Universidade do Minho
Escola de Engenharia

Mestrado Integrado em Engenharia de Telecomunicações e Informática

Laboratórios de Telecomunicações e Informática II

Sistema de Monitorização de Atividade Física

Fase A

Grupo 2:

David José Ressurreição Alves - A79625

José Pedro Afonso Rocha - A70020

Luís Pedro Lobo de Araújo - A73232

Guimarães, 28 de Junho de 2019

Índice

Lista de Figuras

1. Introdução

Na unidade curricular de Laboratórios de Telecomunicações e Informática II, foi-nos proposto realizar um projeto que consiste na criação de um sistema de monitorização de atividade física para doentes internados numa instituição de saúde ou de apoio social.

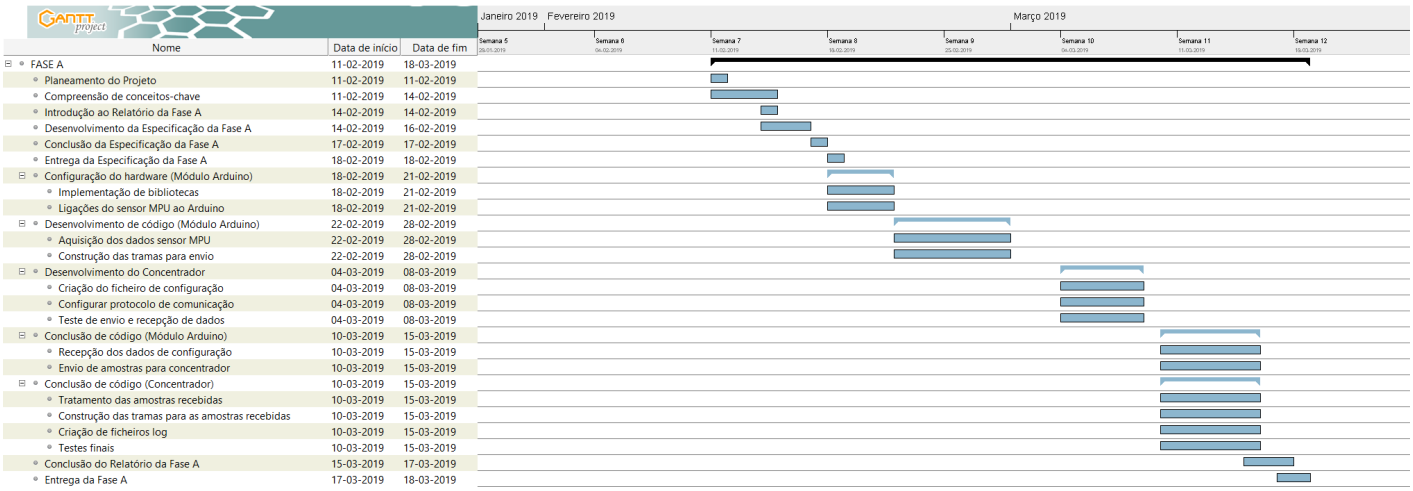
É de extrema importância a alienação da tecnologia com os cuidados de saúde pois permite uma maior atenção aos pacientes de uma instituição de saúde e a obtenção de dados em tempo real maximiza e melhora as prestações de cuidados que os médicos e enfermeiros realizam, obtendo por exemplo, se não for possível estar em acompanhamento pessoal e contínuo com o doente, todos os dados relativos desse mesmo doente em qualquer lugar, sendo alertados para qualquer problema com rapidez e com a devida urgência, se necessário.

Este sistema global terá que ter um conjunto de sistemas críticos, que serão: dispositivos concentradores de dados obtidos, dispositivos sensores atuadores, servidor web e de base de dados e uma aplicação web que tenha uma interface para interacção com o utilizador. Nesta primeira fase (Fase A), o foco será no desenvolvimento dos dispositivos sensores e concentradores e na correta ligação e comunicação entre eles.

2. Planeamento

Nesta secção encontra-se disponível a planificação temporal, do nosso grupo para esta fase A do projeto, bem como o conjunto de ferramentas que serão utilizadas neste projeto.

2.1. Planeamento temporal



Nome	Data de início	Data de fim
FASE A	11-02-2019	18-03-2019
Planeamento do Projeto	11-02-2019	11-02-2019
Compreensão de conceitos-chave	11-02-2019	14-02-2019
Introdução ao Relatório da Fase A	14-02-2019	14-02-2019
Desenvolvimento da Especificação da Fase A	14-02-2019	16-02-2019
Conclusão da Especificação da Fase A	17-02-2019	17-02-2019
Entrega da Especificação da Fase A	18-02-2019	18-02-2019
Configuração do hardware (Módulo Arduino)	18-02-2019	21-02-2019
Implementação de bibliotecas	18-02-2019	21-02-2019
Ligações do sensor MPU ao Arduino	18-02-2019	21-02-2019
Desenvolvimento de código (Módulo Arduino)	22-02-2019	28-02-2019
Aquisição dos dados sensor MPU	22-02-2019	28-02-2019
Construção das tramas para envio	22-02-2019	28-02-2019
Desenvolvimento do Concentrador	04-03-2019	08-03-2019
Criação do ficheiro de configuração	04-03-2019	08-03-2019
Configurar protocolo de comunicação	04-03-2019	08-03-2019
Teste de envio e recepção de dados	04-03-2019	08-03-2019
Conclusão de código (Módulo Arduino)	10-03-2019	15-03-2019
Recepção dos dados de configuração	10-03-2019	15-03-2019
Envio de amostras para concentrador	10-03-2019	15-03-2019
Conclusão de código (Concentrador)	10-03-2019	15-03-2019
Tratamento das amostras recebidas	10-03-2019	15-03-2019
Construção das tramas para as amostras recebidas	10-03-2019	15-03-2019
Criação de ficheiros log	10-03-2019	15-03-2019
Testes finais	10-03-2019	15-03-2019
Conclusão do Relatório da Fase A	15-03-2019	17-03-2019
Entrega da Fase A	17-03-2019	18-03-2019

Figura 1: Diagrama de Gantt com a planeamento temporal da fase A.

2.2. Ferramentas utilizadas

As ferramentas utilizadas serão as seguintes:

- Programa **GanttProject** para planeamento temporal das tarefas do grupo;
- Programa **Arduino IDE**, para editar, compilar e enviar código para a placa Arduino;
- Programa **Visual Studio Code**, para editar e compilar código;
- Plataforma **Slack**, para comunicação entre os membros do grupo;
- Plataforma **GitHub**, para partilha e organização do código desenvolvido pelo grupo.
- Plataforma **Google Drive**, para partilha de ficheiros entre os membros do grupo.
- Plataforma **OverLeaf**, para elaboração de relatórios em LateX.

3. Conceitos teóricos

3.1. Síntese do projeto

Na sua globalidade, este sistema irá conter um conjunto de dispositivos e sistemas críticos para o desenvolvimento deste projeto, sendo estes: sensores, servidores *web* e de base de dados, dispositivos de comunicação e microcontroladores. Este projeto será dividido em 4 fases (A,B,C e Final), cada uma associada a uma etapa específica que pedem o seguinte:

- **Fase A** - Planeamento, desenvolvimento e teste de todo o *hardware* e *software* necessários;
- **Fase B** - Construção de sistemas Gestores de Serviços e implementação da comunicação com os concentradores;
- **Fase C** - Desenvolvimento de *software* que implementa um Sistema Central com uma base de dados para gestão de todo o sistema de monitorização;
- **Fase Final** - Relatório final, correcção de erros, preencher lacunas do projeto e valorização do mesmo através da introdução de funcionalidades opcionais e uma apresentação final.

3.2. O sensor MPU-6050

Este sensor tem como principal função a detecção de movimento, apresentando, para isso, a posição, orientação e temperatura de qualquer objeto conectado ao sensor. Combina um giroscópio de 3 eixos, com um acelerómetro, também de 3 eixos, obtendo assim uma maior precisão nos dados recolhidos.

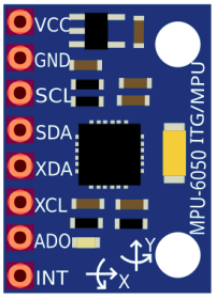


Figura 2: Sensor MPU-6050 [2].

3.2.1. O funcionamento do acelerómetro

O funcionamento do acelerómetro é baseado no efeito piezoeléctrico, segundo o qual, uma dimensão física, transformada em uma força, atua em duas faces opostas do elemento sensor [1].

Resumidamente, este efeito pode-se representar através da figura 3, imaginando-se assim que dentro do sensor existe um cubo com uma bola metálica, revestido de materiais piezoelelétricos, e que sempre que o cubo se inclinar, a bola, devido à força da gravidade, será obrigada a inclinar-se na direção do movimento [2]. Sabendo que existem 3 pares de paredes opostas dentro do cubo, cada par de paredes corresponde assim a um eixo, no espaço 3D.

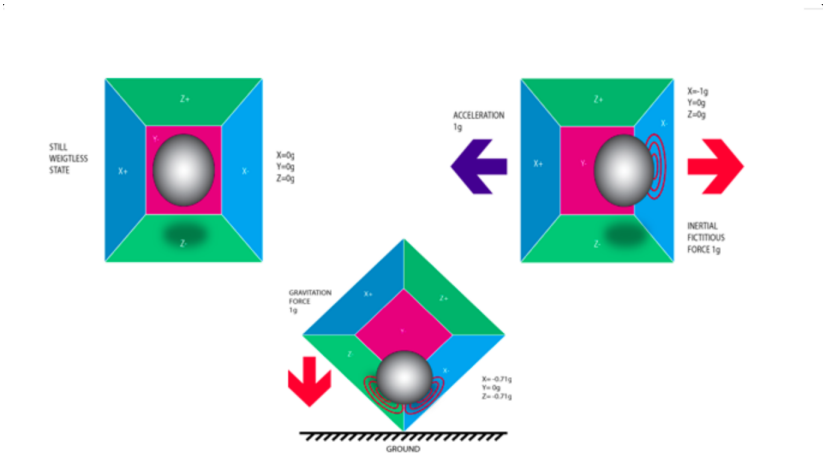


Figura 3: Funcionamento do acelerómetro [2].

3.2.2. O funcionamento do giroscópio

O funcionamento do giroscópio tem por base o princípio de Coriolis [2]. Mais uma vez, recorrendo a uma analogia, podemos imaginar que existe uma estrutura idêntica a um pêndulo, constituído por materiais piezoelétricos, e que se situa dentro de uma base metálica, cada vez que existe uma inclinação, esses materiais fazem com que esse "pêndulo"se incline na direção de uma das paredes da base metálica, tal como está representado na seguinte figura.

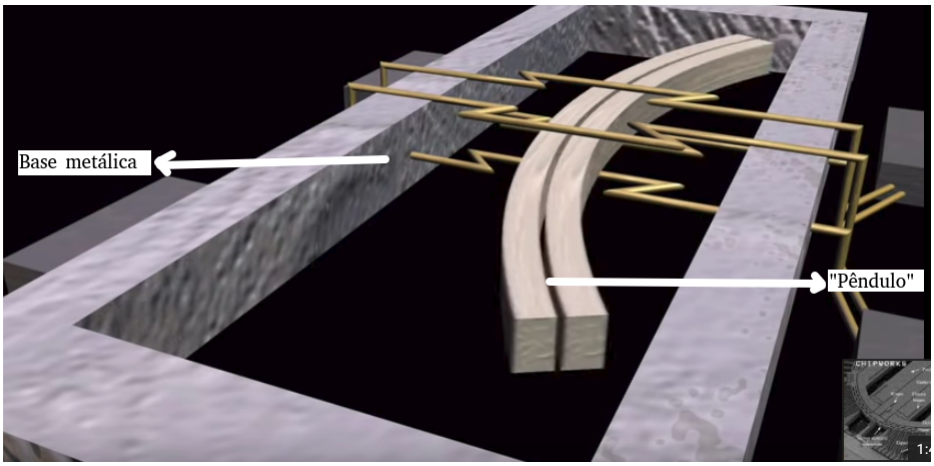


Figura 4: Funcionamento do giroscópio [2].

3.3. O Arduino

As placas Arduino (como o exemplo representado na figura 5), servem fundamentalmente para que se possa fazer uma comunicação entre os sensores e os concentradores, de uma forma mais simplificada e confiável. Além disso as grandes vantagens de usar placas Arduino para esse efeito são: reduzido custo das placas; a existência de um Arduino IDE, para escrita e envio de código para o Arduino; o facto de o projeto Arduino ser *open-source*, o que faz com que haja uma grande comunidade de desenvolvedores que expõem os seus projetos [3].

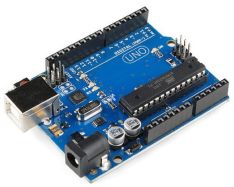


Figura 5: Placa Arduino [4].

3.4. O Concentrador

O concentrador trata-se de um programa de computador cuja função se centra na angariação e tratamento dos dados transmitidos por um ou mais sistemas sensores inseridos na área onde este pertence.

Para a implementação do concentrador iremos utilizar a linguagem C.

3.4.1. Ficheiros de log e configuração

Os ficheiros de log e configuração vão ser gravados no formato CSV (*Comma-Separated values*). Nestes encontramos os dados recebidos do concentrador, ou seja, registo de passos ou procedimentos enquanto o sistema está a ser executado, erros de execução ou funcionamento e todos os dados válidos que são recolhidos dos dispositivos sensores conectados irão ser gravados nestes ficheiros. Cada ficheiro de *log* será constituído por um identificativo de sistema sensor(ISS), identificativo do sujeito(ISu), um *timestamp* a que chamamos etiqueta temporal, o valor do período de tempo, durante a qual a amostra foi recolhida (valor_PA), o número respetivo da amostra recolhida(num_amostra), e o valor da amostra recolhida(valor_amostra).

O ficheiro *log* de dados terá a seguinte estrutura:

ISS;ISu;etiqueta_temporal;valor_PA;num_amostra;valor_amostra

Nos ficheiros de configuração vamos encontrar os parâmetros do sistema como, por exemplo, o período da amostragem (PA), o período entre mensagem de dados (PM), o número de amostras a recolher (NS), tipo de comunicação (TC) e as portas de comunicação (PC) utilizadas na ligação aos sistemas sensores.

A estrutura do ficheiro de configuração será a seguinte:

PM;PA;NS;TC;PC

4. Arquitetura do sistema

4.1. Esquema geral

Para a comunicação entre concentrador e sistema sensor de atividade física, a arquitetura do nosso sistema pode ser visualizada na seguinte figura:

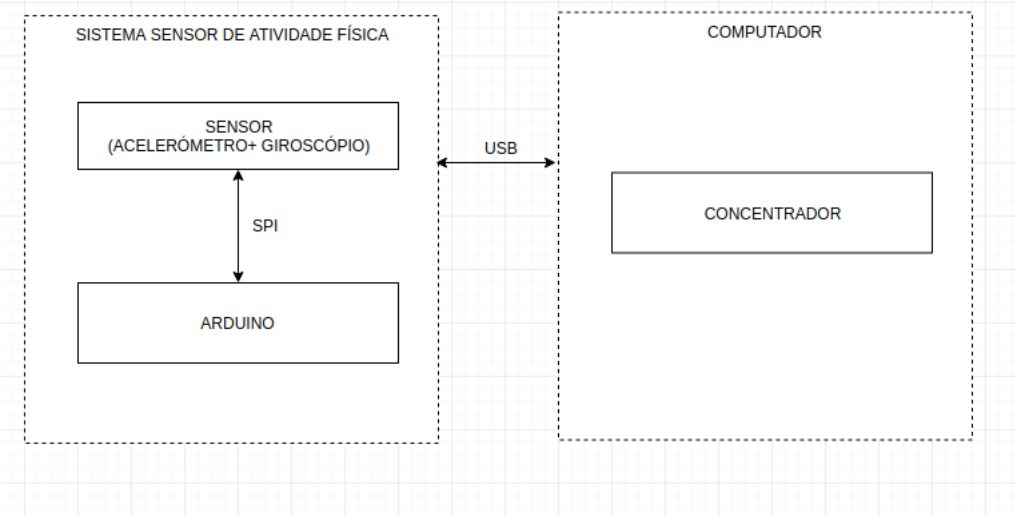


Figura 6: Arquitetura do sistema-Fase A.

Nesta primeira fase, apenas iremos utilizar a ligação por USB ao concentrador, porém é expectável fazer a conversão para um tecnologia sem-fios numa próxima fase.

4.1.1. Dispositivo sensor simulado

Existe ainda a possibilidade de adicionar um ou mais sistemas sensores simulados através de um *socket* UDP. Assim, podemos introduzir no sistema valores de sensores manualmente ou aleatoriamente, permitindo, numa próxima fase, testar a resposta do sistema.

4.2. Protocolo de comunicação

Para que haja uma eficiente interpretação dos dados recebidos e enviados entre concentrador e Arduino, foram definidas as seguintes tramas/tipos de mensagens:

- **DATA** - mensagem enviada do Arduino para o concentrador com os valores das amostras. Tal como se pode observar, definimos 1 *byte* (8 bits) para identificar o tipo de mensagem que esta a ser recebida/enviada, neste caso para o tipo DATA, definimos o valor de 0. Os restantes campos da trama foram definidos de acordo com o tipo e tamanho dos dados que estão a ser enviados/recebidos.

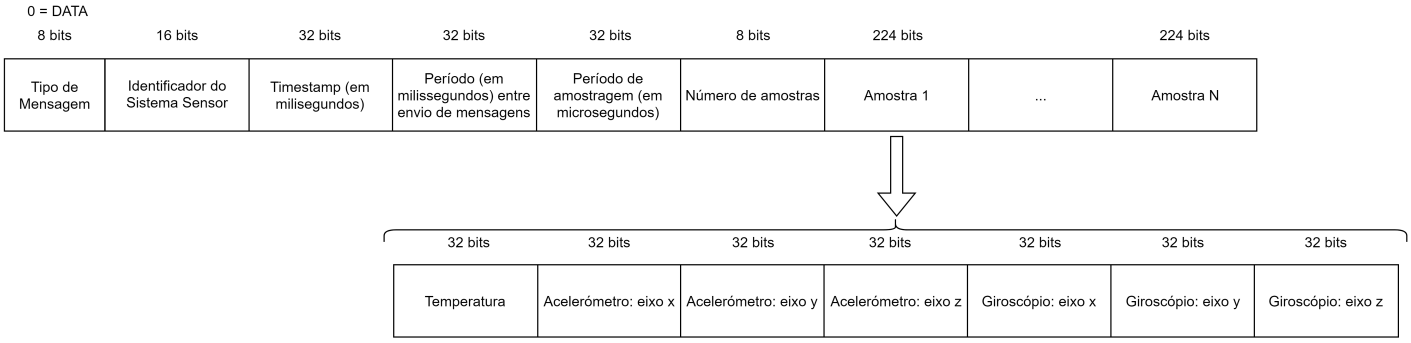


Figura 7: Definição da trama DATA.

- **ERROR** – mensagem enviada do Arduino para o concentrador a indicar uma condição de erro. Para esta trama, definimos 1 *bytes*(8 bits) para identificar o tipo de mensagem que esta a ser recebida/enviada, neste caso para o tipo ERROR, definimos o valor de 1. Os restantes campos da trama foram definidos de acordo com o tipo e tamanho dos dados que estão a ser enviados/recebidos.

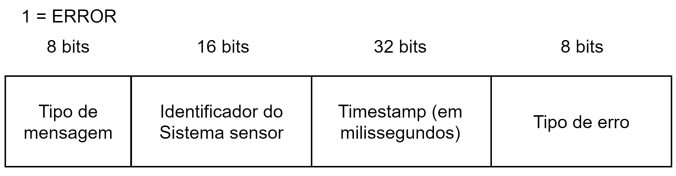


Figura 8: Definição da trama ERROR.

- **START** – mensagem enviada do concentrador para o Arduino a pedir o início da recolha e envio das amostras. Para esta trama, definimos 1 *byte* (8 bits) para identificar o tipo de mensagem que esta a ser recebida/enviada, neste caso para o tipo START, definimos o valor de 2. Os restantes campos da trama foram definidos de acordo com o tipo e tamanho dos dados que estão a ser enviados/recebidos.

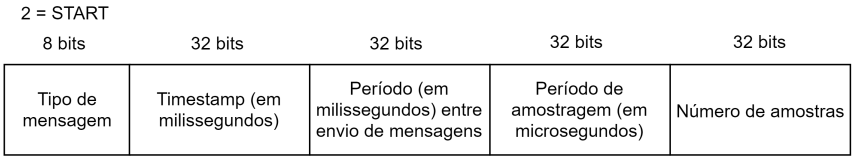


Figura 9: Definição da trama START.

- **STOP** – mensagem enviada do concentrador para o Arduino a pedir o fim da recolha e envio das amostras. Para esta trama, definimos 1 *bytes*(8 bits) para identificar o tipo de mensagem que esta a ser recebida/enviada, neste caso para o tipo STOP, definimos o valor de 3. Os seguintes campos da trama foram definidos de acordo com o tipo e tamanho dos dados que estão a ser enviados/recebidos.

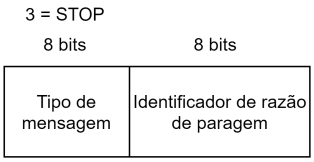


Figura 10: Definição da trama STOP.

5. Requisitos

5.1. Requisitos funcionais

Para o correto funcionamento do sistema num todo, é necessário que os seguintes requisitos sejam cumpridos:

- Captação correta dos valores de x, y, z do acelerómetro e giroscópio, bem como o valor da temperatura;
- Interpretação correta das mensagens enviadas e recebidas pelo Arduino e concentrador;
- Correto armazenamento dos dados nos ficheiros de *log*.

5.2. Requisitos não funcionais

Estes requisitos da fase A que serão abaixo enumerados são não funcionais o que indica que não estão diretamente ligados com as funcionalidades do sistema e estão mais relacionados como o tempo de resposta e a fiabilidade. Os requisitos não funcionais definidos pelo grupo são:

- Fiabilidade da comunicação entre os dispositivos concentradores e os dispositivos sensores;
- Configuração da obtenção dos valores das amostras;
- Configuração dos parâmetros da comunicação entre dispositivos concentradores e dispositivos sensores;
- Veracidade dos dados recolhidos pelos sensores.

6. Implementação

Para proceder à implementação do nosso projeto, (fase A), decidimos escrever código para Arduino, dedicado à recolha de amostras do sensor, e também escrever código em linguagem C para implementar o concentrador.

O concentrador e o sistema sensor de atividade física (Sensor MPU-6050 + Arduino), comunicam via USB.

6.1. Sistema sensor de atividade física

6.1.1. Esquema

A seguinte figura mostra o esquema da implementação do nosso sistema sensor de atividade física, sendo possível observar as respectivas ligações entre o sensor MPU-6050 e a placa Arduino.

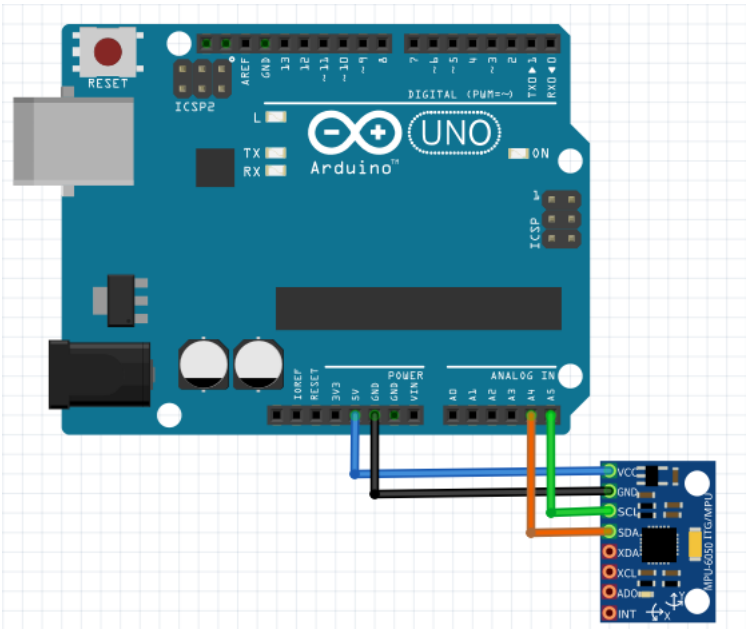


Figura 11: Esquema do sistema sensor de atividade física.

6.1.2. Código

6.1.2.1. Fluxograma

O código de Arduino que escrevemos para o sistema sensor, pode ser representado pelo seguinte fluxograma apresentado nas figuras 12 e 13:

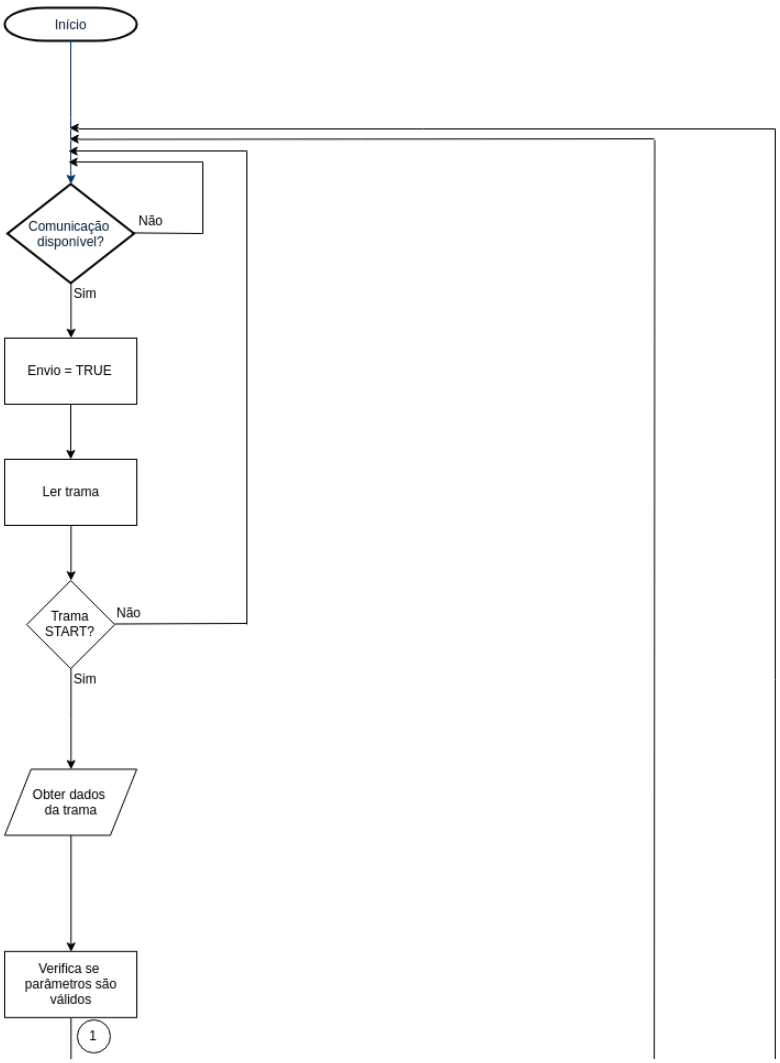


Figura 12: Fluxograma do código Arduino - parte 1.

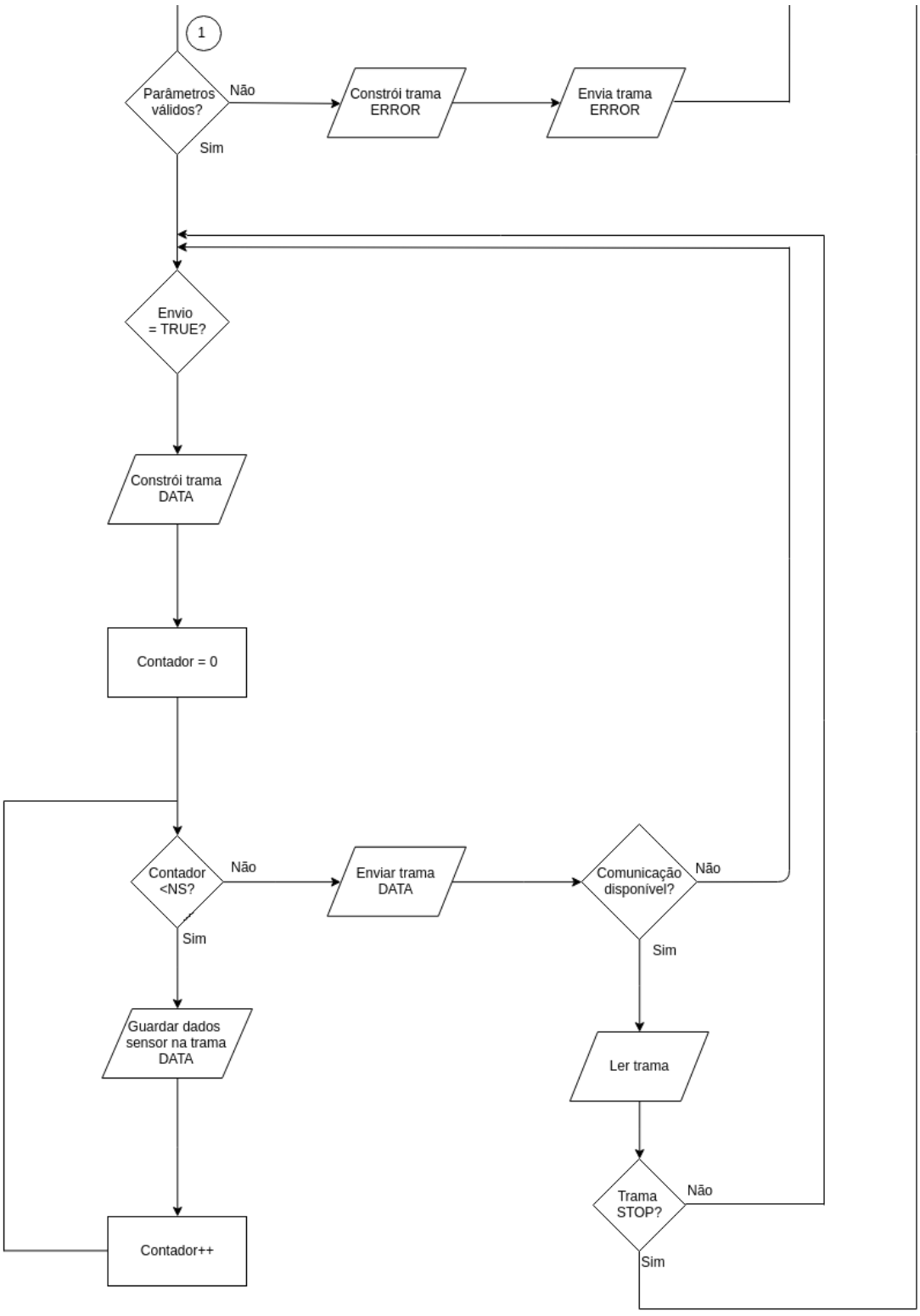


Figura 13: Fluxograma do código Arduino - parte 2.

6.1.2.2. Descrição

Para implementação do código no Arduino foram utilizadas as seguintes bibliotecas:

- <HardwareSerial.h>
- <MPU6050_tockn.h>
- <Wire.h>

A biblioteca <MPU6050_tockn.h> [5], permite-nos receber dados do sensor MPU6050 através de vários métodos já definidos, sendo só preciso a implementação dos mesmos. Para compreender esta implementação utilizamos os exemplos cedidos pela biblioteca <MPU6050_tockn.h> [6].

Numa fase inicial, foram configurados os vários parâmetros *default* do programa a compilar, tais como o tamanho das tramas e valores de amostragem por defeito caso não receba nenhum do ficheiro de configuração:

```
#define ISS 0
#define SIZEPACKETSAMPLES 28
#define SIZEDATAPACKETEMPTY 16
#define SIZEERRORPACKET 8
#define MAXNS 36 //16+(28*36)=1024
//valores default
#define PM_DEFAULT 10000 //ms
#define PA_DEFAULT 10000 //micro sec
#define NS_DEFAULT 1 //numero de amostras
```

Na função *void setup()*, são inicializados os dispositivos utilizados e a porta série é aberta com *baudrate*=115200 :

```
void setup()
{
   Serial.begin(115200);
   Wire.begin();
   mpu6050.begin();
   mpu6050.calcGyroOffsets(false);
}
```

A função *void loop()* está sempre a escutar o meio há espera de tramas vindas do concentrador e, logo que recebe a trama START, configura imediatamente todos os parâmetros recebidos do ficheiro de configuração do concentrador, verifica se não há erros e, posteriormente, começa o seu ciclo de envio de amostras do sensor consoante a quantidade que foram pedidas.

Quando recebe uma trama STOP, o sensor para de enviar amostras e reinicia a sua execução logo no início:

```
void loop()
{
   if (Serial.available())
   {
      //ler trama
      Serial.readBytes(packet, 14);
      // verificar tipo de trama
      if (packet[0] == 2) //trama de START
      {
         //obter timestamp trama e actual
         startTS = join32((char *)&packet[1]);
         initialTS = millis();

         //obter pm, pa e ns
         pm = join32((char *)&packet[5]);
         pa = join32((char *)&packet[9]);
         ns = packet[13];

         //verificar se parametros sao validos
         if (ns > 1)
         {
            if (pm < (pa * (ns - 1)))
            {
               // erro: pm nao aceitavel (id: 1)
               constructErrorPacket(packet,1);
               sizePacket=SIZEERRORPACKET;
               Serial.write(packet,sizePacket);
            }
            if (ns > MAXNS)
            {
               // erro: ns maior que o valor maximo (id: 2)
               constructErrorPacket(packet,2);
               sizePacket=SIZEERRORPACKET;
               Serial.write(packet,sizePacket);
            }
         }else if( (pm+pa+ns)==0 ) // se zero, atribuir valores default
         {
            pm = PM_DEFAULT;
            pa = PA_DEFAULT;
            ns = NS_DEFAULT;
         }else{
            //ciclo de envio
            //for(int i=0;i<1; i++) //envia 1 vez
            while(1) //envia sempre
            { //constroi e envia trama, se Serial.available, ver se recebeu STOP, se sim break
                do ciclo
               constructDataPacket(packet);
               sizePacket = SIZEDATAPACKETEMPTY;
               for (int i = 0; i < ns; i++)
               {
                  //leitura
                  putValuesOnPacket(packet + sizePacket);
                  //esperarPA
                  delayMicroseconds(pa);
                  sizePacket += SIZEPACKETSAMPLES;
               }
               //enviar trama
               Serial.write(packet, sizePacket);
               //se recebeu STOP sai do ciclo
```

```
            if (Serial.available())
            {
               Serial.readBytes(packet, 2);
               if (packet[0] == 3)
               {
                  break;
               }
            }
         }
      }
   }
}
```

Pode-se reparar que na função *void loop()* existem estas seguintes funções implementadas:

- void putValuesOnPacket(uint8_t *buf)
- void constructDataPacket(uint8_t *buf)
- uint32_t currentTimestamp()
- void constructErrorPacket(uint8_t *buf, uint8_t idError)

Na primeira função é feita a recolha dos dados do sensor (temperatura, acelerómetro e giroscópio) e são colocados no *buffer*:

```
void putValuesOnPacket(uint8_t *buf)
{
   mpu6050.update();
   float temp = mpu6050.getTemp();
   float ax = mpu6050.getAccX();
   float ay = mpu6050.getAccY();
   float az = mpu6050.getAccZ();
   float gx = mpu6050.getGyroX();
   float gy = mpu6050.getGyroY();
   float gz = mpu6050.getGyroZ();
   memcpy(buf, &temp, 4);
   memcpy(buf + 4, &ax, 4);
   memcpy(buf + 8, &ay, 4);
   memcpy(buf + 12, &az, 4);
   memcpy(buf + 16, &gx, 4);
   memcpy(buf + 20, &gy, 4);
   memcpy(buf + 24, &gz, 4);
}
```

Na segunda função é feita a construção do trama DATA para posterior envio para o concentrador:

```
void constructDataPacket(uint8_t *buf)
{
   buf[0] = (uint8_t)0; //TIPO: DATA
   uint16_t iss = ISS; //id sistema sensor
   memcpy(buf + 1, &iss, 2);
   uint32_t ts = currentTimestamp(); //timestamp
   memcpy(buf + 3, &ts, 4);
   memcpy(buf + 7, &pm, 4); //periodo entre mensagem (PM)
   memcpy(buf + 11, &pa, 4); //periodo de amostragem (PA)
   memcpy(buf + 15, &ns, 1); //numero de amostras
}
```

A terceira função devolve o valor do *timestamp* atual:

```
uint32_t currentTimestamp(){
   // calcular ts desde start mais ts de enviado do concentrador
   return (initialTS-millis())+startTS; //timestamp
}
```

A quarta função serve para construir a trama ERROR caso haja algum problema identificado:

```
void constructErrorPacket(uint8_t *buf, uint8_t idError){
   buf[0] = (uint8_t)1; //TIPO: ERROR
   uint16_t iss = ISS; //id sistema sensor
   memcpy(buf + 1, &iss, 2);
   uint32_t ts = currentTimestamp(); //timestamp
   memcpy(buf + 3, &ts, 4);
   memcpy(buf+7,&idError,1);
}
```

Como o Arduino, por porta série, apenas escreve de *byte* em *byte* e 1 *byte* equivale a 8 bits então, será preciso em muitos casos, dividir dados de modo a que cada pacote resultante dessa divisão ocupe 1 *byte*. Por exemplo, se uma trama ocupar 32 bits terá que ser separada em 4 pacotes de 8 bits para ser enviada corretamente:

```
void split32(char *buffer, uint32_t value)
{
   UINT32UNION_t aux;
   aux.number = value;
   for (int i = 0; i < 4; i++)
   {
      buffer[i] = aux.bytes[i];
   }
}
```

Ao receber uma trama o processo é o mesmo, mas de modo inverso, pois quem separa é o concentrador que enviou a trama e, neste caso, o Arduino junta os pacotes recebidos para formar uma trama. Por exemplo, se receber 32 bits:

```
uint32_t join32(char *buffer)
{
   UINT32UNION_t aux;
   for (int i = 0; i < 4; i++)
   {
      aux.bytes[i] = buffer[i];
   }
   return aux.number;
}
```

6.2. Concentrador

6.2.1. Código

6.2.1.1. Fluxograma

O código em lingugem C, que escrevemos para o concentrador, pode ser representado pelo seguinte fluxograma, apresentado nas figuras 14 e 15:

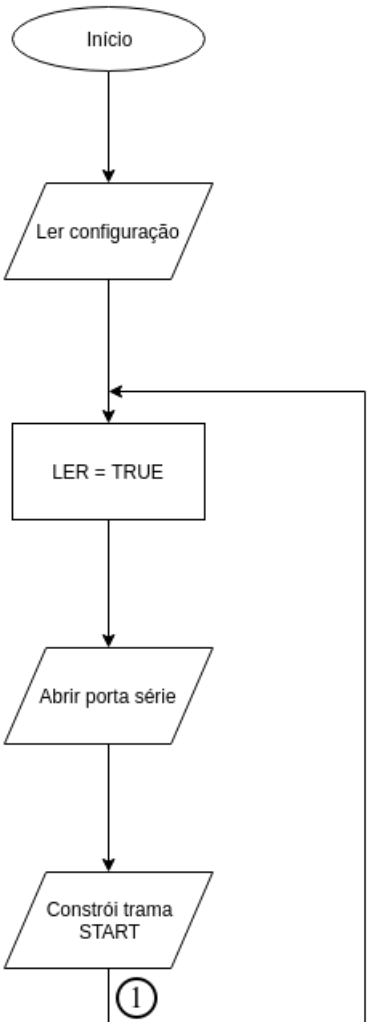


Figura 14: Fluxograma do código C - parte1.

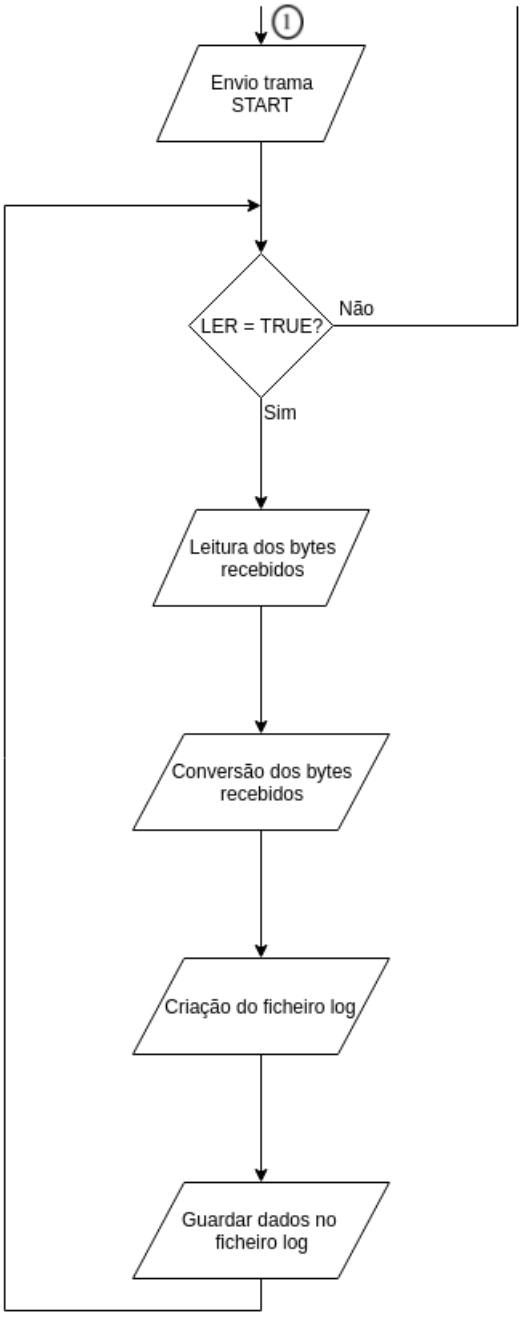


Figura 15: Fluxograma do código C - parte 2.

6.2.1.2. Descrição

Para implementação do código no concentrador, para além das bibliotecas típicas que se têm que introduzir para o funcionamento das várias funções da linguagem C, foram utilizadas a seguinte biblioteca e *header* específico:

- "rs232.h"
- "api.h"

A biblioteca "rs232.h"[7] foi a escolhida pelo grupo para implementação da comunicação porta série do concentrador com o Arduino. Através das funções já implementadas na biblioteca, podemos enviar ou receber dados do sensor interligado ao Arduino.

Também foi acrescentada uma função nessa biblioteca, a função *int comReadBytes(int index, char * buffer, size_t len, int numBytes, uint16_t timeoutMs)*, onde tenta ler N *numBytes* em *timoutMS* milissegundos, caso não seja possível, devolve todos aqueles que já foram recebidos:

```
int comReadBytes(int index, char * buffer, size_t len, int numBytes, uint16_t timeoutMs)
{
    int bytesRead=0, res=0;
    while(bytesRead<numBytes){
        if (index >= noDevices || index < 0)
            return 0;
        if (comDevices[index].handle <= 0)
            return 0;
        res = read(comDevices[index].handle, buffer+bytesRead, (numBytes-bytesRead) );
        if (res < 0)
            res = 0;
        else{
            if(res>0)
                bytesRead=bytesRead+res;
        }
        usleep( 1000 ); // wait 1 msec try again
        timeoutMs--;
        if( timeoutMs==0 ){
            return bytesRead;
        }
    }

    return bytesRead;
}
```

O ficheiro *header* "api.h"foi criado pelo grupo e lá dentro está todos os parâmetros *default* definidos, variáveis, estruturas de configurações e estruturas para conversão de *bytes*.

Na função *main()* do concentrador são realizadas todas as operações precisas para enviar e receber dados.

Numa fase inicial, começa-se por ler o ficheiro de configuração que servirá para criar a trama START, abre-se a porta série para o começo da comunicação, são criados os ficheiros que irão guardar o output do concentrador (log da trama, dados das amostras recebidas e erros recebidos) e é enviada, finalmente, a trama START para o Arduino.

De seguida, o concentrador entra num ciclo de leitura, onde espera pela recepção dos *bytes* relativos aos dados das amostras pedidas de modo a poder processar e aguardar os resultados nos ficheiros log, é também verificado se ocorreu algum erro de configuração.

No excerto abaixo apresenta-se a implementação da função *main()*:

```
int main(int argc, char const *argv[])
{
   //armadilhar sinal
   signal(SIGINT, signalhandler);

   readConfig();

   printConfig();

   init();

   //enumerar as coms para obter coms
   int coms = comEnumerate();
   //abrir porta
   openSerial();
   char str[1024];
   char buf[1024];
   int sizeRead;

   //tempo de espera para fazer a calibraao
   sleep(2);

   build_trama_start(str, actualConfig.pm, actualConfig.pa, actualConfig.ns);
   for (int i = 0; i < 14; i++)
   {
      printf("%u ", (uint8_t)str[i]);
   }
   printf("\n");
   //enviar trama de start
   //criar log do start
   time_t ltime;         /* calendar time */
   ltime = time(NULL); /* get current cal time */
   sprintf(buf, "Timestamp: %s", asctime(localtime(&ltime)));
   write(fdlog, buf, strlen(buf));
   comWrite(actualConfig.serialIndex, str, 14);
   //sleep(2);
   //ciclo de leitura de valores (amostras ou erro), e criaao de logs
   while (1)
   {
      // ler da Serial com timeout 5s
      sizeRead = comReadBytes(actualConfig.serialIndex, buf, sizeof(buf), SIZEDATAPACKETEMPTY
          + (actualConfig.ns * SIZEPACKETSAMPLES), 5000);
      if (sizeRead > 0)
      {
         printf("\n"); //printf("SizeRead: %d\n",sizeRead);

         //verificar tipo de trama
         if (buf[0] == DATA)
         { //se data ler valores
            //escrever para ficheiro output e mostrar valores no ecra
            printf("DATA ISS: %u ISu: %u Timestamp: %u PA: %u NS: %u", join16(buf + 1),
                getISu(join16(buf + 1)), join32(buf + 3), join32(buf + 11), actualConfig.ns);
            sprintf(str, "%u;%u;%u;%u;%u", join16(buf + 1), getISu(join16(buf + 1)), join32(buf
                + 3), join32(buf + 11), actualConfig.ns);
            write(fdout, str, strlen(str));
            uint8_t nsT = actualConfig.ns * 7;
            for (int i = 0; i < nsT; i++)
            {
               float val = joinFloat(buf + 16 + (i * 4));
               switch (i % 7)
               {
               case 0: //temp
                  printf(" Temp: %f", val);
                  sprintf(str, ";%f", val);
```

```
                write(fdout, str, strlen(str));
                break;
            case 1: //offset accX
                printf(" RawAccX: %f AccX: %f", val, val + 0.03);
                sprintf(str, ";%f", val + 0.03);
                write(fdout, str, strlen(str));
                break;
            case 2: //offset accY
                printf(" RawAccY: %f AccY: %f", val, val + 0.14);
                sprintf(str, ";%f", val + 0.14);
                write(fdout, str, strlen(str));
                break;
            case 3: //offset accZ
                printf("\nRawAccZ: %f AccZ: %f", val, val - 0.74);
                sprintf(str, ";%f", val - 0.74);
                write(fdout, str, strlen(str));
                break;
            case 4: //gyroX
                printf(" RawGyroX: %f", val);
                sprintf(str, ";%f", val);
                write(fdout, str, strlen(str));
                break;
            case 5: //gyroY
                printf(" RawGyroY: %f", val);
                sprintf(str, ";%f", val);
                write(fdout, str, strlen(str));
                break;
            case 6: //gyroZ
                printf(" RawGyroZ: %f", val);
                sprintf(str, ";%f", val);
                write(fdout, str, strlen(str));
                break;

            default: //restantes
                printf(";%f", val);
                sprintf(str, ";%f", val);
                write(fdout, str, strlen(str));
                break;
            }
        }
        printf("\n");
        write(fdout, "\n", 1);
    }
    else if (buf[0] == ERROR)
    {
        //envia erro para log e sai do ciclo
        printf("ERROR\n");
        time_t ltime;         /* calendar time */
        ltime = time(NULL); /* get current cal time */
        sprintf(str, "ERROR TYPE: %d Timestamp: %s", buf[7],asctime(localtime(&ltime)));
        write(fderr, str, strlen(str));
        //fechar
        closeFd();
        //sair do ciclo
        break;
    }
  }
 }
}
```

Pode-se reparar que na função *int main()* existem as seguintes funções implementadas:

- int readConfig();
- printConfig();
- openSerial();

- uint32_t currentTimestamp();
- void build_trama_start(char * str,uint32_t pm,uint32_t pa, uint32_t ns);
- void signalhandler(int signal);
- void build_trama_stop(char * str, uint8_t errorId);

Na primeira função, como já foi referido, faz-se a leitura do ficheiro de configuração, onde contém os parâmetros necessários à construção da trama START:

```
int readConfig(){
   FILE *fp;
   char *line = NULL;
   size_t len = 0;
   ssize_t read;

   fp = fopen(CONFIG_FILE, "r");
   if (fp == NULL)
      return -1;

   read = getline(&line, &len, fp);
   printf("%s\n", line);

   int argSize=0;
   int tc=NONE;
   char *token = strtok(line, ";");

   // Keep looping while one of the
   // delimiters present in str[].
   while (token != NULL)
   {
      switch (argSize)
      {
         case 0:
            actualConfig.pm = atoi(token);
            break;

         case 1:
            actualConfig.pa = atoi(token);
            break;

         case 2:
            actualConfig.ns = atoi(token);
            break;

         default:
            //check if Serial or UDP
            if( (argSize%2) == 1){ //tipo de comunicaao
               if(strcmp(token,"Serial")==0){
                  tc=SERIAL;
               }else if(strcmp(token,"UDP")==0){
                  tc=UDP;
               }else{
                  tc=NONE;
               }
            }else //porta de comunicaao
            {
               switch (tc)
               {
                  case SERIAL:
                     strcpy(actualConfig.portSerial,token);
                     break;

                  case UDP:
                     strcpy(actualConfig.portUDP,token);
                     break;

                  default:
```

```
                    break;
                }
            }

            break;
        }
        argSize++;
        token = strtok(NULL, ";");
    }

    fclose(fp);
    if (line)
        free(line);

    if( (actualConfig.pm+actualConfig.pa+actualConfig.ns)==0 ) // se zero, atribuir valores
        default
    {
        actualConfig.pm = PM_DEFAULT;
        actualConfig.pa = PA_DEFAULT;
        actualConfig.ns = NS_DEFAULT;
    }

    return 0;
}
```

Na segunda função, *void printConfig()*, é feito o *print* dos valores de configuração para saber se foram lidos corretamente:

```
void printConfig(){
    printf("pm: %d, pa: %d, ns: %d\n",actualConfig.pm, actualConfig.pa, actualConfig.ns);
    printf("Serial Port: %s\n",actualConfig.portSerial);
    printf("UDP Port: %s\n",actualConfig.portUDP);
}
```

A terceira função, *openSerial()*, permite-nos procurar portas série disponíveis e abrir-las para o começo da comunicação entre o concentrador e o Arduino:

```
void openSerial(){
    actualConfig.serialIndex=comFindPort(actualConfig.portSerial);
    printf("Index: %d\n", actualConfig.serialIndex);
    if(actualConfig.serialIndex==-1){
        actualConfig.serialIndex=8;
    }
    printf("Index: %d\n", actualConfig.serialIndex);
    comOpen(actualConfig.serialIndex,115200);
    sleep(2);
}
```

A quarta função, *uint32_t current_timestamp()*, dá-nos o valor do tempo atual em milissegundos:

```
uint32_t current_timestamp() {
    struct timeval te;
    gettimeofday(&te, NULL); // get current time
    return (uint32_t)te.tv_sec;
}
```

A quinta função, *void build_trama_start(char * str,uint32_t pm,uint32_t pa, uint32_t ns)* faz-nos a construção da trama START consoante os valores recebidos do ficheiro de configuração:

```
void build_trama_start(char * str,uint32_t pm,uint32_t pa, uint32_t ns){
    str[0]=(char)2;
    uint32_t ts = current_timestamp();
    split32(str+1,ts);
```

```
   split32(str+5,pm);
   split32(str+9,pa);
   str[13]=ns;
}
```

A sexta função, *void signalhandler(int signal)*, trata de armadilhar o sinal de interrupção, constrói a trama STOP e envia para o Arduino para terminar o envio das amostras:

```
void signalhandler(int signal){
   char str[1024];
   build_trama_stop(str,1);
   comWrite(actualConfig.serialIndex,str,2);
   sleep(2);
   comTerminate();
   closeFd();
   _exit(0);
}
```

Finalmente, na sétima função, *void build_ trama_ stop(char * str, uint8_ t errorId)*, é construída a trama STOP para depois ser enviada ao Arduino:

```
void build_trama_stop(char * str, uint8_t errorId){
   str[0]=(char)3;
   str[1]=(char)errorId;
}
```

7. Testes e análise de resultados

Neste capítulo iremos apresentar os vários testes realizados ao sistema de modo perceber se todos os dados gerados pelo sensor e enviados ao concentrador estavam de acordo com o que estava previsto e com a funcionalidade do sistema, garantido a fiabilidade de toda a comunicação.

Na figura 16, são mostrados os dados recebidos do Arduino, nomeadamente várias tramas DATA que contêm todos os dados recebidos do sensor MPU6050, períodos da amostragem para cada amostra e a identificação do sistema sensor que as enviou. Além disso também são imprimidos os parâmetros recolhidos do ficheiro de configuração, de forma a confirmar a correta introdução dos mesmos no concentrador.

```
MacBook-Pro-de-Luis:lti2-1819 luispedrolobodearaujo$ ./a.out
3000000;1000000;1;Serial;tty.usbmodem1434101;UDP;127.0.0.1:4444
pm: 3000000, pa: 1000000, ns: 1
Serial Port: tty.usbmodem1434101
UDP Port: 127.0.0.1:4444
Index: 11
Index: 11
Try /dev/tty.usbmodem1434101
Open /dev/tty.usbmodem1434101
2 245 69 141 92 192 198 45 0 64 66 15 0 1

DATA ISS: 0 ISu: 1 Timestamp: 1552762357 PA: 1000000 NS: 1 Temp: 29.447060 RawAccX: -0.026855 AccX: 0.003145 RawAccY: -0.125000 AccY: 0.015000
RawAccZ: 1.744385 AccZ: 1.004385 RawGyroX: -0.166247 RawGyroY: -0.048093 RawGyroZ: -0.012427

DATA ISS: 0 ISu: 1 Timestamp: 1552763360 PA: 1000000 NS: 1 Temp: 29.588236 RawAccX: -0.025635 AccX: 0.004365 RawAccY: -0.125732 AccY: 0.014268
RawAccZ: 1.740234 AccZ: 1.000234 RawGyroX: 0.093295 RawGyroY: -0.139696 RawGyroZ: 0.048641

DATA ISS: 0 ISu: 1 Timestamp: 1552764363 PA: 1000000 NS: 1 Temp: 29.494118 RawAccX: -0.024902 AccX: 0.005098 RawAccY: -0.126709 AccY: 0.013291
RawAccZ: 1.750977 AccZ: 1.010977 RawGyroX: -0.074644 RawGyroY: 0.104579 RawGyroZ: -0.088763

DATA ISS: 0 ISu: 1 Timestamp: 1552765366 PA: 1000000 NS: 1 Temp: 29.447060 RawAccX: -0.021729 AccX: 0.008271 RawAccY: -0.130859 AccY: 0.009141
RawAccZ: 1.748047 AccZ: 1.008047 RawGyroX: 0.032227 RawGyroY: 0.074045 RawGyroZ: -0.119297

DATA ISS: 0 ISu: 1 Timestamp: 1552766369 PA: 1000000 NS: 1 Temp: 29.447060 RawAccX: -0.024170 AccX: 0.005830 RawAccY: -0.135742 AccY: 0.004258
RawAccZ: 1.742676 AccZ: 1.002676 RawGyroX: -0.150980 RawGyroY: 0.012976 RawGyroZ: 0.018107

DATA ISS: 0 ISu: 1 Timestamp: 1552767372 PA: 1000000 NS: 1 Temp: 29.447060 RawAccX: -0.025879 AccX: 0.004121 RawAccY: -0.130371 AccY: 0.009629
RawAccZ: 1.750732 AccZ: 1.010732 RawGyroX: -0.212048 RawGyroY: 0.180915 RawGyroZ: 0.018107
```

Figura 16: Dados das amostras recebidas do Arduino.

Na figura 17, podemos observar que o concentrador regista para um ficheiro de *log*, o *timestamp*, do histórico de inicialização do sistema, apresentado em dia da semana (em inglês), mês (abreviado), dia do mês, horas, minutos, segundos e ano.

```
MacBook-Pro-de-Luis:lti2-1819 luispedrolobodearaujo$ cat log.txt
Timestamp: Sat Mar 16 18:51:51 2019
Timestamp: Sat Mar 16 18:52:37 2019
Timestamp: Sat Mar 16 19:56:19 2019
Timestamp: Sat Mar 16 19:56:33 2019
Timestamp: Sat Mar 16 19:56:52 2019
Timestamp: Sat Mar 16 19:57:20 2019
Timestamp: Sat Mar 16 19:57:44 2019
Timestamp: Sat Mar 16 19:58:06 2019
```

Figura 17: Verificação de escrita para ficheiro de *log*.

Testámos também a ocorrência de possíveis erros, e como o sistema lidaria com os mesmos. Para isso definimos previamente uma lista de associação de erros:

1 - Caso valor pm seja não aceitável;

2 - Caso valor de ns seja maior que valor máximo, que é 36 devido ao *buffer* ser de 1024 *bytes*, dado que esse é o número máximo de amostras que cabem nesse tamanho.

```
MacBook-Pro-de-Luis:lti2-1819 luispedrolobodearaujo$ cat error.txt
ERROR TYPE: 1 Timestamp: Sat Mar 16 20:06:41 2019
ERROR TYPE: 1 Timestamp: Sat Mar 16 20:07:41 2019
ERROR TYPE: 1 Timestamp: Sat Mar 16 20:09:30 2019
ERROR TYPE: 1 Timestamp: Sat Mar 16 20:09:53 2019
```

Figura 18: Verificação de ocorrência de erro.

Os dados das amostras foram todos guardados num ficheiro *output* como se pode verificar a seguir na figura 19, e confirma-se que os dados processados estão correctamente recebidos e guardados.

Ao fim de várias amostras tratadas provou-se que elas estão sempre dentro da gama de valores pretendidos e de acordo com a calibração definida.

```
MacBook-Pro-de-Luis:lti2-1819 luispedrolobodearaujo$ cat output.txt
0;1;1552767215;1000000;1;28.929411;0.004854;0.011826;1.004873;-0.259630;0.102461;0.127731
0;1;1552768218;1000000;1;28.976471;0.005586;0.015732;0.999746;-0.152760;-0.034943;-0.177613
0;1;1552769221;1000000;1;29.023529;0.006807;0.014023;1.008535;-0.045890;0.178797;-0.040208
0;1;1552770224;1000000;1;28.976471;0.006562;0.008408;1.008047;-0.106959;0.087194;-0.116544
0;1;1552766240;1000000;1;29.070589;0.004121;0.012803;1.010732;-0.221523;-0.012847;-0.128173
0;1;1552767243;1000000;1;28.976471;0.007539;0.011094;1.001455;-0.038317;-0.043381;-0.051837
0;1;1552768246;1000000;1;29.023529;0.008271;0.007676;1.007559;-0.053584;-0.073915;-0.036570
0;1;1552769249;1000000;1;29.023529;0.003145;0.004990;1.005361;0.053286;-0.043381;0.055033
0;1;1552770252;1000000;1;28.929411;-0.000518;0.008896;1.001699;0.053286;-0.073915;0.055033
0;1;1552771255;1000000;1;28.929411;0.000459;0.015000;1.013662;-0.099386;0.048222;0.100835
0;1;1552772258;1000000;1;29.023529;0.002900;0.011582;1.007314;-0.068851;-0.028114;-0.021303
0;1;1552773261;1000000;1;28.929411;0.006318;0.012803;1.006338;-0.084118;0.063489;-0.006036
0;1;1552774264;1000000;1;28.882353;0.006074;0.013291;1.001943;0.114355;-0.104450;0.085568
```

Figura 19: Dados das amostras guardados no ficheiro *output*.

8. Conclusão

Após a conclusão desta fase A, podemos afirmar que ficamos satisfeitos com o nosso desempenho, uma vez que cumprimos os objetivos propostos pelo grupo, tendo sido um sucesso a implementação da comunicação entre os dispositivos sensores e o concentrador.

Através do cumprimento rigoroso das tarefas atribuídas a cada elemento do grupo e do planeamento proposto, as dificuldades que foram surgindo ao longo desta fase rapidamente foram ultrapassadas, apesar da inexperiência inicial do grupo em algumas matérias e tecnologias. O empenho de cada elemento no sentido de pesquisa e estudo da melhor implementação foi crucial para a conclusão desta fase.

Contudo é de notar que o grupo ainda ambicionava desenvolver mais o código do concentrador, sobretudo no que diz respeito à implementação de módulos independentes de comunicação, leitura e escrita, algo que pretendemos desenvolver no seguimento das próximas fases.

9. Referências

[1] **"The Piezoelectric Effect - Piezoelectric Motors Motion Systems", Nanomotion,**

Disponível em: https://www.nanomotion.com/piezo-ceramic-motor-technology/piezoelectric-effect/ [Acedido em 11 de março 2019].

[2] **"How to Interface Arduino and the MPU 6050 Sensor | Arduino", Maker Pro,**

Disponível em: https://maker.pro/arduino/tutorial/how-to-interface-arduino-and-the-mpu-6050-sensor [Acedido em 11 de março de 2019].

[3] **"Arduino - Introduction", Arduino.cc,**

Disponível em: https://www.arduino.cc/en/Guide/Introduction [Acedido em 11 de março de 2019].

[4] **G. Bauermeister e A. Thomsen, "Placa Uno R3 + Cabo USB para Arduino - Loja FilipeFlop", FilipeFlop,**

Disponível em: https://www.filipeflop.com/produto/placa-uno-r3-cabo-usb-para-arduino/ [Acedido em 11 de março de 2019].

[5] **"tockn/MPU6050_tockn", GitHub,**

Disponível em: https://github.com/tockn/MPU6050_tockn. [Acedido em 16 de março 2019].

[6] **"tockn/MPU6050_tockn", GitHub,**

Disponível em: https://github.com/tockn/MPU6050_tockn/blob/master/examples/GetAllData/GetAllData.ino. [Acedido em 16 de março 2019].

[7] **"Marzac/rs232", GitHub,**

Disponível em: https://github.com/Marzac/rs232. [Acedido em 16 de março 2019].

I2C