Building Root filesystem

- The organization of a Linux root filesystem in terms of directories is well-defined by the Filesystem Hierarchy Standard

/bin	Basic programs
/boot	Kernel image
/dev	Device files
/etc	System-wide configuration
/home	Directory for the users home directories
/lib	Basic libraries
/media	Mount points for removable media
/mnt	Mount points for static media
/proc	Mount point for the proc virtual filesystem
/root	Home directory of the root user
/sbin	Basic system programs
/sys	Mount point of the sysfs virtual filesystem
/tmp	Temporary files
/usr	
/usr/bin	Non-basic programs
/usr/lib	Non-basic libraries
/usr/sbin	Non-basic system programs
/var	Variable data files. This includes spool directories and files, administrative and logging data, and transient and temporary files

- Creating root filesystem for targer is creating required directories and populating directories with required files

Creating minimal Root filesystem:

- Create a working space for creating root filesystem

```
$ mkdir ~/embedded/roofs
$ cd ~/embedded/rootfs
```

- Now create the root filesystrem structure for target board

```
$ mkdir dev proc sys etc etc/init.d lib mnt usr usr/local
```

Populating bin, sbin, usr/bin, usr/sbin:

- Binaries for targer board is created using bustbox.

- A Linux system needs a basic set of programs to work
  - An init program
  - A shell
  - Various basic utilities for file manipulation and system configuration

- **Busybox** is an alternative solution, extremely common on embedded systems
    - Rewrite of many useful Unix command line utilities
    - Integrated into a single project, which makes it easy to work with
    - Designed with embedded systems in mind: highly configurable, no unnecessary features
- All the utilities are compiled into a single executable, /bin/busybox, symbolic links to /bin/busybox are created for each application integrated into Busybox

- Download Busybox source from **http://busybox.net**

- Configure Buskbox

$ make menuconfig

- Select the following options

Busybox Settings --->

Build Options --->

[*] Build BusyBox as a static binary (no shared libs)

Linux System Utilities --->

[*] mdev

Note : mdev is a mini-udev implementation for dynamically creating device nodes in the /dev directory.

Miscellaneous Utilities --->

[*] nandwrite
[*] nanddump
[*] ubiattach
[*] ubidetach
[*] ubimkvol
[*] ubirmvol
[*] ubirsvol
[*] ubiupdatevol
[*] flash_eraseall

- Cross-compile busybox
    - Export cross-compiler path

$ PATH=$PATH:/usr/local/bin/at91

- Compile

$ **make CROSS_COMPILE=arm-linux-**

- Install

$make CROSS_COMPILE=arm-linux- CONFIG_PREFIX=~/embedded/rootfs install

Populating etc:

init:

- Init is the first process started during booting, and is typically assigned PID number 1.
- Its primary role is to create processes from a scriptstored in the file /etc/inittab file.
- It is started by the kernel, and if the kernel is unable to start it, a kernel panic will result.
- All System V init scripts are stored in /etc/rc.d/init.d/ or /etc/init.d directory. These scripts are used to control system startup and shutdown.

- **BusyBox** can handle the system's startup. BusyBox **/sbin/init** is particularly well adapted to embedded systems, because it provides most of the init functionality an embedded system typically needs without dragging the weight of the extra features found in System V init.
    - BusyBox init does not provide runlevel support.
    - The **init** routine of BusyBox carries out the following main tasks in order:
        1. Sets up signal handlers for init.
        2. Initializes the console(s).
        3. Parses the inittab file, **/etc/inittab**.
        4. Runs the system initialization script. **/etc/init.d/rcS** is the default for BusyBox.
        5. Runs all the inittab commands that block (action type: wait).
        6. Runs all the inittab commands that run only once (action type: once).
    - Once it has done this, the init routine loops forever carrying out the following tasks:
        1. Runs all the inittab commands that have to be respawned (action type: respawn).
        2. Runs all the inittab commands that have to be asked for first (action type: askfirst).

Create inittab file in etc :

Each line in the inittab file follows this format:

id:runlevel:action:process

id	-	specify the tty for the process to be started.
runleven	-	completely ignores the runlevel field
action	-	can be any of the following

Action	Effect
1. sysinit	Provide init with the path to the initialization script.
2. respawn	Restart the process every time it terminates.
3. askfirst	Similar to respawn, but It prompts init to display "Please press Enter to activate this console."
4. wait	Tell init that it has to wait for the process to complete before continuing.
5. once	Run process only once without waiting for them.
6. ctrlaltdel	Run process when the Ctrl-Alt-Delete key combination is pressed.
7. shutdown	Run process when the system is shutting down.
8. restart	Run process when init restarts. Usually, the process to be run here is init itself.

- sample inittab file for mini2440.

$ vim ~/embedded/rootfs/etc/inittab

Copy the following into inittab file and save it.

```
# Startup the system
null::sysinit:/etc/init.d/rcS
# Start getty on serial for login
ttySAC0::respawn:/sbin/getty -L ttySAC0 115200 vt100
# Stuff to do before rebooting
null::shutdown:/bin/umount -a -r
```

This inittab file does the following:

1. Sets **/etc/init.d/rcS** as the system initialization file.
2. Starts getty on serial port.
3. Tells init to run the umount command to unmount all filesystems it can at system shutdown

Create profile file in etc:

- profile file has environment variables.

 # vim ~/embedded/rootfs/etc/profile

- Copy the following into profile file and save it.

```
# Used for prompt format
PS1='[\u@\h:\W]\# '
PATH=$PATH
HOSTNAME=`/bin/hostname`
export HOSTNAME PS1 PATH
```

Create passwd file:

- passwd file has password information of user.

 # vim ~/embedded/rootfs/etc/passwd

- copy the following line for veda user and save it.

```
veda::0:0:root:/root:/bin/sh
```

- Create rcS file under /etc/init.d

- This script can be quite elaborate and can actually call other scripts.
- Use this script to set all the basic settings and initialize the various components of the system like Mount additional filesystems, Initialize and start networking interfaces.
- Start system daemons.

Sample rcS file for mini2440

 # vim ~/embedded/rootfs/etc/init.d/rcS

Copy the follwing script into the rcS file and save it.

```
#!/bin/sh
# ---------------------------------------------
# Mounting procfs
# ---------------------------------------------
mount -n -t proc null /proc
# ---------------------------------------------
# Mounting sysfs
```

```
# ---------------------------------------------
mount -n -t sysfs null /sys
# ---------------------------------------------
# Mounting ramfs to /dev
# ---------------------------------------------
mount -n -t ramfs null /dev
# ---------------------------------------------
# Create /dev/shm directory and mount tmpfs to /dev/shm
# ---------------------------------------------
mkdir -p /dev/shm
/bin/mount -n -t tmpfs tmpfs /dev/shm
# ---------------------------------------------
# Enabling hot-plug
# ---------------------------------------------
echo "/sbin/mdev" > /proc/sys/kernel/hotplug
# ---------------------------------------------
# Start mdev
# ---------------------------------------------
mdev -s
# ---------------------------------------------
# Set PATH
# ---------------------------------------------
export PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin
# ---------------------------------------------
# Set ip address
# ---------------------------------------------
/sbin/ifconfig lo 127.0.0.1 up
/sbin/ifconfig eth0 10.0.0.111 up
# ---------------------------------------------
sleep 3
# ---------------------------------------------
# Assigning hostname
# ---------------------------------------------
/bin/hostname mini2440
# ---------------------------------------------
```

- Give executable permissions to rcS script file.

$chmod +x ~/embedded/rootfs/etc/init.d/rcS

Populating dev:

- As part of inittab & rcs files as we are using "null & ttySAC0" device nodes

$ cd ~/embedded/rootfs/dev
$ mknod console c 5 1
$ mknod null c 1 3
$ mknod ttySAC0 c 204 64

Populating lib:

- As we used buildroot for building cross-compiler copy all libs created by buildroot for target board.

$ cp -Rfp $(Buildroot_src)/output/host/usr/arm-buildroot-linux-uclibcgnueabi/lib/* ~/embedded/rootfs/lib

$ cp -Rfp $(Buildroot_src)/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot/lib/* ~/embedded/rootfs/lib

$ cp -Rfp $(Buildroot_src)/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot/usr/lib/* ~/embedded/rootfs/lib

$ cp -Rfp $(Buildroot_src)/output/host/usr/lib/* ~/embedded/rootfs/lib

- And copy other libraries cross-compiled for targer board.

Installing kernel modules:

- At the time of building kernel if ant service is selected as modules, than we neet to install those modules as part of targer root file system.
- The following command will install kernel modules in target root file system

- Change directory to cross-compiled linux kernel and give following command.

$ cd $(PATH/TO/LINUX-SRC)
$ make ARCH=arm CROSS_COMPILE=arm-linux- modules
$ make ARCH=arm CROSS_COMPILE=arm-linux- INSTALL_MOD_PATH=~/embedded/rootfs/ modules_install

Creating a loopback device to create filesystem image:

- Create a file on hard disk with enough space to hold the filesystem for target
  # **dd if=/dev/zero of=$(path/to/rootfs.img) bs=1024 count=4096**
  creates a file with name rootfs.img of size 4M

- Disk file to be treated as a block device
  # **losetup /dev/loop0 /home/target_images/rootfs.img**

- Create a filesystem on loopback device
  # **mkfs.ext3 /dev/loop0 4096**

- Mount loopback device
  # **mount -t ext3 /dev/loop0 /mnt**

- Copy entire root filsystem structure to mount point (remove lib directory from rootfs)
  # **cp -Rfp rootfs/* /mnt**

- Unmount loopback device
  # **umount /mnt**