YOCTO

Yocto Introduction

Yocto is a open source collaboration project, which helps the developers to create their own custom linux distribution system, irrespective of Hardware.

Why to use Yocto?

- To configure the Linux according to our specs.
- The Image contains what we need.
- No extra packages.
- Small Image Size.
-

What is the Work-flow of Yocto ?

- The Begin ,Developers to specify Architecture,Policies,Patches and Configuration Details.
- The build system then fetches and Downloads the source code from where Ever specifie. The Project support Source Code. After the Extract into a Local Work Area Where Patches Are Applied and Common steps for configuring and Compiling the software will be Run.
- The Binaries Are Created A Binary Package feed is Generated Which is then used to Create the final root file image.
- The file System image is Generated.

Flashing Board

How to Flash ?

- Download Etcher
- Go to the Download folder .
- Double Click on Etcher a Window will be open.
- Insert the SD Card to Host PC.
- Select SD Card.
- Click Flash from file and Browse the Image File that we found Earlier
- Unmount the Safely Remove SD Card.
- Open the minicom → sudo apt-get install minicom

Booting Process ?

- Insert the SD Card in Board is SD-Card Jack .
- There is small button with the label S2 on it.
- Button Pressed on the Board the Power supply in Board.
- Show the Output has the output on your minicom Terminal.

What is Poky?

Poky is a reference distribution of the Yocto Project. It contains the OpenEmbedded Build System (BitBake and OpenEmbedded Core) as well as a set of metadata to get you started building your own distro.

Metadata and Recipes

Poky includes a collection of metadata and recipes that define how packages are built and configured. This metadata specifies dependencies, build instructions, patches, and configuration options for each package.

BitBake

BitBake is a task executor and scheduler designed specifically for building embedded Linux distributions. It reads recipes and executes tasks to build packages and images.

local.conf

What is Local.conf ?

The Local.conf file is Used to Customize the Images and Add Packages.

Default Configurations in local.conf file

- MACHINE ?= "beaglebone-yocto"
- DL_DIR, SSTATE_DIR, TMPDIR
- DISTRO ?= "poky"
- PACKAGE_CLASSES ?= "package_rpm"
- EXTRA_IMAGE_FEATURES ?= "debug-tweaks"
- USER_CLASSES ?= "buildstats"
- PATCHRESOLVE = "noop"
- BB_DISKMON_DIRS
- CONF_VERSION = "2"

Added Configurations

- RM_OLD_IMAGE = "1"
- INHERIT += "rm_work"

Configuration Details

MACHINE

Specifies the target device for which the image is built.

#List of default Machines

MACHINE ?= "qemuarm" MACHINE ?= "qemuarm64" MACHINE ?= "qemumips" MACHINE ?= "qemumips64" MACHINE ?= "qemuppc" MACHINE ?= "qemux86" MACHINE ?= "qemux86-64" MACHINE ?= "genericx86" MACHINE ?= "genericx86-64" MACHINE ?= "beaglebone" MACHINE ?= "edgerouter"

default Machine Conf File locations.

poky/meta/conf/machines.

DL_DIR

DL_DIR is a variable in the Yocto Project build system that specifies the directory where source code archives for packages will be downloaded. This directory is used by the build system to store the downloaded source code for packages so that it does not need to download them again if they are required for a subsequent build.

SSTATE_DIR

SSTATE_DIR is a variable that specifies the directory where shared state cache files are stored. The shared state cache contains pre-built binary packages for software components used in a build.

The purpose of using a shared state cache is to speed up the build process by avoiding the need to rebuild packages that have already been built previously. When a package is built for the first time, the build system stores the compiled binaries, headers, and other artifacts in the shared state cache, and subsequent builds of the same package can reuse these cached artifacts, saving time and resources.

TMPDIR

TMPDIR is an environment variable that specifies the directory to be used for temporary files by various programs and scripts. In Yocto and OpenEmbedded, TMPDIR is used as the location for the build directory, where all the build artifacts are stored during the build process.

DISTRO

DISTRO ?= "poky"

- DISTRO variable specifies the name of the distribution that is being built.
- A distribution is a collection of software components and configuration files that work together to create a complete Linux-based operating system
- Setting DISTRO to "poky" means that the build system will use the configuration files and package recipes that are part of the reference distribution.

PACKAGE_CLASSES

PACKAGE_CLASSES ?= "package_rpm"

PACKAGE_CLASSES?=”package_deb”

- PACKAGE_CLASSES is a configuration variable specifies the types of packages to be created for the target system.
- In this case, it is set to "package_rpm", which means that the Yocto Project will create RPM packages.
- RPM is a package management system that is widely used in many Linux distributions, including Red Hat, Fedora, CentOS, and openSUSE.
- DEB package management is used on Debian and Ubuntu systems.

EXTRA_IMAGE_FEATURES

EXTRA_IMAGE_FEATURES ?= "debug-tweaks"

- EXTRA_IMAGE_FEATURES is a variable that specifies additional features to be included in the target image
- debug-tweaks is one of the features that can be included in the image.
- The resulting image will contain additional tools and utilities that can help with debugging and troubleshooting.
- dbg-pkgs — adds -dbg packages for all installed packages including symbol information for debugging and profiling.
- debug-tweaks — makes an image suitable for debugging. For example, allows root logins without passwords and enables post-installation logging. See the ‘allow-empty-password’ and ‘post-install-logging’ features in the “Image Features” section for more information.
- dev-pkgs — adds -dev packages for all installed packages. This is useful if you want to develop against the libraries in the image.
- read-only-rootfs — creates an image whose root filesystem is read-only. See the “Creating a Read-Only Root Filesystem”** section in the Yocto Project Development Tasks Manual for more information
- tools-debug — adds debugging tools such as gdb and strace.
- tools-sdk — adds development tools such as gcc, make, pkgconfig and so forth.

USER_CLASSES

USER_CLASSES ?= "buildstats"

- The buildstats class records performance statistics about each task executed during the build (e.g. elapsed time, CPU usage, and I/O usage).
- The buildstats class generates statistics about how long tasks take to build, so it can help identify bottlenecks in the build process. It creates a buildstats database that can be used to analyze the build process.
- When you use this class, the output goes into the BUILDSTATS_BASE directory, which defaults to ${TMPDIR}/buildstats/.

PATCHRESOLVE

PATCHRESOLVE = "noop"

- Determines the action to take when a patch fails. You can set this variable to one of two values: noop and user.
- The default value of noop causes the build to simply fail when the build system cannot successfully apply a patch.
- Setting the value to user causes the build system to launch a shell and places you in the right location so that you can manually resolve the conflicts.

BB_DISKMON_DIRS

- Monitors disk space and available inodes during the build and allows you to control the build based on these parameters.
- STOPTASKS,${TMPDIR},1G,100K stops the build after all currently executing tasks complete when the minimum disk space in the `${TMPDIR}` directory drops below 1 Gbyte.
- HALT,${TMPDIR},100M,1K immediately stops the build when the disk space in the `${TMPDIR}` directory drops below 100 Mbytes.

CONF_VERSION

CONF_VERSION = "2"

- CONF_VERSION is used to specify the version of the configuration syntax to use.
- It determines which syntax the build system should use when parsing configuration files like local.conf and bblayers.conf.
- In earlier versions of Yocto, CONF_VERSION defaulted to 1
- In practice, setting CONF_VERSION = "2" in `local.conf` enables the use of newer features and syntax in Yocto

bblayers.conf

What is bblayers.conf?

- bblayers.conf is a configuration file used by the build systems
- In this file the set of layers are defined that should be included in a build.
- The bblayers.conf file specifies the location of each layer on the local file system.

Syntax: Each layer path is specified as an absolute or relative path to the layer's directory. The syntax of the `bblayers.conf` file is simple, typically consisting of lines like:

```
BBLAYERS ?= " \

  /path/to/yocto/poky/meta \

  /path/to/yocto/poky/meta-yocto \

  /path/to/yocto/poky/meta-yocto-bsp \

"
```

Layer in Yocto

What is the Layer :

A layer is a collection of related metadata that provides configuration information, such as recipes, configuration files, and other data required to build and customize an image.

Default Layers:

- meta
- meta-poky
- meta-yocto-bsp

Bitbake basic layers commands

To Create layer from bblayers.conf

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

To Add layer to bblayers.conf

bitbake-layers add-layer ../source/meta-mylayer

To Show layers in bblayers.conf

bitbake-layers show-layers

To Remove layer from bblayers.conf

bitbake-layers remove-layer <path/to/layer>

Add Packages

How to Add Package?

Add the following line in local.conf file.

IMAGE_INSTALL:append = " package_name"

How to Find Packages?

bitbake-layers show-recipes package_name

The above command will show the path of the package

If the layer that contains the required package is not present in bblayers.conf the add by using below command

bitbake-layers add-layer <path/to/layer>

What is layer priority?

Establishes a priority to use for recipes in the layer when the OpenEmbedded build finds recipes of the same name in different layers. A higher numeric value represents a higher priority.

To create layers with our required priority

bitbake-layers create-layer -p 7 <path/to/layer>

How to check priority

bitbake-layers show-layers

Basic Variables

Basic Variables in yocto

- PN (Package Name)
- PV (Package Version)
- PR (Package Revision)
- WORKDIR (Working Directory)
- S (Source)
- D (Destination)
- B (Build Directory)

Recipe Name Pattern

PN_PV_PR.bb

Example: example2_0.2_r0

How to Read Variable Value

bitbake -e <RECIPE_NAME> | grep ^<VARIABLE_NAME>=

Package Name (PN)

PN refers to a recipe name used by the Yocto build system as input to create a package. The name is extracted from the recipe file name.

Package Version (PV)

PV is the version of the recipe. The version is normally extracted from the recipe filename.

Package Revision (PR)

The revision of the recipe. The default value for this variable is “r0”

Working Directory (WORKDIR)

The WORKDIR is the pathname of the work directory in which the Yocto build system builds a recipe. This directory is located within the TMPDIR directory structure and is specific to the recipe being built and the system for which it is being built.

Source (S)

S is the location in the Build Directory where unpacked recipe source code resides. By default, this directory is WORKDIR/BPN-PV, where BPN is the base recipe name and PV is the recipe version.

Destination (D)

D is the destination directory. It is the location in the Build Directory where components are installed by the do_install task. This location defaults to WORKDIR/image.

Build Directory (B)

It is same as S.

Variable Assignment

Types of Variable Assignments

- ?= : This is used to assign the default value to variable. It can be overridden.
- ??= : This is used to assign the default value to variable. But it is a weak assignment. It can be overridden. If multiple assignments are done with this type, the the last one will be considered.
- = : This is a simple variable assignment. It requires " " and spaces are significant. But variables are expanded at the end.
- := : This is an immediate variable expansion. The value assigned is expanded immediately.
- += : This appends a value to a variable. The operator inserts a space between the current value and appended value. It takes effect immediately.
- =+ : This prepends a value to a variable, The operator inserts a space between the current value and prepended value. It takes effect immediately.
- .= : This appends a value to a variable. The operator inserts no space between the current value and appended value. It takes effect immediately.

- =. : This prepends a value to a variable. The operator inserts no space between the current value and prepended value. It takes effect immediately.
- :append : This appends a value to a variable. The operator inserts no space between the current value and appended value. The effects are applied at variable expansion time rather than being immediately applied.
- :prepend : This appends a value to a variable. The operator inserts no space between the current value and appended value. The effects are applied at variable expansion time rather than being immediately applied.
- :remove : This remove values from lists. Specifying a value for removal causes all occurrences of that value to be removed from the variable.

How to Read Variable Value

bitbake -e <RECIPE_NAME> | grep ^<VARIABLE_NAME>=

Assignment Type ?=

TEST ?= “foo”

TEST ?= “ball”

TEST ?= “val”

```
TEST ?= "var"
        # The final value is TEST="foo"
```

Assignment Type ??=

```
TEST ??= "foo"

TEST ??= "bar"

TEST ??= "val"

TEST ??= "var"

         # The final value is TEST="var"

TEST ??= "foo"

TEST ?= "bar"

TEST ?= "val"

TEST ??= "var"

         # The final value is TEST="bar"
```

Assignment Type =

```
# Override

A ?= "foo"

A = "bar"

        # The final value is A="bar"

 # Variable Expansion

A = "foo"

B = "${A}"

A = "bar"

        # The final value is B="bar"
```

Assignment Type :=

```
# Override

A ?= "foo"

A := "bar"

        # The final value is A="bar"
```

Assignment Type += and =+

```
# Prepend

B = "foo"

B =+ "bar"

        # The final value is B="bar foo"
```

```
# Append

A ?= "val"

A += "var"

        # The final value is A="var"
```

Assignment Type .= and =.

```
# Append

A = "foo"

A .= "bar"

        # The final value is A="foobar"

# Prepend

B = "foo"

B =. "Bar"

        # The final value is B="barfoo"
```

Assignment Type :append, :prepend and :remove

```
# Append

A = "foo"

A:append = "bar"

    # The final value is A="foobar"
# Prepend

A = "foo"

A:prepend = "bar"

    # The final value is A="barfoo"
#remove

A = "foo bar"

    A:remove = "foo"
    # The final value is A=" bar"
```

What is the Recipe

Recipes are fundamental components in the Yocto Project environment.

A Yocto/OpenEmbedded recipe is a text file with file extension .bb

Each software component built by the OpenEmbedded build system requires a recipe to define the component

Hello World

A recipe contains information about single piece of software.

- A helloworld.c program
- SUMMARY : A brief description of the Recipe
- LICENSE : Which Type of License are we going to use E.g MIT, GPL, BSD etc.
- LIC_FILES_CHKSUM : License file location and its md5 checksum.
- Calculate checksum using md5sum utility
- SRC_URI : Source Files
- do_compile: Here the compilation takes place.
- do_install : Here we tells the recipe where to put the binary file in final image.

How to Generate md5 Checksum

```
md5sum FILENAME
```

Build Tasks

Tasks can be considered as units of execution to perform a specific function, or a set of related functions that can be combined together.

How to list the build tasks of a recipe?

bitbake -c listtasks <recipe-name>

What are common build tasks in Yocto?

- Fetch (do_fetch) : Fetches the source code
- Unpack (do_upack) : Unpacks the source code into a working directory
- Patch (do_patch) : Locates patch files and applies them to the source code
- Configure (do_configure) : Configures the source by enabling and disabling any build-time and configuration options for the software being built.
- Compile (do_compile) : Compiles the source in the compilation directory
- Install (do_install) : Copies files from the compilation directory to a holding area

Fetch Task

Fetch task fetches the package source from the local or remote repository.

The fetch Repo address has to be stored in SRC_URI variable.

In SRCREV Variable the commit hash of github repo is defined.

The source repo is stored in SRC_URI variable. Normally the build process fetches the source automatically.

There is no explicit need to execute the fetch task.

do_fetch command:

bitbake -c do_fetch recipe-name

Unpack Task

Unpack task unpacks the package that has been downloaded with Fetch task.

Normally the build process unpacks the source automatically.

There is no explicit need to execute the unpack task.

The unpack task unpacks the sources in to WORKDIR folder.

do_unpack command:

bitbake -c do_unpack recipe-name

Patch Task

Patch task locates the patch files and applies the patches to the sources if any patch is available.

A patch files provides some explicit changes for a specific file. That can be applied to that file.

Patch file is also defined in SRC_URI variable.

By default it runs in current source directory ${S}.

The patches are stored in SRC_URI variable.

There is no explicit need to execute the patch task.

do_patch command:

bitbake -c do_patch recipe-name

Configure Task

The Configuration task configures the source by enabling and disabling any build-time and configuration options for the software being built before compilation if any configuration is available.

If there are any configuration steps, then these steps are define in do_configure() function of bitbake

Normally the build applies the defined configuration automatically.

There is no explicit need to execute the configuration task.

```
do_configure()

{

configuration steps

}
```

do_configure Command;

```
bitbake -c do_configure recipie-name
```

Compile Task

The Compilation task compiles the source code if any compilation steps are available and generates a binary file.

It runs in current source directory ${S}.

Normally the build executes the compile step automatically.

There is no explicit need to execute the compilation task.

do_compile()

{

compilation steps

}

do_compile Command:

bitbake -c do_compile recipie-name

Install Task

The Install task copies files that are to be packaged into the holding area ${D}. This task runs with the current working directory ${S} which is the compilation directory.

It runs in current source directory ${S}.

Normaly the build executes the Install task automatically.

There is no explicit need to execute the Install task.

do_install()

{

compilation steps

}

do_install command:

bitbake -c do_install recipie-name

Patch

Patches refer to changes made to the source code of software packages in a Yocto build. These changes are often necessary to adapt the software to the specific requirements or constraints of a target embedded system.

Yocto provides mechanisms to apply patches to the source code of packages during the build process.

Patches are usually stored as separate files within the Yocto project directory structure.

Yocto also provides utilities and conventions for managing patches. This includes features for automatically applying patches, managing dependencies between patches, and specifying the order in which patches should be applied.

patches play a crucial role in customizing and adapting software within a Yocto-based embedded Linux system, allowing developers to tailor software packages to meet the specific requirements of their target devices.

Steps to create and Apply a Patch

- bitbake -c devshell *recipe*
- git init
- git add *
- git commit (sourcetree recorded by git)
- Edit the file in any editor you like and then save it eg.(vi */path/to/file*)
- git status (shows that the file is modified)
- git add */path/to/file*
- git commit -m *"a sutaible comment according to the changes you made"*
- git log (shows that changes have be made and commit history)
- git format-patch HEAD~1 (output the patch file created by the last commit)
- ls (checks if patch file is there)
- Copy the patch file into recipe/files folder
- execute exit to exit devshell
- Edit recipe.bb OR create recipe.bbappend file and add patch file in SRC_URI variable

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"

SRC_URI += " file://patchfile.patch "
```

- Build the image again with bitbake *image_name*

RDEPENDS

In Yocto Project, `RDEPENDS` is a variable used to specify runtime dependencies for a particular package or recipe. It is used to define the other packages or components that must be present on the target system for the package to function correctly during runtime.

The `RDEPENDS` variable is typically defined within the recipe file (.bb or .bbappend) of the package. It contains a space-separated list of package names that the current package depends on for proper runtime execution.

Let's say you have a recipe for a package called "myapp" which is an application that requires the "openssl" library to run. In the recipe file for "myapp" (e.g., `myapp.bb`), you can specify the runtime dependency using the `RDEPENDS` variable like this:

```
RDEPENDS_${PN} = "openssl"
```

In the example above, `${PN}` refers to the package name itself, which in this case is "myapp". So, we are saying that "myapp" has a runtime dependency on "openssl".

You can specify multiple dependencies by separating them with spaces:

```
RDEPENDS_${PN} = "libfoo libbar openssl"
```

In this case, "myapp" has runtime dependencies on "libfoo", "libbar", and "openssl".

The `RDEPENDS` variable helps the package manager in the Yocto Project to automatically include the specified runtime dependencies when generating the root filesystem or image for the target system. This ensures that all the required components are present on the target device for the package to function correctly during runtime.

RPROVIDES

A list of package name aliases that a package also provides. These aliases are useful for satisfying runtime dependencies of other packages both during the build and on the target (as specified by RDEPENDS).

As with all package-controlling variables, you must always use the variable in conjunction with a package name override. Here is an example:

```
RPROVIDES:${PN} = "foobar"
```

SystemD_init_manager

Init Manager

An init manager, or init system, is a fundamental software component in a Unix-like operating system that is responsible for initializing the system during the boot process and managing system processes and services throughout the system's runtime. It is the first program to run when the operating system starts and is assigned the process ID (PID) of 1.

What does Init Manager do?

1. System Boot and Initialization: The init manager is responsible for initializing the system hardware, mounting filesystems, setting up essential system parameters, and starting essential system services. It ensures that the system reaches a functional state.
2. Service Management: It manages system services and daemons. This includes starting, stopping, restarting, and monitoring processes. Services can be system-level processes, user-level processes, or background daemons.
3. Dependency Resolution: An init manager often manages service dependencies, ensuring that services are started in the correct order. For example, a database service may depend on a network service, so the init manager ensures the network service starts before the database service.
4. Process Monitoring and Restart: It monitors running processes and can restart them if they fail or crash. This feature is crucial for maintaining system stability.
5. Shutdown and Reboot: The init manager handles the graceful shutdown or reboot of the system. It stops running services and ensures a clean system shutdown.

Available Init Managers

By default, the Yocto Project uses SysVinit as the initialization manager. There is also support for BusyBox init, a simpler implementation, as well as support for systemd.

What is SystemD Init Manager?

SystemD is a full replacement for init with parallel starting of services, reduced shell overhead, increased security and resource limits for services, and other features that are used by many distributions.

How to integrate SystemD in Yocto?

```
DISTRO_FEATURES:append = " systemd"

VIRTUAL-RUNTIME_init_manager = "systemd"

DISTRO_FEATURES_BACKFILL_CONSIDERED += "sysvinit"

    VIRTUAL-RUNTIME_initscripts = "systemd-compat-units"
```

`DISTRO_FEATURES_append = "systemd"` is used to specify that systemd is among the features your Linux distribution supports. It doesn't directly dictate the init system.

`VIRTUAL-RUNTIME_init_manager = "systemd"` explicitly specifies systemd as the init system for your distribution, ensuring that systemd is used for managing system initialization and services

In practical terms, by setting `DISTRO_FEATURES_BACKFILL_CONSIDERED += "sysvinit"`, you are ensuring that the "sysvinit" feature is included in the consideration process when determining the features that your Linux distribution supports. This can be useful when you want to ensure compatibility or support for software or components that rely on the SysVinit init system or related features.

systemd-compat-units: This specific value for `VIRTUAL-RUNTIME_initscripts` refers to a set of compatibility units designed to work with systemd. These units are used to launch and manage services that rely on SysVinit-style init scripts.

Comparison between Init Managers

In systems with SysVinit or BusyBox init, services load sequentially (i.e. one by one) during init and parallelization is not supported. With systemd, services start in parallel. This method can have an impact on the startup performance of a given service, though systemd will also provide more services by default, therefore increasing the total system boot time. systemd also substantially increases system size because of its multiple components and the extra dependencies it pulls.

On the contrary, BusyBox init is the simplest and the lightest solution and also comes with BusyBox mdev as device manager, a lighter replacement to udev, which SysVinit and systemd both use.

Systemd Service Recipe

A systemd service is a unit of work that can be started, stopped, or managed by systemd. These services can be custom applications, daemons, or system-level tasks. Systemd services are defined by service unit files (usually with a `.service` extension) that contain configuration information for managing the service's behavior.

Why to use SystemD Service?

- Parallelization: Systemd can start and manage services in parallel, reducing boot times.
- Dependency Management: It handles service dependencies, ensuring services start in the correct order.
- Logging: Systemd collects and manages service logs, making it easier to troubleshoot issues.
- Resource Management: Systemd can set resource limits for services, enhancing system stability.
- Service Recovery: It can automatically restart failed services, improving system reliability.
- Standardization: Systemd is widely adopted in modern Linux distributions, providing consistency across systems.

How to write a Recipe for SystemD Service?

- Create a code or script if necessary.
- Create a `.service` unit file that defines the service's behavior.
- Place the `.service` file and code/script files in a location within your recipe (e.g., in the `files` directory).
- In the recipe file (`.bb`), use `do_install` to copy the `.service` and/or code/script files to the appropriate location in the root filesystem.
- Define any dependencies and metadata for your recipe

Where to place Service File in Root Filesystem?

Use the `do_install` task in your Yocto recipe to specify where the `.service` unit file should be placed in the root filesystem. The location typically follows the FHS (Filesystem Hierarchy Standard) and can be something like `${systemd_system_unitdir}/system`.

How to Enable the Service by Default?

```
SYSTEMD_AUTO_ENABLE = "enable"

    SYSTEMD_SERVICE:${PN} = "sysd.service"
```

Some Basic SystemD Commands

- `systemctl status servicename`: Check the status of a systemd service.
- `systemctl start servicename`: Start a systemd service.
- `systemctl stop servicename`: Stop a systemd service.
- `systemctl restart servicename`: Restart a systemd service.
- `systemctl enable servicename`: Enable a service to start at boot.
- `systemctl disable servicename`: Disable a service from starting at boot.
- `systemctl daemon-reload` : Reloads Systemd Daemons.