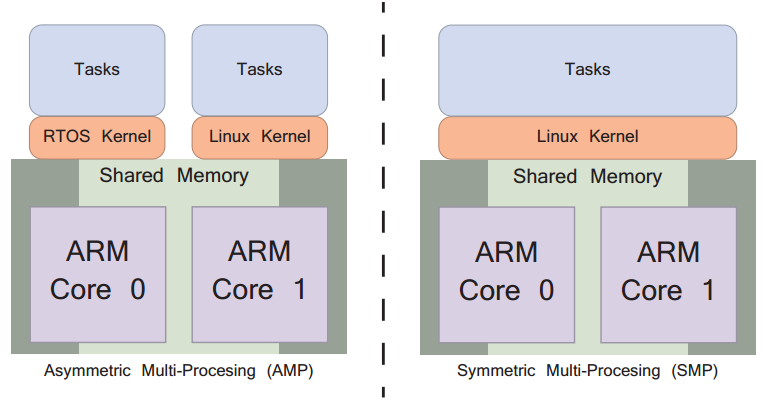
**ZYBO开发板搭建Linux&Bare-metal amp系统**

1. 什么是AMP  
    AMP全称是Asymmetric Multi-Processing，非对称多核处理，也就是一个系统有多个CPU，每个CPU独立跑不同的系统或者裸机程序。

与AMP相对的还有SMP，Asymmetric Multi-Processing，对称多核处理，也就是多个CPU跑同一个操作系统，下图可直观反映两种系统的差异



1. 系统启动过程

Power On Reset

BOOTROM initialize cpu0, load FSBL to OCM, execute FSBL

CPU1 enterWFE status

FSBL load bit file, uboot, cpu1 app to DDR from SD/Flash

FSBL handoff，CPU execute uboot

CPU1 execute a little piece of code in OCM address 0xFFFFFE00, keep WFE status and listen value in 0xFFFFFFF0

Uboot load devicetree, kernel, filesystem to DDR

Linux boot-up, write cpu1 app first address of the code segment to 0xFFFFFFF0

CPU1 aware value changed, jump to first address of the code segment, start running app

CPU0

CPU1

Time line

3. 软件处理

根据上节描述的系统启动过程，软件可以分如下3块：

(1)第一阶段boot loader(FSBL)

FSBL固定运行在CPU0上。这是系统上电以后运行的第一个用户程序。它负责FPGA程序的加载，以及加载可执行的ELF文件到内存，之后FSBL就开始运行第一个加载程序。

ISE® Design Suite 14.3里的FSBL不支持多数据或多ELF文件。FSBL先寻找一个bit file，找到以后加载到FPGA中。之后无论有没有bit file有没有找到，FSBL都会加载一个ELF文件到内存，然后执行，这种操作不支持AMP系统，目前Xilinx已经摒弃了ISE，用Vivado，它生成的FSBL可以直接使用。

满足AMP的FSBL会一直搜索文件，并加载到内存。接下来FSBL加载最后一个文件跳转到第一个非bit file、非boot file的地址去执行程序。

(2)CPU0的linux操作系统和应用程序

在AMP设置下用linux最简单的方法就是将linux设置成SMP，但是限制核心数量为1。确保在多核环境中linux可以正确设置ICD(Interrupt control distributor)和SCU(Snoop control unit)。

rwmem为linux提供片上内存(OCM)的读写访问，当特定的地址中的值变了，CPU1会做出相应的反应。

当整个ROM加载完了以后，CPU1会运行一小段代码，代码位置在OCM的0xfffffe00处。这段代码不断地检查0xFFFFFFF0地址的值是否为0，如果不是，CPU1就会跳转到获取到的地址 0x1A000000，也就是0xFFFFFFF0地址处的值，去执行ELF文件，这个文件是由FSBL写入的。

正常情况下，向0xFFFFFFF0中写入了cpu1的跳转地址后还需要在cpu0上执行一个SEV指令，但是linux系统会不停地服务中断，这些中断也能结束CPU1的WFE状态，因此linux&Bare-metal的系统中不需要额外执行SEV指令。

1. CPU1的裸机程序

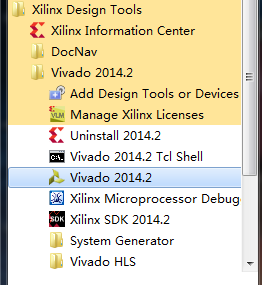
CPU1的裸机程序比较简单，只需要在link scrip中设定好加载到DDR的地址即可，其他用SDK生成的hello world程序就能进行测试

1. 实现AMP具体步骤：

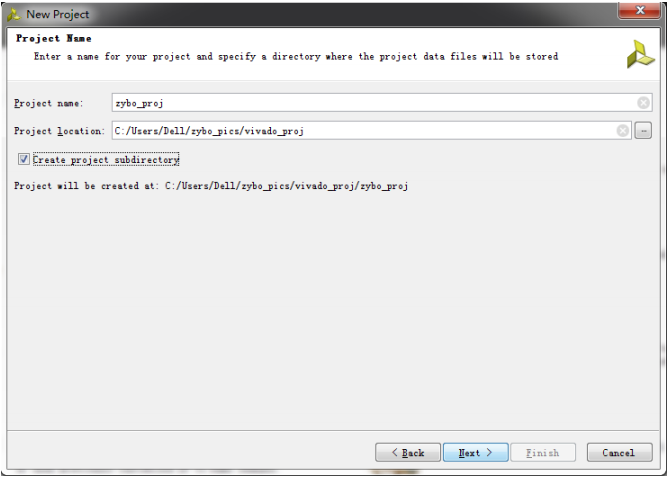
本例程基于Vivado2014.2搭建，参考Xilinx官方例程xapp1079&xapp1078

(1)搭建ZYBO硬件平台

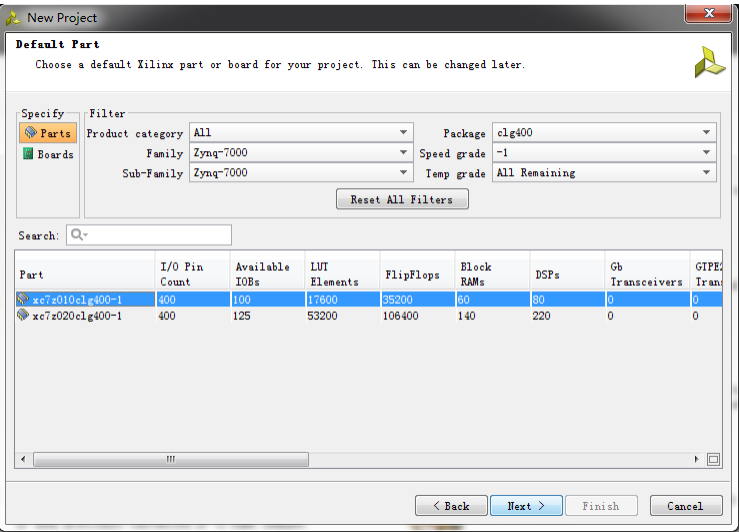
* 打开Vivado 2014.2



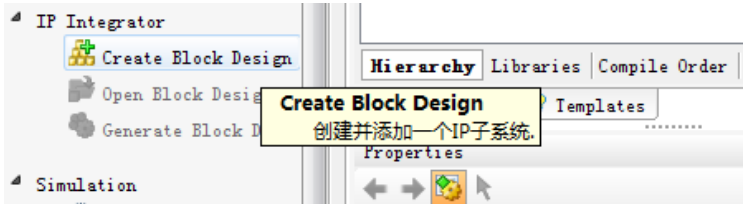
* 新建工程
* 填写工程名称、工程位置信息，选中 Create project subdirectory



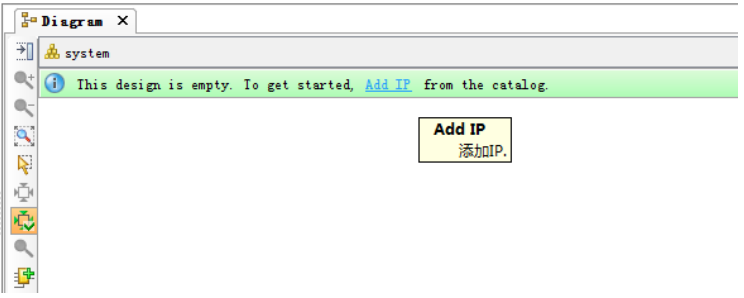
* 选中 RTL Project 和 Do not specify sources at this time
* 选择 Zynq-7000 系列， Package 是 clg400，Speed grade 是-1，选择 xc7z010clg400-1



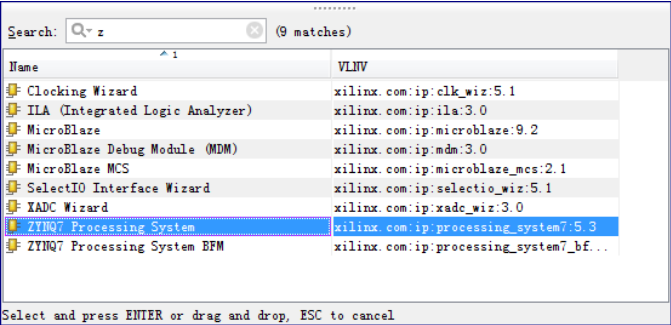
* 选择 Finish
* 选择 Create Block Design 建立子系统



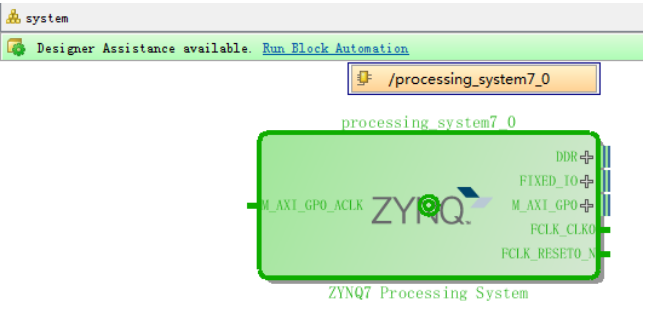
* 命名子系统名称system
* 选择 ADD IP 添加 IP



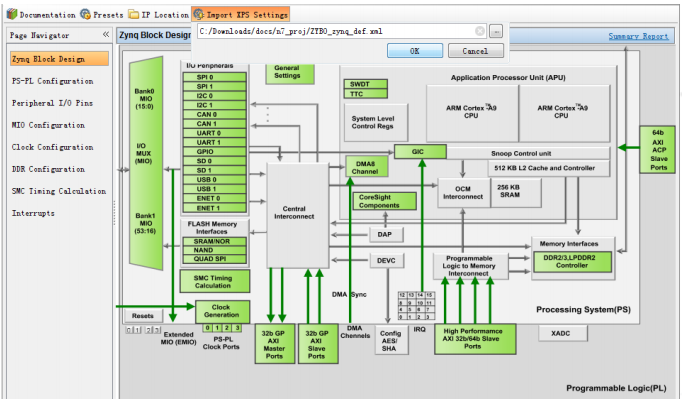
* 在 Search 中写“ z”，双击 ZYNQ7 Processing System



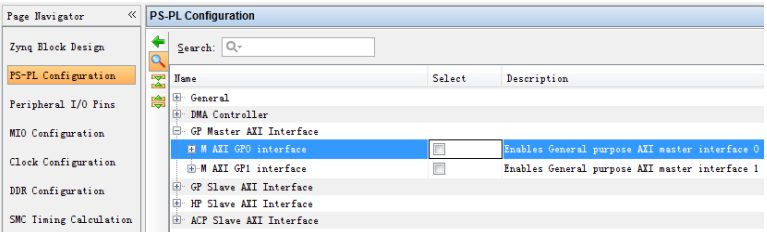
* 使 用 Run Block Automation 的/processing\_system7\_0，点 OK



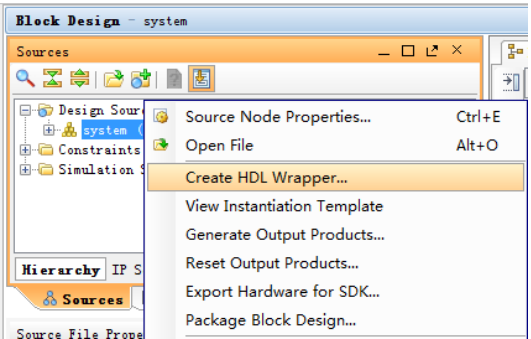
* 双击 processing\_system7\_0框，在跳出的窗口中点 击 Import XPS Settings ， 找 到 并 选择ZYBO\_zynq\_def.xml，点OK



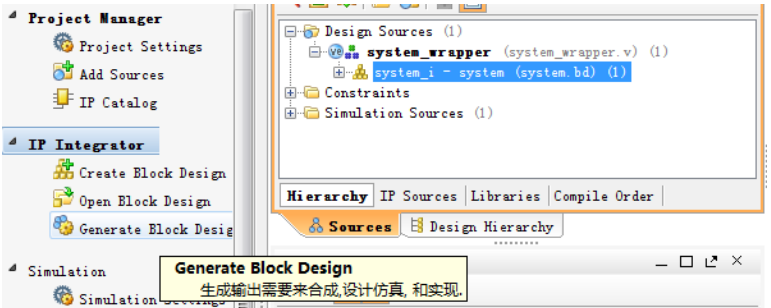
* 进入PS-PL Configuration，取消M\_AXI\_GP0选择，保存设置修改



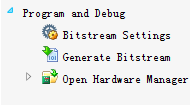
* File->Save Block Design
* 在 system 上点右键，选 Create HDL Wrapper，选择 Let Vivado manage wrapper and autoupdate



* 使用 Generate Block Design



* 点击Generate Bitstream生成FPGA程序



* File->Export->Export Hardware for SDK到处HDF文件，后面SDK创建工程时要用

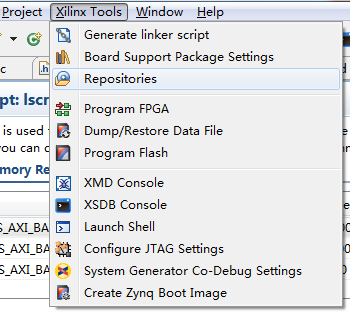
1. 软件实现步骤

(1)配置SDK环境

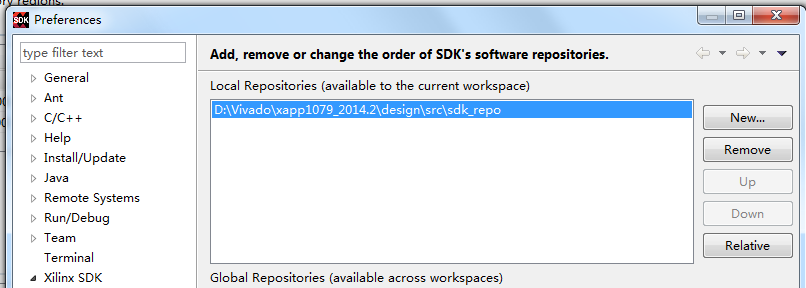
从http://www.wiki.xilinx.com/XAPP1079+Latest+Information?responseToken=a1285b1b8b831f9ffa562a3d2b27da34

下载earlyaccess\_xapp1079\_2014.2.zip，解压，得到design和docs两个文件夹

* 打开SDK，设置Repositories



* 指定design中的Repositories路径，里面包含裸机程序使用的各种库，和Vivado自带的相比，指定的Repositories修改了boot.S 和asm\_vectors.S，专门为启动cpu1而修改的

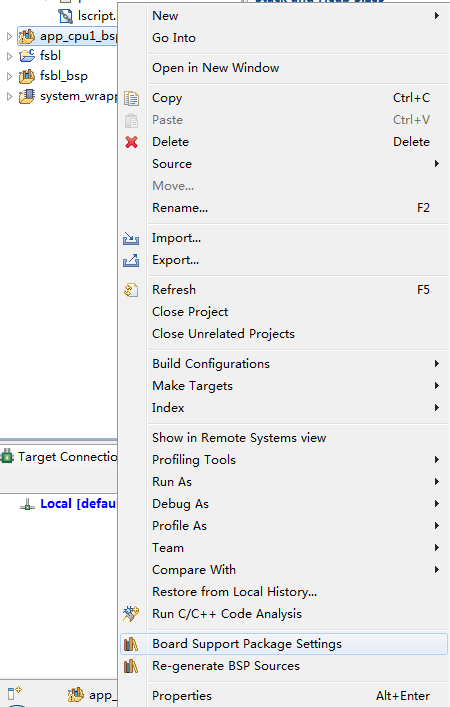


1. 创建FSBL工程

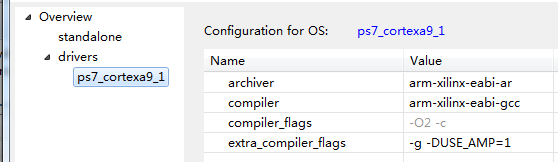
* File->New->Application\_Project
* 输入工程名'fsbl' 点击Next
* 选择'Zynq FSBL' 点击Finish

1. 创建CPU1的程序

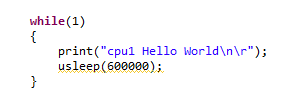
* File->New->Board\_Support\_Package
* 输入工程名 'app\_cpu1\_bsp', 修改CPU to ps7\_cortexa9\_1，点Finish
* 右键app\_cpu1\_bsp点击'Board Support Package Settings



* 选择Overview->drivers->ps7\_cortexa9\_1，修改编译选项 '-g -DUSE\_AMP=1'



* File->New->Application\_Project
* 输入工程名'app\_cpu1'
* 修改CPU，ps7\_cortexa9\_1
* 修改'Board Support Package' 为'Use existing' 'app\_cpu1\_bsp' 点Next
* 选'Hello World' 点Finish。
* 修改代码为while循环，每500ms打印一次cpu1 Hello World



* 修改ld.scrip：ps7\_ddr\_0\_S\_AXI\_BASEADDR的值改为0x1A000000，size为0x5FFFFFF

1. 编译uboot

* 从git导uboot源码git clone git://github.com/Xilinx/u-boot-xlnx.git
* u-boot-xlnx/include/configs可以看到有zynq\_zybo.h头文件，因此通过

make zynq\_zybo\_config

配置uboot，在配置之前要先修改内存大小，ZYBO有512M的DDR，两个CPU需要划分各自的内存空间，目前暂定cpu0用416M，地址从0x0-0x1A000000，cpu1用96M，地址从0x1A000000-0x1FFFFFFF，在头文件zynq\_zybo.h中修改CONFIG\_SYS\_SDRAM\_SIZE即可，如果实际运行没有生效就需要修改uboot设备树，路径为u-boot-xlnx/arch/arm/dts/zynq-zybo.dts，将地址范围从reg = <0x0 0x20000000>;改为reg = <0x0 0x1A000000>;，编译的时候uboot会自动将设备树编译进二进制文件。

* 编译make ARCH=arm CROSS\_COMPILE=arm-xilinx-linux-gnueabi- -j8，然后将u-boot重命名为u-boot.elf
* 编译报错处理：

1.fatal error: openssl/evp.h:

安装openssl：sudo apt-get install libssl-dev

2./bin/sh: 1: dtc: not found

sudo apt-get install device-tree-compiler

1. 编译内核

* 从git导内核源码git clone -b master --single-branch <https://github.com/Xilinx/linux-xlnx.git>
* /linux-xlnx/arch/arm/configs下可以看到xilinx\_zynq\_defconfig.h，因此通过

make ARCH=arm xilinx\_zynq\_defconfig

配置内核

* 编译内核make ARCH=arm UIMAGE\_LOADADDR=0x8000 uImage -j8

1. 生成设备树

设备树文件在目录linux-xlnx/arch/arm/boot/dts中，对应ZYBO开发板有zynq-zybo.dts，内核自带的这个设备树实测起不来的，要用zybo光盘中的设备树，光盘中是已经编译成二进制的dtb文件，实际我们还需要修改内存地址范围，所以要将它转为dts文件，用如下命令：

./linux-xlnx/scripts/dtc/dtc -I dtb -O dts -o zynq-zybo.dts devicetree.dtb

然后将生成的zynq-zybo.dts拷贝至内核的dts目录。

然后需要修改设备树中内存的地址范围，和uboot的设备树一样从reg = <0x0 0x20000000>;改为reg = <0x0 0x1A000000>；最后就是生成dtb文件了。

对于dtb生成我写了一个脚本mkdtb.sh 供参考，脚本放在内核源码同级目录：

#!/bin/bash

HOME=$(pwd)

KERNEL\_SRC=linux-xlnx

DTS\_NAME=zynq-zybo.dts

cd $KERNEL\_SRC

./scripts/dtc/dtc -I dts -O dtb -o $HOME/devicetree.dtb arch/arm/boot/dts/$DTS\_NAME

执行此脚本后，会在同级目录生成devicetree.dtb

1. linux文件系统

按第3小节描述的原理，cpu0需要用rwmem.elf来向0xFFFFFFF0地址写东西，因此要把它放进文件系统，我目前把它放在/usr目录下，然后需要在Ubuntu中下载一个专门生成文件系统的工具genext2fs-1.4.1，然后运行我写的脚本mkramfs.sh ：

genext2fs -b 15360 -d rootfs/ ramdisk.image

if [ $? -ne 0 ]; then

echo "general ramdisk.image failed! make sure your PC has installed genext2fs and check the param of genext2fs"

exit 1

fi

echo "general ramdisk.image.gz"

gzip -9 ramdisk.image

if [ $? -ne 0 ]; then

echo "general ramdisk.image.gz failed!"

exit 1

else

echo "add file head"

mkimage -A arm -T ramdisk -C gzip -n Ramdisk -d ramdisk.image.gz uramdisk.image.gz

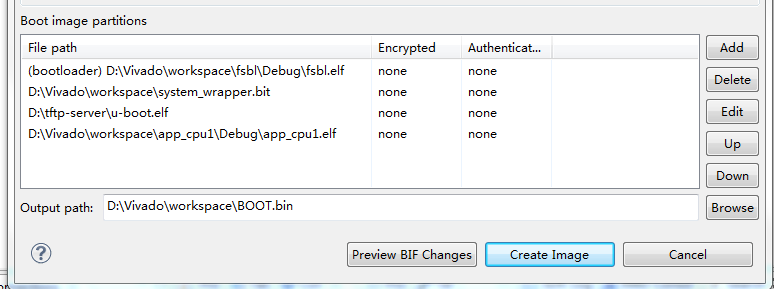
rm ramdisk.image.gz

fi

之后会在同级目录下生成uramdisk.image.gz。

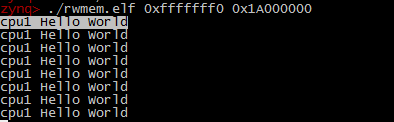
(8)BOOT.bin生成

* 打开SDK
* Xilinx Tools->Create Zynq Boot Image
* 选择Create new BIF file，指定存放路径
* 在Boot image partition中指定fsbl.elf,.bit文件,uboot.elf和app\_cpu1.elf的路径，点Create Image



(9)启动开发板

* 将BOOT.bin、uImage、devicetree.dtb、和文件系统uramdisk.image.gz拷贝至SD卡，连好串口即可，串口波特率115200
* 增加uboot参数：set bootargs 'console=ttyPS0,115200 maxcpus=1 root=/dev/ram rw earlyprintk'
* Cd到usr目录，输入./rwmem.elf 0xfffffff0 0x1A000000，就能看到串口一直会刷cpu1 Hello World



**Zedboard开发板搭建Linux&freeRTOS amp系统**

1. freeRTOS

FreeRTOS是一个迷你的实时操作系统内核。作为一个轻量级的操作系统，功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能、软件定时器、协程等，可基本满足较小系统的需要。

源码可以到这里下载：<https://sourceforge.net/projects/freertos/files/FreeRTOS/，本例程用的是v8.1.2版本。>

先来看看目录结构：

其中freeRTOS为主目录，里面包含内核和平台相关文件和示例程序；freeRTOS-Plus是一些扩展功能，包括命令行、网络等，他们分别有Demo文件夹，对应各种平台、各种功能的示例程序。

FreeRTOSV8.1.2

FreeRTOS

FreeRTOS-Plus

Demo

License

Source

include

portable

Demo

Source

freeRTOS内核相关的文件仅为6个：list.c queue.c tasks.c croutine.c event\_groups.c timers.c，位于FreeRTOS/Source

平台相关文件位于FreeRTOS/Source/portable，这下面还有很多文件夹，先以编译器为大类，然后再以平台（MCU）为小类，zynq用到的3个文件位于FreeRTOS/Source/portable/GCC/ARM\_CA9。

内存相关文件有5个：heap\_1.c,heap\_2.c,heap\_3.c,heap\_4.c,heap\_5.c，只要选其中一个就行，本例程中选的是heap\_4.c，位于FreeRTOS/Source/portable/MemMang

以上这些东西基本上就构成了一个最小的freeRTOS系统。

和linux和nuttx不同的是，freeRTOS内核本身没有main函数，也就是说freeRTOS的源码只是一些api接口罢了，main函数需要我们自己的应用程序来实现，而且也有固定的套路：

int main(void)

{

必要的初始化工作;

创建任务1;

创建任务2;

...

vTaskStartScheduler(); /\*启动调度器\*/

while(1);

}

其中最关键的就是vTaskStartScheduler函数了，它用于启动调度器，调度器是freeRTOS的核心，调用了它也就相当于是freeRTOS跑起来了。

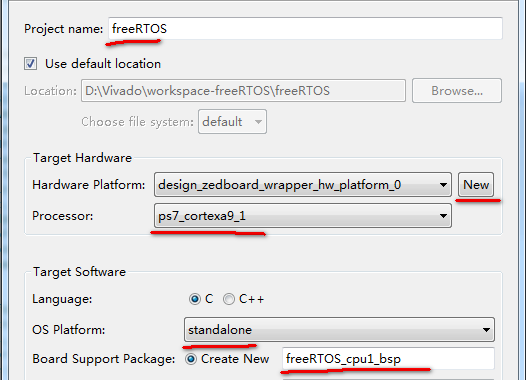
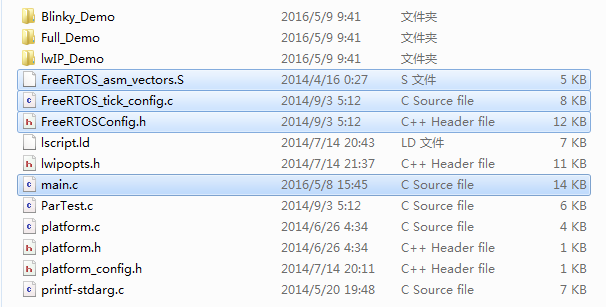
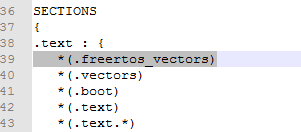
1. Amp系统搭建

官方文档中linux&freeRTOS的amp系统是通过petalinux搭建的，但是那是3年前的文档，当时的petalinux内建了freeRTOS，但是现在已经没了。实际上，由于freeRTOS并没有main函数，只是一系列API接口而已，我们可以参考之前linux&Bare-metal的搭建方式，把freeRTOS当成裸机程序的库就行了。具体步骤如下：

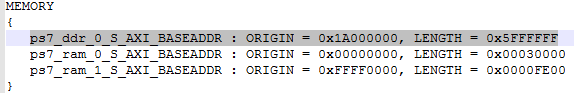
1. 生成hdf文件

按照此网址的教程即可http://zedboard.org/zh-hant/node/1454

1. 创建freeRTOS工程

* 下载freeRTOS源码，解压，路径为<kernel path>
* 点击File->New->Application Product
* Hardware Platform选择(1)中到处的hdf文件，并填好其余选项，点next  
  
* 选择Empty Application，点Finish
* 在src中创建freeRTOS\_Source目录，用于存放freeRTOS内核代码
* 拷贝<kernel path>/FreeRTOS/Source下所有内容到workspace/src/freeRTOS\_Source，portable中只保留GCC/ARM\_CA9和MemMang下的heap\_4.c，其他都删掉
* 进入<kernel path>/FreeRTOS/Demo/CORTEX\_A9\_Zynq\_ZC702/RTOSDemo/src，拷贝下图高亮文件至workspace/src  
  
* 以记事本方式打开ld.scrip，增加如下高亮行  
  

修改如下高亮行，fsbl会把freeRTOS加载到内存的1A000000地址



* 修改main.c：  
  删除/\* Standard demo includes. \*/

#include "partest.h"

#include "TimerDemo.h"

#include "QueueOverwrite.h"

#include "EventGroupsDemo.h"

/\* Xilinx includes. \*/

#include "platform.h"

删除#if ( mainSELECTED\_APPLICATION == 0 )

extern void main\_blinky( void );

#elif ( mainSELECTED\_APPLICATION == 1 )

extern void main\_full( void );

#elif ( mainSELECTED\_APPLICATION == 2 )

extern void main\_lwIP( void );

#else

#error Invalid mainSELECTED\_APPLICATION setting. See the comments at the top of this file and above the mainSELECTED\_APPLICATION definition.

#endif /\* #if mainCREATE\_SIMPLE\_BLINKY\_DEMO\_ONLY == 1 \*/

增加函数

static void prvPrintTestTask( void \*pvParameters )

{

while(1)

{

xil\_printf( "Hello from Freertos\r\n" );

vTaskDelay( 1000 / portTICK\_RATE\_MS );

}

}

修改main函数

int main( void )

{

/\* Configure the hardware ready to run the demo. \*/

prvSetupHardware();

xTaskCreate( prvPrintTestTask, ( const char \* ) "printTest", configMINIMAL\_STACK\_SIZE, NULL, tskIDLE\_PRIORITY, NULL );

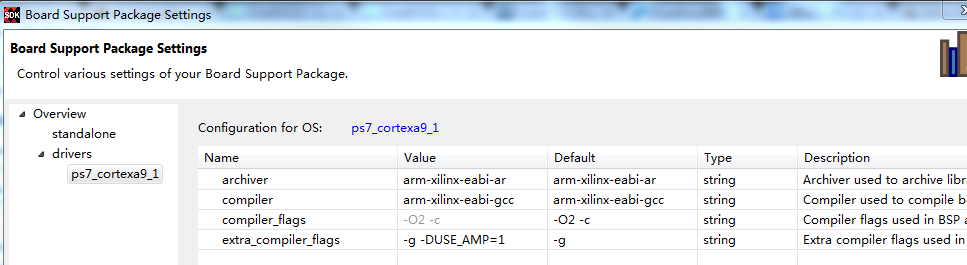
vTaskStartScheduler();

/\* Don't expect to reach here. \*/

return 0;

}

删除prvSetupHardware中的vParTestInitialise

* 修改bsp设置，增加编译选项：-DUSE\_AMP=1  
  
* 直接编译即可

1. 生成FSBL

* File->New->Application\_Project
* 输入工程名'fsbl' 点击Next
* 选择'Zynq FSBL' 点击Finish

(4)编译uboot

* 从git导uboot源码git clone git://github.com/Xilinx/u-boot-xlnx.git
* 修改u-boot-xlnx/include/configs/zynq\_zed.h头文件：

配置内存大小，Zedboard有512M的DDR，两个CPU需要划分各自的内存空间，目前 暂定cpu0用416M，地址从0x0-0x1A000000，cpu1用96M，地址从 0x1A000000-0x1FFFFFFF，在头文件zynq\_zed.h中修改CONFIG\_SYS\_SDRAM\_SIZE为 416\*1024\*1024。输入命令make zynq\_zed\_config使之生效

* 修改uboot设备树:

路径为u-boot-xlnx/arch/arm/dts/zynq-zed.dts，将地址范围从reg = <0x0 0x20000000>; 改为reg = <0x0 0x1A000000>;，编译的时候uboot会自动将设备树编译进二进制文件。

* 编译 make ARCH=arm CROSS\_COMPILE=arm-xilinx-linux-gnueabi- -j8，然后将u-boot重命名为u-boot.elf

编译报错处理：

1.fatal error: openssl/evp.h:

安装openssl：sudo apt-get install libssl-dev

2./bin/sh: 1: dtc: not found

sudo apt-get install device-tree-compiler

(5)编译内核

* 从git导内核源码git clone -b master --single-branch <https://github.com/Xilinx/linux-xlnx.git>
* /linux-xlnx/arch/arm/configs下可以看到xilinx\_zynq\_defconfig.h，因此通过

make ARCH=arm xilinx\_zynq\_defconfig

配置内核

* 编译内核make ARCH=arm UIMAGE\_LOADADDR=0x8000 uImage -j8

(6)生成设备树  
 设备树文件在目录linux-xlnx/arch/arm/boot/dts中，对应Zedboard开发板有zynq-zed.dts，

需要修改设备树中内存的地址范围，和uboot的设备树一样从reg = <0x0 0x20000000>;改为reg = <0x0 0x1A000000>；最后就是生成dtb文件了。

对于dtb生成我写了一个脚本mkdtb.sh 供参考，脚本放在内核源码同级目录，执行此脚本后，会在同级目录生成devicetree.dtb

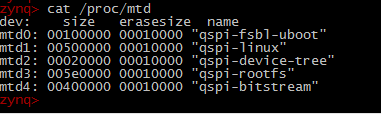
1. 文件系统、boot.bin、启动卡发布

这些步骤和ZYBO开发板搭建Linux&Bare-metal amp系统一模一样，直接参考第2节的第(7)小节开始的内容即可

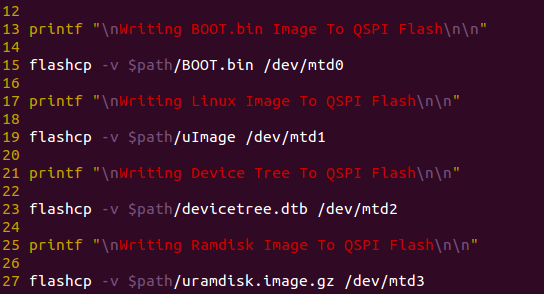
1. 从flash启动

官网上下载的文件系统根目录里已经包含了一个在linux系统下烧录flash的脚本，但是需要修改：

先在linux下输入命令cat /proc/mtd查看分区情况：

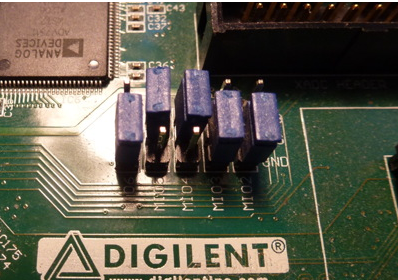


然后根据实际分区对应的设备节点修改脚本：

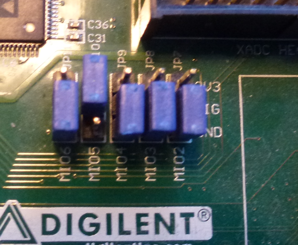


修改完后重新运行脚本mkramfs.sh生成文件系统

接下来先从sd卡启动，对于Zedboard，跳线如下图：



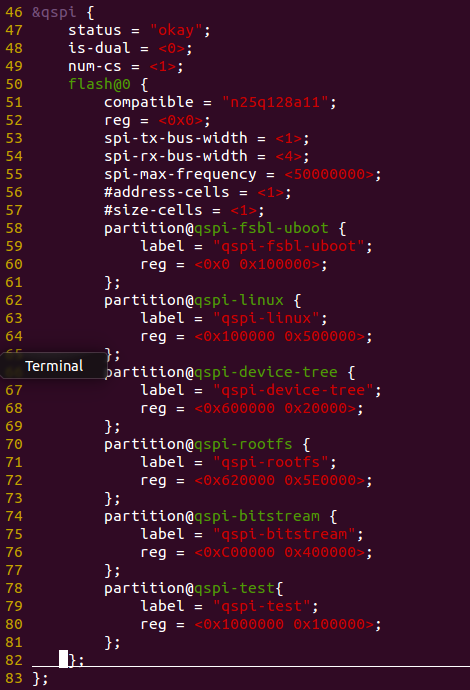
开发上电启动后运行升级flash的脚本：./update\_qspi.sh /mnt (/mnt是sd卡的挂载目录)，然后断电，重新跳线如下图：



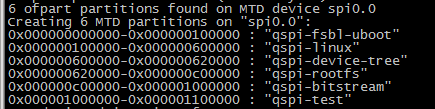
重新上电就是从flash启动了

1. 修改flash分区

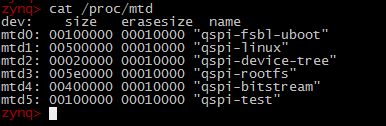
Zynq的flash分区不是由uboot传递也不是写死在内核中，而是写死在设备树中，如果要加一个分区直接修改设备树即可：



如图，qspi-test就是自己加的分区，生成dtb以后放到板子上就可以看到启动打印中出现了自己加的分区：



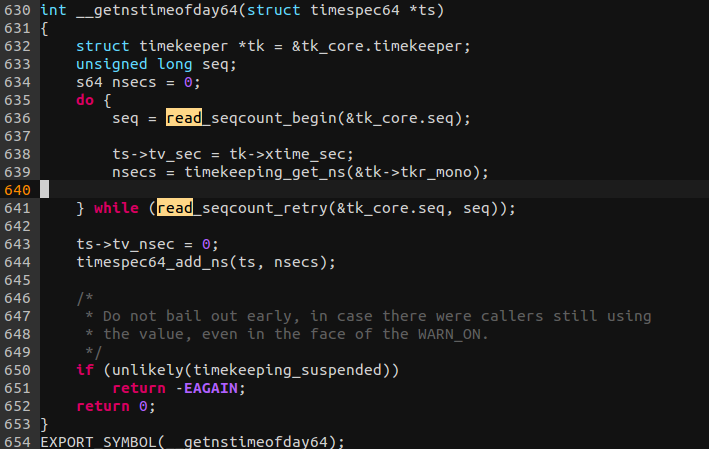
在proc中也能看到：

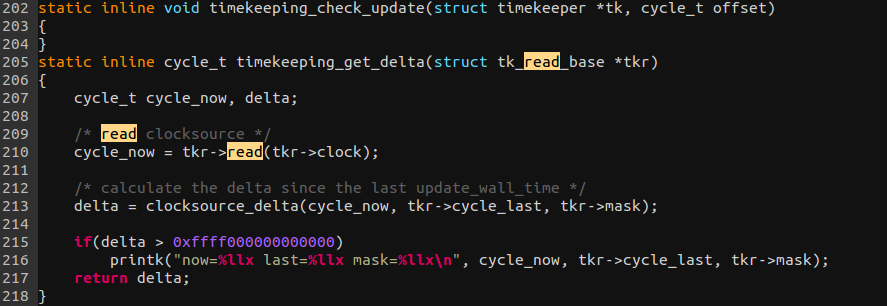


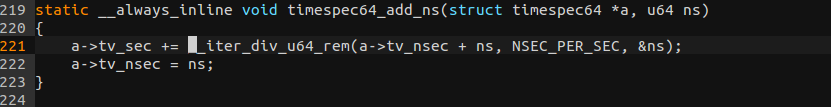
1. Linux&freeRTOS的amp系统搭建遇到的问题
2. ls命令卡住

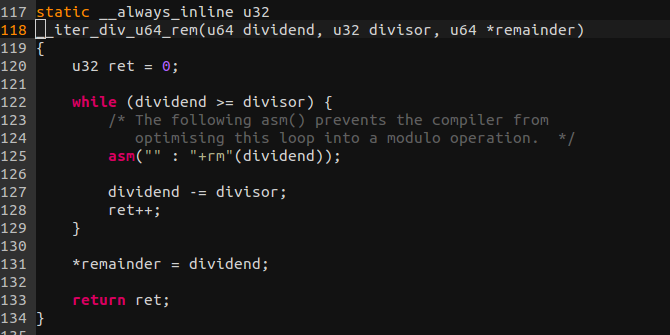
AMP系统搭建完以后，发现当cpu1启动后，在linux的shell下输入ls命令会卡住，连上jtag，打开SDK软件，在xmd中输入：connect arm hw -debugdevice，连接cpu0，再输入rrd查看cpu寄存器，重复多次发现pc指针在一小段地址中徘徊，然后输入arm-xilinx-linux-gnueabi-objdump -D vmlinux查看内核反汇编，发现pc指针卡在函数\_\_getnstimeofday64中，然后查看内核源代码，由于函数中调用了众多inline函数，直接通过反汇编推到难度较大，因此直接加打印，发现pc指针卡在函数timespec64\_add\_ns中，该函数调用了\_\_iter\_div\_u64\_rem，里面有个while循环，因此可以断定是卡在while循环了，google后发现有个老外遇到了一样的问题，他用div64\_u64\_rem替换后问题解决，div64\_u64\_rem函数没有用while循环去做减法，而是直接%取余。

进一步深入，研究为何会卡在while循环，各关键函数定义：

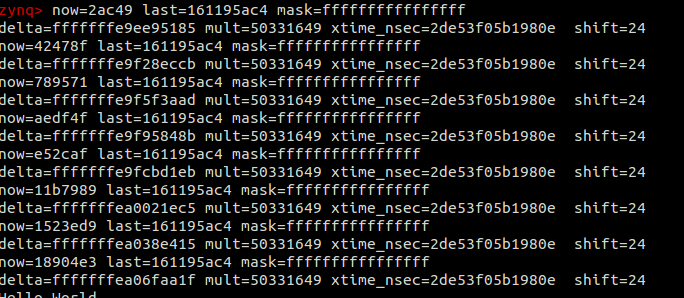








从\_\_iter\_div\_u64\_rem定义可以看出内核卡在了while循环，原因就是dividend出现了超大值，dividend实际就是\_\_getnstimeofday64中的nsecs，在函数timekeeping\_get\_ns和timekeeping\_get\_delta中加打印，结果如下：



可以看到出现了now小于last的情况，根据timekeeping\_get\_delta函数定义，now和last都是无符号64位，delta=now-last，now小于last的时候就会出现超大值。理论上64位数据按纳秒算的话可以存500多万个小时，now是不会出现比last小的情况的，可能的原因是cpu1启动后重新初始化了系统时钟，导致now归0。（未证实）

1. Vivado增加外设后寄存器无法读取

cpu0为裸机时：

在Vivado中增加外设后，要重新用SDK生成ps7\_init.c，然后用它重新编译fsbl，不然新的外设寄存器的内存是无法访问的。

cpu0为linux时：

linux启动后发现APER\_CLK\_CTRL以及寄存器uart0的时钟被关掉了（zynq\_clk\_setup干的），导致无法访问uart0寄存器，打开就能访问了（根本原因）

1. cpu1的uart来中断后linux卡死

目前amp系统的启动机制中，linux是感知不到cpu1的存在的，所以linux内核在初始化gic的时候会把所有中断都绑定到cpu0(详见下面的gic代码分析)，但是uart0又未在linux设备树中注册，导致没有相应中断处理函数，所以uart0产生中断的时候，捕捉中断的是cpu0而不是cpu1，cpu0中又没有中断处理函数，导致内核发生unexpected IRQ trap at vector 00的错误

解决：在初始化uart0中断的时候，增加一步修改ICDIPTR14寄存器的操作，将uart0的中断绑定到cpu

(4)cpu1加入uart0后，线程调度失效

通过jtag和反汇编发现pc卡在assert函数，打印后确认是串口中断处理函数XUartPs\_InterruptHandler中的入参指针错误，通过打印和反汇编可以知道在uart0的中断处理函数中传入了xInterruptController结构体

由于没有gdb跟踪堆栈，只能在XUartPs\_InterruptHandler函数最开头强制死循环，然后连接jtag查看lr寄存器，对应反汇编的代码段地址，发现lr停在vApplicationIRQHandler函数中。

XScuGic\_CfgInitialize第一次在uart0初始化的时候被调用，传入InstancePtr，并且handler和CallBackRef附上正确的值，但是freeRTOS在初始化系统定时器（vConfigureTickInterrupt函数）的时候又调用了一次XScuGic\_CfgInitialize，传入了系统自己的InstancePtr，以及uart0中初始化过的XScuGic\_Config结构体(此结构体包含了vector table)，导致IsReady没有成立，所以进入下面的for循环，由于XScuGic\_Config之前被初始化过，Handler不为0，所以Handler没有重新赋值，而是把.CallBackRef重新赋值成了传入的InstancePtr，也就是xInterruptController。

解决：整个系统用同一个xInterruptController，这样XScuGic\_CfgInitialize检测到IsReady后就不会重新初始化中断向量表了，并且把GIC，初始化放到main函数最前面，在对其他外设初始化时只调用外设中断相关接口即可

(5)以下代码导致串口收中断永远进不来

if(0 != (IsrStatus & (XUARTPS\_IXR\_RXOVR | XUARTPS\_IXR\_RXEMPTY |

XUARTPS\_IXR\_RXFULL)))

{

/\* Recieved data interrupt \*/

while(XUartPs\_IsReceiveData(InstancePtr->Config.BaseAddress))

{

if(iRingBufferGetEmptyCount(&pxUartDrvPrivate->xUartRxRingBuf) > 0)

{

ucVal = XUartPs\_ReadReg(InstancePtr->Config.BaseAddress, XUARTPS\_FIFO\_OFFSET);

// Print\_Warn("IsrStatus=%x val=%d\n", IsrStatus, ucVal);

iRet = xRingBufferPut(&pxUartDrvPrivate->xUartRxRingBuf, &ucVal, sizeof(uint8\_t));

}

else

break;

}

}

原因是这段代码在ringbuffer满的时候就break，不从rx fifo读数据了但此时fifo还是满的，然后中断标志位清除，后面再也没有fifo从不满到满的过程，导致fifo满中断进不来了

(5)莫名其妙死机

连上jtap，打开xmd，输入rrd，寄存器信息如下：

r0: 0x00000000 r1: 0xb36f53e8 r2: 0x00000000 r3: 0x1a0fc45c

r4: 0xb36f53e8 r5: 0x1a0899e0 r6: 0x00000000 r7: 0x1a0899e0

r8: 0x00000000 r9: 0x00000000 r10: 0x60000000 r11: 0x00000000

r12: 0x00000000 sp: 0x1a1038a0 lr: 0x0000000c pc: 0x1a000040

fpscr: 0x0000001c cpsr: 0x200f019b

查看反汇编，发现找不到sp指向的地址，反汇编中发现这个地址超出了栈地址，在link script中把栈大小改大即可

Linux中ZYNQ的通用中断控制器

另外如果要在目录中大量替换, 可以使用find + xargs　　比如:　　find ./model -name \*.sh | xargs fromdos

Oc物理地址映射寄存器Register (slcr) OCM\_CFG：0xF8000910