3.4 双核并行

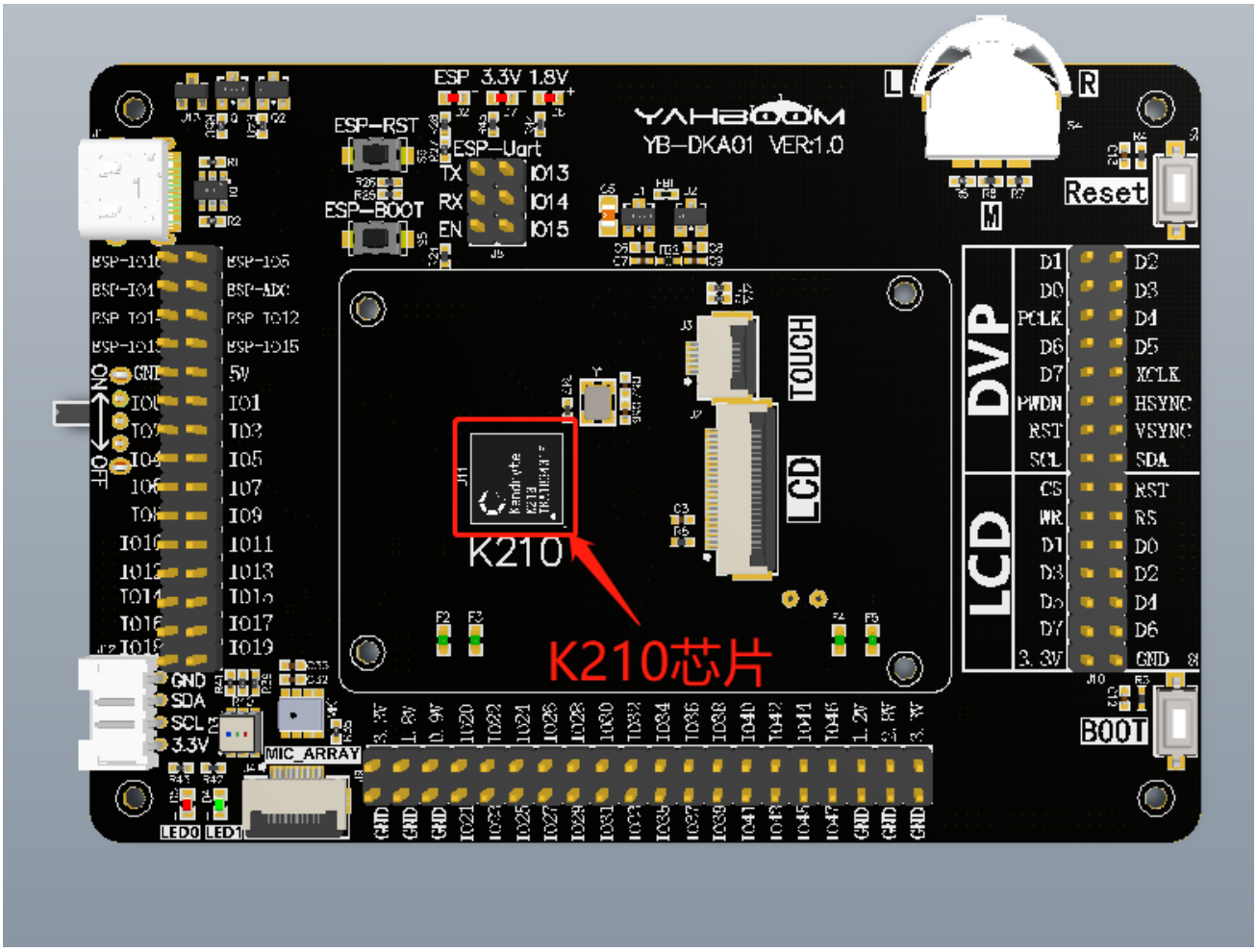
一、实验目的

本节课主要学习 K210 的核心 0 和核心 1 同时运行程序。

二、实验准备

1.实验元件

K210 芯片中的核心 0 和核心 1



2.元件特性

K210 芯片搭载基于 RISC-V ISA 的双核心 64 位的高性能低功耗 CPU，具备以下特性：

项目	内容	描述
核心数量	2 核心	双核对等，各个核心具备独立 FPU
处理器位宽	64 位	64 位 CPU 位宽，为高性能算法计算提供位宽基础，计算带宽充足
标称频率	400Mhz	频率可调，可通过调整 PLL VCO 与分频进行变频
指令集扩展	IMAFDC	基于 RISC-V 64 位 IMAFDC (RV64GC)，胜任通用任务
浮点处理单元	双精度	具备乘法器、除法器与平方根运算器，支持单精度、双精度的浮点计算
平台中断管理	PLIC	支持高级中断管理，支持 64 个外部中断源路由到 2 个核心
本地中断管理	CLINT	支持 CPU 内置定时器中断与跨核心中断
指令缓存	32KIB*2	核心 0 与核心 1 各具有 32 千字节的指令缓存，提升双核指令读取效能
数据缓存	32KIB*2	核心 0 与核心 1 各具有 32 千字节的数据缓存，提升双核数据读取效能
片上 SRAM	8MIB	共计 8 兆字节的片上 SRAM，2 兆用于 AI，6 兆通用

FPU（浮点运算单元）是集成于 CPU 中的专用于浮点运算的处理器。K210 核心 0 和核心 1 都具备独立的 FPU，满足 IEEE754-2008 标准，计算流程以流水线方式进行，具备很强的运算能力，每个 FPU 都具备除法器和平方根运算器，支持单精度和双精度浮点硬件加速运算。

4.SDK 中对应 API 功能

板级对应的头文件 bsp.h

bsp.h 头文件是与平台相关的通用函数，核之间锁的相关操作。

提供获取当前运行程序的 CPU 核编号的接口以及启动第二个核的入口。

为用户提供以下接口：

- register_core1：向核心 1 注册函数，并启动核心 1
- current_coreid：获取当前 CPU 的核心编号（0/1）
- read_cycle：获取 CPU 开机至今的时钟数。可以用使用这个函数精准的确定程序运行时钟。可以配合 sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)计算运行的时间。
- spinlock_lock:自旋锁,不可嵌套,不建议在中断使用,中断中可以使用 spinlock_trylock。
- spinlock_unlock：自旋锁解锁。
- spinlock_trylock：获取自旋锁，成功获取锁会返回 0，失败返回-1。
- corelock_lock：获取核间锁，核之间互斥的锁，同核内该锁会嵌套，只有异核之间会阻塞。不建议在中断使用该函数，中断中可以使用 corelock_trylock。
- corelock_trylock：获取核间锁，同核时锁会嵌套，异核时非阻塞。成功获取锁会返回 0，失败返回-1。
- corelock_unlock：核间锁解锁。
- sys_register_putchar：注册系统输出回调函数，printf 时会调用该函数。系统默认使用 UART3，如果需要修改 UART 则调用 uart_debug_init 函数。

• sys_register_getchar：注册系统输入回调函数，scanf 时会调用该函数。系统默认使用 UART3，如果需要修改 UART 则调用 uart_debug_init 函数。
• sys_stdin_flush：清理 stdin 缓存。
• get_free_heap_size：获取空闲内存大小。
• printk：打印核心调试信息，用户不必理会。

三、实验原理

K210 是双核 CPU，那么什么是双核 CPU 呢？双核 CPU 是在一个 CPU 中拥有两个一样功能的处理器芯片，从而提高计算能力。K210 的核心 0 和核心 1 都可以单独工作，系统默认使用核心 0，如果需要使用核心 1 需要手动开启核心 1 的服务。

四、实验过程

1. 首先进入 main 函数，读取当前的 CPU 核心编号，然后通过 printf 函数打印出来，再注册核心 1 的启动函数为 core1_main。

最后进入 while(1)循环，每个 1 秒钟打印一次 Core 0 is running。

```c
int main(void)
{
    /* 读取当前运行的核心编号 */
    uint64_t core = current_coreid();
    printf("Core %ld say: Hello yahboom\n", core);
    /* 注册核心1，并启动核心1 */
    register_core1(core1_main, NULL);

    while(1)
    {
        sleep(1);
        printf("Core %ld is running\n", core);
    }
    return 0;
}
```

2. core1_main 函数已经被注册为核心 1 的入口函数，在这里可以像 main 函数一样使用，这里同样先读取当前运行的核心编号并打印出来。再进入 while(1)

循环，每隔 0.5 秒打印一次数据。这样和核心 0 的对比就是每秒都比核心 0 多打印一次信息。

```
int core1_main(void *ctx)
{
    int state = 1;
    uint64_t core = current_coreid();
    printf("Core %ld say: Hello world\n", core);

    while(1)
    {
        msleep(500);
        if (state = !state)
        {
            printf("Core %ld is running too!\n", core);
        }
        else
        {
            printf("Core %ld is running faster!\n", core);
        }
    }
}
```

3.编译调试，烧录运行

把本课程资料中的 dual_core 复制到 SDK 中的 src 目录下，然后进入 build 目录，运行以下命令编译。

cmake .. -DPROJ=dual_core -G "MinGW Makefiles"

make

```
[100%] Linking C executable dual_core
Generating .bin file ...
[100%] Built target dual_core
PS C:\K210\SDK\kendryte-standalone-sdk-develop\build>
```

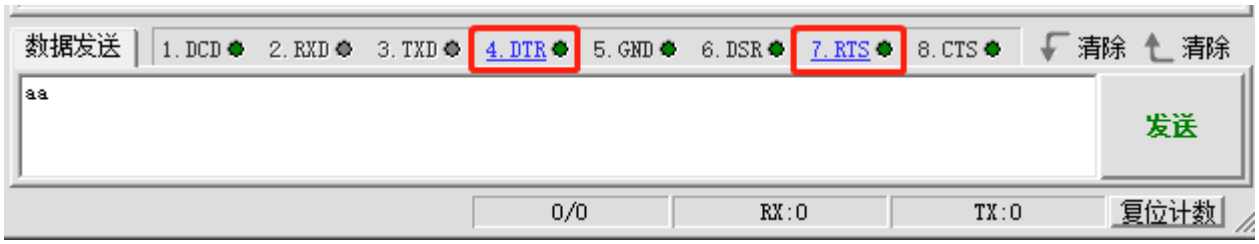
编译完成后，在 build 文件夹下会生成 dual_core.bin 文件。

使用 type-C 数据线连接电脑与 K210 开发板，打开 kflash，选择对应的设备，再将程序固件烧录到 K210 开发板上。

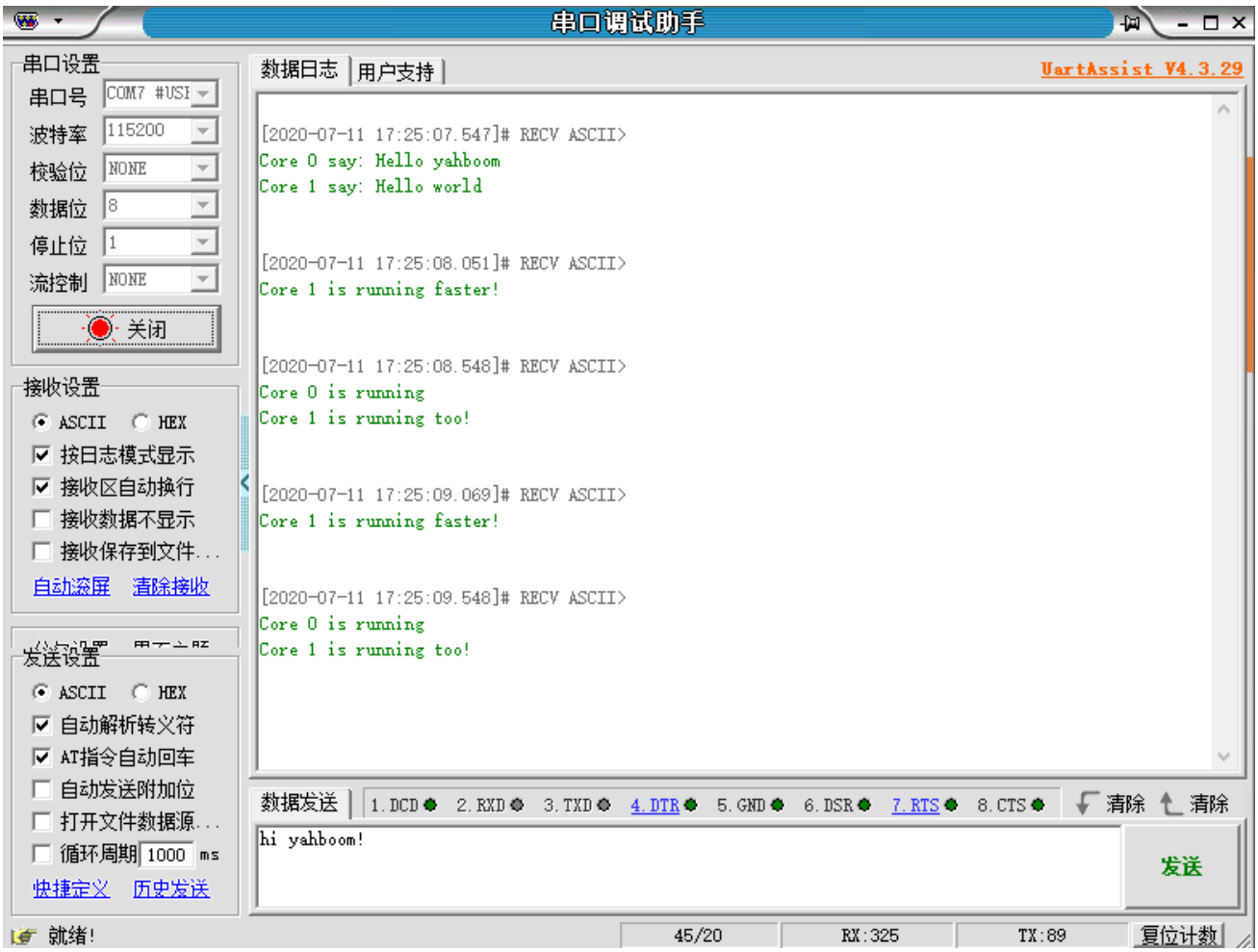
五、实验现象

烧录完成固件后，系统会弹出一个终端界面，如果没有弹出终端界面的可以打开串口助手显示调试内容。

打开电脑的串口助手，选择对应的 K210 开发板对应的串口号，波特率设置为 115200,然后点击打开串口助手。注意还需要设置一下串口助手的 DTR 和 RTS。在串口助手底部此时的 4.DTR 和 7.RTS 默认是红色的，点击 4.DTR 和 7.RTS，都设置为绿色，然后按一下 K210 开发板的复位键。



从串口助手可以接收到核心 0 和核心 1 的欢迎语,然后两个核心都进入循环，只是核心 0 每 1 秒打印一次数据，核心 1 每 0.5 秒打印一次数据。



六、实验总结

1.系统的 printf 默认使用高速串口 UARTHS（UART0）。

2.K210 是双核心 CPU，并且两个核心都可以单独运行。

3.K210 的 SDK 默认是运行核心 0，核心 1 需要手动注册开启。

附：API

对应的头文件 bsp.h

register_core1

描述

向 1 核注册函数，并启动 1 核。

函数原型

```
int register_core1(core_function func, void *ctx)
```

参数

参数名称	描述	输入输出
func	向 1 核注册的函数	输入
ctx	函数的参数，没有设置为 NULL	输入

返回值

返回值	描述
0	成功
非 0	失败

current_coreid

描述

获取当前 CPU 核编号。

函数原型

```
#define current_coreid()       read_csr(mhartid)
```

参数

无。

返回值

当前所在 CPU 核的编号。

read_cycle

描述

获取 CPU 开机至今的时钟数。可以用使用这个函数精准的确定程序运行时钟。可以配合 sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)计算运行的时间。

函数原型

```
#define read_cycle()        read_csr(mcycle)
```

参数

无。

返回值

开机至今的 CPU 时钟数。

spinlock_lock

描述

自旋锁，不可嵌套，不建议在中断使用，中断中可以使用 spinlock_trylock。

函数原型

```
void spinlock_lock(spinlock_t *lock)
```

参数

自旋锁，要使用全局变量。

返回值

无。

spinlock_trylock

描述

获取自旋锁，成功获取锁会返回 0，失败返回-1。

函数原型

```
int spinlock_trylock(spinlock_t *lock)
```

参数

自旋锁，要使用全局变量。

返回值

返回值	描述

返回值	描述
0	成功
非 0	失败

spinlock_unlock

描述

自旋锁解锁。

函数原型

```
void spinlock_unlock(spinlock_t *lock)
```

参数

核间锁，要使用全局变量，参见举例。

返回值

无。

corelock_lock

描述

获取核间锁，核之间互斥的锁，同核内该锁会嵌套，只有异核之间会阻塞。不建议在中断使用该函数，中断中可以使用 corelock_trylock。

函数原型

```
void corelock_lock(corelock_t *lock)
```

参数

核间锁，要使用全局变量，参见举例。

返回值

无。

corelock_trylock

描述

获取核间锁，同核时锁会嵌套，异核时非阻塞。成功获取锁会返回 0，失败返回-1。

函数原型

```
corelock_trylock(corelock_t *lock)
```

参数

核间锁，要使用全局变量，参见举例。

返回值

返回值	描述
0	成功
非 0	失败

corelock_unlock

描述

核间锁解锁。

函数原型

```
void corelock_unlock(corelock_t *lock)
```

参数

核间锁，要使用全局变量，参见举例。

返回值

无。

sys_register_getchar

描述

注册系统输入回调函数，scanf 时会调用该函数。系统默认使用 UART3，如果需要修改 UART 则调用 uart_debug_init 函数，具体请到 uart 章节查看该函数。

函数原型

```
void sys_register_getchar(sys_getchar_t getchar);
```

参数

参数名称	描述	输入输出
getchar	回调函数	输入

返回值

无。

sys_register_putchar

描述

注册系统输出回调函数，printf 时会调用该函数。系统默认使用 UART3，如果需要修改 UART 则调用 uart_debug_init 函数，具体请到 uart 章节查看该函数。

函数原型

```
void sys_register_putchar(sys_putchar_t putchar)
```

参数

参数名称	描述	输入输出
putchar	回调函数	输入

返回值

无。

sys_stdin_flush

描述

清理 stdin 缓存。

参数

无。

返回值

无。

get_free_heap_size

描述

获取空闲内存大小。

函数原型

```
size_t get_free_heap_size(void)
```

参数

无。

返回值

空闲内存大小。

举例

```
/* 1 核在 0 核第二次释放锁的时候才会获取到锁，通过读 cycle 计算时间 */
#include <stdio.h>
#include "bsp.h"
#include <unistd.h>
#include "sysctl.h"

corelock_t lock;

uint64_t get_time(void)
{
    uint64_t v_cycle = read_cycle();
    return v_cycle * 1000000 / sysctl_clock_get_freq(SYSCTL_CLOCK_CPU);
}

int core1_function(void *ctx)
{
    uint64_t core = current_coreid();
    printf("Core %ld Hello world\n", core);
    while(1)
    {
        uint64_t start = get_time();
        corelock_lock(&lock);
        printf("Core %ld Hello world\n", core);
        sleep(1);
        corelock_unlock(&lock);
```

```
        uint64_t stop = get_time();
        printf("Core %ld lock time is %ld us\n",core, stop - start);
        usleep(10);
    }
}

int main(void)
{
    uint64_t core = current_coreid();
    printf("Core %ld Hello world\n", core);
    register_core1(core1_function, NULL);
    while(1)
    {
        corelock_lock(&lock);
        sleep(1);
        printf("1> Core %ld sleep 1\n", core);
        corelock_lock(&lock);
        sleep(2);
        printf("2> Core %ld sleep 2\n", core);
        printf("2> Core unlock\n");
        corelock_unlock(&lock);
        sleep(1);
        printf("1> Core unlock\n");
        corelock_unlock(&lock);
        usleep(10);
    }
}
```

数据类型

相关数据类型、数据结构定义如下：

- core_function：CPU 核调用的函数。
- spinlock_t：自旋锁。
- corelock_t：核间锁。

core_function

描述

CPU 核调用的函数。

定义

```
typedef int (*core_function)(void *ctx);
```

spinlock_t

自旋锁。

定义

```
typedef struct _spinlock
{
    int lock;
} spinlock_t;
```

corelock_t

核间锁。

定义

```
typedef struct _corelock
{
    spinlock_t lock;
    int count;
    int core;
} corelock_t;
```