Braço de Robot em configuração SCARA – versão damas

Diogo CORREIA[1], André MADUREIRA[2]

[1] FEUP/DEEC, Porto, Portugal, up201705099@fe.up.pt
[2] FEUP/DEEC, Porto, Portugal, up201706076@fe.up.pt

Resumo: Ao longo deste século a indústria da robótica apresentou um crescimento enorme, desde a criação do primeiro robot industrial conhecido, em 1937, que se assemelhava a um guindaste com apenas um motor elétrico, passando por uma grande evolução na época da segunda guerra mundial onde existiu uma grande necessidade de evolução tecnológica, tendo o seu auge em 1984. Um ano de elevada importância para o trabalho realizado neste projeto foi 1981, quando foi inventado o primeiro "Selective Compliance Assembly Robot Arm" (também conhecido como SCARA), sendo este o braço robótico utilizado para o desenvolvimento do jogo de damas. Uma possível implementação de tal será apresentada ao longo deste relatório, começando com uma breve explicação das instruções/regras do jogo e, de seguida, explicações mais aprofundadas acerca da implementação do código das damas e respetivo controlo do braço.

1.Introdução

Para este projeto foram utilizados três servos, dois médios (MG996R) e um pequeno (SG-90), um Arduíno Uno, variadas peças constituintes de um braço robótico SCARA, um eletroíman, peças magnéticas (damas) e um tabuleiro, cinco botões e variadas resistências. A funcionalidade de todos estes constituintes será descrita em algum momento ao longo deste relatório.

O objetivo deste trabalho seria integrar os estudantes numa utilização completa e no controlo tanto do Arduíno como dos servos, levando os servos a realizarem em conjunto uma determinada tarefa (a movimentação do braço de forma a movimentar peças de um lado para o outro de acordo com as regras do jogo).

Para terminar, apresenta-se de seguida as instruções/regras para uma melhor compreensão das implementações feitas durante o trabalho:

- Conteúdo: 24 peças e um tabuleiro;
- Objetivo: Remover todas as peças do adversário do tabuleiro, capturando-as, ou impedir que o adversário possa realizar jogadas;
- Configuração do jogo: juntar 12 peças da mesma cor e colocá-las nas três primeiras filas à frente, apenas nas casas pretas. O adversário realiza o mesmo, com as peças da outra cor e do lado oposto do tabuleiro;
- Como jogar: Escolher um jogador para jogar primeiro (este jogador fica conhecido como jogador 1 e o outro como jogador 2). No seu turno, os jogadores devem selecionar um dos movimentos que sigam as regras apresentadas abaixo. Após o movimento de uma dama o turno desse jogador termina e o do outro começa. Os turnos alteram-se consecutivamente até que o jogo termine.
- Regras dos movimentos:
    1. Mover as damas sempre para a frente, no sentido diagonal, em direção ao oponente.

       Nota: Após a peça se tornar um rei, tanto se pode movimentar diagonalmente para a frente ou para trás.
    2. As damas podem-se movimentar um espaço na diagonal para a frente, desde que esta não esteja ocupada;
       Podem movimentar-se duas casas na diagonal para a frente desde que o espaço esteja vazio e o espaço sobre o qual saltou estivesse preenchido por uma dama (ou rei) do adversário. Quando isto acontecer, a dama do adversário é capturada.
    3. Se todos os espaços à volta de uma dama estiverem ocupados, a dama não se pode movimentar.

- Captura de uma peça do adversário: Se uma dama saltar por cima de uma dama do adversário essa dama é considerada capturada;
- Tornar-se um Rei: Quando uma das damas atinge a primeira fila do adversário torna-se um rei. Neste momento, coloca-se outra dama da mesma cor por cima dessa para diferenciar esta nova peça das damas. Esta peça poder-se-á mover na diagonal para a frente e para trás, mas, tal como as damas, apenas um espaço caso não esteja a capturar nenhuma peça.

2.Procedimento

Ao longo deste capítulo será apresentada a solução para o objetivo proposto, começando por uma explicação da implementação do jogo em si, evoluindo, depois, para o controlo dos servos e do eletroíman, ou seja, para o controlo do braço. Visto que as portas utilizadas para a comunicação Arduíno/Computador são algo de elevada importância, são apresentadas, desde já, a inicialização dessas mesmas (Figura 1. e Figura 2.).

```
void io_init(void){
    DDRB |= (1 << PB1) | (1 << PB2); //coloca o pinb 9 e 10 como saidas (PWM)
    // PIN89 servo Base
    // PIN810 servo topo
    DDRD |= (1 << PD5); //  Servo ponta

    DDRD |= (1 << PD2); //  Eletroiman

    DDRB |= (1 << PB0); //  LED Player 1

    DDRD |= (1 << PD7); //  LED Player 2

}
```

Figura 2. - Inicializações

```
void buzzer_init(void){
        DDRB |= (1 << PB3);
}
```

Figura 1. – Inicialização buzzer

2.1 Código relativo ao jogo das damas

Inicialmente foi criada uma struct (boxes [8][8]), em que cada posição corresponderia a uma determinada posição no tabuleiro. Na implementação realizada, o valor -1 corresponde às quadrículas brancas do tabuleiro, 0 a quadrículas pretas não ocupadas e os valores 1 e 2 correspondem às peças do jogador 1 e 2, respetivamente. Ao realizar esta implementação tornou-se mais fácil fazer o ajuste do tabuleiro de acordo com as jogadas realizadas. Além disso, contribuiu bastante para fazer o desenvolvimento das jogadas pois era sempre possível saber qual a peça que estava numa qualquer posição no momento das mesmas.

A partir deste struct foram surgindo todas as outras funções necessárias ao bom funcionamento do jogo, apesar de nem todas dependerem diretamente dela. Foram criadas funções para retirar as peças capturadas, para verificar se existem reis ou não, contar as peças dos jogadores, mover para cima, para baixo, verificar se alguém ganhou, para os movimentos do jogador e para alterar o turno. De forma a não se tornar um relatório muito extenso, apenas serão explicadas as funções mais importantes/complexas.

2.1.1 *PlayerTurn*

Começando pela função que define os turnos dos jogadores (*PlayerTurn*), o que se pretende fazer aqui é entrar num "ciclo infinito" de alteração de turnos, até que alguém ganhe, ou seja, alguém capture as peças todas do adversário ou o impeça de realizar mais jogadas. Dentro do turno, enquanto a jogada for válida, são verificados/ vários pontos: se o jogador ainda tem jogadas (o que, se não se verificar, atribui imediatamente a vitória ao adversário); se existirem jogadas possíveis, qual a que o jogador pretende fazer (através da função *play*, que será explicada mais à frente); se a jogada selecionada for válida, e após fazer a conversão de inteiro para caracter da linha de onde quer jogar e da linha para a qual quer jogar, é chamada a função *playermoves*, cujo objetivo é verificar se a jogada escolhida é válida e, se realmente for, atualizar valores na struct boxes[8][8], atualizando também o tabuleiro mostrado, fazer o braço SCARA realizar o movimento dessa peça e, para o caso de nessa jogada se ter capturado alguma peça, fazê-lo remover a peça do tabuleiro para uma posição pré-definida, denominada lixo que, dependendo do jogador, será LIXOP1 ou LIXOP2 (X e Y).

Seguem-se dois excertos do código desta função (Figura 3. e Figura 4.) sendo que, entre estes, existe a remoção de peças que tenham sido capturadas e ainda a verificação se o jogador atual ainda tem jogadas possíveis.

```
void playerTurn(uint8_t player){

    uint8_t valid=0;
    uint8_t row = 0, col = 0, newRow = 0, newCol = 0;
    char oldRowChar,newRowChar;
    uint8_t moveSelected;

    if(player == 1){
        PORTB |= (1 << PB0);
        PORTD &= ~(1 << PD7);
    }
    else{
```

Figura 3. – Ativação do led referente ao jogador

```
_delay_ms(3000);
printf("\nplayer %d selecione a sua jogada\n", player);

moveSelected = play(&row, &col, &newRow, &newCol);

if(moveSelected){
    oldRowChar = convNum2Char(row);
    newRowChar = convNum2Char(newRow);

    playermoves(player,row,col,newRow,newCol,oldRowChar,newRowChar,&valid);
    // move_servo_AtoB(row,col,newRow,newCol);
    row = 0;
    col = 0;
    newRow = 0;
    newCol = 0;
}
```

Figura 4. – Seleção de jogada

2.1.2 *playermoves*

Esta é dividida em dois casos: dama e rei. Tanto num caso como no outro é verificado se a posição inicial contém uma peça do jogador atual e se existe alguma peça na quadrícula para a qual se quer jogar. A partir daqui é que diversifica, visto que a dama só se pode deslocar em frente, ao contrário do rei. Se a peça que estiver na posição inicial for uma dama, o movimento seguinte depende do jogador. Se for o jogador 1, a função chamada é a *moveDown*. Por outro lado, se for o turno do jogador 2 e a peça for uma dama é chamada a função *moveUp*. Estas duas funções são praticamente iguais, diferenciando apenas no movimento descrito pela peça, a primeira de cima para baixo e a outra o contrário. Ambas serão descritas mais à frente.

Se a jogada for válida o programa apura se o movimento da dama a fez tornar-se um rei e imprime o tabuleiro e, após isto, verifica se o próximo jogador tem jogadas disponíveis. Caso tenha, o turno é alterado.

Por outro lado, se a peça considerada for um rei, o código é dividido em 3 partes: movimento para baixo (nova linha maior do que a linha atual); movimento para cima (nova linha menor do que a linha atual) e movimento para cima e para baixo (linha mantém-se igual), ou seja, de acordo com o movimento que o jogador quer realizar com o rei. Aqui é realizado desta forma pois não faz sentido dividir as opções de movimento de acordo com o jogador, visto que se pode deslocar para onde quiser (desde que na diagonal e de acordo com as regras de deslocação). De qualquer modo, o raciocínio implementado aqui assemelha-se ao raciocínio descrito acima. Se a nova linha for maior do que a linha atual é chamada a função *moveDown,* se for menor do que a atual moveUp e, caso a quadrícula final se encontre na mesma linha, *backFor* (também será apresentada de seguida). Caso essas funções sejam válidas são verificadas as mesmas condições que acima, ou seja, se a dama se transformou num rei, imprimindo o tabuleiro imediatamente a seguir e, de seguida, verifica se o outro jogador tem jogadas disponíveis. Se, por alguma razão, a jogada não for válida é pedido ao mesmo jogador que introduza outra.

Para esta função serão apresentados mais dois excertos de código, um para quando a peça a jogar corresponde a uma dama e o outro para quando corresponde a um rei, o primeiro na Figura 5. e o seguinte na Figura 6.

```
uint8_t playermoves(uint8_t player, uint8_t row, uint8_t col, uint8_t newRow, uint8_t newCol,char oldRowChar, char newRowChar, uint8_t *valid){
    //uint8_t valid=0;                                  //flag value for move validity
    //ja esta preparada para os dois jogadores

    //Se o sitio para onde esta a mover for invalido entao pede para fazer movimento outra vez
        if((boxes[row][col] == -1)||(boxes[newRow][newCol] == -1))
            playerTurn(player);

        if((boxes[row][col] == 0)||(boxes[newRow][newCol] != 0))
            playerTurn(player);

        printf("\n player: %d moves\n",player);
            //for player simple piece
            if(boxes[row][col]== player && boxes[newRow][newCol]==0){
                    *valid = (player == 1? moveDown(boxes[row][col],row,col,newRow,newCol):
                                           moveUp(boxes[row][col],row,col,newRow,newCol));   //player call move down function
                    if(*valid==1){
                        checkKing();                        //checking for king
                        print_board(boxes);                 //refreshing the board
                        if(countPieces((player == 1 ? 2:1))==0){              //checking if player has won
                            win(player);
                        }
                        else{
                            moveStore(player,oldRowChar,col+1,newRowChar,newCol+1);
                            playerTurn((player == 1) ? 2:1);
                        }
                    }
                    else playerTurn(player);
            }
            //for player  king piece ( p1 = 3; p2 = 4)
```

Figura 5. – *playermoves*: validez e movimento (dama)

```
//for player  king piece ( p1 = 3; p2 = 4)
else if(boxes[row][col]==(player + 2)){
        //if moving down
        if(newRow>row){
                *valid = moveDown(boxes[row][col],row,col,newRow,newCol);
                if(*valid==1){
                    checkKing();                        //checking for king
                    print_board(boxes);                 //refreshing the board
                    if(countPieces((player == 1 ? 2:1))==0){
                        win(player);
                    }
                    else{
                        moveStore(player,oldRowChar,col+1,newRowChar,newCol+1);
                        playerTurn((player == 1) ? 2:1);
                    }
                }
                else playerTurn(player);
        }
        //if moving up
        else if(newRow<row){
                *valid = moveUp(boxes[row][col],row,col,newRow,newCol);
                if(*valid==1){
                    checkKing();                        //checking for king
                    print_board(boxes);                 //refreshing the board
                    if(countPieces((player == 1 ? 2:1))==0){
                        win(player);
                    }
                    else{
                        moveStore(player,oldRowChar,col+1,newRowChar,newCol+1);
                        playerTurn((player == 1) ? 2:1);
                    }
```

Figura 6. – *playermoves*: validez e movimento (rei)

2.1.3 *moveUp, moveDown e backFor*

Visto que estas funções são relativamente semelhantes, será feita uma explicação geral das três, sendo que alguns pontos serão esclarecidos mais especificamente para cada uma delas. De qualquer modo, elas apresentam uma grande divergência, que é o sentido de deslocamento. *moveUp* apenas permite deslocar a peça para cima, *moveDown* para baixo e backfor permite tanto para cima com para baixo, desde que termine na mesma linha. Exceto isto, a sua codificação é praticamente igual, como será possível observar de seguida. Inicialmente é verificado se a jogada é válida através de: verificação da presença de peça na posição escolhida; verificação de espaço livre na nova posição escolhida. De notar que, através dos if's utilizados em *playermoves* estas condições já seriam verificadas. No entanto, não só como uma forma de segurança, como também para facilitar a interpretação do código, estas condições são repetidas. A partir daqui são verificadas as várias possibilidades de jogada, sendo essas: o deslocamento, na diagonal, sem a captura de nenhuma peça; o movimento que leva à captura de uma peça ou ainda o movimento que leva à captura de duas peças (no caso de existir a captura de uma peça o programa verifica sempre se a peça capturada pertencia ao adversário). Ao longo das várias possibilidades, se alguma cumprir todos os requisitos necessários para que essa jogada se realize, é chamada a função *move_servo_AtoB.* Esta corresponde à função que interliga todo

o código do jogo de damas com o movimento do braço robótico. Apesar de ser uma das funções com implementação mais acessível é, também, uma das mais importantes. Desta forma, será explicada já de seguida.

Aqui apenas serão apresentados excertos de uma das funções visto que, no geral, todas elas se comportam da mesma forma e já foram explicadas acima. A Figura 7. corresponde á verificação da validez da jogada e as figuras 8., 9. e 10. correspondem a deslocações de uma casa, duas casas e quatro casas na diagonal, respetivamente. Mais uma vez é referido que apenas correspondem a excertos de código o que implica que nem todas as condições/opções de jogada estão demonstradas. Por exemplo, na Figura 7. apenas é visualizável o caso em que a posição final escolhida corresponde à linha antiga + 2 e coluna antiga -2, mas também existe o caso em que o movimento escolhido corresponde a linha antiga +2, coluna antiga +2.

```
uint8_t moveDown(uint8_t type,uint8_t row,uint8_t col,uint8_t newRow,uint8_t newCol){
    /*checking for a valid move*/
        //primeiras duas condicoes ja verificadas, considerar remove las
    //false move if new place is not empty
    if(boxes[newRow][newCol]!=0){
        printf("\n[moveDown] Espaco ja ocupado\n");
        return 0;
    }

    //false move if player piece is not placed at input position
    else if(boxes[row][col]==-1 || boxes[row][col]==0 ){
        printf("\n[moveDown] A sua peca nao esta nessa posicao\n");

        return 0;
    }
```

Figura 7. – Validez da jogada

```
//valid move if new place is in diagonal
else if ((newRow == row + 1) && (newCol == col - 1 || newCol == col + 1)){
    boxes[newRow][newCol] = boxes[row][col];
    boxes[row][col] = 0;
    move_servo_AtoB(row,col,newRow,newCol);
    return 1;
}
```

Figura 8. – Deslocamento na diagonal sem captura

```
//player captures enemies's checker
else if ((newRow == row + 2) && (newCol == col - 2 || newCol == col + 2)){
      if ((boxes[newRow - 1][newCol - 1] == type-1 || boxes[newRow - 1][newCol - 1] == type-3 || boxes[newRow - 1][newCol - 1] == type+1 |
          /*****************************/
          //comidelas em ação
          pecaRetirar[posPecaRetirar].player = ((type == 1 || type == 3) ? 1:2);

          pecaRetirar[posPecaRetirar].X = newRow - 1;
          pecaRetirar[posPecaRetirar++].Y = newCol-1;

          boxes[newRow - 1][newCol - 1] = 0;
          /*****************************/

          boxes[newRow][newCol] = boxes[row][col];
          boxes[row][col] = 0;
          move_servo_AtoB(row,col,newRow,newCol);
          return 1;
      }
```

Figura 9. – Captura de uma peça

```
//double checker by player
else if((newRow == row + 4) && (newCol == col - 4 || newCol == col + 4 || newCol==col)){
      if(newCol == col + 4){
            if((boxes[row+1][col+1]==type-1 || boxes[row+1][col+1]==type+1 ||  boxes[row+1][col+1]==type+3 || boxes[row+1][col+1]==type
                /*********************************/
                //comidelas em ação
                pecaRetirar[posPecaRetirar].player = ((type == 1 || type == 3) ? 2:1);

                pecaRetirar[posPecaRetirar].X = row + 1;
                pecaRetirar[posPecaRetirar++].Y = col + 1;

                pecaRetirar[posPecaRetirar].player = ((type == 1 || type == 3) ? 2:1);

                pecaRetirar[posPecaRetirar].X = row + 3;
                pecaRetirar[posPecaRetirar++].Y = col + 3;


                boxes[row+1][col+1]   = 0;
                boxes[row+3][col+3]   = 0;
                /*********************************/
                boxes[newRow][newCol] = boxes[row][col];
                boxes[row][col]       = 0;
                move_servo_AtoB(row,col,newRow,newCol);
                return 1;
            }
```

Figura 10. – Captura de duas peças

2.2 Controlo do braço robótico

O SCARA é constituído pelos dois servos médios, em que cada um deles funciona como se fossem articulações de um braço humano, um servo pequeno que funciona como uma mão mas que apenas se desloca na vertical e o eletroíman que se comporta como uma mão a agarrar alguma coisa que, neste caso, são apenas materiais magnéticos. Acrescentando a isto, ainda é constituído por certas peças que permitem anexar todos estes materiais descritos.

Falando do controlo do braço robótico, de uma forma remetente para o software, esse controlo cinge-se, essencialmente, numa função já referida anteriormente, *move_servo_AtoB.* Esta, por outro lado, depende de outras 4 que dependem da inicialização de alguns timers. Todos estes passos serão explicados durante este capítulo. Depois desta função estar implementada corretamente, o único problema consistia em chamá-la nos locais corretos do código, de forma a apenas realizar o movimento quando a jogada introduzida fosse válida e não só permitindo retirar as peças que foram capturadas do tabuleiro, bem como inserir peças aquando da formação de um rei. Segue-se então a apresentação da função descrita.

2.2.1 *move_servo_AtoB*

Para um bom entendimento do que esta função realiza será necessário explicar o funcionamento das outras funções das quais esta depende. São essas *move_servo_base*, *move_servo_topo*, *move_servo_ponta* e *íman*. As duas primeiras necessitam de um timer (foi utilizado o timer 1) com fast pwm, o que foi implementado na inicialização do respetivo temporizador. Além disso também foi optado por limpar o temporizador quando atinge um certo valor pré-definido (clear on compare - OCR1A ou OCR1B). Estes OCR's são calculados utilizando a fórmula (1) que os permite definir, de acordo com os limites do temporizador, através de um certo ângulo entre 0 e 180º.

$$\text{OCR1n} = \frac{200 * \hat{a}ngulo}{9} + 1000 \qquad (1)$$

Já a função *move_servo_ponta* depende do timer 0 (inicializado para funcionar como fast pwm e para ser limpo quando igual a outro valor, o OCR0B). Nesta, em vez da utilização da fórmula (1), apenas foram introduzidos dois valores diferentes para OCR0B, um quando tiver de apanhar uma peça, e outro para quando se estiver a mover.

Por fim, em *íman*, apenas é ativada e desativada a porta escolhida (PD2) dependendo se se quer ativar ou desativar o íman, respetivamente. Seguidamente, surgem algumas imagens representativas do afirmado acima, nas quais as duas primeiras correspondem à inicialização dos temporizadores (Figura 11. e Figura 12.) e as quatro seguintes às variadas funções (Figura 13. a Figura 16.).

```
void servo_tc0_PWM_init(void){
  TCCR0B = 0;   //desliga o timer

  TCCR0B = (1 << WGM02);

  TCCR0A = (1 << WGM01) | (1 << WGM00) | (1 << COM0A1) | (1 << COM0B1);
  //COM0A[1:0] = 1 0    controla OC0A
  //COM0B[1:0] = 1 0    controla OC0B
  //WGM0[2:0] = 1 1 1

  TCNT0 = 0;

  // OCR0A  =  125; // Valor do CNT
  OCR0A = 130;
  //Para fint = 500Hz, TP = 256 e CNT = 125

  //OCR0A = 9;
  //OCR0B = 9;

  TCCR0B |= (1<<CS02) | (0<<CS01) | (0<<CS00);

}
```

Figura 11. – Inicialização do temporizador 0

```
void servo_tc1_PWM_init(void) {

  //DDRB = 0b00000110; // coloca o pinb 9 e 10 como saidas

  TCCR1B = 0; // desliga o timer

  TCCR1B = (1<<WGM13) | (1<<WGM12);

  TCCR1A = (1<<COM1A1) | (1<<COM1B1) | (0<<COM1A0) | (0<<COM1B0) | (1<<WGM11);// modo fast PWM e clear on compare match

  TCNT1 = 0;  // Reset do timer

  ICR1 = 40000;  // valor do CNT calculado para freq de 50hz e TP = 8

  // OCR1A = 3000; //Servo começa no meio (posicao 0) 1.5 ms pulse [Servo base]

  // OCR1B = 3000; //Servo começa no meio  (posicao 0) 1.5ms pulse [Servo topo]

  //posicoes de calibracao
  OCR1A = 3711; // OCR calculado apartir de um angulo de 122 graus
  OCR1B = 3111; // OCR calculado apartir de um angulo de 95 graus

  TCCR1B |= (0 << CS12) | (1<<CS11) | (0<<CS10);

}
```

Figura 12. – Inicialização do temporizador 1

```
void move_servo_base(uint8_t angle){
  // angle deve estar entre 0  e 180
  cli();
  OCR1A = (200/9) * angle + 1000;
  sei();
}
```

Figura 13. – Função *move_servo_base*

```
void move_servo_topo(uint8_t angle){
  // angle deve estar entre 0  e 180
  cli();
  OCR1B = (200/9) * angle + 1000;
  sei();
}
```

Figura 14. - Função *move_servo_topo*

```
void move_servo_ponta(uint8_t baixo_cima){
  // Baixo = 1;
  // Cima = 0
  cli();
  // OCR0B = (baixo_cima != 0)? 124:22;//124:22;
  OCR0B = (baixo_cima != 0)? 123:22;

  sei();
}
```

Figura 15. - *Função move_servo_ponta*

```
void iman (uint8_t state) {
  if(state) PORTD |=(1<<PD2);
  else PORTD &= (~(1<<PD2));
}
```

Figura 16. - Função *iman*

Ainda para a implementação desta função, foi criada mais uma struct, desta vez denominada movimentos, com o objetivo de guardar as diversas posições do tabuleiro, ou seja, as posições das quadrículas pretas do tabuleiro. Devido a alguns problemas de montagem do braço robótico, como por exemplo: peso excessivo na parte da frente do braço; falta de um suporte forte que permitisse segurar e aguentar o peso do braço; alguma folga presente nos servos (que surge, possivelmente, devido ao esforço a que estes foram submetidos) e, para terminar, a altura lateral do tabuleiro juntamente com o tamanho necessário dos encaixes para as ligações do eletroíman (o que leva ao contacto entre eles, desviando o braço); os valores retirados para as diversas posições do tabuleiro nem sempre estão de acordo com o que realmente se verifica. Visto ser um problema de montagem que é muito difícil de contrariar (pois a posição do braço não é estável e acaba por ceder, alterando a mesma e, portanto, os erros associados à tentativa de compensação deste desvio aumentam) foi decidido guardar as posições já obtidas nessa struct, mesmo que com algum erro para certas posições. Estas posições são implementadas na struct e prontas a utilizar ao chamar a função *calculo_pos*. Após estarem todas as funções definidas e as posições introduzidas, *move_servo_AtoB* pode ser executado. Não é apresentado um excerto de código para a função *calculo_pos* visto ser, apenas, uma igualdade de valores para certas posições de uma struct.

Consiste, assim, em chamar a função *move_servo_base* e *move_servo_topo* para a posição inicial, posição essa definida na struct referida acima. De seguida é dado algum tempo de espera (através da introdução de um delay) para que o braço se possa mover para a tal posição. Depois, é chamada a função *move_servo_ponta* para baixar o braço, é ativado o íman e volta-se a chamar *move_servo_ponta* para o levantar. De seguida, são chamadas novamente as duas funções de movimento do servo base e topo, para a nova posição, baixa-se o braço, desliga-se o íman e volta-se a levantar o braço.

Há que realçar que, não só na movimentação do braço, mas também em muitas das outras ações desta função, tiveram de ser introduzidos alguns delays para que os vários movimentos possam ser executados quando o têm de ser e durante o tempo necessário para que realizem o seu objetivo.

À parte disto, que consiste no essencial do controlo do SCARA, são introduzidas duas condições para se poder ter um lixo para o jogador 1 e 2, ou seja, um local onde se guardar as peças capturadas ao jogador adversário. Essas condições são necessárias pois, quando foi criada a struct para as posições, não foram definidas posições para o lixo. Sendo assim, caso os valores introduzidos sejam superiores a 10, superiores às posições

existentes nos tabuleiros, o programa assume que correspondem a coordenadas em vez de corresponderem a posições no tabuleiro. Isto permite, após capturar peças, guardá-las todas no mesmo sítio e, caso surja um rei a meio do jogo, colocar uma peça do jogador por cima da outra peça.

Surgem duas imagens representativa de *move_servo_AtoB* para um melhor entendimento do explicado acima (Figura 17. e Figura 18.).

```
//move o braço para as coordenadas
uint8_t move_servo_AtoB (uint8_t Xinicial, uint8_t Yinicial, uint8_t Xfinal, uint8_t Yfinal){
  //Falta definir pos para colocar peças comidas
  if(Xinicial > 10 && Yinicial > 10){
    move_servo_base(Xinicial);
    move_servo_topo(Yinicial);
  }
  else{
    move_servo_base(servo.movimento_base[Xinicial/*-1*/][Yinicial/*-1*/]);
    move_servo_topo(servo.movimento_topo[Xinicial/*-1*/][Yinicial/*-1*/]);
  }
  // printf("1\n");
  _delay_ms(2000);
  move_servo_ponta(1);
  // printf("2\n");
  _delay_ms(500);
  iman(1);
  // printf("3\n");
  _delay_ms(500);
  move_servo_ponta(0);
  _delay_ms(500);
```

Figura 17. – Função *move_servo_AtoB*: 1ª parte

```
if(Xfinal > 10 && Yfinal > 10){
  move_servo_base(Xfinal);
  move_servo_topo(Yfinal);
}
else{
  move_servo_base(servo.movimento_base[Xfinal/*-1*/][Yfinal/*-1*/]);
  move_servo_topo(servo.movimento_topo[Xfinal/*-1*/][Yfinal/*-1*/]);
}

// printf("4\n");
_delay_ms(2000);
move_servo_ponta(1);
// printf("5\n");
_delay_ms(500);
iman(0);
// printf("6\n");
_delay_ms(500);
move_servo_ponta(0);
_delay_ms(1500);
// printf("fim");
// move_servo_base(posicao descanso);
// move_servo_topo(posicao descanso);
return 1;
}
```

Figura 18. - Função *move_servo_AtoB*: 2ª parte

2.3 Extras

Ao programa das damas e ao controlo do SCARA foram ainda adicionados alguns extras como por exemplo: a utilização de dois leds para representar qual o jogador que deve realizar movimentos naquele turno; um buzzer, que apresenta uma música diferente para a vitória de cada um dos jogadores; armazenamento na eeprom do número mínimo de jogadas utilizadas para ganhar um jogo (entre todos os jogos realizados) e, ainda, quais as jogadas realizadas nesse jogo; a seleção da jogada através dos botões e, para finalizar, um LCD para mostrar o tabuleiro.

2.3.1 LED correspondente a cada jogador

De forma a que a indicação do turno de cada jogador fosse mais intuitiva, decidiu-se implementar a funcionalidade de haver dois LEDs (de cor diferente) correspondente ao jogador que estivesse a jogar. Para além disso, esta correspondência LED-jogador também é usada quando o jogador em questão ganha o jogo, de modo que o LED fica aceso durante toda a melodia de vitória. Esta codificação está demonstrada na Figura 3. onde, dependendo do jogador, um determinado LED acende. Ainda, na imagem seguinte (Figura 19.), é demonstrada a configuração quando um jogador ganha.

```
void win(uint8_t result){

    uint8_t i,j;
    if (result == 1){
        printf("Winner: Player 1\n");
        PORTB |= (1 << PB0);
        PORTD &= ~(1 << PD7);
        GameOfThrones();
    }
    else if (result == 2){
        printf("Winner: Player 2\n");
        PORTD |= (1 << PD7);
        PORTB &= ~(1 << PB0);
        EyeOfTheTiger();
    }
```

Figura 19. – LED ativo quando alguém ganha

2.3.2 Buzzer

Uma vez que a uma vitória está sempre associada uma melodia, optou-se por tornar isto possível configurando um buzzer de forma a reproduzir um som diferente alusivo à vitória de cada jogador. Ora, para tal, configurou-se o timer 2 para funcionar no modo CTC usando, portanto, a porta correspondente a OC2A (PB3) para proceder à ativação/desativação do buzzer. Para que este produza um som numa determinada frequência, ele tem de estar ativo durante um determinado intervalo de tempo relacionado com a frequência do som pretendida, sendo que esta pode variar de 1hz a 5kHz ou mesmo até 100kHz para aplicações onde se usem os ultrassons (de notar que o ser humano apenas consegue distinguir som na gama dos 20Hz-20kHz). Para esse efeito, criou-se a função *t2_set_ctc_a* (Figura 20.) que permite converter a frequência do som pretendido para um valor de OCR2A, fazendo com que este valor seja igual à frequência do timer que estamos a utilizar. Neste caso escolheu-se um timer de 125kHz através de um pré escalar de 256. Para além de tudo isto foram ainda definidas diversas notas musicais com as frequências a si associadas de forma a não só ser possível reproduzir as melodias escolhidas, como também permitir ao utilizador criar a sua própria melodia no futuro.

```
void t2_set_ctc_a(uint16_t hz,uint32_t timer_freq){

    OCR2A = div_round(timer_freq, hz*2) - 1 ;
    TCCR2A = (1 << COM2A0) | (1 << WGM21);
}
```

Figura 20. – Função de conversão frequência-OCR2A

2.3.3 EEPROM

Dado se tratar de um jogo de tabuleiro onde se usa a estratégia, é sempre importante registarmos os movimentos que fazemos para, por exemplo, no fim do jogo, desconstruir jogadas e detetar possíveis falhas na estratégia usada. Como tal, a implementação de um sistema que permite armazenar essa informação em memória não volátil é importante e possível de fazer usando a memória EEPROM. Já que este programa permite a realização de mais do que um jogo também seria interessante manter o registo do jogo que acabou com menos jogadas realizadas e memorizá-las para possível estudo da estratégia (Figura 21.). Assim, no fim de cada jogo, é comparado o número de jogadas realizadas no jogo atual com aquele que acabou mais rápido e se o esse número for menor então a informação que estava presente na EEPROM é atualizada e as jogadas do jogo são guardadas em variáveis desse mesmo tipo.

```
if(num_play < minJogadas){
    for (i = 0; i < num_play; i++){
        eeprom_update_byte(&(linha_inicio[i]),convChar2Num(RankJogadas[i].Xinit));
        eeprom_update_byte(&(coluna_inicio[i]),RankJogadas[i].Yinit);
        eeprom_update_byte(&(linha_fim[i]),convChar2Num(RankJogadas[i].Xfinal));
        eeprom_update_byte(&(coluna_fim[i]),RankJogadas[i].Yfinal);
        RankJogadas[i].Xinit = 0;
        RankJogadas[i].Yinit = 0;
        RankJogadas[i].Xfinal = 0;
        RankJogadas[i].Yfinal = 0;
    }
    for (i = num_play; i < minJogadas; i++){
        eeprom_write_byte(&(linha_inicio[i]),0);
        eeprom_write_byte(&(coluna_inicio[i]),0);
        eeprom_write_byte(&(linha_fim[i]),0);
        eeprom_write_byte(&(coluna_fim[i]),0);
    }
    printf(" NOVO RECORDE JOGADAS = %d \n",num_play);
    minJogadas = num_play;
}
else{
    printf("Historico das jogadas do jogo mais rapido:\n");
    for ( i = 0; i < minJogadas; i++)
    {
        printf("%c%d -> %c%d\n",convNum2Char(eeprom_read_byte(&linha_inicio[i])),
                                eeprom_read_byte(&coluna_inicio[i]),
                                convNum2Char(eeprom_read_byte(&linha_fim[i])),
                                eeprom_read_byte(&coluna_fim[i]));
    }
}
printf("----------------------\n");
```

Figura 21. – EEPROM: escrita e acesso

2.3.4 Seleção da jogada

Para a realização do jogo é necessário haver uma plataforma onde o jogador possa comunicar com este, de forma a registar os movimentos que pretende fazer. Para tal, optou-se pela utilização de cinco botões onde quatro se destinam aos movimentos iniciais e finais da peça (os dois primeiros selecionam a posição inicial, os dois últimos a final) e o último, distinguido pela sua cor vermelha, que corresponde a um botão de seleção, através do qual o jogador pode indicar que terminou a escolha da sua jogada. Ora, o funcionamento dos mesmos consiste na aplicação de tensão a cada porta analógica a eles ligada (PORTC0 a PORTC4), que a vai converter para um valor entre 0 e 255 consoante o valor de tensão for 0 a 5 volts, usando um conversor ADC também já configurado para tal. A sua configuração surge abaixo (Figura 22.). Há ainda a salientar que o jogador, à medida que vai pressionando cada botão, vai recebendo informação do movimento que está prestes a fazer, isto é: sempre que pressiona e deixa de pressionar (*falling edge*) o primeiro ou o segundo botão, a linha e a coluna da peça que está a escolher vai aumentando, respetivamente, até voltar ao início ou seja, quando a linha/coluna ultrapassar o limite do tabuleiro esta volta para as coordenadas iniciais. O terceiro e quarto botão têm funções semelhantes e agem de forma idêntica, mas para a seleção da posição final. É de realçar, mais uma vez, que sempre que o jogador fizer uma jogada inválida o programa irá pedir para introduzir outras coordenadas para o movimento da peça.

De seguida será apresentada uma imagem que permite perceber o que foi realizado na leitura dos valores de tensão fornecidos nos botões, mas, por uma questão de não se tornar extenso e repetitivo, apenas será apresentado um desses casos, sendo que os outros podem ser extrapolados a partir deste.

```
void init_adc(void) {
 // Definir Vref=AVcc
 ADMUX = ADMUX | (1<<REFS0);
 // Desativar buffer digital em PC0
 DIDR0 = DIDR0 | (1<<PC0);
 // Pré-divisor em 128 e ativar ADC
 ADCSRA = ADCSRA | (7<<ADPS0)|(1<<ADEN);
}

unsigned int read_adc(unsigned char chan) {
 // escolher o canal...
 ADMUX = (ADMUX & 0xF0) | (chan & 0x0F);
 // iniciar a conversão
 // em modo manual (ADATE=0)
 ADCSRA |= (1<<ADSC);
 // esperar pelo fim da conversão
 while(ADCSRA & (1<<ADSC));
 return ADC;
}
```

Figura 22. – Inicialização ADC

```
while(!end){
    pos_xi = read_adc(0);
    pos_yi = read_adc(1);
    pos_xf = read_adc(2);
    pos_yf = read_adc(3);
    pos_select = read_adc(4);

    switch (xi_state)
    {
        case 0:
            if((pos_xi > 200) & !(yi_state | xf_state |yf_state | select)){
                // printf("pos_xi: %u\n",pos_xi);
                xi_state =  1;
            }
            break;
        case 1:
            if(pos_xi < 5){
                (*x_init) ++;
                xi_state = 0;
                if(*x_init > 7) *x_init = 0;
                printf("POS_INIT: [%c,%u] -> POS_FINAL: [%c,%u]\n",convNum2Char(*x_init),*y_init+1,convNum2Char(*x_fina
            }
            break;
        default:
            break;
    }
```

Figura 23. – Conversão da tensão num botão em movimentos no tabuleiro

2.3.5 LCD

Um outro extra interessante seria a configuração de um LCD para aí representar o tabuleiro, as peças dos dois jogadores (em jogo e capturadas) bem como as jogadas a serem selecionadas pelo utilizador. Para tal, um LCD convencional não funcionaria uma vez que a dimensão do mesmo não suportava o tamanho do tabuleiro (8x8) por isso, escolheu-se o LCD do nokia 5510 já que o mesmo apresenta mais espaço para a visualização de toda a informação. Finda toda a configuração do software a usar no LCD notou-se que o mesmo não respondia à informação que lhe era enviada, respondendo apenas a ligações feitas em hardware (o aparecimento da luz que iluminava o ecrã, já configurada para ser ajustável através de um potenciómetro). Após várias tentativas verificou-se que o problema não era do software, mas sim do LCD em si, já que o mesmo continuava a não funcionar corretamente e como tal não seria possível realizar

aquilo que era pretendido com o mesmo. De forma a substituir esta funcionalidade, optou-se por mostrar no terminal tudo aquilo que apareceria no LCD, de modo a manter a interface jogo-jogador.

3. Conclusões

Tal como referido no início deste relatório, o braço SCARA surgiu há algumas décadas e veio ajudar na alteração de paradigma que se vivia na época, que até aos dias de hoje ainda se mantém. A tendência da procura de novas coisas, da curiosidade acerca daquilo que não se conhece tão bem, da evolução tecnológica foi, e sempre será, uma grande valia do ser humano.
O projeto realizado, claramente mais virado para uma vertente de lazer, teve como propósito principal o enquadramento dos estudantes no funcionamento do Arduíno e na capacidade de obter informação a partir de uma datasheet. Além disso, também apresentou uma vertente mais virada para a construção/montagem que também surge como algo importante na vida de um engenheiro.
Promoveu, assim, uma aprendizagem mais prática sobre configuração de timers, controlo de servos, escrita em memória não volátil (EEPROM), conversão analógica-digital e criação de um buzzer a partir do Arduíno.
Para terminar há alguns pontos que, do ponto de vista do grupo, poderiam contribuir para uma melhor qualidade no projeto, tais como: a falta e um suporte resistente e forte para aguentar o braço, o que fez com que as posições do tabuleiro que iam sendo retiradas apresentavam erro, o que levava a novas medidas que, muitas das vezes, continuavam com erro. Em segundo, e último, lugar, devido ao peso na parte da frente do SCARA, os servos encontravam-se muitas vezes sob pressão, o que pode ter levado às folgas descritas acima e não é aconselhável. Além disso, mais uma vez, fazia com que o braço descaísse e as medições de posição apresentassem erros.