同济大学

TONGJI UNIVERSITY

《计算机系统实验》

实验报告

实验名称　操作系统应用开发——人机井字棋

实验成员

日　　期　二零二三年 六 月 二十九 日

目录

1、实验目的

本实验将在第二次实验（μC/OS-II 操作系统移植）的基础上进行应用程序的开发，通过对μC/OS-II 操作系统进行改写和编译，将 C 语言应用小程序以图形化界面进行展示。本实验要求能够实现图形化界面展示，并且要求能够通过开发板进行用户输入，与 C 程序进行交互。

2、实验内容

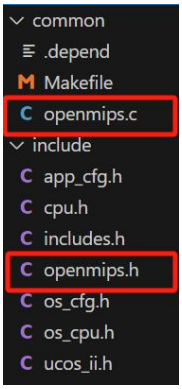
本实验在《自己动手写 CPU》15 章提供的内核程序源码基础上进行应用程序开发，开发过程涉及到对移植操作系统过程的加深理解、开发板串口控制、C 语言游戏逻辑过程实现、图形化界面展示等，大致流程如下：

1. C 语言人机井字棋代码实现
   (1) 应用程序创建与执行流程分析
   (2) 16bit 输入函数分析，字符输出、字符串输出函数实现
   (3) 函数主体逻辑实现
   (4) 主函数循环逻辑衔接
2. 操作系统编译
   (1) 编译过程分析
   (2) 编译 BUG 解决
3. 下板验证
   (1) Vivado 环境配置
   (2) Vivado Synthesis, Implementation, Bitstream generation
   (3) 配置 Memory Device Configuration

3、实验步骤

3.1 应用文件创建

在本实验中创建用户应用程序与开发 C 项目相似，首先在 include 文件夹下建立 openmips.h 头文件，在 common 文件夹下建立 openmips.c 进行具体实现。



同时在 common 文件夹下还需要建立 Makefile 文件指导编译，Makefile 将编译生成的目标代码文件 common.o 与 openmips.o 完成链接，具体内容为：

```
# CFLAGS += -DET_DEBUG -DDEBUG
LIB       = common.o
OBJS      = openmips.o
all:      $(LIB)
$(LIB):   $(OBJS) $(SOBJS)
          $(LD) -r -o $@ $(OBJS) $(SOBJS)
.depend: Makefile $(OBJS:.o=.c)
                  $(CC) -M $(CFLAGS) $(OBJS:.o=.c) > $@
sinclude .depend
```

在操作系统 Makefile 文件中将 common.o (应用程序目标代码), ucos.o (操作系统目标代码)与 port.o (操作系统接口目标代码)进行链接。注意这里链接的顺序有所要求，要求 ucos 操作系统编译在 port 编译之前，因为 port 接口层依赖于 ucos 层的部分函数变量。具体链接部分为：

```
# order is important here:
SUBDIRS   = common ucos port

LIBS  =   common/common.o ucos/ucos.o port/port.o

all: ucosii.om ucosii.bin ucosii.asm OS.bin

ucosii.om: depend subdirs $(LIBS) Makefile
          $(CC) -Tram.ld -o $@ $(LIBS) -nostdlib $(LDFLAGS)

ucosii.bin: ucosii.om
          mips-sde-elf-objcopy -O binary $< $@

OS.bin: ucosii.bin
          ./BinMerge.exe -f $< -o $@

ucosii.asm: ucosii.om
          mips-sde-elf-objdump -D $< > $@
```

3.2 应用文件执行流程

在文件结构构建完成后，我们在 openmips.h 与 openmips.c 中进行应用程序的编写。其中 openmips.c 的主函数中展示了应用程序执行流程。完整的应用执行流程主要分为四个步骤：

1. **OSInit**()：初始化操作系统内核
2. **uart_init**()：初始化 UART 串口设置
3. **gpio_init**()：初始化输入输出端口
4. **OSTaskCreate**()：在内核中创建分配一个用户任务，其中第一个参数为函数指针，指向本文件中实现的游戏逻辑函数 TaskStart，并为用户任务分配栈空间
5. **OSStart**()：运行内核，内核内部进行任务调度等

```
void main()
{
  OSInit();

  uart_init();

  gpio_init();

  OSTaskCreate(TaskStart,
               (void *)0,
               &TaskStartStk[TASK_STK_SIZE - 1],
               0);
```

```
  OSStart();

}
```

需要编写的游戏逻辑集中在 `TaskStart` 所指向的函数中，操作系统会进行进行调度，由于系统中除了 kernel 运行外仅有一个用户进程（即注册的 TaskStart）在运行，该游戏完成一轮后将不断循环。

3.3 工具函数分析

在实现游戏输入输出时所用到的工具函数主要包括捕获输入函数、输出字符函数和输出字符串函数

a) `gpio_in()`：捕获输入函数

该函数用于捕获用户输入的内容，获取内容的格式为 32bit 整形，获取内容的方式是以开发板中寄存器作为中转站。用户在开发板中的输入存储在输入寄存器中，用户任务函数从该寄存器中取出存储内容即可完成信息输入。

在 openmips.h 的实现中，我们首先计算输入存储器的位置，其地址为 `GPIO_BASE + GPIO_IN_REG`，然后从其中取出存储内容即可，其中 32 位中低 16 位对应板子的输入为：

输入实现为：

```
#define REG32(add) *((volatile INT32U *)(add))
#define GPIO_BASE 0x20000000
#define GPIO_IN_REG  0x00000000

INT32U gpio_in()
{
        INT32U temp = 0;
        temp = REG32(GPIO_BASE + GPIO_IN_REG);
        return temp;
}
```

b) ：字符输出函数与字符串输出函数

该函数用于图像界面展示输出，主要实现分为两部分：单个字符输出与字符串输出

单个字符输出函数为：

```
#define WAIT_FOR_THRE \
       do { \
              lsr = REG8(UART_BASE + UART_LS_REG); \
       } while ((lsr & UART_LS_THRE) != UART_LS_THRE)

#define WAIT_FOR_XMITR \
       do { \
              lsr = REG8(UART_BASE + UART_LS_REG); \
       } while ((lsr & BOTH_EMPTY) != BOTH_EMPTY)
```


装 订 线

```
void uart_putc(char c)
{
    unsigned char lsr;
    WAIT_FOR_THRE;
    REG8(UART_BASE + UART_TH_REG) = c;
    if(c == '\n') {
        WAIT_FOR_THRE;
        REG8(UART_BASE + UART_TH_REG) = '\r';
    }
    WAIT_FOR_XMITR;
}
```

在这个函数中首先需要注意 WAIT_FOR_THRE 这里用于 UART 的同步控制，这个宏用于判断 UART 串口是否准备好接收新字符。若准备好(THRE 标志位)则在 UART_BASE + UART_TH_REG 计算寄存器位置写入一个字符。若遇到 '\n' 换行符，我们同时需要在结尾添加 '\r' 确保输出位置能够移动到下一行，即'\n\r'。最后 WAIT_FOR_XMITR 宏同样为 UART 串口的同步控制，其检查 TEMT 和 THRE 标志位，保证在下一步处理之前字符已经被完整传输。

在字符串输出函数中需要注意的是我们希望字符串在输出时不受其他的中断影响，左移在这里的处理是将输出部分放在一个临界区中，也就是在函数 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 之间。这两个函数分别为关中断与开中断。由此保证了字符串在输出时不会被打断。

```
void uart_print_str(char* str)
{
    INT32U i = 0;
    OS_CPU_SR cpu_sr;
    OS_ENTER_CRITICAL();

    while(str[i] != 0)
    {
        i++;
        uart_putc(str[i - 1]);
    }

    OS_EXIT_CRITICAL();
}
```

3.4 游戏函数输入输出控制

游戏函数的输入流程为：用户在板子下方的十六个拨键完成输入，点击确认键后将输入存储到输入 REG 中。

输入确认有两种方法：

1. 在每次游戏循环尾部添加延时，保证按键按下的时候不会有多个连续的高电平输入，主要通过 OSTimeDly() 实现以 10ms 为单位的延时。
2. 另一种确认方式通过反复拨键来进行输入确认，若当前确认键电平与前一帧相同，则说明没有进行确认，否则当作一次有效输入。

本实验采用 V10 为确认键用第二种方法实现确认键。

```
void TaskStart(void *pdata) {
    INT32U gamer = 0;
    for (;;) {
        ...
        INT32U gamer_now = (choice >> 14) & 1;

        if (gamer != gamer_now) {
            gamer = gamer_now;
            ...
        }
    }
}
```

3.5 井字棋数据结构与输入输出

棋盘存储结构：井字棋由一维数组 `INT32U board[9]` 进行存储，其中 0 代表该位置为空，1 代表该位置为电脑棋子，2 代表该位置为人类棋子。

检查赢家：由于井字棋为 9 * 9 格子，赢棋策略相对较少（横向、纵向、对角线），我们可以用一个二维数组将赢棋情况进行静态存储之后再逐个进行判断：

```
INT32U checkWinner(INT32U* board) {
    static const INT32U win_patterns[8][3] = {
        {0, 1, 2}, {3, 4, 5}, {6, 7, 8}, // Rows
        {0, 3, 6}, {1, 4, 7}, {2, 5, 8}, // Columns
        {0, 4, 8}, {2, 4, 6}             // Diagonals
    };
    int i;
    for (i = 0; i < 8; i++) {
        if (board[win_patterns[i][0]] != 0 &&
            board[win_patterns[i][0]] == board[win_patterns[i][1]] &&
            board[win_patterns[i][1]] == board[win_patterns[i][2]]) {
            return board[win_patterns[i][0]];
        }
    }
    return 0;
}
```

棋盘打印：调用串口字符串输出函数进行输出，注意行间及边界边框打印顺序

```
void printBoard(INT32U* board) {
    int i;
    for (i = 0; i < 9; i++) {
        uart_print_str("| ");
        printUnit(board[i]);
        uart_print_str(" ");
        uart_print_str((i + 1) % 3 == 0 ? "|" : "");
        uart_print_str((i + 1) % 3 == 0 ? "\n" : " ");
    }
    uart_print_str("\n");
}
```

3.6 井字棋主逻辑实现

在主流程接收到输入后，首先判断该位置是否有棋子，若有则直接打印 ”Invalid Move”，若无棋子则完成对应位置的人类输入。紧接着进行电脑下棋，此时按照电脑寻路算法进行判断下棋

位置，最后进行胜负检查。无论是否下棋成功，都需要向用户即时进行输出展示棋盘帮助用户进行下一步判断。

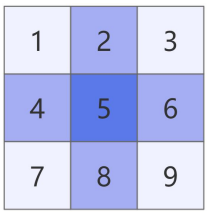
```
void TaskStart(void *pdata) {
    INT32U board[9] = {0};
    INT32U computerPre[9] = {4, 1, 3, 5, 7, 0, 2, 6, 8};
    INT32U gamer = 0;
    INT32U wer = gpio_in() >> 1;
    uart_print_str("This is cross game. Enjoy yourself!:\n");

    for (;;) {
        INT32U data = gpio_in();
        INT32U choice = data >> 1;
        INT32U gamer_now = (choice >> 14) & 1;

        if (gamer != gamer_now) {
          isValid = 1;
            gamer = gamer_now;
            uart_print_str("\nYour turn: \n");

            int index = -1;
            int i;
            for (i = 0; i < 9; i++) {
                if ((choice >> i) & 1) {
                    index = i;
                    break;
                }
            }

            if (index != -1 && board[index] == 0) {
                board[index] = 2;
                printBoard(board);
                if (checkWinner(board) == 2) {
                    uart_print_str("You win!\n");
                    break;
                }
                if (isBoardFull(board)) {
                    uart_print_str("A tie!\n");
                    break;
                }
                uart_print_str("Computer turn: \n");

                int move = expectMaxMove(board, 1);
                if (move == -1) {
                    computerMove(board, computerPre);
                } else {
                    board[move] = 1;
                }

                printBoard(board);
                if (checkWinner(board) == 1) {
                    uart_print_str("Computer wins!\n");
                    break;
                }
                if (isBoardFull(board)) {
                    uart_pr_str("A tie!\n");
                    break;
                }
            }
            else {
                uart_print_str("Invalid move, try again.\n");
                printBoard(board);
```

```
                isValid = 0;
            }
        }
    }
}
```

3.7 电脑寻路方法

本游戏实现了两种电脑寻路算法，分别是按照固定步骤的算法和 ExpectMax 算法

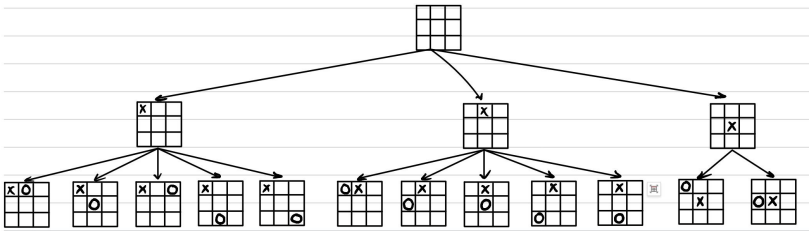
固定步骤算法：在本算法中，我们固定计算机下棋的位置顺序，也就是说计算机将会按照固定的位置序列下棋。由于中间以及边棱优势更大，最终确定顺序为下图所示，其中颜色深的部分为下棋概率更高的位置。



具体实现为：

```
void computerMove(INT32U *board, INT32U *computerPre) {
    int i;
    for (i = 0; i < 9; i++) {
        if (board[computerPre[i]] == 0) {
            board[computerPre[i]] = 1;
            break;
        }
    }
}
```

ExpectMax 算法：算法在井字棋中递归地评估所有可能的走法和结果，假设对手会做出最优回应，尝试最大化计算机的得分。具体实现中，算法在每个空位置模拟落子，计算当前玩家的得分，并递归计算对手的最优响应得分，通过反复比较选择得分最高的走法。即使对手总是做出最优选择，这种方法也能够确保计算机能够选择一条最有利的路径。



在这个决策树中，我们首先计算两步以后的内容，然后根据所有子情况计算局面分数，为第一层的局面进行赋值，从而选择最优策略。

具体实现为：

```
int expectMaxMove(INT32U *board, int player) {
    int bestMove = -1;
    int bestValue = -2;

    for (int i = 0; i < 9; i++) {
        if (board[i] == 0) {
```

```
            board[i] = player;
            int moveValue = -expectMax(board, 3 - player);
            board[i] = 0;

            if (moveValue > bestValue) {
                bestMove = i;
                bestValue = moveValue;
            }
        }
    }
    return bestMove;
}
int expectMax(INT32U *board, int player) {
    int winner = checkWinner(board);
    if (winner != 0 || isBoardFull(board)) {
        return evaluateBoard(board, player);
    }

    int bestValue = -2;

    for (int i = 0; i < 9; i++) {
        if (board[i] == 0) {
            board[i] = player;
            int moveValue = -expectMax(board, 3 - player);
            board[i] = 0;

            if (moveValue > bestValue) {
                bestValue = moveValue;
            }
        }
    }
    return bestValue;
}
```

3.8 局面衔接问题

在以上的实现方式中，我用一个拨键实现了确认信号，但是当一局游戏结束之后，TaskStart函数中记录的上一次电平会重新初始化为低电平，此时若确认拨键为高电平，上一句最后一个输入将会自动输入进新一局游戏中，由此产生局面关联。

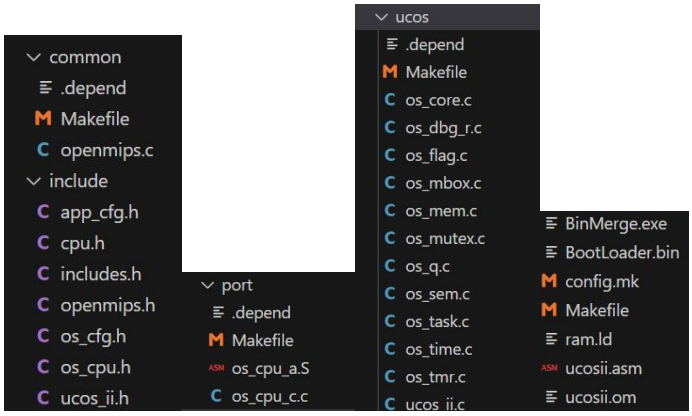
为解决关联，可以引入全局变量对局与局之间进行控制。由于在注册为一个有效进程后，全局变量存储在堆区，在各进程之间保持静态，可以作为标志位进行控制，另一个全局变量控制提示输出。具体实现为：

```
INT32U isFirst = 1;
INT32U isValid = 1;

void TaskStart(void *pdata) {

        if ((wer >>14) & 1) {
              if (isFirst == 1) {
                    uart_print_str("Please make sure V10 port is 0...\n");
                    isFirst = 0;
                    }
                    else {
                              isFirst = 0;
                    }
                    return;
              }
              else {
```

```
                isFirst = 1;
            }

            if (isValid == 0) {
                return;
            }
           ...
    }
}
```

3.9 编译流程分析

编译主要涉及到三个模块和一个操作系统引导模块 Bootloader，在每一个小模块的 Makefile 文件中存有编译指令，在.depend 文件夹下存有编译过程所需要依赖库



具体编译过程为：

1. 在主文件夹下新建链接脚本 ram.Id

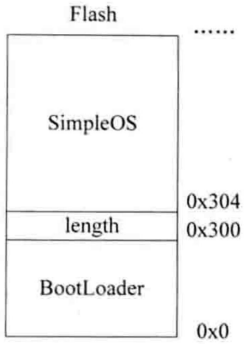
ram.Id 链接脚本定义了程序在内存中的布局。MEMORY 部分定义了两个内存区域：vectors 从地址 0x00000000 开始，长度为 0x80；ram 从 0x80 开始，长度为 0x200000 减去 0x80。SECTIONS 部分将各个段放置在这些内存区域中：.vectors 段放在 vectors 区域，.text、.rodata、.sbss、.scommon、.data、.bss 和 .stack 段都放在 ram 区域。

2. 主文件夹新建 config.mk, Makefile 文件

新建 config.mk 时需要注意 CFLAG 定义中的 ‘-mips32’选项，要求编译器按照 MIPS32 编译源程序。

```
CFLAGS += -I$(TOPDIR)/include -I$(TOPDIR)/ucos -I$(TOPDIR)/common -Wall -Wstrict-prototypes
-Werror-implicit-function-declaration -fomit-frame-pointer -fno-strength-reduce -O2 -g
-pipe -fno-builtin -nostdlib -mips32
```

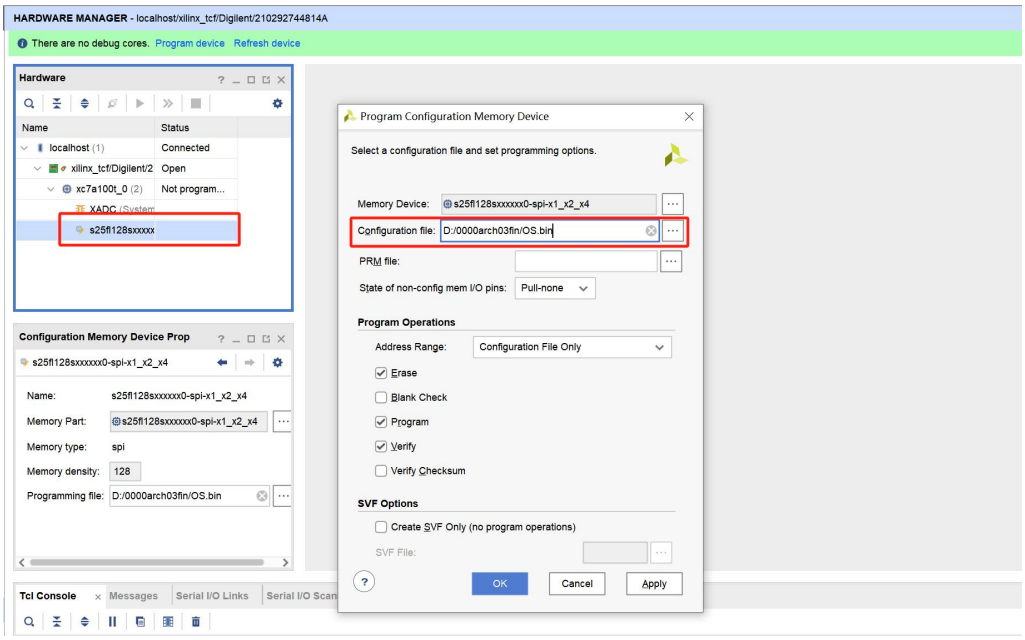
Makefile 文件指导编译器将各个模块的.o 文件链接，其中使用 binMerge.exe 将两个模块的.bin 文件 merge 到一起，其中 binMerge 的过程需要计算 merge 的地址。其中 Bootloader 存放在 Flash 从 0x0 开始的空间，在 0x300 后紧接着存放的是一个整形 4 字节是操作系统引导程序的长度信息。在 OpenMIPS 启动之后首先执行 Bootloader 兄 0x304 处读取 length 长度信息，将其复制到 SDRAM 中，最后跳转到 SDRAM 0x0 地址，将控制权交给操作系统。



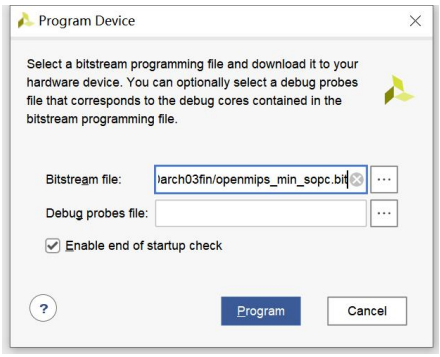
主文件夹下运行 make all 执行 Makefile，生成聚合后的 OS.bin 文件

3.10 Vivado 配置与下板验证

这里使用 Vivado2019.1 进行下板实验，首先配置 Memory Device Configuration



再进行 bitstream 下板



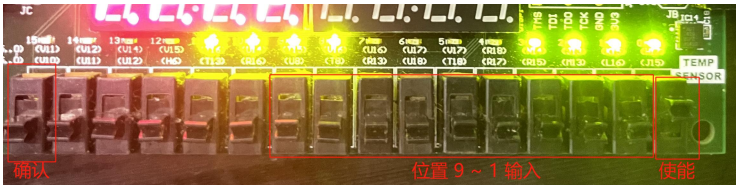
打开串口调试程序，将波特率调节到 9600，观察下板结果即可

4、实验结果

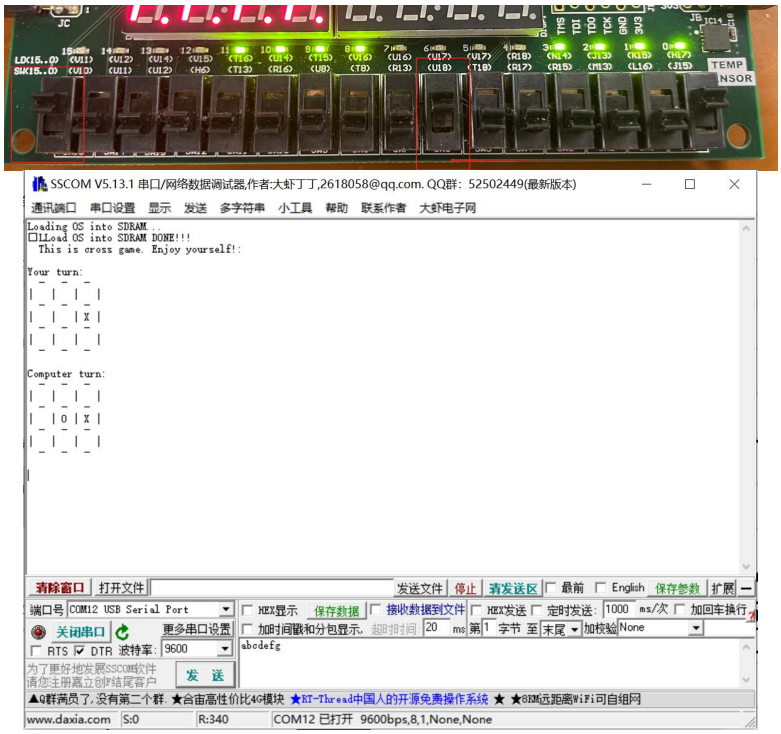
4.1 系统初始化

将 J15 使能开关打开，首先操作系统执行 Bootloader，显示已经将 OS 移动到 SDRAM 中，操作系统从 SDRAM 中取出执行，提示信息显示到输出窗口：

其中按键功能如下：



其中最左侧为确认键，位置 9~1 可以进行输入，比如此时我们需要在(1, 2)的位置下一个 X，这时需要使第六个键为高电平，其余为低电平，拨动确认键即可显示：



4.2 人类赢棋

这里我们首先测试人类赢的情况：

这里产生了局间衔接问题，正确提示 V10 应该为 0，将 V10 恢复为低电平之后进行电脑赢的测试

4.3 电脑赢棋

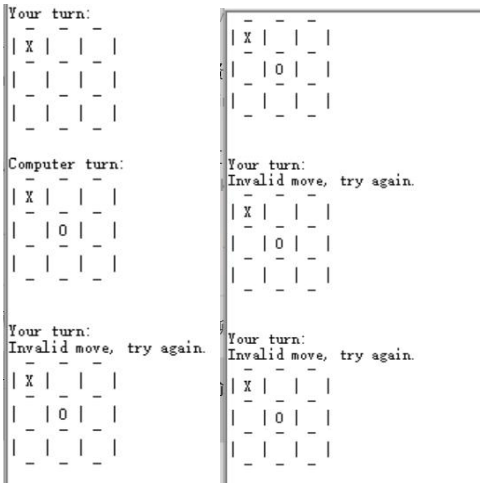
4.3 平局测试

进行平局测试：

4.4 下棋位置检测

当下棋位置有人类棋子或者电脑棋子时都是 invalid move:

测试两次下到同一个位置或者下到电脑位置：



正确显示 invalid move，测试正确。

5、实验总结

在本次实验中我在第二次实验移植操作系统实验的基础上进行了应用程序的开发，最后开发出了基于串口字符显示的人机对战井字棋游戏。

在开发过程中首先遇到的主要问题是对于输入输出方式不熟悉。通过《自己动手写 CPU》书中例程确定 gpio_in 作为二进制输入，通过 uart_putc 进行输出，并封装了 uart_print_str 进行字符串的输出，同时了解了 REG 中转存取输入输出的方式。字符串输出过程中考虑到了字符串输出的连续性，将输出过程放在临界区中避免其他中断打扰。

接着是对于游戏确认键的设计，原本的按键设计在 N17 按钮，但是由于按键过于灵敏，按下时直接输出了长时间的高电平，所以在这里需要添加延时（书中做法）或者直接使用一个拨键作为确认触发，通过与上一个状态进行比较判断是否触发确认，最后选用了后者作为确认方式。

在进行游戏设计时遇到的最大问题是局与局之间的衔接。由于确认按键的设计，在每次进程结束之后，记录上一个状态的变量总会初始化为一个定值，所以在这里我引入了全局变量对确认按键检查进行控制，若开局之前就是高电平确认状态，则让其等待确认按键恢复至正确初始值再开始本局游戏。

在游戏逻辑上我考虑到了多种局面情况，并针对搜索算法进行了设计，在测试时考虑了多种错误输入，修复 BUG 提升了系统鲁棒性。

最后生成 OS.bin 的过程与前一次实验相似，在答辩环节中老师提及关于 OS.bin 两部分：操作系统 ucosii.bin 与 Bootloader.bin 在 OS.bin 中的地址问题，在仔细阅读教程后我在这次的报告中 *3.9 编译流程分析* 中进行了详细解释，同时我还补充了对于 Makefile 部分参数的分析，巩固了我对于操作系统初始化流程的理解。

本次实验让我了解到通过串口程序进行图形化界面的小程序设计并不简单，需要对输入输出、编译下板过程具有充分理解后进行实验。在实验过程中进行 Debug 需要完成多次时间较长的修改编译下板测试过程，锻炼了我们的 debug 与测试能力。

在实现简单的系统应用过程中，我进一步加深了对于操作系统初始化流程与编译流程的理解，同时帮助我巩固了计算机系统结构、操作系统与计算机组成原理的知识，收获颇丰。