Version: 6

On this page

Is this page helpful?

Custom meta layers, recipes and images in Yocto Project (hello-world examples)

Introduction

This article describes how you can create a new meta layer, how you can add your own hello-world application in a recipe, and how you can create your own image in Yocto Project / OpenEmbedded (from now on only called Yocto Project in this article). We don't go into details, it should only give you an easy introduction on how to start. For further details please read the Yocto Project Reference Manual.

This article complies with the Typographic Conventions for Toradex Documentation.

Prerequisites

If you are building a Reference Image for Yocto Project:

- Build a Reference Image with Yocto Project.
  - For starting, we recommend compiling one of our reference images without changes.

If you are building Torizon OS:

- Build Torizon OS With Yocto.
  - For starting, we recommend compiling Torizon OS without changes.

Send Feedback!

Create a Meta Layer

Once you have an image compiled as proposed in the section *Prerequisites* above, create a new meta-customer layer:

INFO

`bitbake-layers create-layer PATH` creates a new layer with a basic directory structure at `PATH`.

```
$ cd ../../build/
$ . export # or source setup-environment when building Torizon
OS
$ bitbake-layers create-layer ../layers/meta-customer
```

The new meta layer will be generated with an example recipe:

```
$ tree ../layers/meta-customer/
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
└── recipes-example
    └── example
        └── example_0.1.bb

3 directories, 4 files
```

After that you can add the path to the newly created layer to the Yocto Project environment in `conf/bblayers.conf`. Below is an example, where the line `${TOPDIR}/../layers/meta-customer \` was added to the `BBLAYERS` variable:

TIP

Don't replace the content of your `bblayers.conf` with this one. Just add the line with your new layer.

Send Feedback!

```
# LAYER_CONF_VERSION is increased each time
build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "7"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  ${TOPDIR}/../layers/meta-toradex-nxp \
  ${TOPDIR}/../layers/meta-freescale \
  ${TOPDIR}/../layers/meta-freescale-3rdparty \
  ${TOPDIR}/../layers/meta-toradex-bsp-common \
  ${TOPDIR}/../layers/meta-openembedded/meta-oe \
  ${TOPDIR}/../layers/meta-openembedded/meta-filesystems \
  ${TOPDIR}/../layers/meta-openembedded/meta-gnome \
  ${TOPDIR}/../layers/meta-openembedded/meta-xfce \
  ${TOPDIR}/../layers/meta-openembedded/meta-networking \
  ${TOPDIR}/../layers/meta-openembedded/meta-multimedia \
  ${TOPDIR}/../layers/meta-openembedded/meta-python \
  ${TOPDIR}/../layers/meta-freescale-distro \
  ${TOPDIR}/../layers/meta-toradex-demos \
  ${TOPDIR}/../layers/meta-qt5 \
  ${TOPDIR}/../layers/meta-toradex-distro \
  ${TOPDIR}/../layers/meta-yocto/meta-poky \
  ${TOPDIR}/../layers/openembedded-core/meta \
  ${TOPDIR}/../layers/meta-customer \
"
```

CAUTION

There is a command to add a new layer to bblayer.conf: `bitbake-layers add-layer`. But this includes the meta layer with absolute paths, which can be avoided by adding it manually.

Initialize a Git Project (Mandatory for Torizon OS)

Revision control is something that one must use during modern software development. We recommend Git for meta layers. Inside your new layer directory, run:

Send Feedback!

```
$ git init
$ git commit -m "Initial Commit" -m "Add <meta-mylayer> from
template"
```

On Torizon OS all your custom layers **must be version controlled by Git**, due to how we include layer revision information with OSTree. If you don't do it, a similar error will pop up during your build:

```
WARNING: Failed to get layers information. Exception: <class
'bb.process.ExecutionError'>
WARNING: torizon-core-docker-1.0-r0 do_image_ostreecommit:
Failed to get layers information. Exception: <class
'bb.process.ExecutionError'>
ERROR: torizon-core-docker-1.0-r0 do_image_ostreecommit:
Execution of '/workdir/build-torizon/tmp-
torizon/work/apalis_imx8-tdx-linux/torizon-core-docker/1.0-
r0/temp/run.do_image_ostreecommit.290621' failed with exit
code 1:
error: Parsing oe.layers=None : 0:expected value
WARNING: exit code 1 from a shell command.
```

Working With Non-Git Revision Systems

While Git is one of the most popular and widely used revision control systems for software, there are of course other solutions out there. For the time being we at Toradex choose to only support Git as a compatible revision control for OSTree, and therefore Torizon by extension. Of course it is possible that Git may not be your chosen system for revision control. In such a case it becomes necessary to work around this limitation in order to avoid the above build error.

Some easy work-arounds:

- Initialize your custom layer as a "fake" git project.
  - This can be done like so: `git init && git commit -m 'fake GIT project'`.
  - You don't need to link this to a remote Git project, a local Git project is all that is required.
- If possible for your infrastructure you can also just mirror your project as a Git project. Then the mirrored Git project can be used for building.

Send Feedback!

  - While this approach requires more work, the benefits of it include having real layer revision information included in OSTree. Which is important if you are interested in using OTA updates.
- Add compatibiltiy for whichever revision control system you use.
  - This file here, defines how we extract layer revision information for OSTree.
  - In theory this file can be edited/added upon to add support for your specific revision control system. In such a case as with all our open-source code we would be willing to review and accept such patches.

These are just a few of the more practical work-arounds for non-Git users. Though in the end there are many ways to get around this. The only thing to keep in mind is whether you are interested in or plan to use the OTA update features of Torizon. In this case it is heavily recommended to use Git to have the best possible compatibility with all our Torizon tools and systems.

Create a Recipe

Now that we have a meta layer we can create a custom recipe e.g. a hello-world:

INFO

Having hello-world/hello-world/ is on purpose. The top folder is used for the recipe, while the subfolder is used for the sources.

```
$ cd ../layers/meta-customer/
$ mkdir -p recipes-customer/hello-world/hello-world
```

Create a recipe file `recipes-customer/hello-world/hello-world_1.0.0.bb`:

```
recipes-customer/hello-world/hello-world_1.0.0.bb
```

```
# Package summary
SUMMARY = "Hello World"
# License, for example MIT
LICENSE = "MIT"
# License checksum file is always required
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/files/c
licenses/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
```

Send Feedback!

```
# hello-world.c from local file
SRC_URI = "file://hello-world.cpp"

# Set LDFLAGS options provided by the build system
TARGET_CC_ARCH += "${LDFLAGS}"

# Change source directory to workdirectory where hello-
world.cpp is
S = "${WORKDIR}"

# Compile hello-world from sources, no Makefile
do_compile() {
    ${CXX} -Wall hello-world.cpp -o hello-world
}

# Install binary to final directory /usr/bin
do_install() {
    install -d ${D}${bindir}
    install -m 0755 ${S}/hello-world ${D}${bindir}
}
```

INFO

Yocto Project also provides the tools devtool and recipetool to create or change recipes. See the manual for more details.

Add the hello world sources to `recipes-customer/hello-world/hello-world/hello-world.cpp`:

```
recipes-customer/hello-world/hello-world/hello-world.cpp
```

```
#include <stdio.h>

int main(int argc, char *argv[]){
    printf("Hello world!\n");
    return 0;
}
```

After creating your recipe and providing the sources, you would be able to build the package:

Send Feedback!

```
$ cd build/
$ bitbake hello-world
```

Add the Package to an Existing Image

To add a recipe to an existing image, you can add the additional packages to be installed to build/conf/local.conf. In the following example we add the hello-world from the chapter Create a recipe.

```
IMAGE_INSTALL:append = " hello-world"
```

After that, for the moment you can rebuild **an existing image** as the Reference Minimal Image or Reference Multimedia Image. Now it will contain the hello-world binary under `/usr/bin/hello-world`.

Build the image using `bitbake` as explained on Build a Reference Image with Yocto Project. Later on in this article, we will focus on creating your own image.

Customize the Kernel

Let's assume you want to add a custom device-tree to the kernel and enable an additional kernel module. How can you do that?

We recommend changing the kernel outside of Yocto to reduce the integration and test cycle time. Please follow the instructions in this article: Build U-Boot and Linux Kernel from Source Code.

After several integration and test cycles you should come up with a devicetree file and kernel configuration that fits your custom application. Now you need to integrate this changes into your custom meta layer.

```
$ cd ../layers/meta-customer/
$ mkdir -p recipes-kernel/linux/linux-toradex
```

You want to append some changes to our linux-toradex kernel recipe need to create a bbappend file recipes-kernel/linux/linux-toradex%.bb

Send Feedback!

note that instead of % you may want to append the change to a specific version (e.g. recipes-kernel/linux/linux-toradex_5.15%.bbappend). Check Build U-Boot and Linux Kernel from Source Code for the kernel version which fits your module.

The bbappend file looks as follows:

```
recipes-kernel/linux/linux-toradex%.bbappend
```

```
FILESEXTRAPATHS:prepend := "${THISDIR}/linux-toradex:"

# Prevent the use of in-tree defconfig
unset KBUILD_DEFCONFIG

CUSTOM_DEVICETREE = "my-custom-devicetree-file.dts"

SRC_URI += "\
    file://${CUSTOM_DEVICETREE} \
    file://custom-display.patch \
    file://defconfig \
    "

do_configure:append() {
    # For arm32 bit devices
    # cp ${WORKDIR}/${CUSTOM_DEVICETREE}
${S}/arch/arm/boot/dts
    # For arm64 bit freescale/NXP devices
    cp ${WORKDIR}/${CUSTOM_DEVICETREE}
${S}/arch/arm64/boot/dts/freescale
}
```

Create the directory recipes-kernel/linux/linux-toradex where you store the additional files which you include in SRC_URI.

First we add a patch which adds a custom-display. Patches are automatically applied by Yocto. You don't have to do anything if the file has the ending .patch.

```
recipes-kernel/linux/linux-toradex/custom-display.patch
```

```
diff --git a/drivers/gpu/drm/panel/panel-simple.c
b/drivers/gpu/drm/panel/panel-simple.c
index e817a71062dbb..2d9f1ca8a04bb 100644
--- a/drivers/gpu/drm/panel/panel-simple.c
+++ b/drivers/gpu/drm/panel/panel-simple.c
```

Send Feedback!

```
@@ -2053,6 +2053,31 @@ static const struct panel_desc
winstar_wf35ltiacd = {
    .bus_format = MEDIA_BUS_FMT_RGB888_1X24,
 };

+static const struct drm_display_mode custom_display_mode = {
+   .clock = 6410,
+   .hdisplay = 320,
+   .hsync_start = 320 + 20,
+   .hsync_end = 320 + 20 + 30,
+   .htotal = 320 + 20 + 30 + 38,
+   .vdisplay = 240,
+   .vsync_start = 240 + 4,
+   .vsync_end = 240 + 4 + 3,
+   .vtotal = 240 + 4 + 3 + 15,
+   .vscan = 60,
+   .flags = DRM_MODE_FLAG_NVSYNC | DRM_MODE_FLAG_NHSYNC,
+};
+
+static const struct panel_desc custom_display = {
+   .modes = &custom_display_mode,
+   .num_modes = 1,
+   .bpc = 8,
+   .size = {
+       .width = 80,
+       .height = 63,
+   },
+   .bus_format = MEDIA_BUS_FMT_RGB888_1X24,
+};
+
 static const struct of_device_id platform_of_match[] = {
    {
        .compatible = "ampire,am-480272h3tmqw-t01h",
@@ -2271,6 +2296,9 @@ static const struct of_device_id
platform_of_match[] = {
        .compatible = "winstar,wf35ltiacd",
        .data = &winstar_wf35ltiacd,
    }, {
+       .compatible = "custom,display",
+       .data = &custom_display,
+   }, {
        /* sentinel */
    }
 };
```

Send Feedback!

You can use any kernel configuration as defconfig. Because we prepend our file path to `FILESEXTRAPAHTS` the one from meta customer will be used, but we have to add it to the `SRC_URI` in our bbappend. On some Toradex BSP versions, we use in-tree defconfigs, so unsetting `KBUILD_DEFCONFIG` prevents that configuration from taking precedence over our custom one.

```
recipes-kernel/linux/linux-toradex/defconfig
```

```
#
# Automatically generated file; DO NOT EDIT.
# Linux/arm64 5.15.40 Kernel Configuration
#
CONFIG_CC_VERSION_TEXT="aarch64-tdx-linux-gcc (GCC) 11.3.0"
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=110300
CONFIG_CLANG_VERSION=0
CONFIG_AS_IS_GNU=y
CONFIG_AS_VERSION=20244508
CONFIG_LD_IS_BFD=y
CONFIG_LD_VERSION=20244508
CONFIG_LLD_VERSION=0
CONFIG_CC_HAS_ASM_GOTO=y
CONFIG_CC_HAS_ASM_GOTO_OUTPUT=y
CONFIG_CC_HAS_ASM_INLINE=y
CONFIG_CC_HAS_NO_PROFILE_FN_ATTR=y
CONFIG_IRQ_WORK=y
CONFIG_BUILDTIME_TABLE_SORT=y
CONFIG_THREAD_INFO_IN_TASK=y

#
# General setup
#
CONFIG_INIT_ENV_ARG_LIMIT=32
....
```

For the device tree file, we added a copy operation in the recipe. In theory, we could also add the device tree file with a patch. Both methods work equally well.

```
recipes-kernel/linux/linux-toradex/my-custom-devicetree-
file.dts
```

```
/dts-v1/;
```

Send Feedback!

```
#include <dt-bindings/pwm/pwm.h>
#include "imx8mp-verdin-wifi-dev.dts"

/ {
    model = "Customer Carrier Board with Toradex Verdin
iMX8MP";
    compatible = "customer,verdin_imx8mp",
                 "toradex,verdin-imx8mp-wifi",
                 "toradex,verdin_imx8mp",
                 "fsl,imx8mp";
};

&pwm2 {
    // We need to disable pwm2 so that we can use GPIO1 IO11
    status = "disabled";
};

&iomuxc {
    // The additonal gpio is not used by any node in the
devicetree
    // therefore we define it here so that the pins are
configured when
    // the iomux driver is loaded
    pinctrl-0 = <&pinctrl_csi_gpio_1 &pinctrl_csi_gpio_2
&additonal_gpio>;

    additonal_gpio: additonal-gpio {

        fsl,pins = <
            MX8MP_IOMUXC_GPIO1_IO11__GPIO1_IO11 0x0
        >;

    };
};

&panel {
    compatible = "custom,display";
};
```

We also need to add the device tree file to the variable KERNEL_DEVICETREE which defines which devicetrees are built and included into the final image.

```
meta-customer/conf/machine/verdin-imx8mp-extra.conf
```

Send Feedback!

```
KERNEL_DEVICETREE:append = " freescale/my-custom-devicetree-
file.dtb"
```

By default this file is not included anywhere. Therefore, we must make sure that it is included by another conf file (e.g. layer.conf). We can do that by adding the following line to conf/layer.conf:

```
include conf/machine/verdin-imx8mp-extra.conf
```

In a last step we need to tell U-Boot to load my-custom-devicetree-file.dtb instead of the standard one. For this we need to replace the original name with the new name by adding a bbappend file again. Create a directory recipes-bsp/u-boot:

```
$ cd ../layers/meta-customer/
$ mkdir -p recipes-bsp/u-boot
```

recipes-bsp/u-boot/u-boot-toradex_%.bbappend

```
do_configure:append() {
    # Remove exisiting fdtfile, if there is one
    sed -i '/"fdtfile=.*\\0" \\/d'
${S}/include/configs/verdin-imx8mp.h
    # Add new fdtfile, "my-custom-devicetree.dtb" should be
replaced with your device tree binary file
    sed -i 's/\("fdt_board=.*\\0" \\\)/\0\n      "fdtfile=my-
custom-devicetree.dtb\\0" \\/' ${S}/include/configs/verdin-
imx8mp.h
}
```

INFO

There are many ways to customize a kernel. If a lot of changes are added to the kernel then it is recommended to maintain your own git repository. In that case you need to change the SRC_URI and the SRCREV within the bbappend file. If only a tiny change is necessary you can also patch the default devicetree file and just leave everything else untouched.

Send Feedback!

Compile a Custom Kernel Module

While it is always preferable to work with sources integrated into the Linux kernel, there may be some reasons you might want to build a custom kernel module outside the kernel source code tree, for example, if the code is not GPLv2 compatible.

Let's say you have a "Hello World" kernel module:

hello.c

```
#include <linux/module.h>

int init_module(void)
{
    printk("Hello World!\n");
    return 0;
}

void cleanup_module(void)
{
    printk("Goodbye Cruel World!\n");
}

MODULE_LICENSE("GPL");
```

And a simple `Makefile` to build this module:

Makefile

```
obj-m := hello.o

SRC := $(shell pwd)

all:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC)

modules_install:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install

clean:
    rm -f *.o *~ core .depend .*.cmd *.ko *.mod.c
```

Send Feedback!

```
    rm -f Module.markers Module.symvers modules.order
    rm -rf .tmp_versions Modules.symvers
```

You can go to your custom layer and create a recipe to build this module:

```
$ cd ../layers/meta-customer/
$ mkdir -p recipes-kernel/hello-mod/
```

In the recipe, you just need to define the location of the kernel module source code, and inherit `module.bbclass` to build the kernel module:

recipes-kernel/hello-mod/hello-mod_1.0.bb

```
SUMMARY = "Example of how to build an external Linux kernel
module"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM =
"file://COPYING;md5=b234ee4d69f5fce4486a80fdaf4a4263"

inherit module

SRCREV = "4f082b755fdf2ef8da30adf2dfbca6fc0745cd2f"
SRC_URI = " \
    git://github.com/toradex/hello-
mod.git;branch=main;protocol=https \
"

PV = "1.0+git${SRCPV}"

S = "${WORKDIR}/git"

RPROVIDES_${PN} += "kernel-module-hello"
```

To include the kernel module in your images, you can add the following line to your machine configuration file:

```
MACHINE_EXTRA_RDEPENDS += " hello-mod"
```

INFO

Send Feedback!

Depending on the build system used by the module sources, you might need to make some adjustments in the recipe, like overriding the `do_compile` task or patching/adjusting the `Makefile`. Check the Yocto Project documentation for more information.

Create an Image

Toradex provides some reference images which can be built on their own or then directly be installed with the Toradex Easy Installer. But these images are considered reference images and should not be used in products directly, instead one should derive a custom image from them.

Using the meta layer created above, we create a recipe folder for our custom images:

```
$ cd ../layers/meta-customer/
$ mkdir -p recipes-images/images/
```

In that folder, we can create recipes and include-files to create one or multiple images. E.g. we can create a simple console image similar to the console-tdx-image containing the hello-world from the chapter Create a recipe. For that we create the `recipes-images/images/custom-console-image.bb`:

```
recipes-images/images/custom-console-image.bb
```

```
SUMMARY = "My Custom Image"
DESCRIPTION = "This is my customized image containing a simple
hello-world"

LICENSE = "MIT"

inherit core-image

#start of the resulting deployable tarball name
export IMAGE_BASENAME = "Custom-Console-Image"
MACHINE_NAME ?= "${MACHINE}"
IMAGE_NAME = "${MACHINE_NAME}_${IMAGE_BASENAME}"

SYSTEMD_DEFAULT_TARGET = "graphical.target"

IMAGE_LINGUAS = "en-us"
```

Send Feedback!

```
ROOTFS_PKGMANAGE_PKGS ?=
'${@oe.utils.conditional("ONLINE_PACKAGE_MANAGEMENT", "none",
"", "${ROOTFS_PKGMANAGE}", d)}'

IMAGE_INSTALL:append = " \
    packagegroup-boot \
    packagegroup-basic \
    udev-extra-rules \
    ${ROOTFS_PKGMANAGE_PKGS} \
    weston weston-init wayland-terminal-launch \
    hello-world \
"

IMAGE_DEV_MANAGER   = "udev"
IMAGE_INIT_MANAGER  = "systemd"
IMAGE_INITSCRIPTS   = " "
IMAGE_LOGIN_MANAGER = "busybox shadow"
```

Finally, we can build the image with:

```
$ cd build/
$ bitbake custom-console-image
```

Edit this page

Send Feedback!