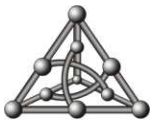
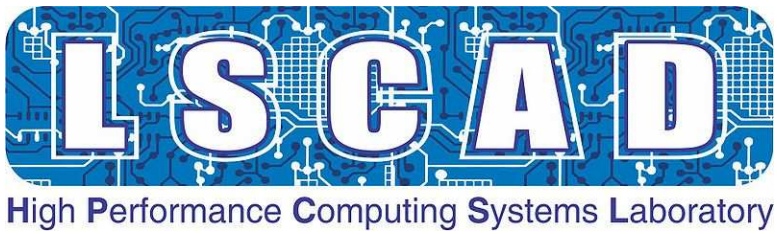
Fundação Universidade Federal de Mato Grosso do Sul
Faculdade de Computação - FACOM



Computação Física com Intel® Galileo: Conceitos Básicos e Exemplos Práticos

Equipe do Laboratório de Sistemas Computacionais de Alto Desempenho - LSCAD
Faculdade de Computação - FACOM
Universidade Federal de Mato Grosso do Sul - UFMS



Prefácio

Este material didático é voltado para a experimentação prática com exemplos que possibilitam a utilização de diferentes recursos disponíveis na plataforma Intel® Galileo. Além da da apresentação dos conceitos fundamentais para entendimento e utilização da plataforma, o material também contempla vários exemplos práticos que exploram novas funcionalidades, assim como a utilização de recursos do sistema operacional.

O material didático aqui apresentado corresponde ao esforço coletivo de professores e estudantes que, ao longo dos anos se dedicaram para preparar este material que é aberto e livre para *download* a partir do endereço `http://lscad.facom.ufms.br/wiki/index.php/Downloads`. Assim, gostaria de expressar sinceros agradecimentos às pessoas a seguir relacionadas pela contribuição direta na preparação, escrita, utilização desse material:

- Estudante de Eng. de Computação Douglas Henrique Buzeti Florido
- Estudante de Eng. de Computação Thiago Rodrigues
- Estudante de Eng. de Computação Vinicius Ribeiro Maschio

Por fim, a equipe responsável pelo desenvolvimento deste material agradece o apoio da Intel do Brasil, pela doação de kits Intel® Galileo para a UFMS. Os agradecimentos são extensíveis à UFMS através das Pró-Reitorias de Ensino de Graduação (PREG), Pesquisa e Pós-Graduação (PROPP) e Extensão e Assuntos Comunitários (PREAE) pelo suporte financeiro e bolsas concedidas para os estudantes nos diversos projetos desenvolvidos no Laboratório de Sistemas Computacionais de Alto Desempenho (LSCAD) da Faculdade de Computação (FACOM) da UFMS.

Prof. Dr. Ricardo Ribeiro dos Santos

LSCAD - FACOM - UFMS

Sumário

Capítulo 1

Descrição da Plataforma Intel® Galileo

A placa Intel® Galileo é a primeira de uma família de equipamentos baseados nas arquiteturas de processadores da Intel® projetada para que seu hardware, software e pinos sejam compatíveis com Arduino Uno* R3. Os pinos digitais 0-13, as entradas analógicas de 0 a 5, o conector de alimentação, o pino ICSP (*In-Circuit Serial Programming*), e os pinos de portas UART (*Universal Asynchronous Receiver/Transmitter*), podem ser encontradas, na placa Galileo, na mesma localização que na pinagem das placas Arduino. A placa Intel® Galileo foi projetada no intuito "faça você mesmo", com várias possibilidades de interfaceamento com componentes e sensores: servo motor, LEDs, LCD, sensor de temperatura, wifi, entre outros.

Galileo possibilita que professores, estudantes ou curiosos da área de eletrônica criem seus próprios projetos e possam desenvolvê-los com amplos recursos que esta plataforma pode oferecer para o usuário de forma totalmente aberta. Existem duas versões da placa Intel® Galileo, a Galileo e a Galileo gen 2. A Galileo gen 2 foi lançada para melhorar a

sua versão anterior trazendo aprimoramentos e novos recursos. A Figura 1.1 apresenta a placa Intel® Galileo.

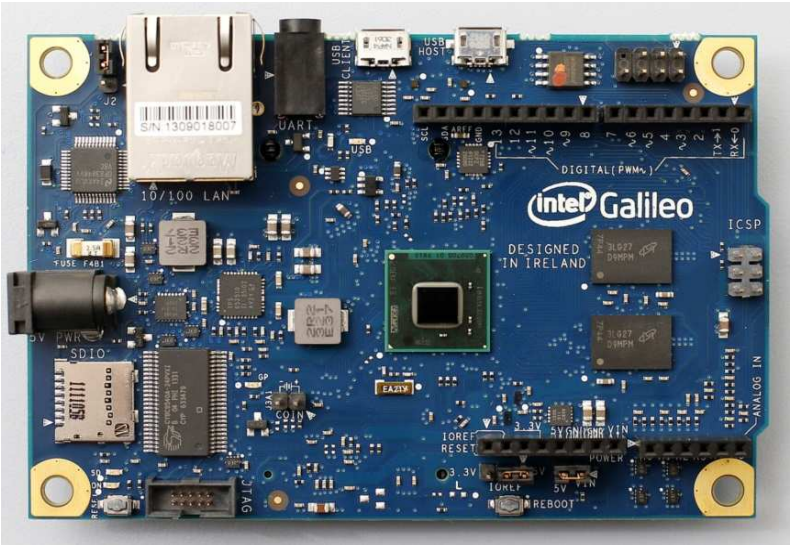


Figura 1.1: Placa Intel® Galileo.

1.1 Principais recursos da plataforma Galileo

O ambiente de desenvolvimento integrado (IDE) e a linguagem de programação disponíveis para o desenvolvimento de aplicações em Galileo é o mesmo usado no Arduino. Esse ambiente é suportado nos sistemas operacionais Microsoft Windows*, Mac OS* e Linux. Destaca-se que a IDE do Galileo é uma versão disponível em: `http://www.intel.com/support/galileo/sb/CS-035101.htm`

A Intel® se preocupou em desenvolver uma plataforma de hardware e software com facilidade de interfaceamento e suporte para ampla variedade de interfaces de E/S que são padrão do setor, inclusive um slot mini-PCI Express* de tamanho completo, porta

Ethernet de 100 Mb, slot microSD*, porta USB host e porta USB cliente e compatibilidade com Arduino Uno R3. Desta forma, possibilita que projetos desenvolvidos na plataforma Arduino possam ser executados sobre o Galileo. A Figura 1.2 detalha os recursos existentes na placa Galileo.

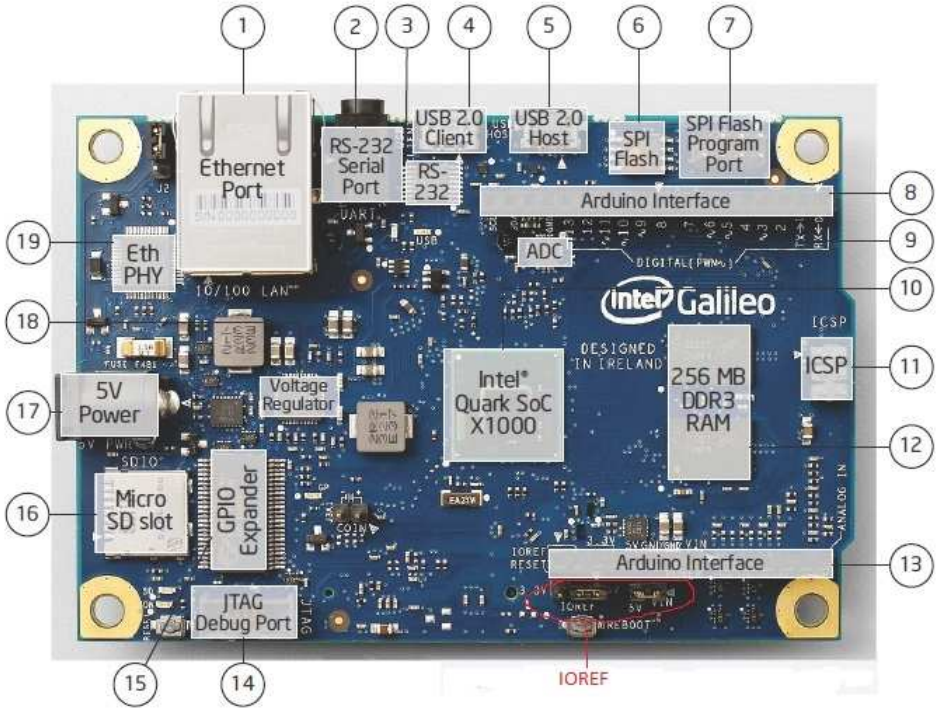


Figura 1.2: Recursos disponíveis na placa Intel® Galileo.

Contando com uma memória DDR3 de 256 MB, SRAM embarcada de 512 KB, NOR Flash de 8 MB e EEPROM padrão de 8 KB *on-board*, além de suporte para placa MicroSD com até 32 GB e um processador de aplicativos Intel® Quark SoC X1000, compatível com arquitetura de conjunto de instruções do processador Intel® Pentium de 32 bits, núcleo único e *thread* único, que opera a velocidades de até 400 MHz, permitindo alto desempenho e suporte para sistemas operacionais como Linux (Poky) e Windows.

Galileo conta com um pino IOREF que permite adaptar a voltagem que placa opera, 5V ou 3V. O controle do pino IOREF é com um *jumper* na placa.

1.2 O que há de novo com a placa Intel® Galileo Gen 2

A Figura 1.3 apresenta a nova versão da placa Galileo, chamada de gen2.

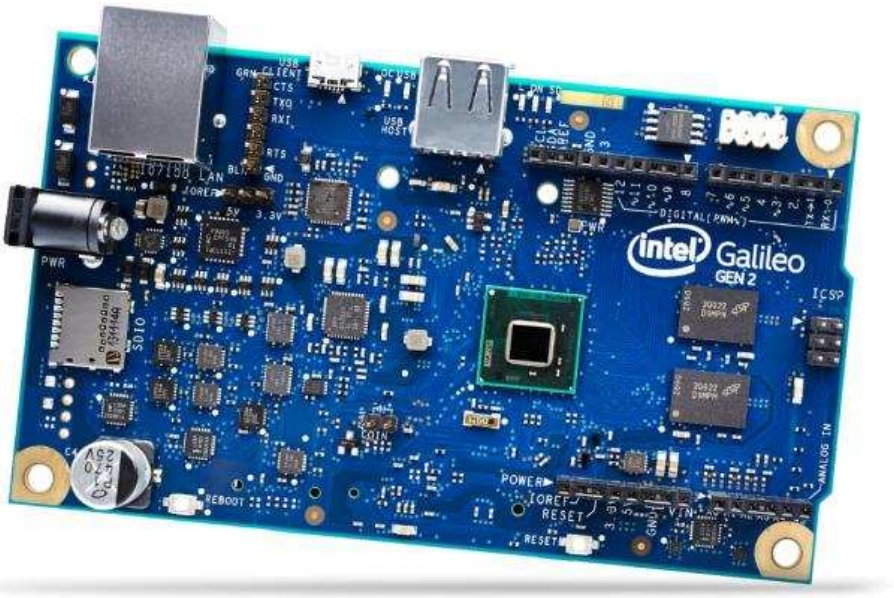


Figura 1.3: Placa Intel® Galileo geração 2.

Para melhorar o desempenho os pinos que funcionam com modulação por largura de pulso (PWM) agora operam em 12 bits para um controle mais preciso. A porta de console RS-232 com tomada de 3,5 mm para depuração Linux foi substituída pelo conector USB UART TTL de 6 pinos e 3,3V que é compatível com o cabo USB serial FTDI* padrão (TTL-232R-3V3)

Sistema de alimentação energética foi alterado para poder suportar fontes de alimentação de 7V a 15V. Adicionado suporte C, C++, Python e Node.js/Javascript para o desenvolvimento de aplicativos de Internet das Coisas com sensor conectado e capacidade de tensão de 12V sobre Ethernet (necessária instalação do módulo de PoE). Além do Yocto Linux de código aberto, a Intel® Galileo Gen 2 suporta VxWorks (RTOS) e a plataforma Wyliodrin que oferece os ambientes C, Python, Node.js e ambientes de programação visual para um navegador conectado remotamente. A versão do sistema operacional Microsoft Windows que pode ser utilizado sobre a placa Galileo recebe suporte diretamente da Microsoft®.

Capítulo 2

Introdução ao Wiring

Wiring é uma biblioteca disponibilizada junto com o ambiente de desenvolvimento do Galileo que possibilita que os programas sejam organizados através de duas funções, embora sejam programas C/C++. Essas duas funções, obrigatórias em todos os programas escritos, são:

- setup(): função que é executada uma única vez no início do programa e é usada para fazer configurações.

- loop(): função que é executada repetidamente até que a placa Galileo seja desligada.

O ambiente de desenvolvimento para o Galileo usa o conjunto de ferramentas de compilação **gnu C** e a biblioteca **AVR libc** para compilar programas. Usa ainda a ferramenta **avrdude** para carregar programas na placa.

2.1 Ambiente de desenvolvimento

O ambiente de desenvolvimento do Galileo contém um editor de texto para escrita do código, uma área de mensagem, uma área de controle de informações, uma barra de ferramentas com botões para funções comuns e um conjunto de menus. Esse ambiente se conecta ao hardware Galileo para transformar os programas e se comunicar com eles. Os programas escritos usando o ambiente de desenvolvimento Galileo são chamados de *sketches*. O ambiente de desenvolvimento (figura 2.1) foi desenvolvido em Java e é derivado do ambiente de desenvolvimento para a linguagem *Processing*.

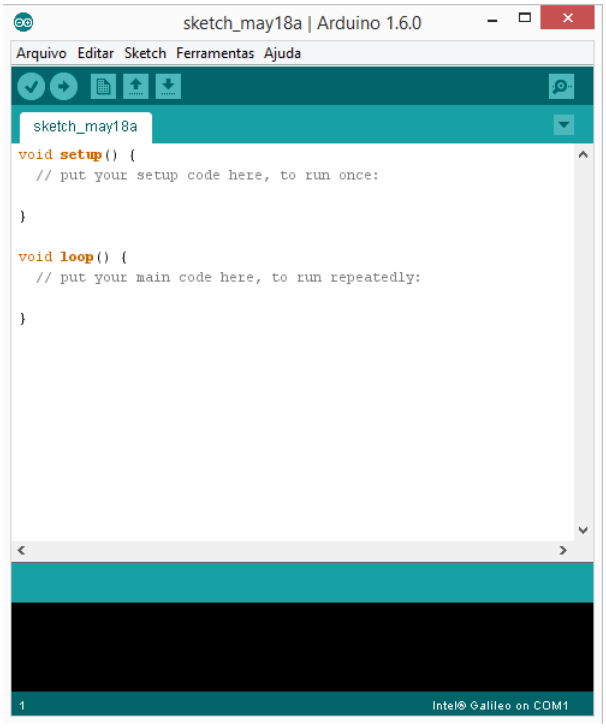


Figura 2.1: Ambiente de desenvolvimento (IDE) do Galileo.

A versão do IDE que possui suporte para o Galileo se encontra apenas na seção suporte do site da Intel® no *link*: `http://www.intel.com/support/galileo/sb/CS-035101.htm`

Todos os comandos são encontrados através dos menus: Arquivo, Editar, Sketch, Ferramentas, Ajuda. As funções disponíveis nos menus *Arquivo*, *Editar* e *Ajuda* são semelhantes às de outros programas bem conhecidos e, por isso, não serão detalhadas aqui.

Principais comandos disponíveis através de botões:

(a) **Verificar/Compilar** - Verifica se o código tem erros.

(b) **Carregar** - Copila o código e transfere para o Galileo.

(c) **Novo** - Cria um novo *sketch*.

(d) **Abrir** - Mostra uma lista de todos os *sketch* de exemplos e a opição de abrir a pasta onde esta todos os *sketch* salvos, abre o que for selecionado.

(e) **Salvar** - Salva o *sketch*.

(f) **Monitor Serial** - Mostra as informações obtidas pelo Galileo via porta serial.

Menu *Sketch*

- **Verificar/Copilar** - Verifica se o código tem erros.
- **Mostrar página do *Sketch*** - Abre a pasta onde o programa está salvo.
- **Adicionar Arquivo** - Adiciona um arquivo fonte ao programa. O novo arquivo aparece em uma nova aba.
- **Importar Biblioteca** - Apresenta várias bibliotecas que podem ser atribuídas ao programa e inclui a selecionada.

Menu Ferramentas

- **Auto Formatação** - Formata o texto para uma melhor leitura, alinhando as chaves indentando seu conteúdo.
- **Arquivar *Sketch*** - Salva a *Sketch* atual.
- **Corrigir codificação e carregar** - Carrega a última *Sketch* salva descartando qualquer alteração feita.
- **Monitor Serial** - Mostra as informações obtidas pelo Galileo via porta serial.
- **Placa** - Seleciona o modelo de placa Galileo utilizada no projeto.
- **Gravar *Bootloader*** - Permite gravar um *bootloader* no Galileo.

2.2 Conceitos e sintaxe da linguagem de programação para Galileo

Como já citado, a linguagem de programação do Galileo é baseada nas linguagens C/C++, preservando sua sintaxe na declaração de variáveis, na utilização de operadores, na manipulação de vetores, na conservação de estruturas, sendo também *case sensitive*. Contudo, ao invés de uma função *main()*, o programa necessita de duas funções elementares: *setup()* e *loop()*.

Pode-se dividir a linguagem de programação para Galileo em três partes principais: as variáveis e constantes, as estruturas e, por último, as funções.

2.2.1 Elementos de sintaxe

- **;** (*ponto e vírgula*) - Sinaliza a separação e/ou finalização de um comando.

  Sintaxe:

  comando ;

- **{ }** - Chaves são utilizadas para delimitar um bloco de instruções referente a uma função(*setup*, *loop*...), a um laço(*for*, *while*...), ou ainda, a uma sentença condicional(*if...else*, *switch case*...).

  Sintaxe:

função/laço/sentença-condicional{

comando;

}

- // (*linhas de comentários simples*) - O conteúdo após // até o final dessa linha, é ignorado pelo compilador e considerado um comentário. O propósito dos comentários é ajudar a entender (ou relembrar) como o programa funciona.

  Sintaxe:

  comando // aqui, todo comentário é ignorado pelo compilador

- /* */(*bloco de comentários*) - Tem por finalidade comentar trechos de código no programa. Assim como as *linhas de comentários simples*, o bloco de comentários geralmente é usado para resumir o funcionamento do que o programa faz e para para comentar e descrever funções. Todo conteúdo inserido dentro /* */ também é desconsiderado pelo compilador.

  Sintaxe:

  /* Use o bloco de comentários para descrever,

  comentar ou resumir funções

  e a funcionalidade do programa.

  */

- ♯**define** - Permite dar um nome a uma constante antes que o programa seja compilado. Constantes definidas no Galileo não ocupam espaço na memória. O compilador substitui referências a estas constantes pelo valor definido. Não deve-se usar ponto-e-virgula (;) após a declaração ♯*define* e nem inserir o operador de atribuição "= ", pois isso gerará erros na compilação.

  Sintaxe:

  ♯ define nome_constante constante

- ♯**include** - É usado para incluir outras bibliotecas no programa. Isto permite acessar um grande número de bibliotecas padrão da linguagem C (de funções pré-definidas), e também as bibliotecas desenvolvidas especificamente para Galileo. De modo similar ao ♯*define*, não deve se usar ponto-e-virgula (;) no final da sentença.

  Sintaxe:

  ♯include <nome_da_biblioteca.h>
  ou
  ♯include "nome_da_biblioteca.h"

Exemplos de programas com utilização dessa sintaxe serão apresentados posteriormente nesta apostila.

2.2.2 *Setup* e *Loop*

Todos programa criado para Galileo deve obrigatoriamente possuir duas função para que o programa funcione corretamente: a função *setup()* e a função *loop()*. Essas duas funções não utilizam parâmetros de entrada e são declaradas como *void*. Não é necessário invocar a função *setup()* ou a função *loop()*. Ao compilar um programa para Galileo, o compilador irá, automaticamente, inserir uma função *main* que invocará ambas as funções.

setup()

A função *setup* é utilizada para iniciar variáveis, configurar o modo dos pinos e incluir bibliotecas. Esta função é executada automaticamente uma única vez, assim que o kit Galileo é ligado ou resetado.
Sintaxe:

```
void setup()
{
.
:
}
```

loop()

A função *loop()* faz exatamente o que seu nome segure: entra em um *looping*(executa

sempre o mesmo bloco de códigos), permitindo ao programa executar as instruções que estão dentro desta função. A função *loop()* deve ser declarada após a função *setup()*.
Sintaxe:

```
void loop()
{
.
:
}
```

2.2.3 Variáveis e constantes

Variáveis são áreas de memória, acessíveis por nomes, que pode-se usar em programas para armazenar valores, por exemplo, a leitura de um sensor conectado em uma entrada analógica.

2.2.4 Tipos de dados

As variáveis podem ser de vários tipos:

- ***boolean*** - Variáveis *boolean* podem ter apenas dois valores: *true*(verdadeiro) ou *false*(falso).

Sintaxe:

```
boolean variável = valor;
// valor = true ou false
```

Exemplos:

```
boolean teste = false;
...
if (teste = true)
i++;
...
```

- **byte** - Armazena um número de 8 bits sem sinal (*unsigned*), de 0 a 255.

Sintaxe:

```
byte variavel = valor;
```

Exemplos:

```
byte x = 1 ;
byte b = B10010;
//B indica o formato binário
//B10010 = 18 decimal
```

- **char** - É um tipo de variável que ocupa 1 byte de memória e armazena o código ASCII de um caractere. Caracteres literais são escritos com ' ' (aspas simples) como: 'N'. Para cadeia de caracteres utiliza-se " " (aspas duplas) como: "ABC".

Sintaxe:

```
char variável = '65';
char variável = "A";
```

Exemplos:

```
char mychar = 'N';
//mychar recebe o valor 78,
//correspondendo ao carácter
//'N' segundo a tabela ASCII
```

- **unsigned int** - Inteiros sem sinal permitem armazenar valores de 2 bytes. Entretanto, ao invés de armazenar números negativos, armazena somente valores positivos

na faixa de 0 a 65.535. A diferença entre inteiros e inteiros sem sinal está no modo como o bit mais significativo é interpretado. No Galileo, o tipo int (que é com sinal) considera que, se o bit mais significativo for 1, o número é interpretado como negativo e se for 0 como positivo. Os tipos com sinal representam números usando a técnica chamada **complemento de 2**.

Sintaxe	Exemplo:
unsigned int var = val ;	unsigned int ledPin = 13;

- ***long*** - Variáveis do tipo *long* têm um tamanho ampliado para armazenamento de números, sendo capazes de armazenar 32 bits (4 bytes), de -2.147.483,648 a 2.147.483.647.

Sintaxe:	Exemplos:
long variavel = valor;	long exemplo = -1500000000 long exemplo2 = 2003060000

- **unsigned long** - longs sem sinal são variáveis de tamanho ampliado para armazenamento numérico. Diferente do tipo long padrão, esse tipo de dado não armazena números negativos, abrangendo a faixa de 0 a 4.294.967.295.

Sintaxe:	Exemplo:
unsigned long variável = valor;	unsigned long var = 3000000;

- **float** - Tipo de variável para números de "ponto flutuante" que possibilita representar valores reais muito pequenos e muito grandes. Números do tipo float utilizam 32 bits e abrange a faixa de -3,4028235E+38 a 3,4028235E+38.

Sintaxe:

float variável = valor;

Exemplo:

float sensorCalibrate = 1.117;

- **double** - Número de ponto flutuante de precisão dupla. A implementação do double no Galileo é a mesma que do float , sem ganho de precisão, ocupando 4 bytes(32bits) também.

  Sintaxe:

  double variável = valor;

  Exemplo:

  double x = 1.117;

- **array** - Um array (vetor) é uma coleção de variáveis do mesmo tipo que são acessados com um índice numérico. Sendo a primeira posição de um vetor V a de índice 0(V[0]) e a última de índice n-1 (V[n-1]) para um vetor de n elementos.
  Um vetor também pode ser multidimensional (*matriz*), podendo ser acessado como: V[m][n], tendo m x n posições. Assim, tem-se a primeira posição de V com índice 0,0 (V[0][0]) e a última sendo m-1, n-1 (V[m-1][n-1]).
  Um *array* pode ser declarado sem especificar seu tamanho.

Sintaxe:

```
tipo_variável var[ ];
tipo_variável var[ ] = valor;
tipo_variável var[índice]= valor;
tipo_variável var[ ][ ];
tipo_variável var[ ][índice] = valor;
tipo_variável var[índ][índ] = valor;
```

Exemplo:

```
int var[6];
int myvetor[ ] = 2, 4, 8, 3, 6;
int vetor[6] = 2, 4, -8, 5, 2;
char massagem [6] = "hello";
int V[2][3] = {0 1 7
               3 1 0};
int A[2][4] = {2 7}
              {2 3 5 6};
```

- **string** - Strings são representadas como um vetor do tipo char e terminadas em *null*. Por serem terminadas em *null* (código ASCII 0), permitem as funções (como Serial.print()) saber onde está o final da string.

Sintaxe:

```
tipo_variável var[índice] = valor;
```

Exemplo:

```
char Str1[15];
char Str2[6] = {'i','n','t','e','l'};
char Str3[6] = {'i','n','t','e','l','\0'};
char Str4[ ] = "galileo";
char Str5[6] = "intel";
```

Como apresentado no exemplo anterior Str2 e Str5 precisam ter 6 caracteres, embora "intel"tenha apenas 5. A última posição é automaticamente preenchida com o

caractere *null*. Str4 terá o tamanho determinado automaticamente como 8 caracteres, um extra para o *null*. Na Str3 foi incluído o caractere *null* (escrito como "\0"). Na Str1 definiu-se uma string com 15 posições não inicializadas, lembrando que a última posição, correspondente à posição 15, é reservado para o caractere *null*.

- **void** - A palavra chave *void* é usada apenas em declarações de função. Ela indica que a função não possui informação de retorno à função que a chamou. Como exemplo com funções declaradas com retorno *void* tem-se as funções *setup* e *loop*.

Sintaxe:

```
void nome_função ( )
void nome_função (parâmetros)
```

Exemplos:

```
void setup ()
{
.
:
}

void loop()
{
.
:
}
```

2.2.5 Constantes

Constantes são nomes com valores pré-definidos e com significado específicos que não podem ser alterados na execução do programa. Ajudam a deixar o programa mais facilmente legível. A linguagem de programação para Galileo oferece algumas constantes acessíveis aos usuários.

Constantes booleanas (verdadeiro e falso)

Há duas constantes usadas para verdadeiro e falso na linguagem Galileo: *true* (verdadeiro), e *false* (falso).

- ***false*** - *false* é a mais simples das duas e é definida como 0 (zero).
- ***true*** - *true* é frequentemente definida como 1, o que é correto, mas true tem uma definição mais ampla. Qualquer inteiro que não é zero é ***true***, num modo booleano. Assim , -1, 2, -200 e 70 são definidos como *true*.

INPUT e *OUTPUT*

Pinos digitais podem ser configurados como *INPUT* ou *OUTPUT*. Mudar um pino de *INPUT* para *OUTPUT* com *pinMide( )* muda drasticamente o seu comportamento elétrico.

- ***INPUT*** - Os pinos do Galileo configurados como *INPUT* com a função *pinMode()* estão em um estado de alta impedância. Pinos de entrada são usados para ler um

sensor mas não para energizar um LED.

- OUTPUT - Pinos configurados como *OUTPUT* com a função *pinMode( )* estão em um estado de baixa impedância. Isto significa que eles podem fornecer grande quantidades de corrente para outros circuitos. Os pinos do Galileo podem fornecer (corrente positiva),até 10 mA (milliamperes), ou drenar (corrente negativa), até 25 mA (milliamperes) de/para outros dispositivos ou circuitos. Isto faz com que eles sejam úteis para energizar um LED mais inapropriado para a leitura de sensores. Pinos configurados como *OUTPUT* também podem ser danificados ou destruídos por curto-circuito com o GND ou com outros pontos de 5 V. A Quantidade de corrente fornecida por um pino do Galileo também não é suficiente para ativar muitos relês e motores e, neste caso, algum circuito de interface será necessário.

HIGH* e *LOW

Quando se está lendo ou escrevendo em um pino digital há apenas dois valores que um pino pode ter: *HIGH* (alto) e *LOW* (baixo).

- ***HIGH*** - O siginficado de *HIGH* (em referência a um pino) pode variar um pouco dependendo se este pino é uma entrada (*INPUT*) ou saída *(OUTPUT*). Quando um pino é configurado como *INPUT* com a função *pinMode*( ) e lido uma como uma função *digitalRead*( ), o microcontrolador considera como *HIGH* se a tensão for 3 V ou mais. Um pino também pode ser configurado como INPUT, e posteriormente receber um *HIGH* com um *digitalWrite*( ), isto vai "levatar"o resistor inteiro de

10 KOhms que vai manter a leitura do pino como *HIGH* a não ser que ela seja alterada para *LOW* por um circuito externo. Quando um pino é configurado como *OUTPUT*, e definimos *HIGH* com o *digitalWrite*( ), ele fica com 5 V. Neste estado ele pode enviar corrente para, por exemplo, acender um LED que está conectado com um resistor em série ao GND, ou a outro pino configurado como *OUTPUT* e definido como *LOW*.

- ***LOW*** - O significado de *LOW* também pode variar dependendo do pino ser definido como *INPUT* ou OUTPUT. Quando um pino é configurado como INPUT com a função pinMode( ), e lido com a função *digitalRead*( ), o microcontrolador considera como *LOW* se a tensão for 2 V ou menos. Quando um pino é configurado como *OUTOUT*, e definido como *LOW*, ele fica com 0 V. Neste estado ele pode "drenar"corretamente para, por exemplo, acender um LED que está conectado com um resistor em série ao +5 V, ou a outro pino configurado como *OUTPUT* e definido como *HIGH*.

2.2.6 Conversão

Converte de	Para	Sintaxe
boolean	char	char(variável)
int		
long		
float		
double		
char	int	int(variável)
boolean		
long		
float		
double		
char	float	float(variável)
boolean		
long		
int		
double		
char	double	double(variável)
boolean		
long		
int		
float		
char	boolean	boolean(variável)
float		
long		
int		
double		
char	long	long(variável)
float		
boolean		
int		
double		

2.2.7 Estrutura de controle

- **if** - Estrutura com finalidade de verificar se uma condição é verdadeira. Em caso afirmativo, executa-se um bloco do código com algumas instruções. Caso ao contrário, o programa não executa o bloco de instruções e pula o bloco referente a essa estrutura.
Sintaxe:

```
if(condição) {
bloco de instruções ;
}
```

Exemplo:

```
if(x > 120) {
int y = 60;
}
```

- **if...else** - Permite um controle maior sobre o fluxo de código do que a sentença *if* básica. Quando usa-se a estrutura *if...else* garante-se que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas. Caso a condição if seja satisfeita, executa o bloco de instruções referente ao if, caso contrário, executa-se obrigatoriamente o bloco de instruções referente ao else.
Sintaze:

```
if(condição){
bloco de instruções 1
}
else {
bloco de instruções 2
}
```

Exemplo:

```
if(x <= 100) {
int y = 35;
}
else {
int x = 500;
}
```

- **switch case** - Permite construir uma lista de "casos" dentro de um bloco delimitado por chaves ({ }). O programa verifica cada caso com a variável de teste e executa determinado bloco de instruções se encontrar um valor idêntico. A estrutura *switch case* é mais flexível que a estrutura *if...else* já que pode-se determinar se a estrutura deve continuar verificando se há valores idênticos na lista de "casos" após encontrar um valor idêntico, ou não. Deve-se utilizar a sentença *break* após a execução do bloco de códigos selecionado por um dos "casos". Nesta situação, se uma sentença break é encontrada, a execução do programa "sai" do bloco de códigos e vai para próxima instrução após o bloco **switch case**. Se nenhuma condição for satisfeita, o código que está no *default* é executado. O uso de *default*(uma instrução padrão) é opcional no programa.

  Sintaxe:

```
switch (variável){
    case 1:
        instrução p/ quando variável == 1
        break;
    case 2:
        instrução p/ quando variável == 2
        break;
    default:
        instrução padrão
}
```

  Exemplos:

```
switch (x) {
    case 1:
        y = 100;
        break;
    case 2:
        y=158;
        break;
    default:
        y=0;
}
```

- **for** - É utilizado para repetir um bloco de código delimitado por chaves. Um contador com incremento/decremento normalmente é usado para controlar e finalizar o laço. A sentença *for* é útil para qualquer operação repetitiva. Há três partes no cabeçalho de um *for*:

  for(inicialização;condição;incremento)

  A *inicialização* ocorre primeiro e apenas uma vez. Cada vez que o laço é executado, a *condição* é verificada, se verdadeira, o bloco de códigos é executado. Em seguida, o *incremento* é realizado, e então a condição é testada novamente. Quando a condição se torna falsa o laço termina.

  Sintaxe:

  ```
  for (inicializa;condição;incremento)
  {
  bloco de instruções
  }
  ```

  Exemplo:

  ```
  for(int i=0; i<=255; i++){
      char str[i] = i;
      .
      :
  }
  ```

- **while** - Permite executar um bloco de código entre duas chaves repetidamente inúmeras vezes até que a *condição* se torce falsa. Essa *condição* é uma sentença booleana em C que pode verificada como verdadeira ou falsa.

Sintaxe :

```
while(condição){
bloco de instruções ;
}
```

Exemplo :

```
int i = 0;
while (i < 51){
    .
    :
    i++;
}
```

- **do...while** - Funciona da mesma maneira que o *while*, com a exceção de que agora a condição é testada no final do bloco de código. Enquanto no *while*, se condição for falsa, o bloco de código não será executado, no *do...while* ele sempre será executado pelo menos uma vez.

Sintaxe :

```
do
{
    bloco de instruções;
}whilecondição;
```

Exemplo :

```
int x = 50;
do {
    .
    :
    x–;
} while (x > 0);
```

- **continue** - È usado para saltar porções de códigos em comandos como *for*, *do...while* e *while*. Ele forca com que o código avance até o teste da condição, saltando todo o

resto.

Sintaxe usando o while :

```
while (continue){
    bloco de instruções;
    if (condição)
      continue;
    bloco de instruções;
}
```

Exemplo de trecho de código usando for:

```
for(x = 0; x < 255; x++){
   if(x > 40 && x < 120)
     continue;
   .
   :
}
```

- **break** - É utilizada para sair de um laço *do...while*, *for*, *while* ou *switch case*, se sobrepondo a condição normal de verificação.

Sintaxe usando do...while :

```
do{
   bloco de instruções;
   if(condição)
     bloco de instruções;
     break;
} while (condição);
```

Exemplo de trecho de código usando while :

```
x = 1;
while ( x < 255 ) {
    y = 12/x;
    if(y < x)){
      x = 0;
      break;
    }
    x++;
}
```

- **return** - Finaliza uma função e retorna um valor, se necessário. Esse valor pode ser uma variável ou uma constante.

  Sintaxe :

  return;

  ou

  return valor;

  Exemplo :

```
if (x = 255)
  return 0;
else
  return 1;
```

2.2.8 Operadores de comparação

Os operadores de comparação, como o próprio nome sugere, permitem que se compare dois valores. Em qualquer das expressões na Tabela 2.2 o valor retornado sempre será um valor booleano. É possível realizar comparações entre números, caracteres e booleanos. Quando se utiliza um caractere na comparação, o código ASCII desse caractere é considerado. Para comparar um array com outro tipo de dado, deve-se indicar uma posição do array para ser comparado

Tabela 2.2: Operadores de Comparação

Operando Direito	Operador	Operando Esquerdo	retorno
boolean	==	boolean	
int	!=	int	
float	<	float	boolean
double	>	double	
char	<=	char	
array[ ]	>=	array[ ]	

2.2.9 Operadores de atribuição

O operador de atribuição (ou operador de designação) armazena o valor à direita do sinal de igual na variável que está à esquerda desse sinal. Esse operador também indica ao microcontrolador para calcular o valor da expressão à direita e armazenar este valor na variável que está à esquerda.

x = y; (a variável x armazena o valor de y)

Tabela 2.3: Operadores Aritméticos com **int**

Operando Direito	Operador	Operando Esquerdo	retorno
int	+	int	int
	-	double	
	*	float	
	/	char	
	%	int	
		char	

Tabela 2.4: Operadores Aritméticos com **char** / **array**

Operando Direito	Operador	Operando Esquerdo	retorno
char	+	int	char
	-	double	
	*	float	
	/	char	
	%	int	
		char	

a = b + c; (calcula o resultado da soma e coloca na variável a)

2.2.10 Operadores aritméticos

Os operadores aritméticos são usados para realizar operações matemáticas.

Tabela 2.5: Operadores Aritméticos com **double** / **float**

Operando Direito	Operador	Operando Esquerdo	retorno
float	+	int	float
	-	double	
	*	float	
double	/	char	double

2.2.11 Operadores booleanos

Os operadores booleanos (ou operadores lógicos) são geralmente usados dentro de uma condição *if* ou *while*. Em geral, os operandos da expressão podem ser números, expressões relacionais e sempre retornam como resposta um valor lógico: Verdadeiro (1) ou Falso (0).

- **&&** (e) exige que os dois operandos sejam verdadeiros para ser verdade, ou seja, a primeira condição "e"a segunda devem ser verdadeiras;

- || (ou) para ser verdadeiro, basta que um dos operando seja verdade, ou seja, se a primeira condição "ou"a segunda "ou"ambas é(são) verdadeira(s), então o resultado é verdadeiro;

- **!** (não) é verdadeiro apenas quando o operando for falso.

2.2.12 Operadores de bits

Os operadores de bits realizam operações ao nível de bits das variáveis. Esses operadores operam somente em dados do tipo char ou int.

- **&** (operador AND bit a bit) é usado entre duas variáveis/constantes inteiras. Ele realiza uma operação entre cada bit de cada variável de acordo com a seguinte regra: se os dois bits de entrada forem 1, o resultado da operação também é 1, caso contrário é 0.

Exemplo :

```
0   0   1   1   a
0   1   0   1   b
_____________
0   0   0   1   ( a & b )
```

&	0	1
1	0	1
0	0	0

- | (operador OR bit a bit) realiza operações com cada bit de duas variáveis conforme a seguinte regra: o resultado da operação é 1 se um dos bits de entrada for 1, caso contrário é 0.
  Exemplo :

```
0   0   1   1   c
0   1   0   1   d
_____________
0   1   1   1   ( c | d )
```

|	0	1
1	1	1
0	0	1

- ^ (operador XOR bit a bit) Conheido omo Exclusive or (ou exclusivo), esse operador realiza uma operação entre cada bit de cada variável de acordo com a seguinte regra: se os dois bits de entrada forem diferentes, o resultado desta operação é 1 , caso contrário, retorna 0.
  Exemplo :

```
0   0   1   1   e
0   1   0   1   f
_____________
0   1   1   0   ( e ^ f )
```

^	0	1
1	1	0
0	0	1

- ~ (operador de bits NOT) diferente dos operadores AND, OR e XOR, este operador é aplicado apenas sobre um operando, retornando o valor inverso de cada bit.
  Exemplo :

  0 1 g

  ———

  1 0 (~g)

~	
0	1
1	0

- < < (deslocamento à esquerda) Desloca para a esquerda os bits do operando esquerdo conforme o valor dado pelo operando direito.
  Exemplo :

  int a = 3;

  int x = a « 3;

  0 0 0 0 1 1 a

  0 1 1 0 0 0 a « 3

byte	< <	retorno
00000001	2	00000100
00000101	3	00101000

- > > (deslocamento à direita) Desloca, para a direita, os bits do operando esquerdo conforme o valor dado pelo operando direito.
  Exemplo :

  int b = 40;

  int y = a » 3;

  0 1 0 1 0 0 0 b

  0 0 0 0 1 0 1 b » 3

byte	> >	retorno
00001000	2	00000010
00001001	3	00000001

2.2.13 Operadores compostos

Os operadores compostos consistem em um recurso de escrita reduzida provido pela linguagem C, havendo sempre a possibilidade de obter-se o resultado equivalente através do uso de operadores simples.

Incremento e decremento

Os incrementos (++) e decrementos (- -) podem ser colocados antes ou depois da variável a ser modificada. Se inseridos antes, modificam o valor antes da expressão ser usada e, se inseridos depois, modificam depois do uso.

- ++ (incremento) aumenta o valor de variáveis em uma unidade;

  Exemplo :

  int x = 2 ;
  int var = ++x;

  o valor de var será 3 e o de x será 3.

  int y = 2;
  int var2 = y++;

  o valor de var2 será 2 e o de y será 3

- – (decremento) diminui o valor de variáveis em uma unidade;

Exemplo :

int x = 7 ;	int y = 7;
int var = –x;	int var2 = y–;
o valor de var será 6 e o de x será 6.	o valor de var2 será 7 e o de y será 6

- += (adição composta) realiza a adição de uma variável com outra constante ou variável.
  Exemplo :

	x = 2;
x += y;	x += 4;
equivalente à expressão x = x + y	x passa a valer 6

- -= (subtração composta) realiza a subtração de uma variável com outra constante ou variável.
  Exemplo :

	x = 7;
x -= y;	x -= 4;
equivalente à expressão x = x - y	x passa a valer 3

- *= (multiplicação composta) realiza a multiplicação de uma variável com outra constante ou variável.
  Exemplo :

	x = 8;
x *= y;	x *= 2;
equivalente à expressão	x passa a valer 16
x = x * y	

- /= (divisão composta) realiza a divisão de uma variável com outra constante ou variáve).
  Exemplo :

	x = 8;
x /= y;	x /= 2;
equivalente à expressão	x passa a valer 4
x = x / y	

Funções

2.2.14 Entrada e saída digital

- **pinMode ( )** Configura o pino especificado para que se comporte como uma entrada ou como uma saída. Deve-se informar o número do pino que deseja-se configurar e em seguida se o pino será uma entrada (**INPUT**) ou uma saída (**OUTPUT**).
  Sintaxe :

  pinMode(pino, modo);

- **digitalWriter( )** Escreve um valor **HIGH** ou **LOW** em um pino digital. Se o pino foi configurado como uma saída, sua tensão será: **5V** para **HIGH** e **0V** para **LOW**. Se o pino está configurado como uma entrada, **HIGH** levantará o resistor interno de **10KOhms** e **LOW** rebaixará o resistor.
  Sintaxe :

  pinWrite(pino, modo);

- **digitalRead( )** Lê o valor de um pino digital especificado e retorna um valor **HIGH** ou **LOW**.
  Sintaxe :

  int digitalRead(pino);

```
/* Exemplo de função sobre de Entrada e Saída Digital */

int ledPin = 13; // LED conectado ao pino digital 13
int inPin = 7;    // botão conetado ao pino digital 7
int val = 0;       // variável para armazenar o valor lido

void setup()
{
      pinMode(ledPin, OUTPUT); // determina o pino digital 13 como
//uma saída
      pinMode(inPin, INPUT); // determina o pino digital 7 como
//uma entrada
}

void loop()
{
      digitalWrite(ledPin, HIGH); // acende o LED
      val = digitalRead(inPin); // lê o pino de entrada
      digitalWrite(ledPin, val); // acende o LED de acordo com o pino de entrada
}

/*Essa função transfere para o pino 13, o valor lido no pino 7 que é uma entrada*/
```

2.2.15 Entrada e saída analógica

- **analogWrite( ) - PWM** (*Pulse Width Modulation* ou Modulação por Largura de Pulso) é um método para obter sinais analógicos com sinais digitais. Essa função, basicamente, define o valor de um sinal analógico. Ela pode ser usada para acender um LED variando o seu brilho, ou controlar um motor com velocidade variável. Depois de realizar um analogWrite(), o pino gera uma onda quadrada estável com o ciclo de rendimento especificado até que um analogWrite(), um digitalRead() ou um digitalWrite() seja usado no mesmo pino.

  Em kits Galileo esta função está disponível nos pinos 3,5,6,9,10 e 11. As saídas PWM geradas pelos pinos 5 e 6 terão rendimento de ciclo acima do esperado. Isto se deve às interações com as funções millis() e delay(), que compartilham o mesmo temporizador interno usado para gerar as saídas PWM. Para usar esta função deve-se informar o pino ao qual deseja escrever e em seguida informar um valor entre 0 (pino sempre em 0V) e 255 (pino sempre em +5V).

  Sintaxe :

  analogWrite(pino, valor);

- **analogRead( )** Lê o valor de um pino analógico especificado. O kit Galileo contém um conversor analógico-digital de 10 bits com 6 canais. Com isto ele pode digitalizar tensões de entrada entre 0 e 5 Volts, em valores inteiros entre 0 e 1023. Isto permite uma resolução entre leituras de 5 Volts / 1023 ou 0,0048 Volts (4,8 mV) por unidade do valor digitalizado.

  Sintaxe :

```
analogRead(pino);

/* Exemplo de função sobre Entrada e Saída Analógica */

int ledPin = 9; // LED conectado ao pino digital 9
int analogPin = 3; // potenciômetro conectado ao pino analógico 3
int val = 0; // variável para armazenar o valor lido

void setup()
{
    pinMode(ledPin, OUTPUT); // pré-determina o pino como saída
}

void loop()
{
    val = analogRead(analogPin); // lê o pino de entrada
    analogWrite(ledPin, val/16); //
}

/*Torna o brilho de um LED proporcional ao valor lido em um potenciômetro*/
```

2.2.16 Entrada e saída avançada

- **pulseIn( )**Lê um pulso (tanto **HIGH** como **LOW**) em um determinado pino. Por exemplo, se o valor for **HIGH**, a função **pulseIn()** espera que o pino tenha o valor **HIGH**, inicia uma cronometragem, e então espera que o pino vá para **LOW** e para essa cronometragem. Por fim, essa função retorna a duração do pulso em microsegundos. Caso nenhum pulso iniciar dentro de um tempo especificado (a determinação desse tempo na função é opcional), **pulseIn( )** retorna 0. Esta função funciona com pulsos entre 10 microsegundos e 3 minutos

Sintaxe :

```
pulseIn(pino, valor);
ou
pulseIn(pino, valor, tempo);
```

Exemplo :

```
int pin = 7;
unsigned long duration;

void setup( )
{
      pinMode(pin, INPUT); }

void loop( )
{
      duration = pulseIn(pin, HIGH);
}
```

- **shiftOut( )** Envia um byte de cada vez para a saída. Pode começar tanto pelo bit mais significativo (mais à esquerda) quanto pelo menos significativo (mais à direita).

Os bits vão sendo escritos um de cada vez em um pino de dados em sincronia com as alterações de um pino de clock que indica que o próximo bit deve ser escrito. Isto é um modo usado para que os microcontroladores se comuniquem com sensores e com outros microcontroladores. Os dois dispositivos mantêm-se sincronizados a velocidades próximas da máxima, desde que ambos compartilhem a mesma linha de clock.

Nesta função deve ser informado o número referente ao pino no qual sairá cada bit (pino de dados). Em seguida, declara-se o número do pino que será alterado quando um novo bit deverá sair no primeiro pino (pino de clock). Depois, informa-se qual é a ordem de envio dos bits. Essa ordem pode ser **MSBFIRST** (primeiro o mais significativo) ou **LSBFIRST**(primeiro o menos significativo). Por último, decara-se a informação que será enviada para a saída.

Obs: O *pino de dados* e o *pino de clock* devem ser declarados como saída (OUTPUT) pela função *pinMode( )*.

Sintaxe :

shitftOut(pino de dados, pino de clock, ordem, informação);

Exemplo :

```
int latchPin = 8;
int clockPin = 12;
int dataPin = 11;
```

```
void setup() {
    pinMode(latchPin, OUTPUT);
    pinMode(clockPin, OUTPUT);
    pinMode(dataPin, OUTPUT);
}

void loop() {
    for(int j = 0; j < 256; j++) {
      digitalWrite(latchPin, LOW);
      shiftOut(dataPin, clockPin, LSBFIRST, j);
      digitalWrite(latchPin, HIGH);
      delay(1000);
    }
}
```

2.2.17 Tempo

- **millis( )** Retorna o número de milisegundos desde que o kit Galileo começou a executar o programa. Este número voltará a zero depois de aproximadamente 50 dias.

  Sintaxe :

  unsigned long tempo;

  void loop

  {

  .

  :

  tempo = millis();

  }

- **delay( )** Suspende a execução do programa pelo tempo (em milisegundos) especificado (1 segundo = 1000 milisegundos).

  Sintaxe :

  delay(tempo);

- **micros( )** Retorna o número de microsegundos desde que o kit Galileo começou a executar o programa. Este número voltará a zero depois de aproximamente 70 minutos (1 segundo = 1000 milisegundos = 1 000 000 microsegundos).

Sintaxe :

```
unsigned long tempo;
void loop
{
.
:
tempo = micros();
.
:
}
```

Exemplo:

```
/* Este programa mostra uma aplicação das funções millis( ) e delay( )
* Para usar a função micros( ), basta substituir millis( ) por micros ( ) */

unsigned long time;
void setup(){
      Serial.begin(9600);
}
void loop(){
      Serial.print("Time: ");
      time = millis();
```

```
    Serial.println(time); //imprime o tempo desde que o programa começou
    delay(1000);
}
```

- **delayMicroseconds( )** Suspende a execução do programa pelo tempo (em microsegundos) especificado. Atualmente, o maior valor que produzirá uma suspensão precisa é da ordem de 16383. Para suspensões maiores que milhares de microsegundos, deve-se utilizar a função *delay( )*.

  Sintaxe:

  ```
  delayMicroseconds(tempo);
  ```

  Exemplo:

  ```
  int outPin = 8;
  void setup() {
      pinMode(outPin, OUTPUT);
  }
  void loop() {
      digitalWrite(outPin, HIGH);
      delayMicroseconds(50);
      digitalWrite(outPin, LOW);
      delayMicroseconds(50);
  }
  ```

2.2.18 Comunicação serial

- **Serial.begin** ( ) Ajusta o taxa de transferência em bits por segundo para uma transmissão de dados pelo padrão serial. Para comunicação com um computador utiliza-se uma destas taxas: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 57600, 115200. Pode-se, entretanto, especificar outras velocidades, por exemplo, para comunicação através dos pinos 0 e 1 com um componente que requer uma taxa específica.

  Sintaxe :

  Serial.begin(taxa);
  Serial1.begin(taxa);
  Serial2.begin(taxa);
  Serial3.begin(taxa);

  Exemplo para Galileo:

  void setup(){
  /* Abre a porta serial 1, 2, 3 e 4 e ajusta a taxa das portas para
  * 9600 bps, 38400 bps, 19200 bps e 4800 bps respectivamente*/

  Serial1.begin(9600);
  Serial2.begin(38400);
  Serial3.begin(19200);

```
Serial4.begin(4800);
        .
        :
}

void loop() {
        .
        :
}
```

- **int Serial.available( )** Retorna o número de bytes (caracteres) disponíveis para leitura no buffer da porta serial. O buffer serial pode armazenar até 128 bytes.

  Sintaxe:

  ```
  Serial.available();
  ```

  Exemplo :

  ```
  void setup() {
        Serial.begin(9600);
        Serial1.begin(9600);
  }
  void loop() {
  /* lê na porta 0 e envia para a porta 1: */
  ```

```
    if (Serial.available( )) {
      int inByte = Serial.read();
      Serial1.print(inByte, BYTE);
    }
/* lê na porta 1 e envia para a porta 0:*/
    if (Serial1.available()) {
      int inByte = Serial1.read();
      Serial.print(inByte, BYTE);
    }
}
```

- **int Serial.read( )** Retorna o primeiro byte disponível no buffer de entrada da porta serial (ou -1 se não houver dados mo buffer).

Sintaxe :

variavel = Serial.read( ):

Exemplo :

```
int incomingByte = 0;
// para entrada serial
void setup() {
      Serial.begin(9600);
}
void loop() {
// envia dados apenas
//quando recebe dados:
      if (Serial.available() > 0) {
// lê o primeiro byte disponível:
         incomingByte = Serial.read();
// imprime na tela o byte recebido:
         Serial.print("Eu recebi: ");
         Serial.println(incomingByte, DEC);
      }
}
```

- **Serial.flush( )** Esvazia o buffer de entrada da porta serial. De modo geral, esta função apaga todos os dados presentes no buffer de entrada no momento de execução da mesma.

Sintaxe :

Serial.flush();

Exemplo:

```
void setup() {
        Serial.begin(9600);
}
void loop(){
        Serial.flush();
/* Apaga o onteúdo
* do buffer de entrada
*/
        .
        :
}
```

- **Serial.print( )** Envia dados de todos os tipos inteiros, incluindo caracteres, pela porta serial. Ela não funciona com floats, portanto é necessário fazer uma conversão para um tipo inteiro. Em algumas situações é útil multiplicar um float por uma potência de 10 para preservar (ao menos em parte) a informação fracionária. Atente-se para o fato de que os tipos de dados sem sinal, char e byte irão gerar resultados incorretos e atuar como se fossem do tipo de dados com sinal.
  Este comando pode assumir diversas formas:

  Serial.print(valor): Sem nenhum formato especificado: imprime o valor como um número decimal em uma string ASCII.

Exemplo:

```
                                    (envia pela porta serial o código ASCII
int b = 79;                         do 7 e o código ASCII do 9).
Serial.print(b);
```

Serial.print(valor, DEC): Imprime valor como um número decimal em uma string ASCII.
Exemplo:

```
int b = 79;                         (imprime a string ASCII "79").
Serial.print(b, DEC);
```

Serial.print(valor, HEX): Imprime valor como um número hexadecimal em uma string ASCII.
Exemplo:

```
int b = 79;                         (imprime a string "4F").
Serial.print(b, HEX);
```

Serial.print(valor, OCT): Imprime valor como um número o tal em uma string ASCII.
Exemplo:

```
int b = 79;                         (imprime a string "117")
Serial.print(b, OCT);
```

Serial.print(valor, BIN): Imprime valor como um número binário em uma string ASCII
Exemplo:

```
int b = 79;                    (imprime a string "1001111").
Serial.print(b, BIN);
```

Serial.print(valor, BYTE): Imprime valor como um único byte.
Exemplo:

```
int b = 79;                    (envia pela porta serial o valor 79, que será mostrado na tela de um terminal como um caractere "0", pois 79 é o código ASCII do "0").
Serial.print(b, BYTE);
```

Serial.print(str): Se str for uma string ou um array de chars, imprime uma string ASCII.
Exemplo:

```
Serial.print("Intel Galileo");    (imprime a string "Intel Galileo")
```

Exemplo:

```
int analogValue;
void setup()
{
    serial.begin(9600);
}
```

```
void loop()
{
    analogValue = analogRead(0);
    serial.print(analogValue); // imprime um ASCII decimal - o mesmo que "DEC"
    serial.print("\t"); // imprime um tab
    serial.print(analogValue, DEC); // Imprime um valor decimal
    serial.print("\t"); // imprime um tab
    serial.print(analogValue, HEX); // imprime um ASCII hexadecimal
    serial.print("\t"); // imprime um tab
    serial.print(analogValue, OCT); // imprime um ASCII octal
    serial.print("\t"); // imprime um tab
    serial.print(analogValue, BIN); // imprime um ASCII binário
    serial.print("\t"); // imprime um tab
    serial.print(analogValue/4, BYTE);
/* imprime como um byte único e adiciona um "cariage return"
* (divide o valor por 4 pois analogRead() retorna número de 0 à 1023,
* mas um byte pode armazenar valores somente entre 0 e 255
*/
    serial.print("\t"); // imprime um tab
    delay(1000); // espera 1 segundo para a próxima leitura
}
```

- **Serial.println(data)** Esta função envia dados para a porta serial seguidos por um *carriage return* (ASCII 13, ou '\r') e por um caractere de linha nova (ASCII 10, ou '\n'). Este comando utiliza os mesmos formatos do *Serial.print( ):*

  Serial.println(valor): Imprime o valor de um número decimal em uma string ASCII seguido por um carriage return e um linefeed.

  Serial.println(valor, DEC): Imprime o valor de um número decimal em uma string ASCII seguido por um carriage return e um linefeed.

  Serial.println(valor, HEX): Imprime o valor de um número hexadecimal em uma string ASCII seguido por um carriage return e um linefeed.

  Serial.println(valor, OCT): Imprime o valor de um número octal em uma string ASCII seguido por um carriage return e um linefeed.

  Serial.println(valor, BIN): Imprime o valor de um número binário em uma string ASCII seguido por um carriage return e um linefeed.

  Serial.println(valor, BYTE): Imprime o valor de um único byte seguido por um carriage return e um linefeed.

  Serial.println(str): Se str for uma string ou um array de chars, imprime uma string ASCII seguido por um carriage return e um linefeed.

  Serial.println(): Imprime apenas um carriage return e um linefeed.

Exemplo:

```
/* Entrada Analógica lê uma entrada analógica no pino analógico 0 e imprime o valor na
porta serial. */
int analogValue = 0; // variável que armazena o valor analógico

void setup() {
    // abre a porta serial e justa a velocidade para 9600 bps:
    Serial.begin(9600);
}

void loop() {
    analogValue = analogRead(0); // lê o valor analógico no pino 0:
    /* imprime em diversos formatos */
    Serial.println(analogValue); // imprime um ASCII decimal - o mesmo que "DEC"
    Serial.println(analogValue, DEC); // imprime um ASCII decimal
    Serial.println(analogValue, HEX); // imprime um ASCII hexadecimal
    Serial.println(analogValue, OCT); // imprime um ASCII octal
    Serial.println(analogValue, BIN); // imprime um ASCII binário
    Serial.println(analogValue/4, BYTE); // imprime como um byte único

    delay(1000);// espera 1 segundo antes de fazer a próxima leitura:
}
```

Capítulo 3

Exemplos de Aplicaçõe com Intel® Galileo

3.1 Utilizando diodos emissores de luz (*LED*) com Galileo

O código apresentado na Figura 3.1 fará com que um *LED* pisque a cada dois segundos.

No código precisa-se definir qual pino será utilizado, foi escolhido o pino 13, que é um pino digital da placa, desta forma o pino 13 será nossa saída (*output*). Então entra-se no ciclo, na qual o pino ficará *High* , ativado com tensão de 5V, depois de um tempo de espera de 2000 milissegundos (2 segundos) ficará *Low* (0V) e mais 2000 milissegundos de espera.

```
void setup(){
pinMode(13,OUTPUT);
}
void loop(){
digitalWrite(13,HIGH);
delay(2000);
digitalWrite(13,LOW);
delay(2000);
}
```

Figura 3.1: Código do circuito utilizando diodos emissores de luz.

3.1.1 Circuito e prototipação

De acordo com a figura 3.2, nesta aplicação serão necessários: Placa Galileo, um resistor de 220 Ohms (Ω), três *jumpers*, um led e uma *protoboard*.

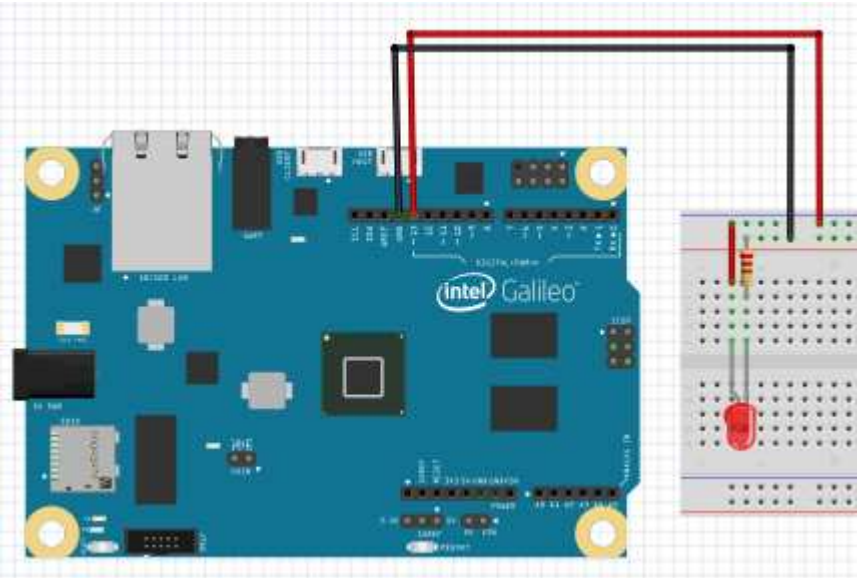


Figura 3.2: Circuito utilizando diodos emissores de luz.

Conecta-se o *ground* na saída do resistor, a entrada do resistor na saída do *led* e a um *jumper* no pino 13 e na entrada do *LED*. O esquema de conexão é apresentado na Figura 3.3.

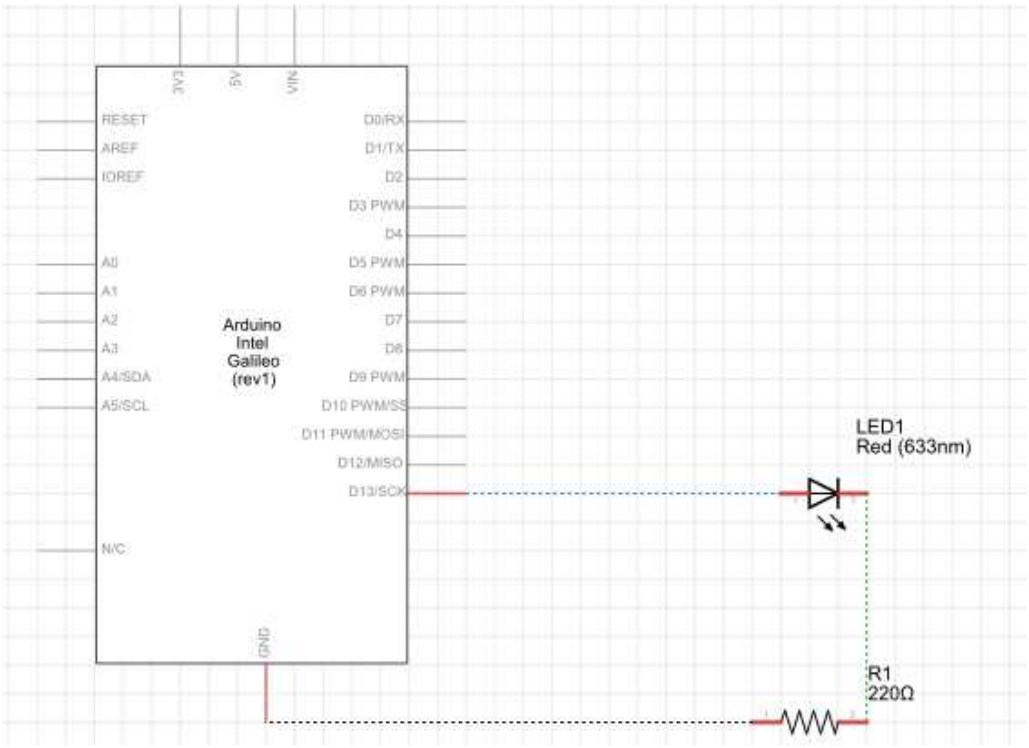


Figura 3.3: Esquemático do circuito utilizando diodos emissores de luz.

3.2 Sensor de luminosidade

Este novo exemplo apresenta a utilização de um sensor de luminosidade (*LDR - Light Dependent Resistance*) com uma placa Galileo, no exemplo será aplicada a porta serial, um diferencial que não foi utilizado no exemplo anterior, na qual foram utilizadas apenas portas digitais. Desta forma segue-se o código na Figura 3.4.

```
int LDR = 0;
int VAL = 0;

void setup(){
  Serial.begin(9600);
}

void loop(){

 VAL = analogRead(LDR);
 Serial.print(VAL);

 if (VAL<512)
 {
  Serial.println(": Escuro");
 }
 else if(VAL > 700)
{
Serial.println(": Claro ");
}

  delay(500);

}
```

Figura 3.4: Código do circuito utilizando sensor de luminosidade.

Para este exemplo define-se duas variáveis, *LDR* e *VAL*, a primeira será nosso padrão, a segunda o valor obtido sensor. Sendo assim defini-se a velocidade da porta serial como 9600. *VAL* fará a leitura do pino analógico, que terá o valor da luminosidade do ambiente, mostrando na tela o valor armazenado. Logo após será verificado se está claro, escuro ou nenhum dos dois, e espera-se 500 milissegundos (meio segundo) para repetir o processo.

3.2.1 Circuito e prototipação

Conforme a figura 3.5, utiliza-se uma placa Galileo, uma *protoboard*, um resistor de 220 Ohms, dois *jumpers*, com ambas pontas *macho*, e um sensor de luminosidade.

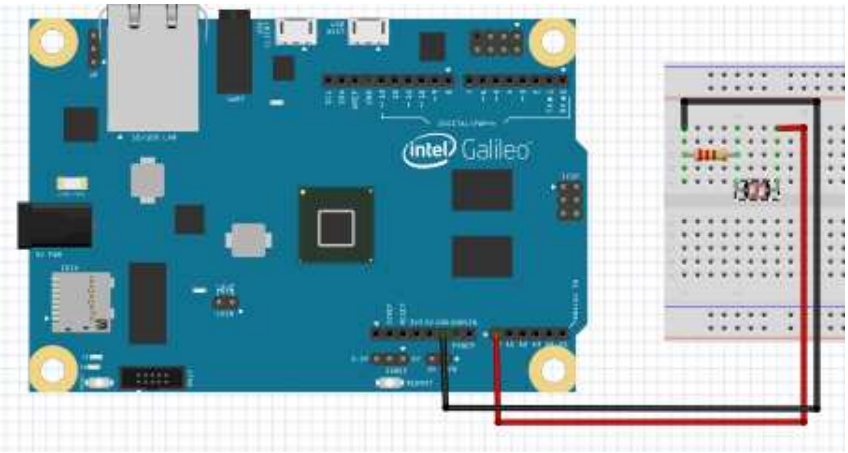


Figura 3.5: Circuito utilizando um sensor de luminosidade.

Conecta-se o primeiro *jumper* no *ground* da placa e em uma das pontas do resistor, depois liga-se o resistor na saída do sensor. Por fim, insere o *jumper* na entrada de alimentação de 5V, como mostra o esquemático da Figura 3.6.

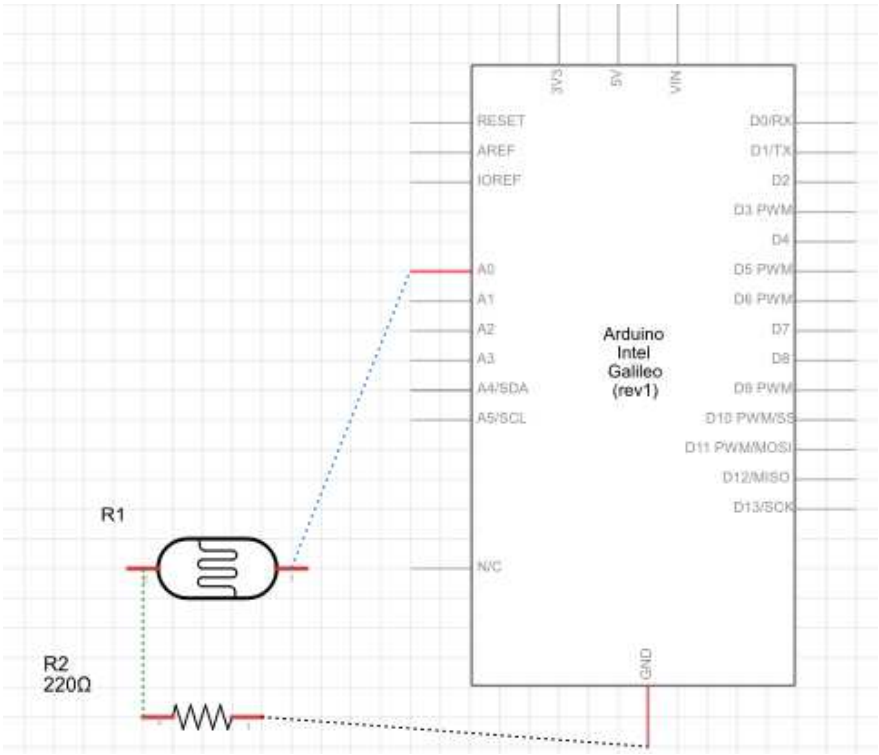


Figura 3.6: Esquemático do circuito utilizando sensor de luminosidade.

3.3 Servomotor

O intuito deste experimento é utilizar um servomotor através de uma placa Galileo. A Figura 3.7 apresenta o código para utilização do servomotor.

Para efetuar este projeto precisa-se incluir a biblioteca *servo.h*. Desta maneira define-se uma variável do tipo *servo* sendo seu nome *servo1*. Escolhido o pino de saída, neste caso o nove, faz-se a movimentação do motor, para 90 graus, depois um tempo de espera de um segundo, para 180 graus, mais um tempo de espera e finalmente 0 graus, e um segundo de espera.

```
#include <servo.h>

Servo servo1;

void setup(){

servo1.attach(9);

}

void loop(){

servo1.write(90):

delay(1000);

servo1.write(180);

delay(1000);

servo1.write(0);

delay(1000);

}
```

Figura 3.7: Código do circuito utilizando servomotor.

3.3.1 Circuito e prototipação

De acordo com a Figura 3.8 foi utilizado uma placa Galileo, uma *protoboard*, servomotor sg90 e três *jumpers*.

A partir desses componentes, conecta-se o circuico conforme indicado na Figura 3.9. Coloca-se o *ground* no servomotor, conectado o servomotor ao pino escolhido e a alimentação.

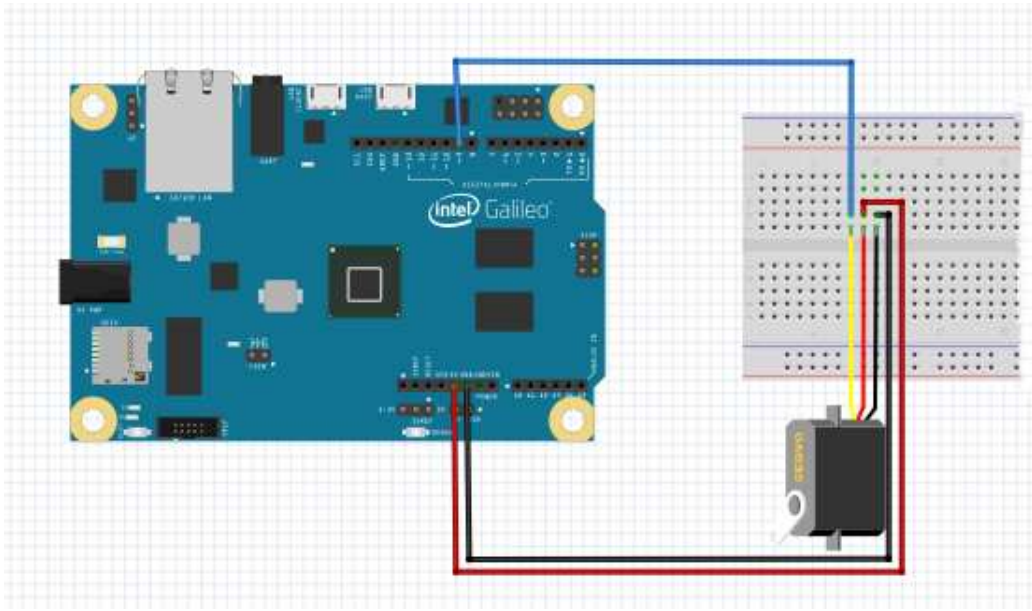


Figura 3.8: Circuito utilizando o servomotor.

3.4 Sensor de temperatura e umidade

O sensor de temperatura e umidade relativa do ar dht11 possui duas finalidades integradas. Para implementar necessita-se incluir no codigo a biblioteca que utilize esse sensor, *dht11.h*, assim como apresentado na Figura 3.10.

Após definido o pino utilizado, neste caso o 8, inicia-se a porta serial. Em seguida, lê-se as informações do pino 8, caso receba 0, não houve erro, -1, houve um erro na soma de verificação (*checksum*), -2 o sensor está ocupado e -3 um erro desconhecido. Depois disso imprime a temperatura e a umidade recebidas pelo sensor.

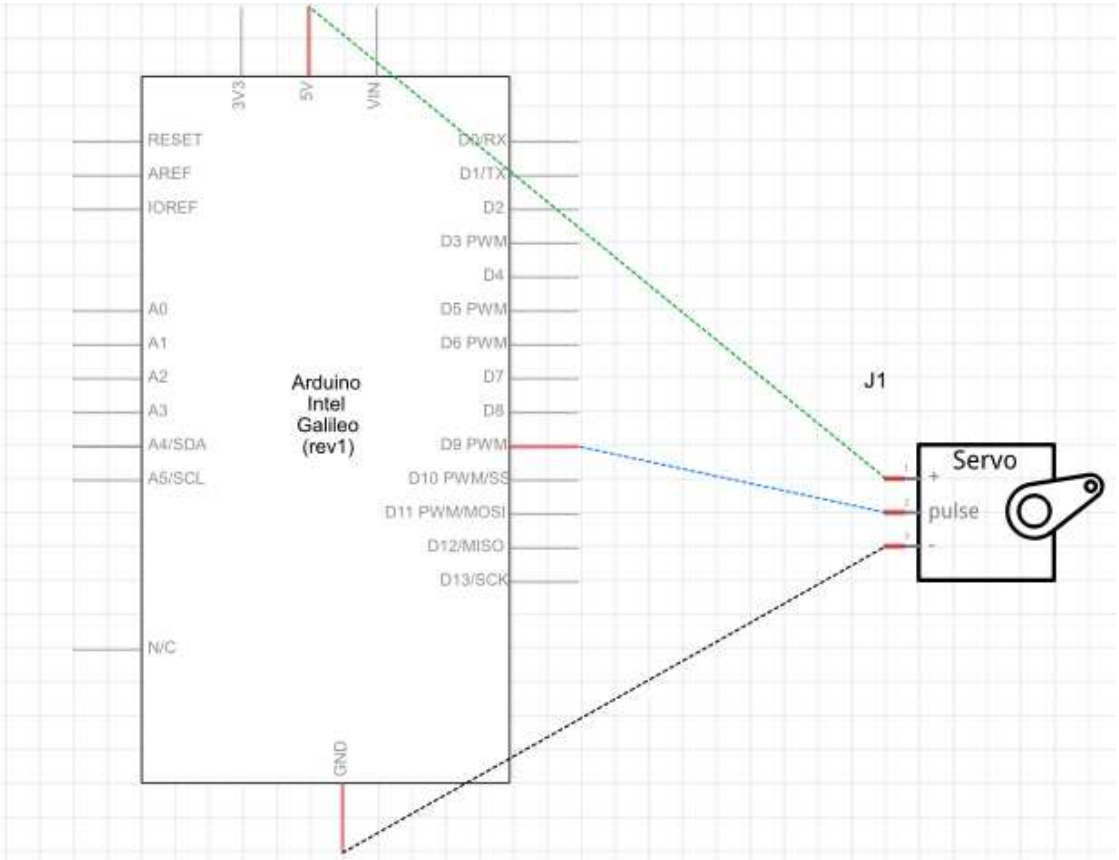


Figura 3.9: Esquemático do circuito utilizando o servomotor.

3.4.1 Circuito e prototipação

Será utilizado uma placa Galileo, uma *protoboard*, 5 *jumpers* com pontas do tipo *macho* e um sensor de temperatura e umidade. A conexão dos componentes é apresentada na Figura 3.11.

Primeiramente é decidido os locais de aterramento e alimentação, ligando ambos os cabos, depois conecta-se o *gnd* do sensor, feito isso liga-se o pino 8 ao sensor, finalizando ao conectar a alimentação no sensor com o ultimo *jumper*, como pode ser visto no esquemático

```
#include <dht11.h>
dht11 DHT11;
#define DHT11PIN 8


void setup()

{
  Serial.begin(9600);
  Serial.println("DHT11 Teste de temperatura e umidade");
  delay(1000);

}

void loop()
{
uint8_t chk = DHT11.read(DHT11PIN);

Serial.print("Status do Sensor: ");
  switch (chk)
  {
    case 0:  Serial.println("OK"); break;
    case -1: Serial.println("Erro de Soma de Verificação"); break;
    case -2: Serial.println("Tempo Excedido"); break;
    case -3: Serial.println("Sensor está ocupado"); break;
    default: Serial.println("Erro desconhecido"); break;
  }

  serial.print("Temp=");
  serial.print(DHT11.temperature, DEC);
  serial.print(" *C");

  serial.print("Umidade=");
  serial.print(DHT11.humidity, DEC);
  serial.print("% ");
  delay(2500);
}
```

Figura 3.10: Código do circuito utilizando sensor de temperatura e umidade DHT11.

da Figura 3.12.

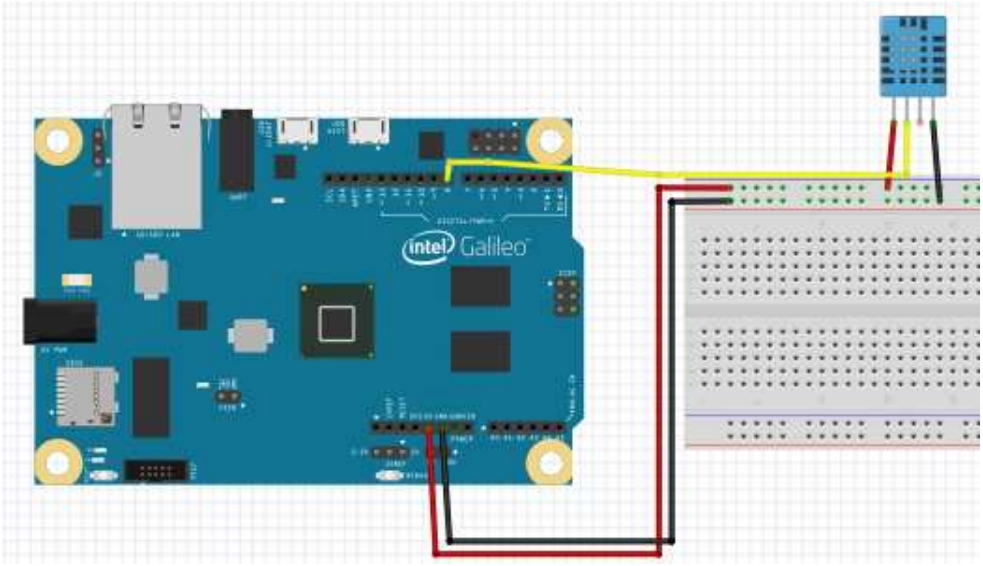


Figura 3.11: Circuito utilizando o sensor de umidade e temperatura.

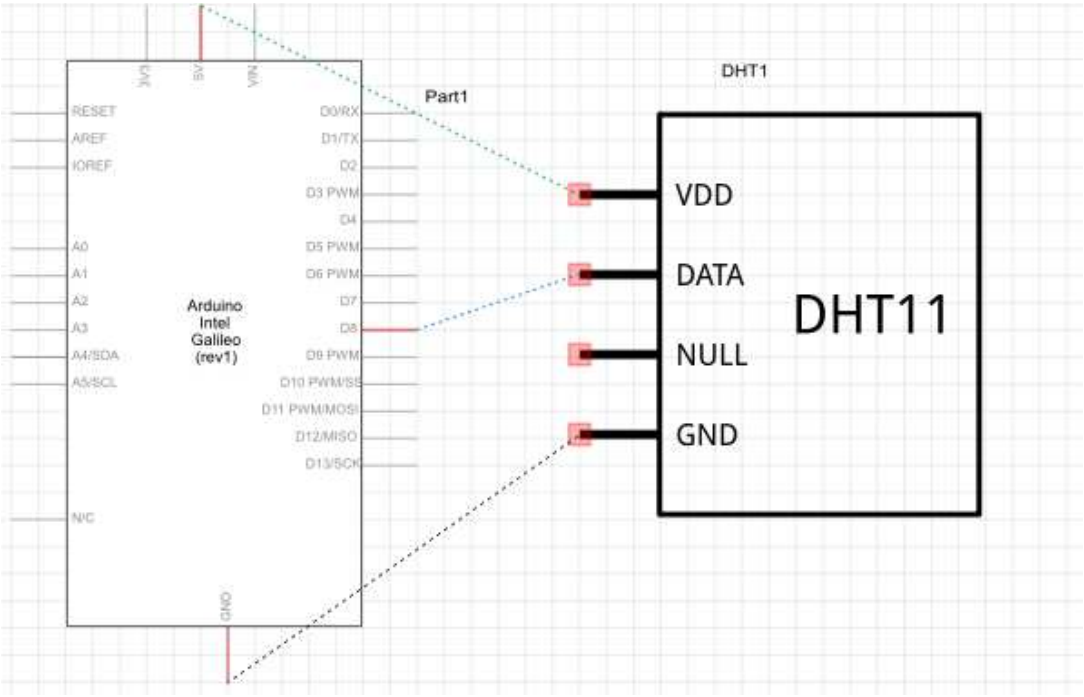


Figura 3.12: Esquemático do circuito utilizando o sensor de umidade e temperatura.

3.5 Sensor ultrassônico

O sensor ultrassônico serve para verificar a distância entre dois pontos basicamente, mas com um principio generalizado como este pode ser utilizado como sensor de presença, distância, entre outras funções. A Figura 3.13 apresenta o código para utilizar o sensor ultrassonico.

```
const int TriggerPin = 8;
const int EchoPin = 9;
long Duration = 0;

void setup(){
  pinMode(TriggerPin,OUTPUT);
  pinMode(EchoPin,INPUT);
  Serial.begin(9600);
}

void loop(){
  digitalWrite(TriggerPin, LOW);
  delayMicroseconds(2);
  digitalWrite(TriggerPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(TriggerPin, LOW);

  Duration = pulseIn(EchoPin,HIGH);

  long Distance_mm = Distance(Duration);

  Serial.print("Distance = ");
  Serial.print(Distance_mm);
  Serial.println(" mm");

  delay(1000);
}

long Distance(long time)
{
    long DistanceCalc;
    DistanceCalc = ((time /2.9) / 2);
    //DistanceCalc = time / 74 / 2;
    return DistanceCalc;
}
```

Figura 3.13: Código do circuito utilizando sensor ultrassônico.

Na Figura 3.13 há uma função a mais, chamada *Distance*, nela calcula-se a distância do objeto até o sensor, multiplica-se o tempo pela velocidade da Onda, descobrindo então o valor desejado. Para utilizar este sensor é necessário definir dois tipos de variáveis: *trigger* é o pino que envia o sinal e o *Echo* é o pino que recebe o retorno do sinal. No final, obtém-se o cálculo da distância através do envio da onda ultrassônica.

3.5.1 Circuito e prototipação

Utiliza-se uma placa Galileo, seis *jumpers* (ambas pontas do tipo *macho*) uma *protoboard* e um sensor ultrassônico *HC-SR01*, conforme apresentado na Figura 3.14.

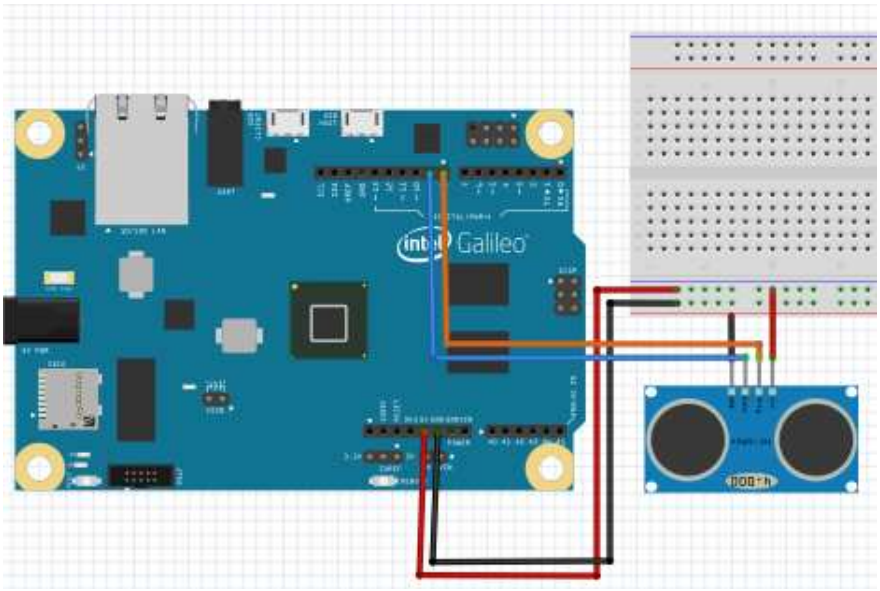


Figura 3.14: Circuito utilizando o sensor ultrassônico.

Primeiramente, defini-se o *ground* na *protoboard* ligando-o ao sensor, em seguida conecta-se os cabos nos pinos nove e oito, a alimentação de 5V na *protoboard* e em seguida no sensor. O esquemático do circuito é apresentado na Figura 3.15.

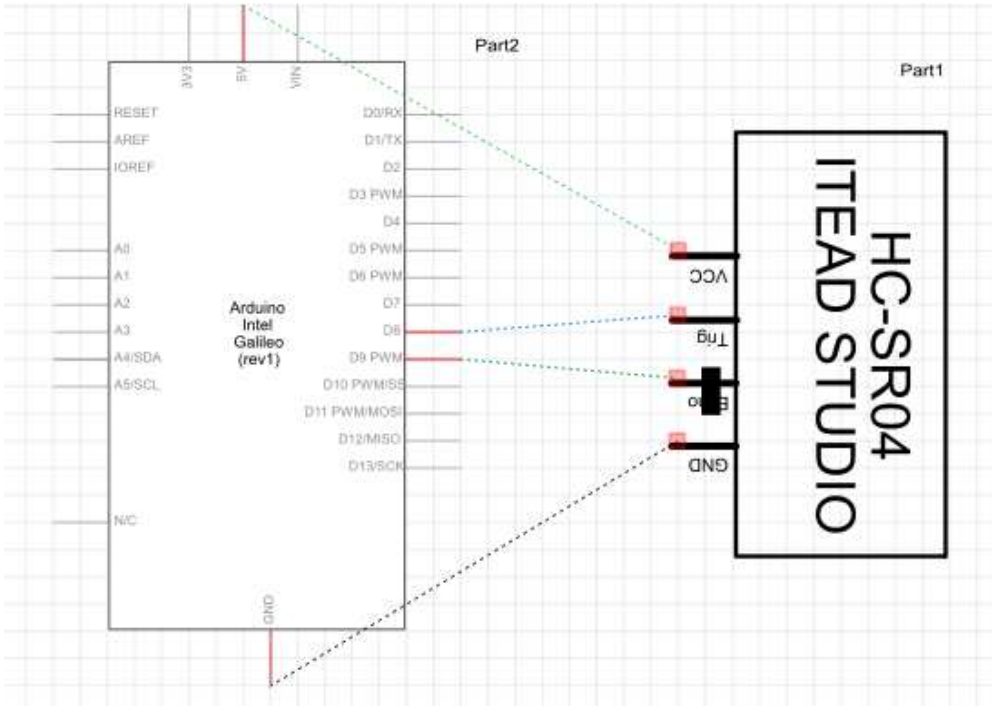


Figura 3.15: Esquemático do circuito utilizando o sensor ultrassônico.

3.6 Utilizando o LCD (*Liquid Crystal display*)

Neste experimento será utilizado um visor de cristal liquido, em inglês *liquid crystal display*, para isso é necessário incluir uma biblioteca para utilizar o visor, sendo ela *LiquidCrystal.h*, desta maneira tem-se o código conforme apresentado na Figura 3.16.

Primeiramente, decide-se o tamanho da área do *display*, então imprime-se *Hello, World!* e finalmente posiciona-se o cursor, na qual decidirá onde o texto irá começar, na posição (0,1), sendo a coluna e linha respectivamente 0 e 1.

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(12,11,5,4,3,2);

void setup(){
lcd.begin(16,2);

lcd.clear();

lcd.print("Hello, World!");


}

void loop(){

lcd.setCursor(0,1);

}
```

Figura 3.16: Código do circuito utilizando o *display* de cristal liquido.

3.6.1 Circuito e prototipação

O *lcd* deve estar conectado ao Galileo conforme apresentado na Figura 3.17. Utilizam-se 16 *jumpers* com ambas pontas macho, uma *protoboard*, um visor, e potenciômetro. Define-se os pinos *ground* e a tensão de 5V sendo colocados respectivamente abaixo e acima da *protoboard*,

Conforme pode ser observado na Figura 3.18, tem-se a ligação dos pinos 12, 11, cinco, quatro, três e dois do Galileo com o lcd. Também deve-se atentar para a conexão do potenciômetro com o *lcd*.

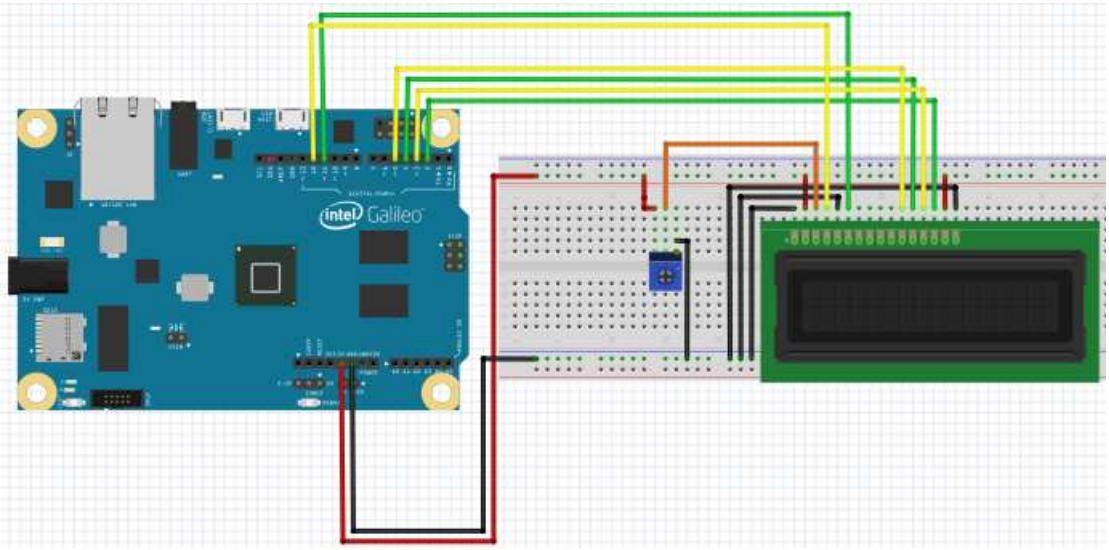


Figura 3.17: Circuito utilizando o *lcd*.

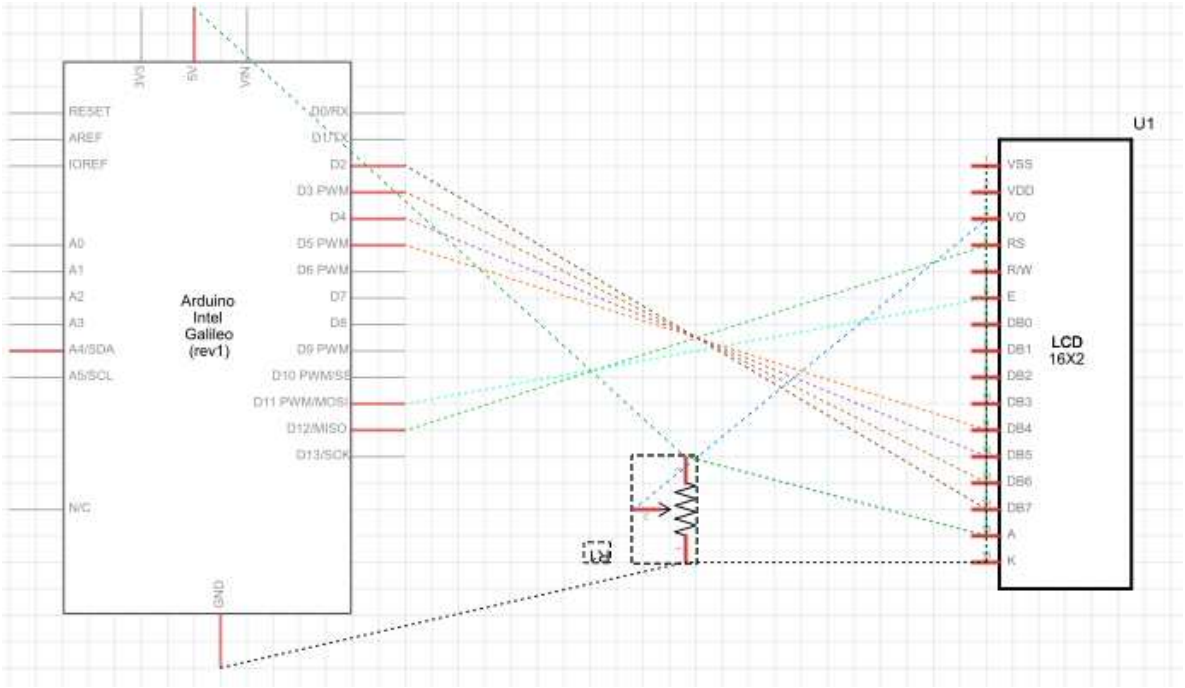


Figura 3.18: Esquemático do circuito utilizando o *lcd*.

Capítulo 4

Suporte a Sistema Operacional no Intel® Galileo

4.1 Projeto Yocto e Ambiente de Desenvolvimento Poky

O Projeto Yocto é um projeto de colaboração *open source* que fornece modelos, ferramentas e métodos para auxiliar na criação de sistemas personalizados baseados em Linux para produtos embarcados, independentemente da arquitetura de hardware[6]. Sendo um projeto *open source*, o Yocto opera com uma estrutura de governança baseada na meritocracia e gerida pelo seu arquiteto chefe, Richard Purdie, da Fundação Linux. Isso permite que o projeto permaneça independente de organizações. Membros e simpatizantes participam de diversas formas, fornecendo recursos para o projeto.

Por que utilizar o Projeto Yocto? É um ambiente de desenvolvimento completo para Linux embarcado com ferramentas, metadados e documentação. As ferramentas são gra-

tuitas e de fácil manipulação, com uma imensa capacidade de desenvolvimento que inclui ambientes de emulação, depuradores, kits de ferramentas para gerar aplicativos, etc.

O Yocto suporta oficialmente as seguintes distribuições GNU/Linux como ambiente de construção:

- Ubuntu 12.04, 13.10 e 14.04;
- Fedora 19 e 20;
- CentOS 6.4 e 6.5;
- Debian 7.0, 7.1, 7.2, 7.3 e 7.4;
- openSUSE 12.2, 12.3 e 13.1.

Existem sub-projetos individuais que compõem o Projeto Yocto. Cada um desses sub-projetos tem um papel fundamental no desenvolvimento de Linux embarcado. Dentre esses há o sistema Poky que é um sistema de referência do Projeto Yocto. O objetivo do sistema Poky é fornecer soluções através de suas ferramentas a desenvolvedores [5].

Poky é a camada de integração independente de plataforma cross-compilação que utiliza *OpenEmbedded Core*. Ele fornece o mecanismo para construir e combinar milhares de projetos distribuídos de código aberto em conjunto para formar uma completa e coerente imagem de softwares Linux inteiramente customizável. Neste capítulo não será abordado como construir uma imagem distribuição linux embarcado com o Yocto. No entanto, será utilizada uma imagem completa de sistema operacional linux embarcado já pronta para instalação sobre o Intel Galileo.

4.2 Instalação Linux sobre o Galileo

Com base no manual de instalação [3], esta seção tem por objetivo instalar e executar no cartão de memória uma versão do Linux Poky.

4.2.1 Materiais necessários

- Computador PC com Windows 7 ou 8
- Cartão de memória MicroSD Card com capacidade mínima de 4GB
- Cabo USB
- Cabo de rede
- Fonte de alimentação 5V – 3A

4.2.2 *Downloads* nescessários de softwares

- Imagem do sistema operacional Linux no cartão SD

  A Intel disponibiliza o arquivo no *link* `https://goo.gl/fMgv6E`. Como mostra a Figura 4.1. O *download* iniciará instantaneamente.

- Putty.exe

  Esse programa está disponível no site da Chiark no *link* `http://goo.gl/5E2gi4`. Como mostra a Figura 4.2 basta clicar no retângulo.

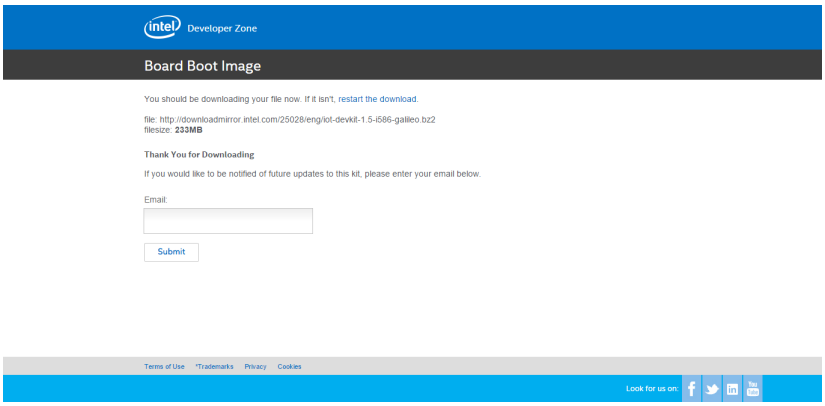


Figura 4.1: Site para *download* da imagem para SO.

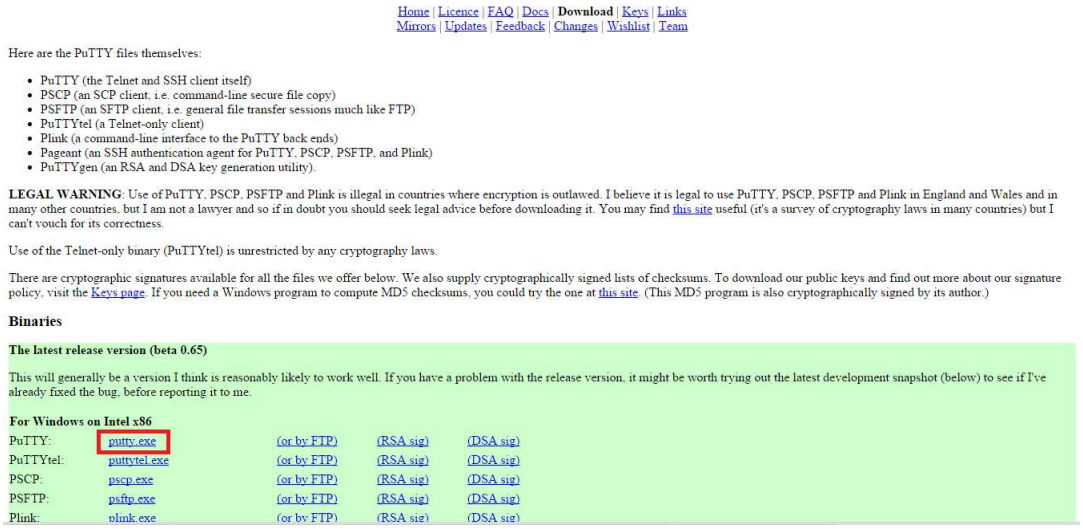


Figura 4.2: Site para *download* do programa Putty.

- win32diskimager

  Esse programa está disponível no site do sourceForge (Figura 4.3 no *link* `http://goo.gl/ZrEznP`.

Figura 4.3: Site para *download* do programa win32diskimager.

4.2.3 Instalação

1. Formatação do cartão de memória

   Vá em **meu computador** clique com o botão direito em cima do nome correspondente ao seu cartão de memória. Em sistema de arquivos selecione a opção FAT32. Dê um novo nome caso queira ao seu cartão de memória. Marque a opção de formatação rápida. E por fim basta clicar no botão **Iniciar**, como indica a Figura 4.4.

2. Descompactação do arquivo iot-devkit-1.5-i586-galileo.bz2

   Descompacte o arquivo iot-devkit-1.5-i586-galileo.bz2 em uma pasta de sua escolha o arquivo possui a imagem Yocto. O arquivo descompactado contém a estrutura apresentada na Figura 4.5.

3. Executar o Win32DiskImager e abrir o arquivo com a imagem do Yocto. Selecionar

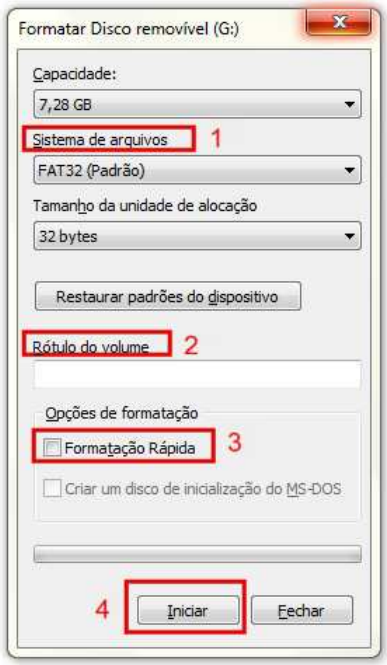


Figura 4.4: Configurações necessárias para a formatação.

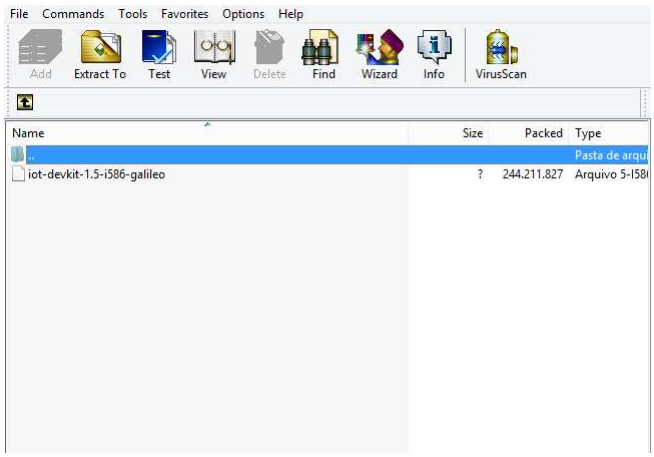


Figura 4.5: Arquivo após a descompactação.

a letra que corresponde ao leitor de cartão e por fim clicar no botão Write para iniciar a gravação no cartão. Como indica a Figura 4.6.

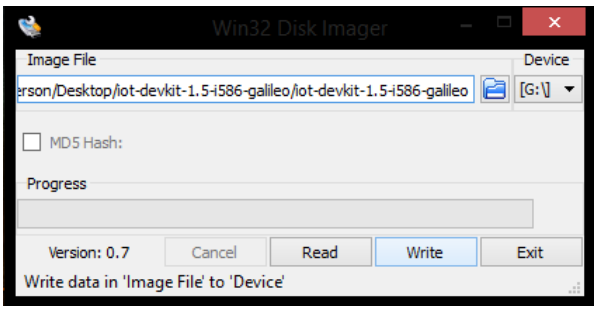


Figura 4.6: Programa Win32DiskImager.

4. Ao final da escrita verifique o cartão que deve ficar com a estrutura de arquivos como a exibida na Figura 4.7.

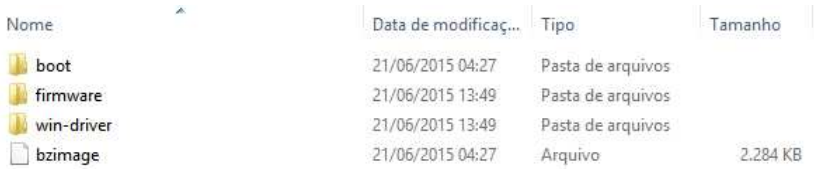


Figura 4.7: Estrutura após a escrita da imagem Linux.

Com a placa Galileo desligada, inserir o cartão de memória na placa e em seguida conecte a fonte de alimentação na placa. O sistema operacional estará contido no cartão de memória e será carregado instantaneamente. Para certificar-se que o Linux está sendo carregado no cartão de memória, fazer o *upload* pela IDE do Galileo do exemplo Blink que liga e desliga o LED da placa gerando o efeito de piscar como mostra a Figura 4.8. Uma vez que instalado o linux mesmo desligando a Galileo ao reiniciar o sketch é carregado e executado automaticamente.

Figura 4.8: LED indicando o funcionamento do sistema operacional.

4.2.4 Manipulação do Linux via SSH

A comunicação com o Linux pode ser feita por alguns meios. Nessa seção será abordada a comunicação telnet utilizando conexão via ethernet.

1. Conectar uma ponta do cabo de rede na placa Intel® Galileo na porta ethernet e a outra a um computador (ou ponto de rede ou equipamento de interconexão). O cabo de alimentação e comunicação USB devem estar conectados ao Galileo.

2. Para a comunicação com a placa Galileo, será nescessário atribuir um IP à placa. Carregue a IDE Galielo e salve o sketch com o nome de sua escolha e compile-o. Através do código a seguir, compilado na placa, ativará o servidor Telnet e atribuirá

um IP ao Galileo. O IP atribuído é o que está no retângulo conforme indica a Figura 4.9.

```
/*********************************************
*        Acesso via Telnet para placa Intel Galileo
*********************************************/

void setup() {
}
void loop() {
        system("ifconfig > /dev/ttyGS0");
        sleep(10);
}
```

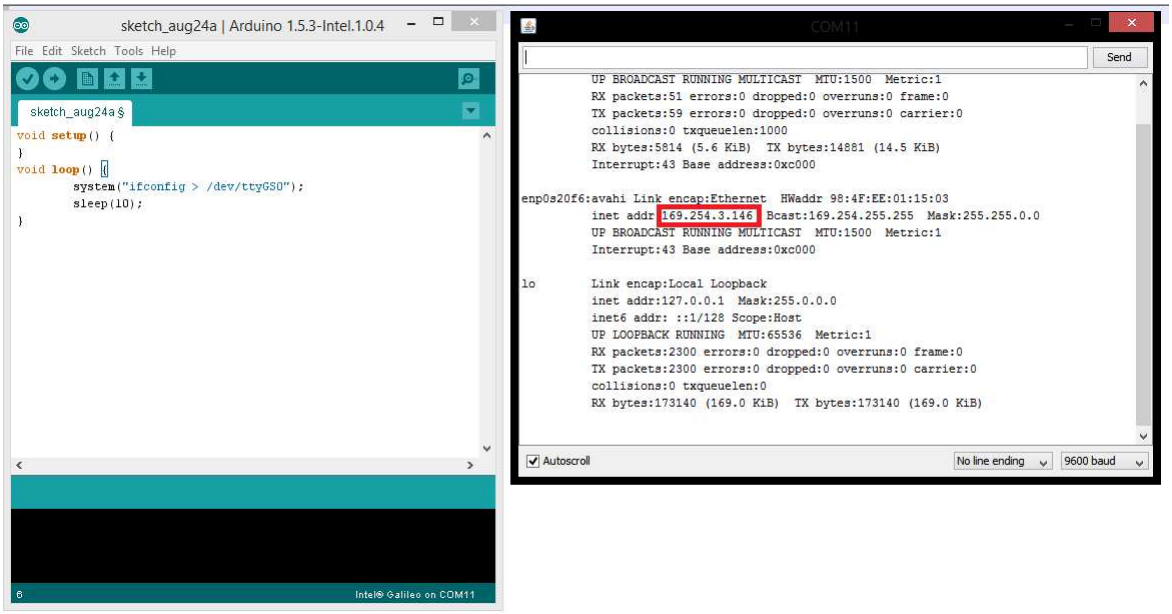


Figura 4.9: Atribuição de número IP.

3. Executar o cliente Telnet, a partir do computador para comunicação com a placa Galileo. O aplicativo putty será utilizado como cliente. Aplicar as seguintes configurações conforme indica a Figura 4.10.

   (a) No campo *Host Name (or Ip address)* inserir o endereço IP que foi atribuído à placa Galileo.

   (b) No campo *Connection type* selecionar SSH.

   (c) Clicar no botão *open* para iniciar a conexão.

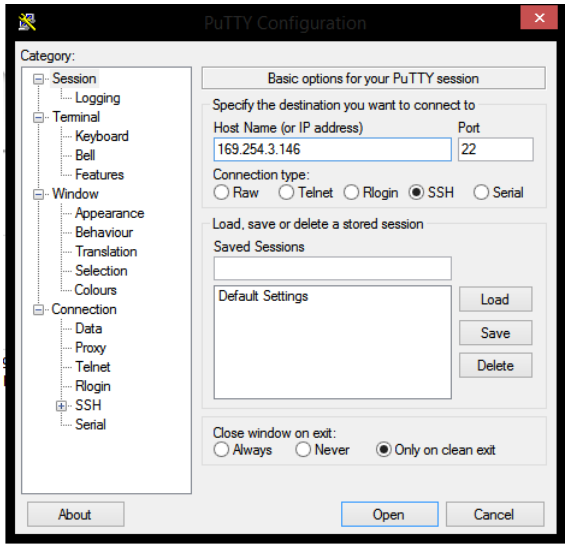


Figura 4.10: Configurações nescessárias para o Putty.

4. Com a ausência de erros, será aberta uma janela com o console do Linux, instalado na placa Galileo, conforme indica a Figura 4.11.

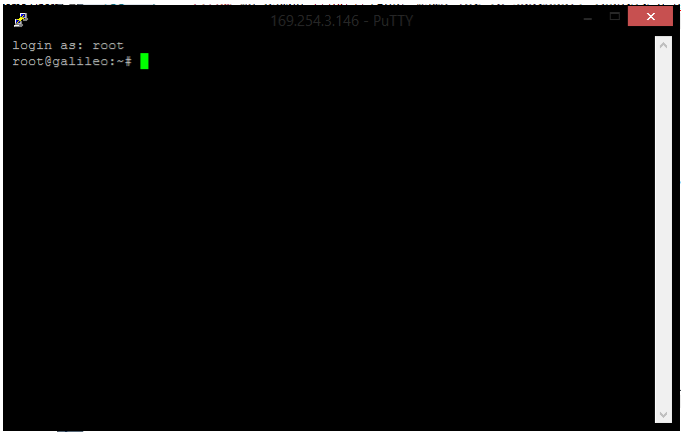


Figura 4.11: Console do Linux Poky.

4.3 Compilação de um program em C

4.3.1 Objetivo

Deseja-se compilar arquivos em C dentro do sistema operacional já instalado no Galileo.

4.3.2 Procedimentos

1. Salvar o arquivo .c em um diretório do cartão SD.
2. Conectar-se ao Galileo via SSH.
3. Entrar no diretorio onde o arquivo .c foi salvo.
4. Digitar o comando *ls*, que indicará se o arquivo .c está dentro do SO. Para esta demonstração utilizou-se o exemplo *LEDblink.c*.
5. Digitar o comando *gcc LEDblink.c -o led* para compilar o programa.

6. Digitar o comando *./led* para executar o programa.

7. O resultado dessa execução (baseando-se no exemplo LEDblink.c) é apresentado na Figura 4.12.

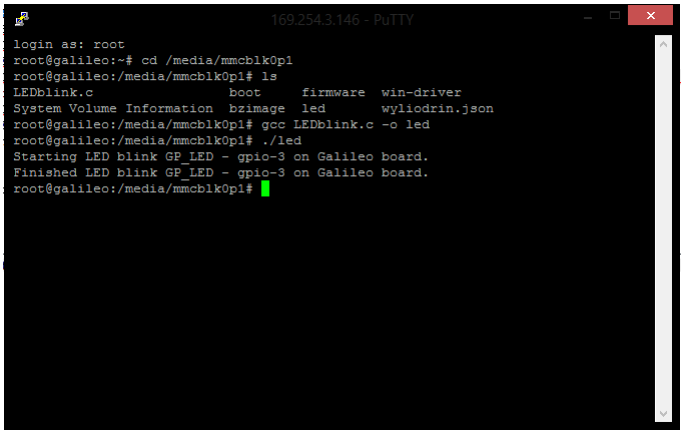


Figura 4.12: Resultado da execução do programa led no Galileo.

Capítulo 5

Exemplos de Aplicação Linux sobre Intel® Galileo

5.1 Manipulação de GPIOs a partir do Linux

O GPIO (*General Purpose Input/Output*), é basicamente um conjunto de pinos responsável por fazer a comunicação de entrada e saída de sinais analógicos e digitais no Intel Galileo. Com a junção de todos os GPIOs é possível manipular um conjunto de 27 pinos. Sendo 14 pinos digitais, 6 pinos analógicos e mais 7 pinos que são pinos dedicados (LED, pushbuton(reset), etc). Com estes pinos é possível acionar LEDs, motores, relês, fazer leitura de sensores e botões, entre outros.

5.1.1 Objetivo

O objetivo desse exemplo é a manipulação de pinos GPIOs da placa Galileo utilizando o Linux Poky [1]. Para este exemplo, a placa Intel® Galileo deve ser inicializada usando a imagem do Linux, conforme apresentado no Capítulo 4.

5.1.2 Construção do circuito

Para a construçãoo do circuito os pinos 5, 6, e 7 serão utilizados, uma vez que são livres e não têm quaisquer outras entradas (ao contrário do pino 3, por exemplo). Em cada um desses pinos será incluido um LED e para o acinamento seguro de cada LED, serão incluidas resistências de 220 ohms, conforme ilustra a Figura 5.1.

5.1.3 Programa em C

Seguem os passos necessários para manipulação dos pinos GPIOs via sistema operacional:

1. Exportar o número GPIO para o arquivo */sys/class/gpio/export*

2. Definir a direção: para a entrada, ou para a saída */sys/class/gpio/numero-gpio/direction*

Esses passos são implementados usando a função openGPIO, que abre o arquivo correspondente e retorna este identificador de arquivo para leitura ou gravação do pino GPIO passado como parâmetro, dependendo da direção declarada.

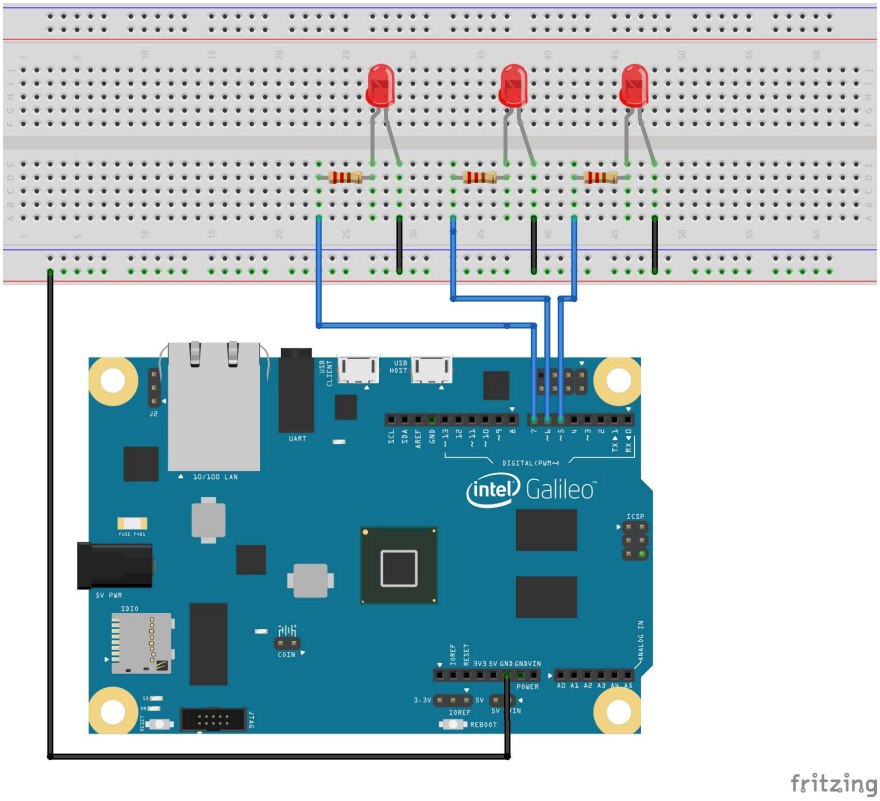


Figura 5.1: Esquemático de pinos.

```
1  int openGPIO(int gpio, int direction ){
2          char buffer[256];
3          int fileHandle;
4          int fileMode;
5
6          //Exporta GPIO
7          fileHandle = open("/sys/class/gpio/export", O_WRONLY);
8
9          if(ERROR == fileHandle){
```

```
10              puts("Erro ao tentar abrir /sys/class/gpio/export")
                    ;
11              return(-1);
12      }
13
14      sprintf(buffer, "%d", gpio);
15      write(fileHandle, buffer, strlen(buffer));
16      close(fileHandle);
17
18      // Abre GPIO para leitura/gravacao da direcao
19      sprintf(buffer, "/sys/class/gpio/gpio%d/direction", gpio);
20      fileHandle = open(buffer, O_WRONLY);
21
22      if(ERROR == fileHandle){
23              puts("Impossivel abrir o arquivo:");
24              puts(buffer);
25              return(-1);
26      }
27
28      if (direction == GPIO_DIRECTION_OUT){
29              // define direcao de saida
30              write(fileHandle, "out", 3);
31              fileMode = O_WRONLY;
32      }
33      else
```

```
34          {
35                  // define direcao de entrada
36                  write(fileHandle, "in", 2);
37                  fileMode = O_RDONLY;
38          }
39          close(fileHandle);
40
41          //Abre GPIO para leitura/gravacao
42          sprintf(buffer, "/sys/class/gpio/gpio%d/value", gpio);
43          fileHandle = open(buffer, fileMode);
44
45          if(ERROR == fileHandle){
46                  puts("Impossivel abrir o arquivo:");
47                  puts(buffer);
48                  return(-1);
49          }
50
51          return(fileHandle); //Este identificador de arquivo sera
               utilizado na leitura/gravacao e na finalizacao das
               operacoes.
52  }
```

Na função principal, invoca-se a função *openGPIO* para cada pino que deseja-se manipular:

```
fileHandleGPIO_5 = openGPIO (GP_5, GPIO_DIRECTION_OUT);
fileHandleGPIO_6 = openGPIO (GP_6, GPIO_DIRECTION_OUT);
fileHandleGPIO_7 = openGPIO (GP_7, GPIO_DIRECTION_OUT);
```

Uma vez que seja verificado que não há erros, muda-se o valor dos pinos GPIOs, sendo 0 para desligado ou 1 para ligado, passando este parametro para a função *writeGPIO* que, de acordo com o valor repassado para pino, possibilitará que os LEDs pisquem:

```
for(i=0; i< 10; i++){
    //Ligando LED
    writeGPIO(fileHandleGPIO_LED, 1);
    writeGPIO(fileHandleGPIO_5, 1);
    writeGPIO(fileHandleGPIO_6, 1);
    writeGPIO(fileHandleGPIO_7, 1);
    sleep(BLINK_TIME_SEC);

    // Desligando LED
    writeGPIO(fileHandleGPIO_LED, 0);
    writeGPIO(fileHandleGPIO_5, 0);
    writeGPIO(fileHandleGPIO_6, 0);
    writeGPIO(fileHandleGPIO_7, 0);
    sleep(BLINK_TIME_SEC);
}
```

O código completo do exemplo está a seguir.

```
1  /*
2  * Copyright (c) 2014 Intel Corporation  All Rights Reserved.
3  * Author: Sulamita Garcia
4  * Based on LEDblink.c by Nandkishor Sonar.
5  */
6
7  /*
8  * Este codigo pisca LEDs ligados ao Galileo GPIO3 (Arduino PIN13),
       GPIO17,
9  * GPIO24 and GPIO27 (pins 5 6 and 7)
10  */
11
12 #include <stdlib.h>
13 #include <stdio.h>
14 #include <fcntl.h>
15 #include <unistd.h>
16 #include <string.h>
17
18 #define GP_LED                (3) // GPIO3 e LED GP - LED conectado
       entre Cypress CY8C9540A e RTC cabecalho da bateria
19 #define GP_5                          (17) //GPIO17 correspondete ao
       PIN5 do Arduino
20 #define GP_6                               (24) //GPIO24
       correspondete ao PIN6 do Arduino
```

```
21 #define GP_7                                (27) //GPIO27
   correspondete ao PIN7 do Arduino
22 #define BLINK_TIME_SEC        (1) // 1 segundos de tempo para ligar
   e desligar o led
23 #define GPIO_DIRECTION_IN     (1)
24 #define GPIO_DIRECTION_OUT    (0)
25 #define ERROR                 (-1)
26
27
28 int openGPIO(int gpio, int direction ){
29         char buffer[256];
30     int fileHandle;
31     int fileMode;
32
33         //Export GPIO
34     fileHandle = open("/sys/class/gpio/export", O_WRONLY);
35     if(ERROR == fileHandle){
36             puts("Error: Unable to opening /sys/class/gpio/export")
                ;
37             return(-1);
38     }
39
40         sprintf(buffer, "%d", gpio);
41     write(fileHandle, buffer, strlen(buffer));
42     close(fileHandle);
```

```
43
44          //Direction GPIO
45      sprintf(buffer, "/sys/class/gpio/gpio%d/direction", gpio);
46      fileHandle = open(buffer, O_WRONLY);
47
48      if(ERROR == fileHandle){
49          puts("Unable to open file:");
50          puts(buffer);
51          return(-1);
52      }
53
54      if (direction == GPIO_DIRECTION_OUT){
55          // Set out direction
56          write(fileHandle, "out", 3);
57          fileMode = O_WRONLY;
58      }
59      else{
60          // Set in direction
61          write(fileHandle, "in", 2);
62          fileMode = O_RDONLY;
63      }
64      close(fileHandle);
65
66
67      //Open GPIO for Read / Write
```

```
68      sprintf(buffer, "/sys/class/gpio/gpio%d/value", gpio);
69      fileHandle = open(buffer, fileMode);
70      if(ERROR == fileHandle){
71          puts("Unable to open file:");
72          puts(buffer);
73          return(-1);
74      }
75
76      return(fileHandle);  //This file handle will be used in read/
            write and close operations.
77  }
78
79  int writeGPIO(int fHandle, int val){
80
81          if(val ==  0){
82                  // Set GPIO low status
83          write(fHandle, "0", 1);
84      }
85      else{
86          // Set GPIO high status
87          write(fHandle, "1", 1);
88      }
89
90      return(0);
91  }
```

```
92
93  int readGPIO(int fileHandle){
94
95          int value;
96
97      read(fileHandle, &value, 1);
98
99      if('0' == value){
100         // Current GPIO status low
101         value = 0;
102     }
103     else{
104         // Current GPIO status high
105         value = 1;
106     }
107
108     return value;
109 }
110
111 int closeGPIO(int gpio, int fileHandle){
112
113         char buffer[256];
114
115     close(fileHandle); //This is the file handle of opened GPIO for
        Read / Write earlier.
```

```
116
117     fileHandle = open("/sys/class/gpio/unexport", O_WRONLY);
118     if(ERROR == fileHandle){
119         puts("Unable to open file:");
120         puts(buffer);
121         return(-1);
122     }
123
124         sprintf(buffer, "%d", gpio);
125     write(fileHandle, buffer, strlen(buffer));
126     close(fileHandle);
127     return(0);
128 }
129
130 int main(void){
131     int fileHandleGPIO_LED;
132     int fileHandleGPIO_5;
133     int fileHandleGPIO_6;
134     int fileHandleGPIO_7;
135     int i=0;
136
137     puts("Starting LED blink GP_LED - gpio-3 on Galileo board.");
138
139     fileHandleGPIO_LED = openGPIO(GP_LED, GPIO_DIRECTION_OUT);
140     fileHandleGPIO_5 = openGPIO(GP_5, GPIO_DIRECTION_OUT);
```

```
141     fileHandleGPIO_6 = openGPIO(GP_6, GPIO_DIRECTION_OUT);
142     fileHandleGPIO_7 = openGPIO(GP_7, GPIO_DIRECTION_OUT);
143
144
145     if(ERROR ==  fileHandleGPIO_LED){
146         return(-1);
147     }
148
149     for(i=0; i< 10; i++){
150
151                 //LED ON
152         writeGPIO(fileHandleGPIO_LED, 1);
153         writeGPIO(fileHandleGPIO_5, 1);
154         writeGPIO(fileHandleGPIO_6, 1);
155         writeGPIO(fileHandleGPIO_7, 1);
156         sleep(BLINK_TIME_SEC);
157
158         //LED OFF
159         writeGPIO(fileHandleGPIO_LED, 0);
160         writeGPIO(fileHandleGPIO_5, 0);
161         writeGPIO(fileHandleGPIO_6, 0);
162         writeGPIO(fileHandleGPIO_7, 0);
163         sleep(BLINK_TIME_SEC);
164     }
165
```

```
166     closeGPIO(GP_LED, fileHandleGPIO_LED);
167     closeGPIO(GP_5, fileHandleGPIO_5);
168     closeGPIO(GP_6, fileHandleGPIO_6);
169     closeGPIO(GP_7, fileHandleGPIO_7);
170
171     puts("Finished LED blink GP_LED - gpio-3 on Galileo board.");
172
173     return 0;
174 }
```

5.2 Alarme contra roubo

5.2.1 Objetivo

Esse exemplo tem como objetivo identificar um invasor através de um sensor de proximidade com a placa Galileo [2]. No exemplo, sempre que um objeto se encontra à uma distância de 3 centímetros a partir do sensor de proximidade, o sistema emite um alarme e um LED pisca continuamente até que o objeto saia fora do alcance do sensor de proximidade.

5.2.2 Componentes eletrônicos necessários

- Placa de desenvolvimento Intel Galileo
- Jumper entre 3,3 V e 5 V da placa Intel Galileo ou 5 V de alimentação externa.

- Sensor de proximidade com saída digital
- Buzzer
- transistor 2N2222
- Resistor 1k
- Protoboard

5.2.3 Distribuição de pinos

O circuito é montado de acordo com a pinagem estabelecida na Tabela 5.2.3. Depois disso, implementa-se o programa em C que controlará os GPIOs para alternar o LED e os alarmes com base na entrada de proximidade.

Conecte a fonte de alimentação na placa Intel Galileo e utilize *jumpers* para alimentar o circuito na protoboard. Siga o esquemático do circuito, conforme a Figura 5.2:

Pino	**Direção**	**Funcionalidade**
GPIO 8	in	Sensor de proximidade
GPIO 13	out	LED
PWM 3	out	Controlador de buzzer
GND 1	n/a	conectar no sensor de proximidade
GND 2	n/a	conectar na placa de ensaios
Power (3.3 V)	out	Sensor de proximidade

Tabela 5.1: Mapeamento dos pinos na placa e sua funcionalidade

1. GPIO 13 e PWM 3 Será usado como saídas e 8 como entrada.
2. Usa-se 3.3 V para alimentar o sensor de proximidade.
3. GPIO 13 será usado para piscar a luz do LED.

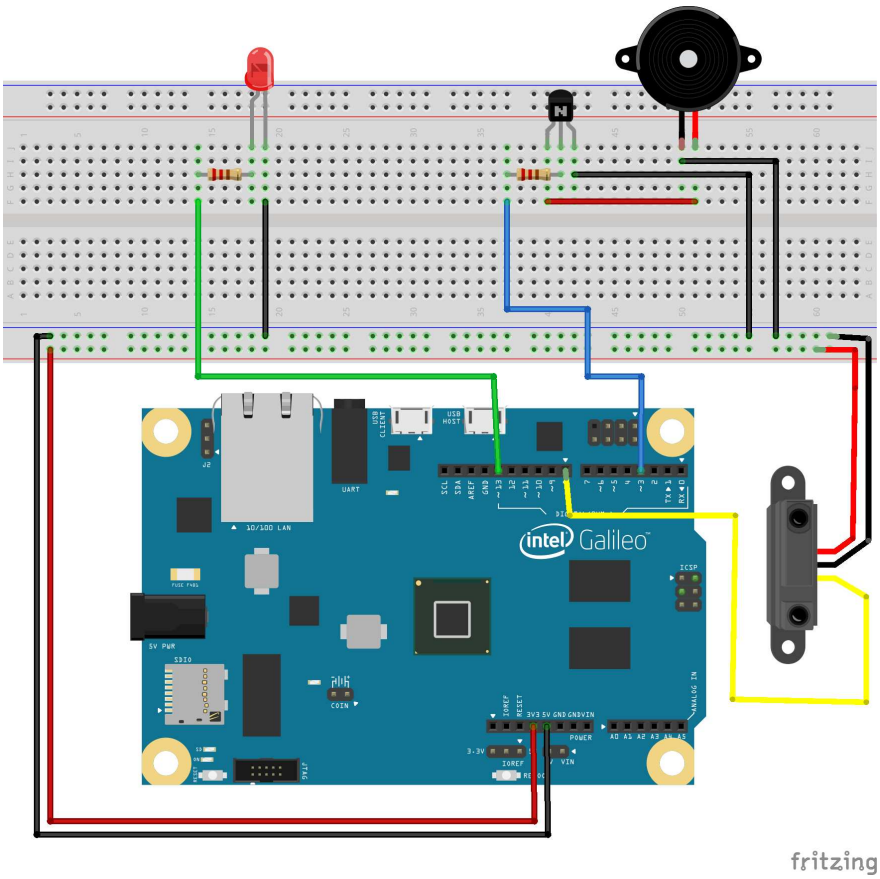


Figura 5.2: Esquemático do circuito de alarme.

4. PWM 3 será usado para modular a quantidade de corrente que passa para a base do transistor. Dependendo desta corrente, o transistor irá variar a quantidade de corrente que vai para o buzzer. Usa-se um resistor de 1K ohm entre PWM 3 e a base de transistor.

5. GPIO 8 será a entrada digital do sensor de proximidade. Quando qualquer objeto estiver dentro do alcance do sensor de proximidade, o sensor retornará alta tensão, caso contrário, será baixa.

6. Dependendo da entrada em GPIO 8, o programa irá definir a tensão GPIO 13 a ALTA para acender o LED. Também vai modular o ciclo de trabalho de PWM 3. Com base nas variações do ciclo de trabalho, o buzzer emitirá um som.

A representação lógica de cada GPIO está disponível no diretório */sys/class/GPIO*, e pinos PWM estão no diretório */sys/class/PWM* do Linux Poky.

5.2.4 Programa em C

O código seguirá as etapas necessárias para manipular os GPIOs, que são:

1. Exportar o número GPIO para o arquivo */sys/class/gpio/export*
2. Definir a direção: para a entrada, ou para a saída */sys/class/gpio/numero-gpio/direction*

Esses passos são implementados usando a função openGPIO, que abre o arquivo correspondente e retorna este identificador de arquivo para leitura ou gravação do pino GPIO passado como parâmetro, dependendo da direção declarada.

```
1 int openGPIO(int gpio, int direction ){
2         char buffer[256];
3         int fileHandle;
4         int fileMode;
5
6         //Exporta GPIO
7         fileHandle = open("/sys/class/gpio/export", O_WRONLY);
8
```

```
9         if(ERROR == fileHandle){
10                puts("Erro ao tentar abrir /sys/class/gpio/export")
                     ;
11                return(-1);
12        }
13
14        sprintf(buffer, "%d", gpio);
15        write(fileHandle, buffer, strlen(buffer));
16        close(fileHandle);
17
18        // Abre GPIO para leitura/gravacao da direcao
19        sprintf(buffer, "/sys/class/gpio/gpio%d/direction", gpio);
20        fileHandle = open(buffer, O_WRONLY);
21
22        if(ERROR == fileHandle){
23                puts("Impossivel abrir o arquivo:");
24                puts(buffer);
25                return(-1);
26        }
27
28        if (direction == GPIO_DIRECTION_OUT){
29                // define direcao de saida
30                write(fileHandle, "out", 3);
31                fileMode = O_WRONLY;
32        }
```

```
33          else
34          {
35                  // define direcao de entrada
36                  write(fileHandle, "in", 2);
37                  fileMode = O_RDONLY;
38          }
39          close(fileHandle);
40
41          //Abre GPIO para leitura/gravacao
42          sprintf(buffer, "/sys/class/gpio/gpio%d/value", gpio);
43          fileHandle = open(buffer, fileMode);
44
45          if(ERROR == fileHandle){
46                  puts("Impossivel abrir o arquivo:");
47                  puts(buffer);
48                  return(-1);
49          }
50
51          return(fileHandle); //Este identificador de arquivo sera
               utilizado na leitura/gravacao e na finalizacao das
               operacoes.
52  }
```

Essa função cuida das etapas necessárias para manipular os pinos PWM, que são:

1. Exportar o número do pino PWM no arquivo */sys/class/pwm/pwmchip0/export*

```c
int openPWM(int port){
        char buffer[256];
        int fileHandle;
        int fileMode;

        //Exporta GPIO
        fileHandle = open("/sys/class/pwm/pwmchip0/export",
           O_WRONLY);

        if(ERROR == fileHandle){
                puts("Error: Impossivel abrir: /sys/class/pwm/
                   pwmchip0/export");
                return(-1);
        }
        sprintf(buffer, "%d", port);
        write(fileHandle, buffer, strlen(buffer));
        close(fileHandle);
        sleep(1);
        return 0;
}
```

2. Ativar o pino PWM

```c
int enablePWM(int enable,int port){
```

```
2         char buffer[256];
3         int fileHandle;
4
5         // Ativar PWM
6         sprintf(buffer, "/sys/class/pwm/pwmchip0/pwm%d/enable"
              , port);
7         fileHandle = open(buffer, O_WRONLY);
8         if(ERROR == fileHandle){
9                 puts("Impossivel abrir o arquivo:");
10                puts(buffer);
11                return(-1);
12        }
13
14        sprintf(buffer, "%d", enable);
15        write(fileHandle, buffer, strlen(buffer));
16        return 0;
17}
```

3. Definir o período do PWM

```
1 int setPWMPeriod(int period, int port)
2 {
3         // Abrir GPIO para leitura / escrita
4         char buffer[256];
5         int fileHandle;
```

```
6
7
8           sprintf(buffer, "/sys/class/pwm/pwmchip0/pwm%d/period"
                , port);
9
10          fileHandle = open(buffer, O_WRONLY);
11          if(ERROR == fileHandle){
12                  puts("Impossivel abrir o arquivo:");
13                  puts(buffer);
14                  return(-1);
15          }
16
17          sprintf(buffer, "%d", period);
18          write(fileHandle, buffer, strlen(buffer));
19
20          close(fileHandle);
21          return(0);
22
23  }
```

4. Definir o ciclo de trabalho

```
1  int setPWMDutyCycle(int dutycycle, int port)
2  {
3          //abre GPIO para leitura / escrita
```

```
4           char buffer[256];
5           int fileHandle;
6
7
8           sprintf(buffer, "/sys/class/pwm/pwmchip0/pwm%d/
                duty_cycle", port);
9
10          fileHandle = open(buffer, O_WRONLY);
11          if(ERROR == fileHandle){
12                  puts("Impossivel abrir o arquivo:");
13                  puts(buffer);
14                  return(-1);
15          }
16          sprintf(buffer, "%d", dutycycle);
17          write(fileHandle, buffer, strlen(buffer));
18          close(fileHandle);
19          return(0);
20
21  }
```

Na função principal:

1. Abrir pino GPIO 8 como entrada

```
1         openGPIO(GP_PROXY, GPIO_DIRECTION_IN);
```

2. Abrir pino GPIO 8 como saída

```
1         openGPIO(GP_LED, GPIO_DIRECTION_OUT);
```

3. Definir o PWM 3 como entrada, estabeleçer o tempo em que disparará o buzzer, e por fim definir o inicio do ciclo de trabalho.

```
1         //definir parametros PWM
2         openPWM(GP_PWM);
3         setPWMPeriod(1000000,GP_PWM);
4         enablePWM(1,GP_PWM);
5         setPWMDutyCycle(0,GP_PWM);
```

Ainda na função principal criar um *loop* infinito para ler continuamente dados do sensor de proximidade. Caso o valor de proximidade seja elevada (1 no programa), alterar o ciclo de funcionamento da saída PWM 3. Alternar entre 200.000 e 500.000 valores do ciclo de trabalho para fazer um tipo de ambulância do som emitido pelo buzzer. Junto com a mudança do ciclo de trabalho em PWM 3, alternar a luz LED no GPIO 13.

A função a seguir faz a leitura verificando o estado do GPIO.

```
1 int readGPIO(int fileHandle,int gpio){
```

```
2         int value;
3         //Reabrir o arquivo novamente em modo de leitura, uma vez
             que os dados nao foi atualizados.
4         fileHandle = openFileForReading(gpio);
5         read(fileHandle, &value, 1);
6
7         if('0' == value){
8               // Current GPIO status low
9                 value = 0;
10        }
11        else{
12              // Current GPIO status high
13                value = 1;
14        }
15        close(fileHandle);
16        return value;
17  }
```

A função a seguir fará o papel de gravar o estado do GPIO.

```
1   int writeGPIO(int fHandle, int val)
2   {
3         if(val ==  0){
4                 // Definir GPIO status baixo
5                 write(fHandle, "0", 1);
```

```
6           }
7           else{
8                   // Definir GPIO status alto
9                   write(fHandle, "1", 1);
10          }
11          return(0);
12
13  }
```

O *loop* na função principal é apresentado a seguir.

```
1   //Inicando um loop infinito para receber informacao do sensor
        proximidade
2   int proxyValue = 0;
3
4   while(1==1){
5           proxyValue = readGPIO(fileHandleGPIO_PROXY,GP_PROXY);
6           if(proxyValue == 1){
7                   if(duty_cycle == 500000){
8                           duty_cycle = 200000;
9                           writeGPIO(fileHandleGPIO_LED, 0);
10                  }
11                  else{
12                          duty_cycle = 500000;
13                          writeGPIO(fileHandleGPIO_LED, 1);
```

```
14                  }
15                  setPWMDutyCycle(duty_cycle,GP_PWM);
16          }
17          else{
18                  duty_cycle = 50000;
19                  setPWMDutyCycle(0,GP_PWM);
20                  writeGPIO(fileHandleGPIO_LED, 0);
21          }
22      usleep(1000*400);
23  }
```

E, finalmente, fechar as portas GPIO e PWM.

```
1           closeGPIO(GP_LED, fileHandleGPIO_LED);
2           closeGPIO(GP_PROXY, fileHandleGPIO_PROXY);
3           closePWM(GP_PWM);
```

O código completo está a seguir. Para executá-lo verifique os passos de compilação presentes no Capítulo 4.

```
1   /*
2   * Copyright (c) 2014 Intel Corporation  All Rights Reserved.
3   * Author: Raghavendra Ural
4   * Based on LEDblink.c by Nandkishor Sonar.
5   */
6
7   #include <stdlib.h>
```

```
8  #include <stdio.h>
9  #include <fcntl.h>
10 #include <unistd.h>
11 #include <string.h>
12
13 #define GP_LED                  (39) // GPI13 is GP LED
14 #define GP_PROXY                (26) // GPIO8 is PROXIMITY INPUT
15 #define GP_PWM                  (3) // PWM3  is PWM output
16 #define GPIO_DIRECTION_IN       (1)
17 #define GPIO_DIRECTION_OUT      (0)
18 #define ERROR                   (-1)
19
20
21 int flag = 1;
22 int duty_cycle = 0;
23
24 int openFileForReading(gpio)
25 {
26     char buffer[256];
27     int fileHandle;
28
29     sprintf(buffer, "/sys/class/gpio/gpio%d/value", gpio);
30
31     fileHandle = open(buffer, O_RDONLY);
32     if(ERROR == fileHandle){
```

```
33              puts("Unable to open file:");
34              puts(buffer);
35              return(-1);
36      }
37      return(fileHandle);  //This file handle will be used in read/
           write and close operations.
38  }
39
40  int openPWM(int port)
41  {
42      char buffer[256];
43      int fileHandle;
44      int fileMode;
45
46      //Export GPIO
47      fileHandle = open("/sys/class/pwm/pwmchip0/export", O_WRONLY);
48
49      if(ERROR == fileHandle){
50          puts("Error: Unable to opening /sys/class/gpio/export");
51          return(-1);
52      }
53
54          sprintf(buffer, "%d", port);
55      write(fileHandle, buffer, strlen(buffer));
56      close(fileHandle);
```

```
57      sleep(1);
58      return 0;
59  }
60
61  int enablePWM(int enable,int port){
62
63          char buffer[256];
64          int fileHandle;
65
66
67          //Enable PWM
68      sprintf(buffer, "/sys/class/pwm/pwmchip0/pwm%d/enable", port);
69      fileHandle = open(buffer, O_WRONLY);
70      if(ERROR == fileHandle){
71          puts("Unable to open file:");
72          puts(buffer);
73          return(-1);
74      }
75
76          sprintf(buffer, "%d", enable);
77      write(fileHandle, buffer, strlen(buffer));
78      return 0;
79  }
80
81  int setPWMPeriod(int period, int port){
```

```
82
83      //Open GPIO for Read / Write
84          char buffer[256];
85          int fileHandle;
86
87
88          sprintf(buffer, "/sys/class/pwm/pwmchip0/pwm%d/period",
                port);
89
90      fileHandle = open(buffer, O_WRONLY);
91      if(ERROR == fileHandle){
92          puts("Unable to open file:");
93          puts(buffer);
94          return(-1);
95      }
96          sprintf(buffer, "%d", period);
97      write(fileHandle, buffer, strlen(buffer));
98
99      close(fileHandle);
100     return(0);
101
102 }
103
104 int setPWMDutyCycle(int dutycycle, int port){
105
```

```
106         //Open GPIO for Read / Write
107         char buffer[256];
108         int fileHandle;
109
110         sprintf(buffer, "/sys/class/pwm/pwmchip0/pwm%d/duty_cycle",
                port);
111
112     fileHandle = open(buffer, O_WRONLY);
113     if(ERROR == fileHandle){
114         puts("Unable to open file:");
115         puts(buffer);
116         return(-1);
117     }
118
119         sprintf(buffer, "%d", dutycycle);
120     write(fileHandle, buffer, strlen(buffer));
121     close(fileHandle);
122     return(0);
123
124 }
125
126
127 int openGPIO(int gpio, int direction ){
128
129     char buffer[256];
```

```
130     int fileHandle;
131     int fileMode;
132
133     //Export GPIO
134     fileHandle = open("/sys/class/gpio/export", O_WRONLY);
135
136     if(ERROR == fileHandle){
137         puts("Error: Unable to opening /sys/class/gpio/export");
138                 return(-1);
139     }
140
141     sprintf(buffer, "%d", gpio);
142     write(fileHandle, buffer, strlen(buffer));
143     close(fileHandle);
144
145     //Direction GPIO
146     sprintf(buffer, "/sys/class/gpio/gpio%d/direction", gpio);
147     fileHandle = open(buffer, O_WRONLY);
148
149     if(ERROR == fileHandle){
150         puts("Unable to open file:");
151         puts(buffer);
152         return(-1);
153     }
154
```

```
155         if (direction == GPIO_DIRECTION_OUT){
156         // Set out direction
157         write(fileHandle, "out", 3);
158         fileMode = O_WRONLY;
159     }
160         else{
161         // Set in direction
162         write(fileHandle, "in", 2);
163         fileMode = O_RDONLY;
164     }
165
166     close(fileHandle);
167
168     //Open GPIO for Read / Write
169     sprintf(buffer, "/sys/class/gpio/gpio%d/value", gpio);
170
171     fileHandle = open(buffer, fileMode);
172
173     if(ERROR == fileHandle){
174                 puts("Unable to open file:");
175         puts(buffer);
176         return(-1);
177     }
178
```

```
179     return(fileHandle);  //This file handle will be used in read/
            write and close operations.
180
181 }
182
183 int writeGPIO(int fHandle, int val){
184
185         if(val ==  0){
186         // Set GPIO low status
187         write(fHandle, "0", 1);
188     }
189     else{
190         // Set GPIO high status
191         write(fHandle, "1", 1);
192     }
193     return(0);
194
195 }
196
197 int readGPIO(int fileHandle, int gpio){
198
199         int value;
200
201         //Reopening the file again in read mode, since data was not
                refreshing.
```

```
202     fileHandle = openFileForReading(gpio);
203     read(fileHandle, &value, 1);
204
205     if('0' == value){
206         // Current GPIO status low
207         value = 0;
208     }
209     else{
210         // Current GPIO status high
211         value = 1;
212     }
213
214         close(fileHandle);
215     return value;
216 }
217
218
219 int closePWM(int pwm){
220
221         char buffer[256];
222     int fileHandle;
223
224     fileHandle = open("/sys/class/gpio/unexport", O_WRONLY);
225
226         if(ERROR == fileHandle){
```

```
227         puts("Unable to open file:");
228         puts(buffer);
229         return(-1);
230     }
231
232     sprintf(buffer, "%d", pwm);
233     write(fileHandle, buffer, strlen(buffer));
234     close(fileHandle);
235
236     return(0);
237 }
238
239
240 int closeGPIO(int gpio, int fileHandle){
241         char buffer[256];
242         close(fileHandle); //This is the file handle of opened GPIO
                for Read / Write earlier.
243         fileHandle = open("/sys/class/gpio/unexport", O_WRONLY);
244         if(ERROR == fileHandle){
245                 puts("Unable to open file:");
246                 puts(buffer);
247                 return(-1);
248         }
249         sprintf(buffer, "%d", gpio);
250         write(fileHandle, buffer, strlen(buffer));
```

```
251         close(fileHandle);
252 
253         return(0);
254 }
255 int main(void){
256     int fileHandleGPIO_LED;
257     int fileHandleGPIO_PROXY;
258     int i=0;
259 
260         puts("Starting proximity reader on Galileo board.");
261     fileHandleGPIO_PROXY = openGPIO(GP_PROXY, GPIO_DIRECTION_IN);
262     if(ERROR ==  fileHandleGPIO_PROXY){
263         puts("Unable to open toggle Proximity port #8");
264         return(-1);
265     }
266     fileHandleGPIO_LED = openGPIO(GP_LED, GPIO_DIRECTION_OUT);
267 
268     if(ERROR ==  fileHandleGPIO_LED){
269         puts("Unable to open toggle LED port #13");
270         return(-1);
271     }
272 
273 
274     //Switch off the LED before starting.
275     writeGPIO(fileHandleGPIO_LED, 0);
```

```
276
277     //set PWM parameters
278     openPWM(GP_PWM);
279     setPWMPeriod(1000000,GP_PWM);
280     enablePWM(1,GP_PWM);
281         setPWMDutyCycle(0,GP_PWM);
282
283
284     //Start an infinite loop to keep polling for proximity info
285     int proxyValue = 0;
286     while(1==1){
287                 proxyValue = readGPIO(fileHandleGPIO_PROXY,GP_PROXY
                        );
288                 if(proxyValue == 1){
289                         if(duty_cycle == 500000){
290                                 duty_cycle = 200000;
291                         writeGPIO(fileHandleGPIO_LED, 0);
292                         }
293                         else{
294                                 duty_cycle = 500000;
295                         writeGPIO(fileHandleGPIO_LED, 1);
296                         }
297                         setPWMDutyCycle(duty_cycle,GP_PWM);
298                 }
299                 else{
```

```
300                         duty_cycle = 50000;
301                         setPWMDutyCycle(0,GP_PWM);
302                 writeGPIO(fileHandleGPIO_LED, 0);
303                 }
304         usleep(1000*400);
305     }
306
307
308     closeGPIO(GP_LED, fileHandleGPIO_LED);
309     closeGPIO(GP_PROXY, fileHandleGPIO_PROXY);
310     closePWM(GP_PWM);
311
312     puts("Finished BURGLER ALARM on Galileo board.");
313     return 0;
314 }
```

5.3 Verificador de E-mails não lidos

5.3.1 Objetivo

O objetivo desse exemplo é exibir o número de mensagens de emails não lidos em tempo real em um *display* utilizando a placa Galileo [4]. Esse exemplo funciona apenas para contas de email do Gmail.

5.3.2 Componentes eletrônicos necessários

- Placa de desenvolvimento Intel Galileo
- OpenSegment Shield
- Placa para conexão a internet Wifi

5.3.3 Script em python

Esse script é a chave para o funcionamento desse exemplo. Ele efetua *logs* no email do usuário e faz uma busca na caixa de entrada do gmail procurando apenas *emails* que não foram lidos e retorna essa quantidade.

```
1  # pyMailCheck.py - Logs em seu gmail e imprime o numero de e-mails
       nao lidos.
2  # Coloque este arquivo no nivel superior de seu cartao SD de
       Galileo.
3
4  # Usado para conectar a um servidor IMAP4.
5  import imaplib
6
7  # Conectar-se a um sever IMAP4 sobre SSL , porta 993
8  obj = imaplib.IMAP4_SSL('imap.gmail.com', '993')
9
10 # Identificar o usuario e senha
```

```
11 obj.login('my_email_address@gmail.com','myPassword')
12
13 obj.select() # Select a the 'INBOX' mailbox (default parameter)
14
15 # Pesquisa caixa de email ( None) charset, com criterio : "
      invisivel ". Ira retornar uma tupla, pegue a segunda parte e
16 # Divida cada string em uma lista, e retorna o comprimento dessa
      lista:
17
18 print len(obj.search(None,'UnSeen')[1][0].split())
```

5.3.4 Programa em *wiring*

```
1 /* Galileo Email Checker
2    by: Jim Lindblom
3        SparkFun Electronics
4    date: January 7, 2013
5 */
6
7 #include <SPI.h> // SPI e usado para controlar o display
      OpenSegment
8 #include <WiFi.h> // WiFi (Pode ser trocado por Ethernet)
9 #include <SD.h> // A biblioteca SD e usado para ler um arquivo
      temporario,
```

```
10                  // onde o script py armazena uma contagem de e-mail
                        nao lidos.
11
12  ////////////////////
13  // definicao de WiFi //
14  ////////////////////
15  char ssid[] = "WiFiSSID"; //  o seu SSID da rede ( nome)
16  char pass[] = "WiFiPassword";    // sua senha de rede ( para usar
       WPA, ou usar como chave para WEP)
17  IPAddress ip;
18  int status = WL_IDLE_STATUS;
19
20  ////////////////////
21  // Definicao de pinos //
22  ////////////////////
23  const int SS_PIN = 10; // SPI escravo seleto pino (10 no display)
24  const int STAT_LED = 13; // O status do Galileo LED no pino 13
25
26  ////////////////////////////////
27  // Email-Checking Variaveis Global //
28  ////////////////////////////////
29  const int emailUpdateRate = 10000; // Taxa de atualizacao em ms (
       10 s)
30  long emailLastMillis = 0; // Armazena o ultimo e-mail de
       verificacao
```

```
31 int emailCount = 0; // Armazena a ultima contagem de e-mail
32
33 // setup () inicializa o OpenSegment , o cartao SD e WiFi
34 // Se existir sucesso, o LED de status acende.
35 void setup()
36 {
37   pinMode(STAT_LED, OUTPUT);
38   digitalWrite(STAT_LED, LOW);
39
40   // Monitor serial e usado para depuracao
41   Serial.begin(9600);
42   // inicia SPI e redefine o display
43   initDisplay();
44   // Inicializa a classe SD
45   initSDCard();
46
47   // Inicializar WiFI . Em caso de sucesso ligar o status de LED.
       Em falha,
48   // Imprimir um erro no LED, ficar em um loop infinito.
49   if (initWiFi() == 1)
50     digitalWrite(STAT_LED, HIGH);
51   else
52   {
53     SPIWriteString("----", 4);
54     while (1);
```

```
55    }
56  }
57
58  // Faz um loop () checa para a mensagem nao lida contar cada
       emailUpdateRate
59  // Milissegundos. Se a contagem mudou, atualiza a exibicao.
60  void loop()
61  {
62    // Verificar e-mail apenas se emailUpdateRate ms se passaram.
63    if (millis() > emailLastMillis + emailUpdateRate)
64    {
65          // atualiza emailLastMillis
66      emailLastMillis = millis();
67
68      // Recebe a contagem de e-mail nao lido, e armazenar na
           variavel temporaria
69      int tempCount = getEmailCount();
70      if (tempCount != emailCount)
71      {
72        // atualiza o valor da varial emailCount
73        emailCount = tempCount;
74
75        // Imprimir a contagem nao lida
76        printEmailCount(emailCount);
77      }
```

```
78    }
79
80    // Bit de protecao no caso de derrapagens Millis:
81    if (millis() < emailLastMillis)
82      emailLastMillis = 0;
83  }
84
85  // PrintEmailCount (emails) imprime o numero no
86  // Monitor serial, e no display OpenSegment.
87  void printEmailCount(int emails)
88  {
89    Serial.print("Voce tem ");
90    Serial.print(emails);
91    Serial.println(" email nao lidos.");
92
93    // Limpa display
94    SPIWriteByte('v');
95    for (int i=3; i>= 0; i--)
96    {
97      SPIWriteByte((int) emails / pow(10, i));
98      emails = emails - ((int)(emails / pow(10, i)) * pow(10, i));
99    }
100 }
101
102 // GetEmailCount executa o script pyMail.py, e le a saida.
```

```
103 // Vai retornar o valor impresso pelo script pyMail.py.
104 int getEmailCount()
105 {
106   int digits = 0;
107   int temp[10];
108   int emailCnt = 0;
109
110   // Enviar uma chamada de sistema para executar o nosso script
        python e via o
111   // Saida do script para um arquivo.
112   system("python /media/realroot/pyMail.py > /media/realroot/emails
        ");
113
114   // Abrir e-mails para a leitura
115   File emailsFile = SD.open("emails");
116
117   // Ler e-mails de ate chegarmos ao final ou uma nova linha
118   while ((emailsFile.peek() != '\n') && (emailsFile.available()))
119     temp[digits++] = emailsFile.read() - 48; // Converter de ASCII
          para um numero inteiro
120
121   // fechar o arquivo de emails
122   emailsFile.close();
123
124   // remove o arquivo emails
```

```
125   system("rm /media/realroot/emails");
126
127   // concatena os digitos inidividual em um unico numero inteiro:
128   for (int i=0; i<digits; i++)
129     emailCnt += temp[i] * pow(10, digits - 1 - i);
130
131   return emailCnt;
132 }
133
134 // InitDisplay () inicia o SPI, e limpa o visor para "0000"
135 void initDisplay()
136 {
137   initSPI();
138   SPIWriteByte('v');
139   SPIWriteString("0000", 4);
140 }
141
142 // initSPI() inicia a classe SPI
143 void initSPI()
144 {
145   pinMode(SS_PIN, OUTPUT);
146   digitalWrite(SS_PIN, HIGH);
147
148   SPI.begin();
149   SPI.setBitOrder(MSBFIRST);
```

```
150   SPI.setClockDivider(SPI_CLOCK_DIV64);
151   SPI.setDataMode(SPI_MODE0);
152 }
153
154 // initSDCard() inicia a classe SD
155 void initSDCard()
156 {
157   SD.begin();
158 }
159
160 // InitWiFi ( ) inicializa WiFi , com ligacao ao SSID / chave de
      acesso
161 // Combinacao definido em cima.
162 // Esta funcao tentara 3 conexoes com WiFi. Se nos
163 // Suceda a 1 e retornado, e o endereco IP e impresso.
164 // Se as conexoes falharem, um 0 e retornado
165 uint8_t initWiFi()
166 {
167   byte timeout = 0;
168
169   while ((status != WL_CONNECTED) && (timeout < 3))
170   {
171     Serial.print("Setting up WiFi : attempt ");
172     Serial.println(timeout + 1);
```

```
173     // Conectar-se a rede WPA / WPA2 . Altere esta linha se estiver
            usando
174     // Rede aberta ou WEP:
175     status = WiFi.begin(ssid, pass);
176     //delay(10000); // Aguarda 10 segundos entre as tentativas de
            conexao
177     timeout++;
178   }
179
180   if (timeout >= 3)
181     return 0;
182   else
183   {
184     Serial.print("Connected to ");
185     ip = WiFi.localIP();
186     Serial.println(WiFi.SSID());
187     Serial.println(ip);
188     // !!! Idea aqui: endereco IP de impressao para display de sete
            segmentos .
189     // Isso vai torna-lo mais facil saber qual o endereco de ssh
            sobre a placa
190     // Se precisarmos .
191     return 1;
192   }
193 }
```

```
194
195 // SPIWriteString vai escrever uma matriz de caracteres (str ) de
       comprimento
196 // Len para SPI . Faca ordem e [ 0 ] , [ 1 ], ... ,[ len - 1 ].
197 void SPIWriteString(char * str, uint8_t len)
198 {
199   digitalWrite(SS_PIN, LOW);
200   for (int i=0; i<len; i++)
201   {
202     SPI.transfer(str[i]);
203   }
204   digitalWrite(SS_PIN, HIGH);
205
206 }
207
208 // SPIWriteByte vai escrever um unico byte de SPI.
209 void SPIWriteByte(uint8_t b)
210 {
211   digitalWrite(SS_PIN, LOW);
212   SPI.transfer(b);
213   digitalWrite(SS_PIN, HIGH);
214 }
```

Para executá-lo, verifique os passos de compilação e execução no Capítulo 4.

Referências Bibliográficas

[1] Blinking leds – manipulating digital gpios on the intel® galileo board with the iot development kit. https://software.intel.com/pt-br/articles/blinking-leds-manipulating-digital-gpios-on-the-intel-galileo-board-with-the-iot. acessado em 17/08/2015.

[2] Burglar alarm using the intel® galileo board. https://software.intel.com/pt-br/articles/burglar-alarm-using-the-intel-galileo-board. acessado em 16/08/2015.

[3] Desvendando yocto project (poky) - primeiros passos. http://pt.slideshare.net/marcochella/galileo-04?related=1. acessado em 15/05/2015.

[4] Enginursday: Exploring the arduino/intel galileo. https://www.sparkfun.com/news/1360. acessado em 16/08/2015.

[5] Oque é poky. https://www.yoctoproject.org/tools-resources/projects/poky. acessado em 14/05/2015.

[6] Oque é yocto. https://www.yoctoproject.org/about. acessado em 13/05/2015.

[7] Intel. Intel galileo gen 1 development board datasheet. `https://www-ssl.intel.com/content/www/br/pt/embedded/products/galileo/galileo-g1-datasheet.html?wapkw=intel+galileo`. acessado em 2/07/2015.

[8] Intel. Intel galileo gen2 development board datasheet. `https://www-ssl.intel.com/content/www/br/pt/embedded/products/galileo/galileo-g2-datasheet.html?wapkw=intel+galileo`. acessado em 2/07/2015.

[9] Intel. Placa de desenvolvimento intel galileo gen2 - dê mais poder ao seu protótipo. `http://www.intel.com.br/content/www/br/pt/do-it-yourself/galileo-maker-quark-board.html`. acesssado em 2/07/2015.