UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

NORTON LIMA BARBIERI

Teste de Software Aviônico baseado em uma Plataforma Virtual de Baixo Custo

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dra. Erika Cota

Porto Alegre
2015

“No. Try not.

Do... or do not.

There is no try.”

— Grand Master Yoda

AGRADECIMENTOS

Agradeço às duas pessoas mais importantes desse mundo que com coragem e força me mostraram o caminho e incansavelmente me indicavam novamente o sentido correto a perseguir quando por vezes me perdia. Meus pais. Que por maior que fosse a dificuldade me apoiaram e me ajudaram a seguir em frente.

Agradeço a todos os professores do Instituto de Informática da UFRGS que se mostraram sempre muito capazes e preparados para nos ensinar não somente o conteúdo da disciplina mas a doutrina de buscar o conhecimento sempre. Em especial agradeço à minha orientadora Prof. Dra. Erika Cota que me auxiliou com seu tempo e vasto conhecimento na construção deste trabalho.

Agradeço à meus colegas e amigos na AEL Sistemas que desde o meu estágio vêm me auxiliando e me engrandecendo com seus conhecimentos e palavras de apoio quando a força me faltou.

RESUMO

Sistemas aviônicos, tanto civil quanto militares, têm se tornado cada vez mais complexos. Passou a época em que os pilotos tinham que lidar com instrumentos analógicos e uma infinidade de botões e ajustes. Com a evolução dos sistemas de controle e navegação a complexidade envolvida criou a necessidade da implantação de técnicas e ferramentas de projeto antes não esperadas nesse campo. Hoje os pilotos se confrontam com sistemas digitais totalmente integrados, com soluções redundantes e todo tipo de segurança que for possível a fim de evitar catástrofes. A proposta desse trabalho é desenvolver um framework que possibilite o teste e simulação de sistemas aviônicos em um ambiente de desenvolvimento de software de aplicação, sem a presença do hardware. Pretende-se com isso facilitar o teste de software e encontrar possíveis falhas em etapas precoces de desenvolvimento.

Palavras-chave: Software embarcado. teste de software embarcado. modelos de dispositivos de hardware. Aviônicos.

Avionics Software testing based on a low cost virtual plataform

ABSTRACT

Avionics systems being them civil or military, are becoming more complex. Its past the age of analogic needles and switches. The evolution of the control and navigation and the complexity has created the need for more advanced development technics and tools in this field. Nowadays pilots face entire digital screens totally integrated, with redundant solutions and every type of safety measures that aim at avoiding disasters. The goal of this work is to propose a framework that enables the test and simulation of an avionic system in a desktop environment without the need of specific hardware. The framework makes the software more testable and enables the developer to find bugs at early stages of development.

Keywords: embedded software, test of embedded software, modeling devices, Avionics.

LISTA DE ABREVIATURAS E SIGLAS

SMP	Symmetric Multi-Processor
NUMA	Non-Uniform Memory Access
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
ABNT	Associação Brasileira de Normas Técnicas
SO	Sistema Operacional
RTOS	*Real time Operational System* Sistema operacional de Tempo Real
SOC	*System on a Chip*
HAL	*Hardware Abstrations Layer*
API	*Application Program Interface*
APEX	*APplication EXecutive*
IMA	*Integrated Modular Avionics*
BSP	*Board Support Package*
HM	*Health Monitor*
DAL	*Design Assurance Level*
ISA	*Instruction Set Architecture*
GPS	*Global Positioning System*
IRS	*Inertial Reference System*
DVI	*Digital Visual Interface*
ADI	*Attitude Director Indicator*
WCET	*Worst Case Execution Time*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Sistemas aviônicos são a evolução dos antigos *cockpits* analógicos como na figura 1.1 para grandes telas independentes como na figura 1.2 capaz de substituir os equipamentos antigos com maior segurança, menor peso e mais conforto ao piloto ao delegar ao sistema verificações e tarefas repetitivas. A grande dificuldade nesse campo se encontra nos altos padrões de segurança DO178B/C (STALLBAUM; RZEPKA, 2010) e confiabilidade que são exigidos de sistemas computacionais embarcados. Além da alta confiança necessária, o determinismo é essencial para esse tipo de plataforma. Para possibilitar o desenvolvimento de tais sistemas são utilizados sistemas operacionais de tempo real (RTOS) específicos para esse fim. Em linhas gerais, tais sistemas operacionais fornecem algumas capacidades comuns a sistemas operacionais de uso geral como escalonadores, gerência de memória e abstração de hardware. No entanto tais funcionalidades são implementadas de maneira a conferir um alto nível de segurança.

Figura 1.1 – Cockpit antigo do avião C95

Fonte: Avionics Services

O RTOS que será utilizado nesta proposta é baseado no padrão Arinc 653 (PRISAZNUK, 2008). Esse padrão define um SO(Sistema Operacional) que fornece um alto controle sobre as aplicações, onde cada uma possui uma ordem e um tempo de execução pré-definido. A única maneira das aplicações se comunicarem é através das portas de comunicação também pré-definidas, assim garantindo um maior determinismo sobre o sistema e a segurança de que uma aplicação problemática não irá afetar outra. Esse tipo de sistema operacional é vastamente utilizado pela indústria em plataformas modernas de empresas como BOEING, AIRBUS, Ra-

fale, Lockhead Martin, Dassault, Sukhoi, BAE Systems. No entanto a utilização de um RTOS especializado não é suficiente se a aplicação a ser executada contiver problemas de lógica ou de codificação. Dessa forma durante o desenvolvimento do software são empregadas as clássicas fases de desenvolvimento envolvendo estudo de requisitos, design, codificação, testes de unidade e por final testes funcionais assim como teste de cobertura. O problema encontra-se na dificuldade encontrada em algumas dessas fases no desenvolvimento de sistemas embarcados. Fases como o teste de unidade e teste funcionais são grandes consumidoras de tempo e recursos, pois muitas vezes dependemos de algum hardware específico ou dispositivo para os executar. A dificuldade é muito grande durante a fase de testes funcionais também, já que muitas funcionalidades são muito complicadas de serem testadas isoladamente pois necessitam de recursos do sistema operacional.

Figura 1.2 – Cockpit Modernizado do avião C95M

Fonte: Avionics Services

Considerando o desafio de teste, este trabalho propõe o desenvolvendo um *framework* para facilitar a criação de testes unitários e funcionais em ambiente de desenvolvimento desktop diminuindo os custos pela utilização de um laboratório e facilitando a criação de cenários de teste. Com esse framework, o desenvolvedor poderá executar seu código sem a preocupação com as chamadas de sistema e respostas dos dispositivos. Tais recursos serão simulados através de diversas técnicas como *mocks* (KARLESKY et al., 2007), *stubs* permitindo que o desenvolvedor teste sua implementação e comprove seu funcionamento independente de qualquer outra

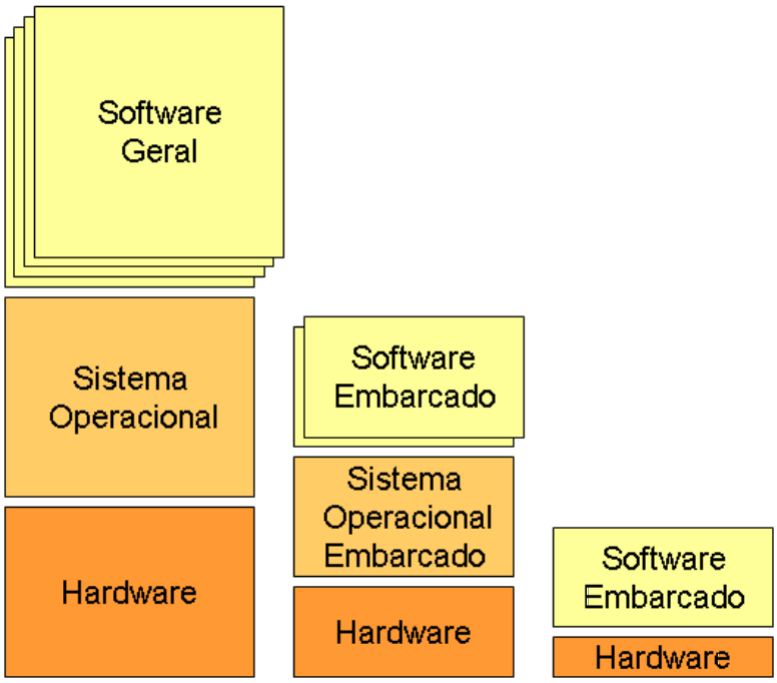
parte de software ou hardware. Os dispositivos(assim como as chamadas de sistema operacional) serão implementados através de modelos simplificados que permitam ao desenvolvedor injetar dados e controlar totalmente a resposta através das interfaces de software definidas no projeto real. Como resultado da utilização do *framework*, espera-se que o tempo de desenvolvimento diminua e a qualidade do software aumente ou se mantenha em um mesmo patamar, pois o software que for para o teste funcional no sistema real já estará mais maduro e com a maior parte dos problemas resolvidos em ambiente de desenvolvimento. Não são apenas testes unitários de software que serão simplificados. Testes funcionais, assim como teste de cobertura e estimativas iniciais de WCET (*Worst Case Execution Time*) poderão ser feitas dentro do framework. A curva de aprendizado também será reduzida, pois os desenvolvedores não necessitarão conhecer tão afundo os dispositivos e como operá-los, já que testarão o software como estão acostumados, em ambiente de desenvolvimento.

2 CONTEXTUALIZAÇÃO

2.1 Sofware e Sistemas Embarcados

O software para sistemas embarcados se diferencia em muito dos softwares desenvolvidos para desktops. A maior diferença nessa comparação reside na filosofia de processamento, memória e armazenamento infinitos ou tão grandes quanto se queira. Em sistemas embarcados existem limitações sérias em todos esses aspectos. Além disso, softwares para desktop possuem um poderoso sistema operacional abstraindo todo o hardware, regulando o acesso aos dispositivos, gerenciando memória, para citar algumas características. Em sistemas embarcados mesmo utilizando sistemas operacionais, SOs são muito mais simples no sentido de abstração dos dispositivos, assim como todas as demais características ou não presentes como podemos ver na figura 2.1.

Figura 2.1 – Tipos de sistemas embarcados



Fonte: (GOMES, 2010)

Com o avanço magnífico que houve nos últimos anos nos SOC (*System on a Chip*), hoje têm-se poderosos sistemas embarcados baseados em sistemas operacionais bastante avançados como Android e iOS. Tais sistemas possuem características que os aproximam de desktops com processadores multi core, com processadores de vídeo dedicados, com grande capacidade de memória RAM atingindo até 1GB e armazenamento ultrapassando a barreira dos 32GB. As aplicações para tais dispositivos se aproximam de aplicações desktop, pois os sistemas operaci-

onais oferecem poderosas HAL (*Hardware Abstraction Layer*) que permitem aos desenvolvedores se preocuparem menos com os recursos dos dispositivos e mais com as APIs providas pelo sistema. Neste trabalho o foco será em sistemas embarcados (segundo tipo como pode ser visto na figura 2.1) de tempo real utilizando um sistema operacional de tempo real específico para aviônicos não tão capaz como os supracitados (WATKINS; WALTER, 2009), mas com recursos de outra natureza envolvendo segurança e tolerância a falhas a fim de torná-lo mais robusto. O software embarcado para aviônicos utiliza vastamente, durante seu desenvolvimento, software geradores de código baseado em modelos. Tais ferramentas simplificam a tarefa de certificação e minimizam a necessidade de testes. Por outro lado, tais ferramentas limitam a programação largamente ao impedir o uso de ponteiros, regular severamente as enumerações, remover todo tipo de iterações, e impedindo a implícita conversão de tipos. Ao mesmo tempo que tais limitações tornam a tarefa do desenvolvedor mais árdua, elas simplificam o teste e garantem mais qualidade ao software final.

2.2 ARINC 653, APEX, IMA e o VxWorks 653

Para facilitar o desenvolvimento de aplicações embarcadas voltadas a aviônicos foi criado o padrão ARINC 653 (LITTLEFIELD-LAWWILL; KINNAN, 2008) que define um conjunto de APIs e especificaçoes para um sistema operacional. Esse SO possui uma interface de comunicação chamada de APEX que lida com a comunição entre as diversas tarefas que podem coexistir. Define um particionamento temporal e espacial para as tarefas respeitando limites rígidos. APEX é o único canal de comunicação entre as tarefas. As tarefas possuem uma interface bem definida e tudo que devem fazer é ler e escrever nesses canais de comunicação. É por esse canal que as tarefas receberão informação vinda dos dispositivos e de outras tarefas.

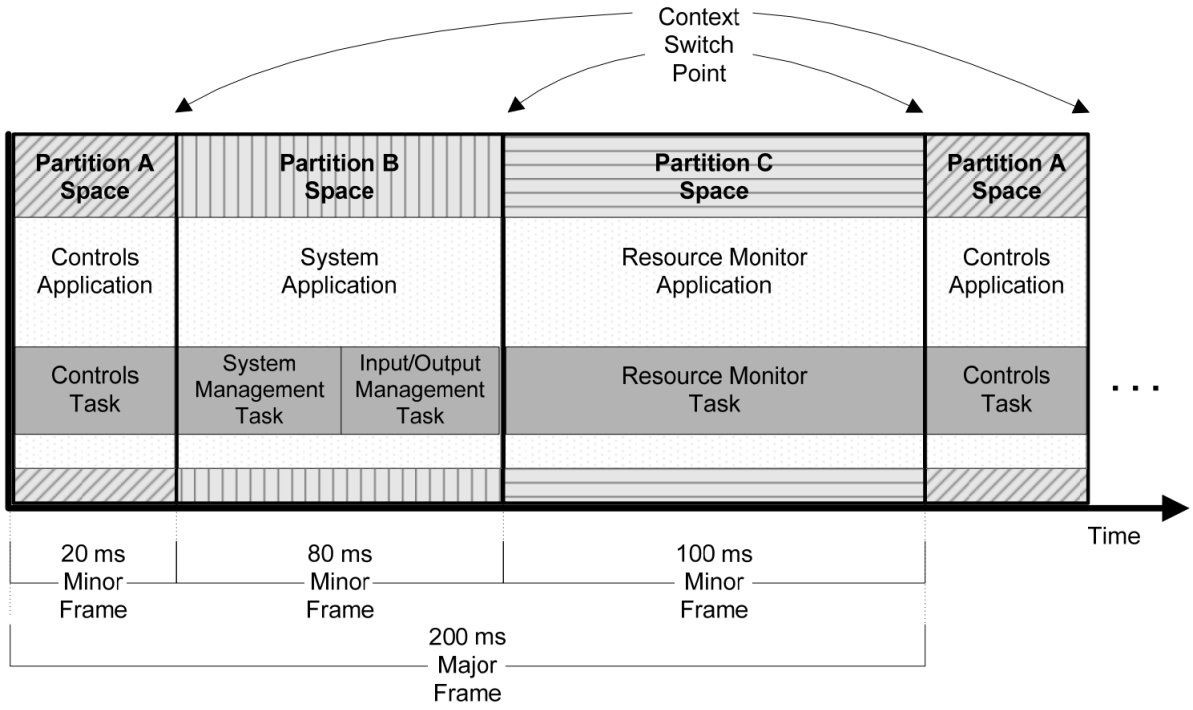
Existem apenas dois tipos de canais de comunicação:

- Existe o canal *sampling* que é utilizado para informações que apenas a última informação exemplos disso são relógios e sensores.
- Exite o canal *queuing*, no qual as mensagens são enfileiradas, que é utilizado para informações que são atualizadas mais rapidamente que o sistema, como botões na tela ou envio de dados sequenciais como um plano de voo.

É importante ressaltar que todos os canais APEX são configurados pelos desenvolvedores através de arquivos próprios do SO que são embutidos no software que vai para o dispositivo. IMA (PRISAZNUK, 2008) é um padrão que define um modelo um pouco diferente do convencional

para as tarefas e para o escalonador. Neste modelo, as tarefas tem um tempo determinado para executar chamado de *minor frame*, elas possuem uma ordem e a soma do tempo de todos os *minor frames* resulta no *major frame*, que significa o tempo que todas as tarefas executam e vão começar a executar novamente como na figura 2.2.

Figura 2.2 – Partições de software

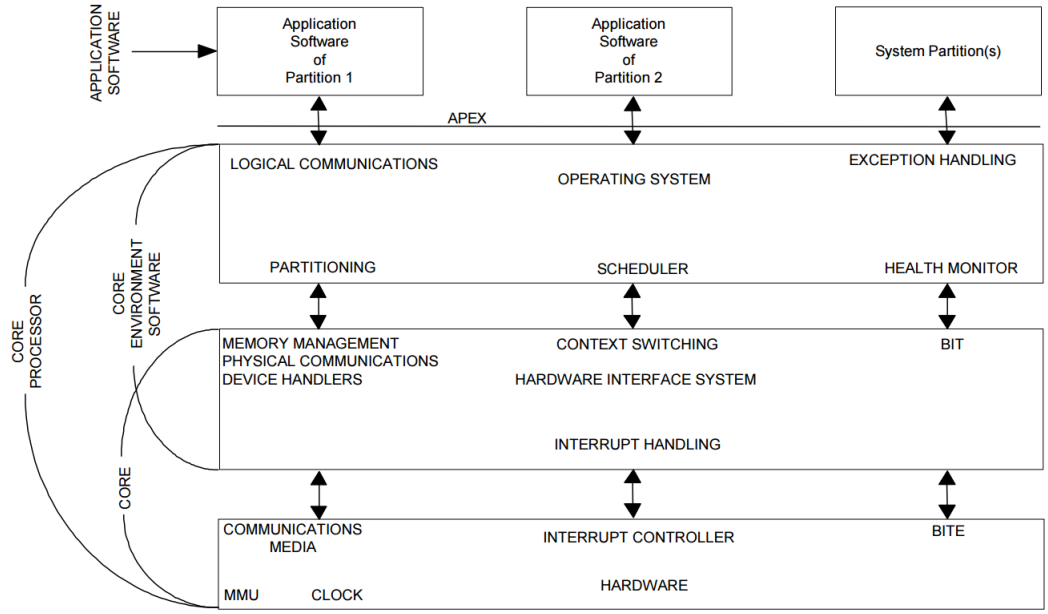


Fonte: VxWorks

Em outras palavras, o *major frame* consiste no período do sistema. Dentro de um *minor frame* podemos ter diversas tarefas e atividades assíncronas como interrupções. Isso exemplifica a separação temporal das tarefas. O particionamento de memória (TOKAR, 2003) é algo não exclusivo deste modelo, pois vemos essa característica em sistemas operacionais convencionais. Cada *processo* consegue indexar apenas endereços de memória que estejam definidos para sí. Como o único meio de comunicação entre tarefas é o canal APEX, essa independência é garantida. Os tempos de *minor frame* e *major frame* são definidos da mesma forma que as portas APEX em arquivos que são necessários para a compilação de todo o software. O VxWorks 653 é um sistema operacional que suporta tanto o IMA quando APEX e, por estar totalmente de acordo com o padrão ARINC 653, que vai além das características citadas, ele é um sistema operacional muito utilizado no campo da aviação. A figura 2.3 ilustra o sistema operacional com seus componentes e a interação da aplicação com o sistema.

Os dispositivos não fazem parte do sistema operacional. A camada que implementa os dispositivos e toda a abstração de hardware é responsabilidade do time de desenvolvimento. No entanto, essa parte não é considerada aplicação e não se encaixa nas restrições citadas até agora

Figura 2.3 – Relacionamento entre SO e Aplicações

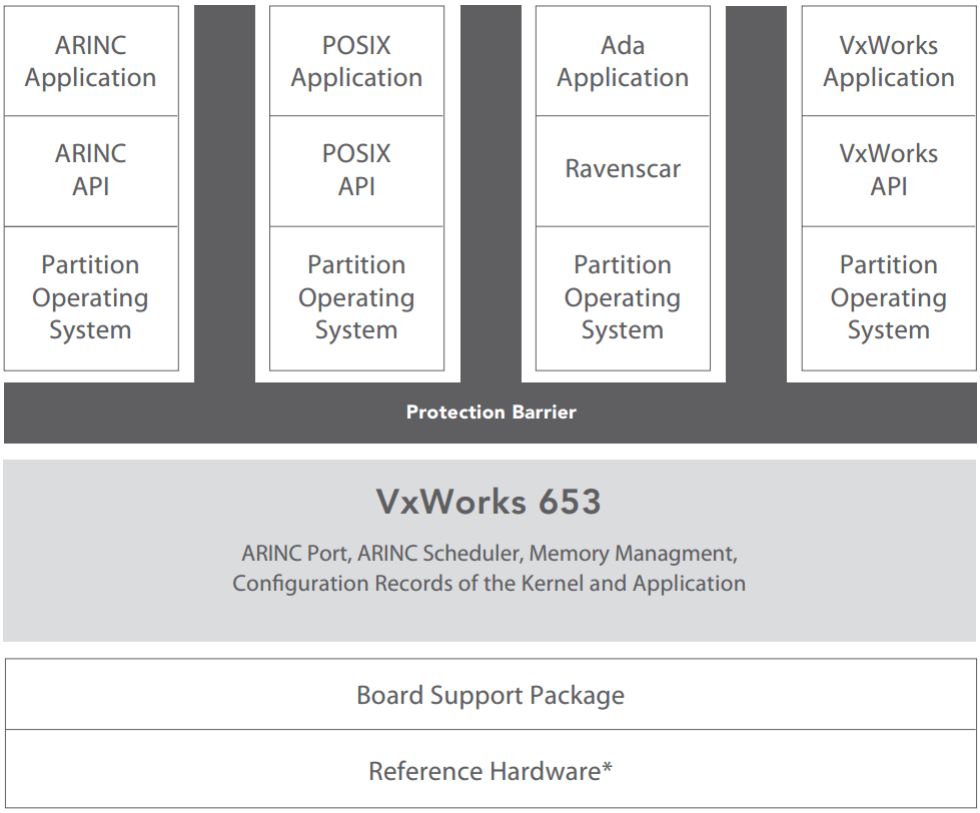


Fonte: VxWorks

como a comunicação restrita a APEX e modelos de memória particionados. A organização do sistema como um todo pode ser melhor visualizada na figura 2.4. Existe o BSP que fica abaixo do SO que pode ser considerado uma HAL. O BSP implementa os *device drivers* e todo tipo de interface com o hardware fornecendo esses dados através de portas APEX para a aplicação. Um exemplo simples seria um driver de ethernet. Da mesma forma que uma aplicação desktop recebe mensagens atraves de *sockets*, a aplicação embarcada recebe mensagens através do canal APEX.

Qualquer problema grave que ocorra com o sistema, por exemplo uma aplicação que não finalizou a sua execução quando o final do seu *minor frame* chegou, gera um evento de HM (*health monitor*) que guarda todas as falhas que a aplicação tiver ou que ela levantar de uma maneira semelhante a exceções. O HM pode ser lido por qualquer aplicação e dessa forma pode-se identificar os defeitos ou saber da condição geral do sistema. Com a divisão que o IMA proporciona, é possível a integração de diversas aplicações com criticidades diferentes em um mesmo sistema. Dessa forma o mesmo dispositivo pode processar informações muito importantes ao sistema ao mesmo tempo que provê funcionalidades que não possuem um grau de criticidade tão elevado. Esse nível de criticidade é chamado de DAL (*Design Assurance Level*) e poder suportar aplicações com diversos DALs é uma característica que fez o padrão ARINC 653 ficar tão popular e reduzir muito a quantidade de equipamento e o peso deles nas aeronaves.

Figura 2.4 – Organizacao em camadas do SO



Fonte: VxWorks

2.3 Teste de sistemas embarcados

Tendo em vista o grande avanço na capacidade de processamento dos novos processadores, tem-se cada vez mais recursos para aplicar algoritmos mais complexos e realizar computações mais avançadas. Com isso, diversas atividades antes feitas em hardware passaram a ser feitas em software (GOMES, 2010). De certa forma essa transição do hardware para o software facilita o teste de sistemas embarcados reduzindo as interfaces software hardware. No entanto, ainda existe um grande acoplamento no que tange as interfaces entre o software e o hardware. Tais interfaces podem ser testadas apenas através da carga do software no dispositivo real e o testando funcionalmente, um processo que sabidamente muito custoso e pouco eficiente (QIAN; ZHENG, 2009). Deve ser considerado também que de tal forma, o software só poderá ser testado e desenvolvido consistentemente quando o projeto do hardware estiver em suas fases finais, o que agrega muitos custos ao elevar o tempo de desenvolvimento do produto. A fim de acelerar o desenvolvimento, o software é desenvolvido sem o hardware destino. Quando este se encontra razoavelmente maduro a integração é realizada em uma abordagem tipo big-bang (SEO et al., 2008). Sabendo que os componentes do software não foram bem testados (JANG; KIM; CHUNG, 2008) e ainda assim uni-los com o hardware pode desencadear muitos proble-

mas, tanto em cada parte do software como problemas de integração, dificultando o processo de desenvolvimento e tomando mais tempo para identificar e isolar os defeitos.

Ambientes de desenvolvimento como o WindRiver (RIVER, 1997) fornecem algumas ferramentas para simular o comportamento do hardware, mas tendem a ser muito complicados de utilizar e extremamente limitados na sua capacidade de simular dispositivos, o que torna esse tipo de simulação inviável. Como sistemas embarcados tendem a se comunicar com outros dispositivos através de interfaces de comunicações, o teste unitário de certos fragmentos do software se torna muito complicado. Por essa razão apenas testes de integração são realizados normalmente, o que não é a melhor abordagem ao problema. Ao realizar testes diretamente nos dispositivos muitas vezes problemas novos aparecem devido ao impacto do baixo desempenho do código não otimizado, ou da resposta muito lenta do sistema ao utilizar um breakpoint, o que faz o sistema reiniciar e gera mais dificuldades para testar no dispositivo.

Sistemas aviônicos possuem comunicação com diversos dispositivos como GPS, ADHARS, Pitot, Sensores meteorológicos, entre outros (WENZEL et al., 2005). O software que implementa a interface entre esses dispositivos é um ponto vago no que se trata de testes. Tanto em desenvolvimento quanto na fase de testes de integração tais sistemas são considerados como funcionando através de terceiros (SAGLIETTI, 2010). Sabemos que a comunicação com o sensor meteorológico está funcionando pois a imagem do sensor aparece na tela. No entanto, é muito complicado testar o funcionamento minucioso do dispositivo e se seu desempenho está de acordo com o requerido. Na mesma situação dos dispositivos, se encontra a interface entre duas partições de software. Como já explicado anteriormente ela funciona através de portas APEX (RUSHBY, 2011), cujo conteúdo e taxa são bastante complexos de serem analisados com o sistema funcionando. Tão importante quanto testes funcionais e unitários são os testes de cobertura (ZHANG; SHEN, 2011), que devem ser realizados a fim de identificar trechos de código que nunca são ativados e, ao mesmo tempo, garantir que os testes funcionais cobriram todos os casos possíveis para tal finalidade.

Ferramentas para analisar a cobertura de código para software não embarcado existem muitas. No entanto ferramentas semelhantes para analisar a cobertura de código nos dispositivos são muito mais complexas e sua disponibilidade é muito menor e a custos muito mais elevados. Ferramentas estado da arte, como LDRA Dynamic Coverage (ZHANG; SHEN, 2011) e VectorCast Dynamic Coverage (WARD, 2011), não são capazes de fornecer os resultados em tempo real. Logo ao escrever o teste o desenvolvedor necessita analisar o código e prever as entradas necessárias para exercitar todos os trechos de código, já que terá apenas o resultado ao final do teste. Ferramentas de cobertura de código para software não embarcado se encontram

muito mais maduras e a valores muito menores, apresentando resultados em tempo real e com muito mais possibilidades, como resultados em tempo real.

3 O PROBLEMA

O desenvolvimento de software embarcado envolve uma série de contratempos específicos a esse tipo de software. Sem o acesso ao hardware diretamente e com soluções de debug não ideais, são diversas as dificuldades nesse campo. Em contrapartida os sistemas embarcados vêm constantemente se tornando mais complexos integrando diversos dispositivos e apresentando diversas interfaces.

Sistemas aviônicos sofrem de problemas similares, porém com agravantes. Os sistemas operacionais não oferecem a mesma flexibilidade na comparação a suas versões no desktop. As ferramentas se tornam ainda menos capazes de auxiliar no desenvolvimento, acarretando, portanto, em um maior tempo de desenvolvimento.

3.1 Desenvolvimento

Uma vez em posse de um ambiente de hardware e software de baixo nível estável, o desenvolvimento das aplicações pode ter início. O desenvolvimento de software aviônico zegue a norma ARINC 653 e baseia-se no conceito de partições, onde cada uma delas deve possuir uma função clara e bem definida. O design mais comum e abrangente envolve uma partição responsável pela interface com os dispositivos e aquisição de dados, seguida por demais partições que podem ter diversas funções como gerenciar a interface gráfica, interagir com dispositivos mais complexos, executar algoritmos críticos ou monitorar a lógica de outras partições como pode ser visto em 2.2. Neste conceito vemos que uma partição claramente terá o seu desenvolvimento envolto na interface com dispositivos enquanto terceiras partições serão apenas clientes, utilizando o dado já recebido e processado. Claramente a partição que possui muitas interfaces terá dificuldades no desenvolvimento já que terá que ser inteiramente desenvolvida no computador alvo e utilizando interfaces internas (acesso aos drivers de dispositivos) que nem sempre possuem um alto grau de maturidade

3.2 Hardware e Integração

Em decorrência da baixa qualidade das ferramentas de desenvolvimento aliada à complexidade dos sistemas desenvolvidos, o tempo de projeto sempre é alvo para estudos e aprimoramentos. Existe uma longa fase na qual a equipe de software encontra-se desconectada do

desenvolvimento.

Nas fases iniciais do projeto, enquanto o hardware e os drivers de dispositivo estão sendo desenvolvidos, o software nada ou pouco pode realizar. Apesar de possuir as interfaces e descrição dos dispositivos, a equipe de software, sem os drivers, não é capaz de iniciar o desenvolvimento apesar de possuir toda ou parte da informação necessária.

Devido ao extenso cartel de dispositivos COTS, muitas vezes existem interfaces com dispositivos novos ou customizados tornando a tarefa de desenvolvimento dos drivers ainda mais complexa. A arquitetura de hardware e os dispositivos utilizados dificulta ainda mais a utilização de drivers genéricos, obrigando desta forma o longo e custoso desenvolvimento de drivers de dispositivos customizados ou adaptados para cada projeto. Por fim, com a diversidade de sistemas operacionais utilizados na área, o tempo de projeto tomado pelo desenvolvimento de baixo nível realmente se torna um problema para os novos projetos ou projetos com atualizações no hardware. Uma vez de posse dos drivers de dispositivos e do sistema operacional, a equipe de aplicação se torna a primeira consumidora dos drivers, portanto encontrando diversos bugs e inconsistências tanto no hardware quanto nos drivers. O hardware aviônico também possui componentes desenvolvidos in-house. O alto nível de acoplamento entre hardware e software de baixo nível que caracteriza esse tipo de desenvolvimento claramente cobra o seu custo e torna o desenvolvimento e a depuração mais complexas e custosas. Mais uma longa e custosa etapa de integração se inicia com as equipes de hardware, drivers e aplicação depurando os problemas encontrados. Tão somente com a conclusão dessa etapa e de posse de hardware e drivers com certo grau de maturidade é que a equipe de aplicação consegue iniciar o desenvolvimento da solução que o cliente procura.

3.3 Fase de Testes

O maior problema e a maior dor de cabeça para os desenvolvedores de sistemas encontra-se na fase de teste. Esta fase é apontada por alguns autores (HILDERMAN; BAGHI, 2007) como a fase mais custosa no desenvolvimento de um sistema aviônico. Nesta fase os desenvolvedores são encarregados de escrever testes e executar os testes no sistema. No entanto existem diversos níveis de testes, a saber:

- Testes de requisitos de software de alto nível
- Testes de requisitos de software de baixo nível
- Testes de robustez nos limites

- Testes de robustez nas condições de erros
- Testes para cobertura estrutural de sentenças
- Testes para cobertura estrutural condições e decisões
- Testes para cobertura estrutural MCDC (HAYHURST et al., 2001)

Todos esses testes devem ser desenvolvidos e verificados manualmente um a um devido ao processo de certificação segundo a norma DO-178B. O desenvolvimento dos testes, principalmente os testes de cobertura, é um processo extremamente complexo e consumidor de tempo e pessoas, gerando alto impacto no desenvolvimento de qualquer sistema. Testes, como o teste de cobertura MCDC, tendem a encontrar pouco auxílio nas ferramentas fora a captação dos dados de uma dada execução mas sem nenhuma capacidade de interatividade on-line ou correlação entre resultados e entradas. Logo, toda a responsabilidade recai sobre o desenvolvedor que deve prover e identificar as entradas e valores necessários para o teste de cobertura completo. Essa fase do desenvolvimento não encontra nas ferramentas muito auxílio, além de coletar os dados de cobertura. Os engenheiros são responsáveis por escrever as rotinas de testes com base no seu conhecimento dos requisitos somente. Isso contrasta com as diversas ferramentas encontradas para mesmo fim no ambiente de desenvolvimento para desktop. Devido às formalidades requeridas pelo processo de desenvolvimento de software aviônico de alto nível de criticidade, todas as etapas devem ser verificadas e revisadas manualmente e independentemente. Logo, a etapa de testes se torna extremamente custosa e foco para melhorias tanto nas ferramentas quanto nos processos.

3.4 Geração de Testes

Testes consistem em um processo de executar um programa ou sistema com a intenção de encontrar erros (PAN, 1999). Para tal, um processo formal deve ser estabelecido. Dessa forma se cria uma rotina de teste composta de uma série de passos definidos por entradas e saídas esperadas.

A parte mais básica e primordial dos testes consiste nos testes de corretude. Nos testes de corretude o propósito é o comportamento correto do software no que tange entradas e saídas frente aos requisitos de software.

Para os testes de corretude o desenvolvedor pode conhecer ou não os detalhes internos do software ou sistema a ser testado. Isso define se o teste será um teste de caixa preta ou um teste de caixa branca. Para os testes de corretude ambos os métodos são aceitáveis.

Portanto testes de corretude devem ser escritos de maneira a testar o software ou sistema frente a todos os requisitos de software a fim de comprovar que o software ou sistema se comporta de acordo com o comportamento descrito nos requisitos de software.

Problemas encontrados nos testes de corretude (KANER, 2006) provêm de uma má interpretação dos requisitos de software ou uma má implementação dos requisitos. Ambos erros comuns decorrentes da natureza humana do desenvolvimento de software.

Além de testes de corretude temos testes de robustez. Testes de robustez têm por objetivo principal testar a capacidade do software ou sistema de permanecer operacional mesmo na presença de falhas em seus componentes ou na presença de entradas fora do limite ou corrompidas.

Testes de robustez por exemplo testam o software ou sistema com entradas que, segundo a especificação dos componentes, não seriam possíveis mas, devido à radiação ou eventos externos ao sistema ou fora da especificação, podem ocorrer.

Em casos extremos não se busca um comportamento específico ou a corretude do sistema, mas sim a capacidade do software ou sistema de permanecer operacional com suas capacidades principais inalteradas ou com mínima degradação (cosmética por exemplo).

3.5 Testes de cobertura

Até esse ponto, o foco dos testes tem sido a comparação de entradas e saídas. Pois testes de cobertura possuem um objetivo diferente. Testes de cobertura tem como foco o fluxo de execução e quais partes do código foram executadas e quais partes não. Testes de cobertura são importantes já que "não é possível atestar a qualidade de partes do software que não foram executadas" (GOMES, 2010).

Portanto testes de cobertura têm o código como parte fundamental e sua justificativa a estrutura do código. A estrutura que foi dada ao código não é algo que possa ser avaliado como certo ou errado através da cobertura. O foco aqui é garantir que existe uma ou mais entradas que façam com que a execução passe por todas partes do código.

Por partes do código se podemos definir quatro níveis de cobertura de código (KIM, 2003):

- Funções
- Sentenças
- Desvios

- Condições
- MCDC

A cobertura por funções é mais simples delas. Consiste apenas na garantia que cada função ou método foi chamada pelo menos uma vez.

A cobertura de sentenças é mais complexa e o foco está nos blocos de execução. Cada bloco deve ter sido executado ao menos uma vez.

A cobertura de desvios tem por objetivo garantir que em cada desvio o software tomou todos os caminhos.

A cobertura de condições tem que garantir que cada sub expressão booleana tenha sido avaliada em ambos os casos.

MCDC tem por objetivo provar que todos os pontos de entrada e de saída de uma função tenham sido executados, todas as decisões tenham provocado todos os resultados, cada condição em uma decisão percorra todos os caminhos possíveis e cada condição em uma decisão é capaz de independentemente afetar o resultado da decisão. Claramente os testes de MCDC são os mais complexos e constituirão a maior quantidade de testes ou passos no teste já que exigem muito mais cenários que os anteriores.

Todas as modalidades de testes de cobertura dependem da estrutura do código mais do que da lógica que está implementada. Testes de cobertura MCDC extrapolam essa medida no sentido de que simples modificações no código podem reduzir drasticamente o número de casos de teste necessário para atingir 100% de cobertura (RAJAN; WHALEN; HEIMDAHL, 2008).

Um exemplo simples pode ser compreendido com a transformação das condições de um IF em um teste simples :

```
int myFunc (bool c1, bool c2, bool c3)
{
bool d2 = (c1 or c2) and c3;
if(d)
   return 1;
else
   return -1;
}
```

transformado em:

```
int myFunc (bool c1, bool c2, bool c3)
{
```

```
bool d1 = c1 or c2
bool d2 = d1 and c3
if(d2)
    return 1;
else
    return -1;
}
```

podemos ver que a criação de variáveis temporárias simplificou o teste de MCDC diminuindo a quantidade de casos a serem criados. Manipulações como esta podem ser realizadas em todo código afim de otimizar a quantidade de casos de teste.

4 SOLUÇÃO PROPOSTA

Diante do exposto, a solução proposta envolve aproximar o desenvolvimento de software para desktop do desenvolvimento de software para sistemas embarcados. Dessa forma aproveitar das facilidades e ampla gama de soluções já existentes na forma de ferramentas e processos. Para atingir esse objetivo serão utilizados Dublês de teste para substituir sistemas ou dispositivos específicos da plataforma alvo.

4.1 Dublês de Teste

Ao testar entidades de software, frequentemente encontramos interfaces que se conectam a outras entidades que não estamos interessados em testar no momento. Portanto tratamos tais entidades como caixas pretas, já que estamos possivelmente interessados na saída dessa entidade e ela pode influenciar o fluxo de execução da entidade atualmente sobre teste (FREEMAN; PRYCE, 2009).

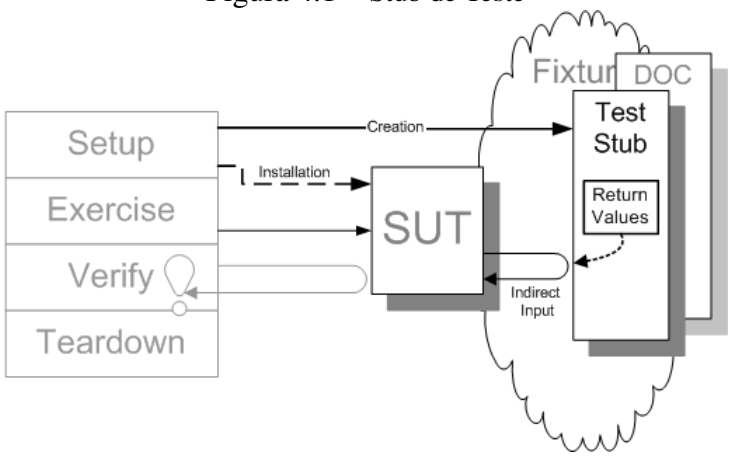
Para contornar esse problema podemos utilizar a técnica de Dublês de Teste. Esta técnica consiste basicamente em substituir as entidades que atualmente não são o foco do teste por componentes de mesma interface, porém com comportamento simplificado, embora semelhante (MESZAROS, 2007).

Gerad Meszaros (MESZAROS, 2007) define cinco tipos de dublês de teste que serão explicados a seguir.

4.1.1 Stub de Teste

Retorna valores previamente definidos independente das entradas providas.

Figura 4.1 – Stub de Teste

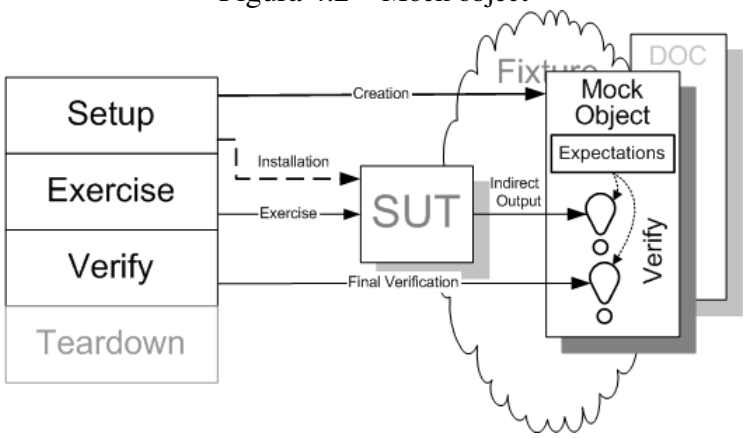


Fonte: xUnitPatterns

4.1.2 Mock object

Retorna valores definidos em tempo de *setup* podendo testar argumentos e possuindo pequena implementação.

Figura 4.2 – Mock object

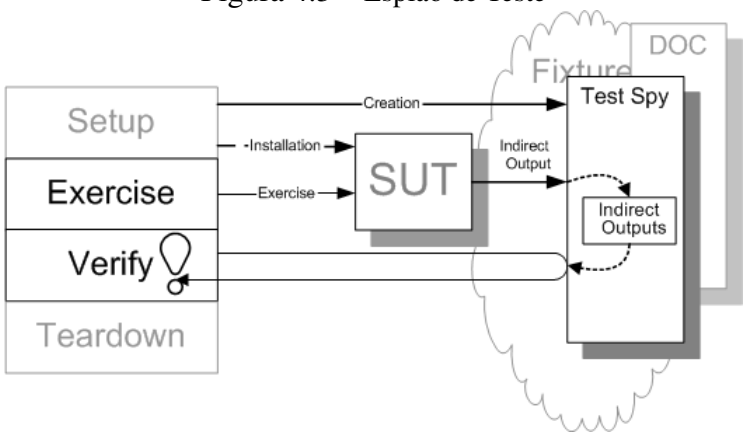


Fonte: xUnitPatterns

4.1.3 Espião de Teste

Esse objeto tem por objetivo não fornecer dados para a entidade sendo testada, mas sim capturar as mensagens trocadas pela entidade e o objeto alvo.

Figura 4.3 – Espião de Teste

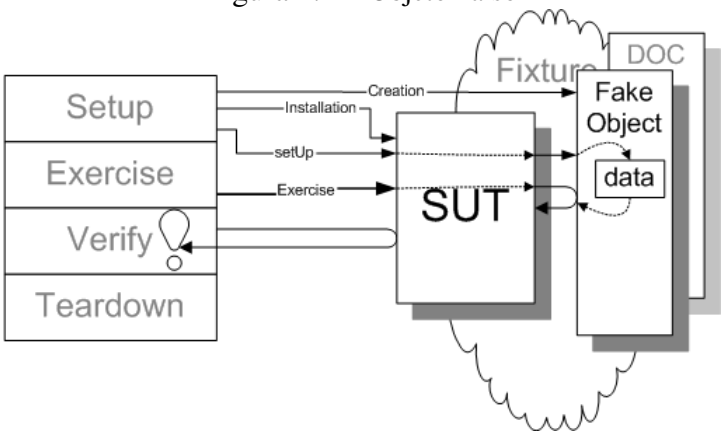


Fonte: xUnitPatterns

4.1.4 Objeto Falso

Nesse caso temos uma implementação da entidade porém de forma mais simples, ou não respeitando requisitos importantes para o funcionamento da entidade. Por exemplo, o uso de um banco de dados em memória para testar interfaces com banco de dados real.

Figura 4.4 – Objeto Falso



Fonte: xUnitPatterns

4.1.5 Objeto Dummy

Neste caso temos apenas a arquitetura da entidade sem comportamento algum. Essa técnica é utilizada quando a simples existência da entidade é suficiente para o teste.

4.2 A Solução

A solução proposta neste trabalho tem como objetivo aproximar o desenvolvimento para dispositivos embarcados com o ambiente desktop e, dessa forma, utilizar-se de ferramentas, ambientes e bibliotecas que possam facilitar o processo de desenvolvimento em toda a sua cadeia.

A solução baseia-se na maior flexibilidade apresentada pelas ferramentas para desktop (Windows/ Linux). Ambientes no qual se encontram com certa facilidade diversas ferramentas para auxiliar o processo de desenvolvimento. Além da maior disponibilidade de ferramentas, ambientes desktop apresentam muitas ferramentas a custos mais atrativos.

Algumas barreiras são encontradas ao desenvolver software *cross-platform* como as chamadas de sistema, os dispositivos específicos, conjunto de instruções especiais, ISAs não compatíveis.

Tendo em foco que o desejo é o auxílio no desenvolvimento de software embarcado em sistemas de tempo real utilizando o sistema operacional VxWorks 653, podemos definir mais claramente quais são os desafios e como eles serão abordados:

- Estamos em um ambiente de sistemas embarcados de tempo real que utiliza um sistema operacional bastante restritivo [ARINC 653].
- Estamos focando no desenvolvimento da APLICAÇÃO e não software de baixo nível.
- A aplicação deve passar por vários tipos de certificações [DO-178B, ARP 4754] ou seja não deve utilizar linguagem de baixo nível.
- A aplicação possui uma interface específica e bem definida [ARINC 653] para se comunicar com o sistema operacional e receber mensagens.
- O sistema operacional apresenta um escalonador fixo, logo não teremos problemas com ambientes multithread.

4.2.1 Especificação da solução

A solução deverá recriar as chamadas de sistema do VxWorks 653 em ambiente Windows/Linux de forma a permitir que as partições sejam executadas em sua ordem correta e respeitando os ciclos de inicialização e normal.

Modelos dos dispositivos reais injetam dados nas portas ARINC 653 de forma a possibilitar a troca de dados entre os modelos e as partições.

Mensagens trocadas pelas partições devem ser processadas e enviadas de uma partição para a outra respeitando o escalonamento.

Chamadas de sistema básicas (get time, get date, change mode) devem ser implementadas a fim de permitir que as partições executem tão perto quanto possível da interface real.

Codificada na linguagem ANSI C, que é a linguagem mais comum no desenvolvimento desse tipo de ambiente.

Construída na forma de framework, será compilada e executada junto com as partições de forma a possibilitar a depuração tanto do fluxo de execução quanto de dados.

Os modelos serão criados baseados nos dispositivos reais servindo como fonte de dados para a aplicação.

Os modelos possuem a liberdade de serem escritos em C/C++ já que devem utilizar as chamadas do sistema operacional para enviar dados para as aplicações

Os modelos não necessariamente precisam representar todos os aspectos do dispositivo real. Abre-se a possibilidade de desenvolver pequenos modelos de maneira gradual/incremental seguindo o fluxo do projeto.

Os modelos podem ser baseados em lógica ou em dados extraídos de execuções de sistemas reais.

Os modelos devem respeitar rigorosamente a interface definida pelo dispositivo real a fim de exercitar o software exatamente com os dados que os dispositivos podem fornecer.

Deve ser mantida a capacidade de injeção de ruído ou interferência na comunicação entre os dispositivos e as partições a fim de simular o software em condições extremas.

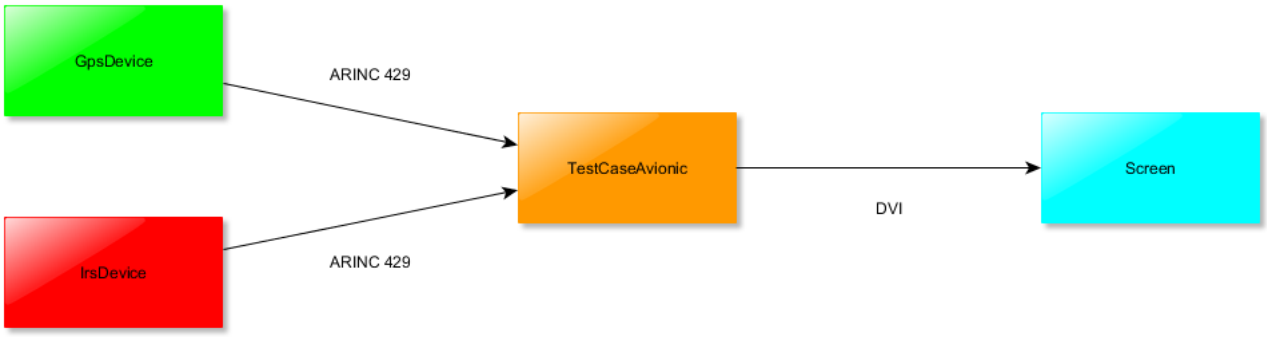
5 CASO DE ESTUDO

A fim de por à prova a solução proposta, foi desenvolvido um pequeno sistema aviônico contendo componentes frequentes e uma arquitetura comumente encontrada em sistemas dessa natureza. Devido ao caráter confidencial deste mercado, não foi possível a utilização de sistemas reais. Contudo o sistema concebido possui as principais características de soluções comerciais sem possuir é claro a mesma complexidade e dimensão.

5.1 O sistema proposto

O sistema proposto consiste de dois dispositivos fornecendo dados para o sistema aviônico que gera imagens como pode ser observado na Figura 5.1.

Figura 5.1 – Arquitetura do Estudo de Caso



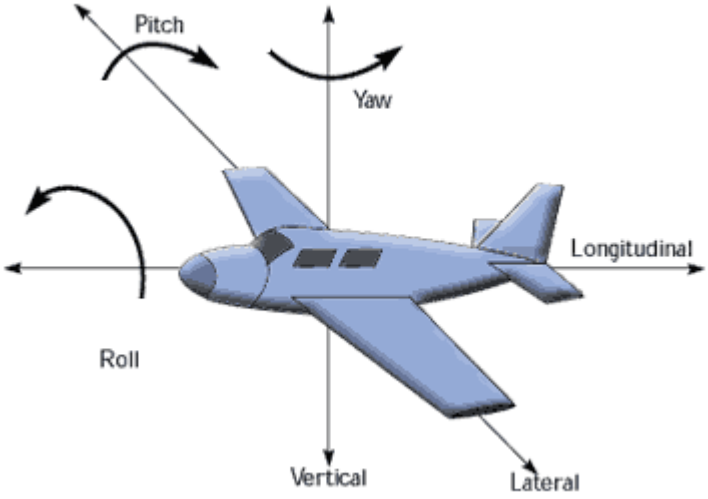
Fonte: O Autor

A aquisição de dados se dá através da interface ARINC 429 para ambos os dispositivos de entrada e o protocolo de saída das imagens é o DVI.

Os dispositivos de entrada para o sistema são um dispositivo GPS e um dispositivo IRS.

- O GPS é um dispositivo muito utilizado por fornecer dados de localização geográfica com precisão. Baseado em satélites geoestacionários pode fornecer informações como latitude, longitude, altitude, hora e data.
- O IRS é um dispositivo muito utilizado por fornecer dados de posicionamento inercial como a atitude da aeronave (ver figura 5.2) e como o ângulo de inclinação da aeronave em relação ao solo em diversos eixos. Enquanto o GPS se baseia em informações externas, o IRS baseia-se na sua própria informação. Através de acelerômetros e giroscópios ele é capaz de determinar o ângulo de rolagem, o ângulo de inclinação vertical e o ângulo de inclinação horizontal.

Figura 5.2 – Informações providas pelo sensor IRS



Fonte: http://www.novatel.com/solutions/attitude/

A saída do sistema é um fluxo de imagens no formato DVI que poderia ser utilizado para fornecer dados a um sistema de aviônicos qualquer como um *Head up display*, como um *heads down display* (Figura: 5.3) ou um capacete (Figura: 5.4).

Figura 5.3 – Aviônicos

Fonte: Elbit

Figura 5.4 – Capacete

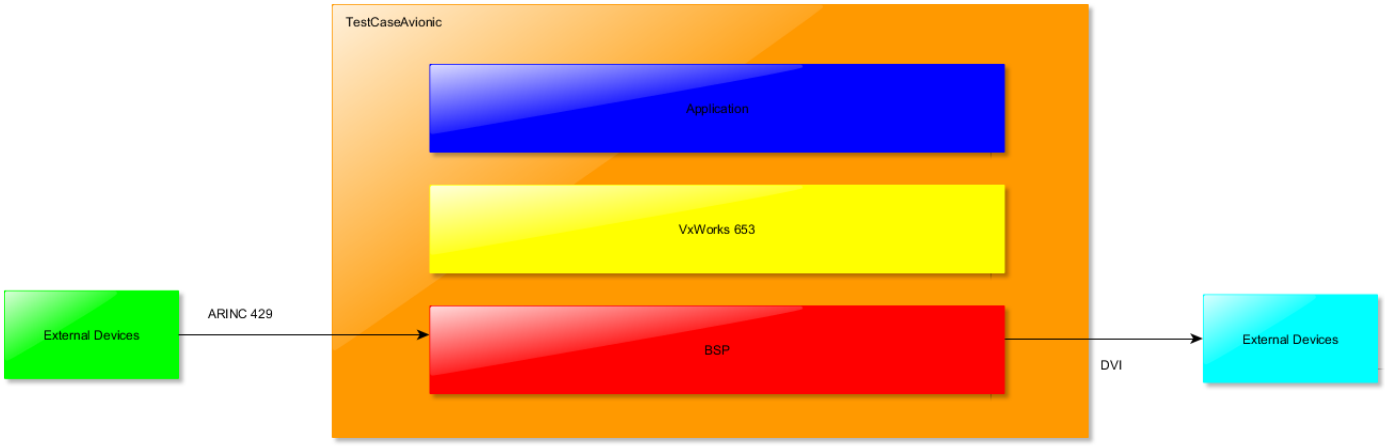
Fonte: Elbit

5.1.1 Arquitetura de Software de alto nível

Como padrão para software aviônicos definido a norma DO-178b, a arquitetura de software se baseia na norma ARINC 653 onde temos a aplicação separada dos drivers e separada do sistema operacional, no caso de estudo foi utilizado o SO VxWorks 653, que implementa completamente a norma supracitada. A arquitetura de software é apresentada na figura 5.5 e contém:

- em azul o software de aplicação
- o sistema operacional VxWorks 653 em amarelo
- o BSP em vermelho com os drivers de dispositivos

Figura 5.5 – Arquitetura de Software



Fonte: O Autor

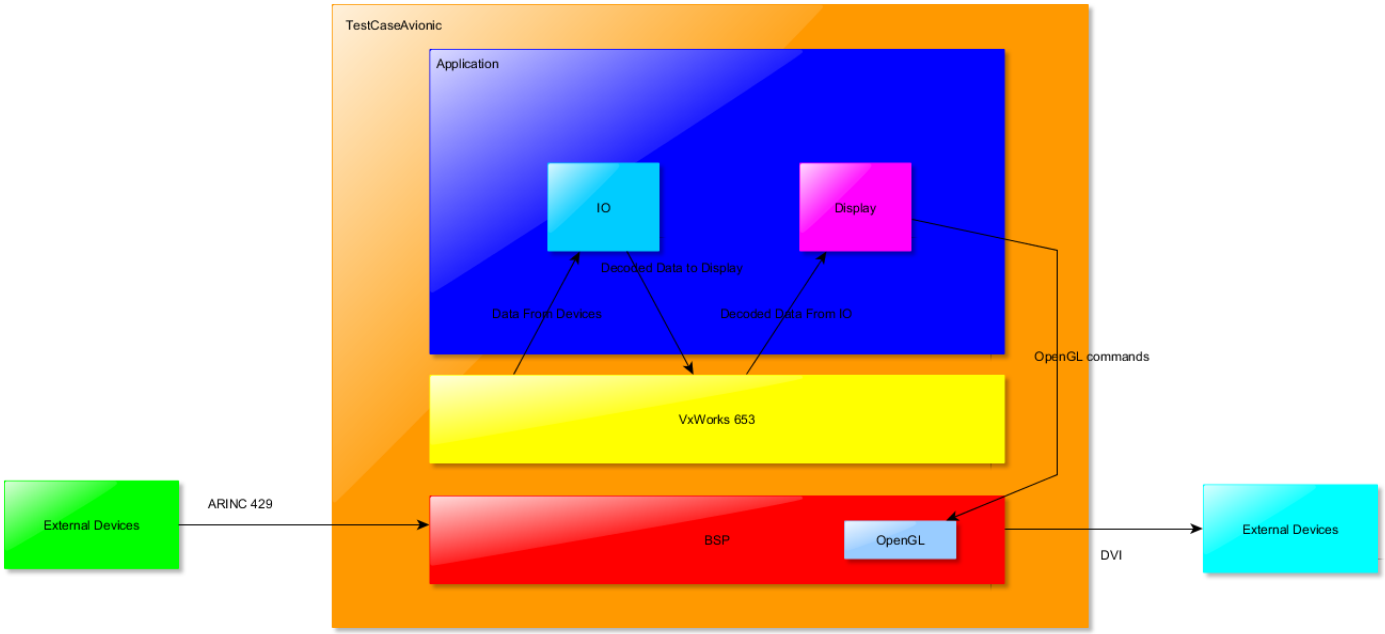
5.1.2 Arquitetura de Software de baixo nível

O software de aplicação foi organizado em duas partições:

- IO - responsável pela aquisição de dados e decodificação
- Display - responsável pela renderização da tela a partir dos dados decodificados

A divisão do software em partições tem por objetivo simplificar os testes e tornar o sistema mais robusto. Dessa forma podemos testar as partições individualmente e comprovar o seu funcionamento correto. A funcionalidade também tem um peso no que se trata da divisão das partições. Essa divisão tem por qualidades designar toda a entrada de dados e decodificação na partição de IO fazendo interface com o driver de Arinc 429 enquanto a partição Display se

Figura 5.6 – Arquitetura de Software Detalhada



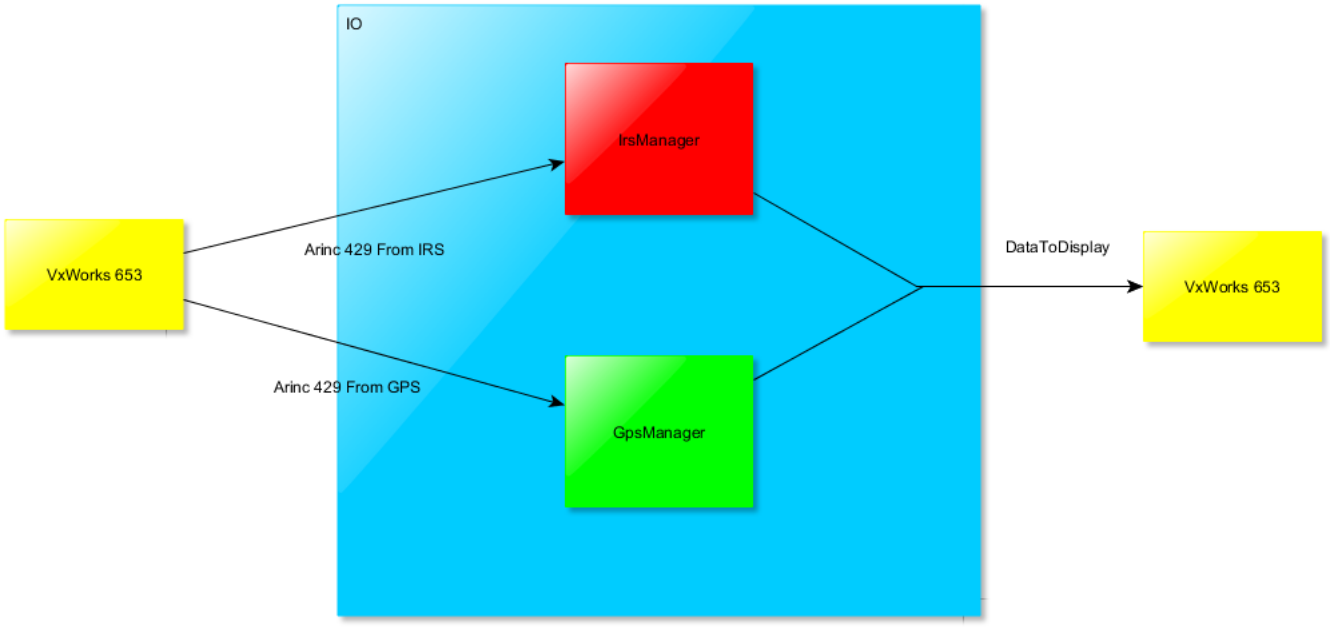
Fonte: O Autor

encarrega da renderização e tem interface com o driver OpenGL.

5.1.3 Partição IO

Internamente a partição IO possui dois componentes principais que gerenciam o recebimento e decodificação de dados de cada um dos dispositivos relevantes figura 5.7.

Figura 5.7 – Arquitetura de Software Detalhada IO

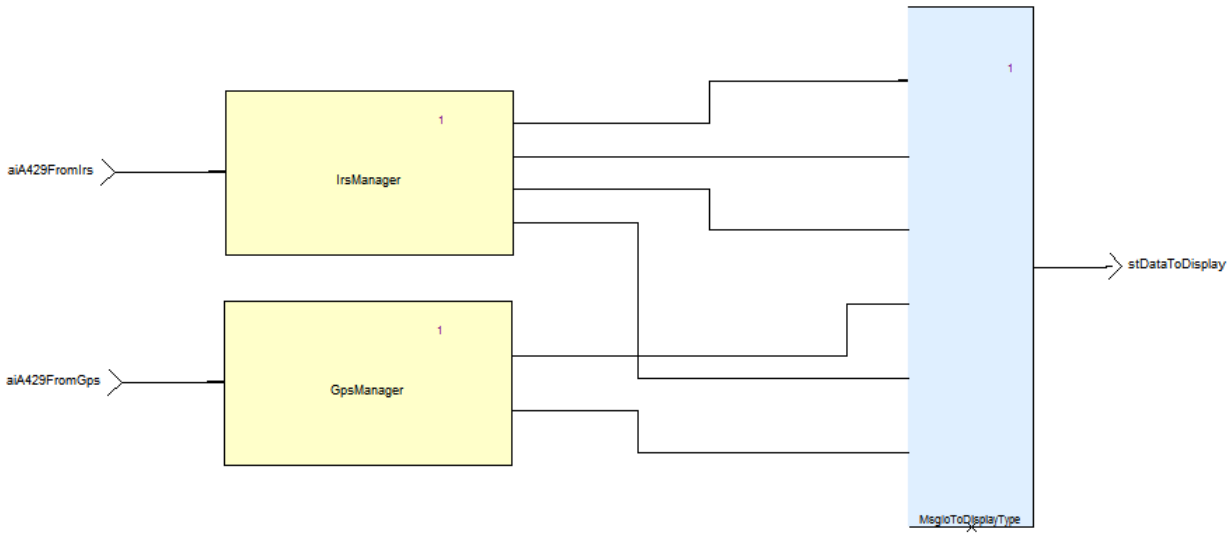


Fonte: O Autor

Ambos os módulos fazem interface com o sistema operacional afim de obter os dados que o driver Arinc 429 capturou. Dentro dos módulos IrsManager e GpsManager temos a decodificação das palavras A429 e seu empacotamento para enviar à partição Display como podemos notar na figura 5.8

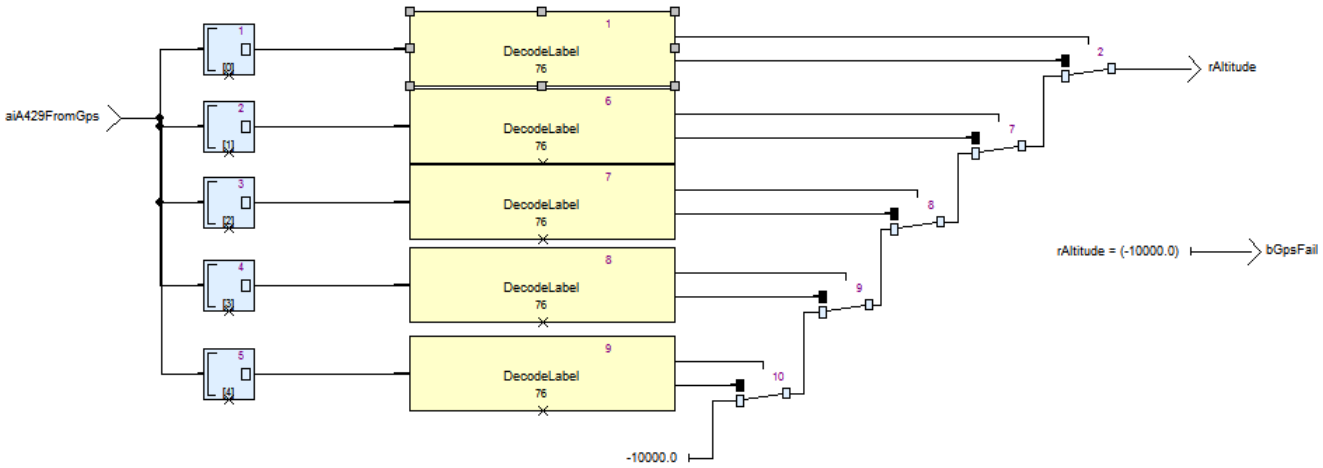
Cada um dos módulos realiza a decodificação das mensagens provenientes dos dispositivos como podemos ver na figura 5.9

Figura 5.8 – Modelo IO



Fonte: O Autor

Figura 5.9 – Modelo Gps Manager IO



Fonte: O Autor

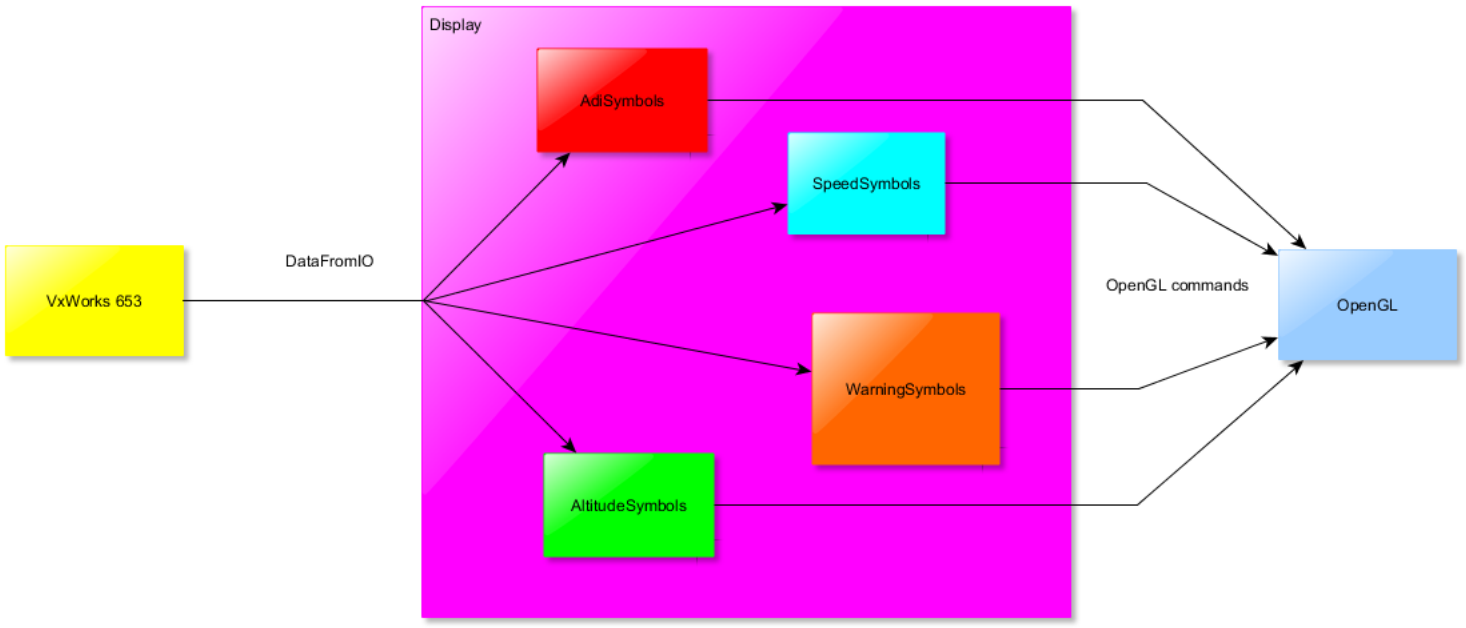
5.1.4 Partição Display

Internamente a partição de Display possui quatro componentes principais (figura 5.10) que recebem os dados da partição de IO e renderizam os componentes da tela baseado nas informações recebidas. Para cada componente da tela temos um módulo responsável pelo tratamento das mensagens e renderização das imagens.

Todos os módulos fazem interface com o sistema operacional para receber as mensagens provenientes da partição IO com os dados dos dispositivos decodificados para exibição como mostrado na figura 5.11.

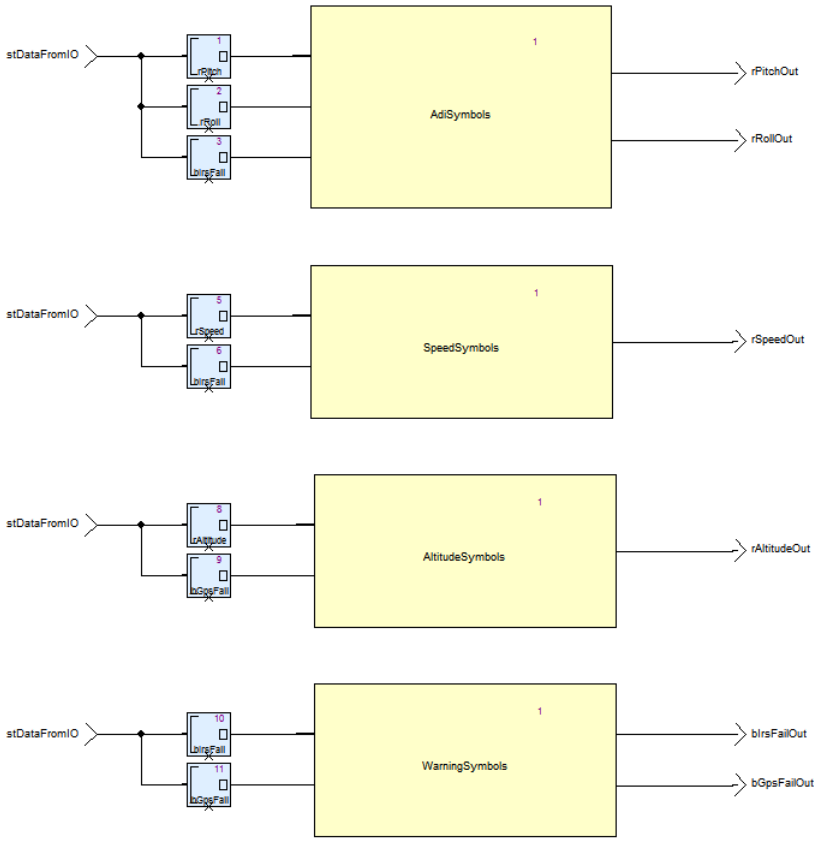
Cada um dos módulos possui dois sub-componentes. Um componente responsável por

Figura 5.10 – Arquitetura de Software Detalhada Display



Fonte: O Autor

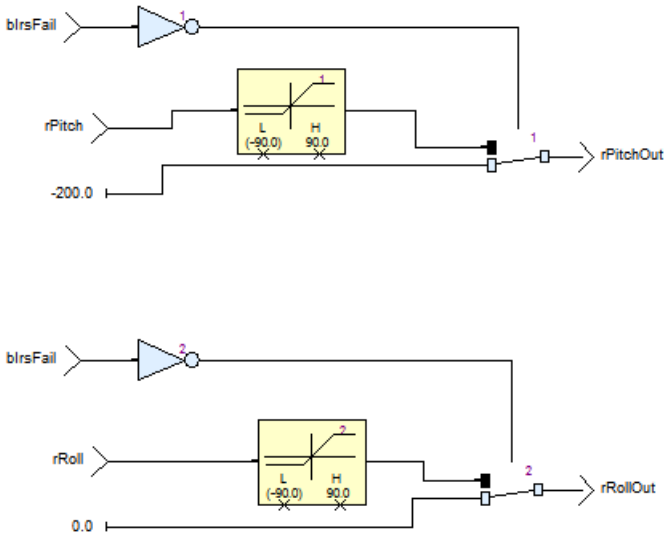
Figura 5.11 – Modelo Display



Fonte: O Autor

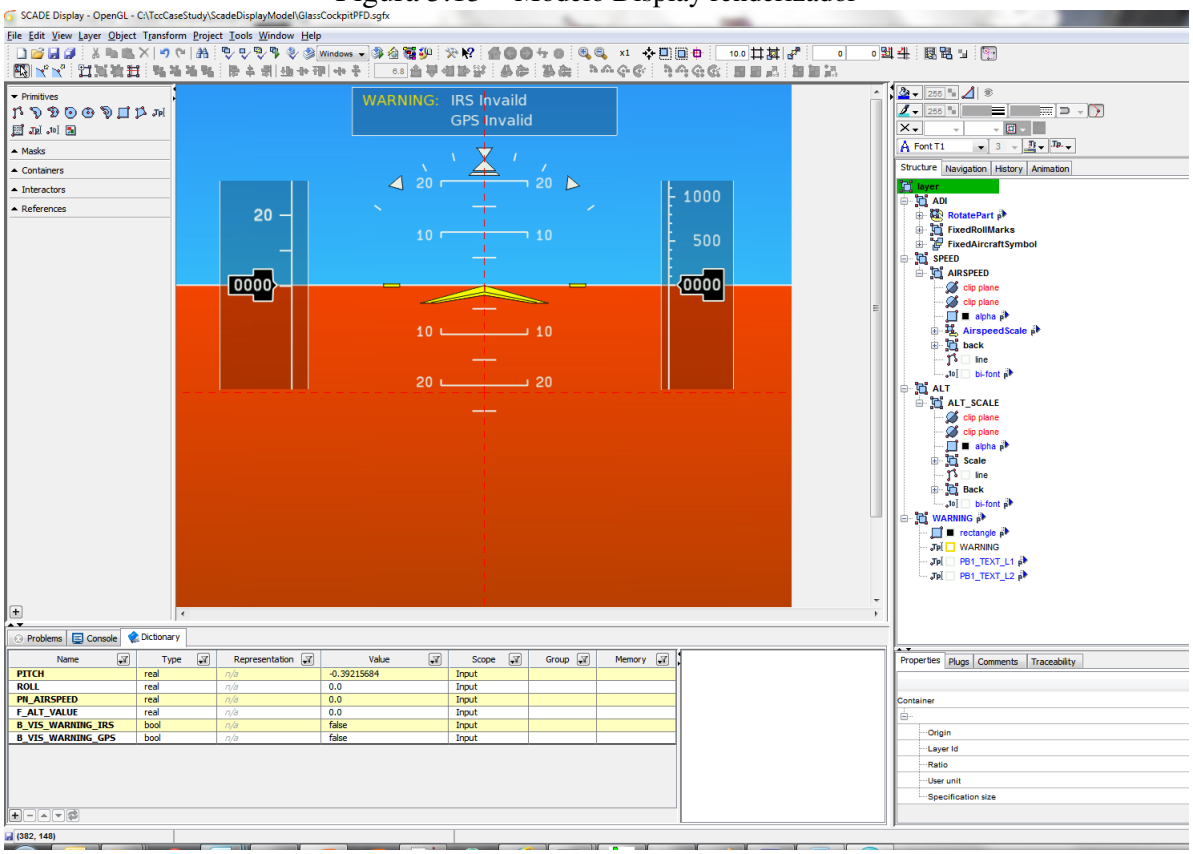
limitar os dados e a preparação final para a renderização (figura 5.12) e o componente renderizador (figura 5.13).

Figura 5.12 – Modelo Display Adi Symbols



Fonte: O Autor

Figura 5.13 – Modelo Display renderizador



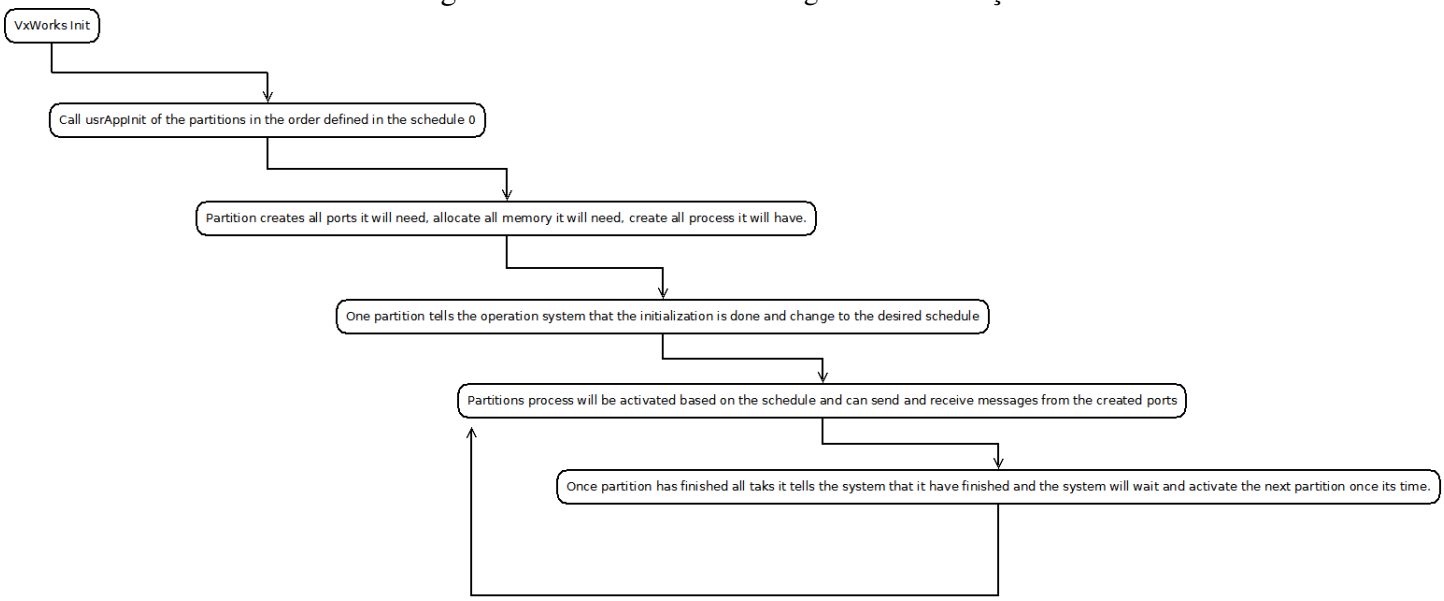
Fonte: O Autor

5.2 Partições e VxWorks 653

O sistema operacional utilizado neste trabalho para o desenvolvimento das partições e sua execução no computador target é o VxWorks 653 da Windriver. Esse sistema operacional foi criado para seguir a norma ARINC 653, que prevê uma série de garantias e exigências que sistemas operacionais de tempo real não forneciam anteriormente. Uma das principais características desse sistema operacional é seguir a norma IMA surgida no início dos anos 90 a fim de prover uma maior proteção para a camada de aplicação separando-a da camada de drivers. Nos anos 2000 esse padrão entrou na aviação civil através das duas companhias gigantes da aviação civil: Airbus, (no avião A380) e então a Boeing, no avião 787.

Esse sistema operacional possui um fluxo de execução bem particular como pode ser visto na figura 5.14

Figura 5.14 – VxWorks 653 Diagrama de Execução



Fonte: O Autor

Como características que diferenciam esse sistema operacional dos demais RTOS:

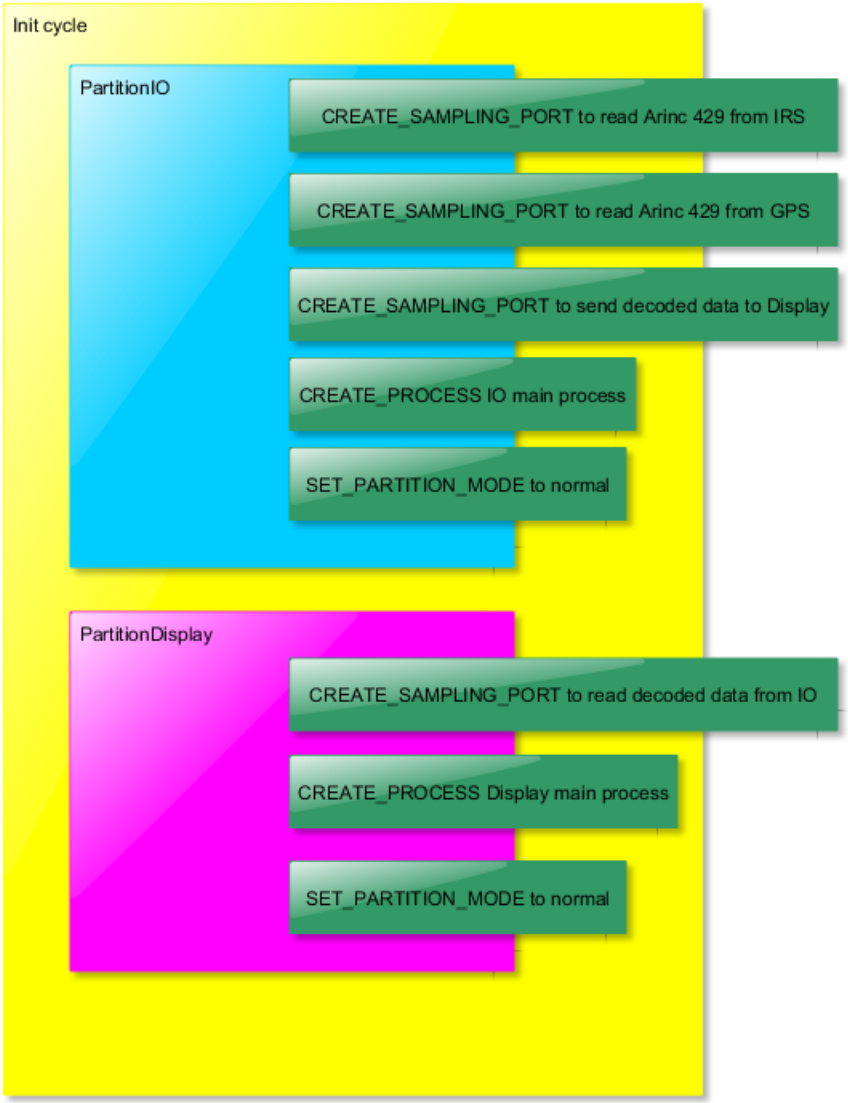
- Escalonador fixo
- Partições não trocam dados diretamente
- Proteção garantida de espaço e tempo
- Alocação estática de memória

Todas essas características têm como foco facilitar a certificação civil dos aviônicos e reduzir o custo de diversas atividades de análise de software.

5.2.1 Partições no VxWorks 653

Para receber os dados tanto dos drivers de dispositivo quanto mensagens das outras partições, no ciclo de inicialização as partições criam todas as portas para leitura e escrita assim como os processos necessários para a execução do código das partições, como pode ser visto na figura 5.15.

Figura 5.15 – VxWorks 653 Chamadas de Sistema na inicialização



Fonte: O Autor

As chamadas de CREATE SAMPLING PORT alocam memória dentro do espaço da partição para o recebimento de mensagens ou o envio.

As chamadas de CREATE PROCESS criam a thread que será executada nos ciclos subsequentes apontando uma função como ponto de entrada.

As chamadas de SET PARTITION MODE avisam o sistema operacional que a partição

concluiu o seu ciclo de inicialização e quando todas as partições indicam o modo como normal o sistema operacional troca do escalonador de inicialização para o escalonador de execução periódica.

Durante o ciclo de execução periódica outras chamadas de sistema são executadas como pode ser visto na figura 5.16.

Figura 5.16 – VxWorks 653 Chamadas de Sistema na execução periódica

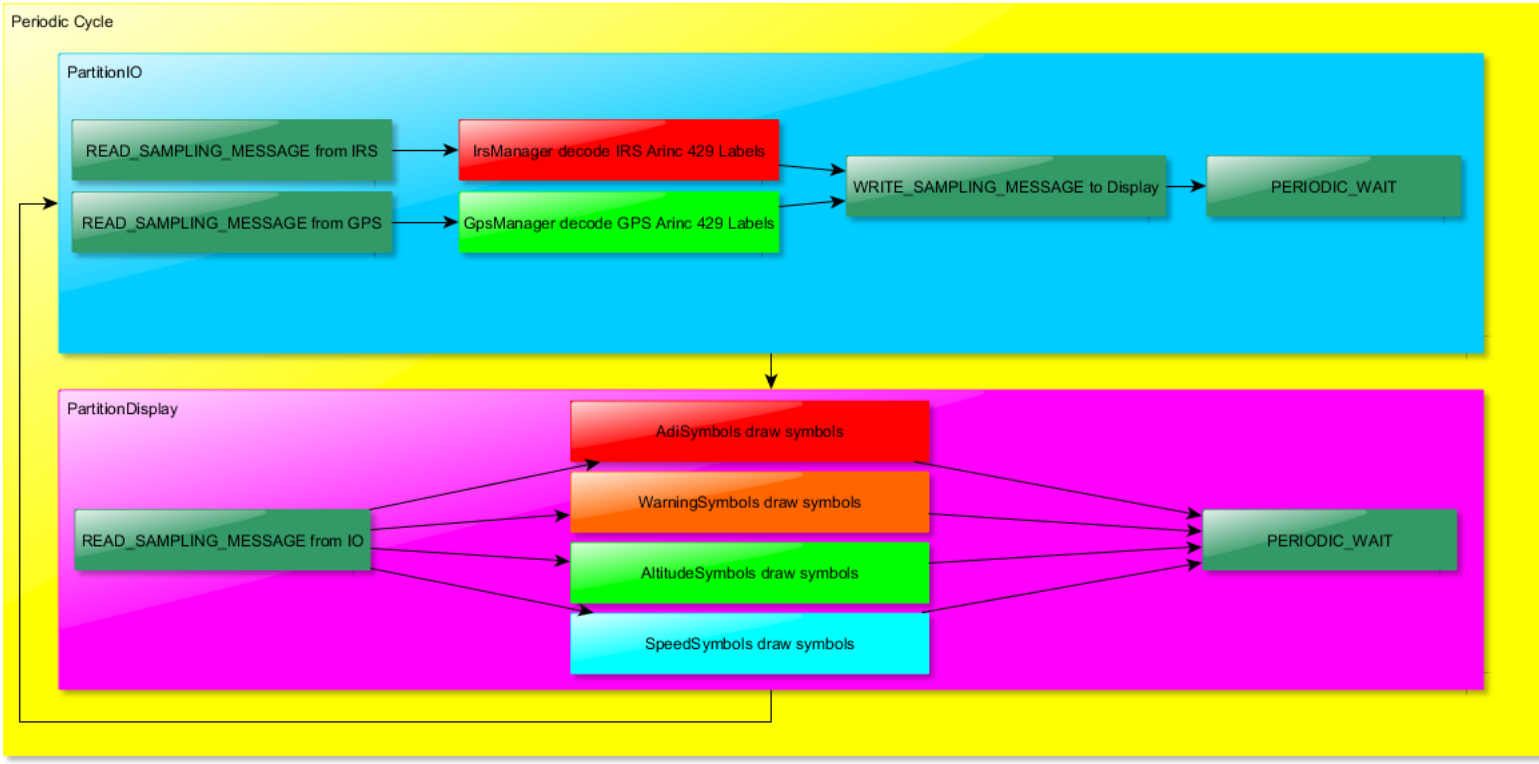
Fonte: O Autor

Para cada porta de leitura, a partição pode chamar READ SAMPLING PORT com o nome ou ID da porta para obter os dados provenientes daquela interface.

Para cada porta de escrita, a partição pode chamar WRITE SAMPLING MESSAGE com o nome ou ID da porta para enviar os dados para a porta.

Ao final da execução da partição, uma chamada à PERIODIC WAIT indica ao sistema operacional que a partição concluiu suas atividades e pode ser preemptada. Caso o tempo da partição expire e nenhuma chamada à PERIODIC WAIT tenha sido feita, o sistema operacional indica que a partição violou o seu espaço e o sistema pode ser encerrado ou seguir executando conforme for a configuração do sistema e a criticidade da partição.

Figura 5.17 – VxWorks 653 Chamadas de Sistema na execução periódica no caso de teste



Fonte: O Autor

No sistema desenvolvido para esse trabalho, a execução segue o seguinte fluxo no que tange as chamadas ao sistema operacional (figura 5.17). A partição de IO inicia sua execução periódica lendo as portas do dispositivo de IRS e de GPS para então decodificar as mensagens, unificar os dados e enviar para a partição Display os dados já decodificados. Então a partição de IO faz uma chamada de PERIODIC WAIT informando que seu ciclo periódico foi concluído. A partição de Display inicia sua execução periódica lendo a porta que recebe dados da partição IO. Então os dados são preparados e renderizados na tela para então a partição chamar PERIODIC WAIT e informar que seu ciclo periódico foi concluído.

5.3 Teste de Cobertura

Como forma de comprovar a metodologia proposta no trabalho, testes de cobertura serão desenvolvidos e realizados de posse de alguns dublês de teste desenvolvidos junto com a execução dos testes.

Inicialmente foram desenvolvidos objetos dummy para os dispositivos IRS e GPS baseados somente na sua interface ignorando qualquer tipo de comportamento. Um teste de cobertura foi executado com base nesses modelos. Logo após, *mock objects* foram desenvolvidos

para ambos os dispositivos consistindo de entradas constantes válidas segundo a interface dos dispositivos. Um teste de cobertura foi executado com base nesses modelos mais o anterior. Finalmente foi desenvolvido um objeto falso com implementação temporal correspondendo a interface e a periodicidade do envio das mensagens dos dispositivos segundo sua interface. Um teste de cobertura foi executado com base nesses modelos mais os anteriores.

5.3.1 Vector Cast COVER

Para a execução dos testes de cobertura foi utilizada a ferramenta VectorCast COVER (WARD, 2011) que é altamente conhecida no setor aeroespacial. Produzida pela empresa Vector Software que possui muita experiência no setor, começando em 1989 fornecendo suporte e consultoria e hoje tem um dos produtos mais utilizados no setor de testes.

Inicialmente o setup do sistema é configurado e os códigos fontes que desejamos analisar são adicionados ao projeto do VectorCast. Logo após o ambiente ser configurado, o VectorCast está pronto para instrumentar o código das partições a serem testadas. No caso deste trabalho a instrumentação é feita e os arquivos instrumentados são compilados utilizando o Microsoft Visual Studio 2010 já que o foco é o teste prematuro sem as partições e sistemas completamente finalizados. Os arquivos são instrumentados para o teste de cobertura de sentenças.

5.4 Resultados

Com o sistema preparado e os arquivos instrumentados partimos para a obtenção de resultados utilizando os diversos dublês de teste desenvolvidos.

5.4.1 Objetos dummy

O primeiro teste consiste na partição de IO instrumentada com os dispositivos de IRS e GPS desenvolvidos como Objetos dummy não fornecendo dado algum para a partição. O resultado do teste de cobertura pode ser analisado na figura 5.18.

Figura 5.18 – Coverage Dummy Object

Unit	Subprogram	Complexity	Statements
BoolVect2Int_digital_19.c	BoolVect2Int_digital_19	3	85% (6 / 7)
TOTALS	**1**	**3**	**85% (6 / 7)**
BoolVect2Int_digital_2.c	BoolVect2Int_digital_2	3	85% (6 / 7)
TOTALS	**1**	**3**	**85% (6 / 7)**
BoolVect2Int_digital_8.c	BoolVect2Int_digital_8	3	85% (6 / 7)
TOTALS	**1**	**3**	**85% (6 / 7)**
DecodeLabel.c	DecodeLabel	1	100% (5 / 5)
TOTALS	**1**	**1**	**100% (5 / 5)**
GetData.c	GetData	1	100% (3 / 3)
TOTALS	**1**	**1**	**100% (3 / 3)**
GetLabelNumber.c	GetLabelNumber	1	100% (3 / 3)
TOTALS	**1**	**1**	**100% (3 / 3)**
GetSSM.c	GetSSM	1	100% (3 / 3)
TOTALS	**1**	**1**	**100% (3 / 3)**
GpsManager.c	GpsManager	6	72% (13 / 18)
TOTALS	**1**	**6**	**72% (13 / 18)**
Int2BoolVect_digital_32.c	Int2BoolVect_digital_32	3	100% (6 / 6)
TOTALS	**1**	**3**	**100% (6 / 6)**
Int2BoolVectElt_digital.c	Int2BoolVectElt_digital	1	100% (2 / 2)
TOTALS	**1**	**1**	**100% (2 / 2)**
IoManager.c	IoManager_reset	1	100% (1 / 1)
	IoManager	1	100% (6 / 6)
TOTALS	**2**	**2**	**100% (7 / 7)**
IrsManager.c	IrsManager	16	70% (36 / 51)
TOTALS	**1**	**16**	**70% (36 / 51)**
GRAND TOTALS	**13**	**41**	**80% (96 / 119)**

Fonte: O Autor

Podemos ver que os arquivos fonte que cuidam da decodificação dos dados foram completamente cobertos já que não possuem nenhum tipo de lógica condicional como a função DecodeLabel por exemplo na (figura 5.19) enquanto as funções de aquisição de dados como BoolVect2Int digital 19 obtiveram um percentual inferior.

Figura 5.19 – Coverage Dummy Object DecodeLabel

Code Coverage for Unit: DecodeLabel.c

```
-- Coverage Type: Statement
-- Unit: DecodeLabel.c
-- Test Case: Aggregate
               /* $*************** KCG Version 6.1.3 (build i6) ***************
               ** Command: s2c613 -config C:/TccCaseStudy/ScadeSuiteModel/Io/KCG\kcg_s2c_config.txt
               ** Generation date: 2015-05-16T17:12:34
               *************************************************************$ */
               #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\kcg_consts.h"
               #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\kcg_sensors.h"
               #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\DecodeLabel.h"
               /* DecodeLabel */
               void DecodeLabel(
                 /* DecodeLabel::iA429Word */kcg_int iA429Word,
                 /* DecodeLabel::iLabelNumber */kcg_int iLabelNumber,
                 /* DecodeLabel::bValidLabel */kcg_bool *bValidLabel,
                 /* DecodeLabel::rData */kcg_real *rData)
               {
                 kcg_int tmp1;
                 kcg_int tmp;
1 1      *       tmp1 = /* 1 */ GetData(iA429Word);
1 2      *       *rData = (kcg_real) tmp1;
1 3      *       tmp1 = /* 1 */ GetLabelNumber(iA429Word);
1 4      *       tmp = /* 1 */ GetSSM(iA429Word);
1 5      *       *bValidLabel = (tmp1 == iLabelNumber) & (tmp == 3);
               }
               /* $*************** KCG Version 6.1.3 (build i6) ***************
               ** DecodeLabel.c
               ** Generation date: 2015-05-16T17:12:34
               *************************************************************$ */
TEST COVERAGE SUMMARY


 5 of 5 Lines Covered ( 100% )
```

Fonte: O Autor

Já as funções que dependem das entradas possuindo lógica condicional não atingiram 100% de cobertura, como a função GpsManager, por exemplo na figura 5.20, onde a cadeia de "If else" ficou em vermelha significando que a execução não passou por ali. Logo será necessário um modelo mais complexo para exercitar todo esse código.

Figura 5.20 – Coverage Dummy Object GpsManager

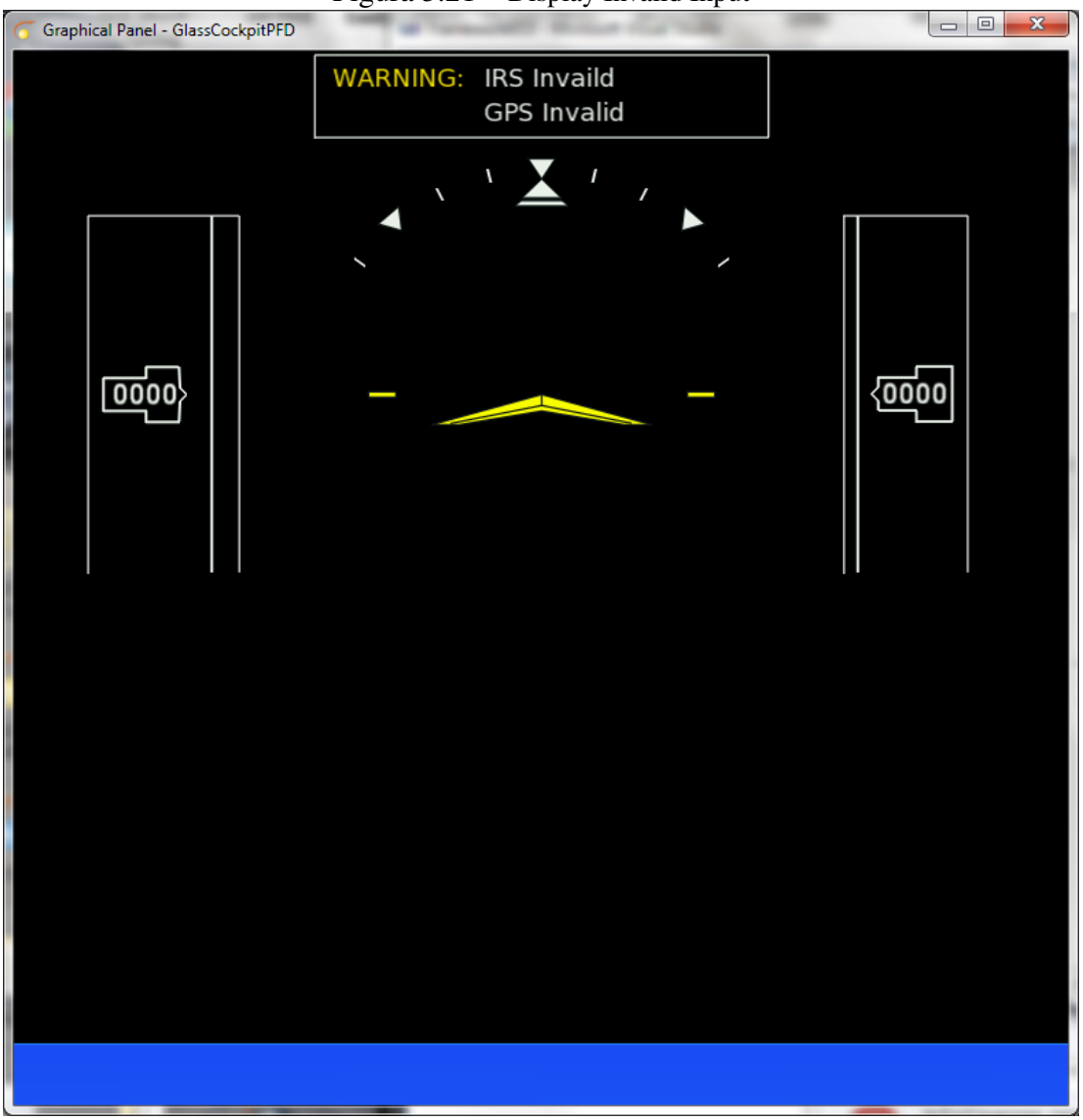
```
Code Coverage for Unit: GpsManager.c
-- Coverage Type: Statement
-- Unit: GpsManager.c
-- Test Case: Aggregate
                /* $*************** KCG Version 6.1.3 (build i6) ***************
                ** Command: s2c613 -config C:/TccCaseStudy/ScadeSuiteModel/Io/KCG\kcg_s2c_config.txt
                ** Generation date: 2015-05-16T17:12:34
                *************************************************************$ */
                #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\kcg_consts.h"
                #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\kcg_sensors.h"
                #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\GpsManager.h"
                /* GpsManager */
                void GpsManager(
                  /* GpsManager::aiA429FromGps */array_int_5 *aiA429FromGps,
                  /* GpsManager::rAltitude */kcg_real *rAltitude,
                  /* GpsManager::bGpsFail */kcg_bool *bGpsFail)
                {
                  /* GpsManager::_L18 */ kcg_real _L18;
                  /* GpsManager::_L17 */ kcg_bool _L17;
                  /* GpsManager::_L33 */ kcg_bool _L33;
                  /* GpsManager::_L34 */ kcg_real _L34;
                  /* GpsManager::_L37 */ kcg_bool _L37;
                  /* GpsManager::_L38 */ kcg_real _L38;
                  /* GpsManager::_L40 */ kcg_bool _L40;
                  /* GpsManager::_L41 */ kcg_real _L41;
                  /* GpsManager::_L42 */ kcg_bool _L42;
                  /* GpsManager::_L43 */ kcg_real _L43;
                  /* GpsManager::_L45 */ kcg_real _L45;
                  /* 1 */
1 1       *       DecodeLabel((*aiA429FromGps)[0], 76, &_L17, &_L18);
                  /* 6 */
1 2       *       DecodeLabel((*aiA429FromGps)[1], 76, &_L33, &_L34);
                  /* 7 */
1 3       *       DecodeLabel((*aiA429FromGps)[2], 76, &_L37, &_L38);
                  /* 8 */
1 4       *       DecodeLabel((*aiA429FromGps)[3], 76, &_L40, &_L41);
                  /* 9 */
1 5       *       DecodeLabel((*aiA429FromGps)[4], 76, &_L42, &_L43);
1 6       *       _L45 = - 10000.0;
1 7       *       if (_L17) {
1 8                 *rAltitude = _L18;
                  }
                  else
1 9       *           if (_L33) {
1 10                    *rAltitude = _L34;
                  }
                  else
1 11      *             if (_L37) {
1 12                    *rAltitude = _L38;
                  }
                  else
1 13      *             if (_L40) {
1 14                    *rAltitude = _L41;
                  }
                  else
1 15      *             if (_L42) {
1 16                    *rAltitude = _L43;
                  }
                  else {
1 17      *             *rAltitude = _L45;
                  }
1 18      *       *bGpsFail = *rAltitude == _L45;
                }
                /* $*************** KCG Version 6.1.3 (build i6) ***************
                ** GpsManager.c
                ** Generation date: 2015-05-16T17:12:34
                *************************************************************$ */
TEST COVERAGE SUMMARY


 13 of 18 Lines Covered ( 72% )
```

Fonte: O Autor

Podemos ver também o resultado da execução da partição Display na figura 5.21

Nota-se que o sistema apresenta informações alertando o piloto que ambos os dispositi-

Figura 5.21 – Display Invalid Input



Fonte: O Autor

vos GPS e IRS estão com problemas e evita fornecer qualquer informação que possa enganar o piloto.

5.4.2 Mock Objects

Nesse segundo teste, a partição IO foi instrumentada e os dispositivos IRS e GPS desenvolvidos como mock objects possuindo dados de acordo com a interface, mas fornecendo dados fixos com pequena implementação. Nesse caso palavras Arinc 429 válidas e inválidas foram fornecidas na interface dos dispositivos. O resultado do teste de cobertura pode ser analisado na figura 5.22

Figura 5.22 – Coverage Mock Object

Unit	Subprogram	Complexity	Statements
BoolVect2Int_digital_19.c	BoolVect2Int_digital_19	3	100% (7 / 7)
TOTALS	**1**	**3**	**100% (7 / 7)**
BoolVect2Int_digital_2.c	BoolVect2Int_digital_2	3	100% (7 / 7)
TOTALS	**1**	**3**	**100% (7 / 7)**
BoolVect2Int_digital_8.c	BoolVect2Int_digital_8	3	100% (7 / 7)
TOTALS	**1**	**3**	**100% (7 / 7)**
DecodeLabel.c	DecodeLabel	1	100% (5 / 5)
TOTALS	**1**	**1**	**100% (5 / 5)**
GetData.c	GetData	1	100% (3 / 3)
TOTALS	**1**	**1**	**100% (3 / 3)**
GetLabelNumber.c	GetLabelNumber	1	100% (3 / 3)
TOTALS	**1**	**1**	**100% (3 / 3)**
GetSSM.c	GetSSM	1	100% (3 / 3)
TOTALS	**1**	**1**	**100% (3 / 3)**
GpsManager.c	GpsManager	6	50% (9 / 18)
TOTALS	**1**	**6**	**50% (9 / 18)**
Int2BoolVect_digital_32.c	Int2BoolVect_digital_32	3	100% (6 / 6)
TOTALS	**1**	**3**	**100% (6 / 6)**
Int2BoolVectElt_digital.c	Int2BoolVectElt_digital	1	100% (2 / 2)
TOTALS	**1**	**1**	**100% (2 / 2)**
IoManager.c	IoManager_reset	1	100% (1 / 1)
	IoManager	1	100% (6 / 6)
TOTALS	**2**	**2**	**100% (7 / 7)**
IrsManager.c	IrsManager	16	52% (27 / 51)
TOTALS	**1**	**16**	**52% (27 / 51)**
GRAND TOTALS	**13**	**41**	**72% (86 / 119)**

Fonte: O Autor

Podemos ver que todos os nodos de decodificação dos dados foram completamente exercitados já que não possuem nenhuma lógica temporal e os nodos de aquisição de dados também obtiveram percentual completo de cobertura. Entretanto, os nodos de controle como o GpsManager e IrsManager obtiveram uma cobertura inferior ao teste com os objetos dummy, como pode ser visto na figura 5.23.

Figura 5.23 – Coverage Mock Object GpsManager

```
Code Coverage for Unit: GpsManager.c
-- Coverage Type: Statement
-- Unit: GpsManager.c
-- Test Case: Aggregate
                /* $************** KCG Version 6.1.3 (build i6) ***************
                ** Command: s2c613 -config C:/TccCaseStudy/ScadeSuiteModel/Io/KCG\kcg_s2c_config.txt
                ** Generation date: 2015-05-30T15:20:09
                *************************************************************$ */
                #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\kcg_consts.h"
                #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\kcg_sensors.h"
                #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\GpsManager.h"
                /* GpsManager */
                void GpsManager(
                  /* GpsManager::aiA429FromGps */array_int_5 *aiA429FromGps,
                  /* GpsManager::rAltitude */kcg_real *rAltitude,
                  /* GpsManager::bGpsFail */kcg_bool *bGpsFail)
                {
                  /* GpsManager::_L18 */ kcg_real _L18;
                  /* GpsManager::_L17 */ kcg_bool _L17;
                  /* GpsManager::_L33 */ kcg_bool _L33;
                  /* GpsManager::_L34 */ kcg_real _L34;
                  /* GpsManager::_L37 */ kcg_bool _L37;
                  /* GpsManager::_L38 */ kcg_real _L38;
                  /* GpsManager::_L40 */ kcg_bool _L40;
                  /* GpsManager::_L41 */ kcg_real _L41;
                  /* GpsManager::_L42 */ kcg_bool _L42;
                  /* GpsManager::_L43 */ kcg_real _L43;
                  /* GpsManager::_L45 */ kcg_real _L45;
                  /* 1 */
1 1       *       DecodeLabel((*aiA429FromGps)[0], 76, &_L17, &_L18);
                  /* 6 */
1 2       *       DecodeLabel((*aiA429FromGps)[1], 76, &_L33, &_L34);
                  /* 7 */
1 3       *       DecodeLabel((*aiA429FromGps)[2], 76, &_L37, &_L38);
                  /* 8 */
1 4       *       DecodeLabel((*aiA429FromGps)[3], 76, &_L40, &_L41);
                  /* 9 */
1 5       *       DecodeLabel((*aiA429FromGps)[4], 76, &_L42, &_L43);
1 6       *       _L45 = - 10000.0;
1 7       *       if (_L17) {
1 8       *          *rAltitude = _L18;
                  }
                  else
1 9                  if (_L33) {
1 10                      *rAltitude = _L34;
                  }
                  else
1 11                      if (_L37) {
1 12                      *rAltitude = _L38;
                  }
                  else
1 13                      if (_L40) {
1 14                      *rAltitude = _L41;
                  }
                  else
1 15                      if (_L42) {
1 16                      *rAltitude = _L43;
                  }
                  else {
1 17                      *rAltitude = _L45;
                  }
1 18      *       *bGpsFail = *rAltitude == _L45;
                }
                /* $************** KCG Version 6.1.3 (build i6) ***************
                ** GpsManager.c
                ** Generation date: 2015-05-30T15:20:09
                *************************************************************$ */
TEST COVERAGE SUMMARY

 9 of 18 Lines Covered ( 50% )
```

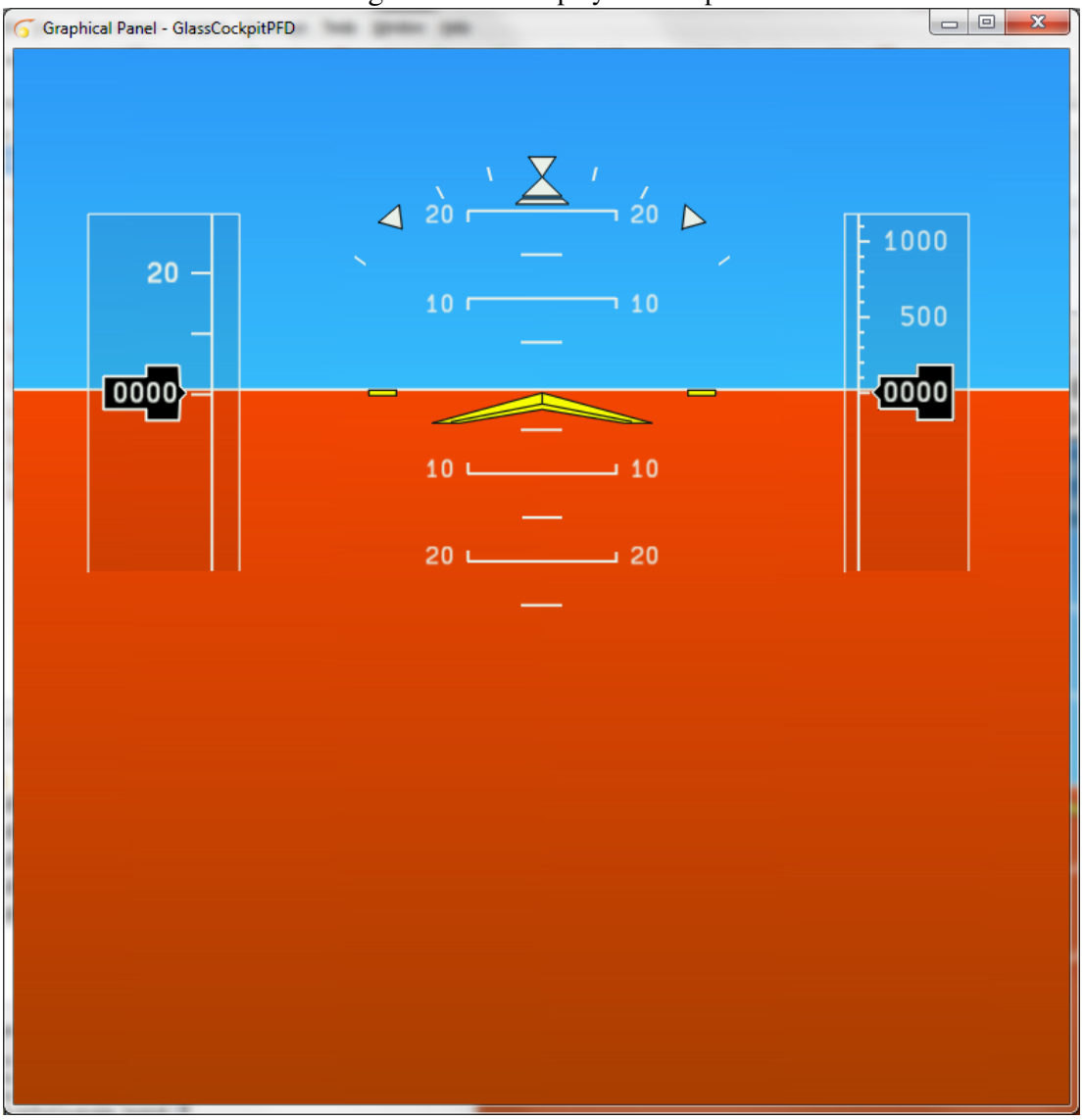
Fonte: O Autor

Esse comportamento se deu pela estrutura do código que, ao encontrar um dado válido, não termina sua cadeia "If else" enquanto que no teste com Dummy Objects, como não havia

entradas, a função percorria toda a cadeia para cair no último else. Logo será necessário um modelo ainda mais complexo para exercitar todo esse código.

Podemos ver também o resultado da execução da partição Display na figura 5.24

Figura 5.24 – Display Valid Input



Fonte: O Autor

Nota-se que o sistema apresenta informações válidas, mas as escalas e a ADI encontram-se paradas em valores estáticos.

5.4.3 Objeto Falso

Nesse terceiro teste, a partição IO foi instrumentada e os dispositivos IRS e GPS desenvolvidos como objetos falsos possuindo dados que variavam de acordo com o tempo seguindo as interfaces e temporização descritas na interface dos dispositivos. Logo, esses modelos pos-

suíam palavras ARINC 429 que variavam no tempo fornecendo uma entrada bastante próxima à realidade para o software de aplicação contendo dados válidos e inválidos. O resultado do teste de cobertura pode ser analisada na figura 5.25.

Figura 5.25 – Coverage Mock False Object

Unit	Subprogram	Complexity	Statements
BoolVect2Int_digital_19.c	BoolVect2Int_digital_19	3	100% (7 / 7)
TOTALS	**1**	**3**	**100% (7 / 7)**
BoolVect2Int_digital_2.c	BoolVect2Int_digital_2	3	100% (7 / 7)
TOTALS	**1**	**3**	**100% (7 / 7)**
BoolVect2Int_digital_8.c	BoolVect2Int_digital_8	3	100% (7 / 7)
TOTALS	**1**	**3**	**100% (7 / 7)**
DecodeLabel.c	DecodeLabel	1	100% (5 / 5)
TOTALS	**1**	**1**	**100% (5 / 5)**
GetData.c	GetData	1	100% (3 / 3)
TOTALS	**1**	**1**	**100% (3 / 3)**
GetLabelNumber.c	GetLabelNumber	1	100% (3 / 3)
TOTALS	**1**	**1**	**100% (3 / 3)**
GetSSM.c	GetSSM	1	100% (3 / 3)
TOTALS	**1**	**1**	**100% (3 / 3)**
GpsManager.c	GpsManager	6	100% (18 / 18)
TOTALS	**1**	**6**	**100% (18 / 18)**
Int2BoolVect_digital_32.c	Int2BoolVect_digital_32	3	100% (6 / 6)
TOTALS	**1**	**3**	**100% (6 / 6)**
Int2BoolVectElt_digital.c	Int2BoolVectElt_digital	1	100% (2 / 2)
TOTALS	**1**	**1**	**100% (2 / 2)**
IoManager.c	IoManager_reset	1	100% (1 / 1)
	IoManager	1	100% (6 / 6)
TOTALS	**2**	**2**	**100% (7 / 7)**
IrsManager.c	IrsManager	16	100% (51 / 51)
TOTALS	**1**	**16**	**100% (51 / 51)**
GRAND TOTALS	**13**	**41**	**100% (119 / 119)**

Fonte: O Autor

Podemos ver que todos os nodos foram completamente exercitados inclusive os nodos que dependiam de lógica temporal como GpsManager e IrsManager na figura 5.26.

Figura 5.26 – Coverage Mock False Object GpsManager

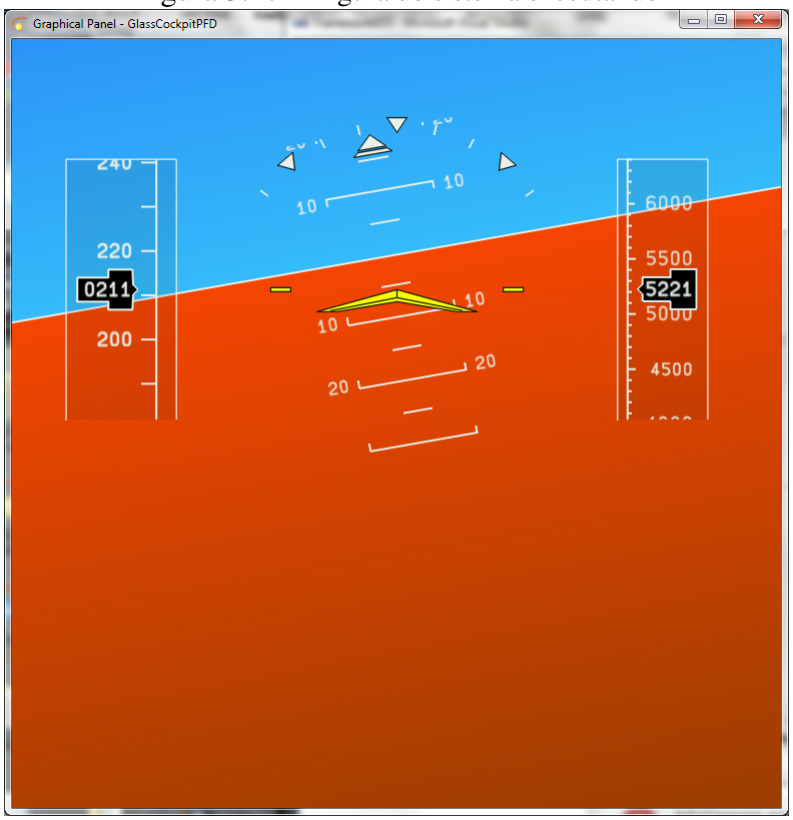
```
Code Coverage for Unit: GpsManager.c
-- Coverage Type: Statement
-- Unit: GpsManager.c
-- Test Case: Aggregate
                /* $************** KCG Version 6.1.3 (build i6) ***************
                ** Command: s2c613 -config C:/TccCaseStudy/ScadeSuiteModel/Io/KCG\kcg_s2c_config.txt
                ** Generation date: 2015-05-30T15:20:09
                *************************************************************$ */
                #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\kcg_consts.h"
                #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\kcg_sensors.h"
                #include "C:\\TccCaseStudy\\ScadeSuiteModel\\Io\\KCG\\GpsManager.h"
                /* GpsManager */
                void GpsManager(
                  /* GpsManager::aiA429FromGps */array_int_5 *aiA429FromGps,
                  /* GpsManager::rAltitude */kcg_real *rAltitude,
                  /* GpsManager::bGpsFail */kcg_bool *bGpsFail)
                {
                  /* GpsManager::_L18 */ kcg_real _L18;
                  /* GpsManager::_L17 */ kcg_bool _L17;
                  /* GpsManager::_L33 */ kcg_bool _L33;
                  /* GpsManager::_L34 */ kcg_real _L34;
                  /* GpsManager::_L37 */ kcg_bool _L37;
                  /* GpsManager::_L38 */ kcg_real _L38;
                  /* GpsManager::_L40 */ kcg_bool _L40;
                  /* GpsManager::_L41 */ kcg_real _L41;
                  /* GpsManager::_L42 */ kcg_bool _L42;
                  /* GpsManager::_L43 */ kcg_real _L43;
                  /* GpsManager::_L45 */ kcg_real _L45;
                  /* 1 */
1 1       *       DecodeLabel((*aiA429FromGps)[0], 76, &_L17, &_L18);
                  /* 6 */
1 2       *       DecodeLabel((*aiA429FromGps)[1], 76, &_L33, &_L34);
                  /* 7 */
1 3       *       DecodeLabel((*aiA429FromGps)[2], 76, &_L37, &_L38);
                  /* 8 */
1 4       *       DecodeLabel((*aiA429FromGps)[3], 76, &_L40, &_L41);
                  /* 9 */
1 5       *       DecodeLabel((*aiA429FromGps)[4], 76, &_L42, &_L43);
1 6       *       _L45 = - 10000.0;
1 7       *       if (_L17) {
1 8       *          *rAltitude = _L18;
                  }
                  else
1 9       *          if (_L33) {
1 10      *             *rAltitude = _L34;
                  }
                  else
1 11      *             if (_L37) {
1 12      *             *rAltitude = _L38;
                  }
                  else
1 13      *             if (_L40) {
1 14      *             *rAltitude = _L41;
                  }
                  else
1 15      *             if (_L42) {
1 16      *             *rAltitude = _L43;
                  }
                  else {
1 17      *             *rAltitude = _L45;
                  }
1 18      *       *bGpsFail = *rAltitude == _L45;
                }
                /* $************** KCG Version 6.1.3 (build i6) ***************
                ** GpsManager.c
                ** Generation date: 2015-05-30T15:20:09
                *************************************************************$ */
TEST COVERAGE SUMMARY

 18 of 18 Lines Covered ( 100% )
```

Fonte: O Autor

Como era esperado, quando a entrada variou no tempo a cadeia "If else"foi exercitada por completo então foi obtido a cobertura completa do código.

Podemos ver a execução da partição Display com os dados válidos na figura 5.27

Figura 5.27 – Figura do sistema executando



Fonte: O Autor

Nota-se que o sistema apresenta informações válidas e as escalas e a ADI encontram-se com valores e em movimento.

6 TRABALHOS RELACIONADOS

O primeiro objetivo do trabalho consistiu em desenvolver e implementar um framework de desenvolvimento que provenha todas as chamadas de sistema necessárias para conectar um software de aplicação a uma gama de dispositivos a fim de realizar o teste em ambiente x86. Outros projetos nesse sentido foram desenvolvidos como em (KOONG et al., 2010), (VANDERLEEST, 2010) e (GOMES, 2010). Em (KOONG et al., 2010) foi desenvolvido um framework com filosofia semelhante ao proposto neste trabalho, no entanto a plataforma alvo não possuía SO. Em (VANDERLEEST, 2010) foi desenvolvido um hypervisor com o intuito de estabelecer uma máquina virtual que executasse um sistema operacional compatível com ARINC 653, o que ajuda em muito o teste formal, mas apresenta diversas dificuldades para o teste em tempo de desenvolvimento, já que a aplicação teria que ser inserida na máquina virtual e ainda assim não teríamos os dispositivos nem uma maior integração para simplificar os testes. Em (GOMES, 2010) foi desenvolvido e aplicado um framework para teste focado em sistemas embarcados, mas novamente a plataforma destino executava um aplicativo único sem a presença de SO.

De posse do framework e da aplicação, o desenvolvedor poderá compilar, executar e depurar sua aplicação em um ambiente de desenvolvimento desktop. As chamadas de sistema implementadas de maneira a simular o comportamento do sistema operacional VxWorks 653 não necessitam ter nenhuma similaridade com a implementação real, mas sim o compromisso de fornecer a mesma interface e comportamento. As principais funcionalidades são a gerência de tarefas e comunicação entre processos. Os dispositivos são simulados através de técnicas de mocking (KARLESKY et al., 2007) e modelos simplificados do seu funcionamento. Utilizando um template para a implementação de novos dispositivos. A profundidade do modelo implementado pode variar desde um modelo superficial do comportamento, até um modelo bastante complexo simulando características mais realistas do dispositivo. Importante ressaltar que é nos modelos dos dispositivos que o trabalho proverá grande contribuição já que lidar com dispositivos reais é uma tarefa complexa, ainda mais complexo é lidar com situações onde eles falham. Situações como falha em comunicações, ruído em linhas e uma ampla gama de falhas poderão ser simuladas. A implementação e alteração dos dispositivos fica a cargo do time de desenvolvimento, pois eles que sabem a necessidade dos dispositivos e qual é a complexidade necessária. As funções que substituirão as chamadas de SO serão implementadas na mesma linguagem de programação da aplicação assim como os modelos para os dispositivos. Os dispositivos serão métodos que executam após todas as partições de software a cada ciclo, dessa forma não interferindo no funcionamento da aplicação e provendo informação através dos

canais de comunicação.

7 CONCLUSÃO

O trabalho demonstrou uma metodologia de desenvolvimento e testes de software embarcado com o objetivo de acelerar a maturidade do software utilizando ferramentas e ambientes típicos desktop. Dessa forma durante todo o desenvolvimento é observada uma maior qualidade de software proveninente da facilidade em testar e automatizar processos que não é atingida facilmente em ambientes embarcados.

No trabalho foi explicitado como proceder com a integração e tarefas importantes ao compilar e executar um software produzido para um sistema operacional de tempo real a executar em um sistema operacional de propósito geral. A construção de modelos de dispositivos de hardware em software permite que a aplicação a ser embarcada possa ser executada completamente em ambiente desktop com um comportamento muito próximo ao esperado da execução no sistema embarcado real. Mais do que permitir a execução, o ambiente é capaz de exercitar o software em casos muito complexos de serem criados em laboratório como casos de falhas. Problemas esses que só seriam detectados em um estágio muito posterior do processo de desenvolvimento onde os custos serão muito maiores e a flexibilidade é muito menor. Esses benefícios são atingidos com poucas linhas de código que são capazes de simular um complexo equipamento.

É necessário ressaltar que um tempo é necessário para desenvolver os modelos de dispositivos de hardware, o que pode consumir tempo importante no processo de desenvolvimento de software. O importante é manter o foco no desenvolvimento do projeto e que o modelo seja implementado tanto quanto for necessário para testar o software. De maneira alguma é necessário recriar o dispositivo em software. Muito utilizada neste trabalho foi a coleta dos dados do dispositivo real e utilizado no modelo, inserindo-se casos complexos de falha comummente observados nos testes finais de integração.

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

8 TRABALHOS FUTUROS

Conforme já explicitado, existem inúmeras vantagens de se executar o software a ser embarcado em ambiente desktop. Nesse trabalho o foco foi em prover para o nível de aplicação mensagens que seriam recebidas dos dispositivos de hardware. No entanto existe uma camada abaixo da camada de aplicação que fica de fora deste enfoque: os drivers de dispositivo. Seria possível ampliar ainda mais a simulação e gerar IoReads e IoWrites semelhantes ao comportamento real dos dispositivos e protocolos e assim testar mais uma camada de software.

Ampliar o suporte para artefatos do sistema operacional embarcado tornando mais automático o framework que assim poderia automaticamente saber as aplicações a serem executadas em inicialização e em ciclos periódicos. Assim como prover mais verificações nas leituras e escritas nas portas APEX verificando se a porta foi criada corretamente e se está conectada corretamente, se o tamanho do buffer está correto entre outras verificações específicas do sistema operacional que não foram abordadas em detalhes neste trabalho.

Permitir a utilização de múltiplas threads em cada aplicação. Algo que é suportado pelo sistema operacional, mas o framework não provê nenhum tipo de suporte atualmente.

Regular o tempo de execução como o sistema operacional de tempo real já que embora executando em sistema operacional de propósito geral devido à ampla abundância de recursos de memória e processador acredita-se que o período de execução poderia ser seguido fornecendo um comportamento mais realista para a aplicação.

No endereço: https://github.com/nortonlb/AvionicsSoftwareTestingVirtualPlataform.

Encontram-se os fontes e arquivos gerados durante a execução do trabalho. Como um trabalho aberto convido a todos para colaborar com o desenvolvimento deste projeto.

REFERÊNCIAS

FREEMAN, S.; PRYCE, N. **Growing object-oriented software, guided by tests**. [S.l.]: Pearson Education, 2009.

GOMES, H. V. Metodologia de projeto de software embarcado voltada ao teste. 2010.

HAYHURST, K. J. et al. **A practical tutorial on modified condition/decision coverage**. [S.l.]: National Aeronautics and Space Administration, Langley Research Center, 2001.

HILDERMAN, V.; BAGHI, T. **Avionics certification: a complete guide to DO-178 (software), DO-254 (hardware)**. [S.l.]: Avionics Communications, 2007.

JANG, S.-J.; KIM, H.-G.; CHUNG, Y.-K. Manual specific testing and quality evaluation for embedded software. In: IEEE. **Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on**. [S.l.], 2008. p. 502–507.

KANER, C. Exploratory testing. In: **Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL**. [S.l.: s.n.], 2006.

KARLESKY, M. et al. Mocking the embedded world: Test-driven development, continuous integration, and design patterns. In: **Proc. Emb. Systems Conf, CA, USA**. [S.l.: s.n.], 2007.

KIM, Y. W. Efficient use of code coverage in large-scale software development. In: IBM PRESS. **Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research**. [S.l.], 2003. p. 145–155.

KOONG, C.-S. et al. Supporting tool for embedded software testing. In: IEEE. **Quality Software (QSIC), 2010 10th International Conference on**. [S.l.], 2010. p. 481–487.

LITTLEFIELD-LAWWILL, J.; KINNAN, L. System considerations for robust time and space partitioning in integrated modular avionics. In: IEEE. **Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th**. [S.l.], 2008. p. 1–B.

MESZAROS, G. **xUnit test patterns: Refactoring test code**. [S.l.]: Pearson Education, 2007.

PAN, J. Software testing. **Dependable Embedded Systems**, Citeseer, v. 5, p. 2006, 1999.

PRISAZNUK, P. J. Arinc 653 role in integrated modular avionics (ima). In: IEEE. **Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th**. [S.l.], 2008. p. 1–E.

QIAN, H.-m.; ZHENG, C. A embedded software testing process model. In: IEEE. **Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on**. [S.l.], 2009. p. 1–5.

RAJAN, A.; WHALEN, M. W.; HEIMDAHL, M. P. E. The effect of program and model structure on mc/dc test adequacy coverage. In: IEEE. **Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on**. [S.l.], 2008. p. 161–170.

RIVER, W. **VxWorks programmer's guide**. 1997.

RUSHBY, J. New challenges in certification for aircraft software. In: ACM. **Proceedings of the ninth ACM international conference on Embedded software**. [S.l.], 2011. p. 211–218.

SAGLIETTI, F. Testing for dependable embedded software. In: IEEE. **Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on**. [S.l.], 2010. p. 409–416.

STALLBAUM, H.; RZEPKA, M. Toward do-178b-compliant test models. In: IEEE. **Model-Driven Engineering, Verification, and Validation (MoDeVVa), 2010 Workshop on**. [S.l.], 2010. p. 25–30.

TOKAR, J. L. Space & time partitioning with arinc 653 and pragma profile. In: ACM. **ACM SIGAda Ada Letters**. [S.l.], 2003. v. 23, n. 4, p. 52–54.

VANDERLEEST, S. Arinc 653 hypervisor. In: IEEE. **Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th**. [S.l.], 2010. p. 5–E.

WARD, C. D. Software verification for a custom instrument using vectorcast and codesonar. 2011.

WATKINS, C.; WALTER, R. Comparing two industry game changers: Integrated modular avionics and the iphone. In: IEEE. **Digital Avionics Systems Conference, 2009. DASC'09. IEEE/AIAA 28th**. [S.l.], 2009. p. 1–A.

WENZEL, I. et al. Impact of dependable software development guidelines on timing analysis. In: IEEE. **Computer as a Tool, 2005. EUROCON 2005. The International Conference on**. [S.l.], 2005. v. 1, p. 575–578.

ZHANG, B.; SHEN, X. The effectiveness of real-time embedded software testing. In: IEEE. **Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference on**. [S.l.], 2011. p. 661–664.