
How to Use Yocto Project to Create a Custom Linux Image

STM32MP157F-DK2 DEVELOPMENT KIT

darkquesth
github.com/darkquesth

September 2022

Contents

| | | |
|----------|--|-----------|
| 1 | Creating a Custom Linux Image | 2 |
| | Tutorials | 2 |
| | Required Hardware | 2 |
| | Required Software | 2 |
| | 1.1 Install Dependencies | 3 |
| | 1.2 Download Layers | 3 |
| | 1.3 Configure Build | 4 |
| | 1.4 Configure Kernel | 5 |
| | 1.5 Build Image | 6 |
| 2 | Final Image for SD Card | 8 |
| | 2.1 Creating the Final Image | 8 |
| | 2.2 Flashing the Image to SD Card | 8 |
| 3 | Testing the Image | 11 |
| | 3.1 Boot into Linux | 11 |
| 4 | Adding Build Tools | 12 |
| 5 | Creating Custom Layer and Image | 13 |
| | 5.1 Default Image Recipes | 13 |
| | 5.2 Create Custom Layer | 13 |
| | 5.3 Create Custom Image | 14 |
| | 5.4 Add Layer to Build Process | 15 |
| 6 | Device Tree Patches | 17 |
| | 6.1 Create Device Tree Patch | 18 |
| | 6.2 Enable I ² C and FDCAN | 19 |
| | 6.3 Applying Patch to Device Tree | 20 |
| | 6.4 Enable i2cdetect and can-utils | 21 |
| 7 | Building Custom Image | 24 |
| | 7.1 Build and Flash the Custom Image | 24 |
| | 7.2 Testing I ² C and FDCAN | 24 |
| | 7.2.1 I ² C Tools and Sensor Connection | 24 |
| | 7.2.2 FDCAN Initialisation and Loopback Test | 25 |
| | Further Reading and Resources | 26 |

Creating a Custom Linux Image

Tutorials

- <https://www.youtube.com/playlist?list=PLEBQazB0HUyTpoJoZecRK6PpDG31Y7RPB>
- <https://www.digikey.com/en/maker/projects/intro-to-embedded-linux-part-2-yocto-project/2c08a1ad09d74f20b9844e566d332da4>
- <https://www.cocoacrumbs.com/blog/2021-10-15-building-a-linux-distro-for-the-stm32mp157f-dk2/>
- <https://bootlin.com/blog/building-a-linux-system-for-the-stm32mp1-basic-system/>

Required Hardware

- STM32MP157F-DK2 Development Kit
- USB-C to USB-A power cable
- Micro USB to USB-A cable for connecting to the board via terminal
- A minimum of 8 GB SD card
- At least 4 GB of RAM on your host PC

Required Software

You will need Linux for this project, as all of the tools we are using must be run in Linux. However, the steps shown below are tested in Ubuntu 20.04 using dual-boot. While you can set up a virtual Linux machine (e.g. Oracle VM VirtualBox) to build a Linux image, it would take way longer time than installing Ubuntu natively on your PC. You may follow the tutorials given below for installing Ubuntu on a virtual machine or alongside another OS (like Windows).

Dual-boot (*recommended*): <https://www.tecmint.com/install-ubuntu-alongside-with-windows-dual-boot>

Virtualbox: <https://itsfoss.com/install-linux-in-virtualbox>

Also, you should allocate more than 70 GB disk space to download and build source files

1.1 Install Dependencies

Boot into Ubuntu on your host computer and run the following commands.

```
sudo apt update
sudo apt upgrade
sudo apt install -y bc build-essential chrpath cpio diffstat gawk git texinfo
wget gdisk python3 python3-pip gedit nano
sudo apt install -y libssl-dev
```

Because the Yocto Project tools rely on the "python" command, you will likely need to alias "python" to "python3". Edit your .bashrc file:

```
gedit ~/.bashrc
```

Scroll to the bottom and add the following to a new line

```
alias python=python3
```

Save and exit ('esc' followed by entering ":wq"). Re-run the .bashrc script to update your shell:

```
source ~/.bashrc
```

1.2 Download Layers

Download the Yocto Project poky reference distribution:

```
mkdir -p ~/Projects/yocto
cd ~/Projects/yocto
git clone https://git.yoctoproject.org/poky.git
cd poky
```

You will want your poky layer branch to match the branches of all other third-party layers you download (such as the STM32MP1 BSP). You can view the available poky release names here: <https://wiki.yoctoproject.org/wiki/Releases>. We are going to have all layers be on the "kirkstone" branch.

```
git checkout kirkstone
```

Next, you will want to download the STM32MP board support package (BSP) as a separate layer:

```
cd ~/Projects/yocto
git clone https://github.com/STMicroelectronics/meta-st-stm32mp
cd meta-st-stm32mp
git checkout kirkstone
```

View the readme to see what other layers are needed for this particular BSP:

```
less README
```

In there, you can see that we need the meta-openembedded layer. Specifically, we need the meta-oe and meta-python layers in the meta-openembedded layer. Also, install any dependencies mentioned in that file.

```
cd ~/Projects/yocto
git clone https://github.com/openembedded/meta-openembedded.git
cd meta-openembedded
git checkout kirkstone
```

1.3 Configure Build

To start using bitbake you need to source the "oe-init-build-env" script located into poky/ directory. So you should do something like this every time using bitbake:

```
cd ~/Projects/yocto
source poky/oe-init-build-env build-mp1
```

You can view the layers that will be included in the build with the following:

```
bitbake-layers show-layers
```

You should only have the default poky layers to start. We need to edit bblayers.conf in our build to add the necessary STM32MP BSP and dependency layers:

```
gedit conf/bblayers.conf
```

Update the BBLAYERS variable to be the following (change <username> to your actual username):

```
BBLAYERS ?= " \
/home/<username>/Projects/yocto/poky/meta \
/home/<username>/Projects/yocto/poky/meta-poky \
/home/<username>/Projects/yocto/poky/meta-yocto-bsp \
/home/<username>/Projects/yocto/meta-openembedded/meta-oe \
/home/<username>/Projects/yocto/meta-openembedded/meta-python \
/home/<username>/Projects/yocto/meta-st-stm32mp \
"
```

Save and exit. Check the layers again with:

```
bitbake-layers show-layers
```

You can view the available machine names and settings in `~/Projects/yocto/meta-st-stm32mp/conf/machine`. We will use "stm32mp1" for our build. To do this, edit local.conf:

```
gedit conf/local.conf
```

Change the MACHINE variable to the following (comment out the "qemu" emulator and add "stm32mp1" as the machine):

```
#MACHINE ??= "qemux86-64"
MACHINE = "stm32mp1"
```

1.4 Configure Kernel

Now that your build system is set up, you can make changes to the kernel. To do that, enter:

```
bitbake -c menuconfig virtual/kernel
```

Note that the first time you run bitbake for a particular build, it will take some time parsing all the required metadata. This could take 15 minutes or more, depending on your host computer, so be patient.

After it finishes, you should be presented with a menu:

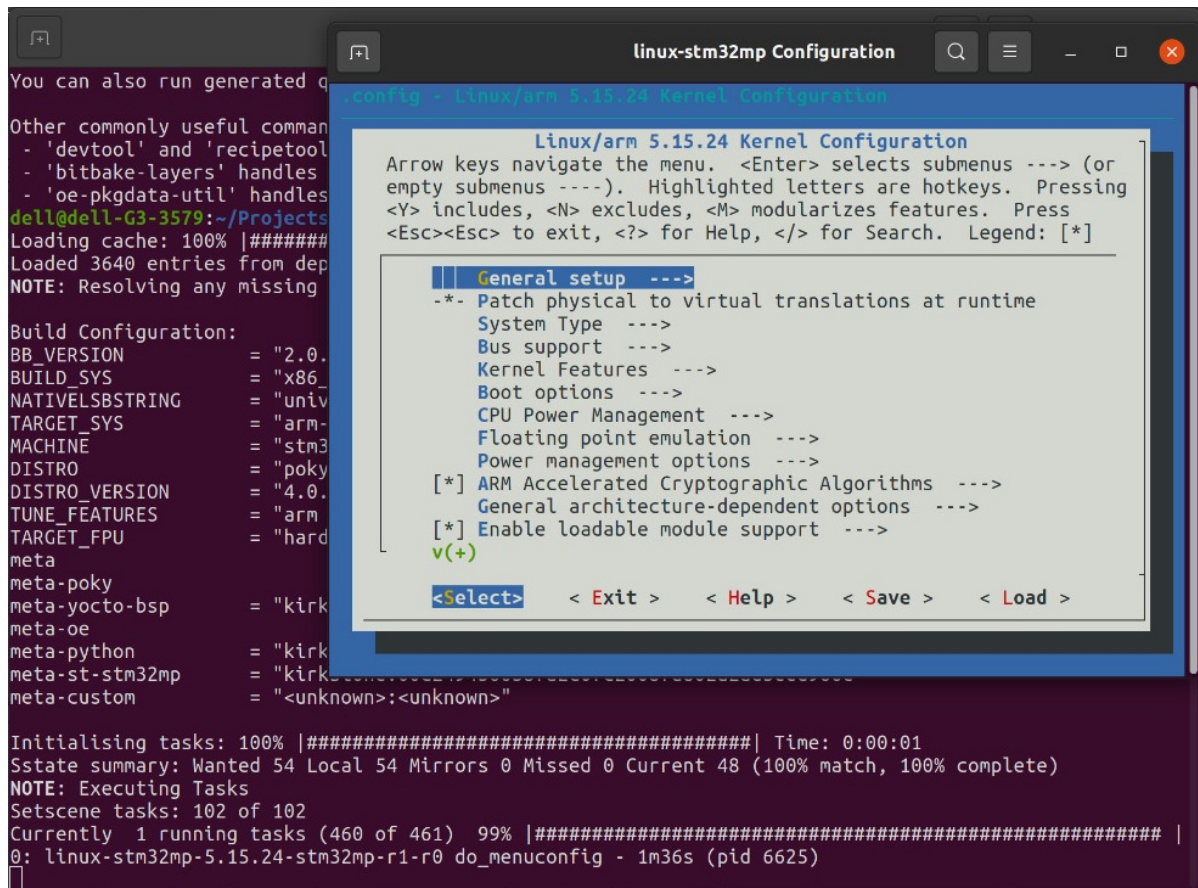


Figure 1.1: Configuring the kernel

You can change various kernel settings. However, we will leave everything at their defaults for now, so just select Exit and press 'enter'.

To make kernel changes permanent whenever you modify the kernel, you should run:

```
bitbake -c savedefconfig virtual/kernel
```

1.5 Build Image

Now, it is time to actually build your image! We need to choose an image to build from various images that are supported by the default poky installation here: <https://docs.yoctoproject.org/ref-manual/images.html>

For now, we won't need any extra packages, so we'll skip adding the OpenSTLinux layer and just focus on the bare minimum, which is provided by the core-image-minimal image.

Once you have everything configured to your liking, just run the following command:

```
bitbake core-image-minimal
```

The first time you build an image with bitbake, it will likely take many hours (the first build might take around 2-3 hours). One big advantage of the Yocto Project is that it builds every-

thing in stages and layers. If you make any changes (e.g. add a layer, change to a different image, tweak kernel settings), subsequent builds will take far less time. This speeds up development process when you are trying to add low-level support in Linux.

If you want to start over (e.g. you press 'ctrl + c' or something gets corrupted/tainted)

```
bitbake -c cleanall core-image-minimal
```

To clean out everything

```
rm -rf tmp
```

Once building is complete without any errors, you can find all of the output images in the deploy folder:

```
ls tmp/deploy/images/stm32mp1
```


Final Image for SD Card

2.1 Creating the Final Image

Finally, we can begin deploying the final image and flash it to an SD card for booting the system. In most cases, booting into Linux requires several bootloader programs to run in sequence. This is known as a "boot chain" or "boot sequence". For embedded Linux, this process will often look something like this:

ROM > First Stage Bootloader (FSBL) > Second Stage Bootloader (SSBL) > Kernel

While it is possible to format partitions manually, there is a script that does it automatically. First, stand in the following directory:

```
cd ~/Projects/yocto/build-mp1/tmp/deploy/images/stm32mp1/scripts
```

A *create_sdcard_from_flashlayout.sh* script can be found in that folder. Let us choose *FlashLayout_sdcard_stm32mp157f-dk2-extensible.tsv* to create the image by executing this command line:

```
./create_sdcard_from_flashlayout.sh ../flashlayout_core-image-minimal/extensib_
le/FlashLayout_sdcard_stm32mp157f-dk2-extensible.tsv
```

2.2 Flashing the Image to SD Card

ST recommends using their STM32CubeProgrammer to flash the SD card. However, we will do things manually so you can get an idea of how to configure an SD card with the various image files.

Navigate to the output directory for your images:

```
cd ~/Projects/yocto/build-mp1/tmp/deploy/images/stm32mp1
```

From here, you can figure out which image files ST would use to flash an SD card by looking at the *flashlayout_core-image-minimal/trusted/FlashLayout_sdcard_stm32mp157f-dk2-trusted.tsv* file in a text editor.

```
gedit flashlayout_core-image-minimal/trusted/FlashLayout_sdcard_stm32mp157f-dk
2-trusted.tsv
```

| #Opt | Id | Name | Type | IP | Offset | Binary |
|------|------|------------|------------|------|------------|--|
| 1 | 0x01 | fsbl-boot | Binary | none | 0x0 | arm-trusted-firmware/tf-a-stm32mp157f-dk2-usb.stm32 |
| 2 | 0x03 | fip-boot | FIP | none | 0x0 | fip/fip-stm32mp157f-dk2-trusted.bin |
| 3 | 0x04 | fsbl1 | Binary | mmc0 | 0x00004400 | arm-trusted-firmware/tf-a-stm32mp157f-dk2-sdcard.stm32 |
| 4 | 0x05 | fsbl2 | Binary | mmc0 | 0x00044400 | arm-trusted-firmware/tf-a-stm32mp157f-dk2-sdcard.stm32 |
| 5 | 0x06 | metadata1 | Binary | mmc0 | 0x00084400 | arm-trusted-firmware/metadata.bin |
| 6 | 0x07 | metadata2 | Binary | mmc0 | 0x000C4400 | arm-trusted-firmware/metadata.bin |
| 7 | 0x08 | fip-a | FIP | mmc0 | 0x00104400 | fip/fip-stm32mp157f-dk2-trusted.bin |
| 8 | 0x09 | fip-b | FIP | mmc0 | 0x00504400 | none |
| 9 | 0x0A | u-boot-env | Binary | mmc0 | 0x00904400 | none |
| 10 | 0x10 | bootfs | System | mmc0 | 0x00984400 | st-image-bootfs-poky-stm32mp1.ext4 |
| 11 | 0x11 | vendorfs | FileSystem | mmc0 | 0x04984400 | st-image-vendorfs-poky-stm32mp1.ext4 |
| 12 | 0x12 | rootfs | FileSystem | mmc0 | 0x05984400 | core-image-minimal-stm32mp1.ext4 |
| 13 | 0x13 | userfs | FileSystem | mmc0 | 0x33984400 | st-image-userfs-poky-stm32mp1.ext4 |

Figure 2.1: Flashing the final image

This will show you the name of the image files to use for the FSBL, metadata, SSBL, bootfs, vendorfs, rootfs and userfs.

To flash the final image, plug your SD card into your host computer and check where it is mounted. We can use `sudo fdisk -l` or `lsblk` command for that.

In my case, the SD card is mounted at `/dev/mmcblk0`.

Make sure to unmount any partitions that were automounted when you plugged in the SD card. (Using `sudo umount /dev/mmcblk0` or through GUI like gparted or file system).

As we will be writing to the SD card, any previous data and partitions should be formatted. To do that run this line:

```
sudo fdisk /dev/mmcblk0
```

```

dell@dell-G3-3579: ~/Projects/yocto/build-mp1/tmp/deploy/imag...
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 97FE5981-927B-436F-9A40-B612C8D4088E

Device            Start      End  Sectors  Size Type
/dev/mmcblk0p1      34         545     512    256K Linux reserved
/dev/mmcblk0p2     546       1057     512    256K Linux reserved
/dev/mmcblk0p3     1058       1569     512    256K Linux reserved
/dev/mmcblk0p4     1570       2081     512    256K Linux reserved
/dev/mmcblk0p5     2082      10273    8192     4M unknown
/dev/mmcblk0p6     10274      18465    8192     4M unknown
/dev/mmcblk0p7     18466      19489    1024    512K Linux reserved
/dev/mmcblk0p8     19490     150561   131072   64M Linux filesystem
/dev/mmcblk0p9     150562     183329   32768   16M Linux filesystem
/dev/mmcblk0p10    183330    1690657  1507328 736M Linux filesystem
/dev/mmcblk0p11    1690658   31116252 29425595 14G Linux filesystem
dell@dell-G3-3579:~/Projects/yocto/build-mp1/tmp/deploy/images/stm32mp1$ sudo fdisk
/dev/mmcblk0

Welcome to fdisk (util-linux 2.34).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Command (m for help):

```

Figure 2.2: Partitions shown in fdisk

In fdisk, perform the following actions:

- 'p' to view the partitions
- 'd' to delete a partition
 - Select one of the partitions
 - Repeat this process until all partitions have been deleted
- If your SD card is not a GPT layout, you will need to change it to GPT.
 - Type 'p' and look at "Disklabel type." It should say "gpt"
 - 'g' to change the layout to GPT
- 'w' to write changes to the SD card and exit

You can confirm that the changes were made by entering the `lsblk` command again.

Now we can flash the SD card:

```
sudo dd if=../flashlayout_core-image-minimal/extensible/../../FlashLayout_sdcard_stm32mp157f-dk2-extensible.raw of=/dev/mmcblk0 bs=8M conv=fdatasync status=progress oflag=direct
```

```
Populate raw image with image content:
part 01: fsbl1, image: arm-trusted-firmware/tf-a-stm32mp157f-dk2-sdcard.stm32 [ FILLED ] part 01:
fsbl1, image: arm-trusted-firmware/tf-a-stm32mp157f-dk2-sdcard.stm32
part 02: fsbl2, image: arm-trusted-firmware/tf-a-stm32mp157f-dk2-sdcard.stm32 [ FILLED ] part 02:
fsbl2, image: arm-trusted-firmware/tf-a-stm32mp157f-dk2-sdcard.stm32
[ FILLED ] part 03: metadata1, image: arm-trusted-firmware/metadata.bin
[ FILLED ] part 04: metadata2, image: arm-trusted-firmware/metadata.bin
[ FILLED ] part 05: fip-a, image: fip/fip-stm32mp157f-dk2-optee.bin
[ FILLED ] part 08: bootfs, image: st-image-bootfs-poky-stm32mp1.ext4
[ FILLED ] part 09: vendorfs, image: st-image-vendorfs-poky-stm32mp1.ext4
[ FILLED ] part 10: rootfs, image: custom-image-stm32mp1.ext4

#####
#####

RAW IMAGE generated: ../flashlayout_custom-image/extensible/../../FlashLayout_sdcard_stm32mp157f-dk2-extensible.raw

WARNING: before to use the command dd, please umount all the partitions
associated to SDCARD.
sudo umount `lsblk --list | grep mmcblk0 | grep part | gawk '{ print $7 }' | tr '\n' ' '`

To put this raw image on sdcard:
sudo dd if=../flashlayout_custom-image/extensible/../../FlashLayout_sdcard_stm32mp157f-dk2-extensible.raw of=/dev/mmcblk0 bs=8M conv=fdatasync status=progress

(mmcblk0 can be replaced by:
sdX if it's a device dedicated to receive the raw image
(where X can be a, b, c, d, e)

#####
#####

dell@dell-G3-3579:~/Projects/yocto/build-mp1/tmp/deploy/images/stm32mp1/scripts$ sudo dd if=../flashlayout_custom-image/extensible/../../FlashLayout_sdcard_stm32mp157f-dk2-extensible.raw of=/dev/mmcblk0 bs=8M conv=fdatasync status=progress oflag=direct
[sudo] password for dell:
1610612736 bytes (1,6 GB, 1,5 GiB) copied, 213 s, 7,5 MB/s
192+0 records in
192+0 records out
```

Figure 2.3: The partitions are created automatically

Testing the Image

3.1 Boot into Linux

Plug the SD card into the STM32MP157F-DK2 board. Connect a USB micro cable from your host computer to the ST-LINK (CN11) port on the board. On your host computer, enter the following:

```
sudo apt install picocom
sudo picocom -b 115200 /dev/ttyACM1
```

`/ttyACM1` part might be different on your host PC, so you could also try `/ttyACM0`. If you wish to exit picocom, press `[Ctrl][A]` followed by `[Ctrl][X]`.

If everything went well, you should see the FSBL (TF-A) post a few lines to the console followed by the SSBL (U-Boot). U-Boot will launch the kernel, and after a few seconds, you should be presented with a login prompt. Enter "root" (no password) to gain access to Linux.

```
$ picocom -b 115200 /dev/ttyACM1
picocom v3.1

port is      : /dev/ttyACM1
...
NOTICE: CPU: STM32MP157FAC Rev.Z
NOTICE: Model: STMicroelectronics STM32MP157F-DK2 Discovery Board
NOTICE: Board: MB1272 Var4.0 Rev.C-02
INFO: Reset reason (0x15):
INFO:   Power-on Reset (rst_por)
INFO: PMIC version = 0x20
INFO: FCONF: Reading TB FW firmware configuration file from: 0x2ffe3000
INFO: FCONF: Reading firmware configuration information for: stm32mp_io
INFO: Using SDMMC
INFO:   Instance 1
INFO:   Boot used partition fsbl1
...

U-Boot 2020.10-stm32mp-r1 (Oct 05 2020 - 15:15:32 +0000)

CPU: STM32MP157FAC Rev.Z
Model: STMicroelectronics STM32MP157F-DK2 Discovery Board
Board: stm32mp1 in trusted mode (st,stm32mp157f-dk2)
Board: MB1272 Var4.0 Rev.C-02
DRAM:  512 MiB
Clocks:
- MPU : 800 MHz
- MCU : 208.878 MHz
- AXI : 266.500 MHz
- PER : 24 MHz
- DDR : 533 MHz
WDT:   Started with servicing (32s timeout)
NAND:  0 MiB
MMC:   STM32 SD/MMC: 0, STM32 SD/MMC: 1
...
Hit any key to stop autoboot:  0
Boot over mmc0!
...
Select the boot mode
1: OpenSTLinux
2:  stm32mp157f-dk2-a7-examples
3:  stm32mp157f-dk2-m4-examples
Enter choice: 1:  OpenSTLinux
Retrieving file: /uImage
...
Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
...
Poky (Yocto Project Reference Distro) 4.0.3 stm32mp1

stm32mp1 login: root
root@stm32mp1:~#
```

Figure 3.1: Successfully booted into Linux

Adding Build Tools

The core-image-minimal system is installed with minimal packages, as suggested by its name. Although it does not have any basic tools such as apt, gcc or nano package; we can add them by configuring our build.

First, navigate to this directory and edit *local.conf*:

```
cd ~/Projects/yocto/build-mp1/conf
gedit local.conf
```

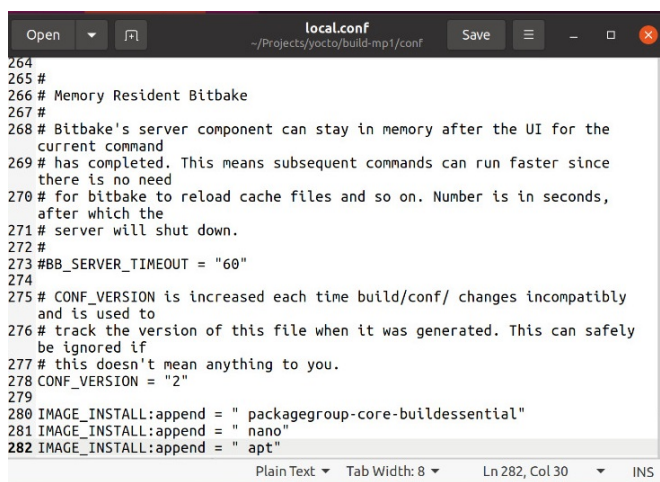


Figure 4.1: Adding tools in local.conf

Add the following lines:

```
IMAGE_INSTALL:append = " packagegroup-core-buildessential"
IMAGE_INSTALL:append = " nano"
IMAGE_INSTALL:append = " apt"
```

The first line will append the typical GCC build tools. The second line will add the Nano editor (in case you prefer not to work with the VI editor that is part of the minimal Linux system).

Also, pay attention to syntax changes in newer versions while changing build configurations:

<https://docs.yoctoproject.org/next/migration-guides/migration-3.4.html>

Now rebuild the Linux system and flash it to an SD Card (you can follow 1.5 and 2.1). If all went well, you can now use GCC and the Nano editor natively on your STM32MP157F-DK2 board.

Creating Custom Layer and Image

Until now, we built our Linux image, flashed it onto an SD card and boot into Linux on the STM32MP157F-DK2 board. In this chapter, we will walk through the process of creating your own layer in the Yocto project and using it to make changes to the Linux image. Specifically, we will expand the rootfs size (to give you more space for modules, packages, and applications).

5.1 Default Image Recipes

The poky reference distribution comes with a main image recipe that is used during the bitbake build process in order to construct the Linux image. Up until now, we have been working with `core-image-minimal` as our target image. `core-image-minimal` inherits the `core-image` class recipe, which can be found here

```
cd ~/Projects/yocto/poky/meta
gedit classes/core-image.bbclass
```

Note that it is a class recipe (`.bbclass`), which acts as a template for other recipes to import (or "inherit"). It assigns various packagegroups to `IMAGE_INSTALL`, which is an important variable used to tell bitbake what things to include in our image (e.g. what modules and packages to include).

Look at the `core-image-minimal` recipe to see what was being included in our previous builds:

```
gedit recipes-core/images/core-image-minimal.bb
```

A screenshot of a code editor window titled "core-image-minimal.bb" with a file path of "~/Projects/yocto/poky/meta/recipes-core/ima...". The editor shows the following content:

```
1 SUMMARY = "A small image just capable of allowing a device to boot."
2
3 IMAGE_INSTALL = "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL}"
4
5 IMAGE_LINGUAS = " "
6
7 LICENSE = "MIT"
8
9 inherit core-image
10
11 IMAGE_ROOTFS_SIZE ?= "8192"
12 IMAGE_ROOTFS_EXTRA_SPACE:append = "${-
  {@bb.utils.contains("DISTRO_FEATURES", "systemd", " + 4096", "", d)}}
```

Figure 5.1: `core-image-minimal.bb`

5.2 Create Custom Layer

One of the main advantages of the Yocto Project is its ability to pull in source material from a variety of places (git repositories, websites, local files, etc.). Most of this is accomplished by

keeping files (e.g. metadata, recipes) in "layers." We previously downloaded the poky and meta-stm32mp board support package (BSP) layers.

By creating your own layer, you can easily keep it under version control (e.g. git) so that you can easily configure custom images for future builds. This is especially important if you are creating a product and want to reproduce the full image at any time during production. All you need to do is create a build directory, include your layer (along with other required layers, such as poky and the BSP), and call "bitbake <name-of-image>".

Start by enabling the OpenEmbedded build environment again:

```
cd ~/Projects/yocto
source poky/oe-init-build-env build-mp1
```

Then, create a custom layer that sits at the same directory level as our other layers:

```
cd ~/Projects/yocto
bitbake-layers create-layer meta-custom
```

The "bitbake-layers" tool automatically constructs the appropriate directory structure for our layer and gives us an example recipe in *../meta-custom/recipes-example/example/example_0.1.bb*. Feel free to open and look at that example.

5.3 Create Custom Image

Instead of using core-image-minimal, we are going to write a recipe that builds a custom image. Start by creating the following directory structure:

```
cd meta-custom
mkdir -p recipes-core/images
```

Then, we shall copy the core-image-minimal.bb recipe to use a starting point for our custom recipe:

```
cp ../poky/meta/recipes-core/images/core-image-minimal.bb
  recipes-core/images/custom-image.bb
  gedit recipes-core/images/custom-image.bb
```

You may change the custom recipe to add a user and password; but we will leave it as it is for now.

5.4 Add Layer to Build Process

We need to add our custom layer to the build process. Do that with the following:

```
cd ../build-mp1/  
gedit conf/bblayers.conf
```

Add `"/home/<username>/Projects/yocto/meta-custom \"` to the BBLAYERS variable.

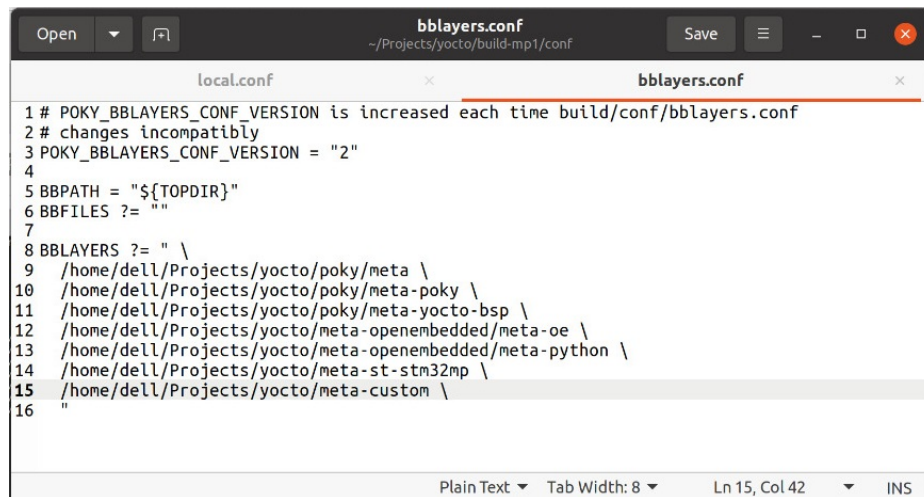


Figure 5.2: Adding custom layer

Next, we are going to include some features to our custom image.

```
cd ../build-mp1/  
gedit conf/local.conf
```

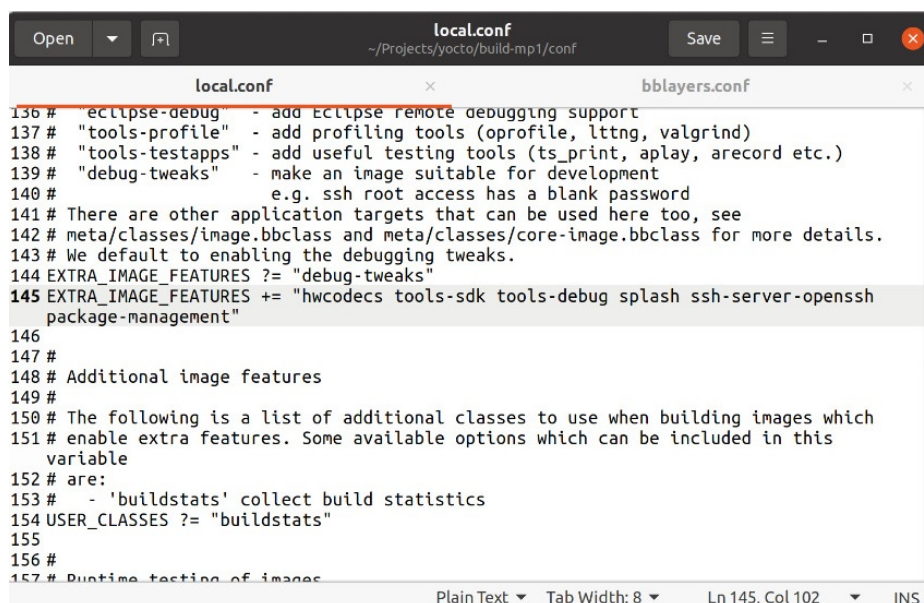
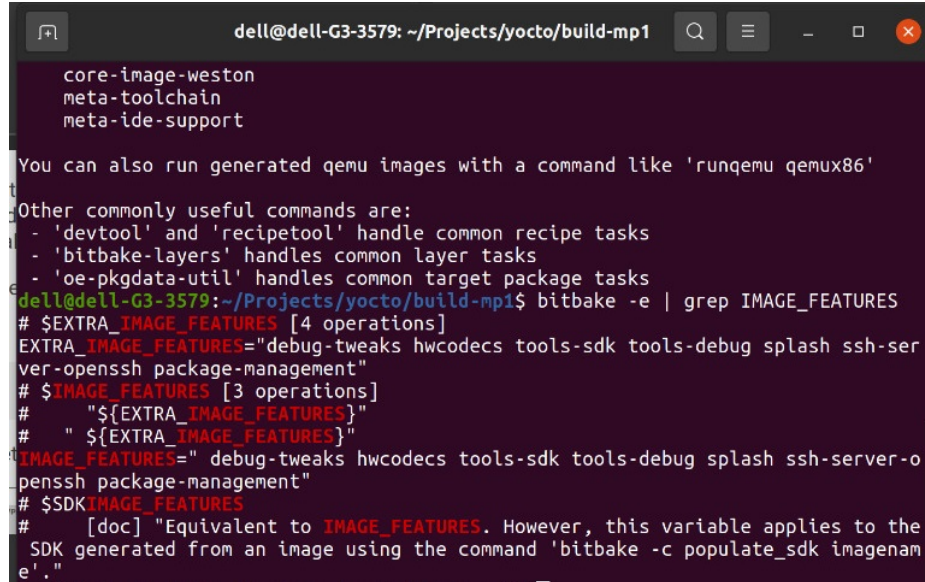


Figure 5.3: Extra image features

Make sure that *debug-tweaks* is enabled and append this line to the configuration file:

```
EXTRA_IMAGE_FEATURES += "hwcodecs tools-sdk tools-debug splash ssh-server-openssh  
package-management"
```

You can view the `IMAGE_FEATURES` variable with the following command:



```
dell@dell-G3-3579: ~/Projects/yocto/build-mp1
core-image-weston
meta-toolchain
meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'

Other commonly useful commands are:
- 'devtool' and 'recipetool' handle common recipe tasks
- 'bitbake-layers' handles common layer tasks
- 'oe-pkgdata-util' handles common target package tasks
dell@dell-G3-3579:~/Projects/yocto/build-mp1$ bitbake -e | grep IMAGE_FEATURES
# $EXTRA_IMAGE_FEATURES [4 operations]
EXTRA_IMAGE_FEATURES="debug-tweaks hwcodecs tools-sdk tools-debug splash ssh-ser
ver-openssh package-management"
# $IMAGE_FEATURES [3 operations]
#   "${EXTRA_IMAGE_FEATURES}"
#   "${EXTRA_IMAGE_FEATURES}"
IMAGE_FEATURES=" debug-tweaks hwcodecs tools-sdk tools-debug splash ssh-server-o
penssh package-management"
# $SDKIMAGE_FEATURES
#   [doc] "Equivalent to IMAGE_FEATURES. However, this variable applies to the
SDK generated from an image using the command 'bitbake -c populate_sdk imagenam
e'."
```

Figure 5.4: Showing which features are enabled

Moreover, you can check out other image features here: https://docs.yoctoproject.org/ref-manual/features.html?highlight=extra_image_features.

Device Tree Patches

If you look at the [datasheet for the STM32MP157D-DK1 development board](#), you can see that there are 6 I2C busses available. By default I2C ports 1 and 4 are enabled and used to control other components on the board. We want to enable port 5 (as it is broken out to the Raspberry Pi-style header on the board) and use it to communicate with a temperature sensor. SDA is on top header (CN2) pin 3 and SCL is on pin 5.

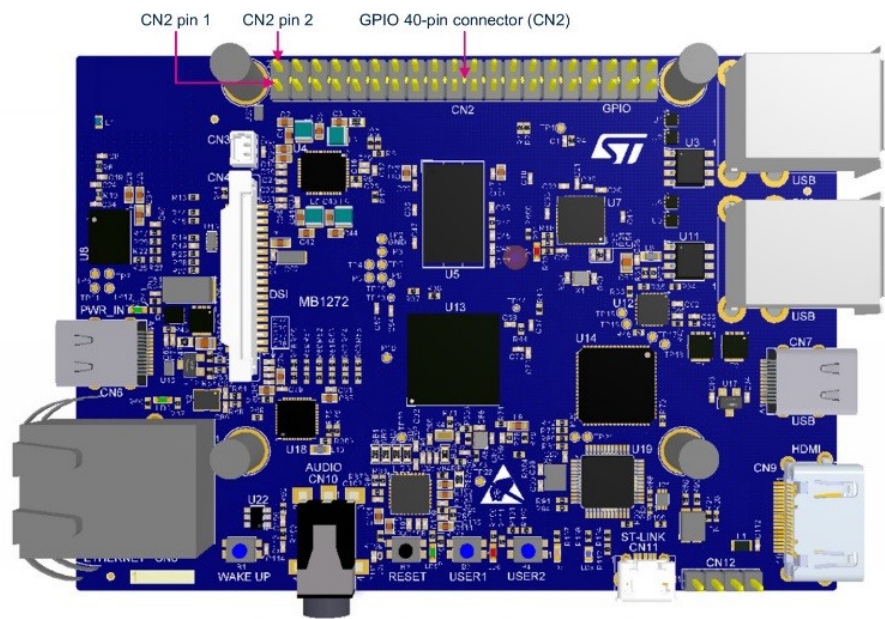


Table 25 describes the pinout of the GPIO connectors.

Table 25. GPIO connectors pinout

| Function | STM32 pin | Pin | Pin | STM32 pin | Function |
|------------------|-----------|-----|-----|-----------|----------|
| 3V3 | - | 1 | 2 | - | 5V |
| GPIO2 / I2C5_SDA | PA12 | 3 | 4 | - | 5V |
| GPIO3 / I2C5_SCL | PA11 | 5 | 6 | - | GND |

Figure 6.1: STM32MP157F-DK2 CN2 GPIO header pinout

If you look at the [STM32MP157 reference manual](#), you can see that I2C port 5 is controlled by registers starting at memory address 0x40015000.

Table 9. Register boundary addresses (continued)

| Bus | Boundary address | Size (Bytes) | Peripheral | Peripheral Register map |
|------|-------------------------|--------------|------------|---------------------------|
| APB1 | 0x4001C400 - 0x43FFFFFF | 65423 KB | Reserved | - |
| | 0x4001C000 - 0x4001C3FF | 1 KB | MDIOS | <i>MDIOS registers</i> |
| | 0x40019400 - 0x4001BFFF | 11 KB | Reserved | - |
| | 0x40019000 - 0x400193FF | 1 KB | UART8 | <i>USART registers</i> |
| | 0x40018400 - 0x40018FFF | 3 KB | Reserved | - |
| | 0x40018000 - 0x400183FF | 1 KB | UART7 | <i>USART registers</i> |
| | 0x40017400 - 0x40017FFF | 3 KB | Reserved | - |
| | 0x40017000 - 0x400173FF | 1 KB | DAC1 | <i>DAC registers</i> |
| | 0x40016400 - 0x40016FFF | 3 KB | Reserved | - |
| | 0x40016000 - 0x400163FF | 1 KB | CEC | <i>HDMI-CEC registers</i> |
| | 0x40015400 - 0x40015FFF | 3 KB | Reserved | - |
| | 0x40015000 - 0x400153FF | 1 KB | I2C5 | <i>I2C registers</i> |
| | 0x40014400 - 0x40014FFF | 3 KB | Reserved | - |
| | 0x40014000 - 0x400143FF | 1 KB | I2C3 | <i>I2C registers</i> |
| | 0x40013400 - 0x40013FFF | 3 KB | Reserved | - |
| | 0x40013000 - 0x400133FF | 1 KB | I2C2 | <i>I2C registers</i> |
| | 0x40012400 - 0x40012FFF | 3 KB | Reserved | - |
| | 0x40012000 - 0x400123FF | 1 KB | I2C1 | <i>I2C registers</i> |
| | 0x40011400 - 0x40011FFF | 3 KB | Reserved | - |
| | 0x40011000 - 0x400113FF | 1 KB | UART5 | <i>USART registers</i> |
| | 0x40010400 - 0x40010FFF | 3 KB | Reserved | - |

Figure 6.2: Memory addressing of I2C5 peripheral

6.1 Create Device Tree Patch

On your host computer, navigate to the build directory and copy the device tree source (.dts) file to a temporary working directory. Then, create a copy of the original. Open it to make changes. Feel free to look through [this guide](#) to learn more about device trees.

```
cd ~/Projects/yocto/build-mp1/
cp tmp/work-shared/stm32mp1/kernel-source/arch/arm/boot/dts/stm32mp157d-dk1.dts
~/Documents
cd ~/Documents
cp stm32mp157f-dk2.dts stm32mp157f-dk2.dts.orig
gedit stm32mp157f-dk2.dts
```

6.2 Enable I²C and FDCAN

At the bottom of the file, add the following device tree nodes in order to enable I2C5 and FDCAN1:

```
&i2c5 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_sleep_pins_a>;
    i2c-scl-rising-time-ns = <185>;
    i2c-scl-falling-time-ns = <20>;
    clock-frequency = <100000>;
    status = "okay";
};

&m_can1 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&m_can1_pins_a>;
    pinctrl-1 = <&m_can1_sleep_pins_a>;
    status = "okay";
};
```



Figure 6.3: I2C5 and FDCAN1 device tree patch

Save and exit. Then, create a diff patch. Note the "--no-index" argument allows us to perform

git diff on two different files that are not part of a git repository.

```
git diff --no-index stm32mp157f-dk2.dts.orig stm32mp157f-dk2.dts >
0001-add-i2c5-userspace-dts.patch
```

However, because bitbake expects such diff files to be part of a repository, we need to make a couple of manual changes to the file so that it will be applied to the correct file (in a particular directory structure in the working area of build-mp1/tmp/). Open the file with:

```
gedit 0001-add-i2c5-userspace-dts.patch
```

Change the file header so that it points to the correct file locations:

```
--- a/arch/arm/boot/dts/stm32mp157f-dk2.dts
+++ b/arch/arm/boot/dts/stm32mp157f-dk2.dts
```



Figure 6.4: Patch file

6.3 Applying Patch to Device Tree

Enable our build environment and navigate to the custom layer we created earlier:

```
cd ~/Projects/yocto/
source poky/oe-init-build-env build-mp1/
cd ../meta-custom/
```

Create a directory structure for kernel recipes. The naming is important! The patch file must be in "linux/<machine-name>."

```
mkdir -p recipes-kernel/linux/stm32mp1/
```

Now, we copy our patch file:

```
cp ~/Documents/0001-add-i2c5-userspace-dts.patch recipes-kernel/linux/stm32mp1/
```

You can run the following to discover the name of the kernel we are working with:

```
oe-pkgdata-util lookup-recipe kernel
```

It should say "linux-stm32mp".

In the linux/ directory, create a custom .bbappend file (which will be added to our main kernel recipe). Once again, the name is important! It must be "<kernel-name>_version.bbappend". We can use '%' as the version number to be a wildcard that matches any version of that file.

```
gedit recipes-kernel/linux/linux-stm32mp_%.bbappend
```

In this file, add the following lines, which tell the kernel recipe to look in "this directory" (the directory containing this .bbappend file) and apply the patch to the kernel.

```
FILESEXTRAPATHS_prepend := "${THISDIR}:"  
SRC_URI += "file://0001-add-i2c5-userspace-dts.patch"
```



Figure 6.5: Custom .bbappend file

6.4 Enable i2cdetect and can-utils

In order to send and receive CAN data, we need the *can-utils* package. First, navigate to the following directory:

```
cd ~/Projects/yocto/build-mp1/conf  
gedit local.conf
```

Then, append these lines to the end of the file:


```
IMAGE_INSTALL:append = " can-utils"
IMAGE_INSTALL:append = " curl"
IMAGE_INSTALL:append = " gnupg"
```

Your final *local.conf* file should look like this

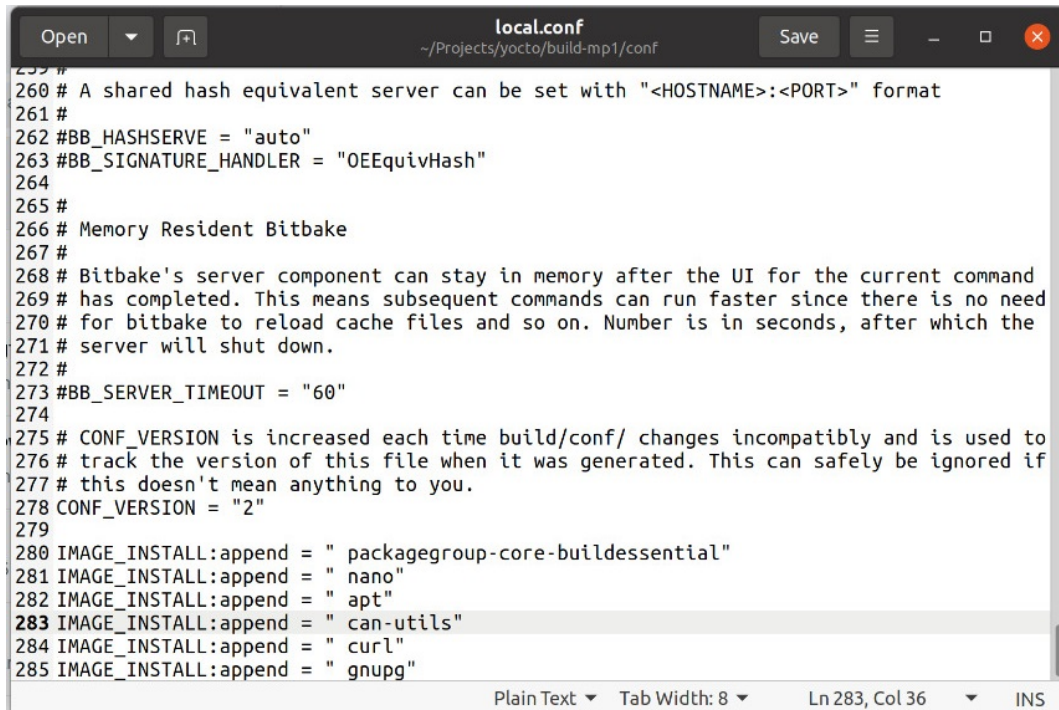


Figure 6.6: Extra packages like can-utils added in local.conf

To test a I²C sensor, we want to probe it on the I2C bus. The easiest way to do that is with the *i2cdetect* tool, which comes with busybox. However, it is not enabled by default for our image, so we need to enable it.

Navigate to the build folder and bring up the busybox menuconfig screen:

```
cd ../build-mp1
bitbake -c menuconfig busybox
```

In there, head to *Miscellaneous Utilities*, highlight *i2cdetect* and the other i2c utilities, and press Y to enable. It should have an asterisk [*] in the select box to denote that the tools will be included with busybox in the next build.

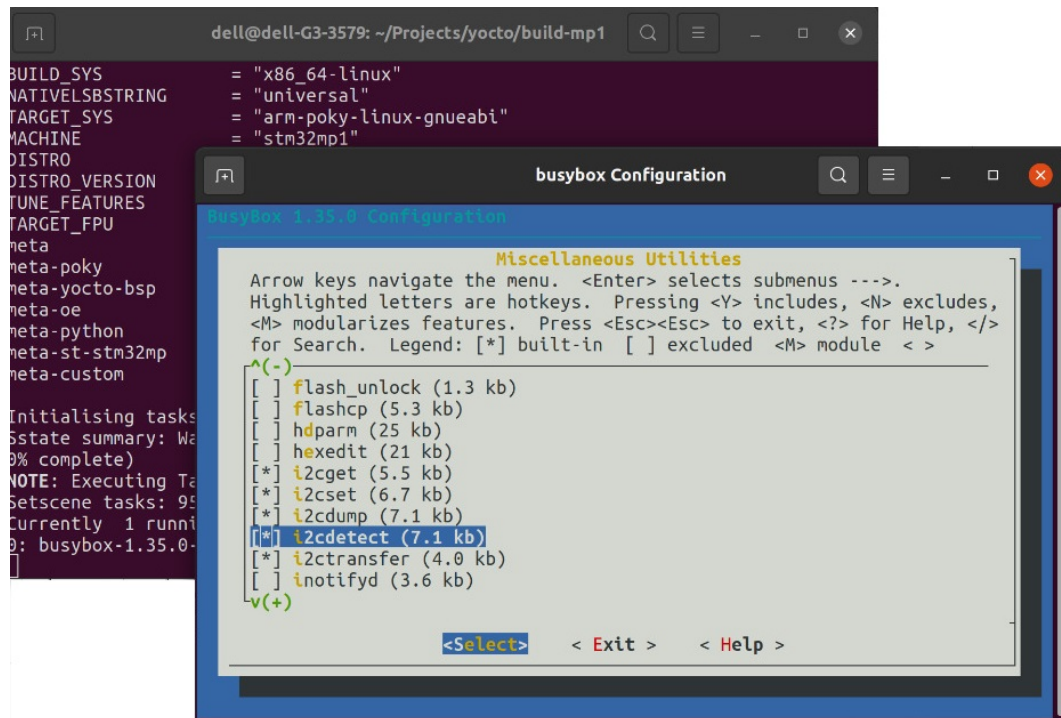


Figure 6.7: Busybox configuration

Select Exit with the arrow keys and press 'enter' to leave that screen. Do that again to exit menuconfig. Save the configuration when asked.

Building Custom Image

7.1 Build and Flash the Custom Image

Build the custom image:

```
bitbake custom-image
```

When the build process is finished, flash the image onto the SD card as discussed previously.

```
cd ~/Projects/yocto/build-mp1/tmp/deploy/images/stm32mp1/scripts

./create_sdcard_from_flashlayout.sh ../flashlayout_custom-image/extensible/FlashLayout_sdcard_stm32mp157f-dk2-extensible.tsv

sudo dd if=../flashlayout_custom-image/extensible/../../../../FlashLayout_sdcard_stm32mp157f-dk2-extensible.raw of=/dev/mmcblk0 bs=8M conv=fdatasync status=progress oflag=direct
```

7.2 Testing I²C and FDCAN

7.2.1 I²C Tools and Sensor Connection

Plug the SD card into your STM32MP157F-DK2 and boot it up. Connect to the serial terminal with the following (you may need to change ttyACM1 to some other device file):

```
sudo picocom -b 115200 /dev/ttyACM1
```

Log in to the board with "root" and run the following command line:

```
ls -l /sys/bus/i2c/devices
```

```
dell@dell-G3-3579: ~
udhcpd: no lease, forking to background
done.

Starting random number generator daemon.
Starting OpenBSD Secure Shell server: sshd
done.
root@stm32mp1:~# ls -l /sys/bus/i2c/devices
total 0
lrwxrwxrwx 1 root root 0 Jan 1 2000 0-0038 -> ../../devices/platform/soc/40012000.i2c/i2c-0/0-0038
lrwxrwxrwx 1 root root 0 Jan 1 2000 0-0039 -> ../../devices/platform/soc/40012000.i2c/i2c-0/0-0039
lrwxrwxrwx 1 root root 0 Jan 1 2000 0-004a -> ../../devices/platform/soc/40012000.i2c/i2c-0/0-004a
lrwxrwxrwx 1 root root 0 Jan 1 2000 2-0028 -> ../../devices/platform/soc/5c002000.i2c/i2c-2/2-0028
lrwxrwxrwx 1 root root 0 Jan 1 2000 2-0033 -> ../../devices/platform/soc/5c002000.i2c/i2c-2/2-0033
lrwxrwxrwx 1 root root 0 Jan 1 2000 i2c-0 -> ../../devices/platform/soc/40012000.i2c/i2c-0
lrwxrwxrwx 1 root root 0 Jan 1 2000 i2c-1 -> ../../devices/platform/soc/40015000.i2c/i2c-1
lrwxrwxrwx 1 root root 0 Jan 1 2000 i2c-2 -> ../../devices/platform/soc/5c002000.i2c/i2c-2
lrwxrwxrwx 1 root root 0 Jan 1 2000 i2c-3 -> ../../devices/platform/soc/40012000.i2c/i2c-0/i2c-3
root@stm32mp1:~#
root@stm32mp1:~#
root@stm32mp1:~#
root@stm32mp1:~#
root@stm32mp1:~#
root@stm32mp1:~#
```

Figure 7.1: i2cdetect list devices

This should show you which device files (in /dev/) are symbolically linked to i2c ports/drivers on the main processor. In my case, /dev/i2c-1 is linked to I2C-5 (address 0x40015000), which is the port we just enabled.

Assuming that we have connected [Waveshare MLX90640 thermal imaging camera](#) (0x33 address) to the I2C5 pins on the board, we should be able to run the following command. Note that the bus number (1) should match the device file number.

```
i2cdetect -y 1
```

```
root@stm32mp1:~# i2cdetect -y 1
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30:  -- -- -- 33 -- -- -- -- -- -- -- -- -- -- --
40:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70:  -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
root@stm32mp1:~#
```

Figure 7.2: Sensor detected at the address 0x33

If all goes well, you should see 0x33 being reported on the bus, which means you can communicate with the MLX90640!

7.2.2 FDCAN Initialisation and Loopback Test

To communicate with the CAN-FD bus on your STM32MP157F-DK2 board, you can refer to the following tutorial:

<https://github.com/darkquesh/stm32mp1/blob/main/fdcan.md>

Further Reading and Resources

- [1] STM32MP15 Discovery kits - getting started, *STMicroelectronics Wiki*
https://wiki.st.com/stm32mpu/wiki/STM32MP15_Discovery_kits_-_getting_started
- [2] STM32MP157F-DK2 User Manual, *STMicroelectronics*,
https://www.st.com/resource/en/user_manual/um2637-discovery-kits-with-increasedfrequency-800-mhz-stm32mp157-mpus-stmicroelectronics.pdf
- [3] STM32MP157F-DK2 Reference Manual, *STMicroelectronics*,
https://www.st.com/resource/en/reference_manual/dm00327659-stm32mp157-advanced-arm-based-32-bit-mpus-stmicroelectronics.pdf
- [4] Yocto Project Reference Manual, *Linux Foundation and Yocto Project*,
<https://docs.yoctoproject.org/ref-manual/index.html>
- [5] Github, <https://github.com/darkquesh/stm32mp1>