3.4 Dual-core parallel

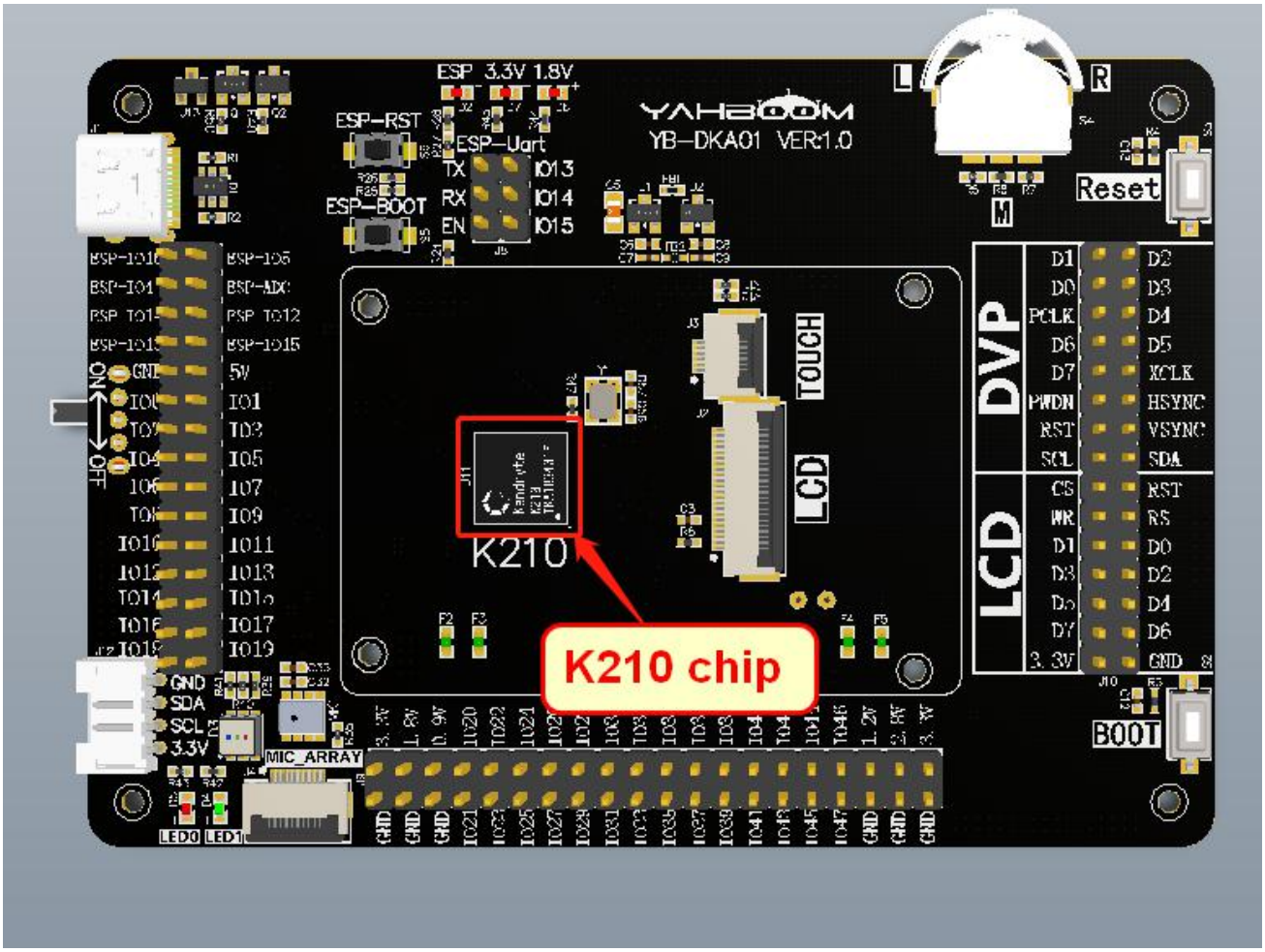
1. Experiment purpose

In this lesson, we mainly learns the core 0 and core 1 of K210 to run programs at the same time.

2.Experiment preparation

2.1 components

core 0 and core 1 of K210.



2.2 Component characteristics

The K210 chip is equipped with a dual-core 64-bit high-performance low-power CPU based on RISC-V ISA. It possess the following features:

Project	Content	Description
Number of core	2 cores	Dual-core equivalence, each core with independent FPU
Processor bit width	64bit	The 64-bit CPU bit width provides a bit width basis for high-performance algorithm calculations and sufficient computing bandwidth
Nominal frequency	400Mhz	Adjustable frequency, can be converted by adjusting PLL VCO and frequency division
Instruction set extension	IMAFDC	Based on RISC-V 64-bit IMAFDC (RV64GC), can be used for general tasks
FPU	Double precision	Equipped with multiplier, divider and square root arithmetic, supporting single-precision and double-precision floating point calculations
Platform interruption management	PLIC	Support advanced interrupt management, support 64 external interrupt sources to be routed to 2 cores
Local interrupt management	CLINT	Support CPU built-in timer interrupt and cross-core interrupt
Instruction cache	32KIB*2	Core 0 and Core 1 each have a 32-kilobyte instruction cache to improve dual-core instruction read performance
Data cache	32KIB*2	Core 0 and Core 1 each have a 32-kilobyte data cache to improve dual-core data read performance
On-chip SRAM	8MIB	A total of 8 megabytes of on-chip SRAM, 2 megabytes for AI, 6 megabytes for general using

FPU (Floating Point Operation Unit) is a processor dedicated to floating point operations integrated in the CPU. K210 core 0 and core 1 have independent FPUs, which meet the IEEE754-2008 standard. The calculation process is carried out in a pipeline manner and has strong computing capabilities. Each FPU has a divider and a square root arithmetic unit, and supports single precision and double precision floating point hardware acceleration operation.

2.3 SDK API function

The header file is bsp.h

The bsp.h header file is a general function related to the platform, and the related operations of the lock between the cores.

It be used to provide an interface to obtain the CPU core number of the currently running program and an entry to start the second core.

We will provide the following ports for users.

• register_core1: Register functions to core 1, and start core 1

• current_coreid: Get the core number of the current CPU (0/1)

• read_cycle: Obtain the number of clocks since the CPU was turned on. You can use this function to accurately determine the program running clock. Can cooperate with sysctl_clock_get_freq

(SYSCTL_CLOCK_CPU) to calculate the running time.
• spinlock_lock: spin lock, cannot be nested, it is not recommended to use it in interrupts, spinlock_trylock can be used in interrupts.
• spinlock_unlock: Unlock the spinlock.
• spinlock_trylock: Acquire a spin lock. If the lock is acquired successfully, it will return 0, and if it fails, it will return -1.
• corelock_lock: Acquire inter-core locks, mutually exclusive locks between cores. The locks within the same core will be nested, and only between different cores will block. It is not recommended to use this function in interrupts. Corelock_trylock can be used in interrupts.
• corelock_trylock: Acquire inter-core locks. The locks will be nested when the core is the same, and non-blocking when the core is different. It returns 0 if the lock is acquired successfully, and -1 if it fails.
• corelock_unlock: unlock the inter-core lock.
• sys_register_putchar: Register system output callback function, which will be called during printf. The system uses UART3 by default. If you need to modify the UART, call the uart_debug_init function.
• sys_register_getchar: Register system input callback function, which will be called during scanf. The system uses UART3 by default. If you need to modify the UART, call the uart_debug_init function.
• sys_stdin_flush: Clear the stdin cache.
• get_free_heap_size: Get the free memory size.
• printk: Print the core debugging information.

3. Experimental principle

Dual-core CPU refers to one CPU having two processor chips with the same function, which can improve computing power. For k210, both core 0 and core 1 can work independently. The system uses core 0 by default. If you need to use core 1, you need to manually enable core 1 services.

4. Experiment procedure

4.1 First, enter the main function, read the current CPU core number.
Then, print it out through the printf function.
Next, register the core 1 start up function --- core1_main.
Finally, enter the while(1) loop, and print Core 0 is running every 1 second.

```c
int main(void)
{
    /* Read the currently running core number */
    uint64_t core = current_coreid();
    printf("Core %ld say: Hello yahboom\n", core);
    /* Register core 1, and start up core 1*/
    register_core1(core1_main, NULL);

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

4.2 The core1_main function has been registered as the entry function of core 1. It can be used like the main function.
First, read the current CPU core number and printf it.
Then, enter the while(1) loop, and print date every 0.5 second.
This contrast with core 1 is that it prints information once more than core 0 every second.

```c
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);
        }
    }
}
```

4.3 Compile and debug, burn and run
Copy the dual_core folder to the src directory in the SDK.

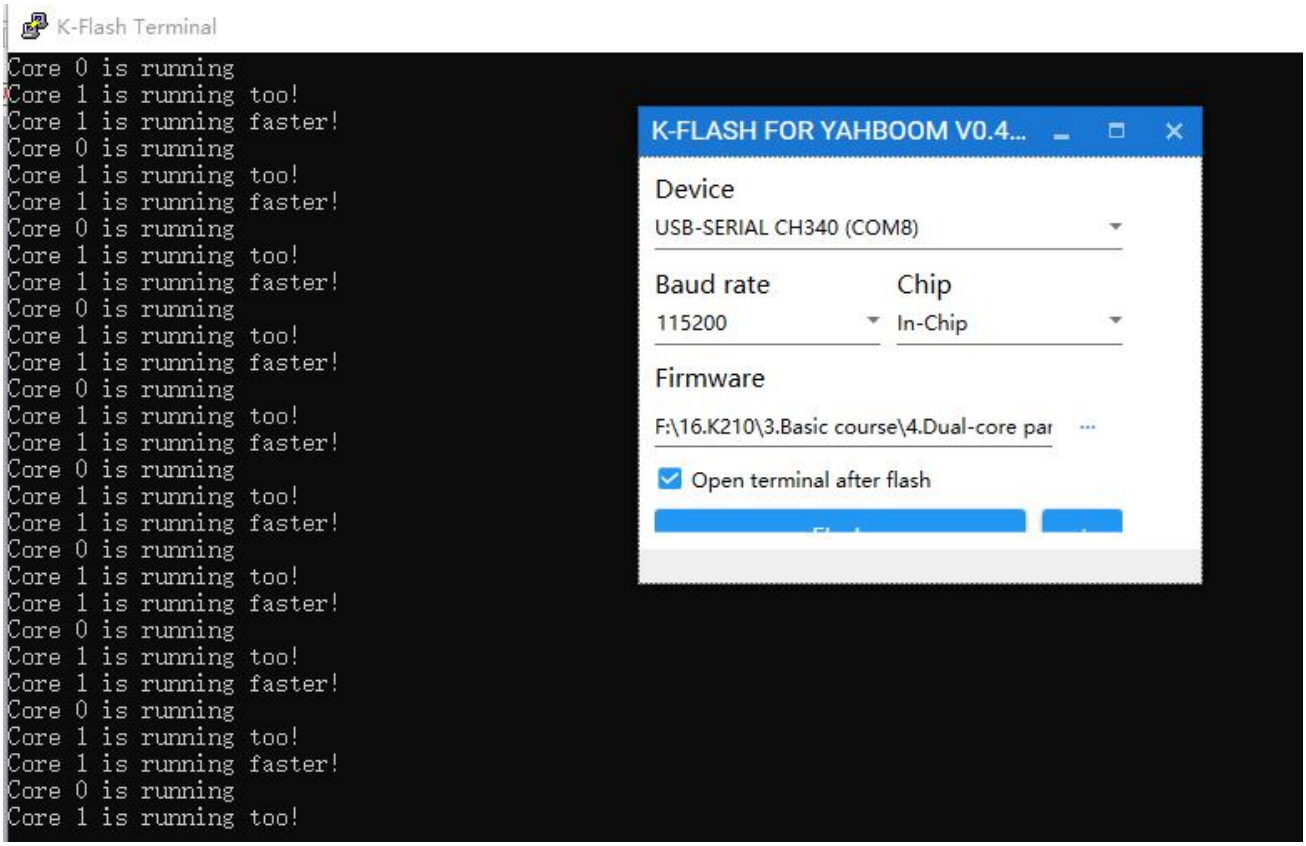
cmake .. -DPROJ=dual_core -G "MinGW Makefiles"
make

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

After the compilation is complete, the uart.bin file will be generated in the build folder.
We need to use the type-C data cable to connect the computer and the K210 development board. Open kflash, select the corresponding device, and then burn the dual_core.bin file to the K210 development board.

5. Experimental phenomenon

After the firmware is burned, a terminal interface will pop up. If the terminal interface does not pop up, we can open the serial port assistant to display the debugging content.



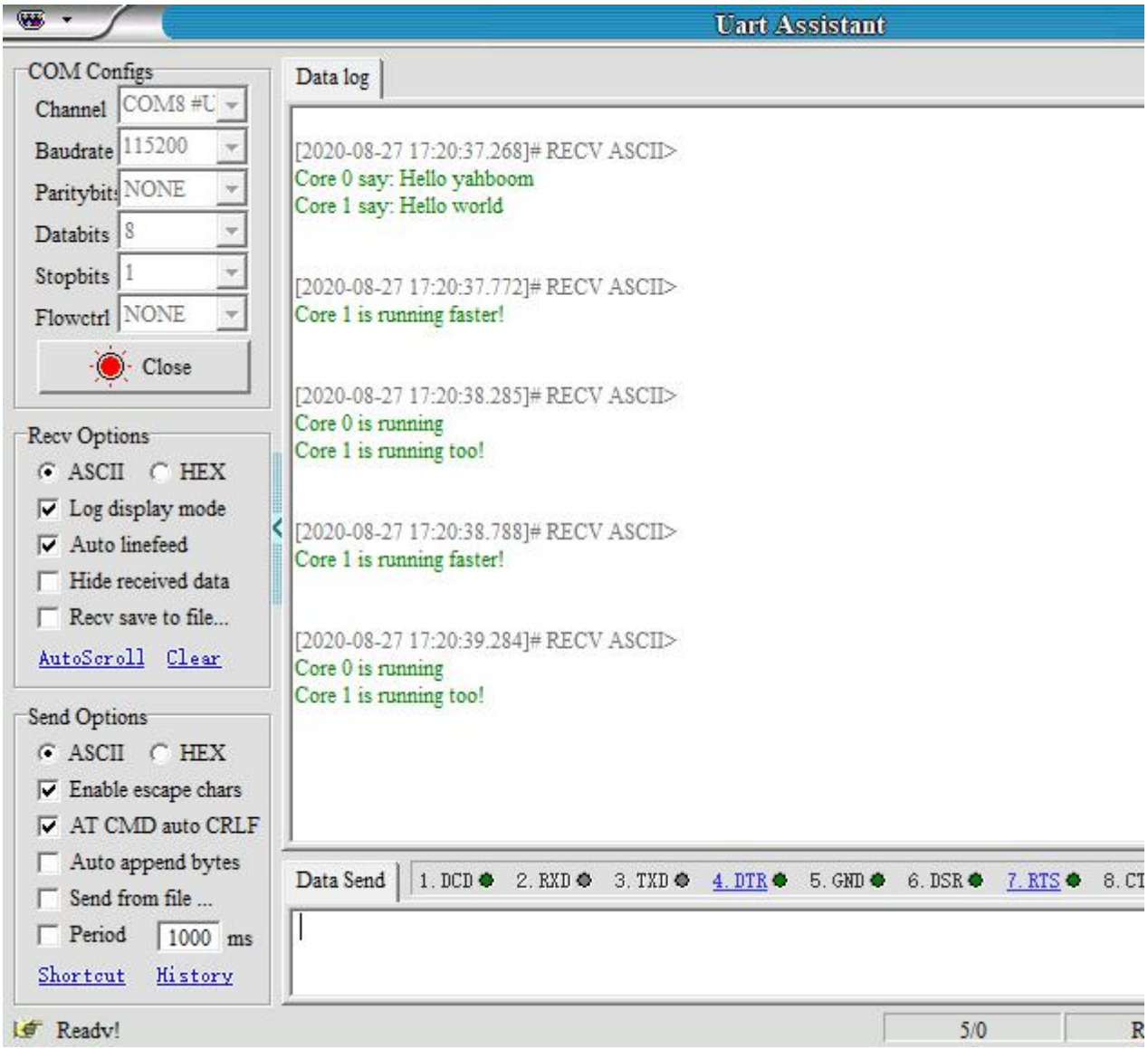
Open the serial port assistant of the computer, select the corresponding serial port number of the corresponding K210 development board, set the baud rate to 115200, and then click to open the serial port assistant.

! Note:

We also need to set the DTR and RTS of the serial port assistant.

At the bottom of the serial port assistant, we can see that 4.DTR and 7.RTS are red by default. Click 4.DTR and 7.RTS to set both to green, and then **press the reset button of the K210 development board.**

From the serial port assistant, we can receive the welcome words of core 0 and core 1,
Then both cores enter the loop, core 0 prints data every 1 second, core 1 prints data every 0.5 seconds.



6. Experiment summary

6.1 The system printf uses high-speed serial port UARTHS(UART0) by default.

6.2 K210 is a dual-core CPU, and both cores can run independently.

6.3 The SDK of K210 runs core 0 by default, and core 1 needs to be manually registered and enabled.

Appendix -- API

Header file is bsp.h

register_core1

Description: Register the function with the core 1 and start the core 1.

Function prototype: **int register_core1(core_function func, void *ctx)**

Parameter:

Parameter name	Description	Input/Output
func	Functions registered with the 1 kernel	Input
ctx	Function parameter, not set to NULL	Input

Return value:

Return value	Description
0	succeed
!0	fail

current_coreid

Description: Gets the current CPU core number.

Function prototype: #define current_coreid() read_csr(mhartid)

Parameter: No

Return value: The number of the CURRENT CPU core.

read_cycle

Description: Get the number of clocks since the CPU has been powered on. You can use this function to accurately determine the program running clock. You also can cooperate with sysctl_clock_get_freq (SYSCTL_CLOCK_CPU) to calculate the running time.

Function prototype: #define read_cycle() read_csr(mcycle)

Parameter: No

Return value: clocks number of CPU since the CPU has been powered on

spinlock_lock

Description: Spin locks cannot be nested. It is not recommended to use them in interrupts. Spinlock_trylock can be used in interrupts.

Function prototype: **void spinlock_lock(spinlock_t *lock)**

Parameter: Inter-kernel locking, using global variables.

Return value: No

spinlock_trylock
Description: Acquire a spin lock, if it succeeds, it returns 0 if it succeeds, and returns -1 if it fails.
Function prototype: **int spinlock_trylock(spinlock_t *lock)**
Parameter: Spin locks, need to use global variables.

Return value

Return value	Description
0	succeed
!0	fail

spinlock_unlock
Description: Spin lock unlocks.
Function prototype: **void spinlock_unlock(spinlock_t *lock)**
Parameter: Interkernel locking, using global variables.
Return value: No

corelock_lock
Description: Acquire inter-core locks and mutually exclusive locks between cores. The locks in the same core will be nested, and only the different cores will block. It is not recommended to use this function in interrupts. **Corelock_trylock** can be used in interrupts.
Function prototype: **void corelock_lock(corelock_t *lock)**
Parameter: Inter-kernel locking, using global variables.
Return value: No

corelock_trylock
Description: Acquire inter-core locks. The locks will be nested when the core is the same. Non-blocking when the core is different. If the lock is acquired successfully, returns 0. If it fails, return -1.
Function prototype: **corelock_trylock(corelock_t *lock)**
Parameter: Inter-core locks use global variables.

Return value

Return value	Description
0	succeed
!0	fail

corelock_unlock

Description: Internuclear lock unlocks.
Function prototype: **void corelock_unlock(corelock_t *lock)**
Parameter: Interkernel locking, using global variables.
Return value: No

sys_register_getchar
Description:
Register the system input callback function, which will be called during use scanf. The system uses UART3 by default. If you need to modify the UART, call the uart_debug_init function. About details, please go to the uart chapter to view this function.

Function prototype: **void sys_register_getchar(sys_getchar_t getchar)**
Parameter

Parameter name	Description	Input/Output
getchar	callback function	Input

Return value: No

sys_register_putchar
Description: Register system output callback function, which will be called when printf. The system uses UART3 by default. If you need to modify the UART, call the uart_debug_init function. For details, please go to the uart chapter to view this function.

Function prototype: **void sys_register_putchar(sys_putchar_t putchar)**

Parameter

Parameter name	Description	Input/Output
putchar	callback function	Input

Return value: No

sys_stdin_flush
Description: Clear the stdin cache.
Parameter: No
Return value: No

get_free_heap_size
Description: Gets the free memory size.
Function prototype: **size_t get_free_heap_size(void)**
Parameter: No
Return value: Free memory size.

Eg:

/*The 1 core will only acquire the lock when the 0 core releases the lock for the second time. The time is calculated by reading cycle*/

```
#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);
        }
}
```

Data type

The related data types and data structures are defined as follows:

core_function: A function called by the CPU core.

spinlock_t: spin lock

corelock_t: Internuclear lock.

core_function

Description: A function called by the CPU core.

Define

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

spinlock_t

spin lock

Define

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

corelock_t

Internuclear lock.

Define

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