Ciência da Computação II

Ciência da Computação II

Jefferson Nataline Rosa do Nascimento

© 2018 por Editora e Distribuidora Educacional S.A.
Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Editora e Distribuidora Educacional S.A.

Presidente
Rodrigo Galindo

Vice-Presidente Acadêmico de Graduação e de Educação Básica
Mário Ghio Júnior

Conselho Acadêmico
Ana Lucia Jankovic Barduchi
Camila Cardoso Rotella
Danielly Nunes Andrade Noé
Grasiele Aparecida Lourenço
Isabel Cristina Chagas Barbin
Lidiane Cristina Vivaldini Olo
Thatiane Cristina dos Santos de Carvalho Ribeiro

Revisão Técnica
Pedro Donizeti Bolanho
Ruy Flávio de Oliveira

Editorial
Camila Cardoso Rotella (Diretora)
Lidiane Cristina Vivaldini Olo (Gerente)
Elmir Carvalho da Silva (Coordenador)
Letícia Bento Pieroni (Coordenadora)
Renata Jéssica Galdino (Coordenadora)

Dados Internacionais de Catalogação na Publicação (CIP)

Nascimento, Jefferson Nataline Rosa do
N244c Ciência da computação II / Jefferson Nataline Rosa do Nascimento. – Londrina : Editora e Distribuidora Educacional S.A., 2018.
232 p.

ISBN 978-85-522-1092-4

1. Sistemas embarcados. 2. Multimídia. 3. Microcontroladores. I. Nascimento, Jefferson Nataline Rosa do. II. Título.

CDD 004

Thamiris Mantovani CRB-8/9491

2018
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: http://www.kroton.com.br/

Sumário

Unidade 1 | Fundamentos de microcontroladores 7
Seção 1.1 - Arquitetura de microcontroladores 9
Seção 1.2 - Periféricos básicos 27
Seção 1.3 - Periféricos avançados 43

Unidade 2 | Desenvolvimento de projetos microcontrolados 61
Seção 2.1 - Ambientes de desenvolvimento e simulação 63
Seção 2.2 - Introdução a projetos microcontrolados 79
Seção 2.3 - Projetos avançados com microcontroladores 97

Unidade 3 | Sistemas embarcados 117
Seção 3.1 - Introdução a sistemas embarcados 119
Seção 3.2 - Sistemas operacionais de tempo real 135
Seção 3.3 - Introdução a linux embarcado 151

Unidade 4 | Desenvolvimento de projetos embarcados 169
Seção 4.1 - Projetos de software utilizando sistemas operacionais de tempo real 171
Seção 4.2 - Desenvolvimento de projetos embarcados 190
Seção 4.3 - Conceitos avançados de sistemas embarcados 205

Palavras do autor

Vivemos atualmente em uma nova era de revolução da indústria. Uma era que pretende transformar a maneira como nós, humanos, nos comunicamos, assim como a forma como nossas máquinas trocam informação entre si, em uma cadeia de rupturas tecnológicas que possui o potencial de reinventar até mesmo nossa vida pessoal e profissional. Nessa disciplina teremos a oportunidade de conhecer os sistemas microcontrolados e embarcados, que estão firmados no centro desse processo tecnológico.

Buscaremos juntos nessa sequência de aprendizado desenvolver competências que são chaves para dominar os conceitos fundamentais do universo de sistemas embarcados. Inicialmente, entenderemos o que são sistemas microcontrolados e quais suas diferenças em relação aos sistemas puramente microprocessados, e com essa base desenvolveremos aplicações simples e avançadas que demonstrem a aplicação de um microcontrolador. Já em um segundo momento, vamos focar em conhecer os sistemas embarcados microprocessados, que possuem maior poder computacional e envolvem paradigmas computacionais mais avançados, como sistemas operacionais Linux e sistemas de tempo real.

Na primeira unidade estudaremos, a princípio, as principais arquiteturas de hardware e software de um sistema microcontrolado. Com esse conhecimento consolidado, tomaremos a arquitetura AVR como base para o estudo dos principais periféricos de um microcontrolador moderno.

Na segunda unidade, vamos aprender a preparar um ambiente de desenvolvimento adequado para o trabalho com microcontroladores. Utilizaremos o conhecimento teórico adquirido na unidade anterior para conceber projetos microcontrolados, desde circuitos básicos até combinações mais avançadas.

Na terceira unidade, focaremos nossos estudos em sistemas computacionais mais avançados: os sistemas embarcados microprocessados. Vamos compreender as principais diferenças entre sistemas microcontrolados e sistemas microprocessados, e assim partiremos para temas mais específicos, como sistemas operacionais para eletrônica embarcada.

Na quarta e última unidade, vamos praticar nosso conhecimento em projetos envolvendo sistemas operacionais Linux e sistemas de tempo real. Veremos ao final as principais tendências de mercado para esse ramo da tecnologia, e serão introduzidos alguns tópicos avançados para a continuidade dos seus estudos.

A área de sistemas microcontrolados e microprocessados é fascinante, por apresentar técnicas e ferramentas que são indispensáveis em todo e qualquer projeto atual que envolva inteligência embarcada. Porém é um campo bastante vasto, que exige do profissional e estudante da área muito entusiasmo e dedicação para manter-se sempre atualizado e disposto a absorver conhecimentos.

Fundamentos de microcontroladores

Convite ao estudo

Nesta primeira unidade de ensino iniciaremos nossa jornada pelo estudo dos microcontroladores, que estão presentes na grande maioria dos produtos que utilizamos cotidianamente, como automóveis, eletrodomésticos e computadores. Começaremos por entender como é projetado e do que é constituído um microcontrolador moderno, considerando principalmente o tipo de arquitetura que foi empregada em sua concepção. Veremos que atualmente existem no mercado centenas de milhares de modelos distintos de microcontroladores, mas que a grande maioria dos mesmos foi projetado através de uma combinação de dois formatos de arquitetura: um para definir como será o acesso do processador à memória e aos periféricos, e outro que definirá como as instruções serão criadas e processadas. Após essa parte, estudaremos os periféricos que constituem um microcontrolador. Entenderemos qual a função de cada um e como configurá-los para que tenhamos um projeto microcontrolado versátil e eficaz.

A escolha de um microcontrolador baseando nos tópicos relacionados anteriormente é muito semelhante ao processo de escolha e compra de um computador. Geralmente, tentamos otimizar a escolha de uma arquitetura tendo em vista a aplicação da mesma bem como o valor que desejamos investir. Nesse cenário, devemos listar as características principais do nosso computador, tais como: velocidade da CPU, quantidade de memória, placa de vídeo, periféricos multimídia, etc. Veremos nos nossos estudos que os microcontroladores possuem um processo análogo e um estudo refinado é necessário para se realizar a melhor escolha.

Ao final da unidade, devemos ser capazes de utilizar esse conhecimento para saber diferenciar e selecionar o melhor microcontrolador para um dado projeto, bem como configurar seus periféricos e implementar lógicas de operação com os mesmos.

Nessa unidade, você tomará o lugar de um recém-formado cientista da computação, que fora convidado para trabalhar em uma empresa que desenvolve projetos de domótica (automação residencial), e seu objetivo será o de auxiliar tecnicamente o time de desenvolvimento de produtos. O projeto em questão refere-se a um sistema de acionamento automático de atuadores, como janelas, luminárias e sistemas multimídia que, através da leitura de entradas discretas (botões e interruptores) e analógicas (sensores de temperatura e umidade), deverá realizar o processamento e controle do ambiente. Seu primeiro desafio será o de apresentar boas soluções de arquiteturas de microcontroladores que possam ser utilizados no mais novo produto que a empresa pretende lançar no mercado. Em um segundo momento, após a escolha do melhor componente, você ajudará o time técnico a compreender e manipular os periféricos desse novo microcontrolador de forma a implementar todos os requisitos funcionais que lhes foram exigidos como escopo de projeto.

Sabemos que muitos são os modelos de microcontroladores presentes no mercado, mas como podemos ser o mais assertivo possível na escolha do melhor componente? Podemos considerar os periféricos como blocos de hardware complementares à CPU e que desempenham funções específicas. Quais critérios devemos seguir para saber quando será necessário utilizar o processamento da CPU e quando é mais eficiente a utilização de um desses periféricos?

Seção 1.1

Arquitetura de microcontroladores

Diálogo aberto

Atualmente vivenciamos uma realidade em que a grande maioria dos objetos com que interagimos em nosso cotidiano apresenta algum tipo de inteligência embarcada, capaz de expandir nossa experiência com eles e com o mundo ao nosso redor. Todos os equipamentos eletrônicos que possuem tal inteligência apresentam dentro de si um sistema de processamento exclusivamente dedicado e otimizado para aquela aplicação, por exemplo: um processador presente em um micro-ondas é completamente diferente de um outro presente em um avião comercial. Porém, apesar de diferirem em termos de capacidade de processamento e diversidade de periféricos, ambos processadores podem ter sido concebidos baseados na mesma arquitetura computacional.

Nesta seção iniciaremos por entender o que é e como opera um microcontrolador. Passaremos por uma breve revisão histórica para visitar seus primórdios até chegarmos em suas configurações atuais. Falaremos um pouco sobre suas similaridades e diferenças com outros dispositivos de processamento, como os microprocessadores, e quais são suas principais diferenças.

Vamos nos aprofundar nas arquiteturas que definem o perfil de um microcontrolador, sejam elas atreladas à comunicação interna entre seu processador e seus periféricos, ou aos seus conjuntos e instruções e seus níveis de complexidade. Por fim, focaremos nossa atenção ao microcontrolador ATMega328, que utilizaremos como exemplo neste curso para concretizar nossos conhecimentos em sistemas microcontrolados.

Lembre-se de que nesta unidade você assumirá o lugar de um recém-contratado cientista da computação, que irá auxiliar uma equipe de técnicos em uma empresa de automação residencial, e cujo o objetivo é o desenvolvimento de um novo produto a ser lançado no mercado: um sistema de controle residencial automático.

Nessa primeira etapa você deve apresentar um relatório técnico no qual descreve as características das principais arquiteturas presentes no mercado. O líder do time de engenharia lhe sugeriu o microcontrolador ATMega328, e lhe solicitou que incluísse em seu relatório uma análise concreta da possibilidade desse modelo atender satisfatoriamente o projeto em questão.

O conteúdo que veremos nessa seção será fundamental para lhe guiar no desenvolvimento dessa atividade, pois entenderemos de maneira geral os tipos de arquiteturas existente e, por fim, focaremos na arquitetura AVR do microcontrolador ATMega328.

Bons estudos!

Não pode faltar

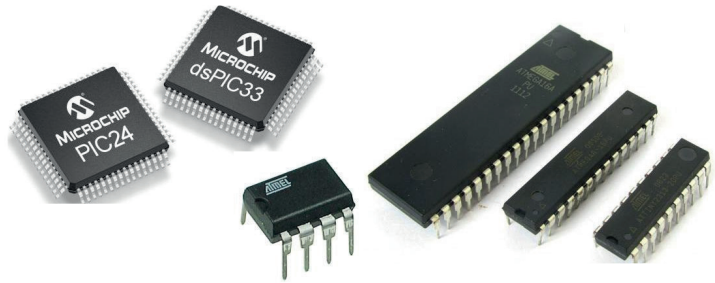
Introdução a microcontroladores

Acredito que você já tenha ouvido falar dos processadores, aqueles que embarcam todos nossos computadores e smartphones. E quanto aos microcontroladores, já ouviu algo a respeito? Nessa seção vamos iniciar nosso estudo sobre esse importante componente tecnológico, partindo do entendimento de suas funcionalidades e de como é constituído.

O microcontrolador pode ser definido como um chip digital embutido que é diretamente responsável pelo controle da grande maioria dos aparelhos eletrônicos que nós utilizamos em nosso cotidiano (LIMA, 2012). Agora mesmo, em sua casa, muitos dos equipamentos que você está habituado a utilizar empregam algum tipo de microcontrolador, como por exemplo sua televisão, seu micro-ondas, seu celular e até mesmo o teclado e o mouse que você deve estar usando para navegar por esse conteúdo. O veículo que você utiliza para se deslocar até seu trabalho ou até a faculdade é repleto de circuitos de controle microcontrolado, desde os mais simples, como os módulos levantadores de vidro e limpadores de para-brisa, até os mais sofisticados, como aqueles que embarcam centrais de injeção de combustível e o módulo de airbag.

Além dos seus periféricos, os microcontroladores também embarcam memórias voláteis e não-voláteis que armazenam seus dados e suas instruções de máquina, a serem executadas sempre que eles estiverem energizados.

Figura 1.1 | Exemplos de chips microcontroladores



Fonte: <http://newtoncbraga.com.br/images/stories/artigos2017/mic0139_0002.jpg>. Acesso em: 8 abr. 2018.

Durante os anos de 1970 e 1971, a empresa Intel Corporation trabalhava na criação do primeiro microprocessador do mundo, e Gary Boone, da Texas Instruments, também trabalhava em um conceito muito parecido, e acabou por criar o microcontrolador. Ele projetou um único chip integrado que poderia armazenar quase todos os circuitos essenciais para compor uma calculadora, restando apenas a tela e o teclado, que não foram incorporados. Esse chip dispunha de um total de cinco mil transistores, fornecendo 3000 bits de memória de programa (375 bytes) e 128 bits de memória de dados, e para programá-lo era necessária a realização de uma série de procedimentos detalhados (CIRCUITSTODAY, 2013).

Desta época até os dias atuais, os microcontroladores passaram por um grande avanço, e outras companhias aderiram à fabricação desse dispositivo. Atualmente, microcontroladores mais modernos dispõem de até um milhão de vezes mais memória que o primeiro exemplar e capacidade de processamento que rivaliza com os melhores processadores do mercado, tudo isso a um custo muito competitivo, visto a enorme demanda dessa tecnologia e a vasta quantidade de fabricantes e modelos.

Arquitetura interna de um microcontrolador

Por que muitos autores de projetos eletrônicos, quando estes são dependentes de controle e processamento de dados, optam por utilizar um microcontrolador no lugar de um microprocessador? Vamos entender as principais vantagens da escolha de um microcontrolador para um projeto eletrônico, e o que o difere da arquitetura de um microprocessador.

Hoje em dia estamos habituados a lidar e interagir com os microprocessadores, que são unidades digitais computacionais presentes em todos os computadores, servidores e até em nossos aparelhos celulares. O microprocessador apresenta a característica determinante de prezar pela velocidade de processamento com a melhor razão de consumo de energia possível, dispondo então em sua arquitetura de componentes totalmente orientados a esta finalidade, destacando-se as unidades lógicas e aritméticas, os registradores e blocos de memória cache (LIMA, 2012). Porém um microprocessador não embarca em si todos os componentes necessários para um completo ciclo computacional, como o caso de memórias não-voláteis para o armazenamento de dados permanentes, memórias voláteis para o carregamento de programas e armazenamento de dados intermitentes e periféricos de entrada e saída, logo todos esses componentes estão presentes externamente ao chip e se conectam com ele através de barramentos especiais.

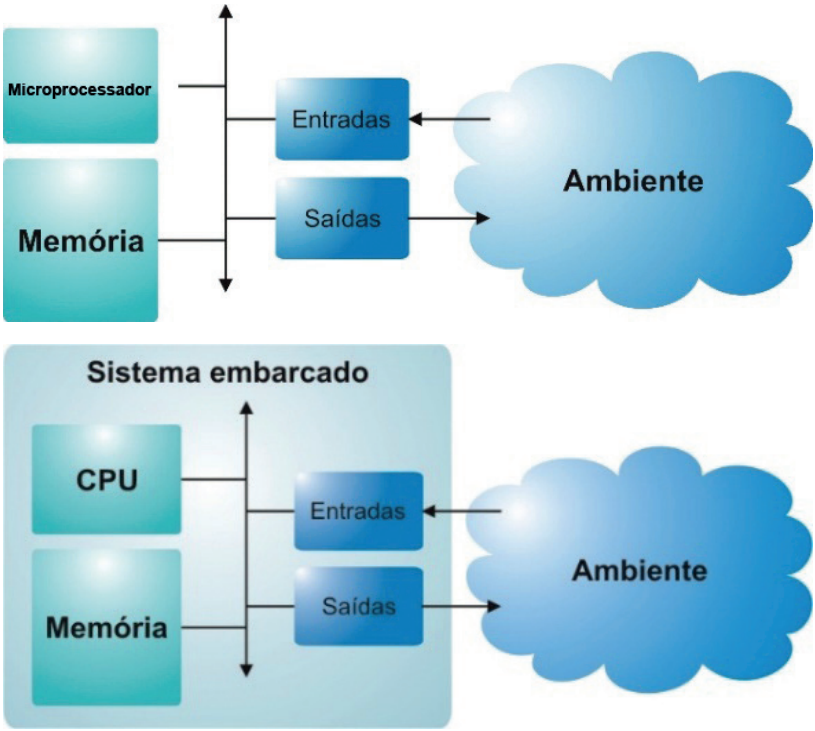
Assimile

Os microprocessadores também desempenham um papel importante no universo da eletrônica embarcada. Em alguns projetos que requisitam grande poder de processamento, ou nos quais seja necessária a utilização de um sistema operacional, o uso de um microprocessador pode ser a única forma de atingir esses objetivos.

Diferentemente do microprocessador, o microcontrolador apresenta como principal característica a de embarcar em si todos ou a maioria dos componentes e periféricos necessários, a fim de lhe conceder total autonomia para armazenar e executar programas, realizar o recebimento de dados por vias diversas e atuar em outros componentes eletrônicos através de saídas digitais ou periféricos de comunicação, ressaltando-se ainda que todo microcontrolador possui uma unidade de processamento.

Toda essa versatilidade vem ao custo do poder computacional desses componentes, que é razoavelmente reduzido em detrimento de unidades de processamento dedicadas.

Figura 1.2 | Sistemas microprocessados e microcontrolados



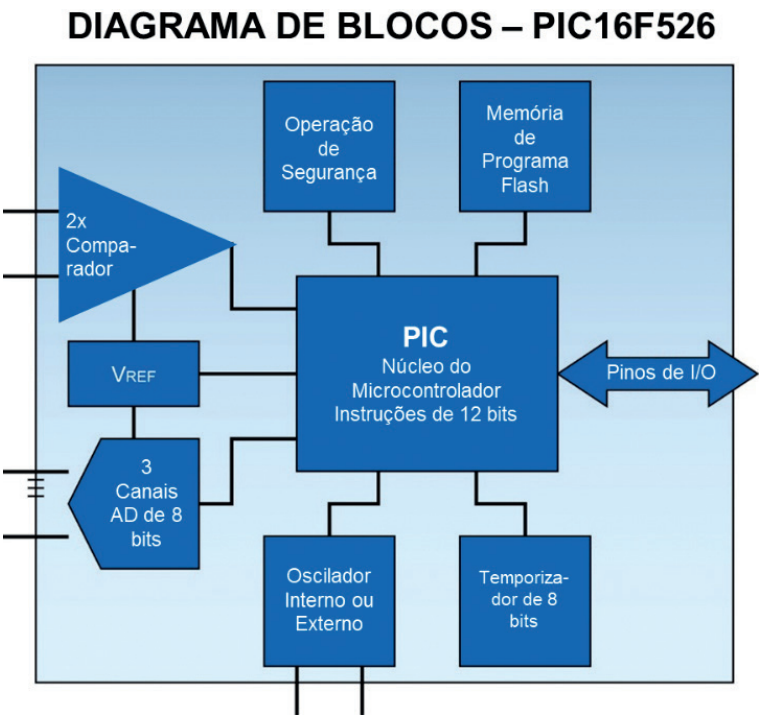
Fonte: elaborada pelo autor.

Como demonstrado na Figura 1.2, em um sistema microprocessado, para que se haja interação entre o ambiente e o sistema de computação é necessário que existam periféricos externos ao processador incumbidos de realizar a leitura de entradas, bem como a atuação condicionada das saídas, evidenciando, da mesma forma, que a memória também é um componente externo ao microprocessador. Já em um sistema embarcado microcontrolado, os componentes principais necessários para a interação com o meio já estão presentes dentro do mesmo chip, inclusive as memórias de dados e programa.

Exemplificando

Uma das famílias de microcontroladores que mais se popularizou entre os estudantes e profissionais da área no Brasil é a linha PIC da empresa Microchip. Esses microcontroladores se destacam pela sua arquitetura consistente, bom desempenho de processamento e grande variedade de periféricos. Na Figura 1.3 a seguir temos o exemplo do modelo PIC16F526 que, além de possuir memórias internas, dispõe também de periféricos de entrada e saída digital, entradas analógicas, temporizadores e circuitos de baixo consumo.

Figura 1.3 | Diagrama de blocos do microcontrolador PIC16F526



Fonte: adaptada de <http://www.embeddedinsights.com/epd/Diagrams/microchip-8-bitpicbaseline.jpg>. Acesso em: 18 mar. 2018.

Arquitetura de microcontroladores e modelos de instruções

Já imaginou a quantidade de diferentes modelos de microcontroladores existentes? O que uma empresa fabricante

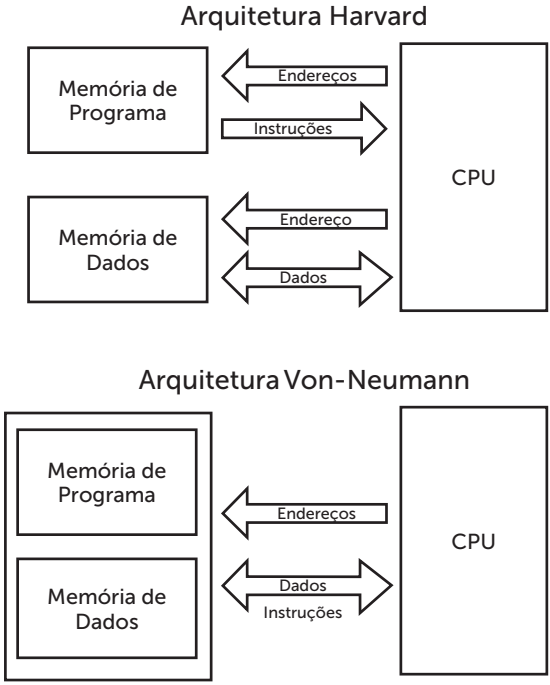
de produtos eletrônicos leva em consideração quando decide escolher um determinado modelo? Vamos compreender agora o cerne desse assunto: os tipos de arquiteturas de microcontroladores.

Como vimos, os microcontroladores apresentam, como uma de suas principais características, a agregação de memórias e periféricos dentro de um mesmo sistema eletrônico. Sabemos também que para que se haja um processo computacional completo é necessário que o processador disponha de um conjunto de instruções armazenadas em memória disponível a ele, e um outro espaço de memória para o armazenamento de variáveis e outros dados voláteis. Dentro de um microcontrolador usualmente encontram-se memórias do tipo *Flash*, que realizam o armazenamento não volátil de suas instruções de programa, e memórias do tipo RAM (Random Access Memory), que são utilizadas para o armazenamento de dados voláteis.

O acesso da unidade de processamento a essas memórias em um microcontrolador ocorre através de barramentos que, de maneira simplificada, são utilizados para trafegar toda a troca de informação dentro do chip.

Quando tratamos da organização de tais barramentos, podemos classificar a arquitetura de um microcontrolador dentre dois tipos distintos: a arquitetura Von-Neumann e a arquitetura Harvard. Na arquitetura Von-Neumann, a busca de dados e instruções não pode ser executada ao mesmo tempo (gargalo da arquitetura), limitação que pode ser superada com a busca antecipada de instruções e/ou com caches de instruções/dados. Já na arquitetura Harvard, os dados e instruções podem ser acessados simultaneamente, o que torna essa arquitetura inerentemente mais rápida que a Von-Neumann (LIMA, 2012).

Figura 1.4 | Arquiteturas clássicas: Harvard e Von-Neumann



Fonte: Lima e Villaça (2012, p. 5).

Outra classificação quanto à arquitetura de um microcontrolador diz respeito ao seu conjunto de instruções, que pode ser:

- CISC: computador de conjunto de instruções complexas (*Complex Instructions Set Computer*).
- RISC: computador de conjunto de instruções reduzidas (*Reduzed Instructions Set Computer*).

A arquitetura RISC utiliza um conjunto de instruções simples, pequeno e geralmente com extensão fixa. Semelhante à RISC, a arquitetura CISC emprega um conjunto simples de instruções, porém utiliza também instruções mais longas e complexas, semelhantes às de alto nível, na elaboração de um programa. As instruções CISC são geralmente variáveis em extensão. Assim, a arquitetura RISC necessita de mais linhas de código para executar a mesma tarefa que uma

arquitetura CISC, a qual dispõe de muito mais instruções (LIMA, 2012).

Para ilustrar a diferença entre ambas as arquiteturas, vamos considerar uma operação simples em que se deseja realizar a multiplicação entre dois números contidos nos endereços 0 e 3 da memória de um microcontrolador, e o resultado dessa operação retornará para a posição 0. Teríamos assim as seguintes instruções em *assembly* para cada tipo de arquitetura:

CISC:

MULT 0, 3

Multiplica o conteúdo do endereço 0 pelo valor contido no endereço 3, e o resultado volta para o endereço 0.

RISC:

LOAD A, 0

LOAD B, 3

MULT A, B

STORE 0, A

Carrega os valores dos endereços 0 e 3 nos registradores fictícios A e B respectivamente, depois realiza a multiplicação dos valores dos registradores, voltando o resultado para A e, por fim, armazena o valor contido no registrador A de volta ao endereço 0.

Perceba que o microcontrolador de arquitetura RISC emprega o conceito de carregar e armazenar (*LOAD*, *STORE*), o que obriga todas as operações aritméticas a passarem pelos registradores de uso geral (A e B), acarretando ao todo em quatro operações.

Já com o microcontrolador de arquitetura CISC o dado pode ser armazenado sem a necessidade do uso explícito dos registradores de uso geral.

Apesar de os microcontroladores de arquitetura RISC utilizarem uma maior quantidade de instruções para executar

a mesma tarefa que um microcontrolador CISC realizaria com somente uma instrução, os microcontroladores RISC são mais velozes que os do tipo CISC, sendo que a grande maioria de suas instruções pode ser executada utilizando somente um ciclo de *clock*. Em contrapartida, os microcontroladores CISC agregam uma significante economia de memória de programa, uma vez que é preciso armazenar somente uma instrução em memória em vez de várias equivalentes a microcontroladores do tipo RISC.

Reflita

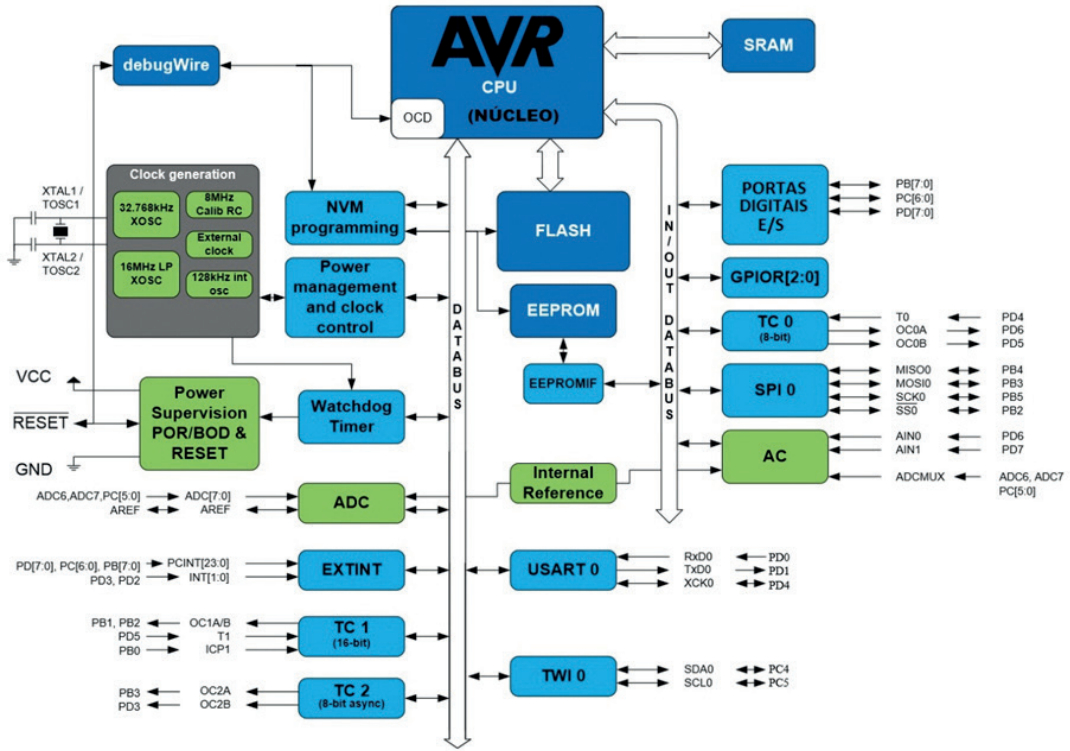
Vimos aqui que microcontroladores com arquitetura RISC apresentam vantagem quando necessitamos de maior velocidade de processamento e possuímos uma quantidade adequada de memória não-volátil; já os microcontroladores CISC, apesar de serem mais lentos em geral, propiciam grande economia quanto à quantidade de instruções armazenadas.

Em um cenário no qual optamos por um microcontrolador com uma das duas arquiteturas detalhadas e nos deparamos com problemas de espaço de memória (RISC) ou velocidade de processamento (CISC), existiriam formas de contornamos esse problema em um projeto?

Arquitetura AVR

Neste curso focaremos nossa atenção a um modelo de microcontrolador muito comum e que se popularizou com grande rapidez através das plataformas de aprendizado Arduino, o ATMega328. Este microcontrolador fabricado pela empresa ATMEL faz parte de uma família de controladores chamada AVR, apresentando 8 bits como tamanho de palavra de processamento, tecnologia CMOS e construído sobre a arquitetura AVR RISC Harvard modificada. Foi projetado para proporcionar um alto desempenho de processamento, podendo alcançar até 20 MIPS (milhões de instruções por segundo), operando em 20MHz. Sua memória Flash para o armazenamento de instruções é de 32Kbytes, e sua memória RAM volátil possui 2KBytes, dispondo ainda de uma memória EEPROM de 1KByte para o armazenamento de dados não-voláteis. O seu diagrama de blocos contendo detalhes de sua arquitetura interna é apresentado na Figura 1.5.

Figura 1.5 | Arquitetura interna ATMega328



Fonte: adaptada de <http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso: 02 maio 2018.

Pesquise mais

Para conhecer mais sobre o microcontrolador ATMega328, leia um artigo que faz uma revisão sobre as principais características desse poderoso dispositivo. ELETRÔNICA 90. **O que é um Microcontrolador? Conheça o ATmega328**. Jan. 2017. Disponível em: <http://eletronica90.com.br/o-que-e-um-microcontrolador/>. Acesso em: 8 abr. 2018.

Sem medo de errar

Nessa unidade, como recém-formado cientista da computação, você foi contratado por uma empresa de automação residencial para auxiliar no desenvolvimento de um sistema automatizado de controle residencial que pretende controlar alguns atuadores como janelas, luminárias e sistemas multimídia através da leitura de entradas discretas e analógicas, como botões, interruptores, sensores ambiente, etc.

Lembre-se de que nessa etapa você foi designado para realizar um relatório comparando as principais arquiteturas de microcontroladores do mercado frente ao projeto proposto, e lhe foi solicitado que, de maneira complementar, você avalie a aplicação do microcontrolador ATMega328 para o devido caso.

Como estudamos nessa seção, um microcontrolador se difere de um microprocessador por incluir periféricos auxiliares em seu chip, bem como memória para programas e dados, e o que difere um modelo de microcontrolador de outro são justamente os tipos de periféricos embarcados, e sua capacidade de memória.

Para essa primeira etapa de construção do relatório, vamos nos ater a realizar um detalhamento dos requisitos do projeto e a listar quais seriam os periféricos necessários para atender o escopo.

Como proposta de controle de saídas, o enunciado do projeto sugere que o sistema deve controlar janelas, luminárias e sistemas multimídias, então vamos elaborar um quadro com cada atuador e sua forma de acionamento.

Quadro 1.1 | Lista de acionamentos e periféricos de saída

Atuador	Tipo de acionamento	Periférico do microcontrolador
Janelas	Motores DC, relê	Saídas digitais, PWM
Luminárias	Relê, Dimmer	Saídas digitais, PWM
Sistema multimídia	Relê, protocolo de comunicação	Saídas digitais, comunicação serial

Fonte: elaborado pelo autor.

Agora faremos o mesmo para os dispositivos de entrada que foram propostos.

Quadro 1.2 | Lista de sensores e periféricos de entrada

Atuador	Tipo de entrada	Periférico do microcontrolador
Botões	Digital	Entradas digitais
Interruptores	Digital	Entradas digitais
Sensor de temperatura	Analógica	Entrada analógica ou protocolo de comunicação
Sensor de umidade	Analógica	Entrada analógica ou protocolo de comunicação

Fonte: elaborado pelo autor.

Dadas as informações que consolidamos, agora podemos realizar uma pesquisa direcionada a escolher modelos de microcontroladores que disponham dos periféricos listados, que são: entradas e saídas digitais (I/O), saída PWM, entradas analógicas, comunicação serial e outros protocolos de comunicação. Como foi solicitado pelo líder técnico, vamos incluir nessa pesquisa o microcontrolador ATMega328, a fim de constatar ou não sua aplicabilidade para este projeto.

Podemos utilizar ferramentas de busca na internet para encontrar os modelos viáveis para a nossa aplicação, mas isso tomaria um tempo considerável e não teríamos a certeza de incluir os melhores microcontroladores em termos de custo/benefício. Uma alternativa é empregar essas ferramentas para conhecer os fabricantes e modelos mais comuns e, a partir daí, utilizar sites dedicados a componentes eletrônicos para realizar uma pesquisa mais refinada em busca dos modelos corretos.

Pesquise mais

Através de uma pesquisa rápida, é possível constatar que, para o mercado brasileiro, os microcontroladores dos fabricantes ATMEL e Microchip são dominantes. Esse fator é importante para um projeto, pois utilizando microcontroladores mais comuns também aumentamos as chances de encontrarmos boas referências para o nosso desenvolvimento, bem como ferramentas de fácil acesso e uso.

Partindo desses dois fabricantes, podemos utilizar sites de vendas de componentes, como a MOUSER ELECTRONICS. Disponível em: <https://br.mouser.com>. Acesso em: 8 abr. 2018 ou a DIGI-KEY ELECTRONICS. Disponível em: <https://www.digikey.com.br>. Acesso em: 8 abr. 2018, para realizar a busca do modelo ideal para o nosso projeto.

Vamos também consolidar essa busca em forma de uma tabela comparativa.

Tabela 1.1 | Tabela comparativa de microcontroladores

Microcontrolador / Periférico	PIC18F452	ATMega328	PIC16F877
Memória Flash	32 kB	32 kB	8 kB
Memória RAM	1,5 kB	2 kB	368 B
I/O Digitais	36	23	33
Entradas analógicas	13	8	8
Resolução analógica	10 bits	10 bits	10 bits
Canais PWM	4	10	2
Comunicação serial	1	2	1
Protocolos de comunicação	I2C, MSSP, PSP, SPI, USART	2-Wire, I2C, SPI, USART	I2C, SPI, USART
Custo unitário	R$ 16,00	R$ 5,30	R$ 19,60

Fonte: elaborada pelo autor.

Podemos concluir que o microcontrolador PIC16F877 é aquele que apresenta pior relação entre custo e periféricos embarcados, e assim a disputa se concentra entre os outros dois modelos. Há um certo equilibro entre o microcontrolador PIC18F452 e o ATMega328, pois cada um dispõe de vantagens pontuais em

relação a seus periféricos, porém o microcontrolador da ATMEL apresenta um custo unitário muito inferior (três vezes menor) que seu concorrente, o que nos levar a concluir que esse modelo é o mais adequado para o projeto em questão.

Avançando na prática

Arquitetura AVR

Descrição da situação-problema

Após ter apresentado o relatório comparativo entre os três modelos de microcontroladores e concluído que o modelo ATMega328 é o mais adequado para a aplicação, a empresa em que você trabalha elaborou uma lista com alguns requisitos mais específicos do produto, e deseja entender quais deles poderiam ser atendidos e quais não seriam atendidos por esse microcontrolador. Os requisitos solicitados foram:

1. O microcontrolador deve possuir uma arquitetura de instruções que preze pela velocidade e não tanto pelo consumo de memória.
2. Deverão ser usados ao menos dois temporizadores para contagem de tempos específicos de acionamento, sendo que um deles deve ter a granularidade de um segundo.
3. O sistema pode incluir uma entrada USB para conexão com outros equipamentos.
4. O microcontrolador deve ser capaz de monitorar a temperatura em sua volta para assegurar que o circuito esteja operando em condições adequadas.

Tendo em vista a arquitetura do microcontrolador ATMega328, você poderia realizar esta análise e definir quais desses requisitos podem ser atingidos com o microcontrolador?

Resolução da situação-problema

Vamos tratar cada item requisitado separadamente:

1. O microcontrolador ATMega328 dispõe de uma arquitetura de barramento Harvard e arquitetura de instruções RISC, dessa forma é possível afirmar que ele preza pela velocidade de

processamento das instruções em detrimento do consumo de memória.

2. O microcontrolador ATMega328 prevê três temporizadores, sendo dois de 8 bits e um de 16 bits. Prevê ainda um módulo que pode ser utilizado em conjunto com um cristal externo de 32,678Hz para contagem de tempo real com escala de um segundo.
3. Esse microcontrolador não possui periférico USB embarcado. Uma alternativa seria utilizar um circuito externo que realizasse a conversão de protocolo USB para TTL (USART), dessa forma o microcontrolador poderia processar a informação de maneira direta.
4. Como vimos, este microcontrolador também apresenta um sensor de temperatura interno, acessível via periférico de conversão analógico-digital e seria, então, capaz de monitorar o ambiente onde for instalado.

Faça valer a pena

1. Um microprocessador difere-se de um microcontrolador por não possuir periféricos e memória embarcados dentro de um mesmo chip, sendo necessário que o microprocessador se conecte a eles via barramento externo em uma placa eletrônica. Já um microcontrolador pode embarcar diversos periféricos distintos em um mesmo componente, em que a variação desses periféricos e suas capacidades de memória definirão o modelo de um determinado microcontrolador.

Quais são as principais vantagens de um microprocessador em relação a um microcontrolador?

a) O microprocessador geralmente apresenta um custo menor que um microcontrolador.

b) O microprocessador pode interpretar qualquer linguagem de programação, diferentemente do microcontrolador que só interpreta linguagem *assembly*.

c) O microprocessador, por não embarcar periféricos e memória interna, consegue atingir velocidades de processamento superiores a um microcontrolador.

d) O microprocessador pode ser utilizado em qualquer aplicação, já o microcontrolador somente em aplicações de controle, como o próprio nome sugere.

e) O microprocessador dispõe de um grau tecnológico superior ao microcontrolador.

2. Podemos classificar a arquitetura de um microcontrolador quanto a seu barramento dentre dois tipos distintos: a arquitetura Von-Neumann e a arquitetura Harvard. Na arquitetura Von-Neumann, a busca de dados e instruções não pode ser executada ao mesmo tempo. Já na arquitetura Harvard, os dados e instruções podem ser acessados simultaneamente, o que torna essa arquitetura inerentemente mais rápida que a Von-Neumann.

Uma outra classificação quanto à arquitetura de um microcontrolador diz respeito ao seu conjunto de instruções, que pode ser: a) **CISC**: computador de conjunto de instruções complexas (*Complex Instructions Set Computer*) e b) **RISC**: computador de conjunto de instruções reduzidas (*Reduced Instructions Set Computer*).

A respeito dos tipos de arquitetura de barramento e instruções de um microcontrolador, considere as seguintes afirmações:

( ) A arquitetura CISC tende a ser mais veloz que a arquitetura RISC, pois suas instruções são processadas com poucos ciclos de *clock*.

( ) O microcontrolador ATMega328 apresenta arquitetura de barramentos Harvard.

( ) Todo processador com arquitetura de barramento Von-Neumann também utiliza arquitetura de instruções CISC.

( ) A arquitetura CISC leva o microcontrolador a utilizar menos espaço de memória de programa do que aqueles com arquitetura RISC.

Considerando a sequência, qual das alternativas abaixo retrata corretamente as afirmações que são falsas e as que são verdadeiras?

a) V, V, F, V

b) F, V, F, V

c) F, V, F, F

d) V, F, F, V

e) V, F, V, F

3. O microcontrolador ATMega328, fabricado pela empresa ATMEL, faz parte de uma família de controladores chamada AVR, possuindo 8 bits como tamanho de palavra de processamento, tecnologia CMOS e construído sobre a arquitetura AVR RISC Harvard modificada. Ele foi projetado para atender a um alto desempenho de processamento, podendo alcançar até 20 MIPS (milhões de instruções por segundo), operando em 20MHz. Sua memória Flash para o armazenamento de instruções é de 32Kbytes, e

sua memória RAM volátil possui 2KBytes, possuindo ainda uma memória EEPROM de 1KByte para o armazenamento de dados não voláteis.

Considere uma aplicação em que se deseja utilizar o microcontrolador ATMega328 para o controle de temperatura de uma estufa climatizada, através de sensores de temperatura e umidade, além de ventiladores para garantir a circulação do ar. Esse sistema deveria prever também uma interface para comunicação com um computador de controle.

Pelas características e periféricos de um microcontrolador ATMega328, este seria adequado para embarcar essa aplicação?

a) Sim, pois ele possui sensor interno de temperatura e humidade, e periféricos para comunicação serial.

b) Não, pois o microcontrolador não possui memória suficiente para armazenar um software para tamanha complexidade de projeto.

c) Não, pois para esta aplicação, o mais correto seria a adoção de um microprocessador, com periféricos externos para leitura de sensores e atuação da ventoinha.

d) Sim, pois o microcontrolador possui entradas analógicas que poderiam ser utilizadas para leitura dos sensores, além de canais digitais de saída e PWM para controle das ventoinhas.

e) Não, pois o microcontrolador não possui canais dedicados para comunicação com um computador.

Seção 1.2

Periféricos básicos

Diálogo aberto

Caro aluno, nessa seção iremos iniciar nosso estudo a respeito dos componentes que definem e diferenciam os microcontroladores de outras tecnologias de processamento: os periféricos. Podemos associar os periféricos a circuitos internos do nosso microcontrolador, cuja função é executar atividades muito específicas que não poderiam ser desempenhadas pelo núcleo de processamento ou que exigiriam uma alta carga de trabalho para ser realizado. Se compararmos com um computador pessoal ou um notebook, os periféricos seriam todos aqueles componentes extras que são conectados à placa mãe e que conferem as mais diversas funcionalidades ao equipamento, como as placas de vídeo, de som, *ethernet*, *wi-fi*, leitor de mídias, etc. Sabemos que uma arquitetura de computador se diferencia de outra pela capacidade desses periféricos além, obviamente, do próprio poder de processamento. Podemos dizer o mesmo dos microcontroladores, que são caracterizados pela capacidade e versatilidade dos periféricos, ainda que estes sejam comuns. Nesta seção vamos focar nos periféricos mais comuns, que são encontrados na maioria dos microcontroladores e, além de entendermos seu funcionamento, com base na arquitetura do ATMega328, vamos estudar como configurá-los para cada propósito.

Nesta unidade você assumirá a posição de um recém-formado cientista da computação, contratado para auxiliar uma grande empresa do ramo de automação residencial a desenvolver o seu mais novo produto. Trata-se de um sistema de acionamento automático de atuadores, como janelas, luminárias e sistemas multimídia que, através da leitura de entradas discretas (botões e interruptores) e analógicas (sensores de temperatura e umidade) deverá realizar o processamento e controle do ambiente.

Nesta segunda etapa do projeto, o time de hardware lhe enviou o diagrama elétrico do primeiro protótipo do produto. Neste

diagrama está prevista a utilização de quatro entradas digitais a serem conectadas a interruptores que serão distribuídos pela instalação da residência, e duas saídas a relê para o acionamento de persianas motorizadas, que deverão ser acionadas precisamente 100 ms após o acionamento de qualquer entrada. Um *led* também foi previsto nesta versão do projeto para indicar que o sistema está em pleno funcionamento, e ele deverá ser acionado em uma frequência de 2Hz. O microcontrolador deve operar com um *clock* de 16MHz. Quais periféricos deveriam ser utilizados para atender a esse requisito de hardware? Uma vez escolhidos os periféricos, qual seria a configuração mais adequada aos mesmos?

O estudo que realizaremos sobre os periféricos básicos do microcontrolador ATMega328 será fundamental para guiá-lo à melhor configuração para atender aos requisitos passados pelo time. Veremos como configurar os pinos de I/O para o controle das entradas e saídas digitais, bem como configurar o temporizador para a contagem precisa de intervalos de tempo.

Dedique-se ao conteúdo dessa seção e descubra toda a capacidade que está embarcado em um microcontrolador.

Não pode faltar

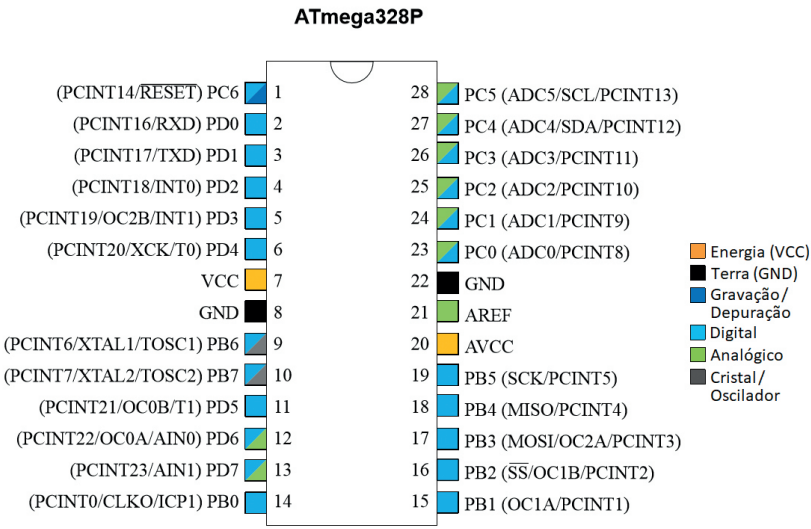
Entradas e saídas digitais

Você já ouviu falar do termo I/O? Esse termo representa entradas e saídas digitais, e está presente em todos os processadores e microcontroladores. O nosso estudo sobre os periféricos dos microcontroladores iniciará por eles, que são os mais básicos e comuns dentre todos os modelos. Esses periféricos dispõem de acesso direto aos pinos do microcontrolador, possibilitando assim a realização do controle digital individual de cada um, podendo configurá-lo como entrada para leitura de chaves e sensores ou saída para acionamento de *leds*, relês, etc.

Em relação ao microcontrolador ATMega328, os pinos de I/O são agrupados em três conjuntos chamados "PORTS", e cada qual representa um total de até oito pinos. Cada PORT é identificado com uma letra (B, C e D) assim, iremos nos referir a eles como PORTB, PORTC e PORTD, respectivamente. Da mesma maneira, os pinos dentro de um determinado PORT recebem um número sequencial,

que se inicia com o valor "0", e iremos nos referir a esse pino a com letra de seu PORT, seguida de seu número, por exemplo: PD4 é o pino 4 do PORTD. Na figura a seguir temos a relação de todos os pinos de I/O do microcontrolador ATMega328.

Figura 1.6 | Pinos de I/O do microcontrolador ATMega328



Fonte: <http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso em: 28 mar. 2018.

Internamente, cada um dos pinos de I/O apresenta um circuito digital responsável por configurar seu modo de operação, ou seja, atuar como saída digital e controlar o pino entre os níveis de saída lógico zero (GND ou 0V) e um (VCC), ou atuar como entrada digital de alta impedância e realizar leitura dos mesmos níveis lógicos.

Para o controle e configuração de seus pinos, cada PORT implementa três registradores distintos. Cada registrador é do tamanho de 8 bits (1 byte), sendo que cada bit possui influência direta sobre o pino da mesma ordem, ou seja, o bit 0 atua sobre o pino 0, o bit 1 sobre o pino 1 e assim sucessivamente até o bit 7. Segue a descrição de cada registrador:

- **DDRB, DDRC e DDRD**: responsáveis por configurar a direção dos pinos em seu respectivo PORT. Ao configurar um bit desse registrador como 0, o respectivo pino atuará como

entrada, e caso configurado como 1, o pino atuará como saída. Exemplo: em DDRD = 0b11110000 (em binário), os pinos PD0 a PD3 foram configurados como entrada, e os pinos PD4 a PD7 foram configurados como saída.

- **PINB, PINC e PIND**: responsáveis por realizar a leitura digital dos pinos, quando os mesmos são configurados como entrada pelo registrador DDR. Cada bit desse registrador representará o valor lógico do pino externo respectivo. Exemplo: PINB em 1000001 (binário), os pinos PB0 e PB7 apresentam o nível lógico 1 (VCC) em seus terminais, e os demais pinos, o nível lógico 0 (GND) em seus terminais.
- **PORTB, PORTC e PORTD**: responsáveis por realizar o acionamento digital dos pinos, quando os mesmos são configurados como saída pelo registrador DDR. Cada bit desse registrador será responsável por gerar um nível lógico no respectivo pino. Exemplo: em PORTC = 0b11100000, os pinos PC0 a PC4 estarão com seus terminais em nível lógico 0 (GND), e os pinos PC5 a PC7 estarão em nível lógico 1 (VCC).

Pesquise mais

Conheça mais sobre como manipular as portas de entrada e saída digitais do microcontrolador ATMega328 através de exemplos práticos em linguagem de programação C com o texto Entrada e Saída – Manipulando Registradores. PICORETI, Rodolfo. Entrada e Saída – Manipulando Registradores. **Vida de silício**. Maio 2017. Disponível em: <https://portal.vidadesilicio.com.br/entrada-e-saida-manipulando-registradores>. Acesso em: 28 mar. 2018.

Interrupções

Você saberia dizer como um computador reconhece que uma tecla do teclado foi pressionada, sem ter que ficar a todo momento realizando a leitura desse periférico? Seria muito ineficiente realizar a leitura constante, sabendo que o usuário passa boa parte do tempo sem digitar; então, como isso é resolvido?

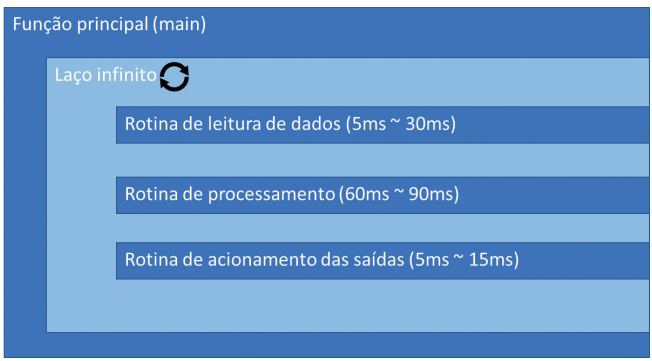
Em um projeto clássico de software em sistemas microcontrolados, é implementado um laço infinito dentro da função principal, e dentro desse laço são invocadas as rotinas de leitura de entradas, de processamento dos dados e ativação das

saídas. Como tais rotinas dependem de inúmeras variáveis para sua execução, não é possível calcular com exatidão, na maioria dos casos, o tempo que cada uma levará para ser completamente executada, e, consequentemente, não é possível ter um valor invariável de tempo para conclusão de um ciclo completo. A Figura 1.7 exemplifica um modelo de software com três rotinas básicas, e um intervalo aproximado de tempo de execução de cada uma.

Assimile

Diferentemente dos microprocessadores, os microcontroladores foram projetados para o trabalho em sistemas de tempo real, ou seja, sistemas que possuem requisitos muito rígidos de tempo de resposta.

Figura 1.7 | Modelo de software microcontrolado



Fonte: elaborada pelo autor.

Esse fator indica que sempre existirá uma latência variável para a resposta do sistema a qualquer que seja o estímulo de entrada e, em alguns casos, essa incerteza ou a demora de resposta não são aceitáveis.

Para contornar esse problema, os microcontroladores implementam em suas arquiteturas as chamadas "interrupções" que, vinculadas a determinados estímulos de software ou hardware em um microcontrolador, fazem com que o programa seja imediatamente desviado para rotinas específicas que tratarão desse evento. Essa característica permite que o microcontrolador seja mais responsivo a eventos especiais, e que seja liberado o tempo

de processamento que era utilizado para realizar a leitura contínua desses eventos.

De maneira global, todos os periféricos do microcontrolador ATMega328 implementam algum nível de interrupção; seguem alguns exemplos:

- Temporizadores: interrupções por término na contagem de intervalo de tempo.
- Conversor analógico / digital: interrupção ao término do processo de conversão.
- Portas de I/O: interrupção por mudança no sinal lógico em um dos pinos externos.

Nosso foco, neste momento, será voltado às interrupções vinculadas aos pinos de I/O, também chamadas de Interrupções Externas.

Interrupções externas

Todos os pinos de I/O do microcontrolador ATMega328 podem ser utilizados como fonte para interrupções externas provenientes de alterações no nível lógico de entrada, porém dois pinos em especial apresentam configurações mais refinadas, o que lhes permite gerar eventos de interrupção em várias situações de alteração lógica. São os pinos PD2 (INT0) e o pino PD3 (INT1).

Para configurarmos os pinos PD2 e PD3 como fonte de interrupção, primeiramente devemos configurá-los como pinos de entrada digital, através dos registradores DDRD, ou seja, fazendo DDRD = 0b00000000.

O próximo passo diz respeito ao tipo de evento lógico que ocorrerá nesses pinos para que uma interrupção seja gerada. Para o ATMega328, os seguintes eventos podem ser fontes de interrupção: nível lógico 0 no pino, mudança de nível 0 para nível 1 (borda de subida), mudança de nível 1 para nível 0 (borda de descida) e qualquer mudança de nível. Para realizar essa configuração é necessário alterar o registrador EICRA, no qual a combinação dos bits ISCx0 e ISCx1 definirá o tipo de evento que deve ocorrer para que o pino gere uma interrupção, e "x" pode assumir o valor de 0 para INT0 ou 1 para a INT1.

Figura 1.8 | Registrador EICRA

	7	6	5	4	3	2	1	0
EICRA	-	-	-	-	ISC11	ISC10	ISC01	ISC00

Fonte: elaborada pelo autor.

Seguem as combinações dos bits ISCx0 e ISCx1 para cada evento de interrupção.

Figura 1.9 | Bits de configuração de interrupção para INT0 e INT1

ISC11	ISC10	Descrição
0	0	Um nível baixo em INT1 gera um pedido de interrupção.
0	1	Qualquer mudança lógica em INT1 gera um pedido de interrupção.
1	0	Uma borda de descida em INT1 gera um pedido de interrupção.
1	1	Uma borda de subida em INT1 gera um pedido de interrupção.
ISC01	**ISC00**	**Descrição**
0	0	Um nível baixo em INT0 gera um pedido de interrupção.
0	1	Qualquer mudança lógica em INT0 gera um pedido de interrupção.
1	0	Uma borda de descida em INT0 gera um pedido de interrupção.
1	1	Uma borda de subida em INT0 gera um pedido de interrupção.

Fonte: Lima e Villaça (2012, p. 164).

Todas as interrupções no microcontrolador ATMega328 são mascaráveis, ou seja, podem ser habilitadas e desabilitadas individualmente através de registradores específicos. No caso dos pinos de interrupção INT0 e INT1, esse mascaramento é feito pelo registrado EIMSK, em que o bit INT0 é usado para ativar a interrupção de mesmo nome e, semelhantemente, o bit INT1.

Figura 1.10 | Registrador EIMSK

	7	6	5	4	3	2	1	0
TIMSKx	-	-	-	-	-	-	-	TOIEx

Fonte: elaborada pelo autor.

Por fim, para que todas as interrupções estejam ativas, é preciso acionar o bit I presente no registrador SREG do microcontrolador.

Esse bit, também chamado de bit de Interrupção Global, atua como uma chave geral de todas as interrupções, e deve ser ativado para que as fontes de interrupção de todos os periféricos passem a atuar.

Temporizadores

Vimos que os tempos de processamento de um *software* no microcontrolador podem ser variados e incertos. O que então pode ser feito quando é necessário realizar a contagem precisa de tempo, ou captura de eventos periódicos? A resposta está nos Temporizadores.

Apesar de os microcontroladores serem projetados para realizar o controle de sistemas no menor tempo de resposta possível, em alguns casos é necessário que sejam gerados atrasos propositais para o acionamento de saídas ou para a leitura sincronizada de entradas.

Uma das formas de se gerar um intervalo de atraso é programar o processador para executar instruções vazias (instrução NOP em *assembly*) em um determinado número de vezes que represente o atraso desejado, tendo em vista a frequência do *clock* do microcontrolador.

Exemplificando

Considere uma aplicação em que se deseja aguardar 100 ms antes de ligar uma saída do microcontrolador. Sabendo que o microcontrolador opera com um *clock* de 16MHz, e que uma instrução NOP é processada em um ciclo de *clock*, quantas vezes essa instrução deveria ser executada para atingirmos o intervalo desejado?

Com uma frequência de 16MHz, temos um tempo de ciclo $Tc = \frac{1}{F} = \frac{1}{16 \times 10^6} = 62,5\ ns$, logo, para atingirmos 100 ms, seriam necessários $\frac{100\ ms}{62,5\ ns} = \frac{100 \times 10^{-3}}{62,5 \times 10^{-9}} = 1.600.000$ ciclos.

Apesar de ser uma solução válida, utilizar o processador para realizar uma aplicação tão simples quanto a de aguardar alguns milissegundos, demanda milhares de instruções vazias, ou seja, instruções que não executam função útil alguma ao processamento.

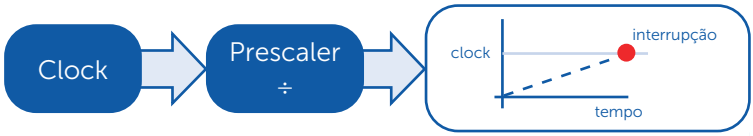
Para contornar esse problema, os microcontroladores implementam em hardware periféricos temporizadores que apresentam, entre outras funções, a capacidade de realizar a contagem de intervalos de tempo de forma paralela ao processamento do programa e, ao final da contagem, informar o processador através de um evento de interrupção.

O periférico temporizador no microcontrolador ATMega328 opera com base no *clock* do sistema, ou seja, ele utiliza a frequência de entrada para calcular a quantidade de ciclos necessários para a temporização de um determinado intervalo de tempo. Esse microcontrolador dispõe de três periféricos distintos: TC0, TC1 e TC2, sendo que o primeiro e o último são temporizadores de 8 bits, e o segundo, um temporizador de 16 bits. A quantidade de bits de um temporizador determina quantos ciclos de *clock* ele pode contar para gerar um evento de interrupção, por exemplo, o TC0 (Temporizador 0), por ser de 8 bits, consegue contar até $2^8 = 256$ ciclos de *clock*.

Se considerarmos como exemplo um microcontrolador com frequência de *clock* de 16MHz, teríamos um tempo de ciclo de 62,5ns e um tempo máximo de contagem de $2^8 \times 62,5ns = 16us$ para TC0 e TC2 e $2^{16} \times 62,5ns = 4,096ms$ para TC1. Para situações reais, esses valores podem ser considerados muito baixos, por isso cada um desses periféricos implementa um bloco *Prescaler*, responsável por reduzir o *clock* através da sua divisão por um fator configurado que pode ser definido como [1,8,64,256,1024] para TC0 e TC1 ou [1,8,32,64,128,254,1024] para TC2.

A partir do momento que o *Prescaler* é configurado, o temporizador inicia sua contagem instantaneamente e, ao atingir o seu valor máximo (256 para TC0 e TC2, e 65536 para TC1), ocorre um evento denominado de "estouro", no qual o valor do temporizador retorna a zero e ele pode gerar um evento de interrupção, informando ao processador que o tempo de contagem foi alcançado.

Figura 1.11 | Funcionamento de um temporizador no microcontrolador ATMega328



Fonte: elaborada pelo autor.

O registrador responsável por armazenar a contagem atual dos temporizadores é o TCNTx (em que x equivale ao temporizador em questão: 0, 1 ou 2). Como o evento de estouro sempre ocorre quanto é atingido o valor máximo do temporizador, é possível manipular esse registrador de forma a inicializá-lo com um valor diferente de zero para obter valores precisos de contagem, técnica essa conhecida como "carregamento". Desta forma, para se calcular o valor exato de contagem de um determinado temporizador, basta utilizar a seguinte equação: $T_{contagem} = \frac{(\text{MAX-Carga}) \times \text{Prescaler}}{F_{clock}}$

Exemplificando

Deseja-se gerar um evento de interrupção pela contagem do intervalo de 100 ms utilizando o microcontrolador ATMega328 com a frequência de clock de 16 MHz. Qual deveria ser o temporizador utilizado e suas configurações para essa aplicação?

Por ser um valor de intervalo relativamente alto, o mais adequado é utilizar o Temporizador 1 (TC1) para esta aplicação, por ser um temporizador de 16 bits. Sabemos que, se utilizarmos o *Prescaler* com o valor 1, teremos uma contagem máxima de 4,096 ms, logo, para atingir o valor de 100 ms teremos que utilizar o valor de *Prescaler* = 64 para termos um intervalo máximo de ~263 ms. Podemos, agora, modificar a equação do tempo de contagem para descobrir o valor de carga inicial em TCNT1:

$$\text{Carga} = MAX - \frac{T_{contagem} \times F_{clock}}{\text{Prescaler}}$$

$$\text{Carga} = 65.536 - \frac{100 \times 10^{-3} \times 16 \times 10^{6}}{64} \rightarrow \text{Carga} = 40.536$$

Reflita

Para diversos problemas de configuração de um temporizador é possível selecionar vários valores válidos para o *Prescaler*, porém a recomendação é que se escolha sempre aquele valor que indique um tempo máximo de contagem mais próximo ao valor de contagem desejado. Qual seria a justificativa para essa escolha?

Os bits **CSx0**, **CSx1** e **CSx2** são os responsáveis pela configuração do valor de *Prescaler* dos temporizadores, e estão presentes no registrador **TCCRxB** (em que x equivale ao número do temporizador: 0, 1 ou 2).

Figura 1.12 | Registrador TCCRxB

	7	6	5	4	3	2	1	0
TCCRxB	-	-	-	-	-	CSx0	CSx0	CSx0

Fonte: elaborada pelo autor.

Tabela 1.2 | Configuração de *Prescaler* dos temporizadores

CS02	CS01	CS00	Prescaler
0	0	0	TC Parado
0	0	1	1
0	1	0	8
0	1	1	64
1	0	0	256
1	0	1	1024

CS12	CS11	CS10	Prescaler
0	0	0	TC Parado
0	0	1	1
0	1	0	8
0	1	1	64
1	0	0	256
1	0	1	1024

CS22	CS21	CS20	Prescaler
0	0	0	TC Parado
0	0	1	1
0	1	0	8
0	1	1	32
1	0	0	64
1	0	1	128
1	1	0	256
1	1	1	1024

Fonte: elaborada pelo autor.

Por fim, para habilitarmos o evento de interrupção ao fim do ciclo de contagem de tempo, basta habilitarmos o bit TOIEx do registrador TIMSKx (em que x equivale ao número do temporizador: 0, 1 ou 2).

Figura 1.13 | Registrador TIMSKx

	7	6	5	4	3	2	1	0
TIMSKx	-	-	-	-	-	-	-	TOIEx

Fonte: elaborada pelo autor.

Sem medo de errar

Lembre-se de que você, contratado por uma empresa de automação residencial, está auxiliando a equipe de projeto, e nessa etapa o time de hardware lhe enviou um diagrama elétrico prevendo quatro entradas digitais que serão conectadas a interruptores da residência e duas saídas digitais, a serem ligadas em relês para o acionamento de persianas motorizadas, que deverão ser acionadas precisamente 100 ms após o acionamento da ligação de qualquer uma das entradas. Além disso, o time também previu um led indicador, que deverá piscar em uma frequência de 2Hz, para indicar o correto funcionamento do sistema. O microcontrolador deve operar com um clock de 16MHz.

Foi solicitado a você que indicasse quais periféricos do microcontrolador ATMega328 poderiam ser utilizados para atender aos requisitos listados, bem como qual a melhor configuração para eles.

Entradas e saídas digitais

Para este problema, vamos iniciar pelas entradas e saídas digitais. O requisito solicita que sejam disponibilizadas quatro entradas digitais e duas saídas, e para isso podemos considerar a utilização de qualquer uma das portas de I/O disponíveis, como o caso da **PORTD.** Vamos inicialmente definir quais pinos seriam conectados a cada entrada ou saída do produto.

Tabela 1.3 | Configuração de *Prescaler* dos temporizadores

PINO	TIPO	DESCRIÇÃO
PD0	ENTRADA	ENTRADA 1
PD1	ENTRADA	ENTRADA 2
PD2	ENTRADA	ENTRADA 3
PD3	ENTRADA	ENTRADA 4
PD4	NÃO UTILIZADO	-
PD5	NÃO UTILIZADO	-
PD6	SAÍDA	SAÍDA 1
PD7	SAÍDA	SAÍDA 2

Fonte: elaborada pelo autor.

Com base nessa associação, agora podemos definir a configuração do registrador DDRD, responsável por definir a direção de cada pino de I/O do PORTD, sabendo que se o bit for 0 o pino atuará como entrada, e caso seja 1, o pino atuará como saída.

Figura 1.14 | Configuração do registrador DDRD para a aplicação

	7	6	5	4	3	2	1	0
DDRD	1	1	0	0	0	0	0	0

Fonte: elaborada pelo autor.

Como foi escolhido o **PORTD** para conexão com as entradas e saídas, podemos agora utilizar o registrador **PIND** para a leituras do pino PD0 a PD3, e o registrador **PORTD** para o acionamento das saídas PD6 e PD7.

Temporizadores

Para este problema, temos dois casos em que é necessária a utilização de temporizadores: o atraso de 100 ms para o acionamento das saídas e o acionamento do *led* em uma frequência de 2Hz (250 ms com *led* ligado e 250 ms com *led* apagado).

Vamos iniciar pelo temporizador do *led*, que precisaria gerar um evento de interrupção por término de contagem a cada 250 ms, a assim ligar o *led* se ele estiver apagado, ou desligá-lo se ele já estiver ligado. Como este intervalo de tempo é maior do que o outro, vamos optar por utilizar o temporizador 1 (TC1) de 16 bits. Para configurá-lo, precisamos descobrir qual o valor de *Prescaler* mais adequado e qual o valor de carga inicial.

Se o *Prescaler* utilizado fosse 1 e não utilizássemos um valor inicial de carta, teríamos um tempo máximo de contagem de: $T_{contagem} = \frac{65.536 \times 1}{16 \times 10^6} \rightarrow T_{contagem} = 4,096\ ms$, ou seja, aproximadamente 62 vezes menor do que o necessitado. Analisando a tabela de *Prescaler* do TC1, temos o valor mais próximo de **64**, que seria o mais correto para o problema. Para o cálculo do valor inicial de carga temos: $\text{Carga} = 65.536 - \frac{250 \times 10^{-3} \times 16 \times 10^6}{64} \rightarrow \text{Carga} = 3.036$.

Para o atraso de 100 ms podemos escolher entre os temporizadores restantes: TC0 ou TC2. Ambos são de 8 bits, o que

nos confere um valor máximo de contagem, utilizando o maior *Prescaler* de 1.024, de $T_{contagem} = \frac{256 \times 1.024}{16 \times 10^6} \rightarrow T_{contagem} = 16,384\ ms$, o que não seria suficiente para atingir o valor de 100 ms e, como já estamos utilizando o TC1 para outra aplicação, não poderemos utilizá-lo. Uma solução para o problema seria realizar a contagem de 10 ms e, para cada evento de interrupção, fazermos uma contagem auxiliar até que o valor atinja 10, contabilizando assim a soma de 100 ms. Desta forma, utilizando o valor de *Prescaler* de **1.024**, vamos calcular o valor de carga para 10 ms: $\text{Carga} = 256 - \frac{10 \times 10^{-3} \times 16 \times 10^6}{1.024} \rightarrow \text{Carga} = 100$

Por fim, basta habilitarmos os bits **TOIEx** nos registradores **TIMSKx** dos temporizadores para habilitarmos os eventos de interrupção para os mesmos.

Avançando na prática

Novo temporizador e interrupção

Descrição da situação-problema

Após a entrega do seu relatório com os periféricos e suas configurações para o atendimento dos requisitos do produto, a equipe de projetos lhe solicitou que configurasse mais um temporizador para gerar eventos de interrupção a cada 5 ms para a leitura das entradas analógicas do sistema. Também foi solicitado a você que incluísse uma interrupção externa que capture uma borda de subida do sinal para ser conectado a um botão de emergência que estará presente na instalação. Utilize o temporizador TC2 para essa aplicação e a interrupção externa INT0, e considere que a frequência de clock é de 16 MHz.

Resolução da situação-problema

Temporizador TC2

Sabemos que TC2 é um temporizador de 8 bits que realiza contagens até 256. O seu tempo máximo de contagem, considerando o maior *Prescaler* de **1024**, é de $T_{contagem} = \frac{256 \times 1024}{16 \times 10^6} \rightarrow T_{contagem} = 16,384 ms$, logo,

podemos considerar esse valor para sua configuração. Agora, para o valor de carga do temporizador para contagem de 5 ms, podemos utilizar o seguinte cálculo: $\text{Carga} = 256 - \frac{5\times10^{-3}\times16\times10^{6}}{1.024} \rightarrow \text{Carga} = 178$. Para ativar o evento de interrupção, basta ativarmos o bit **TOIE2** de seu registrador **TIMSK2**.

Interrupção externa

Foi solicitado que a interrupção INT0 fosse habilitada como borda de subida para captura do evento. Inicialmente, precisamos definir que o pino **PD2**, que é multiplexado com INT0, seja configurado como entrada, e fazemos isso através do registrador **DDRD** = 0b00000000. O próximo passo é configurar INT0 como borda de subida, e isso é feito através da configuração dos bits **ISC10** e **ISC01** do registrador EICRA com os valores 1 e 1 respectivamente. O próximo passo é ativar a máscara da interrupção através do bit **INT0** do registrador **EIMSK**. Por fim, basta ativar o bit de interrupção global I do registrador SREG para que a interrupção passe a ser ativa.

Faça valer a pena

1. Para o controle e configuração de seus pinos, cada PORT do microcontrolador ATMega328 implementa três registradores distintos. Cada registrador tem o tamanho de 8 bits (1 byte), sendo que cada bit exerce influência direta sobre o pino da mesma ordem, ou seja, o bit 0 atua sobre o pino 0, o bit 1 sobre o pino 1 e assim sucessivamente até o bit 7.

Dado o valor do registrador DDRB em sequência binária 11110000, é correto afirmar que a configuração dos pinos PB0 a PB7 é:

a) Todos os pinos estão configurados como entrada.

b) Os pinos pares estão configurados como entrada e os ímpares como saída.

c) Os pinos PB0 a PB3 estão configurados como entrada e os demais como saída.

d) Os pinos PB0 a PB3 estão configurados como saída e os demais como entrada.

e) Todos os pinos estão configurados como saída.

2. Todos os pinos de I/O do microcontrolador ATMega328 podem ser utilizados como fonte para interrupções externas provenientes de alterações no nível lógico de entrada, porém dois pinos em especial dispõem de configurações mais refinadas, o que lhes permite gerar eventos de interrupção em várias situações de alteração lógica. São os pinos PD2 (INT0) e o pino PD3 INT1).

Deseja-se conectar um sinal digital à interrupção INT1 para que sejam capturados e contabilizados todos os eventos de alteração em seu nível lógico. Qual deve ser a configuração mais apropriada para os bits ISC10, ISC11 e INT1 nesta função?

a) ISC10 = 1, ISC11 = 1 e INT1 = 1.

b) ISC10 = 1, ISC11 = 0 e INT1 = 0.

c) ISC10 = 0, ISC11 = 0 e INT1 = 1.

d) ISC10 = 0, ISC11 = 1 e INT1 = 0.

e) ISC10 = 1, ISC11 = 0 e INT1 = 1.

3. Para o problema de latência na geração de atrasos via software, os microcontroladores implementam em hardware periféricos temporizadores que apresentam, entre outras funções, a capacidade de realizar a contagem de intervalos de tempos de forma paralela ao processamento do programa e, ao final da contagem, informar o processador através de um evento de interrupção.

Qual o valor correto de *Prescaler* e carga para que o temporizador TC1 seja configurado para gerar um evento de interrupção a cada 25 ms?

a) *Prescaler* = 1, Carga = 50.000.

b) *Prescaler* = 8, Carga = 15.536.

c) *Prescaler* = 64, Carga = 50.000.

d) *Prescaler* = 8, Carga = 50.000.

e) *Prescaler* = 1, Carga = 15.536.

Seção 1.3

Periféricos avançados

Diálogo aberto

Caro aluno, nessa seção continuaremos nosso estudo a respeito dos periféricos dos microcontroladores, e dessa vez entenderemos como alguns dos principais periféricos avançados funcionam e como podem ser configurados para desempenhar funções específicas que tornam os projetos microcontrolados mais versáteis e capazes de interagir com outros sistemas e com o próprio usuário. Na sua vida profissional, você se deparará com inúmeros equipamentos que necessitarão ser acionados ou monitorados de maneira analógica, como sensores e motores, por exemplo. Veremos nessa seção como um dispositivo de processamento puramente digital, como um microcontrolador, pode lidar com esses equipamentos.

Inicialmente avançaremos nossos estudos com o periférico temporizador a fim de compreender como utilizá-lo para geração de sinais digitais chamados de PWM (*Pulse Width Modulation* ou Modulação por Largura de Pulso), depois entenderemos como os microcontroladores conseguem ler e processar sinais analógicos por meio dos chamados conversores analógicos digitais. De forma a desenvolver protocolos de comunicação entre o sistema microcontrolado e outros dispositivos externos, veremos como funciona o periférico USART (*Universal Synchronous Asynchronous Receiver Transmitter* ou Receptor e Transmissor Universal Síncrono e Assíncrono), responsável por prover um mecanismo simples para implementação de protocolos seriais. E, por fim, veremos como armazenar dados que não podem ser perdidos com a desenergização do sistema, com as memórias EEPROM (*Electrically-Erasable Programmable Read-Only Memory* ou Memória de Somente Leitura Programável e Apagável Eletricamente).

Não se esqueça que você recentemente ingressou em um time de talentosos especialistas de uma empresa de automação residencial, e já em seu primeiro projeto seus conhecimentos sobre sistemas microcontrolados estão sendo colocados à prova em decisões que

devem ser as mais assertivas possíveis para o futuro do projeto. Após ter validado com sucesso a primeira etapa juntamente com uma empresa parceira que tem interesse em adquirir o produto, o time de marketing fez um levantamento de mercado sobre quais funcionalidades extras o público-alvo mais se atentaria, sendo elas: a) Controle de iluminação do ambiente pela dimerização controlada das lâmpadas; b) Abertura automática das persianas de acordo com a leitura das temperaturas ambiente e externa da residência; c) Terminal de comunicação serial para entusiastas que desejam acessar e controlar a automação de sua residência por comandos customizáveis; e d) Possibilidade de salvar programas pessoais com configurações predefinidas dos atuadores da casa.

Como base nessas novas premissas, e com o conhecimento dos periféricos avançados do microcontrolador ATMega328, o chefe da engenharia lhe solicitou um relatório técnico em que você deve descrever como utilizar tais periféricos, individualmente ou em conjunto, para atender esse novo escopo do produto. Quais dos periféricos podemos utilizar para atender essa demanda? Poderíamos ainda utilizar alguns periféricos básicos?

O conteúdo dessa seção lhe proporcionará todo o conhecimento necessário sobre os principais periféricos avançados de um microcontrolador para atender à demanda dessa nova etapa do projeto. Empenhe-se nos estudos dessa seção e aprimore o seu domínio sobre a tecnologia de microcontroladores! Bons estudos!

Não pode faltar

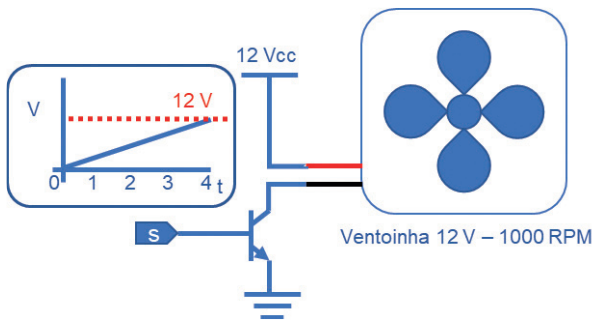
WM (*Pulse Width Modulation* ou Modulação por Largura de Pulso)

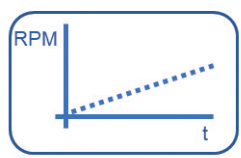
Você sabia que um microcontrolador é capaz de utilizar suas saídas digitais para realizar o controle analógico de outros dispositivos? Isso é possível com a configuração de seu pino digital como um pino PWM que, pelo auxílio de um temporizador, é capaz de gerar formas de ondas digitais de tal maneira que a tensão média na saída do pino possa variar entre o valor mínimo (0V) e o valor máximo (Vcc).

Em muitas aplicações reais em projetos microcontrolados é necessário o controle de dispositivos de maneira mais versátil

do que simplesmente realizar o ligamento e desligamento, necessitando, assim, gerar condições intermediárias entre estes dois estados. Considere a aplicação das ventoinhas utilizadas para o resfriamento de processadores: quanto maior a temperatura, maior é a velocidade de rotação. Sabe-se que a velocidade de rotação do eixo de um motor DC é diretamente proporcional à tensão aplicada ao mesmo, desde o motor parado em 0V até o motor em plena rotação na tensão máxima. Logo, para se controlar a velocidade, é necessário um sinal analógico de tensão que varie entre o valor mínimo e máximo (Figura 1.15).

Figura 1.15 | Controle de velocidade de uma ventoinha por variação de tensão





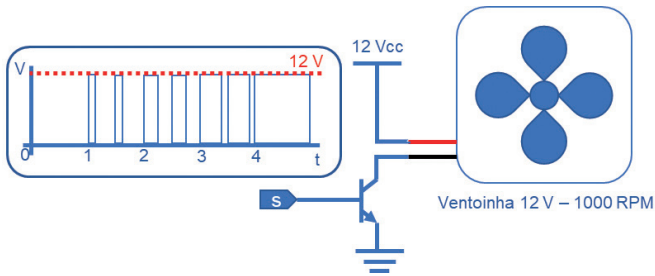
t	V	RPM
0	0	0
1	3	250
2	6	500
3	9	750
4	12	1000

Fonte: elaborada pelo autor.

Em um microcontrolador com saídas puramente digitais, não é possível conceber sinais analógicos para o controle desses dispositivos, porém se utilizarem tais pinos para a geração de ondas digitais que variam entre "zero" e "um" em alta frequência, o valor de tensão resultante nesse pino será um valor médio de tensão proporcional ao tempo que o pino permaneceu ligado (duty cycle) em relação ao tempo total do período dessa onda. Essa técnica é

chamada de PWM, ou Modulação por Largura de Pulso (LIMA, 2012). Veja na Figura 1.16 como um microcontrolador poderia realizar o controle de velocidade da ventoinha por meio um sinal digital com alta frequência e com duty cycle regulável.

Figura 1.16 | Controle de velocidade de uma ventoinha por variação de tensão



t	Duty Cycle	$V_{média}$	RPM
0	0%	0	0
1	25%	3	250
2	50%	6	500
3	75%	9	750
4	100%	12	1000

Fonte: elaborada pelo autor.

Como visto na seção anterior, o microcontrolador ATMega328 dispõe de 3 temporizadores distintos (TC0, TC1 e TC2), que são responsáveis pela geração de eventos em intervalos configuráveis. Outra atribuição desses temporizadores é geração de sinais PWM por intermédio de pinos dedicados do microcontrolador (**OCxA** e **OCxB**) e registradores específicos que têm por função controlar o valor de *duty cycle* do sinal (**OCRxA** e **OCRxB**), em que o valor "x" denota o número do temporizador.

Tomando como exemplo o temporizador **TC0**, para se gerar um sinal PWM no pino **OC0A**, deve-se seguir os seguintes passos:

1. Configurar os bits **WGM01** e **WGM00** como 1 no registrador **TCCR0A**, para que o temporizador passe a atuar como gerador PWM.
2. Configurar os bits **COM0A1** e **COM0A0** como 1 e 0, respectivamente. Dessa forma estamos conectando o pino **OC0A** ao temporizador.

3. Definir a frequência do sinal **PWM** pela equação: $F_{PWM} = \frac{F_{clock}}{512 \times N} \ [Hz]$, em que F_{PWM} representa a frequência do sinal PWM em Hz, F_{clock} representa a frequência de *clock* do microcontrolador e N representa o fator de *Prescaler* configurável (1,8,64,256 ou 1024).
4. Controlar o *duty cycle* do sinal pela equação: $DutyCycle = \frac{OCR0A}{255} \times 100 \ [\%]$, em que o valor de OCR0A pode variar entre 0 (0% de *duty cycle*) e 255 (100% de *duty cycle*).

Assimile

Assim como ocorre para a função de contagem de tempo, quando se deseja obter um sinal PWM mais preciso, o adequado é a utilização do periférico temporizador 1 (TC1), que por ser de 16 bits, proporcionará mais versatilidade para a escolha da frequência de operação e do ajuste do valor de *duty cycle*.

Conversores analógico-digitais

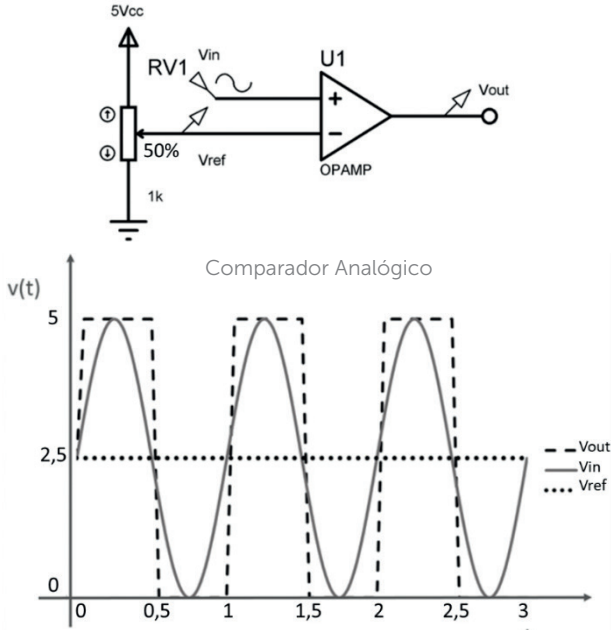
Sabemos que na natureza a grande maioria dos sinais se manifesta de maneira analógica e não digital, como temperatura, tensão elétrica, pressão, etc. Como um microcontrolador, sendo um dispositivo puramente digital, consegue realizar a leitura e interpretação desses sinais?

Diferentemente dos sinais digitais, que só assumem um dentre dois valores distintos (0 ou 1), um sinal analógico pode apresentar qualquer valor dentro de um determinado intervalo. Tal diferença física da natureza desses tipos de sinais caracteriza um desafio quando se pretende realizar a conversão de um modelo para outro. No caso dos microcontroladores, que são dispositivos digitais, é necessária a adoção de circuitos dedicados, chamados de conversores analógico-digitais, que realizam a discretização do sinal analógico de entrada para um valor inteiro que represente o nível desse sinal (LIMA, 2012).

Um exemplo clássico de circuito conversor analógico-digital é o comparador analógico (Figura 1.17) que, pela comparação de um sinal de entrada a um valor de referência, atribui um valor digital na sua saída, sendo o valor 0 atribuído quando o sinal de entrada é

inferior ao valor de referência, e o valor 1, quando o sinal é superior ao valor de referência.

Figura 1.17 | Conversor analógico-digital de 1 bit



Fonte: elaborada pelo autor.

Veja no exemplo da Figura 1.17 que é aplicado um sinal senoidal de amplitude de 5 V na entrada (Vin) e o valor de referência é de 2,5 V. Esse circuito também é chamado de conversos AD (Analógico-Digital) de 1 bit, pois os valores analógicos de entrada são discretizados e posteriormente representados por somente 1 bit, que assume o valor 0 para valores de tensão entre 0 V e 2,5 V, e o valor 1 para tensões entre 2,5 V e 5 V, sendo então o valor de 2,5 V designado como **resolução** deste conversor AD.

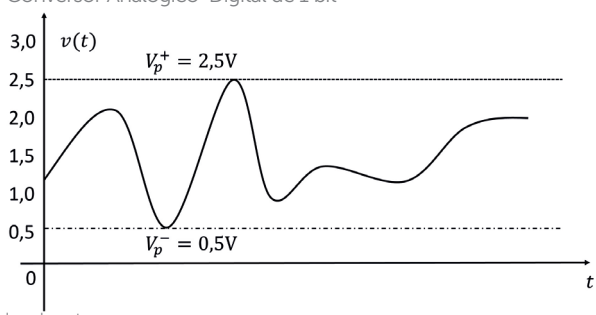
Os microcontroladores, como o ATMega328, dispõem de conversores AD implementados com vários bits de saída, sendo que quanto maior a quantidade de bits do conversor, menor será o valor de resolução do mesmo, tornando-o um conversor cada vez mais preciso com o aumento do número de bits. Um microcontrolador com conversor AD de 8 bits pode representar um sinal analógico como valores inteiros entre 0 e 255 ($2^8 - 1$), já um outro microcontrolador com um conversor AD de 16 bits pode representar um sinal analógico como valores entre 0 e 65535 ($2^{16} - 1$), o que o torna um conversor exponencialmente mais preciso que o seu concorrente.

Outro fator importante quando lidamos com conversores AD é o seu **valor de tensão de referência.** Este valor representa qual será o nível de tensão de fundo de escala do conversor, ou seja, aquele que representará o máximo valor na escala de conversão. O valor de resolução de um conversor sofre influência direta do valor de referência, uma vez que o seu valor dependerá dessa informação, bem como a quantidade de bits de um conversor. Veja a seguir a expressão para o cálculo da resolução de um conversor:

$$\text{Resolução} = \frac{V_{ref}}{2^N - 1} \quad [V / bit]$$

Em que V_{ref} é o valor de tensão de referência, e N representa a quantidade de bits do conversor AD. Podemos então inferir que, quanto menor o valor de V_{ref} menor será a resolução e mais preciso será o conversor. Porém, é preciso ter cuidado ao definir o valor de V_{ref} para que não seja tão baixo a ponto de não representar corretamente os valores mais altos do sinal analógico, e nem tão alto para que prejudique a resolução do conversor desnecessariamente. Veja o exemplo do sinal apresentado na Figura 1.18.

Figura 1.18 | Conversor Analógico-Digital de 1 bit



Fonte: elaborada pelo autor.

Analisando o sinal, é possível afirmar que o valor mais adequado para referência de um conversor AD será de 2,5V, que apresenta o máximo que o sinal pode atingir.

O microcontrolador ATMega328 apresenta seis pinos que podem ser utilizados para entrada de sinal analógico para conversão, e todos eles estão alocados no PORTC (PC0 a PC5), bastando configurá-los como entrada por meio do registrador DDRC para que se possa realizar a leitura dos valores.

O conversor AD desse microcontrolador tem uma resolução configurável de 8 ou 10 bits, podendo assim ter uma representação de até 1024 pontos. O valor de tensão de referência pode ser atribuído por um pino externo chamado VREF, ou utilizando a própria tensão de alimentação como referência. Esse periférico também tem um temporizador interno, que é utilizado para definir o tempo de captura do sinal analógico antes de uma nova conversão. Quão maior o tempo de captura, mais fidedigno é o valor do sinal amostrado, porém menor será a taxa de amostragem do sinal, e dessa forma é sempre importante que se estude a natureza do sinal para que o melhor tempo seja escolhido. Ao final da conversão, o valor resultante será armazenado no registrado **ADC**, sendo possível deduzir a tensão de entrada no canal utilizando a seguinte expressão:

$$V_{in} = \frac{ADC \times V_{ref}}{2^{bits}}$$

Exemplificando

Qual seria a tensão de entrada de um sensor genérico, sabendo que o valor convertido pelo ATmega328 é de 413, sua tensão de referência é de 5V, e sua resolução de 10 bits?

Aplicando a fórmula, temos:

$$V_{in} = \frac{ADC \times V_{ref}}{2^{bits}} \rightarrow V_{in} = \frac{413 \times 5,0}{2^{10}} \rightarrow V_{in} = 2,02\ V$$

Comunicação serial USART

Você já imaginou como é possível para um dispositivo eletrônico como o microcontrolador se comunicar com o usuário do sistema, a fim de enviar e receber informações de sua operação?

Em muitos casos é necessário que o microcontrolador estabeleça um canal de comunicação com outros dispositivos (como um computador) para a troca de informação, por exemplo pelo envio de dados estatísticos do processo ou do recebimento de parâmetros que têm o poder de modificar o modo de operação do sistema.

Uma das formas de se implementar um protocolo de comunicação entre um microcontrolador e o meio externo é pelo

periférico USART (*Universal Synchronous and Assynchronous Serial Receiver and Transmitter*), que é um modulo de comunicação serial com inúmeras possibilidades de configurações de trabalho, o que lhe permite ser aplicado em uma infinidade de sistemas eletrônicos, como nas comunicações RS-232 e RS-485 (FREITAS, 2017) que, apesar de não serem mais utilizadas em computadores pessoais, são ainda largamente usadas em sistemas industriais de controle. A grande vantagem da USART é que muitos dispositivos eletrônicos modernos suportam seu protocolo de comunicação (LIMA, 2012).

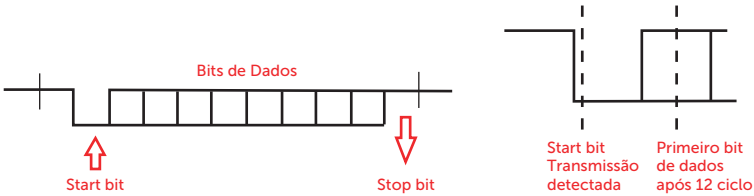
Uma das principais formas de operação do periférico USART é pela transmissão de dados assíncrona, que permite que os dados sejam enviados sem que uma das partes tenha que enviar um sinal de *clock*. Para que a comunicação ocorra, ambos dispositivos devem ter seus tempos de transmissão e recepção alinhados e configurados previamente, e alguns bits especiais são incluídos no pacote para sincronizá-los. Um exemplo é o *Start Bit* (bit de inicialização), que adicionado ao início de cada pacote, é utilizado para alertar o outro dispositivo, de forma que o prepare para o recebimento dos dados.

O valor de *Baud Rate* é a medida padrão para velocidade de transmissão em sistemas assíncronos, dada em bits por segundo (bps). Todos os dispositivos em uma rede assíncrona devem apresentar o mesmo valor de *baud rate* para que não haja falha de comunicação.

Um pacote de dados em uma comunicação assíncrona pode conter alguns bits de controle para garantir a entrega confiável da informação entre os dispositivos. Dentre os principais, podemos destacar:

- Bit de paridade: bit complementar adicionado ao final do pacote como um meio de análise simples de falha. Em um pacote com paridade *par*, o bit de paridade será 0 se a quantidade de bits '1' no pacote de dados for par, em um pacote com paridade *ímpar* a lógica é inversa. O dispositivo que recebe o pacote pode checar a consistência dos dados recebidos recalculando a paridade do pacote e confrontando com o bit recebido de paridade.
- Bit de parada ou *Stop bit:* bit que indica o final do pacote de dados.

Figura 1.19 | Exemplo de pacote de dados em comunicação assíncrona



Fonte: elaborada pelo autor.

Pesquise mais

Entenda mais sobre os registradores utilizados pelo periférico USART no modo de transmissão assíncrona e sobre como podemos configurá-los no seguinte link: <http://www.arnerobotics.com.br/eletronica/Configurando_corretamente_regsAVR_pt5.htm> (SOARES, [s. d.]).

Memória EEPROM em microcontroladores

Em várias situações, um sistema microcontrolado perde sua alimentação, propositalmente ou não. Como é possível reter dados importantes para a aplicação nesses casos?

Para contornar esse problema, alguns microcontroladores implementam em sua arquitetura memórias do tipo EEPROM (*Electrically-Erasable Programmable Read-Only Memory*). As memórias EEPROM apresentam características de não volatilidade semelhantes à memória Flash (utilizadas para o armazenamento do software em um microcontrolador), e têm como vantagem o acesso individual de leitura e escrita para cada byte, não necessitando que todo um bloco de endereços seja apagado. Porém sua principal desvantagem é o seu tempo de acesso: são necessários alguns milissegundos para uma simples escrita. Esse tipo de memória é muito utilizada para o armazenamento de dados críticos para o sistema, que não podem ser perdidos ou apagados quando ocorre o seu desligamento (LIMA, 2012).

A memória EEPROM do microcontrolador ATmega328 apresenta um total de 1Kbytes (1024 bytes) de endereços para armazenamento de dados, sendo que cada byte pode ser sobrescrito minimamente até 100 mil vezes. O acesso à memória EEPROM é feito pela memória de I/O, ou seja, o mesmo espaço de memória utilizado para o acesso e configuração dos demais periféricos do microcontrolador. São

reservados desta forma, registradores específicos para controlar a leitura e escrita de dados, e também para direcionar em qual endereço o valor deve ser armazenado. Os endereços da memória EEPROM vão desde o valor 0x00 (0 em decimal) até 0x3FF (1023 em decimal).

Reflita

Apesar de dispor de até 100 mil ciclos de escrita e leitura, esse número é finito, e quando ultrapassado pode gerar corrupção dos dados armazenados e lidos. Como podemos evitar que isso ocorra em projetos que utilizam a memória EEPROM frequentemente?

A memória EEPROM é um periférico cuja estabilidade de funcionamento depende muito do nível de tensão de alimentação. Em algumas condições, dados corrompidos podem ser escritos ou lidos dessa memória pelo fato de a tensão de entrada (Vcc) ser inferior ao mínimo adequado para operação do microcontrolador. Desta forma, um cuidado especial com as fontes de alimentação deve ser tomado em projetos nos quais se deseja utilizar a memória EEPROM para o armazenamento de dados críticos.

O processo de escrita da memória EEPROM no microcontrolador ATMega328 é relativamente simples, e somente alguns cuidados devem ser tomados para garantir a integridade dos dados lidos ou armazenados. É fundamental que durante esse processo nenhuma interrupção ocorra, sendo assim o bit de interrupção global I do registrador SREG deve ser desabilitado momentaneamente e habilitado logo após o término da operação.

De acordo com os registradores e bits de trabalho da EEPROM, os seguintes passos são necessários para efetuar-se uma escrita (a ordem dos passos 2 e 3 não é importante). Esses passos são necessários para evitar uma eventual escrita acidental (LIMA, 2012).

1. Esperar até que o bit **EEPE** do registrador **EECR** se torne zero.
2. Escrever o novo endereço da EEPROM no registrador **EEAR.**
3. Escrever o dado a ser gravado no registrador **EEDR**.
4. Escrever 1 logico no bit **EEMPE** enquanto escreve 0 no bit **EEPE** do registrador **EECR**.
5. Dentro de quatro ciclos de clock após ativar **EEMPE**, escrever 1 logico no bit **EEPE**.

Para o processo de leitura, deve-se seguir os seguintes passos:

1. Esperar até que o bit **EEPE** do registrador **EECR** se torne zero.
2. Escrever o novo endereço da EEPROM no registrador **EEAR** (opcional).
3. Escrever 1 logico no bit **EERE** do registrador **EECR**.
4. Realizar a leitura do dado pelo registrador **EEDR.**

Sem medo de errar

Lembre-se que você atua junto a um time técnico que está desenvolvendo um novo produto de automação residencial, cuja primeira etapa fora validada com sucesso juntamente com uma empresa parceira que tem interesse em adquirir o produto. O time de marketing fez um levantamento de mercado para saber a quais funcionalidades extras o público-alvo mais se atentaria, e o resultado foi: a) Controle de iluminação do ambiente por meio de dimerização controlada das lâmpadas; b) Abertura automática das persianas de acordo com a leitura da temperatura ambiente e externa da residência; c) Terminal de comunicação serial para entusiastas que desejam acessar e controlar a automação de sua residência por comandos customizáveis; e d) Possibilidade de salvar programas pessoais com configurações predefinidas dos atuadores da casa. O chefe da engenharia lhe solicitou um relatório técnico em que você deve descrever como utilizar os periféricos do microcontrolador para atender esse novo escopo do produto.

Controle de iluminação por dimerização

Por ser um projeto inovador, considera-se a utilização de lâmpadas com tecnologia led para a instalação nas residências a serem automatizadas. Como o objetivo da dimerização é controlar digitalmente a intensidade luminosa das lâmpadas, a técnica mais adequada é a utilização de um sinal PWM, que pode ser implementado por qualquer um dos três temporizadores presentes no microcontrolador ATMega328, e cujo valor de *duty cycle* pode ser ajustado conforme a configuração do usuário.

Abertura das persianas de acordo com sensores de temperatura

Existem diversos tipos de sensores de temperatura, como aqueles que geram um valor de tensão em suas saídas proporcionalmente ao valor mensurado de temperatura, e outros que modificam sua resistência ôhmica em função desta mesma relação. Um exemplo muito comum são os termopares, sensores simples, robustos e muito confiáveis (MUNDO DA ELÉTRICA, [s. d.]. Para o requisito proposto, recomenda-se utilizar um canal de conversão AD para leitura do sensor de temperatura, em conjunto com as saídas digitais do microcontrolador. Deve-se analisar o manual do sensor a fim de identificar o valor máximo de tensão de saída, e assim definir com precisão o valor de tensão de referência do conversor. Podemos optar pela configuração de 10 bits de resolução, para obter valores mais precisos na conversão. Como a temperatura interna e externa ambiente varia lentamente, definiremos um tempo de conversão maior, e assim garantimos a integridade da leitura do sinal.

Terminal de comunicação serial para entusiastas

Pode-se implementar um hardware que possibilite a comunicação serial do sistema por meio do protocolo RS232. No microcontrolador, pode-se habilitar o periférico USART operando em comunicação assíncrona, com *stop bit* e bit de paridade, para garantir a confiabilidade da troca de pacotes de dados. O valor de *baud rate* deve ser alto para que o usuário não perceba nenhum sinal da latência de controle do sistema quando ele executar comandos de controle ou configuração.

Possibilidade de salvar programas pessoais

Para esse requisito é recomendável que se utilize o mesmo canal de comunicação implementado no item anterior em conjunto com a adoção da memória EEPROM, a qual será utilizada para salvar os valores de configuração desejados pelo usuário em uma zona de memória não volátil para que, mesmo em situação de falta de alimentação, os parâmetros não sejam perdidos ou alterados.

Avançando na prática

Controle programável de temperatura

Descrição da situação-problema

Após a entrega do relatório sobre os novos requisitos, o time de projeto se animou com seus conhecimentos sobre os periféricos avançados do microcontrolador e lhe solicitou uma análise sobre um novo requisito funcional que possivelmente envolveria os mesmos periféricos. O novo requisito trata da implementação de controle de temperatura interna do ambiente pela configuração prévia de um valor de temperatura realizada pelo usuário. A temperatura seria controlada com as ventoinhas baseadas em motores DC e aquecedores à base de resistências elétricas. Quais seriam os periféricos envolvidos nessa implementação, e como seriam utilizados?

Resolução da situação-problema

Para esse novo requisito, podemos considerar a utilização dos periféricos: PWM, Conversor Analógico-Digital, USART e EEPROM.

Inicialmente, podemos considerar a utilização de temporizadores em modo PWM para o controle da velocidade das ventoinhas, bem como o controle de potência desempenhada pelas resistências elétricas. Os valores adequados de *duty cycle* para cada acionamento seriam calculados com base na leitura do sensor de temperatura, via conversor analógico-digital. Esses valores seriam comparados a outro valor previamente configurado pelo usuário e que estaria salvo em memória EEPROM, para evitar sua perda em caso de desenergização do sistema. A alteração desse parâmetro, bem como o acompanhamento da operação dessa nova funcionalidade poderia ser realizado com a implementação de um canal de comunicação serial, utilizando o periférico USART.

Uma ferramenta matemática muito utilizada para o controle de sistemas dinâmicos são os Controladores PID. Para conhecer mais sobre o tema, acesse: *Controle PID em sistemas embarcados*. <https://www.embarcados.com.br/controle-pid-em-sistemas-embarcados/>.

Faça valer a pena

1. As memórias EEPROM apresentam características de não volatilidade semelhantes à memória Flash (utilizadas para o armazenamento do software em um microcontrolador), têm como vantagem o acesso individual de leitura e escrita para cada byte, não necessitando assim que todo um bloco de endereços seja apagado.

Para qual aplicação é recomendável a utilização de memórias do tipo EEPROM?

a) Armazenar variáveis locais do software.

b) Armazenamento do programa aplicativo.

c) Armazenar dados e parâmetros que não podem ser perdidos.

d) Operar como registradores para a CPU.

e) Operar como memória cache para auxiliar o processamento da CPU.

2. O conversor AD do microcontrolador ATMega328 apresenta uma resolução configurável de 8 ou 10 bits, podendo assim ter uma representação de até 1024 pontos. O valor de tensão de referência pode ser atribuído por um pino externo chamado VREF ou utilizando a própria tensão de alimentação como referência.

Qual seria a tensão de entrada de um sensor genérico, sabendo que o valor convertido pelo ATmega328 é de 120, sua tensão de referência é de 3,3V e sua resolução de 8 bits?

a) Vin = 1,14 V.

b) Vin = 3,0 V.

c) Vin = 2,36 V.

d) Vin = 1,55 V.

e) Vin = 0,57 V.

3. Em um microcontrolador com saídas puramente digitais, não é possível conceber sinais analógicos para seu controle, porém, se utilizarem tais pinos para a geração de ondas digitais que variam entre zero e um em alta frequência, o valor de tensão resultante nesse pino será um valor médio de tensão proporcional ao tempo em que o pino permaneceu ligado (*duty cycle*) em relação ao tempo total do período dessa onda. Essa técnica é chamada de PWM, ou Modulação por Largura de Pulso.

Considerando o microcontrolador ATMega328 operando a uma frequência de *clock* de 16MHz, qual deve ser a configuração de *Prescaler*

e do registrador OCR0A para que o temporizador TC0 gere um sinal PWM com frequência de 488Hz e *duty cycle* de 50%?

a) *Prescaler* de 64, e OCR0A em 127.

b) *Prescaler* de 8, e OCR0A em 127.

c) *Prescaler* de 64, e OCR0A em 255.

d) *Prescaler* de 1024, e OCR0A em 255.

e) *Prescaler* de 8, e OCR0A em 255.

Referências

ATMEL CORPORATION. **ATmega328/P Datasheet Complete**. San Jose, nov. 2016. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso em: 02 maio 2018.

BRAGA, N. C. **O básico sobre os Microcontroladores – parte 1 (MIC139)**. Disponível em: <http://newtoncbraga.com.br/index.php/eletronica/52-artigos-tecnicos/artigos-diversos/13263-o-basico-sobre-os-microcontroladores-parte-1-mic139>. Acesso em: 8 abr. 2018.

CIRCUITSTODAY. **Microcontroller – Invention History and Story Behind the Scenes**. Out. 2013. Disponível em: <http://www.circuitstoday.com/microcontroller-invention-history>. Acesso em: 8 abr. 2018.

FREITAS, C. M. Controle PID em sistemas embarcados. **Embarcados**, [s. l.], mar. 2014. Disponível em: <https://www.embarcados.com.br/controle-pid-em-sistemas-embarcados/>. Acesso em: 15 abr. 2018.

FREITAS, C. M. Redes de comunicação em RS-485. **Embarcados**, [s. l.], jun. 2017. Disponível em: <https://www.embarcados.com.br/redes-de-comunicacao-em-rs-485>. Acesso em: 6 maio 2018.

LIMA, C. B. de; VILLAÇA, M. V. M. **AVR e Arduino**: Técnicas de Projeto. Florianópolis: Clube de Autores, 2012.

MUNDO DA ELÉTRICA. **O que é um termopar?** [S. l., s. d.]. Disponível em: <https://www.mundodaeletrica.com.br/o-que-e-um-termopar/>. Acesso em: 15 abr. 2018.

PICORETI, Rodolfo. **Entrada e Saída – Manipulando Registradores. Vida de silício**. Maio 2017. Disponível em: <https://portal.vidadesilicio.com.br/entrada-e-saida-manipulando-registradores>. Acesso em: 28 mar. 2018.

SOARES, M. J. Configurando corretamente os registradores dos microcontroladores AVR – parte 5: uso dos registros UCSRA, UCSRB, UCSRC, UDR, UBRRL e UBRRH na comunicação assíncrona. **Arne Robotics**, [s. d.]. Disponível em: <www.arnerobotics.com.br/eletronica/Configurando_corretamente_regsAVR_pt5.htm>. Acesso em: 15 abr. 2018.

Desenvolvimento de projetos microcontrolados

Convite ao estudo

Caro aluno, nesta segunda unidade, continuaremos nossa jornada nos estudos dos microcontroladores. Desta vez focaremos nos aspectos práticos do desenvolvimento, envolvendo os ambientes para criação de programas e exemplos reais utilizando todos os periféricos que estudamos na unidade anterior. Assim como em uma atividade nova que pretendemos nos dedicar, em que primeiramente estudamos seus conceitos e depois entendemos as regras e os processos dessa atividade, em relação a projetos envolvendo microcontroladores o método é muito semelhante. Agora que já dominamos os conceitos que fundamentam as arquiteturas e os periféricos dos microcontroladores, podemos começar a vislumbrar quais são as ferramentas adequadas para o desenvolvimento e como utilizar uma linguagem de programação para realmente desenvolver projetos que utilizem todo o potencial desse importante dispositivo eletrônico, como veículos, eletrodomésticos, e celulares por exemplo.

Na primeira seção, entenderemos como configurar um ambiente de desenvolvimento de maneira adequada, incluindo não somente um bom editor de texto para a criação do código-fonte, mas também todas as demais ferramentas que utilizaremos para geração de código em linguagem de máquina. Na próxima seção, criaremos algumas aplicações práticas utilizando os periféricos básicos do microcontrolador, como IO's, interrupções e temporizadores. Por fim, na última seção desta unidade, desenvolveremos projetos mais complexos, envolvendo os periféricos avançados do microcontrolador.

Ao final dessa unidade, devemos estar aptos a conhecer as principais ferramentas para o desenvolvimento de programas embarcados para microcontroladores, bem como criar aplicações práticas de *software* que empreguem todo o conhecimento adquirido sobre os periféricos do mesmo.

O seu trabalho, durante a primeira fase do projeto, fora muito elogiado pelo time de desenvolvimento. O líder do setor de engenharia de *software* o convidou para participar do planejamento e desenvolvimento do *software* do microcontrolador que será adotado no produto.

Nesta unidade, você embarcará em uma nova fase do desenvolvimento, na qual deverá auxiliar o time técnico a escolher e configurar o melhor ambiente de desenvolvimento para o projeto, bem como participar dos trabalhos para a programação da lógica de configuração e manipulação de todos os periféricos que foram escolhidos e estudados na fase inicial do projeto.

Na sua opinião, o que devemos considerar quando pensamos em iniciar um projeto de *software* para microcontroladores? Já que entendemos quais são e como operam os principais periféricos de um microcontrolador, como utilizar uma linguagem de programação qualquer para gerar códigos executáveis que interajam com esses periféricos? Esses programas são semelhantes a outros que criamos para computadores por exemplo?

Pronto para essa nova fase de conhecimento? Bons estudos e vamos lá!

Seção 2.1

Ambientes de desenvolvimento e simulação

Diálogo aberto

Caro aluno, agora que já aprendemos como funciona um microcontrolador e como são configurados seus periféricos para realizar funções específicas, como podemos transformar todo esse conhecimento em algo prático? Ou seja, você conseguiria imaginar a forma mais adequada para gerar as instruções necessárias para realizar tais configurações e portá-las para dentro de um microcontrolador?

Nesta seção, iniciaremos nossa jornada sobre os aspectos práticos de projetos eletrônicos microcontrolados. Agora que já possuímos todo o conceito teórico sobre como é estruturado um microcontrolador e como funcionam seus periféricos, estamos prontos para iniciar o desenvolvimento do *software* que controlará esse poderoso dispositivo. Veremos que, antes de iniciar o desenvolvimento do código propriamente dito, precisamos preparar *a priori* todas as ferramentas que nos auxiliarão no processo de criação, compilação e depuração do nosso código. Veremos também que podemos utilizar ferramentas valiosas de simulação para estudar e avaliar o desempenho do *software* projetado, seja simplesmente avaliando o comportamento dos periféricos, ou realizando simulações mais avançadas que têm o potencial de integrar o microcontrolador portador do código em um circuito eletrônico complexo, com periféricos externos.

Nesta unidade, você deverá auxiliar o time técnico a escolher e configurar o melhor ambiente de desenvolvimento para o projeto, bem como participar dos trabalhos para a programação da lógica de configuração e manipulação de todos os periféricos que foram escolhidos e estudados na fase inicial do projeto. Nesta primeira etapa, lhe foi solicitado que seja entregue um tutorial contendo os passos a serem seguidos para a perfeita configuração do ambiente ATMEL Studio para que um desenvolvedor possa iniciar o desenvolvimento de *software* com o microcontrolador ATMega328,

incluindo o processo de depuração do *software*. Inicialmente, o time de *software* não poderá dispor de placas de *hardware* para carregar e testar o sistema. Logo, o seu tutorial deve conter também um descritivo de como utilizar um simulador para que o time possa testar virtualmente suas implementações.

Nesta seção, partiremos de alguns conceitos fundamentais sobre o processo de desenvolvimento, desde a configuração de um ambiente adequado, passando pela criação do *software*, depois pela sua compilação, e finalmente realizando um processo chamado *debug* (termo em inglês para depuração).

Por onde devemos começar quando pretendemos configurar um ambiente de desenvolvimento? Quais são as funcionalidades que não devem ser esquecidas para compormos um ambiente mínimo de trabalho? Esta é a sua primeira oportunidade com o seu novo time, então faça acontecer!

Não pode faltar

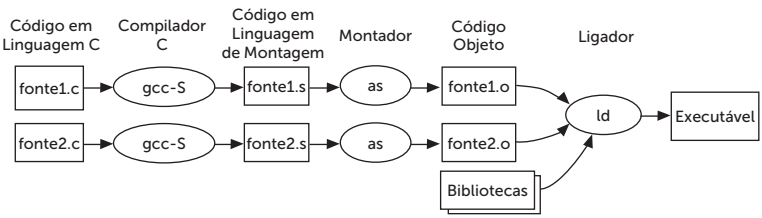
Introdução

Inicialmente, temos que lembrar que um microcontrolador ou um microprocessador não é capaz de interpretar o "texto" daquilo que denominamos código-fonte, visto que tais dispositivos só são capazes de entender e executar comandos binários pré-determinados, representados simbolicamente por textos chamados de mnemônicos. Dessa forma, após a conclusão do *software*, o mesmo é processado por compiladores, que possuem a tarefa de converter toda a abstração presente no código em artefatos equivalente em linguagem de máquina, geralmente representados por arquivos de extensão binária (.bin) ou hexadecimal (.hex). Durante esse processo de transformação, erros sintáticos e semânticos são verificados e otimizações podem ser realizadas, visando eliminar redundâncias ou melhorar o desempenho do programa em termos de velocidade, ocupação de memória, ou ambos.

Exemplificando

Como exemplo desse processo, podemos considerar a linguagem de programação predominante no desenvolvimento de projetos microcontrolados e aquela que utilizaremos nesta unidade: a Linguagem C (Editorial Linguagens para Sistemas Embarcados. Disponível em: <https://www.embarcados.com.br/editorial-linguagens-para-sistemas-embarcados/>. Acesso em 26 abr. 2018). Considere a Figura 2.1 a seguir.

Figura 2.1 | Processo de compilação para um código-fonte em C



Fonte: Laboratórios de microcontroladores - UNICAMP. Disponível em: <http://www.ic.unicamp.br/~edson/disciplinas/mc404/2014-2s/ef/labs/lab01.html>. Acesso: 26 abr. 2018.

Por meio da Figura 2.1, podemos observar que o processo completo de compilação em linguagem C decorre de 3 etapas fundamentais:

- Transformação do código .c em código de montagem .s através do compilador GCC.
- Geração do código objeto .o.
- Ligação dos objetivos a outros objetivos de bibliotecas para geração do código executável .bin ou .hex.

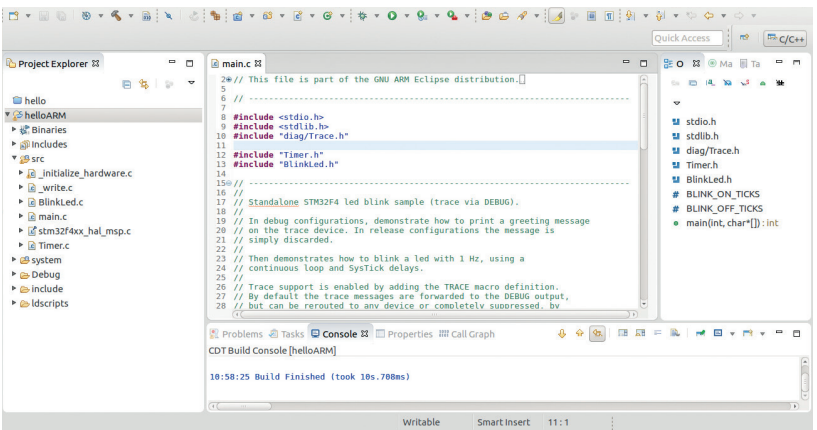
Como produto intermediário desse processo, é possível configurar o compilador para a geração de código *assembly*, que pode ser útil para compreender em maiores detalhes como se dará a configuração do microcontrolador, uma vez que esse código representa diretamente os mnemônicos que serão executados pelo microcontrolador.

Ambiente de Desenvolvimento de *Software*

Você pode estar se questionando agora: "Como concentrar todas essas ferramentas de compilação, além do desenvolvimento de código em si?" A resposta está nos ambientes de desenvolvimento de *software* ou IDE's (*Integreted Develpment Environment*).

Uma IDE é um ambiente integrado que reúne várias ferramentas necessárias para o desenvolvimento do *software*, auxiliando para que todo o processo seja o mais simples e direto possível. Além de prover um bom editor de texto para que o código seja construído e organizado, esse ambiente também proporciona outras ferramentas fundamentais que envolvem todo o ciclo de desenvolvimento, tais como: compiladores, depuradores, visualizadores de memória, gravadores de código executável, controladores de versão, dentre outros. Na Figura 2.2, temos o exemplo da IDE Eclipse, muito utilizada no desenvolvimento para a Linguagem C.

Figura 2.2 | IDE Elipse para Linguagem C



Fonte: *Setup Eclipse with GCC ARM Embedded on 32 bits Ubuntu 16.04 i386.* Disponível em: <https://embedded-things.blogspot.com.br/2016/06/setup-eclipse-with-gcc-arm-embedded-on.html>. Acesso: 26 abr. 2018.

Configuração da IDE Atmel Studio

Como elencamos o microcontrolador ATMega328 como exemplo prático de microcontrolador, qual seria a IDE mais adequada para ele? O seu fabricante, a ATMEL, pertencente ao grupo Microchip, disponibiliza gratuitamente um ambiente integrado de desenvolvimento totalmente voltado para microcontroladores da família AVR e ARM ATMEL, possuindo editores de texto inteligente, compiladores GCC e ferramentas de auxílio ao desenvolvimento, como depuradores e simuladores de periféricos (Atmel Studio 7. Disponível em <http://www.microchip.com/avr-support/atmel-studio-7>. Acesso: 26 maio 2018).

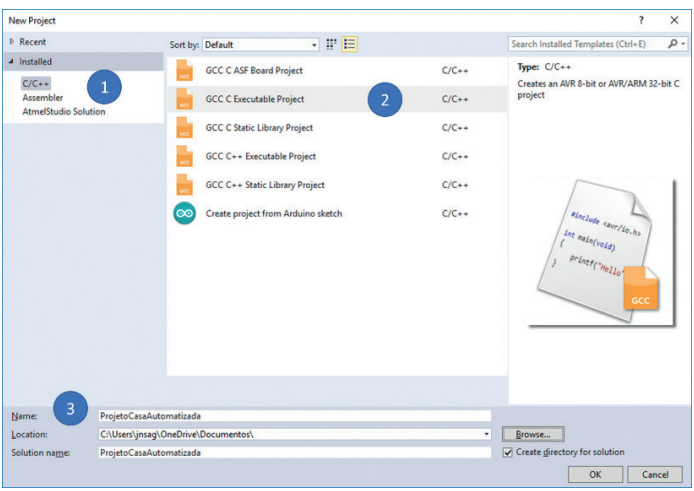
Pesquise mais

Que tal conhecer mais detalhes sobre o ambiente ATMEL Studio e os primeiros passos para configurá-lo corretamente? Acesse: Primeiros passos com Atmel Studio. Disponível em: <https://www.embarcados.com.br/atmel-studio>. Acesso: 22 maio 2018.

Como vimos nas seções anteriores, cada fabricante disponibiliza centenas de microcontroladores com características muito semelhantes entre si, diferindo-se em aspectos como tipos e quantidades de periféricos, processadores, quantidade de memória e etc. Com o objetivo de acelerar o processo de desenvolvimento, e evitar que seja desprendido esforço demasiado, os fabricantes adequam suas IDE's de forma que o usuário simplesmente selecione qual modelo de microcontrolador utilizará, e todo o processo de configuração e seleção das ferramentas mais adequadas é executado automaticamente.

Ao abrir o *software* Atmel Studio e selecionar a opção de "Novo Projeto", nos é apresentado o seguinte formulário representado pela Figura 2.3.

Figura 2.3 | Formulário de criação de um novo projeto com Atmel Studio



Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Este formulário apresenta as seguintes opções de configuração do projeto:

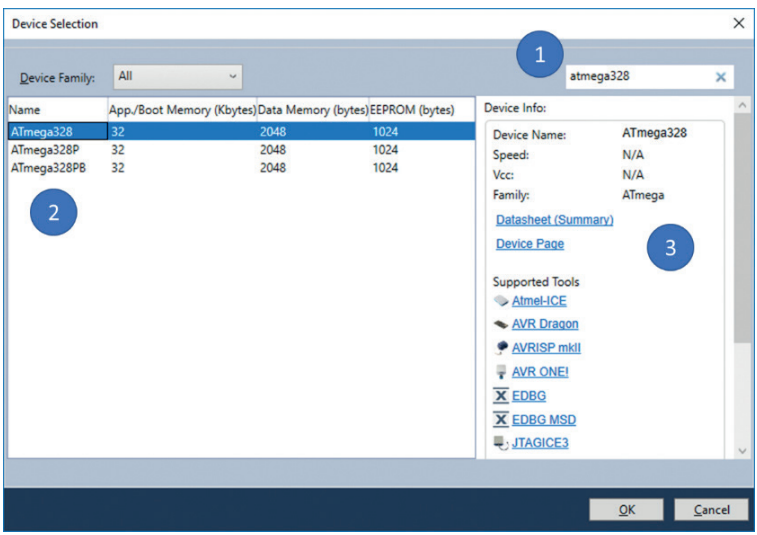
1. Tipo de Linguagem: qual linguagem será utilizada para o desenvolvimento do projeto, ou seja, C / C++ ou assembly.
2. Objetivo do projeto: opção de qual o alvo do projeto, seja um código executável em C ou C++, ou uma biblioteca estática que poderá ser referenciada por outros projetos.
3. Nome e caminho: onde o projeto será salvo, e qual o nome que se dará a ele.

Exemplificando

Se desejarmos realizar um projeto de *software* em mais baixo nível que a linguagem C, poderíamos configurar a IDE para o desenvolvimento em *Assembly,* bastando para isso alterar o tipo de linguagem no item 1 da Figura 2.3

A próxima etapa é a escolha do microcontrolador que será utilizado para o projeto, conforme pode ser visto na Figura 2.4.

Figura 2.4 | Formulário de escolha do microcontrolador



Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor

Nesse formulário, temos as seguintes informações:

1. Busca do microcontrolador: campo para busca de qual família de microcontrolador desejamos trabalhar.
2. Modelos: lista de todos os modelos variantes da mesma família, com características ligeiramente diferentes, como consumo e temperatura de operação.
3. Informações e Ferramentas: são mostradas as principais informações do dispositivo, um link para o seu manual, e também todas as ferramentas de *hardware* compatíveis com o mesmo, como depuradores e placas de desenvolvimento.

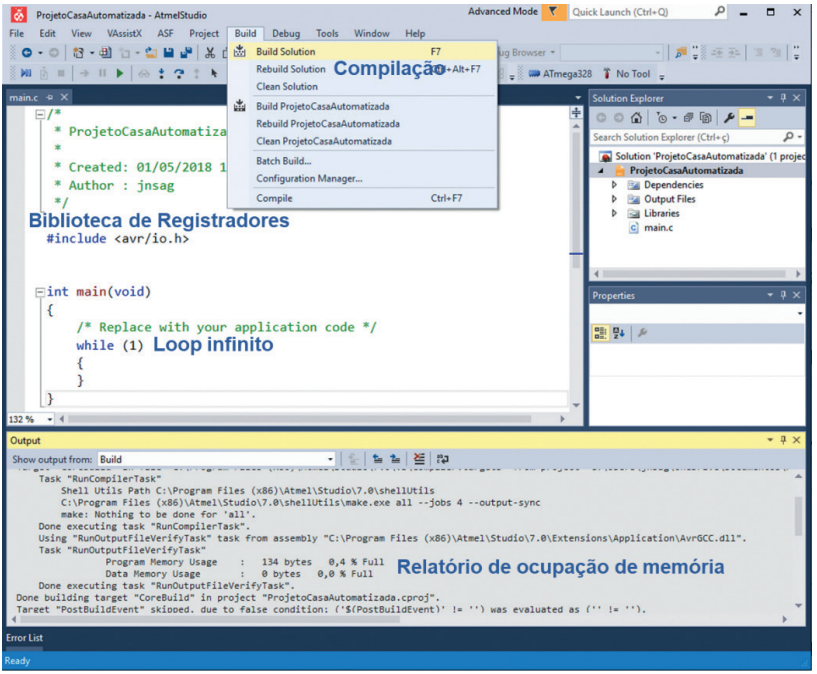
Assimile

Lembre-se de que, quando tratamos de famílias de microcontroladores, nos referimos a dispositivos que possuem arquiteturas semelhantes em termos de unidade de processamento, barramento e conjunto de instruções.

Ao término desse processo, a IDE gerará um arquivo de código-fonte em C, contendo uma função *main* e um loop infinito *while*, típico para projetos de software microcontrolados em linguagem C. Também será incluída a biblioteca **io.h**, a qual conterá todas as referências necessárias para que possamos evocar os nomes dos registradores e seus respectivos bits sem nos preocuparmos com o seu endereçamento físico na memória.

Ao clicar no menu *Build* → *Build Solution* (Termos em inglês para Construção da Solução, que significa a geração dos arquivos binários executáveis), veremos que a IDE evocará o compilador padrão (AVR-GCC) que converterá nosso código-fonte em código binário executável. Ao final desse processo, a IDE executará um *script* que tem por função analisar os arquivos binários e entregar um relatório contendo a taxa de ocupação da memória RAM e taxa de ocupação de memória Flash. Perceba que mesmo em um projeto sem nenhum código útil já existe uma ocupação de memória Flash, devido à inclusão da biblioteca io.h e de funções para redirecionamento das interrupções do microcontrolador.

Figura 2.5 | Compilação do projeto



Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Depuração

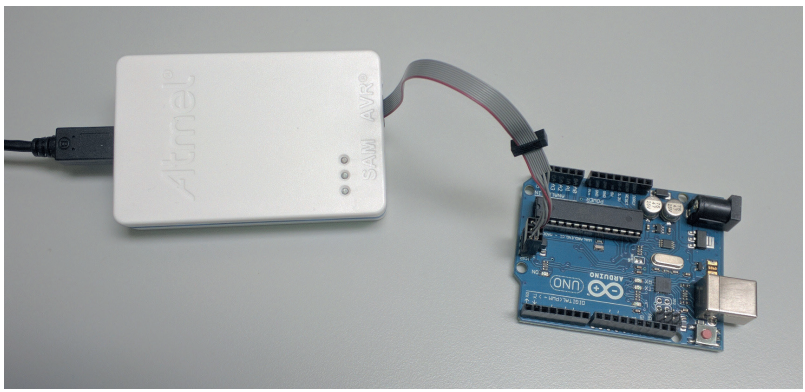
Você já imaginou como é possível compreender projetos de *software* complexos e solucionar problemas que envolvam muitas variáveis? Tudo isso é possível por meio de uma técnica chamada depuração (*debug*), que pode ser auxiliado por ferramentas de *hardware* e que tem por objetivo extrair o máximo de informação possível do contexto da aplicação para o correto entendimento do programa.

Em projetos de *software* que envolvem poucas linhas de código e poucos arquivos é relativamente simples entender e acompanhar o fluxo da informação de forma a compreender o funcionamento do sistema ou encontrar e resolver problemas, porém esse cenário representa uma parcela muito pequena dos projetos reais e comerciais. Em projetos maiores, contendo milhares de linhas de código e algumas dezenas de arquivos, a tarefa de resolver um simples *bug* (termo em inglês para falha) pode tomar várias

horas de trabalho de um desenvolvedor, se não amparado por ferramentas de depuração que lhe auxilie na execução controlada do *software* e que lhe forneça a maior quantidade possível de informação quanto a variáveis do código, valores dos registradores e estado dos seus periféricos.

Existem duas formas de realizar a depuração de um programa de microcontrolador: através de Simulação ou por ferramentas de *Hardware*. No processo de depuração por *Hardware* é possível controlar fisicamente o microcontrolador pela IDE por meio de dispositivos chamados de depuradores. Os microcontroladores que possuem a capacidade de depuração fornecem pinos específicos utilizados para implementar protocolos destinados a esse processo, como os protocolos **JTAG** e **DegubWire** que, além de possibilitarem o controle de execução das instruções do microcontrolador, também possibilitam a leitura de suas memórias RAM e Flash, bem como do estado dos seus registradores e periféricos. Veja, na Figura 2.6, um depurador ATMELICE conectado a uma placa Arduino Uno.

Figura 2.6 | Depurador ATMELICE conectado a uma placa Arduino UNO



Fonte: *Using Arduino with Atmel Studio 7*. Disponível em: <http://www.riptutorial.com/arduino/topic/2567/using-arduino-with-atmel-studio-7>. Acesso: 26 abr. 2018.

Também é possível depurar o *software* por meio de simulação virtual. Nesse método, pode-se utilizar a própria IDE para emular virtualmente a operação do microcontrolador.

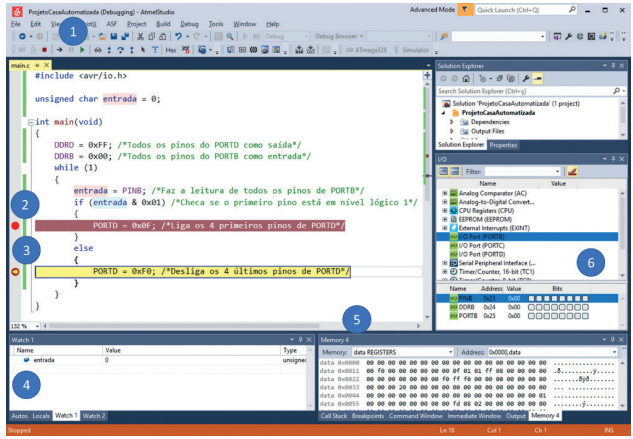
Reflita

É certo afirmar que depurações via *software* são mais econômicas e fáceis de se configurar, então quais seriam as vantagens em se depurar um programa de microcontrolador através de dispositivos de *hardware*?

Vamos considerar um exemplo utilizando o simulador da IDE Atmel Studio, em que desejamos depurar um código que configura todos os pinos do **PORTD** do microcontrolador como saída, e todos os pinos do **PORTB** como entrada. Deseja-se controlar 8 leds conectados ao PORTD através de uma chave ligada ao pino **PB0** que, ao ser acionada liga os leds conectados aos pinos **PD0** a **PD3**, e ao ser desligada inverte-se o acionamento. A Figura 2.7 ilustra o processo de simulação deste *software*, contendo as seguintes características:

1. Botão ***Run*** (Iniciar)*:* utilizado para iniciar o processo de depuração. Caso haja um *hardware* depurador conectado a uma placa externa, inicia-se a depuração via *hardware* e, caso não haja, a IDE inicia a simulação virtual.
2. ***Breakpoint*** (Ponto de parada): através de um duplo-clique em qualquer linha de código é possível ativar um ponto em que o depurador congelará a execução do programa e ficará neste ponto até que um novo comando de *Run* seja dado.
3. *Breakpoint* acionado: quando um programa está em execução via depurador não é possível acompanhar em tempo real o estado do microcontrolador e de suas variáveis, isso só se torna possível quando o fluxo de *software* entra em um *breakpoint*.
4. Monitoramento de variáveis: é possível incluir as variáveis do programa em uma lista de monitoramento e, sempre que o *software* entra em um *breakpoint* o seu valor é atualizado.
5. Monitoramento de memória: através desse recurso é possível monitorar o conteúdo de todos os periféricos de memória do microcontrolador, inclusive registradores.
6. Periféricos: através dessa ferramenta é possível visualizar e manipular todos os registradores que controlam os periféricos internos do microcontrolador.

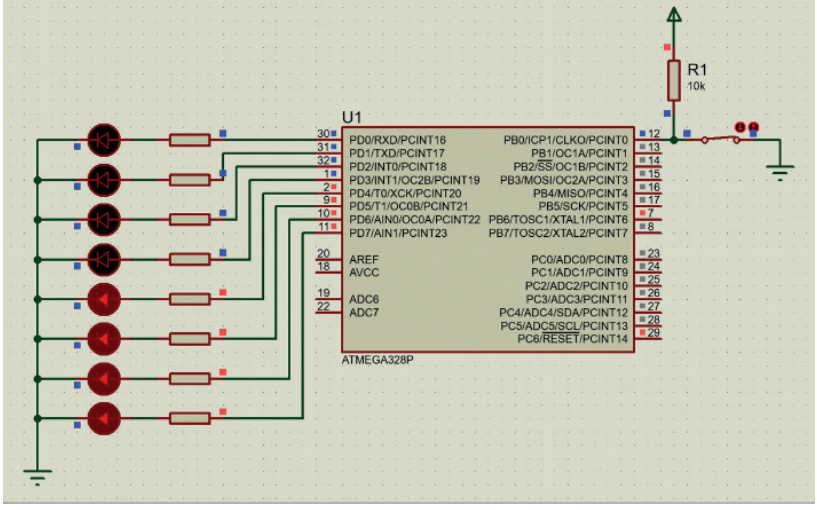
Figura 2.7 | Simulação de *software* através da IDE Atmel Studio



Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Outra forma de realizar uma depuração por simulação é através de simuladores de eletrônica, que são capazes de, não somente simular o microcontrolador agregado ao seu programa, como também incluir componentes eletrônicos a simulação, representando desta forma um ambiente mais rico e que fornece mais recursos para a análise do desenvolvedor. Na Figura 2.8, temos o exemplo da utilização do *software* Proteus para a simulação eletrônica.

Figura 2.8 | Simulação através do *software* Proteus



Fonte: captura de tela do *software* Proteus, elaborada pelo autor.

Sem medo de errar

Nesta unidade, você foi convidado pelo líder do time de desenvolvimento de *software* a se integrar em sua equipe. Foi informado a você que seu novo time precisa iniciar o desenvolvimento do programa que será embarcado ao projeto e você recebeu como primeira tarefa realizar um tutorial detalhando quais seriam os passos a serem seguidos por um desenvolvedor para a configuração da IDE Atmel Studio, visando trabalhar com o microcontrolador ATMega328 para dar início a esse desenvolvimento. Também foi solicitado a você que incluísse um descritivo de como utilizar um simulador para que o time possa testar as implementações.

A informação mais importante que possuímos é qual dispositivo desejamos trabalhar para que seja feita a adequação da IDE ao mesmo, e lhe foi passado que será o ATMega328. Logo podemos proceder com o tutorial através de uma lista de passos que o desenvolvedor deve seguir.

Tutorial de configuração da IDE Atmel Studio para o microcontrolador ATMega328

1º Passo: abra o aplicativo Atmel Studio.

2º Passo: navegue pelo menu e selecione a opção de novo projeto.

3º Passo: será apresentado um formulário para que seja escolhido o tipo de projeto que deseja criar. Selecione a opção de Projeto em C executável (*GCC C Executable Project*).

4º Passo: escolha um nome para o projeto, de preferência um nome de relevância para que seja lembrado futuramente.

5º Passo: escolha uma pasta onde o projeto será salvo, incluindo os arquivos executáveis.

6º Passo: feito isso, será apresentado a você um novo formulário no qual se deve selecionar o dispositivo para o qual o projeto será destinado. No campo de busca digite "ATMega328" e após, selecione o dispositivo de mesmo nome na lista a esquerda e prossiga.

7º Passo: se tudo correu bem até aqui, será apresentado a você uma tela de editor de texto com um código base, contendo a função principal (*main*), a inclusão da biblioteca io.h, e um laço infinito utilizando a diretiva *while*.

8º Passo: por fim, navegue até o menu *Build* → *Build Solution*, e verifique se o código é compilado e se é gerado um relatório contendo a taxa de ocupação das memórias do microcontrolador.

Simulação do *Software*

Para que se possa realizar a simulação do programa, inicialmente confirme que o programa foi compilado corretamente, sem a apresentação de nenhum erro. Para iniciar a simulação, pressione o botão *Run*, o sistema procurará por dispositivos de depuração via *hardware* e, caso não encontre, iniciará o processo de simulação virtual. Para realizar paradas em pontos de interesse no programa, dê um duplo-clique no canto esquerdo da linha e aparecerá um círculo vermelho preenchido, simbolizando um *breakpoint,* se a execução do programa chegar a tal ponto, a simulação congelará, para continuar com a simulação pressione novamente o botão *Run.*

É possível também adicionar variáveis do programa em uma lista de variáveis monitoradas, bem como acompanhar os valores dos registradores, das memórias, e a configuração atual dos periféricos do microcontrolador. Atente-se ao fato que tais valores somente apresentarão seus valores atualizados quando a simulação estiver parada, ou congelada em um *breakpoint*.

Avançando na prática

Configuração da IDE para projeto Assembly

Descrição da situação-problema

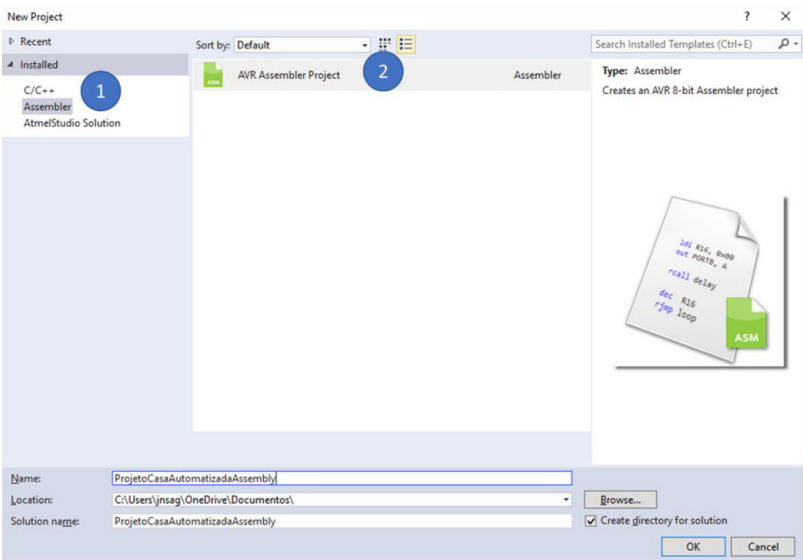
O líder do time de desenvolvimento de *software* gostou do trabalho realizado quanto à criação do tutorial para o desenvolvimento em linguagem C, e lhe passou uma nova demanda. Um dos desenvolvedores trabalhará em uma parcela do código chamada ***bootloader***, destinada a carregar e a gerenciar

as atualizações do programa atual. Porém, como o espaço em memória do microcontrolador ATMega328 é limitado, o mesmo optou por escrever essa parte do programa em linguagem Assembly, mas o mesmo possui algumas dúvidas em como configurar a IDE Atmel Studio para esse desenvolvimento. A sua tarefa é orientá-lo em como realizar essa tarefa, tendo por base o tutorial que já foi apresentado.

Resolução da situação-problema

Todos os passos listados no tutorial anterior são validos para esta nova orientação, com exceção do 3° Passo, em que se seleciona a opção de Projeto Executável em C como objetivo de desenvolvimento do projeto. Para o projeto em linguagem Assembly, deve-se selecionar a opção *Assembler* no campo superior direito do formulário, e depois selecionar o item *AVR Assembler Project* (Projeto AVR em Assembler), conforme demostrado na Figura 2.9. Feito isso, os demais passos continuam os mesmos, salientando que agora não mais se gerará um código automático em C com o um laço infinito, e sim um arquivo *assembly* com uma lógica análoga.

Figura 2.9 | Configuração da IDE Atmel Studio para um projeto em Linguagem Assembly



Fonte: captura de tela do *software* ATMEL, elaborada pelo autor.

Faça valer a pena

1. A respeito do desenvolvimento de um programa, um microcontrolador ou um microprocessador não é capaz de interpretar o "texto" daquilo que denominamos código-fonte, visto que tais dispositivos só são capazes de entender e executar comandos binários pré-determinados, também chamados de mnemônicos.

Qual a ferramenta responsável por transformar o código-fonte em arquivos executáveis?

a) Depurador.

b) Compilador.

c) Simulador.

d) IDE.

e) Decodificador.

2. Em projetos de *software* que envolvem poucas linhas de código e poucos arquivos é relativamente simples entender e acompanhar o fluxo da informação de forma a compreender o funcionamento do sistema ou encontrar e resolver problemas, porém esse cenário representa uma parcela muito pequena dos projetos reais e comerciais. Em projetos maiores, contendo milhares de linhas de código e algumas dezenas de arquivos, a tarefa de resolver um simples *bug* (termo em inglês para falha) pode tomar várias horas de trabalho de um desenvolvedor, se não aparado por ferramentas de depuração que lhe auxilie na execução controlada do *software* e que lhe forneça a maior quantidade possível de informação quanto a variáveis do código, valores dos registradores e estado dos seus periféricos.

Avalie as asserções apresentadas a seguir referentes a projetos de software de microcontrolados e a relação entre elas.

I. Em projetos de *software* microcontrolados é fundamental a utilização de ferramentas de depuração para agilizar o desenvolvimento de forma a aumentar a compreensão do fluxo do programa,

PORQUE

II. a medida que um programa evolui, a quantidade de variáveis e contextos de aplicação aumenta consideravelmente, tornando a compreensão da lógica pelo simples método da leitura do código uma tarefa árdua a sujeita a falhas de interpretação.

A respeito dessas asserções, assinale a opção correta.

a) As asserções I e II são proposições verdadeiras, e a II é uma justificativa da I.

b) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa da I.

c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.

d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.

e) As asserções I e II são proposições falsas.

3. Existem duas formas de realizar a depuração de um programa de microcontrolador: através de Simulação ou por ferramentas de *Hardware*. No processo de depuração por *Hardware,* é possível controlar fisicamente o microcontrolador pela IDE através de dispositivos específicos chamados de depuradores. Também é possível depurar o *software* através de simulação virtual em que pode se utilizar a própria IDE para emular virtualmente a operação do microcontrolador.

A respeito das formas de depuração de um projeto de *software* para microcontrolador considere as seguintes afirmações:

( ) A técnica de depuração via *hardware* é mais precisa que a técnica de simulação.

( ) A técnica de depuração por simulação só pode ser realizada na própria IDE.

( ) É mandatório que se utilize um *hardware* externo para se conectar ao microcontrolador no método de depuração por *hardware*.

( ) É possível realizar a simulação através de simuladores de eletrônica, embarcando o programa criado em um microcontrolador virtual.

Considerando a sequência, qual das alternativas abaixo retrata corretamente quais afirmações são falsas e quais são verdadeiras?

a) V, V, F, V.

b) F, V, F, V.

c) F, V, F, F.

d) V, F, F, V.

e) V, F, V, V.

Seção 2.2

Introdução a projetos microcontrolados

Diálogo aberto

Caro aluno, nesta seção, daremos prosseguimento a esta nova fase de estudos práticos sobre os microcontroladores. Desta vez, utilizaremos a linguagem de programação C para implementar diversos exemplos de aplicações que utilizem os periféricos básicos que foram estudos na Unidade 1, Seção 2. Inicialmente, desenvolveremos um simples código para controle de ligação de um led através da leitura de uma chave convencional, demonstrando a configuração e manipulação das portas de entrada e saída. Posteriormente, veremos como utilizar algumas bibliotecas para implementar rotinas de tratamento dos eventos de interrupção externa. E finalmente, utilizando o mesmo conceito de tais rotinas, configuraremos um temporizador para geração de eventos periódicos. Ao final de toda essa experiência, agregaremos o conhecimento desses três periféricos em um exemplo prático de implementação de um semáforo simples, com controle de pedestre.

Lembre-se de que nesta unidade você auxiliará o time de técnico na estruturação do *software* para o novo produto da empresa. Após sua grande ajuda na configuração do ambiente de desenvolvimento para os seus colegas de trabalho, o seu novo gestor decidiu lhe atribuir uma tarefa diretamente relacionada a construção do *software*: o controle automático da abertura das persianas. O sistema de controle contará com 2 entradas digitais: a) um botão giratório com 4 opções para a seleção da estação do ano, em que cada terminal do botão deve ser ligado a uma entrada digital do microcontrolador; e b) um sensor de presença instalado na entrada da residência que deve ser conectado a uma interrupção externa do microcontrolador.

Estão previstas também três saídas digitais: a) uma saída para o acionamento do motor DC da persiana da sala de estar; b) outra saída para a o motor da persiana da sala de jantar; e c) uma saída ligada a um relê que será ligado a um ventilador presente na sala

de estar que só será ligado em estações específicas do ano. O acionamento das saídas sempre deve ocorrer quando o sensor de presença for acionado, e a configuração para o acionamento será dependente da opção selecionada no botão de estações do ano, como segue:

- Verão: persianas acionadas durante 5 segundos. Ventilador ligado.
- Inverno: persianas acionadas durante 1 segundo. Ventilador desligado.
- Primavera: persianas acionadas durante 3 segundos. Ventilador ligado.
- Outono: persianas acionadas durante 2 segundos. Ventilador desligado.

Escreva um *software* em linguagem C para o microcontrolador ATMega328 que seja capaz de atender a especificação descrita acima, para que possa ser validada posteriormente pelo restante do time. Quais periféricos deveremos utilizar para cumprir essa tarefa? Poderíamos adotar algum periférico avançado em substituição de outro periférico mais básico? Nesta seção, veremos exemplos práticos de *hardware* e *software* utilizando os periféricos básicos do microcontrolador, que lhe guiará em com combiná-lo para resolver mais este desafio. Muito foco nessa importante etapa e faça o seu melhor!

Não pode faltar

Introdução

Agora que já compreendemos os aspectos teóricos de um microcontrolador, como podemos juntar todos esses conceitos e iniciar um projeto prático, manipulando todos seus periféricos? Como a teoria pode nos guiar no desenvolvimento de códigos eficazes, aproveitando ao máximo a arquitetura do microcontrolador?

Após estudarmos os ambientes de desenvolvimento e como configurá-lo, agora estamos habilitados a iniciar a criação dos primeiros *softwares* para o nosso microcontrolador. Iniciaremos pelos periféricos mais simples, e utilizaremos a linguagem C para o desenvolvimento do código.

Pesquise mais

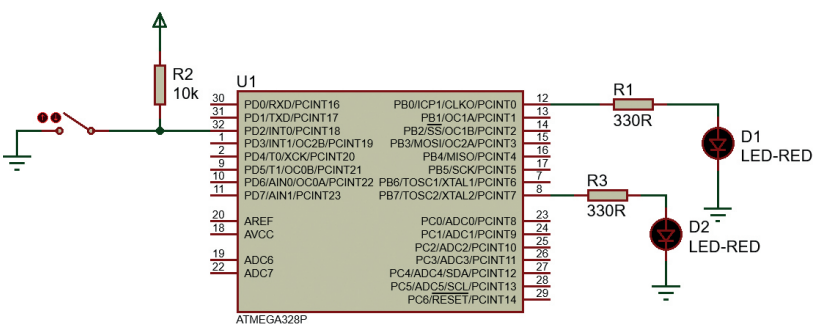
Nesta seção e na próxima, utilizaremos exaustivamente a linguagem C para a construção dos algoritmos de controle de microcontroladores. Que tal relembrar os fundamentos dessa poderosa linguagem? Acesse o acervo da Biblioteca Virtual e busque por obras consagradas, como MIZRAHI, Victorine Viviane. Treinamento em Linguagem C. São Paulo. Pearson Prentice Hall, 2008. Disponível em: <http://unopar.bv3.digitalpages.com.br/users/publications/9788576051916>. Acesso: 29 maio 2018.

Aplicação de IO's em microcontroladores

Como vimos nas seções da unidade anterior, as entradas e saídas digitais, ou simplesmente IO's, representam os mais simples e poderosos periféricos dos microcontroladores, uma vez que permitem o acesso direto a outros periféricos externos. Mas como podemos programá-los para a interface com tais periféricos, e quais cuidados devemos ter em termos físicos do *hardware*?

Para ilustrar a manipulação dos IO's, utilizaremos um circuito base, representado pela Figura 2.10. Nesse circuito, teremos uma chave mecânica ligada ao pino PD2 do microcontrolador, utilizando um resistor de *pull-up,* cujo objetivo é garantir o nível de sinal alto (1) na entrada do pino, evitando dessa forma efeitos colaterais como oscilações externas quando a chave está aberta.

Figura 2.10 | *Hardware* base



Fonte: captura de tela do *software* Proteus, elaborada pelo autor.

Observe que a lógica digital do pino PD2 será o inverso da lógica física da chave, ou seja, quando a chave está aberta, devido ao resistor de *pull*-up que conecta o pino a um potencial positivo, teremos o nível lógico 1 no pino, e na situação oposta, a chave conectará o pino ao referencial GND ao ser ligada, levando o pino ao nível lógico 0. Teremos também dois leds ligados nos pinos PB0 e PB7 respectivamente.

Pesquise mais

Os valores dos resistores do diagrama de *hardware* foram calculados de forma a respeitar o limite máximo de 40mA de fornecimento de corrente por pino de IO. Que tal conhecer mais sobre as limitações elétricas do microcontrolador? Acesse: ***"ATmega328 Datasheet Complete".*** Página 385. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso: 13 maio 2018.

O objetivo desse primeiro exemplo é o de configurar o pino PD2 como entrada digital para realizar a leitura do estado atual da chave, e configurar o pino PB0 como saída de forma a manipular o led correspondente. Feito isso, vamos alterar o estado do led de acordo com a leitura da chave: caso a chave esteja aberta (nível lógico 1) o led deve-se apagar, e caso a chave esteja fechada (nível lógico 0), o led deve-se acender. Analise o código-fonte deste exemplo, presente na Figura 2.11.

Figura 2.11 | Código-fonte para o exemplo de IO's

```
#include <avr/io.h>

#define set_bit(reg,bit) (reg |= (1 << bit))
#define rst_bit(reg,bit) (reg &=~ (1 << bit))
#define le_bit(reg,bit)  ((reg >> bit) & 0x01)

int main(void) {
    set_bit(DDRB, PB0);                 //Configuração de PB0 como saída
    while (1) {                         //Laço infinito
        if (le_bit(PIND, PD2) == 0)     //Faz a leitura da chave
            set_bit(PORTB, PB0);        //Se estiver desligada, ativa o Led
        else
            rst_bit(PORTB, PB0);        //Se estiver ligada, desliga o Led
    }
}
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

O primeiro passo do *software* consiste na inclusão da biblioteca **io.h**, que nos permite trabalhar com os registradores dos periféricos e seus bits diretamente pelos seus respectivos nomes. Logo após, são definidas três **macros auxiliares** que serão utilizadas para manipular os bits dos registradores individualmente.

Pesquise mais

Aprofunde seus conhecimentos sobre como a linguagem C realiza a manipulação de bits. Acesse: *Bits em Linguagem C*. Disponível em: <https://www.embarcados.com.br/bits-em-linguagem-c>.Acesso: 13 maio 2018.

Agora dentro da função principal (*main)*, antes do laço de ciclos infinitos, realizaremos uma operação que deve ser feita sempre que o microcontrolador é iniciado: a configuração dos registradores de IO, que definirá a direção de cada pino trabalhado. Como vimos na Unidade 1, Seção 2, utilizamos os registradores DDR para definir se um pino de IO de uma determinada Porta, será uma entrada ou uma saída. Como o estado natural do registrador DDR é 0x00, em que todos os pinos são configurados como entrada, o pino PD2 não foi explicitamente definido como entrada, somente o pino PB2, configurado como saída. Dentro do laço infinito, a lógica consiste em primeiramente realizar a leitura do bit PD2 dentro do registrador PIND, que possui o estado de todas os pinos da Porta D. Caso esse bit esteja em 0, significa que a chave está fechada, e assim é alterado o bit PD2 do registrador PORTD para 1, de forma que um sinal de VCC seja direcionado a pino, ligando o led. No caso oposto, o mesmo bit é levado a 0, que significa que o sinal de GND será direcionado ao pino, desligando o led.

Exemplo do uso de interrupção externas

Como vimos na Seção 2, da Unidade 1, as interrupções possuem a capacidade de interromper o fluxo natural do programa para o atendimento de eventos específicos. Mas como isso se dá em um código-fonte, uma vez que a programação convencional para a linguagem C é linear (MIZRAHI, 2008)?

Assimile

Bem como várias outras linguagens de programação, a linguagem C é uma linguagem linear, ou seja, as instruções são executadas sequencialmente, e desvios no *software* somente são possíveis através de chamadas de funções ou de instruções específicas para este fim, como instruções condicionais e de laço de execuções.

Nesse novo exemplo, veremos a utilização do pino PD2 utilizando no exemplo anterior configurado agora como fonte de interrupção externa por alteração de borda, ou seja, após configurado como tal, sempre que o sinal do pino passar de VCC para GND ou vice-versa, uma função específica do código será chamada para a tratativa deste evento. Nesse mesmo programa, dentro do laço infinito, o led conectado ao pino PB7 mudará seu estado a cada 1 s, utilizando a função ***_delay_ms()***, implementada pela biblioteca **delay.h**, que executa consecutivos comandos assembly vazios (***nop***) para emular o tempo de espera desejado. Veja a implementação desse código-fonte na Figura 2.12.

Reflita

Antes da inclusão da biblioteca delay.h é mandatório a definição através da macro F_CPU da frequência de *clock* que será utilizada pelo *hardware* em que o microcontrolador será alocado. Sabendo que tal atraso depende da execução de uma determinada quantidade de instruções vazias (nop), qual a utilidade da informação de frequência para este processo?

Figura 2.12 | Código-fonte para o exemplo de Interrupção Externa

```
#define F_CPU 16000000
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#define set_bit(reg,bit)    (reg |= (1 << bit))
#define rst_bit(reg,bit)    (reg &=~ (1 << bit))
#define le_bit(reg,bit)     ((reg >> bit) & 0x01)
#define troca_bit(reg,bit)  (reg ^= (1 << bit))

int main(void) {
    set_bit(DDRB, PB0);     //Configuração de PB0 como saída
    set_bit(DDRB, PB7);     //Configuração de PB7 como saída
    set_bit(EIMSK, INT0);   //Habilita a interrupção externa INT0
    set_bit(EICRA, ISC00);  //Configura INT0 para ativar em alterações de borda
    sei();                  //Habilita o bit de Interrupção Global
    while (1) {             //Laço infinito vazio
        troca_bit(PORTB,PB7); //Troca o estado do led
        _delay_ms(1000);    //Espera 1s executando comandos vazios "nop"
    }
}

ISR(INT0_vect){             //Vector de Interrupção por alteração de borda em INT0
    troca_bit(PORTB, PB0);  //Inverte o sinal do led ao termino do tempo
}
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

É possível observar no código-fonte a inclusão de uma nova biblioteca: a **interrupt.h**. Através dela é possível a utilização das funções **ISR** (*Interrupt Service Routine* ou Rotinas de Serviço de Interrupção). uma vez configurada uma nova fonte de interrupção, assim que o evento programado ocorre, a execução das instruções é interrompida imediatamente, e o fluxo de *software* é direcionado para a função ISR correspondente. Nesse exemplo, utilizamos o registrador EICRA para configurar a interrupção INT0 para ser ativada em condições alteração de borda através da ativação do bit ISC00, como estudamos na Unidade 1, Seção 2. Feito isso, habilitamos a ocorrência da interrupção ativando o bit INT0 no registrador EIMSK, e utilizando a função **sei()** para ativar internamente o bit I do registrador SREG, responsável por habilitar a interrupção global, que permite que as fontes de interrupção sejam acionadas.

Dessa forma, o microcontrolador executará os comandos de troca do estado do led e espera de 1s indefinidamente, até que qualquer alteração do estado da chave seja detectada pela interrupção INT0, que fará com que o código seja desviado para a ISR, que irá alterar o estado do led conectado em PB0.

Aplicação com geração de eventos periódicos

Os temporizadores são periféricos utilizados pelo microcontrolador para a geração de eventos em intervalos predefinidos e periódicos. Uma outra forma de gerar tais eventos é através de funções de atraso, como vimos no exemplo anterior, mas tais métodos são eficientes em termos computacionais e de uso útil do processador?

Ainda utilizando o circuito apresentado pela Figura 2.9, vamos decorrer, nesse novo exemplo, uma forma alternativa de alterar o estado do led conectado ao pino PB0 periodicamente, sem que seja necessário a adoção de funções de atraso que, como já vimos, executam funções vazias, diminuindo muito a taxa de processamento útil do microcontrolador. O código-fonte é apresentado na Figura 2.13 a seguir.

Figura 2.13 | Código-fonte para o exemplo de Temporizadores

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define set_bit(reg,bit)    (reg |= (1 << bit))
#define rst_bit(reg,bit)    (reg &=~ (1 << bit))
#define le_bit(reg,bit)     ((reg >> bit) & 0x01)
#define troca_bit(reg,bit)  (reg ^= (1 << bit))

int main(void) {
    set_bit(DDRB, PB0);     //Configuração de PD0 como saída
    set_bit(TCCR1B, CS12);  //Configura o Prescaler com o valor de 256
    set_bit(TIMSK1, TOIE1); //Habilita a Interrupção do TC0 por Estouro
    TCNT1 = 3036;           //Carrega o Timer 1 para contar 1s
    sei();                  //Habilita bit de Interrupção Global
    while (1) {}            //Laço infinito vazio
}

ISR(TIMER1_OVF_vect){       //Vector de Interrupção por estouro do Timer 1
    troca_bit(PORTB, PB0);  //Inverte o sinal do led ao termino do tempo
    TCNT1 = 3036;           //Recarrega o temporizador para o tempo de 1s
}
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Neste exemplo, o temporizador TC1 é configurado para a geração de um evento de interrupção periódica a cada 1s. Para isso, é necessário inicialmente o cálculo do fator de *Prescaler* que melhor atenda a aplicação. Considerando que o microcontrolador esteja

utilizando um *clock* externo de 16MHz, e que o temporizador TC1 possui 16 bits (possibilidade de contagem até 65536), o intervalo máximo de tempo é de $T_{contagem} = \frac{1}{16 \times 10^6} \times 65.536 = 4,096ms$, logo para a contagem do intervalo de 1s será necessário o fator de 256, que possibilitará o intervalo máximo de $4,096ms \times 256 \cong 1,04s$. A configuração desse fator de *Prescaler* é feita através da ativação do bit CS12 no registrador TCCR1B. Como se deseja contar o intervalo preciso de 1s e não de 1,04s, é necessário a utilização de um valor de carga inicial, que pode ser calculado pela expressão: $\text{Carga} = MAX - \frac{T_{contagem} \times F_{clock}}{\text{Prescaler}} \rightarrow 65.536 - \frac{1 \times 16 \times 10^6}{256} \rightarrow \text{Carga} = 3.036$. Esse valor é então atribuído ao registrador TCNT1, que controla o valor atual de contagem.

Semelhantemente ao exemplo anterior, também utilizaremos uma rotina de interrupção para tratar o evento do término de contagem do intervalo programado, só que desta vez utilizando o vetor **TIMER1_OVF_vect** (LIMA, 2012), que sinaliza a função ISR que a mesma tratará o evento de estouro do temporizado TC1. Interessante notar que todo o código útil de ação do *software* é implementado dentro da rotina de interrupção, e que o laço infinito está vazio, simbolizando que o processador do microcontrolador está totalmente inerte, e só atuará quando o tempo configurado para TC1 for alcançado, isso demonstra a utilização otimizada da arquitetura.

Desenvolvimento de projetos com periféricos básicos

Você acredita que, com o conhecimento adquirido na implementação dos exemplos anteriores, seria possível a implementação de um projeto completo?

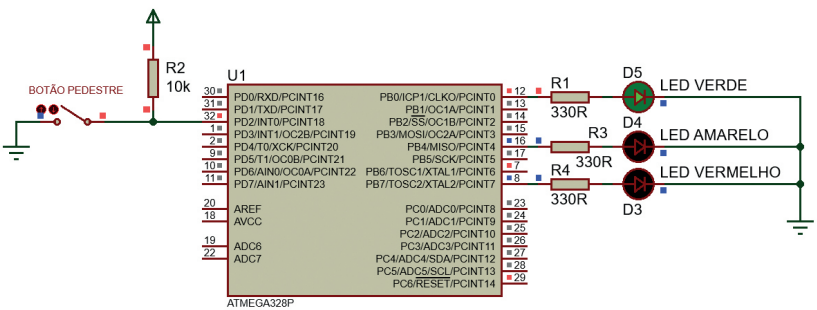
Muitos produtos reais que utilizamos cotidianamente foram desenvolvidos utilizando periféricos e técnicas simples como aquelas demonstradas nos exemplos anteriores.

Exemplificando

Exemplos de alguns produtos que operam simplesmente com operações de manipulação de IO's, temporizadores e interrupção: forno micro-ondas, despertador, máquina de lavar, controle remoto, dentre outros.

Dentre esses produtos, podemos destacar o semáforo urbano, que opera basicamente por meio do acionamento de três indicadores luminosos nas cores verde, amarelo e vermelho, em que cada qual possui um intervalo de acionamento determinado, e que pode ser alterado momentaneamente a partir da ação de um pedestre que aciona um botão no poste, desejando que o tempo em sinal vermelho seja estendido e em sinal verde diminuído, de forma a favorecer sua passagem. Podemos considerar como exemplo, o circuito apresentado na Figura 2.14, que representa um semáforo simples, com 3 leds e uma chave.

Figura 2.14 | Circuito de um semáforo simples



Fonte: captura de tela do *software* Proteus, elaborada pelo autor.

Nesse exemplo, é implementada a lógica para o funcionamento de um semáforo simples, em que os indicadores de cor verde, amarelo e vermelho são acionados por intervalos de 4s, 3s, e 2s respectivamente e sequencialmente e, quando o usuário pressiona a chave, os tempos dos indicadores verde e vermelho são invertidos, ou seja, 2s para o indicador verde e 4s para o vermelho, favorecendo dessa forma a passagem do pedestre.

Conforme mostrado no diagrama de *hardware,* os indicadores luminosos estão conectados aos pinos PB0, PB4 e PB7 logo, o registrador DDRB é utilizado para configurar em uma única operação todos esses pinos como saídas digitas, e os demais como entrada. Para a contagem dos intervalos de tempo, o temporizador TC1 é configurado semelhantemente ao exemplo anterior, para gerar eventos a cada 1s. Dessa vez, uma variável auxiliar chamada **contador** é incrementada a cada nova interrupção, e ela será utilizada como base de tempo para a alteração do acionamento dos indicadores, ou seja, sempre que o ocorre a mudança o valor de contagem é salvo em outra variável (**conta_anterior**) e o valor de contagem atual é continuamente comparado a esse valor anterior acrescido do tempo de espera para o indicador e, quando o tempo é atingido, a troca de indicadores é realizada.

A interrupção INT0 é agora configurada como borda de descida através da ativação do bit ISC01 no registrador EICRA. Quando uma interrupção ocorre, uma variável auxiliar chamada **flag_interrupção** é ativada, e indicará ao *software* que os tempos de contagem devem ser alterados no próximo ciclo do laço infinito. Confira a implementação completa na Figura 2.15.

Figura 2.15 | Código-fonte de implementação de um semáforo simples

```
#define F_CPU 16000000
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#define set_bit(reg,bit)  (reg |= (1 << bit))
#define rst_bit(reg,bit)  (reg &=~ (1 << bit))
#define le_bit(reg,bit)    ((reg >> bit) & 0x01)

volatile char flag_interrupcao = 0;
volatile unsigned int contador = 0;
unsigned int conta_anterior;
char cor_atual = 1,  t_verde = 4, t_verm = 2, t_amarelo = 3;
int main(void) {
  DDRB |= 0b10010001;    //PB0, PB4 e PB7 como saída
  set_bit(DDRB, PB7);    //PB7 como saída
  set_bit(EIMSK, INT0);  //Habilita INT0
  set_bit(EICRA, ISC01); //INT0 como borda de descida
  set_bit(TCCR1B, CS12); //Prescaler de 256
  set_bit(TIMSK1, TOIE1);//Interrupção do TC0 por Estouro
  TCNT1 = 3036;      //Carrega o Timer 1 para contar 1s
  sei();          //Habilita o bit de Interrupção Global
  while (1) {        //Laço infinito vazio
    if ((cor_atual == 1)
      && (contador >= (conta_anterior + t_verm))){
      PORTB = 0x01;    //Liga Led Verde
      cor_atual = 2;
      conta_anterior = contador;
      flag_interrupcao = 0;           }
    else if ((cor_atual == 2)
      && (contador >= (conta_anterior + t_verde))){
      PORTB = 0x10;    //Liga Led Amarelo
      cor_atual = 3;
      conta_anterior = contador;          }
    else if ((cor_atual == 3)
      && (contador >= (conta_anterior + t_amarelo)))  {
      PORTB = 0x80;    //Liga Led Vermelho
      cor_atual = 1;
      conta_anterior = contador;           }

    //Caso tenha ocorrido interrupção,
    //altera o tempo do semáforo
    if (flag_interrupcao == 1)  {
      t_verde = 2;
      t_verm = 4;        }
    else {
      t_verde = 4;
      t_verm = 2;        }
  }
}

ISR(INT0_vect){  //Interrupção por borda de descida
  flag_interrupcao = 1;  }

ISR(TIMER1_OVF_vect){
  contador++;
  TCNT1 = 3036;  }
```

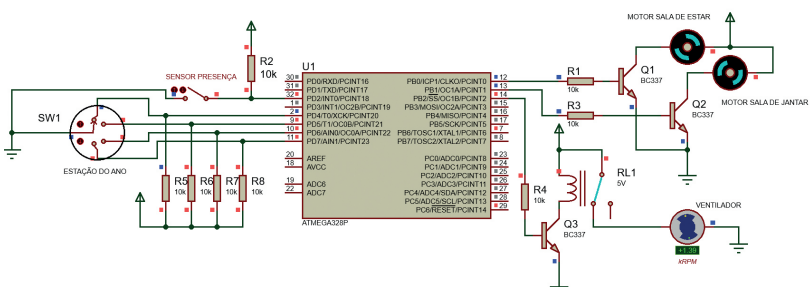
Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Sem medo de errar

Nesta seção, o seu novo gestor decidiu atribuir a você a tarefa de controlar automaticamente a abertura das persianas do produto. Relembrando que o sistema de controle contará com duas entradas digitais: a) um botão giratório com quatro opções para a seleção da estação do ano em que cada terminal do botão deve ser ligado a uma entrada digital do microcontrolador; e b) um sensor de presença instalado na entrada da residência que deve ser conectado a uma interrupção externa do microcontrolador. Estão previstas também três saídas digitais: a) uma saída para o acionamento do motor DC da persiana da sala de estar; b) outra saída para a o motor da persiana da sala de jantar. O tempo em que as saídas serão acionadas determinará qual a porção da persiana que será aberta; e c) uma saída ligada a um relê que será ligado a um ventilador presente na sala de estar que só será ligado em estações específicas do ano. O acionamento das saídas sempre deve ocorrer quando o sensor de presença for acionado, e a configuração para o acionamento será dependente da opção selecionada no botão de estações do ano, em que: a) Verão: persianas acionadas durante 5 segundos. Ventilador ligado; b) Inverno: persianas acionadas durante 1 segundo. Ventilador desligado; c) Primavera: persianas acionadas durante 3 segundos. Ventilador ligado; e d) Outono: persianas acionadas durante 2 segundos. Ventilador desligado.

Para facilitar a resolução do problema em questão, podemos iniciar pela construção de um *hardware* base para nos guiar no desenvolvimento do *software*, como visto na Figura 2.16.

Figura 2.16 | *Hardware* base para situação problema



Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Com exceção do sensor de presença, que deve ser posicionado no pino PD2 ou PD3 (Interrupções externas), a escolha dos demais pinos de IO é livre. O código-base que utilizaremos é semelhante aquele utilizado no exemplo do semáforo, sendo que podemos aproveitar as seguintes configurações: a) temporizador TC1 configurado para contagem de 1s; b) interrupção externa INT0 configurada como borda de descida; c) inclusão da biblioteca de interrupção e macros para manipulação de bits dos registadores.

Inicialmente, devemos configurar os pinos PB0, PB1 e PB2 como saídas, através do registrador DDRB e após, configurar a interrupção externa e o temporizador semelhantemente ao exemplo do semáforo, com o detalhe de não configurar *a priori* o fator de *Prescaler*, para que o temporizador inicie desativado. Como solicitado, os atuadores devem ligar assim que o sensor de presença ligado a interrupção externa for acionado, logo a primeira parte do código do laço infinito é monitorar a variável de interrupção e, caso a mesma esteja ativa, inicia-se o processo de acionamento. No *hardware* base, as entradas são ativas em nível lógico 0, logo a primeira parte do código após interrupção, é a leitura das entradas para encontrar qual das mesmas foi selecionada pelo usuário, e após configurar a variável de tempo de contagem de acordo com o especificado pelo problema, lembrando que cada posição da chave representa uma estação do ano.

Após a leitura da chave e seleção do tempo, os motores das persianas que estão ligadas nas saídas PB0 e PB1 são imediatamente acionadas, bem como o temporizador, que iniciará sua contagem e incrementará a variável de controle até que o tempo configurado seja atingido e os motores das persianas sejam desligados. O motor do ventilador é acionado por tempo indeterminado a depender da estação selecionada (verão ou primavera).

Uma possível solução de *software* é apresentada na Figura 2.17.

Figura 2.17 | Solução de *software* para a situação problema

```
#include <avr/io.h>
#include <avr/interrupt.h>
#define set_bit(reg,bit)  (reg |= (1 << bit))
#define rst_bit(reg,bit)  (reg &=~ (1 << bit))
#define le_bit(reg,bit)    ((reg >> bit) & 0x01)

volatile char flag_interrupcao = 0;
volatile unsigned int contador = 0;
unsigned int conta_tempo = 0;

int main(void) {
  DDRB |= 0b00000111;    //PB0, PB1 e PB2 como saída
  set_bit(EIMSK, INT0);  //Habilita INT0
  set_bit(EICRA, ISC01); //INT0 como borda de descida
  set_bit(TIMSK1, TOIE1);//Interrupção do TC0 por Estouro
  TCNT1 = 3036;      //Carrega o Timer 1 para contar 1s
  sei();          //Habilita o bit de Interrupção Global
  while (1) {        //Laço infinito
    if (flag_interrupcao == 1) { //Sensor de Presença ativo
      if (!le_bit(PIND, PD4)) {conta_tempo = 5;} //Verão
      if (!le_bit(PIND, PD5)) {conta_tempo = 1;} //Inverno
      if (!le_bit(PIND, PD6)) {conta_tempo = 3;} //Primavera
      if (!le_bit(PIND, PD7)) {conta_tempo = 2;} //Outono
      set_bit(TCCR1B, CS12); //Prescaler de 256 e liga Timer
      set_bit(PORTB, PB0);   //Ativa persiana sala de estar
      set_bit(PORTB, PB1);   //Ativa persiana sala de jantar
      //Se verão ou primavera, liga o ventilador
      if ((conta_tempo == 5) || (conta_tempo == 3)) {
        set_bit(PORTB, PB2);
      }
      if (contador >= conta_tempo) {
        rst_bit(PORTB, PB0);   //Desliga persiana sala de estar
        rst_bit(PORTB, PB1);   //Desliga persiana sala de jantar
        flag_interrupcao = 0;  //Limpa as variáveis
        contador = 0;
        rst_bit(TCCR1B, CS12); //Desliga o Timer
      }
    }
  }
}

ISR(INT0_vect){  //Interrupção por borda de descida
flag_interrupcao = 1;  }

ISR(TIMER1_OVF_vect){
  contador++;
TCNT1 = 3036;  }
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Avançando na prática

Um novo controle para o ventilador

Descrição da situação-problema

A sua proposta de *software* foi muito bem aceita pelo time e seu chefe pediu a você para fazer um incremento na lógica apresentada. O cliente notou que o ventilador não desliga após a sua entrada na residência, e gostaria que o mesmo ocorresse 10 segundos após sua passagem da sala de estar para o *hall* dos quartos, onde um outro sensor de presença foi instalado e será ligado a interrupção INT1.

Resolução da situação-problema

Para resolver esta nova situação-problema, temos que incluir a configuração da nova interrupção externa INT1, semelhante ao que fora feito com a interrupção INT0. Da mesma forma, devemos adicionar uma nova rotina ISR para tratar essa nova interrupção, e acionar uma variável de controle que será monitorada pelo ciclo do laço infinito. Logo após, devemos implementar uma nova instrução condicional que avalia a nova variável de interrupção que, caso acionada, aguarda que o contador de tempo atinja o valor 10 (10 segundos) para que o motor do ventilador seja desligado. A implementação completa dessa nova lógica pode ser analisada na Figura 2.18.

Figura 2.18 | Solução de *software* para a nova situação-problema

```
#include <avr/io.h>
#include <avr/interrupt.h>
#define set_bit(reg,bit)  (reg |= (1 << bit))
#define rst_bit(reg,bit)  (reg &=~ (1 << bit))
#define le_bit(reg,bit)    ((reg >> bit) & 0x01)
volatile char flag_interrupcao = 0, flag_interrupcao_2 = 0;
volatile unsigned int contador = 0;
unsigned int conta_tempo = 0;

int main(void) {
  DDRB |= 0b00000111;    //PB0, PB1 e PB2 como saída
  set_bit(EIMSK, INT0);  //Habilita INT0
  set_bit(EIMSK, INT1);  //Habilita INT1
  set_bit(EICRA, ISC01); //INT0 como borda de descida
  set_bit(EICRA, ISC11); //INT1 como borda de descida
  set_bit(TIMSK1, TOIE1);//Interrupção do TC0 por Estouro
  TCNT1 = 3036;      //Carrega o Timer 1 para contar 1s
  sei();          //Habilita o bit de Interrupção Global
  while (1) {        //Laço infinito
    if (flag_interrupcao == 1) { //Sensor de Presença ativo
      if (!le_bit(PIND, PD4)) {conta_tempo = 5;} //Verão
      if (!le_bit(PIND, PD5)) {conta_tempo = 1;} //Inverno
      if (!le_bit(PIND, PD6)) {conta_tempo = 3;} //Primavera
      if (!le_bit(PIND, PD7)) {conta_tempo = 2;} //Outono
      set_bit(TCCR1B, CS12); //Prescaler de 256 e liga Timer
      set_bit(PORTB, PB0);   //Ativa persiana sala de estar
      set_bit(PORTB, PB1);   //Ativa persiana sala de jantar
      //Se verão ou primavera, liga o ventilador
      if ((conta_tempo == 5) || (conta_tempo == 3)) {
        set_bit(PORTB, PB2);
      }
      if (contador >= conta_tempo) {
        rst_bit(PORTB, PB0);   //Desliga persiana sala de estar
        rst_bit(PORTB, PB1);   //Desliga persiana sala de jantar
        flag_interrupcao = 0;  //Limpa as variáveis
        contador = 0;
      }
    }
    if (flag_interrupcao_2 == 1) { //Sensor do Hall ativo
      if (contador >= 10) {    //Se passou 10 segundos
        rst_bit(PORTB, PB2); // Desliga o ventilador
        rst_bit(TCCR1B, CS12); //Desliga o Timer
      }
    }
  }
}

ISR(INT0_vect){  //Interrupção por borda de descida
flag_interrupcao = 1;  }

ISR(INT1_vect){  //Interrupção por borda de descida
flag_interrupcao_2 = 1;  }

ISR(TIMER1_OVF_vect){
  contador++;
TCNT1 = 3036;  }
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Faça valer a pena

1. Em algumas situações, quando se configura um pino de IO para desempenhar a função de entrada digital é necessário a utilização de resistores chamados de resistores de *pull-up* que devem ser ligados entre a potencial positivo (VCC) e a entrada do pino.

Qual a função dos resistores de *pull-up* em um circuito de entrada digital?

a) Limitar a corrente para o pino.

b) Aumentar a tensão de entrada.

c) Atenuar os efeitos de oscilação de sinal.

d) Proteger o microcontrolador contra surtos elétricos.

e) Configurar o pino como entrada digital.

2. Através da inclusão da biblioteca **interrupt.h** é possível a utilização das funções ISR (*Interrupt Service Routine* ou Rotinas de Serviço de Interrupção). Uma vez configurada uma nova fonte de interrupção, assim que o evento programado ocorre, a execução das instruções é interrompida, e o fluxo de *software* é direcionado para a função ISR correspondente.

A respeito das Rotinas de Serviço de Interrupção (ISR), considere as seguintes afirmações:

( ) São implementadas nativamente na linguagem C.

( ) São utilizadas somente para o tratamento de interrupções externas.

( ) Assim que um evento de interrupção ocorre, o microcontrolador aguarda o término do ciclo no laço infinito antes de passar o controle para a ISR.

( ) É utilizado um parâmetro de entrada que denota qual vetor de interrupção deve ser tratado pela ISR.

Considerando a sequência, qual das alternativas abaixo retrata corretamente quais afirmações são falsas e quais são verdadeiras?

a) F, F, F, V.

b) F, V, F, V.

c) F, V, F, F.

d) V, F, F, V.

e) V, F, V, V.

3. O temporizador TC1 é considerado o temporizador mais versátil implementado no microcontrolador ATMega328, por ser uma capacidade de 16 bits permite a contagem de valores até 65.536, sendo muito útil quando é necessário gerar intervalos longos na escala de segundos.

Dada uma aplicação do microcontrolador ATMega328 utilizando um *clock* de 16MHz, deseja-se realizar a contagem de um intervalo de 2s utilizando o temporizador TC1. Qual a melhor configuração do seu fator *Prescaler* e o valor inicial de carga para essa aplicação? Considere que o *Prescaler* pode assumir os valores de 1,8,64,256 ou1024.

a) *Prescaler* = 64, Carga = 60.500.

b) *Prescaler* = 1.024, Carga = 34.286.

c) *Prescaler* = 256, Carga = 60.500.

d) *Prescaler* = 1.024, Carga = 62.500.

e) *Prescaler* = 256, Carga = 34.286.

Seção 2.3

Projetos avançados com microcontroladores

Diálogo aberto

Caro aluno, os produtos eletrônicos mais avançados que conhecemos, como aqueles que controlam grandes aeronaves, ou que embarcam a maioria dos automóveis, são regidos por microcontroladores e alguns de seus periféricos de *hardware* mais avançados, que otimizados, podem realizar diversas funções paralelas, liberando o processador para funções críticas dentro do sistema.

Nesta seção, continuaremos nossos estudos práticos sobre os microcontroladores e a implementação de códigos funcionais para o controle de seus periféricos. Dessa vez, apresentaremos alguns exemplos de aplicações que utilizem os periféricos avançados, que foram estudados em sua forma teórica na Seção 3 da Unidade 1. Inicialmente, desenvolveremos um simples código para controle incremental da velocidade de um motor através de um sinal PWM, que controlaremos via *duty cycle*. Posteriormente, entenderemos como configurar o conversor Analógico Digital para a leitura de um sinal analógico simples, utilizado para o controle proporcional do motor do exemplo anterior. E, finalmente, aprenderemos a configurar o periférico USART para implementar um protocolo serial capaz de enviar e receber caracteres de qualquer terminal externo ao microcontrolador. Ao final dessas experiências, implementaremos um controlador de temperatura completo, com monitoramento, controle de saída PWM e leitura analógica de sensores, de forma a consolidar todo o conhecimento adquirido.

Lembre-se de que, nesta unidade, você faz parte de um importante time de desenvolvedores de *software* que está trabalhando arduamente para a entrega da primeira versão do mais novo produto de automação residencial da empresa. O seu primeiro pacote de *software* para o sistema foi totalmente aceito pelo time de desenvolvimento e já foi repassado para o time de validação

para a integração junto ao *hardware* do sistema. Nesse intervalo, o seu gestor solicitou a você que auxiliasse o time que foi incumbido de projetar as rotinas o controle avançado dos acionamentos. O objetivo do time é o projetar um sistema de controle que atenda o seguinte escopo:

- Um sensor de iluminação LDR que será conectado a uma das portas analógicas do microcontrolador. Com base nos valores lidos pelo sensor, as lâmpadas a led que estão ligadas as saídas deverão ser dimerizadas, em uma razão inversamente proporcional, ou seja, quanto maior a intensidade de luz externa, menor deve ser a intensidade da luz interna controlada pelas lâmpadas. Essa razão foi estipulada como sendo de 2 para 1.
- A velocidade do motor do ventilador deve ser regida pela leitura de temperatura realizar por uma das entradas analógicas conectada a um termopar instalado externamente a residência. A velocidade do motor deve ser diretamente proporcional a temperatura em razão de 3 para 1.

Dessa forma, foi solicitado a você que projetasse um pseudocódigo ou fluxograma que representasse o algoritmo para realizar a implementação desse escopo conforme descrito. Foi solicitado a você também, uma breve descrição de quais periféricos utilizar para esse controle e como configurá-los. Sabemos que, apesar de mais complexos, os periféricos mais avançados do microcontrolador, se bem configurados, podem desempenhar funções muito importantes em um projeto eletrônico. Quais desses periféricos você utilizaria para atender a essas funcionalidades? Qual a sequência correta para a construção desse programa? Bom estudo!

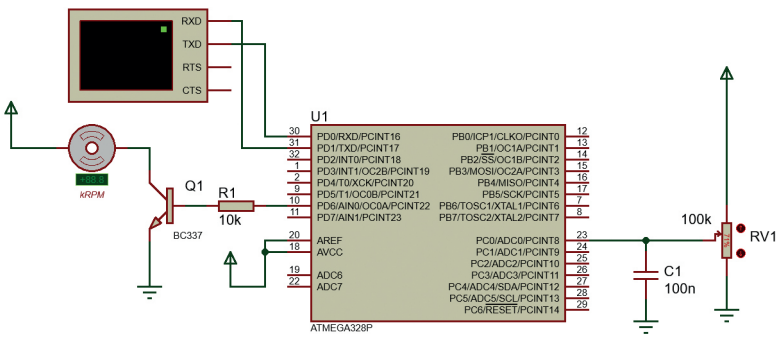
Não pode faltar

Olá!

Será que além das aplicações simples, seria possível projetar sistemas que realizem operações complexas com o microcontrolador, mas sem onerar o seu processamento?

Veremos, nessa seção, alguns exemplos de projetos envolvendo o microcontrolador ATMega328 e seus periféricos avançados, já estudados em sua teoria na Seção 3 da Unidade 1. Para isso, utilizaremos um circuito de *hardware* base conforme a Figura 2.19, que servirá de laboratório para todas as implementações de *software* que realizaremos a seguir.

Figura 2.19 | *Hardware* base para os exemplos práticos



Fonte: captura de tela do *software* Proteus, elaborada pelo autor.

Nesse circuito, temos um motor ligado ao pino PD6, multiplexado à saída OC0A, que será utilizada como pino de PWM para controle de velocidade do mesmo. Temos também um potenciômetro com capacitor para supressão de ruídos, que está ligado a entrada analógica ADC0, multiplexada ao pino PC0. E, por fim, é implementado um terminal serial virtual, conectado aos pinos RXD e TXD, que será utilizado para a imprimir caracteres provenientes da comunicação serial via periférico USART, bem como receber caracteres digitados pelo usuário e processá-los pelo mesmo periférico.

Controle de saídas com PWM (*Pulse Width Modulation* ou Modulação por largura de pulso)

Como é possível, por meio de um processador digital, controlar cargas de alta potência e de maneira proporcional e variada? O que devemos considerar quando configuramos um pino de I/O do microcontrolador como PWM?

Uma das funções mais utilizadas em sistemas de controle industrial é o PWM, ou modulação por largura de pulso, que permite ao controlador gerar pulsos digitais com ciclo ativo (*duty cycle*) configurável, de forma que a tensão média na saída de controle seja proporcional ao intervalo deste ciclo em relação ao intervalo total do pulso (LIMA, 2012).

Pesquise mais

O que acha de conhecer mais sobre o funcionamento da técnica de PWM? Acesse *Domine o PWM*. Disponível em: <https://www.embarcados.com.br/domine-o-pwm>. Acesso: 20 maio 2018. Lá, você encontrará um artigo sobre o tema, bem como um *link* para uma videoaula.

Como primeiro exemplo, controlaremos a velocidade de um motor DC (*Direct Current* ou Corrente Contínua) através do periférico Temporizador 0 e sua função de geração de ondas por PWM. Como vimos na seção anterior, um pino de I/O possui limitações quanto a corrente máxima de saída que o mesmo pode fornecer logo, quando se deseja conectá-lo a uma carga que demande uma corrente superior, é necessário utilizar um componente complementar de potência, como é o caso do transistor apresentado na Figura 2.20.

Assimile

Uma das principais características de um motor DC é que a velocidade angular de seu rotor é diretamente proporcional a tensão média de alimentação do motor, porém, é sempre recomendável conhecer seus limites construtivos para não o danificar devido a valores de sobretensão.

O algoritmo de controle para esse exemplo é apresentado na Figura 2.20. Inicialmente, configura-se o pino PD6 como saída

digital, uma vez que o mesmo será utilizado como pino PWM. Logo após, é feita a configuração do fator de *Prescaler* do Temporizador como 8 que, considerando uma frequência de *clock* padrão de 16MHz, gerará uma forma de onda quadrática com frequência de aproximandamente 3,9 KHz, conforme a fórmula apresentada na Seção 3 da Unidade 1: $F_{PWM} = \frac{F_{clock}}{512 \times N} = \frac{16 \times 10^6}{512 \times 8} \Rightarrow F_{PWM} \cong 3{,}9Hz$.

Figura 2.20 | *Software* para controle via PWM de um motor DC

```
#define F_CPU 16000000
#include <util/delay.h>
#include <avr/io.h>
#define set_bit(Reg,bit)    (Reg = Reg | (1<<bit))
#define rst_bit(Reg,bit)    (Reg = Reg & ~(1<<bit))

uint32_t tick, atualizacao;
int main(void) {
    set_bit(DDRD, PD6);     //Configura o pino PD6 como saída
    set_bit(TCCR0B, CS01);  //Configura a frequência do PWM como sendo ~3,9KHz, com um prescaler = 8
    //Configura o temporizador como PWM rápido com dutycycle positivo no Motor
    TCCR0A |= (1<<WGM00) | (1<<WGM01) | (1<<COM0A1);
    OCR0A = 25; //Motor inicia DutyCycle com 10% ( ou seja 25 de 255)
    while (1) {
        //Incrementa o DutyCycle de PWM do motor a cada 10ms
        if (tick >= atualizacao + 10) {
            atualizacao = tick;
            OCR0A++;
        }
        _delay_ms(10);
        tick++;
    }
}
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

A próxima etapa consiste em configurar o próprio Temporizador como um gerador PWM, através da ativação dos bits **WGM00** e **WGM01** do registrador **TCCR0A,** e também a configuração de *duty cycle* positivo através da ativação do bit **COM0A1** do mesmo registrador, que possibilitará o incremento proporcional do *duty cycle* em função do aumento do valor do registrador de controle **OCR0A** através da seguinte relação: $DutyCycle = \frac{OCR0A}{255} \times 100 \ [\%]$. Isto feito, inicia-se o PWM com *duty cycle* em aproximadamente 10% (OCR0A igual a 25), e entra no laço infinito do programa, em que o valor de *duty cycle* é incrementado unitariamente a cada 10ms e, ao atingir o valor máximo de 100% (OCR0A igual a 255), automaticamente retorna a zero pela ocorrência de estouro no registrador.

Leitura de sinais analógicos

Qual a melhor maneira de se configurar um periférico de conversão analógico-digital para realizar leituras que sejam rápidas e confiáveis?

Nesse segundo exemplo, veremos como utilizar este periférico no microcontrolador ATMega328, de forma simples e efetiva.

O algoritmo para este próximo exemplo ainda utilizará o controle PWM do motor DC, só que, desta vez, a velocidade do motor não será mais incrementada progressivamente como no exemplo anterior, e sim controlada pelo valor lido no canal analógico, que está diretamente conectado a um potenciômetro, cuja tensão de saída pode variar entre 0V e 5V. A Figura 2.21 apresenta o código-fonte para esta nova implementação.

Figura 2.21 | *Software* para controle de velocidade do motor DC via leitura do canal analógico

```
#include <avr/io.h>
#define set_bit(Reg,bit) (Reg|=(1<<bit))
#define rst_bit(Reg,bit) (Reg&=~(1<<bit))
#define le_bit(Reg,bit) ((Reg>>bit)&0x01)

int main() {
    set_bit(ADMUX, REFS0);  // Seleciona a Tensão de Referência como AVCC
    ADCSRA |= (1<<ADPS0) | (1<<ADPS1) | (1<<ADPS2); //Prescaler de 128
    set_bit(ADCSRA, ADEN);  // Habilita o Canal Analógico
    set_bit(DDRD, PD6);     //Configura o pino PD6 como saída
    //Configura a frequência do PWM como sendo ~3,9KHz, com um prescaler = 8
    set_bit(TCCR0B, CS01);
    //ConfigurA o timer como PWM rápido com dutycycle positivo no Motor
    TCCR0A |= (1<<WGM00) | (1<<WGM01) | (1<<COM0A1);
    ADMUX &= 0xF0;  //Seleciona o Canal 0 do conversor
    while(1) {
        set_bit(ADCSRA,ADSC);   //Conversão em Modo Simples
        while(le_bit(ADCSRA,ADSC)); // Espera até a conversão estar completa
        //Após conversão completa, usa os 8 bits mais significativos para controlar o motor
        OCR0A = ADC >> 2;
    }
}
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Como fora visto na Seção 3 da unidade anterior, o primeiro passo para a configuração de um conversor analógico-digital é a definição de sua fonte de tensão de referência, que, no caso do microcontrolador ATMega328, pode ser a tensão de alimentação analógica, um potencial de referência externa, ou 1,1V regulado internamente pelo próprio microcontrolador. Para esse exemplo, utilizaremos a própria alimentação do circuito analógico que, como pode ser visto no diagrama de *hardware*, está conectado ao potencial VCC. Essa configuração é possível ativando o bit **REFS0** do registador **ADMUX.** O próximo passo é definição do fator de *Prescaler* do temporizador que controla o tempo de captura da amostra de conversão, em que valores de intervalos maiores representam menor taxa de amostragem, porém maior confiabilidade no valor da amostra, e valores de intervalo menor

representam as características opostas. Iremos adotar neste caso o maior fator de *Prescaler*, que nos garantirá maior confiabilidade no valor lido, mesmo que isso sacrifique a taxa de amostragem. Fazemos isso ao ativar os bits **ADPS0**, **ADPS1** e **ADPS2** do registrador **ADCSRA.**

Reflita

Quando optamos por maiores taxas de amostragem, estamos abdicando da qualidade de cada amostra coleta. É possível diminuir a incerteza desses valores através de algum método estatístico?

Por fim, habilitamos o conversor através do bit **ADEN** do mesmo registrador e, como o pino utilizado para a leitura é o **PC0**, que representa o canal 0 do conversor, devemos selecioná-lo através do registrador **ADMUX**. A configuração para o controle PWM do motor segue inalterada, em relação ao exemplo anterior.

O fluxo do laço infinito segue uma rotina simples:

- Início da leitura do canal através da ativação do bit ADSC.
- Aguarda-se o termino da leitura, representado pela queda do sinal do mesmo bit.
- Utiliza-se o valor convertido para carregar o registrado OCR0A, que controla o *duty cycle* de PWM do motor.

Vale ressaltar que o valor de conversão do registrador **ADC** possui um tamanho de **10 bits**, por padrão. Para realizar a atribuição desse valor ao registrador OCR0A de 8 bits, foi feita uma operação de deslocamento a direita em 2 bits, para que os 8 bits mais significativos sejam atribuídos ao *duty cycle*.

Comunicação serial

Como podemos tornar a implementação de um *software* para comunicação serial algo intuitivo e de fácil depuração? Nesse exemplo, implementaremos algumas funções para manipulação do periférico **USART**, de forma que possamos transmitir e receber caracteres individuais com relativa facilidade (GIMENEZ, 2002).

O objetivo dessa demonstração é simplesmente aguardar o envio de um caractere via terminal pelo usuário, caso o caractere seja um numeral, vamos incrementá-lo e devolvê-lo pelo mesmo canal serial e, caso ele não seja numérico, simplesmente replicamos na saída o mesmo valor digitado. O código-fonte é apresentado na Figura 2.22.

Figura 2.22 | *Software* de implementação de um protocolo serial

```
#include <avr/io.h>
#define TAXA_TRANS  9600
#define MYUBRR (F_CPU/16/TAXA_TRANS-1)
unsigned char USART_le(void);
void USART_escreve(unsigned char caracter);
void USART_config(unsigned int usart);
unsigned char dado_recebido, dado_envia;

int main(){
    USART_config(MYUBRR);
    while(1) {
        //Aguarda a leitura
        dado_recebido = USART_le();
        //Se for um caracter numeral incrementa
        //e retorna o valor
        if ((dado_recebido >= 0x30) &&
                (dado_recebido <= 0x38)){
            dado_envia = dado_recebido + 1;
            USART_escreve(dado_envia);
        }
        //Se não for numeral simplemente retorna
        //o valor
        else {USART_escreve(dado_recebido);}
    }
}
void USART_config(unsigned int usart){
    //Ajusta a taxa de transmissão
    UBRR0H = (unsigned char)(MYUBRR>>8);
    UBRR0L = (unsigned char)MYUBRR;
    //Habilita os pinos de TXD e RXD
    UCSR0B = (1<<RXEN0)|(1<<TXEN0);
    //Modo assíncrono, 8 bits de dados, sem paridade
    UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);}
unsigned char USART_le(void){
    //Aguarda o dado ser recebido e retorna o valor
    while (!(UCSR0A & (1<<RXC0)));
    return UDR0;}
void USART_escreve(unsigned char caracter){
    //Aguarda o dado ser enviado
    while (!( UCSR0A & (1<<UDRE0)) );
    UDR0 = caracter;}
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Como podemos identificar que um caractere é de natureza numeral? Os caracteres são tratados pela linguagem C através de uma tabela chamada **ASCII** (do inglês *American Standard Code for Information Interchange* ou Código Padrão Americano para o Intercâmbio de Informação), que converte cada caractere em um valor numérico correspondente. No caso dos numerais, o valor representativo inicia-se em 0x30 para o caractere '0' e termina em 0x39 para o caractere '9'.

Exemplificando

É necessário implementar a comunicação entre um programa supervisório que é executado em um computador pessoal, e um circuito eletrônico controlador por um microcontrolador, para o envio do valor de temperatura, mensurado pelo segundo. Para isso, um projetista elabora um protocolo simples utilizando a tabela ASCII, composto de 4 bytes: um byte com o caractere 'I' representando o início da comunicação, outro byte com o caractere 'E', que representa o final da comunicação, e 2 bytes centrais que conterão o valor da temperatura, entre 0 e 99 °C. Qual seria o pacote de envio em hexadecimal, caso a temperatura atual fosse de 45 °C?

Em termos de caracteres, deve-se ser enviado a seguinte sequência: "I45E". Convertendo os valores segundo a tabela ASCII teríamos: 0x49 ('I'), 0x34 ('4'), 0x35 ('5') e 0x45 ('E').

Inicialmente no código, definimos através de uma macro qual será o valor de taxa de transmissão do nosso protocolo, que nesse caso é 9600 bps (bits por segundo). Esse valor é utilizado para carregar o registrador **UBRR0,** responsável por configurar a taxa de *baud rate* no periférico USART. Feito isso, configuramos o periférico para trabalhar de maneira assíncrona, ou seja, sem depender de um sinal de *clock* externo, com tamanho de pacote de 8 bits (1 caractere) e sem bit de paridade.

Como pode ser visto no código-fonte, dentro do laço infinito aguarda-se que o usuário digite algum caractere no terminal do simulador, e quando o mesmo o faz, o valor é comparado com o possível intervalo entre os caracteres '0' e '8' (0x30 e 0x38 respectivamente). Se for um numeral, o valor é incrementado e enviado de volta ao terminal e, caso não seja, o mesmo valor que fora digitado ao terminal é retransmitido ao usuário.

Projetos avançados com microcontroladores

Chegamos agora ao ápice do nosso aprendizado sobre microcontroladores. Você acredita que com o conhecimento adquirido, seria capaz de implementar um sistema avançado com os periféricos que vimos até então?

Nesse exemplo final, concretizaremos todo o conteúdo das seções anteriores em um sistema que englobe os principais periféricos do microcontrolador. Será implementado um controlador digital de temperatura, cujo sensor será simulado pelo potenciômetro do *hardware* base, e a ventoinha de controle será representada pelo motor DC. A velocidade da ventoinha é controlada por um sinal PWM diretamente vinculado a leitura do sinal analógico do sensor, e cujo valor é diretamente proporcional a este, em uma razão de 4 para 1, ou seja, um valor de leitura igual a 400 no canal analógico representa um valor de 100 no registrador de controle de PWM do motor. Além desse aspecto de controle, um monitor serial também é implementado utilizando-se o periférico USART, que enviará em intervalos de 1 segundo mensagens contendo o valor atual do sensor de temperatura e também da velocidade atual do motor. Como se deseja trabalhar com sequência de caracteres (*strings*), ao invés de caracteres individuais, é configurada uma importante função na linguagem C para tal fim: o ***printf***, implementado pela biblioteca *stdio.h*.

Para que esta função funcione de maneira correta, além da inclusão da biblioteca previamente citada, é necessário vincular a mesma a um método que execute a saída dos caracteres da função, e esse método é representado pela função **usart_putchar_printf.** De forma a criar um vínculo dentre ambas as funções, é feita a declaração de uma variável do tipo ***FILE*** (arquivo) que define como a função será utilizada.

Como pode ser visto no código-fonte completo representado pela Figura 2.23, iniciamos pela inclusão da biblioteca *stdio.h*, e pela declaração das funções que utilizaremos no código, e após, declaramos a variável FILE. Dentro da função principal, configura-se os periféricos temporizador TC0 e conversor analógico-digital da mesma maneira que fora feito nos exemplos anteriores, com a exceção que o processo de conversão AD propriamente dito, agora foi encapsulado dentro da função **LeituraADC** para garantir melhor organização e legibilidade do código.

Já dentro do fluxo do laço infinito, a cada interação, o valor de *duty cycle* é atualizado através da leitura do sensor via canal AD e dividido pela razão de 4, conforme definido anteriormente. Logo após, utilizamos a função **printf** para imprimir os valores atuais de Temperatura do sensor e Velocidade do motor da ventoinha. E, por fim, utilizamos a função de atraso **_delay_ms** para atrasar a execução do código em 1 segundo, e garantir o monitoramento pelo usuário.

Figura 2.23 | Código-fonte do controlador de temperatura digital

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#define set_bit(Reg,bit)    (Reg = Reg | (1<<bit))
#define le_bit(Reg,bit)     ((Reg>>bit)&0x01)
#define TAXA_TRANS  19200   //Taxa de Transmissão
#define MYUBRR (F_CPU/16/TAXA_TRANS-1)
unsigned char USART_le(void);
void USART_escreve(unsigned char caracter);
void USART_config(unsigned int usart);
unsigned char dado_recebido, dado_envia, cont, set_vel, i;
int usart_putchar_printf(char var, FILE *stream);
unsigned int LeituraADC(void);
static FILE mystdout =
FDEV_SETUP_STREAM(usart_putchar_printf, NULL, _FDEV_SETUP_WRITE);
int main(){
    stdout = &mystdout;
    USART_config(MYUBRR);
    set_bit(DDRD, PD6);     //Configura o pino PD6 como saída
    set_bit(TCCR0B, CS01);  //Frequência do PWM como sendo ~3,9KHz
    //Configura o timer como PWM rápido com dutycycle positivo no Motor
    TCCR0A |= (1<<WGM00) | (1<<WGM01) | (1<<COM0A1);
    set_bit(ADMUX, REFS0);  // Seleciona a Tensão de Referência como AVCC
    ADCSRA |= (1<<ADPS0) | (1<<ADPS1) | (1<<ADPS2); //Prescaler de 128
    set_bit(ADCSRA, ADEN);  // Habilita o Canal Analógico
    ADMUX &= 0xF0;  //Seleciona o Canal 0 do conversor
    while(1) {
        OCR0A = LeituraADC() / 4;   //Duty cycle do motor é 1/4 do ADC
        //Imprime os valores atuais do sensor e da velocidade
        printf("Temperatura Atual: %d ºC\r\n", LeituraADC() / 10);
        printf("Velocidade Atual: %d ºC\r\n", OCR0A);
        _delay_ms(1000); //Espera 1 segundo
    }
}
```

```
void USART_config(unsigned int usart) {
    UBRR0H = (unsigned char)(MYUBRR>>8); //Ajusta a taxa de transmissão
    UBRR0L = (unsigned char)MYUBRR;
    UCSR0B = (1<<RXEN0)|(1<<TXEN0); //Habilita os pinos de TXD e RXD
    UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);}
void USART_escreve(unsigned char caracter){
    while (!( UCSR0A & (1<<UDRE0)) );//Aguarda o dado ser enviado
    UDR0 = caracter;}
int usart_putchar_printf(char var, FILE *stream) {
    if (var == '\n') USART_escreve('\r');
    USART_escreve(var);
    return 0;}
unsigned int LeituraADC(void) {
    set_bit(ADCSRA,ADSC);   //Conversão em Modo Simples
    while(le_bit(ADCSRA,ADSC)); // Espera conversão completar
    return ADC;}    // Retorna o valor da conversão
```

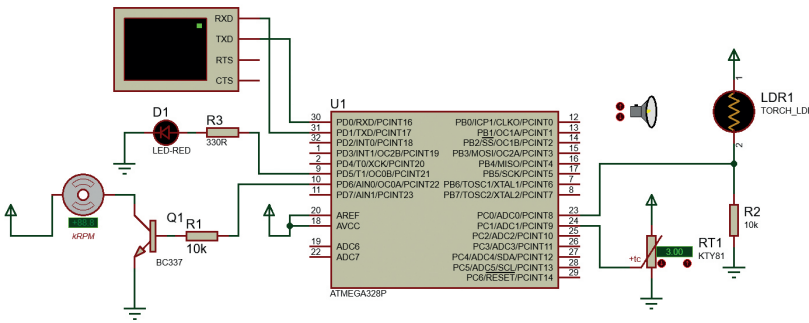
Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Sem medo de errar

Nesta seção, o seu gestor solicitou a você que auxiliasse o time que foi incumbido de projetar as rotinas o controle avançado dos acionamentos. O objetivo do time é o projetar um sistema de controle que atenda o seguinte escopo: a) Um sensor de iluminação LDR será conectado a uma das portas analógicas do microcontrolador que, com base nos valores lidos, as lâmpadas a led que estão ligadas as saídas deverão ser dimerizadas, em uma razão inversamente proporcional de 2 para 1 e; b) A velocidade do motor do ventilador deve ser regida pela leitura de temperatura realizar por uma das entradas analógicas conectada a um termopar instalado externamente a residência e a velocidade do motor deve ser diretamente proporcional a temperatura mensurada, em uma razão de 3 para 1. Dessa forma, foi solicitado a você que projetasse um pseudocódigo ou fluxograma que representasse o algoritmo para realizar a implementação desse escopo conforme descrito. Foi pedido também, uma breve descrição de quais periféricos utilizar para esse controle e como configurá-los.

Para facilitar a resolução do problema em questão, podemos iniciar pela construção de um *hardware* base para nos guiar no desenvolvimento do *software*, como visto na Figura 2.24.

Figura 2.24 | *Hardware* base para situação problema



Fonte: captura de tela do *software* Proteus, elaborada pelo autor.

O primeiro passo para a resolução da situação-problema é encontrar as equações que relacionem as leituras dos sensores analógicos ao valor de *duty cycle* das cargas a serem controladas, de acordo com as razões exigidas pelo escopo. Para normatizar os valores lidos nas entradas, podemos convertê-los de 0 a 1023 (intervalo de leitura do canal AD) para um intervalo de 0 a 100 (representado um valor percentual de 0% a 100%), para isso basta dividirmos o valor lido por 10, que obteremos um valor aproximado e percentual. Como sabemos que os registradores que controlam o *duty cycle* de PWM operam entre os valores de 0 a 255, podemos estabelecer a seguinte relação entre os valores de entrada e saída:

Tabela 2.1 | Relação entre leitura do canal AD e saída PWM para a) Led e b) Ventoinha

Leitura AD (%)	Saída PWM
0	510 (2x 255)
100	0

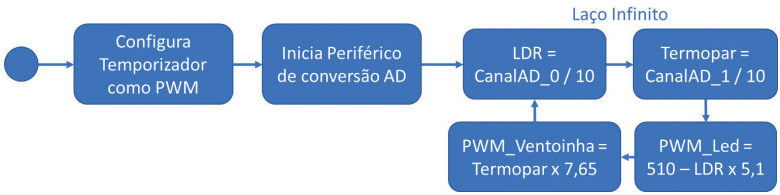
Leitura AD (%)	Saída PWM
0	0
100	765

Fonte: elaborada pelo autor.

Podemos agora calcular a relação entre os sinais de entrada do sensor LDR e o valor de *duty cycle* das lâmpadas led como sendo: $DutyCycle = 510 - 5{,}1 \times LDR$ e a relação do sensor termopar e a ventoinha como sendo: $DutyCycle = 7{,}65 \times Termopar$. Vale ressaltar que os valores devem ser limitados ao valor máximo do registrador, que é de 255.

Um possível fluxograma para a solução do problema é apresentado na Figura 2.25.

Figura 2.25 | Fluxograma de implementação da solução



Fonte: elaborada pelo autor.

Apesar de as escolhas dos canais de conversão analógico-digital, bem como das saídas de sinal PWM serem livres, uma possível implementação do código-fonte, contendo todas as configurações dos periféricos para esta implementação é apresentada na Figura 2.26.

Figura 2.26 | Código-fonte para implementação da questão

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#define set_bit(Reg,bit)    (Reg = Reg | (1<<bit))
#define le_bit(Reg,bit)     ((Reg>>bit)&0x01)
unsigned int ldr, ptc, pwm_led, pwm_motor;
unsigned int LeituraADC(unsigned char canal);

int main(){
    set_bit(DDRD, PD6);     //Configura o pino PD6 como saída para Motor
    set_bit(DDRD, PD5);     //Configura o pino PD5 como saída para Led
    set_bit(TCCR0B, CS01);  //Configura a frequência do PWM como sendo ~7,8KHz
    //Configura o timer como PWM rápido com dutycycle positivo no Motor e no Led
    TCCR0A |= (1<<WGM00) | (1<<WGM01) | (1<<COM0A1) | (1<<COM0B1);
    set_bit(ADMUX, REFS0);  // Seleciona a Tensão de Referência como AVCC
    ADCSRA |= (1<<ADPS0) | (1<<ADPS1) | (1<<ADPS2); //Prescaler de 128
    set_bit(ADCSRA, ADEN);  // Habilita o Canal Analógico
    ADMUX &= 0xF0;  //Seleciona o Canal 0 do conversor
    while(1) {
        ldr = LeituraADC(0) / 10; //Realiza a leitura do canal analógico 0 em % (LDR)
        ptc = LeituraADC(1) / 10; //Realiza a leitura do canal analógico 1 em % (PTC)
        pwm_led = 510 - ((51 * ldr) / 10); //Equação para cálculo do duty cycle do led
        pwm_led > 255 ? (OCR0B = 255) : (OCR0B = (unsigned char)pwm_led); //Limita a 255
        pwm_motor = (765 * ldr) / 100; //Equação para cálculo do duty cycle do motor
        pwm_motor > 255 ? (OCR0A = 255) : (OCR0A = (unsigned char)pwm_motor); //Limita a 255
    }
}
unsigned int LeituraADC(unsigned char canal) {
    ADMUX = (ADMUX & 0xF0) | (canal & 0x0F); //Seleção do Canal
    set_bit(ADCSRA,ADSC);   //Conversão em Modo Simples
    while(le_bit(ADCSRA,ADSC)); // Espera conversão completar
return ADC;}    // Retorna o valor da conversão
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Faça valer a pena

Monitoramento do sistema de controle

Descrição da situação-problema

Apesar da sua solução proposta para a última atividade ter sido implementada com sucesso, seus colegas desenvolvedores pediram a você para implementar alguma ferramenta que os auxiliasse no monitoramento do sistema de controle e suas variáveis. Logo veio em mente a implementação de um canal serial que pudesse ser utilizado para imprimir as principais variáveis do sistema via terminal. Configure e implemente o periférico USART para imprimir as leituras do canal analógico, bem como os valores de PWM dos atuadores.

Resolução da situação-problema

Podemos utilizar o código-exemplo descrito no texto para a implementação do código, utilizando inclusive a configuração para a função **printf.** Segue abaixo na Figura 2.27 uma possível implementação para esse novo desafio.

Figura 2.27 | Código-fonte para o monitoramento das variáveis de controle

```
#include <avr/io.h>
#include <stdio.h>
#define set_bit(Reg,bit)   (Reg = Reg | (1<<bit))
#define le_bit(Reg,bit)    ((Reg>>bit)&0x01)
unsigned int ldr, ptc, pwm_led, pwm_motor;
#define TAXA_TRANS  19200  //Taxa de Transmissão
#define MYUBRR (F_CPU/16/TAXA_TRANS-1)
void USART_escreve(unsigned char caracter);
void USART_config(unsigned int usart);
unsigned char dado_recebido, dado_envia, cont, set_vel, i;
int usart_putchar_printf(char var, FILE *stream);
static FILE mystdout =
FDEV_SETUP_STREAM(usart_putchar_printf, NULL, _FDEV_SETUP_WRITE);
unsigned int LeituraADC(unsigned char canal);
int main(){
  stdout = &mystdout;
  USART_config(MYUBRR);
  set_bit(DDRD, PD6);    //Configura o pino PD6 como saída para Motor
  set_bit(DDRD, PD5);    //Configura o pino PD5 como saída para Led
  set_bit(TCCR0B, CS01);  //Configura a frequência do PWM a ~3,9KHz
  //Configura o PWM rápido  com dutycycle positivo no Motor e no Led
  TCCR0A |= (1<<WGM00) | (1<<WGM01) | (1<<COM0A1) | (1<<COM0B1);
  set_bit(ADMUX, REFS0);  // Seleciona a Tensão de Referência como AVCC
  ADCSRA |= (1<<ADPS0) | (1<<ADPS1) | (1<<ADPS2); //Prescaler de 128
```

```
#include <avr/io.h>
#include <stdio.h>
#define set_bit(Reg,bit)   (Reg = Reg | (1<<bit))
#define le_bit(Reg,bit)    ((Reg>>bit)&0x01)
unsigned int ldr, ptc, pwm_led, pwm_motor;
#define TAXA_TRANS  19200  //Taxa de Transmissão
#define MYUBRR (F_CPU/16/TAXA_TRANS-1)
void USART_escreve(unsigned char caracter);
void USART_config(unsigned int usart);
unsigned char dado_recebido, dado_envia, cont, set_vel, i;
int usart_putchar_printf(char var, FILE *stream);
static FILE mystdout =
FDEV_SETUP_STREAM(usart_putchar_printf, NULL, _FDEV_SETUP_WRITE);
unsigned int LeituraADC(unsigned char canal);
int main(){
  stdout = &mystdout;
  USART_config(MYUBRR);
  set_bit(DDRD, PD6);    //Configura o pino PD6 como saída para Motor
  set_bit(DDRD, PD5);    //Configura o pino PD5 como saída para Led
  set_bit(TCCR0B, CS01);  //Configura a frequência do PWM a ~3,9KHz
  //Configura o PWM rápido  com dutycycle positivo no Motor e no Led
  TCCR0A |= (1<<WGM00) | (1<<WGM01) | (1<<COM0A1) | (1<<COM0B1);
  set_bit(ADMUX, REFS0);  // Seleciona a Tensão de Referência como AVCC
  ADCSRA |= (1<<ADPS0) | (1<<ADPS1) | (1<<ADPS2); //Prescaler de 128
  set_bit(ADCSRA, ADEN);  // Habilita o Canal Analógico
  ADMUX &= 0xF0;  //Seleciona o Canal 0 do conversor
  while(1) {
    ldr = LeituraADC(0) / 10; //leitura do canal analógico 0 em % (LDR)
    ptc = LeituraADC(1) / 10; //leitura do canal analógico 1 em % (PTC)
    pwm_led = 510 - ((51 * ldr) / 10); //cálculo do duty cycle do led
    pwm_led > 255 ? (OCR0B = 255) : (OCR0B = (unsigned char)pwm_led);
    pwm_motor = (765 * ldr) / 100; //cálculo do duty cycle do motor
    pwm_motor > 255 ? (OCR0A = 255) : (OCR0A = (unsigned char)pwm_motor);
    //Imprime os valores de Temperatura, nível de luminosidade via LDR,
    //Duty Cycle dos Leds e Duty Cycle do Ventilador
    printf("Monitoramento: Temp: %d C LDR: %d Led: %d Ventilador: %d\r\n",
      ptc, ldr, OCR0B, OCR0A);
  }
}
unsigned int LeituraADC(unsigned char canal) {
  ADMUX = (ADMUX & 0xF0) | (canal & 0x0F); //Seleção do Canal
  set_bit(ADCSRA,ADSC);  //Conversão em Modo Simples
  while(le_bit(ADCSRA,ADSC)); // Espera conversão completar
return ADC;}  // Retorna o valor da conversão
void USART_config(unsigned int usart) {
  UBRR0H = (unsigned char)(MYUBRR>>8); //Ajusta a taxa de transmissão
  UBRR0L = (unsigned char)MYUBRR;
  UCSR0B = (1<<RXEN0)|(1<<TXEN0); //Habilita os pinos de TXD e RXD
UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);} //8 bits de dados, sem paridade
void USART_escreve(unsigned char caracter){
  while (!( UCSR0A & (1<<UDRE0)) );//Aguarda o dado ser enviado
UDR0 = caracter;}
int usart_putchar_printf(char var, FILE *stream) {
  if (var == '\n') USART_escreve('\r');
  USART_escreve(var);
return 0;}
```

Fonte: captura de tela do *software* ATMEL Studio, elaborada pelo autor.

Avançando na prática

1. Uma das funções mais utilizadas em sistemas de controle industrial é o PWM, ou modulação por largura de pulso, que permite ao controlador gerar pulsos digitais com ciclo ativo (*duty cycle*) configurável, de forma que a tensão média na saída de controle será proporcional ao intervalo deste ciclo em relação ao intervalo total do pulso.

Qual dos registradores abaixo controle efetivamente o valor de *duty cycle* do Temporizador TC0?

bits **WGM00** e **WGM01** do registrador **TCCR0A,** e também a configuração de *duty cycle* positivo através da ativação do bit **COM0A1** do mesmo registrador, que possibilitará o incremento proporcional do *duty cycle* em função do aumento do valor do registrador de controle **OCR0A** através da seguinte relação: $DutyCycle = \frac{OCR0A}{255} \times 100 \quad [\%]$. Isto feito, inicia-se o

a) TCCR0A.

b) OCR0A.

c) WGM00.

d) COM0A1.

e) COM0B1.

2. O fator de *Prescaler* do temporizador do periférico de conversão analógica controla o tempo de captura da amostra de conversão, valores de intervalos maiores representam menor taxa de amostragem, porém maior confiabilidade no valor da amostra, e valores de intervalo menor representam as características opostas.

Dada uma aplicação na qual se deseja realizar a leitura de um sensor de temperatura a cada 1 segundo para o controle de um forno elétrico. Qual a configuração mais recomendada para o tempo de captura de amostra do conversor?

a) Alta taxa de amostragem, devido a criticidade da aplicação.

b) Alta taxa de amostragem, devido a característica do sensor.

c) Baixa taxa de amostragem, dado o intervalo de tempo entre capturas.

d) Baixa taxa de amostragem, devido a velocidade do microcontrolador.

e) Alta taxa de amostragem, devido a confiabilidade da leitura.

3. Os caracteres são tratados pela linguagem C através de uma tabela chamada ASCII (do inglês *American Standard Code for Information Interchange* ou Código Padrão Americano para o Intercâmbio de Informação), que converte cada caractere em um valor numérico correspondente.

Qual seria o valor decimal correspondente ao caractere numeral '6' segundo a tabela ASCII?

a) 6.

b) 0x06.

c) 0x36.

d) 54.

e) 56.

Referências

Atmel Studio 7. Disponível em <http://www.microchip.com/avr-support/atmel-studio-7>. Acesso: 26 maio 2018.

ATmega328 Datasheet Complete. Página 385. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso: 13 maio 2018.

Bits em Linguagem C. Disponível em: <https://www.embarcados.com.br/bits-em-linguagem-c>. Acesso: 13 maio 2018.

LIMA C. B. D; VILLAÇA M. V. M. **AVR e Arduino Técnicas de Projeto.** 2. ed. Florianópolis: Autor Independente, 2012.

Domine o PWM. Disponível em: <https://www.embarcados.com.br/domine-o-pwm>. Data de acesso: 20 maio 2018.

Editorial Linguagens para Sistemas Embarcados. Disponível em: <https://www.embarcados.com.br/editorial-linguagens-para-sistemas-embarcados/>. Acesso: 26 abr. 2018.

Gimenez, S. P. Microcontroladores 8051. **Teoria do Hardware e do Software**. São Paulo. Pearson Educacional do Brasil Ltda, 2012.

GNU GCC – Introdução. Disponível em: <https://www.youtube.com/watch?v=GNIQYU9_0Qs>. Acesso: 25 abr. 2018.

Laboratórios de microcontroladores - UNICAMP. Disponível em: <http://www.ic.unicamp.br/~edson/disciplinas/mc404/2014-2s/ef/labs/lab01.html>. Acesso: 26 abr. 2018.

LIMA C. B. D; VILLAÇA M. V. M. **AVR e Arduino Técnicas de Projeto**. 2. ed. Florianópolis: Autor Independente, 2012.

MIZRAHI, V. V. **Treinamento em Linguagem C**. São Paulo. Pearson Prentice Hall, 2008. Disponível em: <http://unopar.bv3.digitalpages.com.br/users/publications/9788576051916>. Acesso: 29 maio 2018.

Primeiros passos com Atmel Studio. Disponível em: <https://www.embarcados.com.br/atmel-studio>. Acesso: 22 maio 2018.

Setup Eclipse with GCC ARM Embedded on 32 bits Ubuntu 16.04 i386. Disponível em: <https://embedded-things.blogspot.com.br/2016/06/setup-eclipse-with-gcc-arm-embedded-on.html>. Acesso: 26 abr. 2018.

Using Arduino with Atmel Studio 7. Disponível em: <http://www.riptutorial.com/arduino/topic/2567/using-arduino-with-atmel-studio-7>. Acesso: 26 abr. 2018.

Sistemas embarcados

Convite ao estudo

Caro aluno, nesta terceira unidade, iniciaremos nossos estudos voltados aos sistemas embarcados microprocessados que, de certa maneira, estão muito próximos ao que conhecemos e utilizamos corriqueiramente por meio de vários dispositivos, tais como nossos *smartphones*, computadores pessoais e automóveis. Entenderemos quais as principais diferenças entre um sistema microcontrolado e um sistema microprocessado, e como devemos escolher e trabalhar com sistemas operacionais dedicados para sistemas embarcados.

Na primeira seção, teremos uma visão introdutória do que é um sistema embarcado, e como diferi-lo dos sistemas microcontrolados como aqueles vistos nas unidades anteriores. Entenderemos também quais os detalhes dos projetos de hardware para estes sistemas e como os sistemas operacionais desempenham um papel fundamental no desenvolvimento de projetos embarcados.

Na próxima seção, conheceremos a fundo os chamados RTOS (*Real Time Operational System* ou Sistemas Operacionais de Tempo Real), que são embarcados principalmente em produtos de controle industrial, e garantem que todas as tarefas do *software* sejam executadas dentro de intervalos pré-definidos com a maior precisão possível. Por fim, na última seção, abordaremos outra classe de sistemas operacionais, mais voltados para interação com o usuário final e, neste caso, usaremos o conhecido projeto Linux como base dos nossos estudos.

Ao final da unidade, devemos estar aptos a conhecer e diferenciar as principais tecnologias para sistemas embarcados,

bem como compreender como os principais sistemas operacionais operam.

O produto piloto foi lançado ao mercado e se mostrou um sucesso de vendas e aceitação. Devido a este sucesso, a empresa pretende agora investir em mais pesquisa e desenvolvimento para a concepção de uma nova versão do produto, que deverá contar com poder de processamento mais avançado, bem como funcionalidades multimídia e que possibilitem maior interação com o usuário e melhor conectividade. Graças ao seu ótimo desempenho no desenvolvimento do primeiro produto, você foi escolhido para liderar o time de desenvolvimento que se debruçará sobre esse novo desafio. Como líder, o seu primeiro objetivo é pesquisar tecnologias de sistemas embarcados com maiores capacidades que aquelas encontradas no microcontrolador adotado no primeiro produto. Em decorrência desse processo de pesquisa e de escolha, você adentrará no universo de sistema operacionais embarcados, que expandirão as possibilidades de seu time em criar soluções inovadores que agreguem mais valor a esse novo produto.

Na sua opinião, o que devemos considerar quando pretendemos migrar de um projeto puramente microcontrolado para um sistema com maior capacidade, como um sistema embarcado microprocessado? As técnicas de desenvolvimento seriam as mesmas? Precisamos conhecer tão bem o *hardware* como nos projetos microcontrolados?

Espero que você esteja curioso e preparado para esse novo desafio! Boa sorte e ótimos estudos!

Seção 3.1

Introdução a sistemas embarcados

Diálogo aberto

Caro aluno, iniciamos agora mais uma importante etapa no nosso processo de aprendizado sobre sistemas embarcados, desta vez, estudaremos outro importante agente desses sistemas: os microprocessadores. É através deles que produtos que utilizamos cotidianamente são possíveis, como *smartphones*, computadores, automóveis, dentre outros.

Inicialmente, entenderemos quando devemos utilizar um microcontrolador em nosso projeto, e quando é mais adequado a utilização de um microprocessador, faremos isso analisando suas principais diferenças. Logo em seguida, vocês serão apresentados aos principais aspectos do projeto de hardware envolvendo microprocessadores, bem como a algumas plataformas já desenvolvidas que são utilizadas por projetistas para concepção de seus produtos. Por fim, introduziremos ao tema de sistemas operacionais dedicados a produtos embarcados, alguns deles, inclusive, já conhecemos em aplicações para computadores pessoais, e outros são dedicados a projetos nos quais a confiabilidade e o determinismo são chaves para o sucesso de sua aplicação.

Devido ao seu ótimo desempenho no desenvolvimento do primeiro produto, você foi escolhido para liderar o time de desenvolvimento que irá se debruçar sobre o desafio de desenvolver uma nova versão do produto, que contará com maior poder de processamento e novas funcionalidades multimídia para interação com o usuário. Como líder, o seu primeiro objetivo é pesquisar tecnologias de sistemas embarcados com maiores capacidades que aquelas encontradas no microcontrolador adotado no primeiro produto. Nessa etapa, você deve preparar um treinamento introdutório sobre essa nova tecnologia, criando um manual rápido contendo as principais diferenças entre sistemas microcontrolados e sistemas microprocessados. Nesse manual, deve estar contido também exemplos de sistemas de *hardware* microprocessados

e uma breve pesquisa sobre os principais sistemas operacionais utilizados nessa nova tecnologia. Quais as principais diferenças entre sistemas microcontrolados e sistemas microprocessados? Quais as vantagens e desvantagens de cada um? Quais os principais tópicos que devem ser dominados para se adentrar em projetos de sistemas embarcados microprocessados?

O conteúdo que veremos nesta seção será fundamental para vencer mais este desafio. Então ótimos estudos!

Não pode faltar

Conceitos iniciais de sistemas embarcados

Muitos são os sistemas atuais que utilizam em sua arquitetura processadores para fins diversos, desde relógios e máquinas de lavar, até computadores pessoais e supercomputadores. Mas como podemos então classificar um produto como sendo um sistema embarcado?

Um sistema embarcado é um sistema elétrico/eletrônico que é especificamente projetado para uma aplicação em um determinado domínio. É uma combinação única de *hardware* e *software* altamente especializados, que são selecionados de forma a atender os requisitos de uma aplicação em questão. Um sistema embarcado contém obrigatoriamente uma unidade de processamento que pode ser um processador, um microcontrolador, um SoC (System on Chip ou Sistema em um Chip), um ASIC (*Application Specific Integrated Circuit* ou Circuito Integrado de Aplicação Específica) ou PLD (*Programmable Logic Device* ou Dispositivo Lógico Programável). Deve conter também periféricos que auxiliem a interface de sensores e atuadores que agem como mensageiros do **Mundo Real** ao sistema embarcado que nele interage, periféricos de comunicação para interação com outros sistemas, dentre outros (SHIBU, 2009).

Os sistemas embarcados já se tornaram uma parte inevitável de todo e qualquer equipamento ou produto tecnológico em diversas áreas, como telecomunicações, equipamentos médicos, controle industrial, bens de consumo, automóveis, etc.

Microcontroladores e Microprocessadores

Como vimos em seções anteriores, os microcontroladores são dispositivos altamente integrados, que possuem, em um único *chip*,

não somente uma unidade de processamento responsável pela execução dos comandos do programa, como também memórias para armazenamento de instruções e dados, bem como periféricos, utilizados para realizar a interface do microcontrolador com o mundo externo e outros dispositivos. Tais características tornam os microcontroladores elementos ideias para diversos projetos eletrônicos que dependam de uma unidade de controle capaz de processar dados e interagir com outros dispositivos.

Contudo, uma das principais desvantagens de um microcontrolador é o seu poder de processamento e armazenamento de dados. A grande maioria dos microcontroladores trabalha em uma faixa de frequência de clock que pode atingir a escala de algumas dezenas de Mega-hertz, sendo considerado um valor relativamente baixo, e sua memória de programa e dados raramente ultrapassa a escala das dezenas de Megabytes.

De forma a contornar esse problema, produtos que demandam maior poder de processamento e maior capacidade de memória recorrem a tecnologias de sistemas embarcados baseados em microprocessadores que não dispõem internamente de todos os periféricos presentes em um microcontrolador, porém podem atingir altas taxas de processamento e se conectar a bancos de memória externa de alta capacidade. Em resumo, um microprocessador é um processador computacional que incorpora em si as funções de uma unidade central de processamento em um único dispositivo ou em poucos dispositivos integrados (KANT, 2007).

Reflita

Como vimos, os microprocessadores possuem maior capacidade de processamento que os microcontroladores. Então por que ainda se utilizam tantos microcontroladores na indústria, mesmo com tal desvantagem?

Uma das famílias de microprocessadores mais utilizadas atualmente na indústria, é a família de microprocessadores **ARM** (*Advanced RISC Machine* ou Máquina RISC Avançada), que podem ser considerados um marco na indústria de semicondutores, pois nunca uma arquitetura foi tão rapidamente difundida e maciçamente

fabricada como esta, devido a suas características atraentes que incluem um projeto simples, alta velocidade, grande diversidade de modelos e fabricantes e quantidade de *software* disponível conforme apresentado por Pereira (2007).

Os microprocessadores ARM são CPU's de 32 ou 64 bits que utilizam a filosofia RISC (*Reduced Instruction Set Computer* ou Computador com Conjunto Reduzido de Instruções) e um aspecto interessante é que, atualmente, a *ARM Limited* (Consórcio entre as empresas Acorn Computer Group, Apple e VLSI Technology) não fabrica chips, apenas licencia o uso de suas CPU's (propriedade intelectual) por outros fabricantes de microcontroladores e microprocessadores, como STMicroelectronics, ATMEL, Freescale, Texas Instruments, Samsung, Intel, entre outros.

Na Figura 3.1, temos a aplicação de um ARM A5 sendo aplicado no produto iPad da empresa Apple.

O conteúdo que veremos nesta seção será fundamental para vencer mais este desafio. Então ótimos estudos!

Figura 3.1 | Microprocessador ARM A5 embarcado em um produto da empresa Apple

Fonte: IDC: ARM-Powered Devices, Such As Apple's iPad Can Be Classified a PC Device. Disponível em: <https://www.cultofmac.com/93443/idc-arm-powered-devices-such-as-apples-ipad-can-be-classified-a-pc-device/>. Acesso: 4 jun. 2018

Entre as principais vantagens dos microprocessadores ARM, pode-se destacar (PEREIRA, 2007):

1. **Velocidade**: um dos grandes atrativos dos microprocessadores ARM e muitas aplicações podem

necessitar ou se beneficiar da velocidade extra obtida com a migração de uma CPU de 8 ou 16 bits para a plataforma ARM.

2. **Custo reduzido:** a arquitetura utilizada principalmente na família ARM7 consolidou-se e amortizou-se ao longo dos anos. Por isso, o custo de licenciamento reduziu bastante. Ao mesmo tempo, a baixa complexidade de implementação da CPU permite utilizar processos de fabricação mais baratos, que contribuem para a redução geral do custo.
3. **Demandas elevadas da aplicação:** muitas aplicações acabam se tornando tão complexas que obrigam o programador a utilizar várias técnicas para comprimir o código na memória de um microcontrolador de 8 ou 16 bits. Na arquitetura ARM, há um espaço de endereçamento linear de 4GB. O projetista também possui a sua disposição chips com capacidade de memória FLASH interna entre 8kB até1MB. Além disso, diversos microprocessadores ARM permitem o endereçamento de memória externa (FLASH ou RAM), atenuando as limitações de memória impostas por algumas arquiteturas de 8 ou 16 bits.
4. **Facilidade de migração:** nos tempos atuais, a grande maioria dos projetos é escrita em C e o uso desta linguagem, além de facilitar o desenvolvimento da aplicação, facilita a migração de uma arquitetura para outra.
5. **Sistemas operacionais:** Por possuir internamente estruturas de gerenciamento de memória MMU / MPU (*Memory Management Unit / Memory Protection Unit* ou Unidade de Gerenciamento de Memória / Unidade de Proteção de Memória), os dispositivos ARM são ideais para embarcar sistemas operacionais, como *Linux, Android ou Windows*.

Hardware para Sistemas Embarcados

Em um projeto eletrônico de sistemas controlados por um microcontrolador, podemos contar com uma arquitetura

de hardware minimalista, na qual o controlador necessita tão somente de uma fonte confiável de alimentação, uma base de clock que pode ser externa por meio de um oscilador, ou interna, e todos os componentes adicionais para o circuito dependem única e exclusivamente dos periféricos externos.

Já em um projeto de hardware para sistemas embarcados microprocessados, deve-se considerar que o processador em si pode não possuir internamente todos os componentes necessários para o seu correto funcionamento, tais como memória RAM, Flash, conversores e periféricos de comunicação. Dessa forma, para esse tipo de sistema, o projeto de hardware tende a ser mais complexo e, consequentemente, com maior valor agregado.

Outro fator muito relevante diz respeito a complexidade do projeto em si, no qual componentes que trabalham em alta frequência, como memórias externas por exemplo, devem ser posicionadas estrategicamente, e o roteamento de seus barramentos de comunicação que as interligam ao microprocessador devem ser minuciosamente calculados e projetados para se evitar falhas devido a ruídos por condução ou por emissão. Devido a este fator, muitas empresas tendem a adotar sistemas pré-fabricados de hardware para o seu projeto que, além de possuírem os periféricos básicos para o correto funcionamento do microprocessador, ainda embarcam periféricos extras como interfaces multimídia, entradas USB (*Universal Serial Bus* ou Barramento Serial Universal), entradas HDMI (*High-Definition Multimedia* Interface ou Interface Multimídia de Alta Resolução), interface Ethernet, interfaces de áudio, saídas para monitores e telas, entre outros. Grande parcela desses sistemas já possuem homologação que garantem que os mesmos podem operar em condições adversas relativas a ruídos eletromagnéticos, falhas de alimentação, altas amplitudes de temperatura, etc, o que reduz significativamente o custo da empresa que pretende desenvolver algum sistema embarcado microprocessado.

Um dos projetos de hardware mais famosos atualmente e com maior número de usuários entre projetistas, engenheiros e estudantes é a *Raspberry Pi,* que se baseia em um computador

em miniatura, originalmente projetado para fins única e exclusivamente educacionais, inspirados no computador BBC Micro, de 1981. O objetivo de seu criador, Eben Upton, era fazer um dispositivo de baixo custo que pudesse melhorar as habilidades de programação e o entendimento de hardware de alunos em nível pré-universitário. Mas, graças a seu pequeno tamanho e preço acessível, o projeto foi rapidamente adotado por entusiastas, engenheiros e estudantes para projetos que requeriam mais que um simples microcontrolador. O projeto também é considerado *Open Hardware* (*Hardware* Aberto), ou seja, que toda sua documentação de projetos e diagramas é aberto para todos os interessados em conhecer ou até mesmo modifica-lo (OPENSOURCE, 2018).

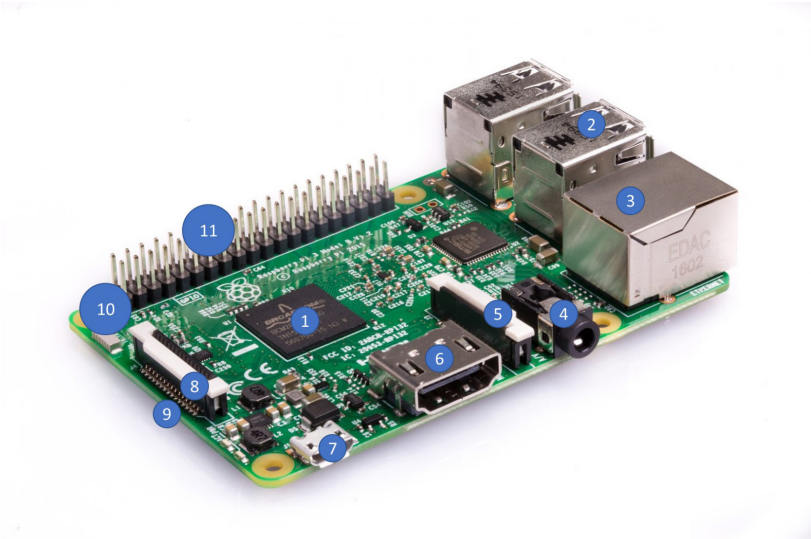
Pesquise mais

O que acha de conhecer um pouco mais sobre esse fascinante projeto? Acesse o site oficial da Raspberry Pi. Disponível em: <https://www.raspberrypi.org/>. Acesso 4 jun. 2018.

Na Figura 3.2, é apresentado o modelo *Raspberry* Pi 3, com as seguintes características:

1. Microprocessador Broadcom BCM2837 – ARM Cortex A53, 64 bits, 4 núcleos, frequência máxima de 1.2GHz, 1GB de memória RAM interna.
2. 4 Portas USB Versão 2.
3. 1 Porta LAN (*Local Area Network* ou Rede de área local).
4. Saída de Aúdio 4 pólos.
5. Porta para câmera CSI (*Camera Serial Interface* ou Interface Serial para Câmera).
6. Saída HDMI.
7. Entrada USB Micro para alimentação.
8. Saída para display DSI (*Display Serial Interface* ou Interface Serial para Display).
9. Entrada para cartão de memória MicroSD.
10. Wi-Fi e Bluetooth embarcados.
11. 40 pinos de IO para extensão.

Figura 3.2 | *Raspberry Pi* Modelo 3



Fonte: adaptada de <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>. Acesso em: 4 jun. 2018.

Sistemas Operacionais Embarcados

Um sistema operacional, por mais complexo que possa parecer, é apenas um conjunto de rotinas executado pelo processador, de forma semelhante aos programas dos usuários. Sua principal função é controlar o funcionamento de uma unidade computacional, gerenciando a utilização e o compartilhamento de seus diversos recursos, como processadores, memórias e dispositivos de entrada e saída. Sem um sistema operacional, um usuário para interagir com um computador deveria conhecer profundamente diversos detalhes sobre o hardware do equipamento, o que tornaria seu trabalho lento e com grandes possibilidades de erros. O sistema operacional tem como objetivo funcionar como uma interface entre o usuário a unidade computacional, tornando sua utilização mais simples, rápida e segura.

A grande diferença entre um sistema operacional e aplicações convencionais é a maneira como suas rotinas são executadas em função do tempo. Um sistema operacional não é executado de forma linear como na maioria das aplicações, com início, meio e fim. Suas rotinas são executadas concorrentemente em função de

> eventos assíncronos, ou seja, eventos que podem ocorrer a qualquer momento. (MACHADO, 2013, p. 3)

Na Figura 3.3, temos uma representação da interação de um sistema operacional.

Figura 3.3 | Visão de um sistema operacional

Fonte: <https://www.tecword.com.br/sistema-operacional/>. Acesso em: 4 jun. 2018.

Para sistemas embarcados, um sistema operacional desempenha um papel semelhante àquele utilizando em computadores pessoais, ou seja, gerenciando os recursos do sistema e controlando o acesso ao hardware via funções de acesso específicas. Da mesma maneira, sistemas operacionais embarcados coordenam aplicações do usuário que são executados de maneira concorrente ou *multitasking* (múltiplas tarefas), em que a parcela de execução de cada rotina é ditada por regras de prioridade. Podemos classificar os sistemas operacionais para embarcados entre dois grupos: sistemas operacionais genéricos e sistemas operacionais de tempo real.

Assimile

Lembre-se de que, para o software desenvolvido para microcontroladores, geralmente não se utilizam sistemas operacionais, sendo que todo o controle do sistema é realizado a partir de uma única função que é executada em laço infinito, e o tempo de execução pode variar entre cada interação do laço.

Sistemas operacionais genéricos são aqueles que o próprio sistema operacional controla a parcela de execução e de priorização das tarefas que serão executadas, dispondo pouco ou nenhum acesso ao usuário para controle da priorização e tempo de execução.

Exemplificando

Alguns exemplos de sistemas operacionais embarcados genéricos são:

- Linux embarcado: encontrado em roteadores, aparelhos de TV e centrais Multimedia.
- Windows CE: utilizado em dispositivos como interfaces gráficas industriais, aparelhos de GPS e *Palmtops*.
- Android: Smartphones, Tablets e Smart TVs.

Já sistemas operacionais de tempo real possuem maior rigidez quanto ao controle de execução das rotinas de software, sendo de responsabilidade do desenvolvedor definir alguns parâmetros para tais rotinas, como prioridade de execução, tempo de execução e parada, quantidade de memória reservada, entre outros.

Muitos sistemas operacionais *multitasking* disponíveis atualmente são descritos como "tempo real". Esses sistemas operacionais fornecem funcionalidades adicionais que permitem aplicações que normalmente interagem diretamente com a arquitetura do microprocessador para manipular interrupções e acionar periféricos, que o façam sem o próprio sistema operacional interfira ou bloqueie tais operações. Muitos sistemas operacionais genéricos impediriam que o usuário acessasse tais recursos. Esse cuidado excessivamente zeloso impede que muitos desses sistemas operacionais sejam utilizados em aplicações específicas, como controle industrial.

Uma característica de um sistema operacional de tempo real é o tempo de resposta pré-definido a estímulos externos. Se um periférico gerar uma interrupção, um sistema em tempo real vai reconhecê-la e começará a servi-la dentro de um tempo máximo definido. Tais tempos de resposta variam de sistema para sistema, mas o tempo máximo especificado é sempre aquele tido como pior caso e não deve ser ultrapassado, mesmo quando o sistema se encontre sobrecarregado (HEATH, 2003).

Sem medo de errar

Nesta nova etapa, fora solicitado a você que criasse um manual rápido contendo as principais diferenças entre sistemas microcontrolados e sistemas microprocessados. Nesse manual, deve estar contido também exemplos de sistemas de hardware microprocessados e uma breve pesquisa sobre os principais sistemas operacionais utilizados nessa nova tecnologia.

Inicialmente, podemos sintetizar a diferenças entre um microcontrolador e um microprocessador por meio de uma tabela, contendo tópicos relevantes para ambas tecnologias, como pode ser visto na Tabela 3.1.

Tabela 3.1 | Diferenças entre microcontrolador e microprocessador

Microprocessador	Microcontrolador
CPU opera sozinha. RAM, Flash, IO, Temporizadores são separados do chip	CPU, RAM, Flash, IO e Temporizadores estão todos em um único chip
O projetista pode decidir a quantidade de memória RAM, Flash e IO, visto que são externos ao chip	Quantidade fixa de memória e IOs
Alto custo	Feitos para aplicações em que custo, consumo e espaço são críticos
Propósito Geral	Propósito Específico
Alto poder de Processamento	Baixo poder de Processamento
Geralmente 32 ou 64 bits	Geralmente 8 ou 16 bits
Microprocessador = CPU	Microcontrolador = CPU + Periféricos + Memória

Fonte: elaborada pelo autor.

Após uma breve pesquisa podemos encontrar alguns modelos de placas de hardware para sistemas microprocessados e são elas:

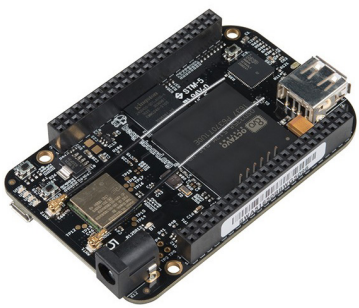
Raspberry Pi 3 (Figura 3.2)

- Processador Broadcom BCM2837 de 1.2GHz 64 bits Quad Core
- 1 GB SDRAM @400 MHz
- Wifi 802.1n e Bluetooth 4.1 integrados
- GPIO 40 pinos
- Adaptador micro SD
- GPU Videocore IV
- 4 Portas USB 2
- 1 Porta Ethernet LAN
- Saída de Áudio
- Saída para vídeo (CSI, DSI e HDMI)

Beaglebone Black Wireless (Figura 3.4)

- Processador: Octavo Systems OSD3358 1GHz ARM® Cortex-A8
- Memória: 512 MB DR3 RAM
- Armazenamento: 4GB 8-bit
- Acelerador gráfico 3D
- Wifi 802.11 b/g/n
- Bluetooth BLE 4.1
- Conector HDMI
- Porta USB host
- 2x GPIO 46 pinos

Figura 3.4 | *BeagleBone Black Wireless*



Fonte: adaptada de <https://www.filipeflop.com/produto/beaglebone-black-wireless/>. Acesso em: 4 jun. 2018.

Dentre os principais sistemas operacionais para dispositivos embarcados, podemos destacar:

- Linux embarcado: versão customizada do Linux, mantendo-se o kernel e adequando seus serviços para as limitações de um hardware para sistemas embarcados. Muito aplicado em produtos de telecomunicação e projetos de interfaces gráficas.
- Android: baseado no projeto Linux, é muito utilizado em smartphones e tablets. Possui vários recursos e interface amigável para o usuário final, possuindo drivers prontos para produtos de mobilidade, como GPS, display sensível ao toque, câmeras, dentre outros.
- FreeRTOS: sistema operacional de tempo real mais utilizado atualmente. É um sistema aberto e gratuito, com capacidade de gerenciamento de centenas de tarefas simultâneas, e customizado para operar com os principais microprocessadores do mercado.

Avançando na prática

Um novo protótipo microprocessado

Descrição da situação-problema

Dado o seu conhecimento recém-adquirido sobre sistemas embarcados, o seu gestor lhe solicitou uma análise sobre o desenvolvimento de um protótipo que será utilizado como teste de conceito para um novo produto. Esse protótipo deve executar um software de interface gráfica para o controle automático da residência e será embarcado com sistema operacional Linux. O seu objetivo é apresentar um resumo sobre quais os passos o time poderia seguir para implementar essa aplicação piloto com menor tempo possível e maior valor agregado.

Resolução da situação-problema

Como o requisito é que o protótipo seja embarcado com sistema operacional Linux para executar uma aplicação de interface gráfica, o mais aconselhável é utilizar um sistema embarcado

microprocessado. Por ser um protótipo para teste de conceito, não é recomendável que seja desenvolvido todo o projeto de hardware para o sistema, uma vez que muito pode se alterar entre o protótipo e a versão final do produto, o que acarretaria em várias modificações no projeto, dessa forma, é recomendável que se utiliza uma plataforma de hardware já desenvolvido, como uma Raspberry Pi ou uma Beagle Bone Black.

Ambas plataformas possuem suporte nativo para o sistema operacional Linux Embarcado, em versões customizadas para cada plataforma. Assim, o desenvolvimento em si se limitaria no *software* de interface gráfica, que pode ser projetado em qualquer computador que execute o sistema Linux e depois portado para a placa protótipo.

Faça valer a pena

1. Uma das principais desvantagens de um microcontrolador é o seu poder de processamento e armazenamento de dados. A grande maioria dos microcontroladores trabalha em uma faixa de frequência de clock que pode atingir a escala de algumas dezenas de Mega-hertz, sendo considerado um valor relativamente baixo, e sua memória de programa e dados raramente ultrapassa a escala das dezenas de Megabytes.

Avalie as asserções apresentadas a seguir referentes a utilização de microcontroladores e microprocessadores e a relação entre elas.

I. Em determinados projetos, nos quais é necessário a utilização de vários periféricos, como temporizadores, conversores analógico-digital e interrupções, o mais aconselhável é a utilização de microcontroladores.

PORQUE

II. Para aplicações que exigem maior diversidade de periféricos, sempre haverá a necessidade do uso de maior poder de processamento e maior capacidade de memória, características típicas dos microcontroladores.

A respeito dessas asserções, assinale a opção correta.

a) As asserções I e II são proposições verdadeiras, e a II é uma justificativa da I.

b) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa da I.
c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
e) As asserções I e II são proposições falsas.

2. Em um projeto de hardware para sistemas embarcados microprocessados, deve-se considerar que o processador em si pode não possuir internamente todos os componentes necessários para o seu correto funcionamento, tais como memória RAM, Flash, conversores, periféricos de comunicação etc. Dessa forma, para esse tipo de sistema, o projeto de hardware tende a ser mais complexo e, consequentemente, com maior valor agregado.
Qual a vantagem em se utilizar produtos hardware já desenvolvidos em relação ao desenvolvimento de hardware customizado para uma aplicação envolvendo microprocessadores?

a) Maior complexidade e confiabilidade.
b) Menor custo e maior complexidade.
c) Maior poder de processamento e capacidade de memória.
d) Maior confiabilidade e disposição de recursos extras.
e) Maior poder de processamento e menor complexidade.

3. Sistemas operacionais de tempo real possuem maior rigidez quanto ao controle de execução das rotinas de software, sendo de responsabilidade do desenvolvedor definir alguns parâmetros para tais rotinas, como prioridade de execução, tempo de execução e parada, quantidade de memória reservada, dentre outros.

A respeito dos Sistemas Operacionais de Tempo Real, considere as seguintes afirmações:

( ) São sistemas nativamente multitasking.
( ) São mais velozes que os sistemas operacionais genéricos, por serem de tempo real.
( ) O acesso ao hardware é totalmente controlado pelo sistema operacional.
( ) O tempo de execução das tarefas deve ser calculado considerando a pior condição de execução.

Considerando a sequência, qual das alternativas abaixo retrata corretamente quais afirmações são falsas e quais são verdadeiras?

a) F, F, F, V.
b) F, V, F, V.
c) F, V, F, F.
d) V, F, F, V.
e) V, F, V, V.

Seção 3.2

Sistemas operacionais de tempo real

Diálogo aberto

Prezado aluno, nesta seção, continuaremos nossos estudos a respeito dos sistemas embarcados. Desta vez, vamos nos aprofundar no entendimento dos sistemas operacionais de tempo real (RTOS - *Real-time Operating Systems*), e como os mesmos se diferem de outros sistemas operacionais e até mesmo de sistemas convencionais de programação para microcontroladores, como aqueles vistos nas unidades anteriores. Os sistemas de tempo real desempenham um papel fundamental no desenvolvimento de produtos para sistemas de controle industrial, automotivo e aeroespacial. Vários são os dispositivos que dependem de seus aspectos determinísticos e de segurança para operar de maneira robusta, minimizando a possibilidade de falha ou de erros de execução, como é o caso de módulos automotivos (*airbag*, central de controle, painel de instrumento), sistemas de aviônica (radares, controladores de voo, controle aerodinâmico) e robôs industriais.

Inicialmente, compreenderemos o que representa um sistema operacional de tempo real, e quais são suas diferenças em relação aos sistemas tradicionais do tipo laço infinito. Serão apresentados, em seguida, os principais agentes desse sistema: o *kernel* e o escalonador. Assim veremos como as tarefas são gerenciadas por esses elementos de forma a maximizar a utilização da CPU por um sistema microprocessado ou microcontrolado. Por fim, veremos meios de realizar a comunicação entre tarefas, seja para a transmissão de dados entre as mesmas, ou para sincronizar a operação entre várias tarefas.

Lembre-se de que agora você é o líder do time de desenvolvimento de *software* da empresa, e sua próxima tarefa é avançar com o desenvolvimento de uma nova plataforma microprocessada para o produto de automação residencial. Para que o time possa se habituar a essa nova tecnologia, você decidiu que a primeira versão do software de controle em tempo real deve implementar as mesmas

funcionalidades implementadas na primeira versão do produto, só que dessa vez utilizando o sistema operacional *FreeRTOS* e algumas de suas funcionalidades, como criação e gerenciamento de tarefas, comunicação via semáforos e troca de dados através de filas. Você deve elaborar um relatório técnico descrevendo inicialmente, todas as tarefas que deverão ser criadas para implementar todo o escopo de *software*. Na descrição das tarefas, deve ser feita uma análise sobre suas prioridades e a quantidade de memória a ser alocada para cada qual. Feito isso, descreva como utilizar semáforos e filas para a comunicação e troca de dados entre essas tarefas. Os tópicos que veremos nesta seção serão de fundamental importância para o entendimento do funcionamento de um sistema de tempo real, e de como podemos utilizar filas e semáforos para a comunicação entre as tarefas.

Como decidiremos quais tarefas possuem mais prioridade em relação a outras? Podemos inferir com precisão o consumo de memória de cada tarefa?

O conteúdo desta seção está estruturado para lhe auxiliar a resolver mais esse desafio, bem como lhe ensinará conceitos fundamentais que podem ser utilizados em problemas práticos da mesma natureza. Então bons estudos, e que esse conhecimento motive você para novas jornadas em sistemas embarcados!

Não pode faltar

Conceitos Básicos de Sistemas de Tempo Real

No universo do desenvolvimento de software para sistemas embarcados, um sistema é projetado para receber um evento, que pode ser de origem interna ou externa, realizar o seu processamento e produzir uma saída. Alguns sistemas trabalham com eventos que possuem restrições de tempo, ou seja, possuem um prazo ou um tempo limite para que o evento seja processado e que uma saída correspondente seja acionada. Esses tipos de sistemas são denominados **Sistemas de Tempo Real**. Portanto, um sistema de tempo real precisa garantir com que todos os eventos sejam atendidos dentro das suas respectivas restrições de tempo, dando-lhe a característica de **determinismo**, e não necessariamente de velocidade de execução.

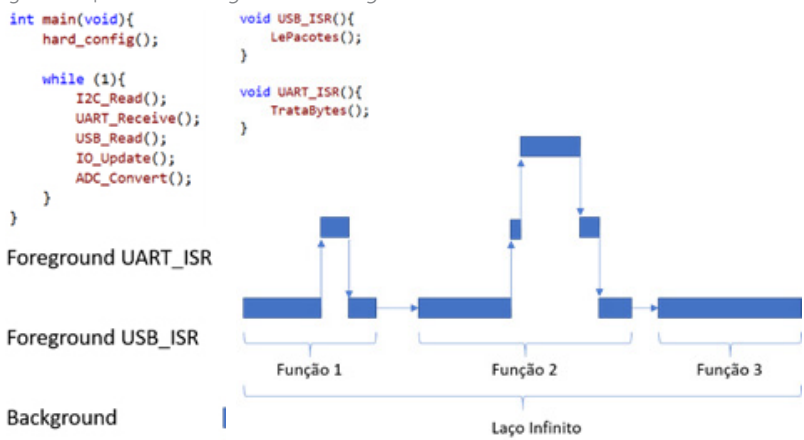
Para um sistema em tempo real, cada serviço de entrada impõe uma tarefa (do inglês: *task*), normalmente associada a uma restrição

computação de tempo real de execução ou simplesmente chamado de restrição de tempo. A **restrição de tempo** de uma tarefa é normalmente especificada em termos de seu prazo de execução, que é a janela de tempo pelo qual sua execução (ou serviço) deve ser concluída (FAN, 2015). Dependendo de quão grave a perda do prazo de execução de uma tarefa é, um sistema de tempo real pode ser classificado como:

- ***Soft real-time*** (Sistema de tempo real flexível): uma restrição de tempo não atingida tem como consequência simplesmente a percepção de baixa qualidade do sistema. Exemplo: um *display* com sistema *touch* que demora para responder ao tocar na tela.
- ***Hard real-time*** (Sistema de tempo real rígido): uma restrição de tempo não atingida pode inutilizar o sistema ou provocar consequências catastróficas. Exemplo: um sistema de *airbag* que não responde no tempo correto no momento da colisão de um veículo.

Em projetos de pequena complexidade, geralmente o seu desenvolvimento é realizado através da técnica conhecida como ***Foreground / Background*** (Primeiro plano / Segundo plano). A aplicação em si consiste em laço infinito, que realiza a chamada de algumas funções para a realização de operações desejadas (*background*), e as rotinas de tratamento de interrupção tratam eventos assíncronos (*foreground*). Na Figura 3.5, temos o exemplo de um sistema desenvolvido com esta técnica.

Figura 3.5 | Sistema *Foreground / Background*



Fonte: elaborada pelo autor.

As funções são executadas linearmente pelo laço infinito e, sempre que uma interrupção (ISR) ocorre, um desvio na execução é gerado para o atendimento da mesma, que retorna ao final de seu tratamento. Dessa forma, o tempo total de execução de uma interação do laço é variado, dependendo do tempo de execução das funções de *background*, e da eventual ocorrência de eventos de interrupção (*foreground*). Este tipo de metodologia é fácil e rápido para o desenvolvimento, não requer nenhum treinamento ou conhecimento específico sobre sistemas operacionais. Além do mais, não consome recursos adicionais, comparado à uma solução baseada em sistema operacional, o que a torna ideal para projetos pequenos, com requisitos modestos de restrições de tempo. Como desvantagens, pode-se destacar que todas as tarefas em background possuem a mesma prioridade de execução, e a demora demasiada em uma delas acarreta em atraso de execução de todo o sistema. Em relação às tarefas de *foreground* (ISRs), caso não se tenha um cuidado em relação a densidade de sua implementação, o sistema pode levar mais tempo que o previsto para a resposta a estímulos.

Para projetos que tais características se tornam limitantes a qualidade do sistema, a utilização de sistemas de tempo real se torna mandatória, pois conseguem contornar as principais deficiências dos sistemas *foreground / background* por meio de rotinas priorizáveis, gerenciamento de interrupções e escalonamento de tarefas.

Introdução ao FreeRTOS

Todo sistema operacional de tempo real, possui um componente central chamado kernel. Um kernel de tempo real é um software que gerencia o tempo e os recursos da CPU, e é baseado no conceito de tarefas e prioridades. Todas as funções do sistema são divididas em tarefas (*tasks*), e o *kernel* decide quando uma tarefa deve ser executada baseado na prioridade da mesma, ficando sob responsabilidade do desenvolvedor dividir o sistema em tarefas, e definir suas prioridades de acordo com as características de tempo real de cada qual conforme indica Prado (2016).

Em um sistema multitarefa, tem-se a impressão que todas as tarefas são executadas ao mesmo tempo, mas, como o processador só pode executar uma única tarefa por vez, é realizado o chaveamento entre as tarefas. Este chaveamento ou troca de contexto de tarefas, pode

acontecer quando uma tarefa é bloqueada esperando a liberação de um recurso, quando a mesma entra em modo de espera (tarefas periódicas), ou quando uma tarefa é involuntariamente suspensa pelo *kernel*. O **escalonador** (do inglês: *scheduler*) é o elemento do *kernel* responsável por decidir qual é a próxima tarefa a ser executada em determinado momento, e seu algoritmo de troca de contexto é também chamado de **política de escalonamento** (SHIBU, 2009).

Além de gerenciar o uso da CPU, um *kernel* de tempo real possui normalmente a responsabilidade de gerenciar a comunicação entre as tarefas, gerenciar o acesso aos recursos de hardware, gerenciar o uso de memória e prover outras funcionalidades, como temporizadores, mecanismos de troca de informação, etc.

Reflita

Dadas todas as vantagens em se utilizar um sistema operacional de tempo real no desenvolvimento de um sistema embarcado, por que ainda existem tantos projetos utilizando a metodologia *foreground / background?* Seria válido migrar esses projetos para um sistema em tempo real?

Nesse livro, utilizaremos como base de estudo, aquele é que considerado o RTOS (*Real Time Operational System* ou Sistema Operacional de Tempo Real) de código aberto mais utilizado no mundo, o ***FreeRTOS.*** Esse sistema foi criado no ano de 2003 por *Richard Barry* e mantido por sua empresa *Real Time Engineers Ltd.* até o ano de 2017, quando a mesma foi adquirida pelo grupo *Amazon*.

Pesquise mais

Que tal saber mais sobre como é criado um projeto utilizando o FreeRTOS? Acesse: *Criando um projeto no IAR com o FreeRTOS*. Disponível em: <https://www.embarcados.com.br/criando-um-projeto-no-iar-com-o-freertos/>. Acesso em: 10 jun. 2018.

O sistema FreeRTOS foi projetado para ser pequeno, simples e de fácil utilização, sendo escrito em linguagem C para ser extremamente portável, e hoje suporta mais de 30 arquiteturas diferentes de microcontroladores e microprocessadores. Seu *kernel* pode trabalhar de

forma **preemptiva** ou **colaborativa**, com capacidade de rastreamento de tarefas e detecção de estouro de memória (*stack overflow*). Sua implementação atual não possui restrição quanto a quantidade de tarefas que podem ser criadas, ou até mesmo a quantidade de prioridades que podem ser utilizadas. Outra grande vantagem é o seu código totalmente aberto, sem royalties de utilização, possuindo diversas outras ferramentas abertas e gratuitas para o desenvolvimento de *software* (FREERTOS, 2018).

Gerenciamento de Tarefas

Cada tarefa dentro do FreeRTOS se comporta como sendo um programa isolado, possuindo um ponto de entrada, implementando um laço infinito dentro de seu código, e geralmente não retorna nenhuma informação, ou seja, se uma tarefa finalizar, é de responsabilidade do desenvolvedor removê-la da lista de tarefas do kernel (BARRY, 2016). Na Figura 3.6, é apresentado um exemplo de criação de uma nova tarefa utilizando o FreeRTOS.

Figura 3.6 | Exemplo de criação de tarefa com FreeRTOS

```
void vTaskExample(void *pvParameters);

int main(void){
    /*Criando uma nova Tarefa*/
    xTaskCreate(vTaskExample, "TaskExemplo", 256, NULL, 1, NULL)
    /*Iniciando o escalonador*/
    vTaskStartScheduler();
}

void vTaskExample(void *pvParameters)
{
    while(1){
        /*Código da Tarefa*/
    }
}
```

Fonte: elaborada pelo autor.

Inicialmente, deve-se declarar o protótipo da função que representará a nova tarefa, possuindo como argumento um ponteiro para parâmetros que se deseja repassar a mesma no momento de sua criação. Feito isso, para se incluir a nova tarefa no

kernel do sistema, deve-se utilizar a função **xTaskCreate**, passando respectivamente como argumentos a função da tarefa, o nome da mesma, a quantidade de bytes de memória que serão alocados (*stack*), o parâmetro de inicialização da tarefa, sua prioridade e, finalmente, um ponteiro para uma função que será executada no momento que a tarefa é criada. Perceba que, no exemplo dado na Figura 3.6, tanto o parâmetro quanto a função de inicialização foram deixados como NULO (NULL), por serem valores opcionais na criação de uma tarefa.

Cada tarefa deve ter sua prioridade definida de acordo com suas características de tempo real, de forma que o escalonador sempre execute a tarefa com maior prioridade. É relativamente fácil atribuir prioridades às tarefas em sistemas mais simples. Normalmente está bem evidente que, por exemplo, determinada tarefa de controle tem mais prioridade que uma tarefa que gerencia a interface com o usuário, mas na maioria das vezes, não é tão simples como parece, devido à complexidade de sistemas de tempo real. Como regra geral, tarefas do tipo *hard real-time* devem ter maior prioridade que aquelas do tipo *soft real-time*, mas outras características como frequência de execução e uso da CPU devem ser levadas em consideração conforme indica Prado (2016). Algumas outras técnicas para a atribuição de prioridades são:

- Tarefas com processamento intensivo devem ter menor prioridade, para evitar o uso excessivo da CPU.
- Tarefas periódicas, executadas em unidades de milissegundos, devem ser realizadas em uma tarefa com prioridade maior.
- Tarefas periódicas, executadas em unidades de microssegundos ou menos, devem ser realizadas dentro de interrupções (ISR).
- Rotinas que manipulam interface com o usuário devem ser executadas na ordem de centenas de milissegundos, em uma prioridade que deve refletir seu tempo máximo de execução.

O tamanho do *stack* de cada tarefa vai depender da aplicação em si, sendo necessário levar em consideração todas as chamadas de função, variáveis locais e contexto da CPU para rotinas de interrupção. É possível calcular o *stack* de uma tarefa manualmente, mas pode ser algo trabalhoso e sujeito à erros, na prática recomenda-se que se defina um tamanho padrão para a *stack* de cada tarefa

e, durante o desenvolvimento e testes da aplicação, seja realizado um monitoramento e ajuste da quantidade de memória de acordo com a utilização, evitando também, o uso de funções recursivas conforme apontado por Prado (2016).

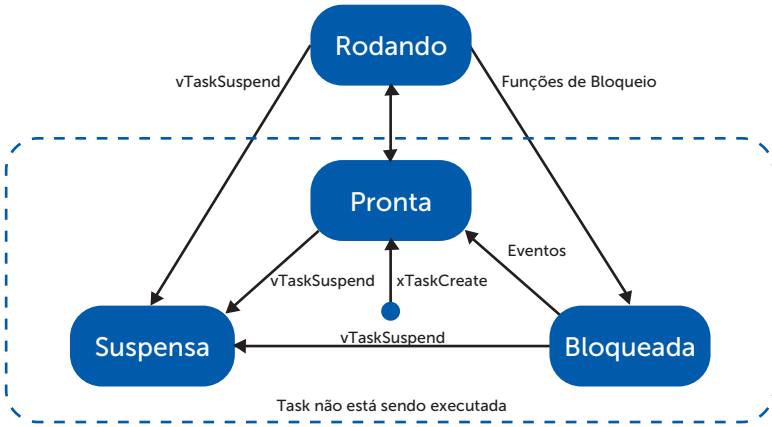
Estado das Tarefas

Quando o escalonador é definido no modo colaborativo, as tarefas não são interrompidas pelo *kernel* durante a execução, neste caso, a mesma precisa liberar a CPU de forma voluntária para que o escalonador possa selecionar outra tarefa para execução. Já no modo preemptivo, sempre que uma tarefa de maior prioridade está pronta para execução, o kernel irá interromper a tarefa de menor prioridade para executar a tarefa de maior prioridade, e para tarefas de mesma prioridade, o *kernel* definirá uma fatia de tempo de execução para cada tarefa conforme apresenta Prado (2016).

De forma a otimizar o uso da CPU utilizando um sistema do tipo RTOS, deve-se projetar as tarefas para que, sempre que possível, as mesmas sejam orientadas à eventos, ou seja, devem aguardar um determinado evento para realizar o seu processamento. Uma tarefa aguardando um evento fica no modo **Bloqueada**, sendo que tais eventos podem ser de origem temporal (rotinas de atraso) ou de sincronização (mecanismos de comunicação como filas e semáforos).

A função **vTaskSuspend** do FreeRTOS pode ser usada para colocar uma tarefa no modo **Suspensa**, ou seja, a mesma não será escalonada pelo *kernel*, sendo liberada da suspensão pela função **vTaskResume**. As tarefas que não estão nem no modo Suspenso, nem no modo Bloqueada, estão no modo **Pronta** (*Ready*), aguardando na fila para serem selecionadas e executadas pelo escalonador. A Figura 3.7 apresenta um diagrama dos possíveis estados de uma tarefa dentro do *kernel* do FreeRTOS.

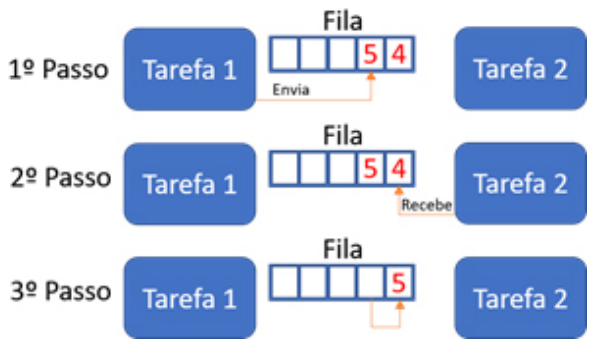
Figura 3.7 | Diagrama de Estado da Tarefas



Fonte: elaborada pelo autor.

O FreeRTOS, assim como a maioria dos sistemas operacionais, é estruturado através de um conjunto de tarefas independentes entre si, e é muito provável que essas tarefas precisem se comunicar. Uma das formas mais eficazes de se realizar troca de dados entre tarefas é por meio de um mecanismo de comunicação chamado *Queue* (Fila), conforme apresentado na Figura 3.8.

Figura 3.8 | Passos de utilização de uma Fila



Fonte: elaborada pelo autor.

Uma Fila não pertence a nenhuma tarefa em específico, sendo que várias tarefas podem compartilhar a mesma fila, tanto para enviar, quanto para receber dados. Cada fila armazena um conjunto finito de

itens (*queue lenght* ou comprimento da fila), que pode ter um tamanho fixo de bytes (item size ou tamanho do item), e ambos os parâmetros devem ser definidos no momento de criação da fila (BARRY, 2016).

As filas são usadas de maneira FIFO (*First In First* Out ou Primeiro que entra Primeiro que sai), em que os dados são escritos no fim da fila e lidos no início, mas não é uma regra fixa, sendo possível escrever logo no início da fila. Entenda-se por escrever na fila como copiar dos dados byte a byte, logo, se o elemento da fila for muito grande, o ideal é utilizar ponteiros. Ao escrever em uma fila, uma tarefa pode entrar no estado Bloqueada se a fila já estiver cheia, aguardando um espaço na mesma para salvar o novo item, além disso, a tarefa pode definir um tempo de espera (*timeout*) de escrita, em que ficará bloqueada aguardando. Quando o elemento que se deseja escrever é escrito com sucesso, ou seja, o *timeout* de escrita se expira, a tarefa é colocada automaticamente no estado de Pronta.

Ao ler um item de uma fila, uma tarefa entra no estado Bloqueada aguardando a chegada do item e, da mesma forma que no processo de escrita, é possível definir um *timeout* para aguardar a chegada do item. A tarefa será colocada automaticamente em estado de Pronta quando um item da fila é lido ou quando o *timeout* expirar.

Assimile

A utilização de filas constitui uma das principais formas de colocar uma tarefa em espera (bloqueada), aguardado a chegada de dados para o seu processamento. Essa boa prática aumenta a eficiência da aplicação, racionando de maneira inteligente o uso da CPU, uma vez que a tarefa só é executada quando de fato existem dados a serem processados.

Semáforos

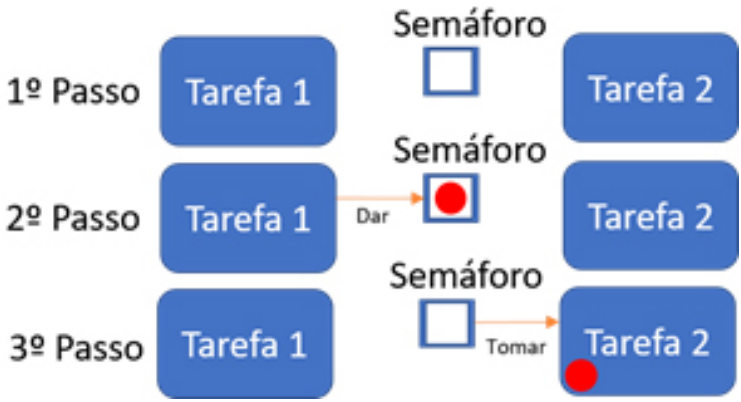
Um semáforo é um mecanismo de sincronização disponibilizado pelo FreeRTOS (e também por outros sistemas operacionais) e pode ser utilizado para desbloquear uma tarefa quando um determinado evento ocorrer, por exemplo, uma interrupção, sincronizando este evento a tarefa (BARRY, 2016).

Exemplificando

Uma tarefa é criada para salvar todos os dados recebidos pelo periférico USB em um cartão de memórias. Como a mesma não sabe quando os dados serão transferidos, é implementado um semáforo para que a interrupção (ISR) do periférico USB desbloqueie a primeira tarefa, de forma a permitir que os dados sejam gravados somente nesse evento.

Da mesma forma que uma Fila, um semáforo não pertence a nenhuma tarefa em específico. Quando uma tarefa necessita de um evento para continuar o seu processamento, a mesma aguarda o recebimento de um sinal do semáforo e fica no estado Bloqueada, e quanto o evento ocorre na tarefa ou interrupção geradora, a mesma envia o sinal ao semáforo, que faz com que o *kernel* coloque a primeira tarefa em estado de Pronta, aguardando para voltar ao seu processamento assim que o escalonador a selecionar. Na Figura 3.9, são apresentados os passos de funcionamento de um semáforo.

Figura 3.9 | Passos de utilização de uma Semáforo



Fonte: elaborada pelo autor.

Sem medo de errar

Nesta seção, você, responsável pelo time de desenvolvimento, decidiu demonstrar a utilização do sistema de tempo real FreeRTOS por meio da criação de tarefas que representem as funções de

software que foram implementadas no produto na primeira versão do programa, em que a técnica de laço infinito fora utilizada.

Relembrando que, inicialmente, o escopo de software para o produto solicitava a leitura de:

a) Um botão giratório com 4 opções para a seleção da estação do ano.

b) Um sensor de presença instalado na entrada da residência que deveria ser conectado a uma interrupção externa do microcontrolador.

Eram previstas também 3 saídas digitais:

a) Uma saída para o acionamento do motor DC da persiana da sala de estar.

b) Uma saída para a o motor da persiana da sala de jantar.

c) Uma saída ligada a um relê que será ligado a um ventilador presente na sala de estar que só será ligado em estações específicas do ano.

Posteriormente, foi solicitado a conexão do sistema a um sensor de iluminação LDR, conectado a uma das portas analógicas do microcontrolador que, com base nos valores lidos pelo sensor, as lâmpadas a led seriam dimerizadas. E era solicitado também que a velocidade do motor do ventilador deveria ser regida pela leitura de temperatura realizar por uma das entradas analógicas conectada a um termopar instalado externamente a residência.

Resolução

Para a resolução deste problema, podemos prosseguir como orientado no livro a respeito do gerenciamento de tarefas, e criar uma tarefa para tratar cada evento do nosso sistema. Dessa forma, podemos propor:

- **TarefaBotaoGira:** realizar a leitura do botão giratório, de forma a informar outras tarefas sobre qual estação do ano foi selecionada. É uma tarefa simples, de baixa prioridade, que ocupará pouco espaço de memória por simplesmente ler 4 entradas digitais do sistema.
- **TarefaSensorPresenca:** tarefa diretamente conectada a interrupção do sensor de presença. Logo a mesma pode ficar

em modo Bloqueada naturalmente, aguardando um evento de interrupção para o tratamento de seus dados. Por ocorrer eventualmente e tratar diretamente uma interrupção, essa tarefa deve ter maior prioridade.

- **TarefaAcionaSaidas:** essa tarefa monitorará os dados provenientes das tarefas anteriores para decidir qual a melhor configuração das saídas digitais, aguardado a recepção dos dados no modo Bloqueada. É também uma tarefa de alta prioridade, mas que também utiliza pouco espaço de memória.
- **TarefaLDR:** responsável por monitorar os eventos do canal de conversão analógico-digital e atualizar o sinal dimerizado que será enviado para o controle das lâmpadas. Pode possuir alta prioridade, uma vez que os eventos de conversão AD são relativamente lentos. Deve-se definir uma quantidade relevante de memória, pois os sinais lidos podem ser tratados por filtros digitais, que requerem tabelas para amostras históricas.
- **TarefaTemperatura:** responsável por monitorar os eventos de leitura do sensor de temperatura e controlar a velocidade do ventilador de maneira proporcional. Também pode ser atribuído prioridade alta devido à baixa frequência de amostragem da temperatura e, semelhantemente à tarefa anterior, pode ser atribuído uma quantidade razoável de memória devido a possíveis técnicas de amostragem do sinal.

Assim, seguindo o modelo de criação de tarefas do FreeRTOS, podemos implementar cada tarefa do projeto como segue na Tabela 3.2.

Tabela 3.2 | Tarefas para o sistema FreeRTOS do projeto de automação residencial

Função	Nome Tarefa	Prioridade	Stack Memória
vTaskBotaoGira	TarefaBotaoGira	1	32 bytes
vTaskSensorPres	TarefaSensorPresenca	5	32 bytes
vTaskAcionaSaidas	TarefaAcionaSaidas	4	32 bytes
vTaskLDR	TarefaLDR	3	256 bytes
vTaskTemp	TarefaTemperatura	3	256 bytes

Fonte: elaborada pelo autor.

Pode ser utilizada uma **Fila** para enviar os dados de leitura do sensor do botão giratório e do sensor de presença para a tarefa que realiza o acionamento das saídas. Como a tarefa do Sensor de Presença ficará a todo momento Bloqueada aguardando o evento de interrupção, pode ser criado um **Semáforo** entre a interrupção e essa tarefa de forma a sincronizá-la quando o estado do sensor for alterado.

Tanto as tarefas do sensor LDR quanto do sensor de Temperatura podem implementar **Semáforos** sincronizados a interrupção do conversor analógico-digital, que indicará as mesmas que uma nova conversão foi realizada e um novo valor está disponível para tratamento.

Avançando na prática

Refinando a configuração das Tarefas

Descrição da situação-problema

Um dos membros do seu time de *software* realizou a implementação das tarefas conforme o que fora demonstrado no exercício anterior, porém, ele percebeu que havia dois problemas durante os testes funcionais:

- O valor do sensor de temperatura não era coerente, devido à suspeita de falha nas leituras do mesmo por baixa amostragem.
- Sempre que a tarefa de acionamento de saídas era acionada, o sistema era reiniciado automaticamente. Depois de alguma análise, foi identificado um estouro de memória (*stack overflow*) originado pela própria tarefa.

Quais as ações você pode propor a este desenvolvedor para a resolução dos problemas listados?

Resolução da situação-problema

Como a tarefa de leitura do sensor de temperatura está apresentando baixo nível de amostragem, isso pode significar que a tarefa não está sendo executada dentro do tempo de restrição calculado para o correto funcionamento da mesma. Isso pode se dar pelo nível de prioridade da tarefa em relação das demais, que podem estar ocupando muito tempo de processamento da CPU e

liberando janelas rápidas para o processamento das demais tarefas, que pode ser insuficiente para a frequência desejada de amostragem do sensor. A solução mais prática é aumentar a prioridade desta tarefa gradativamente (de 3 para 4 por exemplo) de forma a evidenciar se há uma melhora no desempenho de leitura.

Para solucionar o problema de *stack overflow* da tarefa de acionamento das saídas (TarefaAcionaSaidas), a solução mais correta é aumentar cautelosamente a quantidade de memória de sua *stack* e testar novamente o sistema até que o evento de falha pare de ocorrer.

Faça valer a pena

1. No universo do desenvolvimento de software para sistemas embarcados, um sistema é projetado para receber um estímulo (ou evento), que pode ser de origem interna ou externa, realizar o seu processamento e produzir uma saída. Alguns sistemas trabalham com eventos que possuem restrições de tempo, ou seja, possuem um prazo ou tempo limite para que o estímulo seja processado e que uma saída correspondente seja acionada. Esses tipos de sistemas são denominados Sistemas de Tempo Real.

A respeito dos Sistemas Operacionais de Tempo Real, considere as seguintes afirmações:

( ) São determinísticos por natureza.
( ) Sistemas do tipo *Soft real-time* possuem maior restrição quanto ao tempo das tarefas.
( ) Em projetos de maior complexidade é preferível a utilização da técnica de *Foreground / Background*.
( ) Sistemas do tipo Hard real-time possuem esse nome por serem de difícil modificação.

Considerando a sequência, qual das alternativas abaixo retrata corretamente quais afirmações são falsas e quais são verdadeiras?

a) F, F, F, V.
b) F, V, F, V.
c) F, V, F, F.
d) V, F, F, F.
e) V, F, V, V.

2. Em um sistema multitarefa, tem-se a impressão de que todas as tarefas são executadas ao mesmo tempo, mas como o processador só pode executar uma única tarefa por vez, é realizado o chaveamento entre as tarefas. Este chaveamento ou troca de contexto de tarefas, pode acontecer quando uma tarefa é bloqueada esperando a liberação de um recurso, quando a mesma entra em modo de espera (tarefas periódicas), ou quando uma tarefa é involuntariamente suspensa pelo *kernel*.

Dada a descrição de um sistema multitarefa, qual a função de um escalonador em um sistema de tempo real?

a) Decidir qual é a próxima tarefa a ser executada em determinado momento.
b) Controlar a criação e exclusão de tarefas.
c) Controlar a quantidade de memória das tarefas.
d) Permitir a comunicação entre tarefas.
e) Controlar a velocidade de execução das tarefas.

3. No FreeRTOS, uma Fila não pertence a nenhuma tarefa em específico, sendo que várias tarefas podem compartilhar a mesma fila, tanto para enviar, quanto para receber dados. Cada fila armazena um conjunto finito de itens (*queue lenght* ou comprimento da fila), que pode ter um tamanho fixo de bytes (*item size* ou tamanho do item), e ambos os parâmetros devem ser definidos no momento de criação da fila.

A respeito das Filas, considere as seguintes afirmações:

( ) São usadas normalmente em lógica LIFO (*Last In First Out*).
( ) Ao escrever em uma fila, uma tarefa pode entrar no estado Suspensa se a fila já estiver cheia.
( ) É possível atribuir à tarefa um tempo de espera (*timeout*) para leitura de uma fila.
( ) A tarefa será colocada automaticamente em estado de Pronta quando um item da fila é lido.

Considerando a sequência, qual das alternativas abaixo retrata corretamente quais afirmações são falsas e quais são verdadeiras?

a) F, F, F, V.
b) F, V, F, V.
c) F, V, F, F.
d) V, F, F, F.
e) F, F, V, V.

Seção 3.3

Introdução a linux embarcado

Diálogo aberto

Prezado aluno, na seção anterior abordamos os conceitos principais sobre sistemas embarcados dedicados para o controle em tempo real das aplicações. Nesta seção, analisaremos a outra vertente de sistemas operacionais embarcados, voltada a sistemas genéricos, utilizados comumente em computadores pessoais, mas que também podem ser aplicados a pequenos dispositivos embarcados. Utilizaremos como base aquele que é considerado o principal sistema para esse propósito: o Linux, cujo o mesmo Kernel que é utilizado em grandes servidores e avançados computadores, também é aplicado em relógios, celulares e até em automóveis. A utilização do Linux para sistemas embarcados vem crescendo gradativamente, uma vez que, por ser um sistema aberto suportado por uma grande comunidade de desenvolvedores, várias aplicações e ferramentas criadas para sistemas convencionais são customizadas para dispositivos eletrônicos, como o caso das televisões inteligentes (do inglês *smart TVs*) que empregam recursos gráficos e de navegação na internet, que anteriormente só encontrávamos em computadores pessoais.

Nosso estudo se inicia por compreender o que é o Linux a partir da introdução de seu Kernel, depois ampliaremos nosso foco para enxergar toda a arquitetura que faz parte desse sistema e como cada bloco deste modelo tem um papel fundamental para o desenvolvimento de um projeto embarcado. Logo após, compreenderemos as etapas de funcionamento de um *bootloader* e qual a relação do mesmo com o *hardware* e com o Kernel, passando a seguir por enumerar e apresentar todas as ferramentas que fazem parte de uma *Toolchain*, indispensável para geração de aplicações executáveis em diversas arquiteturas. Por fim, veremos como os diretórios e arquivos são organizados pelo Kernel e quais são as premissas que devemos seguir quando desejamos desenvolver aplicações utilizando Linux Embarcado.

Lembre-se de que você foi escolhido como líder de uma nova equipe focada em pesquisa e em desenvolvimento de sistemas inovadores que serão aplicados a novos produtos da empresa. Sua primeira missão foi o de encontrar soluções para atender a maior demanda de processamento e funcionalidades do sistema, que preveem aplicações multimídia, maior conectividade e interação com o usuário. Nessa etapa, o seu objetivo é promover uma alteração significativa na relação do sistema com o usuário, substituindo as interfaces eletromecânicas, por um display sensível ao toque. Para isso, você decidiu utilizar Linux embarcado nessa nova plataforma de hardware, de forma a agregar funcionalidades multimídia ao projeto, porém grande parte do seu time nunca teve nenhum contato com esse tipo de sistema. Elabore um guia apresentando as principais características de um sistema Linux, atentando-se para aspectos fundamentais, como operação do sistema de bootloader, funcionamento do Kernel Linux, importância da criação de uma toolchain adequada e um breve resumo de como funciona o sistema de arquivos.

O que diferencia um sistema Linux de outros sistemas operacionais? Qual a importância de um *bootloader* para esse sistema? Por que uma *toolchain* deve ser cuidadosamente montada para cada tipo de projeto? Esta seção trará a resposta para todos esses questionamentos, então concentre-se em seus estudos e mostre que você é capaz de superar mais este desafio! Bons estudos!

Não pode faltar

Kernel Linux em sistemas embarcados

O Linux é um dos mais importantes avanços tecnológicos do século XXI. Além de seu impacto no crescimento da internet e seu lugar como uma tecnologia de entrada para uma variedade de dispositivos controlados por computador, o desenvolvimento do Linux tem sido um modelo de como projetos podem superar o que indivíduos e empresas podem fazer sozinhos (NEGUS, 2015).

Em setembro de 1991, Linus Torvalds, então aluno de computação finlandês, desenvolveu a primeira versão do Kernel Linux baseado em outro sistema operacional, o UNIX. Após sua primeira versão, Linus envia a seguinte mensagem para o grupo de discussão de desenvolvedores (SOBELL, 2018, p. 2):

> Olá usuários do *minix*. Eu estou desenvolvendo um sistema operacional gratuito (apenas um hobby, não será grande e profissional como *GNU*) para processadores 386. Venho trabalhando nele desde abril e está começando a ficar pronto. Gostaria de receber comentários sobre coisas que vocês gostam ou não no *minix*, como meu sistema operacional se assemelha um pouco a ele (mesmo *layout* físico do sistema de arquivos, devido a razões práticas, entre outras coisas). Todas as sugestões são bem-vindas, mas eu não prometo que vou implementá-las.

Pesquise mais

O que acha de conhecer um pouco mais sobre o Linux e quais a principais diferenças entre sistemas Desktop e sistemas embarcados? Acesse o vídeo *Linux Embarcado - Uma área que vale a pena!* Disponível em: <https://www.youtube.com/watch?v=6rij9HpNkuo/>. Acesso: 23 jun. 2018.

Kernel Linux

É muito importante ressaltar que quando se faz referência ao sistema Linux, está sendo direcionado a seu **Kernel**, e outras denominações como *Ubuntu, Fedora e Debian,* referem-se a **distribuições** Linux, que constituem a integração do Kernel, das bibliotecas e aplicações, sendo o mais correto denominá-las como sistemas operacionais GNU/Linux, sendo o Linux embarcado o uso do Kernel Linux e de diversos componentes de uso livre para sistemas embarcados (SOBELL, 2018).

Dentre suas principais características podemos destacar o fato de ser um código totalmente aberto e livre de *royalties*, tendo sido portado para mais de 28 arquiteturas distintas, e possui grandes escalabilidade, podendo ser encontrado desde pequenos relógios digitais, até grandes servidores e supercomputadores.

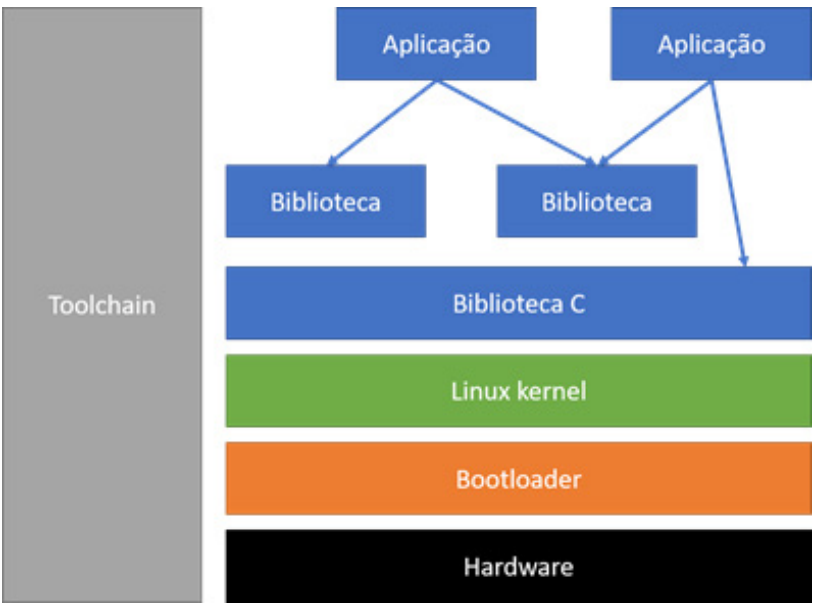
Por possuir o código aberto e suportado por uma comunidade de desenvolvedores muito ativa, uma das grandes vantagens da utilização de Linux para sistemas embarcados é a possibilidade de reuso de componentes, pois é muito fácil encontrar implementações

prontas e validadas para diversas funcionalidades, como drivers, bibliotecas, protocolos, interface gráfica, dentre outros.

Arquitetura Linux

O sistema Linux possui uma arquitetura baseada em uma estrutura de núcleo monolítico, na qual Kernel possui todos os códigos de suporte necessários para sua execução e para as aplicações. O núcleo realiza a abstração do *hardware* dispondo acessos através de chamadas do sistema (FERREIRA, 2008). Na Figura 3.10, temos uma visão simplificada da arquitetura básica de um sistema Linux, muitos desses conceitos são novos no nosso estudo, mas os mesmos serão detalhados nessa seção.

Figura 3.10 | Arquitetura Básica de um sistema Linux



Fonte: elaborada pelo autor.

O ***Hardware*** representa a plataforma computacional que executará o sistema operacional e suas aplicações. O Linux suporta mais de 28 arquiteturas diferentes de processadores, tais como x86, ARM, MIPS e Coldfire, não sendo projetado inicialmente para ser executado por microcontroladores, por não possuírem um elemento vital para o sistema: o MMU (*Memory Management Unit*

ou Unidade de Gerenciamento de Memória). Um sistema Linux customizado de maneira enxuta pode operar com 8MB de memória RAM, porém o recomendável é a alocação de pelo menos 32MB, suportando diversos tipos de armazenamento de memória, como memórias Flash, disco rígido (HD) e cartões SD. Vários protocolos de comunicação e barramentos que são comumente utilizados em sistemas embarcados já são implementados nativamente no Kernel Linux, como exemplo: CAN, SPI, USB, TCP, UDP, Wi-Fi e Bluetooth.

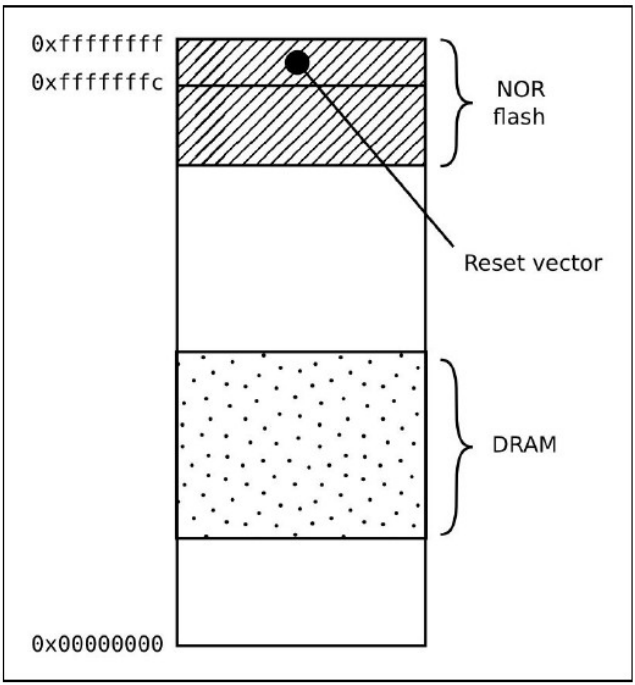
Bootloader em Linux embarcado

O *bootloader* (gerenciador de inicialização) é a parte do sistema embarcado responsável por realizar a inicialização do sistema e carregar o Kernel Linux. Em um sistema Linux embarcado, o *bootloader* tem duas tarefas principais: inicializar o sistema para um nível básico e carregar o Kernel. Quando as primeiras linhas do código do *bootloader* são executadas, após ocorrer a energização ou um reset, o sistema está em um estado mínimo. O controlador de memória RAM ainda não está configurado e, portanto, a memória principal não estaria acessível. Da mesma forma, outras interfaces ainda não foram configuradas nessa etapa, logo o armazenamento via memória Flash, controladores MMC e assim por diante, também não são utilizáveis. Normalmente, os únicos recursos operacionais na inicialização do sistema são um único núcleo de CPU e alguma memória estática no próprio chip. Como resultado, o bootstrap (ponto de partida) do sistema consiste em várias fases de código, cada uma trazendo mais funcionalidades do sistema em operação.

O ato final do *bootloader* é carregar o Kernel na memória RAM e criar uma execução ambiente para isso. Os detalhes da interface entre o *bootloader* e o Kernel são específicos de cada arquitetura, mas em cada caso duas etapas devem ser seguidas. Primeiro, *bootloader* tem que passar um ponteiro para uma estrutura contendo informações sobre a configuração de hardware, e, segundo, ele tem que passar um ponteiro para a linha de comando do Kernel. A linha de comando do Kernel é uma cadeia de texto que controla o comportamento do Linux. Uma vez que o Kernel começou a ser executado, o *bootloader* não é mais necessário e toda a memória que estava sendo utilizada por ele pode ser recuperada. Em sistemas mais simples, basta alocar o *bootloader* na memória

não volátil no vetor de reinicialização do processador (SIMMONDS, 2017). A Figura 3.11, demonstra essa configuração, com o vetor de Reset no endereço 0xfffffffc na extremidade superior de uma área de memória flash. O bootloader é interligado para que haja uma instrução de salto nesse ponto que aponte para o início do código do *bootloader*. A partir desse ponto, o código do bootloader, agora executado na memória flash, pode inicializar o controlador de memória RAM, para que a mesma se torne disponível ao sistema. Depois de totalmente operacional, o *bootloader* pode carregar o Kernel da memória flash para a memória RAM e transferir o controle para ele.

Figura 3.11 | *Bootloader* em um sistema simples



Fonte: Simmonds (2017, p. 103).

Para sistemas mais complexos, o processo de inicialização do sistema através de um *bootloader* é constituído de mais etapas, e dependem ainda mais do tipo de arquitetura de hardware que embarcará o sistema Linux.

Exemplificando

Como exemplo de um processo de inicialização mais complexos, podemos citar o método utilizado para plataformas ARM, como pode ser visto na Figura 3.12 abaixo:

Figura 3.12 | *Bootloader* em plataformas ARM

Fonte: Simmonds (2017, p. 103).

A plataforma que utiliza o processador ARM, geralmente representada por um SoC (*System on Chip* ou Sistema em um Chip) possui um código de inicialização em sua memória ROM interna, responsável por carregar um *bootloader* de primeiro estágio para a memória RAM interna, chamado de SPL (*Secundary Program Loader* ou Carregador de Programa Secundário).

O SPL é responsável por inicializar o *hardware* (especialmente a memória RAM e CPU) e carregar um *bootloader* de segundo estágio para a memória RAM. Esse *bootloader* já é um sistema mais completo e normalmente suporta sistemas de arquivo, interface com portas USB e Ethernet, e protocolos como o TCP/IP. Ele também é responsável por carregar e executar o próprio Kernel do Linux, que então assumirá o controle do sistema, e o *bootloader* será descarregado.

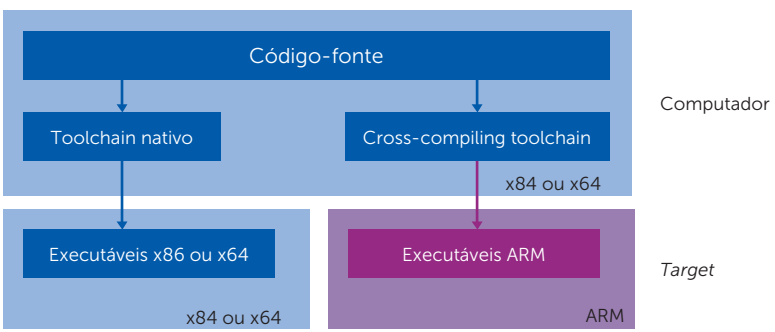
O *bootloader* de primeiro estágio (SPL) é totalmente dependente do *hardware*, logo, cada plataforma e fabricante possuirá o seu código para esse sistema. Para o *bootloader* de segundo estágio, existem vários projetos disponíveis, sejam eles pagos ou de código aberto. Um dos mais conhecidos e utilizados para sistemas embarcados é o U-boot, sendo um dos mais ricos e flexíveis para o desenvolvimento desse tipo de sistemas. Através do **U-boot** é possível visualizar informações do *hardware*, manipular diretamente a memória RAM e a memória Flash, realizar a inicialização diretamente da memória ou via rede. Ele possui interface para leitura de portas USB e cartões SD, podendo manipular diversos sistemas de arquivos como *fat, ext2/3 e jffs2* (YAGHMOUR, 2003).

Toolchain para sistemas embarcados

Um *toolchain* (cadeia de ferramentas) é um conjunto de ferramentas que compilam o código fonte em arquivos executáveis que podem rodar em diversos dispositivos, como o próprio computador que compilou os arquivos ou em dispositivos externos, também chamados de *Targets* (alvos), geralmente incluem um **compilador**, um **linker** (vinculador) e **bibliotecas** que serão evocadas em tempo de execução. Eles são essenciais quando se deseja construir um sistema Linux customizado para sistemas embarcado, no qual se inclui o *bootloader*, o Kernel e o sistema de arquivos. O *toolchain* deve ser capaz de compilar o código escrito em assembly, C e C ++, uma vez que estas são as linguagens usadas nas aplicações de código aberto (SIMMONDS, 2017).

As ferramentas de desenvolvimento normalmente disponíveis em um computador Linux são chamadas de ***toolchain* nativo**, e compilam códigos que serão executados no próprio computador (**host** ou hospedeiro), por exemplo em arquiteturas x86 ou x64. Já para o desenvolvimento de sistemas embarcados normalmente é muito difícil usar um *toolchain* nativo, por demandar muito espaço em disco, capacidade de processamento e memória, logo, para essa finalidade, o mais recomendável é a utilização de um ***cross-compiling toolchain*** (*toolchain* de compilação cruzada), que é executado no computador utilizado como plataforma de desenvolvimento, mas que gera arquivos executáveis para a plataforma *Target*. A Figura 3.13 apresenta um exemplo de uma aplicação compilada para o computador nativo e para um Target em plataforma ARM.

Figura 3.13 | *Bootloader* em plataformas ARM



Fonte: elaborada pelo autor.

Um *Toolchain* é composto pelas seguintes ferramentas:

- **Compilador GCC:** conhecido compilador GNU C, compatível com várias linguagens como C, C++, Fortran, Ada e Java. Pode gerar executáveis para diversas arquiteturas, como ARM, x86, AVR e MIPS.
- **Binutils:** representa um conjunto de ferramentas para manipular arquivos binários para uma arquitetura específica, destacando-se o gerador de código binário baseado em *Assembly* **as**, o *linker ld*, e ferramentas de manipulação e inspeção de arquivos binários **objdump, readelf, size e nm**.
- **Bibliotecas C:** realizam a interface entre as aplicações o Kernel Linux, como a **glibc**, criada pelo próprio projeto GNU e focada em performance e portabilidade, e a **uClibc**, projetada para sistemas embarcados e com maior foco em economia de recursos.
- **Headers (Cabeçalhos) do Kernel:** utilizados para se realizar chamadas no sistema e a interface entre a aplicação e o Kernel.
- **Depurador GDB:** também uma ferramenta desenvolvida pelo projeto GNU que pode ser utilizada para diversas finalidades no sistema, como controlar a execução de uma aplicação, inserir e controlar breakpoints, mudar o valor e estado de variáveis e realizar o *dump* (descarregamento) de memória.

Sistema de Arquivos

O sistema de arquivos Linux é a estrutura na qual todas as informações do sistema são armazenadas. De fato, uma das propriedades que definem o sistema UNIX, no qual o Linux é baseado, é que quase tudo que se precisa identificar em um sistema (dados, comandos, links simbólicos, dispositivos e diretórios) é representado por itens nos arquivos (NEGUS, 2015).

No Linux, os arquivos são organizados dentro de uma hierarquia de diretórios. Cada diretório pode conter arquivos, bem como outros diretórios. Pode-se referir a qualquer arquivo ou diretório usando um caminho completo (por exemplo /home/joe/meuArquivo.txt) ou um caminho relativo (por exemplo, se /home/joe for o diretório do usuário joe, pode-se simplesmente se referir ao arquivo como meuArquivo.txt). Ao se mapear os arquivos e diretórios no Linux, percebe-se que se trata de uma árvore invertida. No topo está o diretório raiz (não deve

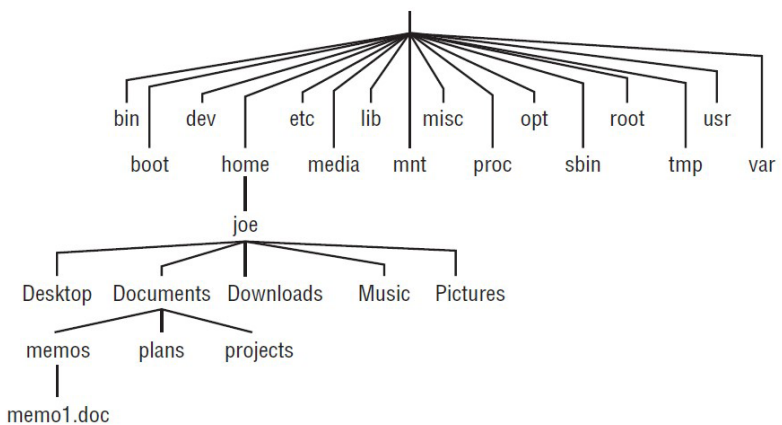
ser confundido com o usuário administrador, ou root), representado por uma barra única (/). A Figura 3.14 apresenta um conjunto de diretórios comuns no sistema Linux, **como bin, dev, home, lib** e **tmp**, para citar alguns. Cada um desses diretórios, bem como diretórios adicionados ao diretório raiz, pode conter subdiretórios. Na Figura 3.14, é possível observar como o sistema de arquivos do Linux é organizado como uma hierarquia e demonstra como diretórios estão conectados, como o diretório /home que contém um subdiretório para o usuário joe. Dentro do diretório joe, estão os subdiretórios Desktop (*Área de Trabalho*), *Documents* (Documentos) e outros.

Reflita

Pelo seu conhecimento de utilização de outros sistemas operacionais, como o *Windows* da empresa *Microsoft*, existe alguma divergência entre a forma com que os diretórios e arquivos são organizados? Quais são as vantagens e desvantagens de cada um?

Quando um sistema de arquivo é montado em um diretório, esse diretório é **chamado** d**e ponto de montagem** ou ***mount point,*** e o conteúdo dele refletirá o conteúdo armazenado no dispositivo de armazenamento. Isso permite que aplicações acessem diretórios e arquivos facilmente independentemente da localização ou do tipo do dispositivo de armazenamento. Um sistema de arquivo específico denominado **root** ou **rootfs** é montando na raiz principal da hierarquia, identificado por /.

Figura 3.14 | Hierarquia do sistema de arquivos Linux



Fonte: Negus (2015, p. 98).

Desenvolvimento de Aplicações

Para o desenvolvimento de aplicações, o processo utilizado em sistemas desktop é basicamente o mesmo dedicado a sistemas embarcados, sendo que se pode reutilizar códigos e bibliotecas sem a necessidade de nenhuma adaptação na maioria dos casos, sempre levando em consideração a limitação de recursos do *Target*.

Assimile

Um sistema Linux embarcado é um sistema Linux comumente utilizado em computadores pessoais, apenas com um conjunto menor e mais enxuto de componentes, entre eles, o próprio kernel, os gerenciadores de rede e o sistema de arquivo. Alguns outros módulos, como o servidor gráfico e navegadores geralmente não são aplicados, uma vez que necessitam de muito recurso e podem ser substituídos por aplicações mais leves em termos computacionais.

A linguagem padrão para o desenvolvimento de aplicações no nível do sistema é a linguagem C, estando a biblioteca C padrão sempre presente em sistemas Linux. Quando se requer um nível maior de abstração, a linguagem C++ ou Java pode ser uma boa escolha também.

Linguagens de script, como *bash* e *shell* são muito úteis em determinadas tarefas, e podem ajudar a diminuir o tempo de desenvolvimento, sem deixar de considerar questões como o consumo de recursos e a segurança da aplicação e do sistema (SOBELL, 2018).

Pesquise mais

Você se interessou pelo conteúdo que aprendeu nesta seção e gostaria de dar mais alguns passos e se tornar um verdadeiro desenvolvedor de sistemas Linux embarcado? Então acesse o artigo *Como se tornar um desenvolvedor de Linux embarcado*. Disponível em: <https://sergioprado.org/como-se-tornar-um-desenvolvedor-de-linux-embarcado/>. Acesso: 23 jun. 2018.

Lembre-se de que agora como líder de uma nova equipe, sua primeira missão é encontrar soluções para novos produtos que utilizarão sistemas operacionais embarcados. Nessa etapa, com o objetivo de utilizar o sistema Linux para novas funcionalidades multimídia, você decidiu elaborar um guia apresentando as principais características de um sistema Linux, focando nos aspectos fundamentais, como o sistema de *bootloader*, funcionamento do Kernel Linux, criação de uma *toolchain* adequada e um breve resumo sobre o sistema de arquivos.

Kernel Linux

Iniciaremos o guia desmistificando o conceito do que é o Linux. Quando tratamos do termo Linux, estamos na verdade nos referindo ao seu Kernel e não às suas distribuições (Ubuntu, Fedora, etc.), que na verdade são implementações gráficas que utilizam o Kernel em si. Outro aspecto importante, é o fato de o projeto Linux ser de código aberto, livre de royalties, e distribuído para mais de 28 arquiteturas diferentes, o que dá a liberdade para qualquer desenvolvedor utilizar, modificar e distribuir suas próprias versões do sistema, respeitando somente algumas regras ditadas pela natureza da sua licença livre.

Arquitetura Linux

O Kernel Linux possui todos os códigos de suporte necessários para sua execução e para as aplicações, sendo que seu núcleo realiza a abstração do *hardware* dispondo acessos através de chamadas do sistema para outros *softwares*. Uma arquitetura simplificada de um sistema Linux conta, além do Kernel, com os seguintes sistemas:

- *Hardware*: plataforma computacional suportada pelo Kernel que embarcará o sistema.
- *Bootloader*: responsável pela inicialização do Hardware e carregamento do Kernel.
- Linux Kernel: sistema operacional em si, responsável por gerenciar as tarefas e processos, e prover uma interface entre as aplicações e o *Hardware*.
- Bibliotecas C: usadas como ferramentas nativas de interface entre as aplicações e o Kernel.

- Aplicações: processos e rotinas de *software* desenvolvidas para o sistema Linux.
- *Toolchain*: plataforma de ferramentas que suporta o desenvolvimento de todas os demais blocos da arquitetura.

Bootloader

Possui como responsabilidade inicializar o sistema para um nível básico e carregar o Kernel. É executado após a energização do sistema, ou quando ocorre uma reinicialização (*reset*). Para sistemas mais simples, é carregado nos primeiros endereços da memória, sendo sempre executado a cada inicialização e, para sistemas mais complexos, é divido em alguns estágios, com os primeiros estágios sendo de responsabilidade de *bootloaders* desenvolvidos pelo fabricante da plataforma de *hardware* e os demais estágios, sendo desempenhados por bootloaders com maior capacidade, alguns inclusive, com possibilidade de acesso a sistemas de arquivos e comunicação com periféricos.

Toolchain

Como o próprio nome diz, uma *toolchain* é uma cadeia de ferramentas utilizada para realizar o desenvolvimento de aplicações, geração e manipulação de arquivos binários para todas as outras partes da arquitetura Linux. Quando uma *toolchain* gera arquivos executáveis para a própria máquina que ela está sendo executada, a denominamos como **nativa** e, quando utilizamos uma máquina hospedeira (host) para a geração de arquivos para outra máquina (**target**), essa *toolchain* é chamada de *cross-compiling toolchain*. Independentemente da plataforma, uma toolchain é composta pelas seguintes ferramentas: Compilador GCC, Binutils, Bibliotecas C, Cabeçalhos do Kernel e Depurador GDB. A *toolchain* deve ser cuidadosamente escolhida para cada projeto, pois, como vimos, ela possui uma relação estreita com o tipo de arquitetura do processador que embarca o sistema bem como a arquitetura do sistema hospedeiro.

Sistema de Arquivos

No Linux, os arquivos são organizados dentro de uma hierarquia de diretórios. Cada diretório pode conter arquivos, bem como outros diretórios. Ao se mapear os arquivos e diretórios no Linux, percebe-

se que se trata de uma árvore invertida. No topo, está o diretório raiz (não deve ser confundido com o usuário administrador, ou **root**), que é representado por uma barra única (/) e é são encontrados um conjunto de diretórios comuns no sistema Linux, como **bin, dev, home, lib** e **tmp**, cada qual com sua funcionalidade específica.

Avançando na prática

Preparando um sistema Linux para uma plataforma de Hardware

Descrição da situação-problema

Vimos que são necessários vários subsistemas e ferramentas para compormos todo o arquebolço necessário para realizar o desenvolvimento em uma plataforma de **hardware** baseada em Linux embarcado, sendo que muitos desses artefatos são encontrados pela própria recomendação do fabricante da plataforma. Neste sentido, faça uma pesquisa e apresente quais são as ferramentas e aplicações que podem ser utilizadas para se realizar o desenvolvimento em uma plataforma Raspberry Pi, contendo a distribuição do Linux e seu *Bootloader*.

Resolução da situação-problema

Distribuição Linux

Após uma breve pesquisa no sistema oficial do projeto Raspberry Pi (Raspbian. Disponível em: <https://www.raspberrypi.org/downloads/raspbian/>. Acesso: 24 jun. 2018), é possível encontrar a versão oficial da distribuição Linux para Raspberry Pi, a **Raspbian**, versão essa baseada em outra distribuição muito utilizada para computadores pessoais, o **Debian**. Após baixar a imagem do sistema operacional, basta gravá-la em um cartão SD e inserir no *hardware* para a execução.

Bootloader

O processo de boot da Raspberry Pi é um pouco diferente dos sistemas convencionais. Inicialmente, quem realiza a primeira etapa de inicialização não é a CPU e sim a GPU, que executa um código presente dentro da memória ROM. Esse código é responsável por

encontrar um arquivo de boot chamado *bootcode.bin* dentro dos dispositivos de armazenamento do *hardware*, como o cartão SD, e o carrega para a memória cache. No segundo estágio, o programa *bootcode.bin* inicializa a memória RAM e carrega um *bootloader* de segundo estágio, chamado *start.elf*, que finalmente carrega a imagem do Kernel e inicia o sistema operacional (Raspberry Pi e o processo de boot. Disponível em: <https://sergioprado.org/raspberry-pi-e-o-processo-de-boot/>. Acesso: 24 jun. 2018).

Faça valer a pena

1. O Linux é um dos mais importantes avanços tecnológicos do século XXI. Além de seu impacto no crescimento da Internet e seu lugar como uma tecnologia de entrada para uma variedade de dispositivos controlados por computador, o desenvolvimento do Linux tem sido um modelo de como projetos podem superar o que indivíduos e empresas podem fazer sozinhos (NEGUS, 2015).

Considere a seguinte sentença incompleta:
O sistema _____ foi desenvolvido inicialmente por _____ em setembro de _____, baseado em outro sistema, o _____.

Qual das alternativas a seguir contém os elementos que, incluídos nas lacunas conforme a sequência, tornam o texto inteiro verdadeiro?

a) Unix, Linus Torvalds, 1997, Linux.
b) Linux, Linus Torvalds, 1991, Unix.
c) Linux, Linux Torvalds, 1991, Minix.
d) Minix, Linus Torvalds, 1997, Unix.
e) Linux, Linux Torvalds, 1997, Minix.

2. Tanto os sistemas computacionais Linux baseados em Desktops quanto aqueles embarcados em sistemas eletrônicos mais enxutos, dependem de outro sistema chamado de *bootloader* (gerenciador de inicialização), responsável por dar os primeiros passos de inicialização do conjunto antes que o Kernel assuma definitivamente o controle sobre as aplicações e o hardware.

A respeito dos *bootloader* para sistemas embarcados, considere as seguintes afirmações:

( ) Possuem somente a função de carregar o Kernel.
( ) Podem ocorrer em um único estágio ou em mais estágios.
( ) São sempre executados quando o sistema é reinicializado.
( ) Alguns *bootloader* possuem funções que o próprio Kernel exerce, como manipulação de arquivos e leitura de entradas.

Considerando a sequência, qual das alternativas abaixo retrata corretamente quais afirmações são falsas e quais são verdadeiras?

a) F, F, F, V.
b) F, V, F, V.
c) F, V, V, F.
d) V, F, F, F.
e) F, F, V, V.

3. Um *toolchain* (cadeia de ferramentas) é um conjunto de ferramentas que compilam o código fonte em arquivos executáveis que podem rodar em dispositivos em diversos dispositivos, como o próprio computador que compilou os arquivos ou em dispositivos externos, também chamados de *Targets* (alvos).

Qual a finalidade dos sistemas denominados *cross-compiling toolchain*?

a) Compilar e manipular arquivos binários tanto para a máquina hospedeira, quanto para o *Target*.
b) Simplesmente gerar ferramentas para manipulação de arquivos binários para o *Target*.
c) Possuem a mesma finalidade que os chamados *toolchain* nativos.
d) Compilar e manipular arquivos binários somente para o *Target*.
e) Depurar códigos criados para máquina hospedeira e para o Target.

Referências

Como se tornar um desenvolvedor de Linux embarcado. Disponível em: <https://sergioprado.org/como-se-tornar-um-desenvolvedor-de-linux-embarcado/>. Acesso: 23 jun. 2018.

FERREIRA, R. **Linux**: guia do administrador do sistema. São Paulo. Novatec, 2008.

Linux Embarcado - Uma área que vale a pena!. Disponível em: <https://www.youtube.com/watch?v=6rij9HpNkuo/>. Acesso: 23 jun. 2018.

NEGUS, C. **Linux Bible**. 9. ed. Indianapolis, USA. Wiley, 2015.

Raspberry Pi e o processo de boot. Disponível em: <https://sergioprado.org/raspberry-pi-e-o-processo-de-boot/>. Acesso: 24 jun. 2018.

Raspbian. Disponível em: <https://www.raspberrypi.org/downloads/raspbian/>. Acesso: 24 jun. 2018.

SIMMONDS, C. **Mastering Embedded Linux Programming**. 4. ed. Birmingham, UK. Packt Publishing Ltd., 2017.

SOBELL, M. **A Practical Guide to Linux**. 4. ed. Boston, USA. Addison-Wesley 2018.

YAGHMOUR, K. **Building Embedded Linux Systems**. Sebastopol, USA. O'Reilly, 2003.

Desenvolvimento de projetos embarcados

Convite ao estudo

Caro aluno, nesta unidade do livro, vamos nos aprofundar no estudo sobre sistemas embarcados microprocessados, a partir de exemplos práticos de aplicações envolvendo sistemas operacionais convencionais e aqueles denominados de tempo real.

Corriqueiramente, já estamos habituados a operar tais sistemas operacionais, porém com uma visão de usuário. Agora compreenderemos como desenvolver aplicações para esses sistemas utilizando algumas de suas principais ferramentas.

Nesta unidade, vamos conhecer e aplicar conceitos de sistemas operacionais em projetos reais envolvendo eletrônica embarcada. Inicialmente, veremos como criar aplicações para sistemas operacionais de tempo real, que possuem rígidas restrições de execução, o que leva o desenvolvedor a ter todo o controle sobre as tarefas criadas e como se comunicarão. Logo após, na segunda seção, teremos um panorama de desenvolvimento em sistemas Linux que, diferentemente dos sistemas de tempo real, não há um controle explicito sobre a forma de execução das tarefas, mas que nos entrega um bom nível de abstração para criação de aplicações de maior interação com o usuário, como aplicações gráficas, manipulação de arquivos, e comunicação em rede. Por fim, na última seção, após dominado os principais conceitos de sistemas embarcados clássicos, teremos uma breve visão das principais tendências para esta área, como sistemas embarcados críticos, conectividade, e criação de distribuições customizadas de sistemas operacionais.

Ao final desta unidade, após dominarmos os principais conceitos sobre sistemas de tempo real, bem como

sistemas genéricos embarcados, como o Linux, deveremos estar aptos a desenvolver aplicações tanto com sistemas determinísticos, como aqueles utilizados em controle industrial, quanto sistemas de maior abstração, como os voltados a aplicações multimídia.

Após todo o processo de pesquisa e de aquisição de conhecimento, você e sua equipe de desenvolvimento de *software* estão prontos para realizar as primeiras implementações de *software* no projeto. A primeira etapa do desenvolvimento focará na arquitetura do sistema de controle em tempo real e na construção das tarefas do sistema operacional. Já na segunda etapa, seu time será direcionado a desenvolver toda a interface gráfica de interação com o usuário, bem como a manipulação de *hardware* e arquivos. Por fim, após a conclusão e teste de todas as funcionalidades deste novo produto, você deverá realizar projeções de novas tecnologias que poderão ser embarcadas nessa nova plataforma que está sendo criada pela empresa.

Na sua opinião, quais critérios devemos considerar no momento de escolha de um sistema operacional embarcado? Como criar sistemas que ao mesmo tempo executem processos dentro de intervalos rígidos de tempo, mas também possam propiciar uma boa interface de interação com o usuário? Esta última unidade será sua oportunidade de demonstrar todo o conhecimento adquirido deste curso, então boa sorte e ótimos estudos!

Seção 4.1

Projetos de software utilizando sistemas operacionais de tempo real

Diálogo aberto

Caro aluno, nesta seção, voltaremos ao estudo de sistemas operacionais de tempo real, desta vez em maior profundidade. Lembre-se de que assim como em um projeto do módulo de segurança para um automóvel, nas aplicações de tempo real todo o controle e parametrização das tarefas é de responsabilidade do próprio desenvolvedor, uma vez que a priorização de execução e a utilização de mecanismos de comunicação são peças fundamentais para estes projetos.

Implementaremos, nesta seção, alguns exemplos de aplicações que utilizem o sistema operacional FreeRTOS e seus mecanismos, de forma a concretizar o conteúdo teórico apresentado sobre sistemas de tempo real.

Inicialmente, será apresentada a forma mais simples de se criar uma tarefa, considerando sua prioridade e ocupação de memória. Logo após, entenderemos o comportamento do escalonador e do kernel, quando duas tarefas estão concorrendo pela ocupação do processador. Em um segundo momento, serão implementados sistemas que utilizam semáforos binários para realizar o sincronismo de execução de duas tarefas distintas, bem como filas para que as mesmas tarefas possam trocar dados e informação da forma mais segura possível. Ao final desta experiência, agregaremos o conhecimento de todos os conceitos abordados em um exemplo prático, muito próximo das aplicações reais de tempo real.

Lembre-se de que agora você ocupa um papel de destaque na empresa de automação residencial e, logo após apresentar os primeiros estudos sobre sistemas embarcados microprocessados, você e seu time estão aptos a desenvolver as primeiras aplicações para essa nova versão de *software* do sistema.

Neste primeiro momento, você deve elaborar um código-fonte baseado no FreeRTOS que represente uma das possíveis

implementações para tarefas que executam todas as funcionalidades solicitadas e implementadas no primeiro protótipo, que fora desenvolvido utilizando microcontroladores e uma metodologia de *software* utilizando laço infinito. Para isso, você deve considerar a utilização dos principais recursos desse sistema operacional, como a criação de tarefas e suas priorizações, manipulação de semáforos binários e utilização de filas para troca de informação entre as tarefas.

Como poderemos substituir um sistema de *software* baseado em loop infinito, por este baseado em tarefas escalonadas? Como realizar uma comunicação mais eficiente entre as tarefas, sem se valer de variáveis globais por exemplo? Dedique-se ao estudo dos exemplos que veremos nesta seção para que estas e outras dúvidas sejam prontamente solucionadas e que seu aprendizado sobre sistemas de tempo real seja consolidado. Então bons estudos e vamos juntos!

Não pode faltar

Introdução

Agora que já compreendemos os conceitos básicos que definem os sistemas operacionais embarcados, em especial, aqueles dedicados a sistemas de tempo real, como podemos criar aplicações que realmente se beneficiam das ferramentas e diretivas disponibilizados por esses sistemas?

Após estudarmos os principais sistemas operacionais de tempo real voltados a operação de microprocessadores, podemos agora iniciar a criação de rotinas e tarefas (do inglês *tasks*) que, além de possuírem restrições de tempo de execução e de utilização de memória, também podem se comunicar umas com as outras por meio mecanismos específicos, como filas e semáforos. A definição da prioridade de cada tarefa, bem como dos momentos exatos de bloquear ou liberar uma tarefa de sua execução determinam o sucesso da arquitetura de um sistema em tempo real.

Nesta seção, implementaremos diversos exemplos utilizando como base o sistema FreeRTOS, criado por *Richard Barry* e

considerado o sistema operacional de tempo real de código aberto mais utilizado em todo o mundo (consulte em: <https://www.freertos.org/RTOS.html>. Acesso em: 10 jun. 2018).

Criação de Tarefas

As tarefas dentro de um sistema operacional de tempo real se comportam como programas isolados, sempre sendo implementadas através de um laço infinito, e geralmente não retornam nenhuma informação.

No primeiro exemplo, implementaremos uma única tarefa, que terá como simples função o envio de uma mensagem ao terminal informando a quantidade de vezes que foi executada e após entrará em estado de bloqueio por um segundo. A Figura 4.1 demonstra a implementação do *software* de criação da tarefa, que apresenta a seguintes estruturas:

1. Inclusão da biblioteca padrão **stdio.h** para utilizar a função *printf* e inclusão das bibliotecas para utilizar funções do *kernel* FreeRTOS e das interfaces de tarefa.
2. Criação de uma macro para definir a prioridade da tarefa, que para este caso foi definida como uma acima da prioridade da tarefa *idle* (vazia). Criação de uma macro para definir o tempo de atraso da tarefa, relativo à quantidade de *ticks* (batidas) do sistema operacional durante um milissegundo. O valor do *tick* é utilizado como base temporal para todo o sistema, assim como a batida de um relógio.
3. Protótipo para a função que representará a tarefa.
4. Função principal da linguagem C.
5. Criação da tarefa.
6. Função de inicialização do escalonador, que gerenciará as tarefas. A partir desse momento, o sistema operacional deve começar a acionar cada tarefa criada, e esta função não deve retornar.
7. Implementação de segurança de um laço infinito vazio, caso o escalonador retorne.

8. Declaração da tarefa propriamente dita.
9. Declaração da variável que será utilizada para realizar a contagem do número de execuções da tarefa. Declaração da variável que ditará o tempo de atraso da tarefa, ou seja, o tempo que a mesma ficará bloqueada até ser executada novamente.
10. Impressão da mensagem com a contagem atual e chamada da função de atraso.

Figura 4.1 | *Software* de criação de uma simples tarefa utilizando o FreeRTOS

```
1  #include <stdio.h>      /* Biblioteca Padrão */
   #include "FreeRTOS.h"   /* Biblioteca do Kernel */
   #include "task.h"       /* Biblioteca para utilização de tarefas*/

   /*Prioridade da Tarefa, sabendo que tskIDLE_PRIORITY tem prioridade 0*/
   #define PRIORIDADE_EX_TASK      ( tskIDLE_PRIORITY + 1 )
2  /*Valor de 1000ms ou 1s convertido para ticks do sistema*/
   #define TEMPO_ATRASO_TAREFA_1   ( 1000 / portTICK_PERIOD_MS )

3  static void ExemploTask_1 (void *pvParameters);

4  void main( void )
   {
       /*Criando a primeira Tarefa*/
5      xTaskCreate( ExemploTask_1, /*Função que implementa a tarefa*/
                    "ExamploTask_1", /*Nome da Tarefa*/
                    configMINIMAL_STACK_SIZE, /*Tamanho da stack*/
                    NULL, /*Parâmetros passados para a Tarefa*/
                    PRIORIDADE_EX_TASK, /*Prioridade da Tarefa*/
                    NULL ); /*Função de manipulação da Tarefa*/

6      vTaskStartScheduler();  /*Inicia o Escalonador de Tarefas*/

7      for( ;; );/*Se tudo correr bem, a execução ficará no Escalonador*/
   }

8  static void ExemploTask_1 (void *pvParameters)
   {
9      unsigned int contador = 0;
       const TickType_t xBlockTime = TEMPO_ATRASO_TAREFA_1;

       for( ;; )
       {
10         printf( "Nova execucao da Tarefa, numero: %d\r\n", contador++);
           fflush( stdout );
           /*Bloqueia a Tarefa por 1 segundo*/
           vTaskDelay(xBlockTime);
       }
   }
```

Fonte: captura de tela do software Notepad++, feita pelo autor.

A tarefa *Idle* é sempre executada quando todas as tarefas estão em estado bloqueada, e ela possui algumas funções específicas, como deletar tarefas marcadas para exclusão, checar se outra tarefa já está pronta para ser executada e monitorar o comando de estado de baixa energia (do inglês *sleep mode*). Essa tarefa sempre possui a menor prioridade possível (geralmente zero), definida pela macro **tskIDLE_PRIORITY**, utilizada como referência para atribuição de prioridade para as demais tarefas (BARRY, 2016).

Para criarmos uma tarefa, utiliza-se a função **xTaskCreate**, passando como informação os seguintes parâmetros: a função que representará a tarefa, o nome da função (para finalidades de depuração), o tamanho de sua *stack*, sua prioridade, a função de manipulação e parâmetros de criação. Já a função **vTaskStartScheduler** tem por objetivo iniciar o escalonador, que será responsável por gerenciar a execução das tarefas. Por fim, a função **vTaskDelay** coloca a tarefa em estado bloqueado por uma quantidade determinada de *ticks*. Como é mais intuitivo a utilização de intervalos reais de tempo, utiliza-se a macro **portTICK_PERIOD_MS** para o cálculo da quantidade de ticks necessários para a representação do intervalo de tempo desejado.

Assimile

Lembre-se de que, diferentemente dos projetos com microcontroladores, em que a rotina principal executava todas suas funções em laço infinito, em um sistema de tempo real o controle de execução das tarefas é repassado ao kernel, no caso do FreeRTOS evocado por seu escalonador.

No próximo exemplo (Figura 4.2), criaremos duas tarefas distintas, que possuem a mesma função desempenhada pela tarefa do exemplo anterior, ou seja, ser executada a cada intervalo conhecido de tempo. Dessa vez, adotaremos valores diferentes de tempo de bloqueio para as tarefas, o que levará as mesmas a competirem pela execução periodicamente.

Figura 4.2 | Execução de duas tarefas com o FreeRTOS

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
/*Tarefa 1 com prioridade 1 e Tarefa 2 com prioridade 2*/
#define PRIORIDADE_EX_1_TASK    ( tskIDLE_PRIORITY + 1 )
#define PRIORIDADE_EX_2_TASK    ( tskIDLE_PRIORITY + 2 )
/*Valor de 1000ms de atraso para Tarefa 1 e 500ms para a 2*/
#define TEMPO_ATRASO_TAREFA_1   ( 1000 / portTICK_PERIOD_MS )
#define TEMPO_ATRASO_TAREFA_2   ( 500  / portTICK_PERIOD_MS )
static void ExemploTask_1 (void *pvParameters);
static void ExemploTask_2 (void *pvParameters);

void main( void ) {
    /*Criando as tarefas*/
    xTaskCreate( ExemploTask_1,
                 "ExamploTask_1",
                 configMINIMAL_STACK_SIZE,
                 NULL,
                 PRIORIDADE_EX_1_TASK,
                 NULL );
    xTaskCreate( ExemploTask_2,
                 "ExamploTask_2",
                 configMINIMAL_STACK_SIZE,
                 NULL,
                 PRIORIDADE_EX_2_TASK,
                 NULL );
    vTaskStartScheduler();
    for( ;; );
}
static void ExemploTask_1 (void *pvParameters) {
    unsigned int contador = 0;
    const TickType_t xBlockTime = TEMPO_ATRASO_TAREFA_1;
    for( ;; )
    {
        printf( "Nova execucao da Tarefa 1, contador: %d\r\n",
            contador++);
        fflush( stdout );
        vTaskDelay(xBlockTime);
    }
}
static void ExemploTask_2 (void *pvParameters) {
    unsigned int contador = 0;
    const TickType_t xBlockTime = TEMPO_ATRASO_TAREFA_2;
    for( ;; )
    {
        printf( "Nova execucao da Tarefa 2, contador: %d\r\n",
            contador++);
        fflush( stdout );
        vTaskDelay(xBlockTime);
    }
}
```

Fonte: captura de tela do *software* Notepad++, feita pelo autor.

Perceba que neste exemplo a tarefa 1 é executada a cada 1 segundo e possui a prioridade igual a 1, e a tarefa 2 é executada a cada 500 milissegundos, e possui prioridade igual a 2. A cada duas execuções da segunda tarefa, a mesma se encontrará em estado de concorrência com a tarefa 1, porém essa situação é resolvida por meio da análise do valor de prioridade de cada tarefa. Como a tarefa 2 é mais prioritária que a tarefa 1, a mesma é executada primeiro e volta ao estado de bloqueio, liberando o *kernel* para a execução da primeira tarefa. A diferença no tempo de execução entre as duas tarefas nessa situação vai depender exclusivamente da velocidade do processador, mas, devido ao código enxuto do *kernel* do sistema FreeRTOS, essa troca pode ser realizada na escala de nanossegundos para alguns cenários.

Semáforos binários

Como sabemos, os semáforos são ótimos recursos de sincronização disponibilizados pelo sistema operacional para realizar a comunicação entre tarefas, liberando do estado de bloqueio quando um determinado evento ocorre e é transmitido (BARRY, 2016). Na Figura 4.3, temos uma implementação semelhante a anterior, contendo duas tarefas que enviam mensagens periodicamente, porém nesse novo cenário um semáforo binário é criado, e a tarefa 2 aguarda o recebimento do sinal do semáforo para sair do estado de bloqueio e enviar sua mensagem. A tarefa 1 continua enviando uma mensagem a cada um segundo, e agora sempre que o valor de contagem de mensagens enviadas for um número ímpar, a tarefa também envia o sinal do semáforo para a segunda tarefa.

Figura 4.3 | Comunicação entre tarefas utilizando semáforo

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
/*Tarefa 1 com prioridade 1 e Tarefa 2 com prioridade 2*/
#define PRIORIDADE_EX_1_TASK        ( tskIDLE_PRIORITY + 1 )
#define PRIORIDADE_EX_2_TASK        ( tskIDLE_PRIORITY + 2 )
/*Valor de 1000ms de atraso para Tarefa 1 e 500ms para a 2*/
#define TEMPO_ATRASO_TAREFA_1   ( 1000 / portTICK_PERIOD_MS )
static void ExemploTask_1 (void *pvParameters);
static void ExemploTask_2 (void *pvParameters);
SemaphoreHandle_t xSemaphore; /*Declara um novo semáforo*/
```

```
void main( void ){
    xSemaphore = xSemaphoreCreateBinary();
    xTaskCreate( ExemploTask_1, "ExamploTask_1",
        configMINIMAL_STACK_SIZE, NULL, PRIORIDADE_EX_1_TASK, NULL );
    xTaskCreate( ExemploTask_2, "ExamploTask_2",
        configMINIMAL_STACK_SIZE, NULL, PRIORIDADE_EX_2_TASK, NULL );
    vTaskStartScheduler();
    for( ;; );
}
static void ExemploTask_1 (void *pvParameters){
    unsigned int contador = 0;
    const TickType_t xBlockTime = TEMPO_ATRASO_TAREFA_1;
    for( ;; ) {
        printf( "Nova execucao da Tarefa 1, contador: %d\r\n", contador++);
        fflush( stdout );
        if(contador % 2) {/*Envia o sinal do semáforo se o contador impar*/
            if( xSemaphoreGive( xSemaphore ) != pdTRUE ) {
                printf( "Falha ao criar o semáforo" );
                fflush( stdout );
            }
        }
        vTaskDelay(xBlockTime);
    }
}
static void ExemploTask_2 (void *pvParameters) {
    unsigned int contador = 0;
    for( ;; ) {
        xSemaphoreTake( xSemaphore, portMAX_DELAY );/*Aguarda o semáforo*/
        printf( "Nova execucao da Tarefa 2, contador: %d\r\n", contador++);
        fflush( stdout );
    }
}
```

Fonte: captura de tela do *software* Notepad++, feita pelo autor.

Perceba que, para a utilização do mecanismo de semáforo (**xSemaphore**), é necessária a referência à biblioteca **semphr.h**. Assim que um novo semáforo é declarado, utiliza-se a função **xSemaphoreCreateBinary** para alocar um novo semáforo binário, e sempre que se deseja enviar um sinal desse semáforo para outra tarefa, é utilizada a função **xSemaphoreGive** passando o semáforo em questão como argumento. Já do lado da tarefa que aguardará o semáforo para continuar seu processamento, é utilizada a função **xSemaphoreTake**, considerando o mesmo semáforo que foi utilizado para o envio como argumento, e também um parâmetro de atraso (do inglês *delay*), que determina por quanto tempo ea tarefa deverá ficar no estado bloqueada aguardando a chegada do sinal e, ultrapassado este tempo, a tarefa retomará o seu processamento. Caso se deseje bloquear a tarefa indefinidamente, aguardando sempre o sinal do semáforo, é utilizada como parâmetro para a função a macro **portMAX_DELAY**.

Pesquise mais

Os semáforos binários são ótimos mecanismos quando se deseja sincronizar a operação de duas tarefas, onde uma delas é a emissora do sinal e a outra é a receptora. Em alguns casos, é necessário a comunicação entre mais de duas tarefas, por exemplo, para o controle de acesso de um recurso de *hardware* do sistema. Para isso é utilizado um tipo especial de semáforo, o chamado **Mutex**. O que acha de conhecer mais sobre esse recurso com um exemplo real? Acesse: *Como utilizar mutex no FreeRTOS do esp32*. Disponível em: <https://www.dobitaobyte.com.br/como-utilizar-mutex-no-freertos-do-esp32/>. Data de acesso: 3 jul. 2018.

Comunicação entre tarefas com filas

Quando desejamos sincronizar duas tarefas, de forma que a execução de uma seja conectada ao comando de outra sabemos que podemos utilizar um semáforo binário. Mas quando é necessário a troca de dados entre tarefas, qual a melhor forma de se realizar esse procedimento?

Uma das alternativas seria a criação de variáveis globais, cujo contexto de acesso permita que ambas as tarefas leiam e escrevam sobre tais variáveis, porém, como o sistema operacional de tempo real é multitarefa, não é possível garantir a integridade da informação da variável devido a seu acesso assíncrono.

Exemplificando

Considere que é declarada uma variável global para operar como contadora de interações em um laço de execução de uma tarefa, e essa mesma variável é manipulável por uma segunda tarefa. Se, durante uma interação do laço da primeira tarefa, a mesma for bloqueada e a segunda tarefa assumir o controle e alterar a variável global, quando o processo retornar à primeira tarefa, a quantidade de ciclos a serem executados pelo laço não será mais o mesmo do momento em que a tarefa foi interrompida, o que certamente é muito grave, pois altera a lógica planejada inicialmente para essa tarefa.

Para contornar esse problema de assincronicidade, o sistema operacional FreeRTOS implementa as **filas** (do inglês queues). A partir delas é possível definir um pacote de dados que será trocado entre tarefas, bem como determinar qual a quantidade de pacotes que a fila poderá armazenar (BARRY, 2016). A Figura 4.4 apresenta um exemplo de um *software* com a criação de duas tarefas que trocam informação através de uma fila. Inicialmente, uma fila é declarada através do tipo **QueueHandle_t**, e, posteriormente, já dentro da função principal a função **xQueueCreate**, é utilizada para alocar uma nova fila, passando como parâmetros a quantidade máxima de elementos que está fila poderá armazenar, e também o tamanho de cada elemento. Na Tarefa 1, a função **xQueueSend** é utilizada para enviar o valor da variável **contador** para a segunda tarefa, passando como parâmetros a fila criada, o endereço da variável contador, e também o tempo que se deseja esperar para enviar a informação antes de prosseguir com a execução da tarefa, sendo que esse tempo pode ser utilizado em casos que a fila venha a estar cheia, e a tarefa aguarda por um determinado período, para que algum elemento seja consumido por outra tarefa, e que um novo espaço vago seja disponibilizado para o armazenamento do dado.

Figura 4.4 | Troca de dados entre tarefas através de filas

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#define PRIORIDADE_EX_1_TASK        ( tskIDLE_PRIORITY + 1 )
#define PRIORIDADE_EX_2_TASK        ( tskIDLE_PRIORITY + 2 )
#define TEMPO_ATRASO_TAREFA ( 1000 / portTICK_PERIOD_MS )
static void ExemploTask_1 (void *pvParameters);
static void ExemploTask_2 (void *pvParameters);
QueueHandle_t xQueue; /*Declara uma nova Fila*/

void main( void ) {
    /*Cria uma Fila com 5 itens que envia itens com tamanho de 32 bits*/
    xQueue = xQueueCreate( 5, sizeof( uint32_t ) );
    xTaskCreate( ExemploTask_1, "ExemploTask_1",
        configMINIMAL_STACK_SIZE, NULL, PRIORIDADE_EX_1_TASK, NULL );
    xTaskCreate( ExemploTask_2, "ExemploTask_2",
        configMINIMAL_STACK_SIZE, NULL, PRIORIDADE_EX_2_TASK, NULL );
    vTaskStartScheduler();
    for( ;; );
}
```

```
static void ExemploTask_1 (void *pvParameters) {
    uint32_t contador = 0;
    const TickType_t xBlockTime = TEMPO_ATRASO_TAREFA;
    for( ;; ) {
        printf( "Nova execucao da Tarefa 1, contador: %d\r\n", contador++);
        fflush( stdout );
        /*Envia o valor do contador para a Tarefa 2 via Fila*/
        xQueueSend( xQueue, &contador, (TickType_t) 0 );
        vTaskDelay(xBlockTime);
    }
}
static void ExemploTask_2 (void *pvParameters) {
    uint32_t contador = 0;
    for( ;; ){
        /*Aguarda receber o valor do contador da Tarefa 1 e incremente 1*/
        xQueueReceive( xQueue, &contador, portMAX_DELAY );
        printf( "Nova execucao da Tarefa 2, contador: %d\r\n", contador + 1);
        fflush( stdout );
    }
}
```

Fonte: captura de tela do *software* Notepad++, feita pelo autor.

A Tarefa 2 aguarda o recebimento de um novo item da fila através da função **xQueueReceive**, que passa como parâmetro a fila em questão, o endereço da variável que receberá o novo elemento da fila, e o tempo que a tarefa irá aguardar até que seja retomado seu processamento. Semelhantemente ao semáforo, se a macro **portMAX_DELAY** for utilizada, a tarefa aguarda o recebimento de um novo item indefinidamente.

Reflita

Como podemos observar, as filas possuem uma limitação ditada pela quantidade de itens que a mesma pode armazenar, sendo que, ao atingir esse limite, a fila não armazena mais nenhum outro elemento. Uma solução seria criar filas com uma quantidade muito alta de elementos, mas por que está alternativa não é recomendável, especialmente em se tratando de aplicações em sistemas embarcados?

Projeto integrado em sistemas embarcados

Nesse último exemplo, apresentado na Figura 4.5, consolidaremos todos os conceitos adquiridos até aqui em uma aplicação simples que resume a criação de tarefas, definição de prioridades e a utilização de semáforos e filas.

Figura 4.5 | Exemplo completo de um sistema de tempo real

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#define PRIORIDADE_EX_1_TASK        ( tskIDLE_PRIORITY + 1 )
#define PRIORIDADE_EX_2_TASK        ( tskIDLE_PRIORITY + 2 )
#define TEMPO_ATRASO_TAREFA ( 1000 / portTICK_PERIOD_MS )
static void ExemploTask_1 (void *pvParameters);
static void ExemploTask_2 (void *pvParameters);
QueueHandle_t xQueue;
SemaphoreHandle_t xSemaphore;

void main( void ){
    xQueue = xQueueCreate( 5, sizeof( uint32_t ) );
    xSemaphore = xSemaphoreCreateBinary();
    xTaskCreate( ExemploTask_1, "ExemploTask_1",
        configMINIMAL_STACK_SIZE, NULL, PRIORIDADE_EX_1_TASK, NULL );
    xTaskCreate( ExemploTask_2, "ExemploTask_2",
        configMINIMAL_STACK_SIZE, NULL, PRIORIDADE_EX_2_TASK, NULL );
    vTaskStartScheduler();
    for( ;; );
}
static void ExemploTask_1 (void *pvParameters){
    uint32_t contador = 0;
    const TickType_t xBlockTime = TEMPO_ATRASO_TAREFA;
    for( ;; ){
        printf( "Enviando semáforo e aguardando chegada da fila\r\n");
        fflush( stdout );
        xSemaphoreGive(xSemaphore);
        xQueueReceive( xQueue, &contador, portMAX_DELAY );
        printf( "Dado da fila recebido, novo valor do contador: %d\r\n", contador);
        fflush( stdout );
        vTaskDelay(xBlockTime);
    }
}
static void ExemploTask_2 (void *pvParameters){
    uint32_t contador = 0;
    for( ;; ){
        xSemaphoreTake( xSemaphore, portMAX_DELAY );
        printf( "Semáforo recebido, enviando dado para fila\r\n");
        fflush( stdout );
        contador++;
        xQueueSend( xQueue, &contador, (TickType_t) 0 );
    }
}
```

Fonte: captura de tela do *software* Notepad++, feita pelo autor.

Dessa vez, é criado um semáforo binário utilizado pela Tarefa 1 para sincronizar sua execução a Tarefa 2, e também uma fila utilizada para enviar pacotes de informação da Tarefa 2 de volta para a Tarefa 1. Em um primeiro momento, a Tarefa 1 envia um sinal do semáforo para a Tarefa 2 e aguarda o recebimento de dados enviados pela Tarefa 2 a partir de uma fila, assim que o recebe, a tarefa aguarda bloqueada por um intervalo de um segundo. A Tarefa 2 está normalmente bloqueada e, quando recebe o sinal do semáforo, volta a execução e envia para a primeira tarefa o valor do contador incrementado, através do mecanismo da fila.

Nesta seção, vimos como utilizar o sistema operacional FreeRTOS para projetar sistemas reais com requisitos de tempo real. Inicialmente, aprendemos a criar tarefas que representam aplicações autônomas, logo após vimos como utilizar mecanismos para realizar a comunicação entre as mesmas, como os semáforos e as filas. Por fim, reunimos todos esses conceitos em um exemplo completo, que nos deu uma breve demonstração de códigos utilizados em projetos profissionais nesta área.

Sem medo de errar

Lembre-se de que agora, como líder do time de desenvolvimento, você e seu time estão empenhados em remodelar o *software* do primeiro protótipo, desenvolvido para microcontroladores, e migrar suas funcionalidades para um sistema de tempo real utilizando o FreeRTOS. Você deve elaborar um código-fonte baseado no FreeRTOS que represente uma das possíveis implementações para as tarefas que executam as funcionalidades solicitadas e implementadas no primeiro protótipo, que fora desenvolvido utilizando laço infinito. Para isso, você deve considerar a utilização dos principais recursos desse sistema operacional, como a criação de tarefas e suas priorizações, manipulação de semáforos binários e utilização de filas para troca de informação entre as tarefas.

Relembrando o escopo inicial do projeto microcontrolado, o sistema possui um botão giratório com quatro opções para escolha da estação do ano, um sensor de presença, uma saída para a persiana da sala de estar e outra para a sala de jantar, e finalmente, uma saída para um ventilador. O acionamento das saídas só ocorre com o acionamento do sensor de presença com a seguinte configuração para cada estação: a) Verão: Persianas acionadas durante 5 segundos. Ventilador ligado; b) Inverno: Persianas acionadas durante 1 segundo. Ventilador desligado; c) Primavera: Persianas acionadas durante 3 segundos. Ventilador ligado; e d) Outono: Persianas acionadas durante 2 segundos. Ventilador desligado. Em um segundo momento, foram adicionados 2 sensores analógicos: um LDR que dimeriza as lâmpadas em uma razão de 2 para 1 e um sensor de temperatura, que controla a velocidade do ventilador em uma razão de 3 para 1.

O primeiro passo para a resolução desse problema consiste em declarar as tarefas tal qual foram apresentadas na solução problema da seção anterior. Também utilizaremos a função **rand** da biblioteca C para gerar valores aleatórios para variáveis que irão simular as entradas do sistema. Declararemos os semáforos e filas que serão utilizados para o sincronismo e comunicação entre as tarefas. A Figura 4.6 apresenta o código-fonte para esta primeira etapa.

Figura 4.6 | Criação das tarefas, semáforos e filas

```
#include <stdio.h>
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#define botaoVirtual    (rand() % 4)
#define presencaVirtual (rand() % 2)
#define ldrVirtual      (rand() % 100)
#define tempVirtual     (rand() % 100)
static void TarefaBotaoGira (void *pvParameters);
static void TarefaSensorPresenca (void *pvParameters);
static void TarefaAcionaSaidas (void *pvParameters);
static void TarefaLDR (void *pvParameters);
static void TarefaTemperatura (void *pvParameters);

QueueHandle_t xFilaBotao, xFilaLDR, xFilaTemp;
SemaphoreHandle_t xSemaforoPresenca;
void main_blinky( void ){
    xFilaBotao = xQueueCreate( 5, sizeof( int ) );
    xFilaLDR = xQueueCreate( 5, sizeof( int ) );
    xFilaTemp = xQueueCreate( 5, sizeof( int ) );
    xSemaforoPresenca = xSemaphoreCreateBinary();
    xTaskCreate(TarefaBotaoGira,"TarefaBotaoGira",configMINIMAL_STACK_SIZE,NULL,1,NULL);
    xTaskCreate(TarefaSensorPresenca,"TarefaSensorPresenca",configMINIMAL_STACK_SIZE,NULL,5,NULL);
    xTaskCreate(TarefaAcionaSaidas,"TarefaAcionaSaidas",configMINIMAL_STACK_SIZE,NULL,4,NULL);
    xTaskCreate(TarefaLDR,"TarefaLDR",256,NULL,3,NULL);
    xTaskCreate(TarefaTemperatura,"TarefaTemperatura",  256,NULL,3,NULL);

    vTaskStartScheduler();
    for( ;; );
}
```

Fonte: captura de tela do *software* Notepad++, feita pelo autor.

Agora podemos implementar as tarefas responsáveis pela leitura de cada variável do sistema. Como o acionamento das saídas dependem do estado do sensor de presença, para essa atividade, utilizaremos um semáforo binário que indicará a tarefa de saídas que as mesmas podem ser atualizadas quando um novo sinal do sensor for detectado. Para as demais tarefas, usaremos filas que irão repassar para a tarefa de saídas, qual o valor atual de cada sensor. A implementação desta próxima etapa pode ser vista na Figura 4.7.

Figura 4.7 | Implementação das tarefas de entrada

```
static void TarefaBotaoGira (void *pvParameters){
    int ValorBotao = 0;
    for( ;; ){
        ValorBotao = botaoVirtual;
        xQueueSend( xFilaBotao, &ValorBotao, (TickType_t) 0 );
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
static void TarefaSensorPresenca (void *pvParameters){
    for( ;; ){
        if(presencaVirtual)
            xSemaphoreGive(xSemaforoPresenca);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
static void TarefaLDR (void *pvParameters){
    int ValorLDR = 0;
    for( ;; ){
        ValorLDR = ldrVirtual;
        xQueueSend( xFilaLDR, &ValorLDR, (TickType_t) 0 );
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
static void TarefaTemperatura (void *pvParameters){
    int ValorTemp = 0;
    for( ;; ){
        ValorTemp = tempVirtual;
        xQueueSend( xFilaTemp, &ValorTemp, (TickType_t) 0 );
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

Fonte: captura de tela do *software* Notepad++, feita pelo autor.

Por fim, nesta última etapa do nosso código, implementaremos a tarefa de saídas, que irá aguardar bloqueada pela chegada do sinal do semáforo binário cuja origem é o sensor de presença. Assim que o sinal for recebido, a tarefa fará a leitura de todas as variáveis através de suas respectivas filas, e atualizará o valor das saídas com base na seleção do botão de estações. A Figura 4.8 apresenta a implementação desta tarefa.

Figura 4.8 | Implementação das tarefas de saída

```
static void TarefaAcionaSaidas (void *pvParameters){
    int botao, ldr, temp, velVentilador, dimmerLed, persiana;
    for( ;; ){
        xSemaphoreTake( xSemaforoPresenca, portMAX_DELAY );
        printf( "Semáforo recebido\r\n");
        fflush( stdout );
        persiana = 1;
        xQueueReceive( xFilaBotao, &botao, portMAX_DELAY );
        xQueueReceive( xFilaLDR, &ldr, portMAX_DELAY );
        xQueueReceive( xFilaTemp, &temp, portMAX_DELAY );
        dimmerLed = ldr / 2;
        switch(botao){
        case 0:
            velVentilador = temp / 3;
            vTaskDelay(5000 / portTICK_PERIOD_MS);
            break;
        case 1:
            velVentilador = 0;
            vTaskDelay(1000 / portTICK_PERIOD_MS);
            break;
        case 2:
            velVentilador = temp / 3;
            vTaskDelay(3000 / portTICK_PERIOD_MS);
            break;
        case 3:
            velVentilador = 0;
            vTaskDelay(2000 / portTICK_PERIOD_MS);
            break;
        }
        printf( "Botao: %d\t LDR: %d\t Temp: %d\t Ventilador: %d\t Led: %d\t Persiana: %d\r\n",
            botao, ldr, temp, velVentilador, dimmerLed, persiana);
        persiana = 0;
    }
}
```

Fonte: captura de tela do *software* Notepad++, feita pelo autor

Avançando na prática

Monitoramento do sistema

Descrição da situação-problema

O *software* desenvolvido pelo seu time teve muita aceitação dentre os demais times de desenvolvimento da empresa, porém outros desenvolvedores estão tendo dificuldade em acessar os dados gerados pelos sensores para que suas tarefas possam ser executadas com maior precisão. Dessa forma, foi solicitado que o software atual foi melhorado por meio da implementação de uma nova fila que comunicação a cada execução da tarefa de saída todos os dados dos sensores em um único pacote. Faça este incremento na solução atual, e também crie uma tarefa de testes que receberá os dados da fila e os apresentará através de um comando **printf**.

Resolução da situação-problema

Inicialmente devemos declarar uma nova tarefa de testes, realizar sua alocação no escalonador, definindo um baixo valor de prioridade e ocupação de memória, e criaremos também uma nova fila responsável pelo envio das informações consolidadas das entradas. Para consolidar todas as informações das entradas em um único elemento para ser enviado, podemos utilizar uma estrutura (do inglês **sctruct**) e utiliza-la como parâmetro de criação da fila. A implementação desta primeira etapa pode ser vista na Figura 4.9.

Figura 4.9 | Implementação da tarefa teste, estrutura de entradas e fila

```
static void TarefaTeste (void *pvParameters);

QueueHandle_t xFilaBotao, xFilaLDR, xFilaTemp, xDadosEntrada;
SemaphoreHandle_t xSemaforoPresenca;

typedef struct s_DadosEntrada
{
    int botao;
    int presenca;
    int ldr;
    int temp;
}t_DadosEntrada;

void main( void ){
    xFilaBotao = xQueueCreate( 5, sizeof( int ) );
    xFilaLDR = xQueueCreate( 5, sizeof( int ) );
    xFilaTemp = xQueueCreate( 5, sizeof( int ) );
    xDadosEntrada = xQueueCreate( 10, sizeof( t_DadosEntrada ) );
    xSemaforoPresenca = xSemaphoreCreateBinary();

    xTaskCreate( TarefaTeste,"TarefaTeste",configMINIMAL_STACK_SIZE,NULL,5,NULL );
```

Fonte: captura de tela do *software* Notepad++, feita pelo autor.

Agora na tarefa de saída, declaramos uma nova variável do tipo **estrutura DadosEntrada** e repassamos a informação dos sensores para esta nova estrutura, e em seguida enviamos esta estrutura preenchida para a Tarefa Teste, que, por sua vez, recebe esta informação via fila e utiliza a função **printf** para exibir seu conteúdo. A Figura 4.10 demonstra a modificação para atender a proposta.

Figura 4.10 | Envio dos dados para Tarefa Teste

```
        dadosEntrada.botao = botao;
        dadosEntrada.ldr = ldr;
        dadosEntrada.temp = temp;
        dadosEntrada.presenca = 1;
        xQueueSend( xDadosEntrada, &dadosEntrada, (TickType_t) 0 );

        persiana = 0;
    }
}

static void TarefaTeste (void *pvParameters){
    t_DadosEntrada dadosEntrada;
    for( ;; ){
        xQueueReceive( xDadosEntrada, &dadosEntrada, portMAX_DELAY );
        printf( "Botao: %d\t LDR: %d\t Temp: %d\t Presenca: %d\r\n",
                dadosEntrada.botao, dadosEntrada.ldr, dadosEntrada.temp, dadosEntrada.presenca);
        fflush( stdout );
    }
}
```

Fonte: captura de tela do *software* Notepad++, feita pelo autor.

Faça valer a pena

1. Um *software* que utiliza um sistema operacional de tempo real terá sua implementação baseada na criação de tarefas, que representarão as funções distintas que este mesmo *software* vai desempenhar. Cada uma delas deverá possuir uma prioridade pré-definida, que irá guiar o *kernel* do sistema operacional a escalonar as tarefas e reservar tempo de processamento de acordo com esta propriedade.

As tarefas em um sistema operacional de tempo real devem possuir mecanismos para sincronismo e troca de informação. Quanto ao sistema FreeRTOS, os principais mecanismos para este fim são:

a) Mutex, Variáveis Globais e Semáforos.
b) Filas, Funções e Variáveis globais.
c) Semáforos, Variáveis Globais e Funções.
d) Semáforos, Filas e Mutex.
e) Funções, Semáforos e Filas.

2. Um semáforo é um mecanismo de sincronização disponibilizado pelo FreeRTOS e pode ser utilizado para desbloquear uma tarefa quando um determinado evento ocorrer, por exemplo, uma interrupção, sincronizando este evento a tarefa (BARRY, 2016).

A respeito dos semáforos binários implementados pelo sistema FreeRTOS, avalie as afirmações a seguir:

I. A função **xSemaphoreGive** é utilizada para enviar um sinal de semáforo da tarefa de origem para outra tarefa.
II. Sobre a função **xSemaphoreTake**, seu segundo parâmetro é utilizado por quanto tempo a tarefa ficará bloqueada após receber o sinal do semáforo.
III. Deve-se incluir ao código fonte o arquivo **task.h** para se utilizar o mecanismo de semáforo.
IV. O semáforo é um mecanismo do *kernel*, logo não pertence a nenhuma tarefa em específico.

É correto o que se afirmar apenas em:

a) I e IV.
b) II e IV.
c) II e III.
d) I, III e IV.
e) Apenas em I.

3. Uma fila não pertence a nenhuma tarefa em específico, sendo que várias tarefas podem compartilhar a mesma fila, tanto para enviar, quanto para receber dados. Cada fila armazena um conjunto finito de itens (*queue lenght* ou comprimento da fila), que pode ter um tamanho fixo de bytes (*item size* ou tamanho do item), e ambos os parâmetros devem ser definidos no momento de criação da fila (BARRY, 2016).

Considere a seguinte instrução utilizando uma Fila do sistema FreeRTOS:

```
xQueue = xQueueCreate( 10, sizeof( uint32_t ) );
```

É possível afirmar que:

a) A fila armazenará o valor inicial igual a 10.
b) A fila será criada para armazenar até 10 itens com tamanho igual a 4 bytes.
c) A fila será criada para armazenar itens com até 10 bytes de tamanho.
d) A fila será criada para armazenar até 32 itens com tamanho igual a 10 bytes.
e) Serão criadas 10 filas que armazenarão itens com tamanho de 4 bytes.

Seção 4.2

Desenvolvimento de projetos embarcados

Diálogo aberto

Caro aluno, nesta seção, prosseguiremos com nossos estudos sobre sistemas embarcados, desta vez retomando nossa atenção aos dispositivos que utilizam o sistema operacional Linux.

Existem diversas vantagens em utilizar o mesmo Linux que usamos para computadores pessoais em dispositivos de menor capacidade computacional, uma vez que praticamente todas as ferramentas e funções presentes no kernel são compatíveis. Veja, por exemplo, o caso das interfaces multimídia presentes em alguns automóveis, que possuem navegadores, comandos por voz, e conexão com a internet. Todas essas funcionalidades só são possíveis pois, por trás deste dispositivo, existe uma distribuição Linux sendo executada, eventualmente com o mesmo kernel utilizado até mesmo por grandes servidores.

Nesta seção, implementaremos aplicações que exemplificam algumas das principais funcionalidades do Linux em projetos de sistemas embarcados. Inicialmente, aprenderemos a utilizar a linguagem C++ e algumas de suas bibliotecas para realizar a manipulação de arquivos dentro do Linux. Logo após, utilizaremos os mesmos conceitos de acesso ao sistema de arquivo para realizar o acesso aos pinos digitais de uma plataforma de *hardware* embarcado, de forma a realizar operações de acionamento dos pinos e também sua leitura. Por fim, será introduzida uma das principais plataformas para o desenvolvimento em sistemas embarcados Linux, o QT, que, juntamente a sua poderosa linguagem de marcação QML, permite a criação de aplicações de interface gráficas ricas que proporcionam ótimas experiências de interação com o usuário.

Lembre-se de que nesta unidade você é o líder de um importante grupo de profissionais que buscam dominar a tecnologia de sistemas embarcados. Após dominar todos os tópicos relevantes para a construção da interface de usuário baseada em Linux, você decidiu que uma ferramenta adequada para a construção da mesma

seria o QT e sua linguagem nativa QML. Dessa forma, foi solicitado que você realizasse o projeto de uma nova interface, na forma de código-fonte, com as seguintes características:

- Campo de Configuração: o usuário poderá configurar o sistema para uma das quatro opções de estação do ano.

- Campo de Engenharia: o usuário poderá acionar cada uma das saídas previstas no projeto individualmente (Persiana 1, Persiana 2, Ventilador). Utilize a manipulação de IO's para realizar essa tarefa.

- Salvar parâmetros: o usuário poderá salvar em um arquivo texto (.txt ou .csv) as configurações que estão presente na tela, de forma a poder restaurá-las posteriormente.

Como realizar a interação entre o *script* QML e o código em C++? Qual melhor forma de arranjar a hierarquia entre os diversos controles em QML? Cada exemplo que passaremos nesta seção trará a você um componente a mais para a resolução desse importante desafio, e no final você se sentirá preparado para desenvolver suas próprias aplicações. Então bons estudos e faça o seu melhor!

Não pode faltar

Introdução

Como vimos, o Linux se tornou um dos principais sistemas operacionais para o uso em diversas plataformas, desde computadores pessoais e servidores até a sistemas com muito pouco recurso computacional, como o caso de pequenos relógios e celulares. Mas, considerando sua arquitetura como fora estudado na Seção 3 da Unidade 3 deste livro, como podemos desenvolver de fato aplicações direcionadas para este importante sistema operacional?

O sistema Linux, desenvolvido inicialmente em Linguagem C, dispõe de bibliotecas para aplicações escritas em C ou C++ para acesso as funções básicas de seu Kernel. que possui todos os códigos de suporte necessários para sua execução e para as aplicações. O núcleo realiza a abstração do *hardware* dispondo acessos por meio de chamadas do sistema (FERREIRA, 2008). Nesta seção, aprenderemos como interagir com o sistema Linux com exemplos práticos, que vão desde acessos a arquivos para leitura

e escrita, passando por controle do *hardware*, e finalmente criando aplicações completas, com o uso de interfaces gráficas amigáveis ao usuário, sem perder o foco da natureza dos sistemas embarcados.

Manipulação de Arquivos e acesso a IO's

O Linux (herdando propriedades conceituais do UNIX) utiliza um sistema de arquivos que representa a estrutura pela qual todas as informações do sistema são armazenadas, desde arquivos comuns de texto, até acessos a aplicações de baixo nível de interface ao *hardware* (NEGUS, 2015). No primeiro exemplo, compreenderemos como uma aplicação, desenvolvida em linguagem C++, consegue manipular um simples arquivo de texto, possibilitando sua leitura, bem como sua escrita.

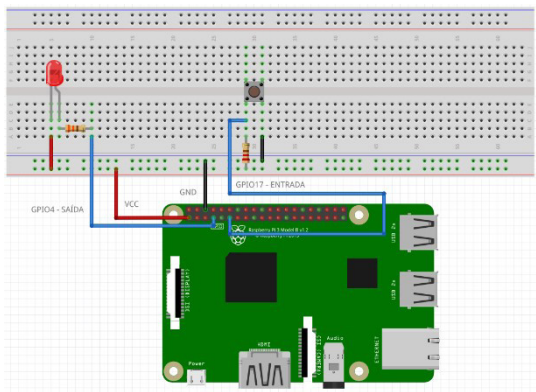
O código-fonte anexo demostra esse exemplo.

	Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/ciencia-da-computacao-II/u4/s2/sistema-de-arquivos.pdf> ou QR Code.

Para este exemplo, foi criada a classe **Armazenamento**, dedicada aos processos de escrita e leitura do arquivo passado como argumento para a instanciação de um novo objeto da classe, a partir dos métodos **GravaDados** e **ApresentaDados**, respectivamente. Os processos de manipulação dos arquivos são realizados utilizando funções implementadas pela biblioteca fstream, que declara classes e outros tipos para realizar operações de entrada e saída com arquivos externos (LISCHNER, 2003). O método ApresentaDados instancia um novo objeto do tipo **ifstream** chamado **arquivo** que abrirá o arquivo cujo endereço fora passado como parâmetro, e após, passará por todas suas linhas e apresentar seu conteúdo. Já o método GravaDados instanciará um objeto do tipo **osftream** que criará um novo arquivo (caso não exista), e gravará a *string* (cadeia de caracteres) de entrada do método neste arquivo. A função principal instanciará um novo objeto da classe Armazenamento, depois criará um arquivo dentro da pasta **Documents** do Linux, gravando três *strings* no arquivo para finalmente apresentar os dados gravados.

No exemplo seguinte, utilizaremos o mesmo conceito do exemplo anterior para manipulação de arquivos, só que dessa vez controlaremos entradas e saídas digitais através do Linux. Como vimos, o sistema de arquivos do Linux é responsável não somente por armazenar e gerenciar os arquivos virtuais do sistema operacional, mas também utilizar arquivos especiais para realizar a interface com *drivers* e periféricos de *hardware*. Como exemplo, adotaremos o caso da plataforma *Raspberry Pi 3*, cujos pinos de GPIO (*General Purpose Input/Output* ou Entradas e Saídas de Propósito Geral) podem ser manipulados a partir dos arquivos presentes na pasta **gpio**, que é encontrada no caminho: /sys/class/gpio. A Figura 4.11 apresenta uma proposta de protótipo que interliga o pino GPIO4 da Raspberry a um led (saída) e o pino GPIO17 a um botão (entrada).

Figura 4.11 | Protótipo para manipulação de IOs via Linux



Fonte: captura de tela do *software* Fritzing, feita pelo autor.

A pasta gpio apresenta um arquivo especial denominado ***export*** (exportar) que, ao gravar o número de um gpio dentro do mesmo, dá acesso aos arquivos destinados à manipulação dos pinos, sendo eles: arquivo ***direction*** (direção), utilizado para definir a direção do pino (entrada ou saída), e arquivo ***value*** (valor), utilizado para acionar um pino (saída) ou realizar a leitura do mesmo (entrada).

Exemplo de código-fonte em C++ para manipulação de IO's no Linux:

Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/ciencia-da-computacao-II/u4/s2/GPIO.pdf> ou QR Code.

Neste exemplo, é demonstrada criação da classe **GPIOClass** que implementa os seguintes métodos:

- **export_gpio:** exporta o arquivo de gpio para conceder acesso aos arquivos de manipulação.
- **setdir_gpio:** configura a direção do pino de gpio, ou seja, entrada ou saída.
- **setval_gpio:** utilizado para controlar um pino configurado como saída.
- **getval_gpio:** utilizado para realizar a leitura de um pino configurado como entrada.

Na função principal são criados dois objetos instanciados da classe GPIOClass: gpio4 e gpio17, que serão utilizados para manipular os pinos 4 e 17 da placa Raspberry, respectivamente. Inicialmente, é realizado o processo de exportação dos pinos, para posteriormente configurá-los como saída e entrada, nessa ordem. Dentro de um a laço infinito, o gpio17 (conectado ao botão) é lido em intervalos de 500ms e, caso o botão seja pressionado, o gpio4 é ligado, ativando dessa forma o led a ele conectado.

Assimile

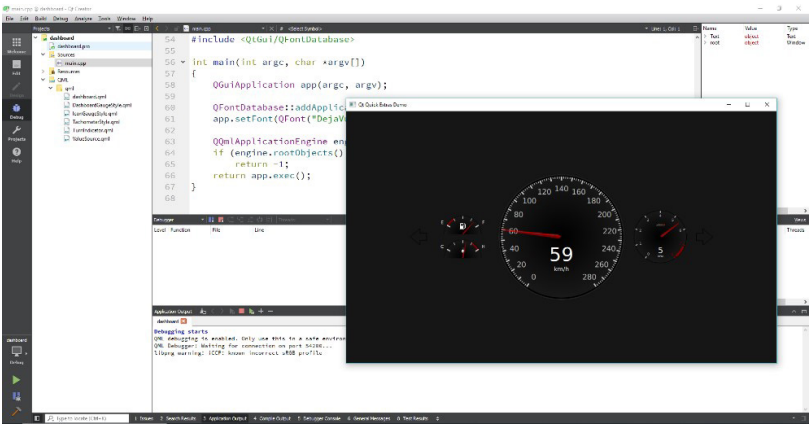
Lembre-se de que assim como já foi estudado nas seções sobre entradas e saídas digitais para microcontroladores, é necessário, em um primeiro momento, definir qual a direção do pino de IO, para posteriormente utilizar suas funções de escrita ou leitura.

Plataforma QT em sistemas embarcados

QT é uma ferramenta gráfica multiplataforma, que permite criar, compilar e executar aplicações em todos os principais sistemas operacionais, como o Windows, o Linux, e o Mac OS X. Grande parte do QT é dedicada a promover uma interface que seja independente

de plataforma para realizar tarefas desde a manipulação de bytes de memória, até a criação de aplicações gráficas multitarefas (TRELIN, 2007). Inicialmente, a plataforma QT foi criada para auxiliar desenvolvedores que utilizavam a linguagem C++ para a criação de projetos gráficos, porém em pouco tempo, a empresa *Trolltech*, sua criadora, começou a disponibilizar ferramentas para o desenvolvimento em outras linguagens, como o Java, JavaScript, Python, .NET e etc. Na Figura 4.12, é apresentada a IDE QT Creator, dedicada ao desenvolvimento de aplicações C++.

Figura 4.12 | IDE QT Creator



Fonte: captura de tela do *software* QT Creator feita pelo autor.

Devido à sua forte característica de ser uma ferramenta multiplataforma, o QT tem se tornado uma das principais plataformas para o desenvolvimento de aplicações Linux para sistemas embarcados, especialmente em situações em que se torna necessário a utilização de uma interface gráfica amigável ao usuário.

Pesquise mais

O que acha de conhecer mais sobre a poderosa plataforma QT através do exemplo prático de um projeto simples e educativo? Então acesse o *Projeto Urna Eletrônica Educativa com Raspberry Pi e Qt5*. Disponível em: <https://www.embarcados.com.br/urna-eletronica-raspberry-pi-e-qt5/>. Acesso: 11 jul. 2018.

Exemplo de software em QT

No próximo exemplo, será demonstrado uma aplicação simples utilizando a plataforma QT que cria uma janela (do inglês *window*) contendo quatro componentes básicos: uma área de texto, dois botões, e uma área de texto não editável, também chamada de *label* (etiqueta). O usuário digita um texto qualquer no campo de texto editável, da parte superior da janela e, ao clicar no botão **Transferir**, o texto digitado será transferido para a *label*. Se o usuário clicar no botão **Apagar**, o texto contido no controle *label* será apagado.

Exemplo de aplicação Window em QT:

Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/ciencia-da-computacao-II/u4/s2/window.pdf> ou QR Code.

A função principal cria um objeto do tipo **MainWindow**, que representa a janela da aplicação, e que será utilizado como base para todos os demais objetos e controles a serem adicionados. Quando o botão Transferir é pressionado, o método **on_pushButton_clicked** é evocado pela aplicação, que transfere o valor da propriedade de texto (*text*) do controle de texto para a *label*. Da mesma forma, ao se clicar no botão Apagar, o método *clear* (limpar) é executado para limpar os caracteres do controle *label*.

Interface gráfica e QML

Para sistemas operacionais que possuem um servidor gráfico, a opção do desenvolvimento de aplicações gráfica utilizando QT e suas classes de *Window* são muito adequadas, porém, para sistemas embarcados, é muito comum encontrar distribuições Linux que operem com recursos enxutos do sistema operacional, não contando, em muitos casos, com um gerenciador de janelas. Para esse cenário, o QT apresenta a solução **QML** (*QT Modeling Language* ou Linguagem de Modelamento do QT), uma linguagem de marcação inspirada no padrão JavaScript, utilizada para a criação de interfaces gráficas interativas com o usuário. Dentre seus principais componentes, encontra-se a biblioteca **QT Quick**, que dispõe de controles aprimorados de interface, focados especialmente a aplicações de telas sensíveis ao toque.

Exemplificando

Podemos criar uma tela bem simples para interação com o usuário, utilizando poucos comandos em QML. A Figura 4.13 apresenta o código-fonte para esta aplicação.

Figura 4.13 | Simples aplicação de uma tela em QML

```
import QtQuick 2.11
import QtQuick.Window 2.11

Window{
    visible: true
    Rectangle    {
        MouseArea.onClicked: {
            Qt.quit();
        }
    }
}
```

Fonte: captura de tela do *software* QT Creator, feita pelo autor.

Inicialmente, são declaradas as bibliotecas necessárias para realizar a referências dos objetos, logo após, é utilizado o objeto principal *Window*, e dentro dele um retângulo com uma área de mouse, que ao clique do usuário, comanda o fechamento da janela.

No último exemplo desta seção, demonstraremos a utilização da linguagem QML em conjunto com a linguagem C++ em uma aplicação real para sistemas embarcados. A aplicação consiste de uma interface simples construída em QML contendo quatro botões de seleção no qual o usuário poderá escolher qual a cor utilizar para o plano de fundo da aplicação. Assim que o usuário realiza a seleção, automaticamente a cor de fundo é alterada, e o usuário pode salvar sua preferência através de um botão denominado **SALVAR**, que utiliza a classe **Armazenamento** demonstrada nos exemplos anteriores para guardar um parâmetro em um arquivo de texto dentro de uma pasta do sistema operacional. Um outro botão denominado **RESTAURAR** também é implementado, e utiliza da mesma classe citada para ler o arquivo de texto, abstrair o parâmetro de cor salvo previamente, e assim restabelecer a cor de fundo selecionada pelo usuário. A Figura 4.14 apresenta a tela de execução desta aplicação.

Figura 4.14 | Exemplo de aplicação QML

Fonte: captura de tela do *software* QT Creator, feita pelo autor.

Reflita

É possível se utilizar uma linguagem de marcação como o QML para a criação de interfaces gráficas em sistemas operacionais que também possuam gerenciadores de janela? Existe alguma vantagem para essa aplicação? Qual outra linguagem de marcação você conhece?

No exemplo abaixo, apresenta o código em C++ criado para a aplicação, e que realiza a integração com a linguagem QML. É criada uma classe denominada ***Task*** (Tarefa), e a partir dela é instanciado um objeto que será passado para a aplicação como função de plano de fundo (do inglês *back-end*), ou seja, este objeto será responsável por tratar os eventos que exijam interface entre a parte gráfica da aplicação e a parte lógica. A tarefa *task* implementa os métodos **salvarConfig** e **resgatarConfig** que, através da já estudada classe Armazenamento, realizam a manipulação de um arquivo de texto para salvar e recuperar o parâmetro salvo pelo usuário.

Integração da Linguagem C++ com QML

Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/ciencia-da-computacao-II/u4/s2/C%2B%2B_e_QML.pdf> ou QR Code.

Para que a linguagem QML tenha acesso a estas funções, é preciso que em suas declarações, seja utilizada a diretiva **Q_INVOKABLE**, que permite que os métodos sejam visíveis a interface gráfica.

No exemplo abaixo, é apresentado o código em QML utilizado. Perceba que os controles são dispostos de maneira hierárquica por meio de chaves, de forma a indicar qual a precedência e correlação entre os mesmos. Por exemplo, os controles de seleção do tipo ***RadioButton*** estão todos em um mesmo nível, porém todos eles estão dentro do controle ***GroupBox***, que, por sua vez, encontra-se dentro do controle principal ***Window***. Para acessar as funções descritas em plano de fundo, basta aplicar o nome do objeto ***Task*** utilizado na função principal em C++ em conjunto com o nome do método que se deseja executar, e, dessa forma, cria-se uma interligação entre a parte gráfica e parte lógica da aplicação.

Código em QML para a aplicação integrada

Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/ciencia-da-computacao-II/u4/s2/QML_Integrado.pdf> ou QR Code.

Nesta seção, tivemos a oportunidade de conhecer mais sobre o desenvolvimento de aplicações utilizando o sistema operacional Linux para projetos embarcados. Utilizando os conceitos teóricos sobre sistemas de arquivo, podemos controlar os periféricos de I/O de nossa plataforma de *hardware*, bem como criar e modificar arquivos de texto. Posteriormente, conhecemos o poderoso ambiente de desenvolvimento QT, e sua linguagem QML, que nos propiciou criar interfaces gráficas modernas e que favorecem a interação com o usuário. Por fim, criamos uma aplicação reunindo todas as técnicas aprendidas na seção, em um projeto completo de interface homem-máquina.

Sem medo de errar

Lembre-se de que nesta unidade você é o líder de um importante grupo de profissionais que buscam dominar a tecnologia de sistemas embarcados. Neste desafio, você deve utilizar a plataforma QT (Linguagem C++) e a linguagem QML para desenvolver o projeto de uma nova interface com as seguintes características: a) **Campo de Configuração**, para configurar o sistema para uma das quatro opções de estação do ano; b) **Campo de Engenharia**, para que o usuário possa acionar cada uma das saídas previstas no projeto individualmente e c) **Salvar parâmetros**, para que o usuário tenha o opção de salvar em um arquivo texto (.txt ou .csv) as configurações que estão presente na tela, de forma a poder restaurá-las posteriormente.

Vamos iniciar a resolução desse problema a partir da elaboração do arquivo QML que representará a interface com o usuário. No exemplo abaixo, é apresentada um possível arranjo para a interface gráfica, na qual será criada uma janela principal contendo dois controles **GroupBox**, sendo que no primeiro são alocados os botões para a seleção da estação do ano, e no segundo são alocados os marcadores para o acionamento individual de cada saída física do sistema. Código em QML para a situação-problema:

Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/ciencia-da-computacao-II/u4/s2/QML_SP.pdf> ou QR Code.

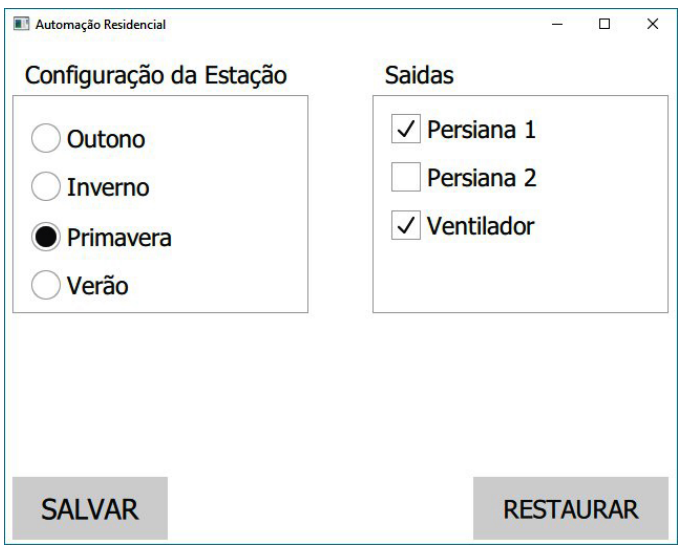
Ao clicar no botão **SALVAR**, o método **salvarConfig** é acionado pelo código QML, que gravará em um arquivo de texto a opção de estação do ano desejada pelo usuário. Já o botão **RESTAURAR** evocará o método **resgatarConfig**, que vai acessar o mesmo arquivo que fora utilizado anteriormente para salvar a configuração da estação e retornar qual valor foi salvo no mesmo. No exemplo, é apresentado o código de plano de fundo, escrito em C++, que implementará as funções do software. Código em C++ para a situação-problema:

Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/ciencia-da-computacao-II/u4/s2/C%2B%2B_SP.pdf> ou QR Code.

Perceba que o código apresentado está utilizando as mesmas classes que foram criadas nos exemplos descritos na seção, tanto para o armazenamento e manipulação de arquivos, quanto para a configuração e acesso aos GPIOs. Inicialmente, são configuras os três primeiros pinos da placa de desenvolvimento como saídas, e também é criado um arquivo chamado **arquivo.txt** que armazenará a configuração da estação do ano.

Ao clicar em qualquer opção do grupo de **Saídas**, o código QML evoca o método de acionamento de saídas **setGPIO**, que ligará ou desligará a saída a depender do estado da seleção do controle. Por fim, na Figura 4.15, é apresentada a tela da aplicação, após compilação e execução.

Figura 4.15 | Código em QML para a aplicação integrada



Fonte: captura de tela do *software* QT Creator, feita pelo autor.

Avançando na prática

Mudança de escopo: armazenando o estado das saídas

Descrição da situação-problema

A solução do seu time para a interface gráfica foi muito bem aceita e elogiada pelos clientes da empresa. Um novo cliente, interessado em continuar com a boa parceria com sua empresa, solicitou uma alteração para o escopo do projeto: deseja que os valores das saídas das persianas sejam armazenados no arquivo texto e posteriormente recuperados, e não mais a estação do ano. Revise o código em QML e em C++ e apresente uma proposta para atender essa mudança de escopo.

Resolução da situação-problema

Para atender o novo escopo, teremos que alterar a lógica utilizada para salvar no arquivo de texto o nome da estação do ano que foi selecionada pelo usuário. Para isso, vamos inicialmente mudar os comandos do evento **onClicked** do botão SALVAR, para que então salvemos um número que representa a combinação das saídas das persianas, sendo 3 para todas as persianas acionadas, 2 para somente a primeira, 1 para somente a segunda, e 0 para nenhuma persiana acionada. Também devemos alterar o método **resgatarConfig**, para que o mesmo leia o arquivo e retorne um número inteiro referente ao valor salvo anteriormente. E, por fim, no evento onClicked do botão RESTAURAR, fazemos o inverso do que fora feito para o botão SALVAR, ou seja, acionamos os controles das persianas de acordo com o valor retornado do arquivo. Alterações no programa para atender novo escopo.

Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link <https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/ciencia-da-computacao-II/u4/s2/codigo_AP.pdf> ou QR Code.

Faça valer a pena

1. O sistema Linux, desenvolvido inicialmente em Linguagem C, dispõe de bibliotecas para aplicações escritas em C ou C++ para acesso as funções básicas de seu kernel, que possui todos os códigos de suporte necessários para sua execução e para as aplicações. O núcleo realiza a abstração do *hardware* dispondo acessos por meio de chamadas do sistema (FERREIRA, 2008).
Utilizando-se do sistema de arquivo do Linux é possível realizar diversas operações, tais como:

a) Alterar prioridade de tarefas, manipular arquivos de texto, e configurar a interface de rede.
b) Fazer login com prioridade de administrador, manipular arquivos, e criar diretórios.
c) Suspender o funcionamento do kernel, alterar prioridade de tarefas e configurar impressoras.
d) Manipular arquivos, criar novos diretórios, acessar interfaces com o *hardware*.
e) Suspender o funcionamento o kernel, configurar a interface de rede e alterar prioridade de tarefas.

2. O sistema de arquivos do Linux é responsável não somente por armazenar e gerenciar os arquivos virtuais do sistema operacional, mas também utilizar arquivos especiais para realizar a interface com *drivers* e periféricos de *hardware*.

A biblioteca _____ pode ser utilizada para realizar operações de _____ e _____ de arquivos externos, por exemplo, para a manipulação de GPIOs. Na plataforma Raspberry Pi, é possível utilizar esta biblioteca para configurar a direção dos pinos através do arquivo _____, e posteriormente ler ou gravar o arquivo _____ para leitura ou manipulação do gpio.

As palavras que completam, corretamente, essas lacunas são:

a) fstream, criação, exclusão, direction, export.
b) ofstream, leitura, escrita, export, value.
c) fstream, entrada, saída, direction, value.
d) ifstream, leitura, exclusão, export, direction.
e) fstream, criação, exclusão, export, direction.

3. Para sistemas operacionais que possuam um servidor gráfico, a opção do desenvolvimento de aplicações gráfica utilizando QT e suas classes de Window são muito adequadas, porém, para sistemas embarcados, é muito comum encontrar distribuições Linux que operem com recursos enxutos do sistema operacional, não contando em muitos casos com um gerenciador de janelas. Para este cenário, o QT apresenta a solução **QML** (*QT Modeling Language* ou Linguagem de Modelamento do QT), que é uma linguagem de marcação inspirada no padrão JavaScript, utilizada para a criação de interfaces gráficas interativas com o usuário.
Avalie as seguintes asserções e a relação proposta entre elas.

I. A linguagem em QML é voltada para o desenvolvimento de interfaces gráficas que possuem total dependência ao código de plano de fundo, escrito em C++.

PORQUE

II. Uma das maneiras de realizar a chamada de um método de plano de fundo através do QML é com a declaração do mesmo método utilizando a diretiva Q_INVOKABLE.

A respeito dessas asserções, assinale a opção correta.

a) As asserções I e II são proposições verdadeiras, e a II é uma justificativa da I.
b) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa da I.
c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
e) As asserções I e II são proposições falsas.

Seção 4.3

Conceitos avançados de sistemas embarcados

Diálogo aberto

Caro aluno, ao decorrer de todas as seções deste livro didático, tivemos a oportunidade de conhecer os principais conceitos dos sistemas embarcados, desde aqueles controlados por microcontroladores e seus periféricos, até os sistemas microprocessados, dedicados a aplicações que requerem maior capacidade computacional. Nesta última seção, abordaremos alguns dos tópicos mais avançados em sistemas embarcados, bem como introduzir algumas das principais tendências para esta área do conhecimento. Tudo o que aprendemos até então nos capacita a desenvolver sistemas de baixa e média complexidade, como aqueles encontrados em alguns produtos do nosso cotidiano, e outros em aplicações de controle de baixa complexidade. Porém, o que devemos dominar a fim de desenvolver produtos de alta complexidade, como aqueles voltados para a área militar (uma aeronave, ou um sistema de telemetria) ou para a área médica (um equipamento de diagnóstico por imagem ou um eletrocardiograma)?

Nesta seção, conheceremos, inicialmente, algumas técnicas que expandem o conhecimento básico que tivemos até então sobre sistemas embarcados, voltados a projetos que exigem maior complexidade e capacidade de processamento, como sistemas multinúcleo e processadores dedicados de sinais. Logo após, conheceremos algumas das principais tendências para a área de sistemas embarcados, iniciando pelo tópico de conectividade focando na temática de IoT (Internet das Coisas) e comunicação entre dispositivos autonomamente. Veremos também alguns conceitos importantes voltados a sistemas críticos que requerem abordagens especiais quanto ao seu desenvolvimento e principalmente à compreensão dos impactos de suas falhas. Por fim, daremos um passo a mais quanto o tema de desenvolvimento embarcado Linux, compreendendo o desafio de embarcar seu Kernel em plataformas heterogêneas e quais ferramentas utilizar para criar distribuições customizadas.

Lembre-se de que, nesta unidade, você assume o papel de líder do desenvolvimento de novas tecnologias em sistemas embarcados de uma importante empresa do setor de automação residencial. Na demanda final, você deve apresentar tendências tecnológicas para as novas gerações de produtos, bem como indicar quais os possíveis caminhos para dominar tais conceitos. Após a conclusão bem-sucedida do novo produto, você deve redigir um relatório de entrega de projeto, no qual você expõe essas novas oportunidades. No relatório, você deve apresentar técnicas e soluções de conectividade para que o produto possa interagir com outros meios e equipamentos, como smartphones, webservers etc. Faça também uma reflexão sobre como melhorar o desempenho do produto corrente e demais produtos utilizando conhecimentos mais refinados na área de sistemas embarcados, como análise de sistemas críticos e criação de distribuições customizadas do Linux. A cada tópico que veremos nesta seção, você encontrará as respostas que precisa para concluir este último desafio. Como os projetos podem se beneficiar das novas vertentes? Quais os impactos positivos e negativos dessas novas tecnologias? Vamos fechar essa jornada com uma chave de ouro? Bons estudos e vamos lá!

Não pode faltar

Introdução e conceitos de sistemas embarcados avançados

Como podemos ver no decorrer de todas as seções deste livro, o universo dos sistemas embarcados é muito vasto e diversificado, exigindo do pesquisar ou do profissional da área contínuo aprimoramento dos seus conhecimentos e reciclagem de conceitos fundamentais. Apesar da aparente complexidade de todo o conteúdo que vimos até então, grande parte dos temas que foram abordados representam simplesmente a parte introdutória sobre esta disciplina, sendo que, para de fato desenvolver projetos de vanguarda na área é necessário o domínio de tópicos mais específicos, de acordo com os requisitos de cada produto (CONRAD, 2014).

Devido ao perfil dinâmico e abrangente, corriqueiramente as empresas concentram seus profissionais de sistemas embarcados em equipes de **pesquisa e desenvolvimento** (do inglês *Research and Development* ou simplesmente R&D), por constatarem que boa parte do tempo dedicado à criação de novos produtos passa pela

aplicação de métodos de pesquisa e de experimentação, antes de iniciar as devidas fases de construção de *hardware* e *software*. Apesar de serem numerosos os conceitos mais avançados para sistemas embarcados, nesta seção, veremos exemplos dos mesmos, de forma a lhe incentivar a continuar nessa jornada do conhecimento.

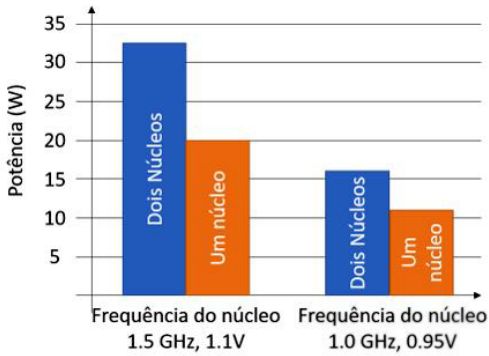
Sistemas embarcados multinúcleo

A indústria de computadores é impulsionada pela busca pelo desempenho de processamento cada vez superior. Desde sistema de propósito geral altamente customizáveis em redes de computadores, telecomunicações e aviônica até computação embarcada de baixa potência em módulos automotivos, sistemas de comunicação e pequenos dispositivos, os clientes esperam produtos cada vez mais rápidos, mais eficientes e mais poderosos.

No entanto, os produtos com núcleos únicos (do inglês *single core*) têm mostrado uma capacidade decrescente de aumentar o seu desempenho frente ao ritmo exigido pelo consumidor. O processamento multinúcleo (do inglês *multicore*) é reconhecidamente um componente-chave para melhorias contínuas de desempenho. A indústria trilha um caminho claro para a utilização de cada vez mais núcleos de processamento em seus sistemas, e dispositivos com dois e quatro núcleos já estabelecidos há vários anos. No entanto, esse tipo de expansão cria um desafio, não apenas para a indústria de semicondutores, mas também para os desenvolvedores de sistemas e software que operam com essa tecnologia (SVENNEBRING, 2009).

Antes do advento dos sistemas multinúcleo, boa parte dos esforços para melhorar o aumento de desempenho possuía um caminho direto: aumentar a frequência do processador! Mas tornou-se bastante evidente que pressionar muito a frequência possui um viés negativo, pois penaliza o consumo de energia, que por sua vez gera mais calor, o que requer um sistema de resfriamento mais avançado, diminuindo a confiabilidade e encurtando a longevidade do dispositivo. Desta forma, resolver os problemas adicionais que surgem com o aumento da frequência acaba se tornando mais oneroso para o custo do projeto. A Figura 4.24 toma como exemplo a família de processadores MPC8641 da empresa *Freescale*, em que são comparadas implementações deste *chip* em configurações que utilizam somente um núcleo e aquelas que utilizam dois núcleos.

Figura 4.24 | Comparação do chip MPC8641 em versões de um e dois núcleos



Fonte: elaborada pelo autor.

Perceba que, na configuração de núcleo simples, ao aumentar a frequência em 50%, há um aumento de aproximadamente duas vezes o consumo de potência, contudo, se utilizamos dois núcleos com frequência de 1 GHz para atingir a mesma velocidade, há um acréscimo de somente 30% no consumo de potência.

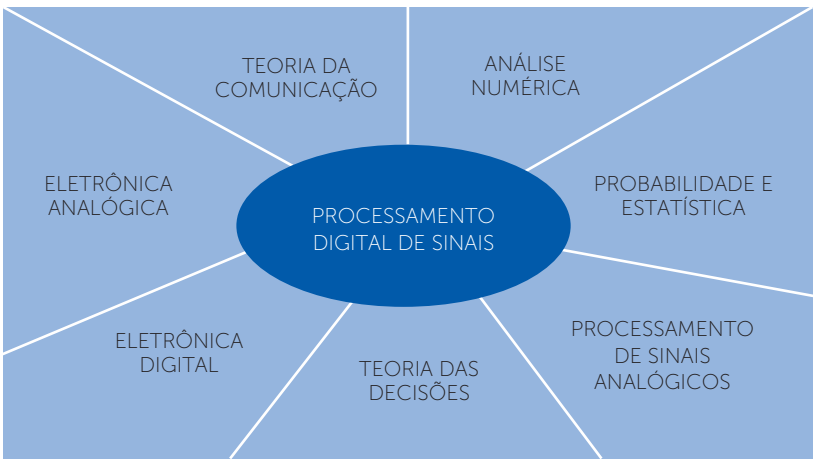
Processamento Digital de Sinal (DSP)

O Processamento Digital de Sinal (do inglês *Digital Signal Processing* ou *DSP*) se distingue de outras áreas da ciência da computação pelo tipo principal de dados que este sistema se utiliza: **sinais**. Na maioria dos casos, os sinais são originados de dados de sensores do mundo real: vibrações sísmicas, imagens, ondas sonoras, etc. O **DSP** representa não somente o dispositivo que processará esses sinais, mas também a matemática, os algoritmos e as técnicas utilizadas para manipulá-los depois dos mesmos já terem sido convertidos para uma forma digital. Isso inclui uma grande variedade de objetivos, como o melhoramento na qualidade de imagens, reconhecimento e geração de voz, compressão de dados para armazenamento e transmissão, dentre outros (SMITH, 1999).

O processamento digital de sinal remonta a década de 60 e 70, quando os computadores digitais se tornaram disponíveis ao público científico. Os computadores eram muito caros naquela época, e os DSPs eram limitados a poucas aplicações mais críticas, e quatro eram as áreas ditas como pioneiras para o avanço dessa

tecnologia: **sonares e radares**, nos quais governos investiam em tecnologia de forma a garantir segurança e soberania; **exploração de óleo e gás**, sendo uma das áreas mais rentáveis a época; **exploração espacial**, na qual toda a informação era tida como insubstituível; e **análise por imagem na área médica**, na qual inúmeras vidas podiam ser salvas. Atualmente, as técnicas e componentes dedicados ao processamento digital de sinais se expandiu a diversas outras áreas, abarcando em sim um perfil multidisciplinar, como pode ser visto na Figura 4.16 algumas das principais áreas de estudo em torno desta temática.

Figura 4.16 | Principais áreas de estudo da disciplina de processamento digital de sinais



Fonte: elaborada pelo autor.

No campo dos sistemas embarcados, os dispositivos dedicados a essa finalidade também são chamados de DSPs (Processadores Digitais de Sinal, do inglês *Digital Signal Processors*) e são microprocessadores com características próprias que podem ser programados e operam em tempo real, com velocidades muito superiores aos microprocessadores para aplicações genéricas. A capacidade de processar grandes quantidades de números em pouco tempo é um dos principais benefícios que os Processadores Digitais de Sinais oferecem ao mundo da eletrônica. Os DSPs foram projetados levando em consideração as operações mais habituais em um processamento digital, como

as de adição, multiplicação e transferência de memória. Para tal existem instruções de repetição que precedem tais operações, tornando possível a execução das mesmas muitas vezes em apenas um ciclo de memória (SMITH, 1999).

Assimile

Os processadores DPS são muito parecidos com os microcontroladores, podendo embarcar os mesmos periféricos que já estudamos anteriormente, somente acrescentando outros especializados ao processamento de sinais.

Conectividade e Comunicação embarcada

Atualmente, com um mundo cada vez mais conectado e com a emersão dos dispositivos inteligências, a tecnologia de comunicação tem passado por constantes evoluções de forma a questionarmos sobre suas novas fronteiras e desafios. A humanidade sempre sentiu a necessidade de se comunicar e de se manter ativo o princípio de comunidade e sociedade. Com o passar do tempo, a velocidade da informação e o volume de dados cresceu em ritmo acelerado, desde os primeiros mensageiros, passando pelas comunicações físicas, e atingindo seu ápice com o advento da internet (ABREU, 2014). Atualmente se busca um novo patamar, em que não somente propicie uma comunicação rápida e eficaz entre nós, seres humanos, e nossas máquinas, mas também um novo modelo que permita que os próprios dispositivos, dotados de algum nível de inteligência e autonomia, possam se comunicar e tomar decisões sem a intervenção direta de seus operadores, sendo um dos grandes motores dessa revolução a chamada **Internet das Coisas**.

A Internet das Coisas (do inglês IoT – *Internet of Things*) é definida como uma infraestrutura de rede global e dinâmica, com capacidade de autoconfiguração baseada em padrões e protocolos de comunicação interoperáveis nos quais "**coisas**" físicas e virtuais possuem identidade, atributos e personalidades virtuais, utilizando interfaces inteligentes e que sejam perfeitamente integráveis dentro da rede de informação (FRIESS,

2014). De forma a tornar esse conceito em uma nova realidade, uma serie de objetivos devem ser explorados, como:

- **Solucionar barreiras tecnológicas**, sempre com foco na segurança da informação e no controle de um número expressivo de dispositivos que estejam conectados em uma mesma rede.
- **Explorar o potencial de integração** das arquiteturas de IoT e seus agentes junto a sistemas já consolidados de computação em nuvem e abordagens de *Big Data* (Volume muito alto de dados).
- **Validar a aceitabilidade do usuário** focando na aplicação e em marcos regulatórios. Um exemplo seria a comunicação entre dois carros, e o impactos na legislação atual.

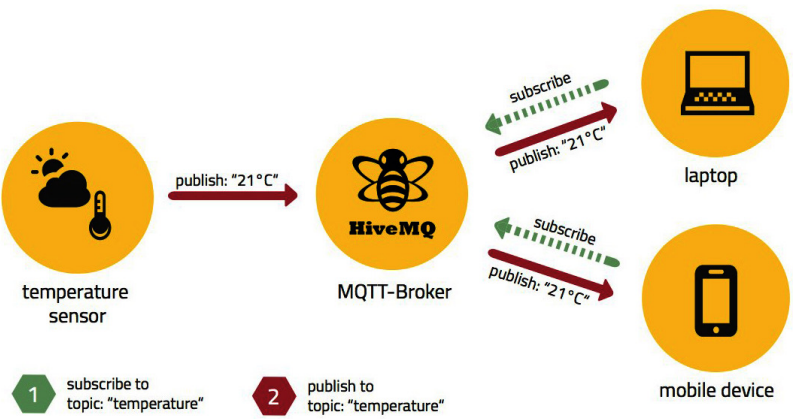
Um dispositivo que atue como um agente IoT pode ser considerado um sistema embarcado que possui integrado um microprocessador ou microcontrolador, com periféricos que possibilitem a implementação de protocolos de comunicação, como *bluetooth*, WiFi, RFID, dentre outros. Uma das premissas para que um dispositivo IoT possa ser conectado à rede e realizar a troca de informação com outro dispositivo é a sua alta resiliência a condições adversas do ambiente, como ruídos, interrupção de comunicação e baixo nível de sinal. De forma a mitigar esses problemas, alguns protocolos de comunicação específicos vêm sendo adotados e aprimorados, como exemplo o **MQTT** (*Message Queuing Telemetry Transport* ou Transporte Telemétrico de Mensagens Enfileiradas).

O MQTT foi desenvolvido em 1999, mas com o crescimento exponencial da **Internet das Coisas** e a necessidade de conexão e comunicação entre dispositivos inteligentes de baixa potência, o MQTT encontrou recentemente um grande mercado. O protocolo usa uma arquitetura baseada em **publicação** e **assinatura** (do inglês *publish* e *subscribe*), em contraste com o protocolo HTTP e seu paradigma de solicitação e resposta (do inglês *request* e *response*). O sistema de publicação e assinatura é totalmente orientado a eventos e permite que as mensagens sejam enviadas aos clientes da rede assim que os eventos são disparados. O ponto central de comunicação do MQTT é o ***Broker*** (Intermediário), encarregado por despachar todas as mensagens entre os remetentes e seus receptores, sendo que cada cliente que publica uma mensagem para o *Broker*, inclui um **tópico**

em sua mensagem que contém as informações de roteamento que serão utilizadas pelo *Broker* para distribuição das mesmas. Cada cliente que quer receber mensagens de um determinado tópico deve assinar (*susbscribe*) ao tópico correspondente, e o *Broker* entregará todas as mensagens assim que as mesmas forem enviadas (FRIESS, 2014).

Dessa forma, os clientes não precisam se conhecer, pois eles só se comunicam a partir dos tópicos. Essa arquitetura permite soluções altamente escaláveis, sem dependências entre quem produzirá os dados, e quem vai consumi-los. Na Figura 4.17, temos um exemplo de aplicação do protocolo MQTT, em que um sensor de temperatura embarcado envia para *Broker* o valor de temperatura atual, por meio do tópico *temperature*. Os demais clientes da rede, como o computador e o celular realizam a assinatura a esse tópico e, assim que uma nova mensagem é enviada a *Broker*, ela é imediatamente transmitida a eles.

Figura 4.17 | Principais áreas de estudo da disciplina de processamento digital de sinais



Fonte: <https://www.hivemq.com/blog/how-to-get-started-with-mqtt>. Acesso: 17 jul. 2018.

Pesquise mais

O protocolo MQTT vem ganhando muito espaço no desenvolvimento de aplicações em IoT, em especial àquelas voltadas a automação comercial e residencial. O que acha de conhecer um pouco mais sobre este protocolo através de uma aplicação prática nesta área? Acesse: *Use o MQTT DASH para controlar uma lâmpada remotamente*. Disponível em: <https://www.embarcados.com.br/mqtt-dash/>. Acesso em: 19 jul. 2018.

Sistemas críticos

Existem poucas áreas da vida moderna em que o *software* não é um componente importante (embora muitas vezes invisível). O *software* em nossas vidas é cada vez mais complexo e sua interação com o mundo real significa que seus requisitos estão em um estado de mudança constante. Muitos sistemas complexos, desde sistemas médicos até sistemas de transporte e infraestruturas de telecomunicações, dependem de um *software* confiável e de alta qualidade.

O *software* em sistemas embarcados frequentemente precisa ser modificado em resposta a mudanças nos requisitos do sistema e mesmo ao ambiente em que opera. Tais modificações podem envolver a adição de novas funcionalidades, o ajuste de funções existentes, ou a substituição de subsistemas inteiros (HINCHEY, 2010). Dessa forma, devido ao grau de confiabilidade que um sistema atinge, de forma a garantir sua integridade, bem com a integridade de seus agentes, surge o conceito de sistemas críticos em que as falhas podem resultar em perdas econômicas significativas ou ameaças à vida humana, e podem ser subdivididos de acordo com suas implicações de falha (SOMMERVILLE, 2011):

- **Sistemas críticos de proteção:** pode levar a perda de vida humana, prejuízos pessoais sérios ou impacto de ordem ambiental.
- **Sistemas de missão crítica:** pode levar a uma incapacidade de completar os objetivos gerais do projeto ou do próprio sistema, por exemplo, a perda de infraestrutura crítica ou perda de dados.
- **Sistemas críticos de negócio:** pode levar a custos econômicos significativos, sejam tangíveis ou intangíveis, como por exemplo, a perda de negócios ou danos à reputação.
- **Sistemas críticos de segurança:** pode levar à perda de dados confidenciais através de roubo ou perda acidental.

Para o desenvolvimento de *software* embarcado voltado para esta categoria de sistemas, a metodologia tradicional de desenvolvimento baseada em requisitos funcionais de *software* deve ser revista, de forma a agregar componentes adicionais de verificação que submetam o processo de desenvolvimento a contínuos questionamentos em face as possíveis condições de

falha do sistema. Alguns métodos foram desenvolvidos para auxiliar esse processo de análise crítica, entre eles:

- FTA - ***Fault Tree Analysis*** (Árvore de Análise de Erros): trata-se de uma representação gráfica de relações lógicas entre eventos de falhas (uma árvore de falha), em que o evento topo é ramificado em eventos contribuintes por meio da análise de causa-efeito.
- FMEA - ***Failure Modes and Effects Analysis*** (Análise de Modos de Falhas e seus Efeitos): tem como objetivo identificar potenciais modos de falha de um produto ou processo de forma a avaliar o risco associado a esses modos de falhas, para que sejam classificados em termos de importância e então receber ações corretivas com o intuito de diminuir a incidência de falhas.

Reflita

Quais propriedades e características um desenvolvedor deve se ater de forma a categorizar um sistema como sendo crítico ou não, a partir do seu escopo? As mesmas técnicas e métodos utilizados para o desenvolvimento de sistemas convencionais podem ser empregadas neste caso?

Distribuições Linux em sistemas embarcados

Trazer projetos de sistemas embarcados a vida utilizando dispositivos complexos de silício, como sistemas-em-um-chip (SoC), operando com aplicações em nível de *software* e de *firmware* é um grande desafio para os fabricantes de equipamentos, que em muitas ocasiões, estão mais confortáveis com o desenvolvimento do *hardware* do que o *software* em si. De muitas maneiras, o movimento de código aberto melhorou a tarefa de desenvolvimento de *software*, e também trouxe seu próprio conjunto de desafios. Felizmente, o desenvolvimento aberto de *software* embarcado continuou a evoluir, simplificando os processos de desenvolvimento, aumentando a eficiência e permitindo aos fabricantes de equipamentos a diferenciar de forma mais eficaz a capacidades do seu *hardware* (STREIF, 2016).

Antes que qualquer aplicação possa ser executada em uma nova plataforma de *hardware*, um kernel e um sistema de arquivos eram necessários, bem como um compilador, um *bootloader* e outros componentes. Só após isso, a aplicação voltada para a arquitetura de *hardware* poderia ser embarcada, sendo modificadas se necessário com ferramentas de código aberto e integrada em um *software* Linux que fosse coerente com a especificação da distribuição escolhida. Dessa forma, surge a necessidade da criação de uma ferramenta de código aberto capaz de auxiliar o desenvolvedor na criação de distribuições Linux específicas para a sua plataforma de *hardware*, e que seja concomitantemente, capaz de favorecer o processo de customização incremental, em que novas funcionalidades possam ser adicionadas, e outras deprecadas, possam ser removidas.

O Projeto **Yocto** surge como uma alternativa a este novo cenário, sendo um projeto colaborativo de código aberto da **Fundação Linux** cujo objetivo é produzir ferramentas e processos que permitam a criação de distribuições Linux para *software* embarcado que sejam independentes da arquitetura atrelada ao *hardware*. O seu foco é melhorar o processo de desenvolvimento de *software* para distribuições Linux embarcadas fornecendo ferramentas, metadados e processos interoperáveis que permitem o desenvolvimento rápido e repetitivo de sistemas embarcados baseados em Linux, nos quais todos os aspectos do processo de desenvolvimento podem ser personalizados (STREIF, 2016).

Exemplificando

Podemos considerar o exemplo de um relógio inteligente que embarcará o Kernel Linux, como o apresentado na Figura 4.18. Esse dispositivo utilizará o mesmo kernel presente em computadores pessoais e servidores, porém, algumas aplicações tipicamente nativas de distribuições para computadores, como navegadores, sistemas de edição de arquivos, e diversos *drivers* não precisam ser adicionados a esta distribuição customizada. Através de ferramentas como o **Yocto** é possível criar uma distribuição customizada e enxuta do Linux, incluindo somente aqueles pacotes que sejam coerentes com a finalidade deste produto.

Figura 4.18 | Exemplo de relógio inteligente que embarca o Kernel Linux

Fonte: <https://www.linux.com/photos/7-android-and-linux-based-smartwatches-available-now>. Acesso em: 17 jul. 2018.

Nesta seção, vimos algumas das principais tendências e tecnologias que devem ser dominadas por aqueles que almejam se aventurar em projetos mais avançados em sistemas embarcados atualmente. Que esses conhecimentos introdutórios possam lhe motivar a continuar seus estudos por esse fascinante mundo da computação, que exige de seus profissionais aprimoramentos contínuos para acompanhar as constantes evoluções tecnológicas.

Sem medo de errar

Você, agora como líder do desenvolvimento de novas tecnologias na empresa, recebeu uma demanda de grande impacto para o futuro da companhia: definir os caminhos de inovação frente as tendências tecnólogas na área de sistemas embarcados. Para isso, você deve elaborar um relatório em que apresenta técnicas e soluções de conectividade para que o produto atual possa interagir com outros meios e equipamentos. Deve também fazer uma reflexão sobre como melhorar o desempenho do produto corrente e demais produtos utilizando conhecimentos mais refinados na área de sistemas embarcados, como análise de sistemas críticos e criação de distribuições customizadas do Linux.

Para elaborar o relatório, vamos dividir o conteúdo do mesmo em tópicos, tais quais aqueles estudados nesta seção. Em cada um dos mesmos, confrontaremos os conceitos ao produto atual desenvolvido pela empresa.

Sistemas embarcados multinúcleo

Visando aumentar a capacidade de processamento do produto, uma das alternativas seria substituir o processador atual por um com maior frequência de processamento. Porém, como fora visto nesta seção, essa alternativa, apesar de plausível, acarretaria na elevação do consumo de energia do processador e elevação da potência dissipada (maior temperatura), sendo necessário a inclusão de elementos arrefecedores para controle de temperatura, o que deixaria o projeto mais oneroso. Outra alternativa mais adequada seria a inclusão de outro núcleo de processamento e, dado a característica do *software* no qual parte de sua lógica é dedicada ao controle em tempo real das entradas e saídas, e outra parte é dedicada a interface gráfica de interação com o usuário, poderia ser utilizado um núcleo para execução do sistema operacional de tempo real e outro núcleo para execução do sistema operacional Linux, que controlaria a interface gráfica.

Processamento Digital de Sinal (DSP)

Apesar da natureza deste projeto não apresentar características predominantes sobre o processamento de sinais, algumas das técnicas incorporadas ao processamento digital de sinais poderiam ser aplicadas para melhorar a qualidade de aquisição de alguns sinais analógicos tratados pela aplicação. Alguns métodos de **probabilidade e estatística** poderiam ser utilizados, por exemplo, para elaborar histogramas sobre o histórico de captura da temperatura ambiente, bem como apresentar análises de tendências com base nos mesmos dados, sendo que gráficos representativos poderiam ser elaborados pela interface gráfica da solução. Outro exemplo, seria a aplicação da **teoria das decisões** para elaborar alguns esquemas de tomadas de decisão autônomas da solução com base no fluxo de informação tratado pelo produto, como por exemplo, o refinamento do controle PWM do ventilador através dos sinais de temperatura.

Conectividade e Comunicação embarcada

Alguns periféricos de comunicação poderiam ser embarcados na solução atual, como a inclusão de um chip WiFi para que a plataforma possa se conectar com o roteador da residência e assim conseguir conexão à internet, e, assim, comunicar-se com um

provedor que possa fornecer informações sobre a previsão do clima, possibilitando e tomada de decisões coordenadas e inteligentes. Um outro periférico *bluetooth* também poderia ser embarcado, possibilitando a conexão do sistema a um *smartphone* que, a partir de um aplicativo dedicado, poderia monitorar todas as entradas do sistema, e controlar manualmente todos os dispositivos de saída.

Sistemas críticos

Durante o processo de desenvolvimento dos novos produtos, poderiam ser feitas análises de possíveis falhas e suas consequências, utilizando, por exemplo, a ferramenta **FMEA**, cujo o objetivo é identificar potenciais modos de falha, classifica-los e atribuir planos de ação corretiva para cada qual. Dessa forma, a possibilidade de falhas que sejam críticas para os usuários, o ambiente, para o negócio da empresa, ou para o próprio sistema poderão ser mitigadas ou anuladas.

Distribuições Linux em sistemas embarcados

Conhecemos muitas vantagens em se utilizar o Kernel Linux em projetos de sistemas embarcados, especialmente quando queremos criar interfaces gráficas para a interação com o usuário. Porém, grande parte das funcionalidades e ferramentas dispostas em uma distribuição não são utilizadas pelas aplicações embarcadas. Dessa forma, poderíamos utilizar ferramentas como o **Yocto** para customizar nossa distribuição, focando em ferramentas para o suporte gráfico multimídia e gerenciamento de redes, e remover outras que não agregaria valor a aplicação.

Avançando na prática

Demonstração de conectividade e distribuição Linux

Descrição da situação-problema

Para a apresentar de maneira concreta o potencial de aplicação dos tópicos estudados nesta seção, descreva como o protocolo **MQTT** poderia ser utilizado para enviar para um servidor informações contendo o estado das entradas e saídas, bem como dos sensores analógicos, e apresente alguns tópicos MQTT que poderiam

ser criados para esta finalidade. Sabendo que a aplicação deverá trabalhar em rede, armazenar dados, e suportar aplicações gráficas, quais funcionalidade do sistema Linux você recomendaria para a criação de uma distribuição customizada.

Resolução da situação-problema

Iniciando pelo protocolo MQTT, o mesmo poderia ser implementado considerando cada produto instalado em uma residência como um **cliente** da aplicação, e seria instalado em um servidor um serviço de **Broker**, no qual todos os clientes apontariam a fim de realizar a troca de informação. Considerando a aplicação atual, os seguintes tópicos poderiam ser implementados:

- **Cliente_x / Sala / Temperatura:** tópico para monitorar a temperatura da sala, onde x representa a identificação do cliente.
- **Cliente_x / Sala_de_Estar / Persiana:** monitora a proporção de abertura da persiana, sendo 0 para persiana totalmente fechada, e 100 para persiana totalmente aberta.
- **Cliente_x / Quarto_1 / Lâmpada:** monitora o estado da lâmpada do quarto, sendo 0 para lâmpada apagada e 1 para lâmpada acesa.

Dadas as especificações para o sistema Linux, os seguintes módulos seriam de interesse para a criação de uma distribuição customizada:

- **Sistema de arquivos:** para armazenar dados e ter acesso ao hardware da plataforma.
- **Gerenciamento de rede:** para configurar os periféricos de comunicação, como WiFi e Ethernet.
- **Bibliotecas gráficas:** como a plataforma QT para suporta a criação e execução de aplicações gráficas.

Faça valer a pena

1. A indústria de computadores é impulsionada pela busca pelo desempenho de processamento cada vez superior. Desde sistema de propósito geral altamente customizáveis em redes de computadores,

telecomunicações e aviônica até computação embarcada de baixa potência em módulos automotivos, sistemas de comunicação e pequenos dispositivos, os clientes esperam produtos cada vez mais rápidos, mais eficientes e mais poderosos.

Avalie as seguintes asserções e a relação proposta entre elas.

I. Uma das principais alternativas para a escalada de poder de processamento em sistemas embarcados é adoção de processadores com mais de um núcleo,

PORQUE

II. Aumentar a quantidade de núcleos possui vantagens em relação a solução de simplesmente aumentar a frequência do processador, pois é uma solução mais econômica energeticamente.

A respeito dessas asserções, assinale a opção correta.

a) As asserções I e II são proposições verdadeiras, e a II é uma justificativa da I.
b) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa da I.
c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
e) As asserções I e II são proposições falsas.

2. O DSP representa não somente o dispositivo que processará estes sinais, mas também a matemática, os algoritmos e as técnicas utilizadas para manipulá-los depois dos mesmos já terem sido convertidos para uma forma digital. Isto inclui uma grande variedade de objetivos, como o melhoramento na qualidade de imagens, reconhecimento e geração de voz, compressão de dados para armazenamento e transmissão, dentre outros (SMITH, 1999).

Quais das opções abaixo lista algumas das áreas de estudo em torno do tema de processamento digital de sinais?

a) Teoria da comunicação, Teoria das Decisões, Computação Gráfica.
b) Eletrônica Analógica, Física e Análise Numérica.
c) Eletrônica Digital, Eletrônica Analógica e Teoria das Decisões.
d) Teoria da comunicação, Eletrônica Analógica e Física.
e) Computação Gráfica, Probabilidade e Estatística e Eletrônica Analógica.

3. Trazer projetos de sistemas embarcados a vida utilizando dispositivos complexos de silício, como sistemas-em-um-chip (SoC), operando com aplicações em nível de *software* e de *firmware* é um grande desafio para os fabricantes de equipamentos, que em muitas ocasiões estão mais confortáveis com o desenvolvimento do *hardware* do que o *software* em si.

A criação de uma _____ customizada Linux permite ao desenvolvedor adequar o _____ para a sua aplicação, além de promover flexibilidade para _____ e _____ de funcionalidades. Uma das principais ferramentas para esta finalidade é o Projeto _____.

As palavras que completam, corretamente, essas lacunas são:

a) aplicação, processador, adição, remoção, Linux.
b) aplicação, sistema, aprimoramento, execução, Yocto.
c) customização, sistema operacional, aprimoramento, remoção, Linux.
d) ferramenta, processador, adição, execução, Linux.
e) distribuição, sistema operacional, adição, remoção, Yocto.

Referências

ABREU, A. **História da Comunicação.** Experiências e Perspectivas. Rio de Janeiro: Editora Mauad, 2014.

BARRY, Richard. **Mastering the FreeRTOS Real Time Kernel.** Real Time Engineers Ltd. Bristol. 2016.

LINUX. **Android and Linux-Based Smartwatches Available Now.** Disponível em: <https://www.linux.com/photos/7-android-and-linux-based-smartwatches-available-now>. Acesso em: 17 jul. 2018.

CONRAD, J. **Advanced Embedded Systems Concepts using the Renesas RX63N Microcontroller.** Weston, EUA: Micrium Press, 2014.

DO BIT AO BYTE. **Como utilizar mutex no freeRTOS do ESP32.** Disponível em: <https://www.dobitaobyte.com.br/como-utilizar-mutex-no-freertos-do-esp32/>. Acesso em: 14 set. 2018.

EMBARCADOS. **Use o MQTT DASH para controlar uma lâmpada remotamente.** Disponível em: <https://www.embarcados.com.br/mqtt-dash/>. Acesso em: 19 jul. 2018.

FERREIRA, R. **Linux**: guia do administrador do sistema. São Paulo: Novatec, 2008.

FREERTOS. **History**. [s.d.]. Disponível em: <https://www.freertos.org/RTOS.html>. Acesso em: 10 jun. 2018.

FRIESS, P. **Internet of Things – From Research and Innovation to Market Deployment.** Aalborg, Dinamarca: River Publishers, 2014.

HINCHEY, M. Evolving Critical Systems: a Research Agenda for Computer-Based Systems. **17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems.** 2010.

HIVEMQ. **MQTT 101 – How to Get Started with the lightweight IoT Protocol.** Disponível em: <https://www.hivemq.com/blog/how-to-get-started-with-mqtt>. Acesso: 17 jul. 2018.

KARIM, Y. **Construindo Sistemas Linux Embarcado.** 2. ed. São Paulo: Elsevier, 2009.

LINUX. **Android and Linux-Based Smartwatches Available Now.** Disponível em: <https://www.linux.com/photos/7-android-and-linux-based-smartwatches-available-now>. Acesso em: 17 jul. 2018.

LISCHNER, R. **C++ in a Nutshell.** Sebastopol, EUA: O'Reilly Media, 2003.

NEGUS, C. **Linux Bible.** 9. ed. Indianapolis, EUA: Wiley, 2015.

SHAW, A. **Sistemas e software de tempo real.** São Paulo: Bookman, 2003.

SMITH, S. **The Scientist and Engineer's Guide to Digital Signal Processing.** 2. ed. California, EUA. California Technical Publishing, 1999.

STEVAN, S. **Industria 4.0. Fundamentos, Perspectivas e Aplicações.** São Paulo: Editora Érica. 2018.

SOMMERVILLE, I. **Engenharia de software.** 9. ed. São Paulo: Wesley, 2011.

STREIF, R. **Embedded Linux Systems with the Yocto Project.** Boston, EUA. Prentice Hall, 2016.

SVENNEBRING, J. **Embedded multicore:** an introduction. Tempe, EUA: NXP, 2009.

TRELIN, J. **Foundations of QT Development.** Nova Iorque, EUA: Apress, 2007.

ISBN 978-85-522-1092-4