My Experience

In Embedded Linux

Why Embedded Liniux Software ?

- Linux is famous among embedded engineers as it is an open-source operating system with a customizable kernel.
- This means anyone can make their own operating system to suit their particular needs by keeping just the necessary parts thus keeping the size small yet support their device functionality.
- Then on-top of the kernel, they can make their own application code to make an embedded device.
- Some famous examples of porting linux to embedded devices include BeagleBone,Raspberry Pi and Jeston devices.

What do we need to successfully boot linux on the hardware such as Beaglebone black or any hardware which supports linux ?

- To boot linux on hardware which supports linux , requires 4 software components.
- They are
  1. RBL(Rom Bootloader)
  2. MLO(Memory Loader) & u-boot
  3. Kernel
  4. Root File system
- To build linux boot images on a host system an SDK or cross toolchain is required to be present on the host system.

A cross toolchain comprises of the following

- Binary tools – for compile and build of application, kernel and boot-loader sources.
- A set of libraries required for application build.
- Run-time libraries which implement ABI standard.
- Support libraries for dynamically linked applications.
- Kernel header files.
- Debug and tracing tools.

Creating a target specific toolchain

- Creating a target specific toolchain can be carried out either through manual integration or an automated build(buildroot).

- **Buildroot :**
- Buildroot is a framework that simplifies and automates the process of building a complete Linux system for an embedded system, using cross-compilation.
- In order to achieve this, Buildroot is able to generate a cross-compilation toolchain, a root filesystem, a Linux kernel image and a bootloader for your target.
- Buildroot can be used for any combination of these options, independently.

Building Toolchain

- Download the Buildroot source form **https://buildroot.org/downloads/** and extract tar file. Go to buildroot root directory and check if board support exsists. The directory **configs** contains **board support defconfigs**, Here beaglebone black board configuration file is **beaglebone_defconfig**.
- Apply board specific config configuration file.

  $ make beaglebone_defconfig
- Apply required changes to the configuration file using menuconfig

  $ make menuconfig.

  *** Toolchain**

  [*] Enable WCHAR support

  [*] Enable RPC support

Building Toolchain

- Now save the configuration & Initiate the build process

  $ make toolchain

- make toolchain - for generating cross compiler tool chain binary.
- The Cross compiler binaries that are generated will be at

  (path_to_buildroot)/output/build/host/usr/bin.

u-boot

- U-boot is open source Embedded boot-loader project.
- Support for most Embedded processor architecture.
- Support for wide range of boards.
- Active and rich community of developers.
- Implements both stage1 and stage2 boot-loader facilities(Build system can generate both stage1 and stage2 images.)
- Support for secure boot and multi-boot images.
- Full support for open firm-ware device tree.
- Standard customization interfaces.
-

Compile & build u-boot

U-boot:

- Boot sequence of an ARM board using linux kernel will be

  Boot ROM code => MLO (1st stage) => U-boot.bin(2nd stage) => kernel => rootfs

- Download u-boot source from: **ftp://ftp.denx.de/pub/u-boot/**
- Export path of cross compiler to PATH variable

  $ PATH=$PATH:(PATH_TO_BUILDROOT)/output/host/usr/bin

- Check board support for beaglebone black in **configs** file.
- Apply default configuration file

  $ make am335x_boneblack_defconfig

Compile & build u-boot

U-boot :

- Compile & build u-boot

  $ make ARCH=arm CROSS_COMPILE=arm-linux-

- Multiple files are generated in the process in u-boot top folder and two files are important:

  MLO (first stage bootloader)

  u-boot.img (second stage bootloader)

  * u-boot.bin is the binary compiled U-Boot bootloader.

  * u-boot.img contains u-boot.bin along with an additional header to be used by the boot ROM/MLO to determine how and where to load and execute U-Boot.

Linux Kernel

Kernel :

- Kernel is the third element of Embedded Linux. It is the component that is responsible for managing resources and interfacing with hardware , and so affects almost every aspect of your final software build.
- Usually kernel is tailored to your particular hardware configuration.
- Kernel has three main jobs to do :

  1. To manage resources
  2. To interface to hardware
  3. To provide an API that offers a useful level of abstraction to user space programs.

Compile & build Linux Kernel

Kernel :

- Download linux kernel source from: **https://www.kernel.org/**
- Export path of cross compiler to PATH variable

  $ PATH=$PATH:(PATH_TO_BUILDROOT)/output/host/usr/bin
- Check board support for beaglebone black in arch/arm/configs file.
- Assign default configuration file

  $ make ARCH=arm omap2plus_defconfig
- Compile & build linux kernel

  $ make ARCH=arm CROSS_COMPILE=arm-linux-

Compile & build Linux Kernel

Kernel :

- Vmlinux is created at the root level in kernel tree . Creates bootable kernel images in

  $ (linux_source)/arch/arm/boot/ we find:

  Image => uncompressed kernel image

  zImage => compressed kernel image

- Building Device Tree Binary:

  dts files for BBB: am335x-boneblack.dts

  dtsi files are: am33xx.dtsi, am335x-bone-common.dtsi

  $ make ARCH=arm CROSS_COMPILE=arm-linux- am335x-boneblack.dtb

- the corresponding **am335x-boneblack.dtb** file is generated a

  $ (linux_source)/arch/arm/boot/dts.

Root Filesystem

Rootfs:

- File system images are needed to separate application binaries from kernel image, this separation allows apps to be loaded into memory as files, and allows kernel to set up address space for each application.
- Now the root file system, as the name indicates, it’s a file system which linux mounts to the “ / “ (root).
- **File system is nothing but a collection of files organized in standard folder structure.**
- In a typical file system you will find the below folder structure even though all these folders are not required for linux to boot and mount the file system successfully.

Root Filesystem

Directory Description:

- **/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 Filesystem

Directory Description:

- **/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.

Creating minimal root file system

Creating root file system for a target machine is, creating required directories and populating them with appropriate files.

1. Create a work space for root filesystem to be built:

   # **mkdir roofs**

   # **cd rootfs**

2. Now create the root filesystem structure for target machine:

   # **mkdir dev proc sys etc bin sbin lib mnt usr usr/bin usr/sbin**

3. Populate **bin , sbin , usr/bin , usr/sbin**

   Binaries for target machine is created using **busybox**.

Busybox

- Busybox is nothing but a software tool, that enables you to create a customized root file system for your embedded linux products.
- Busybox enables you to create customized file system that meets your resources requirements.
- **Busybox has the potential to significantly reduce the memory consumed by various commands by merging all the linux commands in one single binary.**

  (It does not generate individual linux commands binaries, there is only one executable binary thats is called “busybox” which implements all the required linux commands which you can select using the configuration tool.)

Configure & Compile Busybox

1. Download busybox source from **http://busybox.net** and extract it:

   # tar -xvf <archived-busybox-source-file>

2. Configure busybox

   # make menuconfig

3. Compile busybox using cross-compiler:

   # make CROSS_COMPILE=$(PREFIX-CROSSCOMPILER)

4. Install the commands in target root filesystem:

   # make CROSS_COMPILE=arm-linux- CONFIG_PREFIX=<path-to-rootfs> install

Creating minimal root file system

4. Populate **/etc**

Init is the first process started during booting, and is typically assigned PID number 1.

Its primary role is to create processes from a script stored in the file **/etc/inittab** file.

i. Create **inittab** file in /etc

Each line in the inittab file follows this format: **id:runlevel:action:process**

ii. Create **profile** file in /etc

profile file has environment variables

iii. Create **passwd** file in /etc

passwd file has user's password information.

iv. Create **rcS** file under **/etc/init.d** and give executable permissions to **rcS** script file.

Creating minimal root file system

5. Populate **/dev**

   As part of inittab & rcs files as we are using "null & ttyS0" device nodes we need to create them manually.

   Create device nodes for beagle bone:

   # cd <path-to-rootfs>/dev

   # mknod console c 5 1

   # mknod null c 1 3

   # mknod ttyS0 c 204 64

6. Populate **/lib**

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

Install kernel modules on target rootfs

- At the time of building kernel if any service is selected as modules, than we need to install those modules as part of target 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-source>

  # make ARCH=arm CROSS_COMPILE=arm-linux- modules

  # make ARCH=arm CROSS_COMPILE=arm-linux- INSTALL_MOD_PATH=<path-to-rootfs> modules_install

BeagleBone Black Booting

- The BeagleBone Black is a full featured, internet enabled development platform that utilizes the low cost **SitaraTM AM3358 ARM® CortexTM-A8** processor from Texas Instruments and runs a variety of OS including Debian, Angstrom, Ubuntu and Android.

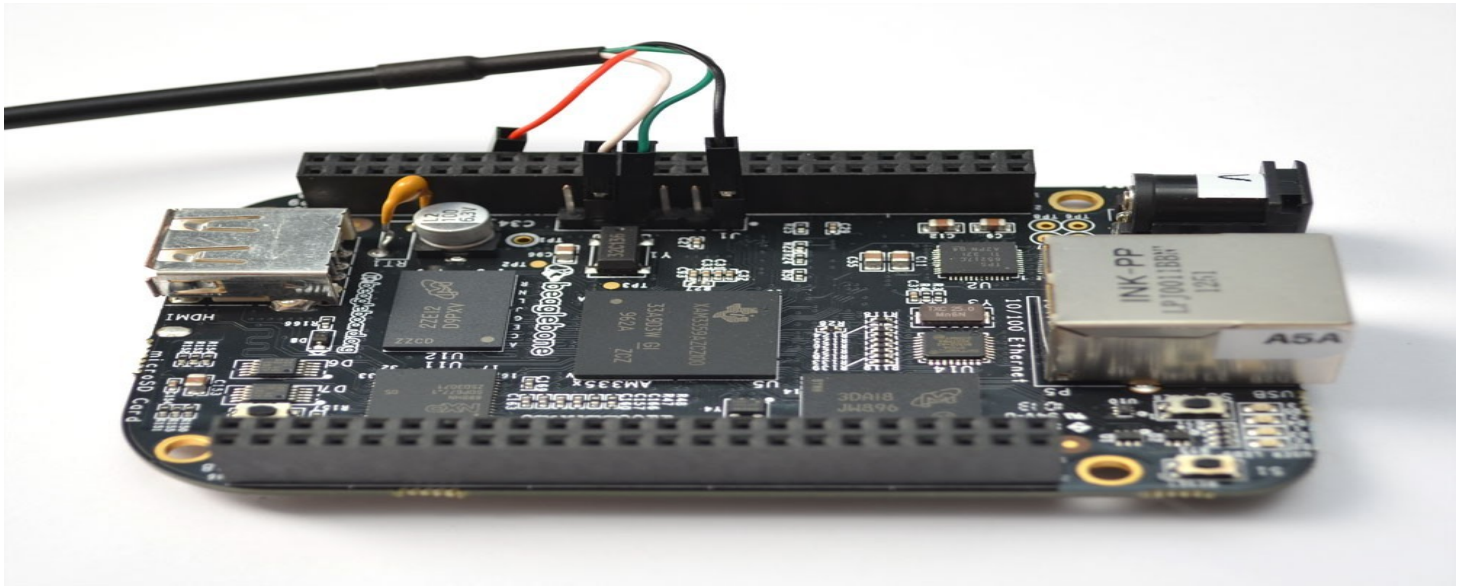
- The BeagleBone Black is designed to address the needs of designers, early adopters, and anyone in the Open Source Community interested in a low cost ARM® CortexTM-A8 based processor.

- The boot sequence of Beaglebone is eMMC, mmc0, usb. This board is having three boot sources. It has internal 4GB eMMC that contains vendor provided preloaded images. When you power on the board it takes eMMC as the default boot device and boot from there. If you want to alter the boot sequence, the board provides a boot switch, which when hit switches the booting from external mmc, like sd card.

BeagleBone Black Booting

Power and Communication channels:

- For booting beaglebone board we must need two connections.
- One is usb power cable for power source.
- Second: TTL to USB cable for serial communication interface to host.
- TTL to USB is having four pins red(VCC), black(GND), green(RXD), white(TXD).
- Connect serial pins to serial port properly as per the fig.below.



BeagleBone Black Booting from SD card

- Connect the SD card to host system and create two partitions one for **BOOT** and one for **ROOT** and format the two partitions as **Fat32** and **ext3** filesystems.
- Copy all bootable files (**MLO, u-boot.img, zImage, am335x-boneblack.dtb**) into **BOOT** partition.
- Copy **rootfs** into **ROOT** partition. unmount the sd card.
- Insert the SD card into the board sd card slot.
- Connect the serial to usb to host and open **minicom** terminal.
- Then press boot button and poweron by using usb cable.
- The board automatically boots from the sd card showing u-boot prompt.
- Press space bar to abort autoboot.

BeagleBone Black Booting from SD card

- Then set the bootargs for the kernel using setenv command:

  => setenv bootargs console=ttyS0,115200 rw root=/dev/mmcblk0p2 rootfstype=ext3 rootwait

- Load the kernel and device tree into RAM

  => load mmc 0:1 0x82000000 zImage

  => load mmc 0:1 0x84000000 am335x-boneblack.dtb

- Boot the kernel from RAM using bootz

  => bootz 0x82000000 - 0x84000000

BeagleBone Black Booting using NFS

- To mount **rootfs** on target from host using **NFS**, open /etc/exports file:

  $ sudo vim /etc/exports

- Copy below line in the /etc/exports file on host machine & save it.

  /(Rootfs_path) 10.0.0.111(rw,sync,no_root_squash,no_all_squash,no_subtree_check)

- Restart NFS server on host

  $ sudo /etc/init.d/nfs-kernel-server restart

- Then set the bootargs for the kernel using setenv command:

  => setenv bootargs console=ttyO0,115200 ip=10.0.0.111:10.0.0.4::255.255.255.0 rw

  root=/dev/nfs nfsroot=10.0.0.4:(/path to rootfs)

BeagleBone Black Booting using NFS

- Load the kernel and device tree into RAM

  => load mmc 0:1 0x82000000 zImage

  => load mmc 0:1 0x84000000 am335x-boneblack.dtb

- Boot the kernel from RAM using bootz

  => bootz 0x82000000 - 0x84000000

Yocto Project

- Yocto is a set of templates, tools and methods that allow user to build custom Embedded linux-based systems(a distribution) regardless of the hardware architecture.
- Yocto project is like buildroot can give us the same end products but with flexible package management and SDK
  1. Boot loaders
  2. Rootfs image for embedded device
  3. Kernel
  4. Compatible Toolchain
  5. Package management System
- Core components of yocto build system 1. Poky 2. OpenEmbedded-core 3. Bitbake

Build for BBB using Yocto

- **~/poky-warrior $ source oe-init-build-env /home/($YOUR HOME DIRECTORY)/bbb-build**
- After execution of the above command, current directory will switch to **build** directory.
- Do following changes

  /home/($YOUR HOME DIRECTORY)/bbb-build $ vim conf/local.conf
- In local.conf file uncomment the following line (commented lines begin with hash "#")

  MACHINE ?= "beaglebone-yocto"
- Trigger build using this command:

  /home/($YOUR HOME DIRECTORY)/bbb-build $ bitbake -k core-image-minimal

Thank You