UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA

Troca de contexto segura em sistemas operacionais embarcados utilizando técnicas de detecção e correção de erros

Rodrigo Maximiano Antunes de Almeida

Itajubá, Dezembro de 2013

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA

Rodrigo Maximiano Antunes de Almeida

Troca de contexto segura em sistemas operacionais embarcados utilizando técnicas de detecção e correção de erros

Tese apresentada ao Curso de Doutorado em Engenharia Elétrica da Universidade Federal de Itajubá como requisito parcial para a defesa de doutorado

Área de Concentração: Automação e Sistemas Elétricos Industriais

Orientador: Prof. Dr. Luís Henrique de Carvalho Ferreira
Coorientador: Prof. Dr. Carlos Henrique Valério de Moraes

Dezembro de 2013
Itajubá - MG

Ficha catalográfica elaborada pela Biblioteca Mauá –
Bibliotecária Margareth Ribeiro- CRB_6/1700

A447t

Almeida, Rodrigo Maximiano Antunes da
Troca de contexto segura em sistemas operacionais embar_ cados utilizando técnicas de detecção e correção de erros / Ro_ drigo Maximiano Antunes de Almeida. -- Itajubá, (MG) : [s.n.], 2013.
92 p. : il.

Orientador: Prof. Dr. Luis Henrique de Carvalho Ferreira
Coorientador: Prof. Dr. Carlos Henrique de Valério Moraes.
Tese (Doutorado) – Universidade Federal de Itajubá.

1. Sistemas embarcados. 2. Segurança. 3. Troca de contexto. 4. Correção de erros. I. Ferreira, Luís Henrique de Carvalho, orient. II. Moraes, Carlos Henrique de Valério, coorient. III. Universidade Federal de Itajubá. IV. Título.

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA

Rodrigo Maximiano Antunes de Almeida

Troca de contexto segura em sistemas operacionais embarcados utilizando técnicas de detecção e correção de erros

Tese aprovada por banca examinadora em 11 de Dezembro de 2013, conferindo ao autor o título de **Doutor em Ciências em Engenharia Elétrica**

Banca Examinadora:

Prof. Dr. Luís Henrique de Carvalho Ferreira - UNIFEI
Prof. Dr. Carlos Henrique Valério de Moraes - UNIFEI
Profa. Dra. Kalinka Regina Lucas Jaquie Castelo Branco - USP São Carlos
Eng. Dr. Levy Ely de Lacerda de Oliveira - PS Soluções
Prof. Dr. Edmilson Marmo Moreira - UNIFEI
Prof. Dr. Maurílio Pereira Coutinho - UNIFEI

Dezembro de 2013
Itajubá - MG

Dedico este trabalho primeiramente a Deus, por me ter concedido a vida e as demais graças que me permitiram chegar aqui; aos meus pais, Paulo e Carminha, pelo exímio exemplo de vida, sabedoria e retidão; àquela amiga, apoiadora, amada, Ana Almeida.

Agradecimentos

Aos meus pais, por todo o apoio e incentivo para meus estudos.

À Ana Paula Siqueira Silva de Almeida, pela compreensão e companheirismo.

Aos meus irmãos, Marcela e Daniel, simplesmente por fazerem parte da minha vida.

Aos professores Luis Henrique de Carvalho Ferreira e Carlos Henrique Valério de Moraes por todo tempo disponibilizado para realização deste trabalho e pelas valiosas orientações.

Ao amigo Enzo pela ajuda na 1ª versão (funcional) do kernel e a amiga Thatyana pelas revisões do documento.

Ao professor Armando pela ajuda nas análises estatísticas e de confiabilidade.

Aos meus alunos de TFG: Adriano, César, Lucas, Henrique e Rafael, pelo auxilio nos *drivers* e nos vários testes.

Ao amigo Alberto Fabiano, que mesmo nas breves conversas sobre segurança e embarcados sempre me trazia novas ideias (*in memoriam*).

Aos colegas e amigos do grupo de engenharia biomédica pelo apoio, infraestrutura, paciência nas duvidas interessantes e, principalmente, nas não tão interessantes.

A todos aqueles que, direta ou indiretamente, colaboraram para que este trabalho conseguisse atingir os objetivos propostos.

"Entre todas as verdadeiras buscas humanas,
a busca pela sabedoria é a mais perfeita,
a mais sublime, a mais útil
e a mais agradável"

São Tomás de Aquino

Resumo

A segurança e a confiabilidade em sistemas embarcados são áreas críticas e de recente desenvolvimento. Além das complicações inerentes à área de segurança, existem restrições quanto a capacidade de processamento e de armazenamento destes sistemas. Isto é agravado em sistemas de baixo custo. Neste trabalho, é apresentada uma técnica que, aplicada à troca de contexto em sistemas operacionais, aumentando a segurança destes. A técnica é baseada na detecção e correção de erros em sequência de valores binários. Para realização dos testes, foi desenvolvido um sistema operacional de tempo real e implementado numa placa de desenvolvimento. Observou-se que o consumo de processamento das técnicas de detecção de erro são inferiores às de correção, cerca de 2% para CRC e 8% para Hamming. Objetivando-se minimizar o tempo de processamento optou-se por uma abordagem mista entre correção e detecção. Esta abordagem reduz o consumo de processamento medida que os processos que exigem tempo real apresentem uma baixa taxa de execução, quando comparados com o período de troca de contexto. Por fim, fica comprovada a possibilidade de implementação desta técnica em qualquer sistema embarcado, inclusive em processadores de baixo custo.

Palavras-chaves: sistemas embarcados, segurança, troca de contexto, correção de erros.

Abstract

Security and reliability in embedded systems are critical areas with recent development. In addition to the inherent complications in the security area, there are restrictions on these systems processing power and storage capacity. This is even worse in low-cost systems. In this work, a technique to increase the system safety is presented. It is applied to the operating systems context switch. The technique is based on the detection and correction of errors in binary sequences. To perform the tests, a real-time operating system was developed and implemented on a development board. It was observed that the use of error detection and error correction techniques are lower than 2% for CRC and 8% to Hamming. Aiming to minimize the processing time a mixed approach between correction and detection was selected. This approach reduces the consumption of processing time as the processes that require real-time presents a low execution rate, when compared to the context switch rate. Finally, it is proved to be possible to implement this technique in any embedded system, including low cost processors.

Key-words: embedded systems, security, context switch, error correction.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de Siglas

ADC - *Analog to digital converter*

CCR - *Condition code register*

CI - Circuito integrado

CPU - *Central processing unit*

CRC - *Cyclic Redundat Check*

DAC - *Digital to analog converter*

DRAM - *Dynamic random access memory*

ECC - *Error-correcting code*

EEPROM - *Eletronic erasable programable read only memory*

EDF - *Earliest deadline first*

FIT - *Failures in time*

FPGA - *Field-programmable gate array*

MTBF - *Mean time between failures*

PC - *Program counter*

PID - Proporcional, Integrador, Derivativo

RR - *Round robin*

RT - *Real time*

SO - Sistema operacional

SP - *Stack pointer*

SPI - *Serial Peripheral Interface*

SRAM - *Static random access memory*

VHDL - *VHSIC Hardware Description Language*

Sumário

1 Introdução

A programação para sistemas embarcados exige uma série de cuidados especiais pois estes sistemas geralmente possuem restrições de memória e processamento (BARROS; CAVALCANTE, 2002). Outra complexidade encontrada é a variedade de arquiteturas e modelos de interfaces disponíveis.

Um dos modos de se reduzir esta complexidade é a utilização de um sistema operacional (SO) que insira uma camada de abstração entre o *hardware* e a aplicação. Esta redução, no entanto, vem acompanhada de uma sobrecarga, tanto no uso de memória, quanto no uso do processador. Esta sobrecarga pode ser proibitiva para alguns dispositivos, principalmente os de menor custo.

Quando se considera a questão de segurança nestes sistemas, a escassez de recursos computacionais, inerentes a projetos embarcados, apenas agrava a situação. Ravi et al. (2004) consideram que "a segurança em sistemas embarcados ainda está em sua infância, principalmente nas questões de desenvolvimento e pesquisa". O progresso observado nos últimos nove anos, no entanto, se concentrou em sistemas embarcados de maior capacidade, como *smartphones* e *tablets* (JEON et al., 2011). A área de segurança em sistemas embarcados segundo Kermani et al. (2013) ainda precisa de "novas abordagens para garantir a segurança de projetos de sistemas embarcados".

A utilização de *chips* dedicados para criptografia, por exemplo, é inviabilizada pela falta de acesso aos barramentos de dados e de endereço na maioria dos microcontroladores. A alternativa é a criação de linhas de microcontroladores com características de criptografia já embutidas no mesmo *chip*. Esta abordagem, no entanto, aumentaria o custo. Das cinco maiores fabricantes (MCGRATH, 2012), apenas quatro possuem alguma linha de microcontroladores que incluem algum tipo de *hardware* dedicado para criptografia ou processamento de mensagens. E mesmo estas empresas não disponibilizam estas soluções na maioria de seus produtos (RENESAS, 2013; FREESCALE, 2013; ATMEL, 2013; MICROCHIP, 2013; INFINEON, 2013).

A proposta deste trabalho consiste em realizar as trocas de contexto de modo seguro, possibilitando que as informações do bloco de contexto do processo sejam verificadas. A troca de contexto é um ponto crítico, pois qualquer falha pode levar o sistema a uma condição instável, geralmente reiniciando o microcontrolador.

1.1 Motivação

Segundo Wyglinski et al. (2013): "Dado o aumento da dependência da sociedade na computação embarcada, nos sistemas de sensoriamento, bem como nas aplicações que eles suportam, uma nova forma de vulnerabilidade é inserida nesta infraestrutura crítica e que apenas agora está começando a ser reconhecida como uma ameaça significante com possibilidade de graves consequências".

Entre os modos de falha de sistemas embarcados, dois vêm sendo objeto de estudo, tanto pela gravidade das consequências, quanto pela tendência de piora nas ocorrências.

O primeiro são as falhas em *bits* de memória. Dependendo da região atingida, mesmo a alteração de um único bit pode levar o sistema inteiro a uma condição de não funcionamento e sem possibilidade de recuperação sem intervenção humana. Grande parte destes problemas vem das interações de partículas atômicas e subatômicas com altas quantidades energéticas com regiões sensíveis (AUTRAN et al., 2010). A tendência é o aumento destes tipos de interação e, consequentemente, das falhas provocadas por elas (IBE et al., 2010). Os dois maiores fatores deste crescimento são a diminuição do tamanho mínimo de transistor (WANG; AGRAWAL, 2008; AUTRAN et al., 2010) e a redução dos níveis de tensão utilizados na alimentação dos circuitos (CHANDRA; AITKEN, 2008; BAUMANN; SMITH, 2000).

A segunda fonte são as vulnerabilidades que podem levar o sistema a apresentar falhas. O maior problema vem da atenção dada a estes dispositivos devido a sua disponibilidade e falta de ferramentas intrínsecas de segurança. Controladores lógicos programáveis (CLP's), centrais automotivas e equipamentos médicos são exemplos de sistemas críticos comercializados que tiveram falhas exploradas recentemente (KOSCHER et al., 2010; LANGNER, 2011; Zawoad; Hasan, 2012). Studnia et al. (2013) concluem em sua pesquisa que a falta de mecanismos de segurança nas atuais redes de automotores tem se tornado um grave problema.

Kermani et al. (2013) apresentam um conjunto de problemas de segurança que surgiram com o aumento da utilização de sistemas embarcados em áreas críticas. Entre as possíveis soluções propostas, os autores citam o desenvolvimento de ambientes de execução seguros, que levem em conta as restrições de *hardware* características deste tipo de dispositivos.

A geração destes ambientes seguros é, em geral, realizada com o suporte do *hardware* (CHAUDHARI; PARK; ABRAHAM, 2013). As abordagens desenvolvidas atualmente para sistemas embarcados ou fazem uso de coprocessadores dedicados (LEMAY; GUNTER, 2012) ou exigem que o equipamento consiga fazer uso de sistemas operacionais convencionais, como *tablets*, celulares ou PDA's (KAI; XIN; GUO, 2012; JEON et al., 2011; YIM et al., 2011).

Apesar de diversas abordagens já serem conhecidas e amplamente utilizadas em *desktops* e servidores, a transposição destas técnicas para sistemas embarcados de baixo custo

é ainda pouco estudada. Borchert, Schirmeier e Spinczyk (2013) apresentam, por exemplo, uma técnica que reduz os problemas de falhas devido a erros em *bits* de memória em 4 ordens de grandeza. No entanto, os autores concluem que a abordagem utilizada não é aplicável em grande parte dos sistemas embarcados devido ao alto consumo de memória e necessidade de ferramentas não disponíveis em sistemas de baixo custo.

1.2 Objetivo

Apresentar uma metodologia aplicável em sistemas embarcados de baixo custo que aumente a robustez do sistema reduzindo os problemas advindos de erros em *bits* de memória. Dentre os objetivos específicos temos:

- O sistema deve consumir o mínimo de recursos possível para ser aplicável em sistemas de baixo custo.
- A técnica não pode prejudicar a capacidade de execução de processos com requisitos de tempo real.
- A metodologia deve proteger o sistema contra vulnerabilidades que possam ser exploradas maliciosamente.

1.3 Organização do documento

Este documento é organizado em seis capítulos. O segundo capítulo apresenta os conceitos e ferramentas necessários para o desenvolvimento do projeto. O terceiro capítulo contém as etapas do desenvolvimento deste projeto, as dificuldades encontradas bem como as soluções propostas. O quarto capítulo contém a metodologia proposta neste trabalho. Os resultados obtidos foram compilados e apresentados no quinto capítulo. O sexto capítulo reúne as conclusões obtidas bem como a continuidade vislumbrada para este trabalho.

2 Revisão Bibliográfica

2.1 Sistemas Embarcados

Os sistemas embarcados são sistemas microprocessados projetados para um propósito ou aplicação específica, possuindo, em geral, poucos recursos de memória e processamento limitado. Na maioria dos casos, são sistemas projetados para aplicações que não necessitem de intervenção humana (HALLINAN, 2007).

Outra característica marcante é a especificidade de suas aplicações, sendo geralmente projetados para realizar apenas uma função não sendo possíveis de alterações pelo usuário final. O usuário pode alterar ou configurar a maneira como o sistema se comporta, porém não pode alterar a função que este realiza (MARWEDEL, 2006; STUDNIA et al., 2013; KERMANI et al., 2013).

Muitos sistemas embarcados possuem requisitos de tempo real em suas funções. Neste tipo de aplicação, a não execução ou não conclusão de uma tarefa no tempo determinado pode resultar em perda de dados, perda de qualidade ou até mesmo causar danos ao sistema ou ao usuário. Mesmo assim, a implementação de ferramentas que garantam que o sistema atinja os requisitos de tempo real, bem como técnicas para melhoria de confiabilidade e disponibilidade, não são amplamente utilizadas em sistemas embarcados (RAVI et al., 2004).

2.2 Sistemas Operacionais

O sistema operacional, SO, é um conjunto de códigos que funciona como uma camada de abstração do *hardware* provendo funcionalidades para as aplicações de alto nível (WULF et al., 1974). Este isolamento permite que a aplicação não sofra alteração quando há mudança no *hardware*. Esta é uma característica muito desejada em sistemas embarcados, onde existe uma pluralidade nos tipos de periféricos dificultando a reutilização de código.

De modo geral, os sistemas operacionais possuem três principais responsabilidades (SILBERSCHATZ; GALVIN; GAGNE, 2009):

- manusear a memória disponível e coordenar o acesso dos processos a ela;
- gerenciar e coordenar a execução dos processos através de algum critério;
- intermediar a comunicação entre os periféricos de *hardware* e os processos.

Estas responsabilidades se relacionam com os três recursos fundamentais de um sistema computacional: o processador, a memória e os dispositivos de entrada e saída. A Figura 1 ilustra estes recursos bem como o papel de interface que um sistema operacional deve realizar.

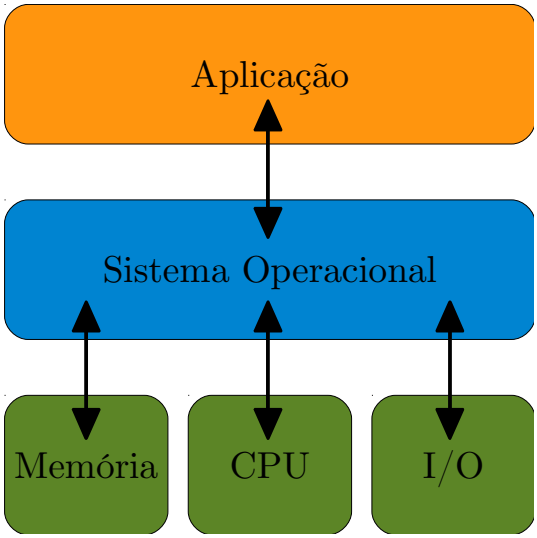


Figura 1: Interfaceamento realizado pelo sistema operacional

A ausência de um sistema operacional implica que toda a responsabilidade de organizar o andamento dos processos, os acessos ao *hardware* e o gerenciamento da memória é do programador. Este aumento de responsabilidade, a baixa capacidade de reutilização de código, e a consequente necessidade de recriar os códigos e rotinas, podem ser causadores de erros nos programas.

A capacidade de se reutilizar os programas é benéfica por dois pontos principais: diminui o tempo para entrega do projeto e permite que o programador utilize melhor o tempo, eliminando os erros ao invés de recriar os códigos.

A Figura 2 apresenta com mais detalhes os componentes de um sistema operacional. Nota-se que o núcleo de um SO é o *kernel*, e, do mesmo modo que o sistema operacional realiza a interface entre a aplicação e o *hardware*, o *kernel* faz a interface entre os códigos de acesso ao *hardware*, conhecidos como *drivers*, e as ferramentas disponibilizadas para que o programador crie as aplicações.

As aplicações, na presença de um sistema operacional, são implementadas como processos, que passam a ser gerenciados pelo *kernel*. A sequência com que os processos são executados fica a cargo de algoritmos conhecidos como escalonadores, que por sua vez dependem de um procedimento de troca de contexto para efetuar a mudança de qual processo será executado. Estes três conceitos: processos, escalonador e troca de contexto, serão aprofundados nos próximos itens.

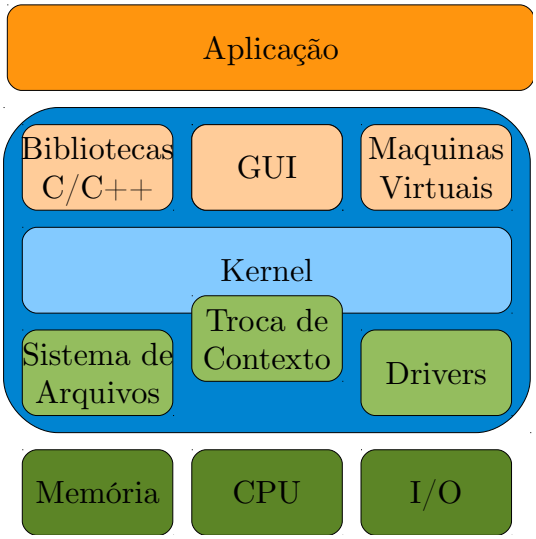


Figura 2: Relação entre troca de contexto e o *kernel*

2.2.1 Processo

Na utilização de um sistema operacional, as tarefas a serem executadas pelo processador são organizadas em programas. O programa é uma sequência de comandos ordenados com uma finalidade específica. No momento em que este programa estiver em execução no processador ele passa a ser definido como processo (STALLINGS, 2009).

Além do código a ser executado, os processos necessitam de posições de memórias extras para armazenar seus dados e variáveis, sejam eles persistentes ou não. São necessárias também regiões de memória, geralmente implementadas em estrutura de pilha, para armazenamento de informações referentes à sequência de execução do programa.

Para realizar o correto gerenciamento dos processos é necessário que o *kernel* possua informações sobre os mesmos, agrupadas de maneira consistente. As informações mínimas necessárias são:

- O código a ser executado;
- As variáveis internas do processo;
- Ponteiros para as informações anteriores, permitindo sua manipulação.

Em geral o código fica numa memória não volátil por questões de custo. Para microcontroladores, essas memórias são implementadas em tecnologia EEPROM, ou *flash*. Já as variáveis são armazenadas em memória volátil, pela maior velocidade de acesso e facilidade de escrita. As duas tecnologias mais utilizadas para este tipo de memória são a SRAM e a DRAM.

O armazenamento das informações de um processo, de modo automático e incremental, em estruturas do tipo pilha, permite que um mesmo programa possa ser executado mais de

uma vez sem que nenhuma das instâncias de execução tenham suas variáveis modificadas indevidamente. Na Figura 3 é apresentada a situação onde dois processos compartilham o mesmo código.

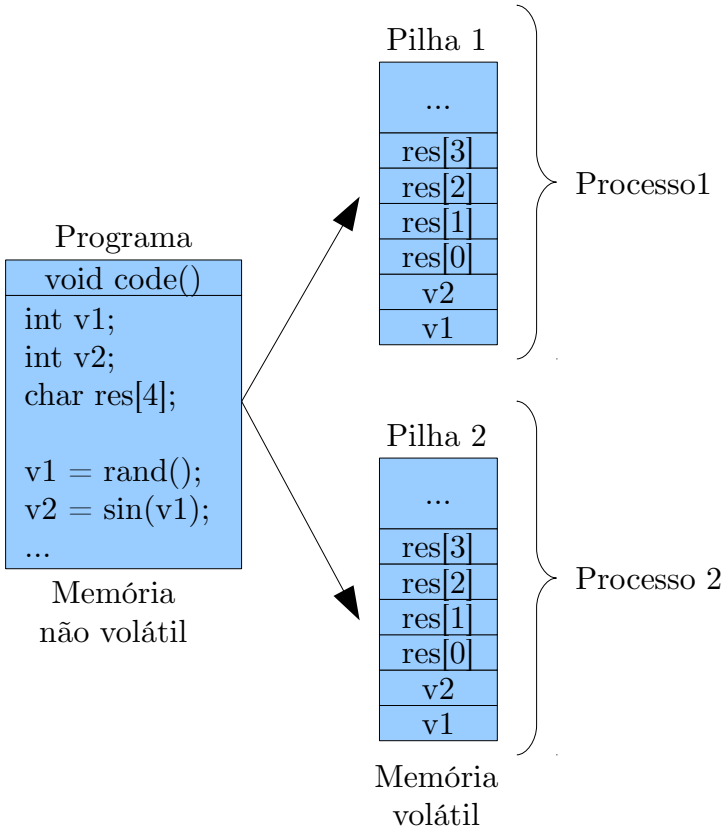


Figura 3: Geração de dois processos de um mesmo programa

Além da possibilidade de múltiplas execuções, utilizando-se da estrutura de pilha, as trocas de contexto são implementadas de maneira muito simples, principalmente se cada processo possuir sua própria região de pilha, bem definida e separada das demais.

2.2.2 Escalonadores

Uma das funções principais do *kernel* de um sistema operacional é o gerenciamento dos processos em execução (SILBERSCHATZ; GALVIN; GAGNE, 2009). Tal tarefa possui maior importância no contexto de sistemas embarcados, nos quais os processos costumam possuir restrições rígidas quanto ao atraso na execução (BARR, 1999).

Os algoritmos responsáveis por gerenciar e decidir qual dos processos será executado pelo processador são conhecidos como escalonadores. Existem diversas abordagens diferentes para realizar este gerenciamento. Em geral os algoritmos visam a equilibrar o atraso entre o início da execução e a quantidade de processos executados por unidade de tempo. Outros parâmetros importantes para a comparação dos escalonadores são o consumo extra de processamento (*CPU overhead*), a quantidade de processos executados

por unidade de tempo (*throughput*), o tempo entre a submissão de um processo e o fim da sua execução (*turnarround time*) e o tempo entre a submissão do processo e a sua primeira resposta válida (*response time*).

Na Tabela 1 são apresentados quatro algoritmos e as características destes.

Tabela 1: Algoritmos para escalonamento, vantagens e desvantagens (RAO et al., 2009)

Algoritmo de escalonamento	*CPU Overhead*	*Throughput*	*Turnaround time*	*Response time*
Escalonador baseado em prioridade	Baixo	Baixo	Alto	Alto
Escalonador round-robin (RR)	Baixo	Médio	Médio	Alto
Deadline mais crítico primeiro (EDF)	Médio	Baixo	Alto	Baixo
Escalonador de fila multinível	Alto	Alto	Médio	Médio

Nota-se pela Tabela 1 que não existe alternativa ótima, sendo necessário escolher então a que mais se ajusta ao sistema que será desenvolvido.

Para um sistema de tempo real, o tempo de resposta é um dos quesitos mais importantes, dado que a perda de um prazo pode impactar negativamente no funcionamento deste. Algumas aplicações podem falhar se estes requisitos não forem atendidos (RENAUX; BRAGA; KAWAMURA, 1999). Apesar disto, nem todos os processos em um sistema embarcado precisam deste nível de determinismo. Deste modo, um escalonador que permita ao programador escolher entre pelo menos dois níveis de prioridade (normal e tempo real) é uma boa alternativa (PEEK, 2013).

A maioria dos sistemas operacionais de tempo real apresentam como opção implementada o escalonador baseado em prioridades. Este fato se deve ao baixo consumo, mas principalmente a capacidade deste sistema de garantir que os processos mais críticos serão sempre executados. Na Tabela 2 é apresentada a tendência entre vários sistemas operacionais, comerciais e de código aberto.

Dois modelos de escalonamento foram escolhidos neste trabalho: EDF e RR. A proposta deste trabalho, de modificar o processo de troca de contexto incrementando sua robustez, pode ser aplicada a qualquer modelo de escalonador. A escolha dos dois modelos foi feita apenas para se realizar uma comparação de consumo de processamento. Deste modo, optou-se por um escalonador, EDF, que apresenta alto consumo e outro em que o consumo é notoriamente menor, RR. Assim o impacto no consumo de CPU da técnica proposta pode ser melhor avaliado em pelo menos duas situações diferentes. A opção por prioridade também foi implementada para a garantia de tempo real. Optou-se por essa solução por ser bastante utilizada, como também pode ser visto na Tabela 2.

Tabela 2: Sistemas operacionais de tempo real e escalonadores utilizados

Sistema Operacional	Escalonador	Preempção
BRTOS (DENARDIN; BARRIQUELLO, 2013)	Prioridade	Sim
eCos (MASSA, 2003)	Prioridade/filas multi-nível	Sim
FreeRTOS (ENGINEERS, 2013)	Prioridade	Sim
Micrium uC/OSII (MICRIUM, 2013)	Prioridade/round robin	Sim
Salvo (KALMAN, 2001)	Prioridade/por interrupções	Não
SDPOS (J.; V., 2013)	Prioridade/round robin	Sim
VxWorks (RIVER, 2013)	Prioridade	Sim

Os escalonadores podem ainda ser divididos entre cooperativos ou preemptivos.

Os cooperativos executam os processos de forma sequencial deixando que o processo tome o tempo que lhe for necessário para que complete sua execução. Deste modo, se um processo atrasar em sua execução todo o sistema é impactado.

Já os preemptivos são aqueles que conseguem pausar um processo que esteja em execução, salvar o estado atual do processo e do sistema, carregar o estado do sistema para um segundo processo e iniciar a execução deste último. Isto permite que, mesmo que um dos processos atrase, seja possível continuar a execução dos demais de acordo com os requisitos destes. O procedimento de substituir um processo em execução por outro é denominado troca de contexto.

2.2.3 Troca de Contexto

A troca de contexto é o procedimento pelo qual um processo A, que está em execução no processador, é pausado e um processo B toma seu lugar para iniciar, ou continuar, sua execução. Este procedimento acontece dentro de uma das rotinas de interrupção do sistema que indica ao *kernel* o momento de realizar essa troca. Esta interrupção pode ser oriunda de um relógio interno ou de algum evento externo. No segundo caso o sistema é denominado *tickless* (SIDDHA; PALLIPADI, 2007).

A troca de contexto esta intimamente ligada com as operações do *kernel*. Toda a gestão dos processos culmina na troca de contexto ditada pelo algoritmo utilizado no escalonador. Por ser necessário operar com os registros internos do processador, esta é uma das poucas rotinas de um sistema operacional que não pode ser escrita totalmente em linguagem de alto nível, além de ser extremamente dependente da arquitetura e até mesmo do modelo de processador utilizado.

O contexto de um processo é formado pelo conjunto dos dados internos ao processador: seus registradores, ponteiro de programa, ponteiro de pilha, códigos de condição de execução entre outros, além das variáveis e estado do próprio processo. É necessário

salvar esse contexto de modo que o processador possa recuperá-lo mais tarde e continuar a execução exatamente do ponto onde parou. É comum que todas estas informações, tanto do processador quanto do processo, se encontrem na pilha no momento da interrupção da troca de contexto. Isto acontece automaticamente por causa da estrutura utilizada para o tratamento de interrupção conforme ilustrado na Figura 4.

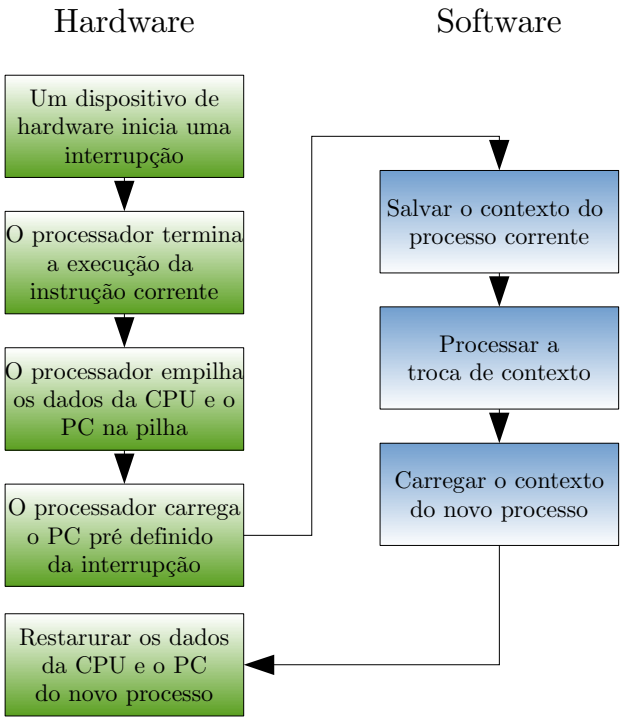


Figura 4: Processo de interrupção e salvamento de contexto(STALLINGS, 2010)

Se o processador utilizado é compatível com a estrutura apresentada na Figura 4, basta alterar a parte do *software* para que, ao invés de retornar para o processo que estava em execução antes da interrupção, o fluxo volte para um segundo processo arbitrário. O modo mais simples de fazê-lo é implementar uma pilha de memória para cada processo, bastando então apenas mudar o ponteiro de pilha para a pilha do segundo processo antes do *hardware* restaurar os dados.

Na Figura 5 são apresentas as quatro etapas básicas da mudança de contexto em um sistema: interrupção do processo em andamento, salvamento do contexto atual, troca dos dados, neste caso apenas a mudança no ponteiro de pilha (*stack pointer* ou SP), e, por fim, restauração do novo contexto.

As regiões demarcadas em azul representam a posição indicada pelo ponteiro de pilha (SP). Na primeira etapa, este valor é incrementado automaticamente devido ao *hardware* de interrupção. Este incremento visa a criar espaço enquanto os valores do contexto atual (em amarelo) são salvos na pilha.

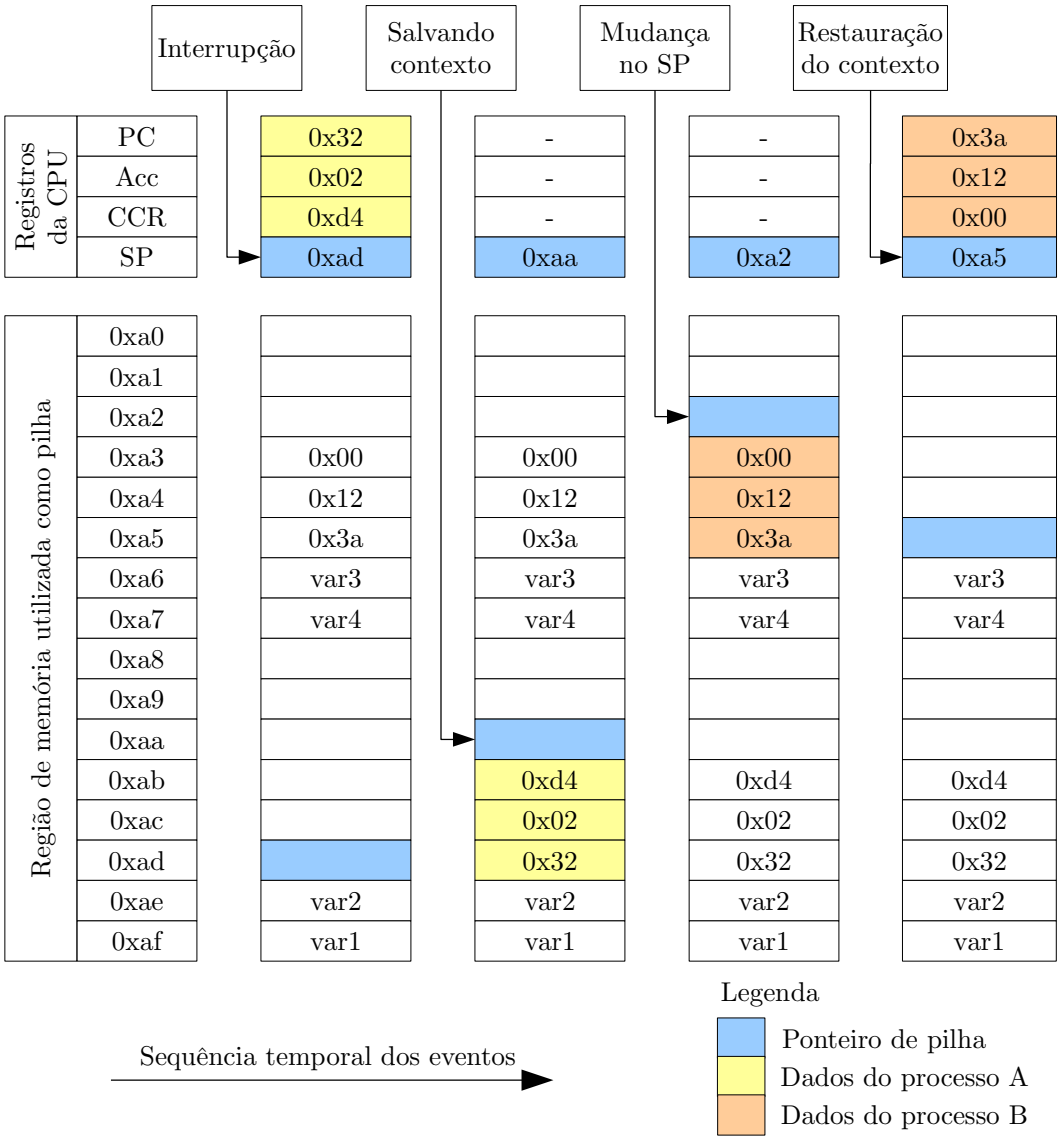


Figura 5: Troca de contexto e manipulação dos dados

A mudança no SP, da segunda para a terceira coluna da Figura 5, permite que o sistema carregue as informações referentes ao segundo processo (em laranja). Após a restauração do contexto, o sistema começa, ou continua a executar, dependendo do estado que o segundo processo se encontrava.

2.3 Segurança e modos de falha

A segurança de um sistema computacional pode ser avaliada sobre três aspectos:

- Disponibilidade: capacidade de um equipamento manter seu funcionamento independente de falhas ou erros.

- Confidencialidade: capacidade de um equipamento de evitar que as informações armazenadas/transmitidas sejam acessadas de modo não autorizado.
- Integridade: garantia que as informações armazenadas/transmitidas não serão alteradas indevidamente.

No contexto de sistemas embarcados, a garantia destes três aspectos envolve tanto o *hardware* quanto o *software*. Grande parte dos sistemas de baixo custo não costumam armazenar informações sigilosas, de modo que a necessidade de garantir a confidencialidade não é um ponto crítico. A questão da integridade se torna mais importante à medida que uma alteração indevida, seja ela intencional ou acidental, comprometa o funcionamento do sistema, o que é diretamente relacionado ao problema da disponibilidade. Este último é o mais crítico, visto que diversos sistemas embarcados, mesmo os de baixo custo, devem operar sem intervenção humana durante longos períodos de tempo.

Nas próximas seções são apresentados dois modos de falha capazes de impactar na disponibilidade de um sistema: os erros em memórias e os acessos não autorizados. O primeiro tem sua ocorrência de modo natural, já o segundo é, em geral, intencional. Ambos os modos de falha têm se agravado nos últimos anos. O primeiro, pela miniaturização e redução das tensões de alimentação dos dispositivos eletrônicos (CHANDRA; AITKEN, 2008; AUTRAN et al., 2010) e o segundo, pelo aumento de dispositivos conectados (KERMANI et al., 2013; KOOPMAN, 2004).

2.3.1 Erros em Memória

Erros na memória RAM podem ser causados, dentre outros fatores, por interferência eletromagnética, problemas físicos, bombardeamento de partículas atômicas ou ter seu valor corrompido no caminho entre a memória e o processador (SEMATECH, 2000).

Estes erros são relativamente comuns. Dos dados apresentados por Schroeder, Pinheiro e Weber (2009), cerca de um terço das máquinas estudadas sofreram algum tipo de falha. Do mesmo estudo, cerca de 8% das memórias DIMM apresentaram pelo menos um erro ao longo de um ano. O estudo aponta ainda que é fundamental o uso de uma camada de correção de erro em *software*, além das já existentes em *hardware*, para sistemas que precisem ser tolerantes à falha.

A questão do bombardeamento é mais crítica em equipamentos para aviação ou espaciais. Na altitude de 2km, as falhas são 14 vezes mais frequentes que na superfície, dada a redução da camada de ar que desvia ou absorve parte destas partículas(ZIEGLER et al., 1996). Já em altitudes superiores à 10km, a taxa de falha chega a ser 300 vezes maior que ao nível do mar (AUTRAN et al., 2012).

Mesmo protegidas pela atmosfera, existe uma outra fonte de partículas, neste caso partículas alfa, que pode afetar as memórias. Os traços de contaminantes radioativos no

processo de fabricação do CMOS e do encapsulamento do circuito integrado são a fonte mais comum destas partículas (ZIEGLER; PUCHNER, 2004; AUTRAN et al., 2010).

Os equipamentos embarcados, principalmente os equipamentos dependentes de baterias, com o objetivo de reduzir o consumo de energia, apresentam níveis de tensão mais baixos, necessitando de uma quantidade menor de energia para que um bit seja trocado. Na Figura 6 é apresentada a taxa de falhas de acordo com as tensões de alimentação envolvidas.

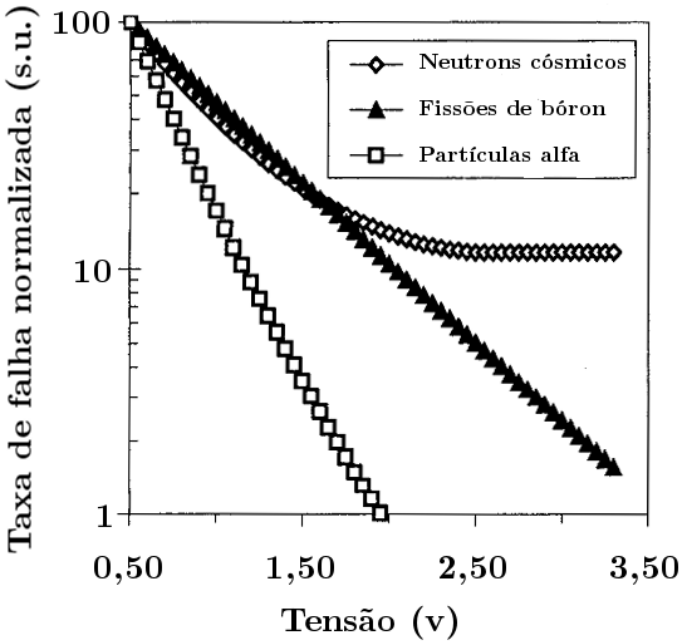


Figura 6: Comparativo das taxas de falhas normalizadas por tipo de elemento.

Fonte: Baumann e Smith (2000)

O bombardeamento de partículas, ou o decaimento de materiais radioativos, pode ser modelado por uma distribuição de Poisson (LI et al., 2006). Esta distribuição modela eventos sem memória, ou seja, a ocorrência de eventos futuros não está relacionada com a ocorrência ou não de eventos passados. A probabilidade de ocorrerem exatamente k eventos num período de tempo τ é dada pela Equação 2.1.

$$Pr(k) = \frac{e^{-\lambda\tau}(\lambda\tau)^k}{k!} \tag{2.1}$$

onde k é a quantidade de eventos, e é a base do logaritmo natural ($e = 2.7182...$), τ é o tempo de observação e λ é, tanto o valor médio de X, quanto a variância da medida.

De posse do valor de λ, que representa a quantidade média de falhas por tempo, é possível calcular o tempo médio entre falhas invertendo seu valor: $MTBF = \frac{1}{\lambda}$.

Na indústria de semicondutores é comum encontrar esta grandeza, MTBF, escrita em FIT, *Failures In Time*. Esta grandeza representa a quantidade de falhas esperadas em

um bilhão (10^9) de horas de operações, a Tabela 3 apresenta alguns valores encontrados na literatura.

Tabela 3: Configuração de *bits* para composição de mensagem

Estudo	FIT
NASA (WHITE; BERNSTEIN, 2008)	1826
DRAM (velocidade cheia) (LEUNG; HSU; JONES, 2000)	100-1.000
SRAM à 0.25 microns (LEUNG; HSU; JONES, 2000)	10.000-100.000

Fazendo uma extrapolação dos números para sistemas embarcados, devido às ordens de grandezas envolvidas, é de se esperar uma taxa de falhas menor para sistemas com apenas dezenas de kilobytes, ao invés de megabytes. No entanto, alguns destes sistemas possuem a expectativa de serem operados por anos ininterruptamente, de modo que a presença de uma falha, apesar de mais rara, pode ser crítica.

Os valores apresentados na Tabela 3, no entanto, devem apresentar piora nas próximas gerações de semicondutores, a medida que os processos de fabricação permitem que os transistores fiquem menores. Na Figura 7 são apresentadas as taxas de falha encontradas em memórias para diversos processos de fabricação com diferentes tamanhos de transistores. Pode-se notar que a piora é exponencial a medida que os transistores são miniaturizados.

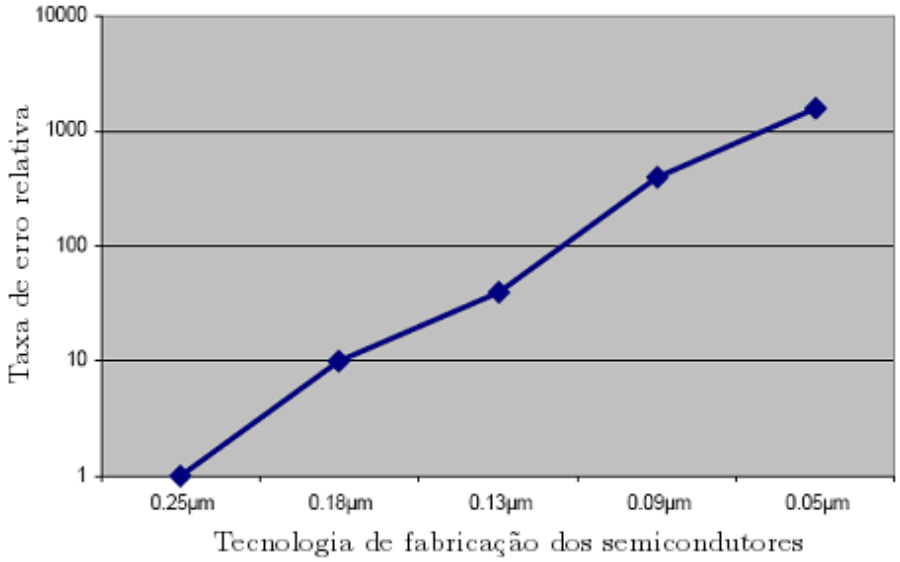


Figura 7: Taxa de erros por geração de tecnologia de fabricação de circuitos integrados.

Fonte Wang e Agrawal (2008)

Sabendo-se que as falhas acontecem seguindo uma distribuição de poisson (LI et al., 2006), a probabilidade de não ocorrer nenhuma falha ($Pr(0)$), num tempo τ é dada pela

equação 2.2.

$$Pr(0) = \frac{e^{-\lambda\tau}(\lambda\tau)^0}{0!} = e^{-\lambda\tau} \tag{2.2}$$

Segundo esta equação, é possível levantar a curva de probabilidade de falhas ao longo do tempo, conforme Figura 8. Foi utilizado um valor de $\lambda = 1,826E-006$ falhas/hora, que representa um dos valores mais críticos encontrados na literatura (WHITE; BERNSTEIN, 2008).

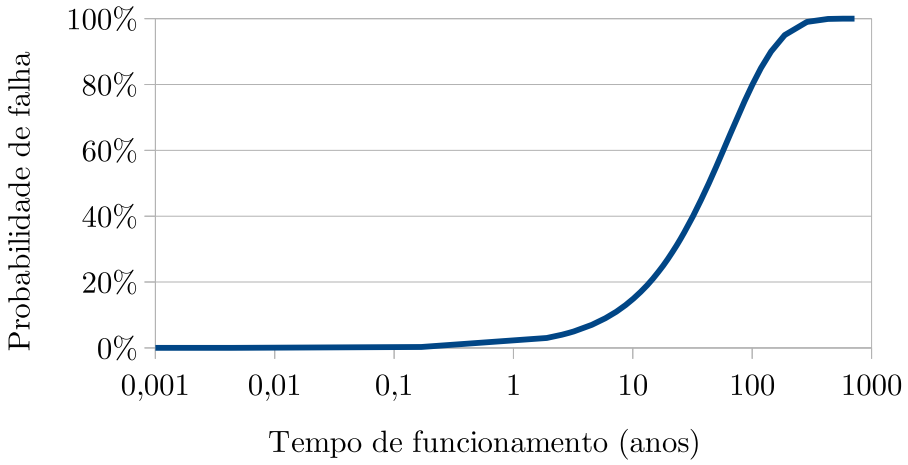


Figura 8: Probabilidade de falha em um sistema computacional ao longo do tempo

Um parâmetro de confiabilidade muito utilizado (BREYFOGLE, 2003) quando se espera que não haja defeitos num determinado lote de produtos é o intervalo de quatro sigma e meio, $4,5\sigma$, que representa $3,4$ falhas a cada milhão de unidades. Este intervalo é utilizado para projetos com a metodologia 6σ. Utilizando-se este valor, pode-se considerar que, na prática, não há falhas num processo do tipo 6σ (GEOFF, 2001).

Levando-se em conta novamente a distribuição de Poisson é possível calcular qual é o tempo decorrido para que a probabilidade de falha atinja o valor de 0.00034% ou 3.4 partes por milhão. Considerando-se um alto volume de unidades, este valor também pode ser interpretado como a quantidade esperada de produtos que possuem ou apresentam falhas. Deste modo, é possível calcular o tempo de funcionamento mínimo para que pelo menos 4 unidades, por milhão em funcionamento, apresentem problemas. Este valor, bem mais restritivo que o MTBF, pode ser ponto de comparação entre a qualidade relativa de diferentes produtos ou tecnologias.

Tabela 4: Probabilidade de falha em níveis sigma

Intervalo sigma	Quantidade de equipamentos q	Tempo em funcionamento antes que q equipamentos tenham 1 bit errado (Horas)
1σ	31,752%	208.996
2σ	4,551%	25.501
3σ	0,27%	1.480
4σ	0,007%	38
$4,5\sigma$	0,00034%	2

2.3.2 Exploração de vulnerabilidades

Todos os sistemas computacionais estão de algum modo susceptíveis a sofrerem ataques externos. Estes ataques nem sempre visam a invasão do sistema para roubo de informação ou uso não autorizado do mesmo. Alguns ataques podem ter como objetivo apenas desabilitar ou destruir os sistemas alvos.

Os sistemas embarcados, apesar de não serem um alvo tradicional, vem recebendo nos últimos anos cada vez mais atenção devido a sua disponibilidade e falta de ferramentas intrínsecas de segurança. A atenção a estes sistemas, principalmente os de baixo custo, é bastante recente (WYGLINSKI et al., 2013). Controladores lógicos programáveis (CLP's), centrais automotivas e equipamentos médicos são exemplos de sistemas críticos que já foram atacados com sucesso (KOSCHER et al., 2010; LANGNER, 2011; Zawoad; Hasan, 2012). Studnia et al. (2013) conclui, por exemplo, que a falta de mecanismos de segurança nas atuais redes de automotores tem se tornado um grave problema.

Excluindo-se as técnicas de engenharia social (IRANI et al., 2011), os ataques se concentram em procurar vulnerabilidades nos programas ou *hardwares* que possam ser exploradas. A vulnerabilidade mais explorada é o *buffer overflow*(NEWSOME; SONG, 2005; COWAN et al., 2000). Esta vulnerabilidade é criada por erros ou *bugs* inseridos pelo programador, em geral não propositais, ou erros existentes em bibliotecas utilizadas no projeto.

A técnica de ataque via *buffer overflow* tenta utilizar uma falha na recepção e escrita de um conjunto de dados num *buffer* de memória de tal modo que outras variáveis tenham seu valor alterado. Em processadores de arquitetura Von Neumman, onde o espaço de endereçamento de dados e programas é único, existe a possibilidade de se alterar até mesmo o código do programa armazenado. O programa apresentado no Código 1 possui a possibilidade de explorar um *bug* de *buffer overflow*(WIKIPEDIA, 2013).

Em geral, os compiladores alocam as variáveis locais na pilha do processo corrente, facilitando o acesso e permitindo que este espaço seja posteriormente desocupado facilmente e liberado para outras aplicações.

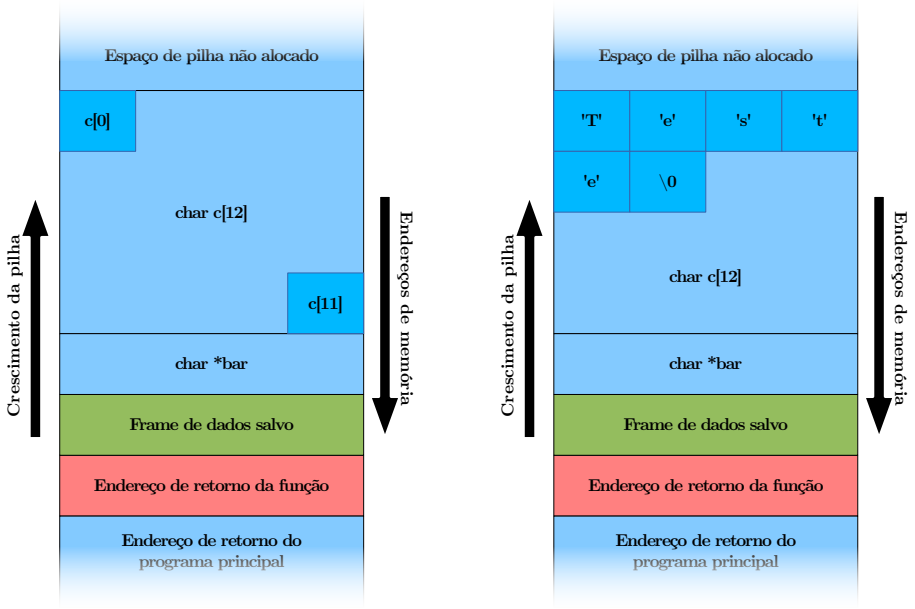
Por causa do modo de operação da pilha, as variáveis são alocadas próximas ao endereço de retorno da função chamada, como pode ser observado na Figura 9a.

Código 1: Exemplo de função com vulnerabilidade de *buffer overflow*

```
1 #include <string.h>
2 void buffOver (char *bar)
3 {
4    char  c[12];
5    strcpy(c, bar);  //sem checagem de tamanho
6 }

8 int main (int argc, char **argv)
9 {
10   buffOver(argv[1]);
11 }
```

Caso seja fornecida uma entrada com menos de 12 caracteres, incluindo o terminador, estes serão armazenados corretamente na pilha sem sobreescrever nenhuma outra variável. A figura 9b apresenta o mapa da memória para a entrada "Teste".



(a) Mapa de memória (b) Entrada de dados normal

Figura 9: Posicionamento das variáveis e informações na pilha da função *buffOver()*

Se o parâmetro passado para a função *buffOver()* tiver mais de 12 caracteres, as outras variáveis serão reescritas. Tendo conhecimento sobre o mapa de memória é possível entrar com uma *string* cujo conteúdo sobreescreva inclusive o endereço de retorno da função. Isto permite que uma pessoa entre com um programa arbitrário, representado pelas letras 'A' da Figura 10, e com um endereço de retorno adulterado, fazendo que seu programa seja executado.

Talvez esta seja uma das alterações mais críticas: o registro de retorno de função. A cada chamada de função, o endereço de retorno da função anterior é armazenado na pilha.

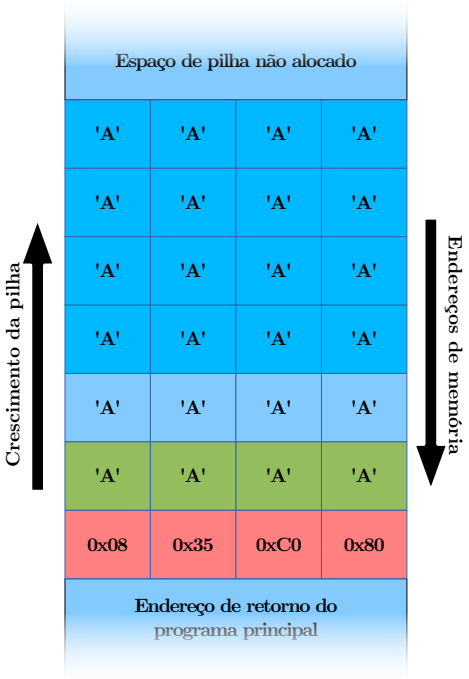


Figura 10: Exploração de falha de *buffer overflow* para reescrita de endereço de retorno

Deste modo é possível inserir um código arbitrário, alterar o endereço de retorno e executar o código inserido sem permissão explícita do sistema. Em geral, os sistemas operacionais conseguem detectar este erro e geram uma falha de segmentação, *segmentation fault*. No entanto, em sistemas operacionais para dispositivos de baixo custo, estas técnicas podem não estar presentes.

A técnica de *format string* se utiliza de falhas na formatação adequada de dados inseridos pelo usuário com o intuito de sobrescrever regiões da memória, de modo similar à vulnerabilidade de *buffer overflow* (NEWSHAM, 2001). Um dos modos mais comuns é a utilização do *token* "%n", que indica ao *printf*, e outras funções similares, que imprima o texto para um endereço armazenado na pilha.

Para sistemas embarcados até mesmo a *libC* pode ser crítica. Esta biblioteca, apesar de largamente utilizada, possui sua implementação dependente do fabricante do compilador, que é diferente para cada tipo de arquitetura de processador. Como o mercado de compiladores para processadores embarcados é bastante pulverizado, é difícil garantir a qualidade destas bibliotecas para todas as arquiteturas e modelos existentes (GANSSLE, 1990).

Uma terceira categoria de ataques, bastante recente, se concentra em utilizar pequenos trechos de códigos já escritos na memória do sistema e que sejam seguidos pela operação *RETURN*. Deste modo é possível construir programas genéricos sem realizar a inserção de códigos maliciosos, bastando apenas encadear os trechos de códigos necessários. Esta técnica é denominada por *Return-Oriented Programming*, ROP, (ROEMER et al., 2012). Langner (2011) apresenta um compilador que gera automaticamente uma pilha que per-

mite executar um programa arbitrário utilizando-se apenas de trechos das funções da biblioteca libC. Uma lista com vários conjuntos de códigos e o processo para executá-los tanto numa arquitetura x86 quanto SPARC é apresentada por Roemer et al. (2012). Esta técnica vem se mostrando perigosa, pois ultrapassa todas as técnicas, desenvolvidas até o momento, que procuram impedir a execução de código malicioso, visto que o "código" a ser executado é composto de trechos de outros códigos que o sistema já conhecia sendo, pela visão do sistema operacional, um código confiável.

2.4 Segurança em aplicações computacionais

Diversas técnicas foram desenvolvidas de modo a tornar as aplicações computacionais mais seguras, seja por redundância, por limitação de recursos ou por verificação de informações.

Grande parte destas técnicas foram inicialmente desenvolvidas em *software*, pela facilidade na prototipagem e teste do sistema. Algumas se tornaram tão populares que os desenvolvedores de *hardware* passaram a integrar em seus processadores rotinas ou funcionalidades que facilitem ou até mesmo implementem completamente algumas destas técnicas (BRAUN et al., 1996).

2.4.1 Redundância em memórias

Um procedimento muito comum para evitar as consequências de erros nas memórias, além de evitar alterações não autorizadas, é armazenar uma cópia de todas as variáveis em uma memória sombra (*shadow memory*). Uma das cópias não pode ser escrita por meios externos, devendo necessariamente passar pelo canal de gravação/leitura, protegendo assim a informação.

A plataforma Samurai (PATTABIRAMAN; GROVER; ZORN, 2008) é um alocador dinâmico de memória que faz uso da replicação de trechos da memória para evitar os erros. Os programas têm que ser manualmente modificados para utilizar a API de modo eficiente. No entanto, esta plataforma só suporta variáveis dinâmicas, que são evitadas em sistemas embarcados de alta criticidade. Esta restrição é inclusive explicitada na regra 20.4 da normativa MISRA-C (MISRA et al., 2004) onde a "memória dinâmica em *heap* não deve ser utilizada" e na norma IEC 61508-3 (IEC et al., 1998) no anexo B que pede a não utilização de "objetos dinâmicos" ou "variáveis dinâmicas".

A técnica de replicação é amplamente utilizada em arranjos de discos rígidos, sendo padronizada sob a arquitetura RAID (*redundant array of independent disks*, originalmente *redundant array of inexpensive disks*) (PATTERSON; GIBSON; KATZ, 1988). No nível 1, também conhecido como espelho, os dados são copiados N vezes, onde N é o número de discos disponíveis (CHEN et al., 1994). O problema com este procedimento é o custo. Para discos rígidos, o custo por memória é baixo quando comparado com do tipo SRAM

ou DRAM. Deste modo, o custo de uma alternativa similar a esta, para a memória RAM, pode não ser viável.

Sobre a questão do custo, uma das soluções é armazenar, ao invés de uma cópia total dos dados, apenas um valor que consiga representar aquela mensagem. Dependendo do modo de geração deste valor, os erros, além de detectáveis, podem ser corrigidos. Esta é a técnica utilizada nas topologias RAID de 2 à 6, onde os dados são divididos em N partes e um código de correção é gerado e armazenado. Deste modo, são necessários $N + 1$ discos, com exceção dos modos 2 e 6, onde, no primeiro, é possível utilizar mais de um disco de paridade e, no segundo, é obrigatório a utilização de pelo menos 2 discos. Se algum dos discos for perdido é ainda possível recuperar a informação. Na Figura 11 são apresentas as diferentes topologias disponíveis.

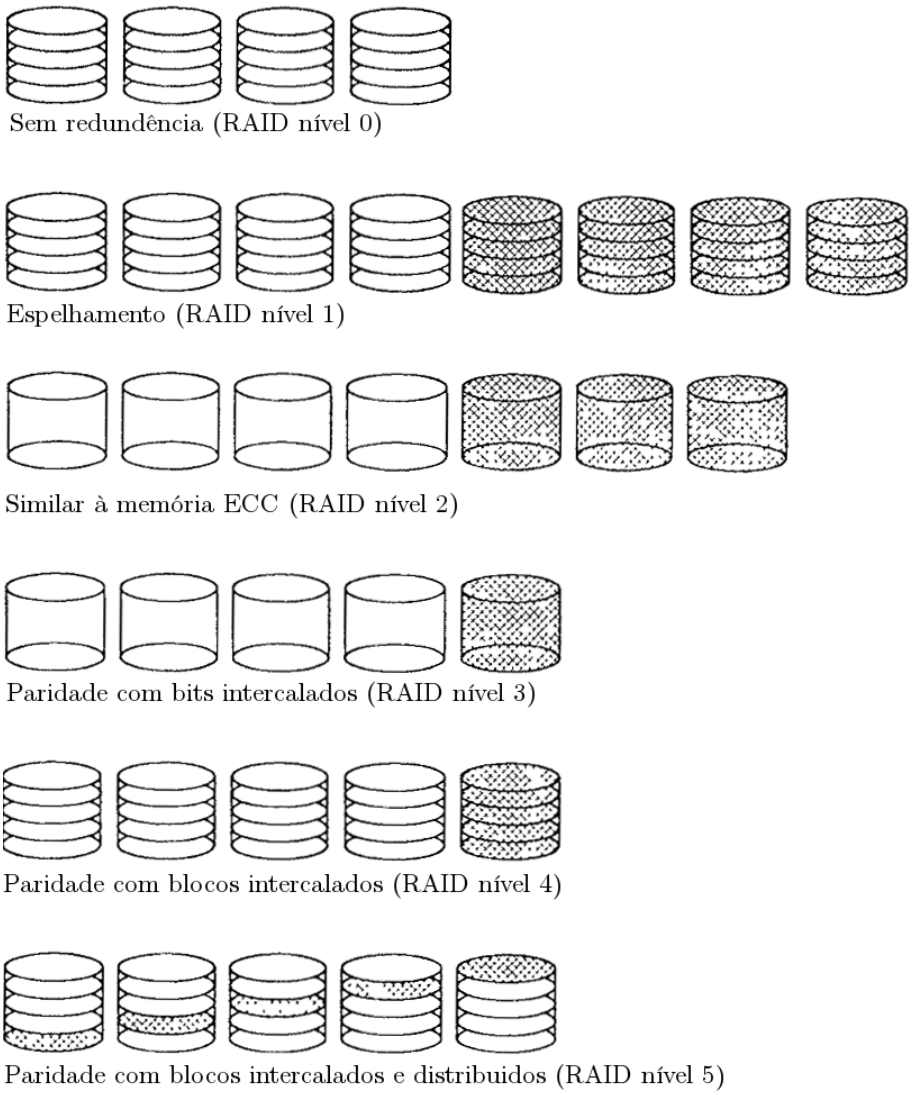


Figura 11: Modelos de topologia RAID disponíveis.

Fonte: Chen et al. (1994)

A técnica utilizada pelo RAID 2 consiste em armazenar um código capaz de corrigir

um eventual erro. Esta abordagem é aplicada em memórias RAM do tipo ECC (*error-correcting code*). Nestas memórias um código *Hamming* é inserido num espaço extra, em geral na razão de 8 *bits* de correção para cada 64 *bits* de dados (LI et al., 2010), gerando um acréscimo no custo na ordem de 1/8, ou 12, 5%.

A implementação em *software* de algoritmos de proteção de memória, detecção ou correção de erros, em geral, apresentam uma sobrecarga de processamento demasiadamente grande para boa parte dos sistemas embarcados de baixo custo. Por esse motivo algumas alternativas (WANG et al., 2009; BORCHERT; SCHIRMEIER; SPINCZYK, 2013) optam por elencar apenas as regiões críticas do sistema para implementar essa proteção.

Wang et al. (2009) propõem a seleção das rotinas do *kernel* e o isolamento destas em páginas dedicadas de memória que serão posteriormente marcadas como apenas leitura, protegendo o sistema de invasões com a intenção de sobrescrever esses códigos.

Borchert, Schirmeier e Spinczyk (2013) apresentam um sistema de proteção aos erros de memória armazenando informações extras para a proteção dos dados originais. Para tal, o artigo apresenta 5 opções para a geração do valor de checagem, sendo 1 (CRC) capaz de detectar e 4 (tripla redundância, CRC + cópia do valor, somatório + cópia do valor, código de Hamming) capazes de corrigir erros.

Segundo o artigo, as áreas mais críticas do sistema estudado (eCos) são as áreas que armazenam os dados das *threads* e da pilha (*stack*) do sistema. Estas áreas podem representar, respectivamente, até 39,4% e 22,4% das falhas observadas. Os autores apresentam uma redução de 12,8% para menos de 0,01% de falhas nos testes realizados. A abordagem apresentada por eles é similar a proposição deste trabalho, de adicionar um código de correção, ou detecção de erros, para proteger regiões críticas da memória.

A solução, no entanto, não é adequada para grande parte dos sistemas embarcados por causa da grande quantidade de código inserida na aplicação. Além disso, esta abordagem só pode ser inserida em sistemas desenvolvidos com linguagem C++, por se utilizar do paradigma de orientação à aspecto, não disponível em C.

2.4.2 Limitação na execução de código

Uma alternativa para evitar a execução de código malicioso, indevidamente injetado no sistema, é proteger as regiões de memória que contém código para que não possam ser escritas. Já as regiões de dados, que podem ser modificadas, são marcadas como não executáveis.

Seshadri et al. (2007) apresentam um hipervisor que consegue prover uma camada de segurança na execução de código em *kernel mode* no *kernel* do Linux com quase nenhuma alteração no código fonte. O autor reporta uma mudança de apenas 93 linhas (12 adições e 81 remoções) de um total de aproximadamente 4.3 milhões.

Para garantir que o sistema de proteção funcione corretamente é necessário que tanto o *hardware* quanto o *software*, neste caso o sistema operacional, tenham capacidade para tal.

A primeira necessidade é que o processador tenha capacidade de marcar regiões de memória como não disponíveis para execução. As duas maiores fabricantes de processadores para *desktops* têm esta tecnologia implementada em sua linha de processadores. A Intel (2013) comercializa esta tecnologia sob o nome XD bit, ou *execute disable*. Já AMD (2013), fazendo a conexão com o benefício direto desta tecnologia, utiliza o nome *enhanced virus protection*. No mercado de embarcados, apenas processadores de maior capacidade possuem este dispositivo, a ARM (2013) com o *execute never*, ou *XN bit* e a MIPS (2013) com o *execute inhibit*.

É possível realizar a emulação da proteção contra execução utilizando apenas camadas de *software*, sem suporte de *hardware*. Pelo menos duas alternativas funcionais existem atualmente: o PaX (PAXTEAM, 2012) e o ExecShield (VEN, 2004). Em contrapartida com a não-utilização de um *hardware* dedicado, estas alternativas aumentam o consumo do processador, podendo chegar a uma piora de até 73.2% na execução de um processo (VENDA, 2005).

A segunda necessidade para o correto funcionamento é o suporte por parte do sistema operacional. A grande maioria dos sistemas operacionais atuais, principalmente os voltados para *desktop's* possuem esse suporte:

- Android: a partir da versão 2.3
- FreeBSD: em versão de teste em 2003 e em versão *release* a partir da 5.3
- Linux: a partir da versão 2.6.8 em 2004
- OS X: a partir da versão 10.4.4
- Windows: a partir do XP Service Pack 2 e Windows 2003

2.4.3 Modificação do programa em tempo de execução

Parte dos ataques necessita de informações sobre a disposição dos programas e das variáveis na memória do sistema alvo, principalmente ataques que explorem falhas do tipo *buffer overflow*. Uma das alternativas para aumentar a segurança é a modificação do mapa de memória em tempo de execução. Deste modo, o invasor não possui tempo hábil de conhecer o posicionamento das variáveis para manipulá-las (XU; KALBARCZYK; IYER, 2003).

A técnica se concentra em duas alterações básicas: modificar o posicionamento das funções na memória, no processo de linkagem, e inserir, aleatoriamente, variáveis desnecessárias entre as existentes no código. Após a geração do novo binário o programa

antigo é substituído pelo novo durante a troca de contexto. Este processo é dinâmico e é executado com o sistema funcionando (XU; CHAPIN, 2006).

Apesar da complexidade inerente ao sistema, Giuffrida, Kuijsten e Tanenbaum (2012) apresentam uma sobrecarga relativamente baixa, de apenas 10% para ciclos de randomização de 4 em 4 segundos.

O problema com esta abordagem é a implementação em sistemas embarcados. A grande maioria dos sistemas de baixo custo não possuem capacidade de executar nativamente um compilador, nem memória suficiente para armazenar o código fonte. Além disso, alguns sistemas não conseguem reescrever sua memória inteira, sendo necessário uma ferramenta externa de gravação.

2.5 Algoritmos de detecção e correção de erros

Todos os algoritmos de detecção e correção inserem informações extras na mensagem ou no arquivo para que os erros sejam identificados. O modo mais simples é a duplicação da mensagem. O erro é detectado quando as mensagens não são iguais. O problema com essa abordagem é que ela insere um *overhead* de 100% na mensagem original.

As alternativas se concentram em gerar um código, geralmente de tamanho fixo, que represente a mensagem de modo que qualquer alteração seja facilmente percebida. Dentre estas alternativas os algoritmos de CRC (*cyclic redundancy check*) são os mais difundidos, sendo utilizados em diversos tipos de comunicação de dados a exemplo dos protocolos CAN (STANDARD, 1993), Bluetooth (GROUP et al., 2009) e até mesmo em comunicações sobre IP (BRAUN; WALDVOGEL, 2001). Baseado na alta utilização, algumas empresas desenvolveram CI's dedicados que implementam o algoritmo em *hardware*. Também é possível encontrar microcontroladores que possuem periféricos dedicados para este processamento (BOMMENA, 2008), soluções já codificadas em VHDL para implementação em FPGA's (BRAUN et al., 1996) e até mesmo circuitos integrados dedicados para a geração do código de verificação (CORPORATION, 2011) ou para a transmissão de dados com o código de verificação já incorporado (SEMICONDUCTORS, 1995).

Alguns algoritmos permitem, além de detectar o erro, que este também possa ser corrigido. Isto permite gerar redundância no sistema ao custo de um *overhead*, consumindo mais memória de armazenamento ou banda na transmissão. Entre os códigos comumente utilizados para correção de erros em dispositivos armazenadores de memória estão os algoritmos de Hamming, Hsiao, Reddy e Bose-Chaudhuri-Hocquenghem (STMICROELECTRONICS, 2004). A escolha depende da quantidade de *bits* que o desenvolvedor deseja que sejam detectados ou corrigidos. Do mesmo modo que o algoritmo de CRC, os algoritmos de correção podem ser implementados em *software* ou em *hardware*.

A utilização de memórias e microcontroladores com os algoritmos de correção implementados em *hardware* acarreta, segundo Cataldo (2001), numa memória com uma área

de silício em geral 20% maior, impactando diretamente em seu custo. Além disso o autor observou uma redução na velocidade entre 3% e 4%, chegando em 33% em memórias de maior desempenho, sendo assim mais rápido que a implementação em *software*.

Apesar do maior consumo das implementações em *software*, em contraste com as feitas em *hardware*, é possível implementar os algoritmos de correção em qualquer tipo de sistema, com ou sem acesso ao barramento de memória, independente da arquitetura do processador.

Nos próximos tópicos serão apresentadas duas técnicas, a primeira, CRC, para detecção e a segunda, Hamming, para correção de erros.

2.5.1 CRC

Os algoritmos de CRC (*Cyclic Redundant Check*) são otimizados para a detecção de erros. Segundo Ray e Koopman (2006), em sistemas com alta exigência na detecção de erros, o algoritmo de CRC pode ser "a única alternativa prática comprovada pela utilização em campo".

Estes algoritmos são baseados na divisão inteira de números binários sobre um campo finito de ordem 2. Para efeitos matemáticos de isolação dos coeficientes (WILLIAMS, 1993), é comum representar os números como um polinômio de x. O termo x^n existe se a posição n da palavra binária for de valor um. Caso o valor seja zero o termo é omitido da representação, conforme exemplo abaixo.

$$P(x) = x^8 + x^6 + x^0 => 1\ 0100\ 0001_2 \tag{2.3}$$

Para o cálculo do valor de CRC de uma mensagem representada pelo polinômio $M(x)$ de tamanho t_m, dado um polinômio gerador $G(X)$ de tamanho t_g, deve-se:

- adicionar t_g zeros ao final da mensagem, o que é feito multiplicando o polinômio $M(x)$ por x^{t_g}
- realizar a divisão deste polinômio $(M(x) \cdot x^{t_g})$ por $G(x)$
- armazenar o resto da divisão, o polinômio $R(x)$, em formato binário, que é o valor de CRC

Como o resultado do procedimento vem do cálculo do resto de uma divisão, o tamanho do CRC é fixo e determinado pelo termo de maior expoente do polinômio $G(x)$ (KUROSE; ROSS, 2012).

Após a transmissão dos dados, ou na leitura de um valor armazenado a priori, o procedimento de validação dos dados $M'(x)$ é o mesmo para a geração do valor de CRC. Após realizar o cálculo da divisão de $(M'(x) \cdot x^{t_g})$ por $G(x)$, se o valor $R'(x)$, recém calculado,

for igual ao valor $R(x)$, armazenado anteriormente, a mensagem foi lida/transmitida corretamente. É possível também realizar a divisão de $M'(x) \cdot x^{t_g} + R(x)$ por $G(x)$ esperando que o resto $R'(x)$ seja igual a 0.

A escolha do polinômio divisor G(x) é pautada basicamente por dois requerimentos, gasto computacional e quantidade de erros identificáveis dado um determinado tamanho de mensagem. A capacidade de identificar uma dada quantidade de erros binários numa mensagem é denominada distância de Hamming (HD, *hamming distance*). Aplicações críticas geralmente requerem altas distâncias de Hamming, HD = 6 para todos os tamanhos de mensagens (RAY; KOOPMAN, 2006).

Koopman e Chakravarty (2004) apresentam, em seu trabalho, uma tabela com os melhores polinômios com tamanho variando entre 3 e 16 *bits*. Deve-se tomar cuidado na seleção, pois alguns polinômios amplamente utilizados na literatura não apresentam um bom resultado, podendo ser melhorados sem impactar no tempo de cálculo. A Figura 12 apresenta os resultados encontrados.

Maior distância de Hamming para o polinômio	Tamanho do CRC (bits)													
	3	4	5	6	7	8	9	10	11	12	13	14	15	16
HD=2	2048+ 0x5	2048+ 0x9	2048+ 0x12	2048+ 0x21	2048+ 0x48	2048+ 0xA6	2048+ 0x167	2048+ 0x327	2048+ 0x64D	–	–	–	–	–
HD=3		11 0x9	26 0x12	57 0x21	120 0x48	247 0xA6	502 0x167	1013 0x327	2036 0x64D	2048 0xB75	–	–	–	–
HD=4			10 0x15	25 0x2C	56 0x5B	119 0x97	246 0x14B	501 0x319	1012 0x583	2035 0xC07	2048 0x102A	2048 0x21E8	2048 0x4976	2048 0xBAAD
HD=5						9 0x9C	13 0x185	21 0x2B9	25 0x5D7	53 0x8F8	none	113 0x212D	136 0x6A8D	241 0xAC9A
HD=6							8 0x13C	12 0x28E	22 0x532	27 0xB41	52 0x1909	57 0x372B	114 0x573A	135 0xC86C
HD=7									12 0x571	none	12 0x12A5	13 0x28A9	16 0x5BD5	19 0x968B
HD=8										11 0xA4F	11 0x10B7	11 0x2371	12 0x630B	15 0x8FDB

Figura 12: Melhores polinômios de CRC.

Fonte: Koopman e Chakravarty (2004)

Um último ponto a ser considerado na escolha é a disponibilidade de *hardwares* dedicados para os cálculos. Alguns polinômios são padronizados para comunicações e possuem seus algoritmos implementados em *hardware*, principalmente em periféricos de comunicação de alta velocidade, reduzindo o tempo de processamento além de liberar o processador para outras tarefas.

2.5.2 Hamming

A proposta de Hamming para um algoritmo que conseguisse realizar a detecção e correção de um erro numa mensagem binária de tamanho m foi adicionar uma quantidade k de *bits* de paridade num código de verificação cv. É necessário que a quantidade de símbolos

endereçáveis por cv, 2^k, seja maior que a quantidade de *bits* da mensagem, ou $m+k+1$. O acréscimo da unidade é devido à utilização do simbolo zero para indicativo de ausência de erros. Isto permite que cada possível mensagem possa ser endereçada de modo único pelo espaço de valores formado pelos *bits* de correção, permitindo sua identificação e correção no caso de um erro (HAMMING, 1950).

A Tabela 5 apresenta valores de sobrecarga para diferentes tamanhos de *bits* de paridade indicando também o tamanho máximo de dados corrigíveis para a quantidade de *bits* usado.

Tabela 5: Configuração de *bits* para composição de mensagem.

Bits de Paridade	*Bits* de Dados	Total de *bits*	Taxa de uso (1 − sobrecarga)
2	1	3	$1/3 \approx 0.333$
3	4	7	$4/7 \approx 0.571$
4	11	15	$11/15 \approx 0.733$
5	26	31	$26/31 \approx 0.839$
6	57	63	$57/63 \approx 0.905$
7	120	127	$120/127 \approx 0.945$
k	$(2^k - 1) - k$	$2^k - 1$	$1 - k/(2^k - 1)$

Adaptada de Hamming (1950)

Para evitar que seja necessária uma tabela de conversão do código cv para a posição do erro na mensagem, é comum reorganizar a mensagem de modo que os *bits* de paridade cv_i sejam intercalados nas posições n, onde $n = 2^i$. A Tabela 6 apresenta o exemplo da mensagem $m = 1010$, de tamanho 4, com um cv de tamanho 3, totalizando 7 *bits*, conforme é apresentado na Tabela 5.

Tabela 6: Exemplo de uma mensagem de 4 *bits* e posicionamento destes na mensagem com os valores de correção

Posição	7	6	5	4	3	2	1	0
Uso	m_3	m_2	m_1	cv_2	m_0	cv_1	cv_0	Não usado
Exemplo	1	0	1	cv_2	0	cv_1	cv_0	-

Para o cálculo dos *bits* de paridade cv_i é necessário explicitar a posição de cada bit em notação binária. O bit da posição j da mensagem m será utilizado se o valor de j em binário b_j apresentar um valor 1 (um) na posição $b_j i$. A Tabela 7 apresenta, em azul, quais são as posições utilizadas no cálculo dos *bits* de paridade de cv.

Da Tabela 7 pode-se concluir que $cv_0 = m_0 \oplus m_1 \oplus m_3 = 0$, $cv_1 = m_0 \oplus m_2 \oplus m_3 = 1$ e $cv_2 = m_1 \oplus m_2 \oplus m_3 = 0$. Os cálculos levam em conta apenas os *bits* de dados. Mesmo que um *bit* de paridade esteja numa posição utilizadas para calcular outro *bit* de paridade,

Tabela 7: Utilização da posição do bit de mensagem no cálculo dos *bits* de correção cv_i

Posição	7	6	5	4	3	2	1	0
1^o bit com valor 1	**111**	110	**101**	100	**011**	010	**001**	000
2^o bit com valor 1	**111**	**110**	101	100	**011**	**010**	001	000
3^o bit com valor 1	**111**	**110**	**101**	**100**	011	010	001	000
Uso	m_3	m_2	m_1	cv_2	m_0	cv_1	cv_0	Não usado
Exemplo	1	0	1	cv_2	0	cv_1	cv_0	-
$cv_0 = 0$	**1**		**1**		**0**			-
$cv_1 = 1$	**1**	**0**			**0**			-
$cv_2 = 0$	**1**	**0**	**1**					-

ele não é inserido na conta. sendo então ignorado. Assim a mensagem com o código de verificação passa a ser $m + cv = 1010010$.

O processo de checagem de erros é o mesmo que para o cálculo inicial. Após a recepção ou leitura a posterior da mensagem m' repete-se o processo de cálculo do novo código de verificação cv'. Deve-se tomar o cuidado de não utilizar os *bits* prévios do cv no cálculo do novo cv', dado que agora a mensagem está entrelaçada com aqueles.

A operação $cv \oplus cv'$ retorna um valor indicando a posição do erro. Caso este valor seja zero não houve erros na transmissão da mensagem. Deve-se atentar pois é possível que o erro esteja nos *bits* de correção. Deste modo é preciso avaliar se existe a necessidade de correção do *bit*, já que em algumas situações apenas os *bits* de dados devem ser corrigidos.

A Tabela 8 apresenta a situação onde há um erro no terceiro bit, m_2, da mensagem m, que teve seu valor alterado de 0 para 1. Deste modo, os novos valores calculados para c'_v serão $cv'_0 = m'_0 \oplus m'_1 \oplus m'_3 = 0$, $cv'_1 = m'_0 \oplus m'_2 \oplus m'_3 = 0$ e $cv'_2 = m'_1 \oplus m'_2 \oplus m'_3 = 1$.

Tabela 8: Detecção de um erro numa mensagem

Posição	7	6	5	4	3	2	1	0
Uso	m_3	m_2	m_1	cv_2	m_0	cv_1	cv_0	Não usado
Exemplo	1	0	1	0	0	1	0	-
Erro	1	1	1	0	0	1	0	-
cv' recalculado				1		0	0	-
$cv \oplus cv' = 110_2 = 6_{10}$				1		1	0	-

O resultado da operação $cv \oplus cv'$ retornou o valor 6, posição do bit m_2 alterado na mensagem.

Conforme apresentado na seção 2.3, a incidência de erros nas memórias, seja acidental ou proposital, é alta o suficiente para que os fabricantes chegassem a produzir sistemas que incluam em seu desenvolvimento as técnicas de correção, sejam as apresentadas nesta seção ou mesmo a simples adição de redundâncias. Várias destas abordagens, no entanto,

acarretam custos, seja por maior necessidade de memória, seja por redução no tempo de processamento disponível, que pode inviabilizar o projeto.

Algumas abordagens, visando a economia de ambos os recursos (memória e processamento), focaram em aumentar a confiabilidade apenas de áreas de memória mais críticas, principalmente aquelas responsáveis pelo núcleo do gerenciamento do sistema e das ações de controle, como em Wang et al. (2009), Borchert, Schirmeier e Spinczyk (2013). No entanto, as abordagens apresentadas focam em sistemas com maior capacidade de processamento, utilizando sistemas operacionais mais complexos e necessitando de recursos de compilação não disponíveis para sistemas embarcados de baixo custo.

Na próxima seção, será apresentado o desenvolvimento de um sistema operacional de tempo real composto de um *microkernel* e uma controladora de *drivers*. Estes foram desenvolvidos para permitir uma integração mais simples da metodologia de correção proposta com o escalonador e a troca de contexto.

3 Desenvolvimento

Tradicionalmente, os sistemas de proteção à memória são implementados em *hardware* por questões de velocidade (CHAUDHARI; PARK; ABRAHAM, 2013; LEMAY; GUNTER, 2012). As soluções em *software*, em geral, apresentam um consumo muito alto (PAXTEAM, 2012; KAI; XIN; GUO, 2012; YIM et al., 2011; VEN, 2004). Uma opção para reduzir esse consumo é realizar a proteção apenas das regiões mais importantes, que normalmente são os objetos do *kernel* (BORCHERT; SCHIRMEIER; SPINCZYK, 2013). No entanto, estas abordagens são focadas em sistemas *desktops* ou para embarcados com maior capacidade computacional, sendo inviáveis para processadores de baixo custo.

Entre sistemas que utilizam processadores de baixo custo estão diversos controladores industriais, painéis de elevadores comerciais, sensores inteligentes, carros e grande parte de eletrônicos com pouca interação humana.

A Figura 13 apresenta a solução proposta: realizar a verificação de erros em toda troca de contexto através de informações extras armazenadas na pilha de dados. Todos os acessos realizados pela troca de contexto terão suporte de um sistema de verificação de integridade da informação utilizando os algoritmos de CRC ou Hamming.

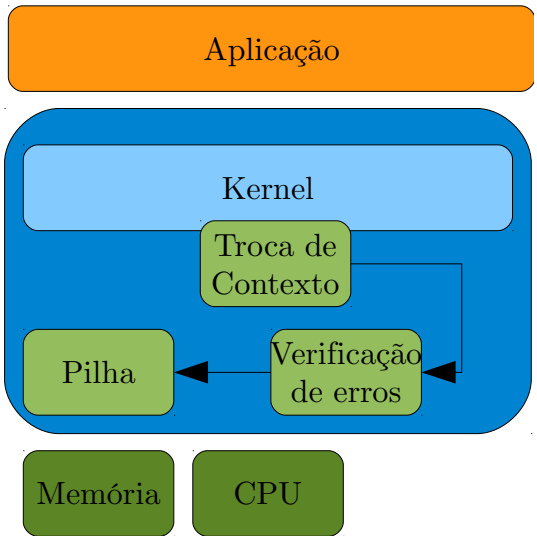


Figura 13: Modelo de sistema com verificação de erros na pilha

Optou-se neste trabalho por realizar a proteção por meio de algoritmos que gerem um código de verificação por bloco de memória, evitando-se assim o gasto demasiado de memória RAM. Com relação ao consumo de processamento foi dada preferência para os algoritmos mais simples com capacidade de correção ou detecção de erros.

As rotinas de troca de contexto de um sistema operacional são bastante complexas, primeiro por serem muito particulares para cada processador e segundo por possuírem códigos em *assembly*, de difícil adaptação.

Por este motivo optou-se pela utilização de um sistema operacional desenvolvido pelos autores (ALMEIDA; FERREIRA; VALÉRIO, 2013), facilitando a adaptação das rotinas necessárias na troca de contexto. O sistema operacional foi separado em 4 camadas: aplicação (amarelo), microkernel (vermelho), controladora de drivers (azul) e os drivers (preto e verde). A Figura 14 apresenta um resumo do sistema desenvolvido, a interligação do *kernel* com a aplicação e a controladora de *drivers* bem como todos os *drivers* implementados.

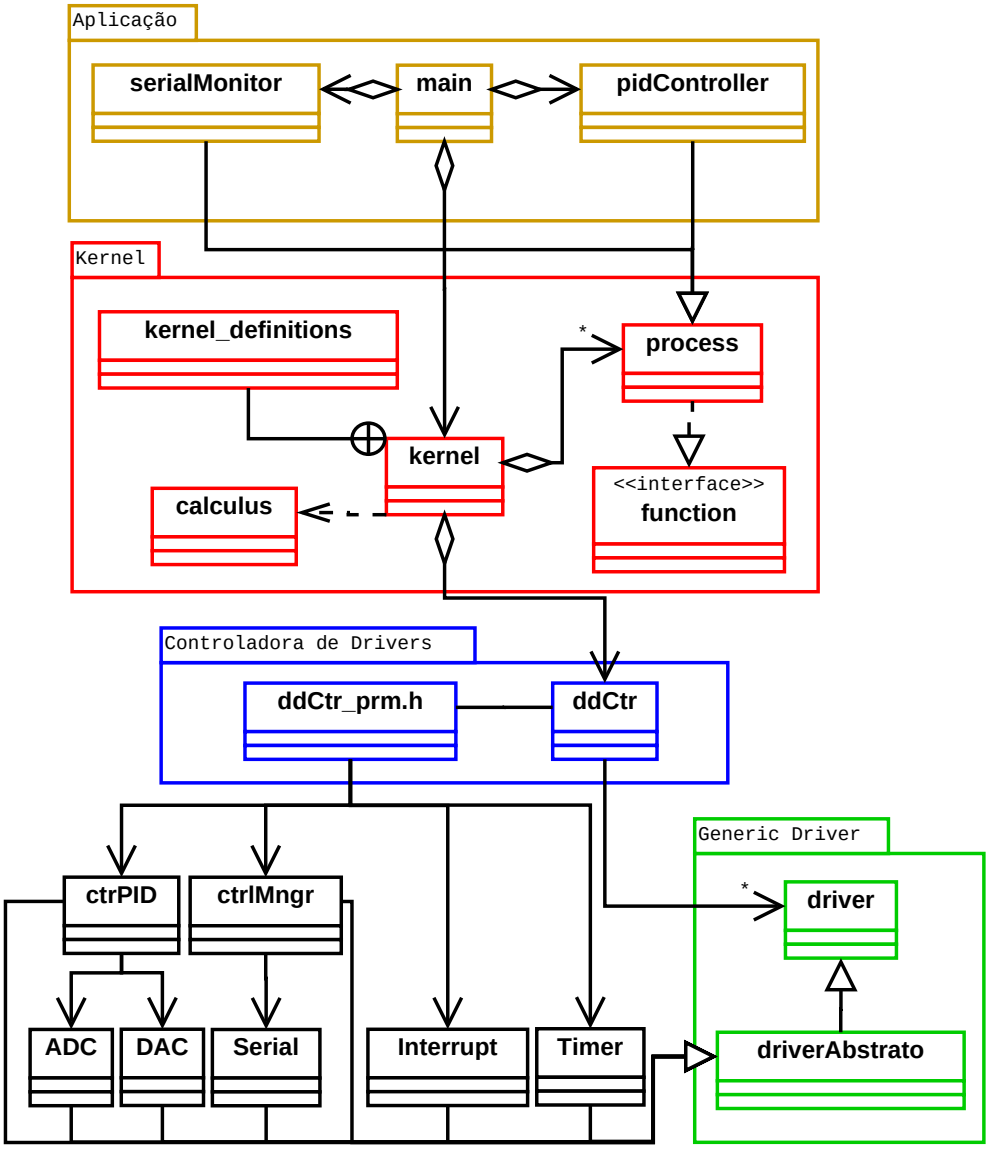


Figura 14: Diagrama UML do sistema desenvolvido

A aplicação pode ser composta de mais de um processo que são gerenciados pelo kernel através de uma estrutura do tipo *process*. Estes processos são implementados como funções contendo *loops* infinitos. A adição, remoção, pausa ou continuidade na execução dos processos é definida pelas interfaces disponibilizadas no kernel. Maiores detalhes do kernel e a implementação das rotinas são apresentados na próxima seção.

O desenvolvimento da controladora de *drivers* pode ser visto no anexo A. Desenvolveu-se uma interface bastante simples para gerenciamento das interações entre a aplicação e os dispositivos, sendo composta de apenas 3 funções. Esta simplificação foi possível pela utilização de uma estrutura comum para todos os drivers, apresentada em verde no diagrama da Figura 14.

Da estrutura apresentada, dois *drivers* devem ser notados: *drvPID* e *ctrlMngr*. Embora gerenciados pela controladora como *drivers* normais, eles não fazem acesso ao *hardware* diretamente. Eles agrupam informações de outros *drivers* ou provém novos modos de uso dos *drivers* apresentados.

O desenvolvimento se concentrou na implementação de uma troca de contexto segura em um *microkernel*. A estrutura de *microkernel* foi escolhida por questões de isolamento e segurança (TANENBAUM; HERDER; BOS, 2006). Foi desenvolvida uma controladora de *drivers* permitindo a exibição, armazenamento e análise dos dados recolhidos do sistema. Optou-se por um sistema de controle real, como plataforma de testes, principalmente por este tipo de sistema necessitar de execução em tempo real.

3.1 Microkernel

O uso de um *microkernel* em detrimento ao *kernel* monolítico se justifica pelo aumento da segurança do sistema (TANENBAUM; HERDER; BOS, 2006). Este aumento é pautado em duas características do *microkernel*: a maior separação entre *kernel*, *drivers* e processos e a maior simplicidade no código quando comparado com um *kernel* monolítico tradicional. A primeira permite que eventuais erros sejam isolados e tratados sem afetar todo o sistema, de modo que este possa se recuperar, aumentando sua robustez. A segunda característica pode ser traduzida num código menor, facilitando o trabalho de teste e validação, reduzindo as chances de que erros apareçam na execução do sistema.

O *microkernel* proposto foi desenvolvido para sistemas embarcados com poucos recursos de memória e processamento. Optou-se por criar uma estrutura mais simples para o gerenciamento dos processos. O gerenciamento é realizado por um *buffer* circular sobre um vetor do tipo *process*. As funções de manipulação são todas internas ao *kernel*. As exceções são as funções de adição de processos, de inicialização do *kernel* e de gerenciamento da interrupção do temporizador. A Figura 15 apresenta o modelo implementado. As funções para verificação de erro estão implementadas e reunidas na seção *error_check*. Elas são utilizadas pela função *KernelClock()*, responsável pela troca de contexto.

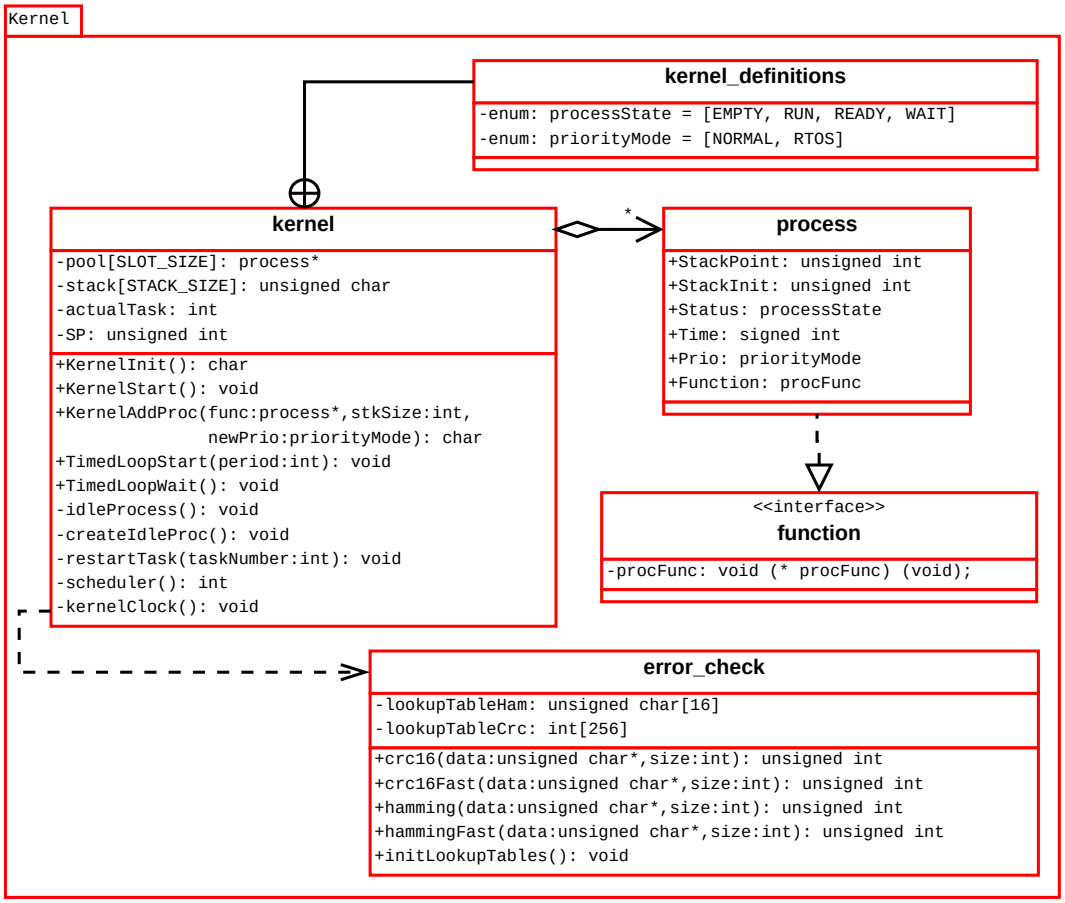


Figura 15: Modelo em UML desenvolvido para o *kernel* implementado

Os blocos de controle dos processos são armazenados na pilha. Por simplicidade optou-se por reservar uma região de memória para que cada processo tenha sua pilha. A criação desta área é feita na inicialização do processo. As pilhas são armazenadas numa região sequencial na memória e definida pelo *kernel*. Esta região é delimitada como um vetor *stack*, de tamanho *STACK_SIZE*, na inicialização do *kernel*.

A troca de contexto segura foi feita adicionando-se aos dados de controle de cada processo um código verificador *cv*, que servirá para detecção ou correção do erro, reduzindo os problemas advindos de erros na memória ou de erros no uso da pilha.

Para implementar esta função corretamente é imprescindível conhecer a arquitetura do processador para projetar a mudança de contexto corretamente.

O microcontrolador utilizado para testes é um Freescale MC9S12DT256, com um processador da família HCS12 com dois acumuladores de 8 *bits* (AccA e AccB), dois apontadores de 16 *bits* (IX e IY), um contador de programa de 16 *bits* (PC) e um apontador de pilha de 16 *bits* (SP). Os *bits* de informação de condição estão agrupados em um byte denominado CCR. Este processador pode ainda operar com um modelo de paginação de memória por meio de um registro de 1 byte denominado Ppage (FREESCALE, 2005). Estas variáveis, totalizando 10 bytes, devem ser armazenadas para guardar o estado atual

do processo.

O processador utilizado possui suporte para interrupções. Deste modo, o empilhamento e desempilhamento destas variáveis na região de pilha é feito automaticamente. A Tabela 9 apresenta o modelo de empilhamento utilizado no processador.

Tabela 9: Representação dos dados da CPU empilhados automaticamente na pilha

Posição na memória	Informação / Variável	Tamanho
stk+0	CCR	1 byte
stk+1	B	1 byte
stk+2	A	1 byte
stk+3	IX	2 bytes (Alto:Baixo)
stk+5	IY	2 bytes (Alto:Baixo)
stk+7	PC	2 bytes (Alto:Baixo)

Como é possível observar na Tabela 9, a variável Ppage não é inserida automaticamente quando ocorre uma interrupção. Além disso, é necessário criar um espaço de 2 bytes que será reservado para o resultado do código de detecção ou correção de dados. A Tabela 10 apresenta o modelo de empilhamento desenvolvido para este trabalho. Conforme citado, as variáveis manipuladas automaticamente pelo processador têm sua posição mantida.

Tabela 10: Dados da CPU empilhados na stack com informações de segurança

Posição na memória	Informação / Variável	Tamanho
stk-3	Resultado do CRC	2 bytes (Alto:Baixo)
stk-1	PPage	1 byte
stk+0	CCR	1 byte
stk+1	B	1 byte
stk+2	A	1 byte
stk+3	IX	2 bytes (Alto:Baixo)
stk+5	IY	2 bytes (Alto:Baixo)
stk+7	PC	2 bytes (Alto:Baixo)

O Código 2 apresenta a função responsável pela troca de contexto das tarefas e foi desenvolvido de acordo com as especificações apresentadas na Tabela 10.

Código 2: Rotina responsável por executar a troca de contexto entre os processos

```
1  void interrupt kernelClock(void){
2    //at this point CCR,D,X,Y,SP are stored on the stack
3    volatile unsigned int SPdummy;      //stack pointer temporary value
4    volatile unsigned int crc_on_stack;//point to the crc on the stack
5    crc_on_stack = 1;             //just to avoid optmization error
6      __asm PULD; __asm PULD;       //remove SPdummy & crc_on_stack
7    __asm LDAA 0x30; __asm PSHA;//storing PPage on the stack
8    __asm PSHD; __asm PSHD;       //recreating crc_on_stack & SPdummy
9    __asm TSX;                    //fill SPdummy with actual stack position
10   __asm STX SPdummy;

12   //storing check value
13   if (pool[actualTask].Prio == RTOS){
14     crc_on_stack  = hamming((unsigned char *)SPdummy+4,10);
15   } else {
16     crc_on_stack  = crc16((unsigned char *)SPdummy+4,10);
17   }
18   __asm TSX;       //save SP value on process info for further recover
19   __asm STX SPdummy;
20   pool[actualTask].StackPoint = SPdummy+2; //+2 to point to stack top
21   if (pool[actualTask].Status == RUNNING){
22     pool[actualTask].Status = READY;
23   }
24   actualTask = Scheduler();
25   pool[actualTask].Status = RUNNING;
26   SPdummy = pool[actualTask].StackPoint;
27   __asm LDX SPdummy; //load the next task SP from process info
28   __asm TXS;
29   __asm PSHD;         //restore space for SPdummy variable

31   //reading check value and checking the data integrity
32   if (pool[actualTask].Prio == RTOS){
33     SPdummy  = hamming((unsigned char ↩
         *)(pool[actualTask].StackPoint+2),10);
34     if (crc_on_stack != SPdummy) { //making XOR to find bit changed
35       crc_on_stack = (crc_on_stackˆSPdummy) - 136;
36       if (crc_on_stack < 80){
37 *((unsigned char *) (pool[actualTask].StackPoint+2+(crc_on_stack/8))) = ↩
    *((unsigned char *) (pool[actualTask].StackPoint+2+(crc_on_stack/8))) ↩
    ˆ (1<<(crc_on_stack%8));
38       }
39     }
40   } else {
41     SPdummy= crc16((unsigned char *)(pool[actualTask].StackPoint+2),10);
42     if (crc_on_stack != SPdummy) {
43        SPdummy = restartTask(actualTask)-2;
44        __asm LDX SPdummy;
45        __asm TXS;
46     }
47   }
48   __asm PULD; __asm PULD;       //remove crc_on_stack & SPdummy
49   __asm PULA; __asm STAA 0x30;//set PPage for the next process
50   CRGFLG = 0x80;                //clearing the RTI flag
51   __asm RTI;                    //All other context loading is done by RTI
52 }
```

O sistema de proteção é inserido na pilha nas linhas de 12 a 17. Já a verificação é realizada nas linhas de 31 a 47. Grande parte da manipulação de variáveis se faz necessária devido à inserção das variáveis locais na pilha, dificultando a montagem correta do bloco de contexto.

As variáveis *SPdummy* e *crc_on_stack*, são criadas na pilha logo após o preenchimento automático com os dados da CPU. Deste modo, para a correta manipulação da pilha é necessário retirar essas variáveis da pilha antes de executar o retorno da interrupção.

A prioridade do processo define se, na troca de contexto, será utilizada uma ferramenta de detecção ou de correção de erros. Esta opção em associar o tipo de ferramenta de detecção/correção de erros com a prioridade do processo é denominada neste trabalho de solução mista, sendo explicada em detalhes no capítulo 3.4.

O procedimento de escalonamento, por fim, é realizado em uma função separada após o processo atual ter sido corretamente armazenado e pausado. Ele retorna o próximo processo a ser executado no processador através de seu índice numa estrutura de armazenamento de processos *pool[]*. Este novo processo então tem seus dados checados, suas variáveis e contadores atualizados e o retorno da interrupção de hardware se encarrega de retirar esses dados da pilha e restaurá-los nos registros do processador, através da função "*__asm RTI;*".

Para manipulação dos processos pelo kernel foram criadas duas estruturas de dados. O Código 3 apresenta estas estruturas implementadas para armazenamento destes dados e a definição da *struct process*.

Código 3: Estruturas desenvolvidas para a gestão dos processos

```
1  //Processes management structures
2    //Reserving memory to use as stack for the processes
3    volatile unsigned char stack[HEAP_SIZE];

5    //Process pool to manage the process
6    volatile process pool[NUMBER_OF_TASKS];

8  //Process struct definition
9    typedef struct {
10     // Current position of virtual stack
11     volatile unsigned int StackPoint;
12     // Virtual stack start position
13     volatile unsigned int StackInit;
14     // Actual process state
15     volatile processState Status;
16     // Countdown timer for each process
17     signed int Time;
18     // Priority level for the process (RTOS(0) or NORMAL(1))
19     int prio;
20   }process;
```

A primeira estrutura (*char stack []*) opera como uma região de memória linear para

implementação da pilha do sistema. Esta estrutura foi implementada como um vetor de bytes. O *kernel* reserva a quantidade requisitada pelo processo no momento de sua criação. O tamanho desta região é definido em tempo de compilação através da definição *HEAP_SIZE*. O tamanho adequado desta estrutura depende da quantidade de processos, da quantidade de variáveis locais de cada processo e da quantidade de funções chamadas em cadeia (aninhadas). Como a memória disponível é baixa e devido a estrutura utilizada, recomenda-se não utilizar nenhum tipo de função recursiva.

A segunda estrutura (*process pool [NUMBER_OF_TASKS]*) armazena as informações relativas aos processos. Estas informações são necessárias para que a troca de contexto seja realizada de maneira correta, além de possuir informações extras para permitir diferentes modos de escalonamento.

O processo pode então ser definido, neste contexto, como: uma estrutura do tipo *process*, uma região de memória reservada para uso como pilha e o código em execução, apontado por um ponteiro de função. A Figura 16 apresenta esta definição graficamente. Enquanto estiver em execução, sua pilha possui apenas as variáveis criadas pelo processo.

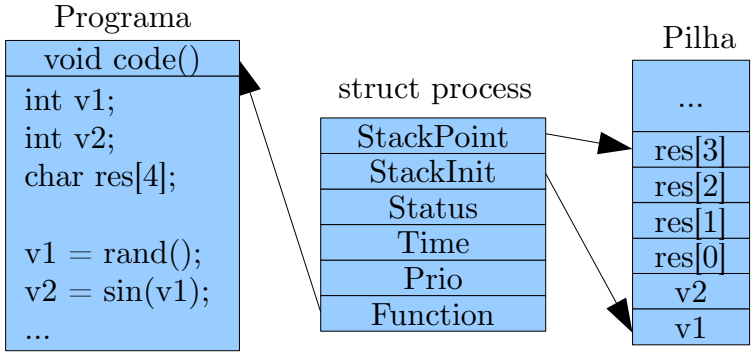


Figura 16: Estruturas desenvolvidas para controle dos processos

A criação de um novo processo, do ponto de vista do *kernel*, envolve dois passos: a inserção da referência de uma estrutura *process* no *pool* de processos e a reserva de uma área de memória para sua pilha particular. Além disto, para que a primeira execução seja realizada corretamente, é necessário preencher a pilha com valores adequados para a primeira mudança de contexto. Esta rotina é apresentada no Código 4.

A definição do tamanho da pilha do processo fica a cargo do programador. Ela deve ser suficiente para armazenar as variáveis locais do processo, um bloco de controle de processo, que no caso do sistema apresentado representa 10 bytes, e um bloco de controle por função chamada. Para um processo que tenha 5 variáveis do tipo int (2 bytes) e possua chamadas de função aninhadas em 3 níveis, deve reservar pelo menos $2 * 5 + 10 + 3 * 10 = 50\, bytes$.

A inicialização do *kernel* tem como principal necessidade a criação de um processo *idleProc* e a configuração do bloco de contexto deste, de modo que ele possa ser executado

Código 4: Função para a criação e adição de novos processos

```
process * kernelAddProc(void * func (void), int stkSize){
  volatile process * temp;
  unsigned int i;
  for(i=0;i<NUMBER_OF_TASKS;i++){
    if (pool[i].Status == EMPTY){
      nextTask = i;
      break;
    }
  }
  temp = &pool[nextTask];  //setting process values
  temp->StackInit = (unsigned int) SP;
  temp->Status = READY;
  temp->Time = 0;

//setting process control block on stack
  SP--;  *((unsigned char*)SP) = (unsigned char)((long)func >> 8); //pc low
  SP--;  *((unsigned char*)SP) = (unsigned char)((long)func >> 16);//pc hi
  SP--;  *((unsigned char*)SP) = 0x66; //y low
  SP--;  *((unsigned char*)SP) = 0x55; //y high
  SP--;  *((unsigned char*)SP) = 0x44; //x low
  SP--;  *((unsigned char*)SP) = 0x33; //x high
  SP--;  *((unsigned char*)SP) = 0x22; //b
  SP--;  *((unsigned char*)SP) = 0x11; //a
  SP--;  *((unsigned char*)SP) = 0x00; //CCR
  SP--;  *((unsigned char*)SP) = (unsigned char)((long)func >> 0); //PPAGE

//verification code generation
  if (temp->Prio == RTOS){
    i = hamming((unsigned char*)SP,10);
  } else {
    i = crc16((unsigned char*)SP,10);
  }
  SP--;  *((unsigned char*)SP) = (unsigned char)(i >> 0);
  SP--;  *((unsigned char*)SP) = (unsigned char)(i >> 8);

  temp->StackPoint = (unsigned int)SP; //stack end position
  SP = temp->StackInit - stkSize;  //point to the next free position
  return temp;
}
```

corretamente. É também importante inicializar as demais variáveis internas. O Código 5 apresenta esta função.

A criação do processo *idleProc* é um pouco diferente da criação dos outros processos. Por simplicidade, sua localização é fixada na última posição do *pool* de processos. Também não é calculado o CRC inicialmente, pois este processo será o primeiro a ser colocado na memória e não passará, em sua primeira execução, pela troca de contexto.

Além disso, o processo *idleProc* é o responsável por habilitar as interrupções. Optou-se por esta abordagem para evitar que alguma interrupção aconteça antes que todas as pilhas dos processos estivessem preenchidas.

Código 5: Função de inicialização do *kernel*

```
1  char kernelInit(void) {
2    unsigned char i;
3    //starting all positions as empty
4    for(i=0;i<NUMBER_OF_TASKS;i++){
5      pool[i].Status = EMPTY;
6    }
7    nextTask = 0;

9    //Pointing the SP to the bottom of stack
10   SP = (unsigned int)&stack + HEAP_SIZE;
11   CreateIdleProc();
12   actualTask = IDLE_PROC_ID;
13   return FIM_OK;
14 }

16 void CreateIdleProc(void){
17  //idle creation
18   pool[IDLE_PROC_ID].StackInit = SP;
19   pool[IDLE_PROC_ID].Status = READY;
20   pool[IDLE_PROC_ID].Function = idleProc;
21   SP--; *((unsigned char*)SP) = (unsigned char)((long)idleProc>> 8);
22   SP--; *((unsigned char*)SP) = (unsigned char)((long)idleProc>> 16);
23   SP--; *((unsigned char*)SP) = 0x66;
24   SP--; *((unsigned char*)SP) = 0x55;
25   SP--; *((unsigned char*)SP) = 0x44;
26   SP--; *((unsigned char*)SP) = 0x33;
27   SP--; *((unsigned char*)SP) = 0x22;
28   SP--; *((unsigned char*)SP) = 0x11;
29   SP--; *((unsigned char*)SP) = 0x00;
30   SP--; *((unsigned char*)SP) = (unsigned char)((long)idleProc >> 0);
31   pool[IDLE_PROC_ID].StackPoint = (unsigned int)SP;
32   SP = pool[IDLE_PROC_ID].StackInit - 20;
33 }

35 void idleProc(void) {
36   asm CLI; //enable interrupts only when everthing is settled up
37   CRGINT |= (unsigned char)0x80U; //start Real Time Interrupt
38   for(;;){  //the perfect place to energy saving
39   }
40 }
```

Por se tratar de um *kernel* cujo um dos objetivos é atingir os requisitos de tempo real, foram criadas duas funções para permitir a utilização de *loops* temporizados: *timedLoopStart()*, no inicio determinado o tempo do loop e a função *timedLoopWait()*, que indica ao *kernel* que o processo já terminou suas atividades do *loop* principal e está apenas aguardando seu temporizador interno chegar à zero. O Código 6 estas funções e também um processo exemplo. No exemplo a cada 1000 ticks o valor da porta analógica é amostrado e enviado ao LCD.

É importante lembrar que o processo ficará preso no laço apenas o tempo necessário para a próxima troca de contexto. Após esse tempo, ele será reexecutado apenas quando

o requisito de tempo for atendido (o contador interno se tornar zero ou negativo) e o escalonador selecioná-lo novamente para execução.

Código 6: Função para inserção de um tempo determinado entre execuções de um mesmo processo

```
1  void timedLoopStart(signed int valor){
2    pool[actualTask].Time = valor;
3  }

5  void timedLoopWait(void){
6    pool[actualTask].Status = WAITING;
7    while(pool[actualTask].Status == WAITING);
8  }

10 void exampleProc(void) {
11   volatile int adValue;
12   for(;;){                 //all process are implemented as infinte loops
13     timedLoopStart(1000); //setup internal clock value
14     calldriver(DRV_ADC, ADC_READ, &adValue);
15     calldriver(DRV_LCD, LCD_WRITE_INT, adValue);
16     timedLoopWait();       //wait the internal value reach zero
17   }
18 }
```

A função *timedLoopStart()* atualiza um contador interno do processo que é decrementado a cada troca de contexto. Deste modo, independente de eventos externos, o processo irá esperar a quantidade de tempo especificada dentro da função *timedLoopWait()*. Esta função faz com que o processo fique pausado até o contador interno chegar a zero.

Deve-se apenas tomar o cuidado para que as instruções entre as duas funções não estourem o tempo especificado.

3.2 Escalonador

Para efeito de análise do impacto da inserção da medida de segurança no tempo para a troca de contexto, além dos possíveis impactos na capacidade de garantia de tempo real, foram implementadas duas técnicas de escalonamento: *deadline* mais crítico primeiro e *round robin*. Para a garantia de tempo real adicionou-se um sistema de prioridades em ambos os escalonadores, dada que esta é uma prática comum em sistemas operacionais embarcados. Os algoritmos de escalonamento foram implementados na função *scheduler*, apresentada no Código 7.

Esta função contém o código fonte dos dois escalonadores utilizados no trabalho: *round robin* e *earliest deadline first*. A escolha entre os escalonadores é feita por *defines* em tempo de compilação. Optou-se por essa abordagem para reduzir o *overhead* de escolher o

Código 7: Função de escalonamento do *kernel* com as opções habilitadas por define

```
1  int Scheduler(void){
2  #ifdef prioRTOS
3    //RTOS priority check
4    for (i = 0; i < NUMBER_OF_TASKS; i++){
5      if((pool[i].Prio == RTOS) && (pool[i].Status == READY))
6        return i;
7    }
8  #endif

10 #ifdef RRS
11   i = (RRactualTask+1);
12   if(i>=NUMBER_OF_TASKS) { i = 0; }
13   while((i != RRactualTask) && (pool[i].Status != READY)){
14      i++;
15      if(i>=NUMBER_OF_TASKS) { i = 0; }
16   }
17   //if the variable i == nextTask there are no other task willing to run
18   if((i == RRactualTask) && (pool[i].Status == WAITING)){
19     return IDLE_PROC_ID;
20   } else {
21     RRactualTask=i;
22     next = i;
23   }
24 #endif

26 #ifdef CTES
27   i=0;
28   while(pool[i].Status != READY){ i++; }
29   next = i;
30   //the loop will iterate until the last process
31   for (i = (next+1); i < NUMBER_OF_TASKS; i++){
32     if((pool[i].Status == READY) && (pool[i].Time < pool[next].Time))
33       next = i;
34   }
35 #endif
36   return next;
37 }
```

algoritmo em tempo de execução, já que esta escolha seria realizada dentro da interrupção para troca de contexto.

A opção pelos dois algoritmos de escalonamento foi pautada no consumo de processamento e capacidade de garantia de tempo real. Além disso observou-se que os algoritmos escolhidos, em conjunto com o sistema de prioridade, são comuns nas aplicações de sistemas operacionais de tempo real.

O algoritmo EDF, (*earliest deadline first*) apresenta um alto consumo de CPU, já que a cada troca de contexto deve-se verificar entre todos os processos qual está com o *deadline* mais próximo. No entanto, é um dos poucos algoritmos de escalonamento que consegue garantir tempo real com 100% de ocupação do processador (PEEK, 2013).

Devido a dificuldade inerente em se mensurar o tempo restante para finalizar o pro-

cesso, optou-se por utilizar o prazo de re-execução como valor de *deadline*. Os processos que estão mais próximos de sua re-execução, ou numa situação crítica os mais atrasados, obtém prioridade do escalonador. Isto é feito com base num contador interno, que é atualizado a cada troca de contexto, mantendo uniformidade na contagem de tempo de todos os processos.

Já o algoritmo *round robin* foi escolhido devido ao baixo consumo de CPU, permitindo variação na comparação dos resultados e também pelo amplo uso deste modelo, ou de suas variações, em diversos sistemas, apesar de uso um pouco mais restrito no ambiente embarcado (RAO et al., 2008). Além disso, este algoritmo apresenta uma boa responsividade, o que é interessante para processos que realizam interface com o usuário. Esta abordagem pode se tornar uma boa alternativa principalmente se conjugada com um sistema de priorização para a garantia de tempo real.

Uma abordagem bastante utilizada pelos sistemas operacionais de tempo real é criar uma lista com os processos de tempo real com um sistema de prioridades, garantindo que ao menos os processos mais críticos serão atendidos antes dos (*deadlines*) programados. Os demais processos entram em outra fila, coordenada por exemplo pelo algoritmo de *round robin*, permitindo que todo o tempo de processamento não utilizado para os processos de tempo real seja dividido de modo uniforme, garantindo uma melhor responsividade.

Isto foi implementado na rotina de escalonamento antes dos dois escalonadores e também pode ser habilitada por *define*. Esta rotina é otimizada para a presença de apenas um processo do tipo RTOS, situação bastante comum nos sistemas embarcados de baixo custo, que, quando possuem a necessidade de tempo real, geralmente é para o controle ou monitoramento de uma única variável. Na presença de mais de um processo de tempo real, a priorização se dá pelo posicionamento na fila.

Solucionar este problema, no entanto, não é foco do trabalho. Além disso se esta situação está ocorrendo, dois processos RTOS querendo executar ao mesmo tempo, pode ser sinal de que o sistema não está corretamente dimensionado, pois está sendo exigido mais do que consegue processar.

3.3 Códigos de correção de erros

3.3.1 CRC

A implementação de CRC utilizada é baseada no polinômio padronizado CCITT-CRC16 com coeficientes $x^{16} + x^{12} + x^5 + 1$ cuja representação em hexadecimal é 0x11021. O décimo sétimo bit é simulado no processo, realizando-se primeiramente o teste do bit mais significativo antes de deslocar os *bits* para a direita. Deste modo, é possível especificar o polinômio com apenas 16 *bits* ou 0x1021.

Para simplificar a implementação, o polinômio é armazenado reversamente, resultando

Código 8: Cálculo do CRC

```
1  unsigned int polinomio  = 0x8408;
2  unsigned int crc16(unsigned char * data_p, unsigned int length)
3  {
4     unsigned char i;
5     unsigned int data;
6     unsigned int crc;
7     crc = 0xffff;
8     if (length == 0) {
9       return (~crc);
10    }
11    do {
12       for (i = 0, data = (unsigned int)0xff & *data_p++; i < 8; i++, ↩
             data >>= 1) {
13          if ((crc & 0x0001) ^ (data & 0x0001)) {
14            crc = (crc >> 1) ^ polinomio;
15          }
16          else{
17            crc >>= 1;
18          }
19       }
20    } while (--length);
21    crc = ~crc;
22    data = crc;
23    crc = (crc << 8) | (data >> 8 & 0xFF);
24    return (crc);
25 }
```

no valor 0x8408, conforme apresentado no Código 8. Nesta implementação, cada um dos bits de cada byte é testado individualmente contra o polinômio de forma cíclica até o fim da mensagem. Duas técnicas comuns para melhorar a capacidade do algoritmo também foram implementadas: inicializar o contador com todos os bits em 1 (um) e inverter o resultado antes de retornar o valor. Embora tais ações possam ser removidas sem problemas para o processamento, optou-se pela manutenção das mesmas por questões de padronização com o algoritmo CCITT-CRC16.

Uma grande vantagem desta implementação é que os coeficientes podem ser facilmente modificados, bastando alterar o valor da variável *polinomio*. Isto permite que sejam utilizados diferentes valores de polinômio para cada processo, ou até mesmo valores aleatórios de polinômio a cada execução. Apesar de permitir que polinômios mais fracos possam ser utilizados(KOOPMAN; CHAKRAVARTY, 2004), esta alteração dificulta a exploração de uma falha de *stack overflow* para tomada de controle da CPU, reduzindo a possibilidade de um invasor injetar um bloco de controle de processo pilha que seja consistente com o polinômio utilizado.

3.3.2 Hamming

A implementação do algoritmo de correção de Hamming foi focada na facilidade de cálculo para um processador de baixo custo. Uma das seções com maior consumo de tempo do algoritmo é a separação dos *bits* da mensagem m e o embaralhamento destes com os *bits* do código de verificação cv, como apresentado na Tabela 7.

A Figura 17 apresenta graficamente o embaralhamento dos *bits* de verificação cv_i e os *bits* de mensagem m_i. As posições marcadas em verde e azul estão disponíveis para armazenar dados da mensagem. No entanto, para uma arquitetura em 8 *bits*, é importante notar que nem todos os bytes podem armazenar apenas *bits* de mensagem. Alguns dos bytes têm *bits* de verificação, marcados em laranja claro, dado a estrutura funcional do algoritmo de Hamming.

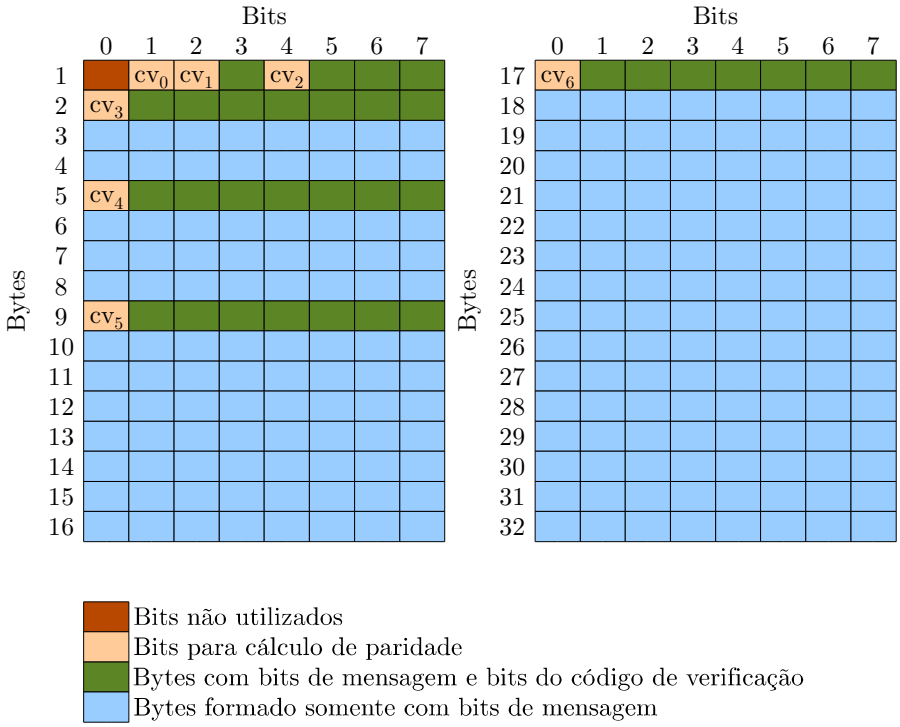


Figura 17: Disposição dos *bits* dos dados e do código de verificação na composição de uma mensagem

Ler a mensagem original e separar os *bits* para armazenar de acordo com o mapa apresentado é um procedimento extremamente custoso. A maioria dos processadores são otimizados para realizar operações sequenciais. A realização de operações combinacionais, principalmente se tais operações envolverem sequências binárias maiores que o tamanho dos registradores, fica prejudicada. A operação com bits individuais é ainda mais crítica, já que gasta-se o mesmo tempo para processar um bit do que para processar um byte, numa arquitetura com registros de 8 bits.

No contexto deste trabalho, o conjunto de dados a serem protegidos possui 10 bytes,

conforme apresentado na Tabela 10. Deste modo, ao intercalar os bits do bloco de controle de processo com os *bits* de verificação, o mapa de memória ficará similar ao mapa 1 da Figura 18. Pode-se observar pelo mapa que praticamente todos os bytes a serem armazenados (em azul e verde) tem seus bits divididos em duas posições de memória. Pelo mapa 1 observa-se também que são necessários 5 bits de correção para assegurar os 10 bytes.

Já o mapa 2 da Figura 18 apresenta uma solução onde nenhum dos bytes da mensagem original precisa ser armazenado em dois endereços de memória ou ter seus *bits* manipulados isoladamente. Esta abordagem, no entanto, requer o cálculo de 1 bit de verificação extra, no caso o bit cv_6. Este cálculo extra é compensado pela economia adquirida na manipulação integral

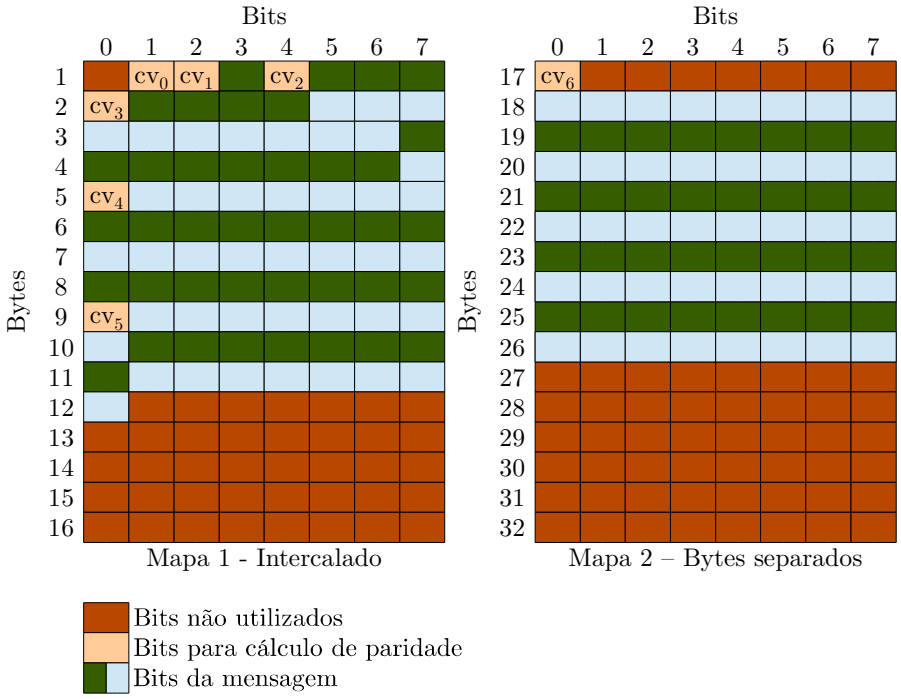


Figura 18: Mapas dos *bits* de dados e código de verificação: 1) interlaçamento normal, 2) utilização proposta

Além da vantagem de manipulação inteira de bytes, a não utilização dos espaços fracionados, do mapa 1, confere uma segunda vantagem. O cálculo dos *bits* de paridade, que envolvem a contagem de apenas algumas posições definidas, pode ser simplificado utilizando uma máscara. Esta máscara é diferente para cada um dos *bits* cv_i. A Tabela 11 apresenta as máscaras utilizadas no cálculo de cada um dos *bits* de paridade. Para os *bits* de paridade 3, 4, 5 e 6, não é necessário aplicar máscaras. O byte inteiro é contado, ou não, dependendo do posicionamento na mensagem.

A Figura 19 apresenta o mapa expandido dos bits utilizados em cada um dos valores cv_i. Como são desprezados os primeiros 8 bytes, por causa da não uniformidade no

Tabela 11: Modelo de mensagem para 4 *bits* de dados

Bit de Paridade	Máscara Utilizada
cv_0	0xAA
cv_1	0xCC
cv_2	0xF0
$cv_i, i >= 3$	0x00 ou 0xFF

posicionamento dos bits de checagem (mapa 1 versus mapa 2 da Figura 18). A contagem de bits começa em 128. Pode-se notar que as mascaras são uniformes para os três primeiros cv_i, conforme a Tabela 11. Para os demais bits a mascara pode ser $0x00$ ou $0xFF$, de acordo com o byte em questão.

cv_i	Bits de mensagem/correção								Máscara	Bits de mensagem/correção								Máscara
	128	129	130	131	132	133	134	135		136	137	138	139	140	141	142	143	
	h	0	1	2	3	4	5	6		0	1	2	3	4	5	6	7	
1		X		X		X		X	AA		X		X		X		X	AA
2			X	X			X	X	CC			X	X			X	X	CC
4					X	X	X	X	F0					X	X	X	X	F0
8									0	X	X	X	X	X	X	X	X	FF
16									0									0
32									0									0
64									0									0
	144	145	146	147	148	149	150	151		152	153	154	155	156	157	158	159	
	8	9	10	11	12	13	14	15	0	16	17	18	19	20	21	22	23	0
1		X		X		X		X	AA		X		X		X		X	AA
2			X	X			X	X	CC			X	X			X	X	CC
4					X	X	X	X	F0					X	X	X	X	F0
8									0	X	X	X	X	X	X	X	X	FF
16	X	X	X	X	X	X	X	X	FF	X	X	X	X	X	X	X	X	FF
32									0									0
64									0									0
	160	161	162	163	164	165	166	167		168	169	170	171	172	173	174	175	
	24	25	26	27	28	29	30	31		32	33	34	35	36	37	38	39	
1		X		X		X		X	AA		X		X		X		X	AA
2			X	X			X	X	CC			X	X			X	X	CC
4					X	X	X	X	F0					X	X	X	X	F0
8									0	X	X	X	X	X	X	X	X	FF
16									0									0
32	X	X	X	X	X	X	X	X	FF	X	X	X	X	X	X	X	X	FF
64									0									0
	176	177	178	179	180	181	182	183		184	185	186	187	188	189	190	191	
	40	41	42	43	44	45	46	47		48	49	50	51	52	53	54	55	
1		X		X		X		X	AA		X		X		X		X	AA
2			X	X			X	X	CC			X	X			X	X	CC
4					X	X	X	X	F0					X	X	X	X	F0
8									0	X	X	X	X	X	X	X	X	FF
16	X	X	X	X	X	X	X	X	FF	X	X	X	X	X	X	X	X	FF
32	X	X	X	X	X	X	X	X	FF	X	X	X	X	X	X	X	X	FF
64									0									0
	192	193	194	195	196	197	198	199		200	201	202	203	204	205	206	207	
	56	57	58	59	60	61	62	63		64	65	66	67	68	69	70	71	
1		X		X		X		X	AA		X		X		X		X	AA
2			X	X			X	X	CC			X	X			X	X	CC
4					X	X	X	X	F0					X	X	X	X	F0
8									0	X	X	X	X	X	X	X	X	FF
16									0									0
32									0									0
64	X	X	X	X	X	X	X	X	FF	X	X	X	X	X	X	X	X	FF
	208	209	210	211	212	213	214	215										
	72	73	74	75	76	77	78	79										
1		X		X		X		X	AA									
2			X	X			X	X	CC									
4					X	X	X	X	F0									
8									0									
16	X	X	X	X	X	X	X	X	FF									
32									0									
64	X	X	X	X	X	X	X	X	FF									

Figura 19: Disposição dos *bits* dos dados e do código de verificação na composição de mensagem final

Para o byte contendo os bits 152 à 159, por exemplo, para i = 5, ou 5o bit de checagem, todos os bits estão marcados com X na linha $cv_i = 16$, devendo ser utilizada a máscara $0xFF$.

O Código 9 apresenta a função para o cálculo do código de verificação de *Hamming* implementada.

Código 9: Cálculo do código Hamming de correção

```
1 #define MSG_SIZE 10
2 #define HMM_SIZE 7
3 #define TWO(c)      (0x1u << (c))
4 #define MASK(c)     (((unsigned int)(-1)) / (TWO(TWO(c)) + 1u))
5 #define COUNT(x,c) ((x) & MASK(c)) + (((x) >> (TWO(c))) & MASK(c))

7 int bitcount (unsigned int n)  {
8    n = COUNT(n, 0);
9    n = COUNT(n, 1);
10   n = COUNT(n, 2);
11   n = COUNT(n, 3);
12   return n;
13 }
14 unsigned int hamming(unsigned char * data_p, unsigned int length){
15   unsigned char poolH=0;
16   unsigned char hammingBits = 0;
17   unsigned char pBit=0;
18   unsigned char msg;
19     //calculating each of the hamming bits
20   for(msg=0; msg < length; msg++){
21     poolH += bitcount(*(data_p+msg) & 0xAA);
22   }
23   hammingBits += (poolH & 0x01) << pBit;
24   poolH = 0;
25   pBit = 1;
26   for(msg=0; msg < MSG_SIZE; msg++){
27     poolH += bitcount(*(data_p + msg) & 0xCC);
28   }
29   hammingBits += (poolH & 0x01) << pBit;
30   poolH = 0;
31   pBit = 2;
32   for(msg=0; msg < MSG_SIZE; msg++){
33     poolH += bitcount(*(data_p + msg) & 0xF0);
34   }
35   hammingBits += (poolH&0x01) << pBit;
36   for(pBit=3; pBit < HMM_SIZE; pBit++){
37     poolH = 0;
38     for(msg=0; msg < MSG_SIZE; msg++){
39       if((msg*8+136) & (1 << pBit)){
40         poolH += bitcount(*(data_p+msg));
41       }
42     }
43     hammingBits += (poolH&0x01)<<pBit;
44   }
45   return (unsigned int)hammingBits;
46 }
```

Pela Tabela 11, nota-se que apenas os três primeiros bits cv_i possuem mascaras definidas. Para todos os demais as mascaras se traduzem nos valores 0x00 ou 0xFF, eliminando a necessidade da mascara. Isto se traduz no código, onde os três primeiros bits possuem códigos ligeiramente diferentes. Os demais bits de verificação são tratados no *loop* final.

O procedimento de cálculo dos *bits* de paridade envolve a contagem de *bits* com valor 1 (um) numa variável binária. Buscando reduzir o tempo de processamento, foi utilizado um algoritmo O(log(n)) conhecido como contagem paralela de *bits*(DEVICES, 2005). Neste algoritmo os bits são somados paralelamente utilizando operações binárias. A quantidade de passos utilizados na operação *bitcount()* está relacionada ao tamanho de palavra do processador.

3.4 Solução mista

No desenvolvimento de sistemas embarcados, nem todos os processos possuem a mesma criticidade. Aqueles que exigem características de tempo real são os mais prejudicados no caso de uma falha.

Dado o custo de processamento para o cálculo dos códigos de correção serem superiores aos de detecção de erros optou-se por adotar uma abordagem diferente para os processos que exigem tempo real (RT) e os que não exigem (normais). Espera-se que deste modo possa-se garantir os requisitos de tempo real para os processos que precisam, sem onerar demasiadamente o consumo de processamento.

O algoritmo de CRC não permite a correção de erros, apenas a detecção. Deste modo, quando um erro é encontrado, é necessário reiniciar o processo. O procedimento de reinicialização envolve a perda de informações referentes ao estado atual do processo: suas variáveis internas e sua posição de execução atual. Há também um consumo considerável de tempo entre realizar a detecção de um erro na pilha e reinicializar o processo para seu estado inicial.

O tempo gasto para a reinicialização pode fazer com que um processo não atinja sua taxa de execução. Contudo, isto deve acontecer apenas uma vez durante a sua reinicialização.

Um ponto mais crítico, porém, é a perda das variáveis internas. O processo, perdendo seu estado atual, pode retornar em um modo não seguro para o sistema, gerando uma situação de risco. Visando a sanar este problema, optou-se por utilizar o algoritmo de Hamming para os processos do tipo RT. Apesar da estrutura utilizada detectar e corrigir apenas 1 bit errado, esta é uma quantidade adequada, visto que a probabilidade de acontecerem duas falhas simultâneas é muito baixa. Este é o motivo que leva os fabricantes de memória com capacidade de correção, do tipo ECC (*error-correcting code*, a utilizarem apenas 1 bit de correção.

Já os processos normais podem arcar com o custo da reinicialização. Assim o sistema

continuará em funcionamento sem um consumo demasiado de CPU. O procedimento de troca de contexto, utilizando a metodologia mista, é apresentado na Figura 20.

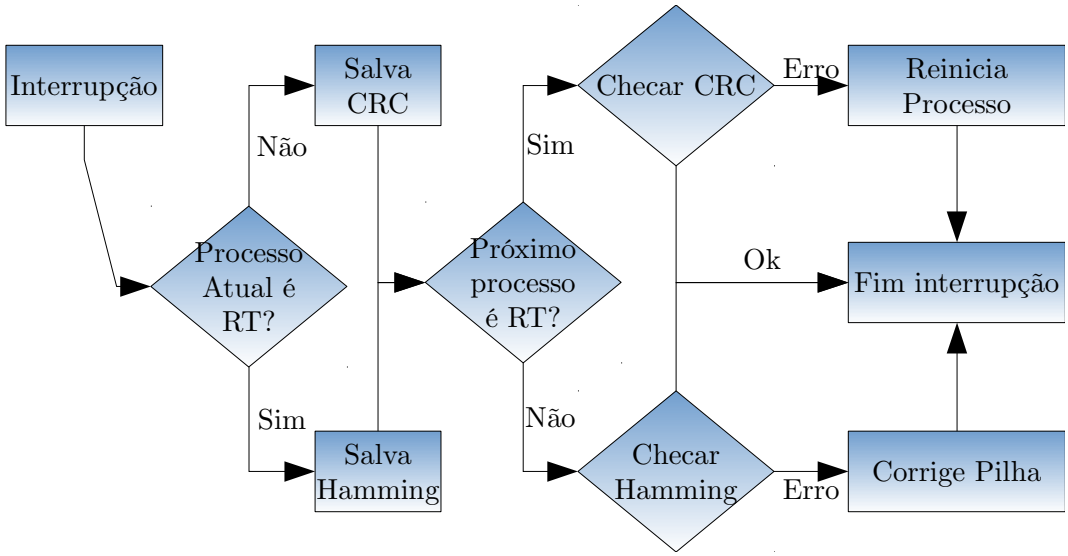


Figura 20: Metodologia mista para troca de contexto

Na realidade, o consumo de processamento será um valor entre o consumo dado por CRC e o do Hamming. Conhecidos os valores de consumo para uma troca de contexto com algoritmo de detecção CRC e uma troca de contexto com algoritmo de detecção Hamming é possível estimar o consumo da abordagem mista, bastando conhecer a razão de execução dos processos de tempo real. Seja f_i a frequência de execução do processo i. Para um sistema com uma troca de contexto temporizada a cada intervalo de tempo t_{tick} pode-se escrever a taxa de execução r_i conforme Equação 3.1.

$$r_i = f_i * t_{tick} \tag{3.1}$$

Sendo i_{rt} os processos de tempo real, define-se r_{rt} como a utilização da CPU para processos de tempo real. Este valor é o somatório das taxas de execução de todos os processos de tempo real, definido na Equação 3.2.

$$r_{rt} = \sum_{i_{rt}=0}^{N\_PROC\_RT} r_{i_{rt}} \tag{3.2}$$

O consumo da CPU, c_{total}, pode então ser definido conhecendo-se: o consumo de um sistema com CRC, c_{crc}, e o consumo de um sistema com Hamming, c_{ham}. A Equação 3.3 apresenta este cálculo.

$$c_{total} = r_{rt} * c_{ham} + (1 - r_{rt}) * c_{crc} \tag{3.3}$$

A utilização dos processos normais pode ser definida como $(1 - r_{rt})$, pois, na ausência de processos de tempo real, são executados os processos normais ou o processo *idleProc*, sendo que ambos utilizam o código CRC.

Para baixos valores de r_{rt}, o consumo da CPU pelo método misto se aproxima do consumo do CRC. Mesmo assim, o sistema mantém as vantagens da correção de Hamming para os processos de tempo real.

O gráfico da Figura 21 apresenta este comportamento para alguns valores de r_{rt}. O gráfico foi obtido para c_{crc} de 20% e para c_{ham} de 50%.

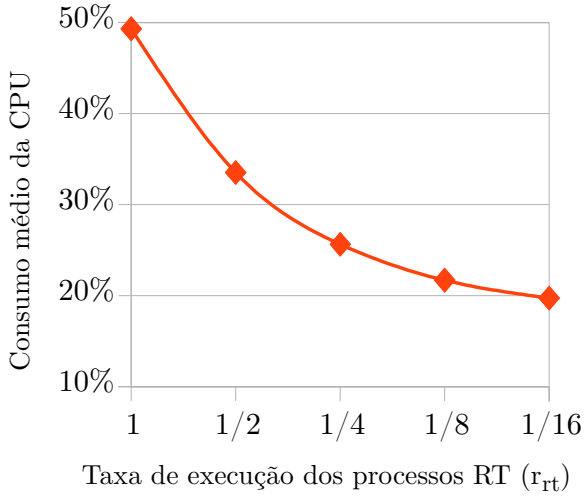


Figura 21: Utilização do processador pela troca de contexto do método misto

Pelo gráfico da Figura 21 percebe-se que há uma economia de quase 25% para um valor de $r_{rt} = 1/4$. Este valor de r_{rt}, por exemplo, é o mesmo obtido para um sistema com um processo de tempo real que tem uma taxa de execução 4 vezes menor que o *tick* do sistema operacional. Supondo um sistema que tenha um *tick* de 1(ms) isso significa uma taxa de execução do processo de tempo real de $250Hz$, ou um período de 4(ms).

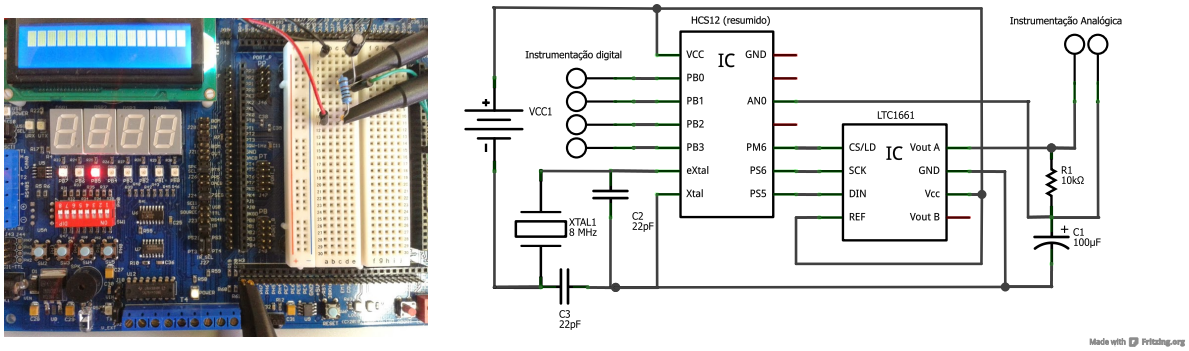
Estes valores, tanto da taxa de execução do processo RT, quanto a economia prevista para a solução mista, podem ser pontos críticos para a viabilidade da implementação desta técnica em sistemas embarcados de baixo custo.

3.5 Aplicação teste para sistemas de controle

Foi desenvolvido um sistema de controle para realização de testes com o objetivo de verificar o impacto da correção de erros na estabilidade e confiabilidade do sistema.

O *hardware* utilizado foi implementado utilizando o microcontrolador MC9S12DT256. A interface com a planta foi feita através de entradas e saídas analógicas. A entrada é lida através de uma entrada analógica do microcontrolador e a saída é enviada via protocolo

SPI para um conversor digital analógico. A Figura 22a apresenta o *hardware* utilizado e a Figura 22b o esquemático das ligações do sistema e da planta.



(a) Foto do sistema instrumentado

(b) Esquemático do *hardware* utilizado

Figura 22: Plataforma de *hardware* utilizada para testes

Para o desenvolvimento dessa aplicação foi necessária a criação de um *driver* para o ADC e outro para o DAC, utilizando o protocolo SPI. Estes *drivers* seguem o padrão de interface da controladora de *drivers*, que é explicada em detalhes no anexo A. Ambos os *drivers* de DAC e ADC possuem apenas uma função cada além das duas exigidas pelo padrão.

O driver do DAC recebe um valor por parâmetro e envia para o LTC1661 via protocolo SPI. A biblioteca implementada permite acionar ambas as saídas do LTC1661 prevendo futuras expansões do sistema. Já o ADC inicializa e aguarda o fim da conversão para retornar o valor por referência no parâmetro enviado.

As demais funções são necessárias para o gerenciamento do *driver* por parte da controladora. A Figura 23 apresenta a modelagem de ambos. Nota-se a similaridade por causa da padronização dos *drivers*.

drvADC
-thisDriver: driver -this_functions: ptrFuncDrv[ ] +enum = [ADC_INIT, ADC_READ, ADC_END]
+getADCDriver(parameters:void*): driver* -initADC(parameters:void*): char -readADC(parameters:void*): char

drvDAC
-thisDriver: driver -this_functions: ptrFuncDrv[ ] +enum = [DAC_OUT, DAC_INIT, DAC_END]
+getADCDriver(parameters:void*): driver* -initDAC(parameters:void*): char -writeDAC(parameters:void*): char

Figura 23: Especificação para os drivers do ADC e DAC

O controlador PID digital foi implementado conforme a Equação 3.4, onde $U(n)$ é o valor do sinal de saída no instante n, $e(n)$ o erro medido no instante n, k_p, k_i e k_d são

as constantes do controlador e T é o período de amostragem. A formulação detalhada da equação e as transformadas utilizadas se encontram no anexo B.

$$\begin{aligned} U(n) = U(n-2) + k_p\left(e(n) - e(n-2)\right) + \\ k_i\left(e(n) + 2.e(n-1) + e(n-2)\right)\frac{T}{2} + \\ k_i\left(e(n) - 2.e(n-1) + e(n-2)\right)\frac{2}{T} \end{aligned} \tag{3.4}$$

Procurando facilitar a utilização deste controle, uma segunda camada de abstração foi desenvolvida: o *drvPID*, apresentado na Figura 24. O objetivo principal é agrupar os *drivers* de entrada e saída analógica, necessários para a execução do sistema e a realização do cálculo da equação apresentada, permitindo realizar a gestão do controlador PID de modo unificado. Todos os valores dos coeficientes, gestão da saturação, ajuste de parâmetros e execução dos cálculos são encapsulados nesta camada, dispensando o programador de implementar os detalhes mais complexos da interligação entre os *drivers* de entrada e saída.

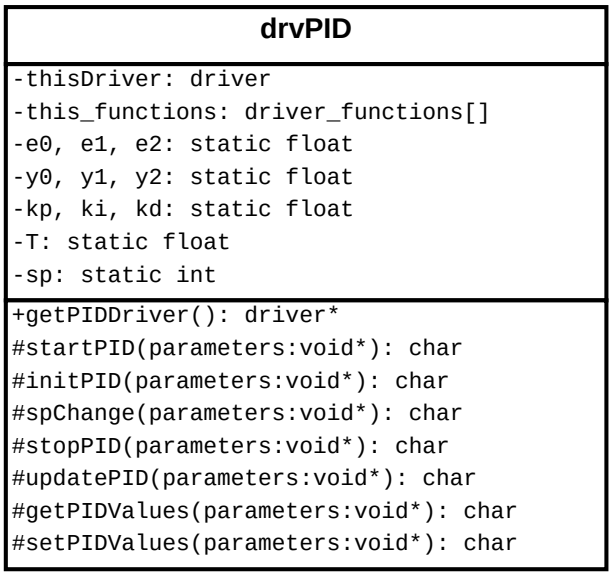


Figura 24: Especificação para o driver da controladora PID

Já o *driver* da comunicação serial, além das duas funções necessárias do modelo (*initSerial()* e *getSerialDriver()*) implementa as funções de envio e recepção de dados. Sua estrutura, bem como as funções implementadas, é apresentada na Figura 25.

Para evitar a perda de bytes na recepção, já que não há *buffer* de *hardware*, o *driver* implementa um sistema de *callback* para que os dados sejam recebidos pela interrupção e armazenados corretamente.

Uma interface simples para gestão e automação de testes com a placa foi desenvolvida com o intuito de gerenciar os comandos recebidos pela serial e executar as ações correspondentes.

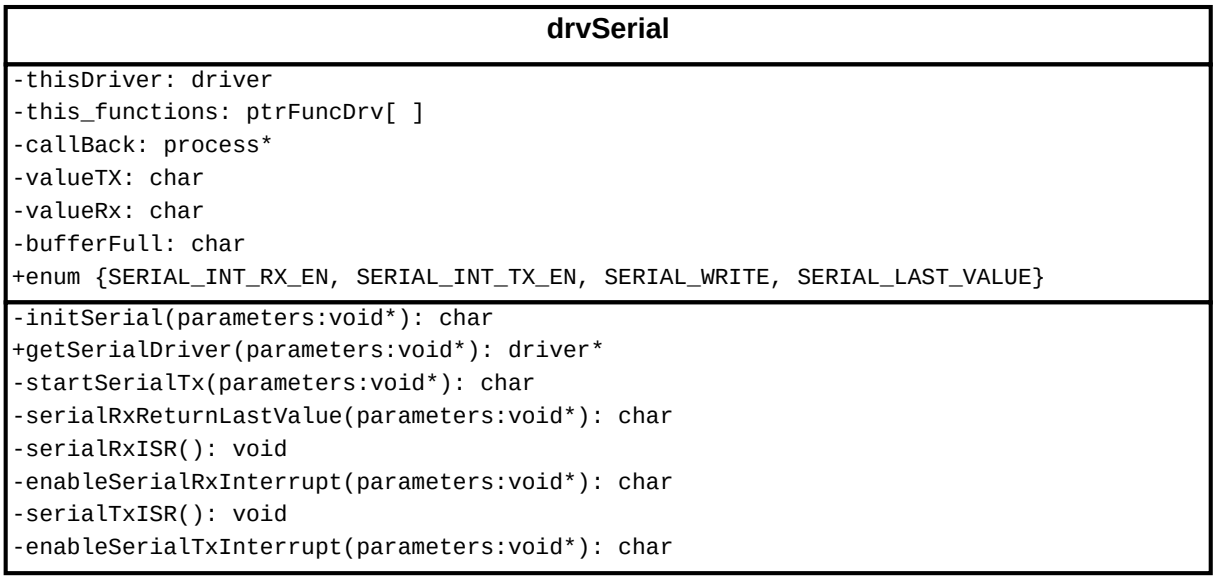


Figura 25: Especificação para o o driver da serial

O gerenciador *ctrlMngr*, conforme Figura 26, implementa um *buffer* de dados que servirá como armazenador temporário dos dados recebidos da serial. Neste buffer estarão as mensagens recebidas da serial. Ele é responsável, portanto, pela detecção, interpretação e checagem da integridade das mensagens.

Os comandos aceitos pelo gerenciador foram padronizados e as mensagens são delimitadas por um byte de início e outro de fim. Também foram inseridos dois bytes que armazenam um código de CRC para garantir a integridade da mensagem.

A definição deste protocolo, os tipos de mensagem e exemplos de utilização são apresentados no anexo C.

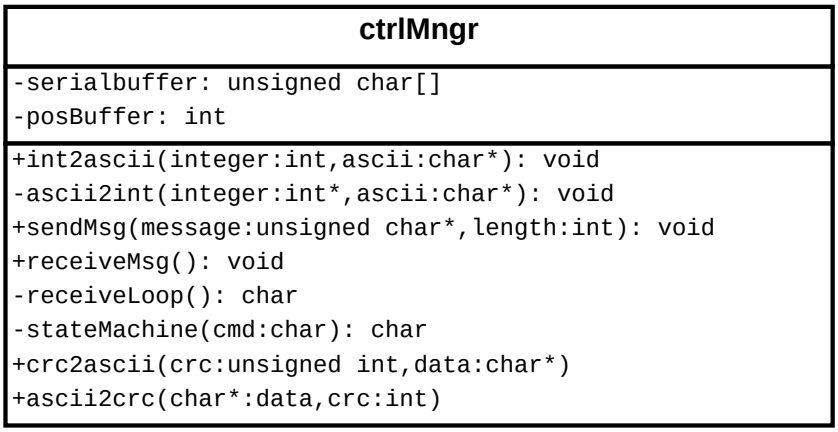


Figura 26: Especificação para os driver da gerenciadora de comandos serial

Três interfaces de gestão foram implementadas na máquina de estado interna do *ctrlMngr*: a gestão de processos com a adição e remoção dinâmica para testes de carga de processamento; a modificação dos parâmetros da controladora PID nos testes de veri-

ficação de dinâmica do sistema e a injeção de erros, para instrumentação e visualização do impacto dos erros em *bits* na continuidade de execução dos processos.

De posse do sistema operacional de tempo real com as alterações de segurança incluídas na troca de contexto, foram desenvolvidos testes para verificação da estabilidade do sistema, do consumo adicional por parte da segurança e da capacidade do sistema de manter um algoritmo de controle digital em funcionamento com as restrições de tempo envolvidas na aplicação teste.

A execução destes testes destes resultados só foi realizada em tempo hábil por causa da interface de gestão e controle da placa via comunicação serial. Estes resultados foram compilados e são apresentados na próxima seção.

4 Resultados

Os testes foram realizados no kit de desenvolvimento Dragon12 com um processador de 8 *bits*, com suporte a algumas operações de 16 *bits*, e um *clock* de 8 MHz. A placa é baseada no microcontrolador MC9S12DT256, com um conjunto de periféricos externos já embutidos. Entre os periféricos externos destaca-se o LTC1661 que é utilizado neste trabalho como saída analógica para o controlador implementado.

O sistema operacional foi desenvolvido tendo-se em mente a simplicidade de modo que a sobrecarga fosse mínima e a adaptação do código fácil. Uma primeira versão do sistema, portado para o processador PIC18f4550 é apresentada por Almeida, Ferreira e Valério (2013). Após melhorias no sistema e a adição da preempção, esta versão foi portada para o processador HCS12. A comparação destas versões, bem como os dados de outros sistemas de tempo real, foram compilados na Tabela 12.

Os cinco bytes extras exigidos pelo sistema proposto com correção não se referem a consumo estático, mas às variáveis internas das funções que são alocadas na pilha. O grande aumento para o sistema proposto com as rotinas de correção otimizadas se deve, principalmente, ao uso de *lookup-tables*, para o CRC de 512 bytes e para o Hamming de

Tabela 12: Comparação de consumo de memória entre sistemas operacionais de tempo real

Sistema Operacional	Consumo de Flash (mínimo)	Consumo de RAM (mínimo)
VxWorks (RIVER, 2013)	> 75.000	-
FreeRTOS (ENGINEERS, 2013)	> 6.000	> 800
uC/OS (MICRIUM, 2013)	> 5.000	-
Microkernel (ALMEIDA; FERREIRA; VALÉRIO, 2013)	2.948	619
uOS (VAKULENKO, 2011)	> 2.000	> 200
BRTOS (DENARDIN; BARRIQUELLO, 2013)	> 2.000	< 100
Proposto (mínimo)	871	71
Proposto com algoritmo de correção sem *lookup table*	1702	76
Proposto com algoritmo de correção com *lookup table* na RAM	1729	620
Proposto com algoritmo de correção com *lookup table* na Flash	2273	76

32 bytes. Este consumo pode ser deslocado da RAM para a memória não volátil. Deste modo, o consumo de memória *flash* passaria de 1729 para 2273, e o consumo de RAM volta ao patamar do sistema não otimizado, de 76 bytes.

Em termos de consumo de memória, o sistema proposto apresenta os menores valores entre os sistemas da Tabela 12. Mesmo quando considerada a versão com o sistema de correção ativo, os valores ainda são compatíveis com as alternativas. O gráfico na Figura 27 apresenta em azul os valores base para o sistema operacional proposto com os dois escalonadores implementados. As barras em vermelho indicam o consumo de memória adicional para os três métodos de proteção deste trabalho (CRC, Hamming e Misto), além da controladora de dispositivos e do sistema de prioridade.

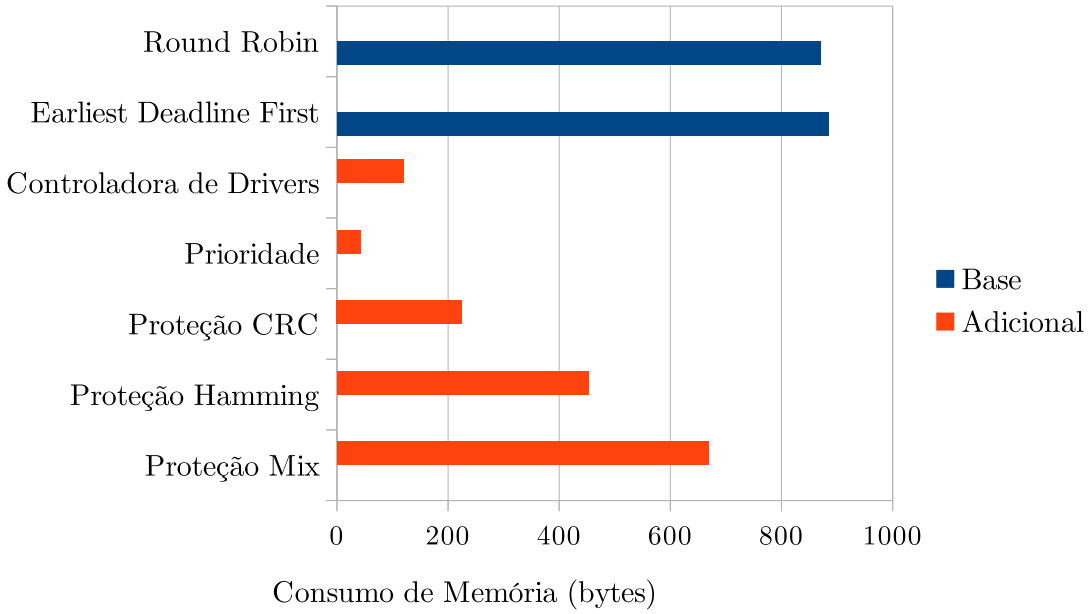


Figura 27: Consumo de memória do sistema operacional e valores adicionais dos métodos de correção

Nota-se que o consumo de memória dos dois escalonadores é praticamente iguais. O EDF consome 1,6% de flash a mais e o Round Robin exige 2,8% a mais de RAM.

As medições de tempo e consumo de processador foram feitas utilizando um osciloscópio Tectronix MDO4054-3 e utilizando quatro terminais de saída do microcontrolador. A opção por utilizar os terminais, em detrimento da saída serial, é que esta abordagem impacta menos no sistema, produzindo uma medição de tempo mais confiável. Os terminais são ligados ao início e desligados ao término do evento que se deseja medir. Foram medidos inicialmente quatro eventos:

- Troca de contexto
- Processo *idle*

- Processos normais
- Processo RT

A troca de contexto foi medida para análise do impacto das técnicas de correção no consumo de processamento. O processo *idle* representa a capacidade ociosa do sistema. Os processos normais são processos idênticos, gerados a partir da mesma função, com o intuito de simular a utilização do sistema. Foi desenvolvida uma função que consome aproximadamente 1,8 (ms), cerca de 90% do tempo entre trocas de contexto.

Nesta primeira análise, foi utilizado apenas um processo RT com duas funções: sincronizar as leituras de consumo pelo osciloscópio, além de permitir um caminho para a adição e remoção dos processos normais permitindo o levantamento mais rápido dos valores.

A Figura 28 apresenta uma das medições. O processo RT foi agendado para ser executado a cada 51,2 (ms), ou 25 trocas de contexto. A medição foi feita pelo canal 4, em verde. Utilizou-se esta janela, entre os dois pulsos em verde, para realizar as medições de tempo.

As trocas de contexto são medidas pelo canal 2, em azul. Os processos normais são lidos pelo canal 3, em rosa, e o canal 1, amarelo, apresenta o tempo que o processo *idle* está em execução, representando o tempo efetivamente livre.

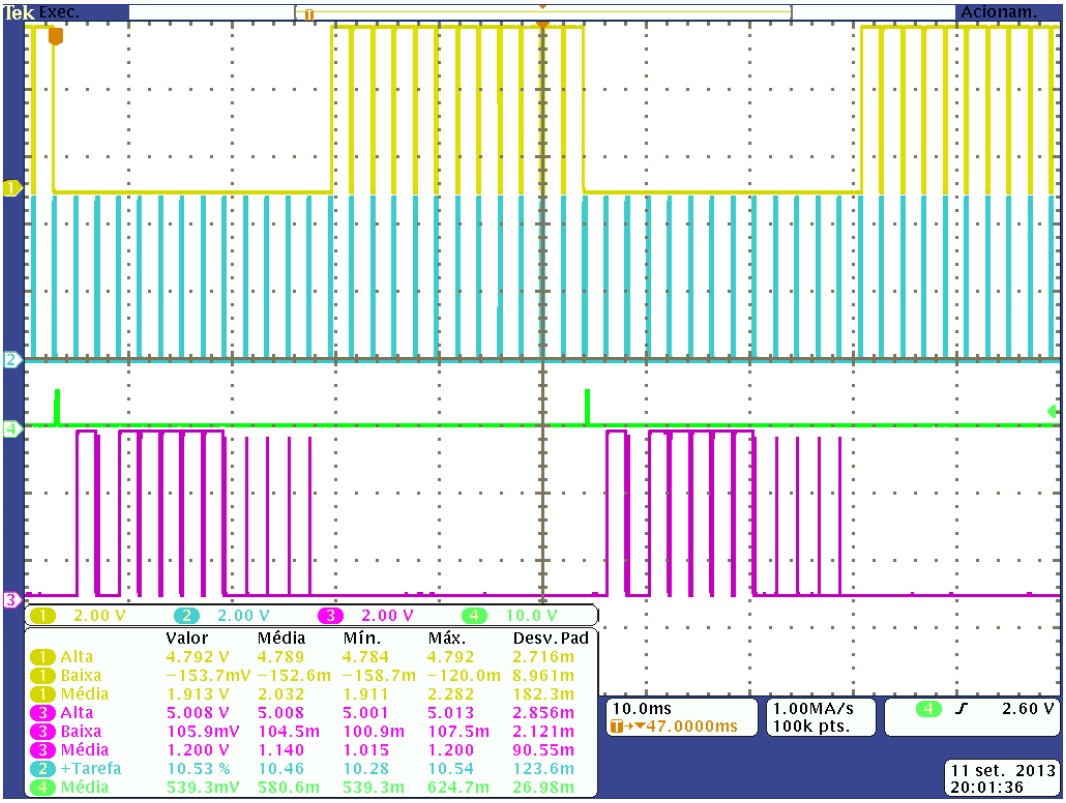


Figura 28: Medição de tempos utilizando um osciloscópio

Utilizou-se de ferramentas internas do osciloscópio para se realizar as medições de consumo de processamento. Para a troca de contexto, foi possível utilizar o cálculo do *duty-cycle*, ou ciclo de trabalho. Este valor indica, em percentagem, quanto do tempo de um ciclo o sinal está em nível alto.

Esta abordagem, no entanto, não pode ser aplicada a sinais não periódicos. Por este motivo utilizou-se da definição de valor médio V_{avg} de um sinal elétrico dada pela equação 4.1.

$$V_{avg} = \frac{1}{T} \int_0^T f(x)\,\mathrm{d}x \tag{4.1}$$

Por ser a representação de um sinal digital, a função só assume dois valores: VCC e 0. Separando-se a integral em duas partes, uma representando a parte positiva do sinal (com duração $t1$) e a segunda representando a parte negativa, é possível reduzir a equação, que irá depender apenas de VCC, $t1$ e T conforme a equação 4.2.

$$V_{avg} = \frac{1}{T}\left(\int_0^{t1} f(x)\,\mathrm{d}x + \int_{t1}^{T} f(x)\,\mathrm{d}x\right) = \frac{1}{T}\left(\int_0^{t1} VCC\,\mathrm{d}x + \int_{t1}^{T} 0\,\mathrm{d}x\right) = \frac{VCC * t1}{T} \tag{4.2}$$

Dividindo-se por VCC tem-se a razão $\frac{t1}{T}$, que é precisamente a razão de tempo do sistema no qual aquele sinal permaneceu em valor alto. Este valor pode ser interpretado como a porcentagem de tempo consumida pelo evento medido por aquele sinal.

Na Figura 28 são apresentados os valores médios das ondas. Na configuração apresentada existem 6 processos normais em execução. Como a troca de contexto consome 10.46%, os processos normais precisam de dois *slots* de tempo para terminarem, totalizando um consumo de 12 *slots*. Descartando-se estes 12 *slots* e o *slot* adicional para o processo RT, tem-se $25 - 12 - 1 = 12$ *slots* de execução de processos *idle*, perfazendo 12/25, ou 48%. No entanto devemos descontar a troca de contexto gasta neste período. Como 10,46% são gastos com a troca de contexto, apenas (100%-10,46%), ou 89,54% podem ser creditados como tempo hábil. Deste modo o tempo disponível para processamento é de $48\% * (89,54\%) = 42,98\%$. Já o valor medido por meio do valor médio é $(1,913 + 0,153)/(4,789 + 0,153) = 41,80\%$, bastante próximo do valor inferido.

O gasto computacional com a troca de contexto é diretamente dependente do número de processos, principalmente pela necessidade de se atualizar os temporizadores internos de cada processo. Este gasto é representado pela linha amarela do gráfico da Figura 29. As linhas vermelha e azul apresentam o consumo por processo dos dois escalonadores implementados.

Descontando-se o gasto com a atualização dos processos, pode-se notar que, para o escalonador *round robin*, a variação com o aumento do número de processos é praticamente nula. Já o escalonador do tipo EDF apresenta um incremento de 0,21% por processo, indicado pela inclinação superior quando comparada com a linha amarela. Por verificar

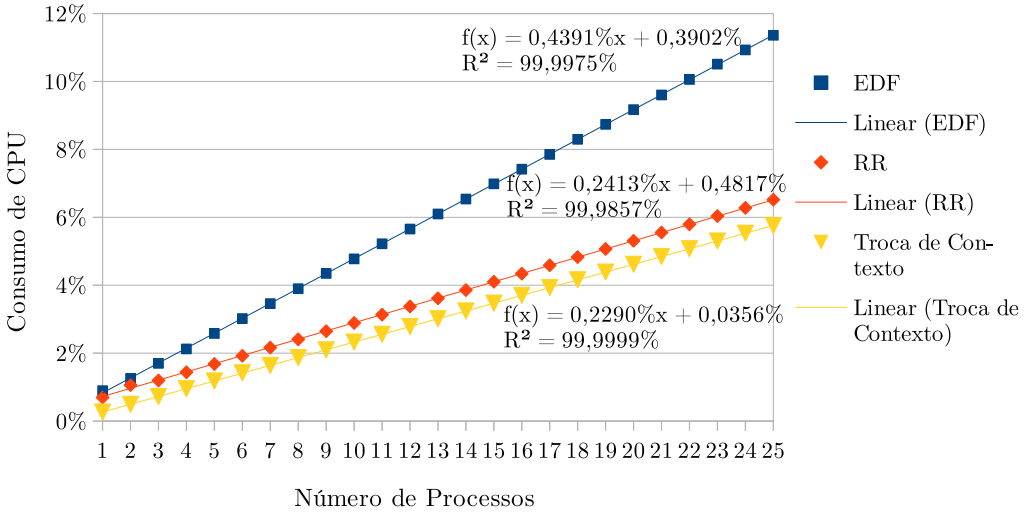
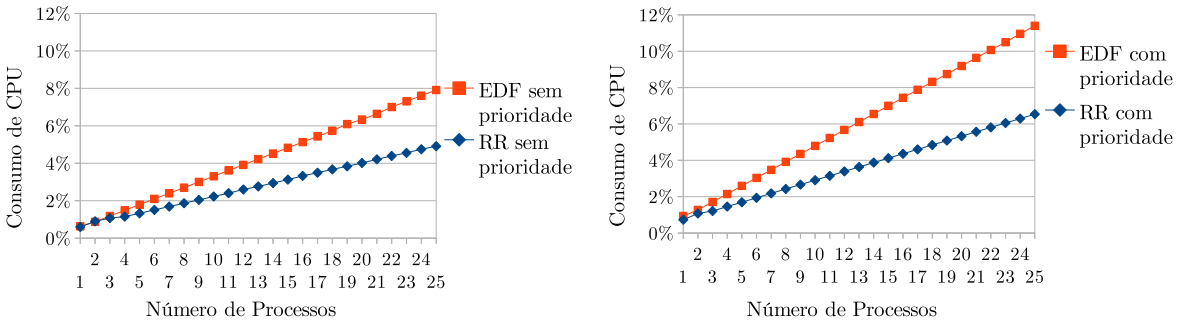


Figura 29: Comparação entre o consumo de processamentos dos escalonadores estudados

todos os processos correntes, antes de optar pelo que possui menor tempo, o EDF acaba consumindo um tempo maior que o RR, fato que se agrava a medida que a quantidade de processos aumenta, chegando a quase dobrar com 25 processos.

O gráfico na Figura 30a apresenta a variação no consumo de processamento com o aumento da quantidade de processos escalonados. Estas medidas foram tomadas com o sistema de prioridades desligado. Já nas medidas apresentadas pelo gráfico da Figura 30b, o sistema de prioridades estava habilitado.

Nota-se que as curvas de ambos os escalonadores, com o sistema de prioridade habilitado, apresentam uma inclinação maior. Isto se dá pelo consumo extra do sistema de prioridade que acarreta num acréscimo entre 0,07% e 0,14% por processo em execução. Esta variação se dá na busca pelo processo RT, que nem sempre está na mesma posição no vetor.



(a) Sistema de prioridade desabilitado (b) Sistema de prioridade habilitado

Figura 30: Variação do consumo de processador pela quantidade de processos em execução

A Figura 31 apresenta o consumo de processamento para a troca de contexto dos dois métodos de correção: Hamming e CRC. A inclinação observada é a mesma do sistema sem nenhuma correção, em azul. O acréscimo inserido pelo método de detecção CRC é, em média, de 12,18%, com um desvio padrão de 0,09%. Já para a implementação com o sistema de correção Hamming habilitado, o consumo aumenta, em média, 45,22%, com desvio padrão de 0,19%.

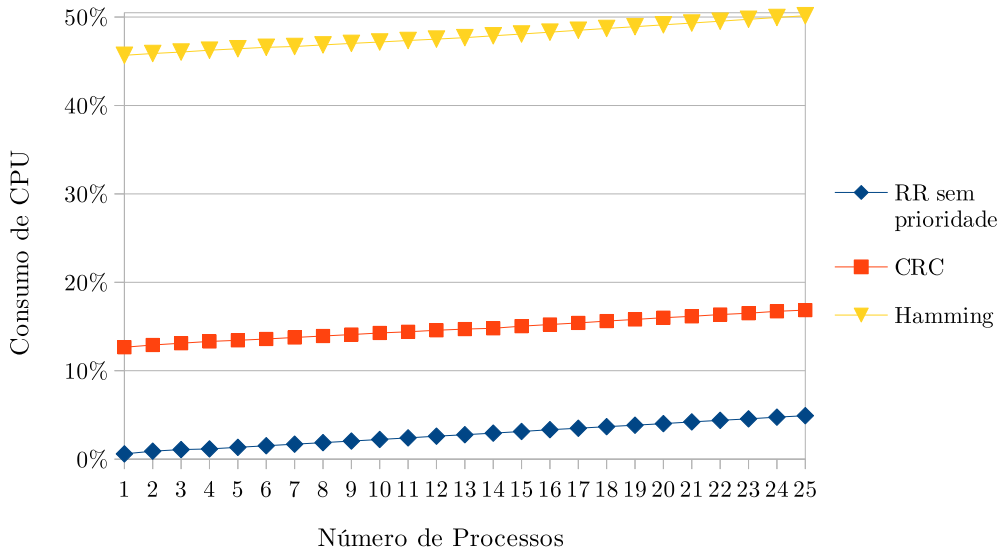
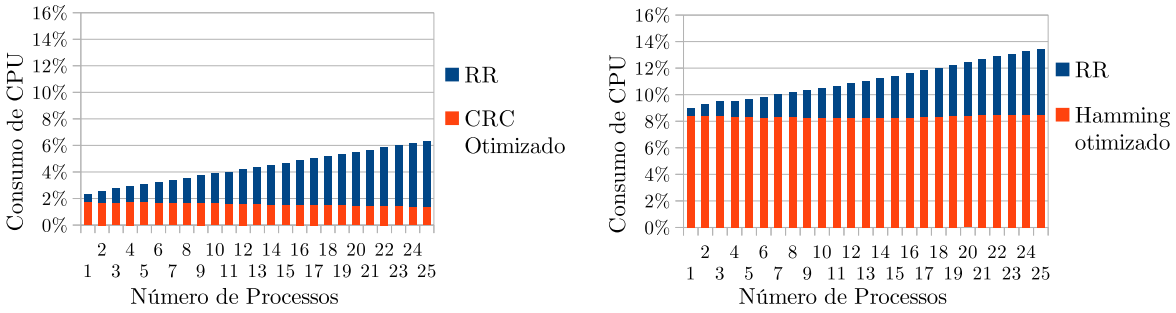


Figura 31: Comparação entre os métodos de correção

Para reduzir estes valores, optou-se por utilizar a rotina de CRC com uma *lookup table*, a fim de trocar tempo de processamento por um conjunto de valores pré-computados. Esta abordagem permitiu reduzir o consumo de processamento em quase 5 vezes, no entanto, aumentando o consumo de memória RAM em 512 bytes, o que praticamente quadruplica a necessidade mínima de memória RAM pelo sistema operacional. O gráfico da Figura 32a apresenta o consumo para a troca de contexto com o algoritmo de CRC otimizado. Pode-se notar que a sobrecarga é praticamente constante com um valor médio de 1,58% e um desvio padrão de 0,11%.

O gráfico da Figura 32b apresenta os resultados de consumo de processamento para o algoritmo de Hamming otimizado.

Na implementação do algoritmo de correção de Hamming, verificou-se que o maior consumo estava na rotina de contagem de *bits*. Optou-se por utilizar também uma *lookup table* para este mapeamento. Esta e outras melhorias conseguiram reduzir o consumo médio para 8,54%, cerca de 5 vezes menor, apresentando um desvio padrão de 0,18%. No entanto, o consumo extra de memória RAM é de apenas 16 bytes. Pode-se notar que, como o algoritmo de CRC, a sobrecarga do algoritmo de Hamming é praticamente constante, conforme o gráfico da Figura 31.



(a) CRC (b) Hamming

Figura 32: Consumo de processamento pela troca de contexto e pelos métodos de correção

A Tabela 13 apresenta os valores de utilização de processamento pelos algoritmos de correção e detecção, com e sem as *lookup table* para 1, 5, 10, 15, 20 e 25 processos. Os valores médios foram calculados com base nas 25 medidas realizadas. O consumo de processamento por parte da troca de contexto e do escalonador foi retirado da medida. Pode-se notar que os valores são praticamente constantes, sendo independentes da quantidade de processos em execução.

Tabela 13: Economia de tempo de processamento dos algoritmos de detecção e correção de erros utilizando *lookup table*

Número de processos	CRC	CRC *lookup table*	Economia	Hamming	Hamming *lookup table*	Economia
1	12,05%	1,77%	85,29%	45,08%	8,41%	81,35%
5	12,11%	1,72%	85,80%	45,09%	8,33%	81,53%
10	12,04%	1,64%	86,35%	44,95%	8,28%	81,57%
15	11,91%	1,55%	86,96%	44,95%	8,29%	81,55%
20	11,97%	1,48%	87,61%	45,10%	8,43%	81,30%
25	11,94%	1,40%	88,28%	45,24%	8,51%	81,19%
Média das 25 medidas	11,99% $\sigma = 0,07$	1,46% $\sigma = 0,09$	86,83%	45,04% $\sigma = 0,11$	8,36% $\sigma = 0,08$	81,44%

O método de correção misto é dependente da frequência de execução dos processos com prioridade RT em comparação com a frequência de execução dos processos normais. Para um sistema com 1 processo com prioridade RT pode-se calcular o consumo com base na Equação 4.3.

$$C_{Mix} = C_{Hamming} * \frac{tick}{p_{RT}} + C_{CRC} * \frac{(p_{RT} - tick)}{(p_{RT})} \tag{4.3}$$

onde: *tick* é o valor em tempo entre as interrupções para troca de contexto, p_{RT} é o período de agendamento do processo com prioridade RT, inverso de sua frequência de

execução, $C_{Hamming}$ é o consumo de um escalonador com método de correção Hamming e C_{CRC} é o consumo de um escalonador com método de detecção com CRC.

O impacto no tempo depende então de quantas vezes, num determinado período, um processo com prioridade RT é executado. O gráfico da Figura 33 apresenta duas curvas, uma calculada, baseando-se nos valores de troca de contexto com os algoritmos otimizados $C_{CRC} = 2,37$ e $C_{Hamming} = 9,01$, e a outra com valores medidos. Para as medições foi criado um processo RT com período p_{RT} com valores variando de 1 a 40 *ticks*.

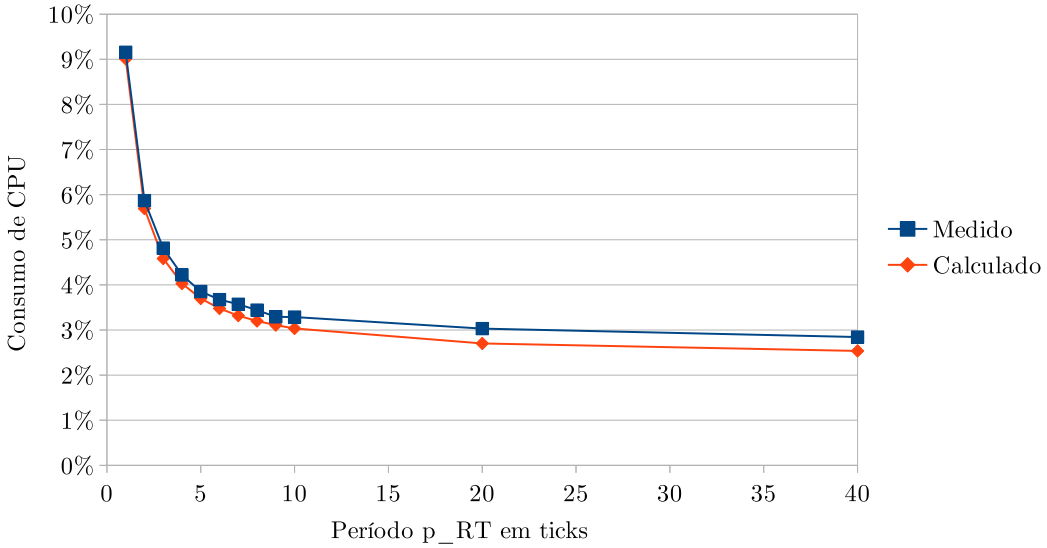


Figura 33: Avaliação dos tempos do método de correção mista

Pode-se observar que o consumo de processamento medido é pouco maior que o calculado. Isto se deve, principalmente, às operações extras para verificação de qual é a prioridade do processo em questão e na escolha adequada do método de correção ou detecção de erros.

A Figura 34 apresenta os sinais obtidos para o tempo gasto na troca de contexto (canal 2 - azul), o tempo gasto no processo *idle* (canal 1 - amarelo) e o tempo gasto no processo RT (canal 4 - verde). Neste teste foi criado apenas um processo com prioridade RT e nenhum processo com prioridade normal. O processo RT foi agendado para acontecer a cada 3 ticks de sistema (6,144 ms). Deste modo, existem três transições possíveis:

- De IDLE para RT
- De RT para IDLE
- De IDLE para IDLE

Cada troca de contexto executa duas conferências. Nas trocas que envolvem o processo RTOS e o processo IDLE, uma conferência é calculada através de Hamming e a segunda

através de CRC. Deste modo, é de se esperar que duas transições apresentem um tempo maior e apenas a terceira apresente um tempo menor. Expectativa esta que é comprovada na Figura 34.

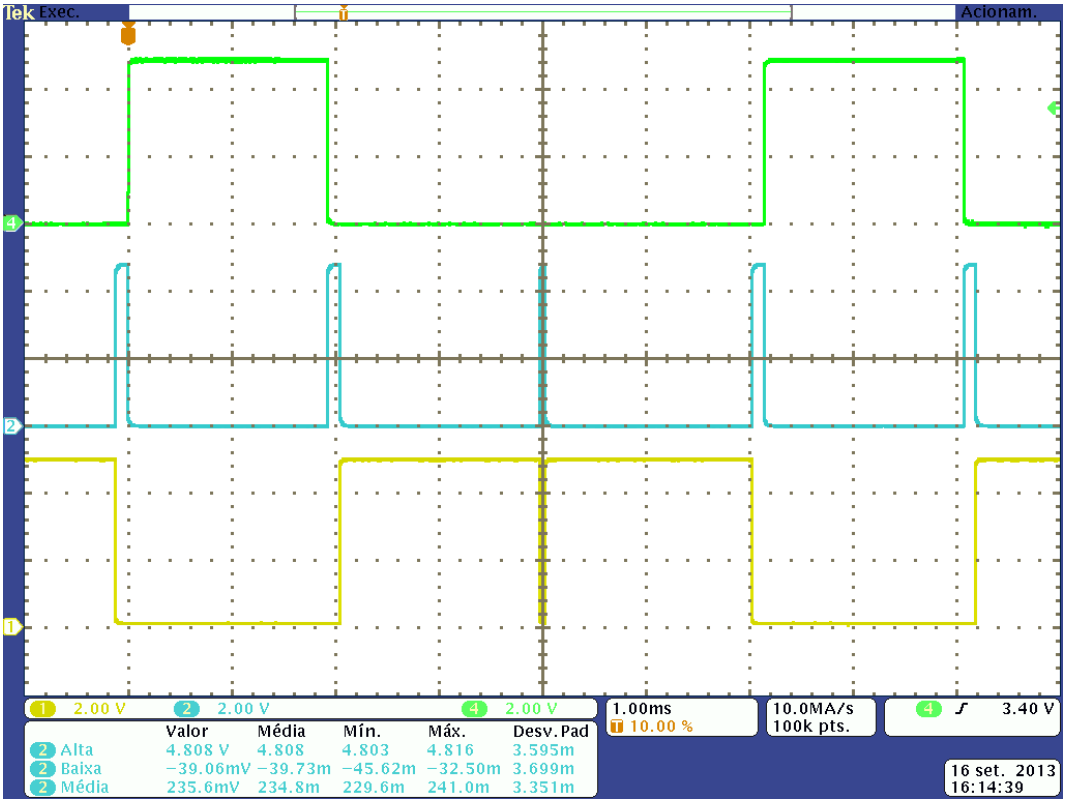


Figura 34: Sinais obtidos para modo de correção misto

O tempo disponível para execução de processos é dependente do tempo gasto na troca de contexto.

Para testar a capacidade de correção do sistema, um primeiro teste temporal foi realizado com o intuito de verificar o sistema de correção ao longo do funcionamento do equipamento. Duas placas foram mantidas em execução durante 120 horas com um processo gerando 1 bit de erro na pilha do sistema simulando um erro transitório. Todos os *bits* foram testados de forma sequencial. A primeira placa foi configurada para utilizar apenas o algoritmo de CRC e o segundo para o algoritmo de Hamming.

Com um *tick* ajustado para realizar uma troca de contexto a cada 2,048 (ms) foram executadas 210.937 trocas de contexto. Com uma pilha de 1 kB, cada um dos *bits* foi testado pelo menos 26 vezes.

Para a placa com o algoritmo de CRC todos os erros foram detectados e os processos reiniciados corretamente.

O algoritmo de correção de Hamming recuperou 100% dos erros de bit único inseridos, evitando qualquer tipo de descontinuidade no funcionamento da placa.

Um segundo teste foi realizado para se verificar o impacto que um erro em um bit pode gerar no sistema, de acordo com sua posição na pilha de um processo. Os *bits* da pilha foram alterados um a um, utilizando-se das funções implementadas pelo protocolo de comunicação serial, e o resultado foi classificado em três categorias de acordo com a gravidade do erro:

1. Nenhum erro observado, o processo continuou em funcionamento e não houve alteração no valor de saída;
2. Houve alteração não desejada nas ações de controle mas o sistema continuou em funcionamento;
3. O sistema parou de responder.

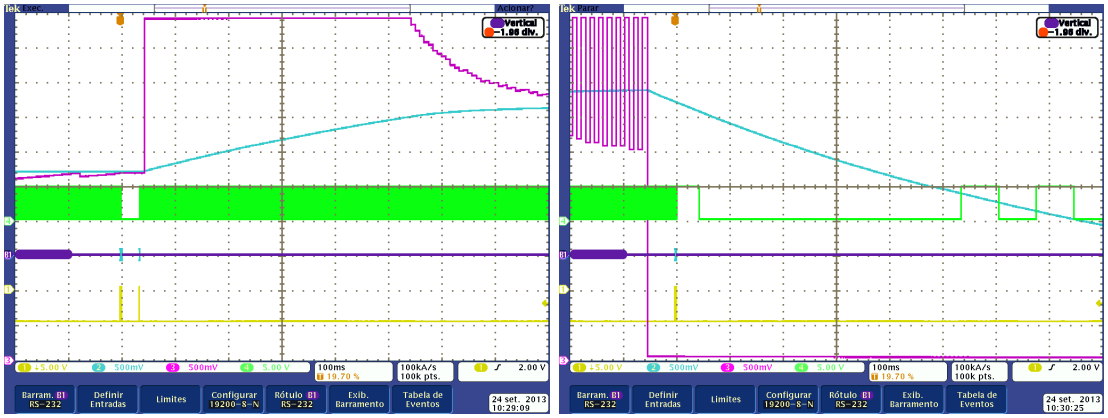
O gráfico da Figura 35 apresenta os resultados encontrados neste teste. Nota-se que nem todos os *bits* geram erros no sistema e alguns geram erros que podem ser recuperados.

Byte	Descrição	Bit							
		7	6	5	4	3	2	1	0
0	CRC (alto)	1	1	1	1	1	1	1	1
1	CRC (baixo)	1	1	1	1	1	1	1	1
2	Paginação	1	1	1	1	3	3	3	3
3	CCR	1	1	1	3	2	2	1	1
4	Acumulador B	2	2	2	2	2	2	2	2
5	Acumulador A	2	2	2	2	2	2	2	2
6	Indexador IX (alto)	2	2	2	2	2	2	2	2
7	Indexador IX (baixo)	2	2	2	2	2	2	2	2
8	Indexador IY (alto)	2	2	2	2	2	2	2	2
9	Indexador IY (baixo)	2	2	2	2	2	2	2	2
10	Contador de Programa (alto)	3	3	3	3	3	3	3	3
11	Contador de Programa (baixo)	3	3	3	3	3	3	3	3

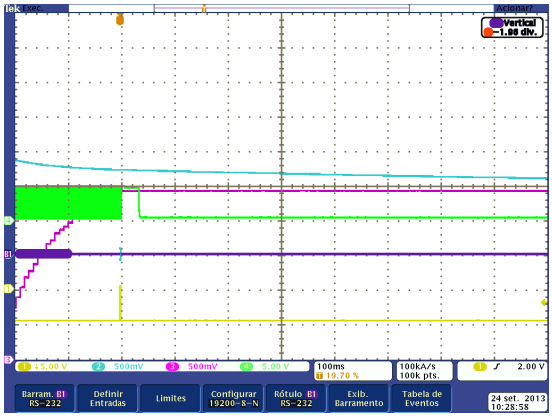
Figura 35: Impacto de um bit errado na pilha da troca de contexto numa aplicação

Nota-se que os erros mais críticos se concentram nas posições que possuem relação com o endereço de retorno do programa: os 16 *bits* do contador de programa e apenas 4 *bits* da variável de paginação. Os demais 4 *bits* não impactaram no sistema por, particularmente na arquitetura estudada, não serem implementados.

Estes erros em geral levam o sistema a um endereço errado o que pode gerar falhas sistêmicas: recuperáveis, quando o endereço ainda é válido e está na região esperada (Figura 36a), irrecuperáveis, reinicializando o sistema (Figura 36b), ou irrecuperáveis não permitindo a reinicialização do sistema (Figura 36c). As linhas verdes representam a troca de contexto, a ausência de transições nesta linha é sinal que o sistema operacional parou de responder. As linhas em amarelo indicam o momento da recepção do comando enviado via comunicação serial, representando, neste caso, o momento em que a falha foi inserida na pilha de dados.



(a) Falha com recuperação do sistema (b) Falha levando à reinicialização da placa



(c) Falha levando à estagnação da placa

Figura 36: Modos de falha das alterações no endereço de retorno

Já o registro CCR é responsável por diversas funções: habilitar as interrupções, gerenciar o modo de baixo consumo e apresentar informações referentes às operações realizadas pelo processador. Por esse motivo, nem todos os seus *bits* causam o mesmo impacto no sistema.

Cinco dos oito *bits* deste registro não causaram nenhum impacto no sistema. Destes cinco, um habilita o sistema de baixo consumo de energia (bit 7), um gerencia as interrupções externas (bit 6), não utilizadas nesta aplicação, e os outros três são responsáveis por informar os eventos de *half-carry* (bit 5), *carry/borrow* (bit 1) e complemento de dois (bit 0). Estas operações não foram utilizadas pelo processo do controlador explicando a ausência de impacto na alteração destes *bits*.

Os *bits* 2 e 3 indicam a condição de número negativo ou de número zero nos acumuladores. Por serem condições muito utilizadas na transcrição de estruturas de decisão e de repetição da linguagem C para Assembly, os impactos são perceptíveis na saída.

O bit 4 é o responsável por habilitar as interrupções de *hardware* e, se desligado por um erro de memória, inutiliza o sistema operacional por desabilitar as trocas de contexto. Este foi o único bit do registro CCR que causou um erro não recuperável pelo sistema.

Este tipo de falha causa uma paralisação do sistema operacional. No entanto, o processo corrente ainda continua em execução. Isto é visualizado na Figura 37, onde, apesar da ausência de trocas de contexto, a ação de controle na linha rosa continua a ser executada.

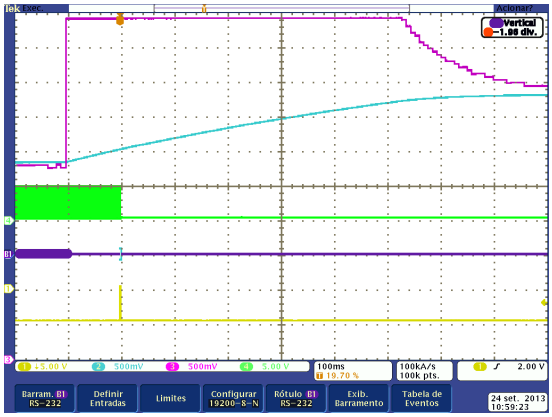
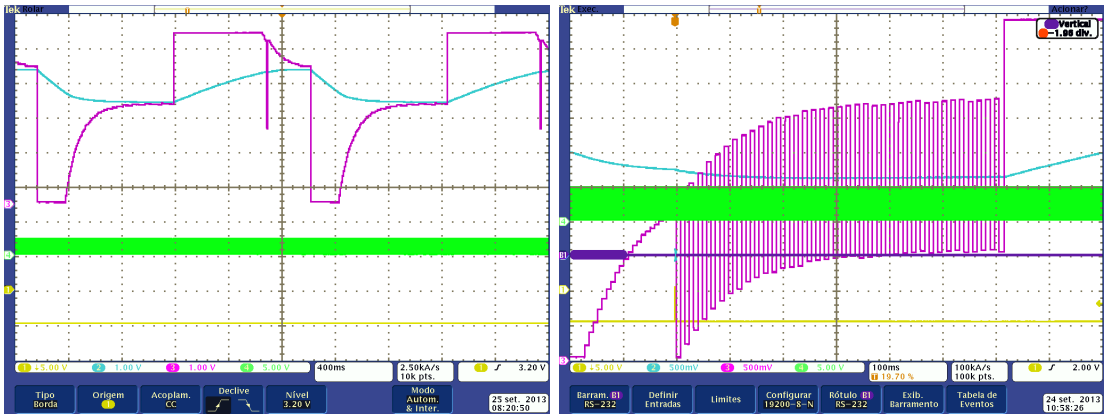


Figura 37: Impacto de um erro no bit de controle das interrupções

Os erros nos *bits* pertencentes aos acumuladores A e B, além dos indexadores IX e IY, apesar de não causarem problemas no andamento do sistema, perturbaram os valores das variáveis internas do processo de controle (Figura 38a), causando distúrbios momentâneos no sinal e até mesmo problemas de instabilidade (Figura 38b) pela alteração nos valores dos coeficientes do controlador PID.



(a) Falha nas variáveis de cálculo do valor de saída

(b) Falha nos coeficientes do controlador PID

Figura 38: Modos de falha das alterações nas variáveis armazenadas nos acumuladores e indexadores

Por fim, a ausência de impacto nos bytes reservados para o CRC era esperado, pois neste primeiro teste os *bits* não foram utilizados.

O mesmo teste foi repetido com os sistemas de correção e detecção ligados. Nenhum erro foi percebido em nenhum dos dois algoritmos. A única diferença entre as alternativas é a instabilidade transitória visualizada quando, na presença de um erro, o processo de

controle é reinicializado pelo algoritmo de CRC, perdendo os valores de suas variáveis internas.

Com o sistema de correção pelo algoritmo de Hamming em funcionamento, os processos precisam de no mínimo dois erros para que haja algum problema no sistema. A quarta coluna da Tabela 14 apresenta o tempo de funcionamento esperado para que uma certa quantidade de itens apresente alguma falha. Este valor já apresenta uma melhora significativa, no entanto, os erros precisam acontecer num intervalo de tempo menor que o período de execução do processo. A última coluna apresenta este valor considerando-se um processo com um período de 10s. Nota-se que a confiabilidade do sistema aumenta várias ordens de grandeza deste modo.

Tabela 14: Probabilidade de uma ou duas falhas em níveis sigma

Intervalo sigma	Quantidade (q) de equipamentos:	Tempo funcionamento antes que (q) equipamentos		
		tenham 1 bit errado (Horas)	tenham 2 *bits* errados (Horas)	falhem com o sistema de correção habilitado (Horas)
1σ	31,752%	208.996	626.389	2,77E+16
2σ	4,551%	25.501	184.589	3,38E+15
3σ	0,27%	1.480	41.178	1,96E+14
4σ	0,007%	38	6.502	5,08E+12
$4,5\sigma$	0,00034%	2	1.420	2,47E+11

Mesmo ocorrendo duas falhas simultâneas no intervalo entre duas execuções de um processo, o sistema ainda é capaz de reinicializá-lo. Porém esta é uma condição extremamente rara, conforme visto na Tabela 14.

4.1 Testes do controlador PID

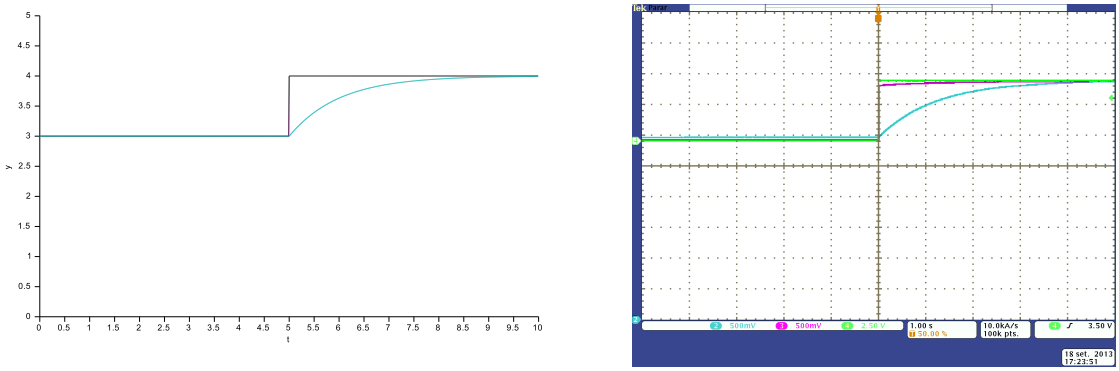
Este teste é focado na capacidade do sistema projetado de reproduzir corretamente as ações de controle mesmo sob falhas de memória e com excesso de processos para serem executados. O resultado do comportamento dinâmico destes testes não são significativos para este estudo, apenas a coerência destes resultados com as simulações é importante.

Durante os testes, o sistema de correção misto estava habilitado e erros na pilha de memória eram gerados a cada troca de contexto para estressar o sistema, tanto com o *overhead* de processamento quanto com a possibilidade de falha para o sistema de controle.

A modelagem da planta (um circuito RC série) pode ser formulada a partir da relação entre tensão e corrente no capacitor. O detalhamento se encontra no anexo D. A equação a ser inserida no simulador, portanto, é:

$$\frac{V_c(s)}{V_e(s)} = \frac{1}{s+1} \tag{4.4}$$

Foi utilizado o *software Scilab 5.4.1* para validação dos resultados obtidos pelo sistema de controle. A Figura 39a apresenta o resultado da simulação da planta em malha aberta. A Figura 39b apresenta as formas de onda obtidas no circuito real, gravadas pelo osciloscópio.



(a) Simulação (b) Teste

Figura 39: Resposta do sistema ao degrau unitário em malha aberta

Pode se notar que o tempo de acomodação é de aproximadamente 4[s], sem *overshoot*, resultado esperado para um circuito RC de primeira ordem.

Para os testes em malha fechada foi utilizado o *Xcos* (ambiente gráfico de simulação) do *Scilab* que está representado na Figura 40. Pode-se notar que, além dos componentes usuais em uma malha de controle, está inserido nesta simulação uma saturação em 0 e 5 [V]. Isto foi feito para que a simulação corresponda ao sistema real, onde as tensões no circuito não podem ultrapassar esta faixa. O bloco de somatório imediatamente antes do gerador do gráfico serve apenas para deslocar o eixo para o valor base utilizado nos testes reais, adicionando-se o valor de tensão DC utilizado no teste.

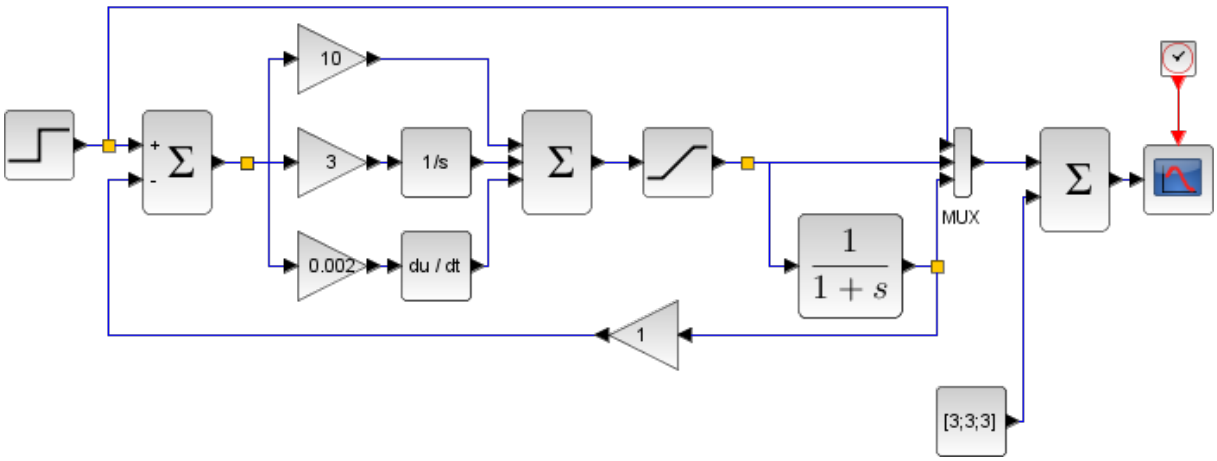
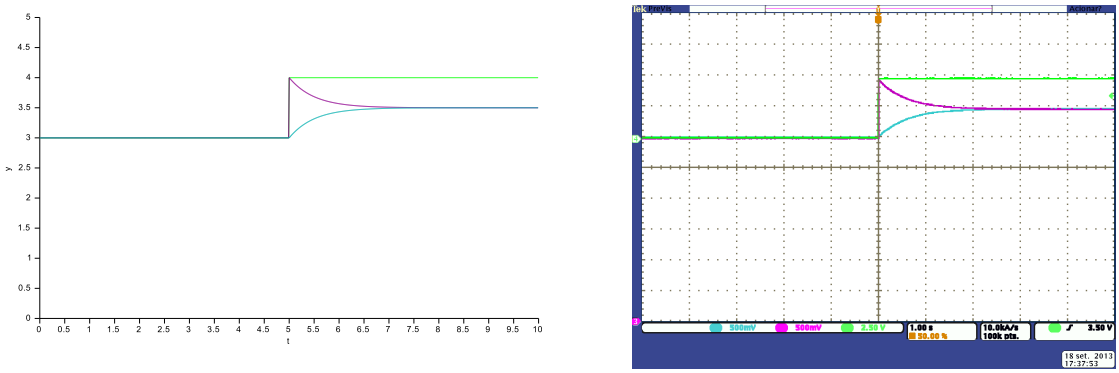


Figura 40: Diagrama de blocos da simulação do sistema de controle PID

Foram realizados 3 testes com valores arbitrários para *ki*, *kp* e *kd*. Em todos estes testes o sistema de correção proposto estava ligado. A cada troca de contexto era inserido 1 bit de erro na pilha com o intuito de verificar a estabilidade do sistema frente aos erros.

Primeiramente, testou-se o controle da planta com a ação de realimentação ligada. Espera-se um resultado melhor, com um menor tempo de acomodação para os sinais do sistema. Isto pode ser verificado na Figura 41a, que apresenta a simulação realizada. Já a Figura 41b apresenta o resultado obtido no teste prático. O erro observado em regime permanente se deve à ausência de um pólo na origem na equação de controle.

Analisando as figuras da simulação e a do teste realizado, observa-se que as duas se apresentam muito semelhantes, pois nas duas não há *overshoot* e o tempo de acomodação é de aproximadamente 1,5 [s].



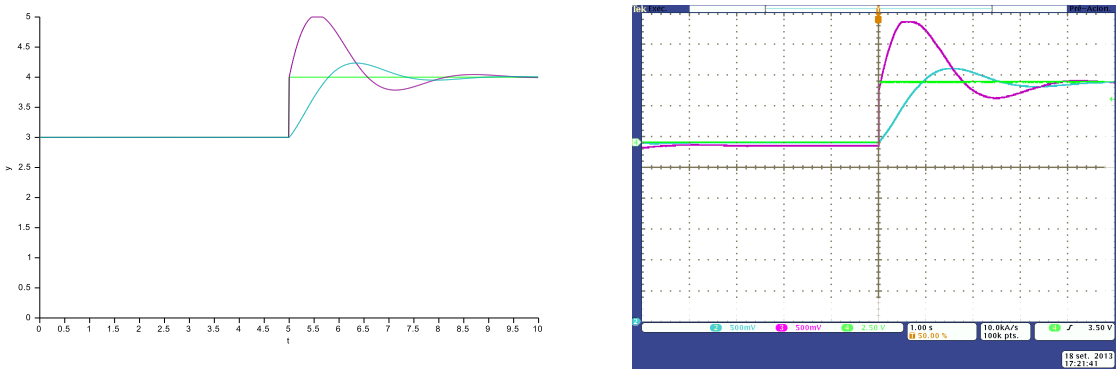
(a) Simulação (b) Teste

Figura 41: Resposta do sistema ao degrau unitário, kp = 1, ki = 0 e kd = 0

No segundo teste, foi introduzida a componente integradora. Por meio da da simulação e da resposta do sistema real, que estão nas Figuras 42a e 42b, respectivamente, observa-se novamente uma reprodução do comportamento dinâmico do sistema, muito similar ao simulado. Houve um *overshoot* de 23.5%. A ação de controle em ambos os ambientes foi ceifada pelas limitações físicas do *hardware*. O tempo de acomodação aumentou para aproximadamente 3,6 [s]. Com a inserção do integrador, e o pólo na origem, pode-se observar que o erro em regime permanente para a entrada degrau é eliminado.

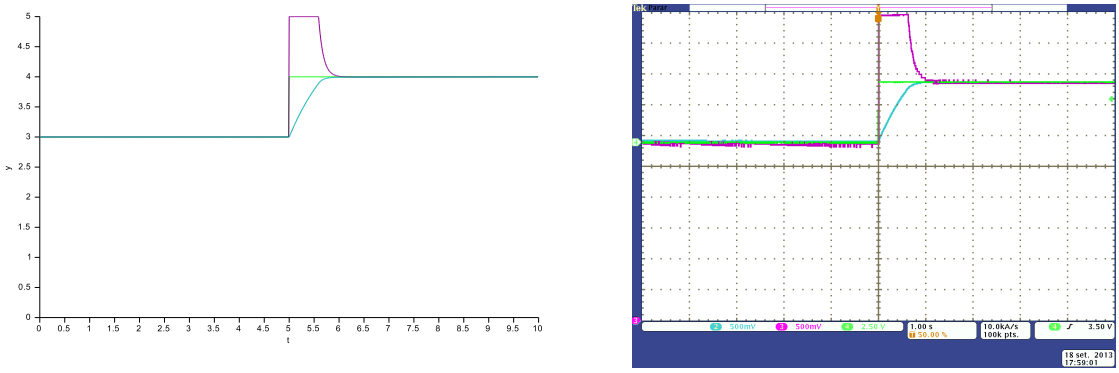
No último teste, foi adicionado o efeito do controle derivativo. Analisando as Figuras 43a e 43b, respectivamente, da simulação e da resposta do sistema real, observa-se novamente um comportamento muito similar: tempo de acomodação de 0,8 [s] sem *overshoot* na resposta.

Em nenhum dos testes do sistema de controle foi observada alguma anomalia na resposta por falha nos *bits*, inserida na pilha de dados. Todas as falhas foram corrigidas corretamente pelo algoritmo de Hamming implementado. Os demais processos, apesar de serem protegidos apenas pelo CRC, também não apresentaram problemas, o que era esperado já que não desempenhavam funções críticas.



(a) Simulação (b) Teste

Figura 42: Resposta do sistema ao degrau unitário, kp = 1, ki = 5 e kd = 0



(a) Simulação (b) Teste

Figura 43: Resposta do sistema ao degrau unitário, kp = 10, ki = 3 e kd = 0,002

Observando-se as simulações realizadas percebe-se que existe uma identidade entre as curvas esperadas e as curvas reais. O sistema de controle se comportou de modo idêntico ao previsto pela teoria. Em todos estes testes não foi observado nenhum tipo de atraso na execução do processo RT, comprovando a capacidade do sistema de manter a execução mesmo com o *overhead* inserido pelo sistema de correção.

Observa-se que mesmo sob falhas simuladas a cada troca de contexto, pior situação em questão de sobrecarga de processamento, o controlador PID implementado continuou funcionando sem os erros ou alterações apresentadas na Figura 35. Isto permite que sistemas críticos como controle de temperatura de incubadoras neonatais, freios ABS ou reguladores de pressão de caldeiras possam operar por mais tempo mesmo com suas memórias apresentando algum tipo de falha.

5 Conclusão

A utilização dos métodos de correção e detecção de erros, na proteção de regiões de memória críticas para a troca de contexto, funcionou conforme o esperado, evitando os erros e protegendo a continuidade da execução do sistema.

A proteção adicionada por estes métodos permite que um sistema microprocessado aumente sua confiabilidade frente a erros em *bits* de memória ou falhas como *stack overflow*. Mesmos nos testes de longa duração (120 horas), com erros sendo simulados a cada troca de contexto, com o sistema de detecção e correção habilitado, não houve problema observado.

Os testes realizados com o sistema de controle demonstraram que, na ausência de qualquer sistema de detecção ou correção, a troca do valor de um único bit pode paralisar todo o sistema.

O consumo extra de memória, quando habilitado o sistema de correção e/ou detecção de erros, é pequeno, de 669 bytes, para memória não volátil (código) e de apenas 5 bytes, para memória volátil (RAM). Estes valores tornam a técnica implementável na maioria dos microcontroladores de baixo custo. A sobrecarga dos métodos otimizados se deve quase exclusivamente às tabelas, acrescendo um total de 544 bytes, de RAM ou de memória não volátil.

Quanto ao consumo de processamento, o algoritmo de detecção aumentou o tempo necessário para realizar uma troca de contexto. Quando observado o sistema original, sem nenhum tipo de correção ou detecção de erros, as trocas de contexto eram responsáveis por um consumo de 0,60% do tempo de processamento disponível. A adição de um sistema de detecção de erros utilizando o algoritmo CCITT-CRC16 elevou este número para 12,65%. Quando considerado o algoritmo de correção de erros de Hamming, o aumento é da ordem de 80 vezes, chegando a consumir quase metade do tempo disponível para processamento, 49,39%. Estes valores, principalmente o último, poderiam inviabilizar o uso desta técnica em processadores de baixo custo, que apresentam também baixa capacidade de processamento.

Estes valores de consumo, no entanto, podem ser reduzidos em mais de 80% com o uso de técnicas de otimização como *lookup-tables*. Aplicando-se estas técnicas nos algoritmos utilizados, o sistema de correção por CRC passa a exigir apenas 1,46% de processamento. No entanto a *lookup-table* utilizada impacta no consumo de memória em cerca de 512 bytes.

Para o o algoritmo de Hamming, no entanto, o consumo de memória adicional é de apenas 16 bytes, mantendo mesmo assim uma redução da ordem de 80%, atingindo o

patamar de apenas 8,36%. Isto viabiliza o uso destas técnicas em sistemas com poucos recursos computacionais. O algoritmo de Hamming otimizado se torna uma boa alternativa, pois apresentar um consumo menor que o algoritmo de CRC sem otimização sem gerar o gasto extra de memória do algoritmo de CRC otimizado.

A proposta de utilização de um modelo misto, de correção para processos de prioridade RT, e detecção para os demais processos, se mostrou muito interessante. Mantém-se um consumo próximo ao de um sistema com capacidade de detecção de erros (CRC) ao mesmo tempo que se garante a confiabilidade trazida pelo método de correção de erros (Hamming) para os processos mais críticos.

Borchert, Schirmeier e Spinczyk (2013) apresentam uma solução similar à desenvolvida neste trabalho. Eles fazem uso das mesmas técnicas de detecção e correção (CRC e Hamming) na proteção das estruturas de dados pertencentes às regiões críticas do *kernel*. A abordagem utilizada, no entanto, é inviável para sistemas embarcados de baixo custo, visto que esta exige a utilização de uma linguagem orientada à objeto com capacidade de programação orientada à aspecto. A solução apresentada, fazendo uso apenas de recursos padronizados na linguagem C, permite a sua aplicabilidade em praticamente qualquer plataforma que tenha um compilador C.

Um ponto crítico em sistemas embarcados é a necessidade de tempo real para alguns processos. Segundo os testes realizados, não houve problemas em garantir o funcionamento do sistema de controle com o sistema de detecção e correção habilitados, mesmo simulando falhas na pilha a cada troca de contexto. Os resultados das ações de controle se mostraram iguais as simulações realizadas.

A questão de um invasor, no entanto, pode ser mais grave. Caso este possua conhecimento sobre o algoritmo utilizado no código de correção a abordagem do modo apresentado será ineficiente. Uma solução é a utilização um valor aleatório que possa ser atualizado automaticamente e utilizado como *seed* na pilha de dados. Como as trocas de contexto acontecem em frequências relativamente altas, o sistema proposto, com a troca constante da *seed*, inviabilizaria diversos tipos de ataques voltados à re-escrita da memória de pilha.

As alternativas para a geração de códigos de checagem, que sejam de difícil quebra por parte do invasor, como algoritmos assimétricos, são caras do ponto de vista computacional. Deste modo, é possível que sua implementação em sistemas de baixo custo seja inviável, tornando a técnica apresentada como a única alternativa viável para este cenário.

5.1 Trabalhos futuros

Tendo em vista a tendência no aumento de dispositivos móveis e a preocupação com a redução de consumo, vislumbra-se a criação de rotinas ainda mais otimizadas para o cálculo dos bytes de correção e detecção dos erros. Isto permitiria que o sistema consumisse

menos recursos computacionais e permanecesse mais tempo em modos de baixo consumo de energia.

Uma segunda frente de trabalho seria a adaptação do código para outras arquiteturas e/ou outros sistemas operacionais. O FreeRTOS é um dos candidatos naturais, pois tem todo seu código disponibilizado gratuitamente devido a licença GPL. Além disso, sua ampla utilização pode ser um bom ponto de disseminação da metodologia proposta.

Por fim a criação de um circuito dedicado que implemente a metodologia apresentada neste trabalho pode ser uma alternativa viável principalmente em sistemas baseados em FPGA e que façam uso de *softcores*. A prototipagem de circuitos de hardware utilizando linguagens como VHDL é simples e retorna sistemas bastante otimizados principalmente no que tange as questões de velocidade. Isto aliado a alta capacidade de sintetizar lógicas combinacionais torna esta alternativa uma solução bastante interessante para implementar os algoritmos de detecção e correção de erros.

5.2 Dificuldades

O desenvolvimento da rotina de preempção foi um dos pontos críticos na implementação da metodologia. Esta rotina é executada por uma interrupção de hardware de modo que vários problemas se apresentam reunidos:

- a falta de acesso as variáveis não globais do sistema operacional;
- uma pilha temporária devido ao salvamento dos registros da CPU, impedindo o acesso imediato à pilha do processo;
- qualquer utilização de variáveis temporárias na rotina de interrupção insere dados extras na pilha atrapalhando sua manipulação;
- a necessidade de baixo consumo, dado que este é um evento recorrente;
- toda manipulação da pilha só pode ser feita em *assembly*, e a integração desta com as linguagens de alto nível não é padronizada e altamente dependente do compilador e do *hardware* utilizado.

Estes fatores impactaram no desenvolvimento do projeto no tempo dispendido para o ajuste e desenvolvimento das rotinas de baixo nível.

A implementação do algoritmo de *hamming* apresentou alguns problemas principalmente pela falta de recursos do processador em realizar operações bit à bit. A abordagem de utilizar apenas os bytes completos viabilizou o desenvolvimento para arquiteturas de baixo nível, no entanto uma comparação com o algoritmo original é interessante para definir os ganhos obtidos.

Para a realização dos testes, uma das primeiras dificuldades encontradas foi o modo de medição dos tempos gastos por cada porção da aplicação. A inserção de instrumentação no código se mostrou inviável pela carga adicional. Isto também faz com que a placa execute uma quantidade de código não necessária para a aplicação, podendo adulterar os resultados. A solução utilizada, de medição por valor médio, insere uma quantidade mínima de instruções apresentando um bom resultado.

A segunda dificuldade na realização dos testes foi o desenvolvimento de uma interface remota para controlar o número de processos em execução, criando-os e removendo-os com o sistema em funcionamento. Esta interface permite ainda a inserção de erros em bits da pilha de memória. A dificuldade se encontrou em desenvolver um protocolo que fosse simples e não influenciável pelos erros simulados na pilha. A utilização de um protocolo com capacidade de detecção de erros via CRC foi suficiente.

Referências

ALMEIDA, R. M. A. de; FERREIRA, L. H. d. C.; VALÉRIO, C. H. Microkernel development for embedded systems. *Journal of Software Engineering and Applications*, 2013. JSEA, v. 6, n. 1, p. 20–28, 2013.

AMD. *Enhanced Virus Protection*. 2013. Disponível em: <http://www.amd.com/us-/products/technologies/enhanced-virus-protection/Pages/enhanced-virus-protection-.aspx>.

ARM. *ARM11 MPCore Processor Technical Reference Manual*. 2013. Disponível em: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/CACHFICI-.html>.

ATMEL. *Trusted Platform Module*. 2013. Disponível em: <http://www.atmel.com-/products/other/embedded/default.aspx>.

AUTRAN, J. et al. Soft-errors induced by terrestrial neutrons and natural alpha-particle emitters in advanced memory circuits at ground level. *Microelectronics Reliability*, 2010. v. 50, n. 9–11, p. 1822 – 1831, 2010. ISSN 0026-2714. ¡ce:title¿21st European Symposium on the Reliability of Electron Devices, Failure Physics and Analysis¡/ce:title¿. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0026271410003069>.

AUTRAN, J.-L. et al. Numerical simulation - from theory to industry. In: _____. [S.l.]: Dr. Mykhaylo Andriychuk (editor), 2012. cap. Soft-Error Rate of Advanced SRAM Memories: Modeling and Monte Carlo Simulation. ISBN 978-953-51-0749-1. Http://www.intechopen.com/books/numerical-simulation-from-theory-to-industry/soft-error-rate-of-advanced-sram-memories-modeling-and-monte-carlo-simulation.

BARR, M. *Programming embedded systems in C and C++*. [S.l.]: O'Reilly Media, Inc., 1999.

BARROS, E.; CAVALCANTE, S. Introdução aos sistemas embarcados. *Universidade Federal de Pernambuco – UFPE*, 2002. 2002.

BAUMANN, R. C.; SMITH, E. B. Neutron-induced boron fission as a major source of soft errors in deep submicron sram devices. In: IEEE. *Reliability Physics Symposium, 2000. Proceedings. 38th Annual 2000 IEEE International*. [S.l.], 2000. p. 152–157.

BOMMENA, S. *Application note 1148 - Cyclic Redundancy Check (CRC)*. [S.l.], 2008. Http://ww1.microchip.com/downloads/en/AppNotes/01148a.pdf.

BORCHERT, C.; SCHIRMEIER, H.; SPINCZYK, O. Generative software-based memory error detection and correction for operating system data structures. In: IEEE. *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. [S.l.], 2013. p. 1–12.

BRAUN, F.; WALDVOGEL, M. Fast incremental crc updates for ip over atm networks. In: IEEE. *High Performance Switching and Routing, 2001 IEEE Workshop on*. [S.l.], 2001. p. 48–52.

BRAUN, M. et al. Parallel crc computation in fpgas. In: *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*. [S.l.]: Springer, 1996. p. 156–165.

BREYFOGLE, F. W. *Implementing Six Sigma: smarter solutions using statistical methods*. [S.l.]: Wiley. com, 2003.

CATALDO, A. Sram soft errors cause hard network problems. *Electronic Engineering Times*, 2001. CMP Media LLC, p. 1–2, Agosto 2001. Disponível em: <http://www-.eetimes.com/story/OEG20010817S0073>.

CHANDRA, V.; AITKEN, R. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale cmos. In: IEEE. *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*. [S.l.], 2008. p. 114–122.

CHAUDHARI, A.; PARK, J.; ABRAHAM, J. A framework for low overhead hardware based runtime control flow error detection and recovery. In: IEEE. *VLSI Test Symposium (VTS), 2013 IEEE 31st*. [S.l.], 2013. p. 1–6.

CHEN, P. M. et al. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 1994. ACM, v. 26, n. 2, p. 145–185, 1994.

CORPORATION, C. S. *16-Bit CRC Generator Datasheet CRC16 V 3.2*. [S.l.], 2011. Http://www.cypress.com/?docID=34119.

COWAN, C. et al. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In: IEEE. *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*. [S.l.], 2000. v. 2, p. 119–129.

DENARDIN, G.; BARRIQUELLO, C. H. *BRTOS - Brazilian Real-Time Operating System*. 2013. Disponível em: <https://code.google.com/p/brtos/>.

DEVICES, A. M. *Software Optimization Guide for AMD64 Processors*. [S.l.], 2005. Http://support.amd.com/TechDocs/25112.PDF.

ENGINEERS, R. T. *FreeRTOS*. 2013. Disponível em: <http://www.freertos.org>.

FREESCALE. *MC9S12DT256 Device User Guide 9S12DT256DGV3 V03.07*. [S.l.], 2005. Disponível em: <http://cache.freescale.com/files/microcontrollers/doc/data_sheet-/9S12DT256_ZIP.zip>.

FREESCALE. *National Institute of Standards (NIST) Certifications*. 2013. Disponível em: <http://www.freescale.com/zh-Hans/webapp/sps/site/overview-.jsp?code=NETWORK_SECURITY_FIPS>.

GANSSLE, J. Microcontroller c compilers. *Electronic Engineering Times*, 1990. November 1990.

GEOFF, T. *Six Sigma: SPC and TQM in manufacturing and services*. [S.l.]: Gower Publishing, Ltd., 2001.

GIUFFRIDA, C.; KUIJSTEN, A.; TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In: *Proceedings of the 21st USENIX conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2012. (Security'12), p. 40–40. Disponível em: <http://dl.acm.org/citation.cfm?id=2362793.2362833>.

GROUP, I. . W. et al. Ieee standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*, 2009. p. C1, 2009.

HALLINAN, C. *Embedded Linux primer: a practical, real-world approach*. [S.l.]: Pearson Education India, 2007.

HAMMING, R. Error detecting and error correcting codes. *Syst. Tech. J.*, 1950. v. 29, p. 147–160, 1950.

IBE, E. et al. Impact of scaling on neutron-induced soft error in srams from a 250 nm to a 22 nm design rule. *Electron Devices, IEEE Transactions on*, 2010. IEEE, v. 57, n. 7, p. 1527–1538, 2010.

IEC, . T. C. et al. *IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic (E/E/PE) Safety Related Systems, Part 3: Software Requirements*. [S.l.]: IEC, Geneva, Swiss, 1998.

INFINEON. *Infineon's Chip Card and Security ICs*. 2013. Disponível em: <http://www-.infineon.com/cms/en/product/applications/chip-card-and-security/index.html>.

INTEL. *Execute Disable Bit and Enterprise Security*. 2013. Disponível em: <http:/-/www.intel.com/cd/business/enterprise/emea/eng/202162.htm>.

IRANI, D. et al. Reverse social engineering attacks in online social networks. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. [S.l.]: Springer, 2011. p. 55–74.

J., A.; V., B. *SDPOS - Small Devices Portable Operating System*. [S.l.], 2013. Disponível em: <http://www.sdpos.org/documentation.html>.

JEON, W. et al. A practical analysis of smartphone security. In: *Human Interface and the Management of Information. Interacting with Information*. [S.l.]: Springer, 2011. p. 311–320.

KAI, T.; XIN, X.; GUO, C. The secure boot of embedded system based on mobile trusted module. In: IEEE. *Intelligent System Design and Engineering Application (ISDEA), 2012 Second International Conference on*. [S.l.], 2012. p. 1331–1334.

KALMAN, A. E. *AN-2, Understanding Changes in Salvo Code Size for Different PICmicro Devices*. Pumpkin, Inc, 2001. Disponível em: <http://www.pumpkininc.com-/content/doc/appnote/an-2.pdf>.

KERMANI, M. M. et al. Emerging frontiers in embedded security. In: *Proceedings of the 2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*. Washington, DC, USA: IEEE Computer Society, 2013. (VLSID '13), p. 203–208. ISBN 978-0-7695-4889-0. Disponível em: <http://dx.doi-.org/10.1109/VLSID.2013.222>.

KOOPMAN, P. Embedded system security. *Computer*, 2004. IEEE, v. 37, n. 7, p. 95–97, 2004.

KOOPMAN, P.; CHAKRAVARTY, T. Cyclic redundancy code (crc) polynomial selection for embedded networks. In: IEEE. *Dependable Systems and Networks, 2004 International Conference on*. [S.l.], 2004. p. 145–154.

KOSCHER, K. et al. Experimental security analysis of a modern automobile. In: IEEE. *Security and Privacy (SP), 2010 IEEE Symposium on*. [S.l.], 2010. p. 447–462.

KUROSE, J. F.; ROSS, K. W. *Computer networking*. [S.l.]: Pearson Education, 2012.

LANGNER, R. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security and Privacy*, 2011. IEEE Educational Activities Department, Piscataway, NJ, USA, v. 9, n. 3, p. 49–51, maio 2011. ISSN 1540-7993. Disponível em: <http://dx.doi.org/10.1109/MSP.2011.67>.

LEMAY, M.; GUNTER, C. A. Cumulative attestation kernels for embedded systems. *Smart Grid, IEEE Transactions on*, 2012. IEEE, v. 3, n. 2, p. 744–760, 2012.

LEUNG, W.; HSU, F.-C.; JONES, M.-E. The ideal soc memory: 1t-sram¡ sup¿ tm¡/sup¿. In: IEEE. *ASIC/SOC Conference, 2000. Proceedings. 13th Annual IEEE International*. [S.l.], 2000. p. 32–36.

LI, H. et al. A model for soft errors in the subthreshold cmos inverter. In: *Proceedings of Workshop on System Effects of Logic Soft Errors*. [S.l.: s.n.], 2006.

LI, X. et al. A realistic evaluation of memory hardware errors and software system susceptibility. In: USENIX ASSOCIATION. *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. [S.l.], 2010. p. 6–6.

MARWEDEL, P. *Embedded System Design*. [S.l.]: Springer, 2006.

MASSA, A. J. *Embedded software development with eCos*. [S.l.]: Prentice Hall Professional, 2003.

MCGRATH, D. *Renesas still dominates MCU market*. EE Times, 2012. Disponível em: <http://www.eetasia.com/ART_8800663707_1034362_NT_3211fcb4.HTM>.

MICRIUM. *uC/OS-II Real Time Kernel*. 2013. Disponível em: <http://micrium.com-/rtos/ucosii/overview/>.

MICROCHIP. *Microchip's SMSC Embedded Security Offerings*. 2013. Disponível em: <http://www.microchip.com/pagehandler/en-us/technology/embeddedsecurity-/technology/trustspan.html>.

MIPS. *SmartMIPS ASE*. 2013.

MISRA, M. I. S. R. A. et al. Misra-c 2004: Guidelines for the use of the c language in critical systems. *ISBN 0*, 2004. v. 9524156, n. 2, p. 3, 2004.

NEWSHAM, T. *Format string attacks*. 2001. Disponível em: <http://hackerproof.org-/technotes/format/formatstring.pdf>.

NEWSOME, J.; SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005. Internet Society, 2005.

PATTABIRAMAN, K.; GROVER, V.; ZORN, B. G. Samurai: protecting critical data in unsafe languages. *ACM SIGOPS Operating Systems Review*, 2008. ACM, v. 42, n. 4, p. 219–232, 2008.

PATTERSON, D. A.; GIBSON, G.; KATZ, R. H. *A case for redundant arrays of inexpensive disks (RAID)*. [S.l.]: ACM, 1988.

PAXTEAM. *PaX - kernel self-protection*. 2012. Disponível em: <http://pax.grsecurity-.net/docs/PaXTeam-H2HC12-PaX-kernel-self-protection.pdf>.

PEEK, J. Complexity analysis of new multi level feedback queue scheduler. *American Journal of Computing and Computation*, 2013. v. 3, n. 2, p. 14–28, 2013.

RAO, M. P. et al. Development of scheduler for real time and embedded system domain. In: IEEE. *Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on*. [S.l.], 2008. p. 1–6.

RAO, M. P. et al. A simplified study of scheduler for real time and embedded system domain. *Georgian Electronic Scientific Journal: Computer Science and Telecommunications*, 2009. n. 5(22), p. 60–73, 2009.

RAVI, S. et al. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems (TECS)*, 2004. ACM, v. 3, n. 3, p. 461–491, 2004.

RAY, J.; KOOPMAN, P. Efficient high hamming distance crcs for embedded networks. In: IEEE. *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. [S.l.], 2006. p. 3–12.

RENAUX, D. P. B.; BRAGA, A. S.; KAWAMURA, A. Perf3: Um ambiente para avaliação temporal de sistemas em tempo real. In: *II Workshop de Sistemas em Tempo Real*. [S.l.: s.n.], 1999.

RENESAS. *AE-5 secure MCU*. 2013. Disponível em: <http://www.renesas.com-/products/smartcard_new/ae5/index.jsp>.

RIVER, W. *Wind River VxWorks Platforms 6.9*. [S.l.], 2013. Disponível em: <http://www.windriver.com/products/product-notes/PN_VE_6_9_Platform_0311.pdf>.

ROEMER, R. et al. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 2012. ACM, New York, NY, USA, v. 15, n. 1, p. 2:1–2:34, mar. 2012. ISSN 1094-9224. Disponível em: <http://doi.acm.org/10.1145/2133375-.2133377>.

SCHROEDER, B.; PINHEIRO, E.; WEBER, W.-D. Dram errors in the wild: a large-scale field study. In: *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2009. (SIGMETRICS '09), p. 193–204. ISBN 978-1-60558-511-6. Disponível em: <http://doi.acm.org/10.1145/1555349.1555372>.

SEMATECH. *Semiconductor Device Reliability Failure Models*. 2000. Disponível em: <http://www.sematech.org/docubase/document/3955axfr.pdf>.

SEMICONDUCTORS, N. *74F401 CRC Generator/Checker Datasheet*. [S.l.], 1995. Http://pdf.datasheetcatalog.com/datasheet/nationalsemiconductor/DS009534.PDF.

SESHADRI, A. et al. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: ACM. *ACM SIGOPS Operating Systems Review*. [S.l.], 2007. v. 41, n. 6, p. 335–350.

SIDDHA, S.; PALLIPADI, V. Getting maximum mileage out of tickless. In: *Linux Symposium*. [S.l.: s.n.], 2007. v. 2, p. 201–207.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating system concepts*. [S.l.]: J. Wiley & Sons, 2009.

STALLINGS, W. *Operating Systems: Internals and Design Principles, 6/E*. [S.l.]: Pearson Education, 2009.

STALLINGS, W. *Arquitetura e organização de computadores*. [S.l.]: Pearson Education do Brasil, 2010.

STANDARD, I. Iso 11898, 1993. *Road vehicles–interchange of digital information–Controller Area Network (CAN) for high-speed communication*, 1993. 1993.

STMICROELECTRONICS. *Error Correction Code in Single Level Cell NAND Flash Memmories*. [S.l.], 2004. Http://www.datasheetarchive.com/AN1823-datasheet.html.

STUDNIA, I. et al. Survey on security threats and protection mechanisms in embedded automotive networks. In: IEEE. *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*. [S.l.], 2013. p. 1–12.

TANENBAUM, A. S.; HERDER, J. N.; BOS, H. Can we make operating systems reliable and secure? *Computer*, 2006. IEEE, v. 39, n. 5, p. 44–51, 2006.

(US), N. M. E. A. *NMEA 0183–Standard for Interfacing Marine Electronic Devices*. [S.l.]: NMEA, 2002.

VAKULENKO, S. *uOS Operating System for Embedded Applications*. 2011. Disponível em: <https://code.google.com/p/uos-embedded/wiki/about>.

VEN, A. van de. New security enhancements in red hat enterprise linux v. 3, update 3. *Red Hat, August*, 2004. 2004.

VENDA, P. *PaX performance impact*. 2005. Disponível em: <http://www.pjvenda.net-/linux/doc/pax-performance/>.

WANG, F.; AGRAWAL, V. D. Soft error rate determination for nanometer cmos vlsi logic. In: IEEE. *System Theory, 2008. SSST 2008. 40th Southeastern Symposium on*. [S.l.], 2008. p. 324–328.

WANG, Z. et al. Countering kernel rootkits with lightweight hook protection. In: *Proceedings of the 16th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2009. (CCS '09), p. 545–554. ISBN 978-1-60558-894-0. Disponível em: <http://doi.acm.org/10.1145/1653662.1653728>.

WHITE, M.; BERNSTEIN, J. B. Microelectronics reliability: physics-of-failure based modeling and lifetime evaluation. 2008. Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2008., 2008.

WIKIPEDIA. *Stack buffer overflow*. 2013. Disponível em: <http://en.wikipedia.org-/wiki/Stack_buffer_overflow>.

WILLIAMS, R. A painless guide to crc error detection algorithms - chap.4: Polynomical arithmetic. *Internet publication, August*, 1993. 1993. Disponível em: <http://ceng2.ktu.edu.tr/˜cevhers/ders_materyal/bil311_bilgisayar_mimarisi-/supplementary_docs/crc_algorithms.pdf>.

WULF, W. et al. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 1974. ACM, v. 17, n. 6, p. 337–345, 1974.

WYGLINSKI, A. M. et al. Security of autonomous systems employing embedded computing and sensors. *Micro, IEEE*, 2013. IEEE, v. 33, n. 1, p. 80–86, 2013.

XU, H.; CHAPIN, S. J. Improving address space randomization with a dynamic offset randomization technique. In: ACM. *Proceedings of the 2006 ACM symposium on Applied computing*. [S.l.], 2006. p. 384–391.

XU, J.; KALBARCZYK, Z.; IYER, R. K. Transparent runtime randomization for security. In: IEEE. *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*. [S.l.], 2003. p. 260–269.

YIM, K. S. et al. Hauberk: Lightweight silent data corruption error detector for gpgpu. In: IEEE. *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. [S.l.], 2011. p. 287–300.

Zawoad, S.; Hasan, R. The Enemy Within: The Emerging Threats to Healthcare from Malicious Mobile Devices. *ArXiv e-prints*, 2012. out. 2012.

ZIEGLER, J. F. et al. Ibm experiments in soft fails in computer electronics (1978–1994). *IBM J. Res. Dev.*, 1996. IBM Corp., Riverton, NJ, USA, v. 40, n. 1, p. 3–18, jan. 1996. ISSN 0018-8646. Disponível em: <http://dx.doi.org/10.1147/rd.401.0003>.

ZIEGLER, J. F.; PUCHNER, H. *SER–history, Trends and Challenges: A Guide for Designing with Memory ICs*. [S.l.]: Cypress, 2004.

ANEXO A – Controladora

A controladora foi projetada de modo que as requisições da aplicação ou do *kernel* pudessem passar diretamente para os *drivers* com o mínimo de *overhead*. A estrutura básica da controladora desenvolvida pode ser vista na Figura 44.

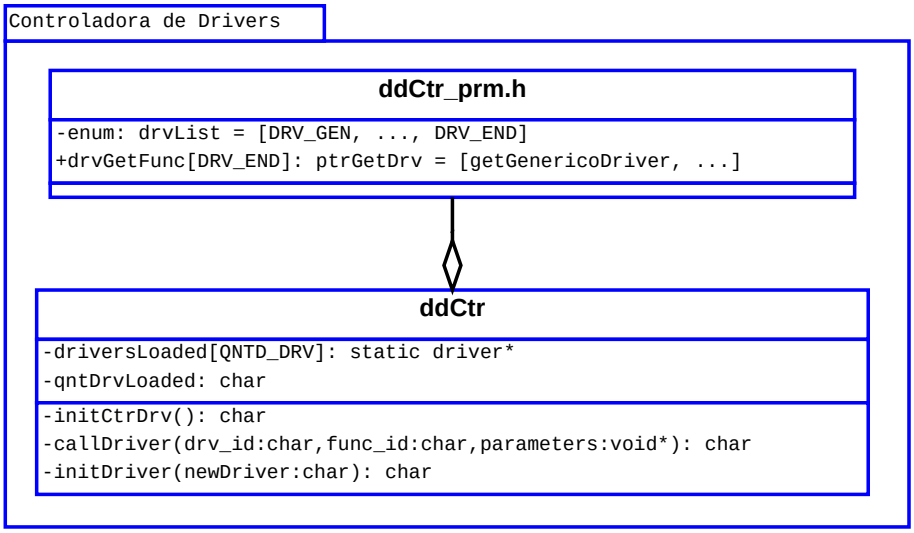


Figura 44: Diagrama da estrutura da controladora de *drivers*

Para que a controladora opere sobre os *drivers* é necessário obter um ponteiro para uma estrutura *driver*. Isto é realizado por meio de uma função padronizada *ptrGetDriver* implementada por cada *driver* do sistema.

Esta função retorna uma *struct* com todas as informações sobre o *driver*. A posição na qual os ponteiros estão armazenados é definida por um enumerado, no arquivo *ddCtr_prm.h*, auxiliando na identificação do ponteiro para seu respectivo *driver*. O código 10 exemplifica como realizar a ligação entre os *drivers* e a controladora.

Os *drivers* são manipulados pela controladora por meio de um um vetor de *drivers* carregados. Procurando manter sua simplicidade e baixo *overhead*, apenas três funções foram implementadas: uma para inicialização da controladora, uma para inicialização dos *drivers* e uma para interface entre o *kernel* e o *driver*.

O carregamento de um *driver* é feito por meio do acesso à lista de *drivers* disponíveis e, em seguida, sua inicialização. Somente depois da rotina de inicialização é que ele passa a ser armazenado na lista. Se não houver espaço para outro *driver*, a função retorna um

Código 10: Definição dos *drivers* disponíveis para uso

```
1 #include "drvInterrupt.h"
2 #include "drvTimer.h"
3 #include "drvLcd.h"
4 // enumerado para melhor acesso aos drivers
5 enum {
6   DRV_INTERRUPT,
7   DRV_TIMER,
8   DRV_LCD,
9   DRV_END
10 };
11 // funcoes de obtencao dos drivers
12 static ptrGetDrv drvInitVect[DRV_END] = {
13   getInterruptDriver,
14   getTimerDriver,
15   getLCDDriver
16 };
```

erro como apresentado no Código 11.

Código 11: Inicialização de um *driver* via controladora

```
1 char initDriver(char nDrv) {
2   char resp = FAIL;
3   if(qntDL < QNTD_DRV) {
4     dLoad[qntDL] = drvGetFunc[nDrv]();
5     resp = dLoad[qntDL]->drv_init(&nDrv);
6     qntDL++;
7   }
8   return resp;
9 }
```

A função que realiza a interface entre o *kernel* e o *driver* percorre a lista de *drivers* carregados para identificar o correto. Quando este é encontrado, a função procurada é chamada e os parâmetros são passados como um ponteiro para void, de modo que um mesmo caminho possa ser utilizado para passar qualquer tipo de parâmetro. Esta função é apresentada no Código 12.

A.1 *Driver*

Para que o sistema funcione corretamente todos os *drivers* devem possuir um mesmo padrão, tanto na recepção dos parâmetros como na inicialização.

O desenvolvimento começa com definições de tipo e assinaturas. Entre as definições mais importantes estão o ponteiro de função, que será utilizado para chamar as funções do *driver*, e a estrutura do *driver*.

Código 12: Função para passagem de parâmetro para as funções dos *drivers*

```
1 char callDriver(char drv_id, char f_id,
2                 void *prm) {
3   char i;
4   for (i = 0; i < qntDL; i++) {
5     if (drv_id == dLoad[i]->drv_id) {
6       return dLoad[i]->f_ptr[f_id](prm);
7     }
8   }
9   return DRV_FUNC_NOT_FOUND;
10 }
```

A estrutura de um *driver* é composta por um identificador, um vetor de ponteiros de *ptrFuncDriver* e uma função especial, também do tipo ptrFuncDriver. Esta última é responsável por inicializar o *driver*, uma vez que seja carregado na controladora. Esta é a estrutura utilizada pela controladora para gerenciar os *drivers*. O Código 13 apresenta a estrutura em questão.

Código 13: Estrutura de um *driver*

```
1 typedef struct {
2   char drv_id;
3   ptrFuncDrv *functions;
4   ptrFuncDrv drv_init;
5 } driver;
```

Para que a controladora acesse e gerencie os *drivers* corretamente, é preciso que possua uma referência para a estrutura do *driver* desejado. Os *drivers* devem então implementar uma única função pública (apenas esta função aparecerá no *header*), que retornará um ponteiro para a estrutura *driver*.

Supondo então um *driver* genérico, este deve implementar pelo menos duas funções: *init()* e *getDriver()*. Deve conter também uma variável do tipo *driver* e um vetor de ponteiros de função *ptrFuncDrv* com todas as funções disponíveis.

A função *getDriver()* é a única função que será conhecida pela controladora em tempo de compilação e permite que a função de inicialização (*init()*) e as demais funções do *driver* sejam acessadas externamente. Ela é a responsável por retornar a estrutura que contém as informações importantes do *driver*.

Já a função *init()* é a responsável pela inicialização da estrutura interna, preenchendo o vetor de funções com os ponteiros, inicializando as variáveis e realizando os procedimentos particulares para cada periférico ou dispositivo.

Supondo um *driver* genérico que possua apenas duas funções, uma para ler o valor de uma porta e uma segunda para configurar a função de *callback* da leitura. O esquema da

estrutura básica deste *driver* pode ser representado como na Figura 45.

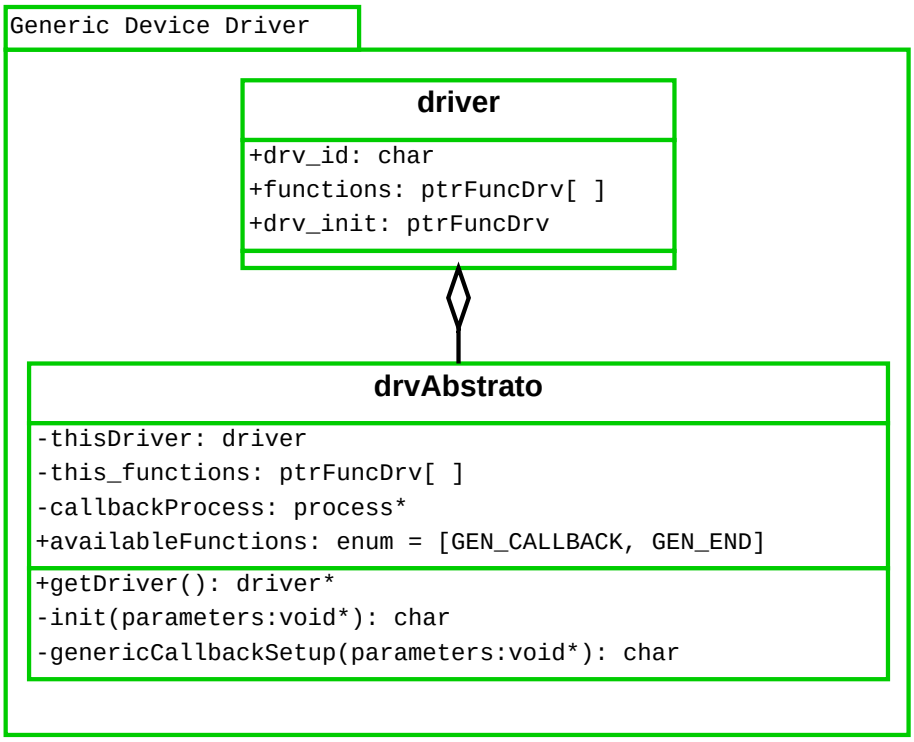


Figura 45: Diagrama da estrutura de um *driver* genérico

A.2 Camada de abstração da interrupção e callback

As interrupções são geradas por *hardware* de modo que o programa em execução não percebe sua ocorrência. Isto pode dificultar para que os desenvolvedores consigam utilizar estes recursos. Outro problema inerente a estas ferramentas é a especificidade e singularidades relacionadas a cada conjunto de processador e periféricos.

Uma alternativa para que as interrupções possam ser utilizadas de modo simplificado é criar estruturas que as transformem em eventos que os programas possam interpretar. O primeiro passo é manter as rotinas de interrupção dentro de um *driver* permitindo ao desenvolvedor da aplicação definir seu comportamento.

Isto pode ser feito por meio de um ponteiro de função que é chamado sempre que a interrupção acontece. A configuração deste é feita por meio de uma função de configuração disponibilizada pelo *driver* de interrupção. A estrutura básica deste procedimento é apresentada no Código 14.

O *driver* de interrupção armazenará um ponteiro dentro de si. Este ponteiro pode ser alterado por meio da função *setInterruptFunc()*. O endereço da função de alto nível que será executada deve ser passado como parâmetro.

Código 14: Camada de Abstração da Interrupção

```
1 typedef void (*intFunc)(void);
2 // ponteiro da ISR
3 static intFunc thisInterrupt;
4 // configuracao do ponteiro a ser chamado
5 char setInterruptFunc(void *prm) {
6   thisInterrupt = (intFunc) prm;
7   return OK;
8 }
```

Utilizando o ponteiro para armazenar a rotina de tratamento de interrupção, os detalhes de programação de baixo nível do compilador tornam-se ocultos para a aplicação. O Código 15 apresenta um exemplo que configura o *kernel* para executar a função *timerISR()* em cada interrupção gerada pelo *hardware* do *timer* do microcontrolador utilizado.

Código 15: Exemplo de configuração de interrupção via controladora

```
1 void timerISR(void) {
2   callDriver(DRV_TIMER, TMR_RESET, 1000);
3   kernelClock();
4 }
5 void main (void){
6   kernelInit();
7   initDriver(DRV_TIMER);
8   initDriver(DRV_INTERRUPT);
9   callDriver(DRV_TIMER, TMR_START, 0);
10  // habilitando as interrupcoes
11  callDriver(DRV_TIMER, TMR_INT_EN, 0);
12  callDriver(DRV_INTERRUPT, INT_TIMER_SET, (void*)timerISR);
13  callDriver(DRV_INTERRUPT, INT_ENABLE, 0);
14  kernelLoop();
15 }
```

Em alguns processos de I/O existe a necessidade de aguardar uma resposta de *hardware*, que em geral pode ser monitorada através da mudança de um bit de *status* em um registro pré-definido.

A leitura do conversor A/D é um bom exemplo: após ativada sua inicialização, o conversor começa a processar o sinal lido. Após finalizado o processo, o bit de *status* do conversor é alterado e uma interrupção ocorre. Sem um tratamento correto, o *kernel* ficaria ocioso, esperando a resposta do conversor, causando perda de tempo e de processamento do microcontrolador.

A técnica de *callback* permite ao sistema continuar funcionando mesmo à espera da resposta de um componente de *hardware*. Sendo assim, um processo pode requisitar que o *driver* inicie seu trabalho e continue executando outras tarefas. Na requisição, o processo passa o endereço de um segundo processo que será inserido no *pool* de processos,

quando uma interrupção acontecer. Deste modo, economiza-se tempo de processamento do microcontrolador enquanto procura-se receber o resultado o mais rápido possível.

Para que isso se cumpra, o *driver* deve ser capaz de gerar uma interrupção no sistema. Esta interrupção insere então o processo de *callback* no *pool* de processos, conforme a Figura 46.

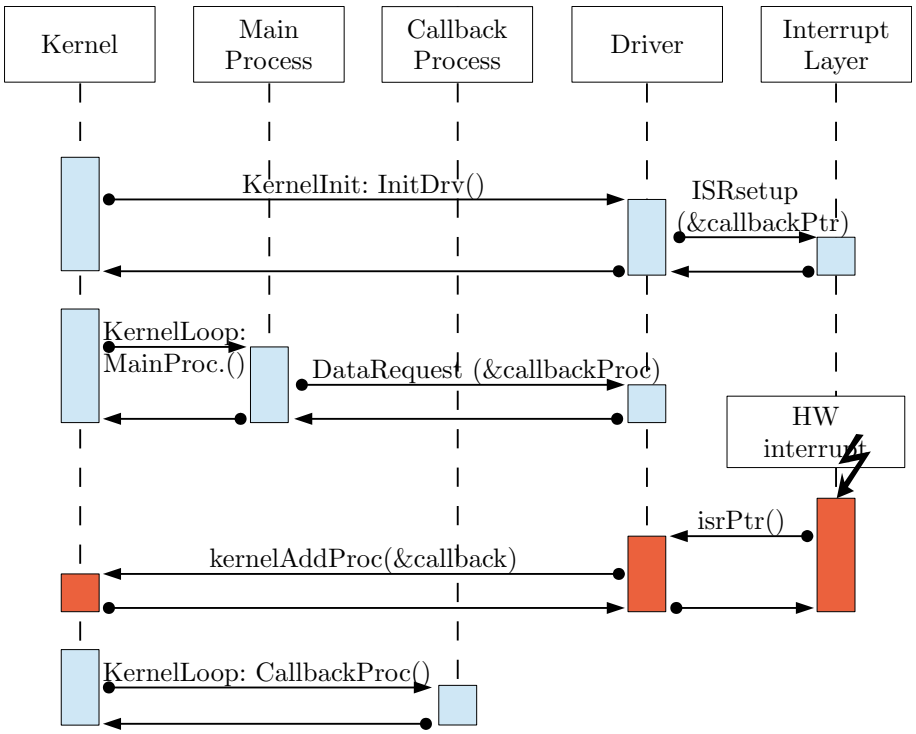


Figura 46: Diagrama de eventos da configuração e execução de um *callback*

ANEXO B – Equacionamento de um controlador digital do tipo PID

Sabe-se que a equação característica de um controlador PID é:

$$G_c(s) = \frac{U(s)}{E(s)} = K_p + K_d.s + \frac{K_i}{s} \tag{B.1}$$

Para obter a sua forma digital é necessário utilizar a transformada Z. A transformação bilinear é um modo de realizar essa conversão de modo simplificado. A transformada é dada pela equação:

$$s = \frac{2}{T}.\frac{z-1}{z+1} \tag{B.2}$$

Aplicando a transformada na equação tem-se:

$$\frac{U(z)}{E(z)} = \frac{k_p.2T(z+1)(z-1) + (2(z-1))^2.K_d + (T(z+1))^2.K_i}{T(z+1).2(z-1)} \tag{B.3}$$

Que expandida pode ser representada por:

$$\begin{aligned} &U(z)z^2 - U(z) = (e(z).k_p.z^2 - e(z).k_p) + \\ &\tfrac{2}{T}(e(z).z^2.k_d - 2.e(z)z.k_d + e(z).k_d) + \\ &\tfrac{T}{2}(e(z).z^2.k_i + 2.e(z).z.k_i + e(z).k_i) \end{aligned} \tag{B.4}$$

Multiplicando todos termos por z^{-2}, tem-se:

$$\begin{aligned} &U(z) = U(z).z^{-2} + (e(z).k_p - e(z).k_p.z^{-2}) + \\ &\tfrac{2}{T}(e(z).k_d - 2.e(z)z^{-1}.k_d + e(z).z^{-2}.k_d) + \\ &\tfrac{T}{2}(e(z).k_i + 2.e(z).z^{-1}.k_i + e(z).z^{-2}.k_i) \end{aligned} \tag{B.5}$$

A transformada Z-inversa reverte cada termo da equação em z para um termo de uma equação a diferenças segundo a equação:

$$z^{-k}X(z) = X(k-n) \tag{B.6}$$

Aplicando-se a transformada inversa na equação do PID em Z, tem-se a equação à diferenças pronta para ser implementada computacionalmente:

$$\begin{aligned} U(k) = U(k-2) &+ k_p\left(e(k) - e(k-2)\right) + \\ &k_i\left(e(k) + 2.e(k-1) + e(k-2)\right)\tfrac{T}{2} + \\ &k_i\left(e(k) - 2.e(k-1) + e(k-2)\right)\tfrac{2}{T} \end{aligned} \tag{B.7}$$

Observa-se, ao final, que a saída do sinal do controlador é dependente da entrada atual, das 2 últimas entradas e também da penúltima saída do PID. Assim, é necessário guardar os últimos 2 valores de entrada e saída durante a execução do programa. Estes valores devem ser atualizados a cada execução da equação de controle.

ANEXO C – Protocolo de comunicação da aplicação teste

Para a comunicação entre a placa utilizada e o computador, foi criado um protocolo serial utilizando um CRC (*Cyclic Redundancy Check*) de 16 *bits* para evitar falhas de comunicação. A padronização seguiu os moldes do protocolo NMEA de comunicação GPS. Os pacotes foram definidos da seguinte forma:

<START_BYTE><COMMAND_BYTE><PARAMETERS><CRC_16><END_BYTE>

Para o <START_BYTE> foi utilizado o caracter $. Já o <END_BYTE> é representado pelo caracter CR (*Carriage Return*). Esta escolha foi feita com base no protocolo NMEA de comunicação dos satélites de GPS ((US), 2002). Para evitar problemas de má interpretação, todos os demais valores devem ser codificados utilizando apenas números e letras em ASCII. Isto elimina a possibilidade de haver um byte válido que seja igual à 13 (*carriage return*), ou 36 (caracter $), fazendo que o interpretador perca a sincronia do pacote.

O <COMMAND_BYTE> representa o comando ou ação que o computador gostaria que a placa executasse. A Tabela 15 apresenta os comandos implementados. Foram implementadas ações para controle do comportamento do controlador digital, gerenciamento remoto de adição e remoção de processos além de um comando para simulação de falha em um bit na memória (na região da pilha).

<PARAMETERS> são os valores passados como parâmetros. A funcionalidade e utilização destes são dependentes do comando enviado.

Por fim, o CRC é um código para verificação de integridade da mensagem. Foi utilizado um CRC de 16 *bits* representando todos os dados entre o START_BYTE e o CRC. Para evitar problemas na transmissão o valor do CRC é enviado codificado em ASCII, utilizando quatro bytes em sua representação hexadecimal. Isto evita que ele seja confundido com o START_BYTE ou o END_BYTE.

Apenas os comandos de atualização dos parâmetros do controlador e o de trocar o valor de um bit exigem parâmetros. Na atualização devem ser enviados 16 bytes, que serão separados em 4 variáveis: k_p, k_i, k_d e $t_{amostragem}$. Os valores dos coeficientes são enviados com duas casas decimais, já o valor do tempo de amostragem é enviado em milisegundos.

A mensagem abaixo apresenta um pacote enviado à placa para que o valor de k_p seja 1,00, k_i seja 5,00 e k_d seja 0. O tempo de amostragem será configurado como 10 ms. Para

Tabela 15: Comandos padronizados para operação de gerenciamento via comunicação serial

Descrição	Comando	Tamanho do parâmetro (Bytes)	Total de Bytes no Pacote
Inicia Controle	'1'	0	7
Termina Controle	'2'	0	7
Reinicia Controle	'3'	0	7
Atualizar Parâmetros	'4'	20	27
Adicionar Processo (simulador de consumo)	'10'	0	7
Remover Processo (simulador de consumo)	'11'	0	7
Trocar Bit (indicando a posição da pilha)	'20'	2	7

este pacote o CRC é dado por 0xC8D4. As variáveis são identificadas pelas letras P, I, D e T.

```
//exemplo de mensagem para kp 1, kd 5, ki 0 e t 10ms
$4P0100I0500D0000T0010C8D4<CR>
```

Um sistema de *handshake* foi desenvolvido mas não fora totalmente testado. Devido às complexidades adicionadas na máquina de interpretação esta função foi desabilitada dos testes principais.

ANEXO D – Descrição e equacionamento da planta de teste para o controlador PID

A modelagem da planta (um circuito RC série) pode ser formulada a partir da relação entre tensão e corrente no capacitor. As tensões no capacitor e na entrada do circuito são representadas por V_c e V_e respectivamente.

$$V_c = \frac{1}{C}\int i(t)dt \tag{D.1}$$

Aplica-se a transformada de Laplace e obtém-se:

$$V_c(s) = \frac{1}{C.s}.I(s) \tag{D.2}$$

A corrente que passa pelo resistor pode ser modelada por:

$$I(s) = \frac{V_e(s) - V_c(s)}{R} \tag{D.3}$$

Substituindo I(s) pela relação anterior, encontra-se:

$$V_c(s) = \frac{1}{C.s}.\frac{V_e(s) - V_c(s)}{R} \tag{D.4}$$

$$V_c(s) = \frac{V_e(s) - V_c(s)}{R.C.s} \tag{D.5}$$

Após algumas manipulações, encontra-se a função de transferência:

$$V_c(s) = \frac{1}{C.s}.\frac{V_e(s) - V_c(s)}{R} \tag{D.6}$$

$$\frac{V_c(s)}{V_e(s)} = \frac{1}{R.C.s + 1} \tag{D.7}$$

Um capacitor de 100 [μF] e um resistor de 10 [kΩ] fornecem um sistema com uma constante de tempo de 1 [s]. Estes valores foram arbitrariamente escolhido de modo que o efeito da discretização da equação de controle seja mínimo, fazendo a constante de tempo do circuito muito maior que o tempo de amostragem. Deste modo, a equação da planta pode ser simplificada para

$$\frac{V_c(s)}{V_e(s)} = \frac{1}{s + 1} \tag{D.8}$$

ANEXO E – Trabalhos publicados

Artigos aprovados em periódicos

- ALMEIDA, Rodrigo Maximiano Antunes de . Microkernel Development for Embedded Systems. Journal of Software Engineering and Applications, v. 06, p. 20-28, 2013.

Apresentações em congressos internacionais

- Escalonador seguro para sistemas embarcados, 6a BSidesSP, São Paulo SP, 2013.
- Desenvolvimento de um microkernel: do projeto à implementação, ESC - Embedded System Conference, São Paulo SP, 2011.
- Embedded System Design: From Electronics To Microkernel Development, 19 DEFCON, Las Vegas EUA, 2011.