Yocto Linux

Pre-requisites

- Linux distribution
- Linux Booting
- Initial RAM disk
- Kernel compilation (for PC & BBB)

Applications of Embedded Linux

- Smart TV
- Drone
- Digital Signage
- Biometric Devices
- Robot control
- Media player
- etc.

Elements of Embedded Linux

- Toolchain: The compiler and other tools needed to create code for your target device.
- Bootloader: The program that initializes the board and loads the Linux kernel.
- Kernel: Heart of the system, managing system resorces and interfacing with hardware.
- Root filesystem: Contains libraries and programs that run once kernel completed its initialization.
- Application specific programs: Collection of programs specific to embedded application.

Yocto

- A smallest SI metric measure. 10^-24.

- Yocto project provides open source, high quality infrastructure, and tools to help developers create their own custom Linux distributions for many hardware architecture.
- Yocto project founded in 2010 to reduce work duplication and provide resources + information to users to make their small Linux distribution.
- Working group of the Linux foundation.
- What Yocto does?

1. Kernel config
2. Hardware details
3. Packages/Binaries to install

→ Yocto Project →

1. Linux kernel
2. Root file system
3. Bootloader
4. Device Tree
5. Toolchain

Advantages

1. Widely Adopted Across the Industry
   - Semiconductor, operating system, software, and service vendors exist whose products and services adopt and support the Yocto Project.
   - Eg. Intel, Facebook, arm, Juniper Networks, LG, AMD, NXP, DELL
2. Architecture Agnostic
   - supports Intel x86, ARM, MIPS, AMD, PPC and other architectures.
   - Chip vendors create and supply BSPs that support their hardware.
   - If you have custom chip, you can create a BSP that supports the architecture
   - Yocto Project fully supports a wide range of device emulation through the QEMU
3. Images and Code Transfer Easily
   - Yocto Project output can easily move between architectures without moving to new development environments.
4. Flexibility

- Through customization and layering, a project group can leverage the base Linux distribution to create a distribution that works for their product needs.

5. Ideal for Constrained Embedded and IoT devices
    - Unlike a full Linux distribution, you can use the Yocto Project to create exactly what you need for embedded devices
    - You only add the feature support or packages that you absolutely need for the device

6. Uses a Layer Model
    - Yocto Project layer infrastructure groups related functionality into separate bundles.
    - You can incrementally add these grouped functionalities to your project as needed
    - Allows to easily extend the system, make customizations, and keep functionality organized.

Setup Machine

- Pre-requisites
    - 60-gb of free disk space
    - Supported Linux distribution for Yocto 4.x
        - Ubuntu 20.04 (LTS)
        - Ubuntu 22.04 (LTS)
        - Fedora 38
        - Debian GNU/Linux 11.x (Bullseye)
        - AlmaLinux 8
    - GIT >= 1.8.3.1
    - tar >= 1.27
    - Python >= 3.4
    - Packages:
        - terminal> sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev python3-subunit mesa-common-dev zstd liblz4-tool file locales libacl1

Poky

- Poky means uncomfortably small.
- Poky is a reference (example) distribution of Yocto project.

- Yocto project uses Poky to build images (kernel, system, and applications) for the target hardware.
- Technically Poky is combined repository of components
    - Bitbake
    - OpenEmbedded Core
    - meta-yocto-bsp
    - Documentation
- Note that, Poky doesn't contain binary files. It is working example of how to build your own custom Linux distribution from source.
- What is Poky and Yocto?
    - Yocto refers to organization; while Poky is source downloaded.

Metadata

- Refers to detailed build instructions.
- Commands and data used to indicate versions/sources of the softwares used.
- Changes/additions to the software (patches) used to fixed bugs or customize software for particular use-case.
- If consists of
    - Configuration files (.conf)
    - Recipe (.bb and .bbappend)
    - Classes (.bbclass)
    - Includes (.inc)

OpenEmbedded Project

- https://www.openembedded.org/wiki/Main_Page
- OpenEmbedded offers a best-in-class cross-compile environment. It allows developers to create a complete Linux distribution for embedded systems.
- OE provides a comprehensive set of metadata for a wide variety of archietctures, features, and applications. It runs on any Linux distribution.
- OE is not a reference distribution.
- OE is designed to be foundation for others.
- Yocto project focuses on providing powerful, easy-to-use, interoperable, well-tested tools, metadata, and BSPs for a core set of architectures and specific boards.
- Yocto project adopted openembedded as its build system. These projects share a core collection of metadata (recipes, classes, and associated files) called openembedded-core (oe-core).

Bitbake

- Bitbake is a core component of Yocto project.
- It basically performs same functionality as of make.
- It is a task scheduler that parses python and shell script mixed code (used to define metadata).
- The code parsed generates and runs tasks, which are set of steps ordered as per code's dependencies.
- It reads recipes and follow them by fetching packages, building them and incorporating the results into bootable images.
- It keeps track of all tasks being processed in order to ensure completion, maximizing use of processing resources to reduce build time and being predictable.

meta-yocto-bsp

- BSP defines how to support particular hardware device, set of devices, or hardware platform.
- It includes information about hardware features present on the device, kernel config information, additional device drivers, and additional software components (on top of generic Linux) for essential platform features.
- The meta-yocto-bsp layer in Poky maintains several BSPs for well-known boards like Beaglebone, EdgeRouter, x86-32, x86-64, etc.
- To develop on other hardware platforms, you need to add hardware-specific Yocto layers to the Poky.

Miscellaneous

- meta-poky: Poky specific metadata.
- Documentation: Yocto Project source files used to make the set of user manuals.

Poky - At a glance

- (Simplified) Poky = BitBake + Metadata
- Poky includes
    - OE-core
    - bitbake
    - BSPs
    - Helper scripts
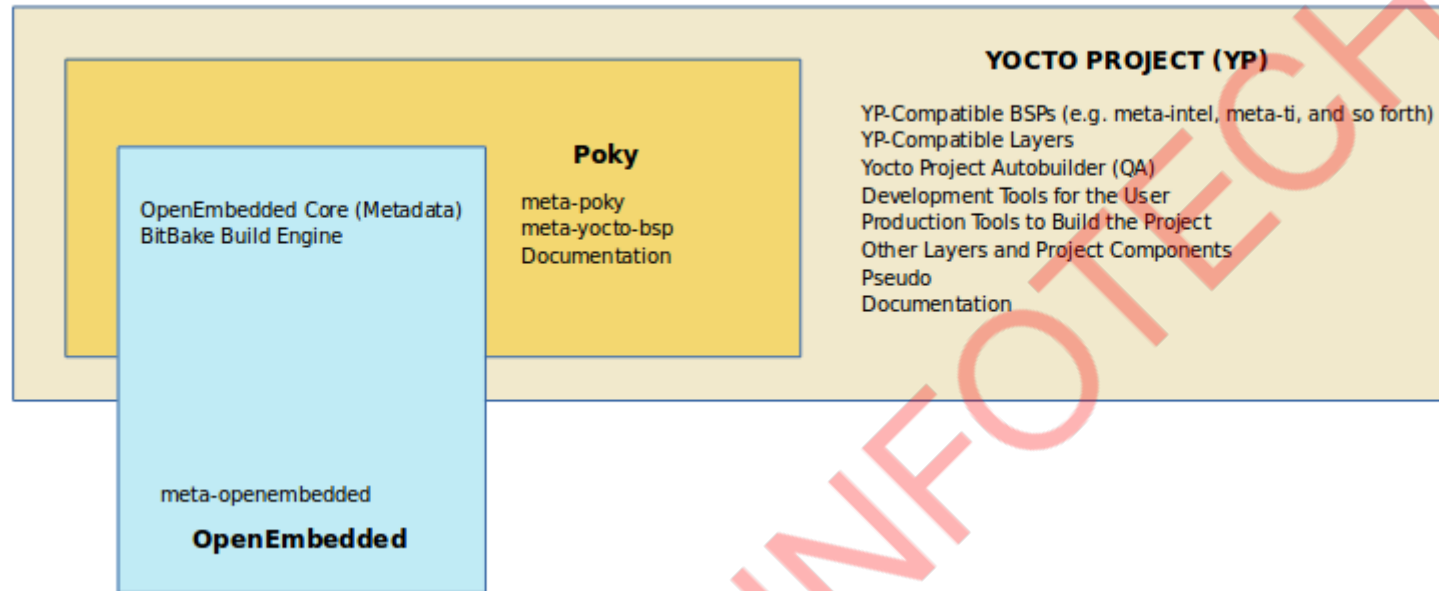
- QEmu

Yocto Steps - Build x86 image

- Step 1: Download the Poky Source code
  - $ git clone git://git.yoctoproject.org/poky
- Step 2: Checkout the latest branch/release (zeus)
  - $ git checkout zeus
- Step 3: Prepare the build environment
  - Poky provides you a script 'oe-init-build-env', which should be used to setup the build environment
  - script will set up your environment to use Yocto build system,including adding the BitBake utility to your path
  - $ source poky/oe-init-build-env [build_directory]
  - e.g. $ source poky/oe-init-build-env ../poky-build
  - Here build_directory is an optional argument for the name of the directory where the environment is set. If not given , it defaults to "build".
  - The above script will move you (cd) in a build folder and create two files (local.conf, bblayers.conf) inside conf folder
- Step 4: Building Linux Distribution
  - $ bitbake <image_name>
  - $ bitbake core-image-minimal

- core-image-minimal: This is a small image allowing a device to boot, and it is very useful for kernel and boot loader tests and development
- Step 5: Run the generated image in QEMU
  - Quick Emulator (QEMU) is a free and open source software package that performs hardware virtualization.
  - The QEMU based machines allow test and development without real hardware.
  - Currently emulations are supported for:
    - ARM
    - MIPS
    - MIPS64
    - PowerPC
    - X86
    - X86_64
  - Poky provides a script 'runqemu' which will allow you to start the QEMU using yocto generated images.
  - The runqemu script is run as:
    - $ runqemu
      - is the machine/architecture to use (qemuarm/qemumips/qemuppc/qemux86/qemux86-64). Default is current image build arch.
      - is the path to a kernel (e.g. zimage-qemuarm.bin)
      - is the path to an ext2 image (e.g. filesystem-qemuarm.ext2) or an nfs directory
    - $ runqemu -- to see help
    - e.g. $ runqemu core-image-minimal
  - To run qemu without graphic (for non-gui images):
    - $ runqemu nographic
  - Exit QEMU by either clicking on the shutdown icon or by typing Ctrl-C in the QEMU transcript window from which you started QEMU.

Generate ARM images

- When you set up the build environment, a local configuration file named local.conf becomes available in a conf subdirectory of the Build Directory
- The defaults are set to build for a qemux86-64 target
- Edit ./build/conf/local.conf
  - MACHINE = "qemuarm"
- $ source poky/oe-init-build-env
- $ bitbake core-image-minimal
- $ runqemu core-image-minimal

Add packages in root file system

- Open your local.conf file and add the recipe for the package (e.g. lsusb)
    - IMAGE_INSTALL += "recipe-name"
    - e.g. IMAGE_INSTALL += "usbutils" # for lsusb
    - or IMAGE_INSTALL_append = " usbutils"
- $ bitbake core-image-minimal
- $ runqemu core-image-minimal

Build GUI based image

- This is the X11 Window-system-based image with a SATO theme and a GNOME mobile desktop environment.
    - $ bitbake core-image-sato

Yocto Terminologies

Metadata

- Metadata is collection of
    - Configuration files (.conf)
    - Recipes (.bb and .bbappend)
    - Classes (.bbclass)
    - Includes (.inc)

Recipes

- In general, a recipe is a set of instructions that describe how to prepare or make something (a dish)
- Yocto: A recipe is a set of instructions that is read and processed by the bitbake.
- Extension of Recipe: .bb
- A recipe describes:
    - from where to get source code
    - which patches to apply
    - Configuration options

- - Compile options (library dependencies)
  - Install
  - License
- It is a software component
- Examples
  - os-release.bb
  - ncurses_6.1+20190803.bb
  - busybox_1.31.0.bb
  - glibc_2.30.bb

Configuration Files

- Config files contains
  - global definition of variables
  - user defined variables and
  - hardware configuration information
- Instructs build system what to build and put into the image to support a particular platform
- Extension: .conf
- Types
  - Machine Configuration Options
  - Distribution Configuration Options
  - Compiler tuning options
  - General Common Configuration Options
  - User Configuration Options (local.conf)

classes

- Class files are used to abstract common functionality and share it amongst multiple recipe (.bb) files
- To use a class file, you simply make sure the recipe inherits the class
- Eg. inherit classname
- Extension: .bbclass
- They are usually placed in classes directory inside the meta* directory

- Examples
  - cmake.bbclass - Handles cmake in recipes
  - kernel.bbclass - Handles building kernels. Contains code to build all kernel trees
  - module.bbclass - Provides support for building out-of-tree Linux Kernel Modules

Layers

- A collection/container (a directory) of related recipes.
- Typical naming convention: meta-
- Poky has the following layers:
  - meta, meta-poky, meta-selftest, meta-skeleton, meta-yocto-bsp.
- Why Layers
  - Layers provide a mechanism to isolate meta data according to functionality, for instance BSPs, distribution configuration, etc.
  - You could have a BSP layer, a GUI layer, a distro configuration, middleware, or an application
  - Putting your entire build into one layer limits and complicates future customization and reuse.
- Examples
  - meta -- OE-Core metadata layer
  - meta-poky -- Distro metadata (based on meta)
  - meta-yocto-bsp -- BSP metadata (based on meta-poky)
- Layers allow to easily to add entire sets of meta data and/or replace sets with other sets e.g. GNOME an be replaced by XFCE.
- Layers used by Poky build system are given by BBLAYERS variable present in build/conf/bblayers.conf file.
- If bblayers.conf is not present when you start the build, the OpenEmbedded build system creates it from bblayers.conf.sample, when you source the oe-init-build-env script.
- To see layers present in current image
  - $ bitbake-layers show-layers
- You can include any number of available layers from the Yocto Project
- Get the layers from
  - OpenEmbedded layers: https://layers.openembedded.org/layerindex/branch/master/layers/
  - Yocto Compatible layers: https://www.yoctoproject.org/software-overview/layers/

Image

- An image is the top level recipe, it has a description, a license and inherits the core-image class
- It is used alongside the machine definition
- Machine describes the hardware used and its capabilities
- Image is architecture agnostic and defines how the root filesystem is built, with what packages.
- By default, several images are provided in Poky.
- $ ls meta*/recipes*/images/*.bb

Packages

- In general, any wrapped or boxed object or group of objects.
- Yocto: A package is a binary file with name *.rpm, *.deb, or *.ipkg
- A single recipe produces many packages. All packages that a recipe generated are listed in the recipe variable PACKAGES.
- $ vim meta/recipes-core/kbd/kbd_2.0.4.bb
  - PACKAGES += "${PN}-consolefonts ${PN}-keymaps ${PN}-unimaps ${PN}-consoletrans"

Exploring Poky
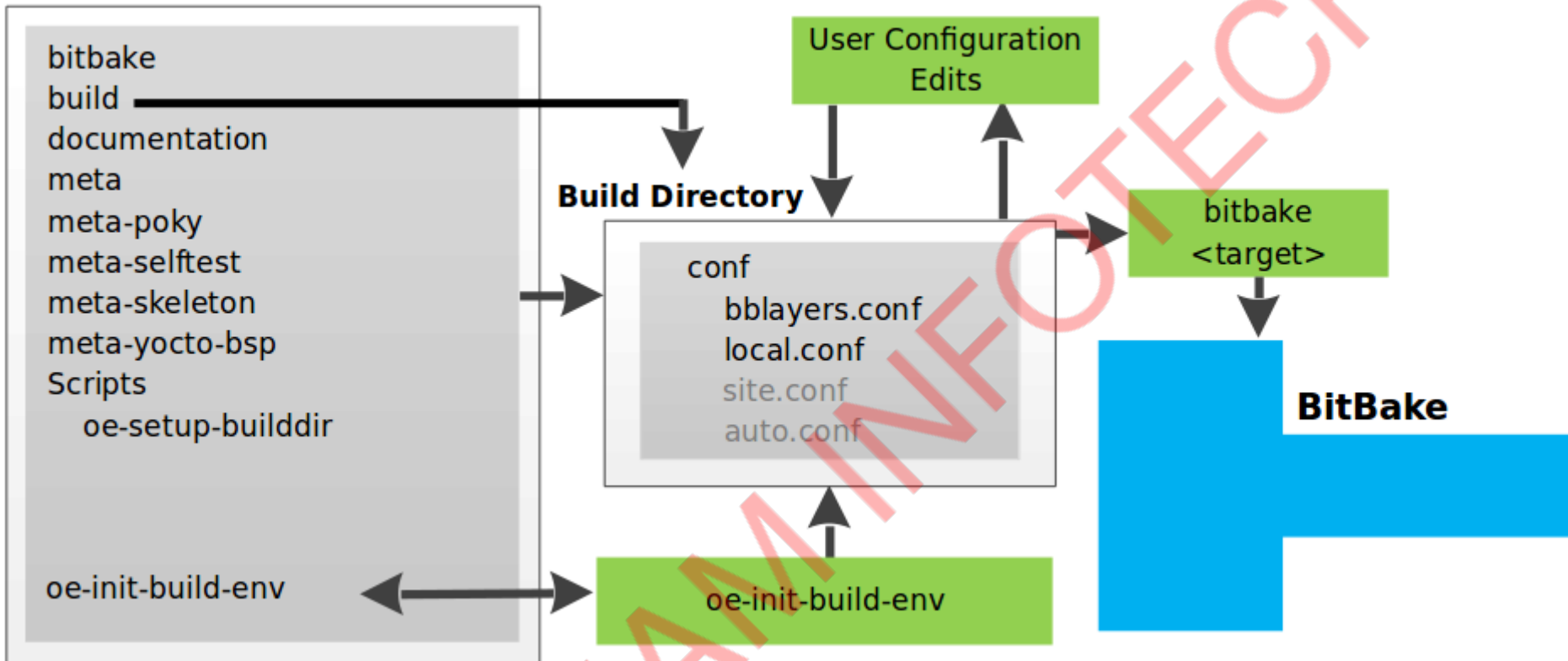
Poky source tree

- bitbake: Holds all Python scripts used by the bitbake command
    - bitbake/bin is placed into the PATH environmental variable so bitbake can be found
- documentation: All documentation sources for the Yocto Project documentation
    - Can be used to generate well-formatted PDFs
- meta: Contains the oe-core metadata
- meta-poky: Holds the configuration for the Poky reference distribution
    - local.conf.sample, bblayers.conf.sample are present here
- meta-skeleton: Contains template recipes for BSP and kernel development

- meta-yocto-bsp: Maintains several BSPs such as the Beaglebone, EdgeRouter, and generic versions of both 32-bit and 64-bit IA machines.
- scripts: Contains scripts used to set up the environment, development tools, and tools to flash the generated images on the target.
- LICENSE: The license under which Poky is distributed (a mix of GPLv2 and MIT).

Poky build directories

- downloads: downloaded upstream tarballs/git repositories of the recipes used in the build
- sstate-cache: shared state cache
- tmp: Holds all the build system output
    - tmp/deploy/images/machine - Images are present here
    - Details discussed in next section
- cache: cache used by the bitbake's parser
- conf: build configuration
    - When "source poky/oe-init-build-env" executed, it creates a "build" directory.
    - Inside this build directory, it will create "conf" folder which contains two files:
        - local.conf
        - bblayers.conf
    - **local.conf**
        - Configures almost every aspect of the build system
        - Contains local user settings
        - MACHINE: The machine the target is built for
            - Eg: MACHINE = "qemux86-64"
        - DL_DIR: Where to place downloads
            - During a first build the system will download many different source code tarballs,from various upstream projects.These are all stored in DL_DIR
            - The default location is a downloads directory under TOPDIR which is the build directory
        - TMP_DIR: Where to place the build output
            - This option specifies where the bulk of the building work should be done and BitBake should place its temporary files(source extraction, compilation) and output
        - local.conf file is a very convenient way to override several default configurations over all the Yocto Project's tools.
        - Essentially, you can change or set any variable, for example, add additional packages to an image file

- Note: though it is convenient, it should be considered as a temporary change as the build/conf/local.conf file is not usually tracked by any source code management system.
  - **bblayers.conf**
    - The bblayers.conf file tells BitBake what layers you want considered during the build.
    - By default, the layers listed in this file include layers minimally needed by the build system
    - However, you must manually add any custom layers you have created
    - e.g: BBLAYERS = "

      /home/linuxtrainer/poky/meta

      /home/linuxtrainer/poky/meta-poky

      /home/linuxtrainer/poky/meta-yocto-bsp

      /home/linuxtrainer/poky/meta-mylayer

      "
    - This example enables four layers, one of which is a custom user defined layer named "meta-mylayer"

Yocto Build Output - Images

- The build process writes images out to the Build Directory inside the tmp/deploy/images/machine/ directory
- kernel-image:
  - A kernel binary file
  - The KERNEL_IMAGETYPE variable determines the naming scheme for the kernel image file.
  - $ bitbake -e core-image-minimal | grep ^KERNEL_IMAGETYPE=
    - e.g. zImage
- root-filesystem-image:
  - Root filesystems for the target device (e.g. *.ext3 or *.bz2 files).
  - The IMAGE_FSTYPES variable determines the root filesystem image type
  - $ bitbake -e core-image-minimal | grep ^IMAGE_FSTYPES=
    - e.g. tar.bz2 ext4
- kernel-modules:
  - Tarballs that contain all the modules built for the kernel
- bootloaders:
  - If applicable to the target machine, bootloaders supporting the image.
- Note: symlinks are generated for most recent built files for each machine.

Yocto/OpenEmbedded Build System Workflow

1. Developers specify architecture, policies, patches and configuration details.
2. The build system fetches and downloads the source code from the specified location
   - supports downloading tarballs and source code repositories systems such as git/svn
3. Extracts the sources into a local work area
4. Patches are applied
5. Steps for configuring and compiling the software are executed
6. Installs the software into a temporary staging area
   - depending on the user configuration, deb/rpm/ipk binaries are generated
7. The build system generates a binary package feed that is used to create the final root file image.
8. finally generates the file system image and a customized Extensible SDK (eSDK) for application development in parallel.

Saving Disk Space while building Yocto

- Yocto Build System take a lot of disk space during build.
- Bitbake provides options to preserve disk space
- You can instruct bitbake to delete all the source code, build files after building a particular recipe by adding the following line in local.conf file
  - INHERIT += "rm_work"
- If you want to exclude bitbake deleting source code of a particular package, you can add it in RM_WORK_EXCLUDE += "recipe-name"
  - e.g: RM_WORK_EXCLUDE += "core-image-minimal"
- Disadvantage: Difficult to debug while build fails of any recipe.

Yocto for Beaglebone

Beagle Bone Black Specification

- Texas Instruments AM335x (ARM Cortex-A8 CPU)
- 512MB DDR3 RAM
- 4 GB of on-board eMMC storage
- 3D graphics accelerator
- NEON floating-point accelerator
- 2x PRU 32-bit micro-controllers

- USB client for power & communications
- USB host
- Ethernet
- HDMI (micro)
- 2x 46 pin headers with access to many expansion buses (I2C, SPI, UART and more)
- A huge number of expansion boards a.k.a. capes

Build Yocto Image for BeagleBone Black

- BeagleBone is one of the reference boards of Yocto Project
- $ source poky/oe-init-build-env build_bbb
- Open build_bbb/local.conf file
  - comment the default selection, which is the qemux86_64
  - uncomment the beaglebone selection

```
MACHINE ?= "beaglebone-yocto"
#MACHINE ??= "qemux86_64"
```

- Trigger build
  - $ bitbake core-image-minimal
- After the build is complete, you will have your images ready at tmp/deploy/images/beaglebone-yocto/
  - first-level bootloader MLO
  - second-level bootloader u-boot
  - kernel image
  - device tree blobs
  - a root filesystem archive
  - a modules archive

Booting Process in Beaglebone black

- The AM335x is powerful chip, but has limited internal RAM (128 kB).

- Because of this limited amount of RAM, multiple bootloader stages are needed.
- These bootloader stages systematically unlock the full functionality of the device so that all complexities of the device are available to the kernel.
- The AM335x has four distinct booting stages:
    1. ROM
    2. SPL (or Secondary Program Loader)
    3. u-BOOT
    4. Linux Kernel

ROM Bootloader (1st Level Bootloader)

- a.k.a. Primary Program Loader
- The first stage bootloader is flashed in ROM on the device by Texas Instruments.
- The ROM code is the first block of code that is automatically run on device start-up or after power-on reset (POR).
- The ROM bootloader code is hardcoded into the device and cannot be changed by the user.
- The ROM code has two main functions:
    - Configuration of the device and initialization of primary peripherals
        - Stack setup
        - Configure Watchdog Timer 1 (set to three minutes)
        - PLL and System Clocks configuration
    - Ready device for next bootloader
        - Check boot sources for next bootloader (SPL)
        - Moves next bootloader code into memory (SRAM) to be run
- The main purpose of the ROM code is to set up the device for the second stage bootloader.
- By default, the ROM code in the Sitara AM3359 will boot from the MMC1 interface first (the onboard eMMC), followed by MMC0 (external uSD), UART0 and USB0.
- If the boot switch (S2) is held down during power-up, the ROM will boot from the SPI0 Interface first, followed by MMC0 (external uSD), USB0 and UART0.

Secondory Program Loader / MLO (2nd Level Bootloader)

- a.k.a. 1st Stage Bootloader
- A fully featured version of U-Boot can be over 400KB, and the internal RAM on the AM335X is 128KB.
- Hence it is not possible to load this immediately

- For this reason, a cut down version of U-Boot called U-Boot SPL (Second Program Loader) is loaded first,
- Once it has initialized the CPU, it chainloads a fully featured version of U-Boot (u-boot.img).
- Name of SPL must be MLO (all-caps)
- It should be located on active first partition of MMC, which must be formatted as FAT12/16/32.
- https://stackoverflow.com/questions/31244862/what-is-the-use-of-spl-secondary-program-loader

u-Boot (3rd Level Bootloader)

- a.k.a. 2nd Stage Bootloader
- u-BOOT allows for powerful command-based control over the kernel boot environment via a serial terminal.
- The user has control over a number of parameters such as boot arguments and the kernel boot command.
- In addition, u-boot environment variables can be configured.
- These environment variables are stored in the uEnv.txt file on your storage medium.
- The built-in environment in u-boot loads a default am335x-boneblack.dts to pass to the kernel at boot.
- In uEnv.txt you can explicitly specify a different DTS as well as the command line arguments to pass to the kernel
- u-boot is also capable of obtaining network information via DHCP and loading it into environmental variables.
- Finally u-boot loads the kernel and a DTS into memory and boots the kernel with some command line arguments
- The kernel initializes and mounts the root filesystem.
- By default, the root filesystem is contained in the second partition (mmcblk0p2) of the microSD card, formatted for an ext3 file system.

Creating partitions and formatting the SD card

- step 1. Attach SD card to PC and unmount any auto-mounted partition, using the umount command:
  - $ umount /dev/sdb1
- step 2. Launch the fdisk utitility and delete all partition(s); If only one partition:
  - $ sudo fdisk /dev/sdb
    - Command (m for help): d
    - Selected partition 1
- step 3. Create new partition called BOOT of 32 MB (30 M+) and type primary:
  - Command (m for help): n
  - Partition type:
    - p primary (0 primary, 0 extended, 4 free)

- e extended
  - Select (default p):
  - Using default response p
    - Partition number (1-4, default 1):
    - Using default value 1
    - First sector (2048-7774207, default 2048):
    - Using default value 2048
    - Last sector, +sectors or +size{K,M,G} (2048-7774207, default 7774207): +32M
- step 4. Create a second partition (200 M+) to hold rootfs. We will give all the remaining space to this partition:
  - Command (m for help): n
  - Partition type:
    - p primary (1 primary, 0 extended, 3 free)
    - e extended
  - Select (default p):
    - Using default response p
    - Partition number (1-4, default 2):
    - Using default value 2
    - First sector (67584-7774207, default 67584):
    - Using default value 67584
    - Last sector, +sectors or +size{K,M,G} (67584-7774207, default 7774207):
    - Using default value 7774207
- step 5. Make the first partition bootable by setting the boot flag:
  - Command (m for help): a
  - Partition number (1-4): 1
- step 6. Set the first partition as WIN95 FAT32 (LBA): (Important)
  - Command (m for help): t
  - Selected partition 1 Hex code (type L to list codes): c
- step 7. We are done with the filesystem modification. So, let's write it by issuing the w command:
  - Command (m for help): w
  - The partition table has been altered!
  - Calling ioctl() to re-read partition table.

- Syncing disks.
- step 8. Format the first partition as FAT, using the following command. We will set the label as BOOT so that we know what directory it will be mounted to by udisks:
  - $ sudo mkfs.vfat -n "BOOT" /dev/sdb1
- step 9. Format the second partition as an ext4 filesystem, using the following command. The label for this is set to ROOT, as it will contain the extracted image of rootfs.
  - $ sudo mkfs.ext4 -L "ROOT" /dev/sdb2

Deploy images to the card

- step 0: The partitions are usually auto mounted under /media/$USER
  - If not, we can use the mount command to mount the partition to our desired location:
    - $ sudo mount /dev/sdb1 /media/$USER/BOOT
    - $ sudo mount /dev/sdb2 /media/$USER/ROOT
- step 1. Copy the u-boot MLO and u-boot bootloader images into the FAT32 partition:
  - $ sudo cp MLO /media/$USER/BOOT
  - $ sudo cp u-boot.img /media/$USER/BOOT
- step 2. Copy the kernel image into the boot partition:
  - $ sudo cp zImage /media/$USER/BOOT
- step 3. Copy the .dtb file, am335x-boneblack.dtb, into the boot partition.
  - $ sudo cp am335x-boneblack.dtb /media/$USER/BOOT
  - This step is required only in the case of core-image-minimal. It is not required if created a core-image-sato, which already has this file placed at the desired location in rootfs.
- step 4. As a root user, uncompress core-image-minimal-beaglebone.tar.bz2 to the ext4 partition:
  - $ sudo tar -xf core-image-minimal-beaglebone-yocto.tar.bz2 -C /media/$USER/ROOT/
- step 5. Unmount both partitions:
  - $ sudo umount /dev/mmcblk0p1
  - $ sudo umount /dev/mmcblk0p2
- step 6. Remove the card from the host machine, and insert it into the SD card slot on BeagleBone Black.

Test image on BeagleBone Black

- step 1. Connect USB TTL-2303(PL2303) for serial communication

    - USB-TTL is connected to the J1 connector of BeagleBone in the following formation:
        - J1 Pin ---> USB TTL Function
        - 1 ---> GND Ground
        - 4 ---> RXL
        - 5 ---> TXL

- step 2. Setup minicom on PC.

    - BeagleBone Black uses a serial debug port to communicate with the host machine. We will use minicom as a serial terminal client to communicate over the serial port. To set up minicom, perform the following steps:
    - Run this setup command as a privileged user:
        - $ sudo minicom -s
        - Set baud rate to 115200 8N1.
        - Set hardware flow control and software flow control to No.
    - Save setup as dfl to avoid reconfiguring every time.

- step 3. While pressing S2 switch, connect power to board via 5V power supply or mini-USB cable. Observe the log.

```
Booting from mmc ...
## Booting kernel from Legacy Image at 82000000 ...
   Image Name:   Linux-3.14.0-yocto-standard
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4985768 Bytes = 4.8 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 88000000
   Booting using the fdt blob at 0x88000000
   Loading Kernel Image ... OK
   Loading Device Tree to 8fff5000, end 8ffff207 ... OK
   Starting kernel …
```

```
Poky (Yocto Project Reference Distro) 1.6.1 beaglebone /dev/ttyO0
beaglebone login: root
root@beaglebone:~#
```