

# MYD-Y6ULX Linux System Development Guide



File Status: [ ] Draft [ ✓ ] Release	<b>FILE ID:</b>	MYIR-MYD-Y6ULX-SW-DG-EN-L5.10.9
	<b>VERSION:</b>	V3.0.0[DOC]
	<b>AUTHOR:</b>	Mark
	<b>CREATED:</b>	2020-08-04
	<b>UPDATED:</b>	2022-08-31

## Revision History

VERSION	AUTHOR	PARTICIPANT	DATE	DESCRIPTION
V2.0.0	Alex		20210118	Initial Version: u-boot2019.03, Linux Kernel 5.4.3, Yocto 3.0
V2.0.1	Alex		20210222	Update pin error in GPIO test,etc.
V3.0.0	Mark		20220831	Uboot is upgraded to 2020.04 and Kernel is upgraded to 5.10.9 based on the previous version Yocto:5.10-gatesgarth

# CONTENT

MYD-Y6ULX Linux System Development Guide .....	- 1 -
Revision History .....	- 2 -
CONTENT .....	- 3 -
1. Overview .....	- 5 -
1.1. Software Resources .....	- 6 -
1.2. Document Resources .....	- 6 -
2. Development Environment .....	- 7 -
2.1. Hardware environment .....	- 7 -
2.2. Software environment .....	- 8 -
2.2.1. Get information .....	- 8 -
2.2.2. Setting up a compilation environment .....	- 8 -
2.2.3. Install the SDK Customized by MYIR .....	- 9 -
3. Build the File System with Yocto .....	- 12 -
3.1. Introduction .....	- 12 -
3.2. Get the Source Code .....	- 13 -
3.2.1. Get Compressed Source Code from CD Image .....	- 13 -
3.2.2. Get Source Code from GitHub .....	- 13 -
3.3. Build Development Board Image .....	- 15 -
3.4. Build SDK (optional) .....	- 18 -
4. How to Burn System Image .....	- 19 -
4.1. How to Flash with UUU .....	- 19 -
4.2. How to Flash with SDcard .....	- 21 -
5. How to Modify Board Level Support Package .....	- 25 -
5.1. Introduction to meta-myir Layer .....	- 25 -
5.2. Introduction to Board Level Support Package .....	- 28 -
5.3. U-Boot Compilation .....	- 29 -

5.4. Kernel Compilation .....	- 32 -
6. How to Fit Your Hardware Platform .....	- 35 -
6.1. How to Create Your Device Tree .....	- 35 -
6.1.1. Board Level Device Tree .....	- 35 -
6.1.2. Add your board level device tree .....	- 36 -
6.2. How to configure function pins according to your hardware .....	- 38 -
6.2.1. GPIO pin configuration .....	- 38 -
6.3. How to use your own configured pins .....	- 42 -
6.3.1. How to use GPIO in uboot .....	- 42 -
6.3.2. How to use GPIO in Kernel driver .....	- 44 -
6.3.3. How to control a GPIO in Userspace .....	- 51 -
7. How to add an application .....	- 57 -
7.1. Makefile-based project .....	- 57 -
7.2. Application based on QT .....	- 61 -
7.3. Automatic application startup at boot time .....	- 62 -
7.4. QT Application .....	- 69 -
8. Reference .....	- 77 -
● Linux kernel open source community .....	- 77 -
● Yocto Development Guide .....	- 77 -
● Yocto Project BSP Development Guide .....	- 77 -
● Yocto Project Linux Kernel Development Guide .....	- 77 -
Appendix A .....	- 78 -
Warranty & Technical Support Services .....	- 78 -

# 1. Overview

There are many open source system build frameworks on the Linux system platform, these frameworks make it easy for developers to build and customize embedded systems, at present common similar software has Buildroot, Yocto, OpenEmbedded and so on. The Yocto project uses a more powerful and customized approach to build Linux systems that suitable for embedded products. Yocto is not only a file system manufacturing tool, but also provides a complete set of Linux-based development and maintenance workflow, so that the embedded developers of the Underlying Software and the High-Level Application can develop under a unified framework, which solves the fragmented and unmanaged development mode in the traditional development mode.

This document mainly introduces the complete process of customizing a complete embedded Linux system based on Yocto project, including the configuration of development environment, how to get the source code, how to port bootloader and kernel, and how to customize rootfs suitable for their own application requirements. First of all, we will introduce how to build a system image for MYD-Y6ULX development board based on the source code provided by us, and how to burn the prebuilt image to the development board. Then, we focus on the methods and key points of porting the system to the user's hardware platform. In addition, if you are developing a project based on MYC-Y6ULX CPU module, we will also take some actual BSP porting cases and rootfs customization cases as examples to guide users to quickly customize the system image suitable for their own base-board hardware.

This document does not include the introduction of Yocto project and the basic knowledge of Linux system, and the user guide is suitable for embedded Linux development engineers with some development experience. For some specific functions that users may use in the process of secondary development, we also provide detailed application notes for reference, please refer to Table 2-4 of "MYD-Y6ULX SDK Release Notes" for the detailed list of documents.

## 1.1. Software Resources

MYD-Y6ULX series development board runs an operating system based on the Linux 5.10.9 kernel, which also provided a wealth of system resources and other software resources. The resource packages provided with the development board are as follows:

- A software development kit (SDK) for cross-development on an host PC.
- Distribution in source code: U-Boot, Linux Kernel and drivers of each module .
- Various debugging tools for Windows desktop and Linux desktop environments.
- Various peripherals application development samples, etc.

For specific software information, please refer to Table 2-4 of “MYD-Y6ULX SDK Release Notes” for the detailed list of documents.

## 1.2. Document Resources

According to the different stages of using the development board, the SDK contains different types of documents and manuals, such as release notes, introduction guide, evaluation guide, development guide, application notes, frequently asked questions and answers, etc. For detailed document list, please refer to table 2-4 of “MYD-Y6ULX\_SDK Release Notes” .

## 2. Development Environment

This chapter mainly introduces some software and hardware environment required in the development process, including the necessary development host environment, necessary software tools, code and resource acquisition, etc. the specific preparatory work will be described in detail below.

### 2.1. Hardware environment

#### ● Necessary Accessories

- 12V power adapter
- No less than 4GB SD card
- USB to TTL debugging cable (used for debugging serial port), pay attention to use 3.3V level, etc. 2).

#### ● Startup Settings

This section mainly introduces the startup method of the development board so that users can better choose the startup method.

Table 2-1. Boot Mode Selector Switch

BOOT MODE	SWITCH(B1/B2/B3/B4)
Boot form eMMC	OFF/OFF/ON/OFF
Boot form NAND Flash	OFF/ON/ON/OFF
Boot form SD Card(eMMC board)	ON/ON/ON/OFF
Boot form SD Card(NAND board)	ON/OFF/ON/OFF
USB Download	X/X/OFF/ON

#### ● Serial port configuration

Connect the USB to TTL cable to the debugging serial port JP1 correctly, connect the USB end to the PC, and use the debugging software to set the baud rate of the PC serial port to 115200, the data bit to 8, the stop bit to 1, and no parity.

Table 2-2. debug port

Baud rate	Data Bit	Stop	Check	other
115200	8	1	No	No

Use a 12V power adapter to connect to the J22 interface of the development board, and it can start normally after power on. The startup information will be printed out under serial debugging on the PC side.

**Note:** The default system user name is root and the password is blank.

## 2.2. Software environment

This section describes how to deploy the i.MX6UL development environment. By reading this section, you will learn about the installation and use of hardware and software tools. And you can quickly deploy the relevant development environment and prepare for subsequent development and debugging.

### 2.2.1. Get information

Download the development board materials before setting up the environment. For detailed information about the development materials, please refer to "MYD-Y6ULX\_SDK Release Notes".

The download address of the development board data is as follows (the data will be updated from time to time, please download the latest version):

<http://d.myirtech.com/MYD-Y6ULX/>

### 2.2.2. Setting up a compilation environment

- **Host Hardware**

To get the Yocto Project expected behavior in a Linux Host Machine, the packages and utilities described below must be installed. An important consideration is the hard disk space required in the host machine. It is recommended that at least 160 GB is provided, which is enough to compile all backends together. In addition, the processor with more than dual core CPU, 8GB memory or higher configuration will better meet the operation requirements. It can be the host with Linux system installed, virtual machine running Linux system, etc.



- **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://docs.yoctoproject.org/> 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 20.04 64bit desktop system, the subsequent development is also based on this system.

- **Prerequisite Package Installation**

```
myir$ sudo apt-get update
myir$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib
build-essential chrpath socat cpio python python3 python3-pip python3-pexpect
xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev
pylint3 xterm
```

- **Create a working directory**

Create a working directory to facilitate the creation of an unified environment variable path. Copy the product CD-ROM source code to the working directory, while setting the DEV\_ROOT variable to enable the follow-up step path accessed.

```
myir$ mkdir -p ~/MYD-Y6ULX-devel
myir$ export DEV_ROOT=~/MYD-Y6ULX-devel
myir$ cp -r <DVDROM>/02_Images $DEV_ROOT
myir$ cp -r <DVDROM>/03_Tools $DEV_ROOT
myir$ cp -r <DVDROM>/04_Sources $DEV_ROOT
```

### **2.2.3.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/Tools\_chain/* directory. 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:

- **View script file**

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

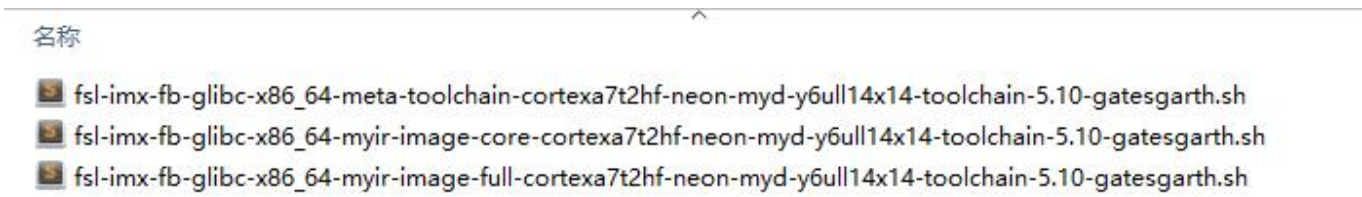


Figure 2-1. toolchain

Table 2-3. toolchain

Toolschain file name	Description
fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.sh	meta-toolchain
fsl-imx-fb-glibc-x86_64-myr-image-full-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.sh	myr-image-full
fsl-imx-fb-glibc-x86_64-myr-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.sh	myr-image-core

- **Run the SDK installation script**

The SDK is installed in the */opt/* directory by default. Users can also choose the appropriate directory according to the prompts:

```
myir@myir-O-E-M:/home/hjx/sdk$ sudo ./fsl-imx-fb-glibc-x86_64-myir-image-full-
cortexa7t2hf-neon-myid-y6ull14x14-toolchain-5.10-gatesgarth.sh
NXP i.MX Release Distro SDK installer version 5.10-gatesgarth
=====
Enter target directory for SDK (default: /opt/fsl-imx-fb/5.10-gatesgarth): /opt/test5.10/
You are about to install the SDK to "/opt/test5.10/". Proceed [Y/n]? y
Extracting
SDK.....
.....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/test5.10/environment-setup-cortexa7t2hf-neon-poky-linux-gnueab
```

### ● Test SDK

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

```
myir@myir-O-E-M:/home/hjx/sdk$ source /opt/test5.10/environment-setup-
cortexa7t2hf-neon-poky-linux-gnueabi
myir@myir-O-E-M:/home/hjx/sdk$ $CC -v
Using built-in specs.

COLLECT_GCC=arm-poky-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/opt/test5.10/sysroots/x86_64-pokysdk-
linux/usr/libexec/arm-poky-linux-gnueabi/gcc/arm-poky-linux-gnueabi/10.2.0/lto-
wrapper
gcc version 10.2.0 (GCC)
```

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](http://www.yoctoproject.org).

MYIR's CD image *04\_Sources* directory contains Yocto metafile data for MYD-Y6ULX development board, which helps developers build different types of Linux system images that can run on MYD-Y6ULX development board, such as the myir-image-full system image with qt5.15 graphics library, the myir-image-core system image without GUI interface, the st official Weston demonstration system image. Next, we will take the implementation of myir image full image as an example to introduce the specific development process, so as to lay a foundation for subsequent customization of system image suitable for ourselves.

**Note:** the SDK toolchain environment variables in Section 2.2.3 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-Y6ULX 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*/MYiR-i.MX6UL-Yocto.tar.gz.

```
myir$ cd $DEV_ROOT/04_Sources
myir$ tar -xzf MYiR-i.MX6UL-Yocto.tar.gz
myir$ ls
myir-setup-release.sh  README  README-IMXBSP  setup-environment  sources
```

### 3.2.2. Get Source Code from GitHub

At present, the BSP source code and yocto source code of MYD-Y6ULX development board are managed by GitHub and will be updated for a long time. Please refer to Section 2.2 of “MYD-Y6ULX\_SDK Release Notes” . Users can use repo to get and synchronize the code on GitHub. The specific operation methods are as follows.

Put the 03\_tools/Repo/repo file in the server /usr/bin directory and add executable permissions. Then use repo to pull the Yocto source code:

```
myir$: export REPO_URL='https://mirrors.tuna.tsinghua.edu.cn/git/git-repo/'
```

```
myir$: repo init -u https://github.com/MYiR-Dev/myir-imx-manifest.git --no-clone-bundle --depth=1 -m myir-i.mx6ul-5.10.9-1.0.0.xml -b i.MX6UL-5.10-gatesgarth
myir$:repo sync
```

Executing repo sync will download the code. It takes a certain amount of time.

Please be patient. The code download in the following figure is being downloaded:

```
ndran@myir-server1: /imx6ul-yocto-5.10/11111 repo sync
remote: Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote: Enumerating objects: 46, done.
remote: Counting objects: 100% (46/46), done.
remote: Enumerating objects: 289, done.
remote: Counting objects: 100% (289/289), done.
remote: Enumerating objects: 775, done.
remote: Compressing objects: 100% (226/226), done.
remote: Enumerating objects: 244, done.
remote: Compressing objects: 100% (40/40), done.
remote: Counting objects: 100% (775/775), done.
remote: Enumerating objects: 281, done.
remote: Counting objects: 100% (281/281), done.
remote: Compressing objects: 100% (587/587), done.
remote: Compressing objects: 100% (265/265), done.
remote: Counting objects: 100% (244/244), done.
remote: Enumerating objects: 7291, done.
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (5/5), done.
remote: Counting objects: 100% (7291/7291), done.
remote: Compressing objects: 100% (5859/5859), done.
remote: Enumerating objects: 698, done.
remote: Compressing objects: 100% (150/150), done.
remote: Total 7 (delta 0), reused 3 (delta 0), pack-reused 0
remote: Counting objects: 100% (698/698), done.
remote: Compressing objects: 100% (586/586), done.
remote: Total 46 (delta 5), reused 19 (delta 2), pack-reused 0
Fetching projects: 18% (2/11) meta-freescale-distromote: Enumerating objects: 20, done.
remote: Counting objects: 100% (20/20), done.
remote: Compressing objects: 100% (17/17), done.
remote: Enumerating objects: 75, done.
remote: Counting objects: 100% (75/75), done.
remote: Compressing objects: 100% (66/66), done.
remote: Total 20 (delta 0), reused 8 (delta 0), pack-reused 0
Fetching projects: 27% (3/11) meta-timesysremote: Total 75 (delta 15), reused 32 (delta 4), pack-reused 0
Fetching projects: 36% (4/11) meta-rustpoky:
remote: Total 289 (delta 49), reused 189 (delta 26), pack-reused 0
Fetching projects: 45% (5/11) meta-freescale-3rdpartyremote: Total 775 (delta 103), reused 486 (delta 64), pack-reused 0
Fetching projects: 54% (6/11) meta-freescale-remote: Total 244 (delta 86), reused 170 (delta 86), pack-reused 0
Fetching projects: 63% (7/11) meta-browserremote: Total 281 (delta 26), reused 129 (delta 4), pack-reused 0
Fetching projects: 72% (8/11) meta-qt5poky:
remote: Enumerating objects: 488191, done.
remote: Counting objects: 100% (488191/488191), done.
remote: Compressing objects: 100% (115647/115647), done.
remote: Total 7291 (delta 892), reused 5975 (delta 854), pack-reused 0
Fetching projects: 81% (9/11) meta-openembeddedremote: Total 698 (delta 32), reused 680 (delta 31), pack-reused 0
Fetching projects: 90% (10/11) meta-myir-imx
```

Figure 3-1. repo sync

### 3.3. Build Development Board Image

This section provides the detailed information along with the process for building an image. Before using Yocto project to build the system, we need to set the corresponding environment variables. MYIR provides a script, `envsetup.sh`, that simplifies the setup for MYIR machines. To use the script, the name of the specific machine to be built for needs to be specified as well as the desired graphical backend. The script sets up a directory and the configuration files for the specified machine and backend. The Yocto Project build uses the `bitbake` command. For example, `bitbake <component>` builds the named component. Each component build has multiple tasks, such as fetching, configuration, compilation, packaging, and deploying to the target rootfs. The bitbake image build gathers all the components required by the image and build in order of the dependency per task. The first build is the toolchain along with the tools required for the components to build. After the build is complete, this directory will contain all the output files.

- **Prepare download file**

In order to reduce the Yocto build time, please unzip downloads to this directory to reduce the time to download the software package.

```
myir$ cd $DEV_ROOT/Yocto
myir$ tar -xzf downloads.tar.gz -C ./
```

- **Execute script to set environment variables**

Use the script provided by NXP to create a build directory, and Yocto will build all under it. There are two options for setting the machine variable, "myd-y6ull14x14" and "myd-y6ul14x14".

```
myir$: $ DISTRO=fsl-imx-fb MACHINE=myd-y6ull14x14 source myir-setup-release.sh -b build_imx6ull
myir@myir-server1:~/imx6ul/yocto$ tree -L 1
.
├── build_imx6ull
├── downloads
├── myir-setup-release.sh -> sources/meta-myir/tools/myir-setup-release.sh
└── README -> sources/base/README
```



```
└─ README-IMXBSP -> sources/meta-imx/README
└─ setup-environment -> sources/base/setup-environment
└─ sources
```

3 directories, 4 files

After the script runs, the working directory is the one just created by the script, specified with the DISTRO and MACHINE option, eg: build-openstlinuxeglfs-myr, and automatically jump to this directory. In this directory, bitbake <component> builds the named component.

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

```
myir$: bitbake myir-image-full
```

- **Building myir-image-core image**

```
myir$: bitbake myir-image-core
```

```
Build Configuration:
BB_VERSION           = "1.44.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "arm-poky-linux-gnueabi"
MACHINE              = "myd-y6ull14x14"
DISTRO               = "fsl-imx-fb"
DISTRO_VERSION        = "5.4-zeus"
TUNE_FEATURES        = "arm vfp cortexa7 neon thumb callconvention-hard"
TARGET_FPU           = "hard"
meta
meta-poky             = "HEAD:a8f6e31bec5a551fab1fec8d67489af80878f71"
meta-oe
meta-multimedia
meta-python           = "HEAD:bb65c27a772723dfe2c15b5e1b27bcc1aed884c"
meta-freescale        = "HEAD:94f4f086c6014cbcf10bda3540d19558c8bf0b0"
meta-freescale-3rdparty = "HEAD:aea3771baa77e74762358ceb673d407e36637e5f"
meta-freescale-distro = "HEAD:ca27d12e4964d1336e662bcc60184bbff526c857"
meta-bsp
meta-sdk
meta-ml               = "i.MX6UL-5.4-zeus:d162f66d830456f7c150cdb8c28cd390b393f6fa"
meta-browser          = "HEAD:5f365ef0f842ba4651afe88787cf9c63bc8b6cb3"
meta-rust             = "HEAD:d8d77be1292064a02adcb5e72e293604b704f69b"
meta-gnome
meta-networking
meta-fileystems       = "HEAD:bb65c27a772723dfe2c15b5e1b27bcc1aed884c"
meta-qt5              = "HEAD:a582fd4c810529e9af0c81700407b1955d1391d2"

Initialising tasks: 100% |#####| Time: 0:00:03
Sstate summary: Wanted 16 Found 14 Missed 2 Current 1588 (87% match, 99% complete)
NOTE: Executing Tasks
NOTE: Setscene tasks completed
Currently 1 running tasks (4549 of 4559) 99% |#####|
0: myir-image-full-1.0-r0 do rootfs - 2s (pid 6051)
```

Figure3-2. build info



```
Build Configuration:
BB_VERSION      = "1.44.0"
BUILD_SYS       = "x86_64-linux"
NATIVELSBSTRING = "universal"
TARGET_SYS      = "arm-poky-linux-gnueabi"
MACHINE         = "myd-y6ulx14x14"
DISTRO          = "fsl-imx-fb"
DISTRO_VERSION  = "5.4-zeus"
TUNE_FEATURES   = "arm vfp cortexa7 neon thumb callconvention-hard"
TARGET_FPU      = "hard"
meta
meta-poky       = "HEAD:a8f6e31bebc5a551fab1fec8d67489af80878f71"
meta-oe
meta-multimedia
meta-python     = "HEAD:bb65c27a772723dfe2c15b5e1b27bcc1a1ed884c"
meta-freescale = "HEAD:94f4f086c6014cbcfd10bda3540d19558c8bf0b0"
meta-freescale-3rdparty = "HEAD:aea3771baa7e74762358ceb673d407e36637e5f"
meta-freescale-distro = "HEAD:ca27d12e4964d1336e662bcc60184bbff526c857"
meta-bsp
meta-sdk
meta-ml         = "i.MX6UL-5.4-zeus:d162f66d830456f7c150c8b8c28cd390b393f6fa"
meta-browser    = "HEAD:5f365ef0f842ba4651efe8878cf9c63bc8b6cb3"
meta-rust       = "HEAD:d8d77be1292064a02adcb5e72e293604b704f69b"
meta-gnome
meta-networking
meta-fileystems = "HEAD:bb65c27a772723dfe2c15b5e1b27bcc1a1ed884c"
meta-qt5        = "HEAD:a582fd4c010529e9af0c81700407b1955d1391d2"

Initialising tasks: 100% |#####| Time: 0:00:03
Sstate summary: Wanted 16 Found 14 Missed 2 Current 1588 (87% match, 99% complete)
NOTE: Executing Tasks
NOTE: Setscene tasks completed
NOTE: Tasks Summary: Attempted 4559 tasks of which 4546 didn't need to be rerun and all succeeded.
hufan@myir-server1:~/imx6ul/yocto-5.4/build_imx$
```

Figure3-3. build success

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
myir-image-core	bitbake myir-image-full
myir-image-full	bitbake myir-image-core

Table 3-2. Common Commands

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

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

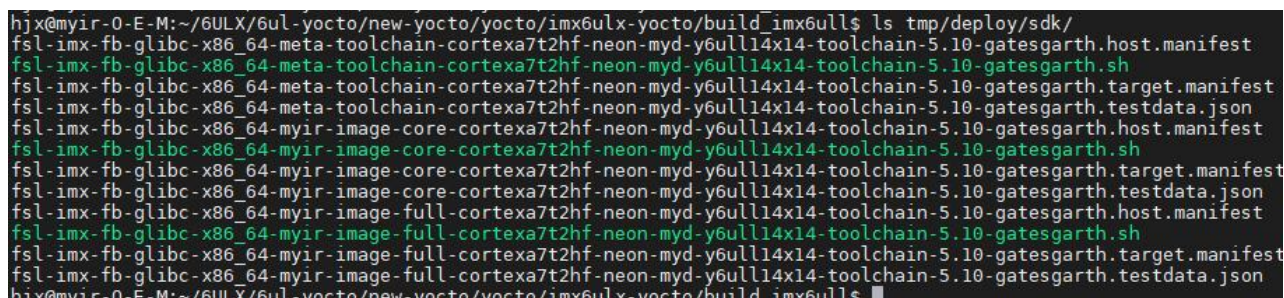
### 3.4. Build SDK (optional)

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:

```
myir$ bitbake -c populate_sdk meta-toolchain
```

After building, the SDK installation package will be generated in the path of "build\_imx6ull/tmp/deploy/sdk/".



```

hjjx@myir-0-E-M:~/6ULX/6ul-yocto/new-yocto/yocto/imx6ulx-yocto/build_imx6ull$ ls tmp/deploy/sdk/
fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.host.manifest
fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.sh
fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.target.manifest
fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.testdata.json
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.host.manifest
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.sh
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.target.manifest
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.testdata.json
fsl-imx-fb-glibc-x86_64-myir-image-full-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.host.manifest
fsl-imx-fb-glibc-x86_64-myir-image-full-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.sh
fsl-imx-fb-glibc-x86_64-myir-image-full-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.target.manifest
fsl-imx-fb-glibc-x86_64-myir-image-full-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.10-gatesgarth.testdata.json
hjjx@myir-0-E-M:~/6ULX/6ul-yocto/new-yocto/yocto/imx6ulx-yocto/build_imx6ull$
  
```

Figure3-4. build SDK

Other commands to build SDK are as follows.

Table 3-1. build sdk list

System Name	Command
myir-image-core	bitbake -c populate_sdk myir-image-core
myir-image-full	bitbake -c populate_sdk myir-image-full
meta-toolchain	bitbake -c populate_sdk meta-toolchain

## 4. How to Burn System Image

The i.MX6UL series products have various startup methods, so different update system tools and methods are needed. Users can choose different ways to update according to their needs. Since the startup mode needs to be adjusted during programming, the user selects and configures the DIP switch according to the following table.

### 4.1. How to Flash with UUU

**Note:** UUU tool is not compatible with win7, please use win10 or linux system.

#### 1) Tools Requirements

- A development board
- Type-A to Micro-B
- Adapter of 12V/2A

#### 2) Flashing

Let's take the MYD-Y6ULY2-V2-4E512D development board as an example to explain how to flash the myir-image-full system image. The other configuration methods are the same, just replace different scripts. The update steps are as follows:

- Switch the third position of the switch to start the DIP switch (SW1) is OFF, and the fourth position is ON
- Use a USB adapter cable (Type-A to Micro-B) to connect the PC USB port and the development board Micro USB OTG port (J26)
- Use the DC 12V power adapter to connect to the power socket of the development board (J22)
- Open the cmd window with administrator privileges, enter the MYD-i.MX6ULX\_UUU\_v1.1 directory, enter: `uuu.exe myd-y6ulx-y2-4e512d-qt.auto` to start programming the system, as shown below:

```
C:\Users\38099\Desktop\6UL\MYD-i.MX6ULX_UUU_v1.1\MYD-i.MX6ULX_UUU_v1.1>uuu.exe myd-y6ulx-y2-4e512d-qt.auto
uuu (Universal Update Utility) for nxp imx chips -- libuuu_1.2.39-0-gdccc404f
```

Figure 4-1. USB Flash

```
C:\Users\38099\Desktop\6UL\MYD-i.MX6ULX_UUU_v1.1\MYD-i.MX6ULX_UUU_v1.1>uuu.exe myd-y6ulx-y2-4e512d-qt.auto
uuu (Universal Update Utility) for nxp imx chips -- libuuu_1.2.39-0-gdccc404f

Success 1    Failure 0
1:331  8/ 8  [Done] FB: done
```

Figure 4-2. USB Flash Success

When the programming is completed, the green Done is displayed, and the Success is 1, as shown in the figure above.

After the programming is completed, power off, set the DIP switch to NAND or eMMC startup mode, and then power on to start from the flash of the board.

MYD-Y6ULX supports two Flash storage methods, NAND and eMMC. When using UUU to program, select different scripts to program.

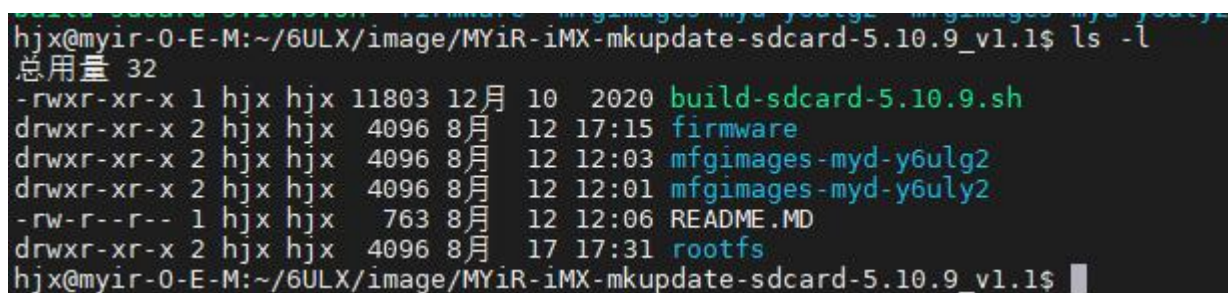
Table 4-1. MYD-Y6ULX Flash Script List

Script name	Description
myd-y6ulx-y2-4e512d-qt.auto	Flash the full file system to the board configured with MYD-Y6ULX-Y2-4D512D
myd-y6ulx-y2-4e512d-core-base.auto	Flash the core file system to the board configured with MYD-Y6ULX-Y2-4D512D
myd-y6ulx-y2-256n256d-qt.auto	Flash the core file system to the board configured with MYD-Y6ULX-Y2-256NADN 256DDR
myd-y6ulx-y2-256n256d-core-base.auto	Flash the core file system to the board configured with MYD-Y6ULX-Y2-256NADN 256DDR
myd-y6ulx-g2-4e512d-qt.auto	Flash the full file system to the board configured with MYD-Y6ULX-G2-4D512D
myd-y6ulx-g2-4e512d-core-base.auto	Flash the core file system to the board configured with MYD-Y6ULX-G2-4D512D
myd-y6ulx-g2-256n256d-qt.auto	Flash the full file system to the board configured with MYD-Y6ULX-G2-256NADN 256DDR
myd-y6ulx-g2-256n256d-core-base.auto	Flash the core file system to the board configured with MYD-Y6ULX-G2-256NADN 256DDR

## 4.2. How to Flash with SDcard

In order to meet the needs of production programming, MYiR provides a method suitable for mass production. Please follow the steps below to complete the specific production process.

MYD-Y6ULX development board provides a tool for making SD card update system image, the structure is as follows:

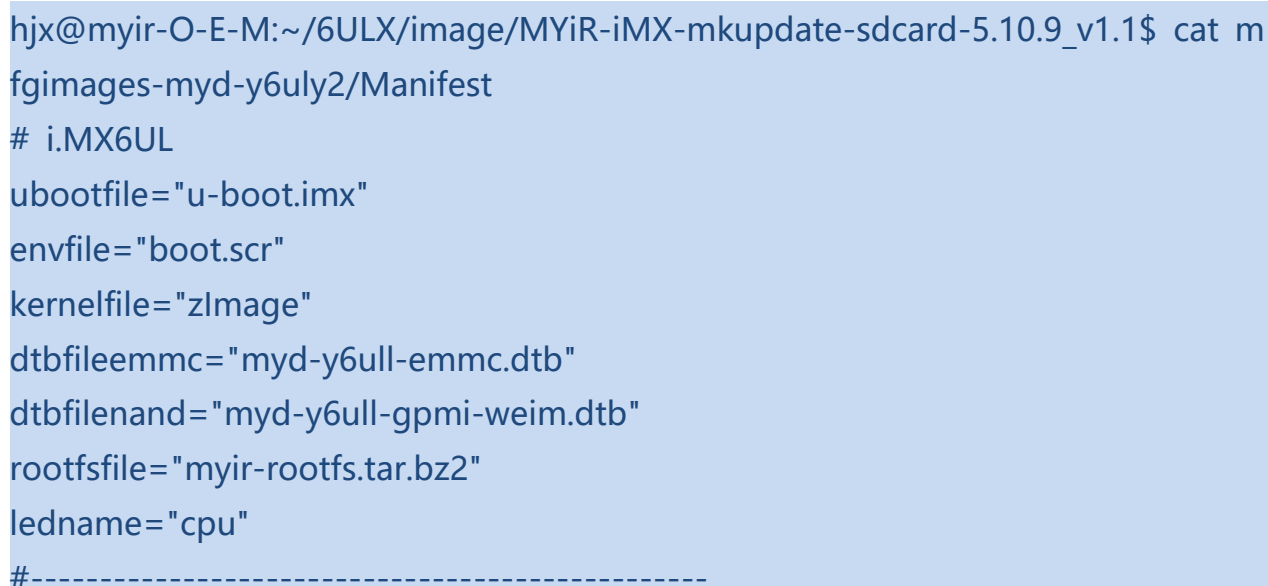


```

hjk@myir-0-E-M:~/6ULX/image/MYiR-mkupdate-sdcard-5.10.9_v1.1$ ls -l
总用量 32
-rwxr-xr-x 1 hjx hjx 11803 12月 10 2020 build-sdcard-5.10.9.sh
drwxr-xr-x 2 hjx hjx 4096 8月 12 17:15 firmware
drwxr-xr-x 2 hjx hjx 4096 8月 12 12:03 mfgimages-my-d-y6ulg2
drwxr-xr-x 2 hjx hjx 4096 8月 12 12:01 mfgimages-my-d-y6uly2
-rw-r--r-- 1 hjx hjx 763 8月 12 12:06 README.MD
drwxr-xr-x 2 hjx hjx 4096 8月 17 17:31 rootfs
hjk@myir-0-E-M:~/6ULX/image/MYiR-mkupdate-sdcard-5.10.9_v1.1$
  
```

Figure 4-3. Sdcard tools

The build-sdcard-5.10.9.sh script is used to make an image to update the system from the SD card. The firmware in the firmware folder is only used to boot from the SD card. Generally, no modification is required. mfgimages-my-d-y6ulg2, mfgimages-my-d-y6ulg2 The firmware stored in the roofs and roofs folders will eventually be burned to the flash of the board; the Manifest text in the mfgimages-my-d-\* folder specifies the burned file name, the red font in the following structure.



```

hjk@myir-0-E-M:~/6ULX/image/MYiR-mkupdate-sdcard-5.10.9_v1.1$ cat m
fimages-my-d-y6uly2/Manifest
# i.MX6UL
ubootfile="u-boot.imx"
envfile="boot.scr"
kernelfile="zImage"
dtbfileemmc="myd-y6ull-emmc.dtb"
dtbfilenand="myd-y6ull-gpmi-weim.dtb"
rootfsfile="myir-rootfs.tar.bz2"
ledname="cpu"
#-----
  
```

```
# user update
```

```
# i.MX6UL-Y2 -- emmc
UBOOT_EMMC256DDR="u-boot-dtb-y2-ddr256-emmc.imx"
UBOOT_EMMC512DDR="u-boot-dtb-y2-ddr512-emmc.imx"
DTBFILE_EMMC="myd-y6ull-emmc.dtb"

# i.MX6UL-Y2 -- nand
UBOOT_NAND256DDR="u-boot-dtb-y2-ddr256-nand.imx"
UBOOT_NAND512DDR="u-boot-dtb-y2-ddr512-nand.imx"
DTBFILE_NAND="myd-y6ull-gpmi-weim.dtb"

KERNELFILE="zImage"

#ROOTFSFILE="rootfs-update.tar.bz2"
```

If you modify the kernel or uboot, replace it in the mfgimages-myd-y6uly2 folder, and note that the file name is consistent with the one defined in the Manifest, or replace it directly and keep the original file name.

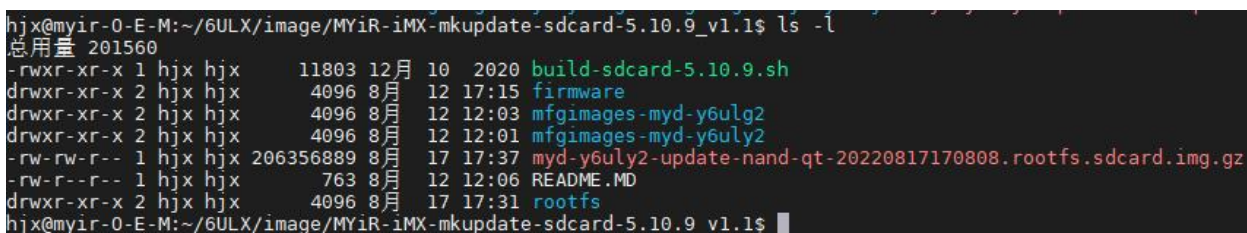
The build-sdcard-5.10.9.sh script provides four parameters:

- '-p' represents the platform, the available parameter is "myd-y6uly2" representing MYD-Y6ULL (MYC-Y6ULY2)
- '-n' means the memory chip on the board is NAND
- '-e' means the on-board memory chip is eMMC
- '-d' indicates the directory of the update file '-s' indicates the size of ddr memory
- '-f' indicates the type of roofs, supports qt and core parameters



## 1) Make image file for SD card upgrade

```
$ cd MYiR-iMX-mkupdate-sdcard-5.10.9_v1.1
$ sudo ./build-sdcard-5.10.9.sh -p myd-y6uly2 -n -d mfgimages-myd-y6uly2
-s 256 -f qt
$ sudo ./build-sdcard-5.10.9.sh -p myd-y6uly2 -n -d mfgimages-myd-y6uly2
-s 256 -f core
```



```
hjsx@myir-0-E-M:~/6ULX/image/MYiR-iMX-mkupdate-sdcard-5.10.9_v1.1$ ls -l
总用量 201560
-rwxr-xr-x 1 hjsx hjsx    11803 12月 10  2020 build-sdcard-5.10.9.sh
drwxr-xr-x 2 hjsx hjsx     4096 8月 12  17:15 firmware
drwxr-xr-x 2 hjsx hjsx     4096 8月 12  12:03 mfgimages-myd-y6ulg2
drwxr-xr-x 2 hjsx hjsx     4096 8月 12  12:01 mfgimages-myd-y6uly2
-rw-rw-r-- 1 hjsx hjsx 206356889 8月 17  17:37 myd-y6uly2-update-nand-qt-20220817170808.rootfs.sdcard.img.gz
-rw-r--r-- 1 hjsx hjsx     763 8月 12  12:06 README.MD
drwxr-xr-x 2 hjsx hjsx     4096 8月 17  17:31 rootfs
hjsx@myir-0-E-M:~/6ULX/image/MYiR-iMX-mkupdate-sdcard-5.10.9_v1.1$
```

Figure 4-4. Completed SD card upgrade image

As shown in the above figure, myd-y6uly2-update-nand-qt-20220817170808.rootfs.sdcard.img.gz is the generated image file compression package.

## 2) Make SD Card that can update the system

In the previous step, an updated image of the SD card was made, and then it was written to the SD card.

### ● Windows

Windows users can use Win32DiskImager tool to write xxxx.rootfs.sdcard.img image into Micro SD. The tool is in the "03\_Tools" directory. After decompression, double-click the "Win32DiskImager.exe" application. In the interface after startup, the "Device" on the right is to select the drive letter of the device to be written. The "image file" on the left is to select the image file to be written, click the folder icon next to it, and select the file to be written (Note: The default in the file selection dialog box is to filter the ".img" file, which can be switched to ".\*").

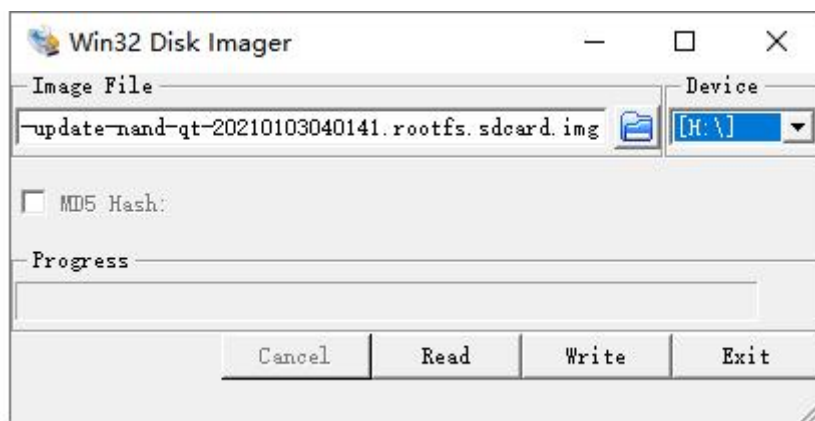


Figure 4-5. Win32disk flashes the image to SD card

- **Linux system**

Generally, linux use "sd[x][n]" format to naming a storage device. The x means which storage device, represent use a ~ z character. The n means partition that storage device, use digit start from 1. You can use "dmesg | tail" command to view device name when you plugin Card Reader. In this case, we use "/dev/sdb" as example.

**Attention: the "/dev/sdb" do not append any digit**

Write sdcard file into USB storage:

```
$ gzip -dc myd-y6uly2-update-nand-qt-20220817170808.rootfs.sdcard.img.gz  
| sudo dd of=/dev/sdb conv=fsync
```

Insert the prepared SD card into the card slot (J8) of the development board, configure the DIP switch (SW1) of the development board as SDCARD boot mode, connect the USB to TTL serial cable to the debug serial port (JP1), and configure the serial terminal on the computer side software. Use a DC 12V power adapter to connect to the power interface (J22) of the development board. Through the serial port, you can see that the system boots from the Micro SD card, execute the update script, and write the Linux system image file into the NAND memory chip. The current update status can also be judged by the user LED light (D30). The update status is flashing. After the update is successful, it will always be on, and it will be off if it fails. After the update is completed, power off and configure the start position dial switch to be the onboard NAND or eMMC start mode.



## 5. How to Modify Board Level Support Package

The previous chapter has described how to build a system image that can run on MYD-Y6ULX development board based on Yocto project. In addition, it also describes in detail the process of burning the image to the development board. Because many pins of MYC-Y6ULX CPU module have the characteristics of multi-function reuse, there will always be some differences between the base-board based on CPU module and MYB-Y6ULX module in the actual project. These differences are mainly reflected in two aspects, one is due to the hardware differences, such as the removal of display function, the addition of more GPIO, the addition of more serial ports, or the expansion of some peripherals through SPI, I2C, USB, etc; the other is reflected in the differences of system components, for example, those focusing on HMI applications may need a relatively complete graphics system and Qt library, while those focusing on background management applications may need more complete network applications and python running environment. Because of these differences, it is necessary for system developers to do some deletion and migration based on the code we provide. Therefore, this chapter will describe the specific process of developing and customizing your own system from the perspective of a system developer, so as to lay a foundation for the later adaptation of their own hardware.

### 5.1. Introduction to meta-myir 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 nxp based board. For example, the meta-myir layer is based on the NXP official meta-imx layer, which is suitable for MYD-Y6ULX 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 NXP Linux distributions by creating a custom Yocto project layer. The contents of meta-myr layer are as follows:

```

├── EULA.txt
├── meta-bsp
│   ├── classes
│   ├── conf
│   ├── recipes-bsp
│   ├── recipes-connectivity
│   ├── recipes-core
│   ├── recipes-daemons
│   ├── recipes-devtools
│   ├── recipes-fsl
│   ├── recipes-graphics
│   ├── recipes-kernel
│   ├── recipes-multimedia
│   ├── recipes-myr
│   ├── recipes-security
│   ├── recipes-support
│   ├── recipes-test
│   └── recipes-utils
├── meta-ml
│   ├── conf
│   ├── recipes-devtools
│   └── recipes-libraries
├── meta-sdk
│   ├── classes
│   ├── conf
│   ├── dynamic-layers
│   ├── recipes-connectivity
│   ├── recipes-devtools
│   ├── recipes-extended
│   ├── recipes-fsl
│   ├── recipes-graphics
│   ├── recipes-multimedia
│   ├── recipes-sato
│   └── recipes-support
├── README
├── SCR-5.10.9_1.0.0.txt
├── tools
│   ├── myir-setup-release.sh
│   ├── readme-bluez.txt
│   └── setup-utils.sh

```

34 directories, 6 files

Table 5-1. meta-myir layer content description

Source Code and Data	Description
conf	Including development board software configuration resource information
recipes-bsp	contains TF-A and uboot configuration resources,etc
recipes-kernel	Contains Linux kernel resources and third-party firmware 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. meta-myir

IP Project	Description
conf	includes development board software configuration resource information
meta-bsp/recipes-bsp	contains configuration resources such as uboot
meta-bsp/recipes-kernel	contains linux kernel resources and third-party firmware resources
meta-bsp/recipes-myir	contains file system configuration information and applications, such as hmi v2.0.

Generally, according to different stages of startup, we divide BSP into bootloader and kernel. Users can refer to the contents of recipes-bsp and recipes-kernel in meta-myir.

The recipes BSP only contains the u-boot of bootloader, which mainly implements the initialization of core hardware, such as DDR, clock and kernel boot.

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

## 5.3. U-Boot Compilation

U-Boot ("the Universal Boot Loader" or U-Boot) is an open-source bootloader that can be used on NXP 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> .

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

### 5.3.1. Get U-boot source code

Use myir-imx-uboot.tar.gz directly under the working directory or use the following command to download the latest source code:

```
git clone https://github.com/MYiR-Dev/myir-imx-uboot.git -b develop_2020.04
```

#### 1) Configuration and Compilation

On the host machine, set the environment with the following command before building for i.MX6UL SoC:

```
myir$ source /opt/test5.10/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

- **Compile source code**

```
myir$ cd myir-imx-uboot
myir$ make distclean
myir$ make <config>
myir$ make -j16
```

<config> is the name of the configuration option. Different startup modes need to use the following different configuration options. MYD-Y6ULX development board has multiple options.

After compiling, u-boot-dtb.imx is generated in the current directory, which is the generated Uboot object file

Table 5-3. uboot configuration list

Compilation List	Description
myd_imx6ull_nand_ddr256_defconfig	MYD-Y6ULY2-V2-256N256D(configuration file)
myd_imx6ull_emmc_defconfig	MYD-Y6ULY2-V2-4E512D(configuration file)
myd_imx6ul_nand_ddr256_defconfig	MYD-Y6ULG2-V2-256N256D(configuration file)
myd_imx6ul_emmc_defconfig	MYD-Y6ULG2-V2-4E512D(configuration file)

### 5.3.2. Use Yocto to Compile uboot

The bb file that specifies the location of the uboot source code in Yocto:

```
sources/meta-myr/meta-bsp/recipes-bsp/u-boot/u-boot-common.inc
UBOOT_SRC ?= "git://github.com/MYiR-Dev/myir-imx-uboot.git;protocol=https"
SRCBRANCH = "develop_2020.04"
SRC_URI = "${UBOOT_SRC};branch=${SRCBRANCH} \
"
#SRCREV = "${AUTOREV}"
SRCREV = "2e966da26f1edd6122e4567247a2338de6aa29e0"
```

- UBOOT\_SRC: uboot code
- SRCBRANCH: branch name
- SRCREV: Corresponding value of commit, set to \${AUTOREV}, the latest commit will be used automatically

If you modify the source code of uboot, you need to submit the modification record, generate a new commit and write it into the u-boot-common.inc configuration file, update the value of SRCREV, and then compile the firmware using your modified code.

#### ● Set the environment

After Yocto exits or is interrupted halfway, you can reopen a new shell terminal and reload the build directory. The command is as follows

```
myir$: source myir-setup-release.sh -b build_imx6ull
```

- **Compiling**

```
myir$ bitbake u-boot -c clean
myir$ bitbake u-boot -c cleansstate
myir$ bitbake u-boot
```

Use Ycoto to compile the generated uboot in the Yocto compilation directory:

```
myir$ build_imx6ull/tmp/deploy/images/myd-y6ull14x14/
```

### 5.3.3. Configure Yocto to use local uboot source code

By default, u-boot-common.inc specifies the source code address of github. When compiling for the first time, it will be pulled from github and then compiled. Later, if users debug and modify the source code and build their own source code library, it will be more convenient to store locally.

```
#UBOOT_SRC ?= "git://github.com/MYiR-Dev/myir-imx-uboot.git;protocol=https"
"
UBOOT_SRC = "git:/// ${HOME}/MYD-Y6ULX-devel/04_Sources/myir-imx-uboot;
protocol=file"
SRCBRANCH = "develop_2020.04"
SRC_URI = "${UBOOT_SRC};branch=${SRCBRANCH} \
"

#SRCREV = "${AUTOREV}"
SRCREV = "2e966da26f1edd6122e4567247a2338de6aa29e0"
```

Modify the UBOOT\_SRC to specify the location of the uboot source code extracted from the working directory created in the previous chapter. After modification, the local source code will be used for subsequent compilation with yocto.

## 5.4. Kernel Compilation

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, i.MX6UL 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-Y6ULX adapts to ST open source community version kernel version, and currently supports the latest version of Linux kernel 5.10.9.

### 5.4.1. Get Kernel source code

Use myir-imx-linux.tar.gz directly under the working directory or use the following command to download the latest source code:

```
git clone https://github.com/MYiR-Dev/myir-imx-linux.git -b develop_lf-5.10.y
```

### 5.4.2. Configuration and Compilation

On the host machine, set the environment with the following command before building for i.MX6UL SoC:



```
myir$ source /opt/test5.10/environment-setup-cortexa7hf-neon-poky-linux-gnu-
eabi
```

- **Compile source code**

```
myir$ tar -xvf myir-imx-linux.tar.gz
myir$ cd myir-imx-linux
myir$ make distclean
myir$ make myd_y6ulx_defconfig
myir$ make zImage dtbs -j16
```

After compiling, the kernel image file zImage will be generated in the "arch/arm/boot" directory, and the DTB file will be generated in the "arch/arm/boot/dts" directory.

Table 5-4. DTB List

DTB File	Description
myd-y6ull-emmc.dtb	MYD-Y6ULY2-V2-4E512D (dtb file)
myd-y6ull-gpmi-weim.dtb	MYD-Y6ULY2-V2-256N256D(dtb file)
myd-y6ul-emmc.dtb	MYD-Y6ULG2-V2-4E512D (dtb file)
myd-y6ul-gpmi-weim.dtb	MYD-Y6ULG2-V2-256N256D(dtb file)

### 5.4.3. Use Yocto to Compilation kernel

The bb file that specifies the location of the kernel source code in Yocto:

```
sources/meta-myr/meta-bsp/recipes-kernel/linux/linux-imx_5.10.bb
```

```
KERNEL_BRANCH ?= "develop_lf-5.10.y"
LOCALVERSION = "-1.0.0"
KERNEL_SRC ?= "git://github.com/MYiR-Dev/myir-imx-linux.git;protocol=https"
SRC_URI = "${KERNEL_SRC};branch=${KERNEL_BRANCH}"
#SRCREV = "${AUTOREV}"
SRCREV = "7c0cb551c77eaa18f4b0333fd5f55c147dad7557"
```

- KERNEL\_SRC : kernel code
- KERNEL\_BRANCH: branch name
- SRCREV: Corresponding value of commit, set to \${AUTOREV}, the latest commit will be used automatically

- **Set the environment**

After Yocto exits or is interrupted halfway, you can reopen a new shell terminal and reload the build directory. The command is as follows

```
myir$: source myir-setup-release.sh -b build_imx6ull
```

- **Compiling**

```
myir$ bitbake linux-imx -c clean
myir$ bitbake linux-imx -c cleansstate
myir$ bitbake linux-imx
```

Use Yocto to compile the generated zImage in the Yocto compilation directory:

```
build_imx6ull/tmp/deploy/images/myd-y6ull14x14/
```

#### 5.4.4. Configure Yocto to use local kernel source code

linux-imx\_5.10.bb specifies the source code address of github by default. When compiling it for the first time, it will be pulled from github and then compiled. Subsequent users will debug and modify the source code and build their own source code library. It will be convenient to store it locally. The reference modification is as follows:

```
KERNEL_BRANCH ?= "develop_lf-5.10.y"
LOCALVERSION = "-1.0.0"
#KERNEL_SRC ?= "git://github.com/MYiR-Dev/myir-imx-linux.git;protocol=https"
KERNEL_SRC = "git:///${HOME}/MYD-Y6ULX-devel/04_Sources/myir-imx-linux;p
rotocol=file"
SRC_URI = "${KERNEL_SRC};branch=${KERNEL_BRANCH}"
#SRCREV = "${AUTOREV}"
SRCREV = "7c0cb551c77eaa18f4b0333fd5f55c147dad7557"
```

In KERNEL\_SRC, modify the location of the kernel source code that is decompressed to the working directory created in the previous chapter. After modification, the local source code will be used for subsequent compilation with yocto.

## 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-Y6ULX development board of MYIR. For specific information, please refer to “MYD-Y6ULX\_SDK Release Notes” .In addition, users also need to refer to the CPU chip manual and the product manual of MYC-Y6ULX 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 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-Y6ULX board, which is very helpful for the development reference of users.

Table 6-1.MYD-Y6ULX Device-tree List

Project	Device Tree	Description
U-Boot	imx6ull-y2.dtsi \ imx6ul-g2.dtsi	Peripheral resource device tree
	myb-imx6ul-14x14-base.dts	The basic dts of MYC-Y6ULX-G2
	myb-imx6ull-14x14-base.dts	The basic dts of MYC-Y6ULX-Y2
	myb-imx6ul-14x14-gpmi-weim.dts	MYC-Y6ULX-G2 nand board dts
	myb-imx6ul-14x14-emmc.dts	MYC-Y6ULX-G2 emmc board dts
	myb-imx6ull-14x14-gpmi-weim.dts	MYC-Y6ULX-Y2 nand board dts
	myb-imx6ull-14x14-emmc.dts	MYC-Y6ULX-Y2 emmc board dts
kernel	myd_y6ulx_defconfig	Kernel configure
	myb-imx6ul-14x14.dtsi	MYC-Y6ULX public part of dtsi
	myb-imx6ul-14x14-base.dts	MYC-Y6ULX-G2 basic dts

	myb-imx6ull-14x14-base.dts	MYC-Y6ULX-Y2 basic dts
	myd-y6ul-emmc.dts	MYC-Y6ULX-G2 emmc board dts
	myd-y6ul-gpmi-weim.dts	MYC-Y6ULX-G2 nand board dts
	myd-y6ull-emmc.dts	MYC-Y6ULX-Y2 emmc board dts
	myd-y6ull-gpmi-weim.dts	MYC-Y6ULX-Y2 nand board dts

### 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-imx6ul-14x14-base-xxx.dts.

```
// Path: arch/arm/boot/dts
myir$cd myir-imx-linux/arch/arm/boot/dts
myir$ ls -l myb-imx6ul* dts -l
-rw-rw-r-- 1 myir myir 274 9月 18 19:39 myb-imx6ul-14x14-base.dts
-rw-rw-r-- 1 myir myir 4460 9月 18 19:39 myb-imx6ull-14x14-base.dts
myir$ ls -l myd*dts -l
-rw-rw-r-- 1 myir myir 485 9月 18 19:39 myd-y6ul-emmc.dts
-rw-rw-r-- 1 myir myir 1822 9月 18 19:39 myd-y6ul-gpmi-weim.dts
-rw-rw-r-- 1 myir myir 512 9月 18 19:39 myd-y6ull-emmc.dts
-rw-rw-r-- 1 myir myir 1756 9月 18 19:39 myd-y6ull-gpmi-weim.dts
```

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

```
// SPDX-License-Identifier: GPL-2.0
//
// Copyright (C) 2015 Freescale Semiconductor, Inc.

/ {
    chosen {
        stdout-path = &uart1;
    };

    memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>;
    };

    reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;

        linux,cma {
            compatible = "shared-dma-pool";
            reusable;
            size = <0x8000000>;
            linux,cma-default;
        };
    };
};

.....
```

For more detailed header file inclusion, please refer to our device tree file:myb-imx6ull-14x14-base.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_SOC_IMX6UL) += \
    .....
    myb-imx6ul-14x14-base.dtb \
    myd-y6ul-gpmi-weim.dtb \
    myd-y6ul-emmc.dtb \
    myb-imx6ul-14x14-base-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 myd-y6ulx-xxx.dtb, please refer to section 5.4 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 i.MX6UL 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 i.MX6UL device is generally configured by MX6 Pins Tool software or manually by referring to datasheet.

### **1) How to configure peripherals using MX6 Pins Tool**

The i.MX application processor of the i.MX6 Pins Tool is the successor to the processor expert® software i.MX processor. The new Pins Tool makes pin configuration easier and faster through an intuitive and easy-to-use user interface, and then generates ordinary C code, which can then be used in any C and C++ applications. The pin tool can configure the pin signal (from multiplexing (multiplexing) to the electrical characteristics of the pin). It can also create a device tree summary include (.dtsi) file and report it in CSV format. This section does not focus on explaining its usage, but introduces important parameters. You can obtain detailed development instructions through the official website.

NXP official:

<https://www.nxp.com/design/designs/pins-tool-for-i-mx-application-processors:PINS-TOOL-IMX>

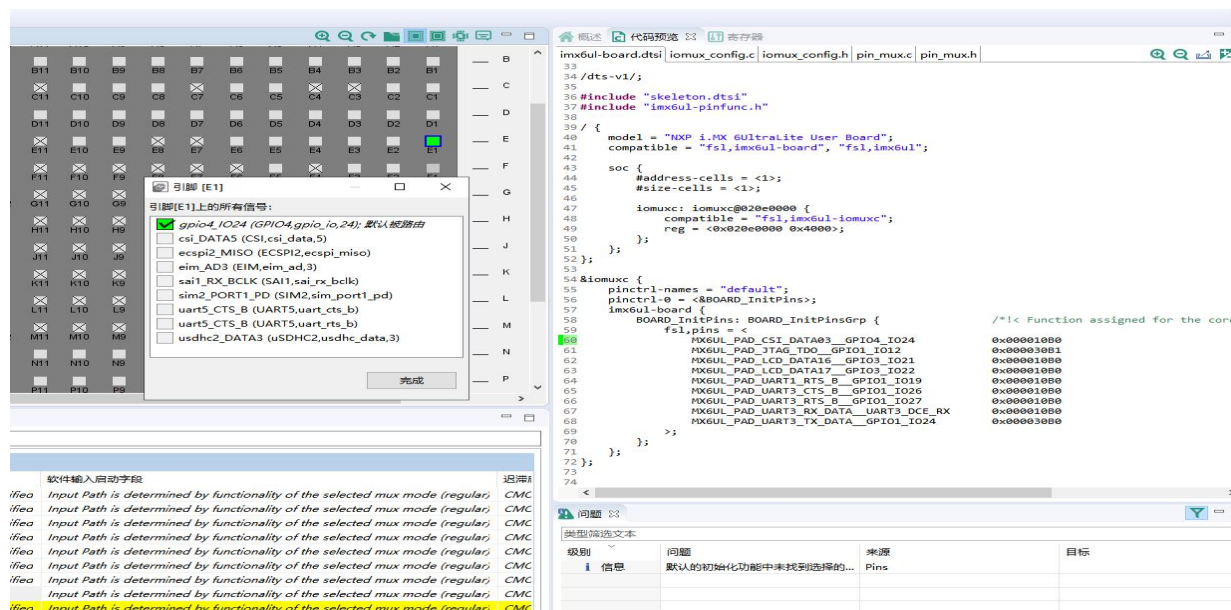


Figure 6-2. dts code

Select a PIN as shown in the figure above, and check a function to generate the corresponding dts code in the right area.

## 2) Configure GPIO in the device tree

This example uses J14's PIN5 (MX6UL\_PAD\_UART3\_TX\_DATA\_GPIO1\_IO24) as the test GPIO. Introduce how to configure the device node in the device tree, and provide the kernel driver for the following chapters.

myir-imx-linux/arch/arm/boot/dts/myb-imx6ul-14x14.dtsi

\*\*\*\*\*

```
gpioctr_device {
    compatible = "myir,gpioctr";
    status = "okay";
    gpioctr-gpios = <&gpio1 24 0>;
};

reg_can_3v3: regulator@0 {
    compatible = "regulator-fixed";
    reg = <0>;
```



```

        regulator-name = "can-3v3";
        regulator-min-microvolt = <3300000>;
        regulator-max-microvolt = <3300000>;
    };

*****

&iomuxc {
    pinctrl-names = "default";
    pinctrl-0 = <&BOARD_InitPins>;
    imx6ul-board {
        BOARD_InitPins: BOARD_InitPinsGrp {
            /*!< Function assigned for the core: Cortex-A7[ca7] */
            fsl,pins = <
                MX6UL_PAD_CSI_DATA03__GPIO4_IO24      0x000010B0
                MX6UL_PAD_JTAG_TDO__GPIO1_IO12          0x000030B1
                MX6UL_PAD_LCD_DATA16__GPIO3_IO21        0x000010B0
                MX6UL_PAD_LCD_DATA17__GPIO3_IO22        0x000010B0
                MX6UL_PAD_UART1_RTS_B__GPIO1_IO19        0x000010B0
                MX6UL_PAD_UART3_CTS_B__GPIO1_IO26        0x000010B0
                MX6UL_PAD_UART3_RTS_B__GPIO1_IO27        0x000010B0
                MX6UL_PAD_UART3_RX_DATA__UART3_DCE_RX    0x000010B0
                0
                MX6UL_PAD_UART3_TX_DATA__GPIO1_IO24      0x000030B0
            >;
        };
    };
};

```

## 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 gpio1\_24, use the following command in uboot shell.

Cmd:

```
=> gpio set 24 1
gpio: pin 24 (gpio 24) value is 1
=> gpio clear 24 1
gpio: pin 24 (gpio 24) value is 0
=>
```

Code examle:

```
myir-imx-uboot/board/myir/myd_imx6ull_14x14/myd_imx6ull14x14.c
int board_init(void)
{

    /* LCD Power */
    imx_iomux_v3_setup_multiple_pads(lcd_pwr_pads, ARRAY_SIZE(lcd_pwr_p
ads));

    gpio_request(IMX_GPIO_NR(3, 4), "power");
    gpio_direction_output(IMX_GPIO_NR(3, 4) , 1);

    *****
```

#### 2) GPIO control through device tree

You can also use the device tree to define the node of IO resources, and then implement the IO function in the code, such as the power reset control of WiFi & BT.

```
// File: arch/arm/dts/myb-imx6ul-14x14-base.dts

&fec2 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_enet2>;
    phy-mode = "rmii";
    phy-handle = <&ethphy1>;
    phy-reset-gpios = <&gpio5 6 GPIO_ACTIVE_LOW>;
    phy-reset-duration = <26>;
    phy-reset-post-delay = <20>;
    status = "okay";

    mdio {
        #address-cells = <1>;
        #size-cells = <0>;

        ethphy0: ethernet-phy@0 {
            compatible = "ethernet-phy-ieee802.3-c22";
            reg = <0>;
        };
    };
};

*****

// file: uboot/drivers/net/fec_mxc.c
/* FEC GPIO reset */
static void fec_gpio_reset(struct fec_priv *priv)
{
    debug("fec_gpio_reset: fec_gpio_reset(dev)\n");
    if (dm_gpio_is_valid(&priv->phy_reset_gpio)) {
        dm_gpio_set_value(&priv->phy_reset_gpio, 1);
        mdelay(priv->reset_delay);
    }
}
```

```
        dm_gpio_set_value(&priv->phy_reset_gpio, 0);
        if (priv->reset_post_delay)
            mdelay(priv->reset_post_delay);
    }
}
#endif
```

### 6.3.2. How to use GPIO in Kernel driver

#### 1) How to use independent GPIO driver

In section 6.2.1, 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 J14-PIN5 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(gpiocr_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: gpioctr-gpios=<...>; */
    gpioctr_gpio = gpiod_get(&pdev->dev, "gpioctr", 0);
    if (IS_ERR(gpioctr_gpio)) {
        dev_err(&pdev->dev, "Failed to get GPIO for led\n");
        return PTR_ERR(gpioctr_gpio);
    }

    /* Register file_operations */

```

```

    major = register_chrdev(0, "myir_gpiotr", &gpiotr_drv); /* /dev/gpiotr
r */

    gpiotr_class = class_create(THIS_MODULE, "myir_gpiotr_class");
    if (IS_ERR(gpiotr_class)) {
        printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
        unregister_chrdev(major, "gpiotr");
        gpiod_put(gpiotr_gpio);
        return PTR_ERR(gpiotr_class);
    }

    device_create(gpiotr_class, NULL, MKDEV(major, 0), NULL, "myir_gpiotr
r%d", 0);

    return 0;
}

static int chip_demo_gpio_remove(struct platform_device *pdev)
{
    device_destroy(gpiotr_class, MKDEV(major, 0));
    class_destroy(gpiotr_class);
    unregister_chrdev(major, "myir_gpiotr");
    gpiod_put(gpiotr_gpio);

    return 0;
}

static const struct of_device_id myir_gpiotr[] = {
    { .compatible = "myir,gpiotr" },
    { },
};

/* define platform_driver */

```

```

static struct platform_driver chip_demo_gpio_driver = {
    .probe      = chip_demo_gpio_probe,
    .remove     = chip_demo_gpio_remove,
    .driver     = {
        .name    = "myir_gpiocr",
        .of_match_table = myir_gpiocr,
    },
};

/* 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 unregistered, 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 `gpioctr.c` file in the `drivers/char/` folder of the kernel source code, copy the above driver code into it, and modify `Kconfig`, `Makefile`, and `myd_y6ulx_defconfig`.

Add configuration in `kconfig` file:

```
// drivers/char/Kconfig

config SAMPLE_GPIO
    tristate "this is a gpio test driver"
    depends on CONFIG_GPIOLIB
```

Edit `makefile`:

```
// drivers/char/Makefile
...
obj-$(CONFIG_SAMPLE_GPIO) += gpioctr.o
```

Add the configuration item in `myd_y6ulx_defconfig` file:

```
//linux/arch/arm/configs/myd_y6ulx_defconfig
CONFIG_SAMPLE_GPIO=y
```

Then compile and update the kernel according to section 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 = KERN_DIR = /home/myir/myir-imx-linux/

obj-m += gpiocr.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:

```
myir$ source /opt/test5.10/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

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

```
myir$:/home/myir/demo_gpiocr$ make
make -C /home/myir/myir-imx-linux/ M=`pwd` modules
make[1]: Entering directory '/home/myir/myir-imx-linux'
  CC [M] /home/myir/demo_gpiocr/gpiocr.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M] /home/myir/demo_gpiocr/gpiocr.mod.o
  LD [M] /home/myir/demo_gpiocr/gpiocr.ko
make[1]: Leaving directory '/home/myir/myir-imx-linux'
```

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-Y6ULX\_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 gpiocdev device node file under *"/dev"* directory, such as *"/dev/gpiocdev0"* corresponds to GPIO0, *"/dev/gpiocdev1"* corresponds to GPIO1, 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:

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

This application toggles GPIO1\_24(J14-PIN5):

```
//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/gpiochip0");

    /* Open device: gpiochip0 for GPIO1 */
    fd = open(chrdev_name, 0);
    if (fd == -1) {
        ret = -errno;
        fprintf(stderr, "Failed to open %s\n", chrdev_name);
    }

    return ret;
}
```

```
}

/* request GPIO line: GPIO1_24 */
req.lineoffsets[0] = 24;
req.flags = GPIOHANDLE_REQUEST_OUTPUT;
memcpy(req.default_values, &data, sizeof(req.default_values));
strcpy(req.consumer_label, "gpio1_24");
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:

```
myir$ source /opt/test5.10/environment-setup-cortexa7hf-neon-poky-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@myd-y6ull14x14:~# 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:

```
myir$ source /opt/test5.10/environment-setup-cortexa7hf-neon-poky-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_gpiotctr0 on
root@myir:~# gpiotest /dev/myir_gpiotctr0 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: main.c
#include "module.h"
void sample_func();
int main()
{
    sample_func();
    return 0;
}

```

```

// File: module.h

#include <stdio.h>
void sample_func();

```

```

// File: module.c

```

```
#include "module.h"
void sample_func()
{
    printf("Hello World!");
    printf("\n");
}
```

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:

```
myir$ source /opt/test5.10/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

Create makefile:

```
# CC="gcc"
all: main.o module.o
    ${CC} main.o module.o -o target_bin
main.o: main.c module.h
    ${CC} -I . -c main.c
module.o: module.c module.h
    ${CC} -I . -c module.c
clean:
    rm -rf *.o
    rm target_bin
```

Execute make command to generate executable file:

```
myir$ 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:

```
# ./target_bin
Hello World!
```

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-Y6ULX uses Qt version 5.15 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/](http://download.qt.io/development_releases/qtcreator/) .

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-y6ulx QT application development notes” 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 “04-Sources/” directory and you can find the MEasy HMI2.0 project source code(*mxapp2.tar.gz*). Then it can be compiled and debugged remotely through qtcreator. For more details, please refer to “MYD-Y6ULX QT Application Development Notes” and MEasy+HMI2.0+Development+Guide.pdf” .

## 7.3. Automatic application startup at boot time

### 7.3.1 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 bitmake 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-openstlinuxeglf-myr$ bitbake -s | grep proftpd
```

proftpd

:1.3.6-r0

Note: it can be seen that the proftpd recipe, version 1.3.6-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-full
```

- **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 -axu | grep proftpd
nobody      584  0.0  0.3   3032   1344 ?        Ss   01:51   0:00 proftpd: (a
ccepting connections)
root        1713  0.0  0.0   1776    336 pts/0    S+   01:59   0:00 grep proftp
d
```

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-myr-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 site(<https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#ref-classes-useradd>) ,then rebuild the image. The specific method will not be discussed here.

- **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>.

### 7.3.2 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 *sources/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_1.3.6.bb
```

1 directory, 8 files

- proftpd\_1.3.6.bb: Recipe for building proftpd service
- proftpd.service: Auto start service at boot time
- proftpd-basic.init: Start script for proftpd

The "*proftpd\_1.3.6.bb*" recipe file specifies the source code path to obtain proftpd service program and some patch files for this version of source code:

```
SRC_URI = "ftp://ftp.proftpd.org/distrib/source/${BPN}-${PV}.tar.gz \
    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
rmkdir ${D}${libdir}/proftpd ${D}${datadir}/locale
[ -d ${D}${libexecdir} ] && rmkdir ${D}${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
pd
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.pamd ${D}${sysconfdir}/pam.d/
proftpd
    sed -i '/ftusers/d' ${D}${sysconfdir}/pam.d/proftpd
    # specify the user Authentication config
    sed -i '/^MaxInstances/a\AuthPAM                                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\_ 1.3.6.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 sources/meta-myir/meta-bsp/recipes-myir/ directory.

## 7.4. QT Application

Qt is a cross-platform graphics application development framework, which is applied on different size devices and platforms, and provides different copyright versions for users to choose from. MYD-Y6ULX uses Qt 5.15 version for application development. In Qt application development, it is recommended to use the QtCreator integrated development environment, which can develop Qt applications under Linux PC and automatically cross-compile to the ARM architecture of the development board. This chapter uses the SDK tool built by Yocto as a cross-compilation system and cooperates with QtCreator to quickly develop graphical applications. Before starting this chapter, please complete the Yocto build process in Chapter 3, or use the pre-compiled SDK toolkit provided on the CD. Before starting this chapter, please install the application SDK development tools.

### 7.4.1. Install QtCreator

The QtCreator installation package is a binary program that can be installed directly by executing it.

```
$ cd $DEV_ROOT
$ chmod a+x 03_Tools/qt-opensource-linux-x64-5.9.4.run
$ sudo 03_Tools/Qt/qt-opensource-linux-x64-5.9.4.run
```

After executing the installation program, keep clicking Next to complete. The default installation directory is `"/opt/"`.

After installation, in order for QtCreator to use Yocto's SDK tools, environment variables need to be added to QtCreator. Modify the `"/opt/qtcreator-5.9.4/Tools/QtCreator/bin/qtcreator.sh"` file and add the Yocto environment configuration before `"#!/bin/sh"`, refer to the following:

```
myir$ source /opt/test5.10/environment-setup-cortexa7t2hf-neon-poky-linux-gn
ueabi
#!/bin/sh
# Use this script if you add paths to LD_LIBRARY_PATH
```

```
# that contain libraries that conflict with the  
# libraries that Qt Creator depends on.
```

When using QtCreator, please execute "qtcreator.sh" from the terminal to start QtCreator. Refer to the following

```
myir$ /opt/qtcreator-5.9.4/Tools/QtCreator/bin/qtcreator.sh &
```

### 7.4.2. Configure QtCreator

Using qtcreator to compile programs available for the target board requires configuration of the compilation chain. The following configuration items need to be reset:

- Configure GCC and G++ compilation chain
- Configure QTversion
- Configure QTdebug
- Configure device
- Create kit

#### 1) Configure GCC and G++ compilation chain

After running QtCreator, click "Tool"->"Options" in turn, an option dialog box appears, click "Build & Run" on the left, and select the "Compilers" tab on the right. Click the "Add" button on the right, after the drop-down list pops up, select "C" in "GCC", fill in the "Name" below as "MYDY6ULx-GCC", and click the "Browse.." button next to "Compiler path" to select Go to /opt/test5.10/sysroots/x86\_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc and click "Apply".

Click the "Add" button on the right again, after the drop-down list pops up, select "C++" in "GCC", fill in "Name" as "MYDY6ULx-GCC++" below, and click the "Browse.." button next to "Compiler path" Select /opt/test5.10/sysroots/x86\_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-g++ and click "Apply".

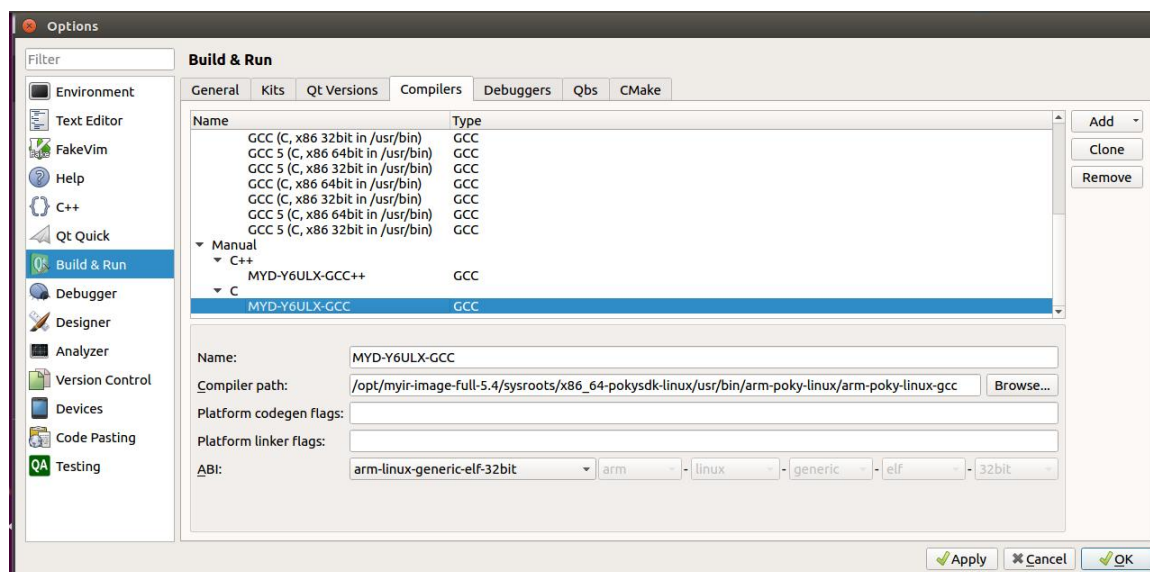


Figure 7-1. Configure GCC

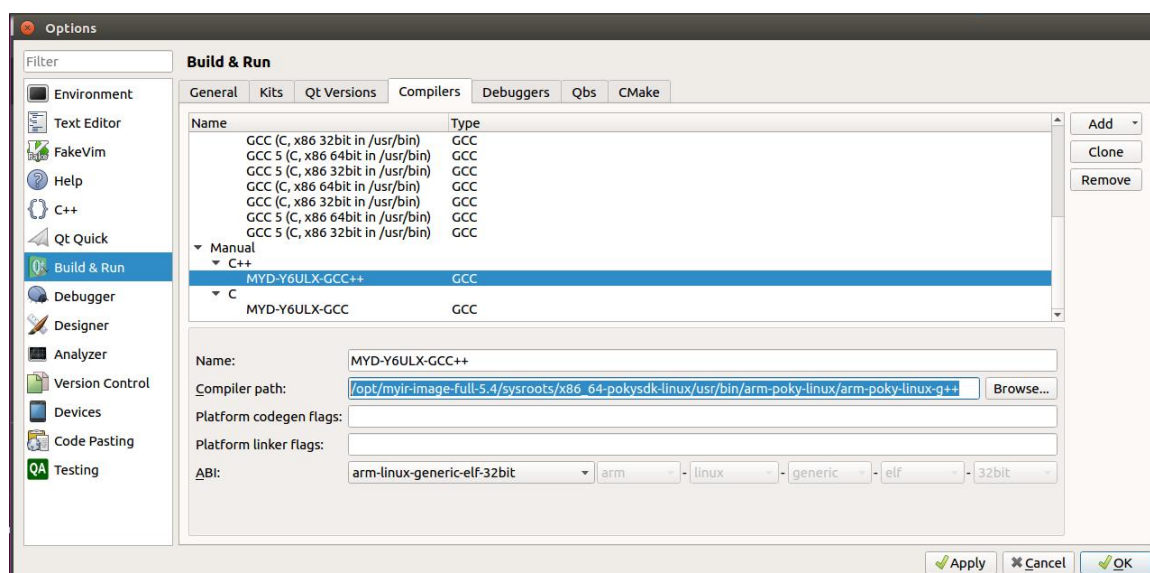


Figure 7-2. Configure G++

## 2) Configure Qtversion

Select the "Qt Version" tab, click "Add..." on the right side, and the qmake path selection dialog box will pop up, where it is named "/opt/test5.10/sysroots/x86\_64-pokysdklinux/usr/bin/qmake" is an example. After selecting the "qmake" file, click the "Open" button. "Version name" is changed to "Qt %{Qt:Version} (System)". Then click the "Apply" button.

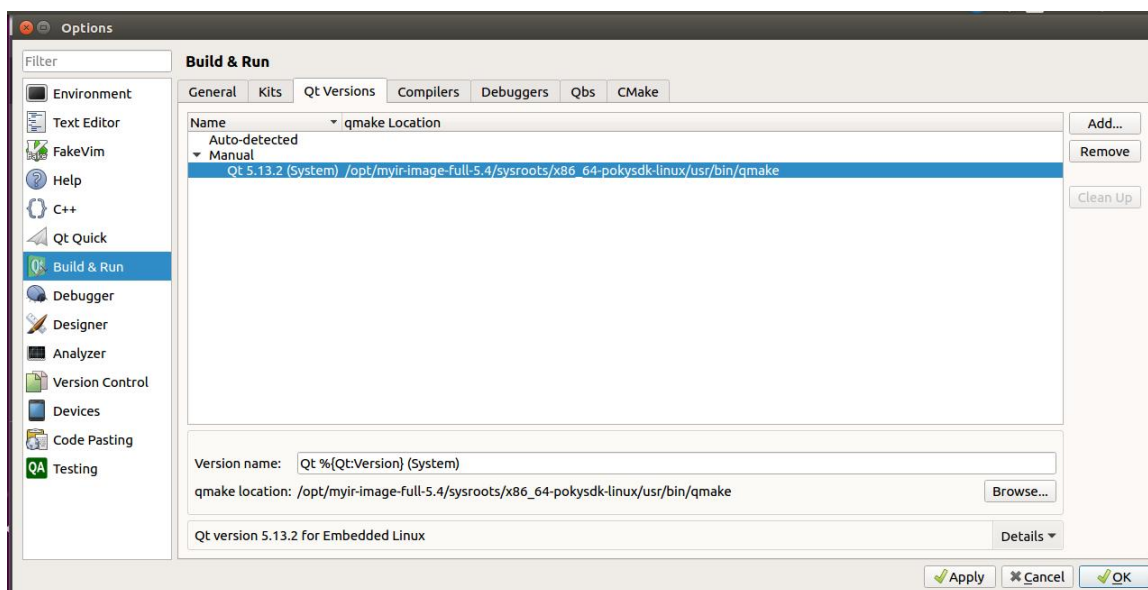


Figure 7-3. Configure Qtversion

### 3) Configure Debug

Select "Debuggers" on the right side, click the "Add..." button on the right, and fill in the content "Name" as "MYD-Y6ULX-DEBUG" in the pop-up dialog box. The path path is:

`/opt/test5.10/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gdb`

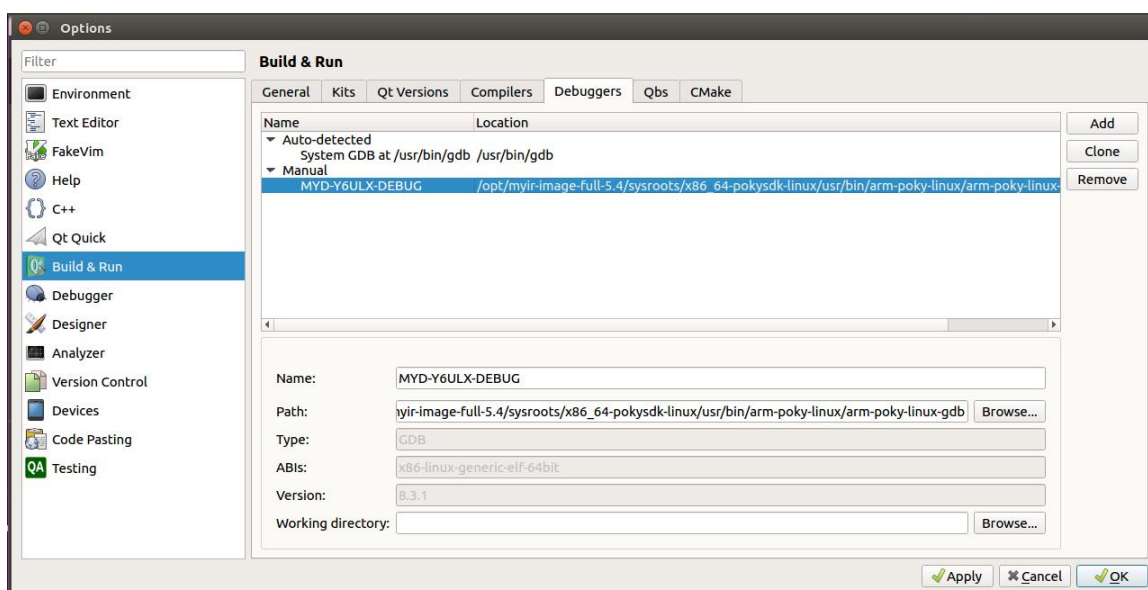


Figure 7-4. Configure Debug

### 4) Configure Device



Select "Device" on the left, click the "Add..." button on the right, select Generic Linux Device in the pop-up dialog box, and fill in the content "Name" as "MYDY6ULx Board" and "Host name" as the IP of the development board Address (you can fill in any address temporarily), "Username" is "root", and then click "Apply".

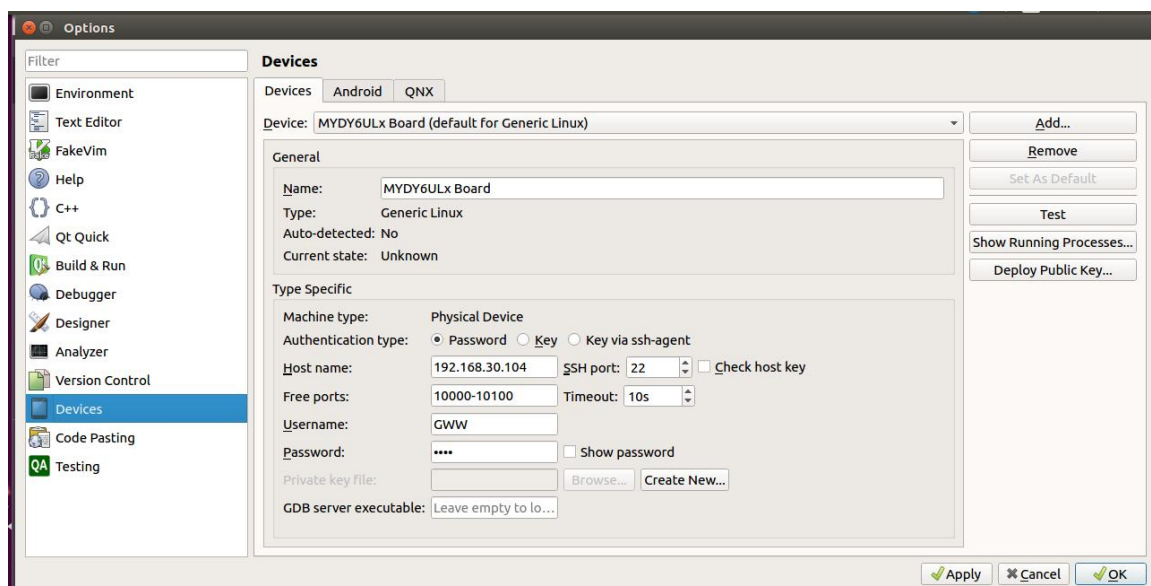


Figure 7-5. Configure DEVICE

## 5) Create kit

Click "Build & Run" on the left to return to the "Kits" tab, "Name" is "MYD-Y6ULX", and "Device" selects the "MYDY6ULx Board" option. "Sysroot" selects the system directory of the target device, here "/opt/test5.10/sysroots" is taken as an example. "Compiler" selects the previously configured names "MYDY6ULx-GCC" and "MYDY6ULx-GCC++", "Qt version" selects the previously configured name "Qt 5.15.0 (System)", and "Qt mkspec" fills in "linux-oe-" g++". Other defaults are fine, and finally click the "Apply" and "OK" buttons.

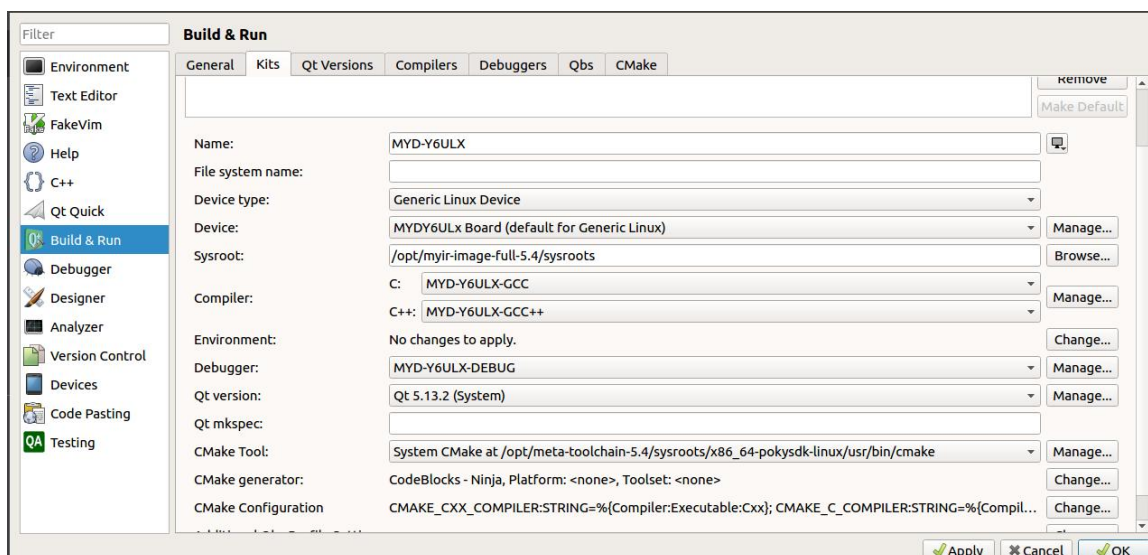


Figure 7-6. Create Kit

### 7.4.3. Test Qt Application

Create a new qt project to run the test.

The first step is to select "File" -> "New File or Project" in the menu bar, the project name is: Y6ULX\_TESSET.

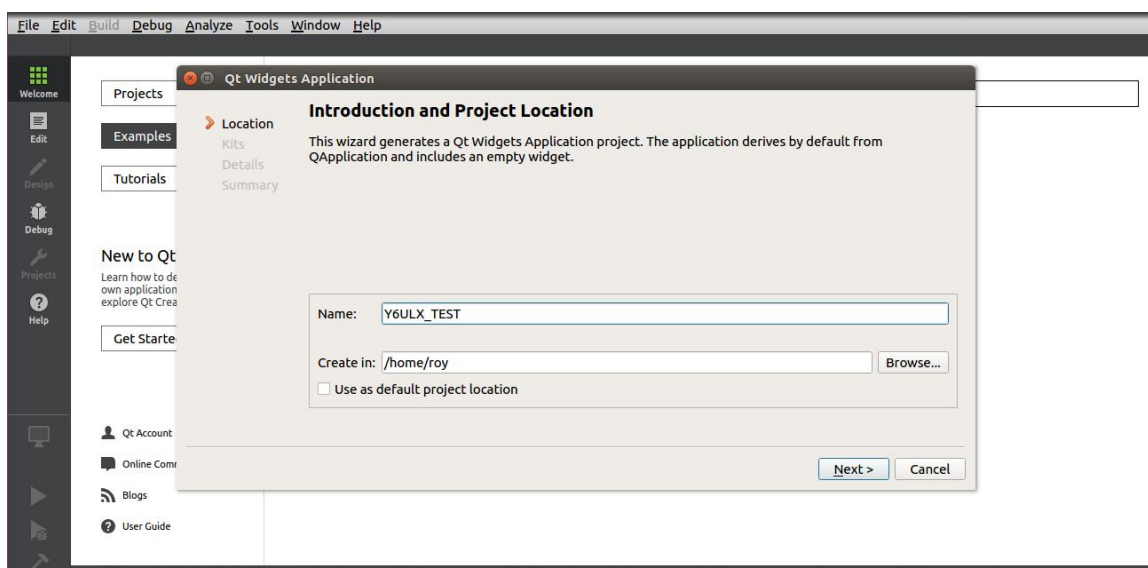


Figure 7-7. Create Project

In the second step, after the project is opened, select the "MYD-Y6ULX" kit to complete the creation.

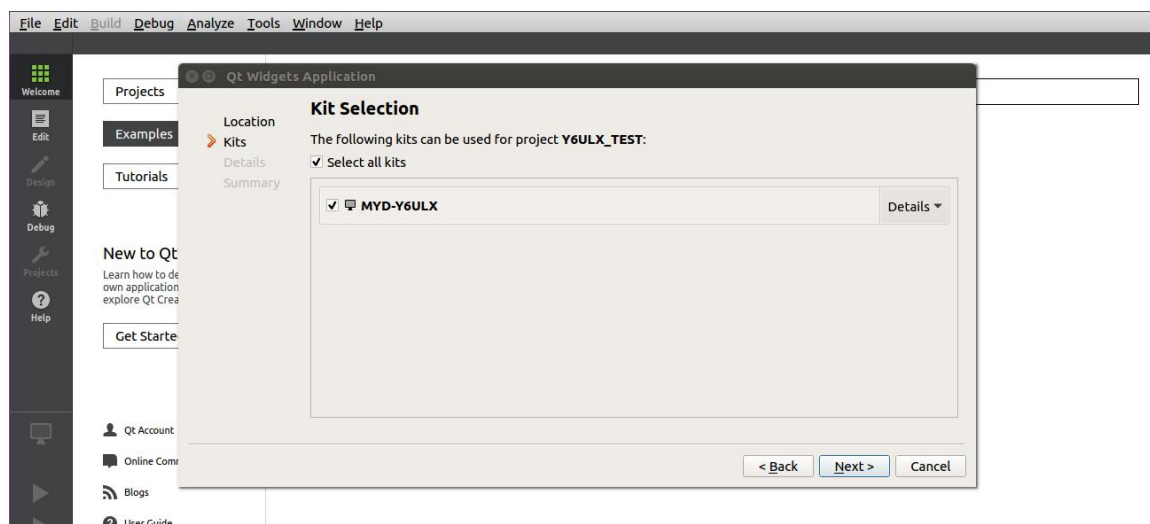


Figure 7-8. Select Kit

The third step is to click the "Build"->"Build Project Y6ULX\_TESSET" button in the menu bar to complete the compilation of the project, and the compilation process will be output at the bottom.

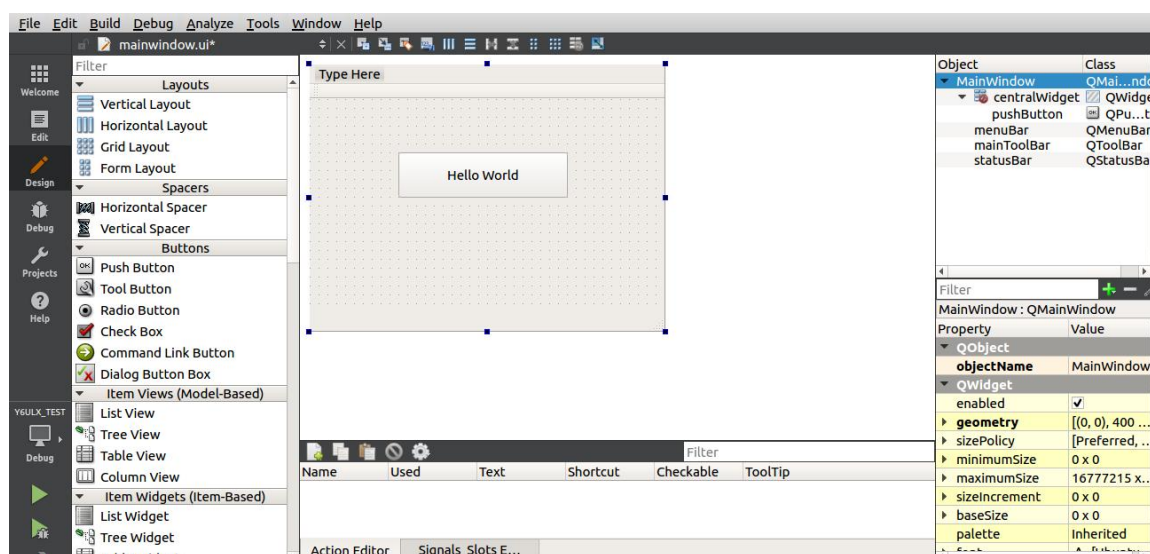


Figure 7-9. Build Qt Application

After QtCreator builds the Y6ULX\_TEST item, the compiled binary file is stored in the "build-Y6ULX\_TEST-MYD\_Y6ULX-Debug" directory. You can use the file command to check whether it is compiled to the ARM architecture.

```
# file Y6ULX_TEST
```

Y6ULX\_TEST: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-, BuildID[sha1]=76a3b0c7d335dd49f6613aee3a2b32e5a537d97b, for GNU/Linux 3.2.0, not stripped

Then copy the Y6ULX\_TEST file to the development board and run it.

```
# ./Y6ULX_TEST --platform linuxfb
```

After running, you will see the Qt window interface on the LCD screen as follows:

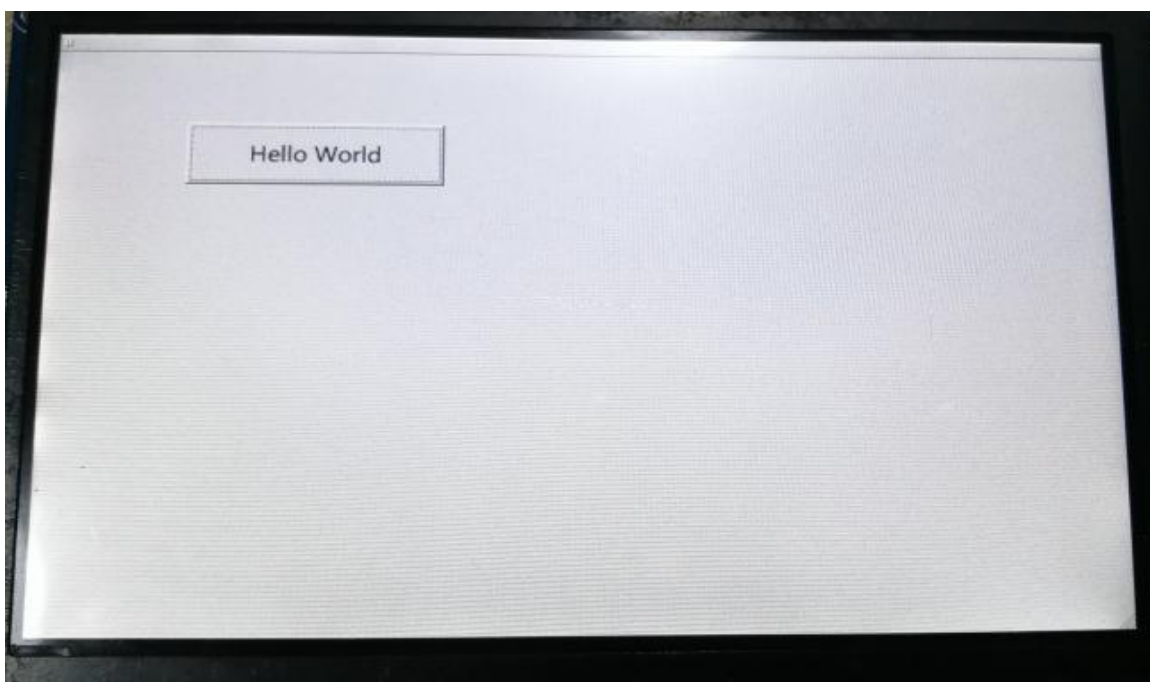


Figure 7-10. Qt Application

## 8. Reference

- **Linux kernel open source community**

<https://www.kernel.org/>

- **Yocto Development Guide**

<https://www.yoctoproject.org/>

- **Yocto Project BSP Development Guide**

<https://www.yoctoproject.org/docs/3.1.1/bsp-guide/bsp-guide.html>

- **Yocto Project Linux Kernel Development Guide**

<https://www.yoctoproject.org/docs/3.1.1/kernel-dev/kernel-dev.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 Inteiligent Park, Bantian, Longgang District.

Support Email: [support@myirtech.com](mailto:support@myirtech.com)

Sales Email: [sales@myirtech.com](mailto:sales@myirtech.com)

Phone: +86-755-22984836

Fax: +86-755-25532724

Website: [www.myirtech.com](http://www.myirtech.com)