Rodrigo Maximiano Antunes de Almeida

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

Brasil
October 22, 2014

Rodrigo Maximiano Antunes de Almeida

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

Tese apresentada ao Curso de Doutorado em Engenharia Elétrica com ênfase em Automação e Sistemas Elétricos da Universidade Federal de Itajubá como requisito parcial para a defesa de doutorado

Universidade Federal de Itajubá – UNIFEI
Programa de Pós-Graduação em Engenharia Elétrica

Orientador: Luis Henrique de Carvalho Ferreira
Coorientador: Carlos Henrique Valério de Moraes

Brasil
October 22, 2014

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.

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 e 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. Segundo (AO et al.,)

1.1 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.

1.2 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 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.

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.

2.1.1 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).

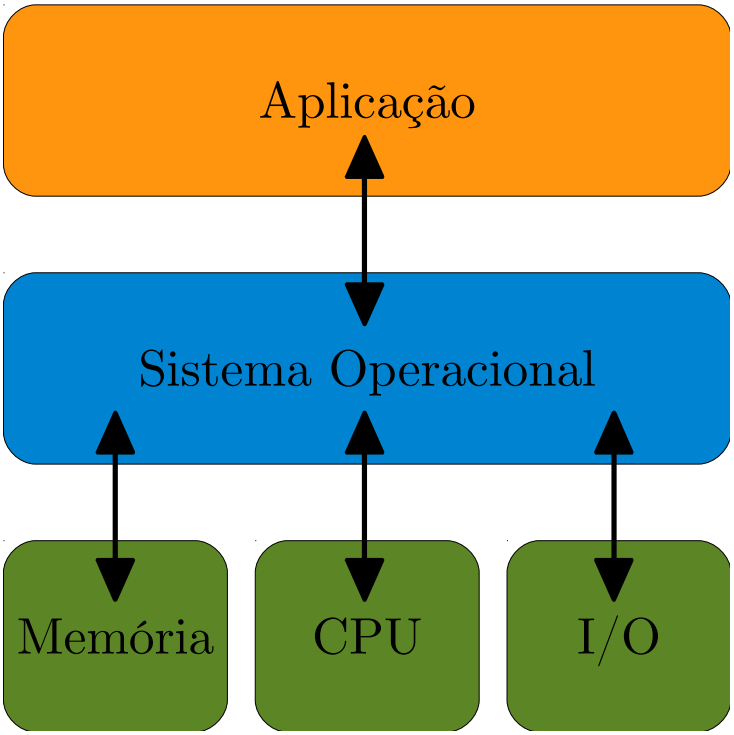


Figura 1: Interfaceamento realizado pelo sistema operacional

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.

2.1.2 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 **??** é apresentada a tendência entre vários sistemas operacionais, comerciais e de código aberto.

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 2.1.

$$Pr(0) = \frac{e^{-\lambda\tau}(\lambda\tau)^0}{0!} = e^{-\lambda\tau} \tag{2.1}$$

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 2: 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

2.1.3 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.

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

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

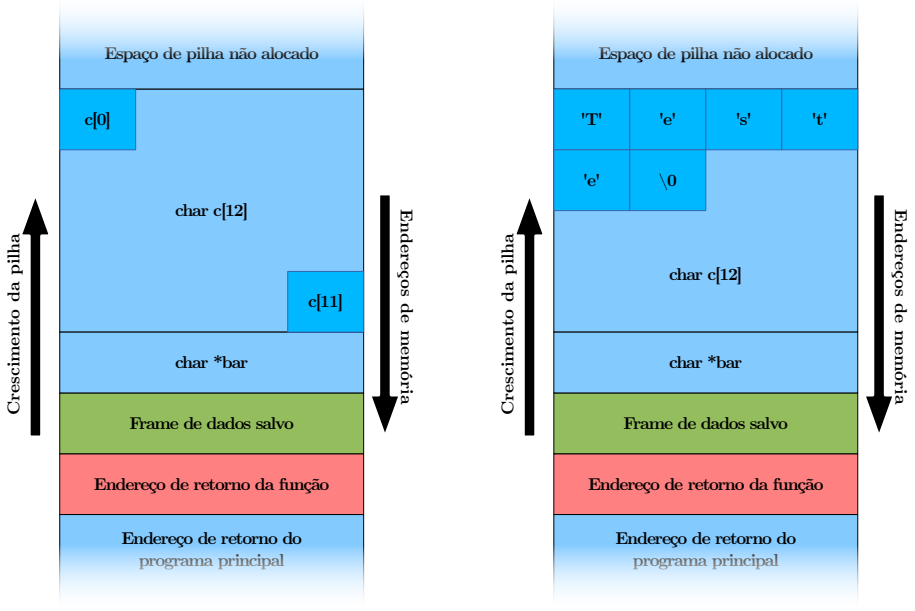
```
1 #include <string.h>
2 void buffOver (char *bar)
3 {
4    char  c[12];
5    strcpy(c, bar);  //sem checagem de tamanho
6 }

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

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 2a.

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 2b apresenta o mapa da memória para a entrada "Teste".



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

Figura 2: Posicionamento das variáveis e informações na pilha da função *buffOver()*

2.1.4 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.

$$P(x) = x^8 + x^6 + x^0 => 1\ 0100\ 0001_2 \tag{2.2}$$

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 **??** apresenta os resultados encontrados.

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 3 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.

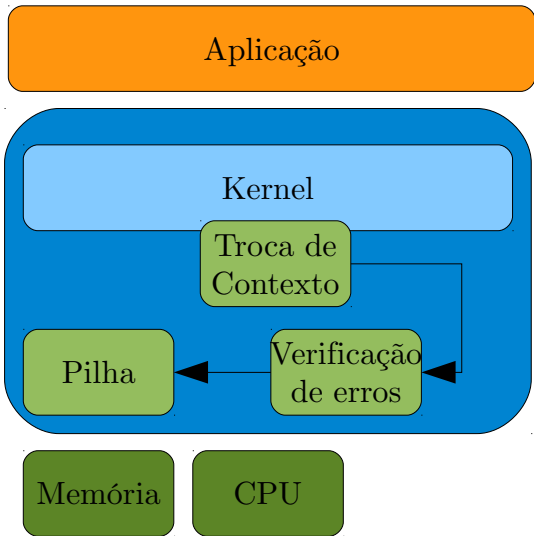


Figura 3: Modelo de sistema com verificação de erros na pilha

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 4 apresenta um resumo do sistema desenvolvido, a interligação do *kernel* com a aplicação e a controladora de *drivers* bem como todos os *drivers* implementados.

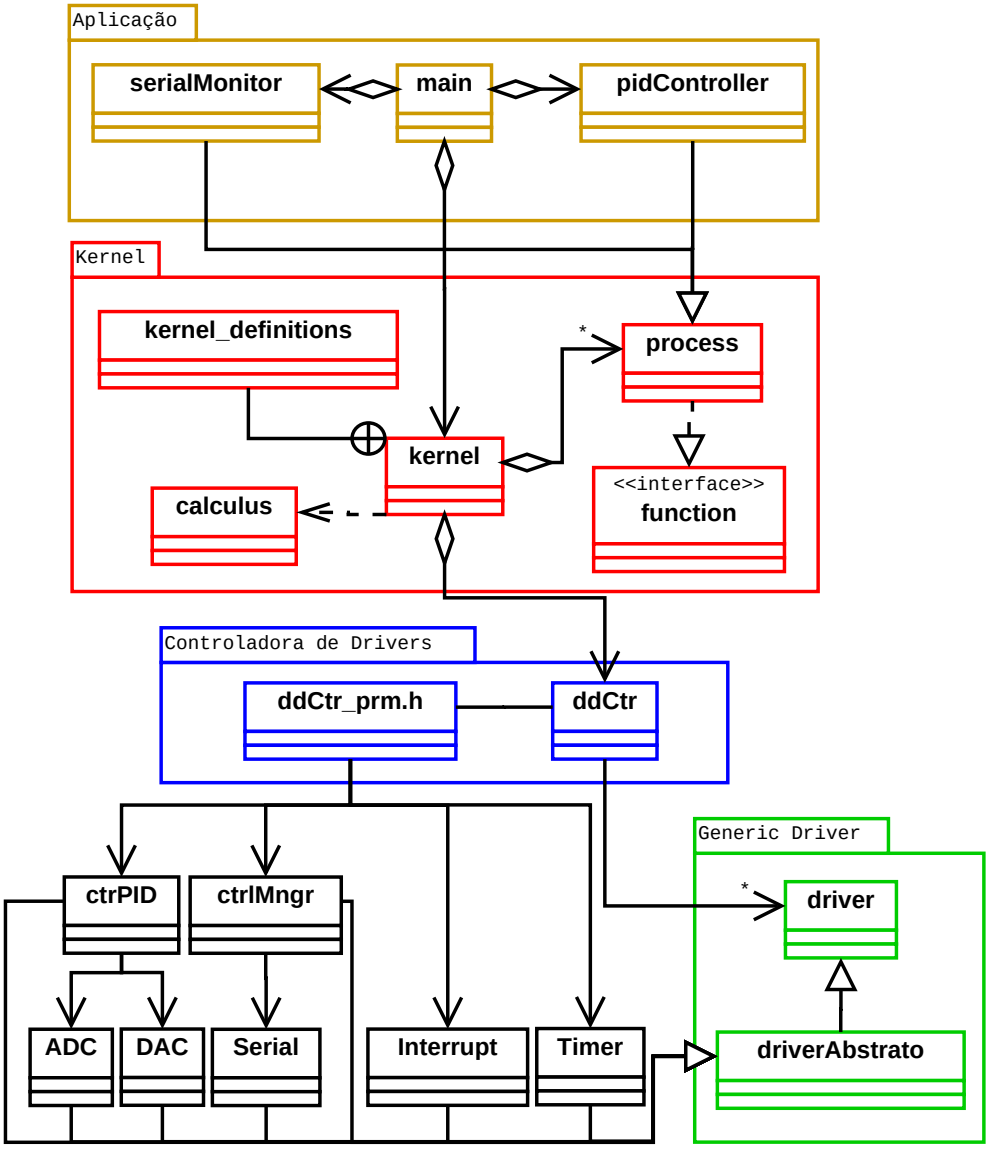


Figura 4: Diagrama UML do sistema desenvolvido

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 4.

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.

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

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_stack^SPdummy) - 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))) ↩
     ^ (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 }
```

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 3.

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 32 bytes. Este consumo pode ser deslocado da RAM para a memória não volátil. Deste

Tabela 3: 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

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 3. Mesmo quando considerada a versão com o sistema de correção ativo, os valores ainda são compatíveis com as alternativas. O gráfico na Figura **??** 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.

4.1 Testes do controlador PID

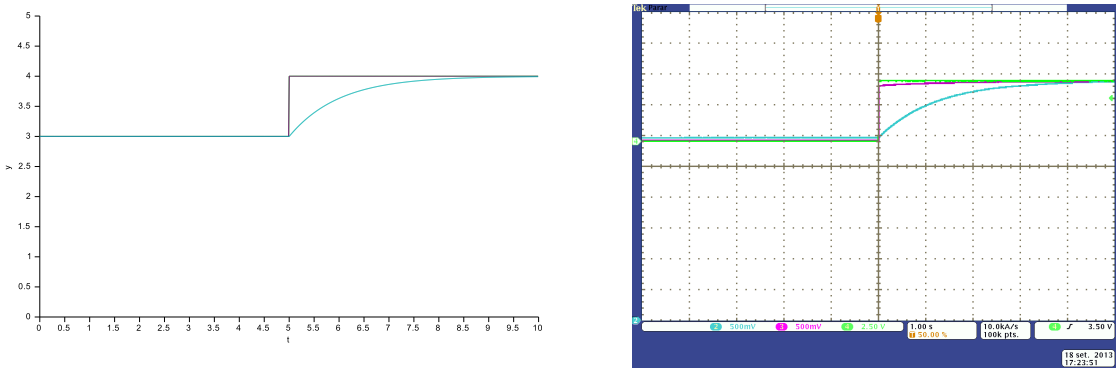
Este teste é focado na capacidade do sistema projetado de reproduzir corretamente as ações de controle mesmo sob falhas de memória e com excesso de processos para serem executados. O resultado do comportamento dinâmico destes testes não são significativos para este estudo, apenas a coerência destes resultados 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 **??**. A equação a ser inserida no simulador, portanto, é:

$$\frac{V_c(s)}{V_e(s)} = \frac{1}{s+1} \tag{4.1}$$

Foi utilizado o *software Scilab 5.4.1* para validação dos resultados obtidos pelo sistema de controle. A Figura 5a apresenta o resultado da simulação da planta em malha aberta. A Figura 5b apresenta as formas de onda obtidas no circuito real, gravadas pelo osciloscópio.



(a) Simulação (b) Teste

Figura 5: Resposta do sistema ao degrau unitário em malha aberta

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*, JSEA, v. 6, n. 1, p. 20–28, 2013.

AO, E. D. C. et al. Montagem de um robˆo autˆonomo utilizando logica fuzzy.

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.

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.

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

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.

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

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

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

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.

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.

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

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

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.

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

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*, v. 3, n. 2, p. 14–28, 2013.

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*, n. 5(22), p. 60–73, 2009.

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.

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`.

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.

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

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.

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. 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*, ACM, v. 17, n. 6, p. 337–345, 1974.

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.

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 6.

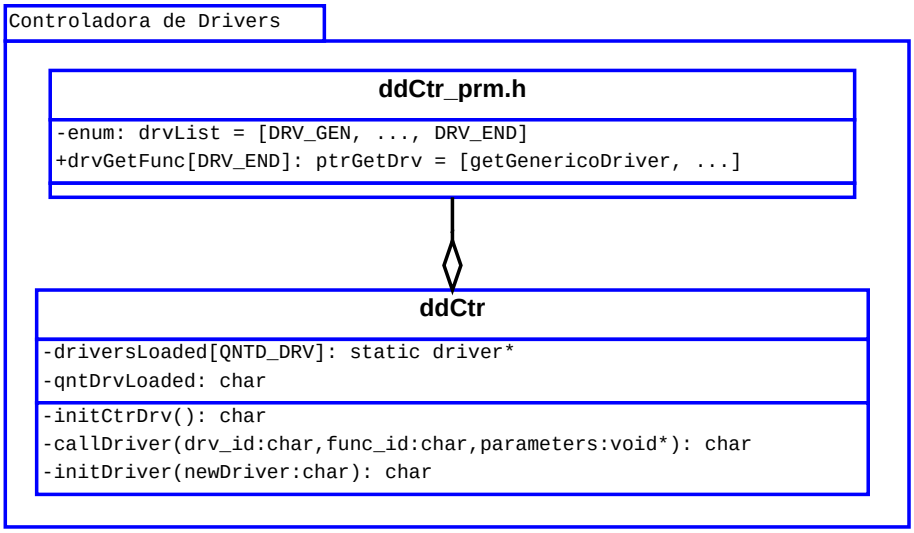


Figura 6: Diagrama da estrutura da controladora de *drivers*

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 3 exemplifica como realizar a ligação entre os *drivers* e a controladora.

Código 3: Definição dos *drivers* disponíveis para uso

```
#include "drvInterrupt.h"
#include "drvTimer.h"
#include "drvLcd.h"
// enumerado para melhor acesso aos drivers
enum {
  DRV_INTERRUPT,
  DRV_TIMER,
  DRV_LCD,
  DRV_END
};
// funcoes de obtencao dos drivers
static ptrGetDrv drvInitVect[DRV_END] = {
  getInterruptDriver,
  getTimerDriver,
  getLCDDriver
};
```

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:

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

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>