UNIVERSIDADE FEDERAL DE SANTA CATARINA

Bruno Fontana Canella

OPENAUTOS:
UM SISTEMA OPERACIONAL VEÍCULAR

Araranguá

2017

Bruno Fontana Canella

OPENAUTOS:
UM SISTEMA OPERACIONAL VEÍCULAR

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como parte dos requisitos necessários para a obtenção do Grau de Bacharel em Engenharia de Computação.
Orientador: Prof. Anderson Luiz Fernandes Perez, Dr.

Araranguá, Julho de 2017.

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Canella, Bruno Fontana
OpenAUTOS : Um Sistema Operacional Veícular / Bruno Fontana Canella ; orientador, Anderson Luiz Fernandes Perez, 2017.
136 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Araranguá, Graduação em Engenharia de Computação, Araranguá, 2017.

Inclui referências.

1. Engenharia de Computação. 2. Automação Veicular. 3. Sistema Operacional Embarcado. 4. AUTOSAR. 5. OpenAUTOS. I. Luiz Fernandes Perez, Anderson. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Computação. III. Título.

Bruno Fontana Canella

OPENAUTOS: UM SISTEMA OPERACIONAL VEÍCULAR

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Engenharia de Computação", e aprovado em sua forma final pela Universidade Federal de Santa Catarina.

Araranguá, 04 de Julho 2017.

Eliane Pozzebon:77921313949
Assinado de forma digital por Eliane Pozzebon:77921313949
Dados: 2017.07.19 18:22:41 -03'00'

Profª. Dr.ª Eliane Pozzebon
Coordenador
Universidade Federal de Santa Catarina

Banca Examinadora:

Anderson Luiz Fernandes Perez:02080345958
Assinado de forma digital por Anderson Luiz Fernandes Perez:02080345958
Dados: 2017.07.18 15:17:42 -03'00'

Prof. Dr. Anderson Luiz Fernandes Perez
Universidade Federal de Santa Catarina

Fabio Rodrigues de La Rocha:96300752020
Digitally signed by Fabio Rodrigues de La Rocha:96300752020
Date: 2017.07.27 13:48:01 -03'00'

Prof. Dr. Fábio Rodrigues De La Rocha
Universidade Federal de Santa Catarina

Marcelo Daniel Berejuck:64017451934
Assinado de forma digital por Marcelo Daniel Berejuck:64017451934
Dados: 2017.07.31 14:51:42 -03'00'

Prof. Dr. Marcelo Daniel Berejuck
Universidade Federal de Santa Catarina

Dedico este trabalho ao meu orientador, Anderson Luiz Fernandes Perez, a todos os amigos que fiz durante meu período de graduação na UFSC Araranguá, incluindo alunos, docentes e funcionários em geral, e a minha família.

AGRADECIMENTOS

Gostaria de começar prestando meus agradecimentos a dois alunos e amigos do curso de Engenharia da Computação da UFSC Araranguá: Alan Kunz Cechinel e Thiago Dal Pont, sem os quais minha jornada até a entrega do TCC e graduação teriam sido bem mais árduas e demoradas. Por todas as tardes me ensinando, pacientemente, os conteúdos de difícil compreensão do curso, bem como nas ajudas nos trabalhos e pela amizade em geral, deixo aqui o meu muito obrigado.

Gostaria também de agradecer formalmente ao professor Anderson Luiz Fernandes Perez por todo o esforço, compreensão e incentivo na produção deste trabalho de conclusão de curso. Não fosse pelo seu encorajamento, este trabalho seria algo bem mais simples e menos desafiador do que meu esforço e dedicação poderiam alcançar.

A minha família, que em nenhum momento duvidou das minhas capacidades e que sempre acreditou que eu terminaria o TCC sem reprovar nenhuma vez, dessa vez, e que também não ficou pegando no meu pé por causa do TCC da faculdade passada.

A todas as pessoas do Laboratório de Automação e Robótica Móvel (LARM), do qual fiz parte, e que conseguiu sempre manter uma atmosfera acolhedora e focada no aprendizado e desenvolvimento do campus e do curso.

E por fim, mas não menos importante, a todas as amizades que eu fiz durante este período da minha vida aqui na UFSC Araranguá.

Escrever um TCC é que nem fazer estrada. Depois que tá feita a base é só passar asfalto.

(Anderson, 2016)

RESUMO

Com a quantidade cada vez maior de dispositivos eletrônicos sendo agregados aos veículos automotivos e, por consequência, de Unidades de Controle Eletrônica (Eletronic Control Units - ECU) para gerência-los, tornou-se necessário a criação de padrões de comportamento e comunicação para estas centrais, afim de garantir que diferentes fabricantes pudessem desenvolver soluções veiculares intercambiáveis, sem que se preocupassem com estes pontos de integração. Como resultado da padronização surgiram padrões tanto para o desenvolvimento de hardware quanto para o de software, sendo que atualmente o padrão AUTOSAR é o mais utilizado pela indústria automotiva. Devido a maioria das soluções existentes hoje no mercado, que respeitam este padrão, serem proprietárias, este trabalho propõe o desenvolvimento de um sistema operacional de qualidade comercial e código aberto, baseado nestas mesmas normas, e que possa ser utilizada como referencia de aprendizagem e, até mesmo, como uma alternativa para a programação de ECUs automotivas.

Palavras-chave: Automação Veicular, Sistema Operacional Embarcado, AUTOSAR, OpenAUTOS.

ABSTRACT

Due to the increasing amount of electronic devices being added to automotive vehicles and, by consequence, of Eletronic Central Units (ECUs) to manage them, it became necessary to create behavioral and communication standards for these centrals in order to ensure that different manufacturers could develop interchangeable vehicle solutions without having to worry about these points of integration. As a result of standardization, standards have emerged for both hardware and software development, and today AUTOSAR standard is the one most used by the automotive industry. Due to the majority of the existing solutions in the market that respect this standard being proprietary, this thesis proposes the development of a commercial quality and open source operating system, based on those same standards, and that can be used as a learning reference and even as an alternative in the development of automotive ECUs.

Keywords: Vehicle Automation, Embedded Operating Systems, AUTOSAR, OpenAUTOS.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ECU	Eletronic Control Unit
ABS	*Antiblockier-Bremssystem*
AUTOSAR	AUTomotive Open System ARchitecture
SO	sistema operacional
OpenAUTOS	*Open AUTomotive Operating System*
RAM	*Random Access Memory*
ROM	*Read-Only Memory*
IHM	Interface Homem-Máquina
ESP	*Electronic Stability Program*
ASP	*Automatic Stability Control*
4WD	*Four-Wheel Drive*
LIN	*Local Interconnect Network*
CAN	*Controller Area Network*
ISO	Organização Internacional de Normalização
UART	*Universal Asynchronous Receiver/Transmitter*
LSD	*Low Side Driver*
HSD	*High Side Driver*
OSEK	***O****ffene* ***S****ysteme und deren Schnittstellen für die* ***E****lektronik in* ***K****raftfahrzeugen*
VDX	***V****ehicle* ***D****istributed e****X****ecutive*
OIL	*OSEK Implementation Language*
RTOS	*Real Time Operating Systems*
UML	*Unified Modeling Language*
E/S	Entrada e Saída
SOE	Sistemas Operacionais Embarcados
ISR	*Interrupt Service Routine*
ASR	*Asynchronous Signal Routine*
API	*Application Programming Interface*
LED	*Light-Emitting Diode*
PCB	*Printed Circuit Board*
RTE	Run-Time Environment

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Desde seu surgimento, popularização e evolução até os dias atuais, os veículos automotivos aumentaram em muito a sua complexidade, a ponto de que apenas o conhecimento mecânico do veículo não é mais suficiente. A quantidade de componentes eletrônicos presentes nos veículos automotivos aumentou consideravelmente passando, inclusive, a substituir sistemas puramente mecânicos. Dentre os fatores que alavancaram estas mudanças destacam-se o barateamento, miniaturização e popularização dos componentes eletrônicos, bem como a adequação da indústria automobilística as novas legislações, que passaram a ditar a emissão máxima de poluentes e a exigir mecanismos de segurança, como o *Antiblockier-Bremssystem* (ABS)[1] e o *Airbag* (SIMONNOT-LION; NAVET, 2008).

Este aumento de componentes eletrônicos nos automóveis passou a exigir, também, um maior número de ECUs para realizar seu gerenciamento, que passaram a ser espalhadas pelo chassi do veículo conforme a proximidade ao sistema tal qual gerenciam. Com o grande número de desenvolvedores de sistemas para veículos e, consequentemente, de ECUs, surgiu um novo desafio no mercado, de garantir que estas centrais pudessem se comunicar umas com as outras, bem como serem diagnosticadas quanto a presença de erros, como quando um sistema restrito temporalmente venha a falhar (MACHER et al., 2014).

Com o propósito de padronizar e, assim, facilitar o desenvolvimento e intercambialidade de auto-peças por terceiros, as principais montadoras e fabricantes de veículos entraram em um consenso, estipulando um padrão de normas de desenvolvimento para veículos automotivos, chamada de AUTomotive Open System ARchitecture (AUTOSAR), a qual se encontra em sua quarta revisão (AUTOSAR, 2016).

Para gerenciar os diversos módulos eletrônicos agora presentes em um veículo, bem como garantir a interoperabilidade entre eles, foram criadas normas referente ao desenvolvimento de sistemas operacionais embarcados. Estas normas visam o estabelecimento de padrões para o funcionamento, comunicação e especificações do sistema operacional (SO), sem sacrificar a liberdade criativa de desenvolvimento do sistema, como a seleção de hardwares e implementação de algoritmos.

[1] Traduzido como: Sistema de Anti-Bloqueio.

1.1 JUSTIFICATIVA E MOTIVAÇÃO

A maioria das soluções em SOs automotivos são exclusivamente comerciais e de código fechado. Embora existam soluções de código aberto para sistemas embarcados, não existe, na atualidade, um SO de código aberto homologado nos padrões do AUTOSAR. O projeto que mais se aproxima deste cenário é o Trampoline, que se encontra em fase de homologação pelo consórcio AUTOSAR (BéCHENNEC; FAUCOU, 2016).

Visando a criação de um SO embarcado que mantivesse um modelo de desenvolvimento de código aberto, surgiu a idealização do *Open AUTomotive Operating System* (OpenAUTOS)[2]. Através do desenvolvimento do OpenAUTOS, deseja-se alcançar um SO nacional que seja referência na área, utilizando componentes e tecnologias com alta disponibilidade e de fácil acesso, além de agregar contribuições com a própria comunidade acadêmica.

1.2 OBJETIVOS

Esta seção apresenta o objetivo geral e os objetivos específicos deste trabalho.

1.2.1 Geral

Desenvolver um sistema operacional embarcado de código aberto que atenda as normas estabelecidas pelo padrão AUTOSAR.

1.2.2 Específicos

1. Levantar o estado da arte com respeito a algoritmos para sistemas operacionais embarcados;
2. Estudar padrões de sistemas automotivos;
3. Levantar os requisitos para implementação de um SO de acordo com a norma AUTOSAR;
4. Estabelecer um projeto de código aberto em um repositório on-

[2]Traduzido como: Sistema Operacional Automotivo Aberto.

line;

5. Documentar o código do projeto;
6. Avaliar o sistema desenvolvido.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho está dividido em 6 capítulos, contando com a introdução, além de 1 apêndice e 3 anexos.

O **Capítulo 2** apresentará os domínios eletrônicos de funcionamento em veículos, seus meios de comunicação, unidades de controle e engenharia de software automotivo.

O **Capítulo 3** abordará os principais conceitos sobre sistemas operacionais. Ao final do capítulo serão relatados alguns estudos de caso a respeito de sistemas operacionais embarcados com foco para a automação veicular.

O **Capítulo 4** apresentará a proposta do SO OpenAUTOS, destacando as escolhas tanto do projeto de software bem como a arquitetura de hardware adotada.

O **Capítulo 5** descreverá os testes e resultados de validação do OpenAUTOS, discutindo o comportamento do SO quanto ao que é esperado pela norma.

O **Capítulo 6** apresentará as considerações finais, bem como sugestões para trabalhos futuros.

O apêndice apresenta os códigos utilizados para a realização dos experimentos do OpenAUTOS.

Os anexos apresentam as interfaces de código do SOE OpenAUTOS divididos em 3 partes: Interfaces do OSEK/VDX, do OpenAUTOS OS e para a plataforma alvo.

2 AUTOMAÇÃO VEICULAR

A automação veicular faz um extenso uso de componentes eletrônicos em seus sistemas. Nos tópicos que seguem serão abordados os domínios eletrônicos de funcionamento em veículos, seus meios de comunicação, unidades de controle e engenharia de software automotivo.

2.1 TRANSIÇÃO DA MECÂNICA PARA ELETRÔNICA

De acordo com Bosch (2014), os motivos para evolução automotiva variam entre ganhos de desempenho e segurança, barateamento de sistemas sem perda de qualidade e a adequação à novas legislações relacionadas ao controle de emissão de poluentes.

Contudo, independente da motivação, é notável que há cada vez mais a adoção de sistemas eletrônicos embarcados. A Figura 1 ilustra veículos de eras diferentes, a primeira dotada exclusivamente de recursos mecânicos, enquanto que a segunda destaca as centrais eletrônicas, as quais vem aumentando em número a cada ano que passa.

Figura 1: Evolução dos componentes eletrônicos.

(a) 1950 (b) 2000

Extraido de: Oliveira (2015) e Museum (2016).

Nos dias de hoje, recursos como injeção eletrônica e freios ABS são exigências para que um novo modelo de carro possa entrar em circulação.

O crescente uso de recursos eletrônicos demanda também que os mesmos possam se comunicar e cooperar entre si, para que o máximo de desempenho possa ser extraído dos sistemas. Neste caso, os domínios funcionais auxiliam na classificação e responsabilidade destes sistemas.

2.2 DOMÍNIOS FUNCIONAIS DE UMA ARQUITETURA VEICULAR

De modo a facilitar a classificação e identificação de sistemas eletrônicos que compõem um veículo, historicamente, foram estabelecidos cinco domínios de funcionalidade na arquitetura veicular, sendo eles o *Trem de Força*, *Chassi*, *Corpo*, Interface Homem-Máquina (IHM) e *Telemática* (SIMONNOT-LION; NAVET, 2008).

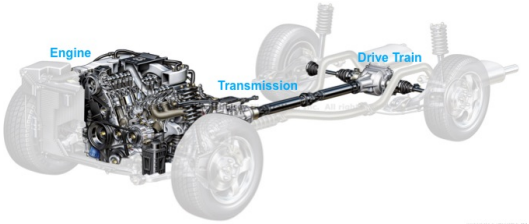
Com o avanço das últimas décadas, recursos veiculares pertencentes a estes domínios que até então eram implementados via sistemas mecânicos e elétricos puderam ser substituídos por elementos da eletrônica, agregando confiabilidade, segurança e, em alguns casos, possibilitando o atendimento de requisitos que antes eram inviáveis de implementação.

Nas demais subseções, serão apresentados em mais detalhes cada um dos domínios funcionais e serão levantados exemplos de como eles se beneficiaram nos últimos anos com o uso a microeletrônica e microprocessadores.

2.2.1 Trem de Força

O domínio trem de força está relacionado aos sistemas que participam na propulsão longitudinal de um veículo. Conforme pode ser observado na Figura 2, estes sistemas são compreendidos pelo *motor*, *transmissão* e demais componentes subsidiários, como o *trem de rodagem*[1].

Figura 2: Trem de força de um veículo automotivo.



Extraido de: Simonnot-Lion e Navet (2008).

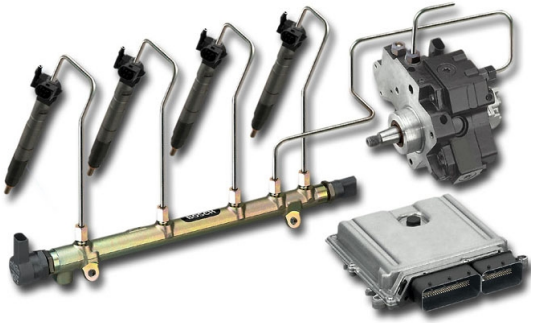
Os sistemas presentes neste domínio buscam otimizar componen-

[1] Drive Train, no inglês.

tes para condução, conforto, economia de combustível, dentre outros. Há uma grande quantidade de sistemas embarcados vinculados principalmente ao motor, onde desde 1939 a empresa Bosch já fazia pesquisas sobre o gerenciamento eletrônico de motores (BOSCH, 2010).

Um sistema bastante conhecido no ramo automobilístico, e que veio a substituir completamente o uso de carburadores, é a injeção eletrônica, implementado comercialmente pela primeira vez pela Bendix Corporation, em 1958[2](MATTAR, 2014). Este sistema é responsável pela injeção de combustível dentro do motor e controla as quantidades e proporção entre ar e comburente, bem como a produção da centelha pelas velas de ignição. O controle de tempos e proporções é feito por uma ECU, componente o qual será discutido em maior detalhes na Seção 2.4. A Figura 3 apresenta um modelo com os bicos injetores e sua respectiva ECU.

Figura 3: Injeção Eletrônica e ECU.



Extraido de: Bosch (2015).

2.2.2 Chassi

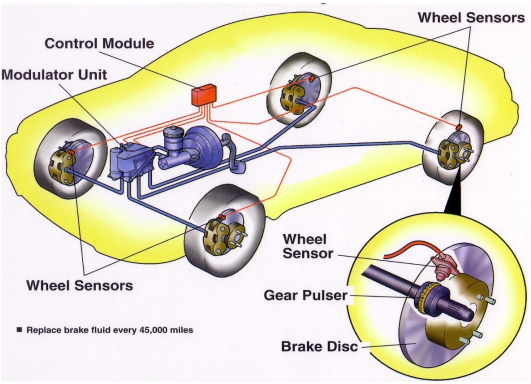
Os elementos que compõem o *Chassi* tem por objetivo controlar a interação do veículo com o ambiente ao qual ele irá circular. Isto é alcançado através de leituras e atuações nas rodas, suspensões, sistemas de freio, direção, aceleração, dentre outros. Em carros mais modernos, também são levados em conta as condições da estrada, velocidade do vento, situação do clima e outros fatores externos ao veículo.

Dos sistemas que compõem este domínio, podem ser citados os

[2]Na época batizado de *Electrojector* pela empresa.

de ABS, *Electronic Stability Program* (ESP)[3], *Automatic Stability Control* (ASP)[4] e o *Four-Wheel Drive* (4WD)[5]. A Figura 4 apresenta uma ilustração dos componentes do ABS.

Figura 4: Componentes do sistema de ABS.



Extraido de: Chase (2015).

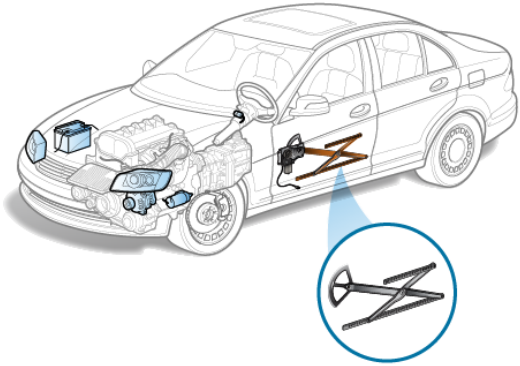
Como o domínio do *chassi* tem por objetivo a segurança dos passageiros e do próprio veículo, seus sistemas utilizam recursos de alta qualidade e o mesmo é tratado como um setor crítico. Características desejáveis são similares a do *trem de força*, fazendo uso de uma técnica adicional de segurança chamada de *X-by-Wire*[6]. Este método consiste em manter um sistema mecânico (ergo, o *"por fio"* do nome) redundante ao eletrônico. Historicamente, o sistema mecânico é o sistema original, sendo mantido apenas por questões de segurança.

2.2.3 Corpo

O domínio do *Corpo* do veículo incluí os demais sistemas que não estão relacionados ao controle de suas dinâmicas. Mecanismos como limpadores de vidro, faróis, portas e janelas, assentos e retrovisores hoje em dia são controlados diretamente por sistemas baseados em software. A Figura 5 provê uma ilustração de alguns destes sistemas e sua localização no veículo.

[3]Traduzido como: Programa Eletrônico de Estabilidade.
[4]Traduzido como: Controle de Estabilidade Automática.
[5]Traduzido como: Tração nas Quatro Rodas.
[6]Traduzico do inglês como *X-por-fio*

Figura 5: Sistemas pertencentes ao domínio do *corpo*. Em destaque o mecanismo de abertura e fechamento dos vidros.



Extraido de: Glass (2016).

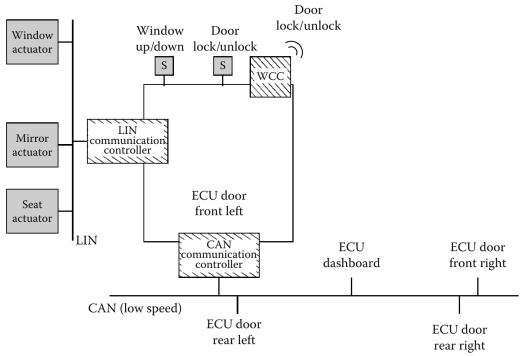
De uma maneira geral, estes sistemas, embora necessários para o devido funcionamento do veículo e conforto dos passageiros, não representam um fator crítico de proteção para o usuário, salvo alguns sistemas de segurança quanto ao controle de acesso do veículo.

Este domínio geralmente envolve a comunicação de diversas funções entre si, o que, consequentemente, gera uma arquitetura complexa distribuída. Desta necessidade emerge o conceito de subsistemas baseados em redes de sensores-atuadores de baixo custo.

Um sistema de controle para portas, por exemplo, poderia utilizar uma ECU principal para realizar as operações da porta do motorista (trancar/destrancar a porta, fechar/abrir janela, ajustar o banco). Esta ECU se comunicaria com as ECUs da trava, janela e banco por uma rede *Local Interconnect Network* (LIN)[7][8] de baixo custo. Como uma operação de trancar/destrancar uma porta pode influenciar nas outras, todas as ECUs das portas estariam conectadas por uma rede *Controller Area Network* (CAN)[9][10], assim como com o painel, afim de indicar o estado das portas. A Figura 6 apresenta uma ilustração destas interações.

[7]Traduzido como: Rede Local Interconectada.
[8]vide tópico 2.3.1.2.
[9]Traduzido como: Área de Rede Controladora.
[10]vide tópico 2.3.1.1.

Figura 6: Comunicação de uma ECU principal com as demais.



Extraido de: Simonnot-Lion e Navet (2008).

2.2.4 IHM e Telemáticos

O domínio da *interface homem-máquina* governa a iteração do condutor e passageiros com as várias funções embarcadas do veículo. Resultados que podem ser apresentados pela IHM são:

- Estados do Veículo: velocidade, nível do óleo, situação das portas, estado das luzes;
- Estado de um dispositivo multimídia: rádio sintonizada, controle de volume;
- Reposta a uma requisição: visualização de um mapa no sistema de navegação.

Sistemas embarcados deste domínio geralmente estão instalados no painel de um veículo, conforme ilustração da Figura 7.

O domínio da *telemática*, em contraste à IHM, governa a troca de informações entre veículos ou entre o veículo e infraestruturas da estrada.

Exemplos de sistemas deste domínio são os coletores automáticos de tarifas para pedágios, no qual o carro pode passar por uma via sem precisar parar para realizar o pagamento em dinheiro. Outras aplicações incluem a comunicação com serviços de trânsito, os quais podem indicar a situação em uma rodovia, como acidentes ou trânsito intenso.

Figura 7: Painel de um carro.

Extraido de: Byers (2016).

Uma das transições que a área está passando atualmente é no desenvolvimento de carros inteligentes, capazes de detectar riscos ao motorista e, até mesmo, dirigir automaticamente, sem que haja a necessidade de intervenção do condutor humano. Empresas como a Google e a Tesla estão com soluções comerciais em fase de avaliação no mercado.

A Figura 8 ilustra a visão do veículo para com seus meios externos, identificando veículos e limites de velocidades.

Figura 8: Visão dos sistemas de um carro automático.

Extraido de: Volvo (2016).

Ambos os domínios estão se aproximando cada vez mais do conceito de Internet das Coisas, onde os veículos serão um dos grandes beneficiados desta integração, devido aos novos tipos de serviços e in-

formações que serão trocadas por eles (EVANS, 2011).

2.3 PROTOCOLOS DE COMUNICAÇÃO AUTOMOTIVOS

Devido a grande quantidade de componentes que necessitam trocar informações em um veículo, é imperativo que existam meios de comunicação padronizados e que atendam aos requisitos dos sistemas, como taxa de transmissão e segurança. Os protocolos de comunicação permitem que estas integrações sejam realizáveis com relativa facilidade.

Embora mais complexas, é comum o uso de redes distribuídas em veículos automotivos, onde mais de um protocolo de comunicação são empregados.

2.3.1 Protocolos Automotivos

Existem diversos protocolos automotivos, os quais estão listados em Guimarães (2007) e devidamente identificados quanto a fabricante, aplicação, tipo de barramento, dentre outros quesitos.

A classificação destes é feita por meio de grupos, os quais definem a taxa de transmissão e/ou aplicação dos mesmos. A Tabela 1 lista estes grupos de uma maneira resumida.

Tabela 1: Grupos de Protocolos.

Grupo	Caracteristicas & Uso	Protocólos
Classe A	Conforto 10Kbps	SINEBUS, I^2C, SAE, LIN
Classe B	Entretenimento 10Kbps a 125Kbps	CAN 2.0, Class 2, SCP, PCI
Classe C	Segurança 125Kbps a 1Mbks	CAN 2.0, ISO, SAE J139
Diagnóstico	Diagnóstico Embarcado	J1850, Class 2, SCP, PCI
Mobile Media	PC-On-Wheels	IDB-C, MOST, MML, USB
Safety Bus	Airbag	BST, DSI, Byte Flight
Drive by-wire	Segurança	TTP, FlexRay, TTCAN

Dos protocolos citados na Tabela 1, existem dois que merecem destaque, devido a versatilidade e baixo custo de implementação, que são os barramentos CAN e LIN.

2.3.1.1 Controller Area Network - CAN

Foi desenvolvido pela empresa alemã Robert Bosch, entre 1983 e 1986, para uso em veículos de transporte. Na atualidade, o protocolo CAN é amplamente utilizado na comunicação veicular, além de estar presente em navios, tratores e outros (BOSCH, 2014).

Das aplicações que o CAN pode assumir em um veículo, é possível destacar as que estão presentes na Tabela 2.

Tabela 2: Aplicações do barramento CAN.

Aplicação	Transmissão	Exemplos
Tempo Real	>125kbps <1Mbps	Motronic, Câmbio, ESP
Multiplex	>10kbps <125kbps	Ar Condicionado, Travas
Comunicação Móvel	<125kbps	Celular, Áudio, Navegação
Diagnóstico	500kbps	Diagnóstico das ECUs

O protocolo define especificações tanto físicas quanto lógicas, e apresentam variações, o que o torna bastante flexível.

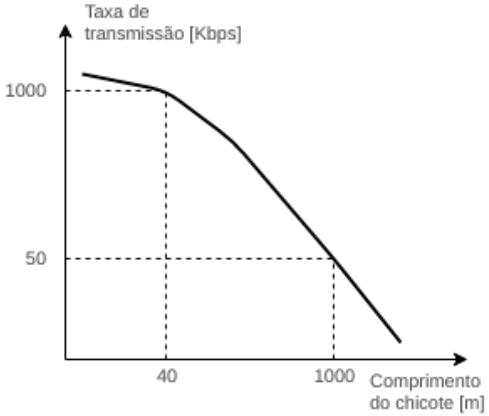
Conforme explica Guimarães (2007) a arquitetura de sua rede é a multimestre, na qual qualquer módulo pode assumir o papel de mestre enquanto que os demais se tornam escravos, a qualquer momento. As mensagens são transmitidas para todos os módulos da rede, via *broadcast*.

Cada rede CAN é formada por um único par trançado de fios, chamado de chicote, no qual a taxa de transmissão máxima varia conforme seu comprimento, conforme ilustrado na Figura 9. Vale ressaltar que um mesmo veículo pode possuir diversas redes CAN dentro dele.

Outro fator que determina seu desempenho é o tamanho das mensagens. Alguns bits são utilizados para identificar qual tipo de mensagem que está sendo enviada. A versão original do protocolo (CAN 2.0A) utiliza 11 bits para esta identificação, resultando em 2048 tipos de mensagem. Devido a necessidade de adicionar ainda mais tipos de mensagens a uma rede CAN, uma segunda versão foi criada (CAN 2.0B), com 29 bits utilizados para identificação. Este *overhead* afeta negativamente a taxa de transmissão da rede, mas virtualmente elimina o limite de mensagens, que passa a ser mais de 537 milhões.

Existem duas normas da Organização Internacional de Normalização (ISO) que especificam os fundamentos do protocolo CAN. A ISO

Figura 9: Relação entre tamanho do chicote e taxa de transferência.



Adaptado de: Guimarães (2007).

11898 determina as características de uma rede CAN de alta velocidade, entre 125Kbps e 1Mbps, enquanto que a ISO 11519-2 possuí especificações para uma rede de baixa velocidades, entre 10Kbps e 125Kbps.

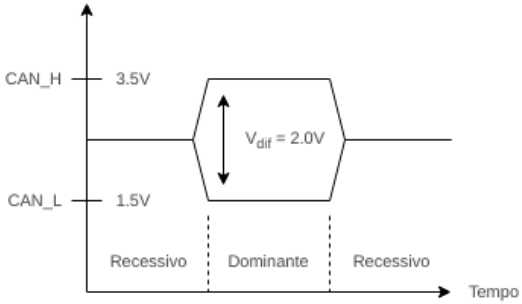
Outro ponto que merece destaque é quanto ao formato de envio dos dados na rede. Diferente do formato eletrônico tradicional, onde são utilizados níveis fixos de tensão para representar os valores 0 e 1, no protocolo CAN são utilizados *bits recessivos e dominantes*. Dois fios trançados, chamados de `CAN_H` e `CAN_L`, transmitem um diferencial de tensão, e é a partir desta diferença que se estipula os valores de 0 e 1. Caso o par trançado sofra ruídos externos em seu sinal, isto não afetara o valor final, pois ambos serão igualmente distorcidos, mantendo o diferencial original. A Figura 10 ilustra um exemplo desta transmissão.

O protocolo ainda conta com diversos mecanismos de tratamento de colisões, como a arbitragem *bit-a-bit* e detecção de falhas, como *bit monitoring*, *bit stuffing*, *cyclic redundancy check*, *frame check* e *acknowledgment error check*.

2.3.1.2 Local Interconnect Network - LIN

Criado por um consórcio de empresas europeias do segmento automotivo, o protocolo LIN foi desenvolvido para servir como um sub-barramento de barramentos maiores. Conforme descrito em ST-Microelectronics (2002), ele foi idealizado para uso em aplicações de

Figura 10: Valores de transmissão do protocolo CAN.

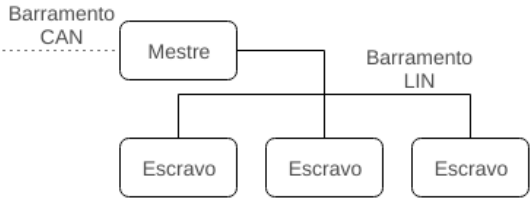


Adaptado de: Guimarães (2007).

troca simples como assentos, travas nas portas, teto solar, sensores de chuva, dentre outras funções, em sua maioria, pertencentes ao domínio do corpo.

O sub-barramento LIN é baseado em protocolos de comunicação serial. Ele faz uso da arquitetura mestre/escravos, utilizando um único fio para transmissão de dados, geralmente em conjunto com uma rede CAN, conforme ilustrado na Figura 11.

Figura 11: Rede CAN/LIN.



Adaptado de: STMicroelectronics (2002).

Para redução de custos, os componentes podem ser guiados sem o uso de um cristal ou ressonador cerâmico, por utilizar sincronização temporal. É um sistema baseado em *Universal Asynchronous Receiver/Transmitter* (UART)[11], disponível na maioria dos microcontroladores. O barramento também é capaz de detectar nós defeituosos na rede através do uso de técnicas para verificação de erros e *checksum* e paridade, detalhadas em Tanenbaum e Bos (2014).

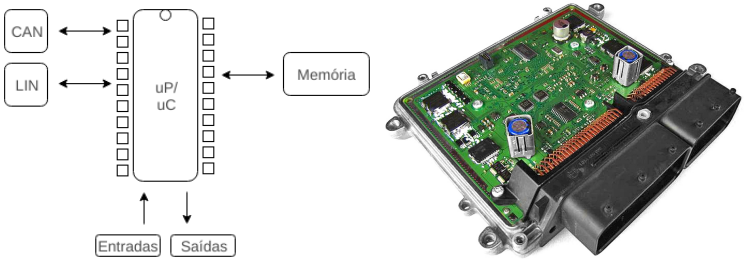
[11]Traduzido como: Receptor/Transmissor Universal Assíncrono.

2.4 UNIDADE DE CONTROLE ELETRÔNICA

Conforme Guimarães (2007), as unidades de controle eletrônicas são os dispositivos responsáveis por fazer a leitura de sensores, o acionamento de saídas, a comunicação entre módulos e pelo gerenciamento do funcionamento de sistemas. Uma ECU pode ser responsável por um ou mais sistemas em um veículo, geralmente pertencentes a um mesmo domínio.

Fisicamente, a ECU pode ser comparada a um computador. Geralmente são compostas por um microprocessador ou microcontrolador, memórias e portas de entrada e saída soldadas em uma placa de circuito integrado. Também possuem *transceivers* para redes CAN e LIN, para que as ECUs possam trocar informações entre sí e entre sensores e atuadores. Uma representação lógica e física de uma ECU pode ser vista na Figura 12.

Figura 12: Modelos lógico e físico de uma ECU.



(a) Lógico (b) Físico

Adaptado de: Guimarães (2007).

As quantidade de entradas e saídas disponíveis está associada ao microprocessador/microcontrolador utilizado, e estas podem ser digitais ou analógicas, sendo que as saídas também podem ser do tipo *Low Side Driver* (LSD)[12] ou *High Side Driver* (HSD)[13].

Em um veículo, existem diversas ECUs, ligadas por uma ou mais redes CAN *bus*. Dependendo da quantidade e distribuição no veículo, podem ser que diversas ECUs pertençam a um mesmo domínio e que uma delas sirva como uma central para as demais, ou ainda que uma ECU seja responsável por repassar informações de uma rede CAN para outra.

[12] Traduzido como: Driver do Lado Inferior.
[13] Traduzido como: Driver do Lado Superior.

Como cada ECU deverá executar um algoritmo específico, o qual varia conforme a função para a qual a mesma foi designada, surge a necessidade de que haja um gerenciamento dos vários módulos espalhados pelo veículo. Esta responsabilidade recai em uma das ECUs, a qual faz uso de um sistema operacional padronizado, que é capaz de se comunicar com as demais unidades de maneira intermitente e que dê prioridade para recursos de maior importância.

2.5 PADRÕES DE DESENVOLVIMENTO DE SOFTWARE AUTOMOTIVO

Com o crescente uso de componentes eletrônicos embarcados nos veículos automotivos, também surgiu a necessidade de padronizar as etapas do desenvolvimento de novos sistemas, afim de garantir a interoperabilidade e a intercambialidade entre eles.

Serão apresentados um padrão relacionado a engenharia de software e outro dedicado a implementação de sistemas veiculares.

2.5.1 Modelo V

Segundo CERTIFICATION (2012), o Modelo V, ou modelo de Verificação e Validação, pode ser considerado uma extensão do modelo Cascata, onde cada etapa deve ser concluída antes de avançar para a próxima. Ao invés de se deslocar para baixo de maneira linear, os passos do processo são curvados para cima após concluida a fase de codificação, tomando formato de um V, conforme ilustra a Figura 13.

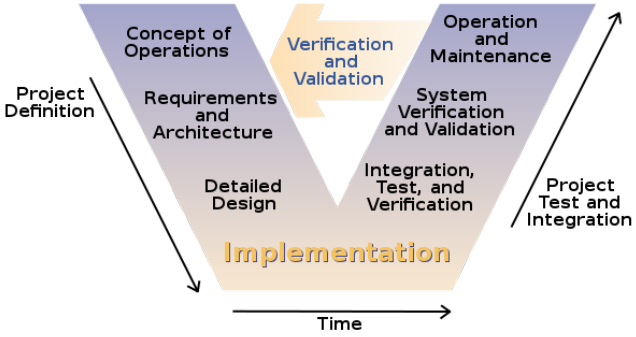
Como vantagens, trata-se de um modelo de fácil utilização, que incentiva a criação de cenários de testes antes mesmo da produção do código. Permite também que defeitos sejam encontrados em fases iniciais e apresenta bom resultados em projetos de pequeno porte.

Entre suas desvantagens, pode-se destacar que trata-se de um modelo pouco flexível. A produção de código somente tem seu início na fase de implementação, tornando impraticável a criação de um protótipo.

2.5.2 OSEK/VDX

Conforme Lee (2012), o padrão OSEK/VDX foi idealizado como uma arquitetura aberta para ECUs veiculares. Ele surgiu a partir

Figura 13: Modelo V.



Extraido de: Transportation (2005).

da união dos padrões ***O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik in **K**raftfahrzeugen* (OSEK)[14], criado por um consórcio de fabricantes de veículos alemãs, com o padrão ***V**ehicle **D**istributed e**X**ecutive* (VDX)[15], criado pelas empresas francesas Renault e Peugeot.

A principal motivação para a criação deste padrão se decorreu por conta do número cada vez maior de sistema eletrônicos nos veículos. Com a introdução do padrão, problemas recorrentes puderam ser corrigidos, como a incompatibilidade entre ECUs de diferentes fabricantes, bem como a contenção de custos com o desenvolvimento de protocolos para as ECUs.

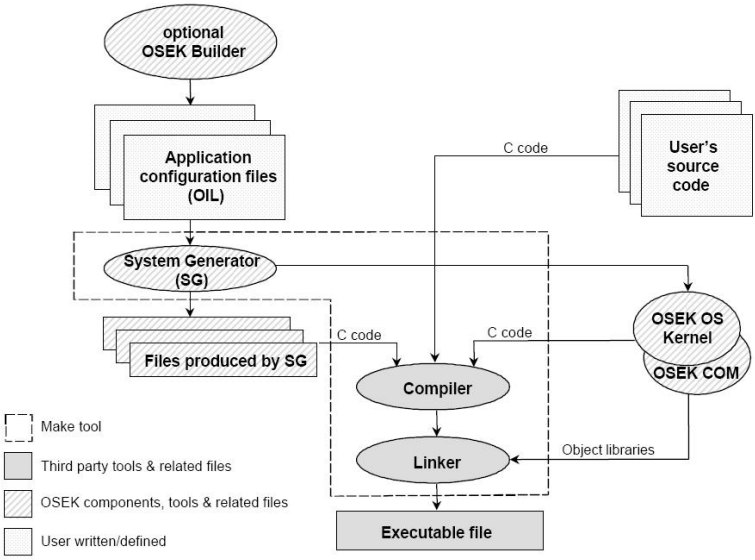
O padrão utiliza uma abordagem estática para as configurações do sistema, forçando com que todas as configurações sejam definidas antes da inicialização do sistema. Isto impede que novas tarefas sejam criadas ou que a memória seja alocada dinamicamente durante a execução do programa.

Para auxiliar na definição destas configurações, o padrão OSEK/VDX utiliza um arquivo de configuração com uma linguagem própria, chamada de *OSEK Implementation Language* (OIL)[16]. Uma vez definido, o arquivo é então processado para gerar o código em C de configuração do *Real Time Operating Systems* (RTOS)[17]. A Figura 14 provê uma

[14]Traduzido como: Sistemas abertos e suas interfaces para eletrônicos em motores veículares.
[15]Traduzido como: Veículo distribuído executivo.
[16]Traduzido como: Linguagem de Implementação do OSEK.
[17]Traduzido como: Sistema Operacional de Tempo-Real.

ilustração do processo de geração envolvendo o arquivo de configuração OIL.

Figura 14: Processo de geração do código em C a partir de um arquivo OIL.



Extraido de: Sciences (2016).

O padrão conta com sete especificações, cada qual atendendo a uma área diferente do sistema, sendo elas:

- **OSEK OS**: Sistema operacional;
- **OSEK Time**: Sistema operacional ativado por tempo;
- **OSEK COM**: Serviços de comunicação;
- **OSEK FTCOM**: Comunicação com tolerância a falhas;
- **OSEK NM**: Gerenciamento de rede;
- **OSEK OIL**: Configuração do kernel;
- **OSEK ORTI**: Configuração do kernel com suporte a debuggers.

Por ser um padrão mais antigo e restrito, atualmente existem soluções comerciais de código aberto, dentre elas: FreeOSEK[18], ERIKA Enterprise[19] e Trampoline[20].

Existem também soluções exclusivas para a melhoria da usabilidade da linguagem OIL, como visto em Macher et al. (2014), aonde são apresentadas novas ferramentas que permitem a extração de informações do arquivo OIL e vice-versa, bem como um modelo integrado com *Unified Modeling Language* (UML)[21].

2.5.3 AUTOSAR

Confome AUTOSAR (2016), o projeto AUTOSAR surgiu da cooperação entre manufaturadores e fornecedores de veículos e industrias de componentes eletrônicos e de software. Desde sua concepção, em 2003, foi idealizado como um projeto aberto, focado na padronização de arquiteturas de software para a indústria automotiva. O AUTOSAR também incorpora a maior parte do OSEK/VDX, expandindo suas funcionalidades mas mantendo a compatibilidade. Sua arquitetura e módulos básicos podem ser visualizados na Figura 15.

A motivação por trás do desenvolvimento do padrão AUTOSAR visava atender as otimizações a nível de sistema, e não apenas em componentes individuais, obtendo assim a máxima reutilização de código possível. Para tal, uma arquitetura aberta, escalar e de módulos de software intercambiáveis era necessária, exigindo um esforço coletivo do consórcio de empresas envolvidas.

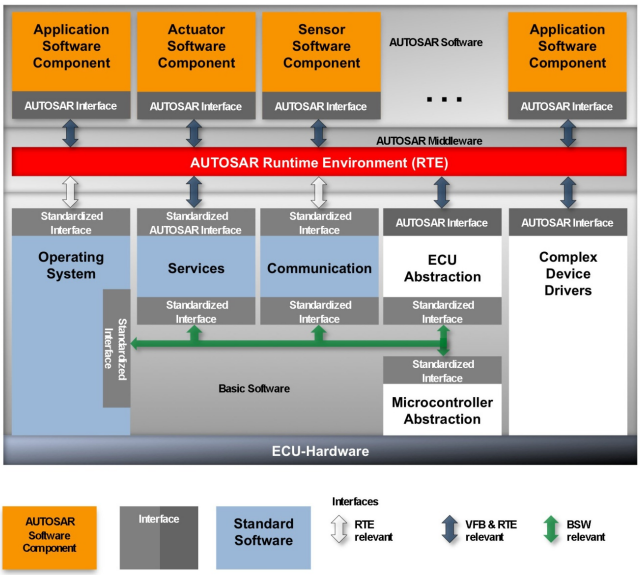
O principal objetivo do AUTOSAR é a padronização de funções básicas de sistemas e interfaces funcionais, garantindo a integração, intercambialidade e transferência de funcionalidades em uma rede veicular entre os parceiros de desenvolvimento, além de permitir que novas aplicações sejam agregadas durante o ciclo de vida do veículo.

Para cumprir seus objetivos técnicos de prover uma infraestrutura comum de software para sistemas automotivos, quatro características foram tidas como essenciais, sendo elas:

- **Modularidade**: permite que partes de código sejam agregados conforme requerimento do software;
- **Escalabilidade**: promove a adaptação de módulos de software

[18] https://github.com/ciaa/Firmware/
[19] http://erika.tuxfamily.org/drupal/
[20] http://trampoline.rts-software.org/
[21] Traduzido como: Linguagem de Modelagem Unificada.

Figura 15: Visão geral técnica do AUTOSAR.



Extraido de: AUTOSAR (2016).

comuns para diferentes veículos;

- **Transferibilidade**: otimiza o uso de recursos disponíveis através da arquitetura eletrônica de um veículo;
- **Reusabilidade**: aumenta a qualidade do produto, utilizando códigos já testados.

3 SISTEMAS OPERACIONAIS EMBARCADOS

Neste capítulo são abordados os principais conceitos sobre sistemas operacionais. Ao final do capítulo são relatados alguns estudos de caso a respeito de sistemas operacionais embarcados com foco para a automação veicular.

3.1 SISTEMAS OPERACIONAIS

O objetivo de um SO é o de gerenciar os recursos de um sistema computacional, tornando transparente seu funcionamento para o usuário final. O SO é o componente de software que faz a união e abstração dos recursos de hardware e os oferece de maneira simplifica para as aplicações utilizadas pelo usuário final. A Figura 16 ilustra a visão abstrata de um SO, ou seja, o SO como um provedor de serviços para as aplicações de usuários.

Figura 16: Visão geral da integração do SO com o sistema computacional.

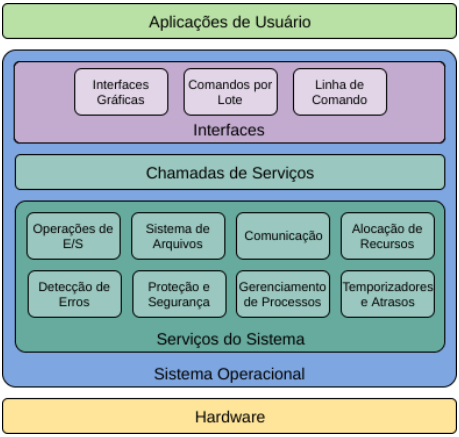
Adaptado de: Silberschatz, Galvin e Gagne (2012).

Segundo Silberschatz, Galvin e Gagne (2012), um SO pode oferecer um número variável de serviços. Os serviços do SO que estão sempre presentes na memória principal fazem parte do kernel, que é descrito na Seção 3.1.1.

Para que os aplicativos tenham acesso aos serviços oferecidos sem que haja um comprometimento de sua integridade, o SO oferece rotinas de acesso aos serviços. A quantidade de serviços varia conforme implementação do SO, sendo o mínimo oferecido as operações de gerenciamento de processos, memória e Entrada e Saída (E/S). A Figura

17 apresenta uma hierarquia de chamada dos serviços do SO.

Figura 17: Serviços de um SO.

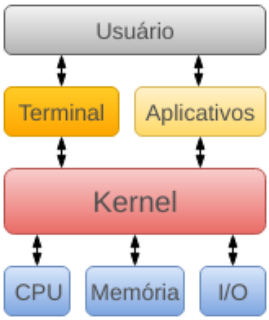


Adaptado de: Silberschatz, Galvin e Gagne (2012).

3.1.1 Kernel

Group (2016) define o *kernel* como sendo o programa que constitui o núcleo central de um sistema operacional. Ele é responsável por fazer o gerenciamento dos recursos de hardware bem como sua abstração para os aplicativos do usuário. A Figura 18 ilustra onde se encontra a camada de abstração do *kernel*.

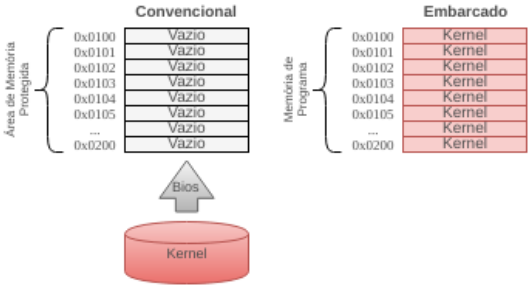
Figura 18: Posição conceitual do *kernel*.



Extraido de: Tanenbaum e Woodhull (2006).

Em sistemas computacionais convencionais, o *kernel* é a primeira parte do SO a ser carregada na memória durante o processo de inicialização. Uma vez carregado, o *kernel* permanece na memória principal do sistema até que o mesmo seja desligado. Para Sistemas Operacionais Embarcados (SOE), o *kernel* sempre está presente na memória de programa. Após uma instrução de *reset*, é sempre o primeiro código a ser executado. A Figura 19 ilustra ambos os modelos.

Figura 19: Posição do *kernel* na memória.



Adaptado de: Li (2003).

3.2 ESTRUTURA DOS SISTEMAS OPERACIONAIS

Independente dos serviços oferecidos, um SO pode ser separado em quatro serviços principais: gerenciamento de processos, de memória, de entrada e saída de dados e sistemas de arquivos.

3.2.1 Gerenciamento de Processos

O gerenciamento de processos consiste no tratamento de interrupções, controle de tarefas e acesso a recursos do sistema, de uma maneira que nenhum dos processos entre em conflito ou pare sua execução permanentemente, que não de maneira espontânea.

A quantidade de elementos que compõem um SO varia conforme implementação, mas geralmente incluem: tarefas, semáforos, filas de mensagem, entre outras.

3.2.1.1 Tarefas

Em um SO convencional, cada programa que é executado no computador toma a forma de um processo, que pode ser definido como uma atividade que possui sua própria pilha, área de memória privada, e que pode alocar memória dinamicamente conforme a necessidade, muitas vezes sem precisar se preocupar com a falta do recurso. Mais programas podem ser adicionados a qualquer momento em um SO para computadores.

Em sistemas embarcados, o conceito de processo é substituído pelo de *tarefas*, onde a principal diferença está no fato de que todas as tarefas ja estão definidas no momento em que o SO inicia. Para criar novos tipos de tarefas é necessário parar o SO e adicioná-las manualmente.

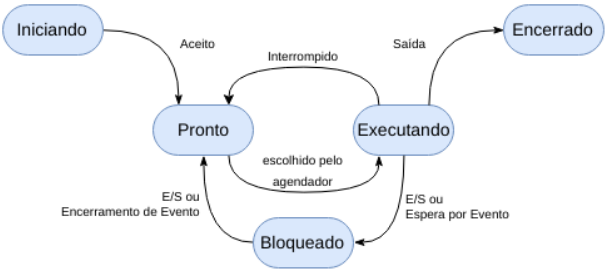
Uma tarefa pode assumir um número de estados pré-determinado, os quais variam conforme o SO utilizado, mas geralmente incluem os estados presentes na Figura 20, conforme visto em Silberschatz, Galvin e Gagne (2012). Estes estados podem ser descritos como:

- **Iniciando**: etapa inicial, onde são executadas instruções de preparo para a tarefa;
- **Pronto**: indica que a tarefa está pronta para ser executada, mas que ela não é a tarefa ativa, no momento;
- **Executando**: estado da tarefa que está em execução no SO;
- **Bloqueado**: a tarefa está aguardado a ocorrência de algum evento externo para que ela possa voltar a executar;
- **Finalizado**: a tarefa libera recursos alocados, antes de ser encerrada definitivamente.

3.2.1.1.1 Fibras

Uma variante das tarefas são as fibras, que são threads de execução leves e não-preemptivas, normalmente utilizadas para executar porções de código responsáveis pelos drivers de dispositivos e outras atividades consideradas de desempenho crítico (ROCKET, 2015).

Figura 20: Estados de uma tarefa.



Extraido de: Silberschatz, Galvin e Gagne (2012).

3.2.1.2 Escalonadores de Tarefas

O escalonamento de tarefas surgiu para permitir que mais de uma tarefa fique em execução no processador, sem que ela tenha que concluir seu funcionamento para que isso aconteça. Na prática, isso gera uma situação na qual várias tarefas aparentam estar em execução ao mesmo tempo, em arquiteturas com um único processador.

Tanenbaum e Woodhull (2006) descreve alguns tipos de algoritmos para escalonamento. Serão apresentados aqui apenas os escalonadores mais utilizados em RTOSes, sendo eles Round Robin, Prioridade e Prioridade com Round Robin.

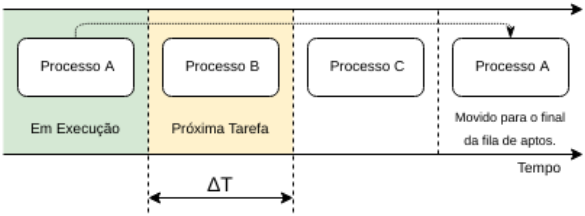
3.2.1.2.1 Escalonador Round Robin

No escalonador Round Robin, uma tarefa permanece em execução apenas por um valor ΔT de tempo, geralmente na casa dos *us*, após a qual ela é movida para o estado de *Pronto*, colocando a próxima tarefa na fila em execução, conforme ilustrado na Figura 21.

3.2.1.2.2 Escalonador por Prioridade

Neste tipo de escalonador, a tarefa é associada a um atributo numérico, o qual indica o nível de sua prioridade. A tarefa de mais alta prioridade que não estiver bloqueada será sempre a que estará em execução no SO. Ela permanece em execução até que se auto-bloquei, seja por uma rotina de *delay*, por tentar acessar um recurso indisponível no momento, ou ainda porque uma tarefa de maior prioridade ficou

Figura 21: Diagrama esquemático de funcionamento do algoritmo de escalonamento Round Robin.

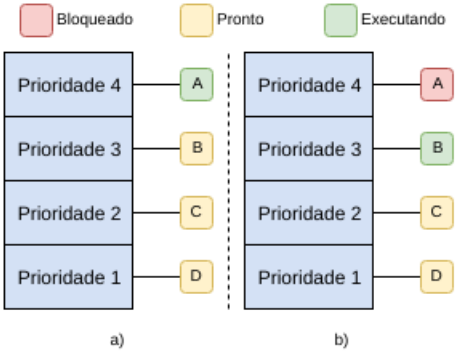


Extraido de: Tanenbaum e Woodhull (2006).

disponível.

A Figura 22 apresenta um escalonador por prioridade em dois momentos. No momento *(a)*, a tarefa em execução chama uma rotina que faz seu auto-bloqueio. No instante de tempo *(b)*, a tarefa em execução passa a ser a próxima tarefa de prioridade mais alta, disponível naquele momento.

Figura 22: Diagrama esquemático de funcionamento do algoritmo de escalonamento baseado em prioridade.



Extraido de: Tanenbaum e Woodhull (2006).

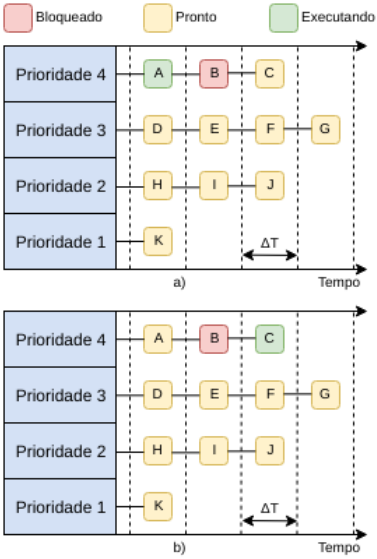
3.2.1.2.3 Escalonador por Prioridade com Round Robin

Faz uma junção dos dois tipos de escalonadores apresentados anteriormente. Funciona da mesma maneira que o escalonador de prioridades, exceto que agora é possível ter mais de uma tarefa com o

mesmo nível de prioridade. Quanto este cenário ocorrer, as tarefas de mesma prioridade ficarão alternando a posição de *em execução*, através do algoritmo Round Robin.

A Figura 23 apresenta um exemplo deste escalonador, onde após um intervalo de tempo ΔT, ocorre a troca de contexto para a próxima tarefa de mesma prioridade que não se encontra bloqueada.

Figura 23: Diagrama esquemático de funcionamento do algoritmo de escalonamento baseado em prioridade com Round Robin.



Extraido de: Tanenbaum e Woodhull (2006).

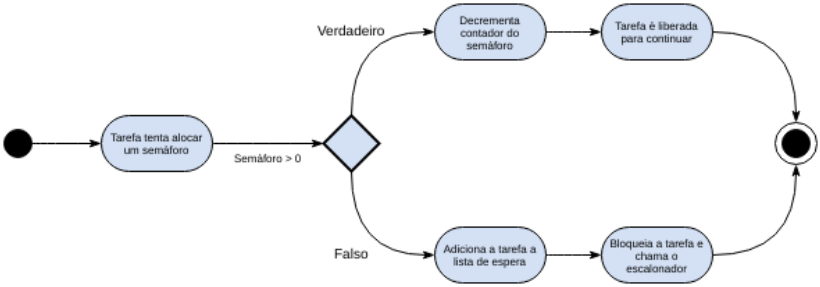
3.2.1.3 Semáforos

Múltiplas tarefas em execução concorrente devem ser capazes de sincronizar suas ações e coordenar o acesso mutuamente exclusivo a recursos compartilhados. Para atender a estes requisitos, o SO pode prover um objeto chamado semáforo.

Um semáforo funciona como uma espécie de chave que permite a uma tarefa acessar algum tipo de operação ou recurso. Se a tarefa puder adquirir um semáforo, ela poderá continuar sua execução normalmente. Do contrário, a tarefa poderá ser bloqueada até que o recurso esteja disponível novamente.

Segundo Tanenbaum e Woodhull (2006), existem semáforos de contagem, binários e de exclusão mútua. Semáforos de contagem e binários apresentam comportamentos similares, tendo como única diferença que um semáforo binário possui seu valor máximo igual a 1. Um diagrama de atividades, demostrando o funcionamento de um semáforo pode ser visto na Figura 24.

Figura 24: Transição de Estados para Semáforos de Contagem.



Adaptado de: Li (2003).

3.2.1.3.1 Mutex

O semáforo do tipo *mutex* atende a um caso especial do semáforo binário, chamado de semáforo de exclusão mútua ou *mutex*. Um *mutex* pode suportar propriedades de posse, trava recursiva, deleção segura de tarefas, dentre outros serviços, afim de evitar problemas inerentes a exclusão mútua.

3.2.1.4 Filas de Mensagens

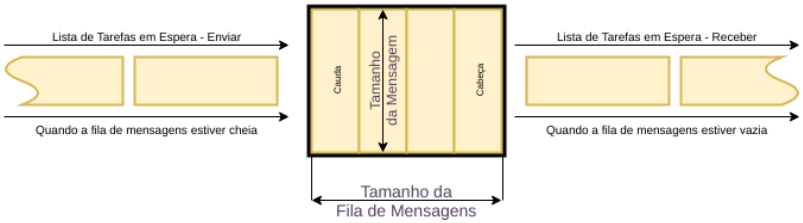
Uma fila de mensagens é um objeto através do qual as tarefas e *Interrupt Service Routine* (ISR)[1] enviam e recebem mensagens para comunicação e sincronização de dados. Elas armazenam temporariamente as mensagem do remetente até que o destinatário esteja pronto para recebe-las. Isto garante o desacoplamento entre a tarefa emissora e receptora.

Uma fila de mensagens é composta por objetos chamados de *elementos*, dos quais cada um pode armazenar uma única mensagem,

[1] Traduzido como: Rotina de Serviço para Interrupções.

a qual pode estar vazia. O número total de elementos na fila equivale ao seu comprimento. A Figura 25 apresenta um esquema para as filas de mensagem.

Figura 25: Diagrama esquemático do funcionamento de uma fila de mensagens..



Adaptado de: Li (2003).

3.2.1.5 Sinais

Um sinal é uma interrupção gerada por software, a qual dispara quando ocorre um evento. Assim como numa interrupção, um sinal faz com que o processo em execução seja interrompido para executar uma outra rotina assíncrona.

Na essência, os sinais notificam as tarefas de eventos que ocorreram durante a execução de outras tarefas ou ISRs. Assim como as interrupções, estes eventos são assíncronos para a tarefa notificada e não ocorrem em nenhum ponto pré-determinado durante sua execução. A principal diferença entre uma interrupção e um sinal é que o primeiro é gerado por hardware, como quando um pino de um microcontrolador passa de $0V$ para $5V$, enquanto o último é gerado por software.

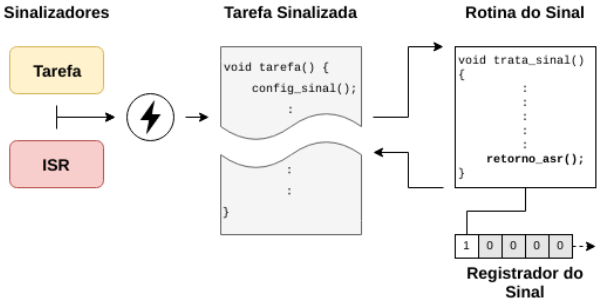
Quando há a chegada de um sinal, a tarefa desvia de seu fluxo normal de execução, e a *Asynchronous Signal Routine* (ASR)[2] correspondente é invocada, conforme ilustrado na Figura 26.

3.2.2 Gerenciamento de Memória

Em muitos dos sistemas embarcados (tais como celulares, câmeras digitais, *tablets*) há um número limitado de aplicações (tarefas) que podem estar em execução em um dado momento. Parte do motivo

[2]Traduzido como: Rotina de Sinal Assíncrona.

Figura 26: Tratamento de sinal em uma tarefa.



Adaptado de: Li (2003).

é que estes dispositivos apresentam um quantidade limitada de memória física. Dispositivos maiores, como roteadores de rede, possuem uma maior quantidade de memória física instalada, mas também fazem maior uso dela e precisam de um gerenciamento ainda maior deste recurso.

Independente do tipo de sistema embarcado, os requisitos comuns a sistemas de gerenciamento de memória são a mínima fragmentação, mínima sobrecarga em operações de gerenciamento e tempos de alocação determinísticos.

3.2.2.1 Alocação Dinâmica de Memória

O código do programa, seus dados e a pilha do sistema ocupam a memória física do sistema embarcado, uma vez carregados e inicializados. A memória restante é utilizada para alocação dinâmica pelo RTOS ou pelo *kernel*. Esta área recebe o nome de *heap*[3].

Um gerenciador de memória mantém informações sobre a *heap* em uma área reservada, chamada de bloco de controle. Informações típicas sobre o controle incluem:

- O endereço inicial do bloco de memória física utilizado para alocação dinâmica;
- O tamanho total de memória disponível;
- A tabela de alocações, a qual indica quais áreas da memória estão

[3]Tanto a *Stack* quanto a *Heap* são áreas de memória para um programa. A diferença entre elas é que a sua alocação é, respectivamente, estática e dinâmica

em uso, quais estão vagas e o tamanho de cada região que ainda está livre.

Um sistema de memória deve ser capaz de executar eficientemente as seguintes operações:

- Determinar se há um bloco livre que comporta a alocação requisitada;
- Manter as informações internas atualizadas;
- Mesclar um ou mais blocos, assim que estes forem liberados.

A estrutura da tabela de alocação é a chave para um gerenciamento de memória eficaz. Esta estrutura gera um *overhead*, já que ocupa espaço de memória que, outrora, poderia ser utilizado para armazenar dados dos programas. Minimizar a tabela de alocações e maximizar o desempenho das operações anteriores é um dos principais desafios no gerenciamento de memórias.

3.2.3 Subsistemas de Entrada/Saída

Em sistemas embarcados, um sistema de entrada/saída é a combinação dos dispositivos de E/S com os *drivers* de dispositivos associados e subsistemas de E/S.

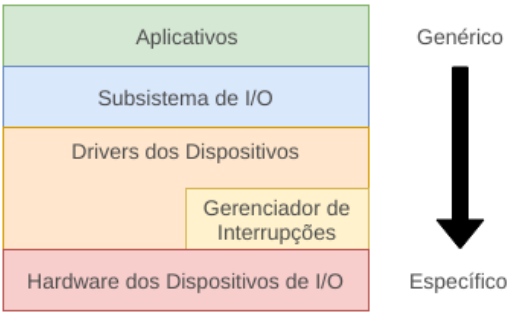
O propósito de um subsistema de E/S é o de esconder do *kernel* as informações especificas de um dispositivo, assim como do desenvolvedor de aplicações, e prover uma método de acesso uniforme aos periféricos de E/S do sistema embarcado.

A Figura 27 ilustra o subsistema de E/S em relação ao resto do sistema em um modelo de camadas de software. Conforme indicado, cada camada descendente agrega mais informações à arquitetura necessária para gerenciar um dado dispositivo.

3.2.3.1 Camada de Abstração

Cada dispositivo de E/S pode oferecer um conjunto específico de interfaces de programação para os aplicativos. Este arranjo requer que cada aplicativo esteja ciente da natureza do dispositivo de E/S subjacente, incluindo as restrições impostas pelo dispositivo. O conjunto da

Figura 27: Subsistema de E/S e o modelo por camadas.



Adaptado de: Tanenbaum e Woodhull (2006).

Application Programming Interface (API)[4] é específico à implementação, o que torna difícil a portabilidade das aplicações utilizando esta API. Para reduzir esta dependência, é implementado no sistema embarcado um subsistema de E/S, a qual atua como uma camada de abstração.

Esta camada de abstração define um conjunto padrão de funções para operações de entrada e saída, de forma a esconder as peculiaridades dos dispositivos da aplicação. Todos os *drivers* de E/S passam a se conformar e a suportar este conjunto de funções, já que o objetivo é o de prover uma camada uniforme de E/S para as aplicações.

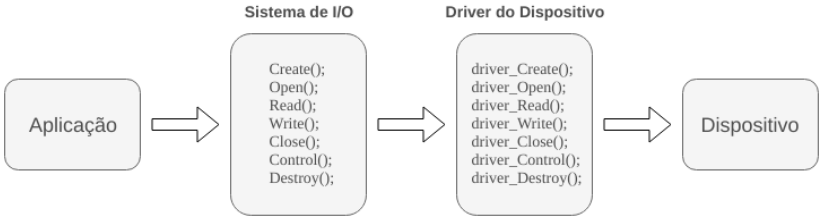
Para se alcançar estas operações de E/S uniformemente no nível de aplicação, os seguintes procedimentos devem ser seguidos:

1. Definir o conjunto de APIs do subsistema de E/S;
2. Implementar cada função do conjunto para o driver do dispositivo;
3. Exportar este conjunto de funções do driver do dispositivo para o subsistema de E/S;
4. Encarregar ao driver do dispositivo o prepara do mesmo para uso;
5. Carregar o dispositivo pelo driver do mesmo e informar o subsistema de E/S.

A Figura 28 ilustra como a camada de E/S abstrai o dispositivo de hardware, garantindo a flexibilidade do sistema.

[4]Traduzido como: Interface de Programação de Aplicação.

Figura 28: Camada de abstração entre o aplicativo e o dispositivo.



Adaptado de: Li (2003).

3.2.4 Sistemas de Arquivos

Segundo Tanenbaum e Bos (2014), um arquivo é uma coleção nomeada de informações relacionadas que são gravadas em uma unidade secundária e persistente de armazenamento. Pela perspectiva do usuário, dados não podem ser salvos em uma unidade secundária, senão pela alocação de um arquivo nela.

Os arquivos são utilizados para guardar informações referentes a dados ou programas. Arquivos de dados podem ser numéricos, alfabéticos, alfanuméricos ou binários. Eles podem também ser estruturados ou não. De maneira geral, um arquivo é uma sequência de bits, no qual o significado varia conforme o criador e usuário do arquivo.

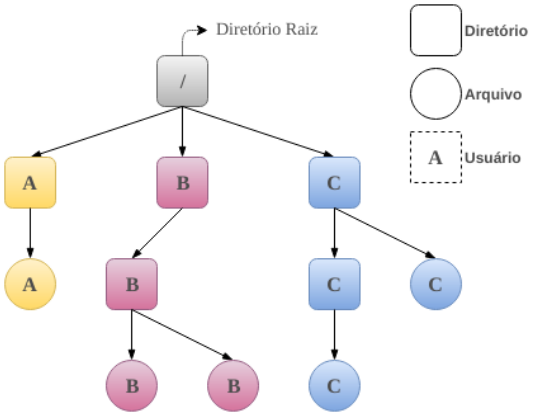
Existem diversos componentes capazes de persistirem estes dados, sendo alguns deles discos e fitas magnéticas, *flash cards* e *Compact Disks*. Para que que o sistema computacional possa utilizar estes dispositivos, o SO deve abstrair as propriedades físicas dos dispositivos de armazenagem e definir uma unidade lógica de armazenagem, capaz de armazenar e localizar facilmente os dados em forma de arquivos.

Um sistema de arquivos provê os meios para organizar os dados, de forma que eles possam ser armazenados, resgatados e atualizados, além de gerenciar o espaço disponível no dispositivo que o contém. Um sistema de arquivos organiza os dados de maneira eficiente e é otimizado para características específicas de um dispositivo (DALEY; NEUMANN, 2016).

Internamente, um sistema de arquivos funciona de maneira similar a alocação dinâmica da memória principal, onde o sistema armazena informações adicionais sobre a área de dados, como seu tamanho e ponto de origem no dispositivo. Contudo, um sistema de arquivo oferece mais níveis de organização, afim de facilitar o encontro de informações posteriormente, como diretórios, ou ainda controle de permissões,

para limitar o acesso de usuários. A Figura 29 apresenta um modelo de organização de diretórios com controle de permissão.

Figura 29: Modelo em árvore de diretórios por usuário.



Adaptado de: Tanenbaum e Woodhull (2006).

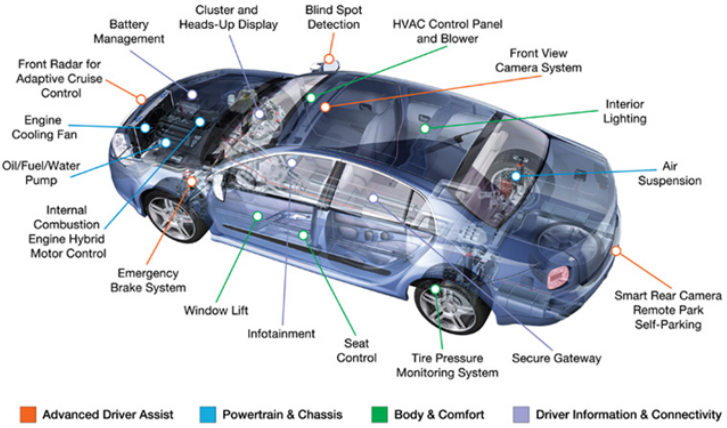
3.3 SISTEMAS EMBARCADOS

Um sistema embarcado é um sistema computacional cujo propósito é bem definido e específico. Ele é geralmente conceituado para atender a uma situação particular, com hardware bem definido e, muitas vezes, imutável.

Diferente de um sistema computacional convencional, como os *Desktops* e *Laptops*, um sistema embarcado possuí recursos de hardware bem mais restritos. Por ser projetado para um propósito específico, o mesmo muitas vezes não requer tecnologia de ponta para executar sua tarefa, o que permite a criação de sistemas de baixo custo.

Um sistema embarcado pode ser composto, por exemplo, de sensores e atuadores, bem como uma central de processamento. Em veículos, um exemplo equivalente seria o dos sensores e atuadores para travamento das portas, bem como a central eletrônica responsável por sua operação. A Figura 30 ilustra diversos sistemas embarcados presentes em um automóvel.

Figura 30: Sistemas embarcados em um veículo.



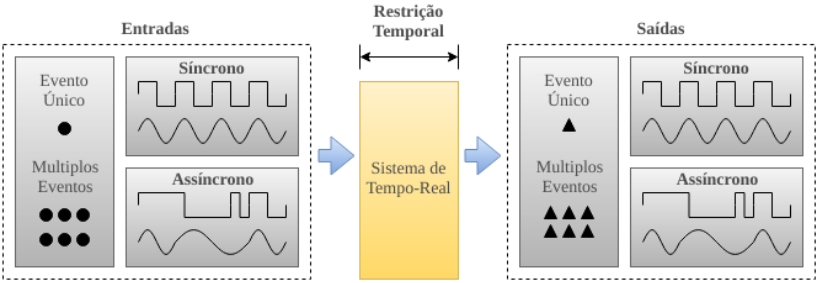
Extraido de: VIZCAYNO (2015).

3.4 SISTEMAS DE TEMPO-REAL

Segundo Li (2003), uma aplicação de tempo-real é aquela que é composta por múltiplas tarefas independentes em execução, as quais competem pelo tempo de processamento em um processador.

Um sistema de tempo-real pode ser definido como um sistema que responde a eventos externos em tempo hábil. Ou seja, onde o tempo de resposta é uma garantia do sistema. A Figura 31 ilustra o conceito de sistema de tempo real, onde independente da entrada, há um tempo limite de resposta, o qual deverá sempre ser respeitado.

Figura 31: Resposta em Tempo-Real.



Adaptado de: Li (2003).

Uma restrição temporal pode ser rígida ou flexível. Restrições rígidas causam uma falha do sistema quando não respeitadas, geralmente causando danos físicos ao equipamento e possível risco humano. Um sistema de restrição flexível não sofre consequências tão graves quando o mesmo falha, mas ainda assim pode causar invalidação do sistema ou de processos. Um veículo possui diversos sistemas embarcados, muitos dos quais possuem restrições temporais, tanto rígidas quanto flexíveis.

3.5 ESTUDOS DE CASO

A seguir são apresentados alguns estudos de casos sobre sistemas operacionais embarcados já existentes no mercado.

3.5.1 FreeRTOS

Conforme FreeRTOS (2016), o FreeRTOS é uma solução padronizada para sistemas operacionais embarcados. Por separar os códigos do sistema operacional dos códigos específicos para a arquitetura do sistema embarcado, ele é considerado um SO extremamente portável, sendo disponibilizado nas principais plataformas do mercado.

Dentre as características oferecidas pelo SO, podem-se destacar:

- Escalonador de tarefas baseado em prioridades com *Round Robin*;
- Ocupa pouco espaço da memória do microcontrolador;
- Oferece recursos como semáforos, filas de mensagens e co-rotinas;
- Timers e interrupções por softwares.

Sua interface é resumida a três arquivos de código fonte. Os demais arquivos atendem a adaptação para o hardware alvo.

Em desenvolvimento a mais de 12 anos, o FreeRTOS é o SOE mais utilizado no mercado. Além de sua versão de código aberto, ele também possui licenças para versões comerciais (CLARKE, 2013).

3.5.2 PICOS18

Conforme Softelec (2002), o PICOS18 é considerado um kernel de tempo-real. Ele surgiu como uma implementação de um SOE para a arquitetura PIC18, utilizando o compilador C18 da Microchip®.

Embora descontinuado, se destaca por ser uma das primeiras soluções de código aberto para sistemas operacionais dedicados a automação veicular, apresentando uma implementação não-homologada da norma OSEK/VDX.

3.5.3 Trampoline

Segundo Béchennec e Faucou (2009), o Trampoline[5] é uma plataforma aberta para sistemas embarcados de pequeno porte com restrições de tempo-real. Sua inspiração veio incialmente do padrão OSEK/VDX e, atualmente, busca ser compatível com o padrão de SO do AUTOSAR.

Ele é composto por:

- Um kernel de tempo-real - trampoline;
- Uma ferramenta de configuração do kernel - GOIL;
- Um ambiente virtual para prototipação de aplicações em estações de trabalho.

Conforme especificado pelo padrão OSEK/VDK, a linguagem OIL é utilizada para gerar o código fonte em C para configuração do SO, neste caso, processada pela ferramenta GOIL, conforme ilustra a Figura 32. É possível observar também que o código do kernel é divido duas partes: uma genérica e a outra específica para a arquitetura alvo.

Por consequência do padrão AUTOSAR OS, todos os objetos de sistemas devem ser estáticos. Isto implica que threads, mutexes, semáforos, etc, devem ser declarados em tempo-de-compilação.

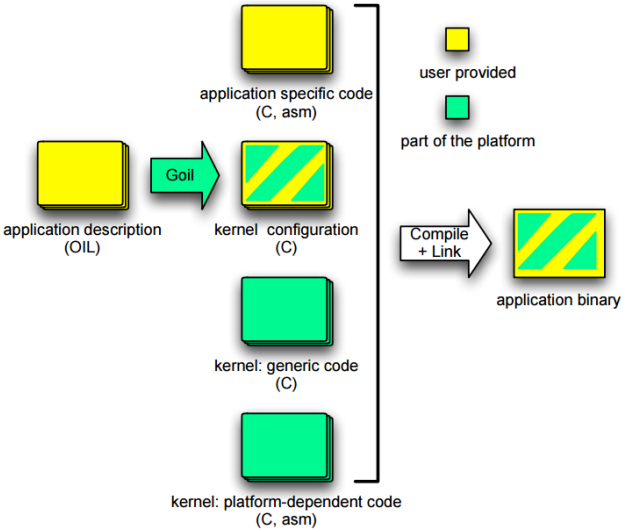
O foco do SO é voltado para hardwares de baixo custo, com especificações de:

- **Arquitetura**: entre 16 ou 32 bits;
- **Clock**: até 20MHz;
- **RAM**: até 32KB;
- **ROM**: até 128KB.

Sua implementação oferece suporte a modelos regido temporalmente e regidos por eventos, ambos necessários à premeditabilidade em tempo-real. Também oferece suporte para computações regidas por eventos.

[5] http://trampoline.rts-software.org

Figura 32: Processo de geração das configurações do kernel.



Extraido de: Béchennec e Faucou (2009).

3.5.4 SDVOS

Desenvolvido em 2015 pelo Dr. Yu Li, o SDVOS é um SO de código aberto, multi-plataforma e que atende ao padrão do OSEK/VDX, além de oferecer os recursos de tabelas de agendamento e temporizadores por software do AUTOSAR (LI, 2015).

Atualmente ele oferece suporte a múltiplas arquiteturas de microcontroladores e processadores, incluindo o AVR5 e ARMv7-M. Ele também pode ser executado como um processo em sistemas operacionais Linux.

4 OPENAUTOS

Este capítulo apresenta o SO OpenAUTOS, oferecendo uma visão geral sobre sua implementação, estrutura de arquivos, funcionamento do SO e sistemas implementados. São descritos em detalhes os algorítimo para troca de contexto, alocação de recursos nas tarefas e prevenção de deadlocks por prioridade-teto.

4.1 IMPLEMENTAÇÃO

A estratégia de implementação adotada para o OpenAUTOS, afim de que em um dado momento ele passe a conformar as normas do AUTOSAR, foi a de oferecer suporte as interfaces da norma do OSEK/VDX, na qual o AUTOSAR foi originado, já que elas também estão de acordo com o AUTOSAR OS.

O SO OpenAUTOS oferece suporte básico as sessões mais criticas de um SO, que são a declaração e instanciação dos componentes utilizados por ele, como tarefas e recursos, a troca de contexto entre as tarefas e um mecanismo para prevenção de deadlocks de sistema, alcançado através do algoritmo de prioridade-teto. O SO também conta com um parser para a linguagem OIL que, embora ainda não ofereça suporte a todas as instruções da linguagem, já se apresenta em um estado funcional.

O SO ainda foi projetado pensando em atender a múltiplas plataformas de microcontroladores, abstraindo e interfaceando as chamadas de funções críticas ao sistema, enquanto que os códigos específicos para as plataformas foram separados e chamados por compilação condicional. A plataforma utilizada como base para realização das compilações foi o Linux, com o compiladores `gcc`, versão 4.8.4, para compilações internas do SO e o compilador `XC8` da Microchip Technology Inc., versão 1.37, para geração do código da plataforma PIC18F25K80.

4.2 ESTRUTURA DE ARQUIVOS

A estruturação dos arquivos do OpenAUTOS foi inspirada no sistema SDVOS, buscando atender ao critério de que o código poderia ser facilmente portado para novas plataformas, além de fazer uma distinção e possível migração dos cabeçalhos da norma OSEK/VDX para

a norma AUTOSAR. A Figura 33 apresenta a estrutura das pastas do sistema.

Figura 33: Estrutura de Arquivos do OpenAUTOS.

A pasta raiz do OpenAUTOS contém um arquivo `make`, chamado de Makefile, o qual é responsável por iniciar a sequência de compilação do programa.

O diretório `app` representa a pasta do programa que irá executar no SO. Esta pasta deve conter dois arquivos, ambos escritos pelo usuário, descritos em mais detalhes na Seção 4.4.

Na pasta `build` serão gerados os arquivos binários compilados, bem como a imagem do SO pronta para ser gravada na memória de programa do microcontrolador.

No diretório `oiler` estão os fontes e executável do parser para linguagem OIL. Estes arquivos são automaticamente gerados e executados pelo comando make, no diretório raiz do OpenAUTOS. A Seção 4.5.1 apresenta mais detalhes sobre sua implementação.

Em `os`, estão os códigos-fonte referentes ao sistema operacional. Nesta pasta, estão contidas as estruturas de dados e funções que dão base para o funcionamento do SO. Também contém o arquivo de cabeçalho `openautos.h`, que contém as declarações de todas as funções que podem ser chamadas pela interface OSEK/VDX.

Na subpasta `osek` estão as interfaces e funções previstas pela norma do OSEK/VDX. Estes arquivos foram assim estruturados para que fique distinguível os elementos que pertencem ao OSEK/VDX e os que, num futuro próximo, pertencerão ao AUTOSAR OS, que também possuirão seu próprio diretório.

Finalmente, no diretório `platform` estão contidos toda a parte não-portável do código. Cada plataforma adotada pelo projeto deverá conter aqui uma pasta com o nome da arquitetura alvo, a qual con-

terá o código responsável por executar uma determinada tarefa nesta arquitetura. Os seguintes itens são exclusivos da plataforma alvo:

- Algoritmos para troca de contexto;
- Definição de algumas constantes[1];
- Rotinas de inicialização da plataforma;
- Rotina de interrupção;
- Contador padrão.

Como desvantagem, esta estrutura permite que apenas um programa seja compilado por vez. Como sugestão, a pasta `app` pode ser substituída por um atalho para a pasta que contenham os fontes do programa que se deseja compilar.

Esta estrutura esta sujeita a alterações futuras, principalmente quando o SO passar a oferecer mais funcionalidades.

4.3 MÓDULOS LÓGICOS DO SO

Além da divisão dos arquivos apresentada na Seção 4.2, houve também uma divisão do código por funcionalidades. A Figura 34 apresenta os principais módulos lógicos presentes no OpenAUTOS. Nela, existem três grupos maiores que indicam o nível de especialização do conjunto, sendo eles:

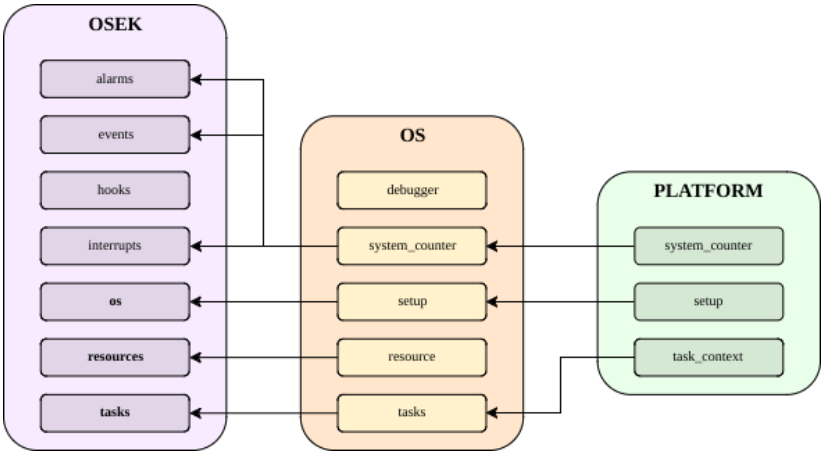
- **OSEK**: agrupam os módulos que são utilizadas para interfaceamento pela pessoa que desenvolve uma solução com o OpenAUTOS;
- **OS**: agrupa os módulos que oferecem o suporte as interfaces da norma do OSEK/VDX. Futuramente, oferecerá suporte, também, as normas do AUTOSAR;
- **PLATFORM**: agrupa os módulos que possuem códigos que são específicos para a plataforma alvo;

Ainda sobre a Figura 34, os módulos que a compõem são descritos como:

- `alarms`: implementa funcionalidades de alarmes do SO;

[1] À exemplo, valores para indicar se uma porta atua como INPUT ou OUTPUT.

Figura 34: Módulos Lógicos do OpenAUTOS.



- `events`: implementa funcionalidades de eventos do SO;
- `hooks`: implementa os métodos de pré e pós chamadas para algumas das rotinas do OSEK/VDX;
- `interrupts`: permite o registro de rotinas de interrupções do usuário no SO;
- `os`: possui as rotinas de inicialização do SO;
- `resources`: implementa funcionalidades de alocação e desalocação de recursos;
- `setup`: realiza a inicialização do SO e é a base para o módulo `os` do OSEK/VDX;
- `system_counter`: agrupa as rotinas que implementam o contador interno do SO. Serve como base para a implementação de funcionalidades nos módulos de `alarms`, `events` e `interrupts` do OSEK;
- `tasks`: implementa as rotinas para declaração e manipulação dos estados das tarefas, bem como para suas trocas contexto;
- `task_context`: implementa funcionalidades de troca de contexto para as tarefas, especificas para uma plataforma.

Dos módulos pertencetes ao grupo OSEK, apenas os que se encontram destacados em negrito estão implementados no sistema. Os demais serão implementados em versões futuras do OpenAUTOS.

4.4 FUNCIONAMENTO DO SO

Conforme mencionado na Seção 4.2, o OpenAUTOS requer que o usuário escreva dois arquivos na pasta `app`. Estes arquivos são:

- **program.oil**: é o arquivo de declarações e configurações dos recursos utilizados pelo SO, escrito de acordo com as normas da linguagem OIL. Um exemplo de arquivo OIL pode ser visto em Algoritmo 1;
- **program.d**: este arquivo corresponde a implementação das rotinas das tarefas declaradas em `program.oil`. Basicamente, envolve chamadas à macro TASK, definida nos cabeçalhos do OSEK/VDX. Um exemplo de um arquivo `program.d` pode ser visto em Algoritmo 2.

Algoritmo 1: Exemplo de arquivo `program.oil`.

```
1  CPU PIC_MASTER {
2      OS OpenAUTOS {
3          STARTUPHOOK = TRUE;
4          PLATFORM = PIC18F25K80;
5      };
6
7      TASK task1 {
8          PRIORITY = 1;
9          SCHEDULE = FULL;
10         ACTIVATION = 2;
11         AUTOSTART = FALSE;
12         RESOURCE = Resource1;
13     } "Tarefa de Testes";
14
15     RESOURCE Resource1 {
16         RESOURCEPROPERTY = STANDARD;
17     };
18 } "Plataforma para testes";
```

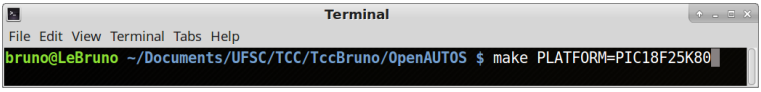
Algoritmo 2: Exemplo de arquivo `program.d`.

```
1  TASK(task1) {
2      while( TRUE ) {
3          // Allocate Resource
4          if( GetResource( Resource1 ) ) {
5              BlinkLed();
6              ReleaseResource( Resource1 );
7          }
8          Sleep(1000);
9      }
10 }
```

Uma vez que ambos os arquivos estejam devidamente escritos, basta que o usuário abra um terminal da plataforma base, navegue até a pasta raiz do OpenAUTOS e execute o comando `make`, conforme ilustrado pela Figura 35, passando para o parâmetro PLATFORM o valor da plataforma desejada[2]. Este comando executará os seguintes passos:

1. Compilar os códigos-fonte do parser `oiler`;
2. Gerar o arquivo `oiler.d` a partir do arquivo `program.oil`;
3. Combinar os arquivos `program.d`, `oiler.d` e `main.d` em um único arquivo `main.c` no diretório `os`. Uma ilustração deste processo pode ser vista na Figura 36;
4. Compilar os arquivos fontes da pasta `os` em arquivos de objeto na pasta `build`;
5. Linkar os arquivos de objeto da pasta `build` em um arquivo de imagem para a plataforma especificada, também localizado na pasta `build`.

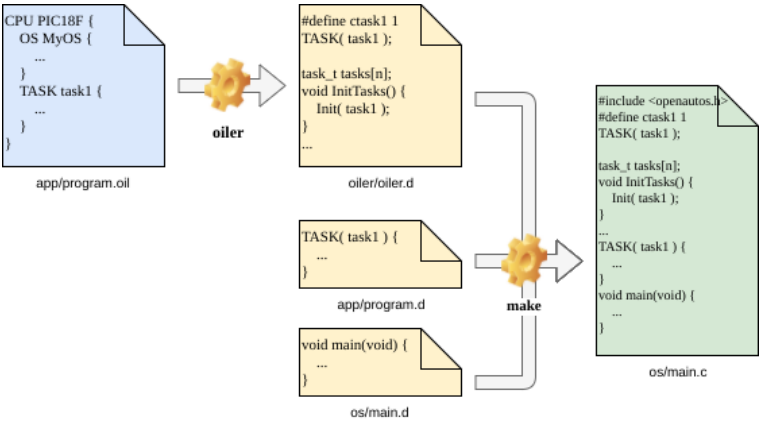
Figura 35: Comando Make pelo terminal.



Encerrado este processo, cabe ao usuário agora gravar a imagem gerada pelo OpenAUTOS na plataforma escolhida.

[2]No momento, o único valor suportado pelo OpenAUTOS é PIC18F25K80.

Figura 36: Geração do arquivo `main.c`.



4.5 SISTEMAS IMPLEMENTADOS

No projeto OpenAUTOS optou-se por iniciar a implementação a partir da norma do OSEK/VDX que, apesar de ser um projeto descontinuado, representa uma base compatível com o AUTOSAR, além de possuir um nível de complexidade bem mais acessível para um projeto iniciante. Infelizmente, este conjunto de normas se provou bastante extenso para a realização no intervalo proposto e uma parte significativa dos recursos por ela oferecido não puderam ser implementados ainda.

O trabalho focou unicamente nos documentos voltado a implementação do SO e da linguagem OIL, deixando para trabalhos futuros os documentos que falam sobre os protocolos de comunicação e tratamento de falhas.

Em seu estado atual, o OpenAUTOS possui um parser parcial para a linguagem OIL, funcionalidades para declaração de tarefas, trocas de contexto e alocação de recursos com uso de prioridade-teto. Na sequencia, estes tópicos serão abordados em maiores detalhes, explicando suas limitações conforme a norma, quando for o caso.

4.5.1 Oiler: parser para linguagem OIL

O parser para linguagem OIL, chamado de oiler, foi escrito utilizando as bibliotecas *flex* e *lemon*[3] para linguagem C. Para sua utilização, é necessário que a plataforma Linux base tenha instalado a biblioteca *flex*. Os fontes da biblioteca *lemon* já estão inclusos com o projeto.

O parser oiler reconhece todas as configurações para tarefas e recursos do OSEK/VDX, respectivamente declarados em arquivos OIL como TASK e RESOURCE. Infelizmente, algumas destas configurações ainda não são devidamente tratadas no OpenAUTOS, sendo elas:

- Em TASK, a opção SCHEDULE é sempre tratada como FULL, caso exista mais de uma tarefa com a mesma prioridade. Para tarefas marcadas como `SCHEDULE = NON`, estas tarefas não podem ser interrompidas pelo escalonamento por tempo;
- Em RESOURCE, a opção RESOURCEPROPERTY está sendo sempre tratada como STANDARD. É necessário ainda oferecer suporte aos valores LINKED e INTERNAL.

4.5.2 Constantes, Tipos e Compilação Condicional

Para manter a portabilidade do sistema, o OpenAUTOS faz amplo uso de constantes externas, as quais são definidas automaticamente no arquivo gerado `os/main.c`. Estas variáveis são responsáveis por especificar o tamanho de arrays globais ou outras constantes utilizadas pelo sistema.

Como o tipo primitivo `int` não possui tamanho definido na linguagem C, o OpenAUTOS faz uso da biblioteca `stdint.h`, quando disponível[4]. Desta forma, sempre é especificado o tamanho real da variável, buscando-se sempre utilizar o menor tamanho possível para cada variável. Conforme pode ser observado também no Algoritmo 4, membros de uma estrutura sempre tem seu tipo especificado por um `typedef`, afim de facilitar adaptações para novas plataformas no futuro.

Para seleção dos códigos da plataforma alvo, utilizou-se o mecanismo de compilação condicional da linguagem C. O Algoritmo 3

[3] Desenvolvido por Hipp (2000) como parte do gerenciador de banco de dados SQLite

[4] Em casos onde ela não se encontra disponível, é possível a utilização de `typedef`s para estabelecer os tamanhos padrões.

demonstra os casos mais comuns onde este recurso foi utilizado. Em síntese, ele é usado para fazer a seleção de trechos de código e o renomeamento de funções definidas nos fontes da plataforma.

Algoritmo 3: Compilação condicional.

```
1 #if defined(PLATFORM) &&  PLATFORM == PIC18F25K80
2 #include "platform/pic18f25k80/task_context.h"
3 #else
4 #error Platform not defined!
5 #endif
6
7 #define SaveTask(TaskRef) PlatformSaveTask((TaskRef))
```

4.5.3 Tarefas

Para realizar o gerenciamento das tarefas, foi criada uma estrutura de dados e funções de manipulação `os/task.h`. Como pode ser observado no Algoritmo 4, esta estrutura armazena informações sobre a tarefa decorrentes do arquivo de configuração OIL, bem como outras estruturas internas para controle da tarefa, como a da troca de contexto e rotina de callback.

Algoritmo 4: Estrutura de dados interna para tarefas.

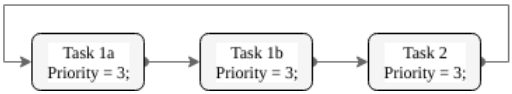
```
1 typedef struct STaskDataType {
2     TaskType id;
3     TaskPriorityType priority;
4     TaskPriorityType priority_base;
5     TaskStateType state;
6     TaskContextType context;
7     TaskCallbackType callback;
8     struct STaskDataType* next_task_same_priority;
9     ResourceDataType resources;
10 } TaskDataType;
```

A estrutura também mantém um ponteiro para uma outra tarefa de mesma prioridade, para que o algoritmo de escalonamento por tempo saiba qual a próxima tarefa a ser executada[5], conforme na Figura 37. Uma alternativa a ela seria uma pilha de ordenação para execução das tarefas ativas de mesma prioridade. Porém, dada a natureza de alocação estática para todos os recursos do sistema, o modelo pensado

[5]Importante ressaltar que o escalonamento por tempo só ocorre quando duas ou mais tarefas de mesma e maior prioridade se encontram no estado `READY`.

originalmente resultaria em um maior uso de memória, que é um recurso geralmente escasso para sistemas embarcados.

Figura 37: Lista encadeada para tarefas de mesma prioridade.



O sistema oferece rotinas de manipulação para esta e outras estruturas. A exemplo, tem-se a rotina de inicialização de tarefas, chamada de `InitializeTaskData`. Estas rotinas internas nunca devem ser utilizadas pelo usuário do sistema. Porém, elas são chamadas internamente pelo próprio SO, que pode utiliza-las, também, nos arquivos gerados pelo sistema, como em `main.c`.

As rotinas de manipulação de tarefas disponíveis para o usuário estão todas definidas em `os/osek/tasks.h`, seguindo as interfaces presentes na norma do OSEK/VDX[6].

Para atender a natureza estática do OSEK/VDX, todas as tarefas, incluindo uma tarefa para quando o sistema estiver ocioso, chamada de IDLE, são declaras por meio de um array global. O cálculo para o tamanho deste array pode ser visto na Equação 1. Basicamente, cada ativação de um mesmo tipo de tarefa é tratado como uma entidade separada.

Equação 1: Cálculo do total de tarefas do sistema.

$$TASKS\_TOTAL = (\sum_{i=1}^{n} task[i].activations) + 1$$

A partir deste array global de tarefas, são criadas referências por ponteiros, os quais servem para realizar a manipulação destas tarefas em funções do sistema.

4.5.4 Troca de Contexto

A troca de contexto das tarefas é sempre realizada em um trecho de código específico para a plataforma alvo. Segundo a norma do OSEK/VDX, uma troca de contexto pode ocorrer apenas nas seguintes ocasiões:

[6] O cabeçalho `openautos.h`, que inclui todas as declarações de rotinas da norma OSEK/VDX já é inclusa automaticamente no arquivo gerado pelo `oiler`.

- Quando a tarefa em execução não for interrompível, o SO só realizará uma troca de contexto quando esta tarefa em execução realizar a chamada a uma das seguintes funções: `ActivateTask`, `TerminateTask`, `ChainTask`, `Schedule`, `GetResource`, `ReleaseResource`;

- Quando a tarefa em execução for interrompivel e existir outra tarefa de mesma prioridade que esteja no estado `READY`, o OpenAUTOS fará a troca de contexto entre estas tarefas a cada intervalo de $1ms$. Este tempo é escolhido arbitrariamente pelo SO, já que a norma do OSEK/VDX não estipula um valor mínimo/máximo para o mesmo.

Na plataforma do PIC18F25K80, sempre que há uma troca de contexto, os dados que são salvo pelo sistema são os registradores de trabalho, seleção de banco de memória e status do microcontrolador, além da pilha de memória.

As rotinas que realizam esta troca de contexto são: `SaveTaskContext` e `LoadTaskContext`. Elas são um dos poucos trechos do código que utilizam explicitamente instruções em Assembly, devido a um bug do compilador XC8, no qual o Assembly gerado pela linguagem C não produzia um código funcional.

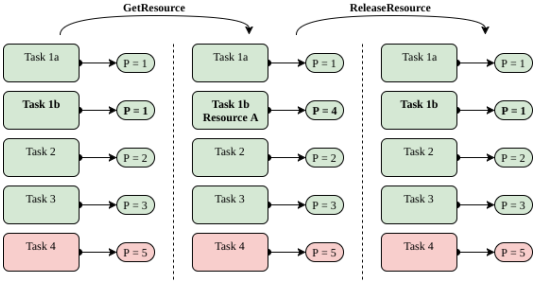
4.5.5 Alocação de Recursos e Elevação de Prioridade

Conforme especificado pelas normas do OSEK/VDX, uma tarefa pode alocar um recurso utilizando a rotina GetResource. Tarefas que conseguem alocar este recurso tem sua prioridade elevada para a prioridade do recurso, que é sempre superior a de todas as tarefas que podem alocá-lo, mas inferior a de tarefas cuja prioridade é maior que a do recurso e que não podem alocá-lo. A Figura 38 demonstra um exemplo de uma tarefa **Task1b**, a qual em um primeiro momento aloca um recurso e tem sua prioridade elevada, para logo em seguida, quando o recurso é liberado, ter sua prioridade restaurada ao seu valor original, representado na estrutura para tarefas como `priority_base`.

Ainda, segundo a norma, caso uma tarefa aloque mais de um recurso, este deve ser feito na ordem de menos prioritário para mais prioritário[7]. A liberação destes recursos deve ocorrer na ordem inversa a alocada. Este controle é alcançado pelo uso de uma pilha de recursos.

[7]Um recurso de menor prioridade não pode ser alocado após a alocação de um recurso com maior prioridade

Figura 38: Elevação de prioridade.



De acordo com a Figura 38, mesmo que uma tarefa aloque um recurso, ela ainda poderá ser interrompida por uma tarefa de maior prioridade, caso esta entre em seu estado de prontidão.

5 AVALIAÇÃO DO OPENAUTOS

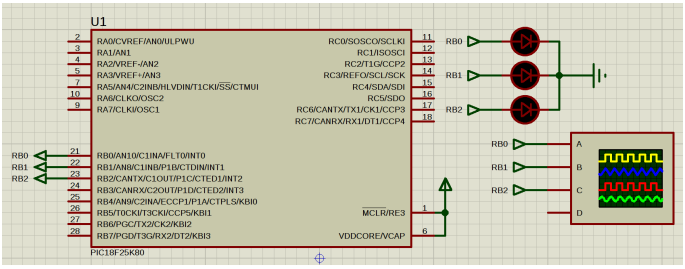
Neste capítulo são apresentados os experimentos realizados para avaliar o funcionamento do SOE OpenAUTOS.

5.1 TESTES E VALIDAÇÃO

Para realizar a validação do SO, foi idealizado um circuito sem entradas e com apenas *Light-Emitting Diode* (LED)[1]s para saída de dados, utilizados para demostrar o escalonamento das tarefas presentes no SO através de seus acionamentos.

O circuito foi projetado no software de simulação e design de *Printed Circuit Board* (PCB)[2] Proteus, versão 8.5, da *Labcenter Electronics Ltd.* Uma versão física deste mesmo circuito foi construída em uma *Protoboard*. Para a obtenção dos resultados nesta versão, utilizou-se um osciloscópio, modelo TDS 2024C, da Tektronix. Ambas versões do circuito podem ser visualizadas na Figura 39 e Figura 40.

Figura 39: Circuito no Proteus.

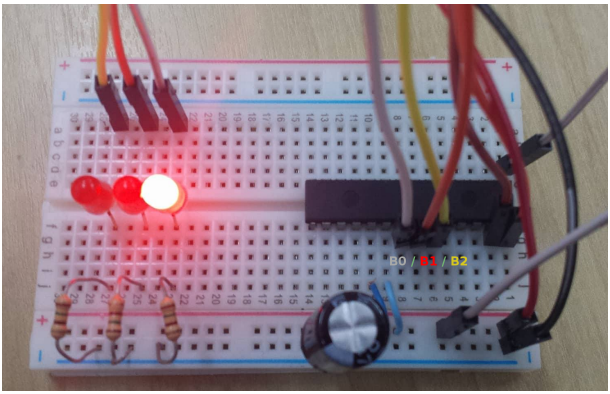


Foram realizados um total de 4 experimentos neste circuito, onde 3 deles buscaram validar os diferentes tipos de escalonamento, enquanto que o quarto testou o comportamento do sistema com a alocação de recursos e protocolo prioridade-teto.

Em todos os experimentos, são utilizadas 3 tarefas, responsáveis por fazer os acendimento dos LEDs do circuito. Nas sessões de resultados de seu respectivo experimento, as ondas mostrada pelos osciloscópios representam, respectivamente, as tarefas `task_b0`, `task_b1` e `task_b2`. A Tabela 3 apresenta uma associação por cores desta rela-

[1]Traduzido como: Diodos Emissores de Luz.
[2]Traduzido como: Placas de Circuito Impresso.

Figura 40: Circuito na Protoboard.



ção entre tarefas e ondas dos osciloscópios.

Tabela 3: Associação das tarefas e cores dos osciloscópios.

Tarefa	TDS 2024C	Proteus
`task_b0`	laranja	amarelo
`task_b1`	ciano	ciano
`task_b2`	roxo	magenta

5.1.1 Experimento 1: Escalonador por Prioridade

O objetivo deste experimento foi o de garantir que o escalonador por prioridades do OpenAUTOS estivesse operando conforme o esperado pela norma, realizando o escalonamento através do uso da chamada das rotinas: `ActivateTask`, `TerminateTask` e `ChainTask`. O código fonte deste experimento pode ser encontrado no Algoritmo 5, junto de seu arquivo de configuração OIL no Algoritmo 6, ambos listados no Apêndice A.

5.1.1.1 Configuração do Experimento

Foram utilizadas 5 tarefas, cada qual com um valor único de prioridade, especificadas na Tabela 4. As funcionalidades de cada tarefa ficaram distribuídas da seguinte maneira:

Tabela 4: Prioridade das tarefas.

Tarefa	Prioridade
`task_init`	254
`task_start`	4
`task_b2`	3
`task_b1`	2
`task_b0`	1

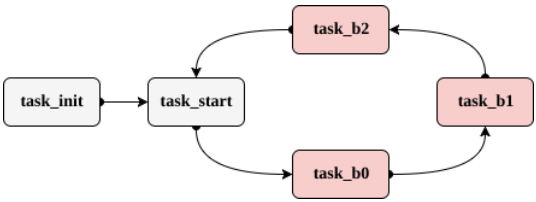
- `task_init`: única tarefa a iniciar no estado `READY`, ela faz a inicialização das portas dos sistema, bem como inicia o looping principal de escalonamento entre `task_start` e `task_bn`;

- `task_start`: apaga os LEDs e ativa as tarefas responsáveis por acende-los;

- `task_bn`: acedem o LED respectivo a porta B a qual a tarefa foi associada.

Com base nesta configuração, o fluxo de escalonamento do SO pode ser representado no diagrama de estados presente na Figura 41.

Figura 41: Diagrama de estados do escalonador por prioridade.



5.1.1.2 Resultados

Todas as rotinas testadas agiram conforme a sua especificação. Neste exemplo, a rotina `ActivateTask` não realizou a chamada a rotina de escalonamento, pois esta deveria escalonar apenas caso estivesse ativando uma tarefa de maior prioridade que a tarefa em execução no momento. O melhor exemplo para este comportamento pode ser visto na tarefa `task_start`, que altera os estados das tarefas `task_bn` de `SUSPENDED` para `READY`.

Sendo assim, o escalonamento do sistema passou a ser executado pelas rotinas `TerminateTask` e `ChainTask`, que ao realizarem a transição das tarefas do estado `RUNNING` para `SUSPENDED`, também fazem uma chamada a rotina de escalonamento. Importante ressaltar que a função `ChainTask` faz a ativação de uma tarefa antes de se suspender, executando tanto as funções de `ActivateTask` e `TerminateTask` em uma única rotina.

A execução destas tarefas podem ser visualizadas nas figuras 42 e 43.

Figura 42: Escalonador por prioridades - Osciloscópio.

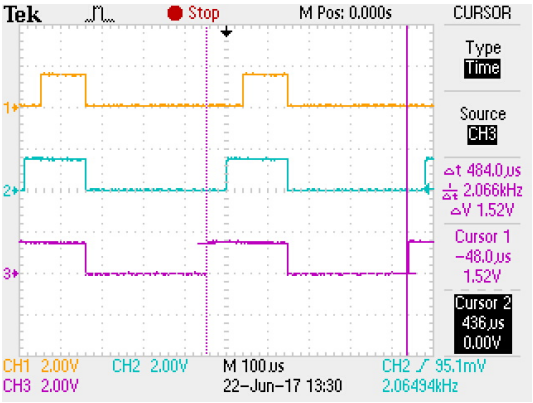
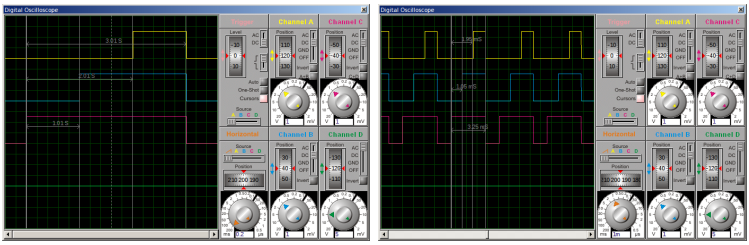


Figura 43: Escalonador por prioridades - Proteus.



(a) Atraso: μs (b) Atraso: s

Uma demonstração do circuito em funcionamento pode ser visualizada em `https://youtu.be/TBPQiTFyujo`. Para melhor observação do comportamento do sistema foram definidos atrasos de 1 segundo.

5.1.2 Experimento 2: Escalonador por Round-Robin

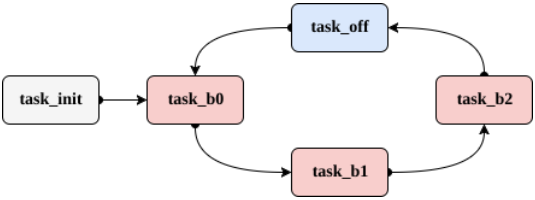
Diferente do escalonador por prioridades, o escalonamento por *Round-Robin* é realizado por uma interrupção temporal, a qual ocorre apenas quando mais de uma tarefa com a mesma prioridade estejam ativas ao mesmo tempo, podendo estas tarefas serem tanto de tipos diferentes quanto múltiplas ativações de um mesmo tipo. Para fins deste teste, todas as tarefas estão limitadas a apenas uma ativação e são de tipos diferentes, cada qual responsável por um dos LEDs do circuito. O código fonte deste experimento pode ser visualizado no Algoritmo 7, junto de seu arquivo de configuração OIL em Algoritmo 8, ambos listados no Apêndice A.

5.1.2.1 Configuração do Experimento

Assim como no escalonador por prioridades[3], foram utilizadas 5 tarefas, onde uma delas ficou responsável pela inicialização das outras, uma pelo desligamento dos LEDs e as demais pelo acendimento do LED de seu respectivo processo. Diferente do teste anterior, com exceção da tarefa `task_init`, nenhuma das outras tarefas encerra sua execução, permanecendo infinitamente ativas, até que o microcontrolador seja desligado.

Como todas as tarefas possuem a mesma prioridade, a ordem de execução se dá exclusivamente pela ordem de ativação das tarefas, conforme especificado pelas linhas 7 à 10 do Algoritmo 7 do Apêndice A. O diagrama de estados presente na Figura 44 ilustra a ordem de execução destas tarefas.

Figura 44: Diagrama de estados do escalonador por Round-Robin.



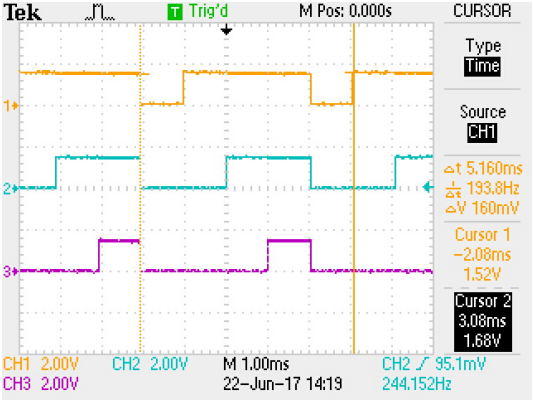
[3]vide tópico 5.1.1.

5.1.2.2 Resultados

Assim como no teste anterior, o comportamento do sistema foi exatamente como o especificado pela norma. A cada intervalo de 1_{ms}, valor adotado pelo OpenAUTOS como o intervalo de escalonamento para Round-Robin, ou seja, o valor do quantum, uma interrupção é gerada pelo temporizador do sistema. Durante esta interrupção, faz-se uma busca pela próxima tarefa de mesma prioridade que se encontra ativa[4], fazendo com que haja uma troca de contexto em caso afirmativo. Importante ressaltar que esta é a única forma encontrada pela norma de troca de contexto em uma interrupção. Nos demais casos, mesmo que uma tarefa seja ativada durante uma interrupção, ela só será considerada no próximo ponto de escalonamento[5].

O resultado da execução deste teste, tanto no modelo físico quanto no virtual, pode ser visualizado na Figura 45 e na Figura 46.

Figura 45: Escalonador por *Round-Robin* - Osciloscópio.

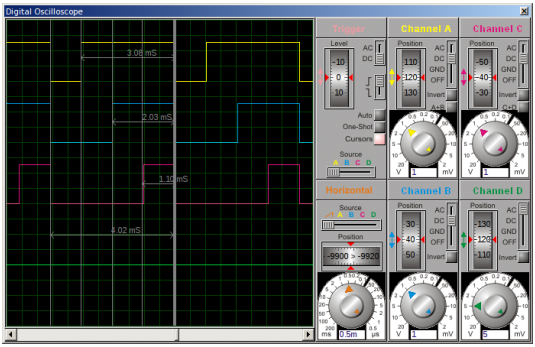


Uma demonstração do circuito em funcionamento está disponível em `https://youtu.be/QC50OuNeoio`. Para uma melhor visualização do experimento foi aumentado para 1_s o tempo de interrupção para chamada do escalonador *Round-Robin*.

[4] No estado `READY`.

[5] Que se resumem as rotinas: `Schedule`, `ActivateTask`, `TerminateTask`, `ChainTask`, `SetEvent`, `WaitEvent` e `ReleaseResource`.

Figura 46: Escalonador por *Round-Robin* - Proteus.



5.1.3 Experimento 3: Escalonador por Prioridade e Round-Robin

Este experimento teve como objetivo avaliar o funcionamento do SO em um cenário onde ambos os modelos de escalonamento seriam requisitados. O código para este experimento está disponibilizado no Algoritmo 9 bem como no arquivo de configuração OIL em Algoritmo 10, ambos listados no Apêndice A.

5.1.3.1 Configuração do Experimento

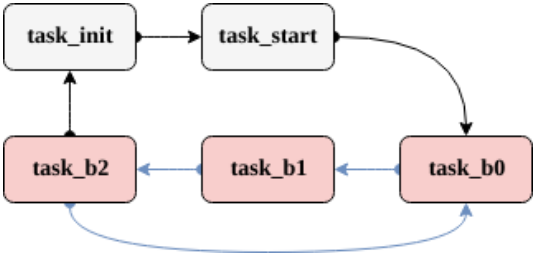
Foram utilizados 5 tipos de tarefas, os quais ficaram organizados da seguinte maneira:

- `task_init`: tarefa com a menor prioridade, e a única que nunca é encerrada. Ela tem por objetivo manter o programa em ciclo após o encerramento de todas as tarefas de maior prioridade que ela;
- `task_start`: ativada pela tarefa `task_init`, seu papel é o de apagar os LEDS, bem como o de ativar as tarefas que os reacenderão;
- `task_bn`: são as tarefas responsáveis pelo acendimento de seu respectivo LED. Todas possuem a mesma prioridade e também fazem uso a uma chamada na rotina de atraso, para permitir que o tempo de escalonamento do *Round-Robin* execute pelo menos uma vez para cada uma destas tarefas.

A Figura 47 apresenta um diagrama de estados deste teste, com os seguintes destaques:

- Em cinza encontram-se as tarefas que são escalonados pelas rotinas de prioridade;
- Em rosa encontram-se as tarefas que possuem mesma prioridade e que são escalonadas por *Round-Robin*;
- Em preto encontram-se as setas indicando transições que ocorrem exclusivamente por `TerminateTask`;
- Em azul encontram-se as setas indicando transições que ocorrem tanto por interrupção quanto por `TerminateTask`;

Figura 47: Diagrama de estados do escalonador por Prioridade e Round-Robin.



5.1.3.2 Resultados

O experimento se comportou conforme o esperado onde, em um primeiro momento, o sistema faz a ativação das tarefas através de chamadas a rotina `ActivateTask`, até então escalonando apenas pela tarefa de maior prioridade ativa na ocasião, que pela lógica do programa acontece na seguinte ordem: `task_init`, `task_start` e `task_b0`.

No momento em que `task_b0` entra em execução, apenas tarefas de prioridade idêntica estão ativas no sistema. A partir deste momento, o sistema passa a escalonar as tarefas por *Round-Robin* nos intervalos de quantum definidos pelo SO. O sistema permanece, então, neste modelo de escalonamento até que as tarefas terminem seu tempo de atraso, encerrando, prematuro ao quantum, seu funcionamento através da rotina `TerminateTask`.

Uma vez encerrada a última das tarefas de ativação de LEDs, a tarefa `task_init` volta a ser a tarefa de maior prioridade no sistema, reiniciando o processo.

As ondas dos osciloscópios das figuras 48 e 49 ilustram o momento em que as tarefas de acionamento dos LEDs são iniciadas e o tempo no qual elas permanecem processando a rotina de atraso.

Figura 48: Escalonador por prioridade e *Round-Robin* - Osciloscópio.

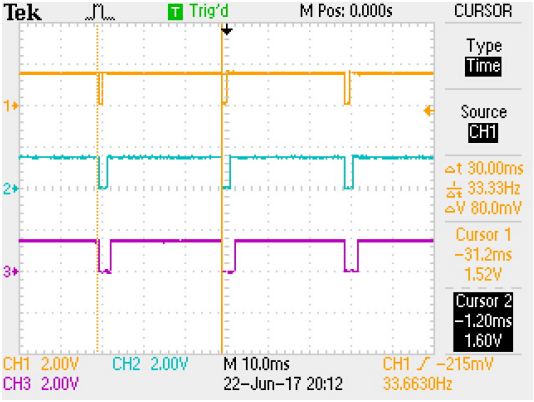
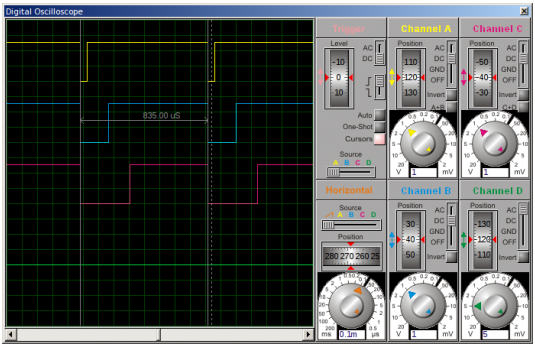


Figura 49: Escalonador por prioridade e *Round-Robin* - Proteus.



Um vídeo demonstrativo do funcionamento deste circuito encontra-se disponível em `https://youtu.be/LSDL_LMfO3o`, também utilizando atrasos de 1_s para permitir uma melhor observação do fluxo de funcionamento.

5.1.4 Experimento 4: Alocação de Recursos

Assim como no modelo de prioridade com *Round-Robin*, este experimento utilizou um cenário bem semelhante, porém, com o uso da alocação de recursos e prioridade-teto para que um processo de baixa prioridade pudesse ativar tarefas de maior prioridade sem ser escalonado imediatamente. O código fonte deste experimento pode ser encontrado em Algoritmo 11, junto de seu arquivo de configuração OIL em Algoritmo 12, ambos listados no Apêndice A.

5.1.4.1 Configuração do Experimento

Diferente dos outros experimentos, foram utilizadas apenas 4 tarefas, onde:

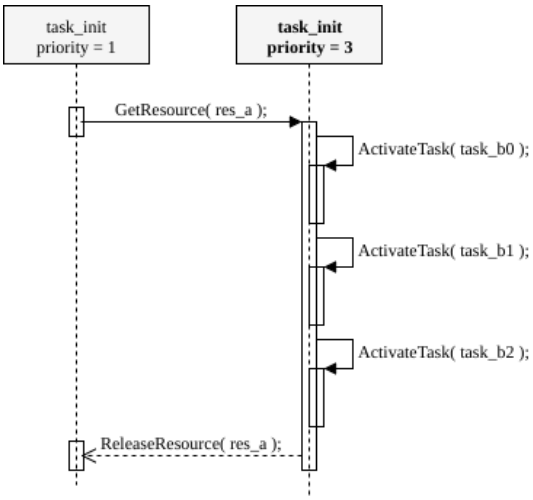
- `task_init`: além de fazer a inicialização das portas utilizadas, também é responsável por apagar os LEDs e ativar as tarefas que os reacenderão. Por possuir uma prioridade inferior a das tarefas dos LEDs, `task_init` primeiro aloca um recurso de maior prioridade que as tarefas antes de ativá-las. Por fim, esta tarefa é escalonada assim que tem sua prioridade restaurada ao normal, logo após a chamada a rotina *ReleaseResource*. este processo de alocação do recurso pode ser visualizado no diagrama de sequência na Figura 50;
- `task_bn`: tarefas que realizam o acendimento dos LEDS.

5.1.4.2 Resultados

Após a alocação do recurso, a tarefa `task_init` passou a ter uma prioridade superior a das tarefas que estaria ativando, permitindo que ativasse todas as tarefas `task_bn` sem que ela própria fosse escalonada.

A partir do momento no qual o recurso alocado por `task_init` é liberado, esta tem sua prioridade reduzida para seu valor original, o qual é inferior ao das demais tarefas que se encontram aptos para execução. Sendo assim, o SO passa a executar as tarefas de maior prioridade, que por possuírem valores idênticos de prioridade, são escalonados por *Round-Robin* até que sua execução encerre com a chamada a rotina *TerminateTask*, reiniciando o clico em `task_init`.

Figura 50: Diagrama de sequencia da alocação de recurso e prioridade-teto.



Caso `task_init` tentasse ativar as demais tarefas sem antes ter feito a alocação do recurso, o SO teria interrompido seu processamento ao retornar da rotina *ActivateTask*, executando cada tarefa até seu encerramento em *TerminateTask*, sem nunca escalonalas por *Round-Robin*.

Os resultados da execução deste experimento podem ser vistos nos gráficos da Figura 51 e Figura 52.

Figura 51: Alocação de recurso e prioridade-teto - Osciloscópio.

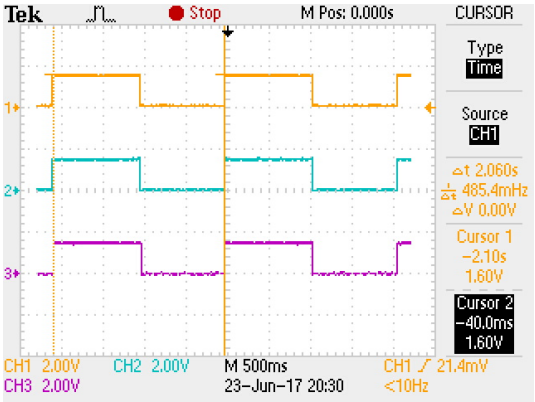
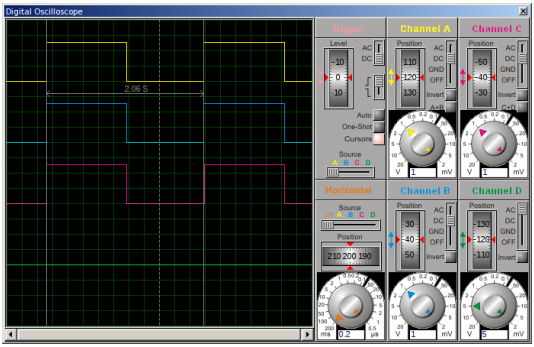


Figura 52: Alocação de recurso e prioridade-teto - Proteus.



Um vídeo demonstrativo do funcionamento deste circuito encontra-se disponível em `https://youtu.be/XbD-STWvOCU`, também utilizando atrasos de 1_s para permitir uma melhor observação do fluxo de funcionamento.

5.2 DIFICULDADES

Diversas foram as dificuldades encontradas para o desenvolvimento deste projeto. Na sequência serão descritas algumas delas.

5.2.1 Geração de Instruções Defeituosas pelo Compilador XC8

Por razões desconhecidas, a instrução `movff`, em Assembly, muitas vezes não realizava nenhuma ação, principalmente quando utilizada com alguns registradores especiais, como os `TOSU`, `TOSH` e `TOSL`, utilizados para manipulação da pilha dos microcontroladores da linha PIC18F, bem como os registradores `TMR0` e `TMR1`, utilizados para calcular o tempo de estouro de seus respectivos temporizadores[6].

Esta instrução é automaticamente utilizada pelo compilador XC8 e, na maioria dos casos, funciona corretamente. Infelizmente, para estes casos onde ela não funcionou, foi necessário que uma parte de código Assembly fosse escrito manualmente para contornar o seu uso.

[6]Escrever um valor literal nestes registradores sempre resulta em um código operante, pois o compilador passa a utilizar o comando `movlf` ao invés de `movff`.

5.2.2 Seleção de Bancos de Memória

Devido a grande quantidade de memória alocada estaticamente pelo OpenAUTOS e tarefas definidas pelo usuário, um dos inconvenientes que passou a ser notado foram as indicações de bancos de memória cheios.

Idealmente, ele deveria ser capaz de declarar estas variáveis que excederiam o limite de um banco de memória em um outro banco automaticamente. Porém, o compilador não estava realizando esta operação, que teve que ser realizada manualmente a partir de instruções Assembly.

Um outro inconveniente que passou a ser observado foi a má distribuição dos espaços presentes nos bancos de memória do microcontrolador, o que pode ser notado no arquivo de extensão *lst* gerado durante a compilação da imagem do hardware, onde há diversas segmentações de memória e a maioria dos bancos apresentam um espaço considerável ainda.

Por este projeto utilizar apenas a versão gratuita do compilador, e também por ter suas compilações apenas em modo de *Debug*, acredita-se que diversas otimizações ainda possam ser alcançadas neste quesito.

5.2.3 Ferramentas de Depuração

Embora de grande ajuda na depuração de *bugs*, algumas vezes as ferramentas MPLAB e Proteus não conseguem acompanhar perfeitamente os formatos de imagem para depuração suportados pelo compilador XC8, que são o *elf* e *cof*, o que acaba forçando o desenvolvedor a buscar meios alternativos de buscar informações em uma determinada área de código, geralmente resultando no uso de LEDs para indicar estados do sistema.

5.2.4 Abrangência da Norma

Mesmo com a alta disponibilidade de informações sobre projetos e artigos que atuam nas normas do OSEK/VDX e AUTOSAR, pouquíssimos são aqueles se aprofundam nos temas. Isto implica que a maior parte das informações devem ser extraídas diretamente das normas, o que passa a ser um processo trabalhoso devido a extensão e complexidade das mesmas.

É, também, difícil de acompanhar e levantar todos os detalhes referentes a cada módulo proposto, já que as normas costumam separar informações entre elas que complementam estes sistemas.

6 CONSIDERAÇÕES FINAIS

O desenvolvimento de um sistema operacional embarcado é uma tarefa extensa, desafiadora e complexa, tendo em vista a quantidade funcionalidades que este deve oferecer, bem como a estabilidade, desempenho, portabilidade, dentre vários outros aspectos possíveis.

Desenvolver um SOE que visa atender a um conjunto de normas como as do OSEK/VDX e AUTOSAR apresentou-se um desafio ainda maior do que o que era originalmente esperado, tamanha a complexidade do sistema proposto por elas.

Embora o trabalho tenha apresentado sucesso no desenvolvimento parcial de um SOE baseado nestas normas, existem ainda diversos módulos e funcionalidades que precisam serem implementadas para que o SOE se adeque totalmente as normas.

Este trabalho tem sua importância marcada como o inicio do SOE OpenAUTOS que, no momento, oferece uma estrutura preparada para o desenvolvimento em múltiplas plataformas de micro-controladores, um compilador para a linguagem OIL, sistemas para o gerenciamento de tarefas, trocas de contexto e alocação de recursos, bem como uma base de código para a implementação das demais funcionalidades.

Embora todas estas funcionalidades tenham sido testadas e validadas conforme especifica a norma, há ainda bastante espaço para melhorias, pois muitas das funções e estruturas de dados utilizadas foram implementadas sem revisões de otimização ou um planejamento de longo prazo, com o intuito de deixar o OpenAUTOS em um ponto usável o mais cedo possível, para que a partir dali ele evoluísse para um sistema mais complexo.

Espera-se que o OpenAUTOS continue a evoluir, mesmo com a conclusão deste trabalho, e que, eventualmente, ele venha a se tornar uma referência no mercado de sistemas embarcados automobilísticos, tanto como uma ferramenta de ensino como em usabilidade. Para isso, seu código está disponível no repositório online GitHub, sobre o endereço `https://github.com/brunocanella/OpenAUTOS`.

6.1 PROPOSTAS PARA TRABALHOS FUTUROS

Neste seção são listadas algumas propostas como trabalhos futuros que visam estender e melhorar o SOE OpenAUTOS:

1. Implementar os módulos restantes para a conclusão do SO, con-

forme a norma do OSEK/VDX;

2. Implementar as funcionalidades de SO acrescentada pelas normas do AUTOSAR;

3. Realizar *benchmarks* comparativos com outras soluções em SOE disponiveis;

4. Fazer o porte do SOE OpenAUTOS para outros microcontroladores;

5. Otimizar o espaço de memória utilizado pelo SO, principalmente quanto ao aproveitamento dos espaços nos bancos de memória dos microcontroladores PIC18F;

6. Iniciar o desenvolvimento dos modulos complementares ao SOE, como a Run-Time Environment (RTE) do AUTOSAR, ou ainda as normas de comunicação do OSEK-COM;

7. Desenvolver um projeto veícular que utilize como SOE o OpenAUTOS.

REFERÊNCIAS

AUTOSAR. Autosar. 2016. Disponível em: <https://www.autosar.org/>. Acesso em: 06/12/2016.

BOSCH, R. **Bosch Automotive A product history**. 2010. 84 p. Disponível em: <history.bosch.com>.

BOSCH, R. **Manual de Tecnologia Automotiva**. 25. ed. [S.l.: s.n.], 2014. 1232 p.

BOSCH, R. Peças auto bosch: Injeção common rail. p. 1, 2015. Disponível em: <http://pt.bosch-automotive.com/pt/internet/parts/parts_and_accessories/motor_and_sytems/ Acesso em: 06/12/2016.

BYERS, T. 2016 toyota tacoma in delaware oh. p. 2, 2016. Disponível em: <http://www.byerstoyota.com/2016-toyota-tacoma-in-delaware-oh.html>. Acesso em: 06/12/2016.

BéCHENNEC, J.-L.; FAUCOU, S. Trampoline: an open platform for (small) embedded systems based on osek/vdx and autosar. p. 29, 2009. Disponível em: <http://2009.rmll.info/IMG/pdf/trampoline-rmll2009.pdf>. Acesso em: 06/12/2016.

BéCHENNEC, J.-L.; FAUCOU, S. Trampoline: Opensource rtos project. 2016. Disponível em: <http://trampoline.rts-software.org/>. Acesso em: 07/12/2016.

CERTIFICATION, I. E. What is v-model- advantages, disadvantages and when to use it? p. 4, 2012. Disponível em: <http://istqbexamcertification.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/>. Acesso em: 06/12/2016.

CHASE, C. Evolution of electronic safety systems. p. 12, 2015. Disponível em: <http://www.autotrader.ca/newsfeatures/20151116/evolution-of-electronic-safety-systems/#SFtxlwwAweZfhOJg.97>. Acesso em: 06/12/2016.

CLARKE, P. Android, freertos top ee times' 2013 embedded survey. p. 4, 2013. Disponível em: <http://www.eetimes.com/document.asp?doc_id=1263083>. Acesso em: 06/12/2016.

DALEY, R. C.; NEUMANN, P. G. A general-purpose file system for secondary storage. 2016. Disponível em: <http://www.multicians.org/fjcc4.html>.

EVANS, D. A internet and das coisas: Como a próxima evolução da internet está mudando tudo. p. 13, 2011. Disponível em: <http://www.cisco.com/c/dam/global/pt_br/assets/executives/pdf/internet_of_thin Acesso em: 30/11/2016.

FREERTOS. Freertos. 2016. Disponível em: <http://www.freertos.org/>. Acesso em: 06/12/2016.

GLASS, L. A. M. A. Regulador de la ventana. p. 1, 2016. Disponível em: <http://www.losamigosmobileautoglass.com/regulador-de-la-ventana>. Acesso em: 06/12/2016.

GROUP, B. L. U. **The Linux Information Project**. 10 2016. Disponível em: <http://www.linfo.org/index.html>.

GUIMARãES, A. de A. **Eletrônica Embarcada Automotiva**. [S.l.]: Saraiva, 2007. 326 p.

HIPP, D. R. Sqlite: Lemon parser. 2000. Disponível em: <https://www.sqlite.org/src/doc/trunk/doc/lemon.html>. Acesso em: 13/06/2017.

LEE, E. A. Osek standard. p. 70, 2012. Disponível em: <https://chess.eecs.berkeley.edu/design/2012/lectures/EE249_14_OSEKstandard.pdf Acesso em: 06/12/2016.

LI, Q. **Real-Time Concepts for Embedded Systems**. [S.l.]: CMP Books, 2003. 294 p.

LI, Y. Sdvos. 2015. Disponível em: <http://www.sdvos.org>. Acesso em: 13/06/2017.

MACHER, G. et al. Automotive real-time operating systems: A model-based configuration approach. **EWiLi'14**, p. 6, 2014. Disponível em: <http://ceur-ws.org/Vol-1291/ewili14_19.pdf>.

MATTAR, G. 1958 desoto electrojector - world's first electronic fuel injection. p. 12, 2014. Disponível em: <http://www.allpar.com/cars/desoto/electrojector.html>. Acesso em: 26/11/2016.

MUSEUM, C. H. Real-time all the time. p. 2, 2016. Disponível em: <http://www.computerhistory.org/revolution/real-time-computing/6/134>. Acesso em: 06/12/2016.

OLIVEIRA, J. A velha escola 288: Dia do fusca. 2015. Disponível em: <http://laviejaescuela288.blogspot.com.br/2015/06/dia-do-fusca-wolkswagen-tipo-1-beetle.html>. Acesso em: 06/12/2016.

ROCKET, W. R. **Rocket Kernel Primer**. [S.l.], 2015. 110 p.

SCIENCES, I. for C. . I. Osek: Osek standard. 2016. Disponível em: <http://www.cs.ru.nl/lab/nxt/nxtosek/OSEK/OSEK.html>. Acesso em: 06/12/2016.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 9th. ed. [S.l.]: John Wiley & Sons. Inc, 2012. 976 p.

SIMONNOT-LION, F.; NAVET, N. **Automotive Computer Controlled Systems**. 1. ed. [S.l.]: CRC Press, 2008. 488 p. (Industrial Information Technology).

SOFTELEC. Free eletronic and software design: Picos18. 2002. Disponível em: <http://softelec.pagesperso-orange.fr/index_us.htm>. Acesso em: 06/12/2016.

STMICROELECTRONICS. **LIN (LOCAL and INTERCONNECT NETWORK) and SOLUTIONS**: An1278 application note. [S.l.], 2002. 44 p. Disponível em: <www.st.com/resource/en/application_note/cd00004273.pdf>. Acesso em: 30/11/2016.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. [S.l.]: Pearson, 2014. 1136 p.

TANENBAUM, A. S.; WOODHULL, A. S. **Operating Systems**: Design and implementation. 3rd. ed. [S.l.]: Prentice Hall, 2006. 1056 p.

TRANSPORTATION, U. D. of. Clarus: Concept of operations. **Federal Highway Administration**, p. 130, 2005. Disponível em: <http://ntl.bts.gov/lib/jpodocs/repts_te/14158_files/14158.pdf>. Acesso em: 06/12/2016.

VIZCAYNO, D. The future of technology, privacy, security and risks (part 3 of 5): Modern car technology. **Daniel Vizcayno's Insights**, p. 3, 2015. Disponível em: <https://dcvizcayno.wordpress.com/2015/08/28/the-future-of-technology-privacy-security-and-risks-part-3-of-5/>. Acesso em: 06/12/2016.

VOLVO. Autopilot: Travel calmer, safer, cleaner. p. 2, 2016. Disponível em: <http://www.volvocars.com/au/about/innovations/intellisafe/autopilot>. Acesso em: 06/12/2016.

APÊNDICE A – Algoritmos dos Testes de Validação do OpenAUTOS

Algoritmo 5: Algoritmo do escalonador por prioridade.

```
TASK( task_init ) {
    TRISB = 0x00; // Output Lower Bits
    LATB = 0x00;

    ChainTask( task_start );
}
#define _XTAL_FREQ 64000000
TASK( task_rb0 ) {
    LATBbits.LATB0 = 1;
    __delay_ms(1);
    ChainTask(task_start);
}
TASK( task_rb1 ) {
    LATBbits.LATB1 = 1;
    __delay_ms(1);
    TerminateTask();
}
TASK( task_rb2 ) {
    LATBbits.LATB2 = 1;
    __delay_ms(1);
    TerminateTask();
}
TASK( task_start ) {
    LATB = 0;
    __delay_ms(1);
    ActivateTask( task_rb0 );
    ActivateTask( task_rb1 );
    ActivateTask( task_rb2 );
    TerminateTask();
}
```

Algoritmo 6: Configuração OIL do escalonador por prioridade.

```
OIL_VERSION = "2.5" "DESC_OIL_VERSION";

CPU PIC18F25K80 {
    OS OpenAUTOS {
        STATUS = EXTENDED;
        STARTUPHOOK = TRUE;
        ERRORHOOK = TRUE;
        SHUTDOWNHOOK = FALSE;
        PRETASKHOOK = FALSE;
        POSTTASKHOOK = FALSE;
        USEGETSERVICEID = TRUE;
        USEPARAMETERACCESS = TRUE;
        USERESSCHEDULER = TRUE;
    } "DESC_OS";
```

```
15
16      TASK task_rb0 {
17          PRIORITY = 1;
18          SCHEDULE = FULL;
19          ACTIVATION = 1;
20          AUTOSTART = FALSE;
21      };
22      TASK task_rb1 {
23          PRIORITY = 2;
24          SCHEDULE = FULL;
25          ACTIVATION = 1;
26          AUTOSTART = FALSE;
27      };
28      TASK task_rb2 {
29          PRIORITY = 3;
30          SCHEDULE = FULL;
31          ACTIVATION = 1;
32          AUTOSTART = FALSE;
33      };
34      TASK task_start {
35          PRIORITY = 5;
36          SCHEDULE = FULL;
37          ACTIVATION = 1;
38          AUTOSTART = FALSE;
39      };
40      TASK task_init {
41          PRIORITY = 254;
42          SCHEDULE = FULL;
43          ACTIVATION = 1;
44          AUTOSTART = TRUE {
45              APPMODE = AppMode0;
46          };
47      };
48  };
```

Algoritmo 7: Algoritmo do escalonador por *Round-Robin*.

```
1  TASK( task_init ) {
2      TRISB = 0x00; // Output Lower Bits
3      LATB = 0x00;
4
5      ActivateTask( task_rb0 );
6      ActivateTask( task_rb1 );
7      ActivateTask( task_rb2 );
8      ActivateTask( task_off );
9      TerminateTask();
10 }
11
12 TASK( task_rb0 ) {
```

```
13      while( TRUE ) {
14          LATBbits.LATB0 = 1;
15      }
16  }
17
18  TASK( task_rb1 ) {
19      while( TRUE ) {
20          LATBbits.LATB1 = 1;
21      }
22  }
23
24  TASK( task_rb2 ) {
25      while( TRUE ) {
26          LATBbits.LATB2 = 1;
27      }
28  }
29
30  TASK( task_off ) {
31      while( TRUE ) {
32          LATB = 0;
33      }
34  }
```

Algoritmo 8: Configuração OIL do escalonador por *Round-Robin*.

```
1   OIL_VERSION = "2.5" "DESC_OIL_VERSION";
2
3   CPU PIC18F25K80 {
4       OS OpenAUTOS {
5           STATUS = EXTENDED;
6           STARTUPHOOK = TRUE;
7           ERRORHOOK = TRUE;
8           SHUTDOWNHOOK = FALSE;
9           PRETASKHOOK = FALSE;
10          POSTTASKHOOK = FALSE;
11          USEGETSERVICEID = TRUE;
12          USEPARAMETERACCESS = TRUE;
13          USERESSCHEDULER = TRUE;
14      } "DESC_OS";
15
16      TASK task_rb0 {
17          PRIORITY = 1;
18          SCHEDULE = FULL;
19          ACTIVATION = 1;
20          AUTOSTART = FALSE;
21      };
22      TASK task_rb1 {
23          PRIORITY = 1;
24          SCHEDULE = FULL;
```

```
25          ACTIVATION = 1;
26          AUTOSTART = FALSE;
27      };
28      TASK task_rb2 {
29          PRIORITY = 1;
30          SCHEDULE = FULL;
31          ACTIVATION = 1;
32          AUTOSTART = FALSE;
33      };
34      TASK task_off {
35          PRIORITY = 1;
36          SCHEDULE = FULL;
37          ACTIVATION = 1;
38          AUTOSTART = FALSE;
39      };
40      TASK task_init {
41          PRIORITY = 254;
42          SCHEDULE = FULL;
43          ACTIVATION = 1;
44          AUTOSTART = TRUE {
45              APPMODE = AppMode0;
46          };
47      };
48  };
```

Algoritmo 9: Algoritmo do escalonador por prioridade e *Round-Robin*.

```
1   #include <stdint.h>
2
3   #define _XTAL_FREQ 64000000
4
5   void delay_s( uint8_t s ) {
6       while( s > 0 ) {
7           s--;
8           for( uint8_t i = 0; i < 100; i++ ) {
9               __delay_ms(10);
10          }
11      }
12  }
13
14  TASK( task_init ) {
15      TRISB = 0x00; // Output Lower Bits
16      LATB = 0x00;
17
18      while( TRUE ) {
19          ActivateTask( task_start );
20      }
21  }
```

```
22
23 TASK( task_start ) {
24     ActivateTask( task_rb0 );
25     ActivateTask( task_rb1 );
26     ActivateTask( task_rb2 );
27     LATB = 0;
28     TerminateTask();
29 }
30
31 TASK( task_rb0 ) {
32     LATBbits.LATB0 = 1;
33     delay_s(2);
34     TerminateTask();
35 }
36
37 TASK( task_rb1 ) {
38     LATBbits.LATB1 = 1;
39     delay_s(2);
40     TerminateTask();
41 }
42
43 TASK( task_rb2 ) {
44     LATBbits.LATB2 = 1;
45     delay_s(2);
46     TerminateTask();
47 }
```

Algoritmo 10: Configuração OIL do escalonador por prioridade e *Round-Robin*.

```
1  OIL_VERSION = "2.5" "DESC_OIL_VERSION";
2
3  CPU PIC18F25K80 {
4      OS OpenAUTOS {
5          STATUS = EXTENDED;
6          STARTUPHOOK = TRUE;
7          ERRORHOOK = TRUE;
8          SHUTDOWNHOOK = FALSE;
9          PRETASKHOOK = FALSE;
10         POSTTASKHOOK = FALSE;
11         USEGETSERVICEID = TRUE;
12         USEPARAMETERACCESS = TRUE;
13         USERESSCHEDULER = TRUE;
14     } "DESC_OS";
15
16     TASK task_rb0 {
17         PRIORITY = 2;
18         SCHEDULE = FULL;
19         ACTIVATION = 1;
```

```
        AUTOSTART = FALSE;
    };
    TASK task_rb1 {
        PRIORITY = 2;
        SCHEDULE = FULL;
        ACTIVATION = 1;
        AUTOSTART = FALSE;
    };
    TASK task_rb2 {
        PRIORITY = 2;
        SCHEDULE = FULL;
        ACTIVATION = 1;
        AUTOSTART = FALSE;
    };
    TASK task_start {
        PRIORITY = 3;
        SCHEDULE = FULL;
        ACTIVATION = 1;
        AUTOSTART = FALSE;
    };
    TASK task_init {
        PRIORITY = 1;
        SCHEDULE = FULL;
        ACTIVATION = 1;
        AUTOSTART = TRUE {
            APPMODE = AppMode0;
        };
    };
};
```

Algoritmo 11: Algoritmo da alocação de recurso e prioridade-teto.

```
#include <stdint.h>

#define _XTAL_FREQ 64000000

void delay_s( uint8_t s ) {
    while( s > 0 ) {
        s--;
        for( uint8_t i = 0; i < 100; i++ ) {
            __delay_ms(10);
        }
    }
}

TASK( task_init ) {
    TRISB = 0x00; // Output Lower Bits
    LATB = 0x00;

```

```
18      while( TRUE ) {
19          GetResource( res_a );
20          ActivateTask( task_rb0 );
21          ActivateTask( task_rb1 );
22          ActivateTask( task_rb2 );
23          LATB = 0;
24          delay_s(1);
25          ReleaseResource( res_a );
26      }
27  }
28
29  TASK( task_rb0 ) {
30      LATBbits.LATB0 = 1;
31      delay_s(1);
32      TerminateTask();
33  }
34
35  TASK( task_rb1 ) {
36      LATBbits.LATB1 = 1;
37      delay_s(1);
38      TerminateTask();
39  }
40
41  TASK( task_rb2 ) {
42      LATBbits.LATB2 = 1;
43      delay_s(1);
44      TerminateTask();
45  }
```

Algoritmo 12: Configuração OIL da alocação de recurso e prioridade-teto.

```
1   OIL_VERSION = "2.5" "DESC_OIL_VERSION";
2
3   CPU PIC18F25K80 {
4       OS OpenAUTOS {
5           STATUS = EXTENDED;
6           STARTUPHOOK = TRUE;
7           ERRORHOOK = TRUE;
8           SHUTDOWNHOOK = FALSE;
9           PRETASKHOOK = FALSE;
10          POSTTASKHOOK = FALSE;
11          USEGETSERVICEID = TRUE;
12          USEPARAMETERACCESS = TRUE;
13          USERESSCHEDULER = TRUE;
14      } "DESC_OS";
15
16      TASK task_rb0 {
17          PRIORITY = 2;
```

```
18          SCHEDULE = FULL;
19          ACTIVATION = 1;
20          AUTOSTART = FALSE;
21      };
22      TASK task_rb1 {
23          PRIORITY = 2;
24          SCHEDULE = FULL;
25          ACTIVATION = 1;
26          AUTOSTART = FALSE;
27      };
28      TASK task_rb2 {
29          PRIORITY = 2;
30          SCHEDULE = FULL;
31          ACTIVATION = 1;
32          AUTOSTART = FALSE;
33      };
34
35      TASK task_init {
36          PRIORITY = 1;
37          SCHEDULE = FULL;
38          ACTIVATION = 1;
39          AUTOSTART = TRUE {
40              APPMODE = AppMode0;
41          };
42          RESOURCE = res_a;
43      };
44      RESOURCE res_a {
45          RESOURCEPROPERTY = STANDARD;
46      };
47  };
```

ANEXO A – OpenAUTOS API: OSEK/VDX

Algoritmo 13: Interfaces do OSEK/VDX - Rotinas de Hooks.

```
1  /**
2  * @brief This hook routine is called by the operating system at the
   ↪   end of a system service which returns StatusType not equal
   ↪   E_OK. It is called before returning to the task level.
3  * @brief This hook routine is called when an alarm expires and an
   ↪   error is detected during task activation or event setting.
4  * @brief The ErrorHook is not called, if a system service called
   ↪   from ErrorHook does not return E_OK as status value. Any error
   ↪   by calling of system services from the ErrorHook can only be
   ↪   detected by evaluating the status value.
5  *
6  * @remark See chapter 11.1 for general description of hook
   ↪   routines.
7  *
8  * @param[in] Error The Error occurred.
9  */
10 void ErrorHook( StatusType Error );
11
12 /**
13 * This hook routine is called by the operating system before
   ↪   executing a new task, but after the transition of the task to
   ↪   the running state (to allow evaluation of the TaskID by
   ↪   GetTaskID).
14 *
15 * @remark See chapter 11.1 for general description of hook
   ↪   routines.
16 */
17 void PreTaskHook( void );
18
19 /**
20 * This hook routine is called by the operating system after
   ↪   executing the current task, but before leaving the task's
   ↪   running state (to allow evaluation of the TaskID by
   ↪   GetTaskID).
21 *
22 * @remark See chapter 11.1 for general description of hook
   ↪   routines.
23 */
24 void PostTaskHook( void );
25
26 /**
27 * This hook routine is called by the operating system at the end of
   ↪   the operating system initialisation and before the scheduler
   ↪   is running. At this time the application can initialise device
   ↪   drivers etc.
28 *
29 * @remark See chapter 11.1 for general description of hook
   ↪   routines.
```

```
30 */
31 void StartupHook( void );
32
33 /**
34 * This hook routine is called by the operating system when the OS
   ↪  service ShutdownOS has been called. This routine is called
   ↪  during the operating system shut down.
35 *
36 * @remark ShutdownHook is a hook routine for user defined shutdown
   ↪  functionality, see chapter 11.4.
37 *
38 * @param[in] Error Error occurred
39 */
40 void ShutdownHook( StatusType Error );
```

Algoritmo 14: Interfaces do OSEK/VDX - Rotinas de Interrupção.

```
1 /**
2 * This service restores the state saved by DisableAllInterrupts.
3 * @remark The service may be called from an ISR category 1 and
   ↪  category 2 and from the task level, but not from hook
   ↪  routines.
4 * @remark This service is a counterpart of DisableAllInterrupts
   ↪  service, which has to be called before, and its aim is the
   ↪  completion of the critical section of code. No API service
   ↪  calls are allowed within this critical section.
5 * @remark The implementation should adapt this service to the
   ↪  target hardware providing a minimum overhead. Usually, this
   ↪  service enables recognition of interrupts by the central
   ↪  processing unit.
6 */
7 void EnableAllInterrupts( void );
8
9 /**
10 * This service disables all interrupts for which the hardware
   ↪  supports disabling. The state before is saved for the
   ↪  EnableAllInterrupts call.
11 *
12 * @remark The service may be called from an ISR category 1 and
   ↪  category 2 and from the task level, but not from hook
   ↪  routines.
13 * @remark This service is intended to start a critical section of
   ↪  the code.
14 * @remark This section shall be finished by calling the
   ↪  EnableAllInterrupts service. No API service calls are allowed
   ↪  within this critical section.
```

```
15 * @remark The implementation should adapt this service to the
   ↪   target hardware providing a minimum overhead. Usually, this
   ↪   service disables recognition of interrupts by the central
   ↪   processing unit.
16 * @remark Note that this service does not support nesting. If
   ↪   nesting is needed for critical sections e.g: for libraries
   ↪   SuspendOSInterrupts/ResumeOSInterrupts or
   ↪   SuspendAllInterrupt/ResumeAllInterrupts should be used.
17 */
18 void DisableAllInterrupts( void );
19
20
21 /**
22 * This service restores the recognition status of all interrupts
   ↪   saved by the SuspendAllInterrupts service.
23 *
24 * @remark The service may be called from an ISR category 1 and
   ↪   category 2, from alarm-callbacks and from the task level, but
   ↪   not from all hook routines.
25 * @remark This service is the counterpart of SuspendAllInterrupts
   ↪   service, which has to have been called before, and its aim is
   ↪   the completion of the critical section of code. No API service
   ↪   calls beside SuspendAllInterrupts/ResumeAllInterrupts pairs
   ↪   and SuspendOSInterrupts/ResumeOSInterrupts pairs are allowed
   ↪   within this critical section.
26 * @remark The implementation should adapt this service to the
   ↪   target hardware providing a minimum overhead.
27 * @remark SuspendAllInterrupts/ResumeAllInterrupts can be nested.
   ↪   In case of nesting pairs of the calls SuspendAllInterrupts and
   ↪   ResumeAllInterrupts the interrupt recognition status saved by
   ↪   the first call of SuspendAllInterrupts is restored by the last
   ↪   call of the ResumeAllInterrupts service.
28 */
29 void ResumeAllInterrupts( void );
30
31 /**
32 * This service saves the recognition status of all interrupts and
   ↪   disables all interrupts for which the hardware supports
   ↪   disabling.
33 * @remark The service may be called from an ISR category 1 and
   ↪   category 2, from alarm-callbacks and from the task level, but
   ↪   not from all hook routines.
34 * @remark This service is intended to protect a critical section of
   ↪   code from interruptions of any kind. This section shall be
   ↪   finished by calling the ResumeAllInterrupts service. No API
   ↪   service calls beside SuspendAllInterrupts/ResumeAllInterrupts
   ↪   pairs and SuspendOSInterrupts/ResumeOSInterrupts pairs are
   ↪   allowed within this critical section.
35 * @remark The implementation should adapt this service to the
   ↪   target hardware providing a minimum overhead.
36 */
```

```
37 void SuspendAllInterrupts( void );
38
39 /**
40 * This service restores the recognition status of interrupts saved
   ↪  by the SuspendOSInterrupts service.
41 *
42 * @remark The service may be called from an ISR category 1 and
   ↪  category 2 and from the task level, but not from hook
   ↪  routines.
43 * @remark This service is the counterpart of SuspendOSInterrupts
   ↪  service, which has to have been called before, and its aim is
   ↪  the completion of the critical section of code. No API service
   ↪  calls beside SuspendAllInterrupts/ResumeAllInterrupts pairs
   ↪  and SuspendOSInterrupts/ResumeOSInterrupts pairs are allowed
   ↪  within this critical section.
44 * @remark The implementation should adapt this service to the
   ↪  target hardware providing a minimum overhead.
45 * @remark SuspendOSInterrupts/ResumeOSInterrupts can be nested. In
   ↪  case of nesting pairs of the calls SuspendOSInterrupts and
   ↪  ResumeOSInterrupts the interrupt recognition status saved by
   ↪  the first call of SuspendOSInterrupts is restored by the last
   ↪  call of the ResumeOSInterrupts service.
46 */
47 void ResumeOSInterrupts( void );
48
49 /**
50 * This service saves the recognition status of interrupts of
   ↪  category 2 and disables the recognition of these interrupts.
51 * @remark The service may be called from an ISR and from the task
   ↪  level, but not from hook routines.
52 * @remark This service is intended to protect a critical section of
   ↪  code. This section shall be finished by calling the
   ↪  ResumeOSInterrupts service. No API service calls beside
   ↪  SuspendAllInterrupts/ResumeAllInterrupts pairs and
   ↪  SuspendOSInterrupts/ResumeOSInterrupts pairs are allowed
   ↪  within this critical section.
53 * @remark The implementation should adapt this service to the
   ↪  target hardware providing a minimum overhead.
54 * @remark It is intended only to disable interrupts of category 2.
   ↪  However, if this is not possible in an efficient way more
   ↪  interrupts may be disabled.
55 */
56 void SuspendOSInterrupts( void );
```

Algoritmo 15: Interfaces do OSEK/VDX - Rotinas de SO.

```
1 /**
2 * This service returns the current application mode. It may be used
   ↪  to write mode dependent code.
```

```
3  *
4  * @remark See chapter 5 for a general description of application
   ↪   modes.
5  * @remark Allowed for task, ISR and all hook routines.
6  *
7  * @return The current application mode.
8  *
9  * @remark [Conformance] BCC1, BCC2, ECC1, ECC2
10 */
11 AppModeType GetActiveApplicationMode( void );
12
13 /**
14 * The user can call this system service to start the operating
   ↪   system in a specific mode, see chapter 5, Application modes.
15 *
16 * @remark Only allowed outside of the operating system, therefore
   ↪   implementation specific restrictions may apply. See also
   ↪   chapter 11.3, System start-up, especially with respect to
   ↪   systems where OSEK and OSEKtime coexist. This call does not
   ↪   need to return.
17 *
18 * @param[in] Mode The application mode to start the OS.
19 *
20 * @remark [Conformance] BCC1, BCC2, ECC1, ECC2
21 */
22 void StartOS( AppModeType Mode );
23
24 /**
25 * @brief The user can call this system service to abort the overall
   ↪   system (e.g. emergency off). The operating system also calls
   ↪   this function internally, if it has reached an undefined
   ↪   internal state and is no longer ready to run.
26 * @brief If a ShutdownHook is configured the hook routine
   ↪   ShutdownHook is always called (with <Error> as argument)
   ↪   before shutting down the operating system. If ShutdownHook
   ↪   returns, further behaviour of ShutdownOS is implementation
   ↪   specific.
27 * @brief In case of a system where OSEK OS and OSEKtime OS coexist,
   ↪   ShutdownHook has to return.
28 * @brief <Error> needs to be a valid error code supported by OSEK
   ↪   OS. In case of a system where OSEK OS and OSEKtime OS coexist,
   ↪   <Error> might also be a value accepted by OSEKtime OS. In this
   ↪   case, if enabled by an OSEKtime configuration parameter,
   ↪   OSEKtime OS will be shut down after OSEK OS shutdown.
29 *
30 * @remark After this service the operating system is shut down.
31 * @remark Allowed at task level, ISR level, in ErrorHook and
   ↪   StartupHook, and also called internally by the operating
   ↪   system.
32 * @remark If the operating system calls ShutdownOS it never uses
   ↪   E_OK as the passed parameter value.
```

```
33 *
34 * @param[in]  Error  The error
35 */
36 void ShutdownOS( StatusType Error );
```

Algoritmo 16: Interfaces do OSEK/VDX - Rotinas de Recursos.

```
1  /**
2  * DeclareResource serves as an external declaration of a resource.
   ↪  The function and use of this service are similar to that of
   ↪  the external declaration of variables.
3  *
4  * @param[in] ResourceIdentifier Resource identifier (Duh).
5  */
6  #define DeclareResource( ResourceIdentifier ) extern ResourceType
   ↪  ##ResourceIdentifier;
7
8  /**
9  * This call serves to enter critical sections in the code that are
   ↪  assigned to the resource referenced by ResID. A critical
   ↪  section shall always be left using ReleaseResource.
10 *
11 * @remark The OSEK priority ceiling protocol for resource
   ↪  management is described in chapter 8.5.
12 * @remark Nested resource occupation is only allowed if the inner
   ↪  critical sections are completely executed within the
   ↪  surrounding critical section (strictly stacked, see chapter
   ↪  8.2, Restrictions when using resources). Nested occupation of
   ↪  one and the same resource is also forbidden!
13 * @remark It is recommended that corresponding calls to GetResource
   ↪  and ReleaseResource appear within the same function.
14 * @remark It is not allowed to use services which are points of
   ↪  rescheduling for non preemptable tasks (TerminateTask,
   ↪  ChainTask, Schedule and WaitEvent, see chapter 4.6.2) in
   ↪  critical sections. Additionally, critical sections are to be
   ↪  left before completion of an interrupt service routine.
15 * @remark Generally speaking, critical sections should be short.
16 * @remark The service may be called from an ISR and from task level
   ↪  (see Figure 12-1).
17 *
18 * @param[in] ResID Reference to resource
19 *
20 * @return [Standard] No error, E_OK.
21 * @return [Extended] Resource <ResID> is invalid, E_OS_ID.
22 * @return [Extended] Attempt to get a resource which is already
   ↪  occupied by any task or ISR, or the statically assigned
   ↪  priority of the calling task or interrupt routine is higher
   ↪  than the calculated ceiling priority, E_OS_ACCESS
23 */
```

```
24 StatusType GetResource( ResourceType ResID );
25
26 /**
27 * ReleaseResource is the counterpart of GetResource and serves to
   ↪  leave critical sections in the code that are assigned to the
   ↪  resource referenced by ResID.
28 *
29 * @param[in] ResID Reference to resource.
30 *
31 * @remark For information on nesting conditions, see
   ↪  particularities of GetResource.
32 * @remark The service may be called from an ISR and from task level
   ↪  (see Figure 12-1).
33 *
34 * @return [Standard] No error, E_OK.
35 * @return [Extended] Resource <ResID> is invalid, E_OS_ID.
36 * @return Attempt to release a resource which is not occupied by
   ↪  any task or ISR, or another resource shall be released before,
   ↪  E_OS_NOFUNC.
37 * @return Attempt to release a resource which has a lower ceiling
   ↪  priority than the statically assigned priority of the calling
   ↪  task or interrupt routine, E_OS_ACCESS.
38 */
39 StatusType ReleaseResource( ResourceType ResID );
```

Algoritmo 17: Interfaces do OSEK/VDX - Rotinas de Escalonamento.

```
1  /**
2  * DeclareTask serves as an external declaration of a task. The
3  * function and use of this service are similar to that of the
   ↪  external
4  * declaration of variables.
5  */
6  #define DeclareTask( TaskName, TaskID ) const TaskType
   ↪  TaskType_##TaskName = (TaskID)
7
8  #define TASK( TaskName ) void TASK_FUNC_##TaskName(void)
9
10 ////////////////////////////////////////////////////////////////////////////
11 //  System Services
12 ////////////////////////////////////////////////////////////////////////////
13
14 /**
15 * The task TaskID is transferred from the suspended state into the
   ↪  ready state. The operating system ensures that the task code
   ↪  is being executed from the first statement.
16 *
```

```
17 * @remark When an extended task is transferred from suspended state
   ↪   into ready state all its events are cleared.
18 *
19 * @remark The service may be called from interrupt level and from
   ↪   task level (see Figure 12-1). Rescheduling after the call to
   ↪   ActivateTask depends on the place it is called from (ISR, non
   ↪   preemptable task, preemptable task).
20 *
21 * If E_OS_LIMIT is returned the activation is ignored.
22 *
23 * @param TaskID[in] Task reference
24 *
25 * @return [Standard] No error, E_OK.
26 * @return [Extended] Too many task activations of TaskID,
   ↪   E_OS_LIMIT
27 * @return [Extended] Task TaskID is invalid, E_OS_ID
28 *
29 * @remark [Conformance] BCC1, BCC2, ECC1, ECC2
30 */
31 StatusType ActivateTask( TaskType TaskID );
32
33 /**
34 * This service causes the termination of the calling task. The
   ↪   calling task is transferred from the running state into the
   ↪   suspended state.
35 *
36 * @remark An internal resource assigned to the calling task is
   ↪   automatically released. Other resources occupied by the task
   ↪   shall have been released before the call to TerminateTask. If
   ↪   a resource is still occupied in standard status the behaviour
   ↪   is undefined.
37 * @remark If the call was successful, TerminateTask does not return
   ↪   to the call level and the status can not be evaluated. If the
   ↪   version with extended status is used, the service returns in
   ↪   case of error, and provides a status which can be evaluated in
   ↪   the application. If the service TerminateTask is called
   ↪   successfully, it enforces a rescheduling.
38 * @remark Ending a task function without call to TerminateTask or
   ↪   ChainTask is strictly forbidden and may leave the system in an
   ↪   undefined state.
39 *
40 * @return Task still occupies resources, E_OS_RESOURCE
41 * @return Call at interrupt level, E_OS_CALLEVEL
42 */
43 StatusType TerminateTask( void );
44
45 /**
```

```
46 * This service causes the termination of the calling task. After
   ↪   termination of the calling task a succeeding task <TaskID> is
   ↪   activated. Using this service, it ensures that the succeeding
   ↪   task starts to run at the earliest after the calling task has
   ↪   been terminated.
47 *
48 * @remark If the succeeding task is identical with the current
   ↪   task, this does not result in multiple requests. The task is
   ↪   not transferred to the suspended state, but will immediately
   ↪   become ready again.
49 * @remark An internal resource assigned to the calling task is
   ↪   automatically released, even if the succeeding task is
   ↪   identical with the current task. Other resources occupied by
   ↪   the calling shall have been released before ChainTask is
   ↪   called. If a resource is still occupied in standard status the
   ↪   behaviour is undefined.
50 * @remark If called successfully, ChainTask does not return to the
   ↪   call level and the status can not be evaluated.
51 * @remark In case of error the service returns to the calling task
   ↪   and provides a status which can then be evaluated in the
   ↪   application.
52 * @remark If the service ChainTask is called successfully, this
   ↪   enforces a rescheduling.
53 * @remark Ending a task function without call to TerminateTask or
   ↪   ChainTask is strictly forbidden and may leave the system in an
   ↪   undefined state.
54 * @remark If E_OS_LIMIT is returned the activation is ignored. When
   ↪   an extended task is transferred from suspended state into
   ↪   ready state all its events are cleared.
55 *
56 * @param TaskID[in] Reference to the sequential succeeding task to
   ↪   be activated.
57 *
58 * @return [Standard] No return to call level.
59 * @return [Standard] Too many task activations of <TaskID>,
   ↪   E_OS_LIMIT.
60 * @return [Extended] Task <TaskID> is invalid, E_OS_ID.
61 * @return [Extended] Calling task still occupies resources,
   ↪   E_OS_RESOURCE.
62 * @return [Extended] Call at interrupt level, E_OS_CALLEVEL.
63 */
64 StatusType ChainTask( TaskType TaskID );
65
66 /**
67 * GetTaskID returns the information about the TaskID of the task
   ↪   which is currently running.
68 *
69 * @remark Allowed on task level, ISR level and in several hook
   ↪   routines (see Figure 12-1).
70 * @remark This service is intended to be used by library functions
   ↪   and hook routines.
```

```
71 * @remark If <TaskID> can't be evaluated (no task currently
   ↪  running), the service returns INVALID_TASK as TaskType.
72 *
73 * @param TaskID[out] Reference to the task which is currently
   ↪  running.
74 *
75 * @return [Standard] No error, E_OK
76 * @return [Extended] No error, E_OK
77 */
78 StatusType GetTaskID( TaskRefType TaskID );
79
80 /**
81 * Returns the state of a task (running, ready, waiting, suspended)
   ↪  at the time
82 * of calling GetTaskState.
83 *
84 * @remark The service may be called from interrupt service
   ↪  routines, task level, and some hook routines (see Figure
   ↪  12-1).
85 * @remark When a call is made from a task in a full preemptive
   ↪  system, the result may already be incorrect at the time of
   ↪  evaluation.
86 * @remark When the service is called for a task, which is activated
   ↪  more than once, the state is set to running if any instance of
   ↪  the task is running.
87 *
88 * @param TaskID[in] Task reference
89 * @param State[out] Reference to the state of the task TaskID
90 *
91 * @return [Standard] No error, E_OK
92 * @return [Extended] Task TaskID is invalid, E_OS_ID
93 */
94 StatusType GetTaskState( TaskType TaskID, TaskStateRefType State );
95
96 /**
97 * If a higher-priority task is ready, the internal resource of the
   ↪  task is released, the current task is put into the ready
   ↪  state, its context is saved and the higher-priority task is
   ↪  executed. Otherwise the calling task is continued.
98 *
99 * @remark Rescheduling only takes place if the task an internal
   ↪  resource is assigned to the calling task during system
   ↪  generation. For these tasks, Schedule enables a processor
   ↪  assignment to other tasks with lower or equal priority than
   ↪  the ceiling priority of the internal resource and higher
   ↪  priority than the priority of the calling task in
   ↪  application-specific locations. When returning from Schedule,
   ↪  the internal resource has been taken again. This service has
   ↪  no influence on tasks with no internal resource assigned
   ↪  (preemptable tasks).
100 *
```

```
101 * @return [Standard] No error, E_OK.
102 * @return [Extended] Call at interrupt level, E_OS_CALLEVEL
103 * @return [Extended] Calling task occupies resources, E_OS_RESOURCE
104 */
105 StatusType Schedule( void );
```

ANEXO B – OpenAUTOS API: Rotinas de Uso Interno do SO

Algoritmo 18: Interfaces do OpenAUTOS - Rotinas de Recursos.

```
1  /**Type for the Resource Id*/
2  typedef uint8_t ResourceType;
3  /**Type for the Resource ceiling Priority*/
4  typedef uint8_t ResourceDataPriorityType;
5
6  /**
7  * @brief The structure that stores data about the resource. Also
   ↪  serves as a linked list node
8  *
9  * @remark Due to an odd bug with the xc8 1.37 compiler, where he
   ↪  does not allow it to declare an extern const
10 * as the size of an array (altough it works sometimes), I changed
   ↪  the model to use "movable" nodes with linked lists.
11 *
12 * Basically, every Task and ISR has a variable which is used as the
   ↪  start of its own linked list. There is also a
13 * global variable (g_resources) which is the main linked list for
   ↪  resources.
14 *
15 * Initially, all resources are allocated to the g_resources linked
   ↪  list. When a Task or ISR requests a resource,
16 * it is moved from g_resources to the linked list in the Task/ISR.
   ↪  When this resource is released, it is put back
17 * in the g_resources list.
18 */
19 typedef struct SResourceDataType {
20     ResourceType id;                    ///< The unique identifier
       ↪  for this resource.
21     ResourceDataPriorityType priority;  ///< The ceiling priority
       ↪  for this resource
22     struct SResourceDataType* next;     ///< Link to the next node
23     struct SResourceDataType* prev;     ///< Link to the previous
       ↪  node
24 } ResourceDataType;
25 /**Type for pointers of the ResourceDataType*/
26 typedef ResourceDataType* ResourceDataRefType;
27
28 /**
29 * Initializes an empty List of resource Data.
30 */
31 void InitializeResourceDataList( ResourceDataRefType List );
32 /**
33 * Moves a global resource data to the global list of resources. It
   ↪  also initializes this node.
34 */
35 void AddResourceDataToResources( ResourceDataRefType Resource,
   ↪  ResourceType ResID, ResourceDataPriorityType Priority );
36
```

```
37 /**
38 * A linked list to the resources of this project.
39 */
40 extern ResourceDataType g_resources;
41
42 /**
43 * Finds the resource with ResID in the given list
44 */
45 ResourceDataRefType FindResource( ResourceDataRefType First,
   ↪  ResourceType ResID );
46 /**
47 * Moves the resource with ResID from one linked list to another.
48 */
49 void MoveResourceData( ResourceType ResID, ResourceDataRefType
   ↪  From, ResourceDataRefType To );
```

Algoritmo 19: Interfaces do OpenAUTOS - Rotinas de Configuração.

```
1 /**
2 * Initializes the internal structures of the OS.
3 */
4 void Setup();
```

Algoritmo 20: Interfaces do OpenAUTOS - Rotinas do Contador do Sistema.

```
1 /**
2 * Initializes the internal system counter
3 */
4 void InitializeSystemCounter();
5
6 /**
7 * Detects if the system counter has triggered
8 */
9 uint8_t HasInterruptSystemCounter();
10
11 /**
12 * A callback registered to the system counter will be called by
   ↪  this rotine.
13 */
14 void TimeoutRoutineSystemCounter();
15
16 /**
17 * Resets the system counter to perform another countdown.
18 */
19 void ResetSystemCounter();
```

```
20
21 /**
22 * The type for the callback rotine for TimeoutRoutineSystemCounter.
23 */
24 typedef void (*TimeoutCallbackSystemCounterType)();
25
26 /**
27 * The callback rotine to be called by TimeoutRoutineSystemCounter
28 */
29 extern TimeoutCallbackSystemCounterType
   ↪  g_callback_timeoutsystemcounter;
```

Algoritmo 21: Interfaces do OpenAUTOS - Rotinas de Tarefas.

```
1  /**Type for task priority*/
2  typedef uint8_t TaskPriorityType;
3  /**Type for task callback method*/
4  typedef CallbackType TaskCallbackType;
5
6  /**
7  * This type represents a Task in the operating system.
8  */
9  typedef struct STaskDataType {
10     TaskType id;                                          ///<
       ↪  Task "unique" identifier. Actually, this indicates the
       ↪  "type" of the task.
11     TaskPriorityType priority;                             ///<
       ↪  Current priority of the task
12     TaskPriorityType priority_base;                        ///<
       ↪  Original priority of the task (Resets to this value after
       ↪  releasing a resource).
13     TaskStateType state;                                  ///<
       ↪  Current state of the task (RUNNING, WAITING, SUSPENDED or
       ↪  READY).
14     TaskContextType context;                              ///<
       ↪  The context of the task. Used for prempting tasks.
15     TaskCallbackType callback;                             ///<
       ↪  The "body" of the task.
16     struct STaskDataType* next_task_same_priority;         ///<
       ↪  Links together tasks that have the same priority.
17     ResourceDataType resources;                           ///<
       ↪  Keeps a stack of the resources allocated, in order to know
       ↪  the next resource that shall be released.
18 } TaskDataType;
19 /**A pointer type for tasks*/
20 typedef TaskDataType* TaskDataRefType;
21
22 /**
23 * Initializes a TaskData item passed by reference.
```

```
24 */
25 void InitializeTaskData( TaskDataRefType Task, TaskType Id,
   ↪  TaskPriorityType Priority, TaskStateType State,
   ↪  TaskCallbackType Callback );
26
27 /**
28 * Groups tasks with the same priority for the Round-Robin
   ↪  Scheduler.
29 */
30 void GroupTasksSamePriority();
```

Algoritmo 22: Interfaces do OpenAUTOS - Rotinas para Contexto de Tarefas.

```
1 typedef PlatformTaskContextType TaskContextType;
2 typedef PlatformTaskContextRefType TaskContextRefType;
3
4 /**
5 * Saves the context data of a task
6 */
7 #define SaveTaskContext(TaskContextRef)
   ↪  PlatformSaveTaskContext((TaskContextRef))
8
9 /**
10 * Loads the context data of a task
11 */
12 #if defined(PLATFORM) &&  PLATFORM == PIC18F25K80
13 #define LoadTaskContext(TaskContextRef,FuncNameStr)
   ↪  PlatformLoadTaskContext((TaskContextRef),FuncNameStr)
14 #else
15 #define LoadTaskContext(TaskContextRef)
   ↪  PlatformLoadTaskContext((TaskContextRef))
16 #endif
```

ANEXO C – OpenAUTOS API: Rotinas especificas para as plataformas

Algoritmo 23: Interfaces da Plataforma - Rotinas de Configuração.

```
/**
* Pragma configuration of the PIC18F25K80 bit settings
*/

// CONFIG1L
#pragma config RETEN = OFF      // VREG Sleep Enable bit (Ultra
↪  low-power regulator is Disabled (Controlled by REGSLP bit))
#pragma config INTOSCSEL = HIGH // LF-INTOSC Low-power Enable bit
↪  (LF-INTOSC in High-power mode during Sleep)
#pragma config SOSCSEL = HIGH   // SOSC Power Selection and mode
↪  Configuration bits (High Power SOSC circuit selected)
#pragma config XINST = OFF      // Extended Instruction Set
↪  (Disabled)

// CONFIG1H
#pragma config FOSC = INTIO1    // Oscillator (Internal RC
↪  oscillator, CLKOUT function on OSC2)
#pragma config PLLCFG = ON      // PLL x4 Enable bit (Enabled)
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor
↪  (Disabled)
#pragma config IESO = OFF       // Internal External Oscillator
↪  Switch Over Mode (Disabled)

// CONFIG2L
#pragma config PWRTEN = OFF     // Power Up Timer (Disabled)
#pragma config BOREN = OFF      // Brown Out Detect (Disabled in
↪  hardware, SBOREN disabled)
#pragma config BORV = 3         // Brown-out Reset Voltage bits
↪  (1.8V)
#pragma config BORPWR = ZPBORMV // BORMV Power level (ZPBORMV
↪  instead of BORMV is selected)

// CONFIG2H
#pragma config WDTEN = OFF      // Watchdog Timer (WDT disabled in
↪  hardware; SWDTEN bit disabled)
#pragma config WDTPS = 1048576  // Watchdog Postscaler (1:1048576)

// CONFIG3H
#pragma config CANMX = PORTB    // ECAN Mux bit (ECAN TX and RX
↪  pins are located on RB2 and RB3, respectively)
#pragma config MSSPMSK = MSK7   // MSSP address masking (7 Bit
↪  address masking mode)
#pragma config MCLRE = ON       // Master Clear Enable (MCLR
↪  Enabled, RE3 Disabled)

// CONFIG4L
#pragma config STVREN = ON      // Stack Overflow Reset (Enabled)
```

```
34 #pragma config BBSIZ = BB2K      // Boot Block Size (2K word Boot
   ↪  Block size)
35
36 // CONFIG5L
37 #pragma config CP0 = OFF         // Code Protect 00800-01FFF
   ↪  (Disabled)
38 #pragma config CP1 = OFF         // Code Protect 02000-03FFF
   ↪  (Disabled)
39 #pragma config CP2 = OFF         // Code Protect 04000-05FFF
   ↪  (Disabled)
40 #pragma config CP3 = OFF         // Code Protect 06000-07FFF
   ↪  (Disabled)
41
42 // CONFIG5H
43 #pragma config CPB = OFF         // Code Protect Boot (Disabled)
44 #pragma config CPD = OFF         // Data EE Read Protect (Disabled)
45
46 // CONFIG6L
47 #pragma config WRT0 = OFF        // Table Write Protect 00800-01FFF
   ↪  (Disabled)
48 #pragma config WRT1 = OFF        // Table Write Protect 02000-03FFF
   ↪  (Disabled)
49 #pragma config WRT2 = OFF        // Table Write Protect 04000-05FFF
   ↪  (Disabled)
50 #pragma config WRT3 = OFF        // Table Write Protect 06000-07FFF
   ↪  (Disabled)
51
52 // CONFIG6H
53 #pragma config WRTC = OFF        // Config. Write Protect (Disabled)
54 #pragma config WRTB = OFF        // Table Write Protect Boot
   ↪  (Disabled)
55 #pragma config WRTD = OFF        // Data EE Write Protect (Disabled)
56
57 // CONFIG7L
58 #pragma config EBTR0 = OFF       // Table Read Protect 00800-01FFF
   ↪  (Disabled)
59 #pragma config EBTR1 = OFF       // Table Read Protect 02000-03FFF
   ↪  (Disabled)
60 #pragma config EBTR2 = OFF       // Table Read Protect 04000-05FFF
   ↪  (Disabled)
61 #pragma config EBTR3 = OFF       // Table Read Protect 06000-07FFF
   ↪  (Disabled)
62
63 // CONFIG7H
64 #pragma config EBTRB = OFF       // Table Read Protect Boot
   ↪  (Disabled)
65
66 // #pragma config statements should precede project file includes.
67 // Use project enums instead of #define for ON and OFF.
68
69 #include <xc.h>
```

```
70
71 /**
72 * Performs the Setup specific to this plataform
73 */
74 void PlatformSetup();
```

Algoritmo 24: Interfaces da Plataforma - Rotinas do Contador do Sistema.

```
1 /**
2 * Performs the reset of the system counter in this platform.
3 */
4 void PlatformResetSystemCounter();
5
6 /**
7 * Performs the initialization of the system clock in this platform.
8 */
9 void PlatformInitializeSystemCounter();
10
11 /**
12 * Checks if there is an interruption due to countdown in this
   ↪  platform.
13 */
14 uint8_t PlatformHasInterruptSystemCounter();
```

Algoritmo 25: Interfaces da Plataforma - Rotinas para Contexto de Tarefas.

```
1 #define PLATFORM_CONTEXT_STACK_SIZE 31
2 // Variables used to temporaly store the registers (otherwise, the
   ↪  value would be lost)
3 extern uint8_t _wreg;
4 extern uint8_t _bsr;
5 extern uint8_t _status;
6
7 /**Retrives the value part of the uController Stack */
8 #define uControllerStackValue() ( STKPTRbits.STKPTR )
9
10 typedef uint32_t PlatformTaskContextStackType;
11 typedef PlatformTaskContextStackType*
   ↪  PlatformTaskContextStackRefType;
12
13 /**
14 * PIC18F25K80 Data struct to store task context information
15 */
16 typedef struct {
17     uint8_t work;
```

```
18      uint8_t bsr;
19      uint8_t status;
20      PlatformTaskContextStackType
        ↪  stack[PLATFORM_CONTEXT_STACK_SIZE];
21      uint8_t stack_top;
22  } PlatformTaskContextType;
23
24  typedef PlatformTaskContextType* PlatformTaskContextRefType;
25
26  /**
27  * @brief (Re)Initializes the context of a task
28  *
29  * @param[in] Context A pointer to the task's context to be reset.
30  *
31  * @return No error, E_OS_OK.
32  */
33  StatusType ResetTaskContext( PlatformTaskContextRefType Context,
    ↪  CallbackType Callback );
34
35  #ifndef PUSH
36  #define PUSH() asm(" PUSH")
37  #endif
38
39  #ifndef POP
40  #define POP() asm(" POP")
41  #endif
42
43
44
45  /**
46  * @brief Clears the uController's stack and saves it in the current
    ↪  task stack.
47  *
48  * @param[in] TaskContextRef Context The context of the current
    ↪  active task
49  */
50  #define PlatformSaveTaskContext( TaskContextRef )
    ↪  \
51  do {
    ↪  \
52      TaskContextRef->work = _wreg;
    ↪  \// Saving Work Register
53      TaskContextRef->bsr = _bsr;
        ↪  \// Saving Bank Select Register
54      TaskContextRef->status = _status;
        ↪  \// Saving Status Register
55      PlatformTaskContextStackRefType stack =
        ↪  (TaskContextRef->stack);            \// Shortcut to the
        ↪  stack
```

```
56      while( uControllerStackValue() > 0 ) {
        ↪  \// Loops Through the values on the Stack
57          uint8_t i = TaskContextRef->stack_top++;
            ↪  \//     Gets the position for the local stack and then
            ↪  increases the top of the local stack
58          stack[i] = TOS;
            ↪  \//     Saves the uController's value that on the top
            ↪  of the stack
59          POP();
            ↪  \//     Removes the top most value from the
            ↪  uController's stack
60      }
        ↪  \
61  } while(0)
62
63  /**
64  * @brief Loads the uController's stack and clears the current task
    ↪  stack.
65  *
66  * @param[in] TaskContextRef Context The context of the current
    ↪  active task
67  */
68  #define PlatformLoadTaskContext( TaskContextRef, FuncNameStr )
    ↪  \
69  do {
    ↪  \
70      PlatformTaskContextStackRefType stack =
    ↪  (TaskContextRef->stack);             \// Shortcut to the stack
71      while( TaskContextRef->stack_top > 0 ) {
        ↪  \// Loops while there are values to push to the Stack
72          uint8_t i = --TaskContextRef->stack_top;
            ↪  \//     Reduces the size of the stack and then gets
            ↪  the index position
73          PUSH();
            ↪  \//     Moves the uC stack position to the next
            ↪  available
74          uint32_t callback = stack[i];
            ↪  \
75          uint8_t tosu = (uint8_t)(callback >> 16);
            ↪  \
76          uint8_t tosh = (uint8_t)(callback >>  8);
            ↪  \
77          uint8_t tosl = (uint8_t)(callback >>  0);
            ↪  \
78          asm("BANKSEL("FuncNameStr"@tosu)");
            ↪  \
79          asm("MOVF (("FuncNameStr"@tosu) and 0FFh),w");
            ↪  \
```

```
80          asm("MOVWF TOSU");
             ↪  \
81          asm("BANKSEL("FuncNameStr"@tosh)");
             ↪  \
82          asm("MOVF (("FuncNameStr"@tosh) and 0FFh),w");
             ↪  \
83          asm("MOVWF TOSH");
             ↪  \
84          asm("BANKSEL("FuncNameStr"@tosl)");
             ↪  \
85          asm("MOVF (("FuncNameStr"@tosl) and 0FFh),w");
             ↪  \
86          asm("MOVWF TOSL");
             ↪  \
87      }
         ↪  \
88      _wreg = TaskContextRef->work;
         ↪  \// Loading the Work Register
89      _bsr = TaskContextRef->bsr;
         ↪  \// Loading Bank Select Register
90      _status = TaskContextRef->status;
         ↪  \// Loading Status Register
91  } while(0)
```