UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

HUMBERTO VARGAS GOMES

Metodologia de Projeto de Software Embarcado Voltada ao Teste

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Luigi Carro
Orientador

Prof$_a$. Dr$_a$. Érika Fernandes Cota
Co-orientadora

Porto Alegre, maio de 2010.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Gomes, Humberto Vargas

Metodologia de Projeto de Software Embarcado Voltada ao Teste / por Humberto Vargas Gomes. – Porto Alegre: Programa de Pós-Graduação em Computação, 2010.

104 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2010. Orientador: Luigi Carro; Co-orientadora: Érika Fernandes Cota.

1.Projeto de software embarcado. 2.Teste de software embarcado. 3.Modelos simplificados de hardware. 4.Casos de teste de software. 5. Medição eletrônica de energia. I. Carro, Luigi. II. Cota, Érika Fernandes. III. Título.

AGRADECIMENTOS

Agradeço primeiramente a minha esposa Alessandra, pelo incentivo e apoio em todas as horas, sem seu auxílio e compreensão este trabalho jamais teria sido concluído. Agradeço aos meus pais, Carlos e Gilda, que jamais permitiram que nada me faltasse e que sempre me incentivaram a estudar, apesar de todas as dificuldades enfrentadas. Agradeço minha irmã Karla e meu cunhado Fábio, pela força em todas as horas e aos meus sogros, João e Ana, pela companhia, pelos cafezinhos e bolinhos, sempre bem vindos.

Não poderia deixar de mencionar o auxílio dos colegas de trabalho, especialmente dos engenheiros Andre Bauermann e Mariélio Silva, que sempre me liberaram, mesmo durante o horário de trabalho, para compromissos referentes ao mestrado, sem oferecer qualquer tipo de resistência.

Também preciso agradecer a gigantesca paciência e força dos meus orientadores, professores Luigi e Érika, que no decorrer dos últimos anos estiveram sempre ao meu lado, me ajudando e orientando na confecção deste trabalho.

Finalmente, agradeço a Deus, pela benção da vida.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

A/D	Analog-to-Digital
ADC	Analog-to-Digital Converter
ANEEL	Agencia Nacional de Energia Elétrica
CACC	Correlated Active Clause Coverage
CFG	Control Flow Graph
COFF	Commom Object File Format
CRC	Cyclic Redundancy Check
D/A	Digital-to-Analog
DFT	Design For Testability
EMITM	Embedded Systems's Interface Test Model
FSM	Finite State Machine
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
I/O	Input/Output
IRDA	Infrared Data Association
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
LTL	Linear Temporal Logic
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MC/DC	Modified Condition/Decision Coverage
MCH	Model Conductor Hardware
MIPS	Million Instructions Per Second
MSC	Message Sequence Chart
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTC	Real Time Clock
RTM	Regulamento Técnico Metrológico

SPI	Serial Peripheral Interface Bus
UART	Universal Asynchronous Receiver Transmitter.
UML	Unified Modeling Language
USB	Universal Serial Bus

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Devido ao crescente incremento de complexidade do software embarcado atual, dada a abundância de recursos disponíveis de hardware, está cada vez mais difícil manter a qualidade do software embarcado desenvolvido sem incorrer em aumentos de custo que inviabilizem o projeto. Com isto, o teste de software embarcado é atualmente uma importante área de pesquisa, onde são buscadas técnicas de teste que maximizem o número de falhas encontradas ainda em tempo de projeto e a um custo satisfatório. Muitas das soluções pesquisadas envolvem aspectos não apenas relativos ao teste propriamente dito, mas ao projeto do produto desde a sua concepção, daí a necessidade de metodologias conjuntas de desenvolvimento e teste. Neste trabalho, é apresentada uma metodologia de desenvolvimento e testes de software embarcado com o objetivo de permitir que grande parte da tarefa de desenvolvimento e teste seja executada em um ambiente de desenvolvimento de software de aplicação, sem a presença do hardware. Neste ambiente, o desenvolvimento é pensado desde o início do projeto visando à qualidade do teste, assim caracterizando esta metodologia como uma técnica DFT (do inglês *design for testability*). Na abordagem proposta, o hardware físico é substituído por modelos funcionais, construídos na mesma linguagem de programação do software em desenvolvimento. O uso destes modelos permite ao desenvolvedor a construção e aplicação de casos de teste capazes de exercitar o software embarcado tanto no ambiente de software de aplicação, quanto na plataforma alvo, sem alterações. Esta dissertação mostra a metodologia sendo aplicada ao software embarcado de um medidor eletrônico de energia, onde cinco modelos de dispositivos de hardware foram construídos, que permitiram a execução tanto de testes de unidade, quanto de testes de integração, em um ambiente de desenvolvimento de software de aplicação. Finalmente, uma análise de cobertura, realizada com o auxílio de uma ferramenta que, de outra forma, não seria compatível com o software da plataforma alvo, mostrou que a execução conjunta do software e dos modelos permite atingir a cobertura de quase a totalidade do software embarcado desenvolvido, onde os casos de teste foram capazes de verificar desde as camadas de software de aplicação até as camadas de software dependente do hardware.

Palavras-Chave: projeto de software embarcado, teste de software embarcado, modelos de dispositivos de hardware.

ABSTRACT

Due to the growing increment of complexity of the current embedded software, given the abundance of hardware resources, it is becoming increasingly difficult to maintain the software quality without requiring high development and test costs that could make the project impracticable. In this context, embedded software testing is an important research area, where test techniques that maximize the number of errors detected during design time at a satisfactory cost have been investigated. Many of the proposed solutions involve aspects not related only to the testing itself, but to the product design since its conception, hence the need of methodologies for the development and test of software. In this work, we present a methodology of development and test of embedded software that allows the execution of most of the task of development and test in an application software development environment, without the physical hardware. In the application software environment, the development is thought, since the first stages, aiming the execution of the test, hence this methodology can be seen as a DFT (design for testability) technique. In the proposed approach, the physical hardware is replaced by functional models, constructed using the same programming language of the embedded software under development. The use of such models allows the developer to construct and apply test cases capable of exercising the embedded software both in the application software environment and in the target platform environment, without any change. In this work, the presented methodology is applied to the embedded software of an electronic energy meter, where five hardware device models were constructed, which enabled the execution of both unit and integration tests in the application software environment. Finally, the coverage analysis, performed with a software tool that otherwise would not be compatible with the target platform, showed that the simultaneous execution of the software and the models make it possible to achieve an almost complete coverage of the developed embedded software, where the test cases were able to verify the software from the application layers to the hardware dependent layers.

Keywords: embedded software design, embedded software testing, hardware device models.

1 INTRODUÇÃO

Apesar de sistemas embarcados serem ainda muito associados a dispositivos microprocessados limitados com relação à capacidade de processamento, memória RAM e armazenamento de código, definições mais modernas deixam as limitações de lado e apontam para dispositivos capazes de desempenhar tarefas extremamente complexas, tanto pela evolução do hardware, quanto do software. De fato, software embarcado tem se tornado complexo. Quando se associa tal complexidade a grandes volumes de produção e, muitas vezes, impossibilidade de atualizações futuras, percebe-se a importância do teste no desenvolvimento do software embarcado atual. Estima-se que 80% das falhas em um sistema embarcado são causadas por software e não por hardware, apesar do software ser considerado de 10% a 20% do sistema embarcado como um todo (SEO et al., 2008). O custo de um defeito grave de software, se descoberto em campo, pode inviabilizar completamente o negócio, gerando prejuízos difíceis de mensurar, tanto financeiros quanto com relação à imagem do fabricante perante os consumidores.

As razões citadas estão obrigando fabricantes de sistemas embarcados, que historicamente garantiam a qualidade de seus produtos apenas com testes no próprio equipamento, a se adaptarem, passando efetivamente a executar testes de software de forma sistemática. É uma mudança significativa, considerando-se que, no universo de sistemas embarcados, ferramentas de teste de software eram apenas conhecidas por empresas de áreas críticas de atuação, como biomédica, militar, automotiva e aeroespacial, onde defeitos, sejam de hardware ou software, jamais puderam ser tolerados.

Na contramão do que se poderia imaginar, o aumento de complexidade do software não vem acompanhado de um proporcional aumento do preço de venda dos produtos ou margens de lucro por parte das empresas. Ao contrário, os preços de venda tendem a diminuir, devido à intensa competição entre fabricantes. Considerando-se que grande parte do custo de desenvolvimento de sistemas embarcados é devido a testes e correções de problemas (PFALLER et al., 2006), uma ferramenta de testes ou metodologia de desenvolvimento que não seja financeiramente viável, não terá aplicação real para grande parte das empresas.

Neste contexto, o teste de software embarcado é atualmente uma importante área de pesquisa, onde são buscadas técnicas de teste que maximizem o número de falhas encontradas ainda em tempo de projeto, e a um custo satisfatório. Muitas das soluções pesquisadas envolvem aspectos não apenas relativos ao teste propriamente dito, mas ao projeto do produto desde a sua concepção.

1.1 Software para Sistemas Embarcados

O software de um sistema embarcado é diferente de um software de aplicação (software executado em um microcomputador) devido, principalmente, à forte interconexão com os dispositivos de hardware periféricos, às restrições impostas pela aplicação na qual ele está inserido (requisitos de tempo real, por exemplo) e ao modo como ele é desenvolvido. Em um computador pessoal o hardware é controlado por um complexo sistema operacional, que gerencia memória, vídeo, teclado e meios de comunicação e o software desenvolvido precisa apenas se deter a problemas referentes à aplicação. O software de um sistema embarcado, por outro lado, acessa diretamente o hardware e deverá ser escrito de forma a se adaptar aos recursos de hardware disponíveis, visando executar uma tarefa bastante específica.

Entre o software embarcado e o software de aplicação, está o software que executa em sistemas operacionais embarcados (Figura 1.1). Estes sistemas operacionais requerem, para a sua execução, uma quantidade considerável de recursos de hardware, o que os torna inviáveis para muitas aplicações. Entretanto, à medida que os sistemas embarcados têm se tornado mais sofisticados e com maior conectividade, o uso de sistemas operacionais como Linux e Windows embarcados têm se tornado maior (KANG; KWON; LEE, 2005). Neste trabalho, será dado foco a sistemas embarcados com menos recursos disponíveis de hardware, em geral, compostos por um único microcontrolador central, contendo uma série de hardwares periféricos e sem qualquer tipo de sistema operacional. Tais sistemas embarcados serão chamados de sistemas embarcados baseados em microcontroladores.

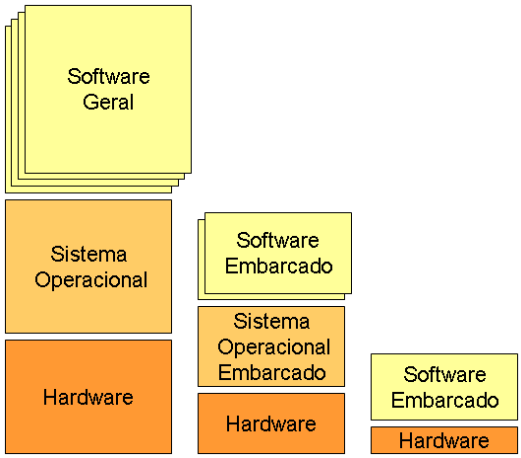


Figura 1.1: Relação entre software embarcado e software geral

Na Figura 1.2 é apresentada a arquitetura interna de um típico microcontrolador para sistemas embarcados. Nela observam-se diversos periféricos de hardware (memória Flash e RAM, circuito de *watchdog*, pinos de entrada e saída, unidade de multiplicação por hardware, temporizadores diversos, relógio de tempo real, unidade de cálculo de CRC e conversor analógico-digital) interconectados a um barramento interno do microcontrolador. No software embarcado, o acesso a estes periféricos é feito, usualmente, através de escritas e leituras em registradores mapeados em locais

específicos da memória e, como será visto mais adiante neste trabalho, as formas como as ferramentas de desenvolvimento de software lidam com estes registradores, são algumas das causas das incompatibilidades existentes entre o software para sistemas embarcados e o software de aplicação.

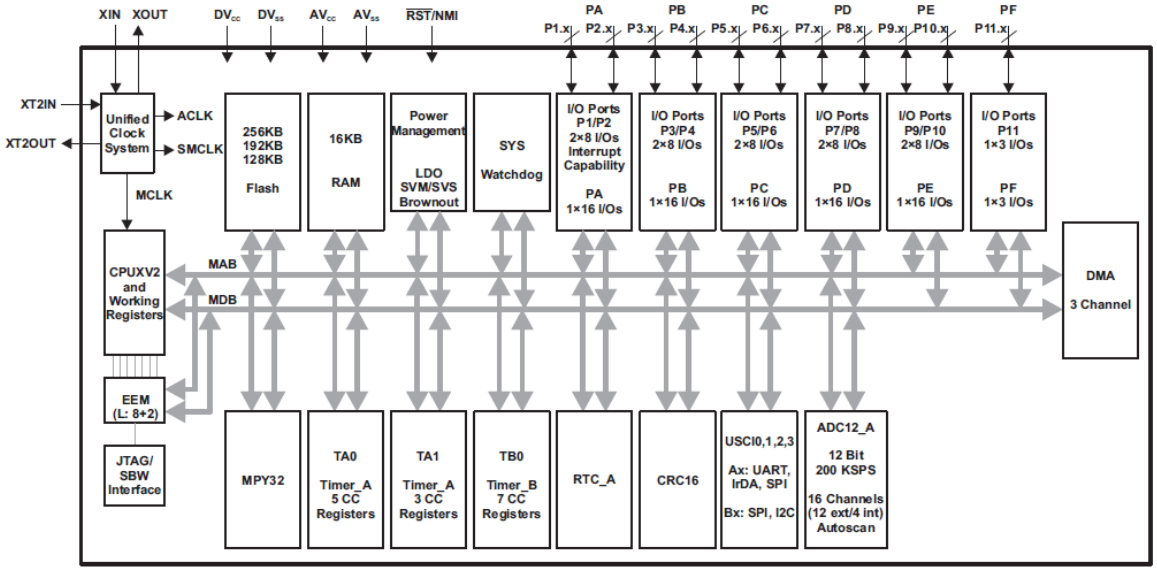


Figura 1.2: Arquitetura do microcontrolador MSP430F5438A fabricado pela Texas Instruments.

1.2 Desafios no Teste de Software Embarcado

Os avanços na tecnologia de fabricação de circuitos integrados têm permitido a fabricação de microcontroladores com cada vez mais recursos de hardware e a custos cada vez menores. Esta disponibilidade de recursos tem sido traduzida em um crescente incremento da complexidade do software embarcado. De fato, o software embarcado está extremamente sofisticado, muitas vezes substituindo o hardware, e vem se tornando a maior parte do sistema embarcado (SEO et al., 2008).

Entretanto, devido à dependência do hardware, o software embarcado, por muito tempo, tem sido desenvolvido após a construção de um protótipo do equipamento ou com o auxílio de kits de desenvolvimento. Nas etapas iniciais, anteriores ao primeiro protótipo, o hardware é avaliado através de uma plataforma de hardware padrão, fornecida pelo fabricante do microcontrolador utilizado. O software é então desenvolvido com um pacote de ferramentas (compiladores, simuladores e debuggers) bastante específico para a plataforma alvo. Testes de software são executados durante as etapas de desenvolvimento, onde programadores testam a funcionalidade de cada módulo individualmente (testes de unidade) e, com o auxílio de protótipos ou hardware de avaliação, verificam a correta execução do software na plataforma alvo, onde é executada a grande maioria dos testes de integração e de sistema. Conforme (SEO et al., 2008), muitas vezes tais testes são executados em uma abordagem do tipo big-bang, onde diferentes partes da aplicação são unidas ainda em um estágio bastante instável. A

validação do sistema embarcado como um todo é deixada para os testes funcionais na plataforma alvo, onde o atendimento aos requisitos do projeto é verificado e o equipamento é sujeito a condições extremas de uso.

Esta abordagem de desenvolvimento, dada a complexidade dos sistemas embarcados atuais, não é capaz de garantir a confiabilidade do equipamento, uma vez que o software embarcado atual é composto por uma intrincada combinação de milhares de linhas de código que não podem ter sua funcionalidade verificada sem que se faça uso de técnicas modernas de teste de software. Na contramão deste processo, as ferramentas de desenvolvimento de sistemas embarcados baseados em microcontroladores não estão evoluindo na mesma velocidade. Métodos de geração de casos de teste, análise de cobertura de teste, simuladores que sejam capazes de simular o hardware como um todo (e não apenas o microprocessador) e eficientes analisadores estáticos de código não são comumente encontrados em tais ambientes de desenvolvimento, como, por exemplo, nas IDEs (do inglês *integrated development environment*) Code Composer (CODE COMPOSER, 2010), e IAR (IAR, 2010), fabricados pela Texas Instruments (TEXAS, 2010) e IAR Sytems, respectivamente. Infelizmente, devido às particularidades do software embarcado, ferramentas de teste de software geral, facilmente encontradas no mercado, não podem ser diretamente aplicadas a este, pela simples razão de que os acessos ao hardware não serão compreendidos pela ferramenta que, na maioria das vezes, não será sequer capaz de executar o código embarcado.

Neste contexto, visando-se à eliminação de algumas das restrições impostas ao software embarcado que o impedem de ser completamente testado, principalmente com relação ao seu acoplamento com o hardware, algumas abordagens têm sido propostas, como em (ENGBLOM; GIRARD; VERNER, 2006) onde são apresentados todos os benefícios do teste de sistemas embarcados em um ambiente totalmente simulado chamado Simics, ou em (KARLESKY; WILLIAMS, 2007) onde a proposta é a substituição de módulos complexos, ou ainda não existentes, por funções que eles chamaram de Mocks, que possuem a mesma interface das funções substituídas e que podem ser utilizadas de forma a tornar possível o ambiente de testes.

Outros trabalhos focam no ponto de que a forma de desenvolvimento do software embarcado é fundamental para o processo de teste. De fato, um software mal projetado e construído, dificilmente poderá ser bem testado. Em (KANG; KWON; LEE, 2005) é apresentada uma interessante organização do sistema embarcado em camadas, sendo constituído pelo software de aplicação, na camada superior, pelo software dependente do hardware na camada intermediária e pelo hardware propriamente dito na camada inferior. Nesta mesma área estão as técnicas de DFT (do inglês *design for testability*) (KANSTRÉN, 2008), onde o teste é pensado desde as primeiras etapas de projeto. Em (GREENE, 2004), técnicas ágeis de desenvolvimento de software foram aplicadas ao universo de sistemas embarcados, onde, dentre outras, ressalta-se a criação de código a partir da especificação de casos de teste de unidade, o que auxiliou os desenvolvedores a pensar em detalhes na funcionalidade do código, antes do início do processo de desenvolvimento de software propriamente dito.

1.3 Contribuições

O crescente aumento de complexidade do software embarcado atual, as limitações das ferramentas de desenvolvimento com relação aos recursos de teste de software e as incompatibilidades que impedem o uso de ferramentas de testes de software de

aplicação para o teste de software embarcado são motivações do presente trabalho, onde uma abordagem de desenvolvimento e teste de software embarcado é proposta com o objetivo de tornar possível o teste do software embarcado, inclusive dos módulos de software dependentes do hardware, com ferramentas de teste de software de aplicação, mesmo sem a presença do hardware nas etapas iniciais do projeto. Com a aplicação desta metodologia espera-se a redução dos tempos de desenvolvimento do software embarcado e o aumento da confiabilidade do software gerado.

O sucesso desta abordagem baseia-se nos seguintes aspectos:

- O crescimento do software embarcado, tanto com relação ao tamanho, quanto com relação ao aumento de complexidade, não se dá na parcela de software dependente do hardware, mas no software de aplicação. Entretanto, a forma como o software embarcado de aplicação se relaciona com o software dependente do hardware o impede de ser testado em ambientes de teste de software de alto nível. A afirmação de que o aumento de complexidade se dá no software de aplicação pode ser verificada pela análise dos novos microcontroladores, onde não é observada a inclusão de um número de periféricos de hardware capazes de consumir a grande expansão da capacidade de código. O impedimento de se testar o software embarcado em ambientes de teste de software de aplicação pode ser verificado pelo simples fato de que, na grande maioria das vezes, o software embarcado simplesmente não pode ser compilado para ser executado em tais ambientes;
- A organização do software embarcado (estruturação em camadas, controle do nível de acoplamento com o hardware e redução da complexidade dos módulos) é crucial para o sucesso do teste. Grande parte da incompatibilidade entre o software embarcado e o software de aplicação pode ser eliminada pela simples forma de construção do código, onde os registradores que acessam o hardware são devidamente mapeados para regiões de memória conhecidas pela ferramenta de testes e aspectos específicos dos compiladores das plataformas alvo são considerados. Estes cuidados tornam possível a execução do software embarcado em ferramentas de teste de software de aplicação, abrindo um leque de possibilidades aos desenvolvedores;
- Uma grande parte da funcionalidade dos dispositivos de hardware pode ser definida por modelos simples de tais dispositivos, fáceis de serem construídos. Desenvolvidos na mesma linguagem de programação do software embarcado e construídos a partir da análise de informações técnicas dos microcontroladores, etapa necessária para o desenvolvimento do código propriamente dito, o uso de tais modelos permite a execução de testes do software embarcado, inclusive do software que acessa o hardware, com ferramentas de teste de software de aplicação. Ainda, os casos de testes desenvolvidos poderão ser completamente aproveitados nos testes executados na plataforma alvo, nas etapas finais de projeto;

1.4 Organização

Esta dissertação está organizada conforme segue:

No capítulo 2 são apresentados aspectos relevantes do teste de software aplicados ao universo de software embarcado. São discutidas técnicas de geração de casos de teste e

mecanismos de abstração de hardware ou software (stubs, mocks e modelos). Além disto, faz-se uma breve apresentação de ferramentas de desenvolvimento e teste de software, tais como visualizadores de código e softwares de análise de cobertura.

O capítulo 3 apresenta a metodologia de desenvolvimento e teste proposta, apontando os principais detalhes para a sua execução, tanto com respeito à organização e estruturação do código, quanto com relação à manutenção da compatibilidade de software e ao uso e construção dos modelos dos dispositivos de hardware.

No capítulo 4 são apresentados aspectos relevantes do software do medidor eletrônico de energia desenvolvido para esta dissertação. Nele, descreve-se a plataforma alvo, faz-se um resumo das tarefas executadas pela aplicação e apresentam-se os módulos de software construídos e suas principais atribuições.

No capítulo 5 é apresentada uma aplicação da metodologia proposta ao software embarcado do medidor eletrônico de energia descrito no capítulo anterior. Neste capítulo são descritas as funcionalidades e recursos de teste dos cinco modelos de dispositivos de hardware criados e executados juntamente com a aplicação. Na seqüência, a aplicação dos modelos é apresentada de forma prática onde, ressaltando-se os principais recursos desenvolvidos, se mostra como podem ser executados testes de unidade e de integração na aplicação embarcada, quando executada juntamente com os modelos dos dispositivos de hardware. Ao final, são mostrados os cenários de testes utilizados para a execução dos casos de teste criados, se apresenta uma análise da cobertura de testes obtida e faz-se uma breve explicação das limitações do método proposto.

Finalmente, no capítulo 6 são apresentadas as conclusões e uma expectativa de trabalhos futuros.

2 TESTE DE SOFTWARE EMBARCADO

O teste de software como um todo pode ser definido como a aplicação de uma série de técnicas de teste e estratégias de desenvolvimento que visam o aumento da confiabilidade do software sendo testado. Tal objetivo, em geral, é atingido quando diferentes aspectos do software em teste são examinados. O uso de ferramentas de análise estática, por exemplo, permite a localização de potenciais problemas na forma como o software foi construído, enquanto a execução de casos de teste bem construídos leva a uma verificação do atendimento dos requisitos de projeto. A análise de cobertura de teste permite a verificação do alcance dos testes executados, expondo regiões não testadas, ou devido a casos de teste mal projetados ou por requisitos de projeto mal documentados. Visando então ressaltar aspectos importantes com relação ao teste de software embarcado, neste capítulo faz-se um apanhado de alguns trabalhos encontrados na literatura atual.

2.1 A Geração de Casos de Teste

Em software, um caso de teste pode ser definido como um conjunto de entradas, condições de execução e um critério de sucesso ou falha (PEZZÉ; YOUNG, 2008, p.173). O problema da geração de casos de teste é clássico e a busca por casos de teste interessantes é um dos mais importantes aspectos do teste de software. Entende-se por um caso de testes interessante, aquele que possui uma grande probabilidade de sucesso na localização de um eventual defeito ou falha (PFALLER et al., 2006).

A seguir, são apresentadas formas utilizadas para a geração de casos de teste a partir de requisitos do projeto, análise de cobertura e análise de interfaces, aplicadas ao universo de software embarcado.

2.1.1 Atendimento de Requisitos do Projeto

De forma intuitiva, a primeira etapa de um procedimento de testes seria a verificação do atendimento aos requisitos do projeto. De fato, a análise dos requisitos, para grande parte dos projetos, ainda é a fonte mais utilizada para a geração de casos de teste. Uma abordagem de teste baseada nos requisitos é dita funcional, uma vez que se preocupa com a funcionalidade do software testado, ou seja, com o que ele realiza e não com a forma como ele está construído.

Entretanto, conforme (PFALLER et al., 2006), não existe ainda uma forma completamente padronizada e definida de se derivar especificações de casos de teste a partir de requisitos de projeto especificados de maneira informal. Neste contexto, neste mesmo trabalho, os requisitos foram especificados como serviços, uma série de ações representadas em MSCs (do inglês *Message Sequence Charts*) (figura 2.1) e as

especificações de casos de teste foram diretamente derivadas destes. Na figura 2.2, observa-se que a especificação dos requisitos na forma de serviços é feita ainda em um elevado nível de abstração, por isto, nesta camada, apenas especificações de casos de teste podem ser geradas, e não os casos de teste propriamente ditos, dada a falta de informações detalhadas para tal. Esta abordagem é bastante interessante, pois garante que todos os requisitos de projeto são contemplados com ao menos um caso de teste. Outras formas de especificação formal de requisitos podem ser observadas em (ESSER; STRUSS, 2007), onde os requisitos são definidos como FSM (do inglês *Finite State Machines*) (figura 2.3) e em (WHALEN et. al, 2006) onde os requisitos são especificados em LTL (do inglês *Linear Temporal Logic*).

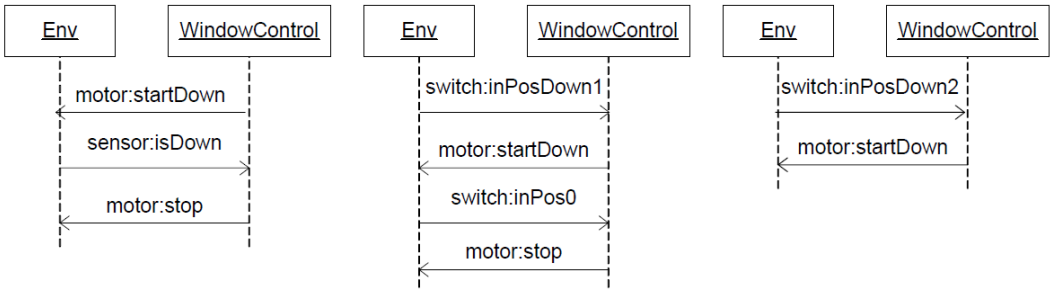


Figura 2.1: Requisitos de projeto especificados como serviços e descritos na forma de MSCs (PFALLER et al., 2006).

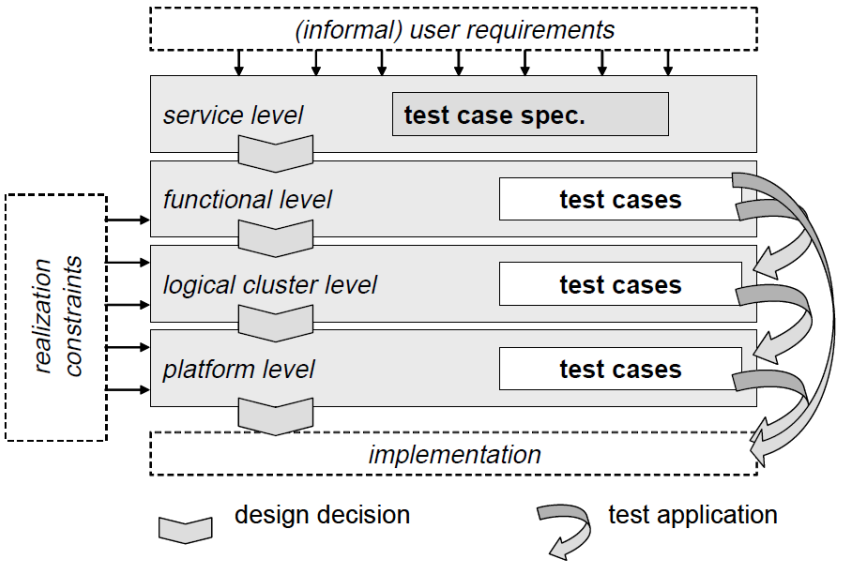


Figura 2.2: Fluxo de geração de casos de teste a partir de requisitos informais do projeto (PFALLER et al., 2006).

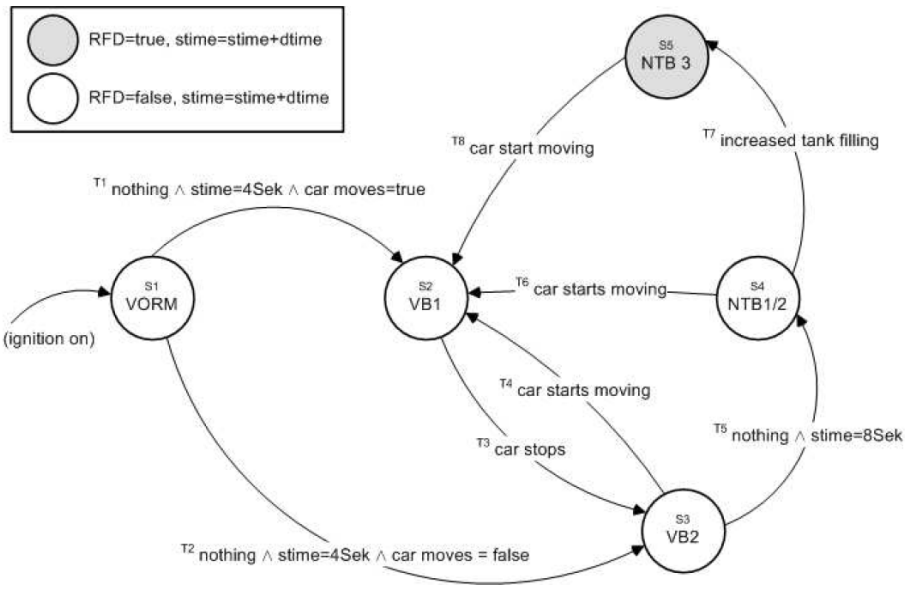


Figura 2.3: Especificação de requisitos de projeto como FSMs (ESSER; STRUSS, 2007).

Para a especificação da aplicação em muitos níveis, o UML (do inglês *Unified Modeling Language*) é uma das ferramentas utilizadas. A partir do UML 2.0 (UML, 2010), treze tipos distintos de diagramas estão disponíveis, sendo classificados como diagramas estruturais, que representam a estrutura estática da aplicação e diagramas comportamentais, onde diferentes tipos de interações são representadas conforme explicado em (BRISOLARA; KREUTZ; CARRO, 2009). Neste mesmo trabalho, visando aumentar a capacidade de UML de representar características específicas de sistemas embarcados, como concorrência e compartilhamento de recursos, uma extensão de UML chamada MARTE (do inglês *Modeling and Analysis of Real-Time and Embedded systems*) é apresentada.

Um importante aspecto a ser considerado é que, enquanto o uso de ferramentas de modelagem para representar características do software embarcado em um alto nível de abstração é recomendável, a representação da implementação propriamente dita pode se tornar demasiadamente onerosa, uma vez que a construção do modelo poderá vir a ser tão ou mais complexa do que a construção do código. Neste sentido, há trabalhos que buscam a geração automática de código a partir dos modelos criados, de forma que os projetistas não dupliquem esforços. Entretanto, enquanto tais ferramentas e métodos não estiverem completamente amadurecidos, é fundamental que a técnica de modelagem utilizada não consuma uma quantidade muito grande de recursos de desenvolvimento. Na prática, exatamente até onde se avança em especificação e modelagem é uma opção de cada empresa e diversos fatores são levados em consideração, tais como, recursos humanos, complexidade da aplicação, prazos de entrega e vida útil da plataforma em desenvolvimento.

2.1.2 Verificação da Cobertura de Teste

A geração de casos de teste a partir da análise de cobertura provém da óbvia constatação de que não é possível atestar a qualidade de partes do software que não

foram executadas. Tais métodos de geração de casos de teste são classificados como métodos de geração a partir da análise estrutural, uma vez que dizem respeito à forma como o software foi construído e não a sua funcionalidade. Seu objetivo primordial é a geração de casos de teste capazes de exercitar todas as estruturas internas do código.

Um importante aspecto da geração de casos de teste a partir da análise estrutural, diz respeito à possibilidade de automação do processo de teste. Enquanto a análise funcional parte de uma especificação abstrata de requisitos, e são buscadas técnicas que permitam a sua especificação formal e uma conexão direta com modelos subseqüentes, a análise estrutural já parte de um aspecto bastante concreto, o código propriamente dito. Desta forma, é perfeitamente possível o uso de software que execute, de forma automática, a extração e aplicação de casos de teste.

Os critérios de cobertura de código podem ser divididos basicamente em dois tipos, baseados no fluxo de controle e baseados no fluxo de dados (PRESSMAN, 1995, p. 807). Quando se baseia no fluxo de controle, vê-se um software na forma de um CFG (do inglês *control flow graph*), onde os nós são os blocos básicos de execução e os arcos representam os saltos para os nós subseqüentes (Figura 2.4). Métodos de análise de cobertura têm por objetivo visitar todos os nós, como nos testes de caminho básico, todos os arcos, como nos testes de decisão ou, ainda, verificar se todas as condições lógicas foram exercitadas, como nos testes de condição. Outra forma de se efetuar a cobertura seria baseando-se no fluxo de dados, onde seriam utilizadas técnicas de análise do tipo produtor-consumidor, por exemplo (PEZZÉ; YOUNG, 2008).

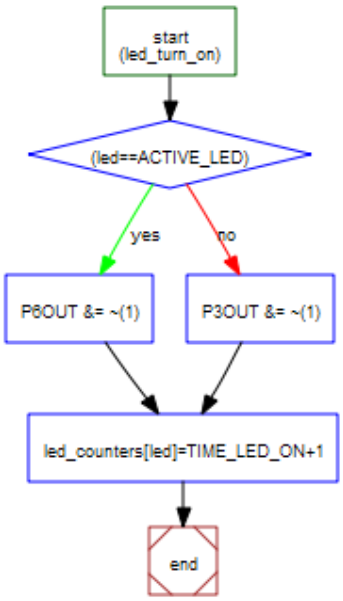


Figura 2.4: Exemplo de uma simples função de acionamento de leds na forma de um CFG gerada pela ferramenta de análise de software Understand, fabricada pela SCI Tools (SCITOOLS, 2010).

Um dos testes de condição amplamente utilizados em aplicações críticas sob o aspecto de segurança é o MC/DC (do inglês, *Modified Condition/Decision Coverage*). Conforme (WHALEN et. al, 2006), o MC/DC é uma métrica de análise de cobertura

que visa demonstrar o efeito independente de expressões booleanas básicas na decisão em que elas ocorrem. Ainda conforme este mesmo trabalho, diz-se que um conjunto de testes satisfaz o critério MC/DC se cada ponto de entrada e saída do módulo foi exercitado ao menos uma vez, cada condição básica em uma decisão foi exercitada com todos os possíveis resultados e, finalmente, se cada condição básica mostrou afetar, de forma independente, o resultado da decisão.

Em (GUAN; OFFUTT; AMMANN, 2006) é apresentada uma análise onde casos de teste funcionais gerados manualmente são comparados com casos de teste estruturais, utilizando-se a técnica chamada CACC (do inglês, *Correlated Active Clause Coverage*), que seria uma simplificação da técnica MC/DC. O artigo conclui que os casos de teste gerados a partir do critério CACC foram capazes de localizar falhas importantes que não foram localizadas nos testes funcionais, além disto, o texto sugere que o custo de implementação do método, desconsiderando-se o treinamento do executor dos testes, foi menor do que na implementação de testes funcionais. Conforme o autor, tal redução nos custos se deve principalmente ao aumento no uso de automação e métodos de simulação visando à redução do esforço humano.

Um importante aspecto com relação à geração de casos de teste por análise de cobertura diz respeito à descoberta de pontos do software que deverão ser testados de forma a se maximizar a cobertura obtida. Apesar de não ser significativo para aplicações simples, onde todo o software poderia ser testado sem qualquer tipo de análise, em aplicações complexas isto pode se tornar um ponto chave de decisão, considerando-se a possibilidade de obtenção de uma grande cobertura, com um reduzido esforço de teste. Neste contexto, em (LI, 2005) é apresentada uma técnica visando a localização destes pontos de teste chamada de método global de cálculo de prioridade. A técnica proposta é uma expansão da análise de dominação que, de forma simples, consiste na definição de que um bloco de software A domina um bloco de software B se a obtenção de cobertura de A implica a obtenção da cobertura de B. Enquanto a análise de dominação é feita apenas em estruturas internas de um módulo ou função, o método proposto considera impactos da cobertura de determinado módulo na cobertura global do software. Fazendo uso desta técnica, uma ferramenta automática de priorização dos pontos de interesse do código a serem testados e geração de casos de teste que exercitem tais pontos é apresentada em (LI; WEISS; YEE, 2006).

Apesar dos claros benefícios de técnicas de análise estrutural a partir da cobertura de código, seu uso ainda é bastante limitado em projetos de software embarcado em geral, onde não há aspectos críticos ou de segurança envolvidos e testes baseados em especificações funcionais são ainda a ampla maioria. Em software embarcado, podem ser apontadas três causas principais para a não utilização em massa de tais técnicas:

I) A ausência de ferramentas de teste que gerem ou gerenciem tais casos de teste para as plataformas específicas de sistemas embarcados;

II) As dificuldades de instrumentação do software embarcado, devido às restrições de memória, requisitos de tempo real e processamento das plataformas alvo, conforme abordado em (WU et. al, 2007);

III) A falta de conhecimento técnico por parte do corpo de desenvolvimento das empresas, gerando a visão de que testes estruturais não são realmente necessários e que consomem uma grande quantidade de recursos, sejam financeiros, sejam de tempo de projeto.

2.1.3 Análise das Interfaces

Visando o teste de interfaces, em (SUNG; CHOI; SHIN, 2006) é apresentado um conjunto de itens de teste, basicamente listas de aspectos a serem contemplados pelos testes, para interfaces de hardware e interfaces de sistema, chamado de EmITM (do inglês *Embedded System's Interface Test Model*). Os itens de teste são agrupados conforme diversas características testáveis, como, por exemplo, memória, dispositivos de entrada e saída, timer, etc.

Neste mesmo conceito, defendendo a definição de um sistema embarcado como a união de camadas heterogêneas de hardware, sistema operacional e software de aplicação (Figura 2.5), (SEO et. al, 2007) apresenta a iteração entre estas diferentes camadas, as interfaces, como um importante critério de seleção de casos de teste e monitoramento dos resultados de teste de software embarcado. Neste trabalho é feita uma classificação das interfaces conforme os padrões de relacionamento entre as diferentes camadas e casos de teste são gerados visando o teste de características pré-definidas de acordo com o tipo de interface classificada, tais características são obtidas do conjunto EmITM, previamente publicado em (SUNG; CHOI; SHIN, 2006).

Posteriormente, com o objetivo de mostrar porquê as interfaces devem ser consideradas como pontos críticos de teste, a mesma autora, em (SEO et al., 2008), executa experimentos visando à resposta de três questões com relação à definição de software embarcado. A primeira questão verifica se o software embarcado é realmente fortemente acoplado, a segunda questão investiga se realmente as interfaces estão mais sujeitas a falhas e a terceira questão visa a verificação de uma relação entre falhas de softwares embarcados localizadas em campo e as interfaces propostas. A partir da análise destes experimentos, os autores concluem sua teoria de que as interfaces são realmente pontos críticos para o teste de software embarcado.

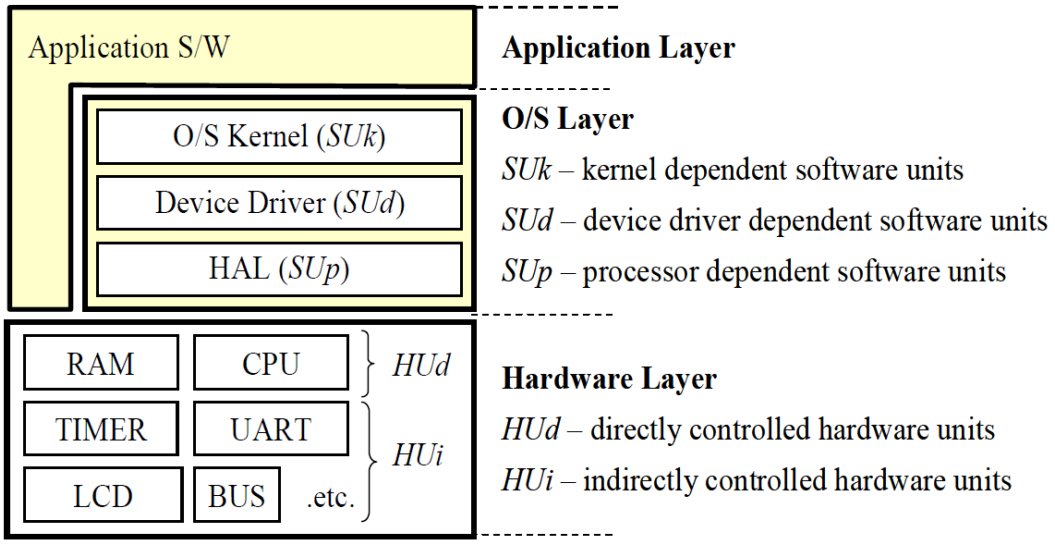


Figura 2.5: Definição de sistema embarcado conforme (SEO et. al, 2007).

Por todos os aspectos citados, verifica-se que as interfaces são importantes sob o ponto de vista de geração de casos de teste. Isto evidencia a necessidade de, desde as

etapas iniciais do projeto, se fazer um correto planejamento das atribuições e responsabilidades de cada módulo de software ou hardware, de modo que as interfaces construídas sejam capazes de atender, de uma forma inteligente, a todos os requisitos das aplicações. Desta forma, se obtém um maior encapsulamento dos módulos, com a conseqüente redução do acoplamento entre eles, dois resultados desejáveis sob o ponto de vista de qualidade de software.

2.2 Cenários de Teste

A execução prática de um caso de teste requer a criação deste dentro de um software chamado driver de teste. Este software é o responsável por criar todo o contexto necessário para a execução do caso de teste em um ambiente que simula as condições reais de execução do módulo em teste. Além disto, o driver também é o responsável pela coleta e análise dos resultados de teste, gerando relatórios quando necessário (Figura 2.6).

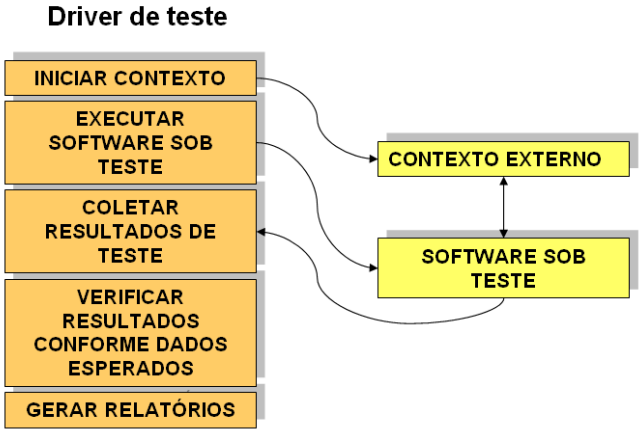


Figura 2.6: Software driver de teste.

Uma vez que o software driver também é um software, que precisa ser testado e depurado, para sistemas complexos a criação de um grande número de software de teste se torna um problema, dado o tempo e custo consumido para tal. Visando a redução do trabalho de desenvolvimento de software de teste (TSAI et. al, 2005) propõe a criação de padrões representando diferentes tipos de cenários de teste, de forma que, sempre que um caso de teste se enquadre dentro de algum padrão de cenário pré-existente, apenas o código referente aos detalhes específicos do caso de teste em si precisaria ser escrito, reduzindo-se, desta forma, o esforço de execução do teste.

Ainda em (TSAI et. al, 2005) oito diferentes tipos de cenários foram definidos e, através da análise de requisitos de um equipamento desfibrilador comercial, diferentes casos de teste foram gerados, buscando-se o enquadramento em algum tipo de cenário padrão. Com isto, o autor sustenta que obteve 95% de cobertura de teste utilizando-se casos de teste enquadrados nestes apenas oito diferentes tipos de cenários, conforme mostrado na Tabela 2.1.

Tabela 2.1: Cenários de teste gerados e percentuais de convergência obtidos (TSAI et. al, 2005).

Pattern descriptor	Coverage (%)
Basic scenario pattern (BSP)	40
Key-event–driven scenario (KSP)	15
Timed key-event–driven scenario (TKSP)	5
Key-event–driven time-sliced scenario (KTSP)	7
Command-response scenario (CSP)	8
Lookback scenario (LSP)	6
Mode-switch scenario (MSP)	8
Interleaving scenario (ISP)	6
Total coverage	95

2.3 Dublês de Teste (Stubs, Mocks e Modelos)

Conforme (KARLESKY; WILLIAMS, 2007) poucas funções operam de forma isolada em um código fonte. A maioria delas efetua chamadas para outras funções ou módulos e a composição destas funções constitui a implementação de um requisito de projeto. A abstração de recursos ou módulos não disponíveis é importante para testes em software de aplicação. Entretanto, em sistemas embarcados, onde dispositivos de hardware podem não estar disponíveis no momento do teste, ou quando características do ambiente externo de execução são fundamentais para a execução do teste, mecanismos de abstração de tais recursos são fundamentais.

Utilizando a nomenclatura proposta por (MESZAROS, 2009), mocks, stubs e modelos podem ser chamados de dublês de teste (Figura 2.7), ou seja, componentes que não se comportam exatamente como os componentes reais aos quais eles substituem, mas provêm a mesma interface, de forma que o software em teste não o diferencie do componente real. Ainda conforme (MESZAROS, 2009), o uso de dublês de teste poderá ser necessário quando um componente real não estiver disponível, quando ele não for capaz de responder conforme esperado pelo software de testes, quando a sua execução acarretar em efeitos indesejados para o teste, ou quando a estratégia de teste requerer uma maior controlabilidade ou visibilidade do processo como um todo.

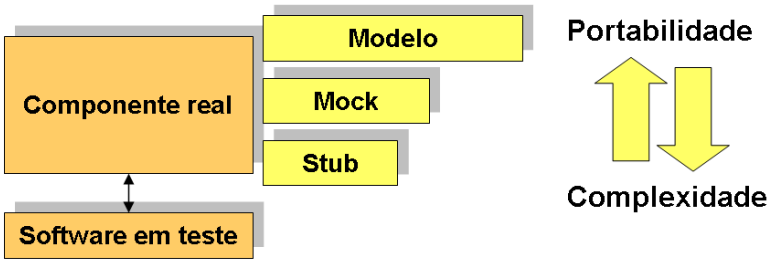


Figura 2.7: Dublês de teste.

Ao construir dublês de teste, cabe ao projetista de software decidir o nível de abstração necessário para o sucesso do teste. Isto dependente fortemente de características da aplicação, da complexidade requerida e dos objetivos que se deseja atingir com o uso de tais dublês. Testes executados com recursos modelados em níveis muito altos de abstração podem não ser eficientes na revelação de um número significativo de defeitos, entretanto, o custo de se modelar um dispositivo ou módulo de software com um comportamento muito próximo do dispositivo real pode se tornar proibitivo para muitas aplicações.

2.3.1 Stubs

O termo stub é utilizado muitas vezes com diferentes significados. Entretanto, em geral, stubs têm como principal função permitir que o teste seja executado, substituindo módulos não disponíveis ou não construídos, ou, ainda, que não possuam um papel fundamental para os objetivos do teste em execução. Stubs são construídos com exatamente a mesma interface do recurso substituído e, muitas vezes, de forma automatizada. Stubs, segundo (FOWLER, 2007), são capazes apenas de gerar respostas previsíveis, planejadas no ambiente de teste, e visando atingir os objetivos deste. Dentre os valores retornados, podem estar valores incrementais, randômicos ou buscados em tabelas. É possível também a atribuição de funcionalidades especiais, tais como, geração de relatórios, asserções visando cobertura de teste e contagem de chamadas.

2.3.2 Mocks

Utilizando a definição feita em (MESZAROS, 2009), a principal diferença de mocks, quando comparados a stubs, está no fato de que mocks podem ser programados para verificar e aguardar um comportamento esperado. Enquanto um stub apenas é capaz de gerar respostas pré-definidas, um mock estará aguardando um determinado comportamento e será capaz de avaliar o software em teste caso este comportamento não tenha sido detectado.

Utilizando mocks, um padrão de desenvolvimento de software embarcado chamado MCH (do inglês, *Model Conductor Hardware*) é proposto em (KARLESKY; BEREZA; ERICKSON, 2006). Conforme este trabalho, modelos guardam a informação de estado do sistema e comunicam-se apenas pelo condutor, não acessando de forma alguma a camada de hardware. O condutor, por sua vez, não possui qualquer informação de estado do sistema e serve apenas como uma interface de comunicação entre os módulos. Finalmente, o hardware representa uma fina camada de software existente sobre o hardware real, ou seja, neste caso, o hardware não é o hardware físico propriamente dito, mas uma camada de software que acessa o hardware. A estratégia propõe que tanto o modelo, quanto o hardware, possam ser substituídos por mocks (figura 2.8), de forma que testes possam ser amplamente executados, tanto em ambientes de simulação, quanto em execução na plataforma alvo.

A abordagem de desenvolvimento proposta é bem interessante, uma vez que visa o desacoplamento das camadas de software de aplicação e das camadas de software que acessam o hardware. Com os mocks, é possível a execução do sistema como um todo, mesmo sem a presença do hardware físico. Entretanto, esta estratégia não visa o teste do software dependente do hardware em si, uma vez os mocks utilizados avançam apenas até a camada de software “hardware”.

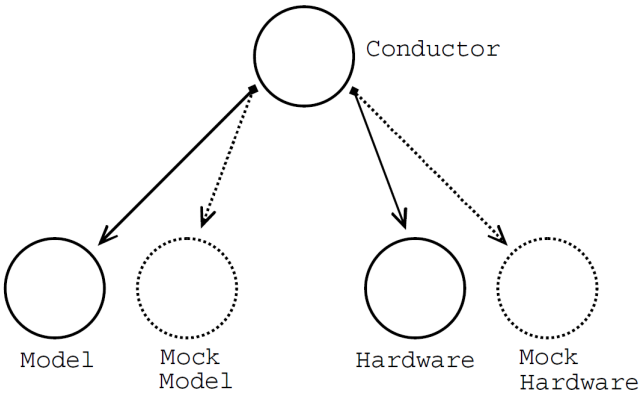


Figura 2.8: Padrão de desenvolvimento MCH (KARLESKY; BEREZA; ERICKSON, 2006).

2.3.3 Modelos

Entende-se um modelo como sendo uma representação abstrata de um objeto real e este conceito pode ser aplicado a diferentes áreas e com diferentes objetivos. Por exemplo, um modelo pode ser construído visando à compreensão do funcionamento do objeto estudado, como no caso de modelos matemáticos que modelam comportamentos físicos reais. Há também situações em que modelos são utilizados visando à redução de riscos ou custos, como em um teste de colisão de veículos, onde modelos de corpos humanos são utilizados no lugar de pessoas vivas e em testes de explosões nucleares, onde, tanto com relação a custos, quanto com relação a riscos, o uso de modelos é mais interessante do que aplicações reais.

No contexto desta dissertação, modelos têm por finalidade a substituição dos objetos modelados, no caso, os dispositivos de hardware, em um nível bem definido de abstração. Não se tem por objetivo a modelagem completa dos dispositivos de hardware, o que tornaria os modelos excessivamente complexos e caros, mas sim a sua modelagem sob o ponto de vista funcional em nível de interface de comunicação. Conforme mostrado na Figura 2.9, os modelos dos dispositivos de hardware construídos neste trabalho comunicam-se com as camadas de software dependente do hardware exatamente da mesma forma que os dispositivos de hardware reais, através de leituras e escritas em registradores especiais, mapeados em memória no microcontrolador. Com esta abordagem, os mesmos casos de teste utilizados no software com os modelos sendo executados poderão ser utilizados, sem qualquer alteração, no software executando na plataforma alvo, onde o hardware físico estará presente.

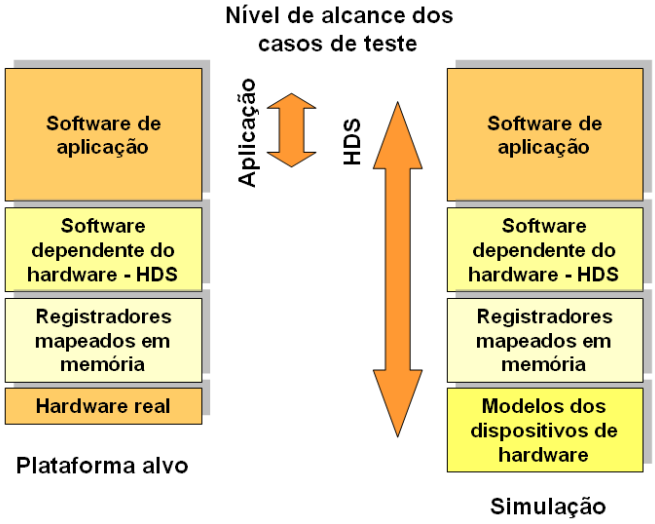


Figura 2.9: Aplicação dos modelos dos dispositivos de hardware e alcance dos casos de teste.

Fazendo intenso uso de modelos de dispositivos de hardware, em (ENGBLOM; GIRARD; VERNER, 2006) é apresentada uma ferramenta chamada Simics, onde a proposta é a simulação completa do hardware, do software de aplicação e do sistema operacional utilizado em um só ambiente. O hardware, neste caso, é composto pelo microprocessador central e todos os periféricos necessários e a simulação é feita em nível de instrução, a partir do código binário do software compilado, de forma síncrona e baseada em eventos, por razões de desempenho. Os autores destacam diversos benefícios do uso do ambiente simulado, tais como determinismo, execução em tempo acelerado e execução simultânea de múltiplos sistemas, como em uma rede (Figura 2.10).

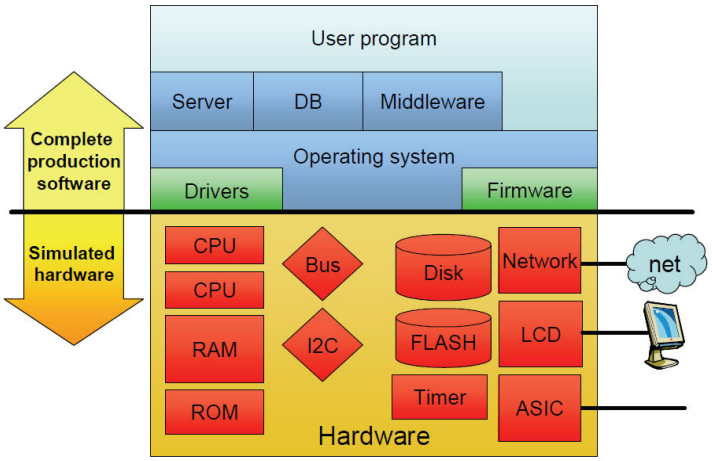


Figura 2.10: Ferramenta Simics (ENGBLOM; GIRARD; VERNER, 2006).

Diferentemente do método que será proposto nesta dissertação, onde os modelos dos dispositivos de hardware são modelados em arquivos fonte, construídos na mesma linguagem do software de aplicação, ferramentas como a Simics requerem modelos mais complexos, capazes de modelar o conjunto de instruções do microprocessador, assim como toda a sua arquitetura interna, de forma a permitir a simulação a partir do código binário compilado.

Um dos objetivos desta dissertação é mostrar que, mesmo fazendo a modelagem dos dispositivos de hardware em um nível ainda elevado de abstração e com modelos simples de serem construídos, uma grande cobertura de falhas de software, tanto no software de aplicação, quanto no software dependente do hardware pode ser obtida.

2.4 Ferramentas de Desenvolvimento e Teste de Software

A automatização, o quanto for possível, do processo de desenvolvimento e teste de software embarcado é uma necessidade por parte das empresas, uma vez que, tanto os tempos de desenvolvimento, quanto as margens de lucro, em geral, estão cada vez menores devido à intensa competição existente entre fabricantes. Isto faz com que a busca por ferramentas de desenvolvimento e teste, que facilitem etapas do processo de desenvolvimento seja constante. Com o uso adequado de tais ferramentas, pode-se multiplicar a produtividade de uma equipe de desenvolvimento, necessidade que, atualmente, poderá significar a permanência ou não da empresa no mercado.

2.4.1 Visualizadores de Código e Métricas de Qualidade de Software

Métricas de qualidade de software, em geral, são obtidas a partir da análise da estrutura interna do software. A análise de tais estruturas é fundamental, uma vez que dois softwares completamente distintos com relação à sua construção podem executar exatamente a mesma tarefa sob o ponto de vista funcional. Entretanto, uma correta organização da estrutura interna do software permitirá uma grande redução da probabilidade de existência de defeitos.

O uso de visualizadores eficientes de código, além de facilitar o desenvolvimento do software, permite uma melhor compreensão do que está sendo construído. Ferramentas modernas são capazes de, em poucos cliques de mouse, fornecer uma série de informações na forma de gráficos e relatórios a respeito do software em desenvolvimento, facilitando em muito a sua compreensão, principalmente quando o projeto está sendo analisado por novos desenvolvedores.

Recursos de visualização do código na forma de gráficos, onde se pode observar, de forma bastante clara, as dependências entre os módulos e funções, assim como o seu nível de acoplamento, são amplamente utilizados. Como exemplo de tais recursos, na Figura 2.11, é apresentada uma janela de informações a respeito da estrutura TSamples, criada internamente no software utilizado nesta dissertação (apresentado no capítulo 4). Com apenas um clique de mouse, o desenvolvedor descobre onde a estrutura está definida, seus campos internos com os correspondentes tipos de dados e em quais pontos do software ela está sendo utilizada. Ainda na Figura 2.11 uma função genérica pode ser automaticamente convertida em um diagrama de fluxo de controle. Enxergar a função na forma de um diagrama de fluxo de controle é bastante útil para a análise do seu comportamento, para a geração de documentação de projeto e para a análise visual da sua complexidade. O visualizador de código utilizado nestes exemplos é a ferramenta

Understand, onde está disponível uma série de outros recursos de visualização e edição de código não demonstrados nesta dissertação (SCITOOLS, 2010).

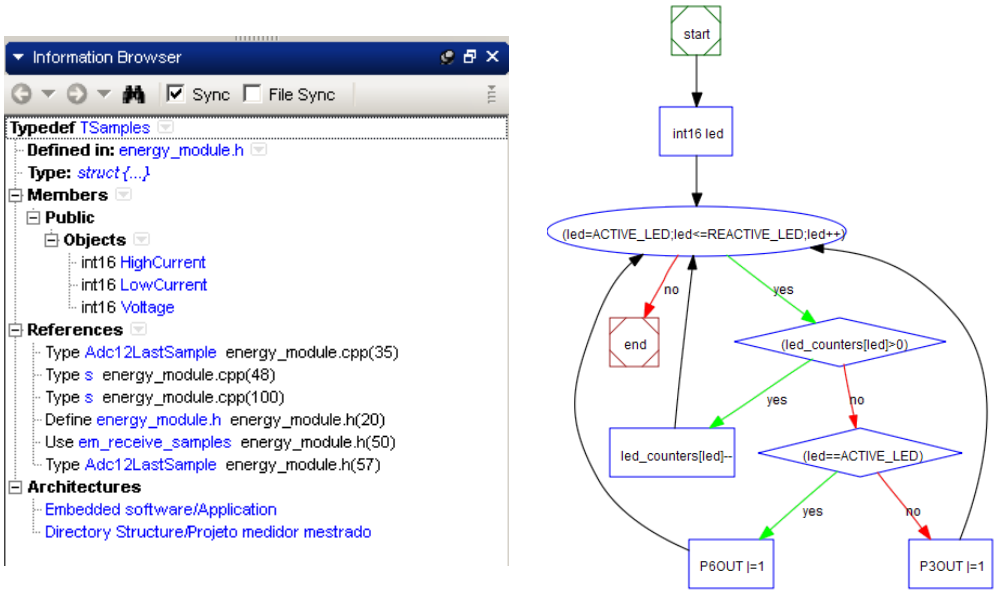


Figura 2.11: Janela de informações e diagrama de fluxo de controle obtidos automaticamente pela ferramenta Undertand (SCITOOLS, 2010).

Além de fornecerem recursos para a construção do código, visualizadores eficientes são capazes de gerar gráficos e relatórios contendo uma série de métricas de qualidade de software. Métricas de qualidade são basicamente utilizadas como uma forma de se medir uma série de características desejáveis e indesejáveis na forma como o software foi construído. A análise de métricas de qualidade de software não é capaz de apontar a existência de erros, mas é capaz de apontar trechos de código com maior probabilidade de ocorrência de erros, seja por complexidade exagerada, por muito acoplamento, má documentação, etc. A seguir são apresentadas algumas das principais métricas de qualidade de software utilizadas.

- Complexidade ciclomática de McCabe (MCCABE, 1976) – esta é uma métrica de complexidade do código gerado obtida pela contagem do número de caminhos independentes dentro de uma função. Sua obtenção é bastante simples, em um diagrama de fluxo de controle, conta-se o número de arcos, subtrai-se o número de nós e soma-se o número de componentes interligados (PEZZÉ; YOUNG, 2008). Como exemplo, a complexidade ciclomática da função representada no diagrama de fluxo de controle da Figura 2.11 é quatro. Em (ANDERSON, 2004) é exibida uma tabela com os riscos estimados conforme a complexidade ciclomática (Tabela 2.2). Nela, observa-se que uma função com complexidade ciclomática superior a 50 é considerada de altíssimo risco e impossível de ser testada.

- Complexidade ciclomática de McCabe modificada – variação do método de cálculo no qual sentenças do tipo switch-case são consideradas como sendo apenas um caminho, ao invés de se considerar um caminho para cada case (ANDERSON, 2004).
- Aninhamento (do inglês, *nesting*) – a medição do aninhamento, ou seja quantos níveis de escopos existem nas estruturas de controle do software pode ser utilizada como uma métrica de complexidade.
- Entradas e saídas – o número de entradas e saídas de uma função ou módulo de software pode revelar o seu nível de acoplamento. Uma função muito acoplada, seja pelo excessivo número de saídas ou por uma quantidade elevada de entradas, terá a sua testabilidade prejudicada, uma vez que a obtenção de uma cobertura adequada poderá requerer a execução de um número muito grande de casos de teste de software.
- Razão comentário/código – esta é uma métrica de qualidade que visa mostrar o quanto o código está bem comentado. Pode ser aplicada tanto para funções individuais quanto para arquivos inteiros do código fonte. Gerentes de projeto podem utilizar esta métrica como uma forma de medir o nível de comentário do código fonte desenvolvido.
- Linhas executáveis de código – o número de linhas executáveis de código, aliado a outros fatores de análise, pode ser utilizado para de certa forma, medir a produtividade de desenvolvedores. A obtenção de históricos de produtividade, para um mesmo segmento de aplicação, é fundamental para o dimensionamento, tanto em nível de tempo, quanto com relação a recursos, de projetos futuros.

Tabela 2.2: Análise de risco conforme a complexidade ciclomática (ANDERSON, 2004).

Complexidade Ciclomatica	**Avaliação de risco**
1-10	Uma função simples, sem maior risco.
11-20	Função relativamente complexa, risco moderado.
21-50	Função complexa, risco elevado.
Maior que 50	Software intestável, risco muito alto.

Há muitas outras métricas a respeito da estrutura do software que podem ser utilizadas para a verificação de sua qualidade, tais como: número de comentários, número de linhas executáveis, etc. De fato, é extremamente importante que os desenvolvedores façam análise das métricas de qualidade de software e reescrevam os módulos de software considerados críticos, visando a sua simplificação e conseqüente aumento de sua testabilidade. Para este tipo de tarefa, o uso de ferramentas automáticas de geração de métricas e análise de código é indispensável.

Como exemplo prático de aplicação desta ferramenta, na Figura 2.12 são mostrados dois diagramas de fluxo de controle de duas funções distintas, ambas extraídas de softwares reais de medidores eletrônicos de energia. Observa-se que mesmo a função

(a), de complexidade ciclomática igual a 10, já possui um certo grau de dificuldade de teste, entretanto, a função (b), com complexidade ciclomática igual a 114, é considerada intestável, dado o elevado número de casos de teste de software que seriam necessários para percorrer todos os seus caminhos e condições lógicas internas. Ao se deparar com este tipo de código, um desenvolvedor deveria, imediatamente, distribuir a tarefa em funções mais simples e com escopos mais bem definidos e fechados.

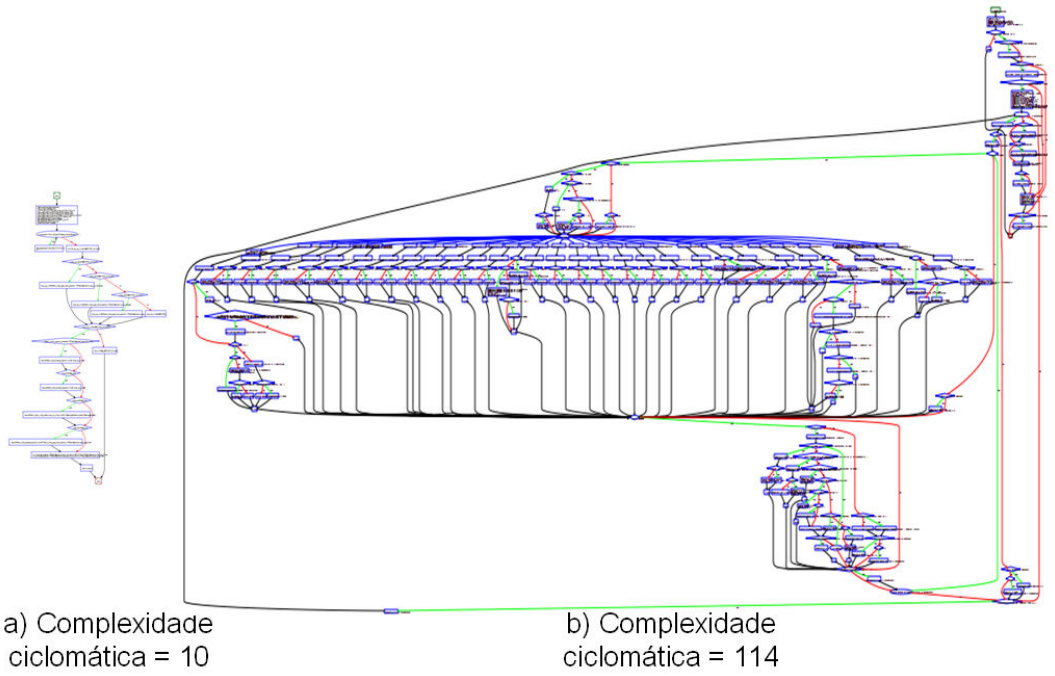


Figura 2.12: Diagrama de fluxo de controle de duas funções distintas e suas respectivas complexidades ciclomáticas.

2.4.2 Análise Estática de Código

Técnicas de análise estática de código obtêm informação a respeito do comportamento do software baseado na sua representação estática, diferentemente da análise dinâmica, que precisa executar o software para a sua verificação e que, por isto, requer o uso de casos de teste (CODE-SONAR, 2010).

Muitas das funções de analisadores estáticos de código estão hoje presentes nos modernos compiladores, tais como: geração de alarmes pelo uso de variáveis não inicializadas, variáveis criadas, mas não utilizadas, verificação de tipos de dados, etc. Entretanto, diferentemente de compiladores que verificam cada módulo de software individualmente, analisadores estáticos podem rastrear inconsistências e redundâncias entre diferentes módulos do software (PC-LINT, 2010).

Muitos dos erros encontrados por analisadores estáticos de código são, depois de descobertos, simples de serem corrigidos. Entretanto, quando um defeito não é descoberto, uma falha simples de programação pode permanecer despercebida por longos períodos e, dependendo do nível de severidade dos testes executados, apenas ser encontrada pelo usuário da aplicação.

Finalmente, o uso sistemático de uma boa ferramenta de análise estática, dá aos desenvolvedores independência, principalmente com relação aos recursos de análise

estática do compilador de código utilizado, que, dependendo do fabricante, podem ser mais ou menos rigorosos e avançados.

Na Figura 2.13, um trecho de código com um erro de programação típico da linguagem C é submetido à análise pela ferramenta PC-Lint (PC-LINT, 2010). Na instrução "if ((load_status & NO_ENERGY) = NO_ENERGY)", o operador de comparação lógica "= =" foi substituído equivocadamente pelo operador de atribuição "=". Muitos compiladores simplesmente permitiriam a atribuição sem qualquer tipo de mensagem e, se o requisito em questão não fosse testado por um caso de testes específico, o problema poderia passar despercebido. Após executado pela ferramenta PC-lint, observam-se as mensagens de erro e alarme geradas.

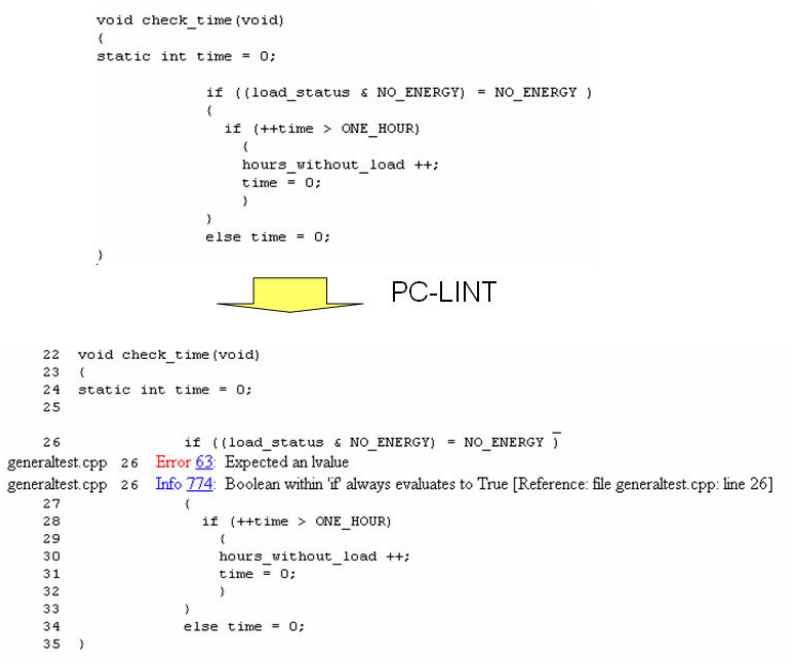


Figura 2.13: Trecho de código analisado por uma ferramenta de estática de código.

Dois outros analisadores estáticos de código com recursos similares ao PC-lint são o Code-Sonar (CODE-SONAR, 2010) e o Astree (ASTREE, 2010), ambos capazes de encontrar erros em tempo de execução, através da execução de sucessivas interações no código do software analisado.

2.4.3 Análise de Cobertura de Teste

As ferramentas de análise de cobertura de teste têm por objetivo a verificação, de forma automática, do alcance dos casos de teste executados. Conforme já mencionado nesta dissertação, não é possível afirmar que um código não executado está funcionando conforme a sua especificação. Além disto, é importante ressaltar que mesmo a execução dos casos de teste funcionais, que rastreiam os requisitos da aplicação, não é suficiente para a verificação completa do software, uma vez que não há garantia de execução de todas as suas estruturas internas. Apenas a análise de cobertura do teste será capaz de rastrear se todos os nós, laços ou condições lógicas do software em teste foram devidamente contemplados por ao menos um caso de teste.

Em geral, ferramentas de análise de cobertura de teste criam uma versão instrumentada do código a ser testado, de forma a verificar a execução de cada estrutura e expressão interna do código em teste. A geração deste código instrumentado é feita de forma automática e o usuário apenas tem acesso aos resultados desta execução, abordagem, por exemplo, utilizada pela ferramenta de análise de cobertura de teste Cantata (CANTATA, 2010). Importante observar que, quando aplicada a software embarcado e executada diretamente na plataforma alvo, a instrumentação do software poderá se tornar um aspecto crítico, dada a limitação de tamanho do código gerado imposta pelo compilador e pela arquitetura de memória da plataforma alvo. Além disto, requisitos de tempo real poderão vir a ser prejudicados, uma vez que o código instrumentado levará mais tempo para executar do que o código originalmente construído.

Uma outra forma de se efetuar a análise de cobertura é a utilizada pela ferramenta Coverage Validator (COVERAGE, 2010), utilizada nesta dissertação. Esta ferramenta não requer a geração de um código instrumentado, ao menos não de forma explícita. A ferramenta faz uso de uma configuração do compilador, onde parâmetros de depuração são inseridos no arquivo executável do código automaticamente, de forma que, em tempo de execução, mensagens de erro sejam geradas, mostrando a linha de código e a instrução em que a falha ocorreu. Uma vez que o formato como as informações de depuração são inseridas no arquivo executável é padronizada (formato COFF, do inglês *commom object file format*), segundo (COVERAGE, 2010), este software é compatível com um grande número de compiladores, como, por exemplo: Microsoft, Intel, Borland, MinGW, QtCreator e Metrowerks.

É possível configurar a ferramenta para contar o número de vezes que cada trecho do código foi executado, ou, simplesmente, sinalizar se o trecho foi ou não executado (modalidade mais eficiente com relação ao tempo de processamento). A ferramenta exibe os dados de cobertura de teste de uma forma bastante amigável, ora na forma de tabelas com relatórios sobre a cobertura obtida em cada módulo ou função, ora diretamente no código fonte, ressaltando os pontos cobertos, conforme exibido na Figura 2.14.

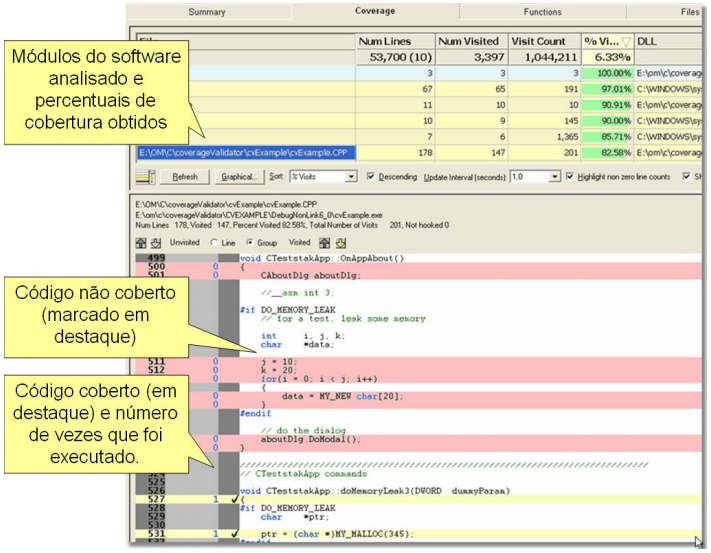


Figura 2.14: Tela do software Coverage Validator (COVERAGE, 2010).

2.4.4 Simuladores

Apesar dos ambientes de desenvolvimento de sistemas embarcados, usualmente, fornecerem simuladores para o conjunto de instruções do microcontrolador, em geral, tais simuladores não englobam os periféricos de hardware. Ou seja, sempre que uma instrução ou função do módulo depender de uma resposta de um periférico de hardware para seguir em frente, ou responder adequadamente, ela simplesmente irá travar ou responder de forma completamente inesperada. A razão do não fornecimento de tais simuladores por parte dos fabricantes de ferramentas de desenvolvimento talvez se deva ao foco no desenvolvimento e depuração de sistemas embarcados baseado no uso de emuladores.

Durante o desenvolvimento de software embarcado, a ausência de simulação dos periféricos de hardware limita bastante a utilidade destes ambientes de simulação, uma vez que se o software dependente do hardware não funcionar adequadamente, este mau funcionamento será inevitavelmente transferido às camadas de aplicação.

Este tipo de problema é ilustrado no diagrama exibido na Figura 2.15. Durante a simulação, um módulo de software de aplicação hipotético, responsável pelo armazenamento de dados persistentes, faz chamadas ao módulo de software dependente do hardware que controla a memória flash. A memória flash, por sua vez, é acessada por escritas e leituras em registradores mapeados em memória que não gerarão qualquer tipo de ação, nem tampouco resultarão em respostas às requisições do software, fazendo com que o software de aplicação executado em camadas superiores deixe de funcionar conforme o esperado. Devido a isto, com o objetivo de viabilizar o teste, desenvolvedores desacoplam completamente o software de aplicação das camadas inferiores, mais dependentes do hardware, fazendo uso intenso de stubs ou mocks. As limitações apresentadas são bastante sérias, uma vez que podem inviabilizar o uso de tais simuladores para grande parte das aplicações embarcadas.

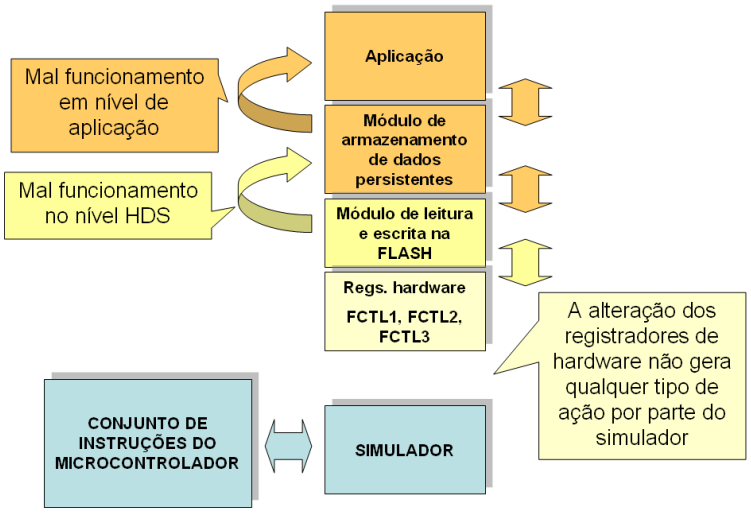


Figura 2.15: Simulação em nível de instrução e propagação de falhas.

Um ambiente de simulação ideal deve ser capaz de compreender o conjunto de instruções do microcontrolador, o protocolo de comunicação com os dispositivos periféricos de hardware e, além disto, ser capaz de gerar e inserir estímulos externos ao sistema embarcado em teste. Tais estímulos deverão ser gerados de forma livre via software, através de interfaces amigáveis e completamente customizáveis, permitindo, ainda, a observação das respostas obtidas em qualquer tempo, onde grandes quantidades de dados podem ser analisadas e registradas em arquivos para verificação futura.

2.4.5 Emuladores, Placas de Avaliação e Protótipos

Devido à intensa interdependência entre o software embarcado e o hardware, emuladores, placas de avaliação (fornecidas pelos fabricantes de microcontroladores) e protótipos são constantemente utilizados durante as etapas de desenvolvimento de sistemas embarcados. O objetivo principal desta abordagem seria lidar, desde as primeiras etapas de projeto, com um ambiente de desenvolvimento e teste que possua um comportamento o mais próximo o possível do que será verificado no sistema embarcado definitivo.

As principais vantagens do uso de tais recursos, dizem respeito às características de tempo real, onde, por exemplo, os períodos de interrupções e temporizadores são programados e verificados exatamente como no produto final, os tempos de execução das instruções são exatos e podem ser facilmente medidos, o tamanho do código gerado é real e o hardware criado pode ser avaliado no contexto em que ele será utilizado. A técnica de teste de software embarcado baseada na análise de interfaces (Figura 2.16), proposta em (SEO et. al, 2007), por exemplo, faz uso de emuladores para a execução dos casos de teste.

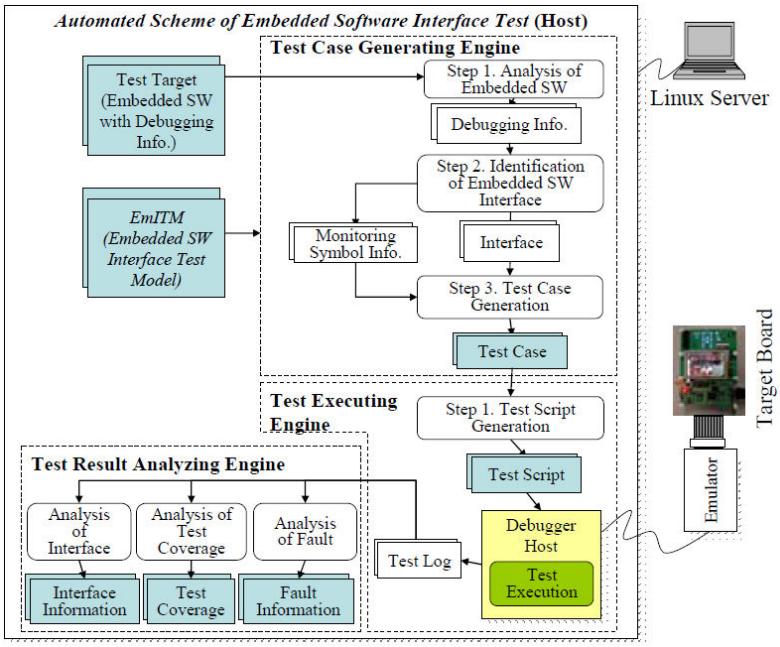


Figura 2.16: Um sistema automatizado de geração e teste de interfaces baseado em emulação (SEO et. al, 2007).

Entretanto, apesar dos benefícios citados, há também uma série de limitações. Os emuladores costumam funcionar pelo acesso direto às estruturas internas dos microcontroladores (apontador da memória de programa em execução, registradores, etc.), através de protocolos padronizados. A preocupação de se implantar este tipo de emulação, visando à redução do overhead de observação do código é antiga, como observado em (KOEHNEMANN; LINDQUIST, 1993). Porém, isto acarreta em algumas limitações. O número de *breakpoints*, por exemplo, que podem ser aplicados à execução do software, dependerá do tipo de microcontrolador utilizado. Tal limitação está diretamente associada aos recursos de hardware do microcontrolador. Microcontroladores mais caros e complexos possuem um número maior de recursos de emulação do que microcontroladores mais simples e baratos. Outra limitação interessante, diz respeito à capacidade de transferência de dados para os emuladores. Se uma aplicação precisa verificar, em tempo real, o conteúdo da memória do microcontrolador, este conteúdo precisará ser transferido ao ambiente de testes onde será analisado. Este tempo de transferência não será instantâneo e, muitas vezes, impedirá a análise de aplicações em tempo real.

Enquanto o uso de emuladores é bastante interessante para a execução de testes de unidade e de integração na plataforma alvo, o uso de protótipos será fundamental para os testes funcionais e de sistema, onde o equipamento será verificado na situação real de aplicação.

Em ambos os casos, quando comparado a um ambiente completo de simulação, mesmo com os recursos dos modernos emuladores atuais, muito se perde com relação à capacidade de observação e controle do que exatamente está ocorrendo no software embarcado. Outro aspecto diz respeito ao custo do teste, que poderá se tornar bastante elevado, caso haja dependência de uma estrutura muito avançada para tal. Isto se tornará evidente quando o software embarcado requerer, para a sua análise, a inserção de uma grande quantidade de estímulos externos, como no caso do software encontrado em medidores eletrônicos de energia, onde, para se executar em protótipos a grande parte dos testes, seria necessário um bom investimento em equipamentos avançados, capazes de estimular o protótipo de diferentes formas, com diversos níveis de tensão e corrente, por exemplo, ora inserindo conteúdo harmônico, ora gerando interrupções, etc.

2.5 Resumo e Conclusões

Neste capítulo, procurou-se apresentar alguns dos aspectos referentes ao teste de software, no contexto de software embarcado, relacionando-os, sempre que possível, a trabalhos publicados na recente literatura. A seqüência da apresentação dos conceitos teve, por objetivo, seguir uma lógica desde o problema da geração dos casos de teste até a sua aplicação, pelo uso de ferramentas específicas. A seguir, se faz uma descrição, de forma sucinta, do que foi apresentado, utilizando este material como base para, em seguida, uma contextualização com a proposta desta dissertação.

Foi visto que a geração de casos de teste através da análise dos requisitos do projeto requer a especificação formal de tais requisitos, de modo que sejam obtidos casos de teste de boa qualidade, tanto com respeito à sua eficácia, quanto com relação à sua abrangência – garantia de que todos os requisitos foram cobertos com ao menos um caso de teste. Apresentou-se, também, o UML como uma ferramenta bastante utilizada para a especificação de tais requisitos, tanto na forma de MSCs, quanto na forma de FSMs.

Com relação à geração de casos de teste pela análise de cobertura, foram explicadas as principais vantagens de tais métodos, que visam à garantia de que todas as estruturas internas do software foram testadas. Apresentou-se, brevemente, os métodos MC/DC e sua variação, chamado CACC, que têm por objetivo a garantia de que, em expressões lógicas, todas as condições lógicas independentes foram devidamente cobertas. Viu-se, também, que o método CACC pôde ser aplicado diretamente aos requisitos de projeto, obtendo-se casos de testes capazes de localizarem falhas não descobertas por casos de teste puramente funcionais. Finalmente, foram apresentados os problemas de priorização de casos de teste e as principais razões da não utilização de tais técnicas em software embarcados.

Na seqüência, as interfaces software-software e hardware-software foram apresentadas como uma importante fonte de casos de teste de software para sistemas embarcados. Foram mostrados trabalhos que visavam a definição de características importantes a serem testadas em tais interfaces e que justificassem a sua importância.

Com relação a cenários, a idéia foi ressaltar sua importância no agrupamento de casos de teste conforme características em comum, de forma a permitir uma padronização do software executor, o driver de testes. Mostrou-se, também, que, se bem planejados, é possível fazer com que um pequeno conjunto de cenários de teste obtenha uma grande cobertura de código.

Intensamente relacionado a esta dissertação, o conceito de dublês de teste foi mostrado, onde as definições de stubs, mocks e modelos, nem sempre utilizadas de forma clara, foram detalhadas. Finalmente, visando a execução dos testes propriamente ditos, ferramentas de teste de software comerciais e métodos de análise e depuração de software, como simuladores e emuladores foram discutidos.

O trabalho apresentado nesta dissertação faz intenso uso de dublês de teste, na forma de modelos, capazes de substituir o hardware durante a execução de um grande número de casos de teste.

Primeiramente, são gerados casos de teste a partir da análise dos requisitos do projeto (especificados, ainda, de forma informal). Estes casos de teste são então organizados em cenários e aplicados, um a um, por um software *drive*r, cuja execução é monitorada por uma ferramenta de análise de cobertura de teste. Em um segundo momento, os pontos de menor cobertura de teste são verificados e, caso tenham relação a requisitos válidos, novos casos de teste são gerados, com o objetivo de se ampliar a cobertura obtida. Tudo isto é executado em ambientes de desenvolvimento e teste voltados a software de aplicação. Nenhuma ferramenta voltada à plataforma alvo é utilizada até então.

São pontos chave da abordagem proposta a organização do software embarcado, onde as interfaces entre o software dependente do hardware e o software de aplicação são cuidadosamente projetadas, e a forma de construção dos modelos dos dispositivos de hardware, construídos de modo completamente natural, na mesma linguagem de programação do software em desenvolvimento.

Alguns dos trabalhos encontrados na literatura atual abordam a simulação total do ambiente de teste, como em (ENGBLOM; GIRARD; VERNER, 2006), outros abordam a substituição de partes do software em desenvolvimento por mocks, como em (KARLESKY; WILLIAMS, 2007).

No primeiro caso, o desenvolvedor se torna completamente dependente da plataforma de simulação, uma vez que os modelos construídos apenas poderão ser utilizados no ambiente comercial no qual eles foram projetados. Além disto, dado o nível de complexidade de tais modelos, caso o modelo desejado não esteja disponível, um grande esforço de engenharia se fará necessário, seja por parte do interessado no modelo, seja pelo fabricante da ferramenta. Finalmente, uma vez que, em algum momento, o software embarcado deverá ser executado e testado no hardware final, por que não deixar os casos de teste que requeiram um maior nível de detalhes, seja pela arquitetura, seja por requisitos de tempo real, para esta etapa?

A segunda abordagem tem por objetivo a substituição de partes do software por mocks, entretanto, conforme dito no próprio trabalho, a idéia não é a modelagem de dispositivos de hardware, mas das interfaces que acessam o hardware, de forma que o hardware não seja necessário para grande parte dos testes. A idéia está alinhada com a proposta desta dissertação, entretanto, ela pressupõe que o hardware já tenha sido testado. Nesta dissertação se propõe o uso de modelos para permitir o teste, inclusive, do hardware.

Finalmente, um dos principais aspectos que tornam o sucesso da abordagem desta dissertação possível diz respeito ao fato de que os microcontroladores modernos dispõem de uma crescente capacidade de armazenamento de código, que vem tornando as aplicações embarcadas cada vez mais complexas. Entretanto, na contramão deste processo, particularidades dos ambientes de desenvolvimento de software voltado às plataformas alvo e a não compreensão do software dependente do hardware, impedem que estas novas e complexas aplicações embarcadas sejam testadas e compiladas com ferramentas de teste de software de alto nível, amplamente disponíveis no mercado. Ainda, os ambientes de desenvolvimento voltados às plataformas alvo não estão completamente amadurecidos com relação ao fornecimento de recursos de teste de software, conforme mostrado pela ausência de simuladores capazes de simular o hardware periférico e de recursos de análise de cobertura de teste, por exemplo.

3 A METODOLOGIA DE DESENVOLVIMENTO E TESTE DE SOFTWARE EMBARCADO

Uma das idéias deste trabalho é propor uma abordagem de desenvolvimento que não necessite do aprendizado de novas técnicas ou ferramentas, além daquelas que os desenvolvedores já estão acostumados a utilizar. Entretanto, apesar do pequeno esforço adicional, os ganhos obtidos são muitos, principalmente com relação à compatibilidade de software, facilidade de depuração e facilidade da execução dos testes. Tudo isto, visando um software embarcado mais bem testado e, consequentemente, de maior qualidade.

Pretende-se a construção de um software embarcado único, capaz de ser executado tanto na plataforma alvo, quanto em ambientes de desenvolvimento de software de aplicação, conforme exibido na Figura 3.1. Quando compilado e executado em um ambiente de desenvolvimento de software de aplicação, os modelos dos dispositivos de hardware estarão ativos, gerando respostas para o software dependente do hardware e permitindo a execução da aplicação e dos casos de teste gerados. Ao ser executado na plataforma alvo, os modelos serão automaticamente ignorados pelo compilador, permitindo que o hardware físico interaja com as camadas de software.

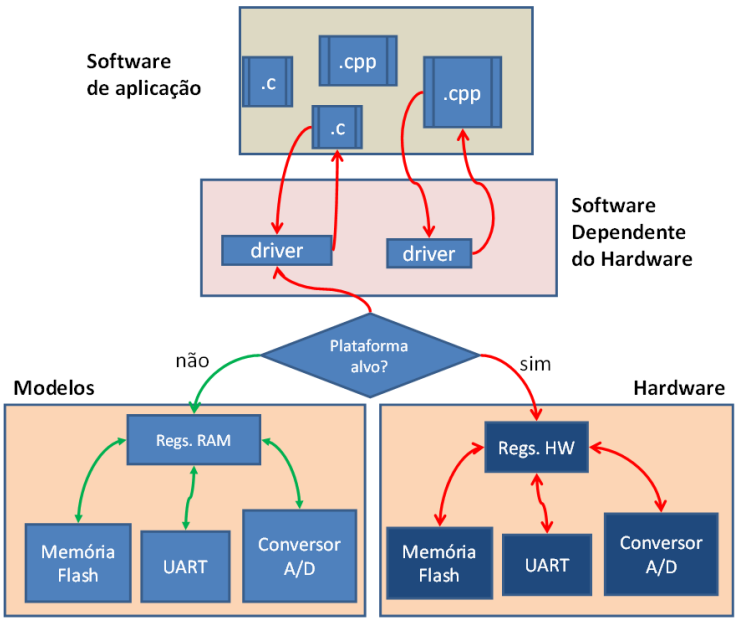


Figura 3.1: Visão geral do software criado.

O que será apresentado, então, é uma forma sistemática de se construir e organizar o software embarcado, de forma que se possa efetuar a grande maioria dos testes, tanto no software de aplicação, quanto no software dependente do hardware, fora do ambiente de desenvolvimento voltado para a plataforma alvo. A idéia é, com isto, se eliminar uma série de restrições impostas por tais ambientes de desenvolvimento, tais como:

- Incompatibilidade do software embarcado com ferramentas comerciais de teste de software;
- Impossibilidade de se testar, em simulação, o software dependente do hardware;
- Impossibilidade de se construir drivers de teste complexos, dadas as limitações do ambiente de desenvolvimento, tais como, limitação no tamanho de código e interfaces para a geração de relatórios;
- Limitação nos tempos de execução dos casos de teste dependentes do hardware, assim como no gerenciamento dos resultados, uma vez que precisam ser executados em emuladores (longos tempos de carga e descarga dos casos de teste);

A seguir, serão apresentados os principais aspectos da execução desta metodologia, envolvendo o detalhamento da construção dos modelos dos dispositivos de hardware e como eles são integrados ao software embarcado em construção.

3.1 Interface de Acesso ao Hardware

Dentro de um microcontrolador, os dispositivos de hardware são periféricos usualmente acessados por barramentos internos de comunicação e controlados através de leituras e escritas a registradores mapeados em memória. Desta forma, o software necessário para acessar determinado periférico se torna tão simples quanto uma seqüência de leituras e escritas nestas regiões específicas onde estão mapeados estes registradores.

Diz-se que estes registradores são mapeados em memória por que eles não estão fisicamente na memória interna do microcontrolador. Fisicamente, eles estão nos dispositivos de hardware aos quais eles pertencem. O que ocorre é que eles são endereçados pelo mesmo barramento de endereçamento que acessa a memória interna. Desta forma, sempre que estas regiões de memória forem lidas ou escritas, os dados são transmitidos para os registradores e não para a memória propriamente dita.

Por exemplo, a memória flash interna do microcontrolador MSP430F5438 (MSP430, 2010) pode ser escrita e apagada através de apenas três registradores de 16 bits, mapeados em memória, chamados FCTL1, FCTL3 e FCTL4. Observa-se que apenas operações de escrita e de apagamento são executadas através destes registradores, pois requerem intervenção do hardware controlador da memória flash. Uma vez que os bancos de memória flash também estão mapeados em memória e são endereçados pelo mesmo barramento de endereçamento da RAM interna, operações de leitura são efetuadas de forma transparente por instruções simples de acesso (Figura 3.2).

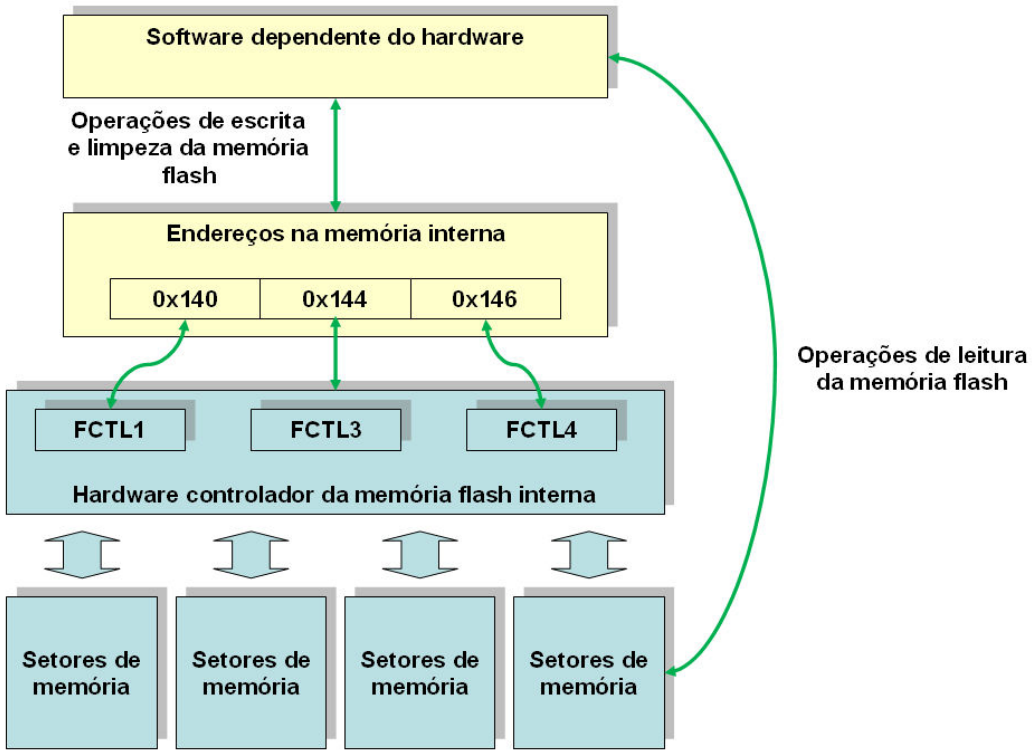


Figura 3.2: Acesso aos registradores que controlam o hardware.

Com relação à compatibilidade entre um software embarcado e um software de aplicação, os problemas começam na forma como estes registradores de acesso ao hardware são definidos pelos compiladores, que é diferente dependendo do fabricante. Na linguagem C, por exemplo, uma linguagem amplamente utilizada para o desenvolvimento de software embarcado, não existe uma forma padronizada de se forçar o endereço de uma variável diretamente no código fonte. Isto se deve ao fato de que, na época em que esta linguagem foi criada, o objetivo era justamente o oposto: se desejava afastar o programador do hardware e dos detalhes de mais baixo nível, como endereços, por exemplo. Então, enquanto o posicionamento de variáveis em regiões fixas da memória era amplamente utilizado em linguagens Assembly, em C isto não era sequer permitido. A idéia era deixar o compilador decidir onde as variáveis seriam posicionadas, liberando os programadores para o projeto dos algoritmos e das soluções dos problemas a serem resolvidos.

Esta lacuna na padronização da linguagem C fez com que os fabricantes de compiladores voltados a microcontroladores tivessem liberdade para, cada um da sua forma, decidir como seria feito o posicionamento destas variáveis nos endereços específicos. Olhando-se os arquivos de inclusão do compilador IAR (IAR, 2010), observam-se as estruturas mostradas na Figura 3.3, onde um operador específico para esta função foi criado. Outra abordagem para a mesma função é a utilizada pela ferramenta Code Composer (CODE COMPOSER, 2010). Nela, as variáveis criadas são declaradas como externas e o seu endereço específico apenas será definido em um arquivo passado para o linker, também exibido na Figura 3.3.

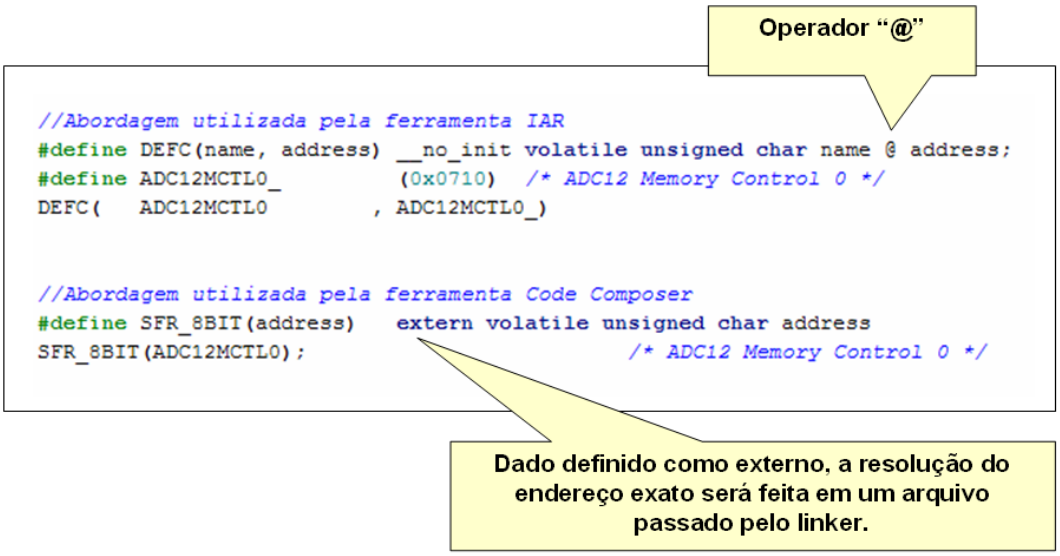


Figura 3.3: Abordagens utilizadas para o posicionamento dos registradores de acesso ao hardware nos locais específicos.

O problema deste tipo de abordagem é que, para se compilar os códigos fonte de um software embarcado em um compilador diferente do qual ele foi desenvolvido, será necessária uma adaptação destes arquivos de inclusão, uma vez que dificilmente este novo compilador será compatível com a abordagem escolhida pelo compilador nativo. O mesmo deverá ser feito quando a abordagem for através do linker, uma vez que dificilmente haverá compatibilidade.

3.2 Memória Compartilhada e Compilação Condicional

Conforme mostrado na Figura 3.1, tem-se por objetivo a construção de um software que possa ser livremente compilado, tanto para a plataforma alvo, quanto para o ambiente de software de aplicação, sem a necessidade de manterem-se duas versões distintas do mesmo software. De forma mais prática, isto é obtido pela inclusão de um arquivo global de definição de tipos de dados, que possui uma diretiva de compilação, onde o desenvolvedor define se quer compilar o código para a plataforma alvo ou para o ambiente de aplicação.

Conforme mostrado na Figura 3.4, dois códigos podem ser compilados, conforme a configuração feita no arquivo "Geral.h". Ao tornar ativa a definição "HIGH_LEVEL", o software incluirá automaticamente um arquivo de definições para o ambiente de aplicação, onde todos os registradores de hardware e tipos de dados utilizados pelo software embarcado são devidamente mapeados. Além disto, a ativação dos modelos fará com que todos os arquivos de código dos modelos se tornem ativos e sejam incluídos na compilação do projeto.

Caso o objetivo seja a execução do código na plataforma alvo, basta que a diretiva "HIGH_LEVEL" seja indefinida. Isto fará com que os modelos se tornem desabilitados e, automaticamente, excluídos do código. Ainda, com a desativação dos modelos, o

arquivo de inclusão que mapeia os registradores de hardware para a plataforma alvo é incluído no projeto.

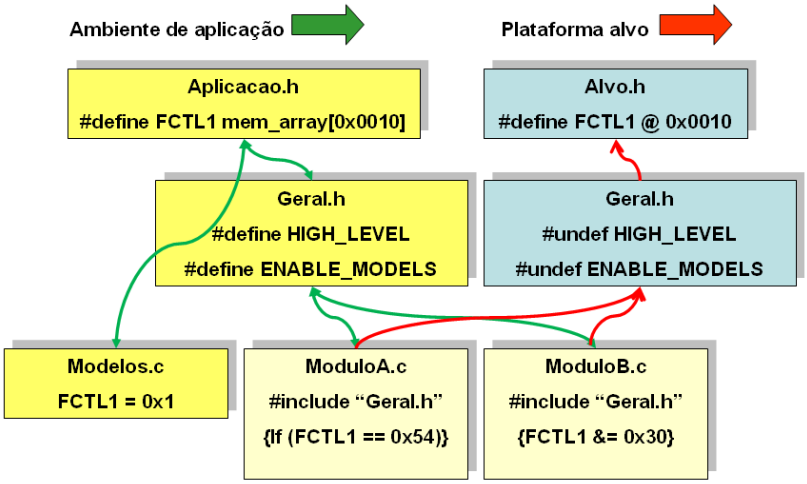


Figura 3.4: Organização dos arquivos para a compilação condicional.

Com o objetivo de permitir a compilação do software embarcado em um compilador para software de aplicação de forma que os registradores de hardware pudessem ser utilizados, a abordagem seguida foi a criação de um array representando a memória interna do microcontrolador. Este array é criado dentro do software driver de testes e é visível pelos modelos dos dispositivos de hardware e pelo software embarcado (Figura 3.5). Observa-se que todos os registradores de hardware são então mapeados para endereços específicos deste array, através de um arquivo de inclusão, exatamente da mesma forma que, na plataforma alvo, estes registradores são mapeados para a memória interna do microcontrolador.

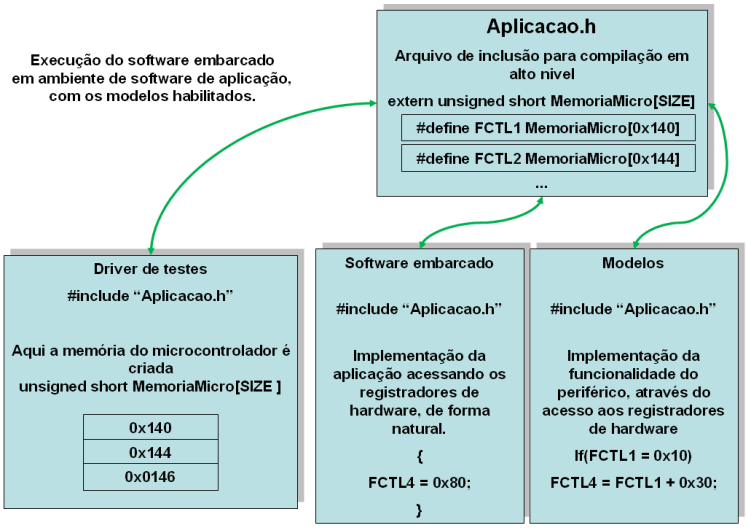


Figura 3.5: Representação da memória do microcontrolador dentro do software driver.

3.3 Nível de Abstração e Protocolo de Comunicação

Pretende-se que os modelos dos dispositivos de hardware se comportem como o hardware com relação à sua funcionalidade apenas. Não é a intenção deste trabalho a construção de modelos que respeitem tempos de execução e execução em paralelo, por exemplo, tais modelos se tornariam extremamente complexos e precisariam ser construídos com uma linguagem específica para tal. SystemC (SYSTEMC, 2010) seria uma alternativa.

Entretanto, mesmo que tais modelos complexos dos dispositivos de hardware estivessem disponíveis, uma das ideias desta dissertação é mostrar que eles não seriam capazes de cobrir um número tão maior de casos de teste que justificassem o seu esforço de construção. Além disto, uma vez que a validação do projeto sempre será feita com o hardware final, considera-se mais simples deixar os testes que requeiram avaliações críticas sob o aspecto de tempo real para esta etapa. Ainda, diferentemente do que se poder-se-ia imaginar, a construção dos modelos dos dispositivos de hardware não requer, por parte dos desenvolvedores, um estudo maior de tais dispositivos do que seria necessário para a construção do software dependente do hardware pura e simplesmente.

Conforme mostrado na Figura 3.2, o software embarcado se comunica com os dispositivos de hardware através de leituras e escritas a registradores específicos. Na Figura 3.5, mostra-se que tais registradores são mapeados para a memória interna do microcontrolador, quando o software é compilado para a plataforma alvo e, mapeados para um array global, quando o software é compilado em um ambiente de desenvolvimento de software de aplicação. Desta forma, os modelos podem ser construídos da mesma forma que o software embarcado, através de acessos a estes mesmos registradores.

Para que os modelos sejam executados, uma vez que não há qualquer tipo de paralelismo, o software embarcado deverá chamar uma função específica para tal. Na construção de cada modelo, esta função de execução é declarada global e se torna disponível ao software embarcado e ao driver de testes quando os modelos estão habilitados (Figura 3.6). Quando os modelos estão desabilitados, as chamadas a estas funções dentro do software embarcado são automaticamente substituídas por linhas em branco, de forma que nenhuma alteração no software embarcado seja necessária.

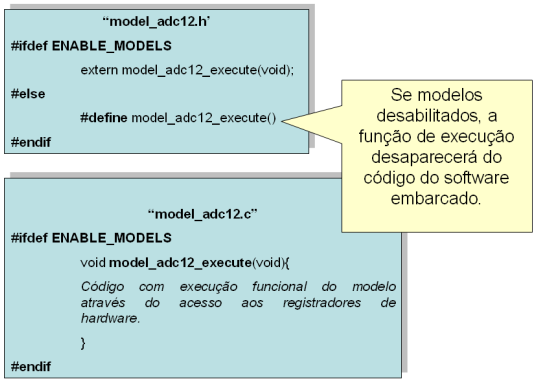


Figura 3.6: Definição da função de execução dos modelos.

A construção das funções de execução dos modelos é feita da mesma forma que o software embarcado que acessa o hardware, através da leitura das especificações da funcionalidade dos dispositivos de hardware, disponibilizadas pelos fabricantes. Em geral, os fabricantes fornecem uma boa documentação a respeito do funcionamento do hardware, com explicações detalhadas dos recursos disponíveis, tabelas onde cada campo dos registradores tem sua funcionalidade explicada, máquinas de estados finitos e diagramas que exibem as conexões internas dos dispositivos de hardware e sua relação com os registradores de hardware (Figura 3.7).

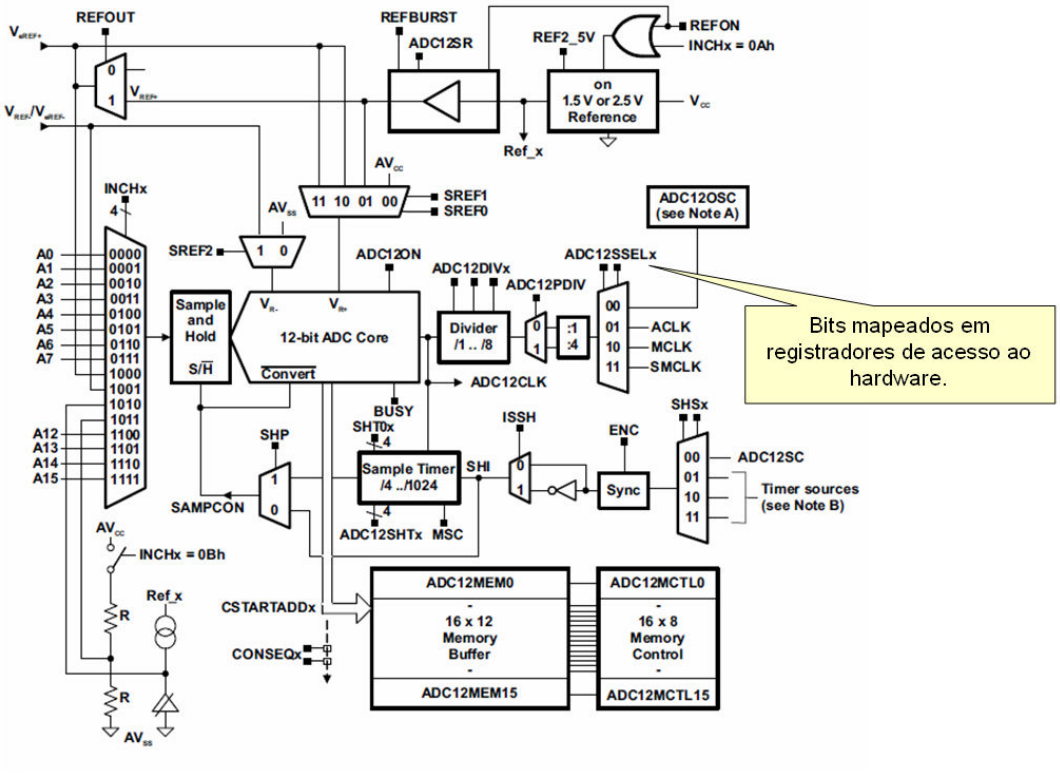


Figura 3.7: Exemplo de diagrama de conexão de um dispositivo de hardware, conversor A/D de 12 bits (SLAU208, 2010).

É importante salientar que a leitura e análise deste tipo de documentação é obrigatória para os desenvolvedores de software embarcado, sempre que se está construindo software dependente do hardware. Desta forma, a construção dos modelos dos dispositivos não requer nenhum esforço adicional de pesquisa.

Outro aspecto importante diz respeito a quais características dos dispositivos de hardware serão modeladas. Visando reduzir o esforço de codificação dos modelos, defende-se a construção destes sob demanda, ou seja, verificam-se no projeto quais funcionalidades dos dispositivos de hardware serão utilizadas e apenas a execução destas funcionalidades é modelada. Por exemplo, o dispositivo de hardware ADC pode ser programado para agir de dois modos distintos. Em um deles, os dados dos canais são adquiridos um a um, em outro, os dados são adquiridos em uma seqüência programada pelo desenvolvedor. Caso o modo de uso deste dispositivo seja determinado, não é

necessária a construção imediata do modelo dando suporte a ambos os modos de funcionamento.

Como exemplo de construção do modelo de um dispositivo de hardware, será analisado o modelo criado para um conversor A/D de 12 bits dos microcontroladores da família MSP430 (MSP430, 2010), fabricados pela Texas Instruments (TEXAS, 2010). O processo de construção do modelo pode ser inicialmente dividido em três etapas básicas:

- Análise da documentação técnica disponibilizada pelo fabricante: neste ponto o material técnico é lido e as principais características do dispositivo são entendidas. No caso do conversor A/D, neste ponto são avaliados os modos de aquisição disponíveis, controle de alimentação do dispositivo, modo de geração de referência, etc;
- Análise dos requisitos de projeto: a partir da especificação dos requisitos do projeto a ser desenvolvido, são levantados quais modos de funcionamento do dispositivo de hardware serão modelados. No caso do A/D, por exemplo, pode-se optar pelo modo de aquisição de canais em seqüência ou pelo modo individual de aquisição. Este tipo de escolha pode ser feita sempre que o dispositivo de hardware possuir máquinas de estados independentes para diferentes características;
- Construção do código do modelo: seguindo a forma de estruturação de arquivos mostrada na Figura 3.6, são gerados inicialmente dois arquivos básicos: o quê contém a definição da função de execução, no caso, model_adc12_execute() e o quê contém as declarações exportadas. Ambos sujeitos às diretivas de habilitação. Inicia-se, então a construção do software que implementará a funcionalidade.

No exemplo do ADC, este dispositivo pode ser completamente acessado através de quatro tipos diferentes de registradores de hardware, organizados conforme as funcionalidades que controlam e simplificados na Tabela 3.1.

Tabela 3.1: Registradores de configuração do conversor A/D (SLAU208, 2010).

Grupo	Itens	Funcionalidades
Controle	ADC12CTL0 até ADC12CTL2	Controle e configuração do dispositivo. Exemplo: tempo de amostragem, seleção do modo de funcionamento, estado da alimentação, tensão de referência, geração da referência, habilitação e início de conversão, etc.
Gerenciamento de interrupções	ADC12IFG, ADC12IE, ADC12IV	São associados às interrupções do ADC, habilitar ou desabilitar interrupções e informar interrupções pendentes.
Resultados das aquisições	ADC12MEM0 até ADC12MEM15	Resultados das aquisições, salvas como números inteiros sem sinal ou no formato complemento de dois.
Aquisição e referência	ADC12MCTL0 até ADC12MCTL15	Canal analógico e os tipos de referência utilizados para a aquisição.

Ainda conforme a Tabela 3.1, os registradores de hardware são mapeados para endereços da memória interna do microcontrolador (Figura 3.2). Estes registradores são usualmente comparados e escritos através do uso de constantes pré-definidas (máscaras) nos arquivos “Alvo.h” ou “Aplicação.h” (Figura 3.4). Isto é feito de forma a simplificar a codificação, aumentando a sua legibilidade. Estas constantes são as mesmas para ambas as compilações, uma vez que dizem respeito à localização dos bits de controle dentro dos registradores de hardware.

De modo geral, é possível se pensar em uma estrutura genérica para a construção do código dos modelos. Sugere-se:

- Análise de impedimentos: verificar se o dispositivo esta apto a funcionar, através da configuração atual;
- Seleção da funcionalidade: decidir em qual modo de operação o dispositivo se encontra;
- Construção da funcionalidade: construir o comportamento do modo de operação configurado;
- Análise de impedimentos: novamente, mas agora dentro da funcionalidade específica, verificam-se situações que possam impedir ou prejudicar o funcionamento adequado.

Desta forma, ter-se-ia uma estrutura padrão para o código do modelo, conforme exibido no diagrama da Figura 3.8.

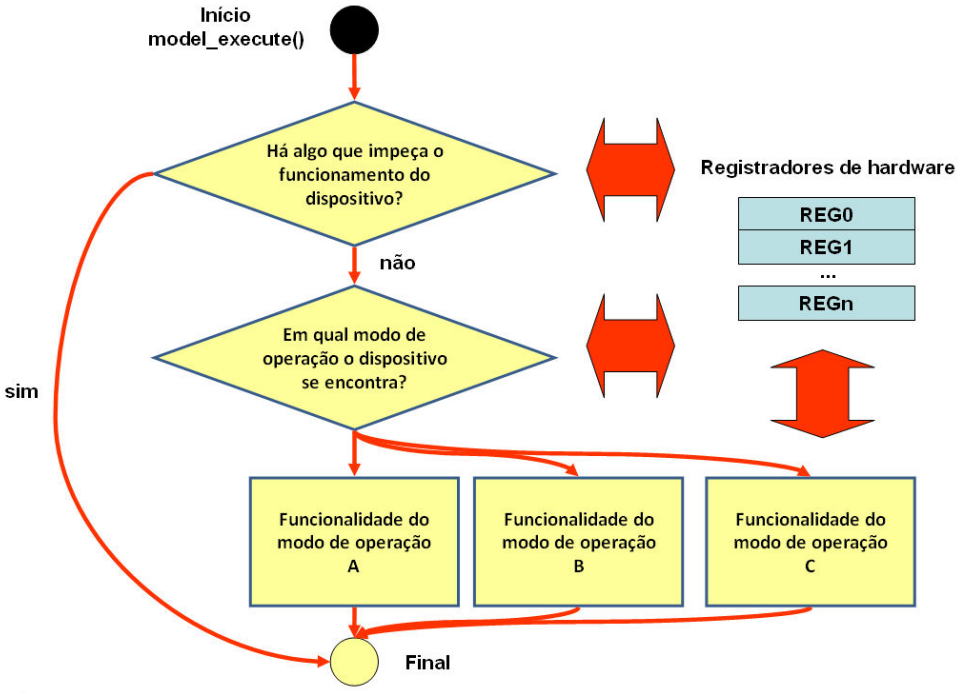


Figura 3.8: Estrutura interna básica de um modelo de dispositivo de hardware.

Ainda seguindo o exemplo do modelo do conversor A/D, como restrições ao seu funcionamento, podem ser citadas:

- De modo geral, o estado do bit responsável pelo controle da sua alimentação (mapeado em um dos registradores de controle da Tabela 3.1);
- Dentro das funcionalidades específicas, o controle das tensões de referência e as configurações dos pinos de entrada e saída. No diagrama da Figura 3.7, observa-se que os bits SREF0 e SREF1 controlam a origem da referência, que poderá ser a tensão de alimentação analógica (AVCC), a referência gerada internamente (2.5 V ou 1.25 V) ou um sinal de referência de origem externa (pino VEREF+). Da mesma forma, as configurações dos pinos de entrada e saída irão afetar o funcionamento do dispositivo.

A idéia da análise é simples: se o bit de alimentação não estiver ligado, o modelo não deverá executar, pois será como se o ADC não estivesse energizado. Durante a construção da funcionalidade, caso a referência não esteja corretamente configurada, pode-se forçar o conversor A/D a sempre resultar um sinal fixo, indicando a má configuração. Da mesma forma, um sinal espúrio pode ser gerado quando o software embarcado estiver tentando adquirir um sinal analógico proveniente de um pino não configurado como entrada do ADC.

As restrições à execução do modelo funcionam, na verdade, como uma série de especificações da forma como o dispositivo deverá ser configurado. Esta característica torna os modelos uma importante ferramenta de depuração do código, como será visto mais adiante neste capítulo. Além disto, pode-se configurar os modelos para gerarem mensagens de erro automáticas quando uma configuração notavelmente equivocada estiver sendo testada.

De forma mais prática, a Figura 3.9 exibe um diagrama do funcionamento do modelo do conversor A/D, onde alguns dos controles internos são mostrados. Observa-se que, assim que a execução do modelo inicia, são testadas configurações que deverão obrigatoriamente estar ajustadas para que o hardware esteja em funcionamento. No modelo do conversor A/D, o bit de estado de alimentação do dispositivo é verificado. Caso este bit não esteja ligado, o modelo considera que o conversor A/D está desligado e nenhuma ação é executada.

É importante salientar que a forma de construção do código do modelo é exatamente a mesma que seria utilizada para se verificar esta funcionalidade no código do software embarcado, uma vez que ambos utilizam os mesmos arquivos de definições. O registrador ADC12CTL0 é um registrador de controle (Tabela 3.1) e ADC12ON é uma constante onde apenas o bit referente ao controle de alimentação está ligado (uma máscara de bits). Da mesma forma, testa-se se o conversor A/D está configurado no modo de aquisição única, neste caso, o registrador de controle testado é o ADC12CTL1 e CONSEQ0 e CONSEQ1 são também máscaras de bits.

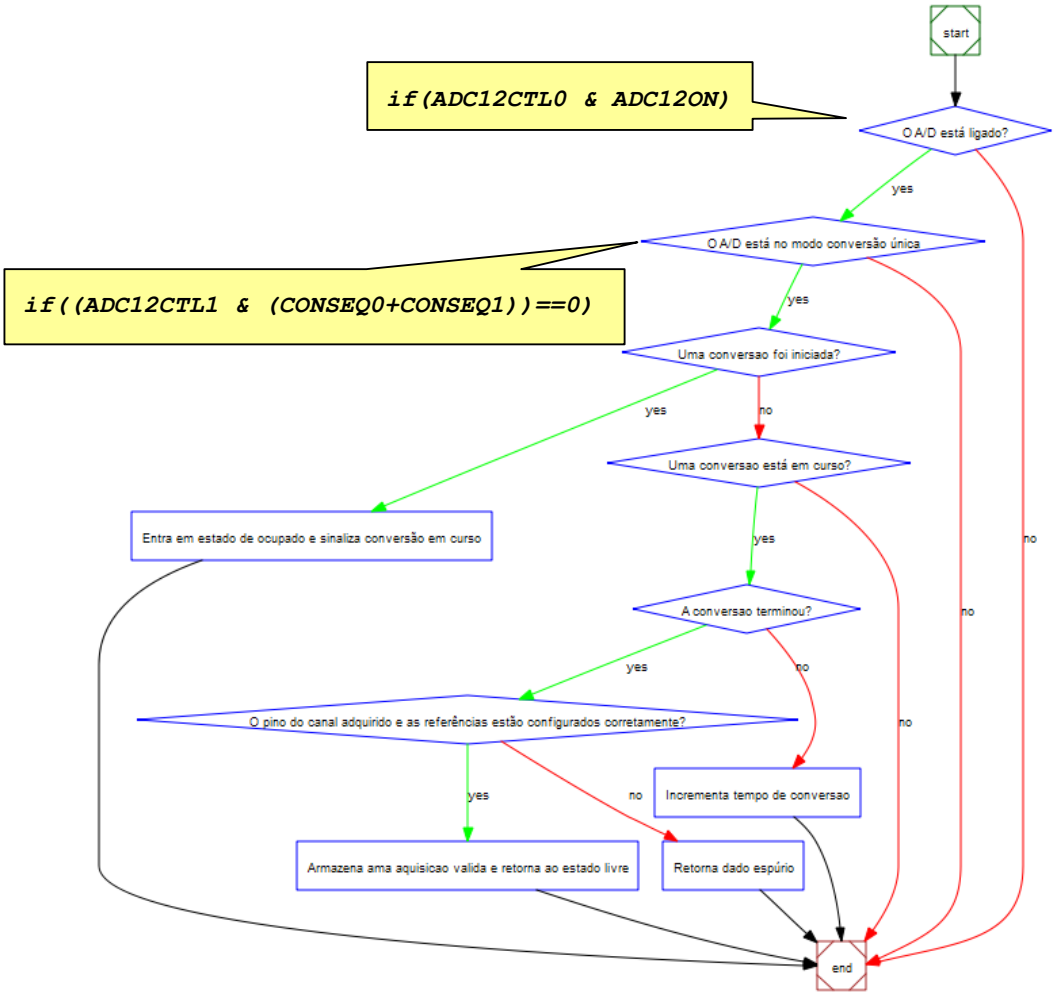


Figura 3.9: Diagrama resumido de funcionamento do modelo do conversor A/D.

3.4 Expansão da Funcionalidade dos Modelos

Um aspecto fundamental na construção dos modelos diz respeito à sua capacidade de simular o comportamento não só do dispositivo modelado, mas também do ambiente no qual o software está sendo executado. Este é um aspecto extremamente desejável no teste de software embarcado, onde o funcionamento do software pode se tornar completamente dependente de parâmetros do contexto onde ele está sendo executado. Por exemplo, um algoritmo de medição de energia que será utilizado em um medidor eletrônico de energia, para ser testado, precisará receber uma série de formas de onda de tensões e correntes como entrada. Estes sinais deverão ser gerados nas mais diferentes condições, onde serão variadas as amplitudes e as defasagens, assim como o seu conteúdo harmônico, por exemplo.

Diferentemente do hardware físico, que precisa ser estimulado através de equipamentos e conexões complexos, caros e limitados, os modelos dos dispositivos de

hardware podem ter a sua funcionalidade expandida livremente com muito pouco esforço. Os recursos adicionais ficam a critério da aplicação e dos requisitos que precisarão ser testados.

Finalmente, o controle sobre a aplicação de tais recursos pode ser dado completamente ao software que estará executando os testes, de forma que os resultados fornecidos pelos modelos dependerão do tipo de teste em execução. No trabalho desenvolvido para esta dissertação, o modelo do conversor A/D, por exemplo, foi construído de forma que os sinais convertidos por cada canal analógico configurado no modelo sejam requisitados através de uma função externa, que é construída dentro do software driver (Figura 3.10).

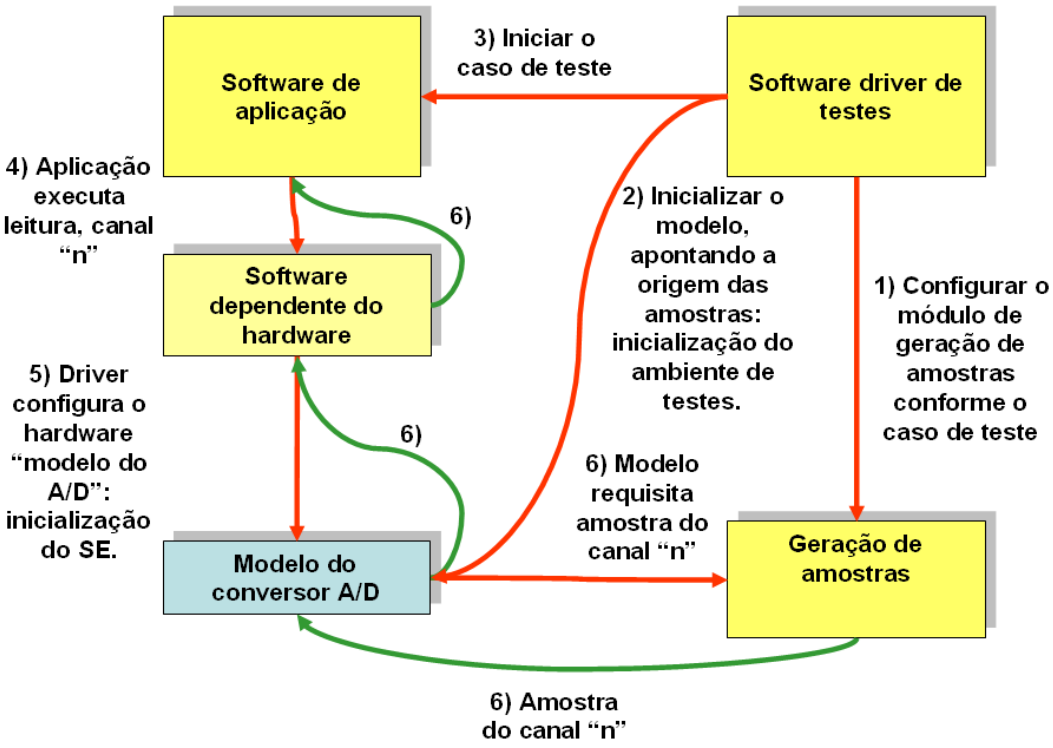


Figura 3.10: Controle sobre a geração de amostras do modelo do conversor A/D.

Outro tipo de expansão de funcionalidade diz respeito à sinalização de status e falhas. Há erros que são evidentes e que podem ser imediatamente notificados pelos modelos sem que um caso de testes específico para a sua detecção seja desenvolvido. Em geral, os microcontroladores possuem pinos de entrada e saída com múltiplas funcionalidades, estes pinos podem ser configurados para serem uma entrada ou saída de algum periférico de hardware, por exemplo, uma entrada de um canal analógico de um conversor A/D ou uma saída analógica de um conversor D/A, ou podem ser configurados como saídas ou entradas digitais. Sempre que o software dependente do hardware tentar executar uma operação notadamente inviável, o modelo do dispositivo poderá ser configurado para gerar uma mensagem de alerta, direcionada para um dispositivo de saída, também configurável (um arquivo ou uma janela de mensagens, por exemplo).

São muitas as situações de falha que poderão ser automaticamente diagnosticadas pelos modelos sem que um caso específico de testes tenha que ser gerado. A seguir, citam-se alguns exemplos.

- Uso indevido de pinos de entrada e saída: um software dependente do hardware está ligando e desligando um pino digital que está configurado como alta-impedância, ou como entrada. Analogamente, o software pode estar lendo o estado de um pino configurado como saída;
- Uso indevido de pinos de acesso aos dispositivos de hardware: o modelo de um dispositivo de hardware tenta acessar pinos que não estão configurados para serem utilizados como tal. Por exemplo, um conversor D/A está enviando dados para um pino que está configurado como entrada ou saída digital. O módulo de geração de referência de tensão está configurado para gerar a referência em um pino configurado como entrada e o conversor A/D está tentando utilizar esta referência;
- Má configuração de periféricos: um periférico de hardware desligado está sendo acessado para leituras ou escritas;
- Mau uso de registradores de acesso ao hardware: o software está escrevendo errado em um registrador específico. Por exemplo, alguns registradores da memória flash devem ser acessados através de senhas, que são constantes de valor conhecido que deverão ser escritas em locais específicos de tais registradores, verificando a presença destas senhas, tentativas de acesso por configurações incorretas podem ser diagnosticadas.

Como exemplo prático da aplicação de tais recursos, os modelos dos dispositivos de hardware desenvolvidos nesta dissertação são inicializados com ao menos uma função básica, a responsável por receber e tratar mensagens de erro e status geradas durante a sua execução. Os modelos geram as mensagens pela chamada de ponteiros para as funções que irão tratá-las, estes ponteiros são inicializados no início da sua execução por uma função do tipo “model_x_start()”, logo de início, uma mensagem do tipo “Modelo X inicializado com sucesso” é criada. Entre os parâmetros necessários para a inicialização dos modelos estão o endereço da função de tratamento das mensagens de erro e os endereços das funções específicas dos modelos, como, por exemplo, a função responsável pela geração das amostras do modelo do conversor A/D.

3.5 A Aplicação dos Modelos como Meios de Depuração de Software

Devido aos modelos serem construídos na mesma linguagem de programação do software embarcado, uma ferramenta de depuração não é capaz de distinguir qual software pertence aos modelos e qual software pertence à aplicação em desenvolvimento. Esta característica permite que o código dos modelos seja depurado da mesma forma que o código da aplicação. Um depurador eficiente, capaz de percorrer os caminhos internos do software mostrando os valores das variáveis internas, permitirá a descoberta de uma grande quantidade de configurações erradas simplesmente percorrendo as estruturas internas do software dos modelos dos dispositivos de hardware.

Como exemplo, considere-se o fragmento de código exibido no diagrama da Figura 3.11, responsável pelo preparo da memória flash para uma operação de escrita. Imagine-se que o desenvolvedor tenha esquecido de configurar corretamente o bit WRT do registrador de controle FCTL1. Este bit deve ser devidamente ligado quando se deseja efetuar uma operação de escrita na memória flash interna do microcontrolador. Ao perceber que o código não está funcionando, o desenvolvedor inicia um processo de depuração do código, executando-o passo a passo. No loop while do software em desenvolvimento, a execução saltará para dentro do código do modelo da memória flash. Acompanhando a execução passo a passo, o desenvolvedor perceberá, facilmente, que o bit de escrita não está habilitado, uma vez que no ponto de teste a execução do software do modelo saltará para o final do código, sem passar pela etapa de escrita.

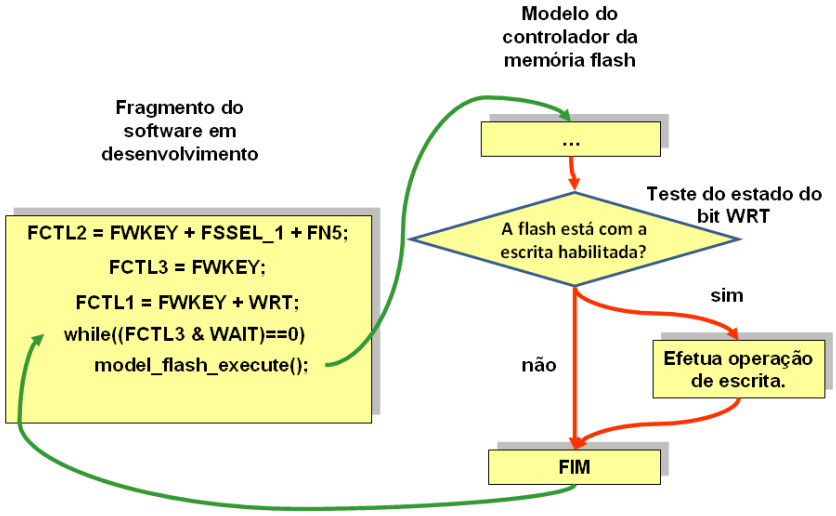


Figura 3.11: Modelos como meios de depuração passo a passo do código.

No mesmo fragmento de código da Figura 3.11, outros erros poderiam ser encontrados, tais como má configuração dos registradores de hardware FCTL2 e FCTL3, por exemplo, cuja configuração seria testada na parte não exibida do código. Apesar de tais problemas parecerem simples, dependendo da experiência do desenvolvedor, seria necessário tempo para leitura e análise da documentação do dispositivo, de forma a compreender as razões do seu mau funcionamento.

Os recursos adicionais de depuração são uma possibilidade única desta abordagem de desenvolvimento, não encontrada em nenhum tipo de simulador. Um simulador fornecido por um fabricante de microcontroladores, por exemplo, que fosse capaz de simular o comportamento dos dispositivos de hardware, permitiria apenas uma análise limitada do problema. Não haveria informação para uma detecção mais precisa da razão do mau funcionamento. Isto se deve ao fato de tais simuladores serem integrados aos ambientes de desenvolvimento de forma completamente transparente ao usuário, não sendo disponíveis, de modo algum, na forma de código fonte.

3.6 Organização em Camadas

Durante toda a apresentação da forma como o software deverá estar organizado, falou-se a respeito de software dependente do hardware e software de aplicação. De fato, esta é uma forma bastante interessante de se enxergar o software embarcado, proposta em (KANG; KWON; LEE, 2005). Nesta proposta, o software embarcado é apresentado em camadas, onde há o software de aplicação na camada superior e o software dependente do hardware na camada inferior, sendo que podem ser considerados software dependente do hardware, sistemas operacionais de tempo real, drivers de dispositivos e de comunicação (Figura 3.12).

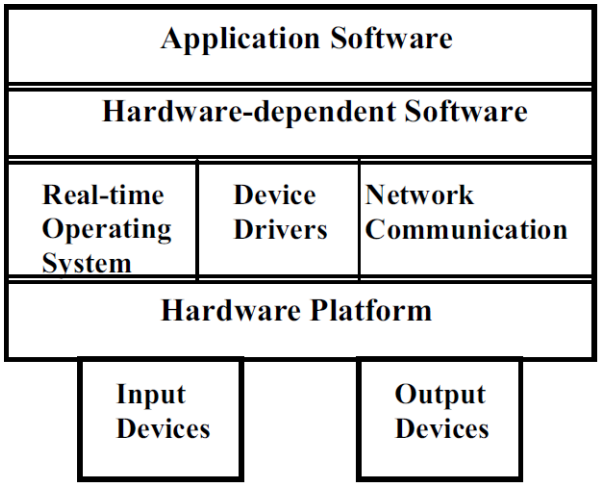


Figura 3.12: Organização do software embarcado em camadas (KANG; KWON; LEE, 2005).

A organização do software embarcado em camadas é fundamental para o sucesso do teste, uma vez que isto permite o encapsulamento dos módulos de software dependente do hardware, onde está localizada a maior parte das incompatibilidades de software. Desta forma, tanto a manutenção do código, quanto a aplicação e gerenciamento dos modelos é facilitada, uma vez que apenas o software dependente do hardware fará acesso aos modelos dos dispositivos de hardware, conforme exibido na Figura 3.1.

Além disto, a correta organização do código permitirá o uso de outras estratégias, como, por exemplo, a proposta em (KARLESKY; WILLIAMS, 2007), onde *mocks* são criados logo na camada acima do software dependente do hardware, em uma camada intermediária de software capaz de abstrair o hardware. Finalmente, o encapsulamento do software dependente do hardware eleva a portabilidade do código, uma vez que o software de aplicação permanecerá praticamente intocado no caso da necessidade de troca de um microcontrolador, por exemplo.

3.7 Compatibilidade de Software

Um aspecto fundamental para atingir o objetivo da compilação condicional para ambas as plataformas diz respeito à compatibilização dos tipos de dados entre os compiladores para a plataforma alvo e para o ambiente de aplicação. Em C, em geral, os tipos de dados são compatíveis, entretanto uma exceção é a representação do tipo inteiro 'int', que poderá ser representado por uma palavra de 16 ou 32 bits, dependendo da arquitetura do microprocessador utilizado. Um compilador C para software de aplicação, usualmente alocará 32 bits para a representação do tipo 'int', enquanto um compilador para uma plataforma alvo com arquitetura de 16 bits, por exemplo, representaria este tipo como uma palavra de 16 bits.

Neste trabalho, o seguinte padrão foi adotado para se atingir este objetivo: nenhum tipo básico de dado foi utilizado no software embarcado e nos modelos dos dispositivos de hardware, mas apenas tipos pré-definidos. Estes tipos pré-definidos estão localizados no arquivo "Geral.h" (Figura 3.4), tornando-se visíveis para todos os arquivos do projeto. A estrutura geral desta abordagem é detalhada na Figura 3.13.

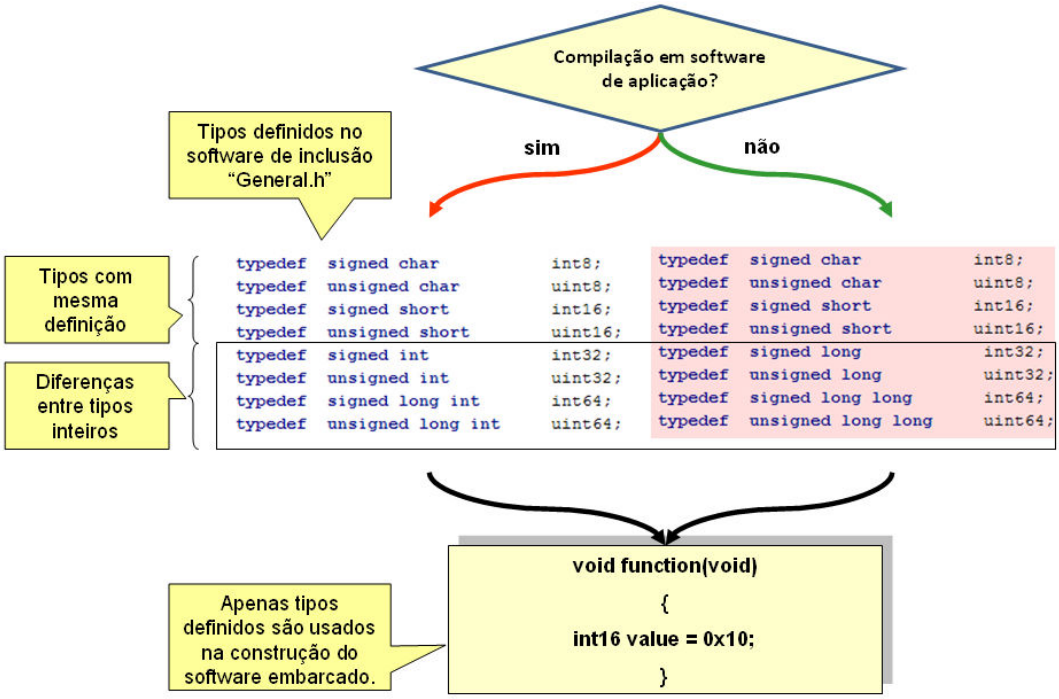


Figura 3.13: Compatibilização de tipos de dados entre compiladores.

Observa-se que, caso esta abordagem não fosse utilizada, um tipo de dado 'int' passaria a ter dois tamanhos diferentes, dependendo do compilador utilizado. Na plataforma alvo seria representado por 16 bits e no ambiente de aplicação por 32 bits. Esta diferença seria automaticamente transmitida para todos os outros componentes de software, como arrays e estruturas de dados.

Além de tipos básicos de dados, no arquivo “Geral.h” deverão estar definidas todas as diferenças entre os compiladores, por exemplo, funções intrínsecas e #pragmas de compilação. Entretanto, é importante ressaltar que não é uma boa prática de programação o uso de funções intrínsecas ou #pragmas específicos de compiladores, uma vez que isto torna o software muito pouco portável para futuras aplicações e arquiteturas.

Outro aspecto que deverá ser considerado diz respeito às estruturas de dados. A linguagem C possui uma forma extremamente simples de se organizar tipos de dados conforme o seu contexto de aplicação: a definição de tipos complexos de dados através do uso da palavra reservada “struct”. Como exemplo, na Figura 3.14 é apresentado um tipo de dado complexo chamado TSamples, sendo constituído por três dados inteiros de 16 bits, Voltage, HighCurrent e LowCurrent. Esta estrutura é utilizada para a organização das amostras adquiridas do conversor A/D que serão utilizadas para os cálculos de energia no software do medidor eletrônico de energia desenvolvido para esta dissertação.

```
/// Amostras adquiridas pelo conversor AD
typedef struct {
        int16 Voltage;          ///< sinal de tensao (escala unica)
        int16 HighCurrent;      ///< sinal de corrente, escala grande
        int16 LowCurrent;       ///< sinal de corrente, escala pequena
}TSamples;

{
/// Exemplo de uso da estrutura
TSamples Amostras;

Amostras.Voltage = 1000;
Amostras.HighCurrent = 10;

}
```

Figura 3.14: Exemplo de tipo de dado definido com uso da palavra reservada “struct”.

Por referência direta, ou seja, quando o software aponta para a variável e seu campo interno, como no exemplo da Figura 3.14, o tipo de dado definido TSamples poderá ser livremente acessado, tanto no software compilado para o ambiente de aplicação, quanto no software compilado para a plataforma alvo, sem qualquer tipo de problema. Entretanto, caso o software embarcado necessite, por exemplo, armazenar este tipo de dado em algum sistema de arquivos, ou memória não-volátil, por exemplo, a análise do tamanho da estrutura poderá ser diferente, dependendo do compilador utilizado.

Apesar dos campos internos da estrutura terem o seu tamanho e tipos bem determinados, inteiros de 16 bits, os compiladores sempre buscam “alinhar” as estruturas conforme a arquitetura da plataforma. Ou seja, por default, um compilador C para uma arquitetura 32 bits organizará os dados de forma que cada campo da estrutura da Figura 3.14 inicie em um endereço múltiplo de 32 bits. Isto fará com que o tamanho

final da estrutura, que contém três dados inteiros de 16 bits, seja de doze bytes e não de seis bytes, como esperado.

É importante dizer que este tipo de problema é enfrentado mesmo dentro da mesma arquitetura, quando campos com tipos de dados de tamanho menor do que o tamanho de palavra padrão da arquitetura são criados. Por exemplo, em um compilador de 16 bits, os campos internos que possuam tamanho de 8 bits deverão ser dispostos sempre dois a dois, de modo que, um campo seguinte de 16 bits possa ser disposto sem que se perca um byte, conforme ilustrado na Figura 3.15.

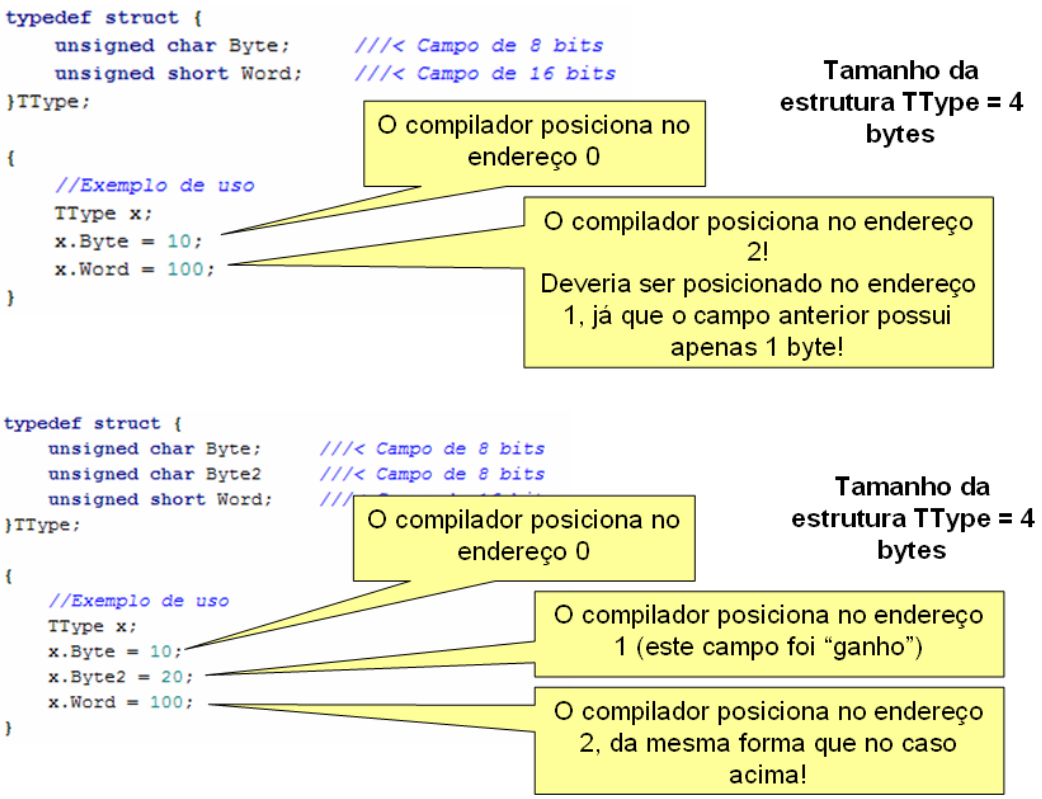


Figura 3.15: Exemplo do problema de alinhamento de memória em tipos de dado complexos.

Finalmente, sempre que o alinhamento das estruturas afetar a lógica de execução do código, os compiladores podem ser configurados para que posicionem as estruturas da forma desejada, independentemente do tamanho de palavra da arquitetura alvo, como exemplificado na janela de configurações da Figura 3.16.

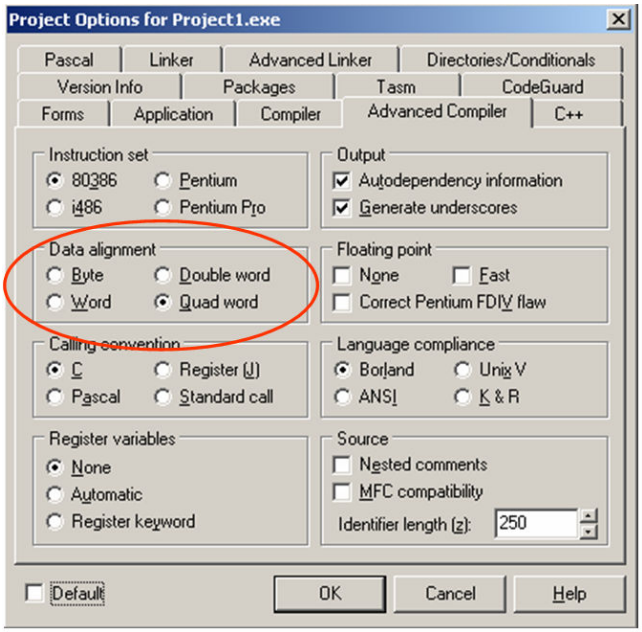


Figura 3.16: Janela de configurações do compilador C++ Borland Builder (BUILDER, 2010), onde é destacada a opção de alinhamento de estruturas de dados.

3.8 Compatibilidade dos Modelos Desenvolvidos

A organização dos dispositivos de hardware como periféricos dentro da arquitetura de um microcontrolador permite que, dentro de uma mesma família de microcontroladores, a compatibilidade dos modelos desenvolvidos seja extremamente alta. De tal forma que, mesmo quando diferenças são encontradas, apenas pequenos ajustes precisam ser feitos nos modelos de forma a torná-los 100% compatíveis entre si.

Com o objetivo de se verificar a abrangência do uso de modelos de um dispositivo de hardware criado para a família de microcontroladores MSP430, fabricada pela Texas Instruments, uma ferramenta on-line de seleção de microcontroladores (TEXAS, 2010) foi utilizada para a execução de um busca por microcontroladores com características semelhantes. Com o uso desta ferramenta, foram filtrados dados a respeito da disponibilidade de periféricos nos microcontroladores desta família, mais especificamente, o critério de filtragem foi a existência de conversores A/D de 12 bits em dispositivos com memória flash interna. A idéia, basicamente, é verificar em quais dispositivos os modelos de conversor A/D e flash poderiam ser utilizados.

A partir desta pesquisa, foram localizados 57 microcontroladores atendendo a ambos os critérios: memória flash interna e conversor A/D de 12 bits. Os resultados, de forma parcial, podem ser vistos na Tabela 3.2, onde se verifica que conversores A/D de 12 bits podem ser encontrados em um grande número de dispositivos, desde dispositivos bastante limitados, como o microcontrolador MSP430F133 (que possui 8 kB de memória flash, 256 bytes de memória RAM, no máximo 8 MIPS de velocidade de processamento e não possui recursos de multiplicação por hardware), até dispositivos bem mais avançados, como o MSP430F5437 (256 kB de memória flash, 16 kB de memória RAM, até 25 MIPS de velocidade de processamento e multiplicação por hardware em 32 bits).

Tabela 3.2: microcontroladores da família MSP430 com conversor A/D de 12 bits e memória flash interna (TEXAS, 2010).

Item	Part Number	Flash	RAM	Max Speed	MPY	Active Current	MAX GPIO	Timers	ADC Channels	ADC Resolution
1	MSP430F5437	256	16	25	32x32	0.165	67	4	16	12
2	MSP430F5438	256	16	25	32x32	0.165	87	4	16	12
3	MSP430F5435	192	16	25	32x32	0.165	67	4	16	12
4	MSP430F5436	192	16	25	32x32	0.165	87	4	16	12
5	MSP430F5418	128	16	25	32x32	0.165	67	4	16	12
6	MSP430F5419	128	16	25	32x32	0.165	87	4	16	12
7	MSP430F5528	128	10	25	32x32	0.16	47	5	12	12
8	MSP430F5529	128	10	25	32x32	0.16	63	5	16	12
9	MSP430F2419	120	4	16	16x16	0.365	64	3	8	12
10	MSP430F2619	120	4	16	16x16	0.365	64	3	8	12
11	MSP430FG4619	120	4	8	16x16	0.4	80	5	12	12
12	MSP430F2418	116	8	16	16x16	0.365	64	3	8	12
13	MSP430F2618	116	8	16	16x16	0.365	64	3	8	12
14	MSP430F2618-EP	116	8	16	16x17	0.365	64	3	8	12
15	MSP430FG4618	116	8	8	16x16	0.4	80	5	12	12
16	MSP430F5526	96	8	25	32x32	0.16	47	5	12	12
17	MSP430F5527	96	8	25	32x32	0.16	63	5	16	12
	...									
45	MSP430F437	32	1	8	0	0.28	48	3	8	12
46	MSP430F447	32	1	8	16x16	0.28	48	3	8	12
47	MSP430F5521	32	8	25	32x32	0.16	63	5	16	12
48	MSP430F5522	32	10	25	32x32	0.16	47	5	12	12
49	MSP430FG437	32	1	8	0	0.3	48	5	12	12
50	MSP430F156	24	1	8	0	0.33	48	3	8	12
51	MSP430F436	24	1	8	0	0.28	48	3	8	12
52	MSP430F135	16	0.5	8	0	0.28	48	3	8	12
53	MSP430F155	16	0.5	8	0	0.33	48	3	8	12
54	MSP430F235	16	2	16	16x16	0.27	48	3	8	12
55	MSP430F435	16	0.5	8	0	0.28	48	3	8	12
56	MSP430F133	8	0.25	8	0	0.28	48	3	8	12
57	MSP430F233	8	1	16	16x16	0.27	48	3	8	12

A análise dos dados da Tabela 3.2 mostra que os modelos de dispositivos de hardware são amplamente utilizáveis, dentro de uma mesma família de microcontroladores. Além disto, fabricantes de sistemas embarcados costumam adotar plataformas de desenvolvimento baseadas em microcontroladores de uma mesma família, uma vez que são plataformas voltadas para o seu mercado de atuação e que, desta forma, se aproveita toda a estrutura interna de desenvolvimento, teste e processo produtivo (*know-how* de projeto, compiladores, depuradores, equipamentos de teste, softwares de programação, etc.). Neste contexto, verifica-se que um modelo de dispositivo de hardware poderá ter a sua aplicação, seguramente, em uma série de projetos futuros desenvolvidos pela empresa, e não somente naquele para o qual foi construído.

3.9 Resumo e Conclusões

Neste capítulo foram apresentados os principais aspectos referentes à forma de organização do software embarcado, mostrando como este software deve ser construído de forma a integrar os modelos dos dispositivos de hardware e permitir a compilação condicional, ora para o ambiente de aplicação, ora para a plataforma alvo.

Foram detalhadas as questões referentes à memória compartilhada, criada dentro do software de execução dos testes e livremente acessada pelos modelos dos dispositivos de hardware e pelo software embarcado em desenvolvimento. O que permite que, em

um ambiente de software voltado ao software de aplicação, o software embarcado e os modelos acessem os registradores de hardware exatamente da mesma forma que o software compilado para a plataforma alvo.

Buscou-se generalizar a estrutura interna dos modelos dos dispositivos de hardware, de forma a sistematizar a sua construção, foi mostrada a aplicação destes modelos na modelagem de aspectos externos ao hardware, fundamental para softwares embarcados, e como uma importante ferramenta de auxílio na depuração de código. Mostrou-se como a simples execução dos modelos já é capaz de eliminar uma série de possibilidades de configurações incorretas.

Apresentaram-se as principais razões das incompatibilidades entre ambientes de desenvolvimento de software de aplicação e ambientes de desenvolvimento de software embarcado e formas de lidar com estas incompatibilidades (mapeamento dos dispositivos de hardware, tipos de dados e alinhamento de estruturas).

Finalmente, fez-se uma breve análise com relação às possibilidades de uso de modelos dos dispositivos de hardware em mais do que um projeto de software embarcado, onde se constatou que os modelos são amplamente compatíveis para microcontroladores de uma mesma família.

4 ESTUDO DE CASO: MEDIDOR ELETRÔNICO DE ENERGIA

Visando a validação da metodologia de desenvolvimento e teste proposta, foi construído, para esta dissertação, o software embarcado de um medidor eletrônico de energia elétrica, onde todos os conceitos expostos no capítulo 3 desta dissertação foram aplicados. A análise deste tipo de software é interessante, uma vez que tais equipamentos são típicos sistemas embarcados, com muitas de suas principais características, tais como:

- São equipamentos microprocessados de propósito específico (medição eletrônica de energia elétrica);
- Possuem limitação de consumo de energia (estipulada por normas);
- Possuem memória RAM e de código limitada. Usualmente, são sistemas embarcados baseados em microcontroladores, onde um microcontrolador central composto por uma série de periféricos de hardware executa a aplicação embarcada;
- Contém restrições de tempo real. Os tempos de aquisição e processamento são críticos pois afetam diretamente os módulos de medição de energia, de comunicação e exibição de dados;
- São equipamentos com pequena margem de lucro, grandes volumes de produção e reduzido *time to market* (do Inglês *tempo de chegada do produto ao mercado*).

Além dos aspectos já mencionados, o projeto de medidores eletrônicos de energia está em alta, uma vez que, no Brasil, o mercado de medição eletrônica vem sofrendo uma grande transformação. Mais especificamente, o ano de 2009 pode ser reconhecido como o ano da virada na medição eletrônica de energia, onde, pela primeira vez, medidores eletrônicos foram vendidos em número igual ou superior aos seus equivalentes eletromecânicos. As causas para esta virada são muitas, mas se pode citar o expressivo aumento de competitividade dos produtos eletrônicos frente aos antigos projetos dos equipamentos eletromecânicos e a capacidade que os medidores eletrônicos possuem de diagnóstico de fraudes.

Porém, a chegada em massa da medição eletrônica ao mercado vem aliada a um conseqüente aumento no número de fabricantes concorrentes. Este aumento no número de fabricantes tende a forçar os preços dos equipamentos para níveis menores dos atuais, que já estão baixos.

Como alternativa para a sobrevivência no mercado, os fabricantes buscam agregar valor aos seus produtos. Atualmente, mesmo medidores eletrônicos considerados básicos, deverão ser capazes de medir energia ativa e reativa, e de transmitir seus dados a partir de algum tipo de porta de comunicação, seja por via óptica ou por cabo. Esta tendência pode ser constatada pela análise de um projeto, em discussão na ANEEL, que pretende a substituição de todo o parque instalado de medidores de energia, em um prazo de 10 anos, por equipamentos com mais recursos e conectividade. Estes novos equipamentos serão controlados por softwares bem mais complexos, o que exigirá de desenvolvedores uma busca por ferramentas e métodos de projeto que reduzam os tempos de desenvolvimento, garantindo, ao mesmo tempo, a qualidade dos softwares criados.

4.1 Visão Geral da Aplicação

Em síntese, as tarefas de um medidor eletrônico de energia básico podem ser resumidas em:

- Adquirir os sinais analógicos de tensão e corrente;
- Executar os cálculos de medição de energia propriamente dita;
- Informar ao usuário a quantidade de energia medida, através de dados exibidos em um display ou através de meios de comunicação;
- Informar, através de pulsos emitidos em LEDs, a quantidade instantânea de energia medida. O número de pulsos é diretamente proporcional à energia, permitindo a aferição do equipamento;
- Recuperar os dados de faturamento, dados de medição, sempre que o equipamento é energizado, e salvá-los de tempos em tempos e na ocorrência de uma queda de energia;

4.2 Plataforma Alvo

O projeto foi desenvolvido utilizando-se um microcontrolador da família MSP430 da Texas Instruments (MSP430, 2010). Com arquitetura RISC de 16 bits, os microcontroladores da família MSP430 são capazes de atingir até 25 MIPs de velocidade de processamento, com uma excelente relação freqüência/consumo. Estes microcontroladores podem ser encontrados com uma ampla variedade de periféricos de hardware integrados, tais como: conversores A/D de 10, 12 e 16 bits, conversores D/A de 12 bits, memória flash interna, driver de LCD, portas de comunicação UART, I2C, SPI, IRDA e USB, multiplicadores por hardware de 16 e 32 bits, sensor de temperatura, temporizadores, RTC, etc.

Amplamente utilizado em sistemas embarcados voltados a aplicações de baixíssimo consumo de energia, usualmente dispositivos portáteis alimentados com bateria, o MSP430 é utilizado em medidores eletrônicos de energia elétrica devido às restrições de consumo deste tipo de equipamento, impostas por norma (RTM, 2010). Especificamente para o trabalho desta dissertação, o microcontrolador utilizado foi o MSP430F235, cujas características principais são exibidas na Figura 4.1. Este item foi escolhido devido à disponibilidade do hardware para a validação do projeto. Entretanto, conforme já mencionado, dentro da família de microcontroladores MSP430, a

abordagem proposta nesta dissertação é muito facilmente portada para qualquer outro microcontrolador.

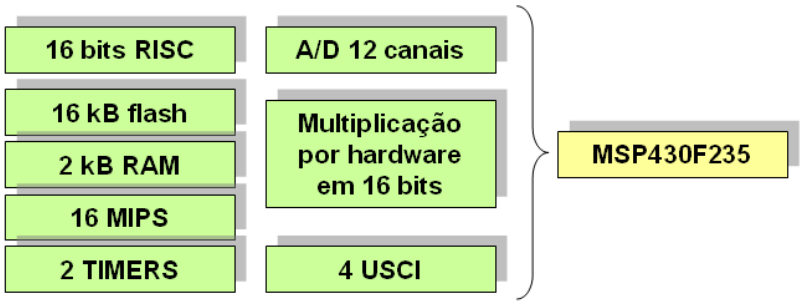


Figura 4.1: Principais características do microcontrolador MSP430F235.

O hardware utilizado para a validação final do projeto, apresentado de forma simplificada na Figura 4.2, é um hardware padrão para dispositivos de medição, constituído por uma fonte de alimentação, circuitos de condicionamento dos sinais analógicos de tensão e corrente, LEDs utilizados para calibração da energia medida e uma porta de comunicação para transmissão dos dados de medição.

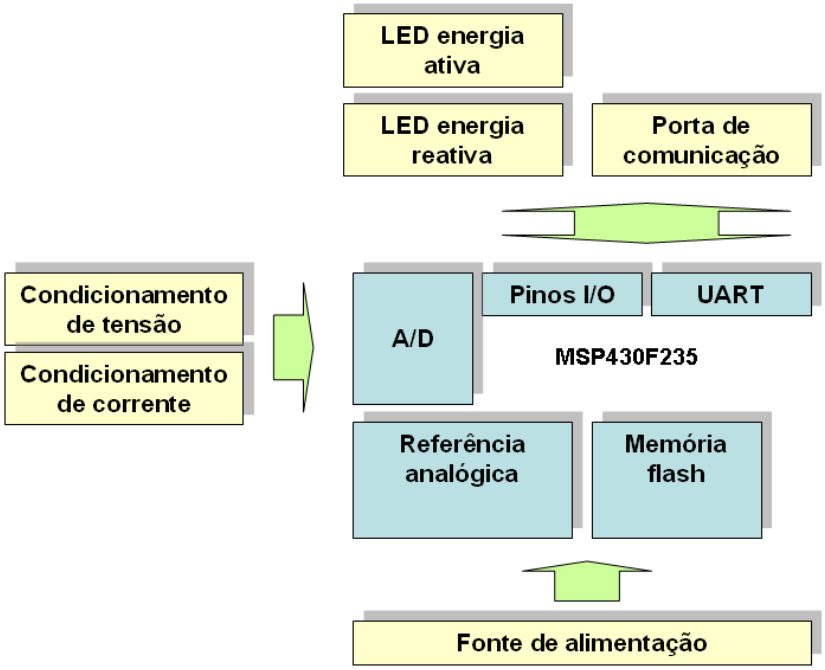


Figura 4.2: Diagrama geral do hardware utilizado para validação.

4.3 Módulos de Software Desenvolvidos

O software do medidor de energia, juntamente com os modelos dos dispositivos de hardware e o software de testes, é constituído por 19 módulos de software, escritos em linguagem C e organizados conforme a sua aplicação. Foram construídos cinco modelos dos dispositivos de hardware, cinco módulos fortemente dependentes do hardware e nove módulos de software de aplicação e testes, conforme apresentado na Figura 4.3.

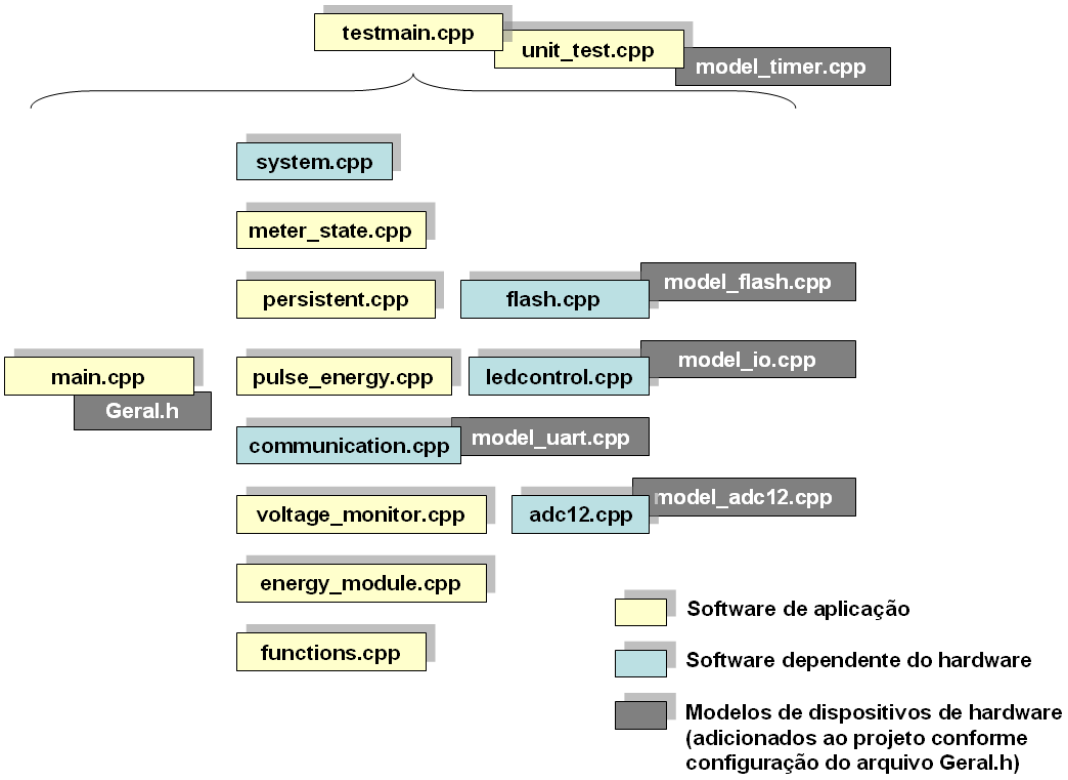


Figura 4.3: Módulos de software construídos para a validação do trabalho.

A seguir, faz-se uma breve explicação do quê exatamente cada módulo de software faz para a implementação da aplicação medidor de energia:

- Main: este é o módulo principal do software, contém o laço principal de execução e é responsável pelo controle geral da aplicação, chamando os demais módulos do projeto;
- System: este módulo é responsável pela inicialização do sistema, ajustando valores iniciais para o clock de operação, temporizadores, estado de pinos de I/O, watchdog, etc;
- Meter_state: faz o controle de uma máquina de estados finitos, representando o estado atual do medidor. Quando alimentado, por exemplo, o medidor encontra-se no estado de energização, onde ele deverá recuperar os dados de faturamento e entrar em operação;

- Persistent: este módulo é responsável pelo armazenamento e recuperação de dados considerados persistentes, como dados de faturamento, por exemplo. Todo o acesso ao módulo flash (dependente do hardware) é efetuado através deste módulo;
- Pulse_energy: é responsável pelo incremento dos contadores de energia e acionamento dos LEDs metrológicos. Faz também o monitoramento da quantidade de energia medida pelo equipamento, sinalizando, se necessário, condição de estado sem energia, "vazio";
- Communication: inicializa e controla a UART, gerencia a interrupção de transmissão e formata os dados transmitidos pela aplicação. Interage diretamente com o modelo da UART, de forma a simular o comportamento real na ausência do hardware;
- Voltage_monitor: monitora continuamente o nível de tensão, verificando se está adequado para a medição ou se está muito baixo, o que indicaria uma possível queda de energia;
- Energy_module: coração do equipamento, este módulo efetua todos os cálculos de medição propriamente ditos, a partir de amostras dos sinais de tensão e corrente. São calculadas as energias ativas e reativas nos quatro quadrantes de medição (energia ativa direta e reversa, energias reativa indutiva e capacitiva direta e reversa). Ainda, de forma a obter a precisão necessária, desde a corrente mínima até a máxima, neste módulo é efetuado o controle de qual escala de medição de corrente utilizar;
- Functions: um módulo onde são armazenas funções de uso geral, como obtenção de valor absoluto e movimentação de dados, por exemplo;
- Flash: contém funções de acesso à memória flash interna, permitindo escrita e apagamento. Este módulo interage diretamente com o modelo do dispositivo de forma a simular o comportamento real, quando executado em ambiente de software de aplicação;
- Led_control: efetivamente liga e desliga os LEDs metrológicos, controlando os tempos em que estes ficam acesos;
- ADC_12: contém as funções de acesso ao conversor A/D, interage com o modelo do dispositivo de forma a simular o comportamento real, quando em ambiente de software de aplicação;
- Model_flash: contém o código que implementa a funcionalidade da memória flash interna do microcontrolador. Utilizado na ausência do hardware, simula operações de escrita e apagamento de setores da flash, entre outras características;
- Model_uart: modelo funcional da uart interna do microcontrolador. Executa uma máquina de estados interna capaz de chamar funções externas simulando interrupções e gerencia configurações específicas, como inversão de bits em um frame, etc;
- Model_io: modelo funcional dos pinos de entrada e saída do microcontrolador. Contém uma memória com o estado real dos pinos de

entrada e saída e apenas altera estes estados ou permite sua leitura se as configurações das portas estiverem corretas;

- Model_adc12: modelo funcional do conversor A/D do microcontrolador, permite a criação de uma função externa capaz de gerar as amostras a serem lidas pelo dispositivo, quando executado em ambiente de software de aplicação.
- Model_timer: modelo funcional simplificado de um dos timers do microcontrolador. Permite apenas a execução de testes de unidade visando à verificação da correta configuração do dispositivo. De forma a permitir a contagem de tempo, cada chamada à função de execução do modelo ocasiona um incremento dos contadores internos.

5 APLICAÇÃO DA METODOLOGIA PROPOSTA

Para o software do medidor eletrônico de energia, foram desenvolvidos cinco modelos de dispositivos de hardware construídos conforme a metodologia proposta, permitindo, assim, o desenvolvimento e teste do software embarcado em um ambiente voltado a software de aplicação, sem a presença do hardware. Observa-se que a escolha dos dispositivos a serem modelados é fortemente dependente da aplicação embarcada. No caso do medidor eletrônico de energia, o conversor A/D e a memória flash interna são os principais dispositivos de interesse, pois estão diretamente associados aos testes dos módulos de medição e salvamento e restauração de dados, ambos considerados críticos para este tipo de projeto.

5.1 Modelos Construídos

A seguir, são ressaltadas as principais características e possibilidades dos modelos dos dispositivos de hardware construídos.

5.1.1 Conversor Analógico-Digital

Para o uso do modelo do conversor A/D, o software driver deverá criar duas funções básicas, uma para o recebimento de mensagens de texto enviadas pelo modelo e outra para a geração das amostras requisitadas. Ambas as funções são passadas ao modelo durante a sua inicialização, através da função "model_adc12_start", conforme a Figura 5.1.

```
//Declara um ponteiro para uma função que retorna um int16 e recebe um parâmetro int16
typedef int16(*TPExternFunc)(int16);
//Declara um ponteiro para uma função que recebe como parâmetro um ponteiro para uma string
typedef void(*TPMessageFunction)(const char*);
//Declara um ponteiro para uma função que não recebe parâmetros nem retorna nada
typedef void(*TPInterruptVector)(void);


void model_adc12_start(TPExternFunc value, TPMessageFunction messagefunc){
        RequestSample=value;
        SendADC12Messages=messagefunc;
        SendADC12Messages("MODEL ADC12: started\n");
        show_adc12_pin_error=true;
}
```

Figura 5.1: Inicialização do modelo do conversor A/D.

A função de fornecimento de amostras dos canais recebe como parâmetro o canal do conversor A/D a ser lido e retorna a amostra convertida. Esta função apenas será chamada pelo modelo do conversor A/D após a avaliação de todas as configurações para o perfeito funcionamento do conversor e após todos os estados internos terem sido adequadamente percorridos. É importante ressaltar que o controle sobre a geração das amostras é de responsabilidade do software driver e não do modelo, que apenas efetua a requisição da amostra conforme a programado pela aplicação embarcada.

O modelo do conversor A/D utiliza a função de transmissão de mensagens para a sinalização de mau funcionamento e status de inicialização. Por exemplo, sempre que o modelo do conversor é inicializado, a mensagem "Model ADC12: started" é enviada, em outra situação, quando o software embarcado tenta efetuar uma leitura de um canal analógico mapeado para um pino externo, esta aquisição apenas será efetuada se o pino estiver devidamente configurado como entrada analógica do conversor, caso contrário, uma mensagem de erro será enviada através desta função e um valor nulo será retornado como amostra (Figura 5.2).

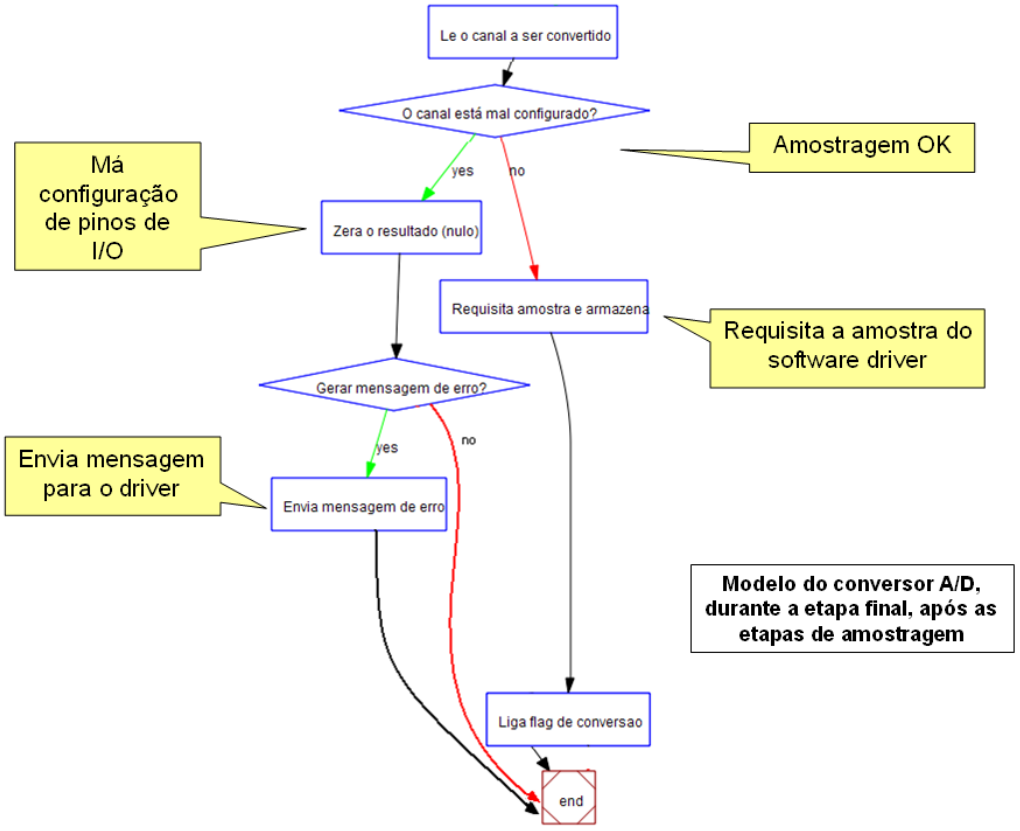


Figura 5.2: Etapa final do modelo do conversor A/D, onde as amostras são requisitadas e mensagens de erro são enviadas, se necessário.

No driver de testes, a função de geração de amostras foi construída de forma a criar dinamicamente as amostras, calculando as formas de onda de tensão e corrente conforme parâmetros de configuração. Além de gerar formas de onda com freqüência, amplitude e fase bem determinadas, a função é capaz de provocar desligamentos em

qualquer dos canais, conforme a configuração corrente, determinada pelo caso de teste em execução. Este tipo de recurso é extremamente útil para testar a capacidade de detecção de queda de energia, quando a tensão vai a zero, por exemplo, ou para testar a capacidade de detecção de ausência de carga (situação de "vazio") quando a corrente fica abaixo de um limiar pré-estabelecido em norma. A Figura 5.3 mostra o diagrama de fluxo de controle da função de geração de amostras para o modelo do conversor A/D, criada dentro do software driver de testes.

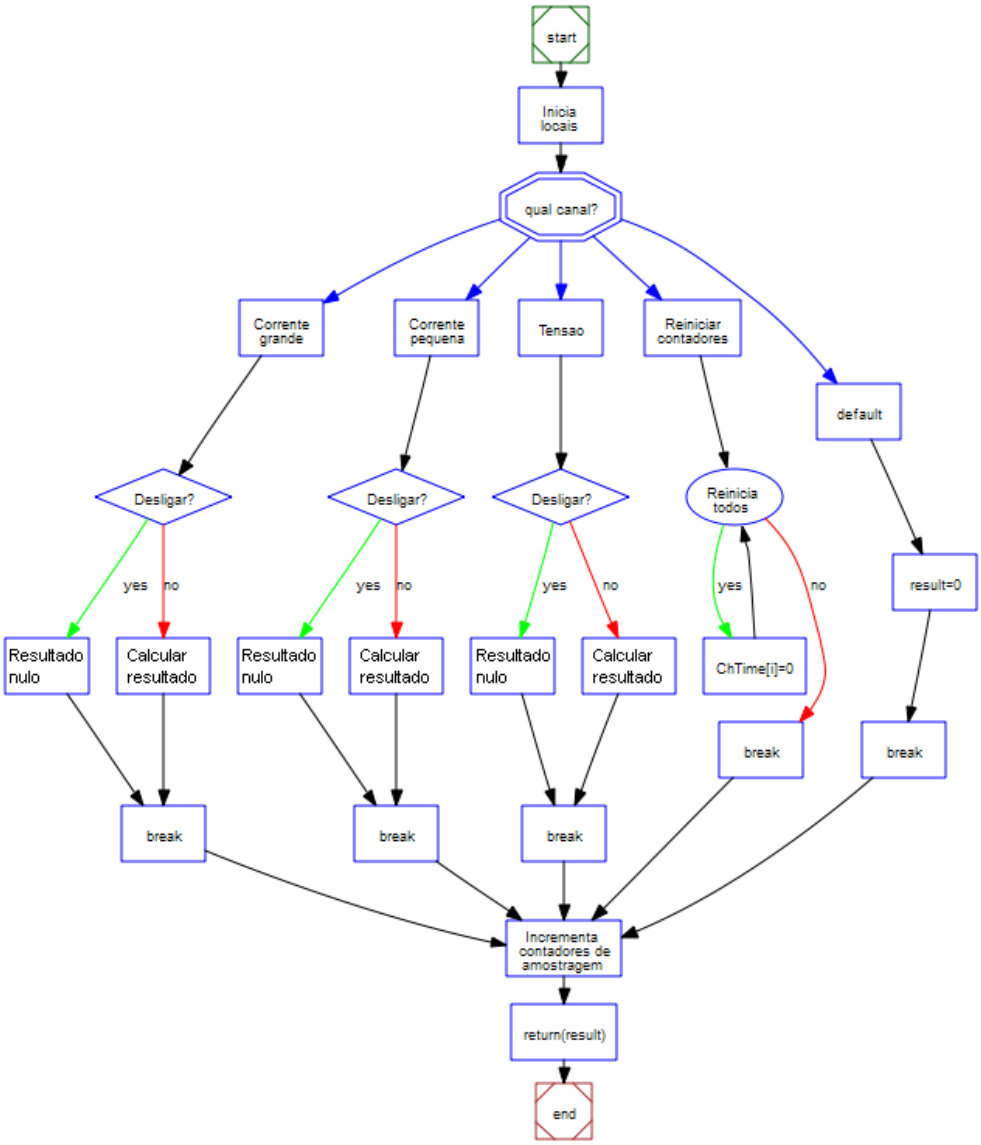


Figura 5.3: Diagrama de fluxo de controle da função de geração de amostras para o modelo do conversor A/D, criada dentro do software driver de testes.

5.1.2 Memória Flash Interna

O modelo da memória flash é capaz de simular as funções de escrita e de apagamento da memória flash interna do microcontrolador, foram modelados um modo de escrita (byte a byte) e três modos de apagamento (segmentos, memória de programa e toda a memória). Um diagrama básico do funcionamento da função de execução do modelo é apresentado na Figura 5.4, onde as principais decisões são mostradas.

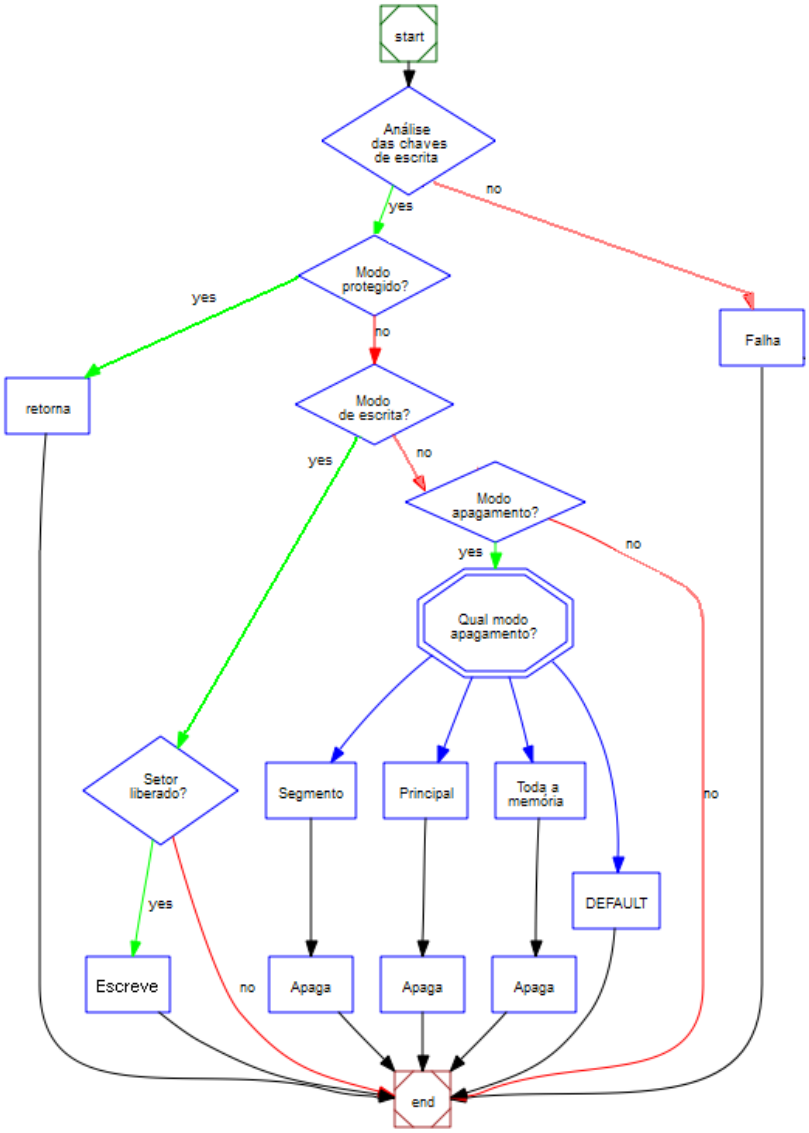


Figura 5.4: Diagrama de fluxo de controle da função de execução do modelo da memória flash.

Para simular o modo de funcionamento da memória flash, dentro do modelo, dois arrays de bytes foram criados: um temporário, representando as novas escritas e um permanente, representando o conteúdo atual da memória flash. Sempre que o software de aplicação escreve dados na memória, este está efetivamente escrevendo dados no

array temporário de memória. Apenas no momento de execução do modelo da memória flash, caso os registradores de hardware estejam corretamente configurados, os dados são transferidos do array temporário para o array permanente, efetivamente executando a operação de escrita. De forma análoga, sempre que o software de aplicação lê dados da memória flash, os dados são lidos do array permanente. Este mecanismo é exibido no diagrama da Figura 5.5.

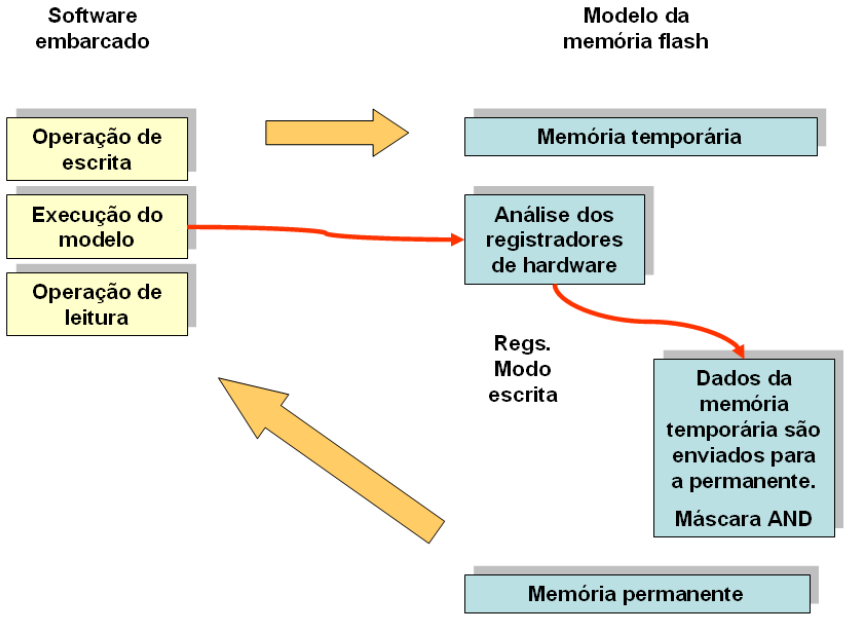


Figura 5.5: Método de gerenciamento de leituras e escritas efetuado pelo modelo da memória flash.

Ainda conforme a Figura 5.5, observa-se que, ao efetuar a transferência do conteúdo da memória temporária para a memória permanente, o modelo o faz através de uma operação lógica AND com o conteúdo de ambas as memórias. Desta forma, não é possível escrever nível lógico um em um bit com nível lógico zero da memória permanente, mas apenas escrever zero lógico zero em um bit com nível lógico um. Com este mecanismo, para se efetuar uma escrita correta na memória permanente, será preciso que esta memória esteja apagada, que no contexto de memórias flash, significa estar com todos os seus bits em nível lógico um. A execução deste mecanismo modela exatamente o que ocorre em uma memória flash real.

No software embarcado, de forma a permitir a sua execução tanto na plataforma alvo, quanto em ambiente de software de aplicação, o acesso à memória flash deve ser feito através de macros de leitura e escrita. Estas macros, quando executadas na plataforma alvo, direcionam as leituras e escritas para a memória interna do microcontrolador. Quando executadas em ambiente de aplicação, as macros direcionam as leituras para a região de memória permanente do modelo da flash e as escritas para a região de memória temporária.

A aplicação do modelo da memória flash, da forma como foi construído, permite ao software driver a execução de casos de teste, em ambiente de software de aplicação, capazes de testar todo o software embarcado de acesso à memória flash, tanto em nível de software dependente do hardware, quanto em nível de software de aplicação. Ainda, estes casos de teste, poderão ser aplicados, sem qualquer tipo de alteração, quando o software embarcado estiver em execução na plataforma alvo, com a presença do hardware real.

Como exemplo, no software do medidor eletrônico de energia, conforme a Figura 4.3, o módulo "persistent.cpp" contém funções que permitem o salvamento e restauração de dados da aplicação embarcada. Estas funções acessam diretamente o módulo de software dependente do hardware, "flash.cpp" que, por sua vez, quando executado em ambiente de software de aplicação, acessa o modelo da memória flash para a escrita e leitura de dados. Com o modelo da memória flash, casos de teste das funções do módulo "persistent.cpp", como a função "pers_restore_meas_data", que é responsável pela recuperação dos dados de medição, podem ser executados tanto em ambiente de software de aplicação, quanto na plataforma alvo, livremente, não sendo necessária qualquer alteração no software sob teste (Figura 5.6).

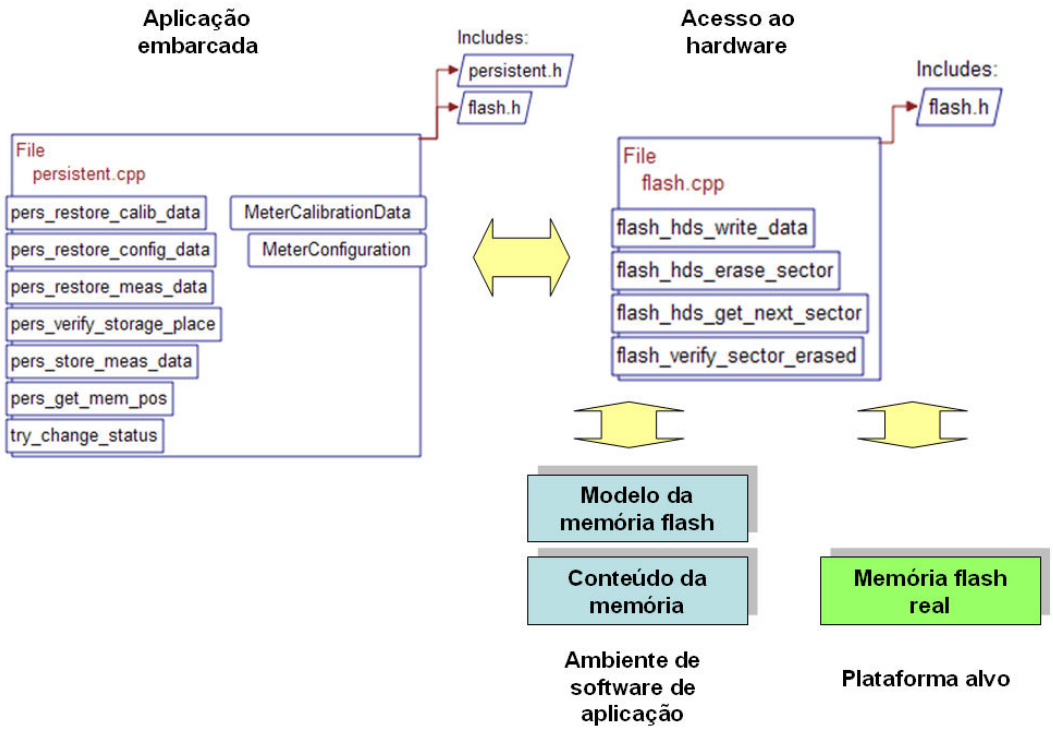


Figura 5.6: Relação entre o modelo da memória flash e os módulos de software dependentes do hardware e de aplicação.

5.1.3 UART

O modelo da UART foi construído de forma a implementar o caminho de transmissão de dados utilizado no software do medidor eletrônico de energia. De forma a tornar a simulação da UART o mais real possível, o modelo recebe como parâmetros, durante a sua inicialização, o endereço de duas funções: uma que será chamada ao final do processamento da UART, quando o byte termina de ser enviado, e outra de sinalização de término de transmissão, simulando uma interrupção. A seguir, faz-se um detalhamento das principais características deste modelo, cujo diagrama simplificado é mostrado na Figura 5.7.

- Formação dos bytes transmitidos: o modelo constrói o byte transmitido, conforme o modo configurado nos registradores de hardware, considerando o número de bits e a ordem em que devem ser enviados (a partir do menos ou do mais significativo). Desta forma, o byte escrito na saída, terá coerência com a configuração atual da UART. Por exemplo, caso o software de aplicação escreva no registrador de transmissão o byte 0x80h e a serial esteja configurada para transmitir o mais significativo primeiro, o byte transmitido na saída será 0x01 e vice-versa. O mesmo ocorre com relação ao número de bits, a UART aceita a configuração de 7 ou 8 bits de transmissão, caso ela esteja configurada para a transmissão de apenas 7 bits, o último ou o primeiro bit do byte do buffer de transmissão será desconsiderado, dependendo da ordem em que estes serão enviados;
- Chamada de função externa para armazenamento dos dados transmitidos: o modelo chama uma função externa, passada como parâmetro durante a sua inicialização, para onde os bytes transmitidos serão enviados. Desta forma, o software driver de testes, pode configurar a UART e simular uma transmissão de dados, verificando se a UART está corretamente configurada pela análise dos dados efetivamente escritos;
- Análise dos pinos de entrada e saída: pela análise da configuração dos pinos de entrada e saída, o modelo decide se os dados devem ser efetivamente escritos na porta de saída. Caso um pino de saída esteja mal configurado, a UART não irá escrever os bytes corretos na porta de saída, de forma que o software de testes possa detectar esta má configuração;
- Indicação de transmissão: ao escrever o dado na porta de saída, o flag de término de transmissão é ligado, de forma que a aplicação possa monitorar o estado da UART, exatamente como ocorre na presença do hardware;
- Chamada de função externa para término de transmissão: ao término da transmissão de um byte, o modelo chama uma função externa, passada como parâmetro durante a sua inicialização, de forma a simular uma interrupção de transmissão. Observa-se que esta função apenas é chamada se os bits de configuração de interrupção estiverem corretamente configurados. Isto se aproxima bastante do que ocorre com a UART real, onde no momento em que o último bit está sendo enviado, a interrupção de final de transmissão é chamada de modo a permitir que a aplicação embarcada escreva um novo byte no registrador de transmissão.

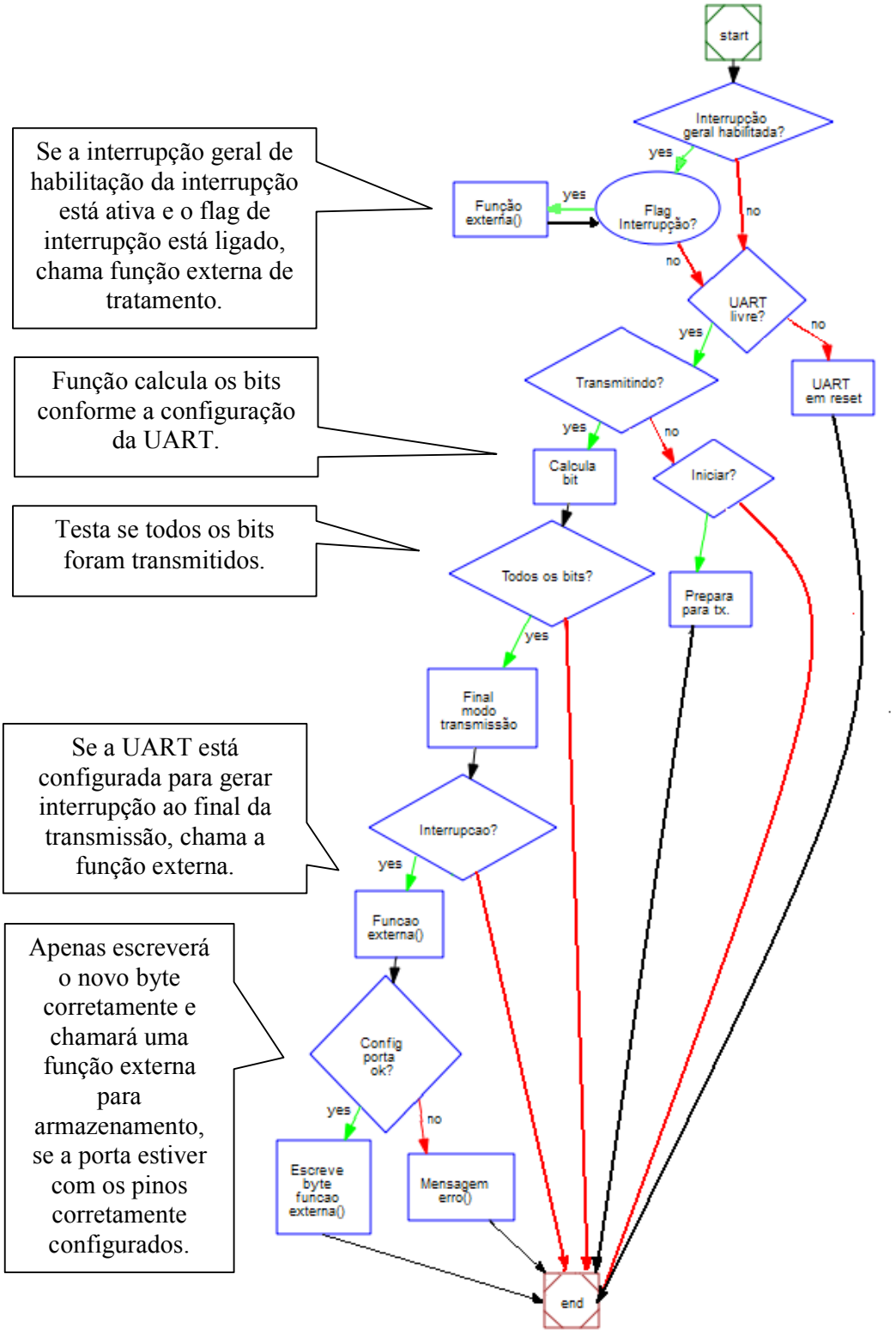


Figura 5.7: Diagrama de fluxo de controle simplificado do modelo da UART e detalhamento dos principais aspectos.

5.1.4 Portas de Entrada e Saída

Na família de microcontroladores MSP430, os pinos do microcontrolador podem ser configurados como entradas digitais, saídas digitais ou de função especial, onde os pinos possuem uma funcionalidade específica, geralmente associada a algum periférico de hardware, como uma entrada analógica de um conversor A/D ou a saída de um conversor D/A. A idéia de se modelar os pinos de entrada e saída tem por objetivo se fazer uma verificação da consistência da configuração atual, uma vez que a funcionalidade dos pinos é programada por registradores específicos mapeados em memória e livremente acessados.

A Figura 5.8 exibe quatro registradores de acesso aos pinos de entrada e saída para a porta P1 do microcontrolador. Observa-se que a escrita no registrador P1OUT não garante que o dado efetivamente será escrito na porta do microcontrolador, uma vez que isto dependerá da configuração do registrador P1DIR (que indicará se o pino é de entrada ou de saída) e da configuração do registrador P1SEL (que indicará se o pino se trata de um pino de entrada ou saída digital, ou se o pino está associado a algum periférico de hardware, como um conversor A/D ou como uma saída de referência analógica).

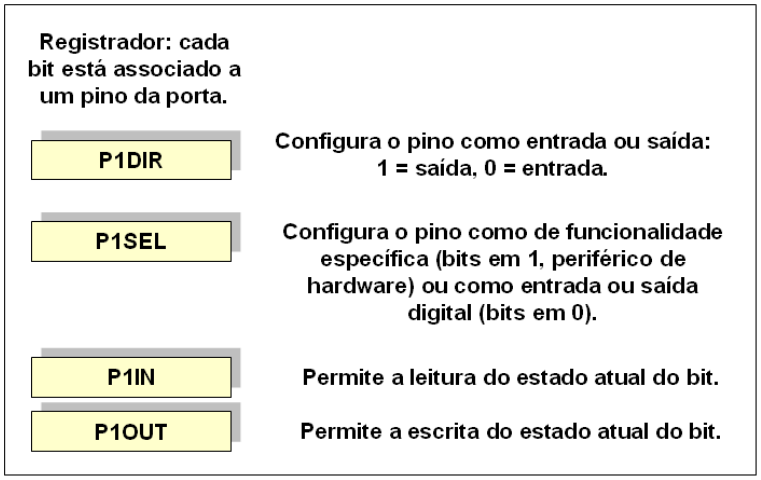


Figura 5.8: Registradores de acesso ao hardware para o controle dos pinos de entrada e saída dos microcontroladores da família MSP430 (MSP430, 2010).

Como exemplo, um fragmento de código do tipo P1OUT = 1, que pretende escrever 1 apenas no primeiro bit da porta P1, poderá não funcionar quando for executado no hardware, caso o pino 1 não esteja corretamente configurado como saída digital.

O modelo dos pinos de entrada e saída é extremamente simples e permite uma verificação bastante eficaz e intuitiva da correta configuração das portas. Internamente, são criados estados reais dos pinos de entrada e saída, estes estados simulam o estado externo do pino do microcontrolador. Sempre que o software embarcado escreve no registrador P1OUT, por exemplo, o modelo dos pinos de I/O irá verificar o estado dos registradores de configuração de direção e funcionalidade, de forma que o estado real do pino, existente dentro do modelo, apenas irá ser alterado caso o pino esteja configurado de forma adequada. O mesmo ocorre em uma situação de leitura, na qual o estado real

do pino, apenas irá ser transferido ao registrador P1IN, caso o pino esteja adequadamente configurado. Na Figura 5.9 é apresentado o diagrama principal da função model_io_execute() responsável pela execução do modelo dos pinos de entrada e saída.

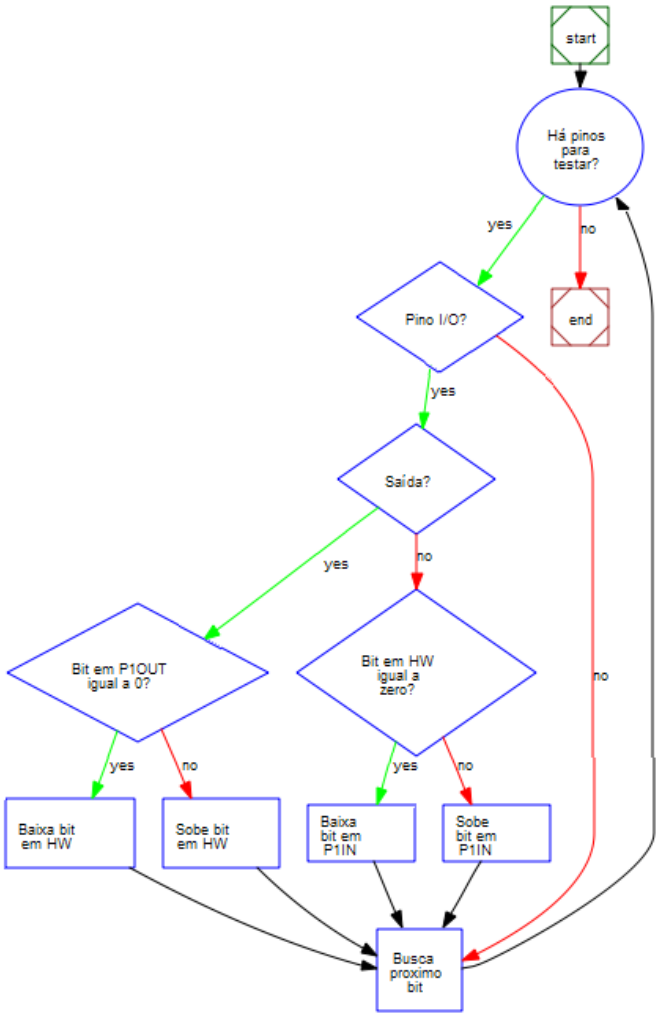


Figura 5.9: Diagrama de fluxo de controle da função model_io_execute(), modelo dos pinos de entrada e saída.

Executando-se o modelo dos pinos de I/O, scripts de teste bastante simples podem ser criados, visando a verificação da correta configuração dos pinos, conforme apresentado na Figura 5.10. A principal vantagem deste tipo de script diz respeito à forma intuitiva como ele é construído, onde, por exemplo, o teste de um pino configurado como saída pode ser feito por escritas e leituras dos bits de interesse.

```
void test_io_pins(void){

//Escreve um no pino 0 da porta P1
P1OUT |= 1;
//Executa o modelo
model_io_execute();
//Testa se o pino foi efetivamente escrito
if(!(P1IN & 1))
    ShowMessage("Fail");
    else
    {
        //Escreve zero no pino 0 da porta P1
        P1OUT &= ~1;
        //Executa o modelo
        model_io_execute();
        //Testa
        if(P1IN & 1)
         ShowMessage("Fail");
    }

}
```

Figura 5.10: Exemplo de script de teste de configuração de pinos de entrada e saída.

5.1.5 Timer

Uma vez que não é objetivo deste trabalho simular aspectos de tempo real, já que a execução da aplicação embarcada em um ambiente de desenvolvimento de software de aplicação não permite a contagem de tempos de execução em linguagem de máquina, perdendo-se, desta forma, a capacidade de análise temporal, o modelo do timer tem por objetivo apenas a execução de testes de unidade, de forma a permitir a verificação do software embarcado. Da forma como foi construído, o modelo permite a verificação do correto funcionamento do software embarcado através de casos de teste de software que utilizam os registradores de acesso ao hardware, exatamente do mesmo modo que a aplicação embarcada. As seguintes funcionalidades foram previstas no modelo do dispositivo de timer:

- Contagem de ciclos: de forma a permitir a análise dos tempos de geração das interrupções, cada chamada à função de execução do modelo do timer é considerada como um ciclo de processamento, onde o contador interno do timer é incrementado;
- Simulação de interrupções: ao ser inicializado, o modelo recebe como parâmetros o endereço de duas funções, chamadas conforme o modo de utilização do timer, ora no momento em que o contador interno do timer atinge o valor de comparação programado, ora quando este contador é reiniciado. Da mesma forma que o hardware físico, o modelo liga os bits de indicação de interrupção no registrador específico e, pela análise dos bits de

habilitação de interrupção, tanto globais, quanto específicos do timer, verifica se deverá chamar as funções externas de tratamento da interrupção;

- Reinício: o modelo implementa a funcionalidade de reinício de contagem, reinicializando as variáveis internas sempre que o bit de reinício, no registrador de controle do timer, é ligado;
- Simulação de quatro modos de operação: o modelo do timer é capaz de simular quatro modos diferentes de operação, são eles, parado, contagem crescente, contagem contínua e contagem crescente e decrescente, todos funcionando no modo de comparação. Outro modo de funcionamento do timer, presente no dispositivo físico, mas não implementado no modelo, é o modo de captura, entretanto, a construção deste modo de operação, onde pinos de entrada são associados a interrupções do timer pode ser implementado seguindo a mesma abordagem;
- Simulação do tamanho do contador: assim como no hardware físico, o modelo permite a configuração de quatro tamanhos pré-estabelecidos para o contador do timer, são eles: 8 bits, 10 bits, 12 bits e 16 bits. Dependendo da configuração atual, o modelo reinicia ou não a contagem, exatamente como no hardware físico.

A Figura 5.11 exibe o diagrama simplificado da função de execução do modelo construído do dispositivo timer, onde se pode verificar o principal fluxo de controle da função, descritos a seguir.

- Ao iniciar, os flags de interrupção são verificados, caso algum esteja ligado e as interrupções habilitadas, o modelo chama a função externa associada. Nesta etapa, tanto o flag global de habilitação de interrupções do microcontrolador, quanto os flags específicos de habilitação de interrupção do timer são testados e a função externa associada à interrupção apenas será chamada caso ambas as habilitações estejam ativas;
- Na sequência, o modelo verifica se a aplicação ligou o bit de reinicio do timer. Este é um bit específico do registrador principal de controle, se este bit estiver ligado, o timer reinicia a contagem e as variáveis internas de controle do modelo são reiniciadas;
- O próximo passo é a verificação da configuração do timer, o modelo continuará o processamento apenas se o timer estiver configurado para operar no modo de comparação;
- Uma vez configurado no modo de comparação, o modelo testa qual modo de operação está atualmente ativo. Foram implementados todos os quatro modos previstos de funcionamento: parado, crescente, contínuo e crescente/decrescente. Cada um destes modos é capaz de gerar interrupções e chamar funções externas, de acordo com as configurações existentes.

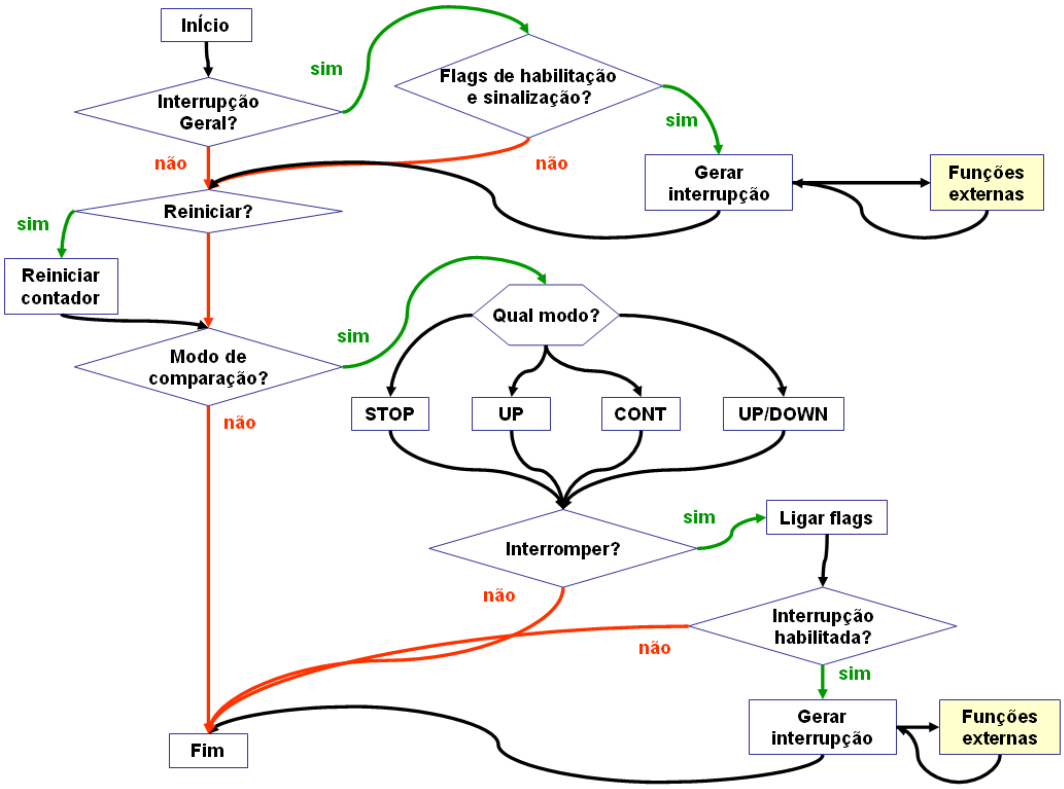


Figura 5.11: Diagrama simplificado de operação da função de execução do modelo do timer.

Uma forma simples de se testar o correto funcionamento do software embarcado que faz uso de timers é exibida no fragmento de código da Figura 5.12. Neste código, uma variável global que simula a contagem de tempos de processamento é criada (simul_clock) e incrementada dentro de um laço com um tempo limite de execução. Neste mesmo laço a função de execução do modelo do dispositivo de timer é chamada, observa-se que o modelo do timer tem como referência de tempo as chamadas a esta função. Na inicialização do modelo, duas funções criadas dentro do software driver (tb_ccr0_int e tb_int) são passadas como parâmetro, estas funções são as funções de tratamento das interrupções geradas. Caso o software embarcado esteja configurando corretamente o timer, a função de interrupção esperada será chamada. Dentro da função de interrupção, duas variáveis são alteradas, a primeira armazena o exato momento em que a função foi chamada, pela cópia da variável de contagem de ciclos de processamento e a segunda contabiliza o número de vezes que a função foi chamada, visando o teste de múltiplas execuções. Observa-se que a análise da variável de contagem de ciclos de clock permitirá saber exatamente o tempo de ocorrência da interrupção. A função de teste apenas retornará sucesso se a função de interrupção esperada tiver sido chamada exatamente o número esperado de vezes e no tempo configurado.

```
//Tempo inicial
int simul_clock = 0;
//Controle de chamada de interrupcoes
int clock_ccr0 = -1; //clock da chamada
int ccro_n = 0; //número de chamadas

int clock_tb = -1; //clock da chamada
int tb_n = 0; //número de chamadas

//Tratamento das interrupcoes
void tb_ccro_int(void){ clock_ccro = simul_clock;   ccro_n ++; }
void tb_int(void){  clock_tb = simul_clock; tb_n ++; }

int testar_timer(void){
//Inicilização do modelo do timer
  model_timer_start(/* função externa */tb_ccr0_int,
                    /* função externa */ tb_int);
//Aplicação embarcada
  TBCCTL2 = 0;               // nao utilizado
  TBCCTL1 = 0;               // nao utilizado
  TBCCR0  = SAMPLING_PERIOD; // Freq. de amostagem 1ms
  TBCCTL0 = CCIE;            // TBCCR0 habilitado por interrupcao
//Habilitação da interrupção global
SREG |= GIE;
//Variável global para simulação de clock
simul_clock = 0;
while(simul_clock++ < (SAMPLING_PERIOD+100))
    //Executa modelo
    model_timer_execute();

//Analise de resultados
//Apenas retorna 1 (sucesso) se:
//a interrupcao ocorreu em SAMPLING_PERIOD
//e o numero de interrupcoes for 1
return((clock_ccro == SAMPLING_PERIOD)&&(ccro_n == 1));
}
```

Figura 5.12: Exemplo de fragmento de código para teste da correta configuração do modelo do dispositivo de timer.

5.2 Análise do Software

O software desenvolvido foi analisado com o uso da ferramenta comercial de análise de código Understand (SCITOOLS, 2010), onde foram obtidas diversas métricas de qualidade de software, tais como número de entradas e saídas, aninhamento, e complexidade ciclomática, conforme dados resumidamente apresentados na Tabela 5.1. A análise destas métricas permite ao desenvolvedor redesenhar módulos ou funções que apresentem complexidade ciclomática exagerada (MCCABE, 1976), com quantidade excessiva de estruturas aninhadas ou muito acopladas aos demais módulos, por exemplo. Com relação à qualidade de software, é desejável que os módulos sejam independentes e encapsulados tanto quanto possível, de forma a possuir um

comportamento bem determinado e simples de ser testado. Adicionalmente, a ferramenta provê uma forma simples de medição da quantidade de comentários adicionados ao código, pela relação comentários/código.

Tabela 5.1: Algumas métricas obtidas com a ferramenta de visualização de código Understand.

Kind	Name	Inputs	Outputs	Code lines	Exe code lines	Paths	Exe statemt.	Cyclomatic Complexity	Nesting	Hds statemt.	HDS %	Ratio Comment/ Code	TYPE
Function	adc12_hds_channel	3	1	16	9	2	6	2	1	6	100,00	0.56	HDS
Function	adc12_hds_init	1	0	15	12	1	5	1	0	5	100,00	0.80	HDS
Function	comm_fill_tx_data	11	8	86	82	26	82	14	1	1	1,22	0.08	APS
Function	comm_hds_init	1	2	11	9	1	9	1	0	8	88,89	0.91	HDS
Function	comm_hds_interruptions	3	0	6	4	2	3	2	1	2	66,67	0.00	HDS
Function	comm_send_data	8	7	17	10	4	9	4	3	2	22,22	0.53	HDS
Function	comm_start	1	4	6	3	1	3	1	0	0	0,00	0.50	APS
Function	comm_stop	0	3	5	2	1	2	1	0	0	0,00	0.40	APS
Function	em_iir_filter	2	1	26	19	1	9	1	0	0	0,00	0.54	APS
Function	em_receive_samples	19	11	65	45	36	35	9	2	0	0,00	0.22	APS
Function	em_sampling_interruption	0	8	10	6	1	6	1	0	0	0,00	0.40	APS
Function	em_start_metering	1	10	18	16	1	16	1	0	0	0,00	0.22	APS
Function	flash_hds_erase_sector	1	2	21	11	8	10	4	1	7	70,00	0.33	HDS
Function	flash_hds_get_next_sector	2	1	12	10	4	7	4	1	0	0,00	0.00	APS
Function	flash_hds_write_data	5	1	33	18	24	17	6	2	9	52,94	0.36	HDS
Function	flash_verify_sector_erased	2	1	9	4	3	5	3	2	0	0,00	0.33	APS
Function	fnc_abs	3	1	5	3	2	3	2	1	0	0,00	0.00	APS
Function	fnc_inc_bcd_at_ptr	4	0	3	1	1	1	1	0	0	0,00	0.33	APS
Function	fnc_move_data32	2	0	4	2	2	3	2	1	0	0,00	0.00	APS
Function	led_control_state	2	0	17	8	4	8	4	3	2	25,00	0.18	HDS
Function	led_turn_on	2	1	7	5	2	4	2	1	2	50,00	0.57	HDS
Function	main	1	21	53	36	136	32	10	4	0	0,00	0.72	APS
Function	mst_get_current_state	2	1	4	1	1	1	1	0	0	0,00	0.00	APS
Function	mst_reset_state	1	1	3	1	1	1	1	0	0	0,00	0.00	APS
Function	mst_state_increment	4	3	31	18	8	15	8	3	0	0,00	0.35	APS
Function	pers_get_mem_pos	3	1	11	8	5	9	5	1	0	0,00	0.45	APS
Function	pers_restore_calib_data	2	1	25	13	7	11	5	3	0	0,00	0.32	APS
Function	pers_restore_config_data	2	1	25	13	7	11	5	3	0	0,00	0.32	APS
Function	pers_restore_meas_data	2	4	17	13	4	10	4	1	0	0,00	0.29	APS
Function	pers_store_meas_data	3	10	45	37	136	31	10	2	0	0,00	0.51	APS
Function	pers_try_change_status	4	1	7	3	3	3	3	2	0	0,00	0.00	APS
Function	pers_verify_storage_place	5	1	16	11	8	9	4	1	0	0,00	0.25	APS

A partir dos dados obtidos com a ferramenta Understand, visando-se caracterizar o software construído, o seguinte procedimento foi executado: em cada uma das funções desenvolvidas no código, foram contados o número de acessos aos registradores de hardware e comparados com o número total de sentenças executáveis. A contagem foi efetuada de forma automática, através de um script desenvolvido para tal. A razão entre os dois valores gerou uma nova métrica, associada ao nível de dependência do hardware da função analisada. Como um critério, funções cuja dependência do hardware era maior que 10% foram consideradas dependentes do hardware, enquanto funções com dependência igual ou inferior a 10% foram consideradas funções de aplicação. A idéia desse limiar era evitar que funções que fizessem um único acesso ao hardware dentre dezenas de operações, por exemplo, fossem consideradas dependente do hardware. Isolando-se, então, as funções desenvolvidas no software em dois grupos distintos: dependentes do hardware e de aplicação, fez-se uma análise de complexidade ciclomática. Os dados obtidos foram organizados na forma de histogramas e as probabilidades cumulativas foram obtidas, conforme exibido na Tabela 5.2.

Tabela 5.2: Histogramas e probabilidade cumulativa associada à complexidade de cada função.

Histograma das funções dependentes do hardware		
Cyclomatic	*Frequency*	*Cumulative %*
1	7	41,18%
2	3	58,82%
3	3	76,47%
4	3	94,12%
5	0	94,12%
6	1	100,00%
More	0	100,00%

Histograma das funções de aplicação		
Cyclomatic	*Frequency*	*Cumulative %*
1	10	33,33%
2	4	46,67%
3	2	53,33%
4	5	70,00%
5	3	80,00%
6	1	83,33%
7	0	83,33%
8	1	86,67%
9	1	90,00%
10	2	96,67%
11	0	96,67%
12	0	96,67%
13	0	96,67%
14	1	100,00%
More	0	100,00%

Os dados da Tabela 5.2 mostram que, para a aplicação embarcada desenvolvida, a complexidade ciclomática das funções dependentes do hardware é visivelmente menor que a complexidade das funções da aplicação embarcada. De fato, 94% das funções consideradas dependentes do hardware possuíam, no máximo, complexidade ciclomática 4, enquanto, na aplicação embarcada, este percentual é atingido para um nível de complexidade ciclomática igual a 10.

É claro que a forma como as funções são desenvolvidas afeta diretamente este tipo de análise, que é fortemente dependente do nível de encapsulamento do código, por exemplo. Porém, a idéia básica é mostrar que, em geral, aplicações embarcadas tendem a ser mais complexas que o software dependente do hardware, sob o ponto de vista do código construído. Isto advém da forma como o hardware é usualmente acessado, através de escritas e leituras a simples registradores mapeados em memória. Entretanto, quando se leva em conta a complexidade do hardware em si, que poderia ser visto como um processo de software executado o tempo todo em paralelo com os módulos dependentes do hardware, esta complexidade tornar-se-ia muito maior.

Finalmente, a idéia da análise executada foi verificar, de forma bastante simples, o nível de complexidade das funções dependentes do hardware e comparar com a complexidade das funções do software da aplicação embarcada, de forma a mostrar que ferramentas de análise de complexidade, quando analisam software dependente do hardware a partir do código fonte, não são capazes de diagnosticar corretamente o nível de complexidade deste software.

Em outra análise, fez-se uma comparação entre a quantidade de software dependente do hardware, comparado com a quantidade de software de aplicação. Os resultados mostraram que 84 % do software construído era software de aplicação e apenas 16 % pode ser considerado software dependente do hardware. Com os novos microcontroladores, capazes de armazenar uma quantidade muito maior de memória de

programa, esta relação tende a ser ainda maior, onde o software de aplicação passará a ocupar praticamente toda a memória reservada para código no microcontrolador.

Estes dados ressaltam a necessidade de ferramentas de teste capazes de testar tanto o software de aplicação, dado a sua complexidade, quanto o software dependente do hardware. Além disto, sempre que possível, a interação entre estes dois tipos de software deve ser considerada.

5.3 Casos de Teste

A aplicação dos modelos dos dispositivos de hardware permite a execução tanto de casos de testes de unidade, onde cada módulo ou parte do software é testada individualmente, quanto de casos de testes de integração, onde é possível verificar os aspectos relacionados à interação entre os diferentes módulos de software.

No entanto, conforme (SEO et. al, 2007), em software embarcado os testes funcionais e de integração são mais importantes e desafiadores do que os testes de unidade, devido à intensa interação entre software de diferentes camadas.

5.3.1 Execução dos Testes de Unidade

A aplicação da metodologia de testes proposta, juntamente com a execução dos modelos dos dispositivos de hardware, facilita bastante o projeto e execução de testes de unidade, uma vez que uma boa parte da tarefa já está pronta. Por exemplo, ao se desenvolver casos de teste de unidade, usualmente, um projetista deverá efetuar os seguintes passos básicos:

- Análise de requisitos: o projetista de teste estuda a especificação do módulo sob teste e desenvolve casos de teste que contemplem os requisitos de projeto;
- Construção do software driver: o software driver, que executará os casos de teste de unidade, deve ser construído de forma a poder excitar os módulos em teste com os valores de entrada específicos dos casos de teste em execução. Nesta etapa, todos os módulos que não dizem respeito ao teste corrente devem ser tratados de forma a permitir a execução do teste. Aqui entram stubs ou mocks, por exemplo;
- Análise de cobertura de teste: com o objetivo de verificar o quão abrangentes são os casos de teste criados, o código sob teste deverá ser analisado com uma ferramenta que permita a análise de cobertura de teste.

Considerando as etapas básicas de teste citadas, verifica-se que tanto a etapa de construção do driver de testes quanto a etapa de análise de cobertura são facilitadas pela aplicação da metodologia de testes proposta nesta dissertação. Ao construir o software embarcado prevendo-se tanto a execução na plataforma alvo quanto em ambiente de software de aplicação, o projetista está tornando o software extremamente portável, o que facilitará a sua execução em qualquer ambiente de testes que esteja a sua disposição. Além disto, é inegável que a construção de casos de teste em um ambiente de desenvolvimento de software de aplicação, sem todas as restrições impostas pela plataforma alvo, é muito mais simples, rápida e eficiente, uma vez que o projetista poderá fazer uso de toda a interface de entrada e exibição de dados disponível neste tipo

de ambiente, além da disponibilidade virtualmente infinita de memória e recursos de hardware.

Com relação aos modelos dos dispositivos de hardware, da forma como foram projetados, estão prontos para o suporte ao teste. O modelo do conversor A/D, por exemplo, é completamente controlado pelo software driver. Desta forma, uma infinidade de casos de teste que envolva a geração de amostras ou diferentes tipos de configuração pode ser muito facilmente construída, como no fragmento de código exibido no exemplo da Figura 5.13, onde um teste de unidade que verifica a função de aquisição de dados do conversor A/D é executado.

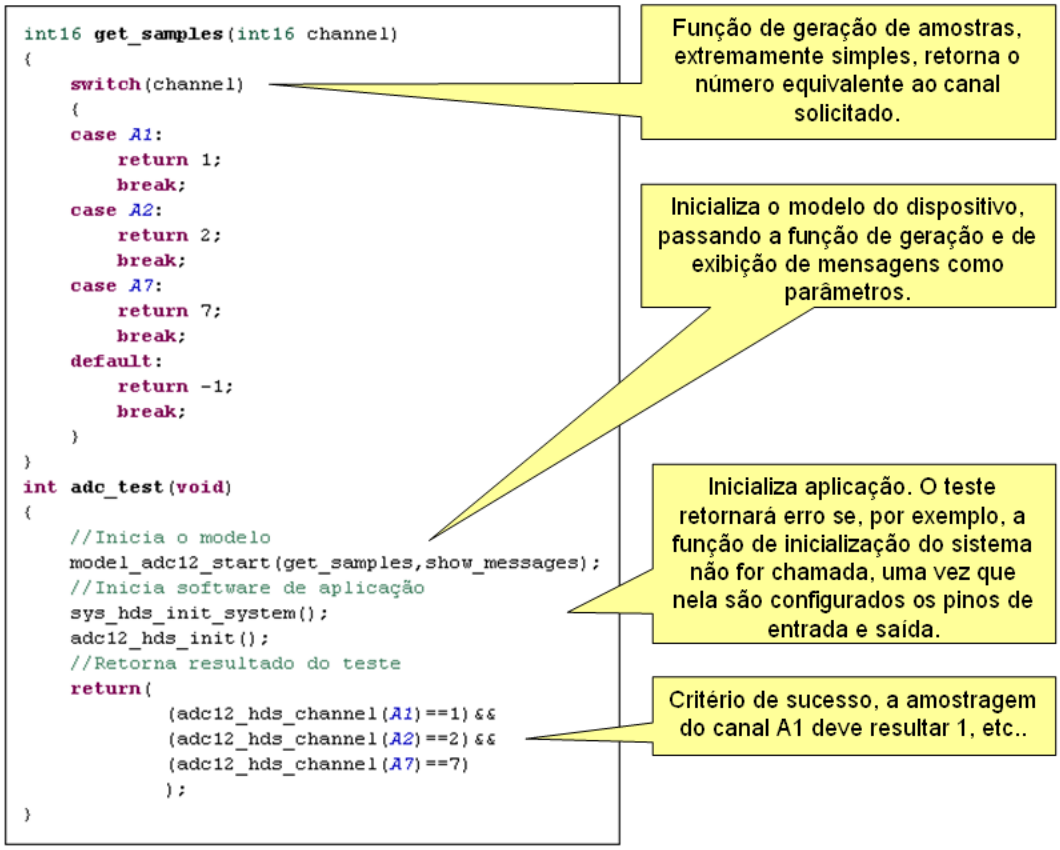
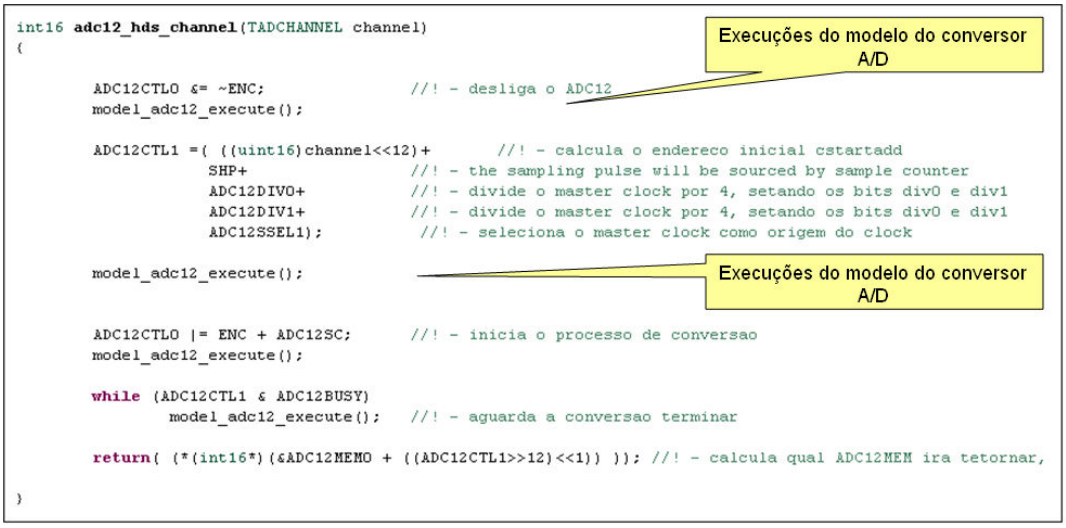


Figura 5.13: Exemplo de teste de unidade da função de aquisição do conversor A/D.

Pela análise da estrutura interna da função adc12_hds_channel (Figura 5.14), observa-se que o caso de teste efetivamente faz uma aquisição e percorre todos os caminhos internos do software de acesso ao hardware, exatamente como faria se estivesse sendo executado na plataforma alvo. Este tipo de caso de teste tem por finalidade a verificação da configuração do dispositivo, onde se testa a habilitação dos canais analógicos e seu correto mapeamento para os pinos de entrada.

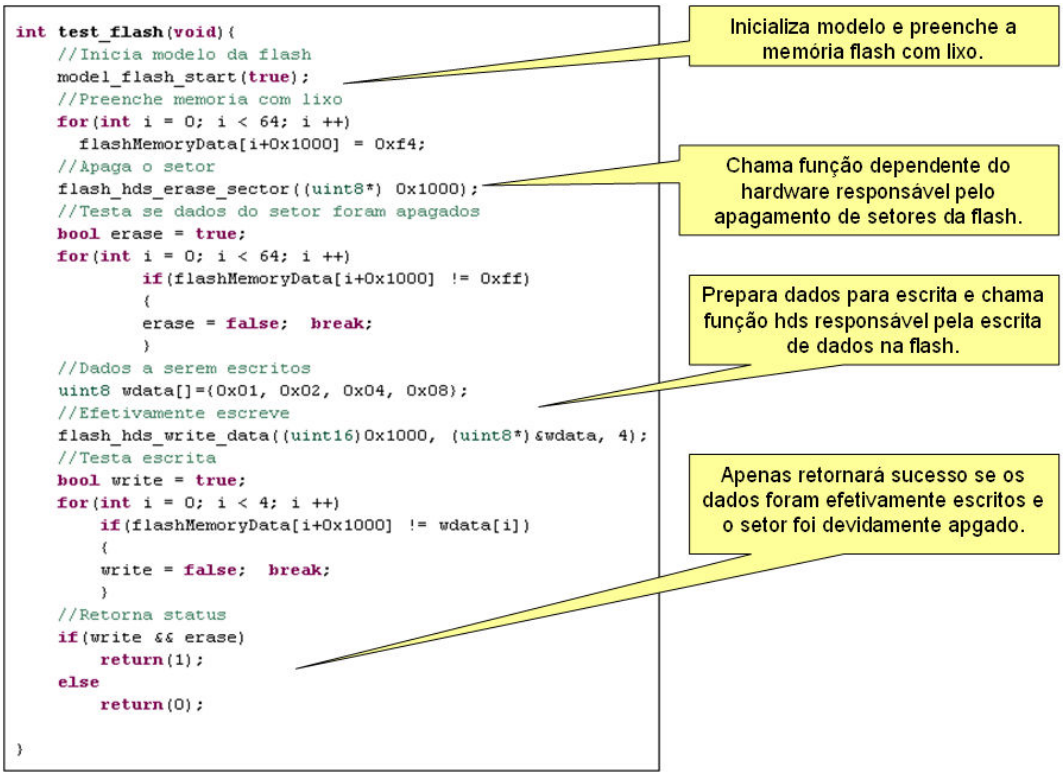



```
int16 adc12_hds_channel(TADCHANNEL channel)
{
        ADC12CTL0 &= ~ENC;              //! - desliga o ADC12
        model_adc12_execute();

        ADC12CTL1 =( ((uint16)channel<<12)+      //! - calcula o endereco inicial cstartadd
                    SHP+                //! - the sampling pulse will be sourced by sample counter
                    ADC12DIV0+          //! - divide o master clock por 4, setando os bits div0 e div1
                    ADC12DIV1+          //! - divide o master clock por 4, setando os bits div0 e div1
                    ADC12SSEL1);         //! - seleciona o master clock como origem do clock

        model_adc12_execute();

        ADC12CTL0 |= ENC + ADC12SC;     //! - inicia o processo de conversao
        model_adc12_execute();

        while (ADC12CTL1 & ADC12BUSY)
                model_adc12_execute();  //! - aguarda a conversao terminar

        return( (*(int16*)(&ADC12MEM0 + ((ADC12CTL1>>12)<<1)) )); //! - calcula qual ADC12MEM ira tetornar,

}
```

Figura 5.14: Detalhamento da função adc12_hds_channel.

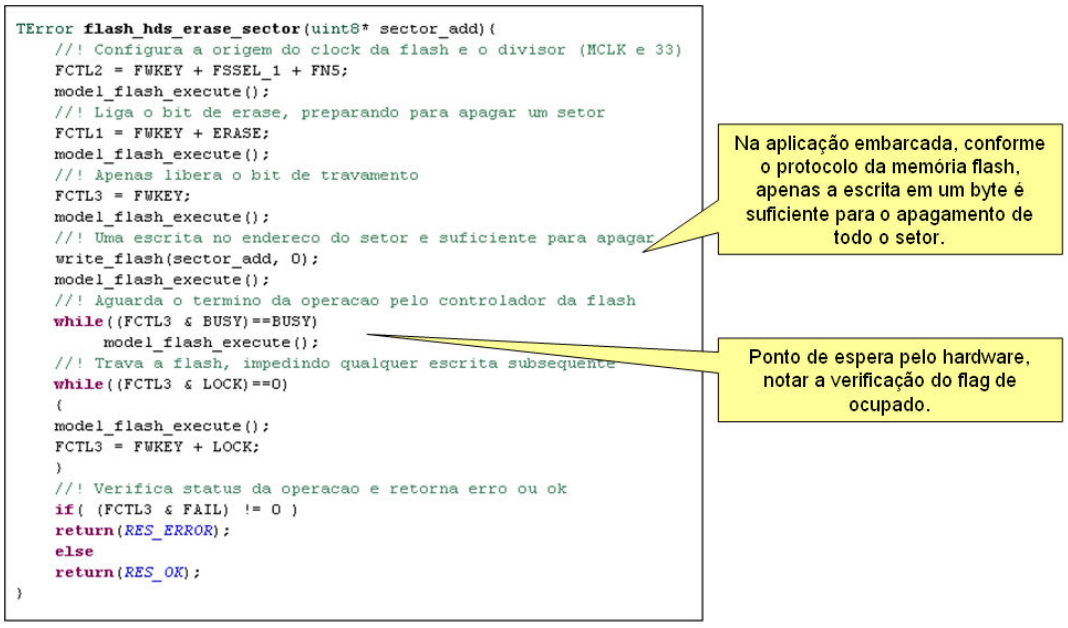
O modelo dos pinos de entrada e saída, permitirá a execução de casos de teste que, de outra forma, poderiam apenas serem executados com a presença do hardware físico, conforme já exibido na Figura 5.10, onde um dado é escrito em um registrador de saída (P1OUT) e verificado pela leitura de um registrador de entrada (P1IN). Neste caso, quem se responsabiliza pela transferência do dado do registrador de saída para o registrador de entrada é o modelo dos pinos de entrada e saída e isto apenas irá ocorrer caso os pinos estejam corretamente configurados. Finalmente, uma vez que o software driver possui acesso aos registradores internos do modelo, que guardam o estado dos pinos de entrada e saída, este pode, muito facilmente, gerar estímulos de entrada, alterando o estado real e executando funções da aplicação embarcada de modo a verificar seu comportamento.

Outro exemplo de criação de caso de teste de unidade capaz de ser executado tanto na plataforma alvo, quanto em ambiente de software de aplicação, pode ser visto no fragmento de código exibido na Figura 5.15, onde duas funções da aplicação embarcada que efetuam acesso à memória flash interna do microcontrolador são testadas. A primeira delas, flash_hds_erase_sector, é responsável por apagar um setor específico da memória, passado como parâmetro. A segunda função é uma simples função de escrita na memória flash, que recebe como parâmetros o endereço onde os bytes serão escritos, o endereço onde se encontram estes bytes e a sua quantidade. Ambas as funções dependentes do hardware foram criadas de forma a implementar o protocolo de comunicação da memória flash, conforme mostrado na Figura 5.16. Observa-se que, sem a execução do modelo da memória flash, estas funções não fariam sentido em um ambiente de software de aplicação e jamais executariam a sua real funcionalidade, como pode ser visto nos pontos destacados da figura. No primeiro ponto, o software efetua apenas a escrita de um valor zero no endereço passado como parâmetro, conforme o protocolo da memória flash, isto seria suficiente para apagar todo o setor, caso os registradores de hardware estejam corretamente configurados para o modo de apagamento. No segundo ponto em destaque, o software fica aguardando um flag de ocupado. Sem a execução dos modelos, este flag poderia jamais baixar, por exemplo, causando um travamento da aplicação.




```
int test_flash(void){
    //Inicia modelo da flash
    model_flash_start(true);
    //Preenche memoria com lixo
    for(int i = 0; i < 64; i ++)
      flashMemoryData[i+0x1000] = 0xf4;
    //Apaga o setor
    flash_hds_erase_sector((uint8*) 0x1000);
    //Testa se dados do setor foram apagados
    bool erase = true;
    for(int i = 0; i < 64; i ++)
            if(flashMemoryData[i+0x1000] != 0xff)
            {
            erase = false;  break;
            }
    //Dados a serem escritos
    uint8 wdata[]={0x01, 0x02, 0x04, 0x08};
    //Efetivamente escreve
    flash_hds_write_data((uint16)0x1000, (uint8*)&wdata, 4);
    //Testa escrita
    bool write = true;
    for(int i = 0; i < 4; i ++)
        if(flashMemoryData[i+0x1000] != wdata[i])
        {
        write = false;  break;
        }
    //Retorna status
    if(write && erase)
        return(1);
    else
        return(0);

}
```

Figura 5.15: Exemplo de teste de unidade de funções de acesso à memória flash interna do microcontrolador.




```
TError flash_hds_erase_sector(uint8* sector_add){
    //! Configura a origem do clock da flash e o divisor (MCLK e 33)
    FCTL2 = FWKEY + FSSEL_1 + FN5;
    model_flash_execute();
    //! Liga o bit de erase, preparando para apagar um setor
    FCTL1 = FWKEY + ERASE;
    model_flash_execute();
    //! Apenas libera o bit de travamento
    FCTL3 = FWKEY;
    model_flash_execute();
    //! Uma escrita no endereco do setor e suficiente para apagar
    write_flash(sector_add, 0);
    model_flash_execute();
    //! Aguarda o termino da operacao pelo controlador da flash
    while((FCTL3 & BUSY)==BUSY)
         model_flash_execute();
    //! Trava a flash, impedindo qualquer escrita subsequente
    while((FCTL3 & LOCK)==0)
    {
    model_flash_execute();
    FCTL3 = FWKEY + LOCK;
    }
    //! Verifica status da operacao e retorna erro ou ok
    if( (FCTL3 & FAIL) != 0 )
    return(RES_ERROR);
    else
    return(RES_OK);
}
```

Figura 5.16: Detalhamento da função flash_hds_erase_sector.

5.3.2 Execução dos Testes de Integração

A possibilidade de execução de casos de testes de integração, onde se pode verificar a interação entre diferentes camadas de software e entre software com diferentes aplicações, com todos os recursos de um ambiente de desenvolvimento de software de aplicação é, sem dúvida, um dos principais ganhos da metodologia de desenvolvimento e testes proposta nesta dissertação. Com o uso dos modelos dos dispositivos de hardware, os testes de integração gerados podem avançar tanto em direção ao hardware, onde rotinas dependentes do hardware são testadas, quanto permanecerem nas camadas de software de aplicação, se necessário. Entretanto, mais interessante, são os casos de teste de software de aplicação que, nas camadas inferiores, acessam os dispositivos de hardware e que, ao mesmo tempo, interagem com múltiplos módulos de software, conforme o esquema exibido na Figura 5.17.

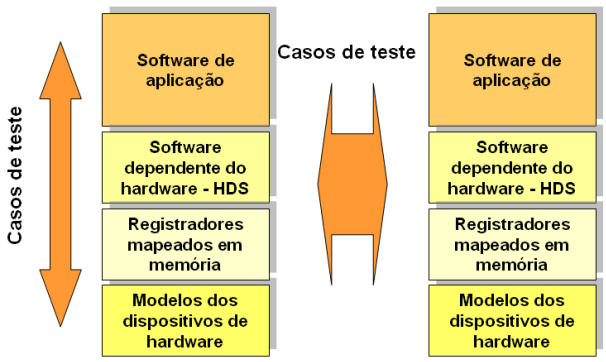


Figura 5.17: Abrangência de execução de casos de teste de integração com o uso dos modelos.

São muitos os exemplos deste tipo de caso de teste. De fato, para a aplicação do medidor eletrônico de energia, foi possível se executar e testar praticamente todo o software em ambiente de software de aplicação. Podem ser citados como exemplos deste tipo de testes, os casos de teste metrológicos, de operação em eventos de queda de energia e de longos tempos de execução.

Nos casos de teste metrológicos, o software sob teste é executado a partir da função main() e permanece em execução durante um período de tempo pré-determinado pelo software driver. Durante esta execução, não apenas os módulos metrológicos são exercitados, mas também os módulos de restauração de dados de calibração, configuração, geração de pulsos e faturamento, por exemplo. A verificação do sucesso ou falha do teste pode ser feita de uma forma extremamente natural, simplesmente pela leitura da quantidade de energia acumulada no medidor. Além disto, devido ao controle pelo software driver da função responsável pela geração das amostras lidas pelo conversor A/D, o software driver pode criar qualquer tipo de comportamento, tanto no sinal de tensão, quanto no sinal de corrente. Testes com sinais puros, com conteúdo harmônico, defasagens diversas, mesmo inversões de fase, podem ser executados automaticamente, de forma simples e rápida.

Um aspecto interessante da abordagem utilizada, diz respeito à forma como o software driver controla o tempo de execução da aplicação. Os dispositivos

microprocessados, por definição, possuem uma base de tempo, seja ela de origem interna ou externa. Em geral, a base de tempo primária, que é a freqüência de clock do dispositivo, deriva contagens de tempo secundárias, em uma freqüência mais lenta, isto é, usualmente, obtido com timers. No projeto do medidor eletrônico de energia, esta base de tempo secundária é a interrupção de amostragem, que executa a cada 1 ms. Deste modo, de forma simples, a contagem de amostras fornecidas ao software em teste, permite a determinação, de forma precisa, do tempo de execução do software e, finalmente, uma vez que é o driver de testes o responsável pelo controle das amostras fornecidas ao software sob teste, o monitoramento do tempo de execução se torna uma tarefa simples. O caminho percorrido pelos casos de teste metrológicos é exibido, de forma simplificada, na Figura 5.18, onde se pode observar o laço de controle do tempo de processamento.

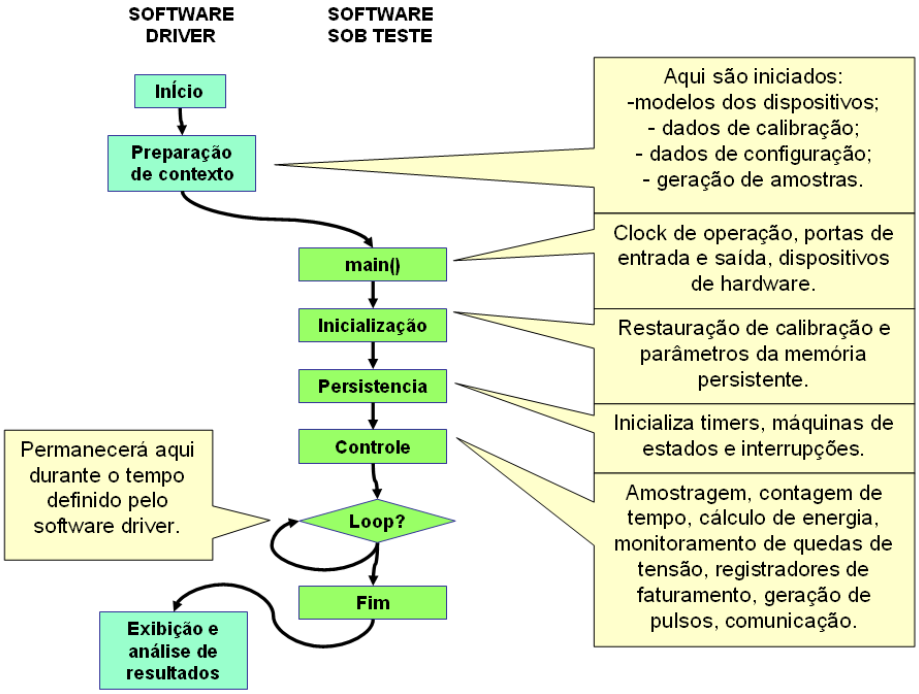


Figura 5.18: Abrangência de ensaio metrológico, realizado a partir do software driver de testes.

Ainda conforme o diagrama da Figura 5.18, observa-se que os casos de teste metrológicos, ainda no software driver, inicializam a memória de calibração do software do medidor. Estes dados são armazenados em flash, e restaurados pelo software através de leituras da memória flash interna, exatamente como quando executado na plataforma alvo. Casos de teste que verifiquem o comportamento na ocorrência de falhas podem ser, desta forma, muito facilmente construídos, pela simples alteração do conteúdo desta região de memória.

Da mesma forma que os casos de teste metrológicos, os casos de teste de monitoramento de quedas de energia percorrem um caminho bastante similar dentro do software sob teste, sendo inicializados a partir da função main(). Estes casos de teste

permitem a verificação do software quando da ocorrência de uma interrupção da tensão de alimentação. Porém, diferentemente dos casos de teste metrológicos, cujo controle do tempo de execução é feito pelo número de amostras fornecidas, os casos de teste de monitoramento de quedas de energia utilizam a função de geração de amostras para gerar as quedas. O funcionamento é simples: a partir de um tempo pré-determinado, a função de fornecimento de amostras ao modelo do conversor A/D passa a enviar amostras equivalentes ao nível de tensão desejado, no caso, um nível considerado queda de energia. Em funcionamento correto, o software sob teste deverá detectar esta queda de tensão e percorrer os caminhos necessários para o salvamento dos dados de faturamento, desta forma, todo o ciclo de monitoramento, detecção e atuação em caso de quedas de energia pode ser verificado (Figura 5.19).

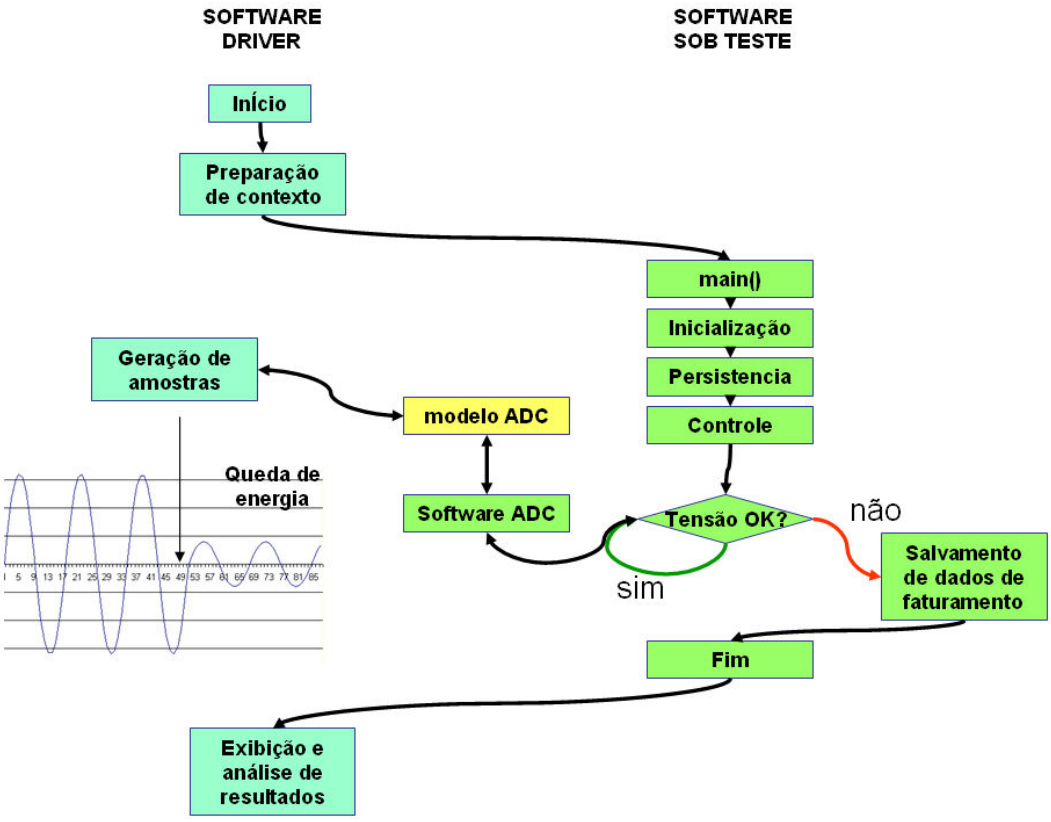


Figura 5.19: Casos de teste de verificação de comportamento durante queda de energia.

Finalmente, da mesma forma que nos casos de testes metrológicos, diferentes níveis de tensão e formas de onda podem ser inseridos de forma bastante simples. Além disto, o tempo em que a tensão é reduzida é controlado com precisão pelo software driver de testes, de forma a permitir o teste de quedas de tensão em diversos cenários de teste.

Outros casos de teste interessantes são os casos de teste de longa duração, não representados em diagramas devido à semelhança, na sua forma de funcionamento, com os casos de teste metrológicos. Neste tipo de caso de teste a idéia é verificar o funcionamento do software por longos períodos de funcionamento, cujo tempo pode ser

medido em minutos, ou horas. A razão para este tipo de teste vem do fato que existem falhas de software que não se tornam evidentes durante períodos curtos de execução, mas que apenas se manifestam após longos períodos de funcionamento, como overflows em variáveis de acumulação e mau funcionamento em módulos de armazenamento de dados, por exemplo. A aplicação dos modelos permite realizar, em curtos espaços de tempo, simulações que, de outra forma, levariam horas.

No software do medidor de energia, por exemplo, testes de longa duração verificaram o comportamento do software simulando horas de execução. Foi observada a detecção de estado "sem energia", a escrita dos dados de faturamento a cada hora cheia de funcionamento, executada de forma circular, e se o software continuava medindo corretamente após longos períodos.

Os exemplos citados têm por objetivo exemplificar a flexibilidade e facilidade de construção e execução de casos de teste de software em ambiente de software de aplicação, quando os modelos estão em execução.

5.3.3 Execução dos Testes de Sistema

Os casos de teste de sistema, assim como em métodos tradicionais de teste, deverão ser executados na plataforma alvo, uma vez que demandariam modelos dos dispositivos de hardware e do contexto de execução extremamente complexos, o quê fugiria do objetivo deste trabalho.

Além disto, não importa qual seja a abordagem de desenvolvimento utilizada, dificilmente se poderia fugir da execução de testes no dispositivo final. A idéia de metodologias de desenvolvimento e teste como esta, não é excluir o hardware dos procedimentos de teste, mas tornar possível a construção de grande parte do projeto sem o hardware físico, de forma que, quando o software for executado neste, já esteja suficientemente maduro e confiável.

5.4 Cenários de Teste

Um cenário de teste pode ser definido como sendo um ambiente especialmente preparado para a execução de um caso de teste. O cenário possui os mecanismos necessários para atender às necessidades do caso de testes a ser executado. Por exemplo, ao serem criados casos de teste metrológicos, independentemente de qual caso de testes será construído, será necessária a construção da função que fornece as amostras ao modelo do conversor A/D, chamar as rotinas de inicialização deste dispositivo, assim como do modelo. Observando características em comum de diferentes casos de teste, pode-se pensar em construir cenários de teste que atendam a estas características.

Conforme mencionado no capítulo 2 desta dissertação, a organização dos casos de teste em cenários facilita a execução dos testes, uma vez que:

- Na situação em que um caso de testes se enquadre em um cenário pré-existente, apenas as características específicas do caso de teste em desenvolvimento terão que ser construídas;
- Uma vez que o software driver de testes, também é um software que deverá ser validado, o reaproveitamento de cenários de teste diminui o tempo de desenvolvimento de novos casos de teste, assim como possibilita um aumento da qualidade dos casos de teste gerados;

- Usualmente, dentro do mesmo segmento de aplicação, os casos de teste serão bastante semelhantes, o que estimula a organização dos casos de testes em cenários.

No software do medidor eletrônico de energia construído para esta dissertação, seis diferentes cenários de teste foram criados:

- Requisitos metrológicos: este cenário prepara o software embarcado para a execução de casos de teste metrológicos, onde diferentes formas de onda de tensão e corrente são inseridas no equipamento de forma a verificar a exatidão da energia medida. Foram criados casos de teste para a verificação da medição de energia ativa e reativa nos quatro quadrantes de operação (ativa direta, ativa reversa, reativa indutiva direta, reativa indutiva reversa, reativa capacitiva direta e reativa capacitiva reversa). O cenário possibilita uma forma extremamente simples de se configurar as formas de onda a serem geradas para o modelo do conversor A/D;

- Energização e desligamento: este cenário provê a estrutura necessária para testes de energização e desligamento do equipamento. Usualmente, equipamentos de medição possuem tensões mínimas que precisam ser respeitadas, tanto para operação quanto para desligamento. Ao ser energizado, o equipamento apenas deverá efetivamente entrar em operação caso a tensão de alimentação seja superior à mínima de energização. Analogamente, durante a operação, caso a tensão caia para um valor inferior ao mínimo de operação, o equipamento deverá desligar, salvando os dados de faturamento, caso necessário, e gerando uma queda de energia;

- Injeção de falhas no registro dos dados de faturamento: neste cenário, os casos de teste alteram os registros dos dados de faturamento, de forma a verificar a capacidade de recuperação. É comum, em instrumentos de medição, o armazenamento redundante de dados, onde, caso uma falha tenha ocorrido e não seja possível recuperar um registro, o sistema tente recuperar um registro válido anterior ou um registro redundante;

- Execução de longa duração: neste cenário, longos tempos de simulação permitem a verificação de requisitos não tão freqüentes. Por exemplo, um equipamento de medição de energia deverá salvar os dados de faturamento em memória não volátil sempre que houver uma queda de energia, entretanto, por segurança, é interessante que o equipamento armazene estes dados de tempos em tempos, de modo que, se no momento da queda de energia, que é uma situação crítica para o equipamento, o registro for escrito com alguma falha, o equipamento possa recuperar o registro salvo automaticamente, momentos antes. Isto reduz também o risco de perda de um grande período de faturamento. Porém, uma vez que, para os microcontroladores modernos, a memória não volátil usualmente é do tipo flash, este período de salvamento automático não pode ser muito freqüente, de forma a não reduzir a vida útil do projeto, dada a limitação das memórias flash com relação ao número de escritas;

- Salvamento e restauração de dados: neste cenário, casos de teste exercitam os módulos de salvamento e recuperação de dados de faturamento. Estes

módulos são considerados críticos uma vez que envolvem a quantidade de energia medida que será cobrada dos consumidores;

- Detecção de carga pequena: equipamentos de medição devem ser capazes de iniciar a medir a partir de uma corrente mínima, chamada corrente de partida. Cargas que resultem em correntes inferiores à corrente de partida não precisam ser medidas. Neste estado, diz-se que o equipamento entra em vazio, estado no qual a energia não precisará ser cobrada. Este cenário de testes tem por objetivo a inserção de correntes inferiores à corrente de partida e verificação da indicação de estado de vazio pelo medidor, o que deve ser feito em um processo diferentemente do executado no cenário de testes metrológicos. Devido à corrente ser muito pequena, não é possível a sua detecção pela simples análise de amplitude do sinal amostrado, ao menos para equipamentos de baixo custo. Devido a isto, o que se faz é analisar a quantidade de energia medida pelo equipamento, o que apenas pode ser feito após um determinado tempo de observação.

Os cenários desenvolvidos para esta dissertação foram organizados na forma de funções, de forma que diferentes casos de teste pudessem ser gerados simplesmente pela chamada da função do cenário correspondente, conforme o cenário para testes metrológicos cujo fragmento de código é exibido na Figura 5.20.

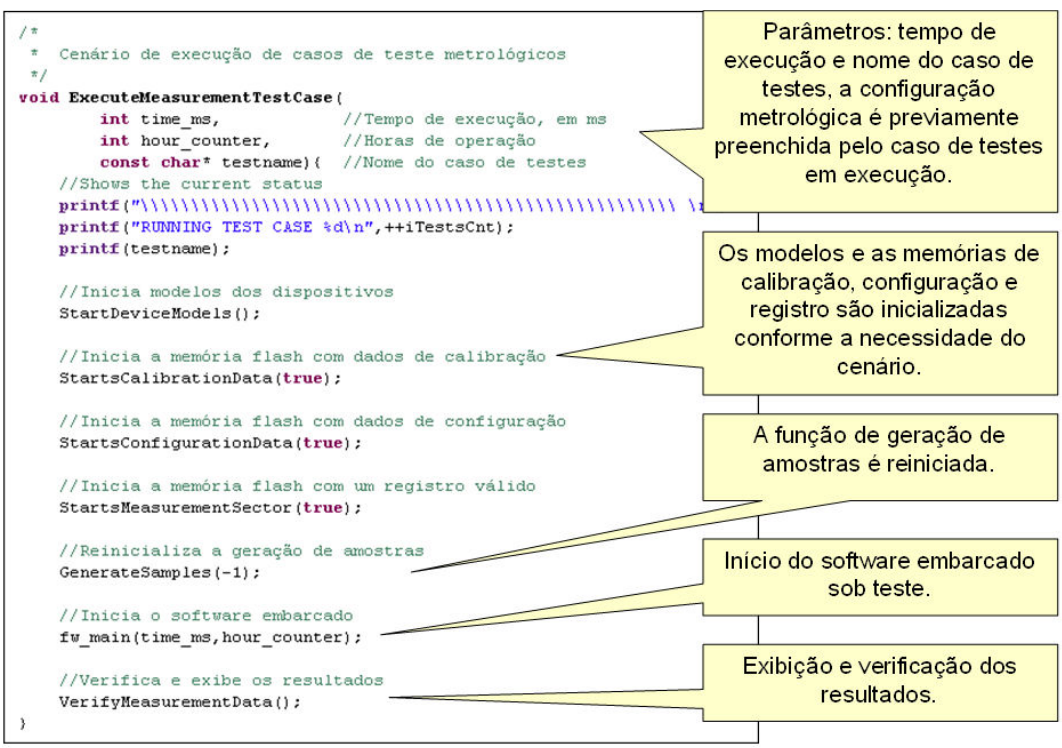


Figura 5.20: Cenário de testes para análise de requisitos metrológicos.

5.5 Análise de Cobertura

Visando validar a metodologia de desenvolvimento e testes proposta nesta dissertação, a análise de cobertura foi efetuada utilizando-se uma ferramenta de testes comercial, voltada a software de aplicação, chamada Coverage Validator (COVERAGE, 2010). A análise de cobertura foi utilizada como uma forma iterativa de aperfeiçoamento dos casos de teste criados. Ao se executar os casos de teste, as parcelas de software não cobertas eram analisadas e, se necessário, os casos de testes eram aperfeiçoados de forma a cobrir estes trechos de código. Há casos em que o software simplesmente existe por um descuido durante o processo de programação e não está associado a qualquer requisito válido de projeto. Nestes casos, o trecho de código deve ser simplesmente eliminado do software, não sendo necessário qualquer tipo de alteração nos casos de teste, conforme exibido no diagrama da Figura 5.21. Como já mostrado nesta dissertação, a ferramenta utilizada possui uma interface bastante amigável, onde o responsável pela execução dos testes pode, muito facilmente, enxergar todos os pontos do software ainda não cobertos pelos casos de teste. Uma das vantagens deste tipo de ferramenta é tornar a criação de casos de testes uma tarefa bastante similar à construção de código propriamente dito, onde, analogamente, em vez de se executar o software, se executa o caso de teste e, em vez de se depurar a funcionalidade, se verifica a sua abrangência.

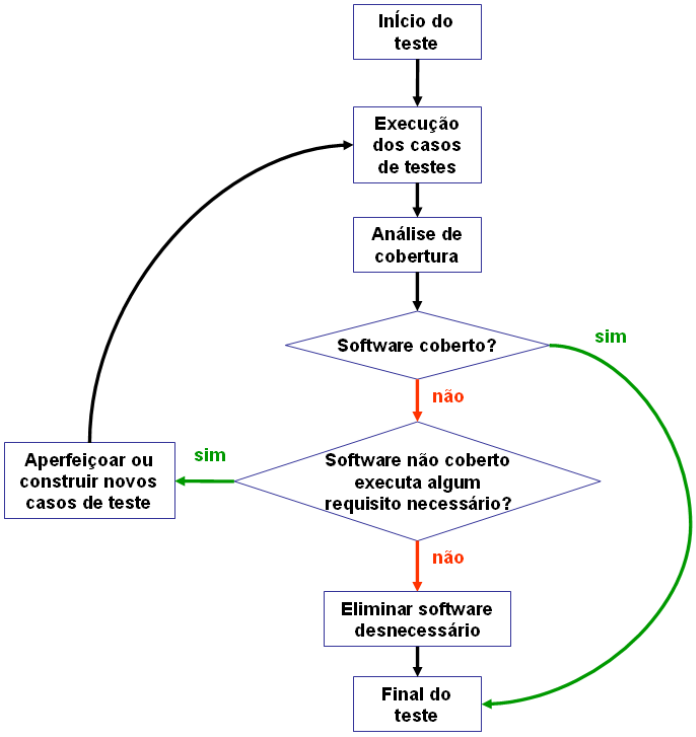


Figura 5.21: Evolução dos casos de teste a partir da análise de cobertura.

A aplicação dos casos de teste construídos, organizados em cenários de teste, permitiu a cobertura de grande parte do software embarcado construído, conforme pode ser visto nos dados da cobertura obtida mostrados na Tabela 5.3, gerados automaticamente pela ferramenta Coverage Validator. Em muitos módulos de software, a cobertura obtida chegou a 100%, sendo que o módulo com menor cobertura obteve 76% de cobertura. Ressalta-se que a cobertura dos testes poderia ainda ser ampliada, caso novos casos de teste fossem gerados, uma vez que grande parte do software se tornou completamente funcional, mesmo quando executado em ambiente de software de aplicação.

Tabela 5.3: Análise de cobertura funcional da execução dos casos de teste construídos, obtida com o software Coverage Validator.

File	Num L...	Num Visited	Visit Count	% Visited
Totals	**600**	**559**	**208.227.002**	**93,17%**
energy_m	81	81	13.711.270	100,00%
main.cpp	39	39	2.781.983	100,00%
ledcontr	14	14	2.779.893	100,00%
function	11	11	2.420.978	100,00%
communic	107	107	361.668	100,00%
adc12.cp	18	18	33.371.371	100,00%
voltage_monitor.cpp	15	15	1.693.284	100,00%
pulse_en	37	35	5.253.482	94,59%
system.c	60	54	3.090.462	90,00%
flash.cp	60	54	2.404	90,00%
persiste	102	86	4.867	84,31%
model_ad	35	29	141.828.057	82,86%
meter_st	21	16	927.283	76,19%

5.6 Limitações do Método Proposto

Devido às limitações dos modelos dos dispositivos de hardware e à compilação e execução do software embarcado em ambiente de software de aplicação, não é possível verificar, com esta metodologia, aspectos referentes a tempo real, uma vez que o software compilado e executado no PC será completamente diferente, com relação à sua estrutura interna, do software compilado e executado na plataforma alvo. Eles apenas serão semelhantes sob o ponto de vista funcional, ou seja, no que diz respeito às tarefas que executam. Dito isto, todos os aspectos críticos com relação a tempos de processamento e gerenciamento de interrupções, por exemplo, deverão ser validados na plataforma alvo, uma vez que o software executado em ambiente de aplicação, juntamente com os modelos dos dispositivos de hardware permitem a verificação da correta configuração do microcontrolador e apenas a simulação de seu comportamento sob o ponto de vista funcional.

Finalmente, os seguintes aspectos podem ser citados como limitações da metodologia de desenvolvimento e testes proposta:

- Uso de pilha e alocação de memória: aspectos que dizem respeito à quantidade de memória RAM utilizada e o seu gerenciamento, como espaço alocado em pilha, por exemplo, estão diretamente associados ao compilador da plataforma alvo e à sua configuração. Por exemplo, não é possível a verificação da alocação de pilha durante a execução do software em ambiente de software de aplicação, uma vez que, estruturalmente, o software executando em ambiente de aplicação é completamente distinto com relação ao software executado na plataforma alvo. Sendo assim, é fundamental que, na plataforma alvo, se faça este tipo de análise;
- Aninhamento e gerenciamento de múltiplas interrupções: os modelos dos dispositivos de hardware, apesar de permitirem alguns testes de unidade relacionados à geração de interrupções, não são capazes de simular o comportamento real com relação ao problema de aninhamento de interrupções (ocorrência de uma interrupção quando outra interrupção já está sendo atendida), uma vez que este tipo de teste apenas poderia ser feito com processos paralelos, como threads, que fogem do escopo deste trabalho;
- Gerenciamento de dispositivos críticos com relação à temporização: neste grupo pode se enquadrar, por exemplo, o controle de dispositivos como um watchdog, onde o seu correto funcionamento está extremamente associado aos tempos de execução do software embarcado. O teste deste tipo de dispositivo é uma tarefa que demanda uma análise do projeto no hardware definitivo;
- Tamanho do código gerado: não há como relacionar o tamanho do código gerado no ambiente de software de aplicação com o tamanho do código gerado pelo compilador da plataforma alvo, por razões óbvias, como sistema operacional e arquitetura, por exemplo;
- Tempos exatos de processamento: funções críticas com relação ao tempo de processamento, como função executadas dentro de interrupções, deverão ser verificadas no compilador da plataforma alvo, onde é possível a contagem exata do número de instruções executadas em linguagem Assembly.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou uma metodologia de desenvolvimento e testes de software embarcado com o objetivo de se permitir que grande parte da tarefa de desenvolvimento e teste fosse executada em um ambiente de desenvolvimento de software de aplicação. Neste ambiente, o desenvolvimento é pensado desde o início do projeto visando à qualidade do teste, assim caracterizando esta metodologia como uma técnica DFT.

Na dissertação, foram explicados métodos práticos de como lidar com as incompatibilidades comumente encontradas quando se tenta executar um software embarcado em um computador de uso geral. Além disto, a construção dos modelos dos dispositivos de hardware e sua execução no ambiente de aplicação permitiram que o software embarcado, mesmo quando executado em um PC, sem a presença do hardware, tivesse um comportamento muito próximo ao verificado quando executado no hardware físico real, plataforma alvo. A aplicação da metodologia proposta permitiu a execução de casos de teste, em ambiente de software de aplicação, que envolvessem tanto o software embarcado dependente do hardware, quanto o software da aplicação embarcada, de forma integrada, exatamente como ocorreriam em testes na plataforma alvo, sem a necessidade da construção de stubs que isolassem o software dependente do hardware para cada módulo em teste. Além disto, a simulação funcional dos dispositivos de hardware, pelos modelos, tornou os casos de teste executados no software embarcado, em ambiente de software de aplicação, compatíveis com o ambiente final de testes, na plataforma alvo.

Contextualizando o trabalho, no capítulo 2, se fez um apanhado dos principais aspectos referentes ao teste de software embarcado, onde foram cobertos conceitos de geração de casos de teste, agrupamento em cenários, criação de dublês de teste e aplicação de ferramentas e métodos de auxílio à execução do teste, utilizados no trabalho desta dissertação. Ainda naquele capítulo, foi ressaltada a importância da execução de testes que envolvessem as interfaces de software embarcado, como mostrado em (SEO et. al, 2007), uma vez que, dada a forte interconexão do software embarcado com o software dependente do hardware, muitas vezes o teste destas camadas, isoladamente, se torna impraticável. Daí se vê a necessidade da construção de stubs, mocks, ou modelos, que podem ser tratados como mecanismos que têm por objetivo a viabilização prática do teste.

No capítulo 3 foram apresentados os aspectos práticos da execução do método proposto, onde o detalhamento da forma usual com que são acessados periféricos de hardware existentes em microcontroladores e a forma como os modelos dos dispositivos de hardware são acessados, permitiu o entendimento de como, pelo compartilhamento de memória entre os modelos dos dispositivos de hardware e o software executado em ambiente de software de aplicação, a troca de informações entre os modelos e a

aplicação embarcada ocorria, de forma similar à executada no hardware da plataforma alvo.

Nos capítulos 4 e 5 foi apresentado um estudo de caso, onde se aplicou a metodologia proposta no software embarcado de um medidor eletrônico de energia, especialmente desenvolvido para esta dissertação. Nesta aplicação cinco modelos de dispositivos de hardware foram construídos e aplicados com sucesso ao software desenvolvido. A execução prática do método proposto permitiu uma melhor avaliação de suas possibilidades e limitações, que contribuíram para o enriquecimento das informações apresentadas nesta dissertação.

É importante ressaltar que, tendo sido possível se executar o software embarcado em um PC, com funcionalidade a ponto de se viabilizar o teste do software em diversos níveis de abrangência, e com o uso de ferramentas de teste voltados a software de aplicação, considera-se que a aplicação do método mostrou viabilidade prática.

Argumenta-se, ainda, em defesa da construção dos modelos dos dispositivos de hardware, um aspecto crítico nesta metodologia. Considera-se que o tempo despendido para construção dos modelos, conforme já mencionado, pode ser visto como tempo gasto com documentação do projeto. Além disto, a construção dos modelos não implica aos programadores um caminho mais longo no entendimento e na construção do software dependente do hardware, uma vez que, neste processo, os programadores estudam o dispositivo e, mentalmente, modelam o seu funcionamento. Em uma abordagem tradicional, apenas o produto desta modelagem, ou seja, o código da aplicação embarcada, permaneceria na empresa, enquanto na abordagem deste trabalho, os modelos construídos são uma forma bastante eficiente de transferência de conhecimento.

Finalmente, este método está atualmente sendo executado, de forma ainda parcial, em um projeto comercial de um equipamento para medição eletrônica de energia, onde os resultados, até o presente momento, têm se mostrado extremamente satisfatórios. Neste projeto, está se conseguindo reduzir drasticamente o tempo de desenvolvimento do software de aplicação e, quando testado no hardware físico, este software tem apresentado resultados funcionalmente idênticos ao obtido em simulação. A título de comparação, em projetos anteriores, no mesmo segmento de aplicação, de forma a se validar o software embarcado, três softwares diferentes chegaram a ser construídos: um modelo abstrato para a aplicação, na ferramenta SciLab (SCILAB, 2010), um modelo da camada de aplicação, com o software Borland Builder (BUILDER, 2010), e o software da plataforma alvo, desenvolvido com o compilador final, IAR (IAR, 2010). A manutenção destes três ambientes de software se mostrou completamente ineficiente, uma vez que, com a evolução do projeto, não era viável mantê-los atualizados. Com isto, na detecção de um problema ou mau funcionamento no software embarcado, era muito difícil fazer o rastreamento do problema nos modelos construídos, dadas as diferenças existentes com relação ao software executado na plataforma alvo.

A partir da experiência no desenvolvimento deste trabalho, surgiram algumas idéias que poderiam ser empregadas em projetos futuros, brevemente comentadas na próxima seção.

6.1 Trabalhos Futuros

6.1.1 Um Novo Analisador Estático de Código

Conforme já mencionado nesta dissertação, ferramentas de análise estática de código não são capazes de extrair a real complexidade do software dependente do hardware, uma vez que, sob a óptica de tais ferramentas, este software pode ser visto como uma simples seqüência de leituras e escritas em posições de memória, sem qualquer complexidade (os registradores de acesso ao hardware).

Da mesma forma, os compiladores atuais, capazes de rastrear o código fonte construído em busca de uma série de prováveis erros na construção deste software, nada informam com relação aos registradores de acesso ao hardware, que, para eles, também são simples posições de memória.

A execução dos modelos, como analisadores de código, permitiria o diagnóstico de uma série de erros de programação no que diz respeito à funcionalidade dos dispositivos de hardware. Assim como um compilador gera um alerta caso o software esteja tentando utilizar uma variável não inicializada, por exemplo, um analisador estático que compreendesse o software dependente do hardware poderia gerar mensagens ao programador sempre que este estivesse tentando utilizar um dispositivo de hardware que, pelo código, não estivesse habilitado ou ligado. Em outro exemplo, a leitura de um pino mapeado como saída poderia gerar um alerta. Estes são exemplos simples, mas que retratam possibilidades.

6.1.2 Modelos Organizados em Classes (Orientação a Objetos)

Uma vez que o software, quando executado em ambiente de desenvolvimento de software de aplicação, está sendo executado em um ambiente orientado a objetos, nada impediria que os modelos dos dispositivos de hardware fossem modelados em classes. Desta forma, a função model_x_execute(), por exemplo, não precisaria existir caso os registradores de hardware fossem mapeados como propriedades, por exemplo, uma vez que o compilador automaticamente substituiria o registrador de hardware associado, por métodos de leitura e escrita, conforme a necessidade do modelo desenvolvido, onde estaria o software responsável pela funcionalidade do modelo desenvolvido.

6.1.3 Modelos Mapeados em Threads

Da mesma forma que seria possível a construção dos modelos em uma linguagem orientada a objetos, é possível imaginar a função de execução principal dos modelos como sendo o laço principal de uma thread, o que acarretaria em uma forma de execução muito similar ao que ocorre no hardware. Como na abordagem proposta nesta dissertação, estas threads compartilhariam a memória com a aplicação embarcada, de modo que, ao alterar um registrador de acesso ao hardware, esta alteração pudesse ser verificada e processada. Muitos aspectos precisariam ser considerados, como semáforos, por exemplo, mas, testes simples com esta técnica foram executados e alguns problemas imediatamente surgiram, principalmente com relação ao tempo em que cada processo permanecia em execução. Mas como estava fora do escopo do projeto, não se avançou neste sentido.

REFERÊNCIAS

ANDERSON, J. How To Produce Better Quality Test Software. **Proceedings** IEEE Instrumentation & Measurement magazine, 2005, v. 8, p. 34-38.

ASTREE. Analisador Estático. Disponível em <http://www.astree.ens.fr/>. Acesso em: fevereiro 2010.

BORLAND BUILDER Homepage. Disponível em <http://www.borland.com/br/products/cbuilder/index.html>. Acesso em: fevereiro 2010.

BRISOLARA, L.; KREUTZ, M.; CARRO, L. UML as front-end language for embedded systems design. In: Luis Gomes; Joao M. Fernandes. (Org.). **Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation**. Hershey: IGI Global, 2009, c. 1.

CANTATA++ Homepage. Disponível em <http://www.ipl.com/products/tools/pt400.uk.php>. Acesso em: fevereiro 2010.

CODE COMPOSER Homepage. Disponível em < http://focus.ti.com/docs/toolsw/folders/print/ccstudio.html>. Acesso em: fevereiro 2010.

CODE-SONAR OVERVIEW. Disponível em <http://www.grammatech.com/products/codesonar/GrammaTechCodeSonarOverview.pdf>. Acesso em: fevereiro 2010.

COUSOT, P. et. al. Varieties of Static Analysers: A Comparison with Astrée. In: FIRST JOINT IEEE/IFIP SYMPOSIUM ON THEORETICAL ASPECTS OF SOFTWARE ENGINEERING, TASE, 2007. Shanghai. **Proceedings** Washington: IEEE Computer Society, 2007, p. 3-20.

COVERAGE VALIDATOR Homepage. Disponível em <http://www.softwareverify.com/cpp/coverage/index.html>. Acesso em: fevereiro 2010.

ENGBLOM, J.; GIRARD, G.; WERNER, B. Testing Embedded Software Using Simulated Hardware. In: 3RD EMBEDDED REAL-TIME SOFTWARE CONFERENCE, ERTS, 2006.

ESSER, M.; STRUSS, P. Fault-model-based Test Generation for Embedded Software. In: 20TH INTERNATIONAL JOINT CONFERENCE ON ARTIFICAL

INTELLIGENCE, 2007. Hyderabad. **Proceedings** San Francisco: Morgan Kaufmann Publishers Inc, 2007, p. 342-347.

FOWLER, M. Mocks Aren't Stubs. Disponível em < http://martinfowler.com/articles/mocksArentStubs.html >. Acesso em: fevereiro 2010.

GREENE, B. Agile Methods Applied to Embedded Firmware Development. In: AGILE DEVELOPMENT CONFERENCE, 2004, Salt Lake City.

GUAN, J.; OFFUTT, J.; AMMANN, P. An Industrial Case Study of Structural Testing Applied to Safety-critical Embedded Software. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING, ISESE, 2006. Rio de Janeiro. **Proceedings** New York: ACM Press, 2006, p. 272-277.

IAR SYSTEMS Homepage. Disponível em <http://www.iar.com/website1/1.0.1.0/3/1/>. Acesso em: fevereiro 2010.

INTEGRATION AND RELIABILITY IMPROVEMENT. **Proceedings** Washington: IEEE Computer Society, 2008, p. 135-142.

KANG, B.; KWON, Y.; LEE, R. A Design and Test Technique for Embedded Software. In: 3RD ACIS INT'L CONFERENCE ON SOFTWARE ENGINEERING RESEARCH, MANAGEMENT AND APPLICATIONS, SERA. **Proceedings** Washington: IEEE Computer Society, 2005, p. 160-165.

KANSTRÉN, T. A Study on Design for Testability in Component-Based Embedded Software. In: SIXTH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING RESEARCH, MANAGEMENT AND APPLICATIONS, SERA, 2008. **Proceedings** Washington: IEEE Computer Society, 2008, p. 31-38.

KARLESKY, M.; BEREZA, W.; ERICKSON, C. Effective Test Driven Development for Embedded Software. Disponível em < http://www.atomicobject.com/files/EIT2006EmbeddedTDD.pdf >. Acesso em fevereiro 2010.

KARLESKY, M.; WILLIAMS, G. Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns. In: EMBEDDED SYSTEMS CONFERENCE SILICON VALLEY, 2007. San Jose.

KOEHNEMANN, H.; LINDQUIST, T. Towards Target-Level Testing and Debugging Tools for Embedded Software. In: ANNUAL INTERNATIONAL CONFERENCE ON ADA, 1993. Seattle. **Proceedings** New York: ACM Press, 1993, p. 288 - 298.

LETTNIN, D. et. al. Verification of temporal properties in automotive embedded software. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2008. Munich. **Proceedings** New York: ACM Press, 2008, p. 164 - 169.

LETTNIN, D.; WINTERHOLER, M. BRAUN, A. Coverage Driven Verification applied to Embedded Software. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, 2007. Porto Alegre. **Proceedings** Washington: IEEE Computer Society, 2007. p. 159-164.

LI, J. Prioritize Code for Testing to Improve Code Coverage of Complex Software. In: 16TH IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING, 2005. Chicago. **Proceedings** Washington: IEEE Computer Society, 2005, p. 75-84.

LI, J.; WEISS, D.; YEE, H. Code-Coverage Guided Prioritized Test Generation. **Proceedings** Information and Software Technology, 2006, v.48, p. 1187-1198.

MCCABE, T. A Complexity Measure. **Proceedings** IEEE Transactions on Software Engineering, TSE, 1976, v. 2, p. 308-320.

MESZAROS, G. XUnit Test Patterns, Refactoring Test Code. 3ª ed. Massachusetts, USA: Pearson Education, Inc, 2009.

MSP430 Homepage. Disponível em <http://focus.ti.com/mcu/docs/mcuprodoverview.tsp?sectionId=95&tabId=140&familyId=342>. Acesso em: fevereiro 2010.

PC-LINT Homepage. Disponível em: <http://www.gimpel.com/>. Acesso em: fevereiro 2010.

PEZZÉ, M.; YOUNG, M. Teste e Análise de Software Processos Princípios e Técnicas. São Paulo, Brasil: Editora Artmed SA, 2008.

PFALLER, C. et al. On the Integration of Design and Test - A Model-Based Approach for Embedded Systems. In: INTERNATION WORKSHOP ON AUTOMATION OF SOFTWARE TEST, 2006. Shanghai. **Proceedings** New York: ACM Press, 2006, p. 15-21.

PRESSMAN, R. Engenharia de Software. São Paulo, Brasil: Editora Pearson Makron Books, 1995.

SCILAB Homepage. Disponível em <http://www.scilab.org/>. Acesso em: fevereiro 2010.

SCITOOLS Homepage. Disponível em: <http://www.scitools.com>. Acesso em: fevereiro 2010.

SEO, J. et al. Which Spot Should I Test for Effective Embedded Software Testing? In: THE SECOND INTERNATIONAL CONFERENCE ON SECURE SYSTEM

SEO, J.; SUNG, A.; CHOI, B.; KANG, S. Automating Embedded Software Testing on an Emulated Target Board. In: SECOND INTERNATIONAL WORKSHOP ON AUTOMATION OF SOFTWARE TEST, AST. **Proceedings** Washington: IEEE Computer Society, 2007, p. 1-7.

SLAU208F, MSP430x5xx User's Guide, p. 497. Disponível em <http://focus.ti.com/lit/ug/slau208f/slau208f.pdf>. Acesso em: fevereiro 2010.

SUNG, A.; CHOI, B.; SIN, S. An interface test model for hardware-dependent software and embedded OS API of embedded system. **Proceedings** Computer Standard & Interface, 2007, v.29 p. 430-443.

SYSTEM C Homepage. Disponível em < http://www.systemc.org/home/>. Acesso em: fevereiro 2010.

TEXAS INSTRUMENTS Homepage. Disponível em <http://focus.ti.com/en/multimedia/flash/selection_tools/mcu/mcu.html>. Acesso em: fevereiro 2010.

TSAI, W.; YU, L. ZHU, F.; PAUL, R. Rapid Embedded System Testing Using Verification Patterns. **Proceedings** IEEE Software, 2005, v.2, p. 68-75.

UML Homepage. Disponível em < http://www.uml.org/>. Acesso em: fevereiro 2010.

WHALEN, M.; RAJAN, A.; HEIMDAHL, M.; MILLER, S. Coverage Metrics for Requirements-Based Testing. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS , ISSTA, 2006. Portland. **Proceedings** New York: ACM Press, 2006, p. 25-36.

WU, X.; LI, J.; WEISS, D.; LEE, Y. Coverage-Based Testing on Embedded Systems. In: SECOND INTERNATIONAL WORKSHOP ON AUTOMATION OF SOFTWARE TEST, AST, 2007. **Proceedings** Washington: IEEE Computer Society, 2007.

YU, R. Fiscal Cash Register Embedded System Test with Scenario Pattern. **Proceedings** International Journal of Computer Science and Network Security, 2006, v. 5A, p. 38-41.