Rockchip RK2118 RT-Thread Quick Start

ID: RK-JC-YF-589

Release Version: V1.0.1

Release Date: 2024-03-15

Security Level: □Top-Secret □Secret □Internal ■Public

Rockchip Electronics Co., Ltd.

No.18 Building, A District, No.89, software Boulevard Fuzhou, Fujian,PRC

Website: www.rock-chips.com

Customer service Tel: +86-4007-700-590

Customer service Fax: +86-591-83951833

Customer service e-Mail: fae@rock-chips.com

Preface

Overview

The document presents the basic usage of Rockchip RK2118 RT-Thread SDK, aiming to help engineers get started with RK2118 RT-Thread SDK faster.

Product Version

Chipset	Kernel Version
RK2118	RT-Thread v4.1.x

Intended Audience

This document (this guide) is mainly intended for:

Technical support engineers

Software development engineers

Revision History

Date	Version	Author	Revision History
2024-02-26	V1.0.0	Roger Hu	Initial version
2024-03-15	V1.0.1	Roger Hu	Add operation debugging description

---

Contents

Set up the Development Environment

The recommended build environment for this SDK is a 64-bit version of Ubuntu 20.04 or Ubuntu 18.04. It has not been tested on other Linux distributions, so it is recommended to install the same distribution as RK developers.

The compilation tools used are SCons + GCC, as officially recommended by RT-Thread. SCons is an open-source build system written in Python, and the GCC cross-compiler is provided by ARM officially. You can directly use the following commands to install all the required tools:

```
sudo apt-get install gcc-arm-embedded scons clang-format astyle libncurses5-dev
build-essential python-configparser
```

Download the compiler from the ARM official website, and specify the path to the toolchain via environment variables, as detailed below:

```
wget https://developer.arm.com/-/media/Files/downloads/gnu/13.2.rel1/binrel/arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz
tar xvf arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz
export RTT_EXEC_PATH=/path/to/toolchain/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-eabi/bin
```

Or use the compiler included in the initial release package of the SDK: arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz, as detailed below:

```
tar -xvf arm-gnu-toolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz
export RTT_EXEC_PATH=/path/to/toolchain/arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-eabi/bin
```

Project Directory

The following is the main SDK directory:

```
├── applications           # Rockchip application demo source
├── AUTHORS
├── bsp                    # Chip related source code
|   ├── rockchip
|   |   ├── common
|   |   |   ├── drivers    # Universal driver of Rockchip OS adaptation layer
|   |   |   ├── hal        # Rockchip HAL (Hardware abstraction layer) implementation
|   |   |   └── tests      # Rockchip driver test code
|   |   ├── rk2118         # RK2118 main directory
|   |   |   ├── board      # Board level configuration
|   |   |   ├── build      # Build main directory and store the intermediate files
|   |   |   ├── drivers    # RK2118 private driver directory
|   |   |   ├── Image      # Stores firmware
|   |   └── tools          # Rockchip commonly used tools
├── ChangeLog.md
├── components             # Various components of the system, including file system, shell and framework layer and other drivers
|   ├── hifi4
|   |   ├── dsp            # DSP code
|   |   ├── rtt            # DSP related code running on mcu
|   |   ├── shared         # mcu/dsp common code
|   |   └── tools          # dsp firmware generation tool
├── documentation          # RT-Thread official documentation
├── examples               # RT-Thread example program and test code
├── include                # RT-Thread official header file directory
├── Kconfig
├── libcpu
├── LICENSE
├── README.md
├── README_zh.md
├── RKDocs                # Rockchip documents
├── src                   # RT-Thread kernel source code
├── third_party           # Directory of third-party code added by Rockchip
```

```
└── tools                  # RT-Thread official tool directory, including
menuconfig and building scripts
```

Project Building Configuration

CPU Firmware Compilation

RT-Thread utilizes SCons for compile control. SCons is an open-source build system written in Python, similar to GNU Make. Unlike traditional Makefile approaches, it uses SConstruct and SConscript files as substitutes. These files are also Python scripts, allowing the use of standard Python syntax for scripting. Therefore, in SConstruct and SConscript files, one can invoke the Python standard library to perform various complex operations, not limited to the rules set by Makefiles.

Automated Compilation and Packaging

To simplify the compilation process, we created a `build.sh` script to encapsulate the compilation and packaging commands. The usage is as follows:

```
cd bsp/rockchip/rk2118
#board name is the name of your board configuration, which can be found in the
board directory; cpu0, cpu1, and dual respectively indicate the compilation for
cpu0, cpu1, and both CPUs together
./build.sh <board name> [cpu0|cpu1|dual]
```

Taking the `adsp_demo` board as an example, since we only provide the configuration for `cpu0` for this board, you can only select `cpu0`. The specifics are as follows:

```
./build.sh adsp_demo cpu0
```

This command actually performs the following operations:

1. Locate the default configuration for cpu0 of the specified board, which is found at: board/adsp_demo/defconfig. Use it to overwrite the default menuconfig configuration file .config in the current directory.
2. Execute the scons menuconfig command. At this point, the menuconfig configuration window will pop up. You can modify the configuration according to your needs and save before exiting. If you wish to use the default configuration, you may choose to exit directly.
3. Execute the scons -j$(nproc) command to compile the firmware for cpu0.
4. Find the default packaging configuration file for cpu0 of the specified board, located at: board/adsp_demo/setting.ini. Search this file to see if it includes the root.img partition, which is used for the root filesystem. If found, and if the partition flag is not set to skip, call the mkroot.sh script to package the resource directory into root.img; otherwise, proceed directly to the next step.
5. Use the configuration file found in the previous step to package the final firmware to be flashed, Firmware.img.

After executing the above commands, these files will be generated in the Image directory:

```
-rw-r--r-- 1 rk rk 1638400 Apr 11 02:41 Firmware.img           # the final
firmware to be flashed
-rw-r--r-- 1 rk rk      16 Apr 11 02:41 Firmware.md5           # firmware MD5
checksum
-rw-rw-r-- 1 rk rk     154 Feb 28 09:04 config.json
-rw-r--r-- 1 rk rk     763 Mar  4 11:39 rk2118_ddr.ini
-rw-rw-r-- 1 rk rk   28672 Apr 11 02:41 rk2118_idb_ddr.img
-rw-r--r-- 1 rk rk   49543 Apr 11 02:41 rk2118_loader_ddr.bin
-rw-r--r-- 1 rk rk     763 Mar  4 12:35 rk2118_no_ddr.ini
-rw-r--r-- 1 rk rk 4325376 Mar 27 03:23 root.img              # root filesystem
image
-rw-r--r-- 1 rk rk  166916 Apr 11 02:41 rtthread.img
```

Compilation Results Cleanup

The SCons build system by default uses MD5 to determine whether files need to be recompiled. If the content of your file has not changed, but only the timestamp has (for example, updated via the `touch` command), it will not recompile that file or its dependencies. Additionally, if you modify unrelated content, such as code comments, it will compile but not link, because the content of the obj file has not changed. Therefore, if you encounter issues where modifications do not take effect during the development process, it is recommended to perform a cleanup before compiling, using the following command:

```
scons -c
```

If anomalies still occur after performing the above cleanup, you can forcefully delete all intermediate files using the following command:

```
rm -rf build
```

For other SCons commands, you can refer to the help or documentation.

```
scons -h
```

Manual Compilation and Packaging

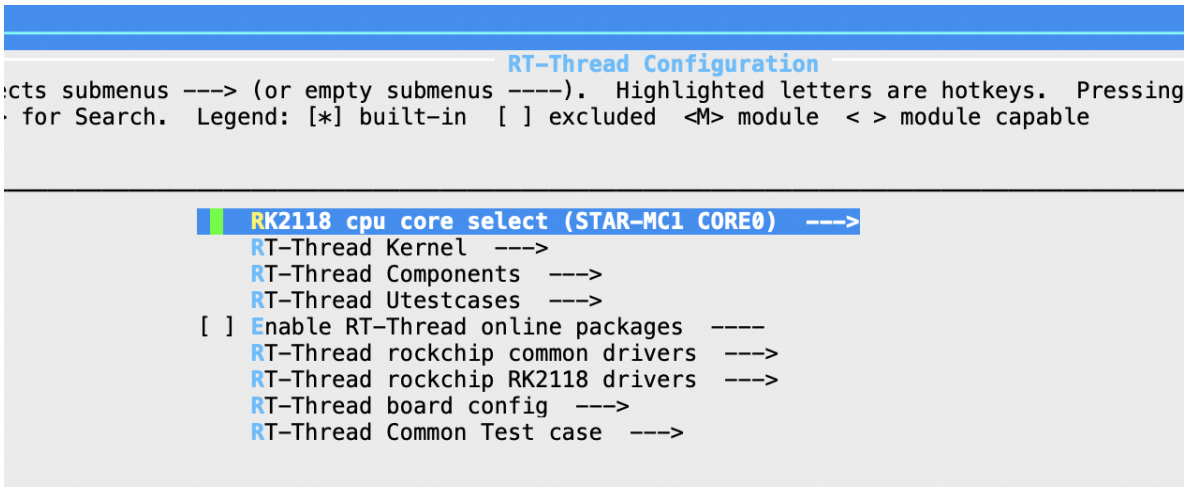
Although using the `build.sh` script for one-click compilation is convenient, it updates the `rtconfig.h` module configuration file each time, which causes the entire CPU project to be recompiled, making it slightly slower than manual compilation. Therefore, manual compilation is often used during development to speed up the process. The manual compilation steps can be divided into three phases: module configuration, compilation, and packaging. If your module configuration does not need changes, you can skip the module configuration step. Similarly, if the CPU firmware does not need changes, you can skip the compilation step and proceed directly to packaging.

Module Configuration

The purpose of module configuration is to select the necessary modules based on the actual needs of the product. This can be done using the following command:

```
cd bsp/rockchip/rk2118
# First, use a base configuration from your board to overwrite the default
configuration file: .config
cp board/adsp_demo/defconfig .config
# Load the default configurations from .config and launch the graphical
configuration interface
scons --menuconfig
```

At this point, the following interface will pop up. You can enable or disable various modules according to the needs of the product. Exiting and saving the configuration will overwrite the `.config` file, and at the same time, automatically generate an `rtconfig.h` file. These two files contain all the selected configurations, and only the `rtconfig.h` file will be used in the final compilation.



In the above figure, the first item is to select the target CPU you want to compile for. RK2118 has two CPUs: cpu0 and cpu1. The next three items are the standard configurations of RT-Thread, followed by RT-Thread's network package, and the rest are the driver configurations and test cases for BSP.

Common operations of the menuconfig tool are as follows:

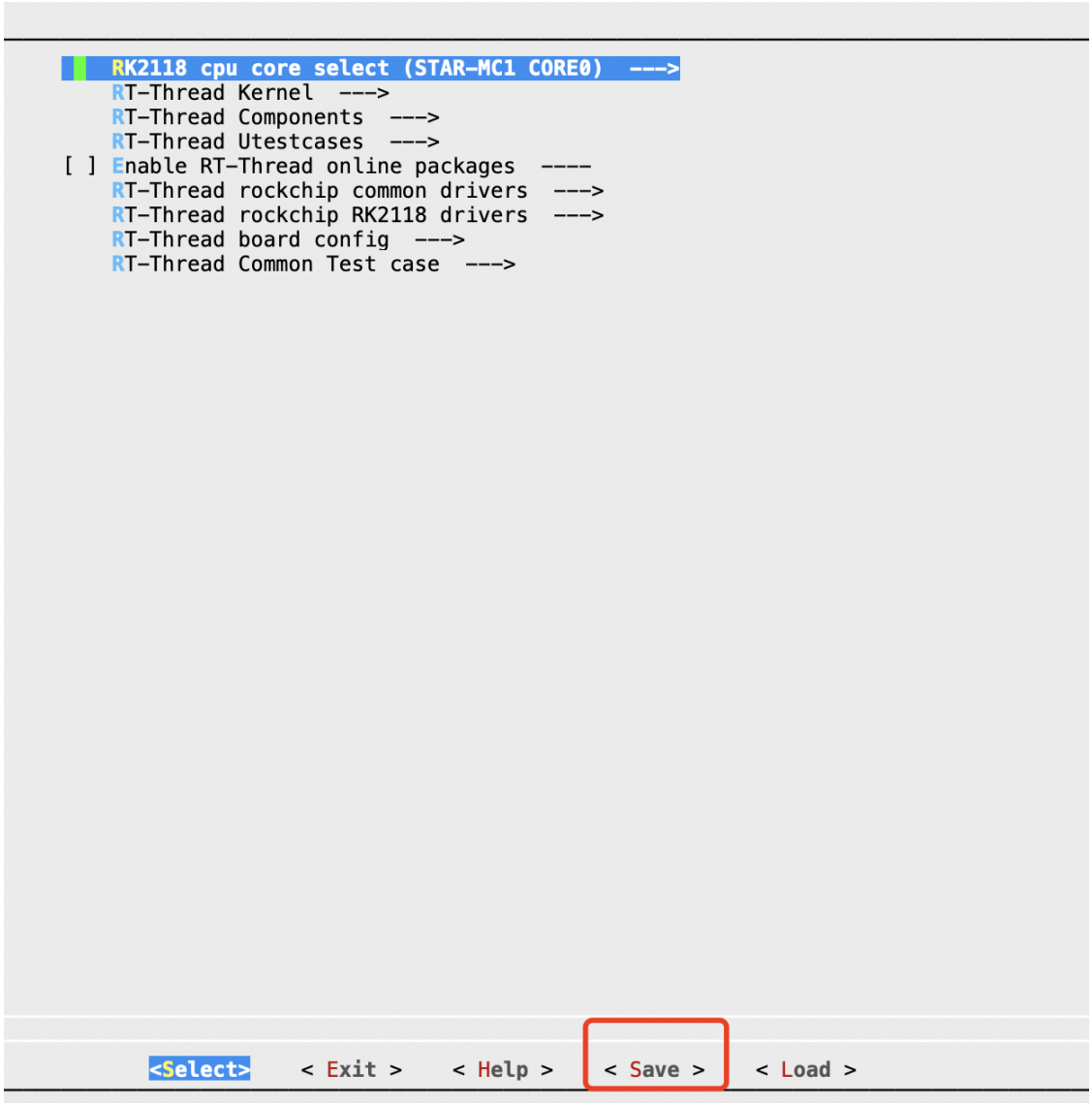
- Up/Down arrows: Move cursor
- Enter: Enter a submenu
- ESC key: Return to the previous menu or exit
- Space, Y key, or N key: Enable/disable [*] configuration options
- Question mark: Brings up a help menu for the highlighted option (to exit the help menu, press Enter)
- Slash (/) key: Search for configuration items

  Each board-level directory contains at least one default configuration file `defconfig`, which is the default configuration for cpu0. If the board level supports dual-cpu boot, there will also be a `cpu1_defconfig`, which is the default configuration for cpu1. When submitting default module configurations, we recommend submitting them to the board-level directory and maintaining this naming convention as much as possible to prevent the automatic compilation script from failing. Do not directly submit to `.config`, so that configurations across different board levels do not overwrite each other. The method is as follows:

```
  # Assume that you have previously run scons --menuconfig and saved the
configuration upon exiting
  cp .config board/xxx/defconfig       # xxx is the name of your board. If it is
the configuration for cpu1, please change it to cpu1_defconfig
```

Please note that as mentioned earlier, only `rtconfig.h` is involved in the compilation, not `.config`. Therefore, if you are not prompted to save the configuration when exiting the `menuconfig` graphical interface, it is because you have not modified any options during this `menuconfig` session. In this case, `rtconfig.h` will not be regenerated, and your `.config` might have been manually overwritten and no longer matches `rtconfig.h`, leading to your current `.config` not being effective during compilation. There are two ways to circumvent this issue, and you can choose one based on your need:

1. Do not exit the `menuconfig` graphical interface directly; instead, save the configuration first and then exit. You can manually save the configuration using this option:



2. After exiting, execute the command `scons --useconfig=.config`. This will forcefully regenerate `rtconfig.h`.

Compilation

Compiling the CPU firmware is very simple; just execute the following command:

```
cd bsp/rockchip/rk2118
scons -j32
```

If the compilation is successful, you will obtain the following files:

```
-rw-r--r-- 1 rk rk  197632 Apr 11 07:08 rtt0.bin           # bin firmware for
cpu0
-rw-r--r-- 1 rk rk 1530528 Apr 11 07:08 rtt0.elf           # elf firmware for
cpu0, with symbol table
-rw-r--r-- 1 rk rk  110592 Apr 11 07:08 rtt1.bin           # bin firmware for
cpu1
-rw-r--r-- 1 rk rk  784044 Apr 11 07:08 rtt1.elf           # elf firmware for
cpu1, with symbol table
```

Packing

Packing is done to integrate the previously compiled firmware, including CPU (NPU firmware is also temporarily placed on the CPU side) and DSP firmware, along with the loader, TFM, etc., into a single image file `Firmware.img`. This image file is what is used for the final flashing.

Manual packing requires specifying the packaging configuration file `setting.ini`. Each board-level directory has at least one such configuration file. If dual-cpu boot is supported, it is generally named `dual_cpu_setting.ini`. The specific command is as follows:

```
./mkimage.sh board/xxx/setting.ini       # xxx is the name of your board
```

DSP Firmware Compilation

Refer to <Rockchip_HiFi4_Quick_Start_EN.pdf>

Firmware Flash

Note

- Since the printed uart and the programmed uart are the same, you need to ensure that the serial port printing client is disconnected first when programming.
- USB cannot be plugged in when programming uart

Windows Upgrade Tool

The tool is located at bsp/rockchip/tools/SocToolKit_v1.9_20240130_01_win.zip

Before programming, make sure the device has been switched to maskrom mode (Press and hold the MASKROM button on the development board, then press the RESET button to power on). Currently, Windows supports uart and usb programming methods. The first step is to select the programming method. For the uart method, you need to specify the serial port number and baud rate. Currently, the highest supported baud rate programming is 1.5M. It also supports full firmware programming and partition programming. The full firmware programming method is as follows:

step1: Choose the correct serial port

step2: Set the baud rate to 1500000

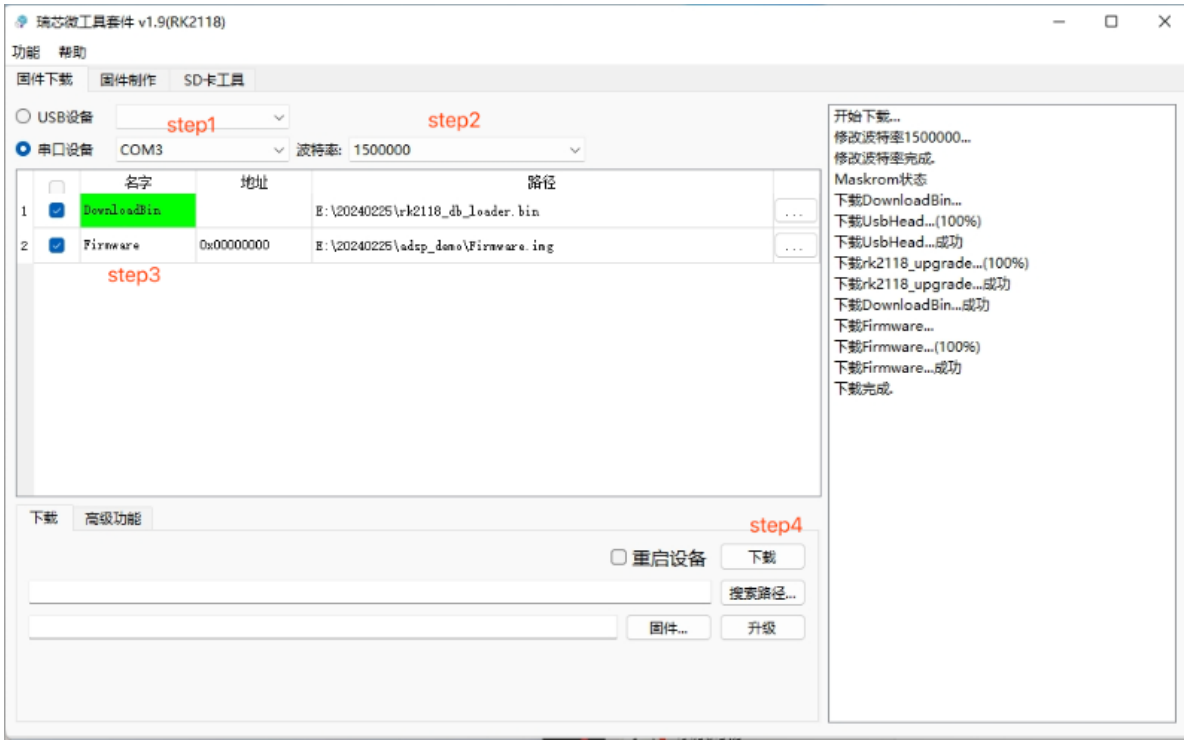
step3: To add firmware: Right-click in a blank area, select 'Add', and then fill in the name, address, and path. After that:

Select loader: bsp/rockchip/rk2118/rkbin/rk2118_db_loader.bin

Select Firmware.img: bsp/rockchip/rk2118/Image/Firmware.img, the address is 0x0

Right-click and choose 'Save Configuration' to your own config_xxx.json, so that you can directly use the saved firmware configuration next time.

step4: Start downloading

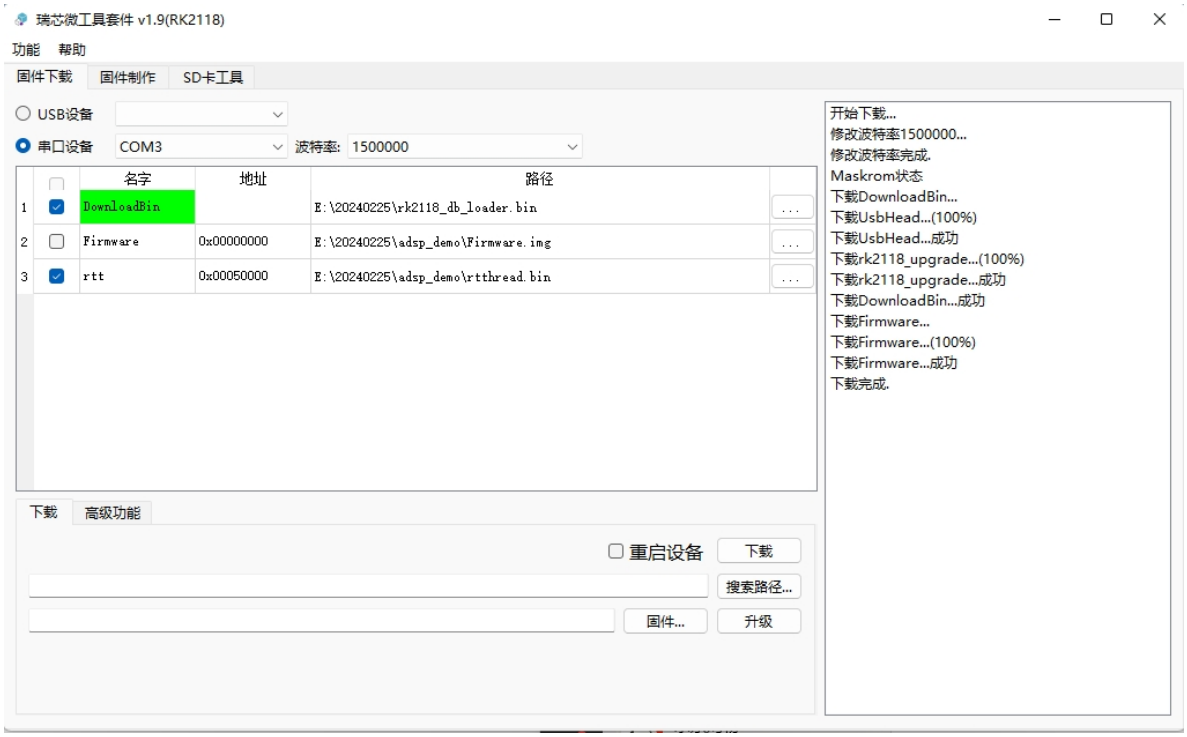


If you want to burn a specified partition (For example, if only rtt0.bin is modified, there is no need to re-upgrade Firmware.img, which can enhance efficiency), you need to specify the address of the partition. The burning method is as follows:

Select loader: bsp/rockchip/rk2118/rkbin/rk2118_db_loader.bin

Select the partition to be upgraded, such as rtthread: bsp/rockchip/rk2118/rtt0.bin, and find the address from the corresponding setting.ini, for example:

```
[UserPart3]
Name=OSA
Type=0x200
PartOffset=0x280  // the location of rtt0.bin
                  // the unit is sector, which needs to be multiplied by 512 to
convert to bytes, such as 0x280 X 512 = 0x5000
PartSize=0xa00
Flag=
File=../../rtt0.bin
```



Linux Upgrade Tool and Command

UART Programming

The default ttyUSB device under Linux does not have write permissions. Modify the permissions through sudo chmod 666 /dev/ttyUSBx (x is your serial port number) before burning.

If you don't want to change the serial port permissions every time, you can add default write permissions in the following way:

```
sudo vi /etc/udev/rules.d/70-rk.rules
# Add the following content when creating, save and exit
KERNEL=="ttyUSB[0-9]*", MODE="0666"
```

The programming method is as follows:

```
sudo chmod 666 /dev/ttyUSB0
# First switch to maskrom and burn db loader
../tools/upgrade_tool/upgrade_tool db /dev/ttyUSB0 ./Image/rk2118_db_loader.bin
# Complete firmware programming
../tools/upgrade_tool/upgrade_tool wl /dev/ttyUSB0 0 ./Image/Firmware.img
# After the complete firmware is burned, the development process can only burn
the updated firmware. For example, the following command is to burn the rtt
firmware of cpu0 alone. The third parameter is the firmware location. By viewing
the UserPart3 partition in setting.ini, the PartOffset=0x280, so fill in 0x280
here
../tools/upgrade_tool/upgrade_tool wl /dev/ttyUSB0 0x280 ./rtthread.bin
```

USB Programming

The programming method is as follows:

```
# First switch to maskrom and burn db loader
../tools/upgrade_tool/upgrade_tool db ./Image/rk2118_db_loader.bin
# Complete firmware programming
../tools/upgrade_tool/upgrade_tool wl 0 ./Image/Firmware.img
# After the complete firmware is burned, the development process can only burn
the updated firmware. For example, the following command is to burn the rtt
firmware of cpu0 alone. The location information needs to be kept equal to the
PartOffset of the partition in setting.ini.
../tools/upgrade_tool/upgrade_tool wl 0x280 ./rtthread.bin
```

Run and Debug

System Start

There are several ways to start the system:

1. After firmware upgrade, restart automatically;
2. Plug in power supply and start directly;
3. For devices with battery power supply, press Reset button to start;

System Debug

RK2118 supports serial port debugging. Different hardware devices have different serial port configurations.

The serial communication configuration information is as follows:

Baud rate: 1500000

Data bits: 8

Stop bit: 1

Parity: none

Flow control: none

The following picture indicates entering debugging successfully:

```
Booting TF-M 20240303
                    clk_init: PLL_GPLL = 800000000
clk_init: PLL_VPLL0 = 1179648000
clk_init: PLL_VPLL1 = 903168000
clk_init: PLL_GPLL_DIV = 800000000
clk_init: PLL_VPLL0_DIV = 294912000
clk_init: PLL_VPLL1_DIV = 150528000
clk_init: CLK_DSP0_SRC = 400000000
clk_init: CLK_DSP0 = 700000000
clk_init: CLK_DSP1 = 200000000
clk_init: CLK_DSP2 = 200000000
clk_init: ACLK_NPU = 400000000
clk_init: HCLK_NPU = 133333333
clk_init: CLK_STARSE0 = 400000000
clk_init: CLK_STARSE1 = 400000000
clk_init: ACLK_BUS = 266666666
clk_init: HCLK_BUS = 133333333
clk_init: PCLK_BUS = 133333333
clk_init: ACLK_HSPERI = 133333333
clk_init: ACLK_PERIB = 133333333
clk_init: HCLK_PERIB = 133333333
clk_init: CLK_INT_VOICE0 = 49152000
clk_init: CLK_INT_VOICE1 = 45158400
clk_init: CLK_INT_VOICE2 = 98304000
clk_init: CLK_FRAC_UART0 = 64000000
clk_init: CLK_FRAC_UART1 = 48000000
clk_init: CLK_FRAC_VOICE0 = 24576000
clk_init: CLK_FRAC_VOICE1 = 22579200
clk_init: CLK_FRAC_COMMON0 = 12288000
clk_init: CLK_FRAC_COMMON1 = 11289600
clk_init: CLK_FRAC_COMMON2 = 8192000
clk_init: PCLK_PMU = 100000000
clk_init: CLK_32K_FRAC = 32768
clk_init: CLK_MAC_OUT = 50000000
clk_init: MCLK_PDM = 100000000
clk_init: CLKOUT_PDM = 3072000
clk_init: MCLK_SPDIFTX = 6144000
clk_init: MCLK_OUT_SAI0 = 12288000
clk_init: MCLK_OUT_SAI1 = 12288000
clk_init: MCLK_OUT_SAI2 = 12288000
clk_init: MCLK_OUT_SAI3 = 12288000
clk_init: MCLK_OUT_SAI4 = 12288000
clk_init: MCLK_OUT_SAI5 = 12288000
clk_init: MCLK_OUT_SAI6 = 12288000
clk_init: MCLK_OUT_SAI7 = 12288000
clk_init: SCLK_SAI0 = 12288000
clk_init: SCLK_SAI1 = 12288000
clk_init: SCLK_SAI2 = 12288000
clk_init: SCLK_SAI3 = 12288000
clk_init: SCLK_SAI4 = 12288000
clk_init: SCLK_SAI5 = 12288000
clk_init: SCLK_SAI6 = 12288000
clk_init: SCLK_SAI7 = 122880?

 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Mar 15 2024 17:10:53
 2006 - 2022 Copyright by RT-Thread team
[I/I2C] I2C bus [i2c2] registered
[DSP0 INFO] Hello RK2118 dsp0, build Mar 15 2024 16:58:49
[DSP1 INFO] Hello RK2118 dsp1, build Mar 15 2024 16:59:09
[DSP2 INFO] Hello RK2118 dsp2, build Mar 15 2024 16:59:30
delay 1s test start
msh >delay 1s test end
```

Development Guide

Please refer <Rockchip_Developer_Guide_RK2118_EN.pdf>