



Android Porting Guide for Embedded Platforms

03/16/2009

Rel. 2.0

HSC Restricted

D -8, Infocity – II, Sector 33,

Gurgaon, Haryana.

India

PROPRIETARY NOTICE

All rights reserved. This publication and its contents are proprietary to Hughes Systique Corporation. No part of this publication may be reproduced in any form or by any means without the written permission of Hughes Systique Corporation, 15245 Shady Grove Road, Suite 330, Rockville, MD 20850.

Copyright © 2006 Hughes Systique Corporation

TABLE OF CONTENTS

<u>SECTION</u>	<u>PAGE</u>
1.0 INTRODUCTION	1
1.1 CONVENTIONS USED IN THIS DOCUMENT	1
1.2 SCREENSHOTS OF ANDROID PLATFORM PORTED ON MARVELL PXA 310 PDK.....	1
2.0 PREREQUISITES AND SETUP.....	3
2.1 ANDROID SOURCE CODE.....	3
2.2 LINUX SOURCE CODE.....	3
2.3 TARGET PLATFORM	4
2.4 SETUP CROSS DEVELOPMENT PLATFORM.....	5
3.0 PORTING STEPS.....	6
3.1 OVERVIEW	6
3.2 PATCHES	6
3.3 PORTING LINUX	8
3.4 STEPS TO ADD NEW PLATFORM IN BOOT LOADER.....	9
3.5 STEPS TO ADD NEW PLATFORM IN LINUX KERNEL.....	12
3.6 BURNING INTO FLASH.....	14
4.0 PORTING ANDROID.....	15
4.1 ANDROID ARCHITECTURE	15
4.2 ADD NEW PLATFORM SUPPORT IN ANDROID	16
4.3 SOURCE CODE PATCHES TO SUPPORT TARGET PLATFORM.....	18
4.4 ANDROID SOURCE CODE COMPILATION	18
4.5 BURNING INTO FLASH.....	19
5.0 APPENDIX.....	20
5.1 LINUX KERNEL – BOOT-UP DETAILS.....	20

LIST OF FIGURES

<u>FIGURE</u>	<u>PAGE</u>
Figure 1 Linux kernel boot up.....	2
Figure 2 Successful boot up of Android on Marvell Board	2
Figure 3 PXA310 Handheld Platform Development Kit [Littleton]	4
Figure 4 Cross development platform setup.....	5
Figure 5 Steps to build the portable Linux.....	8
Figure 7: - Android System Architecture.....	15
Figure 8 Adding new Platform Directory Structure	16
Figure 9 Android Running On the PXA 310 Handheld Platform Development Kit	19
Figure 6 Linux kernel event sequence.....	20

LIST OF TABLES

TABLE

PAGE

1.0 INTRODUCTION

Since the introduction of the open source Android platform for mobile phones by Google, there has been [significant interest](#) in the OEM community to also customize Android for other embedded platforms such as netbooks, set-top boxes, car dashboards and others. The advantage of making Android available to multiple device platforms would mean that an application developed for one device could easily be made available for another platform with minimal porting needs (consider for example, a TiVo recording client that is made available both on your set top box and one your phone – same application, same code but available on two devices to suit your mobility needs)

Android is a Dalvik Virtual Machine based software platform that runs on a Linux based kernel. Therefore, to port an Android platform, one needs to port the underlying Linux OS and then the Android platform SDK as well. This document explains how to port the Android platform to custom ARM based boards. ARM is one of the most popular platforms for embedded devices.

HSC has used the following components/environments for the porting activity, described in this document:

- Android source code release v1.0r
- Marvell PXA 310 Handheld Platform Development Kit (an ARM based board)
- Ubuntu 6.0 host environment
- MHLV Linux kernel version 2.6
- GNU build tools

Intended audiences of this document are developers who wish to port Android on new reference designs boards. It is assumed that audience has prior background of using linux platform.

1.1 Conventions used in this document

Text in bold font	Filenames are highlighted in bold font
Text in blue font	Comments and informational text is highlighted in blue font
Text in boxes	Commands and instructions are highlighted in boxes

1.2 Screenshots of Android platform ported on Marvell PXA 310 PDK

For porting Android onto a reference platform developers will have to achieve two major milestones. These are 1) porting and booting Linux on the reference design board 2) porting the Android application framework onto the reference design board. Both of these activities are largely independent, however, some customization might be required in the Android core libraries once the linux device d

This section captures these achievements, in the form of screenshots, viz-a-viz the Following are screenshots of milestones for porting Android on Marvell PXA 310 PDK platform.

Milestone 1: Successful boot-up of linux on new reference platform. Figure 1 captures screenshot of hyperterminal showing successful linux kernel boot up.

```
marvell - HyperTerminal
File Edit View Call Transfer Help
asoc: I2S HiFi <-> pxa3xx-ssp3 mapping ok
asoc: PCM Voice <-> pxa3xx-ssp4 mapping ok
ALSA device list:
#0: littleton (Micco)
Netfilter messages via NETLINK v0.30.
nf_conntrack version 0.5.0 (1024 buckets, 4096 max)
ip_tables: (C) 2000-2006 Netfilter Core Team
TCP cubic registered
NET: Registered protocol family 1
NET: Registered protocol family 17
Bluetooth: L2CAP ver 2.9
Bluetooth: L2CAP socket layer initialized
Bluetooth: SCO (Voice Link) ver 0.5
Bluetooth: SCO socket layer initialized
Bluetooth: RFCOMM socket layer initialized
Bluetooth: RFCOMM TTY layer initialized
Bluetooth: RFCOMM ver 1.8
Bluetooth: BNEP (Ethernet Emulation) ver 1.2
Bluetooth: HIDP (Human Interface Emulation) ver 1.2
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
XScale iMMXt coprocessor detected.
pxa2xx_v4l2ov2: PXA v4l2 overlay2 driver loaded for/dev/video1
input: pxa27x-keypad as /class/input/input1
pxa3xx-rtc pxa3xx-rtc: setting system clock to 2000-01-01 00:00:22 UTC (94668482)
eth0: link down
IP-Config: Complete:
  device=eth0, addr=192.168.1.101, mask=255.255.255.0, gw=255.255.255.255,
  host=192.168.1.101, domain=, nis-domain=(none),
  bootserver=192.168.1.100, rootserver=192.168.1.100, rootpath=
Freeing init memory: 284K
Warning: unable to open an initial console.
init: /init.rc: 116: user option requires a user id
init: cannot open '/initlogo.rle'
yaffs: dev is 32505858 name is "mtdblock2"
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.2, "mtdblock2"
-
```

Figure 1 Linux kernel boot up

Milestone 2: Successful boot-up of Android platform. Figure 2 captures image of successful boot-up of Android platform on Marvell PXA 310 PDK platform. Rest of the document captures how developers can cross both the milestones to port Android on any reference platform.



Figure 2 Successful boot up of Android on Marvell Board

2.0 PREREQUISITES AND SETUP

This chapter describes the prerequisites for the Android porting activity.

2.1 Android Source Code

The Android source code can be downloaded from <http://source.android.com> .
Instructions for download are available at <http://source.android.com/download>

Note: The instructions in this document refer to Android source code version: 1.0r.
While we believe subsequent versions should follow a similar porting methodology, the reader may need to customize the instructions slightly for different Android versions other than that listed here.

2.2 Linux Source Code

As of the date of authorship of the paper, Android uses Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. HSC used MHLV linux kernel BSP from kernel.org. MHLV stands for Monahans Linux version. Since Marvell uses Monahans processor, It's recommended to use MHLV linux kernel for PXA 310 board. MHLV Linux can be downloaded from the following link:
<http://git.kernel.org/?p=linux/kernel/git/ycmiao/pxa-linux-2.6.git;a=summary>

2.3 Target Platform

The minimum recommended requirements for porting Android on reference platforms are:

- 128 MB RAM
- 256 MB Flash Memory

Source: <http://source.android.com/release-features>

HSC used a Handheld Platform Development Kit (PDK) named "littleton" from [Marvell](#) as a target platform for porting Linux and android. The handheld PDK runs on a PXA310 processor, which is based on Intel's XScale architecture and the ARM instruction set.



Figure 3 PXA310 Handheld Platform Development Kit [Littleton]

The following components are built into the PDK form factor:

- Infra Red port
- Micro SD MMC Card Slot
- VGA Camera
- 2.4 VGA display
- Ear piece Receiver
- Directional Action buttons
- Soft Keys and Rotary Scroll Wheel
- Numeric Keypad
- USB for Charging and Audio

The debug board has a provision to connect to the host PC. It provides the connectivity to the host processor using following connections:

- JTAG Connection with parallel port. Typically used for reading and writing into the NAND flash memory
- Serial Connection: To connect the hyper terminal on the host PC.
- Ethernet Connection: To download the kernel and root file system images into the board SDRAM.

The PXA310 Handheld platform supports the following environment:

- Host PC with Ubuntu 6.06 distribution for Linux OS
or
- Host PC with Windows XP/2K/Vista OS
- arm-linux-gcc 4.1.1 with Embedded Application Binary Interface (EABI)
- glibc 2.5
- Linux Kernel 2.6.25
- NAND Flash (Micron, 128 MB)
- DDR SDRAM, 64 MB
- PXA310 Processor with Intel XScale Core

2.4 Setup cross development platform

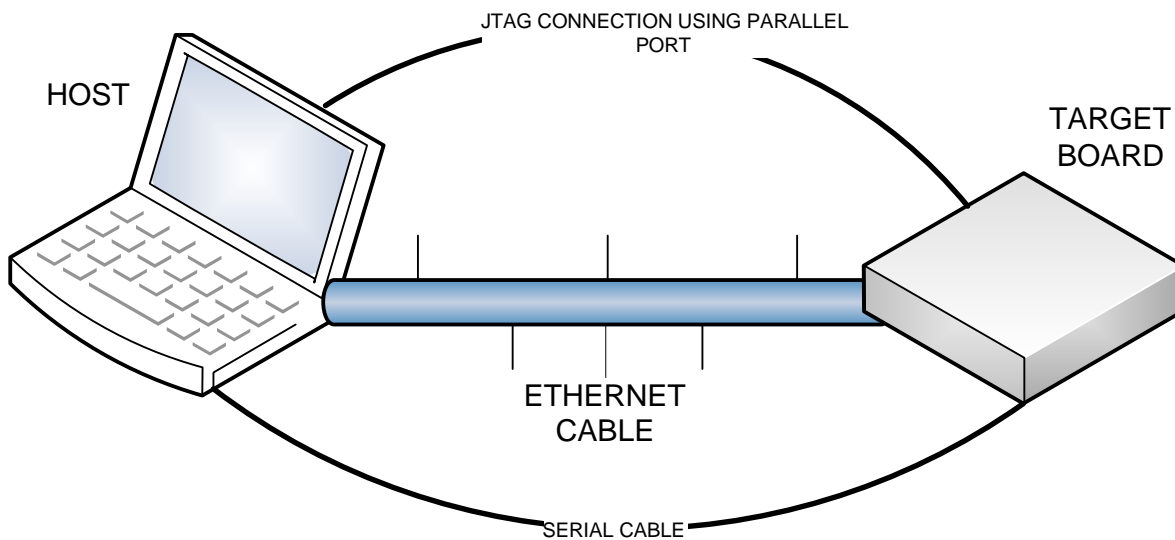


Figure 4 Cross development platform setup

The primary components in the cross compilation development environment were:

- Host system:- Ubuntu 6.0 Linux running on a x86 based host
- Target embedded system:- PXA310 Handheld PDK
- Connection:
 - Serial Cable (required for Hyperterminal)
 - Ethernet (required for network boot, network connectivity of the reference design board)
 - JTAG using parallel port (required to flash the boot loader image on the board).

For cross compilation of the source code, HSC used ARM tool chain with EABI support (refer section: 3.3.1) from GNU build toolchain.

3.0 PORTING STEPS

This chapter describes the process for porting Android (and Linux) onto a ARM based platform.

3.1 Overview

Porting Android on ARM based target platforms can be divided in two stages:

Stage 1: Porting Linux

Following are the main steps for porting Linux:

- o Download the patches of Linux Kernel for supporting PXA310 processor.
- o Prepare the ARM based tool chain for compilation of Linux source code.
- o Choose and Configure boot loader as per target board.
- o Compile the boot loader.
- o Configure Linux kernel as per target platform (this may require developing some additional device drivers or customization of the existing device drivers for the reference design board) .
- o Compile the Linux kernel source
- o Burn compressed Linux image on the target platform

Stage 2: Porting Android

Following are the main steps for porting Android:

- o Download the patches for Android
- o Update RIL (Radio Interface Library) for target platform
- o Update Board specific components such as Codec, Camera, Audio, Wifi, Power Management, Bluetooth etc.
- o Compile Android source code
- o Burn system image on Target platform

Subsequent sections will describe each of the stages in more detail.

3.2 Patches

3.2.1 Linux Kernel Patches

There are several patches available at <http://www.mail-archive.com/git-commits-head@vger.kernel.org/> which to be applied on Linux Kernel for supporting various features on PXA310 processor. Few snapshots of patches are given below.

- **pxa: clear RDH bit after any reset:-** According to PXA300/310 and PXA320 Developer manuals, the ASCR[RDH] "bit needs to be cleared as part of the software initialization coming out of any reset and coming out of D3". The latter requirement is addressed by commit "c4d1fb627ff3072", as for the former (coming out of any reset), the kernel relies on boot loaders and assumes that RDH bit is cleared there. Though, not all bootloaders follow the rule so we have to clear the bit in kernel. We clear the RDH bit in pxa3xx_init() function since it is always invoked after any reset. We also preserve D1S, D2S and D3S bits from being cleared in case we invoke pxa3xx_init() function not from normal hardware reset (e.g. kexec scenario), so these bits can be properly referenced later.
- **pxa: introduce sysdev for pxa3xx static memory controller:-** Introduce a sysdev for pxa3xx static memory controller, mainly for register saving/restoring in PM
- **pxa: add preliminary suspend/resume code for pxa3xx:** clear RDH bit after resuming back from D3, otherwise, the multi function pins will retain the low power state and save/restore essential system registers
- **pxa: introduce sysdev for IRQ register saving/restoring:** This patch introduce sysdev for IRQ register saving and restoring.

- **pxa: Add PXA3 standby code hooked into the IRQ wake scheme:-** Wakeup sources on PXA3 are enabled at two levels. First, the MFP configuration has to be set to enable which edges a specific pin will trigger a wakeup. The pin also has to be routed to a functional unit. Lastly, the functional unit must be enabled as a wakeup source in the appropriate AD*ER registers (AD2D0ER for standby resume. This doesn't fit well with the IRQ wake scheme - we currently do a best effort conversion from IRQ numbers to functional unit wake enable bits. For instance, there's several USB client related enable bits but there's no corresponding IRQs to determine which you'd want. Conversely, there's a single enable bit covering several functional units.
- **pxa: program MFPs for low power mode when suspending:-** Hook the MFP code into the power management code so that the MFPs can be reconfigured when suspending and resuming. However, note the FIXME low power mode MFP configuration may depend on the system state being entered. Also note that we have to clear any detected edge events prior to entering a low power mode - otherwise we immediately wake up
- **pxa: make MFP configuration processor independent:-** There are two reasons for making the MFP configuration to be processor independent, i.e. removing the relationship of configuration bits with actual MFPR register settings:
 1. power management sometimes requires the MFP to be configured differently when in run mode or in low power mode.
 2. for future integration of pxa{25x,27x} GPIO configurations

The modifications include:

 1. Introducing of processor independent MFP configuration bits, as defined in [include/asm-arm/arch-pxa/mfp.h]:
 - bit 0.. 9 - MFP Pin Number (1024 Pins Maximum)
 - bit 10..12 - Alternate Function Selection
 - bit 13..15 - Drive Strength
 - bit 16..18 - Low Power Mode State
 - bit 19..20 - Low Power Mode Edge Detection
 - bit 21..22 - Run Mode Pull State
 - and so on,
 2. Moving the processor dependent code from mfp.h into mfp-pxa3xx.h
 3. Cleaning up of the MFPR bit definitions
 4. Mapping of processor independent MFP configuration into processor specific MFPR register settings is now totally encapsulated within pxa3xx_mfp_config()
 5. Using of "unsigned long" instead of invented type of "mfp_cfg_t" according to Documentation/CodingStyle Chapter 5, usage of this in platform code will be slowly removed in later patches
- **pxa: remove un-used pxa3xx_mfp_set_xxx() functions:-** pxa3xx_mfp_set_xxx() functions are originally provided for overwriting MFP configurations performed by pxa3xx_mfp_config(), the usage of such a dirty trick is not recommended, since there is currently no user of these functions, they are safely removed
- **To add support for Android on Linux Kernel:-** First download the matching version of Android kernel source. Besides complete kernel source this will contain Android specific changes. Second download the linux kernel and diff both kernels to get Android related changes.

Above are the few details of patches which we have used, there are lots of patches which we need to apply on Linux Kernel.

3.2.2 Android Patches

The following lists of patches are applied to android to support the Marvell PXA 310.

- **opencore-update-ipp-codec-libraries.patch:-** This patch provides the updated library of IPP codec for android.
- **frameworks_base_camera.patch:-** This patch provides to add camera hardware into android code.
- **frameworks_base_libs.patch:-** This patch provides the libraries for Audio hardware into android code.

- **frameworks_core_jni.patch**:- This patch provides the Bitmap processing and PIM processing based JNI.
- **framework_base_ui.patch** : This patch provides UI functions to get the surface visible region coordinate and to check the surface is on top or not.
- **hardware_ril.patch**:- This patch provides the radio interface library for the PXA310 handheld PDK.
- **libhardware_legacy_wifi.patch**:- This patch provides the WiFi hardware interface in the android code.
- **opencore_codec_v2.patch**:- This patch provides the support of codec version 2.
- **system_core.patch**: This patch provides the support of Bluetooth.
- **wpa_supplicant.patch**: This patch provides to support the WPA supplicant.
Source : Marvell extranet site : URL <https://www.marvell.com/login/index.jsp>

3.3 PORTING LINUX

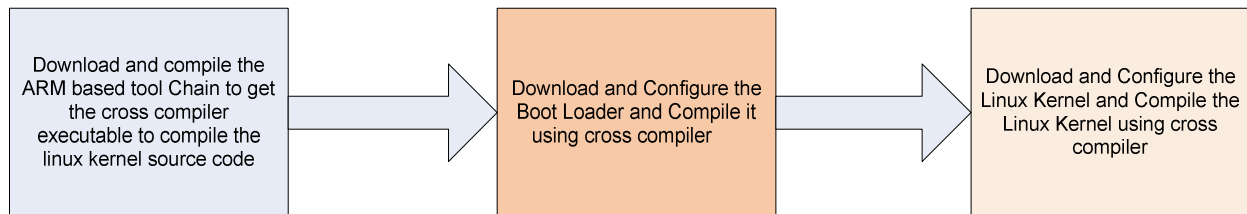


Figure 5 Steps to build the portable Linux

3.3.1 Steps to build ARM based Tool Chain for compiling Linux

The following steps are required to build the ARM based tool chain on Ubuntu 6.0 Linux distribution (for corresponding commands for each step, please refer to URL: <http://www.kegel.com/crosstool/crosstool-0.43/doc/crosstool-howto.html>) :

- Download the following GNU tool-chain software packages
 - o binutils-2.16.92.tar.bz2 : can be obtained from <ftp://ftp.gnu.org/pub/gnu/binutils/>
 - o gcc-4.1.1.tar.bz2 : can be obtained from <ftp://ftp.gnu.org/pub/gnu/gcc/>
 - o glibc-2.5.tar.gz : can be obtained from <ftp://ftp.gnu.org/pub/gnu/glibc/>
 - o glibc-ports-2.5.tar.bz2 : can be obtained from <ftp://gcc.gnu.org/pub/glibc/snapshots>
 - o linux-2.6.14.6.tar.bz2 can be obtained from <http://www.kernel.org/pub/linux/kernel/v2.6/>
 - o Respective patches – If present with respective reference platform site.
- Install source code of the binary utilities binutils-2.16.92 which includes cross-assembler, cross-loader and other cross-utilities.
- Recompile and install cross-utilities with Linux
- Install source code of the compiler gcc 4.1.1
- Recompile and install for cross-compiler.

Following files should be generated after successful compilation of cross tool chain:

Executables after building Cross Compilation for Arm Processor	
<i>arm-iwmmxt-linux-gnueabi-addr2line</i>	<i>arm-iwmmxt-linux-gnueabi-gcc</i>
<i>arm-iwmmxt-linux-gnueabi-objdump</i>	<i>arm-linux-ar</i>
<i>arm-linux-gcc-4.1.1</i>	<i>arm-linux-ranlib</i>
<i>arm-iwmmxt-linux-gnueabi-ar</i>	<i>arm-iwmmxt-linux-gnueabi-gcc-4.1.1</i>
<i>arm-iwmmxt-linux-gnueabi-ranlib</i>	<i>arm-linux-as</i>
<i>arm-linux-gccbug</i>	<i>arm-linux-readelf</i>
<i>arm-iwmmxt-linux-gnueabi-as</i>	<i>arm-iwmmxt-linux-gnueabi-gccbug</i>
<i>arm-iwmmxt-linux-gnueabi-readelf</i>	<i>arm-linux-c++</i>
<i>arm-linux-gcov</i>	<i>arm-linux-size</i>

<i>arm-iwmmxt-linux-gnueabi-c++</i>	<i>arm-iwmmxt-linux-gnueabi-gcov</i>
<i>arm-iwmmxt-linux-gnueabi-size</i>	<i>arm-linux-c++filt</i>
<i>arm-linux-ld</i>	<i>arm-linux-strings</i>
<i>arm-iwmmxt-linux-gnueabi-c++filt</i>	<i>arm-iwmmxt-linux-gnueabi-ld</i>
<i>arm-iwmmxt-linux-gnueabi-strings</i>	<i>arm-linux-cpp</i>
<i>arm-linux-nm</i>	<i>arm-linux-strip</i>
<i>arm-iwmmxt-linux-gnueabi-cpp</i>	<i>arm-iwmmxt-linux-gnueabi-nm</i>
<i>arm-iwmmxt-linux-gnueabi-strip</i>	<i>arm-linux-g++</i>
<i>arm-linux-objcopy</i>	<i>arm-iwmmxt-linux-gnueabi-g++</i>
<i>arm-iwmmxt-linux-gnueabi-objcopy</i>	<i>arm-linux-addr2line</i>
<i>arm-linux-gcc</i>	<i>arm-linux-objdump</i>

3.4 Steps to Add New Platform in Boot loader

The Linux kernel makes some fundamental assumptions when it gets control from a bootloader. Most important among them is that the bootloader must have initialized the DRAM controller. Linux assumes that the system RAM is present and fully functional. Therefore, the boot loader should also initialize the system memory map. This is usually done via a set of processor registers.

In summary Linux requires a boot loader to perform machine specific initialization. The initialization process performs the following tasks:

- Configures the system memory
- Determines the machine type and passes the information to the kernel
- Initializes other boot parameters to pass information to the kernel
- Loads the kernel image at the correct memory address
- Loads the kernel with the appropriate register values.

There are several boot loaders to choose from, for e.g.

1. Blob boot loader (<http://www.lartmaker.nl/lartware/blob/>)
2. U-boot (<http://www.denx.de/wiki/U-Boot>)
3. Redboot(<http://sourceware.org/redboot/>) etc).

HSC used the Blob boot loader for booting PXA310 handheld device.

The following steps were executed to add the new platform support in the Blob boot loader (details for each step will follow in the next section):

- Add machine definition to **configure.in**
- Add machine to **include/blob/arch.h**
- Add machine dependent architecture header file in **include/blob/arch/new_platform.h**. This .h file contains the information about the OFFSET used in RAM for kernel and root file systems.
- Add machine dependent architecture header file to **Makefile.am** in **include/blob/arch** directory.
- Add machine dependent source file in **src/blob**.
- Add machine dependent source file to **Makefile.am** in **src/blob**
- Add architecture number to **include/blob/linux.h**

3.4.1 Porting Blob Boot Loader to PXA310 Handheld PDK

Following are the steps to add PXA310 (target platform) handheld device support in the blob boot loader.

1. Add the machine definition to **configure.in**

```
Littleton)
    board_name="Marvell Littleton"
    BLOB_PLATFORM_OBJS="littleton.o setusbid.o"
    BLOB_FLASH_OBJS=""
    use_cpu="monahans"
    use_lcd="no"
    board_cflags="-DLITTLETON"
    BLOB_NETWORK_DRIVER_OBJS = [set the Network driver objects based
on USBDNET Support. In case usbdnet is supported define
CONFIG_USBDNET_SUPPORT as
AC_DEFINE(CONFIG_USBDNET2_SUPPORT,1,"USBDNET2 support") ]
```

There are two modes by which kernel images can be loaded to target platform:

- Ethernet
- USBDNET – Ethernet over USB.

We have used Ethernet. One can use either of these modes.

2. Add the machine to **include/blob/arch.h**

```
#elif defined LITTLETON
#include <blob/arch/littleton.h>
```

3. Add the machine dependent architecture header file in **include/blob/arch/littleton.h**

```
cd include/blob/arch/
cp assabet.h littleton.h
Edit littleton.h and modify the parameters as per platform
This file defines different offsets which are required during booting. Following are worth
mentioning:
```

- Kernel, RAMDISK offsets in Flash
- Kernel XIP offset
- Registers Memory location
- TFTP download RAM base offset.

4. Add the machine dependent architecture header file to **Makefile.am** in **include/blob/arch** directory.
5. Add the machine dependent source file in **src/blob**

```
cd src/blob
cp assabet.c littleton.c
Edit littleton.c and modify the parameters as per platform
This file defines routines for the following
```

- Config GPIO and Multi Function Pins
- Config Memory Management Unit
- Config serial driver
- Config ethernet driver
- Config Server and Client IP for the TFTP.

6. Add the machine dependent source file to **Makefile.am** in **src/blob**

7. Add the architecture number to **include/blob/linux.h**

```
#elif defined LITTLETON
# define ARCH_NUMBER (1388)
```

Number is generated by Linux kernel when kernel code is compiled. One should ensure that the architecture number in the boot loader and linux kernel are same.

XScale Low Level Initialization and Xscale Low Level Primitives routines are provided by Marvell for the blob boot loader.

Update the **CONFIG_CMDLINE** before building the blob boot loader in include/blob/blob_cmdline.h. This command line parameter is passed to the kernel image.

```
#define CONFIG_CMDLINE "root=/dev/mtdblock2 rootfstype=jffs2 ip=192.168.1.101 :
192.168.1.100 :: 255.255.255.0::eth0:on console=ttyS2,115200 mem=64M comm_v75
uart_dma".
```

Parameters

root = Define the root directory which is to be mounted during booting.

rootfstype = Defines the root file system.

ip = Defines the TFTP Server IP.

console = Defines the serial port and baud rate to support the hyper terminal

mem = Defines the external DRAM. Board has 64 MB DRAM.

3.4.2 Compiling Blob boot loader

The following command is used to build the blob boot loader:

```
{blob base directory}$: ./configure --host=arm-linux --with-board= littleton --with-linux-
prefix='PATH_TO_LINUX_SRC' --with-network=eth --enable-xllp --enable-xlli --enable-
nand --with-cpu-product-id=MonahansLV --with-nand-layout=comm_v75 ;
```

Parameters

host = Defines the host machine

with-board = Defines the Board Name as configured in the configuration file.

with-linux-prefix = Path to the linux source code of boot loader.

with-network = Enables the network support.

Enable-xllp = Enable the xscale low level primitives

enable-xlli = Enable the xscale low level initialization

enable-nand = Enables the NAND supports on the board.

with-cpu-product-id = Defines the CPU name

with-nand-layout = Defines the type nand layout used on the board.

Blob can be compiled by firing the following command sequence on the command line:

```
- arm-linux-gcc -g -O1 -W -Wall -DCONFIG_CPU_MONAHANS_LV -
DCONFIG_NAND_COMM_V75 -DLITTLETON -Wa,--
defsym,CONFIG_CPU_MONAHANS_LV=1 -
/home/praveen/PlatformRel_Linux2.6.25/pxalinux/pxalinux/src/preview-
kit/blob/./linux/include -o blob-elf32 -static -nostdlib -WI,-T,./start-ld-script -WI,-
Map,blob-start-elf32.map start.o testmem.o xlli/littleton/libxlli.a -lgcc -lc
```

During compilation, the Blob boot loader prints the final configuration as

<i>Configuration</i>	

<i>Target board</i>	<i>Marvell Littleton</i>
<i>Target CPU</i>	<i>monahans</i>
<i>C compiler</i>	<i>arm-linux-gcc</i>
<i>C flags</i>	<i>-g -O1 -W -Wall -DCONFIG_CPU_MONAHANS_LV -</i> <i>DCONFIG_NAND_COMM_V75 -DLITTLETON -Wa,--defsym,</i> <i>CONFIG_CPU_MONAHANS_LV=1</i>
<i>Linker flags</i>	<i>-static -nostdlib</i>
<i>Objcopy tool</i>	<i>arm-linux-objcopy</i>
<i>Objcopy flags</i>	<i>-O binary -R .note -R .comment -S</i>
<i>Clock scaling support</i>	<i>no</i>
<i>Memory test support</i>	<i>no</i>
<i>LCD support</i>	<i>no</i>
<i>MD5 support</i>	<i>no</i>
<i>Xmodem support</i>	<i>no</i>
<i>UU Codec support</i>	<i>no</i>
<i>JFFS2 support</i>	<i>no</i>
<i>cramfs support</i>	<i>no</i>
<i>zImage support</i>	<i>no</i>
<i>PCMCIA support</i>	<i>no</i>
<i>CF support</i>	<i>no</i>
<i>NAND flash support</i>	<i>yes</i>
<i>IDE support</i>	<i>no</i>
<i>Generic IO support</i>	<i>no</i>
<i>Network support</i>	<i>no</i>
<i>Blob commands:</i>	<i>reset reboot arp autoip setip tftp</i>
<i>Run-time debug information</i>	<i>no</i>
<i>Serial over USB support</i>	<i>no</i>

After blob compilation is successfully completed, **blob-elf32** is created in **blob/src/blob** directory. From **blob-elf32** remove all the debug symbol (-S), comments (-R .comments) and notes (-R .notes) using **arm-linux-objcopy**. This can be done by executing the to following command.

```
- arm-linux-gcc -g -O1 -W -Wall -DCONFIG_CPU_MONAHANS_LV -
DCONFIG_NAND_COMM_V75 -DLITTLETON -Wa,--
defsym,CONFIG_CPU_MONAHANS_LV=1 -
/home/praveen/PlatformRel_Linux2.6.25/pxalinux/pxalinux/src/preview-
kit/blob/./linux/include -o blob-elf32 -static -nostdlib -WI,-T,./start-ld-script -WI,-
Map,blob-start-elf32.map start.o testmem.o xlli/littleton/libxlli.a -lgcc -lc

- arm-linux-objcopy -O binary -R .note -R .comment -S blob-elf32 blob
```

This command also creates a blob file.

3.5 Steps to Add New Platform in Linux Kernel

ARM machine dependent code is placed in the **linux/arch/arm** directory. Contents in the sub-directories are structured as follows:

- **boot** :- Contains code specific to booting process.
- **kernel**:- Contains architecture dependent kernel files.
- **configs**:- Contains default configuration for specific platform
- **common**:- Contains common file for the ARM architecture
- **mach-*** :- Contains platform and processor specific files.

mach-pxa: The reference platform was based on the PXA310 processor, and thus any future references to mach-pxa* for mach-pxa. Linux provides machine descriptor to define the machine definition. For adding support for a new machine in the kernel, we have to update the platform dependent configuration, which is defined in the following structure:

```

struct machine_desc {
    /*
     * Note! The first four elements are used
     * by assembler code in head.S, head-common.S
     */
    unsigned int      nr;           /* architecture number */
    unsigned int      phys_io;     /* start of physical io */
    unsigned int      io_pg_offst; /* byte offset for io
                                     * page table entry */
    const char        *name;       /* architecture name */
    unsigned long      boot_params; /* tagged list */
    unsigned int      video_start; /* start of video RAM */
    unsigned int      video_end;   /* end of video RAM */
    unsigned int      reserve_lp0 :1; /* never has lp0 */
    unsigned int      reserve_lp1 :1; /* never has lp1 */
    unsigned int      reserve_lp2 :1; /* never has lp2 */
    unsigned int      soft_reboot :1; /* soft reboot */
    void              (*fixup)(struct machine_desc *,
                               struct tag *, char **,
                               struct meminfo *);
    void              (*map_io)(void); /* IO mapping function */
    void              (*init_irq)(void);
    struct sys_timer *timer;       /* system tick timer */
    void              (*init_machine)(void);
};

```

3.5.1 Porting Linux Kernel on PXA310 Handheld PDK

The Linux kernel machine descriptor should be updated in ***littleton.c*** as following:

```

MACHINE_START(LITTLETON, "Marvell Form Factor Development Platform (aka Littleton)")
    .phys_io      = 0x40000000, /* Start of physical io */
    .boot_params  = 0xa0000100, /* tagged list */
    .io_pg_offst  = (io_p2v(0x40000000) >> 18) & 0xfffc, /* byte offset for io page table
entry */
    .map_io       = pxa_map_io, /* IO Mapping function */
    .init_irq     = pxa3xx_init_irq,
    .timer        = &pxa_timer, /* system tick timer */
    .init_machine = littleton_init, /* machine initialization function */
MACHINE_END

```

Above values are default values in linux kernel.

The ***MACHINE_START*** macro is defined in ***include/arch-asm/mach/arch.h***. The ***.init_machine*** is initialized with ***littleton_init()*** function. This ***littleton_init()*** initializes all other components which are platform dependent.

3.5.2 Android Linux Kernel Configuration and Compilation

Update the Linux kernel configuration file and make sure following options are set in configuration file. HSC used `/arch/arm/configs/pxa3xx_android_defconfig` file. After updating the configuration file compile the code and move final kernel image "**zImage**" in `linux/arch/arm/boot/` directory.

Following information captures the configuration values to be updated in the `pxa3xx_android_defconfig` file.

```
CONFIG_BLK_DEV_INITRD=y
CONFIG_INITRAMFS_SOURCE="root"
CONFIG_INITRAMFS_ROOT_UID=0
CONFIG_INITRAMFS_ROOT_GID=0

CONFIG_ASHMEM=y
CONFIG_ARM_THUMB=y
CONFIG_CMDLINE="console=ttyS2,115200 mem=64M ip=192.168.1.101:192.168.1.100::255.255.255.0::eth0:on comm_v75 uart_dma android"

CONFIG_USB_ADB_GADGET=y

CONFIG_ANDROID_RAM_CONSOLE=y
CONFIG_ANDROID_RAM_CONSOLE_ENABLE_VERBOSE=y
CONFIG_ANDROID_POWER=y
CONFIG_ANDROID_POWER_ALARM=y
CONFIG_ANDROID_POWER_STAT=y
CONFIG_ANDROID_LOGGER=y
CONFIG_ANDROID_TIMED_GPIO=y
CONFIG_ANDROID_BINDER_IPC=y
CONFIG_ANDROID_PARANOID_NETWORK=y
```

Following command shall be used to compile the Android Linux kernel.

```
For Compilation of Android Linux Kernel

$ tar zxvf linux-2.6.25.tgz
$ cd linux-2.6.25/
Installed the Linux Patches
$ patch -p1 < [installed_directory]/-xxx.patch
$ ..
$ cp ~/mydroid/out/target/product/littleton/root root -a
$ export ARCH=arm
$ export CROSS_COMPILE=arm-eabi-
$ make pxa3xx_android_defconfig
$ make zImage -j2
```

3.6 Burning into Flash

Once code is successfully compiled and binary image is moved to `/linux/arm/arch/boot` directory, burn the Linux kernel image (**zImage**) at the respective offset in NAND Flash on target platform.

For burning the executables on board, please follow release notes instructions. HSC followed instructions accompanying the Marvell littleton product.

4.0 PORTING ANDROID

Google conceptualized and created the Android Application Platform, initially targeted towards mobile devices. Android, as described above is in-turn based on the Linux OS. Android also features a comprehensive SDK that developers can use create applications. Android applications are created in Java. These applications run inside the Dalvik Virtual Machine and consume the libraries provided by Google as part of the developer toolchain.

4.1 Android Architecture

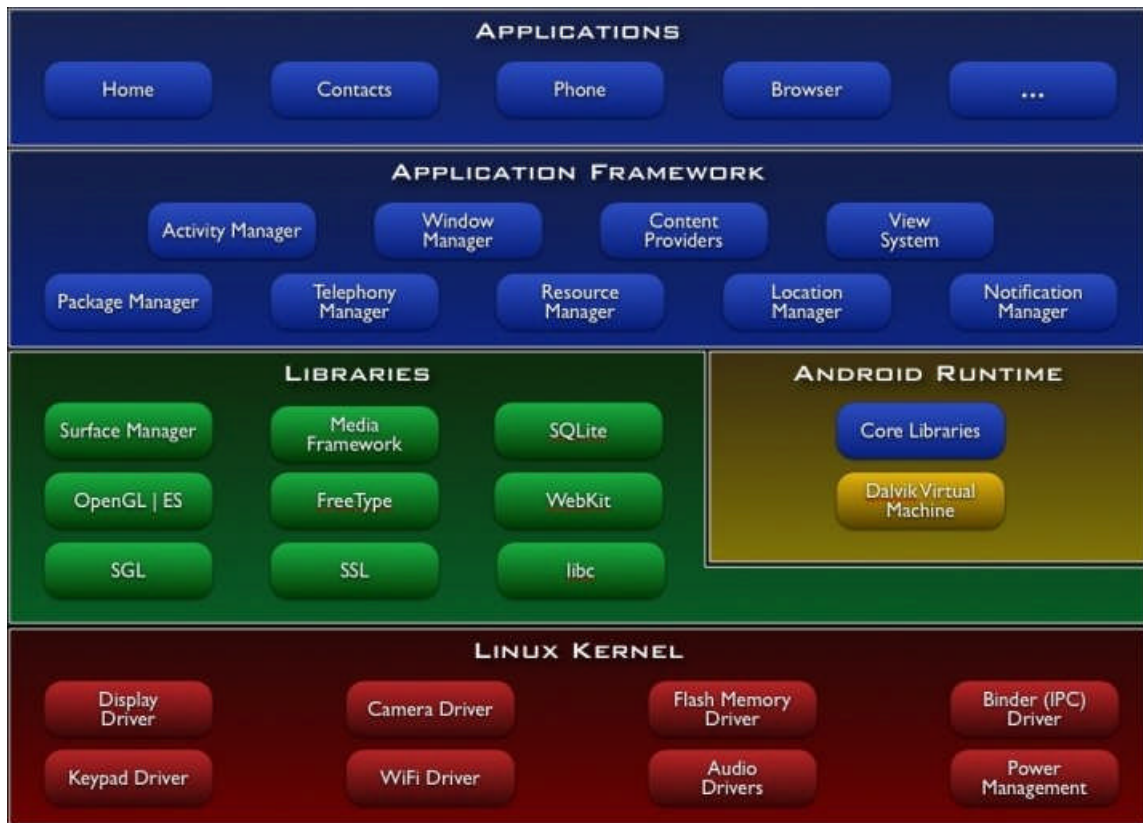


Figure 6: - Android System Architecture

The figure above has been published by the Open Handset Alliance and is available from the following location: <http://developer.android.com/guide/basics/what-is-android.html>.

Few new components integrated in Linux kernel for Android are Debugger, AshMem, Open Binder, Power Manager, Low memory killer and logger. Details of these modules are also captured on the same link.

4.1.1 Android Libraries

Set of core libraries used by the Android application framework are:

- **System C library:**- BSD-derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices
- **Media Libraries:**- Based on PacketVideo's OpenCORE; the libraries support playback and recording of many popular audio and video formats, as well as static image files, including MPEG4, H.264, MP3, AAC, AMR, JPG, and PNG
- **Surface Manager:**- Manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple applications
- **LibWebCore:**- Modern web browser engine which powers both the Android browser and an embeddable web view

- **SGL** - the underlying 2D graphics engine
- **3D libraries**- Implementation based on OpenGL ES 1.0 APIs; the libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer
- **FreeType**- Bitmap and vector font rendering
- **SQLite**- Powerful and lightweight relational database engine available to all applications

(Source - <http://developer.android.com/guide/basics/what-is-android.html>)

4.2 Add New Platform support in Android

First step to port the Android is to download the source code of Android. To download the source code of android, please follow the instructions, which can be obtained, from the following URL <http://source.android.com/download/using-repo>.

Snapshot of the downloaded directory are given as below. '**mydroid**' is the top level directory in which we have downloaded and installed the Android Application Framework source code.

```
~/mydroid$ ls
Makefile  bootable  external  kernel  system
Build     vendor    dalvik    frameworks packages
bionic    development hardware  prebuilt
```

After source code is downloaded, add new reference platform in the Android source code. Android has defined a framework to add a new platform in the source code. Any platform specific configuration can be added in the `~/mydroid/vendor` directory. Under **vendor** directory vendor names are mentioned and each vendor can have different types of platforms. In Figure 7, Marvell is the vendor name and littleton is the platform for which android is to be built.

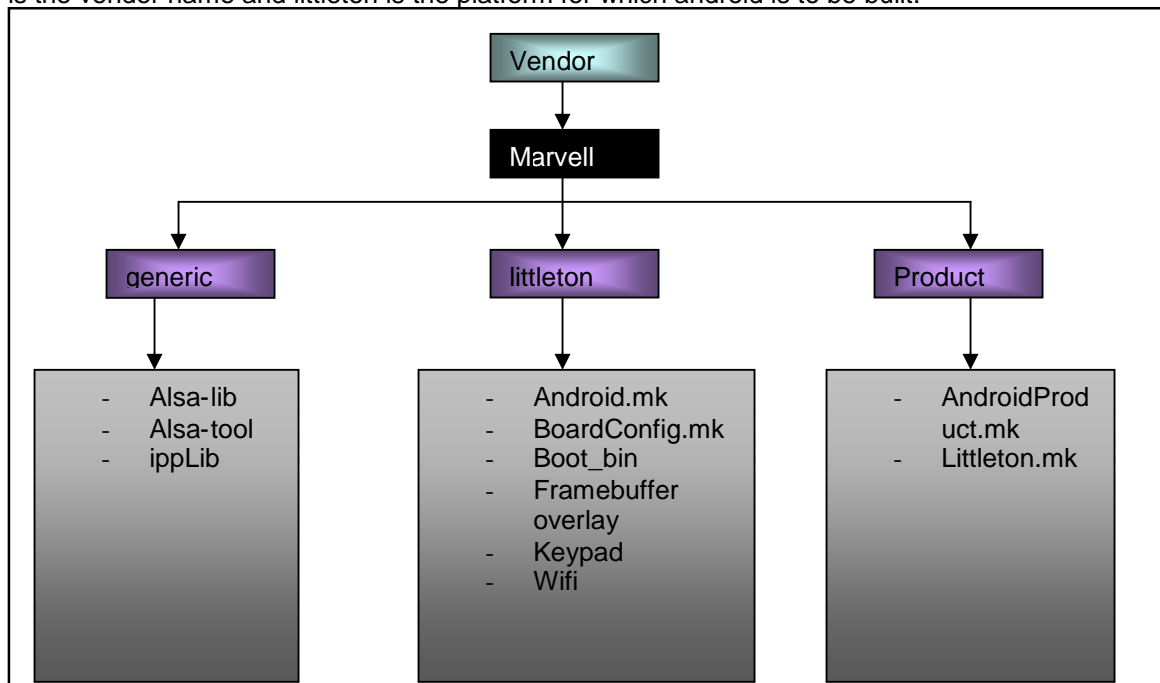


Figure 7 Adding new Platform Directory Structure

Figure 7 shows the structure of directories under the vendor directory. The Marvell directory comprises of three subdirectories:

- **general**:- This directory contains common code which reused across multiple platforms. Also In this directory sound architecture library and ipp (expand) library are stored. These can be used for PXA2xx, PXA3xx and PXA930 platforms
- **littleton**:- This directory contains board specific configuration for the Littleton platform. This directory also contains keypad, framebuffer driver implementation and other hardware information.

Following is the configuration file (BoardConfig.mk) for littleton platform.

```
# BoardConfig.mk
#
# Product-specific compile-time definitions.
#
# The generic product target doesn't have any hardware-specific
pieces.
TARGET_NO_BOOTLOADER := true
TARGET_NO_KERNEL := true
TARGET_NO_RADIOIMAGE := false
HAVE_HTC_AUDIO_DRIVER := false
BOARD_USES_GENERIC_AUDIO := false
USE_CAMERA_STUB := true
HAVE_MARVELL_WIFI_DRIVER := false
BOARD_HAVE_BLUETOOTH := false
```

- **product**:- Every board is considered as a product in Android's terminology. Marvell has multiple products – e.g. littleton, traverb saar. **AndroidProduct.mk** specifies the products which are included in final executable.

```
# AndroidProduct.mk
# This file should set PRODUCT_MAKEFILES to a list of product
makefiles
# To expose to the build system. LOCAL_DIR will already be set to
# The directory containing this file.
#
# This file may not rely on the value of any variable other than
# LOCAL_DIR; do not use any conditionals, and do not look up the
# value of any variable that isn't set in this file or in a file that
# It includes.
#
PRODUCT_MAKEFILES := \
$(LOCAL_DIR)/saar.mk \
$(LOCAL_DIR)/littleton.mk \
$(LOCAL_DIR)/tavorevb.mk
```

littleton.mk provides the product packages which needs to be built

```
#littleton.mk
PRODUCT_PACKAGES := \
  ApiDemos \
  Development \
  GPSEnable \
  Quake \
  AndroidTests \
  Barcodes \
  Email \
```

```
GlobalTime \  
IM \  
SoundRecorder \  
Stk \  
Term \  
ToDoList \  
Updater \  
VoiceDialer \  
WapBrowser \  
WhackAMole \  
TestHarness \  
VideoPlayer \  
SdkSetup
```

```
$(call inherit-product, build/target/product/generic.mk)
```

```
PRODUCT_NAME := littleton  
PRODUCT_DEVICE := littleton
```

Note: There are default product packages which are defined in the following makefiles:

- ***~/mydroid/build/target/product/generic.mk***
- ***~/mydroid/build/target/product/core.mk***
- ***~/mydroid/build/target/product/generic_with_google.mk***

4.3 Source Code Patches to Support Target Platform

After new reference platform is added to the Android framework, update Andoird source code with platform dependent patches. Patches list are provided in the section 3.2.2 .

4.4 Android Source Code Compilation

Once all the patches are applied, following commands should be used to build the Linux kernel:

To build android images:

```
$ cd ~/mydroid  
$ . build/envsetup.sh
```

To build the user build. User build is the optimized build, which contains pre-compiled packages.

```
$ make PRODUCT-littleton-user
```

To build the engineering build.

```
$ make PRODUCT-littleton-eng
```

After successful compilation following images should be generated under ***~/mydroid /out /target /product /Littleton/***.

- ***system.img***- It is yaffs2 flash file system. It is a partition image that will be mounted as / and thus contains all system binaries. This file contains the core of the Android System is stored.
- ***userdata.img***- It is data file system. It is backed by 64 MB NAND, and is using the yaffs2. This is the file system on which new user applications are installed and run from.

4.5 Burning into Flash

After successful compilation of Linux kernel, burn the android images (***system.img*** and ***userdata.img***) and Linux kernel images (***zimage***) at the respective offset in NAND Flash and start booting the PXA310 Handheld PDK. Figure 8 shows an android running over the target platform.

For burning the executables on board, please follow the release notes of the reference platform (HSC used Marvell's littleton's release notes)



Figure 8 Android Running On the PXA 310 Handheld Platform Development Kit

5.0 APPENDIX

5.1 Linux kernel – Boot-up details

Following figure captures the sequence of events from power-on to control given to `start_kernel` module in Linux kernel. On ARM based architecture boot loader passes control to the **head.o** module.

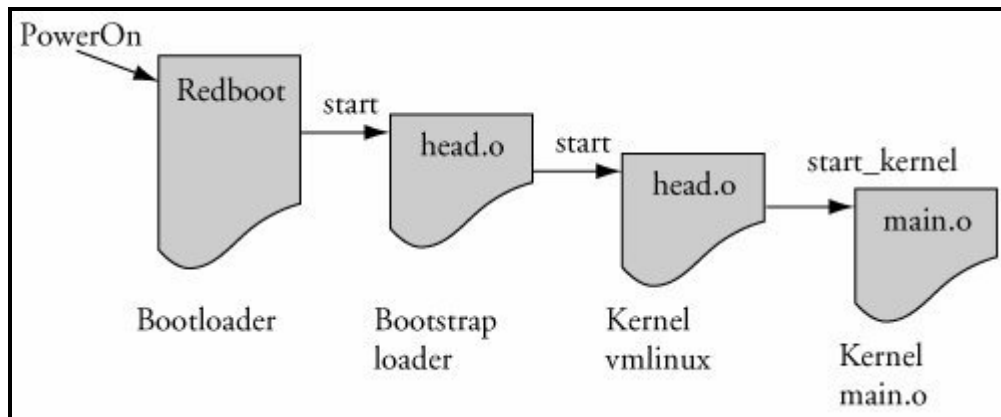


Figure 9 Linux kernel event sequence

The bootstrap loader (which is prepended to the kernel image) has primary responsibility to create proper environment to decompress and relocate kernel and pass control to the Linux kernel. Thus for most of architectures control is passed from the bootstrap loader directly to the kernel through a module called **head.o**. When the bootstrap loader has completed its job, control is passed to the kernel **head.o** and from there to **start_kernel()** in `main.c`.

The **head.o** module performs architecture and often CPU-specific initialization in preparation for the main body of the kernel. Across processor families, CPU-specific tasks are kept as generic as possible. Among other low-level tasks, **head.o** also performs following tasks.

- Checks for valid processor and architecture
- Creates initial page table entries
- Enables the processor's MMU
- Establishes limited error detection and reporting
- Jumps to the start of the kernel, **main.c start_kernel()**

The **start_kernel()** function also calls **setup_arch()**. This function takes a single parameter and a pointer to the kernel command line as captured in function invocation below:

```
setup_arch(&command_line);
```

This statement calls architecture specific setup routine, responsible for performing initialization tasks common across architecture. **setup_arch()** calls functions that identify the specific CPU and provides a mechanism for calling high-level CPU-specific initialization routines. Also, it checks for the configured machine descriptor from the platform configurations and initializes its variables.

```
static void (*init_machine)(void) __initdata;

static int __init customize_machine(void)
{
    /* customizes platform devices, or adds new ones */
    if (init_machine)
        init_machine();
    return 0;
}

arch_initcall(customize_machine);
```

```

void __init setup_arch(char **cmdline_p)
{
    struct tag *tags = (struct tag *)&init_tags;
    struct machine_desc *mdesc;
    char *from = default_command_line;

    setup_processor();
    mdesc = setup_machine(machine_arch_type);
    machine_name = mdesc->name;
    ...
    ...
    ...
    ...
    /*
     * Set up various architecture-specific pointers
     */
    init_arch_irq = mdesc->init_irq; /* This points to pxa3xx_init_irq */
    system_timer = mdesc->timer ; /* This points to pxa_timer */
    init_machine = mdesc->init_machine; /* This points to littleton_init */
    ...
}

```

The ***init_machine()*** function is called from ***customize_function()***, which is registered in the ***.initcallN.init*** section.

The ***customize_function*** is placed in the kernel's object file in a section called ***.initcall3.init***. ****_initcall*** macros are used as a registration functions for kernel subsystem initialization routines that need to be run once at kernel startup and then never used again. These macros provide a mechanism for causing the initialization routine to be executed during system startup, and a mechanism to discard the code and reclaim the memory after the routine has been executed.

The ***customize_function()*** is called from the ***do_initcalls ()***. The ***do_initcalls*** function executes all initialization function pointers present in ***.initcallN.init*** section. For Marvell board ***littleton_init()*** function also gets invoked at this function which does the initialization as captured above.