Kosta Knowledge Transfer

Kosta Zertsekel[1]

April 28, 2019

[1] mailto://zertsekel@gmail.com

Dedicated to good friends and collegues.

Contents

List of Figures

List of Tables

Preface

> *"So if you want to go fast, if you want to get done quickly, if you want your code to be easy to write, make it easy to read."*
>
> – Robert C. Martin, *Clean Code*

1 Yocto

THE YOCTO PROJECT. IT'S NOT AN EMBEDDED LINUX DISTRIBUTION, IT CREATES A CUSTOM ONE FOR YOU.

The Yocto Project (YP) is an open source collaboration project that helps developers create custom Linux-based systems regardless of the hardware architecture.

The project provides a flexible set of tools and a space where embedded developers worldwide can share technologies, software stacks, configurations, and best practices that can be used to create tailored Linux images for embedded and IOT devices, or anywhere a customized Linux OS is needed.

1.1 Build Agema Yocto

Let's focus on Agema Yocto project. This is a very practical approach. There are a number of high-level steps involved in building Yocto Project for Agema. Most steps are implemented as a bash script in `build_yocto_step_*.sh` file. Let's overview what these steps are below.

- Create a new work directory for Yocto Project.

  Rationale: Maintain a clean separation between Yocto builds.

- Clone `init-kconfig` git project

  Rationale: It contains `build_yocto_step_*.sh` bash script.

- Choose a desired Yocto high-level configuration.

  Rationale: Easily choose Yocto-2.5.2 with BCM-SDK 6.5.15.

- Create and customize Yocto Project.

  Rationale: Add meta layers and update `local.conf`.

- Build Yocto Project.

  Rationale: Run `bitbake core-image-full-cmdline`.

- Upload Yocto Project build artifacts.

  Rationale: Share Yocto eSDK with other teams.

Let's do these steps one-by-one.

1.1.1 Create a new work directory for Yocto Project

```
$ cd $HOME
$ mkdir -p workspace/yocto
$ cd workspace/yocto
```

It is highly advised to keep a separate directory for every Yocto Project build!

1.1.2 Clone initial git project

```
$ cd $HOME/workspace/yocto
$ git clone ssh://git@bitbucket.mrv.co.il:7999/yoc/init-kconfig.git
```

The `init-kconfig` is a tiny project that contains build scripts and *high-level* configuration for Yocto Project build.

1.1.3 Choose a desired Yocto high-level configuration

The `init-kconfig` project contains high-level configuration files – they are found under `defconfigs` directory.

```
$ cd init-kconfig
$ ls -1 defconfigs
$ agema_yocto-2.5.1_bcm-sdk-6.5.15
$ agema_yocto-2.5.1_bcm-sdk-6.5.7
$ agema_yocto-2.5.2_bcm-sdk-6.5.15
$ agema_yocto-2.6_bcm-sdk-6.5.15
```

Every one of these configuration files contains a *high-level* configuration for Yocto Agema build *that may change* relatively frequently. Let's take a look at the meaninful part of this configuration:

```
$ cd init-kconfig
$ cat defconfigs/agema_yocto-2.5.2_bcm-sdk-6.5.15 | grep -v "^#"
CONFIG_BCM_SDK_6_5_15=y
CONFIG_KEEPALIVED_2_0_12=y
CONFIG_YOCTO_2_5_2=y
```

That's simple – this file defines the version of `BCM-SDK` package, `keepalived` package and the version of Yocto Project itself. This is the gist of it. We need to choose one of these configuration files and copy it to `.config` file to create the effective configuration for Yocto Project build. Note, that this follows how Kbuild build system that is used in Linux kernel and other projects. [1] The `kbuild_skeleton` project was used here as a reference. [2]

So, let's "activate" the chosen configuration.

```
$ cd init-kconfig
$ cp defconfigs/agema_yocto-2.5.2_bcm-sdk-6.5.15 .config
```

The `.config` file will be used by the following build steps.

[1] https://github.com/embedded-it/kbuild-template

[2] https://github.com/masahir0y/kbuild_skeleton

1.1.4 Create and customize Yocto Project

Now we run the scripts that will create and customize a classical Yocto Project environment.

— **IMPORTANT NOTE** — Use with care the build step that removes a previous Yocto Project build – `build_yocto_step_02_remove_previous.sh` script!

Here, all the needed git sub-projects will be cloned and organized in a file hierarchy expected by Yocto Project build process.

```
$ cd init-kconfig
$ ./build_yocto_step_01_configure.sh
$ ./build_yocto_step_03_clone.sh
$ ./build_yocto_step_04_checkout_branches.sh
$ ./build_yocto_step_05_customize_yocto_build_conf.sh
$ ./build_yocto_step_06_add_layers.sh
```

Let's comment the scripts above.

The gist of build step scripts

- `build_yocto_step_01_configure.sh`

  Converts `.config` to more handy configuration format.

- `build_yocto_step_03_clone.sh`

  Clones the meta layers using Google `repo` tool.

- `build_yocto_step_04_checkout_branches.sh`

  Checks out git branch according to `.config` configuration.

- `build_yocto_step_05_customize_yocto_build_conf.sh`

  Applies customizations to `local.conf` build configuration.

- `build_yocto_step_06_add_layers.sh`

  Adds the needed meta layers via `bitbake-layers add-layer`.

1.1.5 Build Yocto Project

All the previous build steps may be thought of as a preparation (setup and customization) for the actual Yocto Project build. These steps take about 10-15 minutes of running time on a strong machine. The next script actually builds the whole Yocto Project – it supposedly takes 4 to 5 hours on the same strong machine.

Most of the build time is spent on building GCC toolchain and generic Linux packages. Currently, Yocto image that is used for Agema products contains around 300 packages – see the command below.

How many packages in Agema?

```
$ cd init-kconfig/ybld/poky
$ source ./oe-init-build-env ../build
$ bitbake -g core-image-full-cmdline
$ cat recipe-depends.dot \
  | grep -v -e '-native' | grep -v digraph | grep -v -e '-image' \
  | awk '{print $1}' | sort | uniq | wc -l
```

The last command gives us the answer – image contains `292` packages.

Only two bitbake commands

The actually important Yocto commands that build the image are run by `bitbake` utility. So, there are only two commands that we *actually* need to build Yocto for Agema:

- `bitbake core-image-full-cmdline`
- `bitbake core-image-full-cmdline -c populate_sdk_ext`

The first command builds all the packages and creates the Rootfs image. The second command creates Yocto eSDK self-installable file. Most of the build time is spent on `bitbake core-image-full-cmdline` command.

Zoom in to run bitbake commands

Sometimes, we need to circumvent the high-level build scripts and to run the `bitbake` commands by ourselves in Linux terminal.

To run `bitbake` commands separately (not inside the script) open **new terminal window** and run the below commands:

```
$ cd init-kconfig/ybld/poky
$ source ./oe-init-build-env ../build
$ bitbake core-image-full-cmdline
$ bitbake core-image-full-cmdline -c populate_sdk_ext
```

Run the build script

So, let the build begin.

```
$ cd init-kconfig
$ ./build_yocto_step_07_yocto_build.sh
```

1.1.6 Upload Yocto Project build artifacts

There is only one Yocto build artifact that needs to be shared with the outside world – Yocto eSDK self-installable file. Yocto eSDK needs to be installed on the development machine of every application developer before building and creating the self-installable *bundle* file for Agema device.

Yocto eSDK self-installable file is found in the Yocto `deploy` directory as shown below.

```
$ cd init-kconfig/ybld
$ cd build/tmp/deploy/sdk
$ ls poky-glibc-x86_64-core-image-full-cmdline-i586-toolchain-ext-2.5.2.sh
```

Currently the size of Yocto eSDK self-installable image is around 2 GB. The upload is done via simple `scp tmp/deploy/sdk/$name.* <user>@<IP>:...` commands.

Run the upload script

So, let's do the actual upload of Yocto eSDK:

```
$ cd init-kconfig
$ ./build_yocto_step_08_upload.sh
```

— *NOTE* — The upload step should be run *only* by CI/CD (TeamCity) or *only* when you know what you are doing. Otherwise, Yocto eSDK image that is used by all application developers may be overwritten by not-yet-tested image.

1.1.7 Concluding notes

Run all build scripts in one shot

To wrap it up – all the Yocto Agema build script may be executed locally on development machine in one shot as it is done in CI/CD approach (on TeamCity).

Just copy-paste the commands below.

```
$ cd init-kconfig
$ cp defconfigs/agema_yocto-2.5.2_bcm-sdk-6.5.15 .config
$ ./build_yocto_step_01_configure.sh
$ ./build_yocto_step_03_clone.sh
$ ./build_yocto_step_04_checkout_branches.sh
$ ./build_yocto_step_05_customize_yocto_build_conf.sh
$ ./build_yocto_step_06_add_layers.sh
$ ./build_yocto_step_07_yocto_build.sh
```

Build configuration vs development iteration

Build steps `build_yocto_step_[01..06]*.sh` should be done only once – this is the initial configuration step! Build step `build_yocto_step_07_yocto_build.sh` - is the development iteration that may be done many times.

Custom scripts or bitbake

When working extensively with Yocto development - it is advised to work with `bitbake` commands instead of custom `build_yocto_step_*.sh` scripts. Try the below commands to begin working with `bitbake` commands.

— *NOTE* — Always use new fresh terminal windows when working with Yocto commands (to keep bash environment clean).

Just copy-paste the commands below.

```
$ cd init-kconfig/ybld/poky
$ source ./oe-init-build-env ../build
$ bitbake core-image-full-cmdline
$ bitbake keepalived
```

Yocto dynamic vs static configuration

Linux kernel, U-Boot and many other packages use Kbuild configuration system where the *total* project configuration is kept in one file – `.config` file. Yocto, on the other hand, does not have a static configuration saved in a file. Yocto creates the projects configuration (analogous to `.config`) every time when `bitbake` command is executed. To observe this configuration file let's use the `bitbake -e <recipe>` command.

```
$ cd init-kconfig/ybld/poky
$ source ./oe-init-build-env ../build
$ bitbake -e keepalived
```

The configuration is huge – about 20,000 lines – and it contains all the needed configuration to build a recipe.

Log file for build script

The full log of the build is saved in `ylog` file. It is advised to `tail -f` this log file as the Yocto build is running in the background.

```
$ cd init-kconfig
$ tail -f ylog
```

1.2 Agema dev install environment

- Two important variables on the script are `yocto` and `bcm_sdk`.

  ```
  # ~~~~~~~~~~~~~~~~~
  # Script parameters
  # ~~~~~~~~~~~~~~~~~
  yocto="2.5.2" # default
  bcm_sdk="6.5.15" # default
  ```

- Clones OpenClovis on host development machine.

  OpenClovis makefiles MUST be accessed during Agema build.

- Install Yocto eSDK on host development machine.

  Re-create Rootfs image using `devtool build-image` command, because Yocto eSDK installer does not contain Rootfs images.

- Clone metaswitch project (it is separate git repository).

  It is advised to use Google repo tool to clone all the needed git projects:

  ```
  $ git clone
      ↪ ssh://git@bitbucket.mrv.co.il:7999/agema/agema_build.git
  $ cd agema_build
  $ ./setup_build_env.sh
  ```

1.3 Agema build script `build_agema.py`

- Reads the JSON environment file

- Sets version

- Build routing apps – actually calles `SAFplus_build.sh` script

- The `SAFplus_build.sh` script sources `environment-setup-agema` file to activation Yocto eSDK environment and define location of OpenClovis source tree.

- For incremental build – call `build_agema.py -q` (calls to `SAFplus_build.sh`).

- For full build – call `build_agema.py` (calls to `SAFplus_build.sh all`).

- Copy all components to temporary folder (e.g., Yocto Rootfs, `selfextractor.sh`, `rootfs_extras`, kernel image, `SCNodeIO.tgz`).

  `SCNodeIO.tgz` contains all network applications (developed by ADVA) in the form of OpenClovis agents.

  Yocto Rootfs contains only "build-once" artifacts.

- Create `tar.gz` archive from above components and prepend (using `cat` Linux utility) the `selfextractor.sh` script – bundle is ready!

1.4 Agema Yocto upgrade guidelines

- Use official mega manual from Yocto web site as the reference.
- Currently, we use Yocto 2.5.2.
- Yocto 2.6 basically exists, but BCM-SDK build fails.
- Expected changes to swtich to Yocto 2.6:

  Update `init-kconfig`.

  Update vanilla meta layers (update local git from upstream repository).

  Update ADVA meta layers (create "2.6" branches – they will contain ported commits if any).

  Update `agema_install_dev_env.sh` script.

1.5 BCM-SDK upgrade guildlines

- Good reference is CPSS upgrade flow described in this document.
- Download BCM-SDK and KBP release archives from Broadcom.
- Extract BCM-SDK archive and port all commits except KBP commits.
- Extract KBP archive and port remaining KBP commits.

  KBP demands specifal focus and should be done slowly and correctly.

- Compile and boot the device.

1.6 Agema fast build

- To enable fast build run `make menuconfig`

  ```
  $ cd init-kconfig
  $ cp defconfigs/agema_yocto-2.5.2_bcm-sdk-6.5.15 .config
  $ make menuconfig
  ```

  Choose fast build, save and exit (it updated `.config`).

  The `build_yocto_step_05_customize_yocto_build_conf.sh` checks fast build option and updates `local.conf`.

- Fast build uses download cache and artifacts cache.

  Currently, download cache is enabled and works ok, but artifacts cache is disabled, because is prevents Linux kernel build.

1.7 Agema initial installation

- MUST-have assumption – ONIE Linux is always installed on Agema device.
- ONIE's purpose is to install NOS.
- ONIE searches for installation script with standard name – `onie-installer`.
- The `onie-installer` implements the actuall installation procedure.
- To install NOS from USB – disconnect OOB port.
- To install NOS from DHCP server – take out USB stick.
- Some installation details are described in `z4806/onie-tools/README.md`.
- Both USB and network installation options demands the same set of files.

  onie-installer onie-installer_yocto ver file bundle

- The `onie-installer` script installs basic Yocto Linux on `NOS-Rescue` partition and boots into it.
- ONIE Linux environment is too simple to run our bundle installation – hence, we need `NOS-Rescue`.
- The GRUB menu (grub.cfg file) is part of ONIE Linux and we update it during initial installation.

1.8 Agema bundle upgrade

- Bundle is self-extracted with `selfextractor.sh` script.
- Then `bundle_install.sh` script is executed.

  This script contains the logic to understand the where to install the bundle (when no `OTHER_PARTITION` is provided as in initial installation flow.

1.9 Agema bundle uninstall

- Only two steps here - delete all ADVA partitions and restore original `grub.cfg`.
- See `onie-tools/onie-installer_yocto` as reference.
- To delete partitions use:

  `sgdisk -d 4 -d 5 -d 6 -d 7 -d 8 -d 10 -d 11 -d 12 -d 13 -g /dev/sda`.

1.10 Yocto work scenarios

1.10.1 Add new recipe

Add recipe with devtool

```
$ cd init-kconfig/ybld/poky
$ source ./oe-init-build-env ../build
$ devtool add libfixbuf
    ↪ https://tools.netsa.cert.org/releases/libfixbuf-2.3.1.tar.gz
```

Change source and build

```
$ cd workspace/sources/libfixbuf
$ vim ...
$ devtool build libfixbuf
```

Add recipe to existing meta layer

```
$ devtool finish libfixbuf ../poky/meta-agema
```

Finish the flow

```
$ devtool reset libfixbuf
$ rm -rf workspace/sources/libfixbug
```

See official Yocto Project documentation for more details.

1.10.2 Modify existing package

When a recipe exists in some meta layer under `poky` directory, then the proper way to work on this package (for example, changing the source code) is via `devtool modify` command.

```
$ devtool modify keepalived
```

After the `devtool modify` command is finished, the source code of `keepalived` package is found under the `workspace/sources/keepalived` directory. Now we we can build the `keepalived` package using `devtool build` command.

```
devtool build keepalived
```

The recipe build logs are found in `tmp/work/i586.../keepavlied/.../temp` directory – exactly as working with `bitbake` commands.

1.10.3 Create new layer

What is Yocto layer? It is a standard file When to add a new layer and when to use the existing layer? If the software project we add is a big thing (for example, BCM-SDK, OpenClovis, Flexera), then it is better to create a separate layer (meta-*) for this project.

If a software module may be shared between different products (Agema and OptiSwitch), then it is better to keep it in the shared meta layer, say, `meta-adva-networking`. For example, to easily share `keepalived` is it better to move it out from `meta-agema` layer to, say, `meta-adva-networking` layer.

To create Yocto meta layer we may use the command below:

```
bitbake-layers create-layer ...
```

Let’s take a look at the important directories and file in a simple meta layer.

A very simple example:

```
$ cd/ybld/poky/meta-flexera
$ tree
.
|-- conf
|   +-- layer.conf
+-- recipes-flexera
    +-- flexera
        +-- flexera.bb
```

A little bit more complex example includes `bbappend` file and two patch files.

```
$ cd/ybld/poky/meta-agema
$ tree
.
+-- conf
    +-- layer.conf
    +-- recipes-daemons
    +-- keepalived
        |-- keepalived_2.0.12.bb
        |-- keepalived_2.0.12.bbappend
        +-- keepalived-2.0.12
            |-- 0001-Change-VRRP-state.patch
            +-- 0002-Fix-handling-of-VMAC.patch
```

cd .../build !!! bitbake-layers add-layer ...

check in bblayers.conf that the layer is added (bitbake-layers show-layers)

1.10.4 Debug under Yocto eSDK

Application developers do not have a full Yocto build. Instead, every application developer installs Yocto eSDK on the development machine. Yocto eSDK can be used to add new recipes or modify existing recipes via `devtool` command – exactly as it is done in Yocto environment.

Let's try to debug some specefic package – `snmp`.

The first step is to activate Yocto eSDK environment:

```
$ cd $HOME/yocto/2.5.2_bcm-sdk-6.5.15/esdk/poky_sdk
$ source environment-setup-i586-poky-linux
```

Well, the development flow that changes the existing recipe is called "modify". Hence, we should use `devtool modify` command.

```
$ cd $HOME/yocto/2.5.2_bcm-sdk-6.5.15/esdk/poky_sdk
$ source environment-setup-i586-poky-linux
$ devtool modify snmp
```

It fails. There is no `snmp` recipe.

Let's check a proper recipe name in `poky_sdk/conf/local.conf` file under Yocto eSDK – it is called `net-snmp`. Now we can build `net-snmp` recipe under Yocto eSDK.

```
$ cd $HOME/yocto/2.5.2_bcm-sdk-6.5.15/esdk/poky_sdk
$ source environment-setup-i586-poky-linux
$ devtool modify net-snmp
$ devtool build net-snmp
```

1.10.5 Create new layer for Agema

Let's take a look at the real example of adding new meta layer to Agema Yocto. The steps are shown below:

- *Create meta layer*

  Create a new meta layer using `bitbake-layers create-layer` command.

- *Add meta layer*

  Add this meta layer to `bblayers.conf` using `bitbake-layers add-layer` command.

- *Create recipe*

  Create a new recipe in the new layer (say, `flexera.bb` recipe).

- *Build recipe*

  Build this new recipe with `bitbake` command (`bitbake flexera`).

- *Create git*

  Save the new layer in git (`git init; git add .; git commit -m ...`).

- *Verify repo manifest*

  Verify that git default branch is consistent with repo tool manifest file!

- *Push to BitBucket*

  Push git project that contains the new meta layer to BitBucket.

- *Update repo manifest*

  Verify that repo tool manifest is updated and uses the correct git branch.

- *Update add layers script*

  The `build_yocto_step_06_add_layers.sh` script with `bitbake-layers add-layer` command

Now, let's test from scratch that the new layer and the new recipe work OK.

- **Clone** – Clone `init-kconfig` git project.
- **Configure** – Run build steps 1 to 6.
- **Check** – Check in `bblayers.conf` that the layer is really added.
- **Build** – Run the build step 7 to build the Yocto in full.

1.11 Yocto Tips

1.11.1 TIP: Use Yocto documentation!

Yocto project has a great documentation – it is one of the strong sides of the project. We may access the documentation of the Yocto web site.

1.11.2 TIP: Use Google!

Yocto is an open source project with great documentation and a big user base. A lot of answers may be found on the internet – just Google for it.

1.11.3 TIP: To change existing recipe always use devtool!

If we want to debug some recipe and perform a bug fix in the source code – it is always better to use `devtool`. There are a number of reasons for that. One important reasons is that `devtool` **always** creates a local git to track the changes.

```
$ cd $HOME/yocto/2.5.2_bcm-sdk-6.5.15/esdk/poky_sdk
$ source environment-setup-i586-poky-linux
$ devtool modify keepalived
```

Now, under `workspace/sources/keepalived` a new fresh local git is created.

1.11.4 TIP: Recipe build artifacts in Yocto and Yocto eSDK!

The work directory of a recipe is the same for Yocto and Yocto eSDK. This directory is `tmp/work/i586-poky-linux/<package>/<version>`.

For example, in Yocto eSDK:

```
$ cd $HOME/yocto/2.5.2_bcm-sdk-6.5.15/esdk/poky_sdk
$ cd tmp/work/i586-poky-linux/keepalived/2.0.12-r0
ls -1
deploy-rpms
```

```
license-destdir
pkgdata
sstate-install-packagedata
sstate-install-package_qa
sstate-install-package_write_rpm
sstate-install-populate_lic
sstate-install-populate_sysroot
sysroot-destdir
temp
```

1.11.5 TIP: Yocto eSDK contains all Yocto recipes!

Yocto eSDK is actually Yocto with some stuff cut-out. So, all the Yocto recipes found under `poky` directory are present in Yocto eSDK under `poky_sdk/layers/poky/` directory.

```
$ cd $HOME/yocto/2.5.2_bcm-sdk-6.5.15/esdk/poky_sdk
$ ls -1 layers/poky/
bitbake
LICENSE
meta
meta-agema
meta-bcmsdk
meta-flexera
meta-openclovis
meta-openembedded
meta-poky
meta-security
meta-skeleton
meta-sysrepo
meta-virtualization
meta-yocto-bsp
oe-init-build-env
scripts
```

1.11.6 TIP: Run all Yocto commands from one place!

In case of Yocto environment – run all Yocto commands (`bitbake`, `devtool` and `oe-*`) from `build` directory:

```
$ source ./oe-init-build-env ../build
$ cd init-kconfig/ybld/build
$ bitbake ...
$ devtool ...
```

In case of Yocto eSDK – run all Yocto commands from `poky_sdk` directory:

```
$ cd $HOME/yocto/2.5.2_bcm-sdk-6.5.15/esdk/poky_sdk
$ source environment-setup-i586-poky-linux
$ devtool ...
```

1.11.7 TIP: Use fresh terminal environment!

The first step in working with Yocto **and** Yocto eSDK is to activate the proper bash environment. So, Yocto demands `source ./oe-init-build-env ...` as the first command. And Yocto eSDK demands `source environment-setup-i586-poky-linux` as the first command.

These bash environments *cannot* be mixed. Hence, in case of any doubt – just open a new terminal window and perform the proper `source ...` command.

1.11.8 TIP: Do NOT run Yocto commands in parallel!

All Yocto commands *MUST* be executes serially (not in parallel). So, do not run the below commands in parallel:

- `bitbake`
- `devtool`
- `oe-pkgdata-util`
- `oe-*`

1.11.9 TIP: Always use full path!

For `bblayers.conf` file and other Yocto commands use full path.

Pay attention that `Yocto` in **all** the places (command, logs, environment, etc.) uses full paths. So, we also need to use the full paths in all the commands.

1.11.10 TIP: Yocto, Yocto eSDK or vanilla Yocto?

Yocto full from-scratch build takes 4-5 hours. So, we need to be careful with changes and work scenarios that we perform in Yocto build environment. On the other hand, Yocto eSDK may be fully re-created (re-installed) in 10-15 minutes. This is why we should always prefer to work in Yocto eSDK environment.

In addition, vanilla Yocto environment may also be found handy for various development tasks. This environment may be easily created using the commands below.

```
$ git clone https://git.yoctoproject.org/git/poky
$ cd poky
$ source ./oe-init-build-env ../build
$ bitbake core-image-full-cmdline
```

1.11.11 TIP: Yocto simulation

Yocto simualtion is done via QEMU using Yocto-integrated `runqemu` command.

```
$ cd poky
$ source ./oe-init-build-env ../build
$ runqemu qemux86 nographic
```

It is important to understand the difference between running simulation in Yocto, Yocto eSDK and under Z4806 environment. So, the QEMU simulation under Yocto and Yocto eSDK does not contain “our additional” stuff. So, the simulation in this case in running under “clean” Yocto Linux environment.

On the other hand, when running the same QEMU simulation under Z4806 environment, all “our additional” stuff is also being executed. The reason is that `rootfs_extras` directory from Z4806 is applied on top of Yocto Rootfs. This `rootfs_extras` directory contains scripts to invoke “our additional” stuff. One good example for such the script is `S90middleware`.

To run the simulation under Z4806 environment:

```
$ cd z4806
$ ./yocto/agema_sim_step_1_setup_base_rootfs.sh
$ ./yocto/agema_sim_step_2_add_routing_apps.sh
$ ./yocto/agema_sim_step_3_run_simulation.sh
```

1.11.12 TIP: To finish QEMU simulation use poweroff

Yocto QEMU simulation does not have to be “killed”. It is a virtual machine that may be accessed via ssh for many development tasks. But to to “kill” this QEMU simulation just run `poweroff` from `root` user of simulation command line.

```
$ cd poky
$ source ./oe-init-build-env ../build
$ runqemu qemux86 nographic
$ root # password 123456
$ poweroff
```

1.11.13 TIP: SSH connect to QEMU simulation

It is easy and handy to connect to the running QEMU simulation via ssh. When simulation is booted, just login as `root` and find out IP address using `ifconfig`.

```
$ cd poky
$ source ./oe-init-build-env ../build
$ runqemu qemux86 nographic
$ root # password 123456
$ ifconfig
eth0 Link encap:Ethernet HWaddr 52:54:00:12:34:02
         inet addr:192.168.7.2 Bcast:192.168.7.255 Mask:255.255.255.0
         inet6 addr: fe80::5054:ff:fe12:3402/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
         RX packets:33 errors:0 dropped:0 overruns:0 frame:0
         TX packets:101 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:39595 (38.6 KiB) TX bytes:28315 (27.6 KiB)
```

Pay attention that when QEMU simulation is running there is a new network interface (`tap0`) on host machine. Switch to new terminal on host machine and do `ifconfig` to find out the IP address of `tap0` network interface.

```
$ ifconfig
```

```
tap0 Link encap:Ethernet HWaddr e2:88:69:16:63:12
         inet addr:192.168.7.1 Bcast:192.168.7.255 Mask:255.255.255.255
         inet6 addr: fe80::e088:69ff:fe16:6312/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
         RX packets:221 errors:0 dropped:0 overruns:0 frame:0
         TX packets:103 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:83119 (83.1 KB) TX bytes:50191 (50.1 KB)
```

Now, the files may be copied from host machine to the running QEMU simulation machine using a regular `scp` Linux command.

1.11.14 TIP: Update Rootfs after package build

After a recipe is built, we need to update Rootfs image to contain the updated recipe build artifacts.

Update Rootfs in Yocto environment

```
$ cd init-kconfig/ybld/poky
$ source ./oe-init-build-env ../build
$ bitbake core-image-full-cmdline
```

Update Rootfs in Yocto eSDK environment

```
$ cd $HOME/yocto/2.5.2_bcm-sdk-6.5.15/esdk/poky_sdk
$ source environment-setup-i586-poky-linux
$ devtool build-image
```

After the image is updated, QEMU simulation may be run again to access the updated recipe build artifacts.

1.11.15 TIP: Build sub-project in Z4806

```
$ cd z4806
$ source
    ↪ platform/open_clovis/OP9500/scripts/environment-setup-SAFplus-build
$ cd common/license/build
$ make clean
$ make
```

2

BitBucket

2.1 BitBucket Troubleshooting

2.1.1 Recovery mode

The first and most important step in BitBucket debug is to login to the web interface, because all the important configuration is done via the web interface. When no credentials exist to login to the web interface (for example, if LDAP configuration is wrong), then the last resort is to enter the recovery mode – known as `recovery_mode` in BitBucket help. The easiest way to find a detailed help is to Google for "BitBucket recovery mode".

We are going to use two Linux users:

- `mrv` - for login and for general stuff
- `atlbb` - for start and stop of BitBucket daemons

The credentials of the Linux users we are going to use are shown in Table 2.1.

Let's login to BitBucker Linux machine with `mrv` user and switch to `atlbb` user as shown below:

```
$ ssh mrv@10.32.23.224
```

Linux user	Password	Root permissions
mrv	YoramRan2016	Yes
atlbb	123456	No

Table 2.1: Information on BitBucket-related Linux users.

```
$ su atlbb
$ whoami
$ cd $HOME
```

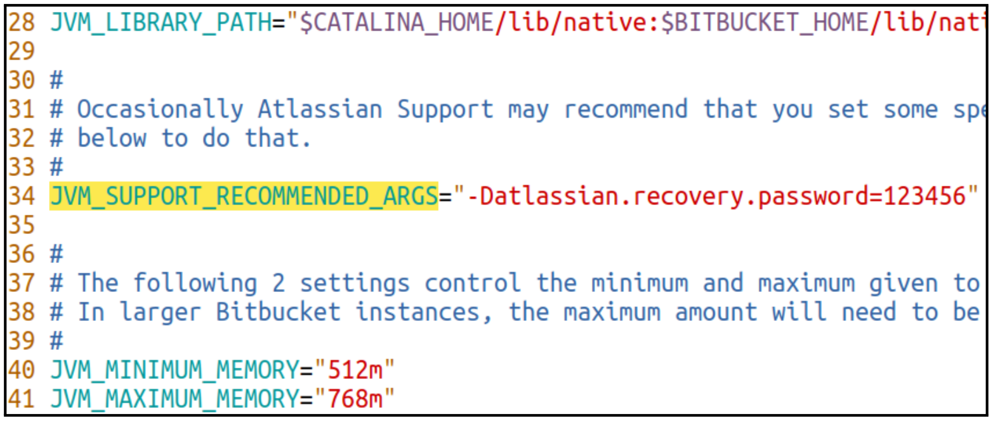
Now, we need to update the **setenv.sh** script to enable BitBucket recovery mode via `JVM_SUPPORT_RECOMMENDED_ARGS` variable. So, we need to open the **setenv.sh** file in editor, find the `JVM_SUPPORT_RECOMMENDED_ARGS` variable in the file and add `"-Datlassian.recovery.password=123456"`.

The end result should like shown below:

```
JVM_SUPPORT_RECOMMENDED_ARGS="-Datlassian.recovery.password=123456"
```

Now, let's find the **setenv.sh** file and open it using an editor of your choice:

```
$ vim atlassian/bitbucket/4.11.2/bin/setenv.sh
```



```
JVM_LIBRARY_PATH="$CATALINA_HOME/lib/native:$BITBUCKET_HOME/lib/nat

#
# Occasionally Atlassian Support may recommend that you set some spe
# below to do that.
#
JVM_SUPPORT_RECOMMENDED_ARGS="-Datlassian.recovery.password=123456"

#
# The following 2 settings control the minimum and maximum given to
# In larger Bitbucket instances, the maximum amount will need to be
#
JVM_MINIMUM_MEMORY="512m"
JVM_MAXIMUM_MEMORY="768m"
```

Figure 2.1: Enabling recovery mode in setenv.sh file.

Now, let's save the file and restart BitBucket server.

— **IMPORTANT NOTE** — BitBake start, stop and restart MUST be done from **atlbb** user.

```
$ su atlbb
$ service bitbucket restart
```

Now, we can open the web browser at `https://10.32.23.224:8443` address and login to recovery mode using “123456” password.

To disable the recovery mode just return `JVM_SUPPORT_RECOMMENDED_ARGS` variable to its original state and restart BitBucket again.

2.1.2 Notes

2.1.3 Root permissions

Only `mrv` user has the root permissions. The BitBucket user (`atlbb`) does not have the root permissions. So, if you need to run a command with the root permissions, do it via `mrv` user.

— **IMPORTANT NOTE** — Note, that BitBucket troubleshooting should in MOST cases be performed WITHOUT the root permissions! If you use `sudo` in command line – most likely, you are wrong!

2.1.4 File ownership

Verify that all the BitBucket-related files and directories are under ownership of a proper Linux user – `atlbb`. In case of uncertainty just set the ownership to `atlbb` as shown below:

```
$ ssh mrv@10.32.23.224
$ su atlbb

$ cd $HOME/atlassian
$ sudo chown -R $USER:$USER .

$ cd $HOME/bbtl_server_data
$ sudo chown -R $USER:$USER .
```

2.1.5 Daemons

Verify that both BitBucket daemons are running.

```
$ ps aux | grep bucket | grep java
```

There are many arguments provided to BitBucket daemons, so the output may look overwhelming.

```
$ ps aux | grep bucket | grep java

atlbb 1845 0.4 5.4 7898356 449648 ? Sl Apr16 10:47
    ↪ /usr/lib/jvm/java-8-oracle/bin/java -Xms256m -Xmx1g
    ↪ -Djava.awt.headless=true -XX:+UseParNewGC -XX:+UseConcMarkSweepGC
    ↪ ...

atlbb 1921 3.4 21.3 7032992 1741448 ? Sl Apr16 87:57
    ↪ /usr/lib/jvm/java-8-oracle/bin/java
    ↪ -Djava.util.logging.config.file=/home/atlbb/atlassian/bitbucket ...
```

2.1.6 Log files

It may be very helpful to observe the changes in BitBucket log files upon starting or stopping BitBucket daemons. Or – when BitBucket web interfaces works OK – on login attempts.

```
$ ssh mrv@10.32.23.224
$ su atlbb
$ cd $HOME/bbtl_server_data/log
$ tail -f atlassian-bitbucket.log
$ tail -f atlassian-bitbucket-access.log
```

3

CPSS Upgrade

Let's look at the high-level steps needed to upgrade CPSS from release 2018.2 to release 2019.2. The steps are general for any CPSS versions – 2018.2 and 2019.2 are used only as example.

3.0.1 High-level CPSS upgrade steps

- Download CPSS release archive file.
- Unzip the archive and run `build_cpss.sh` script.
- Create a new CPSS 2019.2 vanilla commit.
- Create a new git branch based on CPSS 2019.2 vanilla commit.
- Port all "CPSS 2018.2" commits to newly created "CPSS 2019.2" branch.

The CPSS upgrade project is not immediately obvious project, so, that a number of "try-fail-retry" iterations may be required.

Let's try to zoom in into the CPSS upgrade steps and provide some details.

3.0.2 Detailed CPSS upgrade steps

Download CPSS release archive file

The archive file with CPSS release can be downloaded from Marvell Extranet website. The example of the file name is `Cpss-Source-DxCh-4.2_2019_2_008.zip`. Also, let's prepare the work directory for new CPSS release and the current working CPSS git.

```
$ mkdir workspace/Cpss-2019.2
$ mkdir workspace/Cpss-2018.2
```

These directories will be used below.

Unzip CPSS release archive

```
$ cd workspace
$ unzip Cpss-Source-DxCh-4.2_2019_2_008.zip -d Cpss-2019.2
$ cd Cpss-2019.2
$ unzip Cpss-PP-DxCh-4.2_2019_2_008.zip
```

CPSS release archive should be "unzipped twice" – first unzip the release archive itself and then `Cpss-PP-DxCh*.zip` archive.

Unzip CPSS with `build_cpss.sh` script

```
$ cd Cpss-2019.2
$ ./build_cpss.sh MSYS DX_ALL UTF_NO UNZIP NOKERNEL
```

Convert all text files to Unix format

```
$ cd Cpss-2019.2/cpss
$ find . -type f | sort | xargs file | grep ASCII \
  | cut -d':' -f1 | xargs dos2unix
$ find . -type f | sort | xargs file | grep Unicode \
  | cut -d':' -f1 | xargs dos2unix
```

— *NOTE* — This is an important step!

Not all CPSS source code files are in Unix text format. Without `dos2unix` conversion git may fail to easily apply the patches.

Clone current CPSS git project

```
$ cd Cpss-2018.2
$ git clone ssh://git@bitbucket.mrv.co.il:7999/osv2018/cpss.git
```

Commit CPSS 2019.2 vanilla code

```
$ cd workspace/Cpss-2018.2/cpss
$ git checkout marvell_releases
$ rm -rf *
$ cp workspace/Cpss-2019.2/build_cpss.sh .
$ cp -R workspace/Cpss-2019.2/cpss .
$ git add .
$ git commit -m "Marvell Release CPSS-Build_4.2_008-Ver_4.2.2019.2"
```

— **IMPORTANT NOTE** — It is very important to remove all vanilla CPSS files (as it is shown in the listing above) before copying CPSS 2019.2 files.

Now, the local git contains CPSS 2019.2 vanilla code.

Push vanilla CPSS 2019.2 to git server

```
$ git push -u origin marvell_releases:marvell_releases
$ git tag CPSS-Build_4.2_008-Ver_4.2.2018.2
$ git push --tags origin
```

It is very important to save CPSS 2019.2 code *and* git tag to the git server.

Create new branch for CPSS 2019.2

```
$ cd workspace/Cpss-2018.2/cpss
$ git checkout -b cpss-2019.2-master marvell_releases
```

All the current commits need to be ported from CPSS 2018.2 master branch to CPSS 2019.2 master branch. At the end of the porting `cpss-2019.2-master` branch will contain all the accumulative development work that worked on top CPSS 2018.2 release.

Save all commits based on CPSS 2018.2

```
$ cd workspace/Cpss-2018.2/cpss
$ git checkout cpss-2018.2-master
$ git format-patch CPSS-Build_4.2_250-Ver_4.2.2018.2.. /tmp
```

That is, all the commits that were applied after the CPSS 2018.2 *vanilla* commit are saved aside as patch files.

Begin applying commits on top CPSS 2019.2 vanilla

All the previous steps were easy - this was only a preparation. The actual time-intensive work begins now.

```
$ cd workspace/Cpss-2018.2/cpss
$ git checkout cpss-2019.2-master
$ git am /tmp/xxx/0*
```

Currently, there are aboud 100 commits – every one of them needs to be applied with `git am` command on top of `cpss-2019.2-master` branch. This should be done slowly and with focus on every detail. This process may easily take a 2-4 weeks of highly focused work.