

Lab2

一、实验目的

- 1.理解操作系统与硬件的接口方法
- 2.实现一个可打印字符的宏（非系统调用），用于后续的调试和开发

二、实验过程

1.在Linux中安装设备树格式转换工具

设备树 (*Device Tree*, 简称DT) 是一种描述硬件配置的数据结构, 主要用于嵌入式系统 (如Linux内核) 中, 以标准化的方式传递硬件信息, 使操作系统无需依赖硬编码即可适配多种硬件平台。

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

2.通过QEMU导出设备树并转成可读格式

```
qemu-system-aarch64 -machine  
virt,dumpdtb=virt.dtb -cpu cortex-a53 -  
nographic  
dtc -I dtb -O dts -o virt.dts virt.dtb
```

- **dtc**: 设备树编译器 (Device Tree Compiler) 工具。
- **-I dtb**: 指定输入格式为二进制设备树 (**.dtb**)。
- **-O dts**: 指定输出格式为文本设备树 (**.dts**)。
- **-o virt.dts**: 将输出保存到文件 **virt.dts**。
- **virt.dtb**: 输入的二进制设备树文件。

virt.dtb转换后生成的virt.dts中可找到如下内容:

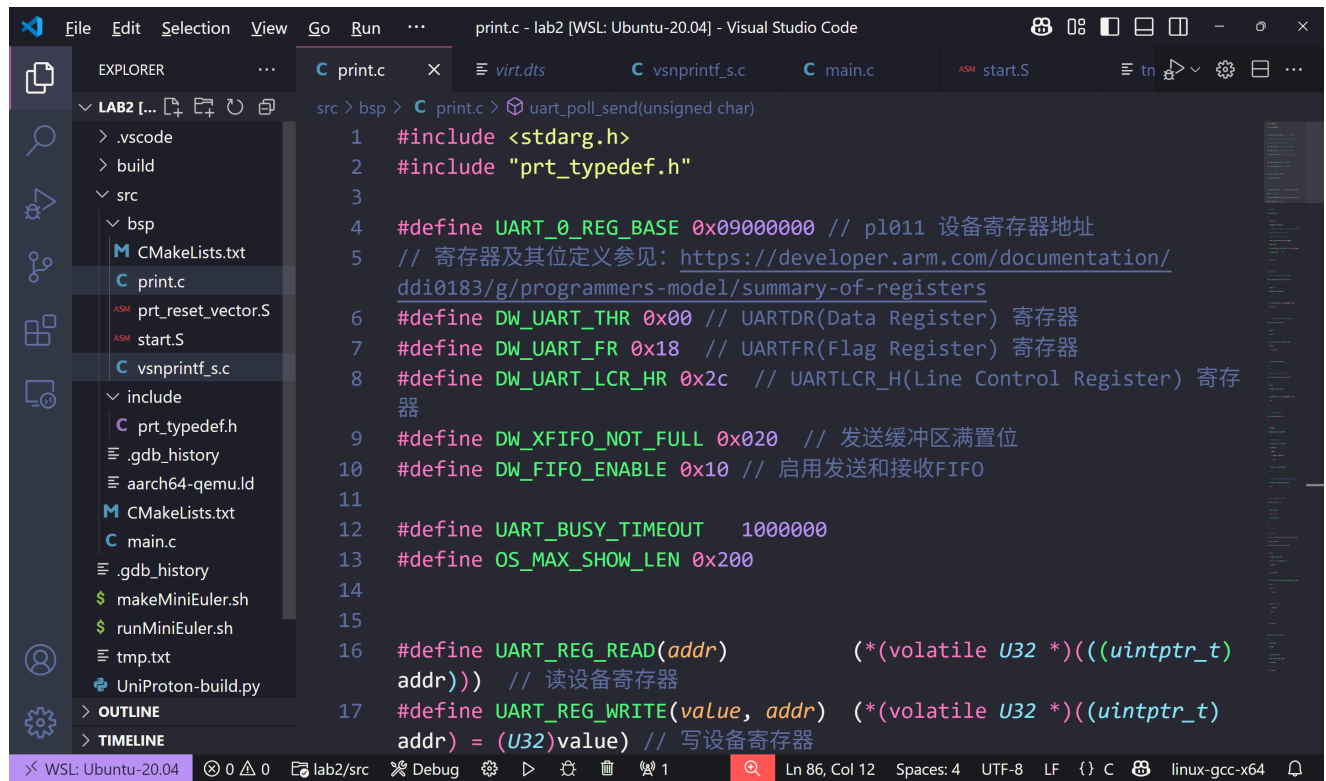
```
pl011@9000000 {
    clock-names = "uartclk\0apb_pclk";
    clocks = <0x8000 0x8000>;
    interrupts = <0x00 0x01 0x04>;
    reg = <0x00 0x9000000 0x00
0x1000>;
    compatible =
"arm,pl011\0arm,primecell";
};
chosen {
    stdout-path = "/pl011@9000000";
};
```

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。

- **UART** (Universal Asynchronous Receiver/Transmitter, 通用异步收发传输器)
- **串口** (Serial Port) 是一种用于逐位 (**bit-by-bit**) 传输数据的通信接口，与“并口” (并行传输) 相对。其核心特点是**通过单条数据线 (或少量线路) 依次发送数据**

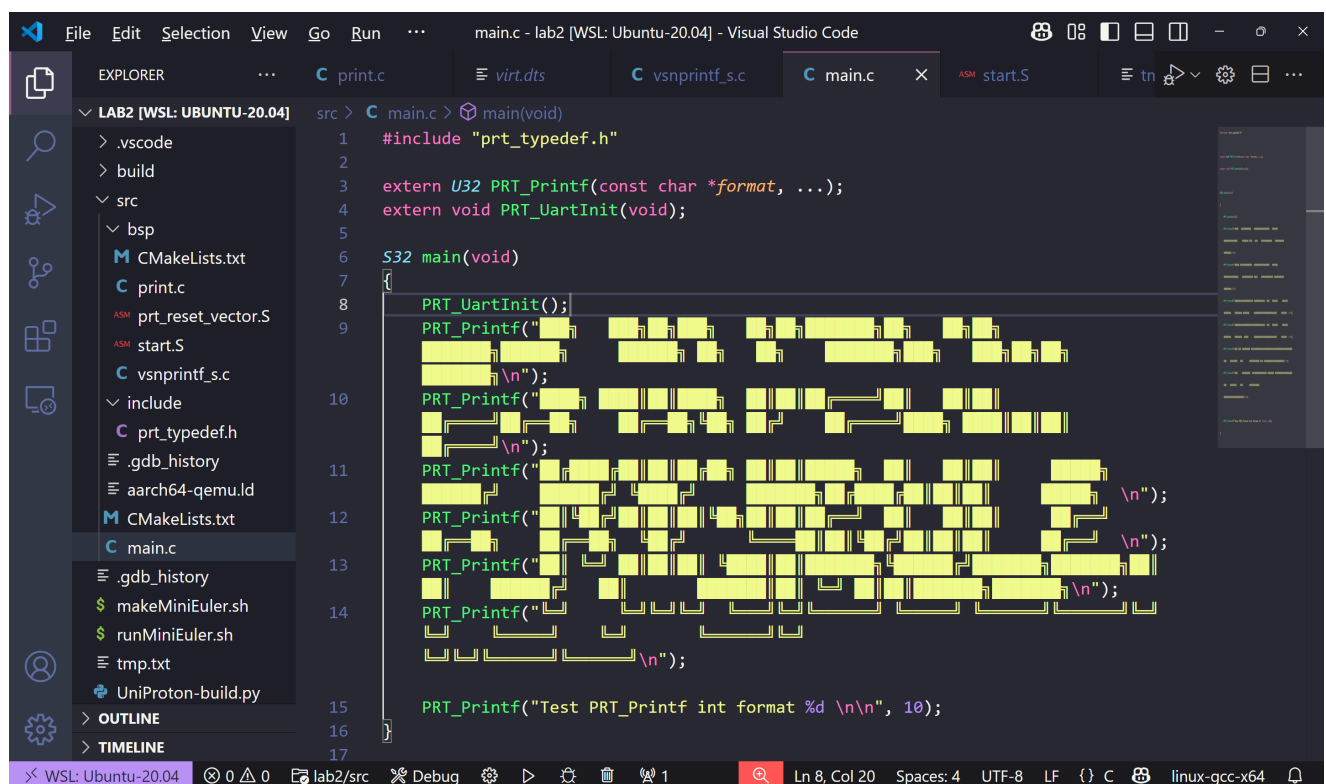
3.实现 PRT_Printf 函数

PRT_Printf 可能表示 通过指定端口输出数据，其中 **PRT** 可能表示端口 **port** 的缩写



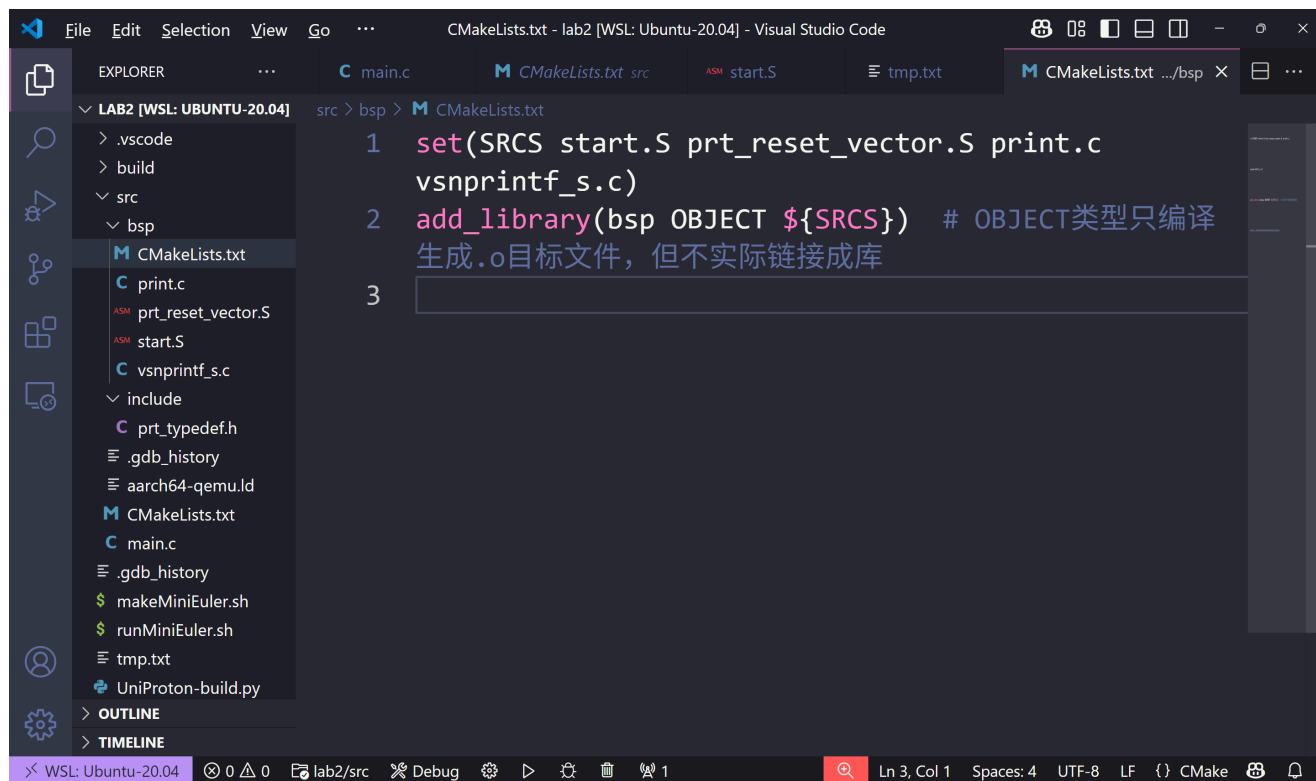
```
src > bsp > C print.c > uart_poll_send(unsigned char)
1  #include <stdarg.h>
2  #include "prt_typedef.h"
3
4  #define UART_0_REG_BASE 0x09000000 // p1011 设备寄存器地址
5  // 寄存器及其位定义参见: https://developer.arm.com/documentation/ddi0183/g/programmers-model/summary-of-registers
6  #define DW_UART_THR 0x00 // 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
10 #define DW_XFIFO_NOT_FULL 0x020 // 发送缓冲区满置位
11 #define DW_FIFO_ENABLE 0x10 // 启用发送和接收FIFO
12
13 #define UART_BUSY_TIMEOUT 1000000
14 #define OS_MAX_SHOW_LEN 0x200
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
```

4.调用PRT_printf函数



```
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      PRT_Printf("UART Poll Send\n");
10     PRT_Printf("UART Poll Send\n");
11     PRT_Printf("UART Poll Send\n");
12     PRT_Printf("UART Poll Send\n");
13     PRT_Printf("UART Poll Send\n");
14     PRT_Printf("UART Poll Send\n");
15     PRT_Printf("Test PRT_Printf int format %d\n\n", 10);
16 }
17
```

5.将新增文件纳入构建系统

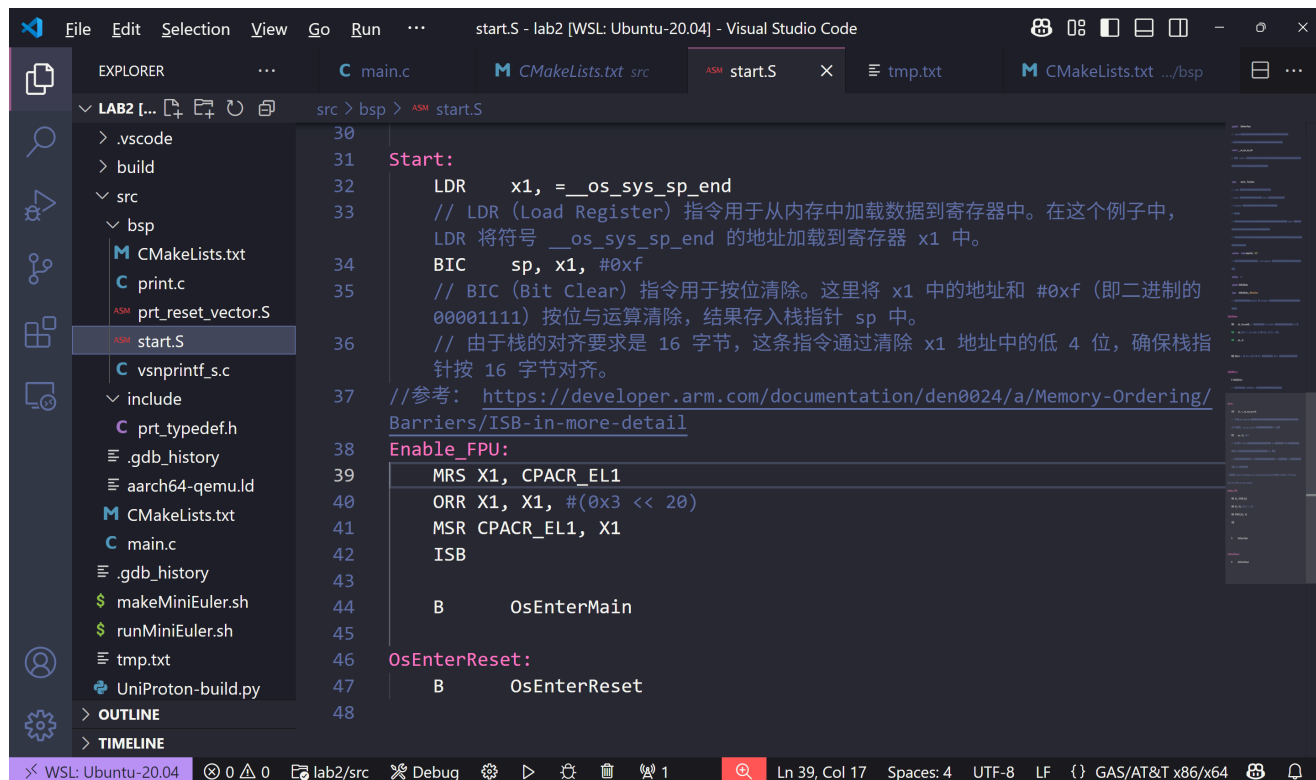


The screenshot shows the Visual Studio Code editor with the CMakeLists.txt file open. The file is located in the 'src' directory of the 'lab2' project. The code defines a project named 'lab2' and adds several source files to the 'bsp' target. A comment explains that the 'add_library' command is used to generate object files but not a library.

```
1 set(SRCS start.S prt_reset_vector.S print.c
2   vsnprintf_s.c)
3 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
```

The Explorer sidebar on the left shows the project structure, including the 'src' directory and the 'bsp' target. The status bar at the bottom indicates the file is in the 'src' directory and is using the 'CMake' build system.

6.在start.S中启用FPU



The screenshot shows the Visual Studio Code editor with the start.S file open. The file is located in the 'src' directory of the 'lab2' project. The code defines the 'Start' function, which initializes the stack pointer and enables the FPU. A comment explains the purpose of the 'BIC' instruction.

```
30
31 Start:
32   LDR    x1, =__os_sys_sp_end
33   // LDR (Load Register) 指令用于从内存中加载数据到寄存器中。在这个例子中，
34   // LDR 将符号 __os_sys_sp_end 的地址加载到寄存器 x1 中。
35   BIC    sp, x1, #0xf
36   // BIC (Bit Clear) 指令用于按位清除。这里将 x1 中的地址和 #0xf (即二进制的
37   // 00001111) 按位与运算清除，结果存入栈指针 sp 中。
38   // 由于栈的对齐要求是 16 字节，这条指令通过清除 x1 地址中的低 4 位，确保栈指
39   // 针按 16 字节对齐。
40   //参考: https://developer.arm.com/documentation/den0024/a/Memory-Ordering/Barriers/ISB-in-more-detail
41   Enable_FPU:
42     MRS    X1, CPACR_EL1
43     ORR    X1, X1, #(0x3 << 20)
44     MSR    CPACR_EL1, X1
45     ISB
46   B       OsEnterMain
47
48 OsEnterReset:
49   B       OsEnterReset
```

The Explorer sidebar on the left shows the project structure, including the 'src' directory and the 'start.S' file. The status bar at the bottom indicates the file is in the 'src' directory and is using the 'GAS/AT&T x86/x64' build system.

三、测试及分析

```
> sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
MINIEULER BY SMILE
Test PRT_Printf int format 10
QEMU: Terminated
~/os/lab2 14s base 11:51:26 PM
```

四、作业

作业1：不启用 FIFO，通过检测 UARTFR 寄存器的 TXFE 位来发送数据。

1.禁用FIFO

FIFO（先进先出缓冲区）在UART中的作用是作为数据中转站，发送时暂存CPU写入的待发送数据，接收时缓存外部设备发来的数据，通过批量处理降低CPU中断频率。

```
29 U32 PRT_UartInit(void)
30 {
31     U32 result = 0;
32     U32 reg_base = UART_0_REG_BASE;
33     // LCR寄存器: https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions/line-control-register--uartlcr-h?lang=en
34     result = UART_REG_READ((unsigned long)(reg_base + DW_UART_LCR_HR));
35     UART_REG_WRITE(result | DW_FIFO_ENABLE, (unsigned long)(reg_base + DW_UART_LCR_HR)); // 启用 FIFO
36
37
38     return OS_OK;
39 }
40
```

分析源代码：

- **reg_base**: UART 0控制器的基地址，即它在内存空间的起始位置
- **DW_UART_LCR_HR**: LCR 寄存器的偏移地址（相对于 **reg_base**）。
- **DW_FIFO_ENABLE**: **FIFO**的使能位，原代码通过将 **result**和**DW_FIFO_ENABLE**做位或启用**FIFO**，所以只要去掉这个位或操作就实现了禁用**FIFO**的效果，如下图所示

```

29  U32 PRT_UartInit(void)
30  {
31      U32 result = 0;
32      U32 reg_base = UART_0_REG_BASE;
33      // LCR寄存器: https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions/line-control-register--uartlcr-h?lang=en
34      result = UART_REG_READ((unsigned long)(reg_base + DW_UART_LCR_HR));
35      // UART_REG_WRITE(result | DW_FIFO_ENABLE, (unsigned long)(reg_base + DW_UART_LCR_HR)); // 启用 FIFO
36
37
38      return OS_OK;
39  }

```

2.查阅用户手册

	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> .
7 TXFE	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.

Holding Register:

- 发送保持寄存器（**THR**）是**UART**控制器中用于暂存待发送数据的核心寄存器。**CPU**将需要发送的数据写入**THR**，**UART**控制器随后将数据转移到发送移位寄存器（**TSR**）进行串行化发送。
- 关键特性：
 - 单字节缓冲：在**FIFO**禁用模式下，**THR**只能暂存1字节数据。
 - 写入阻塞：若**THR**未空（前一个字节未转移到**TSR**），新数据写入会被忽略或覆盖（取决于**UART**设计）。

查阅用户手册，可以发现，当**FIFO**禁用时，**TXFE**会在**holding register**为空的时候被设置为1，为了取出**Flag register**中的**TXFE**位，我们设置一个新的掩码**DW_XHR_NOT_FULL**为0x80（从右至左Bits=7，故为第八位，将0x80转成二进制，对应就是第八位为1，其他位为0，取出第八位）。


```
#define DW_XHR_NOT_FULL 0x080 // 发送缓冲区满置位
```

****名称解析****

- ****DW****: DesignWare (Synopsys公司的IP核命名前缀)
- ****XHR****: Transmit Holding Register (发送保持寄存器) 的缩写
- ****NOT_FULL****: 状态指示信号 ("非满")

组合含义: ****"DesignWare UART的发送保持寄存器未满足"状态信号****

3.修改检查函数为查看TXFE位:

```
45 // 通过检查 holding register寄存器确定缓冲是否满，满时返回1.
46 S32 uart_is_hr_full(S32 uartno)
47 {
48     S32 ret;
49     U32 usr = 0;
50
51     ret = uart_reg_read(uartno, DW_UART_FR, &usr);
52     if (ret) {
53         return OS_OK;
54     }
55
56     return !(usr & DW_XHR_NOT_FULL);
57 }
```

4.修改轮询的函数，将检测缓冲区是否已满的函数改成uart_is_hr_full

```
82 // 通过轮询的方式发送字符到串口
83 void uart_poll_send(unsigned char ch)
84 {
85
86     S32 timeout = 0;
87     S32 max_timeout = UART_BUSY_TIMEOUT;
88
89     // 轮询发送缓冲区是否满
90     int result = uart_is_hr_full(0);
91     while (result) {
92         timeout++;
93         if (timeout >= max_timeout) {
94             return;
95         }
96         result = uart_is_hr_full(0);
97     }
98
99     // 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
100    uart_reg_write(0, DW_UART_THR, (U32)(U8)ch);
101    return;
102 }
```

5.重新构建项目并执行，发现顺利输出

```
> build
> cmake --build .
Scanning dependencies of target bsp
[ 16%] Building C object bsp/CMakeFiles/bsp.dir/print.c.obj
[ 66%] Built target bsp
Scanning dependencies of target miniEuler
[ 83%] Building C object CMakeFiles/miniEuler.dir/main.c.obj
[100%] Linking C executable miniEuler
[100%] Built target miniEuler
>
> sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
MINIEULER BY SMILE
Test PRT_Printf int format 10
QEMU: Terminated
~/os/lab2 11s base 12:19:28 AM
```