

1 实验项目

1.1 项目名称

Hello, miniEuler

1.2 实验目的

`print` 函数是学习几乎任何一种软件开发语言时最先学习使用的函数，同时该函数也是最基本和原始的程序调试手段，但该函数的实现却并不简单。本实验的目的在于理解操作系统与硬件的接口方法，并实现一个可打印字符的函数（非系统调用），用于后续的调试和开发。

1.3 实验资源

实验指导书 <https://os2024lab.readthedocs.io/zh-cn/latest/>

2 实验任务

2.1 了解 virt 机器

操作系统介于硬件和应用程序之间，向下管理硬件资源，向上提供应用编程接口。设计并实现操作系统需要熟悉底层硬件的组成及其操作方法。

本系列实验都会在 QEMU 模拟器上完成，首先来了解一下模拟的机器信息。可以通过下列两种方法：

1. 查看 QEMU 关于 virt 的描述，或者查看 QEMU 的源码，如 github 上的 `virt.h` 和 `virt.c`。
`virt.c` 中可见如下有关内存映射的内容。

2. 通过 QEMU 导出设备树

```
$ sudo apt-get install device-tree-compiler
```

```
$ qemu-system-aarch64 -machine virt,dumpdtb=virt.dtb -cpu cortex-a53 -nographic
```

```
$ dtc -I dtb -O dts -o virt.dts virt.dtb
```

备注：-machine virt 指明机器类型为 virt，这是 QEMU 仿真的虚拟机器。

virt.dtb 转换后生成的 virt.dts 中可找到如下内容：

```
307     pl011@9000000 {
308         clock-names = "uartclk\0apb_pclk";
309         clocks = <0x8000 0x8000>;
310         interrupts = <0x00 0x01 0x04>;
311         reg = <0x00 0x9000000 0x00 0x1000>;
312         compatible = "arm,pl011\0arm,primecell";
313     };

383     chosen {
384         stdout-path = "/pl011@9000000";
385         rng-seed = <0x2418fe85 0x75d55479 0xa0d008cb 0x9a808a76 0xd00a0af3 0x4b15e7bb 0x951c35ff 0xce19dcf5>;
386         kaslr-seed = <0xe04b0c24 0x7fc7cc44>;
387     };
388 };
389
```

由上可以看出，virt 机器包含有 pl011 的设备，该设备的寄存器在 0x9000000 开始处。pl011 实际上是一个 UART 设备，即串口。可以看到 virt 选择使用 pl011 作为标准输出，这是因为与 PC 不同，大部分嵌入式系统默认情况下并不包含 VGA 设备。

```
$ sudo apt-get update
```

```
$ sudo apt-get install qemu
```

```
$ sudo apt-get install qemu-system
```

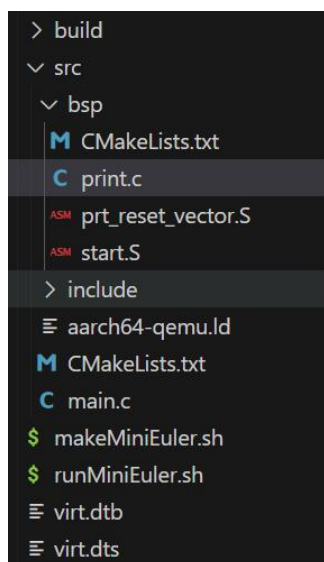
（三）安装 CMake

```
$ sudo apt-get install cmake
```

2.2 项目构建

1.实现 PRT_Printf 函数

（一）新建 src/bsp/print.c 文件



1. 宏定义

在 `print.c` 中包含所需头文件，并定义后续将会用到的宏

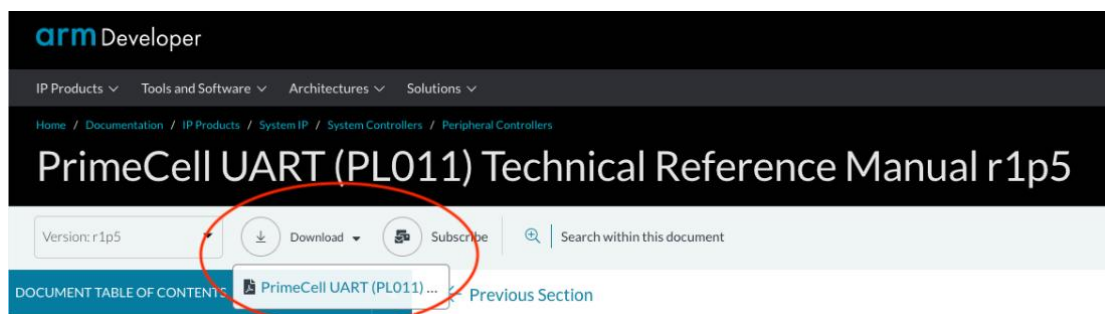
```

lab1 > src > bsp > C print.c > UART_REG_WRITE(value, addr)
1  #include <stdarg.h>
2  #include "prt_typedef.h"
3
4  #define UART_0_REG_BASE 0x09000000 // pl011 设备寄存器地址
5  // 寄存器及其位定义参见: https://developer.arm.com/documentation/ddi0183/g/programmers-model/summary-of-registers
6  #define DW_UART_THR 0x00 // UARTDR(Data Register) 寄存器
7  #define DW_UART_FR 0x18 // UARTFR(Flag Register) 寄存器
8  #define DW_UART_LCR_HR 0x2c // UARTLCR_H(Line Control Register) 寄存器
9  #define DW_XFIFO_NOT_FULL 0x020 // 发送缓冲区满置位
10 #define DW_FIFO_ENABLE 0x10 // 启用发送和接收FIFO
11
12 #define UART_BUSY_TIMEOUT 1000000
13 #define OS_MAX_SHOW_LEN 0x200
14
15
16 #define UART_REG_READ(addr) (*(volatile U32 *)((uintptr_t)addr)) // 读设备寄存器
17 #define UART_REG_WRITE(value, addr) (*(volatile U32 *)((uintptr_t)addr) = (U32)value // 写设备寄存器

```

2. 串口的初始化

如何操作硬件通常需要阅读硬件制造商提供的技术手册。如 pl011 串口设备（PrimeCell UART）是 arm 设计的，其技术参考手册可以通过其官网查看。也可以通过顶部的下载链接下载 pdf 版本，如下图所示。



依据之前 `virt.dts` 中的描述，pl011 的寄存器在 virt 机器中被映射到了 `0x9000000` 的内存位置。通过访问 pl011 的技术参考手册中的“Chapter 3. Programmers Model”中的“Summary of registers”一节可知，第 0 号寄存器是 pl011 串口的数据寄存器，用于数据的收发。其详细描述参见这里。注意到我们只是向 UART0 写入，而没从 UART0 读出（如果读出，会读出其他设备通过串口发送过来的数据，而不是刚才写入的数据，注意体会这与读写内存时是不一样的，详情参见 pl011 的技术手册），编译器在优化时可能对这部分代码进行错误的优化，如把这些操作都忽略掉。在 `UART_REG_READ` 宏和 `UART_REG_WRITE` 宏中使用 `volatile` 关键字的目的是告诉编译器，这些读取或写入有特定目的，不应将其优化（也就是告诉编译器不要瞎优化，这些写入和读出都有特定用途。如连续两次读，编译器可能认为第二次读就是前次的值，所以优化掉第二次读，但对外设寄存器的连续读可能返回不同的值。再比如写，编译器可能认为写后没有读所以写没有作用，或者连续的写会覆盖前面的写，但对外设而言对这些寄存器的写入都有特定作用）。

```
19 //串口的初始化
20 U32 PRT_UartInit(void)
21 {
22     U32 result = 0;
23     U32 reg_base = UART_0_REG_BASE;
24     // LCR寄存器: https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions/line-control-
25     result = UART_REG_READ((unsigned long)(reg_base + DW_UART_LCR_HR));
26     UART_REG_WRITE(result | DW_FIFO_ENABLE, (unsigned long)(reg_base + DW_UART_LCR_HR)); // 启用 FIFO
27
28     return OS_OK;
29 }
30
```

3. 往串口发送字符

```
33 // 读 reg_base + offset 寄存器的值。uartno 参数未使用
34 S32 uart_reg_read(S32 uartno, U32 offset, U32 *val)
35 {
36     S32 ret;
37     U32 reg_base = UART_0_REG_BASE;
38
39
40     *val = UART_REG_READ((unsigned long)(reg_base + offset));
41     return OS_OK;
42 }
43
44 // 通过检查 FR 寄存器的标志位确定发送缓冲是否满，满时返回1.
45 S32 uart_is_txfifo_full(S32 uartno)
46 {
47     S32 ret;
48     U32 usr = 0;
49
50     ret = uart_reg_read(uartno, DW_UART_FR, &usr);
51     if (ret) {
52         return OS_OK;
53     }
54
55     return (usr & DW_XFIFO_NOT_FULL);
56 }
57
58 // 往 reg_base + offset 寄存器中写入值 val。
59 void uart_reg_write(S32 uartno, U32 offset, U32 val)
60 {
61     S32 ret;
62     U32 reg_base = UART_0_REG_BASE;
63
64     UART_REG_WRITE(val, (unsigned long)(reg_base + offset));
65     return;
66 }
```

```
67
68 // 通过轮询的方式发送字符到串口
69 void uart_poll_send(unsigned char ch)
70 {
71
72     S32 timeout = 0;
73     S32 max_timeout = UART_BUSY_TIMEOUT;
74
75     // 轮询发送缓冲区是否满
76     int result = uart_is_txfifo_full(0);
77     while (result) {
78         timeout++;
79         if (timeout >= max_timeout) {
80             return;
81         }
82         result = uart_is_txfifo_full(0);
83     }
84
85     // 如果缓冲区没满, 通过往数据寄存器写入数据发送字符到串口
86     uart_reg_write(0, DW_UART_THR, (U32)(U8)ch);
87     return;
88 }
89
90 // 轮询的方式发送字符到串口, 且转义换行符
91 void TryPutc(unsigned char ch)
92 {
93     uart_poll_send(ch);
94     if (ch == '\n') {
95         uart_poll_send('\r');
96     }
97 }
```

上面的代码很简单, 就是通过轮询的方式向 PL011 的数据寄存器 DR 写入数据即可实现往串口发送字符, 实现字符输出。

4. 支持格式化输出

```
98 //支持格式化输出
99 extern int vsnprintf_s(char *buff, int buff_size, int count, char const *fmt, va_list arg);
100 int TryPrintf(const char *format, va_list vaList)
101 {
102     int len;
103     char buff[OS_MAX_SHOW_LEN];
104     for(int i = 0; i < OS_MAX_SHOW_LEN; i++) {
105         buff[i] = 0;
106     }
107     char *str = buff;
108
109     len = vsnprintf_s(buff, OS_MAX_SHOW_LEN, OS_MAX_SHOW_LEN, format, vaList);
110     if (len == -1) {
111         return len;
112     }
113
114     while (*str != '\0') {
115         TryPutc(*str);
116         str++;
117     }
118
119     return OS_OK;
120 }
121
122 U32 PRT_Printf(const char *format, ...)
123 {
124     va_list vaList;
125     S32 count;
126
127     va_start(vaList, format);
128     count = TryPrintf(format, vaList);
129     va_end(vaList);
130
131     return count;
132 }
```

为了实现与 C 语言中 `printf` 函数类似的格式化功能，我们要用到可变参数列表 `va_list`。而真正实现格式化控制转换的函数是 `vsnprintf_s` 函数。

2.实现 `vsnprintf_s` 函数

新建 `src/bsp/vsnprintf_s.c` 实现 `vsnprintf_s` 函数

`vsnprintf_s` 函数的主要作用是依据格式控制符将可变参数列表转换成字符列表写入缓冲区。

UniProton 提供了 `libboundscheck` 库，其中实现了 `vsnprintf_s` 函数，作为可选作业你可以试着使用 `libboundscheck` 库中的 `vsnprintf_s` 函数。简单起见，我们从另一个国产实时操作系统 RT-Thread 的 `kservice.c` 中引入了一个实现并进行了简单修改。可以在这里下载 `vsnprintf_s.c`。

3.调用 `PRT_Printf` 函数

`main.c` 修改为调用 `PRT_Printf` 函数输出信息。

```
lab2 > src > C main.c > main(void)
1  #include "prt_typedef.h"
2
3  extern U32 PRT_Printf(const char *format, ...);
4  extern void PRT_UartInit(void);
5
6  S32 main(void)
7  {
8      PRT_UartInit();
9
10     PRT_Printf("Test PRT_Printf int format %d \n\n", 10);
11 }
```

4.将新增文件纳入构建系统

修改 `src/bsp/CMakeLists.txt` 文件加入新增文件 `print.c` 和 `vsnprintf_s.c`

```
lab2 > src > bsp > M CMakeLists.txt
1  set(SRCS start.S prt_reset_vector.S print.c vsnprintf_s.c)
2  add_library(bsp OBJECT ${SRCS}) # OBJECT类型只编译生成.o目标文件，但不实际链接成库
3  |
```


5.启用 FPU

构建项目并执行发现程序没有任何输出。需启用 FPU (src/bsp/start.S)。

```
lab2 > src > bsp > ASM start.S
1      .global    OsEnterMain
2      .extern    __os_sys_sp_end
3
4      .type      start, function
5      .section   .text.bspinit, "ax"
6      .balign    4
7
8      .global    OsElxState
9      .type      OsElxState, @function
10     OsElxState:
11         MRS     x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
12         MOV     x2, #0x4 // CurrentEL EL1: bits [3:2] = 0b01
13         CMP     w6, w2
14
15         BEQ     Start // 若 CurrentEL 为 EL1 级别, 跳转到 Start 处执行, 否则死循环。
16
17     OsEl2Entry:
18         B       OsEl2Entry
19
20     Start:
21         LDR     x1, __os_sys_sp_end // ld文件中定义, 堆栈设置
22         BIC     sp, x1, #0xf
23
24     //参考: https://developer.arm.com/documentation/den0024/a/Memory-Ordering/Barriers/ISB-in-more-detail
25     Enable_FPU:
26         MRS     X1, CPACR_EL1
27         ORR     X1, X1, #(0x3 << 20)
28         MSR     CPACR_EL1, X1
29         ISB
30
31         B       OsEnterMain
32     OsEnterReset:
33         B       OsEnterReset
34
```

L2: LDR x1, __os_sys_sp_end: 将地址 __os_sys_sp_end 处的值加载到通用寄存器 x1 中。LDR 是 Load Register 的缩写, =表示加载一个立即数或者地址的偏移量。

L3: BIC sp, x1, #0xf: 这是一条位清除 (Bit Clear) 指令, 它用来清除栈指针 sp 的特定位。x1 寄存器中存储了 __os_sys_sp_end 的值, 而 #0xf 是一个掩码, 用于指定需要清除的位。通过这条指令, 将 __os_sys_sp_end 中的值的低 4 位清零, 以确保 sp 寄存器的值符合栈对齐的要求。

L7: MRS X1, CPACR_EL1: 这是一个特权级访问系统寄存器 (MRS) 的指令, 用于将 CPACR_EL1 寄存器 (EL1 中的协处理器访问控制寄存器) 的值加载到通用寄存器 X1 中。

L8: ORR X1, X1, #(0x3 << 20): 这是一个按位或 (OR) 指令, 将寄存器 X1 和一个立即数 0x3 << 20 进行按位或操作, 然后将结果存回寄存器 X1 中。0x3 << 20 表示将 0x3 左移 20 位, 用于设置 CPACR_EL1 寄存器中的相关位, 以启用浮点数处理单元 (FPU)。

L9: MSR CPACR_EL1, X1: 这是一个特权级访问系统寄存器 (MSR) 的指令, 用于将寄存器 X1 中的值写入 CPACR_EL1 寄存器中, 以修改协处理器访问控制寄存器的相关位, 从而启用 FPU。

L10: ISB: 这是一个指令同步栅栏 (Instruction Synchronization Barrier), 用于确保在修改 CPACR_EL1 寄存器后, 处理器在继续执行下一条指令之前刷新指令流水线。

L12: B OsEnterMain: 这是一个无条件分支指令, 将程序的执行流程转移到标签为 OsEnterMain 的位置, 继续执行后续的指令。通常, 这表示程序将进入主要的程序入口点, 开始执行主程序逻辑。

6. Hello, miniEuler

再次构建项目并执行, 发现已可正常输出。至此, 我们获得了一个基本的输出和调试手段, 如我们可以在系统崩溃时调用 PRT_Printf 函数进行输出。我们可以利用 PRT_Printf 函数来打印一个文本 banner 让我们写的 OS 显得专业一点。manytools.org 可以创建 ascii banner, 选择你喜欢的样式和文字(下面选择你喜欢的样式和文字), 然后在 main.c 的 main 函数中调用 PRT_Printf 输出。

```
lab2 > src > C main.c > ...
1  #include "prt_typedef.h"
2
3  extern U32 PRT_Printf(const char *format, ...);
4  extern void PRT_UartInit(void);
5
6
7  S32 main(void)
8  {
9      PRT_UartInit();
10
11      PRT_Printf("
12      PRT_Printf("
13      PRT_Printf("
14      PRT_Printf("
15      PRT_Printf("
16      PRT_Printf("
17
18
19      PRT_Printf("Test PRT_Printf int format %d \n\n", 10);
20  }
```

7. 构建项目并执行

```
(base) xiaoye@长乐:~/OSLab/Lab2$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s

Test PRT_Printf int format 10
```

2.3

作业

(1) 作业 1

不启用 fifo，通过检测 UARTFR 寄存器的 TXFE 位来发送数据。

```
#define DW_UART_FR 0x18 // UARTFR(Flag Register) 寄存器
#define DW_FIFO_ENABLE 0x10 // 启用发送和接收 FIFO

U32 PRT_UartInit(void)
{
    U32 result = 0;
    U32 reg_base = UART_0_REG_BASE;
    // LCR 寄存器:
https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions/line-control-register--uartlcr-h?lang=en
    result = UART_REG_READ((unsigned Long)(reg_base + DW_UART_LCR_HR));
    UART_REG_WRITE(result | DW_FIFO_ENABLE, (unsigned Long)(reg_base + DW_UART_LCR_HR)); // 启用 FIFO

    return OS_OK;
}

// 读 reg_base + offset 寄存器的值。uartno 参数未使用
S32 uart_reg_read(S32 uartno, U32 offset, U32 *val)
{
    S32 ret;
    U32 reg_base = UART_0_REG_BASE;
    *val = UART_REG_READ((unsigned Long)(reg_base + offset));
    return OS_OK;
}

// 通过检查 FR 寄存器的标志位确定发送缓冲是否满，满时返回 1。
S32 uart_is_txfifo_full(S32 uartno)
{
    S32 ret;
    U32 usr = 0;
    ret = uart_reg_read(uartno, DW_UART_FR, &usr);
    if (ret) {
        return OS_OK;
    } return (usr & DW_XFIFO_NOT_FULL);
}

// 往 reg_base + offset 寄存器中写入值 val。
void uart_reg_write(S32 uartno, U32 offset, U32 val)
{
    S32 ret;
```

```
U32 reg_base = UART_0_REG_BASE;

UART_REG_WRITE(val, (unsigned long)(reg_base + offset));

return;
}

// 通过轮询的方式发送字符到串口

void uart_poll_send(unsigned char ch)
{

    S32 timeout = 0;

    S32 max_timeout = UART_BUSY_TIMEOUT;

    // 轮询发送缓冲区是否满

    int result = uart_is_txfifo_full(0);

    while (result) {

        timeout++;

        if (timeout >= max_timeout) {

            return;

        }

        result = uart_is_txfifo_full(0);

    } // 如果缓冲区没满, 通过往数据寄存器写入数据发送字符到串口

    uart_reg_write(0, DW_UART_THR, (U32)(U8)ch);

    return;

}

// 轮询的方式发送字符到串口, 且转义换行符

void TryPutc(unsigned char ch)
{

    uart_poll_send(ch);

    if (ch == '\n') {

        uart_poll_send('\r');

    }

}
```

Table 3-4 UARTFR Register

Bits	Name	Function
15:9	-	Reserved, do not modify, read as zero.
8	RI	Ring indicator. This bit is the complement of the UART ring indicator, nUARTRI , modem status input. That is, the bit is 1 when nUARTRI is LOW.
7	TXFE	Transmit FIFO empty. The meaning of this bit depends on the state of the FEN bit in the <i>Line Control Register, UARTLCR_H</i> on page 3-12. If the FIFO is disabled, this bit is set when the transmit holding register is empty. If the FIFO is enabled, the TXFE bit is set when the transmit FIFO is empty. This bit does not indicate if there is data in the transmit shift register.
6	RXFF	Receive FIFO full. The meaning of this bit depends on the state of the FEN bit in the <i>UARTLCR_H Register</i> . If the FIFO is disabled, this bit is set when the receive holding register is full. If the FIFO is enabled, the RXFF bit is set when the receive FIFO is full.
5	TXFF	Transmit FIFO full. The meaning of this bit depends on the state of the FEN bit in the <i>UARTLCR_H Register</i> . If the FIFO is disabled, this bit is set when the transmit holding register is full. If the FIFO is enabled, the TXFF bit is set when the transmit FIFO is full.

(1) 首先, PRT_UartInit 函数中, UART_REG_WRITE () 函数中各个参数的作用:

result|DW_FIFO_ENABLE: result 是先前通过 UART_REG_READ 从 DW_UART_LCR_HR 寄存器读取的当前值; DW_FIFO_ENABLE 是一个宏定义, 表示需要设置的 FIFO 使能位; 按位或操作 (|): 将

result 的原始值与 DW_FIFO_ENABLE 进行按位或运算。将计算后的值 (result | DW_FIFO_ENABLE) 写入目标寄存器地址 (reg_base + DW_UART_LCR_HR)。

(2) 其次, 通过 S32 uart_is_txfifo_full(S32 uartno), 检查发送 FIFO 是否满。ret = uart_reg_read(uartno, DW_UART_FR, &usr); 读取 Flag Register (FR) 到 usr 变量。读取失败返回 OS_OK。return (usr & DW_XFIFO_NOT_FULL); 读取成功, 返回 FIFO 未满标志位状态。

其中, FR 寄存器: 通常包含 UART 的状态标志, 如: 发送 FIFO 是否满 (TX FIFO Full)、接收 FIFO 是否空 (RX FIFO Empty)、校验错误标志等。通过位掩码操作, 检查发送 FIFO 未满标志位的状态。

(3) uart_reg_write(S32 uartno, U32 offset, U32 val)。写入寄存器值, 向 UART 寄存器写入数据。同样固定操作 UART0, 参数 uartno 未使用。

(4) uart_poll_send(unsigned char ch), 轮询发送字符。通过轮询确保 FIFO 未满后发送字符, 防止溢出。超时机制避免死循环。

(5) TryPutc(unsigned char ch), 发送字符并转义换行符。发送字符时, 若字符是 \n, 则额外发送 \r, 实现跨平台换行兼容 (如 Windows 的 \r\n)。

不启用 FIFO 时, 通过检测 UARTFR 寄存器的 TXFE 位来发送数据。

(1) 提取 UARTFR 寄存器的 TXFE 位,
return ((usr & DW_XFIFO_NOT_FULL) == 0); DW_XFIFO_NOT_FULL 是一个掩码 (Mask), 用于提取状态寄存器 FR 中的 TXFE 位 (Transmit FIFO Empty)。

TXFE 表示发送 FIFO 是否为空。查阅用户手册发现, 当 TXFE 为 1 时, 表示发送 FIFO 有空间 (未滿); TXFE 为 0 时, 表示发送 FIFO 已滿。

通过与掩码 DW_XFIFO_NOT_FULL 按位与 (usr & DW_XFIFO_NOT_FULL), 可以检查 TXFE 位的值, 由此判断判断 FIFO 是否满。

(2) 在不启用 FIFO 的情况下,

1. 硬件方面: 当 FIFO 被禁用时, UART 的发送 FIFO 退化为单字节缓冲区 (即 THR 寄存器本身)。每写入一个字节到 THR 寄存器, 硬件会立即启动发送流程:

数据从 THR 移动到发送移位寄存器 (Shift Register)。

移位寄存器逐位发送数据 (如 8 位数据 + 停止位)。

发送完成后, THR 才允许接收新数据。

2. 代码逻辑: 在发送新字节前, 通过 uart_is_txfifo_full 检查 THR 是否已满 (即前一个字节是否仍在发送中)。如果 THR 满 (发送未完成), 则循环等待 (while 循环)。因此, 每次只能发送 1 字节, 必须等待当前字节发送完成 (THR 变空) 才能发送下一个。

具体代码如下:

/ 通过检查 FR 寄存器的标志位确定发送缓冲区是否满, 满时返回 1.

```
S32 uart_is_txfifo_full(S32 uartno)
{
    S32 ret;
    U32 usr = 0;

    ret = uart_reg_read(uartno, DW_UART_FR, &usr);
    if (ret) {
        return OS_OK;
    }
    // 当 TXFE 位为 0 时表示发送缓冲区满, 返回 1; 否则返回 0
    return ((usr & DW_XFIFO_NOT_FULL) == 0);
}
// 通过轮询的方式发送字符到串口 (禁用 FIFO 适配版)
void uart_poll_send(unsigned char ch)
```

```

{
    S32 timeout = 0;

    S32 max_timeout = UART_BUSY_TIMEOUT;

    // 轮询检查发送缓冲状态: 当缓冲区满时等待
    while (uart_is_txfifo_full(0)) {
        timeout++;

        if (timeout >= max_timeout) {
            return; // 超时退出
        }
    }

    // 发送字符到数据寄存器
    uart_reg_write(0, DW_UART_THR, (U32)(U8)ch);
}

```

总结:

(1) 对比 FIFO 启用时的行为:

若 FIFO 启用, 硬件会缓存多个字节 (如 16 字节 FIFO), 允许连续写入多个字节, 由硬件自动管理发送节奏;

禁用 FIFO 时, 发送完全依赖 THR 寄存器的单字节容量, 必须逐字节发送。

(2) 对比 FIFO 启用时的发送流程

CPU 写入 ch0 到 THR → 硬件开始发送 ch0 → THR 满 → CPU 必须等待发送完成 → 发送完成后 THR 变空 → CPU 写入 ch1 → 重复过程

启用 FIFO 时的发送流程

CPU 连续写入 ch0, ch1, ch2 → FIFO 缓存 3 字节 → 硬件自动逐字节发送 → CPU 可继续写入更多字节

(2) 作业 2 (可选)

采用 UniProton 提供的 libboundscheck 库实现 vsnprintf_s 函数。

(1) 首先克隆 libboundscheck 库 <https://gitee.com/openeuler/libboundscheck>;

(2) 其次, 将下载的 libboundscheck 目录下 src 目录中的函数以及头文件放入我们操作系统内核的 bsp 目录, 将 libboundscheck 目录下的文件 include 目录中的头文件加入我们操作系统核

中的 include 目录中。

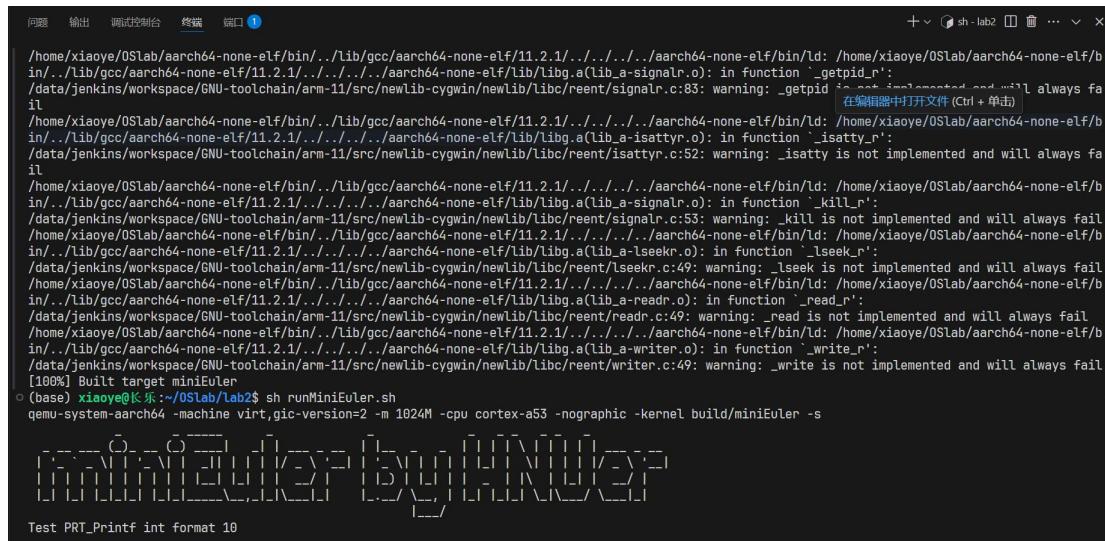
名称	修改日期	类型	大小
CMakeLists.txt	2025/4/18 21:08	文本文档	1 KB
fscanf_s.c	2025/4/18 20:53	C Source File	3 KB
fwscanf_s.c	2025/4/18 20:53	C Source File	3 KB
gets_s.c	2025/4/18 20:53	C Source File	3 KB
input.inl	2025/4/18 20:53	INL 文件	77 KB
memcpy_s.c	2025/4/18 20:53	C Source File	22 KB
memmove_s.c	2025/4/18 20:53	C Source File	5 KB
memset_s.c	2025/4/18 20:53	C Source File	19 KB
output.inl	2025/4/18 20:53	INL 文件	62 KB
print.c	2025/4/16 1:50	C Source File	4 KB
prt_reset_vector.S	2025/4/14 15:28	S 文件	1 KB
scanf_s.c	2025/4/18 20:53	C Source File	2 KB
secinput.h	2025/4/18 20:53	C Header File	7 KB
secureutil.c	2025/4/18 20:53	C Source File	3 KB
secureutil.h	2025/4/18 20:53	C Header File	21 KB
secureinput_a.c	2025/4/18 20:53	C Source File	2 KB
secureinput_w.c	2025/4/18 20:53	C Source File	3 KB
secureprintoutput.h	2025/4/18 20:53	C Header File	6 KB
secureprintoutput_a.c	2025/4/18 20:53	C Source File	4 KB
secureprintoutput_w.c	2025/4/18 20:53	C Source File	2 KB
snprintf_s.c	2025/4/18 20:53	C Source File	5 KB

名称	修改日期	类型	大小
prt_typedef.h	2025/4/13 21:25	C Header File	3 KB
securec.h	2025/4/18 20:53	C Header File	28 KB
securectype.h	2025/4/18 20:53	C Header File	19 KB

(3) 更改/src/bsp/CMakeLists.txt, 将新加入 bsp 目录的文件纳入 cmake 构建系统。

```
lab2 > src > bsp > M CMakeLists.txt
1  set(SRCS start.S prt_reset_vector.S print.c
2  fscanf_s.c  memset_s.c  secureinput_a.c  sprintf_s.c
3  strtok_s.c  vsnprintf_s.c  wscat_s.c  wmemmove_s.c
4  fwscanf_s.c  output.inl  secureinput_w.c  sscanf_s.c
5  swprintf_s.c  vsprintf_s.c  wcsncpy_s.c  wscanf_s.c
6  gets_s.c  scanf_s.c  secureprintoutput.h  strcat_s.c
7  swscanf_s.c  vsscanf_s.c  wcsncat_s.c  input.inl
8  secinput.h  secureprintoutput_a.c  strcpy_s.c  vfscanf_s.c
9  vswprintf_s.c  wcsncpy_s.c  memcpy_s.c  secureutil.c
10 secureprintoutput_w.c  strncat_s.c  vfwscanf_s.c  vswscanf_s.c
11 wcstok_s.c  memmove_s.c  secureutil.h  snprintf_s.c
12 strncpy_s.c  vscanf_s.c  vwscanf_s.c  wmemcpy_s.c)
13 add_library(bsp OBJECT ${SRCS}) # OBJECT类型只编译生成.o目标文件，但不实际链接成库
14
```

(4) 构建工程，并运行程序。



```
/home/xiaoye/OSlab/aarch64-none-elf/bin/..../aarch64-none-elf/bin/ld: /home/xiaoye/OSlab/aarch64-none-elf/b
in/..../aarch64-none-elf/lib/libg.a(lib_a-signalr.o): in function `_getpid_r':
/data/jenkins/workspace/GNU-toolchain/arm-11/src/newlib-cygwin/newlib/libc/reent/signalr.c:83: warning: `_getpid
在编辑器中打开文件 (Ctrl + 单击)
/home/xiaoye/OSlab/aarch64-none-elf/bin/..../aarch64-none-elf/bin/ld: /home/xiaoye/OSlab/aarch64-none-elf/b
in/..../aarch64-none-elf/lib/libg.a(lib_a-isatty.o): in function `_isatty_r':
/data/jenkins/workspace/GNU-toolchain/arm-11/src/newlib-cygwin/newlib/libc/reent/isatty.c:52: warning: `_isatty
/home/xiaoye/OSlab/aarch64-none-elf/bin/..../aarch64-none-elf/bin/ld: /home/xiaoye/OSlab/aarch64-none-elf/b
in/..../aarch64-none-elf/lib/libg.a(lib_a-signalr.o): in function `_kill_r':
/data/jenkins/workspace/GNU-toolchain/arm-11/src/newlib-cygwin/newlib/libc/reent/signalr.c:53: warning: `_kill
/home/xiaoye/OSlab/aarch64-none-elf/bin/..../aarch64-none-elf/bin/ld: /home/xiaoye/OSlab/aarch64-none-elf/b
in/..../aarch64-none-elf/lib/libg.a(lib_a-lseekr.o): in function `_lseek_r':
/data/jenkins/workspace/GNU-toolchain/arm-11/src/newlib-cygwin/newlib/libc/reent/lseekr.c:49: warning: `_lseek
/home/xiaoye/OSlab/aarch64-none-elf/bin/..../aarch64-none-elf/bin/ld: /home/xiaoye/OSlab/aarch64-none-elf/b
in/..../aarch64-none-elf/lib/libg.a(lib_a-readr.o): in function `_read_r':
/data/jenkins/workspace/GNU-toolchain/arm-11/src/newlib-cygwin/newlib/libc/reent/readr.c:49: warning: `_read
/home/xiaoye/OSlab/aarch64-none-elf/bin/..../aarch64-none-elf/bin/ld: /home/xiaoye/OSlab/aarch64-none-elf/b
in/..../aarch64-none-elf/lib/libg.a(lib_a-writer.o): in function `_write_r':
/data/jenkins/workspace/GNU-toolchain/arm-11/src/newlib-cygwin/newlib/libc/reent/writer.c:49: warning: `_write
[100%] Built target miniEuler
(base) xiaoye@长东:~/OSlab/Lab2$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s

miniEuler by WINKER

Test PRT_Printf int format 10
```

3 心得体会

- (1) 对操作系统与硬件的接口方法有了更为深刻的理解；
- (2) 了解了通用虚拟机模板 virt 机器；
- (3) 了解并实现了串口输出；
- (4) 理解了底层不同寄存器的功能与函数实现。