Desenvolvimento de firmware robusto e multiplataforma

Jonathan Gonzaga[1] **and Orientador**[2]

[1]Graduando em Engenharia de Computação, UNISAL São José - Campinas, jonathan.s.gonzaga@gmail.com
[2]Professor do UNISAL São José - Campinas, orientador@sj.unisal.br

Resumo - *Este artigo apresenta técnicas e ferramentas para o desenvolvimento de firmware robusto e multiplataforma, com o intuito de minimizar bugs, permitir a implementação de testes unitários e incentivar o estudo de conceitos de Engenharia de software aplicados a sistemas embarcados.*

Palavras-chave: *sitemas embarcados, firmware, TDD, testes unitários*

Abstract - *This article presents techniques and tools for robust and multiplatform firmware development, in order to minimize bugs, allow the implementation of unit tests and encourage the study of software engineering concepts applied to embedded systems.*

Keywords: *embedded systems, firmware, TDD, unit tests*

I. INTRODUÇÃO

Com o avanço da eletrônica e da computação, a miniaturização de circuitos integrados e a redução de custos de fabricação, surgiram na indústria diversos dispositivos eletrônicos com poder de processamento. A maioria desses dispositivos conta com processadores, memórias e periféricos integrados, de forma que seu *software* é destinado e *embarcado* em uma aplicação específica. Sistemas dessa natureza são conhecidos como *sistemas embarcados*.

Produtos como eletrodomésticos, eletrônicos em geral, equipamentos médicos, equipamentos de telecomunicações, ferramentas eletrônicas, sistemas de controle e automação e sistemas de tempo real em veículos são ou possuem sistemas embarcados em sua concepção.

Devido ao alto nível de criticidade de alguns sistemas embarcados, é esperado que o *software* executado neles seja altamente confiável e com o mínimo de *bugs* possível. A realidade da indústria de eletrônicos mostra que essa afirmação nem sempre é verdadeira.

Erros e falhas de *software* são um problema em diversas áreas da tecnologia. Desde vulnerabilidades que facilitam a ação de *hackers* em redes sociais, falhas em *smartphones* que causam travamento do sistema operacional a inconsistências no acionamento de sistemas de freios que causam acidentes em veículos. Esse último caso é ainda mais grave, pois o sistema lida diretamente com vidas humanas.

Especialistas em desenvolvimento de *firmware* e *software* embarcado, como Jack Ganssle, James W. Grenning e Jacob Beningo, dedicaram-se a criar literatura de qualidade, escrevendo livros e artigos com técnicas e metodologias de desenvolvimento de *software*. Alguns títulos relevantes são: *Test-Driven development for Embedded C* [1], *The Art of Designing Embedded Systems*[2] e *Reusable Firmware Development*[3]. A partir dessas obras muito se tem discutido sobre como criar sistemas mais seguros, com menor probabilidade de erros e menos suscetíveis a má utilização por parte do usuário.

No processo de desenvolvimento de *software*, o custo de uma alteração no código tende a ficar maior conforme as etapas do desenvolvimento avançam, por isso o custo de um *bug* encontrado em campo após o produto ser lançado, é muito maior que o custo do mesmo sendo encontrado ainda na fase de desenvolvimento[4]. Esse cenário por si só já mostra a necessidade de se detectar problemas nas fases iniciais do projeto.

Software para sistemas embarcados muitas vezes é difícil de ser testado e validado, extremamente dependente da plataforma de *hardware target*, e tende a demonstrar problemas de integração com outras partes do sistema após a inserção de novas funcionalidades. Devido a limitação de recursos e a extrema dependência do *hardware*, testes automatizados não são realizados, o que acaba prejudicando a qualidade do código.

Metodologias e conceitos como *TDD (Test-Driven development)*, pirâmide de testes e *SOLID*, são historicamente e erroneamente atribuídos apenas aos *softwares* de "alto nível", como *web* ou *mobile* por exemplo, afastando os desenvolvedores de *software* embarcado desses conceitos e da utilização de abstrações mais inteligentes.

Com essas dificuldades em mente, percebe-se a necessidade de se estimular as boas práticas de desenvolvimento de *software* embarcado, a utilização de testes automatizados e independentes do *hardware*, e o estudo de conceitos de engenharia de *software* que auxiliem na redução de *bugs* e consequentemente, redução de custos do projeto e aumento da qualidade do código gerado.

II. REFERENCIAL TEÓRICO

A. Firmware

Firmware é um tipo específico de *software* executado diretamente num circuito integrado (ou *chip*). Não necessita de outros programas para ser executado (como sistemas operacionais), além de servir a um propósito único. Em outras palavras, é o *software* executado em

um sistema embarcado[5].

Devido a limitações de tamanho e recursos desse tipo de sistema, o *firmware* precisa manipular o *hardware* diretamente e toda a sua arquitetura costuma ser voltada a eventos do mundo externo.

Diferente de sistemas computacionais mais complexos e com alto nível de abstração, onde geralmente o *kernel* do sistema operacional é modularizado e abstrai o acesso a dispositivos de *hardware*[6], num sistema embarcado o *firmware* é responsável pela gerência dos recursos e eventos de *hardware* (interrupções e exceções do processador) e também pelo código da aplicação (regras de negócio, interface com usuário e demais especificidades).

Muitas vezes ele faz parte de um sistema computacional maior, por exemplo, computadores *desktop* possuem circuitos integrados para aplicações específicas, como a *BIOS* (*Basic Input/Output System*), que trata-se de um mecanismo responsável pela inicialização dos componentes de *hardware* do computador. [7]

Algumas literaturas não fazem distinção entres os termos *firmware* e *software embarcado*, sendo ambos utilizados como sinônimos. Por uma questão de organização, será realizada uma distinção entre esses termos: *firmware* é o *software* executado num circuito integrado comumente escrito em linguagem *C, C++*, ou mesmo *Assembly*. Já o *software embarcado* possui características de alta abstração, sendo uma aplicação (um processo rodando num sistema operacional, geralmente baseado em *GNU/Linux*)[8] porém executado em dispositivos de propósito específico (como roteadores e terminais de autoatendimento). Pode ser escrito em linguagens de programação como *C, C++, Rust, Go, Python* entre outras.

Em um sistema embarcado executando um *firmware*, o cérebro por trás de todo o processamento é um componente chamado *microcontrolador*.

B. Microcontroladores

Microcontroladores são processadores de pequeno porte contendo *CPU*, memórias (*RAM* e *FLASH*) e demais recursos atrelados no mesmo encapsulamento ou *SoC (System on Chip)*. Esses recursos são denominados periféricos, e são inclusos no *chip* com o intuito de tornar possível o interfaceamento entre a *CPU* e o mundo externo, permitir a comunicação com outros dispositivos e aquisitar dados para posterior processamento através dos pinos físicos do componente.

Os recursos mais comuns são as interfaces de comunicação serial (*USART, I2C, SPI*), conversores analógico-digital e digital-analógico (*ADC* e *DAC*), temporizadores/contadores (*TIMERS*) e interface de pinos (*GPIO*).

Com tamanho e custo reduzidos em comparação ao microprocessadores, os microcontroladores tornam-se economicamente viáveis, além da escolha óbvia, para controlar digitalmente dispositivos e processos[9].

C. Linguagem de programação - *C*

C é uma linguagem de programação procedural de propósito geral que suporta programação estruturada, escopo de variável lexical e recursão, com um sistema de tipo estático. Por design, *C* fornece construções que mapeiam de forma eficiente para instruções de máquina típicas. Dispõe de compiladores para a maioria das arquiteturas de computador e sistemas operacionais existentes.[10]

Devido a essas características, vem sendo utilizada em aplicações previamente codificados em linguagem *assembly*. Algumas dessas aplicações são: sistemas operacionais, compiladores, *drivers* de dispositivos, bibliotecas com acesso a *hardware* e sistemas embarcados.

Leva esse nome por ser sucessora da linguagem de programação *B*. Foi desenvolvida no *Bell Labs* por Dennis Ritchie entre 1972 e 1973 para construir utilitários rodando em *Unix*. Em 1989 foi padronizada pelo *ANSI* (*ANSI C*) e pela *International Organization for Standardization (ISO)*. [10]

D. Testes de *software*

Testes de *software* são procedimentos pelos quais o código fonte de um *software* é submetido afim de validar seu comportamento mediante os requisitos pelos quais foi projetado.

Existem diversos tipos de testes de *software*, cada um responsável por testar partes e situações diferentes. A chamada *Pirâmide de testes* [11] foi concebida com o intuito de seguimentar os testes em três grandes níveis:

- *Testes end-to-end* - simulam a aplicação final
- *Testes de integração* - testa a integração entre testes de unidade
- *Testes unitários* - testes de unidade, verificam a menor unidade de código testável

Apenas o último nível, os testes unitários, serão abordados nesse documento.

E. Testes unitários

Testes unitários são a base dos testes de *software*. Na *Pirâmide de testes* encontram-se no nível mais baixo, o que indica que são o tipo mais barato e fácil de implementar[11], fazendo com que sejam os mais numerosos em relação aos demais tipos de teste.

Como seu nome sugere, são testes de *unidade*, onde é possível testar a menor unidade testável do código (sejam funções, classes ou módulos).

F. *TDD - Test-Driven development*

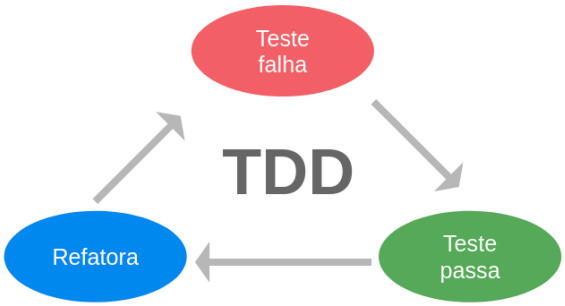
Test-Driven development (ou *Desenvolvimento orientado a testes*) é uma técnica incremental de construção de *software*. Nessa técnica, nenhum código de produção é escrito sem que primeiro seja escrito um teste unitário que falhe na primeira execução.

Ao contrário da prática comum de desenvolvimento de *software*, onde primeiro é desenvolvido o código de produção e só depois os testes, no TDD o desenvolvedor expressa o comportamento desejado do código em um teste. O teste é executado e falha. Só então ele escreve o código de produção, fazendo o teste passar. A automação de testes é a chave para o *TDD*. Os testes são pequenos e automatizados. A cada nova funcionalidade implementada, novos testes unitários são escritos, seguidos imediatamente por um código de produção que satisfaça aqueles testes.

Conforme o código de produção cresce, também crescem em conjunto os testes unitários, que são ativos tão valiosos quanto o próprio código de produção. A cada mudança de código, o conjunto de testes é executado, verificando a funcionalidade da nova implementação, mas também a compatibilidade com o código já existente[1].

Um esquema simplificado do *TDD* é apresentado na Figura 1:

Fig. 1. Esquema simplificado do TDD



Fonte: acervo do autor

Três grande etapas são necessárias: *Teste falha*, *Teste passa* e *Refatora*. Na primeira, são escritos apenas os testes unitários, com base nos requisitos do projeto, porém sem nenhum código de produção ainda, o que obviamente fará com que os testes falhem após a execução.

Logo após, são escritos apenas os trechos de código necessários para que aqueles testes passem e nada mais.

Na última etapa, o código passará pelo processo de *refatoração*, onde serão eliminados possíveis erros, duplicidade, e será formatado de acordo com o padrão requisitado.

G. *Framework* de testes - *Unity*

Unity é um *framework* de testes escrito em linguagem C, pensado principalmente para *software* em sistemas embarcados.

Trabalha com o conceito de *asserts*, que nada mais são que formas de se garantir e verificar parâmetros (entrada) e resultados (saída) de funções no código fonte.[12]

H. Ferramenta de dublês de testes - *CMock*

CMock é uma ferramenta de dublês de testes, usada para simular o comportamento de funções que possuam dependência externa (*hardware* ou bibliotecas externas).

Trabalha com os conceitos de *mocks* e *stubs*, que nada mais são que funções *fake*, criadas com o intuito de emular o comportamento (entradas e saídas) de componentes do código que tenham alguma dependência.[13]

I. *Build system - Ceedling*

Ceedling é um *build system* (sistema de construção de *software*) para projetos escritos em linguagem C. É uma espécie de extensão do sistema de construção *Rake (make-ish)* do *Ruby*. O *Ceedling* é voltado principalmente para o *Desenvolvimento Orientado a Testes (TDD)* em linguagem C e reune o *framework* de testes *Unity*, a ferramenta de dublês de testes *CMock* e a ferramenta de manipulação de exceções *CException* - três outros projetos de código aberto que auxiliam na dinâmica de testes automatizados[14].

J. Sistema operacional de tempo real - *FreeRTOS*

FreeRTOS é um *kernel* de tempo real, ou um sistema operacional de tempo real (*Real-Time Operating System*) para dispositivos embarcados. Foi desenvolvido para ser pequeno, simples e portável. Seu *kernel* é composto por apenas 3 arquivos em linguagem C. O *FreeRTOS* permite a fácil implementação de multitarefa preemptiva ou não preemptiva com diversos níveis de prioridade de tarefas[15].

K. *Build system - GNU Make*

GNU Make é um *build system* (sistema de construção de *software*) que controla a geração de executáveis de um *software* a partir dos arquivos de código fonte.

A ferramenta *Make* recebe as instruções de como construir o programa a partir de um arquivo chamado *makefile*, que lista cada um dos arquivos de código fonte e os comandos para transformá-los em executáveis.

Ao escrever um programa, deve-se escrever um *makefile* para ele, de modo que seja possível usar o *Make* para construção e execução do mesmo. [16]

L. Compilador - *GCC*

O *GNU Compiler Collection (GCC)* é um compilador de otimização produzido pelo *Projeto GNU* que oferece suporte a várias linguagens de programação, arquiteturas de *hardware* e sistemas operacionais.

A *Free Software Foundation (FSF)* distribui o *GCC* como *software livre* sob a licença *GNU General Public License (GNU GPL)*. O *GCC* é um componente chave da cadeia de ferramentas *GNU* e o compilador padrão para a maioria dos projetos relacionados ao *GNU* e ao *kernel Linux*.[17]

M. Editor de textos - *Visual Studio Code*

O *Visual Studio Code* é um editor de código-fonte *freeware* feito pela *Microsoft* para *Windows*, *Linux* e *macOS*. Os recursos incluem suporte para depuração, destaque de sintaxe, autocompletar de código inteligente, *snippets*, refatoração de código e *Git* incluso.[18]

III. METODOLOGIA

A. MATERIAIS E MÉTODOS

Serão utilizadas as seguintes ferramentas de *software* na execução do projeto:

- *FreeRTOS* - sistema operacional de tempo real
- *GNU make - Build system* para compilação do código de produção
- *GCC(x86)* e *GCC(arm)*- compiladores para ambas arquiteturas
- *VS Code* - editor de textos
- *Ceedling - Build system* para compilação e execução dos testes
- *Unity - framework* de testes
- *Cmock* - ferramenta de dublê de testes

A metodologia utilizada para o desenvolvimento do projeto segue a técnica de *TDD* descrita anteriormente.

Nas três etapas (*Teste falha*, *Teste passa* e *Refatora*) serão utilizadas ferramentas e mecanismos de *software* para auxílio na implementação dos testes, execução e automatização de processos

B. DESENVOLVIMENTO

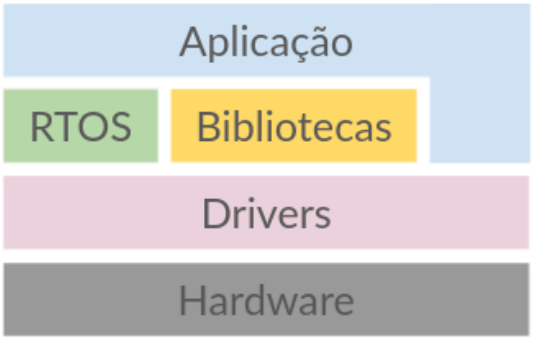
O processo de desenvolvimento do projeto seguiu o padrão especificado anteriormente, juntamente com algumas regras e boas práticas:

- *Clean Code* (Código limpo) - o código precisa ser de fácil entendimento[19]
- *KISS (Keep it simple stupid)* - as soluções implementadas precisam ser simples[20]
- *SOLID principles* - seguir os padrões do *SOLID* [21]
- *Boy Scout Principle* (Regra do escoteiro) - *"Deixe o campo mais limpo do que quando o encontrou"* - Sempre deixe o código melhor do antes de você trabalhar nele[19]
- *Refactoring* (Refatoração) - o código precisa ser refatorado e aprimorado constatemente[19]

B.1. Arquitetura de software do sistema

Os componentes de *software* separados por nível de abstração estão ilustrados na Figura 2.

Fig. 2. Arquitetura de software do projeto



Fonte: acervo do autor

A camada de *Aplicação* faz uso dos *Drivers* para controle dos periféricos de *hardware*, utiliza o *RTOS* para criar e gerenciar *threads* e recursos compartilhados por elas e as *Bibliotecas* fornecem alguma funcionalidade extra como *file system* (sistema de arquivos) ou cálculo de *PID (Proporcional - Integral -Derivativo)* por exemplo.

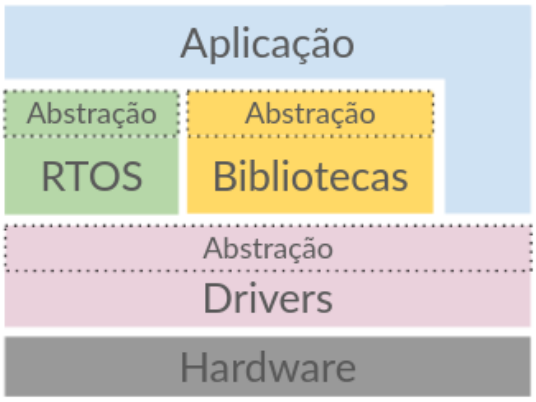
B.2. Camadas de abstração - drivers, Bibliotecas e RTOS

Um dos pilares do desenvolvimento de *software* multiplataforma é a abstração. Utilizar interfaces bem definidas a partir de *header files* (arquivos de extensão *.h*), variando apenas a implementação dessas interfaces nos *source files* (arquivos de extensão *.c*).

Para se escrever código portável é extremamente necessário a não dependência de *Drivers*, *RTOS - (Real Time Operating Systems)* e demais bibliotecas

A abordagem consiste em criar camadas de abstração que acessem essas camadas, como pode ser visto na Figura 3:

Fig. 3. Arquitetura de software com abstrações



Fonte: acervo do autor

Essas camadas são apenas funções que encapsulam as funções dos módulos originais, fazendo com que o código da *Aplicação* nunca realize uma chamada de função diretamente a esses componentes, mas sim as suas abstrações.

Imaginemos um exemplo de módulo dependente de *hardware*, responsável pelo acionamento de pinos de um microcontrolador. O módulo de *GPIO - General Purpose Input Output*. Esse módulo é um *driver* e está localizado na camada de mesmo nome:

```
#ifndef __GPIO_H__
#define __GPIO_H__

void gpio_config(int port, int pin);
int gpio_read(int port, int pin);
void gpio_write(int port, int pin, int level);

#endif
```

Exemplo 1. Interface do módulo de GPIO - gpio.h

```
#include "gpio.h"

void gpio_config(int port, int pin)
{
    GPIO.port.CFG |= pin;
}

int gpio_read(int port, int pin)
{
    return GPIO.port.RD.pin;
}

void gpio_write(int port, int pin, int level)
{
    GPIO.port.WR.pin = level;
}
```

Exemplo 2. Implementação do módulo de GPIO - gpio.c

O módulo contém funções que dependem do *hardware*, porém as mesmas não devem ser chamadas diretamente pela aplicação, mas sim por uma abstração:

```
#ifndef __GPIO_IFACE_H__
#define __GPIO_IFACE_H__

#include "gpio.h"

void gpio_iface_config(int port, int pin);
int gpio_iface_read(int port, int pin);
void gpio_iface_write(int port, int pin,
                      int level);
#endif
```

Exemplo 3. Interface pública do módulo - gpio_iface.h

```
#include "gpio_iface.h"

void gpio_iface_config(int port, int pin)
{
    // Chama a funcao do modulo original
    gpio_config(port, pin);
}

int gpio_iface_read(int port, int pin)
{
    // Chama a funcao do modulo original
    return gpio_read(port, pin);
}

void gpio_iface_write(int port, int pin, int
    level)
{
    // Chama a funcao do modulo original
    gpio_write(port, pin, level);
}
```

Exemplo 4. Implementação da abstração para acesso ao módulo de GPIO - gpio_iface.c

A abstração do módulo, chamada de de *gpio_iface.c e gpio_iface.h* (*iface* é uma abreviação para *interface*) abstrai o acesso, permitindo que seja feita a utilização de dublês de testes (*mocks, fakes e stubs*), visando a implementação de testes unitários.

A aplicação final faz uso apenas do módulo de abstração:

```
#include <stdio.h>
// Camada de aplicacao nao inclui o modulo
// diretamente, mas sim sua abstracao
#include "gpio_iface.h"

int main(void)
{
    gpio_iface_config(PORT_C, PIN_13);

    // Loop infinito
    while (1) {
        // Classico exemplo de pisca led
        gpio_iface_write(PORT_C, PIN_13, 1);
        delay_ms(500);
        gpio_iface_write(PORT_C, PIN_13, 0);
        delay_ms(500);
    }
    return 0;
}
```

Exemplo 5. Camada de aplicação - main.c

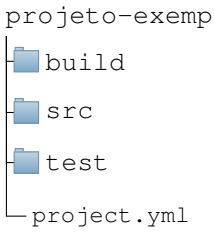
B.3. Implementando testes unitários

A implementação de testes unitários exige a utilização de um *framework* de testes. Foi utilizado o *Unity*, ferramenta já contida no *Ceedling*. Usando os comandos do próprio *build system* para a criação do projeto, temos a seguinte linha de comando:

```
$ ceedling new projeto-exemplo
```

Exemplo 6. Criando um projeto com o *Ceedling*

Uma estrutura de diretórios semelhante a mostrada abaixo é criada:



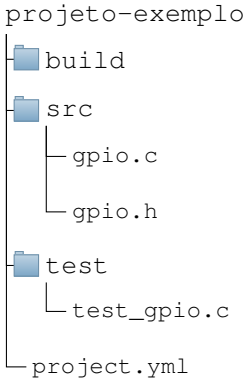
- *build* - contém os artefatos de *software*, arquivos temporários e executáveis
- *src* - contém o código fonte da aplicação
- *test* - contém o código fonte dos testes
- *project.yml* - contém a configuração relativa ao projeto

Para criar o primeiro módulo, usamos *ceedling module:create* passando o nome do módulo como argumento:

```
$ ceedling module:create[gpio]
```

Exemplo 7. Criando um módulo com o *Ceedling*

A ferramenta criará arquivos *source* e *header* em *src* e um arquivo de testes em *test*:



No arquivo de testes, são criadas as funções, *setUp()* e *tearDown()* e adicionado o *header file unity.h*:

```c
#include "unity.h"
#include "gpio.h"

void setUp(void)
{
}

void tearDown(void)
{
}

void test_gpio_NeedToImplement(void)
{
    TEST_IGNORE_MESSAGE(
        "Need to Implement gpio");
}
```

Exemplo 8. Arquivo de testes inicial

Utilizando as abstrações demonstradas anteriormente, os primeiros testes podem ser implementados:

```c
#include "unity.h"
#include "gpio.h"

void setUp(void)
{
}

void tearDown(void)
{
}

void test_gpio(void)
{
    TEST_ASSERT_EQUAL(0,
        gpio_iface_read(PORT_C, PIN_13));
}
```

Exemplo 9. Arquivo de testes com um teste simples

B.4. Utilizando dublês de teste

Dublês de teste são utilizados nos casos em que há dependência de algum componente externo e o módulo precisa ser testado. O *driver* de *GPIO* é um exemplo de dependência de *hardware*, nesse caso, é interessante o uso de dublês de teste. A ferramenta responsável pela geração desses dublês é o *Cmock*.

```c
#include "unity.h"
#include "mock_gpio.h"

void setUp(void)
{
}

void tearDown(void)
{
}

// Callback utilizada ser chamada pelo Cmock
int gpio_iface_read_callback(int
    cmock_num_calls)
{
    return 1;
}
```

```

void test_gpio(void)
{
    // Registrando a callback
    gpio_iface_read_StubWithCallback(
        gpio_iface_read_callback);

    TEST_ASSERT_EQUAL(0,
        gpio_iface_read(PORT_C, PIN_13));
}
```

Exemplo 10. Arquivo de testes com mock

No código anterior uma função de *callback* é definida e registrada no *Cmock*, de modo que toda vez que a função original for chamada, *gpio_iface_read()*, a ferramenta redireciona a chamada para *gpio_iface_read_callback()*. Para isso é necessário apenas incluir o *header file* com o prefixo *mock_*: *mock_gpio.h*.

O argumento *cmock_num_calls* na *callback* é uma obrigatoriedade da ferramenta, para que haja o controle do número de chamadas realizadas a *callback*.

```
$ ceedling test:all
Test 'test_gpio.c'
-----------------------------
Generating include list for gpio.h...
Creating mock for gpio...
Generating runner for test_gpio.c...
Compiling test_gpio_runner.c...
Compiling test_gpio.c...
Compiling mock_gpio.c...
Compiling unity.c...
Compiling gpio.c...
Compiling cmock.c...
Linking gpio.out...
Running gpio.out...

---------------------
OVERALL TEST SUMMARY
---------------------
TESTED:  1
PASSED:  1
FAILED:  0
IGNORED: 0
```

Exemplo 11. Rodando o primeiro teste com *Ceedling*

B.5. Dual-targeting: compilando para duas arquiteturas

O processo de *Dual-targeting* só é possível quando se consegue independência entre os módulos de software.

IV. CONCLUSÃO

Conclui-se que

Referências

1 GRENNING, J. W. *Test Driven Development for Embedded C*. [S.l.]: Pragmatic bookshelf, 2011.

2 GANSSLE, J. *The art of designing embedded systems*. [S.l.]: Newnes, 2008.

3 BENINGO, J. *Reusable Firmware Development*. [S.l.]: Springer, 2017.

4 EMBARCADOS. Editorial: custo do firmware. 2015. Disponível em: <https://www.embarcados.com.br/editorial-custo-do-firmware/>. Acesso em: 26.02.2021.

5 GANSSLE, J. *The firmware handbook*. [S.l.]: Elsevier, 2004.

6 TANENBAUM, A. S.; BOS, H. *Modern operating systems*. [S.l.]: Pearson, 2015.

7 TERZIĆ, A.; AKELJIĆ, B. Basic input/output system bios functions and modifications. *International University Travnik*.

8 SIMMONDS, C. *Mastering embedded Linux programming*. [S.l.]: Packt Publishing Ltd, 2015.

9 GRIDLING, G.; WEISS, B. Introduction to microcontrollers. *Vienna University of Technology Institute of Computer Engineering Embedded Computing Systems Group*, 2007.

10 RITCHIE, D. M. The development of the c language. *ACM Sigplan Notices*, v. 28, n. 3, 1993.

11 CONTAN, A.; DEHELEAN, C.; MICLEA, L. *Test automation pyramid from theory to practice*. [S.l.: s.n.], 2018.

12 THROWTHESWITCH. Unity - unit testing for c (especially embedded software). 2021. Disponível em: <http://www.throwtheswitch.org/cmock>. Acesso em: 25.05.2021.

13 THROWTHESWITCH. How cmock works. 2021. Disponível em: <http://www.throwtheswitch.org/cmock>. Acesso em: 25.05.2021.

14 GOMES, E. C.; AMORA, P. R.; TEIXEIRA, E. M.; LIMA, A. G.; BRITO, F. T.; CIOCARI, J. F.; MACHADO, J. C. Uttos: A tool for testing uefi code in os environment. In: SPRINGER. *IFIP International Conference on Testing Software and Systems*. [S.l.], 2016. p. 218–224.

15 ZHU, M.-Y. Understanding freertos: A requirement analysis. *CoreTek Syst., Inc., Beijing, China, Tech. Rep*, 2016.

16 FOUNDATION, F. S. Gnu make. 2021. Disponível em: <https://www.gnu.org/software/make/>. Acesso em: 25.05.2021.

17 FOUNDATION, F. S. Gcc, the gnu compiler collection. 2021. Disponível em: <https://gcc.gnu.org/>. Acesso em: 25.05.2021.

18 CORPORATION, M. Visual studio code. 2021. Disponível em: <https://code.visualstudio.com/>. Acesso em: 25.05.2021.

19 MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. [S.l.]: Pearson Education, 2009.

20 MARTIN, R. C. *Clean architecture: a craftsman's guide to software structure and design*. [S.l.]: Prentice Hall, 2018.

21 MARTIN, R. C. *Agile software development: principles, patterns, and practices*. [S.l.]: Prentice Hall, 2002.