arm

Embedded Linux

Configuration & Build Process of an Embedded Linux System

Goal

To illustrate the configuration & build process of an embedded Linux system

To illustrate the concept of build systems

Summary

Introduction

The workflow

The build systems

Yocto

Summary

Introduction

The workflow

The build systems

Yocto

Introduction

An embedded Linux system requires the following components to operate:

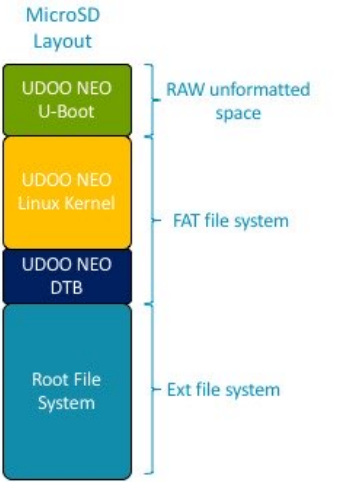
- The bootloader
- The Linux Kernel
- The device tree blob
- The Root File System

All these components shall be:

- Configured for the embedded system hardware platform
- Compiled and linked into an executable format
- Deployed into the embedded system persistent storage for booting and operations

Example:

- UDOO NEO-tailored U-Boot, Linux Kernel, DTB, and Root File System deployed into the MicroSD



arm

Summary

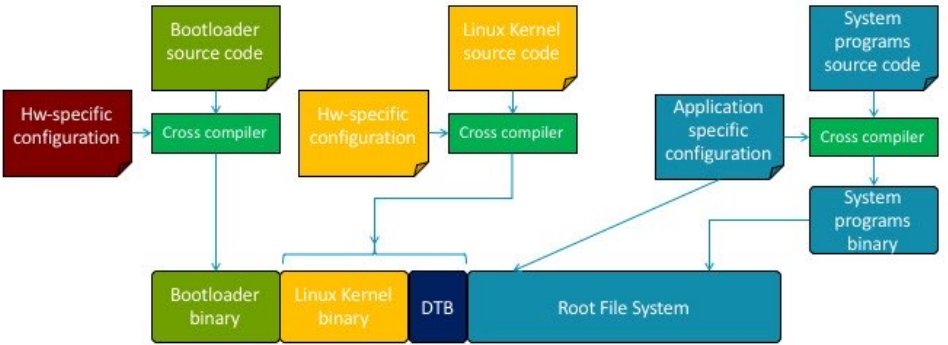
Introduction

The workflow

The build systems

Yocto

The Workflow



arm

The Workflow

The Bootloader source code shall be procured.

It shall be configured for the specific hw of the embedded system.

(example)

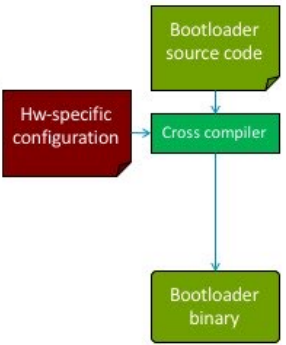
- The proper CPU shall be selected.
- The proper board shall be selected.
- Custom configuration may be needed if the hw is non-standard.

It shall be cross-compiled for the CPU of choice obtaining the executable code.

It shall be copied into the boot device.

(example)

- First sector of a MicroSD card
- Bootflash device on the embedded system board



arm

Linux kernel binary AU device tree blob are prepare in similar way to the bootloader binary it's begin with obtaining kernel source code and it can config for specific hardware of the embedded system

The Workflow

The Linux Kernel source code shall be procured.

It shall be configured for the specific hw of the embedded system.

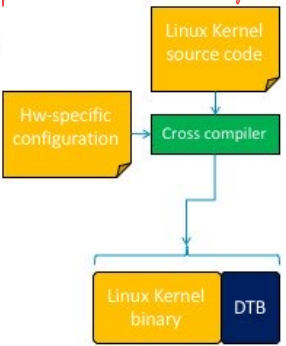
- The proper CPU shall be selected.
- The proper board shall be selected.
- Custom configuration may be needed if the hw is non-standard.

It shall be cross-compiled for obtaining

- Executable Linux kernel
- Executable Linux kernel modules
- Device Tree Blob

The obtained files shall be copied into the boot device.

- Partition on a MicroSD device
- Bootflash device on the embedded system board



arm

The Workflow

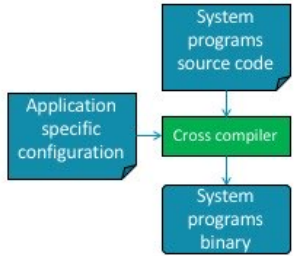
The system programs source code shall be procured.

- Downloaded from Internet (sourc)
- Received from operating system vendor

They shall be configured for the specific application.

- Only the system programs needed for the application the embedded system is intended for shall be considered.

They shall be cross-compiled obtaining the executable binary.



arm

The Workflow

The root file system shall be prepared.

It typically requires

- To create a file and mount it as a volume on the development host
- To format it using any of the file systems Linux supports (e.g., ext3)
- To create the required directory tree
- To populate it with the needed configuration files
- To populate it with the system program binary

The root file system shall be copied into the embedded system persistent storage.

- Partition on a MicroSD device

or

- Bootflash device on the embedded system board

```
/                                          # Disk root
  /bin                  # Repository for binary files
  /lib                  # Repository for library files
  /dev                  # Repository for device files
    console c 5 1       # Console device file
    null c 1 3          # Null device file
    zero c 1 5          # All-zero device file
    tty c 5 0           # Serial console device file
    tty0 c 4 0          # Serial terminal device file
    tty1 c 4 1          #
    tty2 c 4 2          #
    tty3 c 4 3          #
    tty4 c 4 4          #
    tty5 c 4 5          #
  /etc                  # Repository for config files
    inittab             # The inittab
    /init.d             # Repository for init config files
      rcS               # The script run at sysinit
  /proc                 # The /proc file system
  /sbin                 # Repository for accessory binary files
  /tmp                  # Repository for temporary files
  /var                  # Repository for optional config files
  /usr                  # Repository for user files
  /sys                  # Repository for system service files
  /media                # Mount point for removable storage
```

Summary

Introduction

The workflow

The build systems

Yocto

arm

Build Systems

Building an embedded Linux system is a complex operation.

- Multiple sources shall be configured and compiled. which different depmodency
- Root file system shall be updated at each build through a non-trivial task.
- In cases of multiple hw and multiple hw configurations, manual iteration is needed.

Tools, known as build systems, are available to automate such operations.

Build systems takes care of:

- Building the cross compiler for the selected embedded system CPU
- Managing bootloader/kernel/system programs configuration
- Managing bootloader/kernel/system program build
- Preparation of the root file system and boot device image preparation

Build Systems

Several solutions are available.

Hw-vendor custom-built systems

- NXP Linux Target Image Builder (LTIB)

Open-source build systems, among which the most popular are Yocto and Buildroot

- Very actively maintained and developed projects
- Widely used in the industry
- Built from scratch from source toolchain, bootloaders, kernel, and root file system

arm

Buildroot vs Yocto: General Aspects

Buildroot

- Focus on simplicity
- Use existing technologies: kconfig, make
- Open community

Yocto (more complex)

- Provides core recipes and use layers to get support for more packages and more machines
- Custom modifications should stay in a separate layer
- Versatile build system: tries to be as flexible as possible and to handle most use cases
- Open community but governed by the Yocto Project Advisory Board

Buildroot vs Yocto: Configuration

Buildroot reuses kconfig from the Linux kernel

- Entire configuration stored in a single `.config/defconfig`
- Defines all aspects of the system: architecture, kernel version/config, bootloaders, user-space packages, etc.
- Building the same system for different machines to be handled separately

In Yocto the configuration is separated in multiple parts:

- Distribution configuration (general configuration, toolchain selection, etc.)
- Machine configuration (defines the hw architecture, hw features, BSP) ← board support package
- Image recipe (what system programs should be installed on the target)
- Local configuration (e.g., how many threads to use when compiling, whether to remove build artifacts, etc.)
- Allows to build the same image for different machines or using different distributions or different images for one machine

↳ เห็นว่ามันทำให้ Yocto powerful

arm

Buildroot vs Yocto: Purpose

Buildroot is intended for

- Very small root file systems (< 8 MB)
- Simple embedded system (with limited number of system programs)
- Non-dedicated build engineers (e.g., engineers that are not focused only in building embedded Linux)

Yocto is intended for

- Large root file systems
- Large embedded systems
- Support for multiple hw configurations
- Dedicated build engineers

yocto PROJECT

Summary

Introduction

The workflow

The build systems

Yocto

The Yocto Project

Open-source project hosted by the Linux Foundation

Collaboration of multiple projects that make up the “Yocto Project”

- Bitbake: build tool
- OpenEmbedded core: software framework used for creating Linux distributions
- Poky: a reference distribution of the Yocto Project, containing the OpenEmbedded Build System (BitBake and OpenEmbedded Core) and a set of metadata to start building custom embedded Linux systems
- Application Development Toolkit: provides application developer a way to write sw running on the custom-built embedded Linux system without the need for knowing build systems

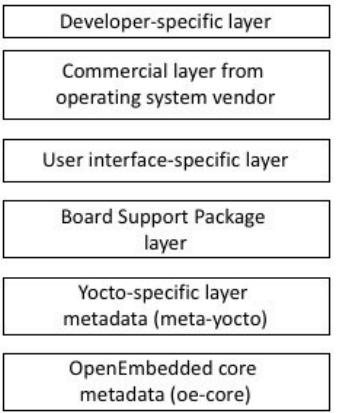
Support for Arm, PPC, MIPS, and x86

The Yocto Build System

It is composed of multiple layers which are containers for the building blocks of the system.

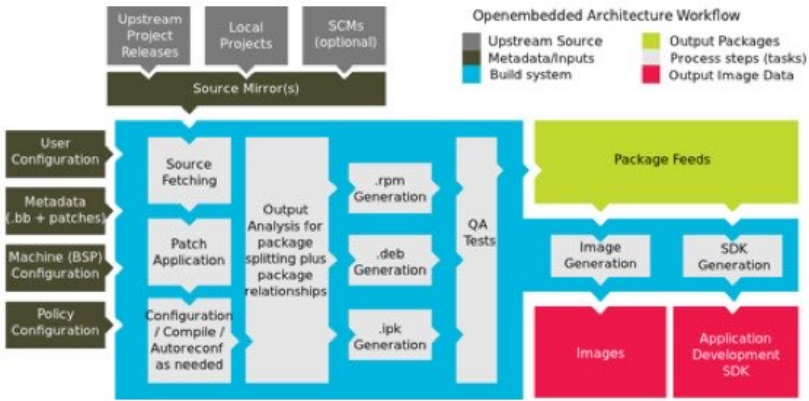
Layers do not contain components source code, only their metadata, called recipes.

Recipes define how to build binary outputs called packages.



arm

The Build System Workflow



Source: http://www.yoctoproject.org/docs/2.1/mega-manual/mega-manual.html

arm

The Build System Workflow: Configuration Files

Yocto has a host of different configuration file for different proposes

`meta/conf/bitbake.conf` – default configuration

`build/conf/bblayers.conf` – layers to be used during build process

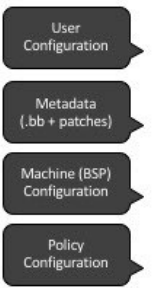
`*/conf/layers.conf` – layer configuration

`build/conf/local.conf` – user-define configuration

`meta-yocto/conf/distro/poky.conf` – distribution policy

`meta-yocto-bsp/conf/machine/board-name.conf` – configuration of the board support pacakge

`meta/conf/machine/include/tune-CPU-name.inc` – CPU-specific configuration



arm

The Build System Workflow: User Configuration

`build/conf/local.conf` is used to override the default configuration and define what to build.

- `BB_NUMBER_THREADS` and `PARALLEL_MAKE`
- `MACHINE` settings
- `DISTRO` settings
- `INCOMPATIBLE_LICENSE = "GPLv3"`
- `EXTRA_IMAGE_FEATURES`

`build/conf/bblayers.conf` is used to configure which layers to use.

- Add Yocto Project Compatible layers to the `BBLAYERS`
- Default: meta (oe-core), meta-yocto, and meta-yocto-bsp

Example for the UDOO-NEO

local.conf (fragment)

```
MACHINE ??= 'udooneo'
DISTRO ?= 'poky'
```

bblayer.conf (fragment)

```
BBLAYERS = " \
  ${BSPDIR}/sources/poky/meta \
  ${BSPDIR}/sources/poky/meta-yocto \
  ${BSPDIR}/sources/poky/meta-yocto-bsp \
  \
  ${BSPDIR}/sources/meta-openembedded/meta-oe \
  ${BSPDIR}/sources/meta-openembedded/meta-multimedia \
  ${BSPDIR}/sources/meta-openembedded/meta-networking \
  ${BSPDIR}/sources/meta-openembedded/meta-python \
  \
  ${BSPDIR}/sources/meta-fsl-arm \
  ${BSPDIR}/sources/meta-fsl-arm-extra \
  ${BSPDIR}/sources/meta-fsl-demos \
  ${BSPDIR}/sources/meta-udoo \
```


arm

The Build System Workflow: Metadata

Recipes for building packages

Recipes inherit the system configuration and adjust it to describe how to build and package the software.

Can be extended and enhanced via layers

Compatible with OpenEmbedded

Example for the UDOO-NEO

udoo-image-full-cmdline.bb (fragment)

```
DESCRIPTION = "A console-only image with \
more full-featured Linux system \
functionality installed. \
Tailored for the UDOO boards"

IMAGE_FEATURES += "splash ssh-server-openssh \
package-management"

UDOO_EXTRA_INSTALL = " \
    resize-rootfs \
    screen \
    imx-gpu-viv \
    imx-gpu-viv-demos \
    packagegroup-fsl-tools-gpu \
    binutils \
    minicom \
    i2c-tools \
"
```


arm

The Build System Workflow: Machine (BSP) Configuration

Configuration files that describe a machine

- Define board specific kernel configuration
- Processor/SOC Tuning files

Machine configuration refers to kernel sources.

Compatible with OpenEmbedded

Example for the UDOO-NEO

udooneo.conf (fragment)

```
#@TYPE: Machine
#@NAME: UDOO Neo i.MX6 SoloX
#@SOC: i.MX6S
#@DESCRIPTION: Machine configuration for i.MX6 UDOO \
Neo SoloX
#@MAINTAINER: Christian Ege <ch@ege.io>

include conf/machine/include/imx-base.inc
include conf/machine/include/tune-cortexa9.inc

SOC_FAMILY = "mx6:mx6sx"

PREFERRED_PROVIDER_virtual/kernel ?= "linux-udooboard"
PREFERRED_VERSION_linux-udooboard ?= "3.14%"
PREFERRED_PROVIDER_u-boot ?= "u-boot-udooboard"
KERNEL_IMAGETYPE = "zImage"
```


arm

The Build System Workflow: Distribution Policy

Defines distribution policies that affect the way individual recipes are built

- May set alternative preferred versions of recipes
- May enable/disable features
- May configure specific package rules
- May adjust image deployment settings

Enabled via the DISTRO setting

Four predefined settings:

- poky-bleeding: enables bleeding edge packages
- poky: core distribution definition, defines the base
- poky-lsb: enable items required for LSB support
- poky-tiny: construct a smaller than normal system

arm

The Build System Workflow: Source Fetching

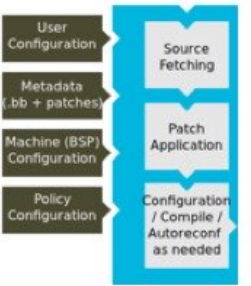
Recipes tell the location of all sources, patches, and files.

- These may exist on the internet or be local (See `SRC_URI` in the `*.bb` files).

Bitbake can get the sources from git, svn, bzr, or tarballs.

Versions of packages can be fixed or updated automatically (Add `SRCREV_pn-PN = "${AUTOREV}"` to `local.conf`).

The Yocto Project mirrors sources to ensure source reliability.



arm

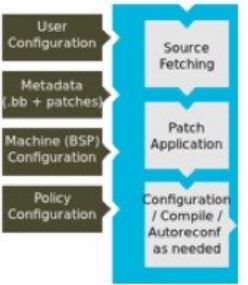
The Build System Workflow: Patching

Once sources are obtained, they are extracted.

Patches are applied in the order they appear in `SRC_URI`.

In this stage, application-specific patches are applied.

- Custom modifications to the open-source code that are mandated by the specific application for which the embedded system is intended for

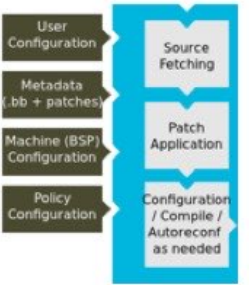


arm

The Build System Workflow: Configure/Compile/Install

The recipe specifies configuration and compilation rules.

- Various standard build rules are available, such as autotools and gettext.
- Standard ways to specify custom environment flags
- Install step runs under “pseudo”, allows special files, permissions, and owners/ groups to be set.



arm

The Build System Workflow: Output Analysis/Packaging

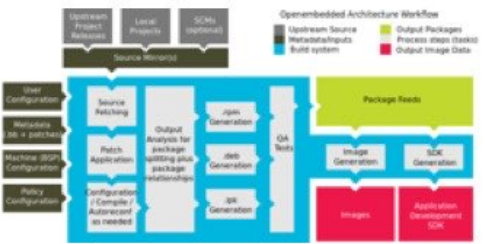
Output Analysis

- Categorize generated software (debug, dev, docs, and locales)
- Split runtime and debug information

Perform QA tests (sanity checks)

Package Generation

- Support the popular formats: RPM, Debian, and ipk
- Set preferred format using `PACKAGE_CLASSES` in `local.conf`
- Package files can be manually defined to override automatic settings.



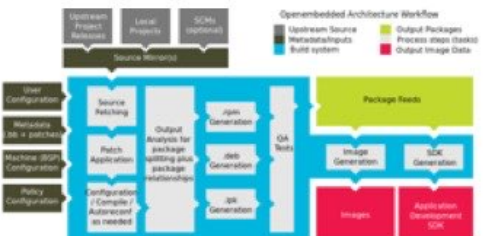
arm

The Build System Workflow: Image Generation

Images are constructed using the packages built earlier and put into the Package Feeds.

Decisions of what to install on the image is based on the minimum defined set of required components in an image recipe. This minimum set is then expanded based on dependencies to produce a package solution.

Images may be generated in a variety of formats (tar.bz2, ext2, ext3, jffs, etc.).



arm

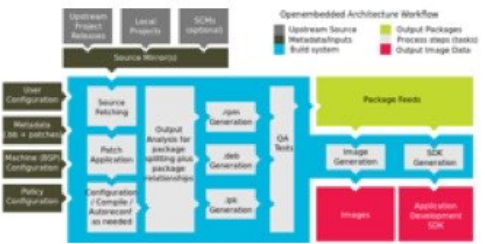
The Build System Workflow: SDK Generation

A specific SDK recipe may be created .

SDK may be based on the contents of the image generation.

SDK contains native applications, cross toolchain, and installation scripts.

May be used by the Eclipse Application Developer Tool to enable App Developers



arm