

INTRODUCTION TO RISC-V

RISC-V入门教程

ISA | 汇编指令 | 系统编程 | 组成原理 | 嵌入式应用

访问外设

主讲 邢建国



中国开放指令生态（RISC-V）联盟 | 浙江中心
China RICS-V Alliance | Zhejiang Center



浙江图灵算力研究院
ZHEJIANG TURING INSTITUTE





访问外设

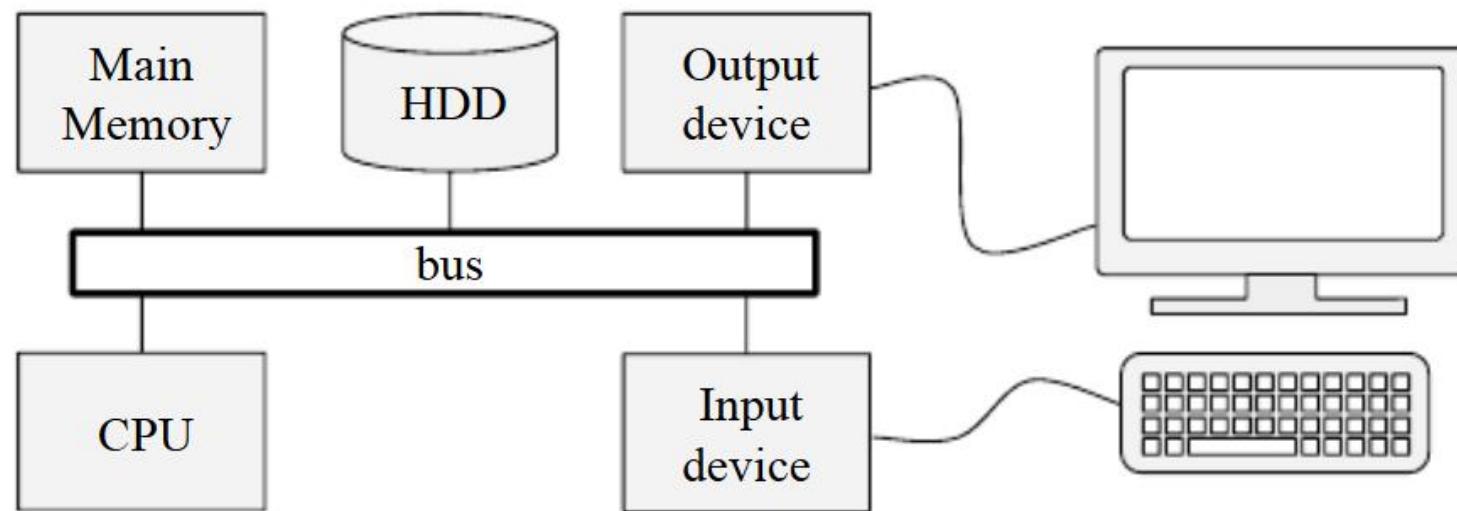


- 读写外设端口
- 示例1：led、按键
- 示例2：键盘与终端
- 示例3：计数器和运行时间
- 示例4：图形显示
- 嵌入式系统常用外设及接口

■ 外设

输入和输出设备通过**总线**与中央处理器**CPU**接口。

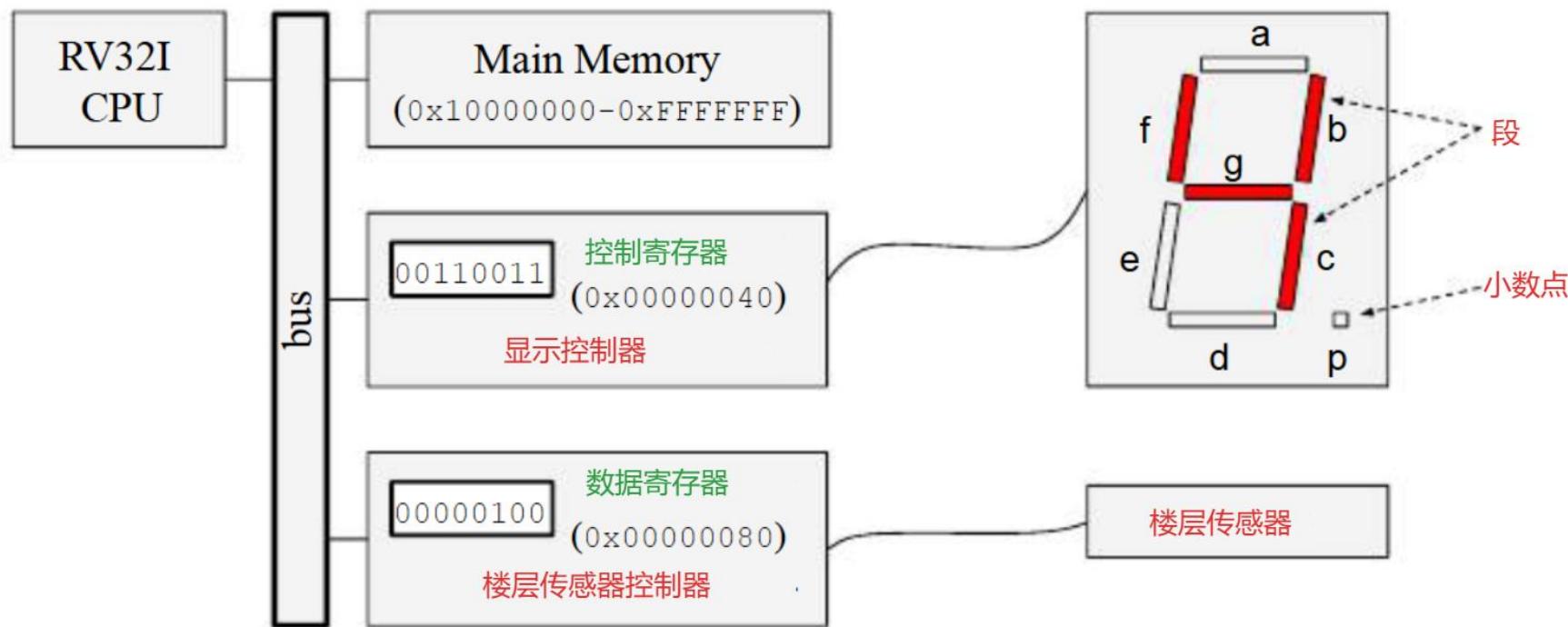
总线是在计算机组件之间传输信息的通信系统。该系统通常由负责传输信息的线路和协调通信的相关电路组成。



■ RISC-V中I/O操作

RISC-V ISA (包括RV32I ISA) 的输入和输出操作是使用内存映射输入/输出方法执行的。

输入操作是通过执行加载指令 (load) 来执行的，而输出操作是通过存储指令 (store) 来执行的。



■ C语言中读写外设端口

```
#define io_base 0x00000040
```

volatile: 告诉编译器不要进行优化

读一个字: int val = *((volatile int *)io_base);

写一个字: *((volatile int *)io_base) = val

读一个字节: char val = *((volatile char *)io_base);

写一个字节: *((volatile char *)io_base) = val

```
#define IOW(addr) *((volatile int *)(addr))
```

```
#define IOB(addr) *((volatile char *)(addr))
```

读设备 (字) : val = IOW(addr)

写设备 (字) : IOW(addr) = val

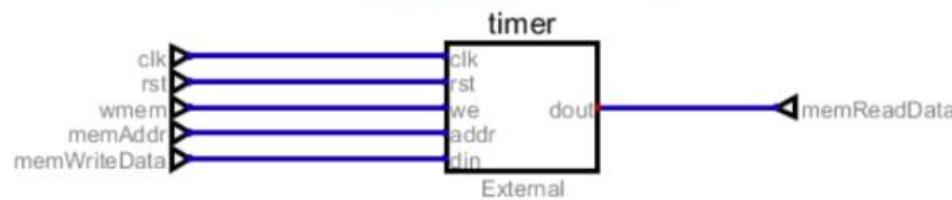
■ 单周期处理器外设

下面几个例子以第三章中单周期处理器RV32I为例。

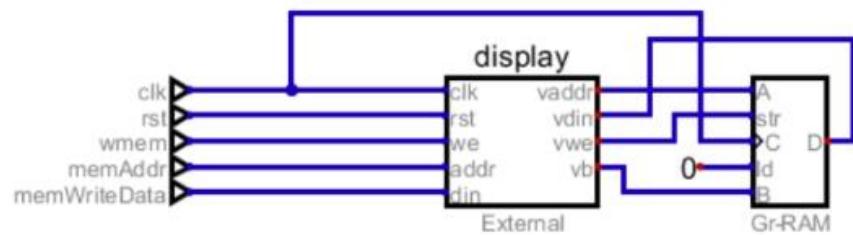
- 内存布局
 - 代码段: 0x0040_0000 , 64k
 - 数据段: 0x1001_0000 , 64k
- 外设:
 - 定时器 : 0xffff_0000
 - Led灯: 0xffff_0010
 - 按键: 0xffff_0020
 - 终端: 0xffff_0030
 - 图形显存: 0xffff_8000 – 0xffff_ffff

单周期处理器外设

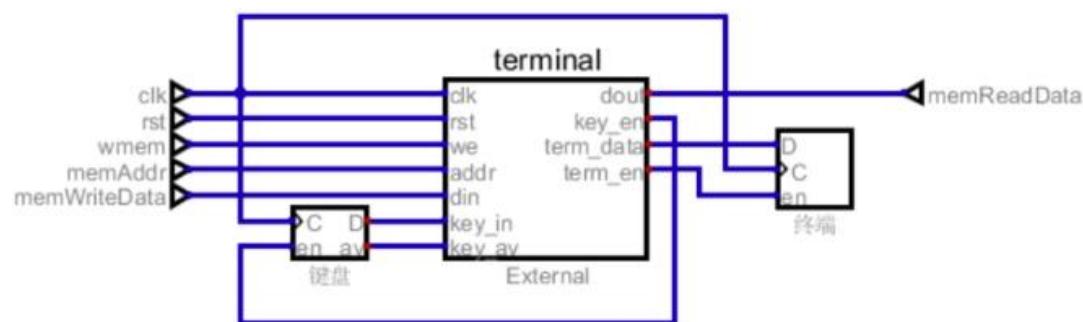
定时器: 0xffff_0000



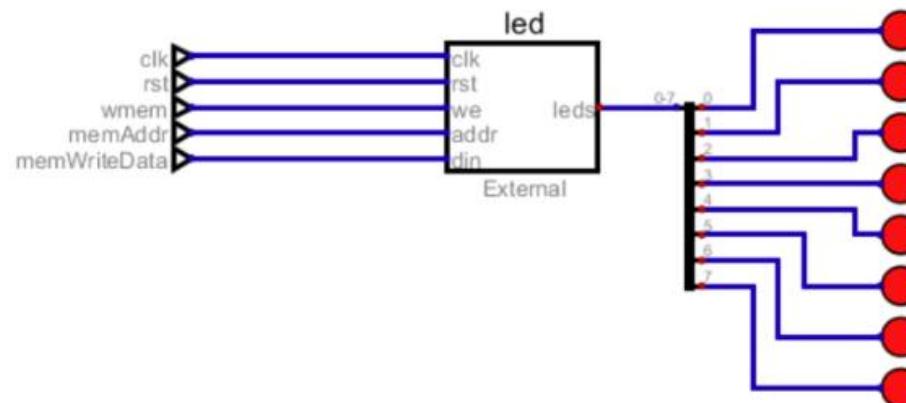
图形显示器: 0xffff_8000 - 0xffff_ffff



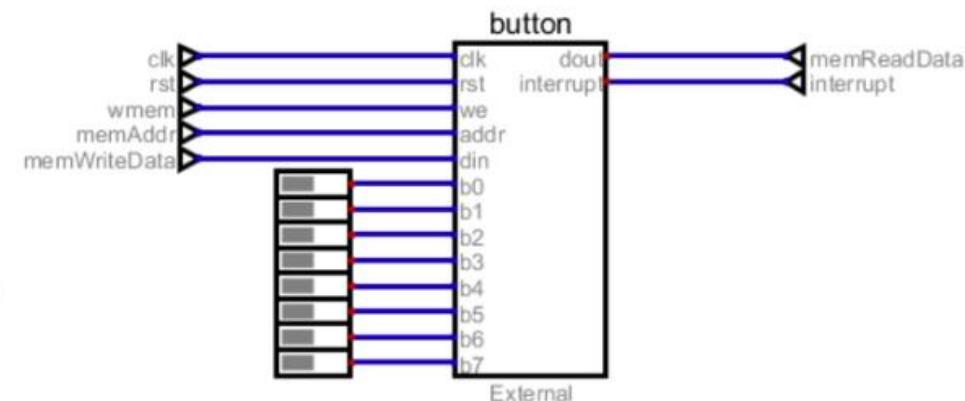
终端: 0xffff_0030



led: 0xffff_0010



按键: 0xffff_0020



■ 示例代码结构 (同4-2节)



■ 链接脚本：my.ld (同4-2节)

```
OUTPUT_ARCH("riscv")
ENTRY(_start)

SECTIONS {
    . = 0x00400000;
    .text : {
        *(.text)
        *(.text.*)
    }

    . = 0x10010000;
    .data : {
        *(.data)
        *(.data.*)
        *(.rodata)
        *(.rodata.*)
    }
}

.bss : {
    _bss_start = .;
    *(.bss)
    *(.bss.*)
    *(COMMON)
    _bss_end = .;
}
```

■ Makefile (図4-2)

```
CROSS = riscv32-unknown-elf-
CC = $(CROSS)gcc
OBJCOPY = $(CROSS)objcopy

CFLAGS = -march=rv32i -mabi=ilp32 -nostartfiles
LDFILE = my.lds

TARGET = x.elf
SOURCE = start.s main.c

.PHONY: all clean

all: code.hex data.hex
```

```
$(TARGET): $(SOURCE)
    $(CC) $(CFLAGS) -T $(LDFILE) -o $@ $(SOURCE)

code.bin: $(TARGET)
    $(OBJCOPY) --dump-section .text=@ <
    data.bin: $(TARGET)
    $(OBJCOPY) --dump-section .data=@ <

code.hex: code.bin
    hexdump -v -e '1/4 "%08x\n'" < > @@
data.hex: data.bin
    hexdump -v -e '1/4 "%08x\n'" < > @@
clean:
    rm -f $(TARGET) code.hex data.hex code.bin data.bin
```

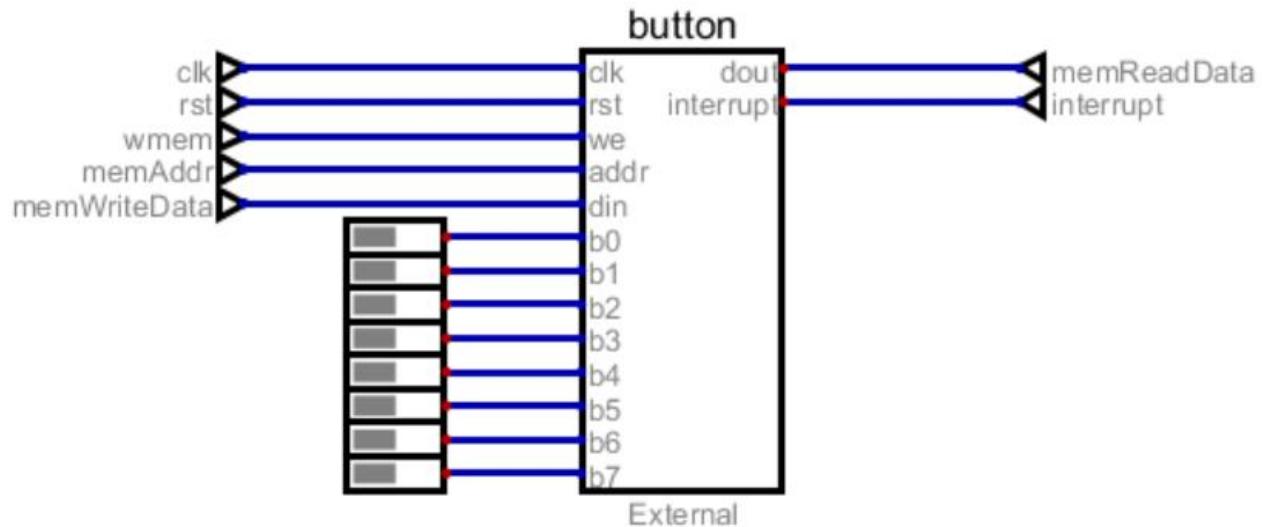
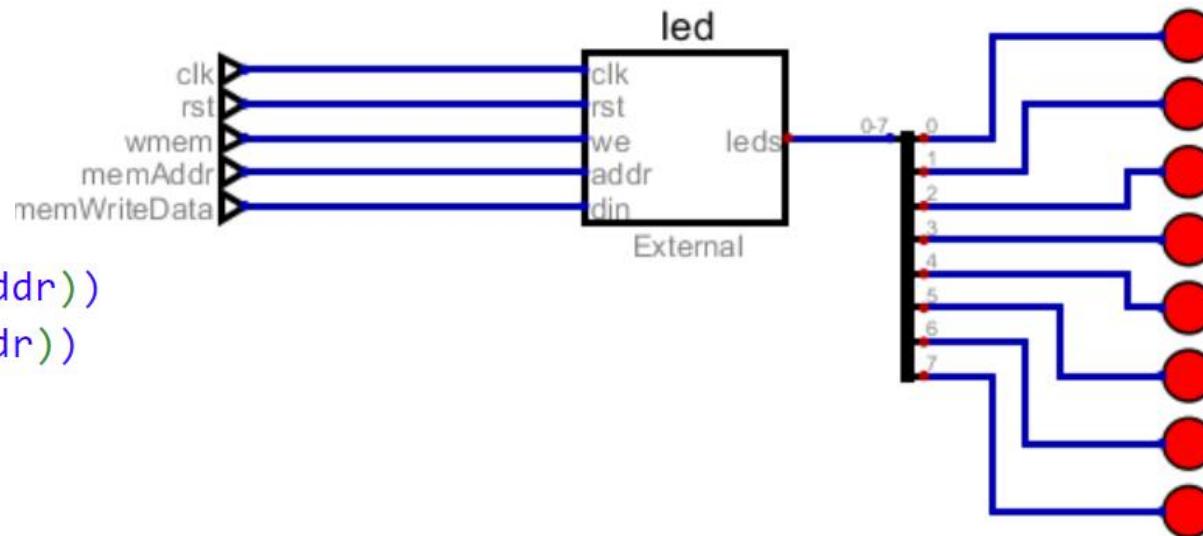
示例1：led与按键

```
#define IOB(addr) *((volatile char *)addr)  
#define IOW(addr) *((volatile int *)addr)
```

```
#define led_base 0xffff0010  
#define button_base 0xffff0020
```

```
char button(){  
    return *IOB(button_base);  
}
```

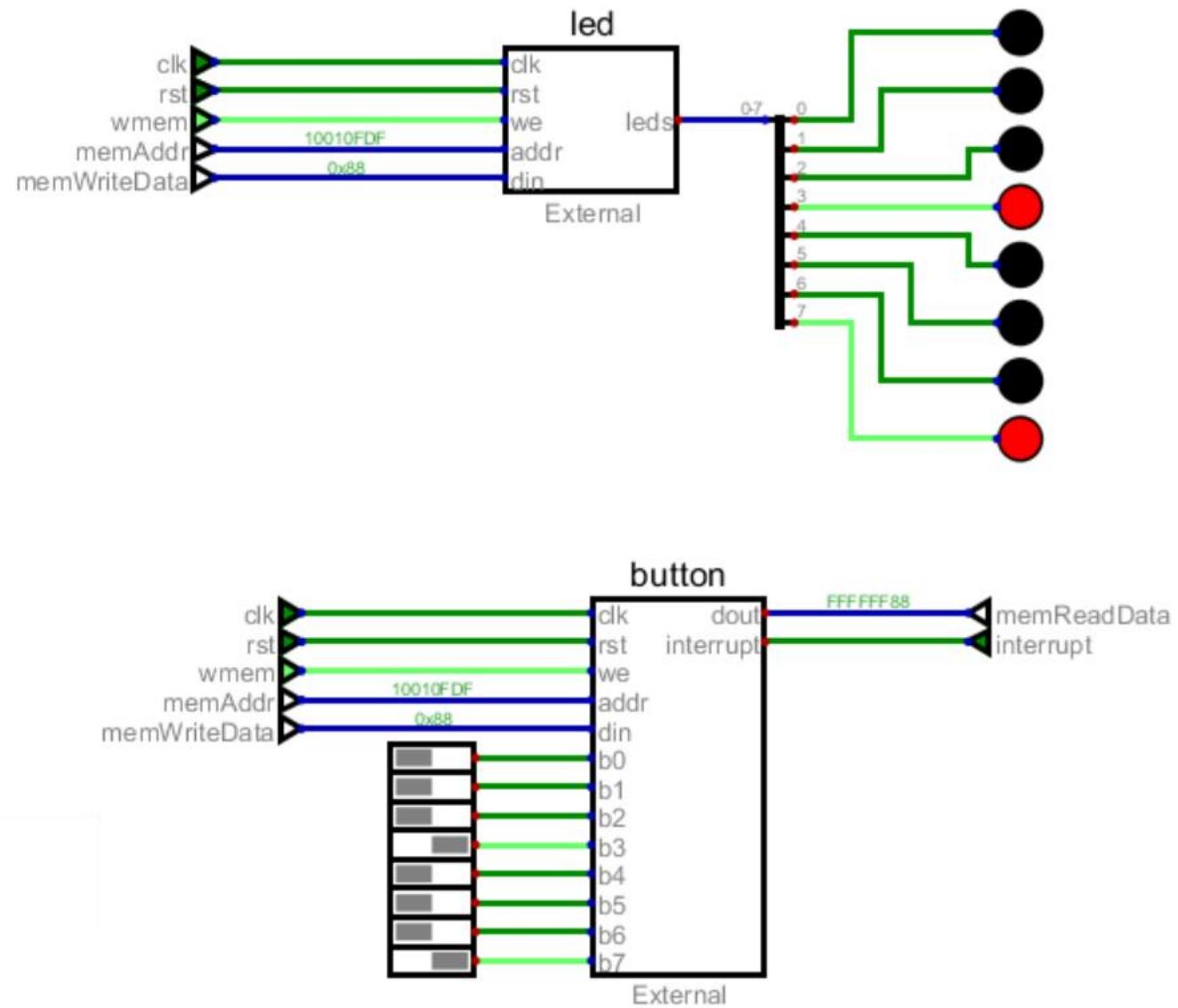
```
void led(char val){  
    *IOB(led_base) = val;  
}
```



■ 示例1：led与按键

读按键，然后显示在对应led灯上

```
int main(){
    for(;;){
        led(button());
    }
}
```

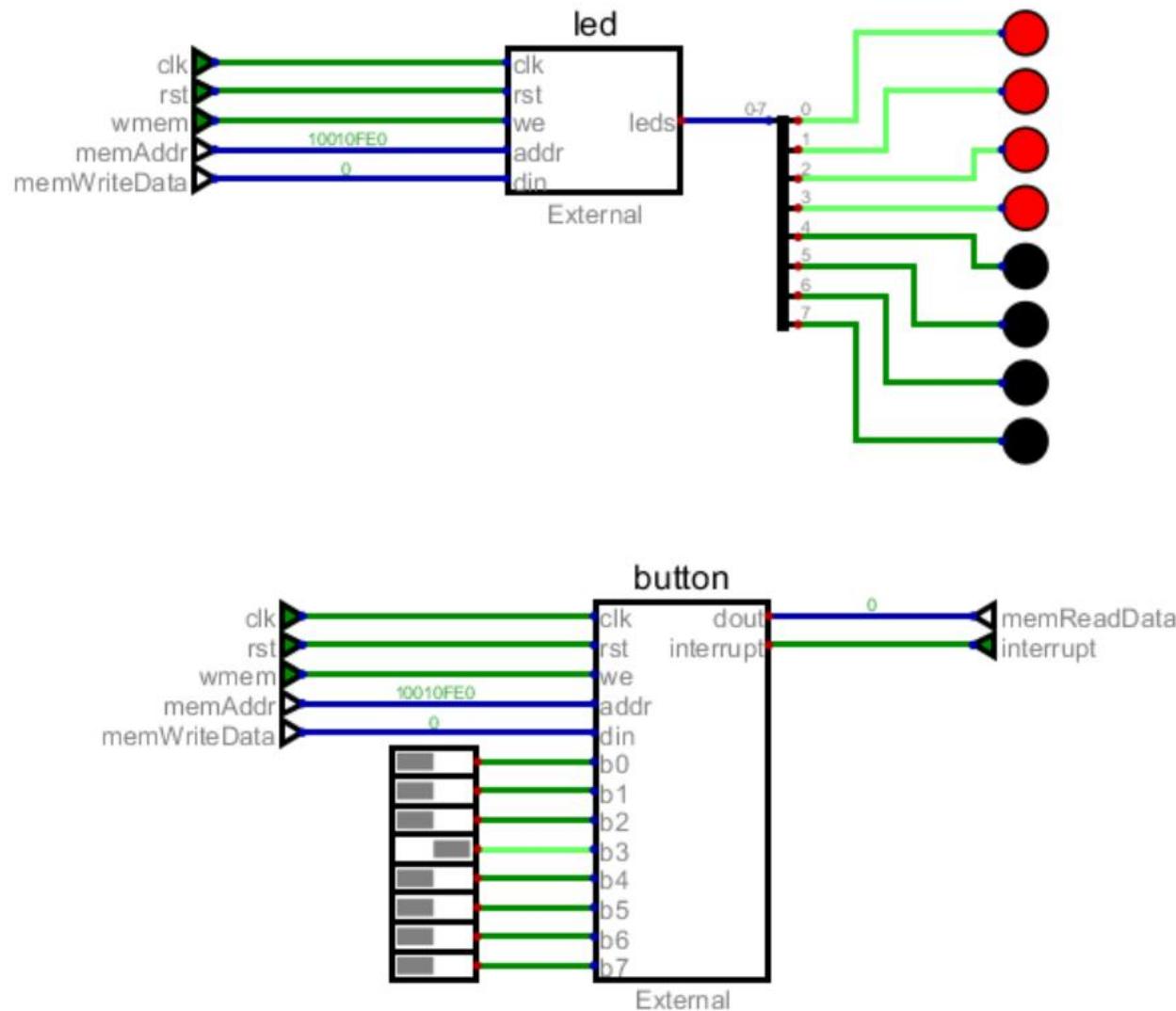


■ 示例1：led与按键

改进：按第一个键，显示第一个led；
按第二个键，显示前两个led；

...

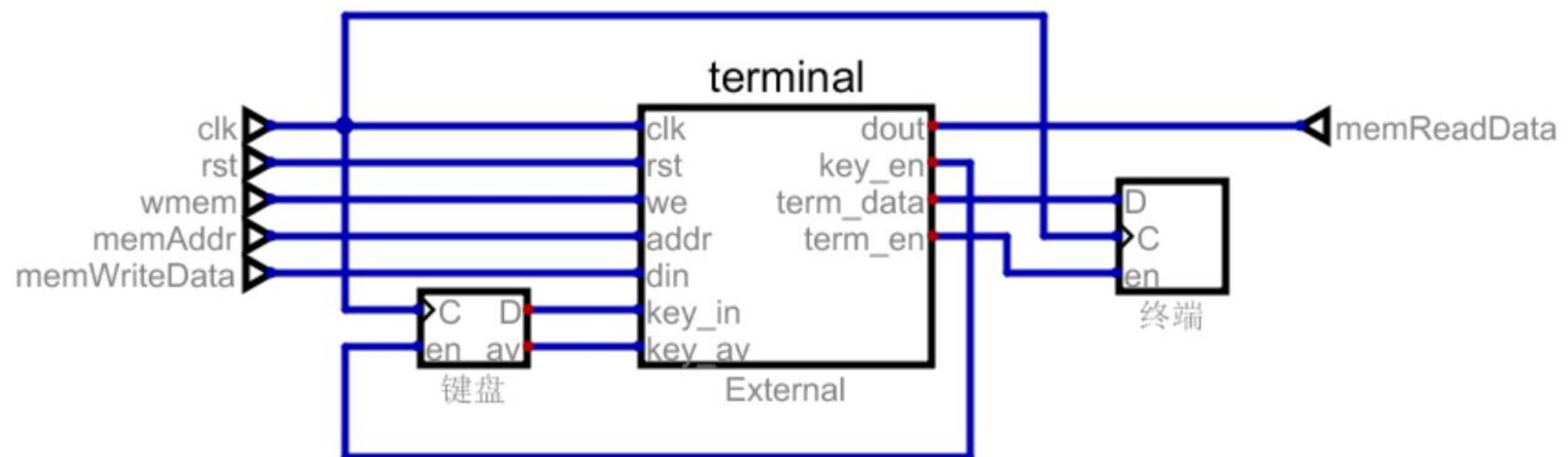
```
int main(){
    for(;;){
        switch(button()){
            case 1:
                led(1);
                break;
            case 2:
                led(3);
                break;
            case 4:
                led(7);
                break;
            case 8:
                ...
        }
    }
}
```



■ 示例2：键盘与终端

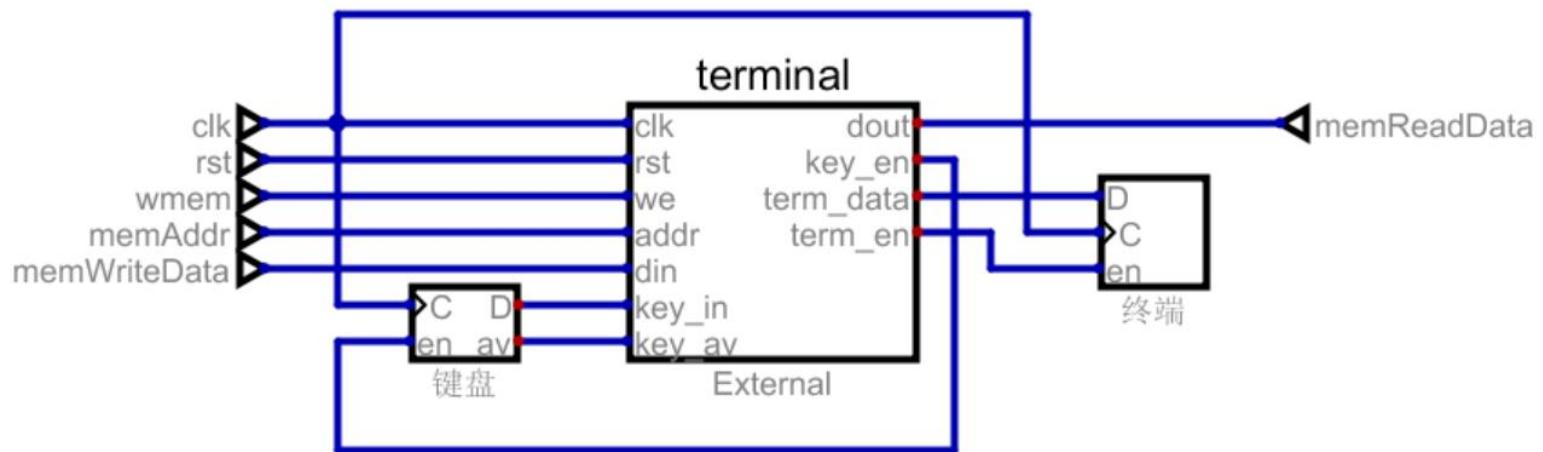
终端：0xffff_0030

读：返回一个字符
写：显示字符

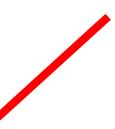


■ 示例2：键盘与终端（接口逻辑）

```
wire sel;  
assign sel = addr == 32'hffff_0030;  
  
//显示字符  
assign term_en = sel & we;  
assign term_data = din[7:0];  
  
//读键盘  
always @ (posedge clk)  
begin  
    if(sel && ~we && key_av)  
        key_en = 1;  
    else  
        key_en = 0;  
end  
  
assign dout = ~sel ? 32'bz :  
    we ? 32'bz :  
    ~key_av ? 32'd0 : {24'd0, key_in[7:0]};
```



没有输入的话，返回0；否则返回输入字符



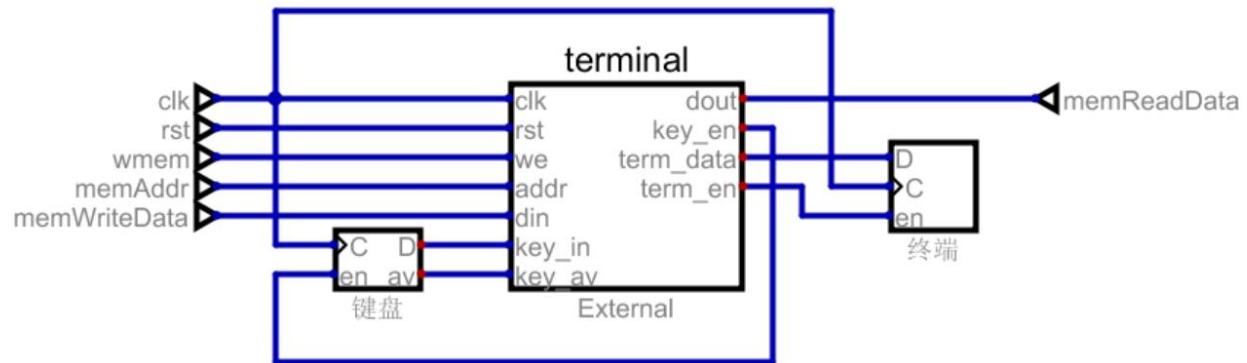
示例2：键盘与终端

```
#define term_base 0xffff0030

void putc(char c){
    IOW(term_base) = c & 0xff;
}
```

```
main.c:18:6: warning: conflicting types for built-in function 'putc'; expected 'int(in
builtin-declaration-mismatch]
 18 | void putc(char c){
     | ^~~~
main.c:1:1: note: 'putc' is declared in header '<stdio.h>'
```

```
int getc(){
    return IOW(term_base) & 0xff;
}
```

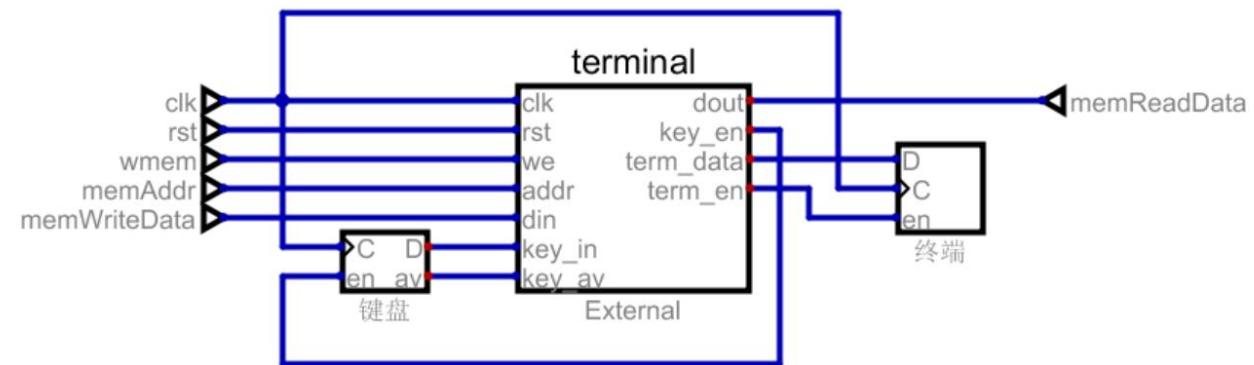


■ 示例2：键盘与终端

在终端上循环显示: hello

```
int main(){
    for(;;){
        putc('h');
        putc('e');
        putc('l');
        putc('l');
        putc('o');
        putc('\n');

        delay(10);
    }
}
```

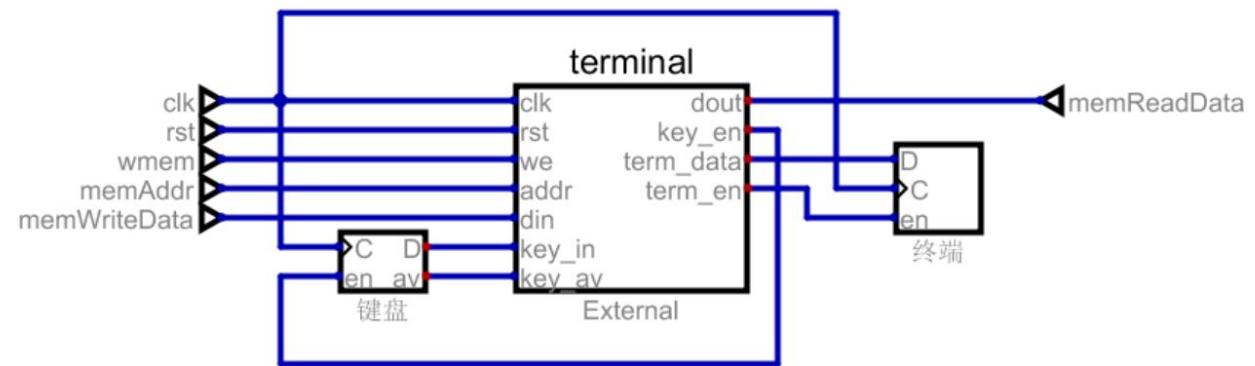


```
void delay(int c){
    while(c--){}

}
```

■ 示例2：键盘与终端

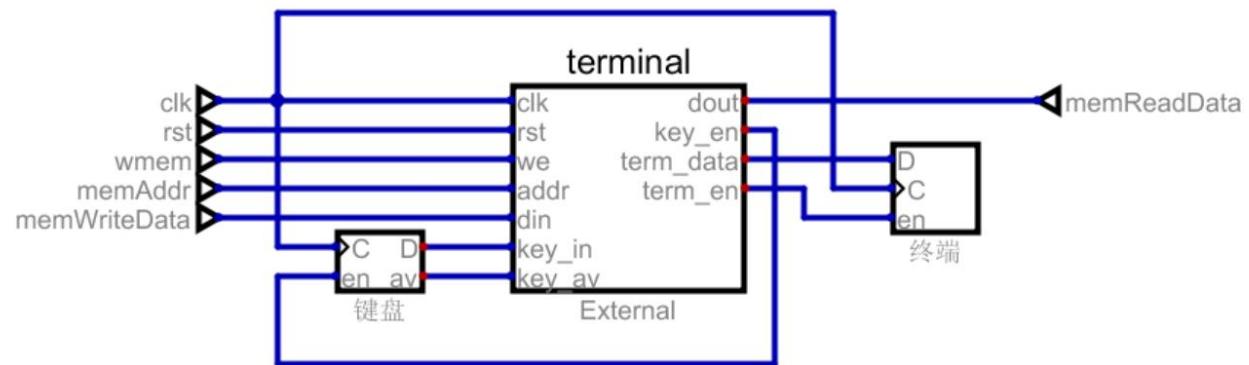
在终端上循环显示：hello



示例2：键盘与终端

显示字符串：

```
void puts(char *s){  
    while(*s){  
        putc(*s++);  
    }  
}  
  
int main(){  
    for(;;){  
        print("hello world!\n");  
  
        delay(10);  
    }  
}
```

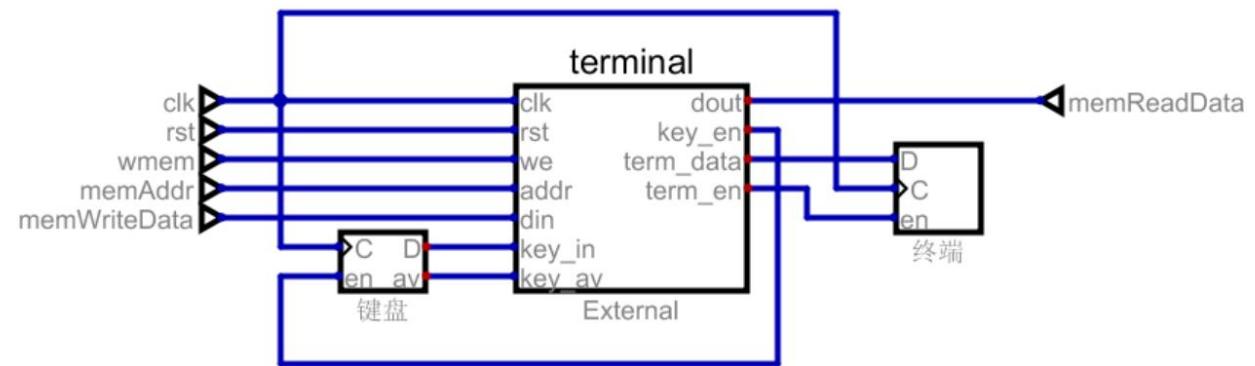


■ 示例2：键盘与终端

显示整数：

- puth: 十六进制
- putd: 十进制

```
void puth(int d){  
    for(int i = 7;i>= 0; i--){  
        putc("0123456789abcdef"[(d >> (i * 4)) & 0xf]);  
    }  
}
```

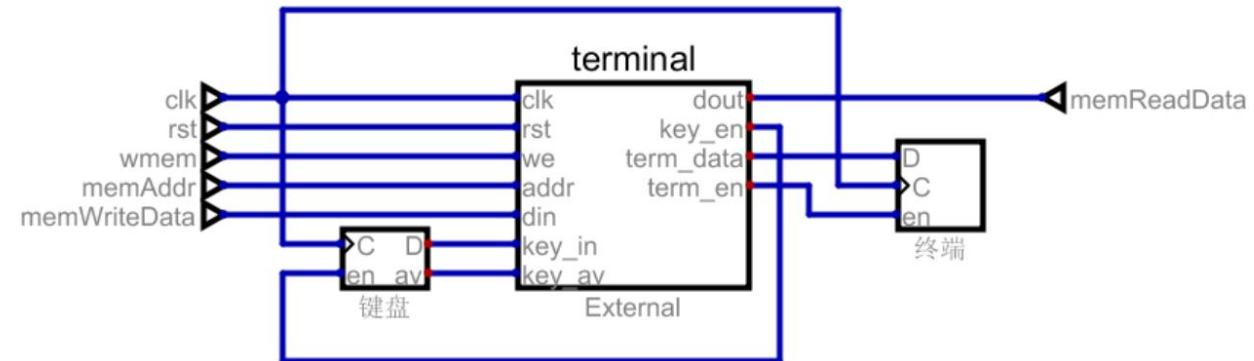


■ 示例2：键盘与终端

显示整数：

- putd: 十进制

```
void putd(int d){  
    if(d < 0){  
        putc('-'); 递归版本  
        d = -d;  
    }  
  
    if(d < 10){  
        putc(d + '0');  
    } else {  
        putd(d/10);  
        putc((d % 10) + '0');  
    }  
}
```



处理器不支持整数乘除、取模等运算
整数的除法、取模运算由libgcc支持__divsi3、__modsi3

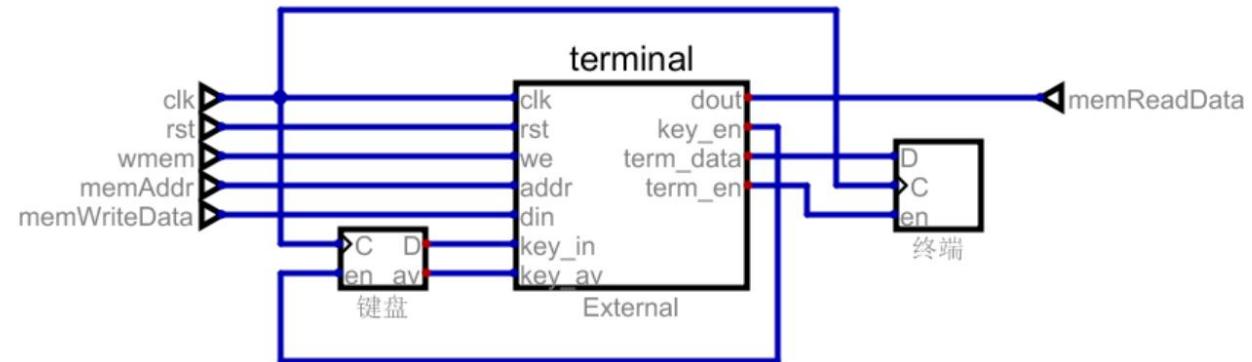
■ 示例2：键盘与终端

显示整数：

- putd: 十进制

```
void putd_iter(int d){  
    if(d < 0){  
        putc('-');  
        d = -d;  
    }  
  
    int divisor = 1;  
    while(d / divisor > 9){  
        divisor = divisor * 10;  
    }  
  
    while(divisor > 0){  
        putc(d / divisor + '0');  
        d = d % divisor;  
        divisor = divisor / 10;  
    }  
}
```

非递归版本



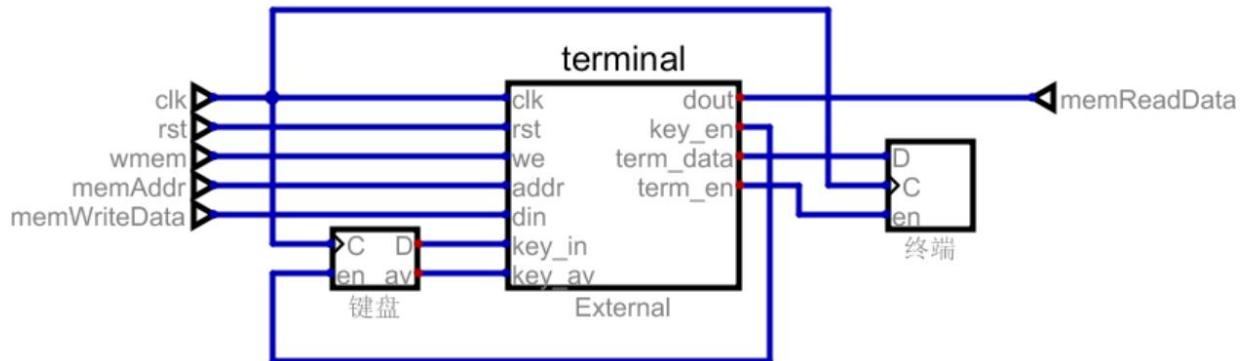
■ 示例2：键盘与终端

简化版的printf: 支持%d、%x、%s

可变参数:

```
#include <stdarg.h>
```

```
void printf(char *fmt, ...){  
    va_list args;  
  
    va_start(args, fmt);  
  
    while(*fmt){  
        if(*fmt != '%')  
            putc(*fmt);  
        else {  
            fmt++;  
            switch(*fmt){  
                case 0:  
                    putc('%');  
                    goto end;  
                case '%':  
                    putc('%');  
                    break;  
                case 's':  
                    char *s = va_arg(args, char *);  
                    puts(s);  
                    break;  
                case 'd':  
                    int d = va_arg(args, int);  
                    putd(d);  
                    break;  
                case 'x':  
                    int x = va_arg(args, int);  
                    puth(x);  
                    break;  
            }  
            fmt++;  
        }  
    }  
end:  
    va_end(args);  
}
```



```
case 's':  
    char *s = va_arg(args, char *);  
    puts(s);  
    break;  
case 'd':  
    int d = va_arg(args, int);  
    putd(d);  
    break;  
case 'x':  
    int x = va_arg(args, int);  
    puth(x);  
    break;  
}  
}  
fmt++;  
}  
}  
end:  
va_end(args);  
}
```

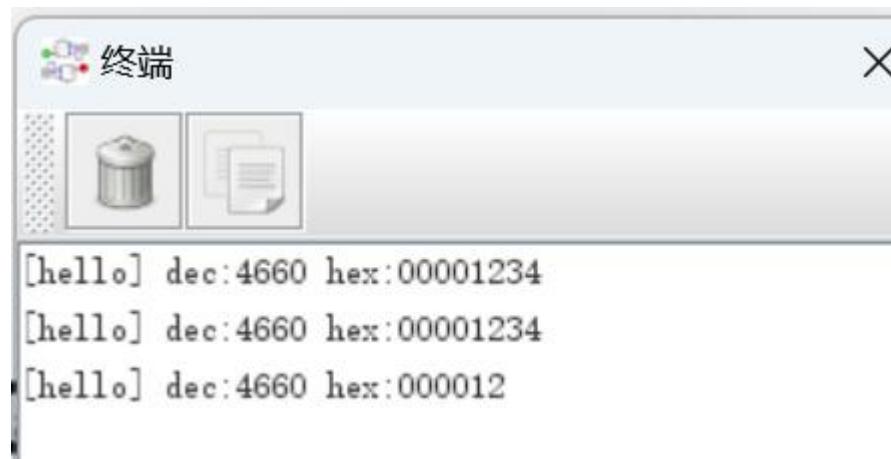
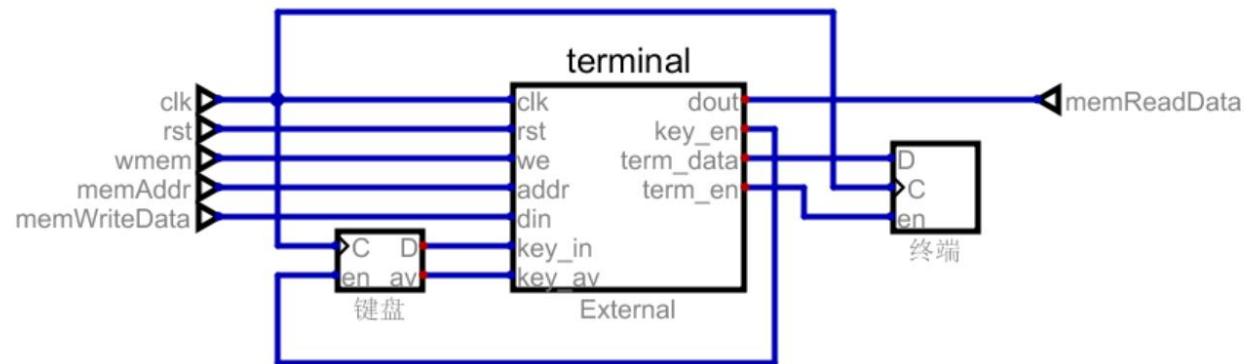
■ 示例2：键盘与终端

简化版的printf: 支持%d、%x、%s

可变参数:

```
#include <stdarg.h>
```

```
int main(){
    for(;;){
        printf("[%s] dec:%d hex:%x\n", "hello", 0x1234, 0x1234);
        delay(10);
    }
}
```

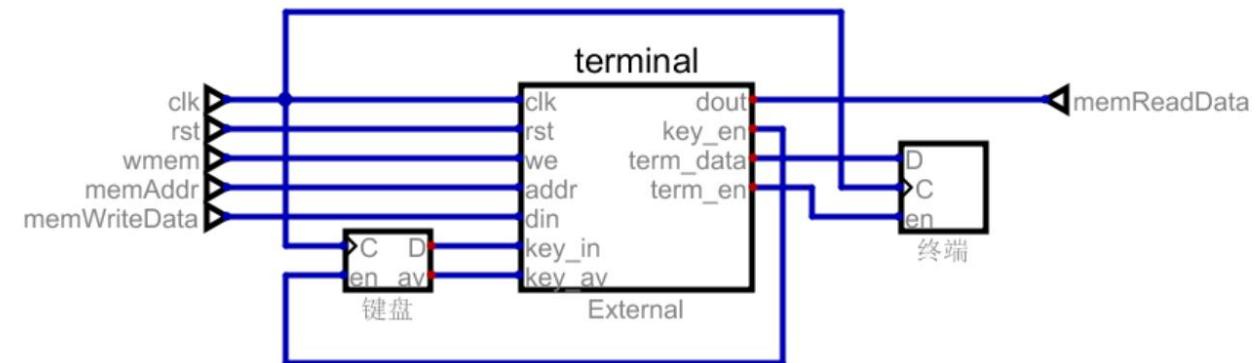


■ 示例2：键盘与终端

读字符

```
char getc(){
    return IOB(term_base);
}
```

```
int main(){
    char c;
    for(;;){
        c = getc();
        if(c != 0)
            putc(c);
    }
}
```



■ 示例2：键盘与终端

读字符串（回车结束，不包括回车符）

```
int main(){
    char c;
    char buf[100];
    char *s;

    for(;;){
        s = gets(buf);

        printf("%s\n", s);
    }
}
```

```
char *gets(char *s){
    char *p = s;
    char t;
    while(1){
        t = getc();

        if(t == 0)
            continue;

        if(t == '\n'){
            putc('\n');

            *p = 0;
            break;
        }

        putc(t); //回显

        *p = t;
        p++;
    }

    return s;
}
```

■ 示例2：键盘与终端

读整数（回车结束，不包括回车符）

```
#include <stdlib.h>

int getd(){
    char buf[100];
    char *p = gets(buf);

    return atoi(p);
}
```

```
int main(){
    for(;;){
        int i = getd();
        printf("%d\n", i);
    }
}
```

■ 示例3：计数器及程序运行时间（时钟周期数）

读计数器

64位计数器
long long

```
#define time_base 0xff

long long gettime(){
    int t0,t1;
    int low;

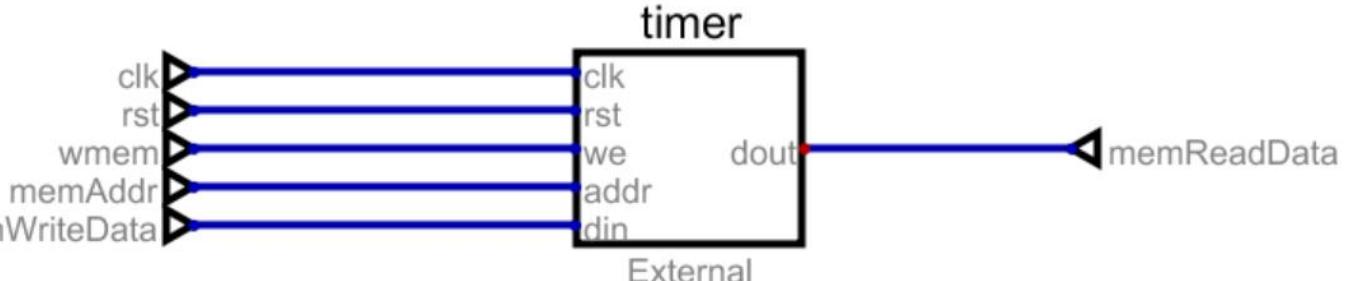
    t0 = IOW(time_base+4);

    while(1){
        low = IOW(time_base);
        t1 = IOW(time_base + 4);

        if(t1 == t0 )
            break;

        t0 = t1;
    }

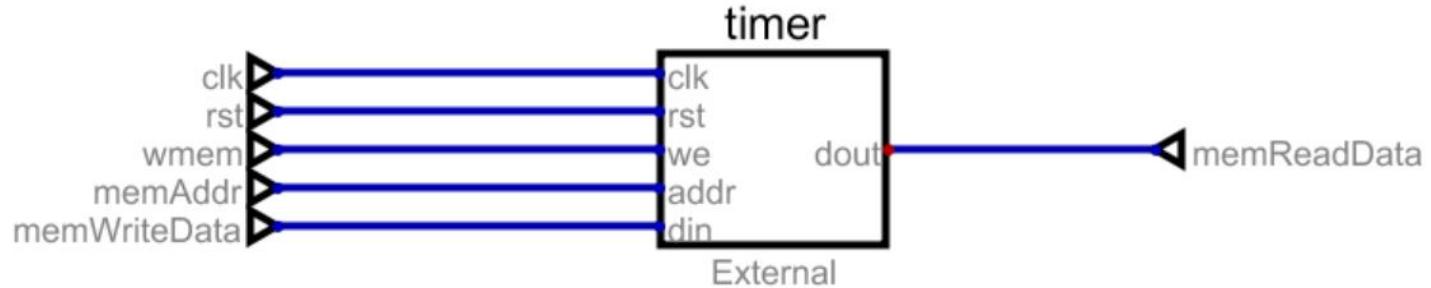
    return ((long long)t1) << 32) + low;
}
```



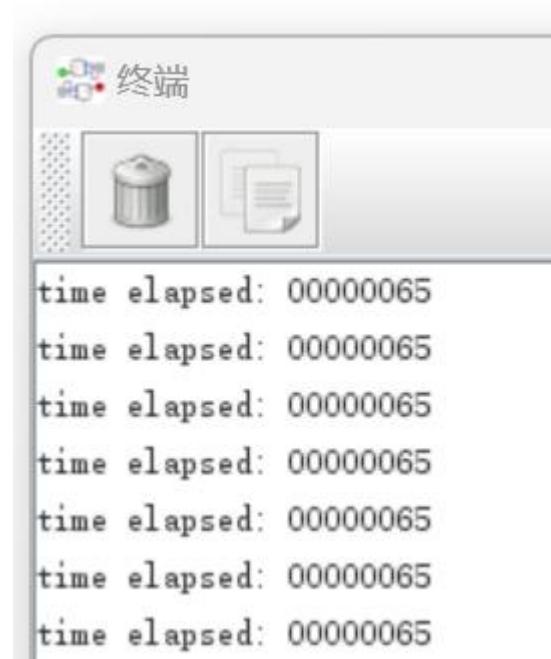
```
int main(){
    for(;;){
        int i = gettime();
        printf("%x\n",i);
    }
}
```

■ 示例3：计数器及运行时间（时钟周期数）

运行时间



```
int main(){
    for(;;){
        int start = gettimeofday();
        delay(10);
        int end = gettimeofday();
        printf("time elapsed: %x\n",end - start);
    }
}
```



■ 示例4：图形显示

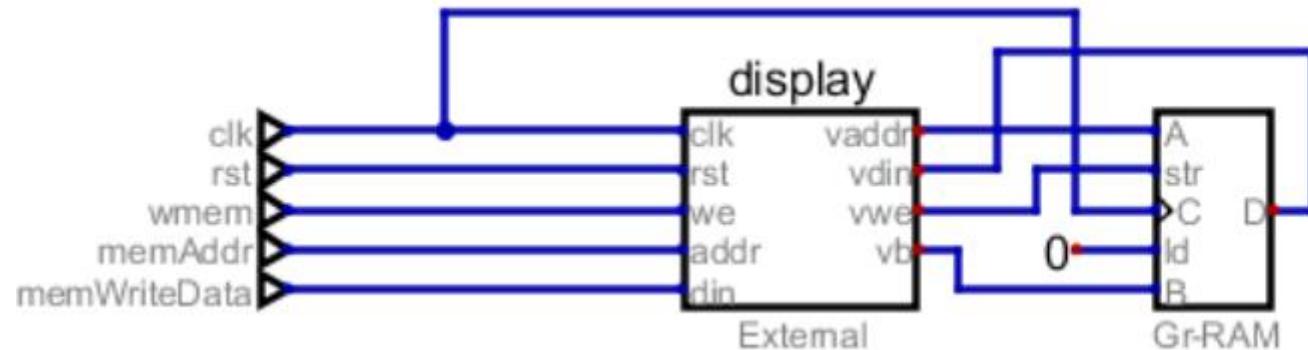
显存地址:

0xffff_8000 – 0xffff_ffff

大小: 128 * 128

画像素点: pixel(x, y, color)

```
void pixel(int x, int y, int color){  
    IOB(disp_base + WIDTH * x + y) = color;  
}
```



■ 示例4：图形显示

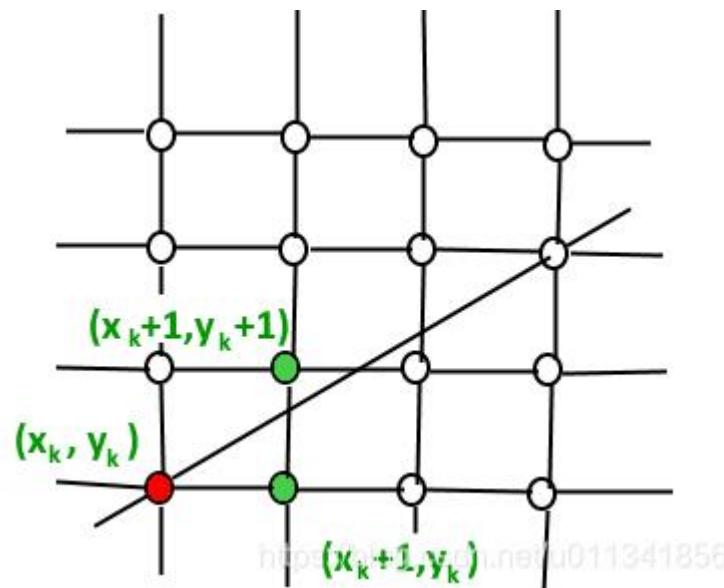
随机画点

```
#include <stdlib.h>
int main(){
    for(;;){
        int x,y,c;
        x = random() % WIDTH;
        y = random() % HEIGHT;
        c = random() % 10;
        pixel(x,y,c);
    }
}
```



■ 示例4：图形显示

直线 (Bresenham 算法)



```
void line(int x0, int y0, int x1, int y1, int color) {  
    int dx = abs(x1 - x0);  
    int dy = abs(y1 - y0);  
    int sx = (x0 < x1) ? 1 : -1;  
    int sy = (y0 < y1) ? 1 : -1;  
    int err = dx - dy;  
  
    while (1) {  
        pixel(x0, y0, color);  
        if (x0 == x1 && y0 == y1) break;  
        int e2 = 2 * err;  
        if (e2 > -dy) {  
            err -= dy;  
            x0 += sx;  
        }  
        if (e2 < dx) {  
            err += dx;  
            y0 += sy;  
        }  
    }  
}
```

■ 示例4：图形显示

直线（Bresenham 算法）

```
int main(){

    int x0,y0,x,y,c;
    x0 = random() % WIDTH;
    y0 = random() % HEIGHT;

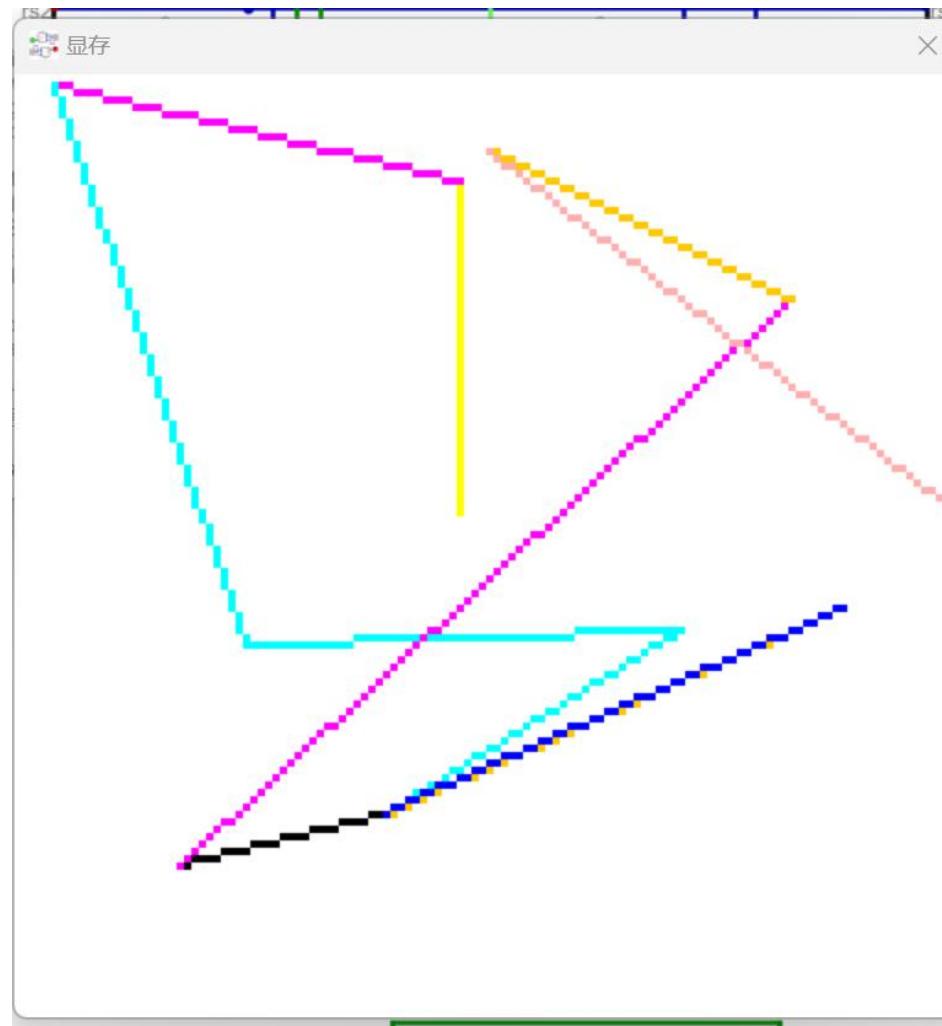
    for(;;){

        x = random() % WIDTH;
        y = random() % HEIGHT;

        c = random() % 10;

        line(x0,y0,x,y,c);
        x0 = x;
        y0 = y;

    }
}
```



■ 示例4：图形显示

矩形

```
void rectangle(int x_min, int y_min, int x_max, int y_max, int color) {  
    // 顶边  
    line(x_min, y_min, x_max, y_min, color);  
    // 右边  
    line(x_max, y_min, x_max, y_max, color);  
    // 底边  
    line(x_max, y_max, x_min, y_max, color);  
    // 左边  
    line(x_min, y_max, x_min, y_min, color);  
}
```

■ 示例4：图形显示

矩形

```
int main(){

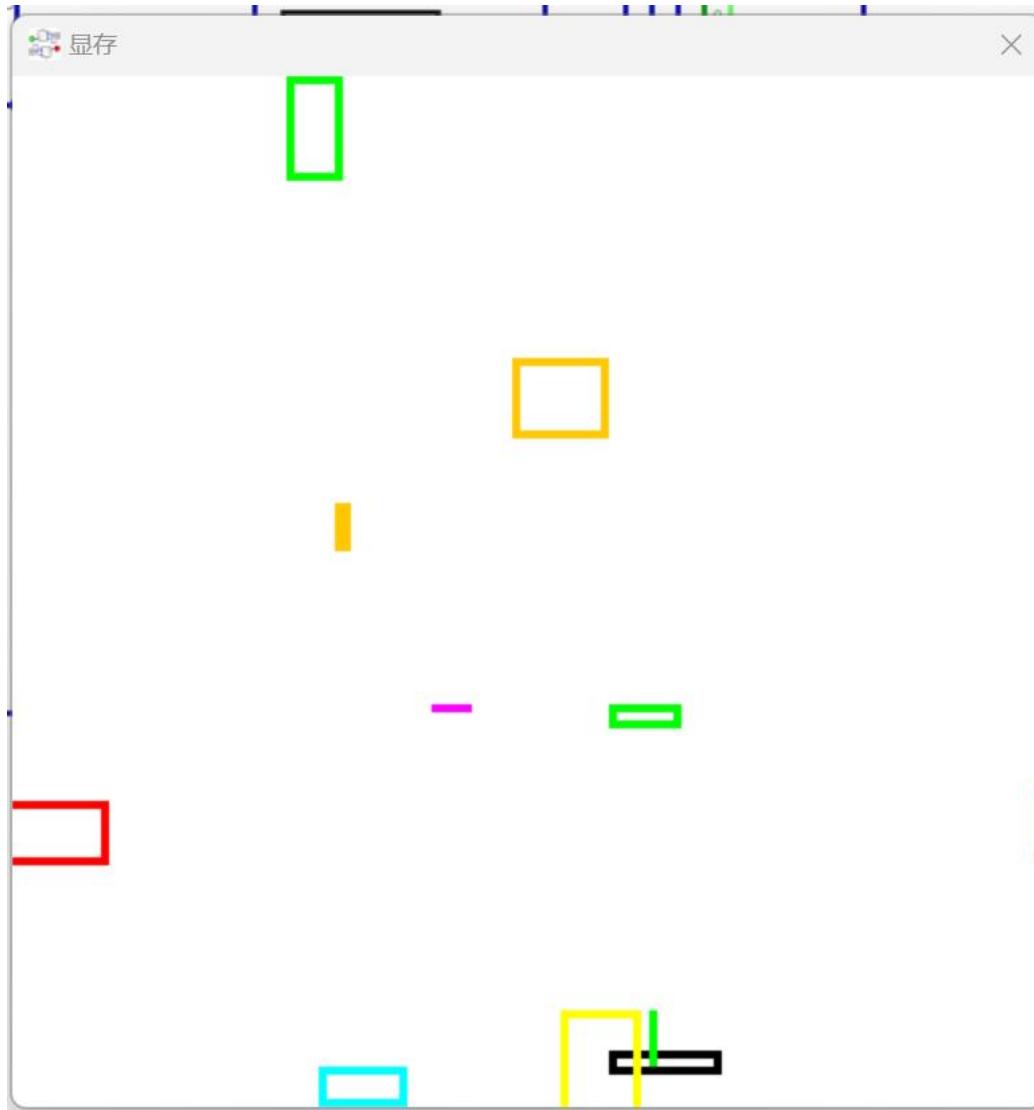
    int x,y,w,h,c;

    for(;;){

        x = random() % WIDTH;
        y = random() % HEIGHT;
        w = random() % (WIDTH/8);
        h = random() % (HEIGHT/8);

        c = random() % 10;

        rectangle(x,y,x+w,y+h, c);
    }
}
```



■ 示例4：图形显示

绘制多边形

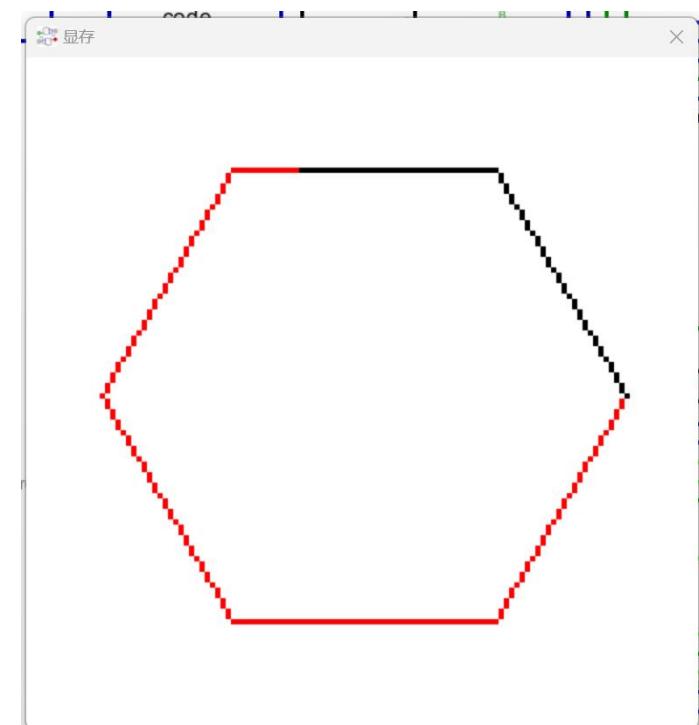
假定顶点个数为n，顶点坐标放在二
维数组vertices中

```
void polygon(int n, int vertices[][][2], int color) {  
    for (int i = 0; i < n; i++) {  
        int x0 = vertices[i][0];  
        int y0 = vertices[i][1];  
        int x1 = vertices[(i + 1) % n][0];  
        int y1 = vertices[(i + 1) % n][1];  
        line(x0, y0, x1, y1, color);  
    }  
}
```

■ 示例4：图形显示

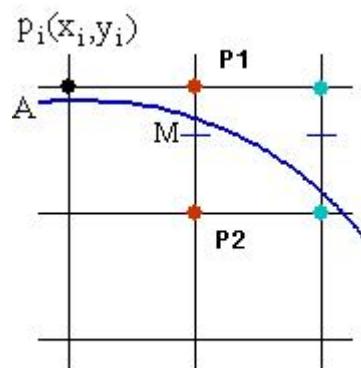
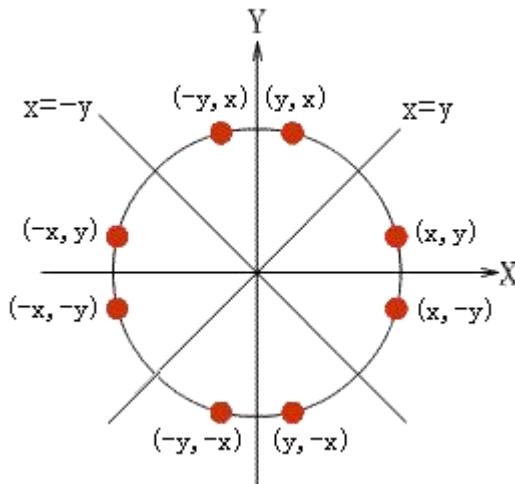
正六边形

```
int v[][2] = {  
    {64,114},  
    {21,89},  
    {21,39},  
    {64,14},  
    {107,39},  
    {107,89}  
};  
  
for(;;){  
    c = random() % 10;  
    polygon(6, v, c);  
}
```



示例4：图形显示

圆形（中点画圆算法）

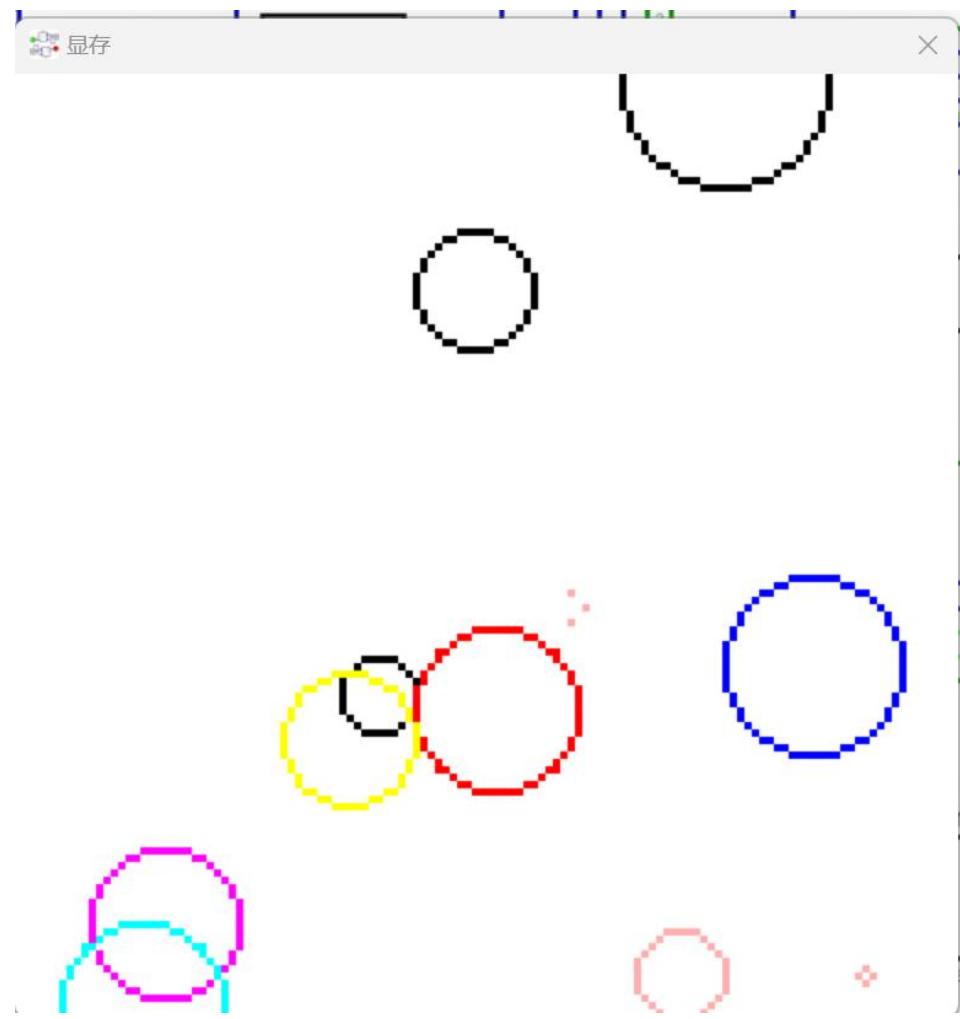


```
void circle(int x_center, int y_center, int radius, int color) {  
    int x = 0;  
    int y = radius;  
    int d = 3 - 2 * radius;  
  
    while (x <= y) {  
        pixel(x_center + x, y_center + y, color);  
        pixel(x_center + y, y_center + x, color);  
        pixel(x_center - y, y_center + x, color);  
        pixel(x_center - x, y_center + y, color);  
        pixel(x_center - x, y_center - y, color);  
        pixel(x_center - y, y_center - x, color);  
        pixel(x_center + y, y_center - x, color);  
        pixel(x_center + x, y_center - y, color);  
  
        if (d < 0) {  
            d += 4 * x + 6;  
        } else {  
            d += 4 * (x - y) + 10;  
            y--;  
        }  
        x++;  
    }  
}
```

■ 示例4：图形显示

圆形（中点画圆算法）

```
int main(){
    int x,y,r,c;
    for(;;){
        x = random() % WIDTH;
        y = random() % HEIGHT;
        r = random() % (WIDTH/8);
        c = random() % 10;
        circle(x,y,r,c);
    }
}
```

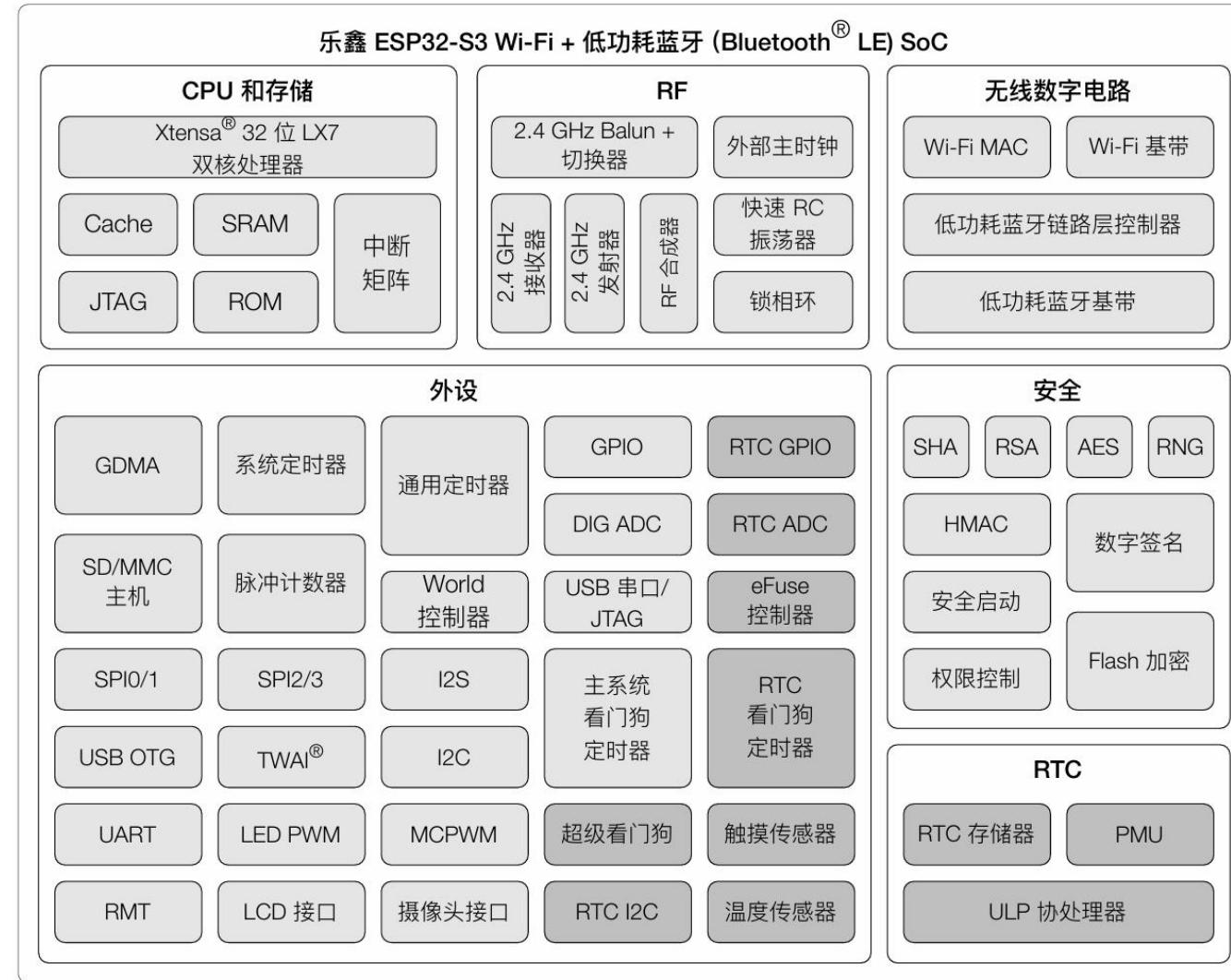


■ 嵌入式系统常用外设及接口

- GPIO
- PWM
- ADC/DAC
- 通信接口
 - UART
 - I2C
 - SPI
 - CAN
 - IIS
 - ...
- SDCard

■ 嵌入式系统常用外设及接口

- GPIO
- PWM
- ADC/DAC
- 通信接口
 - UART
 - I2C
 - SPI
 - CAN
 - IIS
 - ...
- SDCard

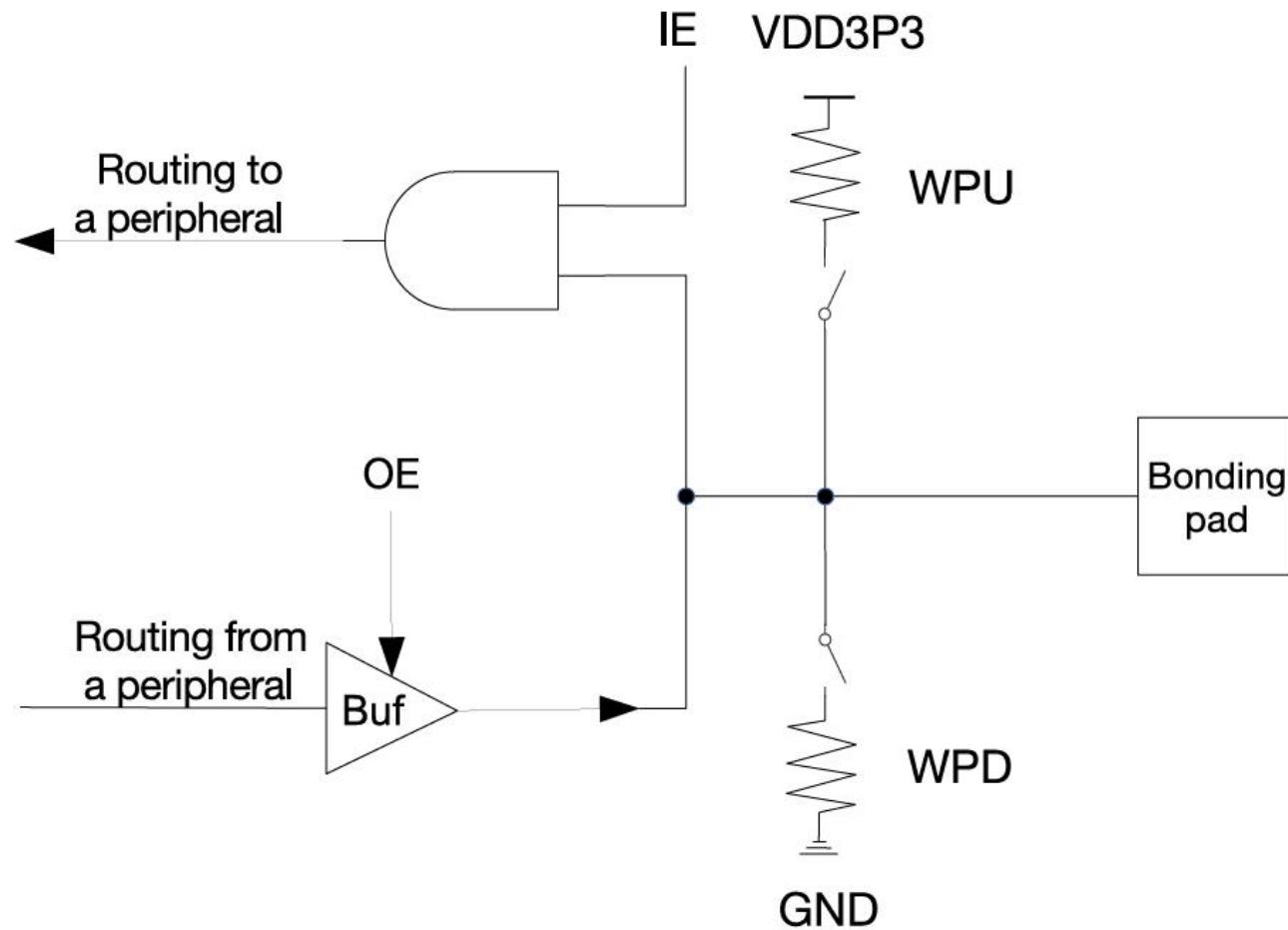


■ GPIO

- GPIO (General Purpose Input/Output, 通用输入/输出引脚) 是一种常见的硬件接口，广泛应用于微控制器 (MCU)、微处理器 (MPU) 和其他嵌入式系统中。GPIO 引脚可以被配置为输入模式或输出模式，用于与外部设备进行交互。数字量。
- 输入模式：
 - 用于读取外部信号，例如按键状态、传感器数据等。
 - 当 GPIO 配置为输入模式时，它可以检测引脚上的电平变化（高电平或低电平）。
- 输出模式：
 - 用于控制外部设备，例如点亮 LED、驱动继电器等。
 - 当 GPIO 配置为输出模式时，可以通过设置引脚的电平来控制外部设备。

■ GPIO内部结构

- IE: 输入使能
- OE: 输出使能
- WPU: 内部弱上拉
- WPD: 内部弱下拉



■ GPIO的应用场景

- 控制 LED：
 - 将 GPIO 配置为输出模式，通过设置引脚电平来点亮或熄灭 LED。
- 读取按键状态：
 - 将 GPIO 配置为输入模式，检测按键是否被按下（通过电平变化判断）。
- 读取传感器数据：
 - 将 GPIO 配置为输入模式，读取传感器输出的电平信号。
- 驱动继电器：
 - 将 GPIO 配置为输出模式，通过控制继电器的开关状态来控制大功率设备。

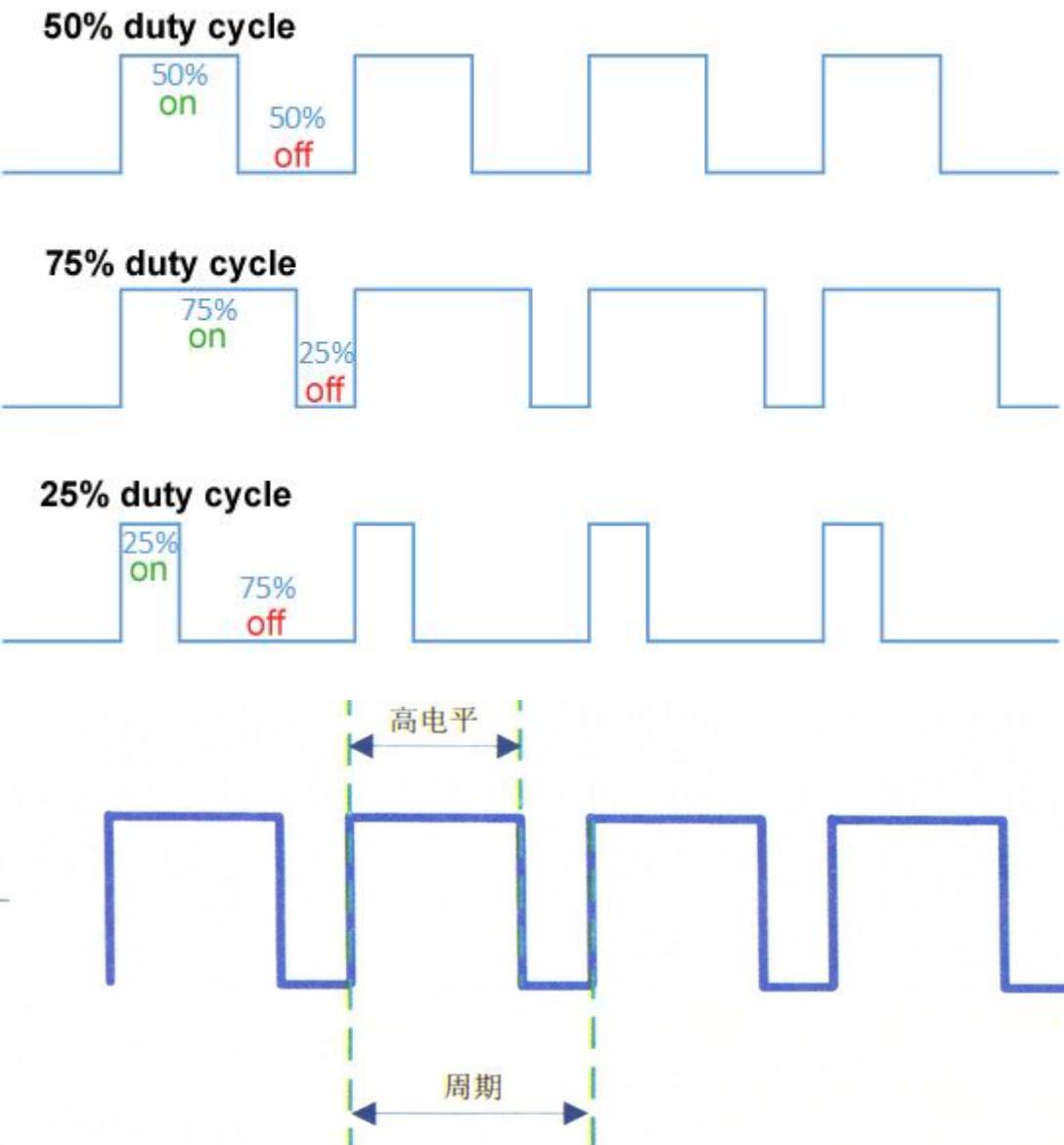
■ PWM

- PWM (Pulse Width Modulation, 脉冲宽度调制) 是一种通过改变脉冲信号的占空比来控制信号输出的调制技术。它广泛应用于电子、电气和嵌入式系统中，用于控制电机速度、LED 亮度、音频信号等。
- 数字量输出，连续变化
- PWM 的基本原理
 - PWM 信号是一种周期性变化的方波信号，其占空比 (Duty Cycle) 可以改变。占空比是指脉冲高电平时间与周期的比值。通过改变占空比，可以控制输出功率或信号强度。

PWM

- 关键参数：
 - 周期 (Period)：PWM 信号的周期，通常用时间（如微秒或毫秒）表示。
 - 频率 (Frequency)：PWM 信号的频率，等于 1/周期。
 - 占空比 (Duty Cycle)：高电平时间与周期的比值，通常用百分比表示。

$$\text{占空比} = \frac{\text{高电平时间}}{\text{周期}}$$



■ PWM的应用场景

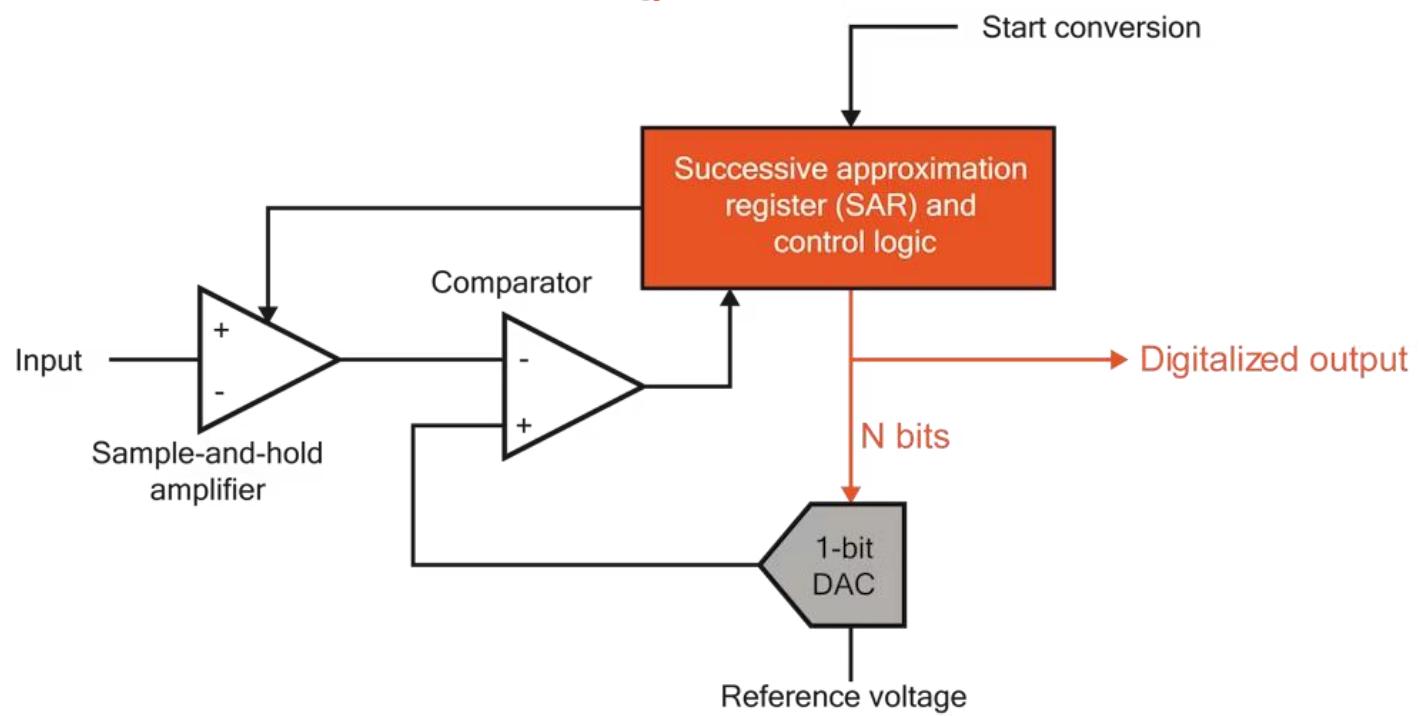
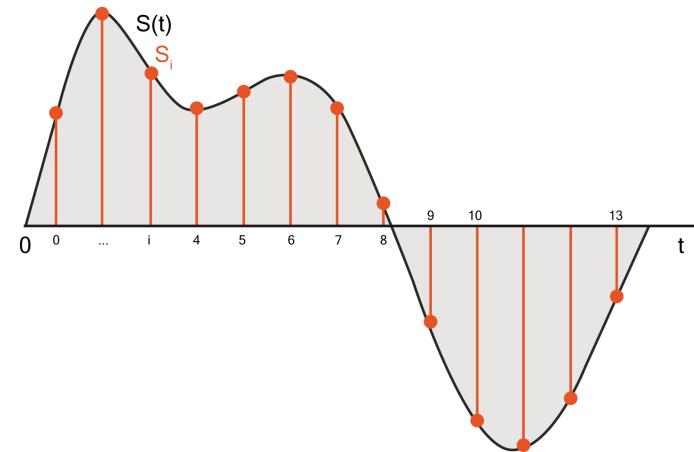
- 电机控制
 - 直流电机：通过改变 PWM 的占空比，可以控制直流电机的转速。
 - 步进电机：通过 PWM 信号控制步进电机的步进速度。
- LED 调光
 - 通过改变 PWM 的占空比，可以控制 LED 的亮度。高占空比使 LED 更亮，低占空比使 LED 更暗。
- 音频信号处理
 - PWM 可以用于生成音频信号，通过改变占空比和频率，可以生成不同音调的声音。
- 电源管理
 - PWM 可以用于控制电源的输出，例如在开关电源中调节输出电压。

■ ADC/DAC

- ADC 和 DAC 是数字系统与模拟世界之间的桥梁。
 - ADC 将模拟信号转换为数字信号，使得数字系统能够处理和分析来自外部世界的模拟信息；DAC 将数字信号转换为模拟信号，使得数字系统能够与外部世界进行交互和控制。
- ADC (Analog-to-Digital Converter, 模数转换器)
 - 是一种将模拟信号转换为数字信号的设备。其主要功能是将连续变化的模拟信号（如电压、电流等）转换为离散的数字信号，以便数字系统（如微控制器、计算机等）进行处理和分析。
 - ADC 的工作原理通常包括采样、保持、量化和编码四个步骤。
- DAC (Digital-to-Analog Converter, 数模转换器)
 - 是一种将数字信号转换为模拟信号的设备。其主要功能是将离散的数字信号转换为连续变化的模拟信号，以便与外部世界进行交互。
 - DAC 的工作原理是通过数字信号控制一系列的电阻或电流源，生成对应的模拟电压或电流输出。

■ ADC

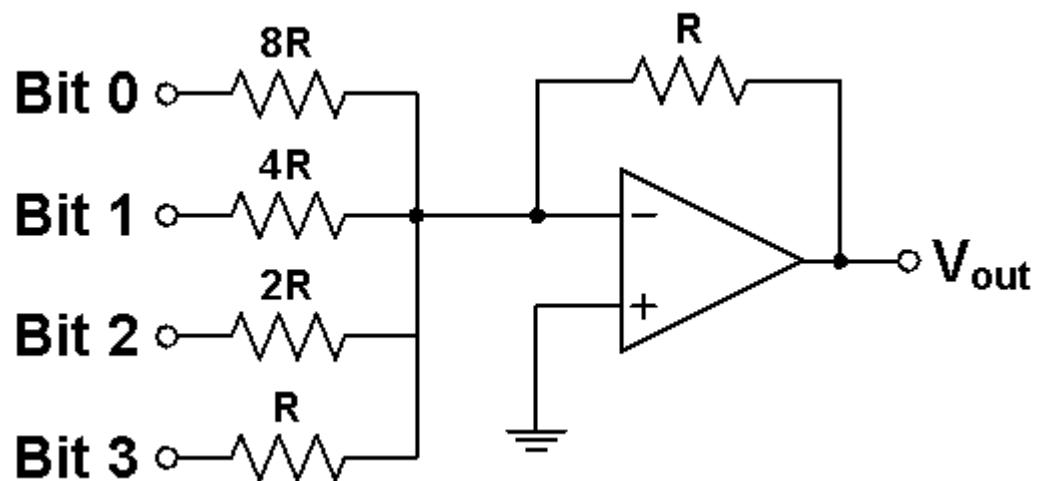
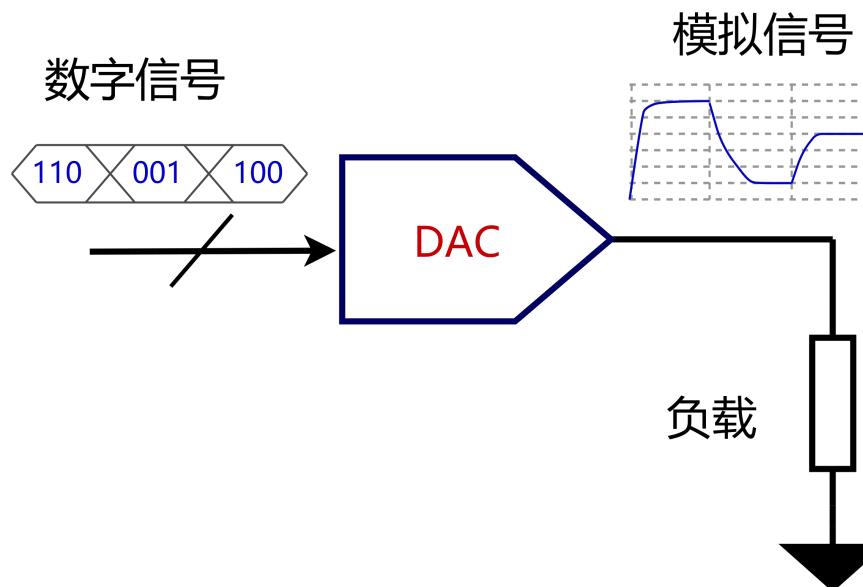
- 分辨率 (Resolution)
- 采样率 (Sampling Rate)
- 输入范围 (Input Range)
- 量化误差 (Quantization Error)
- 信噪比 (SNR)
- 线性度 (Linearity)
- 转换速度 (Conversion Speed)



DAC

- 分辨率 (Resolution)
- 失调和增益误差
- 建立时间 (Settling Time)
- 信噪比 (SNR) 和无杂波动态范围 (SFDR)
- 线性度 (Linearity)

$$V_{OUT} = -V_{ref} \left(\frac{1}{1} Bit3 + \frac{1}{2} Bit2 + \frac{1}{4} Bit1 + \frac{1}{8} Bit0 \right)$$



■ ADC/DAC的应用场景

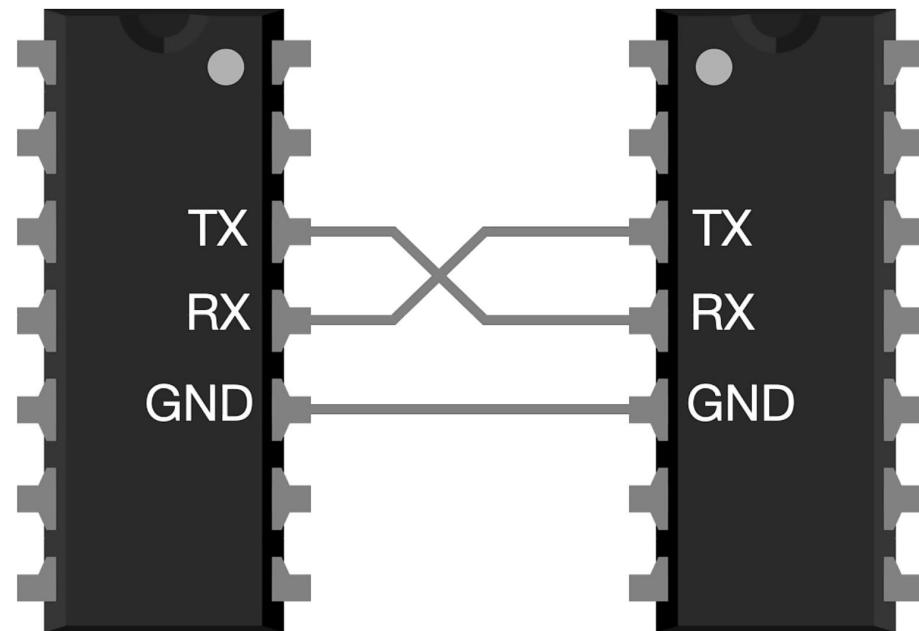
- ADC应用场景：
 - 传感器数据采集：将传感器采集到的模拟信号（如温度、压力、光强等）转换为数字信号，供微控制器或计算机进行处理和分析。
 - 音频信号处理：将麦克风采集到的模拟音频信号转换为数字信号，便于数字音频处理和存储。
 - 图像信号处理：将摄像头采集到的模拟图像信号转换为数字信号，便于数字图像处理和显示。
- DAC应用场景：
 - 音频信号输出：将数字音频信号转换为模拟信号，驱动扬声器或耳机播放声音。
 - 控制系统：将数字控制信号转换为模拟信号，用于控制电机、继电器等设备。
 - 信号发生器：生成各种模拟信号，用于测试和校准电子设备。

■ UART

- UART (Universal Asynchronous Receiver/Transmitter, 通用异步收发传输器) 是一种常用的异步串行通信协议，用于在电子设备之间进行数据传输。它是一种硬件通信协议，广泛应用于嵌入式系统、计算机、调制解调器和其他电子设备中。
- UART 通过串行通信的方式，将数据一位一位地发送或接收。其主要特点是异步通信，即发送和接收双方不需要共享时钟信号，而是通过起始位、数据位、停止位等来同步数据传输。

■ UART

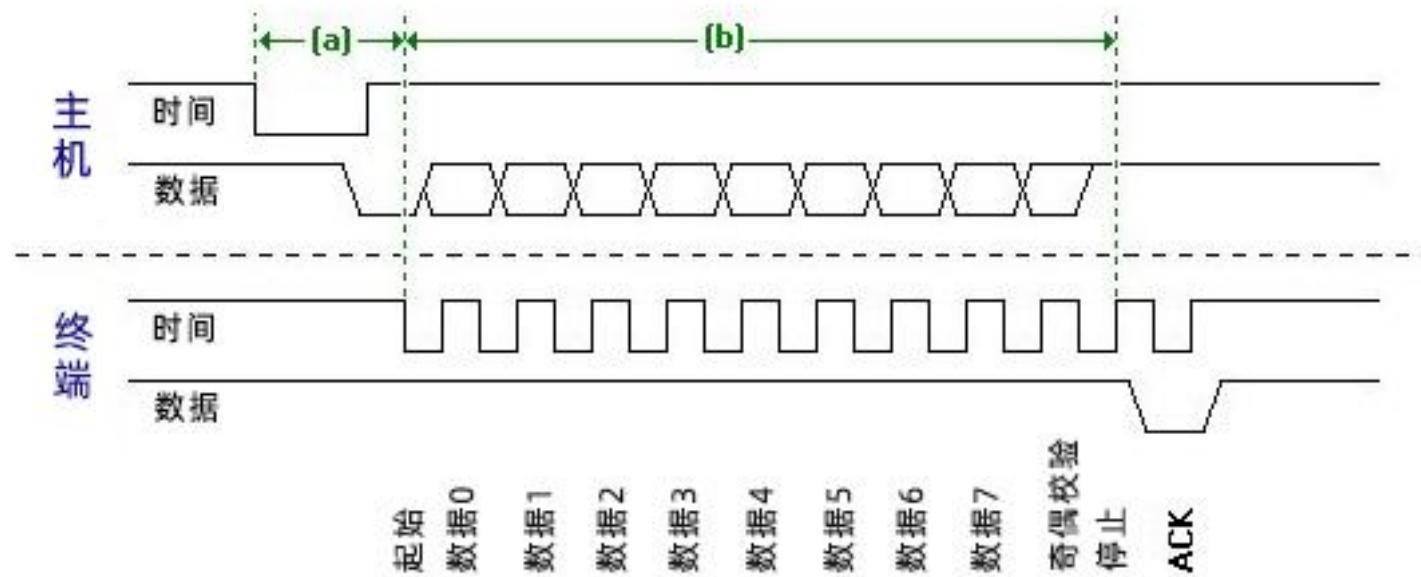
- UART 通常通过以下引脚进行通信：
 - TX (Transmit) : 发送数据引脚
 - RX (Receive) : 接收数据引脚
 - GND (Ground) : 地线
- RS232、RS485
 - UART是通信协议
 - RS232/485电气标准（物理层），定义信号的电气特性
- USB转串口、以太转串口、wifi转串口



■ UART

- 关键参数：

- 波特率 (Baud Rate) : 表示每秒传输的比特数，用于衡量数据传输的速度
- 数据位 (Data Bits) : 表示每个数据帧中用于传输实际数据的位数，通常为 5 到 8 位。
- 停止位 (Stop Bits) : 表示每个数据帧结束时的空闲位数，通常为 1 位或 2 位。
- 校验位 (Parity Bit) : 用于检测数据传输中的错误，可以是奇校验、偶校验或无校验。

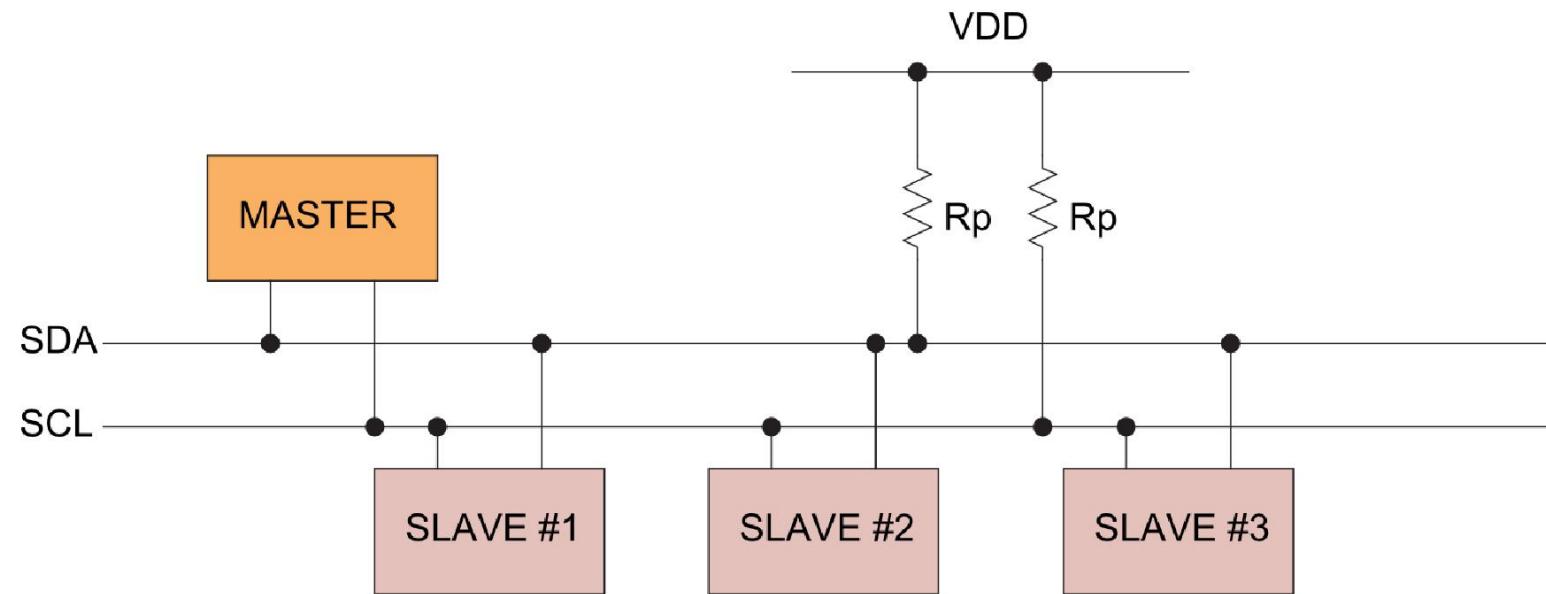


I2C

- I2C是一种用于连接微控制器和外围设备的两线式串行总线协议，由飞利浦半导体（现为恩智浦半导体）在1980年代开发。它主要用于在短距离内进行低速通信，广泛应用于各种电子设备和嵌入式系统中。有如下特点：
 - 简单的两线接口：I2C总线仅需要两根线进行通信：一根是串行数据线（SDA），另一根是串行时钟线（SCL）。
 - 多主多从架构：I2C总线上可以有多个主设备和从设备，主设备负责发起通信，从设备响应主设备的请求。
 - 设备地址：每个I2C设备都有一个唯一的7位或10位地址，用于在总线上标识设备。
 - 支持多种通信速率：I2C总线支持不同的通信速率，包括标准模式（100 kbps）、快速模式（400 kbps）、高速模式（3.4 Mbps）等。
 - 低功耗：I2C总线在空闲时消耗的电流非常小，适合用于电池供电的设备。

I2C工作原理

- 引脚定义
 - SDA (Serial Data) : 串行数据线, 用于传输数据。
 - SCL (Serial Clock) : 串行时钟线, 用于同步数据传输。

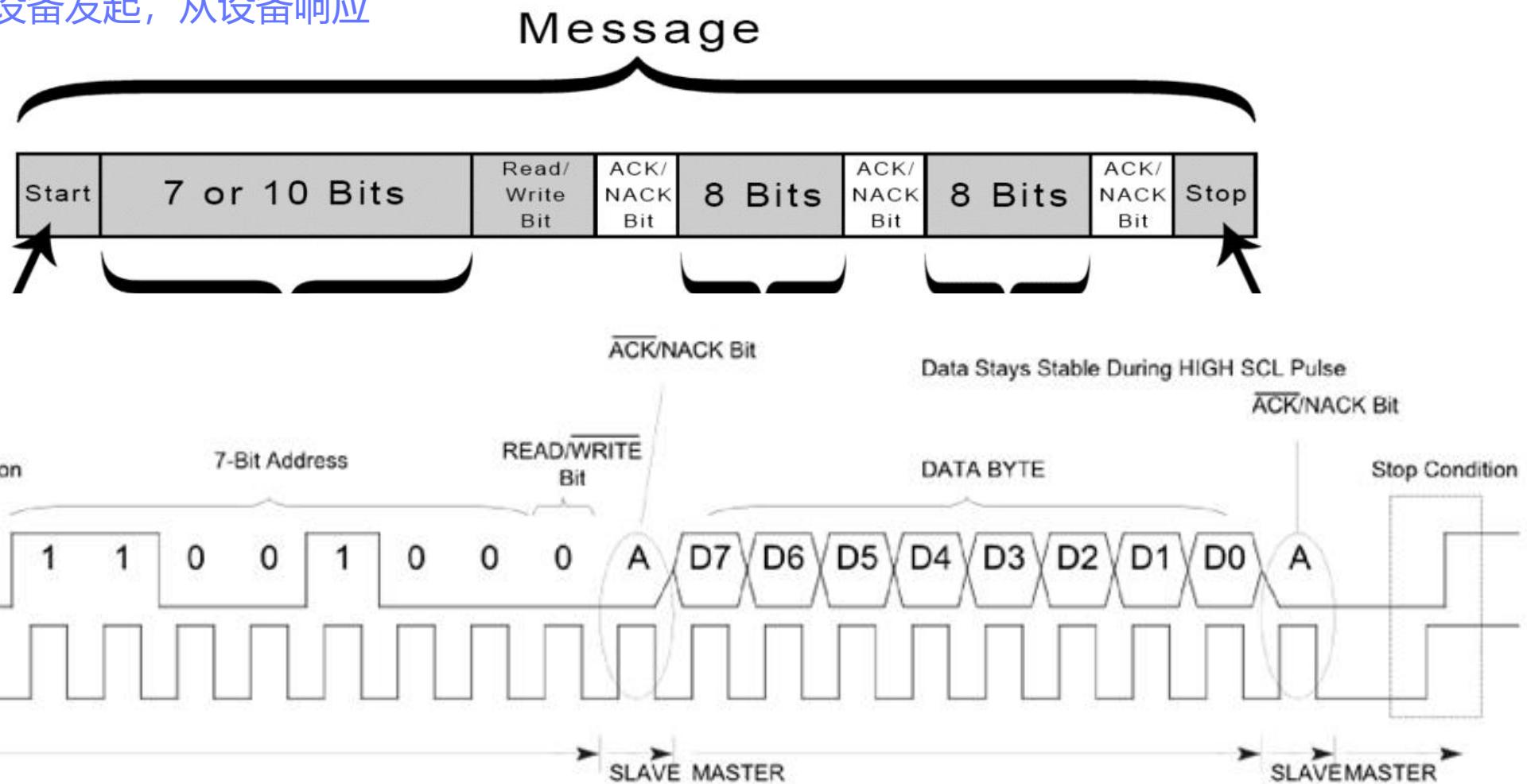


I2C工作原理

- 总线空闲状态：SDA和SCL均为高电平
- 起始条件：SCL为高电平时，SDA从高电平跳变到低电平，这表示一次通信的开始。
- 停止条件：SCL为高电平时，SDA从低电平跳变到高电平：这表示一次通信的结束。
- 数据传输
 - 数据有效性：在SCL的高电平期间，SDA上的数据必须保持稳定。
 - 数据采样：在SCL的低电平期间，主设备会采样SDA上的数据。
- 应答信号
 - 从设备应答：在每次数据传输后，从设备会发送一个应答信号（ACK）或非应答信号（NACK）。
 - 应答信号的产生：在SCL的第9个时钟周期，从设备会将SDA拉低以发送ACK信号。

I2C工作原理

- 通信是由主设备发起，从设备响应



I2C设备

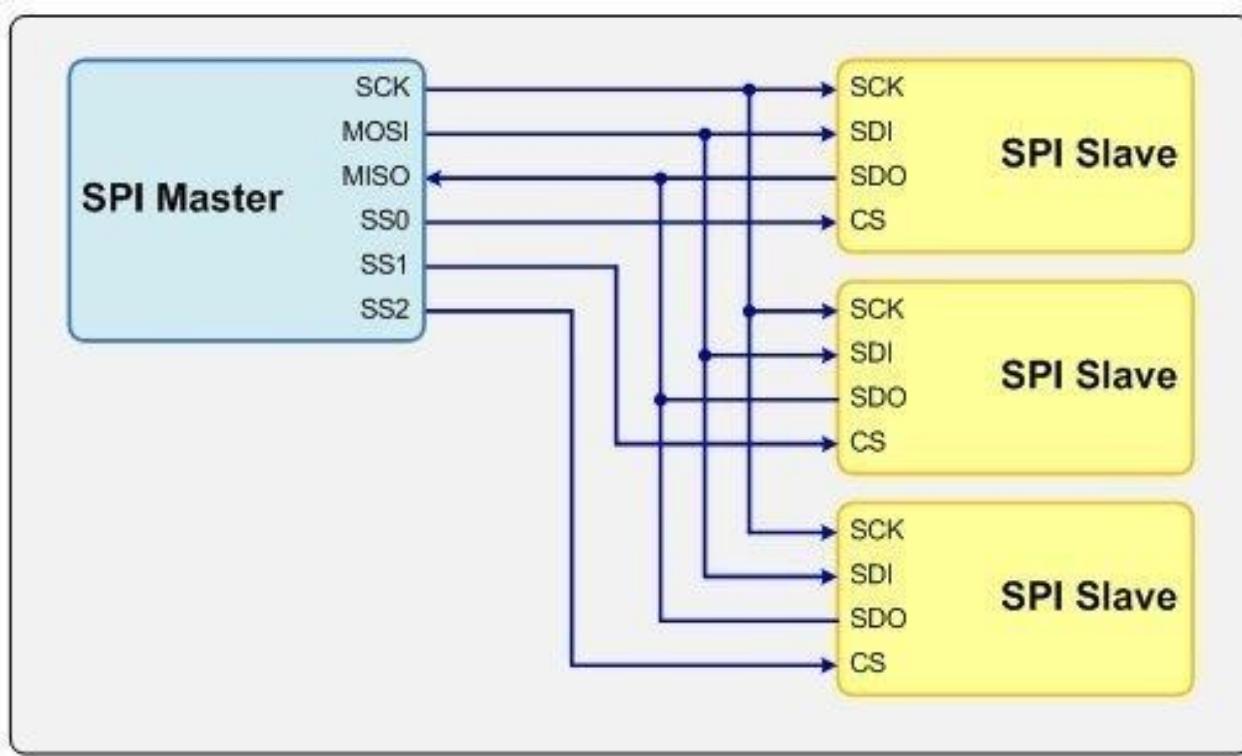
- 温度传感器：如LM75、TMP102等，通过I2C总线与微控制器通信，报告温度数据。
- 加速度传感器：如MPU6050、ADXL345等，通过I2C总线传输加速度数据。
- EEPROM芯片：如24LC02、24LC64等，通过I2C总线与微控制器通信，用于存储数据。
- 实时时钟 (RTC)：如DS1307、RTC_DS3231等，通过I2C总线与微控制器通信，提供时间信息。
- 电源管理芯片：如INA219、INA260等，通过I2C总线报告电流、电压等信息。
- ...

■ SPI

- SPI (Serial Peripheral Interface) 总线
 - 是一种同步串行通信协议，用于在微控制器（MCU）和外部设备之间进行数据传输。它由摩托罗拉公司在20世纪80年代中期开发，后逐渐发展成为行业规范。
 - SPI以其高效的数据传输能力和简单的硬件接口设计，在嵌入式系统、微控制器与各种外围设备之间的通信中占据重要地位。
- 特点
 - 高速的，全双工，同步。（I2C/RS485：半双工, RS232：全双工）
 - 主从模式：SPI通信采用主从模式，通常由一个主设备和一个或多个从设备组成。主设备负责控制通信过程，包括时钟信号的生成、从设备的选择以及数据的发送与接收。从设备则根据主设备的控制信号进行响应，完成数据的接收或发送。

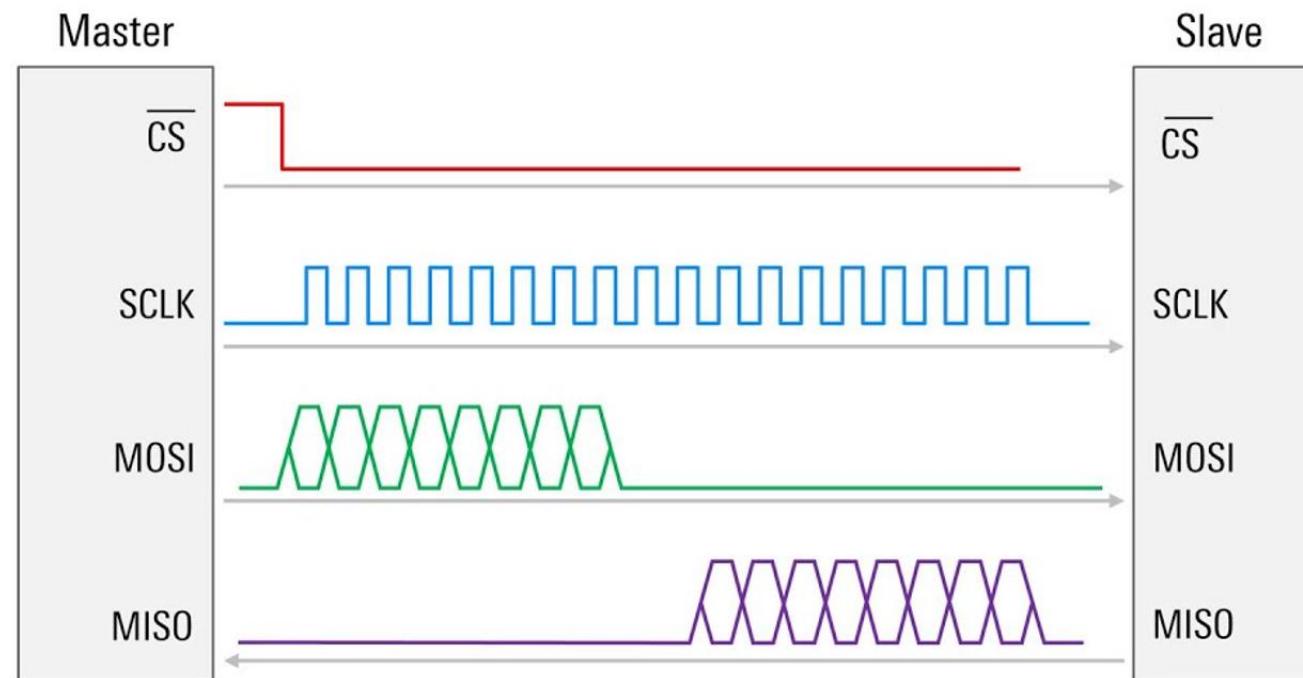
SPI

- 四根信号线：SPI接口一般使用四条信号线通信
 - MISO（主设备输入/从设备输出引脚）
 - MOSI（主设备输出/从设备输入引脚）
 - SCK（串行时钟信号）
 - CS（从设备片选信号）



SPI工作原理

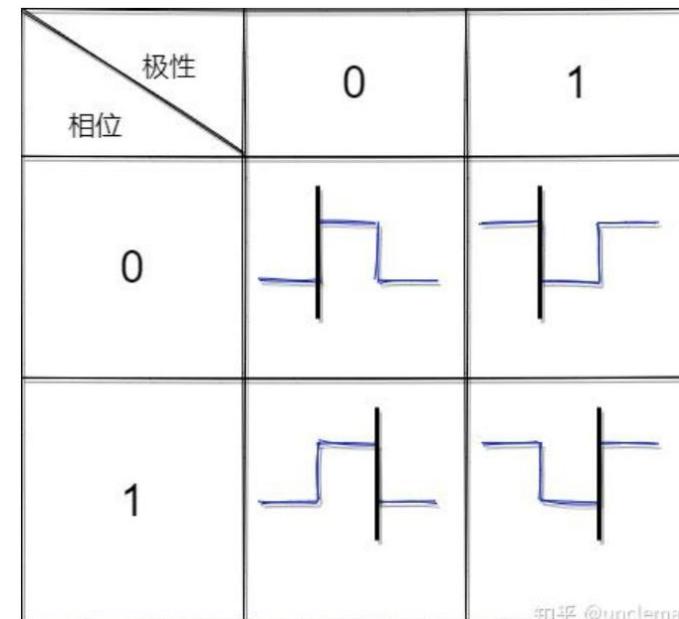
- 起始条件：通信开始时，主设备通过拉低CS信号线来选择从设备，并开始发送时钟信号SCK。
- 数据传输：在每个时钟周期内，主设备通过MOSI线发送一位数据，从设备通过MISO线发送一位数据，实现全双工通信。
- 停止条件：通信结束时，主设备拉高CS信号线，结束与从设备的通信。



SPI通信模式

- SPI 通信有四种时钟模式，由时钟极性（CPOL）和时钟相位（CPHA）的组合决定
 - CPHA (Clock Phase) : 0: 时钟信号的第一个跳变沿, 1: 第二个跳变沿
 - CPOL (Clock Polarity) : 0: 空闲态低电平; 1: 空闲态高电平

SPI Mode	CPOL	CPHA	数据采样边沿
Mode 0	0	0	上升沿采样
Mode 1	0	1	下降沿采样
Mode 2	1	0	下降沿采样
Mode 3	1	1	上升沿采样



■ SPI应用

- 存储器芯片：SPI广泛应用于与存储器芯片（如EEPROM、SRAM）和闪存芯片（如SD卡、SPI Flash）进行通信，实现数据的高速读写和存储。
- 传感器：许多传感器（如温度传感器、加速度传感器）使用SPI接口与微控制器通信，传输传感器数据。
- 显示器驱动器：SPI用于驱动液晶显示器和OLED显示器，传输图像数据和控制信号。
- 无线模块：SPI接口也用于无线通信模块（如Wi-Fi、蓝牙模块）与微控制器之间的数据传输



访问外设



- 读写外设端口
- 示例1：led、按键
- 示例2：键盘与终端
- 示例3：计数器和运行时间
- 示例4：图形显示
- 嵌入式系统常用外设及接口