RK2118 开发向导

文件标识：RK-KF-YF-C00

发布版本：V1.0.0

日期：2024-04-10

文件密级：□绝密 □秘密 □内部资料 ■公开

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

本文提供了RK2118的开发向导，包括基础的开发和调试介绍。

产品版本

芯片名称	内核版本
RK2118	RT-Thread 4.1.1

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

版本号	作者	修改日期	修改说明
V1.0.0	陈谋春	2024-04-10	初始版本

目录

1. 介绍

RK2118内部有两个CPU（star mc1）、3个DSP（hifi4）和1个NPU，总共6个计算模块，这些模块都是独立运行各自的任务，中间通过IPC接口来交互。其中两个CPU都是跑RT-Thread 4.1.1，三个DSP则是跑Bare Metal，NPU则是需要CPU中运行的驱动给它喂数据。

2. 开发环境搭建

2.1 代码工程下载

目前RK2118 SDK 是通过 repo 来管理的，具体下载命令如下：

2.1.1 对外工程下载

1. repo 工具下载，如果已经安装了 repo ，请略过

```
git clone ssh://git@www.rockchip.com.cn/repo/rk/tools/repo -b stable
export PATH=/path/to/repo:$PATH
```

2. 工程下载

```
mkdir rt-thread
cd rt-thread
repo init --repo-url ssh://git@www.rockchip.com.cn/repo/rk/tools/repo -u
ssh://git@www.rockchip.com.cn/rtos/rt-thread/rk/platform/release/manifests -b master -m
rk2118.xml
.repo/repo/repo sync
```

2.2 环境配置

本SDK推荐的编译环境是64位的 Ubuntu16.04 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
```

如无法安装 toolchain ，还可从 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-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
```

3. 目录结构

RK2118 SDK 的标准目录结构如下：

```
|-- AUTHORS
|-- ChangeLog.md
|-- Kconfig
|-- LICENSE
|-- README.md
|-- README_zh.md
|-- applications                    # 这里是一些应用demo源码
|-- bsp                             # 所有的芯片相关代码都在这个目录
|-- components                      # 系统各个组件，包括文件系统，shell和框架层驱动
|-- 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
|-- libcpu                          # 各种CPU和RT-Thread的适配层
|-- src                             # RT-Thread内核源码
|-- third_party                     # 我们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
```

CPU主要代码在 bsp/rockchip 目录下：

```
├── common
│   ├── drivers                     # 驱动适配层
│   ├── fwmgr                       # 固件管理，AB固件和OTA实现
│   ├── hal                         # 硬件抽象层
│   └── tests                       # 测试代码
├── rk2118                          # 2118 bsp主目录
│   ├── applications                # 应用程序主入口
│   ├── board                       # 板级相关代码，每个子目录对应一个板子
│   │   ├── adsp_demo
│   │   ├── amp_demo
│   │   ├── common                  # 所有板级的公共目录
│   │   ├── evb
```

```
│   │   ├── fpga
│   │   ├── iotest
│   │   ├── soundbar_core
│   │   └── test1
│   ├── build                                   # 编译生成文件的中间目录
│   │   ├── applications
│   │   ├── board
│   │   ├── common
│   │   ├── kernel
│   │   └── secure_fw
│   ├── drivers                                 # 芯片专用驱动
│   ├── Image                                   # 打包后的固件目录
│   ├── resource                                # 资源目录，文件系统打包脚本会导成镜像
│   │   └── userdata
│   ├── rkbin                                   # 二进制发布的固件：loader和TFM等
│   └── secure_fw
└── tools                                       # 工具目录：固件打包和烧写相关工具
    ├── cfu
    ├── firmware_merger
    ├── serial
    ├── SerialUpgrade
    │   └── log
    └── upgrade
```

DSP主要代码目录结构如下：

```
├── dsp
│   ├── core
│   │   └── excep
│   ├── drivers
│   │   ├── asrc
│   │   ├── dma
│   │   ├── facc
│   │   ├── sai
│   │   └── spdifrx
│   ├── rkstudio
│   │   ├── basic
│   │   └── library
│   ├── rockit
│   │   ├── inc
│   │   ├── library
│   │   └── stub
│   └── samples
│       └── demo
├── hal -> ../../bsp/rockchip/common/hal/
├── rtt                                         # 在cpu中运行的dsp相关驱动和固件
│   ├── app                                     # dsp相关的应用程序demo
│   │   └── usb_play_demo
│   ├── codecs                                  # codec驱动
│   │   ├── ad2428
│   │   └── tda7708
```

```
│   ├── dsp_drp                                  # dsp驱动，负责dsp的固件加载及复位释放
│   ├── dsp_fw                                   # dsp固件，打包脚本会打包这些固件
│   ├── rk_ss_comm
│   ├── rkstudio_bin
│   ├── rkstudio_parser
│   │   ├── src
│   │   └── test
│   └── rockit
│       ├── inc
│       └── lib
├── samples                                      # dsp的工程demo，通过xtensa的ide来编译调试
│   └── rk2118
│       ├── adsp_demo
│       ├── facc_demo
│       ├── helloworld_demo
│       ├── vocal_separate_demo
│       └── vocal_separate_spdifrx_demo
├── shared                                       # dsp和cpu交互相关的代码，包括锁和ipc
│   └── rk2118
│       ├── core
│       ├── ipc
│       ├── media
│       └── rkstudio
└── tools                                        # dsp相关工具，如：固件打包脚本
```

NPU主要代码目录结构如下：

```
├── examples                                     # demo代码
├── include                                      # npu库头文件
└── lib                                          # npu静态库
```

4. 配置和编译

4.1 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。

4.1.1 自动编译打包

为了简化开发流程，我们做了一个`build.sh`脚本来实现一键配置、编译和打包，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
```

4.1.2 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
```

4.1.3 手动编译打包

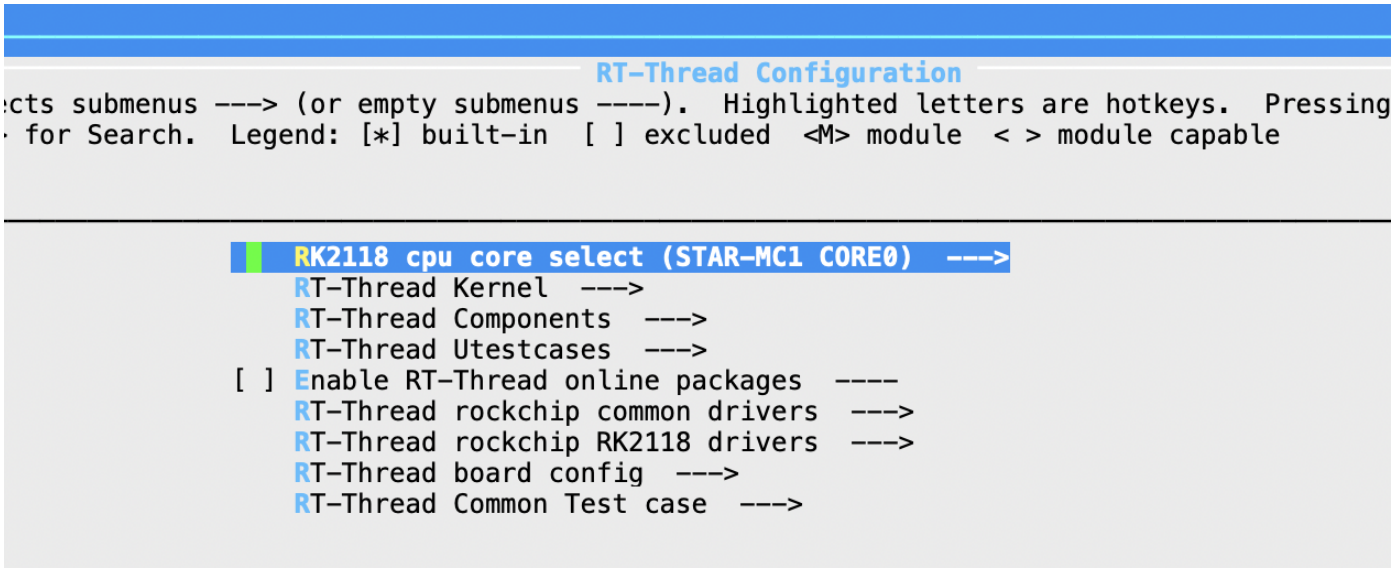
前面通过`build.sh`脚本来一键编译的方式虽然方便，但是因为每次都会更新`rtconfig.h`这个模块配置文件，所以会整个CPU工程重新编译，速度上稍微会比手动编译慢一些。所以在开发过程中，也会经常用手动编译来加速开发。手动编译步骤可以分为：模块配置、编译、打包三个步骤，只要你的模块配置不需要改动，就可以跳过模块配置，同时如果CPU固件也不需要变动，也可以跳过编译只做打包。

4.1.3.1 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，`core0`对应cpu0，`core1`对应cpu1。接下来三项是 RT-Thread 公版的配置，再下来是RT-Thread的网络包，剩下都是我们 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

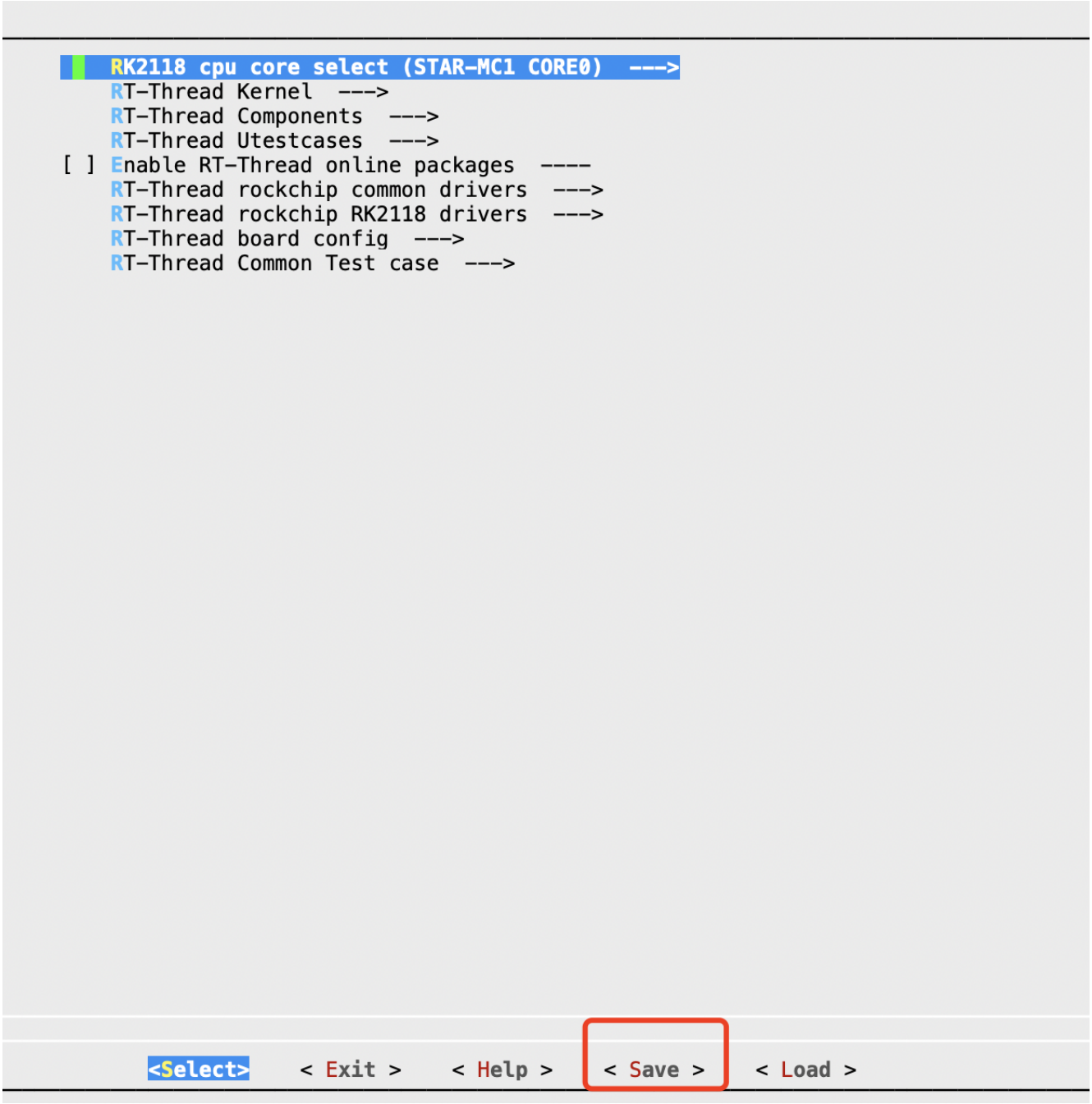
- ESC 键：返回上级菜单或退出
- 空格、Y 键或 N 键：使能/禁用 [*] 配置选项
- 英文问号：调出有关高亮选项的帮助菜单（退出帮助菜单，请按回车键）
- / 键：寻找配置项目

  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sons --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，可以根据自己喜好选一个：

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`

4.1.3.2 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
```

4.1.3.3 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
```

4.1.3.4 固件烧录

RK2118支持两种烧写方式：USB和UART，两种方法都有一些限制：

- USB烧写：只有板子贴的是24M晶振，才能支持USB烧写
- UART烧写：
  1. 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
  2. uart烧写的时候不能插usb

4.1.3.4.1 Linux 下载

Linux下我们支持两种烧写方式：图形工具`SocToolKit`和命令行工具`upgrade_tool`，图形化工具操作的方式和Windows下是一样的，这里就略过了，有需要就参考下一章的`Windows 下载`。

1. USB烧写

   烧写方式如下：

```
# 先切到maskrom下烧db loader
../tools/upgrade/upgrade_tool db ./Image/rk2118_db_loader.bin
# 完整固件烧写
../tools/upgrade/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，第三个参数是固件位置，算法从上面的setting.ini里UserPart4分区，其PartOffset=0x300，所以这里就填0x300
../tools/upgrade/upgrade_tool wl 0x300 ./rtt0.bin
```

2. UART烧写

   `upgrade_tool`默认是走USB烧写，所以要切到UART方式，还需要改一个配置文件`bsp/rockchip/tools/upgrade/config.ini`，其中最后两行的配置打开：

```
#lba_parity=
#rb_check_off=
#nor_single_idb=
#log_off=
#stdout_buffer_off=
#force_data_band=
#log_dir=
#usb3_transfer_on=true
use_serial_transfer_on=true
use_serial_baudrate=1500000
```

需要注意：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：

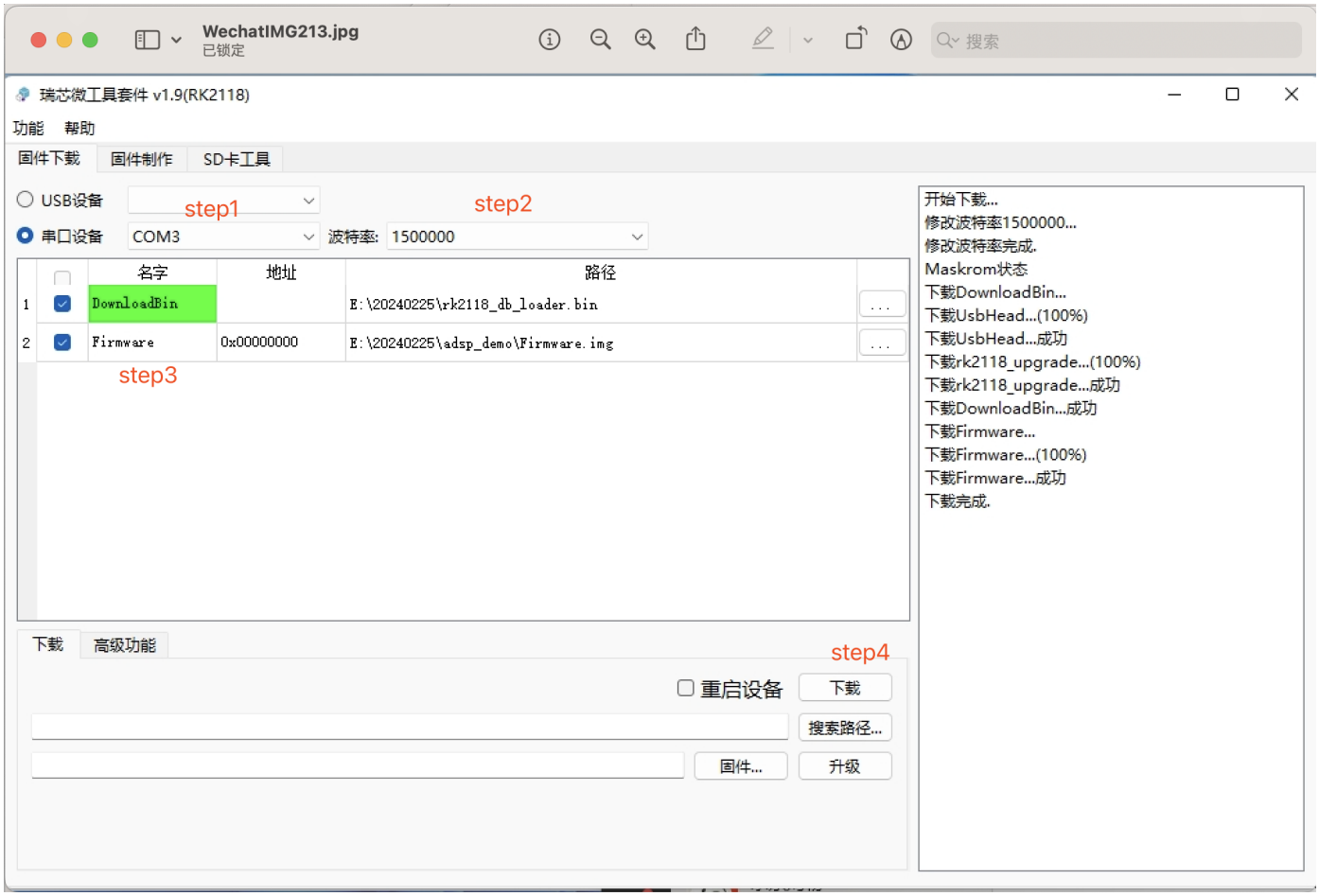
```
sudo vi /etc/udev/rules.d/70-rk.rules
# 在创建中添加如下内容，保持退出即可
KERNEL=="ttyUSB[0-9]*", MODE="0666"
```

UART烧写命令与USB完全一样，只是要多一个参数来指定串口，具体命令如下：

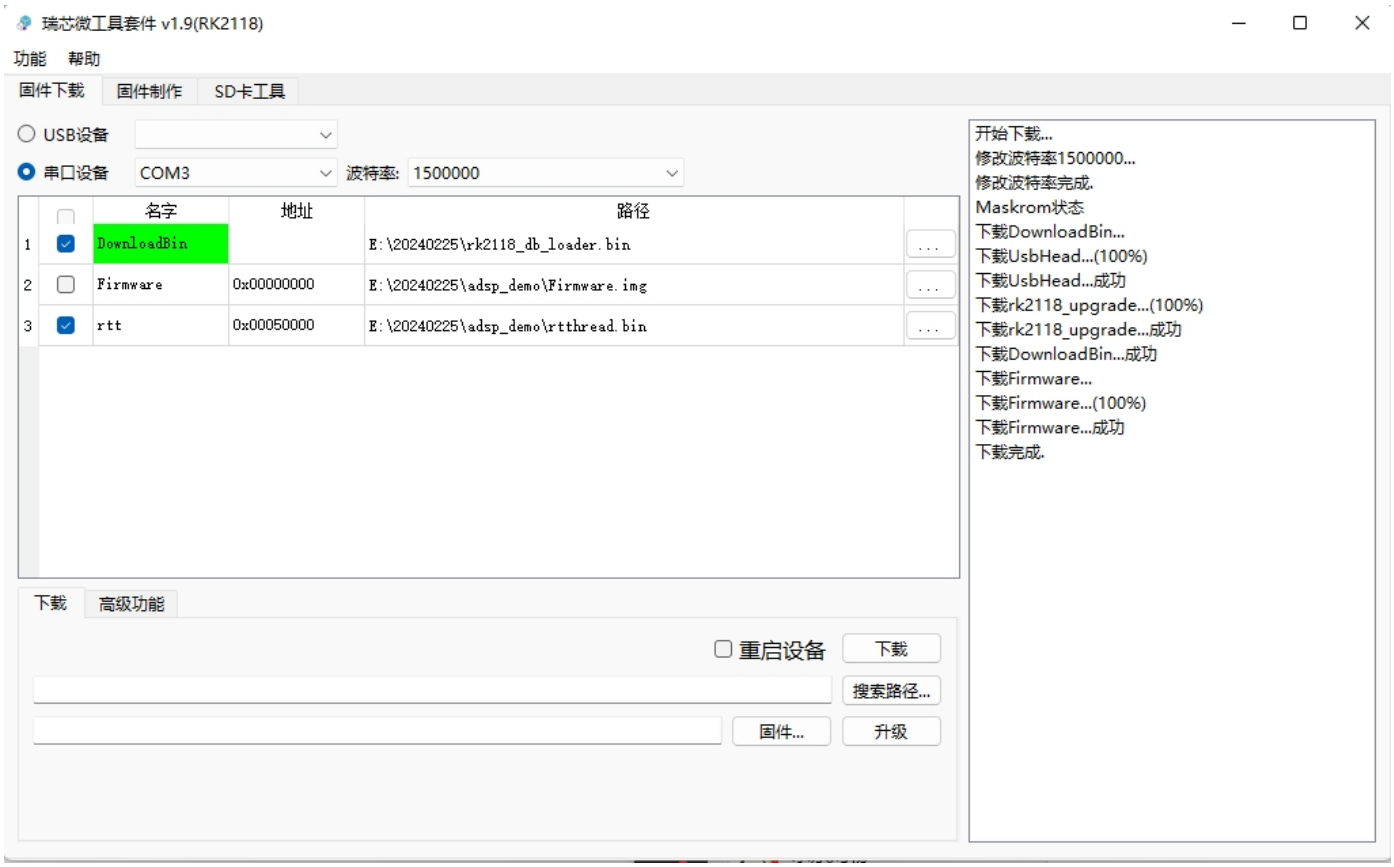
```
sudo chmod 666 /dev/ttyUSB0
# 先切到maskrom下烧db loader
../tools/upgrade/upgrade_tool db /dev/ttyUSB0 ./Image/rk2118_db_loader.bin
# 完整固件烧写
../tools/upgrade/upgrade_tool wl /dev/ttyUSB0 0 ./Image/Firmware.img
# 在完整固件烧完之后，开发过程可以只烧录自己有更新的固件，例如下面命令就是单独烧cpu0的rtt固件，第三个参数是固件位置，算法从上面的setting.ini里UserPart4分区，其PartOffset=0x300，所以这里就填0x300
../tools/upgrade/upgrade_tool wl /dev/ttyUSB0 0x300 ./rtt0.bin
```

4.1.3.4.2 windows 下载

烧写前先确保设备已经切到maskrom模式，第一步先选择烧写方式：USB or UART，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：



如果想烧写指定分区，则需要指定分区的地址，烧写方式如下：



4.2 双核启动

如果需要同时跑CPU0和CPU1，需要注意资源竞争问题，我们推荐的方式是，CPU0作为主核，负责所有驱动和任务的管理，而CPU1则作为专用计算核，做一些纯计算任务，同时只会访问少量CPU0不会用到的外设。我们的`board/iotest/cpu1_defconfig`是一个CPU1的推荐配置，这个配置关闭了大部分外设，唯一和CPU0可能共享的打印串口则在打印的时候有互斥保护，可以放心使用。

4.3 DSP固件编译

待补充

4.4 NPU固件编译

待补充

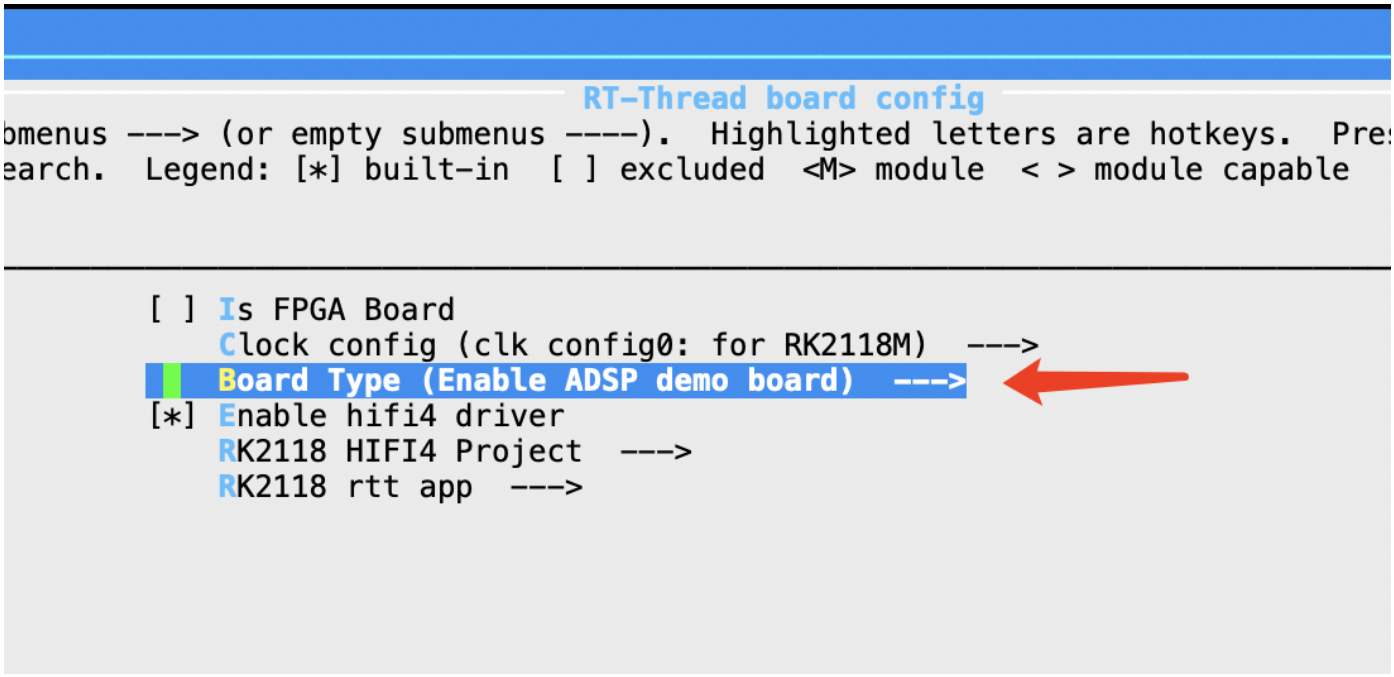
5. 开发

5.1 板级配置

所有的板级配置都放在芯片 BSP 主目录下的 board 目录下，例如 RK2118 的板级配置的目录结构如下：

```
├── adsp_demo                # adsp_demo板级配置目录
│   ├── board.c              # 板级配置主入口，开机启动会自动调用这个文件的`rt_hw_board_init`函数
│   ├── board.h
│   ├── defconfig            # cpu0的默认模块配置文件，用于`menuconfig`命令
│   ├── iomux.c              # pin脚复用配置，启动过程会自动调用这个文件的`rt_hw_iomux_config`函数
│   ├── iomux.h
│   ├── SConscript
│   └── setting.ini          # 固件分区配置脚本，用于固件打包成镜像
├── common                   # 所有板级的底层公共目录，大部分函数都是WEAK的，允许上级板子覆盖这些函数
│   ├── board_base.c         # 公共板级主入口
│   ├── board_base.h
│   ├── clk_base.c           # 公共时钟配置
│   ├── iomux_base.c         # 公共pin脚复用配置
│   ├── iomux_base.h
│   ├── mnt_base.c           # 公共文件系统自动挂载表
│   ├── SConscript
│   └── setting.ini          # 固件分区配置模版
├── Kconfig                  # menuconfig配置文件
├── SConscript               # scons编译脚本
```

可以在`menuconfig`的选项来选择你要编译的板级配置，界面如下：



如果要添加新的板级配置，可以先拷贝一个相近的板子作为基点，具体如下：

```
cd bsp/rockchip/rk2118/board
cp -r adsp_demo new_board
# 修改这个板级配置的编译开关，把RT_USING_BOARD_ADSP_DEMO改成RT_USING_BOARD_NEW_BOARD
vi board/adsp_demo/SConscript
# 增加板级配置选项，在"Board Type" choice下增加RT_USING_BOARD_NEW_BOARD
vi board/Kconfig
cd new_board
# 修改CONFIG_RT_USING_BOARD_ADSP_DEMO=y为CONFIG_RT_USING_BOARD_NEW_BOARD=y
vi defconfig
# 根据实际板子情况修改其他板级配置，后续章节会介绍一些常见的配置修改方法
```

5.1.1 频率配置

RK2118有三组频率配置，分别对应了三种产品形态，基础频率配置位于`board/common/clk_base.c`，对应关系如下：

- RT_USING_CLK_CONFIG0: RK2118M，用于车载音频产品，不带DDR
- RT_USING_CLK_CONFIG1: RK2118BM，用于车载音频产品，带DDR
- RT_USING_CLK_CONFIG2: RK2118G 和 RK2118B，对应消费类产品

  正常情况下你只需要根据你的产品形态，选择对应的频率配置就好，如果你想调整频率，则可以从基础配置文件`board/common/clk_base.c`中把对应的频率拷贝出来，放到你对应的板级配置文件主入口文件`board/xxx/board.c`中，具体如下：

```
/* 请删掉下面这行代码，替换成拷贝的频率表 */
//extern const struct clk_init clk_inits[];
/* 你可以在下面拷贝过来的频率表基础上，修改你要变动的频率 */
RT_WEAK const struct clk_init clk_inits[] =
{
```

```
    INIT_CLK("PLL_GPLL", PLL_GPLL, 800000000),
    INIT_CLK("PLL_VPLL0", PLL_VPLL0, 983040000),
    INIT_CLK("PLL_VPLL1", PLL_VPLL1, 903168000),
    INIT_CLK("PLL_GPLL_DIV", PLL_GPLL_DIV, 200000000),
    INIT_CLK("PLL_VPLL0_DIV", PLL_VPLL0_DIV, 122880000),
    INIT_CLK("PLL_VPLL1_DIV", PLL_VPLL1_DIV, 112896000),
    INIT_CLK("CLK_DSP0_SRC", CLK_DSP0_SRC, 400000000),
    INIT_CLK("CLK_DSP0", CLK_DSP0, 700000000),
    INIT_CLK("CLK_DSP1", CLK_DSP1, 491520000),
    INIT_CLK("CLK_DSP2", CLK_DSP2, 491520000),
    INIT_CLK("ACLK_NPU", ACLK_NPU, 400000000),
    INIT_CLK("HCLK_NPU", HCLK_NPU, 150000000),
    INIT_CLK("CLK_STARSE0", CLK_STARSE0, 400000000),
    INIT_CLK("CLK_STARSE1", CLK_STARSE1, 400000000),
    INIT_CLK("ACLK_BUS", ACLK_BUS, 300000000),
    INIT_CLK("HCLK_BUS", HCLK_BUS, 150000000),
    INIT_CLK("PCLK_BUS", PCLK_BUS, 150000000),
    INIT_CLK("ACLK_HSPERI", ACLK_HSPERI, 150000000),
    INIT_CLK("ACLK_PERIB", ACLK_PERIB, 150000000),
    INIT_CLK("HCLK_PERIB", HCLK_PERIB, 150000000),
    INIT_CLK("CLK_INT_VOICE0", CLK_INT_VOICE0, 49152000),
    INIT_CLK("CLK_INT_VOICE1", CLK_INT_VOICE1, 45158400),
    INIT_CLK("CLK_INT_VOICE2", CLK_INT_VOICE2, 98304000),
    INIT_CLK("CLK_FRAC_UART0", CLK_FRAC_UART0, 64000000),
    INIT_CLK("CLK_FRAC_UART1", CLK_FRAC_UART1, 48000000),
    INIT_CLK("CLK_FRAC_VOICE0", CLK_FRAC_VOICE0, 24576000),
    INIT_CLK("CLK_FRAC_VOICE1", CLK_FRAC_VOICE1, 22579200),
    INIT_CLK("CLK_FRAC_COMMON0", CLK_FRAC_COMMON0, 12288000),
    INIT_CLK("CLK_FRAC_COMMON1", CLK_FRAC_COMMON1, 11289600),
    INIT_CLK("CLK_FRAC_COMMON2", CLK_FRAC_COMMON2, 8192000),
    INIT_CLK("PCLK_PMU", PCLK_PMU, 100000000),
    INIT_CLK("CLK_32K_FRAC", CLK_32K_FRAC, 32768),
    INIT_CLK("CLK_MAC_OUT", CLK_MAC_OUT, 50000000),
    /* Audio */
    INIT_CLK("MCLK_PDM", MCLK_PDM, 100000000),
    INIT_CLK("CLKOUT_PDM", CLKOUT_PDM, 3072000),
    INIT_CLK("MCLK_SPDIFTX", MCLK_SPDIFTX, 6144000),
    INIT_CLK("MCLK_OUT_SAI0", MCLK_OUT_SAI0, 12288000),
    INIT_CLK("MCLK_OUT_SAI1", MCLK_OUT_SAI1, 12288000),
    INIT_CLK("MCLK_OUT_SAI2", MCLK_OUT_SAI2, 12288000),
    INIT_CLK("MCLK_OUT_SAI3", MCLK_OUT_SAI3, 12288000),
    INIT_CLK("MCLK_OUT_SAI4", MCLK_OUT_SAI4, 12288000),
    INIT_CLK("MCLK_OUT_SAI5", MCLK_OUT_SAI5, 12288000),
    INIT_CLK("MCLK_OUT_SAI6", MCLK_OUT_SAI6, 12288000),
    INIT_CLK("MCLK_OUT_SAI7", MCLK_OUT_SAI7, 12288000),
    INIT_CLK("CLK_TSADC", CLK_TSADC, 1200000),
    INIT_CLK("CLK_TSADC_TSEN", CLK_TSADC_TSEN, 12000000),
    INIT_CLK("SCLK_SAI0", SCLK_SAI0, 12288000),
    INIT_CLK("SCLK_SAI1", SCLK_SAI1, 12288000),
    INIT_CLK("SCLK_SAI2", SCLK_SAI2, 12288000),
    INIT_CLK("SCLK_SAI3", SCLK_SAI3, 12288000),
    INIT_CLK("SCLK_SAI4", SCLK_SAI4, 12288000),
```

```
    INIT_CLK("SCLK_SAI5", SCLK_SAI5, 12288000),
    INIT_CLK("SCLK_SAI6", SCLK_SAI6, 12288000),
    INIT_CLK("SCLK_SAI7", SCLK_SAI7, 12288000),
    { /* sentinel */ },
};
```

RK2118还支持两档频率电压表，在休眠的降频降压来节省功耗，下面是一个用PWM来实现`voltage regulator`功能的例子，详细代码可以参考`board/evb/board.c`:

```
RT_WEAK struct pwr_pwm_info_desc pwm_pwr_desc[] =
{
    {
        .name = "pwm0",
        .chanel = 3,
        .invert = true,
    },
    {
        .name = "pwm0",
        .chanel = 2,
        .invert = true,
    },
    { /* sentinel */ },
};

RT_WEAK struct regulator_desc regulators[] =
{
    {
        .flag = REGULATOR_FLG_PWM | REGULATOR_FLG_LOCK,
        .desc.pwm_desc = {
            .flag = DESC_FLAG_LINEAR(PWR_CTRL_VOLT_RUN | PWR_CTRL_PWR_EN),
            .info = {
                .pwrId = PWR_ID_CORE,
            },
            .pwrId = PWR_ID_CORE,
            .period = 25000,
            .minVolt = 810000,
            .maxVlot = 1000000,
            .voltage = 900000,
            .pwm = &pwm_pwr_desc[0],
        },
    },
    {
        .flag = REGULATOR_FLG_PWM | REGULATOR_FLG_LOCK,
        .desc.pwm_desc = {
            .flag = DESC_FLAG_LINEAR(PWR_CTRL_VOLT_RUN | PWR_CTRL_PWR_EN),
            .info = {
                .pwrId = PWR_ID_DSP_CORE,
            },
            .pwrId = PWR_ID_DSP_CORE,
            .period = 25000,
            .minVolt = 800000,
```

```
            .maxVlot = 1100000,
            .voltage = 900000,
            .pwm = &pwm_pwr_desc[1],
        },
    },
    { /* sentinel */ },
};
RT_WEAK const struct regulator_init regulator_inits[] =
{
    /* name, ID, 运行电压，运行电压启用， 休眠电压，休眠电压启用 */
    REGULATOR_INIT("vdd_core",  PWR_ID_CORE,       900000, 1, 900000, 1),
    REGULATOR_INIT("vdd_dsp",   PWR_ID_DSP_CORE,   1000000, 1, 900000, 1),
    /* 所以上面的配置就是：vdd_core 运行和休眠电压都是0.9v，这一路供电目前是给两个cpu和dsp1和dsp2；
     * vdd_dsp 运行电压是1v，休眠电压是0.9v，这一路供电目前是给dsp0 */
    { /* sentinel */ },
};
```

5.1.2 IOMUX

芯片的管脚有限，大部分功能都存在 IO 复用的情况，所以每个功能模块都要根据实际硬件板图配置 IOMUX ，RK2118有两种IO脚，一种是普通IO，它只能在有限几个功能之间做复用；另一种叫矩阵IO，简称RMIO，它分成5组矩阵，每一组内的IO都可以在这组矩阵的功能列表里任意复用。下面就是两种IO的复用配置代码：

```
/* 这是一个4线spi的IO配置，它有4个数据线，一个clk和一个cs，用的是普通IO，对应的IO脚是：
 * gpio1_a4, gpio1_a5, gpio1_a6, gpio1_a7, gpio1_b0, gpio1_b1 */
RT_WEAK  void fspi0_iomux_config(void)
{
    HAL_PINCTRL_SetIOMUX(GPIO_BANK1,
                         GPIO_PIN_A4 |  // FSPI_D3
                         GPIO_PIN_A5 |  // FSPI_CLK
                         GPIO_PIN_A6 |  // FSPI_D0
                         GPIO_PIN_A7 |  // FSPI_D2
                         GPIO_PIN_B0 |  // FSPI_D1
                         GPIO_PIN_B1,   // FSPI_CSN
                         PIN_CONFIG_MUX_FUNC2);
     /* 通过查TRM手册我们可以找到这几个IO的spi功能对应的功能号是2，所以这里选PIN_CONFIG_MUX_FUNC2
*/
}
/* 这是一个uart2的IO配置，它有两个脚：tx和rx，用的是RMIO，所以每一个脚需要单独配置功能 */
RT_WEAK  void uart2_iomux_config(void)
{
    /* 通过查找`bsp/rockchip/common/hal/lib/CMSIS/Device/RK2118/Include/soc.h`中的
     * RMIO_Name定义，我们可以找到gpio3_b3和gpio3_b4对应的RMIO组，以及这个组内对应的
     * uart2_tx和uart2_rx的功能名分别是：RMIO_UART2_TX_RM1 和 RMIO_UART2_RX_RM1 */
    HAL_PINCTRL_SetRMIO(GPIO_BANK3,
                        GPIO_PIN_B3,   // UART2_TX_AUDIO_DEBUG
                        RMIO_UART2_TX_RM1);

    HAL_PINCTRL_SetRMIO(GPIO_BANK3,
```

```
                        GPIO_PIN_B4,    // UART2_RX_AUDIO_DEBUG
                        RMIO_UART2_RX_RM1);
}
```

在添加自己的IOMUX之前，可以先检查公共的IOMUX配置文件`iomux_base.c`是否有现成的功能配置，如果pin脚功能选择和自己硬件板子能匹配，则可以直接在自己的IOMUX配置文件`board/xxx/iomux.c`的`rt_hw_iomux_config`函数中调用。只有没有现成可用的公共IOMUX函数的情况下，才需要在`board/xxx/iomux.c`增加新函数。

5.1.3 外设相关板级配置

板级配置中，还有一部分是和具体外设相关的，一般放在`board.c`和`board.h`，如果外设的板级代码比较复杂，比如涉及到复杂的IO控制逻辑，可以自己独立一个文件，然后在`board.h`里去包含独立的头文件声明。下面是一些比较简单的外设板级配置的例子：

```
/* 大部分的外设板级配置都能直接从结构成员名了解其含义，有不清楚可以查看具体外设的开发文档 */
#if defined(RT_USING_TSADC)
RT_WEAK const struct tsadc_init g_tsadc_init =
{
    .chn_id = {0},
    .chn_num = 1,
    .polarity = TSHUT_LOW_ACTIVE,
    .mode = TSHUT_MODE_CRU,
};
#endif /* RT_USING_TSADC */

#if defined(RT_USING_UART0)
RT_WEAK const struct uart_board g_uart0_board =
{
    .baud_rate = UART_BR_1500000,
    .dev_flag = ROCKCHIP_UART_SUPPORT_FLAG_DEFAULT,
    .bufer_size = RT_SERIAL_RB_BUFSZ,
    .name = "uart0",
};
#endif /* RT_USING_UART0 */

#if defined(RT_USING_UART1)
RT_WEAK const struct uart_board g_uart1_board =
{
    .baud_rate = UART_BR_1500000,
    .dev_flag = ROCKCHIP_UART_SUPPORT_FLAG_DEFAULT,
    .bufer_size = RT_SERIAL_RB_BUFSZ,
    .name = "uart1",
};
#endif /* RT_USING_UART1 */
```

5.1.4 文件系统自动挂载表

目前RK2118 SDK支持Littlefs和FAT/EXFAT文件系统，如果产品需要文件系统，可以在`menuconfig`里把相应的文件系统打开，并在板级目录下新建一个文件`mnt.c`，用来配置文件系统自动挂载表，下面是一个挂载表的配置示例：

```
#ifdef RT_USING_DFS_MNTTABLE
#include <dfs_fs.h>

#define PARTITION_ROOT          "root"

/********************* Private Variable Definition ***************************/
/** @defgroup MNT_Private_Variable Private Variable
 *  @{
 */

/**
 * @brief  Config mount table of filesystem
 * @attention The mount_table must be terminated with NULL, and the partition's name
 * must be the same as above.
 */
const struct dfs_mount_tbl mount_table[] =
{
    /* 设备节点名，挂载点目录，文件系统名字（elm对应fat），rw(1为读写，0为只读，elm不支持这个标识，可以随便填），文件系统私有数据（elm不支持，可以随便填） */
    {PARTITION_ROOT, "/", "elm", 0, 0},
#ifdef RT_SDCARD_MOUNT_POINT
    {"sd0", RT_SDCARD_MOUNT_POINT, "elm", 0, 0},
#endif
    {0}
};
#endif
```

5.1.5 固件分区配置

RK2118支持多种存储介质：SPI Nor、SPI Nand和EMMC，系统的固件和数据都是存放在这里，所以这个存储是所有master共享的，包括：两个CPU、3个DSP和1个NPU，为了避免资源冲突，就需要对存储介质划分分区，我们是在`board/xxx/setting.ini`来配置的，如果是双核启动的固件，则是`board/xxx/dual_cpu_setting.ini`。下面是一个配置例子：

```
# 下面的注释是解释每个配置属性的含义，比较重要的是PartSize和PartOffset单位都是512Bytes，然后分区大小需要按64KBytes对齐
#Flag:
#bits filed:
#  [0]    : skip                  : 0 - disabled(default), 1 - enable
#  [2]    : no partition size     : 0 - diabled(default), 1 - enable
#  [8, 9] : property              : 0 - do not register(default), 1 - read only, 2 - write only, 3 - rw
#  [10]   : register type         : 0 - block partition(default), 1 - MTD partition
```

```
#type can suppot 32 partiton types,0x0:undefined 0x1:Vendor 0x2:IDBlock ,bit3:bit7
reserved by loader, 0x100:TFM, 0x200:RTT, 0x400:DSP Firmware, 0x800:Root FS, bit24:bit30
reserved for user.
#PartSize and PartOffset unit by sector
#Gpt_Enable 1:compact gpt,0:normal gpt
#Backup_Partition_Enable 0:no backup,1:backup
#Loader_Encrypt 0:no encrypt,1:rc4
#nano 1:generate idblock in nano format
[System]
FwVersion=1.0
Gpt_Enable=
Backup_Partition_Enable=
Nano=
Loader_Encrypt=
IDB_Boot_Encrypt=
Chip=
Model=
BLANK_GAP=1
[UserPart1]
Name=NIDB
Type=0x2
PartOffset=0x80
PartSize=0x100
Flag=
File=../../rkbin/rk2118_loader.bin,../../rkbin/rk2118_ddr.bin
[UserPart2]
Name=cpu0s
Type=0x100
PartOffset=0x180
PartSize=0x100
Flag=
File=../../rkbin/tfm_s.bin
[UserPart3]
Name=cpu1s
Type=0x100
PartOffset=0x280
PartSize=0x80
Flag=
File=../../rkbin/cpu1_loader.bin
# 前面都是系统分区，不允许修改，后面开始则是放cpu和dsp的固件，可以根据需求来调整
[UserPart4]
Name=cpu0
Type=0x200
PartOffset=0x300
PartSize=0x200
Flag=
File=../../rtt0.bin
[UserPart5]
Name=dsp0
Type=0x400
PartOffset=0x500
PartSize=0x800
```

```
Flag=
File=../../../../../components/hifi4/rtt/dsp_fw/dsp0.bin
[UserPart6]
Name=dsp1
Type=0x400
PartOffset=0xd00
PartSize=0x800
Flag=
File=../../../../../components/hifi4/rtt/dsp_fw/dsp1.bin
[UserPart7]
Name=dsp2
Type=0x400
PartOffset=0x1500
PartSize=0x800
Flag=
File=../../../../../components/hifi4/rtt/dsp_fw/dsp2.bin
# 下面是放音频工具的参数
[UserPart8]
Name=rkstudio
Type=0x800
PartOffset=0x1d00
PartSize=0x2300
Flag=
File=../../../../../components/hifi4/rtt/rkstudio_bin/rkstudio.bin
```

需要注意的是如果固件启用了XIP(eXecute In Place)，此时CPU固件则是位置相关的，即分区配置里的cpu0固件位置，必须和`mem_layout.h`里的`XIP_CPU0_RTT_BASE`能匹配上。计算方法如下：

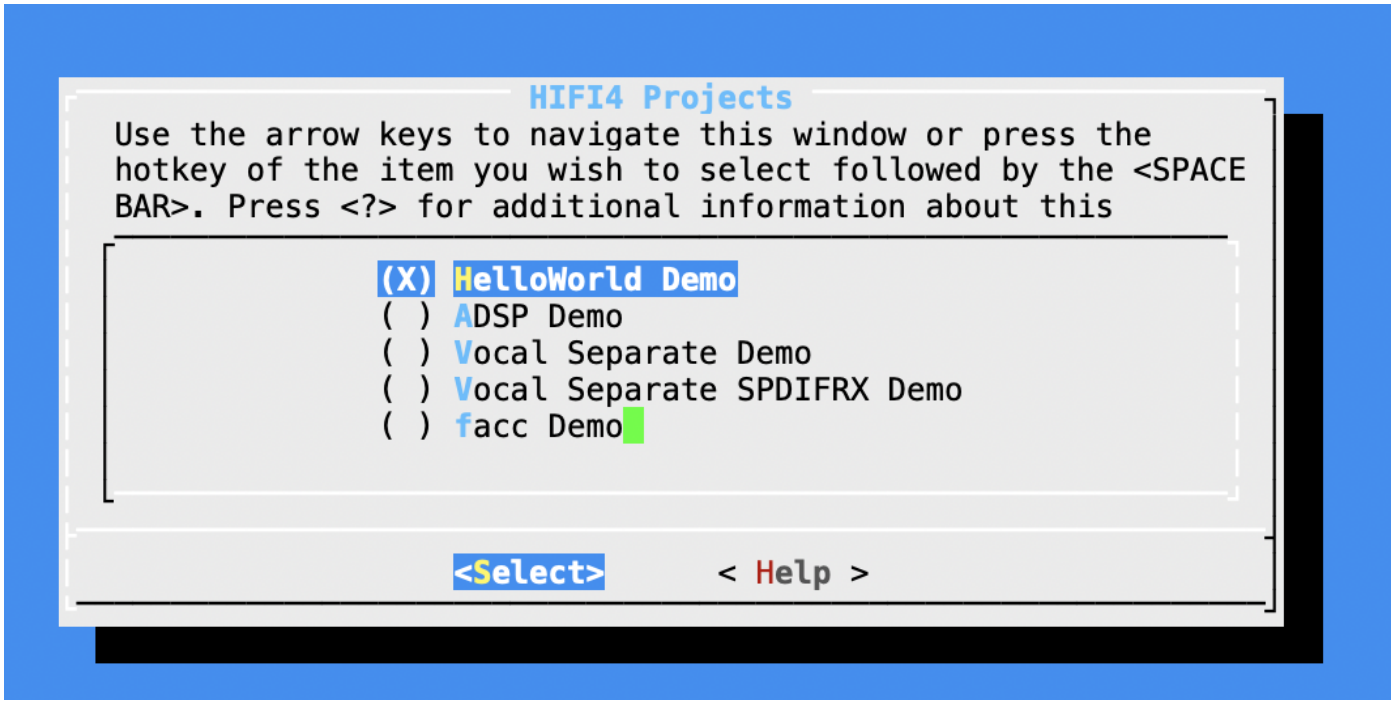
```
# 从mem_layout.h找到XIP_BASE=0x11000000;，从setting.ini里找到PartOffset=0x300
# 所以，setting.ini里可以算出cpu0的分区位置所对应的xip地址为：
cpu0_xip_base_from_setting_ini = 0x11000000 + 0x300*512 = 0x11060000
# 从mem_layout.h找到XIP_CPU0_RTT_BASE=(XIP_CPU1_LOADER_BASE + XIP_CPU1_LOADER_SIZE);
# 计算得到0x11060000，所以是匹配的
```

在`mkimage.sh`脚本里会自动完成XIP地址的匹配检查，如果不匹配，会出现如下警告：

```
found File=../../rtt0.bin PartOffset: 0x11060000
rtt0.elf part_offset=0x11060000, xip_address=0x11160000
# 上面的log意思是setting.ini检查到的xip地址是0x11060000，而cpu固件实际的xip地址是0x11160000，所以不匹配
```

5.1.6 Memory layout

同上一章，为了分配系统的各种memory资源，包括：nor/nand/emmc、sram和DDR等，我们要求每个项目都要有一个memory的分配文件`mem_layout.h`，因为RK2118的项目，一般都是DSP来主导，所以这个文件一般放在DSP的工程目录，CPU这边通过menuconfig来选择当前要用的DSP工程，就会自动关联到到这个文件，界面如下：



下面是`mem_layout.h`的常规格式，允许用户修改的只有注释了`User-modifiable`的区间，也就是每个分区的大小和DDR大小，具体如下：

```
/* ------------------------------------------------------------------------- */
/* Notes:                                                                    */
/* 1. Users are encouraged to modify the "size" definitions as needed,       */
/*    the base addresses will automatically adjust to these changes.         */
/* 2. Always ensure that total allocated sizes do not exceed the physical    */
/*    memory size available on the device.                                   */
/* ------------------------------------------------------------------------- */

/* ------------------------------------------------------------------------- */
/* Physical Memory Sizes                                                     */
/* ------------------------------------------------------------------------- */
XIP_SIZE        =         0x00400000; /* 4 MB - Total size of XIP memory */
SRAM_SIZE       =         0x00100000; /* 1 MB - Total size of SRAM */
/* Sizes - User-modifiable */
DRAM_SIZE       =         0x04000000; /* 64 MB - Total size of DRAM */

/* ------------------------------------------------------------------------- */
/* XIP Memory Layout                                                         */
/* ------------------------------------------------------------------------- */
XIP_BASE        =         0x11100000;

/* Sizes - User-modifiable, 这里用户可以根据实际情况来调整的分区大小 */
XIP_RKPARTITIONTABLE_SIZE  =  0x00010000; /* 64 KB */
XIP_IDBLOCK_SIZE           =  0x00020000; /* 128 KB */
XIP_CPU0_TFM_SIZE          =  0x00020000; /* 128 KB */
XIP_CPU1_LOADER_SIZE       =  0x00010000; /* 64 KB */
XIP_CPU0_RTT_SIZE          =  0x00040000; /* 256 KB */
XIP_DSP0_FIRMWARE_SIZE     =  0x00100000; /* 1 MB */
XIP_DSP1_FIRMWARE_SIZE     =  0x00100000; /* 1 MB */
XIP_DSP2_FIRMWARE_SIZE     =  0x00100000; /* 1 MB */
```

```
XIP_CPU1_RTT_SIZE          =  0x00040000; /* 256 KB */
XIP_USER_DATA_SIZE         =  0x00020000; /* 128 KB */

/* Automatically calculated base addresses */
XIP_RKPARTITIONTABLE_BASE  = XIP_BASE;
XIP_IDBLOCK_BASE           = (XIP_RKPARTITIONTABLE_BASE + XIP_RKPARTITIONTABLE_SIZE);
XIP_CPU0_TFM_BASE          = (XIP_IDBLOCK_BASE + XIP_IDBLOCK_SIZE);
XIP_CPU1_LOADER_BASE       = (XIP_CPU0_TFM_BASE + XIP_CPU0_TFM_SIZE);
XIP_CPU0_RTT_BASE          = (XIP_CPU1_LOADER_BASE + XIP_CPU1_LOADER_SIZE);
XIP_DSP0_FIRMWARE_BASE     = (XIP_CPU0_RTT_BASE + XIP_CPU0_RTT_SIZE);
XIP_DSP1_FIRMWARE_BASE     = (XIP_DSP0_FIRMWARE_BASE + XIP_DSP0_FIRMWARE_SIZE);
XIP_DSP2_FIRMWARE_BASE     = (XIP_DSP1_FIRMWARE_BASE + XIP_DSP1_FIRMWARE_SIZE);
XIP_CPU1_RTT_BASE          = (XIP_DSP2_FIRMWARE_BASE + XIP_DSP2_FIRMWARE_SIZE);
XIP_USER_DATA_BASE         = (XIP_CPU1_RTT_BASE + XIP_CPU1_RTT_SIZE);

/* ------------------------------------------------------------------------ */
/* SRAM Memory Layout                                                       */
/* ------------------------------------------------------------------------ */
SRAM_BASE               = 0x30200000;

/* User-modifiable sizes */
SRAM_CPU0_TFM_SIZE      = 0x00010000; /* 64 KB */
SRAM_CPU0_RTT_SIZE      = 0x00010000; /* 64 KB */
SRAM_DSP0_SIZE          = 0x00050000; /* 320 KB */
SRAM_DSP1_SIZE          = 0x00048000; /* 288 KB */
SRAM_DSP2_SIZE          = 0x00047000; /* 284 KB */
SRAM_SPI2APB_SIZE       = 0x00001000; /* 4 KB */

/* Automatically calculated base addresses for SRAM */
SRAM_CPU0_TFM_BASE     = SRAM_BASE;
SRAM_CPU0_RTT_BASE     = (SRAM_CPU0_TFM_BASE + SRAM_CPU0_TFM_SIZE);
SRAM_DSP0_BASE         = (SRAM_CPU0_RTT_BASE + SRAM_CPU0_RTT_SIZE);
SRAM_DSP1_BASE         = (SRAM_DSP0_BASE + SRAM_DSP0_SIZE);
SRAM_DSP2_BASE         = (SRAM_DSP1_BASE + SRAM_DSP1_SIZE);
SRAM_SPI2APB_BASE      = (SRAM_DSP2_BASE + SRAM_DSP2_SIZE);

/* ------------------------------------------------------------------------ */
/* DRAM Memory Layout                                                       */
/* ------------------------------------------------------------------------ */
DRAM_BASE               = 0xA0000000;

/* User-modifiable sizes */
DRAM_CPU0_TFM_SIZE       =   0x00100000; /* 1 MB */
DRAM_CPU0_RTT_SIZE       =   0x00100000; /* 1 MB */
DRAM_NPU_SIZE            =   0x00400000; /* 4 MB */
DRAM_DSP0_SIZE           =   0x01300000; /* 19 MB */
DRAM_DSP1_SIZE           =   0x01300000; /* 19 MB */
DRAM_DSP2_SIZE           =   0x01300000; /* 19 MB */
DRAM_CPU1_LOADER_SIZE    =   0x00008000; /* 32 KB */
DRAM_CPU1_RTT_SIZE       =   0x000F8000; /* 992 KB */

/* Automatically calculated base addresses for DRAM */
```

```
DRAM_CPU0_TFM_BASE        =   DRAM_BASE;
DRAM_CPU0_RTT_BASE        =   (DRAM_CPU0_TFM_BASE + DRAM_CPU0_TFM_SIZE);
DRAM_NPU_BASE             =   (DRAM_CPU0_RTT_BASE + DRAM_CPU0_RTT_SIZE);
DRAM_DSP0_BASE            =   (DRAM_NPU_BASE + DRAM_NPU_SIZE);
DRAM_DSP1_BASE            =   (DRAM_DSP0_BASE + DRAM_DSP0_SIZE);
DRAM_DSP2_BASE            =   (DRAM_DSP1_BASE + DRAM_DSP1_SIZE);
DRAM_CPU1_LOADER_BASE     =   (DRAM_DSP2_BASE + DRAM_DSP2_SIZE);
DRAM_CPU1_RTT_BASE        =   (DRAM_CPU1_LOADER_BASE + DRAM_CPU1_LOADER_SIZE);
```

5.2 XIP 模式说明

RK2118 支持 XIP 方式运行，即代码可以直接在 NOR FLASH 中执行 ，这样可以节省内存占用。要编译生成 XIP 格式的固件有两种方式：直接修改 rtconfig.py 和设置环境变量 RTT_BUILD_XIP，具体如下：

方法1：修改`bsp/rockchip/rk2118/rtconfig.py`文件，找到 “XIP”，修改成 “Y”

```
XIP = 'Y'
#XIP = 'N'
```

方法2：申明环境变量 RTT_BUILD_XIP：

```
export RTT_BUILD_XIP=Y
```

5.3 Scons 编译脚本

大部分驱动和应用并不需要关心编译脚本，目前的编译脚本会自动搜索驱动、板级配置、应用和测试等目录的所有源文件进行编译，所以即使增加模块，一般也不需要改脚本。只有在目录结构有变更，或者需要修改编译标志的时候会需要改动编译脚本。

bsp/rockchip/rk2118 目录下有一个 rtconfig.py 文件，这里可以修改 toolchain 和全局的编译链接标志，具体如下：

```
if  CROSS_TOOL == 'gcc':
    PLATFORM = 'gcc'
    EXEC_PATH = '/usr/bin'

# 通过RTT_EXEC_PATH环境变量获取toolchain的路径
if os.getenv('RTT_EXEC_PATH'):
    EXEC_PATH = os.getenv('RTT_EXEC_PATH')

#BUILD = 'debug'
BUILD = 'release'

# 指定XIP模式是否启用
XIP = 'Y'
#XIP = 'N'
if os.getenv('RTT_BUILD_XIP'):
```

```
    XIP = os.getenv('RTT_BUILD_XIP').upper()

if PLATFORM == 'gcc':
    # toolchains
    PREFIX = 'arm-none-eabi-'
    CC = PREFIX + 'gcc'
    AS = PREFIX + 'gcc'
    AR = PREFIX + 'ar'
    CXX = PREFIX + 'g++'
    LINK = PREFIX + 'gcc'
    TARGET_EXT = 'elf'
    SIZE = PREFIX + 'size'
    OBJDUMP = PREFIX + 'objdump'
    OBJCPY = PREFIX + 'objcopy'
    STRIP = PREFIX + 'strip'

    # 全局编译和链接标识
    DEVICE = ' -mcpu=cortex-m33 -mthumb -mfpu=fpv5-sp-d16 -mfloat-abi=hard -ffunction-
sections -fdata-sections'
    CFLAGS = DEVICE + ' -g -Wall -Werror '
    AFLAGS = ' -c' + DEVICE + ' -x assembler-with-cpp -Wa,-mimplicit-it=thumb -
D__ASSEMBLY__ '
    LFLAGS = DEVICE + ' -lm -lgcc -lc' + ' -nostartfiles -Wl,--gc-sections,-
Map=rtthread.map,-cref,-u,Reset_Handler '

    CPATH = ''
    LPATH = ''

    if XIP == 'Y':
        AFLAGS += ' -D__STARTUP_COPY_MULTIPLE -D__STARTUP_CLEAR_BSS_MULTIPLE'
        CFLAGS += ' -D__STARTUP_COPY_MULTIPLE -D__STARTUP_CLEAR_BSS_MULTIPLE -
DRT_USING_XIP'
        LINK_SCRIPT = 'gcc_xip_on.ld'
    else:
        AFLAGS += ' -D__STARTUP_CLEAR_BSS'
       CFLAGS += ' -D__STARTUP_CLEAR_BSS'
        LINK_SCRIPT = 'gcc_xip_off.ld'

    LFLAGS += '-T %s' % LINK_SCRIPT

    if BUILD == 'debug':
        CFLAGS += ' -O0 -gdwarf-2'
        AFLAGS += ' -gdwarf-2'
    else:
        CFLAGS += ' -O2'

    # 编译后处理命令
    POST_ACTION = OBJCPY + ' -O binary $TARGET rtthread.bin\n' + SIZE + ' $TARGET \n'
    POST_ACTION += './align_bin_size.sh rtthread.bin;./rename_rtt.py\n'

    M_CFLAGS = CFLAGS + ' -mlong-calls  -Dsourcerygxx -fPIC '
```

```
    M_LFLAGS = DEVICE + ' -Wl,--gc-sections,-z,max-page-size=0x4 -shared -fPIC -e main -
nostartfiles -nostdlib -static-libgcc'
    M_POST_ACTION = STRIP + ' -R .hash $TARGET\n' + SIZE + ' $TARGET \n'
```

接下来一级的编译脚本是 bsp/rockchip/rk2118 目录下的 SConscript，具体如下：

```
import os
Import('RTT_ROOT')

PROJECT = 'RK2118'
Export('PROJECT')
cwd = str(Dir('#'))
objs = []
list = os.listdir(cwd)

# 加入HAL层编译脚本
objs = SConscript(os.path.join(cwd, '../common/HalSConscript'), variant_dir =
'common/hal', duplicate=0)

# 遍历一级子目录，有发现SConscript就加入编译
for d in list:
    path = os.path.join(cwd, d)
    if os.path.isfile(os.path.join(path, 'SConscript')):
        objs = objs + SConscript(os.path.join(d, 'SConscript'))

# 加入驱动适配层编译脚本
objs = objs + SConscript(os.path.join(RTT_ROOT,
'bsp/rockchip/common/drivers/SConscript'), variant_dir = 'common/drivers', duplicate=0)
# 加入测试代码编译脚本
objs = objs + SConscript(os.path.join(RTT_ROOT, 'bsp/rockchip/common/tests/SConscript'),
variant_dir = 'common/tests', duplicate=0)
# 加入DSP驱动编译脚本
if(os.path.exists(os.path.join(RTT_ROOT, 'components/hifi4/rtt/SConscript'))):
    objs = objs + SConscript(os.path.join(RTT_ROOT, 'components/hifi4/rtt/SConscript'),
variant_dir = 'common/rtt', duplicate=0)
# 加入核间通信编译脚本
if(os.path.exists(os.path.join(RTT_ROOT, 'components/hifi4/shared/SConscript'))):
    objs = objs + SConscript(os.path.join(RTT_ROOT,
'components/hifi4/shared/SConscript'), variant_dir = 'common/shared', duplicate=0)

Return('objs')
```

如果要修改某个子模块的编译标志，可以把这个子模块的编译独立一个 GROUP，然后修改局部标志，下面是一个加局部宏定义的例子：

```
Import('RTT_ROOT')
Import('rtconfig')
from building import *

cwd = GetCurrentDir()
src = Glob('*.c')
CPPPATH = [cwd]                                    # 配置头文件搜索目录，全局有效
LOCAL_CPPDEFINES = ['BOARD_M1_TEST_MARCO']         # 局部宏定义，局部有效

group = DefineGroup('BoardConfig', src, depend = ['RT_USING_BOARD_ADSP_DEMO'], CPPPATH =
CPPPATH, LOCAL_CPPDEFINES = LOCAL_CPPDEFINES )     # 这个宏只会在adsp_demo板编译的时候生效

Return('group')
```

其他一些局部和全部定义，可以参考下面的列表介绍：

```
LOCAL_CCFLAGS                                      # 局部编译标志
LOCAL_CPPPATH                                      # 局部头文件搜索路径
LOCAL_CPPDEFINES                                   # 局部宏定义
LOCAL_ASFLAGS                                      # 局部汇编标志
CCFLAGS                                            # 全局编译标志
CPPPATH                                            # 全局头文件搜索路径
CPPDEFINES                                         # 全局宏定义
ASFLAGS                                            # 全部汇编标志
```

5.4 静态库编译

RT-Thread 支持静态库编译，模块可以先剥离成一个独立的 Group，每一个 Group 都是一个独立的编译单元，可以有自己的编译标志和链接标志，也可以很方便的编译成静态库，下面以 FileTest 模块为例，先看看这个模块的编译脚本 /path/to/rtthread/examples/file/SConscript，具体如下：

```
Import('RTT_ROOT')
Import('rtconfig')
from building import *

cwd = GetCurrentDir()
src = Glob('*.c')
CPPPATH = [cwd, str(Dir('#'))]

group = DefineGroup('FileTest', src, depend = ['RT_USING_FILE_TEST'], CPPPATH = CPPPATH)

Return('group')
```

从上面可以看到，静态库不需要特殊的编译脚本，也不需要设置标志说明要编译成静态库，是否编译成静态库完全取决于编译命令，例如，要将此模块编译成一个静态库，只需要编译时使用如下命令：

```
scons --buildlib=FileTest      # FileTest即编译脚本中我们定的Group名字
```

若编译成功，会有如下输出：

```
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build
AR libFileTest_gcc.a
ranlib libFileTest_gcc.a
Install compiled library... FileTest_gcc
Copy libFileTest_gcc.a => /path/to/rt-thread/examples/file/libFileTest_gcc.a
scons: done building targets.
```

可以看到 RT-Thread 直接把生成的静态库放到编译脚本的同级目录下，要使用这个静态库只需将此静态库加入到 RT-Thread 的静态库列表中，路径加入到 RT-Thread 的静态库搜索路径中，具体如下：

```
from building import *

cwd     = GetCurrentDir()

src     = []
CPPPATH = [cwd]

LIBS    = ['libFileTest_gcc.a']
LIBPATH = []

if GetDepend('SOC_RK2108'):
    LIBPATH = [cwd + '/rk2108']

group = DefineGroup('file-test', src, depend = ['RT_USING_FILE_TEST'], CPPPATH = CPPPATH,
LIBS = LIBS, LIBPATH = LIBPATH)
```

综上，某些不方便开源的模块，可以按上述方法做成静态库，以库文件的形式对外发布。

5.5 访问资源文件

有时候固件会需要访问一些资源文件，例如：音频参数文件，如果这些资源文件较少的情况下，可以直接放到独立分区，然后通过XIP直接读，需要写的话，通过Flash编程接口直接写分区。下面是一个例子：

1. 先在`setting.ini`里新建一个资源分区userdata，具体如下：

```
# 原本最后一个分区的容量全部都给rkstudio，现在我们抠掉128KB给资源分区userdata
[UserPart8]
Name=rkstudio
Type=0x800
PartOffset=0x1d00
PartSize=0x2300
Flag=
File=../../../../../components/hifi4/rtt/rkstudio_bin/rkstudio.bin
```

```
# 改完以后，具体如下：
[UserPart8]
Name=rkstudio
Type=0x800
PartOffset=0x1d00
PartSize=0x2200
Flag=
File=../../../../../components/hifi4/rtt/rkstudio_bin/rkstudio.bin
[UserPart9]
Name=userdata
Type=0x800
PartOffset=0x3f00
PartSize=0x100
Flag=
File=../../userdata.bin
```

2. 打包并烧录`Firmware.img`，也可以直接烧录`userdata.bin`到0x3f00扇区

```
# 下载db_loader，让设备从maskrom切到loader模式
../tools/upgrade/upgrade_tool db ./rkbin/rk2118_db_loader.bin
# 方式1：打包并烧录Firmware.img
./mkimage.sh path/to/your/setting.ini
../tools/upgrade/upgrade_tool wl 0 ./Image/Firmware.img
# 方式2：直接烧userdata.bin
../tools/upgrade/upgrade_tool wl 0x3f00 ./userdata.bin
```

3. 读的时候可以通过如下方式访问：

```
#include <drv_flash_partition.h>

void main(int argc, char **argv)
{
    void *ptr;
    /* 这里的函数参数就是分区名，需要和setting.ini匹配 */
    ptr = get_addr_by_part_name("userdata");
    /* 到这里，就可以通过ptr指针，直接访问userdata.bin的内容了 */
}
```

Note: 因为XIP是只读的，所以上面拿到ptr以后也要确保只读，写的话会触发异常

4. 在XIP模式下不管是通过Flash编程接口来写，还是下一章节文件系统方式来写，都会对CPU性能有较大影响，因为在写的过程中，CPU需要关XIP和关中断，要等编程结束才能自动恢复，所以写的时候需要避开性能敏感的场景。下面是一个Flash编程的例子：

```
int modify_userdata(void* data, uint32_t count)
{
    rt_uint32_t offset;
    rt_err_t ret;
    rt_size_t writes;
    struct rt_mtd_nor_device *snor_device;
```

```
    snor_device = (struct rt_mtd_nor_device *)rt_device_find("snor");
    if (snor_device ==  RT_NULL)
    {
        rt_kprintf("Did not find device: snor....\n");
        return RT_ERROR;
    }

    offset = get_addr_by_part_name("userdata") - XIP_MAP0_BASE0;
    ret = rt_mtd_nor_erase_block(snor_device, offset, count);
    if (ret != RT_EOK)
    {
        rt_kprintf("erase device error: snor....\n");
        return RT_ERROR;
    }
    writes = rt_mtd_nor_write(snor_device, offset, data, count);
    if (writes != count)
    {
        rt_kprintf("write device error: snor....\n");
        return RT_ERROR;
    }

    return RT_EOK;
}
```

5. 因为分区要求64KBytes对齐，如果有多个小文件，每个都单独一个分区显然比较浪费空间，这时候可以把这些资源文件合并成一个文件，然后在文件头加一个每个文件的offset索引，后面通过这个索引去计算地址偏移，然后再访问

5.6 启用文件系统

如果文件非常多，上面那种资源访问方式就不太合适了，此时就要考虑文件系统了，首先就需要通过`menuconfig`打开如下编译开关：

```
    RT_USING_DFS [=y]  #文件系统总开关
    DFS_FILESYSTEMS_MAX [=2] #最大允许的挂载节点数目，即你要挂载几个分区，因为devfs要占用一个，所以这个值要>=（需要挂载的物理分区+1）
    DFS_FILESYSTEM_TYPES_MAX [=2] #最大允许的文件系统数目，你需要支持几种文件系统，我们支持elmfat和devfs，所以是2
    DFS_FD_MAX [=16] #最大可同时操作的文件句柄
    RT_USING_DFS_MNTTABLE [=y] #启用自动挂载表
    RT_USING_DFS_ELMFAT [=y] #启用elmfat文件系统，兼容fat文件系统
    RT_USING_DFS_DEVFS [=y] #启用设备文件系统，可以通过文件接口访问设备驱动
    RT_DFS_ELM_MAX_SECTOR_SIZE [=4096] # 设置扇区大小为4096，以兼容底层spi flash的擦除块大小
```

除了编译开关，还需要把资源文件打包成一个文件系统镜像，可以通过如下方式完成：

```
# resource可以换成你自己的资源文件目录，setting.ini里需要包含名字为root的分区，后面打包出来的镜像需要烧到这个root分区
./mkroot.sh resource board/xxx/setting.ini
ls -l Image/root.img         # 这个文件就是生成的文件系统镜像
-rw-r--r-- 1 rk rk 4587520 Apr 12 08:39 Image/root.img
```

最后就是要确保你的`mnt.c`文件里配置了root分区的挂载，具体如下：

```
const struct dfs_mount_tbl mount_table[] =
{
    {"root", "/", "elm", 0, 0},
    {0}
};
```

6. 驱动开发

驱动的开发实际上分两个部分：HAL 和 Driver，前者可以参考 HAL 的开发指南，这里主要说明后者开发过程中的注意事项：

首先，所有工程师开发前都应该看一下 RT-Thread 的 coding style 文档，路径如下：

```
cd path/to/rt-thread
ls documentation/coding_style_cn.md -l
-rw-rw-r-- 1 rk rk 11955 Feb 28 09:04 documentation/contribution_guide/coding_style_cn.md
```

tools/as.sh 是 RT-Thread 提供的代码风格检查的脚本，不需要手动调用 astyle。

其次，在开始开发前有必要看一下 RT-Thread 的官方开发指南，了解一下驱动会用到的一些系统接口，例如同步与通信、内存管理和中断管理，需要后台线程的话，可以看一下系统的线程和任务管理。这些都可以在开发指南找到介绍，也有一些简单的 demo 可以参考。同时各类驱动，特别是总线型驱动要看一下系统是否有现成的驱动框架，如果有则要按框架要求来实现，所有的驱动框架都在 path_to_rtthread/components/drivers 目录下，目前可以看到如下驱动都有现成的框架：serial、can、timer、i2c、gpio、pwm、mtd、rtc、sd/mmc、spi、watchdog、audio、wifi、usb 等；反之如果 RT-Thread 没有现成的框架，在自己实现前也可以找一下其他 BSP 目录下是否已经有相关驱动可以作为参考。

目前驱动程序被分为两类：公共和私有，前者指的是多个芯片可以共用的驱动，可以放如下目录：

```
/path/to/rt-thread/bsp/rockchip/common/drivers
```

而私有驱动，则只适用特定芯片，可以放到这个芯片 BSP 主目录下的 drivers 目录，例如：

```
/path/to/rt-thread/bsp/rockchip/rk2118/drivers
```

所有的驱动都要以 drv_xxx.c 和 drv_xxx.h，其中 xxx 为模块名或相应缩写，要求全部小写，不能有特殊字符存在，如果必要可以用"_"分割长模块名，如“drv_sdio_sd.c”。各个模块不需要修改编译脚本，目前的脚本已经可以自动搜索 drivers 下所有的源文件，自动完成编译，但是推荐模块加上自己的 Kconfig 配置开关，并且考虑多芯片之间的复用，方便裁剪和调试，具体可以参考如下实现：

```
menu "RT-Thread bsp drivers"

config RT_USING_UART0
    bool "Enable UART0"
    default n

config RT_USING_UART1
    bool "Enable UART1"
    default n

config RT_USING_DSP
    bool "Enable DSP"
    default n

endmenu
```

此外，HAL 目前是通过 hal_conf.h 来做模块开关的，为了方便配置，可以让 Kconfig 和 hal_conf.h 做一个关联，例如串口的 hal_conf.h 配置如下：

```
#if defined(RT_USING_UART0) || defined(RT_USING_UART1) || defined(RT_USING_UART2)
#define HAL_UART_MODULE_ENABLED
#endif
```

驱动的源文件 drv_xxx.c，一定要用 Kconfig 的开关包起来，并且公共驱动要考虑多芯片复用，例如：

```


#if defined(RT_USING_I2C)

#if defined(RKMCU_RK2118)
void do_someting(void)
{

}
#endif

#endif
```

如果驱动有汇编文件，尽量三种编译器都支持：gcc、keil(armcc)、iar，文件名可以按如下规则：xxx_gcc.S、xxx_arm.s、xxx_iar.s，目前汇编文件不会自动加入编译，要手动修改编译脚本，参考 bsp/rockchip/rk2108/drivers/SConscript：

```
Import('RTT_ROOT')
Import('rtconfig')
from building import *
```

```
cwd     = os.path.join(str(Dir('#')), 'drivers')

src = Glob('*.c')
if rtconfig.CROSS_TOOL == 'gcc':
    src += Glob(RTT_ROOT + '/bsp/rockchip/common/drivers/drv_cache_gcc.S')
elif rtconfig.CROSS_TOOL == 'keil':
    src += Glob(RTT_ROOT + '/bsp/rockchip/common/drivers/drv_cache_arm.s')
elif rtconfig.CROSS_TOOL == 'iar':
    src += Glob(RTT_ROOT + '/bsp/rockchip/common/drivers/drv_cache_iar.s')

CPPPATH = [cwd]

group = DefineGroup('PrivateDrivers', src, depend = [''], CPPPATH = CPPPATH)

Return('group')
```

还有就是设备驱动中的定时器延迟，可以简单的使用系统的 tick，如 rt_thread_delay 和 rt_thread_sleep 来实现，但注意不能在中断上下文使用，也不能用 rt_tick_get 代替，因为默认情况 tick 中断的优先级不会比其他中断高，有可能出现某个中断太耗时，导致 tick 出现没有及时更新的情况。中断上下文可以用`HAL_DelayUs`和`HAL_DelayMs`，当然中断里最好还是不要加任何延迟。

最后，RT-Thread 提供了一个自动初始化的接口，驱动如果需要自动初始化，可以调用这些宏：

```
/* board init routines will be called in board_init() function */
#define INIT_BOARD_EXPORT(fn)           INIT_EXPORT(fn, "1")

/* pre/device/component/env/app init routines will be called in init_thread */
/* components pre-initialization (pure software initilization) */
#define INIT_PREV_EXPORT(fn)            INIT_EXPORT(fn, "2")
/* device initialization */
#define INIT_DEVICE_EXPORT(fn)          INIT_EXPORT(fn, "3")
/* components initialization (dfs, lwip, ...) */
#define INIT_COMPONENT_EXPORT(fn)       INIT_EXPORT(fn, "4")
/* environment initialization (mount disk, ...) */
#define INIT_ENV_EXPORT(fn)             INIT_EXPORT(fn, "5")
/* appliation initialization (rtgui application etc ...) */
#define INIT_APP_EXPORT(fn)             INIT_EXPORT(fn, "6")
```

如上所示，其初始化顺序是从上到下，我们约定 BOARD 组只放板级的初始化如 CLK，需要注意在 BOARD 组的初始化过程中由于系统调度子系统还没有初始化，不能使用系统的互斥和同步模块，如 MUTEX 等，因为此时系统的中断还没有开，所有操作都是串行的，不需要考虑并发和竞争。需要注意`rt_malloc`也用到了锁来保证线程安全，所以此时也不能用动态内存分配。而 PREV 组我们可以放一些总线驱动的初始化，如 I2C、SDIO 等，而大部分设备驱动用 INIT_DEVICE_EXPORT 就够了。

如果两个模块有依赖关系，可以放到上面的不同初始化组里，来控制顺序。当然也自己在代码里去控制，举例，如果有两个模块 A 和 B，A 的初始化依赖于 B 的初始化，则最好只把 B 的初始化 EXPORT 出来，然后在 B 里再去调用 A 的初始化。

同一初始化级别则是根据函数名来排序的，编程中最好不要依赖这种顺序，一旦其他工程师不清楚这种顺序依赖，改了函数名就可能破坏顺序，导致异常。

7. 测试用例

提交驱动的同时，最好同步提交测试程序，目前我们的 BSP 测试被分为两个部分：公共和私有，前者是可以多个芯片共用的测试，后者是这个芯片特有的测试，目录分别如下：

```
/path/to/rt-thread/bsp/rockchip/common/tests
/path/to/rt-thread/bsp/rockchip/rk2118/tests
```

可以把测试程序做成 RT-Thread 的 shell 命令，文件名可以命名为 test_xxx.c，其中 xxx 是模块名，例如串口可以用 test_uart.c。下面是一个简单例子：

```
#include <rthw.h>
#include <rtthread.h>

/* 可以沿用驱动的宏定义开关，确保在驱动未启用的时候不会被编译 */
#ifdef RT_USING_DEMO_TEST

#include <stdint.h>

void demo_test(int argc, char **argv)
{
    rt_kprintf("this is demo_test\n");
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 导出msh shell命令: demo_test, 可以传递参数 */
MSH_CMD_EXPORT(demo_test, demo test for driver);
#endif

#endif
```

要启动测试，只需要在命令行按如下命令操作：

```
msh /> demo_test
this is demo_test
```

RT-Thread 还支持一个单元测试框架`utest`，新加入的模块可以用这套测试框架来提交单元测试代码，好处是后面单元测试可以统一管理，统一的入口和测试结果输出也方便后续lava自动化测试集成，进一步维护代码的健壮性。具体的开发教程可以参考RT-Thread官方文档。

8. 调试

8.1 内存问题

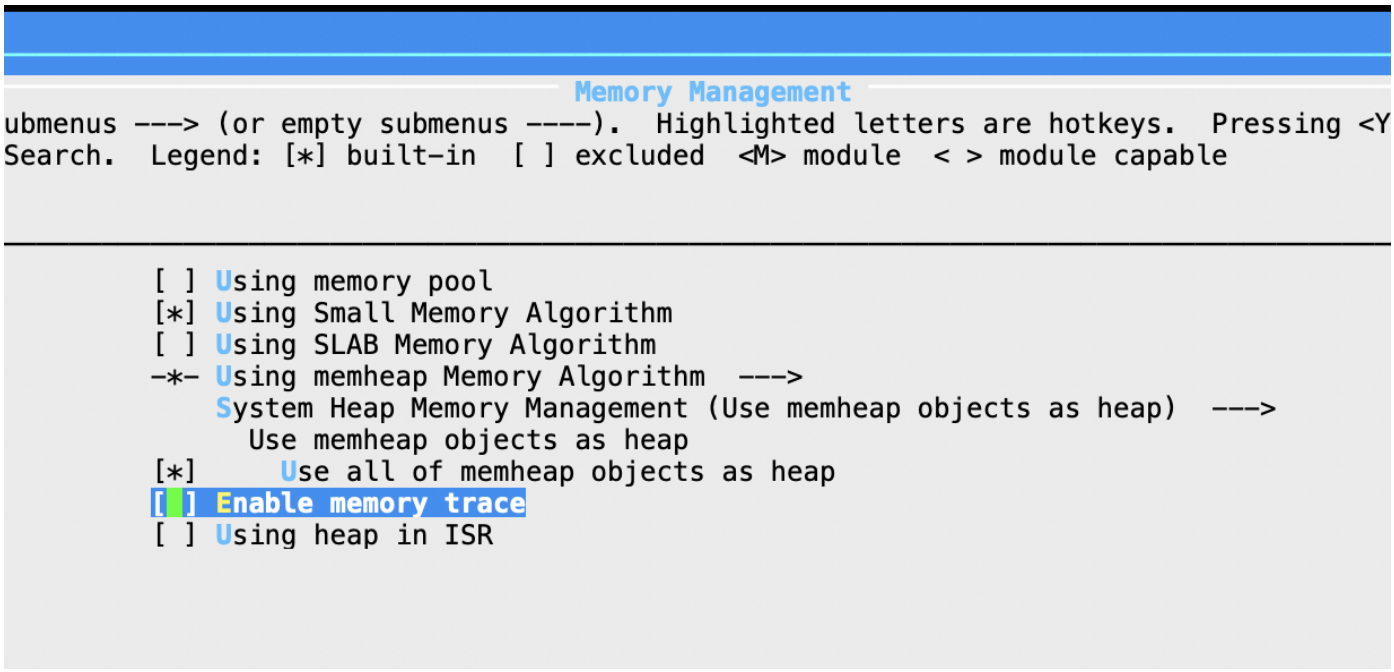
目前RK2118上我们配置了三种堆：系统堆、large 和 uncache，后两者是可选的。下面通过一个表格来说明三者的差异：

	适用场景	api接口
系统堆	通用场景	rt_malloc/rt_free/rt_malloc_align/rt_free_align
large	只有带DDR的产品才会开，用于大块内存分配	rt_malloc_large/rt_free_large/rt_dma_malloc_large/rt_dma_free_large
uncache	不想经过cpu cache的内存，性能比较差，目前主要一些特殊硬件需要	rt_malloc_uncache/rt_free_uncache

查看堆内存分配情况，可以用`free`命令，前者对应系统堆，后者可以查看large和uncache的分配情况，具体如下：

```
memheap   pool size  max used size available size
-------- ---------- ------------- --------------
# large堆情况，max就是到目前为止这个堆最大的使用峰值
large    4194300    48            4194252
# 系统堆情况
heap     1048576    29264         1037792
```

针对最常见的两种内存问题：内存泄漏和访问越界，RT-Thread 另外提供了辅助调试手段，需要打开 RT_USING_MEMTRACE 开关，具体如下：



重新编译后，MSH 会多出两个命令：memtrace 和 memcheck，前者可以导出当前系统的动态内存分配情况，包括当前已用内存大小、最大使用内存大小和内存跟踪情况，具体如下：

```
msh />memtrace
total memory: 462312                    # 系统堆总大小
used memory : 14360                     # 系统堆当前使用大小
maximum allocated memory: 16728         # 系统堆峰值使用大小

memory heap address:
heap_ptr: 0x200871f8                    # 堆起始地址
lfree   : 0x200871f8                    # 最低可用内存地址
```

```
heap_end: 0x200f7ff0                    # 堆结束地址

--memory item information --            # [起始地址 - 大小] 线程名
[0x200871f8 -    1K]                    # 线程名为空，表示是free内存
[0x20087698 -  292] init                # 线程名为init，表示是init线程分配的
[0x200877cc -   48] init
[0x2008780c -  128] init
[0x2008789c -  512] init
[0x20087aac -   12] init
[0x20087ac8 -   32] init
[0x20087af8 -  524] init
[0x20087d14 -   48] init
[0x20087d54 -  128] init
[0x20087de4 -  256] init
[0x20087ef4 -   68] init
[0x20087f48 -   36] init
[0x20087f7c -   44] init
[0x20087fb8 -   96] init
[0x20088028 -   64] init
[0x20088078 -  284] init
[0x200881a4 -   36] init
[0x200881d8 -  216] init
[0x200882c0 -   28] init
[0x200882ec -   28] init
[0x20088318 -   32] init
[0x20088348 -   64] init
[0x20088398 -   64] init
[0x200883e8 -   16] init
[0x20088408 -   16] init
[0x20088428 -   24] init
[0x20088450 -   32] init
[0x20088480 -   32] init
[0x200884b0 -   32] init
[0x200884e0 -  112] init
[0x20088560 -   48] init
[0x200885a0 -   92] init
[0x2008860c -  100] init
[0x20088680 -   48] init
[0x200886c0 -   92] init
[0x2008872c -  100] init
[0x200887a0 -   92] init
[0x2008880c -  100] init
[0x20088880 -  428] init
[0x20088a3c -   16] init
[0x20088a5c -   36] init
[0x20088a90 -   12] init
[0x20088aac -   12] init
[0x20088ac8 -   76] init
[0x20088b24 -   12] init
[0x20088b40 -   12] init
[0x20088b5c -   12] init
[0x20088b78 -    4K] init
```

```
[0x20089bc4 -    36] init
[0x20089bf8 -   520] init
[0x20089e10 -   128] init
[0x20089ea0 -    4K] init
[0x2008aeb0 -  436K]
```

根据上面的信息，可以找到哪个线程吃掉太多内存，也可以找到内存泄漏的可疑线程名称和内存块大小，有了这些信息就可以大大缩小排查范围。

memcheck 是用来检查越界访问的情况，下面是一个越界访问的例子：

```
void test_mem(void)
{
  uint32_t *test = NULL;

  test = (uint32_t *)rt_malloc(16);
  test[4] = 0x55aa55aa;
}
```

放到 shell 里执行后，执行 memcheck，可以看到如下输出：

```
Memory block wrong:
address: 0x62028968
  magic: 0x55aa
   used: 21930
  size: 1136
  thread:
Prev:
address: 0x62028948
  magic: 0x1ea0
   used: 1
  size: 16
  thread: tsh
Next:
address: 0x62028de8
  magic: 0x1ea0
   used: 1
  size: 32
  thread: init
```

分析可以得到如下信息：

```
出错地址：0x62028968
出错原因：magic和used被冲，怀疑前面地址越界访问
前面地址：0x62028948，大小为16，线程是tsh，根据这些信息可以缩小范围
```

需要注意的，memcheck 只能检查一小部分的内存越界，即刚好冲了内存块管理结构的前 12 字节才能检查到，所以即使 memcheck 没检查出错误，也不代表没有越界存在。

8.2 死锁问题

对于死锁问题，RT-Thread 也提供了几个命令，可以列出所有的任务同步和通信的状态，通过这些就可以比较容易定位死锁关系：两个死锁线程的名字，死锁的锁名字，这些信息可以大幅缩小排查范围，具体命令如下：

```
msh >list_msgqueue
msgqueue entry suspend thread
-------- ----  --------------
msh >list_mailbox
mailbox entry size suspend thread
------- ----  ---- --------------
msh >list_mutex
mutex      owner  hold suspend thread
-------- -------- ---- --------------
snorLock (NULL)   0000 0
msh >list_event
event      set    suspend thread
-----  ---------- --------------
msh >list_sem
semaphore v   suspend thread
-------- --- --------------
shrx     000 0
heap     001 0
```

8.3 模块调试

RT-Thread 提供了大部分内核模块的调试 log 开关，默认情况下都是关闭的，开发者可以根据需求打开，具体如下：

```
#define RT_DEBUG     /* 总开关 */

/* log颜色开关，调用dbg_log(level, fmt, ...)等接口,不同level会显示不同颜色 */
#define RT_DEBUG_COLOR
#define RT_DEBUG_INIT_CONFIG  /* 开机初始化调试信息 */
#define RT_DEBUG_THREAD_CONFIG   /* 线程状态切换信息，包括start，suspend和resume */
#define RT_DEBUG_SCHEDULER_CONFIG     /* 调度器调试信息 */
#define RT_DEBUG_IPC_CONFIG   /* 所有任务同步和通信机制的调试信息，调试死锁的时候可以看看 */
#define RT_DEBUG_TIMER_CONFIG  /* 定时器相关信息 */
#define RT_DEBUG_IRQ_CONFIG   /* 调试中断嵌套时可以打开 */
#define RT_DEBUG_MEM_CONFIG   /* 动态内存分配调试信息 */
#define RT_DEBUG_SLAB_CONFIG  /* slab分配器调试信息 */
#define RT_DEBUG_MEMHEAP_CONFIG    /* 内存堆调试信息 */
#define RT_DEBUG_MODULE_CONFIG   /* 动态模块加载的相关信息 */
```

8.4 Fault 调试

目前我们默认启用了 CMBacktrace，出现 Fault 后会自动打印堆栈和调用信息，可以用自带的测试命令 cmb_test 来看 dump 的格式，具体如下：

```
msh >cmb_test DIVBYZERO
thread pri  status      sp     stack size max used left tick  error
------ ---  ------- ---------- ----------  ------  ---------- ---
tshell  20  ready   0x00000090 0x00001000    06%   0x0000000a 000
dma     10  suspend 0x0000005c 0x00000100    35%   0x0000000a 000
tidle   31  ready   0x00000050 0x00000100    43%   0x00000001 000
Firmware name: rtthread, hardware version: 1.0, software version: 1.0
Fault on thread tshell
===== Thread stack information =====
  addr: 20023040    data: 00000000
  addr: 20023044    data: 00000000
  addr: 20023048    data: 040095a6
  addr: 2002304c    data: 20020910
  addr: 20023050    data: 80000000
  addr: 20023054    data: 00000002
  addr: 20023058    data: 04019f6c
  addr: 2002305c    data: 04019f00
  addr: 20023060    data: 04019fd8
  addr: 20023064    data: deadbeef
  addr: 20023068    data: 20021fb0
  addr: 2002306c    data: 0400c329
  addr: 20023070    data: deadbeef
  addr: 20023074    data: 0dadbeef
  addr: 20023078    data: deadbeef
  addr: 2002307c    data: deadbeef
  addr: 20023080    data: deadbeef
  addr: 20023084    data: deadbeef
  addr: 20023088    data: deadbeef
  addr: 2002308c    data: deadbeef
  addr: 20023090    data: deadbeef
  addr: 20023094    data: deadbeef
  addr: 20023098    data: deadbeef
  addr: 2002309c    data: 0400afb1
=====================================
==================== Registers information ====================
  R0 : 00000000  R1 : 0401836c  R2 : 0000000a  R3 : 00000000
  R12: ffffffff  LR : 04010aa1  PC : 04010abe  PSR: 61080000
===============================================================
Usage fault is caused by Indicates a divide by zero has taken place (can be set only if
DIV_0_TRP is set)
Show more call stack info by run: addr2line -e rtthread.elf -a -f 04010abe 04010a9d
0400c325 0400afad
```

最后一行的命令直接复制到 bsp 的主目录下执行，就可以看到错误时候的堆栈，从而找到错误位置：

```
cd /path_to_rtthread_home/bsp/rockchip/rk2118
addr2line -e rtthread.elf -a -f 04010abe 04010a9d 0400c325 0400afad
```

8.5 Backtrace

在软件调试过程中，如果想看软件调用栈，可以通过故意触发`RT_ASSERT`的方式：

```
/* 以下代码会触发当前位置fault，然后打印调用栈 */
RT_ASSERT(0);
```

8.6 Cache一致性

RK2118 CPU有两个 Cache：ICache 和 Dcache ，所以外设如果存在和 CPU 共用内存的场景，就必须要考虑 Cache 一致性的问题，在这个过程中需要注意两点：共用的内存要保持 Cache Line 对齐，在合适的位置做 Cache Maintain。

由于 Cache 都是以 Cache Line 为单位操作的，如果使用的内存范围不以 Cache Line 对齐，在做 Cache Maintain 操作的时候就会影响到相邻的内存（因为在同一个 Cache Line 内）。而不幸的是，可能由于大部分 MCU 都不带 Cache ，所以 RT-Thread 在这一块做的并不好，所以我们额外增加了几个做了 Cache Line 对齐的分配释放函数，具体如下：

```
void *rt_dma_malloc(uint32_t size);                 /* 在系统堆分配DMA内存 */
void rt_dma_free(void *ptr);

void *rt_dma_malloc_large(rt_size_t size);          /* 在large堆分配DMA内存 */
void rt_dma_free_large(void *ptr);
```

所有需要做 Cache Maintain 操作的内存，动态分配的时候都要用带 dma 名字的分配释放函数，如果是静态分配，也要通过属性来保证对齐，例如：

```
/* CACHE_LINE_SIZE定义了Cache Line的大小，内存大小必须是它的整数倍，HAL_CACHELINE_ALIGNED则可以保证起始地址的对齐 */
HAL_CACHELINE_ALIGNED uint8_t setupBuf[CACHE_LINE_SIZE];
```

上面一直在提 Cache Maintain ，这其实可以分为两类操作：Clean 和 Invalidate，不同 OS 对它们的叫法可以略有差异，我们这里沿用最通用的叫法。它们的含义分别如下：

- Clean: 如果 Cache Line 的标识是 dirty 的，即数据加载到 Cache 以后被CPU修改过了，这个操作会把数据写回到下一级存储，对RK2118来说就是写回到内部SRAM或外部DDR
- Invalidate: 把Cache中的数据置为无效

  理解了上面 Clean 和 Invalidate 的差异以后，对于它们的用途就比较好理解了，下面是一个伪代码的例子（假设Cache 下一级是 sram）：

```
camera_record_data(buf);                /* camera录制一段视频到sram */
cache_invalidate(buf);                  /* cpu invalidate这段sram */
cpu_load_and_modify_data(buf);          /* 由于这段sram被invalidate，即cache里可能存在的旧数据都被无效掉了，这样cpu修改前会从sram再次加载，此时就能看到camera录制的最新数据，cpu在这个基础对数据做处理，比如加一些特效 */
cache_clean(buf);                       /* cpu clean这段视频，cpu修改过的视频数据会被写回到sram */
vop_display_data(buf);                  /* 此时vop看到的数据就是cpu做过特效处理的数据 */
```

所以，上面的例子中 invalidate 是为了让 CPU 能看到外设修改过的数据，而 clean 则是为了让外设看到 CPU 修改过的数据，这个关系要理清。除了二者作用不要弄混以外，还要注意在合适的位置操作，请看下面的错误例子：

```
camera_record_data(buf);                /* camera录制一段视频到sram */
cache_invalidate(buf);                  /* cpu invalidate这段sram */
camera_modify_data(buf);                /* camera修改这段数据，比如做一些isp的后处理 */
cpu_load_and_modify_data(buf);          /* 注意这里cpu可能看不到camera修改过的数据 */
cache_clean(buf);                       /* cpu clean这段视频，cpu修改过的视频数据会被写回到sram */
cpu_modify_data(buf);                   /* cpu继续修改视频数据，比如再加一些特效或缩放 */
vop_display_data(buf);                  /* 注意这里vop可能看不到cpu最后修改的数据 */
```

上面的例子告诉我们：在 Maintain 操作之后的数据修改，很可能会导致数据一致性问题，所以一定要注意在合适的位置做 Maintain 。

下面介绍一下 RT-Thread 提供的 Cache Maintain 接口，具体如下：

```
/* ops有两个：RT_HW_CACHE_FLUSH对应我们说的clean操作
 * RT_HW_CACHE_INVALIDATE对应invalidate操作，
 * addr对应你要做maintain操作的起始地址，
 * size对应maintain操作的大小。*/
rt_hw_cpu_dcache_ops(ops, addr, size);
```

还有一个要注意的是 maintain 操作的地址范围要覆盖你修改的数据，但不要超过，下面是一个具体例子：

```
buf = rt_dma_malloc(buf_size);
modify_start = &buf[50];
device_modify_data(modify_start, buf_size-50);
/* 注意起始地址不要用buf，大小也不要用buf_size，只要覆盖你修改的数据范围就够了 */
rt_hw_cpu_dcache_ops(RT_HW_CACHE_INVALIDATE, modify_start, buf_size-50);
```

8.7 JTAG调试