如何将 CoreMark 程序移植到 STM32 上

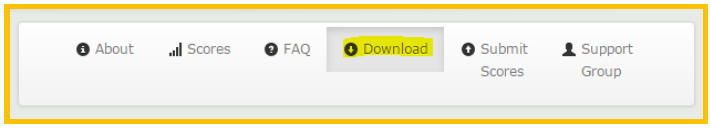
前言

CoreMark 是一项测试处理器性能的基准测试。代码使用 C 语言写成，包含：列举，数学矩阵操作和状态及 CRC 等运算法则。目前 CoreMark 已迅速成为测量与比较处理器性能的业界标准基准测试。CoreMark 的得分越高，意味着性能更高。在 CoreMark 的官网上可以大家可以看到各家处理器型号的 CoreMark 得分。也可以从 CoreMark 的官网上下载测试代码，亲自测一下自己手中的片子的性能。CoreMark 官网的连接地址：http://www.eembc.org/coremark/index.php。本文将一步步来介绍如何将下载的 CoreMark 测试代码移植到 STM32MCU 上进行测试。

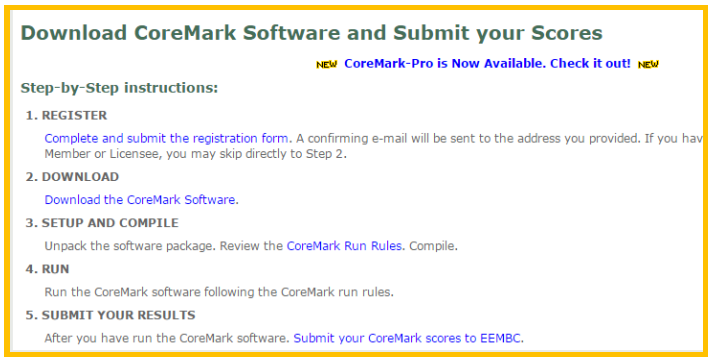
下载 CoreMark 测试代码

通过上文给出的链接进到 CoreMark 官网。

点击 Download,



根据页面的指导，先注册再下载测试代码

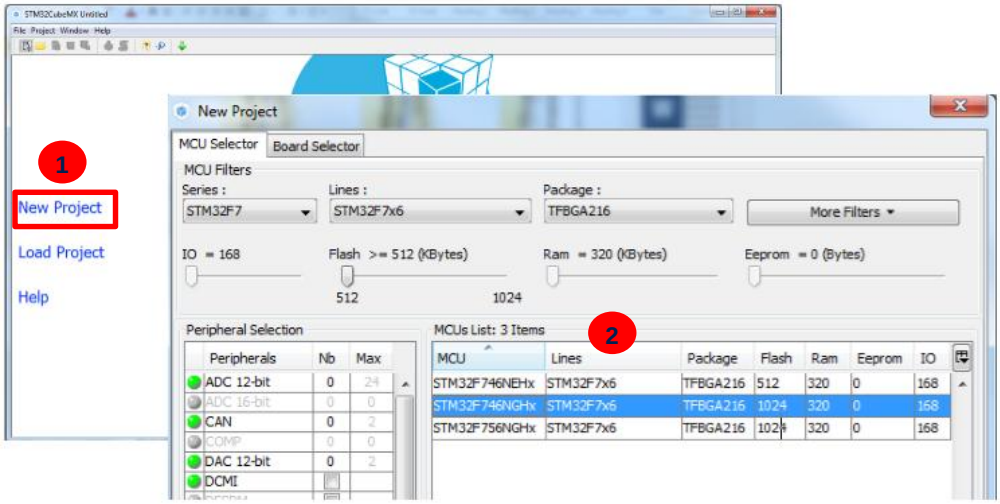


CoreMark 的测试代码文件包括：

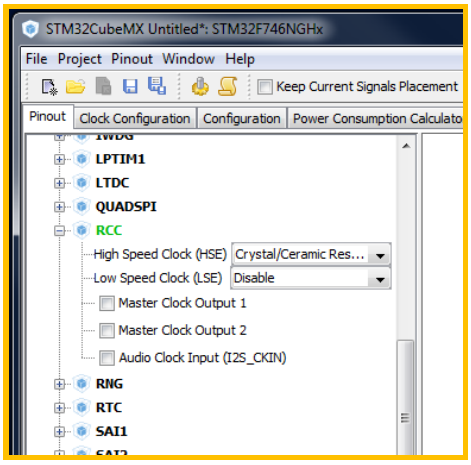
- core_list_join.c
- core_main.c
- core_matrix.c
- core_state.c
- core_util.c
- coremark.h
- simple/core_portme.c
- simple/core_portme.h

新建 CoreMark STM32 工程

1）打开 STM32CubeMX,选择新建 Project, 在接下来的窗口中选择目标 MCU 的型号。可以通过 MCU 筛选器进行筛选，见下图。这里我们选择 STM32F746NG。



2）选择使用外部晶振

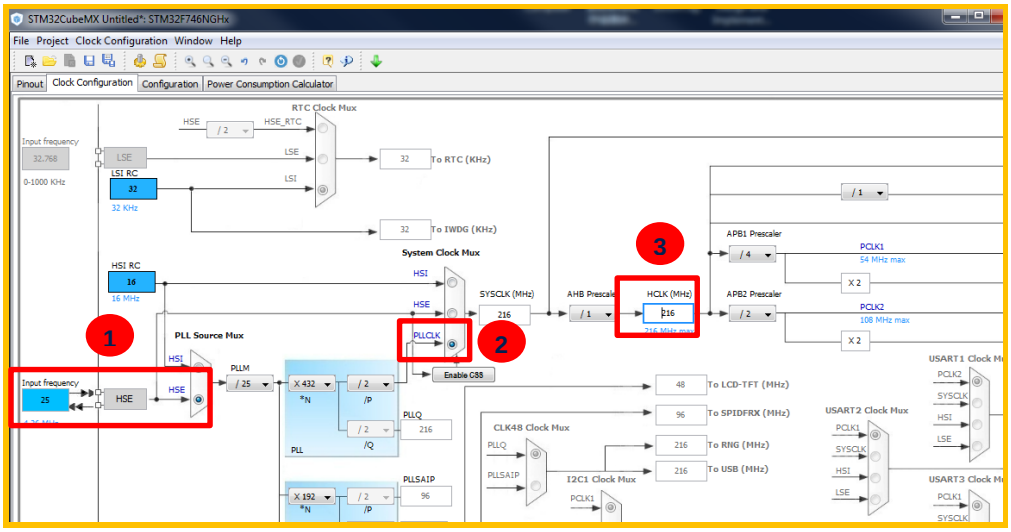


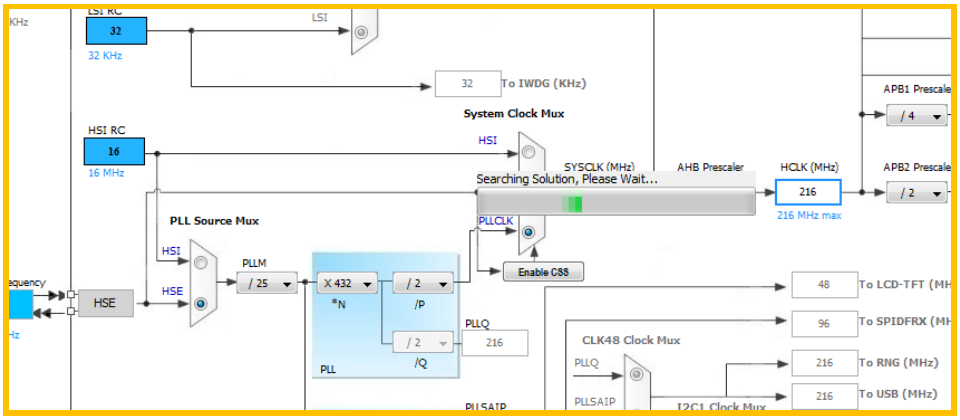
3）配置时钟

Step1 :PLL source 选择外部高速时钟（HSE，25MHz）

Step2 :系统时钟源选择 PLLCLK

Step3 :HCLK 设置为 216MHz,回车后工具会自动计算出合适的 PLL 配置参数。



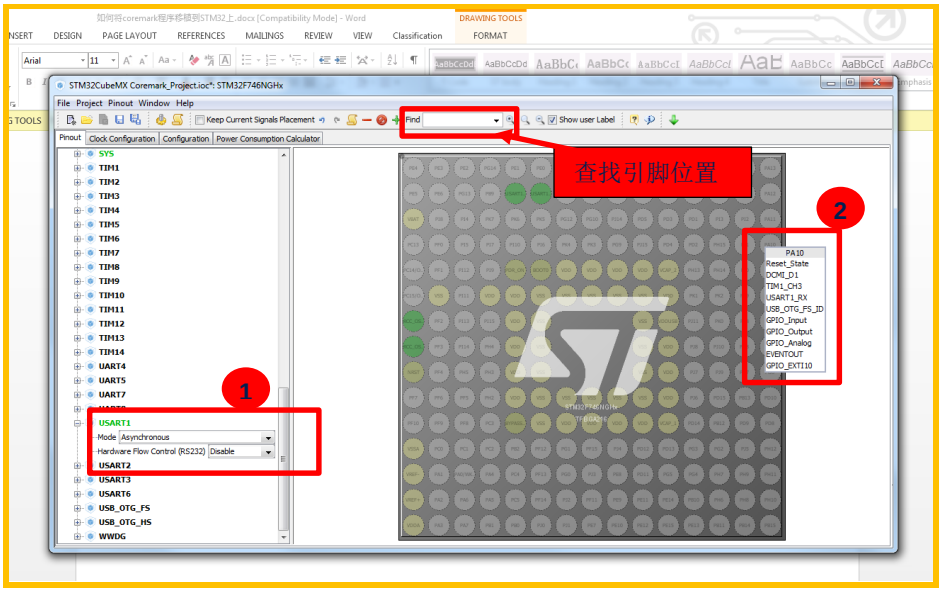


4）配置串口

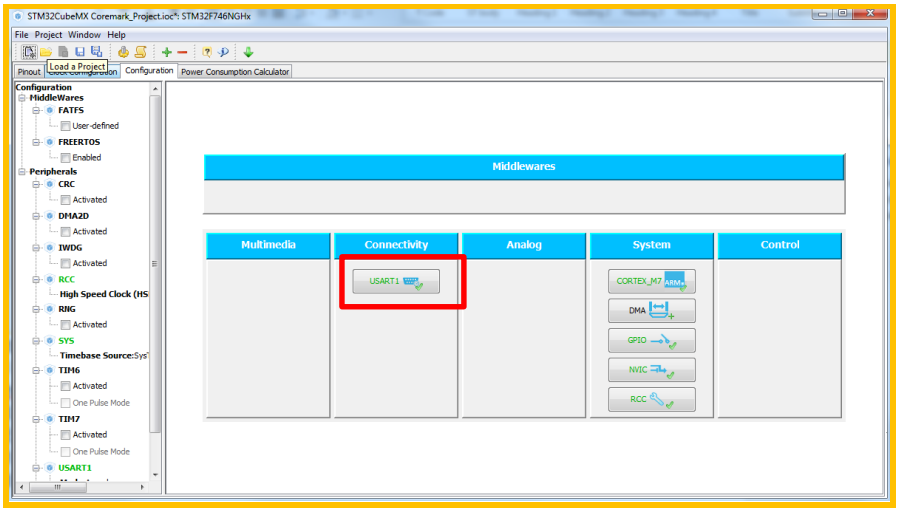
根据所用测试板子具体使用的串口进行配置。这里我们使用的 USART1（PA9，PA10 端口）。

Step1：使能 USART1。

Step2：将 USART1 重映射到 PA9 和 PA10 引脚。可以在右图直接点击对应引脚，在跳出的列表中选择 USART 的第二功能。不知道引脚的位置可以在上方的 Find 窗口内输入引脚的名称来查找引脚的位置。



Step3 :到 Configuration 页面，对串口进行参数配置。



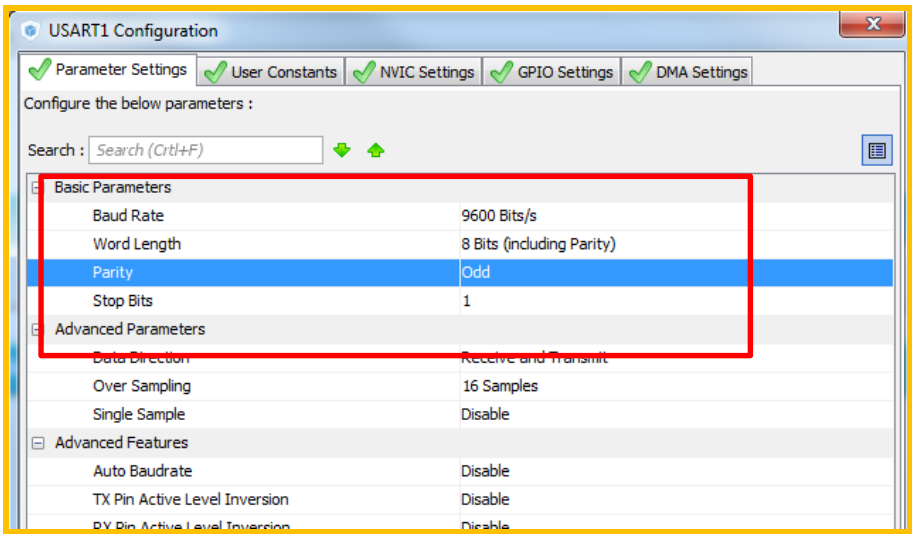
设置串口参数为：

波特率：9600Bits/s

数据长度：8bit(包括奇偶校验位)

校验：ODD

停止位：1 bit



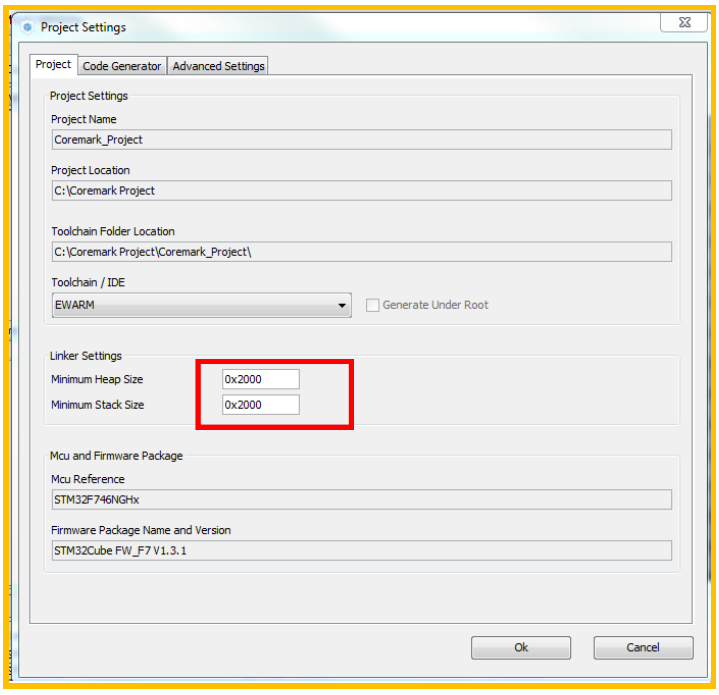
5） 生成 IAR 项目代码

做完上面的设置后，就可以让 CubeMX 帮我们生成代码了。

选择 Projec-->Generate Code,在跳出的 Project 配置窗口中指定项目名称和保存路径。选择要使用的工具链，这里选择 EWARM。

配置最小堆栈大小。

点击“OK”后，CubeMX 会自动在指定路径生成一个 IAR 的工程。这个工程已经包含了所有用到的底层驱动和并已经添加了系统初始化的代码。



添加 CoreMark 代码

现在我们已经有了一个初步的项目工程。接下来要做的就是添加 CoreMark 代码。

1）将前面下载的 CoreMark 代码文件拷贝到新建的工程中。

文件名	目标路径
core_list_join.c/core_main.c/core_matrix.c/core_state.c/core_util.c/ coremark.h	Coremark_Project\Src\Coremark
core_portme.c	Coremark_Project\Src
core_portme.h	Coremark_Project\Inc

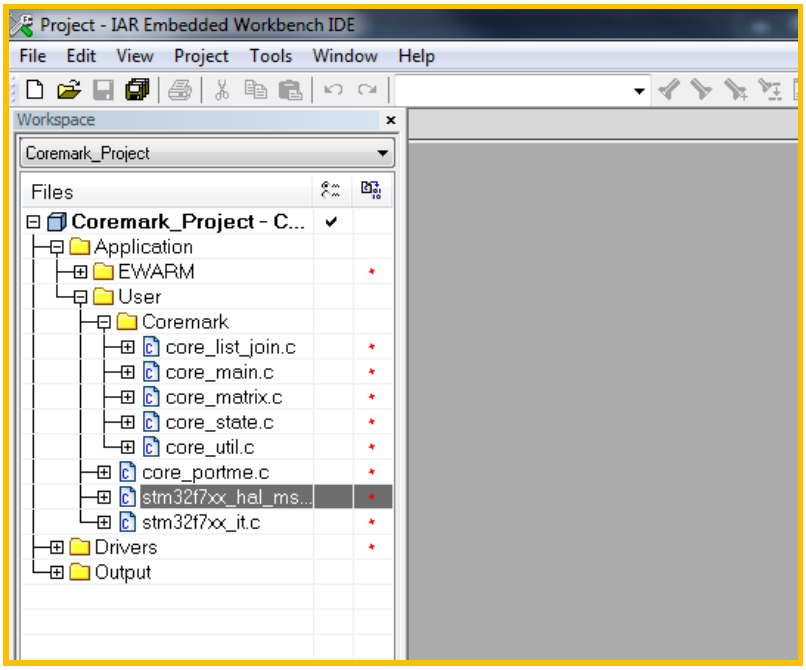
2）添加文件到工程

打开新建的工程 Coremark_Project。在 Application/User 目录下新建一个目录 Coremark,将 core_list_join.c /core_main.c/core_matrix.c/core_state.c/core_util.c 这 5 个文件添加进去。（选中左边工程中 User 目录->单击右键->Add->Add Group/Add Files）

再将 core_portme.c 添加到 User 目录下。

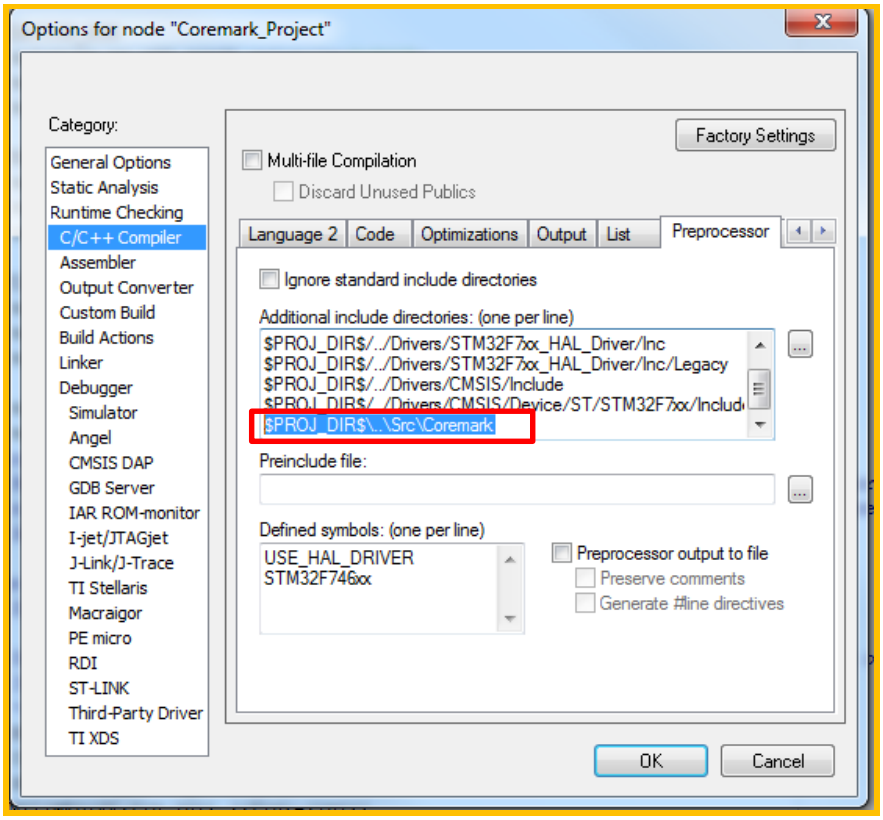
因为 core_main.c 文件里已经包含了一个 main 函数，所以需要在工程中将默认创建的 main.c 文件删除。

完成后的工程文件结构如下：



3）添加 include 路径

在 Options->C/C++ Compiler->Preprocessor 下增加 include 路径：$PROJ_DIR$\..\Src\Coremark。



配置 Coremark 文件

我们已经添加了所有需要的文件，但现在程序还是不能正常运行。因为默认生成的 main.c 文件已经被从项目中删除了，我们需要在 Core_portme.c 中添加初始化的代码，并根据不同的计时方法修改 Core_portme.c 中计时相关函数和代码。

添加初始化代码

1) portable_init 函数

Core_portme.c 中的 portable_init 函数在 Core_main.c 的 main 函数中首先被调用，平台的初始化的函数（时钟，GPIO，串口，缓存）可以放在这里。所以我们将 CubeMX 生成的 Main 函数中的初始化代码拷贝到 portable_init 函数中。

修改前：

```
void portable_init(core_portable *p, int *argc, char *argv[])
{
	if (sizeof(ee_ptr_int) != sizeof(ee_u8 *)) {
		ee_printf("ERROR! Please define ee_ptr_int to a type that holds a
pointer!\n");
	}
	if (sizeof(ee_u32) != 4) {
		ee_printf("ERROR! Please define ee_u32 to a 32b unsigned type!\n");
	}
	p->portable_id=1;
}
```

修改后：

```
void portable_init(core_portable *p, int *argc, char *argv[])
{
	/* Reset of all peripherals, Initializes the Flash interface and the Systick. */
	HAL_Init();

	/* Configure the system clock */
	SystemClock_Config();
	/* Initialize all configured peripherals */
	  MX_GPIO_Init();
	  MX_USART1_UART_Init();
	/* Enable I-Cache */
	SCB_EnableICache();

	/* Enable D-Cache */
	SCB_EnableDCache();

	if (sizeof(ee_ptr_int) != sizeof(ee_u8 *)) {
		ee_printf("ERROR! Please define ee_ptr_int to a type that holds a
pointer!\n");
	}
	if (sizeof(ee_u32) != 4) {
		ee_printf("ERROR! Please define ee_u32 to a 32b unsigned type!\n");
	}
	p->portable_id=1;
}
```

STM32F7 内核有 4K Bytes 的数据缓存（DCache）和指令缓存(ICache)，程序在 Flash 中通过 AXI 总线运行时，为了达到最高的性能需要把数据缓存和指令缓存打开。STM32 其他的系列没有缓存也就不需要添加这部分代码。另外，如果在 linker 文件里配置将代码放在了其他的位置，缓存也不一定要打开，比如程序在 Flash 中通过 ITCM 总线运行，具体看程序的配置。

2）添加下面函数

将 main.c 中的 SystemClock_Config，MX_USART1_UART_Init 和 MX_GPIO_Init 函数拷贝过来。并添将加 printf 重定向的代码。

```
/** System Clock Configuration
*/
void SystemClock_Config(void)
{

  RCC_OscInitTypeDef RCC_OscInitStruct;
  RCC_ClkInitTypeDef RCC_ClkInitStruct;
  RCC_PeriphCLKInitTypeDef PeriphClkInitStruct;

  __HAL_RCC_PWR_CLK_ENABLE();

  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
```

```
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 25;
  RCC_OscInitStruct.PLL.PLLN = 432;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  HAL_RCC_OscConfig(&RCC_OscInitStruct);

  HAL_PWREx_EnableOverDrive();

  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
  HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_7);

  PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_USART1;
  PeriphClkInitStruct.Usart1ClockSelection = RCC_USART1CLKSOURCE_PCLK2;
  HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct);

  HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

  HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

  /* SysTick_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
}

/* USART1 init function */
void MX_USART1_UART_Init(void)
{

  huart1.Instance = USART1;
  huart1.Init.BaudRate = 9600;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_ODD;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  HAL_UART_Init(&huart1);
```

```
}

/** Configure pins as
        * Analog
        * Input
        * Output
        * EVENT_OUT
        * EXTI
*/
void MX_GPIO_Init(void)
{

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();

}
```

```
#ifdef __GNUC__
/* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
   set to 'Yes') calls __io_putchar() */
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */
/**
  * @brief  Retargets the C library printf function to the USART.
  * @param  None
  * @retval None
  */
PUTCHAR_PROTOTYPE
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the EVAL_COM1 and Loop until the end of transmission */
  HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);

  return ch;
}
```

3）在文件开头添加函数声明和变量定义：

```
UART_HandleTypeDef huart1;
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART1_UART_Init(void);
```

4）添加新的 include 文件

```
#include <inttypes.h>
#include "system_stm32f7xx.h"
```

```
#include "stm32f7xx_hal.h"
```

修改计时相关代码

start_time/ stop_time/ get_time 这几个函数，是 coremark 程序运行时计算程序运行时间所用。这里使用 system tick 进行计时，system tick 配置为 1ms 的中断间隔。system tick 中断函数中更新 Tick 的值，每进一次中断加 1。所以还需要修改 system tick 的中断处理函数。

1）在 Core_portme.c 中按下表找到需要修改的地方，并按表格的内容进行修改：

修改前	修改为

`void start_time(void) {`
`  GETMYTIME(&start_time_val );`
`}`
	`void start_time(void) {`
	`  Tick = 0;`
	`  SysTick_Config(SystemCoreClock/1000);`
	`}`

`void stop_time(void) {`
`  GETMYTIME(&stop_time_val );`
`}`
	`void stop_time(void) {`
	`  /* Stop the Timer and get the encoding time  */`
	`  SysTick->CTRL &= SysTick_Counter_Disable;`
	`  /* Clear the SysTick Counter */`
	`  SysTick->VAL = SysTick_Counter_Clear;`
	`}`

`CORE_TICKS get_time(void) {`
`  CORE_TICKS elapsed=(CORE_TICKS)(MYTIMEDIFF(stop_time_val, start_time_val));`
`  return elapsed;`
`}`
	`CORE_TICKS get_time(void) {`
	`  CORE_TICKS  elapsed = (CORE_TICKS)Tick;`
	`  return elapsed;`
	`}`

`#define NSECS_PER_SEC CLOCKS_PER_SEC`
`#define CORETIMETYPE clock_t`
`#define GETMYTIME(_t) (*_t=clock())`
`#define MYTIMEDIFF(fin,ini) ((fin)-(ini))`
`#define TIMER_RES_DIVIDER 1`
`#define SAMPLE_TIME_IMPLEMENTATION 1`
……
`static CORETIMETYPE start_time_val, stop_time_val;`
	`//#define NSECS_PER_SEC CLOCKS_PER_SEC`
	`//#define CORETIMETYPE clock_t`
	`//#define GETMYTIME(_t) (*_t=clock())`
	`//#define MYTIMEDIFF(fin,ini) ((fin)-(ini))`
	`//#define TIMER_RES_DIVIDER 1`
	`//#define SAMPLE_TIME_IMPLEMENTATION 1`
	……
	`//static CORETIMETYPE start_time_val, stop_time_val;`

`#define EE_TICKS_PER_SEC (NSECS_PER_SEC / TIMER_RES_DIVIDER)`	`#define EE_TICKS_PER_SEC 1000`

2）在 Core_portme.c 文件中添加新定义的变量和函数

```
#define SysTick_Counter_Disable        ((uint32_t)0xFFFFFFFE)
#define SysTick_Counter_Enable         ((uint32_t)0x00000001)
#define SysTick_Counter_Clear          ((uint32_t)0x00000000)
__IO uint32_t Tick;
```

system tick 的中断处理函数在 stm32f7xx_it.c 中。stm32f7xx_it.c 文件包含所有中断处理入口函数。根据不同的平台，这个文件的名字稍有不同。找到 SysTick_Handler 函数进行修改。

修改前：

```
void SysTick_Handler(void)
{
  /* USER CODE BEGIN SysTick_IRQn 0 */

  /* USER CODE END SysTick_IRQn 0 */
  HAL_IncTick();
  HAL_SYSTICK_IRQHandler();
  /* USER CODE BEGIN SysTick_IRQn 1 */

  /* USER CODE END SysTick_IRQn 1 */
}
```

修改后：

```
void SysTick_Handler(void)
{
  /* USER CODE BEGIN SysTick_IRQn 0 */
  extern __IO uint32_t Tick;
  Tick++;
  /* USER CODE END SysTick_IRQn 0 */

  /* USER CODE BEGIN SysTick_IRQn 1 */

  /* USER CODE END SysTick_IRQn 1 */
}
```

CoreMark 运行配置

1）设置迭代次数

CoreMark 要求程序运行的最短时间至少是 10s, 根据使用的系统时钟等情况，可以在 Core_portme.h 中修改迭代次数。

```
#define ITERATIONS      12000
```

2）设置打印信息

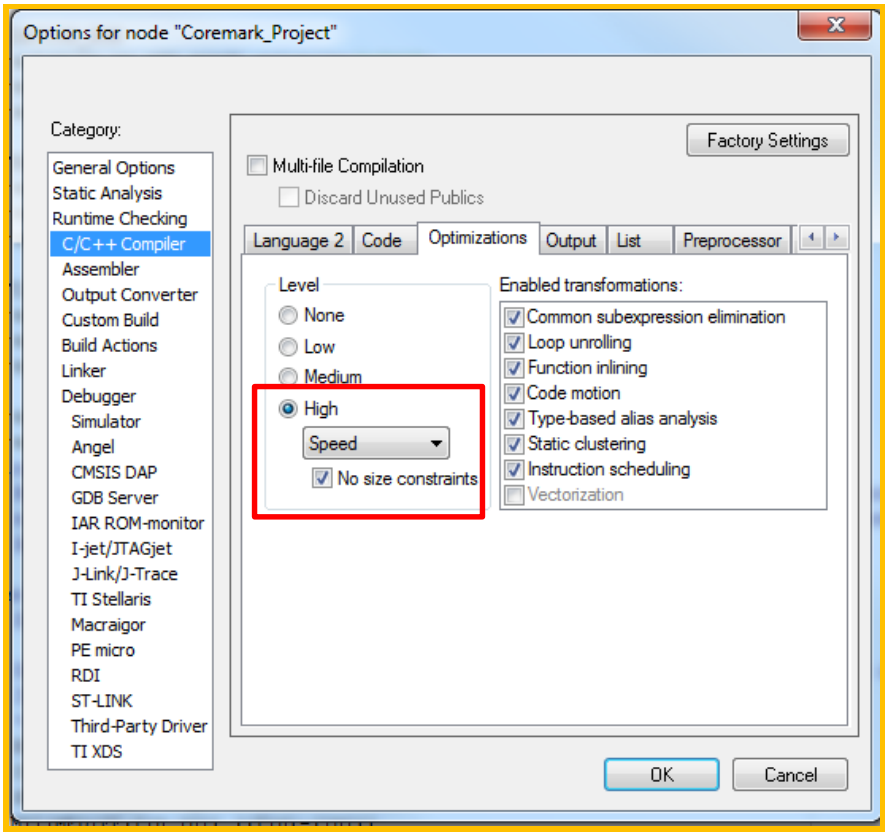
根据具体所用的编译器版本，优化配置进行修改。

找到	修改为

`#ifndef COMPILER_FLAGS`
`#define COMPILER_FLAGS FLAGS_STR /*`
`"Please put compiler flags here (e.g. -o3)"`
`*/`
`#endif`
	`#ifndef COMPILER_FLAGS`
	`#define COMPILER_FLAGS "-Ohs -`
	`no_size_constraints"`
	`#endif`

3）修改优化等级。

Options->C/C++ Compiler->Optimizations, 选择 High for speed 和 No size constraints 以达到最优的运行速度。



运行结果

程序已经完全配置好，并编译成功。

现在我们连接 STM32F746Discovery 板，打开串口调试助手，看看运行结果。

调试信息

1. 如果出现“ERROR! Must execute for at least 10 secs for a valid result!“的错误提示，说明 ITERATIONS 设定太小，可适当增加 ITERATIONS 的值。

2.如果串口接受不到调试信息，请检查代码中配置的串口是否是板子上所使用的串口。并检查串口的参数配置是否正确（波特率，数据位个数，校验位等）

3.如果测试的结果与所预知的结果相差很大（比如从 CoreMark 网站上查到的结果），请检查系统时钟是否配置正确（SystemCoreClock 的值是否正确），system tick 配置是否正确（Tick 的值是否正常）

重要通知 – 请仔细阅读

意法半导体公司及其子公司（“ST”）保留随时对ST 产品和/ 或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。买方在订货之前应获取关于ST 产品的最新信息。ST 产品的销售依照订单确认时的相关ST 销售条款。

买方自行负责对ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的ST 产品如有不同于此处提供的信息的规定，将导致ST 针对该产品授予的任何保证失效。

ST 和ST 徽标是ST 的商标。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。