MYD-Y6ULX Linux 软件开发指南

文件状态: [ ] 草稿 [√] 正式发布	**文件标识:**	MYIR-i.MX6UL-SW-DG-ZH-L5.4.3
	当前版本:	V2.0.1
	作　　者:	Alex
	创建日期:	2020-08-04
	最近更新:	2020-11-26

版 本 历 史

版本	作者	参与者	日期	备注
V2.0.1	Alex		202001204	Linux 系统开发指导，适用于 MYD-Y6ULX 开发板

目 录

1. 概述

Linux 系统平台上有许多开源的系统构建框架，这些框架方便了开发者进行嵌入式系统的构建和定制化开发，目前比较常见的有 Buildroot, Yocto, OpenEmbedded 等等。其中 Yocto 项目使用更强大和定制化的方法，来构建出适合嵌入式产品的 Linux 系统。Yocto 不仅仅是一个制做文件系统工具，同时提供整套的基于 Linux 的开发和维护工作流程，使底层嵌入式开发者和上层应用开发者在统一的框架下开发，解决了传统开发方式下零散和无管理的开发形态。

本文首先讲述在 MYD-Y6ULX 系列开发板上使用 Yocto 项目安装运行 Linux 系统以及嵌入式 Linux 驱动和应用程序的开发流程，其中包括部署开发环境、构建系统、Linux 应用程序的实例分析、镜像的更新等。系统开发人员熟悉第三章的 Yocto 的开发流程之后，就可以参照第四章的移植指南针对实际项目需求对 BSP 进行差异化的定制，就可以将很快的将系统移植到基于 MYC-Y6ULX 核心板设计的硬件平台之上。

本文档并不包含 Yocto 项目以及 Linux 系统相关基础知识的介绍，适合有一定开发经验的嵌入式 Linux 系统开发人员。针对用户在进行二次开发过程中可能会使用到的一些具体功能，我们也提供了详细的应用开发手册供发人员参考，具体的信息参见《MYD-Y6ULX SDK 发布说明》表 2-4 中的文档列表。

1.1. 硬件资源

MYD-Y6ULX 系列开发板是米尔电子基于 NXP 公司的高性能嵌入式 ARM 处理器 i.MX 6ULX 系列开发的一套开发平台。该开发平台由核心板 MYC-Y6ULX-及底板 MYB-Y6ULX 两部分组成。

1). MYC-Y6ULX:

核心板采用 NXP 公司 i.MX6ULX 处理器作为主控平台、 DDR3、 eMMC/Nand Flash、百兆以太网 PHY。核心板底板采用邮票孔焊接的方式，有助于降低成本，也能确保核心板与底板连接的稳固性。

图 1-1. MYC-Y6ULX 核心板

2). MYB-Y6ULX:

底板集成了丰富的外设资源，支持 12V DC 输入方式供电。主要有 RGB、CSI、WIFI、USB HOST、RS232、RS485、CAN、SPI 等外设接口。

图 1-2. 底板正面

关于 MYD-Y6ULX 开发板的更详细硬件资料可以查看光盘镜像中的产品手册，也可以关注网站上的产品链接(http://www.myir-tech.com/product/myc_y6ulx.htm)，了解该产品信息的实时更新。

1.2. 软件资源

MYD-Y6ULX 搭载基于 Linux 5.4.3 版本内核的操作系统，提供了丰富的系统资源和其他软件资源。开发板出厂附带嵌入式 Linux 系统开发所需要的交叉编译工具链，U-Boot 源

代码，Linux 内核和各驱动模块的源代码，以及应用开发样例等。具体的包含的软件信息请参考《MYD-Y6ULX SDK 发布说明》中第 2 章软件信息中的说明。

1.3. 文档资源

根据用户使用开发板的各个不同阶段，SDK 中包含了发布说明，入门指南，评估指南，开发指南，应用笔记，常用问答等不同类别的文档和手册。具体的文档列表参见《MYD-Y6ULX SDK 发布说明》表 2-4 中的说明。

2. 部署开发环境

本章主要介绍基于 MYD-Y6ULX 开发板进行系统移植，应用开发，固件烧录等整个开发流程所需的一些软硬件环境，包括必要的硬件配件，软件工具等，具体的准备工作下面将进行详细介绍。

2.1. 硬件环境

1). 必要配件

- 12V 电源适配器
- 不少于 4GB SD 卡
- USB 转 TTL 调试线（调试串口使用），注意使用 3.3V 电平等

2). 启动设置

本节主要介绍开发板的启动方式，以便用户更好的选择启动方式。

表 2-1. 拨码启动方式

启动模式	拨码状态(B1/B2/B3/B4)	备注
eMMC 启动	OFF/OFF/ON/OFF	
NAND Flash 启动	OFF/ON/ON/OFF	
TF Card 启动(eMMC 版本)	ON/ON/ON/OFF	
TF Card 启动(NAND 版本)	ON/OFF/ON/OFF	
USB Download	X/X/OFF/ON	USB_OTG1 下载系统镜像

3). 串口配置

将 USB 转 TTL 线正确接到调试串口 JP1，USB 端连到 PC 上，并使用调试软件设置 PC 串口的波特率设为 115200，数据位为 8，停止位为 1，无奇偶校验。

表 2-2. 调试串口波特率设置

波特率	数据位	停止位	校验位	其他
115200	8	1	No	No

使用 12V 电源适配器连接开发板 J22 接口，通电后即可正常启动。在 PC 端串口调试下将打印出启动信息。

注意：默认出厂的系统用户名为 root，密码为空。

2.2. 软件环境

本章节将介绍如何搭建 i.MX6UL 的开发环境。通过阅读本章节，您将了解相关硬件工具，软件开发调试工具的安装和使用。并能快速的搭建相关开发环境，为后面的开发和调试做准备。

2.2.1. 获取资料

搭建环境前先下载开发板资料，关于开发资料的详细信息请查阅《MYD-Y6ULX SDK 发布说明》。

开发板资料下载地址如下（资料会不定期更新，请下载最新版）：

http://down.myir-tech.com/MYD-Y6ULX/

2.2.2. 搭建编译环境

- **操作系统说明**

安装 Linux 操作系统,推荐使用 Ubuntu16.04 64bit，内存至少 4G，硬盘至少 150G，构建过程需要下载软件包，所以要配置好 Linux 系统的网络，使其能联网。

- **在 Linux 下安装必备的软件包**

```
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
```

hufan@myir-server1:~$

- **建立工作目录：**

建立工作目录，方便设置统一的环境变量路径。拷贝产品光盘中的源码到工作目录下，同时设置 DEV_ROOT 变量，方便后续步骤的路径访问，所拷贝的代码用户和拥有者必须是个人账号，不能用 root 账号操作.

```
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. 安装 SDK

SDK 提供了一个独立的交叉开发工具链和库，这些库是根据板子上烧写的特定镜像的内容定制的。下面以米尔公司定制的 SDK 为例安装 SDK，SDK 在开发包资料 03_Tools/Tools_chain/目录下

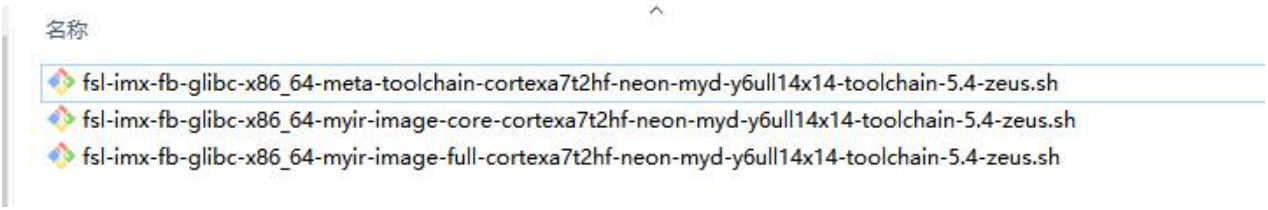

图 2-1. 交叉编译工具链

1). SDK 安装步骤

下载 SDK 压缩包，然后将 SDK 压缩包拷贝到 Ubuntu 下的工作目录，解压文件，得到安装脚本文件。此 SDK 包已经包含了交叉工具链，Linux 运行库。

Yocto 提供的工具链有两种，一种是底层开发的 meta-toolchain，用于编译 uboot 和 kernel，另一种是用于应用开发的工具链。前者和 Linaro 类似，后者包含应用开发中的相关库，可以直接使用 pkg-config 工具来解决头文件或库文件的依赖关系。MYD-Y6ULX 的资源包中有提供两种工具链，三个编译工具链。

表 2-3. 工具链说明

工具链文件名	描述
fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.sh	meta-toolchain 基础工具链
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.sh	myir-image-full 系统的应用工具链
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.sh	myir-image-core 系统的应用工具链

meta-toolchain 的编译工具链用于编译 uboot 和 kernel，myir-image-full 和 myir-image-core 用于编译应用程序，myir-image-full 支持的系统库较全（支持 QT5.13），myir-image-core 不支持 QT。

Yocto 编译器是以 SDK 工具包方式来提供，需要先安装 SDK 包后，才可以使用。安装方法如下,以普通用户权限执行 shell 脚本，运行中会提示安装路径，默认在/opt 目录下，用户可修改安装目录，同时会提示输入用户密码以便有写入目录的权限。安装完成后，可以使用"source"或"."命令加 载工链接环境到当前终端。

此处以 meta-toolchain 为例安装方式如下,红色字体为作者此处设置的安装目录，用户可以自己定义：

```
myir$ ./fsl-imx-fb-glibc-x86_64-myir-image-full-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.sh
NXP i.MX Release Distro SDK installer version 5.4-zeus
==========================================================
Enter target directory for SDK (default: /opt/fsl-imx-fb/5.4-zeus): /home/alex/workspace/meta_toolschain_imx6ul
You are about to install the SDK to "/home/alex/workspace/tools_qt5_imx6". 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.
 $ . /home/alex/workspace/meta_toolschain_imx6ul/environment-setup-cortexa7t2hf-neon-poky-linux-gnueabi
```

设置环境变量，并测试安装是否完成

```
myir$source /home/alex/workspace/meta_toolschain_imx6ul/environment-setup-cortexa7t2hf-neon-poky-linux-gnueabi
myir$ $CC --version
arm-poky-linux-gnueabi-gcc (GCC) 9.2.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

其他编译器也是同样方法安装，请指定不同目录，请勿使用相同目录，出现文件相互覆盖情形。

3. 使用 Yocto 构建开发板镜像

3.1. 简介

Yocto 是一个开源的“umbrella”项目，意指它下面有很多个子项目，Yocto 只是把所有的项目整合在一起，同时提供一个参考构建项目 Poky，来指导开发人员如何应用这些项目，构建出嵌入式 Linux 系统。它包含 Bitbake、 OpenEmbedded-Core, 板级支持包，各种软件包的配置文件。可以构建出不同类需求的系统，如带 Qt5.13 图形库的 myir-image-full 系统，全功能命令行系统 myir-image-core。 MYD-Y6ULX 提供了符合 Yocto 的配置文件，帮助开发者构建出可烧写在 MYD-Y6ULX 板上的 Linux 系统像。Yocto 还提供了丰富的开发文档资源，让开发者学习并定制自己的系统。由于篇幅有限，不能完整介绍 Yocto 的使用，请用户自行上网搜索。

本节适合需要对文件系统进行深度定制的开发者，希望从 Yocto 构建出符合 MYD-Y6ULX 系列开发板的文件系统，同时基于它的定制需求。初次体验使用或无特殊需要的开发者可以直接使用 MYD-Y6ULX 已经提供的文件系统。由于 Yocto 构建前需要下载文件系统中所有软件包到本地，为了快速构建，MYD-Y6ULX 已经把相关的软件打包好，可以直接解压使用，减少重复下载的时间。 myir 提供了两种获取 yocto 的方法，请选择一种合适的即可。

下面以构建 myir-image-full 镜像为例进行介绍具体的开发流程，为后续定制适合自己的系统镜像打下基础。

注意：构建 Yocto 不需要加载 2.3 节中的 SDK 工具链环境变量，请创建新 shell 或打开新的终端窗口。

3.2. 获取源码

1). 下载米尔官方 ISO 资源包

源码都在米尔开发包资料的 04_Sources 目录，MYiR-i.MX6UL-Yocto.tar.gz 为 Yocto 的源码包，将其解压：

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

2). 通过 github 获取

目前 i.MX6UL 开发板的 BSP 源代码和 Yocto 源代码均使用了 github 托管并将保持长期更新，可以直接从 git 下克隆最新的代码。

表 3-1. 源码列表

	Github	分支名
Manifest	https://github.com/MYiR-Dev/myir-imx-manifest	i.MX6UL-5.4-zeus
meta-myir-imx	https://github.com/MYiR-Dev/meta-myir-imx	i.MX6UL-5.4-zeus
U-Boot	https://github.com/MYiR-Dev/myir-imx-uboot	develop
Kernel	https://github.com/MYiR-Dev/myir-imx-linux	develop

把 03_tools/Repo/repo 文件放到 /usr/bin 目录，加上可执行权限。

然后使用 repo 拉取 Yocto 源码：

```
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-5.4.3-2.0.0.xml -b  i.MX6UL-5.4-zeus
myir$:repo sync
```

执行 repo sync 会去下载代码，需要一定的时间，请耐心等待，如下图代码下载正在下载中：

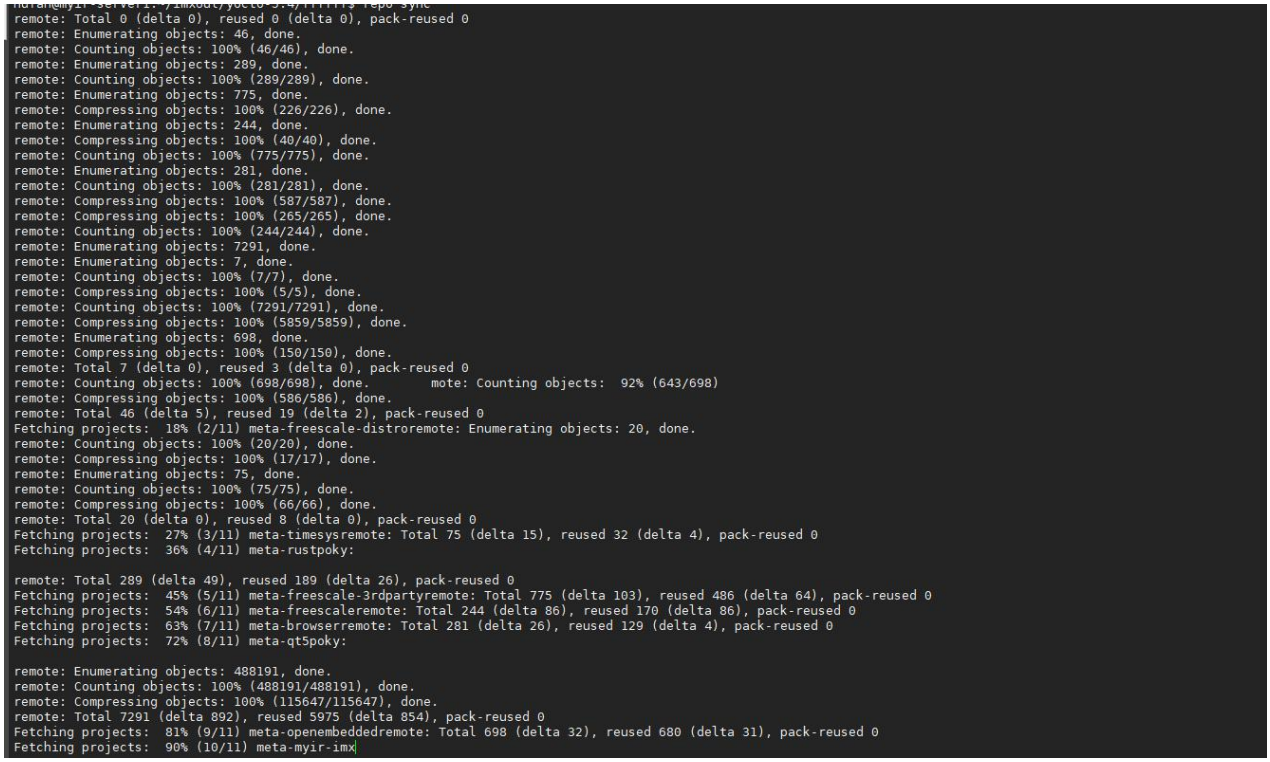

图 3-1. repo 拉取代码

3.3. Yocto 构建系统

解压工作目录的 MYiR-i.MX6UL-Yocto.tar.gz 文件，或者通过 repo 下载了文件后，会存生成如下文件：

```
├── 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
```

图 3-2. Yocto 源码目录预览

为了减少 Yocto 的构建时间，请将 downloads 解压到此目录下，以减少下载软件包的时间。

```
myir$ cd $DEV_ROOT/Yocto-5.4
myir$ tar  -xzf  downloads.tar.gz
  -C  ./
```

1). 构建系统：

使用提供的 myir-setup-release.sh 脚本，会创建一个工作空间，然后在此空间下构建镜像。执行脚本后会先要求阅读并同意版权声明后才会进入构盡过目录。同时，脚本会默认创建并进入 build 目录。如果需要特定目录名称，可以使用-b 参数，如"-b myir"。 MACHINE 可选参数为"myd-y6ull14x14"或"myd-y6ul14x14"，分别对应 MYD-Y6ULL（MYC-Y6UL-Y2）和 MYD-Y6UL(MYC-Y6UL-G2)开发板。

```
myir$: $ DISTRO=fsl-imx-fb MACHINE=myd-y6ull14x14 source myir-setup-release.sh -b build_imx6ul
myir@myir-server1:~/imx6ul/yocto-5.4$ tree -L 1
.
├── build_imx6ul
├── 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
```

- **构建 GUI Qt5 版的系统：**

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

- **构建非 GUI 版的系统：**

```
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:a8f6e31bebc5a551fab1fec8d67489af80878f71"
meta-oe
meta-multimedia
meta-python          = "HEAD:bb65c27a772723dfe2c15b5e1b27bcc1a1ed884c"
meta-freescale       = "HEAD:94f4f086c6014cbcfd10bda3540d19558c8bf0b0"
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:5f365ef0f842ba4651efe88787cf9c63bc8b6cb3"
meta-rust            = "HEAD:d8d77be1292064a02adcb5e72e293604b704f69b"
meta-gnome
meta-networking
meta-filesystems     = "HEAD:bb65c27a772723dfe2c15b5e1b27bcc1a1ed884c"
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)
```

图 3-3. 构建信息

```
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:a8f6e31bebc5a551fab1fec8d67489af80878f71"
meta-oe
meta-multimedia
meta-python          = "HEAD:bb65c27a772723dfe2c15b5e1b27bcc1a1ed884c"
meta-freescale       = "HEAD:94f4f086c6014cbcfd10bda3540d19558c8bf0b0"
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:5f365ef0f842ba4651efe88787cf9c63bc8b6cb3"
meta-rust            = "HEAD:d8d77be1292064a02adcb5e72e293604b704f69b"
meta-gnome
meta-networking
meta-filesystems     = "HEAD:bb65c27a772723dfe2c15b5e1b27bcc1a1ed884c"
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
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$
```

图 3-4. 构建完成

在建文件系统完成后，会在输出目录下有 manifest 文件，这个文件里包含了对应文件系统中已安装的软件包。Yocto 第一次构建会需要很长时间，取决于计算机的 CPU 核心数和硬件读写速度。Yocto 建议可以使用八核和 SSD 硬盘可以加速构建速度。第一次构建完

成后会生成缓存，后面修改的构建，时间会减少很多。在构建完成后在会" tmp/deploy/images/myd-y6ull14x14/"目录下生成不同的文件，以下是构建后生成的文件信息列表：

```
hufan@myir-server1:~/imx6ul/yocto-5.4/build_imx$ ls tmp/deploy/images/myd-y6ull14x14/
modules--5.4-r0-myd-y6ull14x14-20200920045735.tgz                myir-image-full.env
modules-myd-y6ull14x14.tgz                                       myir-image-full-imx-uboot-bootpart.wks
myb-imx6ull-14x14-base--5.4-r0-myd-y6ull14x14-20200920045735.dtb myir-image-full-myd-y6ull14x14-20200921021923.rootfs.manifest
myb-imx6ull-14x14-base.dtb                                       myir-image-full-myd-y6ull14x14-20200921021923.rootfs.sdcard.bz2
myb-imx6ull-14x14-base-myd-y6ull14x14.dtb                        myir-image-full-myd-y6ull14x14-20200921021923.rootfs.tar.bz2
myd-y6ull-emmc--5.4-r0-myd-y6ull14x14-20200920045735.dtb         myir-image-full-myd-y6ull14x14-20200921021923.rootfs.wic.bmap
myd-y6ull-emmc.dtb                                               myir-image-full-myd-y6ull14x14-20200921021923.rootfs.wic.bz2
myd-y6ull-emmc-myd-y6ull14x14.dtb                                myir-image-full-myd-y6ull14x14-20200921021923.testdata.json
myd-y6ull-gpmi-weim--5.4-r0-myd-y6ull14x14-20200920045735.dtb    myir-image-full-myd-y6ull14x14.manifest
myd-y6ull-gpmi-weim.dtb                                          myir-image-full-myd-y6ull14x14.sdcard.bz2
myd-y6ull-gpmi-weim-myd-y6ull14x14.dtb                           myir-image-full-myd-y6ull14x14.tar.bz2
myir-image-core.env                                              myir-image-full-myd-y6ull14x14.testdata.json
myir-image-core-imx-uboot-bootpart.wks                           myir-image-full-myd-y6ull14x14.wic.bmap
myir-image-core-myd-y6ull14x14-20200920093904.rootfs.manifest    myir-image-full-myd-y6ull14x14.wic.bz2
myir-image-core-myd-y6ull14x14-20200920093904.rootfs.sdcard.bz2  tee.bin
myir-image-core-myd-y6ull14x14-20200920093904.rootfs.tar.bz2     tee.mx6ullevk.bin
myir-image-core-myd-y6ull14x14-20200920093904.rootfs.wic.bmap    u-boot.imx
myir-image-core-myd-y6ull14x14-20200920093904.rootfs.wic.bz2     u-boot.imx-sd-optee
myir-image-core-myd-y6ull14x14-20200920093904.testdata.json      u-boot-myd-y6ull14x14.imx
myir-image-core-myd-y6ull14x14.manifest                          u-boot-myd-y6ull14x14.imx-sd-optee
myir-image-core-myd-y6ull14x14.sdcard.bz2                        u-boot-sd-optee-2019.04-r0.imx
myir-image-core-myd-y6ull14x14.tar.bz2                           uTee-6ullevk
myir-image-core-myd-y6ull14x14.testdata.json                     zImage
myir-image-core-myd-y6ull14x14.wic.bmap                          zImage--5.4-r0-myd-y6ull14x14-20200920045735.bin
myir-image-core-myd-y6ull14x14.wic.bz2                           zImage-myd-y6ull14x14.bin
```

图 3-5. 构建完成目录预览

生成的文件中，有一些是链接文件，下面是不同文件的用途说明：

表 3-2. 生成文件列表

文件名	用途
*.rootfs.manifest	文件系统内的软件列表
*rootfs.tar.bz2	文件系统压缩包
*rootfs.wic.bz2	解压后直接写入 sd 卡，可从 sd 卡启动的镜像
u-boot-myd-y6ull14x14.imx	Uboot

表 3-3. 构建方法：

系统名	命令	描述
myir-image-core	bitbake myir-image-core	通用文件系统
myir-image-full	Bitbake myir-image-full	带 QT5.13 的文件系统

表 3-4. Bitbake 补充命令：

Bitbake 参数	描述
-k	有错误发生时也继续构建
-c cleanall	清空整个构建目录
-c fetch	从 recipe 中定义的地址，拉取软件到本地

-c deploy	部署镜像或软件包到目标 rootfs 内
-c compile	重新编译镜像或软件包

3.4. 构建 SDK（可选）

Yocto 提供可构建出 SDK 工具的功能，用于底层或上层应用开发者使用的工具链和相关的头文件或库文件，免去用户手动制作或编译依赖库。用于编译 u-boot 和 linux 内核代码，附带目标系统的头文件和库文件，方便应用开发者移植应用在目标设备上。SDK 工具都是 shell 自解压文件，执行后，默认安装在/opt 目录下。米尔已经提供了编译好的 SDK 安装包，如果用户有需要增加其它组件，可以参考下面的讲解构建自己的工具链 SDK。

- **构建工具链**

生成路径 build_imx6ul/tmp/deploy/sdk/

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

```
ls tmp/deploy/sdk/ -l

fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.host.manifest
fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.sh
fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.target.manifest
fsl-imx-fb-glibc-x86_64-meta-toolchain-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.testdata.json
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.host.manifest
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.sh
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.target.manifest
fsl-imx-fb-glibc-x86_64-myir-image-core-cortexa7t2hf-neon-myd-y6ull14x14-toolchain-5.4-zeus.testdata.json
```

图 3-6. 构建 SDK

如上图构建除了三个编译工具链的安装文件，我们开发资源包种提供的也是这三个。

构建其他工具链命令：

```
myir$ bitbake -c populate_sdk myir-image-full
myir$ bitbake -c populate_sdk myir-image-core
```

4. 如何烧录系统镜像

i.MX6UL 系列产品的启动方式多样，所以需要不同的更新系统工具与方法。用户可以根据需求选择不同的方式进行更新。由于烧写时需要调整启动方式，用户根据下表选择配置拨码开关。

表 4-1. 拨码启动方式

启动模式	拨码状态(B1/B2/B3/B4)	备注
eMMC 启动	OFF/OFF/ON/OFF	
NAND Flash 启动	OFF/ON/ON/OFF	
TF Card 启动(eMMC 版本)	ON/ON/ON/OFF	
TF Card 启动(NAND 版本)	ON/OFF/ON/OFF	
USB Download	X/X/OFF/ON	USB_OTG1 下载系统镜像

4.1. USB 方式更新

USB 更新方法的烧写工具是由 NXP 公司提供的 UUU 实现，资源包的目录“03_Tools/MYD-i.MX6ULX_UUU_xx.xx.xx”。

1). 准备工作

此步骤均在 Windows10 系统下制作（UUU 工具无法兼容 win7，请使用 win10 系统）。

工具：

- MYD-Y6ULX 开发板
- USB 转接线(Type-A 转 Micro-B)
- 电源适配器 12V2A

2). 更新步骤：

下面以 MYD-Y6UL-Y2 4E512D 配置的开发板为例，烧写 myir-image-full 系统镜像，其他配置的方法相同，替换不同的脚本即可。更新步骤如下：

a. 切换启动拨码开关(SW1)的第三位位 OFF,第四位为 ON

b. 使用 USB 转接线(Type-A 转 Micro-B)连接 PC 机 USB 端口与开发板 Micro USB OTG 端口(J26)

c. 使用 DC 12V 电源适配器连接至开发板的电源座(J22)

d. 以管理员权限打开 cmd 窗口，进入 MYD-i.MX6ULX_UUU_1.0.0_Down 目录，输入： uuu.exe myd-y6ulx-y2-4e512d-qt5.13.auto 开始烧写系统，如下图：

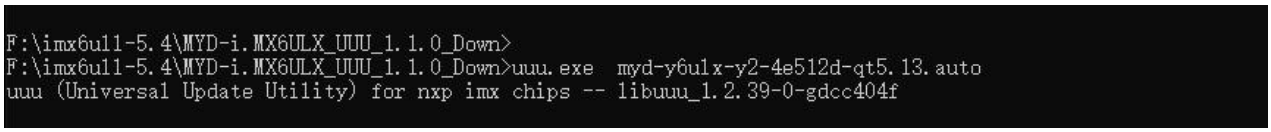

图 4-1. USB 方式烧写

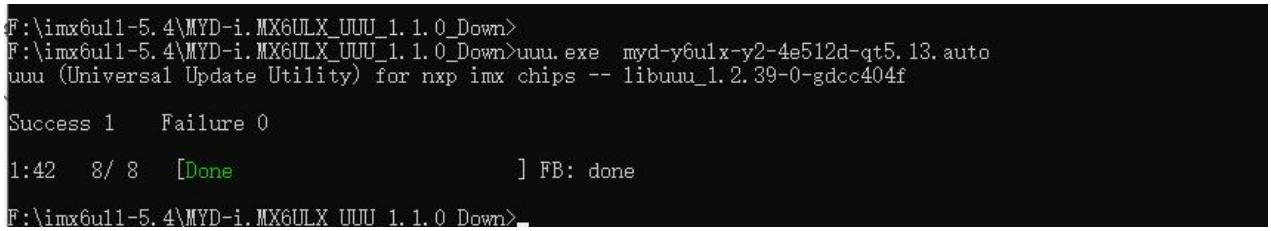

图 4-2. USB 烧写完成

e. 烧写完成显示绿色 Done，Success 显示 1，如上图。

f.烧写完成后，断电，设置拨码开关为 NAND 或者 eMMC 的启动方式，再上电即可从板子的 flash 启动。

MYD-Y6ULX 支持两种 Flash 存储方式 NAND 和 eMMC。使用 UUU 烧写时选择不同的脚本烧写。

表 4-2. MYD-Y6ULX 烧写脚本列表

文件名	描述
myd-y6ulx-y2-4e512d-qt5.13.auto	烧写 full 文件系统至 MYD-Y6ULX-Y2-4D512D 配置的板子
myd-y6ulx-y2-4e512d-core-base.auto	烧写 core 文件系统至 MYD-Y6ULX-Y2-4D512D 配置的板子
myd-y6ulx-y2-256n256d-qt5.13.auto	烧写 full 文件系统至 MYD-Y6ULX-Y2-256NADN 256DDR 配置的板子
myd-y6ulx-y2-256n256d-core-base.auto	烧写 core 文件系统至 MYD-Y6ULX-Y2-256NADN 256DDR 配置的板子
myd-y6ulx-g2-4e512d-qt5.13.auto	烧写 full 文件系统至 MYD-Y6ULX-G2-4D512D 配置的板子

myd-y6ulx-g2-4e512d-core-base.auto	烧写 core 文件系统至 MYD-Y6ULX-G2-4D512D 配置的板子
myd-y6ulx-g2-256n256d-qt5.13.auto	烧写 full 文件系统至 MYD-Y6ULX-G2-256NADN 256DDR 配置的板子
myd-y6ulx-g2-256n256d-core-base.auto	烧写 core 文件系统至 MYD-Y6ULX-G2-256NADN 256DDR 配置的板子

如果烧写自己编译的固件到 flash，只许替换相应的文件即可，如下面的文件结构，替换 file 目录中的固件，这些是会烧写到板子 flash 的固件。.

```
├── bootfile
│   ├── fsl-image-mfgtool-initramfs-myd-y6ull14x14.rootfs.cpio.gz.u-boot
│   └── zImage
├── EULA.txt
├── file
│   ├── dtb
│   ├── kernel
│   ├── rootfs
│   └── uboot
├── GPLv2
├── myd-y6ulx-g2-256n256d-core-base.auto
├── myd-y6ulx-g2-256n256d-qt5.13.auto
├── myd-y6ulx-y2-256n256d-core-base.auto
├── myd-y6ulx-y2-256n256d-qt5.13.auto
├── myd-y6ulx-y2-4e512d-core-base.auto
├── myd-y6ulx-y2-4e512d-qt5.13.auto
├── other
│   └── example_kernel_emmc.uuu
├── readme-myir.md
├── README.uuu
└── uuu.exe
```

4.2. SD 卡方式更新

为满足生产烧录的需要，米尔开发了适用于大批量生产的烧录方法。具体制作过程请按照下列步骤完成。

MYD-Y6ULX 开发板提供了一个制作 SD 卡更新系统镜像的工具，结构如下图：

图 4-3. SD 卡升级镜像制作工具

build-sdcard-5.4.sh 脚本是用于制作从 SD 卡更新系统的镜像，firmware 文件夹下的固件只是用于 SD 卡启动，一般情况不需要修改，mfgimages-myd-y6ulg2、mfgimages-myd-y6ulg2 和 roofs 文件夹中存放的固件是最终会烧写到板子的 flash 中；mfgimages-myd-* 文件夹中 Manifest 文本中指定了烧写的文件名，如下结构中红色字体，

```
alex@myir:~/imx6ul-5.4/MYiR-iMX-mkupdate-sdcard-5.4$ cat mfgimages-myd-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"
```

如果修改了 kernel 或 uboot，将其替换到 mfgimages-myd-y6uy2 文件夹中，注意文件名和 Manifest 中定义的保持一致，或者直接替换，保持原来的文件名。

build-sdcard.sh 提供了四种参数：

- -p' 表示平台，可用参数为"myd-y6uly2"代表 MYD-Y6ULL (MYC-Y6ULY2)
- '-n' 表示板上存储芯片是 NAND
- '-e' 表示板上存储芯片是 eMMC
- '-d' 表示更新文件的目录
- '-s' 表示 ddr 内存的大小
- '-f' 表示 roofs 的类型，支持 qt 和 core 参数

1). 制作 SD 卡更新的镜像：

```
$ cd MYiR-iMX-mkupdate-sdcard-5.4
$  sudo ./build-sdcard-5.4.sh -p myd-y6uly2 -n -d mfgimages-myd-y6uly2 -s 256
   -f  qt
$ sudo ./build-sdcard-5.4.sh -p myd-y6uly2 -n -d mfgimages-myd-y6uly2 -s 256
   -f  core
```

```
alex@myir:~/imx6ul-5.4/MYiR-iMX-mkupdate-sdcard-5.4$ ls -l
total 157536
-rwxr-xr-x 1 alex alex     11580 Sep 24 00:00 build-sdcard-5.4.sh
drwxr-xr-x 3 alex alex      4096 Sep 23 21:34 firmware
drwxr-xr-x 2 alex alex      4096 Sep 23 23:54 mfgimages-myd-y6ulg2
drwxr-xr-x 2 alex alex      4096 Sep 23 23:42 mfgimages-myd-y6uly2
-rw-r--r-- 1 root root 161283575 Sep 24 00:00 myd-y6uly2-update-emmc-20200924000928.rootfs.sdcard.img.gz
-rw-r--r-- 1 alex alex       331 Sep 23 05:17 README.MD
drwxr-xr-x 2 alex alex      4096 Sep 23 05:14 rootfs
alex@myir:~/imx6ul-5.4/MYiR-iMX-mkupdate-sdcard-5.4$
```

图 4-4. 制作完成 SD 卡升级镜像

如上图 myd-y6uly2-update-nand-qt-20201214211252.rootfs.sdcard.img.gz 即生成的镜像文件压缩包。

2). 制做可更新系统的 SD 卡

上面制作了 SD 卡更新的镜像，接下来将其写入 SD 卡中。

- **Windows 系统:**

Windows 用户可以使用 Win32DiskImager 工具把 xxxx.rootfs.sdcard.img 镜像写入 Micro SD 里。工具在"03_Tools"目录下，解压后，双击"Win32DskImager.exe"应用程序。启动后的界面中，右侧的"Device"是选择要写入的设备盘符。左侧的"映像文件"是选择将要写的镜像文件，点击旁边的文件夹图标，选中要写入的文件即可(注意：文件选择对话框中默认是过滤".img"文件，可以切换成".*")。

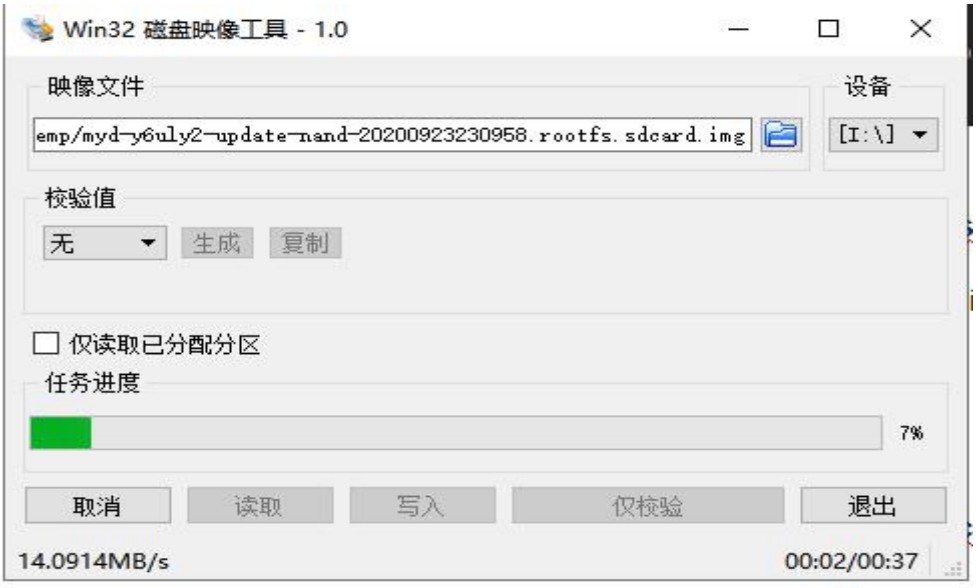

图 4-5. win32disk 烧写镜像到 SD 卡

- **Linux 系统:**

通常 Linux 下的存储设备名为"sd[x][n]"形式，x 表示第几个存储设备，一般使用字母 a ~z 表示。n 表示存储设备的分区，一般使用数字，从 1 开始。插入后可以使用"dmesg | tail"命令查看新设备的设备名称。注意这里以"/dev/sdb"设备为例，sdb 后面不写任务分区数字。

```
$ gzip -dc myd-y6uly2-update-nand-qt-20201214211252.rootfs.sdcard.img.gz | sudo dd of=/dev/sdb conv=fsync
```

把制作好的 SD 卡插入开发板的卡槽（J8），配置开发板的拨码开关（SW1）为 SDCARD 启动方式，连接 USB 转 TTL 串口线至调试串口(JP1)，配置好电脑端的串口终端软件。使用 DC 12V 电源适配器连接至开发板的电源接口(J22)。通过串口可以看到系统从 Micro SD 卡启动，并执行更新脚本，把 Linux 系统镜像文件写入 NAND 存储芯片内。也可以通过用户 LED 灯(D30)来判断当前更新状态，更新中为闪烁状态，更新成功后会常亮，失败则会熄灭。更新完成后断电，配置启动位拨码开关为板载的 NAND 或 eMMC 启动方式。

5. 如何修改板级支持包

前面的章节已经比较完整的讲述了基于 Yocto 项目构建运行在 MYD-Y6ULX 开发板上的系统镜像，并且将镜像烧录到开发板上的完整流程。由于 MYC-Y6ULX 核心板的很多管脚都具有多种功能复用的特性，所以实际项目中基于 MYC-Y6ULX 核心板设计的底板与 MYB-Y6ULX 相比总会有一些差异。这些差异可能是去掉显示，增加更多 GPIO, 或者需要增加更多串口，还有可能通过 SPI，I2C，USB 等扩展一些外设等等；除了硬件上的差异，还有一些系统组件上的差异，比如侧重 HMI 应用的，可能需要比较完备的图形系统，QT 库等，侧重后台管理应用的，可能需要更完备的网络应用，Python 运行环境等。这就需要系统开发人员在我们提供的代码基础上做一些裁剪和移植的工作。本章从一个系统开发人员的角度来讲述开发和定制自己系统的具体过程，为后面适配自己的硬件打下基础。

5.1. meta-myir 层介绍

Yocto 项目的“层模型”是一种用于嵌入式和物联网 Linux 创建的开发模型，它将 Yocto 项目与其他简单的构建系统区别开来。层模型同时支持协作和定制。层是包含相关指令集的存储库，这些指令集告诉 OpenEmbedded 构建系统应该做什么。

meta-myir 层基于在 NXP 官方的 meta-imx 层建立的适用于 MYD-Y6ULX 开发板的层，其中包含 BSP、GUI、发行版配置、中间件或应用程序的各种元数据和配方。用户可以在这个“层模型”的基础上适配基于 MYC-Y6ULX 核心板设计的硬件，定制自己的应用，从而构建适合自己的系统镜像，meta-myir 层包含的具体内容如下：

```
meta-myir/
├── EULA.txt
├── meta-bsp
│   ├── classes
│   ├── conf
│   ├── recipes-bsp
│   ├── recipes-connectivity
│   ├── recipes-core
│   ├── recipes-devtools
│   ├── recipes-graphics
│   ├── recipes-kernel
│   ├── recipes-multimedia
│   ├── recipes-security
```

```
|   ├── recipes-support
|   └── recipes-utils
├── meta-ml
|   ├── conf
|   ├── recipes-devtools
|   └── recipes-libraries
├── meta-sdk
|   ├── conf
|   ├── dynamic-layers
|   ├── recipes-benchmark
|   ├── recipes-connectivity
|   ├── recipes-devtools
|   ├── recipes-extended
|   ├── recipes-fsl
|   ├── recipes-graphics
|   ├── recipes-multimedia
|   ├── recipes-myir
|   ├── recipes-sato
|   └── recipes-support
├── README
└── tools
    ├── myir-setup-release.sh
    ├── readme-bluez.txt
    └── setup-utils.sh

31 directories, 5 files
```

表 5-1. meta-myir-层内容说明

源代码与数据	描述
conf	包括开发板软件配置资源信息
meta-bsp/recipes-bsp	包含 uboot 等配置资源
meta-bsp/recipes-kernel	包含 linux 内核的资源与第三方固件资源
meta-sdk/recipes-myir	包含文件系统的配置信息及应用程序,如 hmi v2.0

在进行系统移植时，需要重点关注的是负责硬件初始化和系统引导的 recipes-bsp 部分，负责 Linux 系统的内核和驱动实现的 recipes-kernel 部分以及应用程序定制的 recipes-myir 部分。

5.2. 板级支持包介绍

板级支持包(BSP)是定义如何支持特定硬件设备、设备集或硬件平台的信息集合。BSP 包括有关设备上的硬件特性的信息和内核配置信息，以及所需的任何其他硬件驱动程序。

在某些情况下，BSP 包含一个或多个组件的单独许可的知识产权(IP)。对于这些情况，必须接受需要某种明确的最终用户许可协议(EULA)的商业或其他类型许可证的条款。一旦您接受了许可证，米尔的开发板使用的 BSP 以遵守开源协议许可，同时 BSP 的源码码将全部开源。

表 5-2. BSP 开源协议

IP 项目	开源协议	描述
u-boot	GPLv2+	uboot
Linux	GPLv2	Linux kernel

通常根据硬件启动的不同阶段，我们将 BSP 分成 Bootloader 部分和 Kernel 部分，采用 MYC-Y6ULX 核心板设计的硬件 BSP 代码可以查看 meta-myir 中的 recipes-bsp 和 recipes-kernel 这两个配方的内容。

recipes-bsp 中主要包含了 Bootloader 部分的 u-boot 及其他的一些工具，这一部分主要实现核心硬件，如 DDR，Clock 的初始化及内核的引导。基于 MYC-Y6ULX 核心板硬件修改这部分的内容。

```
meta-myir/meta-bsp/recipes-bsp/
├── alsa-state
├── firmware-brcm43362
├── firmware-imx
├── firmware-qca
├── imx-atf
├── imx-kobs
├── imx-mkimage
├── imx-sc-firmware
├── imx-seco
├── imx-test
├── imx-uuc
```

```
├── imx-vpu
├── imx-vpu-hantro
├── imx-vpu-hantro-vc
└── u-boot
```

recipes-kernel 中包含 Linux 内核和 Linux Firmware 两个部分，主要实现内核及外设固件内容。

```
meta-myir/meta-bsp/recipes-kernel/
├── cryptodev
├── kernel-modules
├── linux
├── linux-firmware
└── linux-libc-headers
```

在使用米尔的核心板设计产品时，如无特殊的需求，bootloader 部分可不必修改。你需要更多的关注产品内核驱动的开发与调试以及应用软件的设计。后续章节将详细描述内核开发与应用开发。

5.3. 板载 u-boot 编译

U-Boot 是一个功能非常丰富的开源启动引导程序，包括内核引导，下载更新等众多方面，在嵌入式领域应用十分广泛，可查看官网获取更多信息 http://www.denx.de/wiki/U-Boot/WebHome。

前面 3.3 章节 Yocto 构建系统时也会一起的编译 uboot，此章节介绍单独编译 uboot 的方法，方便用于调试时使用。

1). 获取 U-Boot 源代码

直接使用工作目录下 myir-imx-uboot.tar.gz 或者使用如下命令下载最新的源码：

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

2). 单独编译 U-Boot

加载工具链环境变量：

```
myir$ source  /home/alex/workspace/meta_toolschain_imx6ul/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

编译：

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

<config>是配置选项名称，不同的启动模式需使用下表种不同的配置选项，MYD-Y6ULX 开发板有多种选项。

表 5-3. uboot defconfig 配置选项

编译选项	描述
myd_imx6ull_nand_ddr256_defconfig	MYD-Y6ULX-Y2 256N256
myd_imx6ull_emmc_defconfig	MYD-Y6ULX-Y2 4E512D
myd_imx6ul_nand_ddr256_defconfig	MYD-Y6ULX-G2 256N256
myd_imx6ul_emmc_defconfig	MYD-Y6ULX-G2 4E512D

编译完成后会在当前目录生成 u-boot-dtb.imx，即我们我们编译生成的 uboot 目标文件。

3). 使用 Yocto 单独编译 uboot

Yocto 中 指定 uboot 源码位置在：

```
sources/meta-myir/meta-bsp/recipes-bsp/u-boot/u-boot-common.inc
```

相关代码如下：

```
UBOOT_SRC ?= "git://github.com/MYiR-Dev/myir-imx-uboot.git;protocol=https"
SRCBRANCH = "develop"
SRC_URI = "${UBOOT_SRC};branch=${SRCBRANCH} \
"
SRCREV = "${AUTOREV}"
```

- UBOOT_SRC: uboot 代码下载位置
- SRCBRANCH: 分支名称
- SRCREV: commit 对应值， 设置为${AUTOREV} 则自动使用最新的 commit

如果修改了 uboot 的源码，需要提交修改记录，生成新的 commit 并写入到 u-boot-common.inc 配置文件，对应更新 SRCREV 的值，然后再编译则使用的你修改后的代码生成固件。

- **加载环境变量**

Yocto 中途退出或中断后，可以重新打开新的 shell 终端，重新加载 build 构建目录，命令如下

```
myir$: source myir-setup-release.sh -b build_imx6ul
```

- **编译**

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

使用 Ycoto 编译生成的 uboot 在 yocto 的编译目录:

```
myir$ build_imx6ul/tmp/deploy/images/myd-y6ull14x14/
```

4). 设置 Yocto 使用本地 uboot 源码

u-boot-common.inc 中默认指定是 github 的源码地址，第一次编译时会从 github 上拉取然后编译，后续如果用户自己调试修改源码和建立自己源码库，存放在本地会更便捷。

参考修改如下：

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

UBOOT_SRC 中修改指定为前面章节建立的工作目录解压出的 uboot 的源码位置。修改后，后面使用 yocto 编译则使用本地的源码。

5.4. 板载 kernel 编译

1). 获取 kernel 源代码

直接使用工作目录下 myir-imx-linux.tar.gz 或者使用如下命令下载最新的源码：

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

2). 单独编译 kernel

加载工具链环境变量：

```
myir$ source  /home/alex/workspace/meta_toolschain_imx6ul/environment-setup
-cortexa7hf-neon-poky-linux-gnueabi
```

编译：

```
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
```

编译完成后在"arch/arm/boot"目录会生成内核镜像文件 zImage，在"arch/arm/boot/dts"目录会生成 DTB 文件

表 5-4. dtb 列表

DTB 文件	描述
myd-y6ull-emmc.dtb	MYD-Y6ULX-Y2 4E512D
myd-y6ull-gpmi-weim.dtb	MYD-Y6ULX-Y2 256N256
myd-y6ul-emmc.dtb	MYD-Y6ULX-G2 4E512D
myd-y6ul-gpmi-weim.dtb	MYD-Y6ULX-G2 256N256

3). 使用 Yocto 单独编译 kernel

Yocto 中 指定 kernel 源码位置在：

```
sources/meta-myir/meta-bsp/recipes-kernel/linux/linux-imx_5.4.bb
```

相关的代码如下：

```
KERNEL_BRANCH ?= "develop"
```

```
LOCALVERSION = "-2.0.0"
KERNEL_SRC ?= "git://github.com/MYiR-Dev/myir-imx-linux.git;protocol=https"
SRC_URI = "${KERNEL_SRC};branch=${KERNEL_BRANCH}"
SRCREV = "${AUTOREV}"
```

- KERNEL_SRC : kernel 代码下载位置
- KERNEL_BRANCH: 分支名称
- SRCREV: commi tid 对应值，设置为${AUTOREV} 则自动使用最新的 commit

如果修改了 kernel 的源码，需要提交修改记录，生成新的 commit 并写入到 linux-imx_5.4.bb 配置文件，更新 SRCREV 的值，然后再编译则使用的你修改后的代码生成固件。

- **加载环境变量**

Yocto 中途退出或中断后，可以重新打开新的 shell 终端，重新加载 build 构建目录，命令如下

```
myir$: source myir-setup-release.sh -b build_imx6ul
```

编译

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

使用 Ycoto 编译生成的 zImage 在 yocto 的编译目录:

```
build_imx6ul/tmp/deploy/images/myd-y6ull14x14/
```

4). 设置 Yocto 使用本地 kernel 源码

linux-imx_5.4.bb 中默认指定是 github 的源码地址，第一次编译时会从 github 上拉取然后编译，后续用户自己调试修改源码和建立自己源码库，存放在本地会便捷。

参考修改如下:

```
KERNEL_BRANCH ?= "develop"
LOCALVERSION = "-2.0.0"
#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;protocol=file"
```

```
SRC_URI = "${KERNEL_SRC};branch=${KERNEL_BRANCH}"
SRCREV = "${AUTOREV}"
```

KERNEL_SRC 中修改指定为前面章节建立的工作目录解压出的 kernel 的源码位置。修改后，后面使用 yocto 编译则使用本地的源码。

6. 如何适配您的硬件平台

为了适配用户新的硬件平台，首先需要了解米尔的 MYD-Y6ULX 开发板提供了哪些资源，具体的信息可以查看《MYD-Y6ULX SDK 发布说明》。除此之外用户还需要对 CPU 的芯片手册，以及 MYC-Y6ULX 核心板的产品手册，管脚定义有比较详细的了解，以便于根据实际的功能对这些管脚进行正确的配置和使用。

6.1. 如何创建您的设备树

6.1.1. 板载设备树

用户可以在 BSP 源码里创建自己的设备树，一般情况下不需要修改 Bootloader 部分中的 u-boot。用户只需要根据实际的硬件资源对 Linux 内核设备树进行适当的调整即可。在此将 MYD-Y6ULX 的 BSP 各个部分中的设备树列表罗列出来，方便用户开发参考，具体内容如下表所示：

表 6-1. MYD-Y6ULX 设备树列表

项目	设备树	说明
U-boot	imx6ull.dtsi	资源设备树
	myb-imx6ul-14x14-base.dts	MYC-Y6ULX-G2 的基础 dts
	myb-imx6ull-14x14-base.dts	MYC-Y6ULX-Y2 的基础 dts
	myb-imx6ul-14x14-gpmi-weim.dts	MYC-Y6ULX-G2 的 nand 板基础 dts
	myb-imx6ul-14x14-emmc.dts	MYC-Y6ULX-G2 的 emmc 板基础 dts
	myb-imx6ull-14x14-gpmi-weim.dts	MYC-Y6ULX-Y2 的 nand 板基础 dts
	myb-imx6ull-14x14-emmc.dts	MYC-Y6ULX-Y2 的 emmc 板基础 dts
kernel	myd_y6ulx_defconfig	内核配置文件
	myb-imx6ul-14x14.dtsi	MYC-Y6ULX 公共部分的 dtsi
	myb-imx6ul-14x14-base.dts	MYC-Y6ULX-G2 的基础 dts
	myb-imx6ull-14x14-base.dts	MYC-Y6ULX-Y2 的基础 dts
	myd-y6ul-emmc.dts	MYC-Y6ULX-G2 的 emmc 基础 dts
	myd-y6ul-gpmi-weim.dts	MYC-Y6ULX-G2 的 nand 基础 dts
	myd-y6ull-emmc.dts	MYC-Y6ULX-Y2 的 emmc 板基础 dts
	myd-y6ull-gpmi-weim.dts	MYC-Y6ULX-Y2 的 nand 板 dts

6.1.2. 设备树的添加

Linux 内核设备树是一种数据结构，它通过特有的语法格式描述片上片外的设备信息。由 BootLoader 传递给 kernel，kernel 进行解析后形成和驱动程序关联的 dev 结构供驱动代码使用。

在内核源码下 arch/arm/boot/dts 下可以看到大量的平台设备树。如适合 MYD-Y6ULX 的设备树，可在当前路径下增加自定义设备树,如：myb-imx6ul--xxx.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
```

我们将 MYC-Y6ULX 核心板相关的资源编写进 myb-imx6ul-14x14.dtsi,其它扩展的接口和设备可以对它们进行引用，如下所示（仅供参考）：

```
// 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;
        };
    };
.....
```

用户增加了新的设备树源文件之后，还需要在同目录下的 Makefile 里添加设备树编译信息，这样就可以在编译内核的时候生成对应的设备树二进制文件。

```
dtb-$(CONFIG_SOC_IMX6UL) += \
        ......
            myb-imx6ul-14x14-base.dtb \
            myd-y6ul-gpmi-weim.dtb \
            myd-y6ul-emmc.dtb \
        ......
```

增加完成后即可增加设备驱动的板载描述，通过 5.4 节的方法编译生成设备树 dtb 文件 myd-xxx.dtb。上述过程是新建设备树文件过程，但由于添加新设备树后还需要修改 U-boot 中的加载文件名称，修改 Yocto 的配置文件和元数据，所以建议用户直接在我们的设备树上进行修改。

6.2. 如何根据您的硬件配置 CPU 功能管脚

实现一个功能引脚的控制是一个较为复杂的系统开发过程之一，其中包含了引脚的配置，驱动的开发，应用的实现等等步骤，本节不具体分析每个部分的开发过程，而是以实例来讲解功能管脚的控制实现。

6.2.1. GPIO 管脚配置的方法

GPIO: General-purpose input/output，通用的输入输出口，在嵌入式设备中是一个十分重要的资源，可以通过它们输出高低电平或者通过它们读入引脚的状态-是高电平或是低电平。

i.MX6UL 封装大量的外设控制器，这些外设控制器与外部设备交户一般是通过控制 GPIO 来实现，而将 GPIO 被外设控制器使用我们称为复用（Alternate Function），给它们赋予了更多复杂的功能，如用户可以通过 GPIO 口和外部硬件进行数据交互(如 UART)，控制硬件工作(如 LED、蜂鸣器等),读取硬件的工作状态信号（如中断信号）等。所以 GPIO 口的使用非常广泛。

i.MX6UL 的 GPIO 管脚配置方法一般使用 i.MX6 Pins Tool 配置或使用 Datasheet 查表的方式来配置。

- **MX6 Pins Tool 配置 PIN**

  i.MX6 Pins Tool 引脚工具的 i.MX 应用处理器是处理器专家的继任者®软件的 i.MX 处理器。新的 Pins Tool 通过直观易用的用户界面使引脚配置更加容易和快捷，然后生成普通的 C 代码，然后可在任何 C 和 C ++应用程序中使用该 C 代码。引脚工具可配置引脚信号（从多路复用（复用）到引脚的电气特性），还可以创建设备树摘要包含（.dtsi）文件并以 CSV 格式报告。本节不重点讲解其使用方法，介绍重要参数，您可以通过官方网站获取详细的开发指导说明。

  i.MX6 Pins Tool 官方链接：

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

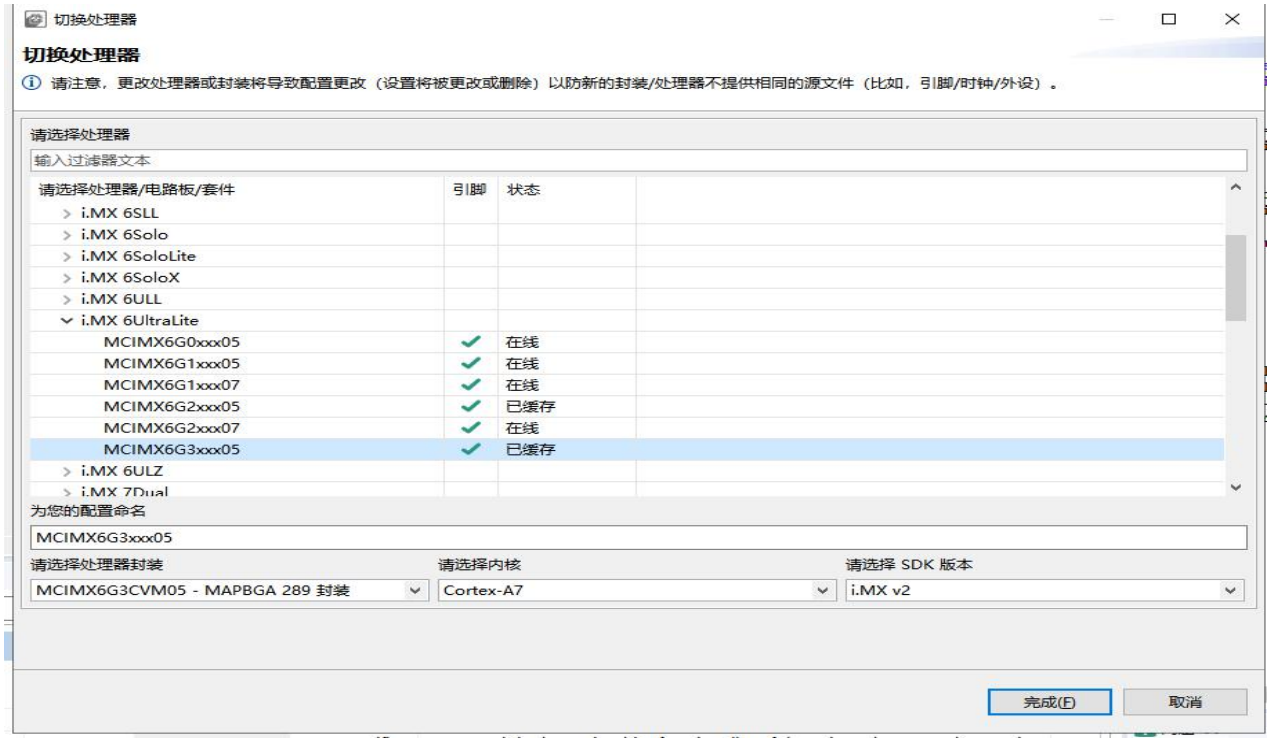

图 6-1. i.MX6 Pins Tool 选择处理器

如上图，打开 i.MX6 Pins Tool 工具后现在对应的处理器，我司标准品发布的是 MCIMX6G2xxx05 和 MCIMX6Y2xxx05。

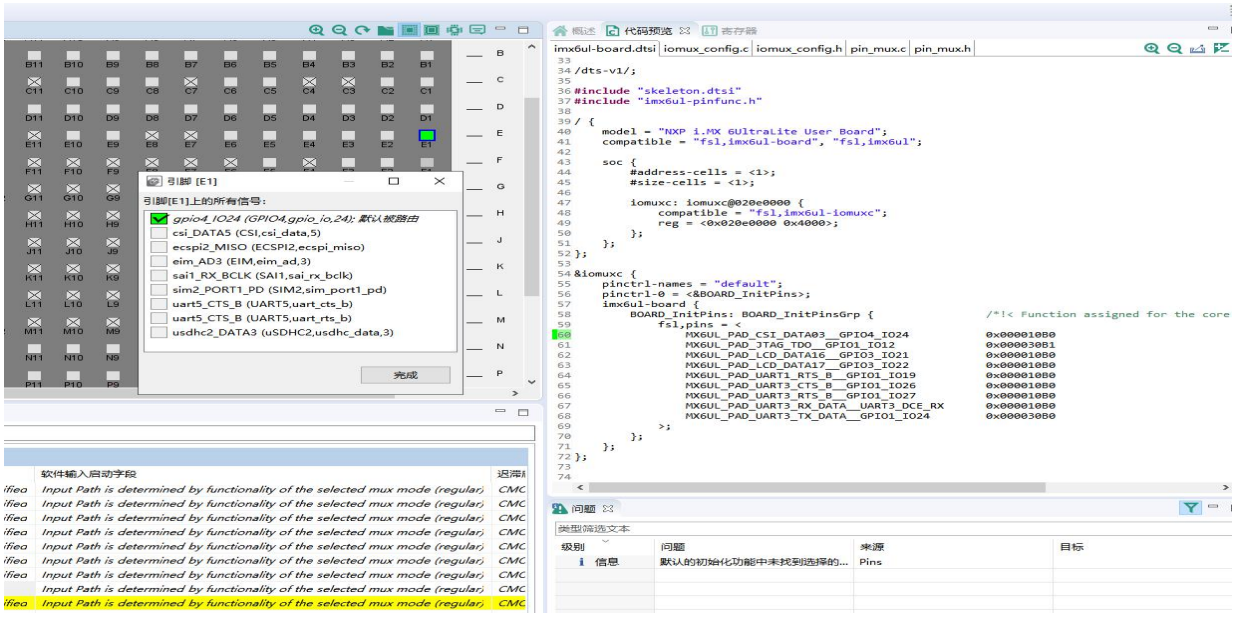

图 6-2. dts 参考代码

如上图选择一个 PIN,勾选一个功能则在右侧区域生成对应的 dts 代码

6.2.2. 设备树中引用 GPIO

此实例使用 J14 的 PIN5(MX6UL_PAD_UART3_TX_DATA__GPIO1_IO24)作为测试 GPIO。介绍如何在设备树里配置设备节点，并为后面章节供内核驱动使用。

```
// 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 th
e 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
            MX6UL_PAD_UART3_TX_DATA__GPIO1_IO24      0x000030B0
        >;
    };
  };
};
```

6.3. 如何使用自己配置的管脚

我们在 u-boot 或 Kernel 的设备树中配置后的管脚，可以在相应在 u-boot 或 Kernel 中进行使用，从而实现对管脚的控制。

6.3.1. U-boot 中使用 GPIO 管脚

1). 终端命令控制

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

2). 代码控制

```
//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_pad
s));
      gpio_request(IMX_GPIO_NR(3, 4), "power");
      gpio_direction_output(IMX_GPIO_NR(3, 4) , 1);

******
```

6.3.2. 内核驱动中使用 GPIO 管脚

1). 独立 IO 驱动的使用

在 6.2.3 节中的第一个设备树示例中，已经定义完成了 gpio 节点信息，下面将使用内核驱动来实现 GPIO 的控制（对 J14 的 PIN5 管脚进行置 1 与置 0，如需检测需使用万用表测试管脚电平的变化）。

```
//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. 确定主设备号  */
static int major = 0;
static struct class *gpioctr_class;
static struct gpio_desc *gpioctr_gpio;


/* 2. 实现对应的 open/read/write 等函数，填入 file_operations 结构体*/
static ssize_t gpio_drv_read (struct file *file, char __user *buf, size_t size, loff_t *offs
et)
{
        printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
        return 0;
}
```

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

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

        gpiod_set_value(gpioctr_gpio, status);

        return 1;
}

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

        return 0;
}

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

/* 定义自己的 file_operations 结构体*/
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,
};

/* 从 platform_device 获得 GPIO
 *    把 file_operations 结构体告诉内核：注册驱动程序
 */
static int chip_demo_gpio_probe(struct platform_device *pdev)
{
        /*  设备树中定义有: 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);
        }

        /*  注册 file_operations      */
        major = register_chrdev(0, "myir_gpioctr", &gpioctr_drv);  /* /dev/gpioctr */

        gpioctr_class = class_create(THIS_MODULE, "myir_gpioctr_class");
        if (IS_ERR(gpioctr_class)) {
            printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
            unregister_chrdev(major, "gpioctr");
            gpiod_put(gpioctr_gpio);
            return PTR_ERR(gpioctr_class);
        }

        device_create(gpioctr_class, NULL, MKDEV(major, 0), NULL, "myir_gpioctr%
d", 0);

    return 0;

}
```

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

    return 0;
}

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

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

/* 在入口函数注册 platform_driver */
static int __init gpio_init(void)
{
    int err;
    err = platform_driver_register(&chip_demo_gpio_driver);

        return err;
}
```

```
/* 有入口函数就应该有出口函数：卸载驱动程序时，就会去调用这个出口函数
 *     卸载 platform_driver
 */
static void __exit gpio_exit(void)
{
    platform_driver_unregister(&chip_demo_gpio_driver);
}

/* 其他完善：提供设备信息，自动创建设备节点 */
module_init(gpio_init);
module_exit(gpio_exit);

MODULE_LICENSE("GPL");
```

将驱动程序代码使用单独的 Makefile 编译成模块也可以直接配置进内核，

2). 将驱动示例直接配置进内核

在内核源代码的 drivers/char/文件夹下新建 gpioctr.c 文件，将上述驱动代码拷贝进去，并修改 Kconfig 与 Makefile 及 myd_y6ulx_defconfig。

修改 drivers/char/Kconfig 中添加如下代码：

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

修改 drivers/char/Makefile：

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

修改 myd_y6ulx_defconfig：

```
CONFIG_SAMPLE_GPIO=y
```

最后按照 5.4 节编译与更新内核即可。

3). 驱动示例编译成单独模块

在工作目录下增加 gpioctr.c 并拷贝上述驱动代码，同目录下编写独立 Makefile 程序。

```
# 修改 KERN_DIR
#KERN_DIR = # 板子所用内核源码的目录
KERN_DIR = /home/myir/myir-imx-linux/

obj-m += gpioctr.o

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

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

# 要想把 a.c, b.c 编译成 ab.ko, 可以这样指定:
# ab-y := a.o b.o
# obj-m += ab.o
```

加载 SDK 环境变量到当前 shell。

```
myir$ source  /home/alex/workspace/meta_toolschain_imx6ul/environment-setup
-cortexa7hf-neon-poky-linux-gnueabi
```

执行 make 命令，即可生成 gpioctr.ko 驱动模块文件。

```
myir$:/home/myir/demo_gpioctr$ 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_gpioctr/gpioctr.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M] /home/myir/demo_gpioctr/gpioctr.mod.o
  LD [M] /home/myir/demo_gpioctr/gpioctr.ko
make[1]: Leaving directory '/home/myir/myir-imx-linux'
```

编译成功之后，将 gpioctr.ko 文件可通过以太网、WIFI、U 盘等传输介质传输到开发板然后使用 insmod 命令加载驱动。

6.3.3. 用户空间使用 GPIO 管脚

Linux 操作系统的体系架构分为用户态和内核态（或者用户空间和内核）。用户态即上层应用程序的活动空间，应用程序的执行必须依托于内核提供的资源，包括 CPU 资源、存储资源、I/O 资源等。为了使上层应用能够访问到这些资源，内核必须为上层应用提供访问的接口：即系统调用。

Shell 是一个特殊的应用程序，俗称命令行，本质上是一个命令解释器，它下通系统调用，上通各种应用。使用 Shell 脚本，通常短短的几行 Shell 脚本就可以实现一个非常大的功能，原因就是这些 Shell 语句通常都对系统调用做了一层封装。为了方便用户和系统交互。

本节将讲述在用户态如何使用 GPIO 管脚的控制三种基本方式。

- Shell 命令
- 系统调用
- 库函数

1). Shell 实现管脚控制

Shell 控制管脚实质上是调用 Linux 提供的文件操作接口俩实现的，本节不做详细的说明，可查看《**MYD-Y6ULX_Linux 软件评估指南**》第 3.1 节描述。

2). 库函数实现管脚控制

Libgpiod 库函数实现由于 gpiochip 的方式，基于 C 语言，所以开发者实现了 Libgpiod，提供了一些工具和更简易的 C API 接口。Libgpiod (Library General Purpose Input/Output device) 提供了完整的 API 给开发者，同时还提供了一些用户空间下的应用来操作 GPIO。

Libgpiod 常用基本接口描述：

- gpiodetect - 列出系统中出现的所有 gpiochip，它们的名称，标签和 GPIO 行数。
- gpioinfo - 列出指定的 gpiochips 的所有行、它们的名称、使用者、方向、活动状态和附加标志。
- gpioget -读取指定的 GPIO 行值。

- gpioset - 设置指定的 GPIO 行值，潜在地保持这些行导出并等待超时、用户输入或信号。
- gpiofind - 查找给定行名称的 gpiochip 名称和行偏移量。
- gpiomon - 等待 GPIO 行上的事件，指定要观察哪些事件，退出前要处理多少事件，或者是否应该将事件报告到控制台。

更多描述，可查看 libgpiod 源代码 https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/。

下列将以 J14 的 PIN5 脚做为操作 GPIO 管脚来实现 C 语言的代码控制实例(交替置高置低)。

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

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

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

        /* Open device: gpiochip5 for GPIO bank F */
        fd = open(chrdev_name, 0);
        if (fd == -1) {
            ret = -errno;
```

```
        fprintf(stderr, "Failed to open %s\n", chrdev_name);

    return ret;
    }

    /* request GPIO line: GPIO_F_14 */
    req.lineoffsets[0] = 14;
    req.flags = GPIOHANDLE_REQUEST_OUTPUT;
    memcpy(req.default_values, &data, sizeof(req.default_values));
    strcpy(req.consumer_label, "gpio_f_14");
    req.lines = 1;

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

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

    /* release line */
    ret = close(req.fd);
```

```
        if (ret == -1) {
                perror("Failed to close GPIO LINEHANDLE device file");
                ret = -errno;
        }
        return ret;
}
```

将上述代码拷贝到一个 example-gpio.c 文件下，加载 SDK(myir-image-full 系统的应用工具链)环境变量到当前 shell:

```
myir$ source  /home/alex/workspace/fsl_toolschain_imx6ul/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

使用编译命令$CC 可生成可执行文件 example-gpio。

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

将可执行文件通过网络（scp 等），u 盘等传输介质拷贝到开发板的/usr/sbin 目录下，即可在终端下输入命令可直接运行。

```
root@myd-y6ull14x14:~# example-gpio
```

3). 系统调用实现管脚控制

操作系统提供给用户程序调用的一组“特殊”接口。用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务，比如用户可以通过文件系统相关的调用请求系统打开文件、关闭文件或读写文件，可以通过时钟相关的系统调用获得系统时间或设置定时器等。

同时管脚也是资源，也可以通过系统调用的方式实现控制。在 6.3.2 中我们已经完成了管脚的驱动的实现，即可对该驱动程序所控制的管脚进行系统调用控制。

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

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

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

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

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

将上述代码拷贝到一个 gpiotest.c 文件下，加载 SDK（myir-image-full 系统的应用工具链）环境变量到当前 shell:

```
myir$ source  /home/alex/workspace/fsl_toolschain_imx6ul/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

使用编译命令$CC 可生成可执行文件 gpiotest。

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

将可执行文件通过网络（scp 命令等），u 盘等传输介质拷贝到开发板的/usr/sbin 目录下，即可在终端下输入命令可直接运行(on 表示置高，off 表示置低)。

```
root@myd-y6ull14x14:~#  gpiotest /dev/myir_gpioctr on
root@myd-y6ull14x14:~#  gpiotest /dev/myir_gpioctr off
```

7. 如何添加您的应用

Linux 应用的移植通常分为两个阶段，开发调试阶段和生产部署阶段。开发调试阶段我们可以使用米尔构建好的 SDK 对我们编写好的应用进行交叉编译然后远程拷贝到目标主机进行测试。生产部署阶段需要为应用编写配方文件，并使用 BitBake 构建生产镜像。

7.1. 基于 Makefile 的应用

Makefile 其实就是一个文档，里面定义了一系列的编译规则，它记录了原始码如何编译的详细信息！ Makefile 一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。在开发 Linux 程序时，不管是内核，驱动，应用，Makefile 得到了普遍的应用。

make 是一个命令工具，是一个解释 makefile 中指令的命令工具。它可以简化编译过程里面所下达的指令，当执行 make 时，make 会在当前的目录下搜寻 Makefile (or makefile) 这个文本文件，执行对应的操作。make 会自动的判别原始码是否经过变动了，而自动更新执行档。

下列将以一个实际的示例（在 MYD-Y6ULX 开发板上实现按键控制 LED 灯开关）来讲述 Makfile 的编写与 make 的执行过程。Makefile 有其自身的一套规则。

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

- target 可以是一个 object file(目标文件)，也可以是一个执行文件，还可以是一个标签（label）。
- prerequisites 就是要生成那个 target 所需要的文件或是目标。
- command 也就是 make 需要执行的命令。

```
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：C 编译器的名称
- CXX: C++编译器的名称
- clean: 是一个约定的目标
- Key_led 实现代码如下：

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

/* ./key_led /dev/input/event0 noblock */
int main(int argc, char **argv)
{
        int fd,bg_fd;
        int err, len, i;
        unsigned char flag;
        unsigned int data[1];
        char *bg = "/sys/class/leds/heartbeat/brightness";

        struct input_event event;

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

        if (argc == 3 && !strcmp(argv[2], "noblock"))
```

```
        {
            fd = open(argv[1], O_RDWR | O_NONBLOCK);
        }
        else
        {
            fd = open(argv[1], O_RDWR);
        }
        if (fd < 0)
        {
            printf("open %s err\n", argv[1]);
            return -1;
        }

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

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

```
                                system("echo 0 > /sys/class/leds/heartbeat/brightness
");//led on
                        }


                }


          }
       return 0;
}
```

使用 make 命令进行编译并生成目标机器上的可执行文件 target_bin。

加载 SDK 环境变量到当前 shell:

```
myir$ source /opt/st/myir/3.1-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

执行 make:

```
myir$ make
```

从上一个命令的结果可以看到，使用的编译器是通过设置脚本中定义的 CC 变量建立的编译器。

将 target_bin 可执行文件通过网络（scp 等），u 盘等传输介质拷贝到开发板的/usr/sbin 目录下:

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

说明：如果使用交叉工具链编译器构建 target_bin，并且构建主机的体系结构与目标机器的体系结构不同，则需要在目标设备上运行项目。

7.2. 简单应用移植

应用的移植通常分为两个阶段，开发调试阶段和生产部署阶段。开发调试阶段我们可以使用 5.4 节构建好的 SDK 对我们编写好的应用进行交叉编译然后远程拷贝到目标主机进行测试。生产部署阶段需要为应用编写配方文件，并使用 BitBake 构建生产镜像。

这里以最简单的 helloworld 为例说明使用 SDK 进行应用开发调试的过程(参考 https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#makefile-based-projects）。

- **创建工作目录并填充代码:**

为应用创建一个工作目录并进入该目录。

```
$ mkdir $HOME/helloworld
$ cd $HOME/helloworld
```

目录创建之后，我们需要创建下面三个源文件：

```
// 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");
}
```

- **设置交叉编译环境:**

在前面章节中有描述过，安装 SDK 会创建一个设置交叉编译环境的脚本，执行这个脚本即可设置交叉编译工具链运行环境:

```
myir$ source /opt/st/myir/2.6-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

- **创建 Makefile:**

这个例子中，创建的 Makefile 文件内容如下:

```
# 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
```

- **编译 Helloworld 应用:**

使用 make 命令进行编译并生成目标机器上的可执行文件 target_bin:

```
myir$ make
```

从上一个命令的结果可以看到，使用的编译器是通过设置脚本中定义的 CC 变量建立的编译器。

- **执行 helloworld 应用:**

将 target_bin 拷贝到目标机器上，在控制台下执行，输出我们想要的“Hello World!”：

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

说明：如果使用交叉工具链编译器构建 target_bin，并且构建主机的体系结构与目标机器的体系结构不同，则需要在目标设备上运行项目。

7.3. 应用程序开机自启动

1). 应用程序在 Yocto 的配置

通常我们的应用还需要实现开机自启动，这些也可以在配方中实现。下面以一个稍微复杂一点的 FTP 服务应用为例说明如何使用 Yocto 构建包含特定应用的生产镜像，这里的 FTP 服务程序采用的是开源的 Proftpd，各个版本源码位于 ftp://ftp.proftpd.org/distrib/source/。

在我们从头开始写一个配方之前，我们可以在当前源码仓库中查找一下是否已经存在该应用，或者类似应用的配方，查找方法如下：

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

注意：执行 bitbake 命令之前，确保您已经执行了构建 Yocto 项目的环境变量设置脚本。

我们也可以在 OpenEmbedded 的官方网站层索引（http://layers.openembedded.org/layerindex/branch/master/layers/）中查找是否有同样或者类似应用的配方。

编写新配方的方法参见 Yocto 项目完全手册编写新的配方章节 https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#new-recipe-writing-a-new-recipe。

本节重点描述如何移植 FTP 服务到目标机器中的方法。通过搜索当前源代码仓库发现 Yocto 项目中已经存在 proftpd 的配方，只是没有添加的系统镜像中。下面详细描述具体的移植过程。

- **查找 Yocto 的 proftpd 配方**

```
PC $ ~/Yocto/build-openstlinuxeglfs-myir$ bitbake -s | grep proftpd
proftpd                                 :1.3.6-r0
```

注意：这里可以看到 Yocto 项目中已经存在 proftpd 配方，版本为 1.3.6-r0。

- **单独编译 proftpd**

```
PC $ bitbake proftpd
```

- **打包 proftpd 到文件系统**

在 conf/local.conf 中增加一行语句：

```
IMAGE_INSTALL_append = "proftpd"
```

- **重新构建镜像**

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

- **烧录新镜像**

系统构建完成之后，需重新烧录镜像并查看 proftpd 服务是否运行：

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

这里补充说明一下 FTP 的账户设置。FTP 客户端有三种类型登录账户，分别为匿名账户，普通账户和 root 账户。

- **匿名账户**

用户名为 ftp ，不需要设置密码，用户登录后可以查看系统/var/lib/ftp 目录下的内容，默认没有写权限。由于系统默认不存在/var/lib/ftp 目录，所以需要用户在目标机器上创建一个目录/var/lib/ftp。为了尽量不修改 meta-openembbed, 我们可以通过为 proftpd 配方添加 Append 文件“proftpd_1%.append”来实现/var/lib/ftp 目录的创建。

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

编辑好的 proftpd_1%.append”需要放置到 meta-myir 下面 recipes-daemons\proftpd 目录。然后重复上面添加应用的步骤，重新构建镜像文件进行测试。

- **普通账户**

在目标机器上使用 useradd 和 passwd 命令可以创建普通用户，并设置用户密码之后，客户端也可以使用该普通账户登录到该用户的 HOME 目录。如果需要在编译镜像时包含普通用户，可以参照 https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#ref-classes-useradd 添加普通用户，然后重新构建镜像文件，具体方法这里不再赘述。

- **root 账户**

如果需要开放 root 账户登录 FTP 服务器，需要先修改/etc/proftpd.conf 文件，在文件中增加一行配置 "RootLogin on"。与此同时，也需要为 root 账户设置密码，重启 proftpd 服务之后，客户端也可以使用 root 账户登录到目标机器上。

```
# systemctl restart proftpd
```

注意：修改/etc/proftpd.conf 使能 root 账户登录仅用于测试目的，关于/etc/proftpd.conf 的更多配置，参见 http://www.proftpd.org/docs/example-conf.html。

2). 实现应用程序的自启动

本节将以 proftpd 配方为例从配方源码的层面介绍如何添加应用程序配方并实现程序的开机自启动。proftpd 配方位于源代码仓库 layers/meta-openembedded/meta-networking/recipes-daemons/proftpd，目录结构如下。

```
├── 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 为构建 proftpd 服务的配方
- proftpd.service 为开机自启动服务
- proftpd-basic.init 为 proftpd 的启动脚本

proftpd_1.3.6.bb 配方中指定了获取 proftpd 服务程序的源代码路径以及针对该版本源码的一些补丁文件：

```
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 \
```

"

配方中还指定了 proftpd 的配置（do_configure)和安装过程（do_install）：

```
FTPUSER = "ftp"
FTPGROUP = "ftp"

do_install () {
    oe_runmake DESTDIR=${D} install
    rmdir ${D}${libdir}/proftpd ${D}${datadir}/locale
    [ -d ${D}${libexecdir} ] && rmdir ${D}${libexecdir}
    sed -i '/ *User[ \t]*/s/ftp/${FTPUSER}/' ${D}${sysconfdir}/proftpd.conf
    sed -i '/ *Group[ \t]*/s/ftp/${FTPGROUP}/' ${D}${sysconfdir}/proftpd.conf
    install -d ${D}${sysconfdir}/init.d
    install -m 0755 ${WORKDIR}/proftpd-basic.init ${D}${sysconfdir}/init.d/proftpd
    sed -i 's!/usr/sbin/!${sbindir}/!g' ${D}${sysconfdir}/init.d/proftpd
    sed -i 's!/etc/!${sysconfdir}/!g' ${D}${sysconfdir}/init.d/proftpd
    sed -i 's!/var/!${localstatedir}/!g' ${D}${sysconfdir}/init.d/proftpd
    sed -i 's!^PATH=.*!PATH=${base_sbindir}:${base_bindir}:${sbindir}:${bindir}!'
${D}${sysconfdir}/init.d/proftpd

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

    # create the pub directory
    mkdir -p ${D}/home/${FTPUSER}/pub/
    chown -R ${FTPUSER}:${FTPGROUP} ${D}/home/${FTPUSER}/pub
    if ${@bb.utils.contains('DISTRO_FEATURES', 'pam', 'true', 'false', d)}; then
        # install proftpd pam configuration
        install -d ${D}${sysconfdir}/pam.d
        install -m 644 ${S}/contrib/dist/rpm/ftp.pamd ${D}${sysconfdir}/pam.d/proft
pd
        sed -i '/ftpusers/d' ${D}${sysconfdir}/pam.d/proftpd
        # specify the user Authentication config
```

```
    sed -i '/^MaxInstances/a\AuthPAM                    on\nAuthPAMConfig
      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 finishs,
# 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
}
```

这两个函数对应 BitBake 构建过程的 config 和 install 任务(关于任务的更多信息，参见 https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#ref-tasks)。

proftpd_1.3.6.bb 配方通过继承 systemd.class（具体内容查看 layers/openembedded-core/meta/classes/systemd.bbclass）默认使能了 SYSTEMD_AUTO_ENABLE 变量并实现开机自启动，用户自己编写的配方也可以通过设置变量 SYSTEMD_AUTO_ENABLE 实现开机自启动，示例如下：

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

当前目标机器采用 systemd 作为初始化管理子系统，systemd 是一个 Linux 系统基础组件的集合，提供了一个系统和服务管理器，运行为 PID 1 并负责启动其它程序。Yocto 项目下使用 systemd 的配置参见 https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#selecting-an-initialization-manager。

Proftpd 服务的开机自启动服务文件 proftpd.service 内容如下：

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

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

[Install]
WantedBy=default.target
```

- After 表是此服务在 network 启动后再启动。
- Type 表示启动的方式为 forking。
- ExecStart 表示需要启动的程序，及对应的参数。

如需了解更多关于 systemd 的信息请查看此网站 https://wiki.archlinux.org/index.php/systemd。

用户在添加自己编写的应用时，也可以参照上面的示例创建配方，设置

开机自启动，并打包进系统镜像。自己编写的配方建议放置到 layers/meta-myir-st/recipes-app 目录。

7.4. QT 应用开发

Qt 是一个跨平台的图形应用开发框架，被应用在不同尺寸设备和平台上，同时提供不同版权版本供用户选择。MYD-Y6ULX 使用 Qt 5.13 版本进行应用开发。在 Qt 应用开发中，推荐使用 QtCreator 集成开发环境，可以在 Linux PC 下开发 Qt 应用，自动化地交叉编译为开发板的 ARM 架构。

本章使用 Yocto 构建的 SDK 工具作为交叉编译系统，配合 QtCreator 快速开发图形类应用程序。开始本章前，请先完成第三章的 Yocto 构建过程,或者使用光盘中提供的预编译好的 SDK 工具包。本章开始前，请安装好应用 SDK 开发工具。

7.4.1. 安装 QtCreator

QtCreator 安装包是一个二进制程序，直接执行就可以完成安装。

```
$ 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
```

执行安装程序后，一直点击下一步即可完成。默认安装目录在"/opt/ qtcreator-5.9.4 "。qtcreator-5.9.4 文件夹需要在/opt 目录下提前使用 sudo mkdir qtcreator-5.9.4 新建。

安装完成后，为了让 QtCreator 使用 Yocto 的 SDK 工具，需要对 QtCreator 加入环境变量。修改" /opt/qtcreator-5.9.4/Tools/QtCreator/bin/qtcreator.sh "文件，在"#! /bin/sh"前加入 Yocto 的环境配置即可，参考如下：

```
myir$ source /opt/myir-image-full-5.4/environment-setup-cortexa7t2hf-neon-poky-linux-gnueabi
#! /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.
```

使用 QtCreator 时请从终端执行"qtcreator.sh"来启动 QtCreator 参考如下：

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

7.4.2. 配置 QtCreator

想要利用qtcreator 编译出目标板可用的程序需要进行编译链的配置。需要重新设置以下配置项：

- 配置 GCC 和 G++ 编译链
- 配置 QTversion
- 配置 QTdebug
- 新增 device 设备
- 创建 kit，把上述 4 项添加到一起组成编译 QT 配置

1). 配置 GCC,G++

运行 QtCreator 后，依次点击"Tool"->"Options"，出现选项对话框，在左侧点击"Build & Run"，右边选择"Compilers"标签。 点击右侧"Add"按钮，弹出下拉列表后，选择"GCC"中的“C”，在下面填写"Name"为"MYDY6ULx-GCC"，"Compiler path"点击旁边的"Browse.."按钮选 择到/opt/myir-image-full-5.4/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux/arm-poky-linux-gcc 点击"Apply"。

再次点击右侧"Add"按钮，弹出下拉列表后，选择"GCC"中的“C++”，在下面填写"Name"为"MYDY6ULx-GCC++"，"Compiler path"点击旁边的"Browse.."按钮选 择到/opt/myir-image-full-5.4/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux/arm-poky-linux-g++点击"Apply"。

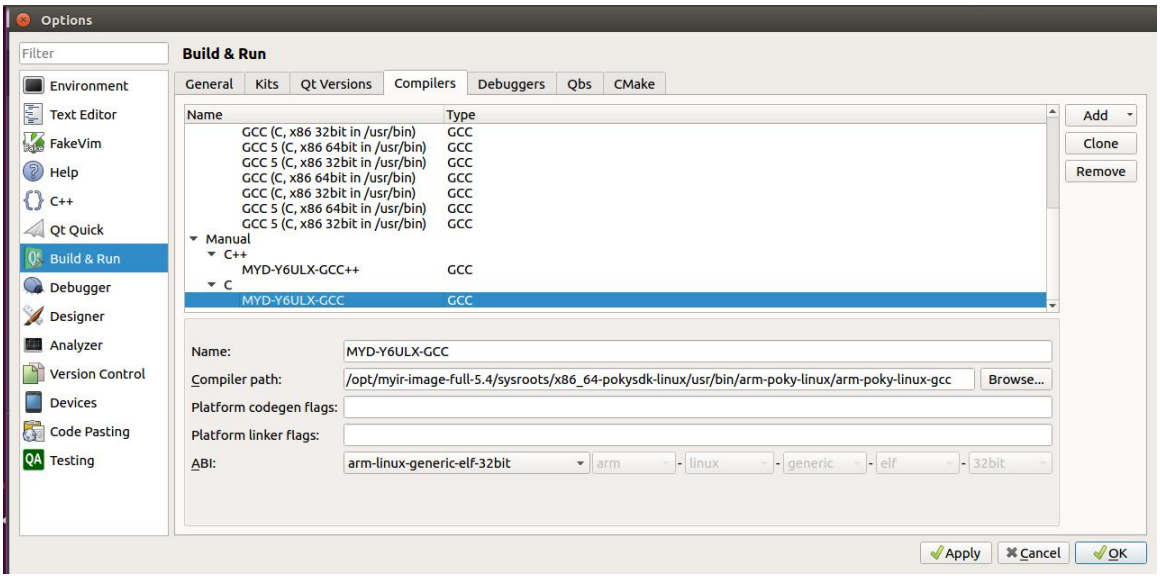

图 7-1. 配置编译器 GCC

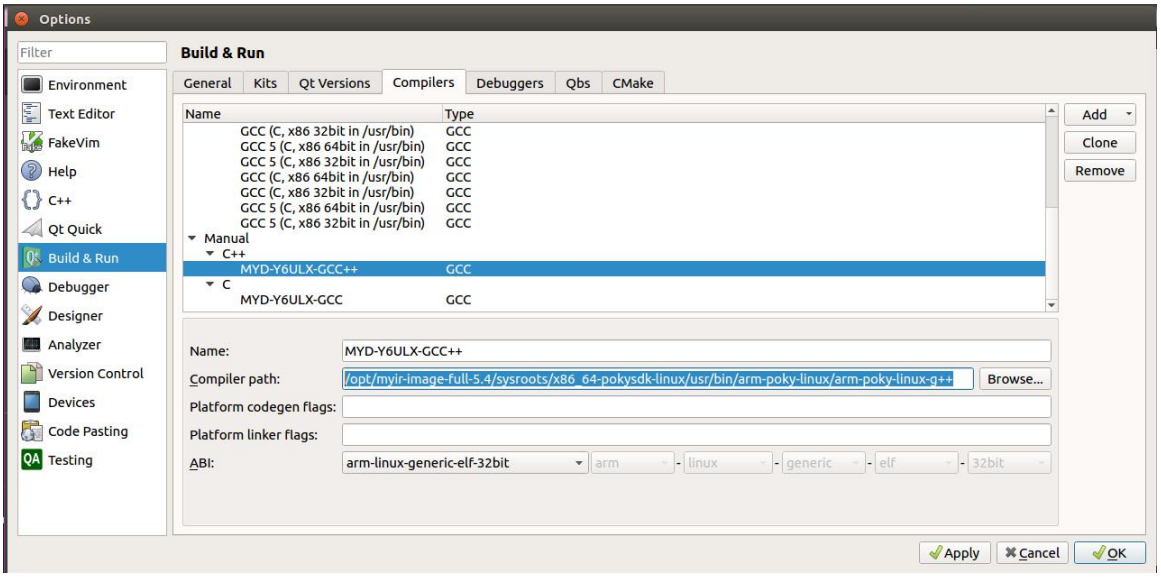

图 7-2. 配置编译器 GCC++

2). 配置 QTversion

选择"Qt Version"标签，在右侧点击"Add..."，会弹出 qmake 路径选择对话框，这里以"/opt/myir-image-full-5.4/sysroots/x86_64-pokysdklinux/usr/bin/qmake"为例子。选择" qmake" 文件后，点击"Open"按钮。"Version name"改为"Qt %{Qt:Version} (System)".然后点击"Apply"按钮。

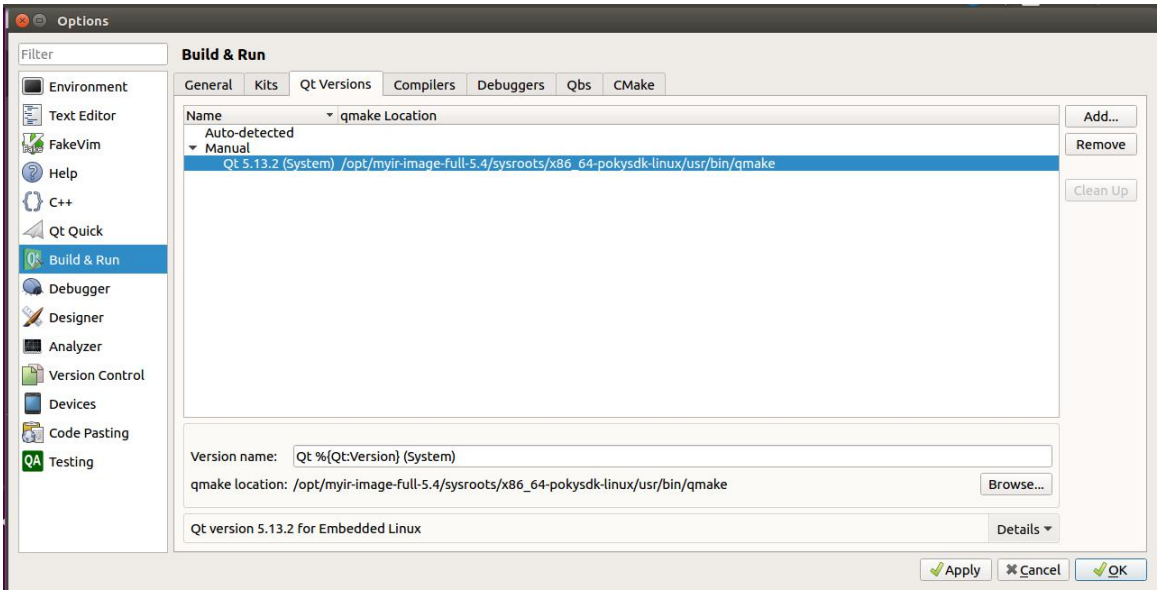

图 7-3. 配置 Qt 版本

3). 配置 QT Debug:

选择右侧侧"Debuggers"，点击右边的"Add..."按钮，在弹出的对话框中填写内容"Name"为"MYD-Y6ULX-DEBUG"。path 路径为：

/opt/myir-image-full-5.4/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux/arm-poky-linux-gdb

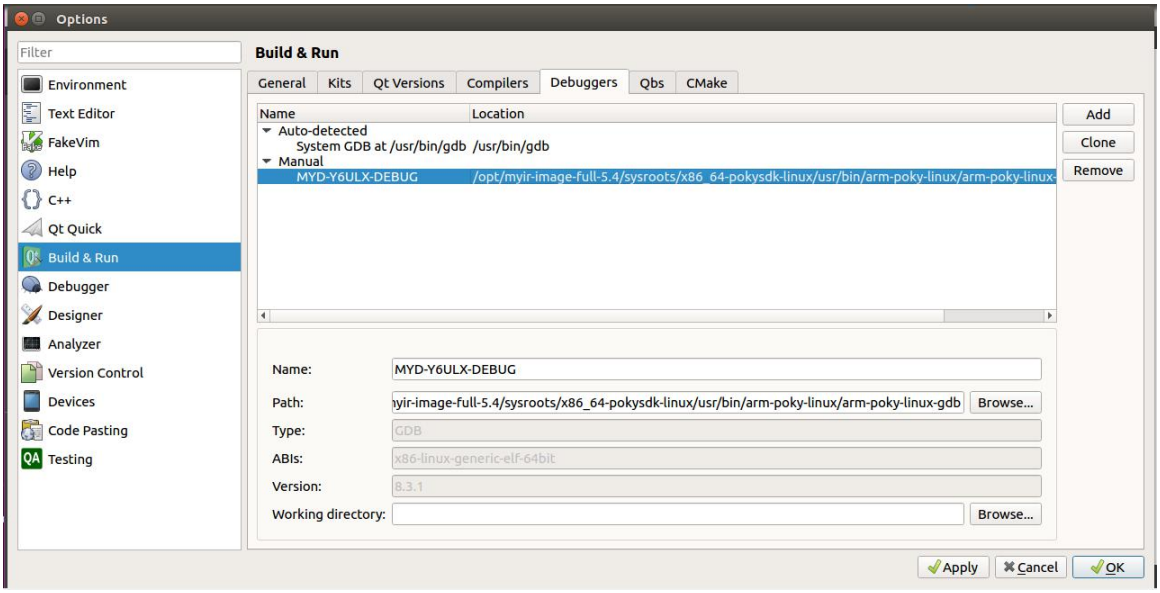

图 7-4. 配置 DEBUG

4). 新增 Device 设备

选择左侧"Device"，点击右边的"Add..."按钮，在弹出的对话框中选择 Generic Linux Device,再填写内容"Name"为"MYDY6ULx Board"，"Host name"为开发板的 IP 地址(可以暂时填写任意一个地址)，"Username"为"root"，然后点击"Apply"。

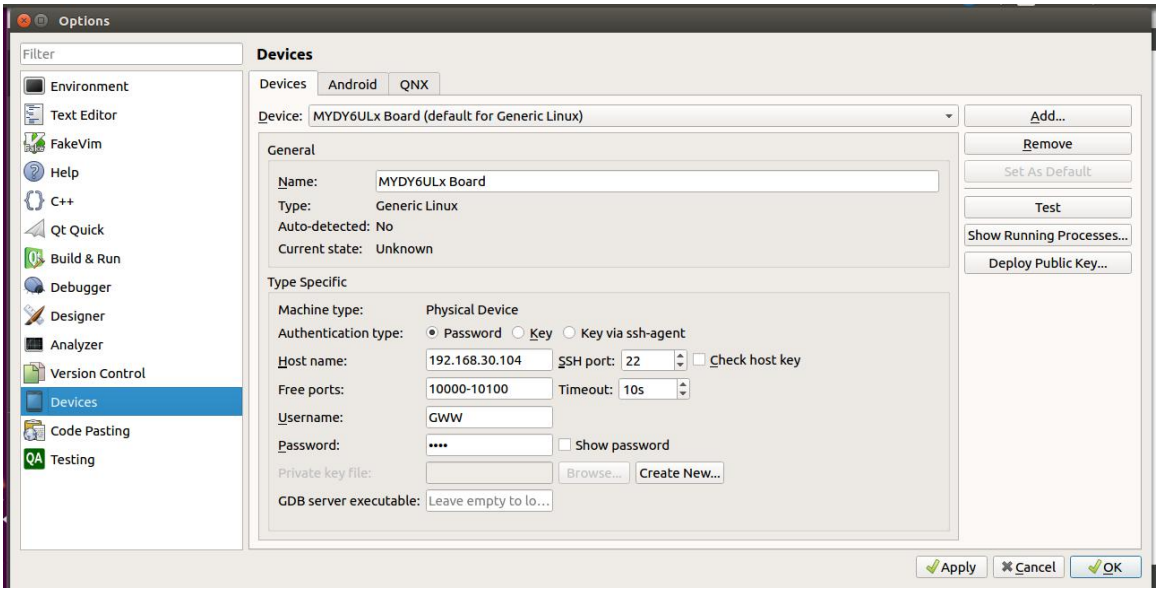

图 7-5. 配置 DEVICE

5). 创建一个 kit

点击左侧"Build & Run"回到"Kits"标签下，"Name"为"MYD-Y6ULX "，"Device"选择"MYDY6ULx Board"选项了。"Sysroot"选择目标设备的系统目录，这里以" /opt/myir-image-full-5.4/sysroots "为例。"Compiler"选择之前配置的名称"MYDY6ULx-GCC"和“MYDY6ULx-GCC++”，"Qt version"选择之前配置的名称"Qt 5.13.2 (System)"，"Qt mkspec"填写为"linux-oe-g++"。其它默认即可，最后点击"Apply"和"OK"按钮。

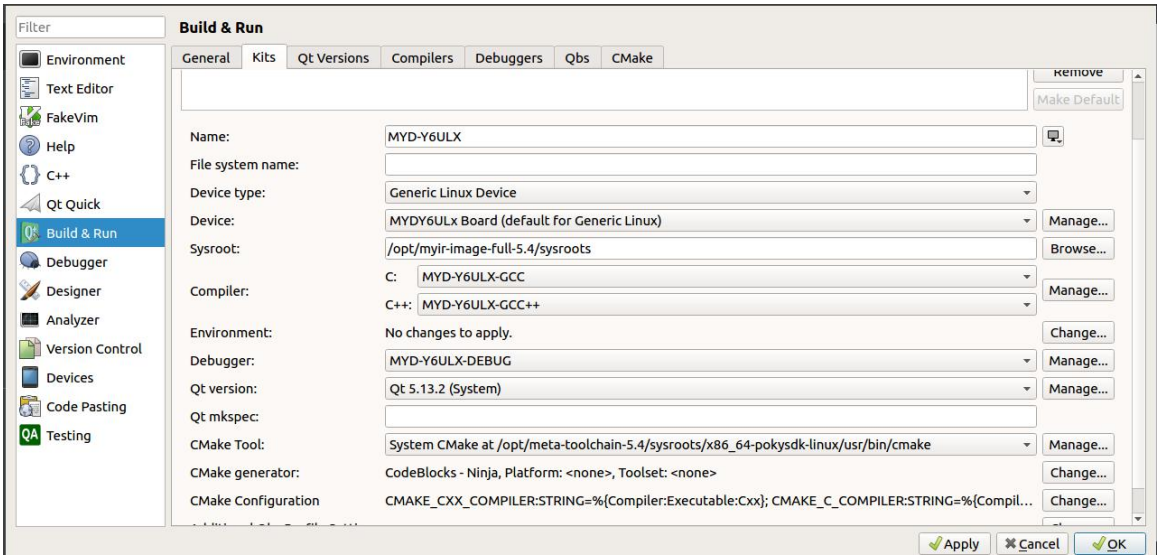

图 7-6. 配置 Kit

7.4.3. 测试 Qt 应用

为了方便测试之前的配置是否正确，新建一个项目。

第一步，在菜单栏选择"File"->"New File or Project"，项目名为：Y6ULX_TESET

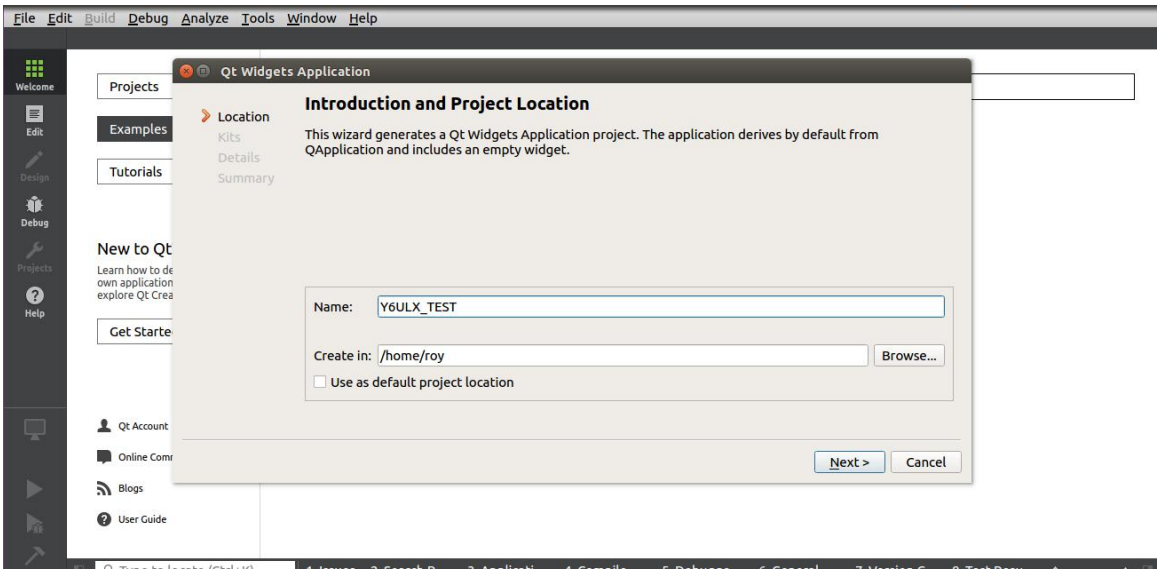

图 7-7. 开启 Qt

第二步，项目打开后，选择"MYD-Y6ULX"选项，完成创建。

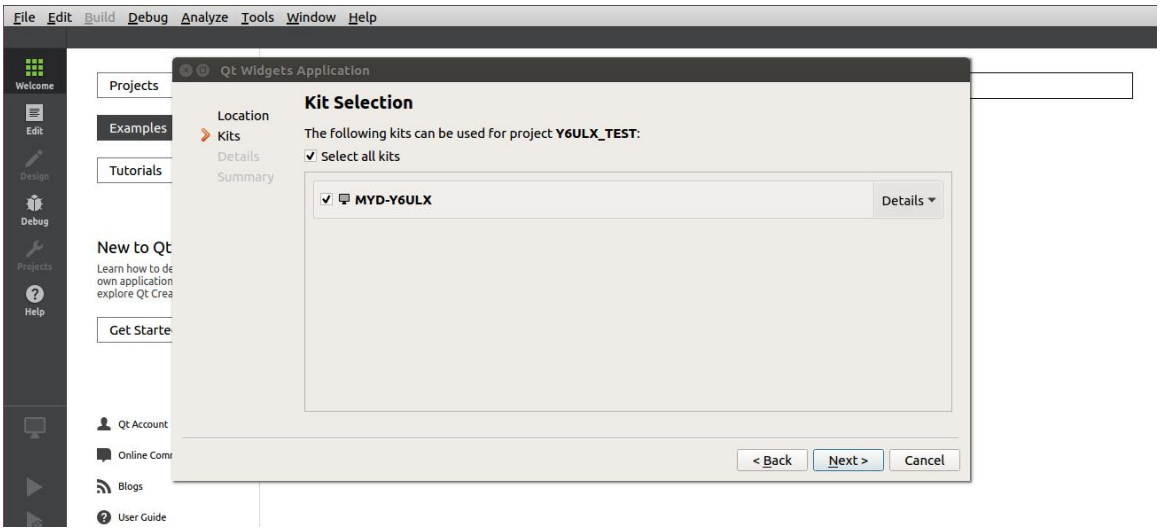

图 7-8. 创建 Qt

第三步，点击菜单栏"Build"->"Build Project Y6ULX_TESET"按钮，即可完成项目的编译，同时下侧会有编译过程输出。

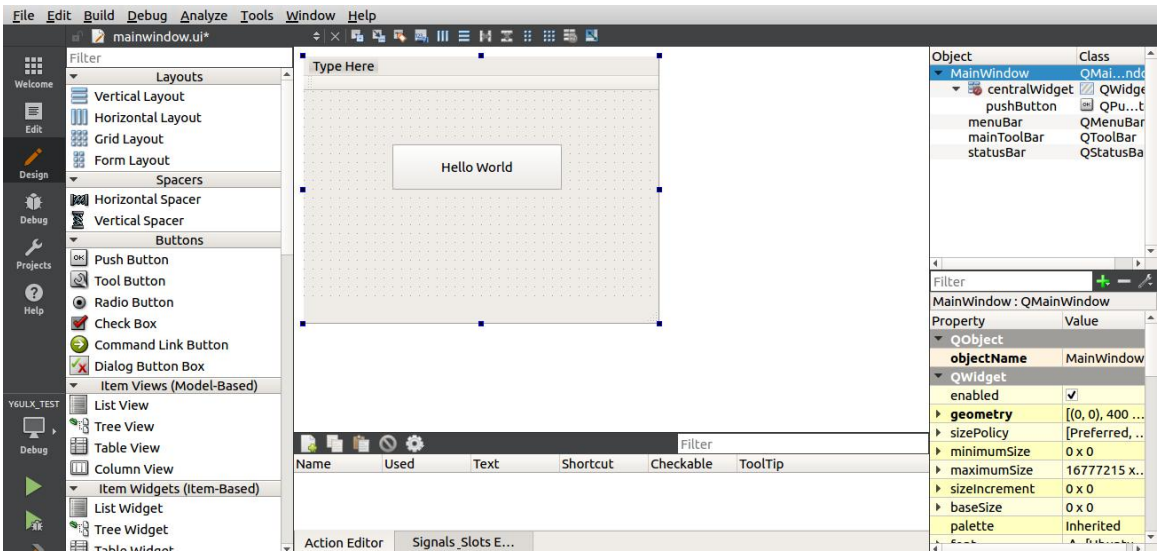

图 7-9. 编辑 Qt 程序

QtCreator 构建 Y6ULX_TEST 项目后，编译好的二进制文件存放在" build-Y6ULX_TEST-MYD_Y6ULX-Debug "目录下，可以使用 file 命令查看，是否编译为 ARM 架构。

```
# 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
```

然后将 Y6ULX_TEST 文件拷贝到开发板下运行即可。

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

运行后将会在 LCD 屏幕上看到 Qt 窗口界面如下：

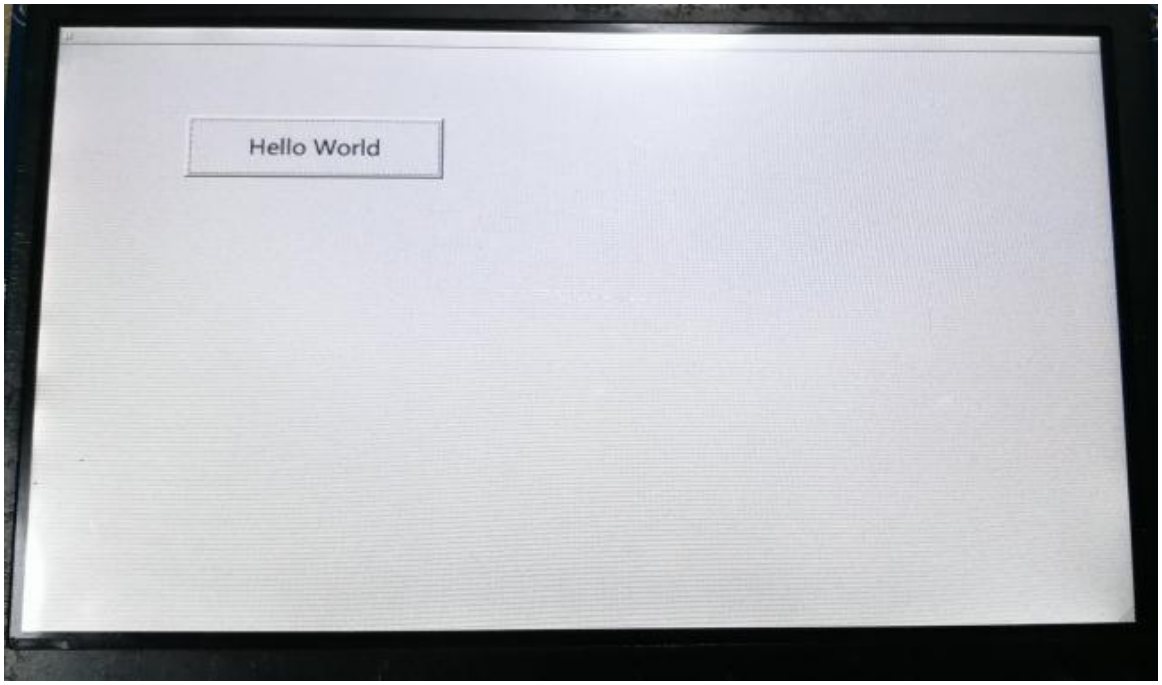

图 7-10. Qt 程序在屏上显示效果

8. 参考资料

- **Yoto 项目 BSP 开发指南**

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

- **Yocto 项目 Linux 内核开发手册**

  https://www.yoctoproject.org/docs/3.1.1/kernel-dev/kernel-dev.html

- **Embedded Linux for i.MX Applications Processors | NXP**

  https://www.nxp.com/design/software/embedded-software/i-mx-software/embedded-linux-for-i-mx-applications-processors:IMXLINUX

2. 附录一 联系我们

深圳总部

负责区域：广东 / 四川 / 重庆 / 湖南 / 广西 / 云南 / 贵州 / 海南 / 香港 / 澳门
电话：0755-25622735 / 18924653967
传真：0755-25532724
邮编：518020
地址：深圳市龙岗区坂田街道发达路云里智能园 2 栋 6 楼 04 室

上海办事处

负责区域：上海 / 湖北 / 江苏 / 浙江 / 安徽 / 福建 / 江西
电话：021-60317628 / 18924632515
传真：021-60317630
邮编：200062
地址：上海市普陀区中江路 106 号北岸长风 I 座 302

北京办事处

负责区域：北京/天津/陕西/辽宁/山东/河南/河北/黑龙江/吉林/山西/甘肃/内蒙古/宁夏
电话：010-84675491 / 13316862895
传真：010-84675491
邮编：102218
地址：北京市昌平区东小口镇中滩村润枫欣尚 2 号楼 1009

销售联系方式

网址：www.myir-tech.com
邮箱：sales.cn@myirtech.com

技术支持联系方式

电话：027-59621648
邮箱：support.cn@myirtech.com

如果您通过邮件获取帮助时，请使用以下格式书写邮件标题：

[公司名称/个人--开发板型号] 问题概述

这样可以使我们更快速跟进您的问题，以便相应开发组可以处理您的问题。

3. 附录二 售后服务与技术支持

凡是通过米尔科技直接购买或经米尔科技授权的正规代理商处购买的米尔科技全系列产品，均可享受以下权益：

1、6 个月免费保修服务周期
2、终身免费技术支持服务
3、终身维修服务
4、免费享有所购买产品配套的软件升级服务
5、免费享有所购买产品配套的软件源代码，以及米尔科技开发的部分软件源代码
6、可直接从米尔科技购买主要芯片样品，简单、方便、快速；免去从代理商处购买时，漫长的等待周期
7、自购买之日起，即成为米尔科技永久客户，享有再次购买米尔科技任何一款软硬件产品的优惠政策
8、OEM/ODM 服务

如有以下情况之一，则不享有免费保修服务：

1、超过免费保修服务周期
2、无产品序列号或无产品有效购买单据
3、进液、受潮、发霉或腐蚀
4、受撞击、挤压、摔落、刮伤等非产品本身质量问题引起的故障和损坏
5、擅自改造硬件、错误上电、错误操作造成的故障和损坏
6、由不可抗拒自然因素引起的故障和损坏

产品返修：

用户在使用过程中由于产品故障、损坏或其他异常现象，在寄回维修之前，请先致电米尔科技客服部，与工程师进行沟通以确认问题，避免故障判断错误造成不必要的运费损失及周期的耽误。

维修周期：

收到返修产品后，我们将即日安排工程师进行检测，我们将在最短的时间内维修或更换并寄回。一般的故障维修周期为 3 个工作日（自我司收到物品之日起，不计运输过程时间），由于特殊故障导致无法短期内维修的产品，我们会与用户另行沟通并确认维修周期。

维修费用：

在免费保修期内的产品，由于产品质量问题引起的故障，不收任何维修费用；不属于免费保修范围内的故障或损坏，在检测确认问题后，我们将与客户沟通并确认维修费用，我们仅收取元器件材料费，不收取维修服务费；超过保修期限的产品，根据实际损坏的程度来确定收取的元器件材料费和维修服务费。

运输费用：

产品正常保修时，用户寄回的运费由用户承担，维修后寄回给用户的费用由我司承担。非正常保修产品来回运费均由用户承担。