

Rockchip RK2118 RT-Thread 快速入门

文档标识: RK-JC-YF-589

发布版本: V1.0.2

日期: 2024-03-15

文件密级: 绝密 秘密 内部资料 公开

免责声明

本文档按“现状”提供，瑞芯微电子股份有限公司（“本公司”，下同）不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因，本文档将可能在未经任何通知的情况下，不定期进行更新或修改。

商标声明

“Rockchip”、“瑞芯微”、“瑞芯”均为本公司的注册商标，归本公司所有。

本文档可能提及的其他所有注册商标或商标，由其各自拥有者所有。

版权所有 © 2024 瑞芯微电子股份有限公司

超越合理使用范畴，非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址: 福建省福州市铜盘路软件园A区18号

网址: www.rock-chips.com

客户服务电话: +86-4007-700-590

客户服务传真: +86-591-83951833

客户服务邮箱: fae@rock-chips.com

前言

概述

本文主要描述了 RK2118 的基本使用方法，旨在帮助开发者快速了解并使用 RK2118 SDK 开发包。

各芯片系统支持状态

芯片名称	内核版本
RK2118	RT-Thread v4.1.x

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

修订记录

日期	版本	作者	修改说明
2024-02-26	V1.0.0	Roger Hu	初始版本
2024-03-15	V1.0.1	Roger Hu	增加运行调试说明
2024-04-17	V1.0.2	Roger Hu	增加RkStudio调音工具说明

目录

Rockchip RK2118 RT-Thread 快速入门

[开发环境搭建](#)

[目录结构](#)

[配置和编译](#)

[CPU固件编译](#)

[自动编译打包](#)

[编译结果清理](#)

[手动编译打包](#)

[模块配置](#)

[编译](#)

[打包](#)

[DSP固件编译](#)

[rk2118 evb板partybox demo](#)

[固件烧录](#)

[注意](#)

[Windows版升级工具](#)

[Linux版烧录工具及命令](#)

[UART烧写](#)

[USB烧写](#)

[RkStudio调音工具](#)

[运行调试](#)

[系统启动](#)

[系统调试](#)

[开发指南](#)

开发环境搭建

本SDK推荐的编译环境是64位的 Ubuntu 20.04 或 Ubuntu18.04， 在其它 Linux 上尚未测试过，所以推荐安装与RK开发者一致的发行版。

编译工具选用的是RT-Thread官方推荐的 SCons + GCC， SCons 是一套由 Python 语言编写的开源构建系统， GCC 交叉编译器由ARM官方提供，可直接使用以下命令安装所需的所有工具：

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

从 ARM 官网下载编译器，通过环境变量指定 toolchain 的路径即可，具体如下：

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

或者使用SDK初始发布包中的编译器: arm-gnutoolchain-13.2.rel1-x86_64-arm-none-eabi.tar.xz, 具体如下:

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

目录结构

以下是SDK主要目录对应的说明:

├── applications	# Rockchip应用demo源码
├── AUTHORS	
├── bsp	# 所有芯片相关代码
│ ├── rockchip	
│ │ ├── common	
│ │ │ ├── drivers	# Rockchip OS适配层通用驱动
│ │ │ └── hal	# Rockchip HAL(硬件抽象层)实现
│ │ └── tests	# Rockchip 驱动测试代码
│ ├── rk2118	# RK2118 主目录
│ │ ├── board	# 板级配置
│ │ ├── build	# 编译主目录, 存放中间文件
│ │ ├── drivers	# RK2118 私有驱动目录
│ │ └── Image	# 存放固件
│ └── tools	# Rockchip 通用工具
├── ChangeLog.md	
├── components	# 系统各个组件, 包括文件系统, shell和框架层等驱动
│ ├── hifi4	
│ │ ├── dsp	# dsp代码
│ │ ├── rtt	# 运行在mcu上dsp相关代码
│ │ ├── shared	# mcu/dsp 公共代码
│ │ └── tools	# dsp固件生成工具
├── documentation	# RT-Thread官方文档
├── examples	# RT-Thread例子程序和测试代码
├── include	# RT-Thread官方头文件目录
├── Kconfig	
├── libcpu	
├── LICENSE	
├── README.md	
├── README_zh.md	
├── RKDocs	# Rockchip 文档
├── src	# RT-Thread内核源码
├── third_party	# Rockchip增加的第三方代码的目录
└── tools	# RT-Thread官方工具目录, 包括menuconfig和编译脚本

配置和编译

CPU固件编译

RT-Thread 用 SCons 来实现编译控制，SCons 是一套由 Python 语言编写的开源构建系统，类似于 GNU Make。它采用不同于通常 Makefile 文件的方式，而使用 SConstruct 和 SConscript 文件来替代。这些文件也是 Python 脚本，能够使用标准的 Python 语法来编写。所以在 SConstruct、SConscript 文件中可以调用 Python 标准库进行各类复杂的处理，而不局限于 Makefile 设定的规则。

自动编译打包

为了简化编译流程，我们做了一个 `build.sh` 脚本来封包编译和打包命令，用法如下：

```
cd bsp/rockchip/rk2118
# board name就是你的板级配置名字，可以在board目录下找到；cpu0, cpu1和dual分别表示要编译
cpu0, cpu1和两个cpu一起
./build.sh <board name> [cpu0|cpu1|dual]
```

以 `adsp_demo` 板子为例，因为这个板子我们只提供 `cpu0` 的配置，所以只能选 `cpu0`，具体如下：

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

这个命令实际会执行如下操作：

1. 找到指定板子的 `cpu0` 默认配置，路径为：`board/adsp_demo/defconfig`，用它覆盖当前目录下的 `menuconfig` 默认配置文件 `.config`
2. 执行 `scons menuconfig` 命令，此时会弹出 `menuconfig` 的配置窗口，你可以按照你的需要修改配置并保存退出，如果想用默认配置，可以选择直接退出
3. 执行 `scons -j$(nproc)` 命令，编译 `cpu0` 的固件
4. 找到指定板子的 `cpu0` 默认打包配置文件，路径为：`board/adsp_demo/setting.ini`，查找这个文件中是否包含了 `root.img` 分区，这是用来放根文件系统的。如果找到，并且分区Flag没有设置 `skip` 标识，则调用 `mkroot.sh` 脚本把 `resource` 目录打包成 `root.img`，否则就直接跳到下一步
5. 用上一步找到的配置文件，来打包出最后烧录的固件 `Firmware.img`

执行完上面的命令，会在 `Image` 目录生成这些文件：

```
-rw-r--r-- 1 rk rk 1638400 Apr 11 02:41 Firmware.img          # 最后烧录的固件
-rw-r--r-- 1 rk rk      16 Apr 11 02:41 Firmware.md5          # 固件的md5校验
-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                 # 根文件系统镜像
-rw-r--r-- 1 rk rk 166916 Apr 11 02:41 rtthread.img
```

编译结果清理

SCons 构建系统默认是通过 MD5 来判断文件是否需要重新编译，如果你的文件内容没变，而只是时间戳变了（例如通过 `touch` 更新时间戳），是不会重新编译这个文件及其依赖的。另外，如果仅修改无关内容，例如代码注释，则只会编译，而不会链接，因为 `obj` 文件内容没变。因此，在开发过程中如果碰到各种修改后实际并未生效的问题，建议在编译前做一次清理，命令如下：

```
scons -c
```

如果做完上面的清理以后，还有异常，可以强制删除所有中间文件，命令如下：

```
rm -rf build
```

其他 SCons 命令，可以看帮助或文档

```
scons -h
```

手动编译打包

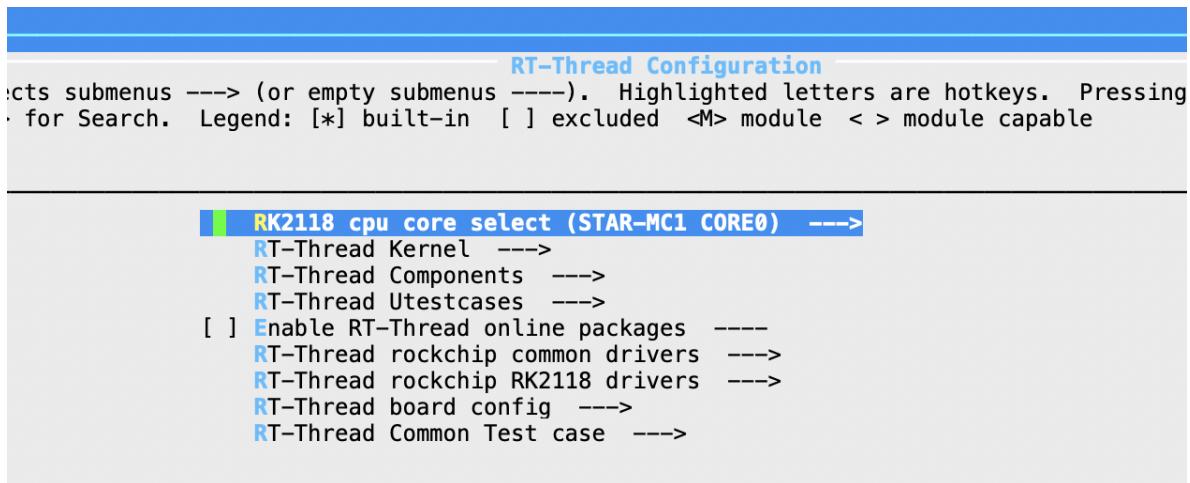
前面通过 `build.sh` 脚本来一键编译的方式虽然方便，但是因为每次都会更新 `rtconfig.h` 这个模块配置文件，所以会整个CPU工程重新编译，速度上稍微会比手动编译慢一些。所以在开发过程中，也会经常用手动编译来加速开发。手动编译步骤可以分为：模块配置、编译、打包三个步骤，只要你的模块配置不需要改动，就可以跳过模块配置，同时如果CPU固件也不需要变动，也可以跳过编译只做打包。

模块配置

模块配置的目的是按产品的实际需要来选择需要的模块，可以通过如下命令来操作：

```
cd bsp/rockchip/rk2118
# 先从你的板级找一个基础配置来覆盖默认的配置文件: .config
cp board/adsp_demo/defconfig .config
# 从.config载入默认配置，并启动图形化配置界面
scons --menuconfig
```

此时会弹出如下界面，你可以根据产品需要来开关各个模块，退出保存配置会覆盖 `.config`，同时自动生成一个 `rtconfig.h` 文件，这2个文件包含了我们选中的各种配置，最终参与编译的只有这个 `rtconfig.h`。



上图中其中第一项是选你要编译的目标CPU，RK2118有两个CPU：cpu0和cpu1。接下来三项是 RT-Thread 公版的配置，再下来是RT-Thread的网络包，剩下都是 BSP 的驱动配置和测试用例。

menuconfig 工具的常见操作如下：

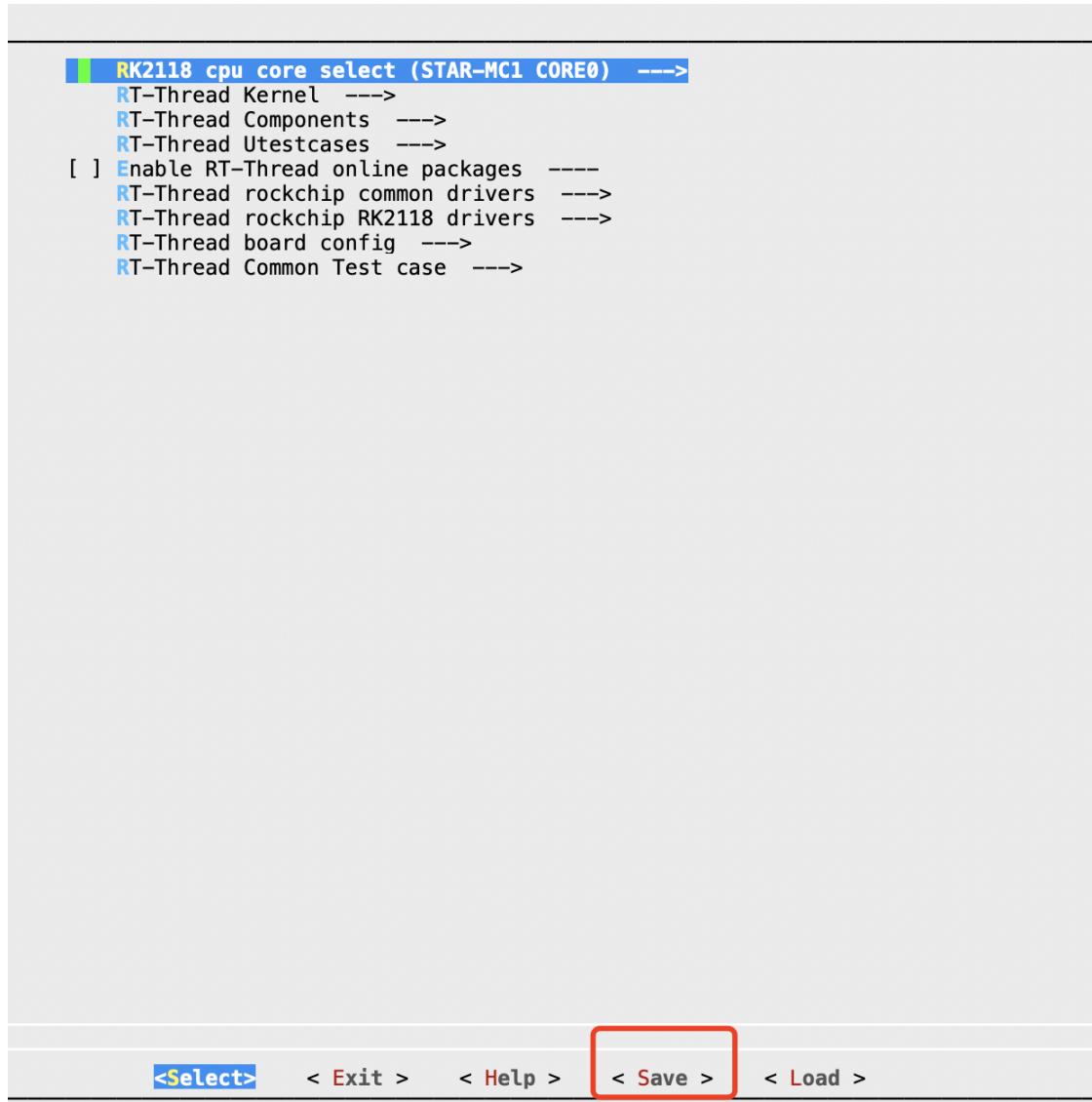
- 上下箭头：移动
- 回车：进入子菜单
- ESC 键：返回上级菜单或退出
- 空格、Y 键或 N 键：使能/禁用 [*] 配置选项
- 英文问号：调出有关高亮选项的帮助菜单（退出帮助菜单，请按回车键）
- / 键：寻找配置项目

每个板级目录下都至少有一个默认的配置文件 `defconfig`，这是cpu0的默认配置，如果这个板级支持双核启动，还会有一个 `cpu1_defconfig`，这是cpu1的默认配置。我们建议提交默认模块配置的时候，都提交到板级目录，并且尽量保持这个命名规则，防止自动编译脚本失效，不要直接提交到 `.config`，这样多个板级配置之间就不会互相覆盖，方法如下：

```
# 假设前面你已经执行过scons --menuconfig，并且退出时有保存配置
cp .config board/xxx/defconfig          # xxx就是你的板级名字，如果是cpu1的配置，请改成
cpu1_defconfig
```

请注意前面提到的参与编译的只有 `rtconfig.h` 而不是 `.config`，所以如果你退出 `menuconfig` 图形界面前没弹出让你保存配置的选项，这是因为你本次执行 `menuconfig` 并没有修改任何选项，此时也就不会重新生成 `rtconfig.h`，而你的 `.config` 可能被你手动覆盖过，已经和 `rtconfig.h` 不匹配，进而导致你当前的 `.config` 并不会在编译的时候生效。有两种方法可以绕过这个问题，可以根据自己需求选一个：

1. 在 `menuconfig` 的图形界面不要直接退出，而是先保存配置再退出，可以通过这个选项来手动保存配置



2. 退出后，执行命令 `scons --useconfig=.config`，这样可以强制重新生成 `rtconfig.h`

编译

编译CPU固件非常简单，执行下面命令即可：

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

如果编译成功，会得到如下文件：

-rw-r--r-- 1 rk rk 197632 Apr 11 07:08 rtt0.bin	# cpu0的bin固件
-rw-r--r-- 1 rk rk 1530528 Apr 11 07:08 rtt0.elf	# cpu0的elf固件，带符号表
-rw-r--r-- 1 rk rk 110592 Apr 11 07:08 rtt1.bin	# cpu1的bin固件
-rw-r--r-- 1 rk rk 784044 Apr 11 07:08 rtt1.elf	# cpu1的elf固件，带符号表

打包

打包是为了把前面编译生成的固件，包括CPU(NPU固件暂时也放CPU这边)和DSP固件，以及loader、TFM等一起整合成一个镜像文件 `Firmware.img`，最后烧录就是用这个镜像。

手动打包需要指定打包配置文件 `setting.ini`，每个板级目录都至少有一个这样的配置文件，如果支持双核启动，一般会命名为 `dual_cpu_setting.ini`，具体命令如下：

```
./mkimage.sh board/xxx/setting.ini      # xxx是你的板级名字
```

DSP固件编译

参考<Rockchip_HiFi4_Quick_Start_CN.pdf>

rk2118 evb板partybox demo

SDK预置了一份编译好的DSP固件：components/hifi4/rtt/dsp_fw/evb_partybox_demo/

实现mp3解码(dsp2)，防啸叫+混音(dsp1)，人声分离(dsp0)功能。

menu(按键长按三秒)：人声分离算法+防啸叫算法 开关，默认开机人声分离关闭，防啸叫算法开。

v- /v+: 背景音音量调节，步进5db，开机默认-15db。

可通过以下操作生成Firmware.img测试：

```
cd components/hifi4/rtt/dsp_fw
cp evb_partybox_demo/* .
cd ../../..
./build.sh evb
```

固件烧录

注意

- 由于打印的uart和烧写的uart是同一个，所以需要确保烧写的时候，要把串口打印的客户端先断开
- uart烧写的时候不能插usb

Windows版升级工具

工具位于：bsp/rockchip/tools/SocToolKit_v1.9_20240130_01_win.zip

烧写前先确保设备已经切到maskrom模式(**按住开发板上的MASKROOM按键，再按RESET按键开机**)，目前Windows也支持uart和usb两种烧写方式，所以第一步先选择烧写方式，选择uart的话要指定串口号和波特率，目前最高支持1.5M波特率烧写。同时支持全固件烧写和分区烧写两种方式，其中全固件烧写方式如下：

step1：选择正确的串口

step2：选择波特率1500000

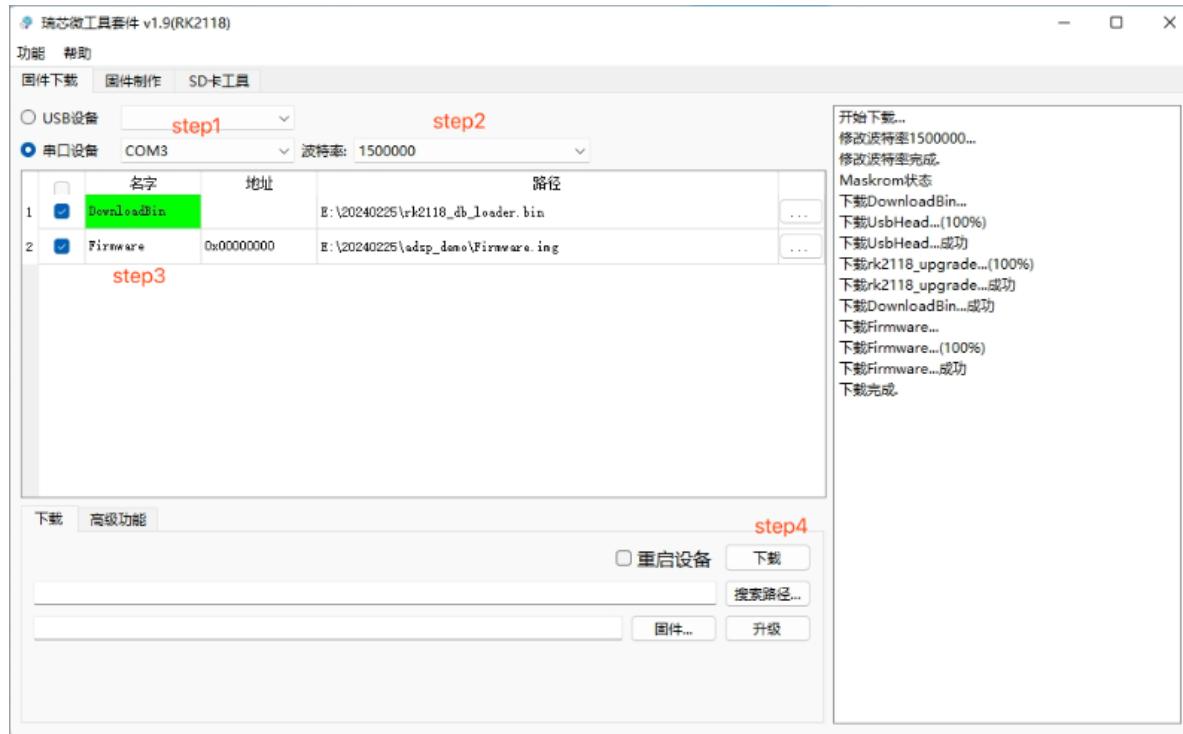
step3：添加下载固件：在空白位置右键选择添加，并填写名字，地址，路径，然后：

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

打勾Firmware.img: bsp/rockchip/rk2118/Image/Firmware.img，地址为0x0

右键选择保存配置到自己的config_xxx.json，以便下次可以直接使用保存的固件配置。

step4：开始下载

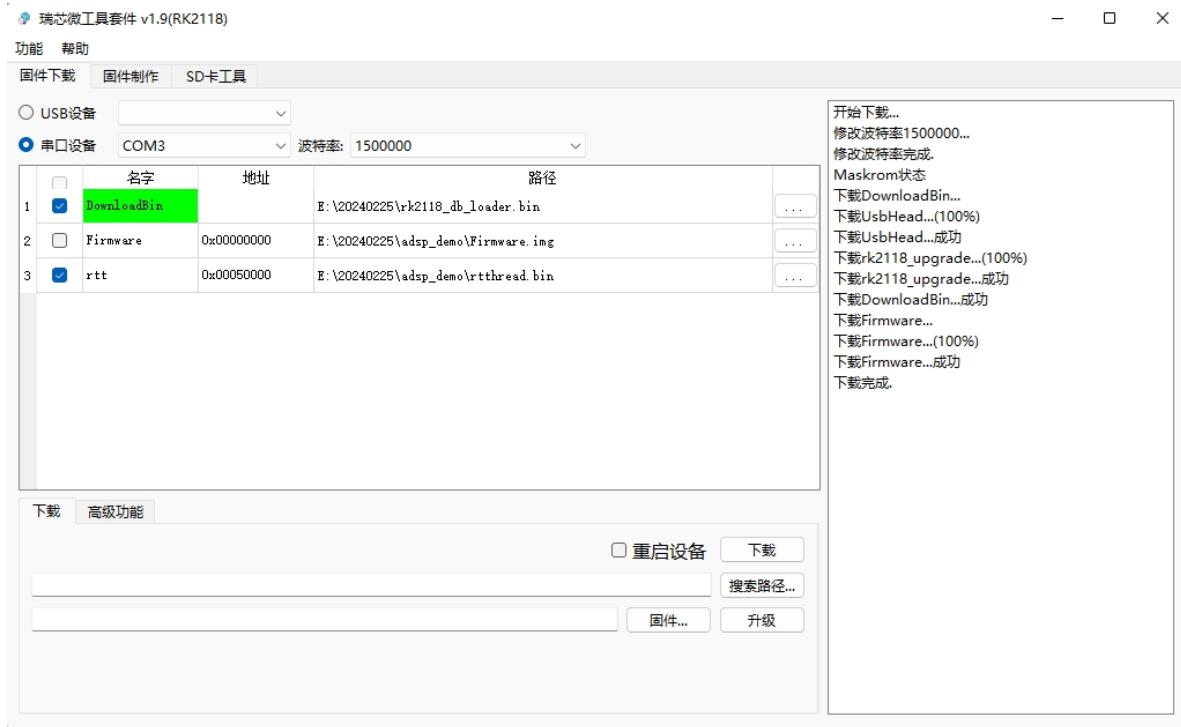


如果想烧写指定分区（例如只修改了rtt0.bin，不需要重新升级Firmware.img以提高效率），则需要指定分区的地址，烧写方式如下：

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

选择要升级的分区，如rttthread: bsp/rockchip/rk2118/rtt0.bin，地址从对应的setting.ini中查找，例如：

```
[UserPart3]
Name=OSA
Type=0x200
PartOffset=0x280 #rtt0.bin所在的位置,
#sector为单位，需要*512转换为字节，如0x280 * 512 = 0x5000
PartSize=0xa00
Flag=
File=../../rtt0.bin
```



Linux版烧录工具及命令

UART烧写

linux下默认ttyUSB设备是没有写权限的，所以烧之前可以通过`sudo chmod 666 /dev/ttyUSBx` (x是你的串口号) 修改权限。

如果不想要每次都去改串口权限，可以通过如下方式添加默认写权限：

```
sudo vi /etc/udev/rules.d/70-rk.rules
# 在创建中添加如下内容，保存退出即可
KERNEL=="ttyUSB[0-9]*", MODE="0666"
```

烧写方式如下：

```
sudo chmod 666 /dev/ttyUSB0
# 先切到maskrom下烧db_loader
./tools/upgrade_tool/upgrade_tool db /dev/ttyUSB0 ./Image/rk2118_db_loader.bin
# 完整固件烧写
./tools/upgrade_tool/upgrade_tool wl /dev/ttyUSB0 0 ./Image/Firmware.img
# 在完整固件烧完之后，开发过程可以只需要烧录有更新的固件，例如下面命令就是单独烧cpu0的rtt固件，第三个参数是固件位置，通过查看setting.ini里UserPart3分区，其PartOffset=0x280，所以这里就填0x280
./tools/upgrade_tool/upgrade_tool wl /dev/ttyUSB0 0x280 ./rtthread.bin
```

USB烧写

烧写方式如下：

```
# 先切到maskrom下烧db loader
./tools/upgrade_tool/upgrade_tool db ./Image/rk2118_db_loader.bin
# 完整固件烧写
./tools/upgrade_tool/upgrade_tool wl 0 ./Image/Firmware.img
# 在完整固件烧完之后, 开发过程可以只烧录自己有更新的固件, 例如下面命令就是单独烧cpu0的rtt固件,
其中位置信息需要保持和setting.ini中该分区的其PartOffset相等即可
./tools/upgrade_tool/upgrade_tool wl 0x280 ./rttthread.bin
```

RkStudio调音工具

可从以下FTP下载最新版本：

ftp://www.rockchip.com.cn

用户名：rkwifi

密码：Cng9280H8t

目录：14-RK2118\RkStudioTool

运行调试

系统启动

系统启动方式有以下几种：

1. 固件升级后, 自动重新启动;
2. 插入电源供电直接启动;
3. 按Reset键启动;

系统调试

RK2118 支持串口调试。不同的硬件设备, 其串口配置也会有所不同。

串口通信配置信息如下：

波特率：1500000

数据位：8

停止位：1

奇偶校验：none

流控：none

成功进入调试的截图：

```
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 = 1228800?
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
```

开发指南

请参考<Rockchip_Developer_Guide_RK2118_CN.pdf>