Pontifícia Universidade Católica do Rio de Janeiro

Projeto Final de Graduação de Engenharia da Computação

Departamento de Informática - DI
Centro Técnico Científico - CTC
Curso de Engenharia da Computação

Desenvolvimento de drivers assíncronos para a plataforma Céu-Arduino

---

Aluno:
Guilherme SIMAS

Orientador:
Ana LUCIA DE MOURA
Noemi RODRIGUEZ

Rio de Janeiro, 28 de fevereiro de 2018

Guilherme Simas

Desenvolvimento de drivers assíncronos para a plataforma Céu-Arduino

Relatório de Projeto Final, apresentado ao programa Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Ana Lúcia de Moura

Rio de Janeiro

28 de fevereiro de 2018

Resumo

Comparamos a linguagem de programação Céu em termos de eficiência e modelagem frente a modelos de programação procedural para sistemas embarcados. Desenvolvemos drivers não bloqueantes na plataforma Céu-Arduino utilizando interrupções a partir de sua contraparte bloqueante em C já existente. Finalmente, descrevemos em Céu uma aplicação distribuída de iluminação inteligente, inpirada em Internet das Coisas, para exemplificar o uso dos drivers que desenvolvemos.
Palavras-chave: céu; sistemas distribuídos; eficiência; sistemas embarcados

Abstract

We compare the Céu programming language in terms of eficiency and modelling over widespread procedural programming models. We develop non blocking drivers in the Céu-Arduino platform using interruptions based on their currently developed blocking counterpart in C. Finally, we implement in Céu a distributed smart lighting application inpired in the Internet of Things to exemplify the use of the drivers we developed.
Keywords: céu; distributed systems; eficiency; embedded systems

Agradecimentos aos meus professores, por todos ensinamentos e as ferramentas me entregues. À minha família, pelo apoio incondicional. A todos meus amigos e companheiros que fazem cada dia valer a pena.

Sumário

1 Introdução

Existem várias definições para o termo “Internet das Coisas”, porém a grande maioria delas compartilha a ideia de que a conexão e troca de dados entre elementos é parte vital do conceito. Por esse motivo, a área de Sistemas Distribuídos possui um papel importante nesse desenvolvimento, já que toda aplicação deve ser capaz de trocar mensagens segura e corretamente de forma a se sincronizar, e de forma escalável.[1][2][3] Outro ponto sobre aplicações em Internet das Coisas é a necessidade de unidades computacionais de custo baixo e consumo eficiente de energia. Por esse motivo microcontroladores são outra parte vital do desenvolvimento de soluções, sendo capazes de processamento de dados por possuírem processadores e de serem facilmente integrados com sensores e atuadores, possuindo hardware especializado para interfacear com esses componentes.[4]

Muitas ferramentas de desenvolvimento fornecidos para microcontroladores incluem rotinas que causam um bloqueio na aplicação, ou seja, enquanto está realizando a chamada correspondente àquela funcionalidade, o software entra em um estado onde realiza tarefas inúteis até que o hardware conclua sua parte. Esse comportamento é indesejável, visto que a aplicação desperdiça tempo aguardando o hardware enquanto poderia estar realizando outras tarefas como, por exemplo, um processamento de dados recebidos por uma mensagem, ou a troca de mensagens em si. Essa ineficiência marca um desperdício de tempo e consumo de energia. O motivo pelo qual tal comportamento bloqueante é comumente utilizado é pela simplicidade decorrente da programação sequencial.

O bloqueio de aplicações é um desafio enfrentado frequentemente em aplicações que envolvem sistemas nos quais o tempo de processamento ou reação a estímulos externos é pertinente ao funcionamento da aplicação. O paradigma de programação orientada a eventos é utilizado como alternativa nessas situações. Em uma aplicação orientada a eventos, o sistema segue seu fluxo normal até a chegada de um “evento”, como a chegada de uma mensagem, ou a conclusão de um trabalho por parte do hardware. A chegada de tal evento emite uma interrupção no sistema, que irá executar uma rotina de tratamento desse evento, e após terminado, irá retomar seu fluxo normal de execução, a partir de onde estava no momento da interrupção. Esse

paradigma busca evitar que quaisquer estímulos sejam ignorados involuntariamente pela aplicação ou que ciclos computacionais sejam desperdiçados. Essa estruturação introduz uma espécie de paralelismo e imprevisibilidade no fluxo de execução da aplicação, algo que linguagens comumente utilizadas para programação de sistemas embarcados, como C, não fazem um bom papel de representar, por serem historicamente procedurais (espera-se que o fluxo de execução siga naturalmente a ordem de leitura do código, a linha de baixo imediatamente seguindo a execução da linha de cima).

Céu é uma linguagem estruturada síncrona reativa, onde a orientação a eventos é inerente à sua programação, assim como o paralelismo entre seções de código que surgem com esse paradigma. Outro benefício de Céu é o fato de que a ordem de execução de trechos de programa que estão descritos para rodar em paralelo é previsível dado a chegada de um determinado evento.[5] Devido às vantagens dessas características para a programação de componentes de sistemas embarcados, se encontra disponibilizado um kit de desenvolvimento em Céu para a família de microcontroladores Arduino, chamada Céu-Arduino.

Dentre um número enorme de microcontroladores, a plataforma Arduino[6] possui uma comunidade estabelecida de desenvolvedores e é open-source. Contribuições na forma de exemplos de aplicações e desenvolvimento de drivers e bibliotecas são frequentes e são o que tornam a comunidade tão bem-sucedida. Por último, a plataforma é de fácil acesso e muitas vezes utilizada como parâmetro, devido à sua popularidade.

Céu-Arduino permite a programação de microcontroladores Arduino utilizando a linguagem Céu, o que facilita a programação orientada a eventos. Céu-Arduino, porém, não reimplementa os drivers e bibliotecas já desenvolvidos para Arduino, e, portanto, não pode impedir o bloqueio da aplicação que é consequência da chamada de funções bloqueantes pré-desenvolvidas. A re-implementação desses módulos eliminaria mais uma possibilidade de bloqueio de uma aplicação que as utilize.[7]

Esse trabalho propõe reimplementar drivers e bibliotecas de Arduino, que atualmente causam bloqueio da aplicação, de forma a que esse bloqueio não ocorra mais. A abordagem utilizada nesse desenvolvimento foi do uso de interrupções suportadas por hardware especializado com orientação a eventos e, por esse motivo, a linguagem

escolhida para esse desenvolvimento foi Céu-Arduino. Os resultados desse desenvolvimento foram publicados na página open-source de desenvolvimento do kit, para que futuros desenvolvedores possam fazer uso de tais módulos em suas próprias aplicações.

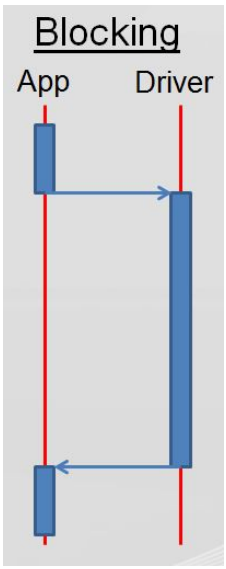
Para demonstrar o uso dos drivers, desenvolvemos uma aplicação em Sistemas Distribuídos para exemplificar a pertinência dos módulos desenvolvidos. A aplicação consiste em uma rede de sensores e atuadores na plataforma Arduino, utilizando troca de mensagens. A aplicação escolhida como estudo de caso foi um sistema de iluminação inteligente. Esse trabalho tem como meta servir como uma contribuição para a comunidade desenvolvedora de sistemas embarcados e de aplicações de IoT, assim como desenvolvedores da plataforma Arduino.

2 Fundamentos

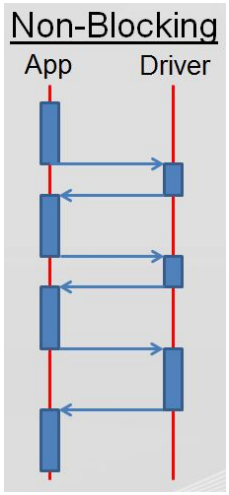
2.1 Bloqueio de Aplicações

Bloqueios em chamadas de drivers de microcontroladores são comumente causados por implementações que usam polling para verificar se a operação foi concluída. Polling é caracterizado quando o software constantemente verifica um estado, aguardando uma mudança, para só então prosseguir. Funcionalidades de hardware especializado em microcontroladores costumam sinalizar sua conclusão mudando o estado de um registrador (uma flag). Implementações não-bloqueantes de chamadas de drivers podem envolver fazer com que a mudança de estado da flag cause uma interrupção, de modo que o software não precisa ficar no estado de polling e possa usar o tempo para realizar outras operações na aplicação, só retornando à chamada do driver quando este concluiu sua tarefa. Caso não haja nenhuma tarefa a ser realizada, o microcontrolador pode entrar em um modo de baixo consumo de energia, portanto sempre há ganhos por utilizar a abordagem não- bloqueante.

Existem microcontroladores, como Arduino, que oferecem suporte a interrupções externas. A implementação desse suporte se dá em um pino do microcontrolador que dispara uma interrupção na presença de uma mudança de estado específica no pino. Para que um driver de um dispositivo forneça suporte a interrupções, portanto, basta que o dispositivo possua uma saída que emita uma mudança de estado na ocorrên-



(a) Comportamento bloqueante



(b) Comportamento não-bloqueante

Figura 1: Comparação entre comportamento bloqueante e não-bloqueante

cia de eventos relevantes. Ao conectar esta saída ao pino de interrupção externa do microcontrolador, é possível mapear um evento do dispositivo a uma interrupção do microcontrolador, permitindo a orientação a evento que torna possível evitar o bloqueio desnecessário da aplicação.

Esse modelo é uma convenção de ambas as partes e se repete nos microcontroladores mais populares do mercado, incluindo Arduino, assim como nos dispositivos externos fabricados para sistemas embarcados. Isso facilita o desenvolvimento de drivers e bibliotecas para ambas as partes.

Kits de desenvolvimento para microcontroladores que ajudam a abstrair a implementação do hardware do programador propõem tornar o processo de desenvolvimento de aplicações para aquele módulo mais simples e acessível. Os drivers e bibliotecas disponibilizados pela própria Arduino apresentam atualmente funções e módulos que causam o bloqueio da aplicação por utilizarem a técnica de polling, embora existam reimplementações por parte da comunidade de alguns desses módulos de forma a eliminar o bloqueio.

2.2 Arduino

Arduino é uma plataforma eletrônica open-source que foca em hardware e software de fácil aprendizado e acesso. Comumente utilizado para prototipação, Arduino conta atualmente com a contribuição de uma vasta comunidade que realiza os mais diversos projetos de sistemas-embarcados, de complexidade variada.[6]

A plataforma consiste em uma placa de prototipação com um microcontrolador embarcado. A placa possui pinos de entrada e saída com grande foco em GPIO (General Purpose Input and Output). A programação se dá via linguagem C adaptada com bilbiotecas e drivers, com a implementação obrigatória de duas funções que definem a aplicação (ver Exemplo 1). O conjunto de ferramentas permite a abstração de elementos específicos à programação de microcontroladores como acessos a registradores, por exemplo, permitindo um desenvolvimento mais alto-nível, acessível e portável. Arduino oferece dezenas de placas diferentes, de especificações variadas de forma a atender diversas demandas. A placa mais popular, porém, é a Arduino Uno, que possuiu embarcado o microcontrolador ATmega328P, da Atmel [8]. Todo o desenvolvimento realizado neste projeto foi focado únicamente na portabilidade para o ambiente deste microcontrolador, embora a adaptação para outros produtos da Arduino seja facilitada pela proposta de portabilidade da empresa.

```
1 void setup(void){
2   // Esse bloco é executado uma vez somente, na inicialização da aplicação
3 }
4 void loop(void){
5   // Este bloco é repetido em loop durante toda o período de execução da
      aplicação
6 }
```

Exemplo 1: Estrutura de uma aplicação Arduino

Como já mencionado, microcontroladores Arduino são programados em um ambiente de linguagem C, uma linguagem procedural. A linguagem segue a lógica de procedência temporal e lógica dos comandos por ordem de leitura, o que significa que espera-se que um comando só seja executado quando o precedente acima dele termina sua execução. Esse método sequencial de programação busca reforçar a proposta de simplicidade da plataforma Arduino.

Como estudo de caso tenhamos uma aplicação cujo objetivo é piscar um LED até o momento que o usuário aperte um botão. O Exemplo 2 apresenta uma implementação simples utilizando as funções disponíveis da plataforma Arduino. O programa executa a rotina de piscar o LED enquanto uma variável de controle indica que o botão nunca foi apertado. Durante a iteração, a aplicação checa o estado do botão e, caso esteja apertado, modifica a variável de controle para que a rotina do LED não seja mais executada.

```
1  int buttonWasPressed = FALSE;
2  void setup() {
3    // initialize digital pin LED_BUILTIN as an output.
4    pinMode(LED_BUILTIN, OUTPUT);
5    pinMode(BUTTON, INPUT);
6  }
7  // the loop function runs over and over again forever
8  void loop() {
9    if(digitalRead(BUTTON_PIN) == HIGH){
10     buttonWasPressed = TRUE;
11   }
12   if(!buttonWasPressed){
13     digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the
       voltage level)
14     delay(1000);                       // wait for a second
15     digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the
       voltage LOW
16     delay(1000);                       // wait for a second
17   }
18 }
```

Exemplo 2: Aplicação bloqueante

Embora a estrutura da aplicação seja simples, seu comportamento não é o esperado. Como a aplicação é sequencial, a verificação do estado do botão só ocorre uma vez que toda a rotina de piscar o LED tenha sido executada. Caso o usuário aperte e solte o botão dentro do intervalo de 2 segundos no qual o LED está piscando, o input será perdido. Inserir checagens no meio da rotina de piscar o LED não resolve o problema, já que o botão ainda pode ser apertado e soltado no segundo de execução tomado pelas chamadas à função bloqueante delay(). Além disso, a aplicação seria

mais eficiente se o microcontrolador entrasse em um estado de baixo consumo de energia durante os momentos em que está aguardando a passagem do tempo. O uso da função delay(), por ser implementada com polling, faz com que o microcontrolador desperdice recursos durante esse tempo de espera.

Para que o comportamento responsivo desejado seja obtido, é necessário não utilizar a função bloqueante para a contagem do tempo. O Exemplo 3 a seguir propõe essa abordagem.

```
1  int ledState = LOW;             // ledState used to set the LED
2  int buttonWasPressed = FALSE;
3
4  unsigned long previousMillis = 0;        // will store last time LED was
       updated
5  const long interval = 1000;           // interval at which to blink (
       milliseconds)
6
7  void setup() {
8    // set the digital pin as output:
9    pinMode(ledPin, OUTPUT);
10   pinMode(BUTTON, INPUT);
11 }
12
13 void loop() {
14   unsigned long currentMillis = millis();
15
16   if(digitalRead(BUTTON_PIN) == HIGH){
17     buttonWasPressed = TRUE;
18   }
19   if (!buttonWasPressed && currentMillis - previousMillis >= interval) {
20     // save the last time you blinked the LED
21     previousMillis = currentMillis;
22
23     // if the LED is off turn it on and vice-versa:
24     if (ledState == LOW) {
25       ledState = HIGH;
26     } else {
27       ledState = LOW;
28     }
29
```

```
30      // set the LED with the ledState of the variable:
31      digitalWrite(LED_BUILTIN, ledState);
32    }
33  }
```

Exemplo 3: Aplicação não bloqueante

Embora o comportamento da aplicação esteja correto, perdemos a simplicidade refletida no Exemplo 2. Mais variáveis de controle foram necessárias no código e a legibilidade é prejudicada. Essa abordagem também não resolve a questão do desperdício de recursos, visto que embora não realizemos o polling da implementação da função delay, estamos constantemente checando o estado do botão e da passagem do tempo, o que também caracteriza polling.

Com o uso de interrupções, podemos nos livrar da ineficiência. No Exemplo 4 uma interrupção se encontra atrelada ao pressionamento do botão. Deste modo a aplicação fica livre da checagem e pode entrar em um modo de baixo consumo com uma chamada a sleep(). Embora o desperdício de ciclos agora seja evitado, a necessidade de variáveis de controle, e a complexidade decorrente, ainda permanece.

```
1  volatile int buttonWasPressed = FALSE;
2  int ledState = LOW;
3  void onButtonPress()
4  {
5    buttonWasPressed = TRUE;
6  }
7  void setup()
8  {
9   pinMode(LED_BUILTIN, OUTPUT);
10  attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), &onButtonPress, RISING);
11 }
12 void loop()
13 {
14   sleep(1000);
15   if (!buttonWasPressed) {
16     if (ledState == LOW) {
17       ledState = HIGH;
18     } else {
19       ledState = LOW;
20     }
```

```
21      digitalWrite(LED_BUILTIN, ledState);
22    }
23  }
```

Exemplo 4: Aplicação utilizando interrupção

Vemos portanto que a utilização de linguagens procedurais, como C, para programação em sistemas embarcados, embora bem difundida, sofre com a dificuldade de se implementar a orientação a eventos de forma concisa e clara. Conforme a complexidade e a necessidade de responsividade a estímulos externos crescem, mais variáveis de controle são necessárias e aplicações simples se convertem em pedaços de código de difícil interpretação. A proposta de simplicidade decorrente da programação sequencial, portanto, se mostra ilusória.

2.3 Céu-Arduino

Céu é uma linguagem síncrona reativa estruturada open-source desenvolvida na PUC-Rio.[5][9] A proposta da linguagem é resolver os desafios descritos acima apresentados pelas linguagens procedurais. A semântica de Céu permite ao usuário escrever uma aplicação reativa orientada a eventos de forma determinística e estruturada, reduzindo o uso de variáveis de controle.

O foco da linguagem na orientação a eventos se dá pelas primitivas de aguardar um evento (”await”) e emitir um evento (”emit”). Eventos em Céu podem ser tanto internos, emitidos pelo programa para coordenação, como externos, como timers e I/O. Céu permite definir uma ISR para as interrupções geradas pelo ambiente e, dentro destas ISRs, a emissão de eventos externos para a aplicação, permitindo a modelagem de interrupções como eventos.

A linguagem expressa a diferenciação lógica entre rotinas que executam sequencialmente e paralelamente. Céu possuiu primitivas para expressar blocos de código que devem rodar em paralelo (par, par/or, par/and). A execução desses blocos, chamados de trilhas, deve incluir ao menos uma espera de um evento, expressado por uma diretiva da linguagem. Quando uma trilha bloqueia à espera de um evento, a outra trilha em paralelo pode executar. Esse modelo permite que os dois blocos de código executem se intercalando porém sem paralelismo real, dado que todo trecho de código delimitado por esperas a eventos (as trilhas) executam sem preempção. A

semântica de Céu assume que o tempo de execução das trilhas é imediato se comparado com a frequência dos eventos. Caso esta diretiva seja respeitada, a aplicação permanece reativa.

O Exemplo 5 a seguir implementa a mesma aplicação dos exemplos anteriores em C.

```
1  par/or do
2      loop do
3          await 1s;
4          emit LED_PIN(on);
5          await 1s;
6          emit LED_PIN(off);
7      end
8  with
9      await BUTTON_PRESS;
10 end
```

Exemplo 5: Aplicação em Céu

O código define dois blocos para executarem em paralelo. O primeiro bloco de código consiste em um loop que pisca o LED, aguardando o evento de passagem de 1 segundo representado pela primitiva await de Céu. O segundo bloco aguarda o evento de pressionamento do botão. O qualificador ”or”utilizado com a primitiva ”par”expressa que, caso um dos blocos em paralelo termine sua execução, todos os outros blocos são abortados e a aplicação segue seu fluxo. Isso significa que caso o botão seja apertado, a rotina rodando em paralelo, neste caso a de piscar o LED, é abortada. Como não há mais comandos a serem executados, a aplicação termina sua execução.

O comportamento reativo de implementação complexa do Exemplo 3 é reproduzido aqui de forma clara e estruturalmente simples, como proposto pelo Exemplo 2.

A compilação de um programa em Céu consiste na tradução de seu código para C para então ser compilado no compilador C de escolha, gerando o executável final. Isso torna possível utilizar o compilador da plataforma Arduino e desenvolver programas em Céu para os microcontroladores Arduino. Basta que uma camada de ambiente seja implementada para que eventos externos possam ser captados, como o pressionamento do botão, e que a emissão de eventos externos possa ser interpretada, como

ligar e desligar o LED. Esta camada de ambiente se encontra desenvolvida e é parte do kit de desenvolvimento Céu-Arduino[7].

A camada de ambiente permite que os eventos de entrada possam ser implementados utilizando interrupções. No caso do Exemplo 5, basta que a interrupção atrelada ao pino dispare o evento BUTTON_PRESS para que a aplicação possua a eficiência proposta pelo Exemplo 4.

Contribuimos para esta camada de ambiente neste projeto com os drivers que desenvolvemos, de módulos da plataforma Arduino, utilizando interrupções, de forma a enriquecer o kit.

3 Drivers

O microcontrolador ATMega328p possuiu, como esperado de um MCU, periféricos programáveis de entrada e saída. A plataforma Arduino disponibiliza drivers em C para esses módulos periféricos, que são implementados utilizando polling apesar do hardware suportar interrupções que poderiam ser utilizadas para evitar este bloqueio. Partimos do estudo de tais drivers em C e do hardware para produzirmos versões em Céu que utilizem interrupção e, assim, evitam o polling, oferecendo as vantagens já descritas para o desenvolvimento de aplicações. As soluções desenvolvidas neste projeto se encontram disponíveis no repositório open-source do kit Céu-Arduino. Para que o driver fosse implementado, foi necessário um estudo não só da plataforma Arduino e do microcontrolador, mas também do ambiente de desenvolvimento disponibilizado pela Arduino e das bibliotecas já implementadas e disponibilizadas em Céu, principalmente as de Céu-Arduino.

Os drivers desenvolvidos atendem somente a plataforma Arduino Uno, por ser a mais utilizada e disponível, e os cenários de uso mais frequentes. O motivo é a busca por uma melhor utilização do tempo do projeto e redução da carga computacional para a execução dos drivers.

O ambiente de desenvolvimento que usa a linguagem C é o mesmo tanto para Arduino quanto para Céu-Arduino. Isto é, ambos os códigos são em alguma etapa compilados utilizando o mesmo compilador. Esse compilador é disponibilizado pela AVR e é uma versão customizada de GCC, chamada de AVR-GCC. Em conjunto com

o compilador, são utilizadas bibliotecas da AVR que tratam da abstração do hardware, como acesso a registradores, para variados chips da Atmel, incluindo o ATMega328p, presente no Arduino Uno, plataforma para qual os drivers deste projeto são destinados.

Outra ferramenta essencial para o andamento do projeto foi a capacidade de se registrar rotinas de serviço a interrupções (Interrupt Service Routines, ISR), que são rotinas de código associadas a interrupções do microcontrolador. Em outras palavras, quando o hardware emite uma interrupção, o software executa a ISR atrelada àquela interrupção. O AVR-GCC disponibiliza uma biblioteca de tratamento de interrupções que torna simples a implementação de tais ISRs.

Atmel disponibiliza um documento detalhado [8] das especificações do microcontrolador presente no Arduino Uno, o ATMega328p. Esse documento, chamado datasheet, possui a definição de todas as funcionalidades e hardwares dedicados presentes, incluindo a definição e descrição dos registradores responsáveis e atrelados a cada um. Esse documento é vital para o entendimento das capacidades e limitações do chip e, principalmente, de como operá-lo.

A datasheet está no centro da primeira etapa do desenvolvimento que realizamos, que é estudar o funcionamento do hardware para que, em conjunto com o estudo do código da implementação atual (se existente), seja possível compreender o comportamento de uma API tradicional. O foco desse passo é criar uma base de conhecimento sobre as informações já disponíveis acerca do módulo, principalmente em relação a interrupções. Esclarecer em quais momentos o hardware opera sem necessidade de interferência do software, indica onde a aplicação poderá entrar em modo de espera, sendo acordado pela interrupção. Esse processo, acompanhado do estudo da implementação já existente dos drivers bloqueantes disponibilizados, permite identificar onde o bloqueio desnecessário do driver, o polling, ocorre, e como ele pode ser evitado.

Com o conhecimento do suporte a interrupção e a identificação do bloqueio, pudemos modificar o driver para que não seja mais bloqueante. A função que realizava a operação por completo se divide em funções que iniciam a operação, funções que consultam o estado da operação e, caso necessário, funções que finalizam a operação. Isso torna possível que a aplicação possa realizar outras operações entre consultas ao estado, o que na chamada bloqueante não era possível. Utilizar interrupções permite

que o hardware entre em um modo de baixo consumo e seja acordado pela interrupção, contribuindo para a eficiência da solução no uso de recursos.

O passo final foi, com o driver não bloqueante em C em mãos, convertê-lo para Céu. As funções são transformadas em eventos de entrada e saída. Expressamos o início de uma operação no hardware por parte da aplicação como a emissão de um evento de saída. Ao finalizar a operação, o driver emite um evento de entrada, o qual a aplicação pode aguardar com await. Deste modo fica explícito no bloco de código que o programa emite uma requisição ao módulo e aguarda sua resposta, podendo realizar tarefas descritas pelas primitas de paralelismo da linguagem no meio tempo. Quando maior complexidade era necessária, recursos de abstração da linguagem foram usadas.

Escolhemos módulos que causam maior impacto devido à frequência de sua utilização em aplicações de sistemas embarcados. As escolhas foram Entrada e Saída Analógica (Analog I/O) e Comunicação SPI (SPI).

3.1 Analog I/O

Foi levantado que as funcionalidades de entrada (leitura de valor analógico) e saída (emissão de valor analógico) são atreladas a hardwares dedicados distintos no microcontrolador.

Os valores de saída analógicos são simulados utilizando ondas PWM (o pino liga e desliga em intervalos regulares, onde o valor analógico seria a razão entre o tempo que permanece ligado e o tempo que permanece desligado). Essa funcionalidade já se encontrava implementada em Céu e, portanto, não foi abordada.

Já os valores de leitura são obtidos por um hardware de Conversão Analógica para Digital (ADC). Esse componente dedicado recebe um valor de voltagem analógico e, após ciclos de processamento, guarda em seus registradores a representação de 10 bits deste valor, quando comparado a uma referência de valor máximo. Em outras palavras, se a voltagem de entrada for o ground, o resultado será “0”. Caso seja igual ou acima da voltagem de referência, será 1023.

Embora exista mais de um pino de leitura analógico no microcontrolador, existe somente um hardware de ADC. Para que possa atender a todos os seis pinos, o hardware conta com um multiplexador que seleciona o pino para qual a leitura deve ser

feita. Não é possível ter mais de um pino tendo seu valor lido pelo hardware ao mesmo tempo.

Para efetuar uma conversão, o multiplexador, cujo valor é controlado por um registrador modificável pelo desenvolvedor, deve receber o valor do pino para qual a leitura será feita. Do mesmo modo, os outros valores de configuração devem ser atribuídos de acordo com a configuração desejada. Para iniciar a conversão, basta que o software escreva o valor lógico 1 no bit do registrador especificado. Esse bit terá seu valor 1 mantido pelo hardware enquanto a conversão está em progresso, e será zerado no momento que a conversão tiver fim e o resultado estiver disponível. Caso o hardware seja configurado para habilitar interrupção, a ISR será chamada no fim da conversão. O software pode então ler o registrador que guarda o valor final da conversão e a operação terá terminado.

A implementação atual de uma função bloqueante de leitura analógica, presente na biblioteca de funções disponibilizada pela Arduino, possui como único argumento o pino para qual a conversão deve ser feita, devolvendo o resultado da mesma. Sua implementação pode ser vista abaixo.

```
1  int analogRead(uint8_t pin) {
2    uint8_t low, high;
3
4    ADMUX = (analog_reference << 6) | (pin & 0x07);
5
6    // start the conversion
7    sbi(ADCSRA, ADSC);
8
9    // ADSC is cleared when the conversion finishes
10   while (bit_is_set(ADCSRA, ADSC));
11
12   low  = ADCL;
13   high = ADCH;
14
15   // combine the two bytes
16   return (high << 8) | low;
17 }
```

Exemplo 6: Leitura analógica bloqueante

```
1  void setup(){
2      pinMode(SENSOR_PIN, INPUT);
3      pinMode(LED_BUILTIN, OUTPUT);
4
5      Serial.begin(9600);
6  }
7  void loop(){
8      int value = analogRead(SENSOR_PIN);
9      Serial.println(value);
10 }
```

Exemplo 7: Aplicação utilizando driver bloqueante

A função configura os registradores do hardware e, em seguida, inicia a conversão. Para aguardar o fim da conversão, o código faz polling no bit que representa uma conversão em andamento. Como já discutido, isso faz a aplicação apresentar um comportamento bloqueante. Após o bit ter sido zerado pelo hardware, a função segue sua execução e retorna o valor do resultado.

A implementação não-bloqueante que desenvolvemos reaproveita a implementação original, modificando a lógica do polling. Enquanto a original possuía apenas uma função que encapsulava todo o processo de leitura, desde a configuração do hardware dedicado até o retorno do valor, a API nova é dividida em três funções, que contam com o suporte de uma ISR atrelada à interrupção do ADC e uma variável de controle que guarda o estado de uma conversão.

```
1  void analogRead_begin( uint8_t pin ) {
2
3    ADMUX = ( analog_reference << 6 ) | ( pin & 0x07 );
4
5    //configures interrupt
6    sbi( ADCSRA, ADIE );
7
8    // start the conversion
9    sbi( ADCSRA, ADSC );
10
11   adc_state = ADC_READING;
12 }
13
14 ISR( ADC_vect ) {
```

```
15   adc_state = ADC_DONE;
16 }
17
18 bool analogRead_available() {
19   return adc_state != ADC_READING;
20 }
21
22 int analogRead_read() {
23
24   uint8_t low, high;
25
26   low  = ADCL;
27   high = ADCH;
28
29   // combine the two bytes
30   return ( high << 8 ) | low;
31 }
```

Exemplo 8: Leitura analógica não-bloqueante

```
1  void setup(){
2      pinMode(SENSOR_PIN,INPUT);
3      pinMode(LED_BUILTIN,OUTPUT);
4
5      Serial.begin(9600);
6  }
7  void loop(){
8      int value = 0;
9      analogRead_begin(SENSOR_PIN);
10     while(!analogRead_available()){
11         doWork(); // or sleep()
12     }
13     value = analogRead_read();
14     Serial.println(value);
15 }
```

Exemplo 9: Aplicação utilizando driver não-bloqueante

A primeira função é equivalente à função de leitura da implementação original até o momento do início da conversão, isto é, ela termina quando o bit recebe o valor lógico 1. Entretanto ela configura o hardware para que as interrupções ocorram, algo

que não estava previsto na implementação da Arduino. Essa função altera o valor da variável de estado para que esta reflita que uma conversão está em andamento.

A ISR atrelada à interrupção do ADC altera o valor da variável de estado para refletir que uma interrupção terminou.

A segunda função é utilizada para consultar a variável de estado de modo a saber se uma conversão está acontecendo. Isso é necessário pois caso esteja, os valores contidos nos registradores que guardam o resultado da conversão são indefinidos.

A terceira função é equivalente à parte final da função da implementação original, após o polling. Ela simplesmente retorna o resultado da conversão contido nos registradores.

Para efetuar uma leitura de um valor analógico utilizando esta API, o usuário deve chamar a primeira função para iniciar a conversão e, depois que a segunda função retorne um valor negativo (representando que uma conversão não está em progresso), chamar a terceira função para obter o resultado. A aplicação pode realizar outras atividades entre chamadas à segunda função, o que era impossível na implementação original.

No desenvolvimento do driver em Céu, o início da conversão foi modelado como um evento de saída e a obtenção do resultado da conversão como um evento de entrada emitido pelo driver. Como a própria API é responsável por emitir o resultado quando a conversão terminar, não há necessidade de manter uma variável de estado.

Para efetuar uma conversão, a aplicação, agora em Céu, deve requisitar uma conversão e "aguardar" o evento resultado. Como previsto no ambiente Céu, enquanto a aplicação está "aguardando" o evento, o microcontrolador fica livre para executar outras linhas de código pendentes ou, caso nenhuma exista, entrar em um estado de baixo consumo de energia. Quando o hardware concluir a conversão, o evento emitido irá "acordar" a aplicação, que seguirá seu fluxo.

```
1 _Serial.begin(9600);
2 par do
3     loop do
4         emit ADC_REQUEST(SENSOR_PIN);
5         var int value = await ADC_DONE;
6         _Serial.println(value);
7     end
```

```
8  with
9      loop do
10         emit DO_WORK;
11         await WORK_DONE;
12     end
13 end
```

Exemplo 10: Aplicação utilizando driver em Céu

Nesse ponto o driver, embora funcional, possui uma limitação. Tenhamos como exemplo uma aplicação onde realizamos duas leituras analógicas em paralelo. Pelo fluxo de execução de Céu, as duas requisições serão realizadas em sequência, e aplicação terá duas trilhas aguardando o mesmo evento, que quando for emitido acordará ambas. O evento de entrada que carrega o valor, entretanto, não diferencia as requisições. Deste modo, o valor recebido por ambas as trilhas será o da última requisição, que sobreescreve a anterior.

```
1  _Serial.begin(9600);
2  par do
3      loop do
4          emit ADC_REQUEST(SENSOR_PIN1);
5          var int value = await ADC_DONE;
6          _Serial.print("PIN1 value:"); _Serial.println(value);
7      end
8  with
9      loop do
10         emit ADC_REQUEST(SENSOR_PIN2);
11         var int value = await ADC_DONE;
12         _Serial.print("PIN2 value:"); _Serial.println(value);
13     end
14 end
```

Exemplo 11: Duas leituras analógicas concorrentes não suportadas

Resolvemos esta questão incluindo no driver um vetor armazenando, para cada pino de leitura analógica, o valor mais recente e um indicador para representar que a conversão está em andamento. Um vetor de bit também foi incluído contendo todas as conversões pendentes. Com esta adição o driver, ao terminar uma conversão, pode imediatamente começar a outra com base no vetor de bits. O usuário do driver pode consultar os vetores para garantir que sua conversão terminou e obter o valor final.

Como ao final de toda conversão o driver emite um evento, a aplicação pode realizar as consultas de forma eficiente, aguardando o evento antes da consulta.

Encapsulamos esse processo de requisição e espera do resultado de forma que o usuário final do driver não precise tomar conhecimento dos vetores. Abaixo segue a mesma aplicação utilizando a abstração, resultando no comportamento desejado.

```
1  _Serial.begin(9600);
2  par do
3      loop do
4          var int value = await Analog(SENSOR_PIN1);
5          _Serial.print("PIN1 value:"); _Serial.println(value);
6      end
7  with
8      loop do
9          var int value = await Analog(SENSOR_PIN2);
10         _Serial.print("PIN2 value:"); _Serial.println(value);
11     end
12 end
```

Exemplo 12: Duas leituras analógicas concorrentes utilizando a abstração

3.2 SPI

SPI, ou Serial Peripheral Interface (interface periférica serial) é um protocolo de comunicação serial síncrono. Transmissão de dados de forma serial significa que a informação é transmitida bit a bit por uma porta. Para que os bits sejam corretamente interpretados, é necessário que haja um entendimento entre as duas partes sobre o momento que marca a transferência de um bit. No caso da comunicação serial síncrona, as duas partes dividem um sinal de clock além da linha de dados, como é representado na Figura 2. As bordas no sinal de clock marcam as transferências e, portanto, não há necessidade de se estabeler previamente a frequência, como é o caso de protocolos assíncronos como UART.

No caso do protocolo SPI, o sinal de clock é controlado por somente um dos dispositivos. Esse dispositivo é chamado de master, e é responsável por iniciar toda comunicação. A outra parte é chamada de slave, e somente interpreta o sinal do clock. Existem duas linhas de dados entre um master e um slave. Uma das linhas é

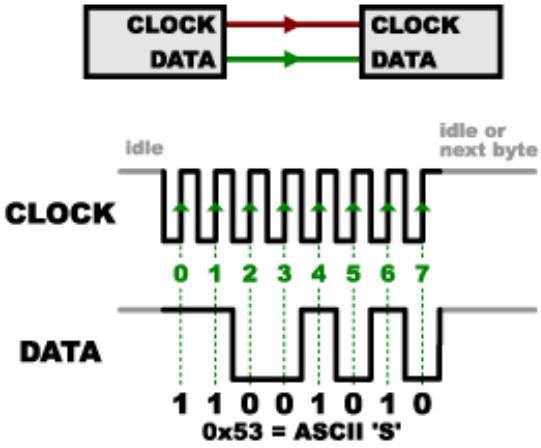


Figura 2: Comunicação serial síncrona

utilizada para transferência de dados do master para o slave e uma segunda linha para transferências do slave para o master. Essas linhas são chamadas de MOSI (Master Out Slave In) e MISO (Master In Slave Out), respectivamente. Esse arranjo permite que uma transferência SPI seja full-duplex, ou seja, para o mesmo ciclo do clock, dados são transmitidos em ambas as direções. A Figura 4 demonstra as conexões entre um master e um slave e exemplifica a transferência de um byte do master para o slave seguido de um byte na direção contrária.

As duas partes utilizando SPI não precisam se preocupar com pré-estabelecer a frequência do clock, sendo somente necessário respeitar as limitações eletrônicas dos dispositivos. Existem dois parâmetros, porém, que devem ser acordados entre as duas partes. Os dados podem ser transmitidos por ordem crescente ou decrescente de significância dos bits. O outro parâmetro que deve ser igual em ambos é o chamado de ”modo SPI”, que define a fase e polaridade do sinal de clock relativo à transferência dos bits. A Figura 3 demonstra a influência da fase (CPHA) e polaridade (POL) sobre a operação.

Uma quarta linha ainda é prevista no protocolo SPI. Essa linha, chamada de SS (Slave Select) é controlada pelo master e representa a seleção do slave. Isso significa que caso uma transação de dados esteja sendo realizada com o slave, a linha deve refletir essa seleção. A convenção é de que um estado de LOW na linha (diferença de potencial perto da referência terra) representa a seleção do dispositivo. A Figura 5 demonstra a mesma situação da Figura 4 com a adição do SS.

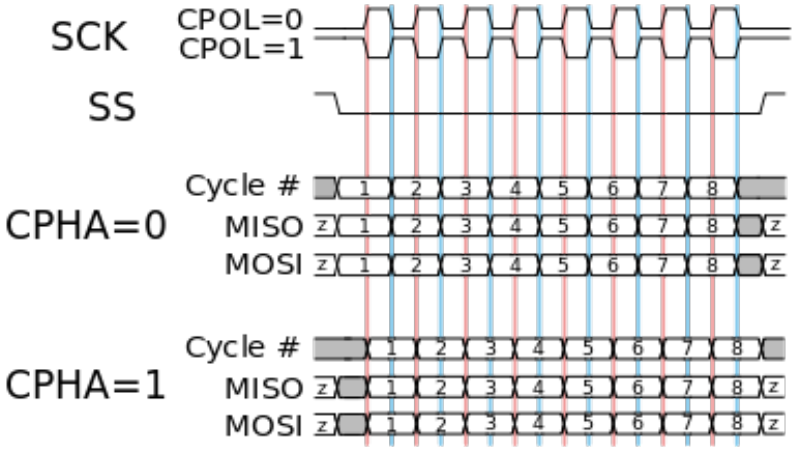


Figura 3: Comunicação serial síncrona

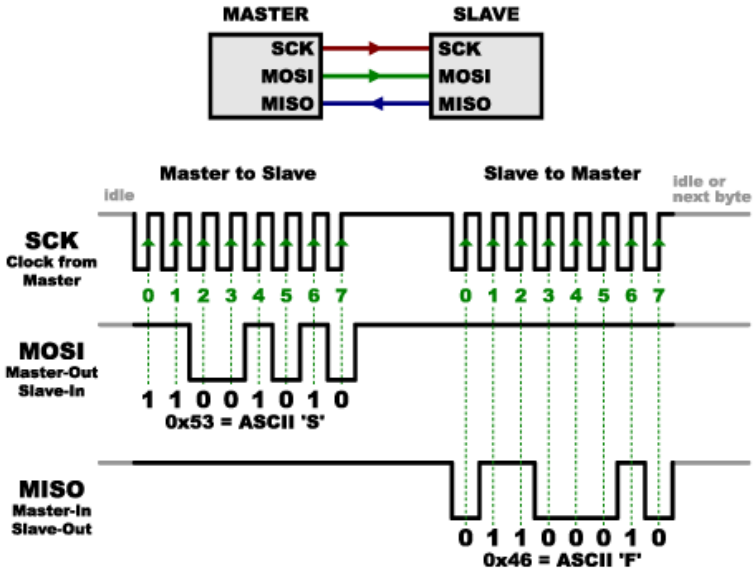


Figura 4: Transação de dados SPI

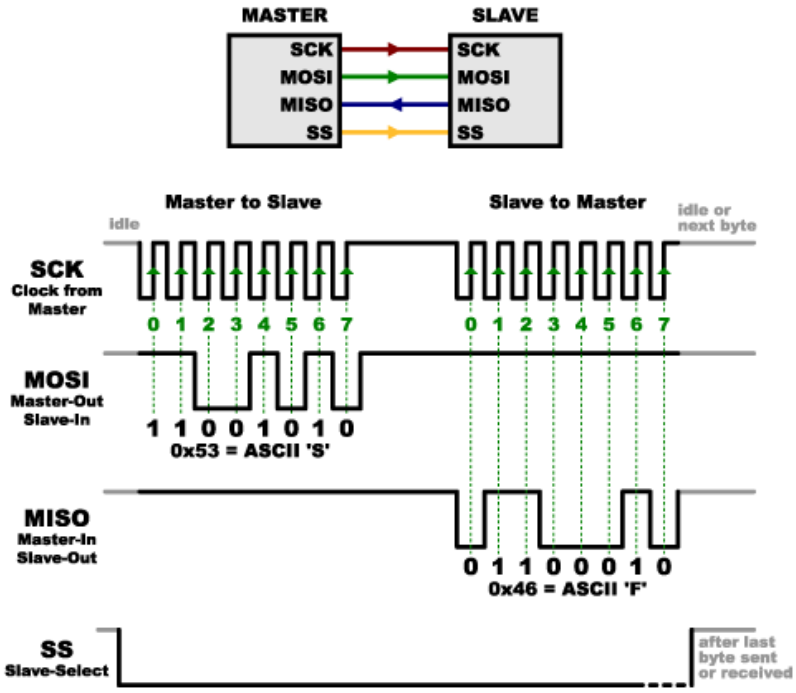


Figura 5: Transação de dados SPI com SS

O modelo de conexão do SPI, com o Slave Select, permite que mais de um slave esteja conectado ao mesmo master. Basta que haja uma linha de SS individual para cada slave para que o master seja capaz de selecionar entre os múltiplos slaves. Como o controle permite garantir que só um slave seja capaz de transmitir dados em um dado momento, a linha MISO pode ser compartilhada por todos os dispositivos. Como só o master escreve no MOSI e no clock, essas linhas também podem ser compartilhadas por todos. A Figura 6 exemplifica uma conexão entre um master e múltiplos slaves.

O ATMega328p possui hardware dedicado a SPI. O módulo consiste em múltiplos registradores de configurações e um registrador de 8 bits para dados. O hardware é responsável por transmitir o registrador de dados bit a bit pelo canal MOSI e preencher o registrador com os 8 bits recebidos pelo MISO. O hardware também controla o clock, permitindo que o microcontrolador se preocupe somente com escrever e ler dos registradores, sem se preocupar com a lógica da transmissão. Devido à limitação do tamanho do registrador, somente um byte pode ser transmitido por vez, sendo necessárias leituras e escritas ao registrador entre transmissões.

A biblioteca SPI disponibilizada pela Arduino utiliza os conceitos de transferência e transação. Enquanto uma transferência é o envio (ou recebimento) de um ou

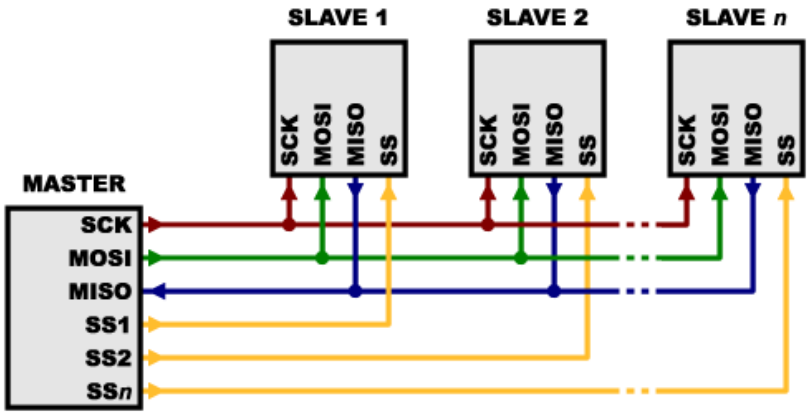


Figura 6: Transação de dados SPI com SS

uma série de bytes, uma transação define os parâmetros da comunicação e garante o acesso exclusivo ao módulo SPI. Abaixo o Exemplo 13 demonstra uma aplicação simples onde, ao apertar um botão, a aplicação trasmite por SPI o estado de um LED que está piscando.

```
1  #include <SPI.h>
2
3  int ledState = LOW;             // ledState used to set the LED
4  int buttonWasPressed = FALSE;
5
6  unsigned long previousMillis = 0;        // will store last time LED was
       updated
7  const long interval = 1000;           // interval at which to blink (
       milliseconds)
8
9  void setup() {
10   // set the digital pin as output:
11   pinMode(ledPin, OUTPUT);
12   pinMode(BUTTON, INPUT);
13   pinMode(SS, OUTPUT);
14 }
15
16 void loop() {
17   unsigned long currentMillis = millis();
18
19   if(digitalRead(BUTTON_PIN) == HIGH){
20     buttonWasPressed = TRUE;
```

```
21    }
22    if (!buttonWasPressed && currentMillis - previousMillis >= interval) {
23      // save the last time you blinked the LED
24      previousMillis = currentMillis;
25
26      // if the LED is off turn it on and vice-versa:
27      if (ledState == LOW) {
28        ledState = HIGH;
29      } else {
30        ledState = LOW;
31      }
32
33      // set the LED with the ledState of the variable:
34      digitalWrite(LED_BUILTIN, ledState);
35    } else {
36      char state = 0;
37
38      if(ledState) state = 'H';
39      else state = 'L';
40
41      digitalWrite(SS, LOW);
42
43      SPI.begin();
44      SPI.beginTransaction(SPISettings(CLOCK_RATE, BITORDER, SPI_MODE));
45      SPI.transfer(state);
46      SPI.endTransaction();
47      SPI.end();
48
49      digitalWrite(SS, HIGH);
50    }
51  }
```

Exemplo 13: Aplicação em C utilizando driver SPI bloqueante

Podemos ver que o programa, após emitir o sinal no SS, utiliza funções do driver disponibilizado para Arduino para estabelecer o escopo da transação, definir os parâmetros e obter controle do driver. O módulo é inicializado e, depois, finalizado. Isso é porque o hardware SPI pode ser desligado para não consumir recursos de energia do microcontrolador, sendo portanto uma implementação mais eficiente. Após transferir o

byte, a transação é explicitamente terminada, liberando o acesso exclusivo ao driver, e a linha SS volta ao normal. Percebemos nesse exemplo que o controle do SS é responsabilidade da aplicação, não contemplado pelo driver.

Embora já mencionado que o hardware executa a operação de transferência sem necessidade de interferência da aplicação, a função do driver é bloqueante, utilizando polling e caracterizando um desperdício de ciclos. A implementação da função pode ser vista abaixo no Exemplo 14.

```
1  inline static uint8_t transfer(uint8_t data) {
2    SPDR = data;
3    /*
4     * The following NOP introduces a small delay that can prevent the wait
5     * loop form iterating when running at the maximum speed. This gives
6     * about 10% more speed, even if it seems counter-intuitive. At lower
7     * speeds it is unnoticed.
8     */
9    asm volatile("nop");
10   while (!(SPSR & _BV(SPIF))) ; // wait
11   return SPDR;
12 }
```

Exemplo 14: Implementação bloqueante da função de transferência SPI

O comportamento bloqueante é análogo ao do módulo Analog. O byte a ser transferido pelo master é escrito no registrador, o que inicia a operação de transferência do hardware que, quando finalizar, mudará o valor de uma flag em um registrador. A função faz polling na consulta a essa flag, aguardando a finalização para, então, retornar o valor recebido do slave. Vale lembrar que, como a comunicação é full-duplex (bytes podem ser transferidos e recebidos simultaneamente pelas linhas diferentes), a função é suficiente para enviar e receber um byte. Como o master conhece o protocolo do dispositivo, qualquer valor recebido e não esperado é ignorado pela aplicação.

Como na leitura de valores analógicos, o hardware do SPI também pode ser configurado para disparar uma interrupção ao final de sua operação de transferência. Separar, portanto, a função de transferência em 3 funções (uma de início da operação, uma de consulta ao estado da operação, e uma de finalização) é suficiente para superarmos os desafios descritos já na sessão do módulo anterior.

A conclusão em relação à conversão da função de transferência em eventos

de Céu é, portanto, a mesma do módulo Analog. Como temos outras funções de configuração do driver, assim como as funções de definir o escopo da transação, modelamos tais operações como eventos de saída que realizavam as mesmas tarefas de sua contraparte no módulo original. A aplicação do Exemplo 13 tem seu mesmo comportamento refletido no Exemplo 15, utilizando o driver que desenvolvemos em Céu.

```
1  var char state = _;
2  par do
3      loop do
4          emit LED_13(on);
5          state = 'H';
6          await 1s;
7          emit LED_13(off);
8          state = 'L';
9          await 1s;
10     end
11 with
12     loop do
13         await BUTTON;
14         emit SPI_BEGIN;
15         emit PIN_SS(off);
16         emit SPI_TRANSACTION_BEGIN(CLOCK_RATE, BITORDER, SPI_MODE);
17         emit SPI_TRANSFER_REQUEST(state);
18         await SPI_TRANSFER_DONE;
19         emit SPI_TRANSACTION_END;
20         emit PIN_SS(on);
21         emit SPI_END;
22     end
23 end
```

Exemplo 15: Aplicação utilizando driver SPI em Céu

Embora o driver esteja funcional, enfrentamos um problema quando queremos realizar transações em paralelo. O motivo é que, ao iniciar uma transação, o controle exclusivo do driver é obtido, e uma nova emissão do evento de início de uma transação irá bloquear aguardando o final da transação anterior. Esse comportamento é herdado da implementação original da função, porém não pode ser mantido em Céu. O bloqueio impede que a trilha atinja o comando de await. Como a trilha não termina, a aplicação

não é capaz de reagir a eventos e, portanto, o programa congela. O Exemplo 16 exemplifica este caso.

```
1  par do
2      loop do
3          await BUTTON_1;
4          emit SPI_BEGIN;
5          emit PIN_SS1(off);
6          emit SPI_TRANSACTION_BEGIN(CLOCK_RATE, BITORDER, SPI_MODE);
7          emit SPI_TRANSFER_REQUEST(0x11);
8          await SPI_TRANSFER_DONE;
9          emit SPI_TRANSACTION_END;
10         emit PIN_SS1(on);
11         emit SPI_END;
12     end
13 with
14     loop do
15         await BUTTON_2;
16         emit SPI_BEGIN;
17         emit PIN_SS2(off);
18         emit SPI_TRANSACTION_BEGIN(CLOCK_RATE, BITORDER, SPI_MODE);
19         emit SPI_TRANSFER_REQUEST(0x22);
20         await SPI_TRANSFER_DONE;
21         emit SPI_TRANSACTION_END;
22         emit PIN_SS2(on);
23         emit SPI_END;
24     end
25 end
```

Exemplo 16: Aplicação congela pelas transações concorrentes

Para que o usuário pudesse saber quando uma transação está em andamento, introduzimos uma variável de global de controle que possuiu seu valor modificado no evento de início da transação e restaurado no evento de final. A variável pode ser verificada durante a aplicação, para que, caso o driver esteja em uso, o usuário saiba aguardar o evento de finalização da transação e, só então, iniciar a sua. O Exemplo 17 demonstra a utilização dessa variável para solucionar o comportamento do Exemplo 16. Como não há preempção dentro da execução de uma trilha, não ocorre condição de corrida em relação ao uso da variável de controle.

```
par do
    loop do
        await BUTTON_1;
        emit SPI_BEGIN;
        while spi_busy_transaction is true do
            await SPI_TRANSACTION_END;
        end
        emit PIN_SS1(off);
        emit SPI_TRANSACTION_BEGIN;(CLOCK_RATE, BITORDER, SPI_MODE)
        emit SPI_TRANSFER_REQUEST(0x11);
        await SPI_TRANSFER_DONE;
        emit SPI_TRANSACTION_END;
        emit PIN_SS1(off);
        emit SPI_END;
    end
with
    loop do
        await BUTTON_2;
        emit SPI_BEGIN;
        while spi_busy_transaction is true do
            await SPI_TRANSACTION_END;
        end
        emit PIN_SS2(off);
        emit SPI_TRANSACTION_BEGIN;(CLOCK_RATE, BITORDER, SPI_MODE)
        emit SPI_TRANSFER_REQUEST(0x22);
        await SPI_TRANSFER_DONE;
        emit SPI_TRANSACTION_END;
        emit PIN_SS2(on);
        emit SPI_END;
    end
end
```

Exemplo 17: Aplicação utiliza a variável para não congelar

A aplicação agora não congela e tem o comportamento esperado, aguardando o fim de uma transação para que a outra se inicie, divindo o controle do driver. Uma variável de contagem também é utilizada para que o hardware do SPI só seja desligado de fato quando todos os usuários do driver emitem o evento de desligamento. Esse comportamento é herdado do driver original.

Encapsulamos a solução para transações concorrentes, assim como a operação de transferência. Utilizamos um recurso de Céu que permite que o encapsulamento gere um escopo (watching). O recurso permite, ainda, definir um bloco de código que será chamado no momento que a execução sai do escopo. Aproveitamos esse comportamento para que, quando a execução da operação terminar, o evento de finalização correspondente seja chamado. A implementação pode ser vista do Exemplo 18. O resultado pode ser visto no Exemplo 19 que utiliza as operações encapsuladas, assim como no Exemplo 22 que repete o comportamento do Exemplo 15.

```
1  code/await SPI_Transaction (var u32 freq, var u8 byte_order, var u8 mode) ->
       none do
2      if outer.spi_transaction_busy then
3          await outer.spi_transaction_end until outer.spi_transaction_busy ==
       false;
4      end
5      outer.spi_transaction_busy = true;
6
7      emit SPI_TRANSACTION_BEGIN(freq,byte_order,mode);
8      do finalize with
9          emit SPI_TRANSACTION_END;
10         outer.spi_transaction_busy = false;
11         emit outer.spi_transaction_end;
12     end
13     await FOREVER;
14 end
```

Exemplo 18: Implementação do encapsulamento para transação

```
1  par do
2      loop do
3          await BUTTON_1;
4          watching SPI_Begin() do
5              emit PIN_SS1(off);
6              watching SPI_Transaction(CLOCK_RATE, BITORDER, SPI_MODE);
7                  await SPI_Transfer(0x11);
8              end
9              emit PIN_SS1(off);
10         end
11     end
```

```
12 with
13     loop do
14         await BUTTON_2;
15         watching SPI_Begin() do
16             emit PIN_SS2(off);
17             watching SPI_Transaction(CLOCK_RATE, BITORDER, SPI_MODE);
18                 await SPI_Transfer(0x22);
19             end
20             emit PIN_SS2(off);
21         end
22     end
23 end
```

Exemplo 19: Transações concorrentes utilizando operações encapsuladas

```
 1 var char state = _;
 2 par do
 3     loop do
 4         emit LED_13(on);
 5         state = 'H';
 6         await 1s;
 7         emit LED_13(off);
 8         state = 'L';
 9         await 1s;
10     end
11 with
12     loop do
13         await BUTTON;
14         watching SPI_Begin() do
15             emit PIN_SS(off);
16             watching SPI_Transaction(CLOCK_RATE, BITORDER, SPI_MODE);
17                 await SPI_Transfer(state);
18             end
19             emit PIN_SS(off);
20         end
21     end
22 end
```

Exemplo 20: Aplicação utilizando driver SPI em Céu com operações encapsuladas

4 Aplicação

4.1 Proposta

Como estudo de caso, desenvolvemos uma aplicação em sistemas embarcados que utiliza os drivers desenvolvidos. O objetivo é demonstrar o uso dos drivers em uma aplicação de sistemas embarcados com aplicações práticas. Uma aplicação distribuída foi escolhida, por tirar proveito da eficiência promovida por Céu e os drivers que desenvolvemos. Inspirado na Internet das Coisas, escolhemos um sistema de iluminação inteligente como alvo do desenvolvimento.

O objetivo do sistema é reduzir a iluminação desnecessária em ambientes. A proposta é que a iluminação a potência máxima somente ocorra nas seções onde indivíduos estão localizados. As outras seções, neste caso, devem iluminar a uma potência menor de forma a economizar recursos, que seriam gastos caso o ambiente estivesse inteiro iluminado a máxima potência. Caso não existam indivíduos no local, todas as luzes devem estar desligadas.

O sistema consiste em múltiplos nós, unidades individuais rodando uma instância da mesma aplicação. A proposta é que cada nó seja responsável por uma seção do ambiente, e que possa interagir com todos os outros nós utilizando comunicação sem fio.

O comportamento do nó é ditado por dois fatores externos: o sensor de presença e o rádio. Caso o sensor esteja apontando presença, o nó deve ligar sua iluminação, aqui representada por um LED, em sua potência máxima. Além disso, deve usar seu rádio para transmitir uma mensagem a todos os nós, periodicamente, informando da presença. Caso o sensor não aponte ou deixe de apontar presença, o nó desliga seu LED e passa a aguardar mensagens dos outros nós. Caso o nó receba uma mensagem, liga seu LED a meia potência, continuando a escutar por mensagens subsequentes. Se não receber mensagens em um intervalo de tempo, volta ao estado do LED desligado. Na situação em que o sensor aponta presença enquanto o nó está no estado de meia potência, deve ligar o LED na potência máxima e passar a enviar as mensagens, no mesmo comportamento já descrito. O diagrama de estado do comportamento descrito pode ser visto na Figura 7.

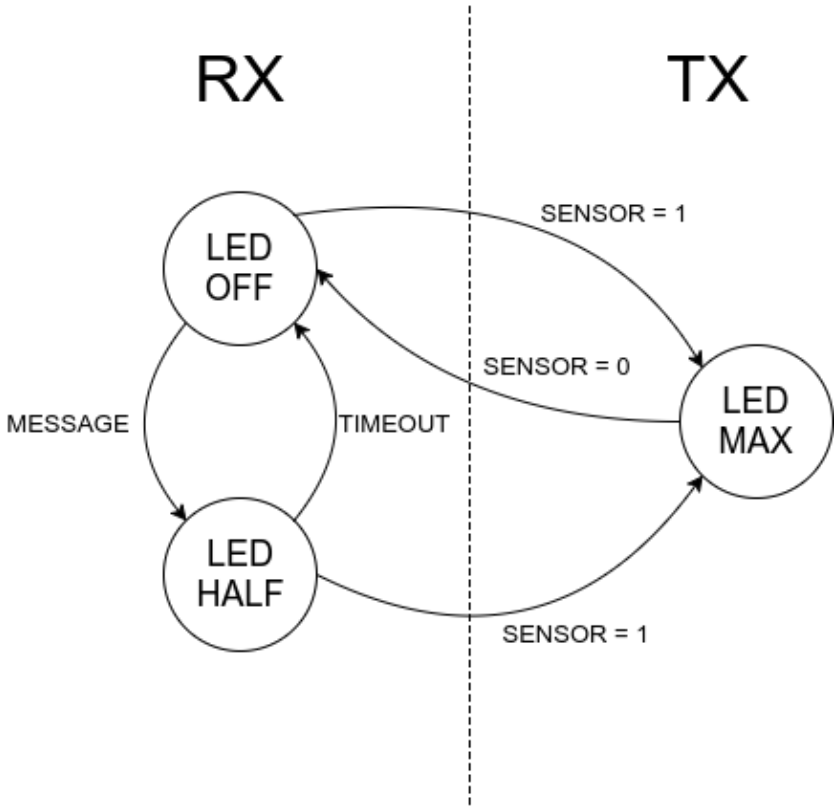


Figura 7: Diagrama de estado do comportamento de um nó do sistema

4.2 Desenvolvimento

Para o sensor de presença do sistema, assumimos utilização de medidores de distância por leituras analógicas, como sonares e sensores infra-vermelhos. Isso nos possibilita utilizar o driver Analog que desenvolvemos.

Escolhemos radio-frequência como solução de comunicação sem fio para o sistema. O módulo de rádio nRF24L01N foi escolhido por atender os requisitos necessários, estar disponível para desenvolvimento da aplicação e possuír um driver em C desenvolvido, que facilita sua utilização. O nRF24L01N é um poderoso transmissor e receptor de rádio, com uma gama de funcionalidades complexas para necessidades avançadas[10]. Para este projeto, só utilizamos as funcionalidades básicas de transmissão e recepção de um byte, somente tirando proveito das configurações do componente e seus protocolos internos para aumentar a chance de transmissão bem-sucedida.

O componente utiliza interface SPI para toda sua operação. Todos os comandos passados para o rádio, acesso a registradores e leitura de valores é feita através de transferências em SPI. Esse fato permitiu que utilizássemos o driver SPI que implementamos.

Implementamos a aplicação em C utilizando os driver desenvolvidos pela Arduino para comparação. A aplicação pode ser vista no Exemplo 21. Não mantivemos no exemplo, por motivos de clareza, sessões de código da aplicação desenvolvida cujo propósito era configuração do rádio.

```
1  // The role of the current running sketch
2  role_e role = role_none;
3  role_e newRole = role_none;
4
5  unsigned long timeLastMessage = 0;
6
7  void setup(void)
8  {
9    pinMode(SENSOR_PIN, INPUT);
10   pinMode(LED_PIN, OUTPUT);
11
12   radio.begin();
13 }
```

```
14
15 void loop(void)
16 {
17
18  if(analogRead(SENSOR_PIN) > SENSOR_THRESHOLD){
19   newRole = role_speak;
20  } else {
21   newRole = role_listen;
22  }
23  // If new role, change
24  if(newRole != role){
25   role = newRole;
26   if(role == role_listen){
27    timeLastMessage = millis();
28    radio.startListening();
29   } else {
30    analogWrite(LED_PIN,FULL_POWER);
31    radio.stopListening();
32   }
33  }
34   if (role == role_speak)
35   {
36     uint8_t radio_id = RADIO_ID;
37     radio.write( &radio_id, sizeof(uint8_t) );
38     delay(SEND_INTERVAL);
39   }
40   if ( role == role_listen )
41   {
42     // if there is data ready
43     if ( radio.available() )
44     {
45       uint8_t signal_id = 0;
46       radio.read( &signal_id, sizeof(uint8_t) );
47     }
48       analogWrite(LED_PIN,HALF_POWER);
49       timeLastMessage = millis();
50     }
51     if(millis() - timeLastMessage > SEND_INTERVAL * 2){
52       analogWrite(LED_PIN,LOW);
```

```
53      }
54    }
55 }
```

Exemplo 21: Aplicação de iluminação inteligente em C

A aplicação mostra todos os problemas já discutidos das linguagens procedurais para sistemas deste tipo. Além de utilizar as funções bloqueantes dos drivers, para que saiba se uma mensagem foi recebida, polling é feito no estado do rádio.

Além dos pinos dedicados ao protocolo SPI, o componente de rádio possuiu um pino para interrupção. Quando configurado, o pino emite um sinal em momentos pré-definidos da operação do rádio. No desenvolvimento desse sistema, configuramos o pino de interrupção para reagir somente quando o rádio recebe uma mensagem. Isso permitiu que conectássemos esse sinal ao pino de interrupção externa do Arduino, possibilitando a execução de uma interrupção na chegada de uma mensagem e, consequentemente, a implementação de um evento em Céu de chegada de mensagem, analogamente aos eventos de entrada correspondente às interrupções dos drivers desenvolvidos. Isso permitiu tornar o processo de aguardar uma mensagem do rádio mais eficiente. A alternativa em C, como é possível ver no Exemplo 21, é realizar polling.

A aplicação deve constantemente verificar se há presença no ambiente. Utilizamos o driver que desenvolvemos e, portanto, as leituras estão sendo feitas de forma eficiente e utilizando interrupção.

Embora toda a operação ao rádio seja feita com SPI, possibilitando o uso do driver que desenvolvemos para todas as etapas da operação, a aplicação em Céu-Arduino resultante acaba por ser de um tamanho superior ao suportado pela ROM do microcontrolador. Além disso, a RAM disponível não é suficiente para comportar a pilha da aplicação durante múltiplas chamadas aninhadas. Portanto, por limitação do kit de desenvolvimento Céu-Arduino só utilizamos o driver na operação mais executada no envio de mensagens, a transferência do byte a ser transmitido pelo rádio. Para as demais operações, optamos por utilizar as funções já implementadas do driver em C, mesmo que bloqueantes. Devido à frequência inferior que essas operações são executadas quando comparada com as operações de envio e recebimento, essa foi a decisão de menos impacto.

O produto final em Céu pode ser visto no Exemplo **??**. Do mesmo modo que no Exemplo 21, não mantivemos no exemplo, por motivos de clareza, sessões de código da aplicação desenvolvida cujo propósito era configuração do rádio. A primitiva spawn inicializa uma trilha que executa em paralelo com o resto da aplicação, nete caso sendo responsável por gerar os eventos correspondentes à detecção de presença. A aplicação foi capaz de evitar os bloqueios causados pelos drivers SPI e Analog de sua contraparte em C, assim como evitar o polling no estado do rádio.

```
1  code/await Write_register (var u8 reg, var u8 value) -> u8 do
2      var u8 status = _;
3      watching SPI_Transaction(4 * MHZ, SPI_MSBFIRST, SPI_MODE0) do
4          csn(off);
5          status = await SPI_Transfer(_W_REGISTER | ( _REGISTER_MASK & reg ) )
       ;
6          await SPI_Transfer(value);
7          csn(on);
8      end
9      escape status;
10 end
11 emit SPI_BEGIN;
12 emit RADIO_BEGIN;
13 // Keep checking the button
14 spawn do
15     loop do
16         var int value = await Analog(0);
17         if value > SENSOR_THRESHOLD then
18             emit presence;
19         else
20             emit no_presence;
21         end
22     end
23 end
24 loop do
25     par/and do
26         par/or do
27             // abort when PRESENCE
28             await presence;
29         with
30             // LISTEN TO NEIGHBOURS
```

```
31              emit RADIO_RECEIVER;
32              loop do
33                  par/or do
34                      await Radio_receive();
35                      _analogWrite(LED_PIN,HALF_POWER);
36                  with
37                      await (SEND_INTERVAL * 2)ms;
38                      _analogWrite(LED_PIN,off);
39                  end
40              end
41          end
42      with
43          await presence;
44          // SEND TO NEIGHBOURS
45          _analogWrite(LED_PIN,FULL_POWER);
46          emit RADIO_TRANSMITTER;
47          par/or do
48              await no_presence;
49              _analogWrite(LED_PIN,off);
50          with
51              loop do
52                  await 1500 us;
53                  // write the payload
54                  await Write_register(_W_TX_PAYLOAD, RADIO_ID);
55                  ce(on);
56                  await 15 us;
57                  {delayMicroseconds(15);}
58                  ce(off);
59                  await SEND_INTERVAL ms;
60              end
61          end
62      end
63  end
```

Exemplo 22: Aplicação utilizando driver SPI em Céu com operações encapsuladas

5 Comentários Finais

Implementar os drivers utilizando interrupção e utilizar o ambiente de Céu permitiu gerar um produto final de aplicação mais eficiente e de lógica mais clara que originalmente em C, como desejado. Produzimos um sistema de aplicação real, e obtivemos o funcionamento desejado.

A maior dificuldade encontrada foi a utilização do kit Céu-Arduino. O kit ainda se encontra em estado de desenvolvimento e sofre mudanças constantes. A documentação não é atualizada compativelmente, o que dificulta o uso por terceiros. O tamanho da aplicação após a compilação ainda é grande demais para soluções mais complexas. O uso de reursos de abstrações em Céu, como definição de chamadas e escopos, aumenta a pilha em um ritmo que torna impossível realizar chamadas aninhadas, visto que o espaço disponível para a pilha do AtMega328p não suporta o tamanho. Esses fatores foram limitantes quando desenvolvendo o driver SPI e inviabilizou o desenvolvimento de um driver para o nRF24L01.

O trabalho que realizamos nos drivers e na aplicação se encontra publicado no projeto Céu-Arduino[7], como uma contribuição à comunidade desenvolvedora. Embora o ambiente Céu-Arduino ainda possua suas limitações, o desenvolvimento dos drivers provou recompensar em ganho de eficiência, além de ser reproduzível para outros módulos. O desenvolvimento realizado pode ser continuado em forma da implementação de outros drivers, continuando o enriquecimento do ambiente Céu-Arduino e estimulando o amadurecimento do kit.

Referências

[1] S. Dhananjay e T. Gaurav, “A survey of internet-of-things: Future vision, architecture, challenges and services”, *2014 IEEE World Forum on Internet of Things, WF-IoT 2014*, mar. de 2014.

[2] F. Wortmann e K. Flütcher, “Internet of things - technology and value added”, *Business & Information Systems Engineering*, vol. 57, pp. 221–224, 3 2015.

[3] M. Chui, M. Löffer e R. Roberts, eds., *The Internet of Things*, McKinsey Quarterly, mar. de 2010. endereço: `https://www.mckinsey.com/industries/high-tech/our-insights/the-internet-of-things`.

[4] S. Edwards, L. Lavagno, E. A. Lee e A. Sangiovanni-Vincentelli, “Design of embedded systems: Formal models, validation, and synthesis”, *Proceedings of the IEEE*, vol. 85, nº 3, pp. 366–390, mar. de 1997, issn: 0018-9219. doi: `10.1109/5.558710`.

[5] F. Sant’Anna, N. de La Rocque Rodriguez e R. Ierusalimschy, “Céu: Embedded, safe, and reactive”, *Monografias em Ciência da Computação*, vol. 9, 2012, issn: 0103-9741.

[6] Arduino. (2017). Arduino blog, endereço: `https://blog.arduino.cc`.

[7] F. Sant’Anna. (2017). Github céu-arduino. Acessado em 22 de Janeiro de 2018, endereço: `https://github.com/fsantanna/ceu-arduino`.

[8] AtMel, *Atmel atmega328/p datasheet*, ATMel, 2016.

[9] F. Sant’Anna. (2017). Github céu. Acessado em 22 de Janeiro de 2018, endereço: `https://github.com/fsantanna/ceu`.

[10] N. Semiconductor, *Nrf24l01 datasheet*, Nordic Semiconductor, 2007.