Uboot相关问题总结

开发板

底板：MYB-SAMA5D3X

核心板:SAMA5D36

目录

[Uboot相关问题总结 1](#_Toc37145497)

[U-BOOT配置相关 5](#_Toc37145498)

[0 开发环境基本情况说明 5](#_Toc37145499)

[0.1开发环境 5](#_Toc37145500)

[0.2开发板情况 5](#_Toc37145501)

[1 Uboot ping不通问题 5](#_Toc37145502)

[1.1解决方法： 6](#_Toc37145503)

[2 tftp设置 6](#_Toc37145504)

[2.1 开发板tftp客户端设置 6](#_Toc37145505)

[2.2 ubuntu tftp服务器端设置 7](#_Toc37145506)

[3 编译内核，设置其可以通过NFS挂载根文件系统 7](#_Toc37145507)

[4 设置nfs服务器 8](#_Toc37145508)

[5 uboot通过tftp加载Linux kernel，并启动 9](#_Toc37145509)

[5.1 内核及.dtb文件的准备 9](#_Toc37145510)

[5.2 uboot下载uImage及sama5d36ek.dtb 9](#_Toc37145511)

[5.3 启动内核 10](#_Toc37145512)

[6 通过NFS启动PC上制作的ROOTFS 10](#_Toc37145513)

[6.1设置uboot给内核传递的启动参数bootargs 10](#_Toc37145514)

[7 Kernel sources 10](#_Toc37145515)

[7.1 获取源代码 10](#_Toc37145516)

[7.2 练习使用补丁命令patch 10](#_Toc37145517)

[Eclipse+Jlink实现单步调试uboot 12](#_Toc37145518)

[一 软件的安装 12](#_Toc37145519)

[0 安装交叉编译工具 12](#_Toc37145520)

[1 Windows下的安装（暂时不用） 12](#_Toc37145521)

[1 jre-8u211-windows-x64.exe 12](#_Toc37145522)

[2 eclipse-inst-win64.exe(直接从官网下载)安装在了D:\eclipse目录下 12](#_Toc37145523)

[3 安装Zylin插件(在线安装) 12](#_Toc37145524)

[4 JLink\_V500l.exe 该文件在本机D:\安装软件目录下 13](#_Toc37145525)

[5 安装ARM编译工具 13](#_Toc37145526)

[2 Ubuntu下的安装 13](#_Toc37145527)

[2.1 安装eclipse-platform 13](#_Toc37145528)

[2.2 安装CDT 13](#_Toc37145529)

[2.3 安装Zylin插件(在线安装)（似乎没用到） 13](#_Toc37145530)

[2.4 安装JLINK 13](#_Toc37145531)

[3 GNU MCU Eclipse（暂时不用） 14](#_Toc37145532)

[3.1 xpm - xPack Manager安装 14](#_Toc37145533)

[3.2 工具链 15](#_Toc37145534)

[3.3 SEGGER J-LINK 15](#_Toc37145535)

[3.4 SEGGER J-LINK链接设备 17](#_Toc37145536)

[3.5 J-Link debugging插件 18](#_Toc37145537)

[二 Eclipse说明 20](#_Toc37145538)

[1 插件说明 20](#_Toc37145539)

[三 使用科大镜像更新插件 23](#_Toc37145540)

[四 在eclipse中编译u-boot(eclipse-platform CDT ubuntu-16.04.6-desktop-i386.iso) 24](#_Toc37145541)

[0.在64位系统上跑32位程序的配置1 24](#_Toc37145542)

[1.新建u-boot工程并配置 24](#_Toc37145543)

[2.配置工程： 28](#_Toc37145544)

[3.新建build target: 32](#_Toc37145545)

[4.编译工程 32](#_Toc37145546)

[五 在GNU MCU Eclipse中编译u-boot 34](#_Toc37145547)

[文件系统的制作 34](#_Toc37145548)

[1 使用BusyBox制作嵌入式Linux根文件系统 34](#_Toc37145549)

[1.1 BusyBox源码的获取 34](#_Toc37145550)

[1.2 构建嵌入式Linux根文件系统 35](#_Toc37145551)

[2 制作UBIFS文件系统镜像 38](#_Toc37145552)

[2.1制作UBS镜像，需要确定以下几个参数 38](#_Toc37145553)

[2.2使用mkfs.ubifs命令将某个文件夹制作为UBIFS镜像 39](#_Toc37145554)

[2.3 ubinize转换镜像格式 40](#_Toc37145555)

[SSH功能的添加（基于busybox文件系统） 41](#_Toc37145556)

[1 dropbear的移植 41](#_Toc37145557)

[1.1 编译zlib-1.2.8 41](#_Toc37145558)

[1.2 编译dropbear 42](#_Toc37145559)

[1.3移植到目标板 42](#_Toc37145560)

[1.4遇到的问题 43](#_Toc37145561)

[2 openSSH的移植 45](#_Toc37145562)

[2.1 交叉编译zlib-1.2.8 45](#_Toc37145563)

[2.2交叉编译openssl 45](#_Toc37145564)

[2.3编译openssh 45](#_Toc37145565)

[2.4 将文件拷贝至busybox制作的未压缩文件系统目录下 45](#_Toc37145566)

[2.5 PC使用ssh登录时存在的问题（**permitrootlogin 问题**） 46](#_Toc37145567)

[gcc更换引起的下载到arm上的程序无法运行问题解决 47](#_Toc37145568)

# U-BOOT配置相关

## 0 开发环境基本情况说明

### 0.1开发环境

系统：PC端为win10，在win10下安装virtualBox，virtaulBox里安装ubuntu16.04

主机与虚拟机网络情况如下图：



其中win10 ip：192.168.137.1

Ubuntu ip:192.168.137.72

### 0.2开发板情况

开发板底板：MYB-SAMA5D3X

主板：SAMA5D36

## 1 Uboot ping不通问题

情况说明:

主机PC

ip:192.168.137.1 netmask:255.255.255.0

烧写完uboot直接ping 192.168.137.1不通

### 1.1解决方法：

#### 1.1.1 设置ethaddr

**U-Boot>setenv ethaddr 12:34:56:ab:cd:ef**

#### 1.1.2 设置ip

**U-Boot>setenv ipaddr 192.168.137.3**

此时如果直接ping 192.168.137.1会显示：

host 192.168.137.1 is alive，说明配置成功，PC ping仍ping不通。

#### 1.1.3 设置serverip

**U-Boot>setenv serverip 192.168.137.72**

(注意因为在使用tftp中开发板是与ubuntu通信，从ubuntu中下载文件，因此这里设置serverip为192.168.137.72,而不因该是192.168.137.1,此处要特别注意)

#### 1.1.4设置autostart

**U-Boot>setenv autostart no**

#### 1.1.5 保存设置

**U-Boot>saveenv**

然后按reset重启开发板，此时再次ping 192.168.137.1 显示内容：host 192.168.137.1 is alive

然后PC去ping 192.168.137.3，仍然ping不通。

#### 1.1.6 运行tftp

**U-Boot>tftp**

此时主机再ping 192.168.137.3发现可以ping通。

## 2 tftp设置

### 2.1 开发板tftp客户端设置

注意要设置setenv serverip 192.168.137.72

### 2.2 ubuntu tftp服务器端设置

   接下来，就是要在开发环境上安装TFTP服务器，使开发板可以通过TFTP协议下载的uImage这个文件。在ubuntu下，可以通过下面的命令安装TFTP服务器，这个服务是通过inet监听的。

**sudo apt-get install atftpd openbsd-inetd**

安装完以后,需要配置一下TFTP的默认查找目录，我将其设定为/srv/tftp。确认/etc/inetd.conf文件中有如下一行：

|  |
| --- |
| ＃：BOOT：TFTP服务主要用于启动。大多数网站  ＃仅在充当“启动服务器”的计算机上运行。  **tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /srv/tftp** |

因为TFTP服务主要是给UBOOT提供内核镜像文件，为了避免每次内核编译完以后都拷贝到/srv/tftp目录中,在/srv/tftp目录中，建立了一个符号文件，指向/opt/linux/Linux-at91/linux-at91/arch/arm/boot/uImage。

**ln -s /opt/linux/Linux-at91/linux-at91/arch/arm/boot/uImage /srv/tftp/uImage**

可以通过下面的命令重启inetd，保证这个supper服务器能够监听TFTP端口：

**sudo service openbsd-inetd restart**

可以通过查看端口确认inetd是否真的在监听TFTP端口：

**root@ep-VirtualBox:~#netstat -a |grep tftp**

**udp 0 0 \*:tftp \*:\***

可以通过下面的命令测试一下TFTP服务是否正常工作：

**root@ep-VirtualBox:~# tftp localhost**

**tftp>get uImage**

**在0.3秒内收到3578106字节。**

## 3 编译内核，设置其可以通过NFS挂载根文件系统

对于sama5d3xek开发板提供的内核源码，解压后直接运行:

#./make\_image.sh linux-512mb

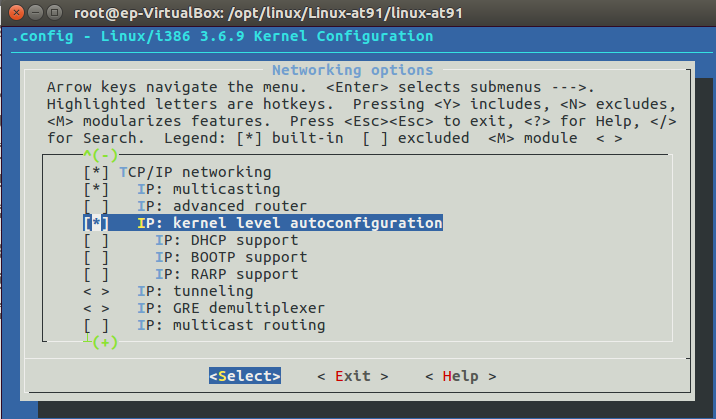
对于通用的内核源码，解压后直接:

make menuconfig

关于NFS挂在有关的选项如下：

1. Networking support->Networking options->IP:kernel level autoconfiguration

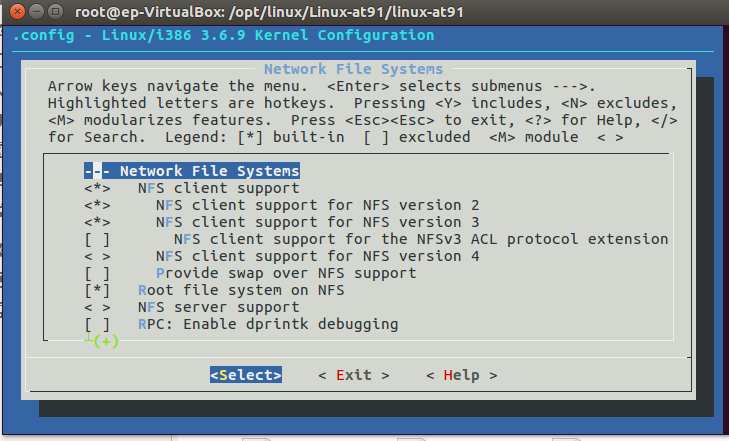
 注意，这个选项下面的 IP:DHCP support / IP:BOOTP support / IP:RARP aupport 不能选。因为我的开发环境中没有安装 DHCP server。开发板的 IP 是在内核启动参数中指定的。



1. File systems -> Network File System -> NFS client support

File systems -> Network File System -> NFS client support for NFS version 3

File systems -> Network File System ->Root file system on NFS



注：若通过make menuconfig修改了内核配置后，直接make，此时只更新了zImage。若要产生相应的uImage,则需要使用mkimage命令：

mkimage -A arm -O linux -T kernel -C none -a 0x20008000 -e 0x20008000 -n Linux-3.6.9 -d zImage uImage

## 4 设置nfs服务器

接下来是配置NFS服务器，用于开发板上内核启动以后挂载开发环境的ROOTFS。通过下面的命令安装nfs服务器：

**sudo apt-get install nfs-kernel-server**

安装完成以后，还需要修改/etc/exports文件，设置NFS共享的文件目录。我们需要将/opt/rootfs设置为NFS共享目录。

# Example for NFSv2 and NFSv3:

#/srv/homes hostname1(rw,sync,no\_subtree\_check) hostname2(ro,sync,no\_subtree\_check)

#Example for NFSv4:

#/srv/nfs4 gss/krb5i(rw,sync,fsid=0,crossmnt,no\_subtree\_check)

#/srv/nfs4/homes gss/krb5i(rw,sync,no\_subtree\_check)

/opt/rootfs 192.168.137.3(rw,no\_root\_squash,no\_subtree\_check)

注：192.168.137.3为开发板的ip。如果此处设置为了开发板ip,下面本机测试时将挂在deny。要想本机挂载测试成功可将ip改为\*

每一次修改/etc/exports这个文件,都需要重新启动NFS服务器：

**sudo service nfs-kernel-server restart**

可以通过下面的命令测试NFS服务器是否设置正确：

zoulz@Seagate:/tmp$ rm a b

zoulz@Seagate:/tmp$ mkdir /tmp/a

zoulz@Seagate:/tmp$ sudo mount -t nfs -o nolock localhost:/opt/rootfs /tmp/a

zoulz@Seagate:/tmp$ ls /tmp/a

bin  dev  etc  home  lib  linuxrc  proc  root  run  sbin  sys  tmp  usr  var

zoulz@Seagate:/tmp$ ls /opt/rootfs

bin  dev  etc  home  lib  linuxrc  proc  root  run  sbin  sys  tmp  usr  var

也可以通过df命令查看挂载情况：

zoulz@Seagate:/tmp$ df

Filesystem 1K-blocks Used Available Use% Mounted on

localhost:/opt/rootfs 125439744 36394112  89045632  30% /tmp/a

## 5 uboot通过tftp加载Linux kernel，并启动

### 5.1 内核及.dtb文件的准备

在虚拟机的/srv/tftp目录下放入linux内核镜像uImage及sama5d36ek.dtb文件或采用本说明文档中第2部分“tftp设置”中软连接的方式。

### 5.2 uboot下载uImage及sama5d36ek.dtb

#### 5.2.1 下载sama5d36ek.dtb

**U-Boot>tftp 0x21000000 sama5d36ek.dtb**

注：如果下载的时候找不到文件，请在ubuntu下用ls -al命令查看下文件的读权限，有可能是读权限的问题

#### 5.2.2 下载内核镜像uImage

**U-Boot>tftp 0x22000000 uImage**

### 5.3 启动内核

**U-Boot>bootm 0x22000000 – 0x21000000**

注：注意使用uboot的帮助命令查看命令“help”,好像还有个bootp命令直接从网络启动内核镜像，有兴趣可以研究一下。

注：0x22000000是内核镜像下载的内存地址，0x21000000是FDT（sama5d36ek.dtb）下载的内存地址。

说明：由于通过nfs启动rootfs，需要文件系统.

## 6 通过NFS启动PC上制作的ROOTFS

### 6.1设置uboot给内核传递的启动参数bootargs

U-Boot>**setenv bootargs** (注：删除原有的bootargs)

U-Boot> **setenv bootargs console=ttyS0,115200 root=/dev/nfs ip=192.168.137.3:::::eth0 nfsroot=192.168.137.72:/opt/rootfs,nfsvers=3 rw**

**setenv bootargs console=ttyS0,115200 root=/dev/nfs ip=192.168.137.3:::::eth0 nfsroot=192.168.137.72:/opt/test/embedded-linux-labs/bootloader/rootfs,nfsvers=3 rw**

## 7 Kernel sources

### 7.1 获取源代码

**wget** [**https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.17.1.tar.xz**](https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.17.1.tar.xz)

**解压缩：**

**tar xvf**

### 7.2 练习使用补丁命令patch

#### 7.2.1 Linux patch命令

**Linux patch命令用于修补文件。**

**patch指令让用户利用设置修补文件的方式，修改，更新原始文件。倘若一次仅修改一个文件，可直接在指令列中下达指令依序执行。如果配合修补文件的方式则能一次修补大批文件，这也是Linux系统核心的升级方法之一。**

#### 7.2.1.1 语法

patch [-bceEflnNRstTuvZ][-B <备份字首字符串>][-d <工作目录>][-D <标示符号>][-F <监别列数>][-g <控制数值>][-i <修补文件>][-o <输出文件>][-p <剥离层级>][-r <拒绝文件>][-V <备份方式>][-Y <备份字首字符串>][-z <备份字尾字符串>][--backup-if -mismatch][--binary][--help][--nobackup-if-mismatch][--verbose][原始文件 <修补文件>] 或 path [-p <剥离层级>] < [修补文件]

Eclipse+Jlink实现单步调试uboot

# 一 软件的安装

## 0 安装交叉编译工具

安装必要工具和解压交叉编译工具

sudo apt-get install build-essential git-core libncurses5-dev u-boot-tools

sudo apt-get install flex bison texinfo zip unzip zlib1g-dev gettext

报错：E: Failed to fetch

http://211.162.127.23/files/9068000005B8E996/us.archive.ubuntu.com/ubuntu/pool/main/m/m4/m4\_1.4.18-1\_amd64.deb File has unexpected size (201724 != 197166). Mirror sync in progress? [IP: 211.162.127.23 80]

将链接通过浏览器下载.deb，右键安装。

重新执行sudo apt-get install flex bison texinfo zip unzip zlib1g-dev gettext

sudo apt-get install gperf libsdl-dev

sudo apt-get isntall libesd0-dev （安装失败找不到）

sudo apt-get isntall libwxgtk2.6-dev安装失败找不到

## 1 Windows下的安装（暂时不用）

### 1 jre-8u211-windows-x64.exe

JRE,Java Runtime Environment,因为eclipse是基于java开发的，所以你需要安装java运行时才可以运行eclipse

### 2 eclipse-inst-win64.exe(直接从官网下载)安装在了D:\eclipse目录下

安装软件选择CDT安装

（或者采用安装插件形式<https://download.eclipse.org/tools/cdt/releases/8.8/> 未测试）

### 3 安装Zylin插件(在线安装)

下载地址：<http://opensource.zylin.com/embeddedcdt.html>

打开eclipse->Help->Installation New Software->Add…->Location处填写[http://opensource.zylin.com/zylincdt ，Name](http://opensource.zylin.com/zylincdt%20，Name)名字不用填写。

### 4 JLink\_V500l.exe 该文件在本机D:\安装软件目录下

### 5 安装ARM编译工具

<https://dl.bintray.com/gnu-mcu-eclipse/updates/> （此部分已经）

打开eclipse->Help->Installation New Software->Add…->Location处填写<https://dl.bintray.com/gnu-mcu-eclipse/updates/> ，选择全部建立。导入uboot项目后右键项目->属性->C/C++ Build->Tool Chain Editor->Current toolchain:No ToolChain没有工具链可以选择。

重新删掉uboot项目。

Eclipse->File->Import->C/C++->Existing Code as Makefile Project弹出Import Existing Code窗口，在该窗口中的Toolchain for Indexer Settings选择Cross ARM GCC，选择好uboot所在的项目后点击Finish

## 2 Ubuntu下的安装

### 2.1 安装eclipse-platform

sudo apt install eclipse-platform

### 2.2 安装CDT

sudo apt-get install eclipse eclipse-cdt

### 2.3 安装Zylin插件(在线安装)（似乎没用到）

打开eclipse->Help->Installation New Software->Add…->Location处填写[http://opensource.zylin.com/zylincdt ，Name](http://opensource.zylin.com/zylincdt%20，Name)名字不用填写。

### 2.4 安装JLINK

进入Segger官网，[www.segger.com](http://www.segger.com);

点击上栏菜单的“Downloads”，在弹出的选择中点击“J-Link/J-Trace”；

在新页面中，找到“J-Link Software and Documentation Pack”一栏，点击下方的“Click for downloads”；

找到“J-Link Software and Documentation pack for Linux, DEB installer, 64-bit”一栏，点击右方的“DOWNLOADS”按钮；

在新页面中，下滚到最后，在“I accept......”前打钩，再点击“Download Software”；

等待一段时间后（在国内下载速度较慢），得到文件“JLink\_Linux\_V633c\_x86\_64.deb”；

将此文件复制到Ubuntu中，例如 /home 中；

点击“Files”应用，进入到 /home；

右键选中上面提到的文件，选择“Open with Software install”，完成；

如何测试是否安装成功：在Terminal中，输入JLink，然后按3下Tab键，如有新命令显示，如JLinkExe，即表明安装成功。

## 3 GNU MCU Eclipse（暂时不用）





### 3.1 xpm - xPack Manager安装

开源项目路径<https://github.com/xpack/xpm-js>

安装node

**Unbuntu 16.04.6**

sudo apt install curl

curl -sL https://deb.nodesource.com/setup\_8.x | sudo -E bash -

sudo apt-get install -y nodejs

~$ node --version

v8.16.0

**ubuntu18.04.2**

sudo apt install curl

curl -sL https://deb.nodesource.com/setup\_12.x | sudo -E bash -

sudo apt-get install -y nodejs

~$ node --version

v12.4.0

安装npm

$ mkdir -p "${HOME}/opt/npm"

$ sudo npm config set prefix "${HOME}/opt/npm"

$ echo 'export PATH="${HOME}/opt/npm/bin:${PATH}"' >> "${HOME}/.profile"

$ source "${HOME}/.profile"

$ sudo npm install --global xpm

$ xpm --version

0.5.0

### 3.2 工具链

$ xpm install --global @gnu-mcu-eclipse/arm-none-eabi-gcc

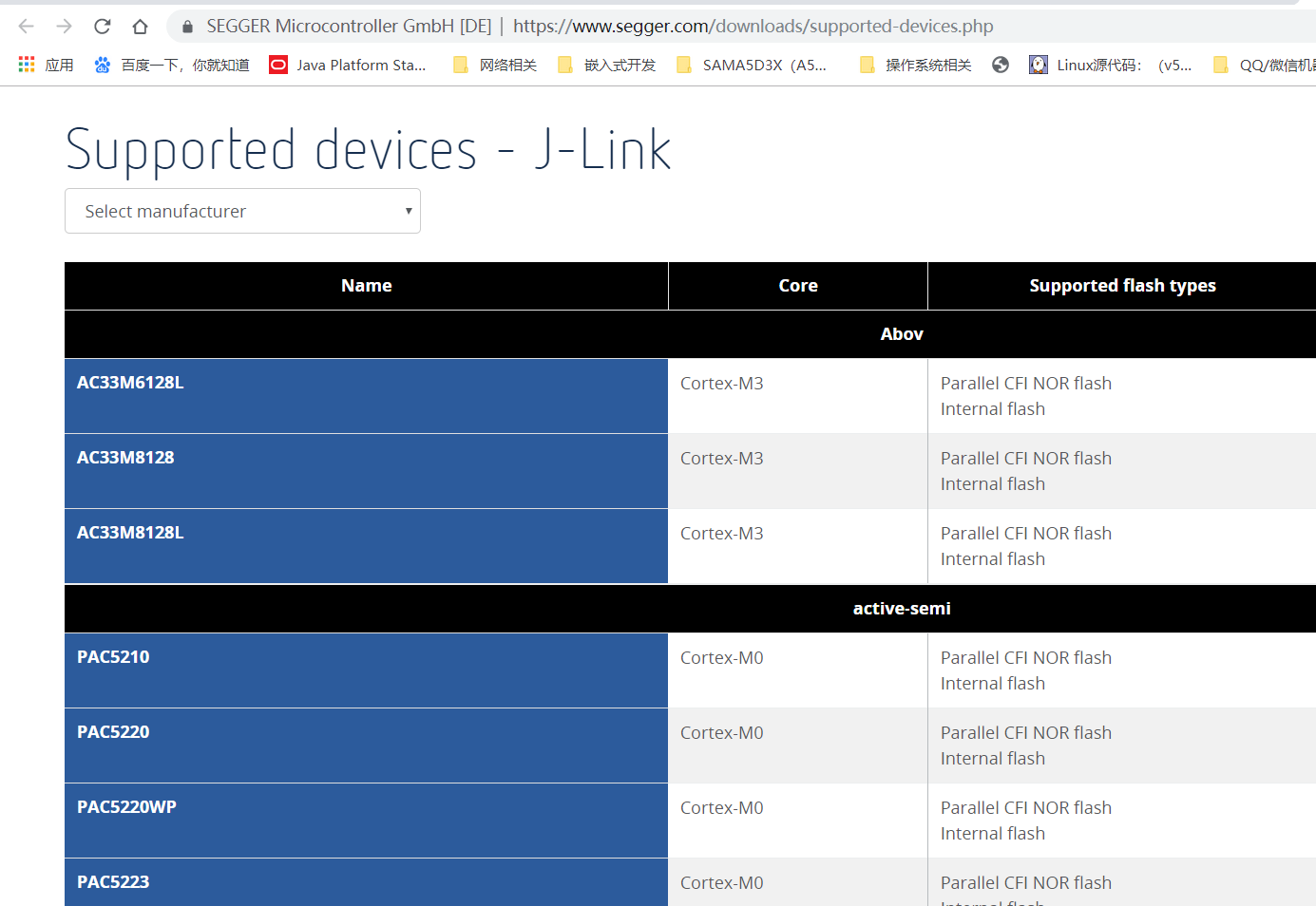
$ xpm install --global @gnu-mcu-eclipse/riscv-none-gcc

### 3.3 SEGGER J-LINK

安装步骤见ubuntu下的安装“安装JLINK”

测试

要测试J-Link是否能够连接到特定板，通常需要指定接口（JTAG或SWD）和设备名称。默认情况下，J-Link GDBServer将尝试JTAG，但如果仅连接SWD（这在自定义硬件上很常见），则需要指定接口（-if SWD）。对于需要在连接上进行特殊处理的目标，需要设备名称（例如，由于硅片错误导致无法进行自动检测）。有关可用设备名称的列表，请参阅SEGGER  [支持的设备](http://www.segger.com/jlink_supported_devices.html)  页面



<https://www.segger.com/downloads/supported-devices.php>

选择atmel，发现支持的设备为ATSAMA5D36

ep@ep-VirtualBox:~$ /usr/bin/JLinkGDBServer -if JTAG -device ATSAMA5D36(注意，要将SEGGER J-Link[0100]挂载到虚拟机上才行)

SEGGER J-Link GDB Server V6.46c Command Line Version

JLinkARM.dll V6.46c (DLL compiled Jun 3 2019 18:35:05)

Command line: -if JTAG -device ATSAMA5D36

-----GDB Server start settings-----

GDBInit file: none

GDB Server Listening port: 2331

SWO raw output listening port: 2332

Terminal I/O port: 2333

Accept remote connection: yes

Generate logfile: off

Verify download: off

Init regs on start: off

Silent mode: off

Single run mode: off

Target connection timeout: 0 ms

------J-Link related settings------

J-Link Host interface: USB

J-Link script: none

J-Link settings file: none

------Target related settings------

Target device: ATSAMA5D36

Target interface: JTAG

Target interface speed: 4000kHz

Target endian: little

Connecting to J-Link...

J-Link is connected.

Firmware: J-Link V9 compiled Jun 2 2222 22:22:22

Hardware: V9.40

S/N: 59401308

Feature(s): GDB, RDI, FlashBP, FlashDL, JFlash, RDDI

Checking target voltage...

Target voltage: 3.24 V

Listening on TCP/IP port 2331

Connecting to target...

J-Link found 1 JTAG device, Total IRLen = 4

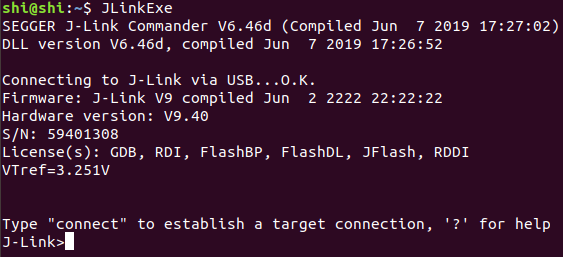
JTAG ID: 0x4BA00477 (Cortex-A5)

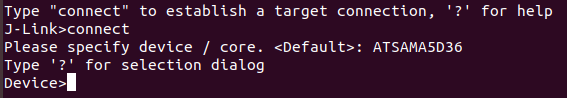
Connected to target

Waiting for GDB connection...

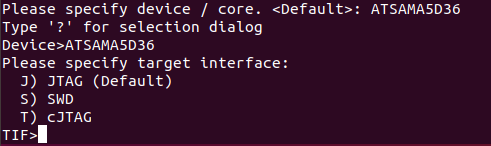
### 3.4 SEGGER J-LINK链接设备

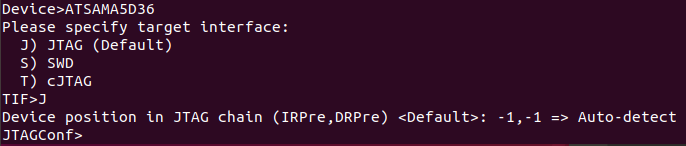
shi@shi:~$ JlinkExe

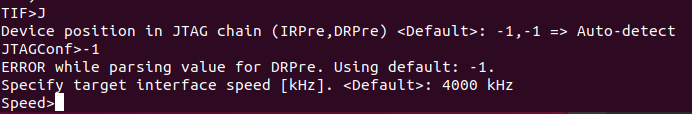


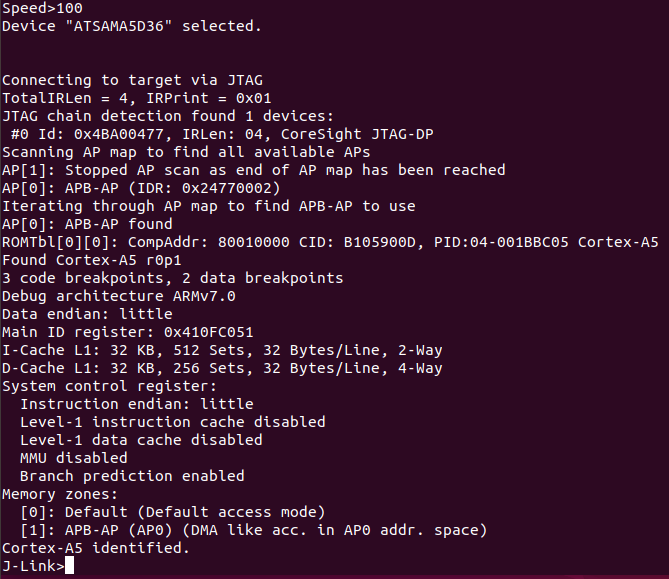


设备名字根据3.3描述部分选择









### 3.5 J-Link debugging插件

#### 3.5.1 为什麽要使用新的插件

Eclipse中的j-link探针调试是通过GDB硬件调试插件完成的。添加到此插件的功能包括：

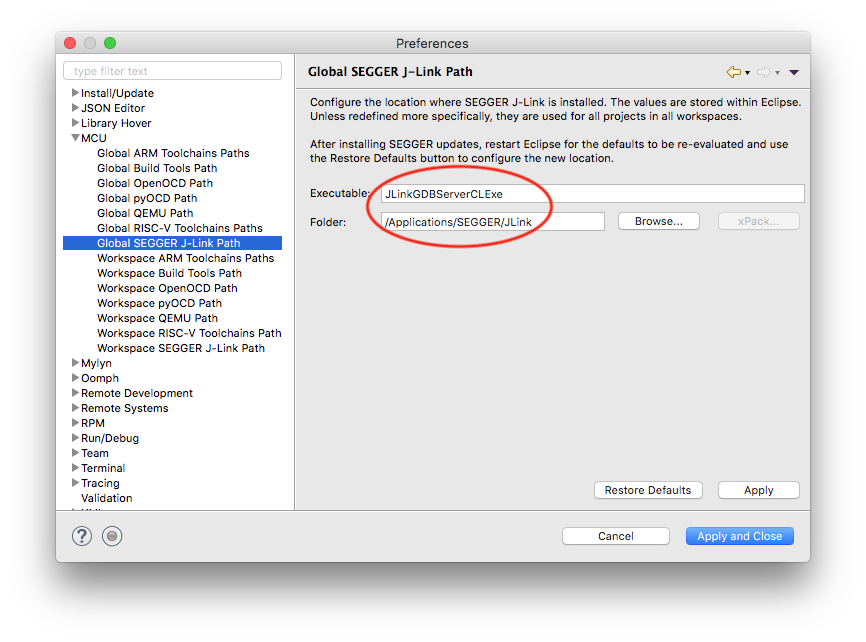
* 透明的集成了GDB Server，为每一个调试会话自动启动/停止
* 高级的图形用户界面用于配置大多数GDB客户端和GDB服务器的SEGGER特定选项
* 透明的将半主机和串行线查看器(SWV)窗口集成为标准Eclipse控制台
* 功能重启按钮，用于重置目标和重启调试会话，而无需重写可执行镜像。
* 直接使用项目工具链定义（前缀，后缀，路径）来简化GDB客户端的配置
* 基于宏的J-LINK文件夹路径定义，用于在升级到新的SEGGER版本时简化GDB服务器位置的管理。
* 支持SWO寄存器配置以匹配给特定的系统时钟频率

#### 3.5.2 先决条件

在能够将J-Link探针与GNU MCU Eclipse

* J-Link调试插件
* GDB调试器(客户端)应用程序，作为GNU工具链的一部分
* SEGGER J-Link GDB服务器和驱动程序。

#### 3.5.3 定义J-Link文件夹位置



3.5.4 将设备与项目关联

* 3.5.4.1概述
* 3.5.4.2 先觉条件

由GNU MCU Eclipse插件创建的项目

包可用：

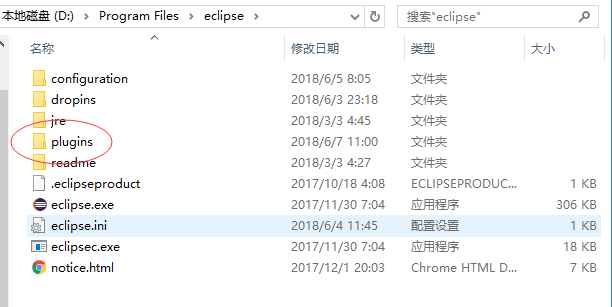
首先安装Packs插件(Packs manager)

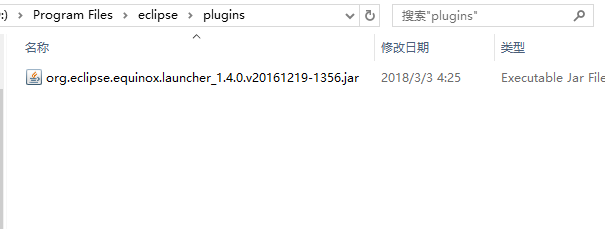
安装与正在开发的项目使用的设备相关的软件包

# 二 Eclipse说明

## 1 插件说明

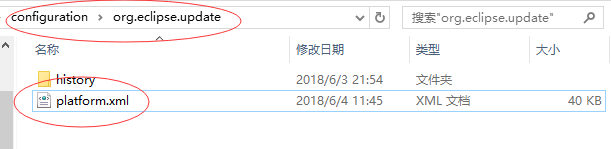
最近，发现新版Eclipse安装的插件不再像以前那样，安装在目录下的plugins的文件夹下，那么，有时候我们自己下载的离线的插件包要放在哪里呢，像往前版本放在目录下的plugins的文件夹下已经不生效了：



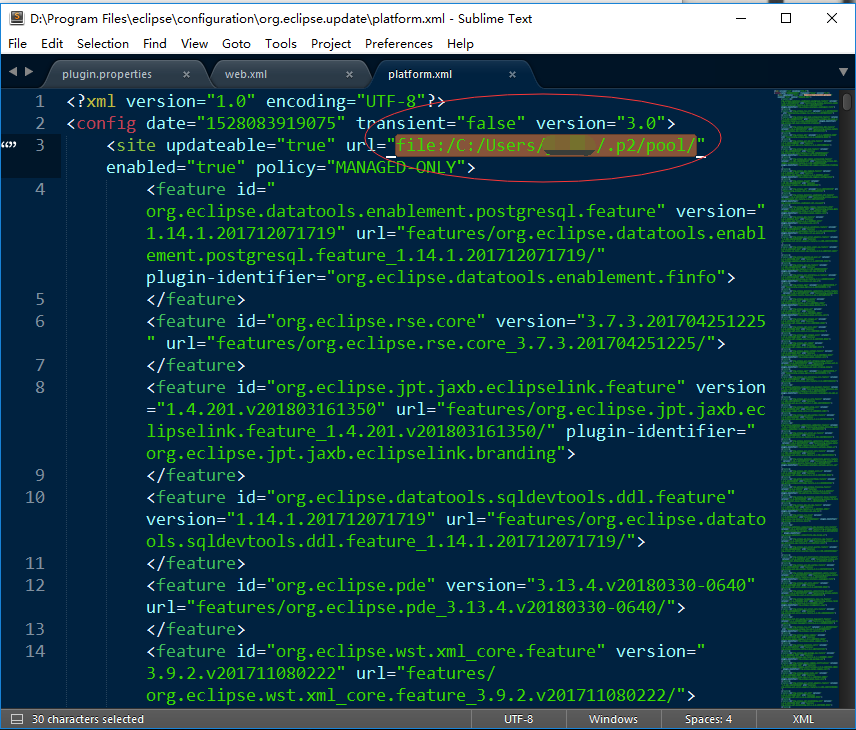


那么问题来了，我们在那里找到已经安装的插件包和解压离线的插件放哪里呢？为了解决这个问题，开始我将安装目录的每一个文件都搜了一遍。果然，功夫不负有心人，肯定会隐藏起来，通过配置文件进行关联。

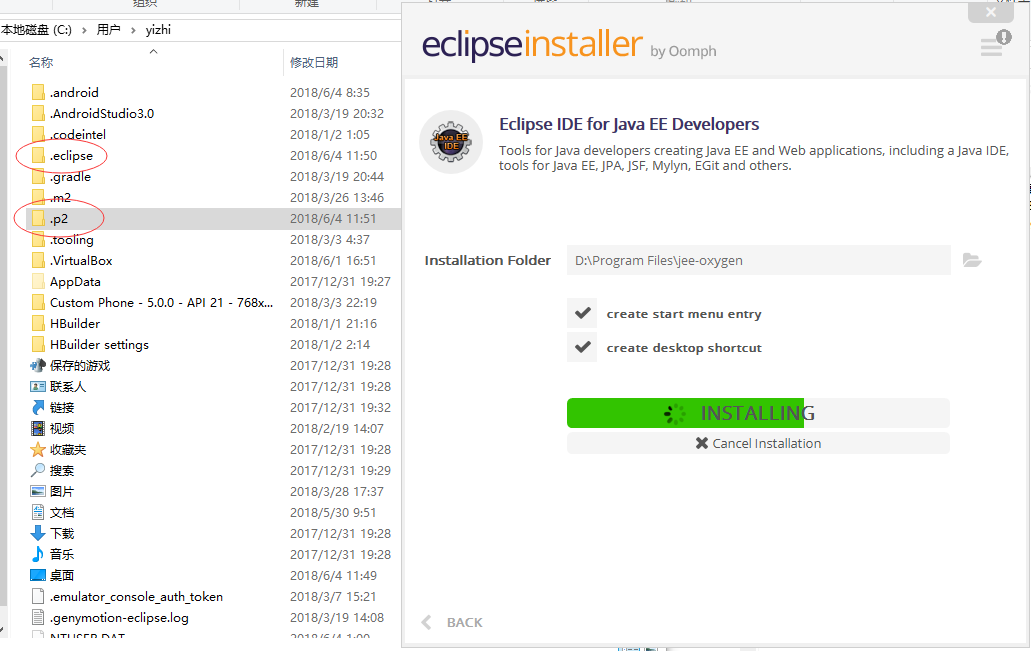
1. 找到安装目录的configuration -->> org.eclipse.update (为什么是在这个文件夹里面呢，新版本嘛，所以就在update的配置文件里咯) -->> platform.xml；



1. 打开platform.xml文件，看到关键关联代码：url="file:/C:/Users/\*\*\*/.p2/pool/",原来千呼万唤始出来，隐藏在此处：



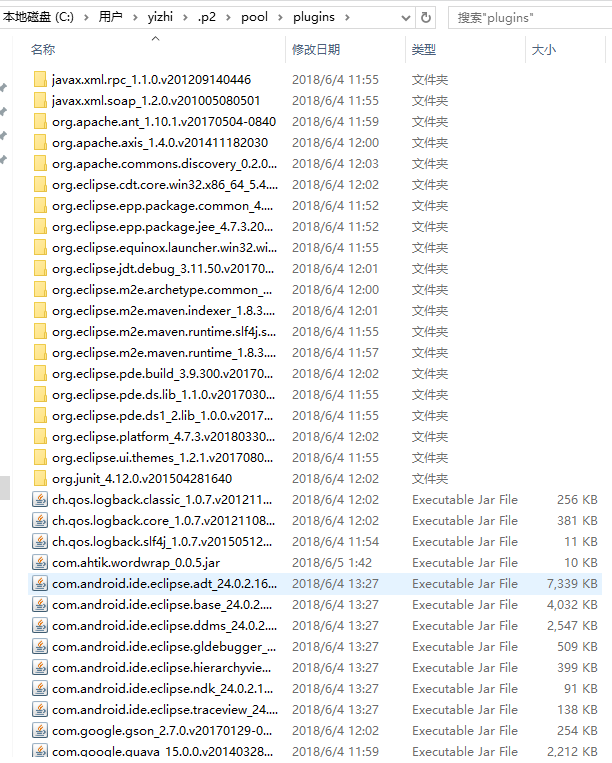
1. 原来新版安装Eclipse的时候，会在C盘用户文件生成两个关键的配置文件：.eclipse和.p2文件；



1. 打开C盘的.p2文件夹（C:/Users/本机电脑/.p2/pool/）,找到pool,豁然开朗，发现原来插件文件夹features和plugins在此：



1. 打开plugins，所有插件尽览无遗：



以后插件就安装在此目录就可以了，完毕。

思考:Eclipse安装以前的方式不是挺好的吗？为什么要这样呢？原因其实很简单，按照以前做法，试想如果我的电脑装了好多个eclipse，都要用到相同的是不是需要装好多次？按照新版本做法，无论安装多少个版本的Eclipse,插件只需安装一次，所有的eclipse都共用一个插件：

# 三 使用科大镜像更新插件

以 Luna 为例，点击 Help→Install New Software...→Available Software Sites 可以看到所有的更新源，将其中的download.eclipse.org全部替换成mirrors.ustc.edu.cn/eclipse即可。

# 四 在eclipse中编译u-boot(eclipse-platform CDT ubuntu-16.04.6-desktop-i386.iso)

## 0.在64位系统上跑32位程序的配置1

1)在64位debian上安装32位兼容库

$ sudo apt-get install ia32-libs

$ sudo apt-get install lib32c-dev

无法安装，找不到库，用下面这个方法，打开多架构支持，然后更新：

$ sudo dpkg --add-architecture i386

$ sudo apt-get update

$ sudo apt-get install ia32-libs

$ sudo apt-get install lib32c-dev

如果没有ia32-libs就用

sudo apt-get install lib32ncurses5

sudo apt-get install lib32c-dev

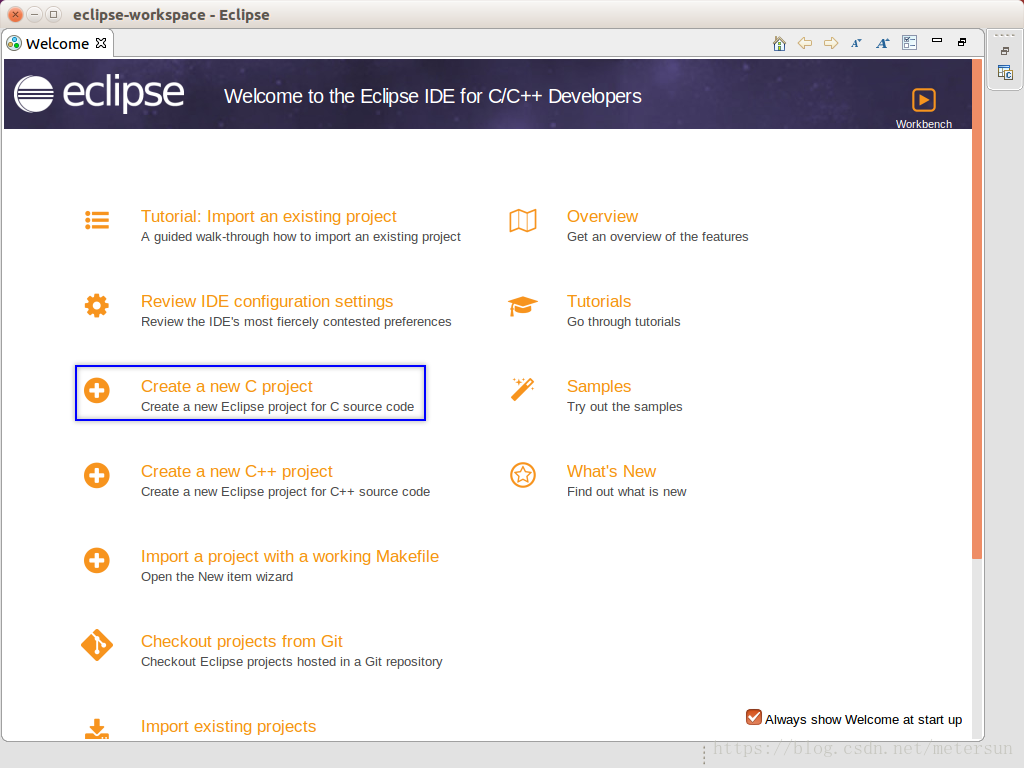
2)拷贝其他需要用到的32位库到相应的目录(注：应该只要这一步就可，没测试，只是查找相关内容，最终可以运行，记录下进行了的操作)

sudo dpkg --add-architecture i386

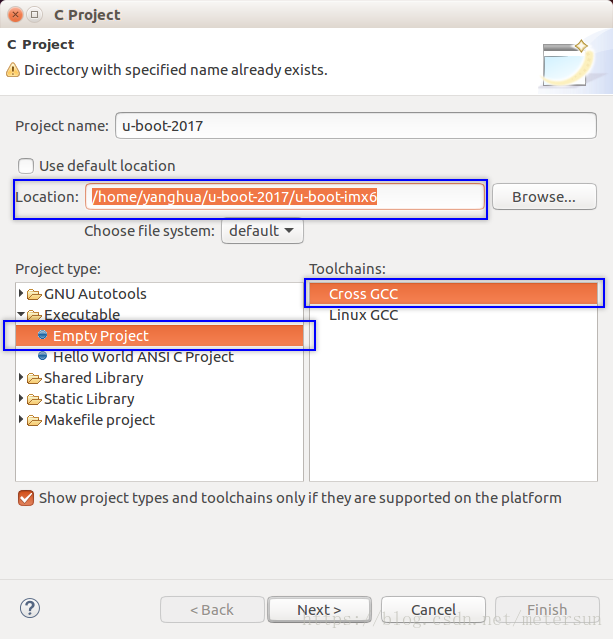
sudo apt install libc6:i386

## 1.新建u-boot工程并配置

1.新建工程：选择Create a new C Project



2.下一步

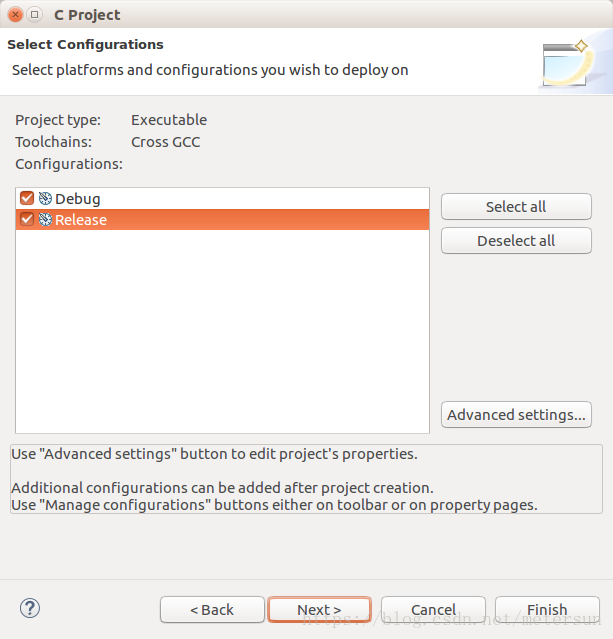


选择Executable-->Empty Project

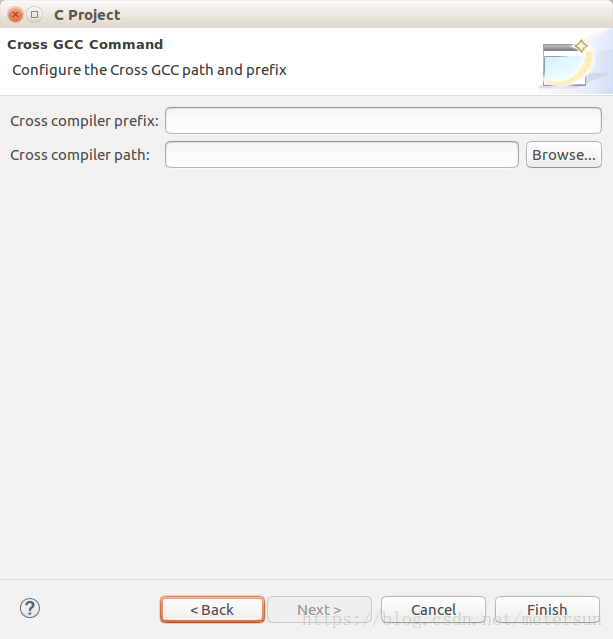
Location为u-boot目录(在对应的Browse...找到u-boot源码目录)

Toolchains为Cross GCC

3.下一步



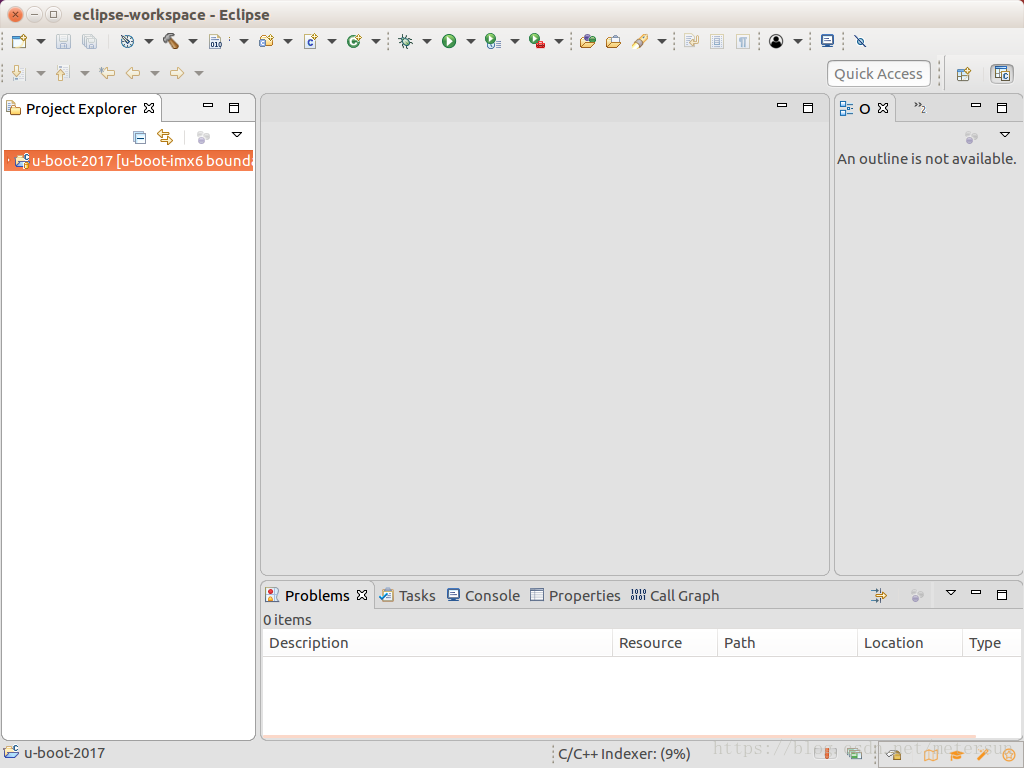
4.下一步



注意上面的Cross compiler prefix和Cross compiler path均为空。

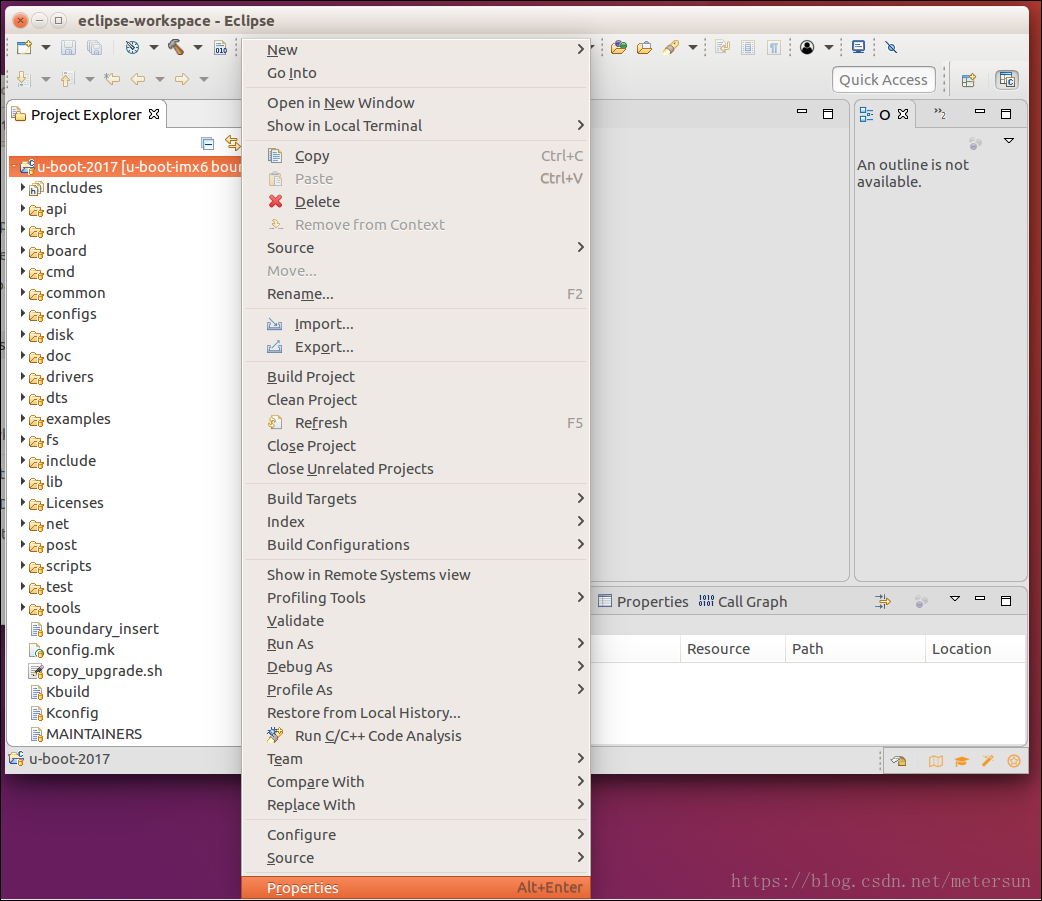
点击Finish。

5.Finish，并等待Indexer完成(状态栏有Indexer完成进度)：

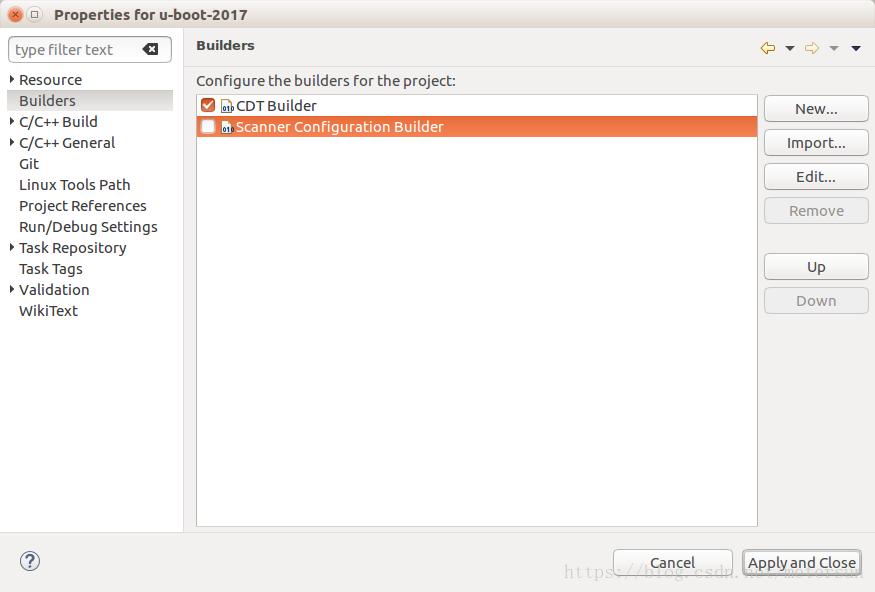


## 2.配置工程：

点击工程，在弹出的右键菜单中选择Properties:



1. Builders:



2. C/C++ Build:



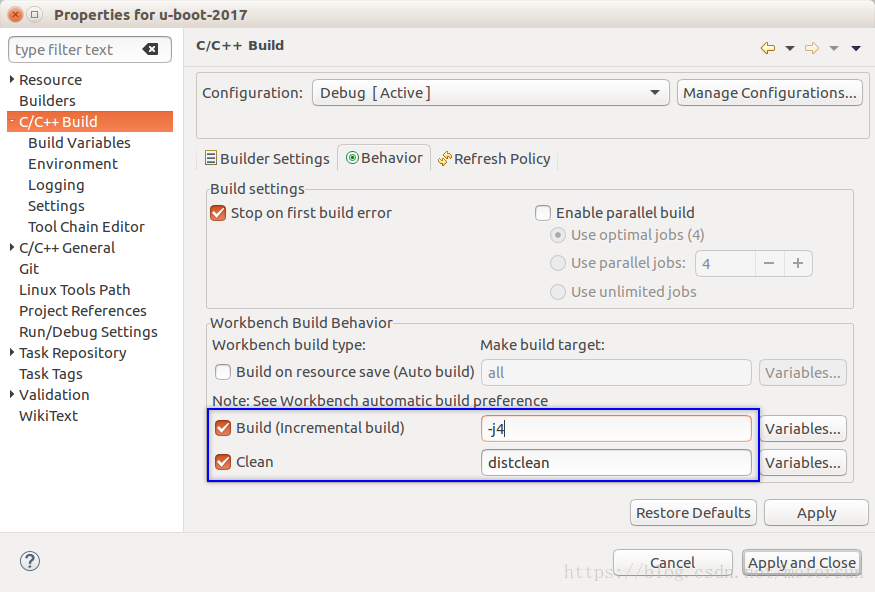
在Builder Settings选项卡勾选掉：

Use default build command

Generate Makefiles automatically

点击File system，选择u-boot源码目录。

3. Behavior选项卡中的设置:

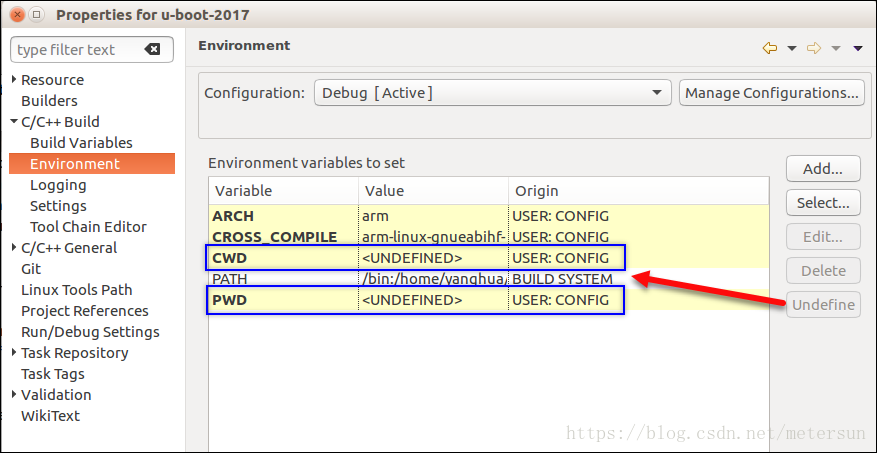


Build一栏，增加-j4，即使用4核编译

Clean一栏，填写distclean

4. C/C++ Build-->Environment:

并Undefine变量CWD，PWD，如下图：



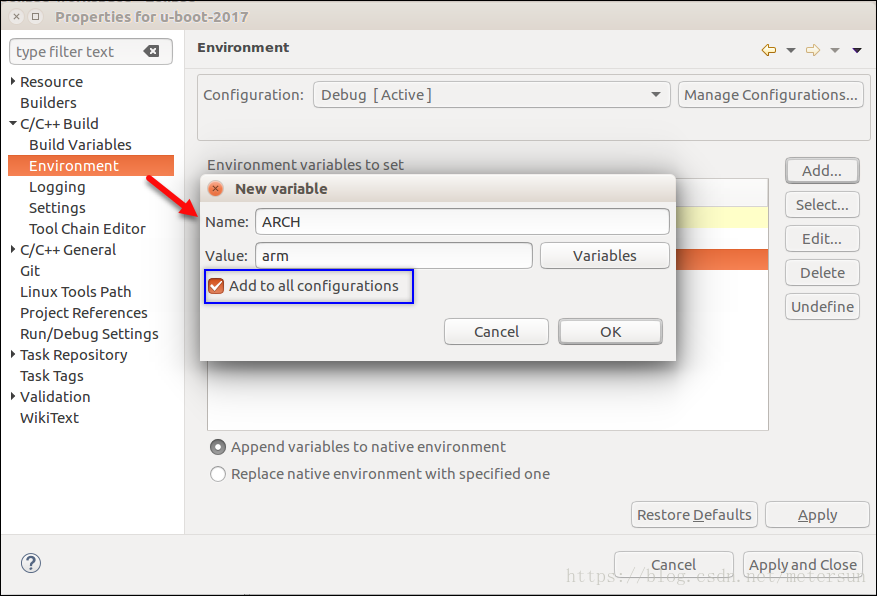
新建环境变量：

ARCH值为arm

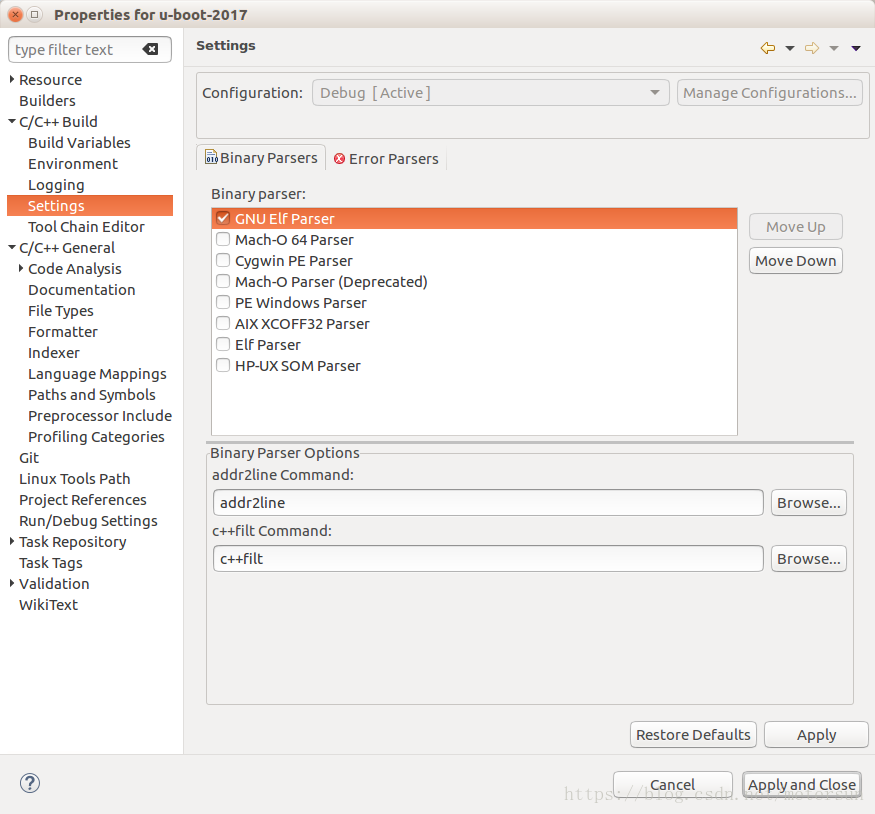
CROSS\_COMPILE值为:

/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-

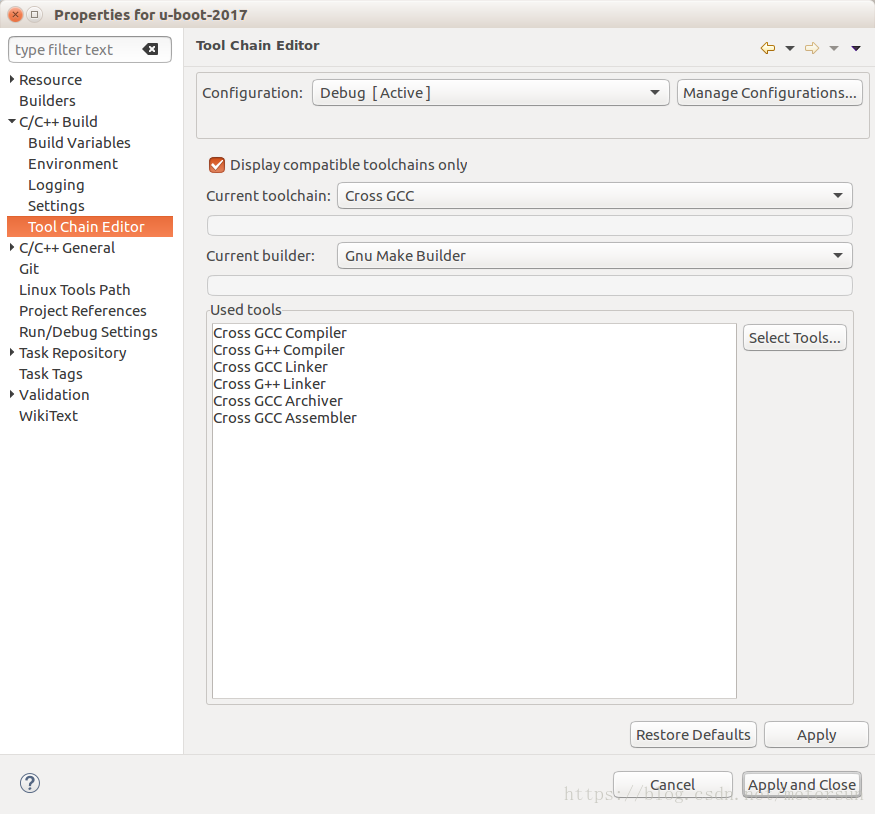
且都勾选Add to all configurations



5. C/C++ Build-->Settings:选择GNU Elf Parser

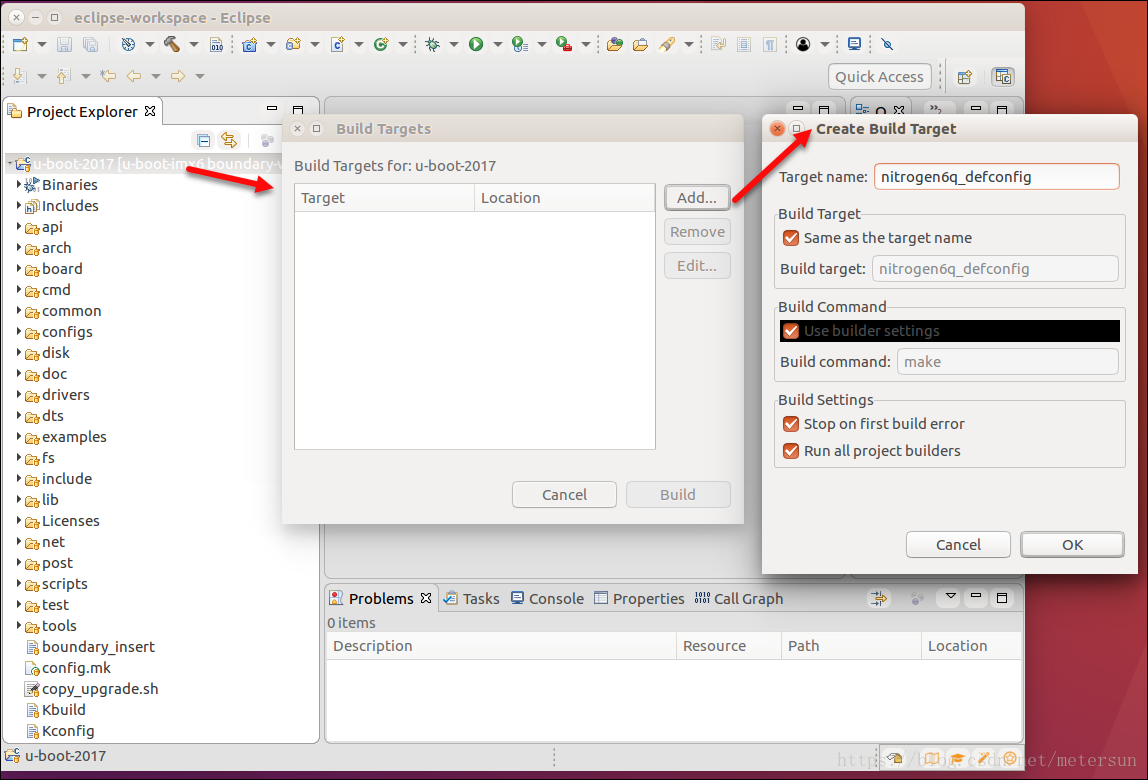


6. C/C++ Build-->Tool Chain Editor:选择Cross GCC



## 3.新建build target:

Shift + F9:

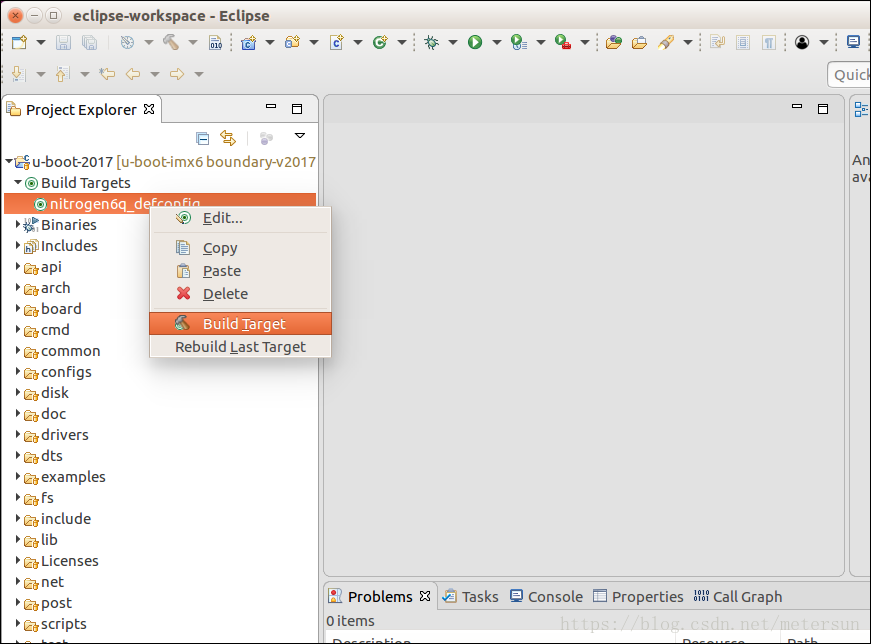


在新建的Target name一栏输入：nitrogen6q\_defconfig(sama5d3xeknf\_512MB\_config)

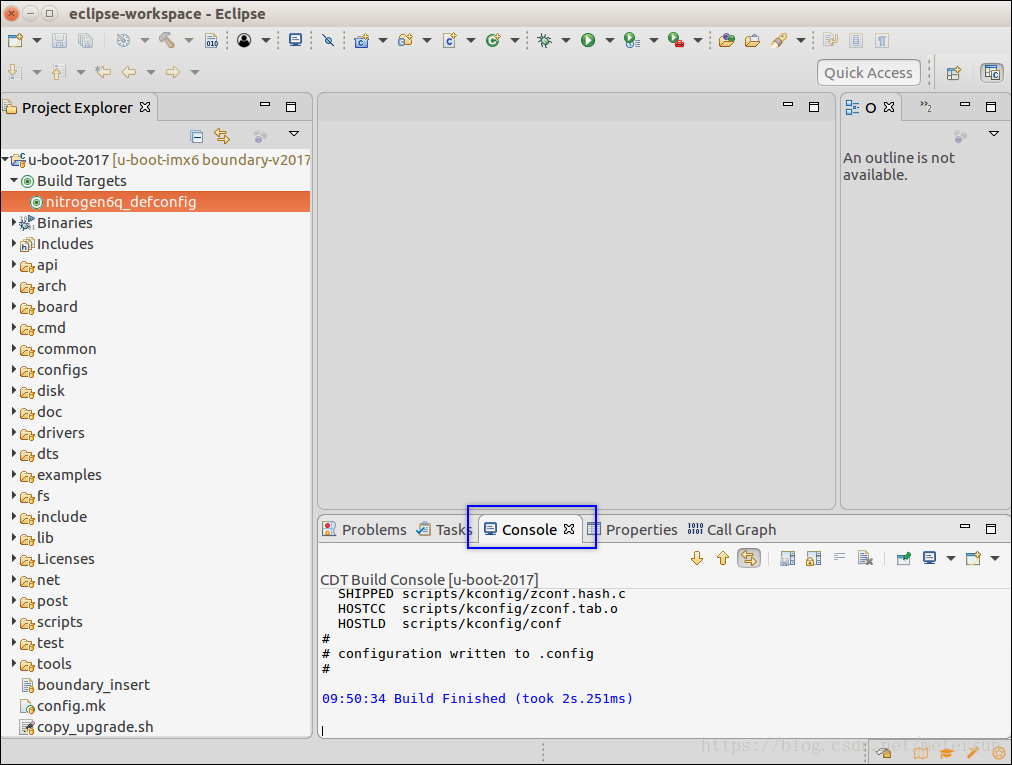
## 4.编译工程

1.先编译build Targets--> sama5d3xeknf\_512MB\_config，右键菜单上选择"Build Target"，

此时将执行make sama5d3xeknf\_512MB\_config。

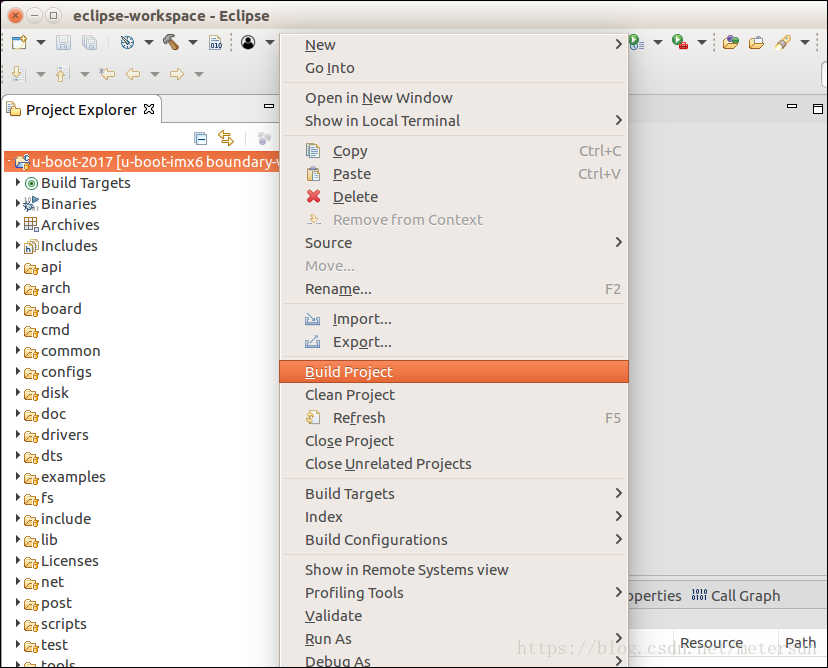


下方的Console选项卡中会显示编译输出结果：

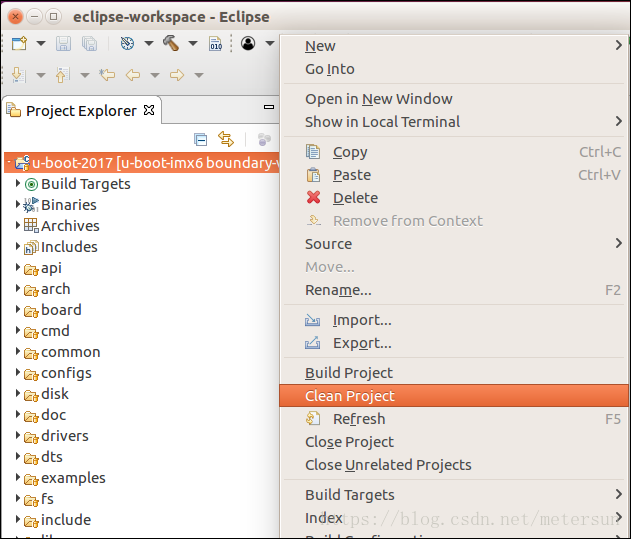


2. uboot2017-->Build Project

工程上单击右键，在弹出的菜单上选择"Build Project"，会执行make -j4命令：



3. clean，会执行make distclean命令



# 五 在GNU MCU Eclipse中编译u-boot

# 文件系统的制作

说明本部分此种颜色表示操作

## 1 使用BusyBox制作嵌入式Linux根文件系统

### BusyBox源码的获取

官网网址：<https://busybox.net/>

去官网下载当前最新的稳定版本[busyBox 1.30.1](https://busybox.net/downloads/busybox-1.30.1.tar.bz2)

busybox-1.30.1.tar.bz2

将其解压到embedded-linux-labs/bootloader目录下

### 构建嵌入式Linux根文件系统

假设创建的rootfs目录在/opt/下，即/opt/rootfs

#### STEP1: 构建目录结构

#makedir ./rootfs

#cd ./rootfs

创建跟文件系统目录，主要包括以下目录

/dev /etc /lib /usr /var /proc /tmp /home /root /mnt /bin /sbin /sys

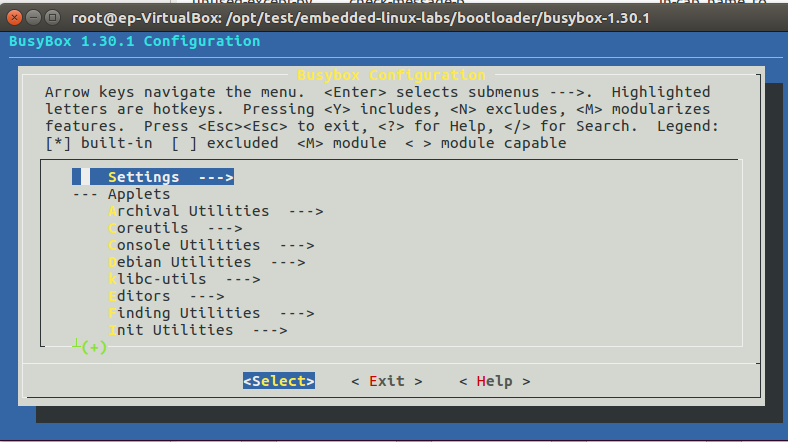
#mkdir dev etc lib usr var proc tmp home root mnt sys

#### 1.2.2 STEP2: 使用busybox构建/bin /sbin linuxrc

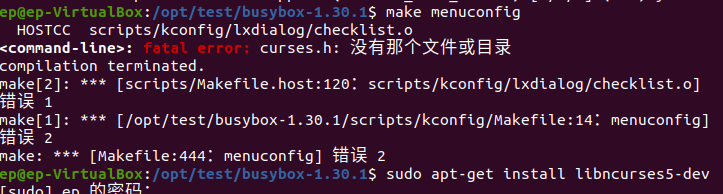
进入busybox-1.30.1目录，执行

#make defconfig

#make menuconfig



**若出现以下错误：**



**Ubuntu的需要安装libncurses5-dev，安装即可**

以下为busybox Configuration配置

Busybox Settings---->

--- Build Options----->

//1. 选择将busybox进行静态编译

**[\*] Build static binary (no shared libs)**

//2. 指定交叉编译器为

**(/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-)Cross Compiler prefix**

//选择生成的文件存放目录，也可以直接放在rootfs下就不用拷贝了，默认是在busybox根目录下生成\_install目录

--- Installation Options(“make install” behavior) ----->

(./\_install) Destination path for ‘make install’

//3. 选择上Don’t use /usr

--- Support –long-options ----->

**[\*] Don’t usr /usr**

// 4. 编译出的busybox的shell命令解释器支持显示当前路径及主机信息

--- Library Tuning----->

[\*]Username completion

[\*]Fancy shell prompts

**[\*]Query cursor position from terminal**

**保存退出**

#make

#make install

在busybox目录下会看见\_install目录，里面有/bin /sbin linuxrc三个文件。将这三个文件目录或文件拷到第一步所建的rootfs文件夹下。

#cp bin/ sbin/ linuxrc /opt/rootfs -ra

切记一定要带上-a的参数，因为bin目录里大部分都是链接，如果不带-a的参数，拷过去之后会做相应的复制，不再是链接的形式

#### 1.2.3 STEP3: 构建etc目录

1)进入根文件系统rootfs的etc目录，执行如下操作：

拷贝busybox-1.30.1/examples/bootfloopy/etc/\* 到当前目录下

cp -r busybox-1.30.1/examples/bootfloppy/etc/\* rootfs/etc

修改inittab文件

原始文件为：

::sysinit:/etc/init.d/rcS

::respawn:-/bin/sh

tty2::askfirst:-/bin/sh

::ctrlaltdel:/bin/umount -a -r

修改后为：

(1): 开机免登陆，直接打开shell（前面数字为行号）

1 ::sysinit:/etc/init.d/rcS

2 #::respawn:-/bin/sh

3 #::respawn:-/bin/login

4 console::askfirst:-/bin/sh

5 #tty2::askfirst:-/bin/sh

6 ::ctrlaltdel:/bin/umount -a -r

(2): 开机需要登陆（前面数字为行号）

1 ::sysinit:/etc/init.d/rcS

2 #::respawn:-/bin/sh

3 ::respawn:-/bin/login

4 #console::askfirst:-/bin/sh

5 #tty2::askfirst:-/bin/sh

6 ::ctrlaltdel:/bin/umount -a -r

2)拷贝虚拟机上的/etc/passwd, /etc/group, /etc/shadow到rootfs/etc下

# cp /etc/passwd /opt/rootfs/etc

# cp /etc/group /opt/rootfs/etc

# cp /etc/shadow /opt/rootfs/etc

对以下三个文件修改，只保存与root相关的项，根据具体情况内容会有所不同。

修改passwd为root:x:0:0:root:/root:/bin/ash （此处后面要改为sh，不用ash）

修改group为root:x:0:root

修改shadow为root:$1$x9yv1WlB$abJ2v9jOlOc9xW/y0QwPs.:14034:0:99999:7:::

登陆开发板时需输入用户名密码，同虚拟机相同

3)修改profile

PATH=/bin:/sbin:/usr/bin:/usr/sbin

export LD\_LIBRARY\_PATH=/lib:/usr/lib

/bin/hostname shizhaopeng

USER="`id -un`"

LOGNAME=$USER

HOSTNAME='/bin/hostname'

PS1='[\u@\h \W]#'

4)修改etc/init.d/rc.S文件

#! /bin/sh

/bin/mount -n -t ramfs ramfs /var

/bin/mount -n -t ramfs ramfs /tmp

/bin/mount -n -t sysfs none /sys

/bin/mount -n -t ramfs none /dev

/bin/mkdir /var/tmp

/bin/mkdir /var/modules

/bin/mkdir /var/run

/bin/mkdir /var/log

/bin/mkdir -p /dev/pts //telnet服务需要

/bin/mkdir -p /dev/shm //telnet服务需要

#echo /sbin/mdev > /proc/sys/kernel/hotplug //USB自动挂载需要

/sbin/mdev -s //启动mdev在/dev下自动创建设备文件节点

/bin/mount -a

#################配置网络#########################

#/sbin/ifconfig lo 127.0.0.1 netmask 255.0.0.0

/sbin/ifconfig eth0 192.168.137.3

/sbin/ifconfig eth0 netmask 255.255.255.0

/sbin/route add default gw 192.168.137.1 eth0

/sbin/ifconfig eth0 up

5) 修改etc/fstab文件，增加以下文件

none /dev/pts devpts mode=0622 0 0

tmpfs /dev/shm tmpfs defaults 0 0

#### 1.2.4 STEP4: 构建lib目录

（/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/arm-linux-gnueabihf/lib我的只是把这个目录下的SO文件复制就可以了）

1)#cd /opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/arm-linux-gnueabihf/libc/lib/arm-linux-gnueabihf

将以下动态库拷贝到rootfs/lib下

#cp \*so\* /opt/rootfs/lib -a

2)#cd /opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/arm-linux-gnueabihf/lib

将以下动态库拷贝到rootfs/lib下

#cp ./libstdc++.so.\* /opt/rootfs/lib -a

至此通过NFS启动的roofs文件系统制作完毕

## 制作UBIFS文件系统镜像

### 2.1制作UBS镜像，需要确定以下几个参数

MTD partition size; //对应的FLASH分区大小

flash physical eraseblock size; //FLASH物理擦除块大小

minimum flash input/output unit size;  //最小的FLASH输入输出单元大小

for NAND flashes - sub-page size;  //对于nand flash来说，子页大小

logical eraseblock size； //逻辑擦除块大小

参数获取的方式：

1)

* MTD partition size：从内核的分区表或cat /proc/mtd获得
* flash physical eraseblock size：从flash芯片手册或cat /proc/mtd
* minimum flash input/output unit size:

1) norflash 通常为1个字节

2)nandflash 一个页面

* sub-page size：通过flash手册获得
* logical eraseblock size：对于有子页的NANDFLASH来说，等于“物理擦除块大小-2页的大小”

2)也可通过ubi和mtd连接时的产生的信息获取

UBI: attaching mtd1 to ubi0

UBI: physical eraseblock size: 131072 bytes (128 KiB)

UBI: logical eraseblock size: 126976 bytes

UBI: smallest flash I/O unit: 2048

UBI: VID header offset: 2048 (aligned 2048)

UBI: data offset: 4096

UBI: max. sequence number: 0

UBI: volume 0 ("rootfs") re-sized from 1033 to 1930 LEBs

UBI: attached mtd1 to ubi0

UBI: MTD device name: "rootfs"

UBI: MTD device size: 248 MiB

UBI: number of good PEBs: 1972

UBI: number of bad PEBs: 12

UBI: number of corrupted PEBs: 0

UBI: max. allowed volumes: 128

UBI: wear-leveling threshold: 4096

UBI: number of internal volumes: 1

UBI: number of user volumes: 1

UBI: available PEBs: 0

UBI: total number of reserved PEBs: 1972

UBI: number of PEBs reserved for bad PEB handling: 38

UBI: max/mean erase counter: 1/0

UBI: image sequence number: 538007531

atmel\_spi f0004000.spi: Using dma0chan3 (tx) and dma0chan4 (rx) for DMA transfers

atmel\_spi f0004000.spi: Atmel SPI Controller at 0xf0004000 (irq 18)

UBI: background thread "ubi\_bgt0d" started, PID 443

UBIFS: mounted UBI device 0, volume 0, name "rootfs"

UBIFS: file system size: 243666944 bytes (237956 KiB, 232 MiB, 1919 LEBs)

UBIFS: journal size: 9023488 bytes (8812 KiB, 8 MiB, 72 LEBs)

UBIFS: media format: w4/r0 (latest is w4/r0)

UBIFS: default compressor: lzo

UBIFS: reserved for root: 0 bytes (0 KiB)

VFS: Mounted root (ubifs filesystem) on device 0:11

对于sama5d3xek来说（16进制）：

MTD partition size; 0f800000（16进制 248Mb）

flash physical eraseblock size; 00020000（16进制 128Kb 131072 b）

minimum flash input/output unit size; 2048 byte

for NAND flashes - sub-page size; 2048 byte

logical eraseblock size； 126976 byte

注：131072-126976=4096

### 2.2使用mkfs.ubifs命令将某个文件夹制作为UBIFS镜像

具体命令为

root@ep-VirtualBox:/opt# mkfs.ubifs -r /opt/rootfs -m 2048 -e 126976 -c 1984 -o ubifs.img

以上命令的含义为将/opt/rootfs文件夹制作为UBIFS文件系统镜像，输出的镜像名为ubifs.img.

mkfs.ubifs相关参数说明：

-m 最小 I/O 单元大小，一般是页大小， 2048 byte

-r 是指定哪个文件系统作为文件系统。

-e 是可擦除逻辑块大小，一般等于“物理擦除块的大小 - 2\* 页大小” = 131072-2\*2048=126976

-c 是最大可擦除逻辑块总数，这个是从 ubinize.cfg 里面的 vol\_size / ubinize 里面的 -p 参数

= （960 \* 1024） / 256 == 3840

-o 生成的rootfs.ubi

-F 自动调整大小。

**至此，直接烧录发现用不了**。

-m 4096

-e 253952

-c 2016

***自制板：mkfs.ubifs -r /opt/test/rootfs -m 4096 -e 253952 -c 2016 -o ubifs.img***

### 2.3 ubinize转换镜像格式

使用ubinize命令可以将使用mkfs.ubifs命令制作的UBIFS文件系统镜像转换成可以直接在FLASH上烧写的格式（带有UBI文件系统镜像卷标）

root@ep-VirtualBox:/opt# ubinize -o rootfs.ubi -m 2048 -p 128KiB -s 512 -O 2048 ubinize.cfg

自制板：ubinize -o rootfs.ubi -m 4096 -p 256KiB -s 512 -O 4096 ubinize.cfg

ubinize.cfg文件内容如下：

[ubifs]

mode=ubi

image=ubifs.img

vol\_id=0

vol\_size=100MiB

vol\_type=dynamic

vol\_alignment=1

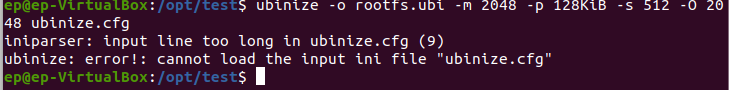
vol\_name=rootfs

vol\_flags=autoresize

注：vol\_size不能大于248，256-8=248, 8M为boostrap,uboot,kernel用，又由于坏块等，写248也不能用，此处应写小一点，如100，在测试时232也可以。

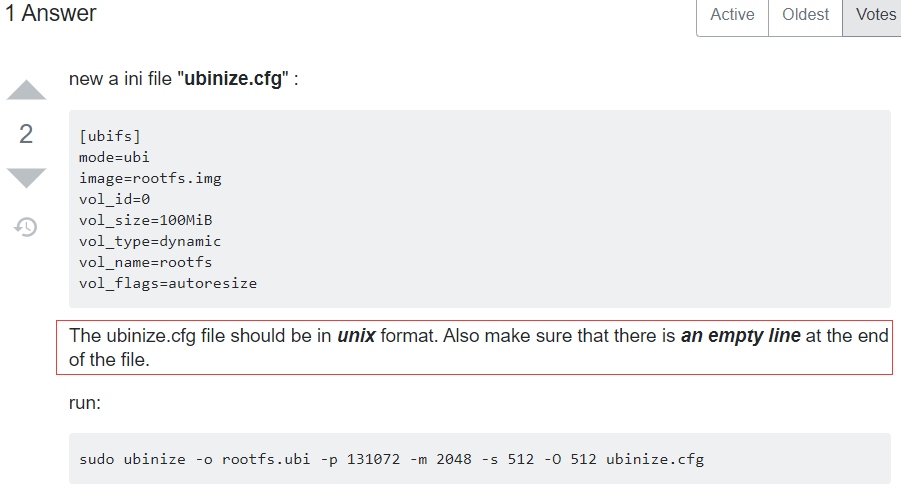
**至此制作的文件镜像rootfs.ubi烧录可用（sama5d3xek nandflash启动测试过）**

注：如果使用新的gcc，然后重新制作，出现：



需要将

**ubinize.cfg最后留出一行空行:**



**启动时相关信息如下:**

2 cmdlinepart partitions found on MTD device atmel\_nand

Creating 2 MTD partitions on "atmel\_nand":

0x000000000000-0x000000800000 : "bootstrap/uboot/kernel"

0x000000800000-0x000010000000 : "rootfs"

UBI: attaching mtd1 to ubi0

UBI: physical eraseblock size: 131072 bytes (128 KiB)

UBI: logical eraseblock size: 126976 bytes

UBI: smallest flash I/O unit: 2048

UBI: VID header offset: 2048 (aligned 2048)

UBI: data offset: 4096

UBI: max. sequence number: 7

UBI: attached mtd1 to ubi0

UBI: MTD device name: "rootfs"

UBI: MTD device size: 248 MiB

UBI: number of good PEBs: 1972

UBI: number of bad PEBs: 12

UBI: number of corrupted PEBs: 0

UBI: max. allowed volumes: 128

UBI: wear-leveling threshold: 4096

UBI: number of internal volumes: 1

UBI: number of user volumes: 1

UBI: available PEBs: 0

UBI: total number of reserved PEBs: 1972

UBI: number of PEBs reserved for bad PEB handling: 38

UBI: max/mean erase counter: 2/0

UBI: image sequence number: 953068048

UBIFS: recovery completed

UBIFS: mounted UBI device 0, volume 0, name "rootfs"

UBIFS: file system size: 124563456 bytes (121644 KiB, 118 MiB, 981 LEBs)

UBIFS: journal size: 9023488 bytes (8812 KiB, 8 MiB, 72 LEBs)

UBIFS: media format: w4/r0 (latest is w4/r0)

UBIFS: default compressor: lzo

UBIFS: reserved for root: 0 bytes (0 KiB)

VFS: Mounted root (ubifs filesystem) on device 0:11

相关命令说明(关于命令的解释说明有待完善)

# SSH功能的添加（基于busybox文件系统）

使用busybox为基于ARM的板卡定制了一个极简单的根文件系统，由于busybox仅支持telnet而不支持ssh，本文将详细描述如何交叉编译dropbear、openSSH并将其移植到目标板卡上。

## 1 dropbear的移植

所需软件包：

[zlib-1.2.8](http://prdownloads.sourceforge.net/libpng/zlib-1.2.8.tar.gz?download)

[dropbear-2019.78.tar.bz2](https://matt.ucc.asn.au/dropbear/releases/dropbear-2019.78.tar.bz2)

### 1.1 编译zlib-1.2.8

下载zlib-1.2.8,将其放在/opt/test目录下（个人根据自己本机情况放置），命令行进入/opt/test目录，将其解压：

***tar xvzf zlib-1.2.8.tar.gz***

在/opt/test目录下新建一个build，在build下建立zlib和dropbear目录，用于将zlib-1.2.8和dropbear的编译结果存放于该目录下。

***mkdir -pv build/zlib build/dropbear***

***cd zlib-1.2.8***

进入解压缩后的zlib目录，配置zlib:

***./configure --prefix=/opt/test/build/zlib***

修改生成的Makefile文件

***CC=/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-gcc***

***AR=/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-ar***

***RANLIB=/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-ranlib***

***LDCONFIG=/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-ldconfig***

***LDSHARED=$(CC) -shared -Wl,-soname,libz.so.1,--version-script,zlib.map***

***CPP=$(CC) -E***

编译并安装

***make && make install***

在/opt/test/build/zlib下就生成了zlib库的相关文件

### 1.2 编译dropbear

回到/opt/test,解压缩dropbear,进入dropbear目录，配置dropbear:

***./configure --prefix=/opt/test/build/dropbear --with-zlib=/opt/test/build/zlib CC=/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-gcc --host=arm***

编译并安装

***make***

***make scp***

***make install***

拷贝scp到安装目录

***cp scp /opt/test/build/dropbear/bin***

### 1.3移植到目标板

拷贝库及可执行文件到用busybox创建的rootfs文件系统（此时还未打包,本作者将/opt/rootfs拷贝了一份到/opt/test下，即/opt/test/rootfs用于测试更改），

***cp -frP /opt/test/build/zlib/lib/\* /opt/test/rootfs/usr/lib***

***cp -frP /opt/test/build/dropbear/bin/\* /opt/test/rootfs/usr/sbin***

***cp -frP /opt/test/build/dropbear/sbin/\* /opt/test/rootfs/usr/sbin***

**注：还需要拷贝**

**/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/arm-linux-gnueabihf/lib/libgcc\_s.so.1到 /opt/test/rootfs/usr/lib下，该库在dropbearkey生成密钥时使用到了。**

将/opt/test/rootfs按之前方法重新打包,烧入板件sama5d3x-EK.重启板件

在SecureCRT超级终端上，创建dropbear配置目录，并生成密钥

***mkdir /etc/dropbear***

***cd /etc/dropbear***

***dropbearkey -t rsa -f dropbear\_rsa\_host\_key***

***dropbearkey -t dss -f dropbear\_dss\_host\_key***

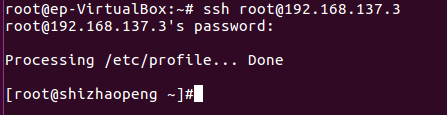
添加系统启动服务：

***vi /etc/init.d/rcS***

添加

***/usr/sbin/dropbear***

重启目标板，用PC通过SSH远程登陆即可（用已有的账号密码）：



注：后面可通过修改开发板上的/etc/init.d/rcS文件自动判断是否有ssh密钥，没有就产生，有就不产生，然后自动启动dropbear。（这是后来做的，可参考sama5d3xek的文件系统中的/etc/init.d/rcS的做法）

### 1.4遇到的问题

#### 1.4.1 ssh连接问题

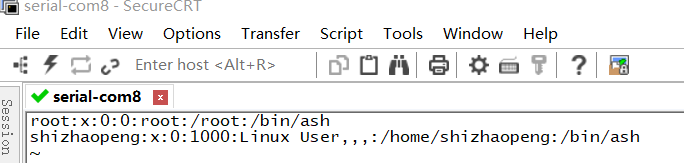
在嵌入式arm板输入：

#dropbear -E

[746] Jan 01 03:30:41 Running in background （提示在后台运行了）

PC通过SSH连接时，ARM板总是提示：user 'root' has invalid shell, rejecte

最后通过修改/etc/passwd下的shell解决。出现问题时是因为配置的shell是ash.将其改为sh后重启板件，即可通过ssh登陆了。



以下是通过查找资料的说明：

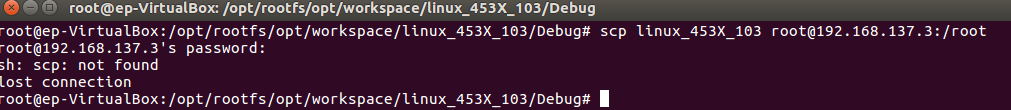
输入正确的密码，但总是登录不成功（在目标板上的终端提示：user 'root' has invalid shell, rejected），可能是由于 root 帐号没有设置正确的登录shell。各大多数程序一样，Dropbear 只允许 /etc/passwd 中列举的用户登录，并且在/etc/passwd中该用户需要设置正确的shell（/bin/sh 或 /bin/csh）。有些busybox用的是/bin/bash，所以被Dropbear拒绝了。这里我们使用 /bin/sh 以后，就可以正常登录了。

问题解决参考：

<https://blog.csdn.net/haifengid/article/details/51724748>

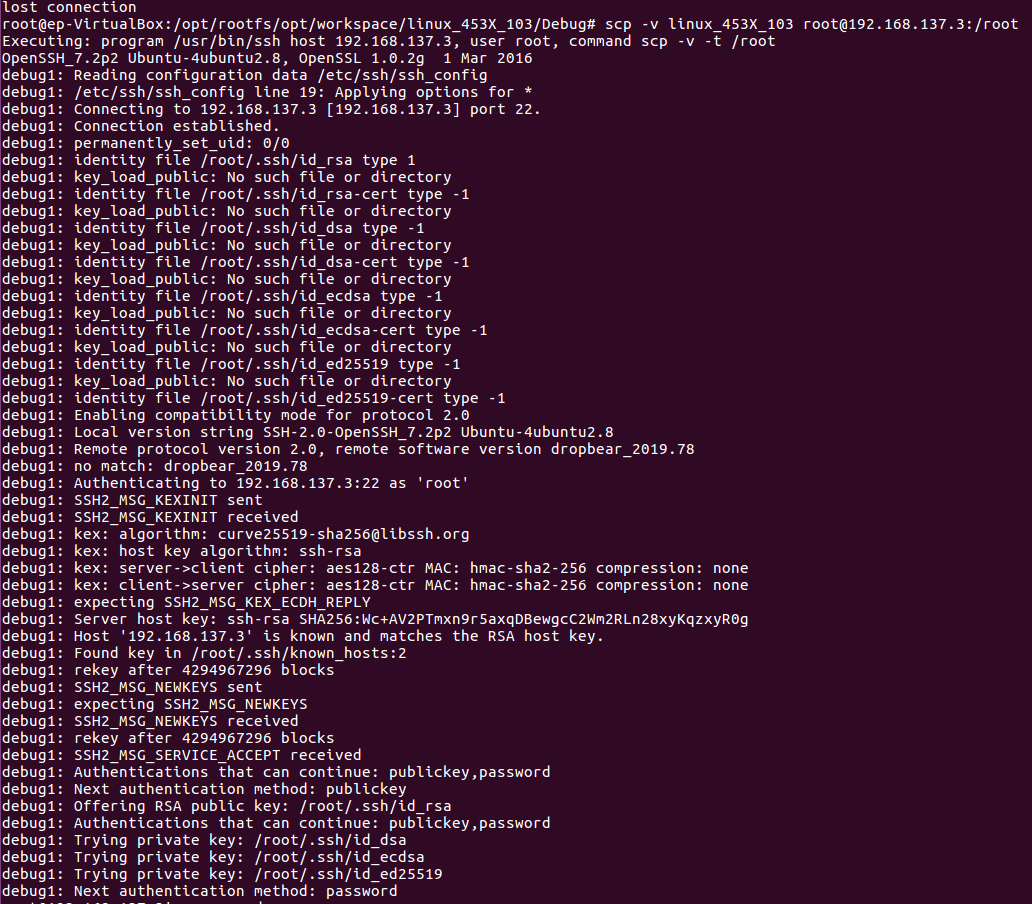
#### 1.4.2 scp传输文件问题

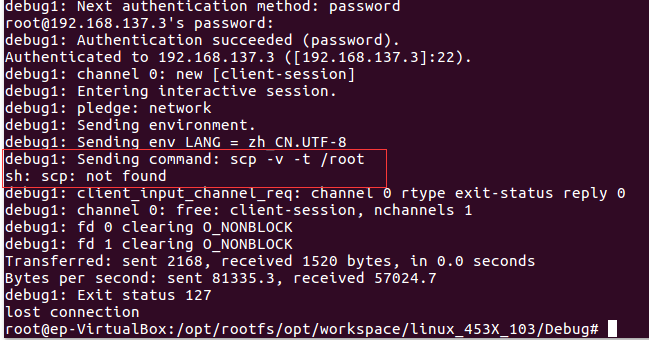
虽然将dropbear成功的移植到了开发板，且已经可以通过ssh连接登陆。但当我通过eclipse RSE插件的远程文件系统去拷贝编译好的文件到系统板时，总是拷贝出错。通过PC的命令行查看：



提示scp未发现。但是PC端肯定没问题，问题就出在了arm板的scp.

继续用-v选项查看具体信息：





发现问题出在了远端sh: scp :not found

猜测是sh找不到scp，

***故将开发板上/usr/sbin/scp拷贝到/bin下：cp /usr/sbin/scp /bin***

重新测试发现成功。

## openSSH的移植

所需软件包：

[zlib-1.2.8](http://prdownloads.sourceforge.net/libpng/zlib-1.2.8.tar.gz?download)

[openssl-1.1.1f.tar.gz](https://www.openssl.org/source/openssl-1.1.1f.tar.gz)

[openssh-8.2p1.tar.gz](https://cdn.openbsd.org/pub/OpenBSD/OpenSSH/portable/openssh-8.2p1.tar.gz)(openssh要找可移植版本,解压后根目录带configure文件)

### 交叉编译zlib-1.2.8

参考dropbear部分的zlib-1.2.8编译

### 2.2交叉编译openssl

解压openssl源码，进入目录，命令行执行：

**./config no-asm -shared --prefix=/opt/test/build/openssl/ //配置，--prefix为安装目录**

**修改Makefile文件中的：**

**CROSS\_COMPILE=/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-**

配置完成后：手动修改生成的Makefile，并且找到有-m32（-m64）的地方，全删之（共2处）(因为使用的编译器版本过旧，此处删除-m32,需依据情况)

**make && make install**

### 2.3编译openssh

下载：https://cdn.openbsd.org/pub/OpenBSD/OpenSSH/portable/openssh-8.2p1.tar.gz

**./configure --host=arm-linux --prefix=/opt/test/build/openssh/ --with-zlib=/opt/test/build/zlib --with-ssl-dir=/opt/test/build/openssl --disable-etc-default-login --disable-strip CC=/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-gcc AR=/opt/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415\_linux/bin/arm-linux-gnueabihf-ar**

之后：

**make**

不需要make install。make install会产生一些问题。不过为了方便，我make install提取了生成的文件到/opt/test/build/openssh/目录。至此编译完成

### 2.4 将文件拷贝至busybox制作的未压缩文件系统目录下

假设busybox制作的根文件系统所在目录为：/opt/test/rootfs

首先进入/opt/test/rootfs目录，然后在以rootfs文件夹为根系统的文件系统下建立以下目录：

**/usr/local/openssh/bin**

**/usr/local/openssh/etc**

**/usr/local/libexec**

openSSH编译完成后，会生成很多可执行文件，我们将以下文件拷贝到开发机的固定位置(注：此处的开发机目录，即为前文busybox构建文件系统方法创建的/opt/test/rootfs下，等把openssh拷贝好了，重新打包，烧录到开发板，这样就支持ssh sftp scp了)

以下文件拷贝到**/opt/test/rootfs/usr/local/openssh/bin**目录下:

**scp sftp ssh ssh-add ssh-agent ssh-keygen ssh-keyscan**

**(注：即前面make install到的目录/opt/test/build/openssh/bin下的文件)**

**特别注意：dropbear编译出的scp和openssh编译出的scp不是一个，所以不能同时使用，如果设置了openssh的scp，会导致dropbear的scp老是传输文件时错误。**

以下文件拷贝到**/opt/test/rootfs/usr/local/openssh/etc**目录下：

**moduli ssh\_config sshd\_config**

**即/opt/test/build/openssh/etc下的文件**

以下文件拷贝到开发机**/opt/test/rootfs/usr/local/libexec**目录下

**sftp-server ssh-keysign**

**即/opt/test/build/openssh/libexec下的文件(有四个文件，我全拷贝了，令两个为ssh-pkcs11-helper ssh-sk-helper)**

**注意**： ***这里可能应该把sftp-server拷贝到/usr/libexec路径下，与sshd\_config配置文件的设置默认路径相匹配***。后来我在连接时找不到sftp-server，根据提示信息，**最终将libexec文件夹放在了/usr/下**

以下文件拷贝到开发机**/opt/test/rootfs/usr/sbin**目录下

**sshd**

后来我在将rootfs文件系统打包烧到开发板测试sftp时，提示我找不到**libcrypto.so.1.1**文件，所以去查找该文件发现是编译的openssl下的文件，故将

**/rootsf/test/build/openssl/lib**下的**libcrypto.so.1.1**提前放入**/opt/test/rootfs/usr/lib**下

添加

至此完成。

重新按照文件系统制作章节中的制作UBIFS文件系统镜像将文件打包，烧入开发板即可。

### 2.5 PC使用ssh登录时存在的问题（**permitrootlogin 问题**）

默认PC直接使用ssh登录时，直接使用root存在以下问题：

**/etc/ssh/ssh\_config: Bad configuration option: permitrootlogin 问题**

编辑PC端的/etc/ssh/ssh\_config，将permitrootlogin相关行隐掉。

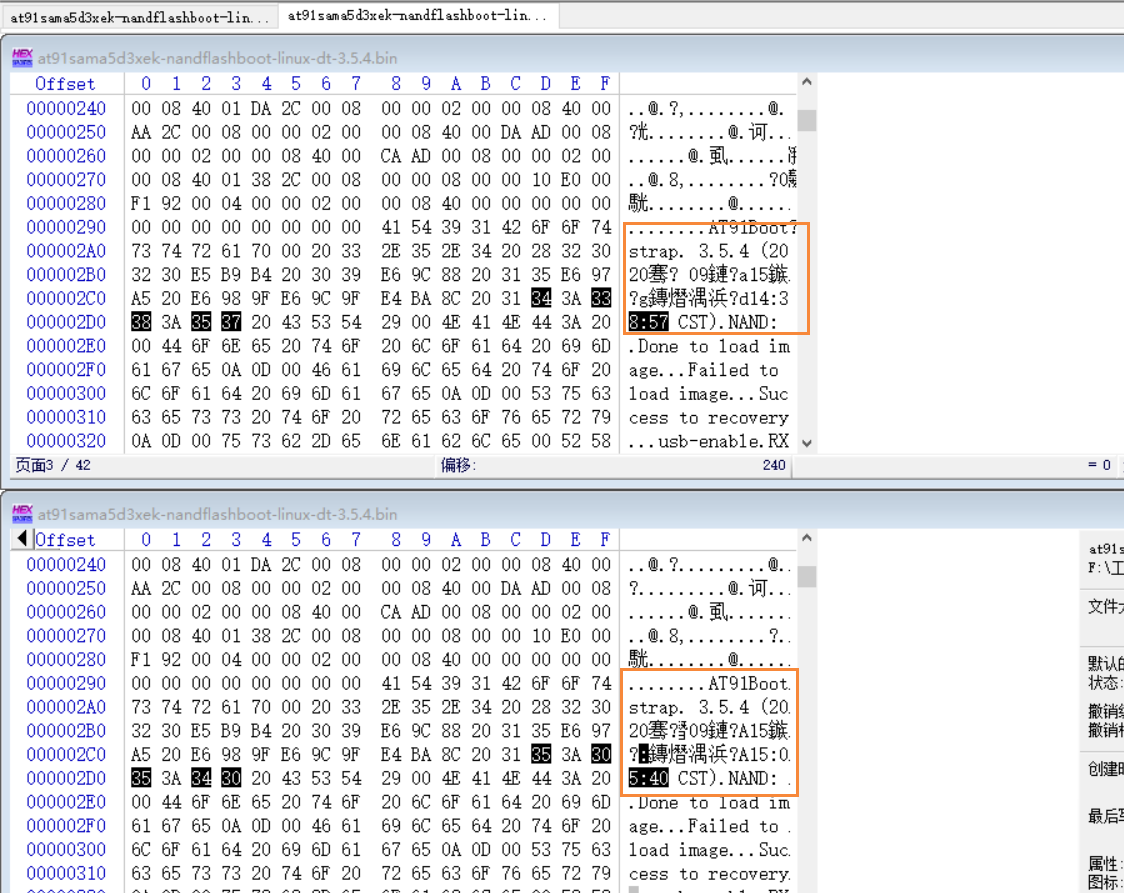
这个配置应该放到sshd\_config（服务器段配置）中，ssh\_config是客户端配置，所以直接把这行去掉就OK了。

# 版本控制相关

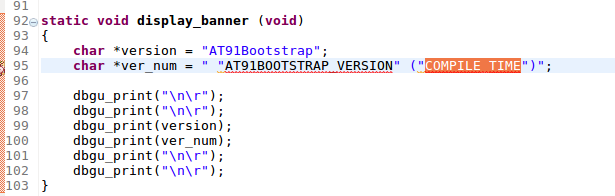
对于linux平台产生的bootstrap,sama5d35.dtb,uImage,rootfs.ubi文件要合成一个bin文件，需要保证，上述四个文件在不同的PC，不同时间点产生相同文件。这样通过crc等计算才能保证是相应版本的。

不同时间，不同PC编译出来的结果不同的原因分析：

## Bootstrap



对比发现是编译时间信息不同:

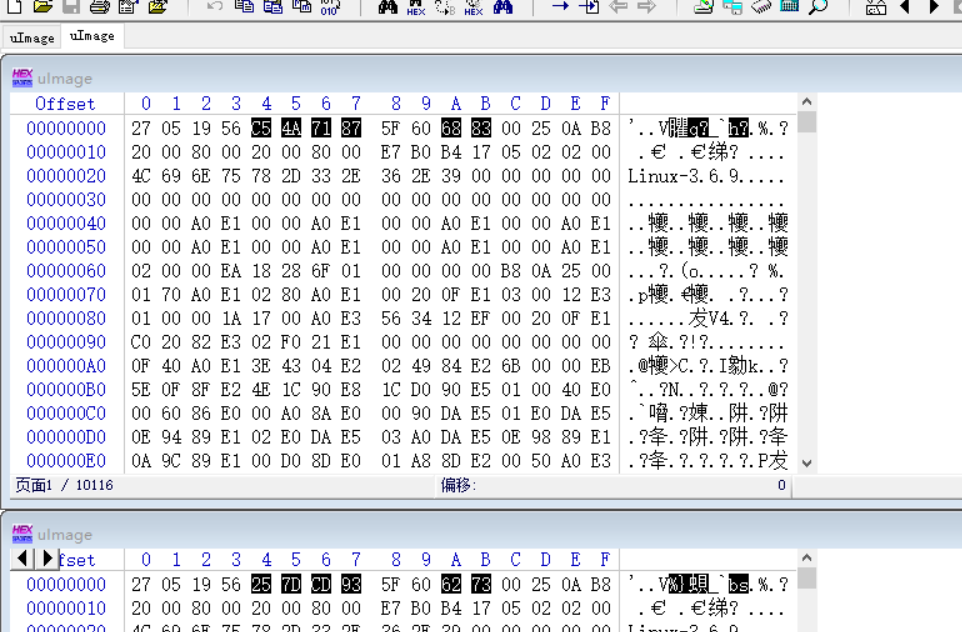


该函数在主函数中调用。如果不想产生差异，可以将main函数中对于此函数的调用引掉即可。

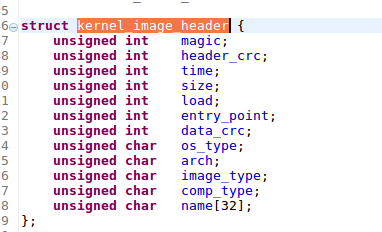
## sama5d35.dtb

目前为止相同

## uImage

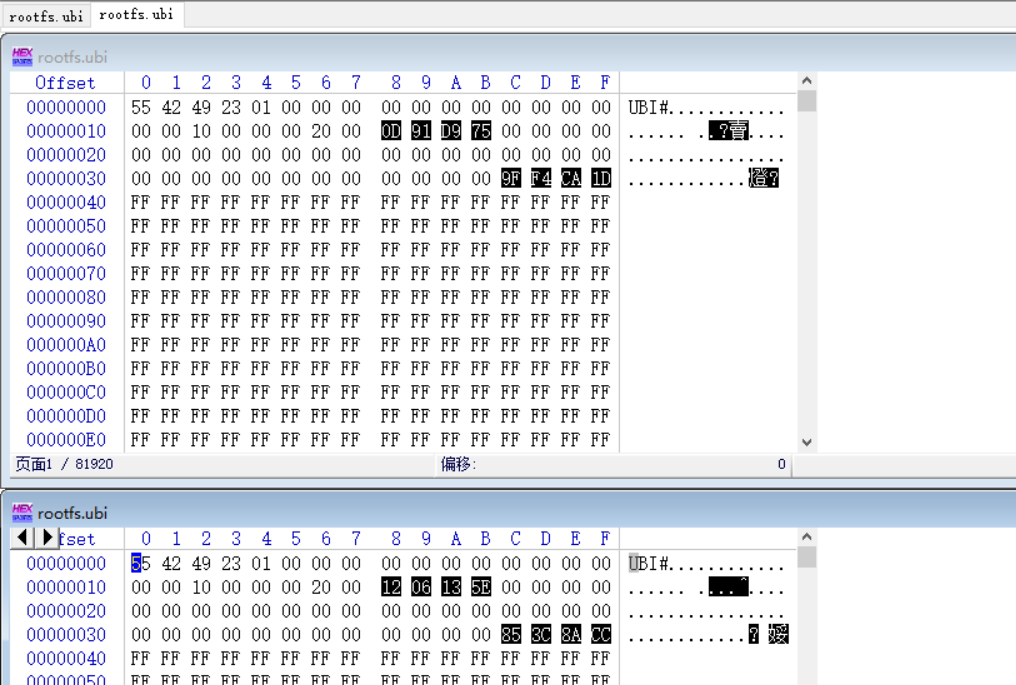


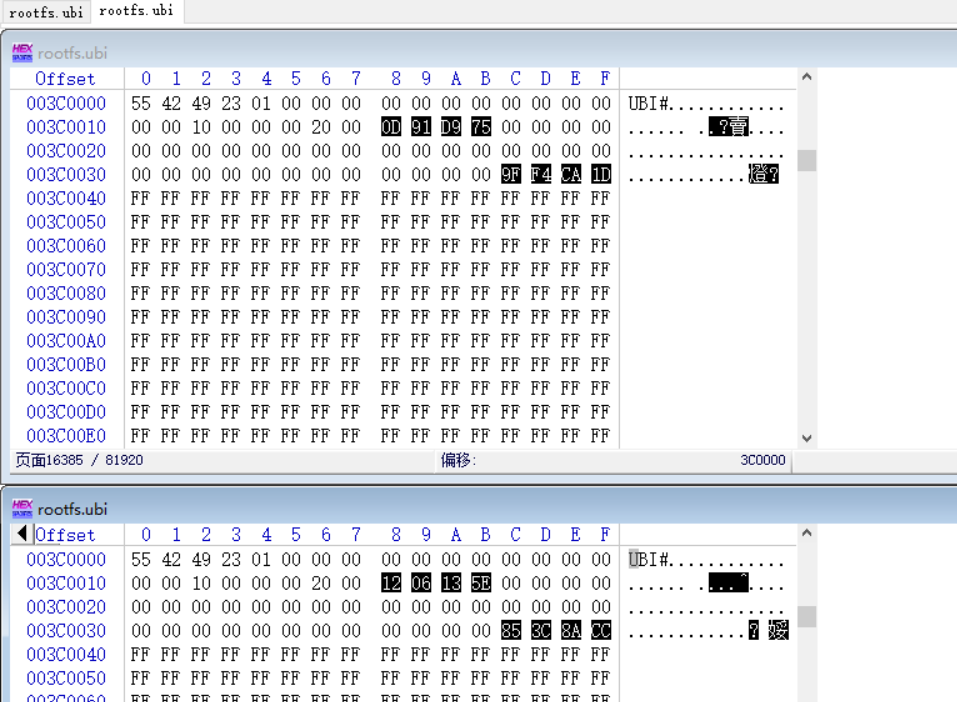
uImage不同的原因是由于时间不同导致的。在uImage的前64个字节是由zImage编译而来，对比了zImage发现相同。uImage的4-8个字节，9-12个字节存在差异。其中9-12为时间。由于时间不同，导致header\_crc也不同。

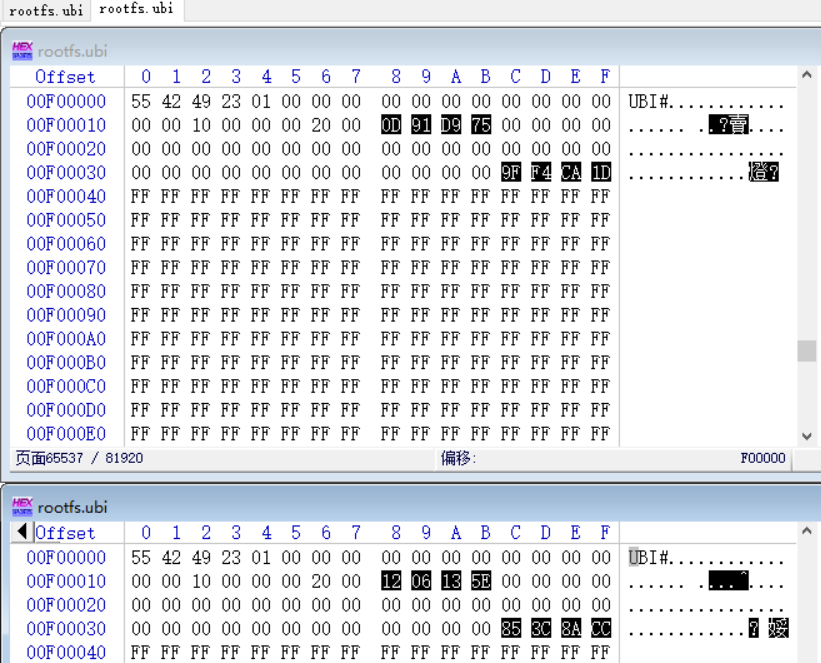
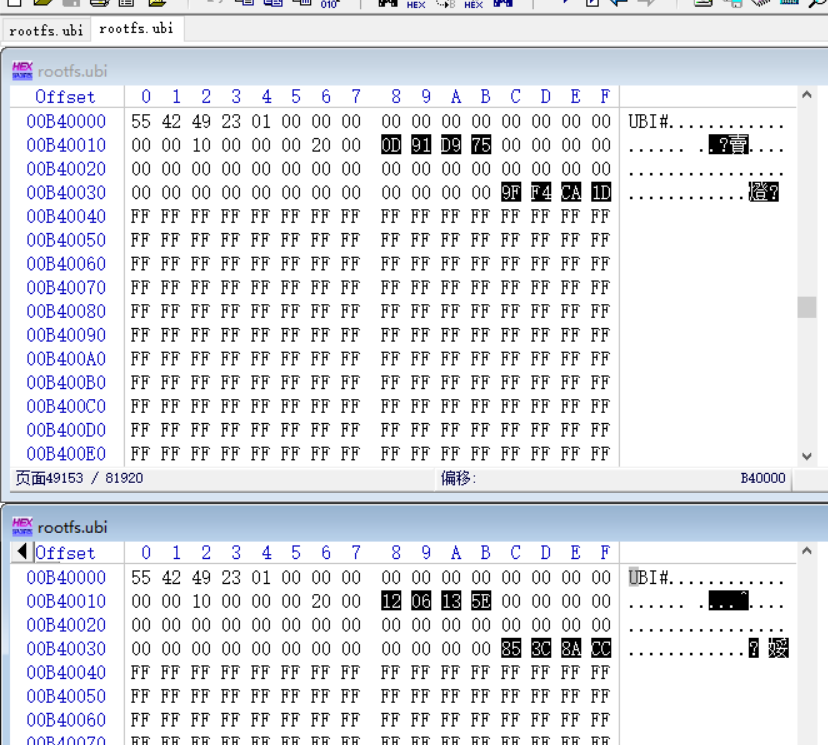


## rootfs.ubi

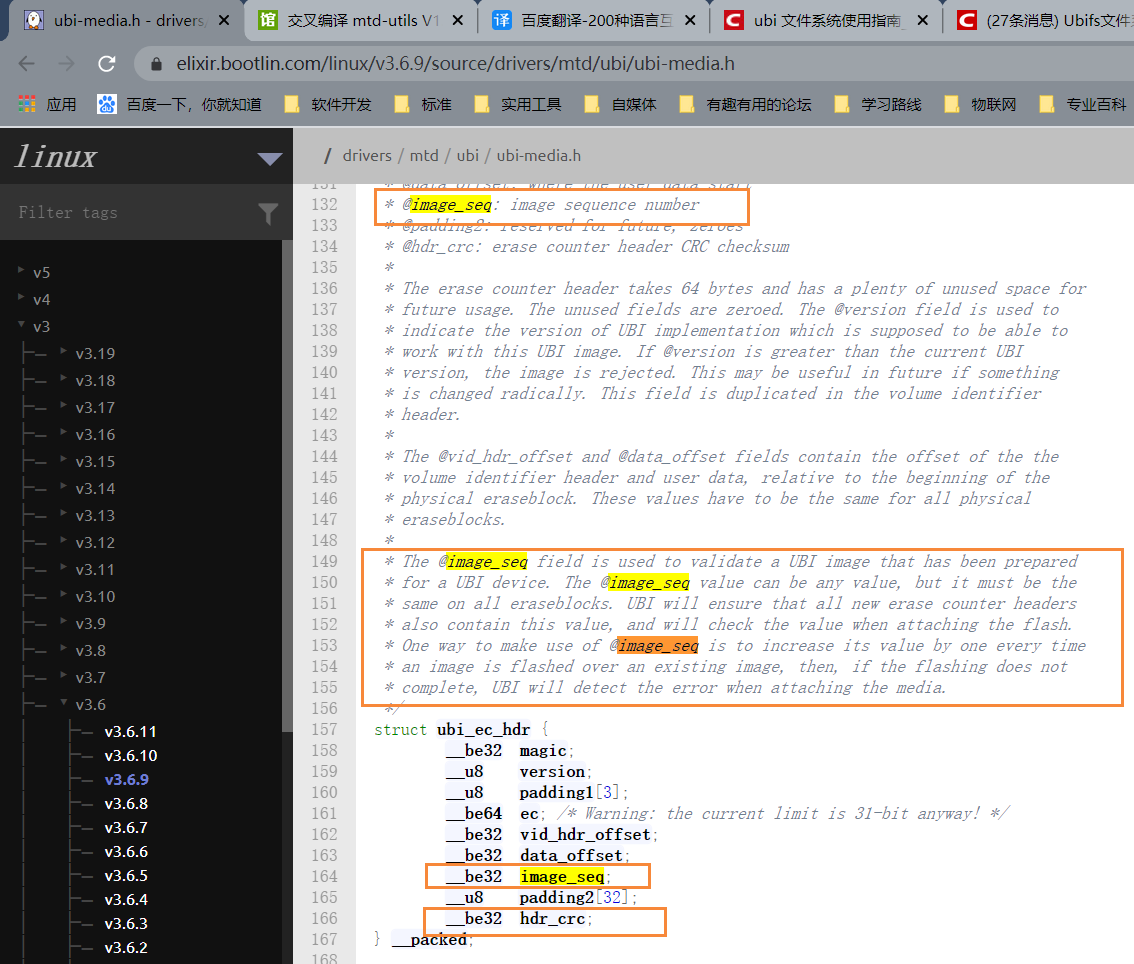
对比生成rootfs.ubi的ubifs.img文件，发现存在差异。分别在如下四个位置：

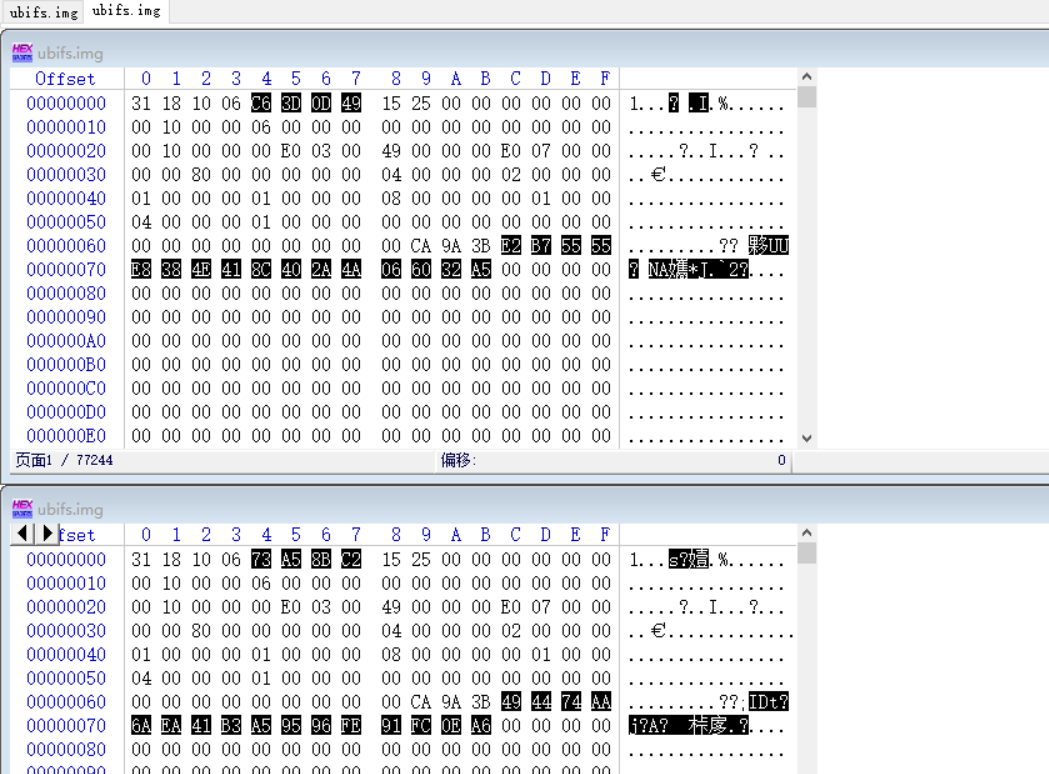


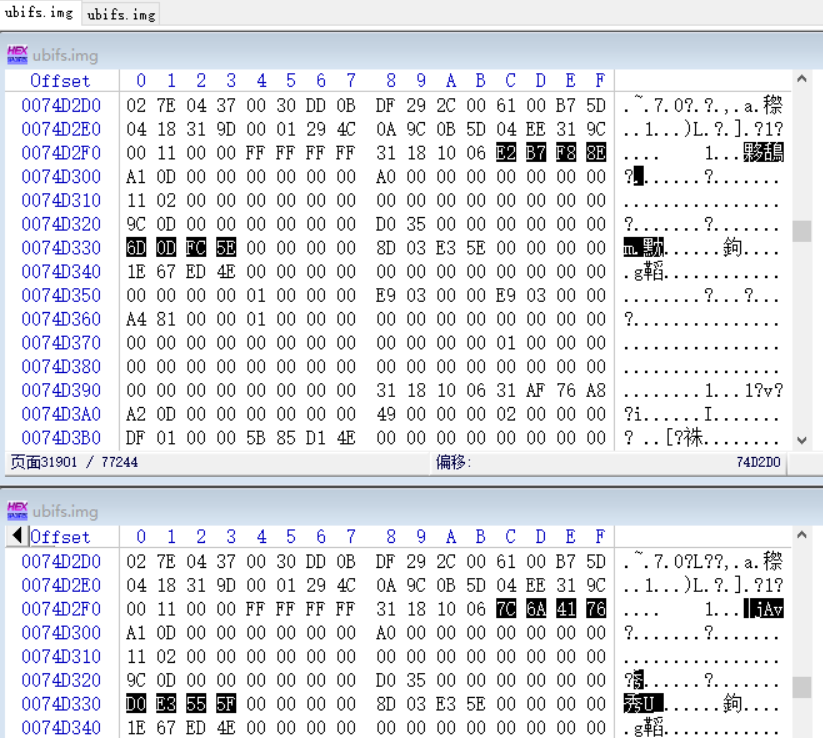




查看ubi说明文档可知相应字段为image\_seq，hdr\_crc字段。具体image\_seq如何产生，尚不明确。







# gcc更换引起的下载到arm上的程序无法运行问题解决

提示xxx.so.x version `xxxx' not found时，可以通过以下命令查看：

例如对于提示“/lib/libc.so.6: version `GLIBC\_2.28' not found“，可通过以下命令查看其内部版本：

**strings /lib/libc.so.6 |grep GLIBC\_ 查看**

参考文件系统制作STEP4: 构建lib目录，将相应的so文件复制过去即可。

后发现ssh不能使用，需要将ssh部分的软件使用新的gcc工具链重新编译放入。一定要注意工具链的一致性。

# 附录

## ARM指令

### A

### B

### C

### D

### E

### F

### G

### H

### I

### J

### K

### L

### M

### N

### O

### P

### Q

#### orr

语法：

ORR{S}{*cond*} *Rd*, *Rn*, *Operand2*

***S***: 是一个可选的后缀。如果指定了S，则条件标志将根据操作结果被更新。

***cond***: 是可选的条件

***Rd***: 是目的寄存器

***Rn***: 是存放第一个操作数的寄存器

***Operand***: 是灵活的第二个操作数

orr指令对Rn和Operand2中的值执行按位或运算。

在某些情况下，汇编程序可以用ORN替换ORR，或者用ORR替换ORN。阅读反汇编列表时请注意这一点。

**32位T32指令中PC的使用**

不能将PC（R15）用作Rd或ORR指令的任何操作数

**A32指令中PC和SP的使用**

可以将PC和SP与ORR指令一起使用，但是不建议使用。

如果将PC作为Rn,使用的值是将这个指令的地址加8。

如果将PC用作Rd:

* 执行分支到与结果对应的地址。
* 如果使用S后缀，请参见SUBS pc，lr指令。

在任何(带有寄存器控制移位的)数据处理指令中，都不能将PC用于任何操作数

**条件标志**

如果指定了S，则ORR指令：

* 根据这个结果更新N和Z标志
* 可以在计算Operand2期间更新C标志
* 不影响V标志

### R

### S

### T

#### tst

tst:逻辑处理指令，用于把一个寄存器的内容和另一个寄存器的内容或立即数进行按位的与运算，并根据运算结果更新CPSR中条件标志位的值。当前运算结果为1，则Z=0;当前运算结果为0，则Z=1

### U

### V

### W

### X

### Y

### Z

## ELF文件格式



### 1 组成

ELF文件由4部分组成，分别是**ELF头(ELF Header)**、**程序头表(Program header table)**、**节(section)**和**节头表(Section header table)**。实际上，一个文件中不一定包含全部内容，且他们的位置也未必如同所示这样安排，**只有ELF头的位置是固定的**，**其余各部分的位置、大小等信息由ELF头中的各项值来决定**。

### 2 ELF header

格式代码如下：

**#define EI\_NIDENT 16  
　　typedef struct{  
　　unsigned char e\_ident[EI\_NIDENT]; //16字节的e\_ident，其中包含用以表示ELF文件的字符，以及其他一些与机器无关的信**

**//息，开头的4字节值固定不变，为0x7f和ELF三个字符  
　　Elf32\_Half e\_type; //2它标识的是该文件的类型  
　　Elf32\_Half e\_machine; //2表明运行该程序需要的体系结构。  
　　Elf32\_Word e\_version; //4表明文件的版本  
　　Elf32\_Addr e\_entry; //4程序的入口地址  
　　Elf32\_Off e\_phoff; //4表示Program header table在文件中的偏移量(以字节计数)  
　　Elf32\_Off e\_shoff; //4表示Section header table在文件中的偏移量(以字节计数)  
　　Elf32\_Word e\_flags; //4对IA32而言，此项为0  
　　Elf32\_Half e\_ehsize; //2表示ELF header大小(以字节计数)  
　　Elf32\_Half e\_phentsize; //2表示Program header table中每一个条目的大小  
　　Elf32\_Half e\_phnum; //2表示Program header table中有多少个条目  
　　Elf32\_Half e\_shentsize; //2表示Section header table中的每一个条目的大小  
　　Elf32\_Half e\_shnum; //2表示Section header table中有多少个条目  
　　Elf32\_Half e\_shstrndx; //2包含Section名称的字符串是第几个节(从零开始计数)。  
　　}Elf32\_Ehdr;**

数据类型说明：

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **大小** | **对齐** | **用途** |
| **Elf32\_Addr** | **4** | **4** | **无符号程序地址** |
| **Elf32\_Half** | **2** | **2** | **无符号中等大小整数** |
| **Elf32\_Off** | **4** | **4** | **无符号文件偏移** |
| **Elf32\_Sword** | **4** | **4** | **有符号大整数** |
| **Elf32\_Word** | **4** | **4** | **无符号大整数** |
| **unsigned char** | **1** | **1** | **无符号小整数** |

### Program header

Program header描述的是一个段在文件中的位置、大小以及它被放进内存后所在的位置和大小。

**typedef struct {**

**Elf32\_Word p\_type; //当前Program header所描述的段section的类型**

**Elf32\_Off p\_offset; //段(section)的第一个字节在文件中的偏移**

**Elf32\_Addr p\_vaddr; //段的一个字节在内存中的虚拟地址**

**Elf32\_Addr p\_paddr; //在物理内存定位相关的系统中，此项是为物理地址保留**

**Elf32\_Word p\_filesz; //段在文件中的长度**

**Elf32\_Word p\_memsz; //段在内存中的长度**

**Elf32\_Word p\_flags; //与段相关的标志**

**Elf32\_Word p\_align; //根据此项值来确定段在文件及内存中如何对齐**

**}Elf32\_Phdr;**

## 网络命令

查看ssh端口是否占用

netstat -tnlp | grep :22

## linux启动说明

init的进程号是1（ps -aux），从这一点可以看到，init进程是系统所有进程的起点，Linux在完成核内引导以后，就开始运行init程序。

init程序需要读取配置文件/etc/inittab。inittab是一个不可执行的文本文件，它有若干行指令所组成。

### 理解Runlevel

Runlevel 0 是让init关闭所有进程并终止系统

Runlevel 1 是用来将系统转到单用户模式，单用户模式只能有系统管理员进入，在该模式下处理那些在有登录用户的情况下不能进行更改的文件，改runlevel的编号1也可以用S代替

Runlevel 2 是允许系统进入多用户的模式，但并不支持文件共享，这种模式很少应用。

Runlevel 3 是最常用的运行模式，主要用来提供真正的多用户模式，也是多数服务器的缺省模式

Runlevel 4 一般不被系统使用，用户可以设计自己的系统状态并将其应用到runlevel 4阶段，尽管很少使用，但使用该系统可以实现一些特定的登录请求.

Runlevel 5 是将系统初始化为专用的X Window终端。对功能强大的Linux系统来说，这并不是好的选择，但用户如果需要这样，也可以通过在runlevel启动来实现该方案

Runlevel 6 是关闭所有运行的进程并重新启动系统

## Inittab文件内容

在inittab文件中以#开头的所有行都是注释行。注释行有助于用户理解inittab文件，inittab文件中的值都是如下格式：

***label:runlevel:action:process***

label是1~4个字符的标签，用来标示输入的值。一些系统只支持2个字符的标签。鉴于此原因，多数人都将标签字符的个数限制在2个以内。该标签可以是任意字符构成的字符串，但实际上，某些特定的标签是常用的，在Red Hat Linux中使用的标签是：

**代码：**

　　id 用来定义缺省的init运行的级别

　　si 是系统初始化的进程

　　ln 其中的n从1~6,指明该进程可以使用的runlevel的级别

　　ud 是升级进程

　　ca 指明当按下Ctrl+Alt+Del时运行的进程

　　pf 指当UPS表明断电时运行的进程

　　pr 是在系统真正关闭之前，UPS发出电源恢复的信号时需要运行的进程

　　x 是将系统转入X终端时需要运行的进程

runlevel字段指定runlevel的级别。可以指定多个runlevel级别，也可以不为runlevel字段指定特定的值。

action字段定义了该进程应该运行在何种状态下：

**代码:**

　　boot 在系统启动时运行，忽略runlevel

　　bootwait 在系统启动时运行，init等待进程完成。忽略runlevel

　　ctrlaltdel 当Ctrl+Alt+Del三个键同时按下时运行，把SIGINT信号发送给init。忽略runlevel

　　initdefault 不要执行这个进程，它用于设置默认runlevel

　　kbrequest 当init从键盘中收到信号时运行。这里要求键盘组合符合KeyBoardSigral(参见/usr/share/doc/kbd-\*关于键盘组合的文档)

　　off 禁止进入，因此该进程不运行

　　once 每一个runlevel级别运行一次

　　ondemand 当系统指定特定的运行级别A、B、C时运行

　　powerfail 当init收到SIGPWR信号时运行

　　powerokwait 当收到SIGPWD信号且/etc/文件中的电源状态包含OK时运行

　　powerwait 当收到SIGPWD信号，并且init等待进程结束时运行

　　respawn 不管何时终止都重新启动进程

　　sysinit 在运行boot或bootwait进程之前运行

　　wait 运行进程等待输入运行模式

process字段包含init执行的进程，该进程采用的格式与在命令行下运行该进程的格式一样，因此process字段都以该进程的名字开头，紧跟着是 运行时，紧跟着是运行时要传递给该进程的参数。比如/sbin/shutdown -t3 -r now，该进程在按下Ctrl+Alt+Del时执行，在命令行下也可以直接输入来重新启动系统。

特殊目的的记录