

MYD-YM62X Linux 软件开发指南



文件状态： [] 草稿 [√] 正式发布	文件标识：	MYIR-MYD-YM62X-SW-DG-ZH-L6.1.46
	当前版本：	V1.0[文档]
	作 者：	Miller
	创建日期：	2023-09-20
	最近更新：	2023-09-20

版本历史

版本	作者	参与者	日期	备注
V1.0[文档]	Miller		20220920	初始版本: uboot 2023.04, kernel 6.1.46, arago 2023.04



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. 安装编译链	- 10 -
3. 使用 Yocto 构建开发板镜像	- 13 -
3.1. 简介	- 13 -
3.2. 获取源码	- 14 -
3.2.1. 从光盘镜像获取源码压缩包	- 14 -
3.2.2. 通过 github 获取	- 14 -
3.3. 快速编译开发板镜像	- 15 -
3.4. 构建 SD 卡烧录器镜像	- 21 -
3.5. 构建 SDK	- 22 -
4. 如何烧录系统镜像	- 23 -
4.1. 制作 TF 卡启动器	- 23 -
4.2. 制作 TF 卡烧录器	- 26 -
5. 如何修改板级支持包	- 27 -
5.1. meta-bsp 层介绍	- 27 -
5.2. 板级支持包介绍	- 29 -
5.3. 板载 BootLoader 编译与更新	- 30 -
5.3.1. 如何在独立环境下编译 bootloader	- 30 -
5.3.2. 在 Yocto 项目下编译 u-boot	- 32 -
5.3.3. 如何单独更新 bootloader	- 34 -
5.4. 板载 Kernel 编译与更新	- 36 -



5.4.1. 编译 Linux	- 36 -
5.4.2. 在独立的交叉编译环境下编译 Kernel	- 36 -
5.4.3. 在 Yocto 项目下编译 Kernel	- 37 -
5.4.4. 如何单独更新 Kernel 和设备树	- 39 -
6. 如何适配您的硬件平台	- 41 -
6.1. 如何创建自己的 machine	- 41 -
6.1.1. 在 Yocto 创建一个板子配置	- 41 -
6.1.2. 在 uboot 创建板子配置文件	- 42 -
6.2. 如何创建您的设备树	- 45 -
6.2.1. 板载设备树层级的介绍	- 46 -
6.2.2. 设备树的添加	- 47 -
6.3. 如何根据您的硬件配置 CPU 功能管脚	- 48 -
6.3.1. GPIO 管脚配置的方法	- 48 -
6.4. 如何使用自己配置的管脚	- 50 -
6.4.1. U-boot 中使用 GPIO 管脚	- 50 -
6.4.2. 内核驱动中使用 GPIO 管脚	- 52 -
6.4.3. 用户空间使用 GPIO 管脚	- 58 -
7. 如何添加您的应用	- 61 -
7.1. 基于 Makefile 的应用	- 61 -
7.2. 基于 Qt 的应用	- 65 -
7.3. 应用程序开机自启动	- 65 -
8. 参考资料	- 72 -
附录一 联系我们	- 73 -
附录二 售后服务与技术支持	- 75 -



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 不仅仅是一个制作文件系统工具，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-YM62X 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，就可以很快地将系统移植到基于 MYD-YM62X 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。

1.1. 软件资源

MYD-YM62X 搭载基于 Linux 6.1.46 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，ATF 源代码，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-YM62X 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，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-YM62X 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-YM62X 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. 开发主机环境

本章节将介绍如何搭建 MYD-YM62X 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。并能快速地搭建相关开发环境，为后面的开发和调试做准备。AM62X 系列处理器是一个多核异构的处理器，其包含：

- 1-4 个 ARM Cortex A53 内核，可以运行嵌入式 Linux 系统，使用嵌入式 Linux 系统的开发工具。
- 1 个 ARM Cortex M4 内核，可以运行裸机代码或其他实时操作系统（RTOS），使用 TI 官方提供的 Cortex M4 软件开发工具。

1) 主机软硬件要求

● 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，500GB 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。

● 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 **Ubuntu20.04 64bit** 版本，后续开发也是以此系统为例进行介绍。

2) 主机环境配置

当安装完 ubuntu 20.04 64bit 系统后，可以先进行适当的配置，为后续开发做准备。

● 设置 root 密码

```
myir@system1:~$ sudo passwd root
```



● 更换清华源

Ubuntu 安装工具常用命令就是 apt-get，但是安装完系统后，默认的源为国外服务器，下载会很慢，所以需要更换为清华源。打开清华大学开源软件镜像站：<https://mirrors.tuna.tsinghua.edu.cn/help/ubuntu/>，选择你的 ubuntu 版本，如 20.04，Ubuntu 的软件源配置文件是 /etc/apt/sources.list。将系统自带的该文件做个备份，将该文件替换为下面内容，即可使用 TUNA 的软件源镜像。这个不是必须更新，但是更新后安装包会比较快速。

```
# 默认注释了源码镜像以提高 apt update 速度，如有需要可自行取消注释 deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal main restricted universe multiverse# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal main restricted universe multiversedeb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-updates main restricted universe multiverse# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-updates main restricted universe multiversedeb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-backports main restricted universe multiverse# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-backports main restricted universe multiverse# deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-security main restricted universe multiverse# # deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-security main restricted universe multiversedeb http://security.ubuntu.com/ubuntu/ focal-security main restricted universe multiverse# deb-src http://security.ubuntu.com/ubuntu/ focal-security main restricted universe multiverse# 预发布软件源，不建议启用# deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-proposed main restricted universe multiverse# # deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ focal-proposed main restricted universe multiverse
```

● 安装 ssh

安装完 ssh 服务后，可以在 window 环境下，用 sercureCRT 工具 ssh2 的方式连接到 ubuntu 进行后续开发。

```
myir@system1:~$ sudo apt-get install openssh-server
```

给用户生成密钥：

```
myir@system1:~$ su user
```

```
myir@system1:~$ ssh-keygen -t rsa
```



- 配置 samba

samba 可以直接在 window 下以文件夹形式访问 ubuntu 的内容，读写更方便。安装 samba:

```
myir@system1:~$ apt-get install samba
```

在/etc/samba/smb.conf 中加入用户配置，如 linux 用户名为 "duxy"，如下配置：

```
[duxy]
path = /home/duxy
valid users = duxy
browseable = yes
public = yes
writable = yes
```

创建账号并设置密码：

```
myir@system1:~$ sudo smbpasswd -a duxy
New SMB password:
Retype new SMB password:
Added user duxy.
```

/etc/init.d/smbd restart 重启 samba 服务：

```
myir@system1:~$ /etc/init.d/smbd restart
[ ok ] Restarting smbd (via systemctl): smbd.service.
```

- 配置 git

```
myir@system1:~$ git config --global user.name "user"
myir@system1:~$ git config --global user.email "email"
myir@system1:~$ git config --list
```

- 安装 SDK 必要工具

```
myir@system1:~$ sudo apt-get update
myir@system1:~$ sudo apt-get -f -y install git build-essential \
diffstat texinfo gawk chrpath socat doxygen dos2unix python3 bison \
flex libssl-dev u-boot-tools mono-devel mono-complete curl lrzsz lzop \
python3-distutils pseudo python3-sphinx g++-multilib bc python3-pip \
libc6-dev-i386 jq git-lfs pigz zstd liblz4-tool cpio file autoconf automake \
xinetd tftpd nfs-kernel-server minicom libncurses5-dev dos2unix screen \
zstd lz4 python3-pyelftools python3-setuptools swig repo
```



```
myir@system1:~$ sudo pip3 install jsonschema pyelftools
```



2.2. 安装编译链

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两个编译好的 SDK 包，位于：03-Tools/Toolchains/，两个 SDK 文件功能描述如下表：

表 2-1.编译工具链

工具链文件名	描述
./arago-2023.04-toolchain-2023.04.sh	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

这里先介绍 SDK 的安装步骤，如下：

- 拷贝 SDK 到 Linux 目录

将 SDK 压缩包拷贝到 Ubuntu 下的用户工作目录，得到安装脚本文件，如下：

```
arago-2023.04-toolchain-2023.04.sh
```

- 执行安装脚本

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。本例程把 qt 工具链安装在 /media/home/wujl/AM62/tool_chain/myir 目录，如下：

```
myir@system1:~/AM62/tool_chain$ ./arago-2023.04-toolchain-2023.04.sh
Arago SDK installer version 2023.04
=====
Enter target directory for SDK (default: /opt/arago-2023.04): /media/home/wujl/AM62/tool_chain/myir
You are about to install the SDK to "/media/home/wujl/AM62/tool_chain/myir". Proceed [Y/n]? y
Extracting SDK
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.

初始化环境变量:

```
wujl@system1:~/AM62/tool_chain$ source /media/home/wujl/AM62/tool_chain/myir/environment-setup-aarch64-oe-linux
```

● 测试 SDK

安装完成后, 使用以下命令加载环境变量到当前 shell, 测试 SDK 是否完成:

```
wujl@system1:~/AM62/tool_chain$ $CC -v
Using built-in specs.
COLLECT_GCC=aarch64-oe-linux-gcc
COLLECT_LTO_WRAPPER=/media/home/wujl/AM62/tool_chain/myir/sysroots/x86_64-arago-linux/usr/libexec/aarch64-oe-linux/gcc/aarch64-oe-linux/11.3.0/lto-wrapper
Target: aarch64-oe-linux
Configured with: ../../../../work-shared/gcc-11.3.0-r0/gcc-11.3.0/configure --build=x86_64-linux --host=x86_64-arago-linux --target=aarch64-oe-linux --prefix=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr --exec_prefix=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr --bindir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr/bin/aarch64-oe-linux --sbindir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr/bin/aarch64-oe-linux --libexecdir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr/libexec/aarch64-oe-linux --datadir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr/share --sysconfdir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/etc --sharedstatedir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/com --localstatedir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/var --libdir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr/lib/aarch64-oe-linux --includedir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr/include --oldincludedir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr/include --infodir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr/share/info --
```



```
mandir=/usr/local/oe-sdk-hardcoded-buildpath/sysroots/x86_64-arago-linux/usr/  
share/man --disable-silent-rules --disable-dependency-tracking --with-libtool-sys  
root=/media/home/wujl/AM62/tisdk/build/arago-tmp-default-glibc/work/x86_64  
-nativesdk-arago-linux/gcc-cross-canadian-aarch64/11.3.0-r0/recipe-sysroot --wit  
h-gnu-ld --enable-shared --enable-languages=c,c++ --enable-threads=posix --e  
nable-multilib --enable-c99 --enable-long-long --enable-symvers=gnu --enable-li  
bstdcxx-pch --program-prefix=aarch64-oe-linux- --without-local-prefix --disable-  
install-libiberty --disable-libssp --enable-libitm --enable-lto --disable-bootstrap --  
with-system-zlib --with-linker-hash-style=gnu --enable-linker-build-id --with-ppl  
=no --with-cloog=no --enable-checking=release --enable-headers=c_global --w  
ithout-isl --with-gxx-include-dir=/not/exist/usr/include/c++/11.3.0 --with-build-ti  
me-tools=/media/home/wujl/AM62/tisdk/build/arago-tmp-default-glibc/work/x8  
6_64-nativesdk-arago-linux/gcc-cross-canadian-aarch64/11.3.0-r0/recipe-sysroot-  
native/usr/aarch64-oe-linux/bin --with-sysroot=/not/exist --with-build-sysroot=/  
media/home/wujl/AM62/tisdk/build/arago-tmp-default-glibc/work/x86_64-native  
sdk-arago-linux/gcc-cross-canadian-aarch64/11.3.0-r0/recipe-sysroot --enable-st  
andard-branch-protection --with-plugin-ld=ld --enable-poison-system-directorie  
s --enable-nls --with-glibc-version=2.28 --enable-initfini-array --enable-__cxa_at  
exit
```

Thread model: posix

Supported LTO compression algorithms: zlib zstd

gcc version 11.3.0 (GCC)

同样方法请自行安装用于基础开发的工具链。安装两个工具链的时候，请指定不同目录，请勿使用相同目录，否则会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。

MYD-YM62X 提供了符合 Yocto 的配置文件，帮助开发者构建出可烧写在 MYD-YM62X 板上的 Linux 系统镜像。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，不能完全介绍 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人员，希望从 Yocto 构建出符合 MYD-YM62X 系列开发板的文件系统，同时基于它的定制化方法。初次体验使用或无特殊需要的开发者可以直接使用 MYD-YM62X 已经提供的文件系统。

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。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-YM62X 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。米尔光盘镜像可以从如下链接获取：

<http://down.myir-tech.com/MYD-YM62X/>

3.2.1. 从光盘镜像获取源码压缩包

源码包在米尔开发包资料 04-Sources/myd-ym62x-yocto.zip。拷贝压缩包到用户指定目录，如解压 Yocto 源码包到工作目录 myd-ym62x-yocto：

```
myir@system1:~$ mkdir -p myd-ym62x-yocto
myir@system1:~$ cd myd-ym62x-yocto
myir@system1: ~/myd-ym62x-yocto$ unzip myd-ym62x-yocto.zip
```

如果您已经按照本节的内容获取到 Yocto 源码，您可以跳过下面的 3.2.2 章节。

3.2.2. 通过 github 获取

目前 MYD-YM62X 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-YM62X 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：

```
myir@system1:~$ mkdir -p myd-ym62x-yocto
myir@system1:~$ cd myd-ym62x-yocto
myir@system1: ~/myd-ym62x-yocto$ git clone https://github.com/MYIR-TI/oe-layersetup.git -b develop_2023.04
myir@system1: ~/myd-ym62x-yocto$ ls
oe-layersetup
myir@system1: ~/myd-ym62x-yocto$ cd oe-layersetup
myir@system1: ~/myd-ym62x-yocto/oe-layersetup$ ./oe-layertool-setup.sh -f configs/processor-sdk/processor-sdk-09.00.00-myd-am62x-config.txt
```

代码同步成功之后，同样在 myd-ym62x-yocto 目录下得到 04-Sources/myd-ym62x-yocto.zip 源码包一样的目录内容。



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 *build/conf/setenv* 脚本进行环境变量的设置，

1) 查看 Yocto 源码包内容

当在 Ubuntu20.04 中解压了 myd-ym62x-yocto.zip 文件，或者通过 repo 下载了文件后，会生成如下文件，列出 myd-ym62x-yocto 目录内容如下：

```
myir@system1:~$ tree -a -L 1 myd-ym62x-yocto
myd-ym62x-yocto
├── configs
├── .git
├── .gitignore
├── git_retry.sh
├── oe-layertool-setup.sh
├── sample-files
└── sources
4 directories, 3 files
```

2) 执行环境变量设置脚本

脚本设置编译环境的语法如下：

```
myir@system1:~$ cd myd-ym62x-yocto
myir@system1: ~/myd-ym62x-yocto$ ./oe-layertool-setup.sh -f configs/processor
-sdk/processor-sdk-09.00.00-myd-am62x-config.txt
```

配置脚本执行完成后将生成 *build* 目录下，进入 *build* 目录下使能 Ubuntu20.04 的环境变量，如下：

```
myir@system1: ~/myd-ym62x-yocto/build$ cd build
myir@system1: ~/myd-ym62x-yocto/build$ . conf/setenv
```

注意：编译 yocto 需要使用普通用户，不能使用 root 用户

3) 构建 myir-image-full 镜像



这里编译时会有在网络下载第三方源码的 fetch 过程，如果网络不好经常出现 fetch 错误，所以建议将网盘上的 downloads 文件下载下来，在放置在同级目录，如这里是 myd-ym62x-yocto 目录。这里演示编译过程。

- 拷贝 downloads 文件到指定目录

从网盘下载 downloads 文件，首先需要拷贝到对应目录（如：myd-ym62x-yocto），如红色文件所示：

```
myir@system1: ~/myd-ym62x-yocto$ ls
build configs git_retry.sh oe-layertool-setup.sh sample-files sources
downloads
```

downloads 文件也可以直接从网上拉取，但是拉取速度取决于用户的网速，所以一般建议还是用网盘的 downloads 文件。一般网上拉取文件的命令如下所示：

```
myir@system1: ~/myd-ym62x-yocto/build$ bitbake myir-image-full --runall=fetch
```

- 构建完整镜像

download 文件拷贝完成之后，接着执行以下命令构建系统：

```
myir@system1:~/AM62/tisdk/build$ bitbake myir-image-full
NOTE: Started PRServer with DBfile: /media/home/wujl/AM62/tisdk/build/cache/prserv.sqlite3, Address: 127.0.0.1:33603, PID: 1305201
Loading cache: 100% |#####| Time: 0:00:08
Loaded 9740 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:05
Parsing of 6624 .bb files complete (6616 cached, 8 parsed). 9748 targets, 638 skipped, 0 masked, 0 errors.
WARNING: No recipes in default available for:
/media/home/wujl/AM62/tisdk/sources/meta-myr/meta-myr-bsp/recipes-core/packagegroups/packagegroup-myr-qte-toolchain-target.bbappend
/media/home/wujl/AM62/tisdk/sources/meta-arago/meta-arago-distro/recipes-connectivity/linuxptp/linuxptp_3.0.bbappend
```



```
/media/home/wujl/AM62/tisdk/sources/meta-arago/meta-arago-distro/recipes-  
tisdk/ocl-rtos/openc1-examples-rtos_git.bbappend
```

No recipes in k3r5 available for:

```
/media/home/wujl/AM62/tisdk/sources/meta-myir/meta-myir-bsp/recipes-core/  
packagegroups/packagegroup-myir-qte-toolchain-target.bbappend
```

```
/media/home/wujl/AM62/tisdk/sources/meta-arago/meta-arago-distro/recipes-  
connectivity/linuxptp/linuxptp_3.0.bbappend
```

```
/media/home/wujl/AM62/tisdk/sources/meta-arago/meta-arago-distro/recipes-  
tisdk/ocl-rtos/openc1-examples-rtos_git.bbappend
```

NOTE: Resolving any missing task queue dependencies

NOTE: Resolving any missing task queue dependencies

NOTE: Resolving any missing task queue dependencies

NOTE: Resolving any missing task queue dependencies

NOTE: Resolving any missing task queue dependencies

NOTE: Resolving any missing task queue dependencies

Build Configuration (mc:default):

BB_VERSION = "2.0.0"

BUILD_SYS = "x86_64-linux"

NATIVELSBSTRING = "ubuntu-20.04"

TARGET_SYS = "aarch64-oe-linux"

MACHINE = "myd-am62x"

DISTRO = "arago"

DISTRO_VERSION = "2023.04"

TUNE_FEATURES = "aarch64"

TARGET_FPU = ""

meta-edgeai = "HEAD:70a45ea10967e4dd986b5b477179bcc91490f8a7"

meta-processor-sdk = "HEAD:02a8a582ac4b08ea0fac6be8cf10a0d89defb88a"

meta-arago-distro

meta-arago-extras

meta-arago-demos = "HEAD:3750439eca3436a4f4cdc345215abab60261df74"

meta-qt5 = "HEAD:9285c220f39dca57e91aece38f4ff31e90966281"

meta-virtualization = "HEAD:b3b3dbc67504e8cd498d6db202ddcf5a9dd26a9d"



```
meta-networking
meta-python
meta-oe
meta-gnome
meta-filesystems
meta-multimedia    = "HEAD:4af7c61f6e9b1cf3ac9ea2c5ef0d3441ce77d488"
meta-myrir-extras
meta-myrir-bsp      = "master:abfc1530837d5f6cb749de727b6171ece4344e83"
meta-arm
meta-arm-toolchain  = "HEAD:96aad3b29aa7a5ee4df5cf617a6336e5218fa9bd"
meta                = "HEAD:f20a12ead2d5890e88e7f4ce149a777de47edc48"
```

Build Configuration:

```
BB_VERSION          = "2.0.0"
BUILD_SYS           = "x86_64-linux"
NATIVELSBSTRING     = "ubuntu-20.04"
TARGET_SYS          = "arm-oe-linux-gnueabi"
MACHINE             = "myd-am62x-k3r5"
DISTRO              = "arago"
DISTRO_VERSION       = "2023.04"
TUNE_FEATURES        = "arm armv7a vfp thumb callconvention-hard"
TARGET_FPU           = "hard"
meta-edgeai          = "HEAD:70a45ea10967e4dd986b5b477179bcc91490f8a7"
meta-processor-sdk   = "HEAD:02a8a582ac4b08ea0fac6be8cf10a0d89defb88a"
meta-arago-distro
meta-arago-extras
meta-arago-demos     = "HEAD:3750439eca3436a4f4cdc345215abab60261df74"
meta-qt5             = "HEAD:9285c220f39dca57e91aece38f4ff31e90966281"
meta-virtualization = "HEAD:b3b3dbc67504e8cd498d6db202ddcf5a9dd26a9d"
meta-networking
meta-python
meta-oe
```



```
meta-gnome
meta-filesystems
meta-multimedia    = "HEAD:4af7c61f6e9b1cf3ac9ea2c5ef0d3441ce77d488"
meta-myrir-extras
meta-myrir-bsp      = "master:abfc1530837d5f6cb749de727b6171ece4344e83"
meta-arm
meta-arm-toolchain  = "HEAD:96aad3b29aa7a5ee4df5cf617a6336e5218fa9bd"
meta                = "HEAD:f20a12ead2d5890e88e7f4ce149a777de47edc48"
```

```
Initialising tasks: 100% |#####|
#####| Time: 0:00:17
NOTE: Deferring mc:k3r5:/media/home/wujl/AM62/tisdk/sources/meta-myrir/meta-
myr-bsp/recipes-bsp/ti-sci-fw/ti-sci-fw_git.bb:do_populate_lic after /media/hom
e/wujl/AM62/tisdk/sources/meta-myrir/meta-myrir-bsp/recipes-bsp/ti-sci-fw/ti-sci-
fw_git.bb:do_populate_lic
NOTE: Deferring mc:k3r5:/media/home/wujl/AM62/tisdk/sources/meta-myrir/meta-
myr-bsp/recipes-bsp/u-boot/u-boot-ti-staging_2023.04.bb:do_populate_lic after
/media/home/wujl/AM62/tisdk/sources/meta-myrir/meta-myrir-bsp/recipes-bsp/u-
boot/u-boot-ti-staging_2023.04.bb:
```

构建完成之后，编译出来的镜像在“*build/arago-tmp-default-glibc/deploy/images/myd-am62x*”目录。

表 3-1. 镜像文件列表

编译生成文件	描述
myir-image-full-myd-am62x.wic.xz	完整镜像
Image	内核镜像
myd-ym62x-6231.dtb	适用与 myc-ym6231 的 hdmi 显示
myd-ym62x-6252.dtb	适用与 myc-ym6252 的 hdmi 显示
myd-ym62x-6254.dtb	适用与 myc-ym6254 的 hdmi 显示
tispl.bin	Cortex-A53 SPL 镜像，由 Cortex-A53 引导 U-Boot 启动
tiboot3.bin	Cortex-R5F SPL 镜像，由 Cortex-R5F 对 Cortex-A53 进行初始化，并引导 Cortex-A53 启动。



u-boot.img	U-Boot 启动镜像。加载内核启动
------------	--------------------

生成的镜像文件按照 4.1 节更新即可。



3.4. 构建 SD 卡烧录器镜像

为满足生产烧录镜像到 eMMC 的需要，米尔开发了适用于大批量生产的烧录方法。通过 SD 卡中的系统将需要烧录的系统刷写进板载的 eMMC 中。这里已经把烧录器镜像资源整合到 Yocto 项目中，所以用户可以利用 yocto 项目构建出烧录器镜像。

1) 执行环境变量设置脚本

脚本设置编译环境的语法如下：

```
myir@system1: ~/myd-ym62x-yocto$ cd build  
myir@system1: ~/myd-ym62x-yocto/build$ . conf/setenv
```

2) 构建 myir-image-full 镜像

由于 sd 卡烧录器镜像就是把完整系统 (myir-image-full)，拷贝到某些目录，所以构建 sd 卡烧录器镜像之前需要先构建 myir-image-full 镜像，构建详情请参考 3.3 节。

3) 构建 sd 卡烧录器镜像

- 构建镜像

执行以下命令，构建 sd 卡烧录器镜像：

```
myir@system1: ~/myd-ym62x-yocto/build$ bitbake -c cleansstate fac-burn-emmc-full  
myir@system1: ~/myd-ym62x-yocto/build$ bitbake -c cleansstate myir-image-burn  
myir@system1: ~/myd-ym62x-yocto/build$ bitbake fac-burn-emmc-full  
myir@system1: ~/myd-ym62x-yocto/build$ bitbake myir-image-burn
```

生成镜像文件为 *tmp/deploy/images/myd-ym62x/myir-image-burn-myd-ym62x.wic.bz2*，然后按照 4.2 节更新即可。



3.5. 构建 SDK

Yocto 提供可构建出 SDK 工具链功能，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。下面讲解如何构建工具链 SDK。

本节只简单对米尔提供的 SDK 做构建说明，使用如下命令生成 SDK 包：

```
myir@system1:~/am62/tisdk/build$ bitbake -c populate_sdk myir-image-full
```

等待构建完成后，将在 “*bbuild/arago-tmp-default-glibc/deploy/sdk*” 路径下生成 SDK 安装包，安装方法请查看 2.3 节。

```
arago-2023.04-toolchain-2023.04.host.manifest  x86_64-buildtools-nativesdk-standalone-2023.04.host.manifest
arago-2023.04-toolchain-2023.04.sh             x86_64-buildtools-nativesdk-standalone-2023.04.sh
arago-2023.04-toolchain-2023.04.target.manifest x86_64-buildtools-nativesdk-standalone-2023.04.target.manifest
arago-2023.04-toolchain-2023.04.testdata.json  x86_64-buildtools-nativesdk-standalone-2023.04.testdata.json
```



4. 如何烧录系统镜像

米尔公司设计的 MYD-YM62X 系列开发板是基于 TI 公司的 AM62X 系列微处理器，其启动方式多样，所以需要不同的更新工具与方法。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：

- 制作 TF 卡启动器：适用于研发调试，快速启动等场景。
- 制作 TF 卡烧录器：适用于批量生产烧写 emmc。

另外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	SW2[1:4]	SW3[1:3]
ARM 启动	TF Card 启动	0001	001
	eMMC 启动	1001	000

4.1. 制作 TF 卡启动器

1) 准备工作

此步骤均在 Windows 系统下制作。

- TF 卡（不少于 8GB）
- MYD-YM62X 完整镜像
- 制作镜像工具 Win32DiskImager-1.0.0-binary.zip（路径：/03-Tools/tools）

表 4-2. 镜像包列表

镜像名	包名
myir-image-full	myir-image-full-myd-am62x.wic.xz

2) 制作 TF 卡启动（以 myir-image-full 系统为例）

- 解压完整镜像

window 直接右键解压出来。

Win32DiskImager-1.0.0-binary.zip
myir-image-full-myd-am62x.wic.xz



- 将镜像文件写入 Micro SD Card

将 TF 卡使用读卡器插入电脑，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 4-3.工具配置

选择好系统包后点击打开即可，如下图红色箭头所示。

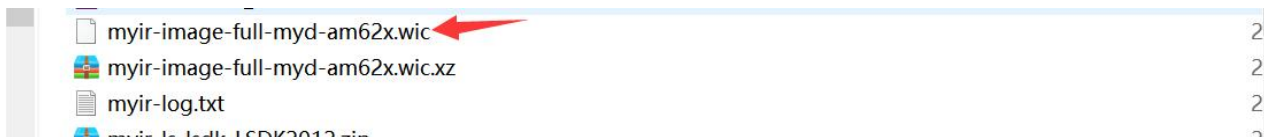


图 4-4.选择镜像

加载完镜像后点击 “Write” 按钮即可，会弹出警告，点击 “Yes” 等待写入完成。

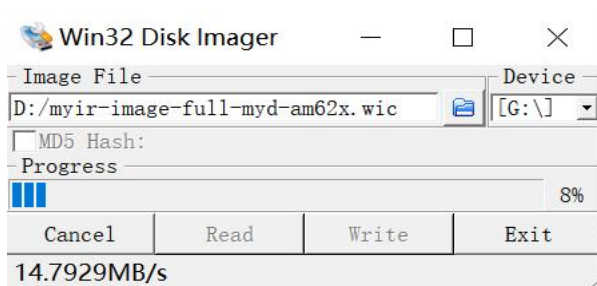


图 4-5.烧写指示

等待写入完成，大约 3-4 分钟完成，此速度取决于 TF 的读写速度。

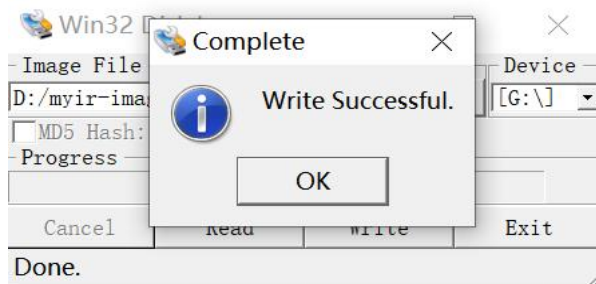


图 4-6.烧写完成



● 检查是否烧写成功

当写入完成后，即可使用此 TF 进行启动，将 TF 插入开发板 TF 卡槽；并将拨码开关调至 TF 卡启动（请参考表 4-1）即可启动系统。



4.2. 制作 TF 卡烧录器

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。

1) 制作 TF 卡烧录器镜像

制作 TF 卡烧录器镜像(myir-image-burn-my-d-am62x.wic)，详情请查看 3.4 节。

2) 将烧录器镜像烧写进 TF 卡

制作 TF 卡启动器即将 sd 卡烧录镜像写进 TF 卡，请参考 4.2 节。

3) 烧录镜像到 eMMC

● eMMC 无镜像

设置 TF 卡方式启动（请参考表 4-1），将 TF 插入开发板的 TF 卡槽（J2），插上电源。系统启动后开始自动烧写镜像至 eMMC。可以接上 debug 串口(J3)查看烧录状态。

● eMMC 有镜像

设置 eMMC 方式启动（请参考表 4-1），将 TF 插入开发板的 TF 卡槽，插上电源。用 TF 卡烧录镜像到 eMMC 时，首先要确认 eMMC 是否已经烧录过系统，如果已经烧录过系统，此时默认会从 eMMC 启动，需要使用以下命令破坏 eMMC 的内容：

```
myir@system1:~$ dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=5;mmc bootp  
art enable 0 1 /dev/mmcblk0
```

然后按复位键或者重新上电，由于此时 eMMC 内容已经破坏，将会自动从 TF 卡启动，然后自动烧写镜像到 eMMC。



5. 如何修改板级支持包

前面的章节已经比较完整地讲述了基于 Yocto 项目构建运行在 MYD-YM62X 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。本节将对 MYIR 创建的层级 BSP 进行说明，而且还描述如何通过独立环境和 Yocto 项目编译内核、U-Boot，以及如何下载内核和 uboot。

5.1. meta-bsp 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 层基于在 TI 官方的 meta-ti 层建立的适用于 MYD-YM62X 开发板的层，其下面的 meta-bsp 层中包含 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 MYD-YM62X 开发板设计的硬件，定制自己的应用，从而构建适合自己的系统镜像，本节主要介绍 meta-myr 层下面的 meta-bsp 层，其包含的具体内容如下：

```
myir@system1:~$ tree -a -L 1 myd-ym62x/sources/meta-myr/meta-bsp/
meta-bsp/
myir@system1:~/AM62/tisdk/sources$ tree -a -L 1 meta-myr/meta-myr-bsp/
meta-myr/meta-myr-bsp/
├── classes
├── conf
├── COPYING.MIT
├── licenses
├── README
├── recipes-bsp
├── recipes-connectivity
├── recipes-core
├── recipes-devtools
├── recipes-graphics
├── recipes-kernel
└── recipes-myr
```



```
├─ recipes-security
├─ recipes-ti
└─ wic
```

13 directories, 2 files

部分层说明：

表 5-1.meta-bsp 层内容说明

源代码与数据	描述
conf	包含当前层路径信息和机器软件配置
recipes-bsp	包含 atf、uboot 以及固件等配置信息
recipes-kernel	包含 linux 内核的资源与第三方固件资源
recipes-myrir	包含文件系统的配置信息

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。



5.2. 板级支持包介绍

为了适配用户新的硬件平台，首先需要了解米尔的 MYD-YM62X 开发板提供了哪些资源，具体的信息可以查看《MYD-YM62X SDK 发布说明》。除此之外我们也对 BSP 各个部分中需要改动的一些文件列表整理出来，方便用户查找和修改，具体内容如下表所示：

表 5-2.添加配置信息

项目	设备树	说明
U-boot	arch/arm/dts/myb-am62x-dev.dt arch/arm/dts/myb-am62x-dev-u-boot.dtsi	资源设备树
Kernel	arch/arm64/configs/defconfig	内核配置表
	arch/arm64/boot/dts/myir/myd-ym62x-6254.dts	myd-y6254 的设备树
	arch/arm64/boot/dts/myir/myd-ym62x-6252.dts	myd-y6252 的设备树
	arch/arm64/boot/dts/myir/myd-ym62x-6231.dts	myd-y6231 的设备树
	arch/arm64/boot/dts/myir/myd-ym62x-base.dts	基本资源设备树
	arch/arm64/boot/dts/myir/myd-ym62x-lvds-dual.dtso	双路 lvds 的 dtso 文件
	arch/arm64/boot/dts/myir/myd-ym62x-lvds.dtso	单路 lvds 的 dtso 文件

MYD-YM62X 的正常启动过程如下：

- 上电复位 mcu 的 rom 先启动。ROM 是嵌入处理器芯片内的一小块 ROM 或写保护闪存。它包含处理器在上电或复位时执行的第一个代码。根据某些带式引脚或内部保险丝的配置，它可以决定从哪里加载要执行的代码的下一部分以及如何或是否验证其正确性或有效性。但是基于 K3 架构的处理器，ROM 只支持通过 MCU(R5)启动，也就是上电复位后，MCU 检测到复位释放信号后，开始执行 MCU 的 ROM 代码。
- MCU 的 ROM 向 TIFS 服务核心请求请求加载和验证 MCU 的 SPL 镜像 tiboot3.bin 并加载系统固件到 TIFS 核心。然后执行 R5 SPL 镜像，初始化 DDR 配置并加载 A53 SPL 镜像 tispl.bin 和 R5 固件。接着向 TIFS 核心发送请求，启动 A53 和跳转到设备管理固件镜像，执行设备的配置、监控、通讯等服务。
- TIFS 核心发送复位释放信号给 A53 核，A53 接受到复位释放信号后，开始执行 ATF 或 OPTEE 固件。ATF 和 OP-TEE 是两个独立的固件，具有不同的功能和用途。ATF 主要用于设备的启动过程中，提供安全启动和运行环境，保护设备免受攻击。而 OP-TEE 则用于安全敏感的应用程序，提供可信执行环境，保护敏感数据和操作。这一步可以根据客户的要求增加或者去掉。
- 启动 A53 的 SPL 镜像，spl 主要是把 Uboot 拷贝到外部内存中运行，所以它自己是运行在内部内存中的。



- 启动 UBOOT 镜像，建立了适当的系统软硬件环境,为最终调用操作系统内核做好准备。

下面章节重点讲述用户基于我们提供的 Bootloader, Kernel 和 Yocto 源代码和数据进行的流程。

5.3. 板载 BootLoader 编译与更新

AM62X 平台分为 R5 的引导和 A53 的引导。R5 的引导根据 u-boot 的不同配置生成，然而 A53 的引导会需要 U-boot、OPTEE 和 ATF 共同生成，引导列表如下：

表 5-3.引导镜像列表

二进制文件	说明
tiboot3.bin	R5 核的引导镜像，负责 DDL 配置、A53 引导以及 R5 固件加载等。由 U-boot 生成
tispl.bin	A53 的引导镜像，负责提供安全启动环境以及加载 u-boot 。由 U-boot、Atf、Optee 共同生成
u-boot.img	U-boot 启动镜像，建立适当的运行环境，并为加载内核做好准备

下面就介绍如何更新和编译 AM62X 的引导镜像。

5.3.1. 如何在独立环境下编译 bootloader

1) 初始化 SDK

在单独编译引导加载程序之前，需要初始化编译链：

```
myir@system1:~$ wget https://developer.arm.com/-/media/Files/downloads/gnu/11.3.rel1/binrel/arm-gnu-toolchain-11.3.rel1-x86_64-arm-none-linux-gnueabi.tar.xz
myir@system1:~$ tar -Jxvf arm-gnu-toolchain-11.3.rel1-x86_64-arm-none-linux-gnueabi.tar.xz -C $HOME
myir@system1:~$ wget https://developer.arm.com/-/media/Files/downloads/gnu/11.3.rel1/binrel/arm-gnu-toolchain-11.3.rel1-x86_64-aarch64-none-linux-gnu.tar.xz
myir@system1:~$ tar -Jxvf arm-gnu-toolchain-11.3.rel1-x86_64-aarch64-none-linux-gnu.tar.xz -C $HOME
```



```
myir@system1:~$ export PATH=$PATH:$HOME/arm-gnu-toolchain-11.3.rel1-x86_64-aarch64-none-linux-gnu/bin:$HOME/arm-gnu-toolchain-11.3.rel1-x86_64-arm-none-linux-gnueabi/bin
```

使能编译链后，需要设置几个组件的源码路径：

```
$ export UBOOT_DIR=<path-to-ti-u-boot>
$ export TI_LINUX_FW_DIR=<path-to-ti-linux-firmware>
$ export TFA_DIR=<path-to-arm-trusted-firmware>
$ export OPTEE_DIR=<path-to-ti-optee-os>
```

2) 准备 bootloader 源码

用户可以在米尔的光盘资料 04-sources/目录中找到 bootloader 源代码，并将其复制到您的工作目录中并解压文件。

如创建并进入工作目录 myd-ym62x-bsp。或者用户可以直接使用如下命令从 github 下载源码：

```
myir@system1:~$ mkdir myd-ym62x-bsp
myir@system1:~$ cd myd-ym62x-bsp
myir@system1:~/myd-ym62x-bsp$ git clone https://github.com/MYIR-TI/myir-ti-uboot.git -b myd-ym62x-uboot-2023.04
```

3) 如何编译 tiboot3.bin

tiboot3.bin 是 R5 的 SPL 文件，它由 uboot 生成，编译步骤如下：

- 使能环境变量

请参考 5.3.1 的初始化 SDK 部分。这里我使能 uboot 的路径如下：

```
myir@system1:~/AM62/myir-ti-bootloader/myir-ti-uboot$ export UBOOT_DIR=/media/home/myir/AM62/myir-ti-bootloader/myir-ti-uboot
```

- 编译 R5 引导镜像

```
export BINMAN_DIRS=/media/home/wujl/AM62/bootloader/binman
myir@system1:~/AM62/myir-ti-bootloader/myir-ti-uboot$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- myc_am62x_r5_defconfig O=$UBOOT_DIR/out/r5 BINMAN_INDIRS=$BINMAN_DIRS -j10
```



```
myir@system1:~/AM62/myir-ti-bootloader/myir-ti-uboot$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- O=$UBOOT_DIR/out/r5 -j10 BINMAN_INDIRS=$BINMAN_DIRS
```

4) 如何编译 tispl.bin

tispl.bin 是 A53 的引导镜像，需要 u-boot、optee 和 atf 共同生成。

- 编译 optee

进入 myir-ti-optee 目录，执行下面命令编译：

```
myir@system1:~/AM62/myir-ti-bootloader/myir-ti-optee$ make CROSS_COMPILE=aarch64-none-linux-gnu- CROSS_COMPILE=arm-none-linux-gnueabi- PLATFORM=k3-am62x CFG_ARM64_core=y
```

- 编译 atf

进入 myir-ti-atf 目录，执行下面命令编译：

```
myir@system1:~/AM62/myir-ti-bootloader/myir-ti-atf$ make ARCH=aarch64 CROSS_COMPILE=aarch64-none-linux-gnu- PLAT=k3 TARGET_BOARD=lite SPD=optee
```

- 编译 tispl.bin 和 u-boot.img

进入 myir-ti-uboot 目录，设置 BINMAN_DIR 变量(路径根据用户的目录决定)，执行下面命令编译：

```
myir@system1:~/AM62/myir-ti-bootloader/myir-ti-uboot$ export BINMAN_DIR=/media/home/wujl/AM62/bootloader/binman
myir@system1:~/AM62/myir-ti-bootloader/myir-ti-uboot$ make ARCH=arm CROSS_COMPILE=aarch64-none-linux-gnu- myc_am62x_a53_defconfig O=$UBOOT_DIR/out/a53
myir@system1:~/AM62/myir-ti-bootloader/myir-ti-uboot$ make ARCH=arm CROSS_COMPILE=aarch64-none-linux-gnu- BL31=$TFA_DIR/build/k3/lite/release/bl31.bin TEE=$OPTEE_DIR/out/arm-plat-k3/core/tee-pager_v2.bin O=$UBOOT_DIR/out/a53 BINMAN_INDIRS=$BINMAN_DIRS
```

5.3.2. 在 Yocto 项目下编译 u-boot

当用户修改 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本地 git 仓库，同时修改元层的对应源代码的拉取地址（SRC_U



R) 和 commit 值 (SRCREV)。以便配方能够找到并获取本地 uboot 源码，参考示例如下：

1) 查看 commit 值

进入 uboot 源码修改 uboot 源码，并查看 commit 值，如红色字符串所示：

```
myir@system1:~$ cd myd-ym62x-bsp/myir-ti-uboot/  
commit f4e7b03f15da6cd88d31d29a2b9b7e0812534554 (HEAD -> ti-u-boot-2023.04, master)  
Author: duxy <568988005@qq.com>  
Date: Fri Sep 15 09:45:12 2023 +0800  
  
FEAT: add myd-am62x support  
  
commit c7f2ba34eff21cd4200aaafaa0823f09d03653fa  
Author: duxy <568988005@qq.com>  
Date: Fri Sep 15 09:41:06 2023 +0800  
  
INIT:base tag: cicd.kirkstone.202307061739, tag: 09.00.00.006
```

2) 修改源码位置

yocto 中指定 uboot 源码的位置在 `sources/meta-myir/meta-myir-bsp/recipes-bsp/u-boot/u-boot-ti.inc` 文件，修改如下红色显示：

```
BRANCH ?= "master"  
#UBOOT_GIT_URI ?= "git://git.ti.com/git/ti-u-boot/ti-u-boot.git"  
UBOOT_GIT_URI ?= "git:///media/home/wujl/AM62/bsp/myir-ti-uboot"  
#UBOOT_GIT_PROTOCOL ?= "https"  
UBOOT_GIT_PROTOCOL ?= "file"  
SRC_URI = "${UBOOT_GIT_URI};protocol=${UBOOT_GIT_PROTOCOL};branch=${BRANCH}"
```

- UBOOT_GIT_URI: uboot 代码下载位置
- RANCH: 分支名称
- UBOOT_GIT_PROTOCOL: 下载协议，这里是本地下载，所以设置成 “file”

3) 初始化 SDK 环境变量



进入 uboot 目录，初始化环境变量：

```
myir@system1: ~/myd-ym62x-yocto$ cd build
myir@system1: ~/myd-ym62x-yocto/build$ . conf/setenv
```

4) 编译 uboot

执行下面的命令即编译 uboot：

```
myir@system1:/media/myir/myd-ym62x/build$ bitbake -c cleansstate u-boot-ti-s
tagging
myir@system1:/media/myir/myd-ym62x/build$ bitbake u-boot-ti-staging
```

生成镜像文件为 *build/arago-tmp-default-glibc/deploy/images/myd-am62x/tispl.bin*。

5) 确认 uboot 是否更新

当修改完 uboot 之后，需要确认 uboot 是否已经更新，执行以下命令：

```
myir@system1:/media/myir/myd-ym62x/build$ sstrings arago-tmp-default-glibc/
deploy/images/myd-am62x/tispl.bin | grep 2023
Built : 00:42:57, Jan 13 2023
Jun 15 2023
Jun 15 2023
v2023.06.0.0-REL.MCUSDK.09.00.00.12
U-Boot SPL 2023.04-gf4e7b03f15 (Sep 15 2023 - 01:45:12 +0000)
```

由上面 “gf4e7b03f15” 字符串可知，修改后的 uboot 版本与 SRCREV 前面字符一致，即 uboot 已经更新。

5.3.3. 如何单独更新 bootloader

1) 烧录镜像到 eMMC

拷贝编译的镜像到开发板，查看对应分区：

```
root@myd-am62x:~# cat /proc/partitions
major minor #blocks name
.....
179      32  15267840 mmcblk0
179      33   122880 mmcblk0p1
179      34  15042560 mmcblk0p2
```



```
179    64    4096 mmcblk0boot0
```

```
179    96    4096 mmcblk0boot1
```

执行下面命令，把镜像拷贝到对应磁盘：

- 单独更新 **tiboot3.bin**

```
myir@system1:~$ dd if=tiboot3.bin of=/dev/mmcblk0boot0 bs=512 seek=0
```

```
573+1 records in
```

```
573+1 records out
```

```
293585 bytes (294 kB, 287 KiB) copied, 0.0321277 s, 9.1 MB/s
```

- 单独更新 **tispl.bin**

```
myir@system1:~$ dd if=tispl.bin of=/dev/mmcblk0boot0 bs=512 seek=1024
```

```
2228+1 records in
```

```
2228+1 records out
```

```
1141043 bytes (1.1 MB, 1.1 MiB) copied, 0.113484 s, 10.1 MB/s
```

- 单独更新 **u-boot.img**

```
root@myd-am62x:~# dd if=u-boot.img of=/dev/mmcblk0boot0 bs=512 seek=5  
120
```

```
1868+1 records in
```

```
1868+1 records out
```

```
956855 bytes (957 kB, 934 KiB) copied, 0.0971334 s, 9.9 MB/s
```



5.4. 板载 Kernel 编译与更新

5.4.1. 编译 Linux

1) 获取源码

创建并进入 BSP 工作目录 myd-ym62x-bsp，然后使用如下命令下载源码：

```
myir@system1:~$ mkdir myd-ym62x-bsp
myir@system1:~$ cd myd-ym62x-bsp
myir@system1:~/myd-ym62x-bsp$ git clone https://github.com/MYIR-TI/myir-ti-l
inux.git -b myd-am62x-linux-6.1.46
```

5.4.2. 在独立的交叉编译环境下编译 Kernel

1) 加载 SDK 环境变量到当前 shell

单独编译 Kernel 之前需要先申明编译链，具体方法请参考 2.3 节。

```
myir@system1:~$ source ~/AM62/tool_chain/myir/environment-setup-aarch64-
oe-linux
```

2) 进入 Kernel 源码，配置并编译

- 整体编译

进入 Kernel 源码，使用下面命令配置并编译源码：

```
myir@system1:~$ cd myd-ym62x-bsp/myir-ti-linux
myir@system1:~/myd-ym62x-bsp/myir-ti-linux$ make distclean
myir@system1:~/myd-ym62x-bsp/myir-ti-linux$ make defconfig ti_arm64_prune.c
onfig
myir@system1:~/myd-ym62x-bsp/myir-ti-linux$ make Image modules dtbs -j16
myir@system1:~/myd-ym62x-bsp/myir-ti-linux$ mkdir ../test
myir@system1:~/myd-ym62x-bsp/myir-ti-linux$ make modules_install INSTALL_
MOD_PATH=../test
```

生成的 Image 在 *arch/arm64/boot/Image*；生成的 dtb 文件在 *arch/arm64/boot/dts/myir*；生成的模块在 test 目录，里面有一个 kernel 源码的链接需要删掉后才能拷贝到开发板，或者将 test 目录打包压缩也可以避免因为存在链接让 scp 会错误传输 kernel 源码的问题。



- 单独编译设备树

```
myir@system1:~/myd-ym62x-bsp/myir-ti-linux$ make dtbs
```

生成的 dtb 文件在 arch/arm64/boot/dts/myir/ 目录。

5.4.3. 在 Yocto 项目下编译 Kernel

当用户修改 Kernel 代码之后，也可以使用 Yocto 进行整个镜像的构建。此时修改元层的对应源代码的拉取地址（SRC_URI）和 commit 值（SRCREV）。以便配方能够找到并获取本地 Linux 源码，参考示例如下。

1) 查看 commit 值

进入源码目录，修改 Kernel 源码，并查看 commit 值，如红色字符串所示：

```
myir@system1:~$ cd myd-ym62x-bsp/myir-ti-linux
myir@system1:~/myd-ym62x-bsp/myir-ti-linux$ git log
commit 4c20dc4003c94a10472fab1e28cefbda5335ab41 (HEAD -> ti-linux-6.1.y)
Author: duxy <568988005@qq.com>
Date: Tue Sep 19 08:41:14 2023 +0800
    FIX: add keep-power-in-suspend to sdhci2 for wifi can suspend
commit b5fbb4ba2bf7f37379e6cb1d594972a304caa230
Author: duxy <568988005@qq.com>
Date: Fri Sep 15 16:53:38 2023 +0800
    FIX: add usbss0 okay
```

2) 修改源码位置

yocto 中指定 kernel 源码的位置在 sources/meta-myir/meta-myir-bsp/recipes-kernel/linux/linux-ti-staging_6.1.bb 文件，然后修改如下红色显示：

```
BRANCH ?= "ti-linux-6.1.y"
#SRCREV ?= "40c32565ca0e213fb653570cc618408ee8e9c6cf"
SRCREV = "${AUTOREV}"
PV = "6.1.33+git${SRCPV}"

# Append to the MACHINE_KERNEL_PR so that a new SRCREV will cause a rebuild
```



```
MACHINE_KERNEL_PR:append = "b"
PR = "${MACHINE_KERNEL_PR}"

#KERNEL_GIT_URI ?= "git://git.ti.com/git/ti-linux-kernel/ti-linux-kernel.git"
KERNEL_GIT_URI ?= "git:///media/home/wujl/AM62/bsp/myir-ti-linux"
#KERNEL_GIT_PROTOCOL ?= "https"
KERNEL_GIT_PROTOCOL ?= "file"
```

- KERNEL_GIT_URI: Kernel 代码下载位置
- SBRANCH: 分支名称
- SRCREV: 自动识别到 commit 值
- KERNEL_GIT_PROTOCOL: 本地下载协议

3) 初始化 SDK 环境变量

编译需要执行如下命令初始化环境:

```
myir@system1: ~/myd-ym62x-yocto$ cd build
myir@system1: ~/myd-ym62x-yocto/build$ . conf/setenv
```

4) 编译 Kernel

执行下面的命令即编译 kernel

```
myir@system1: ~/myd-ym62x-yocto$ bitbake -c cleansstate virtual/kernel
myir@system1: ~/myd-ym62x-yocto$ bitbake virtual/kernel
```

生成镜像文件为 *build-xwayland/tmp/deploy/images/myd-ym62x/Image*。



5.4.4. 如何单独更新 Kernel 和设备树

1) 烧录镜像到 TF 卡

将 TF 卡用读卡器插入主机，boot 分区会自动挂载到主机(fat 格式)，拷贝内核和设备树到此目录即可，如：

```
myir@system1:~$ cp -rf myd-ym62x-bsp/myir-ti-linux/arch/arm64/boot/dts/myir  
/*.dtb <boot_partition>/boot/dtb/myir  
myir@system1:~$ cp -rf myd-ym62x-bsp/myir-ti-linux/arch/arm64/boot/Image  
<boot_partition>/boot/
```

2) 烧录镜像到 eMMC

拷贝编译的镜像 (Image) 和设备树文件 (*.dtb) 到开发板 mmcblk0p2 分区，可以用 scp 或者 U 盘拷贝。启动开发板，挂载 rootfs 分区，直接把镜像拷贝到 rootfs 分区的 boot 目录。操作步骤如下：

挂载文件系统分区：

```
root@myd-y62x:~# mount /dev/mmcblk0p2 /mnt
```

在主机上使用 SCP 拷贝内核到文件系统分区 boot 目录 (scp 需要使用网络)：

```
myir@system1:~# scp Image root@192.168.40.102:/mnt/boot
```

在主机上使用 SCP 拷贝设备树到系统分区目录：

```
myir@system1:~# scp *.dtb root@192.168.40.102:/mnt/boot/dtb/myir
```

在开发板执行同步命令：

```
root@myd-y62x:~# sync
```

3) 更新模块文件到 eMMC

在 5.4.2 节 2) 进入 Kernel 源码，配置并编译，这一小节中，模块文件被编译生成了位于内核源码上层目录的 test 中，本小节将说明如何将模块文件更新到 eMMC。

将 test 目录打包压缩为 tar.gz

```
myir@system1:~# tar cvf test.tar.gz test
```

通过 SCP 拷贝到文件系统根目录：

```
myir@system1:~# scp test.tar.gz root@192.168.40.102:/
```



在开发解压 test 包：

```
root@myd-y62x:/# tar xvf test.tar.gz
```

将 test 目录下的 lib/modules 复制到开发板根目录的/lib/modules：

```
root@myd-y62x:/# cp test/lib/modules /lib/ -rf
```

删除 test 目录并同步保存修改：

```
root@myd-y62x:/# rm test -rf && sync
```

重启开发板使用 lmod 查看模块加载情况：

```
root@myd-y62x:~/everest-utils/docker# lsmod
```

Module	Size	Used by
xt_nat	16384	6
xt_tcpudp	16384	10
veth	32768	0



6. 如何适配您的硬件平台

6.1. 如何创建自己的 machine

在开发过程中，用户有时需要创建自定义板配置。本节将通过一个实例讲解用户如何创建属于自己的 machine。

6.1.1. 在 Yocto 创建一个板子配置

1) 选择类似 machine 文件

复制一个类似的 machine 文件，并重命名为一个你板子的指定名字。如 myd-am62x 开发板 machine 文件 myd-am62x.conf 在 sources/meta-myir/meta-myir-bsp/conf/machine 这个目录，进入这个目录，machine 文件如下：

```
myir@system1:~/myd-ym62x-yocto$ cd myd-ym62x-yocto/sources/meta-myir/meta-myir-bsp/conf/machine
myir@system1:~/myd-ym62x-yocto/sources/meta-myir/meta-myir-bsp/conf/machine$ ls myd-am62x.conf
myd-am62x.conf
```

2) 复制并重命名

找到类似的 machine 文件后，复制并重命名为你自己的 machine 文件，如：

```
myir@system1:~/myd-ym62x-yocto/sources/meta-myir/meta-myir-bsp/conf/machine$ cp myd-am62x.conf myd-test.conf
myir@system1:~/myd-ym62x-yocto/sources/meta-myir/meta-myir-bsp/conf/machine$ ls myd-test.conf
myd-test.conf
```

3) 修改 machine 文件

创建好 machine 文件后，修改 build/conf/local.conf 文件的 machine 属性如下：

```
myir@myir_server:~/myd-ym62x-yocto$ ls -l
MACHINE ?= "myd-test"
```

4) 编译并测试



当 machine 文件创建完成之后，您可以通过编译并下载测试，执行如下命令编译最小镜像测试：

```
myir@system1:~/myd-ym62x-yocto$ cd build
myir@system1:~/myd-ym62x-yocto/build$ . conf/setenv
myir@system1:~/myd-ym62x-yocto/build$ bitbake myir-image-core
```

编译完成后生成的镜像 arago-tmp-default-glibc/deploy/images/myd-test/myir-image-core-myd-test.wic.xz，拷贝并解压按照 4.2 节烧写到 TF 卡，启动测试：

```
root@myd-test:~#
```

6.1.2. 在 uboot 创建板子配置文件

在开发过程中，用户一般需要根据自己的板子要求创建属于自己的板子配置文件，本节将通过一个简单实例演示如何一步步创建自己的板子配置文件。

1) 创建 board

在创建自己的 board 配置时，用户可以在相对应的板子目录通过复制并重命名建立自己的板子配置文件，一般板子配置文件都在 uboot 源码的 board 目录。进入 uboot 源码 board 子目录下的 myir 子目录，复制 myc_am62x 文件夹为 test_am62x，如下所示：

```
myir@myir_server:~/myd-ym62x-bsp/myir-ti-uboot/board/myir$ ls
common keys myc_am62x mys_6ulx test_am62x
```

进入 test_am62x 目录，修改 myc_am62x.env 为 test_am62x.env。

2) 制作板子.config 文件

- 修改新创建的板子 Kconfig

进入 test_am62 目录，修改 Kconfig 文件如下红色标记：

```
choice
    prompt "MYIR AM62x based boards"
    optional

config TARGET_TEST_AM62X_A53_DEV
    bool "MYIR based AM625 EVM running on A53"
    select ARM64
    select BINMAN
```



```
config TARGET_TEST_AM62X_R5_DEV
    bool "MYIR based AM625 EVM running on R5"
    select CPU_V7R
    select SYS_THUMB_BUILD
    select K3_LOAD_SYSPW
    select RAM
    select SPL_RAM
    select K3_DDRSS
    select BINMAN
    imply SYS_K3_SPL_ATF

endchoice

if TARGET_TEST_AM62X_A53_DEV
config SYS_BOARD
    default "test_am62x"

config SYS_VENDOR
    default "myir"

config SYS_CONFIG_NAME
    default "test_am62x"

source "board/myir/common/Kconfig"

endif

if TARGET_TEST_AM62X_R5_DEV

config SYS_BOARD
    default "test_am62x"
```



```
config SYS_VENDOR
    default "myir"

config SYS_CONFIG_NAME
    default "test_am62x"

config SPL_LDSCRIPT
    default "arch/arm/mach-omap2/u-boot-spl.lds"
source "board/myir/common/Kconfig"

endif
```

- 创建新板子的头文件

从源码根目录下进入 include/configs 目录，复制 myc_am62x.h 为 test_am62x.h，如下：

```
myir@myir_server:~/myd-ym62x-bsp/myir-ti-uboot/include/configs$ ls -l myc_am62x.h
-rwxr--r-- 1 myir myir 987 9月 15 09:49 myc_am62x.h
myir@myir_server:~/myd-ym62x-bsp/myir-ti-uboot/include/configs$ ls -l test_am62x.h
-rwxr--r-- 1 wujl wujl 987 9月 23 15:27 test_am62x.h
```

注意：由于 SYS_CONFIG_NAME 的名字改为了 test_am62x,所以此头文件改为 test_am62x.h。

- 定制系统板子配置文件

从源码根目录下进入 configs 目录，复制 myc_am62x_a53_defconfig 为 ttest_am62x_a53_defconfig，复制 myc_am62x_r5_defconfig 为 test_am62x_r5_defconfig，如下：

```
myir@myir_server:~/myd-ym62x-bsp/myir-ti-uboot/configs$ cp myc_am62x_a53_defconfig test_am62x_a53_defconfig
myir@myir_server:~/myd-ym62x-bsp/myir-ti-uboot/configs$ cp myc_am62x_r5_defconfig test_am62x_r5_defconfig
```

然后修改 test_am62x_a53_defconfig 文件，如下：

```
#CONFIG_TARGET_MYC_AM62X_A53_DEV=y
```



```
CONFIG_TARGET_TEST_AM62X_A53_DEV=y
```

修改 test_am62x_r5_defconfig 文件，如下：

```
#CONFIG_TARGET_MYC_AM62X_R5_DEV=y  
CONFIG_TARGET_TEST_AM62X_R5_DEV=y
```

- 使能 Kconfig 文件

以上步骤完成之后，还需要使能 Kconfig，修改文件 arch/arm/mach-k3，增加如下内容：

```
source "board/myir/myc_am62x/Kconfig"  
source "board/myir/test_am62x/Kconfig"
```

- 修改设备树

修改 myb-am62x-dev-binman.dtsi 设备文件，如下红色表示：

```
//#ifdef CONFIG_TARGET_MYC_AM62X_R5_DEV  
#ifdef CONFIG_TARGET_TEST_AM62X_R5_DEV  
.....  
//#ifdef CONFIG_TARGET_MYC_AM62X_A53_DEV  
#ifdef CONFIG_TARGET_TEST_AM62X_R5_DEV  
#define SPL_NODTB "spl/u-boot-spl-nodtb.bin"  
#define SPL_AM625_SK_DTB "spl/dts/myb-am62x-dev.dtb"
```

经过这几个步骤基本上是已经定制好了板子，如果用户需要用 yocto 编译，那么还需要根据 6.1.1 节修改 machine 里面的设备树配置信息。

3) 编译

执行以下命令，编译：

```
myir@system1:~/myd-ym62x-bsp/myir-ti-uboot$ make ARCH=arm CROSS_COM  
PILE=arm-none-linux-gnueabihf- distclean  
myir@system1:~/myd-ym62x-bsp/myir-ti-uboot$ make ARCH=arm CROSS_COM  
PILE=arm-none-linux-gnueabihf- test_am62x_r5_defconfig  
myir@system1:~/myd-ym62x-bsp/myir-ti-uboot$ make ARCH=arm CROSS_COM  
PILE=arm-none-linux-gnueabihf- -j10
```

6.2. 如何创建您的设备树



6.2.1. 板载设备树层级的介绍

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。如将部分设备信息结构存放到 device tree 文件中。uboot 或内核最终将其 device tree 编译成 dtb 文件，使用过程中通过解析该 dtb 来获取板级设备信息。

1) myir-ti-uboot 设备树介绍

下面 MYD-YM62X 的 uboot 各部分设备树的信息列表，以便用户参考：

表 6-1.MYD-YM62X U-boot 设备树列表

项目	设备树	描述
U-boot	myb-am62x-r5-dev.dts	MYD-AM62X R5 的设备树
	myb-am62x-dev.dts	MYD-AM62X A53 的设备树
	myb-am62x-dev-binman.dtsi	通过使用 binman 工具，可以配置和生成特定于 AM62x 平台的 U-Boot 二进制映像文件，该文件包含了引导程序、设备树 (Device Tree)、启动脚本等信息
	myb-am62x-dev-u-boot.dtsi	针对 myb-am62x 系列特定配置

编译 MYD-YM62X 板 uboot 源码时会合并相关的所有 dts/dtsi 文件，生成 R5 阶段的设备树 myb-am62x-r5-dev.dtb 和 uboot 阶段默认使用的 myb-am62x-dev.dtb 文件。

2) myir-ti-linux 设备树介绍

myir-ti-linux 的设备树大概层级关系为：k3-am62-main.dtsi->k3-am62.dtsi->k3-am625.dtsi->myd-ym62x 系列设备树。下面是 MYD-YM62X 的各部分设备树的详细信息列表，以便用户开发参考：

表 6-2.MYD-YM62X Linux 设备树列表

项目	设备树	描述
Kernel	k3-am62-main.dtsi	包含 myd-ym62x 系列的所有资源配置
	myd-ym62x-6231.dts、myd-ym62x-6252.dts、myd-ym62x-6254.dts	针对不同的 myc-ym62x 核心板的设备树配置，默认 HDMI 显示
	myd-ym62x-hdmi-audio.dtso、myd-ym62x-lvds-dual.dtso、myd-ym62x-lvds.dtso、myd-ym62x-lvds-j6.dtso	针对基本设备树的外设资源补充，可以动态加载



6.2.2. 设备树的添加

1) Uboot 创建设备树

- 选择类似的设备树文件

从源码根目录下进入 `arch/arm/dts` 目录，复制 `myb-am62x-dev.dts` 为 `myb-am62x-test.dts`，如：

```
myir@system1:~/myd-ym62x-bsp/myir-ti-uboot$ cp arch/arm/dts/myb-am62x-dev.dts arch/arm/dts/myb-am62x-test.dts
myir@system1:~/MYD-YM62X-bsp/myir-ti-uboot$ cp arch/arm/dts/myb-am62x-dev-u-boot.dtsi arch/arm/dts/myb-am62x-test-u-boot.dtsi
```

- 修改设备树 Makefile

接着修改目录下的设备树 Makefile，增加内容如下：

```
dtb-$(CONFIG_SOC_K3_AM625) += myb-am62x-dev.dtb \
    myb-am62x-r5-dev.dtb \
    myb-am62x-test.dtb \
    myb-am62x-r5-test.dtb
```

- 修改板子配置文件

如然后修改 R5 的配置文件 `test_am62x_r5_defconfig` 配置文件默认的设备树，如下：

```
CONFIG_DM_GPIO=y
CONFIG_SPL_DM_SPI=y
CONFIG_DEFAULT_DEVICE_TREE="myb-am62x-r5-test"
```

2) Kernel 创建设备树

MYIR 系列设备树在 `arch/arm64/boot/dts/myir` 目录，用户在该目录也可以通过复制 MYIR 的设备树然后重命名，方法过程如 uboot 类似，请参考以上步骤。



6.3. 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，本节将以实例来讲解功能管脚的控制实现。

6.3.1. GPIO 管脚配置的方法

MYD-YM62X 板的 IO 管脚大部分定义在 arch/arm64/boot/dts/myir/myd-ym62x-common.dtsi 设备树文件中。TI 的引脚配置宏定义在文件 arch/arm64/boot/dts/myir/k3-pinctrl.h 中，下面将介绍一个配置的实例。

1) 查看 pinctrl 配置规则

TI pinctrl 配置格式

```
#define AM62X_IOPAD(pa, val, muxmode)      (((pa) & 0x1fff) ((val) | (muxmode))  
#define AM62X_MCU_IOPAD(pa, val, muxmode) (((pa) & 0x1fff) ((val) | (muxmode))
```

其中：

- AM62X_IOPAD 代表 A53 域的引脚配置，AM62X_MCU_IOPAD 代表 R5 侧的引脚配置
- pa: PADCONFIG 寄存器地址复用寄存器偏移地址
- val: 引脚上下拉等属性的配置
- muxmod: 表示引脚可以复用成哪些功能

例如引脚配置如下：

```
AM62X_IOPAD(0x0094, PIN_INPUT, 7)
```

通过 datasheet 手册可知设置 GPIO0_36 为 gpio 输入功能。

2) 在设备树上配置 gpio

下面将使用 DTS 文件来进行设备硬件资源的申请及分配，用户可以在 myd-y62x-common.dtsi 文件下操作 DTS，定义 gpio 设备节点如下：

```
gpioctr_device {  
    compatible = "myir,gpioctr";  
    #pinctrl-names = "default";
```



```
#pinctrl-0 = <&pinctrl_gpio_blue>;  
  
status = "okay";  
gpiocr-gpios = <&main_gpio0 31 0>;  
};
```



6.4. 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。

6.4.1. U-boot 中使用 GPIO 管脚

1) uboot 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 GPIO0_31，可使用下列命令。

```
u-boot=> gpio set 31
gpio: pin 31 (gpio 31) value is 1
u-boot=> gpio clear 31
gpio: pin 31 (gpio 31) value is 0
```

可以用万用表测量到 J11 上 5 脚电压拉高/拉低。

2) Uboot 代码控制

用户可以在 uboot 阶段，在代码中直接实现 IO 的功能，如下列 phy 的电源复位控制。

- 在 uboot 代码控制 phy 复位引脚

在 myir-ti-uboot/board/myir/myc_am62x/som.c 文件中，加入下面的 lvds 电源控制代码：

```
#define GPIO_TO_PIN(bank, gpio)    (32 * (bank) + (gpio))
#define DUAL_LVDS_POWER            GPIO_TO_PIN(0,4)

static void gpio_rst_and_power(void)
{
    int ret;
    int lvds_power;

    lvds_power = DUAL_LVDS_POWER;
```



```
ret = gpio_request(lvds_power, "lvds_power");
if (ret < 0) {
    printf("Unable to get GPIO %d\n", lvds_power);
    return;
}

/* Configure as output */
gpio_direction_output(lvds_power, 0);

gpio_set_value(lvds_power, 1);
}
```



6.4.2. 内核驱动中使用 GPIO 管脚

1) 独立 IO 驱动的使用

在 6.3.1 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 GOIO0_31 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(gpiocr_gpio);
        return PTR_ERR(gpiocr_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。下面以编译成模块为例。

2) 驱动示例编译成单独模块

在工作目录下增加 gpiocr.c 并拷贝上述驱动代码，同时编写独立 Makefile 程序：

```
# 修改 KERN_DIR
# #KERN_DIR = # 板子所用内核源码的目录
KERN_DIR = /media/myir/myd-ym62x-bsp/myir-ti-linux
obj-m += gpiocr.o

all:

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



clean:

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

#

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@system1:/media/myir/myd-ym62x/gpioctrl$ source ~/AM62/tool_chain/myir/environment-setup-aarch64-oe-linux
```

执行 make 命令, 即可生成 gpioctrl.ko 驱动模块文件:

```
myir@system1:/media/myir/myd-ym62x/gpioctrl$ make
make -C /media/myir/myd-ym62x-bsp/myir-ti-linux M=`pwd` modules
make[1]: Entering directory '/media/myir/myd-ym62x-bsp/myir-ti-linux'
CC [M] /media/myir/myd-ym62x/gpioctrl/gpioctrl.o
MODPOST /media/myir/myd-ym62x/gpioctrl/Module.symvers
CC [M] /media/myir/myd-ym62x/gpioctrl/gpioctrl.mod.o
LD [M] /media/myir/myd-ym62x/gpioctrl/gpioctrl.ko
make[1]: Leaving directory '/media/myir/myd-ym62x-bsp/myir-ti-linux'
```



6.4.3. 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 管脚的控制两种基本方式。

- Shell 命令
- 系统调用

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-YM62X 软件评估指南》第 3.1 节描述。

2) 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 6.4.2 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:

```
myir@system1:/media/myir/myd-ym62x/gpioctrl$ source ~/AM62/tool_chain/myir/  
environment-setup-aarch64-oe-linux
```

使用编译命令\$CC 可生成可执行文件 gpiotest。

```
myir@system1:/media/myir/myd-ym62x/gpioctrl$ $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@myd-ym62x:~# gpiotest /dev/myir_gpioctrl0 on  
[ 643.651418] /media/myir/myd-ym62x/gpioctrl/gpioctr.c gpio_drv_write line 39  
[ 643.658779] /media/myir/myd-ym62x/gpioctrl/gpioctr.c gpio_drv_close line 56  
root@myd-ym62x:~# gpiotest /dev/myir_gpioctrl0 off  
[ 674.176149] /media/myir/myd-ym62x/gpioctrl/gpioctr.c gpio_drv_write line 39  
[ 674.183509] /media/myir/myd-ym62x/gpioctrl/gpioctr.c gpio_drv_close line 56
```

可以用万用表测量到 J11 上 3 脚的电压被拉高和拉低。



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，它记录了原始代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，极大地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 会自动地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-YM62X 开发板上实现按键控制）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
```



部分参数说明:

- \$(CURDIR): 表示 Makfile 当前目录全路径
- \$(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/myir/MYD-YM62X/key_led*, 执行为就变为 *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 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/event1 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/am62-sk\:d53/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/user/brightness"); //led on
        else
            system("echo 0 > /sys/class/leds/user/brightness");//le
d 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:

```
myir@system1:/media/myir/myd-ym62x/app_test/key_led$ source ~/AM62/tool_
chain/myir/environment-setup-aarch64-oe-linux
```

执行 make:

```
myir@system1:~$ 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 key_led_test 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，执行以下命令，然后按 USER 按键，可以看到蓝灯亮灭的情况：

```
root@myd-ym62x:~# key_led_test /dev/input/event1 noblock
key test
key test
key test
```



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-YM62X 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。

如需获得安装与配置详情，可从 QtCreator 官方网站获得更多开发指导 <https://www.qt.io/product/development-tools>。

7.3. 应用程序开机自启动

1) 应用程序在 Yocto 的配置

要使用 Yocto 构建生产部署阶段的镜像文件并且包含我们编写的应用，就需要为我们编写的应用创建一个配方（Recipe），helloworld 不具备开机自启动的功能，它配方的编写参见 <https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#hello-world-example>。

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：

```
myir@system1:/media/myir/myd-ym62x/build$ bitbake -s | grep proftpd
proftpd :1.3.7c-r0
```

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，详情请参考第 3 章。

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

```
myir@system1:/media/myir/myd-ym62x/build$ 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

```
myir@system1:/media/myir/myd-ym62x/build$ bitbake proftpd
```

- 打包 proftpd 到文件系统

在 sources/meta-myir/meta-myir-bsp/recipes-core/images/myir-image-core.bb 中增加 proftpd 包：

```
IMAGE_INSTALL_append = "proftpd"
IMAGE_INSTALL += "\
    packagegroup-myir-base \
    packagegroup-myir-console \
    .....
    hwmac \
    proftpd \
```

- 重新构建镜像

```
myir@system1:/media/myir/myd-ym62x/build$ 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：

```
root@myd-am62x:~# ps | grep proftpd
647 nobody  4180 S  proftpd: (accepting connections)
767 root    3088 S  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 `sources/meta-myr/meta-myr-bsp/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) 开机自启动应用

本节还是以 `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 `sources/meta-openembedded/meta-networking/recipes-daemons/proftpd` 目录，目录结构如下。

```
myir@system1:~$ tree myd-ym62x/sources/meta-openembedded/meta-networking/recipes-daemons/proftpd
├── 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:

```
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 '/ftusers/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 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) 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 表示此服务在 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 `sources/meta-myir/meta-bsp` 目录。



8. 参考资料

- **Linux kernel** 开源社区
<https://www.kernel.org/>
- **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>
- **Yocto** 开发指导
<https://www.yoctoproject.org/>



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@myir.cn

技术支持联系方式

邮箱：support.cn@myir.cn

武汉研发中心电话：027-59621648

深圳总部技术电话：0755-22316235



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。

