Jogo Para Auxiliar na Reabilitação de Pessoas Com Debilidade Motora no Punho

-Projeto Final-

Microprocessadores e Microcontroladores

Vitor Carvalho de Oliveira
RA: 14/0165498
Universidade de Brasília, Campus Gama
vitor.carvalhoo@icloud.com

Douglas Afonso Xavier de Rezende
RA: 13/0107816
Universidade de Brasília, Campus Gama
douglasafonso@outlook.com

***Resumo* — O intuito deste projeto é modernizar um clássico jogo chamado Marble Maze, incluindo neste a eletrônica embarcada, utilizando o microcontrolador MSP430, para auxiliar na reabilitação de pessoas que sofreram algum tipo de lesão envolvendo a região do punho ou que possuem algum tipo de patologia específica desta região, além de melhorar a coordenação motora e desenvolver um maior envolvimento cognitivo.**

Palavras-chave – Jogo, MSP430, Reabilitação, Punho, Microcontroladores, Eletrônico, Movimento.

I. INTRODUÇÃO

Nos últimos anos, os jogos eletrônicos e digitais de movimento, assim chamados por tornarem importante o corpo e os movimentos na interação com o jogo, ganharam visibilidade e passaram a receber grandes investimentos da indústria de jogos eletrônicos. Grandes empresas como; Sony, Microsoft e Nintendo investiram fortemente em acessórios e periféricos que dessem suporte para jogos de movimento na sétima geração de consoles (Ps3, Xbox 360 e Wii).

Figura 1: Acessórios que dão suporte aos jogos de movimento da sétima geração de consoles.

Muitos dos jogos de movimento desenvolvidos, envolviam e simulavam práticas esportivas, como por exemplo, lutas, corrida, “snowboard”, golf, entre outros, que estimulavam os jogadores à prática de exercícios físicos, promovendo gasto calórico e melhorando a coordenação motora dos jogadores, além da diversão e do entretenimento garantido.

Atualmente os jogos eletrônicos de movimento têm ganhado um novo propósito, o de auxiliar na reabilitação de pessoas com alguma deficiência motora ou que sofreram algum tipo de lesão corporal que necessitam de sessões de fisioterapia para recuperar certos movimentos. Segundo o terapeuta ocupacional Fábio Galvão, formado pela Universidade de Potiguar, a união dos jogos eletrônicos de movimento com o acompanhamento profissional, pode trazer uma série de benefícios, tais como:

- Fortalecimento muscular;
- Melhorar a amplitude dos movimentos;
- Ampliação da atividade cerebelar;
- Estímulo à concentração e ao equilíbrio;
- Sensação de superação;
- Maior envolvimento cognitivo;

Já existem relatos da utilização, com resultados satisfatórios, da Terapia Ocupacional e da Fisioterapia, aliadas aos jogos eletrônicos de movimento em vários países, como Estados Unidos, Alemanha, Inglaterra, entre outros.

Seguindo essa tendência, este projeto visa modernizar e incluir elementos da eletrônica embarcada em um clássico jogo chamado “Marble Maze” (Labirinto) para que ele possa ser utilizado na reabilitação de pessoas com algum tipo de deficiência motora no punho. Este jogo criado em 1946 pela empresa de brinquedos BRIO, consiste em um tabuleiro, geralmente de madeira, com um labirinto e obstáculos, cujo objetivo é levar uma bola de gude até um ponto específico do labirinto por meio de inclinações manuais do tabuleiro.

Figura 2: Jogo Marble Maze.

II. JUSTIFICATIVA

Pessoas que sofreram fratura de punho: Ossos do antebraço na região distal tanto no osso do rádio quanto no osso da ulna ou lesões de tendões: tendão flexor e extensor de punho de dedos ou que convivem com alguma patologia específica de punho: Síndrome do Túnel do Carpo, Síndrome de Canal de Guyon, Tendinites, Cistos Sinoviais, Dedo em gatilho, Artrite localizada no punho, Síndrome do Túnel Radial, Tenossinovite de Quervain, ou que foram vítimas de um Acidente Vascular Cerebral (AVC) ou paralisia cerebral que sofrem de espasticidade (distúrbio de controle muscular caracterizado por músculos tensos ou rígidos) geralmente passam por diversas sessões intensivas e massivas de fisioterapia.

Essas sessões, apesar de serem fundamentais na reabilitação e estimularem os pacientes a se manterem sempre ativos, fortalecendo músculos, articulações, e promovendo uma recuperação eficaz, na maioria dos casos, são vistas como monótonas, repetitivas e entediosas pelos pacientes.

A motivação desse projeto, portanto, é tornar essas sessões de reabilitação do punho mais atraentes, divertidas, estimulantes e lúdicas para os pacientes, mediante a utilização da versão eletrônica do jogo Marble Maze.

Os movimentos fisiológicos executados pelo punho e os graus de amplitude de movimento são: Flexão (para baixo 90º) extensão (para cima 70º) desvio radial (para o lado do polegar 20º) desvio ulnar (para o lado do dedo mínimo 45º) pronação (90º) e supinação (90º). Esses são os movimentos que são explorados na fisioterapia.

Figura 3: Exemplo de movimento fisiológico do punho.

Figura 4: Exemplo de movimento fisiológico do punho.

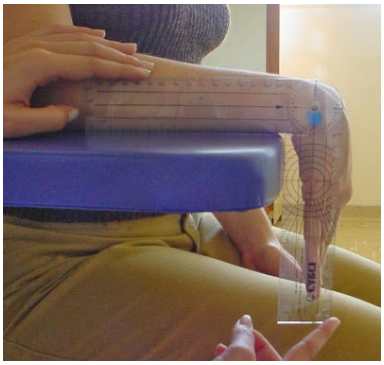

Figura 5: Exemplo de movimento fisiológico do punho.

III. OBJETIVOS

A proposta deste projeto é modernizar e incluir a eletrônica embarcada utilizando o MSP430 em um clássico jogo chamado “Marble Maze” O objetivo é que os movimentos fisiológicos do punho sejam os responsáveis por inclinar o tabuleiro, e para isso, será utilizado sensores de movimento (giroscópio e acelerômetro) para capturar as rotações e inclinações do punho e traduzir em sinais que atuarão em servomotores que estarão distribuídos por baixo do tabuleiro para realizar as inclinações.

Com isso o jogo visa ser um auxiliador na reabilitação de pessoas com alguma deficiência motora na região do punho, promovendo a exercitação focalizada dos movimentos fisiológicos do punho e trabalhando a amplitude desses movimentos, além de proporcionar sessões de fisioterapia mais diversificadas, como citado anteriormente.

IV. REQUISITOS

O jogo deverá ser capaz de identificar os 3 eixos de inclinação/movimentação do punho além dos eixos combinados de outros dois eixos, por exemplo as inclinações nas diagonais, e para isso será utilizado um módulo eletrônico que contem giroscópio e acelerômetro, para capturar essas inclinações e traduzir em sinais:

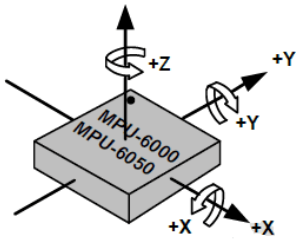

Figura 6: Módulo giroscópio e acelerômetro e os graus de liberdade possíveis.

Esses sinais serão enviados para o MSP430 que irá traduzí-los e convertê-los em sinais de saída para os servomotores, distribuídos embaixo do tabuleiro, para realizar as inclinações de acordo com o grau de amplitude do movimento do punho.

O jogador deverá segurar o módulo giroscópio/acelerômetro com a posição vertical do punho e com o dedo polegar apontado para cima, essa será a posição inicial do jogo onde o tabuleiro sofrerá nenhuma inclinação. A partir dessa posição o jogador deverá realizar as inclinações do punho (esquerda, direita, flexão e extensão além das combinações desses movimentos) para efetuar o controle dos servomotores e explorar os movimentos fisiológicos do punho: supinação, pronação, desvio ulnar e desvio radial

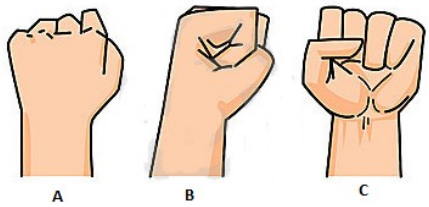

Figura 7: (a) Movimento de rotação do punho pra direita (b) Posição inicial do jogo (c) Movimento de rotação do punho pra esquerda. Observação: o polegar deverá estar estendido.

Haverá 4 servomotores, cada um localizado no meio de cada aresta do tabuleiro. Caso seja realizado, por exemplo, o movimento (a) da figura 7, o servomotor "A", ilustrado na imagem abaixo, atuará inclinando o tabuleiro para a direita. Caso seja efetuado uma combinação de movimentos, como por exemplo, rotação para esquerda e extensão do punho, os servomotores "B" e "D" atuarão inclinando o tabuleiro na diagonal. Essa será a mecânica do jogo.

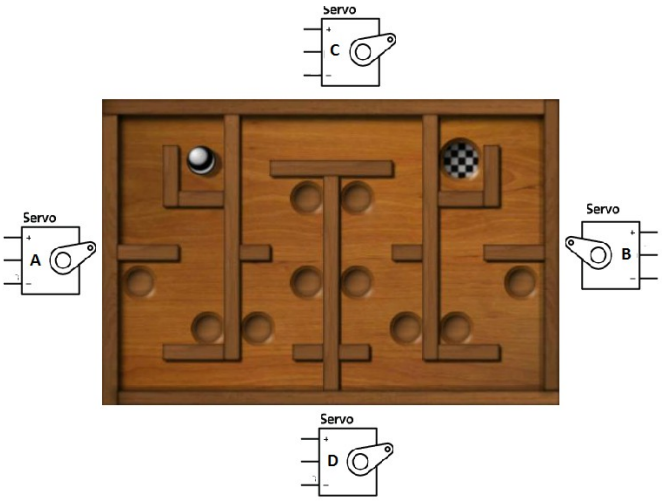

Figura 8: Disposição dos servomotores no tabuleiro.

O tabuleiro, confeccionado em madeira MDF bem como o labirinto e os obstáculos, serão planejados para que o jogo possa explorar e trabalhar diferentes movimentos de punho com diferentes amplitudes.

Além disso, o jogo contará com um sistema de feedback para avaliar as progressões e melhorias do paciente, esse sistema consiste em displays que informarão o tempo que o paciente demorou para solucionar o labirinto.

V. Materiais Utilizados

Nome	Quantidade
Servomotores TowerPro 9g SG-90	4
Módulo Acelerômetro/Giroscópio MPU-6050	1
MSP430G2553	1
Resistor 1k	3
Resistor 56	7
Transistor BC557	3
CI 7447	1
"Push Button"	1
Display 7 Segmentos (Ânodo Comum)	3

VI. Descrição de hardware

Para a alimentação do circuito, decidiu-se separá-lo em dois blocos, um alimentado com 5V (módulo de aquisição de dados e controle dos servos) e o outro com 3.3V (sistema de feedback). Para obter uma saída de 5V do MSP soldou-se um pino na saída TP1 da placa do MSP430 para obter 5V de

alimentação do circuito, proveniente do cabo USB ligado ao computador, visto que os servomotores e o MPU-6050 operam com esse nível de tensão. Conectou-se os pinos SDA e SCL do MPU-6050 nos pinos P1.6 e P1.7 do MSP430 respectivamente, que são os pinos capazes de realizar a comunicação I2C, além disso, conectou-se os fios de controle dos servomotores nos pinos P2.1 e P2.4, como o MSP430G2553 possui apenas três saídas PWM (para controlar os servos motores) decidiu-se utilizar duas, uma para cada par de servo diametralmente oposto, e posicioná-los de tal forma que o movimento deles fossem espelhados.

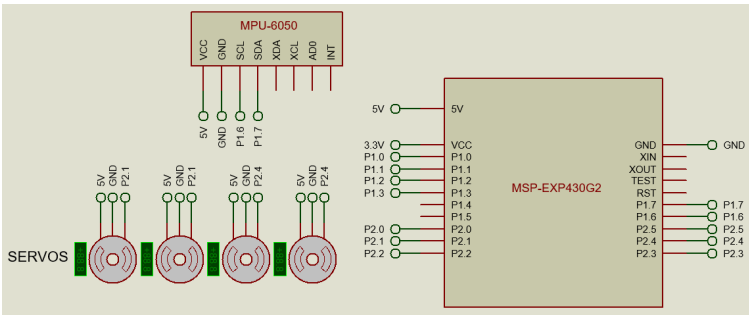

Figura 9: Esquemático do bloco do circuito alimentado com 5V.

No sistema de feedback, com objetivo de economizar pinos do MSP, utilizou-se o circuito integrado (CI) 7447 para decodificar a saída BCD que sai dos pinos: P1.0, P1.1, P1.2 e P1.3 para os displays de 7 segmentos. Cada saída do CI 7447 é conectado em um resistor de 56 Ohms (para evitar queima dos leds do display) e depois conectado nos segmentos dos display (a, b, c, d, e, f, g). Para a multiplexação dos displays utilizou-se os pinos P2.0, P2.2 e P2.3 do MSP, cada um desses pinos é conectado em um resistor de 1k Ohms e depois na base de um transistor (BC557) para o chaveamento e amplificação de corrente para os displays, o emissor de cada transistor é ligado em 3.3V e o coletor é conectado do pino comum de cada display. O botão para iniciar, pausar, zerar e reiniciar a contagem foi ligado no ground do circuito e no pino P2.5 do MSP.

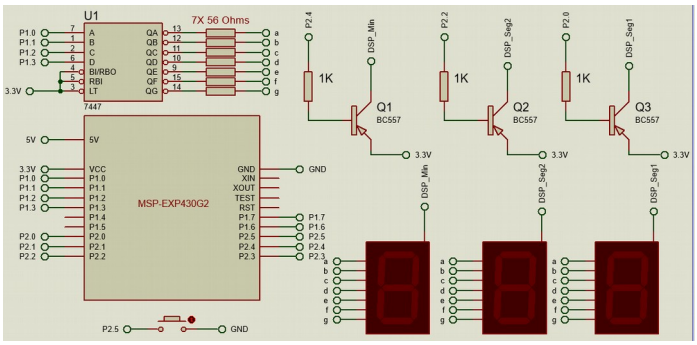

Figura 10: Esquemático do bloco do circuito alimentado com 3.3V.

VII. Descrição de Software

Para desenvolver o software embarcado, utilizou-se o software "Code Composer". A linguagem de programação utilizada foi C. O diagrama de blocos abaixo, resume as etapas de funcionamento do código:

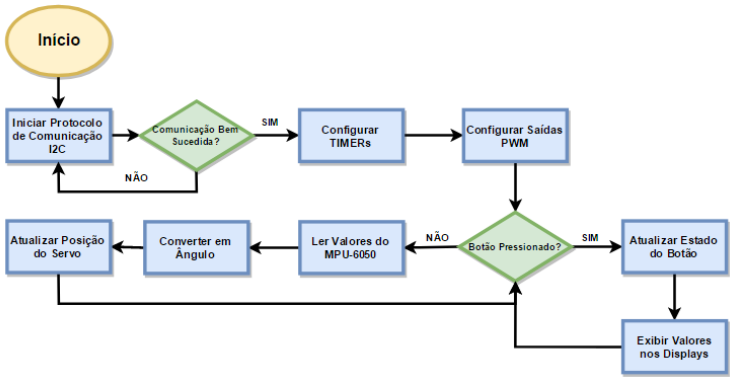

Figura 11: Diagrama de blocos do software embarcado.

O código inicia tentando efetuar o protocolo de comunicação I2C com o MPU-6050, esse tipo de protocolo opera no modelo "mestre-escravo" em que um dispositivo atua como mestre (MSP430) e os demais como escravo, neste caso o MPU6050. A função do mestre é coordenar a comunicação, ou seja, é ele quem envia as informações a determinado escravo ou obtém informações.

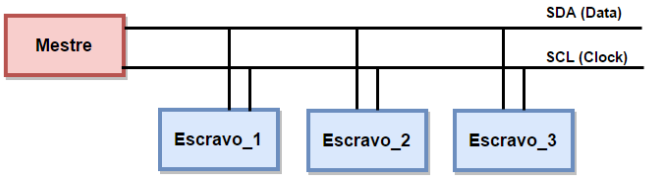

Figura 12: Esquemático de ligações da comunicação I2C

Os dispositivos I2C possuem um endereço, geralmente de 7 bits e expresso em hexadecimal, que os identifica, o MPU6050, por exemplo, possui endereço 0x68. Além desses bits de endereçamento, existe o oitavo bit para identificar quando é uma operação é de escrita ou de leitura (read/write).

Figura 13: Bits de endereçamento do dispositivo escravo.

Após inicializar o protocolo I2C, é feito a configuração das saídas PWM para controlar os servomotores, para isso, utilizou-se os TIMERs disponíveis no MSP430G2553 (TIMER0 e o TIMER1) O TIMER0 possui dois registradores de captura disponíveis (TA0CCR0 e o TA0CCR1) já o TIMER1 possui três (TA1CCR0, TA1CCR1, TA1CCR2) Os registradores TAxCCR0 possuem um endereço de interrupção de maior prioridade e, portanto, são eles que controlam o funcionamento dos TIMERs de forma geral. Além disso existem dois registradores responsáveis por configurar o modo de funcionamento dos TIMERs: o TA0CTL e o TA1CTL.

Para gerar as saídas PWM configurou-se TIMERs no modo "Up" (MC_1) para que o parâmetro de parada da contagem sejam os valores armazenados em TAxCCR0 e no modo de comparação para que os sinais de saída sejam alterados por hardware quando a contagem chegar no valor armazenado nos registradores de captura (modo 7 de operação) gerando assim, o sinal PWM, conforme a imagem abaixo:

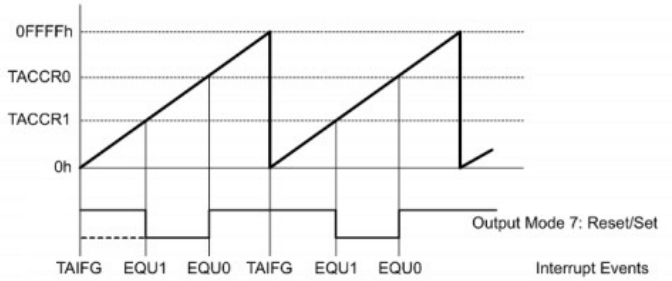

Figura 14: Sinal de saída com base nos valores armazenados nos registradores de captura.

O servomotor utilizado neste projeto é controlado com um sinal com frequência de 50 Hz e pulsos com larguras de 1 até 2 ms, variando a largura do pulso dentro desses limites é possível alterar a posição do servo em até 180º, conforme observa-se na imagem abaixo:

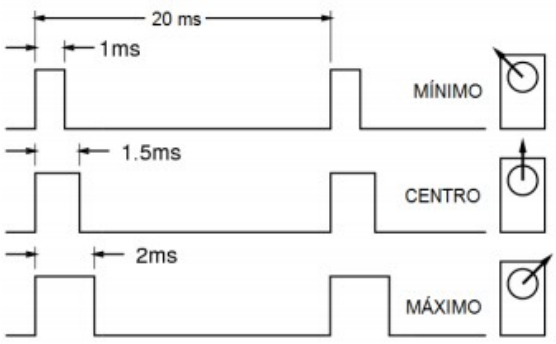

Figura 15: Sinais de controle do servomotor.

Os valores obtidos pelo MPU-6050, são convertidos pela função map, que parametriza o intervalo de valor lido pelo MPU em um intervalo de angulação dos servos.

```
long map(long X, long in_min, long in_max, long out_min, long out_max)
{
  return (X - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

Figura 2: Descrição operacional da função map.

- X: Número a ser mapeado, geralmente armazenado em uma variável (AcX e Acy);
- in_min: Limite inferior do intervalo de valores atuais (0);
- in_max: Limite superior do intervalo de valores atuais (-17000 ou 17000);
- out_min: Limite inferior do novo intervalo de valores mapeados (90º);
- out_max: Limite superior do novo intervalo de valores mapeados (55º);

Do jeito que foram estabelecidos os parâmetros da função "map" garantiu-se que cada motor inclinaria de acordo com a direção e sentido de movimento do punho. Após feito a conversão, esses valores são repassados para os registradores dos TIMERs, responsáveis por gerar o sinal PWM com largura de pulso entre 1 e 2 ms.

Para o sistema de feedback o TIMER responsável foi configurado de forma que aconteça uma interrupção a cada 100ms, e a cada interrupção uma variável inteira "unidade" é incrementada. Como os display acionam com nível baixo, as saídas foram todas predefinidas com nível alto no início do programa. Em um loop infinito está a multiplexação dos displays e o envio do sinal de saída para os segmentos, na forma de switch-case para cada dígito dos displays. O switch-case que varia de acordo com a "unidade" tem 11 casos, para valores de 0 a 10, de forma que cada um dos casos de 0 a 9 envia o digito para os displays, e o caso 10 retorna o valor da "unidade" para 0 e incrementa a variável "dezena". O mesmo ocorre com as variáveis "dezena" e "centena", mas no caso da "dezena" há incremento da "centena", e no caso da "centena" não há incremento de outras variáveis. Em cada switch-case as saídas P2.0, P2.2 e P2.3 são alternadas de forma que os displays apaguem e no fim de cada switch-case o display correspondente é ligado por uma dessas portas.

Para o botão que inicia, pausa, zera e reinicia a contagem foi programado uma interrupção que era gerada toda vez que o botão era pressionado, nessa interrupção o valor da variável "botão" era atualizada e dependendo do valor dessa variável alguma das ações: iniciar, pausar, zerar ou reiniciar era efetuada.

```
#pragma vector = PORT2_VECTOR
__interrupt void Port_2(void){

    if((P2IN&BIT5)==0)
    {
        if(botao == 0)
            botao = 1;
        else if(botao == 1)
            botao = 2;
        else if(botao == 2)
            botao = 0;
    }
    __delay_cycles(60000);
    P2IFG &= ~button;
    P2IES ^= button;

}
```

Figura 16: Interrupção associada ao botão.

VIII. Descrição da estrutura

Para confecção da estrutura adquiriu-se uma caixa de madeira MDF com dimensões: 21 cm de comprimento, 21 cm de largura e 8,5 cm de altura onde foram posicionados os 4 servomotores no centro de cada aresta da caixa. Para o tabuleiro, onde ficará o labirinto, também utilizou-se madeira MDF com dimensões: 16 cm de comprimento, 16 cm de largura e 2 cm de altura que foi posicionado sobre os servomotores, conforme as imagens abaixo:

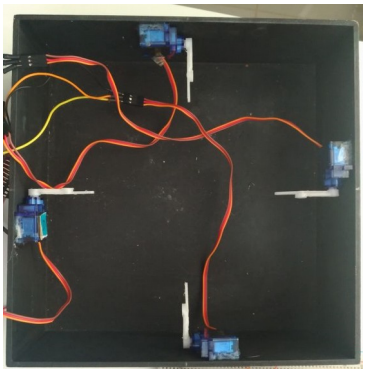

Figura 17: Estrutura do projeto visto de cima.

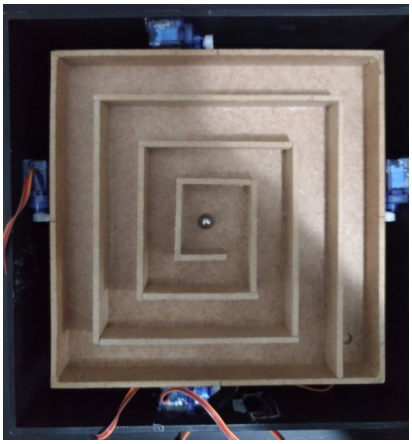

Figura 18: Estrutura do tabuleiro visto de cima.

IX. Resultados

Os primeiros testes do protótipo do projeto foram realizados utilizando o software "Energia" e as bibliotecas e funções prontas disponíveis para validar a viabilidade do projeto. Nesses primeiros testes foi avaliado a aquisição de dados do módulo MPU-6050 e a conversão desses dados em sinais de controle para os servos, que funcionaram como esperado. Nos testes seguintes, foi realizado a migração do código para o software "Code Composer" com o objetivo de otimizá-lo utilizando uma programação com um nível de abstração menor comparado com a anterior, no qual a definição de variáveis e o mapeamento das portas I/O foram feitos manipulando os registradores do microcontrolador, apesar de ter encontrado algumas dificuldades em migrar o código, obteve-se o resultado esperado.

Após validar essa parte do circuito, implementou-se separadamente o código do circuito de feedback direto no "Code Composer" e obteve o resultado quase esperado, pois o botão que inicia, pausa, zera e reinicia o contador estava com problema de "bouncing" e as vezes algum estado do botão era pulado. Após esse teste, verificou-se que o MSP430 não teria portas suficientes para todo o circuito do projeto, a solução foi utilizar o CI 7447 que decodifica informação em BCD para display de 7 segmentos, com isso, ao invés do MSP40 utilizar 7 pinos para controlar os segmentos do displays, passou para 4.

Por último, foi feito a integração dos dois circuitos e códigos e observou-se o funcionamento adequado, porém os displays dos segundos ficaram com a luz fraca. Uma possível explicação para esse evento pode ser uma demanda elevada de corrente pelo circuito na qual o MSP não era capaz de fornecer. Além disse, o problema de "bouncig" do botão persistiu. Uma possível solução para esse problema seria aumentar o tempo de atraso após a identificação do sinal de entrada pelo botão.

Em relação ao funcionamento mecânico do jogo, nos primeiros testes observou-se que o tabuleiro se deslocava demais sobre os servos o que implicava na instabilidade do controle, a solução encontrada foi fazer um desnível na madeira na posição onde cada alavanca do servo sustentava o tabuleiro, de tal forma a evitar o deslocamento excessivo do mesmo e melhorar a estabilidade do controle. Além disso, observou-se que a bola do jogo escorregava com facilidade sobre o tabuleiro, o que limitava a amplitude de movimento do punho, uma possível solução seria aumentar o atrito do tabuleiro colocando algum tipo de material que limitasse um pouco mais o movimento da bola, como por exemplo o tecido camurça ou algum emborrachado.

X. Conclusão

Após a elaboração do projeto e algumas conversas com fisioterapeutas observou-se que o jogo apresenta

potencial no campo de tecnologias assistivas e no conceito de gameterapia que está bastante em voga atualmente. O jogo que é uma modernização do clássico jogo Marble Maze, utiliza recursos de tecnologias de captura de movimento e da eletrônica embarcada para fornecer sessões de fisioterapia mais lúdicas e estimulantes para os pacientes que possuem alguma debilidade motora no punho.

REFERÊNCIAS

[1] Ana Paula Salles da Silva, "Os jogos de movimento e as práticas corporais na percepção de jovens" Tese de pós graduação em Educação Física, 2012.

[2] Erik Nardini Medina, "Realidade virtual pode tornar sessões de fisioterapia mais estimulantes" Reportagem da Revista Inovação, 2016.

[3] BRIO catalog. Retrieved 2011-02-14.

[4] Rodrigo Flausino, "Os jogos eletrônicos e seus impactos na sociedade", 2008.

[5] ACE, Gestão em Saúde, "Manual de Goniômetria Medição dos Ângulos Articulares"

[6] Marques, Amélia Pasqual – Manual de Goniômetria – 2. Ed. Barueri, SP: Manole, 2003. ISBN 85-204-1627-6

[7] Davies, J. H – MSP430 Microcontroller Basics .

APÊNDICE

Código Completo:

```
#include <msp430.h>

#define MCU_CLOCK            1000000
#define PWM_FREQUENCY        46

#define SERVO_STEPS          180
#define SERVO_MIN            650
#define SERVO_MAX            2700

#define P1off 0x0f;
#define P2off (BIT0+BIT2+BIT3);

#define P1_0 0xf0;
#define P1_1 0xf1;
#define P1_2 0xf2;
#define P1_3 0xf3;
#define P1_4 0xf4;
#define P1_5 0xf5;
#define P1_6 0xf6;
#define P1_7 0xf7;
#define P1_8 0xf8;
#define P1_9 0xf9;

#define P2_u BIT0;
#define P2_d BIT2;
#define P2_c BIT3;

#define button BIT5;

unsigned int PWM_Period     = (MCU_CLOCK / PWM_FREQUENCY);  // PWM Period
unsigned int PWM_Duty       = 0;
// %

unsigned char RX_Data[6];
unsigned char TX_Data[2];
unsigned char RX_ByteCtr;
unsigned char TX_ByteCtr;

int xAccel;
int yAccel;

int val_dir;
int val_cima;

int preVal_dir;
int preVal_cima;

int unidade = 0;
int dezena = 0;
int centena = 0;
int aux = 0;
int botao = 0;
int unidade_s = 0;
int dezena_s = 0;
int centena_s = 0;

unsigned char slaveAddress = 0x68;

const unsigned char PWR_MGMT_1   = 0x6B;
// MPU-6050 register address
const unsigned char ACCEL_XOUT_H = 0x3B;
// MPU-6050 register address
const unsigned char ACCEL_XOUT_L = 0x3C;
// MPU-6050 register address
const unsigned char ACCEL_YOUT_H = 0x3D;
// MPU-6050 register address
const unsigned char ACCEL_YOUT_L = 0x3E;
// MPU-6050 register address
const unsigned char ACCEL_ZOUT_H = 0x3F;
// MPU-6050 register address
const unsigned char ACCEL_ZOUT_L = 0x40;
// MPU-6050 register address

void i2cInit(void);
void i2cWrite(unsigned char);
void i2cRead(unsigned char);
long map(long,long,long,long,long);
void Begin_Pause_Reset(int);
void Uni(int);
void Dez(int);
```

```
void Cent(int);

int main(void)
 {
    unsigned int servo_stepval,
servo_stepnow;
    unsigned int servo_lut[ SERVO_STEPS+1 ];
    unsigned int i;


    servo_stepval   = ((SERVO_MAX -
SERVO_MIN) / SERVO_STEPS );
    servo_stepnow   = SERVO_MIN;


     for (i = 0; i < SERVO_STEPS; i++) {
        servo_stepnow += servo_stepval;
        servo_lut[i] = servo_stepnow;
    }

    WDTCTL = WDTPW + WDTHOLD;
// Stop WDT

    // Set clock speed (default = 1 MHz)
    BCSCTL1 = CALBC1_1MHZ;
// Basic Clock System CTL (1,8,12 16_MHZ
available)
    DCOCTL  = CALDCO_1MHZ;
// Digitally-Controlled Oscillator CTL

    TA1CCTL1 = OUTMOD_7;
    TA1CCTL2 = OUTMOD_7;

    TA0CCTL0 = CCIE;
    TA0CCTL0 &= ~CCIFG;

    TA1CTL   = TASSEL_2 + MC_1;
    TA0CTL   = TASSEL_2 + ID_2 + MC_1;

    TA1CCR0  = PWM_Period-1;
    TA1CCR1  = PWM_Duty;
    TA1CCR2  = PWM_Duty;

    TA0CCR0 = 0x61A8;

    // set up I2C pins
    P1SEL |= BIT6 + BIT7;
// Assign I2C pins to USCI_B0
    P1SEL2|= BIT6 + BIT7;
// Assign I2C pins to USCI_B0

    P1DIR   |= 0x0F;
//seta output direction para os bits
    P2DIR   |= BIT0+BIT2+BIT3;
//seta output direction para os bits 0,2,3  -
para multiplex dos displays
    P2DIR   |= BIT1+BIT4;
    P2SEL   |= BIT1+BIT4;

    P2OUT |= BIT0+BIT2+BIT3;   //apaga com
sinal alto
    P1OUT |= 0x0F;  //apaga com sinal alto,
com 1111 o 74ls47 apaga todos os leds

    P2DIR &= ~button;
    P2REN |= button;
    P2IE  |= button;          // button
interrupt enabled
    P2IFG &= ~button;        // P2.3 IFG
cleared

    //Habilita interrupção
      //__bis_SR_register(GIE); //habilita
interrupção
      __enable_interrupt(); //outra forma de
habilitar interrupção
      //_BIS_SR(GIE);

    // Initialize the I2C state machine
    i2cInit();

    // Wake up the MPU-6050
    slaveAddress = 0x68;
// MPU-6050 address
    TX_Data[1] = 0x6B;
// address of PWR_MGMT_1 register
    TX_Data[0] = 0x00;
// set register to zero (wakes up the MPU-
6050)
    TX_ByteCtr = 2;
    i2cWrite(slaveAddress);

    while (1)
    {
        // Point to the ACCEL_ZOUT_H register
in the MPU-6050
        slaveAddress = 0x68;
// MPU-6050 address
        TX_Data[0] = 0x3B;
// register address
        TX_ByteCtr = 1;
        i2cWrite(slaveAddress);

        Begin_Pause_Reset(botao);
        Uni(unidade_s);
        Dez(dezena_s);
        Cent(centena_s);

        if(unidade == 10)
        {
          dezena++;
          unidade = 0;
```

```
        }

        if(dezena == 6)
        {
          centena++;
          dezena = 0;
        }

        if(centena == 10)
        {
          botao++;
        }

        if(aux == 10)
        {
          unidade++;
          aux = 0;
        }

        // Read the two bytes of data and
store them in zAccel
        slaveAddress = 0x68;
// MPU-6050 address
        RX_ByteCtr = 6;
        i2cRead(slaveAddress);
        xAccel  = RX_Data[5] << 8;
// MSB
        xAccel |= RX_Data[4];
// LSB
        yAccel  = RX_Data[3] << 8;
// MSB
        yAccel |= RX_Data[2];
// LSB

        //zAccel  = RX_Data[1] << 8;
// MSB
        //zAccel |= RX_Data[0];
// LSB
        val_dir = map(xAccel,0,17000,90,55);
        val_cima = map(yAccel,0,-
17000,90,55);

        if(val_dir != preVal_dir){

            TA1CCR1 = servo_lut[val_dir];
            preVal_dir = val_dir;
        }

        if(val_cima != preVal_cima){

            TA1CCR2 = servo_lut[val_cima];
            preVal_cima = val_cima;
        }

        // do something with the data
            __no_operation();
// Set breakpoint >>here<< and read

;           __delay_cycles(20000);
    }
}

void i2cInit(void)
{
    // set up I2C module
    UCB0CTL1 |= UCSWRST;                  //
Enable SW reset
    UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;
// I2C Master, synchronous mode
    UCB0CTL1 = UCSSEL_2 + UCSWRST;
// Use SMCLK, keep SW reset
    UCB0BR0 = 10;                       // fSCL =
SMCLK/12 = ~100kHz
    UCB0BR1 = 0;
    UCB0CTL1 &= ~UCSWRST;                 //
Clear SW reset, resume operation
}


void i2cWrite(unsigned char address)
{
    __disable_interrupt();
    UCB0I2CSA = address;                  //
Load slave address
    IE2 |= UCB0TXIE;                    // Enable
TX interrupt
    while(UCB0CTL1 & UCTXSTP);            //
Ensure stop condition sent
    UCB0CTL1 |= UCTR + UCTXSTT;           // TX
mode and START condition
    __bis_SR_register(CPUOFF + GIE);
// sleep until UCB0TXIFG is set ...
}

void i2cRead(unsigned char address)
{
    __disable_interrupt();
    UCB0I2CSA = address;                  //
Load slave address
    IE2 |= UCB0RXIE;                    // Enable
RX interrupt
    while(UCB0CTL1 & UCTXSTP);            //
Ensure stop condition sent
    UCB0CTL1 &= ~UCTR;                  // RX
mode
    UCB0CTL1 |= UCTXSTT;                  //
Start Condition
    __bis_SR_register(CPUOFF + GIE);
// sleep until UCB0RXIFG is set ...
}
```

```
long map(long Accel, long in_min, long
in_max, long out_min, long out_max)
{
  return (Accel - in_min) * (out_max -
out_min) / (in_max - in_min) + out_min;
}

void Uni(int unidade_s){

    switch (unidade_s) {
        case 0:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_0;
          break;
        case 1:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_1;
          break;
        case 2:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_2;
          break;
        case 3:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_3;
          break;
        case 4:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_4;
          break;
        case 5:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_5;
          break;
        case 6:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_6;
          break;
        case 7:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_7;
          break;
        case 8:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_8;
          break;
        case 9:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_9;
          break;
        case 10:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_0;
          break;
        }
      P2OUT &= ~P2_u; //acende o led da
unidade
}

void Dez(int dezena_s){

    switch (dezena_s) {
        case 0:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_0;
          break;
        case 1:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_1;
          break;
        case 2:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_2;
          break;
        case 3:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_3;
          break;
        case 4:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_4;
          break;
        case 5:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_5;
          break;
        case 6:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_6;
          break;
        case 7:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_7;
```

```
          break;
        case 8:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_8;
          break;
        case 9:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_9;
          break;
        case 10:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_0;
          break;
      }
    P2OUT &= ~P2_d; //acende o led da dezena
}

void Cent(int centena_s){

    switch (centena_s) {
        case 0:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_0;
          break;
        case 1:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_1;
          break;
        case 2:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_2;
          break;
        case 3:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_3;
          break;
        case 4:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_4;
          break;
        case 5:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_5;
          break;
        case 6:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_6;
          break;
        case 7:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_7;
          break;
        case 8:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_8;
          break;
        case 9:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_9;
          break;
        case 10:
          P1OUT |= P1off;
          P2OUT |= P2off;
          P1OUT &= P1_0;
          break;
      }
    P2OUT &= ~P2_c; //acende o led da centena
}

void Begin_Pause_Reset(int botao){

    if(botao == 0)
    {
      unidade = 0;
      dezena = 0;
      centena = 0;

      unidade_s = 0;
      dezena_s = 0;
      centena_s = 0;
    }
    else if(botao == 1)
    {
      unidade_s = unidade;
      dezena_s = dezena;
      centena_s = centena;
    }
}

// USCIAB0TX_ISR
#pragma vector = USCIAB0TX_VECTOR
__interrupt void USCIAB0TX_ISR(void)
{
    if(UCB0CTL1 & UCTR)                 // TX
mode (UCTR == 1)
    {
        if (TX_ByteCtr)
// TRUE if more bytes remain
        {
```

```
            TX_ByteCtr--;                // Decrement TX byte counter
            UCB0TXBUF = TX_Data[TX_ByteCtr];
// Load TX buffer
        }
        else                         // no more bytes to send
        {
            UCB0CTL1 |= UCTXSTP;
// I2C stop condition
            IFG2 &= ~UCB0TXIFG;          // Clear USCI_B0 TX int flag

__bic_SR_register_on_exit(CPUOFF);  // Exit LPM0
        }
    }
    else // (UCTR == 0)                 // RX mode
    {
        RX_ByteCtr--;
// Decrement RX byte counter
        if (RX_ByteCtr)
// RxByteCtr != 0
        {
            RX_Data[RX_ByteCtr] = UCB0RXBUF;
// Get received byte
            if (RX_ByteCtr == 1)
// Only one byte left?
            UCB0CTL1 |= UCTXSTP;
// Generate I2C stop condition
        }
        else                         // RxByteCtr == 0
        {
            RX_Data[RX_ByteCtr] = UCB0RXBUF;
// Get final received byte

__bic_SR_register_on_exit(CPUOFF);  // Exit LPM0
        }
    }
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void interrupt_timer(void)
{
  aux++;

  TA0CCTL0 &= ~CCIFG; //limpa flag de interrupção
}

// Port 2 interrupt service routine
#pragma vector = PORT2_VECTOR
__interrupt void Port_2(void){
    if((P2IN&BIT5)==0)
    {
        if(botao == 0)
            botao = 1;
        else if(botao == 1)
            botao = 2;
        else if(botao == 2)
            botao = 0;
    }
    __delay_cycles(60000);
    P2IFG &= ~button; // P2.3 IFG cleared
    P2IES ^= button; // toggle the interrupt edge,

}
```