AN12387

在 LPC5500 上使用双 CM33 内核和 PowerQuad 运行 CoreMark 基准测试

Rev. 2 — 04/2020 Application Note

目录

1 简介

LPC55S69 MCU 具有两个 Arm® Cortex®-M33 内核，每个内核的最大工作频率为 150 MHz。 还集成了两个协处理器，以增强特定用途的计算能力。 CASPER 加密协处理器是针对某些非对称加密算法的硬件加速，而 PowerQuad 硬件加速器则用于 DSP 功能的硬件加速（定点和浮点）。 本应用笔记描述了 CoreMark 测试程序的设计，以获得 LPC55S69 MCU 的最佳性能。 在演示项目中，两个 Arm Cortex-M33 内核正在运行 CoreMark 标准测试任务，而 PowerQuad 被设置为执行 FFT 计算任务。

2 在 LPC5500 上移植 coremark 基准测试项目

在 CoreMark 官方网站的原生 CoreMark 模板项目中，CoreMark 标准测量过程作为整个应用程序项目执行，并且在应用程序的整个生命周期中仅运行一次。 这种用法是有道理的，因为当用户想要衡量核心指标性能时，不会受到其他运行功能的影响。

《Cortex-M33 移植指南上的 LPC55(S)xx CoreMark》（文档 AN12284）展示了如何将 CoreMark 项目移植到 LPC5500 平台上。

在演示案例中，核心测试是在不同条件下使用不同的 PowerQuad 任务逐步运行。 修改了调用 CoreMark 过程的原始方式，使其成为可以在需要时调用的函数。

```
 95 //#if MAIN_HAS_NOARGC
 96 //MAIN_RETURN_TYPE main(void) {
 97 int coremark_start(void) {
 98     int argc=0;
 99     char *argv[1];
100 //#else
101 //MAIN_RETURN_TYPE main(int argc, char *argv[]) {
102 //void coremark_start(void) {
103 //#endif
104     ee_u16 i,j=0,num_algorithms=0;
105     ee_s16 known_id=-1,total_errors=0;
106     ee_u16 seedcrc=0;
```

图 1. 将 coremark 的原始 main()函数更改为 coremark_start()

如 图 1 所示，原始的 `main()` 函数名称在 `core_main.c` 源文件中更改为 `coremark_start()`。此功能是应用程序中核心标记任务的条目。在 *core_portme.c* 源文件中，将 *gCoreMarkDone* 标志变量插入 *Portable_fini()*函数中，以指示当前循环的 coremark 任务已完成。

```
void portable_fini(core_portable *p)
{
    ...
```

```
    gCoreMarkDone = true;
}
```

在演示项目中，每次都会调用 CoreMark 任务，如下所示：

```
gCoreMarkDone = false;
coremark_start();
while (!gCoreMarkDone)
{
}
```

注

只有 CORE0 一次又一次地执行 CoreMark 任务。 对于 CORE1，CoreMark 任务仅执行一次，因为在带有 CORE0 的各种 PowerQuad 任务的整个演示过程中，其运行条件不会改变。

3 启用双核项目

3.1 内存分配以获得最佳性能

为了获得最佳的工作性能，请以适当的方式为两个内核分配代码和数据存储区，并减少硬件系统中访问总线的仲裁。

在硬件系统图中，存储器分为多个区域，并分别连接到 AHB 总线矩阵。 使用这种设计，不同的总线主控器可以同时访问不同的存储区，而不会产生任何仲裁延迟。

对于 LPC55S69 MCU，可用存储器如 图 2 所示。

AHB port	Non-secure start address	Non-secure end address	Secure start address	Secure end address	Function [1]
0	0x0000 0000	0x0009 FFFF	0x1000 0000	0x1009 FFFF	Flash memory, on CM33 code bus. The last 20 pages (10 KB) are reserved.
	0x0300 0000	0x0301 FFFF	0x1300 0000	0x1301 FFFF	Boot ROM, on CM33 code bus.
1	0x0400 0000	0x0400 7FFF	0x1400 0000	0x1400 7FFF	SRAM X on CM33 code bus, 32 KB. SRAMX_0 (0x1400 0000 to 0x1400 0FFF) and SRAMX_1 (0x1400 4000 to 0x1400 4FFF) are used for Casper (total 8 KB). If CPU retention used in power-down mode, RAMX_3 (0x1400 7000 to 0x1400 73FF) is used (total 1 KB) by default in power API and this is user configurable within RAMX_2 and RAMX_3.
2	0x2000 0000	0x2000 FFFF	0x3000 0000	0x3000 FFFF	RAM 0 on CM33 data bus, 64 KB.
3	0x2001 0000	0x2001 FFFF	0x3001 0000	0x3001 FFFF	RAM 1 on CM33 data bus, 64 KB.
4	0x2002 0000	0x2002 FFFF	0x3002 0000	0x3002 FFFF	RAM 2 on CM33 data bus, 64 KB.
5	0x2003 0000	0x2003 FFFF	0x3003 0000	0x3003 FFFF	RAM 3 on CM33 data bus, 64 KB.
6	0x2004 0000	0x2004 3FFF	0x3004 0000	0x3004 3FFF	RAM 4 on CM33 data bus, 16 KB. SRAM4_0 (0x3004 0000 to 0x3004 0FFF), SRAM4_1 (0x3004 1000 to 0x3004 1FFF), SRAM4_2 (0x3004 2000 to 0x3004 2FFF), and SRAM4_3 (0x3004 3000 to 0x3004 3FFF) are used for PowerQuad (total 16 KB).

图 2. LPC5500 内存映射概述（部分）

以下内容适用于此演示中的 CORE0 应用程序：

- 代码保存在 FLASH 存储器中，介于 0x0000_0000 和 0x0003_FFFF 之间（共 256 KB）。 该代码在闪存中运行。
- 数据（包括堆栈和堆）保存在 0x2001_0000 和 0x2002_FFFF 之间的 RAM1 和 RAM2 内存中（总共 128 KB）。

以下内容适用于此演示中的 CORE1 应用程序：

- 代码被保存在 0x0004_0000 和 0x0009_7FFF 之间的 FLASH 存储器中（总共 352 KB）。 启动后，代码将复制到 RAMX 内存块中，并在 0x0400_0000 和 0x0400_7FFF 之间运行（总共 32 KB）。
- 数据（包括堆栈和堆）保存在 RAM3 内存中，介于 0x2003_0000 和 0x2003_FFFF 之间（总共 64 KB）。 RAM0 存储器保留共享数据（在本文档中进一步讨论）。

访问路径如 图 3 所示。

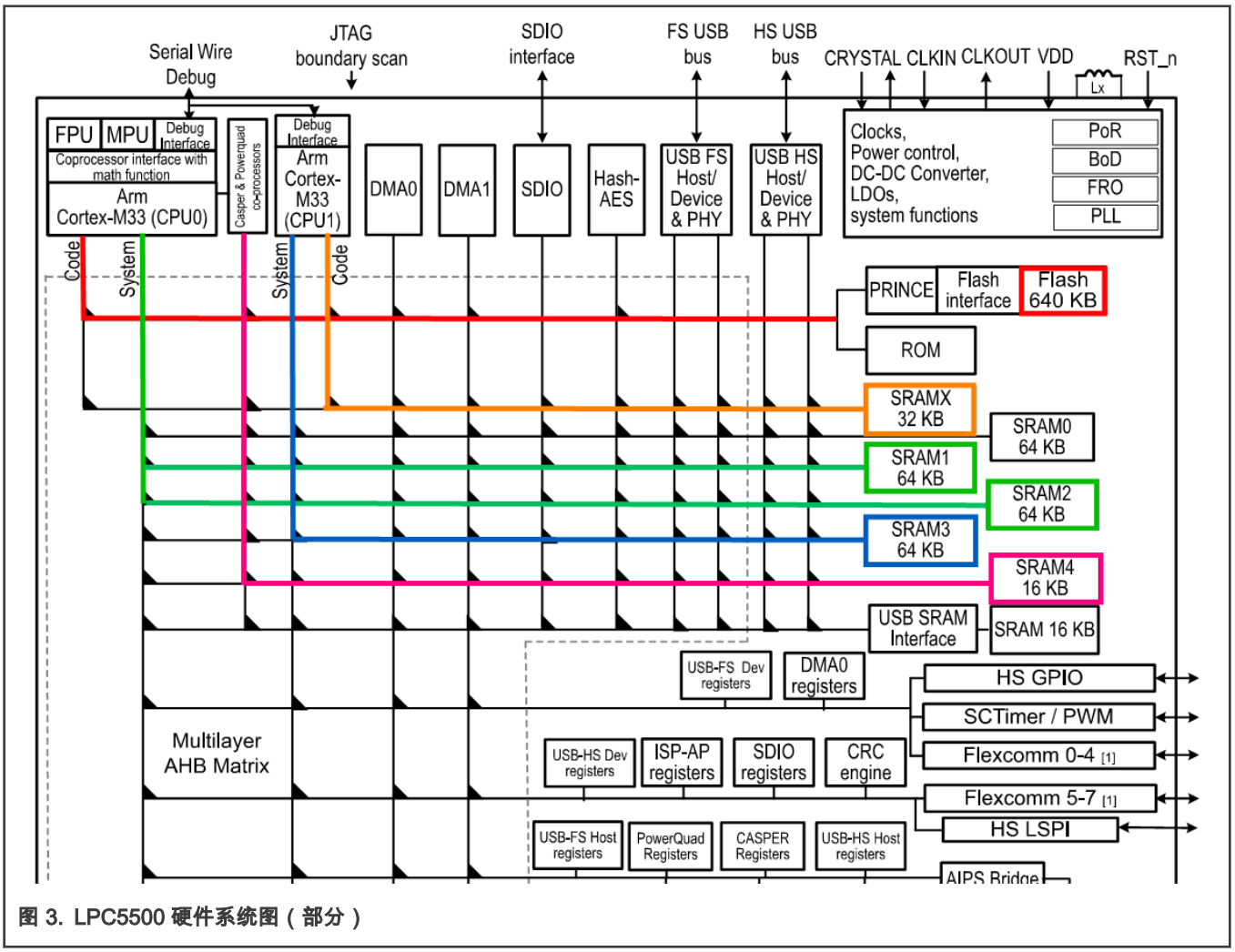

图 3. LPC5500 硬件系统图（部分）

图 3 显示了两个内核（AHB 矩阵中的主设备）不拥有任何存储区域（AHB 矩阵中的从设备）。 唯一的例外是 RAM0 内存，该内存用于在两个内核之间共享数据。

在创建项目的链接器文件时，将配置内存分配。

- CORE0 的链接器文件如下：

```
define symbol m_interrupts_start          = 0x00000000;    //FLASH
define symbol m_interrupts_end            = 0x0000013F;    //FLASH
define symbol m_text_start                = 0x00000140;    //FLASH
define symbol m_text_end                  = 0x0003FFFF;    //FLASH
define exported symbol CORE1_image_start      = 0x00040000;    //FLASH
define exported symbol CORE1_image_end    = 0x00097FFF;    //FLASH
define symbol m_xcode_start               = 0x20000000;    //RAM0
define symbol m_xcode_end                 = 0x2000FFFF;    //RAM0
```

```
define symbol m_data_start                = 0x20010000;    //RAM1-2
define symbol m_data_end                  = 0x2002FFFF;    //RAM1-2
```

- CORE1 的链接器文件如下：

```
define symbol m_interrupts_start          = 0x04000000;    //RAM-X
define symbol m_interrupts_end            = 0x0400013F;    //RAM-X
define symbol m_text_start                = 0x0400013F;    //RAM-X
define symbol m_text_end                  = 0x04007FFF;    //RAM-X
define symbol m_data_start                = 0x20030000;    //RAM3
define symbol m_data_end                  = 0x2003FFFF;    //RAM3
```

3.2 CORE0，CORE1 和 PowerQuad 任务之间的互通

芯片复位后，CORE0，CORE1 和 PowerQuad 会同时工作。 消息由 CORE0 打印。作为主核，CORE0 将工作与另一个核或协处理器同步，并将结果标记收集在一起。

CORE0 和 PowerQuad 之间的互通使用在 `task_powerquad_benchmark.c` 源文件：

```
volatile uint32_t gTaskPowerQuadCounter = 0u;
```

每当 CORE0 运行 CoreMark 任务时执行 PowerQuad 任务时，此变量都会增加。当 CORE0 完成其 CoreMark 任务时，`core_portme.c` 源文件中 CORE0 的 `Portable_fini()` 函数将收集 `gTaskPowerQuadCounter` 的值并在以后打印。

CORE0 和 CORE1 之间的互通使用 RAM0 中的共享内存。在 `core_portme.c` 源文件中，两个内核的绝对地址都定义了一些变量：

```
volatile int CORE1_coremark attribute ((section(".ARM.__at_0x20038000")));
volatile int CORE1_costtime attribute ((section(".ARM.__at_0x20038010")));
volatile int CORE1_finish_flag attribute ((section(".ARM.__at_0x20038020")));
```

完成 CORE1 核心标记任务后，它将其结果输入 `CORE1_coremark` 和 `CORE1_costtime`，并将 `CORE1_finish_flag` 标记为 **true**。CORE0 等待直到 `CORE1_finish_flag` 为 **true**，收集 `CORE1_coremark` 和 `CORE1_costtime` 中的值，然后进行打印。

4 启用 PowerQuad 任务

PowerQuad 任务在 ISR（中断服务例程）中处理，该例程定义为向量表中的 *PQ_IRQHandler()*函数。 在演示案例中，PowerQuad 有三种类型的计算任务：128 点的 FFT，256 点的 FFT 和 512 点的 FFT。

```
void App_PQTask_CFFT512Case(void)
{
    PQ_TransformCFFT(DEMO_POWERQUAD, 512u, inputData, cfftResult);
}

void App_PQTask_CFFT256Case(void)
{
     PQ_TransformCFFT(DEMO_POWERQUAD, 256u, inputData, cfftResult);
}
void App_PQTask_CFFT128Case(void)
{
     PQ_TransformCFFT(DEMO_POWERQUAD, 128u, inputData, cfftResult);
}
```

软件回调旨在使 ISR 运行不同的任务。 在启动 ISR 之前，可以将不同的 PowerQuad 任务安装到 ISR 中。

```
/* ISR for PowerQuad*/
void PQ_IRQHandler(void)
{
   uint32_t flags = POWERQUAD->INTRSTAT; /* PQ_GetStatusFlags(). */
```

```
    /* A software workaround.
     * Use the OVERFLOW flag instead of DONE flag to detect the DONE interrupt. /
    if (POWERQUAD_INTREN_INTR_OFLOW_MASK == (POWERQUAD_INTREN_INTR_OFLOW_MASK and flags) )
    {
        gTaskPowerQuadCounter++;
        //PQ_TransformCFFT(DEMO_POWERQUAD, N, inputData, cfftResult); /* start the new task. */
        if (gAppPowerQuadCallback)
        {
            (*gAppPowerQuadCallback)();
        }
     }
     POWERQUAD->INTRSTAT = flags; /* PQ_ClearStatusFlags(). */
}

void App_PQInstallCallback(void (*callback)(void))
{
    gAppPowerQuadCallback = callback;
 }
```

在该应用程序中，PowerQuad 任务被一一触发：

```
void task_pq_fft_128(void);
void task_pq_fft_256(void);
void task_pq_fft_512(void);

void (*cAppLcdDisplayPageFunc[])(void) =
{
    task_pq_fft_512,
    task_pq_fft_256,
    task_pq_fft_128 /* merge the last task into this one. */
    //task_end
};

void task_pq_fft_512(void)
{
   PRINTF("%s\r\n", func );
   ...
   App_PQInstallCallback(App_PQTask_CFFT512Case);
   gCoreMarkDone = false;
   coremark_start();
   while (!gCoreMarkDone)
   {
   }
   ...
}

void task_pq_fft_256(void)
{
    PRINTF("%s\r\n", func );
    ...
    App_PQInstallCallback(App_PQTask_CFFT256Case);
    gCoreMarkDone = false;
    coremark_start();
    while (!gCoreMarkDone)
    {
    }
    ...
}

void task_pq_fft_128(void)
```

```
{
    PRINTF("%s\r\n", func );
    ...
    App_PQInstallCallback(App_PQTask_CFFT128Case);
    gCoreMarkDone = false;
    coremark_start();
    while (!gCoreMarkDone)
    {
    }
    ...
}

int main(void)
{
    ...
    while (1)
    {
        keyValue = App_GetUserKeyValue();
        if (keyValue != keyValuePre)
        {
            App_DeinitUserKey(); /* disable detecting key when changing the lcd display. */
           (*cAppLcdDisplayPageFunc[keyValue])(); /* switch to new page on lcd module. */
            keyValuePre = keyValue;
            App_InitUserKey(); /* enable detecting key for next event. */
         }

         WFI(); /* sleep when in idle. */
      }
}
```

启动 PowerQuad 任务后，它将与运行 CoreMark 任务的 CORE0 连续运行。CORE0 在完成 CoreMark 任务后停止 PowerQuad 任务。`gTaskPowerQuadCounter` 变量用于计算 PowerQuad 任务运行的周期。 同时，此数字由 CORE0 coremark 任务计算。 然后可以确定每秒执行 PowerQuad 任务多少次，并且该数字可用作报告中的性能基准值。

5 运行项目以进行基准记录

运行演示项目，请首先构建 CORE1 项目并为 CORE1 生成映像。 然后构建 CORE0 项目，其中包括 CORE1 映像。 此后，一个映像文件包含两个内核的应用程序。

该项目有两个版本：

- LCD 显示版本：此版本使用 LCD 模块显示结果（lpc5500_coremark_dualcore_powerquad_lcd_display）。
- UART 端子版本：当没有 LCD 模块时，此版本使用 UART 端子输出结果。组装 LCD 模块时，它在 LCD 上显示相同的内容（lpc5500_coremark_dualcore_powerquad_uart_terminal）。此版本是第一个的修改。

当 LCD 显示版本工作时，结果将显示在 LCD 模块上，如 图 4 所示。

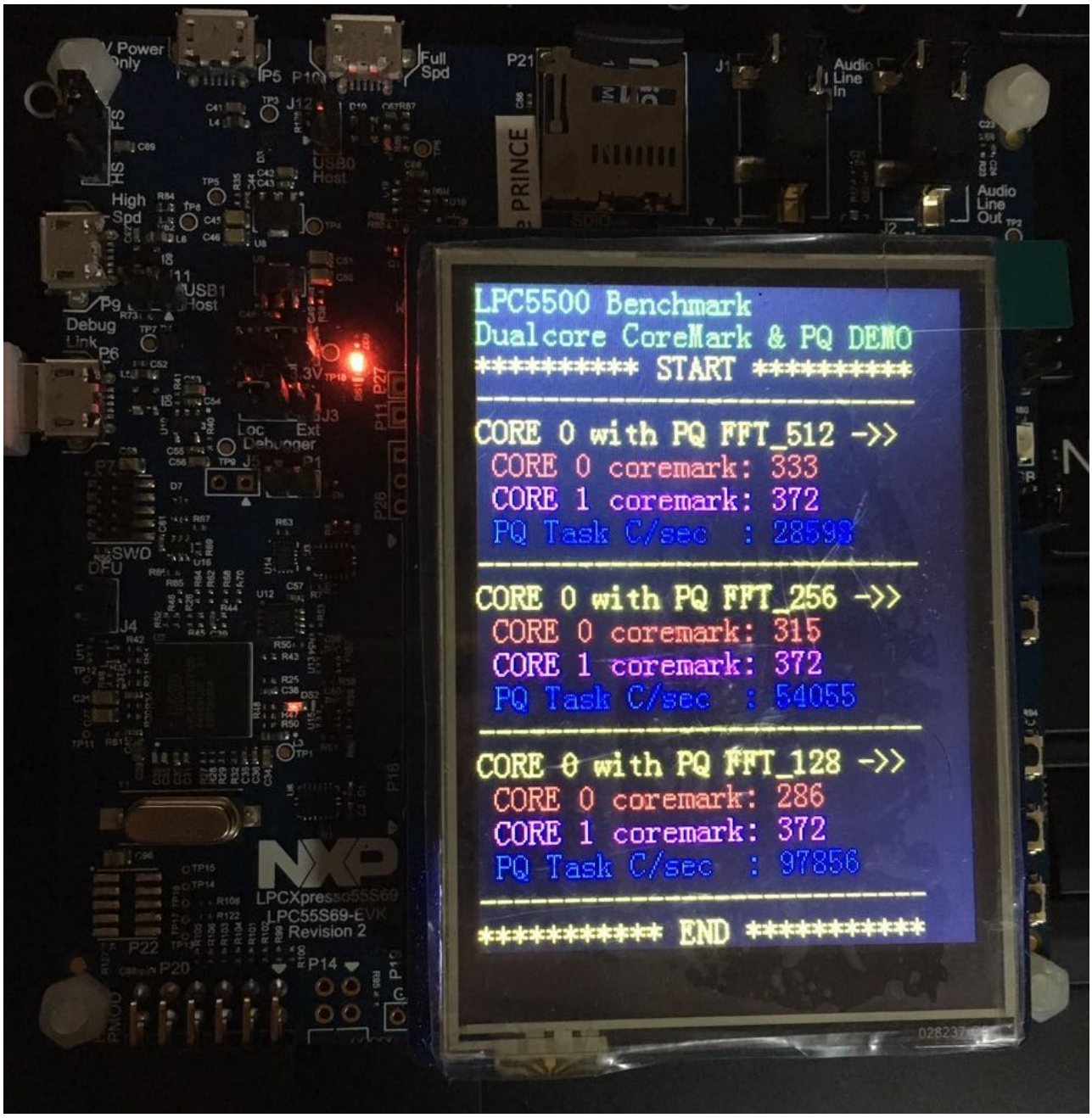

图 4. LCD 屏幕上的记录

当 UART 终端版本工作时，结果将显示在 UART 终端中，如 图 5 所示。

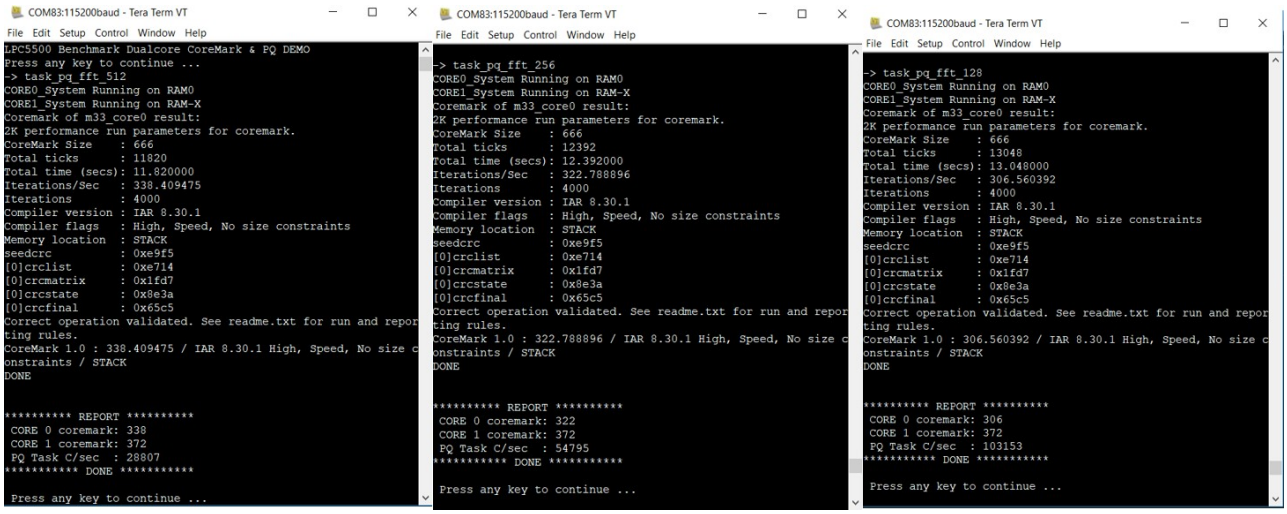

图 5. UART 终端中的记录

表 1 汇总了此信息。

表 1. 96 MHz 核心时钟时的基准摘要

Core clock = 96 MHz	CORE 0 coremark	CORE 1 coremark	PowerQuad cycles/second
DualCore + PQ FFT512	338	372	28807
DualCore + PQ FFT256	322	372	54795
DualCore + PQ FFT128	306	372	103153

当使用 150 MHz 核心时钟运行时，记录显示在 表 2 中。

表 2. 150 MHz 核心时钟时的基准摘要

Core clock = 96 MHz	CORE 0 coremark	CORE 1 coremark	PowerQuad cycles/second
DualCore + PQ FFT512	523	582	44577
DualCore + PQ FFT256	493	582	83939
DualCore + PQ FFT128	445	582	149489

CORE1 的分数高于 CORE0 的分数，因为 CORE1 的代码在 SRMAX 存储器中运行，而 CORE0 的代码在 FLASH 存储器中运行。 点数较少的 FFT 任务运行频率更高，并中断 CORE0 以在 PowerQuad 的 ISR 中重新启动任务。 当 PowerQuad 运行更多时，CORE0 的得分会降低一点。

6 修订历史

修订历史总结了自初始发行以来对该文档所做的更改。

表 3. 修订历史

版本号	日期	重要更新
1	05/2019	添加了购买此演示中使用的 LCD 模块。
2	04/2020	修改了 表 1 和 表 2。

A 购买此演示中使用的 LCD 模块

通过以下链接购买 LCD 板：

- https://www.waveshare.com/product/modules/oleds-lcds/arduino-lcd/2.8inch-tft-touch-shield.htm
- http://www.waveshare.net/shop/2.8inch-TFT-Touch-Shield.htm

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 04/2020
Document identifier: AN12387

arm