

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

FACULDADE DE ENGENHARIA MECÂNICA

UBERLÂNDIA

**Exercício Prático**

Comunicação Integrada Entre Dispositivos

REDES INDUSTRIAIS

JOSÉ JEAN-PAUL ZANLUCCHI DE SOUZA TAVARES

|  |  |  |
| --- | --- | --- |
| **Data** | **Editor** | **Revisão** |
| 19/07/2021 | José Jean-Paul Z. S. Tavares | 0 |
| 03/12/2021 | José Jean-Paul Z. S. Tavares | 1 – Atualização Exercícios |
| 08/05/2022 | José Jean-Paul Z. S. Tavares | 2 – Atualização Exercícios |
| 03/10/2022 | José Jean-Paul Z. S. Tavares | 3 – Atualização Exercícios |
| 05/03/2023 | José Jean-Paul Z. S. Tavares | 4 – Atualização Exercícios |

Sumário

[INTRODUÇÃO 3](#_Toc77580033)

[OBJETIVOS 4](#_Toc77580034)

[FUNDAMENTAÇÃO TEÓRICA 5](#_Toc77580035)

[ARDUINO 5](#_Toc77580036)

[COMUNICAÇÃO SERIAL 9](#_Toc77580037)

[IDE (*Integrated Development Environment*) 10](#_Toc77580038)

[MÉTODO DE ACESSO 14](#_Toc77580039)

[COMUNICAÇÃO ETHERNET 19](#_Toc77580040)

[ZigBee 25](#_Toc77580041)

[Iniciando um XBEE 26](#_Toc77580042)

[Configuração *AT-Mode* 28](#_Toc77580043)

[Configuração *API-Mode* 31](#_Toc77580044)

[**Exercício Prático (28 Pontos)** 35](#_Toc77580045)

[BIBLIOGRAFIA 37](#_Toc77580046)

# INTRODUÇÃO

O desenvolvimento de sistemas integrando eletrônica, mecânica, controle e informática gera, entre outros desafios, necessidade de garantir que a comunicação entre dispositivos ocorra em um prazo pré-determinado conhecido, mesmo que seja no pior caso, bem como que equipamentos possam acessar dados de maneira confiável e segura.

A comunicação de dados trata da troca de informações entre computadores e outros dispositivos microprocessados através de um meio de transmissão comum. A comunicação envolve no mínimo uma rede e dois dispositivos capazes de trocar informações. Os dispositivos envolvidos na comunicação podem estar distanciados em alguns metros (por exemplo no sistema Bluetooth) ou por distâncias ilimitadas (como no caso da Internet).

A comunicação entre dispositivos pode variar de acordo com características do ambiente e com a hierarquia dos dispositivos. Por isso há várias formas de se comunicar. Um exemplo é a topologia mestre-escravo em que o dispositivo considerado mestre pode comunicar livremente com qualquer dispositivo escravo, enquanto o escravo só responde se receber uma mensagem de algum mestre.

Este módulo apresenta diversos tipos de comunicação entre dispositivos. Chamamos de comunicação serial o método de transmissão de dados por meio de um único par de cabos e o envio da informação ocorre de forma sequencial. O protocolo I²C, por exemplo, se utiliza de um barramento serial e emula uma comunicação do tipo mestre e escravo. Outro tipo de comunicação é a Token Ring entre dispositivos, uma comunicação em uma rede ponto a ponto onde cada dispositivo se conecta fisicamente com outros dois de forma a criar uma rede em anel. O token, permissão ou ficha, é uma mensagem especial enviada de um dispositivo para o seu sucessor até retornar ao dispositivo inicial que serve para indicar que o meio está livre para comunicação. Há também a comunicação sem fio ou wireless. Esse tipo de comunicação é ponto a ponto em uma rede no padrão ZigBee, que foi criado devido inexistência de uma solução normalizada de como se operar com redes sem fios para fins como controle e telemetria, principalmente nas áreas de automação industrial e residencial.

# OBJETIVOS

O objetivo principal deste exercício prático é ensinar o aluno a estabelecer uma comunicação entre dispositivos.

Os objetivos específicos consistem em:

1) Compreender como se implementa um programa de comunicação com protocolo I²C entre dois Arduinos.

2) Compreender como se implementa programas de comunicação entre Arduinos utilizando comunicação Ethernet entre dispositivos.

3) Compreender como se implementa programas de comunicação entre Arduinos no padrão Wireless IEEE802.15 entre dispositivo (Zigbee).

4) Desenvolver o projeto de um sistema misto de comunicação entre Arduinos aplicado a um sistema eletropneumático.

# FUNDAMENTAÇÃO TEÓRICA

## ARDUINO

Mesmo que as pessoas não se deem conta, utilizamos, hoje em dia, uma média de nove microcontroladores diariamente. Dentre estes, os mais comuns estão inseridos em televisores, máquinas de lavar (roupa ou louça), controle remoto, DVD e alarmes.

Microcontroladores são equipamentos compostos por uma central aritmética, capaz de fazer cálculos matemáticos, uma memória capaz de manipular e armazenar dados e programas, e diversos componentes de entrada e saída para comunicação com o exterior. Simplificando, o microcontrolador é um computador encapsulado em um único *chip*.

Os microcontroladores possuem baixo custo, são fáceis de encontrar no mercado, possuem compiladores para linguagem de alto nível e integração com controladores ethernet.

ARDUINO é uma plataforma livre de protótipos eletrônicos baseada em flexibilidade, e facilidade de usar o *hardware* e o *software*. Ele é destinado a artistas, designers, *hobbistas* e qualquer pessoa interessada em criar objetos interativos ou ambientes.

O hardware é de fácil aquisição e seu IDE (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado) está disponível para *download* gratuito no *site* http://arduino.cc/.

A seguir são apresentados algumas declarações, funções, parâmetros utilizados na programação de um ARDUINO.

***Constantes e Variáveis***

As constantes booleanas são valores predefinidos que nunca podem ser alterados. Na linguagem C do ARDUINO são 3 os grupos de constantes booleanas; os dois componentes de cada grupo sempre podem ser representados pelos números binários 1 e 0.

TRUE/FALSE: são constantes booleanas que definem estados lógicos. Verdadeiro é qualquer valor que não seja zero. Falso é sempre o valor zero.

HIGH/LOW: são constantes que definem as tensões nos pinos digitais do ARDUINO. High (Alto) é uma tensão de 5 Volts e Low (baixo ou terra) é uma tensão de 0 Volts.

INPUT/OUTPUT: são constantes programadas pela função **pinMode()** para os pinos do ARDUINO. Eles podem ser entradas (sensores) ou saídas (controle).

As variáveis são posições na memória de programa do ARDUINO marcadas com um nome e o tipo dessa informação que irão guardar. Essas posições podem estar vazias ou podem receber um valor inicial.

BYTE: armazena 8 bits (0-255);

INT: armazena números inteiros de até 16 bits;

LONG: armazena números inteiros de até 32 bits;

FLOAT: armazena números fracionários de até 32 bits.

CHAR: armazena um tipo de dado que ocupa 1 byte de memória

STRING: armazena um vetor de dados do tipo ‘char’

***Lógica e Aritmética***

Símbolos compostos são aqueles que combinam os símbolos aritméticos entre si e com o sinal de atribuição. A Tabela 1 apresenta as principais operações.

**Tabela 1: Lista de símbolos para as operações aritméticas**

|  |  |
| --- | --- |
| x ++ | x = x +1 |
| x -- | x = x -1 |
| x += y | x = x + y |
| x -= y | x = x – y |
| x \*= y | x = x\*y |
| x /= y | x = x/y |

Operadores de comparação comparam uma variável com uma constante, ou variáveis entre si. São usados para testar se uma condição é verdadeira. Os principais operadores estão descritos na tabela 2.

**Tabela 2: Principais operadores de comparação**

|  |  |
| --- | --- |
| x == y | x é igual a y |
| x != y | x não é igual a y |
| x < y | x é menor que y |
| x > y | x é maior que y |
| x <= y | x é menor ou igual a y |
| x >= y | x é maior ou igual a y |

Operadores lógicos são usados para comparar duas expressões. Retornam 1 ou 0 (TRUE/FALSE). A Tabela 3 mostra os principais operadores lógicos.

**Tabela 3: Principais operadores lógicos**

|  |  |  |
| --- | --- | --- |
| && | AND | porta lógica ‘E’ |
| || | OR | porta lógica ‘OU’ |
| ! | NOT | porta lógica NÃO |

***Funções matemáticas e de tempo***

As funções existentes podem ser conhecidas no site arduino.cc em *Learning*/ *Example*. A funções de tempo mais utilizada é a *delay*(ms) e a função *random*(min,max) será apresentada abaixo para serem utilizadas posteriormente nesse documento.

**delay(ms):** essa função pausa o programa por um período em milissegundos indicado pelo parâmetro em parênteses. Durante o período em que essa função está ativa qualquer outra função no programa é suspensa.

**random(min,max):** gera números pseudo-aleatórios entre os limites ‘min’ e ‘max’ especificados como parâmetros. O parâmetro ‘min’ é opcional e se excluído o limite mínimo é 0.

***Funções das portas de Entrada e Saída***

**pinMode(pino, modo)**: serve para estabelecer a direção do fluxo de informações em qualquer dos 14 pinos digitais, para o caso do Arduino Uno. Dois parâmetros devem ser passados à função: o primeiro indica qual pino vai ser usado; o segundo, se esse pino pode ser configurado como entrada (modo INPUT) ou saída digital (modo OUTPUT). Essa função é sempre escrita dentro do bloco de função ***setup()***.

**digitalRead(pino)**: lê a informação (estado lógico verdadeiro/falso) presente no pino e armazena em uma variável (HIGH/ LOW). É necessário configurar previamente o pino como entrada ou saída com a função ***pinMode()***.

**digitalWrite(pino, valor)**: serve para enviar um nível lógico para qualquer pino digital. Dois parâmetros são requeridos: o número do pino e o estado lógico (HIGH/ LOW) em que esse pino deve permanecer. É necessário configurar previamente o pino como entrada ou saída com a função ***pinMode()***.

**analogRead(pino)**: lê o nível analógico presente no pino indicado pelo parâmetro entre parênteses e, após a conversão para o seu equivalente em bits, o guarda em uma variável determinada pelo programador. Os pinos analógicos são reconhecidos tanto como A0 a A5 como 14 a 19 no Arduino Uno.

***Funções para controle de fluxo***

**if** é um controle de fluxo usado para selecionar uma ou mais instruções baseado no resultado de um teste de comparação. Todas as instruções entre as chaves **{ }** são executadas somente se o resultado desse teste for verdadeiro.

|  |
| --- |
| if (expressão) {  Bloco de instruções // executa se ‘expressão’ for verdadeiro  } |

**if...else** é um controle de fluxo para fazer um teste novo quando o resultado da expressão for falsa.

|  |
| --- |
| if (expressão) {  Bloco de instruções // executa se ‘expressão’ for verdadeiro  }  Else {  Bloco de instruções 2 // executa se ‘expressão’ for falsa  } |

**switch...case** é o comando que permite comprar uma mesma variável inteira ou uma expressão que retorne um inteiro com vários valores possíveis

|  |
| --- |
| switch (expressão) {  case 1: bloco de instruções 1;  break;  case 2: bloco de instruções 2;  break;  defaut: bloco de instruções 3;  } |

**while** é um comando em que o programa repete um grupo de instruções até que uma condição inicialmente verdadeira se torne falsa.

|  |
| --- |
| while (expressão) {  bloco de instruções;  } |

**for** é um comando parecido com o **while**, porém com um contador que registra cada execução do bloco de instruções. A ‘variável’ é inicializada normalmente com 0 ou 1; o parâmetro ‘expressão’ deve conter o valor máximo ou mínimo que o contador deve alcançar e o ‘incremento’ é o valor que será incrementado ou decrementado da variável cada vez que o bloco de instruções é executado.

|  |
| --- |
| for (variável; expressão; incremento) {  Bloco de instruções;  } |

## COMUNICAÇÃO SERIAL

O Arduino Uno é capaz de se comunicar serialmente pelas portas RX e TX. Outros modelos de Arduino possuem mais portas de comunicação serial. Esse tipo de comunicação envia dados sequencialmente pela porta TX e recebe dados sequenciais pela porta RX. É possível configurar a velocidade de comunicação. As principais funções são as seguintes:

**Serial.begin(taxa):** define a taxa de transferência em bits por segundo.

**Serial.println()**: imprime dados na porta serial como texto ASCII e pula uma linha.

**Serial.available():** pega o número de bytes (caracteres) disponíveis para leitura na porta serial.

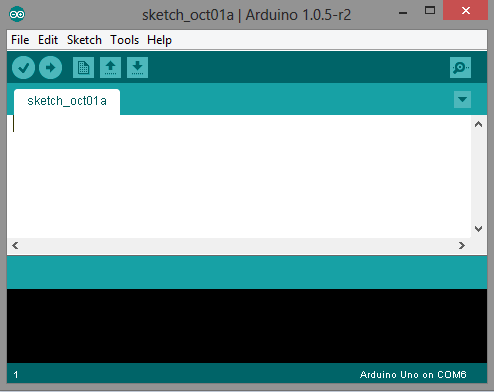
**Serial.read():** lê os dados que chegam na porta serial.

**Serial.write():** escreve dados binários na porta serial.

**Serial.flush():** espera a transmissão de dados acabar e limpa o buffer.

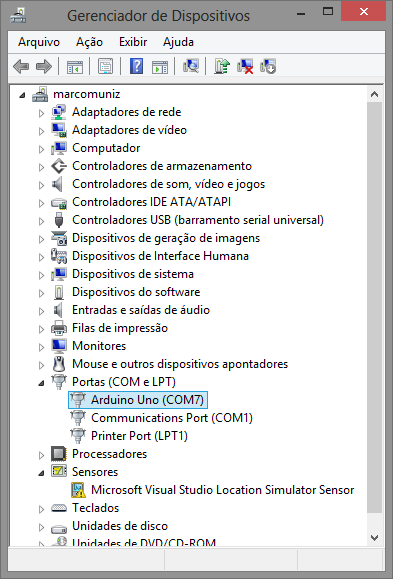
## IDE (*Integrated Development Environment*)

O download do IDE do ARDUINO é disponibilizado gratuitamente no site <http://arduino.cc/en/Main/Software>. Após efetuar o download e executar o programa será aberta uma área como a da Figura 1.



**Figura 1: IDE do Arduino.**

Para efetuar o download do programa no ARDUINO a ser utilizado clique em ***Tools => Board*** para selecionar o microcontrolador. Neste caso deve-se escolher a opção ARDUINO Uno. Para escolher por qual porta será realizada a comunicação basta clicar em ***Tools => Serial Port => COMX***, no qual ***X*** representa a porta que o ARDUINO foi instalado. O número da porta pode ser encontrado em ***Gerenciador de Dispositivos => Portas (COM e LPT)*** como mostra a Figura 2***.***



**Figura 2: Gerenciador de Dispositivos.**

O código a ser executado no microcontrolador deve conter minimamente três partes distintas:

1. ***Declaração de Variáveis***

As variáveis que serão utilizadas na parte principal devem ser declaradas como apresentado abaixo:

|  |
| --- |
| int led = 3; // Declara que uma variável do tipo inteira  char c; // Declara uma variável do tipo caractere |

1. ***Setup de rotinas***

Devem ser definidos os pinos usados como entradas ou saídas, as comunicações realizadas como apresentado abaixo:

|  |
| --- |
| void Setup() {  Serial.begin(9600); // Inicia a comunicação Serial na taxa de 9600bps  pinMode(2, OUTPUT); // Define um pino 2 como saída  pinMode(led, INPUT); // Define o pino led como entrada  } |

Sempre que o ARDUINO é resetado estas configurações são refeitas.

1. ***Programa principal***

Dentro do loop deve ser inserido toda lógica que será executada pelo ARDUINO enquanto estiver ligado. A sintaxe é apresentada abaixo:

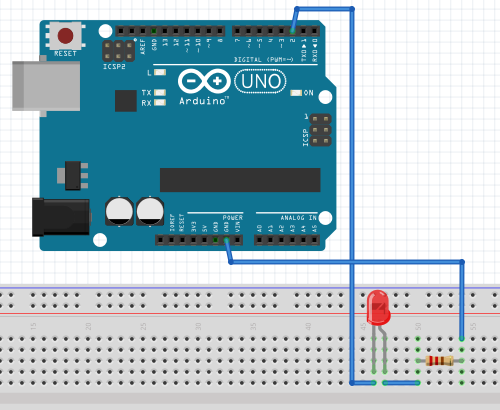
|  |
| --- |
| void loop() {  // Todo código deve ser inserido neste local  } |

***Comunicação Serial***

Conforme já mencionado, a comunicação Serial é utilizada entre o ARDUINO e um computador ou outros dispositivos. Todo ARDUINO possui pelo menos uma porta serial. A comunicação utiliza os pinos digitais 0 (RX) e 1 (TX). A utilização da comunicação serial invalida as entradas e saídas digitais dos pinos 0 e 1 no caso do ARDUINO Uno. Pode-se também utilizar o *Serial Monitor* do IDE para comunicar com o ARDUINO. Para acessá-lo basta clicar em ***Tools => Serial Monitor*.**

Os valores inseridos no Serial Monitor apesar de apresentar em decimal estão de acordo com a tabela ASCII (<http://www.asciitable.com/>). Assim ao digitar o valor ‘5’ no Serial Monitor é lido pelo ARDUINO como ‘35’ de acordo com a tabela ASCII no Anexo 1.

O código apresentado na Tabela 4 é um exemplo para utilizar a comunicação serial. Se o usuário digitar ‘2’ no Serial Monitor o microcontrolador acende um LED que é ligado ao pino 2. Se o usuário digitar ‘3’ o microcontrolador apaga este mesmo LED. O esquema da Figura 3 mostra como foi feita a ligação no *protoboard* para testar o código.



**Figura 3: Esquema para Códigos da Tabela 4 e 5.**

**Tabela 4: Código de Comunicação Serial com Arduino**

|  |
| --- |
| Código 1 utilizando o Serial Monitor |
| char c; // define uma variável do tipo char  void setup() {  pinMode(2,OUTPUT); // define o pino 2 como saída  Serial.begin(9600); // define a taxa de transmissão da comunicação serial  }  void loop()  {  if (Serial.available()>0) // Se receber algum valor pelo serial monitor  {  c = Serial.read() - '0'; // Lê o número e converte para decimal  Serial.flush(); // espera completar a transmissão dos dados e limpa o buffer  if(c==2) // Se o valor lido for igual a 2  {  digitalWrite(2,HIGH); // Coloca o pino 2 como nivel lógico alto  }  if(c==3)  {  digitalWrite(2,LOW); // Coloca o pino 2 como nível lógico baixo  }  }  } |

Uma outra forma de programar é apresentada na Tabela 5. Neste caso sempre que o usuário digitar o valor ‘2’ é verificado o *status* do pino. Se estiver em nível lógico baixo passa a ter nível logico alto (acende o LED). Se estiver em nível lógico alto passa a ter nível lógico baixo (desliga o LED).

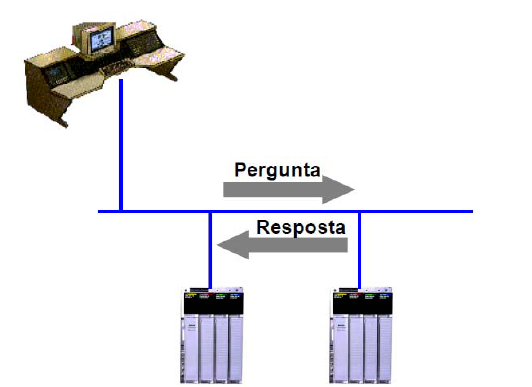
**Tabela 5: Código para Envio do Número 2 via Comunicação Serial do Arduino**

|  |
| --- |
| Código 2 utilizando o Serial Monitor |
| char c;  void setup(){  pinMode(2,OUTPUT);  Serial.begin(9600);  }  void loop()  {  if (Serial.available()>0)  {  c = Serial.read()-'0';  Serial.flush();  digitalWrite(c,!digitalRead(c)); // escreve na porta ‘c’ o oposto do sinal que o ARDUINO lê nesta porta, ou seja, se estivar HIGH escreve LOW e se estiver LOW escreve HIGH  }  } |

## MÉTODO DE ACESSO

É a forma de gerenciamento entre os pontos de comunicação (nós) da rede no tocante à comunicação de dados. São do tipo mestre/escravo e produtor/consumidor.

Em um sistema mestre/escravo (Fig.4), somente a estação chamada de mestre pode agir como detentora do direito de transmissão. Esse direito de acesso ao meio físico é distribuído por ela para as estações escravas por um determinado tempo. Tal configuração deixa o sistema dependente da estação central, mas mesmo assim é bastante utilizado em instalações industriais. Este método garante um tempo entre transmissões consecutivas a qualquer estação da rede, pela realização de um controle distribuído com supervisão centralizada (ROSARIO, 2005).



***Figura 4:* Método Mestre/Escravo (Rosario; 2005)**

Um escravo é um periférico (dispositivos inteligentes de Entrada/Saída, drivers, Interfaces Homem-Máquina, Válvulas, Transdutores, etc.), que recebe uma informação do processo e/ou utiliza informações de saída do mestre para atuar na planta. Escravos são dispositivos passivos que somente respondem a requisições diretas vindas do mestre, que pode ser:

• Monomestre: Há somente um mestre no barramento durante a operação. Geralmente a CPU do CLP é o componente do controle central. Os escravos são descentralizadamente acoplados no barramento através do meio de transmissão de dados.

• Multimestre: A imagem das entradas e saídas pode ser lida por todos os mestres, porém somente um mestre pode controlar um dado escravo (MORAES; CASTRUCCI, 2007).

Em um sistema Produtor-Consumidor os dados possuem um identificador único, origem ou destino. Todos os nós podem ser sincronizados. Usando esse modelo, múltiplos nós (produtores) podem transmitir dados para outros nós (consumidores). Também alguns nós podem assumir na rede os papéis de produtor e consumidor.

Toda essa característica operacional traz as seguintes vantagens: economia na transmissão de dados, pois eles só são enviados aos dispositivos que os requisitarem e determinismo, já que o tempo para entrega dos dados é independente do número de dispositivos que os solicitam, pois diferentemente do sistema mestre-escravo esse processo não trabalha em varredura (MORAES; CASTRUCCI, 2007).

O algoritmo de acesso ao barramento é utilizado pelos nós ou dispositivos para acessar ou disponibilizar informações na rede. Algoritmos típicos de acesso ao barramento são processos de varredura ou cíclicos, CSMA/CD (*Carrier Sense Multiple Access with Colision Detection*), *Token Passing*, mudança de estado (CoS ou *Change of State*) e CTDMA (*Code and Time-Division Multiple Access)*.

Cíclo ou varredura (*Cyclic Polling*) ocorre quando os dispositivos produtores transmitem dados a uma taxa configurada pelo usuário (entrada/saída). As características dessa forma de transmissão cíclica são:

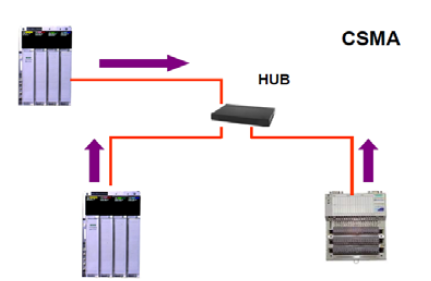
• Os dados são transferidos numa taxa adequada ao dispositivo/aplicação;

• Os recursos podem ser preservados para dispositivos com alta variação.

Esse método de troca de dados é eficiente para aplicações em que os sinais transmitidos se alteram lentamente. Como, por exemplo, sinais analógicos de entrada e saída. Por outro lado, sinais discretos cuja variação pode ser muito rápida para mudança e retorno do estado original nesse sistema de acesso poderão ter sua informação perdida.

No CSMA/CD um dispositivo começa a transmitir dados assim que detecta que o canal está disponível. Quando dois dispositivos transmitem simultaneamente, ocorre uma colisão. Com isso a transmissão irá parar e depois de um período aleatório a estação tentara retransmitir (Fig.5).

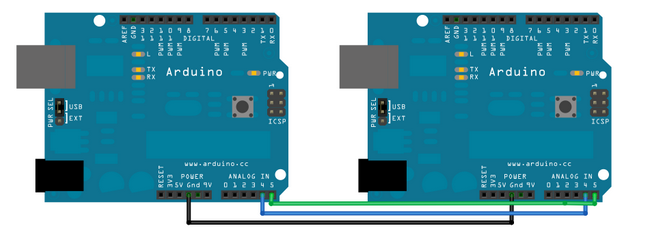
Todas as estações devem ser capazes de suportar detecção de colisão, evasão de colisão e regime de recuperação. É muito eficiente para maximizar o uso da banda disponível e oferece grande flexibilidade e opções para configuração. Foi desenvolvido para ser usado em topologia de barramento. A comunicação Ethernet é baseada nessa tecnologia (COBUS, 2003).



***Figura 5:* Método CSMA (Rosario 2005)**

MESTRE ESCRAVO I2C

Em um sistema mestre/escravo (Figura 6), somente a estação chamada de mestre pode agir como detentora do direito de transmissão. Esse direito de acesso ao meio físico é distribuído por ela para as estações escravas por um determinado tempo. Tal configuração deixa o sistema dependente da estação central, mas mesmo assim é bastante utilizado em instalações industriais. Este método garante um tempo entre transmissões consecutivas a qualquer estação da rede, pela realização de um controle distribuído com supervisão centralizada (ROSARIO, 2005).



**Figura 6 Exemplo de um esquema da Comunicação Mestre-Escravo I2C entre Dois Arduinos.**

Há diversas formas de se montar uma comunicação mestre-escravo entre Arduinos, uma delas é a do tipo I2C ou *Inter-Integrated Circuit* (Circuito Inter-integrado). Nesse caso, conforme a Figura 6, a comunicação é estabelecida de forma serial pelos mesmos pinos das entradas analógicas A4 e A5, no caso do Arduino Uno. Cada tipo de Arduino podem ter pinos distintos para essa comunicação.

Existem duas formas de implementação dessa comunicação. A primeira é denominada Mestre Escritor/ Escravo Recebedor. Nesse o escravo apenas recebe dados do mestre e executa as ações previamente implementadas em seu código. As Tabelas 6 e 7 apresentam exemplos desse código.

Tabela 6: Código para Mestre Escritor para comunicação I2C

|  |
| --- |
| Código Mestre Escritor |
| *// Wire Master Writer by Nicholas Zambetti* [*http://www.zambetti.com*](http://www.zambetti.com/) #include <Wire.h>  void setup() {  Wire.begin(); *// inicia i2c bus (endereço opc para mestre)*  } byte x = 0; void loop() {  Wire.beginTransmission(4); *// transmite ao equip #4*  Wire.write("x is ");        *// envia 5 bytes tipo char*  Wire.write(x);              *// envia 1 byte  (x=0)*  Wire.endTransmission();    *// para transmissão*  x++; *// incremente x*    delay(500); *// espera 0,5s*  } |

Tabela 7: Código para Escravo Recebedor para comunicação I2C

|  |
| --- |
| Código Escravo Recebedor |
| *#include <Wire.h>*  *void setup(){   Wire.begin(4);                // inicia i2c bus com endereço #4*  *Wire.onReceive(receiveEvent); // registra evento como escravo*  *Serial.begin(9600);           // inicia canal serial para saída*  *}* *void loop(){* *delay(100);}*  *void receiveEvent(int howMany)// define função receiveEvent*  *{   while(1 < Wire.available()) // para cada caracter*  *{char c = Wire.read(); // receibebyte como um caracter*  *Serial.print(c);         // imprime o caracter*  *}   int x = Wire.read();    // receib byte como um integer*  *Serial.println(x);         // imprime o integer* |

A segunda forma de implementação dessa comunicação é denominada Mestre Leitor/ Escravo Enviador. Nesse o escravo envia os dados ao mestre. As Tabelas 8 e 9 apresentam exemplos desse código.

Tabela 8: Código para Mestre Recebedor para comunicação I2C

|  |
| --- |
| Código Mestre Recebedor |
| *// Wire Master Writer by Nicholas Zambetti* [*http://www.zambetti.com*](http://www.zambetti.com/) #include <Wire.h>  void setup()  {   Wire.begin();        *// inicia i2c bus - mestre é opcional ter endereço*    Serial.begin(9600);  *// inicia canal serial para saída*  } void loop() {   Wire.requestFrom(2, 6);    *// requer 6 bytes do escravo #2*    while(Wire.available())    *// escravo deve enviar mensagem até o tamanho definido*    {      char c = Wire.read(); *// receibe um byte como caracter*      Serial.print(c);         *// imprime o caracter*    }   delay(500); } |

Tabela 9: Código para Escravo Enviador para comunicação I2C

|  |
| --- |
| Código Escravo Enviador |
| *#include <Wire.h>*  *void setup() {   Wire.begin(2);                // inicia i2c bus com endereço #2*  *Wire.onRequest(requestEvent); // registra evento*  *} void loop() {   delay(100); } void requestEvent() // define a função requestEvent {   Wire.write("hello "); // responde com menssagem de 6 bytes como esperado pelo mestre*  *}* |

## COMUNICAÇÃO ETHERNET

*Ethernet* é uma arquitetura de interconexão para redes locais - Rede de Área Local (LAN) - baseada no envio de pacotes. Ela define cabeamento e sinais elétricos para a camada física, e formato de pacotes e protocolos para a subcamada de controle de acesso ao meio (*Media Access Control* - MAC) do modelo OSI. A *Ethernet* foi padronizada pelo IEEE como 802.3 (barramento CSMA). A partir dos anos 90, ela vem sendo a tecnologia de LAN mais amplamente utilizada e tem tomado grande parte do espaço de outros padrões de rede como *Token Ring*, FDDI e ARCNET.1

*USER DATAGRAM PROTOCOL* (UDP)

O *User Datagram Protocol* (UDP) é um protocolo simples da camada de transporte. Ele permite que a aplicação escreva um datagrama encapsulado num pacote IPv4 ou IPv6, e então enviado ao destino. Mas não há qualquer tipo de garantia que o pacote irá chegar ou não pois não há conexão entre as partes. O UDP é utilizado em comunicações sem necessidade de segurança no envio e recebimento de dados ou quando a velocidade de comunicação é mais importante que a perda do pacote trafegado.

O protocolo UDP não é confiável. Caso garantias sejam necessárias, é preciso implementar uma série de estruturas de controle, tais como *timeouts*, retransmissões, *acknowlegments* ou aviso de recebimento, controle de fluxo dentre outros. Cada datagrama UDP tem um tamanho e pode ser considerado como um registro indivisível, diferentemente do TCP, que é um protocolo orientado a fluxos de bytes sem início e sem fim.

Também dizemos que o UDP é um serviço sem conexão, pois não há necessidade de manter um relacionamento longo entre cliente e o servidor. Assim, um cliente UDP pode criar um *socket*, enviar um datagrama para um servidor e imediatamente enviar outro datagrama com o mesmo socket para um servidor diferente. Da mesma forma, um servidor poderia ler datagramas vindos de diversos clientes, usando um único socket.

O UDP também fornece os serviços de *broadcast* e *multicast*, permitindo que um único cliente envie pacotes para vários outros na rede.

O cabeçalho UDP é extremamente simples, contendo apenas os números de porta, comprimento da mensagem e o dígito verificador ou *checksum*. O cabeçalho dos datagramas UDP é colocado a seguir ao cabeçalho IP. A Figura 7 apresenta a estrutura da mensagem Ethernet.

|  |  |  |  |
| --- | --- | --- | --- |
| Porta origem | Porta destino | Comprimento da mensagem | *Checksum* |

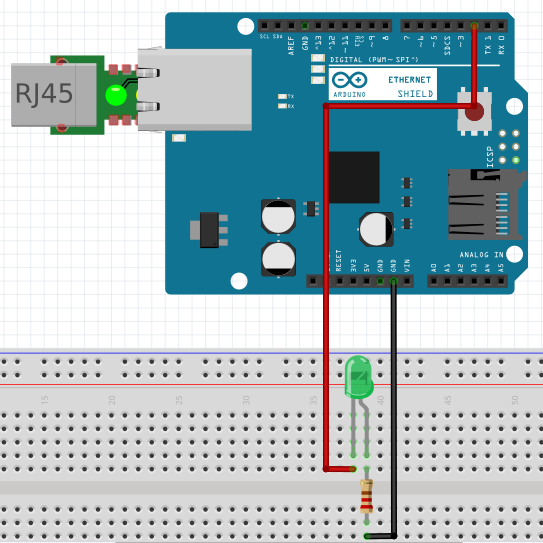
**Figura 7: Estrutura da mensagem para protocolo Ethernet UDP**

Cada campo da mensagem é composto por 2 bytes (16 bits).

Os campos em laranja são opcionais. A porta de origem geralmente especifica a porta desejada de resposta, mas pode ser omitida. Isso tipicamente ocorre em comunicações *broadcast* ou mensagens de pânico, que notificam sobre a queda de um equipamento.

***Comunicação Ethernet***

O código para acender ou apagar um LED no pino 2 por comando URL, de acordo com o esquema na Figura 8, está apresentado na Tabela 10.

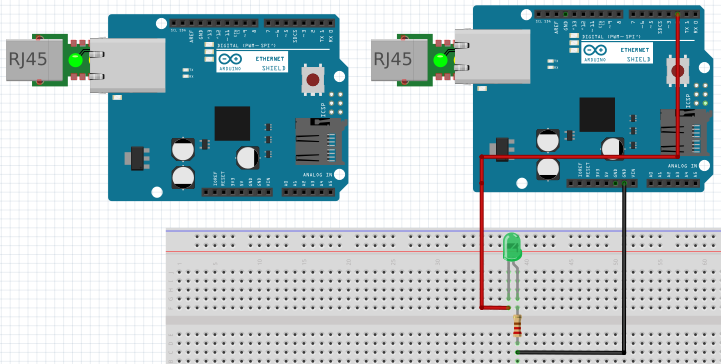


**Figura 8: Esquema do Arduino para Código da Tabela 6.**

**Tabela 10: Código para Comunicação Ethernet com Arduino.**

|  |
| --- |
| Código alterar o status de um LED através de um comando URL. |
| /\*  Permite ligar e desligar um LED por um commando em  Para ligar:  http://your-IP-address/$1  Para desligar:  http://your-IP-address/$2  \*/  #include <SPI.h> // inclui a biblioteca SPI que utiliza o arduino como mestre  #include <Ethernet.h> // inclui a biblioteca Ethernet para utilizar as funções de Ethernet  boolean incoming = 0;  // Entra com o endereço MAC e endereço de IP  byte mac[] = { 0x90, 0xA2, 0xDA, 0x00, 0x64, 0x50 };// deve ser o endereço MAC específico do Shield Ehernet do Arduino  IPAddress ip(192,168,1,121); //<<< Entre com o endereço IP aqui!!!  // (porta 80 é default para HTTP):  EthernetServer server(80);  void setup() {  pinMode(2, OUTPUT);  // inicia a conexão Ethernet e server.  Ethernet.begin(mac, ip);  server.begin();  Serial.begin(9600);  }  void loop()  {  // escuta por clients disponíveis  EthernetClient client = server.available();  if (client) {  // an http request ends with a blank line  boolean currentLineIsBlank = true;  while (client.connected()) {  if (client.available()) {  char c = client.read();  // if you've gotten to the end of the line (received a newline  // character) and the line is blank, the http request has ended,  // so you can send a reply    //reads URL string from $ to first blank space  if(incoming && c == ' '){  incoming = 0;  }  if(c == '$'){  incoming = 1;  }    //Checks for the URL string $1 or $2  if(incoming == 1){  Serial.println(c);    if(c == '1'){  Serial.println("ON");  digitalWrite(2, HIGH);  }  if(c == '2'){  Serial.println("OFF");  digitalWrite(2, LOW);  }    }  if (c == '\n') {  // you're starting a new line  currentLineIsBlank = true;  }  else if (c != '\r') {  // you've gotten a character on the current line  currentLineIsBlank = false;  }  }  }  // give the web browser time to receive the data  delay(1);  // close the connection:  client.stop();  }  } |

Na comunicação ethernet é necessário definir os endereços IP dos dispositivos. Nesse caso os IPs devem ser distintos. A Figura 9 e a Tabela 11 apresentam um exemplo de comunicação ethernet entre dois dispositivos.

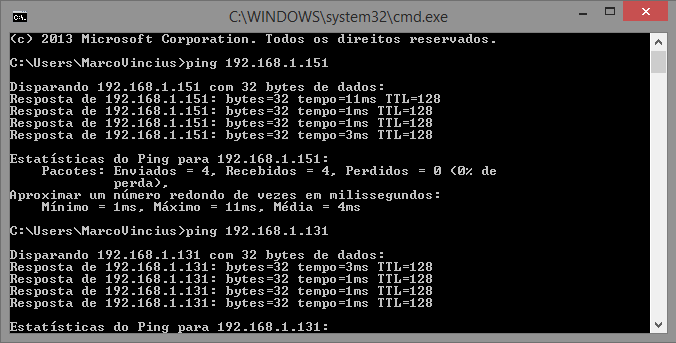


**Figura 9: Esquema da Comunicação Ethernet entre Dois Arduinos.**

**Tabela 11: Código para Comunicação Ethernet entre dois Arduinos.**

|  |  |
| --- | --- |
| **Emissor** | **Receptor** |
| #include <SPI.h>  #include <Ethernet.h>  #include <EthernetUdp.h>  EthernetUDP Udp;  byte mac[]={0x90, 0xA2, 0xDA, 0x00, 0x64, 0x44};  byte ip[] = {192,168,1,131};  byte remoteip[] = {192,168,1,151};  unsigned int port = 8888;  char packetBuffer[UDP\_TX\_PACKET\_MAX\_SIZE];  int led = 2;  void setup(){  Ethernet.begin(mac,ip);  Udp.begin(port);  Serial.begin(9600);  }  void loop(){  Udp.beginPacket(remoteip, port);  Udp.write('1');  Serial.println('1');  Udp.endPacket();  delay(3000);  Udp.beginPacket(remoteip, port);  Udp.write('2');  Serial.println('2');  Udp.endPacket();  delay(3000);    } | #include <SPI.h>  #include <Ethernet.h>  #include <EthernetUdp.h>  EthernetUDP Udp;  byte mac[]={0x90, 0xA2, 0xDA, 0x00, 0x64, 0x50};  byte ip[] = {192,168,1,151};  unsigned int port = 8888;  char packetBuffer[UDP\_TX\_PACKET\_MAX\_SIZE];  int led = 2;  void setup(){  Ethernet.begin(mac,ip);  Udp.begin(port);  Serial.begin(9600);  pinMode(led,OUTPUT);  }  void loop(){  int packetSize = Udp.parsePacket();  if(packetSize)  {  Udp.read(packetBuffer, UDP\_TX\_PACKET\_MAX\_SIZE);  Serial.println(packetBuffer[0]);  if(packetBuffer[0] == '1'){  digitalWrite(led,HIGH);  }  if(packetBuffer[0] == '2'){  digitalWrite(led,LOW);  }  }  delay(10);  } |

É possível avaliar se o dispositivo está se comunicando via ethernet através do comando ping “IP” via CMD do Windows, conforme Figura 10.



**Figura 10: A configuração dos 2 ARDUINOS foram confirmadas através do MS-DOS.**

## ZigBee

ZigBee é um dispositivo que foi criado devido inexistência de uma solução normalizada de como se operar com redes sem fios para fins como controle e telemetria, principalmente nas áreas de automação industrial e residencial.

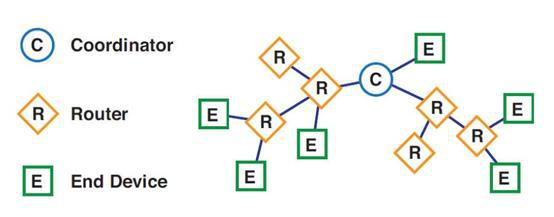
O mesmo foi concebido com o enfoque de se ter alta confiabilidade na transmissão de dados juntamente com baixo consumo, o que justifica o fato deste não possuir largura de banda elevada.

A base do padrão Wirelless Hart é o uso do ZigBee em redes industriais de campo.

Existem três tipos pré-definidos de módulos operacionais do ZigBee:

* Coordenadores (Coordinators): Estes são responsáveis por armazenar as informações principais que caracterizam a rede, como endereço da rede, chaves de segurança e outros aspectos. É importante ressaltar que existe apenas um coordenador para cada rede.
* Roteadores (Routers): Estes são responsáveis por intermediar as informações entre dispositivos, são comumente utilizados para estender a rede.
* Dispositivos de Fim (End Devices): Estes são os dispositivos utilizados no fim de malha da rede, são conectados usualmente aos sensores e enviam suas informações para os roteadores e coordenador da rede.

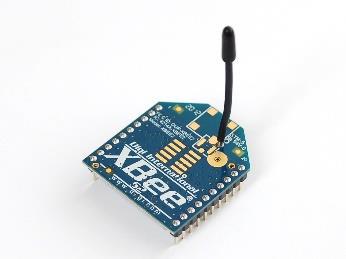
A Figura 12 apresenta um exemplo de rede com ZigBee.

**Figura 12:Ilustração de um esquema de rede com ZigBees**

Dentre os módulos definidos, o *End Device* possui uma particularidade significativa, ele apresenta um modo de baixo consumo, *Sleep Mode*. Isto indica que nos intervalos em que o mesmo está fora de operação, seu consumo energético é reduzido a baixos níveis.

Uma marca comercial de ZigBee é o XBee, a ser apresentado a seguir.

## Iniciando um XBEE

O modelo a ser utilizados na sequência é o XBee Series 2 (Figura 13), e o software XCTU clássico (XCTU legacy) para configuração dos dispositivos, lembrando que outras versões possuem o mesmo padrão de configuração. A programação do setup do Xbee se dá por meio de um dispositivo denominado explorer (Figura 14) via porta USB.

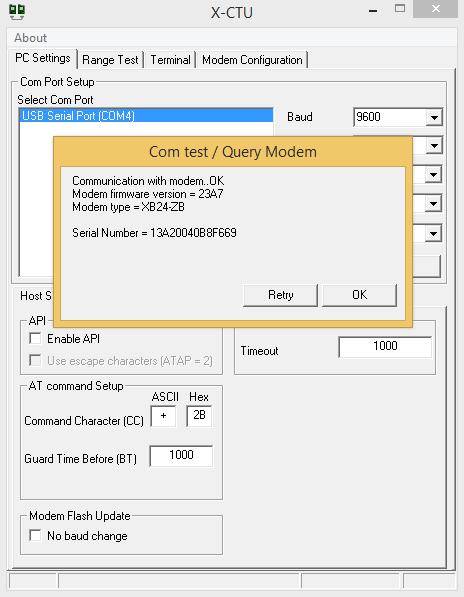
***Figura 9:* - XBee series 2**

Será utilizado também o microcontrolador arduino para exemplificação de algumas aplicações, juntamente com a biblioteca XBee. É possível também conectar o XBee a um *shield* (Figura 15) para o arduino e assim configurá-lo. Entretanto ao utilizar o microcontrolador é importante ressaltar que durante a configuração do dispositivo, a código do arduino deve estar com as funções *setup* e *main* vazias e ambos os *jumpers* devem estar conectados aos pinos centrais e direitos para configuração, sendo que para execução eles devem estar conectados aos pinos centrais e esquerdos.

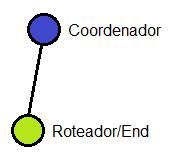
***Figura 10:* Explorer para XBee**

**Figura 15 – *Shield* XBee para Arduino**

Após o XBee estar devidamente conectado, a janela inicial do XCTU (Figura 16) deverá reconhecer a porta serial respectiva ao dispositivo. Para realizar a verificação da conexão, após pressionar “Test / Query”, uma mensagem contendo as informações do dispositivo utilizado deverá aparecer. Vale ressaltar que por definição, o XBee conectado ao *shield*, trabalha a uma taxa de transmissão de 9600.

**Figura 16 – Janela inicial do XCTU**

Alterando agora para a aba “Modem Configuration”, após pressionar “Read”, todas as informações atuais do XBee devem ser listadas.

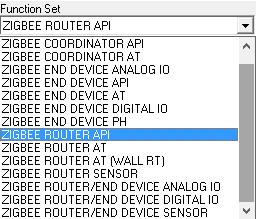
Existem mais de uma maneira de realizar a comunicação entre os dispositivos XBee. A primeira a ser abordada somente poderá ser utilizada quando a comunicação ocorrer entre dois XBees (Figura 17), e é denominada “AT Mode”.

**Figura 17 – Diagrama Esquemático de Comunicação *AT-Mode***

## Configuração *AT-Mode*

Para a criação de qualquer rede, um coordenador deve existir. Desta forma, será configurado inicialmente o coordenador.

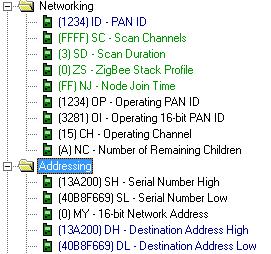
Na aba “Modem Configuration”, após listadas as configurações iniciais, na caixa “Function Set” (Figura 18), deverá ser escolhido o modo “ZIGBEE COORDINATOR AT” e pressionar “Write”. Na sequência os parâmetros da rede poderão ser definidos.

**Figura 18 – Configuração de Coordenador *AT-Mode***

Para que os dispositivos constituam uma mesma rede é necessário que todos eles estejam com o mesmo nome da rede (PAN ID), seja ele qual for.

Cada XBee possui um endereço próprio e único que os diferenciam, são compostos por 64 bits, e podem ser obtidos através do *Serial Number* apresentado na verificação “Teste / Query” ou na etiqueta presente na parte inferior do dispositivo.

Com isso, ainda na configuração do coordenador, deve ser inserido o nome da rede na seção “PAN ID” e o endereço para qual dispositivo ele deverá trocar informações, sendo que este é divido entre *High*, inserido na seção “Destination Address High”, e *Low*, inserido na seção “Destination Address Low”. Vale lembrar que feito as alterações, “Write” deverá ser pressionado para salvá-las.

Como exemplo, será definido o nome da rede “1234” e o endereço ao qual o coordenador irá enviar informações “13A20040B8F669”, sendo High “13A200” e Low “40B8F669”. Assim a configuração final deverá ficar como na Figura 19.

**Figura 10 – Configuração *AT-Mode* Final**

Finalizado o coordenador, a configuração do roteador/*end* segue a mesma sequência, o que diferencia é que na caixa “Function Set” deverá ser escolhido “ZIGBEE ROUTER AT” ou “ZIGBEE END DEVICE AT” e o endereço a ser inserido deverá ser daquele ao quais suas informações serão enviadas, logo o coordenador.

Será apresentado um exemplo realizado utilizando o microcontrolador arduino, cujo objetivo é apenas demonstrar uma maneira de enviar e receber informações através do “AT Mode”.

void setup() {

Serial.begin(9600);

}

void loop() {

Serial.write("H");

delay(100);

Serial.write("I");

delay(100);

}

Essa parte do código envia através do XBee conectado ao arduino as letras “H” e “I”.

char read;

void setup() {

Serial.begin(9600);

}

void loop() {

read = Serial.read();

Serial.println(read);

delay(100);

}

Essa outra parte do exemplo os dados são enviado através do primeiro XBee e arduino, e o segundo XBee e arduino imprimi na tela os valores lidos, neste caso, “H” e “I”.

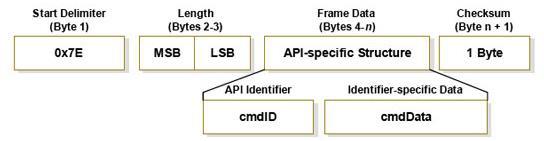
A partir desse exemplo, é visto que em “AT Mode” não é necessário inserir nenhuma função especial para realização da comunicação básica entre os dispositivos, funcionando igualmente a uma comunicação serial.

A segunda forma de comunicação é por meio da configuração *API-Mode.*

## Configuração *API-Mode*

A configuração *API-Mode* apresenta a capacidade de trabalhar com portas de entrada e saída de dados do próprio dispositivo, além de funcionar através do envio de pacotes que contém desde o endereço do dispositivo que enviou até *byte* para verificação da integridade do pacote enviado.

Cada pacote enviado é composto pelos elementos representados na Figura 20.

**Figura 20 – Quadro da Mensagem da Configuração *API-Mode***

Onde:

*Start Delimiter* – *Byte* que representa início de pacote enviado;

*Length* – *Byte* que contém o tamanho do conteúdo;

*Frame Data* – *Byte* que contém o tipo de informação (envio de dado, comandos) e o dado;

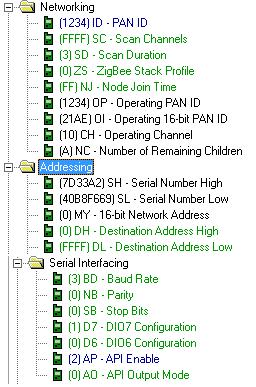
*Checksum* – *Byte* de verificação de integridade do conteúdo enviado;

Assim como no modo passado, este também precisa de um coordenador. Inicialmente então, na aba “Modem Configuration” do XCTU, depois de feito a leitura dos dados iniciais, deve-se escolher na caixa “Function Set” a opção “ZIGBEE COORDINATOR API” e pressionar “Write”.

Na sequência deve-se definir um nome de rede (PAN ID) que será comum a todos os dispositivos pertencentes a essa rede.

Neste modo, é importante compreender que o dispositivo não se comunica apenas com um outro, e sim envia informações para todos que estão na rede, definindo assim um exemplo de *BroadCast*. Para configurar esse método de comunicação, na área “Destination Address High” deve ser inserido zero e na área “Destination Address Low” deve ser inserido FFFF que é o padrão para esta forma de envio de dados.

Por último, na seção “Serial Interfacing” deve-se inserir na opção “AP – API Enable” o valor 2.

Como exemplo de configuração, será definido o nome da rede “1234”. Assim a configuração *API-Mode* final deverá ficar como na Figura 21.

**Figura 21 – Configuração *API-Mode* Final**

O exemplo de código apresentado também utiliza o microcontrolador arduino sendo necessário, desta vez, a incorporação da biblioteca XBee. O objetivo deste código é apresentar uma possibilidade de realizar a comunicação através do “API Mode”.

#include <XBee.h>

XBee xbee = XBee();

XBeeAddress64 add64 = XBeeAddress64(0x0013A200, 0x40B79B62);

uint8\_t payload[ ] = { 'H', 'I' };

Inicialmente deve-se inserir a biblioteca “XBee.h”, definir o “XBee” que possui as funções a serem utilizadas. Deve-se definir o endereço do XBee ao qual se deseja enviar alguma informação através da função

void setup() {

Serial.begin(9600);

xbee.setSerial(Serial);

}

void loop() {

ZBTxRequest data = ZBTxRequest(add64, payload, sizeof(payload));

xbee.send(data);

delay(100);

}

Nesta segunda parte do código de envio, é importante iniciar a comunicação serial e definir qual porta serial o XBee irá operar.

Dentro da função *loop*, cria-se um pacote (“data”) composto pelos elementos característicos deste modo. Na sequência utiliza-se a função “xbee.send()” para enviar o pacote.

Para o recebimento das informações, inicialmente define-se o objeto “XBee” e o “ZBRxResponse”, este responsável por receber o conteúdo do pacote, conforme abaixo.

#include <XBee.h>

XBee xbee = XBee();

ZBRxResponse rx = ZBRxResponse();

void setup() {

Serial.begin(9600);

xbee.setSerial(Serial);

}

Para se concluir o código de recebimento, cria-se uma string denominada “sample”. Através da função “xbee.readPacket”, lê-se o pacote recebido e verifica-se a integridade do mesmo, caso esteja tudo certo, utiliza-se a função “xbee.getResponse().getApiId()” para comparar com “ZB\_RX\_RESPONSE”, que é definido na biblioteca como valor que representa dado recebido (RX *Received*). Em seguida, percorre-se o vetor do dado através do “for” e armazena na string “sample”, imprimindo-a na tela depois.Define-se também, neste caso como exemplo a string “HI”, o conteúdo a ser enviado.

void loop() {

String sample;

xbee.readPacket();

if (xbee.getResponse().isAvailable()) {

if (xbee.getResponse().getApiId() == ZB\_RX\_RESPONSE) {

xbee.getResponse().getZBRxResponse(rx);

for (int i = 0; i < rx.getDataLength(); i++) {

sample += (char)rx.getData(i);

}

Serial.print(sample);

}

}

delay(100);

}

### **Exercício Prático (28 Pontos)**

Monte um sistema eletropneumático composto por 1 Botão, 2 cilindros de dupla ação (A e B), dois fins de curso (A0/A1 B0/B1), duas válvulas simples solenoide 5x2 com retorno por mola, conforme mostra a Figura 22.

O Botão deverá estar na entrada digital 2 do Arduino 1. O Arduino 1 se comunica com o Arduino 2 por meio de rede Ethernet. Os fins de curso A0 e A1 deverão estar na entrada digital 3 e 4, respectivamente, do Arduino 2 que será escravo numa rede I2C com um mestre (Arduino 3), que é responsável pela lógica de acionamento da válvula A+ (saída digital 8). Por sua vez, os fins de curso B0 e B1 deverão estar na entrada digital 5 e 6, respectivamente, do Arduino 4 que deverá ser escravo numa rede I2C do Arduino 5 vinculado com a lógica de atuação da válvula B+ (saída digital 9). Os Arduínos relacionados com as válvulas se comunicam via Xbee (Wireless).

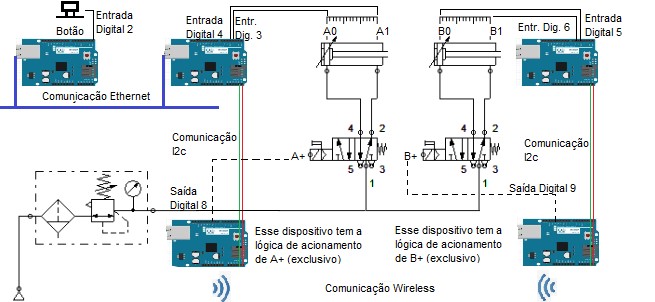
Cada grupo terá uma sequência de atuação distinta, conforme se segue:

Grupo 1: A+{A-//B+}A+{B-//A-}, Grupo 2: A-{A+//B+}A-{ B-//A+},

Grupo 3: A+{A-//B-}A+{B+//A-}, Grupo 4: A-{A+//B-}A-{B+//A+},

Grupo 5: A+{A-//B+}{A+//B-}A-, Grupo 6: A-{A+//B+}{A-//B-}A+,

Grupo 7: A+{A-//B-}{A+//B+}A-, Grupo 8: A-{A+//B-}{A-//B+}A+.



**Figura 22: Esquema eletropneumático do exercício 1**

Montagem física do sistema eletropneumático + Arduinos (4 pontos).

Teste do Sistema I2C funcionando (4 pontos).

Teste do Sistema Wireless funcionando (4 pontos).

Teste do Sistema Ethernet funcionando (4 pontos).

Sequência indicada funcionando conforme especificado (12 pontos).

**Para aqueles que não tem familiaridade com Arduino e comunicação entre dispositivos, uma apostila de autoria do discente Rubens Rodrigues Teixeira Filho está disponível.**

# BIBLIOGRAFIA

[1] http://www.asciitable.com/

[2] http://arduino.cc/

**ANEXO 1 – Tabela AscII**

