HPM6000 系列

HPM6000 系列微控制器 DSP/FFT 使用介绍

先楫半导体HPM6000系列微控制器DSP/FFT使用介绍

目录

1 简介 4
2 DSP 例程介绍 4
3 FFT 性能测试 10
3.1 测试环境介绍 11
3.2.1 复数 FFT 测试过程介绍 12
3.2.2 复数 FFT 测试结果 18
3.3.1 实数 FFT 测试过程介绍 19
3.3.2 实数 FFT 测试结果 20
3.4.1 DSP 硬件测试点数的增加 21
3.4.2 软件算法测试点数的增加 22
4 计算性能测试 25
4.1 测试环境搭建 25
4.2 测试过程介绍 25
4.3 测试结果 27
5 总结 28

版本：

日期	版本号	说明
2023-4-10	1.0	初版

1 简介

HPM6000 系列 MCU 是来自上海先楫半导体科技有限公司的高性能实时 RISC-V 微控制器，为工业自动化及边缘计算应用提供了极大的算力、高效的控制能力。上海先楫半导体目前已经发布了如 HPM6700/6400、HPM6300 等多个系列的高性能微控制器产品。

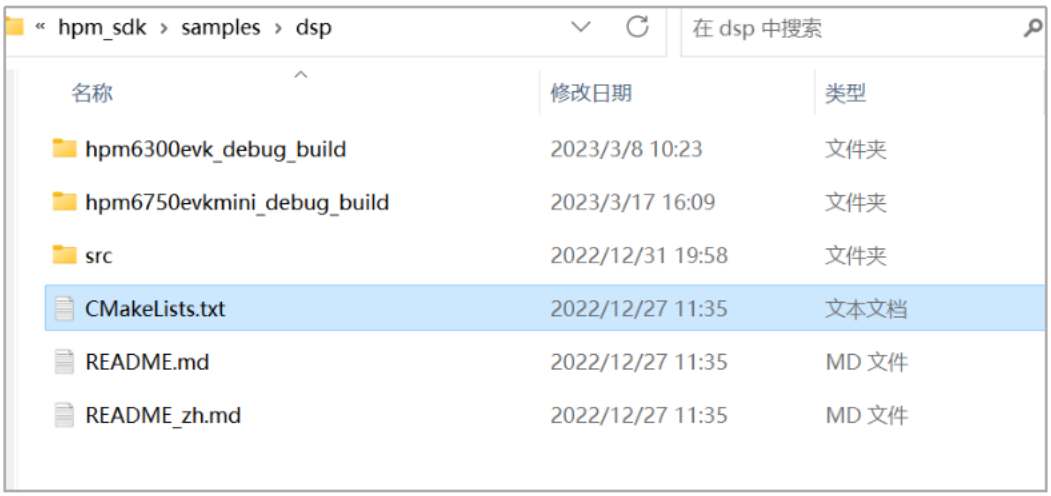
在 32 位处理器中，都支持 RV32-IMAFDCP 指令集，包含整数指令集，乘法指令集，原子指令集，单精度浮点数指令集，双精度浮点数指令集，压缩指令集，DSP 单元支持 SIMD 和 DSP 指令并且兼容 RV32-P 扩展指令集。

用户可以访问 ：http://www.andestech.com/en/products-solutions/product/ 并下载《AndeStar V5 DSP ISA Extension Specification》来获取 DSP 扩展指令的详细信息。

2 DSP 例程介绍

在先楫官方的 SDK 下，dsp_demo 示例工程演示了 DSP CFFT 和 CIFFT 计算，通过串口打印结果。本节将为用户介绍 hpm_dsp 的有关环境搭建。环境为 SDK1.0。

在 SDK 环境下生成代码都需要一个 CMakeLists 文件，内容包含了此工程相关的一些定义，添加 C 文件和头文件等功能。使用硬件 DSP 单元以及启用数学运算库就需要在自己工程的 CMakeLists.txt 中加入相应配置语句，在下图路径下可以找到此例程的 CMakeLists.txt。



打开后可以看到，由于本例程使用到了单精度浮点数，默认已经添加了：

```
set(SES_TOOLCHAIN_VARIANT "Andes")
set(CONFIG_HPM_MATH 1)
set(CONFIG_HPM_MATH_DSP 1)
set(HPM_MATH_SES_LIB "libdspf")
sdk_ses_compile_options(-mabi=ilp32f)
sdk_ses_compile_options(-march=rv32imafc)
```

前三句话将 Andes 的工具链以及 hpm_math 库进行添加，后三句是硬件浮点数的相关配置。

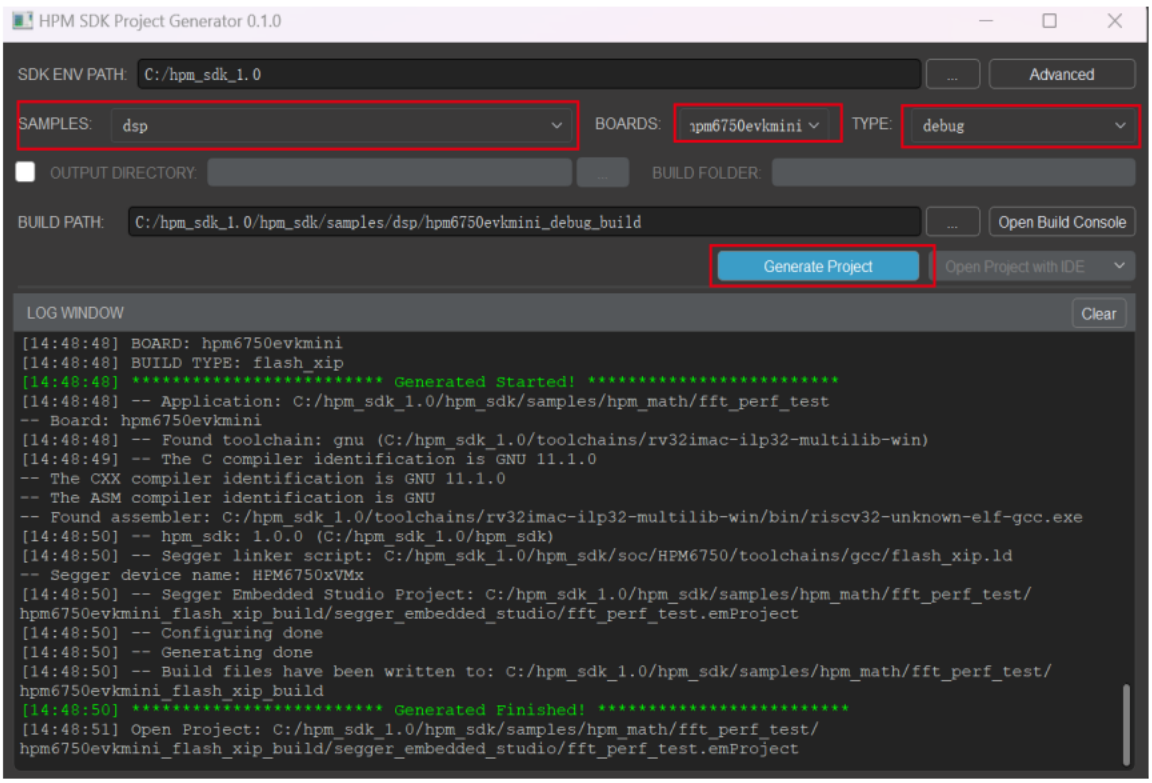
如果需要使用到双精度浮点数，那么需要在工程下的 CMakeLists.txt 中加入如下语句（需要注意，每次如果修改了 CMakeLists.txt 的内容，一定要重新生成工程，配置才会生效）：

```
set(SES_TOOLCHAIN_VARIANT "Andes")
set(CONFIG_HPM_MATH 1)
set(CONFIG_HPM_MATH_DSP 1)
set(HPM_MATH_SES_LIB "libdspd")
sdk_ses_compile_options(-mabi=ilp32d)
sdk_ses_compile_options(-march=rv32gc)
```

如果用户想要不开启硬件浮点数，只需要在工程下的 CMakeLists.txt 中加入如下语句：

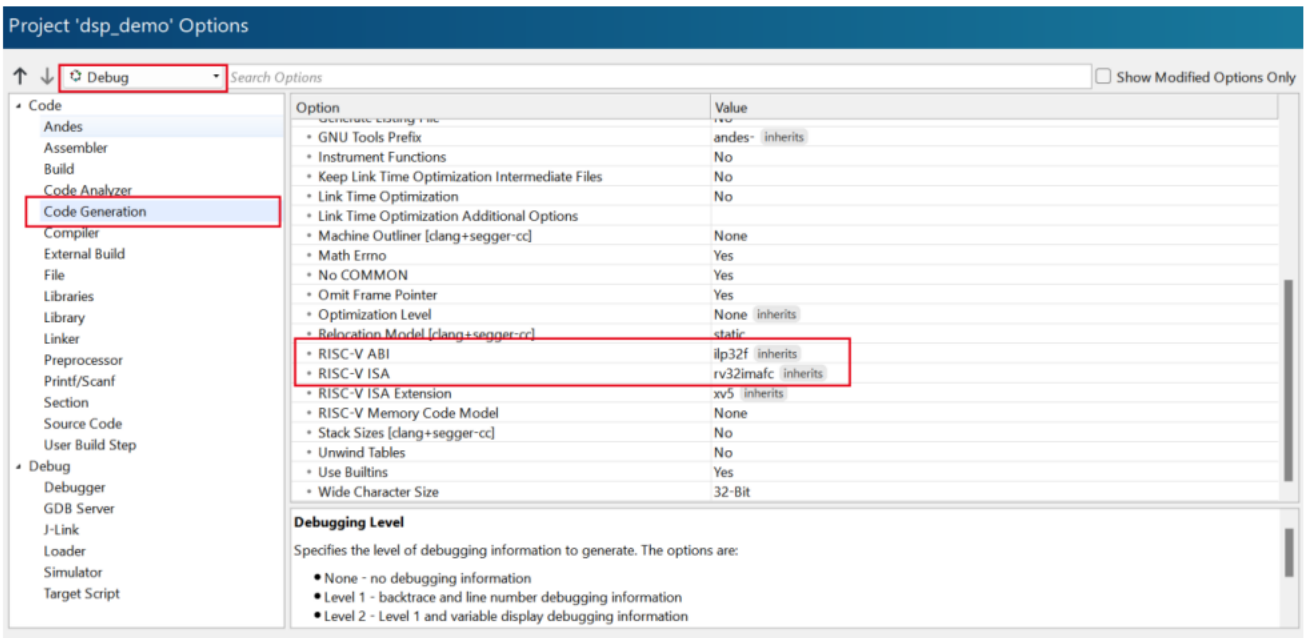
```
set(SES_TOOLCHAIN_VARIANT "Andes")
set(CONFIG_HPM_MATH 1)
set(CONFIG_HPM_MATH_DSP 1)
set(HPM_MATH_SES_LIB "libdsp")
```

用户完成相关配置后就可以生成工程了。在 SDK 文件中有一个 GUI 生成工具，如下图所示，打开后 Samples 选择 **DSP**，开发板为 HPM6750EVKMINI，工程类型选择 debug 生成 ram 工程。点击生成工程，提示一键生成成功后，点击 open project with IDE 可以直接打开工程文件进行调试。



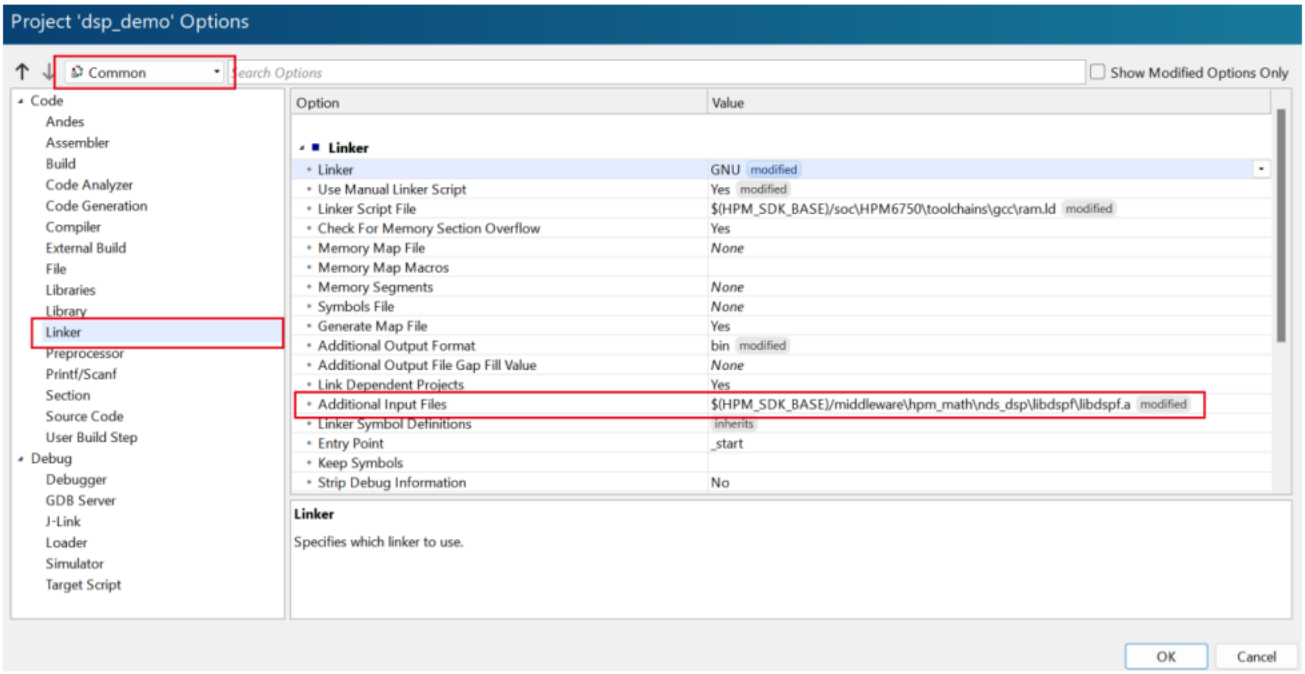
对于选择单精度，双精度的硬件浮点数或是关闭硬件浮点，以上介绍的通过 CMakeLists.txt 来修改是第一种方法。

第二种方法是在集成开发环境下进行修改，生成工程后，打开 SEGGER，在 SEGGER 中右击工程打开选项菜单。点击进入 Code Generation，**单精度**可以修改下图位置，在 DEBUG 菜单界面中，将图中位置依次改为 **ilp32f 以及 rv32imafc**：

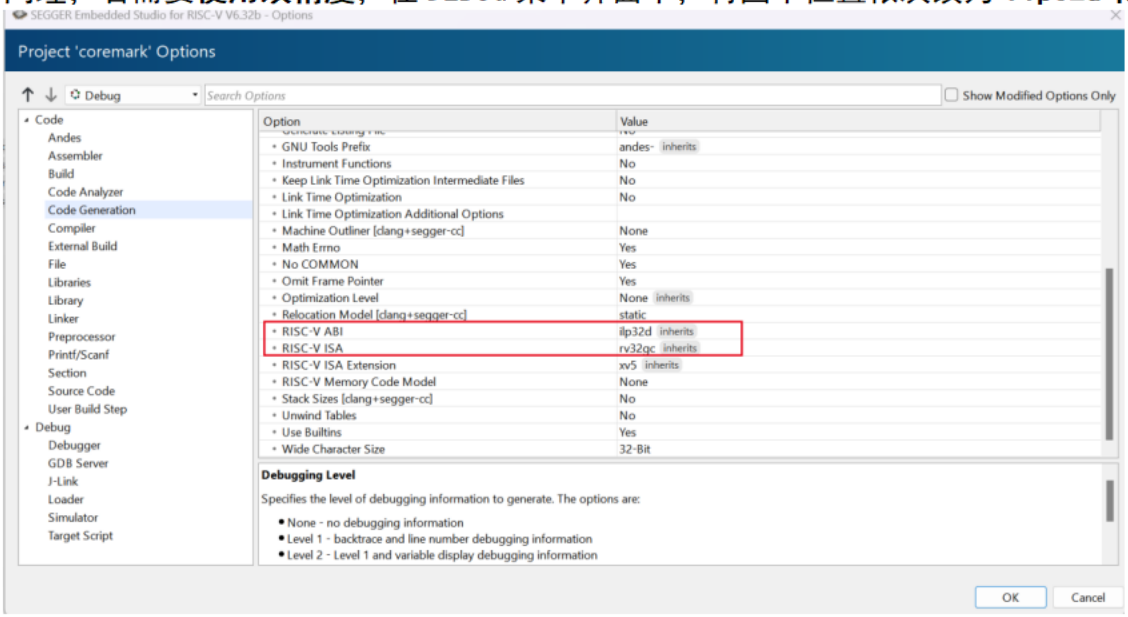


改完此处后，还需要把 andes 的 dsp_lib 库添加进集成环境中。在下图路径中存放的就是三个不同的 lib 库，从上到下分别对应不开启硬件浮点，开启双精度，开启单精度：

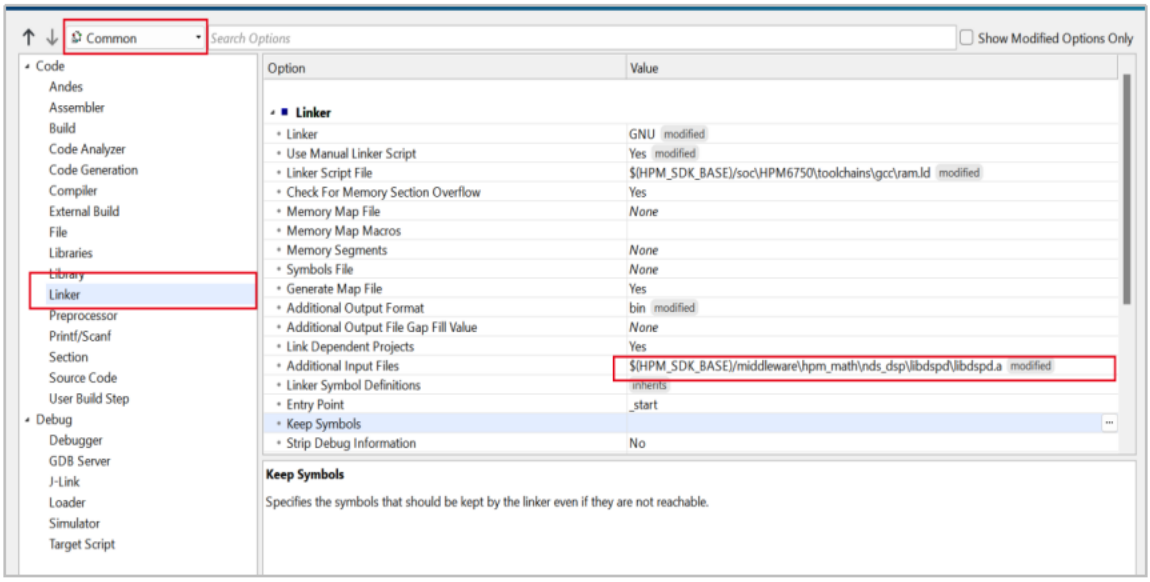
同样打开工程的选项菜单，需要将左上角的 debug 切换到 common，选择 Linker，在 Additional Input Files 中添加单精度的路径：...\libdspf\libdspf.a，如下图所示：



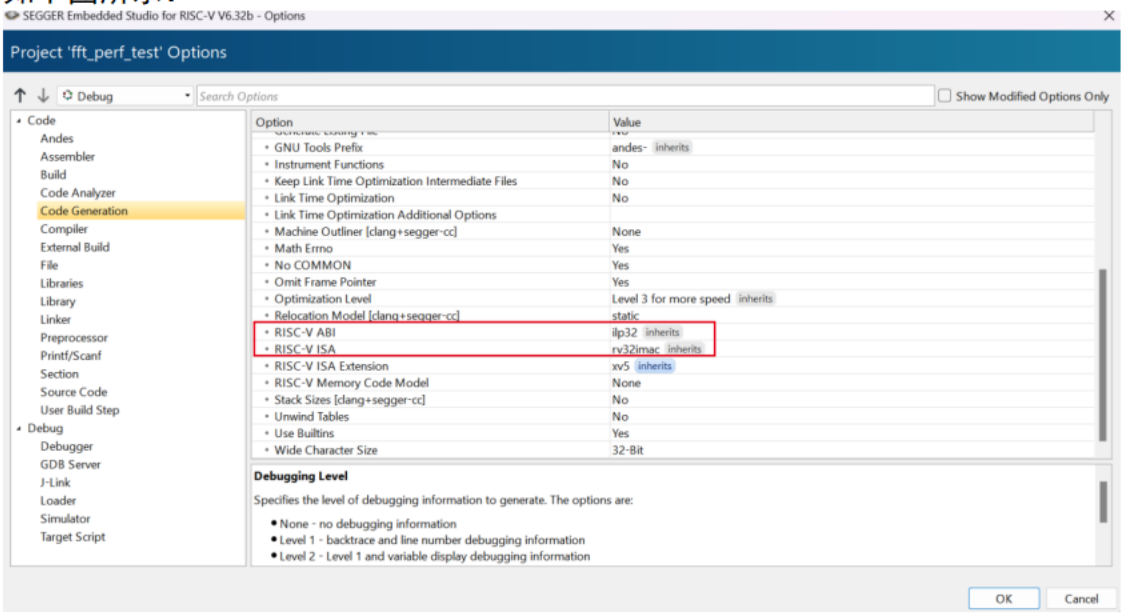
同理，若需要**使用双精度**，在 DEBUG 菜单界面中，将图中位置依次改为 **ilp32d 和 rv32gc**：



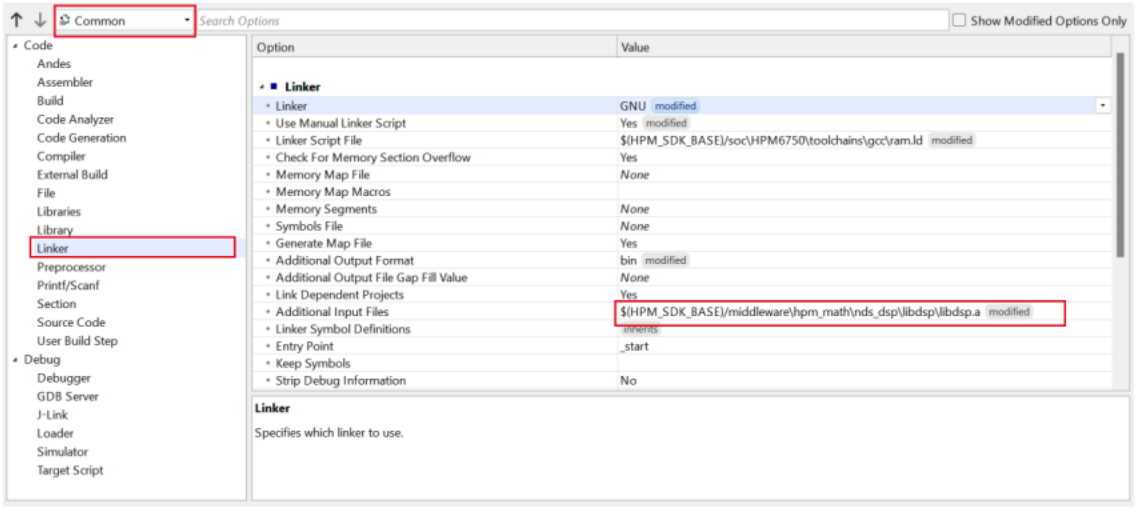
并且在 common 界面中添加双精度的 dsp_lib 库，如下图所示：



如若**不开启硬件浮点数**，在 DEBUG 菜单界面中，将图中位置依次修改为 **ilp32，rv32imac**，如下图所示：



并且在 common 界面添加不带硬件浮点数的 dsp_lib 库，如下图所示：



需要用户注意的是，添加对应 lib 库与 sdk_ses_compile_options 的配置一定需要同步。如果两处修改不同步，运算会出现问题。

```
set(CONFIG_HPM_MATH 1)
set(CONFIG_HPM_MATH_DSP 1)
set(HPM_MATH_DSP_SES_LIB "libdspf")
find_package(hpm-sdk REQUIRED HINTS $ENV{HPM_SDK_BASE})

project(dsp_demo)

sdk_ses_compile_options(-mabi=ilp32f)
sdk_ses_compile_options(-march=rv32imafc)
```

完成以上配置之后，打开 dsp 工程，dsp 例程已经默认添加了以下头文件：

```
#include <math.h>
#include "hpm_math.h"
```

用户如果自创工程需要完成以上介绍的 CMakeList 的配置：添加 Andes 的工具链以及 hpm_math 库。然后在 SEGGER 中加入这两个头文件，就可以成功调用库了。

前往 hpm_math.h 的定义，存放地址：hpm_sdk/middleware/hpm_math，如下图部分截图所示。这里面已经集成了 hpm_dsp 的各种 api 指令与相应注释，用户可以直接调取使用。其中就包括后文测试需要使用到的：各种格式下的实数 fft 运算，虚数 fft 运算，数学运算公式等等。

dsp_demo.c hpm_math.h

```
79  // Maximum
80  /**
81   * @brief Maximum value of the floating-potint vector.
82   * @param[in]       *src        points to the input vector.
83   * @param[in]       size        size of the vectors.
84   * @param[out]      *index      index of the maximum value.
85   * @return maximum value.
86   */
87  static inline float32_t hpm_dsp_max_f32(const float32_t *src, uint32_t size, uint32_t *index)
88  {
89  #if HPM_DSP_CORE == HPM_DSP_HW_NDS32
90      return riscv_dsp_max_f32(src, size, index);
91  #endif
92  }
93  static inline float32_t hpm_dsp_max_val_f32(const float32_t *src, uint32_t size)
94  {
95  #if HPM_DSP_CORE == HPM_DSP_HW_NDS32
96      return riscv_dsp_max_val_f32(src, size);
97  #endif
98  }
99  /**
```

如下图所示，先楫半导体官方 API 介绍文档中也可以查阅到，地址为：hpm_sdk\doc\output\api_doc\html\index.html 。

DSP 例程中的 CFFT 和 CIFFT 计算就是调用的 DSP 库中的 API：**hpm_dsp_cifft_rd2_q15(q15_t *src, uint32_t m)；**在 Q15 格式下对 Radix-2 复数进行 fft 与 cfft 计算。在第三章节中，将详细介绍 HPM_FFT 的相关运算。

3 FFT 性能测试

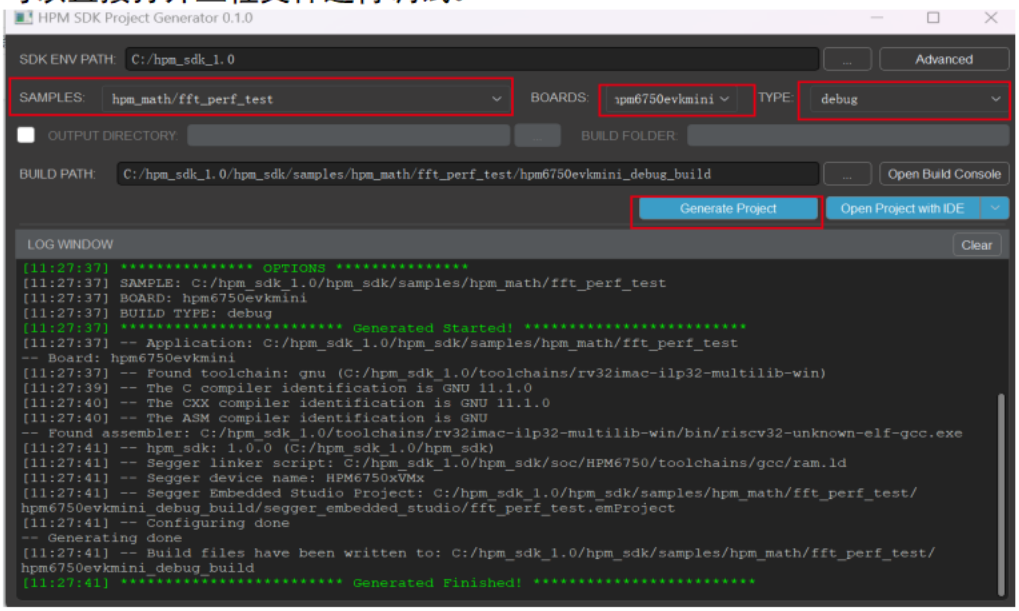
例程基于 HPM 官方 SDK 中 hpm_math 文件下的 fft_perf.test 进行测试。测试程序通过软件生成几个幅值和相位不同的正弦波并叠加生成一个波形，然后使用 FFT 变换求出幅频特性数据，最后做 IFFT 逆变换恢复原始数据， 并比对 IFFT 的数据和 FFT 变换的数据是否一致。当工程正确运行后，串口终端会输出计算的名称，采样点数，以及幅频特性的数据，以及每一个单元的运行时间。测试会放在不同存储区域下，是否使能 cache 情况下，做相应测试。

3.1 测试环境介绍

本次测试使用芯片型号为 HPM6750，开发板使用 HPM6750EVKMINI，芯片主频高达 816Mhz。软件包使用先楫半导体的是 SKD1.0 版本，使用 SEGGER Embedded Studio for RISC-V 6.32b 进行编译调试。

有关如何开启硬件 DSP 以及浮点数单元，可参考 2.1 节中的介绍，在工程路径下的 CMakeLists.txt 加入相应语句，此节测试与 2.1 节配置相同，使用单精度的硬件浮点数，唯一不同的是此例程已经开启 O3 优化。

Samples 选择 hpm_math/fft_perf_test，开发板为 HPM6750EVKMINI，工程类型选择 debug 生成 ram 工程。点击生成工程，提示一键生成成功后，点击 open project with IDE 可以直接打开工程文件进行调试。

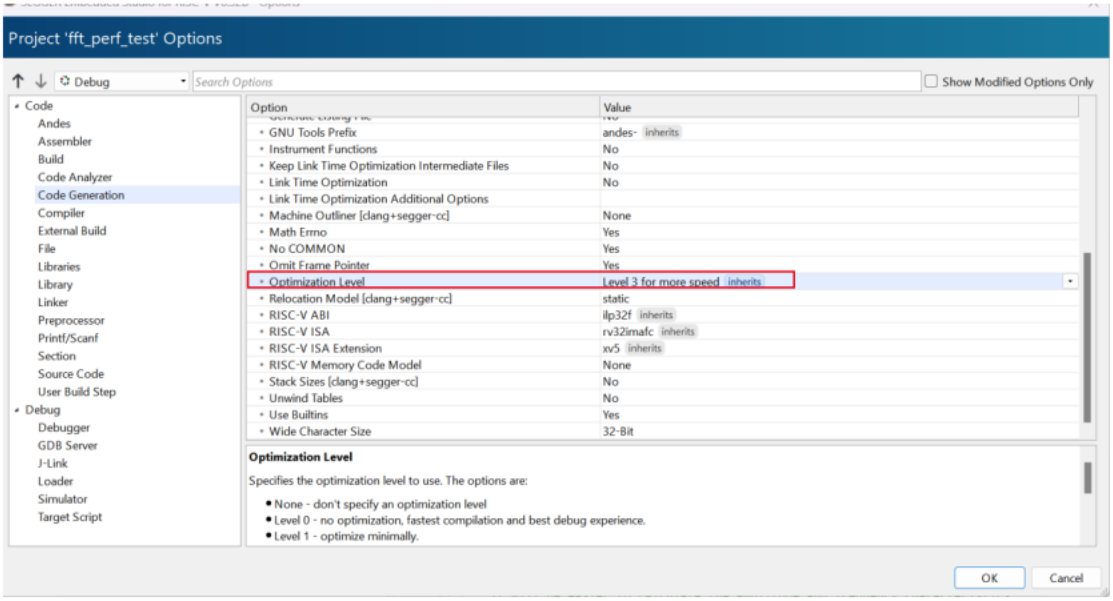


此例程中默认打开了 O3 优化，可以看到在工程下的 CMakeLists.txt 中加入了：

sdk_compile_options("-O3")

set(SEGGER_LEVEL_O3 1)

也可以通过 SEGGER 开发环境中来设置优化，如下图所示，选择 **level 3 for more speed**：



在 **SDK1.0** 版本中，本例程默认支持的采样点 8/16/32/64/128/256/512 /1024 点。点数相关代码存放在 hpm_math_sw.c 中，路径如下图所示：

3.2.1 复数 FFT 测试过程介绍

（1）周期数计算代码

本次测试例程中，各个测试项和测试点数所需 tick 为 CPU 周期数，cpu 一个周期时间运行是主频的倒数，所以用运行所需的周期数（tick）除以主频（Hz），得到一秒内执行的次数。相关代码如下所示：

```
void clear_cycle(void)
{
   write_csr(CSR_MCYCLE, 0);
}
uint32_t read_cycles(void)
{
   uint32_t cycles;
   cycles = read_csr(CSR_MCYCLE);
   return cycles;
}
```

通过使用符合 RISC-V 规范的硬件性能监视器：CSR_MCYCLE，64 位的计数器，统计特定时刻以来处理器运行的时钟周期数。

（2）Software Cooley-tukey

软件库中以 Software Cooley-tukey 的测试是先楫半导体 SDK 提供软件 FFT 数学算法库，可以实现经典的蝶形算法。计算接口为软件算法存放在 middleware/hpm_math_sw.c 中，部分定义如下图所示：

```
902  void hpm_software_cfft_float(float *src, uint32_t m)
903  {
904      uint32_t len;
905      complex t;
906      complex w = {1, 0};
907      complex *c = (complex *)src;
908      len = 1 << m;
909  #ifdef HPM_MATH_SW_FFT_CHECKLIST
910      uint16_t *memory;
911      memory = hpm_math_sw_fft_str[m-1];
912  #else
913      uint32_t *memory;
914      memory = (uint32_t *)(src + 2 * len);
915      for (int i = 0; i < len; i++) {
916          memory[i] = (memory[i >> 1] >> 1) | ((i & 1) << (m - 1));
917      }
918  #endif
```

测试代码如下所示：

```
for (uint8_t i = 6; i <= 10; i++) {
      point = 1 << i;
      shift = i;
      init_fft_inputbuf(&fft_buf[0], point);
      clear_cycle();
      hpm_software_cfft_float(&fft_buf[0], shift);
      run_times = read_cycles();
      printf("Software fft  cooley tukey Total samples: %d.\r\n", point);
      printf("total times:%d tick.\r\n", run_times);
      fft_printf(&fft_buf[0], &fft_mag_output[0], point);
   }
   printf("**************************************\r\n\r\n\r\n\r\n");
```

（3）DSP radix-2 complex samples

Radix-2 复数快速傅里叶变换（CFFT）和快速傅里叶逆变换（CIFFT）函数实现了著名的 Cooley-Tukey 算法来转换时域信号到频域。SDK 中已经集成了 Andes 的扩展 DSP Library，已经有相应性能加速，用户直接调用即可：**hpm_dsp_cfft_rd2_f32(float32_t *src, uint32_t m)；**输入浮点数以及点数。计算后打印出需要多少机器周期数。对应 FFT 以及 IFFT 的测试代码片段如下：

```
for (uint8_t i = 6; i <= 10; i++) {
      point = 1 << i;
      shift = i;
      init_fft_inputbuf(&fft_buf[0], point);
      clear_cycle();
      hpm_dsp_cfft_rd2_f32(&fft_buf[0], shift);
      run_times = read_cycles();
      printf("dsp fft radix-2 Total samples: %d.\r\n", point);
      printf("total times:%d tick.\r\n", run_times);
      fft_printf(&fft_buf[0], &fft_mag_output[0], point);

      clear_cycle();
      hpm_dsp_cifft_rd2_f32(&fft_buf[0], shift);
      run_times = read_cycles();
      printf("dsp ifft radix-2 Total samples: %d.\r\n", point);
      printf("total times:%d tick.\r\n", run_times);
      for (uint32_t m = 0; m < 2 * point; m++) {
         if ((fft_buf_copy[m] > fft_buf[m] + FFT_PRECISION) ||
         (fft_buf_copy[m] < fft_buf[m] - FFT_PRECISION)) {
            err_num++;
         }
      }
      printf("-----------------------------------\r\n\r\n\r\n");
   }
```

（4）Q31 DSP radix-2 complex samples

调用 **hpm_dsp_cfft_rd2_q31(q31_t *src, uint32_t m);** 为了满足输入的 Q31 格式，需要提前执行算术移位操作。转换的接口也可以在 hpm_math.h 中找到：**hpm_dsp_convert_f32_q31(float32_t *src, q31_t *dst, uint32_t size);**

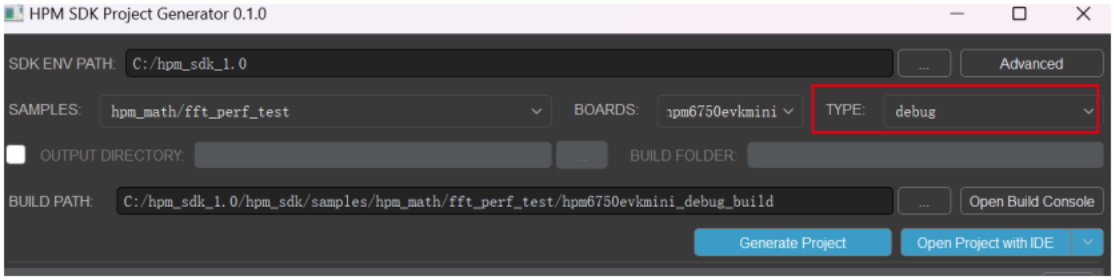
```
for (uint8_t j = 6; j <= 10; j++) {
      point = 1 << j;
      shift = j;
      init_fft_inputbuf(&fft_buf[0], point);
      clear_cycle();
      hpm_dsp_cfft_rd2_q31(&ffa_buf[0], shift);
      run_times = read_cycles();
      printf("dsp fft q31 radix-2 Total samples: %d.\r\n", point);
      printf("total times:%d tick.\r\n", run_times);
      hpm_dsp_convert_q31_f32(ffa_buf, fft_buf, point * 2);
      for (uint32_t i = 0; i < 2 * point; i++) {
         fft_buf[i] = fft_buf[i] * point * point * 2;
      }
      fft_printf(&fft_buf[0], &fft_mag_output[0], point);
```

```
    clear_cycle();
    hpm_dsp_cifft_rd2_q31(ffa_buf, shift);
    run_times = read_cycles();
    hpm_dsp_convert_q31_f32(ffa_buf, fft_buf, point * 2);
    for (uint32_t i = 0; i < 2 * point; i++) {
        fft_buf[i] = fft_buf[i] * point;
    }
    printf("dsp ifft q31 radix-2 Total samples: %d.\r\n", point);
    printf("total times:%d tick.\r\n", run_times);
    for (uint32_t m = 0; m < 2 * point; m++) {
        if ((fft_buf_copy[m] > fft_buf[m] + FFT_PRECISION) ||
        (fft_buf_copy[m] < fft_buf[m] - FFT_PRECISION)) {
            err_num++;
        }
    }
    printf("-----------------------------------\r\n\r\n\r\n");
}
```

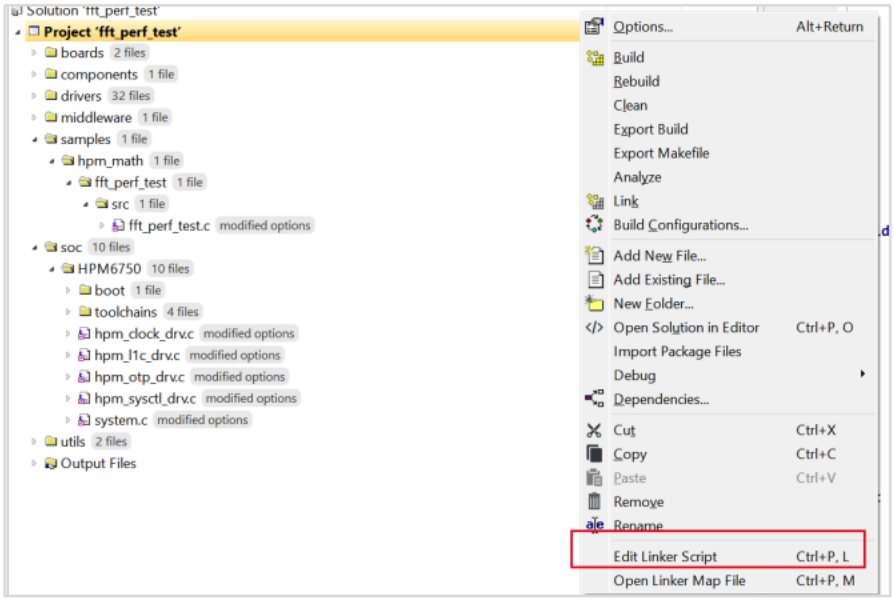
（5）代码运行区域和 cache 设定

例程创建后为 cache enable 的状态，可以在代码初始化之后加入 **l1c_ic_disable()**；这样可以使 cache disable。

同时本次测试在三种区域中进行了测试，分别是 ILM，XPI0，AXI_SRAM。**ILM 分配区域方法：**在创建工程的 GUI 中，工程类型选择 debug，如下图所示：



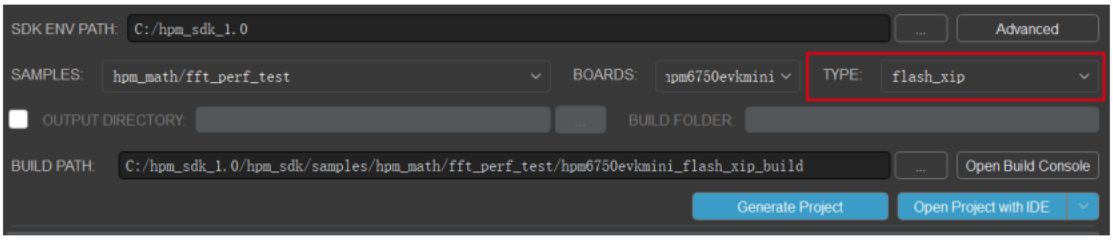
打开工程后，右击工程，打开选择菜单，点击 Edit Linker Script，可以在这里查看并且编辑工程的 Linker 文件。



由于创建的是 debug 类型，可以在下图中看出，在 SECTIONS 的分配区域代码中，已经分配了 ILM。

```
SECTIONS
{
    .start : {
        . = ALIGN(8);
        KEEP(*(.start))
    } > ILM

    .vectors : {
        . = ALIGN(8);
        KEEP(*(.isr_vector))
        KEEP(*(.vector_table))
        . = ALIGN(8);
    } > ILM

    .rel : {
        KEEP(*(.rel*))
    } > ILM

    .text : {
        . = ALIGN(8);
        *(.text)
        *(.text*)
        *(.rodata)
        *(.rodata*)
        *(.srodata)
        *(.srodata*)

        *(.hash)
        *(.dyn*)
        *(.gnu*)
        *(.pl*)

        KEEP(*(.eh_frame))
        *(.eh_frame*)

        KEEP (*(.init))
        KEEP (*(.fini))

        /* section information for usbh class */
        . = ALIGN(8);
        __usbh_class_info_start__ = .;
        KEEP(*(.usbh_class_info))
        __usbh_class_info_end__ = .;

        . = ALIGN(8);
        PROVIDE (__etext = .);
        PROVIDE (_etext = .);
        PROVIDE (etext = .);
    } > ILM
```

XIP0 区域分配方法：在创建工程的 GUI 中，工程类型选择 flash_xip，如下图：



同样点击 Edit Linker Script，打开 linker 文件，可以看到区域配置已经为 XIP0。

AXI_SRAM 区域分配方法：在创建工程的 GUI 中，工程类型选择 flash_xip，生成的工程中点击 Edit Linker Script，打开 linker 文件，将下图中标注位置的 XPI0 位置修改为 AXI_SRAM。

```
__nor_cfg_option_load_addr__ = ORIGIN(AXI_SRAM) + 0x400;
__boot_header_load_addr__ = ORIGIN(AXI_SRAM) + 0x1000;
__app_load_addr__ = ORIGIN(AXI_SRAM) + 0x3000;
__boot_header_length__ = __boot_header_end__ - __boot_header_start__;
__app_offset__ = __app_load_addr__ - __boot_header_load_addr__;


SECTIONS
{
    .nor_cfg_option __nor_cfg_option_load_addr__ : {
        KEEP(*(.nor_cfg_option))
    } > AXI_SRAM

    .boot_header __boot_header_load_addr__ : {
        __boot_header_start__ = .;
        KEEP(*(.boot_header))
        KEEP(*(.fw_info_table))
        KEEP(*(.dc_info))
        __boot_header_end__ = .;
    } > AXI_SRAM

    .start __app_load_addr__ : {
        . = ALIGN(8);
        KEEP(*(.start))
    } > AXI_SRAM

    .text (__vector_load_addr__ + __vector_ram_end__ - __vector_ram_start__) : {
        . = ALIGN(8);
        *(.text)
        *(.text*)
        *(.rodata)
        *(.rodata*)
        *(.srodata)
        *(.srodata*)

        *(.hash)
        *(.dyn*)
        *(.gnu*)
        *(.pl*)

        KEEP(*(.eh_frame))
        *(.eh_frame*)

        KEEP (*(.init))
        KEEP (*(.fini))
        . = ALIGN(8);
    } > AXI_SRAM

    .rel : {
        KEEP(*(.rel*))
    } > AXI_SRAM

    /* section information for usbh class */
    .usbh_class_info : {
        . = ALIGN(4);
        __usbh_class_info_start__ = .;
        KEEP(*(.usbh_class_info))
        __usbh_class_info_end__ = .;
        . = ALIGN(8);
    } > AXI_SRAM
```

具体有关存储区域的介绍，用户可以在先楫官网查阅应用文档“HPM6000 系列微控制器 片上 SRAM 使用指南”。

3.2.2 复数 FFT 测试结果

	ILM cache enable	ILM cache disable	AXI_SRAM cache enable	AXI_SRAM cache disable	XPI0 cache enable	XPI0 cache disable
64_fft radix-2	9255	9295	10808	79807	20557	209433
64_ifft radix-2	9573	9528	9923	88216	20135	223312
64_fft radix-2 (Q31)	4443	4323	5916	89346	34738	212696
64_ifft radix-2 (Q31)	4573	4689	5258	102211	18529	254786
64_CPU (Cooley-Tukey)	14541	14489	14495	151733	16254	269759
256_fft radix-2	47822	47724	48534	396081	46795	864244
256_ifft radix-2	48856	48761	47702	434131	47766	918071
256_fft radix-2 (Q31)	21587	20984	21437	447950	21252	826084
256_ifft radix-2 (Q31)	22507	22847	22596	523638	22445	1147580
256_CPU (Cooley-Tukey)	61130	61416	61176	616229	60335	740987
512_fft radix-2	106564	106389	106368	859878	104261	1770967
512_ifft radix-2	108288	108178	106254	952959	106219	1889769
512_fft radix-2 (Q31)	47492	47078	47661	983466	46482	1701529
512_ifft radix-2 (Q31)	49120	49535	49982	1157196	49520	2402649
512_CPU (Cooley-Tukey)	131070	131395	131084	1297328	129650	1391009
1024_fft radix-2	234646	234613	232623	1859826	230620	3607783
1024_ifft radix-2	238250	238278	234335	2069733	234296	3887344
1024_fft radix-2 (Q31)	104836	100628	103379	2133397	101970	3535237
1024_ifft radix-2 (Q31)	107794	110985	108935	2550691	109091	5028570
1024_CPU (Cooley-Tukey)	284926	285672	283441	2773958	281524	2782023

- 本次测试中结果为运行时间，单位都为 CPU 周期数
- 可见 Q31 格式的性能比浮点数格式更有优势，DSP 硬件加速比软件算法更有优势
- 在 ILM 中执行时，是否使能 cache 对计算速度影响不大
- 相同算法，当 cache 打开时，在 ILM 内部执行，与从 AXI SRAM，XPI0 连接的外部 flash 执行，效率基本相同
- 当 cache 关闭时，AXI SRAM 和 XPI0 的外部 flash 代码执行耗时增加明显，体现了不同存储器的访问开销
- 64_fft radix-2 和 64_ifft radix-2 算法由于运行时间较短，代码首次由 flash 载入 cache 耗时较大，因此即使 cache 打开，耗时增加较明显

3.3.1 实数 FFT 测试过程介绍

实数快速傅里叶变换（RFFT）和实数快速傅里叶逆变换（RIFFT）函数将由实数值组成的信号从时域变换到频域。软件生成正弦波后使用 RFFT 进行变换，然后找出正弦波幅值和频率。测试结果单位为所需 CPU 周期数，测试相关代码介绍可见 2.3.1（1）。

（1）Floating-point Real FFT/IFFT Function samples

接口调用：**hpm_dsp_rfft_f32(float32_t *src, uint32_t m);** 部分测试代码如下：

```
for (uint8_t i = 6; i <= 10; i++) {
      point = 1 << i;
      shift = i;
      init_fft_inputbuf(&fft_buf[0], point);
      clear_cycle();
     hpm_dsp_rfft_f32(&fft_buf[0], shift);
      run_times = read_cycles();
      printf("dsp rfft Total samples: %d.\r\n", point);
      printf("total times:%d tick.\r\n", run_times);
      fft_printf(&fft_buf[0], &fft_mag_output[0], point);

      clear_cycle();
      hpm_dsp_rifft_f32(&fft_buf[0], shift);
      run_times = read_cycles();
      printf("dsp rifft Total samples: %d.\r\n", point);
      printf("total times:%d tick.\r\n", run_times);
    for (uint32_t m = 0; m < 2 * point; m++) {
         if ((fft_buf_copy[m] > fft_buf[m] + FFT_PRECISION) ||
         (fft_buf_copy[m] < fft_buf[m] - FFT_PRECISION)) {
            err_num++;
         }
      }

      printf("-----------------------------------\r\n\r\n\r\n");
```

```
}
```

（2）Q31 Real FFT/IFFT Function samples

对于 Q31 格式的 RFFT 和 RIFFT 函数，用户需要在调用之前进行算术右移。接口调用：**hpm_dsp_rfft_q31(q31_t *src, uint32_t m);** 部分测试代码如下：

```
for (uint8_t j = 6; j <= 10; j++) {
      point = 1 << j;
      shift = j;
      init_fft_inputbuf(&fft_buf[0], point);
      clear_cycle();
      hpm_dsp_rfft_q31(&ffa_buf[0], shift);
      run_times = read_cycles();
      printf("dsp rfft q31 Total samples: %d.\r\n", point);
      printf("total times:%d tick.\r\n", run_times);
      hpm_dsp_convert_q31_f32(ffa_buf, fft_buf, point * 2);
      for (uint32_t i = 0; i < 2 * point; i++) {
         fft_buf[i] = fft_buf[i] * point * point * 2;
      }
      fft_printf(&fft_buf[0], &fft_mag_output[0], point);

      clear_cycle();
       hpm_dsp_rifft_q31(&ffa_buf[0], shift);
      run_times = read_cycles();
      hpm_dsp_convert_q31_f32(ffa_buf, fft_buf, point * 2);
      for (uint32_t i = 0; i < 2 * point; i++) {
         fft_buf[i] = fft_buf[i] * point;
      }
      printf("dsp rifft q31 Total samples: %d.\r\n", point);
      printf("total times:%d tick.\r\n", run_times);
   for (uint32_t m = 0; m < 2 * point; m++) {
         if ((fft_buf_copy[m] > fft_buf[m] + FFT_PRECISION) ||
         (fft_buf_copy[m] < fft_buf[m] - FFT_PRECISION)) {
            err_num++;
   }
      }
      printf("-----------------------------------\r\n\r\n\r\n");
   }
```

3.3.2 实数 FFT 测试结果

	ILM cache enable	ILM cache disable
64_rfft	5098	5105
64_rifft	5223	5206
64_rfft (Q31)	2189	2184
64_rifft (Q31)	2362	2349
256_rfft	25289	25262
256_rifft	25726	25726
256_rfft (Q31)	9843	9853
256_rifft (Q31)	10980	10940
512_rfft	37846	37893
512_rifft	38458	38466
512_rfft (Q31)	17180	17211
512_rifft (Q31)	19030	19066
1024_rfft	122405	122400
1024_rifft	124093	124078
1024_rfft (Q31)	45638	45300
1024_rifft (Q31)	51704	51581

- 本次测试中结果为运行时间，单位都为 CPU 周期数
- 在 ILM 中执行时，是否使能 cache 对计算速度影响不大
- 比上节复数 FFT 测试的速度快一倍左右

3.4.1 DSP 硬件测试点数的增加

在 SDK1.1.0 版本更新后的版本，基于 Andes DSP Library 中的硬件 dsp_fft 支持的最大点数可以到 8192 个点。由于先楫官方的 fft_perf_test 例程中，默认支持的最高采样点为 1024 点，所以如果想要测更高的点数，用户需要做一些代码上的修改。本小节以 4096 点为

例，教学用户如何添加 DSP 硬件测试的点数。测试环境为 SDK1.1.0，开发板为 HPM6750EVKMINI。

在例程中，每个 fft 测试都有一个 for 循环，代表循环计算各种点数，比如：8/16/32/64/128/256/512 /1024 点。例程循环中，默认为 i<=10，那么 2^10，代表的便是测试的最大点数为 1024 点，所以如果用户需要测到最大为 4096 点，将 i 改为 12，如下所示：

```
for (uint8_t i = 6; i <= 12; i++)
```

同时将定义部分的空间分配也同步扩大到 2*4096。由于例程会比对 ifft 的数据和 fft 变换的数据是否相一致，所以扩大测试点数后，也要将 FFT_PRECISION 精度数扩大，可以调整到 0.1。这两处代码如下所示：

```
#define FFT_PRECISION (0.1)
#define FFT_COMPLEX_MAX (2*4096)
```

改完这三处地方，运行程序就会得到最大为 4096 点的结果了(此方法不适用于 Software Cooley-tukey)，测试结果如下所示：

	ILM cache enable
4096_cfft radix-2	113205
4096_cifft radix-2	1153302
4096_cfft (Q31) radix-2	416793
4096_cifft (Q31) radix-2	504818
4096_rfft radix-2	585649
4096_rifft radix-2	592481
4096_rfft (Q31) radix-2	228319
4096_rifft (Q31) radix-2	260567

- 本次测试中结果为运行时间，单位都为 CPU 周期数
- 测试为硬件 DSP 的结果

3.4.2 软件算法测试点数的增加

本文 3.2.1（2）中介绍的 Software Cooley-tukey 这种软件算法也可以添加到 8196 点，先楫半导体 SDK 提供软件 FFT 数学算法库，如果用户想计算这种软件的点数，也需要做一些配置。(由于 SDK1.1 版本之后，硬件 DSP 已经可以实现 1024 点数以上，所以不建议软件实现，在速度上没有优势)

```
#define FFT_PRECISION (0.1)
#define FFT_COMPLEX_MAX (2*4096)
```

```
for (uint8_t i = 6; i <= 12; i++)
```

如上的三句配置方法与 3.4.1 的硬件 DSP 相同，不同的是用户还需要在中间层的数学库中新增点数的数组，默认最高只有 1024 点。打开 middleware/hpm_math/hpm_math_sw.c 后，找到如下代码片段，将下方代码拷贝到任意的在线 c 环境。

```c
#include <stdio.h>
#define FFT_MAX_POINT 10
#define FFT_MIN_POINT 2
#define BUFF_FORMAT "uint16_t"
int main()
{
    unsigned int memory[1<<FFT_MAX_POINT]={0};
    unsigned int len;
    unsigned int m;
    for(unsigned int j = FFT_MAX_POINT; j >= FFT_MIN_POINT; j--) {
        m = j;
        len = 1 << j;
        printf("%s hpm_math_sw_fft_tbl%d[%d] = {",BUFF_FORMAT,m,len);
        for (int i = 0; i < len; i++) {
            memory[i] = (memory[i >> 1] >> 1) | ((i & 1) << (m - 1));
            if(i%15 == 0){
                printf("\r\n");
            }
            printf("%d ",memory[i]);
            if(i != len-1){
                printf(",");
            }
        }
        printf("};\r\n");
    }
    printf("%s *hpm_math_sw_fft_str[%d] = {\r\n",BUFF_FORMAT,FFT_MAX_POINT);
    for(int j = 1; j <= FFT_MAX_POINT; j++) {
        len = 1 << j;
        if (j < FFT_MIN_POINT) {
            printf("NULL");
        } else {
            printf("(%s *)hpm_math_sw_fft_tbl%d",BUFF_FORMAT, j);
        }
        if(j != FFT_MAX_POINT){
            printf(",\r\n");
        }
    }
    printf("\r\n};\r\n");

   return 0;
}
```

以上代码第 64 行中，默认#define FFT_MAX_POINT 10，对应 1024 点，所以如果要改为 4096 点的话，需改为 12，对应 2^12=4096。改完之后运行这段代码，会生成一段数组，将这段数组直接替换掉如下图所示的部分，此部分代码也在 hpm_math_sw.c 中，生成数组代码的下方（箭头所指包含的区域，此图中为 111 行到 275 行）。

```
111  //uint16_t hpm_math_sw_fft_tbl10[1024] = {
112  //0 ,512 ,256 ,768 ,128 ,640 ,384 ,896 ,64 ,576 ,320 ,832 ,192 ,704 ,448 ,
113  //960 ,32 ,544 ,288 ,800 ,160 ,672 ,416 ,928 ,96 ,608 ,352 ,864 ,224 ,736 ,
114  //480 ,992 ,16 ,528 ,272 ,784 ,144 ,656 ,400 ,912 ,80 ,592 ,336 ,848 ,208 ,
115  //720 ,464 ,976 ,48 ,560 ,304 ,816 ,176 ,688 ,432 ,944 ,112 ,624 ,368 ,880 ,
116  //240 ,752 ,496 ,1008 ,8 ,520 ,264 ,776 ,136 ,648 ,392 ,904 ,72 ,584 ,328 ,
117  //840 ,200 ,712 ,456 ,968 ,40 ,552 ,296 ,808 ,168 ,680 ,424 ,936 ,104 ,616 ,
118  //360 ,872 ,232 ,744 ,488 ,1000 ,24 ,536 ,280 ,792 ,152 ,664 ,408 ,920 ,88 ,
119  //600 ,344 ,856 ,216 ,728 ,472 ,984 ,56 ,568 ,312 ,824 ,184 ,696 ,440 ,952 ,
120  //120 ,632 ,376 ,888 ,248 ,760 ,504 ,1016 ,4 ,516 ,260 ,772 ,132 ,644 ,388 ,
121  //900 ,68 ,580 ,324 ,836 ,196 ,708 ,452 ,964 ,36 ,548 ,292 ,804 ,164 ,676 ,
122  //420 ,932 ,100 ,612 ,356 ,868 ,228 ,740 ,484 ,996 ,20 ,532 ,276 ,788 ,148 ,
123  //660 ,404 ,916 ,84 ,596 ,340 ,852 ,212 ,724 ,468 ,980 ,52 ,564 ,308 ,820 ,
124  //180 ,692 ,436 ,948 ,116 ,628 ,372 ,884 ,244 ,756 ,500 ,1012 ,12 ,524 ,268 ,
125  //780 ,140 ,652 ,396 ,908 ,76 ,588 ,332 ,844 ,204 ,716 ,460 ,972 ,44 ,556 ,
126  //300 ,812 ,172 ,684 ,428 ,940 ,108 ,620 ,364 ,876 ,236 ,748 ,492 ,1004 ,28 ,
127  //540 ,284 ,796 ,156 ,668 ,412 ,924 ,92 ,604 ,348 ,860 ,220 ,732 ,476 ,988 ,
```

```
262  //0 ,2 ,1 ,3 };
263  //uint16_t *hpm_math_sw_fft_str[10] = {
264  //NULL,
265  //(uint16_t *)hpm_math_sw_fft_tbl2,
266  //(uint16_t *)hpm_math_sw_fft_tbl3,
267  //(uint16_t *)hpm_math_sw_fft_tbl4,
268  //(uint16_t *)hpm_math_sw_fft_tbl5,
269  //(uint16_t *)hpm_math_sw_fft_tbl6,
270  //(uint16_t *)hpm_math_sw_fft_tbl7,
271  //(uint16_t *)hpm_math_sw_fft_tbl8,
272  //(uint16_t *)hpm_math_sw_fft_tbl9,
273  //(uint16_t *)hpm_math_sw_fft_tbl10
274  //};
275  //#endif
276
```

替换好之后，就可以进行运算了，测试 4096 点的软件 FFT 性能测试，运算所需周期数结果如下：

	ILM cache enable
4096_CPU（Cooley-Tukey）	1394716

4 计算性能测试

先楫半导体 SDK 中的 hpm_math 库中已经集成了数学运算接口，例如正弦函数以及余弦函数等。 计算速度上，dsp 和 math 库中的性能一致，用户需要打开相应的浮点数开关，加上 include<math.h>，这样调用的都是 math 库的实现。本节将对 HPM6700 系列 MCU 的基本数学运算性能进行测试，算出每次运算所需要的机器周期数。

4.1 测试环境搭建

本次测试使用芯片型号为 HPM6750，开发板使用 HPM6750EVKMINI，芯片主频高达 816Mhz。软件包使用先楫半导体的 SKD1.1 版本，使用 SEGGER Embedded Studio for RISC-V 6.32b 进行编译调试。将基于 HPM 官方 SDK 中 hpm_math 文件下的 fft_perf.test 进行测试，生成在 ram 中。

有关如何开启硬件 DSP 以及浮点数单元，可参考 2.1 节中的介绍，在工程路径下的 CMakeLists.txt 加入相应语句，此节测试与 3.1 节配置相同，使用单精度的硬件浮点数，开启 o3 优化。

4.2 测试过程介绍

为了验证 HPM6700 系列 MCU 的的基本数学运算性能，将对以下表格中几种数学运算进行对比测试，输入表格中的计算数据，得出所需机器周期数。选取 Q31 格式，F32 格式，CPU 计算三种方式。

运算	sin	cos	arcsin	arccos	arctan	arctan2
输入	0.3492	0.3492	0.9492	0.7492	2.66951	2, 2.66951

运算	exp	log	sqrt	mag	mul	div
输入	0.3492	0.7492	0.3492	0.7492, 0.66952	0.3492, 0.7492	0.3492/0.7492

本次计算需要用到 Q31 格式的 DSP 指令进行运算，上述浮点输入数据需要被转化为相应的 Q31 格式数据，然后输入给相应的 DSP 指令。Q31 格式的运算相比 F32 格式，性能会有优势。转换之后，连续调用 math 库中运算接口 50 次，在运算结束之后，读取所用的机器周期数，得到 50 次运算周期。机器周期数代码调用与 2.3.1（1）小节相同。

以 sin 计算为例，Q31 格式的计算：首先库中找到 **hpm_dsp_convert_f32_q31(float32_t *src, q31_t *dst, uint32_t size)**；将想要输入的数据 0.3492 转化为 Q31 格式。部分代码如下：

```
typedef float FFT_BUF_TYPE;
ffa_q31_t test_buf[1];
FFT_BUF_TYPE test_conversion[1]={0.3492};
ffa_q31_t ffa_buf[1];

uint32_t run_times;
void clear_cycle(void)
{
   write_csr(CSR_MCYCLE, 0);
}
uint32_t read_cycles(void)
{
   uint32_t cycles;
   cycles = read_csr(CSR_MCYCLE);
   return cycles;
}
```

进行格式转化，重置一下周期数，接下来连续调用 50 次 Q 格式的 sin 计算接口：**hpm_dsp_sin_q31(q31_t src)**；src 为输入值（弧度），取值范围[0x800000000，0x7FFFFFFF]，并映射到范围[-π，π]。最后输出 50 次所需的机器周期数以及计算结果。主函数代码如下所示：

```
int main(void)
{
   board_init();
   board_init_led_pins();
   printf("cpu0:\t\t %dHz\n\n\n", clock_get_frequency(clock_cpu0));

    hpm_dsp_convert_f32_q31(test_conversion, ffa_buf, 1);
    clear_cycle();
  for (uint8_t i = 0; i < 50; i++) {
      test_buf[0] = hpm_dsp_sin_q31(ffa_buf[0]);
    }
   run_times = read_cycles();
    printf("total times:%d tick buf: %d.\r\n", run_times, test_buf[0]);
   while (1) {

   }
   return 0;
}
```

F32 格式的计算：同样以 sin 计算为例，调用 DSP 库中 **hpm_dsp_sin_f32(src)**;src 为 float32 数据，部分代码如下所示：

```
float32_t src = 0.3492;
float32_t dst;
int main(void)
{
   board_init();
   board_init_led_pins();
   printf("cpu0:\t\t %dHz\n\n\n", clock_get_frequency(clock_cpu0));
   clear_cycle();
    for (uint8_t i = 0; i < 50; i++) {
       dst = hpm_dsp_sin_f32(src);
}
```

而对于直接 CPU 进行的计算，不需要调用 hpm_dsp 库，使用 f32 浮点数格式进行计算，以 log_f 为例，计算直接使用 logf（src)；部分测试代码如下 ：

```
float32_t src = 0.7492;
float32_t dst;
int main(void)
{
   board_init();
   board_init_led_pins();
   printf("cpu0:\t\t %dHz\n\n\n", clock_get_frequency(clock_cpu0));
   clear_cycle();
    for (uint8_t i = 0; i < 50; i++) {
            dst = logf(src);
}
   run_times = read_cycles();
     printf("total times:%d tick buf: %d.\r\n", run_times, dst);
   while (1) {

   }
   return 0;
}
```

4.3 测试结果

测试结果中，后缀为_q31 的是使用 q31 定点格式的计算结果，后缀为_f32 的是 f32 格式的计算结果，下图其中_cpu 后缀的含义是直接使用 CPU 进行计算，不调用 HPM_DSP 库中的计算指令。

运算	sin_q31	cos_q31	arctan_q31	arctan2_q31	sqrt_q31	cmag_q31	mul_q31
结果	21.9	19.8	98.5	74.6	59.6	92.3	16

运算	cmul_q31	div_q31	sin_f32	cos_f32	arctan_f32	arctan2_f32	exp_f32
结果	48	35.9	161	174	214	81	100.1

运算	log_f32	sin_cpu	cos_cpu	arctan_cpu	arctan2_cpu
结果	85.3	161.5	174	214	270.6

运算	arcsin_cpu	arccos_cpu	exp_cpu	log_cpu
结果	201.4	213.1	176.8	133

- 本次测试中的单位都为 CPU 周期数
- 结果为 50 次计算后的均值，保留一位小数部分
- cmag 与 cmul 以 c 开头的是相应复数运算

HPM6750 DSP 除了上述基础算法，还提供了更多优秀算法，比如矩阵运算、傅里叶变换等相关的算法，为用户提供更加丰富的选择。同时需要特别指出，HPM6750 是双核高性能微控制器，上述测试内容仅使用了其中一个 DSP 核心。HPM6750 两个 DSP 核心可同时进行不同种类的运算，因此通过合理的优化，可以获得更好的运算性能。

5 总结

本文介绍了 HPM6000 系列上 DSP 库的使用，基于先楫官方 SDK 中 DSP 和 FFT 例程讲解了 DSP 有关的使用方法：如何在工程中，添加使用单双精度浮点数指令集，或是不启用硬件浮点数的方法，也介绍了 FFT 测试中如何添加更多点数的方法。如何从 SDK 中调用接口来实现功能，可以看出 DSP 库使用还是非常方便好上手的。最后在几组测试中也向用户们展示了 HPM6750 芯片的 FFT 和数学运算的性能结果。