Introdução

A presente documentação vem com o intuito de descrever tecnicamente o firmware de um PIC18F4431 usado na placa do robô de sumô 3kg Autônomo MetalGarurumon. O documento não tem a intenção de discutir linha por linha do código, mas antes abordar os fatores mais importantes de cada bloco de código (Configurações, Programa Principal, Análise de Sensores, entre outros). Todas as linhas de código estão disponíveis como um anexo. Com o intuito de não tornar este documento desnecessariamente grande, será feito uso de diagramas e fluxogramas para explicar o conceito empregado em determinada parte do código ao invés de explicação linha a linha. Abaixo segue uma tabela com as principais características do software.

Linguagem	**C**
Compilador	**CCS**
Última Versão	**4.0**
Microcontrolador	**PIC18F4431**
Robô	**MetalGarurumon**

Tabela 1. Características principais do software

Estrutura de Arquivos.

O software é composto por por 4 arquivos e estão relacionados entre si como a figura abaixo mostra.

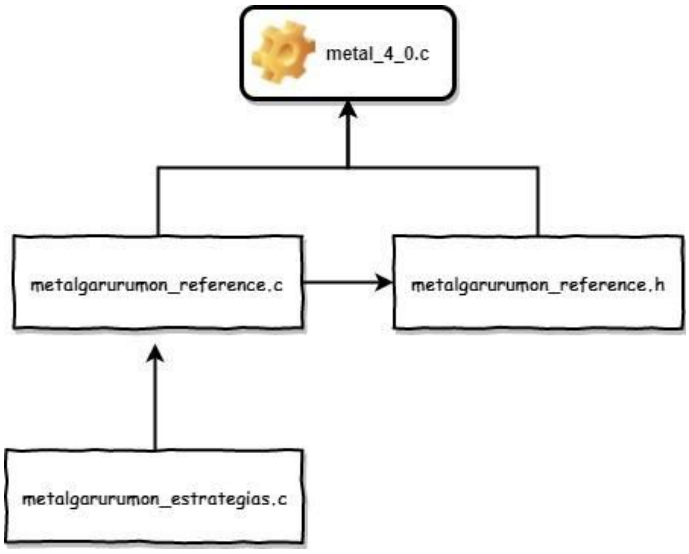

Figura 1. Relação entre arquivos do Firmware.

metal_4_0.c - Esse é o arquivo principal do programa, o que contém a função main() e chama apenas algumas funções. O objetivo foi tornar esse arquivo o mais reduzido possível para que ele focasse em alguns funcionamentos mais essenciais e primários do funcionamento do robô, como a declaração de rotina de interrupção e processo de acionamento do robô.

metalgarurumon_reference.h - Aqui são realizadas as declarações de todas as variáveis globais e também a declaração (mas não implementação) das funções que serão usadas no arquivo metalgarurumon.c. É neste arquivo que os #define para os pinos do PIC são feitos.

metalgarurumon_reference.c - É o arquivo que mais vamos discutir ao longo da documentação, nele são declaradas a maioria das funções usadas ou não no programa (algumas funções tem mais de uma versão, como um processo evolutivo). As funções mais importantes como o tratamento de sensores, a leitura de sensores, a configuração, entre outros processos aparecem neste arquivo.

metalgarurumon_estrategias.c - As funções usadas no arquivo *metalgarurumon_reference.c* dependem muito de diferentes estratégias e devido ao fato desse assunto ser um estudo específico e separado do resto da rotina do programa (que meramente usa as estratégias no momento apropriado), adotou-se o uso de um arquivo apenas para configurar, declarar e trabalhar com funções de estratégia. O tópico de estratégias irá surgir mais a frente neste documento.

Configuração

Para que a gente possa entender a configuração, é preciso entender pelo menos superficialmente o Hardware que estamos trabalhando. Para informações mais profundas, use a referência **Anexo 2.** O Hardware que estamos trabalhando tem como esquemático a figura abaixo.

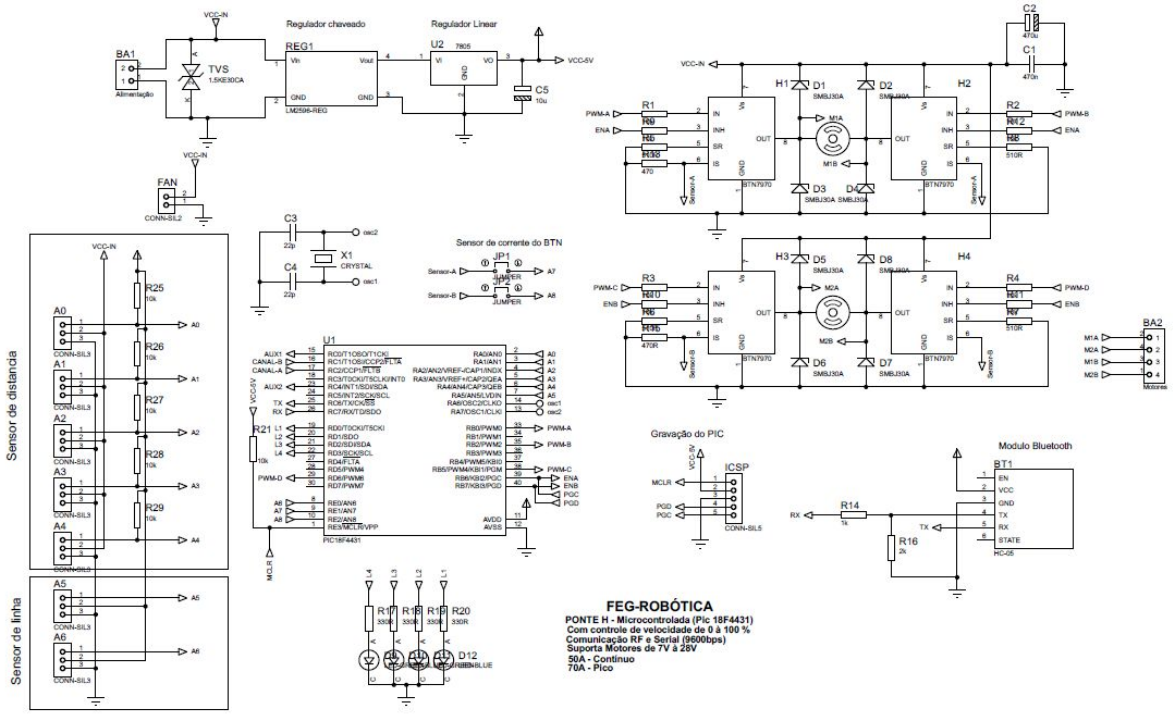

Figura 2. Esquemático Placa do robô MetalGarurumon

O funcionamento específico da eletrônica do robô é abordado na documentação da eletrônica.

```
void config()
{
   /*Definindo a direcao de cada uma das portas, A, B, C, D, E.
1 indica input e 0 output e a ordem funciona da seguinte maneira
e.g.

set_tris_a(a7,a6,a5,a4,a3,a2,a1,a0);

*/
   set_tris_a(0b11111111);
   set_tris_b(0b00000000);
   set_tris_c(0b10111111);
   set_tris_d(0b00000000);
   set_tris_e(0b1101);
```

Nessa parte do programa são definidas as direções dos pinos. A direção de um pino representa se ele irá funcionar como um pino de saída ou de entrada. A função **set_trix_x(int palavra_direcao)** recebe uma palavra (escrita em binário neste caso para facilitar o entendimento) onde 1 representa um pino de entrada (Input) e 0 um pino de saída (Output). Este trecho de código está de acordo com o **Anexo 1**.

```
//Configuracao dos modulos PWM do PIC18F2431
   setup_power_pwm_pins(PWM_BOTH_ON,PWM_BOTH_ON,PWM_BOTH_ON,PWM_BOTH_ON);
setup_power_pwm(PWM_FREE_RUN, 1, 0, POWER_PWM_PERIOD, 0, 1,30);

   /*
   FREE RUN eh o modo mais aconselhado para mover motores DC, outros modos podem controlar de
maneiras especificas, dividindo a frequencia
   o modo UP/DOWN aparentemente eh mais eficiente, eh preciso mais testes para garantir isso.

   1 indica o postscale usado na saida do PWM, o postscale divide a frequencia que recebe por um
nÃºmero, no caso nÃ£o estamos dividindo por nada

   0 indica o valor inicial da contagem do Timer responsavel pela geracao do pwm

   POWER_PWM_PERIOD eh o periodo do PWM, isso pode controlar a frequencia do PWM de acordo com o
periodo, ele eh dado em ciclos do clock

   0  o compare, ele compara esse valor com o valor do timer para verificar se algum evento
especial deveria acontecer

   1 o postscale compare,ele afeta o compare utilizado no parÃ¢metro anterior.

   30 o dead time, ele altera a diferenÃ§a de tempo entre o ON de um PWM e o OFF de seu
complementar

   */
```

Para o sentido e velocidade do movimento dos motores adotou-se pela eletrônica o uso dos pinos de Power PWM, eles são bastante configuráveis e conseguem produzir sinais PWM de frequências mais altas do que outras maneiras de tratar o PWM. Uma análise mais aprofundada sobre o assunto é tratada no **Anexo 3.** Aqui foi configurado todos os pinos como *BOTH_ON,* como os sinais são tratados em pares, ambos são configurados de maneira que de ambos os pinos sairá o mesmo sinal. De um ponto de vista de precaução, se um dos pinos queimar (fato que ocorreu na fase de testes) basta fazer um jumper do original para o pino par correspondente.

```
setup_adc_ports(sAN0 | sAN1 | sAN2 | sAN3 | sAN4 | sAN5 | sAN6);
//Define que todas as portas que possuem conversao A/D serao usadas como conversao A/D
  setup_adc(ADC_CLOCK_INTERNAL); //Utiliza o mesmo clock do PIC para o conversor A/D
```

```
   enable_motors(); //Controla a habilitacao das duas ponte H, para desabilitar uma delas, basta mudar o valor das constantes ENA e ENB
```

O código acima configura primeiramente as portas de conversar Analógico Digital. Elas são configuradas uma a uma afim de que no eventual caso de adicionar ou retirar algum sensor. A conversão A/D é uma operação dependente do tempo, por isso é importante dizer ao programa qual o Clock que será usado para fazer a operação, ou seja, qual a base de tempo para a operação. No caso usamos o próprio clock que é inserido no PIC que é de 20MHz (ao entrar no PIC ele se torna 5MHz). Ao final deste trecho de código a função *enable_motors();* é chamada com o único intuito de mandar um sinal alto em pinos específicos que "ligam" a Ponte H (eletrônica usada para controle dos motores).

```
enable_interrupts(global); //Habilita interrupções globais
   enable_interrupts(INT_TIMER1); //Habilita interrupção do Timer1
   setup_timer_1(T1_INTERNAL|T1_DIV_BY_8); //Configura Timer1 e uso o pre scaler para dividir por 8;

   disable_interrupts(INT_TIMER0);
   setup_timer_0(RTCC_INTERNAL|RTCC_DIV_2|RTCC_8_BIT); //Configura Timer1 e uso o pre scaler para dividir por 8;

   enable_interrupts(INT_TIMER5); //Habilita interrupção do Timer1
   setup_timer_5(T5_INTERNAL|T5_DIV_BY_1);//Configura Timer5 e divide o seu clock por 1.

   enable_interrupts(INT_RDA);
```

Nesse trecho de código são configuradas as interrupções. O fato de elas começarem habilitadas ou não é irrelevante, visto que em outra parte inicial do código elas são inicializadas de acordo. O Timer 1 é inicializado com uma divisão do clock, *prescale*, de 8. O Timer 0 é configurado da mesma maneira e também para que sua contagem seja de 8 bits, (portanto sofre overflow após contar até 256). O timer 5 é dividido por 5. A interrupção RDA representa uma interrupção acionada cada vez que um byte está pronto para ser lido na porta serial, ele é usado para ler um comando vindo do celular por Bluetooth. A tabela abaixo resume quais Timers são usados no código e quais suas funções.

Interrupção	Tempo de Overlow	Função
Timer 0	102.4 us	Conta o tempo de giro do robô
Timer 1	104.896 us	Conta o tempo que
Timer 5	0.2 us	Conta o tempo que o robô deve ficar atacando.
RDA	ND	Acionar uma função sempre que um caracter está pronto para ser lido na Serial

Tabela 2. Resumo das interrupções do programa

Programa Principal

O programa principal trata mais algumas configurações, como fuses e prioridade das interrupções e também é responsável pelo controle do acionamento do robô e suas funções gerais, quando acionado ou não.

```
#include<18f4431.h> //Adiciona biblioteca referente ao microcontrolador adequado(PIC15F4431)
#device adc=10 //Define a resolução do conversor A/D do microcontrolador
#fuses HS, NOWDT, NOPROTECT, NOBROWNOUT, NOPUT, NOLVP //Define algumas Configurações do PIC
#use delay(clock=20000000) //definição do cristal externo usado(na linha superior foi definido usar um
cristal rápido HS), em Hz, no caso 20Mhz

/*
HS - Significa que estamos usando um cristal de alta velocidade (no caso 20Mhz)
NOWDTH - Não desejamos nos usar do WatchDog Timer, que impede que o programa "trave" dentro de loops
infinitos ou algo assim. Nosso robô depende de um loop infinitos
NOPROTECT - O código não é protegido,portanto é possível acessar ele a partir do microcontrolador
NOLVP - Desabilita programação em baixa tensão.
*/
```

No código acima primeiramente incluimos a biblioteca principal do dispositivo. Depois definimos que o nosso conversor A/D (que vamos discutir mais a frente) é de 10 bits. Portanto ele converta 5v em um valor digital de 0 a 1024. Os fuses estão comentados acima e seguem um antigo confiável padrão de funcionamento e configuração dos fuses, as configurações de funcionamento do hardware. Na última linha do excerto é definido que o Clock que entra é de 20MHz(uma configuração da própria placa do robô).

```
//Configuração da comunicação rs232 usada para comunicar com o módulo Bluetooth
#use rs232(BAUD=9600,UART1, XMIT=PIN_C6, RCV=PIN_C7, PARITY= N, BITS = 8, STREAM = BT)
```

O método de acionamento do robô é por um dispositivo bluetooth ligado a porta serial, para tal funcionar configuração a nossa porta serial com Baud Rate (taxa de comunicação de bits) em 9600 bits por segundo, declaramos o parâmetro UART1 para dizer ao PIC que estamos usando o hardware da serial, em outros casos podemos simular apenas a comunicação serial e desta maneira não podemos usar sua interrupção, em seguida determinamos em quais pinos estão ligados os pinos de transmissão e recepção de informações. A paridade é uma ferramenta de comunicação para garantir que uma informação chegou a seu destino da mesma maneira que sua origem, neste caso não usamos esse recurso, é algo que pode ser melhorado em novas versões do software. O número de bits foi definido como 8 bits que é o mesmo número de bits de um char, um carácter que é o que enviamos do celular para o robô. Por fim, algumas funções requerem que você especifique qual porta serial você está tratando (PIC's podem ter múltiplas portas serial), pra isso foi dado um nome para essa porta, demos de BT.

Antes de discutir o código da interrupção RDA da Serial, que é onde começa o acionamento do robô, vamos observar dois diagramas, um mais geral sobre esse processo e outro apenas sobre a rotina de interrupção em si.

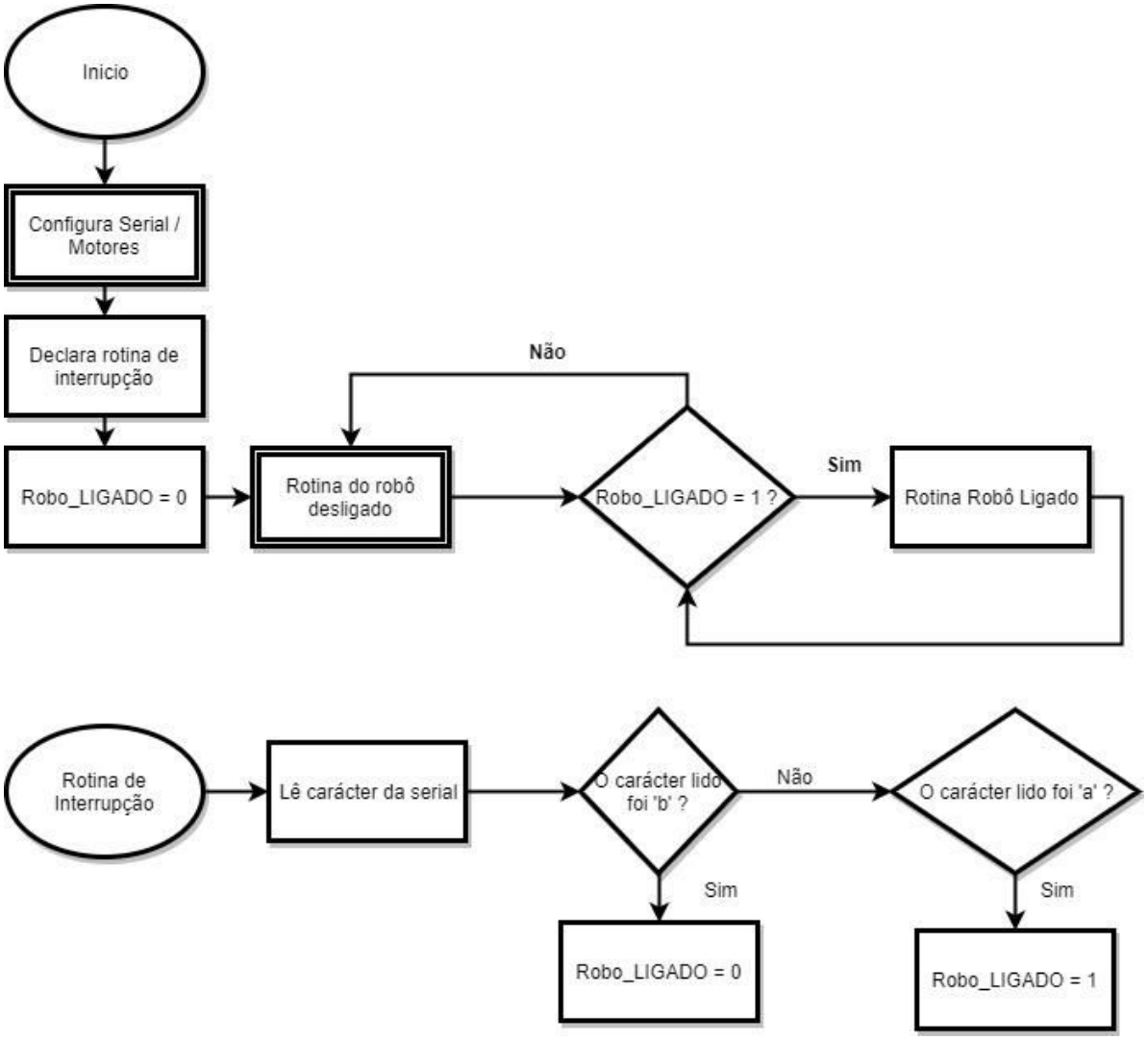

Figura 3. Diagrama do acionamento do robô

Basicamente, o robo começa com um variável que indica que ele está desligado. Mais a frente vamos discutir quais ações o robô toma quando está “desligado”. Enquanto isso o robô segue seu fluxo original de código e aguarda que a rotina de interrupção seja acionada, quando o módulo Bluetooth recebe um caracter.

Quando o programa entra na rotina de interrupção, ele verifica qual foi o caráter lido. Se o caracter for ‘b’ o robô desliga. Caso seja algo diferente, ele liga com algumas considerações já de estratégias e modos de operação. Ao fim de rotina de interrupção, o programa eventualmente vai verificar novamente o estado em que o robô se encontra e tomar a ação necessária. Desnecessário dizer, mas tudo isso ocorre de maneira bem rápida, o que garante a segurança entre ligar e desligar o robô. Do ponto de vista estratégico é bom que o robô ao ligar já receba a maior quantidade de informações possível, ao estudarmos como as estratégias são tratadas no software, isso ficará mais claro.

```
#INT_RDA
void RDA_isr(void)
{

  if(LIGA == 0)
  {
  leitura_bt = getc(BT);

  switch(leitura_bt)
  {
   case '0': LIGA = 1; primeiro_inicio = 1; busca_inicial = 0; busca_padrao = 0;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case '1': LIGA = 1; primeiro_inicio = 1; busca_inicial = 0; busca_padrao = 1;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case '2': LIGA = 1; primeiro_inicio = 1; busca_inicial = 0; busca_padrao = 2;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case '3': LIGA = 1; primeiro_inicio = 1; busca_inicial = 1; busca_padrao = 0;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case '4': LIGA = 1; primeiro_inicio = 1; busca_inicial = 1; busca_padrao = 1;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case '5': LIGA = 1; primeiro_inicio = 1; busca_inicial = 1; busca_padrao = 2;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case '6': LIGA = 1; primeiro_inicio = 1; busca_inicial = 2; busca_padrao = 0;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case '7': LIGA = 1; primeiro_inicio = 1; busca_inicial = 2; busca_padrao = 1;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case '8': LIGA = 1; primeiro_inicio = 1; busca_inicial = 2; busca_padrao = 2;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case '9': LIGA = 1; primeiro_inicio = 1; busca_inicial = 3; busca_padrao = 0;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case 'a': LIGA = 1; primeiro_inicio = 1; busca_inicial = 3; busca_padrao = 1;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case 'd': LIGA = 1; primeiro_inicio = 1; busca_inicial = 3; busca_padrao = 2;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case 'e': LIGA = 1; primeiro_inicio = 1; busca_inicial = 4; busca_padrao = 0;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case 'f': LIGA = 1; primeiro_inicio = 1; busca_inicial = 4; busca_padrao = 1;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
   case 'g': LIGA = 1; primeiro_inicio = 1; busca_inicial = 4; busca_padrao = 2;printf("\rSelecione o lado do
oponente:\r");lado = getc(BT); break;
  }
  }
  else
  {
   leitura_bt = getc(BT);
   if(leitura_bt == 'b')
   {
      LIGA = 0;
   }
```

```
  }

}
```

O código acima é a implementação da interrupção. Todas as implementações ocorrem no arquivo **main.c.** Cada um dos *case* trata uma entrada de estratégia diferente, note logo aqui que cada estratégia é uma variação de duas variáveis, a *busca_inicial* e a *busca_padrao.* Existem maneiras mais eficientes de realizar a mesma operação, usando menos *cases* , isso será implementado em uma versão nova do software. A função que lê o carácter da serial é a getc(*stream)*. Onde *stream* é o nome que demos a comunicação serial. Essa função já é bastante conhecida na linguagem C pura. Falaremos agora sobre como a variável *LIGA* é usada dentro do código para controlar o acionamento do robô em si.

```
while(true) //Ciclo principal do software
   {
    while(LIGA == 0)
    {

    if(primeiro_inicio == 0)
    {
      printf("\rDigite a string responsavel pela estrategia\r");
      primeiro_inicio = 1;
    }
    //Liga dois leds para indicar que o robô está ligado
    output_high(LED_0);
    output_low(LED_1);
    //Desliga os motores quando o robô é desligado via celular
    motor_1(0,'f');
    motor_2(0,'f');
    //Desabilita interrupções quando o robô não foi acionado
    disable_interrupts(INT_TIMER1);
    disable_interrupts(INT_TIMER0);
    disable_interrupts(INT_TIMER5);
    enable_interrupts(INT_RDA);
    enable_interrupts(global);

    //Reseta as variáveis de controle do overflow dos Timers
    overflow_timer1 = 0;
    overflow_timer0 = 0;
    cont_timer5 = 0;
    primeira_busca = 1;
    }
```

O trecho de código acima mostra um dos possíveis 3 estados da variável *LIGA* que é quando ela tem valor **0** , este é o valor em que a variável é inicializada, logo ao ser declarada, portanto este é o estado em que o robô se encontra desligado. Aqui, é enviada uma mensagem para o usuário, que irá visualizar no aplicativo de celular, indicando que o robô está aguardando uma mensagem. Para que a mensagem não fique sendo repetida o tempo todo, uma variável controla que ela apareça apenas uma vez ao iniciar o robô ou desativá-lo. Existem 4 Leds na placa, dois deles são usados para indicar se o robô está sob operação regular ou desligado. O comando *output_high* ou *output_low* manda um sinal positivo ou negativo, respectivamente, para um determinado pino do PIC, no caso o pino em que se encontra um led foi chamado de *LED_0* e outro *LED_1*, ao longo do código todos os pinos recebem um nome na biblioteca de referência *metalgarurumon_reference.h*. Em seguida todas as interrupções são desabilitadas, todos os contadores de *overflow* também são inicializados com **0.** Algo muito importante que ocorre aqui também é que os motores são desligados, uma necessidade de segurança muito importante. O funcionamento da função *motor_1* e *motor_2* será explicado mais a frente, mas o mais importante é que o primeiro parâmetro indica a velocidade em porcentagem do motor, portanto indicando o valor **0** lá, estamos para todos os efeitos parando o funcionamento do motor. (Ele continua sendo alimentado por tensão da bateria, portanto acredito que seria errôneo dizer que o motor está desligado).

```
while(LIGA == 1)
  {
     if(primeiro_inicio == 1)
        {
        printf("Estrategia Inicial:  \r");
        switch(busca_inicial)
        {
           case 0: printf("Busca Passiva\r"); break;
           case 1: printf("Busca Twister\r"); break;
           case 2: printf("Busca Arco\r"); break;
           case 3: printf("Busca Agressiva\r"); break;
           case 4: printf("Busca Pulso Movimento \r"); break;
           default: printf("Estrategia Invalida\r"); LIGA = 0; primeiro_inicio = 0; break;
        }
        printf("Estrategia Padrao:  \r");
        switch(busca_padrao)
        {
           case 0: printf("Busca Passiva\r"); break;
           case 1: printf("Busca Twister\r"); break;
           case 2: printf("Busca Agressiva\r"); break;
           default: printf("Estrategia Invalida\r"); LIGA = 0; primeiro_inicio = 0; break;
        }

        if(primeiro_inicio == 1) primeiro_inicio = 2;
```

```
        }
        LIGA = 2;
  }
```

A variável *LIGA* como foi observado anteriormente, é alterada dentro da interrupção. O momento em que *LIGA* recebe o valor **1.** É um momento intermediário entre o robô estar completamente desabilitado (*LIGA* com o valor **0**), e ele ligar. Neste estado o código verifica o valor das variáveis de estratégia e imprime para o usuário qual foi a variável selecionada. Por fim, *LIGA* recebe o valor de **2**, que efetivamente irá acionar o robô.

```
 while(LIGA == 2)
   {
        disable_interrupts(INT_TIMER0);
        disable_interrupts(INT_TIMER1);
        disable_interrupts(INT_TIMER5);

        if(primeiro_inicio == 2)
        {
        printf("\rBem vindo ao software do MetalGarurumon\r");
        set_timer0(0); //Inicializa o timer0
        set_timer1(0); //Inicializa o timer1
        set_timer5(0); //Inicializa o timer5
        primeiro_inicio = 0;
        printf("\r Lado: %c \r",lado);
        delay_ms(4700);
        }

        disable_interrupts(INT_TIMER0);
        disable_interrupts(INT_TIMER1);
        disable_interrupts(INT_TIMER5);
        enable_interrupts(INT_RDA);
        enable_interrupts(global);
        //Liga o Led que indica que o robô está ligado
        output_high(LED_1);
        output_low(LED_0);

        sensores();


} //While LIGA


   }
```

Quando a variável *LIGA* tem seu valor como 2 o robô entra em sua rotina competitiva. Uma mensagem de boas vindas é lançada para o usuário, os timers são , novamente, desabilitados por precaução, os leds que indicam que o robô está em operação regular são acionados e uma única função é chamada, a função *sensores()*, vamos começar a discutir as principais funções do robô a partir dela. A ideia é que o programa principal pudesse tratar do robô em um nível macro, sua inicialização, timers, leds indicativos e apenas uma função que chamasse todas as outras necessárias no decorrer do programa. Este é o fim da descrição do programa principal, as ações mais importantes no que corresponde o quanto o robô foi efetivo competitiva, serão apresentadas a seguir quando falamos sobre a maior arquivo escrito no projeto, o *metalgarurumon_reference.c*

metalgarurumon_reference.c

Este arquivo contém as principais funções usadas no programa. Vamos partir de funções mais gerais para chegarmos em funções mais específicas, as estratégias empregadas no robô que surgem nesta parte do firmware, serão estudadas em um tópico específico separado, ao final da documentação.

```
void sensores()
{
          //faz a leitura do canal adc indicado e poe a resposta no bit indicado da palavra
indicada (Sensores de linha)
         resposta_sensores = leitura_adc(750,canal_sensor_linha_1, resposta_sensores, 0);
//Sensor Direito
         resposta_sensores = leitura_adc(750,canal_sensor_linha_2, resposta_sensores, 1);
//Sensor Esquerdo

         //(Sensores de distancia)

         resposta_sensores = leitura_adc(320,canal_sensor_distancia_direita,
resposta_sensores, 2);
         resposta_sensores = leitura_adc(500,canal_sensor_distancia_frente_direita,
resposta_sensores, 3);
         resposta_sensores = leitura_adc(500,canal_sensor_distancia_frente_esquerda,
resposta_sensores, 4);
         resposta_sensores = leitura_adc(500,canal_sensor_distancia_esquerda,
resposta_sensores, 5);

         trata_sensores_2(); //responde com o motor em relacao aos sensores usando uma
maquina de estados
}
```

O trecho acima de código dá um vislumbre em como os sensores são tratados no código. Uma função de leitura dos sensores, que generalizando podemos dizer que é uma função para ler uma porta de conversão analógico-digital; é chamada ***n*** vezes, sendo ***n*** o número de sensores usados no robô. Dizemos a função qual seria o valor médio da leitura, o valor cujo qualquer valor acima dele é considerado que o sensor foi acionado e abaixo o contrário, o pino em que

este sensor está conectado, a variável que irá receber o resultado desta comparação e em *bit* da palavra isso será guardado. No nosso caso a variável se chama *resposta_sensores* e ela é formada da seguinte maneira.

Figura 4. Estrutura da variável que armazena o resultado dos sensores (resposta_sensores)

Desta maneira, ao fim da execução desta função, haverá uma variável que indica exatamente qual o estado de todos os sensores como acionado ou não, 0 ou 1. A função que é chamada a seguir *trata_sensores2()* lida justamente com a análise de cada parte desta variável de 8 bits. Antes de partirmos para a sua análise que demanda atenção; vamos antes falar sobre a leitura de um canal A/D

```
int8 leitura_adc(int16 threshold, int canal, int8 resposta_leitura, int bit)
{

   switch(canal) //switch para preparar o canal desejado para a conversao A/D
   {
      case 0: set_adc_channel(0); break;
      case 1: set_adc_channel(1); break;
      case 2: set_adc_channel(2); break;
      case 3: set_adc_channel(3); break;
      case 4: set_adc_channel(4); break;
      case 5: set_adc_channel(5); break;
      case 6: set_adc_channel(6); break;
   }

   delay_us(10);            //delay de 10 us (valor suscetivel a erro devido ao uso da
interrupÃ§Ã£o do Timer1), importante ser um valor pequeno
   leitura = read_adc();    //Realiza a leitura do canal analÃ³gico-digital
  /*
   if(canal == canal_sensor_linha_1)  printf("*G%Lu*",leitura);
   else if(canal == canal_sensor_linha_2) printf("*P%Lu*",leitura); */

   if (leitura > threshold) bit_clear(resposta_leitura,bit); //Caso o sensor retorne 0, a
palavra recebe 0 em seu primeiro bit
   else bit_set(resposta_leitura,bit); //Caso sensor retorne 1 , a palavra recebe 1 em seu
primeiro bit


   return resposta_leitura; //devolve a variavel com seus bits alterados

}
```

Um dos parâmetros da função é o canal que será lido, então este valor passa por um switch que aciona a função *set_adc_channel* que prepara o canal para ser lido. Como recomendação da própria Microchip, existe um atraso, um delay de 10 us antes de realizar a leitura. A leitura então é efetivamente realizada pela função *read_adc()* que joga este valor para uma variável local chamada de *leitura*. No código acima existe um trecho comentado que mostra o valor lido

no aplicativo de celular, num formato que transforma o que foi lido em um gráfico para melhor visualização. O valor da leitura é então comparado com o limiar, que foi um parâmetro da função. Por fim escreve **0** no bit indicado da palavra caso o valor da leitura seja menor que o limiar e **1** caso contrário. Para mudar apenas um bit de uma variável foram usadas as funções: *bit_clear* e *bit_set(variavel a ser alterada, bit da variavel a ser alterado)*.

Vamos direcionar agora a nossa atenção duas funções complementares (portanto apenas o excerto de código de uma delas aparecerá aqui), que movimentam o robô.

```
void motor_1(int duty_cycle, char sentido)
{
   if(sentido == 'f')
   {
   //gira o motor para frente com duty_cycle% de sua potencia total
      set_power_pwm0_duty((int16)((POWER_PWM_PERIOD *4) * (duty_cycle*0.01)));
      set_power_pwm2_duty((int16)(0));

       /*Aqui multiplica-se por 4 pois o maior valor do periodo do PWM eh 4095 e o valor mÃ¡ximo
do parametro da funcao,
      que daria 100% de duty cycle Ã© 16383, portanto cria-se assim uma proporcao entre o periodo
e a velocidade multiplicando por um numero entre 0 e 1
      que seria a porcentagem desejada.
      */
   }

   if(sentido == 'b')
   {
     //gira o motor para trÃ¡s com duty_cyle% de sua potencia total
      set_power_pwm0_duty((int16)(0));
      set_power_pwm2_duty((int16)((POWER_PWM_PERIOD *4) * (duty_cycle*0.01)));
    /*Aqui multiplica-se por 4 pois o maior valor do periodo do PWM eh 4095 e o valor mÃ¡ximo do
parametro da funcao,
      que daria 100% de duty cycle Ã© 16383, portanto cria-se assim uma proporcao entre o periodo
e a velocidade multiplicando por um numero entre 0 e 1
      que seria a porcentagem desejada.
      */


   }
}
```

Com o Power PWM devidamente configurado, a função *motor_1* é usada para efetivamente mover o motor. Cada motor é movido separadamente, portanto existe uma função complementar chamada de *motor_2*. A ponte H projetada pela equipe tem para cada motor dois “lados”, para que o motor se mova em um determinado sentido, é preciso que em um lado da ponteH que o controla seja enviado um sinal de PWM, e no outro lado seja constantemente enviado um sinal **0.** A função *set_power_pwmX_duty()* define qual vai ser a ciclo de trabalho da sua função PWM em um canal de Power PWM. O parâmetro que ela recebe vai de 0 até um valor 4 vezes o valor do período do seu PWM (dado em ciclos de trabalho do PIC). No caso,

este valor foi mais uma vez multiplicado por um valor entre 0 e 1 para que o programador pudesse facilmente adotar uma porcentagem de velocidade (que é proporcional ao ciclo de trabalho do PWM que é exercido sobre ele), do motor. Um outro parâmetro desta função é o lado. Caso este valor seja **'f'** então o motor se locomove num sentido, se for **'b'** se move em outro sentido. Os sentidos são ajustados de acordo com a montagem do robô e principalmente os fios do motor para a placa.

Podemos finalmente abordar a função *trata_sensores2()* importante notar que o "2" no final do seu nome se deve ao fato de que inicialmente o robô teria seus sensores tratados de uma determinada maneira, mas esta maneira se provou pouco eficiente nos testes, ele não encontrava sempre o robô e constantemente era pego em pontos cegos de leitura. Com esta mudança o robô não só competiu em alto nível como não errou o alvo sequer uma vez durante o campeonato.
Esta presente versão da função de tratamento do sinal dos sensores se baseia numa máquina de estados cujo diagrama é apresentado abaixo

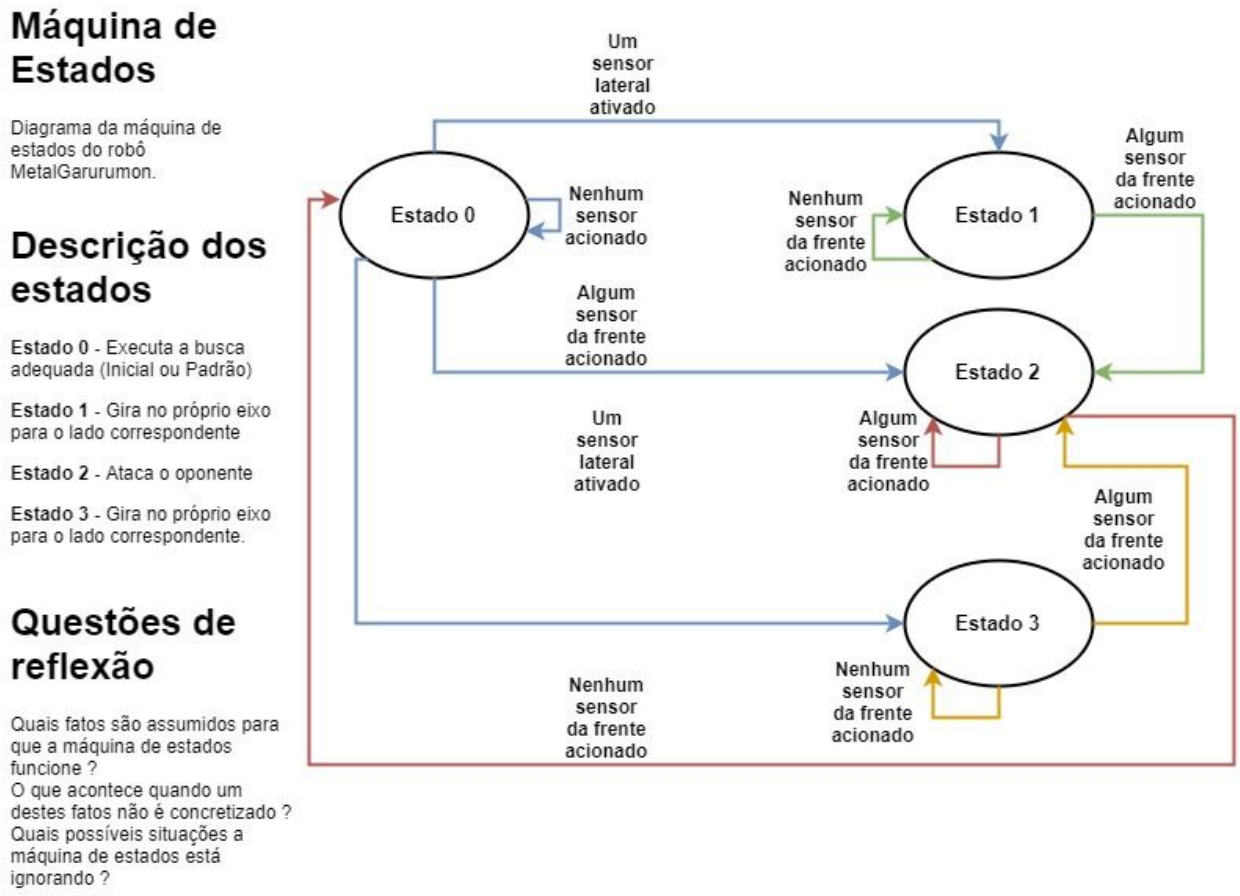

Figura 5. Máquina de estados do tratamento do sensor.

A própria figura lança alguns questionamentos, que a explicação sobre a sua implementação irá responder. Antes de apresentar em detalhes o código de cada estado, é apresentado abaixo o código que lida com a linha. Como se manter no dojô é a maior prioridade do desafio, os sensores de linha são sempre tratados e antes de todos os outros sensores.

```
 void trata_sensores_2() //Trata a resposta aos sensores usando uma espécie de maquina de
estados
{

 /*
     Diagramação dos bits da palavra resposta_sensores

     bit 0 - Sensor Linha Direita
     bit 1 - Sensor Linha Esquerda
     bit 2 - Sensor Lateral Direita
     bit 3 - Sensor Frente Direita
     bit 4 - Sensor Frente Esquerda
     bit 5 - Sensor Lateral Esquerda

     */
      if ( bit_test(resposta_sensores,0) ) //Verifica isoladamente os sensores de linha
   {

      motor_1(0,'b');
      motor_2(0,'b');
      delay_ms(200);

      //Anda para trás
      motor_1(VELOCIDADE_LINHA,'b');
      motor_2(VELOCIDADE_LINHA,'b');
      delay_ms(TEMPO_LINHA_TRAS_MS);

      //Para o motor rapidamen


      //Gira o robô
      motor_1(VELOCIDADE_GIRO_LINHA,'b');
      motor_2(VELOCIDADE_GIRO_LINHA,'f');
      delay_ms(TEMPO_GIRO_LINHA);

      estado_sensores = 0;
      mover = 0;

      bit_clear(resposta_sensores,0);

   }
```

Existe um outro **if** que lida com o outro sensor de linha, a função é complementar. Primeiro é verificado se o bit que representa o sensor de linha tem valor **1**, com a função *bit_test*. Caso isso seja verdadeiro, é preciso se esquivar da linha. Aqui entra um caso bastante interessante. A princípio o código foi programado para que o robô parasse na linha, e então fosse para trás, e por fim girasse. Todavia, devido a alta inércia que o robô proporciona, o robô acabava

escapando do dojô, limitando a velocidade de trabalho de várias estratégias de busca, prejudicando muito o desempenho competitivo. Foi decidido então que para superar essa inércia, o primeiro comando de reação a uma linha no dojô seria de ir para trás imediatamente. Isso, em testes iniciais se provou uma estratégia eficiente e aumentou a velocidade com que o robô conseguia escapar da linha, chegando quase a 100% de sua velocidade máxima. Todavia, o ato de imediatamente mudar o sentido de rotação do robô causava uma corrente de pico muito alta e eventualmente o robô parava de funcionar, porque o transistor que o controlava, cortava, parando o motor até que o mesmo fosse propriamente resfriado. Como solução intermediária, foi adotado um tempo menor de parada, e isso nos permitiu alcançar um velocidade maior que a primeira tentativa. Voltando a análise do código acima, é importante notar que a velocidade com que o robô volta para trás e que ele gira são definidas por constantes, isso se provou muito mais eficiente e prático para fazer mudanças no código. Um problema desse trecho do código é que o giro final feito por delay impede que o robô busque um adversário enquanto gira, uma versão mais recente do código deve corrigir isso. Outra coisa importante a ser notada é que o estado da máquina de estados é automaticamente ressetado como **0**, que é o estado que indica a busca mais geral, quando se assume que nenhum robô foi visto. A variável *mover* é uma variável de controle que será abordada em detalhes na implementação das estratégias, mas que controla se o robô deve fazer um pulso de movimento, ela é forçada a **0** para impedir que uma movimentação desnecessária ocorra neste trecho do código. É comum que esse tipo de coisa aconteça ao longo do código, de variáveis serem ressetadas, sem ficar muito claro se elas precisavam ou não ser ressetadas lá, entretanto é difícil prever exatamente o valor de cada variável em cada momento do código, portanto essas medidas são adotadas por precaução e aceleram o desenvolvimento do código. Partiremos agora para a análise do primeiro estado, o estado **0.**

```
if(estado_sensores == 0) //Estado em que nenhum sensor foi visto, estado inicial do robo
   {
   disable_interrupts(INT_TIMER5);
   disable_interrupts(INT_TIMER0);
      if(!primeira_busca)
      {
      //Executa estrategia padrao para localizacao do adversario em qualquer ponto da
partida depois de ter achado uma vez

        switch(busca_padrao)
      {
         case 0: busca_passiva(); break;
         case 1: busca_twister(lado); break;
         case 2: busca_agressiva(); break;
      }

      }
      else{

        switch(busca_inicial)
      {
         case 0: busca_passiva(); break;
         case 1: busca_twister(lado); break;
```

```
            case 2: busca_arco(lado); break;
            case 3: busca_agressiva(); break;
            case 4: busca_pulso(); break;
        }

        }

        if(mover) //Executa um pequeno movimento caso tenha dado overflows o suficiente
        {
            motor_1(50,'f');
            motor_2(50,'f');
            delay_ms(50);
        //  printf("Moveu\n");
            mover = 0;
        }


        switch(resposta_sensores)
        {
        case 0b00000000: estado_sensores = 0; break;
        case 0b00000100: estado_sensores = 1; break;
        case 0b00001100: estado_sensores = 2; break;
        case 0b00011000: estado_sensores = 2; break;
        case 0b00001000: estado_sensores = 2; break;
        case 0b00100000: estado_sensores = 3; break;
        case 0b00110000: estado_sensores = 2; break;
        case 0b00010000: estado_sensores = 2; break;

        default: estado_sensores = 0; break;
        }
    }
```

Esse é o estado em que o robô é iniciado, portanto é assumido que neste estado ele não está vendo nenhum adversário, é aqui que as buscas de adversário ocorrem. A busca do adversário se divide em duas etapas. Caso seja a primeira vez que ele tenha chegado a esse estado e nunca tenha visto um adversário, ele verifica qual foi a *busca_inicial* definida pela estratégia, lá na interrupção RDA, de acordo com seu valor, ele chama uma função diferente, cada uma representando uma diferente maneira de encontrar o robô na primeira tentativa. As estratégias são separadas assim porque no início da partida nós temos um claro conhecimento sobre a posição do robô e podemos assim nos arriscar mais em estratégias mais específicas. Para o caso de não ser o começo da partida, realizamos um *switch* também, mas para a estratégia padrão. O conteúdo de cada função de estratégia será abordado mais na frente no documento. Fora disso tudo, existe um **if** que verifica a condição da variável *mover*. Existe uma estratégia que consiste em deixar o robô parado e se mover em pequenos pulsos de movimento a cada X segundos. Essa ação é controlada por uma interrupção, quando a interrupção é acionada um determinado número de vezes, a variável assume um valor não nulo e então o robô se move por alguns instante apenas e para.

Após lidar com as estratégias, a palavra é verificada e o software assume um novo estado (que pode ser o próprio estado atual), de acordo com o valor da variável *resposta_sensores*. Se ele não houver acionado algum sensor de distância, ele permanece no mesmo estado. Caso o

sensor direito lateral tenha sido acionado ele vai para o estado **1**, caso o sensor esquerdo lateral ele vai para o estado **3**. Caso algum sensor frontal tenha sido acionado ele então vai para o estado **2.** Analisamos então cada um destes casos. O caso 1 e 3 são complementares, portanto serão explicados apenas uma vez.

```
if(estado_sensores == 1)
   {
      motor_1(50,'f');
      motor_2(50,'b');
      primeira_busca = 0;
      disable_interrupts(INT_TIMER5);
      disable_interrupts(INT_TIMER1); //Timer para estratégia inicial de busca
      enable_interrupts(INT_TIMER0); //Timer para fazer volta em torno do proprio eixo
     // printf("Estado Atual: %d \r",estado_sensores);
      if(estado_sensores == 0) goto fim1;
      if(estado_sensores == 0) goto fim1;
      if(estado_sensores == 0) goto fim1;
      if(estado_sensores == 0) goto fim1;
      if(estado_sensores == 0) goto fim1;
      if(estado_sensores == 0) goto fim1;
      if(estado_sensores == 0) goto fim1;
      if(estado_sensores == 0) goto fim1;
      if(estado_sensores == 0) goto fim1;
      if(estado_sensores == 0) goto fim1;

      switch(resposta_sensores)
      {
      case 0b00011000: estado_sensores = 2; break;
      default: estado_sensores = 1;  break;
      }
      fim1:
      estado_sensores = estado_sensores;
   }
```

O estado 1 gerava um certo problema na primeira vez que foi programado, assumia-se que quando um sensor lateral era visto, um robô estava ao lado do robô e que com um movimento para o correspondente lado acharia rapidamente o robô, isso mostrou-se falho e constantemente o robô ficava girando repetidamente. Para evitar isso, adotou-se um timer que é acionado sempre que o robô entra nesse estado. Quando passe-se uma quantidade de tempo suficiente para fazer um giro de aproximadamente 80 graus, a interrupção força a variável *estado_sensores* a assumir o valor **0.** A ideia era que nesse estado, o único outro estado possível seria o estado **2** que é quando o robô vai atacar o adversário ou o próprio estado, então ao forçar o estado **0** o robô daria apenas um giro curto no máximo e logo em seguida iria seguir em uma rotina de busca, tornando-o mais dinâmico. O problema que essa solução gerou foi que é difícil prever em que linha do código a interrupção será acionada, e assim muitas vezes a variável *estado_sensores* recebia o valor **0**, porém novamente era forçada ao valor **1** e então ficava novamente girando. A solução apresentada acima é assumidamente ruim e ineficiente, uma versão mais recente do firmware corrige isso. Nesta

versão , para solucionar isso a linha que verificava se a interrupção havia forçado o valor **0** na variável de estado aparece várias vezes para que o robô passe mais tempo lá do que no resto do código, isso funcionou melhor, mas ainda claramente não era a melhor solução. E para evitar que ela fosse logo em seguida ressetada para **1** o programa usa a função **goto** para pular as linhas que poderiam forçar a variável a **1** novamente.

```
if(estado_sensores == 2)
   {
      disable_interrupts(INT_TIMER5); //Aciona Timer para ataque cronometrado
      //Move ambos motores para o araque
      motor_1(VELOCIDADE_ATAQUE,'f');
      motor_2(VELOCIDADE_ATAQUE,'f');
      //Indica que o robo ja achou um adversario uma vez pelo menos
      primeira_busca = 0;
      //Desabilita interrupções de giro do motor e de pulso de movimento
      disable_interrupts(INT_TIMER1);
      disable_interrupts(INT_TIMER0);
      //Zera contagem de overflows do timer0
      overflow_timer0 = 0;

      switch(resposta_sensores)
      {
      case 0b00011000: estado_sensores = 2; break;
      case 0b00010000: estado_sensores = 2; break;
      case 0b00001000: estado_sensores = 2; break;
      case 0b11111110: evasao(); estado_sensores = 0; resposta_sensores = 0b00000000; break;

      default: estado_sensores = 0; break;

      }

   }
```

Acima encontra-se o excerto de código de ataque, novamente observa-se que o parâmetro de velocidade do robô é colocada como uma constante pelo mesmo motivo anteriormente apresentado no tratamento do sensor de linha. Aqui usamos o Timer5 para controlar quanto tempo o robô passa em um ataque, caso ele passe tempo demais atacando (caso em que esteja enfrentando robôs muito mais fortes), ele aplica uma técnica de evasão, que se provou muito eficiente, mas que será melhorada em uma nova versão do código. No código acima, quando o Timer5 é acionado um determinado número de vezes, ele força o valor **0b1111110** na variável *resposta_sensores* levando o *switch* a forçar a variável de estado ao valor **0** e chamar a função de evasão. Tudo que a função de evasão faz é ir para trás por um tempo. Depois a leitura dos sensores é forçada a dizer que não achou nenhum sensor para forçar o robô a encontrar novamente o robô e talvez se reposicionar.

Estratégias

Visando obter resultados mais significativos na competição, notou-se necessário um estudo interessante sobre as diferentes estratégias a serem aplicadas na competição de sumô de robos. Primeiramente como dito anteriormente, foram adotadas estratégias de busca inicial e de uma busca padrão. As de busca inicial são as mais importantes, pois queremos acabar rapidamente com a partida e ser o robô que encontra primeiro seu adversário.

Busca Passiva

```
void busca_passiva()
{
         //Para os dois motores
         motor_1(0,'b');
         motor_2(0,'f');
         //Forca o motor a nao dar um pulso de movimento
         mover = 0;
         //desabilita o timer responsavel por impedir que o robo fique num mesmo estado
muito tempo
         disable_interrupts(INT_TIMER0);
         disable_interrupts(INT_TIMER5);
         //reseta o contador do timer0
         overflow_timer0 = 0;
}
```

A busca passiva é uma busca usada apenas para fins de teste, tudo que ela faz é manter o robô parado, é um modo eficiente de observar as reações do robôs aos sensores, sem ter que lidar com a constante movimentação do robô. O resto da função zera alguns parâmetros e desabilita algumas interrupções.

Busca Twister

```
void busca_twister(char lado)
{
      //Gira de acordo com o lado mais proximo do robo adversario
        if(lado == 'e')
        {
        motor_1(50,'b');
        motor_2(50,'f');
        }
        else if(lado == 'd')
        {
        motor_1(50,'f');
        motor_2(50,'b');
        }

         //Forca o motor a nao dar um pulso de movimento
         mover = 0;
         //desabilita o timer responsavel por impedir que o robo fique num mesmo estado
muito tempo
         disable_interrupts(INT_TIMER0);
         disable_interrupts(INT_TIMER5);
         //reseta o contador do timer0
         overflow_timer0 = 0;
}
```

Essa é uma busca pouco usada. Ela gira em torno do próprio eixo. Considerando que o adversário estará presente em cerca de 30 graus de um movimento circular completo, existem 330 graus desperdiçados pela busca, isso a torna pouco eficiente. O código faz uma verificação de lado para ele considerar o lado em que o adversário é observado antes do início da partida. Economizando assim um tempo.

Busca em Arco

```
void busca_arco(char lado)
{
      //Gira de acordo com o lado mais proximo do adversario
      if(lado == 'e')
      {
      motor_1(30,'f');
      motor_2(60,'f');
      }
      else if (lado == 'd')
      {
      motor_1(60,'f');
      motor_2(30,'f');
      }

         //Forca o motor a nao dar um pulso de movimento
         mover = 0;
         //desabilita o timer responsavel por impedir que o robo fique num mesmo estado
muito tempo
         disable_interrupts(INT_TIMER0);
         disable_interrupts(INT_TIMER5);
         //reseta o contador do timer0
         overflow_timer0 = 0;
}
```

Essa foi a busca mais usada no torneio. O robô é posicionado de acordo para que um movimento em arco consiga atingir o adversário em sua lateral, garantindo que ele não irá conseguir esboçar muita reação. Para tal, primeiramente ele verifica o lado ideal do arco e o movimento em si se dá por uma diferença de velocidade entre as rodas. Pode ser que movimento retilíneos com ambas as rodas em velocidades maiores seja mais eficaz do que a busca em arco, isso será modificado em um código posterior.

Busca Agressiva

```
void busca_agressiva()
{
      //Para os dois motores
         motor_1(VELOCIDADE_BUSCA_AGRESSIVA,'f');
         motor_2(VELOCIDADE_BUSCA_AGRESSIVA,'f');
         //Forca o motor a nao dar um pulso de movimento
         mover = 0;
         //desabilita o timer responsavel por impedir que o robo fique num mesmo estado
muito tempo
         disable_interrupts(INT_TIMER0);
         disable_interrupts(INT_TIMER5);
         //reseta o contador do timer0
         overflow_timer0 = 0;
}
```

Essa é a busca mais simples, porém é bastante eficiente. Ela ataca em linha reta e devido ao fato do tratamento do sensor de linha proporcionar um giro no robô, o mesmo acaba fazendo uma rápida busca em estrela pelo dojô.

Conclusão

O robô MetalGarurumon conseguiu obter o terceiro lugar no Winter Challenge de 2017 com o código que foi aqui descrito. Alguns pontos importantes de melhoria é o uso de diferentes estratégias mais eficientes, o uso de pré estratégias que teriam como propósito reposicionar o robô logo antes de começar a buscar o adversário. Tirar todos os **goto**. Fazer com que o giro ao identificar uma linha também busca um adversário. Mesmo com todos esses pontos a melhorar, o código foi bastante eficiente, encontrou todos os seus adversários primeiro, não errou nenhuma vez seu alvo e não saiu de nenhuma luta sem vencer pelo menos um round. Garantindo com tudo isso a classificação para o INTERNATIONAL ALL JAPAN ROBOT SUMO TOURNAMENT em dezembro de 2017.