

MYD-YF13X Linux

系统开发指南



文件状态： [] 草稿 [√] 正式发布	文件标识：	MYIR-MYD-YF13X-SW-DG-ZH-L5.15.67
	当前版本：	V1.0[文档]
	作 者：	Nene
	创建日期：	2023-05-05
	最近更新：	2023-05-11

版本历史

版本	作者	参与者	日期	备注
V1.0	Nene		20230505	初始版本: uboot 2021.1, kernel 5.15.67, yocto4.1



目 录

版本历史.....	- 2 -
目 录.....	- 3 -
1. 概述.....	- 5 -
1.1. 软件资源.....	- 5 -
1.2. 文档资源.....	- 5 -
2. 开发环境准备.....	- 6 -
2.1. 开发主机环境.....	- 6 -
2.2. 软件开发工具介绍.....	- 8 -
2.3. 安装米尔定制的 SDK.....	- 9 -
3. 使用 Yocto 构建开发板镜像.....	- 12 -
3.1. 简介.....	- 12 -
3.2. 获取源码.....	- 13 -
3.2.1. 从光盘镜像获取源码压缩包.....	- 13 -
3.2.2. 通过 github 获取源码.....	- 13 -
3.3. 快速编译开发板镜像.....	- 15 -
3.4. 构建 SDK (可选).....	- 24 -
4. 如何烧录系统镜像.....	- 25 -
4.1. CubeProg 烧写.....	- 25 -
4.2. 制作 SD 卡启动器.....	- 29 -
5. 如何修改板级支持包.....	- 32 -
5.1. meta-myir-st 层介绍.....	- 32 -
5.2. 板级支持包介绍.....	- 34 -
5.3. 板载 TF-A 编译与更新.....	- 36 -
5.3.1. 在独立的交叉编译环境下编译 TF-A.....	- 36 -



5.3.2. 在 Yocto 项目下编译 TF-A	37 -
5.3.3. 如何单独更新 TF-A	39 -
5.4. 板载 u-boot 编译与更新	41 -
5.4.1. 在独立的交叉编译环境下编译 u-boot	41 -
5.4.2. 在 Yocto 项目下编译 u-boot	42 -
5.4.3. 如何单独更新 U-boot	45 -
5.5. 板载 Kernel 编译与更新	46 -
5.5.1. 在独立的交叉编译环境下编译 Kernel	46 -
5.5.2. 在 Yocto 项目下编译 Kernel	49 -
5.5.3. 如何单独更新 Kernel	52 -
6. 如何适配您的硬件平台	56 -
6.1. 如何创建您的设备树	56 -
6.1.1. 板载设备树	56 -
6.1.2. 设备树的添加	57 -
6.2. 如何根据您的硬件配置 CPU 功能管脚	60 -
6.2.1. GPIO 管脚配置的方法	60 -
6.2.2. 设备树中引用 GPIO	62 -
6.3. 如何使用自己配置的管脚	62 -
6.3.1. U-boot 中使用 GPIO 管脚	62 -
6.3.2. 内核驱动中使用 GPIO 管脚	62 -
6.3.3. 用户空间使用 GPIO 管脚	70 -
7. 如何添加您的应用	76 -
7.1. 基于 Makefile 的应用	76 -
7.2. 基于 Qt 的应用	81 -
7.3. 应用程序开机自启动	82 -
8. 参考资料	90 -
附录一 联系我们	91 -
附录二 售后服务与技术支持	92 -



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。它不仅仅是一个制作文件系统的工具，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。

本文主要介绍基于 Yocto 项目和米尔核心板定制一个完整的嵌入式 Linux 系统的完整流程，其中包括开发环境的准备，代码的获取，以及如何进行 Bootloader, Kernel 的移植，定制适合自身应用需求的 Rootfs 等。我们首先介绍如何基于我们提供的源代码构建适用于 MYD-YF13X 开发板的系统镜像，如何将构建好的镜像烧录到开发板。针对那些基于 MYC-YF135 核心板进行项目开发的用户，我们重点介绍了将这一套系统移植到用户的硬件平台上的方法和一些要点，并通过一些实际的 BSP 移植案例和 Rootfs 定制的案例，使用户能够迅速定制适合自己硬件的系统镜像。

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笔记供开发人员参考，具体的信息参见《MYD-YF13X SDK 发布说明》表 2-4 中的文档列表。

1.1. 软件资源

MYD-YF13X 搭载基于 Linux 5.15.67 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，TF-A 源代码，Optee-os 源码，U-boot 源代码，Linux 内核和各驱动模块的源代码，以及适用于 Windows 桌面环境和 Linux 桌面环境的各种开发调试工具，应用开发样例等。具体的包含的软件信息请参考《MYD-YF13X SDK 发布说明》中第 2 章软件信息中的说明。

1.2. 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，开发指南等不同类别的文档和手册。具体的文档列表参考《MYD-YF13X SDK 发布说明》表 2-4 中的说明。



2. 开发环境准备

本章主要介绍基于 MYD-YF13X 开发板在开发流程所需的一些软硬件环境，包括必要的开发主机环境，必备的软件工具，代码和资料的获取等，具体的准备工作下面将进行详细介绍。

2.1. 开发主机环境

本节将介绍如何搭建适用于 STM32MP13X 系列处理器平台的开发环境。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。STM32MP13X 系列处理器是一个多核异构的处理器，不同的处理器核心会运行不同的系统，也会使用不同的开发环境和工具，具体说明如下：

- 1 个 ARM Cortex A7 内核，可以运行嵌入式 Linux 系统，使用常用的嵌入式 Linux 系统的开发工具。

- 主机硬件

Yocto 项目的构建对开发主机的要求比较高，要求处理器具有双核以上 CPU，8GB 以上内存，160GB 硬盘或更高配置。可以是安装 Linux 系统的主机，也可以是运行 Linux 系统的虚拟机，Windows 系统下的 WSL2 等。

- 主机操作系统

构建 Yocto 项目的主机操作系统可以有很多种选择，详细的信息请参考 Yocto 官方说明 <https://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html#dev-preparing-the-build-host>。一般选择在安装 Fedora, openSUSE, Debian, Ubuntu, RHEL 或者 CentOS 等 Linux 发行版的本地主机上进行构建，这里推荐的是 Ubuntu18.04 64bit 桌面版系统，后续开发也是以此系统为例进行介绍。

- 安装必备软件包

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

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



```
libxml2-utils xmlto docbook bsdmainutils iputils-ping cpio python-wand  
python-pycryptopp python-crypto
```

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

```
PC$ sudo apt-get install ncurses-dev bc linux-headers-generic gcc-multilib libncur  
ses5-dev libncursesw5-dev lrzsz dos2unix lib32ncurses5 repo libssl-dev
```



2.2. 软件开发工具介绍

在定制适用于 ARM Cortex A7 核心的 Linux 系统过程中会用到许多调试，烧写的工具，在米尔提供的光盘镜像目录 03_Tools 下提供了部分工具，除此之外还会用到一系列 ST 提供的工具，简单介绍如下：

- **STM32CubeProg**

ST 推出了新的高集成度的编程工具 STM32CubeProgrammer，它可以在烧录的过程中对未分区的存储设备进行分区，一旦分区就可以使用已经编译好的二进制文件对某个分区单独进行更新。用户可以根据需要选择使用合适的版本，下载地址如下：<https://www.st.com/zh/development-tools/stm32cubeprog.html>。

- **STM32CubeMX**

STM32CubeMX 是 ST 公司推出的专门用于生成 STM32 的 HAL 代码的代码生成软件。它利用可视化界面来进行 STM32 时钟、定时器、DMA、串口、GPIO 等就各种资源的配置。目前 ST 已经将 STM32MPU 系列 CPU 添加进 STM32CubeMX，我们也可以用此工具来配置 TF-A，U-boot 以及 Kernel 的设备树和时钟，所以它可以用来在 Cortex A7 开发时生成设备树和时钟，下载地址如下：<https://www.st.com/zh/development-tools/stm32cubemx.html>。



2.3. 安装米尔定制的 SDK

我们在使用 Yocto 构建完系统镜像之后，还可以使用 Yocto 构建一套可扩展的 SDK。在米尔提供的光盘镜像中包含一个编译好的 SDK 包，位于：03_Tools/QT-SDK/sdk-qt.tar.xz，这个 SDK 中除了包含一个独立的交叉开发工具链还提供 qmake，目标平台的 sysroot，Qt 应用开发所依赖的库和头文件等。用户可以直接使用这个 SDK 来建立一个独立的开发环境，单独编译 Bootloader，Kernel 或者编译自己的应用程序，具体过程在后面的章节中将会详细介绍。这里先介绍 SDK 的安装步骤，如下：

- 拷贝 SDK 到 Linux 目录并解压

将 SDK 压缩包拷贝到 Ubuntu 下的用户工作目录，如 \$HOME/work 下，解压文件，得到安装脚本文件，如下：

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

- 查看脚本文件

进入 SDK 目录，可以看到下面安装脚本文件：

```
meta-toolchain-qt5-openstlinux-weston-myr-yf13x-x86_64-toolchain-4.0.4-snapshots
hot.host.manifest
meta-toolchain-qt5-openstlinux-weston-myr-yf13x-x86_64-toolchain-4.0.4-snapshots
hot.sh
meta-toolchain-qt5-openstlinux-weston-myr-yf13x-x86_64-toolchain-4.0.4-snapshots
hot.target.manifest
meta-toolchain-qt5-openstlinux-weston-myr-yf13x-x86_64-toolchain-4.0.4-snapshots
hot.testdata.json
```

- 执行安装脚本

```
PC$ ./meta-toolchain-qt5-openstlinux-weston-myr-yf13x-x86_64-toolchain-4.0.4-snapshots.sh
```

- 选择安装目录目录

SDK 默认被安装到 /opt/st/myir-yf13x/4.0.4-snapshot 目录下，用户也可以根据提示自己选择合适的目录，具体根据提示进行操作：



ST OpenSTLinux - Weston - (A Yocto Project Based Distro) SDK installer version 4.0.4-snapshot

```
=====
=====
```

Enter target directory for SDK (default: /opt/st/myir-yf13x/4.0.4-snapshot):
The directory "/opt/st/myir-yf13x/4.0.4-snapshot" already contains a SDK for this architecture.

If you continue, existing files will be overwritten! Proceed [y/N]? y

Extracting SD

```
K.....
.....done
```

Setting it up...done

SDK has been successfully set up and is ready to be used.

Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.

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

● 测试 SDK

安装完成后，使用以下命令设置环境变量，测试 SDK 是否完成：

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

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

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

```
Copyright (C) 2021 Free Software Foundation, Inc.
```

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

米尔提供的 SDK 中除了包含交叉工具链，还包含 Qt 库，qmake 等开发 Qt 应用程序所需的资源，这些是后续使用 QT Creator 进行应用程序开发和调试的基础。





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。关于 Yocto 的基础知识，请参考 www.yoctoproject.org。

米尔提供的光盘镜像中 04_sources 目录下提供了适用于 MYD-YF13X 开发板的 Yocto 元文件和数据，帮助开发者构建出可运行在 MYD-YF13X 开发板上的不同类型 Linux 系统镜像，如带 Qt5.15.3 图形库的 myir-image-full 系统镜像，不带 GUI 界面的 myir-image-core 系统镜像。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

我们提供两种获取源码的方式，一种是直接从米尔光盘镜像 04_sources 目录中获取压缩包，另外一种是使用 repo 获取位于 github 上实时更新的源码进行构建，请用户根据实际需要选择其中一种进行构建。

提示：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-YF13X 已经把相关的软件打包好，用户可以直接解压使用，减少重复下载的时间。

3.2.1. 从光盘镜像获取源码压缩包

压缩的源码包位于米尔开发包资料 04_Sources/yf13x-yocto-stm32mp1-5.15.67.tar.bz2。拷贝压缩包到用户指定目录，如 \$HOME/Yocto 目录，这个目录将作为后续构建的顶层目录，按照下面的方式解压后出现 layers 目录：

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

列出 layers 目录内容如下：

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

5 directories
```

3.2.2. 通过 github 获取源码

目前 MYD-YF13X 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，代码仓库地址请查看《MYD-YF13X_SDK 发布说明》。用户可以使用 repo 获取和同步 github 上的代码。具体操作方法如下：



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

代码同步成功之后，同样在\$HOME/github 目录下得到一个 layers 文件夹，里面包含 MYD-YF13X 开发板相关的源码或者源码仓库的路径。

注意：服务器的 python 版本需要 3.0 以上。



3.3. 快速编译开发板镜像

在使用 Yocto 项目进行系统构建之前都需要先设置相应环境变量，我们在构建 myir-image-full 之前需要使用米尔提供的 envsetup.sh 脚本进行环境变量的设置，设置过程中会创建一个构建目录（如 build-openstlinuxweston-myd-yf13x-emmc），后续构建过程，以及输出文件都包含在这个目录。

- 执行环境变量设置脚本

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

注：NAND 版本 MACHINE 需要选择为 myd-yf13x-nand。

执行上述配置会列出所有适用于 MYD-YF13X 的系统镜像，如下所示：

```
[HOST DISTRIB check]
```

```
Linux Distro: Ubuntu
```

```
Linux Release: 18.04
```

```
Required packages for Linux Distro:
```

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

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

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

```
=====
```

```
Configuration files have been created for the following configuration:
```

```
DISTRO : openstlinux-weston
```



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

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

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

WITH_EULA_ACCEPTED: NO
```

```
=====
=====
```

Available images for OpenSTLinux layers are:

- The system based on QT framework

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

- Other OpenSTLinux images:

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

You can now run 'bitbake <image>'



配置脚本执行完成后将进入 build-openstlinuxweston-myd-yf13x-emmc 目录下，在此目录下就可以开始构建系统。

● 构建 myir-image-full 镜像

选择构建不同的系统镜像，需使用不同的 bitbake 命令参数，具体命令参见下表，我们选择 myir-image-full 为例进行介绍。

表 3-1.系统镜像可选列表

系统名	命令
基于 qt5.15 的 measy-hmi2.0 系统	bitbake myir-image-full
基于 openstlinux 的 core 系统	bitbake myir-image-core

注：如是 NAND 镜像，只有 myir-image-core，带 lvgl 的 demo，整体编译镜像执行 **bitbake myir-image-core**

表 3-2. Bitbake 部分常用命令

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

注意：构建过程中下载大量的包资源，为了节约时间，推荐使用 myir 打包好的资源 Yocto-qt-downloads.tar.xz 解压到 downloads。

```
PC$:/build-openstlinuxweston-myd-yf13x-emmc$ bitbake myir-image-full -k
NOTE: Started PRServer with DBfile: /media/system1/home/nene/ST/build-openstlinuxweston-myd-yf13x-emmc/cache/prserv.sqlite3, Address: 127.0.0.1:38703, PID: 6386
Loading cache: 100% |#####
#####
#####| Time: 0:00:01
Loaded 4483 entries from dependency cache.
```



```
Parsing recipes: 100% |#####  
#####  
#####| Time: 0:00:01
```

Parsing of 2939 .bb files complete (2938 cached, 1 parsed). 4483 targets, 386 skipped, 0 masked, 0 errors.

NOTE: Resolving any missing task queue dependencies

Build Configuration:

```
BB_VERSION      = "2.0.0"  
BUILD_SYS       = "x86_64-linux"  
NATIVELSBSTRING = "universal"  
TARGET_SYS      = "arm-ostl-linux-gnueabi"  
MACHINE         = "myd-yf13x-emmc"  
DISTRO          = "openstlinux-weston"  
DISTRO_VERSION  = "4.0.4-snapshot-20230506"  
TUNE_FEATURES   = "arm vfp cortexa7 neon vfpv4 thumb callconvention-hard"  
TARGET_FPU      = "hard"  
DISTRO_CODENAME = "kirkstone"  
ACCEPT_EULA_myd-yf13x-emmc = "1"  
GCCVERSION      = "11.%"  
PREFERRED_PROVIDER_virtual/kernel = "linux-myir"  
meta-python  
meta-oe  
meta-gnome  
meta-initramfs  
meta-multimedia  
meta-networking  
meta-webserver  
meta-fileystems  
meta-perl  
meta-st-stm32mp
```



```
meta-qt5
meta-st-openstlinux = "<unknown>:<unknown>"
meta-myr-st-yf13x = "master:8132dfabf8abf729d5869bcd9c596f6168b4ab67"
meta = "<unknown>:<unknown>"

Initialising tasks: 100% |#####
#####
#####| Time: 0:00:01
Sstate summary: Wanted 64 Local 59 Mirrors 0 Missed 5 Current 390 (92% match,
98% complete)
NOTE: Executing Tasks
```

等待系统构建完成后，在" build-openstlinuxweston-myd-yf13x-emmc/tmp-glibc/deploy/images/myir "目录下生成各种格式的系统镜像文件，以下是构建后生成的文件信息列表：

```
myir/
├── arm-trusted-firmware
├── build-myr-image-full-openstlinux-weston-myd-yf13x-emmc
├── fip
├── flashlayout_myr-image-full
├── kernel
├── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253-license_content.html
├── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.rooftfs.ext4
├── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.rooftfs.manifest
├── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.rooftfs.tar.xz
└── myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.testdata.json
```



```

└─ myir-image-full-openstlinux-weston-myd-yf13x-emmc.ext4 -> myir-image-f
ull-openstlinux-weston-myd-yf13x-emmc-20230508025253.rootfs.ext4
└─ myir-image-full-openstlinux-weston-myd-yf13x-emmc-license_content.html
-> myir-image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253-licens
e_content.html
└─ myir-image-full-openstlinux-weston-myd-yf13x-emmc.manifest -> myir-ima
ge-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.rootfs.manifest
└─ myir-image-full-openstlinux-weston-myd-yf13x-emmc.tar.xz -> myir-image-
full-openstlinux-weston-myd-yf13x-emmc-20230508025253.rootfs.tar.xz
└─ myir-image-full-openstlinux-weston-myd-yf13x-emmc.testdata.json -> myir-
image-full-openstlinux-weston-myd-yf13x-emmc-20230508025253.testdata.json
└─ optee
└─ scripts
└─ st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bo
otfs.ext4
└─ st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bo
otfs.manifest
└─ st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bo
otfs.tar.xz
└─ st-image-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.tes
tdata.json
└─ st-image-bootfs-openstlinux-weston-myd-yf13x-emmc.ext4 -> st-image-bo
otfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bootfs.ext4
└─ st-image-bootfs-openstlinux-weston-myd-yf13x-emmc.manifest -> st-image
-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bootfs.manifest
└─ st-image-bootfs-openstlinux-weston-myd-yf13x-emmc.tar.xz -> st-image-bo
otfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.bootfs.tar.xz
└─ st-image-bootfs-openstlinux-weston-myd-yf13x-emmc.testdata.json -> st-i
mage-bootfs-openstlinux-weston-myd-yf13x-emmc-20230508025253.testdata.jso
n

```



```

|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spi
nand_2_128.ubinize.cfg.ubi
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spi
nand_2_128.userfs.ubi
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spi
nand_2_128.userfs.ubifs
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.test
data.json
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.use
rfs.ext4
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.use
rfs.manifest
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.use
rfs.tar.xz
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc.ext4 -> st-image-user
fs-openstlinux-weston-myd-yf13x-emmc-20230507094753.userfs.ext4
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc.manifest -> st-image
-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.userfs.manifest
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubi ->
st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spinan
d_2_128.userfs.ubi
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubifs
-> st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_spina
nd_2_128.userfs.ubifs
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubiniz
e.cfg.ubi -> st-image-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094
753_spinand_2_128.ubinize.cfg.ubi
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc.tar.xz -> st-image-us
erfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.userfs.tar.xz
|— st-image-userfs-openstlinux-weston-myd-yf13x-emmc.testdata.json -> st-im
age-userfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.testdata.json

```



```

└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_
spinand_2_128.ubinize.cfg.ubi
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_
spinand_2_128.vendorfs.ubi
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_
spinand_2_128.vendorfs.ubifs
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.t
estdata.json
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.v
endorfs.ext4
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.v
endorfs.manifest
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.v
endorfs.tar.xz
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.ext4 -> st-image-v
endorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.vendorfs.ext4
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.manifest -> st-ima
ge-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.vendorfs.ma
nifest
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubi
-> st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_s
pinand_2_128.vendorfs.ubi
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubi
fs -> st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753_
spinand_2_128.vendorfs.ubifs
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc_spinand_2_128.ubi
nize.cfg.ubi -> st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc-2023050
7094753_spinand_2_128.ubinize.cfg.ubi
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.tar.xz -> st-image-
vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.vendorfs.tar.xz

```



```
└─ st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.testdata.json -> st-
image-vendorfs-openstlinux-weston-myd-yf13x-emmc-20230507094753.testdata.
json
└─ st-initrd-openstlinux-weston-myd-yf13x-emmc
└─ u-boot

7 directories, 48 files
```

表 3-3. 生成 images 文件清单说明 (非全部)

名称	描述
arm-trusted-firmware	Tf-a 烧录包生成文件
uboot	uboot 烧录包生成文件
kernel	内核与设备树生成文件
flashlayout_myr-image-full	烧录 TSV 文件夹
st-image-bootfs-openstlinux-weston-myd-yf13x-emmc.ext4	bootfs 分区文件, 主要包含了内核 uImage,设备树, 启动配置文件等
st-image-userfs-openstlinux-weston-myd-yf13x-emmc.ext4	Userfs 分区文件, 主要包含了适用于 STM32MP1 的 DEMO 程序
st-image-vendorfs-openstlinux-weston-myd-yf13x-emmc.ext4	Vendorfs 分区文件, 主要包含了第三方库文件, 如 eglfs 库等
myir-image-full-openstlinux-weston-myd-yf13x-emmc.ext4	Rootfs 分区文件, 主要包含根文件系统
scripts	create_sdcard_from_flashlayout.sh 制作 SD 启动包



3.4. 构建 SDK (可选)

米尔已经提供较完整的 SDK 安装包，用户可直接使用。但当用户需要在 SDK 中引入新的库，则需要重新使用 Yocto 构建出新的 SDK 工具。

本节只简单对米尔提供的 SDK 做构建说明，使用如下构建命令生成 SDK 包：

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

等待构建完成后，将在 “build-openstlinuxeglfs-myd-yf13x-emmc/tmp-glibc/deploy/sdk” 路径下生成 SDK 安装包，安装方法请查看 2.3 节。

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



4. 如何烧录系统镜像

米尔公司设计的 MYC-YF13X 系列核心板与开发板是基于 ST 公司的 STM32MP135 系列微处理器,其启动方式多样,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。更新方式主要有以下几种:

- CubeProg 烧写: 适用于研发调试, 测试等场景。
- 制作 SD 卡启动器: 适用于研发调试, 快速启动等场景。

4.1. CubeProg 烧写

1) 工具需求

- 开发板一块
- USB Type_C 一根
- 电源适配器
- STM32CubeProgrammer 官方软件

2) 设置硬件

选择启动模式, 将拨码开关拨到 Download 模式 (B2/B1/B0 : 0 0 0)。连接好硬件, 将开发板的 J14 OTG 接口与电脑连接, 插入电源适配器。如下图所示:

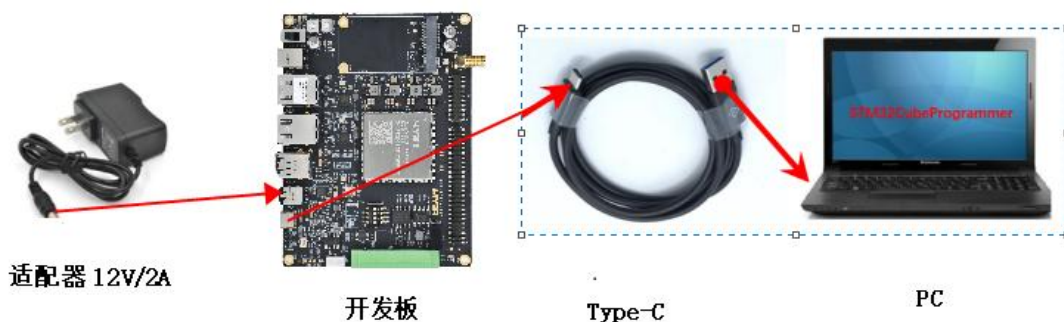


图 4.1.硬件连接图

关于更新方式,根据你使用的 PC 平台不同,也有两种推荐方式,可以二选一:



Windows 平台下,建议您使用 Window 平台下安装的 STM32CubeProgrammer, 通过 USB 烧写。Linux 平台下,建议您使用 Linux 平台下安装的 STM32CubeProgrammer, 通过 USB 烧写。

3) 在 windows 下烧录系统

打开 Windows 平台已经安装好的 STM32CubeProgrammer 软件, 选择 USB (标号 1) 烧录方式。点击 connect (标号 2) 按钮连接。检查能否正常显示开发板的信息如下图。

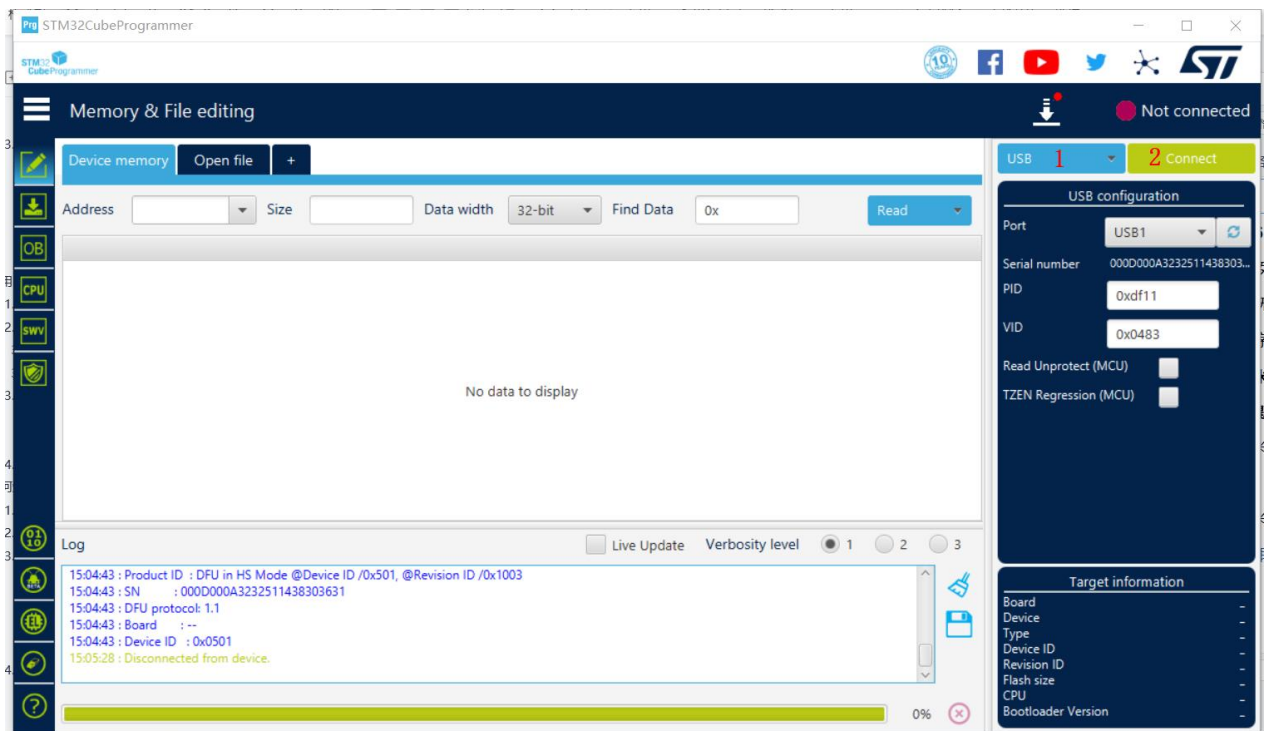


图 4-2. 连接设备

选择烧录系统配置文件, 不同的存储设备, 配置文件不同, 这里选择 SD Card 为例。解压开发包资料 02_Image/myir-image-full/, 然后选择图 4-3 (标号 1) 位置选择 SD Card 的 TSV 文件, 如文件: FlashLayout_emmc_myb-stm32mp135x-512m-optee.tsv

点击 Browse 选择 image 所在的文件夹名即可, 如图 4-3 中红色矩形框部分。



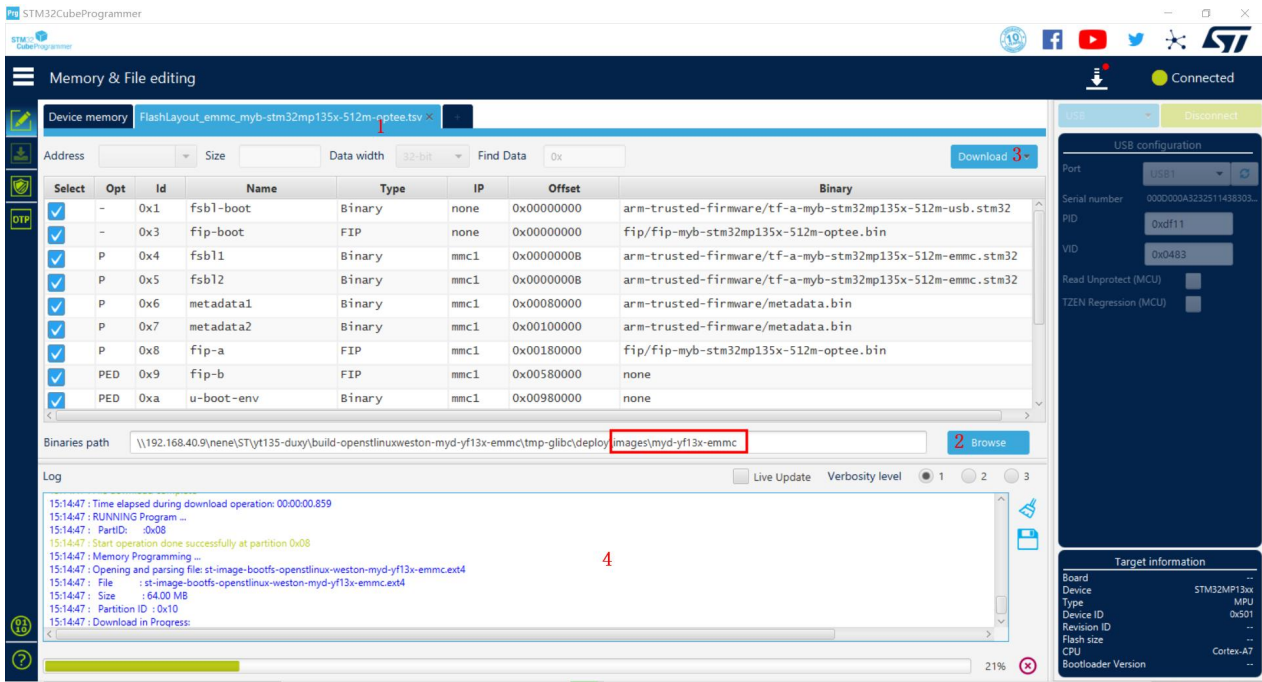


图 4-3. 选择镜像目录

配置完成后点击图 4-3（标号 3）位置（download）下载更新文件，烧录系统时间较久，请耐心等待更新完成，其他存储设备如 NandFlash/TF 在 windows 的更新步骤一样。

4) Linux 上进行更新

安装 linux 版本 STM32CubeProgrammer

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

安装完毕后将生成目录 STMicroelectronics/STM32Cube/STM32CubeProgrammer

新建工作目录 work，然后拷贝开发包资料/02_Image/myir-image-full/所有解压完的文件到工作目录，如：/home/myir/work。

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

```
PC$ cd /home/myir/work
```

添加 STM32CubeProgrammer 到 PATH 环境变量：

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



启动安装好的 STM32CubeProgrammer 软件进行烧写更新，执行下面命令更新二进制文件，以 Micro SD Card 为例：

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

烧录过程时间较长，请耐心等待更新完成，其他存储设备如 NandFlash/eMMC 在 Linux 上的更新步骤一样。



4.2. 制作 SD 卡启动器

以下步骤均在 Windows 系统下制作。

1) 准备工作

- SD 卡 (不少于 4GB)
- MYD-YF13X 开发板
- 制作镜像工具 Win32DiskImager-1.0.0-binary (路径: \03_Tools\myir tools)

表 4-1. 镜像包列表

镜像名	包名	适用核心板
myir-image-full	flashlayout_myir-image-full_FlashLayout_sdcard_yf13x-4e512d.zip	MYD-YF13X-4E512D
myir-image-core	flashlayout_myir-image-core_FlashLayout_sdcard_myb-stm32mp135x-512m-optee.zip	MYD-YF13X-4E512D
myir-image-lvgl	FlashLayout_sdcard_myb-stm32mp135x-256m-optee.zip	MYD-YF13X-256N256D

2) 制作 SD 卡启动 (以 myir-image-full 系统为例)

● 解压下面资源

Win32DiskImager-1.0.0-binary.zip

flashlayout_myir-image-full_FlashLayout_sdcard_yf13x-4e512d.zip

● 将镜像文件写入 Micro SD Card

将 Micro SD Card 放入读卡器读卡器, 然后插入电脑, 双击打开 Win32DiskImager.exe 读出 U 盘分区, 点击箭头方向加载镜像文件。



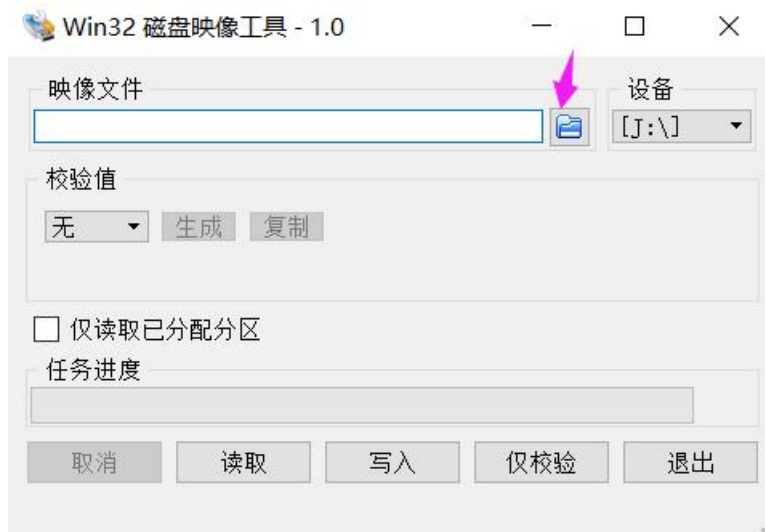


图 4-4.工具配置 1

注意选择 “.” 才能显示系统包，选择好系统包后点击打开即可。

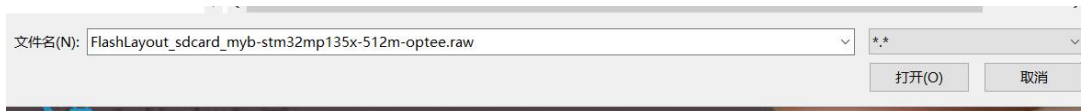


图 4-5.工具配置

加载完镜像后点击“写入”按钮即可，会弹出警告，点击“Yes”等待写入完成。

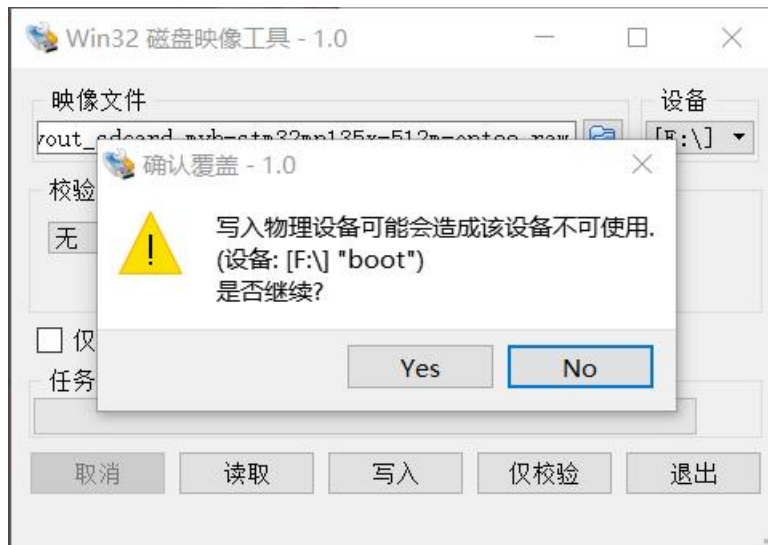


图 4-6.工具配置

等待写入完成，大约 3-4 分钟完成，此速度取决于 SD 的读写速度。



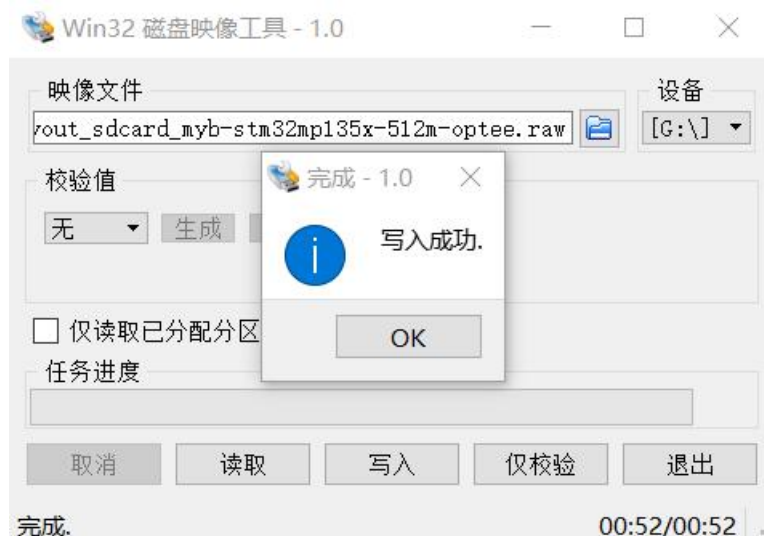


图 4-7.烧写成功

- 检查是否烧写成功

当写入完成后，即可使用此 SD 进行启动，将 SD 插入开发板背面的 SD 卡槽（J18）即可使用 SD 卡启动系统。



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-YF13X 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-YF13X 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-YF13X 核心板设计的底板与 MYB-YF13X 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-myr-st 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-myr-st 层基于在 ST 官方的 meta-st 层建立的适用于 MYD-YF13X 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-YF13X 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-myr-st 层包含的具体内容如下：

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



9 directories

表 5-1. meta-myir-st 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
recipes-app	包含多种米尔的应用程序,如 measy-hmi2.0
recipes-bsp	包含 tf-a 与 uboot 等配置资源
recipes-kernel	包含 linux 内核的资源与第三方固件资源
recipes-security	包含 optee 配置资源
recipes-myir	包含文件系统的配置信息
scripts	Yocto 环境配置

源代码与数据	描述
conf	包括开发板软件配置资源信息
recipes-app	包含多种米尔的应用程序,如 measy-hmi2.0
recipes-bsp	包含 tf-a 与 uboot 等配置资源
recipes-kernel	包含 linux 内核的资源与第三方固件资源
recipes-myir	包含文件系统的配置信息
scripts	Yocto 环境配置

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-app 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 的源码将全部开源。

表 5-2.BSP 支持的开源协议

IP 项目	开源协议	描述
tf-a-myrir	BSD-3-Clause	Trusted Firmware-A
u-boot-myrir-extlinux	MIT	Uboot extlinux.conf
u-boot-myrir	GPL-2.0-or-later	uboot
linux-myrir	GPL-2.0-only	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-YF13X 核心板设计的硬件 BSP 代码可以查看 meta-myrir-st 中的 recipes-bsp 和 recipes-kernel 这两个配方的内容。

recipes-bsp 中只包含了 Bootloader 部分的 trusted-firmware-a 和 u-boot，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-YF13X 核心板硬件修改这部分的内容。

```
recipes-bsp
├── trusted-firmware-a
└── 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。

```
recipes-kernel/
├── linux
└── linux-firmware
```



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 TF-A 编译与更新

TF-A(Trusted Firmware-A)是由 ARM®提供的安全类软件的参考实现。TF-A 最初设计是为 Armv8-A 平台，现在由 STMicroelectronics 适配并使用在 Armv7-A 平台。现在该项目已经将 Trusted Firmware 项目移交给 Linaro 作为开源项目来管理。这部分代码遵守 BSD-3-Clause 版权。

在 meta-myr-st 层模型中，TF-A 是 recipes-bsp 的一个部分。在使用 Trusted Boot Chain（关于 Boot Chain 的详细信息，请查看 https://wiki.st.com/stm32mpu/wiki/Boot_chain_overview）方式启动时，它用作 FSBL（First Stage Boot Loader），主要实现 DDR, Clock 等核心资源的初始化，基于 MYC-YF13X 核心板设计的硬件一般不需要修改 TF-A，如果特殊情况下，需要修改某个时钟，或者需要对某个管脚进行早期的控制时，可以参照下面的方法进行移植。

5.3.1. 在独立的交叉编译环境下编译 TF-A

1) 获取 TF-A 源代码

拷贝开发包 04_Source/Tf-a/MYiR-STM32-tf-a.tar.bz2 到指定自定义的 work 目录（如 /home/work），解压进入源码目录并查看对应的文件信息，如拷贝到 work 目录：

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

- 源代码目录：myir-st-arm-trusted-firmware
- 编译脚本：Makefile.sdk
- 指导说明：README.HOW_TO.txt

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

2) 配置和编译源代码

- 加载 SDK 环境变量到当前 shell

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



- 进入源代码目录

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

- 配置与编译源代码

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

编译成功后，在 build/目录下会生成 TF-A 的二进制文件如下表所示：

表 5-3. TF-a 编译生成的二进制文件

名称	描述
tf-a-myb-stm32mp135x-256m-sdcard.stm32	包含 STM32 信息头的 U-Boot 二进制文件
tf-a-myb-stm32mp135x-256m-usb.stm32	
tf-a-myb-stm32mp135x-512m-sdcard.stm32	
tf-a-myb-stm32mp135x-512m-usb.stm32	
tf-a-bl2-usb.elf	可用于 debug 调试的 elf 文件

5.3.2. 在 Yocto 项目下编译 TF-A

系统开发中，建议使用 git 来管理源代码，更便捷的管理代码的多个版本。如果在修改代码并测试后，代码无误，可以提交到本地仓库中，这样使用 Yocto 来构建系统也更容易。

如果希望按照第三章中的流程完整的构建新的镜像，则需要将修改后的源代码提交到 git 仓库，同时修改 layers/meta-myr-st/recipes-bsp 中对应的配方。

用户可以自己建立 git 仓库管理代码，然后修改配方下的 SRC_URI 以及 SRCREV 来匹配自己的代码。下面以 github 下的建立的仓库为例加以说明。

首先找到最新提交的 commit，在 github 仓库下找到最新的 commit 值，并将值复制。

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

修改 layers/meta-myr-st/recipes-bsp/trusted-firmware-a/tf-a-myr-common.inc 下的 SRCREV 值。

```
FILESEXTRAPATHS:prepend := "${THISDIR}/tf-a-myr:"
```



```
SECTION = "bootloaders"
```

```
LICENSE = "BSD-3-Clause"
```

```
LIC_FILES_CHKSUM = "file://license.rst;md5=1dd070c98a281d18d9eefd938729b031"
```

```
SRC_URI += "git://github.com/MYiR-Dev/myir-st-arm-trusted-firmware.git;protocol=https;branch=develop-yf13x-v2.6"
```

```
SRCREV= "4cc966a5372b13517343009b2f8797cb99828ce8"
```

```
TF_A_VERSION = "v2.6"
```

```
TF_A_SUBVERSION = "stm32mp"
```

```
TF_A_RELEASE = "r2"
```

```
PV = "${TF_A_VERSION}-${TF_A_SUBVERSION}-${TF_A_RELEASE}"
```

```
ARCHIVER_ST_BRANCH = "develop-yf13x-v2.6"
```

```
ARCHIVER_ST_REVISION = "${PV}"
```

```
ARCHIVER_COMMUNITY_BRANCH = "develop-v2.6-stm32mp"
```

```
ARCHIVER_COMMUNITY_REVISION = "${TF_A_VERSION}"
```

```
S = "${WORKDIR}/git"
```

```
# -----
```

```
# Configure devupstream class usage
```

```
# -----
```

```
BBCLASSEXTEND = "devupstream:target"
```

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

```
SRCREV:class-devupstream = "4cc966a5372b13517343009b2f8797cb99828ce8"
```



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

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

修改完成后，可在构建目录下单独使用 “bitbake tf-a-myr” 命令构建 arm-trusted-firmware 或者按照 3.3 章重新构建整个系统。

5.3.3. 如何单独更新 TF-A

编译成功之后，将 TF-A 镜像烧录进 Micro SD 卡（Micro SD 卡必须已分好区，如未分区，可按照 3.5.1 节烧录系统到 Micro SD 卡即可完成分区）。

/dev/mmcblk 分区信息

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

使用 dd 命令将镜像烧录到 SD 卡指定分区：



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

烧录到核心板 eMMC 储存器中指定分区：

注意：以下命令操作在开发板上运行，执行错误或文件写入错误，会导致开发板无法启动。如出现无法启动的情况，请重新烧写开发板即可。

```
root@myir~#: echo 0 > /sys/class/block/mmcblk1boot1/force_ro
root@myir~#: echo 0 > /sys/class/block/mmcblk1boot2/force_ro
root@myir~#: dd if=tf-a-myb-stm32mp135x-512m-sdcard.stm32 of=/dev/mmcblk1boot1 conv=fdatasync
root@myir~#: dd if=tf-a-myb-stm32mp135x-512m-sdcard.stm32 of=/dev/mmcblk1boot2 conv=fdatasync
root@myir~#: echo 1 > /sys/class/block/mmcblk1boot1/force_ro
root@myir~#: echo 1 > /sys/class/block/mmcblk1boot2/force_ro
```

/dev/mtd 分区信息

```
/dev/mtdblock0 -- tf-a-myb-stm32mp135x-256m-sdcard.stm32
/dev/mtdblock1 -- tf-a-myb-stm32mp135x-256m-sdcard.stm32
/dev/mtdblock2 -- metadata.bin
/dev/mtdblock3 -- metadata.bin
/dev/mtdblock4 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock5 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock6 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock7 -- fip-myb-stm32mp135x-256m-optee.bin
/dev/mtdblock8 --myir-image-core-openstlinux-weston-my-d-yf13x_spinand_2_128.ubi
```

烧录到核心板 NAND 储存器中指定分区：




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

5.4. 板载 u-boot 编译与更新

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>

STM32MP1 平台也使用 boot chains 做启动引导程序，不同的 boot chains 模式会对应不同启动阶段。

Boot chains 的引入是为了区分两类启动流程中用的的各个软件组件或程序，STM32MP1 平台的 Boot chains 有两种方式，也可以查看 https://wiki.st.com/stm32mpu/wiki/Boot_chain_overview 了解更多详情。

- Basic boot chain：使用 U-Boot SPL 作为 FSBL，U-Boot 作为 SSBL
- Trusted boot chain：使用 Trusted Firmware-A (TF-A) 作为 FSBL，使用 U-Boot 作为 SSBL

5.4.1. 在独立的交叉编译环境下编译 u-boot

1) 获取 u-boot 源代码

拷贝开发包 04_Source/Bootloader/MYiR-STM32-u-boot.tar.bz2 到指定自定义的 work 目录（如/home/work），解压进入源码目录并查看对应的文件信息，如拷贝到 work 目录：

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

- 源代码目录：myir-st-u-boot
- 编译脚本：Makefile.sdk
- 指导说明：README.HOW_TO.txt

```
-rwxrwxrwx 1 root root 3816 Jun 9 20:51 Makefile.sdk
```



```
drwxr-xr-x 25 root root 4096 Sep 18 01:16 myir-st-u-boot
-rw----- 1 licy licy 10579 Apr 21 03:46 README.HOW_TO.txt
```

2) 配置与编译

- 加载 SDK 环境变量到当前 shell

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

- 进入源代码目录

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

- 配置与编译源代码

(1) 在源码目录下配置编译方式：（此方式不推荐）

```
PC$ make myc_stm32mp13_512m_defconfig
PC$ make DEVICE_TREE=myb-stm32mp135x-512m all
```

(2) 使用 Makefile.sdk 脚本配置编译方式：

此方式将所有生成目标定义在 Makefile.sdk 脚本中，可使用单条命令完成：（推荐方式）

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

进入到编译输出文件夹：

```
PC$ cd ../deploy
```

表 5-4.u-boot 烧录文件描述

名称	描述
u-boot-stm32mp13.elf	elf 文件的 uboot,可用于 gdb 调试

5.4.2. 在 Yocto 项目下编译 u-boot

当用户按照 5.4.1 中的迭代开发过程改好 U-boot 的代码之后，也可以使用 Yocto 进行整个镜像的构建。此时需要将修改后的源代码提交到 git 仓库，同时修改元层的对应源代码的 commit 值。首先在 git 仓库下找到最新的 commit 值，并将值复制，然后通过 SRC_URI 和 SRCREV 变量将用户自己创建的 git 仓库 URL 和对应的 commit 值提供给 BSP 配方，以便配方能够找到并获取正确的代码，参考示例如下。



```
PC$: git log --pretty=oneline
```

```
7e709a983f0816e63e8c13387d66f01225f01848 (HEAD -> develop-v2021.10-stm32mp, tag: V0.1.0_20230510_Alpha, origin/develop-v2021.10-stm32mp, origin/HEAD) FEAT: modify the boot parameter to
```

修改 layers/meta-myr-st/recipes-bsp/u-boot/u-boot-myr-common_2021.10.inc 下的 SRCREV 值。

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

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

```
SECTION = "bootloaders"
```

```
LICENSE = "GPL-2.0-or-later"
```

```
LIC_FILES_CHKSUM = "file://Licenses/README;md5=5a7450c57ffe5ae63fd732446b988025"
```

```
DEPENDS += "dtc-native bc-native"
```

```
DEPENDS += "flex-native bison-native"
```

```
DEPENDS += "python3-setuptools-native"
```

```
COMPATIBLE_MACHINE = "(stm32mpcommon)"
```

```
SRC_URI = "git://github.com/MYiR-Dev/myir-st-u-boot.git;protocol=https;branch=${SRCBRANCH}"
```

```
SRCREV = "7e709a983f0816e63e8c13387d66f01225f01848"
```

```
SRCBRANCH = "develop-yf13x-v2021.10"
```

```
# debug and trace
```

```
SRC_URI += "${@bb.utils.contains('ST_UBOOT_DEBUG_TRACE', '1', 'file://0098-silent_mode.patch', d)}"
```

```
U_BOOT_VERSION = "v2021.10"
```



```

U_BOOT_SUBVERSION = "stm32mp"
U_BOOT_RELEASE = "r2"

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

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

S = "${WORKDIR}/git"

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

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

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

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

```



修改完成后，可在构建目录下单独使用“bitbake u-boot-myr”命令构建 u-boot 或者按照 3.3 章重新构建这个系统。

5.4.3. 如何单独更新 U-boot

编译成功之后，将 U-boot 镜像烧录进 Micro SD 卡（Micro SD 卡必须已分好区，如未分区，可按照 4.2 节烧录一次系统到 SD 卡即可完成分区）。

注意/dev/mmcblk 分区信息

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

使用 dd 命令将镜像烧录到 SD 卡指定分区：

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

烧录到核心板 eMMC 存储器中指定分区：

注意：以下命令操作在开发板上运行，执行错误或文件写入错误，会导致开发板无法启动。如出现无法启动的情况，请重新烧写开发板即可。

```
root@myir~#: dd if=fip-myb-stm32mp135x-512m-optee.bin of=/dev/mmcblk1p
    1 conv=fdatasync
```

注意/dev/mtd 分区信息

```
/dev/mtdblock0 -- tf-a-myb-stm32mp135x-256m-sdcard.stm32
```



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

烧录到核心板 NAND 存储器中指定分区：

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

5.5. 板载 Kernel 编译与更新

Linux kernel 是个十分庞大的开源内核，被应用在各种发行版操作系统上，Linux kernel 以其可移植性，多种网络协议支持，独立的模块机制，MMU 等诸多丰富特性，使 Linux kernel 能在嵌入式系统中被广泛采用。

同时 STM32MP1 也支持 Linux 内核，并被添加到了内核主线，将得到长期稳定的更新，可查看内核主线了解最新消息 <https://www.kernel.org/>，MYD-YF13X 使用 ST 开源社区版内核移植，最新支持 Linux kernel 5.15.67 版本。

5.5.1. 在独立的交叉编译环境下编译 Kernel

1) 获取 kernel 源代码



```
PC$ cd /home/work
PC$ tar -jxvf MYiR-STM32-kernel.tar.bz2
PC$ cd MYiR-STM32-kernel
```

➤ 源代码目录: myir-st-linux

```
drwxr-xr-x 26 root root 4096 Aug 31 04:50 myir-st-linux
```

- 进入内核源码目录

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

- 创建输出文件夹 build

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

- ## ● 配置内核

```
PC$ make ARCH=arm O="$PWD/../build" myir stm32mp135x defconfig
```

如需配置内核或者想打开内核某一个驱动功能也可使用如下方式。

```
PC$ cd build
```

PC\$ make menuconfig

.config - Linux/arm 5.15.67 Kernel Configuration

```
Linux/arm 5.15.67 Kernel Configuration ↵
Arrow keys navigate the menu. <Enter> selects submenus ---- (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press | <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

[*]
┌─┐
│   General setup ----
│   *- Patch physical to virtual translations at runtime
│     System Type ----
│     Bus support ----
│     Kernel Features ----
│     Boot options ----
│     CPU Power Management ----
└─┘
└─┘
└─┘

<Select>    < Exit >    < Help >    < Save >    < Load >
```



图 4-4. 内核配置界面

3) 编译内核

- 加载 SDK 环境变量

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

- 配置内核

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

- 编译内核

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

耐心等待编译完成.

- 生成输出文件

```
PC$ make ARCH=arm INSTALL_MOD_PATH="$PWD/../build/install_artifact" mod  
ules_install O="$PWD/../build"  
PC$ mkdir -p $PWD/../build/install_artifact/boot/  
PC$ cp $PWD/../build/arch/arm/boot/ulmage $PWD/../build/install_artifact/boot/  
PC$ cp $PWD/../build/arch/arm/boot/dts/my*.dtb $PWD/../build/install_artifact/b  
oot/
```

这样就会在 build/install_artifact 看到编译输出文件，其中 build/install_artifact/lib 目录存放的是内核模块文件，build/install_artifact/boot 目录存放的是设备树与内核镜像文件。

由于上述配置项较多，可将所有命令写到一个脚本里，通过运行脚本完成整个配置与编译，可参看源码目录的 build 文件。

4) 生成镜像

表 5-5. 烧录文件列表:

类型	烧录文件	烧录文件目录
内核文件	ulmage	build/install_artifact/boot
设备树	myb-stm32mp135x-256m.dtb	build/install_artifact/boot



	myb-stm32mp135x-512m.dtb	
内核模块	modules	build/install_artifact/lib/modules

表 5-6. 设备树列表：

设备树	支持核心板	备注
myb-stm32mp135x-512m.dtb	MYC-YF135-4E512D-100-I	Flash 为 EMMC
myb-stm32mp135x-256m.dtb	MYC-YF135-256N256D-100-I	Flash 为 NAND

5.5.2. 在 Yocto 项目下编译 Kernel

将修改后的源代码提交到 github 仓库，同时修改元层的对应源代码的 commit 值。在 github 仓库下找到最新的 commit 值，并将值复制。

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

修改 layers/meta-myr-st/recipes-kernel/linux/linux-myr_5.15.bb 下的 SRCREV 值。

```
SUMMARY = "Linux STM32MP Kernel"
SECTION = "kernel"
LICENSE = "GPL-2.0-only"
#LIC_FILES_CHKSUM = "file://COPYING;md5=bbea815ee2795b2f4230826c0c6b8814"
LIC_FILES_CHKSUM = "file://COPYING;md5=6bc538ed5bd9a7fc9398086aedcd7e46"

include linux-myr.inc

LINUX_VERSION = "5.15"
LINUX_SUBVERSION = "67"
```



```

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

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

LINUX_TARGET = "stm32mp"
LINUX_RELEASE = "r2"

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

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

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

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

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

```



```

SRCREV:class-devupstream = "9030618a5710463c8b96b3a8523ace9075fde00d"

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

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

# -----
# Configure archiver use
# -----
include ${@oe.utils.ifelse(d.getVar('ST_ARCHIVER_ENABLE') == '1', 'linux-myr-arc
hiver.inc', '')}

# -----
# Defconfig
#
KERNEL_DEFCONFIG      = "myir_stm32mp135x_defconfig"
KERNEL_CONFIG_FRAGMENTS = "${@bb.utils.contains('KERNEL_DEFCONFIG', 'def
config', '${S}/arch/arm/configs/fragment-01-multiv7_cleanup.config', '', d)}"
KERNEL_CONFIG_FRAGMENTS += "${@bb.utils.contains('KERNEL_DEFCONFIG', 'd
efconfig', '${S}/arch/arm/configs/fragment-02-multiv7_addons.config', '', d)}"
#KERNEL_CONFIG_FRAGMENTS += "${@bb.utils.contains('DISTRO_FEATURES', 'sy
stemd', '${WORKDIR}/fragments/${LINUX_VERSION}/fragment-03-systemd.config',
'', d)} "
#KERNEL_CONFIG_FRAGMENTS += "${WORKDIR}/fragments/${LINUX_VERSION}/f
ragment-04-modules.config"

```



```
#KERNEL_CONFIG_FRAGMENTS += "${@oe.utils.ifelse(d.getVar('KERNEL_SIGN_EN  
ABLE') == '1', '${WORKDIR}/fragments/${LINUX_VERSION}/fragment-05-signature.  
config','')} "  
#KERNEL_CONFIG_FRAGMENTS += "${@bb.utils.contains('MACHINE_FEATURES', '  
nosmp', '${WORKDIR}/fragments/${LINUX_VERSION}/fragment-06-smp.config', '  
d')) "
```

修改完成后，可在构建目录下单独使用“bitbake linux-myr”命令构建 kernel 或者按照 3.3 章重新构建这个系统。

5.5.3. 如何单独更新 Kernel

编译成功之后，将 ulmage 和 dtb 文件可通过以太网，USB otg, U 盘等传输介质传输到开发板的/boot 分区下（SD 卡启动模式）即可完成更新，内核模块文件传输到 root 分区的/lib/modules 路径下。当传输完成后执行 sync & reboot 完成同步和重启。

1) 通过以太网更新

网络更新方式，实质上是将需要更新的文件传输到/boot 替换掉原有的文件，以有线以太网为例。

将编译的服务器与开发板同时连接到同一子网中，如开发板以太网 IP 为：192.168.30.100。编译服务器 IP 为：192.168.30.101。

在生成输出文件下 build/install_artifact/使用 scp 方式传输需要更新的文件。

更新 ulmage 与设备树：

```
root@myir:~# mount /dev/mmcblk1p6 /mnt  
PC$ scp -r boot/* root@192.168.30.100:/mnt/
```

删除编译生成文件 build/install_artifact/lib 下的软连接文件：

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

更新内核 modules：

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

2) USB OTG 传输更新



- USB 虚拟网络更新

用 Type-C 将 USB OTG 接口与编译服务器直接连接。开发板上 Linux 系统默认设置 USB OTG 从设备为模拟网卡设备。IP 地址默认为：192.168.7.1

在生成输出文件下 build/install_artifact/使用 scp 方式传输需要更新的文件。

更新 ulmage 与设备树：

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

删除编译生成文件 build/install_artifact/lib 下的软链接文件：

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

更新内核 modules：

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

- 使用 ums 更新

UMS 名为 USB 大容量存储器，将开发板的模拟成大容量的储存器。可使用这个来更新各个分区的文件。

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

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

将 USB otg 与编译服务器直接连接，开发板进入 uboot 命令行终端（uboot 计时状态输入任意键），输入 ums 0 mmc 0(SD card)，编译服务器将识别出 MMC0 上的分区信息。

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

```
STM32MP> ums 0 mmc 0
```

```
UMS: LUN 0, dev mmc 0, hwpart 0, sector 0x0, count 0x3b72400
```

编译服务器将自动挂载到 media 分区下，即可识别出开发板启动设备的分区信息：

```
PC$ ./media/nene# ls -all
```

```
total 15
```

```
drwxr-x---+ 6 root root 4096 Oct 15 19:37 .
```

```
drwxr-xr-x 4 root root 4096 Aug 13 20:57 ..
```



```
drwxr-xr-x  5 root root 1024 Oct 13 18:53 bootfs
drwxr-xr-x 21 root root 4096 Feb  7 2020 rootfs
drwxr-xr-x  7 root root 1024 Sep 30 01:53 userfs
drwxr-xr-x  4 root root 1024 Sep 30 01:51 vendorfs
```

在生成输出文件下 build/install_artifact/使用 cp 方式传输需要更新的文件。

更新 ulmage 与设备树：

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

删除编译生成文件 build/install_artifact/lib 下的软连接文件：

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

更新内核 modules：

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

注：\$USER 为编译服务器的用户名，media 为挂载目录

3) 通过 SD 卡直接更新

将启动的 SD 卡通过读卡器等介质，连接到编译服务器。编译服务器将自动挂载到 media 目录（有些系统无法自动挂载，及需要手动挂载 bootfs 分区）。如下

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

在生成输出文件下 build/install_artifact/使用 cp 方式传输需要更新的文件。

更新 ulmage 与设备树：

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



删除编译生成文件 build/install_artifact/lib 下的软连接文件：

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

更新内核 modules：

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

注：\$USER 为编译服务器的用户名，media 为挂载目录



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-YF13X 开发板提供了哪些资源，具体的信息可以查看《MYD-YF13X SDK 发布说明》。除此之外用户还需要对 CPU 的芯片手册，以及 MYC-YF13X 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 TF-a 和 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-YF13X 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-YF13X 设备树列表

项目	设备树	说明
Tf-a	myb-stm32mp135x-base.dts	资源设备树
	myb-stm32mp135x-256m.dts	核心板设备描述
	myb-stm32mp135x-256m-fw-config.dts	
	myb-stm32mp135x-512m.dts	
	myb-stm32mp135x-512m-fw-config.dts	
	stm32mp13-ddr3-1x2Gb-1066-binF.dtsi	单片 256MB DDR3 描述
	stm32mp13-ddr3-1x4Gb-1066-binF.dtsi	单片 512MB DDR3 描述
U-boot	myb-stm32mp135x-base.dts	资源设备树
	myb-stm32mp135x-256m.dts	核心板设备描述
	myb-stm32mp135x-256m-u-boot.dtsi	
	myb-stm32mp135x-512m.dts	
	myb-stm32mp135x-512m-u-boot.dtsi	
	stm32mp13-ddr3-1x2Gb-1066-binF.dtsi	单片 256MB DDR3 描述
	stm32mp13-ddr3-1x4Gb-1066-binF.dtsi	单片 512MB DDR3 描述
	myb-stm32mp13-pinctrl.dtsi	stm32mp135 系列处理器的 pin 脚定义
Kernel	myir_stm32mp135x_defconfig	内核配置文件



	myb-stm32mp135f.dts	资源设备树
	myb-stm32mp135x-base.dtsi	
	myb-stm32mp135x-256m.dts	256MB DDR3 的设备树
	myb-stm32mp135x-512m.dts	512MB DDR3 的设备树
	myb-stm32mp13-pinctrl.dtsi	MYD-YF13X 开发板 pin 脚定义

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-YF13X 的设备树，可在当前路径下增加自定义设备树,如：myb-stm32mp135f-xxx.dts

```
// Path: arch/arm/boot/dts
-rwxrwxr-x 1 nene nene 14761 5月 7 11:56 myb-stm32mp135f.dts
-rwxrwxr-x 1 nene nene 1622 5月 7 11:56 myb-stm32mp135x-256m.dts
-rwxrwxr-x 1 nene nene 869 5月 7 11:56 myb-stm32mp135x-512m.dts
-rwxrwxr-x 1 nene nene 14332 5月 7 11:56 myb-stm32mp135x-base.dtsi
-rwxrwxr-x 1 nene nene 17899 5月 7 11:56 myb-stm32mp13-pinctrl.dtsi
```

我们将 MYC-YF13X 核心板相关的资源编写进 stm32mp135.dts 以及 myb-stm32mp135x-base.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+ OR BSD-3-Clause)
/*
 * Copyright (C) MYiR 2022 - All Rights Reserved
 * Author: Alexhu <fan.hu@myirtech.com> for MYiR.
 */
```



```
/dts-v1/;

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

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

    aliases {
        ethernet0 = &eth1;
        ethernet1 = &eth2;
        serial0 = &uart4;
        serial1 = &uart7;
        serial2 = &uart5;
    };

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

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。



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

增加完成后即可增加设备驱动的板载描述，通过 5.5 节的方法编译生成设备树 dtb 文件 myb-stm32mp135f-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。

STM32MP1 封装大量的外设控制器，这些外设控制器与外部设备交互一般是通过控制 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。

STM32MPU 的 GPIO 管脚配置方法一般使用 STM32CubeMX 配置或使用 Datasheet 查表的方式来配置。

1) STM32CubeMX 配置方法

目前 ST 已经将 STM32MPU 系列 CPU 添加进 STM32CubeMX，我们也可以用此工具来配置 TF-A，U-boot 以及 Kernel 的 GPIO 功能设备树和外设时钟。本节不重点讲解其使用方法，您可以通过官方网站获取详细的开发指导说明。

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

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

2) 查询手册方式

GPIO 的配置可通过米尔整理的 Datasheet 找到描述文件（01_Documents\Datasheet\STM32MP135DAF7.pdf）与核心板引脚清单（01_Documents\HardwareFiles\MYC-YF135 Pin List V1.0），示例如下：

- 通用计算方法

$((\text{port} * 16 + \text{line}) \ll 8) | \text{function}$



- port: The gpio port index (PA = 0, PB = 1, ..., PK = 11)
- line: The line offset within the port (PA0 = 0, PA1 = 1, ..., PA15 = 15)
- function: The function number, can be:
 - * 0 : GPIO
 - * 1 : Alternate Function 0
 - * 2 : Alternate Function 1
 - * 3 : Alternate Function 2
 - * ...
 - * 16 : Alternate Function 15
 - * 17 : Analog
 - * 18 : Reserved

其中 0 表示通用数字 GPIO；1-15 表示复用功能 AF0-AF15；Analog 表示模拟信号；Reserved 表示保留功能。

● 配置 pinmux

下面以扩展接口上的 GPIOF14 为例，说明管脚功能的配置方法。

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

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

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

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



```
};
```

6.2.2. 设备树中引用 GPIO

1) 配置功能管脚为 GPIO 功能实例

此实例使用 PF14 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。此示例还可以给控制外部设备的复位，电源等控制功能提供参考。

只需在设备树里增加节点即可。

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

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 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

uboot 可以直接使用命令来控制 GPIO 的设置。如设置 GPIOF14，可使用下列命令。

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

6.3.2. 内核驱动中使用 GPIO 管脚

1) 独立 IO 驱动的使用



在 6.2.2 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 PF14 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 *offset)
{
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    return 0;
}

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

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

    gpiod_set_value(gpioctr_gpio, status);

    return 1;
}

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

    return 0;
}

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




```

        return 0;
    }

/* 定义自己的 file_operations 结构体*/
static struct file_operations gpiocr_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: gpiocr-gpios=<...>; */
    gpiocr_gpio = gpiod_get(&pdev->dev, "gpiocr", 0);
    if (IS_ERR(gpiocr_gpio)) {
        dev_err(&pdev->dev, "Failed to get GPIO for led\n");
        return PTR_ERR(gpiocr_gpio);
    }

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

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

```



```

        gpiod_put(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_gpiocr",
        .of_match_table = myir_gpiocr,
    },
};

/* 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) 驱动示例将直接配置进内核



在内核源代码的 sample 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 myir_stm32mp135x_defconfig。

在 Kconfig 文件中添加：

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

在 Makefile 文件中添加：

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

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

在 myir_stm32mp135x_defconfig 文件中添加：

```
//linux/arch/arm/configs/myir_stm32mp135x_defconfig
CONFIG_SAMPLES=y
CONFIG_SAMPLE_GPIO=y
CONFIG_SAMPLE_RPMSG_CLIENT=m
```

按照 5.5.3 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/nene/MYiR-stm32-linux/build/

obj-m += gpioctr.o
```



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

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

# 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。

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

执行 make 命令，即可生成 gpioctr.ko 驱动模块文件。

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

编译成功之后，将 gpioctr.ko 文件可通过以太网，USB otg, U 盘等传输介质传输到开发板的/lib/modules 目录下即可使用 insmod 命令加载驱动。

不同的外部设备各自具有独立的驱动代码和架构实现，在对不同外设驱动修改，调试时，需要遵守各自的驱动框架。如触摸屏，键盘等需要使用 input 驱动架构；ADC 与 DAC



使用 IIO 架构，显示设备使用 DRM 驱动架构等等，本节不对所有驱动开发做具体的讲解。

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实现的，本节不做详细的说明，可查看《MYD-YF13X_Linux 软件评估指南》第 3.1 节描述。

2) 库函数实现管脚控制

从 Linux 4.8 版本开始，Linux 引入了新的 gpio 操作方式，GPIO 字符设备。不再使用以前 SYSFS 方式在"/sys/class/gpio"目录下来操作 GPIO，而是，基于"文件描述符"的字符设备，每个 GPIO 组在"/dev"下有一个对应的 gpiochip 文件，例如"/dev/gpiochip0 对应 GPIOA, /dev/gpiochip1 对应 GPIOB"等等。

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 PF14 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 环境变量到当前 shell:

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-n
eon-vfpv4-ostl-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@myir:~# ./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_gpiotctr0 on
 * ./gpiotest /dev/myir_gpiotctr0 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 环境变量到当前 shell:

```
PC$ source /opt/st/myir-yf13x/4.0.4-snapshot/environment-setup-cortexa7t2hf-n
eon-vfpv4-ostl-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 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@myir:~# gpiotest /dev/myir_gpiotctr0 on
root@myir:~# gpiotest /dev/myir_gpiotctr0 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 会自动的判别原始文件是否经过了变动，从而自动重新编译更改的源代码。

下列将以一个实际的示例（在 MYD-YF13X 开发板上实现按键控制 LED 灯开关）来讲述 Makefile 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} -I . -c key_led.c
```

- \$(notdir \$(path)): 表示把 path 目录去掉路径名, 只留当前目录名, 比如当前 Makefile 目录为 /home/nene/key_led, 执行完就变为 TARGET = key_led
- \$(patsubst pattern, replacement, text): 用 replacement 替换 text 中符合格式 "pattern" 的字符, 如 \$(patsubst %c, %o, \$(shell ls *.c)), 表示先列出当前目录后缀为.c 的文件, 然后换成后缀为.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/devices/platform/leds/leds/blue: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/devices/platform/leds/leds/blue:heart
tbeat/brightness");//led on
        else
            system("echo 0 > /sys/devices/platform/leds/leds/blue:
heartbeat/brightness");//led off
    }

}

}
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:

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

执行 make:

```
PC$ 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:~# key_led_test /dev/input/event0 noblock
```

在 MYD-YF13X 开发板上按 S2 键即可控制 LED（D3）灯开关。

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. 基于 Qt 的应用

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-YF13X 使用 Qt 5.15 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 架构程序。

1) QtCreator 安装与配置

从 QT 官网或 MYIR 官方包获得 qtcreator 安装包 QT 官网下载：http://download.qt.io/development_releases/qtcreator/4.1/4.1.0-rc1/。

QtCreator 安装包是一个二进制程序，直接执行就可以完成安装 ./qt-creator-opensource-linux-x86_64-4.1.0-rc1.run 即可，如需获得安装与配置详情请查看《MYD-YF13X_QT 及 MEasy HMI2.0 软件开发指南》(暂未发布)或从 QtCreator 官方网站获得更多开发指导 <https://www.qt.io/product/development-tools>。

2) MEasy HMI2.0 编译和运行

MEasy HMI 2.0 是深圳市米尔科技有限公司开发的一套基于 QT5 的人机界面框架。项目采用 QML 与 C++ 混合编程，使用 QML 高效便捷地构建 UI，而 C++ 则用来实现业务逻辑和复杂算法。

在米尔的软件发布包里可获得 MEasy HMI2.0 项目源代码 “MYD-YF13X-2023xxx\04_Sources\mxapp2.tar.gz”。可通过 Qtcreator 进行加载编译，远程调试等。



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-openstlinuxweston-myd-yf13x-emmc$ bitbake -s | grep proftpd
proftpd                               :1.3.7c-r0
```

注：这里可以看到 Yocto 项目中已经存在 proftpd 配方，版本为 1.3.7c-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-myr 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，可以参照以下步骤添加普通用户，然后重新构建镜像文件：

在 conf/local.conf 中添加如下内容：

```
INHERIT += "extrausers"
```



```
EXTRA_USERS_PARAMS += "\
    usermod -p 'FPSDUIQTQvUn2' root; \
    useradd ucas; \
    usermod -p 'FPSDUIQTQvUn2' tester; \
"
```

● 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_1.3.7c.bb
```

1 directory, 8 files

- proftpd_1.3.7c.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_1.3.7c.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:

```
SUMMARY = "Secure and configurable FTP server"
SECTION = "net"
HOMEPAGE = "http://www.proftpd.org"
LICENSE = "GPL-2.0-or-later"
LIC_FILES_CHKSUM = "file://COPYING;md5=fb0d1484d11915fa88a6a7702f1dc184"
"

SRCREV = "75aa739805a6e05eeb31189934a3d324e7862962"
BRANCH = "1.3.7"

SRC_URI = "git://github.com/proftpd/proftpd.git;branch=${BRANCH};protocol=ht
tps \
    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"

# proftpd uses libltdl which currently makes configuring using
# autotools.bbclass a pain...
do_configure () {
```



```

install -m 0755 ${STAGING_DATADIR_NATIVE}/gnu-config/config.guess ${S}
install -m 0755 ${STAGING_DATADIR_NATIVE}/gnu-config/config.sub ${S}
oe_runconf
}

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|-ffile-prefix-map=[^ ]*||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

```



```
}
```

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_1.3.7c.bb 配方通过继承 systemd.class (具体内容查看 [layers/openembedded-core/meta/classes/systemd.bbclass](https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#selecting-an-initialization-manager)) 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, 运行于 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。



8. 参考资料

- Linux kernel 开源社区
<https://www.kernel.org/>
- STM32MPU 开发社区
https://wiki.st.com/stm32mpu/wiki/Development_zone
- Yocto 开发指导
<https://www.yoctoproject.org/>
- Yocto 项目 BSP 开发指南
<https://docs.yoctoproject.org/4.0.9/bsp-guide/index.html>
- Yocto 项目 Linux 内核开发手册
<https://docs.yoctoproject.org/4.0.9/kernel-dev/index.html>



附录一 联系我们

深圳总部

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

负责区域：广东、广西、海南、重庆、云南、贵州、四川、西藏、香港、澳门

传真：0755-25532724 电话：0755-25622735

武汉研发中心

地址：武汉东湖新技术开发区关南园一路 20 号当代科技园 4 号楼 1601 号

电话：027-59621648

华东地区

地址：上海市浦东新区金吉路 778 号浦发江程广场 1 号楼 805 室

负责区域：上海、福建、浙江、江苏、安徽、山东

传真：021-62087085 电话：021-62087019

华北地区

地址：北京市大兴区荣华中路 8 号院力宝广场 10 号楼 901 室

负责区域：辽宁、吉林、黑龙江、北京、天津、河北、山西、内蒙古、湖北、湖南、江西、河南、陕西、甘肃、宁夏、青海、新疆

传真：010-64125474 电话：010-84675491

销售联系方式

网址：www.myir.cn

邮箱：sales.cn@myirtech.com

技术支持联系方式

电话：0755-22316235（深圳）027-59621647/027-59621648（武汉）

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。



附录二 售后服务与技术支持

凡是通过米尔电子直接购买或经米尔电子授权的正规代理商处购买的米尔电子全系列产品，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，以及米尔电子开发的部分软件源代码
- 6、可直接从米尔电子购买主要芯片样品，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，即成为米尔电子永久客户，享有再次购买米尔电子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，请先致电米尔电子客服部，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。

