

Rockchip RTOS DSP 开发指南

文件标识：RK-KF-YF-302

发布版本：V2.2.0

日期：2021-03-30

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 © 2021 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

本文档主要介绍 Rockchip DSP 开发的基本方法。

产品版本

芯片名称	版本
RK2108	RT-Thread
PISCES	RT-Thread
RK2206	RKOS

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
2019-06-24	V1.0.0	廖华平	初始版本
2019-08-02	V1.1.0	谢科迪	增加 Floating License 服务器安装说明
2019-09-03	V1.2.0	廖华平	增加固件打包说明
2019-10-10	V1.3.0	廖华平	增加rkos说明
2019-10-16	V1.4.0	廖华平	增加ubuntu安装说明
2020-03-10	V1.5.0	廖华平	增加配置文件安装描述图
2020-05-22	V1.6.0	钟勇汪	修改编译工具源码路径
2020-06-18	V1.7.0	吴佳健	更新打包工具说明
2020-08-06	V1.8.0	吴佳健	新增Vendor Key校验说明
2020-09-18	V1.9.0	吴佳健	更新Map配置信息修改方式和XIP模式说明
2021-02-03	V2.1.0	吴佳健	更新软件环境搭建说明，新增算法库相关说明
2021-03-30	V2.2.0	吴佳健	新增Cache相关说明

目录

Rockchip RTOS DSP 开发指南

1. Rockchip DSP 简介
2. HIFI3 软件环境搭建
 - 2.1 Xplorer 工具安装
 - 2.1.1 Windows环境
 - 2.1.2 Ubuntu环境
 - 2.2 License 安装
 - 2.3 工程导入及编译
 - 2.3.1 工程导入
 - 2.3.2 编译选项说明
 - 2.3.3 编译
 - 2.4 DSP 固件生成
 - 2.5 固件打包配置文件
 - 2.6 固件转换配置文件
 - 2.7 Map 配置信息修改
 - 2.7.1 修改配置文件
 - 2.7.2 修改链接脚本
3. RT-THREAD 代码解析
 - 3.1 代码路径
 - 3.2 配置
 - 3.3 驱动调用
 - 3.4 测试case
 - 3.5 Vendor Key校验测试
4. RKOS 代码解析
 - 4.1 代码路径
 - 4.2 配置
 - 4.3 驱动调用
 - 4.4 测试case
5. 算法库创建及使用
 - 5.1 工程创建
 - 5.2 库文件生成
 - 5.3 算法库导入
 - 5.4 常见编译报错
6. XIP模式
 - 6.1 开发包
 - 6.2 Map修改
 - 6.3 固件打包
 - 6.4 固件烧录
7. 通信协议
 - 7.1 通信协议分析
 - 7.2 Cache说明
 - 7.3 开启Cache

1. Rockchip DSP 简介

DSP 即数字信号处理技术。DSP 作为数字信号处理器将模拟信号转换成数字信号，用于专用处理器的高速实时处理。它具有高速，灵活，可编程，低功耗的界面功能，在图形图像处理，语音处理，信号处理等通信领域起到越来越重要的作用。如下为 Cadence® Tensilica® HiFi3 DSP 的简介。

- HiFi3 DSP 是一种 ISA，支持 2-way SIMD 处理。
- HiFi3 DSP 支持同时处理两个 32x32 或 24x32 bit 数据，4 个 24x24、16x32 或 16x16 bit 数据。
- HiFi3 DSP 支持同时处理两个 IEEE-754 浮点数据。

目前，Rockchip SoC 上集成的 DSP 说明如下：

- RK2108、RK2206 和 PISCES 集成 HIFI3 DSP。

2. HIFI3 软件环境搭建

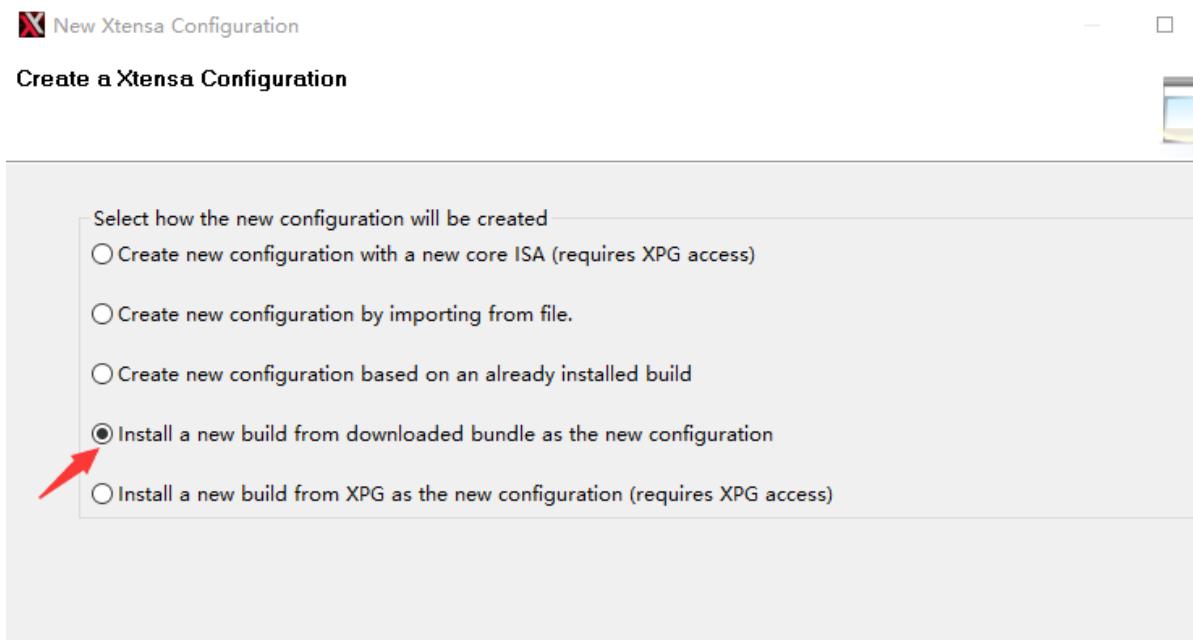
2.1 Xplorer 工具安装

2.1.1 Windows 环境

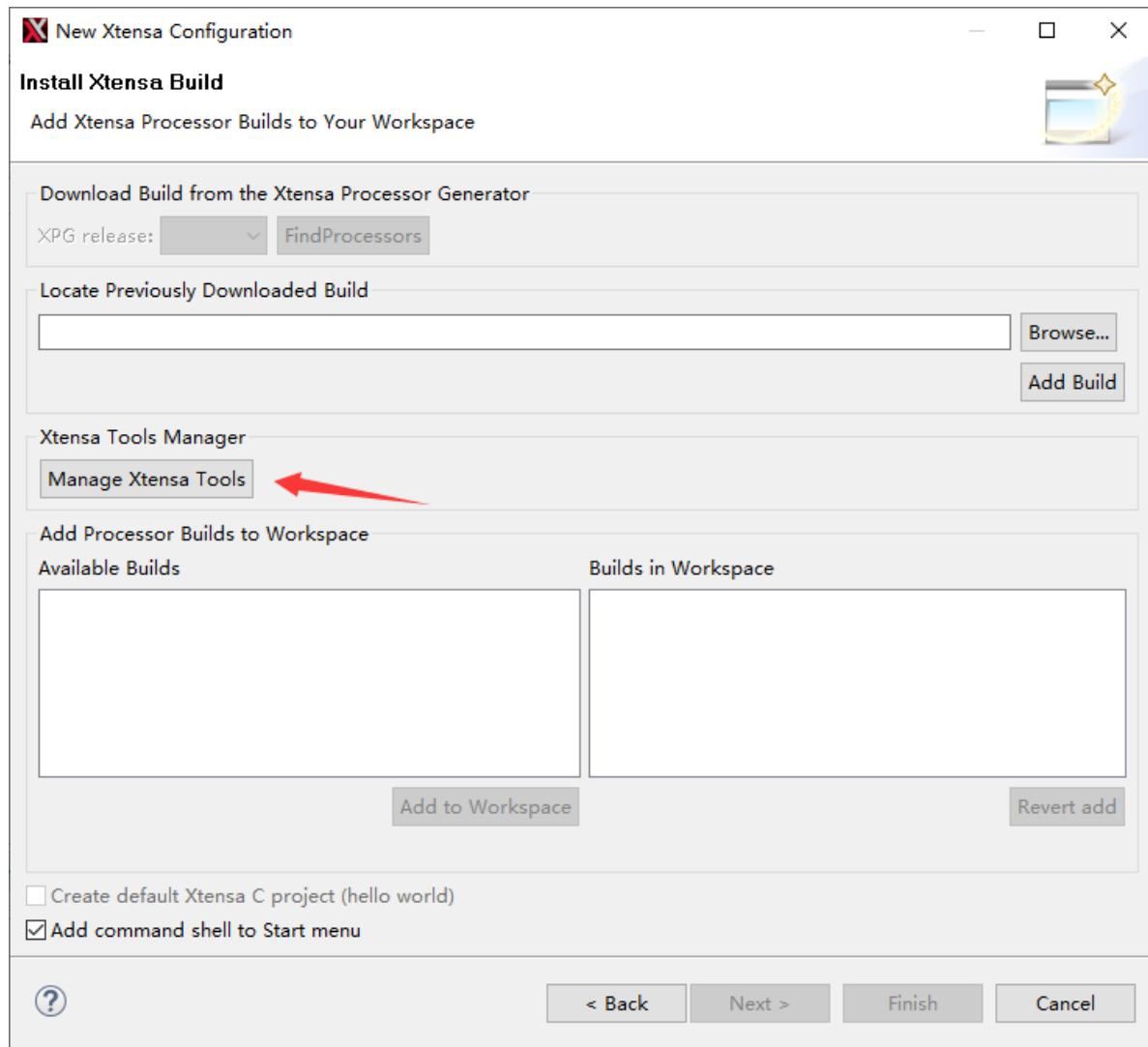
HiFi3 开发工具全称为“Xtensa Xplorer 8.0.8”，License 需要联系 Cadence 获取。我们当前使用的工具安装包为“Xplorer-8.0.8-windows-installer.exe”，配置包为“HiFi3Dev181203_win32.tgz”，配置包基于 RG-2018.9 的基础工具安装包“XtensaTools_RG_2018_9_win32.tgz”。相关安装包都需要找开发人员获取。

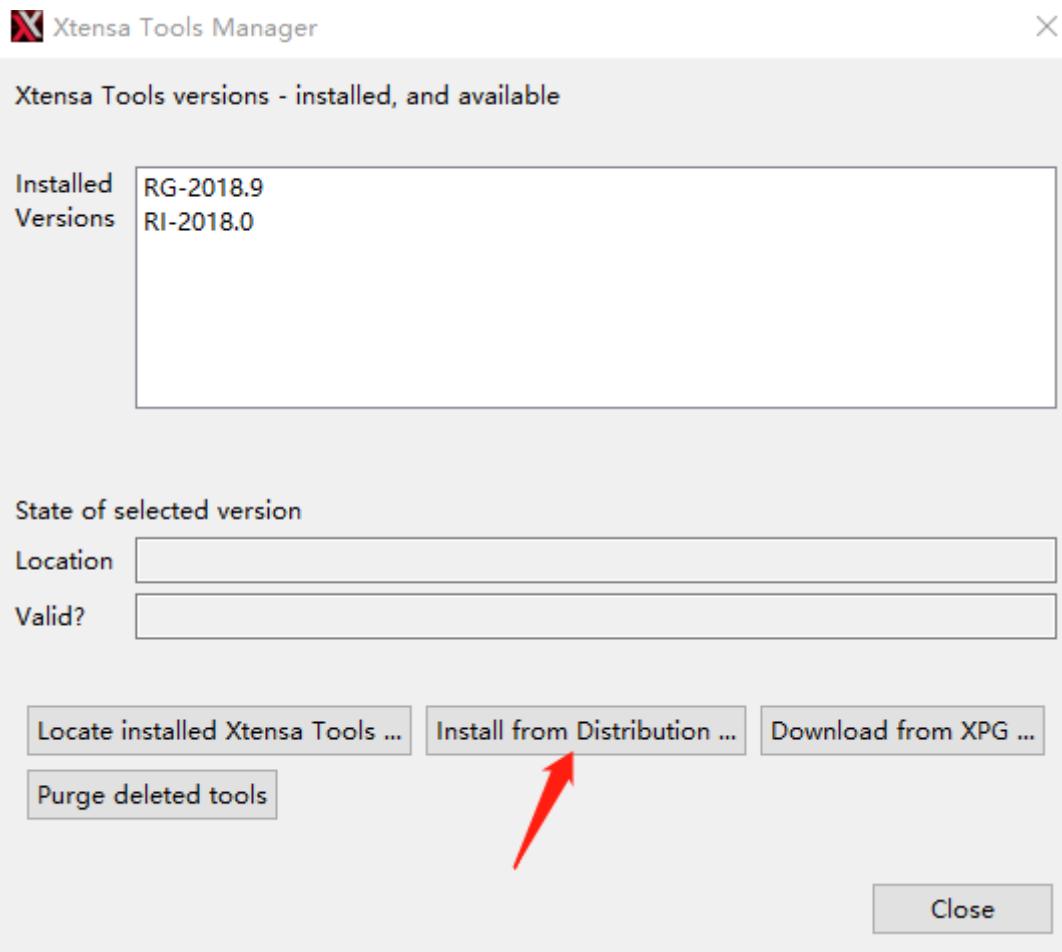
先安装 Xplorer-8.0.8-windows-installer.exe，按照提示即可完成安装。

安装完成后，需要安装工具包，在 Xplorer 中，“File”-->“New”-->“Xtensa Configuration”，找到下图的配置页面并点击 Install 选项，然后点击 Next：

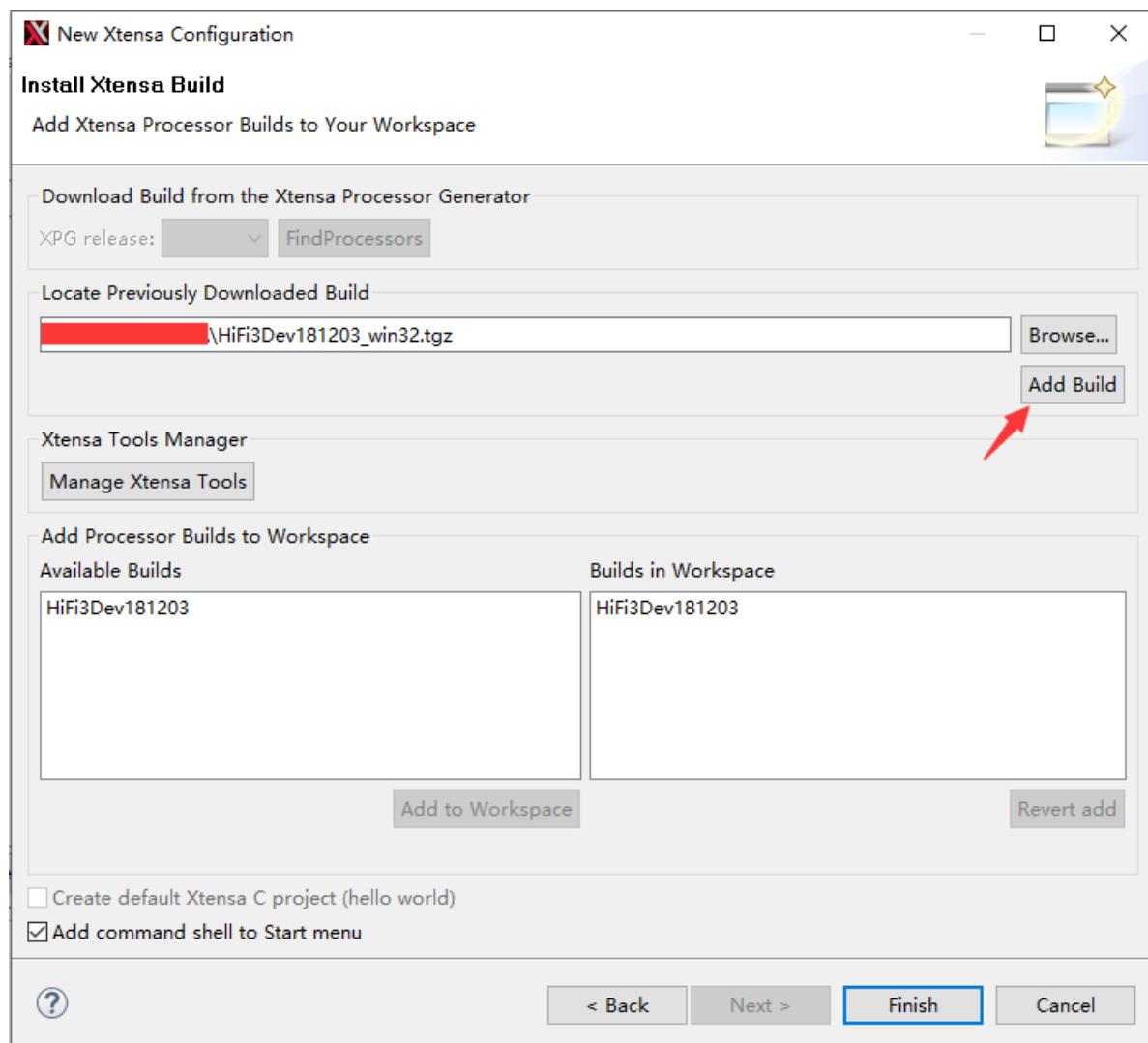


先点击Manage Xtensa Tools，在弹出的窗口点击Install from Distribution，选择XtensaTools_RG_2018_9_win32.tgz工具包路径，按提示安装。





安装完工具包后，返回下图界面，点击Browse选择HiFi3Dev181203_win32.tgz配置包，再点击Add Build添加配置包，待下方窗口显示出对应包名后，点击Finish完成安装。



数据包安装完成后，会在工具栏看到"C:(Active configuration)"，点击箭头出现下拉框，可在栏目中看到 HiFi3Dev181203，点击并选中：



这时候软件左下角的 System Overview 就会看到相关 HiFi3Dev181304 的配置文件，点击相关文件，会看到当前 Core 的配置信息。可以看到对应的 ITMC、DTCM、中断号等。连接外部 INTC 的中断为 INTerrupt0。

System Overview		Workspace Config	Config in XPG
<ul style="list-style-type: none"> Configurations <ul style="list-style-type: none"> HiFi3Dev181203 (RG-2018.0) <ul style="list-style-type: none"> HiFi3Dev181203 HiFi3Dev190311NoWind (R) hifi3_bd5 (RI-2018.0) (D:\an) hifi4_bd7_vfpu (RI-2018.0) (D:\an) hifi_mini (RI-2018.0) (D:\an) sample_config (RI-2018.0) (D:\an) sample_controller (RI-2018.0) (D:\an) sample_fix (RI-2018.0) (D:\an) tie_dev1 (RI-2018.0) (D:\an) tie_dev2 (RI-2018.0) (D:\an) XRC_FusionF1_All_cache (RI) Uninstalled Configs 	<p>Configuration: HiFi3Dev181203 [LX7] Dec 3, 2018 2:37:28 PM hifi3_bd5</p> <p>Edit View Details</p> <p>TIE</p> <p>Build: COMPLETED</p>	rockchips_sw/sw	

2.1.2 Ubuntu环境

由于开发工具在Ubuntu环境下有未知的UI适配问题和使用问题，所以我们建议尽量在windows下开发。

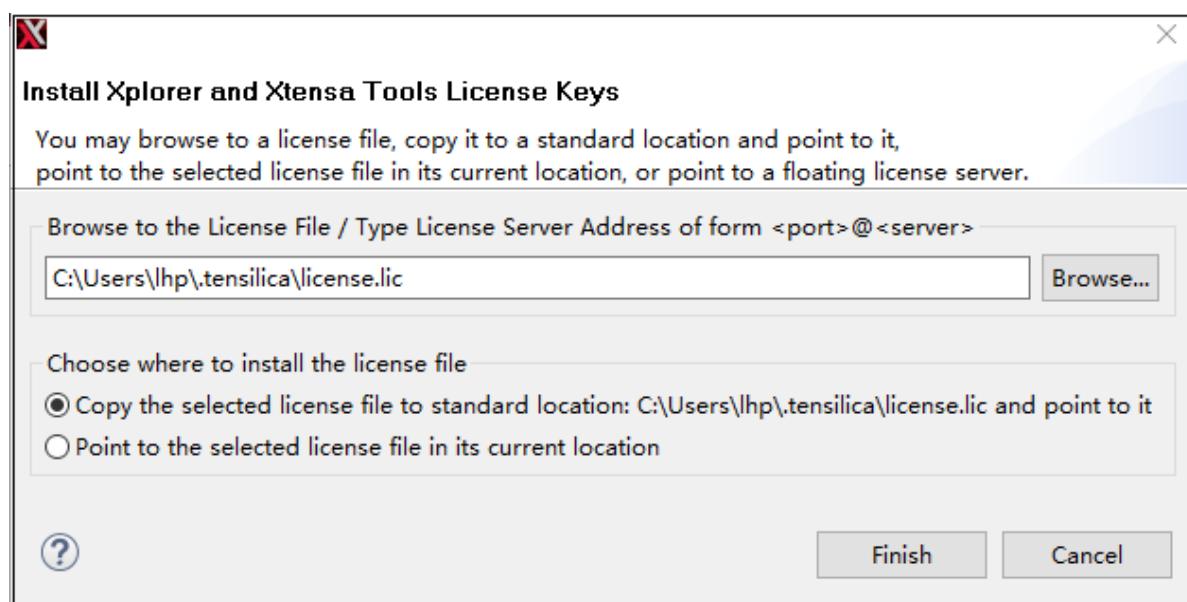
Ubuntu 64bit 版本系统，我们推荐的工具安装包为“Xplorer-8.0.8-linux-x64-installer.bin”，如果是Ubuntu 32bit 版本系统，推荐工具安装包“Xplorer-7.0.9-linux-installer”。配置包为“HiFi3Dev181203_linux.tgz”和“XtensaTools_RG_2018_9_linux.tgz”。安装过程和Windows一致。

因为配置包为32bit，为了兼容64bit系统，需要执行以下命令：

```
sudo apt-get install libgtk2.0-0:i386 gtk2-engines:i386 libc6:i386 libcanberra-gtk3-0:i386 libxtst6:i386 libncurses5:i386
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libstdc++6:i386
sudo apt-get update -y
sudo apt-get update kernel* -y
sudo apt-get update kernel-headers kernel-devel -y
sudo apt-get install compat-libstdc++-33.i686 -y
sudo apt-get install libstdc++.i686 -y
sudo apt-get install gtk2.i686 -y
sudo apt-get install libcanberra-gtk2.i686 -y
sudo apt-get install PackageKit-gtk3-module.i686 -y
sudo apt-get install libXtst.i686 -y
sudo apt-get install ncurses-libs.i686 -y
sudo apt-get install redhat-lsb.i686 -y
```

2.2 License 安装

打开 Xplorer 工具，打开 “Help” -- “Xplorer License Keys”，点击 ”Install License Keys”，输入文件路径名。完成后，点击 ”License Options” 或 ”Check Xtensa Tools Keys” 确认 License 状态。需要注意mac地址要和license中的host id一致。



2.3 工程导入及编译

2.3.1 工程导入

工程目录在根目录的 Projects 下，路径为hifi3/rkdsp/projects，存放不同工程的配置文件和工程文件。

通过 "File" --> "Import" --> "General" --> "Existing Projects into Workspace"，点击Browse，找到projects 目录，按需要导入工程代码，不同项目对应不同的工程名称，如RK2108 对应工程名是 RK2108，RK2206 对应工程名是 RK2206。

导入工程后，可在工具栏No Active Project标签处，点击右侧箭头，下拉选择目标工程，如P:RK2108。

2.3.2 编译选项说明

在工具栏选择编译的Target，分为 Debug、Release 和 ReleaseSize。不同Target设置了不同的优化等级，对代码有不同程度的优化，具体的优化内容可以进入Target修改界面查看和修改，或根据需要自行构建Target。各配置可点击菜单栏T:Debug右侧下拉箭头进行切换。点击其中的Modify可以进入Target修改界面。

Target修改界面常用标签栏有：

Symbols：添加宏定义。

Optimization：修改编译选项。

Linker：选择MAP目录，即链接脚本目录。

Libraries：管理库文件路径。

Addl Linker：添加链接库或编译选项。

并可在窗口下方的All Options确认编译选项是否与预期一致。

2.3.3 编译

目标工程、配置包、Target都选择完毕后，点击工具栏的“Build Active”即可正常进行编译，编译结果存放在工程目录的 bin 目录下。

2.4 DSP 固件生成

工具生成的执行文件只能用于工具仿真，不能直接跑在设备上。运行 CMD 控制台，找到固件生成脚本 generate_dsp_fw.bat 文件，进入到该文件所在目录执行该脚本，使用方式如下：

```
generate_dsp_fw.bat <project name> [config file]
```

其中<project name>为必选项，用于指定工程名，如RK2108

[config file]为可选项，用于指定打包配置文件，如FwConfig.xml（不指定则默认为FwConfig.xml）

注意：若出现“无法找到指定文件”的错误，请确认当前Xplorer中Project是否正确选择，generate_dsp_fw.bat基于工程目录下Makefile自动查找工具路径、Target、Configuration等，如果当前选择的工程不正确，则对应的Makefile不会生成，正确生成的Makefile如下图所示：

```

# Generated by Xplorer, do not modify.
# This Makefile builds _single_build_Debug

%: RCS/%,v
%: RCS/%
%: s.%_
%: SCCS/s.%_
.SUFFIXES: .a .o .c .cc .C .cpp .s .S .h

# There should not be an 'sh.exe' in path
export SHELL = cmd

# There should never be any unknown unix-like tools on the path
# In particular there should not be an 'sh.exe' in the system32 directory
export PATH = [C:\Programs\Xtensa\XtDevTools\install\tools\RG-2018.9-win32\XtensaTools\bin];C:\Programs\Xtensa\XtDevTools\install\tools\RG-2018.9-win32\XtensaTools\lib\iss;c:\Windows\system32

all:
cmd /c "cd /D ""X:\\MCU\\hifi3\\rkdsp\\projects\\RK2108\\bin\\HiFi3Dev181203\\Debug"" && $(MAKE) all"

clean:
cmd /c "cd /D ""X:\\MCU\\hifi3\\rkdsp\\projects\\RK2108\\bin\\HiFi3Dev181203\\Debug"" && $(MAKE) clean"

```

generate_dsp_fw.bat 脚本会将对应工程目录的 FwConfig.xml、Bin2Array.xml、固件等拷贝至 tool 目录下，并调用 HifiFirmwareGenerator.exe 打包固件，最终固件存放于

tools/HifiFirmwareGenerator/output/rkdsp.bin。HifiFirmwareGenerator.exe 的源码存于：

/components/hifi3/rkdsp/tools/source_code/HifiFirmwareGenerator

同时脚本会执行程序“FirmwareArrayGenerator.exe”根据Bin2Array.xml配置将rkdsp.bin转换为相应的C数组文件。FirmwareArrayGenerator.exe 的源码存于：

/components/hifi3/rkdsp/tools/source_code/FirmwareArrayGenerator

不同文件的加载方式请参考3.2节的说明。

2.5 固件打包配置文件

在每个工程目录下，均有一个 FwConfig.xml 文件，该文件采用 Xml 定义一些固件配置。当运行 HifiFirmwareGenerator.exe 时，会解析当前目录的 FwConfig.xml，这里列出几个关键字段的含义：

- **CoreName:** 编译的 Core 的名称，脚本会根据Makefile自动替换 `tag_corename` 字段，开发人员也可手动更改。
- **ToolsPath:** 安装 Xplorer 的工具目录，脚本会根据Makefile自动替换 `tag_toolspath` 字段，开发人员也可手动更改。
- **ExecutableFile:** 输入固件名。
- **SourceCodeMemStart:** DSP 端代码内存空间的起始地址。
- **SourceCodeMemEnd:** DSP 端代码内存空间的结束地址。
- **DestinationCodeMemStart:** MCU 端对应的代码内存空间的地址，因为可能存在内存空间映射情况不同的情况。比如同一块物理内存地址 TCM，DSP 的访问的地址是 0x30000000，MCU 访问的地址是 0x20400000，它们分别对应 SourceCodeMemStart 和 DestinationCodeMemStart。如果地址映射相同，那么填入对应即可。
- **Image:** 输出固件。用于支持可拆分DSP固件。
- **Image/Name:** 固件名，用以区分固件类型。MAIN为主要固件，生成固件名为rkdsp.bin，生成的固件中包含各Section头信息（地址，大小等），用于CPU解析加载DSP固件使用；EXT为额外固件，生成固件名为ext_rkdsp.bin，无Section头信息，按Section地址顺序排列，使用时打包至CPU固件，烧录至Flash指定位置，DSP运行时直接读取对应数据，无需CPU加载。
- **Image/AddrRange:** 指定属于该固件的Section地址范围。以XIP为例，地址范围为 0x60000000~0x60800000，则该范围内的Section将打包至指定固件中（一般为EXT固件）。需要额外注意，生成固件过程中，会将所有代码段解析至Section数组中，按FwConfig.xml中Image的排列顺序分发代码段，如某一代码段在该Image的范围内，则打包至Image并从Section数组中移除，如果当前Image未指定AddrRange，则默认当前Section数组中所有Section都打包至当前Image，因此有指定AddrRange的Image一定要写在未指定AddrRange的Image之前，如下：

```

<Image>
    <Id>2056</Id>
    <Name>EXT</Name>
    <Type>Permanent</Type>
    <AddrRange>0x60000000:0x60800000</AddrRange>
</Image>
<Image>
    <Id>2046</Id>
    <Name>MAIN</Name>
    <Type>Permanent</Type>
</Image>

```

若MAIN写在EXT之前，打包MAIN时，Section数组内所有Section都将打包至MAIN，则打包EXT时，已无有效Section。或在MAIN内也添加AddrRange字段限制，则可以无视排序顺序。AddrRange可分段多次指定，如下：

```

<Image>
    <Id>2056</Id>
    <Name>EXT</Name>
    <Type>Permanent</Type>
    <AddrRange>0x60000000:0x60800000</AddrRange>
</Image>
<Image>
    <Id>2046</Id>
    <Name>MAIN</Name>
    <Type>Permanent</Type>
    <AddrRange>0x30000000:0x30010000</AddrRange>
    <AddrRange>0x30200000:0x30280000</AddrRange>
</Image>

```

2.6 固件转换配置文件

在工程目录下存在Bin2Array.xml文件，用以指定转换模板。文件中字段说明如下：

- **Type:** 生成的C数组类型。
- **Name:** 生成的C数组名称。
- **Input:** 待转换的.bin文件。
- **Output:** 转换输出文件。

2.7 Map 配置信息修改

Xplorer 在链接阶段需要根据 Map 配置信息进行各个数据段的空间分配。在 “T:(active build target) ” --> “Modify”，选择 Linker。可以看到 Standard 选项，可以选择默认的 Map 配置，Xplorer 为开发者提供了 min-rt、sim 等配置，这些配置文件目录存放在“<工具安装目录>\explor8\XtDevTools\install\builds\RG-2018.9-win32\HiFi3Dev181203\xtensa-elf\lib”目录下。配置相关信息可以查看文档“<工具安装目录>\XtDevTools\downloads\RI-2018.0\docs\lsp_rm.pdf”。

2.7.1 修改配置文件

段配置文件为 `memmap.xmm`，以 `map\min-rt\memmap.xmm` 为例，其中设置了 `sram`、`iram0`、`dram0` 三个段空间，其中 `iram0` 和 `dram0` 属于 DSP 的 TCM 内存，访问速度较快，`sram` 属于系统内存，DSP 可以访问，但速度较 TCM 慢。`sram` 和 `iram0` 属性为可读写、可执行，`dram0` 属性为可读写。因此，通常在 `iram0` 中放置代码段，在 `dram0` 中放置数据段，在 `iram0` 或 `dram0` 空间不足时，可以考虑将代码段或数据段放至 `sram`。

将代码段放至 `sram` 有两种方式，一种为在函数前加上 `__attribute__((section(".sram.text"))))`。另一种直接将默认段移至 `sram`。以将默认代码段放至 `sram` 为例，作如下修改：

```
@@ -51,7 +51,7 @@
BEGIN sram
0x20000000: sysram : sram : 0x100000 : executable, writable ;
- sram0 : C : 0x200C0000 - 0x200fffff : .sram.rodata .sram.literal .sram.text
.sram.data .sram.bss;
+ sram0 : C : 0x200C0000 - 0x200fffff : .sram.rodata .sram.literal .sram.text
.literal .text .sram.data .sram.bss;
END sram

BEGIN iram0
@@ -61,7 +61,7 @@ BEGIN iram0
iram0_2 : F : 0x3000057c - 0x3000059b : .DebugExceptionVector.text
.KernelExceptionVector.literal;
iram0_3 : F : 0x3000059c - 0x300005bb : .KernelExceptionVector.text
.UserExceptionVector.literal;
iram0_4 : F : 0x300005bc - 0x300005db : .UserExceptionVector.text
.DoubleExceptionVector.literal;
- iram0_5 : F : 0x300005dc - 0x3000ffff : .DoubleExceptionVector.text
.iram0.literal .iram0.text .text;
+ iram0_5 : F : 0x300005dc - 0x3000ffff : .DoubleExceptionVector.text
.iram0.literal .iram0.text;
END iram0
```

修改完后需要进入 `map` 目录下，使用工具根据该配置文件重新生成链接脚本：

```
cd rkdsp\projects\RK2108\map
<install path>\XtDevTools\install\tools\RG-2018.9-win32\XtensaTools\bin\xt-
genldscripts.exe -b <map directory> --xtensa-core=HiFi3Dev181203
```

其中 `<install path>` 为工具安装目录，如 `C:\usr\XtDevTools\...`，`<map directory>` 为目标 `map` 文件夹，如 `min-rt`。

成功生成会有如下提示：

```
New linker scripts generated in min-rt/ldscripts
```

若生成失败，会打印相应错误，请根据错误提示进行调整。

注意：由于 `sram` 为系统内存，若使用到 `sram`，则需与 CPU 端开发人员协商 `sram` 空间分配，避免出现双方同时访问同一块内存导致出错，如在 `sram0` 中指定起始地址为 `0x200C0000`，则该地址往前的空间为 CPU 可用，该地址往后的空间为 DSP 可用，CPU 端也需要进行相应设置。

2.7.2 修改链接脚本

修改memmap.xmm文件的方式，只能以段为单位进行修改，在某些情况下，存在将某一文件指定至某一段空间的需求，则可以直接修改链接脚本实现：

```
--- a/rkdsp/projects/RK2108/map/min-rt/ldscripts/elf32xtensa.x
+++ b/rkdsp/projects/RK2108/map/min-rt/ldscripts/elf32xtensa.x
@@ -211,6 +211,8 @@ SECTIONS
{
    _iram0_text_start = ABSOLUTE(.);
    *(.iram0.literal .iram.literal .iram.text.literal .iram0.text .iram.text)
+   *file1.o(.literal .literal.* .text .text.*)
+   *file2.c.o(.literal .literal.* .text .text.*)
+   *libx.a:*.*(.literal .literal.* .text .text.*)

    . = ALIGN (4);
    _iram0_text_end = ABSOLUTE(.);
} >iram0_5_seg :iram0_5_phdr
```

注意：存在.o和.c.o两种情况，一般情况下，.c.o为由Xplorer编译产生，.o为标准库内文件，具体是以.o或.c.o结尾，可以直接查看.map文件确认。`*libx.a:*.*(.literal .literal.* .text .text.*)`则是将库内所有.o指定到该段。另外执行[修改配置文件](#)中的命令重新生成链接脚本，将会丢失手动部分修改，因此如果有手动修改，执行自动生成工具前注意做好备份。

修改完成后在Xplorer的Target->Linker 中指定 map 目录，重新编译即可。如果选中“Generate linker map file”，那么就会在编译完成后生成.map 文件，里面记录了详细的地址空间分配，以验证上述修改是否生效。

3. RT-THREAD 代码解析

3.1 代码路径

DSP 框架：

```
bsp/rockchip/common/drivers/dsp.c
bsp/rockchip/common/drivers/dsp.h
```

DSP 驱动适配层：

```
bsp/rockchip/common/drivers/drv_dsp.c
bsp/rockchip/common/drivers/drv_dsp.h
```

DSP 驱动调用流程可以参考以下测试用例：

```
bsp/rockchip/common/tests/dsp_test.c
```

3.2 配置

打开 DSP driver 配置如下，下面以RK2108工程为例：

```
RT-Thread bsp drivers --->
    RT-Thread rockchip rk2108 drivers --->
        Enable DSP --->
            [*] Enable DSP
            [*]   Enable firmware loader to dsp
                Dsp firmware path (Store firmware data in file) --->
                    (/rkdsp.bin) Dsp firmware path
            [ ]   Enable dsp send trace to cm4
            (-1) Config dsp debug uart port
```

“Enable firmware loader to dsp”表示 DSP 驱动启动的时候，会下载 DSP 固件；

“Dsp firmware path”有两个选项有以下两个选项：

- 一个选项是“Store firmware data in file”，固件使用flash中的rkdsp.bin，固件地址在“Dsp firmware path”中指定。“/rkdsp.bin”可以是文件系统中的路径，也可以是一个固件节点（在setting.ini中加入dsp固件分区）。
- 另一个选项是“Store firmware data in builtin”，表示将DSP固件编入到m4的固件中，编译的时候会将工程目录dsp_fw目录下的rkdsp_fw.c¹ 编译，rkdsp_fw.c参考[2.5节 DSP 固件生成](#)中的操作生成。因为工程默认支持XIP，DSP固件会被编译到XIP中。使用这种方式的好处是简单方便，不需要走文件系统操作。但是尽量在支持XIP的时候使用，否则DSP固件会被加载到M4的内存中，浪费内存空间。

“Enable dsp send trace to cm4”表示使能 trace 功能，使得部分 DSP 中的打印 log 可以在 ARM 中打印出来，那么打印 log 就不需要依赖于单独的串口。

“Config dsp debug uart port”表示设置DSP打印的 UART 端口。如果值是-1那么将不会设置。DSP代码中默认使用UART0。

3.3 驱动调用

驱动调用方式可以参考“bsp/rockchip-common/tests/dsp_test.c”。

```
struct rt_device *dsp_dev = rt_device_find("dsp0");
rt_device_open(dsp_dev, RT_DEVICE_OFLAG_RDWR);
rt_device_control(dsp_dev, RKDSP_CTL_QUEUE_WORK, work);
rt_device_control(dsp_dev, RKDSP_CTL_DEQUEUE_WORK, work);
rt_device_close(dsp_dev);
```

调用 `rt_device_open` 时候，会调用到驱动的“`rk_dsp_open`”函数，会执行启动 DSPcore 以及下载固件，并且将 DSP 代码运行起来。

调用“`rt_device_control(dsp_dev, RKDSP_CTL_QUEUE_WORK, work)`”的时候，传入 `work` 指针，驱动会通过 mailbox 将 `work` 发送给 DSP，DSP 解析 `work`，并进行相应的算法操作，将 `work` 处理结果传回来。调用“`rt_device_control(dsp_dev, RKDSP_CTL_DEQUEUE_WORK, work)`”可以取回 DSP 的算法处理结果，如果 DSP 仍在处理中，那么该函数会阻塞，直到 DSP 处理完成。

3.4 测试case

打开 DSP TEST 和 AUDIO TEST 配置如下：

```
RT-Thread bsp test case --->
RT-Thread Common Test case --->
[*] Enable BSP Common TEST
[*] Enable BSP Common AUDIO TEST
[*] Enable BSP Common DSP TEST
[*] Enable Dsp wakeup function
```

编译固件烧录后，在控制台输入dsp_vad_test，可以看到如下log：

```
msh />dsp_vad_test
dsp wakeup_test
Hmsh />ififi3: Hifi3 config done
Hifi3: kwsSetConfig ok
Hifi3: init uv_asr ok
ringbuf_addr:0x30260000, period_size:0x00000280
```

输入audio_capture后，对着麦喊，”xiaoduxiaodu“，可以检测到唤醒词：

```
msh />audio_capture
audio_capture
vad buf: 0x30260000, size: 0x20000 bytes
vad periodsize: 0x280 kbytes
msh />Hifi3: xiaodu_wakeup-----xiaoduxiaodu-----
Hifi3: process return value = 1
work result:0x00000001
```

3.5 Vendor Key校验测试

以flash uuid为例，校验测试基本流程如下：

1. 使用FlashKeyTool读取flash uuid
2. 使用上一步得到的flash uuid计算出校验时使用的key
3. 使用FlashKeyTool工具将key写入到flash的vendor分区
4. CPU测试时从vendor分区读取key，发送至DSP
5. DSP读取flash uuid，经过算法计算后和CPU发送的key进行比对
6. DSP将比对结果发送回CPU
7. CPU和DSP根据校验结果做相应处理

注：步骤1~3为烧录时步骤，也可使用相关源码定制工具，在一个工具内完成读取-计算-烧录工作。

CPU端

测试代码路径为bsp/rockchip/common/tests/dsp_test.c。

控制台输入测试命令dsp_vendor_test，即可开始测试，测试需DSP固件内开启校验支持。

DSP端

代码路径为rkdsp/application/RK2108/key_verify.cpp。

编译前需在Target内定义 `DSP_VENDOR_VERIFY = 1`，并将校验算法接口对接至 rkdsp/application/RK2108/key_verify.cpp 文件内的snor_key_verify函数。该函数将传入CPU发送的key，并将校验函数的返回值返回给CPU。

4. RKOS 代码解析

4.1 代码路径

DSP 驱动层：

```
src/driver/dsp/DspDevice.c  
include/driver/DspDevice.h
```

DSP 驱动调用流程可以参考以下测试用例：

```
src/subsys/shell/Shell_DspDevice.c
```

4.2 配置

打开 DSP Driver 配置如下：

```
BSP Driver    --->  
  Enable DSP   --->  
    [*] Enable DSP  
    [*]   Enable firmware loader to dsp  
          Dsp firmware path (Store firmware data in file)  --->  
          (/rkdsp.bin) Dsp firmware path  
    [ ]   Enable dsp send trace to cm4  
    (-1) Config dsp debug uart port  
    [ ]   Enable dsp jtag
```

menuconfig选项和3.2基本一致，这里说下两个不同的地方：

1. rkdsp_fw.h的存放目录改为了"src/driver/dsp/dsp_fw"。
2. 添加了“Enable dsp jtag”选项，表示使能DSP JTAG。

4.3 驱动调用

驱动调用方式可以参考“src/subsys/shell/Shell_DspDevice.c”。

```
rkdev_create(DEV_CLASS_DSP, 0, NULL);
HDC dsp_dev = rkdev_open(DEV_CLASS_DSP, 0, NOT_CARE);
rk_dsp_open(dsp_dev, 0);
rk_dsp_control(dsp_dev, RKDSP_CTL_QUEUE_WORK, work);
rk_dsp_control(dsp_dev, RKDSP_CTL_DEQUEUE_WORK, work);
rk_dsp_close (dsp_dev);
rkdev_close(dsp_dev);
rkdev_delete(DEV_CLASS_DSP, 0, NULL);
```

调用说明可以参考3.3节中的介绍，只是函数名有些不同，执行方式是一样的。

4.4 测试case

打开 DSP Test配置如下：

```
Components Config --->
  Command shell --->
    [*] Enable DSP shell command
```

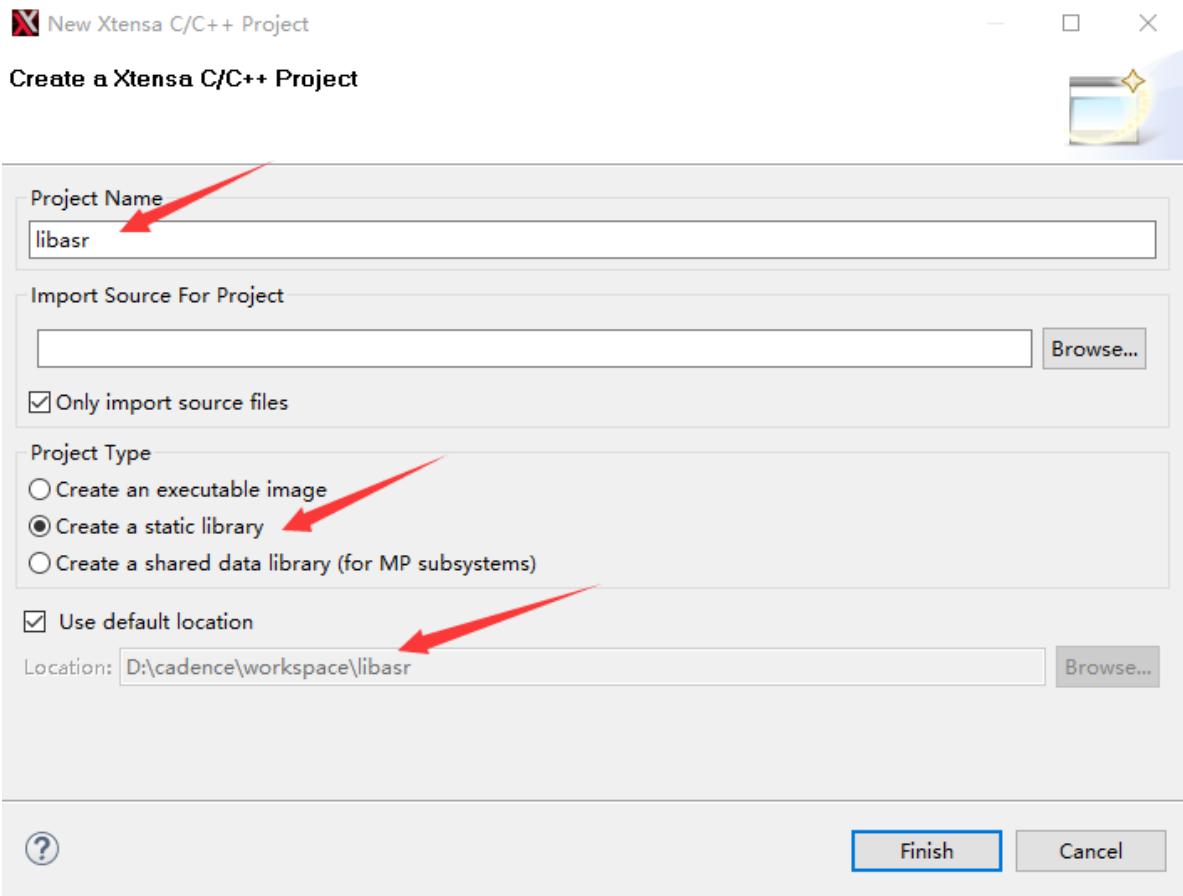
编译固件烧录后，在控制台输入dsp_test，可以看到如下LOG:

```
dsp_test
Hifi3: Hifi3 config done
Hifi3: kwsSetConfig ok
Hifi3: init uv_asr ok
config end
[A.DspTe] [000024.61]workresult:0x00000000
[A.DspTe] [000024.61]work result:0x00000000
[A.DspTe] [000024.61]work result:0x00000000
```

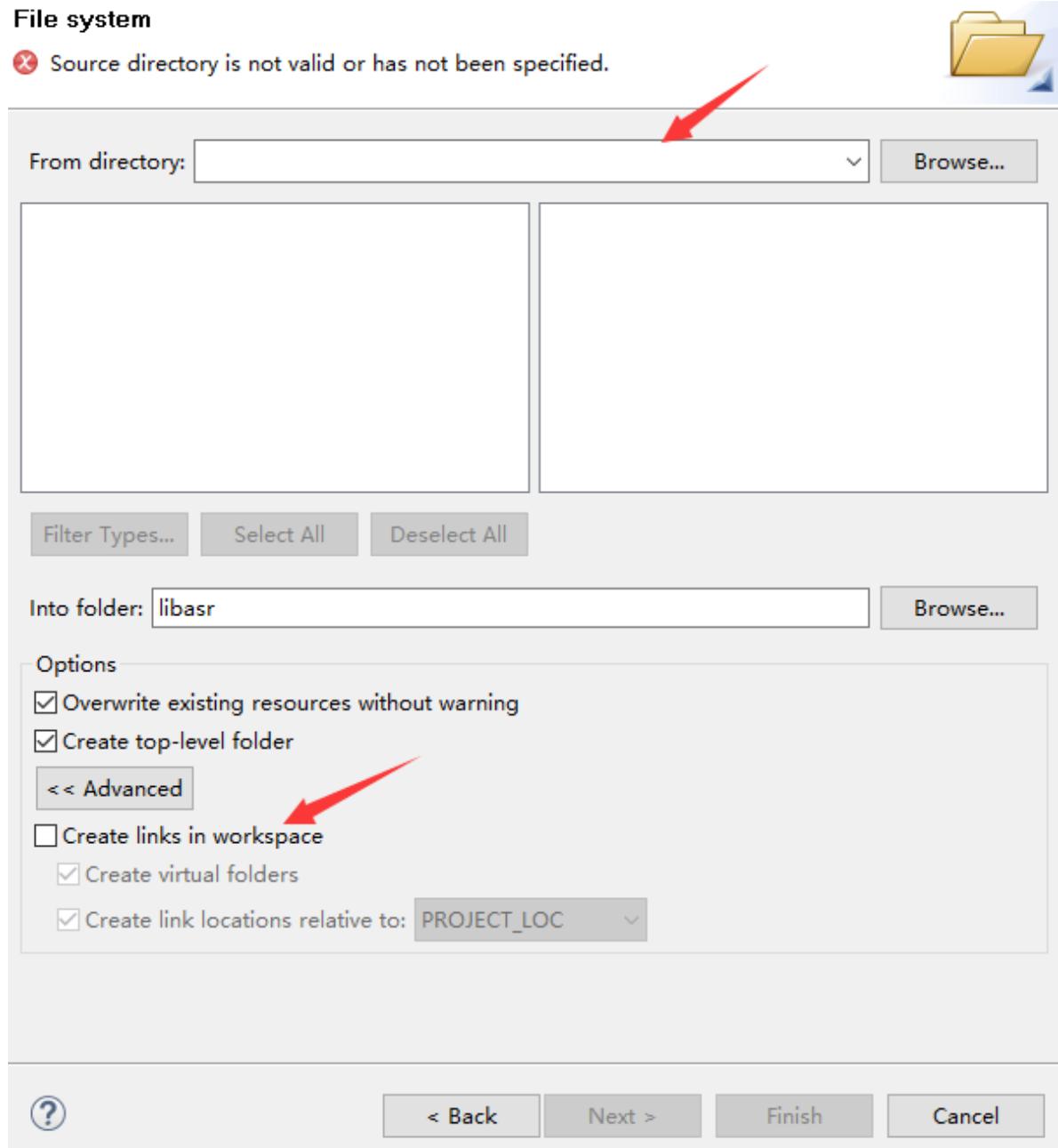
5. 算法库创建及使用

5.1 工程创建

点击File->New->Xtensa C/C++ Project，在接下来的界面中填入工程相关信息。Location也就是工程目录，可以用默认的，也可以自己选择。点击Finish后，在“Project Explorer”栏目中可以看到新建的工程“libasr”。



相关的头文件和库文件可以直接放到工程目录下，也可以右击工程，点击File->Import->General->File System，在Browse中选取代码文件或者目录，在Into folder选择导入的位置。“Create links in workspace”表示只创建文件的链接，文件本身不会拷贝到工程目录下，这样方便维系代码路径结构。我司提供的工程也是采用这种文件形式。



5.2 库文件生成

参考[编译选项说明](#)，对编译选项进行修改，确认后，可点击Build Active开始编译。编译完成后，将在project/bin/HiFi3Dev181203//中生成libasr.a文件。Target指的是[编译选项说明](#)中提到的Debug、Release、ReleaseSize等。接下来把库文件和需要的头文件放到新建目录libasr，供给应用工程使用。

5.3 算法库导入

将生成的libasr.a放入hifi3/rkdsp/library/libasr中，然后通过点击File->Import->File System导入工程。然后进入Target修改界面，在Add linker标签页中添加“-lasr”。在Librarys标签页中添加路径“\${xt_project_loc}\..\..\library\libasr”。在Linker标签页中的选取合适的map。

接下来进行接口适配，一般会在libasr文件夹中添加一个适配文件libasr.cpp和libasr.h，并导入工程，具体接口参考工程示例。然后在algorithm_handler.cpp添加调用接口。

5.4 常见编译报错

1. 找不到库文件

确认Librarys标签页中路径是否正确，libxxx.a文件是否命名正确(必须为libxxx.a的形式，且中间的xxx为库名，添加链接时使用-lxxx)，且放在正确的路径，并且导入工程成功。

2. 找不到函数定义

先与问题1一样，排查库导入是否正确。再确认.a对应的.h文件中，接口函数声明处是否有使用extern "C"声明，形式如下：

```
#ifndef __LIBXXX_H__
#define __LIBXXX_H__

#ifndef __cplusplus
extern "C" {
#endif

int libxxx_init(int arg);
int libxxx_deinit(int arg);

#ifndef __cplusplus
}
#endif
#endif // __LIBXXX_H__
```

或将接口文件如libasr.cpp改为libasr.c重新编译。

6. XIP模式

在TCM和SRAM都不够使用的情况下，允许将.text和.rodata两段放至Flash中，需要CPU端开启XIP支持，可咨询相关开发人员。放至Flash中的代码和数据读取较慢，会比较明显的影响执行效率，因此关键函数和数据建议优先放至TCM和SRAM中。

6.1 开发包

XIP模式下需要使用call0方式跳转，目前使用的开发包为HiFi3Prod200605_Call0_win32.tgz，请咨询相关开发人员获取，编译时请注意开启Target->Optmization->Enable long calls选项。

6.2 Map修改

在memmap.xmm文件添加如下配置，重新生成链接脚本及指定代码到该段的方法请参考[Map配置信息修改一章](#)：

```
BEGIN srom
0x60000000: sysrom : srom : 0x800000 : executable ;
srom0 : C : 0x607C0000 - 0x607fffff : .srom.info.rodata .srom.rodata
.srom.literal .srom.text .rom.store;
END srom
```

注意：该段仅可以放置只读段，如`.literal .text .rodata`。

6.3 固件打包

编译固件后，打包固件时需要使用FwConfigXIP.xml配置文件，使用方式如下：

```
generate_dsp_fw.bat RK2108 FwConfigXIP.xml
```

成功打包后会有LOG提醒当前是否为XIP模式，如果与预期不符请检查FwConfigXIP.xml中EXT段的地址范围与Map中地址范围是否一致。

注：若工程目录下不存在FwConfigXIP.xml文件，可以参考[固件打包配置文件](#)在FwConfig.xml文件基础上自行添加EXT段即可。

6.4 固件烧录

固件打包后将会生成rkdsp_fw.c和ext_rkdsp.bin两个文件，其中rkdsp_fw.c文件使用方法不变。
ext_rkdsp.bin文件直接烧录至Flash指定位置。以[Map修改](#)中的地址为例，srom0的起始地址为0x607C0000，与SROM起始地址0x60000000偏移为0x7C0000，Flash的块大小为0x200，因此0x7C0000 / 0x200 = 0x3E00。烧录固件时则将ext_rkdsp.bin烧录至0x3E00位置。

7. 通信协议

7.1 通信协议分析

MCU 和 DSP 通过 Mailbox 进行通信，Mailbox 包含 4 个通道，一个通道传输 32bit 的 CMD 和 Data 数据。每次发送消息，CMD 通道传输命令码，表示这次消息进行哪些操作；Data 通道传输数据，一般为 work 或者 config 的 buffer 指针。命令码存于在 drv_dsp.h 中，DSP_CMD_WORK、DSP_CMD_READY、DSP_CMD_CONFIG 等。

当 DSP 启动后，DSP 会进行自身的初始化等操作。初始化完成后，DSP 会发送 DSP_CMD_READY 命令，MCU 端接收到后，会调用“rk_dsp_config”函数对 DSP 进行 trace 等相关信息的配置。DSP 接收到 DSP_CMD_CONFIG 并且配置完成后，会发送 DSP_CMD_CONFIG_DONE，表示配置已经完成，可以进行算法工作。这三次消息发送相当于一个握手过程，握手完成后就可以进行算法调用。

7.2 Cache说明

在MCU与DSP进行数据交互的过程中，由于双方Cache独立（在DSP上为16k icache和16k dcache），因此存在Cache一致性问题，即双方向某一地址中写入数据时，会先往Cache中写入，在某些条件下（如Cache块替换、显式调用write back或Cache策略为write through等）才会实际往内存中写，这将导致双方获取对方数据时，可能获取的不是最新数据。因此，为避免相关问题，通常建议使用时显式调用invalidate、writeback相关接口，各平台接口如下：

```
// RK2108:  
rt_hw_cpu_dcache_ops(RT_HW_CACHE_FLUSH | RT_HW_CACHE_INVALIDATE, (void *)addr,  
size);  
rt_hw_cpu_dcache_ops(RT_HW_CACHE_FLUSH, (void *)addr, size);  
rt_hw_cpu_dcache_ops(RT_HW_CACHE_INVALIDATE, (void *)addr, size);  
// RK2206:  
rk_dcache_ops(RK_HW_CACHE_CLEAN | RK_HW_CACHE_INVALIDATE, (void *)addr, size);  
rk_dcache_ops(RK_HW_CACHE_CLEAN, (void *)addr, size);  
rk_dcache_ops(RK_HW_CACHE_INVALIDATE, (void *)addr, size);  
// HIFI3 DSP:  
xthal_dcache_region_writeback_inv((void *)addr, size);  
xthal_dcache_region_writeback((void *)addr, size);  
xthal_dcache_region_invalidate((void *)addr, size);
```

在MCU发送消息给DSP前，通常由MCU输出到DSP的buffer，MCU端需要使用writeback接口，DSP端需要使用invalidate接口，由DSP输出到MCU的buffer，MCU端需要使用invalidate接口²。在DSP处理完发送消息给MCU前，DSP输出的buffer需要使用writeback接口。以RK2108平台接口为例，示例如下：

```
char *buf_to_dsp; // write by MCU, read by DSP  
int buf_len_to_dsp;  
char *buf_to_mcu; // write by DSP, read by MCU  
int buf_len_to_mcu;  
  
/* MCU side start */  
rt_hw_cpu_dcache_ops(RT_HW_CACHE_FLUSH, buf_to_dsp, buf_len_to_dsp);  
rt_hw_cpu_dcache_ops(RT_HW_CACHE_INVALIDATE, buf_to_mcu, buf_len_to_mcu);  
rt_device_control(dsp_dev, RKDSP_CTL_QUEUE_WORK, work); // Send message to DSP  
/* MCU side end */  
  
/* DSP side start */  
xthal_dcache_region_invalidate(buf_to_dsp, buf_len_to_dsp);  
/* processing */  
xthal_dcache_region_writeback(buf_to_mcu, buf_len_to_mcu);  
/* DSP side end */  
  
/* MCU side start */  
rt_device_control(dsp_dev, RKDSP_CTL_DEQUEUE_WORK, work); // Wait message from  
DSP  
/* MCU side end */
```

7.3 开启Cache

ITCM、DTCM、SRAM的Cache通常为开启状态，XIP、PSRAM等地址空间的Cache可能为关闭状态，可以使用以下接口开启Cache：

```
// XCHAL_CA_WRITEBACK仅为其中一种策略，其他策略可跳转定义处查看  
xthal_set_region_attribute((void *)addr_base, size, XCHAL_CA_WRITEBACK, 0);
```

注意，开启Cache并不一定对性能有提升，还取决于Cache命中率，若命中率较低，可能开启Cache会导致性能下降，因此建议结合实际选择。目前DSP平台没有Cache命中率统计工具，建议结合软件仿真和代码流程分析。

1. 实际目标文件由menuconfig中RT_DSPFW_FILE_NAME指定。 [↳](#)

2. 建议MCU在发消息前使用invalidate接口，因为MCU为多线程环境，此时申请的Buffer，可能因为其他线程使用过而存在Cache中，并且大概率为“脏”数据，若在DSP执行写回操作后，MCU端Cache出现块替换，可能导致脏数据被写回，从而污染正确数据。 [↳](#)