MYD-YT113X_Linux

System Development Guide Notes

File Status: [ ] Draft [√] Release	**FILE ID:**	MYIR-MYD-YT113X-SW-DG-EN-L5.4.61
	VERSION:	V1.1[doc]
	AUTHOR:	Nico
	CREATED:	2023-05-01
	UPDATED:	2023-09-05

Revision History

VERSION	AUTHOR	PARTICIPANT	DATE	DESCRIPTION
V1.0	Nico	Licy	20230501	Initial Version, applicable to MYD-YT113X
V1.1	Nico	Licy	20230905	Added "MYD-YT113-I" model

CONTENT

1. Overview

For system building and customization development, the more common ones are Buildroot, Yocto, OpenEmbedded and so on. Among them, the buildroot project uses a lighter and more customized approach to build Linux systems for embedded products. It is not only a tool to create a file system, but also provides a complete set of Linux-based development and maintenance work, so that the bottom of the embedded developers and application developers in the upper layers of the development of a unified framework, to solve the traditional development approach to the fragmented and unmanaged development pattern .

This article mainly introduces the SDK project based on Allwinner T113 processor and the complete process of customizing a complete embedded Linux system on the MYIR core board, including the preparation of the development environment, the acquisition of code, and how to carry out Bootloader, Kernel porting, customizing the root filesystem rootfs to fit the needs of their own applications.

Firstly, we introduce how to build the system image for "MYD-YT113X" development board based on the source code provided by MYIR, and how to download the built image to the development platform. Secondly, for those users who customize their projects based on the "MYC-YT113X" core board, we will focus on how to use this set of SDK to port to the user's hardware platform and the analysis of the key points. Finally, we will pass some actual cases of driver porting and Rootfs customization, so that users can quickly develop their own hardware to meet the system image.

This document does not contain the buildroot project and the introduction of Linux system-related basics, and is suitable for embedded Linux system developers and embedded Linux BSP developers who have some development experience. For some specific functions that users may use in the process of secondary development, we also provide detailed application notes for developers to refer to, see the document list in Table 2-4 of the "**MYD-YT113X SDK Release Notes**" for specific information.

This article applies to the list of development boards and core boards:

Table 1-1.List of Core Boards and Development Boards

Core Board	Development Board
MYC-YT113-S3	MYD-YT113-S3
MYC-YT113-i	MYD-YT113-i

1.1. Software Resources

MYD-YT113X is equipped with an operating system based on Linux kernel version 5.4.61, which provides rich system resources and other software resources. The development board is shipped with cross-compilation tool chain, U-boot source code, source code of Linux kernel and driver modules, as well as various development and debugging tools and application development routines for Windows desktop environment and PC Linux system. Please refer to Chapter 2 Software Information in the "*MYD-YT113X SDK Release Notes*" for detailed information on the included software.

1.2. Document Resources

Depending on the user's use of the development board for each different purpose. A complete SDK package (Software Development Kit) will be provided to customers. The SDK contains different types of documents and manuals such as Release Notes, Getting Started Guide, Evaluation Guide, Development Guide, Application Notes, and Frequently Asked Questions. The specific list of documentation is described in Table 2-4 of the "*MYD-YT113X SDK Release Notes*".

2. Development Environment

This chapter introduces some of the hardware and software environments required for the development process based on the MYD-YT113X development board, including the necessary development host environment, essential software tools, code and data acquisition, etc. Specific preparations will be described in detail below.

2.1. Developing Host Environment

How to build the development environment for Allwinner T1 series processor platform. By reading this chapter, you will understand the installation and use of related hardware tools, software development and debugging tools. And you can quickly build the relevant development environment to prepare for the later development and debugging. Allwinner T1 series processors are SMP multi-core architecture processors with 2 ARM Cortex A7, which can run embedded Linux system and use the common development tools for embedded Linux system.

- **Host Hardware**

The entire SDK package project is built on a development host with high requirements, requiring a processor with a dual-core CPU or higher, 4GB or more memory, and a 100GB hard drive or higher configuration. It can be a PC or server with Linux system installed, or a virtual machine running Linux system, WSL2 under Windows system, etc.

- **Host Operating System**

There are many options for the host operating system to build the buildroot project, generally you can choose to build on the local host with Fedora, openSUSE, Debian, Ubuntu, RHEL or CentOS Linux distributions installed, here the recommended system is Ubuntu 18.04 64bit desktop version (Ubuntu20.04 64bit can also be used), the subsequent development is also introduced as an example of this system.

- **Prerequisite Package Installation**

Install the necessary development dependencies on the host side first

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

```
PC$: sudo apt install -y git gnupg flex bison gperf build-essential zip curl
libc6-dev libncurses5-dev:i386 x11proto-core-dev libx11-dev:i386
libreadline6-dev:i386 libgl1-mesa-glx:i386 libgl1-mesa-dev g++-multilib
tofrodos python markdown libxml2-utils xsltproc zlib1g-dev:i386 gawk texinfo
gettext build-essential gcc libncurses5-dev bison flex zlib1g-dev gettext libssl-dev
autoconf libtool linux-libc-dev:i386 wget patch dos2unix tree u-boot-tools
```

Other non-required configuration packages

```
PC$: sudo dpkg-reconfigure dash # Select no
PC$: sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-gnu/lib
GL.so
PC$: sudo apt-get install zlib1g-dev # Install if libz.so is missing
PC$: sudo apt-get install uboot-mkimage # Install or install u-boot-tools when mk
imge is missing
```

2.2. Software Environment

2.2.1. Information Acquisition

Please refer to the "**MYD-YT113X SDK Release Notes**" for more information on the development board before building the environment.

http://d.myirtech.com/MYD-YT113

2.2.2. Install cross compilation tool chain

In the process of using the SDK to build this system image, you also need to install the cross tool chain, MYIR provides this SDK contains a variety of source code in addition to provide the necessary cross tool chain, can be used directly to compile applications and so on. Users can use the cross-compilation toolchain directly to build an independent development environment that can compile Bootloader, Kernel or compile their own applications separately, which will be described in detail in later chapters. Here the installation steps of the SDK are described first, as follows:

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

Copy the SDK archive to the user working directory under Ubuntu, such as "$HOME/T113X", which is defined according to your actual situation, and then unzip the file to get the SDK source code file as follows:

```
PC$ cd $HOME/T113X
PC$ tar -jxf YT113X-buildroot-T1-5.4.61-X.X.X.tar.bz2
```

Note: where X.X.X represents the current version number

- **View compiled chain file**

Go to the SDK directory and you can find it under the "build/toolchain" directory:

```
PC$ cd $HOME/T113X/T113Xauto-t113x-linux/build/toolchain
PC$ $HOME/T113X/T113Xauto-t113x-linux/build/toolchain$ ls
gcc-linaro-5.3.1-2016.05-x86_64_aarch64-linux-gnu.tar.xz
gcc-linaro-7.4.1-2019.02-x86_64_aarch64-linux-gnu.tar.xz
gcc-linaro-arm.tar.xz
gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi.tar.xz
gcc-linaro-aarch64.tar.xz
gcc-linaro.tar.bz2
```

The file names marked in red are the compilation chain of T113

- **Extracting compilation chain files**

Extract to the host's "/opt" directory, or you can choose your own directory if prompted

```
PC$ $HOME/T113X/T113Xauto-t113x-linux/build/toolchain
tar -xf gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi.tar.xz -C /opt
```

- **Installing and testing the compilation chain**

Set the environment variables and test that the installation is complete.

```
PC$  export PATH=$PATH:/opt/gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi/bin
PC$ arm-linux-gnueabi-gcc -v
Using built-in specs.
```

```
COLLECT_GCC=arm-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/home/sur/opt/gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi/bin/../libexec/gcc/arm-linux-gnueabi/5.3.1/lto-wrapper
Target: arm-linux-gnueabi
Configured with: /home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg/target/arm-linux-gnueabi/snapshots/gcc-linaro-5.3-2016.05/configure SHELL=/bin/bash --with-mpc=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg/target/arm-linux-gnueabi/_build/builds/destdir/x86_64-unknown-linux-gnu --with-mpfr=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg/target/arm-linux-gnueabi/_build/builds/destdir/x86_64-unknown-linux-gnu --with-gmp=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg/target/arm-linux-gnueabi/_build/builds/destdir/x86_64-unknown-linux-gnu --with-gnu-as --with-gnu-ld --disable-libstdcxx-pch --disable-libmudflap --with-cloog=no --with-ppl=no --with-isl=no --disable-nls --enable-c99 --with-tune=cortex-a9 --with-arch=armv7-a --with-fpu=vfpv3-d16 --with-float=softfp --with-mode=thumb --disable-multilib --enable-multiarch --with-build-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg/target/arm-linux-gnueabi/_build/sysroots/arm-linux-gnueabi --enable-lto --enable-linker-build-id --enable-long-long --enable-shared --with-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg/target/arm-linux-gnueabi/_build/builds/destdir/x86_64-unknown-linux-gnu/arm-linux-gnueabi/libc --enable-languages=c,c++,fortran,lto --enable-checking=release --disable-bootstrap --build=x86_64-unknown-linux-gnu --host=x86_64-unknown-linux-gnu --target=arm-linux-gnueabi --prefix=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg/target/arm-linux-gnueabi/_build/builds/destdir/x86_64-unknown-linux-gnu
Thread model: posix
gcc version 5.3.1 20160412 (Linaro GCC 5.3-2016.05)
```

You can see that the last line prints the same version as the one installed, which proves that the installation was successful.

3. Build development board image using SDK

3.1. Introduction

Linux SDK development kit, which integrates BSP, build system, linux application, test system, standalone IP, tools and documentation, both as a development, verification and release platform for BSP, IP and also as an embedded Linux system.

It is a unified use linux development platform. It integrates BSP, build system, standalone IP and test, and can be used both as a BSP development and IP verification platform and as a mass-produced embedded linux system.

The features of the Linux SDK include the following four parts:

- BSP development, including bootloader, uboot and kernel.
- Linux file system development, including mass-produced embedded linux systems.
- IP validation and publishing platform, and give the IP usage and system integration demo program, easy for third parties to use quickly.
- Testing, including board-level testing and system testing.

The 04_sources directory in the CD image provided by MYIR provides linux SDK files and data for the "MYD-YT113X" development board to help developers build different types of Linux system images that can be run on the "MYD-YT113X" development board, as shown below.

Table 3-1. MYD-YT113X Image file description

Image Files Name	Content Description
myir-image-yt113s3-emmc-core	Building an image without QT GUI with buildroot
myir-image-yt113s3-emmc-full	Build image with QT GUI, 7" LVDS display with buildroot
myir-image-yt113s3-nand	Building an image without QT GUI with buildroot
myir-image-yt113i-emmc-core	Building an image without QT GUI with buildroot
myir-image-yt113i-emmc-full	Build image with QT GUI, 7" LVDS display with buildroot

The following to build "myir-image-yt113s3-emmc-full" image as an example to introduce the specific development process for the subsequent customization of their own system image to lay the foundation for the user to select the corresponding operation according to their own development board model, the basic operation with the production of "myir-image-yt113s3-emmc-full" image The basic operation is the same as making "myir-image-yt113s3-emmc-full image", if there is a different operation this article will be specific point out, please watch this document carefully to avoid improper operation caused by the failure of the image production.

3.2. Get the source code

We provide two ways to get the source code, one is to get the zip package directly from the MYIR CD image 04_sources directory, and the other is to use repo to get the source code located on github real-time updates to build, please choose one of them according to the actual needs of the user to build.

3.2.1. Use the SDK source code provided by MYIR (recommended)

The compressed source package is located in MYIR Development Kit Profile "04_Sources/YT113X-buildroot-T1-5.4.61-X.X.X.tar.bz2" (X.X.X stands for the current version number). Copy the tarball to a user-specified directory, such as the "$HOME/T113X" directory, which will serve as the top-level directory for subsequent builds, and extract it as follows to bring up all the contents of the SDK:

```
PC$ cd $HOME/T113X
PC$ $HOME/T113X$ tar -jxf YT113X-buildroot-T1-5.4.61-X.X.X.tar.bz2
PC$ $HOME/T113X/ T113Xauto-t113x-linux$ tree -L 1
.
├── brandy
├── build
├── buildroot
├── build.sh
├── device
├── kernel
├── out
```

```
├── platform
├── test
└── tools

9 directories, 1 file
```

3.2.2. Get the source code via github

Currently the BSP source code and Buildroot source code of MYD-YT113X development board are hosted on github and will be kept updated for a long time, please see "*MYD-YT113X_SDK Release Notes*" for the code repository address. Users can use repo to get and synchronize the code on github. Here's how to do it:

```
PC$ mkdir $HOME/T113X
PC$ cd $HOME/T113X
PC$ export REPO_URL='https://mirrors.tuna.tsinghua.edu.cn/git/git-repo/'
PC$ repo init -u git@github.com:MYIR-ALLWINNER/myir-t1-manifest.git --no-clone-bundle --depth=1 -m myir-t113-5.4.61-1.0.0.xml -b develop-yt113x-manifest
PC$ repo sync
```

After successful code synchronization, you will also get an SDK folder in the "$HOME/T113X" directory, which contains the path to the source code or source code repository related to the MYD-YT113X development board, with the same directory structure as the one extracted from the zip package.

3.3. Understanding the linux SDK structure

```
├── brandy
├── build
├── buildroot
├── device
├── kernel
├── out
├── platform
├── test
└── tools
```

Mainly composed of brandy, buildroot, kernel, platform.

- brandy contains uboot2018
- buildroot is responsible for ARM toolchain, application packages, Linux root file system generation
- kernel is the linux kernel
- platform is a platform-related library and sdk application

3.3.1. buildroot Introduction

Buildroot is a set of Makefiles and patches that simplify and automate the building of complete, bootable Linux environments for embedded systems (including bootloader, Linux kernel, filesystem with various APPs). Embedded Linux platform.

Buildroot can automatically build the required cross-compilation toolchain, create the root filesystem, compile the Linux kernel image, and generate the bootloader for the target embedded system, or it can perform any independent combination of these steps. For example, the installed cross-compilation toolchain can be used alone, while Buildroot creates only the root filesystem.

Reference URL

Buildroot User Manuals https://buildroot.org/downloads/manual/manual.html

Buildroot Source code download site https://buildroot.org/downloads/

The directory structure is as follows

```
buildroot-201902/
├── arch
├── board
├── boot
├── build.sh
├── CHANGES
├── Config.in
├── Config.in.legacy
├── configs
├── COPYING
├── DEVELOPERS
├── dl
```

```
├── docs
├── fs
├── linux
├── Makefile
├── Makefile.legacy
├── package
├── README
├── scripts
├── support
├── system
├── toolchain
└── utils
```

The configs directory holds the predefined configuration files, as follows:

Table 3-2. MYD-YT113X buildroot configuration file description

Models	Configuration File	Description
MYD-YT113-S3	myd_yt113s3_emmc_core_br_defconfig	emmc core system buildroot configuration
	myd_yt113s3_emmc_full_br_defconfig	emmc full system buildroot configuration
	myd_yt113s3_nand_br_defconfig	nand system buildroot configuration
MYD-YT113-I	myd_yt113i_emmc_core_br_defconfig	emmc core system buildroot configuration
	myd_yt113i_emmc_full_br_defconfig	emmc full system buildroot configuration

The dl directory holds downloaded packages, the scripts compiled by buildroot, “mkcmd.sh”, “mkcommon.sh”, mkrule and “mksetup.sh”, etc.

The target directory holds the rules files for generating the root filesystem, which is important for code and tool integration. The most important for us is the package directory, which holds the rules for generating almost 3000 packages, where we can add our own packages or middleware

For more information about buildroot, please visit buildroot's official website http://buildroot.uclibc.org.

3.3.2. kernel

The linux kernel source code directory. The current kernel version is linux 5.4.61. The above directory structure is consistent with the standard linux kernel except for the modules directory, which is where we store external modules that are not integrated with the kernel's menuconfig.

3.3.3. brandy

There is a brandy2.0 version in the brandy directory. T113 currently uses brandy2.0, and its directory

structure is

```
brandy-2.0/
├── build.sh -> tools/build.sh
├── spl-pub
├── tools
└── u-boot-2018
```

3.3.4. platform

Platform Private Package Catalog

```
platform/
├── apps
├── base
├── config
├── core
├── external
├── framework
├── Makefile -> /home/XXX/T113X/auto-t113x-linux/build/Makefile
└── tools
```

In particular, "framework/auto" contains the SDK interface and examples for the T1 linux version.

```
platform/framework/auto/
├── rootfs
├── sdk_demo
└── sdk_lib
```

where rootfs will force an overwrite to the corresponding target in the out directory (target is the machine's root filesystem directory) each time "build.sh" is executed at the top level.

"framework/qt" contains the source code for QT5.12.5.

3.3.5. tools

```
tools/
├── build
├── codecheck
├── pack
└── tools_win
```

3.3.6. Allwinner test system

test is a test system called dragonboard. dragonboard provides fast board-level testing

3.3.7. device

This catalog contains two products, t113(MYD-YT113-S3) and t113_i(MYD-YT113-I). The t113 product catalog contains three types of image configurations, and the t113_i product catalog contains two types of image configurations. The core and full system configuration directory structure of MYD-YT113-S3 and MYD-YT113-I are basically the same, only the kernel and buildroot configurations are different, please refer to the "*MYD-YT113X_SDK Release Notes*" section 2.1 for the specific differences. Since there are many models, and the directory structure of full and core system of each model is not much different, so this article only explains the development method of different configurations."

/device/config/chips/t113 is the MYD-YT113-S3 model chip configuration directory, /device/config/chips/t113_i is the MYD-YT113-I model chip configuration directory, which contains multiple board configurations, and each

board configuration has different configuration files like board.dts, sys_config.fex and so on. config.fex and other configuration files.

- **MYD-YT113-S3 Model Configuration Explained**

The main contents are as follows:

```
device/config/chips/t113
├── bin
│   ├── boot0_nand_sun8iw20p1.bin          boot0 boot file for nand
│   ├── boot0_sdcard_sun8iw20p1.bin        boot0 boot file for emmc
│   ├── dsp0.bin
│   ├── fes1_sun8iw20p1.bin                Initialization file for burn tool
│   ├── optee_sun8iw20p1.bin               optee
│   ├── sboot_sun8iw20p1.bin               Safe start of bin
│   └── u-boot-sun8iw20p1.bin
├── boot-resource
│   ├── boot-resource
│   │   ├── bat
│   │   ├── bootlogo.bmp
│   │   ├── fastbootlogo.bmp
│   └── boot-resource.ini
├── configs
│   ├── default                        Not valid under normal conditions
│   ├── myir-image-yt113s3-emmc-full                              EMMC
│   │   ├── board.dts                  EMMC type dts configuration
│   │   ├── bsp
│   │   │   ├── BoardConfig.mk
│   │   │   ├── BoardConfig_nor.mk
│   │   │   ├── bootlogo.bmp
│   │   │   ├── env.cfg
│   │   │   ├── env_nor.cfg
│   │   │   ├── sys_partition.fex
│   │   │   └── sys_partition_nor.fex
│   │   ├── linux-5.4
│   │   │   ├── board.dts
```

```
|   |   |   └── config-5.4
|   |   ├── longan
|   |   |   ├── BoardConfig.mk                Kernel, buildroot, toolchain and other
                                                configurations
|   |   |   ├── BoardConfig_nor.mk
|   |   |   ├── bootlogo.bmp                  EMMC board type bootlog picture
|   |   |   ├── env_ab.cfg
|   |   |   ├── env.cfg                       EMMC board type environment variables
|   |   |   ├── env-recovery.cfg
|   |   |   ├── sys_partition.fex             EMMC board type default partition file
|   |   ├── sys_config.fex                    EMMC board type sys_config configuration
|   |   └── uboot-board.dts                   EMMC board type uboot using the dts file
|   ├── myir-image-yt113s3-nand               SPI NAND Board Level Catalog
|   |   ├── BoardConfig.mk
|   |   ├── board.dts -> linux-5.4/board.dts  SPI NAND board-level dts configura
                                                tion
|   |   ├── bsp
|   |   |   ├── bootlogo.bmp
|   |   |   ├── env.cfg
|   |   |   └── sys_partition.fex
|   |   ├── env.cfg
|   |   ├── linux-5.4
|   |   |   ├── board.dts
|   |   |   └── config-5.4
|   |   ├── longan
|   |   |   ├── BoardConfig.mk                Kernel, buildroot, toolchain and other
                                                configurations
|   |   |   ├── bootlogo.bmp                  SPI NAND board type bootlog picture
|   |   |   ├── env.cfg                       SPI NAND board type environment variables
|   |   |   └── sys_partition.fex             SPI NAND board type default partition file
|   |   ├── sys_config.fex                    SPI NAND board type sys_config configuration
|   |   ├── sys_partition.fex
|   |   └── uboot-board.dts                   SPI NAND board type dts file used by uboot
```

- **MYD-YT113-I Model Configuration Explained**

```
device/config/chips/t113_i
├── bin
│   ├── boot0_nand_sun8iw20p1.bin              boot0 boot file for nand
│   ├── boot0_sdcard_sun8iw20p1.bin            boot0 boot file for emmc
│   ├── boot0_spinor_sun8iw20p1.bin
│   ├── dsp0.bin
│   ├── fes1_sun8iw20p1.bin                    Initialization file for burn tool
│   ├── optee_sun8iw20p1.bin                   optee
│   ├── sboot_sun8iw20p1.bin                   Safe start of bin
│   ├── u-boot-spinor-sun8iw20p1.bin
│   └── u-boot-sun8iw20p1.bin
├── boot-resource
│   ├── boot-resource
│   │   ├── bat
│   │   ├── bootlogo.bmp
│   │   ├── fastbootlogo.bmp
│   │   ├── font24.sft
│   │   ├── font32.sft
│   │   └── wavefile
│   └── boot-resource.ini
├── configs
│   ├── default                                Not valid under normal conditions
│   └── myir-image-yt113i-full                 EMMC
│       ├── board.dts                          EMMC type dts configuration
│       ├── bsp
│       ├── linux-5.4
│       ├── longan
│   │   │   ├── BoardConfig.mk                 Kernel, buildroot, toolchain and other
                                                configurations
│   │   │   ├── BoardConfig_nor.mk
│   │   │   ├── bootlogo.bmp                   EMMC board type bootlog picture
│   │   │   ├── env_ab.cfg
```

```
|   |   |   ├── env.cfg                EMMC board type environment variables
|   |   |   ├── env_nor.cfg
|   |   |   ├── sys_partition_ab.fex
|   |   |   ├── sys_partition.fex      EMMC board type default partition file
|   |   ├── sys_config.fex             EMMC board type sys_config configuration
|   |   └── uboot-board.dts            EMMC board type uboot using the dts file
```

3.4. Linux SDK configuration and build

This section describes the detailed steps of full compilation and partial compilation. After compilation, the final img is generated by packaging.

This section explains how to generate "myir-image-yt113s3-emmc-full.img" and "myir-image-yt113i-full.img" images by performing the following steps:

First go to the source top-level directory: "T113Xauto-t113x-linux", then execute the following command.

```
PC$: cd $HOME/T113X/T113Xauto-t113x-linux
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh config
After the implementation of the configuration of the corresponding development board model, the details of the selection explained later
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh
====If you don't need the qt function, you can skip these two commands=====
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh qt
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh
============================================================
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh pack
```

The commands required for the entire compilation process are described above, and each command will be explained step by step below

1). Step 1 "./build.sh config"

Execute build.sh config and select the configuration in the subsequent dialog.

Enter the string corresponding to the number.

- **MYD-YT113-S3 Model Configuration Selection**

```
PC$ ./build.sh config
```

```
Welcome to mkscript setup progress
All available platform:
   0. linux
Choice [linux]: 0
All available linux_dev:
   0. bsp
   1. dragonboard
   2. longan
   3. tinyos
Choice [longan]: 2
All available kern_ver:
   0. linux-5.4
Choice [linux-5.4]: 0
All available ic:
   0. t113
   1. t113_i
Choice [t113]: 0
All available board:
   0. myir-image-yt113s3-emmc-core
   1. myir-image-yt113s3-emmc-full
   2. myir-image-yt113s3-nand
Choice [myir-image-yt113s3-emmc-full]: 1

"S3" model "nand" development board only "core" system can only choose "2".
"EMMC" development boards can be selected "core", "full" system
All available flash:
   0. default
   1. nor
Choice [default]: 0
All available gnueabi:
   0. gnueabi
   1. gnueabihf
Choice [gnueabi]: 0
```

- **MYD-YT113-I Model Configuration Selection**

```
PC$ ./build.sh config
Welcome to mkscript setup progress
All available platform:
  0. linux
Choice [linux]: 0
All available linux_dev:
  0. bsp
  1. dragonboard
  2. longan
  3. tinyos
Choice [longan]: 2
All available kern_ver:
  0. linux-5.4
Choice [linux-5.4]: 0
All available ic:
  0. t113
  1. t113_i
Choice [t113_i]: 1
All available board:
  0. myir-image-yt113i-core
  1. myir-image-yt113i-full
Choice [myir-image-yt113i-full]: 1
All available flash:
  0. default
  1. nor
Choice [default]: 0
All available gnueabi:
  0. gnueabi
  1. gnueabihf
Choice [gnueabi]: 0
```

If after executing ". /build.sh config", the following error message appears

```
File "<string>", line 1
```

```
  import os.path; print os.path.relpath('/home/XXX/T113X/auto-t113x-linux/kern
el/linux-5.4/arch/arm/configs/sun8iw20p1smp_t113_auto_defconfig', '/home/XXX
/T113X/auto-t113x-linux/kernel/linux-5.4/arch/arm/configs')
                    ^
SyntaxError: invalid syntax
ERROR: Can't find kernel defconfig!
```

This is caused by the wrong version of python, python2 is required. Please check if python2 is already installed in your development environment, if not, please check section 2.1 or run the following command directly:

```
PC$ sudo apt-get install python
```

Check if the current version is python2

```
PC$ python --version
Python 2.7.18
```

If python2 is installed but the version is still other versions, then the host environment has more than one version of python installed, at this time you need to switch the python version by yourself, execute the following command to switch the version:

Switch according to your actual hosting environment

```
sudo update-alternatives --install /usr/bin/python python /usr/bin/python2.7 2
sudo update-alternatives --install /usr/bin/python python /usr/bin/python3.6 1
sudo update-alternatives --config python


------------------------------------------------------------
  0            /usr/bin/python2.7   2
* 1            /usr/bin/python2.7   2
  2            /usr/bin/python3.6   1
```

2). Step 2 "./build.sh"

After the configuration is selected, execute the following command to start compiling the system, this process will take a long time.

```
PC$: cd $HOME/T113X/T113Xauto-t113x-linux
```

```
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh
ACTION List: mklichee;========
Execute command: mklichee
INFO: ----------------------------------------
INFO: build lichee ...
INFO: chip: sun8iw20p1
INFO: platform: linux
INFO: kernel: linux-5.4
INFO: board: emmc_full
INFO: output: /home/koi/T113X/T113Xauto-t113x-linux/out/t113/myir-image-yt1
13s3-emmc-full/longan
INFO: ----------------------------------------
INFO: build buildroot ...
.......
INFO: pack rootfs ok ...
INFO: ----------------------------------------
INFO: build lichee OK.
INFO: ----------------------------------------
```

The following error may be encountered when performing this step

```
gdbusconnection.c: In function 'check_initialized':
gdbusconnection.c:567:8: warning: unused variable 'flags' [-Wunused-variable]
  567 |   gint flags = g_atomic_int_get (&connection->atomic_flags);
      |        ^~~~~
gdbusmessage.c: In function 'parse_value_from_blob':
gdbusmessage.c:1712:29: warning: variable 'item' set but not used [-Wunused-but-set-variable]
 1712 |                   GVariant *item;
      |                             ^~~~
gdbusmessage.c: In function 'append_value_to_blob':
gdbusmessage.c:2326:24: warning: unused variable 'end' [-Wunused-variable]
 2326 |           const gchar *end;
      |                        ^~~
gdbusauth.c: In function '_g_dbus_auth_run_server':
gdbusauth.c:1302:11: error: '%s' directive argument is null [-Werror=format-overflow=]
 1302 |           debug_print ("SERVER: WaitingForBegin, read '%s'", line);
      |           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  CC       libgio_2_0_la-gdbusinterface.lo
cc1: some warnings being treated as errors
Makefile:3633: recipe for target 'libgio_2_0_la-gdbusauth.lo' failed
make[5]: *** [libgio_2_0_la-gdbusauth.lo] Error 1
make[5]: *** Waiting for unfinished jobs....
gdbusmessage.c: In function 'g_dbus_message_to_blob':
gdbusmessage.c:2702:30: error: '%s' directive argument is null [-Werror=format-overflow=]
 2702 |       tupled_signature_str = g_strdup_printf ("(%s)", signature_str);
      |                              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
gdbusintrospection.c: In function 'g_dbus_interface_info_generate_xml':
gdbusintrospection.c:751:3: warning: 'access_string' may be used uninitialized in this function [-Wmaybe-uninitialized]
  751 |   g_string_append_printf (string_builder, "%*s<property type=\"%s\" name=\"%s\" access=\"%s\"",
```

Figure 3-1. Compile failure 1

Modify step 1: (Note that the full name of the path may not be the same, find "gdbusauth.c" according to your actual path).

```
PC$: $HOME/T113X/T113Xauto-t113x-linux$ vim ./out/t113/myir-image-yt113s3-emmc-full/longan/buildroot/build/host-libglib2-2.56.3/gio/gdbusauth.c
```

Add this judgment code in the following position:

```
line = _my_g_input_stream_read_line_safe (g_io_stream_get_input_stream (auth->priv->stream),
&line_length,
cancellable,
error);
 if (line != NULL)
        debug_print ("SERVER: WaitingForBegin, read '%s'", line);
if (line == NULL)
```

Modify step 2: (Note that the path may be different in full, find "gdbusmessage.c" according to your actual path).

```
PC$: $HOME/T113X/T113Xauto-t113x-linux$ vim ./out/t113/myir-image-yt113s3-emmc-full/longan/buildroot/build/host-libglib2-2.56.3/gio/gdbusmessage.c
```

Add this judgment code in the following position:

```
signature_str = g_variant_get_string (signature, NULL);
if (message->body != NULL)
{
        gchar *tupled_signature_str;
        if (signature != NULL)
            tupled_signature_str = g_strdup_printf ("(%s)", signature_str);
        if (signature == NULL)
```

```
gawk -f ./mkerrnos.awk ./errnos.in >code-to-errno.h
gawk -f ./mkerrcodes1.awk ./errnos.in >_mkerrcodes.h
gawk -f ./mkstrtable.awk -v textidx=2 -v nogettext=1 \
       ./err-sources.h.in >err-sources-sym.h
gawk -f ./mkstrtable.awk -v textidx=2 -v nogettext=1 \
       ./err-codes.h.in >err-codes-sym.h
gawk -f ./mkstrtable.awk -v textidx=2 -v nogettext=1 \
      -v prefix=GPG_ERR_ -v namespace=errnos_ \
       ./errnos.in >errnos-sym.h
gawk: ./mkerrnos.awk:86: warning: regexp escape sequence `\#' is not a known regexp operator
gawk: ./mkerrcodes1.awk:84: warning: regexp escape sequence `\#' is not a known regexp operator
gawk: ./mkstrtable.awk:113: warning: regexp escape sequence `\#' is not a known regexp operator
/home/cat/T113/auto-t113-linux/out/t113/evb1_auto/longan/buildroot/host/bin/arm-linux-gnueabi-cpp -D_LARGEFILE_SOURCE -D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64  -P _mk
errcodes.h | grep GPG_ERR_ | \
             gawk -f ./mkerrcodes.awk >mkerrcodes.h
gawk: ./mkstrtable.awk:113: warning: regexp escape sequence `\#' is not a known regexp operator
/usr/bin/gcc -g -O0 -I. -I. -o mkheader ./mkheader.c
gawk: fatal: cannot use gawk builtin `namespace' as variable name
make[4]: *** [Makefile:1615: errnos-sym.h] Error 2
make[4]: *** Waiting for unfinished jobs....
gawk: ./mkerrcodes.awk:88: warning: regexp escape sequence `\#' is not a known regexp operator
rm _mkerrcodes.h
make[3]: *** [Makefile:508: all-recursive] Error 1
make[2]: *** [Makefile:440: all] Error 2
make[1]: *** [package/pkg-generic.mk:241: /home/cat/T113/auto-t113-linux/out/t113/evb1_auto/longan/buildroot/build/libgpg-error-1.33/.stamp_built] Error 2
make: *** [Makefile:96: _all] Error 2
make: Leaving directory '/home/cat/T113/auto-t113-linux/buildroot/buildroot-201902'
ERROR: build buildroot Failed
```

Figure 3-2. Compilation failure 2

```
PC$ cd $HOME/T113X/T113Xauto-t113x-linux/out/t113/myir-image-yt113s3-emmc-full/longan/buildroot/build/libgpg-error-1.33/src
```

Change "namespace" to "pkg_namespace" in "Makefile", "Makefile.am", "Makefile.in", and "mkstrtable.awk" in this directory, and then re-execute the compile command.

3). Qt Compilation

If you don't need the qt function, you can skip this step and execute the following command to compile qt:

```
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh qt
ACTION List: mkqt;========
Execute command: mkqt
INFO: build Qt ...
INFO: build arm-linux-gnueabi version's Qt
$HOME/T113X/T113Xauto-t113x-linux/platform/framework/qt/qt-everywhere-src-5.12.5
.......

INFO: build buildroot OK.
INFO: build Qt and buildroot Ok.
```

At this point, qt is compiled successfully, and then you must execute ". /build.sh" command, this command will just compile qt generated by the relevant files,

libraries, demos compiled into the system, otherwise the system will not support qt.

4). Step 3 "./build.sh pack"

The last step is to package the image to generate the system by executing the following command:

```
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh pack
ACTION List: mkpack ;========
Execute command: mkpack
INFO: packing firmware ...
INFO: Use BIN_PATH: $HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/bin
......
Dragon execute image.cfg SUCCESS !
----------image is at----------

size:456M  $HOME/T113X/T113Xauto-t113x-linux/out/myir-image-yt113s3-emmc-full.img
```

3.5. Onboard u-boot compilation

U-boot is a very feature-rich open source boot program, including kernel boot, download updates and many other aspects, in the embedded field is very widely used, check the official website for more information http://www.denx.de/wiki/U-Boot/WebHome

The T1 platform also uses boot chains as the bootloader, and different boot chains modes will correspond to different boot phases.

3.5.1. Compile u-boot separately

1). Obtain u-boot source code

Copy the development package "04 Source/YT113X-buildroot T1-5.4.61-X.X.X.X.tar. bz" (X.X.X represents the current version) to the specified custom T113 directory (such as "$HOME/T113X"), unzip and enter the source directory to view the corresponding file information, such as copying to the T113 directory:

```
PC$ cd $HOME/T113X
PC$ $HOME/T113X$ tar -jxf YT113X-buildroot-T1-5.4.61-X.X.X.tar.bz
```

- Source Code Directory：u-boot-2018
- SPL source code directory： spl-pub
- Compile Scripts：build.sh

```
PC$ $HOME/T113X$ cd T113Xauto-t113x-linux/brandy/brandy-2.0$
PC$: $HOME/T113X/T113Xauto-t113x-linux/brandy/brandy-2.0$ tree -L 1
├── build.sh -> tools/build.sh
├── spl-pub
├── tools
└── u-boot-2018
```

2). Configuration and Compilation

- **Go to the source code directory**

```
PC$: $HOME/T113X/T113Xauto-t113x-linux/$ cd brandy/brandy-2.0
```

- **Load the toolchain in the SDK**

```
PC$: $HOME/T113X/T113Xauto-t113x-linux/brandy/brandy-2.0$ ./build.sh -t
Prepare toolchain ...
```

- **Load the defconfig configuration file**

  - MYD-YT113-S3

```
PC$: $HOME/T113X/T113Xauto-t113x-linux/brandy/brandy-2.0$ cd u-boot-2018
eMMC：
PC$: $HOME/T113X/T113Xauto-t113x-linux/brandy/brandy-2.0/u-boot-2018$ ma
ke sun8iw20p1_auto_defconfig

Nand：
PC$: $HOME/T113X/T113Xauto-t113x-linux/brandy/brandy-2.0/u-boot-2018$ ma
ke sun8iw20p1_auto_nand_defconfig
```

  - MYD-YT113-I

```
PC$: $HOME/T113X/auto-t113x-linux/brandy/brandy-2.0/u-boot-2018$
make sun8iw20p1_auto_t113_i_defconfig
```

Note: Be sure to load the corresponding defconfig file according to your board model.

- **Modify uboot configuration**

  ➢ make menuconfig GUI changes

After loading the "defconfig" file, execute "make menuconfig" to open the uboot GUI configuration.

```
PC$: $HOME/T113X/auto-t113x-linux/brandy/brandy-2.0/u-boot-2018$
make menuconfig
```

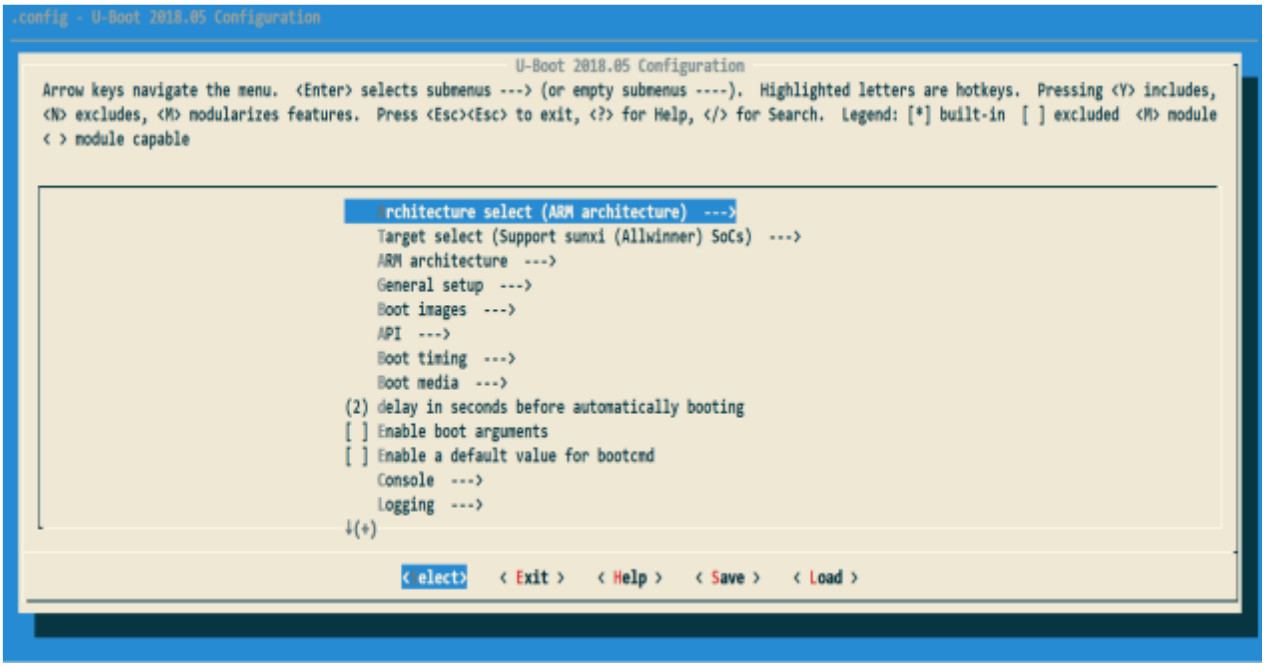

Figure 3-3. Uboot Graphical Configuration Interface

At this point, users only need to check or uncheck a configuration option according to their actual situation, you can make corresponding changes to the uboot configuration.

However, the configuration modified by this operation is only saved in an intermediate temporary configuration file, so if the next time you load another model's configuration file, and then come back to load that configuration file, then the changes made earlier will be restored.

➢ Modify the defconfig configuration file directly (recommended)

Users can directly modify the corresponding "defconfig" file to change the uboot configuration, take the MYD-YT113-S3 EMMC board as an example, the corresponding "defconfig" file needs to be modified as "sun8iw20p1_auto_defconfig", users only need to open the file and modify it

according to their actual situation. The corresponding "defconfig" file is "sun8iw20p1_auto_defconfig", users only need to open the file and modify it according to their actual situation. This method can save the changes made to the configuration of uboot, and will not disappear because of loading the configuration for the second time.

The defconfig file for all MYD-YT113X models is in the following path

```
PC$: $HOME/T113X/auto-t113x-linux/brandy/brandy-2.0/u-boot-2018/configs$
```

- **Compile and update uboot**

The configuration file corresponding to "defconfig" has been loaded and modified above, so let's start compiling uboot.

```
PC$: $HOME/T113X/auto-t113x-linux/brandy/brandy-2.0/u-boot-2018$ make
  CHK     include/config/uboot.release
  CHK     include/generated/version_autogenerated.h
  CHK     include/generated/timestamp_autogenerated.h
  CHK     include/generated/generic-asm-offsets.h
  CHK     include/generated/asm-offsets.h
'u-boot.bin' -> 'u-boot-sun8iw20p1.bin'
'u-boot-sun8iw20p1.bin' -> '/media/home/nico/RC/T113-I/SDK/device/config/chips/t113/bin/u-boot-sun8iw20p1.bin'
'u-boot-sun8iw20p1.bin' -> '/media/home/nico/RC/T113-I/SDK/out/t113/myd_yt113_s3_emmc_full/longan/u-boot-sun8iw20p1.bin'
  CHK     include/config.h
  CFG     u-boot.cfg
  CFGCHK  u-boot.cfg
```

After the compilation is complete, you need to go back to the top-level directory of the SDK source code and execute the following command to package the changed uboot compiled files into the out directory

```
PC$: cd $HOME/T113X/auto-t113x-linux$
PC$: $HOME/T113X/auto-t113x-linux$ ./build.sh pack
```

At this time in the "$HOME/T113X/auto-t113x-linux/out/pack_out" directory to find the "boot_package.fex" file, this file into the USB flash disk Insert the USB

flash drive into the development board, the USB flash drive will be automatically mounted in the following path, and then execute the following commands to update the uboot individually

```
root@myd-yt113-s3:~# cd /mnt/usb/sda1/
root@myd-yt113-s3:/mnt/usb/sda1# ota-burnuboot boot_package.fex
Burn Uboot Success
```

At this time uboot update success, at present the separate update uboot method only applies to EMMC board type development board, nand development board does not support separate update for the time being. Therefore, nand boards do not support separate update for the time being, so after nand boards execute the ". /build.sh pack" and then burn it into the board according to the method in chapter 4.1, then uboot can be updated.

3.5.2. Compile u-boot under linux SDK (recommended)

Once the user has changed the U-boot code according to the iterative development process in 3.5.1, the entire image can also be built using the SDK.

It is not possible to modify the uboot configuration through the GUI, you can only modify the defconfig file directly and then compile it using this method.

Compiling uboot source code:

```
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh brandy
```

Return to the top-level directory of the SDK source code and package the image:

```
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh pack
```

After compiling the uboot source code, follow the steps in section 3.5.1 to update uboot separately, or follow the method in section 4.1 to burn into the development board and update uboot.

3.6. Onboard Kernel Compilation and Update

Linux kernel is a very large open source kernel that is used in various distributions of operating systems. Linux kernel is widely used in embedded systems due to its portability, multiple network protocol support, independent module mechanism, MMU and many other rich features.

At the same time, T1 also supports Linux kernel, will get long-term stable update, MYD-YT113X using T1 kernel port, the latest support Linux kernel 5.4.61 version.

1). Get the kernel source code

Copy the development package "04_Source/YT113X-buildroot-T1-5.4.61-X.X.X.tar.bz" (X.X.X represents the current version) to the specified custom work directory (such as "$HOME/T113X"), unzip it into the source directory and check the corresponding file information, e.g. copy it to the work directory:

```
PC$ cd $HOME/T113X
PC$ $HOME/T113X$ tar -jxf YT113X-buildroot-T1-5.4.61-X.X.X.tar.bz
PC$ $HOME/T113X$ cd T113Xauto-t113x-linux/kernel
PC$ $HOME/T113X/T113Xauto-t113x-linux/kernel$ tree -L 1
.
└── linux-5.4
```

The catalog contains:

- Source Code：linux-5.4

2). Modify kernel configuration

MYIR already integrates most of the functionality into the kernel and generally does not need to be configured. To add special features, the peripheral drivers should be configured as follows.

- **Loading compilation chain**

Configure the compilation chain according to section 2.2, then configure the compilation chain environment.

```
PC$  export PATH=$PATH:/opt/gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi/bin
```

- **Go to the kernel directory**

```
PC$ cd $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4$
```

- **Load defconfig configuration**
  - MYD-YT113-S3

```
emmc:
full:
```

```
PC$ $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4$ make ARCH=arm myd_yt113s3_emmc_full_defconfig

core:
PC$ $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4$ make ARCH=arm myd_yt113s3_emmc_core_defconfig

nand:
PC$ $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4$ make ARCH=arm myd_yt113s3_nand_defconfig
```

- MYD-YT113-I

```
full:
PC$ $HOME/T113X/auto-t113x-linux/kernel/linux-5.4$ make myd_yt113i_emmc_full_defconfig

core:
PC$ $HOME/T113X/auto-t113x-linux/kernel/linux-5.4$ make myd_yt113i_emmc_core_defconfig
```

Note: After loading the corresponding "defconfig" file, a temporary ".config" file will be created in the current directory, make a copy of this file to directory A and name it "1.con" (you can choose the directory according to your own situation), which will be needed in the following steps.

- **Open the kernel configuration screen**

```
PC$ $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4$ make ARCH=arm menuconfig
```

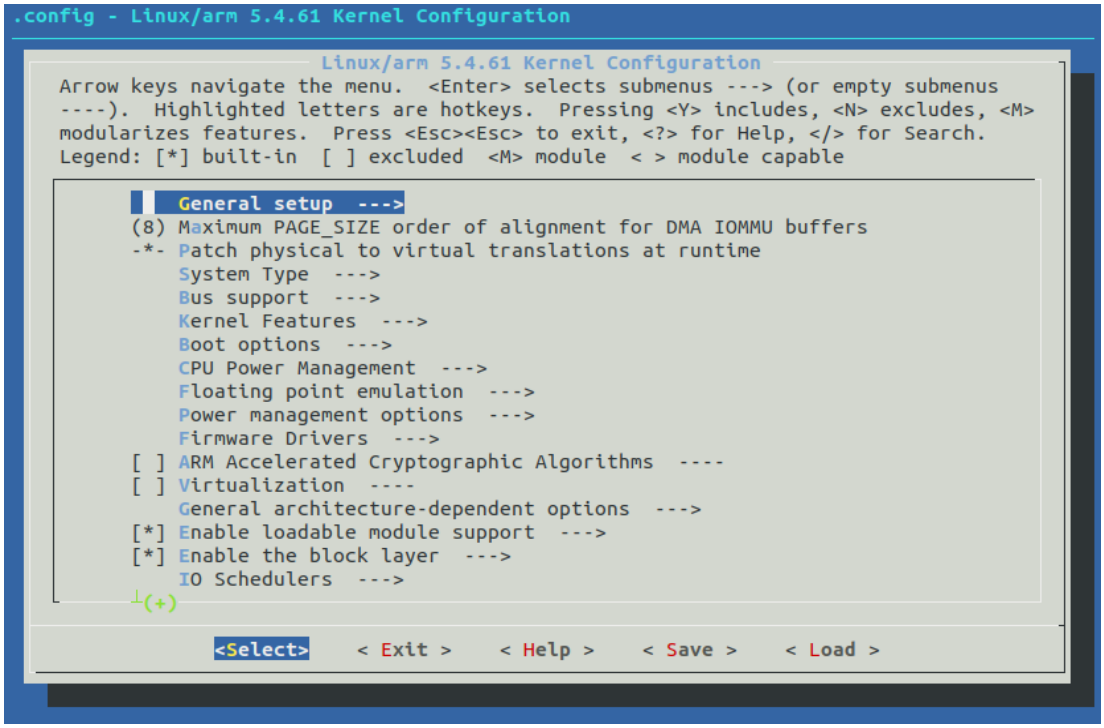

Figure 3-4. Kernel configuration interface

After opening the kernel configuration interface, you can add or some kernel configuration, this time add System V IPC for example, the configuration can be found under "General setup--->", press "y" to select the configuration, press "n" to cancel the configuration, press "m" to compile into a module.

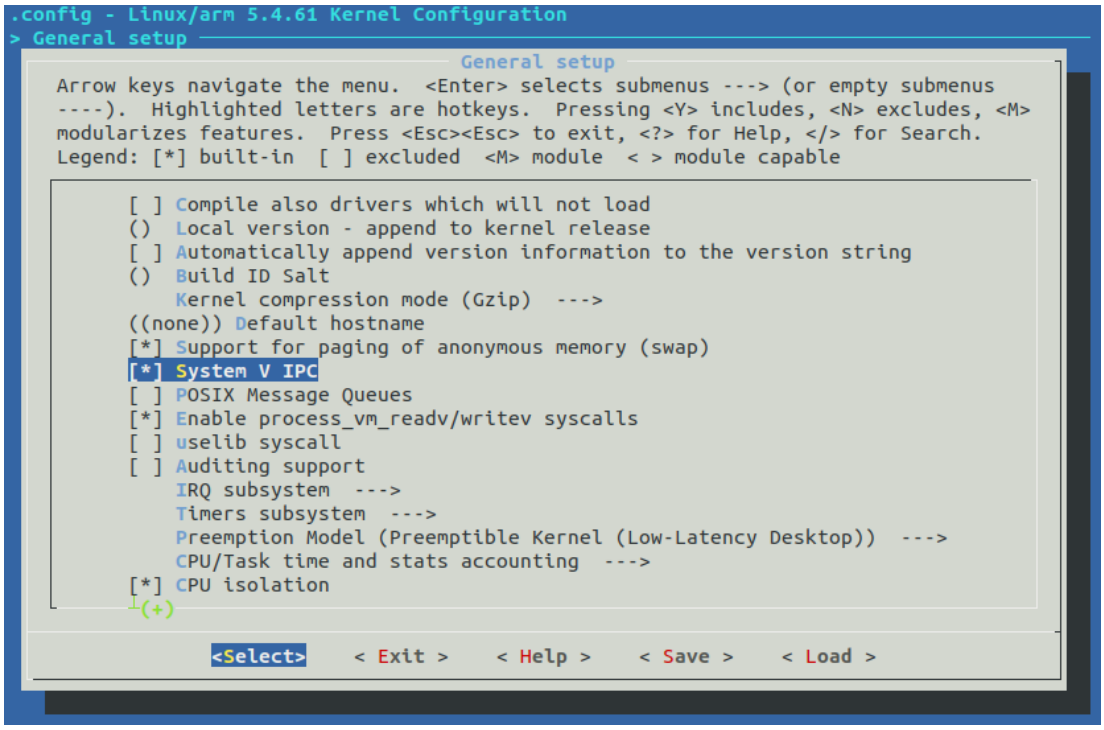

Figure 3-5. Checking the kernel configuration

Note: After adding the configuration, you need to make corresponding changes in the corresponding "defconfig" file, otherwise the configuration will not take effect. Exit the graphical interface after adding the configuration. At this time, the ".config" file in the current directory has saved the configuration just now, copy the new ".config" file to the directory A and name it "2. con", compare it with "1.con" in the previous section, and modify the corresponding "defconfig" file.

All defconfig files for the MYD-YT113X kernel are in the following directory

```
PC$: $HOME/T113X/auto-t113x-linux/kernel/linux-5.4/arch/arm/configs$
```

Go back to the top-level directory of the SDK source code and execute the following commands to select the configuration, compile, and package images

```
PC$: cd $HOME/T113X/T113Xauto-t113x-linux$
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh config
  (Same configuration as step 1 of section 3.4)
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh kernel
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh pack
```

At this point in the "$HOME/T113X/auto-t113x-linux/out/pack_out" directory to find the "boot.fex" file, this file into the USB flash drive, insert the USB flash drive into the development board, the USB flash drive will be automatically mounted in the following path, then execute the following command to update the kernel individually. Insert the USB flash drive into the development board, the USB flash drive will be automatically mounted in the following path, then execute the following command to update the kernel separately.

```
root@myd-yt113-s3:~# dd if=/mnt/usb/sda1/boot.fex of=/dev/mmcblk0p4
41200+0 records in
41200+0 records out
21094400 bytes (21 MB, 20 MiB) copied, 3.51428 s, 6.0 MB/s
```

At this time, the kernel update is successful, at present, the separate update kernel method is only applicable to EMMC board type development board, nand development board does not support the separate update for the time being. The nand board does not support separate update at the moment, so the nand board should follow the 4.4.2 procedure after executing the ". /build.sh pack" and then burn the kernel into the board according to the method in section 4.1, then the kernel can be updated.

3). Update the device tree

The board-level dts and the dts used by uboot are in the "$HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/ myir-image-yt113s3-emmc-full" directory (take the emmc full image configuration as an example).

```
PC$: cd $HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/myir-image-yt113s3-emmc-full$ tree -L 1
.
├── board.dts
├── bsp
├── linux-5.4
├── longan
├── sys_config.fex
└── uboot-board.dts
```

After modifying the device tree, go back to the top-level directory of the SDK source code and execute the following commands to compile and package the image.

```
PC$: cd $HOME/T113X/T113Xauto-t113x-linux$
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh kernel
PC$: $HOME/T113X/T113Xauto-t113x-linux$ ./build.sh pack
```

Currently, it is not possible to update the device tree alone, so after compiling the device tree file, packing the image, and burning it to the development board according to the method in chapter 4.1, you can update the device tree file.

4. Burning the system image

MYIR company designed MYC-YT113X series core board and development board is based on Allwinner's T1 series microprocessor, which has various boot methods, so different tools and methods are needed to update the system. Users can choose different ways to update according to their needs. The update methods are mainly as follows:

- Create SD card bootloader: suitable for R&D debugging, fast boot scenarios, etc.
- Make SD card burner: suitable for mass production burning eMMC

4.1. Create SD card image

The following steps are made under Windows system.

- **Preparation**
  - SD card (not less than 8G)
  - MYD-YT113X Development Board
  - Create image tool PhoenixCard (path: \03_Tools\myir tools)

Table 4-1. list of mirror packages

Image Name	Package Name	Applicable core boards
myir-image-yt113s3-emmc-core	myir-image-yt113s3-emmc-core.img	MYC-YT113S3-4E128D
myir-image-yt113s3-emmc-full	myir-image-yt113s3-emmc-full.img	MYC-YT113S3-4E128D
myir-image-yt113s3-nand	myir-image-yt113s3-nand.img	MYC-YT113S3-256N128D
myir-image-yt113i-core	myir-image-yt113i-core.img	MYC-YT113i-4E512D MYC-YT113i-8E1D
myir-image-yt113i-full	myir-image-yt113i-full.img	MYC-YT113i-4E512D MYC-YT113i-8E1D

4.1.1. Making an SD card bootloader (using the myir-image-yt113s3-emmc-full system as an example)

1). Modify the configuration file to make the SD image

Currently the image generated by the SDK does not support SD card boot, you need to modify the following configuration:

```
PC$ cd $HOME/T113X/auto-t113x-linux/device/config/chips/t113/configs/myir-image-yt113s3-emmc-full
PC$ $HOME/T113X/auto-t113x-linux/device/config/chips/t113/configs/ myir-image-yt113s3-emmc-full$ vim sys_config.fex
……
;------------------------------------------------------------------------------
;storage_type    = boot medium, 0-nand, 1-sd, 2-emmc, 3-nor, 4-emmc3, 5-spinand -1(defualt)auto scan
;------------------------------------------------------------------------------
[target]
storage_type     = 1
burn_key         = 0
;------------------------------------------------------------------------------

PC$ $HOME/T113X/auto-t113x-linux/device/config/chips/t113/configs/ myir-image-yt113s3-emmc-full /longan$ vim


#kernel command arguments
earlycon=uart8250,mmio32,0x02501400
initcall_debug=0
console=ttyAS5,115200
nand_root=ubi0_5
mmc_root=/dev/mmcblk1p5
mtd_name=sys
rootfstype=ubifs,rw
init=/init
loglevel=7
……..
```

Note: nand image does not support SD card boot

2). SD boot image burning steps

Copy PhoenixCard_.zip from the user profile tools directory to any directory in windows, double click PhoenixCard.exe file in PhoenixCard directory. Insert the 16GB SD card into the windows USB port via SD card reader, as shown below, select "Firmware" path; select "Boot Card" and click "Burn Card" button. "button to automatically complete the production.

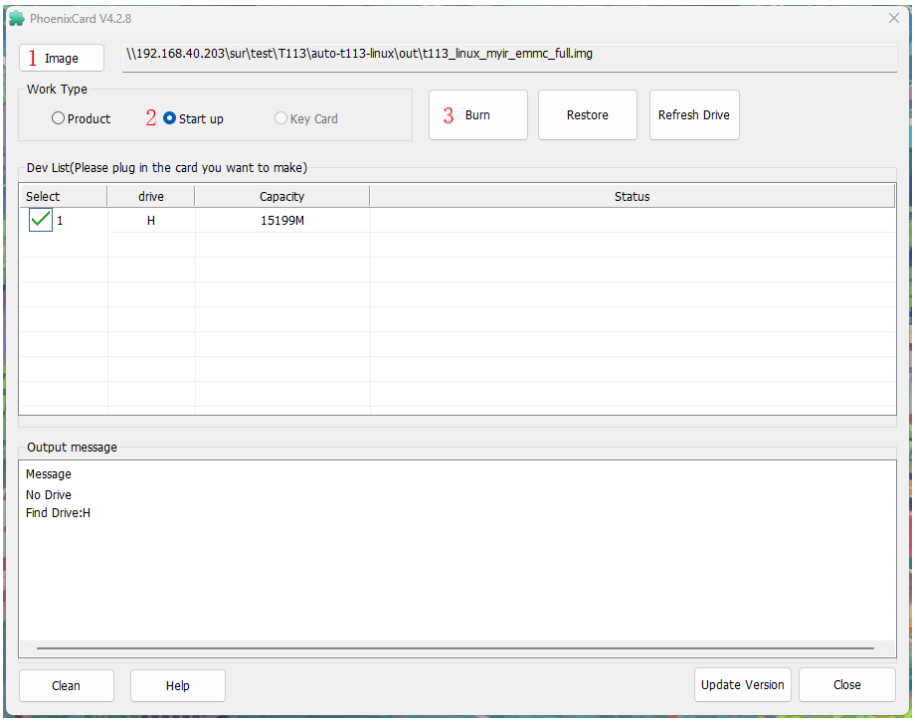

Figure 4-1. Brush writing steps

The figure below (Figure 4-2) shows that the card is being burned and the process is expected to take 3-5 minutes to complete (the time depends on the size of the mirror)

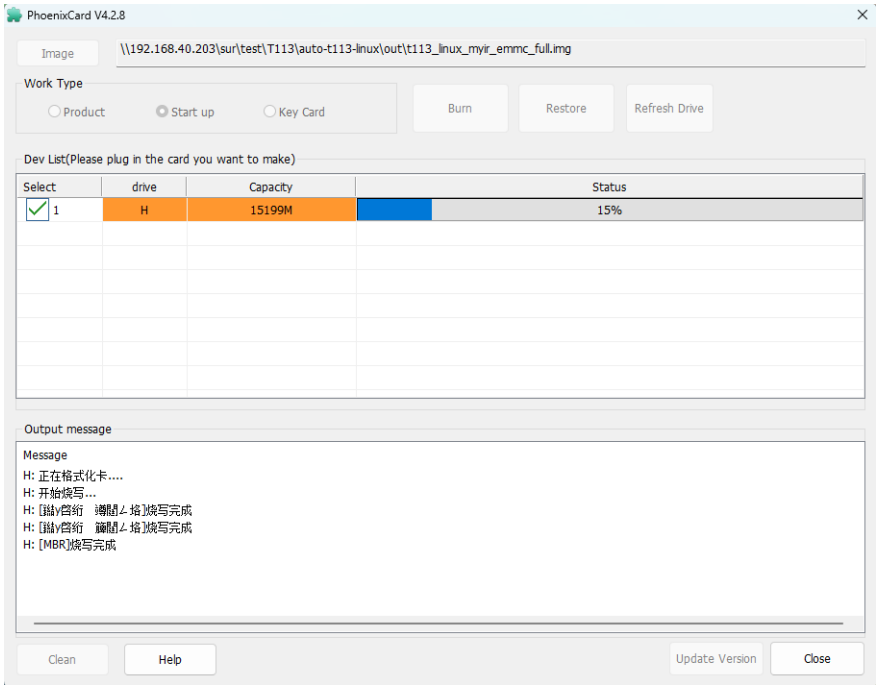

Figure 4-2. Swipe process

The following figure (Figure 4-3) shows that the burning of the card is complete, while note that the output message indicates that the burning is complete, at this time the SD card booter is successfully created, insert the SD card into the SD card slot (J5) of the emmc or nand board, and then power on the development board can be started.

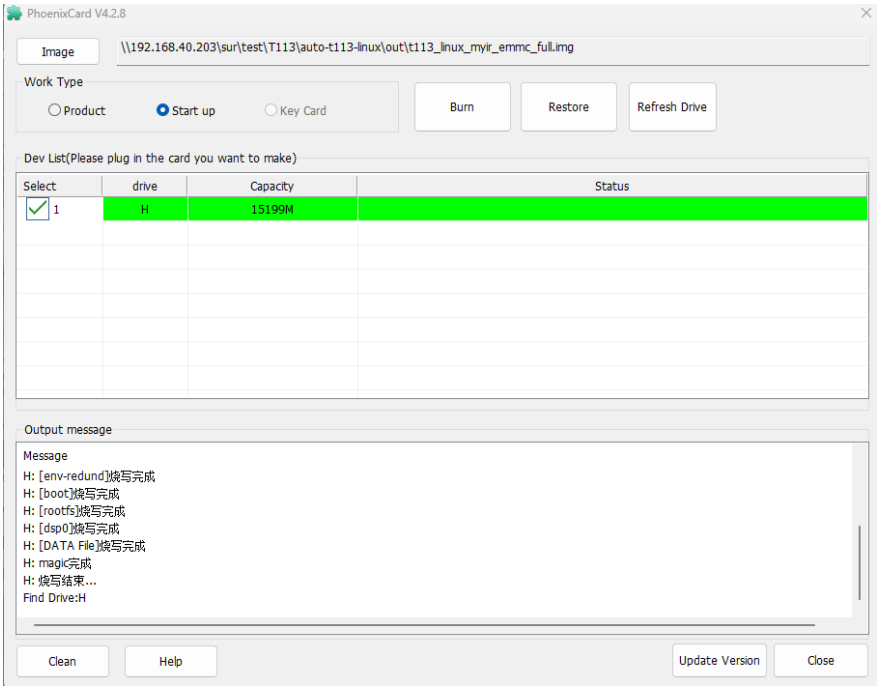

Figure 4-3. Successful swipe

MYIR Electronics Limited
Web: www.myirtech.com Mail: sales@myirtech.com Tel: 0755-22316235

4.1.2. Making an SD card burner

1). Production card

In order to meet the needs of production burn-in, this method is suitable for mass production burn-in method. The system to be burned is written to the onboard eMMC or spi nand via the system on the SD card. Please follow the steps below to complete the process:

The preparation work is the same as in section 4.1, and the operation steps are more or less the same.

First open PhoenixCard program under windows (the location of the program is explained in chapter 4.1 Preparation). Insert the 16GB SD card into the windows USB port through the SD card reader, as shown in the figure below, select the "Firmware" path; choose "Mass Production Card" and click the "Burn Card" button. Click the "Burn" button to finish automatically.

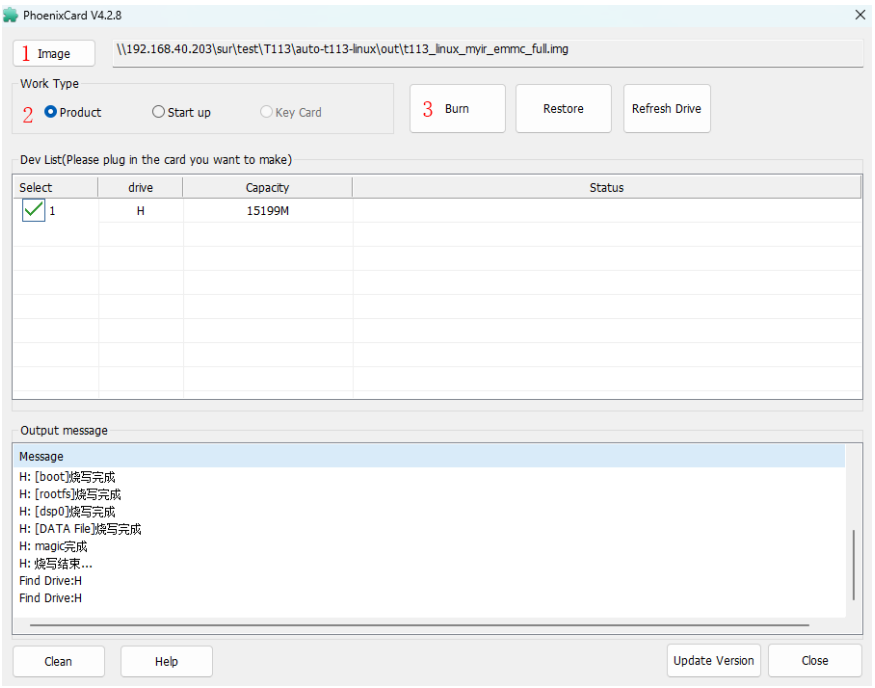

Figure 4-4. production card

The subsequent production method is consistent with section 4.1.1, so we won't go over it here.

2). Verify eMMC boot

MYIR Electronics Limited
Web: www.myirtech.com Mail: sales@myirtech.com Tel: 0755-22316235

Insert the SD burner card into the SD card slot (J5) of the emmc or nand board, then power up the development board and wait for the burner to finish printing, then power down and unplug the SD card and restart the development board.

Note: If the development board is started without unplugging the SD card after burning, the image will be re-burned.

Because the burn-in printed information is more, only part of the important data intercepted to display, the last Flash Success message indicates the completion of the burn-in (nand board burn-in process information is different from the emmc board, but the final burn-in will appear Flash Success message)

```
[07.110]begin to download part boot
partdata hi 0x0
partdata lo 0x1423000
sparse: bad magic
[09.469]successed in writting part boot
origin_verify value = 262bacf0, active_verify value = 262bacf0
[09.959]successed in verify part boot
[09.963]successed in download part boot
[09.966]begin to download part rootfs
partdata hi 0x0
partdata lo 0xf962f40
chunk 0(8515)
chunk 1(8515)
chunk 2(8515)
chunk 3(8515)
chunk 4(8515)
chunk 5(8515)
chunk 6(8515)
chunk 7(8515)
chunk 8(8515)
chunk 9(8515)
chunk 10(8515)
......
[84.409]successed in downloading boot0
```

```
current bitmap buffer size is 0 and new bitmap size is 483.
pitch abs is 21 and glyph rows is 23.
current bitmap buffer size is 483 and new bitmap size is 529.
pitch abs is 23 and glyph rows is 23.
CARD OK
[84.431]sprite success
sprite_next_work=3
next work 3
SUNXI_UPDATE_NEXT_ACTION_SHUTDOWN
sunxi board shutdown
[87.441][mmc]: mmc exit start
[87.459][mmc]: mmc 2 exit ok
*** Flash Success ***
*** Flash Success ***
*** Flash Success ***
```

5. Adapt to your own hardware platform

In order to adapt to the user's new hardware platform, the first thing you need to know is what resources are provided by MYIR's MYD-YT113X development board, the specific information can be found in the "*MYD-YT113X SDK Release Notes*". In addition, users need to have a more detailed understanding of the CPU chip manual, as well as the MYC-YT113X core board product manual, pin definitions, in order to facilitate the correct configuration and use of these pins according to the actual function.

5.1. Configure sys_config.fex

sys_config.fex is a set of functional configuration files defined by Allwinner for T1. This file can be used to define the pins, attributes, power supply, etc. of each node, so that users can quickly configure the functions of the resources. In order to let users master the sys_config.fex configuration and usage. This chapter will explain how to use

Path to sys_config.fex file:

```
PC$: $HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/myir_xxx/sys_config.fex  (xxx stands for different configurations)
```

Defining the property class method:

```
; version:
; machine:
;----------------------------------------------------------------------------
[product]
version = "100"
machine = "emmc"

;----------------------------------------------------------------------------
;debug_mode     = 0-close printf, > 0-open printf
;----------------------------------------------------------------------------
```

```
[platform]
eraseflag   = 1
debug_mode  = 8

;----------------------------------------------------------------------------------
;storage_type   = boot medium, 0-nand, 1-sd, 2-emmc, 3-nor, 4-emmc3, 5-spinand -1(defualt)auto scan
;----------------------------------------------------------------------------------
[target]
storage_type    = 2
burn_key        = 0

[card0_boot_para]
card_ctrl       = 0
card_high_speed = 1
card_line       = 4
sdc_d1          = port:PF0<2><1><2><default>
sdc_d0          = port:PF1<2><1><2><default>
sdc_clk         = port:PF2<2><1><2><default>
sdc_cmd         = port:PF3<2><1><2><default>
sdc_d3          = port:PF4<2><1><2><default>
sdc_d2          = port:PF5<2><1><2><default>
bus-width = 4
cap-sd-highspeed =
cap-wait-while-busy =
no-sdio =
no-mmc =
sunxi-power-save-mode =

;----------------------------------------------------------------------------------
[card2_boot_para]
card_ctrl       = 2
card_high_speed = 1
```

```
card_line       = 4
sdc_clk         = port:PC02<3><1><3><default>
sdc_cmd          = port:PC03<3><1><3><default>
sdc_d0          = port:PC06<3><1><3><default>
sdc_d1          = port:PC05<3><1><3><default>
sdc_d2          = port:PC04<3><1><3><default>
sdc_d3          = port:PC07<3><1><3><default>
sdc_tm4_hs200_max_freq = 150
sdc_tm4_hs400_max_freq = 100
sdc_ex_dly_used = 2
;sdc_io_1v8     = 1
sdc_tm4_win_th = 8
sdc_dis_host_caps = 0x180
;sdc_erase      = 2
;sdc_boot0_sup_1v8 = 1
;sdc_type       = "tm4"
```

* Due to the need to obtain the relevant license for Allwinner materials, please contact MYIR's technical support to obtain the document "*sys_config.fex usage configuration instructions.pdf*" for the detailed meaning of the above two types of configuration definitions.

5.2. Creating a device tree

5.2.1. Onboard Device Tree

Users can create their own device trees in the BSP source code, and generally do not need to modify the code in the Bootloader section. Users only need to make appropriate adjustments to the Linux kernel device tree according to the actual hardware resources. The device trees in each part of the BSP of MYD-YT113X are listed here for the user's reference, as shown in the following table:

Table 5-1. MYD-YT113X device tree list

Projects	Device Tree	Description
U-boot	uboot-board.dts	dts used by uboot
	myir-t113-lvds.dtsi	7" single channel LVDS device tree configuration

	myir-t113-lvds-dual.dtsi	19" dual LVDS device tree configuration
Kernel	board.dts	Base plate configuration resource, pin resource configuration
	sun8iw20p1.dtsi	Core resourcing
	myir-t113-lvds.dtsi	7" single channel LVDS device tree configuration
	myir-t113-lvds-dual.dtsi	19" dual LVDS device tree configuration

This SDK source code provides t113 and t113_i two models of five types of image type configuration, their use of board.dts, uboot-board.dts in the following path respectively:

PC$: $HOME/T113X/auto-t113x-linux/device/config/chips/t113/configs/ myir-image-yt113s3-emmc-core

PC$: $HOME/T113X/auto-t113x-linux/device/config/chips/t113/configs/ myir-image-yt113s3-emmc-full

PC$: $HOME/T113X/auto-t113x-linux/device/config/chips/t113/configs/ myir-image-yt113s3-nand

PC$: $HOME/T113X/auto-t113x-linux/device/config/chips/t113_i/configs/ myir-image-yt113i-core

PC$: $HOME/T113X/auto-t113x-linux/device/config/chips/t113_i/configs/ myir-image-yt113i-full

The device tree configuration file for using LVDS under uboot is at the following path:

PC$: $HOME/T113X/T113Xauto-t113x-linux/brandy/brandy-2.0/u-boot-2018/arch/arm/dts

The device tree configuration file for using LVDS under kernel is in the following path:

PC$: $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4/arch/arm/boot/dts

5.2.2. Adding device trees

The Linux kernel device tree is a data structure that describes the on-chip and off-chip device information in a unique syntax format. It is passed from the BootLoader to the kernel, which parses it to form a dev structure associated with the driver for use by the driver code.

Under "$HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4/arch/arm/boot/dts" in the kernel source code, you can see a large number of platform device trees. If the device tree is suitable for MYD-YT113X, you can add a custom device tree under the current path, such as:

```
PC$: $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4/arch/arm/boot/dts
```

The resources related to the MYC-YT113X core board are written into "sun8iw20p1.dtsi" and "board.dts". Other extended interfaces and devices can be referenced as follows (for reference only):

If you want to modify the configuration of different monitors, you need to modify the following files (since only the full image provides a graphical display, only the "board.dts" file in the following path will be modified)

```
$HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/myir-image-yt113s3-emmc-full/board.dts
```

The board.dts configuration is as follows:

The following example is a 7" LVDS display configuration. If you need to change to a 19" dual LVDS display, turn on the corresponding configuration comment and then comment on the original display configuration, as it does not support simultaneous display.

```
// $HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/myir-image-yt113s3-emmc-full/board.dts

#include "sun8iw20p1.dtsi"
#include "myir/myir-t113-lvds.dtsi"
//#include "myir/myir-t113-lvds-dual.dtsi"
```

```
/{
        model = "sun8iw20";
        compatible = "allwinner,r528", "arm,sun8iw20p1";

        reg_vdd_cpu: vdd-cpu {
                compatible = "pwm-regulator";
                pwms = <&pwm 3 5000 0>;
                regulator-name = "vdd_cpu";
                regulator-min-microvolt = <810000>;
                regulator-max-microvolt = <1160000>;
                regulator-settling-time-us = <4000>;
                regulator-always-on;
                regulator-boot-on;
                status = "okay";
        };
```

After modifying board.dts, you also need to modify the uboot-board.dts file in the same level directory, the modification method is the same as board.dts.

```
// $HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/myir-image-yt113s3-emmc-full/uboot-board.dts

/*
 * Allwinner Technology CO., Ltd.
 */

/dts-v1/;

/* optee used 7MB: SHM 2M: OS: 1M: TA:4M*/
/*/memreserve/ 0x41900000 0x00100000;*/
/* DSP used 1MB */
/* /memreserve/ 0x42000000 0x00100000; */

#include "sun8iw20p1.dtsi"
#include "myir-t113-lvds.dtsi"
```

```
//#include "myir-t113-lvds-dual.dtsi"
.....
```

Finally the dual LVDS display also needs to be modified in the env.cfg file by adding lvds_if_reg to the bootcmd

```
$HOME/T113X/auto-t113x-linux/device/config/chips/t113/configs/ myir-image-yt113s3-emmc-full/longan/env.cfg


.............
boot_dsp0=sunxi_flash read 43000000 ${dsp0_partition};bootr 43000000 0 0
boot_normal=sunxi_flash read 43000000 boot;bootm 43000000
boot_recovery=sunxi_flash read 43000000 recovery;bootm 43000000
boot_fastboot=fastboot
lvds_if_reg=mw.l 0x05461084 0xE0100000

#uboot system env config
bootdelay=3
#default bootcmd, will change at runtime according to key press
#default nand boot
bootcmd=run lvds_if_reg setargs_mmc boot_dsp0 boot_normal
```

5.3. Configuring CPU Function Pins

Realizing the control of a functional pin is one of the more complex system development processes, which includes the configuration of the pin, the development of the driver, the implementation of the application and other steps, this section does not specifically analyze each part of the development process, but rather to explain the implementation of functional pin control by example.

5.3.1. GPIO pin multiplexing

GPIO: General-purpose input/output, a very important resource in embedded devices, which can be used to output high and low levels or read the state of the pins as high or low.

MYD-YT113X package has peripheral controllers, these peripheral controllers and external devices are generally through the control of GPIO to achieve, and the

GPIO is used by peripheral controllers we call multiplexing (Alternate Function), to give them more complex functions, such as the user can use the GPIO port and external hardware for data interaction (such as UART), control the work of hardware (such as LED, buzzer, etc.), read the working status of hardware signals (such as interrupt signals), so the GPIO port is very widely used.

1). GPIO pin multiplexing uart4 function

To configure uart4 function, you need to find which pins can be multiplexed into uart4 function first, the multiplexing relationship between pins can refer to "***MYC-YT113X-PinList***" core board PinList list, the following will take uart4 as an example to introduce how to configure gpio pins. (Explained by myir-image-yt113s3-emmc-full mirror configuration)

- **Reference uart4 node**

PC$: $HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/myir-image-yt113s3-emmc-full/board.dts

```
&uart4 {
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&uart4_pins_a>;
        pinctrl-1 = <&uart4_pins_b>;
        status = "okay";
};
```

- **View pin schematic connections**

Check the schematic of the base board can be seen corresponding to the core module pin 109 and 107, as shown in Figure 5-1:

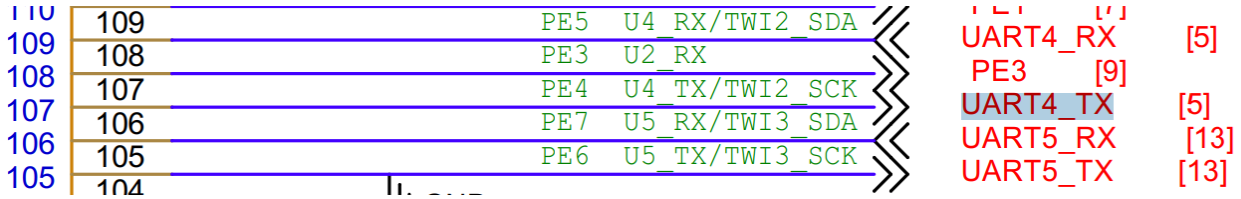

Figure 5-1. Pinout schematic

- **View uart4 core module pinout**

Check the core module PinList pins 109 and 107 can be seen corresponding to PE4 and PE5, as shown in Figure 5-2:

MYC Module Pin No	MYC Module Pin Name	PIn Is GPIO	T113-S3 Pin Name	Default function	IO Voltage
97	GND	/	/	/	0V
98	PE2	PE2	PE2/NCSI0_PCLK/UART2_TX/TWI0_SCK/CLK_FANOUT0/UART0_TX/RGMII_RXD1/RMII_RXD1/PE_EINT2		0/3.3V
99	PE12	PE12	PE12/TWI2_SCK/NCSI0_FIELD/I2S0_DOUT2/I2S0_DIN2/RGMII_TXD3/PE_EINT12		0/3.3V
100	NC	/	/	/	3.3V
101	PE8	PE8	PE8/NCSI0_D4/UART1_RTS/PWM2/UART3_TX/JTAG_MS/MDC/PE_EINT8		0/3.3V
102	PE9	PE9	PE9/NCSI0_D5/UART1_CTS/PWM3/UART3_RX/JTAG_DI/MDIO/PE_EINT9		0/3.3V
103	NC	/	/	/	/
104	GND	/	/	/	0V
105	PE6	PE6	PE6/NCSI0_D2/UART5_TX/TWI3_SCK/SPDIF_IN/D_JTAG_DO/R_JTAG_DO/RGMII_TXCTRL/RMII_TXEN/PE_EINT6		0/3.3V
106	PE7	PE7	PE7/NCSI0_D3/UART5_RX/TWI3_SDA/SPDIF_OUT/D_JTAG_CK/R_JTAG_CK/RGMII_CLKIN/RMII_RXER/PE_EINT7		0/3.3V
107	PE4	PE4	PE4/NCSI0_D0/UART4_TX/TWI2_SCK/CLK_FANOUT2/D_JTAG_MS/R_JTAG_MS/RGMII_TXD0/RMII_TXD0/PE_EINT4		0/3.3V
108	PE3	PE3	PE3/NCSI0_MCLK/UART2_RX/TWI0_SDA/CLK_FANOUT1/UART0_RX/RGMII_TXCK/RMII_TXCK/PE_EINT3		0/3.3V
109	PE5	PE5	PE5/NCSI0_D1/UART4_RX/TWI2_SDA/LEDC_DO/D_JTAG_DI/R_JTAG_DI/RGMII_TXD1/RMII_TXD1/PE_EINT5		0/3.3V
110	PE1	PE1	PE1/NCSI0_VSYNC/UART2_CTS/TWI1_SDA/LCD0_VSYNC/RGMII_RXD0/RMII_RXD0/PE_EINT1		0/3.3V
111	PE0	PE0	PE0/NCSI0_HSYNC/UART2_RTS/TWI1_SCK/LCD0_HSYNC/RGMII_RXCTRL/RMII_CRS_DV/PE_EINT0		0/3.3V

Figure 5-2. Pin Correspondence

- **Configuring Multiplexing Relationships**

From Figure 5-2, we can see that PE4 and PE5 pins can be configured as UART4_TX, UART4_RX.

```
PC$: $HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/myir-image-yt113s3-emmc-full/board.dts

        uart4_pins_a: uart4_pins@0 {
                pins = "PE4", "PE5";
                function = "uart4";
                drive-strength = <10>;
                bias-pull-up;
        };

        uart4_pins_b: uart4_pins@1 {
                pins = "PE4", "PE5";
                function = "gpio_in";
        };
```

- **Add serial port aliases**

At this point, you need to add the serial port alias to the following path.

```
PC$: $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4/arch/arm/boot/dts/sun8iw20p1.dtsi
/ {
```


MYIR Electronics Limited
Web: www.myirtech.com Mail: sales@myirtech.com Tel: 0755-22316235

```
        model = "sun8iw20";
        compatible = "allwinner,sun8iw20p1";
        interrupt-parent = <&gic>;
        #address-cells = <2>;
        #size-cells = <2>;

        aliases {
                serial0 = &uart0;
                serial1 = &uart1;
                serial2 = &uart2;
                serial3 = &uarT1;
                serial4 = &uart4;
                serial5 = &uart5;
};
```

Finally, update the device tree according to Chapter 3.6, Subsection 3, and burn the new device tree to the development board.

5.3.2. Configure the function pin as GPIO function

This example uses the PD20 as a test GPIO and describes how to configure the device nodes in the device tree and for use by the kernel driver in later chapters. This example also gives a reference to control the reset, power and other control functions of external devices. (using emmc full image as an example)

Just add nodes to the device tree。

```
PC$: $HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/myir-image-yt113s3-emmc-full/board.dts

gpioctr_device {
                compatible = "myir,gpioctr";
                status = "okay";
                gpioctr-gpios = <&pio PD 20 1 1 1 1>;
                };
```

5.3.3. LCD resource pin reallocation

MYD-YT113X development board defines and implements many rich functions, but also occupies a large number of pin resources, such as the user directly using MYD-YT113X based on design development, will need to redefine and reconfigure the pins. The following is an example of LCD multiplexing pin function, multiplexing relationship view "*MYC-YT113X-PinList*" document。

See the following directory in the dts file, you can know that the lvds0 function occupies PD0~10 pins, to freely allocate the use of these pins, first of all, these pins will be released。

PC$: $HOME/T113X/T113Xauto-t113x-linux/device/config/chips/t113/configs/myir-image-yt113s3-emmc-full/board.dts

Before modification:

```
/*
 * Allwinner Technology CO., Ltd.
 */

/dts-v1/;

/* optee used 7MB: SHM 2M: OS: 1M: TA:4M*/
/*/memreserve/ 0x41900000 0x00100000;*/
/* DSP used 1MB */
/* /memreserve/ 0x42000000 0x00100000; */

#include "sun8iw20p1.dtsi"
#include "myir/myir-t113-lvds.dtsi"
//#include "myir/myir-t113-lvds-dual.dtsi"

.....
```

```
        lvds0_pins_a: lvds0@0 {
```

```
            allwinner,pins  = "PD0", "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7
", "PD8", "PD9";
            allwinner,pname = "PD0", "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD
7", "PD8", "PD9";
            allwinner,function = "lvds0";
            allwinner,muxsel = <3>;
            allwinner,drive = <3>;
            allwinner,pull = <0>;
        };

        lvds0_pins_b: lvds0@1 {
            allwinner,pins  = "PD0", "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD7
", "PD8", "PD9";
            allwinner,pname = "PD0", "PD1", "PD2", "PD3", "PD4", "PD5", "PD6", "PD
7", "PD8", "PD9";
            allwinner,function = "lvds0_suspend";//io_disabled
            allwinner,muxsel = <7>;
            allwinner,drive = <3>;
            allwinner,pull = <0>;
        };
```

The following is the modified display, here we need to pay attention to the display of the reference to comment out (#include "myir-t113-lvds.dtsi")。

Note: these pins are also referenced in “sun8iw20p1.dtsi” and “uboot-board.dts” in the same level directory, so they should also be modified in this way, which is not shown here。

After modification：

```
/*
 * Allwinner Technology CO., Ltd.
 */

/dts-v1/;
```

```
/* optee used 7MB: SHM 2M: OS: 1M: TA:4M*/
/*/memreserve/ 0x41900000 0x00100000;*/
/* DSP used 1MB */
/* /memreserve/ 0x42000000 0x00100000; */

#include "sun8iw20p1.dtsi"
//#include "myir/myir-t113-lvds.dtsi"
//#include "myir/myir-t113-lvds-dual.dtsi"

.....

        lvds0_pins_a: lvds0@0 {
                allwinner,pins  = " ";
                allwinner,pname = " ";
                allwinner,function = "lvds0";
                allwinner,muxsel = <3>;
                allwinner,drive = <3>;
                allwinner,pull = <0>;
        };

        lvds0_pins_b: lvds0@1 {
                allwinner,pins  = " ";
                allwinner,pname = " ";
                allwinner,function = "lvds0_suspend";//io_disabled
                allwinner,muxsel = <7>;
                allwinner,drive = <3>;
                allwinner,pull = <0>;
};
```

5.4. Use your own configured pins

The pins configured in the device tree of u-boot or Kernel can be used in Kernel to control the pins。

5.4.1. Use of GPIO pins in the kernel driver

- **Use of standalone IO drivers**

In the first device tree example in section 5.3.2, the gpio node information has been defined, the following will use the kernel driver to implement the GPIO control (the PD20 pin to set 1 and set 0, if you need to test the change in pin level using a multimeter).

```
//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>

static int major = 0;
static struct class *gpioctr_class;
static struct gpio_desc *gpioctr_gpio;


static ssize_t gpio_drv_read (struct file *file, char __user *buf, size_t size, loff_t *offs
et)
{
```


MYIR Electronics Limited
Web: www.myirtech.com Mail: sales@myirtech.com Tel: 0755-22316235

```
        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;
}

static struct file_operations gpioctr_drv = {
        .owner      = THIS_MODULE,
```

```
        .open    = gpio_drv_open,
        .read    = gpio_drv_read,
        .write   = gpio_drv_write,
        .release = gpio_drv_close,
};

static int chip_demo_gpio_probe(struct platform_device *pdev)
{
        gpioctr_gpio = gpiod_get(&pdev->dev, "gpioctr", 0);
        if (IS_ERR(gpioctr_gpio)) {
            dev_err(&pdev->dev, "Failed to get GPIO for led\n");
            return PTR_ERR(gpioctr_gpio);
        }

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

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

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

    return 0;

}

static int chip_demo_gpio_remove(struct platform_device *pdev)
{
```

```
        device_destroy(gpioctr_class, MKDEV(major, 0));
        class_destroy(gpioctr_class);
        unregister_chrdev(major, "myir_gpioctr");
        gpiod_put(gpioctr_gpio);

    return 0;
}

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

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,
    },
};

static int __init gpio_init(void)
{
    int err;
    err = platform_driver_register(&chip_demo_gpio_driver);

        return err;
}

static void __exit gpio_exit(void)
{
    platform_driver_unregister(&chip_demo_gpio_driver);
```

```
}

module_init(gpio_init);
module_exit(gpio_exit);

MODULE_LICENSE("GPL");
```

Compiling the driver code into modules using a separate Makefile can also be configured directly into the kernel。

- **Configure the driver example into the kernel**

Create a new "gpioctr.c" file in the "/drivers/char" folder of the kernel source code, copy the above driver code into it, and modify the "Kconfig " and "Makefile" and "defconfig".

In the "/drivers/char/Kconfig" file add:

```
config SAMPLE_GPIO
        tristate "this is a gpio test driver"
```

In the "/drivers/char/Makefile" file, add:

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

In the "arch/arm/configs/myd_yt113s3_emmc_full_defconfig" file, add

```
CONFIG_SAMPLE_GPIO=y
```

Update the kernel according to section 3.6, and then burn it to the development board.

- **Driver examples compiled into separate modules**

Add "gpioctr.c" to the working directory and copy the above driver code, then write a separate Makefile program in the same directory.

```
PC$: $HOME/gpioctr$ ls
gpioctr.c  Makefile
PC$: $HOME/gpioctr$ vi Makefile
KERN_DIR = $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4/
```

```
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

# ab-y := a.o b.o
# obj-m += ab.o
```

Load SDK environment variables to the current shell.(Load according to your actual installation compilation chain directory)

```
PC$ export PATH=$PATH:/opt/gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi/bin
```

Execute the make command to generate the "gpioctr.ko" driver module file。

```
PC$: $HOME/gpioctr$ make
make -C $HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4/ M=`pwd` modules
make[1]: Entering directory '$HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4/'
  CC [M]  /home/sur/gpioctr/gpioctr.o
  MODPOST /home/sur/gpioctr/Module.symvers
  CC [M]  /home/sur/gpioctr/gpioctr.mod.o
  LD [M]  $Home/gpioctr/gpioctr.ko
make[1]: Leaving directory '$HOME/T113X/T113Xauto-t113x-linux/kernel/linux-5.4/'
PC$: $HOME/gpioctr$ ls
gpioctr.c  gpioctr.ko  gpioctr.mod  gpioctr.mod.c  gpioctr.mod.o  gpioctr.o  Makefile  modules.order  Module.symvers
```

After successful compilation, the gpioctr.ko file can be transferred to the development board via Ethernet, WIFI, U disk and other transfer media to load the driver using the insmod command.

Different external devices have their own independent driver code and architecture implementation, in different peripheral driver modification, debugging, need to comply with their own driver framework. Such as touch screen, keyboard, etc. need to use input driver architecture; ADC and DAC use IIO architecture, display devices use DRM driver architecture, etc., this section does not do specific explanation of all driver development。

5.4.2. User space using GPIO pins

The Linux operating system architecture is divided into user state and kernel state (or user space and kernel). The user state is the active space for upper-level applications, which must rely on the resources provided by the kernel, including CPU resources, storage resources, and I/O resources. In order for the upper-layer application to access these resources, the kernel must provide an interface for the upper-layer application to access them: i.e., system calls。

A shell is a special application, commonly known as a command line, which is essentially a command interpreter that goes down to system calls and up to various applications. The reason for using Shell scripts, which can usually achieve a very large function in just a few lines, is that these Shell statements usually do a layer of encapsulation of system calls. To facilitate user-system interaction。

This section describes how to use the GPIO pins in the user state to control the three basic ways。

- Shell Command
- System calls
- Library functions

- **Shell implementation of pin control**

Shell control pins are essentially implemented by calling the file manipulation interface provided by Linux. Using the debug interface to configure, each gpio's PIN has four attributes, namely, multiplexing (function), data (data), drive capability (dlevel), and pull up and down status (pull). The operation is as follows:

```
[root@myir:/]# mount -t debugfs none /sys/kernel/debug
[root@myir:/]# cd /sys/kernel/debug/sunxi_pinctrl
```

To view the configuration of the pin:

```
[root@myir:/]# echo PD20 > sunxi_pin
[root@myir:/]# cat sunxi_pin_configure
pin[PD20] funciton: f
pin[PD20] data: 0
pin[PD20] dlevel: 20mA
pin[PD20] pull up: 0xffffff
pin[PD20] pull down: 0xffffff
pin[PD20] pull disable: 0x0
```

Modify pin properties:

```
[root@myir:/]# echo PD20 1 > pull
[root@myir:/]# cat sunxi_pin_configure
pin[PD20] funciton: f
pin[PD20] data: 0
pin[PD20] dlevel: 20mA
pin[PD20] pull up: 0x1
pin[PD20] pull down: 0xffffff
pin[PD20] pull disable: 0xffffff
```

- **Library functions for pin control**

Starting with Linux 4.8, Linux introduced a new way of operating gpio, the GPIO character device. Instead of using the previous SYSFS method of operating GPIOs under the "/sys/class/gpio" directory, each GPIO group has a corresponding gpiochip file under "/dev", e.g. "/dev/gpiochip0 corresponds to GPIOA, /dev/gpiochip1 corresponds to GPIOB", etc.

Libgpiod library function implementation is based on C language due to the way of gpiochip, so developers have implemented Libgpiod to provide some tools and a simpler C API interface. libgpiod (Library General Purpose Input/Output device) provides a complete API to developers, and also provides some applications under user space to operate GPIO.

The following will use PD20 as the operation GPIO pin to implement the C code control example (alternately set high and low).

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

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

	strcpy(chrdev_name, "/dev/gpiochip0");

	/* Open device: gpiochip5 for GPIO bank F */
	fd = open(chrdev_name, 0);
	if (fd == -1) {
		ret = -errno;
		fprintf(stderr, "Failed to open %s\n", chrdev_name);

	return ret;
	}

	/* request GPIO line: P4_1 */
	req.lineoffsets[0] = 33;
	req.flags = GPIOHANDLE_REQUEST_OUTPUT;
	memcpy(req.default_values, &data, sizeof(req.default_values));
```

```
    strcpy(req.consumer_label, "P4_1");
    req.lines = 1;

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

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

    /* release line */
    ret = close(req.fd);
    if (ret == -1) {
        perror("Failed to close GPIO LINEHANDLE device file");
        ret = -errno;
    }
    return ret;
```

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

```
PC$ export PATH=$PATH:/opt/gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi/bin
```

Use the compile command $CC to generate the executable gpioctr.

```
PC$ $HOME/gpioctr$ $CC gpioctr-test.c -o gpioctr-test
```

Copy the executable file to the /usr/sbin directory of the development board through the network (scp, etc.), u disk and other transfer media, you can enter commands in the terminal can be run directly, using a multimeter can see the P4_1 pin alternately high and low level changes.

```
[root@myir:/]# ./gpioctr-test
```

- **System call for pin control**

The operating system provides a set of "special" interfaces for user programs to call. For example, the user can request the system to open a file, close a file, or read or write a file through a file system-related call, get the system time or set a timer through a clock-related call, etc.

The pins are also resources that can be controlled by system calls. In 5.3.2 we have completed the implementation of the driver for the pins, so we can make system calls to control the pins controlled by this driver.

```
//gpiotest.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

/*
 * ./gpiotest /dev/myir_gpioctr0 on
 * ./gpiotest /dev/myir_gpioctr0 off
 */
int main(int argc, char **argv)
{
        int fd;
        char status;
```

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

    fd = open(argv[1], O_RDWR);
    if (fd == -1)
    {
        printf("can not open file %s\n", argv[1]);
        return -1;
    }

    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-API.c file and load the SDK environment variables into the current shell:

```
PC$ export PATH=$PATH:/opt/gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi/bin
```

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

```
PC$ $HOME/gpioctr$ $CC gpiotest-API.c -o gpiotest-API
```

Copy the executable file to the "/usr/sbin" directory of the development board through the network (scp, etc.), u disk and other transfer media, you can enter commands in the terminal can be run directly (on means set high, off means set low).

```
[root@myir:/]# ./gpiotest-API /dev/myir_gpioctr0 on
[root@myir:/]# ./gpiotest-API /dev/myir_gpioctr0 off
```

6. How to add your application

The porting of Linux applications is usually divided into two phases, the development and debugging phase and the production deployment phase. The development debugging phase allows us to cross-compile our application using the Mil-built SDK and copy it remotely to the target host for testing, while the production deployment phase requires writing recipe files for the application and building the production image using Buildroot.

6.1. Makefile based applications

A Makefile is actually a document that defines a set of compilation rules, it records the details of how the source code is compiled! Once the Makefile is written, only one make command is needed and the whole project is compiled completely automatically, which greatly improves the efficiency of software development. Makefile is commonly used in the development of Linux programs, whether kernel, driver, or application.

make is a command tool that interprets the instructions in a makefile. When make is executed, it searches for the Makefile (or makefile) text file in the current directory and performs the corresponding operation. make automatically determines whether the original file has been changed and automatically recompiles the changed source code.

The following will be a practical example (in the MYD-YT113X development board to implement the key control LED light switch) to describe the Makfile writing and make execution process, Makefile has its own set of rules.

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

- target can be an object file (target file), or an executable file, or a label.
- prerequisites are the files or targets needed to generate that target.
- command is also the command that make needs to execute.

```
sur@ubuntu:~/key-led$ vi Makefile
TARGET = $(notdir $(CURDIR))
objs := $(patsubst %c, %o, $(shell ls *.c))
```

```
$(TARGET)_test:$(objs)
	$(CC) -o $@ $^
%.o:%.c
	$(CC) -c -o $@ $<
clean:
	rm -f  $(TARGET)_test *.all *.o
	${CC} -I . -c key_led.c
```

- $(notdir $(path)): means to remove the path name from the path directory, leaving only the current directory name, for example, the current Makefile directory is /home/sur/key_led, the execution will become TARGET = key_led
- $(patsubst pattern, replacement,text) : Replace the characters in the text that match the format "pattern" with replacement, such as $(patsubst %c, %o, $(shell ls *.c)), which means that the files in the current directory with the .c suffix are listed first, and then replaced with the .o suffix
- CC: Name of C compiler
- CXX: Name of C++ compiler
- clean: It is an agreed target

Key_led implementation code is as follows:

```
sur@ubuntu:~/key-led$ vi key_led.c
/File: Key_led.c
#include <linux/input.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
/* ./key_led /dev/input/event0 noblock */
int main(int argc, char **argv)
{
	int fd,bg_fd;
	int err, len, i;
	unsigned char flag;
	unsigned int data[1];
	char *bg = "/sys/class/leds/LED0/trigger";
	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);
                    printf("flag =%d\n",flag);
                    if(flag == '0')
                    {
                        system("echo heartbeat > /sys/class/leds/LED0/trigger");
//led off
                        //system("echo 0 > /sys/class/leds/LED0/brightness"); //l
ed off
                    }
                    else
                    {
                        system("echo none > /sys/class/leds/LED0/trigger"); //led
 off
                        sleep(3);
                        system("echo heartbeat > /sys/class/leds/LED0/trigger");
                    }
                }
            }
        }
        return 0;
```

Use the make command to compile and generate an executable file on the target machine.

Load the SDK environment variables to the current shell:

```
PC$ export PATH=$PATH:/opt/gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi/bin
```

make:

```
PC$ $HOME/key-led$ make
PC$ $HOME/key-led$ ls
key_led.c  key_led.o  key-led_test  Makefile
```

As you can see from the results of the previous command, the compiler used is the compiler created by setting the CC variables defined in the script, and the key-led_test executable is copied to the /usr/sbin directory of the development board via network (scp, etc.), u disk, and other transfer media:

```
[root@myir:/]# key-led_test /dev/input/event0
```

Note: If you build the target executable using the cross toolchain compiler and the architecture of the build host is different from the architecture of the target machine, you need to run the project on the target device.

6.2. Qt based applications

Qt is a cross-platform graphical application development framework that is used on different size devices and platforms, while providing different copyright versions for users to choose from. MYD-YT113X uses Qt version 5.12 for application development. In Qt application development, it is recommended to use QtCreator IDE, which can develop Qt applications under Linux PC and automatically cross-compile them into ARM architecture programs for development boards.

6.2.1. QtCreator installation and configuration

Get qtcreator installation package from QT official website or MYIR official package QT official website download：
https://download.qt.io/development_releases/qtcreator/.

The QtCreator installer is a binary program that can be directly executed to complete the installation . /qt-creator-opensource-linux-x86_64-5.0.0-rc1.run. For installation and configuration details, please see *"MYD-YT113X_MEasy HMI Software Development Guide"* or get more development guidance from the official QTCreator website https://www.qt.io/product/development-tools。

6.2.2. MEasy HMI2.x Compile and run

MEasy HMI 2.x is a set of QT5-based HMI framework developed by MYIR. The project uses a mixture of QML and C++ programming, using QML to build UI efficiently and conveniently, while C++ is used to implement business logic and complex algorithms.

The source code of MEasy HMI2.x project "MYD-YT113X-2023xxx\04_Sources\mxapp2.tar.gz" is available in MYIR's software distribution package. It can be loaded and compiled by Qtcreator, remote debugging, etc. See *"MYD-YT113X_MEasy HMI Software Development Guide"*.

7. References

- **Linux kernel Open Source Community**
  https://www.kernel.org/
- **Buildroot Official Site**
  https://buildroot.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 Inteiligent Park, Bantian, Longgang District.

Support Email: support@myirtech.com

Sales Email: sales@myirtech.com

Phone: +86-755-22984836

Fax: +86-755-25532724

Website: www.myirtech.com