

# MYD-YM62X Linux System Development Guide



File status:  [ ] Draft  [√] released	<b>FILE ID:</b>	MYIR-MYD-YM62X-SW-DG-EN-L6.1.46
	<b>VERSION:</b>	V1.0[DOC]
	<b>AUTHOR:</b>	Miller
	<b>CREATED:</b>	2023-09-20
	<b>UPDATED:</b>	2023-09-25

# Revision History

Editions	Author	Participants	Date	Notes
V1.0[doc]	Miller		20220925	Initial version: uboot 2023.04, kernel 6.1.46, yocto 2023.04



# CONTENT

Revision History .....	- 2 -
CONTENT .....	- 3 -
1. Overview .....	- 5 -
1.1. Software Resources .....	- 5 -
1.2. Documentation Resources .....	- 6 -
2. Development Environment Preparation .....	- 7 -
2.1. Development Host Environment .....	- 7 -
2.2. Install the build chain .....	- 10 -
3. Build the board image with Yocto .....	- 13 -
3.1. Introduction .....	- 13 -
3.2. Get the source code .....	- 14 -
3.2.1. Get the SDK compressed package from the ROM image .....	- 14 -
3.2.2. Obtain via github .....	- 14 -
3.3. Build development board image .....	- 15 -
3.4. Build the SD card burner image .....	- 21 -
3.5. Build the SDK .....	- 22 -
4. How do I burn a system image .....	- 23 -
4.1. Make a TF card starter .....	- 23 -
4.2. Make a TF card burner .....	- 26 -
5. How to modify the board level support package .....	- 27 -
5.1. Introduction to the meta-bsp layer .....	- 27 -
5.2. Board level support package introduction .....	- 29 -
5.3. Compilation and Update of Bootloader .....	- 30 -
5.3.1. How to compile bootloader in a standalone environment .....	- 31 -
5.3.2. Compile u-boot under Yocto project .....	- 33 -
5.3.3. How to update bootloader separately .....	- 35 -
5.4. Kernel Compilation and Update .....	- 37 -



5.4.1. How to build Kernel in standalone environment .....	37 -
5.4.2. Compile the Kernel in a separate cross-compile environment. ....	37 -
5.4.3. Compile the Kernel under the Yocto project.....	38 -
5.4.4. How do I update the Kernel and device tree separately .....	41 -
<b>6. How to adapt to your hardware platform.....</b>	<b>43 -</b>
6.1. How to create your own machine .....	43 -
6.1.1. Create a board configuration in Yocto .....	43 -
6.1.2. Creating board configuration file in uboot.....	44 -
6.2. How do you create your device tree .....	48 -
6.2.1. Board Level Device Tree .....	48 -
6.2.2. Add your board level device tree .....	49 -
6.3. How to configure CPU function pins according to your hardware .....	51 -
6.3.1. Method of GPIO pin configuration .....	51 -
6.4. How to use your own configured pins .....	53 -
6.4.1. GPIO pins are used in U-boot .....	53 -
6.4.2. How to use GPIO in Kernel driver .....	55 -
6.4.3. How to control a GPIO in Userspace .....	61 -
<b>7. How to add your application .....</b>	<b>65 -</b>
7.1. Makefile-based apps .....	65 -
7.2. Application based on QT .....	70 -
7.3. How to start an application automatically .....	70 -
<b>8. Resources .....</b>	<b>78 -</b>
<b>Appendix A.....</b>	<b>79 -</b>
Warranty & Technical Support Services.....	79 -



# 1. Overview

There are many open source system building frameworks on Linux platform, which facilitate developers to build and customize embedded systems. At present, the most common ones are Buildroot, Yocto, OpenEmbedded, etc. Among them, the Yocto project uses a more powerful and customized approach to build Linux systems suitable for embedded products. Yocto is not only a file system tool, but also provides a complete set of Linux-based development and maintenance workflow, so that the bottom embedded developers and upper application developers under a unified framework to develop, to solve the traditional development mode of scattered and unmanaged development.

This paper first describes the development process of installing and running Linux system and embedded Linux driver and application using Yocto project on MYD-YM62X series development board, including deploying development environment, building system, example analysis of Linux application, image update, etc. After system developers are familiar with the development process of Yocto in Chapter 3, they can refer to the migration guide in Chapter 4 for the actual project needs to customize the BSP, and they can quickly transplant the system to the hardware platform based on the MYD-YM62X core board design.

This document does not include the Yocto project and the introduction of Linux system related basic knowledge, suitable for embedded Linux system developers with a certain development experience. For users in the process of secondary development may use some specific functions, we also provide a detailed application development manual.

## 1.1. Software Resources

MYD-YM62X is equipped with an operating system based on Linux 6.1.46 kernel, which provides a wealth of system resources and other software resources. The development board comes with the cross-compilation tool chain needed for embedded Linux system development, ATF source code, U-boot source code,

Linux kernel and driver module source code, as well as a variety of development and debugging tools for Windows desktop environment and Linux desktop environment, application development samples, etc. For specific software information, please refer to the instructions in Chapter 2 software information in *"MYD-YM62X SDK Release Notes"*.

## 1.2. Documentation Resources

According to the different stages of the user's use of the development board, the SDK contains different types of documents and manuals such as release instructions, getting started guides, evaluation guides, development guides, application notes, common questions and answers. See the instructions in Table 2-4 of the *MYD-YM62X SDK Release Notes* for a detailed list of documents.

## 2. Development Environment

### Preparation

This chapter mainly introduces some software and hardware environment required for the whole development process based on MYD-YM62X development board, such as system transplantation, application development, firmware burning, including necessary hardware accessories, software tools, etc. The specific preparation work will be introduced in detail below.

#### 2.1. Development Host Environment

This section describes how to set up a development environment for MYD-YM62X. By reading this chapter, you will understand the installation and use of the related hardware tools, software development and debugging tools. And can quickly set up the relevant development environment, for later development and debugging preparation. The AM62X series processor is a multi-core heterogeneous processor, which includes:

- 1-4 ARM Cortex A53 cores, can run embedded Linux system, use the development tools of embedded Linux system.
- 1 ARM Cortex M4 core, which can run bare-metal code or other real-time operating systems (RTOS), using Cortex M4 software development tools officially provided by TI.

##### 1) Host software and hardware requirements

###### ● Host hardware

The construction of Yocto project has relatively high requirements for the development host, requiring the processor to have more than dual core CPU, more than 8GB memory, 500GB hard disk or higher configuration. It can be a host with Linux system installed, or a virtual machine running Linux system.

###### ● Host operating system

There are many choices of host operating systems to build Yocto projects on, Detailed information please refer to the Yocto official instructions at

<https://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html#dev-preparing-the-build-host>.

The general choice is to build on a local host with a Linux distribution such as Fedora, openSUSE, Debian, Ubuntu, RHEL, or CentOS. Ubuntu20.04 64bit or higher is recommended here. The subsequent development is also introduced as an example of this system.

## 2) Host Environment Configuration

After installing the ubuntu 20.04 64bit system, you can set up the appropriate configuration first to prepare for the subsequent development.

- **Set the root password**

```
myir@myir-server1:~$ sudo passwd root
```

- **Install ssh**

After installing ssh service, you can use sercureCRT tool ssh2 in the window environment to connect to ubuntu for subsequent development.

```
myir@myir-server1:~$ sudo apt-get install openssh-server
```

Generate the key for the user:

```
myir@myir-server1:~$ su user
```

```
myir@myir-server1:~$ ssh-keygen -t rsa
```

- **Configuring samba**

samba can access the contents of ubuntu in the form of folder directly under the window, which is more convenient to read and write. To install samba:

```
myir@myir-server1:~$ apt-get install samba
```

Add user configuration in /etc/samba/smb.conf, such as linux user name "duxy", as follows:

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

Create an account and set a password:

```
myir@myir-server1:~$ sudo smbpasswd -a duxy
New SMB password:
Retype new SMB password:
Added user duxy.
```

/etc/init.d/smbd restart Restart the samba service:

```
myir@myir-server1:~$ /etc/init.d/smbd restart
[ ok ] Restarting smbd (via systemctl): smbd.service.
```

- **Setting up git**

```
myir@myir-server1:~$ git config --global user.name "user"
myir@myir-server1:~$ git config --global user.email "email"
myir@myir-server1:~$ git config --list
```

- **Install the necessary tools for the SDK**

```
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. Install the build chain

After building a system image with Yocto, you can also build an extensible SDK with Yocto. The ROM image provided by MYIR contains two compiled SDK packages, located at 03-Tools/Toolchains/. The functional description of the two SDK files is shown in the following table:

Table 2-1. Compile toolchains

Toolchain filenames	Description
arago - 2023.04 - toolchain - 2023.04. Sh	Contains a separate cross-development toolchain that also provides qmake, sysroot for the target platform, libraries and headers that Qt application development depends on, etc. Users can directly use this SDK to set up a standalone development environment

Here are the steps to install the SDK:

- **Copy the SDK to your Linux directory**

Copy the SDK zip into the user's working directory on Ubuntu to get an installation script file like this:

`Arago-2023.04-toolchain-2023.04.sh`

- **Run the install script**

Run the shell script with normal user privileges, and you will be prompted for the installation path, which defaults to /opt. This routine is the qt tool chain installed at `/media/home/wujl/AM62/tool_chain/myir` directory, as follows:

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

Setting it up... done

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

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

Initialize the environment variables:

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

### ● Test the SDK

Once the installation is complete, test the SDK by loading the environment variables into the current shell using the following command:

```
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
--sysroot=/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 -
-with-gnu-ld --enable-shared --enable-languages=c, C ++ --enable-threads=posix
--enable-multilib --enable-c99 --enable-long-long --enable-symvers=gnu --ena
ble-libstdc++-pch --program-prefix=aarch64-oe-linux- --without-local-prefix --dis
able-install-libiberty --disable-libssp --enable-libitm --enable-lto --disable-bootstrap
--with-system-zlib --with-linker-hash-style=gnu --enable-linker-build-id --wit
h-ppl=no --with-cloog=no --enable-checking=release --enable-headers=c_glob
al --without-isl - with GXX - include - dir = / not/exist/usr/include/c + + / 11.3.0 --
with-build-time-tools=/media/home/wujl/AM62/tisdk/build/arago-tmp-default-g
libc/work/x86_64-nativesdk-arago-linux/gcc-cross-canadian-aarch64/11.3.0-r0/re
cipe-sysroot-native/usr/aarch64-oe-linux/bin --with-sysroot=/not/exist --with-bui
ld-sysroot=/media/home/wujl/AM62/tisdk/build/arago-tmp-default-glibc/work/x
86_64-nativesdk-arago-linux/gcc-cross-canadian-aarch64/11.3.0-r0/recipe-sysroo
t --enable-standard-branch-protection --with-plugin-ld=ld --enable-poison-syste
m-directories --enable-nls --with-glibc-version=2.28 --enable-initfini-array --enab
le-__cxa_atexit
Thread model: posix
Supported LTO compression algorithms: zlib zstd
gcc version 11.3.0 (GCC)
```

Similarly, install your own toolchain for basic development. When installing two toolchains, specify a different directory instead of using the same directory, or the files will overwrite each other.



## 3. Build the board image with Yocto

### 3.1. Introduction

Yocto is an open source "umbrella" project, which means that there are many sub-projects under it. Yocto just integrates all the projects together, and provides a reference build project Poky to guide developers how to apply these projects to build embedded Linux systems. It contains Bitbake, OpenEmbedded-Core, board support packages, and configuration files for various packages.

MYD-YM62X provides Yocto-compliant configuration files to help developers build Linux system images that can be burned on the MYD-YM62X board. Yocto also provides rich development documentation resources for developers to learn and customize their own systems. Due to the limited space, the use of Yocto cannot be fully introduced, so users are invited to search online on their own.

This section is suitable for developers who need to deeply customize the file system, and want to build a file system from Yocto that conforms to the MYD-YM62X series development board, and based on its customization method. Developers who are new to using MYD-YM62X or have no special needs can directly use the file system already provided by MYDD-YM62x.

**Note:** To build Yocto, you do not need to load the SDK toolchain environment variables in Section 2.2. Please create a new shell or open a new terminal window.



## 3.2. Get the source code

We provide two ways to obtain the source code, one is directly from the ROM image 04-sources directory to obtain the compressed package, and the other is to use the repo to obtain the source code located on github to update in real time for building. Users are invited to choose one of them according to their actual needs. Because Yocto needs to download all the packages in the file system to the local before building, in order to quickly build, MYD-YM62X has packaged the relevant software, which can be directly unpacked and used to reduce the time of repeated download.

### 3.2.1. Get the SDK compressed package from the ROM image

The source packages are available at MYIR development package information *04-Sources/myd-ym62x-yocto.zip*. Copy the zip package to a user-specified directory, for example, unpack the Yocto source package into the working directory myd-ym62x-yocto:

```
myir@myir-server1:~$ mkdir -p myd-ym62x-yocto
myir@myir-server1:~$ unzip myd-ym62x-yocto.zip
```

### 3.2.2. Obtain via github

At present, the BSP source code and Yocto source code of the MYD-YM62X development board are hosted by github and will remain updated for a long time. Please see the MYD-YM62X SDK Release Notes for the code repository address. Users can use the repo to obtain and sync the code on github. Here's how:

Place the *03-tools/repo* file in the *~/bin* directory, add executable permissions, and add it to the PATH variable:

```
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.shbak -f
configs/processor-sdk/processor-sdk-09.00.00-my-d-am62x-config.txt
```

After the sync is successful, you will also get the same directory contents as the source package 04-Sources/myd-ym62x-yocto.zip in the myd-ym62x-yocto directory.

### 3.3. Build development board image

The Yocto project needs to set the corresponding environment variables before building the system. We need to use the *build/conf/setenv* script to set the environment variables before building myir-image-full.

#### 1) View the Yocto source code package content

After unpacking the myd-ym62x-yocto.zip file, or downloading the file via the repo, the following file will be saved, listing the myd-ym62x-yocto directory as follows:

```
myir@myir-server1:~$ 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) Execute the environment variable setting script

The syntax for a script to set the compilation environment is as follows:

```
myir@myir-server1:~$ cd myd-ym62x-yocto
```

```
myir@myir-server1: ~/myd-ym62x-yocto$ ./oe-layertool-setup.sh -f configs/processor-sdk/processor-sdk-09.00.00-my-d-am62x-config.txt
```

After the execution of the configuration script, the build directory will be generated, and the enable environment variable will be entered in the build directory, as follows:

```
myir@myir-server1: ~/myd-ym62x-yocto/build$ cd build
myir@myir-server1: ~/myd-ym62x-yocto/build$ . conf/setenv
```

**Note:** You need to use a normal user to compile yocto, not root

### 3) Build the myir-image-full image

There will be a fetch process to download third-party source code on the network when compiling here. If the network is not good, fetch errors often occur, so it is recommended to download the downloads file on the network disk and place it in the same level directory, such as myd-ym62x-yocto directory. Here to demonstrate the compilation process.

- **Copy the downloads file to the specified directory**

To download the downloads file from the network disk, you first need to copy it to the corresponding directory (e.g., myd-ym62x-yocto), as shown in the red file:

```
myir@myir-server1: ~/myd-ym62x-yocto$ ls
build configs git_retry.sh oe-layertool-setup.sh sample-files sources
downloads
```

downloads can also be pulled directly from the Internet, but the speed depends on the user's Internet connection, so it's generally recommended to stick with the online downloads. The general command to pull files from the Internet is as follows:

```
myir@myir-server1: ~/myd-ym62x-yocto/build$ bitbake myir-image-full --runall=fetch
```

- **Build full image**

Once the download files have been copied, run the following command to build the system:

```
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-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/opencl-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/opencl-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"

```

NATIVESBSTRING = "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-fileystems
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"
NATIVESBSTRING = "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-myrir-bsp/recipes-bsp/ti-sci-fw/ti-sci-fw\_git.bb:do\_populate\_lic after /media/home/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 - myrir - BSP/recipes - BSP/u - boot/u - the 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 - the boot - ti - staging\_2023. 04. Bb:

After the build is complete, the compiled image is in the "*build/arago-tmp-default-glibc/deploy/images/myd-am62x*" directory.

Table 3-1. List of image files

Compile the generated files	Description
myir-image-full-my-d-am62x.wic.xz	Full image
Image	Kernel image
myd-ym62x-6231.dtb	Suitable for hdmi display with myc-ym6231
myd-ym62x-6252.dtb	Suitable for hdmi display with myc-ym6252
myd-ym62x-6254.dtb	Suitable for hdmi display with myc-ym6254
tispl.bin	Cortex-A53 SPL mirror, launched by Cortex-A53 Boot U-Boot
tiboot3.bin	Cortex-R5F SPL mirror, initialized by Cortex-R5F to Cortex-A53 and guided Cortex-A53 to boot.
u-boot.img	U-Boot starts mirroring. Load kernel boot

The resulting image file can be updated as described in Section 4.1.



### 3.4. Build the SD card burner image

To meet the needs of producing burned images to eMMC, MYIR developed the burning method suitable for mass production. The system to be burned is flushed into the onboard eMMC through the system in the SD card. Here, the burner image resource has been integrated into the Yocto project, so users can use the yocto project to build the burner image.

#### 1) Execute the environment variable setup script

The syntax for a script to set the compilation environment is as follows:

```
myir@myir-server1: ~/myd-ym62x-yocto$ cd build
myir@myir-server1: ~/myd-ym62x-yocto/build$ . conf/setenv
```

#### 2) Build the myir-image-full image

Since the sd card burner image is to copy the complete system (myir-image-full) to some directory, it is necessary to build the myir-image-full image before building the sd card burner image. Please refer to Section 3.3 for construction details.

#### 3) Build the sd card burner image

- Build the image

Build the sd card burner image by running the following command:

```
myir@myir-server1: ~/myd-ym62x-yocto/build$ bitbake -c cleansstate fac-burn-
emmc-full
myir@myir-server1: ~/myd-ym62x-yocto/build$ bitbake -c cleansstate myir-imag
e-burn
myir@myir-server1: ~/myd-ym62x-yocto/build$ bitbake fac-burn-emmc-full
myir@myir-server1: ~/myd-ym62x-yocto/build$ bitbake myir-image-burn
```

Create the image file tmp/deploy/images/myd-ym62x/myir-image-burn-myd-ym62x.wic.bz2 and update it as described in section 4.2.

### 3.5. Build the SDK

Yocto provides the ability to build out the SDK toolchain function, which is used by the bottom or upper application developers to use the toolchain and related header files or library files, eliminating the need for users to manually make or compile dependency libraries. It is used to compile u-boot and linux kernel code, with target system header files and library files, which is convenient for application developers to transplant applications on target devices. Here's how to build the toolchain SDK.

In this section, we'll be just going through the SDK provided by MYIR, using the following command to generate the SDK package:

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

After waiting for the build to complete, the sdk installation package will be generated under the path "*build/arago-tmp-default-glibc/deploy/sdk*", see Section 2.2 for installation methods.

```
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. How do I burn a system image

MYD-YM62X series development board designed by MYIR company is based on TI's AM62X series microprocessor, which has various boot modes, so different update tools and methods are needed. Users can choose different ways to update according to their needs. The main ways to update are as follows:

- Make TF card launcher: suitable for research and development debugging, fast start and other scenarios.
- Make TF card burner: suitable for mass production burning emmc.

In addition, because the start mode needs to be adjusted when burning, the user chooses to configure the dial switch according to the table below.

Table 4-1. Dial start mode

	boot mode	SW2[1:4]	SW3[1:3]
ARM start	Boot from TF Card	0001	001
	Boot from eMMC	1001	000

### 4.1. Make a TF card starter

#### 1) Preparation

This step is made under the Windows system.

- TF card (no less than 8GB)
- MYD-YM62X Full image
- Make image Tool Win32DiskImager-1.0.0-binary.zip (path: /03-Tools/tools)

Table 4-2. List of mirror packages

Mirror name	Bag name
myir-image-full	myir-image-full-myd-am62x.wic.xz

#### 2) Make TF card startup (Taking myir-image-full system as an example)

- Unzip the full image

Right click on window and unzip it.

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

- **Write the image file to the Micro SD Card**

Insert the TF card into your computer using a card reader, double-click to open Win32DiskImager.exe to read out the USB disk partition, and click the arrow direction to load the image file.



Figure 4-3. Tool configuration

Select the system package and click Open, as shown by the red arrow in the figure below.

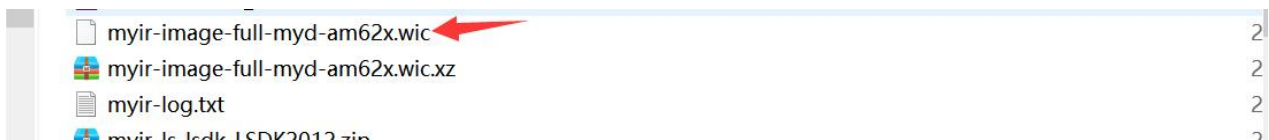


Figure 4-4. Select Mirror

Just click the "Write" button once the image is loaded, and a warning will pop up. Click "Yes" to wait for the writing to complete.



Figure 4-5. Write instructions

Wait for the write to complete, about 3-4 minutes to complete, this speed depends on the TF read and write speed.

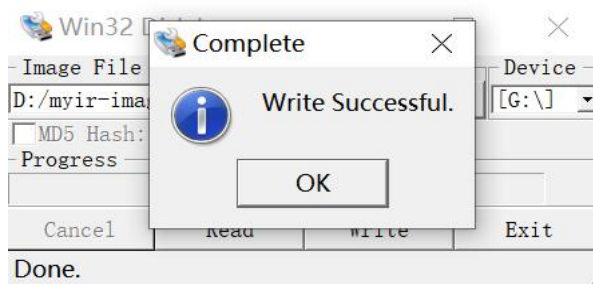


Figure 4-6. Burn finish

- **Check if the burn was successful**

When the writing is completed, you can use this TF to start and insert the TF into the TF card slot of the development board; And adjust the dial switch to TF card start (please refer to Table 4-1) to start the system.

## 4.2. Make a TF card burner

To meet the needs of production burning, MYIR developed burning methods suitable for mass production. Please follow the following steps to complete the specific production process.

### 1) Make the TF card burner image

Make a TF card burner image (myir-image-burn-my-d-am62x.wic), see Section 3.4 for details.

### 2) Write the burner image into the TF card

Make the TF card starter is to write the sd card burn image into the TF card, refer to Section 4.2.

### 3) Burn the image to eMMC

- **There is no mirroring in eMMC**

Set the TF card mode to start (refer to Table 4-1), insert the TF into the TF card slot (J2) of the development board, and plug it in. After the system starts, it will automatically burn and write the image to eMMC. You can connect debug serial port (J3) to check the burning status.

- **eMMC has mirroring**

Set up eMMC mode to start (refer to Table 4-1), plug the TF into the TF card slot of the development board, and plug it in. When using TF card to burn the image to eMMC, first confirm whether eMMC has burned the system. If the system has been burned, it will be started from eMMC by default. You need to use the following command to destroy the contents of eMMC:

```
myir@myir-server1:~$ dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=5; mmc  
bootpart enable 0 1 /dev/mmcblk0
```

Then press the reset button or repower, because the eMMC content has been destroyed at this time, it will automatically boot from the TF card, and then automatically burn the image to the eMMC.

## 5. How to modify the board level

### support package

The previous chapters have given a relatively complete description of the complete process of building a system image to run on the MYD-YM62X development board based on the Yocto project and burning the image to the development board. This section will explain the hierarchical BSP created by MYIR, but also describe how to compile the kernel, U-Boot, and download the kernel and uboot through the standalone environment and the Yocto project.

#### 5.1. Introduction to the meta-bsp layer

The Yocto project's "layer model" is a development model for embedded and IoT Linux creation that distinguishes the Yocto project from other simple build systems. The layer model supports both collaboration and customization. Layers are repositories containing relevant instruction sets that tell the OpenEmbedded build system what to do.

The meta-myir layer is based on the layer suitable for the MYD-YM62X development board established in the official meta-ti layer of TI, which contains various metadata and recipes for BSP, GUI, distribution configuration, middleware or application in the meta-bsp layer below it. On the basis of this "layer model", users can adapt to the hardware designed based on the MYD-YM62X development board, customize their own applications, and build their own system images. This section mainly introduces the meta-bsp layer below the meta-myir layer, which contains the specific content as follows:

```
myir@myir-server1:~$ tree -a -L 1 myd-ym62x/sources/meta-myir/meta-bsp/  
meta-bsp/  
myir@myir-server1:~/AM62/tisdk/sources$ tree -a -L 1 meta-myir/meta-myir-bsp/  
/  
meta-myir/meta-myir-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

### Partial layer Notes:

Table 5-1.meta-bsp Layer Content Description

Source Code and Data	Description
conf	Conf Contains current layer path information and machine software configuration
recipes-bsp	Contains configuration information like atf, uboot as well as firmware
recipes-kernel	Contains resources for the linux kernel and third-party firmware resources
recipes-myr	Contains configuration information for the file system

When porting a system, it is important to pay attention to the recipes-bsp part which is responsible for hardware initialization and system boot, and the recipes-kernel part which is responsible for the kernel and driver implementation of the Linux system.

## 5.2. Board level support package introduction

In order to adapt to the user's new hardware platform, you first need to understand what resources are provided by MYIR's MYD-YM62X development board. For specific information, you can see the MYD-YM62X SDK Release Notes. In addition, we have also sorted out a list of some files that need to be changed in each part of BSP, so that users can find and modify them. The specific content is shown in the table below:

Table 5-2. Adding configuration information

Projects	Equipment tree	Instructions
U-boot	arch/arm/dts/myb-am62x-dev.dt arch/arm/dts/myb-am62x-dev-u-boot.dtsi	Resource Device Tree
Kernel	arch/arm64/configs/defconfig	Kernel configuration table
	arch/arm64/boot/dts/myir/myd-ym62x-6254.dts	Device tree for myd-y6254
	arch/arm64/boot/dts/myir/myd-ym62x-6252.dts	Device tree for myd-y6252
	arch/arm64/boot/dts/myir/myd-ym62x-6231.dts	Device tree for myd-y6231
	arch/arm64/boot/dts/myir/myd-ym62x-base.dts	Basic Resource device tree
	arch/arm64/boot/dts/myir/myd-ym62x-lvds-dual.dtso	dtso file for dual-way lvds
	arch/arm64/boot/dts/myir/myd-ym62x-lvds.dtso	dtso file for single-way lvds

The normal startup process for MYD-YM62X is as follows:

- The rom of the mcu is started first when it is powered on. The ROM is a small piece of ROM or write-protected flash memory embedded inside the processor chip. It contains the first code that the processor executes when it is powered on or reset. Depending on the configuration of certain ribbon pins or internal fuses, it can decide from where to load the next part of the code to be executed and how or whether to verify its correctness or validity. But with processors based on K3 architecture, ROM only supports boot through MCU(R5), that is, after power-on reset, MCU detects the reset release signal and starts executing ROM code of MCU.

- The ROM of the MCU requests the TIFS service core to load and verify the SPL image tiboot3.bin of the MCU and load the system firmware to the TIFS core. The R5 SPL image is then executed to initialize the DDR configuration and load the A53 SPL image tislpl.bin and R5 firmware. Then it sends a request to the TIFS core to start A53 and jump to the device management firmware image to execute the device configuration, monitoring, communication and other services.
- The TIFS core sends a reset and release signal to the A53 core. After receiving the reset and release signal, the A53 starts to execute the ATF or OPTEE firmware. ATF and OP-TEE are two independent firmware with different functions and purposes. ATF is mainly used in the boot process of a device to provide a secure boot and run environment and protect the device from attacks. While OP-TEE is used for security sensitive applications to provide a trusted execution environment and protect sensitive data and operations. This step can be added or removed according to the customer's requirements.
- Start the SPL image of A53. spl mainly copies Uboot to run in external memory, so it itself runs in internal memory.
- The UBOOT image is started, and the proper system software and hardware environment is established to prepare for the final call to the operating system kernel.

The following sections focus on the user flow based on the Bootloader, Kernel and Yocto source code and data we provide.

### 5.3. Compilation and Update of Bootloader

The AM62X platform is divided into R5 bootloader and A53 bootloader. The R5 boot is generated according to different u-boot configurations, while the A53 boot requires a combination of U-boot, OPTEE, and ATF. The boot list is as follows:

Table 5-3. List of boot images

Binaries	Instructions
tiboot3.bin	R5 core boot image, responsible for DDL configuration, A53 boot, R5 firmware loading, etc. Generated by U-boot
tislpl.bin	A53's boot image, responsible for providing a secure boot environment as well as loading u-boot. It is jointly generated by U-boot, Atf and Optee



u-boot.img	U-boot starts the image, sets up the proper running environment, and gets the kernel ready to load
------------	--

Here's how to update and compile the AM62X boot image.

### 5.3.1. How to compile bootloader in a standalone environment

#### 1) Initializing the SDK

Before compiling the bootloader separately, you need to initialize the cross-compile toolchain.

```
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
```

After enabling compile chaining, you need to set the source code paths of a few components:

```
$ 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) Prepare bootloader source code

Users can find the bootloader source code in MYIR's ROM material in the 04-sources/ directory and copy it into your working directory and unpack the file.

As create and go to the working directory myd-ym62x-bsp. Alternatively, you can download the source from github using the following command:

```
myir@myir-server1:~$ mkdir myd-ym62x-bsp
myir@myir-server1:~$ cd myd-ym62x-bsp
myir@myir-server1:~/myd-ym62x-bsp$ git clone https://github.com/MYIR-TI/myir-ti-uboot.git -b myd-ym62x-uboot-2023.04
```

### 3) How to compile tiboot3.bin

tiboot3.bin is an SPL file for R5, which is generated by uboot and compiled as follows:

- **Enable environment variables**

See Section 5.3.1 Initializing the SDK. Here's where I enable uboot:

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

- **Compile the R5 boot image**

```
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) How to compile tispl.bin

tispl.bin is the boot image of A53 and needs to be generated by u-boot, optee and atf together.

- **Compiling optee**

Go to the myir-ti-optee directory and run the following command to compile:

```
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
```

- **Compile atf**

Go to the myir-ti-atf directory and execute the following command to compile:

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

- **Compile tispl.bin and u-boot.img**

Go to the myir-ti-uboot directory, set the BINMAN\_DIR variable (the path is determined by the user's directory), and run the following command to compile:

```
myir@system1:~/AM62/myir-ti-bootloader/myir-ti-uboot$ export BINMAN_DIRS
=/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=$TFA_DIR/build/k3/lite/release/bl31.bin TEE=$OPTTEE_DIR/out/arm-plat-k3/core/tee-pager_v2.bin O=$UBOOT_DIR/out/a53 BINMAN_INDIRS=$BINMAN_DIRS
```

### 5.3.2. Compile u-boot under Yocto project

After the user has modified U-boot's code, Yocto can also be used to build the entire image. At this point, the modified source code needs to be committed to the local git repository, and the pull address (SRC\_UR) and commit value (SRCREV) of the corresponding source code of the meta-layer need to be changed. In order for the recipe to find and fetch the local uboot source code, the reference example is as follows:

#### 1) Viewing commit values

Go to the uboot source code Modify the uboot source code, and view the commit values, shown as red strings:

```
myir@myir-server1:~$ cd myd-ym62x-bsp/myir-ti-uboot/
Commit f4e7b03f15da6cd88d31d29a2b9b7e0812534554 (HEAD -> ti-u-the-bo
ot - 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, the tag: 09.00.00.006
```

## 2) Change the source location

The location of the uboot source code in yocto is specified in *"sources/meta-myir/meta-myir-bsp/recipes-bsp/u-boot/U-boot-ti.inc"* file, and the change is shown in red as follows:

```
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=${B
RANCH}"
```

- UBOOT\_GIT\_URI: uboot code download location
- RRANCH: Branch name
- UBOOT\_GIT\_PROTOCOL: Download protocol, this is local download, so set to "file"

## 3) Initialize SDK environment variables

Go to the uboot directory and initialize the environment variables:

```
myir@myir-server1: ~/myd-ym62x-yocto$ cd build
myir@myir-server1: ~/myd-ym62x-yocto/build$ . conf/setenv
```

#### 4) Compile uboot

To compile uboot, run the following command:

```
myir@myir-server1:/media/myir/myd-ym62x/build$ bitbake -c cleansstate u-boot
-ti-staging
myir@myir-server1:/media/myir/myd-ym62x/build$ bitbake u-boot-ti-staging
```

Generate the image file as "*build/arago-tmp-default-glibc/deploy/images/myd-am62x/tispl.bin*" .

#### 5) Verify that uboot compiles successfully

Once you have modified uboot, you need to confirm that uboot has been updated by running the following command:

```
myir@myir-server1:/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)
```

From the "gf4e7b03f15" string above, the modified version of uboot is consistent with the previous character of SRCREV, that is, uboot has been updated.

### 5.3.3. How to update bootloader separately

#### 1) Burn the image to eMMC

Copy the compiled image to the development board to view the corresponding partition:

```
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
```

To copy the image to the appropriate disk, execute the following command:

- **Update tiboot3.bin separately**

```
myir@myir-server1:~$ 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
```

- **Update tispl.bin separately**

```
myir@myir-server1:~$ 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
```

Update u-boot.img separately:

```
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. Kernel Compilation and Update

### 5.4.1. How to build Kernel in standalone environment

#### 1) Get the source code

Create and go to the BSP working directory myd-ym62x-bsp, then download the source code using the following command:

```
myir@myir-server1:~$ mkdir myd-ym62x-bsp
myir@myir-server1:~$ cd myd-ym62x-bsp
myir@myir-server1:~/myd-ym62x-bsp$ git clone https://github.com/MYIR-TI/myir-ti-linux.git -b myd-am62x-linux-6.1.46
```

### 5.4.2. Compile the Kernel in a separate cross-compile environment

#### 1) Load the SDK environment variables into the current shell

Before compiling the Kernel separately, you need to declare the compilation chain, as described in Section 2.2.

```
myir@myir-server1:~$ source ~/AM62/tool_chain/myir/environment-setup-aarch64-oe-linux
```

#### 2) Go to the Kernel source, configure and compile

- Compile as a whole

Go to the Kernel source and use the following command to configure and compile the source:

```
myir@myir-server1:~$ cd myd-ym62x-bsp/myir-ti-linux
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-linux$ make distclean
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-linux$ make defconfig ti_arm64_prune.config
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-linux$ make Image modules dtbs -j 16
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-linux$ mkdir ../test
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-linux$ make modules_install INSTALL_MOD_PATH=../test
```

The resulting Image is in `"arch/arm64/boot/Image"` ; The generated dtb file is at `"arch/arm64/boot/dts/myir"` ; The generated module is in the test directory, and there is a link to the kernel source that needs to be deleted before it can be copied to the development board.

- **Compile the device tree separately**

```
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-linux$ make dtbs
```

DTB file generated in the `arch/arm64 / boot/DTS/myir/` directory.

### 5.4.3. Compile the Kernel under the Yocto project

After the user has modified the Kernel code, Yocto can also be used to build the entire image. At this time, the pull address (SRC\_URI) and commit value (SRCREV) of the corresponding source code of the meta-layer are modified. So that the recipe can find and fetch the local Linux source code, the reference example is as follows.

#### 1) Viewing commit values

Go to the source directory, modify the Kernel source code, and view the commit values, shown as red strings:

```
myir@myir-server1:~$ cd myd-ym62x-bsp/myir-ti-linux
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-linux$ git log
Commit 4 c20dc4003c94a10472fab1e28cefbda5335ab41 (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) Change the source location

In yocto, specify the location of the kernel source in the "*sources/meta-myr/meta-myr-bsp/recipes-kernel/linux/linux-ti-staging\_6.1.bb*" file, and change it as follows to show in red:

```
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: the Kernel code download location
- SBRANCH: Branch name
- SRCREV: Automatically identify to commit value
- KERNEL\_GIT\_PROTOCOL: Download the protocol locally

### 3) Initialize SDK environment variables

To compile, run the following command to initialize the environment:

```
myir@myir-server1: ~/myd-ym62x-yocto$ cd build
myir@myir-server1: ~/myd-ym62x-yocto/build$ . conf/setenv
```

### 4) Compile the Kernel

To compile the kernel, execute the following command:

```
myir@myir-server1: ~/myd-ym62x-yocto$ bitbake -c cleansstate virtual/kernel
myir@myir-server1: ~/myd-ym62x-yocto$ bitbake virtual/kernel
```

To generate the Image file for the "build/arago-tmp-default-glibc/deploy/images/myd-am62xImage" .

#### 5.4.4. How do I update the Kernel and device tree separately

##### 1) Burn the image to the TF card

Insert the TF card into the host computer with a card reader, and the boot partition will be automatically mounted to the host computer (in fat format). Just copy the kernel and device tree to this directory, such as:

```
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) Burn the image to eMMC

Copy the compiled Image (Image) and the device tree file (\*.dtb) to the root-mmcbk0p2 partition of the development board, either using scp or a USB key.

Start the development board, mount the rootfs partition, and directly copy the image to the boot directory of the rootfs partition. The steps are as follows:

Mounting the file system partition:

```
root@myd-y62x:~# mount /dev/mmcbk0p2 /mnt
```

Copying the kernel to the boot directory of the file system partition using SCP on the host (SCP requires network connectivity):

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

Copying device tree to the system partition directory using SCP on the host:

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

Execute synchronization command on the development board:

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

##### 3) Update Module Files to eMMC

In section 5.4.2, step 2) Enter Kernel Source Code, Configure, and Compile, the module files are compiled and generated in the "test" directory located in the upper-level directory of the kernel source code. This section explains how to update the module files to the eMMC.

Package and compress the "test" directory into a tar.gz file:

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

Copy the tar.gz file to the root directory of the file system using SCP:

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

On the development board, extract the contents of the "test" package:

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

Copy the "lib/modules" directory from the "test" directory to the root directory of the development board's "/lib/modules":

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

Remove the "test" directory and synchronize to save the changes:

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

Restart the development board and use "lsmod" to check the module loading status:

```
root@myd-y62x:~/everest-utils/docker# lsmod
Module                Size  Used by
xt_nat                 16384  6
xt_tcpudp              16384  10
veth                   32768  0
```

# 6. How to adapt to your hardware platform

## 6.1. How to create your own machine

During development, users sometimes need to create custom board configurations. This section will walk you through an example of how you can create your own machine.

### 6.1.1. Create a board configuration in Yocto

#### 1) Select something like machine file

Copy a similar machine file and rename it to a designated name for your board. For example, myd-am62x development board machine file “myd-am62x.conf” in “*sources/meta-myir/meta-myir-bsp/conf/machine*” directory, enter this directory, machine file is as follows:

```
myir@myir-server1:~/myd-ym62x-yocto$ cd myd-ym62x-yocto/sources/meta-myir/meta-myir-bsp/conf/machine
myir@myir-server1:~/myd-ym62x-yocto/sources/meta-myir/meta-myir-bsp/conf/machine$ ls myd-am62x.conf
myd-am62x.conf
```

#### 2) Copy and rename

After finding a similar machine file, copy and rename your own machine file, such as:

```
myir@myir-server1:~/myd-ym62x-yocto/sources/meta-myir/meta-myir-bsp/conf/machine$ cp myd-am62x.conf myd-test.conf
myir@myir-server1:~/myd-ym62x-yocto/sources/meta-myir/meta-myir-bsp/conf/machine$ ls myd-test.conf
myd-test.conf
```

### 3) Modify the machine file

Once you've created the machine file, change the machine property of the “*build/conf/local.conf*” file as follows:

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

### 4) Compile and test

Once the machine file is created, you can compile the minimum image tests by compiling and downloading the tests, running the following command:

```
myir@myir-server1:~/myd-ym62x-yocto$ cd build
myir@myir-server1:~/myd-ym62x-yocto/build$ . conf/setenv
myir@myir-server1:~/myd-ym62x-yocto/build$ bitbake myir-image-core
```

After compiling, the generated image “*build/arago-tmp-default-glibc/deploy/images/myd-test/myir-image-core-myd-test.wic.xz*”, copy and extract it, then follow Section 4.2 to burn it to the TF card :

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

## 6.1.2. Creating board configuration file in uboot

In the development process, users generally need to create their own board profile according to their own board requirements. This section will demonstrate how to create their own board profile step by step through a simple example.

### 1) Creating a board

When creating their own board configuration, users can create their own board configuration file in the corresponding board subdirectory by copying and renaming. Generally, the board configuration file is in the board directory of the uboot source code. Go to the myir subdirectory under the uboot source code board subdirectory and copy the myc\_am62x folder to test\_am62x as follows:

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

Go to the test\_am62x directory and change myc\_am62x.env to test\_am62x.env.

## 2) Make system configuration file

- **Modify the newly created board Kconfig**

Go to the test\_am62 directory and modify the Kconfig file with the following red marker:

```
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_SYSPFW
    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
```

- **Create the header file for the new board**

From the source directory, go to include/configs and copy myc\_am62x.h to test\_am62x.h, as follows:

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

Note: Since the name of `SYS_CONFIG_NAME` was changed to `test_am62x`, this header file is changed to `test_am62X.h`.

- **Customize the system board subconfiguration file**

From the source root directory, go to configs directory, copy “`myc_am62x_a53_defconfig`” to “`test_am62x_a53_defconfig`”, Copy “`myc_am62x_r5_defconfig`” to “`test_am62x_r5_defconfig`” as follows:

```
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
```

Then modify the “`test_am62x_a53_defconfig`” file as follows:

```
#CONFIG_TARGET_MYC_AM62X_A53_DEV=y
CONFIG_TARGET_TEST_AM62X_A53_DEV=y
```

Modify the “`test_am62x_r5_defconfig`” file as follows:

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

- **Enable the Kconfig file**

After completing the above steps, you also need to enable Kconfig and modify the file “`arch/arm/mach-k3`”, adding the following content:

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

- **Modifying the device tree**

Modify the “`myb-am62x-dev-binman.dtsi`” device file as shown below in red:

```
//#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"
```

After these steps, you have basically customized the board. If you want to build with yocto, you also need to modify the device tree configuration information in the machine as described in section 6.1.1.

### 3) Compiling

Execute the following command to compile:

```
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-uboot$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- distclean
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-uboot$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- test_am62x_r5_defconfig
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-uboot$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- -j10
```

## 6.2. How do you create your device tree

### 6.2.1. Board Level Device Tree

A device tree is a data structure that describes on-chip and off-chip device information in a unique syntax format. For example, part of the device information structure is stored in the device tree file. uboot or the kernel eventually compiles its device tree into a dtb file, and obtains board-level device information by parsing the dtb during use.

#### 1) Introduction of myir-ti-uboot device tree

The following information list of MYD-YM62X's uboot parts of the device tree is for user reference:

Table 6-1.MYD-YM62X U-boot device tree list

Projects	Equipment tree	Description
U-boot	myb-am62x-r5-dev.dts	Device tree for MYD-AM62X R5
	myb-am62x-dev.dts	MYD-AM62X A53's device tree

	myb-am62x-dev-binman.dtsi	By using the binman tool, you can configure and generate an AM62x platform-specific U-Boot binary file, which contains information about the bootloader, the Device Tree, the boot script, and more
	myb-am62x-dev-u-boot.dtsi	For myb-am62x series specific configuration

When compiling the MYD-YM62X board uboot source code, all relevant dts/dtsi files are merged to generate the device tree “*myb-am62x-r5-dev.dtb*” for stage R5 and “*myb-am62x-dev.dtb*” for stage uboot by default.

## 2) Introduction of myir-ti-linux device tree introduction

The approximate hierarchy of the device tree in myir-ti-linux is: k3-am62-main.dtsi -> k3-am62.dtsi -> k3-am625.dtsi -> myd-ym62x series device tree. Below is the detailed information list of each part of the device tree for MYD-YM62X:

Table 6-2.MYD-YM62X Linux device tree list

Projects	Equipment tree	Description
Kernel	k3-am62-main.dtsi	Contains all resource configurations for the myd-ym62x series
	myd-ym62x-6231.dts, myd-ym62x-6252.dts, myd-ym62x-6254.dts	HDMI display by default for different device tree configurations of myc-ym62x core board
	myd-ym62x-hdmi-audio.dtso, myd-ym62x-lvds-dual.dtso, myd-ym62x-lvds.dtso, myd-ym62x-lvds-j6.dtso	Peripheral resource supplement for the base device tree, which can be loaded dynamically

## 6.2.2. Add your board level device tree

### 1) Add board level device tree under the uboot

- **Select a similar device tree file**

Go to the uboot source code *arch/arm/dts* directory, copy “myb-am62x-dev.dts” file and rename it to “myb-am62x-test.dts”, as shown below:

```
myir@myir-server1:~/myd-ym62x-bsp/myir-ti-uboot$ cp arch/arm/dts/myb-am62x-dev.dts arch/arm/dts/myb-am62x-test.dts
myir@myir-server1:~/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
```

- **Modify the device tree Makefile**

Next, modify the makefile of the device tree in the directory, and add the following contents:

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

- **Modify the board configuration file**

Then change the R5 configuration file "test\_am62x\_r5\_defconfig" configuration file default device tree as follows:

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

## 2) Kernel creates the device tree

MYIR series device tree is in "*arch/arm64/boot/dts/myir*" directory. Users can also copy myir device tree and rename it in this directory. The method is similar to uboot. Please refer to the above steps.

## 6.3. How to configure CPU function pins according to your hardware

Realizing the control of a function pin is one of the more complex system development processes, which includes the pin configuration, the development of the driver, the implementation of the application and so on. This section will explain the control of the function pin by an example.

### 6.3.1. Method of GPIO pin configuration

The IO pins of MYD-YM62X board are generally defined in

*"arch/arm64/boot/dts/myir/myd-ym62x-common.dts"* device tree file. The pin configuration macros for TI are defined in the file

*"arch/arm64/boot/dts/myir/k3-pinctrl.h"*. Please refer to the am625.pdf datasheet for details. An example of configuration is described below.

#### 1) Check out the pinctrl configuration rules

TI pinctrl configuration format:

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

Parameter Description:

- AM62X\_IOPAD represents the pin configuration of the A53 domain and AM62X\_MCU\_IOPAD represents the pin configuration of the R5 side
- pa: PADCONFIG register address Multiplexes the register offset address
- val: Configuration for attributes such as pin pull up and down
- muxmod: Indicates which functions a pin can be multiplexed into

For example, the pin configuration is as follows:

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

Set GPIO0\_36 to the gpio input function as shown in the datasheet manual.

## 2) Configure gpio on the device tree

The following is an example of how to apply for and allocate device hardware resources in the DTS file. For example, users can define gpio device nodes in the "myd-y62x-common.dtsi" DTS file, as follows:

```
gpioctr_device {  
    compatible = "myir,gpioctr";  
    #pinctrl-names = "default";  
    #pinctrl-0 = <&pinctrl_gpio_blue>;  
  
    status = "okay";  
    gpioctr-gpios = <&main_gpio0 31 0>;  
};
```

## 6.4. How to use your own configured pins

The pins that we configure in the device tree of u-boot or Kernel can be used in the corresponding u-boot or Kernel to control the pins.

### 6.4.1. GPIO pins are used in U-boot

#### 1) uboot terminal command control

uboot can use commands directly to control GPIO Settings. To set GPIO0\_31, use the following command:

```
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
```

It can be measured with a multimeter to the J11 upper 5 pin voltage pull up/pull down.

#### 2) Uboot code control

Users can implement IO functions directly in the code during the uboot phase, such as the following phy power reset control.

- The uboot code controls the phy reset pin

In the "*myir-ti-uboot/board/myir/myc\_am62x/som.c*" file, add the following lvds power control code:

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

### 1) How to use independent GPIO driver

Now that we have defined the gpio node information in the first device tree example in Section 6.3.1, we will use the kernel driver to control the GPIO (set the GOIO0\_31 pin to 1 and 0, and use a multimeter to test pin levels if needed).

```
//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. Determine major number */
static int major = 0;
static struct class *gpioctr_class;
static struct gpio_desc *gpioctr_gpio;
```

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

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

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

    gpiod_set_value(gpioctr_gpio, status);

    return 1;
}

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

    return 0;
}

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

```

        return 0;
    }

/* Define our own file_operations struct */
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,
};

/* Get the GPIO from platform_device
 * Tell the kernel the file_operations structure to register drivers
 * /
static int chip_demo_gpio_probe(struct platform_device *pdev)
{
    /* As defined in the device tree: gpiocr-gpios=<... >      * /
    gpiocr_gpio = gpiod_get(&pdev->dev, "gpiocr", 0);
    if (IS_ERR(gpiocr_gpio)) {
        dev_err(&pdev->dev, "Failed to get GPIO for led\n");
        return PTR_ERR(gpiocr_gpio);
    }

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

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

```

```

        return PTR_ERR(gpioctr_class);
    }

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

    return 0;
}

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

    return 0;
}

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

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

```

```
};

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

    return err;
}

/* An entry function should have an exit function: this exit function is called when
the driver is unmounted
* Uninstall platform_driver
* /
static void __exit gpio_exit(void)
{
    platform_driver_unregister(&chip_demo_gpio_driver);
}

/* Other improvements: Provide device information, automatically create device nodes */
module_init(gpio_init);
module_exit(gpio_exit);

MODULE_LICENSE("GPL");
```

Compiling driver code into modules using a separate Makefile can also be configured directly into the kernel. Let's take compiling to modules as an example.

## 2) The driver example compiles into a separate module

Add "gpioctr.c" to your working directory and copy the driver code, while writing a separate Makefile:

```
# modify KERN_DIR
```

```
# #KERN_DIR = # The directory of the kernel source used by the board
KERN_DIR = /media/myir/myd-ym62x-bsp/myir-ti-linux
obj-m += gpioctr.o

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

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

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

Then set up the host terminal window toolchain environment:

```
myir@myir-server1:/media/myir/myd-ym62x/gpioctrl$ source ~/AM62/tool_chain
/myir/environment-setup-aarch64-oe-linux
```

To generate the gpioctr.ko driver module file, run the make command:

```
myir@myir-server1:/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/gpioctr.o
MODPOST /media/myir/myd-ym62x/gpioctrl/Module.symvers
CC [M] /media/myir/myd-ym62x/gpioctrl/gpioctr.mod.o
LD [M] /media/myir/myd-ym62x/gpioctrl/gpioctr.ko
make[1]: Leaving directory '/media/myir/myd-ym62x-bsp/myir-ti-linux'
```

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



### 6.4.3. How to control a GPIO in Userspace

The architecture of the Linux operating system is divided into user mode and kernel mode (or userspace and kernel). The user mode is the active space of the upper application, and the execution of the application must rely on the resources provided by the kernel, including CPU resources, storage resources, I/O resources, etc. In order to enable the upper applications to access these resources, the kernel must provide an interface for the upper applications to access: that is, system calls.

Shell is a special application program, commonly known as the command line, is essentially a command interpreter, it passes down the system call, on a variety of applications. Using Shell scripts, usually a few short lines of Shell scripts can achieve a very large function, the reason is that these Shell statements are usually a layer of encapsulation of system calls. In order to facilitate the user and the system interaction.

In this section, we will describe two basic ways to control GPIO pins in user mode.

- Shell commands
- System calls

#### 1) The Shell implements the pin control

The Shell control pin is essentially implemented by calling the file operation interface provided by Linux. This section does not give a detailed explanation, but you can see the description in Section 3.1 of the MYD-YM62X Software Evaluation Guide.

#### 2) The system call realizes the pin control

The operating system provides a set of "special" interfaces that the user program calls. Through this set of "special" interfaces, user programs can obtain services provided by the operating system kernel. For example, users can request the system to open files, close files, or read or write files through file system-related calls, and can obtain the system time or set timers through clock related system calls.

At the same time, pins are also resources, which can also be controlled by system calls. In 6.4.2 we have completed the implementation of the driver of the pin, that is, we can control the pin controlled by the driver by system calls.

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

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

```

    }

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

    close(fd);

    return 0;
}

```

Copy the above code into a gpiotest.c file and load the SDK environment variables into the current shell:

```
myir@myir-server1:/media/myir/myd-ym62x/gpiotest$ source ~/AM62/tool_chain
/myir/environment-setup-aarch64-oe-linux
```

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

```
myir@myir-server1:/media/myir/myd-ym62x/gpiotest$ $CC gpiotest.c -o gpiotest
```

The executable file through the network (scp, etc.), u disk and other transmission media copy to the development board of /usr/sbin directory, you can enter the command in the terminal can run directly (on means high, off means low).

```

root@myd-ym62x:~# gpiotest /dev/myir_gpiotest0 on
[643.651418] /media/myir/myd-ym62x/gpiotest/gpiotest.c gpio_drv_write line 39
[643.658779] /media/myir/myd-ym62x/gpiotest/gpiotest.c gpio_drv_close line 56
root@myd-ym62x:~# gpiotest /dev/myir_gpiotest0 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
```

The voltage that can be measured with a multimeter to pin 3 on J11 is pulled up and down.

## 7. How to add your application

Linux application porting is usually divided into two phases, development and debugging phase and production deployment phase. In the development and debugging phase, we can use the SDK built by MYIR to cross-compile the application we have written and then remotely copy it to the target host for testing. Production deployment involves writing a recipe file for the application and using Bitbake to build a production image.

### 7.1. Makefile-based apps

A Makefile is simply a document that defines a set of compilation rules. It records the details of how the original code is compiled! Once the Makefile is written, only one make command is needed, and the whole project is completely automatically compiled, which greatly improves the efficiency of software development. In the development of Linux programs, whether kernel, driver, application, Makefile has been widely used.

make is a command tool, is a command tool to explain the instructions in the makefile. It can simplify the instructions given in the compilation process, when executing make, make will search for Makefile (or makefile) this text file in the current directory, and execute the corresponding operation. make will automatically determine whether the original file has been changed, so as to automatically recompile the changed source code.

A practical example (keystroke control on the MYD-YM62X development board) will illustrate the process of writing makfiles and executing make. The Makefile has its own set of rules.

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

- A target can be an object file, an execution file, or a label.
- The prerequisites are the files or targets that are required to generate the target.

- command is the command that make needs to execute.

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

Some parameter notes:

- \$(CURDIR) : Indicates Makfile current directory full path
- \$(notdir \$(path)) : This means that the path directory is removed from the path name, and only the current directory name is left. For example, the current Makefile directory *is* `/home/myir/myd-ym62x/key_led`, which becomes `TARGET = key_led`
- \$(patsubst pattern, replacement,text) : Replace characters in text that match the format "pattern" with replacement, such as `$(patsubst %c, %o, $(shell ls *.c))`, to list the files in the current directory with the.c suffix and then replace them with the.o suffix
- CC: The name of the C compiler
- CXX: The name of the C++ compiler
- clean: is a convention target

The Key implementation code is as follows:

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

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/* ./key_led /dev/input/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"); /
/led off

            }

        }

    }

    return 0;
}

```

Use the make command to compile and generate the target\_bin executable on the target machine.

Load the SDK environment variables into the current shell:

```
myir@myir-server1:/media/myir/myd-ym62x/app_test/key_led$ source ~/AM62/tool_chain/myir/environment-setup-aarch64-oe-linux
```

Execute make:

```
myir@myir-server1:~$ make
```

As you can see from the results of the previous command, the compiler used is the one established by setting the CC variable defined in the script. Copy the key\_led\_test executable file to the /usr/sbin directory of the development board through the network (scp, etc.), u disk and other transmission media. Execute the following command, and then press the USER button to see the blue light on and off:

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

## 7.2. Application based on QT

Qt is a cross-platform graphics application development framework that is used on different sizes of devices and platforms and offers different copyright versions for users to choose from. MYD-JX8MMA7 uses Qt version 5.15 for application development. In Qt application development, it is recommended to use QtCreator integrated development environment. Qt application can be developed under Linux PC, which can be automatically cross-compiled into the ARM architecture of development board.

The QtCreator installation package is a binary program that can be installed by executing it directly: `./qt-creator-opensource-linux-x86_64-4.1.0-rc1.run`, for details of installation and configuration, please refer to official website:

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

## 7.3. How to start an application automatically

### 1) Application configuration in Yocto

If you want to use Yocto Project to build the image file in the production deployment phase and include the application, you need to create a recipe for the application. Please refer to the site:

<https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#hello-world-example>

In general, our application also needs to realize the boot itself, which can also be implemented in the recipe. Below with a slightly more complicated FTP service application, for example shows how to use Yocto building contains application-specific production mirror, the FTP service here Proftpd program USES is open source, each version of the source at <ftp://ftp.proftpd.org/distrib/source/>.

Before we write a recipe from scratch, we can look in the current source code repository to see if this application, or a recipe for a similar application, already exists, by doing the following:

```
myir@myir-server1:/media/myir/myd-ym62x/build$ bitbake -s | grep proftpd
```

Proftpd: 1.3.7 c - r0

**Note:** Before executing the bitbake command, make sure you have executed the environment variable setting script for building the Yocto project, see Chapter 3 for details.

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

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

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

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

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

- **Find Yocto's proftpd recipe**

```
myir@myir-server1:/media/myir/myd-ym62x/build$ bitbake -s | grep proftpd
Proftpd: 1.3.7 c - r0
```

**Note:** It can be seen here that the recipe for proftpd already exists in the Yocto project, version 1.3.7c-r0.

- **Compile proftpd separately**

```
myir@myir-server1:/media/myir/myd-ym62x/build$ bitbake proftpd
```

- **Package proftpd to the file system**

Add a line to conf/local.conf:

```
IMAGE_INSTALL_append = "proftpd"
```

- **Rebuild the image**

```
myir@myir-server1:/media/myir/myd-ym62x/build$ bitbake myir-image-core
```

- **Burn the new image**

Once the system is built, you need to re-burn the image and see if the proftpd service is running:

```
root@myd-am62x:~# ps -axu | grep proftpd
```

```
Nobody 0.0 0.0 4776 1636 534? Ss 00:00 0:00 proftpd: (accepting connections)
root 814 0.0 0.0 2832 872 ttyS1 S+ 02:14 0:00 grep proftpd
```

I would like to add a note about FTP account Settings. There are three types of FTP client login accounts: anonymous, regular, and root.

- **Anonymous accounts**

The user name is ftp, no password needs to be set, the user can view the contents of the system `/var/lib/ftp` directory after logging in, and there is no write permission by default. Since the system does not exist `/var/lib/ftp` directory by default, the user is required to create a directory `/var/lib/ftp` on the target machine. In order to try not to modify meta-openembedded, we can implement the `/var/lib/ftp` directory creation by adding the append file "proftpd\_1%.append" to the proftpd recipe.

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

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

- **Ordinary account settings**

Using the commands of `useradd` and `passwd` on the target machine, you can create an ordinary user, and after setting the user password, the client can also log in to the user's home directory with the common account. If you need to include ordinary users when packaging images, you can refer to the site(<https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#ref-classes-useradd>) ,Then rebuild the image. The specific method will not be discussed here.

- **Root account settings**

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

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

```
# systemctl restart proftpd
```

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

## 2) How to start service automatically at boot time

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

```
myir@myir-server1:~$ 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.b
```

1 directory, 8 files

- proftpd\_1.3.7c.b is the recipe for building the proftpd service
- proftpd.service is a bootable service
- Proftpd-basic.init is the startup script for proftpd

The recipe for proftpd\_1.3.7c.b specifies the source code path to obtain the proftpd server and some patch files for that version of the source code:

```
SRC_URI = "ftp://ftp.proftpd.org/distrib/source/${BPN}-${PV}.tar.gz \
file://basic.conf.patch \
```

```
file://proftpd-basic.init \
file://default \
file://close-RequireValidShell-check.patch \
file://contrib.patch \
file://build_fixup.patch \
file://proftpd.service \
"
```

The configuration (do\_configure) and installation procedure (do\_install) for proftpd are also specified in the recipe:

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

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

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

    # create the pub directory
    mkdir -p ${D}/home/${FTPUSER}/pub/
```

```

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

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

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

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

```

```
rm -rf ${D}${mandir}/man1/ftpmail.1
}
```

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

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

“proftpd\_1.3.6.bb” recipe inheritance systemd.class class(Please refer to: layers/openembedded-core/meta/classes/systemd.bbclass). If you want to run the application service in the boot phase, you need to use the default enable variable(SYSTEMD\_AUTO\_ENABLE). For example, the user can set the variable(SYSTEMD\_AUTO\_ENABLE) to start the application service. The example is as follows:

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

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

<https://www.yoctoproject.org/docs/3.1.1/mega-manual/mega-manual.html#selecting-an-initialization-manager> .

The contents of the proftpd service file are as follows:

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

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

[Install]
WantedBy=default.target
```

- After means this service is started after the network is started.
- Type indicates that the way to start is forking.
- ExecStart indicates the program to be started, along with the appropriate parameters.

If you want to learn more about the information systemd please check this web site <https://wiki.archlinux.org/index.php/systemd>.

When adding your own applications, you can also follow the example above to create a recipe, set it to boot, and pack it into your system image. Your recipes are suggested to be placed in the sources/meta-myr/meta-bsp directory.

## 8. Resources

- **Linux kernel open source community**  
<https://www.kernel.org/>
- **Yoto project BSP development guidelines**  
<https://www.yoctoproject.org/docs/3.1.1/bsp-guide/bsp-guide.html>
- **The Linux kernel development manual Yocto project**  
<https://www.yoctoproject.org/docs/3.1.1/kernel-dev/kernel-dev.html>
- **Yocto development guidance**  
<https://www.yoctoproject.org/>

# Appendix A

## Warranty & Technical Support Services

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

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

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

### Service Guarantee

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

### Price

MYIR insists on providing customers with the most valuable products. We do not pursue excess profits which we think only for short-time cooperation. Instead, we hope to establish long-term cooperation and win-win business with customers. So we will offer reasonable prices in the hope of making the business greater with the customers together hand in hand.

### Delivery Time

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

---

## Technical Support

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

### After-sale Service

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

#### Technical support service

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

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

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

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

#### After-sales maintenance service

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

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

- Due to unauthorized weld or dismantle parts or repair the products which has caused the damage of the products or defects of appearance;
- Due to unauthorized installation of the software, system or incorrect configuration or computer virus which has caused the damage of products.

### **Warm tips**

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

### **Maintenance period and charges**

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

### **Shipping cost**

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

### **Products Life Cycle**

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

### **Value-added Services**

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

### **MYIR Electronics Limited**

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

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

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

Phone: +86-755-22984836

Fax: +86-755-25532724

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