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. 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* [10] 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.

D. 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, mas também os mais fáceis de se implementar[10].

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

E. *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.

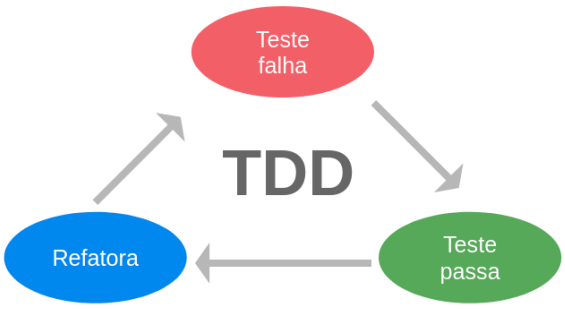
Ao contrário da prática comum de desenvolvimento de *software*, onde primeiro é desenvolvido o código de produção e só depois os testes, no TDD o desenvolvedor expressa o comportamento desejado do código em um teste. O teste é executado e falha. Só então ele escreve o código de produção, fazendo o teste 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 é apresentado na Figura 01:

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.

F. *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.

G. 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.

H. *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 exceção *CException* - três outros projetos de código aberto que auxiliam na dinâmica de testes automatizados[11].

I. 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[12].

III. METODOLOGIA

A. MATERIAIS E MÉTODOS

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

- *FreeRTOS* - sistema operacional de tempo real
- *Ceedling* - *Build system* com *framework* de testes *Unity*
- *GCC(x86)* e *GCC(arm)*- compiladores para duas arquiteturas distintas
- *VS Code* - editor de textos
- *make* - ferramenta de *Build system* para compilação

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:

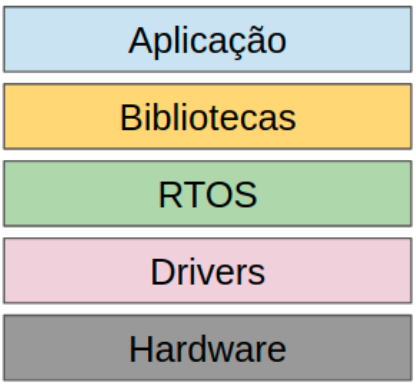
- *Clean Code* (Código limpo) - o código precisa ser de fácil entendimento[13]

- *KISS (Keep it simple stupid)* - as soluções implementadas precisam ser simples[14]
- *SOLID principles* - seguir os padrões do *SOLID* [15]

B.1. Arquitetura de software do sistema

Os componentes de *software* separados por nível de abstração estão ilustrados na Figura 02.

Fig. 2. Arquitetura de software do projeto



Fonte: acervo do autor

B.2. Dual-targeting: compilando para duas arquiteturas

Um dos pilares do desenvolvimento de *software* multiplataforma é a abstração. Utilizar interfaces bem definidas apartir de *header files*, variando apenas a implementação dessas interfaces nos *source files*.

B.3. Camadas de abstração - drivers e RTOS

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

B.4. Testes unitários

Para escrever testes unitários, é necessário a utilização de um *framework*

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

11 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.

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

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

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

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