

Desenvolvimento de firmware robusto e multiplataforma

Jonathan Gonzaga¹ and Orientador²

¹Graduando em Engenharia de Computação, UNISAL São José - Campinas, jonathan.s.gonzaga@gmail.com

²Professor 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. É utilizada uma aplicação de exemplo, uma interface gráfica para requisição de amostras de números verdadeiramente aleatórios (TRNG - True Random Number Generator), capaz de rodar em ambiente desktop e embarcado.

Palavras-chave: sistemas 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. An example application is used, a graphical interface for requesting samples of truly random numbers (TRNG - True Random Number Generator), capable of running in both desktop and embedded environments.

Keywords: embedded systems, firmware, TDD, unit tests

1. 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 entre 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 adquirir 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 seguir 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

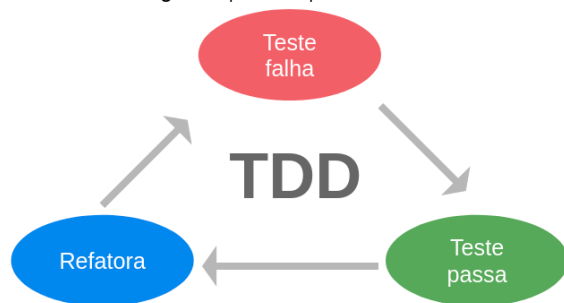
Test-Driven development (ou *Desenvolvimento orientado a testes*) é uma técnica incremental de construção de *software*. Nessa técnica, nenhum código de produção é escrito sem que primeiro seja escrito um teste unitário que falhe na primeira execução.

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^[11].

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 reúne 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 portátil. 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]

N. *TRNG* - *True Random Number Generator*

Um gerador de números verdadeiramente aleatórios, ou *TRNG* - *True Random Number Generator* é um circuito capaz de gerar números aleatórios através de algum processo físico ao invés dos métodos convencionais de uso de algoritmos para o mesmo fim. Alguns dos possíveis fenômenos físicos que podem ser utilizados por uma *TRNG* são:

- Ruído térmico
- Ruído analógico
- Fenômeno fotoelétrico
- Fenômeno quântico

O objetivo geração de números verdadeiramente aleatórios remete ao uso da criptografia, como geração de chaves criptográficas.

Em microcontroladores que possuem periféricos avançados, os *TRNG* são implementados em silício, num circuito que utiliza os ruídos analógicos de alimentação e temperatura.^[19]

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^[20]
- *KISS* (*Keep it simple stupid*) - as soluções implementadas precisam ser simples^[21]
- *SOLID principles* - seguir os padrões do *SOLID*^[22]
- *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^[20]
- *Refactoring* (Refatoração) - o código precisa ser refatorado e aprimorado constantemente^[20]

B.1. Definindo a aplicação e suas funcionalidades

A aplicação escolhida para demonstração das técnicas trata-se de uma interface gráfica para geração de números verdadeiramente aleatórios (*TRNG - True Random Number Generator*).

Essa interface gráfica será executada em dois ambientes diferentes: embarcado e *desktop*. As funcionalidades, interfaces de aplicação (*API - application programming interface*) e visual serão exatamente os mesmos, variando-se apenas as implementações de código dependente de plataforma.

Haverão duas abas disponíveis para o usuário, a primeira, chamada de *Mass storage*, permitirá que seja salvo um arquivo no sistema de arquivos de tamanho e nomes a serem escolhidos pelo usuário. Essa aba conterá uma caixa de texto para receber o nome do arquivo, um *spinbox* para se definir o tamanho do arquivo, e um botão de salvar.

A segunda aba chamada de *Random*, permitirá que o usuário escolha um limite mínimo e máximo, e ao clicar em *Get random*, irá obter um número aleatório dentro desse limite anteriormente definido. Essa aba conterá dois *spinbox* para escolha dos valores mínimo e máximo, e uma *label* para mostrar o resultado.

B.2. Definindo os casos de teste

A prática do *TDD* exige que os requisitos de um *software* estejam muito bem definidos desde o início para que se possa começar a escrever os testes, por isso é necessário que se defina uma lista de testes.

Testes na aba *Mass storage*:

- Checar a sanidade do *RNG* (dados não devem se repetir com frequência)
- Checar tamanho do nome do arquivo (deve respeitar o limite do sistema de arquivos)
- Checar se o arquivo foi realmente criado

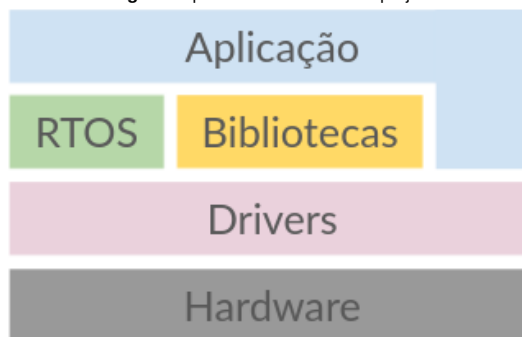
Testes na aba *Random*:

- Checar se os limites mínimo e máximo são respeitados (mínimo deve ser menor que máximo)
- Checar se o resultado dado realmente está no intervalo definido

B.3. 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 criação de *GUI* (*Graphical User Interface*) por exemplo.

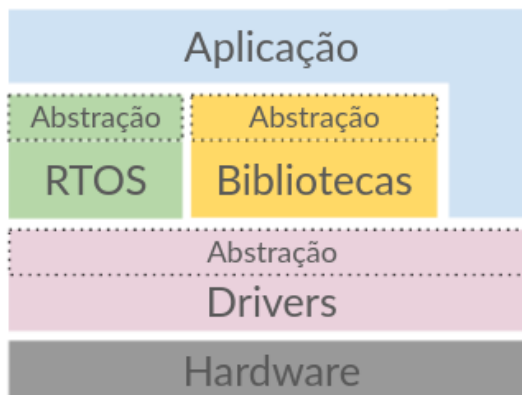
B.4. Camadas de abstração - drivers, Bibliotecas e RTOS

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 portátil 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. A prática da escrita de código dessa maneira é chamada de *Dual-targeting*.

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

O processo de *Dual-targeting* exige independência entre os módulos de *software* e principalmente da arquitetura de *hardware*. Escrever *software* embarcado independente de arquitetura exige claro entendimento dos limites entre o código da aplicação e demais códigos (bibliotecas, *drivers*).

B.6. Abstraindo um módulo com dependência

Um módulo dependente de *hardware*, responsável pela aquisição de amostras de números verdadeiramente aleatórias, módulo de *RNG - Random Number Generator*. Este módulo é um *driver* e está localizado na camada de mesmo nome, sua interface e implementação estão representadas nas Figuras 4 e 5.

Fig. 4. Interface do módulo de RNG - *rng.h*

```
1 #ifndef __RNG_H__
2 #define __RNG_H__
3
4 void rng_init(rng_config_st *rng);
5 uint8_t rng_get(rng_config_st *rng);
6
7 #endif
```

Fonte: acervo do autor

Fig. 5. Implementação do módulo de RNG - *rng.c*

```
1 #include "rng.h"
2
3 void rng_init(rng_config_st *rng)
4 {
5     // Acesso a registradores de hardware
6     RNG->periph.cfg_st = rng;
7 }
8
9 uint8_t rng_get(rng_config_st *rng)
10 {
11     // Acesso a registradores de hardware
12     return RNG->periph.data;
13 }
```

Fonte: acervo do autor

O módulo contém funções que dependem do *hardware*, porém as mesmas não devem ser chamadas diretamente pela aplicação, mas sim por uma camada de abstração mostrada nas Figuras 6 e 7.

Fig. 6. Interface pública do módulo - *rng_iface.h*

```
1 #ifndef __RNG_IFACE_H__
2 #define __RNG_IFACE_H__
3
4 void rng_iface_init(rng_config_st *rng);
5 uint8_t rng_iface_get(rng_config_st *rng);
6
7 #endif
```

Fonte: acervo do autor

Fig. 7. Implementação da abstração para acesso ao módulo de RNG *rng_iface.c*

```
1 #include "rng_iface.h"
2
3 // Inclui header do modulo original
4 #include "rng.h"
5
6 void rng_iface_init(rng_config_st *rng)
7 {
8     // Chama funcao do modulo original
9     rng_init(rng);
10 }
11
12 uint8_t rng_iface_get(rng_config_st *rng)
13 {
14     // Chama funcao do modulo original
15     return rng_get(rng);
16 }
```

Fonte: acervo do autor

A abstração do módulo, chamada de *rng_iface.c* e *rng_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, como apresentado na Figura 8:

Fig. 8. Camada de aplicação - *main.c*

```
1 #include <stdio.h>
2 // Camada de aplicacao nao inclui o modulo
3 // diretamente, mas sim sua abstracao
4 #include "rng_iface.h"
5 #define RNG_SAMPLES 32
6
7 int main(void)
8 {
9     rng_config_st rng;
10     rng_iface_init(&rng);
11     uint8_t rng_buffer[RNG_SAMPLES];
12
13     // Loop infinito
14     while (1) {
15         // Obtem 32 amostras de RNG
16         for (int i = 0; i < RNG_SAMPLES; i++) {
17             rng_buffer[i] = rng_iface_get(&rng);
18         }
19     }
20     return 0;
21 }
```

Fonte: acervo do autor

B.7. 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, tem-se a seguinte linha de comando apresentada na Figura 9.

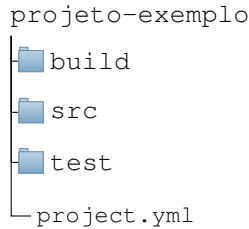
Fig. 9. Criando um projeto com o Ceedling

```
1 $ ceedling new projeto-exemplo
```

Fonte: acervo do autor

Foi criada uma estrutura de diretórios semelhante a mostrada na Figura 10.

Fig. 10. Estrutura de diretórios criada pelo Ceedling



Fonte: acervo do autor

- *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, utilizou-se *ceedling module:create* passando o nome do módulo como argumento segundo mostra a Figura 11:

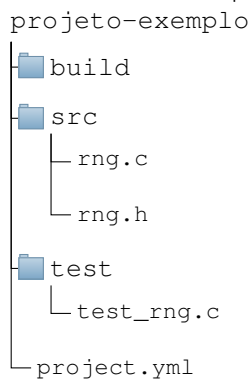
Fig. 11. Criando um módulo com o Ceedling

```
1 $ ceedling module:create[rng]
```

Fonte: acervo do autor

A ferramenta criará arquivos *source* e *header* em *src* e um arquivo de testes em *test* como mostrado na Figura 12:

Fig. 12. Estrutura de diretórios com arquivos criados



Fonte: acervo do autor

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

como mostra a Figura 13:

Fig. 13. Arquivo de testes inicial - *test_rng.c*

```
1 #include "unity.h"
2 #include "rng.h"
3
4 void setUp(void)
5 {
6 }
7
8 void tearDown(void)
9 {
10 }
11
12 void test_rng_NeedToImplement(void)
13 {
14     TEST_IGNORE_MESSAGE("Need to Implement rng");
15 }
```

Fonte: acervo do autor

Utilizando as abstrações demonstradas anteriormente, os primeiros testes podem ser implementados, como demonstrado na Figura 14:

Fig. 14. Arquivo de testes com um teste simples - *test_rng.c*

```
1 #include "unity.h"
2 #include "rng.h"
3
4 void setUp(void)
5 {
6     rng_config_st rng;
7     rng_iface_init(&rng);
8 }
9
10 void tearDown(void)
11 {
12 }
13
14 void test_rng(void)
15 {
16     uint8_t rng_buffer;
17     rng_buffer = rng_iface_get();
18     TEST_ASSERT_NOT_EQUAL(0, rng_buffer);
19 }
```

Fonte: acervo do autor

B.8. 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 *RNG* 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*. A Figura 15 mostra um simples exemplo de utilização:

Fig. 15. Arquivo de testes com mock - *test_rng.c*

```
1 #include "unity.h"
2 #include "mock_rng.h"
3 #include <stdlib.h>
4
5 // Callback chamada pelo Cmock
6 int rng_iface_get_callback(int cmock_num_calls)
7 {
8     // Usa funcao rand() da linguagem C
9     return rand();
10 }
11
12 void test_rng(void)
13 {
14     uint8_t rng_buffer;
15
16     // Registrando a callback
17     rng_iface_get_StubWithCallback(
18         rng_iface_get_callback);
19
20     rng_buffer = rng_iface_get();
21     TEST_ASSERT_NOT_EQUAL(0, rng_buffer);
22 }
```

Fonte: acervo do autor

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

A Figura 16 mostra a saída do comando resultado da execução de todos os testes:

Fig. 16. Rodando o primeiro teste com *Ceedling*

```
1 $ ceedling test:all
2 Test 'test_rng.c'
3 -----
4 Generating include list for rng.h...
5 Creating mock for rng...
6 Generating runner for test_rng.c...
7 Compiling test_rng_runner.c...
8 Compiling test_rng.c...
9 Compiling mock_rng.c...
10 Compiling unity.c...
11 Compiling rng.c...
12 Compiling cmock.c...
13 Linking rng.out...
14 Running rng.out...
15
16 -----
17 OVERALL TEST SUMMARY
18 -----
19 TESTED: 1
20 PASSED: 1
21 FAILED: 0
22 IGNORED: 0
```

Fonte: acervo do autor

B.9. Desenvolvendo interface gráfica

O processo de desenvolvimento da interface gráfica

B.10. Executando a aplicação em ambiente desktop

B.11. Executando a aplicação em ambiente embarcado

B.12. Code coverage - cobertura de código dos testes

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. Utos: 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 VASYLTSOV. *Fast digital TRNG*. [S.l.: s.n.], 2008.
- 20 MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. [S.l.]: Pearson Education, 2009.
- 21 MARTIN, R. C. *Clean architecture: a craftsman's guide to software structure and design*. [S.l.]: Prentice Hall, 2018.
- 22 MARTIN, R. C. *Agile software development: principles, patterns, and practices*. [S.l.]: Prentice Hall, 2002.