AN0067
应用笔记

AT32移植RT-Thread

前言

这篇应用笔记描述了将RT-Thread移植到AT32平台时的注意事项和步骤，并对RT-Thread的驱动如何编写进行举例说明。

支持型号列表：

支持型号	AT32F 系列

目录

表目录

图目录

1 简介

RT-Thread的核心内容主要包括两部分：1、RT-Thread源码，2、ENV工具。源码部分包括了操作系统内核、外设驱动、应用组件等，ENV工具是一个具有裁剪配置灵活、支持在线软件包升级并集成多种开发编译环境等诸多强大功能的工具集。故RT-Thread的移植主要包括RT-Thread的ENV环境移植以及RT-Thread设备驱动的移植两大部分。

2 RT-Thread 移植

2.1 移植前准备

开始移植前需要将所需要的开发工具和编译环境等准备就绪。1、工程生成和自动配置主要采用 RT-Thread 原生的 ENV 工具，ENV 工具不仅可以按裁剪配置生成第三方 IDE 开发工具的工程文件，还可以直接采用内部集成的 gcc 交叉编译工具链进行编译，故 ENV 工具是不可或缺的。2、常用的第三方 IDE 开发工具主要是我们所熟悉的 Keil 和 IAR，这都是非常常用开发工具，此部分的准备工作就不再累述，大家按自己的开发使用习惯安装配置好即可。3、要想将 RT-Thread 移植到 AT32 平台，和硬件驱动相关的源码采用 AT32 官方的标准库最为合适不过了。

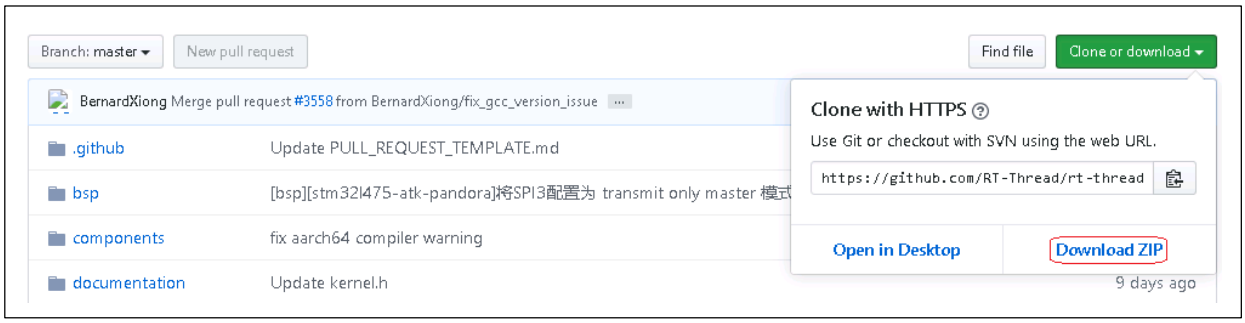
2.1.1 下载 ENV 工具

ENV 工具是 RT-Thread 官方为其系统源码开发编译等所配套制作的工具包，可将其视作 RT-Thread 的一部分。可到 RT-Thread 官方网站的下载页面进行下载。地址如下：https://www.rt-thread.org/page/download.html

2.1.2 下载 RT-Thread 源码包

在 ENV 工具的同一下载界面也可看到源码包的下载链接。
RT-Thread 源码包放在 github 上进行托管，故想获取最新的 RT-Thread 源码包，可使用 github 或者码云的下载渠道进行下载。因只需获得源码即可，在进入对应页面后选择下载 ZIP 包的方式就可以。以 github 的下载方式为例，示意如下

图 1. ZIP 包下载



2.1.3 下载 AT32 外设库

对于AT32底层驱动部分最好采用雅特力官方提供的AT32外设标准库，其可在雅特力官方网站进行下载，在进入后可看到AT32各个系列的产品讯息，对于不同系列选择不同栏位并进入下载。在进入下载页面后即可进行所需资源的下载，以下载标准库为例，选择点击如下链接：

图 2. 标准库下载

如需官方提供的其他资料（如 Pack、RM 等）也可以一并在此处进行下载。

2.2 ENV 环境的移植

在之前的介绍中可知道ENV对于RT-Thread是非常重要的一部分，其重要性主要体现在以下三部分：

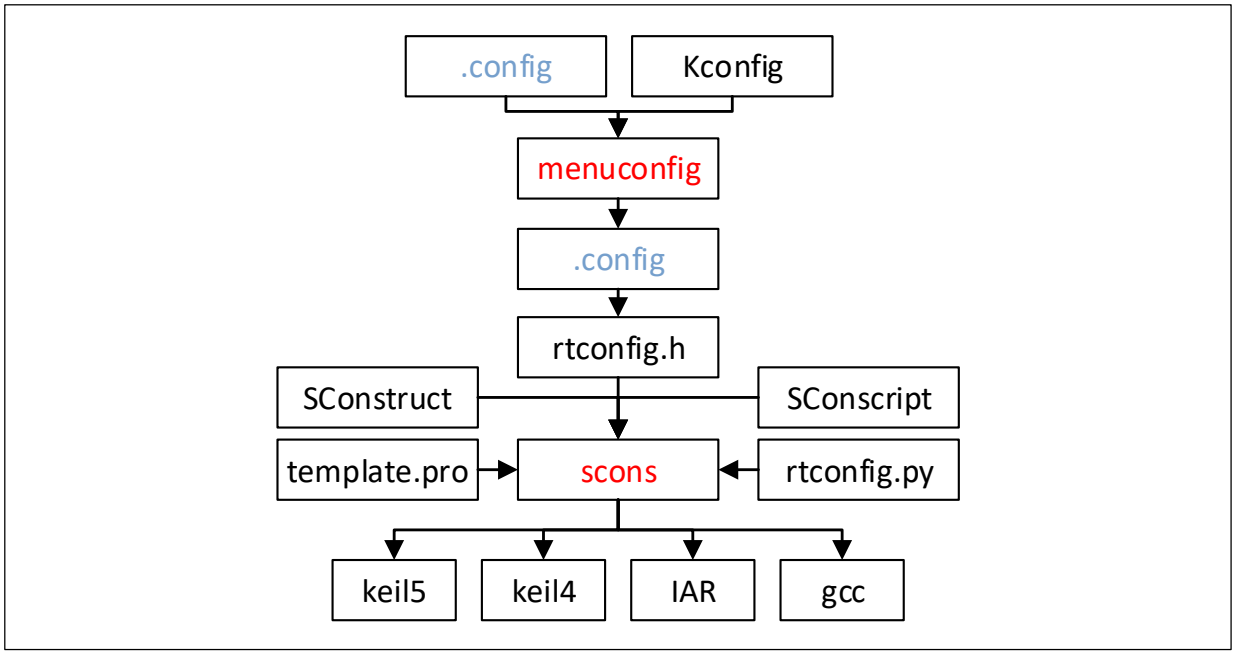
1、 构建工程，自动将源码添加到工程中。

2、 解决依赖，自动添加头文件到工程中。

3、 系统裁剪，自动生成宏定义到工程中。

以上三个部分所谓的自动配置其实是依赖于脚本文件和配置文件所共同决定的内容，按照文件内设置的规则来完成，其中构建工程和解决依赖是由SCons脚本来完成，系统裁剪部分是通过menuconfig调用配置的Kconfig来实现交互式配置。其基本框架如下：

图 3. ENV 框架



通过以上流程分析可知想要搭建一个rt-thread系统的ENV开发环境，需要准备四个部分的文件：

1、 编译器配置相关：rtconfig.py。

2、 源码添加、头文件依赖处理相关：SConstruct和SConscript脚本。

3、 menuconfig图形配置相关：Kconfig脚本。

4、 项目工程生成相关：template.pro模板工程。

5、 添加AT32外设库。

2.2.1 rtconfig.py 配置

rtconfig.py主要做的工作是选择处理器内核和配置编译器相关的设置。我可从其他地方拷贝一份相似的文件到板级支持包的根目录或者在此新建一份。此部分内容在修改时应特别注意以下两点：

1、 AT32F403A系列采用的是cortex-m4内核，采用拷贝方式的文件应注意各处内核配置项的修改，如‘ --cpu cortex-m3’就应该改为‘ --cpu cortex-m4’。

2、 因可选择的编译方式多样化，编译环境又与个人软件安装习惯有关，因此在配置这部分时有必要注意一下，因按实际安装情况来对EXEC_PATH参数进行配置。

以RT-Thread/bsp/at32/at32f403a-start包中的此文件为例，内容如下：

```
import os
# toolchains options
```

```
ARCH='arm'
CPU='cortex-m4'
CROSS_TOOL='gcc'
# bsp lib config
BSP_LIBRARY_TYPE = None
if os.getenv('RTT_CC'):
    CROSS_TOOL = os.getenv('RTT_CC')
if os.getenv('RTT_ROOT'):
    RTT_ROOT = os.getenv('RTT_ROOT')
# cross_tool provides the cross compiler
# EXEC_PATH is the compiler execute path, for example, CodeSourcery, Keil MDK, IAR
if   CROSS_TOOL == 'gcc':
    PLATFORM      = 'gcc'
    EXEC_PATH     = r'C:\Users\XXYYZZ'
elif CROSS_TOOL == 'keil':
    PLATFORM      = 'armcc'
    EXEC_PATH     = r'C:/Keil_v5'
elif CROSS_TOOL == 'iar':
    PLATFORM      = 'iar'
    EXEC_PATH     = r'C:/Program Files (x86)/IAR Systems/Embedded Workbench 8.0'
if os.getenv('RTT_EXEC_PATH'):
    EXEC_PATH = os.getenv('RTT_EXEC_PATH')
BUILD = 'debug'
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'
    DEVICE = ' -mcpu=cortex-m4 -mthumb -mfpu=fpv4-sp-d16 -mfloat-abi=hard -ffunction-
```

```
sections -fdata-sections'
    CFLAGS = DEVICE + ' -Dgcc'
    AFLAGS = ' -c' + DEVICE + ' -x assembler-with-cpp -Wa,-mimplicit-it=thumb '
    LFLAGS = DEVICE + ' -Wl,--gc-sections,-Map=rt-thread.map,-cref,-u,Reset_Handler -T board/linker_scripts/link.lds'
    CPATH = ''
    LPATH = ''
    if BUILD == 'debug':
        CFLAGS += ' -O0 -gdwarf-2 -g'
        AFLAGS += ' -gdwarf-2'
    else:
        CFLAGS += ' -O2'
    CXXFLAGS = CFLAGS
    POST_ACTION = OBJCPY + ' -O binary $TARGET rtthread.bin\n' + SIZE + ' $TARGET \n'
elif PLATFORM == 'armcc':
    # toolchains
    CC = 'armcc'
    CXX = 'armcc'
    AS = 'armasm'
    AR = 'armar'
    LINK = 'armlink'
    TARGET_EXT = 'axf'
    DEVICE = ' --cpu Cortex-M4.fp '
    CFLAGS = '-c ' + DEVICE + ' --apcs=interwork --c99'
    AFLAGS = DEVICE + ' --apcs=interwork '
    LFLAGS = DEVICE + ' --scatter "board\linker_scripts\link.sct" --info sizes --info totals --info unused --info veneers --list rt-thread.map --strict'
    CFLAGS += ' -I' + EXEC_PATH + '/ARM/ARMCC/include'
    LFLAGS += ' --libpath=' + EXEC_PATH + '/ARM/ARMCC/lib'
    CFLAGS += ' -D__MICROLIB '
    AFLAGS += ' --pd "__MICROLIB SETA 1" '
    LFLAGS += ' --library_type=microlib '
    EXEC_PATH += '/ARM/ARMCC/bin/'
    if BUILD == 'debug':
        CFLAGS += ' -g -O0'
        AFLAGS += ' -g'
```

```
    else:
        CFLAGS += ' -O2'
    CXXFLAGS = CFLAGS
    CFLAGS += ' -std=c99'
    POST_ACTION = 'fromelf --bin $TARGET --output rtthread.bin \nfromelf -z $TARGET'
elif PLATFORM == 'iar':
    # toolchains
    CC = 'iccarm'
    CXX = 'iccarm'
    AS = 'iasmarm'
    AR = 'iarchive'
    LINK = 'ilinkarm'
    TARGET_EXT = 'out'
    DEVICE = '-Dewarm'
    CFLAGS = DEVICE
    CFLAGS += ' --diag_suppress Pa050'
    CFLAGS += ' --no_cse'
    CFLAGS += ' --no_unroll'
    CFLAGS += ' --no_inline'
    CFLAGS += ' --no_code_motion'
    CFLAGS += ' --no_tbaa'
    CFLAGS += ' --no_clustering'
    CFLAGS += ' --no_scheduling'
    CFLAGS += ' --endian=little'
    CFLAGS += ' --cpu=Cortex-M4'
    CFLAGS += ' -e'
    CFLAGS += ' --fpu=VFPv4_sp'
    CFLAGS += ' --dlib_config "' + EXEC_PATH + '/arm/INC/c/DLib_Config_Normal.h"'
    CFLAGS += ' --silent'
    AFLAGS = DEVICE
    AFLAGS += ' -s+'
    AFLAGS += ' -w+'
    AFLAGS += ' -r'
    AFLAGS += ' --cpu Cortex-M4'
    AFLAGS += ' --fpu VFPv4_sp'
    AFLAGS += ' -S'
```

```
    if BUILD == 'debug':
        CFLAGS += ' --debug'
        CFLAGS += ' -On'
    else:
        CFLAGS += ' -Oh'
    LFLAGS = ' --config "board/linker_scripts/link.icf"'
    LFLAGS += ' --entry __iar_program_start'
    CXXFLAGS = CFLAGS
    EXEC_PATH = EXEC_PATH + '/arm/bin/'
    POST_ACTION = 'ielftool --bin $TARGET rtthread.bin'
```

2.2.2 源码文件依赖

源码文件和头文件的依赖关系会直接影响到工程的编译结果，所以需要有规范严谨的脚本来严格控制文件与文件间的关系及文件与工程间的关系。

SConstruct文件对于一个单独的BSP来说，有且只有一份，它作为文件关系处理的起始，是scons命令的入口，在其中规定了相关联的文件目录路径和一些环境参数的配置。最终会在规定的关联路径下查找SConscript脚本文件并执行。此文件可参考被移植模板的内容再结合自身文件目录结构的特点来针对性修改，以RT-Thread/bsp/at32/at32f403a-start包中的此文件为例，内容如下：

```
import os
import sys
import rtconfig
if os.getenv('RTT_ROOT'):
    RTT_ROOT = os.getenv('RTT_ROOT')
else:
    RTT_ROOT = os.path.normpath(os.getcwd() + '/../../..')
sys.path = sys.path + [os.path.join(RTT_ROOT, 'tools')]
try:
    from building import *
except:
    print('Cannot found RT-Thread root directory, please check RTT_ROOT')
    print(RTT_ROOT)
    exit(-1)
TARGET = 'rtthread.' + rtconfig.TARGET_EXT
DefaultEnvironment(tools=[])
env = Environment(tools = ['mingw'],
    AS = rtconfig.AS, ASFLAGS = rtconfig.AFLAGS,
```

```
        CC = rtconfig.CC, CFLAGS = rtconfig.CFLAGS,
        AR = rtconfig.AR, ARFLAGS = '-rc',
        LINK = rtconfig.LINK, LINKFLAGS = rtconfig.LFLAGS)
env.PrependENVPath('PATH', rtconfig.EXEC_PATH)
if rtconfig.PLATFORM == 'iar':
        env.Replace(CCCOM = ['$CC $CFLAGS $CPPFLAGS $_CPPDEFFLAGS
$_CPPINCFLAGS -o $TARGET $SOURCES'])
        env.Replace(ARFLAGS = [''])
        env.Replace(LINKCOM = env["LINKCOM"] + ' --map project.map')
Export('RTT_ROOT')
Export('rtconfig')
SDK_ROOT = os.path.abspath('./')
if os.path.exists(SDK_ROOT + '/libraries'):
        libraries_path_prefix = SDK_ROOT + '/libraries'
else:
        libraries_path_prefix = os.path.dirname(SDK_ROOT) + '/libraries'
SDK_LIB = libraries_path_prefix
Export('SDK_LIB')
# prepare building environment
objs = PrepareBuilding(env, RTT_ROOT, has_libcpu=False)
at32_library = 'f403a_407'
rtconfig.BSP_LIBRARY_TYPE = at32_library
# include libraries
objs.extend(SConscript(os.path.join(libraries_path_prefix, at32_library, 'SConscript')))
# make a building
DoBuilding(TARGET, objs)
```

SConscript是文件依赖导入的实际脚本文件，此文件详细的描述了源码文件导入工程的方式和依赖关系，并且对源码文件相对应的头文件的包含也是必须的一部分。整个工程所需编译的源码文件管理由多个相类似的SConscript文件构建并形成树状结构，其可能存在于各级目录下，在执行或调用的过程中是由顶层的SConscript进行递归的查找与执行。在此简单的讲解一下此文件的语法与规则，以RT-Thread/bsp/at32/at32f403a-start/board中的此文件举例。内容如下：

```
import os
import rtconfig
from building import *
Import('SDK_LIB')
```

```
cwd = GetCurrentDir()
# add general drivers
src = Split('''
src/board.c
src/at32_msp.c
''')
path = [cwd]
path += [cwd + '/inc']
startup_path_prefix = SDK_LIB
if rtconfig.CROSS_TOOL == 'gcc':
    src += [startup_path_prefix +
'/f403a_407/firmware/cmsis/cm4/device_support/startup/gcc/startup_at32f403a_407.s']
elif rtconfig.CROSS_TOOL == 'keil':
    src += [startup_path_prefix +
'/f403a_407/firmware/cmsis/cm4/device_support/startup/mdk/startup_at32f403a_407.s']
elif rtconfig.CROSS_TOOL == 'iar':
    src += [startup_path_prefix +
'/f403a_407/firmware/cmsis/cm4/device_support/startup/iar/startup_at32f403a_407.s']
CPPDEFINES = ['AT32F403AVGT7']
group = DefineGroup('Drivers', src, depend = [''], CPPPATH = path, CPPDEFINES =
CPPDEFINES)
Return('group')
```

在工程配置下，有多个必要源文件需要导入的情况可采用以下的方式进行配置

```
src = Split('''
src/board.c
src/at32_msp.c
''')
```

此处指定了一个src变量，目的是将当前SConscript脚本文件目录下的src/main.c、src/at32_msp.c源文件无需任何依赖和限制的赋值到src，并在最后配置到工程环境里。以keil环境的来举例说明此项对工程的影响，在工程生成后打开可在工程的文件列中看到导入的*.c文件。

有条件依赖的单个源文件导入工程配置的方法，可以参考如下方式。

```
if GetDepend(['BSP_USING_SDIO']):
    src += ['mnt.c']
```

此处目的是在BSP_USING_SDIO宏使能的情况下将当前SConscript脚本文件目录下的mnt.c源文件导入到工程配置中，其通俗的表达方式是在使能SDIO接口驱动的情况下才将源文件中有调用到SDIO驱动的mnt.c导入到工程中。这样能让工程的配置和管理更灵活、更严谨。

在源文件导入工程后，所对应的头文件应该被编译环境所需要。在此也应该将所依赖的头文件的存放

路径加入到对应的编译环境中。其头文件路径导入方式可以参考如下

```
path = [cwd]
path += [cwd + '/inc']
```

头文件的配置是在脚本中指定了一个path变量，并将所需的头文件路径都赋值到path变量中，并在最后导入到工程环境里。多个路径可以采用这样的方式逐个增加。以keil环境的来举例说明此项对工程的影响，在工程生成后打开Options for Target → C/C++ → Include Paths定义框中看到所加入的各头文件路径。

在SConscript中也可以对工程中所需要的宏定义进行定义，例如在工程中定义一个AT32F403AVGT7的宏即可采用如下的方式。

```
CPPDEFINES = ['AT32F403AVGT7']
```

以keil环境的来举例说明此项对工程的影响，在工程生成后打开Options for Target → C/C++ → Define定义框中看到所加入的AT32F403AVGT7宏定义。

最终SConscript对于当前目录环境下所需要处理的源文件、头文件和宏定义等参数都定义完成后调用如下语句可让所定义内容生效。

```
group = DefineGroup('Drivers', src, depend = [''], CPPPATH = path, CPPDEFINES = CPPDEFINES)
```

2.2.3 图形界面裁剪配置

RT-Thread系统具有图形界面可裁剪配置方式，是基于menuconfig工具运行，针对RT-Thread源码特点来编写了一整套的Kconfig配置文件。对此如果需要将自己写的驱动、应用等源码融入到RT-Thread的图形界面裁剪配置框架，也需要编写源码内容相对应的Kconfig文件。

Kconfig文件的主要是将各配置项的内容相对应的生成宏定义，以在源码达到可裁剪、文件依赖可配置的目的。

在移植过程中可参照相似芯片架构和文件布局的Demo包来进行，比如RT-Thread/bsp/at32系列的文件夹及文件模式。整体分析后知道有三个Kconfig文件需要重点关注和处理，第一个是Board BSP包的根目录Kconfig，第二个是与主控芯片配置相关的Kconfig，部署在Libraries文件夹下，第三个是与开发板外设驱动相关的Kconfig，部署在board文件夹。

第一个Board BSP包根目录下Kconfig示例内容如下

```
mainmenu "RT-Thread Configuration"
config BSP_DIR
    string
    option env="BSP_ROOT"
    default "."
config RTT_DIR
    string
    option env="RTT_ROOT"
    default "../../.."
config PKGS_DIR
```

```
    string
    option env="PKGS_ROOT"
    default "packages"
source "$RTT_DIR/Kconfig"
source "$PKGS_DIR/Kconfig"
source "../libraries/Kconfig"
source "board/Kconfig"
```

此部分内容主要是定义了BSP_DIR、RTT_DIR、PKGS_DIR三个宏定义，主要指定了BSP根目录、RT-Thread源码根目录与Board BSP的相对路径关系，故此处的路径需按实际情况进行配置。最后会对几大板块的Kconfig文件进行解析与调用。

第二个与主控芯片配置相关的Kconfig示例内容如下

```
config SOC_FAMILY_AT32
    bool
config SOC_SERIES_AT32F403
    bool
    select ARCH_ARM_CORTEX_M4
    select SOC_FAMILY_AT32
config SOC_SERIES_AT32F403A
    bool
    select ARCH_ARM_CORTEX_M4
    select SOC_FAMILY_AT32
config SOC_SERIES_AT32F407
    bool
    select ARCH_ARM_CORTEX_M4
    select SOC_FAMILY_AT32
config SOC_SERIES_AT32F413
    bool
    select ARCH_ARM_CORTEX_M4
    select SOC_FAMILY_AT32
config SOC_SERIES_AT32F415
    bool
    select ARCH_ARM_CORTEX_M4
    select SOC_FAMILY_AT32
config SOC_SERIES_AT32F425
    bool
    select ARCH_ARM_CORTEX_M4
```

```
    select SOC_FAMILY_AT32
config SOC_SERIES_AT32F435
    bool
    select ARCH_ARM_CORTEX_M4
    select SOC_FAMILY_AT32
config SOC_SERIES_AT32F437
    bool
    select ARCH_ARM_CORTEX_M4
    select SOC_FAMILY_AT32
```

此部分主要是进行芯片系列等信息的宏定义配置。

第三个board外设相关的Kconfig示例内容如下

```
menu "Hardware Drivers Config"
config SOC_AT32F403AVGT7
    bool
    select SOC_SERIES_AT32F403A
    select RT_USING_COMPONENTS_INIT
    select RT_USING_USER_MAIN
    default y
menu "Onboard Peripheral Drivers"
    config BSP_USING_SERIAL
        bool "Enable USART (uart1)"
        select BSP_USING_UART
        select BSP_USING_UART1
        default y
endmenu
menu "On-chip Peripheral Drivers"
    config BSP_USING_GPIO
        bool "Enable GPIO"
        select RT_USING_PIN
        default y
    menuconfig BSP_USING_UART
        bool "Enable UART"
        default y
        select RT_USING_SERIAL
        if BSP_USING_UART
            config BSP_USING_UART1
```

```
                bool "Enable UART1"
                default y
            config BSP_USING_UART2
                bool "Enable UART2"
                default n
            config BSP_USING_UART3
                bool "Enable UART3"
                default n
        endif
        # … add other configuration
endmenu
endmenu
```

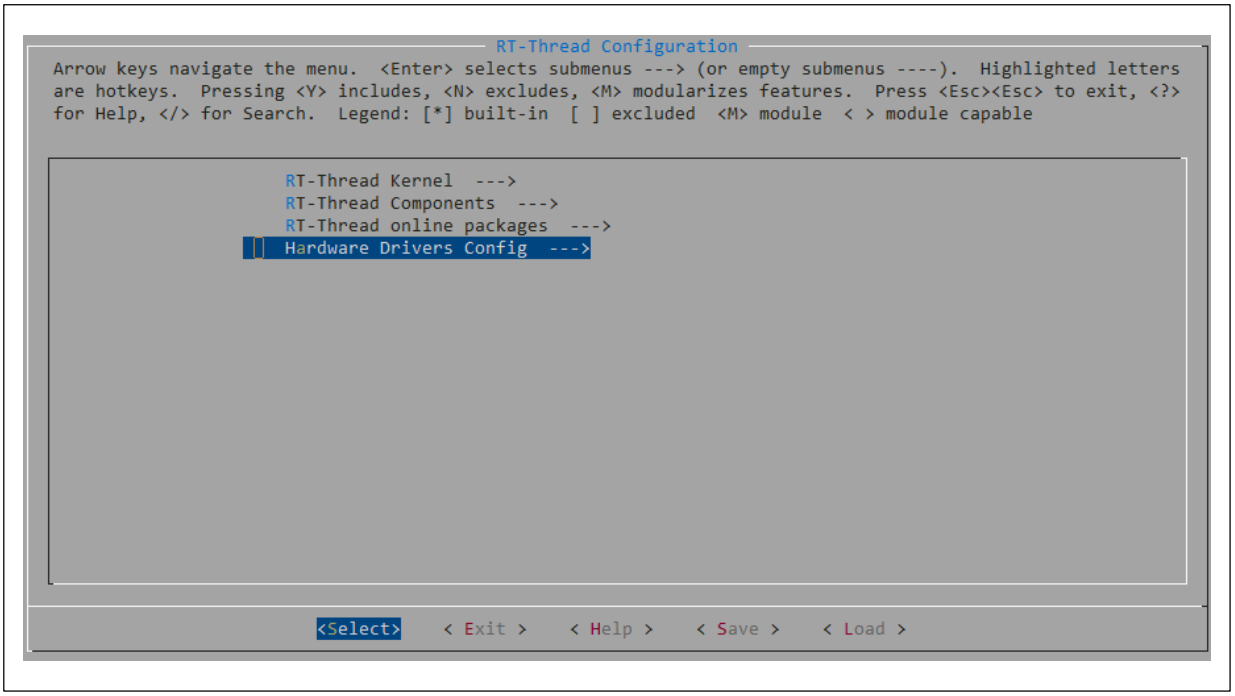
截取一部分内容，参照此对Kconfig在RT-Thread中所使用到的语法规则进行简要的说明。

```
menu "On-chip Peripheral Drivers"
    config BSP_USING_GPIO
        bool "Enable GPIO"
        select RT_USING_PIN
        default y
```

1. menu "On-chip Peripheral Drivers"：会在menuconfig运行后有一项"On-chip Peripheral Drivers"的菜单显示。
2. config BSP_USING_GPIO：对BSP_USING_GPIO宏定义进行配置。default y表示默认将BSP_USING_GPIO这个宏定义配置打开。
3. bool "Enable GPIO"：此处的配置是bool型（开关型），显示内容为"Enable GPIO"
4. select RT_USING_PIN：在配置开发了BSP_USING_GPIO的同时自动将RT_USING_PIN宏定义也打开。

将这三部分Kconfig都部署完毕后，可以尝试打开ENV工具并切换路径到board BSP的根目录下运行menuconfig来进行使用（注：ENV工具的详细使用方法请参考《Env_User_Manual_zh.pdf》）。在编写的Kconfig配置文件无语法或其他问题的情况下，运行menuconfig后的效果如下图所示。

图 4. menuconfig 运行效果



成功运行后可以看到，board文件夹下Kconfig的配置内容menu "Hardware Drivers Config"已经正确显示出来，按menuconfig操作方式可以进一步对此菜单栏进行选择和配置。

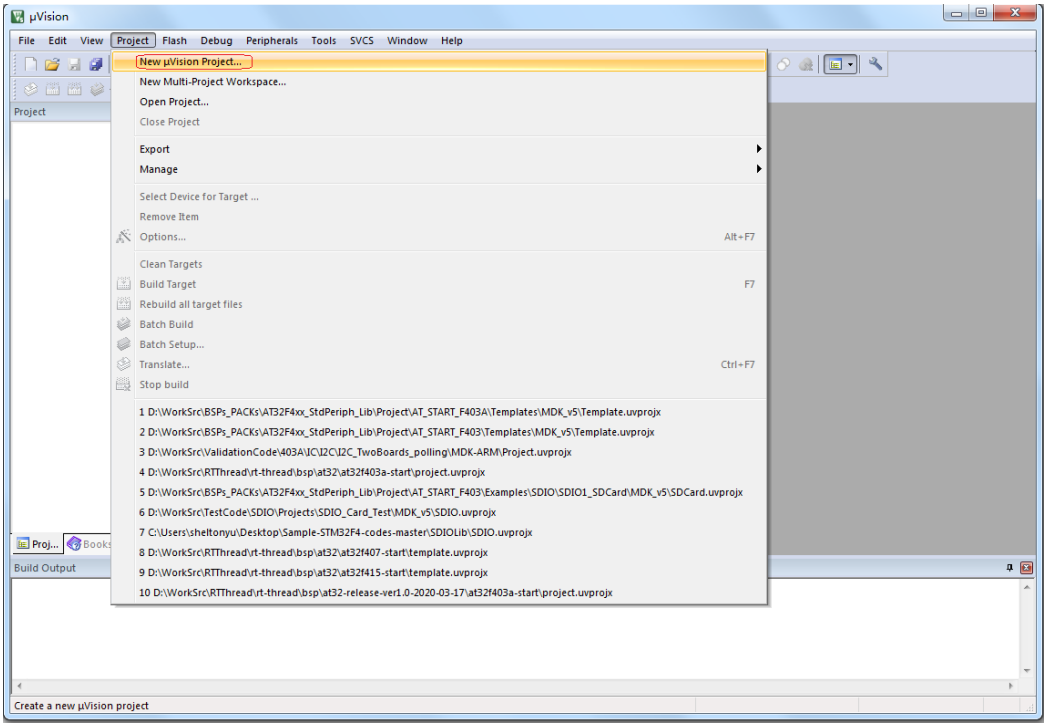
2.2.4 模板工程设置

此处的模板工程文件主要是用来配置keil_v5、keil_v4、IAR的项目工程，设置它的目的是为了在此放一份只进行了简单配置的初始文件，以方便后续menuconfig裁剪配置完成后，按裁剪的结果和文件依赖等信息来对应修改工程文件，生成后的工程文件便于直接用IDE工具打开就可以进行开发和调试，省去了繁琐的工程配置步骤。

模板工程文件的命名规则必须以template来命名，通过ENV工具配置后最终生成以project命名的工程文件。下面以keil_v5来进行创建模板工程来进行示例。

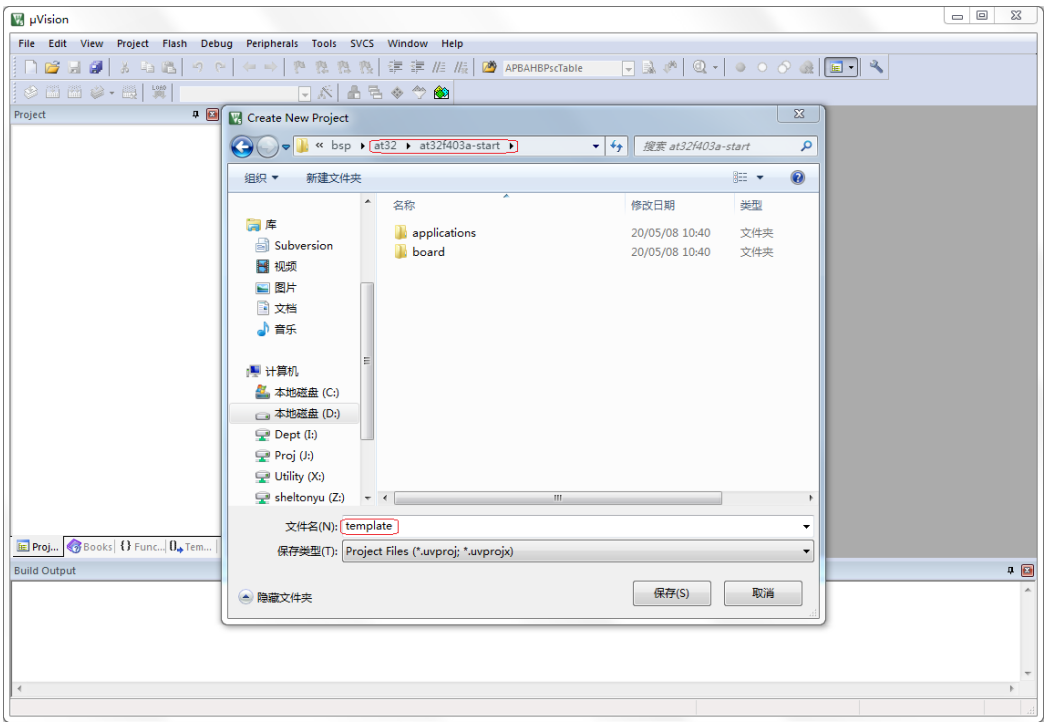
打开keil_v5工具，点击Project → New uVision Project…

图 5. 工程创建



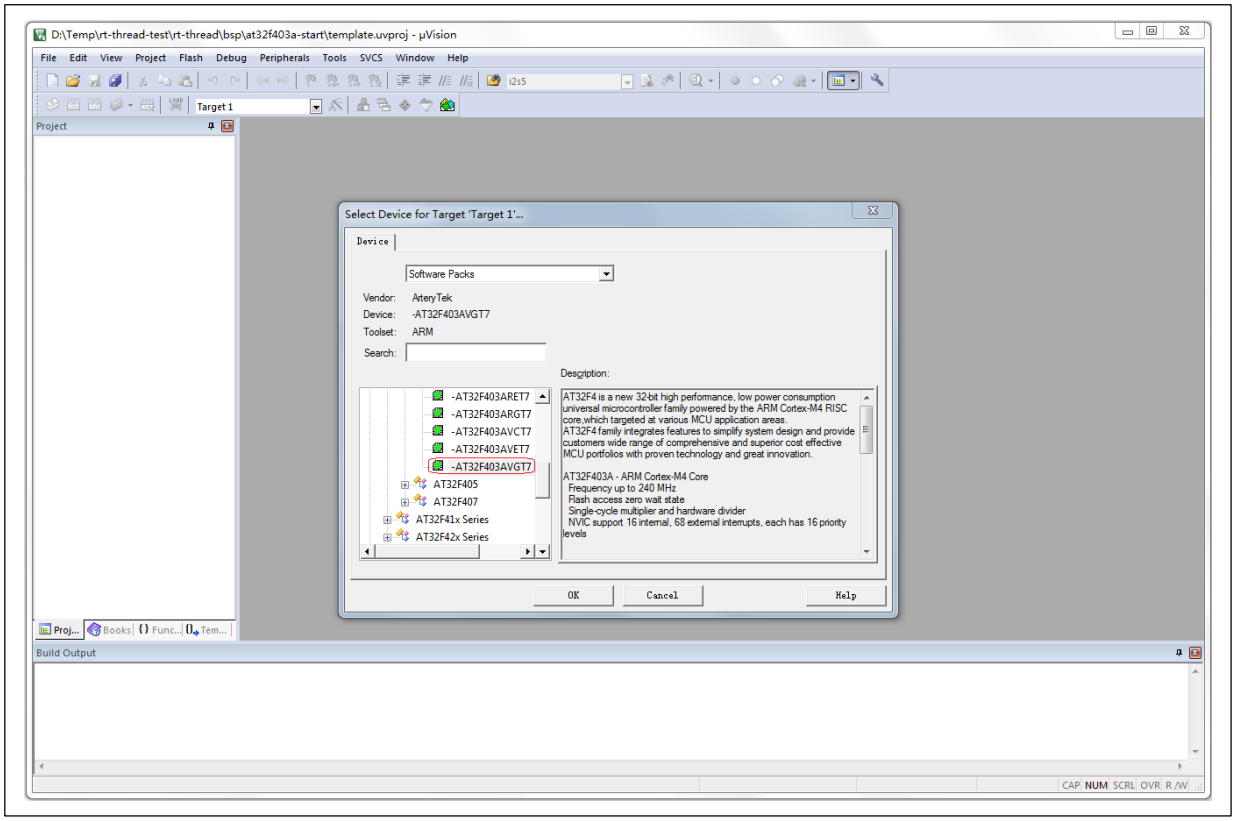
选择到需要创建模板工程的目录，此处示例是在新建的rt-thread/bsp/at32/at32f403a-start目录创建模板工程，工程命名为template。

图 6. 配置工程名



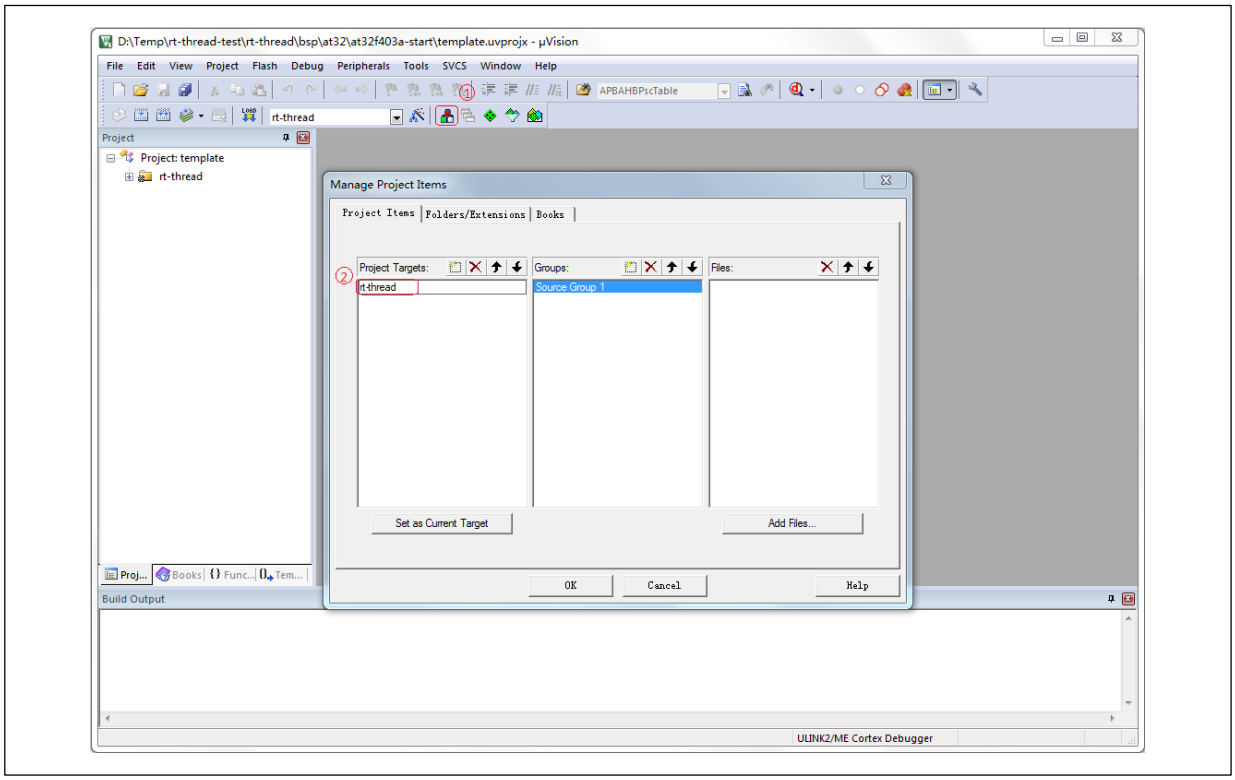
工程创建后选择芯片Device，示例选择的是AT32F403AVGT7型号（注：应先安装雅特力支持的Pack安装包）。

图 7. 选择 Device



最后点击OK，跳转的RTE配置可以不用管直接关闭就可以。接下来重命名Target，点击“Manage Project Items”，双击Target1可改为你期望的target名，示例Target改名rt-thread的图示步骤如下

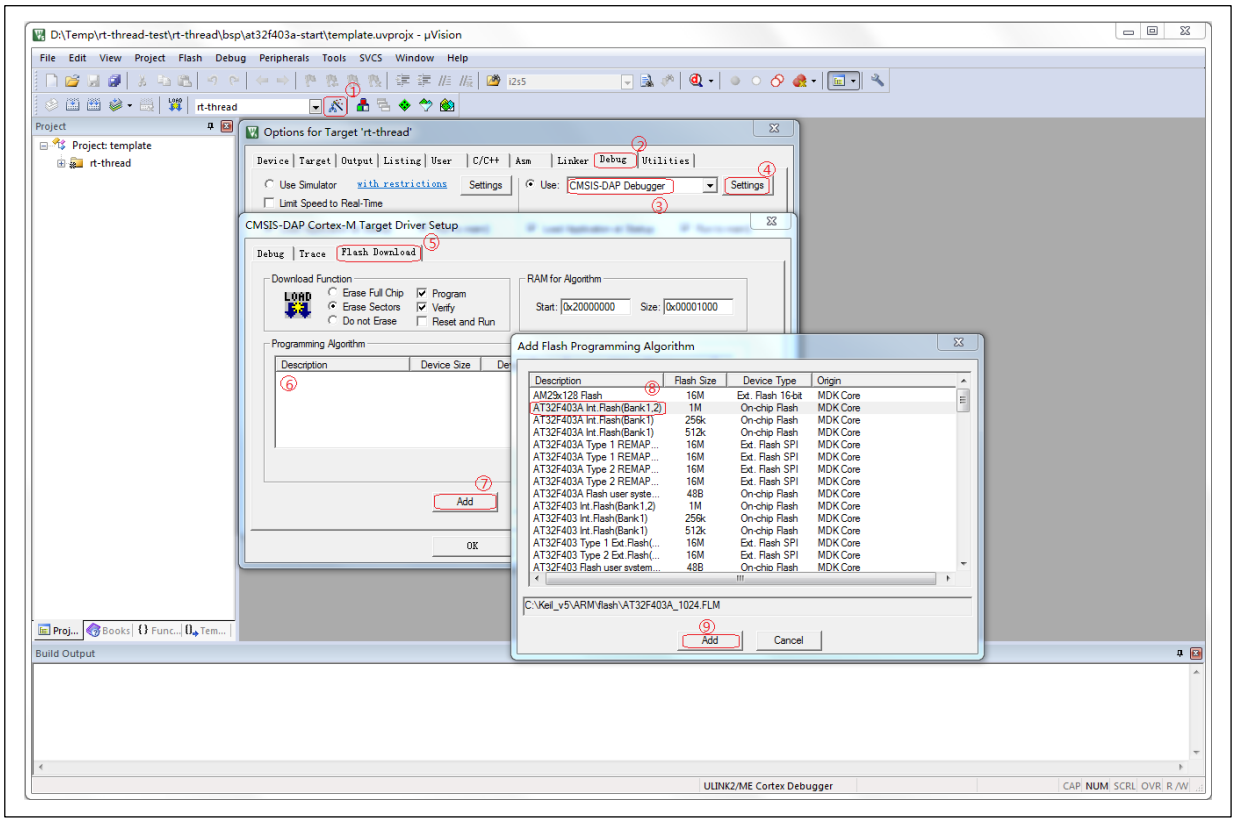
图 8. 重命名 Target



调试接口和算法文件的配置。可按如下步骤进行，①点击魔术棒“Option for Target…” → ②Debug选

项卡 → ③在下拉框中选择CMSIS-DAP Debugger → ④Settings → ⑤Flash Download选项卡 → ⑥查看算法文件是否已配置 → ⑦没有配置点击Add → ⑧选择AT32F403A Int.Flash(Bank1,2) → ⑨点击Add。步骤位置如下图

图 9. DAP Debugger 配置

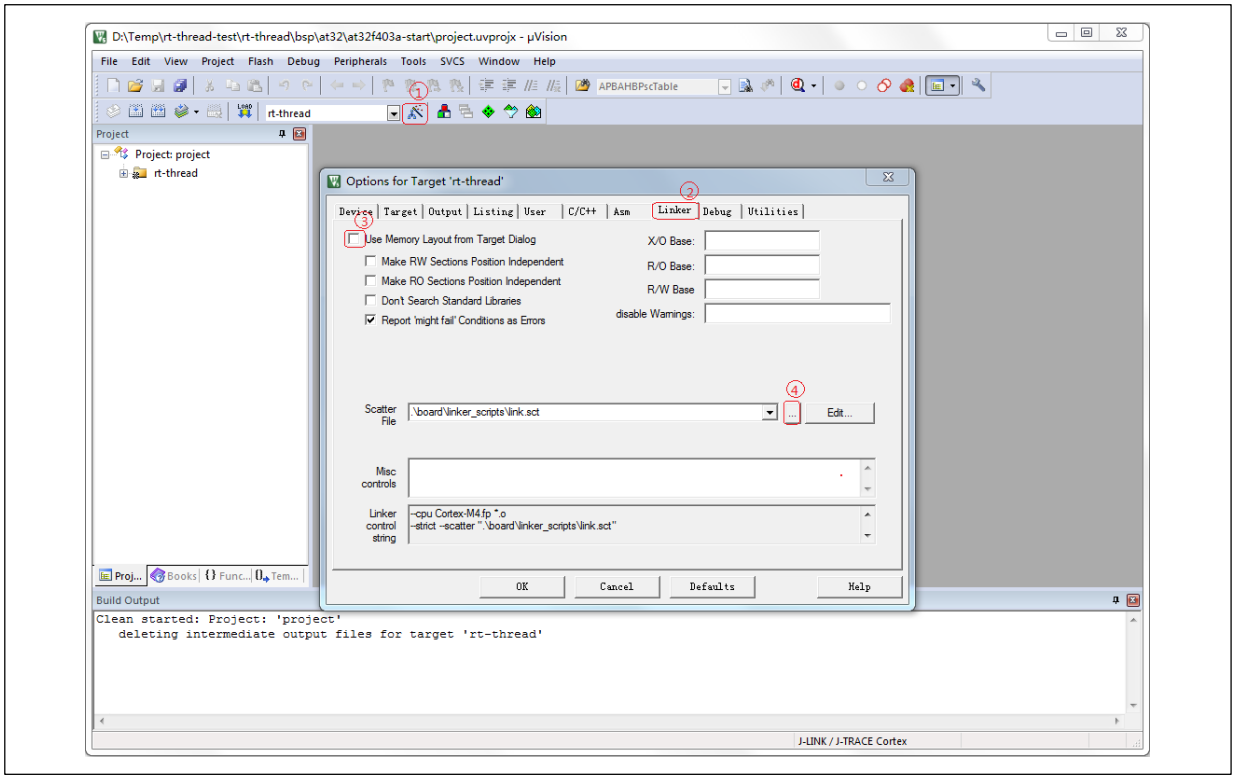


配置链接文件。首先应创建keil下的链接文件，新建board\linker_scripts\link.sct文件，并修改内容为

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************
LR_IROM1 0x08000000 0x00100000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00100000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x20000000 0x00018000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

当*.sct文件修改好后，可按如下步骤进行，①点击魔术棒“Option for Target…” → ②选择Linker栏 → ③取消复选框的勾选→ ④点击按钮对新增的*.sct进行选择。步骤位置如下图

图 10. 链接文件配置

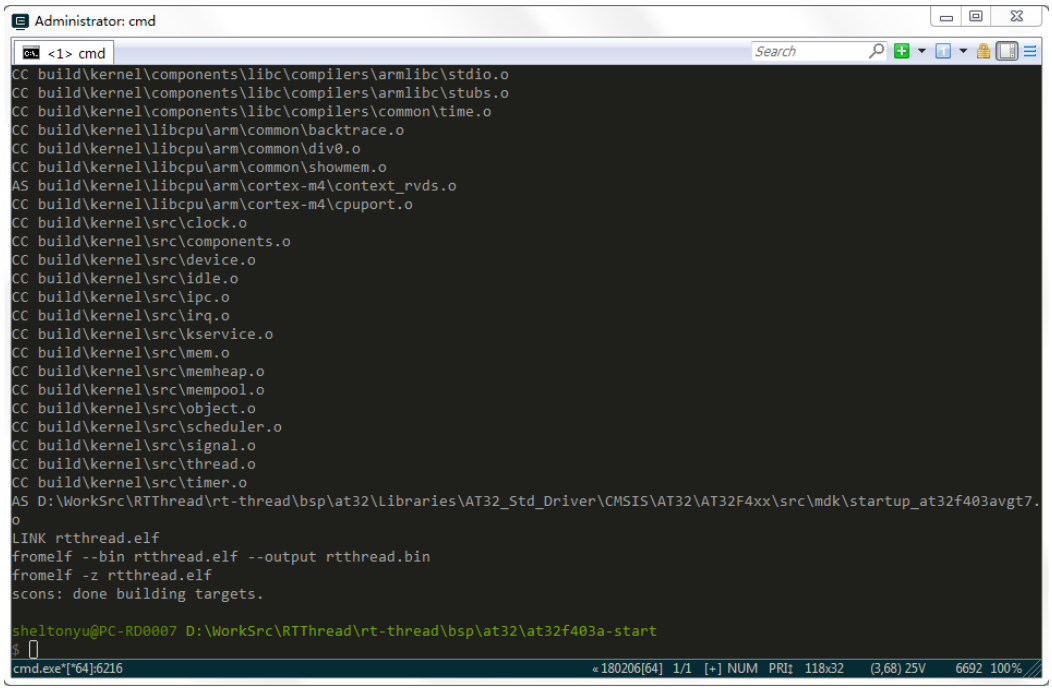


在裁剪配置完成后生成项目工程，可在ENV中输入命令可直接按配置来处理文件依赖后生成相应IDE开发环境的工程文件。常用命令如下：

```
scons --target=mdk5
scons --target=mdk4
scons --target=iar
scons --target=gcc
```

以生成mdk5工程为例，输入命令后运行结果如下图所示：

图 11. 生成工程文件



可以看到工程生成工具会去处理裁剪配置项所对应的依赖文件等。运行成功后会在Board BSP包根目录下生成一份名为project.uvprojx的keil5工程文件。

2.2.5 添加 AT32 外设库

在前文我们已对SConscript脚本文件的编写格式和语法等有了一定的了解。接下来就讲解如何使用SConscript文件来添加AT32的外设库。首先我们需要对整个BSP的文件和目录结构进行梳理，有了明确的目录结构后才好进行SConscript文件的编写。目录结构的排布方式可参考rt-thread/bsp/at32。下图示意主要进行AT32外设库部分的文件结构进行梳理，其他目录文件处理方式与此类似。结构如下：

图 12. 外设库文件结构

```
at32
 |---at32f403a-start
 |---libraries
     |---f403a_407
         |---rt_drivers
         |---firmware
             |---cmsis
             |    |---cm4
             |        |---core_support
             |        |    |---arm_common_tables.h
             |        |    |---arm_const_structs.h
             |        |    |---arm_math.h
             |        |    |---...
             |        |---device_support
             |             |---startup
             |             |    |---gcc
             |             |    |    |---startup_at32f403a_407.s
             |             |    |
             |             |    |---iar
             |             |    |    |---startup_at32f403a_407.s
             |             |    |
             |             |    |---mdk
             |             |    |    |---startup_at32f403a_407.s
             |             |
             |             |---at32f403a_407.h
             |             |---system_at32f403a_407.h
             |             |---system_at32f403a_407.c
             |
             |---drivers
             |    |---inc
             |    |    |---at32f403a_407_acc.h
             |    |    |---at32f403a_407_adc.h
             |    |    |---...
             |    |
             |    |---src
             |        |---at32f403a_407_acc.c
             |        |---at32f403a_407_adc.c
             |        |---...
             |
             |---SConscript  // 添加外设库的SConscript
     |---f435_437
     |---...
```

由上可以看出外设库里主要包含了cmsis和drivers两部分。cmsis是提供一些架构和芯片相关的头文件还有启动文件，drivers包含了片上各外设的驱动源文件和头文件。所以编写SConscript就主要完成添加这几部分的内容。此处SConscript的脚本内容可参考如下：

```
import rtconfig
from building import *
# get current directory
cwd = GetCurrentDir()
# The set of source files associated with this SConscript file.
```

```
src = Split("""
cmsis/cm4/device_support/system_at32f403a_407.c
drivers/src/at32f403a_407_acc.c
drivers/src/at32f403a_407_adc.c
drivers/src/at32f403a_407_bpr.c
drivers/src/at32f403a_407_can.c
drivers/src/at32f403a_407_crc.c
drivers/src/at32f403a_407_crm.c
drivers/src/at32f403a_407_dac.c
drivers/src/at32f403a_407_debug.c
drivers/src/at32f403a_407_dma.c
drivers/src/at32f403a_407_emac.c
drivers/src/at32f403a_407_exint.c
drivers/src/at32f403a_407_flash.c
drivers/src/at32f403a_407_gpio.c
drivers/src/at32f403a_407_i2c.c
drivers/src/at32f403a_407_misc.c
drivers/src/at32f403a_407_pwc.c
drivers/src/at32f403a_407_rtc.c
drivers/src/at32f403a_407_sdio.c
drivers/src/at32f403a_407_spi.c
drivers/src/at32f403a_407_tmr.c
drivers/src/at32f403a_407_usart.c
drivers/src/at32f403a_407_usb.c
drivers/src/at32f403a_407_wdt.c
drivers/src/at32f403a_407_wwdt.c
drivers/src/at32f403a_407_xmc.c
""")
path = [
    cwd + '/cmsis/cm4/device_support',
    cwd + '/cmsis/cm4/core_support',
    cwd + '/drivers/inc',]
CPPDEFINES = ['USE_STDPERIPH_DRIVER']
group = DefineGroup('Libraries', src, depend = [''], CPPPATH = path, CPPDEFINES =
CPPDEFINES)
Return('group')
```

cwd = GetCurrentDir()：获取当前路径。

src = Split()：指将片上各外设的驱动源码文件添加到工程，推荐是添加里面的所有驱动文件，且路径是按与当前SConscript的相对路径来索引。

path = [cwd + '…']：指将所需的头文件路径都添加到工程中，方便编译环境对头文件的索引。

CPPDEFINES = ['USE_STDPERIPH_DRIVER']：在工程中声明一个宏定义。

group = DefineGroup("Libraries', src, depend = [''], CPPPATH = path, CPPDEFINES = CPPDEFINES)：最后把所有内容都定义为一个组来进行管理。

因为在当前示例的目录结构中，设备驱动和外设驱动的SConscript不能被bsp/at32/at32f403a-start/Sconscript这顶层脚本搜索到，因此需要在SConstruct中单独添加，内容如下：

```
# include libraries
objs.extend(SConscript(os.path.join(libraries_path_prefix, at32_library, 'SConscript')))
```

以上就是添加AT32外设库的内容，其中的路径等设置由实际工程的目录结构而确定的，但方式与此相同。从之前的目录文件结构分析来看，此SConscript还缺少对启动文件（startup_at32f4xxxx.s）和时钟配置文件(system_at32f4xx.c)的添加。在这个bsp包中这两部分的添加内容放到了rt-thread/bsp/at32/at32f403a-start/board/SConscript中。

最后还需要添加一份board.c的文件，可参考rt-thread\bsp\at32\at32f403a-start\board\src\board.c。其主要内容为实现系统时钟的配置。

3 快速修改

目前AT32 RT-Thread已基于Arterytek AT-START-F403A V1.0开发板进行了BSP包的开发并上传到了rt-thread github，但包中芯片型号和硬件相关的配置信息等都已固定，对于需要更改芯片型号的需求，接下来会介绍如何进行型号修改的方法以供参考。此示例基于rt-thread\bsp\at32\at32f403a-start来进行介绍。

3.1 型号修改

芯片型号修改主要涉及工程配置和启动文件等。首先需要修改的是template. uvprojx中的Device的选择和算法文件的配置。这部分的具体操作流程和步骤前文已进行了详细介绍。芯片宏定义部分需要修改rt-thread\bsp\at32\at32f403a-start\board\SConscript

```
CPPDEFINES = ['AT32F403AVGT7']
```

加粗部分都应该修改为实际用到的芯片型号，最后进行工程内芯片宏定义配置。再一个就是rt-thread\bsp\at32\at32f403a-start\board\Kconfig内的芯片型号配置

```
config SOC_AT32F403AVGT7
    bool
    select SOC_SERIES_AT32F403A
    select RT_USING_COMPONENTS_INIT
    select RT_USING_USER_MAIN
    default y
```

此处也需要对应的修改为实际芯片型号。

3.2 参数修改

芯片参数修改需要根据所采用芯片型号的实际情况来进行修改，主要包含flash大小、内存大小、主频和片上外设等。

内存大小主要修改rt-thread\bsp\at32\at32f403a-start\board\inc\board.h文件

```
#define AT32_SRAM_SIZE      96
#define AT32_SRAM_END       (0x20000000 + AT32_SRAM_SIZE * 1024)
```

需要将AT32_SRAM_SIZE宏定义按实际芯片SRAM size来进行修改，此处示例为96KB。

Flash大小需要修改链接文件。rt-thread\bsp\at32\at32f403a-start\board\linker_scripts目录下存放了三份链接文件， *.sct/*.icf/*lds分别对应于keil/iar/gcc，具体修改方法自行参考。主频的修改请参考rt-thread\bsp\at32\at32f403a-start\board\src\board.c，修改的方式可参考对应的Start_Guid文档，如：《AN0082_AT32F403A_407_CRM_Start_Guide》。

外设pin脚的配置可修改rt-thread\bsp\at32\at32f403a-start\board\src\at32_msp.c。

4 驱动编写

RT-Thread操作系统为各外设驱动规范了统一的调用接口和框架。所以驱动开发需要严格遵守RT-Thread的驱动框架来实现对硬件底层的驱动。

4.1 驱动框架

RT-Thread中各外设驱动的实现框架在rt-thread\components\drivers\include\drivers目录下的各文件中进行了声明，以usart为例来进行说明，打开目录下的serial.h文件，其中定义了上层应用中对串口初始化和配置的宏和函数，通常的驱动框架接口都是采用rt_*_ops为名的结构体来进行声明，特殊的驱动框架接口就特殊处理。此结构体中的成员变量和成员函数就直接对应着硬件底层，串口部分的详细内容如下：

```
/**
 * uart operators
 */
struct rt_uart_ops
{
    rt_err_t (*configure)(struct rt_serial_device *serial, struct serial_configure *cfg);
    rt_err_t (*control)(struct rt_serial_device *serial, int cmd, void *arg);
    int (*putc)(struct rt_serial_device *serial, char c);
    int (*getc)(struct rt_serial_device *serial);
    rt_size_t (*dma_transmit)(struct rt_serial_device *serial, rt_uint8_t *buf, rt_size_t size, int direction);
};
```

以上内容可知串口的主要成员函数指针为configure、control和数据收发相关的函数，在我们进行底层驱动编写时就主要是对应的实现这几个函数的功能。

4.2 驱动实现

驱动的实现就是结合硬件外设的使用方法来对应实现规定的框架函数，具体的函数实现流程这里就不做详细的介绍，需要结合硬件来编写。当所需要完善的驱动内容都已经实现以后，将各函数初始化到设备结构体的驱动操作方法中，然后再注册到系统驱动链表，待上层应用进行调用。AT32 usart驱动可参考rt-thread\bsp\at32\Libraries\rt_drivers\drv_usart.c，当所需的函数都已经实现完成后，其核心的初始化操作接口和注册驱动内容类似如下：

```
// 初始化已实现的函数接口到操作接口结构体
static const struct rt_uart_ops at32_usart_ops = {
        at32_configure,
        at32_control,
        at32_putc,
        at32_getc,
        RT_NULL };
```

下面就以发送一个字符的函数举例，来说明如何调用AT32外设库来驱动硬件进行串口发送的。

```
static int at32_putc(struct rt_serial_device *serial, char ch) {
    struct at32_usart *usart;
    RT_ASSERT(serial != RT_NULL);
    usart = (struct at32_usart *) serial->parent.user_data; // 取出统一设备描述符中的设备对象
    RT_ASSERT(usart != RT_NULL);
    usart_data_transmit(usart->usart_x, (uint8_t)ch); // 调用AT32外设库中的串口发送函数usart_data_transmit来驱动硬件实现数据的发送
    while (usart_flag_get(usart->usart_x, USART_TDC_FLAG) == RESET); // 调用AT32外设库中的函数进行发送状态的处理
    return 1;
}
```

当以上都实现完成后，在串口驱动初始化函数中需要将驱动的操作接口结构体赋值到设备对象结构体中，并调用系统注册函数将设备对象注册到系统中。此初始化函数会在系统初始化流程中进行调用，完成串口的注册和初始化。初始化函数示意如下：

```
// usart驱动初始化函数，在rt-thread的系统初始化流程中已经进行了初始化调用
int rt_hw_usart_init(void) {
    rt_size_t obj_num; int index;
    obj_num = sizeof(usart_config) / sizeof(struct at32_usart);
    struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT;
    rt_err_t result = 0;
    for (index = 0; index < obj_num; index++) {
        usart_config[index].serial.ops = &at32_usart_ops; // 操作接口结构体赋值到设备对象结构体中
        usart_config[index].serial.config = config;
        /* register uart device */
        result = rt_hw_serial_register(&usart_config[index].serial,
                    usart_config[index].name,
                    RT_DEVICE_FLAG_RDWR |
                    RT_DEVICE_FLAG_INT_RX |
                    RT_DEVICE_FLAG_INT_TX,
                    &usart_config[index]);
        RT_ASSERT(result == RT_EOK); // 串口设备注册到系统驱动链表
    }
    return result;
}
```

5 版本历史

表 1. 文档版本历史

日期	版本	变更
2022.04.25	2.0.0	最初版本

重要通知 – 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独力负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。