REVISTA BRASILEIRA DE MECATRÔNICA
FACULDADE SENAI DE TECNOLOGIA MECATRÔNICA

MIGRAÇÃO EFICAZ DE PROCESSADORES EM SISTEMAS EMBARCADOS: UMA PROPOSTA METODOLÓGICA, DESAFIOS E APLICAÇÃO EM CASO REAL

EFFECTIVE MIGRATION OF PROCESSORS IN EMBEDDED SYSTEMS: A METHODOLOGICAL PROPOSAL, CHALLENGES AND APPLICATION IN REAL-WORLD CASE

Leandro Poloni Dantas[1], [i]

Data de submissão: (3/6/2024) Data de aprovação: (22/11/2024)

RESUMO
O início da década de 2020 foi marcado por uma escassez global de semicondutores que acelerou a necessidade de modernização de processadores aplicados a sistemas embarcados em vários setores. Este artigo apresenta uma metodologia de onze etapas para migração de código legado e substituição de arquitetura de processador, com um estudo de caso focado na atualização de um processador da década de 1990 utilizado em um controlador de semáforo para uma arquitetura ARM moderna, preservando a compatibilidade do firmware. Esta abordagem estruturada oferece orientação prática para portabilidade de código e adaptação de hardware, contribuindo para as metodologias de desenvolvimento contínuo em sistemas embarcados.

Palavras-chave: metodologia; portabilidade; arquitetura de processador; sistema embarcado.

ABSTRACT
The early 2020s were marked by a global shortage of semiconductors that accelerated the need to modernize processors applied to embedded systems in various sectors. This paper presents an eleven-step methodology for legacy code migration and processor architecture replacement, with a case study focused on upgrading a 1990s-era processor used in a traffic light controller to a modern ARM architecture while preserving firmware compatibility. This structured approach offers practical guidance for code portability and hardware adaptation, contributing to continuous development methodologies in embedded systems.

Keywords: methodology; portability; processor architecture; embedded system.

1 INTRODUÇÃO

O início da terceira década do século XXI nos mostrou a imensa dependência da

[1] Professor Dr. no Centro Universitário SENAI São Paulo – Campus “Anchieta” e do Insper. E-mail: leandro.poloni@sp.senai.br.

humanidade pela indústria de semicondutores. A pandemia de covid-19 associada à falta de chuvas em Taiwan criaram o senário perfeito para quebra na cadeia produtiva dos semicondutores (Junqueira, 2021). O impacto disso se viu em toda a indústria que produz bens e equipamentos que faz seu uso. Vimos montadoras parando, entregando carros em atraso ou com acessórios faltando. Fabricantes de eletroeletrônicos com dificuldades de abastecer o mercado e preços disparando. No topo pela disputa por semicondutores estiveram os fabricantes de celulares e computadores que foram fortemente impactados pela falta de processadores de última geração. Nesse seguimento, que trabalha de forma frenética e depende fortemente de inovação para manter suas vendas, essa crise gerou impacto tal, que fez os EUA repensar sua estratégia de produção na Ásia e investir bilhões de dólares na criação de um novo parque industrial dentro do país (Thorbecke, 2022).

Dadas essas condições, os grandes fabricantes de semicondutores, como TSMC, priorizaram seus grandes clientes e não tiveram mais como abastecer o mercado *low-end* (de baixa tecnologia) nos prazos usuais. Isso fez com que diversos modelos de chips sumissem do mercado e os preços disparassem. Viu-se casos em que um microcontrolador de 8 bits fosse vendido no mercado paralelo a mais de 10 vezes seu preço usual. Também ocorreram casos de falsificação de componente (Kaur, 2022) s. O próprio autor deste artigo passou por isso, quando descobriu que um sensor de temperatura analógico que houvera comprado na verdade era um transistor bipolar rotulado como sensor.

Tentando fugir dessa escassez, diversos fabricantes precisaram adequar seus produtos e modernizá-los passando a utilizar processadores mais modernos e que ainda se encontravam no mercado. Uma vez que a obsolescência de certos processadores já não recomendados para novos projetos foi antecipada pelos fabricantes.

Uma vítima, dessa obsolescência antecipada, foi a indústria de máquinas e equipamentos para fins industriais que, mesmo tendo se modernizado e sendo muito competitiva, apresenta ciclos de renovação de produtos mais longos, quando comparado aos ciclos de bens de consumo já citados. Em alguns casos, o uso de um mesmo processador se perpetua por décadas. Uma vez que, a confiabilidade do produto se deve, até um certo ponto, à confiança no processador e ao *firmware* embarcado.

Este artigo traz um estudo de caso sobre os desafios, técnicas aplicadas e meandros inerentes a migração de família de processadores e portabilidade de código aplicado um sistema embarcado. O caso apresentado, trata de uma unidade central de processamento (CPU) desenvolvida na década de 1990 e que é aplicada em controladores semafóricos.

Apesar do assunto portabilidade de código ter se tornado mais frequente em livros e artigos publicados nos últimos anos, a maior parte dos trabalhos trata o assunto de maneira teórica e com exemplos pontuais (Martin, 2008; Software Reliability Enhancement Center, Technology Headquarters e Information-technology Promotion Agency, 2018) , muitas vezes exclusivos a sistemas computacionais. Este artigo, por outro lado, traz uma abordagem mais ampla aplicada a um caso real com considerável nível de complexidade. Essa complexidade permitiu que o autor colecionasse uma série de exemplos e estratégias adotadas durante seu desenvolvimento entre os anos de 2023 e 2024. Além disso, permitiu a estruturação de uma metodologia criada exclusivamente para potabilidade de código e migração de processadores em sistemas embarcados.

A metodologia proposta tem como objetivo servir como pilar balizador a outros projetos de caráter semelhante, porém sem a pretensão de ser um modelo de solução absoluta e inflexível. Pelo contrário, abre portas para colaborações e discussão a respeito do tema.

2 REVISÃO DE LITERATURA

Barr (2018), no livro *Embedded C coding standard*, ainda na introdução, cita a importância da portabilidade de código e faz um alerta quanto a linguagem C permitir uma grande variação de compiladores por conta da linguagem apresentar alguns padrões indefinidos ou não especificados. Isso pode implicar que, um mesmo código compilado em diferentes compiladores se comporte de maneira diferente, reduzindo assim a portabilidade de código mal elaborado. Faz ainda um alerta aos programadores de que a portabilidade associada a confiabilidade, a legibilidade e a eficiência são mais importantes que conveniências pessoais.

Seguindo sua introdução, deixa claro que *bugs* são inerentes a todo código criado ou legado e sua geração pode ser minimizada quando padrões de codificação são seguidos.
No seu quinto capítulo, Barr apresenta um dos grandes empecilhos à portabilidade de códigos C, os tipos de dados com tamanhos dependentes dos compiladores/processadores (e.g. *short*, *int* e *long*). Apesar de não haver solução definitiva, o padrão C99 introduziu o arquivo *header* stdint.h com tipos inteiros de tamanhos explícitos (e.g. *int8_t*, *int16_t* e *int32_t*).

Outra referência importante no que tange o desenvolvimento de código C para sistemas embarcados é publicada pela agência japonesa de tecnologia da informação (IPA), intitulado de *Embedded system development coding reference guide* (Software Reliability Enhancement Center, Technology Headquarters e Information-technology Promotion Agency, 2018), em sua terceira versão é dividida em quatro pilares relevantes para o desenvolvimento de código embarcado e padronização de estilos visando a qualidade de código-fonte. São esses os quatro pilares: confiabilidade, manutenibilidade, portabilidade e eficiência. Dois desses pilares estão fortemente relacionados a este trabalho, a manutenibilidade e a portabilidade. O capítulo sobre manutenibilidade, parte da constatação de código-fonte é reutilizado e mantido por engenheiros que não foram seus criadores originais e por conta disso, é necessário escrever o código da forma mais simples de entender. A partir disso, traz uma série de dicas e recomendações de estilo de codificação com exemplos da forma correta e errada de proceder. Já o capítulo sobre portabilidade, inicia fazendo o mesmo alerta feito por Barr, (2018) quanto as variações de compiladores e sua dependência. Deixa claro que, um bom estilo de programação é aquele com implementação independente do compilador. Seguindo o estilo do capítulo sobre manutenibilidade, faz uma série de recomendações de como proceder durante a codificação para evitar a dependência do compilador. Em todos os casos apresenta exemplo em concordância e discordância com as recomendações. O documento é finalizado por um capítulo sobre erros comuns em codificação de sistemas embarcados.

James O. Coplien faz no prefácio do famoso livro *Clean code* (código limpo em português) (Martin, 2008) uma correlação entre as linhas de manufatura orientadas ao aumento de produção e metodologias de trabalho, como Scrum e Agile, que são amplamente aplicadas por equipes de desenvolvimento de software visando maior velocidade na produção de código. Citando o caso das linhas de produção de veículos japonesas, onde a maior parte do trabalho não reside na produção, e sim na manutenção ou prevenção. Afirma que 80% do tempo gasto em software está na manutenção, no sentido de reparar algo. Por isso, sugere que ao invés de pensar no “bom software”, o foco deveria de voltar a manutenibilidade e exemplifica isso no sistema japonês criado em 1951 chamado de Manutenção Produtiva Total (TPM) que utiliza como um pilar fundamental os princípios 5S. Dentre os princípios, enfatiza a

importância do *shutsuke* (disciplina ou autodisciplina em português) relacionado a capacidade de seguir práticas e disposição a mudar. Sugere a aplicação do TPM, de forma a evitar que os problemas apareçam, deve-se inspecionar o código e refatorá-lo impiedosamente. Ainda cita a Fred Brooks, que adverte da necessidade de refazer grandes partes do software do zero a cada sete anos ou mais para eliminar o "lixo" crescente.

No restante do livro, são descritas boas práticas de como criar um código eficaz e traz uma série de estudos de caso com foco no código limpo.

Quando nos referimos a trabalhos com certa similaridade ao aqui proposto, percebemos uma escassez de trabalhos contemporâneos. Mullins e Pack (2002), abordam a migração entre dois modelos de microcontrolador com uma abordagem estritamente acadêmica e aplicado as aulas de laboratório da Academia das Forças Armados dos Estados Unidos. Jijiraj (2013), discute a necessidade de reestruturar aplicativos para sistemas embarcados de vários núcleos na indústria automotiva e enfatiza a importância da portabilidade de aplicativos para plataformas *multicore*. Além disso, apresenta desafios que incluem segurança de *thread*, comunicação, sincronização e possíveis problemas, como inversões de prioridade. Descreve o procedimento adotado e faz comparações entre diferentes compiladores e os desempenhos alcançados.

Um ponto em comum nos dois primeiros trabalhos é o embasamento no padrão MISRA C (MISRA Consortium Limited, [s.d.]), que apresenta diretrizes de codificação para a linguagem C definidas pela Associação de Confiabilidade de Software da Indústria Automobilística (MISRA) no Reino Unido. Essa associação foi uma das pioneiras a definir um padrão a ser obrigatoriamente seguido na indústria europeia. Atualmente, as regras MISCA C estão incorporadas em muitas ferramentas de desenvolvimento de código, automatizando a tarefa de verificação, feita muitas vezes de forma manual.

Outro ponto comum, agora aos três primeiros trabalhos, é o consenso quanto a importância na manutenção e portabilidade de código, porém em nenhuma das referências existe uma abordagem clara de como proceder.

A constatada escassez de trabalhos publicados sobre migração tecnológica e portabilidade, associada a falta detalhamento na maneira de proceder são as lacunas que este trabalho pretende começar a preencher.

3 CONTEXTUALIZAÇÃO E METODOLOGIA FPFV

Iniciamos está seção pela apresentação do projeto que serviu de inspiração e referência para o desenvolvimento da metodologia de portabilidade de *firmware* e validação funcional de hardware dedicado, identificada pela sigla FPFV (*Firmware Portability and Functional Validation* em inglês). A segunda parte desta seção trata da metodologia em si e a descrição detalhada de suas onze etapas.

3.1 Migração de processador em controlador semafórico

O projeto base deste artigo trata da migração tecnológica entre famílias de processadores de gerações diametralmente opostas aplicados a um sistema embarcado. Nos parágrafos seguintes são descritas as condições para execução do projeto. Por questões de sigilo industrial, todas as informações são apresentadas de forma a ocultar a empresa envolvida e detalhes estratégicos do produto.

Durante um período de 18 meses, entre os anos de 2023 e 2024, foi executada a

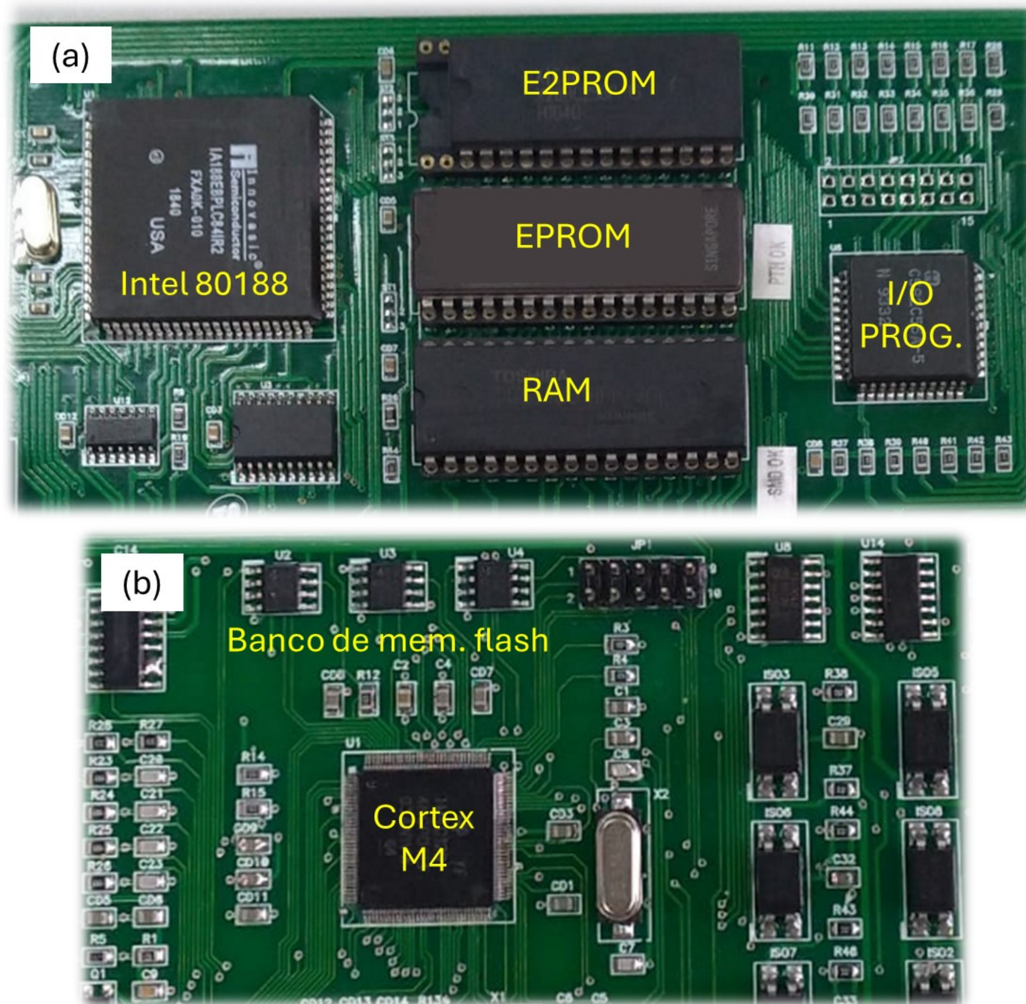
migração entre um modelo de microprocessador de 16 bits baseado no conjunto de instruções Intel x86, o Intel 80188, para um microcontrolador ARM de 32 bits série Cortex-M4, o Texas Tiva TM4C1294NCPDT. O fato de um ser um microprocessador (baseado em uso de memória externa) e o outro um microcontrolador (baseado em uso de memória interna) representa um grande desafio para portabilidade de código. Acrescente a isso a diferença de aproximadamente três décadas entre as gerações. Enquanto um foi lançado em 1982 e o outro teve sua arquitetura divulgada pela ARM em 2010. A junção desses fatores caracterizou uma ótima oportunidade para estruturar uma proposta metodológica para nortear ações semelhantes realizadas por outros desenvolvedores.

O produto em questão é a placa CPU de um controlador semafórico. Dentre suas atribuições está o controle da execução dos planos de operação de um conjunto de grupos focais (popularmente chamados de semáforos), armazenamento dos registros de falhas, comunicação com a central de monitoramento e com outros controladores. Dentre as premissas do projeto, destacamos a necessidade de que o novo *firmware* fosse 100% compatível com o anterior e que fosse capaz de incorporar novos recursos de hardware que não faziam parte do projeto original. Como exemplo, o ajuste do relógio interno via GPS.

Uma vez que esse desenvolvimento foi feito por equipe contratada não pertencente ao fabricante do controlador, foi disponibilizada à equipe um controlador semafórico completo contendo tanto a CPU original (baseada no processador 80188), como algumas placas CPUs com o novo processador (ARM) originalmente desenvolvida para uma versão diferente de controlador semafórico, mas com retro compatibilidade de pinagem com o modelo original.

A Figura 1 apresenta um recorte comparativo entre as duas versões de placa utilizadas durante a primeira fase do desenvolvimento. Observa-se no item (a) a placa original e no item (b) a placa usada para desenvolvimento da migração de *firmware* do produto. Em ambas é dado destaque apenas aos circuitos integrados (CIs) principais.

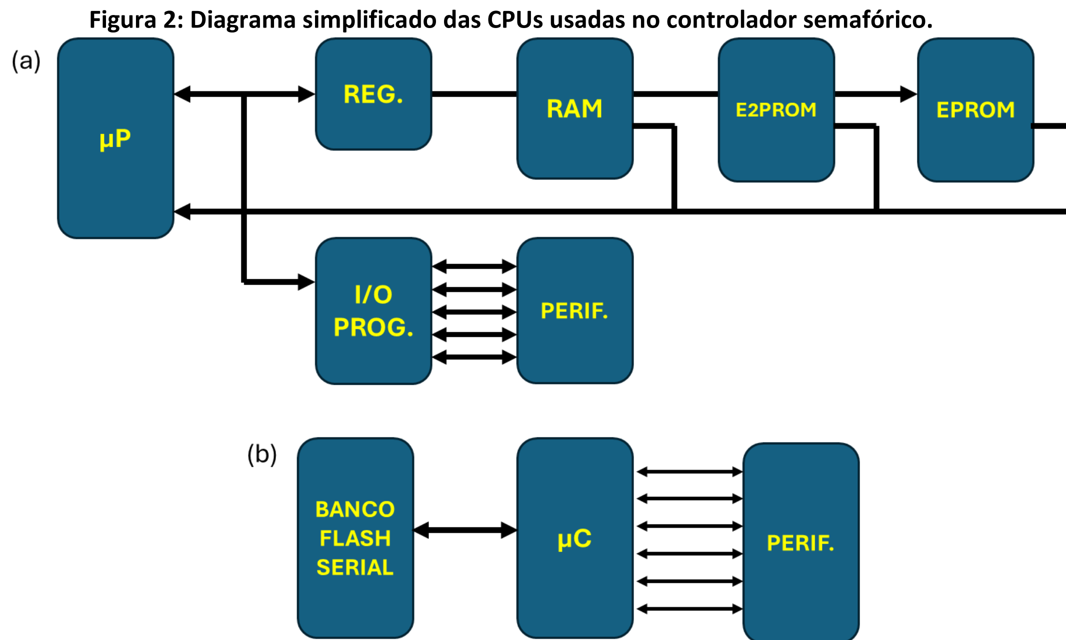
Figura 1: Modelos de CPUs usadas no controlador semafórico.



Fonte: Autor.

A Figura 2 apresenta um diagrama simplificado, que destaca as principais diferenças de hardware e arquitetura dos sistemas. Observa-se no item (a) a arquitetura da placa original e no item (b) a nova arquitetura utilizada.

Figura 2: Diagrama simplificado das CPUs usadas no controlador semafórico.



Fonte: Autor.

Como é possível observar na Figura 1 e na Figura 2, as duas placas apresentam arquiteturas extremamente distintas. Enquanto a primeira é baseada em microprocessador (μP) e tem todo o conjunto de memórias de dados e programas externas, a segunda é baseada em microcontrolador (μC) e possui apenas um banco de memórias não voláteis do tipo flash serial que passam a substituir a memória E2PROM paralela usada para armazenamento das programações semafóricas no projeto original.

Apesar de serem memórias não-voláteis, a forma como são manipuladas é totalmente diferente. Não citamos isso apenas por conta da variação de barramentos, mas principalmente pela forma como os dados são atualizados. Enquanto a E2PROM pode ser atualizada byte a byte, a memória flash deve ser manipulada setor a setor, que neste caso são de 4 kB. Isso quer dizer que, se apenas um byte de um setor precisar ser atualizado, exemplo alteração do tempo em cor vermelha de um grupo focal, todo o setor que ele se encontra precisa ser apagado e reescrito.

Esse exemplo é uma das ações que demonstram que a portabilidade de código não se resume a ajuste em registradores, variáveis e pinos de entrada/saída de dados. Existem muitas especificidades não só internamente aos processadores, mas em todos os periféricos externos envolvidos.

Muitas dessas especificidades são discutidas na próxima seção. Onde o método descrito a seguir tem sua aplicação exemplificada.

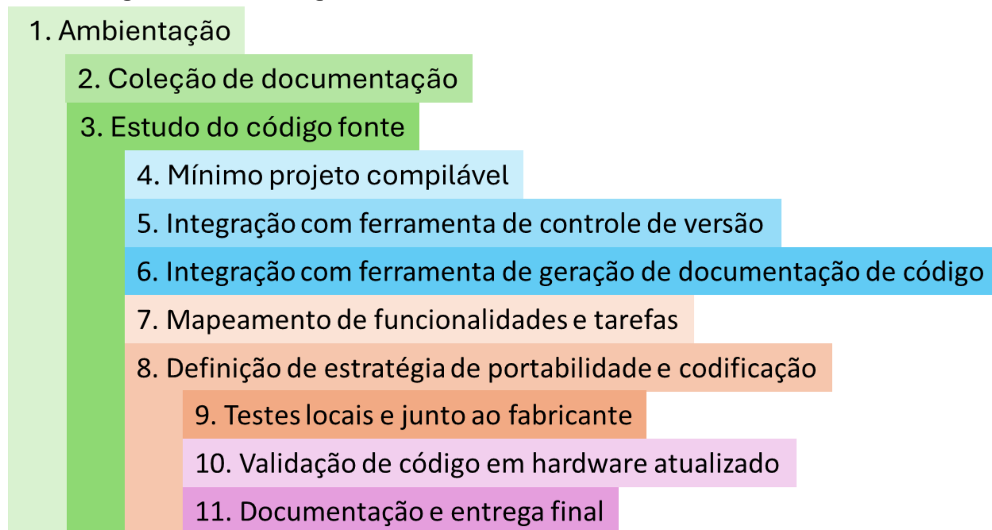
3.2 Método FPFT

Trabalhar sobre código legado é sempre um desafio. Sem nos aprofundarmos demais sobre essa questão, existem duas justificativas básicas para isso: estilos de programação próprios e familiaridade com a solução a ser modificada. Os programadores, muitas vezes, preferem começar algo do zero ao ter que modificar algo feito por outra pessoa ou equipe. Nestes casos, o amadurecimento na análise do código é fundamental.

O projeto apresentado na subseção anterior, nos fez pensar muito a esse respeito e a estruturar uma proposta metodológica de abordagem que atenda tanto a migração de arquitetura de processador, como alterações de hardware e *firmware* visando a portabilidade de código legado.

A Figura 3 representa o encadeamento de etapas da metodologia proposta, para realização de portabilidade de *firmware* e validação de suas funcionalidades orientada a sistemas embarcados. Como descrito no início desta seção, essa metodologia foi batizada *Firmware Portability and Functional Validation* e é referenciada pela sigla FPFV.

Figura 3: Metodologia FPFV.



Fonte: Autor.

A descrição de cada uma das etapas da metodologia FPFV, bem como a justificativa de sua importância é apresentada a seguir. Na seção seguinte, relacionamos cada etapa com o projeto base deste estudo.

1 - Ambientação

Qualquer projeto que apresente algum grau de complexidade, demanda ações que precedem qualquer interação com o código. A primeira delas é a ambientação com o tipo de equipamento/produto que está sendo manipulado. Nesta etapa, destacam-se ações como:

- Estudo do princípio de funcionamento;
- Identificação de especificidades do equipamento;
- Familiarização com nomenclaturas, termos e jargões usados.

Essa etapa, do ponto de vista do autor, é fundamental e tem duração indefinida. Depende muito do projeto, da experiência prévia do programador e complexidade da operação. Por conta disso, é fundamental que existam profissionais experientes e acessíveis com domínio sobre o equipamento, que possam servir como fonte de consulta rápida e contínua.

A etapa de ambientação é contínua e se sobrepõe as próximas etapas do projeto. Ela se refletirá no aprimoramento da equipe e ganho de experiência. Nela, o foco está em “o que é feito”, deixando o “como é feito” para etapa de estudo de código fonte.

Restringir o aprendizado sobre um equipamento/produto apenas pela análise do código fonte pode gerar mal-entendidos e suposições falsas. Comece pela operação, entenda o processo e particularidades. Isso tornará o estudo do código mais fácil.

2 - Coleção de documentação

A documentação técnica do produto envolve diagramas elétricos e mecânicos, manual de instalação e operação, esquemas elétricos e leiaute das placas com o devido versionamento, listas de materiais, aplicativos usados e código fonte dos *firmwares* gravados nos processadores.

A qualidade da documentação implica diretamente no tempo de desenvolvimento das tarefas de manutenção, modernização e *retrofit*.

Pode parecer obvio, mas em muitos casos, o próprio fabricante não tem domínio sobre a documentação técnica de seus produtos.

3 - Estudo do código fonte

Uma das etapas mais importantes e que se repete durante todo o projeto.

Quanto mais extenso o código, mais lentamente ocorre o estudo do código fonte. Como boa prática ele deve ocorrer, quando possível, em camadas, partindo do mais geral em direção ao mais específico. Parta da inicialização do sistema em direção a cada módulo funcional. Esse avanço pode ocorrer na mesma medida do avanço da portabilidade e cabe ao programador decidir o quanto deve se aprofundar. Normalmente nos aprofundamos naquilo que é específico do projeto e somos mais superficiais naquilo que é mais genérico, como drivers de periféricos e sistemas operacionais.

Nessa etapa, o tipo e quantidade de periféricos é mapeada e permite estimar os recursos necessários e que serão utilizados do novo processador. Além disso, permite prever alterações de topologias e acréscimo de hardware especializado em alguns casos. Isso é bastante comum quando precisamos atualizar interfaces de usuário como displays e protocolos de comunicação.

Antes de seguirmos, vale um esclarecimento quantitativo sobre código extenso. A extensão de um código pode ser medida de diferentes formas: número de bytes da memória de programa consumidos, número de linhas de código fonte, número de módulos (bibliotecas) do projeto ou qualquer outra forma que relacione essas informações. Está métrica não trata de dimensionar facilidade ou dificuldade em compreensão, pois isso está fortemente relacionado à experiência do programador.

Sendo assim, na Tabela 1 é apresentada uma proposta de classificação do nível de extensão de código embarcado baseada nessas características.

Tabela 1: Classificação quantitativa de código aplicada a sistema embarcados.

Extensão	Memória consumida (kB)	Quantidade de linhas	Número de módulos
baixa	≤ 10	≤ 1.000	≤ 5
média	≤ 50	≤ 10.000	≤ 20
alta	> 50	> 10.000	> 20

Fonte: Autor.

A Tabela 1 tem como objetivo servir como referência aos desenvolvedores de sistemas embarcados, principalmente aqueles que trabalham com código C/C++, onde não existem sistemas operacionais (normalmente Linux) e execução de código em RAM.

Uma ferramenta útil para análise de seu código é o C and C++ Code Counter (CCCC) (Littlefair, 2006). Essa ferramenta de uso livre, fruto de uma pesquisa de mestrado/doutorado de seu criador, permite extrair informações estáticas sobre os arquivos de um projeto. Com ela podemos identificar a quantidade efetiva de linhas de código e comentários por arquivo ou conjunto de arquivos entre outras informações.

4 - Criação do novo projeto e ajustes mínimos para compilação

Está etapa visa permitir ao programador ter um ponto de referência quanto a portabilidade do código.

Podemos chamar essa primeira versão compilável de projeto mínimo viável e compilável (MVCP, *minimum viable and compilable project*).

É importante resaltar que o MVCP não tem qualquer pretensão de ser funcional, apenas tem por objetivo ser compilável no ambiente de destino da portabilidade. Para isso, é desejável que todos os módulos originais façam parte do novo projeto, mesmo que parte do código seja momentaneamente comentada e/ou desativada.

Ter um código compilável, mesmo que não-funcional, é importante porque garante que os modelos de dados/variáveis, estruturas de codificação da linguagem e vínculo entre módulos são confiáveis e seguem os padrões da linguagem adotada. Isso facilita a continuidade da etapa anterior (o estudo do código fonte) e testes pontuais.

Durante essa etapa, o programador passa a enxergar mais claramente aquilo que será legado com maior ou menor dificuldade. Detecta como as variáveis foram declaradas e a forma como os módulos foram separados. Normalmente todo sistema de gerenciamento de interrupções precisa ser desativado por total incompatibilidade entre as famílias de processadores. Outro gargalo é a detecção de código Assembly incorporado às bibliotecas C/C++. Como sabido, cada arquitetura tem seu próprio conjunto de instruções e em caso de migração para uma arquitetura diferente, nada pode ser aproveitado. Além disso, o código precisará ser interpretado para compatibilização com a nova arquitetura.

5 - Integração com ferramenta de controle de versão

Uma vez de posse de um projeto compilável, é recomendável que se passe a controlar as modificações no código através de ferramentas especializadas no controle de versões. Dentre elas, temos o popular Git (Git, 2024), o Mercurial (Mercurial, 2024) e o antigo Subversion (SVN) (Apache Software Foundation, 2024). Também é importante que seu código fonte seja preservado em uma plataforma de hospedagem. Isso garante mais flexibilidade e segurança no acesso, além disso é uma proteção contra perdas. As ferramentas mais populares atualmente são o GitHub (GitHub, 2024), Bitbucket (Atlassian, 2024) e GitLab (GitLab B.V., 2024).

6 - Integração com ferramenta de geração de documentação de código

As ferramentas de geração de documentação de código são programas que automatizam a criação de documentação a partir do código-fonte. Elas podem ser usadas para gerar documentação em diversos formatos, como HTML, PDF, RTF e ePub. Essas ferramentas ajudam a economizar tempo e esforço na criação de documentação. Além disso, elas podem garantir que a documentação esteja sempre consistente e atualizada com o código-fonte.

Existem diversas ferramentas de geração de documentação de código disponíveis, cada uma com suas características e vantagens. Algumas das ferramentas mais populares são Doxygen (Heesch, 2023), Sphinx (Turner *et al.*, 2024) e Read the Docs (Holscher *et al.*, 2022).

7 - Mapeamento de funcionalidades e tarefas

Uma vez atingido o modelo mínimo compilável, a próxima etapa é mapear todas as funcionalidades e tarefas executadas pelo código. Essa tarefa demanda leitura um pouco mais aprofundada do código de forma a entender o papel de cada biblioteca, periférico usado e dinâmica de execução do código.

Nesta etapa, cabe ao programador criar suas próprias anotações, fluxogramas e diagramas de forma a permitir a compreensão estrutural da dinâmica de execução do programa.

Durante o mapeamento, o programador pode classificar o nível complexidade na execução da portabilidade de cada funcionalidade e tarefa. Essa informação é importante para definição da próxima etapa.

8 - Definição de estratégia de portabilidade e codificação

Uma vez que as funcionalidades foram identificadas, é necessário definir a estratégia para realização da portabilidade do código de forma fracionada. Isso permite simplificar ações e trabalhar com testes pontuais.

Uma ação bastante comum é começar pela compatibilização de GPIOs e periféricos de comunicação. Uma boa prática é criar uma biblioteca de depuração, onde funções auxiliares permitem testar, de forma desvinculada à logica de funcionamento do equipamento, todos os GPIOs e periféricos.

Após isso, comece pelo mais importante, ou seja, aquilo que representa a funcionalidade principal do equipamento.

9 - Testes locais e junto ao fabricante

Esse passo da metodologia é válido apenas quando o desenvolvimento é feito por equipe externa a empresa detentora dos direitos sobre o produto desenvolvido. Possuir um ambiente de teste confiável, não substitui a necessidade de realização de testes junto ao fabricante/contratante do serviço.

Normalmente esses testes são feitos em laboratório técnico com ambiente preparado para simulação de condições semelhantes às encontradas em campo. Mas, em muitos casos, pode ser uma exigência do contratante que ocorra o acompanhamento do desenvolvedor para realização de testes em campo, a fim de confirmar o desempenho do produto.

10 - Validação de versão em nova placa dedicada ao projeto

É natural que de tempos em tempos novas versões de hardware, software e *firmware* sejam desenvolvidas e lançadas para produção.

Em um projeto de portabilidade de *firmware*, é natural que haja paralelamente o desenvolvimento de uma nova placa para comportar o processador e periféricos novos. Em muitos casos, ela será fundamental para início dos testes de *firmware*.

Dessa forma, esse passo da metodologia proposta, pode vir a ser executado em um momento anterior no desenvolvimento. Optamos por deixá-lo no final, por ser o momento em que ocorrem os ajustes finais de projeto.

11 - Documentação e entrega final

Esta etapa consiste na coleção de todos os documentos de projeto como manuais de operação e código, arquivos de leiaute e versão final do *firmware*.

Toda documentação colecionada deve ser revisada e entregue formalmente ao contratante do projeto. Está etapa marca o fim do projeto e é fundamental para que não haja mal-entendido quanto a continuidade do desenvolvimento e verificação de possíveis *bugs* tardios.

4 APICAÇÃO DA METODOLOGIA E DISCUSSÕES

Esta seção é dividida em duas subseções. Na primeira é descrito como a metodologia FPFV foi aplicado ao projeto de portabilidade do *firmware* da CPU de um controlador semafórico. Além disso, são destacados os desafios encontrados, que naturalmente se transformaram em uma coleção de dicas e boas práticas de projetos. A segunda subseção traz um resumo dos resultados do projeto e aprofunda a descrição de alguns dos principais desafios encontrados e as soluções aplicadas a cada um deles. Complementarmente, apresenta considerações sobre a metodologia proposta e suas limitações.

4.1 Aplicação da metodologia FPFV

Os itens a seguir exemplificam como a metodologia FPFV foi aplicada no desenvolvimento da portabilidade e migração de processador em uma placa controladora semafórica.

4.1.1 - Ambientação

É evidente que todos os leitores deste artigo sabem o que é um semáforo de trânsito. Porém, assim como poderia ocorrer com outros equipamentos, iria se surpreender com suas especificidades na dinâmica de funcionamento, regras de segurança e programação, considerações sobre horários e sincronismo com funcionamento de outros grupos semafóricos etc. Entende-se como grupo semafórico um conjunto de grupos focais, popularmente chamados de semáforos, que estão ligados em um mesmo circuito elétrico (CET Companha de Engenharia de Tráfego, 2018).

Antes e durante todo o desenvolvimento e portabilidade do *firmware* da controladora semafórica, vários contatos com o fabricante foram feitos usando diferentes canais de comunicação (escrita e voz). Por vezes, uma dúvida conceitual, que poderia impactar o prosseguimento do trabalho, foi esclarecida em minutos. Isso foi fundamental para seu sucesso.

Dicas:

- Nunca subestime a complexidade de um projeto.
- A ambientação é um processo contínuo e se estende por todo desenvolvimento do projeto.
- Tenha sempre um contato de alguém mais experiente que você e acessível.

4.1.2 - Coleção de documentação

Ter um pacote de documentação que reflita diferentes aspectos funcionais do produto em desenvolvimento é fundamental e contribui para a ambientação.

Neste projeto, a documentação disponível refletia quase perfeitamente a realidade.

Sendo que, a última versão do *firmware* aplicada no produto serviu com pedra fundamental. A partir dela, todo o trabalho foi desenvolvido. Paralelamente ao *firmware*, os esquemas elétricos da placa original e da placa de destino foram cruciais para criação de um modelo de transferência das conexões elétricas e pinagem. Comumente chamamos o modelo de transferência de "tabela de-para". Por meio dela é explicitado parte do trabalho a ser feito na rotina de inicialização do *firmware*.

Dicas:

- Cheque se a documentação disponível realmente corresponde ao produto desenvolvido.
- Tenha em mente que mesmo a documentação oficial tem erros. Não confie 100%.
- Em caso de dúvida, procure um responsável pelo produto.
- Se achar um erro, comunique ao responsável.
- Seja ético e jamais use documentos oficiais para fins diferentes do acordado.

4.1.3 - Estudo do código fonte

Por se tratar de um código classificado como extenso, de acordo com o proposto na Tabela 1, optou-se por um aprofundamento gradual nos meandros do código à medida que o projeto avançava.

No primeiro momento, observou-se a dinâmica do código de maneira mais superficial. O foco foi saber como as tarefas da placa controladora eram disparadas e gerenciadas. Logo foi observado a presença de um sistema operacional de tempo real (RTOS) customizado para o controle de diferentes tarefas. A utilização de funções nativas do antigo sistema operacional DOS também foram observadas. Esses dois pontos somados a diferença total na pinagem do processador foram os primeiros pontos de atenção que precisariam ser tratados.

Assim como a etapa de ambientação do produto, o estudo do código fonte é contínuo. Não tente absorver tudo de uma vez, isso causa mais stress do que segurança.

Dicas:

- Sempre faça anotações. Não confie em sua memória.
- Estruturar o código na forma de diagramas ajuda na compreensão de sua macro dinâmica.
- Tenha em mente que o código pode ter erros. Não confie 100%.
- Se não entender algo, é sinal que ainda não está totalmente ambientado com o produto. Espere um pouco mais.
- Comentários falsos ou desatualizados fazem parte do desenvolvimento de software/*firmware*. Não confie 100%.
- Procure sempre aprender observando diferentes estilos de programação. Essa é uma das coisas mais enriquecedoras.

4.1.4 - Criação do novo projeto e ajustes mínimos para compilação

Para isso, foi criado um projeto para microcontrolador ARM com a importação dos arquivos originais do controlador baseado no microprocessador Intel 80188.

Essa etapa foi muito delicada e demorada, pois diversos arquivos precisaram ser modificados ou ter seu código parcialmente comentado a fim de evitar erros naturais decorrentes da mudança de arquitetura de hardware e processador.

O foco principal foi simplesmente conseguir uma versão compilável sem preocupação

com a dinâmica de funcionamento da controladora.

Segue uma lista parcial com os principais obstáculos enfrentados:

- Funções da biblioteca nativa DOS tiveram que ser reescritas ou substituídas.
- Funções que executavam funções nativas do DOS que tiveram que ser alteradas.
- Trechos de códigos em Assembly que não apresentam nenhuma compatibilidade com o novo processador foram temporariamente comentados.
- Acessos diretos aos registradores internos da CPU original foram temporariamente comentados.
- Inicialização dos periféricos precisou ser totalmente comentada.
- Ajuste no sistema de interrupções e rotinas de tratamento (ISRs). Toda parte de configuração de interrupções foi comentada. Apenas as funções de tratamento foram mantidas.
- Desativação do modelo de *watchdog timer* original.
- Diretivas e modificadores de variáveis e funções não compatíveis com o compilador ARM foram comentados.
- Integração de RTOS comercial ao projeto. Neste caso, optamos pelo FreeRTOS (Amazon Web Services, 2022).

Como dito, na descrição dessa etapa, ela não tem objetivo gerar uma compilação para gravação, ela apenas visa garantir um modelo mínimo compilável que possa permitir estimar a fração de código que pode ser reaproveitado sem modificações significativas.

Apesar de parecer radical, essa prática permite a criação de uma lista de pontos a serem atacados. A partir dela uma estratégia de priorização pode ser definida.

4.1.5 - Integração com ferramenta de controle de versão

Neste projeto foi utilizada a ferramenta de controle de versões Git em conjunto com o GitHub. A importância do GitHub foi de garantir o backup do projeto fora de qualquer computador utilizado no projeto. Permitindo assim a recuperação do código a qualquer momento e em qualquer lugar.

Dicas:

- Garanta que toda a equipe esteja familiarizada com as ferramentas escolhidas.
- Gere pelo menos uma versão por dia de trabalho.
- Muitos ambientes de desenvolvimento (IDEs) atuais possuem integração direta com ferramentas de versionamento. Isso facilita o trabalho.

4.1.6 - Integração com ferramenta de geração de documentação de código

Paralelamente a etapa anterior, foi criado um arquivo de integração do Doxigen com o código fonte. Essa ferramenta é capaz de ler todo código selecionado e gerar excelente documentação de todos os arquivos, variáveis e funções. Além de identificar vínculos. Para tirar maior proveito desta ferramenta, toda função alterada passou a receber comentários compatíveis com o modelo Doxigen. Assim, ao ler o código a ferramenta consegue documentar com qualidade e detalhes.

Neste projeto, a documentação gerada em formato RTF foi de suma importância para complementar a documentação e manual de software entregue a presença contratante do serviço. Isso nos poupou muitas horas de trabalho apenas com um clique.

A documentação de código está fortemente ligada a inserção de comentários em código e como sugerido no item 3, comentários falsos ou desatualizados fazem parte do jogo. Para minimizar isso, faça apenas comentários considerados importantes e relevantes.

Por outro lado, em uma missão como a de estruturar e realizar a portabilidade de código legado, o uso de comentários favorece o desenvolvedor. Aqui caímos em uma condição paradoxal.

Dicas:

- Priorize sempre o seu entendimento como desenvolvedor. Comente o código quando achar necessário. Modifique sempre que precisar. Apague os comentários quando puder.
- Bibliotecas básicas de periféricos não necessitam de detalhamento, crie apenas os comentários necessários para que elas sejam reconhecidas corretamente pela ferramenta de geração de documentação.
- Código legado pode ser mal escrito. Documente para facilitar sua vida.

4.1.7 - Mapeamento de funcionalidades e tarefas

Uma vez que o projeto em questão faz uso de um RTOS customizado, a identificação das tarefas executadas está diretamente associada às tarefas disparadas pelo escalonador e baseadas em interrupções de programa, principalmente disparadas por temporizadores. Além dessas, destacamos as ações decorrentes aos planos de operação dos grupos semafóricos, que são definidos pelos técnicos de trânsito em campo. Isso demandou um aprofundamento em como operar o controlador semafórico em condições de trânsito reais. Como citado no passo 1 da metodologia, a ambientação é contínua e ocorre ao longo de todo tempo do projeto. Essa *expertise* foi fundamental para programação operacional do controlador e criação de diferentes cenários para testar o avanço no desenvolvimento do *firmware*.

Dicas:

- Separe o *firmware* em tarefas para facilitar sua compreensão. Mesmo que ele não seja estruturado formalmente como tarefas, pode ser separado desta forma.
- Essa separação facilita o próximo passo: definição de estratégia de portabilidade. Seja cartesiano, faça uma coisa de cada vez.

4.1.8 - Definição de estratégia de portabilidade e codificação

A sequência de etapas a seguir é uma simplificação do modelo adotado. Uma vez que o projeto usado como referência para demonstração da metodologia proposta demandou um período longo de desenvolvimento e tem natureza naturalmente complexa.

- Integração das bibliotecas de abstração do hardware do processador ARM. Essas bibliotecas foram escritas de forma a separar totalmente o acesso ao hardware da dinâmica do programa. Essa implementação teve por objetivo facilitar a portabilidade futura deste código para outro dispositivo;
- Criação de funções de depuração baseadas em interface serial. Uma porta serial do tipo UART foi dedicada à depuração, assim a qualquer momento mensagens poderiam ser visualizadas em terminal serial. Isso facilitou o entendimento da dinâmica de execução do código e detecção de defeitos;
- Reescrita das rotinas de inicialização de periféricos (GPIOS, portas seriais, *timers* etc.). Neste momento a "tabela de para" criada durante a etapa 2 foi de

suma importância;
- Criação de rotinas de teste para todos os periféricos;
- Teste de todos os periféricos independentemente da dinâmica do produto;
- Migração do RTOS customizado para o FreeRTOS;
- Teste do FreeRTOS através de tarefas de depuração;
- Teste de escalonamento das tarefas nativas através do FreeRTOS;
- Habilitação da interface com o programador manual;
- Substituição de periféricos (modelos de RTC - externo para interno ao uC, ajuste do relógio - rede elétrica para GPS, salvamento de programação - E2PROM paralela externa para flash serial externa, registro de log de erros - E2PROM externa para E2PROM interna etc.);
- Testes de programação de diferentes formas de operação dos conjuntos semafóricos;
- Comunicação com a central de controle via RS-485.

É importante resaltar que o desenvolvimento dessas atividades não ocorreu necessariamente em ordem cronológica. Em vários momentos precisamos recuar e avançar. Ao trabalhar em um programa com tamanha complexidade, é humanamente impossível ter controle total sobre tudo que é processado e como é processado. Dessa forma, o amadurecimento do entendimento da dinâmica do código e metodologia adotada no seu desenvolvimento ocorreu gradualmente durante os meses de trabalho.

Vale resaltar, que o código original recebido para portabilidade foi desenvolvido ao longo de um longo período, provavelmente por diferentes programadores. Do ponto de vista do autor deste trabalho, em nenhum momento os quatro pilares citados para qualidade software defendidos por Software Reliability Enhancement Center; Technology Headquarters; Information-technology Promotion Agency (2018), (confiabilidade, manutenibilidade, portabilidade e eficiência) foram importantes ou levados em conta. Isso dificultou de maneira exacerbada a interpretação do código e a execução das alterações necessárias. Como o foco deste projeto era apenas realizar a portabilidade do código original e criar uma solução equivalente a pré-existente, somente reescrevemos código quando necessário e passamos a documentar tudo aquilo que foi alterado e/ou compreendido a fim de facilitar exclusivamente essa tarefa.

É importante resaltar que, essa tarefa somente foi possível uma vez que, o código fonte original disponibilizado estava totalmente escrito em linguagem C e seria portado usando a mesma linguagem. Os poucos trechos de código escrito em linguagem Assembly x86 eram exclusivos de uso do RTOS customizado nativo ou de acesso às memórias externas e ao sistema de interrupções. Todos esses recursos foram substituídos por aqueles disponíveis na arquitetura ARM.

Outro ponto importante, que destacamos como uma boa prática, é a manutenção de todo código original na forma de comentário, junto ao código que o substituiu. Isso nos permitiu observar, a qualquer tempo, como era antes da alteração proposta e rever nossa interpretação de funcionamento de determinadas rotinas.

4.1.9 - Testes locais e junto ao fabricante

Uma vez que o projeto exemplificado foi feito por empresa de desenvolvimento independente do fabricante, os testes preliminares foram feitos utilizando um controlador completo cedido pelo fabricante. Já os testes no fabricante, foram executados por equipe

competente e especializada. Os testes foram repetidos a cada entrega de versões parciais e da versão completa.

A partir do momento que se iniciou a integração das rotinas operacionais da controladora, foi de suma importância ter um controlador completo com placas sobressalentes para testes e gigas desenvolvidas especialmente para isso.

Dicas:

- Tenha boas ferramentas de gravação, depuração e análise. Pois a tarefa de depuração é enfadonha por natureza. Não a torne mais difícil.
- Tenha em mente que defeitos podem e vão ocorrer a qualquer momento. Na maior parte das vezes, são decorrentes do código escrito. Porém, não raramente, podem estar no hardware. Outras vezes menos comuns, são decorrentes de falhas de projeto, que podem afetar tanto o hardware como o *firmware* em condições específicas de operação.
- Valide o desenvolvimento com o fabricante sempre que possível. Isso ajuda observar condições que por vezes não foram previstas.

O laboratório utilizado durante este desenvolvimento dispunha desde recursos básicos como como osciloscópios, fontes ajustáveis digitais e multímetros. Mas também micro ohmímetros, terminais seriais programáveis e analisadores de protocolos. Fato é que, todos foram utilizados em diferentes momentos e foram cruciais para o sucesso do projeto.

Como destacado na segunda dica deste item, podem existir erros no projeto original. Durante esse desenvolvimento nos deparamos por mais de uma vez com erros de projeto que perpetuaram por mais de 20 anos. Esses erros estavam presentes no hardware e no *firmware* e demandaram retrabalhos que não contávamos, mas eram fundamentais para o sucesso da tarefa.

Os testes feitos pelo fabricante começaram apenas após um ano de trabalho, quando o código foi completamente portado para arquitetura ARM.

Somente após validado pelo fabricante, começou-se uma próxima etapa no *firmware*, que foi a remoção do código original comentado e otimizações que se mostraram pertinentes. Nesse momento também, os comentários de parte do código foram melhorados e foi feita a exclusão de código criado exclusivamente para depuração.

4.1.10 - Validação de versão em nova placa dedicada ao projeto

Apesar de, desde o início termos feito uso de uma placa utilizada em produto similar com retro compatibilidade com a CPU deste projeto. Foi uma condição estabelecida em seu início que um novo leiaute de placa seria desenvolvido paralelamente ao *firmware*, trazendo todos os ajustes necessários e novas features que seriam implementadas em um segundo momento.

A validação do novo *firmware* nessa placa serviu de consagração do sucesso na portabilidade do código e suas novas features.

Por se tratar de um novo leiaute e com novos periféricos implementados, a depuração precisou ser feita de forma mais criteriosa. Uma vez que os problemas poderiam estar tanto no hardware, como do *firmware*. Para isso, separamos a validação em duas etapas: a validação de periféricos e a validação funcional.

A validação de periféricos consiste em testar as funcionalidades de cada periférico independentemente do seu uso na aplicação. Cada GPIO, porta serial, memória etc. foi configurado de forma semelhante ao uso pretendido e testado através de rotinas exclusivas para tal.

A validação funcional é basicamente aplicar o *firmware* no produto e testar o mesmo em ambiente similar ao campo. Onde diferentes condições são geradas e observadas as respostas. Esse foi o mesmo processo feito para validação do *firmware* na versão anterior da placa.

4.1.11 - Documentação e entrega final

Destacamos o uso da ferramenta Doxygen que, como citado no começo deste artigo, é responsável por geração automática de documentação do código, uma vez que comentários adequados sejam incluídos no mesmo. Para uma melhor qualidade na documentação, todo o código passou por revisão, onde a formatação dos comentários foi ajustada e trechos de código que ainda não haviam sido documentados foram alterados.

Além disso, nesta etapa, foi feita a checagem e revisão da inclusão de bibliotecas feitas em todos os arquivos, diretivas feitas ao pré-processador, códigos de teste ativos que precisariam ser removidos, comentários ambíguos e observações irrelevantes ou duvidosas. Complementarmente, todos os *warnings* de compilação restantes foram revisados e tratados.

A documentação pelo Doxygen foi gerada em formato HTML e RTL. Este último pode ser facilmente editado no Word com informações complementares como índice, figuras e novos comentários. O primeiro é mais prático para consultas via navegador de internet com visualização das correlações entre diferentes módulos e presença de inúmeros *hiperlinks* que facilitam a navegação.

4.2 Principais desafios e resultados

Na subseção anterior, à medida que a metodologia FPFV era exemplificada, foram citados alguns dos desafios mais significativos do projeto de migração do processador da placa controladora semafórica. Nesta subseção, são apresentados mais detalhes e estratégias usadas para resolver os grandes desafios deste projeto.

4.2.1 - Mudança radical de arquitetura

A mudança de uma arquitetura de 16 bits dos anos 1990 para uma arquitetura de 32 bits ARM moderna representa um grande salto em capacidade de processamento e traz com si muitas mudanças na arquitetura do sistema.

Na subseção 3.1 foram apresentadas as mudanças de arquitetura, do ponto de vista de processador e acesso às memórias. Na placa original, o microprocessador Intel 80188 faz uso de memórias paralelas externas tanto para armazenamento de programa, como de dados voláteis e não voláteis. A troca por um microcontrolador ARM, em parte simplificou o acesso às memórias, uma vez que o programa e dados voláteis passaram a ser armazenados em memórias internas, flash e RAM respectivamente. Por outro lado, a memória externa do tipo E2PROM paralela utilizada anteriormente precisou ser substituída por uma memória flash serial.

Essa memória é utilizada para armazenamento da programação básica do controlador semafórico e os planos de operação, que definem horários de atuação e duração da ativação de cada um dos focos de todos os grupos focais. A principal dificuldade do uso de memória flash sem um sistema de gerenciamento de arquivos, como o FAT32, é que qualquer troca de valor armazenado na memória implica em prévio apagamento de todo o setor de 4 kB onde o dado precisará ser atualizado. Essa operação é muito delicada e pode impactar em perda de programação, caso procedimento não seja executado corretamente.

Para contornar essa limitação, a estratégia utilizada foi trabalhar com todos os dados em memória RAM. No momento do *boot* do controlador, os dados armazenados em memória flash são transferidos para memória RAM e todas as funções que demandam por esses dados, fazem acesso exclusivamente via memória RAM. Após qualquer atualização de dados feita na memória RAM, é disparada uma rotina especialmente criada para atualização da memória flash. Para isso, foram utilizados dois blocos de 64 kB alternadamente para backup de todo o conteúdo das programações do controlador. Isso implica que, sempre existe um backup com a versão anterior dos dados de programação. Em caso de falha na gravação, a versão anterior é automaticamente recuperada.

4.2.2 - Código legado não direcionado à portabilidade

Quando falamos de código portável, estamos nos referindo ao grau de eficácia e eficiência com que o *firmware* de um produto pode ser transferido para outro.

Uma parte fundamental para isso é que o *firmware* seja estruturado em camadas, onde as camadas mais baixas estão intimamente relacionadas com o hardware e as camadas mais altos totalmente orientadas à aplicação.

O *firmware* legado não foi estruturado dessa forma, na verdade não foi observada qualquer estrutura em camadas. A única separação identificada entre os módulos do projeto estava relacionada a aplicação de cada um deles. Porém, muito frequentemente os módulos executavam acesso direto ao hardware a partir de registradores específicos da arquitetura antiga.

Além disso, foi observado alto grau de acoplamento entre os módulos, ou seja, um módulo faz acesso excessivo aos demais. Isso caracteriza o chamado código *spaghetti*, que dificulta muito a interpretação do código como um todo.

Apesar do objetivo deste projeto não ser realizar a reestruturação de código, optou-se por criar um conjunto com nove módulos especializados em acesso ao hardware. Assim, foi possível desvincular os módulos de aplicação ao acesso direto ao hardware, permitindo independência entre hardware e aplicação. Uma vez validado o acesso ao hardware, o foco do projeto se concentrou exclusivamente nas regras de negócio.

4.2.3 - Estruturas de dados

Um dos obstáculos mais complexos enfrentados neste projeto foi a compatibilização de todas as estruturas de dados legadas. Neste projeto, as estruturas (*struct* em linguagem C) definem toda lógica de funcionamento desejado para cada plano de trabalho e as características do conjunto de vias controladas.

Um problema comum, ao declarar variáveis do tipo inteiras em linguagem C (*int* ou *unsigned int*), é que esse não é um tipo com comprimento padronizado, é dependente do compilador. No caso da arquitetura legada, ela representava um valor de 16 bits, já na arquitetura ARM Cortex-M4 representa um valor de 32 bits.

Em um primeiro momento da portabilidade, optou-se por não alterar os tipos de variáveis. Como primeiro impacto, observou-se o aumento de memória RAM consumida para alocação dos dados. Isso não se mostrou crítico, uma vez que o processador utilizado possuí capacidade suficiente para tal.

A incompatibilidade na alteração do comprimento de variáveis se mostrou crítica quando detectamos que a verificação da integridade dos dados era feita via *checksum* e que o cálculo era baseado um incremento de ponteiro para leitura da estrutura. Esse ponteiro era incrementado um número de vezes constante, equivalente a quantidade original de bytes de

cada estrutura.

Além disso, foi observado uma série de leituras parciais das estruturas feitas a partir de ponteiros com deslocamentos relativos a números inteiros pré-definidos.

Em um primeiro momento, pensou-se que a solução para tais inconveniente estaria no ajuste dos tipos das variáveis. Isso quer dizer, passar todos as variáveis *int* e *unsigned int* para *int16_t* e *uint16_t*.

Ao fazer isso, o problema foi parcialmente resolvido, mas gerou um problema ainda maior decorrente do desalinhamento de memória. Na arquitetura ARM de 32 bits quando alocamos variáveis sequenciais os endereços de variáveis devem seguir uma regra básica:

- Variáveis de 8 bits são alocadas em endereços sequenciais;
- Variáveis de 16 bits são alocados em endereços pares;
- E variáveis de 32 bits são alocadas em endereços múltiplos de quatro.

A consequência disso é que, ao criar estruturas heterogêneas (diferentes tipos de dados), podem existir endereços nulos entre uma variável e outra.

Esse efeito colateral, impactou diretamente no tamanho das estruturas, no cálculo do *checksum* e acessos relativos usando deslocamentos calculados em bytes.

Para solucionar todos esses efeitos, algumas estratégias foram utilizadas, porém o trabalho demandou validação caso a caso. Algumas das estratégias empregadas:

- Quando possível, o acesso relativo foi substituído por acesso absoluto, o nome do argumento passou a ser utilizado em substituição ao deslocamento na estrutura;
- Algumas estruturas foram reorganizadas, onde a mudança de alguns elementos de posição reduziu os bytes nulos ou até eliminou sua ocorrência;
- O tipo de algumas variáveis foi alterado, gerando resultado semelhante ao anterior;
- Os limites para cálculo do *checksum* foi ajustado para todos os casos.

Essa incompatibilidade foi responsável por grande parte do trabalho de portabilidade e foi sendo resolvida durante todo o desenrolar do projeto.

4.2.4 - Uso excessivo de constantes numéricas e *flags*

Durante as etapas de estudo do código fonte e codificação, ficou claro aos desenvolvedores que o código legado tinha características muito claras de código desenvolvido por profissional acostumado a programar em linguagem Assembly.

Como característica marcante desse estilo de programação estão o uso excessivo de *flags* binárias para sinalização de estados, múltiplas constantes numéricas espalhadas pelo código sem definição clara da representatividade de seus valores, operações booleanas com máscaras de bits para definição de modos de operação e/ou condição do programa.

Essas características sempre dificultam a interpretação de código, obrigando o programador ir além de parâmetros objetivos e passando a trabalhar de forma subjetiva ao tentar imaginar qual a intenção do programador do código legado.

Neste projeto, na medida do possível o uso de novas *flags* foi evitado e às constantes numéricas foram substituídas por descrições com significado claro.

4.2.5 - Necessidade de mudar o RTOS

A migração de um sistema operacional de tempo real para outro é sempre delicada por conta da especificidade de cada um. Algumas dessas especificidades:

- Forma de gerir as tarefas;
- Velocidade de escalonamento;
- Algoritmo de escalonamento;
- Gestão de memória;
- Tipos de operações possíveis;
- E estados atribuídos às tarefas.

O projeto legado era totalmente baseado em um sistema customizado que fazia uso de funções do antigo sistema operacional DOS. Além disso, todo sistema de gestão de troca de tarefas era escrito em Assembly. Essas características foram mandatórias para substituição do RTOS original. A opção aplicada foi o FreeRTOS, que além de ser gratuito, tem uma comunidade muito ativa e milhares de aplicações bem-sucedidas.

Esse processo demandou a reescrita de muitas das funções originais, principalmente aquelas que eram responsáveis por interromper execução de tarefas, criar *sleep* e reiniciar tarefas. Além disso, o FreeRTOS precisou ser parametrizado de forma a executar comportamento semelhante ao legado. Assim como o ajuste nas estruturas de dados, esses ajustes ocorreram durante boa parte do processo de codificação.

4.2.6 - Código mal comentado ou não comentado

Todo o código legado apresentava pouquíssimos comentários. Não havia sequer referência ao propósito de cada módulo. Os poucos comentários eram extremamente pontuais e frequentemente estavam desatualizados.

A estratégia adotada para mitigar essa dificuldade foi comentar cada novo entendimento e documentar corretamente todos os novos desenvolvimentos, quando possível seguindo o padrão Doxigen, já citado.

Quando lidamos com códigos muito extensos e mal estruturados, a falta de comentários precisos e claros é um grande dificultador e impacta diretamente no tempo para conclusão do projeto. Por outro lado, os diagramas elétricos e todos os manuais de operação serviram como ferramenta de apoio para o preenchimento das lacunas observadas durante o estudo e o desenvolvimento da portabilidade do código.

Para finalizar esta subseção, é apresentada a Tabela 2 com um resumo das características quantitativas e qualitativas do código legado e código resultante da portabilidade.

Tabela 2: Comparativo entre código legado e portado da CPU do controlador semafórico.

Características	Código	
	Legado	Portado
Processador	Intel 80188	ARM Cortex-M4
Clock (MHz)	12,288	80
Programa armazenado	memória EPROM externa	memória flash interna
Programa compilado (kB)	~222	~140
Consumo de RAM (kB)	-	~120
Quantidade de linhas	~14.000	~27.000 (sem o FreeRTOS)
Número de módulos	26	39 (sem o FreeRTOS)
RTOS	customizado	FreeRTOS
Sincronização relógio	rede elétrica/cristal	GPS/cristal
Programação semafórica	memória E2PROM paralela externa	memória flash serial externa
Estruturação de código	pouco ou nenhuma	separação em duas camadas (acesso ao hardware e aplicação)
Comentários/documentação	poucos e/ou desatualizados	padrão Doxigen, aplicado a todos os módulos de código modificados ou criados

Fonte: Autor.

Um ponto interessante a ser notado nos dados sintetizados na Tabela 3 é que, apesar do aumento do número de módulos e linhas de código C totais, o código portado consome apenas 63% de memória de programa em relação ao código legado. Outro ponto importante é o incremento no clock do processador, passamos de pouco mais de 12 MHz para 80 MHz com ciclo de máquina de 1 *clock* por instrução.

5 CONCLUSÃO

Neste artigo, se propôs uma metodologia para migração de processadores e portabilidade de código com foco em sistemas embarcados.

A metodologia proposta, identificada pela sigla FPFV, abreviação de *Firmware Portability and Functional Validation*, é resultado da experiência prática do autor deste artigo e vai ao encontro da realidade enfrentada por outros profissionais desenvolvedores de *firmware*, que diariamente se deparam com código legado e diferentes arquiteturas de processadores (microcontroladores e microprocessadores).

Os onze passos da metodologia foram descritos e sua aplicação foi exemplificada através de um estudo de caso real, onde uma placa CPU de um controlador semafórico desenvolvida nos anos 1990 precisou ser atualizada passando de arquitetura de microprocessador Intel 80188 para a arquitetura de microcontrolador ARM Cortex-M4 contemporânea. A justificativa para tal ação não se encontrava puramente na obsolescência da arquitetura original, mas foi fortemente demandada pela crise dos semicondutores decorrente epidemia de covid-19 e falta de chuvas em Taiwan no mesmo período, isso gerou escassez e interrupção na produção de circuitos integrados, principalmente aqueles que não faziam parte da categoria *high-end*, direcionada à fabricação de celulares e computadores.

Durante a exemplificação da metodologia, dicas foram sugeridas aos desenvolvedores, algumas vezes quanto a ferramentas e estratégias, outras quanto a o que esperar de um projeto e sua documentação. Assim, acredita-se ter criado uma boa referência para desenvolvimento de projetos com a mesma natureza, mas entendemos também que aqui se encontra uma contribuição que abre caminhos para discussões e complementações que visam auxiliar toda comunidade envolvida. De forma alguma, o método proposto é estático ou definitivo, é apenas uma primeira proposta de arcabouço. É evidente que a contribuição de outros profissionais é bem-vinda e será de grande valia para toda a comunidade de desenvolvedores.

Por fim, ressaltamos que durante essa pesquisa ficou claro que a velocidade com que as ferramentas de software direcionadas aos aplicativos de computadores e celulares se desenvolvem é muito superior que aquelas direcionadas aos sistemas embarcados. Muitos dos recursos atuais para análise de código ou documentação não atendem as necessidades dos desenvolvimentos que ocorrem com relação mais intima com o hardware. Por isso, identificamos uma oportunidade de trabalhos futuros em ferramentas de apoio ao projeto de sistemas embarcados, tanto em seu desenvolvimento original, como a partir de código legado. Paradoxalmente, isso demanda a contribuição indispensável de desenvolvedores de sistemas computacionais.

REFERÊNCIAS

AMAZON WEB SERVICES, INC. **FreeRTOS**, 2022. Disponível em: https://www.freertos.org/ . Acesso em: 28 mar. 2024

APACHE SOFTWARE FOUNDATION. **Apache Subversion**, 2024. Disponível em: https://subversion.apache.org/ . Acesso em: 27 mar. 2024

ATLASSIAN. **Bitbucket**. Disponível em: https://bitbucket.org/ . Acesso em: 27 mar. 2024.

BARR, M. **Embedded C coding standard**. BARR-C:2018 ed. [s.l.] Barr Group, 2018.

CET COMPANHA DE ENGENHARIA DE TRÁFEGO. **Controlador de tempo fixo: sistema INFLUUNT**. São Paulo: [s.n.]. Disponível em: https://www.cetsp.com.br/consultas/publicacoes/especificacoes-tecnicas/sistema-de-controle-semaforico.aspx . Acesso em: 14 mar. 2024.

GIT. 2024. Disponível em: https://git-scm.com/ . Acesso em: 27 mar. 2024

GITHUB, INC. **GitHub**. Disponível em: https://github.com/ . Acesso em: 27 mar. 2024.

GITLAB B.V. **GitLab**. Disponível em: https://about.gitlab.com/ . Acesso em: 27 mar. 2024.

HEESCH, D. VAN. **Doxigen**, 25 dez. 2023. Disponível em: https://www.doxygen.nl/ . Acesso em: 28 mar. 2024

HOLSCHER, E. *et al.* **Read the Docs**, 2022. Disponível em: https://about.readthedocs.com/ . Acesso em: 28 mar. 2024

JIJINRAJ, P. Porting of automotive application on to embedded multicore platform. p. 2275–2279, jul. 2013.

JUNQUEIRA, F. **Seca em Taiwan pode tornar escassez de chips ainda pior do que o esperado**. Disponível em: https://canaltech.com.br/hardware/seca-em-taiwan-pode-tornar-escassez-de-chips-ainda-pior-do-que-o-esperado-184120/. Acesso em: 26 fev. 2024.

KAUR, D. **Counterfeit chip a problem as global shortage increases semiconductor fraud**. Disponível em: https://techwireasia.com/06/2022/is-the-global-chip-shortage-causing-more-semiconductor-fraud-counterfeits/ . Acesso em: 26 fev. 2024.

LITTLEFAIR, T. **C and C++ Code Counter**, 2006. Disponível em: https://cccc.sourceforge.net/ . Acesso em: 27 mar. 2024.

MARTIN, R. C. **Clean code:** a handbook of agile software craftsmanship. 1. ed. Boston: Pearson Education, Inc, 2008. v. 1.

MERCURIAL. 2024. Disponível em: https://www.mercurial-scm.org/ . Acesso em: 27 mar. 2024.

MISRA CONSORTIUM LIMITED. **MISRA**. Disponível em: https://misra.org.uk/misra-c/ . Acesso em: 4 abr. 2024.

MULLINS, B. E.; PACK, D. J. **Migration from the MC68HC11 to the MC68HC12 within an electrical & computer engineering curriculum.** *In*: AMERICAN SOCIETY FOR ENGINEERING EDUCATION ANNUAL CONFERENCE & EXPOSITION - ASEE. **[Proceedings...]**, 2002.

SOFTWARE RELIABILITY ENHANCEMENT CENTER; TECHNOLOGY HEADQUARTERS; INFORMATION-TECHNOLOGY PROMOTION AGENCY, J. **ESCR, embedded system development coding reference guide**. 3. ed. [s.l.] SEC BOOKS, 2018.

THORBECKE, C. **EUA estão investindo US$ 52 bilhões para impulsionar fabricação de chips no país**. Disponível em: https://www.cnnbrasil.com.br/economia/eua-estao-investindo-us-52-bilhoes-para-impulsionar-fabricacao-de-chips-no-pais/. Acesso em: 26 fev. 2024.

TURNER, A. *et al.* **Sphinx**, 2024. Disponível em: https://www.sphinx-doc.org/en/master/index.html . Acesso em: 28 mar. 2024.

AGRADECIMENTOS
Agradeço primeiramente a Deus pela saúde e oportunidade de poder me dedicar ao progresso da ciência e educação no Brasil.
Agradeço à minha esposa, Priscila, pelo incentivo e compreensão durante as horas dedicadas a esse trabalho.
Também agradeço ao amigo Paulo Costamilan pela parceria no desenvolvimento do projeto utilizado como exemplo para a metodologia proposta.
Por fim, agradeço ao SENAI São Paulo pela confiança e apoio ao desenvolvimento científico e tecnológico.

Sobre o Autor

____

[i] **Leandro Poloni Dantas**

Engenheiro (2004) e Doutor (2018) em Engenharia Elétrica pelo Centro Universitário FEI. Atuou por 15 anos na indústria eletrônica no desenvolvimento de novos produtos. Desde 2009, vem lecionando em cursos de pós-graduação, graduação e de nível técnico em diferentes instituições paulistanas. Atualmente é professor no Centro Universitário SENAI São Paulo e no Insper. https://orcid.org/0000-0003-3674-336X