**Trainings**

1. Languages:

- [C]  <https://lnkd.in/d7hdTeFp>

1. CPUs:

- [ARM-M]  <https://lnkd.in/d62SH6me>

- [ARM-A]  <https://lnkd.in/dKF3NKu9>

- [RISC-V and CPU design]  <https://lnkd.in/dnvTYuz2>

1. DSP:

- [Think DSP]  <https://lnkd.in/dNvTtU_i>

- [Implementation and Theory]  <https://lnkd.in/dw7VK4qy>

- [Fourier transform and applications]  <https://lnkd.in/diKf--w2>

1. OS:

- [unix on risc-v]  <https://lnkd.in/e87aZ56b>

- [Linux on RPi]  <https://lnkd.in/dXzBXyNW>

1. FPGA-based Design:

-  <https://lnkd.in/drMRkajM>

-  <https://lnkd.in/deNk2_u5>

- [hardware software co-design]  <https://lnkd.in/dEKKTPJs>

1. DS/A (Bonus)

-  <https://lnkd.in/djkUZ3-x>

////////////////////////////////////////////////////////////////////////////////////////////////

* 1. Clone the busybox
     1. git clone git://busybox.net/busybox.git
  2. export ARCH=arm
  3. export CROSS\_COMPILE=arm-linux-gnueabi-
  4. make -j8 defconfig
  5. Enable static linking in Busybox
     1. make menuconfig
     2. Select Build BusyBox as a static binary (no shared libs)
  6. make -j8
  7. make install
  8. Create a dir **arm-busybox** , where we store our rootfs
     1. mkdir -pv arm-busybox
     2. Cd arm-busybox
     3. mkdir -pv {bin,dev,sbin,etc,proc,sys/kernel/debug,usr/{bin,sbin},lib,lib64,mnt/root,root}

* 1. copy \_intsall from busybox to arm-busybox dir
     1. cp -av ../\_install/\* .
     2. sudo cp -av /dev/{null,console,tty,sda1} arm-busybox/dev/
  2. Create init and make it executable
     1. Create a init in arm-busybox dir
     2. Copy below in init

#!/bin/sh  
mount -t proc none /proc  
mount -t sysfs none /sys  
mount -t debugfs none /sys/kernel/debug  
   
echo -e "\nBoot took $(cut -d' ' -f1 /proc/uptime) seconds\n"  
echo "This is Busybox created for LSG build"   
exec /bin/sh

* 1. Make init as executable
     1. Chmod +x init
  2. Create a initramfs
     1. find . | cpio -H newc -o > initramfs.cpio
     2. find . | sort | cpio --reproducible -o -H newc > initramfs.cpio [LSG]

* 1. Extract the initramfs
     1. cpio -id < **<root-fs>**

**Extraxt initramfs**

* 1. Create a directory
     1. Mkdir initramfs
     2. Cd initramfs
  2. Extract the cpio
     1. Cpio -id < <root-fs>

**OpenBmc (mtd image Generation)**

dd if=/dev/zero bs=1k count=**65536** | tr '\000' '\377' > **fit.img**

dd bs=1k conv=notrunc **seek=0** if=/mnt/work/Yocto/ast\_openbmc/openbmc/build/evb-ast2600/tmp/deploy/images/evb-ast2600/**u-boot-spl.bin** of=**fit.img**

dd bs=1k conv=notrunc **seek=64** if=/mnt/work/Yocto/ast\_openbmc/openbmc/build/evb-ast2600/tmp/deploy/images/evb-ast2600/**u-boot.bin** of=**fit.img**

#dd bs=1k conv=notrunc **seek=1024** if=/mnt/work/Yocto/ast\_openbmc/openbmc/build/evb-ast2600/tmp/deploy/images/evb-ast2600/**fitImage-obmc-phosphor-initramfs-evb-ast2600-evb-ast2600** of=**fit.img**

dd bs=1k conv=notrunc seek=**10240** if=/mnt/work/Yocto/ast\_openbmc/openbmc/build/evb-ast2600/tmp/work/evb\_ast2600-openbmc-linux-gnueabi/obmc-phosphor-image/1.0-r0/deploy-obmc-phosphor-image-image-complete/**obmc-phosphor-image-evb-ast2600.squashfs-xz** of=**fit.img**

dd bs=1k conv=notrunc seek=**43008** if=/mnt/work/Yocto/ast\_openbmc/openbmc/build/evb-ast2600/tmp/work/evb\_ast2600-openbmc-linux-gnueabi/obmc-phosphor-image/1.0-r0/deploy-obmc-phosphor-image-image-complete/**obmc-phosphor-image-evb-ast2600.jffs2** of=**fit.img**

**Try:**

* 1. Generate fit image using mkimage tool (copy zImage , dtb and rfs from OpenBmc)
     1. Result: Able to load on qemu
  2. Now took rootfs from OpenBmc and compressed with find . | cpio -H newc -o > initramfs.cpio Create a fit image and mtd image
     1. Result: Able to load on qemu
  3. In rootfs changed the init (basic one just printing hello word) and create initramfs and mtd image
     1. Result : Kernel Panic

* 1. require conf/machine/include/tune-cortexa7.inc
     1. DEFAULTTUNE = nothing
     2. Result : Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000004

* 1. require conf/machine/include/tune-cortexa7.inc
     1. DEFAULTTUNE = "armv7ahf-vfpv4d16"
     2. Result : Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000004

* 1. require conf/machine/include/tune-cortexa7.inc
     1. TUNE\_PKGARCH = "armv7ahf-vfp"
     2. Result : Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000004

* 1. require conf/machine/include/tune-cortexa7.inc
     1. DEFAULTTUNE = "cortexa7thf-neon"
     2. Result : Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000004

* 1. include conf/machine/include/tune-cortexa7.inc // ST32MPU
     1. DEFAULTTUNE = "cortexa7thf-neon-vfpv4"
     2. Result : Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000004

* 1. require conf/machine/include/arm/arch-armv7a.inc
     1. DEFAULTTUNE = "cortexa7thf-neon-vfpv4"
     2. Result : Compilation error

///////////

* 1. Compile the init with necessary flags
  2. Install : creating the dir and copying to images (dev mnt init)

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

**U-Boot**

**mkimage -ftest.its fit.bin**

**U-boot:**

* 1. git clone <https://gitlab.denx.de/u-boot/u-boot>
  2. Openbmc kernel:
     1. <https://github.com/openbmc/u-boot.git>
  3. export CROSS\_COMPILE=arm-linux-gnueabi-
  4. export ARCH=arm
  5. make evb-ast2600\_defconfig
  6. Make -j4
  7. make ast2600\_openbmc\_spl\_defconfig
  8. <https://shenki.github.io/debugging-u-boot-after-relocation/> : Debugging
  9. Command to debugging
  10. CONFIG\_SYS\_PROMPT="ast# " // we change the u-boot promot
  11. **CONFIG\_BOOTCOMMAND** // define the bootm command address

**Creating a full image (u-boot + fit-image) [source : meta-msft ]**

* 1. dd if=/dev/zero of=test-2600.img bs=1024 count=0 seek=32768
  2. dd bs=1k conv=notrunc seek=0 if=u-boot.bin of=test-2600.img
  3. dd bs=1k conv=notrunc seek=512 if=fitImage-initramfs-msft-tiny of=test-2600.img

* 1. **AST2500**
     1. Denx u-boot is not working with fit-image of msft-kernel
     2. Openbmc u-boot is working with fit-image of msft-kernel
     3. dd if=/dev/zero of=test.img count=32 bs=1M (count : 32mb image)
     4. dd if=u-boot.bin of=test.img conv=notrunc seek=0
     5. dd if=fitimage.bin of=test.img bs=1 conv=notrunc seek=1048576 (copied fit image to test.img after 1MB)
     6. ../qemu/qemu/build/qemu-system-arm -M ast2500-evb -nographic -drive file=test.img,format=raw,if=mtd,readonly -s -no-reboot
     7. bootm 0x20080000 (address define in config file of ast2500 on u-boot **CONFIG\_BOOTCOMMAND**)

* 1. **AST2600**
     1. Build the openbmc bootloader
     2. make ast2600\_openbmc\_spl\_defconfig
     3. If error comes:
        1. CONFIG\_DEFAULT\_DEVICE\_TREE="ast2600-evb"
     4. There are 2 bootloader need to boot the ast2600 (u-boot.bin and spl\u-boot-spl.bin)
     5. dd if=/dev/zero of=test.img bs=1M count=64
     6. dd if=spl/u-boot-spl.bin of=test.img bs=1 conv=notrunc
     7. dd if=u-boot.bin of=test.img bs=1 seek=65536 conv=notrunc
     8. dd if=fitimage.bin of=test.img bs=1 conv=notrunc seek=1048576
     9. ./disk-img.sh $i/p fit image $opdisk image
     10. ../qemu/qemu/build/qemu-system-arm -M ast2600-evb -nographic -drive file=test.img,format=raw,if=mtd,readonly -s -no-reboot

* 1. Using **linux.bin and dtb (Yocto openbmc)** + initramfs (msft layer for ast2600)

* 1. Questions:
     1. Why GPIO and I2C conf are enabled in u-boot
  2. **Relocation:**
     1. Copy u-boot code from flash to RAM for faster execution
     2. On most systems now, relocation is done by calculating the delta between the ROM address (TEXT\_BASE) and the relocated address in RAM. The relocation code first copies the entire U-Boot image to RAM (apart from a few structures that are used only during the relocation calculation). It then scans the relocation entries (\*.rel sections) to find where in the U-Boot executable all the references into the data sections. At each reference point in the executable, the relocation code modifies the

reference by the relocation offset. None of this can happen if the U-Boot executable is in Flash.

* 1. The other advantage of relocating U-Boot to RAM is that it makes it possible to flash a new U-Boot image from U-Boot. If you try to do this while U-Boot is running from flash, it will crash as you are overwriting the executable code while it's executing.

**I2C Bus access or Reading data from**

* 1. Find device with using bus
     1. uclass\_get\_device\_by\_seq(UCLASS\_I2C, <bus num>, &bus);
  2. Probe the device on particular bus
     1. dm\_i2c\_probe(bus, <i2c\_addr>, 0, &dev);
  3. Set offset length on device , by default its 1byte
     1. i2c\_set\_chip\_offset\_len(dev, <offset\_len>);
  4. Now we are able to read and write from I2C bus
     1. dm\_i2c\_write
     2. dm\_i2c\_read

**////////////////////////////////////////////////////////////////////////////////////**

**Reference :**

1. [**https://software-dl.ti.com/processor-sdk-linux/esd/docs/06\_03\_00\_106/AM437X/linux/How\_to\_Guides/Board\_Port/U-Boot.html**](https://software-dl.ti.com/processor-sdk-linux/esd/docs/06_03_00_106/AM437X/linux/How_to_Guides/Board_Port/U-Boot.html)
2. Bootlin

<https://elinux.org/images/2/2a/Schulz-how-to-support-new-board-u-boot-linux.pdf>

1. LSG\_u-boot
   1. <https://dev.azure.com/AzureCSI-SE/Overlake/_git/uboot?path=%2F&version=GBuboot-msft-bmc-v2020.07&_a=contents>

1. U-boot driver
   1. <https://lxr.missinglinkelectronics.com/uboot+v2018.07/doc/driver-model/README.txt>

**Intro:**

A bootloader, also known as a boot program or bootstrap loader, is a **special operating system software** that loads into the working memory of a computer after start-up. For this purpose, immediately after a device starts, a bootloader is generally launched by a bootable medium like a hard drive, a CD/DVD or a [USB stick](https://www.ionos.com/digitalguide/server/know-how/make-a-bootable-usb-drive/). The boot medium receives information from the computer’s **firmware** (e.g. BIOS) about where the bootloader is. The whole process is also described as “booting”.

**BootRom code:**

1. Bootrom (or Boot ROM) is a small piece of mask ROM or write-protected flash embedded inside the processor chip. It contains the very first code which is executed by the processor on power-on or reset.
2. BootRom code reside inside the SOC flash.
3. ROM code looks for bootloader (MLO [Memory Loader] or in ast2600 it would be SPL)

**Why MLO or SPL:**

1. SPL is required when the system internal memory can not hold the uboot completely so we need to initialize memory using a minimal piece of code called SPL.
2. Rom code is copies the content of MLO / SPL to SRAM (on chip memory , that is very less)
3. SRAM memory is limited (due to physical reasons). Usually regular bootloader (e.g. U-Boot) binary is bigger than that. So we need to create some additional bootloader, which will initialize regular RAM and copy regular bootloader from MMC to RAM, and then will jump to execute that regular bootloader. This additional bootloader is
4. usually referred as *first-stage bootloader* (in two-stage bootloader scenario) [SPL or MLO]

**Boot 
stage 
number 
Terminology 
pr nary 
Program 
Loader 
Secondary 
Program 
Loader CPL) 
Terminology 
1st stage 
boottoader 
2nd stage 
bootloader 
Actual 
program 
name 
ROM code I 
u—boot 
SPL 
u—boot 
kernel **

1. It performs the initial CPU and board configuration (clocks and DDR memory).
2. It loads the SSBL (U-Boot) into the DDR memory.
3. boot sequence is next: ROM code -> SPL -> u-boot -> kernel.

CONFIG\_SYS\_TEXT\_BASE == relocation address! This prevents that uboot code

is copied again in relocate\_code().

*From <*[*https://github.com/ARM-software/u-boot/blob/master/doc/README.arm-relocation*](https://github.com/ARM-software/u-boot/blob/master/doc/README.arm-relocation)*>*

**U-boot Code**

1. **Arch :** anything arch or platform related: DTS, CPU init, pinmux controller, DRAM, clocks, …
2. **Board** : code board specific (init, pinmuxing configuration, etc), Kconfig file specifying board header file, board file, paths, Makefile for board file,
3. **configs** : all board configs
4. Drivers : all peripheral's drivers
5. Include : all header

**Adding a new Board**

1. Doc : [**https://software-dl.ti.com/processor-sdk-linux/esd/docs/06\_03\_00\_106/AM335X/linux/How\_to\_Guides/Board\_Port/U-Boot.html**](https://software-dl.ti.com/processor-sdk-linux/esd/docs/06_03_00_106/AM335X/linux/How_to_Guides/Board_Port/U-Boot.html)
2. **MYBOARD** : Name identifying your custom board in upper-case letters. It will be used as a name for a new U-Boot CONFIG symbol associated with your board allowing to customize various build and runtime aspects.
3. **myboard** : Name identifying your custom board in lower-case letters. Used to establish the custom board platform files in the U-Boot source tree hierarchy and as part of board-specific file names, amongst other things
4. mycompany : Name of your company. Used to establish a folder in the U-Boot source tree hierarchy containing the board-specific files for <myboard> as part of the board port.

1. Create the board file
   1. My\_board.c
      1. **board\_init()**
         1. Initialization of board (call necessary func if required)
         2. Board specific
            1. Power setting
            2. GPIO setting
      2. **dram\_init(void)**
         1. Must pass the dram size and base address
      3. **DECLARE\_GLOBAL\_DATA\_PTR;**
         1. Represent by **gd** variable
         2. that a resource (for example, a CPU register) will be reserved for pointing to a struct global\_data
         3. The global data pointer gives easy access to the data elements which are most useful in this boot phase
         4. Defined in include/asm-generic/global\_data.h
2. Create the board Kconfig file,
   1. Define target config TARGET\_MY\_BOARD
   2. SYS\_BOARD, SYS\_VENDOR, SYS\_CONFIG\_NAME
3. Create the board Makefile,
   1. File to compile
4. Create the board defconfig,
   1. Define the board configuration
   2. Placed in config folder
5. Create the board header file,
   1. **Path** : include/configs/board.h
   2. Contains all global variables (constant)
   3. All board related info
6. Source board’s Kconfig in the architecture’s Kconfig, (source Kconfig in arch/arm)
7. Define the TARGET Kconfig option in its CPU’s Kconfig,

**///////////////////////////////////////////////////////////////////////////////////////////////////////**

**U-boot Debugging**

**Envoke Qemu**

/mnt/work/qemu/qemu/build/qemu-system-arm -M ast2600-evb -nographic -drive file=test.img,format=raw,if=mtd,readonly **-s -S** -no-reboot

/mnt/work/qemu/qemu/build/qemu-system-arm -M palmetto-bmc -nographic -drive file=test.img,format=raw,if=mtd,readonly **-s -S** -no-reboot

**Machine:**

Palmetto-bmc : ast2400

**Run GDB in another Terminal**

gdb-multiarch u-boot -ex "target remote localhost:1234"

/mnt/work/qemu/qemu/build/qemu-system-arm -M ast2600-evb -nographic -drive file=test.img,format=raw,if=mtd,readonly **-s -S** -no-reboot

**Debugging u-boot in Qemu with GDB after relocation:**

1. Break the code at relocate\_done func
   1. b relocate\_done
   2. Continue ( c )
2. Print the relocate addr to load the symbol file

p ((gd\_t \*)$r9)->relocaddr

1. Load the symbol file to corresponding address
   1. add-symbol-file u-boot $1

**(gdb) b reloacte_done 
Function "reloacte done" not defined. 
Slake breakpoint pending on future shared library load? (y or (n] ) 
(gdb) b relocate_done 
Breakpoint 1 at 
(gdb) c 
:ont inuing. 
file arch/arm/lib/relocate.s, line 134. 
rhread I hit Breakpoint I, relocate _ code 
(gdb) p ( (gd t $r9) ->relocaddr 
32aan2686R 
(gdb ) 
symbol table f rom file "u-boot" at 
. text addr = Oxbef99000 
(y or n) Y 
Readin symbols from u -boot... 
at arch/ arm/ lib/ relocate.S:134 **

**GDB command:**

1. Step in to func : s
2. Next line : n
3. Set breakpoint : b "funcname"
4. Remove breakpoint

clear linenum  
clear filename:linenum

**///////////////////////////////////////////////////////////////////////////////**

**DTS:**

**Without DTS:**

1. Bootloader passes the board information like arch type, memory size and other information needed to boot the kernel with register r1 (ATAGS) and r2(machine type)
2. The register address and other peripherals information is hardcoded in kernel (drivers)
3. The kernel image is specific to each target board

**With DTS file**

1. Bootloader passes DTB address in r2 register and no machine type information is passed, it defined in dts file
2. A device tree is a tree data structure with nodes that describe the physical devices in a system.
3. Define the Basic hardware information of board like CPU, peripherals info, controller info and their connection
4. No need to hardcode the register address and other peripheral's info .
5. Each module in device tree is defined by a node and all its properties (memory info, register info, interrupt info ) are defined under that node. Depending on the driver it can have child nodes or parent node.
   1. For example a device connected by i2c bus, will have i2c as its parent node, and that device will be one of the child node of i2c node, i2c may have apd bus as its parent and so on. All leads up to root node, which is parent of all.
6. **chosen** : defines parameters chosen or defined by the system firmware at **boot time**. In practice, one of its usage is to pass the kernel command line.

**Device Tree organization: top-level nodes 
Under the root of the Device Tree, one typically finds the following 
top-level nodes: 
cpus node, which sub-node* describing each CPU in the 
system. 
node, which defines the location and size of the 
memory 
RAM. 
node, which defines parameters chosen or defined 
chosen 
by the system firmware at boot time. In practice, one of its 
usage is to pass the kernel command line. 
node, to define shortcuts to certain nodes. 
aliases 
One or more nodes defining the buses in the SOC. 
One or mode nodes defining on-board devices. **

1. **Basic 
   Device Tree s ntax 
   Node name 
   Unit address 
   roperty name 
   Property value 
   node@O { 
   a-string-property = "A string"; 
   a-string-list-property "first string", "second string"; 
   properties Of node@O 
   a-byte-data-property = (Ox01 ex23 ox34 ox561; 
   child-node@ø { 
   first-child-property; 
   second-child-property = 
   Bytestring 
   Label 
   a-reference-to-something 
   A phandle 
   (reference to another node) 
   nodel: node@l { 
   an -empty-property; 
   a-cell •property = 2 3 
   child-node@O { 
   Four cells (32 bits values) **

**A simple example, DT side 
auartO : 
seriat@8006aOOO { 
Defines the "programming model" for the device. Allows the 
operating system to identify the corresponding device driver. 
compatible = "fsI,imx28-auart", "fst, imx23-auart" ; 
Address and length of the register area. 
reg = <Ox8006aOOO 
Interrupt number. 
interrupts = <112>; 
DMA engine and channels, with names. 
dmas = 8>, 9>; 
dma-names = "rx", "tx"; 
Reference to the clock. 
clocks = 45>; 
The device is not enabled. 
= disabled"; 
status ' **

1. **#address-cells :**  property indicate how many cells (i.e 32 bits values) are needed to form the base address part in the reg property
2. **$Size cell :**

///////////////////////////////////////////////////////////////////////////////////////////////////////

Driver Model

Link : <https://u-boot.readthedocs.io/en/latest/develop/driver-model/index.html>

Design Principle : <http://www.denx.de/wiki/U-Boot/DesignPrinciples>

Good : <https://elinux.org/images/c/c4/Order_at_last_-_U-Boot_driver_model_slides_%282%29.pdf>

[https://docs.google.com/document/d/1\_zZLey1JcYvW9RcuzxbCzipLdfr5SVyWEdh7sHhuAWg/edit#](https://docs.google.com/document/d/1_zZLey1JcYvW9RcuzxbCzipLdfr5SVyWEdh7sHhuAWg/edit)

Design Principle:

* 1. Small
     1. its primary purpose in the shipping system is to load some operating system. That means that U-Boot is necessary to perform a certain task, but it's nothing you want to throw any significant resources
     2. Nor flash is more expensive, so keep it small (not more than 128KB)
  2. Fast
     1. The end user is not interested in running U-Boot. In most embedded systems he is not even aware that U-Boot exists. The user wants to run some application code, and that as soon as possible after switching on his device
     2. Don’t do much more initialization , data xfer or any other thing that takes time
  3. Simple
     1. Easy to port, use and validate
  4. Portable
  5. Configurable
     1. Easy to configure acc to our board
  6. Debug gable
  7. Useable
  8. Maintainable
  9. Beautiful
  10. OPen

**Hard to scale:**

However it is hard to scale with this approach. As SoCs have become more complex, U-Boot's lack of a built-in driver model has made it harder to support all use cases. In the above example, if an SoC supports two different I2C controllers (e.g. normal 400KHz and fast 1MHz) then these need to be munged into a single driver, with the bus number used to distinguish between them. Similar situations arise with SoCs that supports both USB2 and USB3. In some cases the limitation is that there can be only one. For example it is not possible to support multiple displays in U-Boot (although this might be more of a commentary on how rarely it matters in a boot loader). It was painful to support I2C GPIO expanders using the generic U-Boot GPIO API (e.g. the 'gpio' command) if there were other GPIOs in the system.

**Configuration**

Configuration was through CONFIG\_... #defines in the board configuration file. This made it very quick and easy to add a new option that controls board-specific or generic code. But it led to a very large number of options (around 6000 at its peak in 2014) and quite a bit of #ifdef'd code. CONFIG options provided the address of a peripheral, the number of I2C buses, the bus address of an I2C PMIC, the architecture, the type of CPU, the PCI IDs of supported SATA controllers, the size of the malloc() region and so on.

Struct driver :

Include/dm/device.h

*of\_to\_plat: Called before probe to decode device tree data*

*@priv\_auto: If non-zero this is the size of the private data \* to be allocated in the device's ->priv pointer. If zero, then the driver \* is responsible for allocating any data required.*

//////////////////////////////////////////////////////////////////////////////////////////////////////////////

**Yocto:**

Require : local file

Include : poky related file

**Inherit :** directive to inherit the functionality of a class (.bbclass)

Example:

**Inherit autotools**

In this case, BitBake would search for the directory **classes/autotools.bbclass** in BBPATH. The class file would contain common functionality for using Autotools that could be shared across recipes

[**https://www.yoctoproject.org/docs/2.5/kernel-dev/kernel-dev.html**](https://www.yoctoproject.org/docs/2.5/kernel-dev/kernel-dev.html)

Bitbake command

1. bitbake virtual/kernel -c menuconfig
2. Bitbake core-image-minimal

Various operators can be used to assign values to configuration variables: 
— expand the value when using the variable 
immediately expand the value 
append (with space) 
prepend (with space) 
append (without space) 
prepend (without space) 
assign if no other value was previously assigned 
same as previous, with a lower precedence 

A picture containing text, screenshot, font, diagram

Description automatically generated

Linux Build System Tools: Yocto/OpenEmbedded, Buildroot, OpenWRT,

**Yocto:** Build a complete linux distribution with binary packages, but somewhat complex

**BuildRoot** : Build a root file system, no binary packages. Much simpler to use and modify.

**YOCTO:**

1. Open source project
2. Help to build a linux distribution
3. Its allow to build a custom linux
4. Support all major arch (x86, MIPS, arm, PPC etc)

**Build System: POKY**

1. Poky = bitbake + metadata
2. **Bitbake:**
   1. Build engine, task executer,
   2. Manage all build steps.
   3. it interpretate the recipe (metadata) and conf file to perform set of task [download, configure and build specific task (kernel, bootloader, root file system or SDK)].
3. **Metadata:**
   1. Set of base layers (recipe, layers and classes)
   2. Conf file (.conf) : all global variable definition
   3. Class (.bbclass) : define build logic and packaging
   4. recipe (.bb) : define individual piece of software/image to build.

The Yocto Project 
Pm"ct 
lexicon 
Project 
O*nEmbedded Core 
reci p 
n ity 
reci p 
m et bsp 
meta qts 
recip 
Entity 
Layer 
ADT 
R«ipø 
0th" 
meta 
meta ti 
r«ipes- t 
reci 
reci phics 

**OpenEmbedded Core**

**Recipe:**

1. Repo or path for source code to build
2. Patches to apply
3. Dependencies from other recipe or library
4. Conf and compilation options

**Layers:**

Set of recipes , matching for set of purpose (above example for TI, meta-ti is layer and had set of recipes)

**POKY**

poky 
build 
bitbake 
documentation 
meta 
meta-yocto 
meta-yocto-bsp 
meta-self-test 
meta-skeleton 
scripts 
conf 
downloads 
tmp 
deploy 
sysroots 
work 
Configuration files for the 
build environment 
downloaded upstream 
source tarballs 
deployed images, sdk and 
packages 
shared header files and 
libraries for share files 
between packages 
where all packages has its 
own directory and where 
bitbake unpack, patch, 
configure and compile 

poky 
build 
bitbake 
bitbake executable 
documentation Yocto project documentation 
meta 
meta-yocto 
meta-yocto-bsp 
meta-self-test 
meta-skeleton 
Layers 
Yocto scripts for extra functionalities (like qemu, hob, . 

poky 
build 
bitbake 
documentation 
meta 
meta-yocto 
meta-yocto-bsp 
meta-self-test 
meta-skeleton 
scripts 
Contains the OpenEmbedded Core metadata. 
conf 
classes 
Core recipes 
Core set of configuration files 
Contains the *.bbclass files 
that are used to abstract 
common code so it can be 
reused by multiple 
packages 

bitbake/ Holds all scripts used by the BitBake command. Usually matches the 
stable release of the BitBake project. 
documentation/ All documentation sources for the Yocto Project documentation. Can 
be used to generate nice PDFs. 
meta/ Contains the OpenEmbedded-Core metadata. 
meta-skeleton/ Contains template recipes for BSP and kernel development. 

meta-poky/ Holds the configuration for the Poky reference distribution. 
meta-yocto-bsp/ Configuration for the Yocto Project reference hardware board 
support package. 
LICENSE The license under which Poky is distributed (a mix of GPLv2 and MIT). 
oe-init-build-env Script to set up the OpenEmbedded build environment. It will create 
the build directory. It takes an optional parameter which is the build 
directory name. By default, this is 
. This script has to be sourced 
build 
because it changes environment variables. 
scripts Contains scripts used to set up the environment, development tools, 
and tools to flash the generated images on the target. 

**Local.conf**

**Recipe:**

1. It describe how to handle an application
2. Its set of instruction to describe fetch, retrieve, compile, install and generate the binary packages for a given applications.
3. Its are parsed by bitbake build engine
4. Format: application\_name .bb
   1. Contain recipe variables (name, license, dependencies)
   2. Path to retrieve source code
5. Variables
   1. PN: package name
   2. PV: package version
   3. PR: package revision (exp: r0)
   4. Example: recipe linux\_4.3.bb
      1. ${PN} = "linux"
      2. ${PV} = "4.3"
6. Recipe-uboot
   1. Uboot.inc
   2. Uboot\_<vesrion>.bb
7. An application had more than one recipe (some are common and will be used by some recipe too).
   1. To handle these kind of recipe, we will include these recipe to main recipe
8. There are 3 parts a recipe will devide
   1. **Header : what/who** 
      1. Describe the Configuration variables
         1. DESCRIPTION:
         2. HOMEPAGE: url for
         3. LICENCE
   2. **Source: where**
      1. SRC\_URI : define where and how to retrieve the needed element (local or remote)
      2. Local : [file://](NULL)
         1. These are copied to build/tmp/work
      3. Remote (Git, SVN, ftp )
         1. Git:// , svn:// ftp://
         2. An md5 or sha256 must be provided when the protocol used to retrieve the file
         3. To generate the md5, use checksum for that tar file or patch file
      4. By default the source are fetched in build/download dir
      5. If source are licence
         1. License had their own checksum
         2. LIC\_FILES\_CHKSUM : source path and as well as checksum
      6. **Dependencies**
         1. A recipe have dependencies during the build or at runtime, to reflect these use 2 variables
            1. DEPENDS : build time dependencies
            2. RDEPENDS : run time
            3. the local 
               task depends on the 
               " recipe-b": 
               do_configure 
               DEPENDS = 
               task of recipe-b. 
               do_populate_sysroot 
               the local 
               "recipe-b": 
               do_build 
               task depends on the 
               task of recipe b. 
               do_package_wri format> 
         2. Sometimes recipe depends on specific version
            1. DEPENDS = recipe-b (>=1.2)

1. **Tasks: how**
   1. Do\_fetch
   2. Do\_unpack
   3. Do\_patch
   4. Do\_configure
   5. Do-install
   6. Do\_compile
   7. Do\_rootfs
   8. Syntex:
      1. do\_install() {

Action\_1;

Action\_2;

………

…….

}

1. Explame:
   1. do\_compile () {

oe\_runmake

}

1. Do\_install () {

Install -d ${D}${bindir} // **D : the destination dir , where files are copied**

}

**Example to write a recipe:**

Hello.bb

**Machine conf**

1. Define the basic machine conf and also define the qemu conf (if required)
   1. u-boot conf
   2. Device tree
   3. Kernel image
   4. Arch include file (if any)

**Apply Patches:**

1. File : .patch and **apply=yes** ,then only patches will apply to source code
2. **do\_patch** task
3. SRC\_URI +="file://patch\_1.patch"
4. Patches are apply in order they are provide

Devtool:

1. The devtool command-line tool provides a number of features that help you build, test, and package software. This command is available alongside the bitbake command. Additionally, the devtool command is a key part of the extensible SDK
2. devtool has more functionality than simply adding a new recipe and the supporting Metadata to a temporary workspace layer
3. Command
   1. Devtool add recipe-name
      1. Devtool add stock-kernel
4. It create a recipe stock-kerel.bb file under workspace/recipes/stock-kernel/stock-kernel.bb
5. We can edit this file too
6. To run recipe
   1. Bitbake stock-kernel

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

1. Init the bitbake env

source oe-init-build-env

1. Crete a layer

bitbake-layers create-layer ../meta-mylayer

1. Inform bitbake about your layer
   1. bitbake-layers add-layer ../meta-mylayer
2. Layer.conf
   1. Contains the layers conf file (reciepe\_kernel, recipe-u-boot) and their .bb files and append files
3. Create the required recipe in meta\_mylayer
   1. Recipe-kernel
   2. Recipe-u-boot
   3. Recipe-core
   4. Recipe-kernel/linux

**IMAGE\_FEATURES** is made to enable special features for your image, such as empty password for root, debug image, special packages, x11, splash, ssh-server...

IMAGE\_INSTALL : to add any packages

IMAGE\_INSTALL\_remove += "chromium" : to remove any packages

**// To passs any file**

**FILESEXTRAPATHS\_prepend** := "path\_to\_home\_folder\_of\_source\_folders:"  
**SRC\_URI** = "[file://Source\_floder/\*](file:///\\Source_floder\*)"

**Demo:**

1. Create a layer
   1. bitbake-layers create-layer ../meta-mylayer
   2. bitbake-layers add-layer ../meta-mylayer
   3. There are only one recipe in meta-mylayer called example
   4. Run bitbake example (for building a recipe example)
      1. To check the example recipe is build or not
         1. /mnt/work/Yocto/openbmc/build/tmp/work/armv7vet2hf-neon-openbmc-linux-gnueabi/test
      2. To generate a image (core-image-minimal or any other images that conatin some of your packages)
      3. Add the image folder in recipe-test dir
      4. Add a recipe with test-images.bb
         1. See the example (poky/meta/recipe-core/images/core-image-minimal-dev.bb)
      5. Now add the description and IMAGE\_INSTALL variable to add any packages (exp bc [calcultor], vim (test editer) or any other )
      6. **Error** : core-image-minimal.bb I not parsed
         1. The add the fill path for this recipe from poky/meta/recipes-core/images/core-images-minimal.bb
      7. Then generate an image with corresponding .bb name
         1. **bitbake test\_images** 
            1. It will generate a image for test\_images containing the bc or vim package.
         2. Then check the test-images is generated in tmp/deploy/images/machine\_name/test-image\*
         3. Load that image on qemu and test it
            1. **runqemu qemuarm nographics**

Check the packages that are giving while compiling the images

1. **Linux kernel**
   1. Yocto wiki
      1. <https://wiki.yoctoproject.org/wiki/How_do_I>

1. Preference of linux yocto will be provide through with version (poky/meta/recipe-kernel/linux/)
   1. In local conf

 PREFERRED\_PROVIDER\_virtual/kernel = "linux-yocto"

 PREFERRED\_VERSION\_linux-yocto = "4.14"

1. Compile again and bitbake core-image-minimal
2. To create a new machine
   1. Go to your recipe (meta-aspeed2600)/conf/machine
   2. Create your own machine conf
      1. Define the qemu conf also and define own machine info too
   3. Add this machine to local.conf (aspeed)
3. Create a recipe-kernel in your recipe (meta-aspeed)
   1. Copy the linux conf from meta-skeleton/recipes-kernel/linux (exp recipe for reference)
   2. Linux-yocto-custom.bb
      1. Add your machine (locally created option:b) in
         1. COMPATIBLE\_MACHINE = "aspeed"
         2. SRCREV\_machine to SRCREV\_aspeed
         3. Edit the path for linux kernel
         4. SRCREV\_aspeed : commit it (git show HEAD)

1. SRC\_URI\_append = " [file://ipresolve.cfg](file:///\\ipresolve.cfg)" // to append the source

**Build your machine or add new machine:**

1. Create a layer
   1. bitbake-layers create-layer ../meta-mylayer
   2. bitbake-layers add-layer ../meta-mylayer
2. Define machine conf with your machine (demoarm.cfg)
   1. Took a help from qemuarm.conf file (poky/meta/conf/machine/qemuarm.cfg)
3. Now run bitbake core-image-minimal to build a image
   1. **EXTRA\_IMAGEDEPENDS :** A list of recipes to build that do not provide packages for installing into the root filesystem. Sometimes a recipe is required to build the final image but is not needed in the root filesystem. You can use the EXTRA\_IMAGEDEPENDS variable to list these recipes and thus, specify the dependencies. A typical example is a required bootloader in a machine configuration.

* 1. **CORE\_IMAGE\_EXTRA\_INSTALL** :  is a convenience variable that enable you to add extra packages to an image based on the core-image class
     1. Foe example, you want add vim package to be a part of image then set this variable
        1. CORE\_IMAGE\_EXTRA\_INSTALL = "vim"
  2. tmp/image/deplay/machine\_name/core\_image\_minimal\*.manifest
     1. It tell what are packages are part of your image
  3. **IMAGE\_FEATURE**S is made to enable special features for your image, such as empty password for root, debug image, special packages, x11, splash, ssh-server...
  4. **IMAGE\_INSTALL** : to add any packages
  5. **IMAGE\_INSTALL\_remove** += "chromium" : to remove any packages

// To passs any file

* 1. **FILESEXTRAPATHS\_prepend** := "path\_to\_home\_folder\_of\_source\_folders:"  
     SRC\_URI = "[**file://Source\_floder/\***](file:///\\Source_floder\*)"
  2. **IMAGE\_CLASSES:** A list of classes that all images should inherit. You typically use this variable to specify the list of classes that register the different types of images the OpenEmbedded build system creates.The default value for IMAGE\_CLASSES is image\_types. You can set this variable in your local.conf or in a distribution configuration file.

For more information, see meta/classes/image\_types.bbclass in the [Source Directory](http://www.yoctoproject.org/docs/1.5/dev-manual/dev-manual.html#source-directory).

IMAGE\_CLASSES += "qemuboot" // basically it generate the qemu.conf filr in tmp/deploy/machine/\*.qemuconf.conf

* 1. **LINUX\_VERSION\_EXTENSION** : it is used to varify the kernel you had compiled after load the kernel on target (uname -a)

LINUX\_VERSION\_EXTENSION ?= "-aspeed-${LINUX\_KERNEL\_TYPE}"