



File Status: <input type="checkbox"/> Draft <input checked="" type="checkbox"/> Release	FILE ID:	MYIR-MYD-YF13X-SW-DG-EN-L5.15.67
	VERSION:	V1.0[DOC]
	AUTHOR:	Nene
	CREATED:	2023-05-20
	UPDATED:	2023-05-20

Copyright © MYIR Electronics Limited 2011-2023 all rights reserved.



Revision History

VERSION	AUTHOR	PARTICIPANT	DATE	DESCRIPTION
V1.0	Nene		20230520	Initial Version: u-boot2021.1, Linux Kernel 5.15.67, Yocto 4.1.



CONTENT

Revision History	- 2 -
CONTENT	- 3 -
1. Overview	- 5 -
1.1. Software Resources	- 5 -
1.2. Document Resources	- 6 -
2. Development Environment	- 7 -
2.1. Developing Host Environment.....	- 7 -
2.2. Introduction of Software Development Tools.....	- 9 -
2.3. Install the SDK Customized by MYIR.....	- 9 -
3. Build the File System with Yocto	- 13 -
3.1. Introduction	- 13 -
3.2. Get the Source Code	- 14 -
3.2.1. Get Compressed Source Code from CD Image.....	- 14 -
3.2.2. Get Source Code from GitHub.....	- 15 -
3.3. Build Development Board Image.....	- 16 -
3.4. Build SDK (optional)	- 26 -
4. How to Burn System Image	- 27 -
4.1. How to Flash with STM32CubeProgrammer.....	- 27 -
4.2. Make TF Card Starter	- 32 -
5. How to Modify Board Level Support Package	- 35 -
5.1. Introduction to meta-myir-st Layer.....	- 35 -
5.2. Introduction to Board Level Support Package.....	- 38 -
5.3. TF-A Compilation and Update	- 40 -
5.3.1. How to build TF-A in standalone environment.....	- 40 -
5.3.2. How to build TF-A in Yocto Project	- 41 -



5.3.3. How to update TF-A separately	- 43 -
5.4. U-boot Compilation and Update	- 45 -
5.4.1. How to build U-Boot in standalone environment	- 46 -
5.4.2. How to build and load U-Boot in Yocto Project	- 47 -
5.4.3. How to update U-Boot separately	- 50 -
5.5. Kernel Compilation and Update	- 52 -
5.5.1. How to build Kernel in standalone environment	- 53 -
5.5.2. How to build Linux in Yocto Project	- 56 -
5.5.3. How to update Kernel separately	- 59 -
6. How to Fit Your Hardware Platform	- 64 -
6.1. How to Create Your Device Tree	- 64 -
6.1.1. Board Level Device Tree	- 64 -
6.1.2. Add your board level device tree	- 65 -
6.2. How to configure function pins according to your hardware	- 68 -
6.2.1. GPIO pin configuration	- 68 -
6.2.2. How to use GPIO in device tree	- 70 -
6.3. How to use your own configured pins	- 71 -
6.3.1. How to use GPIO in uboot	- 71 -
6.3.2. How to use GPIO in Kernel driver	- 71 -
6.3.3. How to control a GPIO in Userspace	- 78 -
7. How to add an application	- 85 -
7.1. Makefile-based project	- 85 -
7.2. Application based on QT	- 90 -
7.3. Automatic application startup at boot time	- 91 -
Appendix A	- 100 -
Warranty & Technical Support Services	- 100 -



1. Overview

There are many open source system construction frameworks on the Linux system platform, which facilitate developers to build and customize embedded systems. Buildroot, Yocto, OpenEmbedded and so on are common at present. The Yocto project takes a more powerful and customized approach to building Linux systems for embedded products. It is not only a tool to make file system, but also provides a whole set of development and maintenance workflow based on Linux, so that the bottom embedded developers and the upper application developers in the unified framework of development, to solve the traditional way of scattered and unmanaged development form.

This document mainly introduces the complete process of customizing a complete embedded Linux system based on Yocto project and Mill core board, including the preparation of development environment, the acquisition of code, and how to carry out Bootloader, Kernel transplantation, customize Rootfs suitable for their own application requirements. We first show you how to build a system image for the MYD-YF13X development board based on the source code we provided, and how to burn the built image to the development board. For those users who carry out project development based on the MYC-YF135 core board, we focus on introducing the methods and some key points to transplant this set of system to the user's hardware platform, and through some actual BSP transplant cases and Rootfs customization cases, users can quickly customize the system image suitable for their hardware.

This document does not contain the introduction of the Yocto project and basic knowledge of Linux system. It is suitable for embedded Linux system developers with certain development experience. For some specific functions that may be used by users in the secondary development process, we also provide detailed application notes for the reference of developers. For specific information, refer to the document list in Table 2-4 of MYD-YF13X SDK Release Instructions.

1.1. Software Resources



MYD-YF13X is equipped with the operating system based on Linux kernel version 5.15.67, providing rich system resources and other software resources. The development board comes with the cross-compilation tool chain required by the development of embedded Linux system, TF-A source code, Optee-os source code, U-boot source code, Linux kernel and each driver module source code, as well as a variety of development and debugging tools suitable for Windows desktop environment and Linux desktop environment. Application development sample, etc.

For specific software information, ,please refer to Table 2-4 of “MYD-YF13X SDK Release Notes” for the detailed list of documents.

1.2. Document Resources

The SDK contains various types of documentation and manuals, such as release instructions, introduction guides, evaluation guides, and development guides, depending on the different stages of the user's use of the development board. For detailed document list, please refer to table 2-4 of “MYD-YF13X SDK Release Notes” .



2. Development Environment

This chapter mainly introduces some software and hardware environments required in the development process of MYD-YF13X development board, including the necessary development host environment, necessary software tools, code and data acquisition, etc. Specific preparatory work will be detailed in the following.

2.1. Developing Host Environment

This section describes how to set up a development environment suitable for the STM32MP13X series processor platform. By reading this chapter, you will learn about the installation and use of hardware tools, software development and debugging tools. And can quickly build the relevant development environment, for the later development and debugging preparation. STM32MP13X series processor is a multi-core heterogeneous processor. Different processor cores will run different systems and use different development environments and tools. The details are as follows:

- Dual-core ARM Cortex A7 kernel that can run embedded Linux systems using common embedded Linux system development tools.

- **Host Hardware**

The construction of Yocto project has relatively high requirements on the development host, which requires the processor to have dual-core CPU or above, 8GB memory or above, 160GB hard disk or higher configuration. The host running the Linux OS, VM running the Linux OS, or WSL2 running the Windows OS can be used

- **Host Operating System**

There are many options for the host operating system used to build the yocto project. Please refer to the official Yocto instructions for details:

<https://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html#dev-preparing-the-build-host>. Generally, we choose to build it on the local host with

Fedora, openSUSE, Debian, Ubuntu, RHEL or Cent OS Linux distributions. Here, we recommend the Ubuntu 18.04 64bit desktop system, the subsequent development is also based on this system.

● Prerequisite Package Installation

```
PC$ sudo apt-get update
```

```
PC$ sudo apt-get install u-boot-tools libyaml-dev bison flex sed wget curl cvs subversion git-core coreutils unzip texi2html texinfo docbook-utils gawk python-pysqlite2 diffstat help2man make gcc build-essential g++ chrpath libxml2-utils xmlto docbook bsdmainutils iputils-ping cpio python-wand python-pycryptopp python-crypto
```

```
PC$ sudo apt-get install libsdl1.2-dev xterm corkscrew nfs-common nfs-kernel-server device-tree-compiler mercurial u-boot-tools libarchive-zip-perl
```

```
PC$ sudo apt-get install ncurses-dev bc linux-headers-generic gcc-multilib libncurses5-dev libncursesw5-dev lrzsz dos2unix lib32ncurses5 repo libssl-dev
```



2.2. Introduction of Software Development Tools

In the process of customizing a Linux system suitable for the ARM Cortex A7 core, you will use a number of debugging, burning and writing tools, some of which are provided in the CD image directory 03_Tools provided by MYiR, in addition to a series of tools provided by ST, which are briefly described as follows:

- **STM32CubeProg**

STM32CubeProgrammer is a new high - integration tool, which is the official STMicroelectronics tool for creating partitions into any Flash device available on STM32 platforms, once created, STM32CubeProgrammer allows populating and updating the partitions with the prebuilt binaries. MYiR offers both Linux and Windows versions for the user to choose from. This chapter covers the installation steps for both platforms. Users can choose the appropriate version according to their needs. Download link: <https://www.st.com/zh/development-tools/stm32cubeprog.html> .

- **STM32CubeMX**

ST introduced code generation software specifically designed to generate HAL code for STM32. The STM32CubeMX application helps developers to use the STM32 by means of a user interface, and guides the user through the initial configuration of a firmware project. Therefore, all clocks, tick timers, DMA, serial ports, GPIO, etc, are not configured according to the data manual to operate the standard library. In addition, since ST has added STM32MPU series CPUs to STM32CubeMX, we can also use this tool to configure TF-A, U-boot and Kernel device trees and clocks, so it can be used to generate the code for Cortex M4 as well as generate device trees and clocks during Cortex A7 development. It is currently available on the Windows platform. Download link: <https://www.st.com/zh/development-tools/stm32cubemx.html> .

2.3. Install the SDK Customized by MYiR

After using Yocto to build the system image, we can also use Yocto to build a set of extensible SDK. The CD image provided by MYIR contains a compiled SDK package, which is located in the *03_tools/Qt-SDK* directory, and the file name is *sdk-qt.tar.xz*. This SDK not only contains an independent cross development tool chain, but also provides qmake, sysroot of the target platform, libraries and header files that QT application development depends on, etc. Users can directly use this SDK to establish an independent development environment, compile bootloader, kernel or their own applications. The specific process will be described in detail in the following chapters. Here we will first introduce the installation steps of the SDK, the steps are as follows:

- **Copy the SDK to the Linux directory and run the tar command to decompress the package**

Copy the SDK package to the working directory in ubuntu(eg:\$HOME/work). Then uncompress the tarball file to get the following SDK package installation script, as follows:

```
PC$ cd $HOME/work
PC$ tar -Jxvf sdk-qt.tar.xz
sdk
```

- **View script file**

Go to the SDK directory, you can find the installation script:

```
meta-toolchain-qt5-openstlinux-weston-myir-yf13x-x86_64-toolchain-4.0.4-snaps
hot.host.manifest
meta-toolchain-qt5-openstlinux-weston-myir-yf13x-x86_64-toolchain-4.0.4-snaps
hot.sh
meta-toolchain-qt5-openstlinux-weston-myir-yf13x-x86_64-toolchain-4.0.4-snaps
hot.target.manifest
meta-toolchain-qt5-openstlinux-weston-myir-yf13x-x86_64-toolchain-4.0.4-snaps
hottestdata.json
```

- Run the SDK installation script

```
PC$ ./meta-toolchain-qt5-openstlinux-weston-myir-yf13x-x86_64-toolchain-4.0.4
-snapshot.sh
```

- Select the installation directory

The SDK is installed in the `/opt/st/myir-yf13x/4.0.4-snapshot` directory by default. Users can also choose the appropriate directory according to the prompts:

```
ST OpenSTLinux - Weston - (A Yocto Project Based Distro) SDK installer version 4.0.4-snapshot
=====
=====
Enter target directory for SDK (default: /opt/st/myir-yf13x/4.0.4-snapshot):
The directory "/opt/st/myir-yf13x/4.0.4-snapshot" already contains a SDK for this architecture.
If you continue, existing files will be overwritten! Proceed [y/N]? y
Extracting SD
K.....done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
$ . /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

- Test SDK

Initialize cross-compilation via SDK and ensure that the environment is correctly setup:

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

```
PC$ $CC --version
```

arm-ostl-linux-gnueabi-gcc (GCC) 11.3.0

Copyright (C) 2021 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PU
RPOSE.

The SDK provided by MYIR includes not only cross tool chain, but also Qt library,
qmake and other resources needed to develop QT applications. These are the
basis for the subsequent application development and debugging with QT creator.

3. Build the File System with Yocto

3.1. Introduction

The Yocto Project is an open-source "umbrella" project, meaning it has many sub-projects under it. Yocto just puts all the projects together and provides a reference build project Poky to guide developers on how to apply these projects to build an embedded Linux system. Yocto also Contains Bitbake tool, OpenEmbedded-Core, board level support packages, configuration files of various software, so you can build systems with different requirements. For more information about the Yocto project, please refer to the site: www.yoctoproject.org .

MYIR's CD image *04 sources* provides Yocto meta-files and data applicable to MYD-YF13X development board to help developers build different types of Linux system images that can run on MYD-YF13X development board, such as myir-image-full system image with Qt5.15.3 graphics library. myir-image-core system image without GUI interface. The following takes the construction of myir-image-full image as an example to introduce the specific development process, so as to lay a foundation for the subsequent customization of system image suitable for ourselves.

Note: the SDK toolchain environment variables in Section 2.3 do not need to be loaded to build the yocto system. Please create a new shell or open a new terminal window.

3.2. Get the Source Code

We provide two ways to obtain the source code. One is to obtain the compressed package directly from the *04_Sources* directory of the MYIR CD image, and the other is to use repo to obtain the source code updated in real time on GitHub for construction. Users can choose one of them.

Note: before building the Yocto system, all software packages in the file system need to be downloaded to the local. In order to build quickly, MYD-YF13X has packaged the relevant software, and users can directly unzip and copy it to the build directory, so as to reduce the repeated download time.

3.2.1. Get Compressed Source Code from CD Image

You can find the Yocto compressed source package in the development kit package *04_Sources/yf13x-yocto-stm32mp1-5.15.67.tar.bz2*. Copy the compressed package to the user specified directory, such as the *\$HOME/yocto* directory. This directory will be used as the top-level directory for subsequent construction. After decompressing, the layers directory will appear:

```
PC$ cd $HOME/Yocto
PC$ tar -jxvf yf13x-yocto-stm32mp1-5.15.67.tar.bz2
layers
```

List the layers directory as follows:

```
PC$ $ tree -d -L 1 layers
layers
├── meta-myir-st
├── meta-openembedded
├── meta-qt5
├── meta-st
└── openembedded-core
```

5 directories

3.2.2. Get Source Code from GitHub

At present, the BSP source code and yocto source code of MYD-YF13X development board are managed by GitHub and will be updated for a long time. Please refer to Section 2.2 of “MYD-YF13X SDK Release Notes”. Users can use repo to get and synchronize the code on GitHub. The specific operation methods are as follows:

```
PC$ mkdir $HOME/github
PC$ cd $HOME/github
repo init -u https://github.com/MYiR-Dev/myir-st-manifest.git --no-clone-bundle --depth=1 -m myir-stm32mp1-kirkstone.xml -b develop-yf13x
PC$ repo sync
```

After the synchronization code is completed, you will get a layers folder under the *\$home/GitHub* directory, which contains the source code or source repository path related to MYD-YF13X development board. The directory structure is the same as that extracted from the compressed package.

3.3. Build Development Board Image

Before using the Yocto project to build the system, we need to set the corresponding environment variables. Before building myir-image-full, we need to use the envsetup.sh script provided by Mir to set the environment variables. A build directory (eg: build-openstlinuxweston-myd-yf13x-emmc) is created during setup, and subsequent builds, as well as output files, are contained in this directory.

- **Execute script to set environment variables**

The syntax for the envsetup.sh script is shown below:

```
PC$: DISTRO=openstlinux-weston MACHINE=myd-yf13x-emmc source layers/meta-myir-st/scripts/envsetup.sh
```

Note: The NAND version MACHINE needs to be selected as myd-yf13x-nand.

The Yocto Project provides some images which are available on different layers, so it will also list all the project images for MYD-YF13X. As shown below:

```
[HOST DISTRIB check]
```

```
Linux Distrib: Ubuntu
```

```
Linux Release: 18.04
```

```
Required packages for Linux Distrib:
```

```
bsdmainutils build-essential chrpath cpio debianutils diffstat gawk gcc-multilib  
git iutils-ping libegl1-mesa libgmp-dev liblz4-tool libmpc-dev libsdl1.2-dev libssl-dev  
pylint3 python3 python3-git python3-jinja2 python3-pexpect python3-pip socat texinfo  
unzip wget xterm xz-utils zstd
```

```
Check OK: all required packages are installed on host.
```

```
[source layers/openembedded-core/oe-init-build-env][with previous config]
```

=====

=====

Configuration files have been created for the following configuration:

```
DISTRO          : openstlinux-weston
DISTRO_CODENAME : kirkstone
MACHINE         : myd-yf13x-emmc
BB_NUMBER_THREADS : 6
PARALLEL_MAKE    : <no-custom-config-set>

BUILDDIR        : build-openstlinuxweston-myd-yf13x-emmc
DOWNLOAD_DIR     : <disable>
SSTATE_DIR       : <disable>

SOURCE_MIRROR_URL : <no-custom-config-set>
SSTATE_MIRRORS   : <disable>

WITH_EULA_ACCEPTED: NO
```

=====

=====

Available images for OpenSTLinux layers are:

- The system based on QT framework

 myir-image-full - MYiR HMI demo of image based on QT framework (require 'openstlinux-weston' distro)

- Other OpenSTLinux images:

 myir-image-core - OpenSTLinux core image

and more images are available on meta-myir-st/recipes-myir/images.

You can now run 'bitbake <image>'

After the configuration script is executed, go to the build-openstlinuxweston-myd-yf13x-emmc directory, where you can start building the system.

- **Building myir-image-full image**

If you choose to build different system images, you need to use different bitbake commands. For specific commands, refer to the table below. We choose myir image full as an example to illustrate.

Table 3-1.System image optional list

System Name	Command
measy-hmi2.0 system based on qt5.15	bitbake myir-image-full
core system based on openstlinux	bitbake myir-image-core

Note: If you compile the NAND image, only myir-image-core and demo with lvgl are available. **bitbake myir-image-core** is executed when the whole image is compiled

Bitbake Parameter	Description
-k	Continue building when there are errors
-c cleanall	Clear the entire build directory
-c fetch	From the address defined in recipe, pull the software source code to the local
-c deploy	Deploy the image or software package to the target rootfs
-c compile	Recompile image or package

Table 3-2. Common Commands

Note: It is recommended to decompress yocto qt-downloads.tar.xz Package to the build-openstlinuxeglfs-myir directory so that users can save a lot of time.

The following command is an example on how to build an image:

Build Configuration:

```
BB_VERSION          = "2.0.0"
BUILD_SYS           = "x86_64-linux"
NATIVELSBSTRING    = "universal"
TARGET_SYS          = "arm-ostl-linux-gnueabi"
MACHINE             = "myd-yf13x-emmc"
DISTRO              = "openstlinux-weston"
DISTRO_VERSION      = "4.0.4-snapshot-20230506"
TUNE_FEATURES       = "arm vfp cortexa7 neon vfpv4 thumb callconvention
-hard"
TARGET_FPU          = "hard"
DISTRO_CODENAME     = "kirkstone"
ACCEPT_EULA_myd-yf13x-emmc = "1"
GCCVERSION          = "11.%"
PREFERRED_PROVIDER_virtual/kernel = "linux-myir"
meta-python
```

```

meta-oe
meta-gnome
meta-initramfs
meta-multimedia
meta-networking
meta-webserver
meta-filesystems
meta-perl
meta-st-stm32mp
meta-qt5
meta-st-openstlinux = "<unknown>:<unknown>"
meta-myir-st-yf13x = "master:8132dfabf8abf729d5869bcd9c596f6168b4ab67"
meta                  = "<unknown>:<unknown>"

```

Initialising tasks: 100% |#####| #####| #####| #####| #####| #####| #####| #####| #####| Time: 0:00:01

Sstate summary: Wanted 64 Local 59 Mirrors 0 Missed 5 Current 390 (92% match, 98% complete)

NOTE: Executing Tasks

After the system is compiled successfully, the system image files of various formats are generated under the directory "*build-openstlinuxweston-myd-yf13x-emmc/tmp-glibc/deploy/images/myir*". The following is a list of file information generated after the build:

```

myir/
├── arm-trusted-firmware
├── build-myir-image-full-openstlinux-weston-myd-yf13x-emmc
├── fip
├── flashlayout_myir-image-full
└── kernel

```

```
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253-lic  
ense_content.html  
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.roo  
tfs.ext4  
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.roo  
tfs.manifest  
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.roo  
tfs.tar.xz  
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.tes  
tdata.json  
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc.ext4 -> myir-image-  
full-openstlinux-weston-myd-yf13x-emmc-20230508025253.rootfs.ext4  
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc-license_content.html  
-> myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253-lice  
nse_content.html  
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc.manifest -> myir-im  
age-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.rootfs.manifest  
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc.tar.xz -> myir-image-  
full-openstlinux-weston-myd-yf13x-emmc-20230508025253.rootfs.tar.xz  
|── myir-image-full-openstlinux-weston-myd-yf13x-emmc.testdata.json -> myi  
r-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253testdata.jso  
n  
|── optee  
|── scripts  
|── st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bo  
otfs.ext4  
|── st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bo  
otfs.manifest  
|── st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bo  
otfs.tar.xz  
|── st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.tes  
tdata.json
```

```
|── st-image-bootfs-openstlinux-weston-myd-yf13x-emmc.ext4 -> st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bootfs.ext4
|── st-image-bootfs-openstlinux-weston-myd-yf13x-emmc.manifest -> st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bootfs.manifest
|── st-image-bootfs-openstlinux-weston-myd-yf13x-emmc.tar.xz -> st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bootfs.tar.xz
|── st-image-bootfs-openstlinux-weston-myd-yf13x-emmctestdata.json -> st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253testdata.json
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spinand_2_128.ubinize.cfg.ubi
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spinand_2_128.userfs.ubi
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spinand_2_128.userfs.ubifs
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753testdata.json
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.userfs.ext4
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.userfs.manifest
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.userfs.tar.xz
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc.ext4 -> st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.userfs.ext4
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc.manifest -> st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.userfs.manifest
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubi -> st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spinand_2_128.userfs.ubi
```

```
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubifs
-> st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spi
nand_2_128.userfs.ubifs
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubini
ze.cfg.ubi -> st-image-userfs-openstlinux-weston-myd-yf13x-emmc-2023050709
4753_spinand_2_128.ubinize.cfg.ubi
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmc.tar.xz -> st-image-u
serfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.userfs.tar.xz
|── st-image-userfs-openstlinux-weston-myd-yf13x-emmctestdata.json -> st-i
mage-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753testdata.jso
n
|── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_
spinand_2_128.ubinize.cfg.ubi
|── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_
spinand_2_128.vendorfs.ubi
|── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_
spinand_2_128.vendorfs.ubifs
|── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.
testdata.json
|── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.
vendorfs.ext4
|── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.
vendorfs.manifest
|── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.
vendorfs.tar.xz
|── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.ext4 -> st-image-
vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.vendorfs.ext4
|── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.manifest -> st-im
age-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.vendorfs.m
anifest
```

```

└── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubi
    -> st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_
        spinand_2_128.vendorfs.ubifs
└── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubifs
    -> st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_
        spinand_2_128.vendorfs.ubifs
└── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubinize.cfg.ubi
    -> st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spinand_2_128.ubinize.cfg.ubi
└── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.tar.xz -> st-image
    -vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.vendorfs.tar.xz
└── st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.testdata.json -> s
    t-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.testdat
    a.json
└── st-initrd-openstlinux-weston-myd-yf13x-emmc
    └── u-boot

```

7 directories, 48 files

Table 3-3. Description of images file list (not all)

Name	Description
arm-trusted-firmware	TF-A burning package file
uboot	Uboot burning package file
kernel	Generating files of kernel and device tree
flashlayout_myir-image-full	TSV layout file
st-image-bootfs-openstlinux-weston-myd-yf13x-emmc.ext4	Bootfs partition file, including kernel uimage, device tree, boot configuration file, etc
st-image-userfs-openstlinux-weston-myd-yf13x-emmc.ext4	The userfs partition file mainly contains the demo program for stm32mp1
st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.ext4	Vendorfs partition file, including third-party library files, such as eglfs library
myir-image-full-openstlinux-weston-myd-yf13x-emmc.ext4	Rootfs partition file that contains the root file system
scripts	create_sdcard_from_flashlayout.sh -Making SD



startup package



3.4. Build SDK (optional)

MYIR has provided a relatively complete SDK installation package, which can be directly used by users. However, when users need to introduce new libraries into the SDK, they need to reuse yocto to build new SDK tools.

This section simply describes how to build the SDK .The following command is an example on how to build the SDK package:

```
bitbake -c populate_sdk meta-toolchain-qt5
```

After building, the SDK installation package will be generated in the path of "*build-openstlinuxeglfs-myd-yf13x-emmc/tmp-glibc/deploy/sdk*". Please refer to Section 2.3 for the installation method.

```
meta-toolchain-qt5-openstlinux-weston-myir-yf13x-x86_64-toolchain-4.0.4-snaps
hot.host.manifest
meta-toolchain-qt5-openstlinux-weston-myir-yf13x-x86_64-toolchain-4.0.4-snaps
hot.sh
meta-toolchain-qt5-openstlinux-weston-myir-yf13x-x86_64-toolchain-4.0.4-snaps
hot.target.manifest
meta-toolchain-qt5-openstlinux-weston-myir-yf13x-x86_64-toolchain-4.0.4-snaps
hottestdata.json
```

4. How to Burn System Image

MYD-YF13X series CPU Module and development board designed by MYIR company are equipped with STM32mp135 microprocessor of ST company. MYD-YA157C series products have a variety of start-up methods, so different update system tools and methods are required .Users can choose different platforms to update according to their needs.The update methods mainly include the following:

- flash with STM32CubeProgrammer:It is suitable for R&D, debugging, testing, etc.
- Make TF card starter : It is suitable for R&D,debugging, quick start, etc.

4.1. How to Flash with STM32CubeProgrammer

1) Tools requirements

- A development board
- Type_C cable
- Adapter of 12V/2A
- Official STM32CubeProgrammer tool

2) Hardware Connection and Configuration

Set the dial switch to Select Download mode(B2/B1/B0 : 0 0 0).Connect the USB Type-C to USB Type-A cable between your PC and the USB Type-C OTG port of the development board(J14) and then connect the power adapter. As shown in the figure below:

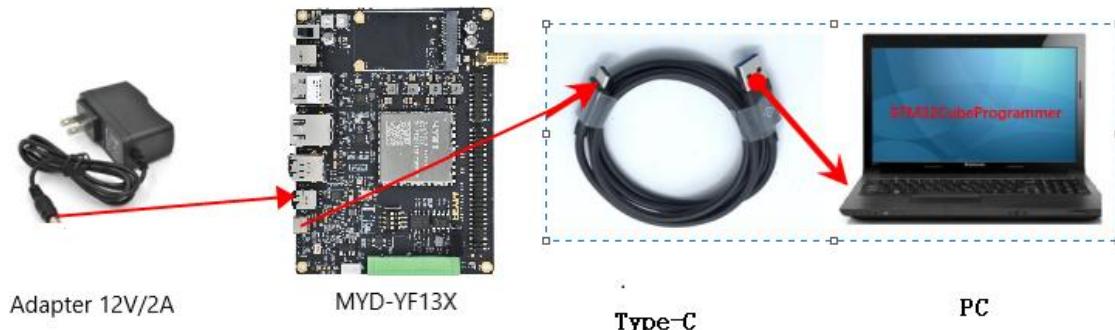


Figure 4.1.Connection Diagram

About the update method, depending on the PC platform you use, there are also two recommended ways, Optionally choose one:

In Windows platform, it is recommended to use the program installed under windows, but in Linux platform, it is recommended to use the program installed on Linux platform.

3) Burning System Under Windows

Run the STM32CubeProgrammer software installed on the Windows platform, and select USB (label 1) burning mode. Click connect (label 2) button to connect. Check whether the development board information can be displayed normally as shown in the following figure:

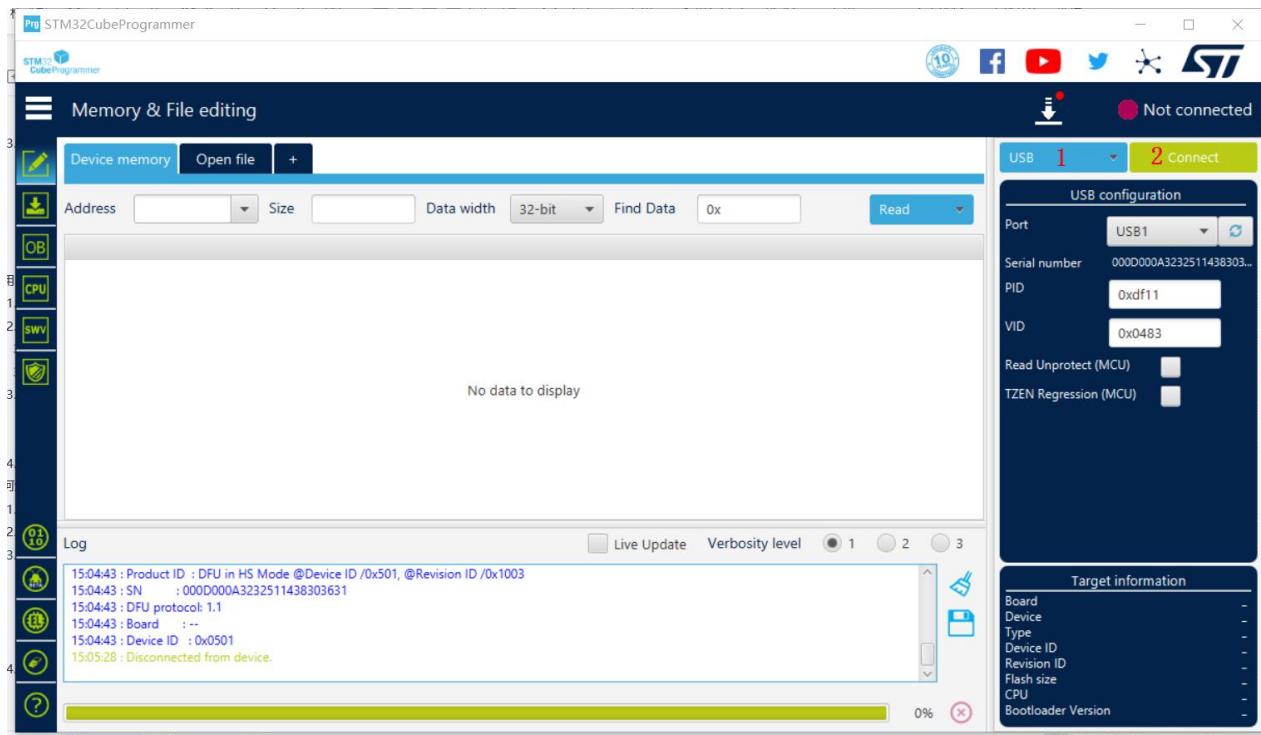


Figure 4-2. Connect information

Go to the specified directory that contains the binaries and the Flash layout files .Different storage devices require corresponding configuration files.Take eMMC for example,click the [Open file] (Label 1 as shown in figure 4.3) to locate to the default configuration file directory. A flash programming example is by default located at: <*MYIR source Patch*>/02_Image/myir-image-full, then select the default eMMC Layout file:*FlashLayout_emmc_myb-stm32mp135x-512m-optee.tsv*.

Go to [Browse] button and select the folder name where the Flash image layout is locate,which Circled in red in figure 4.3.

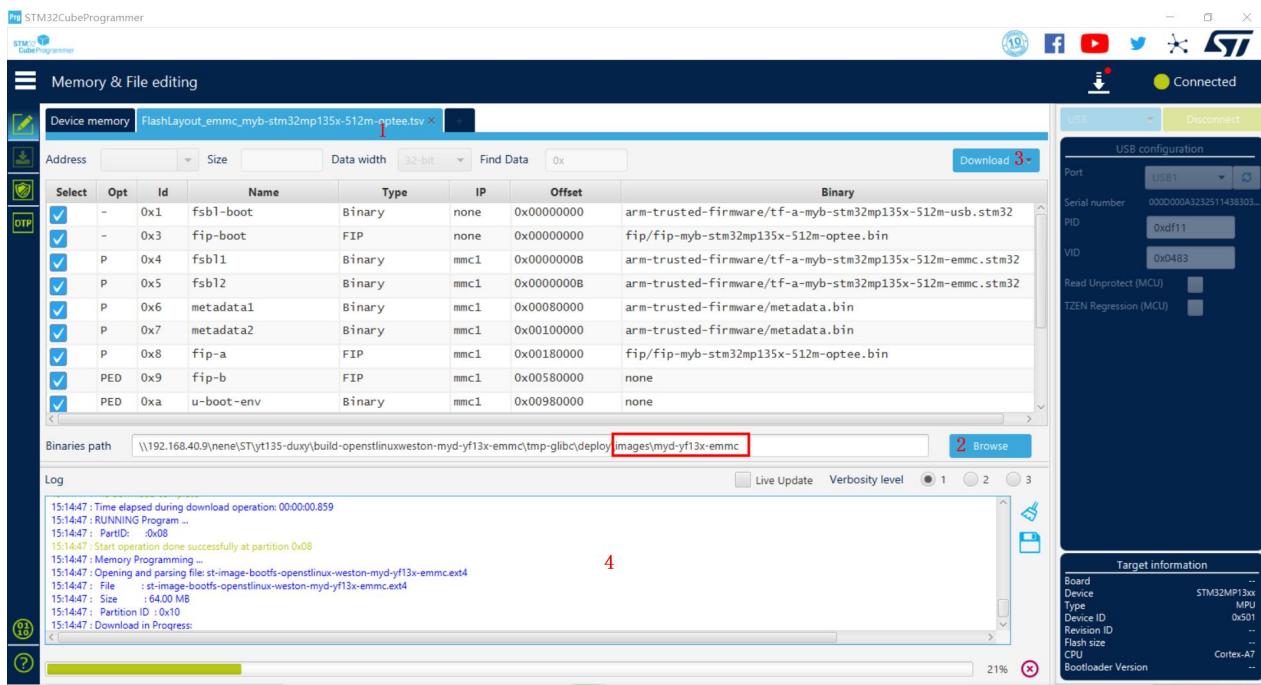


Figure 4-3. Select Image Directory

Then Start to update file though clicking on [Download] button (Label 3 as shown in figure 4.3).It takes longer to download the system, please be patient. For other storage devices, such as NandFlash/TF CARD, there are the same update steps as above.

4) Burning System under Linux

This section describes how to install the linux version STM32CubeProgrammer

```
PC$ ./SetupSTM32CubeProgrammer-2.10.0.linux
```

After the installation directory will generate

STMicroelectronics/STM32Cube/STM32CubeProgrammer

Create a new working directory, then copy the development kit materials *02_Image/myir-image-full/<all>* into a working directory, such as */home/myir/work*.

```
PC$ mkdir -p /home/myir/work
PC$ cd /home/myir/work
```

Add the STM32CubeProgrammer binary path to your PATH environment variable:

```
PC$ export PATH=$HOME/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin:$PATH
```

The following command is an example on how to start the installed stm32cuprammer software for burning,take micro SD card as an example:

```
PC$ STM32_Programmer_CLI -c port=usb1 -w flashlayout_myir-image-full/FlashLayout_sdcard_myb-stm32mp135x-512m-optee.tsv
```

It takes longer to download the system, please be patient. For other storage devices, such as NandFlash/TF CARD, there are the same update steps as above.

4.2. Make TF Card Starter

Make TF card starter under Windows.

1) Preparation

- One TF card (not less than 4GB)
- MYD-YF13X development board
- Flash tool Win32DiskImager-1.0.0-binary (Path: /03_Tools/myirtools)

Table 4-1. Image Package List

Image Name	Package Name	Applicable board
myir-image-full	flashlayout_myir-image-full_FlashLayout_sdcard_yf13x-4e512d.zip	MYD-YF13X-4E512D
myir-image-core	flashlayout_myir-image-core_FlashLayout_sdcard_myb-stm32mp135x-512m-optee.zip	MYD-YF13X-4E512D
myir-image-lvgl	FlashLayout_sdcard_myb-stm32mp135x-256m-optee.zip	MYD-YF13X-256N256D

2) How to make SD card starter

Take myir-image-full system as an example.

- Extract tarball:

Win32DiskImager-1.0.0-binary.zip

flashlayout_myir-image-full_FlashLayout_sdcard_yf13x-4e512d.zip

- Write image file to micro SD card

Insert the TF card into the computer with the card reader, double-click to open Win32DiskImager. Click the arrow to the location to load the image file :

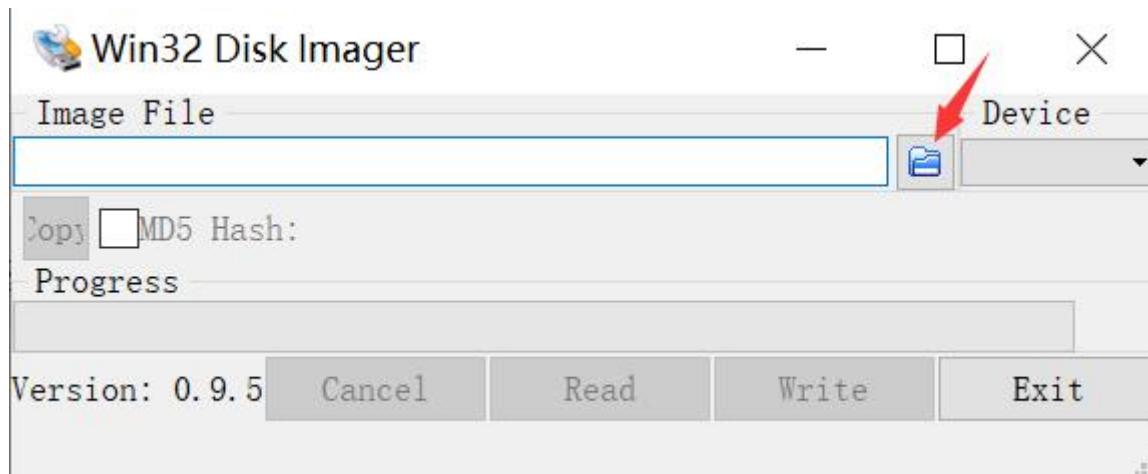


Figure 4-4.Win32 Disk Imager

Then go to [Select a disk image] interface and select [...] in the bottom left corner ,as shown in the red box below :



Figure 4-5.Tool Configuration

After loading the image, click the [write] button and the warning will pop up. Click [Yes] and wait for the write to complete.

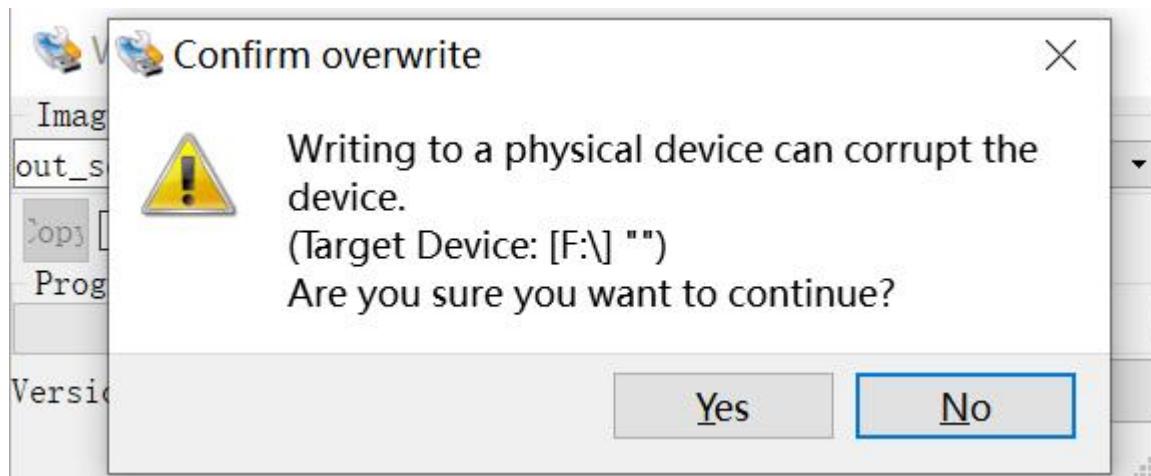


Figure 4-6.Writing

Wait for the write to complete, approximately 3-4 minutes, depending on the TF read/write speed :

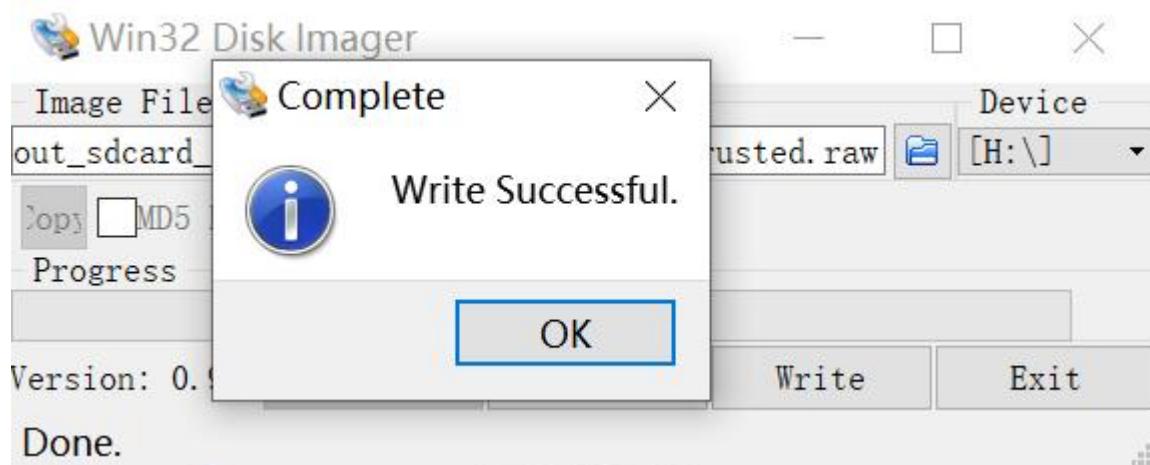


Figure 4-7.Write Successful

- **Check for success**

When the writing is complete, insert the TF into the TF card slot (J18) on the back of the development board. Set the boot switch to (B2/B1/B0: 1, 0, 1), and then use the TF to start.

5. How to Modify Board Level Support Package

The previous chapters have told a relatively complete process of constructing the system image running on the MYD-YF13X development board based on the Yocto project, and burning the image to the development board. Because many pins of MYC-YF13X core board have the characteristics of multi-function reuse, there will always be some differences between the bottom plate designed based on MYC-YF13X core board and MYB-YF13X in the actual project. These differences may be removing the display, adding more GPIOs, or needing to add more serial ports, or expanding some peripherals through SPI, I2C, USB, etc.; In addition to the hardware differences, there are some differences in system components, such as focusing on HMI application, may need relatively complete graphics system, QT library, etc., focus on background management applications, may need more complete network applications, Python operating environment. This requires system developers to do some tailoring and porting from the code base we provide. From the perspective of a system developer, this chapter describes the specific process of developing and customizing their own system, laying the foundation for their own hardware later.

5.1. Introduction to meta-myir-st Layer

The "layer model" of Yocto project is a development model created for embedded and Internet of things Linux, which distinguishes Yocto project from other simple build systems. The layer model supports both collaboration and customization. A layer is a repository of related instruction sets that tell openembedded what to do with the build system.

The Yocto Project makes it easy to create and share a BSP for a new ST based board. For example, the meta-myir-st layer is based on the ST official meta-st layer, which is suitable for MYD-YF13X development board, and the layer contains BSP, GUI, release configuration, middleware or application metadata and

recipes. Therefore, users can add or remove packages from the BSP provided for ST Linux distributions by creating a custom Yocto project layer. The contents of meta-myir-st layer are as follows:

```
meta-myir-st/
├── classes
├── conf
├── files
├── recipes-app
├── recipes-bsp
├── recipes-kernel
├── recipes-myir
├── recipes-security
└── scripts
```

9 directories

Table 5-1. meta-myir-st layer content description

Source Code and Data	Description
conf	Including development board software configuration resource information
recipes-app	Applications , such as measy-hmi2.0
recipes-bsp	Contains TF-A and uboot configuration resources,etc
recipes-kernel	Contains Linux kernel resources and third-party firmware resources
recipes-security	Contains Optee configuration resources
recipes-myir	Contains configuration information for the file system
scripts	Yocto environment configuration

In the process of system transplantation, we should focus on the recipes-bsp part which is responsible for hardware initialization and system boot, the recipes-kernel part responsible for Linux kernel and driver implementation, and the recipes-app part responsible for application customization.

5.2. Introduction to Board Level Support Package

Board level support package (BSP) is a collection of information that defines how to support a specific hardware device, device set or hardware platform. The BSP includes information about the hardware features on the device and kernel configuration information, as well as any other hardware drivers required.

In some cases, BSP contains separately licensed intellectual property (IP) for one or more components, so the terms of commercial or other types of licenses that require some explicit end user license agreement (EULA) must be accepted. Once you accept the license, MYIR's development board uses BSP to comply with the open source agreement license, and the source code of BSP will be fully open source.

Table 5-2. Open source protocol supported by BSP

IP Project	Open Source Agreement	Description
tf-a-myir	BSD-3-Clause	Trusted Firmware-A
u-boot-myir-extlinux	MIT	Uboot extlinux.conf
u-boot-myir	GPL-2.0-or-later	uboot
linux-myir	GPL-2.0-only	Linux kernel

In general, we divide BSP into Bootloader part and Kernel part according to different stages of hardware startup. The hardware BSP code designed with the MYC-YF13X core board can view the contents of recipes-bsp and recipes-kernel in the meta-myir-st.

recipes-bsp includes only the Bootloader section, trusted-firmware-a and u-boot, which does the core hardware, such as DDR, Clock initialization, and kernel booting. Modify this part based on MYC-YF13X core board hardware.

```
recipes-bsp
└── trusted-firmware-a
└── u-boot
```

The recipes kernel contains two parts: Linux kernel and Linux firmware, which mainly realizes kernel configuration and peripheral firmware addition.

```
recipes-kernel/
```

```
  └── linux
```

```
    └── linux-firmware
```

When users are ready to design their own hardware with myir's CPU module, if there is no special requirement, the bootloader part can not be modified. Instead, you need to pay more attention to the development and debugging of product kernel drivers and the design of application software, and the following chapters will describe the kernel development and application development in detail.

5.3. TF-A Compilation and Update

Trusted Firmware-A(Trusted Firmware-A) is a reference implementation of secure-world software provided by Arm. It was first designed for Armv8-A platforms, and has been adapted to be used on Armv7-A platforms by STMicroelectronics. Arm is transferring the Trusted Firmware project to be managed as an open-source project by Linaro. The code is open source, under a BSD-3-Clause licence, and can be found on github .including an up-to-date documentation about Trusted Firmware-A implementation.

In the meta myir-st-layer model, TF-A is a part of the recipes BSP. When booting from Trusted Boot Chain(For more information on Boot Chain, refer to https://wiki.st.com/stm32mpu/wiki/Boot_chain_overview) mode, it is used as fsbl (first stage boot loader) to initialize core resources such as DDR and clock. When users are ready to design their own hardware with myir's CPU module, if there is no special requirement, the TF-A part can not be modified. However, if you need to modify a peripheral clock, or need to control a pin early, you can refer to the following method.

5.3.1. How to build TF-A in standalone environment

1) Get TF-A source code

Copy *04_Sources/MYiR-STM32-tf-a.tar.bz2* file to the specific work directory of linux(eg:\$HOME/work), extract tarball of TF-A and view the corresponding file:

```
PC$ cd $HOME/work
PC$ tar -jxvf MYiR-STM32-tf-a.tar.bz2
PC$ cd MYiR-STM32-tf-a
```

- Source directory : myir-st-arm-trusted-firmware
- Compiled script : Makefile.sdk
- Instruction : README.HOW_TO.txt

```
-rwxrwxrwx 1 root root 2977 Jul 5 06:58 Makefile.sdk
drwxr-xr-x 18 root root 4096 Jul 5 06:51 myir-st-arm-trusted-firmware
-rw----- 1 licy licy 4705 Apr 20 02:25 README.HOW_TO.txt
```

2) Configuration and Compilation

- Initialize cross-compilation via SDK

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

- Go into the source code

```
PC$ cd myir-st-arm-trusted-firmware
```

- Configure and Compile source code

```
PC$ make -f $PWD/../Makefile.sdk all
```

After successful compilation, TF-A binary files will be generated in the *build/* directory, as shown in the following table:

Table 5-3. TF-A Binary File List

Name	Description
tf-a-myb-stm32mp135x-256m-sdcard.stm32	The u-boot binary file containing STM32 header is used for fsbl under trusted boot chain
tf-a-myb-stm32mp135x-256m-usb.stm32	
tf-a-myb-stm32mp135x-512m-sdcard.stm32	
tf-a-myb-stm32mp135x-512m-usb.stm32	
tf-a-bl2-usb.elf	ELF file for debugging

5.3.2. How to build TF-A in Yocto Project

In the process of system development, it is generally recommended to use git tools to manage source code, so as to better manage multiple versions of code. If the code is modified and tested correctly, it can be submitted to the repository, which makes it easier to build the system using Yocto.

If you want to set up git repository to manage the code and build a new image completely according to the process in Chapter 3, first, you need to submit the modified TF-A source code to git repository, and then modify the corresponding source pull address(SRC_URI) and source revision(SRCREV). Take GitHub repository as an example:

First, you need to enter the TF-A source code directory to submit the modified TF-A source code, and then find the latest commit value in GitHub repository and copy the value, the following command is an example on how to get the latest commit value:

```
PC$: git log --pretty=oneline
4cc966a5372b13517343009b2f8797cb99828ce8 (HEAD -> develop-v2.6-stm32mp, origin/develop-v2.6-stm32mp, origin/HEAD) FEAT: add spi-nand support
```

Next, go to *layers/meta-myir-st/recipes-bsp/trusted-firmware-a*/directory, then open the recipe *tf-a-myir-common.inc* and modify the SRCREV value, As shown in the red string below:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/tf-a-myir:"
SECTION = "bootloaders"
LICENSE = "BSD-3-Clause"
LIC_FILES_CHKSUM = "file://license.rst;md5=1dd070c98a281d18d9eef938729b031"
SRC_URI += "git://github.com/MYiR-Dev/myir-st-arm-trusted-firmware.git;proto=https;branch=develop-yf13x-v2.6"
SRCREV= "4cc966a5372b13517343009b2f8797cb99828ce8"

TF_A_VERSION = "v2.6"
TF_A_SUBVERSION = "stm32mp"
TF_A_RELEASE = "r2"
PV = "${TF_A_VERSION}-${TF_A_SUBVERSION}-${TF_A_RELEASE}"

ARCHIVER_ST_BRANCH = "develop-yf13x-v2.6"
ARCHIVER_ST_REVISION = "${PV}"
ARCHIVER_COMMUNITY_BRANCH = "develop-v2.6-stm32mp"
```

```
ARCHIVER_COMMUNITY_REVISION = "${TF_A_VERSION}"

S = "${WORKDIR}/git"

# -----
# Configure devupstream class usage
# -----
BBCLASSEXTEND = "devupstream:target"

SRC_URI:class-devupstream = "git://github.com/MYiR-Dev/myir-st-arm-trusted-firmware.git;protocol=https;branch=${ARCHIVER_ST_BRANCH}"
SRCREV:class-devupstream = "4cc966a5372b13517343009b2f8797cb99828ce8"

# -----
# Configure default preference to manage dynamic selection between tarball
and github
# -----
STM32MP_SOURCE_SELECTION ?= "tarball"

DEFAULT_PREFERENCE = "${@bb.utils.contains('STM32MP_SOURCE_SELECTION',
'github', '-1', '1', d)}"
```

The following command is an example on how to build TF-A in Yocto Project :

```
PC$:bitbake tf-a-myir
```

In addition, you can also rebuild the whole system according to chapter 3.3.

5.3.3. How to update TF-A separately

This section describes how to manually update the TF-A binaries(fsbl) to the boot storage device.The prerequisite is that the TF binary has been generated and the storage device has been partitioned.If it is not partitioned, the partition can be completed by burning the system according to section 3.5.1.

- **Burn to the specified partition in SD card**

/dev/mmcblk Partition information

```
/dev/mmcblk0p1 -- tf-a-myb-stm32mp135x-512m-sdcard.stm32
/dev/mmcblk0p2 -- tf-a-myb-stm32mp135x-512m-sdcard.stm32
/dev/mmcblk0p3 -- metadata.bin
/dev/mmcblk0p4 -- metadata.bin
/dev/mmcblk0p5 -- fip-myb-stm32mp135x-512m-optee.bin
/dev/mmcblk0p6 -- none
/dev/mmcblk0p7 -- uboot env
/dev/mmcblk0p8 --bootfs
/dev/mmcblk0p9 --vendorfs
/dev/mmcblk0p10 --rootfs
/dev/mmcblk0p11 --userfs
```

You can use the Linux dd command to copy the FSBL1 and FSBL2 directly to the correct partition:

```
PC$: dd if=tf-a-myb-stm32mp135x-512m-sdcard.stm32 of=/dev/mmcblk0p1 co
nv=fdatasync
PC$: dd if=tf-a-myb-stm32mp135x-512m-sdcard.stm32 of=/dev/mmcblk0p2 co
nv=fdatasync
```

- **Burn to specified partition in eMMC**

Note: the following operation commands are executed on the development board. Once the command is executed incorrectly or the file is written to the mismatched partition, the development board will not be able to boot. In case of failure to boot, please re burn the development board.

The following command is an example on how to write to eMMC target partition via dd command,since the boot partition has been set with read-only protection, it is necessary to remove the read-only protection before writing:

```
root@myir~#: echo 0 > /sys/class/block/mmcblk1boot1/force_ro
root@myir~#: echo 0 > /sys/class/block/mmcblk1boot2/force_ro
```

```
root@myir~#: dd if=tf-a-myb-stm32mp135x-512m-sdcard.stm32 of=/dev/mm
cblk1boot1 conv=fdatasync
root@myir~#: dd if=tf-a-myb-stm32mp135x-512m-sdcard.stm32 of=/dev/mm
cblk1boot2 conv=fdatasync
root@myir~#: echo 1 > /sys/class/block/mmcblk1boot1/force_ro
root@myir~#: echo 1 > /sys/class/block/mmcblk1boot2/force_ro
```

- **Burn to specified partition in NAND**

/dev/mtd Indicates the partition information

```
/dev/mtdblock0 -- tf-a-myb-stm32mp135x-256m-sdcard.stm32
/dev/mtdblock1 -- tf-a-myb-stm32mp135x-256m-sdcard.stm32
/dev/mtdblock2 -- metadata.bin
/dev/mtdblock3 -- metadata.bin
/dev/mtdblock4 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock5 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock6 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock7 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock8 --myir-image-core-openstlinux-weston-myd-yf13x_spinand_2_12
8.ubi
```

Burn to the designated partition in the core board NAND storage:

```
root@myir~#: nandwrite -p /dev/mtd0 -p tf-a-myb-stm32mp135x-512m-sdc
ard.stm32
root@myir~#: nandwrite -p /dev/mtd1 -p tf-a-myb-stm32mp135x-512m-sdc
ard.stm32
```

5.4. U-boot Compilation and Update

U-Boot ("the Universal Boot Loader" or U-Boot) is an open-source bootloader that can be used on ST boards to initialize the platform and load the Linux kernel. it is widely used in embedded system, do go further, please refer to the official site:

<http://www.denx.de/wiki/U-Boot/WebHome> .

The STM32 MPU boot chain uses Trusted Firmware-A (TF-A) as FSBL and U-Boot as SSBL, and two different boot chain are supported depending on the configuration:

- **Basic boot chain:** The basic boot chain is also available to generate both the FSBL (U-Boot SPL) and SSBL (U-Boot) from a unique piece of source code, U-Boot.
- **Trusted boot chain:** The trusted boot chain uses Trusted Firmware-A (TF-A) as the FSBL.

Please refer to the official wiki for more details:https://wiki.st.com/stm32mpu/wiki/Boot_chain_overview.

This chapter describes how to download, build, and load the ST U-Boot in a standalone environment and through the Yocto Project.

5.4.1. How to build U-Boot in standalone environment

1) Get U-boot source code

Copy *04_Sources/MYiR-STM32-u-boot.tar.bz2* file to the specific work directory of linux(eg:\$HOME/work), extract tarball of U-Boot and view the corresponding file :

```
PC$ cd $HOME/work
PC$ tar -jxvf MYiR-STM32-u-boot.tar.bz2
PC$ cd MYiR-STM32-u-boot
```

- Source directory: myir-st-u-boot
- Compiled script: Makefile.sdk
- Instruction: README.HOW_TO.txt

```
-rwxrwxrwx 1 root root 3816 Jun  9 20:51 Makefile.sdk
drwxr-xr-x 25 root root 4096 Sep 18 01:16 myir-st-u-boot
-rw----- 1 licy licy 10579 Apr 21 03:46 README.HOW_TO.txt
```

1) Configuration and Compilation

On the host machine, set the environment with the following command before building for STM32MP1 SoC :

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

Then find the configuration for the target boot in the configs/ directory of the U-boot source code. In the following two examples, stm32mp135 is the target.

Compiling uboot source code manually:

- Go into the source code

```
PC$ cd myir-st-u-boot
```

- Create a compilation output folder

```
PC$ export KBUILD_OUTPUT=../build-trusted
```

- Configure and Compile source code

(1) Configure compilation in the source directory :**(this is not recommended)**

```
PC$ make myc_stm32mp13_512m_defconfig
```

```
PC$ make DEVICE_TREE=myb-stm32mp135x-512m all
```

(2) Use the Makefile.SDK script to configure the compilation mode:**(Recommended)**

```
PC$ make -f $PWD/./Makefile.sdk all
```

The compilation output folder of the two methods is consistent, and users can directly enter the output folder, you will find the compiled output binaries, as shown in the following table:

```
PC$ cd ../build-trusted
```

Table 5-4.U-boot binary file

Name	Description
u-boot-stm32mp13.elf	ELF file uboot, can be used for GDB debugging

5.4.2. How to build and load U-Boot in Yocto Project

If you want to set up git repository to manage the code and build a new image completely according to the process in Chapter 3, first, you need to submit the

modified U-boot source code to git repository, and then modify the corresponding source pull address(SRC_URI) and source revision(SRCREV). Take GitHub repository as an example:

First, you need to enter the U-Boot source code directory to submit the modified U-Boot source code, and then find the latest commit value in GitHub repository and copy the value, the following command is an example on how to get the latest commit value in the u-boot source code:

```
PC$: git log --pretty=oneline
7e709a983f0816e63e8c13387d66f01225f01848 (HEAD -> develop-v2021.10-stm
32mp, tag: V0.1.0_20230510_Alpha, origin/develop-v2021.10-stm32mp, origin/H
EAD) FEAT: modify the boot parameter to
```

Next, go to *layers/meta-myir-st/recipes-bsp/u-boot/* directory, then open the recipe *u-boot-myir-common_2021.10.inc* and modify the SRCREV value, as shown in the red string below:

```
# Adaptation from u-boot-common_${PV}.inc

HOMEPAGE = "http://www.denx.de/wiki/U-Boot/WebHome"
SECTION = "bootloaders"

LICENSE = "GPL-2.0-or-later"
LIC_FILES_CHKSUM = "file://Licenses/README;md5=5a7450c57ffe5ae63fd73244
6b988025"

DEPENDS += "dtc-native bc-native"
DEPENDS += "flex-native bison-native"
DEPENDS += "python3-setuptools-native"

COMPATIBLE_MACHINE = "(stm32mpcommon)"
```

```

SRC_URI = "git://github.com/MYiR-Dev/myir-st-u-boot.git;protocol=https;branch
=${SRCBRANCH}"
SRCREV  = "7e709a983f0816e63e8c13387d66f01225f01848"
SRCBRANCH = "develop-yf13x-v2021.10"
# debug and trace
SRC_URI += "${@bb.utils.contains('ST_UNBOOT_DEBUG_TRACE', '1', "", 'file://0098
-silent_mode.patch', d)}"

U_BOOT_VERSION = "v2021.10"
U_BOOT_SUBVERSION = "stm32mp"
U_BOOT_RELEASE = "r2"

PV = "${U_BOOT_VERSION}-${U_BOOT_SUBVERSION}-${U_BOOT_RELEASE}"

ARCHIVER_ST_BRANCH = "develop-${U_BOOT_VERSION}-${U_BOOT_SUBVERSION}"
ARCHIVER_ST_REVISION = "${PV}"
ARCHIVER_COMMUNITY_BRANCH = "YF13X_V2021.10"
ARCHIVER_COMMUNITY_REVISION = "${U_BOOT_VERSION}"

S = "${WORKDIR}/git"

# -----
# Configure devupstream class usage
# -----
BBCASSEXTEND = "devupstream:target"

SRC_URI:class-devupstream = "git://github.com/MYiR-Dev/myir-st-u-boot.git;pro
tocol=https;branch=${ARCHIVER_ST_BRANCH}"
SRCREV:class-devupstream = "7e709a983f0816e63e8c13387d66f01225f01848"

# -----

```



```
# Configure default preference to manage dynamic selection between tarball
and github
#
STM32MP_SOURCE_SELECTION ?= "tarball"

DEFAULT_PREFERENCE = "${@bb.utils.contains('STM32MP_SOURCE_SELECTION',
'github', '-1', '1', d)}"
```

The following command is an example on how to build U-Boot in Yocto Project :

```
PC$ bitbake u-boot-mir
```

In addition, you can also rebuild the whole system according to chapter 3.3.

Note: If you want to set up your own git repository to manage the code, in addition to modifying the revision value(SRCREV), you should also modify the source pull address(SRC_URI)

5.4.3. How to update U-Boot separately

This section describes how to manually update the U-boot binaries(ssbl) to the boot storage device. The prerequisite is that the U-boot binary has been generated and the storage device has been partitioned.If it is not partitioned, the partition can be completed by burning the system according to section 3.5.1.

- **Burn to the specified partition in SD card**

Note the /dev/mmcblk partition information

```
/dev/mmcblk0p1 -- tf-a-myb-stm32mp135x-512m-sdcard.stm32
/dev/mmcblk0p2 -- tf-a-myb-stm32mp135x-512m-sdcard.stm32
/dev/mmcblk0p3 -- metadata.bin
/dev/mmcblk0p4 -- metadata.bin
/dev/mmcblk0p5 -- fip-myb-stm32mp135x-512m-optee.bin
/dev/mmcblk0p6 -- none
/dev/mmcblk0p7 -- uboot env
/dev/mmcblk0p8 --bootfs
/dev/mmcblk0p9 --vendorfs
```

```
/dev/mmcblk0p10 --rootfs  
/dev/mmcblk0p11 --userfs
```

To burn the boot image to the SD card, execute the following command:

```
PC$: dd if=fip-myb-stm32mp135x-512m-optee.bin of=/dev/mmcblk0p3 conv=f  
datasync
```

- **Burn to specified partition in eMMC**

Note: the following operation commands are executed on the development board. Once the command is executed incorrectly or the file is written to the mismatched partition, the development board will not be able to boot. In case of failure to boot, please re burn the development board.

The following command is an example on how to write to eMMC target partition via dd command:

```
root@myir~#: dd if=fip-myb-stm32mp135x-512m-optee.bin of=/dev/mmcblk1  
p1 conv=fdatasync
```

- **Burn to specified partition in NAND**

Note the /dev/mtd partition information

```
/dev/mtdblock0 -- tf-a-myb-stm32mp135x-256m-sdcard.stm32
/dev/mtdblock1 -- tf-a-myb-stm32mp135x-256m-sdcard.stm32
/dev/mtdblock2 -- metadata.bin
/dev/mtdblock3 -- metadata.bin
/dev/mtdblock4 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock5 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock6 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock7 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock8 --myir-image-core-openstlinux-weston-myd-yf13x_spinand_2_12
8.ubi
```

Burn to the designated partition in the core board NAND storage:

```
root@myir~#:nandwrite -p /dev/mtd4 -p fip-myb-stm32mp135x-256m-optee.
bin
root@myir~#:nandwrite -p /dev/mtd5 -p fip-myb-stm32mp135x-256m-optee.
bin
root@myir~#:nandwrite -p /dev/mtd6 -p fip-myb-stm32mp135x-256m-optee.
bin
root@myir~#:nandwrite -p /dev/mtd7 -p fip-myb-stm32mp135x-256m-optee.
bin
```

5.5. Kernel Compilation and Update

Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single UNIX Specification compliance.

It has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-

on-write executables, proper memory management, and multistack networking including IPv4 and IPv6.

It is distributed under the GNU General Public License v2 - see the accompanying COPYING file for more details.

Linux kernel is also a very large open source kernel, which is applied to various distribution operating systems. Linux kernel is widely used in embedded system with its portability, network protocol support, independent module mechanism, MMU and other rich characteristics.Linux kernel.

At the same time, stm32mp1 also supports the Linux kernel and has been added to the kernel mainline. It will be updated stably for a long time. Please check the kernel mainline for the latest version: <https://www.kernel.org/>, MYD-YF13X adapts to ST open source community version kernel version, and currently supports the latest version of Linux kernel 5.15.67.

5.5.1. How to build Kernel in standalone environment

1) Get the Linux source code

Copy *04_Sources/MYiR-STM32-kernel.tar.bz2* file to the specific work directory of linux(eg:\$HOME/work),extract tarball of Kernel and view the corresponding file :

```
PC$ cd $HOME/work
PC$ tar -jxf MYiR-STM32-kernel.tar.bz2
PC$ cd MYiR-STM32-kernel
```

Catalog contains:

- Source directory:myir-st-linux
- Instruction:README.HOW_TO.txt

```
drwxr-xr-x 26 root root 4096 Aug 31 04:50 myir-st-linux
-rw----- 1 root root 15366 Jun 9 22:49 README.HOW_TO.txt
```

2) Configure Kernel Source Code

- Go to the kernel source directory

```
PC$ cd myir-st-linux
```

- Create a compilation output folder

```
PC$ mkdir -p ./build
```

- Configure kernel

```
PC$ make ARCH=arm O="$PWD/../build" myir_stm32mp135x_defconfig
```

If you need to add a driver function of the kernel, you can also use the following methods.

```
PC$ cd build
```

```
PC$ make menuconfig
```

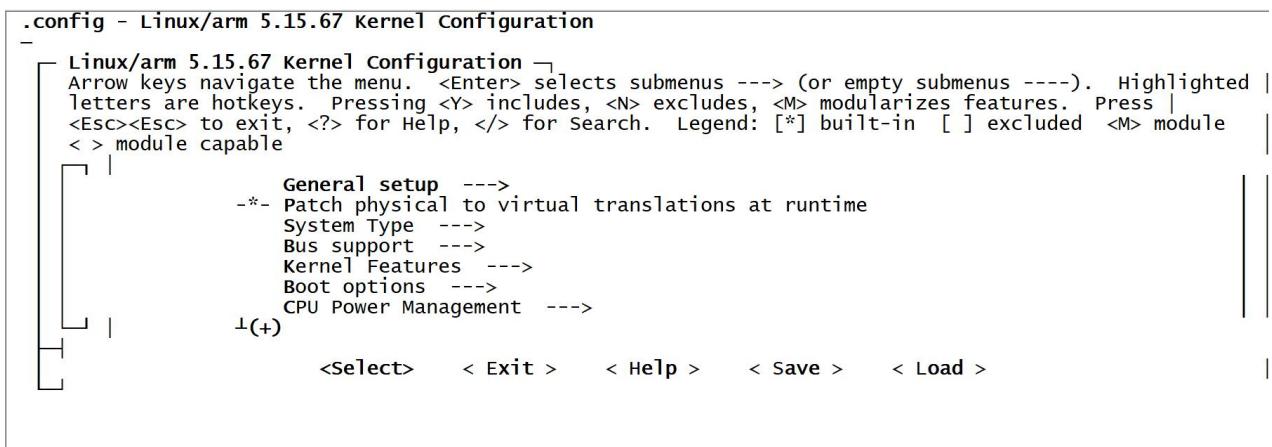


Figure 5-1. MenuConfig interface

3) Compile Kernel Source Code

- Set up the host terminal window toolchain environment

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

- Initialize the configuration

```
PC$ make ARCH=arm O="$PWD/../build" myir_stm32mp135x_defconfig
```

- Build the kernel sources

```
PC$ make ARCH=arm ulimage vmlinux dtbs LOADADDR=0xC2000040 O="$PWD/../.build"
PC$ make ARCH=arm modules O="$PWD/../.build"
```

Be patient for the compilation to complete.

- Install the output file to the specified directory

```
PC$ make ARCH=arm INSTALL_MOD_PATH="$PWD/../.build/install_artifact" modules_install O="$PWD/../.build"
PC$ mkdir -p $PWD/../.build/install_artifact/boot/
PC$ cp $PWD/../.build/arch/arm/boot/ulimage $PWD/../.build/install_artifact/boot/
PC$ cp $PWD/../.build/arch/arm/boot/dts/my*.dtb $PWD/../.build/install_artifact/boot/
```

- build/install_artifact/lib: The directory stores the kernel driver modules.
- build/install_artifact/boot: The directory stores the device tree and kernel binary files .

4) List of generated files

Table 5-5.Flash files list

Type	Flash Files	Flash Files Directory
Kernel image	ulimage	build/install_artifact/boot
Device Tree	myb-stm32mp135x-256m.dtb myb-stm32mp135x-512m.dtb	build/install_artifact/boot
Kernel modules	modules	build/install_artifact/lib/modules

Table 5-6.Device Tree List

Device Tree	SOM Verion	Notes
myb-stm32mp135x-512m.dtb	MYC-YF135-4E512D-100-I	The Flash is EMMC
myb-stm32mp135x-256m.dtb	MYC-YF135-256N256D-100-I	The Flash is NAND

5.5.2. How to build Linux in Yocto Project

You need to enter the Kernel source code directory to submit the modified Kernel source code, and then find the latest commit value in GitHub repository and copy the value, the following command is an example on how to get the latest commit value in the Kernel source code:

```
PC$ git log --pretty=oneline
9030618a5710463c8b96b3a8523ace9075fde00d (HEAD -> develop-stm32mp-L5.
15, tag: V0.1.0_20230510_Alpha, origin/develop-stm32mp-L5.15, origin/HEAD) F
EAT: add MY-RGB2HDMI support
```

Next, go to *layers/meta-myir-st/recipes-kernel/linux/* directory, then open the recipe *linux-myir_5.15.bb* and modify the SRCREV value, as shown in the red string below:

```
SUMMARY = "Linux STM32MP Kernel"
SECTION = "kernel"
LICENSE = "GPL-2.0-only"
#LIC_FILES_CHKSUM = "file://COPYING;md5=bbea815ee2795b2f4230826c0c6b8
814"
LIC_FILES_CHKSUM = "file://COPYING;md5=6bc538ed5bd9a7fc9398086aedcd7e
46"

include linux-myir.inc

LINUX_VERSION = "5.15"
LINUX_SUBVERSION = "67"
LINUX_TARNAME = "linux-${LINUX_VERSION}.${LINUX_SUBVERSION}"

SRC_URI = "git://github.com/MYiR-Dev/myir-st-linux.git;protocol=https;branch=
${SRCBRANCH}"
SRCREV = "9030618a5710463c8b96b3a8523ace9075fde00d"
SRCBRANCH = "develop-yf13x-L5.15"
```

```
LINUX_TARGET = "stm32mp"
LINUX_RELEASE = "r2"

#PV = "${LINUX_VERSION}.${LINUX_SUBVERSION}-${LINUX_TARGET}-${LINUX_RELEASE}"
PV = "${LINUX_VERSION}.${LINUX_SUBVERSION}"

ARCHIVER_ST_BRANCH = "v${LINUX_VERSION}-${LINUX_TARGET}"
ARCHIVER_ST_REVISION = "v${LINUX_VERSION}-${LINUX_TARGET}-${LINUX_RELEASE}"
ARCHIVER_COMMUNITY_BRANCH = "linux-${LINUX_VERSION}.y"
ARCHIVER_COMMUNITY_REVISION = "v${LINUX_VERSION}.${LINUX_SUBVERSION}"

#S = "${WORKDIR}/linux-${LINUX_VERSION}.${LINUX_SUBVERSION}"
S = "${WORKDIR}/git"

# -----
# Configure devupstream class usage
# -----
BBCLASSEXTEND = "devupstream:target"

SRC_URI:class-devupstream = "git://github.com/MYiR-Dev/myir-st-linux.git;proto=https;branch=develop-yf13x-L5.15"
SRCREV:class-devupstream = "9030618a5710463c8b96b3a8523ace9075fde00d"

# -----
# Configure default preference to manage dynamic selection between tarball
and github
# -----
STM32MP_SOURCE_SELECTION ?= "tarball"
```

```

DEFAULT_PREFERENCE = "${@bb.utils.contains('STM32MP_SOURCE_SELECTION',
'github', '-1', '1', d)}"

# -----
# Configure archiver use
# -----
include ${@oe.utils.ifelse(dgetVar('ST_ARCHIVER_ENABLE') == '1', 'linux-myir-ar
chiver.inc','')}

# -----
# Defconfig
#
KERNEL_DEFCONFIG      = "myir_stm32mp135x_defconfig"
KERNEL_CONFIG_FRAGMENTS = "${@bb.utils.contains('KERNEL_DEFCONFIG', 'de
fconfig', '${S}/arch/arm/configs/fragment-01-multiv7_cleanup.config', "", d)}"
KERNEL_CONFIG_FRAGMENTS += "${@bb.utils.contains('KERNEL_DEFCONFIG', 'de
fconfig', '${S}/arch/arm/configs/fragment-02-multiv7_addons.config', "", d)}"
#KERNEL_CONFIG_FRAGMENTS += "${@bb.utils.contains('DISTRO_FEATURES', 's
ystemd', '${WORKDIR}/fragments/${LINUX_VERSION}/fragment-03-systemd.confi
g', "", d)} "
#KERNEL_CONFIG_FRAGMENTS += "${WORKDIR}/fragments/${LINUX_VERSION}
/fragment-04-modules.config"
#KERNEL_CONFIG_FRAGMENTS += "${@oe.utils.ifelse(dgetVar('KERNEL_SIGN_E
NABLE') == '1', '${WORKDIR}/fragments/${LINUX_VERSION}/fragment-05-signat
ure.config','')} "
#KERNEL_CONFIG_FRAGMENTS += "${@bb.utils.contains('MACHINE_FEATURES',
'nosmp', '${WORKDIR}/fragments/${LINUX_VERSION}/fragment-06-smp.config',
", d)} "

```

The following command is an example on how to build Kernel in Yocto Project :

PC\$ bitbake linux-myir

In addition, you can also rebuild the whole system according to chapter 3.3.

Note: If you want to set up your own git repository to manage the code, in addition to modifying the revision value(SRCREV), you should also modify the source pull address(SRC_URI).

5.5.3. How to update Kernel separately

By default, U-Boot loads kernel image and device tree blob from the bootfs partition. Users can copy their images to this partition, for example, Users only need to transfer the generated uimage and DTB binary files to the /boot partition of the development board (SD card boot mode).However, if you need to update the driver module, you only need to transfer the kernel module file to the /lib/modules directory of the root partition. After the transmission is completed, execute the commands(sync & reboot) to complete synchronization and restart.

This section describes several ways to update the kernel separately.

1) Update via network

Take wired Ethernet update as an example. Connect the compiled server and the development board to the same router at the same time. For example, the Ethernet IP of the development board is 192.168.30.100. The IP of the compilation server is 192.168.30.101.Then perform the following steps:

Enter the output file directory(If you are not clear about the output directory, please refer to section 5.5.1):

```
cd build/install_artifact/
```

Copy kernel and devicetree:

```
PC$ scp -r boot/* root@192.168.30.100:/boot/
```

Remove the link on *install_artifact/lib/modules/<kernel version>/*.

```
PC$ rm lib/modules/<kernel version>/source lib/modules/<kernel version>/buil  
ld
```

Copy kernel modules:

```
PC$ scp -r lib/modules/* root@192.168.30.100:/lib/modules/
```

2) Update with USB virtual network

Connect USB Type A to Type C cable between PC and USB_OTG port of the MYD-YF13X board. The USB device will be treated as a network card device named usb0. The default IP address is 192.168.7.1. Then perform the following steps:

Enter the output file directory (If you are not clear about the output directory, please refer to section 5.5.1):

```
cd build/install_artifact/
```

Copy kernel and device tree:

```
PC$ scp -r boot/* root@192.168.7.1:/boot/
```

Remove the link on install_artifact/lib/modules/<kernel version>/:

```
PC$ rm lib/modules/<kernel version>/source lib/modules/<kernel version>/buil  
ld
```

Copy kernel modules:

```
PC$ scp -r lib/modules/* root@192.168.7.1:/lib/modules/
```

3) Update via SDCARD on your BOARD (via U-Boot)

You must configure first, via U-Boot, the board into usb mass storage, On U-Boot shell:ums <USB controller> <dev type: mmc|usb> <dev[:part]>. For example: the following commands lists various mass storage device:

```
For SDCARD:    ums 0 mmc 0
```

```
For USB Disk:  ums 0 usb 0
```

Plug the SDCARD on Board and then start the board and stop on U-boot shell:

```
Hit any key to stop autoboot: 0
STM32MP>
```

plug an USB cable between the PC and the board via USB OTG port. On U-Boot shell, call the usb mass storage functionnality:

```
STM32MP> ums 0 mmc 0
UMS: LUN 0, dev mmc 0, hwp 0, sector 0x0, count 0x3b72400
```

Then the compiler server will automatically recognize the partition information on mmc0. The SD card partition will also be automatically mounted to the server media partition, as shown below:

```
root@ubuntu:/media/nene# ls -all
total 15
drwxr-x---+ 6 root root 4096 Oct 15 19:37 .
drwxr-xr-x  4 root root 4096 Aug 13 20:57 ..
drwxr-xr-x  5 root root 1024 Oct 13 18:53 bootfs
drwxr-xr-x 21 root root 4096 Feb  7  2020 rootfs
drwxr-xr-x  7 root root 1024 Sep 30 01:53 userfs
drwxr-xr-x  4 root root 1024 Sep 30 01:51 vendorfs
```

Enter the output file directory (If you are not clear about the output directory, please refer to section 5.5.1):

```
cd build/install_artifact/
```

Copy kernel and devicetree:

```
PC$ cp -rf boot/* /media/$USER/bootfs/
```

Remove the link on *install_artifact/lib/modules/<kernel version>/*.

```
PC$ rm lib/modules/<kernel version>/source lib/modules/<kernel version>/buil  
ld
```

Copy kernel modules:

```
PC$ cp -rf lib/modules/* /media/$USER/rootfs/lib/modules/
```

Note: \$user is the user name of the compilation server, and media is the mount directory.

4) Update via SDCARD on Your Linux PC

Update via SDCARD on your Linux PC. Insert the Micro SD card into the computer with the card reader of a PC. Verify the Micro SD card are mounted on your Linux PC: /media/\$USER/bootfs(Some systems can't mount automatically and need to mount bootfs partition manually).

```
root@ubuntu:/media/nene# ls -all  
total 15  
drwxr-x---+ 6 root root 4096 Oct 15 19:57 .  
drwxr-xr-x  4 root root 4096 Aug 13 20:57 ..  
drwxr-xr-x  5 root root 1024 Oct 13 18:53 bootfs  
drwxr-xr-x 21 root root 4096 Feb  7  2020 rootfs  
drwxr-xr-x  7 root root 1024 Sep 30 01:53 userfs  
drwxr-xr-x  4 root root 1024 Sep 30 01:51 vendorfs
```

Enter the output file directory(If you are not clear about the output directory, please refer to section 5.5.1):

```
cd build/install_artifact/
```

Copy kernel and devicetree:

```
PC$ cp -rf boot/* /media/$USER/bootfs/
```

Remove the link on *install_artifact/lib/modules/<kernel version>/*:

```
PC$ rm lib/modules/<kernel version>/source lib/modules/<kernel version>/buil  
ld
```

Copy kernel modules:

```
PC$ cp -rf lib/modules/* /media/$USER/rootfs/lib/modules/
```

Note: \$user is the user name of the compilation server, and media is the mount directory.

6. How to Fit Your Hardware Platform

In order to adapt to the new hardware platform of users, it is necessary to know what resources are provided by MYD-YF13X development board of MYIR. For specific information, please refer to "MYD-YF13X SDK Release Notes". In addition, users also need to refer to the CPU chip manual and the product manual of MYC-YF13X module, so as to have a more detailed understanding of the CPU pin definition and CPU performance, and finally be able to correctly configure and use these pins according to the actual functions.

6.1. How to Create Your Device Tree

Follow the sequences described in the below chapters to create the device-tree on your board.

6.1.1. Board Level Device Tree

Users can create their own device tree in BSP source code. Generally, users do not need to modify TF-A and u-boot in bootloader and you just need to adjust the Linux kernel device tree according to the actual hardware resources. The following table lists various key device trees of MYD-YF13X board, which is very helpful for the development reference of users.

Table 6-1.MYD-YF13X Device-tree List

Project	Device Tree	Description
Tf-a	myb-stm32mp135x-base.dts	Peripheral resource device tree
	myb-stm32mp135x-256m.dts	Device tree for CPU module
	myb-stm32mp135x-256m-fw-config.dts	
	myb-stm32mp135x-512m.dts	
	myb-stm32mp135x-512m-fw-config.dts	
	stm32mp13-ddr3-1x2Gb-1066-binF.dtis	Description of dual chip 256MB DDR3
U-boot	stm32mp13-ddr3-1x4Gb-1066-binF.dtis	Description of dual chip 512MB DDR3
	myb-stm32mp135x-base.dts	Peripheral resource device tree
	myb-stm32mp135x-256m.dts	Device tree for CPU module

	myb-stm32mp135x-256m-u-boot.dtsi myb-stm32mp135x-512m.dts myb-stm32mp135x-512m-u-boot.dtsi	
	stm32mp13-ddr3-1x2Gb-1066-binF.dtsi stm32mp13-ddr3-1x4Gb-1066-binF.dtsi	Description of dual chip 256MB DDR3 Description of dual chip 512MB DDR3
	myb-stm32mp13-pinctrl.dtsi	Pin definition of stm32mp135series processor
Kernel	myir_stm32mp135x_defconfig	Kernel configuration file
	myb-stm32mp135f.dts	Peripheral resource device tree
	myb-stm32mp135x-base.dtsi	
	myb-stm32mp135x-256m.dts	256MB DDR3 device tree
	myb-stm32mp135x-512m.dts	512MB DDR3 device tree
	myb-stm32mp13-pinctrl.dtsi	Pin definition of MYD-YF13X board

6.1.2. Add your board level device tree

A device tree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent. ... Rather than hard coding every detail of a device into an operating system, many aspect of the hardware can be described in a data structure that is passed to the operating system at boot time. In other words, a device tree describes the hardware that can not be located by probing. Then perform the following steps to add your device tree.

- **Add board level device tree**

Go to the arch/arm/boot/dts kernel device tree directory, you will find the device tree files that are suitable for various platforms, and then add your own board level device tree files, eg: *myb-stm32mp135f-xxx.dts*.

```
// Path: arch/arm/boot/dts
-rwxrwxr-x 1 nene nene 14761 5月 7 11:56 myb-stm32mp135f.dts
-rwxrwxr-x 1 nene nene 1622 5月 7 11:56 myb-stm32mp135x-256m.dts
```

```
-rwxrwxr-x 1 nene nene 869 5月 7 11:56 myb-stm32mp135x-512m.dts
-rwxrwxr-x 1 nene nene 14332 5月 7 11:56 myb-stm32mp135x-base.dtsi
-rwxrwxr-x 1 nene nene 17899 5月 7 11:56 myb-stm32mp13-pinctrl.dtsi
```

- Fill in your device tree content

Next, you need to include some device tree header files files into your newly created board level device tree,as follows:

```
// SPDX-License-Identifier: (GPL-2.0+ OR BSD-3-Clause)
/*
 * Copyright (C) MYiR 2022 - All Rights Reserved
 * Author: Alexhu <fan.hu@myirtech.com> for MYiR.
 */

/dts-v1/;

#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/input/input.h>
#include <dt-bindings/leds/common.h>
#include <dt-bindings/rtc/rtc-stm32.h>
#include "stm32mp135.dtsi"
#include "stm32mp13xf.dtsi"
#include "myb-stm32mp13-pinctrl.dtsi"

{
    model = "STMicroelectronics STM32MP135F-DK Discovery Board";
    compatible = "st,stm32mp135f-dk", "st,stm32mp135";

    aliases {
        ethernet0 = &eth1;
        ethernet1 = &eth2;
        serial0 = &uart4;
        serial1 = &uart7;
```

```

        serial2 = &uart5;
    };

chosen {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
    stdout-path = "serial0:115200n8";
}
    
```

For more detailed header file inclusion, please refer to our device tree file:*stm32mp135.dts*.

- **Add your device tree file to makefile**

After adding a new device tree source file, users need to add device tree compilation information in makefile under the same directory, so that the corresponding device tree binary file can be generated when compiling the device tree.

```

// File: arch/arm/boot/dts/Makefile
dtb-$(CONFIG_ARCH_STM32) += \
//      .....
      myb-stm32mp135x-512m.dtb \
      myb-stm32mp135x-256m.dtb \
      myb-stm32mp135f-xxx.dtb \
    
```

After adding the device tree, you can continue to fill the device tree according to your hardware resources, then compile the DTB file *myb-stm32mp135f-xxx.dtb*, please refer to section 5.5 for compilation. The above process is the process of creating a new device tree file. However, after adding a new device tree, it is necessary to modify the name of the loading file in the u-boot and modify the yocto configuration file and metadata. Therefore, it is recommended that users modify the MYIR device tree directly.

6.2. How to configure function pins according to your hardware

Realizing the control of a function pin is one of the more complex system development processes, including pin configuration, driver development, application realization and other steps. This section does not specifically analyze the development process of each part, but explains the control implementation of function pin with examples.

6.2.1. GPIO pin configuration

GPIO refers to the general input and output port, which is a very important resource in embedded devices. It can output high and low level or read the status of pins - high level or low level.

The STM32MP135 devices encapsulates a large number of peripheral controllers. These peripheral controllers and external devices are generally implemented by controlling GPIO. The use of GPIO by peripheral controllers is called multiplexing, which gives them more complex functions. The operation of the functional pin, which can be subdivided into either major function or one alternate function, is controlled by a specific hardware module. If it is configured as a GPIO pin, the pin is controlled by the user through software with further configuration through the GPIO module. For example, If this pin is connected to an external UART transceiver, it should be configured as the primary function or if this pin is connected to an external Ethernet controller for interrupting the core, then it should be configured as GPIO input pin with interrupt enabled.

The GPIO pin configuration of stm32mp135 device is generally configured by stm32ubemx software or manually by referring to datasheet.

1) How to configure peripherals using CubeMX

At present, ST has added STM32MPU series CPU to STM32CubeMX. We can also use this tool to configure TF-A, U-boot and Kernel GPIO function device tree and

peripheral clock. This section does not focus on how to use it. You can obtain detailed development guidance from the official website.

WIKI: <https://wiki.st.com/stm32mpu/wiki/STM32CubeMX>

ST official: <https://www.st.com/zh/development-tools/stm32cubemx.html>

2) How to configure manually

You may need to refer to the file

(01_Documents|Datasheet|STM32MP135DAF7.pdf) and the MYC-YF13X pin manifest file(01_Documents|HardwareFiles |MYD-YF13X|MYC-YF135 Pin List V1.0) during manual configuration. The general manual configuration process requires the following steps:

- **Pin multiplexing formula**

The pin multiplexing of GPIO on STM32MP135 follows the formula below:

```
#define PIN_NO(port, line) (((port) - 'A') * 0x10 + (line))
#define STM32_PINMUX(port, line, mode) (((PIN_NO(port, line)) << 8) | (mode))
    - port: The gpio port index (GPIOA = 'A' , PB = 'B' , ..., GPIOZ = 'Z' )
        - line: The line offset within the port (PA0 = 0, PA1 = 1, ..., PA15 = 15)
            - mode: The function number, can be:
                * 0 : GPIO
                * 1 : Alternate Function 0
                * 2 : Alternate Function 1
                * 3 : Alternate Function 2
                * ...
                * 16 : Alternate Function 15
                * 17 : Analog
                * 18 : Reserved
```

Where '0' represents the universal number GPIO;"1-15" means multiplexing function AF0-AF15;"Analog" means Analog signal;"Reserved" means Reserved.

● Pinmux configuration

Take GPIOF14 on the extension interface as an example to illustrate the configuration method of pin function.

```
/* GPIO F14 set as alternate function 5 */
    ...
    pinmux = <STM32_PINMUX('F', 14, AF5)>;//set i2c
};

/* GPIO F14 set as GPIO */
    ...
    pinmux = <STM32_PINMUX('F', 14, GPIO)>;
};

/* GPIO F14 set as analog */
    ...
    pinmux = <STM32_PINMUX('F', 14, ANALOG)>;
};

/* GPIO F14 reserved for co-processor */
    ...
    pinmux = <STM32_PINMUX('F', 14, RSVD)>;
};
```

6.2.2. How to use GPIO in device tree

1) Configure the function pin as a GPIO function instance

This example describes how to configure the device node in the device tree, which is used by the kernel driver for later chapters. This example can also provide reference for controlling the reset of external equipment, power supply and other control functions. This section will take pf14 pin as an example to test GPIO:

Just add nodes to the device tree:

```
//myb-stm32mp135f-xxx.dtb
gpiocr_device {
    compatible = "myir,gpiocr";
    status = "okay";
    gpiocr-gpios = <&gpiof 14 0>;
};
```

6.3. How to use your own configured pins

The pin we configured in the u-boot or kernel device tree can be used in u-boot or kernel, so as to realize the control of the pins.

6.3.1. How to use GPIO in uboot

1) GPIO control through uboot command

In the uboot shell, you can directly use the command to control GPIO. For example, to set up gpiof14, use the following command in uboot shell.

```
STM32MP> gpio set GPIOF14
gpio: pin GPIOF14 (gpio 94) value is 1
STM32MP> gpio clear GPIOF14
gpio: pin GPIOF14 (gpio 94) value is 0
```

6.3.2. How to use GPIO in Kernel driver

1) How to use independent GPIO driver

In section 6.2.2, the GPIO device node information has been defined in the GPIO sample device tree. Next, we will use the kernel driver to realize the control of GPIO (set the PF14 pin to 1 or 0, and use a multimeter to test the change of pin level if necessary).

```
//gpioctr.c
#include <linux/module.h>
#include <linux/of_device.h>
#include <linux/fs.h>
```

```

#include <linux/errno.h>
#include <linux/miscdevice.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/mutex.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/stat.h>
#include <linux/init.h>
#include <linux/device.h>
#include <linux/tty.h>
#include <linux/kmod.h>
#include <linux/gfp.h>
#include <linux/gpio/consumer.h>
#include <linux/platform_device.h>

/* 1. Define master device number */
static int major = 0;
static struct class *gpiocr_class;
static struct gpio_desc *gpiocr_gpio;

/* 2. Implement the corresponding open/read/write functions and fill in the file_operations structure*/
static ssize_t gpio_drv_read (struct file *file, char __user *buf, size_t size, loff_t *offset)
{
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    return 0;
}

```

```

static ssize_t gpio_drv_write (struct file *file, const char __user *buf, size_t size,
loff_t *offset)
{
    int err;
    char status;

    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    err = copy_from_user(&status, buf, 1);

    gpiod_set_value(gpioctr_gpio, status);

    return 1;
}

static int gpio_drv_open (struct inode *node, struct file *file)
{
    gpiod_direction_output(gpioctr_gpio, 0);

    return 0;
}

static int gpio_drv_close (struct inode *node, struct file *file)
{
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    return 0;
}

/*Define your own file_operations structure*/
static struct file_operations gpioctr_drv = {
    .owner      = THIS_MODULE,
    .open       = gpio_drv_open,
    .read       = gpio_drv_read,

```

```
.write    = gpio_drv_write,
.release = gpio_drv_close,
};

/*get GPIO resources from platform_Device
 * Register driver */
static int chip_demo_gpio_probe(struct platform_device *pdev)
{
    /* Defined in device tree: gpiocr-gpios=<...>;      */
    gpiocr_gpio = gpiod_get(&pdev->dev, "gpiocr", 0);
    if (IS_ERR(gpiocr_gpio)) {
        dev_err(&pdev->dev, "Failed to get GPIO for led\n");
        return PTR_ERR(gpiocr_gpio);
    }

    /* Register file_operations      */
    major = register_chrdev(0, "myir_gpiocr", &gpiocr_drv); /* /dev/gpiocr */
}

gpiocr_class = class_create(THIS_MODULE, "myir_gpiocr_class");
if (IS_ERR(gpiocr_class)) {
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    unregister_chrdev(major, "gpiocr");
    gpiod_put(gpiocr_gpio);
    return PTR_ERR(gpiocr_class);
}

device_create(gpiocr_class, NULL, MKDEV(major, 0), NULL, "myir_gpiocr%d", 0);

return 0;
}
```

```
static int chip_demo_gpio_remove(struct platform_device *pdev)
{
    device_destroy(gpioctr_class, MKDEV(major, 0));
    class_destroy(gpioctr_class);
    unregister_chrdev(major, "myir_gpioctr");
    gpiod_put(gpioctr_gpio);

    return 0;
}

static const struct of_device_id myir_gpioctr[] = {
    { .compatible = "myir,gpioctr" },
    { },
};

/* define platform_driver */
static struct platform_driver chip_demo_gpio_driver = {
    .probe      = chip_demo_gpio_probe,
    .remove     = chip_demo_gpio_remove,
    .driver     = {
        .name      = "myir_gpioctr",
        .of_match_table = myir_gpioctr,
    },
};

/* Register platform_driver in entry function */
static int __init gpio_init(void)
{
    int err;
    err = platform_driver_register(&chip_demo_gpio_driver);
```

```

        return err;
    }

/* If there is an entry function, there should be an exit function: when the driver is unregister, the exit function will be called
   unregister platform_driver
*/
static void __exit gpio_exit(void)
{
    platform_driver_unregister(&chip_demo_gpio_driver);
}

/* Other improvements: provide equipment information and automatically create device nodes */
module_init(gpio_init);
module_exit(gpio_exit);

MODULE_LICENSE("GPL");

```

The driver code can be compiled into a module using a separate Makefile, or it can be directly configured into the kernel.

2) Driver samples are compiled directly into the kernel

Create a new *gpiocr.c* file in the sample folder of the kernel source code, then copy the above driver code, and modify *kconfig*, *makefile* and *myir_stm32mp135x_defconfig*.

Add configuration in kconfig file:

```

//linux/sample/Kconfig
config SAMPLE_GPIO
    tristate "this is a gpio test driver"
    depends on CONFIG_GPIOLIB

```

Edit makefile:

```
//linux/sample/Makefile
# SPDX-License-Identifier: GPL-2.0
# Makefile for Linux samples code

obj-$(CONFIG_SAMPLE_ANDROID_BINDERFS)    += binderfs/
...
obj-$(CONFIG_SAMPLE_GPIO)                  += gpioctr.o
```

Add the configuration item in *myc-ya157c-defconfig* file:

```
//linux/arch/arm/configs/myc-ya157c_defconfig
CONFIG_SAMPLES=y
CONFIG_SAMPLE_GPIO=y
CONFIG_SAMPLE_RPMMSG_CLIENT=m
```

Then compile and update the kernel according to section 5.5.3.

3) Compiling drivers outside the kernel source tree

Add the *gpioctr.c* file in the working directory and copy the above driver code to the file, and write the independent *Makefile* program in the same directory. As shown below:

```
# Modify KERN_DIR
#KERN_DIR = # The directory of the kernel source code used by the board
KERN_DIR = /home/nene/MYiR-stm32-linux/build/

obj-m += gpioctr.o

all:
    make -C $(KERN_DIR) M=`pwd` modules

clean:
```

```
make -C $(KERN_DIR) M=`pwd` modules clean
rm -rf modules.order

# If you want to compile a.c, b.c into ab.ko, To specify:
# ab-y := a.o b.o
# obj-m += ab.o
```

Then set up the host terminal window toolchain environment:

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-
neon-vfpv4-ostl-linux-gnueabi
```

Next, execute the make command to generate *gpiocr.ko* driver module file:

```
root@ubuntu:/home/myir# make
make -C /home/nene/MYiR-stm32-linux/build/ M=`pwd` modules
make[1]: Entering directory '/home/nene/MYiR-stm32-linux/build'
  CC [M]  /home/myir/gpiocr.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/myir/gpiocr.mod.o
  LD [M]  /home/myir/gpiocr.ko
make[1]: Leaving directory '/home/nene/MYiR-stm32-linux/build'
```

Finally, copy *gpiocr.ko* File to the */lib/modules* directory of the development board, and then use the *insmod* command to load the driver.

6.3.3. How to control a GPIO in Userspace

The architecture of Linux operating system is divided into user mode and kernel mode (or user space and kernel). User mode is the active space of the upper application. The execution of the application must rely on the resources provided by the kernel, including CPU resources, storage resources, I/O resources, etc. In

order to enable the upper application to access these resources, the kernel must provide the access interface for the upper application: system call.

However, shell is a special application program, commonly known as the command line. It is a command interpreter in essence. It passes through system calls and various applications. With shell scripts, a very large function can be realized in a few short shell scripts, because these shell statements usually encapsulate the system calls. In order to facilitate the interaction between users and the system.

This article shows three ways to control a GPIO in userspace:

- Shell command
- System call
- Library function

1) Realize pin control through shell command

Shell control pins are essentially implemented by calling the file operation interface provided by Linux. This section does not give a detailed description. Please refer to "MYD-YA157C_Linux_Software_Evaluation_Guide" , Section 3.1.

2) GPIO control through libgpiod

From Linux version 4.8, Linux introduces a new GPIO operation mode, GPIO character device. Each GPIO group has a corresponding gpiochip device node file under "/dev" directory, such as "/dev/gpiochip0 corresponds to GPIOA, /dev/gpiochip1 corresponds to GPIOB", etc.

libgpiod provides a C library and tools for interacting with the linux GPIO character device (gpiod stands for GPIO device).

For more descriptions, please refer to the following website:

ST official wiki:https://wiki.st.com/stm32mpu/wiki/How_to_control_a_GPIO_in_userspace#cite_note-1.

Libgpiod source code: <https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/>。

This application toggles GPIO PA14 (GPIO bank A, line 14):

```
//example-gpio.c
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/gpio.h>

int main(int argc, char **argv)
{
    struct gpiohandle_request req;
    struct gpiohandle_data data;
    char chrdev_name[20];
    int fd, ret;

    strcpy(chrdev_name, "/dev/gpiochip5");

    /* Open device: gpiochip5 for GPIO bank F */
    fd = open(chrdev_name, 0);
    if (fd == -1) {
        ret = -errno;
        fprintf(stderr, "Failed to open %s\n", chrdev_name);

        return ret;
    }

    /* request GPIO line: GPIO_F_14 */
```

```
req.lineoffsets[0] = 14;
req.flags = GPIOHANDLE_REQUEST_OUTPUT;
memcpy(req.default_values, &data, sizeof(req.default_values));
strcpy(req.consumer_label, "gpio_f_14");
req.lines = 1;

ret = ioctl(fd, GPIO_GET_LINEHANDLE_IOCTL, &req);
if (ret == -1) {
    ret = -errno;
    fprintf(stderr,"Failed to issue GET LINEHANDLE IOCTL (%d)\n",ret);
}
if (close(fd) == -1)
    perror("Failed to close GPIO character device file");

/* Start GPIO ctr*/
while(1) {
    data.values[0] = !data.values[0];
    ret = ioctl(req.fd, GPIOHANDLE_SET_LINE_VALUES_IOCTL, &data);
    if (ret == -1) {
        ret = -errno;
        fprintf(stderr,"Failed to issue %s (%d)\n", ret);
    }
    sleep(1);
}

/* release line */
ret = close(req.fd);
if (ret == -1) {
    perror("Failed to close GPIO LINEHANDLE device file");
    ret = -errno;
}
return ret;
```

}

Copy the above code to an *example-gpio.c* file and Initialize cross-compilation via SDK:

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

Use the compile command \$CC to generate the executable file example-gpio:

```
$CC example-gpio.c -o example-gpio
```

Finally, copy the executable file to the board directory (*/usr/bin*), the following command is an example on how to run directly:

```
root@myir:~# example-gpio
```

3) System call to realize pin control

A set of "special" interfaces provided by an operating system to a user program. The user program can obtain the services provided by the operating system kernel through this group of "special" interfaces, such as applying to open files, closing files or reading and writing files, and obtaining system time or setting timers through clock related system call.

At the same time, the pin is also a resource and can be controlled by system call. In section 6.3.2, we have completed the implementation of the pin driver. Now we can call and control the pin controlled by the driver.

```
//gpiotest.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
```

```
/*
 * ./gpiotest /dev/myir_gpioctr0 on
 * ./gpiotest /dev/myir_gpioctr0 off
 */
int main(int argc, char **argv)
{
    int fd;
    char status;

    /* 1. Parameter judgment */
    if (argc != 3)
    {
        printf("Usage: %s <dev> <on | off>\n", argv[0]);
        return -1;
    }

    /* 2. Open file */
    fd = open(argv[1], O_RDWR);
    if (fd == -1)
    {
        printf("can not open file %s\n", argv[1]);
        return -1;
    }

    /* 3. write file */
    if (0 == strcmp(argv[2], "on"))
    {
        status = 1;
        write(fd, &status, 1);
    }
    else
```

```
{  
    status = 0;  
    write(fd, &status, 1);  
}  
  
close(fd);  
  
return 0;  
}
```

Copy the above code to an *example-gpio.c* file and Initialize cross-compilation via SDK:

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

Use the compile command \$CC to generate the executable file gpiotest:

```
$CC gpiotest.c -o gpiotest
```

Finally, copy the executable file to the board directory (*/usr/bin*), the following command is an example on how to run directly(“on” means set high, “off” means set low):

```
root@myir:~# gpiotest /dev/myir_gpioctr0 on  
root@myir:~# gpiotest /dev/myir_gpioctr0 off
```

7. How to add an application

The porting of Linux applications is usually divided into two stages: development debugging and production deployment. In the development and debugging stage, thanks to the SDK Customized by MYIR (Please refer to Section 2.3 for details), it is easy to develop and debug a application in standalone environment. However, in the production deployment stage, thanks to the Yocto project, users only need to write the recipe file for the tested application and put the source code in the corresponding directory. Then you can use bitbake command to rebuild the image and automatically package the application into the system.

7.1. Makefile-based project

Makefile is actually a file, which defines a series of Compilation Rules to guide the compilation of source code. After defining the Compilation Rules in Makefile, users only need one make command, and the whole project will be compiled automatically, which greatly improves the efficiency of software development. In the development of Linux programs, no matter the kernel, driver, application, makefile has been widely used.

However, make is a command tool to explain the rules of makefile file. When the make command is executed, the make command will search for makefile (or makefile) in the current directory, and then execute the operations defined in makefile. It can not only simplify the command line of the compiler terminal, but also automatically judge whether the original file has been changed, so as to automatically recompile the changed source code.

The following will take an example (to realize the key control LED light on and off) to describe the preparation of makfile and the execution process of make. The makefile rules are as follows:

```
target ... : prerequisites ...
           command
```

- target:"target" can be an object file, an execution file, or a label..
- prerequisites:It is the file needed to generate target.
- command:This is the command that make needs to execute.

```

TARGET = $(notdir $(CURDIR))
objs := $(patsubst %.c, %.o, $(shell ls *.c))
$(TARGET)_test:$(objs)
    $(CC) -o $@ $^
%.o: %.c
    $(CC) -c -o $@ $<
clean:
    rm -f $(TARGET)_test *.all *.o

```

- CC: Name of C compiler
- CXX: Name of C++ compiler
- clean: It's an agreed goal

The detailed codes are as follows:

```

//File: Key_led.c
#include <linux/input.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/* ./key_led /dev/input/event0 noblock */
int main(int argc, char **argv)
{
    int fd,bg_fd;
    int err, len, i;

```

```
unsigned char flag;
unsigned int data[1];
char *bg = "/sys/class/leds/heartbeat/brightness";

struct input_event event;

if (argc < 2)
{
    printf("Usage: %s <dev> [noblock]\n", argv[0]);
    return -1;
}

if (argc == 3 && !strcmp(argv[2], "noblock"))
{
    fd = open(argv[1], O_RDWR | O_NONBLOCK);
}
else
{
    fd = open(argv[1], O_RDWR);
}
if (fd < 0)
{
    printf("open %s err\n", argv[1]);
    return -1;
}

while (1)
{
    len = read(fd, &event, sizeof(event));
    if (event.type == EV_KEY)
    {
        if (event.value == 1)//key down and up
```

```

    {

        printf("key test \n");
        bg_fd = open(bg, O_RDWR);
        if (bg_fd < 0)
        {
            printf("open %d err\n", bg_fd);
            return -1;
        }
        read(bg_fd,&flag,1);
        if(flag == '0')
            system("echo 1 > /sys/class/leds/heartbeat/bright-
htness"); //led off
        else
            system("echo 0 > /sys/class/leds/heartbeat/brightnes-
s");//led on
        }

    }

return 0;
}

```

Then execute the make command to compile and generate the executable file target on the target machine.

Set up the host terminal window toolchain environment:

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-
neon-vfpv4-ostl-linux-gnueabi
```

Execute make command to generate executable file:



PC\$ make

Finally, copy the executable file(target_bin) to the board directory(/usr/bin), the following command is an example on how to run directly:

```
root@myir:~# target_bin /dev/input/event0 noblock
```

Press S2 key on MYD-YF13X development board to control LED (D3) on and off.

Note: if you use the cross tool chain compiler to build target_Bin, and the architecture of the building host is different from that of the target machine, so you need to run the project on the target device.

7.2. Application based on QT

Qt is a cross-platform graphics application development framework that is used on different sizes of devices and platforms and offers different copyright versions for users to choose from. MYD-YF13X uses Qt version 5.14 for application development. In Qt application development, it is recommended to use QtCreator integrated development environment. Qt application can be developed under Linux PC, which can be automatically cross-compiled into the ARM architecture of development board.

1) Qtcreator installation and configuration

Get the qtcreator installation package from the QT website or the MYIR official package: http://download.qt.io/development_releases/qtcreator/4.1/4.1.0-rc1/ .

The QtCreator installation package is a binary program that can be installed by executing it directly: `./qt-creator-opensource-linux-x86_64-4.1.0-rc1.run`, for details of installation and configuration, please refer to "MYD-YF13X QT and MEasy HMI2.0 software development guide" (not released yet) or get more development guidance from qtcreator official website:

<https://www.qt.io/product/development-tools> .

2) Compiling and running of MEasy HMI2.0

MEasy HMI 2.0 is a set of QT5 based human-machine interface framework developed by Shenzhen Myir Technology Co., Ltd. The project uses QML and C ++ mixed programming, uses QML to efficiently and conveniently build the UI, and C ++ is used to implement business logic and complex algorithms.

Go to the "MYD-YF13X-2023xxx\04_Sources\" directory and you can find the MEeasy HMI2.0 project source code(`mxapp2.tar.gz`). Then it can be compiled and debugged remotely through qtcreator..

7.3. Automatic application startup at boot time

1) Application configuration in Yocto

Usually, our application also needs to realize self running after startup, which can also be realized in the recipes. Take a slightly more complex FTP service application as an example to illustrate how to use Yocto to build a production image containing specific applications. The FTP service program described in this section adopts open source proftpd, and the source codes of each version are located in <ftp://ftp.proftpd.org/distrib/source/>.

Before we start to write a recipe, we can find out whether the application, or a similar application's recipe, already exists in the current source code repository. The search method is as follows:

```
PC $ bitbake -s | grep proftpd
```

Note: before executing the bitbake command, make sure you have executed the environment variable settings script that builds the yocto project. Please refer to Chapter 3 for details.

You can also find recipes for the same or similar applications in openembedded's official website layer index:

<http://layers.openembedded.org/layerindex/branch/master/layers/> .

For the method of writing new recipes, please refer to the new recipes section of yocto project complete manual:

<https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#new-recipe-writing-a-new-recipe> .

This section focuses on how to port FTP services to the target machine. By searching the current source code repository, it is found that the recipe of proftpd already exists in the yocto project, but it is not added to the system image. The specific porting process is described in detail below.

- Find proftpd recipe of yocto project

```
PC $ ~/Yocto/build-openstlinuxweston-myd-yf13x-emmc$ bitbake -s | grep proftpd
proftpd
:1.3.7c-r0
:1.3.6-r0
```

Note: it can be seen that the proftpd recipe, version 1.3.7c-r0, already exists in yocto project.

- Compiling proftpd with the bitmake command

```
PC $ bitbake proftpd
```

- Package proftpd into the file system

Add a line in conf/ local.conf As follows:

```
IMAGE_INSTALL_append = "proftpd"
```

- Rebuild file system

```
PC $ bitbake myir-image-core
```

- View services

Check whether the service is running after burning the new image, the following command is an example on how to check the proftpd service:

```
# ps -aux | grep proftpd
nobody      584  0.0  0.3   3032  1344 ?          Ss   01:51   0:00 proftpd: (accepting connections)
root        1713  0.0  0.0   1776   336 pts/0     S+   01:59   0:00 grep proftpd
```

Here is a supplementary explanation of FTP account settings. FTP client has three types of login accounts, which are anonymous account, normal account and root account.

- Anonymous account settings

The user name is FTP, and there is no need to set a password. After logging in, the user can view the contents in the system /var/lib/ftp directory, and has no write

permission by default. Since the `/var/lib/ftp` directory does not exist by default, users need to create a directory `/var/lib/ftp` on the target machine. To avoid modifying the meta openembedded layer, this can be done by using a `bbappend` recipe. For example, Folder creation and permission changes provided in a `proftpd_1%.append` with these lines.

```
do_install_append() {
    install -m 755 -d ${D}/var/lib/${FTPUSER}
    chown ftp:ftp ${D}/var/lib/${FTPUSER}
}
```

After editing `proftpd_1%.append`, place it in the `recipes-daemons/proftpd/` directory under the meta-myir-st layer. Then rebuild the image file for testing.

- Ordinary account settings

Using the commands of `useradd` and `passwd` on the target machine, you can create an ordinary user, and after setting the user password, the client can also log in to the user's home directory with the common account. If you need to include ordinary users when packaging images, you can refer to the following steps, then rebuild the image.

Add the following information to `conf/local.conf`.

```
INHERIT += "extrausers"
EXTRA_USERS_PARAMS += "\n
    usermod -p 'FPSDUiQTQvUn2' root; \
    useradd ucas; \
    usermod -p 'FPSDUiQTQvUn2' tester; \
"
```

- Root account settings

If you need to log in to the FTP server with root account, you need to modify `"/etc/proftpd.conf"` file, adding a row of configuration "RootLogin on" to the file. At the same time, you need to set the password for the root account. After

the proftpd service is restarted, the client can log in to the target machine using the root account.

```
# systemctl restart proftpd
```

Note: in order to enable the root account to log in, the user generally needs to modify the "/etc/proftpd.conf" file configuration, which is only used for testing. For more configuration of this file, please refer to the site:<http://www.proftpd.org/docs/example-conf.html>.

2) Application service starts automatically at boot time

This section will take proftpd recipe as an example to introduce how to add the application recipe and realize the startup of the program. Proftpd recipes are located in the source code repository layers(*/meta-openembedded/meta-networking/recipes-daemons/proftpd*). The directory structure is as follows:

```
├── files
│   ├── basic.conf.patch
│   ├── build_fixup.patch
│   ├── close-RequireValidShell-check.patch
│   ├── contrib.patch
│   ├── default
│   ├── proftpd-basic.init
│   └── proftpd.service
└── proftpd_1.3.7c.bb
```

1 directory, 8 files

- proftpd_1.3.7c.bb: Recipe for building proftpd service
- proftpd.service: Auto start service at boot time
- proftpd-basic.init: Start script for proftpd

The "*proftpd_1.3.7c.bb*" recipe file specifies the source code path to obtain proftpd service program and some patch files for this version of source code:

SUMMARY = "Secure and configurable FTP server"

```

SECTION = "net"
HOMEPAGE = "http://www.proftpd.org"
LICENSE = "GPL-2.0-or-later"
LIC_FILES_CHKSUM = "file://COPYING;md5=fb0d1484d11915fa88a6a7702f1dc18
4"

SRCREV = "75aa739805a6e05eeb31189934a3d324e7862962"
BRANCH = "1.3.7"

SRC_URI = "git://github.com/proftpd/proftpd.git;branch=${BRANCH};protocol=h
https \
    file://basic.conf.patch \
    file://proftpd-basic.init \
    file://default \
    file://close-RequireValidShell-check.patch \
    file://contrib.patch \
    file://build_fixup.patch \
    file://proftpd.service \
    "

```

In addition, the configuration(`do_configure`) and installation process(`do_install`) of proftpd are also specified in the recipe:

```

FTPUSER = "ftp"
FTPGROUP = "ftp"

do_install () {
    oe_runmake DESTDIR=${D} install
    rmdir ${D}${libdir}/proftpd ${D}${datadir}/locale
    [ -d ${D}${libexecdir} ] && rmdir ${D}${libexecdir}
    sed -i '/ *User[ \t]*s/ftp/${FTPUSER}/' ${D}${sysconfdir}/proftpd.conf
    sed -i '/ *Group[ \t]*s/ftp/${FTPGROUP}/' ${D}${sysconfdir}/proftpd.conf

```



```

install -d ${D}${sysconfdir}/init.d
install -m 0755 ${WORKDIR}/proftpd-basic.init ${D}${sysconfdir}/init.d/proftpd
sed -i 's!/usr/sbin/!${sbindir}!g' ${D}${sysconfdir}/init.d/proftpd
sed -i 's!/etc/!${sysconfdir}!g' ${D}${sysconfdir}/init.d/proftpd
sed -i 's!/var/!${localstatedir}!g' ${D}${sysconfdir}/init.d/proftpd
sed -i 's!^PATH=.*/!PATH=${base_sbindir}:${base_bindir}:${sbindir}:${bindir}!' \
${D}${sysconfdir}/init.d/proftpd

install -d ${D}${sysconfdir}/default
install -m 0755 ${WORKDIR}/default ${D}${sysconfdir}/default/proftpd

# create the pub directory
mkdir -p ${D}/home/${FTPUSER}/pub/
chown -R ${FTPUSER}: ${FTPGROUP} ${D}/home/${FTPUSER}/pub
if ${@bb.utils.contains('DISTRO_FEATURES', 'pam', 'true', 'false', d)}; then
    # install proftpd pam configuration
    install -d ${D}${sysconfdir}/pam.d
    install -m 644 ${S}/contrib/dist/rpm/ftp.pam ${D}${sysconfdir}/pam.d/
proftpd
    sed -i '/ftpusers/d' ${D}${sysconfdir}/pam.d/proftpd
    # specify the user Authentication config
    sed -i '/^MaxInstances/a\AuthPAM' ${D}${sysconfdir}/pam.d/proftpd
on\nAuthP
AMConfig           proftpd' \
${D}${sysconfdir}/proftpd.conf
fi

install -d ${D}/${systemd_unitdir}/system
install -m 644 ${WORKDIR}/proftpd.service ${D}/${systemd_unitdir}/system
sed -e 's,@BASE_SBINDIR@,$base_sbindir,g' \
-e 's,@SYSCONFDIR@,$sysconfdir,g' \
-e 's,@SBINDIR@,$sbindir,g' \

```

```

-i ${D}${systemd_unitdir}/system/*.service

sed -e 's|--sysroot=${STAGING_DIR_HOST}||g' \
-e 's|${STAGING_DIR_NATIVE}||g' \
-e 's|-fdebug-prefix-map=[^ ]*||g' \
-e 's|-fmacro-prefix-map=[^ ]*||g' \
-i ${D}/${bindir}/prxs

# ftpmail perl script, which reads the proftpd log file and sends
# automatic email notifications once an upload finishes,
# depends on an old perl Mail::Sendmail
# The Mail::Sendmail has not been maintained for almost 10 years
# Other distribution not ship with ftpmail, so do the same to
# avoid confusion about having it fails to run
rm -rf ${D}${bindir}/ftpmail
rm -rf ${D}${mandir}/man1/ftpmail.1
}

```

For more information about how to install tasks, refer to:

<https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#ref-tasks>.

proftpd_1.3.7c.bb recipe inheritance systemd.class class(Please refer to:
layers/openembedded-core/meta/classes/systemd.bbclass). If you want to run the application service in the boot phase, you need to use the default enable variable(SYSTEMD_AUTO_ENABLE). For example, the user can set the variable(SYSTEMD_AUTO_ENABLE) to start the application service. The example is as follows:

```
SYSTEMD_AUTO_ENABLE_${PN} = "enable"
```

At present, the target machine uses systemd tool as the initialization management subsystem. Systemd tool is a collection of basic components of Linux system, which provides a system and service manager. The running process number is PID 1 and is responsible for starting other programs. For an example of how to

configure systemd under yocto project, please refer to:<https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#selecting-an-initialization-manager> .

The contents of the proftpd service file are as follows:

```
[Unit]
Description=proftpd Daemon
After=network.target

[Service]
Type=forking
ExecStart=@SBINDIR@/proftpd -c @SYSCONFDIR@/proftpd.conf
StandardError=syslog

[Install]
WantedBy=default.target
```

- After:Indicates that the service will run after the network service is started.
- Type:systemd considers the service started up once the process forks and the parent has exited. For classic daemons use this type unless you know that it is not necessary. You should specify PIDFile= as well so systemd can keep track of the main process.
- ExecStart:Indicates the program to be started and its parameters.

For more information about systemd, please check this website:<https://wiki.archlinux.org/index.php/systemd> .

If you are adding your own application, you can also refer to the above example to create a recipe,enable a unit to start automatically at boot, and package it into the system image.It is generally recommended to place your own recipes in the */layers/meta-myir-st/recipes-app/* directory.

Reference

- **Linux kernel open source community**
<https://www.kernel.org/>
- **STM32MPU Development Zone**
https://wiki.st.com/stm32mpu/wiki/Development_zone
- **Yocto Development Guide**
<https://www.yoctoproject.org/>
- **Yocto Project BSP Development Guide**
<https://docs.yoctoproject.org/4.0.9/bsp-guide/index.html>
- **Yocto Project Linux Kernel Development Guide**
<https://docs.yoctoproject.org/4.0.9/kernel-dev/index.html>

Appendix A

Warranty & Technical Support Services

MYIR Electronics Limited is a global provider of ARM hardware and software tools, design solutions for embedded applications. We support our customers in a wide range of services to accelerate your time to market.

MYIR is an ARM Connected Community Member and work closely with ARM and many semiconductor vendors. We sell products ranging from board level products such as development boards, single board computers and CPU modules to help with your evaluation, prototype, and system integration or creating your own applications. Our products are used widely in industrial control, medical devices, consumer electronic, telecommunication systems, Human Machine Interface (HMI) and more other embedded applications. MYIR has an experienced team and provides custom design services based on ARM processors to help customers make your idea a reality.

The contents below introduce to customers the warranty and technical support services provided by MYIR as well as the matters needing attention in using MYIR' s products.

Service Guarantee

MYIR regards the product quality as the life of an enterprise. We strictly check and control the core board design, the procurement of components, production control, product testing, packaging, shipping and other aspects and strive to provide products with best quality to customers. We believe that only quality products and excellent services can ensure the long-term cooperation and mutual benefit.

Price

MYIR insists on providing customers with the most valuable products. We do not pursue excess profits which we think only for short-time cooperation. Instead, we hope to establish

long-term cooperation and win-win business with customers. So we will offer reasonable prices in the hope of making the business greater with the customers together hand in hand.

Delivery Time

MYIR will always keep a certain stock for its regular products. If your order quantity is less than the amount of inventory, the delivery time would be within three days; if your order quantity is greater than the number of inventory, the delivery time would be always four to six weeks. If for any urgent delivery, we can negotiate with customer and try to supply the goods in advance.

Technical Support

MYIR has a professional technical support team. Customer can contact us by email (support@myirtech.com), we will try to reply you within 48 hours. For mass production and customized products, we will specify person to follow the case and ensure the smooth production.

After-sale Service

MYIR offers one year free technical support and after-sales maintenance service from the purchase date. The service covers:

Technical support service

MYIR offers technical support for the hardware and software materials which have provided to customers:

- To help customers compile and run the source code we offer;
- To help customers solve problems occurred during operations if users follow the user manual documents;
- To judge whether the failure exists;
- To provide free software upgrading service.

However, the following situations are not included in the scope of our free technical support service:

- Hardware or software problems occurred during customers' own development;
- Problems occurred when customers compile or run the OS which is tailored by themselves;
- Problems occurred during customers' own applications development;
- Problems occurred during the modification of MYIR's software source code.

After-sales maintenance service

The products except LCD, which are not used properly, will take the twelve months free maintenance service since the purchase date. But following situations are not included in the scope of our free maintenance service:

- The warranty period is expired;
- The customer cannot provide proof-of-purchase or the product has no serial number;
- The customer has not followed the instruction of the manual which has caused the damage the product;
- Due to the natural disasters (unexpected matters), or natural attrition of the components, or unexpected matters leads the defects of appearance/function;
- Due to the power supply, bump, leaking of the roof, pets, moist, impurities into the boards, all those reasons which have caused the damage of the products or defects of appearance;
- Due to unauthorized weld or dismantle parts or repair the products which has caused the damage of the products or defects of appearance;
- Due to unauthorized installation of the software, system or incorrect configuration or computer virus which has caused the damage of products.

Warm tips

1. MYIR does not supply maintenance service to LCD. We suggest the customer first check the LCD when receiving the goods. In case the LCD cannot run or no display, customer should contact MYIR within 7 business days from the moment get the goods.
2. Please do not use finger nails or hard sharp object to touch the surface of the LCD.
3. MYIR suggests user purchasing a piece of special wiper to wipe the LCD after long time use, please avoid clean the surface with fingers or hands to leave fingerprint.
4. Do not clean the surface of the screen with chemicals.
5. Please read through the product user manual before you using MYIR' s products.
6. For any maintenance service, customers should communicate with MYIR to confirm the issue first. MYIR' s support team will judge the failure to see if the goods need to be returned for repair service, we will issue you RMA number for return maintenance service after confirmation.

Maintenance period and charges

- › MYIR will test the products within three days after receipt of the returned goods and inform customer the testing result. Then we will arrange shipment within one week for the repaired goods to the customer. For any special failure, we will negotiate with customers to confirm the maintenance period.
- › For products within warranty period and caused by quality problem, MYIR offers free maintenance service; for products within warranty period but out of free maintenance service scope, MYIR provides maintenance service but shall charge some basic material cost; for products out of warranty period, MYIR provides maintenance service but shall charge some basic material cost and handling fee.

Shipping cost

During the warranty period, the shipping cost which delivered to MYIR should be responsible by user; MYIR will pay for the return shipping cost to users when the product is repaired. If the warranty period is expired, all the shipping cost will be responsible by users.

Products Life Cycle

MYIR will always select mainstream chips for our design, thus to ensure at least ten years continuous supply; if meeting some main chip stopping production, we will inform customers in time and assist customers with products updating and upgrading.

Value-added Services

1. MYIR provides services of driver development base on MYIR' s products, like serial port, USB, Ethernet, LCD, etc.
2. MYIR provides the services of OS porting, BSP drivers' development, API software development, etc.
3. MYIR provides other products supporting services like power adapter, LCD panel, etc.
4. ODM/OEM services.

MYIR Electronics Limited

Room 04, 6th Floor, Building No.2, Fada Road,
Yunli Intelligent Park, Bantian, Longgang District.

Support Email: support@myirtech.com

Sales Email: sales@myirtech.com

Phone: +86-755-22984836

Fax: +86-755-25532724

Website: www.myirtech.com