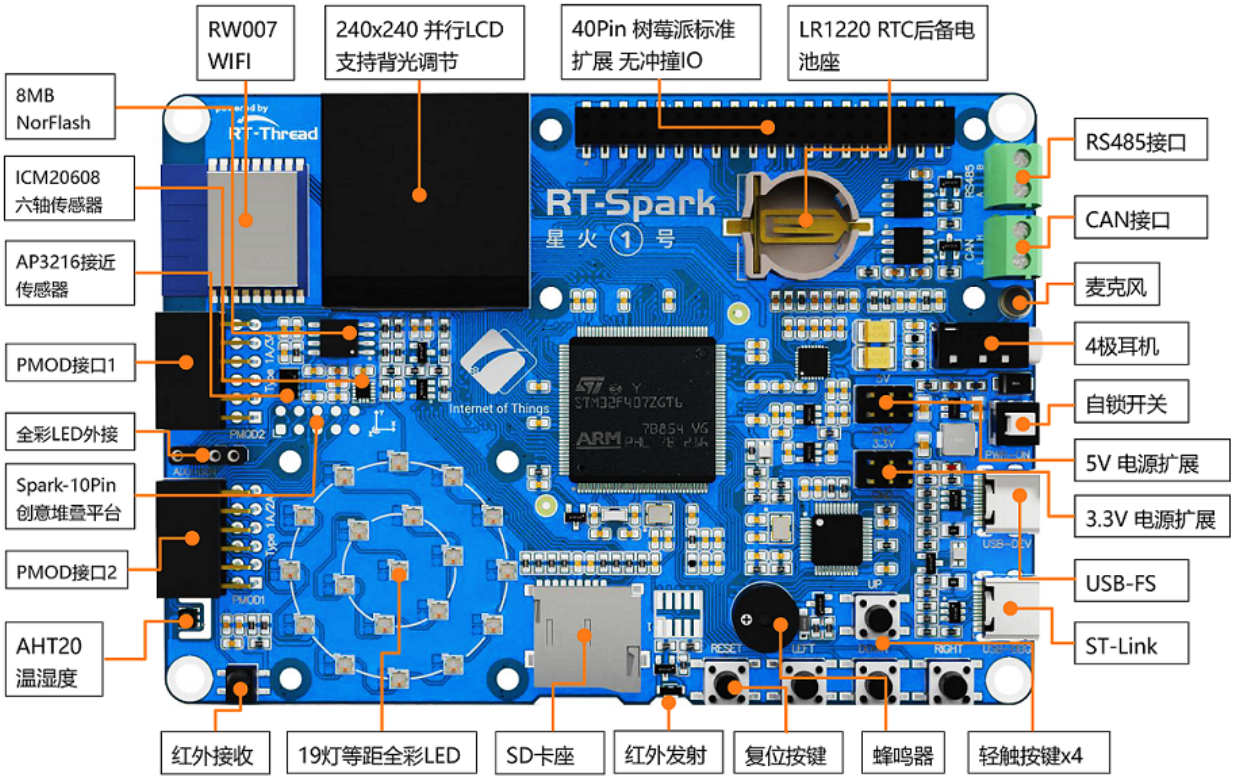
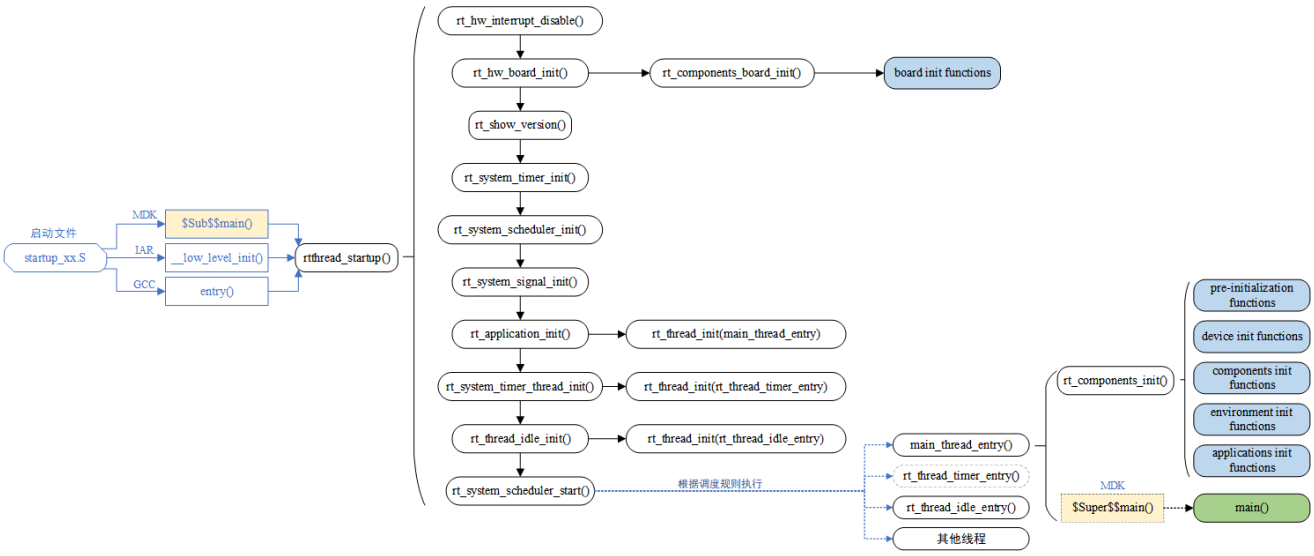
RT-Thread BSP移植思路分享

> 首先很抱歉各位，我写文章都是随心情去写，不会特别的规范，有问题最好是直接联系我！！不要死啃文档。



1 启动流程

首先来看一下rtthread的启动流程：



通过这张图，其实我们可以看到最核心的一个东西其实是**rtthread_startup**函数。这里我们必须要搞清两个点：

- 谁来调用这个函数？？
- 这个函数需要做什么？？

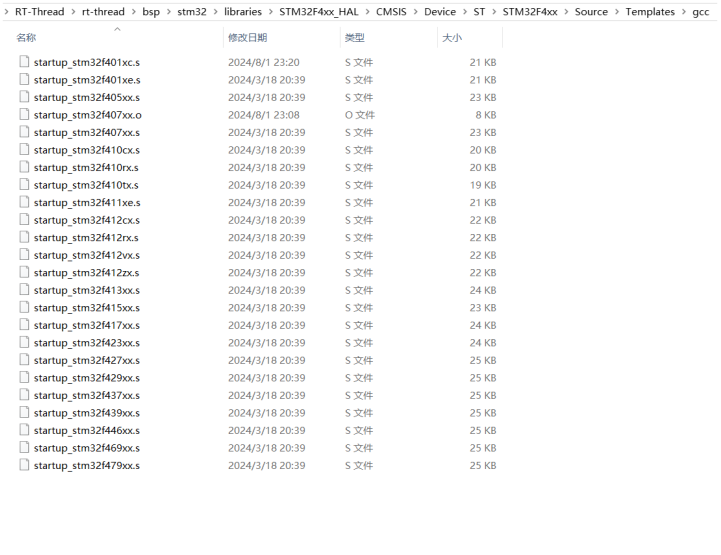
1.1 谁来调用

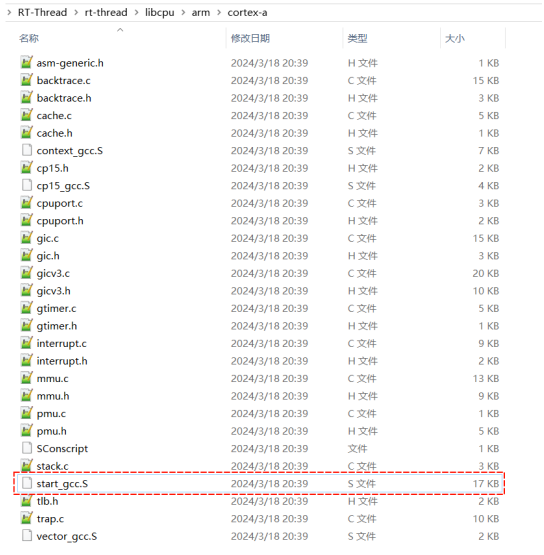
首先大家都知道一个点，那就是这个函数的调用肯定来自于一个.s汇编文件，毕竟启动文件肯定是.s文件，为我们准备好c语言的运行环境。

所以我们在移植的时候，需要关心一下启动文件，也就是上图的startup_xx.S，由于本次咱们移植的是一个stm32的板子，大家都知道，stm32基本每一个系列都会有一个启动文件，主要就是启动文件会定义一些中断入口，而每个型号基本都是不同的，所以stm32可能会有很多个，也就没办法做到统一了，像这种无法做到统一的东西，就只能咱们自行进行移植了。

> 如果有小伙伴接触过*A*核的一些芯片，中断类型基本是固定的：
>
> Cortex-A7有8个异常中断，如下表所示：
>
> | 向量地址 | 中断类型 | 中断模式 |
> | --- | --- | --- |
> | 0X00 | 复位中断(Rest) | 特权模式(SVC) |
> | 0X04 | 未定义指令中断(Undefined Instruction) | 未定义指令中止模式(Undef) |
> | 0X08 | 软中断(Software Interrupt,SWI) | 特权模式(SVC) |
> | 0X0C | 指令预取中止中断(Prefetch Abort) | 中止模式 |
> | 0X10 | 数据访问中止中断(Data Abort) | 中止模式 |
> | 0X14 | 未使用(Not Used) | 未使用 |
> | 0X18 | IRQ 中断(IRQ Interrupt) | 外部中断模式(IRQ) |
> | 0X1C | FIQ 中断(FIQ Interrupt) | 快速中断模式(FIQ) |
>
> 这是我在网上截的一张图，大家可以看到，基本这些中断向量是定死的，所以像这种，启动文件其实是可以做到统一的，所以这种*rtthread*会进行接管。*A*核的中断处理大家可自行在网上去查阅资料，这里不详细介绍。

下图就是stm32的启动文件核cortex-a核的启动文件！！，大家自己体会。。。。





所以，对于像stm32这种芯片，咱们需要移植他的启动文件(我现在的思路是建立在rtthread现在没有支持stm32的思路上进行讲解的，所以会从最根本的东西讲起走)，所以咱们可以自己去找一个stm32f407zg的启动文件就可以了，里面还是要稍作修改，比如最起码的要让他跳转到entry(gcc环境)。

```c
#ifdef RT_USING_USER_MAIN

void rt_application_init(void);
void rt_hw_board_init(void);
int rtthread_startup(void);

#ifdef __ARMCC_VERSION...
#elif defined(__GNUC__)
/* Add -eentry to arm-none-eabi-gcc argument */
int entry(void)
{
    rtthread_startup();
    return 0;
}
#endif

#ifndef RT_USING_HEAP
/* if there is not enable heap, we should use static thread and stack. */
t_align(RT_ALIGN_SIZE)
static rt_uint8_t main_thread_stack[RT_MAIN_THREAD_STACK_SIZE];
struct rt_thread main_thread;
```

注意，只有打开这个宏才会走官方的流程，不打开这个宏大家可自行操作！！

```
    adds r2, r2, #4

LoopFillZerobss:
  cmp r2, r4
  bcc FillZerobss

/* Call the clock system initialization function.*/
  bl  SystemInit
/* Call static constructors */
    bl __libc_init_array
/* Call the application's entry point.*/
  bl  main
  bx  lr
.size  Reset_Handler, .-Reset_Handler
```

bl entry

> 当然里面还是有一些可以修改的地方，比如咱们打算用rtt的内存堆管理，那可以把这个启动文件里面的堆大小配置为0，节省空间。

1.2 做了什么

做了什么咱们就可以来看一下这个rtthread_startup函数了！！

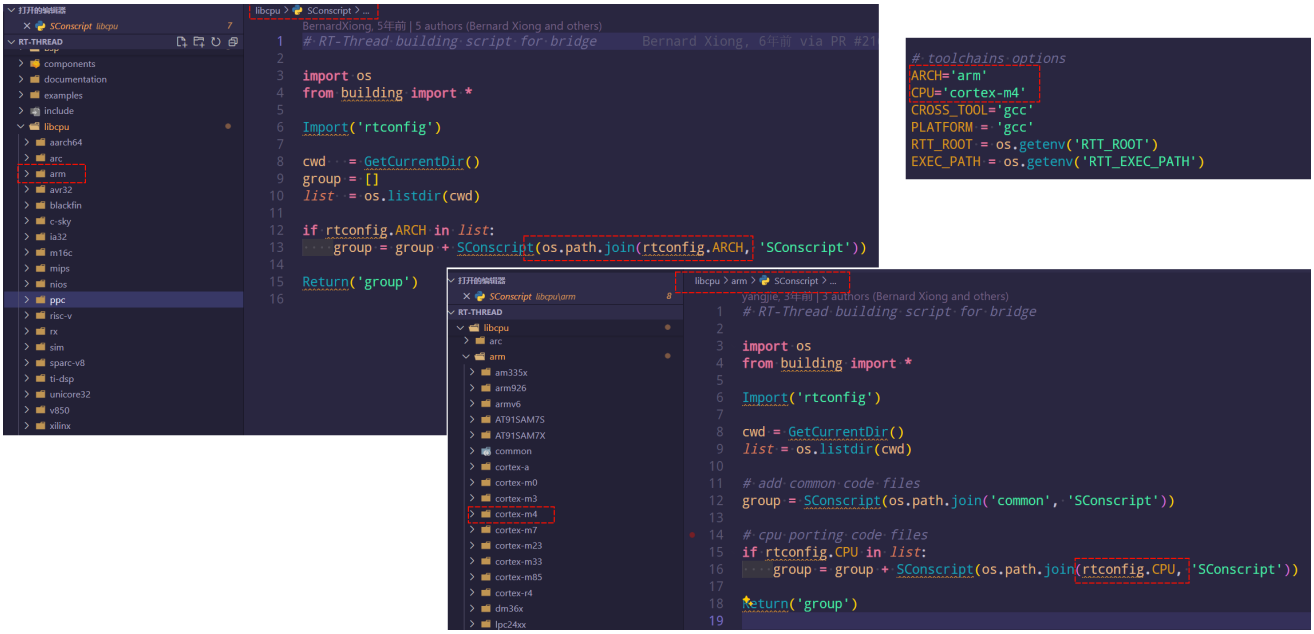
它调用的函数如下：

函数	芯片型号强相关	描述
rt_hw_interrupt_disable	是	跟芯片架构相关(Cortex-M3、Cortex-M4、……)
rt_hw_board_init	是	跟芯片系列相关(STM32F1xx、STM32F2xx、……..)
rt_show_version	否	
rt_system_timer_init	否	
rt_system_scheduler_init	否	
rt_system_signal_init	否	
rt_application_init	否	
rt_system_timer_thread_init	否	
rt_thread_idle_init	否	
rt_system_scheduler_start	是	跟芯片架构相关(Cortex-M3、Cortex-M4、……)

根据上面表格，要想保证rtthread的正确运行，我们只需要要完成rt_hw_interrupt_disable、rt_hw_board_init、rt_system_scheduler_start这个三个部分即可。

但是！！**rt_hw_interrupt_disable**和**rt_system_scheduler_start**这种跟芯片架构相关的，rtthread帮我们做好了，也就是项目目录下的**libcpu**文件夹里面的东西！架构的东西基本是通用的，对于stm32，大家想要了解的话，可以看一下cm3和cm4的权威指南。

我们只需要保证这几个文件被编译即可，一般来说，这几个文件就是由我们的rtconfig.py文件控制的：



一般来说这样就可以了。

```
CC build\kernel\libcpu\arm\common\atomic_arm.o
CC build\kernel\libcpu\arm\common\div0.o
CC build\kernel\libcpu\arm\common\showmem.o
AS build\kernel\libcpu\arm\cortex-m4\context_gcc.o
CC build\kernel\libcpu\arm\cortex-m4\cpuport.o
```

所以，对于我们移植来说，其实是非常轻松的，只需要完成**rt_hw_board_init**函数即可，我们就以使用cortex-m4内核的板子来说，咱们国内GD32、HC32、MM32以及我们常用的STM32，他们可能存在一些其他IP设计的差异，所以rtthread也是不知道每一个厂商是如何设计的，这就需要那些芯片厂家，或者像我们这样的社区爱好者来帮助完成了。

要想保证一个板子能运行咱们的rtthread操作系统，其实我们需要完成的主要内容也不是很多，主要包括：

- 需要配置一个定时器为系统提供心跳时钟，一般来说是Systick滴答定时器。
- 配置一下时钟系统，保证系统的正常运行。
- 根据板子的Flash或者Ram的情况来配置系统堆的空间。
- 最基本的pin驱动和uart驱动可以使用，最重要的是uart驱动，至少我们需要看到系统的运行状态。

这里我梳理了几个比较重要的大方向，不同的板子可能还有一些细节。

1.3 总结一下

主要需要完成的事情：

- 移植并处理启动文件，对于a核可能不需要我们做。
- 想办法把咱们芯片架构的几个文件编译进去，这里面rtthread帮我们实现了一些基本的东西，比如调度、中断啥的。a核可能更丰富。
- 完成rt_hw_board_init函数。

- 保证操作系统运行的心跳正常。
- 保证MCU的时钟系统正常。
- 保证堆空间的正常，因为rtt里面用了很多类似rt_malloc去动态分配内存。
- 前期咱们保证uart驱动的正常工作吧，能为我们打印启动信息啥的。

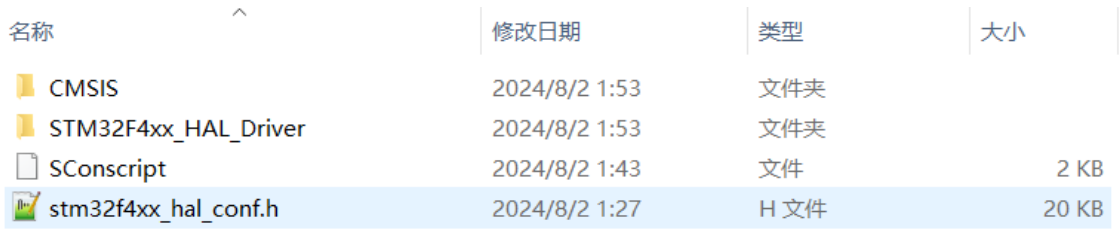
2 创建基本的文件

对于一个bsp我们需要一些必要的文件，这里我尽量做到约简介越好，不然复杂了容易吃不消。。

文件/文件夹	是否必须	描述
application/mian.c	是	mian主程序(线程)
application/SConscript	是	控制main.c的编译
board/startup_stm32f407xx.s	是	启动文件
board/board.c	是	主要为了完成rt_hw_board_init函数的实现
board/SConscript	是	控制board文件夹内的文件参与编译
board/linker_scripts/link.lds	是	控制程序的链接脚本
.config	否	menuconfig会生成
.gitignore	否	git相关
Kconfig	是	bsp的顶层Kconfig文件，menuconfig入口
rtconfig.h	否	rtthread会使用到的宏定义文件，会有脚本生成
rtconfig.py	是	编译选项的配置文件
SConscript	是	SCons工程管理文件
SConstruct	是	SCons工程管理文件

> 基本上，这样几个文件，咱们就可以移植成功了，最复杂的部分是***board.c***里面的部分。如果要以最简的文件来完成适配的话，这里我可能需要去把寄存器查死！！！！！！然后才能给你们写出驱动来，不是我不想写，我就给你们做个分享，把自己累死，我觉得完全不值！！所以我打算用*sdk*来完成驱动，还请各位谅解！！

在board文件夹下，创建一个stm32_sdk的子文件夹来放stm32的sdk。当然，有能力的，完全可以不用sdk，怼寄存器就行了(简而言之，我无能)。



名称	修改日期	类型	大小
CMSIS	2024/8/2 1:53	文件夹	
STM32F4xx_HAL_Driver	2024/8/2 1:53	文件夹	
SConscript	2024/8/2 1:43	文件	2 KB
stm32f4xx_hal_conf.h	2024/8/2 1:27	H 文件	20 KB

最后来看一下最终的目录树结构

```
|-- applications
|   |-- SConscript
|   `-- main.c
|-- board
|   |-- SConscript
|   |-- board.c
|   |-- board.h
|   |-- linker_scripts
|   |   `-- link.lds
|   |-- startup_stm32f407xx.s
|   `-- stm_sdk
|       |-- CMSIS
|       |-- SConscript
|       |-- STM32F4xx_HAL_Driver
|       `-- stm32f4xx_hal_conf.h
|-- Kconfig
|-- SConscript
|-- SConstruct
`-- rtconfig.py
```

创建好我们目录结构后，接下来就是重头戏了：

- rtthread构建环境搭建
- 实现rt_hw_board_init函数

3 实现RT-Thread构建环境

3.1 Kconfig编写

bsp路径下的Kconfig文件为menuconfig的根文件，我们需要在这里引入两个比较关键性的Kconfig文件：

- rtthread工程路径下的Kconfig，通过这个这个我们可以去配置rtthread的一些内核、组件等等。
- 软件包的Kconfig，方便使用rtthread的软件包系统。

另外，很多时候大家可能还会看见其他的source kconfig，其实这些都是移植用到的，比如stm32的那个libraries里面的Kconfig，和那个board的Kconfig，这些都是移植的时候用到的，不同的芯片可能这些都是不一样的。但是rtthread的已经形成了一个约定俗成的习惯。一般都会去包含这两个Kconfig。所以这里对这两个做一个解释：

- libraries里面的Kconfig主要是定义一些跟芯片架构相关的宏。
- board里面的Kconfig主要定义一些跟驱动配置相关的宏。

> 我们这次移植不会使用那么多的*Kconfig*，只有足够简介，大家才能深入理解！！！！

所以本次咱们的Kconfig很简单：

```
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"

config BOARD_NEED_CONFIG
    bool
    select RT_USING_USER_MAIN
    select ARCH_ARM_CORTEX_M4
    default y

source "$RTT_DIR/Kconfig"
source "$PKGS_DIR/Kconfig"
```

这里我们就用了**rtthread**工程和软件包的**Kconfig**，然后自己加了一个**BOARD_NEED_CONFIG**把一些咱们移植需要用到的宏配置进工程。

3.2 rtthread.py编写

```
import os

# toolchains options
ARCH='arm'
CPU='cortex-m4'
```

```
CROSS_TOOL='gcc'
PLATFORM = 'gcc'
RTT_ROOT = os.getenv('RTT_ROOT')
EXEC_PATH = os.getenv('RTT_EXEC_PATH')

# type: debug, release
BUILD = 'debug'

# toolchains flags
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 += ' -Os'

CXXFLAGS = CFLAGS

POST_ACTION = OBJCPY + ' -O ihex $TARGET rtthread.hex\n' + SIZE + ' $TARGET \n'
POST_ACTION += OBJCPY + ' -O binary $TARGET rtthread.bin\n' + SIZE + ' $TARGET \n'
```

这个文件一般就是定义咱们使用的一个架构平台，然后指定一下编译器路径，已经编译器信息和一堆编译标志。

主要分为两个部分：

- 定义架构和编译平台的一些信息

- 编译链接的一些FLAGS。

> 这个文件主要是跟架构和编译平台相关的，跟具体的*bsp*没有太大关系，所以！！！大家可以去*RTT*的*bsp*里面去找一个相同架构的来抄，一般来说不会出问题，但是建议大家去学习一下编译的这些*FLAG*的含义，这样遇到问题也好查！！

这里的POST_ACTION其实是可有可无的，不重要，编译完成会执行的一些指令，大家在项目开发过程了可以尝试多用用**POST_ACTION**，帮你执行个啥脚本之类的，还是很nice的！

这里还有一个**board/linker_scripts/link.lds**文件，这个也是比较重要的，主要是告诉咱们的固件的一个分布情况，一般来说也是找一个直接修改即可。比较重要的就是：

```
/* Program Entry, set to mark it as "used" and avoid gc */
MEMORY
{
    CODE (rx) : ORIGIN = 0x08000000, LENGTH = 1024k /* 1024KB flash */
    RAM1 (rw) : ORIGIN = 0x20000000, LENGTH =  128k /* 128K sram */
    RAM2 (rw) : ORIGIN = 0x10000000, LENGTH =   64k /* 64K sram */
    MCUlcdgrambysram (rw) : ORIGIN = 0x68000000, LENGTH = 1024k
}
ENTRY(Reset_Handler)
```

这些内存的描述，尽量修改跟自己的板子相对应。大家都知道rtt内部使用了很多的程序段，这些链接标本其实也可以指定咱们的每一个段放在什么地方。

```
/* SECTION: finsh, a C-Express shell */
#define RT_USING_FINSH
/* Using symbol table */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION
#define __fsymtab_start _alt_partition_
#define __fsymtab_end   _alt_partition_
#define __vsymtab_start _alt_partition_VSymTab_start
#define __vsymtab_end   _alt_partition_VSymTab_end
```

```
    . = ALIGN(4);
    __vsymtab_start = .;
    KEEP(*(VSymTab))
    __vsymtab_end = .;
```

> 链接脚本建议大家去学习一下，还是比较重要的，对咱们*rtthread*的固件分布都能有一个比较清晰的认识！！

3.3 SConstuct脚本编写

```
1  import os
2  import sys
3  import rtconfig
4
5  if os.getenv('RTT_ROOT'):
6      RTT_ROOT = os.getenv('RTT_ROOT')
7  else:
8      RTT_ROOT = os.path.normpath(os.getcwd() + '/../..')
9
10 sys.path = sys.path + [os.path.join(RTT_ROOT, 'tools')]
11 try:
12     from building import *
```

```
except:
    print('Cannot found RT-Thread root directory, please check RTT_ROOT')
    print(RTT_ROOT)
    exit(-1)

TARGET = 'rt-thread.' + 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',
    CXX = rtconfig.CXX, CXXFLAGS = rtconfig.CXXFLAGS,
    LINK = rtconfig.LINK, LINKFLAGS = rtconfig.LFLAGS)
env.PrependENVPath('PATH', rtconfig.EXEC_PATH)

Export('RTT_ROOT')
Export('rtconfig')

# prepare building environment
objs = PrepareBuilding(env, RTT_ROOT, has_libcpu=False)

# make a building
DoBuilding(TARGET, objs)
```

这个是我们scons命令的一个入口，很重要的！这个脚本一般来说都是几个关键部分：

- 重新定义一下RTT_ROOT，预防你的env没有这个环境变量。
- 添加RTT的tools(那一堆的python脚本)路径到系统路径，这样才能保证执行到这些脚本(有精力建议看一下这些脚本)。
- 定义咱们的scons的**Environment**，这个是非常重要的，咱们前面rtconfig.py的那堆东西就是在这儿用的。很重要！这个env也会贯穿咱们编译的始终。
- 导出我们的RTT_ROOT和rtconfig.py里面定义的各个变量到python的各个脚本里面去使用，这一步也是很重要的。
- 执行PrepareBuilding，大家不用关心这个函数做了什么，但是！！我这里给大家列一些主要做的事情：
    - 根据编译平台，继续处理一下env的一些东西，感兴趣的可以去细看。
    - 处理一些option，scons里面的概念，比如scons --menuconfig， scons --target=vsc等，但是大家可以看见这个代码里面并没有处理--target的option，**why**？？？留个问题。
    - 执行一些子的SConscript脚本，之前经常有朋友问我，rtt工程里面的src、libcpu路径下的SConscript脚本是在哪儿调用的呢？？其实就是这里。
- 上面PrepareBuilding里面会执行一些SConscript脚本，那么对于咱们自定义的一些SConscipt脚本，就得咱们自己引入了。
- 最后执行DoBuilding，这个函数也不用关心，我还是把主要做的事情列出来：

- 继续执行一些option，这里其实就会处理--target的option了。解释：？？？
- 编译源码。

> 当然我上面的总结也不是特别详细，但是基本提到了比较重要的点，大家感兴趣可以自己深入分析一下上面说的那两个函数。

下面的东西就不用写了，上面的知识点大家好好理解一下！！下面这些东西我认为比较简单。课程带大家做一下即可。

3.4 其他文件

c文件如何写，这个应该不用我讲了，去抄就行。

SConscript文件咋写，估计你们夏令营教过，这里带大家走读一下。

> 还是那句话，有疑问？？可以问我、、

4 实现rt_hw_board_init函数

假设，大家已经搭建好一份很nice的编译构建环境，课程我会带大家去做的。

5 测试移植是否成功

log能正常输出，能调度，咱们就完成了一半了。

6 完善驱动，对接框架

7 跟官方BSP做对比

8 提出你们的问题

9 总结

10 多多指教！！