
zyng> █
```

在 HDMI 显示器上也可以看到相关信息。



S04_CH04_自动挂载 8GB EMMC 板载内存

4.1 概述

EMMC 内存是直接焊接在 PCB 板上的，因此不能像 TF 卡那样方便的通过插入 USB 接口进行分区格式化。在第一次使用的时候，首先要分区，然后格式化，最后挂载到系统。本课程承接上一课程，请读者学习的时候注意学习顺序。（MZ701Amini 没有 EMMC 所以跳过此课程）

4.2 执行 source setup_env.sh

不管如何第一条指令我们要执行 source setup_env.sh 设置环境变量

4.3 修改 zynq-7000.dtsi 文件

Step1:进入如下目录并且用 vim 打开 zynq-7000.dtsi 文件

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel/arch/arm/boot/dts# vim zynq-7000.dtsi
```

Step2:通过 vim 打开文件后入下图

```

root@osrc-virtual-machine:/mnt/workspace/linux/kernel/arch/arm/boot/dts
/*
 * Copyright (c) 2011 - 2014 Xilinx
 *
 * This software is licensed under the terms of the GNU General Public
 * License version 2, as published by the Free Software Foundation, and
 * may be copied, distributed, and modified under those terms.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 */
/include/ "skeleton.dtsi"

/ {
    compatible = "xlnx,zynq-7000";

    cpus {
        #address-cells = <1>;
        #size-cells = <0>

        cpu0: cpu@0 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <0>;
            clocks = <&clkc 3>;
            clock-latency = <1000>;
            cpu0-supply = <&regulator_vccpint>;
            operating-points = <
                /* kHz      uV */

```

Step3:输入/sdhci0 (vim 搜索命令)(在左下角可以看到输入的命令)按 回车

```

/ {
    compatible = "xlnx,zynq-7000";
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu0: cpu@0 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <0>;
            clocks = <&clkc 3>;
            clock-latency = <1000>;
            cpu0-supply = <8regulator_vccpint>;
            operating-points = <
                /* kHz      uV */
            /sdhci@e0100000

```

Step4:找到如下位置(完成 vim 搜索 可以看到 vim 搜索是效率非常高的)(修改前一定要备份下, 因为第七课开始需要继续用到未修改前的)

```

sdhci0: sdhci@e0100000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 21>, <&clkc 32>;
    interrupt-parent = <&intc>;
    interrupts = <0 24 4>;
    reg = <0xe0100000 0x1000>;
    broken-adma2;
};

sdhci1: sdhci@e0101000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 22>, <&clkc 33>;
    interrupt-parent = <&intc>;
    interrupts = <0 47 4>;
    reg = <0xe0101000 0x1000>;
    broken-adma2;
};

```

Step5:现在需要把 sdhci0 和 sdhci1 掉个顺序, 这里也是使用 vim 命令操作。输入 v 后可以看到左下角变成了大写的 VISUAL

```

sdhci0: sdhci@e0100000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 21>, <&clkc 32>;
    interrupt-parent = <&intc>;
    interrupts = <0 24 4>;
    reg = <0xe0100000 0x1000>;
    broken-adma2;
};

-- VISUAL --

```

Step6:按键盘 j(vim 命令 每按一次鼠标下移一行)选中 sdhci0 整个段落

```
sdhci0: sdhci@e0100000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 21>, <&clkc 32>;
    interrupt-parent = <&intc>;
    interrupts = <0 24 4>;
    reg = <0xe0100000 0x1000>;
    broken-adma2;
};
```

Step7:按键盘 d 实现剪切，然后，把光标移动到 sdhci0 段落最后一行再输入 p 就完成剪切了

```
sdhci1: sdhci@e0101000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 22>, <&clkc 33>;
    interrupt-parent = <&intc>;
    interrupts = <0 47 4>;
    reg = <0xe0101000 0x1000>;
    broken-adma2;
};

sdhci0: sdhci@e0100000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 21>, <&clkc 32>;
    interrupt-parent = <&intc>;
    interrupts = <0 24 4>;
    reg = <0xe0100000 0x1000>;
    broken-adma2;
};
```

Step8:输入:wq!保存并关闭 zynq-7000.dtsi 文件

4.4 设置 mount_emmc.sh 批处理命令的开机启动

Step1:首先看下已经编写好的 mount_emmc.sh 文件。可以看到执行 mount_emmc.sh 批处理命令，首先是在根目录下创建 emmc 文件夹路径。之后再把 ext2 格式的 EMMC 挂载到/mnt/emmc.。如果开机发现没有分区，就会分区后格式化，再挂载。

```
mkdir -p /mnt/emmc
mount -t ext2 /dev/mmcblk0p1 /mnt/emmc
ret=$?
```

```

echo $ret
if [ $ret -ne 0 ]; then
    echo -e "n \n p \n 1 \n \n \n w \n" | fdisk /dev/mmcblk0
    mkfs.ext2 /dev/mmcblk0p1
    mount -t ext2 /dev/mmcblk0p1 /mnt/emmc
fi

```

Step2:挂载 mount_emmc.sh 到根文件系统，并且拷贝 mount_emmc.sh 到 rootfs 路径

```

root@osrc-virtual-machine:/mnt/workspace/linux/output# mount_rootfs.sh
根文件系统已经挂载在/mnt/workspace/linux/output/rootfs。
root@osrc-virtual-machine:/mnt/workspace/linux/output# cd rootfs
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# cp ../../scripts/mount_emmc.sh
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs#

```

Step3:打开根文件系统下的 etc/init.d/rcS 文件

```

root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# vi etc/init.d/rcS

```

Step4:增加如下代码并且输入:wq!保存退出

```

if [ -f /mnt/init.sh ]
then
    echo "++ Running user script init.sh from SD Card"
    source /mnt/init.sh
fi

if [ -f /mount_emmc.sh ] . .
then
    echo "++Running user script mount_emmc.sh"
    source /mount_emmc.sh
fi

```

```

echo "rcS Complete"
~
```

可以看到红色框线内的代码就是在开机后可以执行 mount_emmc.sh 文件，当然 LINUX 下面开机自动运行的方式很多，这里不一一列举。

Step5:切换到 rootfs 路径之外的任何路径最简单地可以输入 cd 命令

```

root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# cd
root@osrc-virtual-machine:~#

```

Step6:执行 umount_rootfs.sh

```

root@osrc-virtual-machine:~# umount_rootfs.sh
Unmount the initrd image...
Compress the initrd image...
Wrapping the image with a U-Boot header...
Image Name:
Created:      Sun Apr  9 22:21:30 2017
Image Type:   ARM Linux RAMDisk Image (gzip compressed)
Data Size:    5307435 Bytes = 5183.04 kB = 5.06 MB
Load Address: 00000000
Entry Point:  00000000
root@osrc-virtual-machine:~#

```

4.5 烧写程序到 QSPI FLASH

Step1：硬件设置拨码开关到 ON ON ON OFF 这样为 QSPI flash 模式,连接下载器和串口。

Step2:分别执行 mk_qspi_image.sh 和 program_qspi_flash.sh 打包并烧录 QSPI 镜像

```
root@osrc-virtual-machine:~# mk_qspi_image.sh
root@osrc-virtual-machine:~# program_qspi_flash.sh

***** Xilinx Program Flash
***** Program Flash v2015.4 (64-bit)
**** SW Build 1412921 on Wed Nov 18 09:44:32 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Connecting to hw_server @ TCP:127.0.0.1:3121

WARNING: Failed to connect to hw_server at TCP:127.0.0.1:3121
Attempting to launch hw_server at TCP:127.0.0.1:3121

Connected to hw_server @ TCP:127.0.0.1:3121
Available targets and devices:
Target 0 : jsn-JTAG-HS1-210249855627
    Device 0: jsn-JTAG-HS1-210249855627-4ba00477-0

JTAG chain configuration
-----
Device   ID Code      IR Length  Part Name
1        4ba00477      4         arm_dap
2        23727093      6         xc7z020
-----
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".
```

4.6 验证测试

Step1:烧写完成后，输入 ls /dev 查看串口是否连接好，如下图红框所示设备

```
root@osrc-virtual-machine:~# ls /dev
apggart      hwrng      ppp      tty0  tty32  tty56      ttyS20  vcs2
autofs       initctl    psaux    tty1   tty33  tty57      ttyS21  vcs3
block        input      ptmx     tty10  tty34  tty58      ttyS22  vcs4
bsg          kmsg      pts      tty11  tty35  tty59      ttyS23  vcs5
btrfs-control lightnvm  random   tty12  tty36  tty6      ttyS24  vcs6
bus          log       rfkill   tty13  tty37  tty60      ttyS25  vcs7
cdrom        loop0     rtc      tty14  tty38  tty61      ttyS26  vcsa
cdrw         loop1     rtc0     tty15  tty39  tty62      ttyS27  vcsa1
char         loop2     sda      tty16  tty4   tty63      ttyS28  vcsa2
console      loop3     sda1     tty17  tty48  tty7      ttyS29  vcsa3
core         loop4     sda2     tty18  tty41  tty8      ttyS30  vcsa4
cpu          loop5     sda5     tty19  tty42  tty9      ttyS31  vcsa5
cpu_dma_latency loop6     sdb      tty2   tty43  ttysize  ttyS31  vcsa6
cuse         loop7     sdb1    tty20  tty44  tty50      ttyS4   vcsa7
disk         loop-control serial  tty21  tty45  tty51      ttyS5   vfio
dm-midi      mapper    sg0      tty22  tty46  tty510     ttyS6   vga_arbiter
dri          mclog     sg1      tty23  tty47  tty511     ttyS7   vhci
dvd          mem       sg2      tty24  tty48  tty512     ttyS8   vhost-net
ecryptfs     memory_bandwidth  snapshot  tty25  tty49  tty513     ttyS9   vmcl
fb0          midi      snd      tty26  tty5   tty514      ttyS10  vsock
fd           mqueue   sr0      tty27  tty50  tty515     ttyS11  uhid
full         net       stderr   tty28  tty51  tty516     ttyS12  zero
fuse         network_latency  stdin    tty29  tty52  tty517     ttyS13  uinput
hidraw0     network_throughput  stdout   tty3   tty53  tty518     ttyS14  urandom
hpet         null      stderr   tty30  tty54  tty519     ttyS15  userio
hugepages    port      tty     tty31  tty55  tty52      ttyS16  vcs
root@osrc-virtual-machine:~# s
```

Step2:用 screen 打开/dev/ttyUSB0

```
fb0          midi      snapshot  tty26  tty5   tty514      ttyS10  vsock
fd           mqueue   snd      tty27  tty50  tty515     ttyS11  uhid
full         net       sr0      tty28  tty51  tty516     ttyS12  zero
fuse         network_latency  stderr   tty29  tty52  tty517     ttyS13  uinput
hidraw0     network_throughput  stdin   tty3   tty53  tty518     ttyS14  urandom
hpet         null      stderr   tty30  tty54  tty519     ttyS15  userio
hugepages    port      tty     tty31  tty55  tty52      ttyS16  vcs
root@osrc-virtual-machine:~# screen /dev/ttyUSB0 115200
```

Step3:断电后重启开发板

Step4:用“ls /mnt/emmc/”可以查看到 emmc 已经挂载好了，然后我们往里面写入一个 hello.txt 文件

```
zynq> ls /mnt/emmc/
lost+found
zynq> vi /mnt/emmc/hello.txt
```

Step5:输入 hello emmc 几个单词(先按 i 再输入单词，输入完成后 :wq!保存并退出)

```
hello emmc
~
~
```

Step6:后卸载 emmc，再重新挂载 emmc，确认挂载的是否是 emmc 中还有 hello.txt 文件

```
zynq> ls /mnt/emmc/
hello.txt  lost+found
zynq> umount /mnt/emmc/
zynq> ls /mnt/emmc/
zynq> mount -t ext2 /dev/mmcblk0p1 /mnt/emmc/
zynq> ls /mnt/emmc/
hello.txt  lost+found
zynq>
```

Step7:打开 hello.txt 查看文件内容是否是我们之前输入的

```
zynq> vi /mnt/emmc/hello.txt
hello emmc
~
```

Step8：请读者断电重启，查看 hello.txt 是否还存在(结果是存在)。

```
devtmpfs: mounted
Freeing unused kernel memory: 1024K (c0800000 - c0900000)
EXT4-fs (mmcblk0p1): couldn't mount as ext3 due to feature incompatibilities
EXT4-fs (mmcblk0p1): filesystem too large to mount safely on this system
EXT2-fs (mmcblk0p1): warning: mounting unchecked fs, running e2fsck is recommended
random: sshd urandom read with 4 bits of entropy available
zynq> ls /mnt/emmc/
emmc      hello.txt  lost+found
```

4.7 思考为什么

因为你的 emmc 是挂在 SD1 上是吧，这样的话，当系统启动后，它扫描了设备树，就把 SD0 放在了第一个设备，SD1 放在了第二个设备，如果你没有插 SD 卡的话，那 emmc 就是第一个设备了，这样，你的脚本就比较难判断到底哪个才是 emmc

S04_CH05_在线升级 QSPI 镜像(U 盘方式)

5.1 概述

在线升级 QSPI 镜像，是指从 QSPI 模式或 SD 模式下启动 Linux，然后通过 U 盘、网络等下载 QSPI 镜像，然后在 Linux 下升级 QSPI 镜像，而不是通过 JTAG 的方式烧录 QSPI 镜像，有点类似于当前安卓手机的 OTA 升级方式。

5.2 执行 source setup_env.sh

不管如何第一条指令我们要执行 source setup_env.sh 设置环境变量

5.3 烧写程序到 QSPI FLASH

Step1：硬件设置拨码开关到 ON ON ON OFF 这样为 QSPI flash 模式,连接下载器和串口。

Step2:分别执行 mk_qspi_image.sh 和 program_qspi_flash.sh 打包并烧录 QSPI 镜像

```
root@osrc-virtual-machine:~# mk_qspi_image.sh
root@osrc-virtual-machine:~# program_qspi_flash.sh

***** Xilinx Program Flash
***** Program Flash v2015.4 (64-bit)
**** SW Build 1412921 on Wed Nov 18 09:44:32 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Connecting to hw_server @ TCP:127.0.0.1:3121

WARNING: Failed to connect to hw_server at TCP:127.0.0.1:3121
Attempting to launch hw_server at TCP:127.0.0.1:3121

Connected to hw_server @ TCP:127.0.0.1:3121
Available targets and devices:
Target 0 : jsn-JTAG-HS1-210249855627
    Device 0: jsn-JTAG-HS1-210249855627-4ba00477-0

JTAG chain configuration
-----
Device   ID Code          IR Length      Part Name
1        4ba00477           4            arm_dap
2        23727093           6            xc7z020

-----
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".
```

5.4 查看系统根目录

Step1:烧写完成后，输入 ls /dev 查看串口是否连接好，如下图红框所示设备

```
root@osrc-virtual-machine:~# ls /dev
agpgart      hwrng      ppp      tty0      tty32      tty56      tty520      vcs2
autofs       initctl    psaux    tty1      tty33      tty57      tty521      vcs3
block        input      ptmx     tty10     tty34      tty58      tty522      vcs4
bsg          kmsg      pts      tty11     tty35      tty59      tty523      vcs5
btrfs-control lightnvm  random   tty12     tty36      tty6      tty524      vcs6
bus          log       rfkill   tty13     tty37      tty60      tty525      vcs7
cdrom        loop0     rtc      tty14     tty38      tty61      tty526      vcsa
cdrw        loop1     rtc0     tty15     tty39      tty62      tty527      vcsa1
char        loop2     sda      tty16     tty4      tty63      tty528      vcsa2
console      loop3     sda1     tty17     tty40      tty7      tty529      vcsa3
core         loop4     sda2     tty18     tty41      tty8      tty53      vcsa4
cpu          loop5     sda5     tty19     tty42      tty9      tty530      vcsa5
cpu_dma_latency loop6     sdb      tty2      tty43      ttyprintk  tty531      vcsa6
cuse         loop7     sdb1     tty20     tty44      tty50      tty54      vcsa7
disk         loop-control serial   tty21     tty45      tty51      tty55      vfio
dmidi        mapper    sg0      tty22     tty46      tty510     tty56      vga_arbiter
dri          mcelog    sg1      tty23     tty47      tty511     tty57      vhci
dvd          mem       sg2      tty24     tty48      tty512     tty58      vhost-net
ecryptfs     memory_bandwidth  sgm      tty25     tty49      tty513     tty59      vmci
fb0          midi      snapshot  tty26     tty5      tty514     ttyUSBO    vsock
fd           mqueue   snd      tty27     tty50      tty515     uhid      zero
full         net       sr0      tty28     tty51      tty516     uinput
fuse         network_latency  stderr   tty29     tty52      tty517     urandom
hidraw0     network_throughput stdin    tty3      tty53      tty518     userio
hpet         null      stdout   tty30     tty54      tty519     vcs
hugepages    port      tty      tty31     tty55      tty52      vcs1
root@osrc-virtual-machine:~# s
```

Step2:用 screen 打开/dev/ttyUSB0

```
fb0          midi      snapshot  tty26      tty5      tty514     ttyUSBO    vsock
fd           mqueue   snd      tty27      tty50      tty515     uhid      zero
full         net       sr0      tty28      tty51      tty516     uinput
fuse         network_latency  stderr   tty29      tty52      tty517     urandom
hidraw0     network_throughput stdin    tty3      tty53      tty518     userio
hpet         null      stdout   tty30     tty54      tty519     vcs
hugepages    port      tty      tty31     tty55      tty52      vcs1
root@osrc-virtual-machine:~# screen /dev/ttyUSBO 115200
```

Step3:开发板根目录

设置开发板从 QSPI 模式启动，启动后，我们可以看到根目录下有 update_qspi.sh 脚本，这个脚本是 xilinx 提供的 ramdisk 就自带的，不过我们这里不是使用这种方法来升级 QSPI 镜像。

```
Freeing unused kernel memory: 1024K (c0800000 - c0900000)
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
random: sshd urandom read with 4 bits of entropy available
zynq> ls /
README      home      lost+found      root      update_qspi.sh
bin         lib       mnt      sbin      usr
dev         licenses  opt      sys      var
etc         linuxrc  proc      tmp
zynq>
```

5.5 基于 U 盘在线升级

Step1:为了展示升级的 QSPI 镜像与升级前的 QSPI 镜像不一样，或者说证明我们在线升级成功，我们将刚才看的到“update_qspi.sh”文件删掉，首先在 ubuntu 开发环境下执行 mount_rootfs.sh 脚本挂载根文件系统。

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# mount_rootfs.sh
根文件系统已经挂载在/mnt/workspace/linux/output/rootfs.
```

Step2:然后切换到 output/rootfs 目录下，执行 rm update_qspi.sh 命令将“update_qspi.sh”脚本删掉，最后执行 umount_rootfs.sh 脚本卸载、处理根文件系统。

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# cd ../output/rootfs/
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# ls
bin etc lib linuxrc mnt opt README sbin tmp usr
dev home licenses lost+found mount_emmc.sh proc root sys update_qspi.sh var
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# rm update_qspi.sh
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# ls
bin etc lib linuxrc mnt opt README sbin tmp var
dev home licenses lost+found mount_emmc.sh proc root sys usr
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# cd ..
root@osrc-virtual-machine:/mnt/workspace/linux/output# cd ..
root@osrc-virtual-machine:/mnt/workspace/linux# cd scripts
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# umount_rootfs.sh
Unmount the initrd image...
Compress the initrd image...
Wrapping the image with a U-Boot header...
Image Name:
Created:      Sun Apr  9 23:15:47 2017
Image Type:   ARM Linux RAMDisk Image (gzip compressed)
Data Size:    5307433 Bytes = 5183.04 kB = 5.06 MB
Load Address: 00000000
Entry Point:  00000000
root@osrc-virtual-machine:/mnt/workspace/linux/scripts#
```

Step3:执行 mk_qspi_image.sh 重新打包 QSPI 镜像，并将 images/qspi_image 目录的 QSPI 镜像拷贝到 U 盘中(可以用我们配套的 TF 卡+读卡器)。注意：0403-0201 这是系统自动命名的，大家要注意，不要变了名字就不认识了。

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# cp ../images/qspi_image/ /media/osrc/0403-0201/ -r
root@osrc-virtual-machine:/mnt/workspace/linux/scripts#
```

Step4:在正在运行的开发板子上插入 U 盘，在 screen 打开的终端下输入 ls /dev/命令查看 U 盘对应的设备，我这边对应的是/dev/sda 设备，通过 mount /dev/sda1 /mnt 将 U 盘挂载到/mnt 目录下，然后切换到 U 盘里存放 QSPI 镜像的目录下

```
zynq> mount /dev/sda1 /mnt/
FAT-fs (sda1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
zynq> cd /mnt/qspi_image/
zynq> ls
qspi_image.bin
zynq> 
```

Step5:通过 dd 命令烧录 QSPI 镜像，可以看到，在线升级会比通过 JTAG 方式快。

```
zynq> ls /mnt/qspi_image/qspi_image.bin
/mnt/qspi_image/qspi_image.bin
zynq> dd if=/mnt/qspi_image/qspi_image.bin of=/dev/mtdblock0
random: nonblocking pool is initialized
31106+1 records in
31106+1 records out
15926776 bytes (15.2MB) copied, 66.021045 seconds, 235.6KB/s
zynq> 
```

Step6:执行 reboot 重启开发板

```
zynq> reboot
zynq> 
```

Step7: 现在可以看到，已经没有 update_qspi.sh 文件了，说明升级成功了。

```
zynq> ls /
README          home        lost+found      proc        tmp
bin             lib         mnt           root        usr
dev             licenses    mount_emmc.sh  sbin        var
etc             linuxrc    opt            sys
zynq> 
```

S04_CH06_hello_linux

6.1 概述

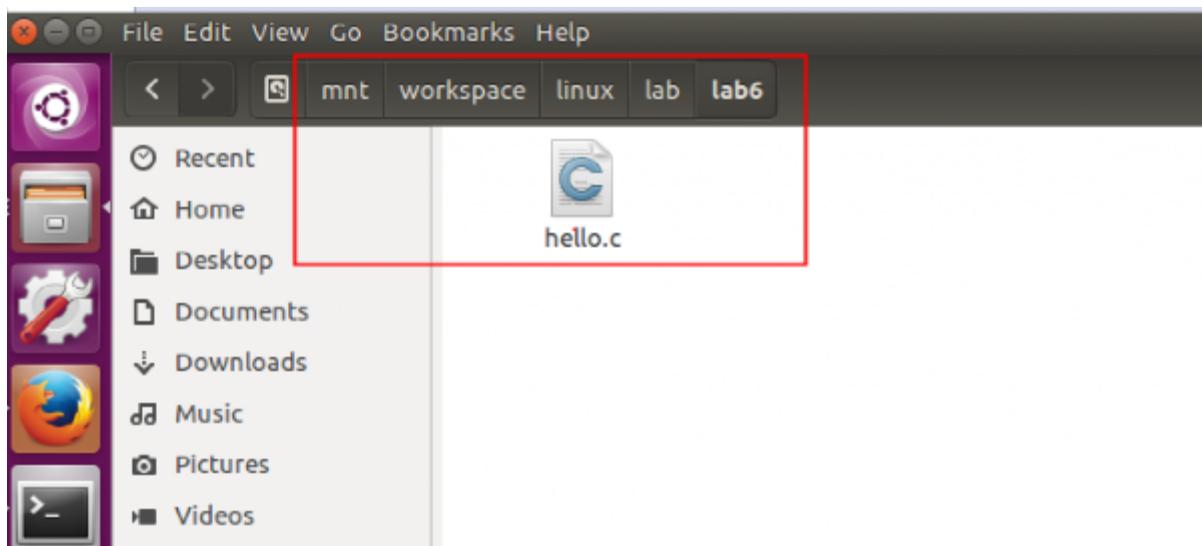
hello 是一个经典的程序，是学习入门必学的一个简单程序，能跑 hello linux 意味着编程者已经跨入编程的大门了。笔者这里的 hellolinux 程序需要在开发板的 linux 系统上执行 SD 卡运行和 EMMC 运行。

6.2 执行 source setup_env.sh

不管如何第一条指令我们要执行 source setup_env.sh 设置环境变量。

6.3 SD 卡手动运行 hello 程序

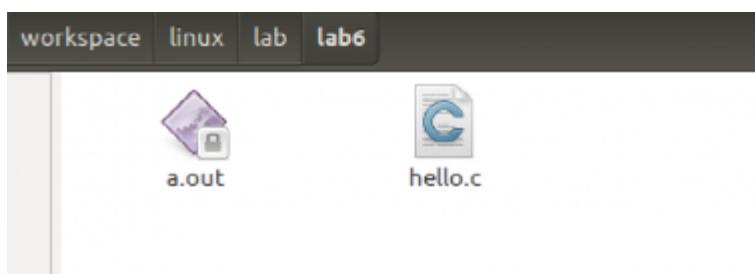
Step1:在新建 lab/lab6 文件夹，并且拷贝已经准备好的 hello.c 程序



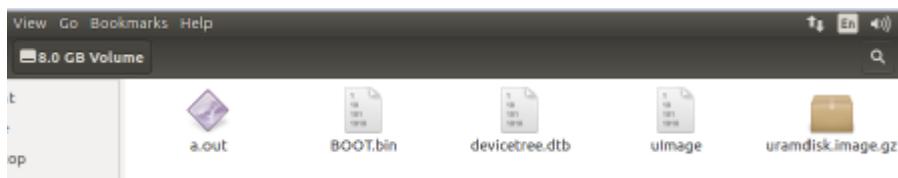
Step2:进入 lab6 文件夹

```
root@osrc-virtual-machine:/mnt/workspace/linux/lab/lab6#
```

Step3:执行 arm-xilinx-linux-gnueabi-gcc hello.c



Step3:拷贝 lab6 文件夹和 sd_image 文件中的文件到 TF 卡



Step4: 卸载 TF 卡，然后插入到开发板中，连接串口到系统，并且打开串口

```
root@osrc-virtual-machine:/mnt/workspace/linux/lab/lab6# ls /dev
agpgart      hw RNG   ppp    tty0   tty32  tty56   ttyS20  vcs2
autofs       initctl  psaux  tty1   tty33  tty57   ttyS21  vcs3
block        input    pts     tty10  tty34  tty58   ttyS22  vcs4
bsg          kmsg    random  tty11  tty35  tty59   ttyS23  vcs5
btrfs-control lightnvm  rfkill  tty12  tty36  tty6   ttyS24  vcs6
bus          log     rtc     tty13  tty37  tty60   ttyS25  vcs7
cdrom        loop0   sda     tty14  tty38  tty61   ttyS26  vcsa
cdrw          loop1   sda1    tty15  tty39  tty62   ttyS27  vcsa1
char          loop2   sda2    tty16  tty40  tty63   ttyS28  vcsa2
console       loop3   sda5    tty17  tty41  tty64   ttyS29  vcsa3
core          loop4   sdb     tty18  tty42  tty65   ttyS30  vcsa4
cpu           loop5   sdb1    tty19  tty43  tty66   ttyS31  vcsa5
cpu_dma_latency loop6   serial  tty20  tty44  tty67   ttyS32  vcsa6
cuse          loop7   sg0     tty21  tty45  tty68   ttyS33  vcsa7
disk          loop-control  sg1     tty22  tty46  tty69   ttyS34  vfio
dmmidi        mapper  sg2     tty23  tty47  tty70   ttyS35  vga_arbiter
dri          mcelog  sg3     tty24  tty48  tty71   ttyS36  vhci
dvd           mem    sg4     tty25  tty49  tty72   ttyS37  vhost-net
ecryptfs      memory_bandwidth  sg5     tty26  tty50  tty73   ttyS38  vmci
fb0          mid1    snapshot  tty27  tty51  tty74   ttyS39  vsock
fd          nqueue  stdio   tty28  tty52  tty75   ttyS40  zero
full         net     sr0     tty29  tty53  tty76   ttyS41
fuse          network_latency  stdio  tty30  tty54  tty77   ttyS42
hidraw0      network_throughput  stdio  tty31  tty55  tty78   ttyS43
hpet          null    stdio   tty32  tty56  tty79   ttyS44
hugepages     port    stdio   tty33  tty57  tty80   ttyS45
root@osrc-virtual-machine:/mnt/workspace/linux/lab/lab6# screen /dev/ttyUSB0 115200
```

Step5: 通电启动开发板，启动完成后，输入 ls /dev

```
zynq> ls /dev
bus          port      tty15      tty47
console      port      tty16      tty48
cpu_dma_latency ptmx     tty17      tty49
fbe          pts       tty18      tty5
full         ram0     tty19      tty50
gpiochip0    ram1     tty2       tty51
i2c          ram0     tty20      tty52
iio:device0  ram1     tty21      tty53
input         ram1     tty22      tty54
kmsg          ram1     tty23      tty55
loop-control  ram1     tty24      tty56
loop0         ram1     tty25      tty57
loop1         ram2     tty26      tty58
loop2         ram3     tty27      tty59
loop3         ram4     tty28      tty6
loop4         ram5     tty29      tty60
loop5         ram6     tty3       tty61
loop6         ram7     tty30      tty62
loop7         ram8     tty31      tty63
mem           ram9     tty32      tty7
memory_bandwidth random   tty33      tty8
rice          root     tty34      tty9
rncblk0      sda      tty35      ttyPS0
rncblk0boot0 sdai     tty36      urandom
rncblk0boot1 sg0      tty37      vcs
rncblk0pi    EMMC    snd      tty38      vcsi
rncblk0rpmb  timer   tty39      vcsa
rncblk1      tty      tty4       vcsa1
rncblk1pi   TF卡    tty0      tty40      vga_arbiter
rtdo          tty1     tty1       watchdog
rtdbo        tty10    tty41      watchdog
rtdbo        tty11    tty42      xdevcfg
network_latency  tty12  tty43      zero
network_throughput  tty13  tty44
null          null    tty14      tty45
null          null    tty15      tty46
```

由于本教程是基于做完第五课教材来做的，所以可以看到，mmcblk0p1 是 EMMC,而 mmcblk1p1 是 TF 卡（第四课里面调换了顺序 zynq-7000.dtsi）

Step6:

```
ls tmp (创建 tmp 文件夹)  
mount /dev/mmcblk1p1 /tmp (挂载 SD 到 tmp)  
cd /tmp (查看 tmo 下的目录)  
ls  
.a.out  
最后输出：Hello Linux!
```

```
zynq> ls tmp  
zynq> mount /dev/mmcblk1p1 /tmp  
FAT-fs (mmcblk1p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.  
zynq> cd /tmp  
zynq> ls  
BOOT.bin      devicetree.dtb      uramdisk.image.gz  
.a.out        uImage  
zynq> ./a.out  
Hello Linux!
```

6.4 EMMC 卡手动运行 hello 程序

Step1: 复制 a.out 可执行程序到 emmc

```
zynq> cd  
zynq> ls  
README      home      lost+found      proc      tmp  
bin         lib       mnt          root      update_qspi.sh  
dev         licenses  mount_emmc.sh  sbin      usr  
etc         linuxrc   opt          sys       var  
zynq> cp /tmp/a.out /mnt/emmc/ -r  
zynq> cd /mnt/emmc/  
zynq> ls  
.a.out      emmc      hello.txt    lost+found
```

Step2: 执行 ./a.out

```
zynq> ./a.out  
Hello Linux!  
zynq>
```

Step3: 断电重启，查看 EMMC 是否还有 a.out 程序

```
zynq> ls /mnt/emmc  
.a.out      emmc      hello.txt    lost+found  
zynq>
```

S04_CH07_Hello_Qt 在开发板上的运行

7.1 概述

本课程讲解 Qt 交叉编译环境的搭建，Qte 的安装，以及实现在开发板上运行 Qt 程序，并且实现开机启动。本教程使用的 QT 交叉编译环境版本是 qt-everywhere-opensource-src-5.7.0。Linux PC 端 安 装 的 环 境 是 qt-opensource-linux-x64-5.8.0。

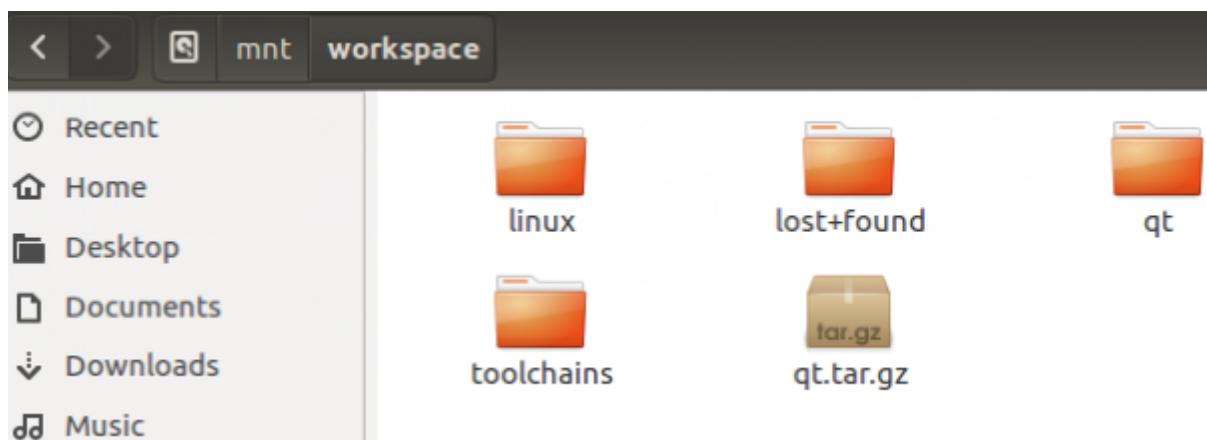
注意：本节课使用新 ramdisk，老的 ramdisk 不能正常运行。另外本节课开始使用的

另外要注意：SD 位于 /dev/mmcblk0p1，鼠标位于 /dev/event0 而第四节课

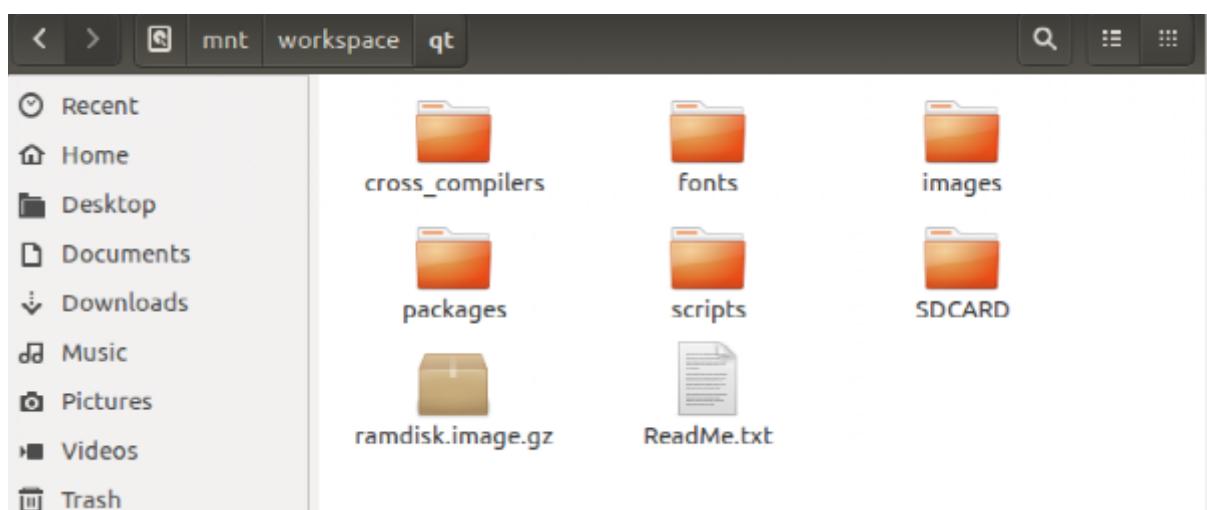
7.2 搭建交叉编译环境

7.2.1 使用批处理命令搭建交叉编译环境

Step1:首先把 qt.tar.gz 复制到 mnt/workspace 路径然后解压



查看 qt 文件夹



cross_compilers 文件夹：里面是交叉编译环境需要的文件

fonts 文件夹：里面放了 qt 需要用到的字库

images 文件夹：编译好的镜像文件和 init.sh 开机 packages 文件夹

packages 文件夹：里面有基于 xilinx 公司的 c/c++ 交叉编译文件、嵌入式交叉编译环境

qt-everywhere-opensource-src-5.7.0.tar.gz、 LINUX PC 端 Qt 安 装 环 境

qt-opensource-linux-x64-5.8.0.run

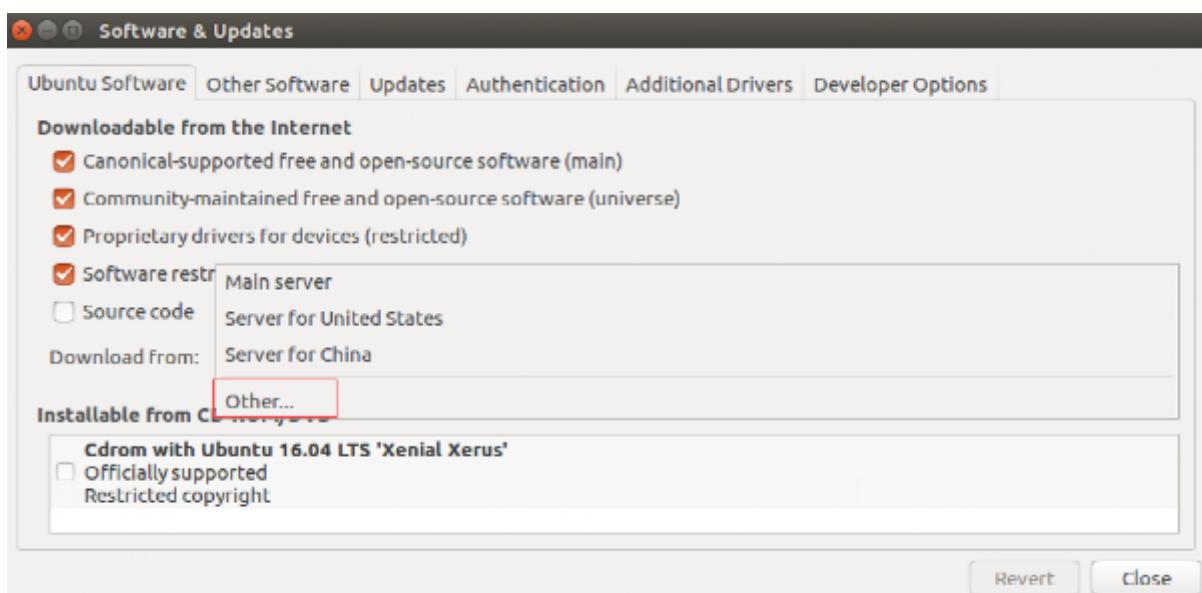
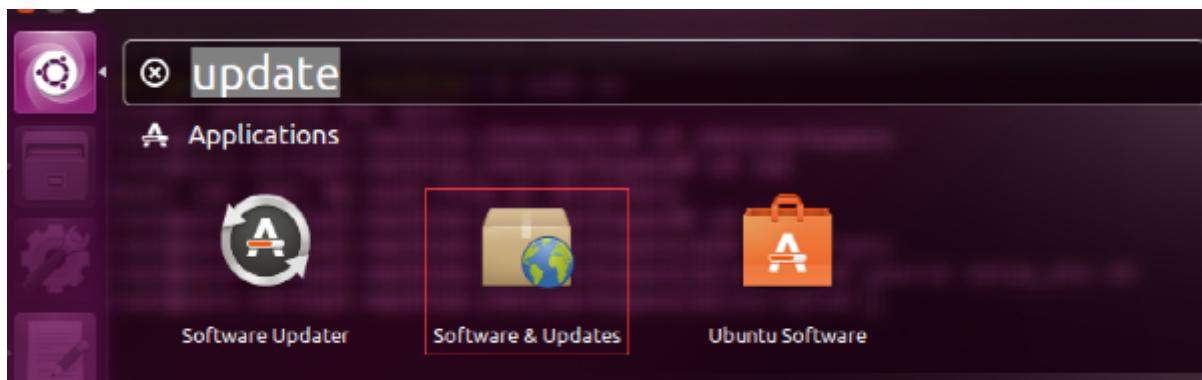
scripts 文件夹：下有三个批处理文件。

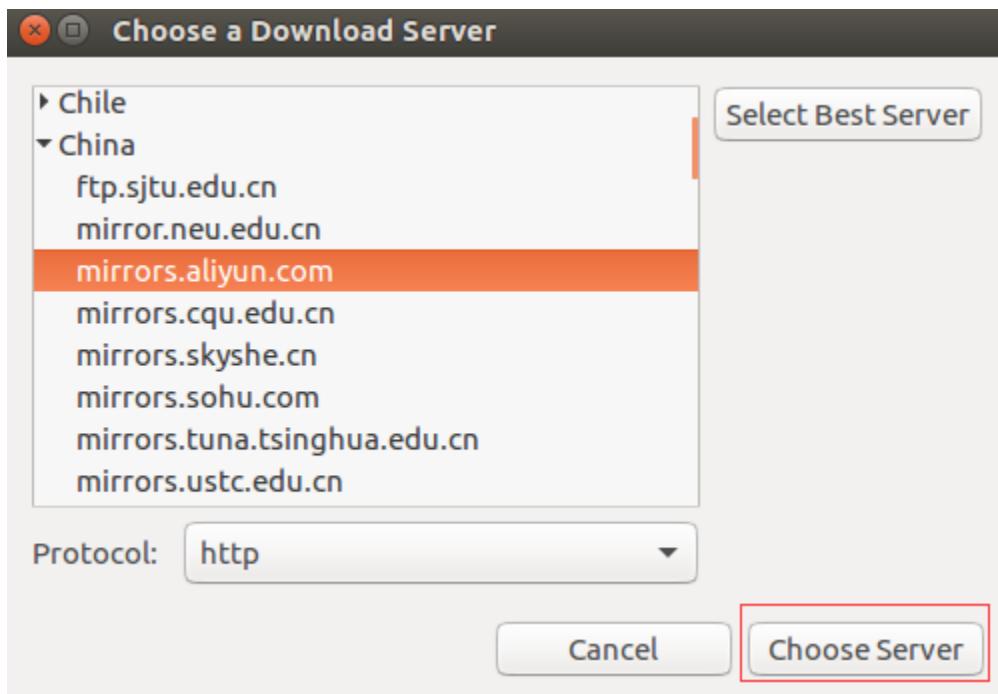
Setup_env.sh 设置或者创建了交叉编译器的路径、相关文件路径。

get_qt_sources.sh 获取需要使用到的资源

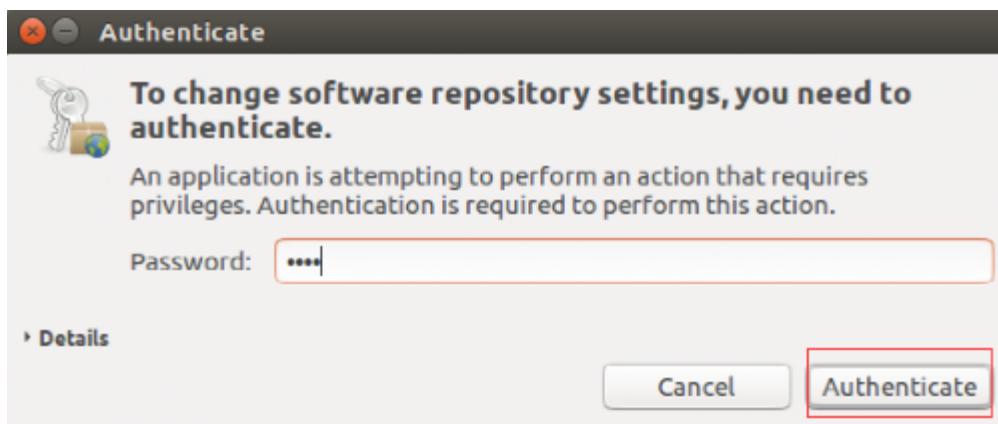
mk_qt_img.sh 包括了配置交叉编译环境、编译交叉编译环境、安装交叉编译器、编译自带的程序、制作 image 镜像。

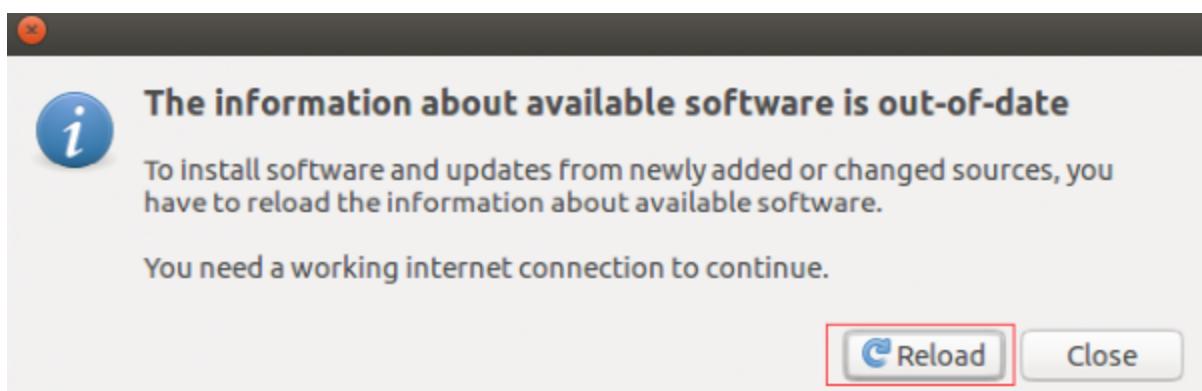
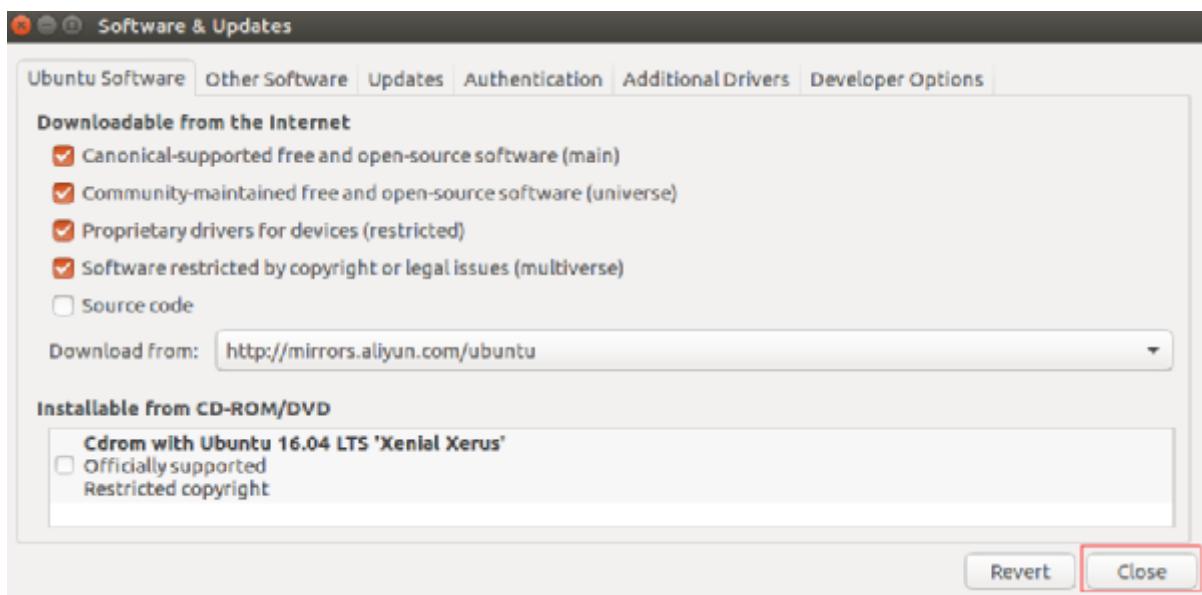
Step2: 设置更新的源





输入密码:root





Step3:安装一些必要的文件

```
sudo apt-get upgrade
```

```
sudo apt-get update
```

```
sudo apt-get install libgtk2.0-0:i386 libxtst6:i386 gtk2-engines-murrine:i386 \lib32stdc++6  
libxt6:i386 libdbus-glib-1-2:i386 libasound2:i386
```

```
sudo apt-get install build-essential
```

Step3:切换到 scripts 目录下，执行 source setup_env.sh 配置所需要的 QT 开发环境

```
root@osrc-virtual-machine:/mnt/workspace/qt/scripts# source setup_env.sh  
root@osrc-virtual-machine:/mnt/workspace/qt/scripts#
```

Step3:执行 get_qt_sources.sh，获取所需源码

```
.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/main.cpp
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/BasicCamera.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/HousePlant.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/SortedForwardRen-
derer.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/PlaneEntity.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/materials.qrc
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/src/
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/src/material-
s.qdoc
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/images/
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/images/mater-
ials.png
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/Barrel.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/main.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/materials.pro
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/TrefoilKnot.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/Chest.qml
qt-everywhere-opensource-src-5.7.0/qt3d/LICENSE.LGPL3
qt-everywhere-opensource-src-5.7.0/qt3d/LICENSE.GPL
qt-everywhere-opensource-src-5.7.0/qt3d/LICENSE.GPL2
root@osrc-virtual-machine:/mnt/workspace/qt/scripts#
```

Step4: 执行 mk_qt_img.sh 脚本。

如果是一开始构建的话，可以在第一步“gmake confclean”选择 n，其它选择 y，这样就完成了 QT 库的编译。当然，也可以是一路 y。而如果你只是想执行

其中的某一步，如最后一步，可以其它 n，最后一步才是 y，总之，可以根据实际情况进行

选择。另外，请根据实际情况，修改脚本中的要相关配置。

```
root@osrc-virtual-machine:/mnt/workspace/qt/scripts# mk_qt_img.sh
Run 'gmake confclean'? (y/n)
```

```
XCB ..... no
Session management .... yes
SQL drivers:
  DB2 ..... no
  InterBase ..... no
  MySQL ..... no
  OCI ..... no
  ODBC ..... no
  PostgreSQL ..... no
  SQLite 2 ..... no
  SQLite ..... yes (plugin, using bundled copy)
  TDS ..... no
  tslib ..... no
  udev ..... no
  xkbcommon-x11..... no
  xkbcommon-evdev..... no
  zlib ..... yes (bundled copy)
```

```
NOTE: Qt is using double for qreal on this system. This is binary incompatible a-
gainst Qt 5.1.
Configure with '-qreal float' to create a build that is binary compatible with 5
.1.
Info: creating stash file /mnt/workspace/qt/qt-src-5.7.0/.qmake.stash
Info: creating super cache file /mnt/workspace/qt/qt-src-5.7.0/.qmake.super
```

```
Qt is now configured for building. Just run 'make'.
Once everything is built, you must run 'make install'.
Qt will be installed into /mnt/workspace/qt/qt-arm-5.7.0
```

```
Prior to reconfiguration, make sure you remove any leftovers from
the previous build.
```

```
Run 'qmake'? (y/n) y
```

编译完成后进行安装

```
/mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/lrelease_wrapper.sh q
txmlpatterns_ru.ts -qm qtxmlpatterns_ru.qm
Updating 'qtxmlpatterns_ru.qm'...
    Generated 455 translation(s) (455 finished and 0 unfinished)
    Ignored 25 untranslated source text(s)
/mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/lrelease_wrapper.sh q
txmlpatterns_sk.ts -qm qtxmlpatterns_sk.qm
Updating 'qtxmlpatterns_sk.qm'...
    Generated 151 translation(s) (151 finished and 0 unfinished)
    Ignored 308 untranslated source text(s)
/mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/lrelease_wrapper.sh q
txmlpatterns_uk.ts -qm qtxmlpatterns_uk.qm
Updating 'qtxmlpatterns_uk.qm'...
    Generated 49 translation(s) (49 finished and 0 unfinished)
    Ignored 431 untranslated source text(s)
make[2]: Leaving directory '/mnt/workspace/qt/qt-src-5.7.0/qttranslations/transl
ations'
make[1]: Leaving directory '/mnt/workspace/qt/qt-src-5.7.0/qttranslations'
Run 'gmake install'? (y/n) y
```

接下来编译自带的 qttranslations 程序

```
install -m 644 -p /mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/qtx
mlpatterns_pl.qm /mnt/workspace/qt/qt-arm-5.7.0/translations/
uninstall -m 644 -p /mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/qtx
mlpatterns_ru.qm /mnt/workspace/qt/qt-arm-5.7.0/translations/
uninstall -m 644 -p /mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/qtx
mlpatterns_sk.qm /mnt/workspace/qt/qt-arm-5.7.0/translations/
uninstall -m 644 -p /mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/qtx
mlpatterns_uk.qm /mnt/workspace/qt/qt-arm-5.7.0/translations/
make[2]: Leaving directory '/mnt/workspace/qt/qt-src-5.7.0/qttranslations/transl
ations'
make[1]: Leaving directory '/mnt/workspace/qt/qt-src-5.7.0/qttranslations'
是否编译例子? (y/n) y
mkdir: cannot create directory '/mnt/workspace/qt/Images': File exists
make: *** No targets specified and no makefile found. Stop.
cp: omitting directory '/mnt/workspace/qt/qt-src-5.7.0/qtbase/examples/widgets/a
nimation/animatedtiles'
To create a image? (y/n) y
+ Create a image? (y/n) y
80+0 records in
80+0 records out
83886080 bytes (84 MB, 80 MiB) copied, 0.09341 s, 898 MB/s
mke2fs 1.42.13 (17-May-2015)
Discarding device blocks: done
Creating filesystem with 81920 1k blocks and 20480 inodes
Filesystem UUID: fe4b529f-088a-4e3b-afc5-6cf9311837e7
Superblock backups stored on blocks:
    8193, 24577, 40961, 57345, 73729

Allocating group tables: done
Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done
root@osrc-virtual-machine:/mnt/workspace/qt/scripts#
```

Step5: 编译好后, 将 images 目录下的 animatedtiles、QT5.7.0.image、init.sh 拷贝到 SD 卡中。

Step6: 确保你的 SD 位于 /dev/mmcblk0p1, 鼠标位于 /dev/event0, 这样, 开机将自动启动 QT。

cd /usr/bin && sudo ln -s make gmake 如果 gmake 不能正确安装

7.2.2 setup_env.sh 批处理文件源码

```
#!/bin/bash
#####
# Qt is a C++ framework for GUI application development. With the release of Qt 5.0,
```

```
# Qt no longer contains its own window system implementation. The Qt Platform
# Abstraction (QPA) provides multiple platform plugins that are potentially usable
# on Embedded Linux systems: EGLFS, LinuxFB, DirectFB, Wayland. In this tutorial we
# configure Qt to use the LinuxFB plugin which is well suited for embedded devices
# without GPU running Linux.

#####
#
# 提示使用 source 命令
#
# source 命令也称为“点命令”，也就是一个点符号(.)；通常用于重新执行刚修改的初始
# 化文件，使之立即生效，而不必注销并重新登录。
if[ $BASH_SOURCE == $0 ];then
    echo -e "\nThis script should be sourced, like \"source ${0##*/}\".\n"
    exit 1
fi

# 脚本所在目录
export SCRIPTSDIR="$(dirname $(readlink -f $0))"
# 项目根目录
export SRCROOT="$(dirname $SCRIPTSDIR)"
# 方便在各个地方执行 scripts 里的脚本
export PATH=${SCRIPTSDIR}:$PATH

# QT 版本号
export QT_MAJOR_VERSION=5
export QT_MINOR_VERSION=7
export QT_PATCH_VERSION=0
export
QT_VERSION=${QT_MAJOR_VERSION}.${QT_MINOR_VERSION}.${QT_PATCH_VERSION}

# 交叉编译器
export PATH=${SRCROOT}/cross_compilers/bin:$PATH
# for Zynq-7000 (Linaro - hard float)
# if which arm-linux-gnueabihf-gcc >/dev/null; then
#     export CROSS_COMPILE=arm-linux-gnueabihf-
# for Zynq-7000 (CodeSourcery - soft float)
# elif which arm-xilinx-linux-gnueabi-gcc >/dev/null; then
#     export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
# else
#     echo "无法找到交叉编译器！！！"
#     return 1
# fi
```

```
# 资源包
export PACKAGES=${SRCROOT}/packages

# 镜像
export IMAGES=${SRCROOT}/images

# QT 源码目录
export QTSRC=${SRCROOT}/qt-src-${QT_VERSION}

# QT 库目录
export QTDIR=${SRCROOT}/qt-arm-${QT_VERSION}
export PATH=${QTDIR}/bin:$PATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${QTDIR}/lib

# APP
export QTAPP=${SRCROOT}/app

# Invoke a second make in the output directory, passing relevant variables
# check that the output directory actually exists
mkdir -p ${PACKAGES} ${IMAGES} ${QTDIR}
```

7.2.3 get_qt_sources.sh 批处理文件源码

```
#!/bin/bash
#####
# 文件名: get_qt_sources.sh
# 功 能: 获取 QT 源码
# 版本号: V 0.2
#####
source echo_color.sh
#
# 提示使用 source 命令
#
# source 命令也称为“点命令”，也就是一个点符号(.)；通常用于重新执行刚修改的初始
# 化文件，使之立即生效，而不必注销并重新登录。
if [ ! "${CROSS_COMPILE}" ];then
    echo_red "请切换到 script 目录下，执行 'source setup_env.sh' 命令来配置环境变量。"
    exit 1
fi

# 准备开发环境
```

```
# Download the Qt sources and extract the archive to your Qt build area.  
tar zxvf ${PACKAGES}/qt-everywhere-opensource-src-${QT_VERSION}.tar.gz  
mv ./qt-everywhere-opensource-src-${QT_VERSION} ${QTSRC}  
  
# 2. Prepare a mkspec  
#  
# Before we can do the configuration for the target system, we will need a set  
# of mkspecs that tells qmake which tool chain it should reference when it creates  
# the Makefiles. In this example we will provide an mkspec to go along with the  
# ARM GNU tools.  
cp ${PACKAGES}/xlnx-gnueabi-g++ ${QTSRC}/qtbase/mkspecs/ -R  
cp ${PACKAGES}/xlnx-gnueabihf-g++ ${QTSRC}/qtbase/mkspecs/ -R
```

7.2.4 mk_qt_img.sh 批处理文件源码

```
#!/bin/bash  
#  
# sudo apt-get install build-essential perl python git  
#  
# http://doc.qt.io/  
# http://doc.qt.io/qt-5/embedded-linux.html  
# http://www.it165.net/embed/html/201606/3507.html  
#  
#####  
# Qt is a C++ framework for GUI application development. With the release of Qt 5.0,  
# Qt no longer contains its own window system implementation. The Qt Platform  
# Abstraction (QPA) provides multiple platform plugins that are potentially usable  
# on Embedded Linux systems: EGLFS, LinuxFB, DirectFB, Wayland. In this tutorial we  
# configure Qt to use the LinuxFB plugin which is well suited for embedded devices  
# without GPU running Linux.  
#####  
source echo_color.sh  
#  
# 提示使用 source 命令  
#  
# source 命令也称为“点命令”，也就是一个点符号(.)；通常用于重新执行刚修改的初始  
# 化文件，使之立即生效，而不必注销并重新登录。  
if [ ! "${CROSS_COMPILE}" ];then  
    echo_red "请切换到 script 目录下，执行 'source setup_env.sh' 命令来配置环境变量."  
    exit 1  
fi  
# 检查源码是否存在
```

```
if[ ! -d "${QTDIR}" ]; then
    echo_red "请执行 'get_qt_sources.sh' 下载所需源码。"
    exit 1
else
    cd ${QTDIR}
fi

# 1. To reconfigure, run 'gmake confclean' and 'configure'.
# 注意：仅在重新配置里需要执行此步骤
read -p "Run 'gmake confclean'? (y/n) " REPLY
if[ "$REPLY" == "y" ]; then
    make distclean
    rm ${QTDIR}/* -r
fi

# 2. Configure the Target Build
read -p "Configure the Target Build (y/n) " REPLY
if[ "$REPLY" == "y" ]; then
# 请根据实际需要修改些配置
./configure -opensource -confirm-license -verbose -release \
    -xplatform xlnx-gnueabi-g++ -prefix ${QTDIR} \
    -nomake examples \
    -no-tslib \
    -no-directfb \
    -no-opengl \
    -no-eglfs \
    -no-xcb \
    -no-iconv \
    -qt-libpng \
    -qt-libjpeg \
    -qt-freetype \
    -skip virtualkeyboard \
    -skip qtdoc
fi

# 3. Run make to build the cross-compiled target version of Qt.
read -p "Run 'gmake'? (y/n) " REPLY
if[ "$REPLY" == "y" ]; then
    make
fi

# 4. Once the build has completed it is time to install Qt. You may need to su
# to root to do this part depending upon what prefix you configured the build with.
```

```
read -p "Run 'gmake install'? (y/n) " REPLY
if [ "$REPLY" == "y" ]; then
    sudo make install
    # 如果根文件系统中缺少 c++库，需要补全该库
    cp -P ${SRCROOT}/cross_compilers/arm-xilinx-linux-gnueabi/libc/usr/lib/libstdc++.so* $QTDIR/lib
fi

# 5. 编译 QT 源码中自带的例子
read -p "是否编译例子? (y/n) " REPLY
if [ "$REPLY" == "y" ]; then
    mkdir ${IMAGES}
    cd ${QTDIR}/qtbase/examples/widgets(animation/animatedtiles
    make
    cp ${QTDIR}/qtbase/examples/widgets(animation/animatedtiles(animatedtiles ${IMAGES}
fi
#####
# 编译完 Qt 后，只需将生成的 lib 和 plugins 文件夹拷贝到开发板，另外，当在嵌入式 Linux
# 平台上运行应用程序前，应根据自己平台的实际情况提前设置好下面几个环境变量：
#
# export QT_QPA_PLATFORM_PLUGIN_PATH=/opt/Qt-5.3.2/armv7-a/plugins/platforms
# export QT_QPA_PLATFORM=linuxfb:tty=/dev/fb0
# export QT_QPA_FONTDIR=/opt/Qt-5.3.2/armv7-a/lib/fonts
# export QT_QPA_GENERIC_PLUGINS=tslib:/dev/touchscreen-1wire #使用 tslib 插件
#
# 然后就可以运行 Qt 程序了
#####

# 6. Create a File System Image with Pre-Compiled Qt/Qwt Libraries
# http://www.wiki.xilinx.com/Zynq+Qt+and+Qwt+Base+Libraries-Build+Instructions
read -p "To create a image? (y/n) " REPLY
if [ "$REPLY" == "y" ]; then

    TARGET=${IMAGES}/QT${QT_VERSION}.image
    TEMP=${SRCROOT}/temp
    mkdir -p ${TEMP}

    # 制作一个 80M 的镜像文件，请根据实际情况修改
    dd if=/dev/zero of=${TARGET} bs=1M count=80

    mkfs.ext4 -F ${TARGET} -L "QT"
    chmod a+rwx ${TARGET}
    sudo mount -o loop ${TARGET} ${TEMP}
```

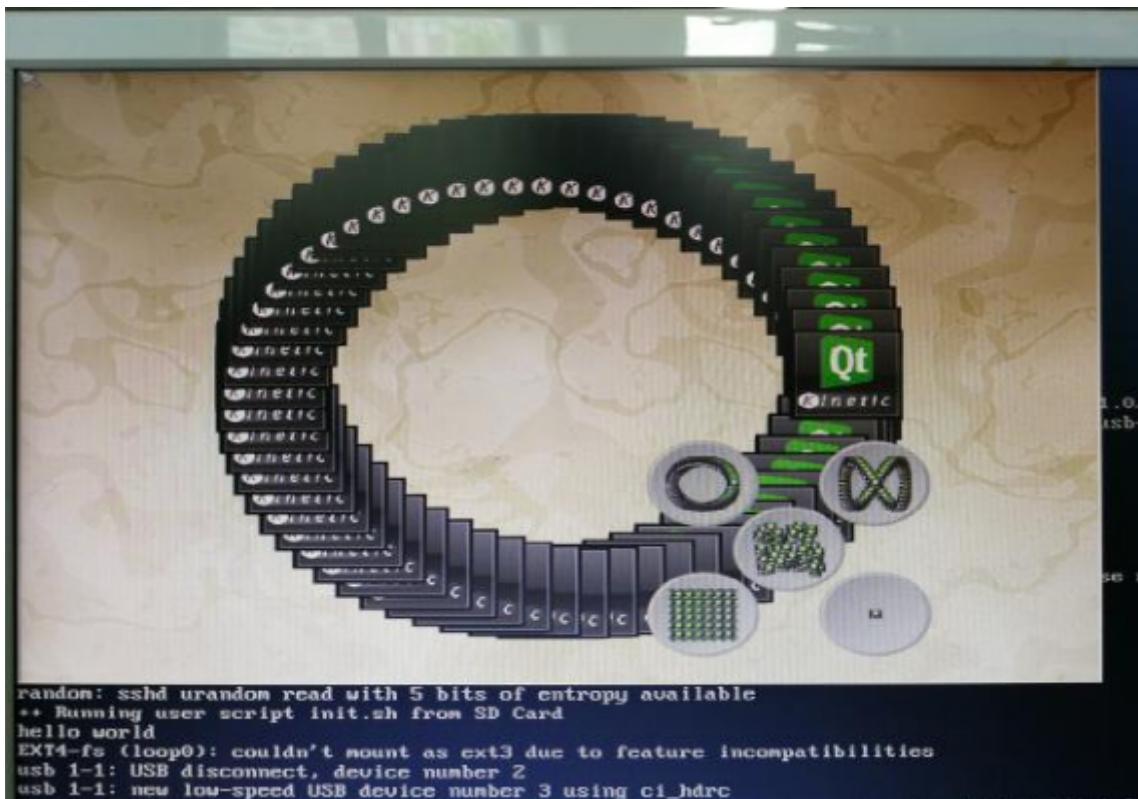
```
cp -r ${QTDIR}/lib ${TEMP}  
cp -r ${QTDIR}/plugins ${TEMP}  
  
umount ${TEMP}  
  
fi
```

7.2.4 init.sh 文件

```
#!/bin/sh  
  
echo "hello world"  
  
export QTDIR=/opt/QT5.7  
mkdir -p ${QTDIR}  
  
mount -o loop /mnt/QT5.7.0.image ${QTDIR}  
  
export QT_QPA_FONTPATH=${QTDIR}/lib/fonts  
export QT_QPA_PLATFORM_PLUGIN_PATH=${QTDIR}/plugins/  
export LD_LIBRARY_PATH=${QTDIR}/lib:$LD_LIBRARY_PATH  
  
export QT_QPA_PLATFORM=linuxfb:fb=/dev/fb0  
  
#http://www.360doc.com/content/14/1215/10/18578054_433033534.shtml  
export QT_QPA_EVDEV_MOUSE_PARAMETERS=evdevmouse:/dev/event0  
  
/mnt/animatedtiles &
```

7.2.5 测试结果

编译好后，将 images 目录下的 animatedtiles、QT5.7.0.image、init.sh 拷贝到 SD 卡中。确保你的 SD 位于/dev/mmcblk0p1，鼠标位于/dev/event0，这样，开机将自动启动 QT。



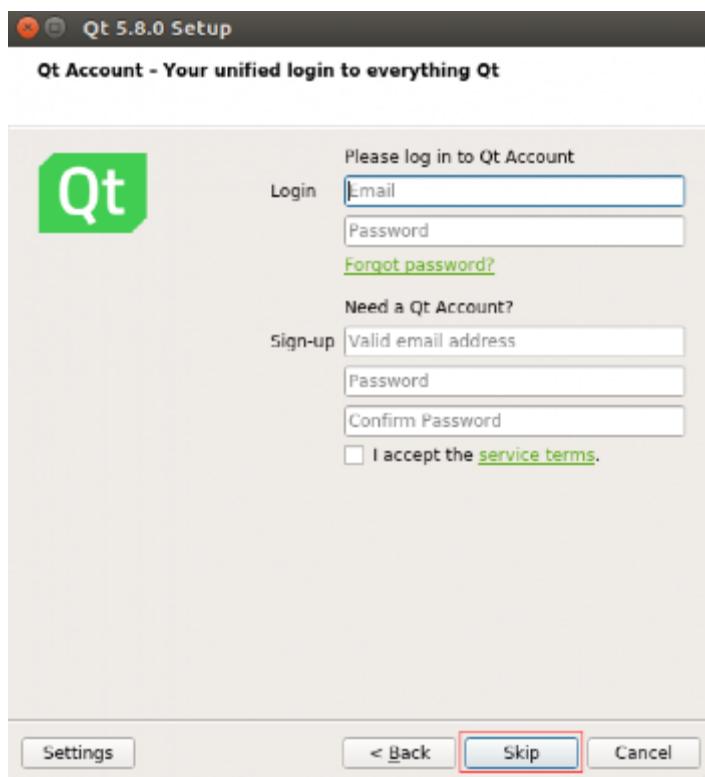
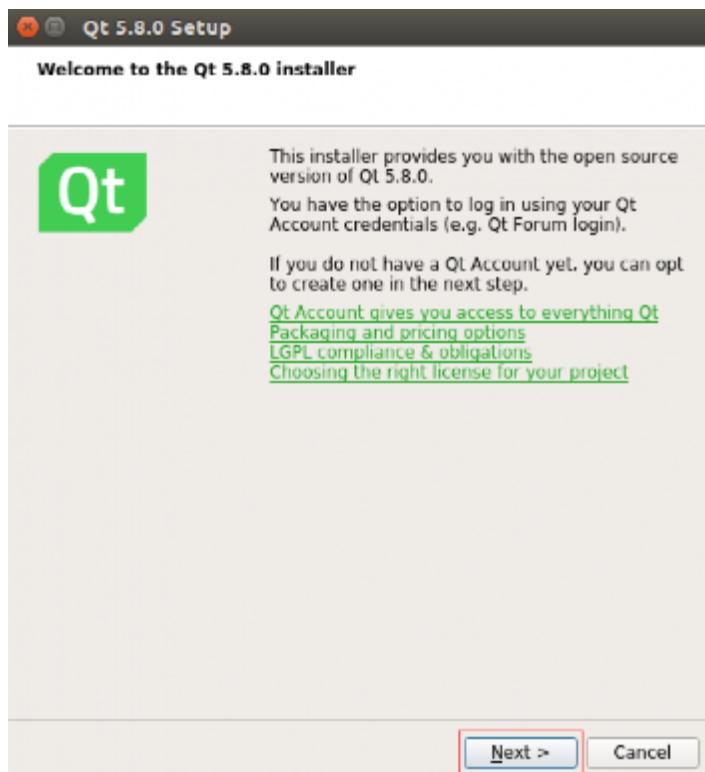
7.3 在 PC 端 LINUX 安装 qt5.8.0

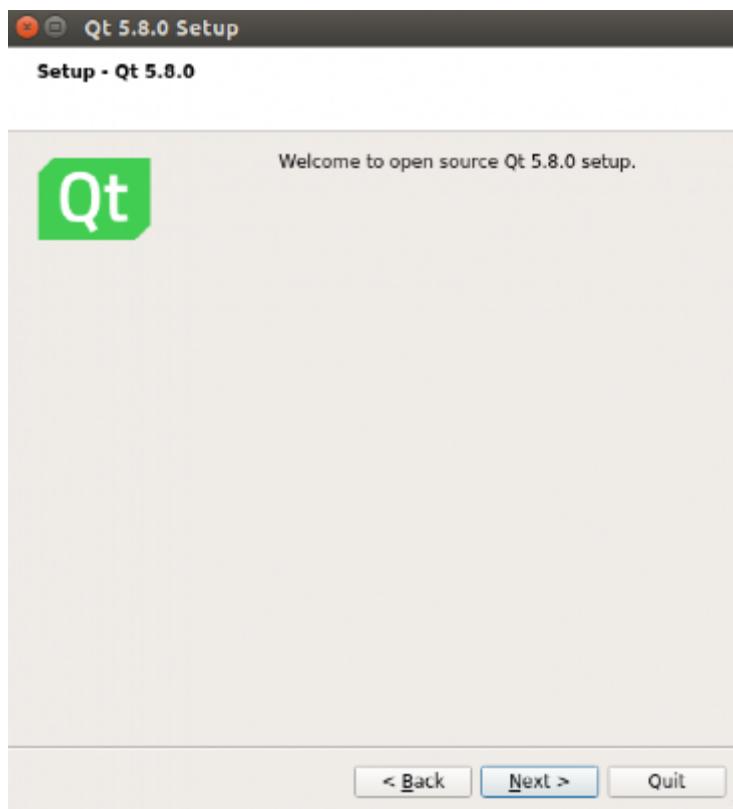
Step1:进入 mnt/workspace/qt/packages 文件夹下

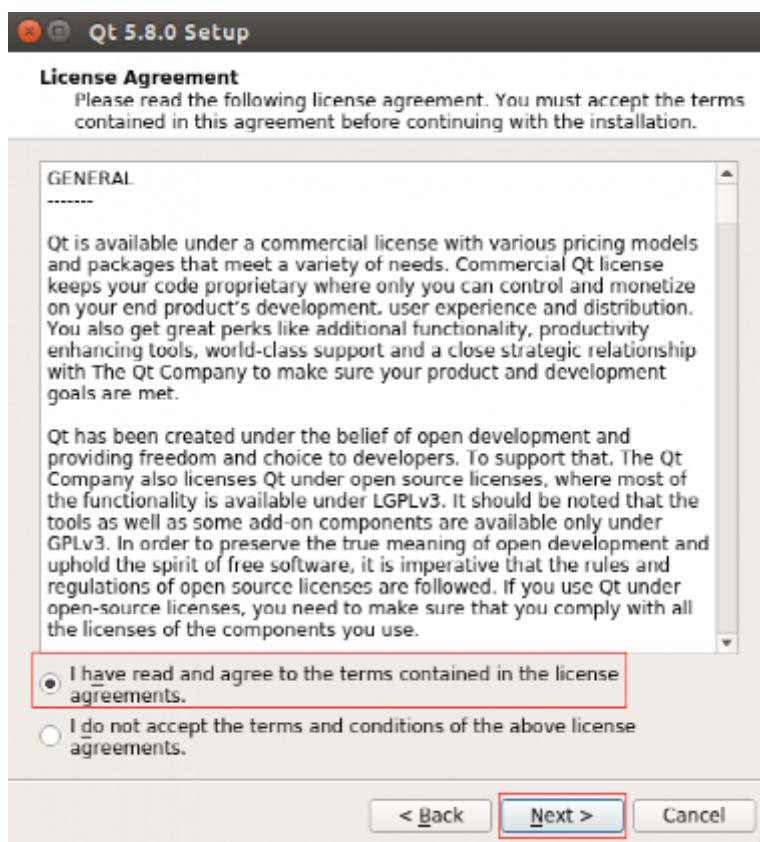
Step2: 执行 chmod +x qt-opensource-linux-x64-5.8.0.run

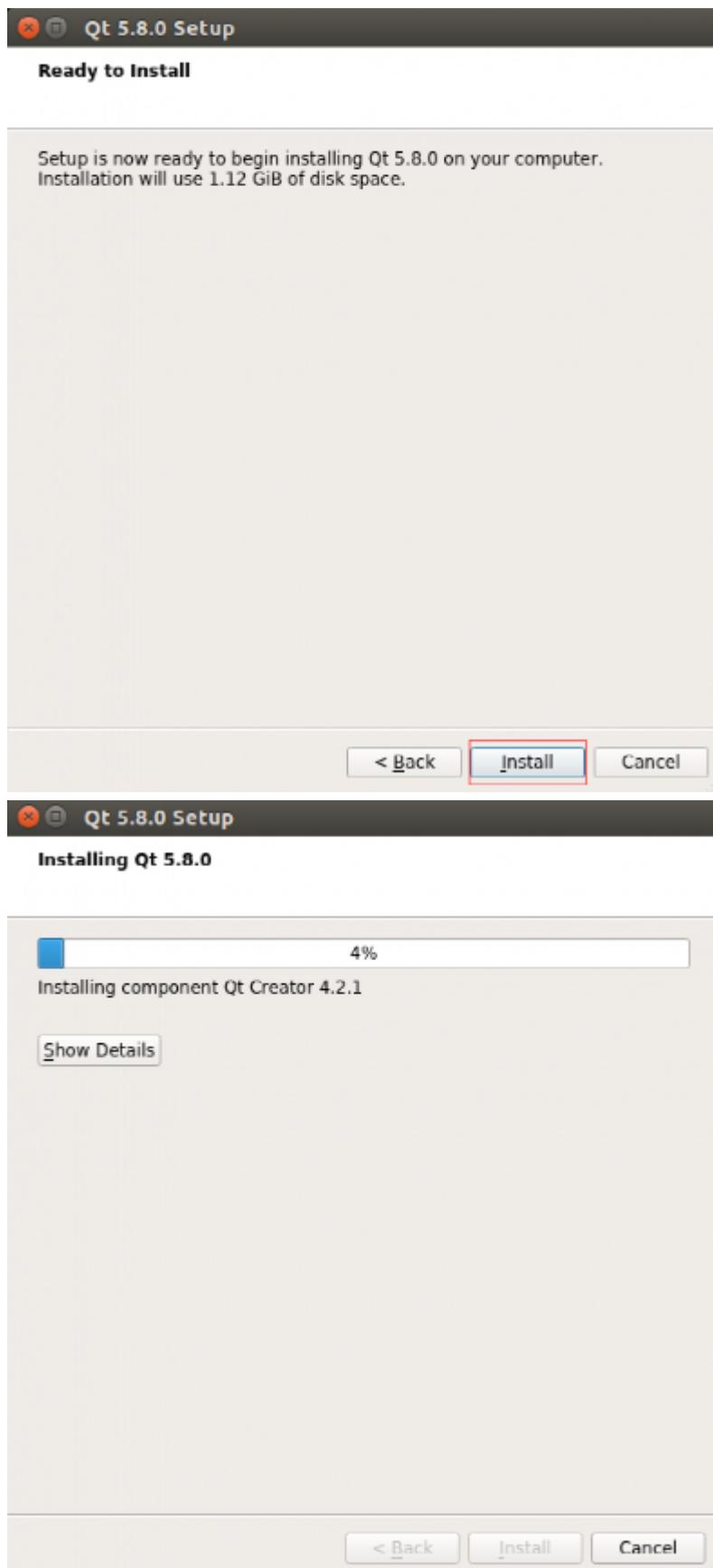
Step3: 执行./qt-opensource-linux-x64-5.8.0.run

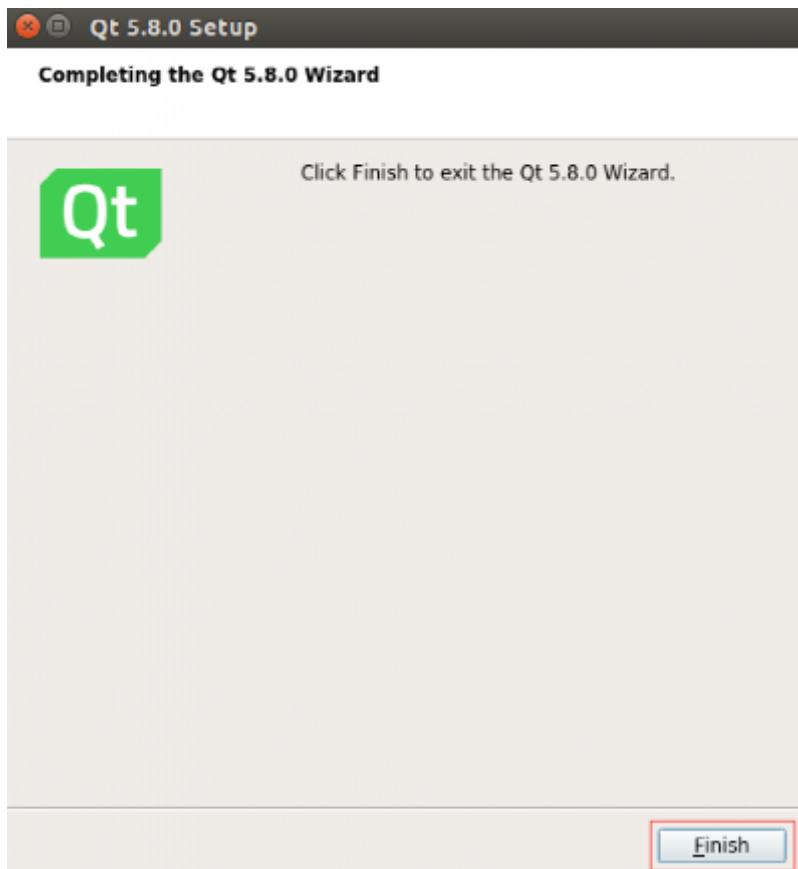
```
root@osrc-virtual-machine:/mnt/workspace/qt/scripts# cd ../packages
root@osrc-virtual-machine:/mnt/workspace/qt/packages# chmod +x qt-opensource-lin
ux-x64-5.8.0.run
root@osrc-virtual-machine:/mnt/workspace/qt/packages# ./qt-opensource-linux-x64-
5.8.0.run
```









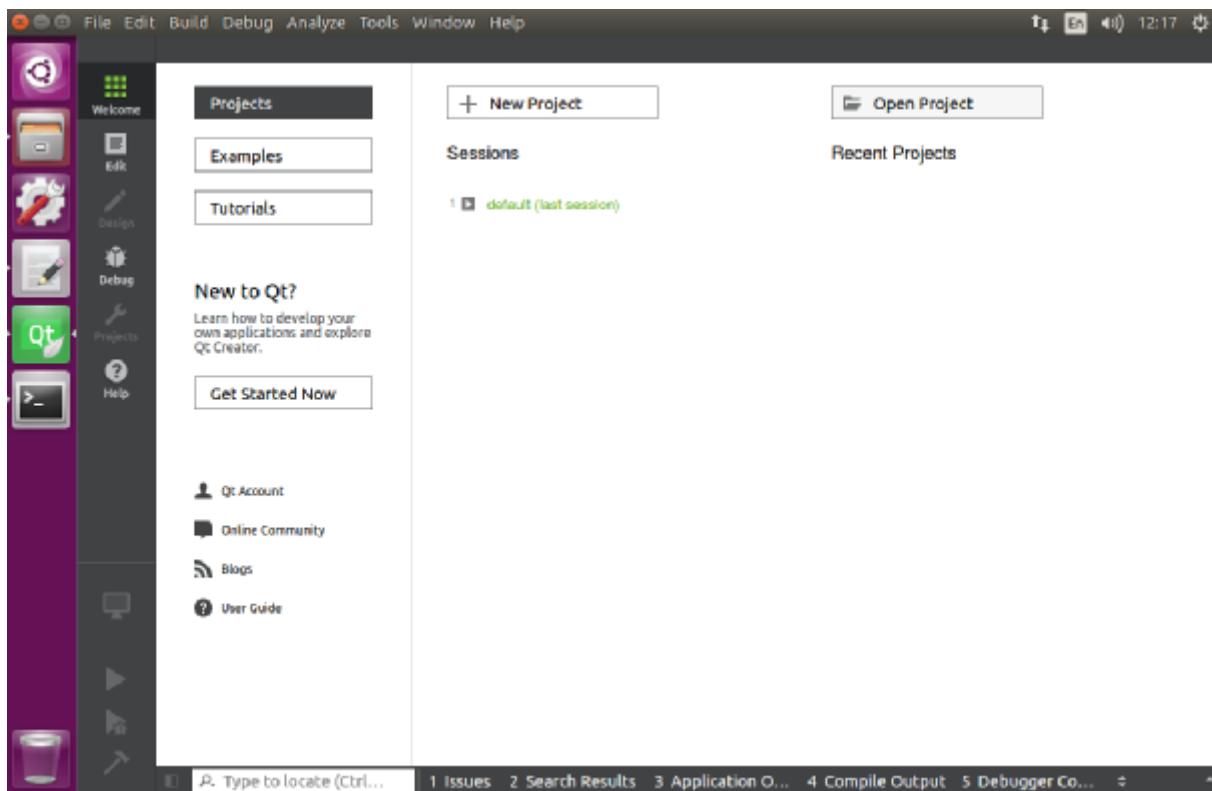


安装完成

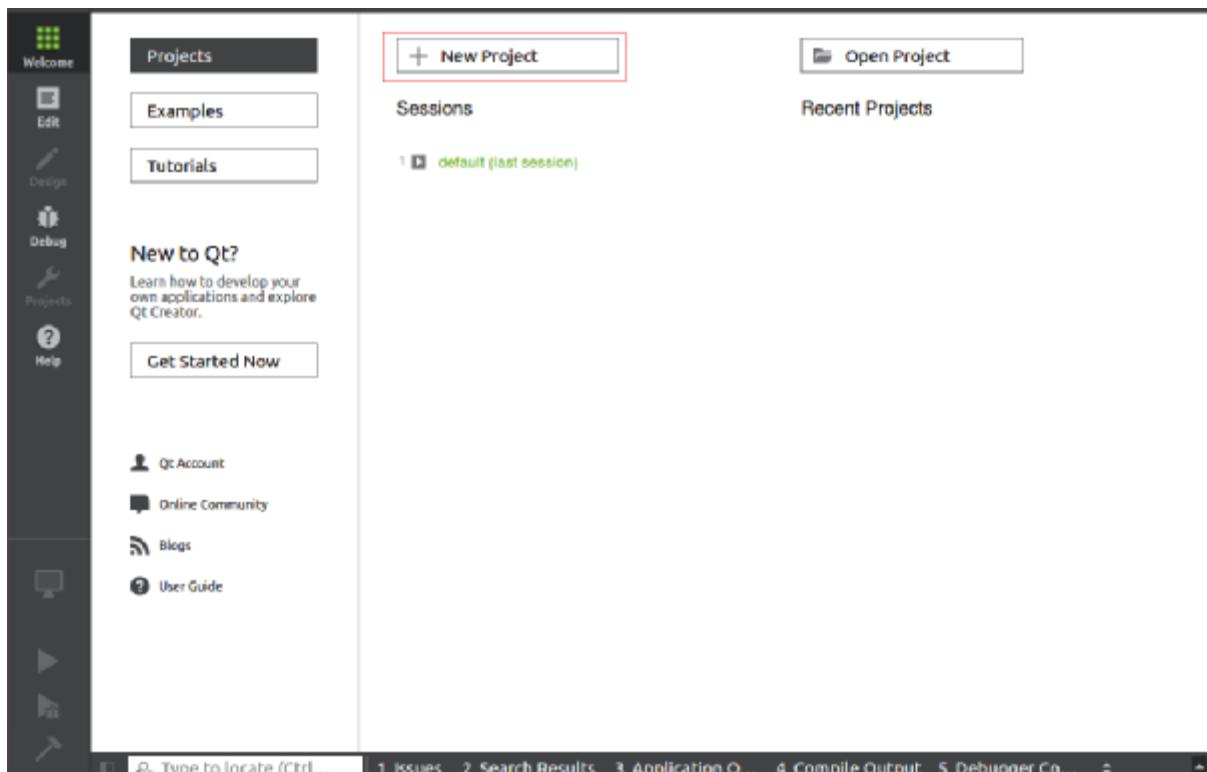
7.4 QtE LINUX PC 端创建工程

Step1: 双击 qtcreator 启动 qte

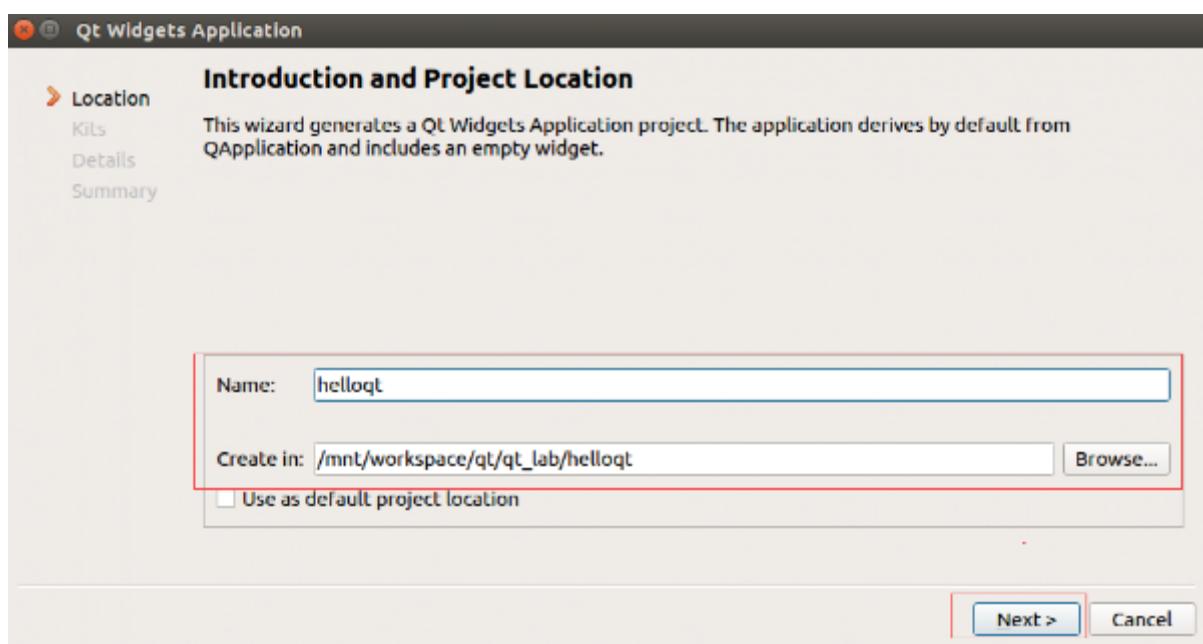
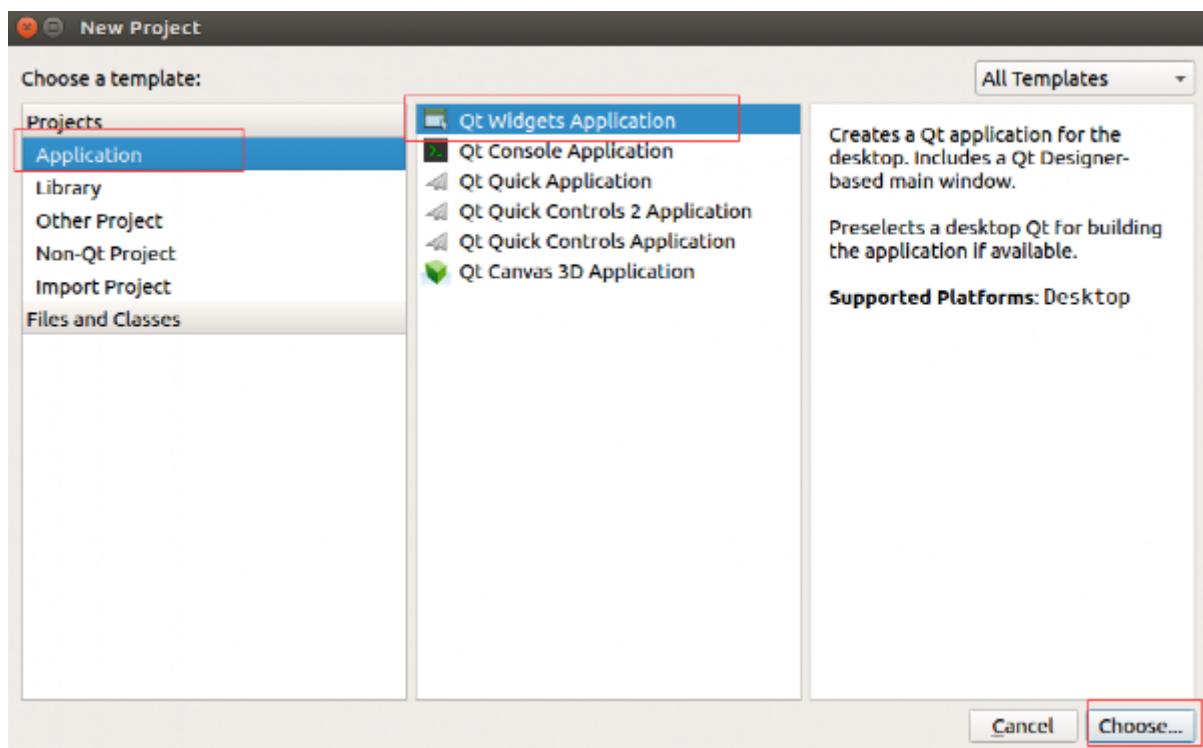




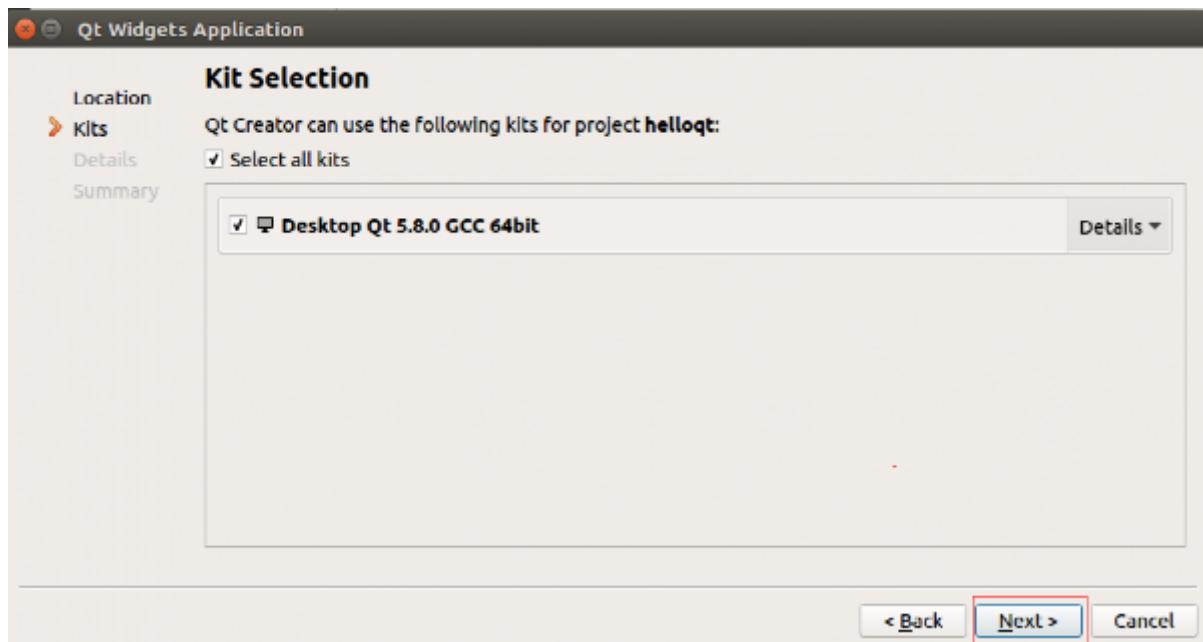
Step2:单击 New Project 新建一个 qt 工程



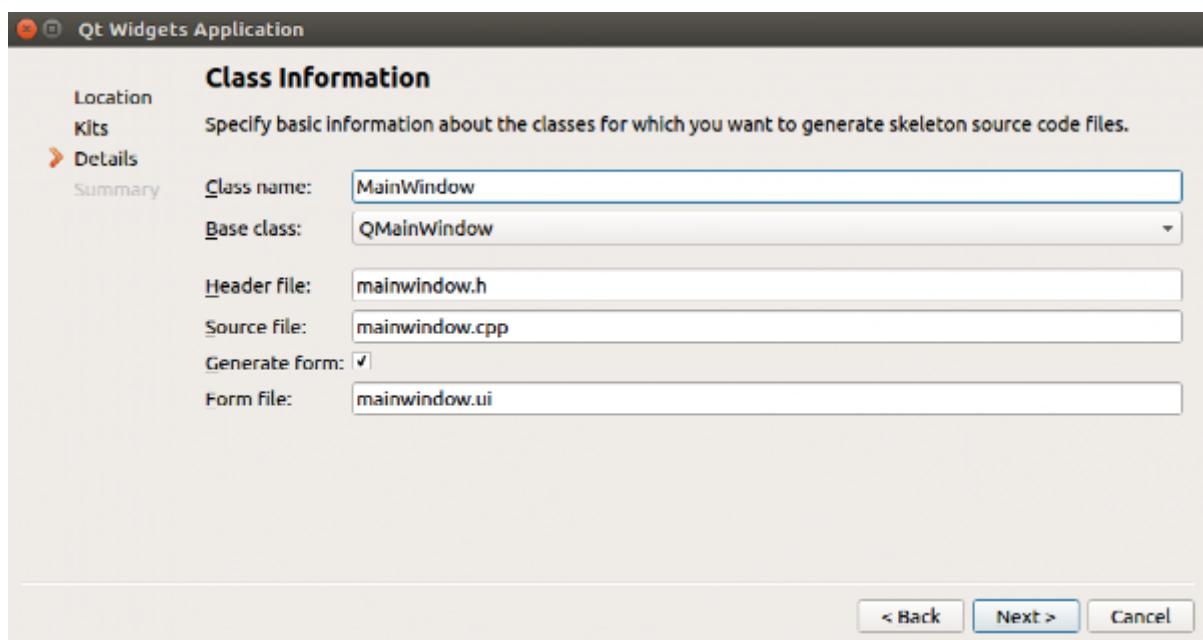
Step3:设置路径和工程名字



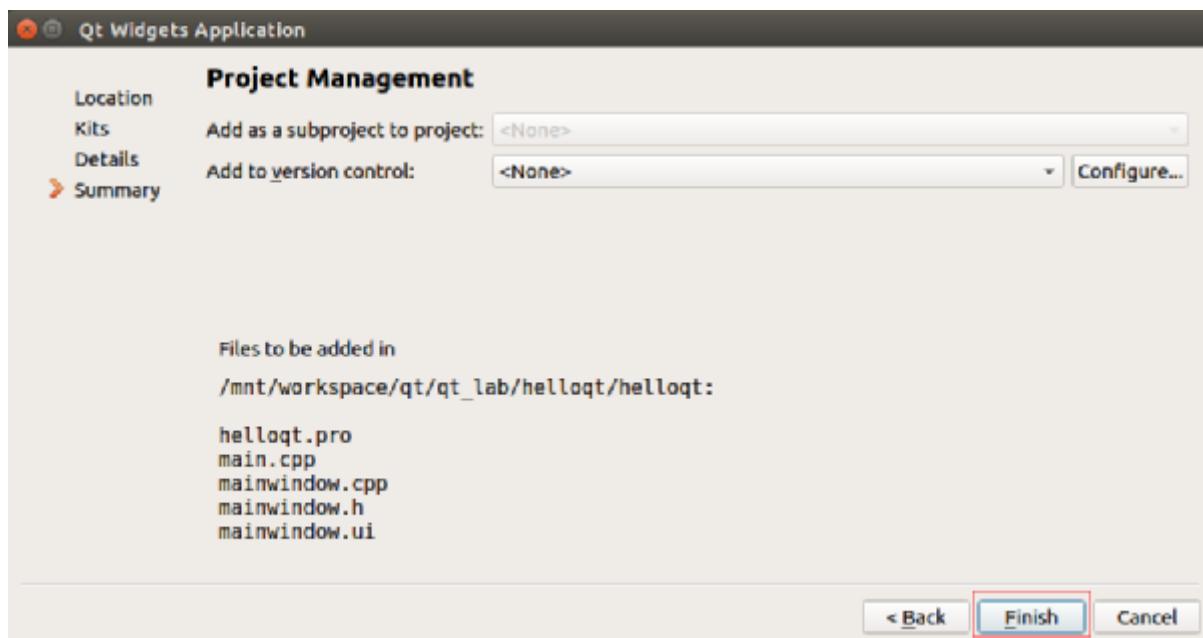
Step4:单击 NEXT



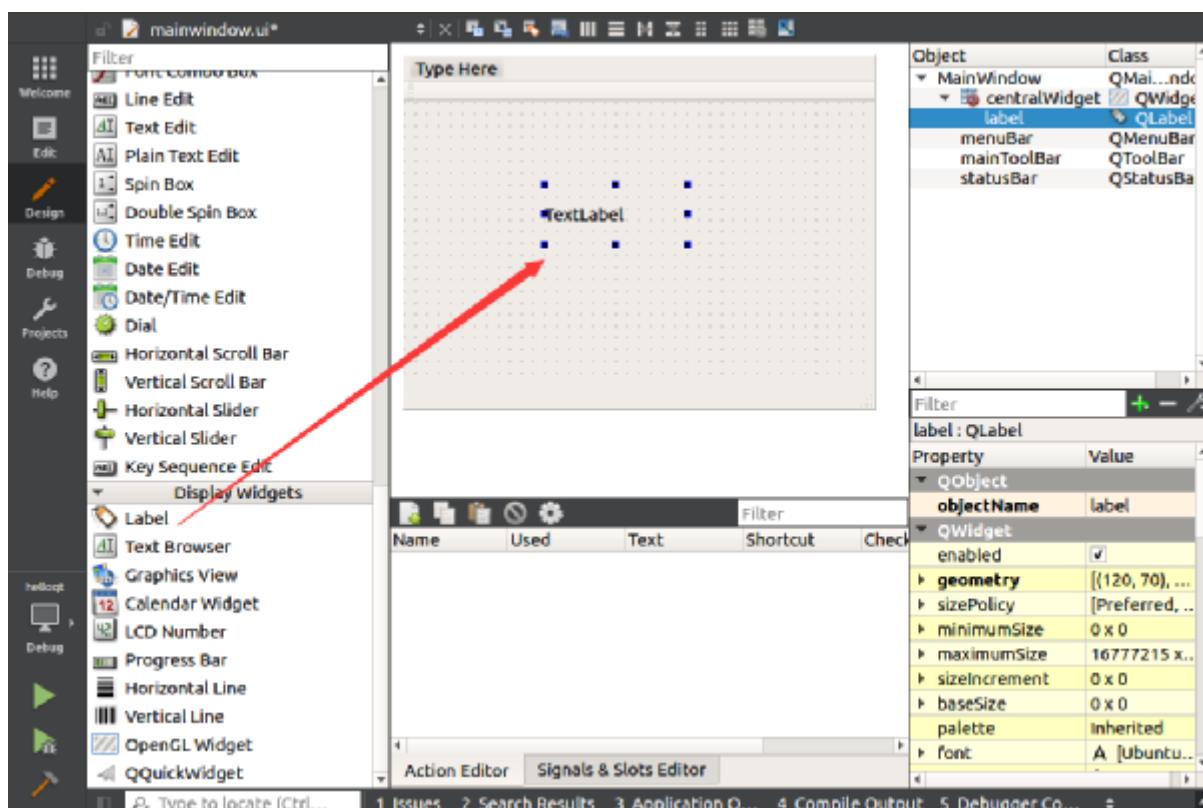
Step5:单击 NEXT



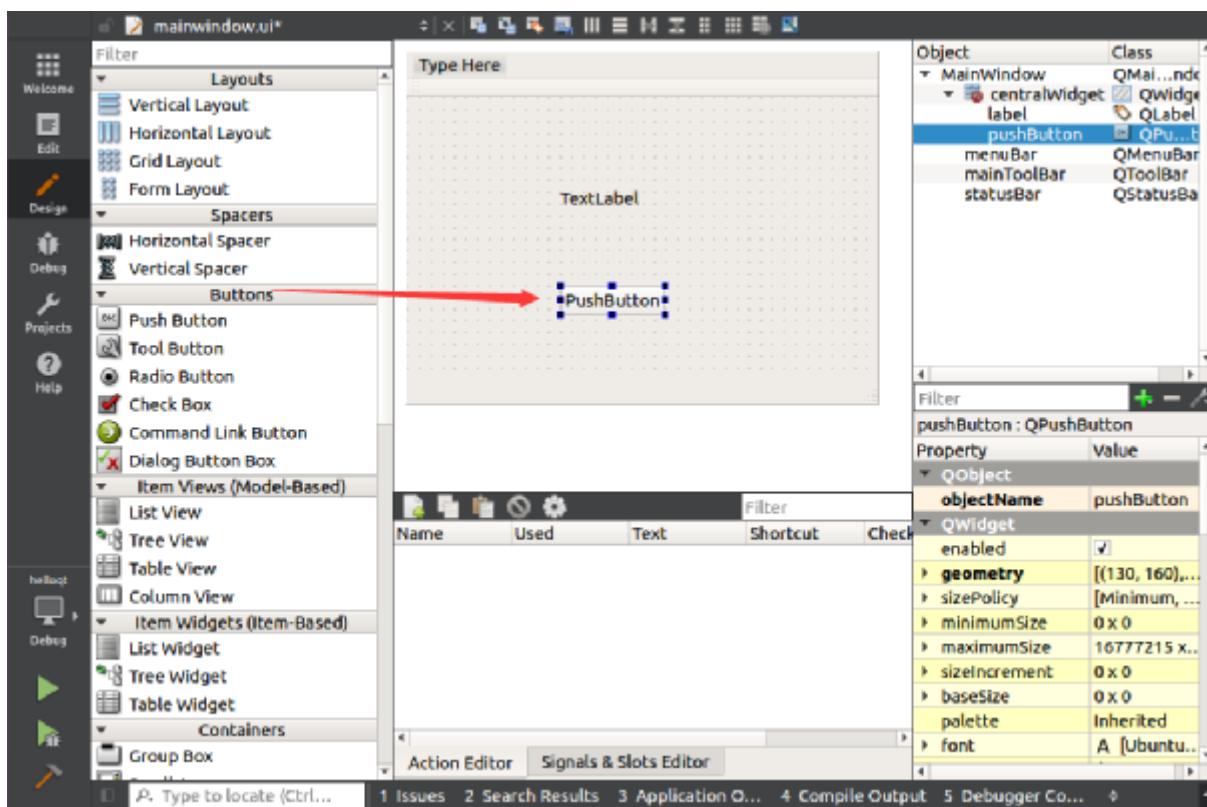
Step6:单击 Finish



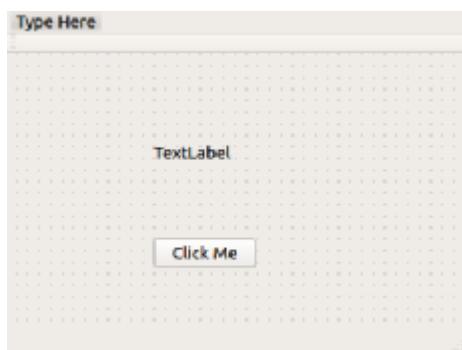
Step7:拖动 Label 标签控件到窗口



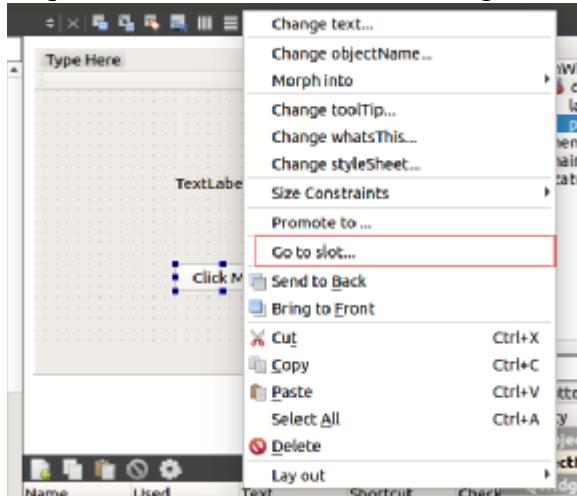
Step8:拖动 Push Button 控件到窗口



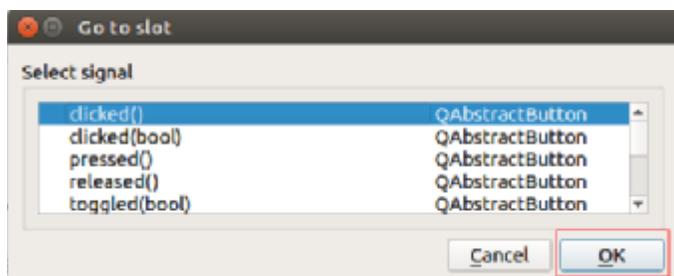
Step9:右击PushButton 控件修改控件显示文本为



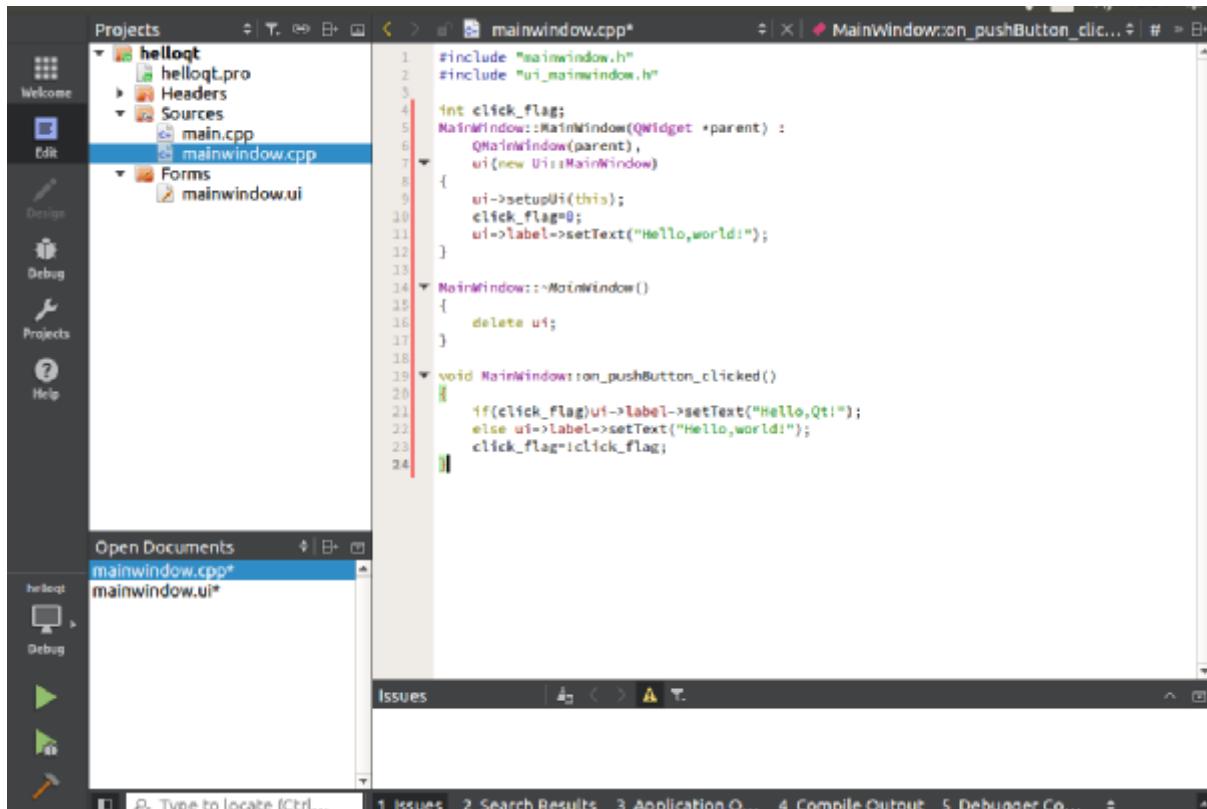
Step10:继续右击PushButton 选择 goto slot



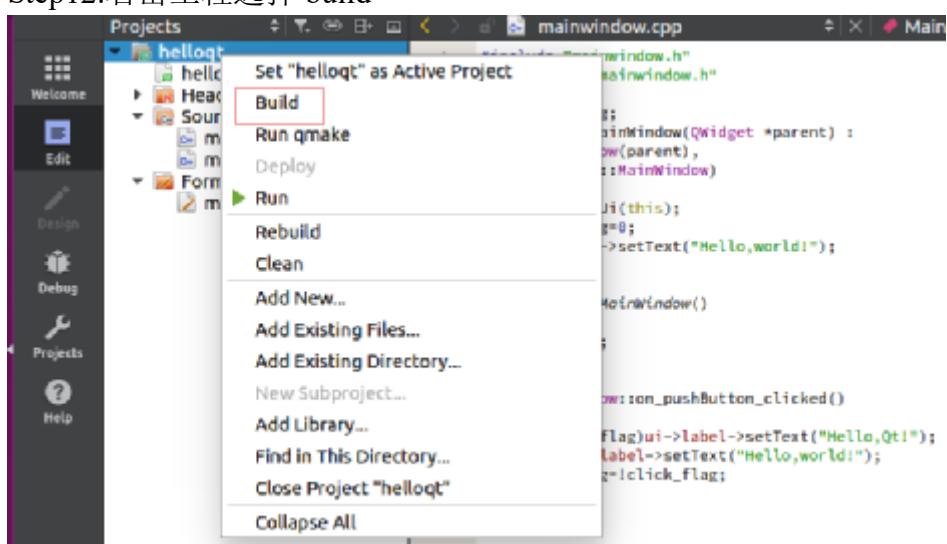
Step11:选择相应单击事件



Step11:为控件添加如下代码



Step12:右击工程选择 build



Step13: 提示错误信息

```

Compile Output | □ < > ■ + -
12:41:56: Starting: "/usr/bin/make"
g++ -Wl,-rpath,/mnt/workspace/qt/Qt5.8.0/5.8/gcc_64/lib -o helloqt main.o mainwindow.o
moc_mainwindow.o -L/mnt/workspace/qt/Qt5.8.0/5.8/gcc_64/lib -lQt5Widgets -lQt5Gui -lQt5Core -
-lGL -lpthread
/usr/bin/ld: cannot find -lGL
Makefile:236: recipe for target 'helloqt' failed
collect2: error: ld returned 1 exit status
make: *** [helloqt] Error 1
12:41:56: The process "/usr/bin/make" exited with code 2.
Error while building/deploying project helloqt (kit: Desktop Qt 5.8.0 GCC 64bit)
When executing step "Make"
12:41:56: Elapsed time: 00:00.

```

这是由于缺少必要的库导致，在控制台输入：

sudo apt-get install libgl1-mesa-dev

再次编译就可以通过

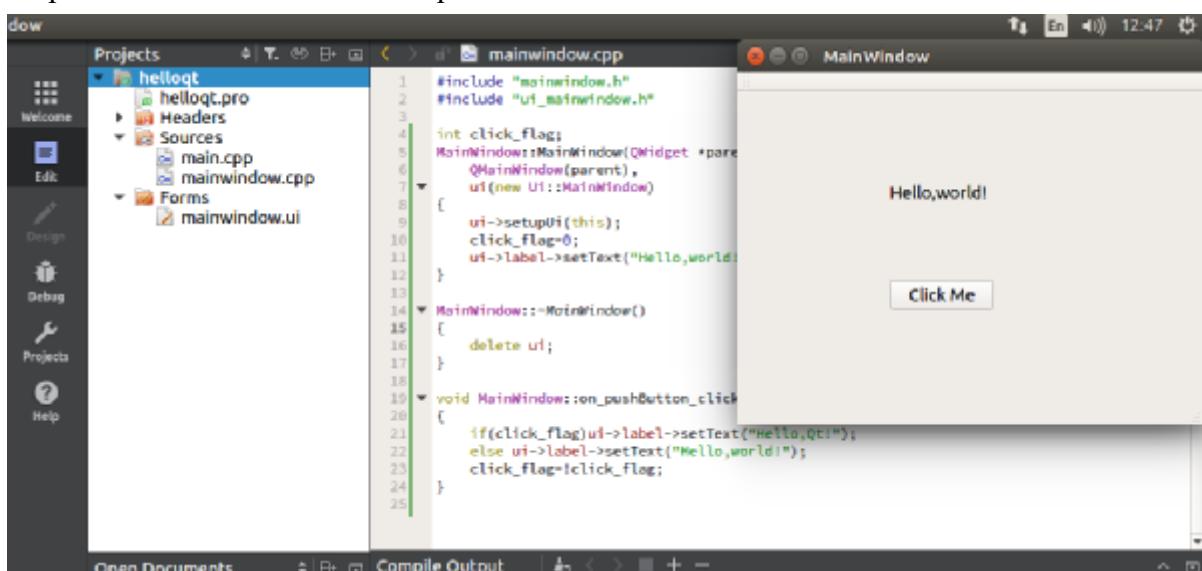
```

Compile Output | □ < > ■ + -
12:44:50: Running steps for project helloqt...
12:44:50: Configuration unchanged, skipping qmake step.
12:44:50: Starting: "/usr/bin/make"
g++ -Wl,-rpath,/mnt/workspace/qt/Qt5.8.0/5.8/gcc_64/lib -o helloqt main.o mainwindow.o
moc_mainwindow.o -L/mnt/workspace/qt/Qt5.8.0/5.8/gcc_64/lib -lQt5Widgets -lQt5Gui -lQt5Core -
-lGL -lpthread
12:44:50: The process "/usr/bin/make" exited normally.
12:44:50: Elapsed time: 00:00.

1 Issues 2 Search Results 3 Application O... 4 Compile Output 5 Debugger Co...

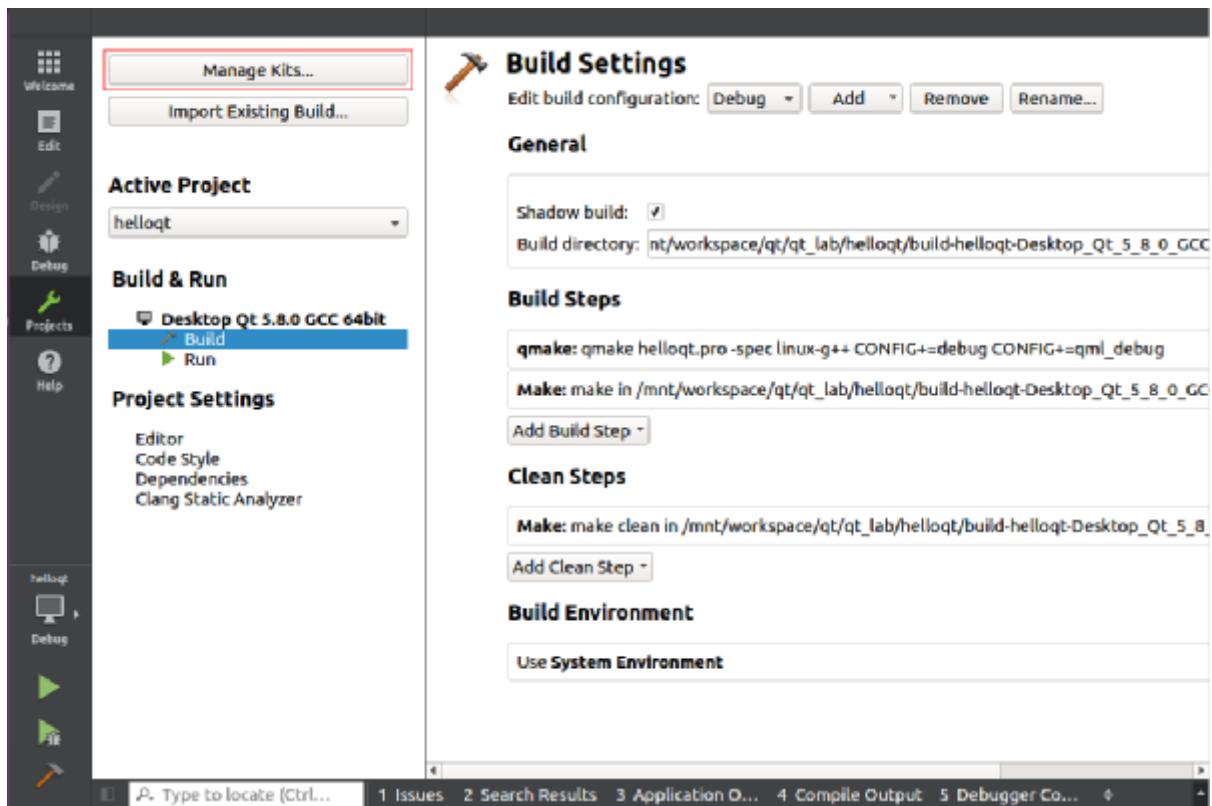
```

Step14: 右击工程选择 run 运行 qt 界面

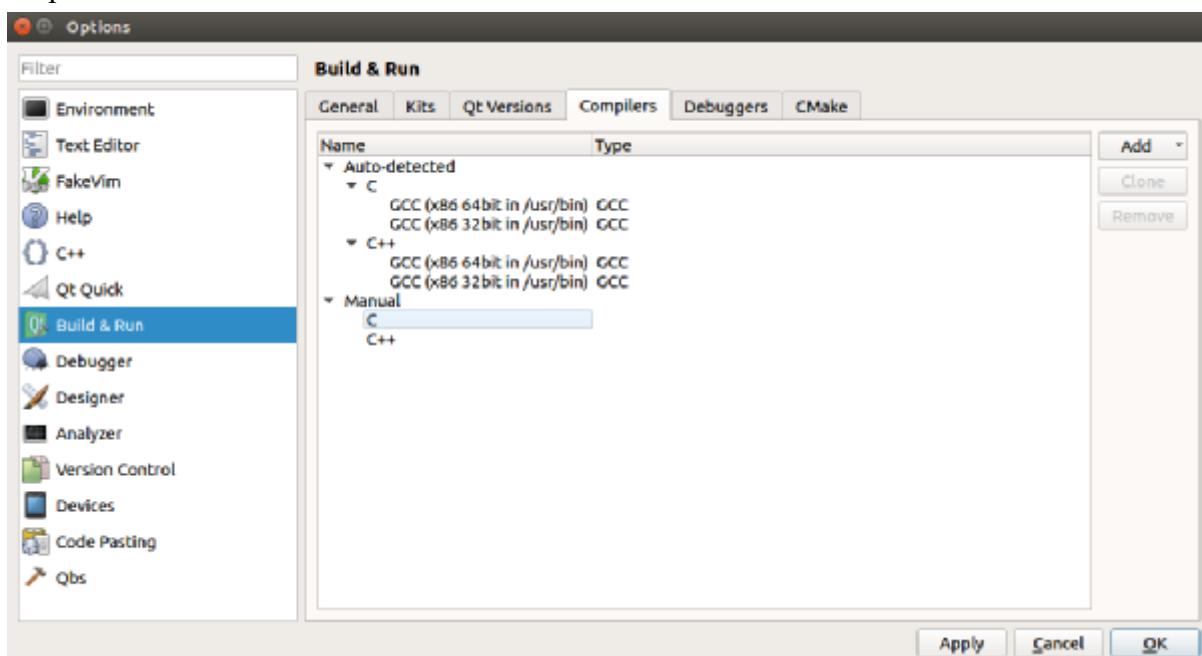


7.5 对 QtE 设置交叉编译

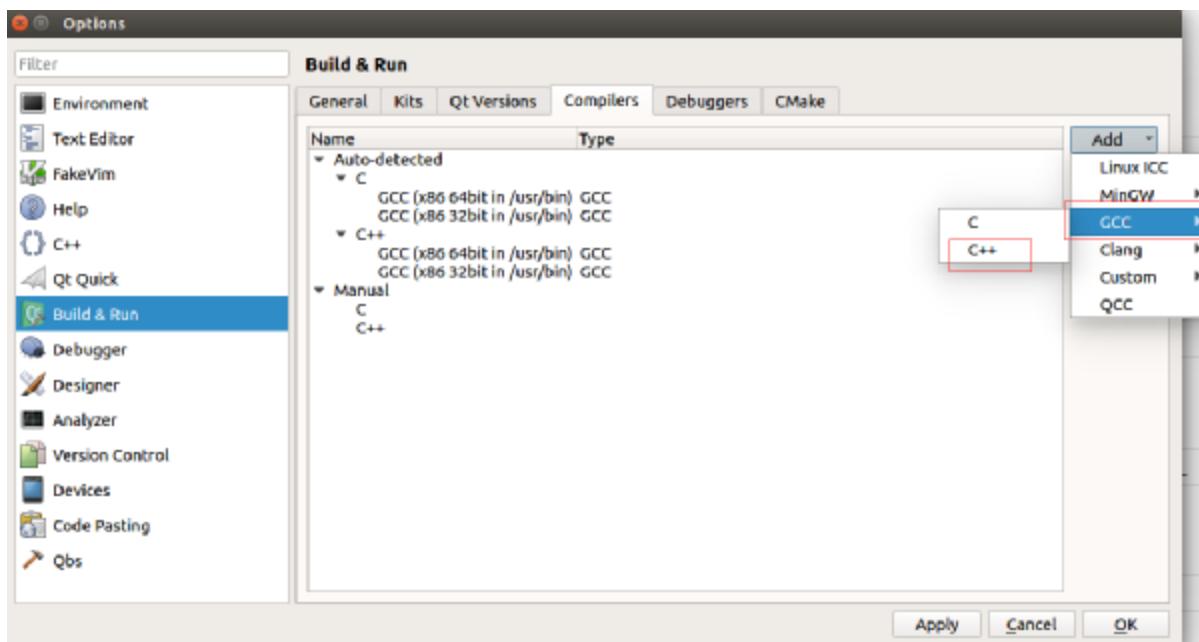
Step1: 选择 Manage Kits



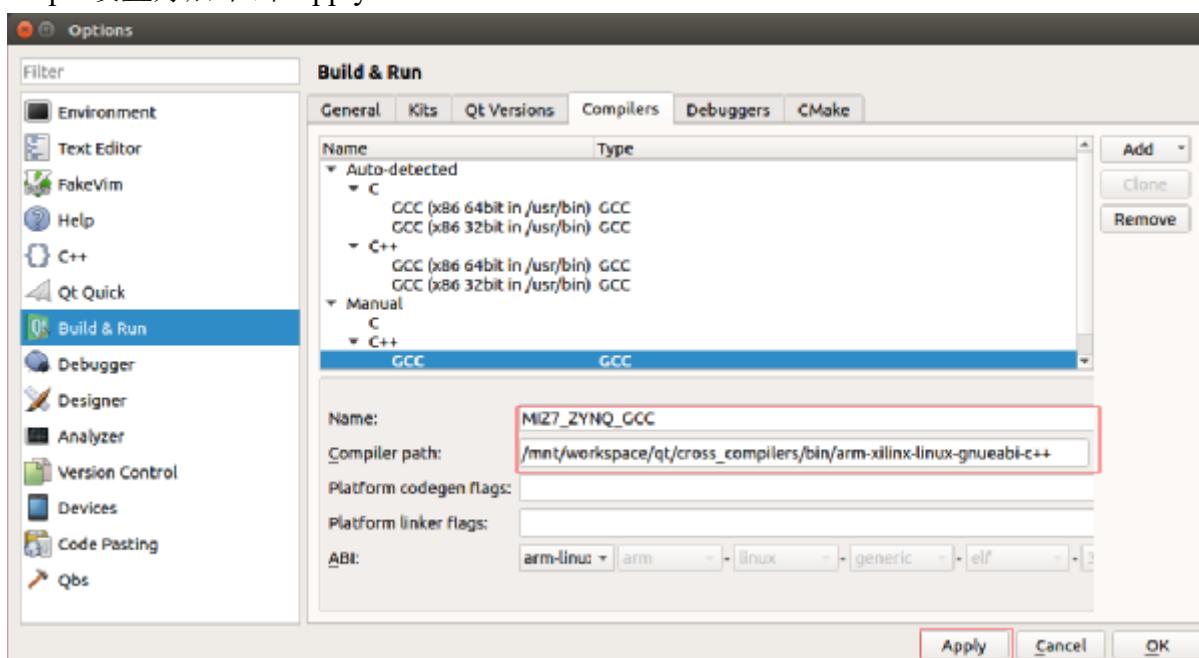
Step2:设置交叉编译器



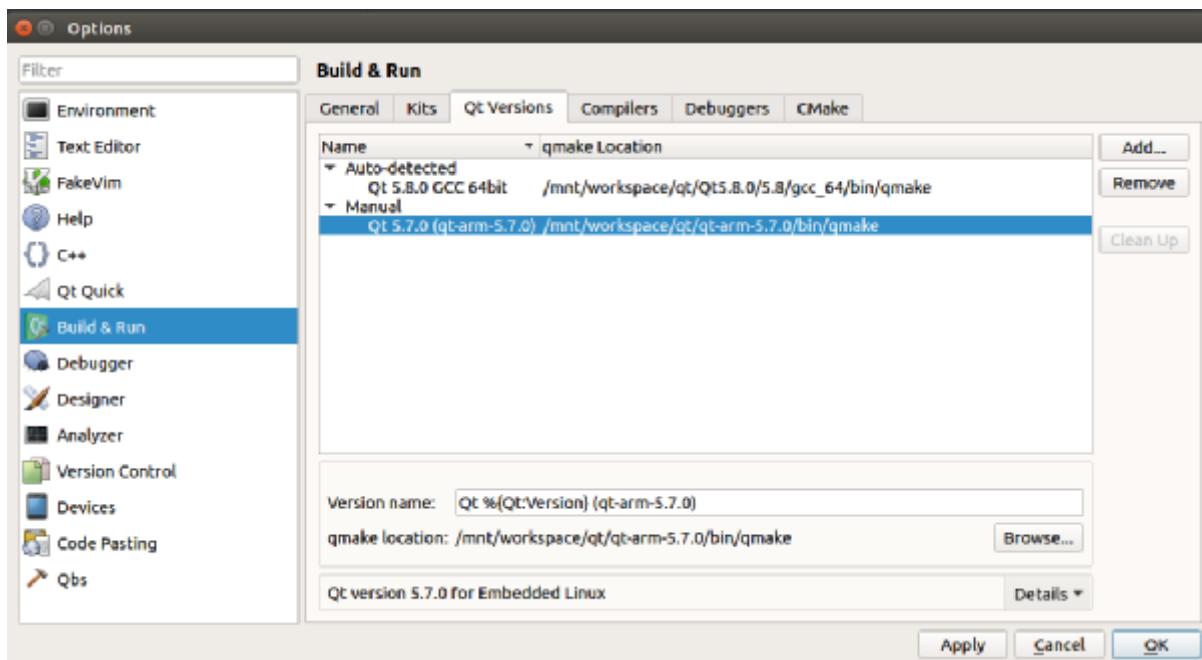
Step3:设置 C++编译器



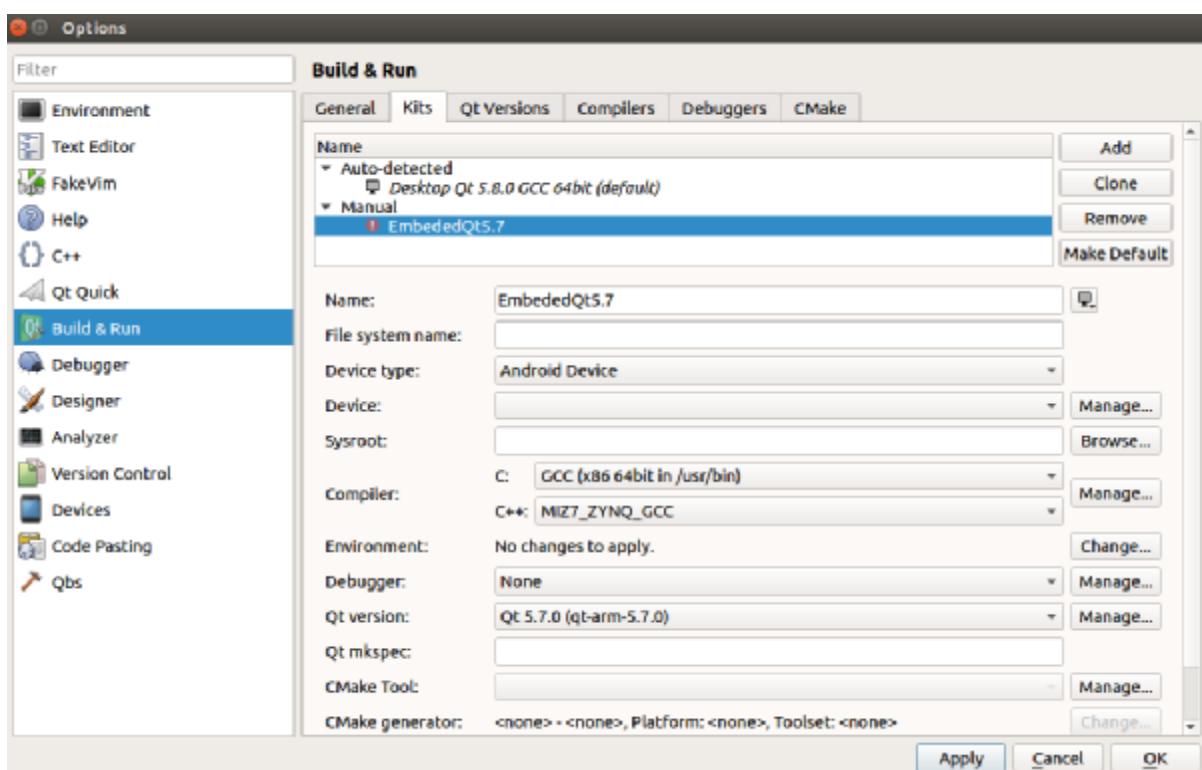
Step4:设置好后单击 Apply



Step5:设置 Qt Versions

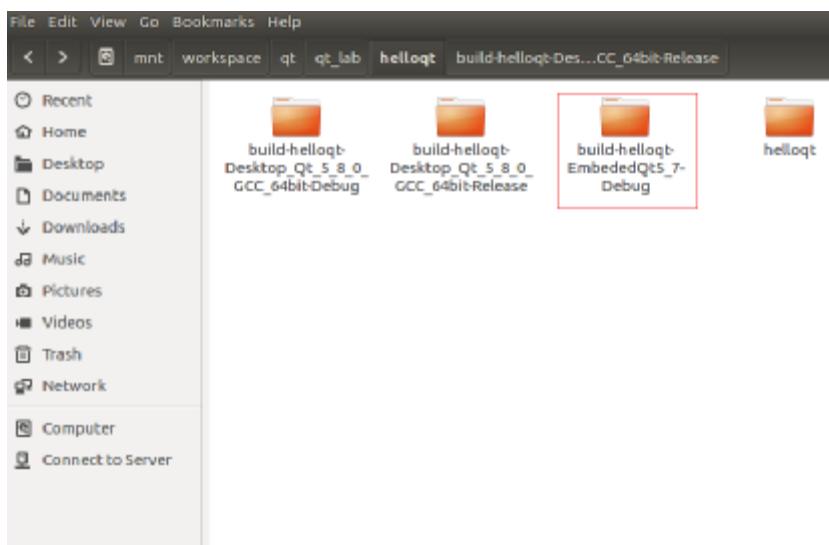


Step6:设置 Qt Versions



Step7:单击 OK

Step8:右击工程再次编译后可以看到如下文件夹下增加了基于嵌入式 LINUX 系统的可执行文件



Step8: 复制 helloqt 到 SD 卡并且修改 init.sh 文件的开机自动启动为 helloqt
Init.sh 文件修改如下

```
#!/bin/sh

echo "hello world"

export QTDIR=/opt/QT5.7
mkdir -p ${QTDIR}

mount -o loop /mnt/QT5.7.0.image ${QTDIR}

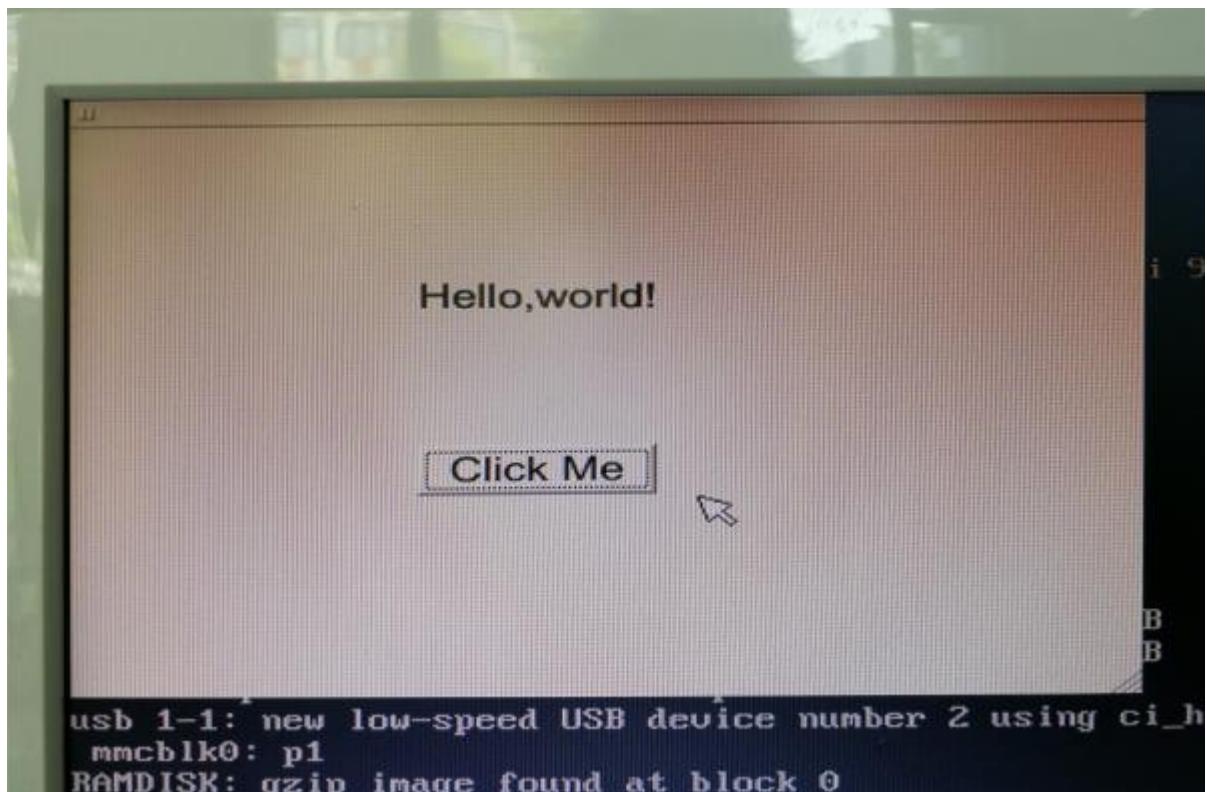
export QT_QPA_FONTDIR=${QTDIR}/lib/fonts
export QT_QPA_PLATFORM_PLUGIN_PATH=${QTDIR}/plugins/
export LD_LIBRARY_PATH=${QTDIR}/lib:$LD_LIBRARY_PATH

export QT_QPA_PLATFORM=linuxfb:fb=/dev/fb0

#http://www.360doc.com/content/14/1215/10/18578054_433033534.shtml
```

```
export QT_QPA_EVDEV_MOUSE_PARAMETERS=evdevmouse:/dev/event0  
/mnt/helloqt&
```

7.6 测试结果



一个完美的结束
意味着一个新的开始！

www.osrc.cn

米联客
技术论坛
秀出你的风采！