# Documentação CPU : Sistema de Monitoramento de Motor DC controlado por PWM

Caio Turnes Silvestri

Engenharia Eletrônica
Turma: 08235
Matrícula: 19200578
caiots11@gmail.com

## 1 Diagrama de Classes

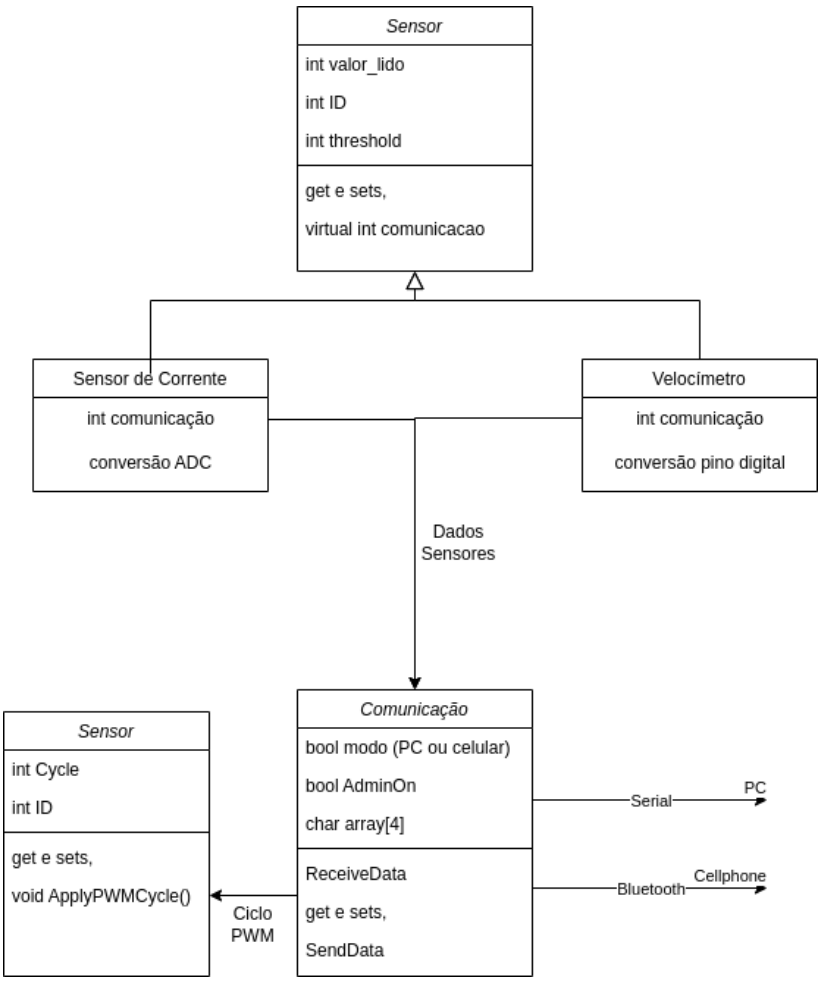


Figure 1: Diagrama de Classes Microcontrolador

Na Figura 1 temos o Diagrama de Classes do Microcontrolador atualizada, que até este ponto ficou mais simplificado. A mudança mais impactante foi a remoção do acelerômetro, devido a maior dificuldade de simulá-lo, por precisar fazer comunicação I2C, resolvi tirá-lo do projeto. Outra mudança é que a transmissão dos dados dos sensores não se dá mais pelos friends estipulados. E não foi implementado polimorfismo para ler os dados de tensão nos sensores com a outra classe virtual de sensor inicialmente estipulada. Isto aconteceu pois são implementações mais complexas e que optei por não utilizar por enquanto até que o projeto esteja funcional.

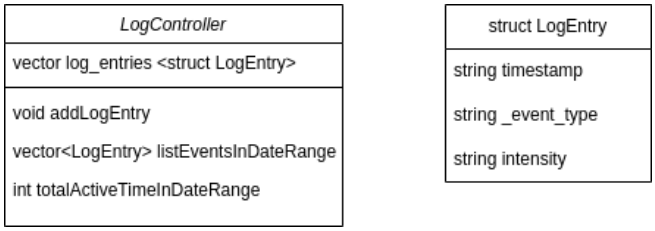


Figure 2: Diagrama de Classes CPU

Na Figura 2 temos o Diagrama de Classes da CPU, ela possui apenas uma classe que tem como função fazer toda a interação com a fila transmitida pelo microcontrolador. Tendo as funções de: Adicionar uma dado novo no vector "log entries". Listar os Eventos dentro de um intervalo de tempo. E mostrar o tempo em que o controlador esteve ativo também durante um intervalo de tempo. Cada dado deste vetor é composto de 3 strings: o tempo em que ocorreu o evento, o tipo de evento, e a intensidade do evento.

O Diagrama de classes do celular seguirá uma estrutura parecida, visto que ambos tem as mesmas funções

Essas funções funcionam devido a estrutura em que os dados estão sendo transmitidos, e o protocolo criado para que possam se comunicar adequadamente. Este protocolo será explicado a seguir

# 2 Protocolo e Comunicação

O Protocolo gira em torno da string de tipo de evento que está condito dos dados. Abaixo temos uma tabela o que significa cada tipo de evento.

| Tipos de Evento | |
|---|---|
| Evento | Significado |
| A | Sinaliza microcontrolador que admin conectou |
| D | Sinaliza microcontrolador que admin desconectou |
| P | Modo Comunicação com PC |
| C | Modo Comunicação com celular |
| M | Altera duty-cycle do PWM |
| I | Sinaliza admin que microcontrolador ligou |
| F | Sinaliza admin que microcontrolador desligou |

A Comunicação Serial é feita transmitindo 1 string de tamanho fixo, que é divida posteriormente pela cpu nas 3 strings do vector da classe LogController.

O modelo desta string é a seguinte: "ano-mes-dia minutos:segundos-tipo do evento-intensidade".

Para a comunicação Serial optei por uma baudrate de 9600 bits/s, 8 bits de dados, 1 stop bit, 0 start bit e sem bit de paridade.

# 3 Rotina da CPU e do celular

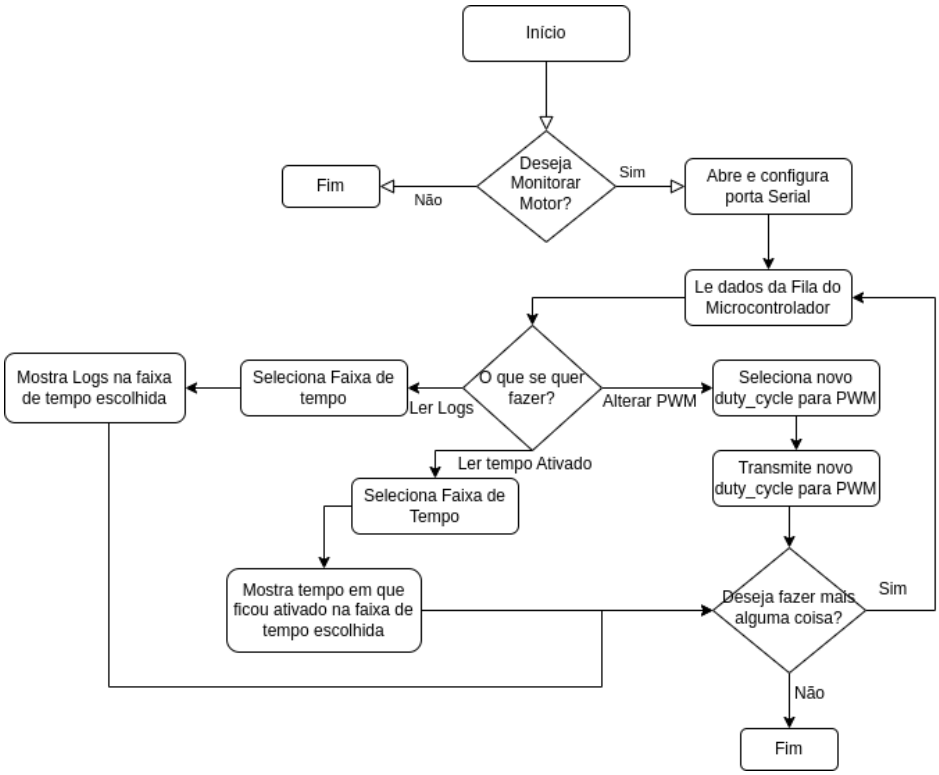


Figure 3: Rotina CPU

# 4 Algoritimos com estruturas de dados

Para a CPU, a variável que armazenava os strings vindos do microcontrolador é uma variável de tipo vector de structs composto por 3 strings.

```
struct LogEntry {
    string timestamp; //Tempo que ocorreu
    string event_type; // Tipo de Evento
    string intensity;//Valor Do Evento
};

vector<LogEntry> log_entries
```

A função de adicionar valores ao vector é a seguinte:

```
void addLogEntry(const string& timestamp, const string& event_type, const
string& intensity) {
        log_entries.push_back({timestamp, event_type, intensity});
    }
```

Vemos que ela necessita das 3 strings já separadas para adiciona-las ao log.
Para isso Dividimos a string transmitida da seguinte forma:

```
buf.copy(buf_time,16,0);
        buf_time[16] = '\0';

        buf.copy(buf_event,1,17);
        buf_event[1] = '\0';

        buf.copy(buf_intensity,3,19);
        buf_intensity[3] = '\0';

        logController.addLogEntry(buf_time, buf_event, buf_intensity);
```

Como o padrão de transmissão é sempre o mesmo, podemos separar copiando sempre as mesmas posições da string maior para as strings menores.

A seguir serão copiadas as funções de Verificar os eventos ocorridos no intervalo de tempo, o tempo que o microncontrolador permaneceu ativo, e o controle do PWM. Como estão todos comentados, não vou descrever o código aqui.

```
//Lista os eventos no intervalo de tempo desejado
vector<LogEntry> listEventsInDateRange(const string& start_date, const string& end_date) {
    vector<LogEntry> filtered_entries;
    for (const auto& entry : log_entries) {
        //laço for que acessa todos os log_entries do vetor de cada vez
        //Limita a apenas os eventos dentro da faixa de tempo observando os time-stamps
        if (entry.timestamp >= start_date && entry.timestamp <= end_date) {
            filtered_entries.push_back(entry);
            //Adiciona dados da log_entries ao filtered_entries que estão dentro da faixa de tempo
        }
    }
return filtered_entries; //Retorna Vector com os eventos do intervalo de tempo
}
```

Figure 4: listEvents Algorithm

```cpp
//Função que verifica quanto tempo o controlador ficou ligado
int totalActiveTimeInDateRange(const  string& start_date, const  string& end_date) {
   int total_minutes = 0;
   int maquina_estado = 0;
   int ano,mes,dia,hora,min,seg;
   int nano,nmes,ndia,nhora,nmin;
   int dano =0,dmes =0,ddia =0,dhora =0,dmin =0;
   string last_event = "S"; //Inicia com ultimo Evento em S de start, para caso no inicio da faixa de tempo o controlador já estivesse ligado
   sscanf(start_date.c_str(), "%d-%d-%d %d:%d",&ano,&mes, &dia, &hora, &min); //Converte a string dos dados em variáveis inteiras

   for (const auto& entry : log_entries) { //Laço for que passa por todos os eventos da log_entries
      if (entry.timestamp >= start_date && entry.timestamp <= end_date) { //Limita para os eventos dentro da faixa de tempo
         //O mic vai mandar "I" e o horário que ativou o controlador e F e o horário que desligou,
         //Se ler um F antes de um I, é porque já estava ligado desde o início do start_date,
         //Mesma coisa serve para caso acabe com um I
            if (entry.event_type == "I" && (last_event == "F" || last_event == "S" )) {
               // Se foi lido como uma ativação (I) e o ultimo evento foi desativação ou é o primeiro evento de atuação
               sscanf(entry.timestamp.c_str(), "%d-%d-%d %d:%d",&ano,&mes, &dia, &hora, &min);//Nova referencai para o horário de ativação
               last_event = "I"; //Indica que o ultimo evento foi uma atuação
            }
            else if (entry.event_type == "F"&& (last_event == "I" || last_event == "S" )){
               // Se foi lido como uma desativação (I) e o ultimo evento foi ativação ou é o primeiro evento de atuação
               sscanf(entry.timestamp.c_str(), "%d-%d-%d %d:%d",&nano,&nmes, &ndia, &nhora, &nmin);//Novos dados que indicam fim da atuação
               //Obtem a variação do tempo da ativação até a desativação
               dano += nano - ano;
               dmes += nmes - mes;
               ddia += ndia - dia;
               dhora += nhora - hora;
               dmin += nmin - min;
               last_event = "F";//Indica que o ultimo evento foi uma desativação
            }
      }
   }
    if (last_event == "I"){
      // Se percorreu todo o intervalo de tempo e o ultimo evento foi uma atuação,
      //devemos considerar que até o fim do intervalo de tempo analisado, o controlador permaneceu ativado
         sscanf(end_date.c_str(), "%d-%d-%d %d:%d",&nano,&nmes, &ndia, &nhora, &nmin);//Adquire dados de fim do intervalo como sendo fim da atuação
         dano += nano - ano;
         dmes += nmes - mes;
         ddia += ndia - dia;
         dhora += nhora - hora;
         dmin += nmin - min;
      }
      total_minutes = 525960*dano +  43800*dmes + 1440*ddia + 60*dhora + dmin;  //Soma todas as variações de tempo convertidas para minutos

   return {total_minutes};
}
```

Figure 5: ActiveDataTime Algorithm

```cpp
    string PWM_val;
    string PWM_send = "M";//Valor para que o micocontrolador indica
    //que é para alterar o microncontrolador
    cout << "Duty Cycle do PWM desejado (000 a 100): ";
    cin >> PWM_val;
    PWM_send.append(PWM_val);
    /*if (write(serial_fd, &PWM_send, 4) != 4) {
        cerr << "Error writing to serial port" <<  endl;
        close(serial_fd);
        return 1;
    }
    else{
        cout << "String Transmitida: " << PWM_send << endl;
        char buf_pwm[5];
        int ret;
        cout << "PWM Atualizado com sucesso"<< endl;
        ret = read(serial_fd,buf_pwm,sizeof(buf_pwm));
        cout << "Valor Lido: " << buf_pwm << ret;
    }*/
```

Figure 6: Alterar duty-cycle PWM