Luís Fernando Arcaro

IMPLEMENTAÇÃO DE UM SISTEMA OPERACIONAL COMPATÍVEL COM A ESPECIFICAÇÃO ARINC 653

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas para a obtenção do Grau de Mestre em Engenharia de Automação e Sistemas.
Orientador: Rômulo Silva de Oliveira, Dr.

Florianópolis

2015

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

Arcaro, Luís Fernando
Implementação de um Sistema Operacional Compatível com a Especificação ARINC 653 / Luís Fernando Arcaro ; orientador, Rômulo Silva de Oliveira - Florianópolis, SC, 2015.
351 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de Automação e Sistemas. 2. Sistemas de Tempo Real. 3. Sistemas Operacionais. 4. ARINC 653. I. Oliveira, Rômulo Silva de. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. III. Título.

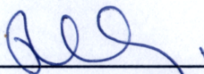
IMPLEMENTAÇÃO DE UM SISTEMA OPERACIONAL COMPATÍVEL COM A ESPECIFICAÇÃO ARINC 653

Luís Fernando Arcaro

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Engenharia de Automação e Sistemas.

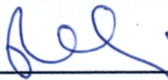
Florianópolis, 24 de fevereiro de 2015.

Rômulo Silva de Oliveira, Dr.
Orientador



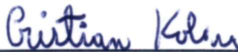
Rômulo Silva de Oliveira, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia de Automação e Sistemas

Banca Examinadora:

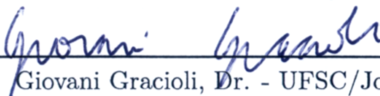


Rômulo Silva de Oliveira, Dr.
Presidente

Carlos Alberto Maziero, Dr. - UTFPR/Curitiba



Cristian Koliver, Dr. - UFSC/Florianópolis



Giovani Gracioli, Dr. - UFSC/Joinville

AGRADECIMENTOS

Agradeço aos meus amados pais, Izabel e Lourenço, pelo apoio incondicional e pela educação que me deram desde sempre: não há palavras que expressem minha gratidão. Agradeço também à minha namorada, Vanessa, pelo amor, pela compreensão (inclusive da distância) e por estar ao meu lado incansavelmente durante todo este tempo. Agradeço ainda à minha irmã Katia e ao cunhado Guilherme: a ela pelas primeiras aulas de minha vida, e a ambos pelo companheirismo que lhes é característico. Vocês sempre foram, sem dúvida, o alicerce de todo projeto que já realizei.

Aos professores, agradeço pelo conhecimento transferido e pelas experiências compartilhadas, e repito uma frase já dita por mim em outra ocasião: se há um caminho para um melhor futuro, esse caminho passa por vocês. Agradeço especialmente ao professor Rômulo pela orientação e pela paciência durante o desenvolvimento deste trabalho, e ao professor Ricardo Vargas Dorneles, pela amizade e pela sempre construtiva troca de ideias sobre a vida acadêmica.

Agradeço também aos colegas e amigos conquistados nesta jornada, pelo compartilhamento das alegrias e dificuldades inerentes ao desafio ao qual nos propusemos e por estarem sempre dispostos a prestar auxílio, trocar experiências e comemorar conquistas.

Finalmente agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo financiamento que tornou possível este trabalho.

A todos vocês, minha sincera gratidão.

Luís Fernando Arcaro

RESUMO

Sistemas de Tempo Real (STRs) são sistemas computacionais que estão submetidos, além de a requisitos lógicos, a requisitos de natureza temporal. A especificação ARINC 653 descreve a interface funcional a ser oferecida ao *software* de aplicação e os requisitos temporais a serem atendidos por Sistemas Operacionais (SOs) sobre os quais são executados STRs aviônicos, ou seja, aplicações relacionadas aos sistemas eletrônicos empregados em aeronaves. Uma das principais definições dessa especificação é a utilização de partições com isolamento temporal e espacial, permitindo a execução de múltiplas aplicações com diferentes objetivos e criticalidades numa mesma plataforma de *hardware*, e garantindo, ainda, que eventuais falhas ocorridas em uma partição não afetem a execução das demais. Este trabalho descreve a implementação de um SO compatível com a especificação ARINC 653 destinado ao treinamento de profissionais e à experimentação de novas aplicações aviônicas, não tendo por objetivo, portanto, a certificação para operação em voo, executado sobre a plataforma de *hardware* comercialmente conhecida como *BeagleBone*. Foi desenvolvido ainda um conjunto de ferramentas de configuração destinadas à validação de arquivos de configuração no padrão exigido pela ARINC 653 e à geração de modelos de aplicação para execução no SO. O atendimento às exigências da ARINC 653 pelo SO foi evidenciado através de um conjunto de casos de teste, que servem também como exemplos de utilização e auxiliam, ainda, no processo de migração do SO a outras plataformas.

Palavras-chave: Tempo real. Sistemas operacionais. ARINC 653.

ABSTRACT

Real-Time Systems (RTSs) are computer systems that are subject to, in addition to logical requirements, temporal requirements. The ARINC 653 specification describes the functional interface that must be offered to the application software and the time requirements that must be met by Operating Systems (OSs) on which avionics RTSs are executed, i.e. applications related to electronic systems used on aircrafts. One of the key definitions of this specification is the use of temporally and spatially isolated partitions, allowing the execution of multiple applications with different objectives and criticality levels on the same hardware platform, and also ensuring that any failures in one partition do not affect the execution of the others. This work describes the implementation of an ARINC 653 compatible OS aimed for professionals training and test of new avionics applications, and thus does not aim certification for in-flight operation, which is executed on the hardware platform commercially known as BeagleBone. We also developed a set of configuration tools for the validation of ARINC 653 standard configuration files and the generation of application templates to be run in the OS. The compliance of the OS with the ARINC 653 requirements is evidenced through a set of test cases, which also serve as examples of use and are useful in the process of migration of the OS to other platforms.

Keywords: Real-time. Operating systems. ARINC 653.

LISTA DE FIGURAS

Figura 1 Hierarquia de elementos de um sistema 28
Figura 2 Estados de processos 38
Figura 3 Plataforma *BeagleBone* 53
Figura 4 Tabelas de tradução de descritores curtos 61
Figura 5 Componentes do SO 72
Figura 6 Processo de compilação e execução 75
Figura 7 Tarefas de sistemas ARINC 653 81
Figura 8 Alocação das regiões de pilha 113
Figura 9 Geração e atualização de modelos 143
Figura 10 Bancada de testes 152
Figura 11 Preparação da *BeagleBone* 153
Figura 12 Captura do teste 001 155
Figura 13 Gerenciamento, compilação e execução de modelos de aplicação 177

LISTA DE TABELAS

Tabela 1 Mapeamento TTBR0-TTBR1 61
Tabela 2 Regiões de memória 93
Tabela 3 Permissões para a região VECTBL 96
Tabela 4 Permissões para a região PERPHR 97
Tabela 5 Subdivisão da região SYS.STK 98
Tabela 6 Permissões para a região SYS.STK 98
Tabela 7 Permissões para a região SYS.COD 99
Tabela 8 Permissões para a região SYS.DAT 99
Tabela 9 Permissões para a região SYS.HP 100
Tabela 10 Permissões para a região SYS.FLTT 101
Tabela 11 Permissões para a região SYS.SLTT 101
Tabela 12 Permissões para a região MOD.COD 101
Tabela 13 Permissões para a região MOD.DAT 102
Tabela 14 Permissões para a região MOD.HMC.STK 102
Tabela 15 Permissões para a região PAR.COD 103
Tabela 16 Permissões para a região PAR.DAT 104
Tabela 17 Permissões para a região PAR.DAT.IMG 104
Tabela 18 Permissões para a região PAR.HP 105
Tabela 19 Permissões para a região PAR.DEF.STK 105
Tabela 20 Permissões para a região PAR.EH.STK 105
Tabela 21 Permissões para a região PAR.HMC.STK 106
Tabela 22 Permissões para a região PRO.STK 106
Tabela 23 Cenário do teste 001 154
Tabela 24 Estatísticas de código 160

LISTA DE ABREVIATURAS E SIGLAS

ADC *Analog-to-Digital Converter*

AEEC *Airlines Electronic Engineering Committee*

AFDX *Avionics Full Duplex Switched Ethernet*

APEX *APplication/EXecutive*

API *Application Programming Interface*

ARINC *Aeronautical Radio, Incorporated*

ARM® *Advanced RISC Machines*

ARMv7 *ARM®* versão 7

ASID *Address Space Identifier*

CAN Controller Area Network

DDR2 *Double Data Rate* tipo 2

DDR3 *Double Data Rate* tipo 3

DMA *Direct Memory Access*

ELF *Executable and Linkable Format*

FIFO *First In First Out*

FIQ *Fast Interrupt Request*

FTDI *Future Technology Devices International*

GPIO *General Purpose Input/Output*

HM *Health Monitoring*

I^2C *Inter-Integrated Circuit*

IDE *Integrated Development Environment*

IP *Internet Protocol*

IRQ *Interrupt Request*

IEEE	*Institute of Electrical and Electronics Engineers*
LCD	*Liquid Crystal Display*
LED	*Light-Emitting Diode*
MAC	*Media Access Control*
mDDR	*Mobile DDR*
MMC	*Multimedia Card*
MMU	*Memory Management Unit*
MPU	*Memory Protection Unit*
OTG	*On-The-Go*
PA	*Physical Address*
PMSAv7	*Protected Memory System Architecture* versão 7
PRU	*Programmable Real-time Unit*
PWM	*Pulse-Width Modulation*
RAM	*Random Access Memory*
RNDIS	*Remote Network Driver Interface Specification*
ROM	*Read-Only Memory*
SD	*Secure Digital*
SDRAM	*Synchronous Dynamic RAM*
SIMD	*Single Instruction Multiple Data*
SO	Sistema Operacional
SOTR	Sistema Operacional de Tempo Real
STR	Sistema de Tempo Real
SPI	*Serial Peripheral Interface*
SRAM	*Static Random Access Memory*
TFTP	*Trivial File Transfer Protocol*

TI	*Texas Instruments*
TLB	*Translation Lookaside Buffer*
UART	*Universal Asynchronous Receiver/Transmitter*
UDP	*User Datagram Protocol*
USB	*Universal Serial Bus*
VA	*Virtual Address*
VFP	*Vector Floating-Point*
VFPv3	*Vector Floating-Point* versão 3
VLAN	*Virtual Local Area Network*
VMSAv7	*Virtual Memory System Architecture* versão 7
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO 23
1.1 TRABALHOS RELACIONADOS 24
1.2 ORGANIZAÇÃO DO TRABALHO 25
2 *ARINC 653* 27
2.1 ELEMENTOS DO SISTEMA 28
2.2 CONFIGURAÇÃO 29
2.3 SERVIÇOS 32
2.3.1 Gerenciamento de partições 33
2.3.2 Gerenciamento de processos 35
2.3.3 Gerenciamento de tempo 39
2.3.4 Comunicação interpartição 40
2.3.4.1 Portas de amostragem 41
2.3.4.2 Portas de enfileiramento 42
2.3.5 Comunicação intrapartição 43
2.3.5.1 *Buffers* 44
2.3.5.2 *Blackboards* 45
2.3.5.3 Semáforos 46
2.3.5.4 Eventos 47
2.3.6 Monitoramento 48
2.3.6.1 Nível de processo 49
2.3.6.2 Nível de partição 50
2.3.6.3 Nível de módulo 51
2.3.6.4 Nível de sistema 51
2.4 CONSIDERAÇÕES FINAIS 51
3 PLATAFORMA DE *HARDWARE* 53
3.1 PROCESSADOR 54
3.1.1 Modos de execução 56
3.1.2 *Memory Management Unit* (*MMU*) 58
3.1.3 Coprocessador *Vector Floating-Point* (*VFP*) 63
3.1.4 *Timers* 64
3.2 MEMÓRIA 64
3.3 PERIFÉRICOS 65
3.4 INTERFACES DE COMUNICAÇÃO 66
3.5 FERRAMENTAS DE *SOFTWARE* 67
3.6 INICIALIZAÇÃO DA PLATAFORMA 68
3.7 CONSIDERAÇÕES FINAIS 69
4 IMPLEMENTAÇÃO 71

4.1 ORGANIZAÇÃO DO CÓDIGO FONTE 71
4.2 GERÊNCIA DE PROCESSADOR: FILA DE PRIORIDADES 76
4.3 GERÊNCIA DE PROCESSADOR: CONTEXTOS DE EXECUÇÃO 79
4.4 GERÊNCIA DE PROCESSADOR: ESCALONAMENTO DE TAREFAS 83
4.5 GERÊNCIA DE PROCESSADOR: ESTADOS DO SISTEMA 88
4.6 GERÊNCIA DE MEMÓRIA: ALOCAÇÃO DINÂMICA 89
4.7 GERÊNCIA DE MEMÓRIA: PROTEÇÃO DE MEMÓRIA 91
4.7.1 Regiões de memória 92
4.7.2 Mapeamento de permissões 94
4.7.2.1 Tabela de vetores de exceção (VECTBL) 96
4.7.2.2 Registradores de periféricos (PERPHR) 97
4.7.2.3 Pilha do sistema (SYS.STK) 97
4.7.2.4 Código do sistema (SYS.COD) 98
4.7.2.5 Dados do sistema (SYS.DAT) 99
4.7.2.6 Alocação dinâmica do sistema (SYS.HP) 100
4.7.2.7 Tabelas de tradução de primeiro nível (SYS.FLTT) 100
4.7.2.8 Tabelas de tradução de segundo nível (SYS.SLTT) 101
4.7.2.9 Código do módulo (MOD.COD) 101
4.7.2.10 Dados do módulo (MOD.DAT) 102
4.7.2.11 Pilha do *HM callback* do módulo (MOD.HMC.STK) 102
4.7.2.12 Código de partição (PAR.COD) 103
4.7.2.13 Dados de partição (PAR.DAT) 103
4.7.2.14 Imagem de dados de partição (PAR.DAT.IMG) 104
4.7.2.15 Alocação dinâmica de partição (PAR.HP) 104
4.7.2.16 Pilha do processo padrão de partição (PAR.DEF.STK) .. 105
4.7.2.17 Pilha do tratador de erros de partição (PAR.EH.STK) .. 105
4.7.2.18 Pilha do *HM callback* de partição (PAR.HMC.STK) 106
4.7.2.19 Pilha de processo (PRO.STK) 106
4.7.3 Atribuição de permissões para depuração 106
4.7.4 Atribuição de permissões para partições de sistema 107
4.7.5 Biblioteca da *MMU* 107
4.8 TRATAMENTO DE ERROS 109
4.8.1 Erros de memória 111
4.8.2 Perda de *deadline* 113
4.8.3 Erros numéricos 114
4.8.4 Tratamento de erros a nível de sistema 115

4.9 CONFIGURAÇÃO 115
4.9.1 Extensão do esquema de configuração 115
4.9.1.1 SysHM_Ext:Extension 116
4.9.1.2 Mod_HM_Ext:Extension 116
4.9.1.3 PortExt:Extension 116
4.9.1.4 Proc_Ext:Extension 117
4.9.1.5 PartitionExt:Extension 118
4.9.1.6 Part_HM_Ext:Extension 119
4.9.1.7 ModExt:Extension 119
4.9.1.8 ModExt:Extension_AM335X 119
4.9.2 Convenções 120
4.9.3 Tradução 121
4.9.3.1 Sistema 121
4.9.3.2 Módulo 122
4.9.3.3 Partição 124
4.9.3.4 Processo 125
4.10 PARTIÇÃO DE SISTEMA PARA ENTRADA/SAÍDA . . 126
4.11 CARACTERÍSTICAS E LIMITAÇÕES 129
4.12 COMENTÁRIOS GERAIS 131
4.13 CONSIDERAÇÕES FINAIS 138
5 FERRAMENTAS DE CONFIGURAÇÃO 139
5.1 GERENCIADOR DE MODELOS 139
5.2 PRÉ-PROCESSADOR DE CONFIGURAÇÃO 140
5.3 GERADOR DE MODELOS 141
5.4 GERENCIAMENTO DE MODELOS 142
5.5 CONSIDERAÇÕES FINAIS 143
6 CASOS DE TESTE 145
6.1 VISÃO GERAL 146
6.1.1 Escalonamento e execução 147
6.1.2 Serviços ARINC 653 básicos 147
6.1.3 Serviços internos 148
6.1.4 Serviços ARINC 653 de comunicação 149
6.1.5 Monitoramento 150
6.1.6 Especiais 150
6.2 EXECUÇÃO DOS TESTES 150
6.2.1 *Bootloader* 151
6.2.2 Ambiente 151
6.2.3 Procedimento 153
6.2.4 Resultados e cobertura 156
6.3 CONSIDERAÇÕES FINAIS 156
7 CONCLUSÃO 157

REFERÊNCIAS 161
APÊNDICE A – Exemplo de código de aplicação 167
APÊNDICE B – Exemplo de execução 173
APÊNDICE C – Implementação 181
APÊNDICE D – Casos de teste 219

1 INTRODUÇÃO

Define-se Sistemas de Tempo Real (STRs) como os sistemas computacionais que estão submetidos, além de a requisitos de natureza lógica, a requisitos de natureza temporal. Nesse tipo de sistema os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no intervalo de tempo correto. Aplicações com requisitos de tempo real têm se tornado cada vez mais comuns e variam muito em tamanho, complexidade e criticalidade; dentre elas encontram-se desde simples controladores embutidos em utilidades domésticas até os complexos e críticos sistemas relacionados à aviação, por exemplo (FARINES; FRAGA; OLIVEIRA, 2000).

Os STRs podem ser classificados de acordo com a criticalidade dos seus requisitos temporais. Nos STRs críticos ou *hard real-time*, o não atendimento a esses requisitos pode resultar em consequências catastróficas, tanto no sentido econômico quanto no sentido de poder causar a perda de vidas. Em STRs não críticos ou *soft real-time*, os requisitos temporais descrevem o comportamento desejado, porém o não atendimento a eles não invalida os resultados nem tem consequências catastróficas, apesar de reduzir a utilidade da aplicação (FARINES; FRAGA; OLIVEIRA, 2000).

Uma das classes de STRs críticos mais importantes é a das aplicações aviônicas, ou seja, das aplicações relacionadas aos sistemas eletrônicos empregados em aeronaves. A especificação ARINC 653 (ARINC, 2006a) descreve a interface funcional a ser oferecida ao *software* de aplicação e os aspectos temporais a serem atendidos por Sistemas Operacionais (SOs) sobre os quais são executadas aplicações dessa classe. Para operação nesse tipo de equipamento são utilizados SOs comerciais compatíveis com a ARINC 653 que, por necessitarem de certificação, apresentam custo financeiro bastante elevado (ROMANSKI, 2011); porém são também necessários SOs desse tipo para treinamento de engenheiros aeronáuticos especializados em aviônica e para experimentação de novas aplicações, propósitos para os quais a certificação não é necessária. Um dos pontos centrais da especificação ARINC 653 é a utilização de partições com isolamento temporal e espacial, permitindo a execução de múltiplas aplicações com diferentes objetivos e criticalidades na mesma plataforma de *hardware*, e ainda garantindo que eventuais falhas ocorridas em uma partição possam ser tratadas de forma a não afetar a execução das demais (ARINC, 2006a; PRISAZNUK, 2008; VANDERLEEST, 2010; HAN; JIN, 2013).

Este trabalho descreve a implementação de um SO compatível com a parte 1 da especificação ARINC 653, que define o conjunto mínimo de serviços por ela requeridos, sobre uma plataforma de *hardware* de baixo custo. Esse SO destina-se exclusivamente ao treinamento de pessoas envolvidas no desenvolvimento de sistemas empregados em aeronaves e à experimentação nesse mesmo contexto, e portanto não tem por objetivo a certificação para operação em voo.

1.1 TRABALHOS RELACIONADOS

Diversos Sistemas Operacionais de Tempo Real (SOTRs) de licença livre, como o **FreeRTOS** (REAL TIME ENGINEERS LTD., 2014), o **RTEMS** (OAR CORPORATION, 2014) e o **EPOS** (LISHA/UFSC, 2014), suportam vários dos recursos exigidos pela especificação ARINC 653, como por exemplo o escalonamento de processos por prioridade e semáforos. Portanto, a implementação realizada neste trabalho poderia ter sido feita com base em um desses SOs, através da alteração ou da adição de recursos conforme as definições da especificação, de forma semelhante ao proposto em (RUFINO; FILIPE, 2007). Porém, a maioria desses SOTRs não oferece suporte à execução particionada de processos e não possui quaisquer mecanismos de monitoramento semelhantes àqueles exigidos pela ARINC 653, de forma que a implementação desses recursos sobre eles exigiria grandes alterações infraestruturais. Alterações dessa magnitude podem provocar a descaracterização do *software* utilizado como base, e ainda gerar produtos pouco estáveis, dadas as restrições impostas pela infraestrutura reaproveitada. Por esses motivos, optou-se neste trabalho pela implementação de um SO projetado exclusivamente para atendimento à ARINC 653, e portanto sobre o qual existe total liberdade para elaboração dos mecanismos exigidos por ela, porém valendo-se de SOTRs de licença livre como fonte de informação.

Alguns trabalhos recentes abordam e avaliam a utilização de SOs tradicionais (geralmente Linux) como base para o oferecimento dos serviços exigidos pela ARINC 653, como (HAN; JIN, 2012) e (HAN; JIN, 2013), tanto a nível de usuário (utilizando chamadas do SO hospedeiro para gerenciamento de tarefas) quanto a nível de núcleo (adaptando o *kernel* do SO hospedeiro). Outros trabalhos propõem a implementação da ARINC 653 sobre uma plataforma de simulação, como (PASCOAL, 2008), permitindo assim o desenvolvimento e teste de aplicações aviônicas sem necessidade de acesso à plataforma de

hardware sobre a qual essas serão executadas. Essas abordagens eliminam desafios inerentes à utilização de sistemas particionados, por exemplo quanto à interação com elementos de *hardware*, que precisam ser adequadamente inicializados e gerenciados para que possam ser utilizados nesse tipo de ambiente. Dessa forma, o emprego dessas abordagens mostra-se pouco vantajoso com finalidades didáticas e, por esse motivo, elas não foram utilizadas neste trabalho.

Existem também trabalhos que abordam a utilização de recursos de virtualização para implementação de particionamento em SOs compatíveis com a ARINC 653, como (HAN; JIN, 2011), (HAN; JIN, 2013) e (VANDERLEEST, 2010), permitindo assim a execução de um diferente SO em cada partição. Porém, processadores utilizados em plataformas de *hardware* de baixo custo geralmente não oferecem esse tipo de recurso, e portanto seu emprego limitaria as plataformas sobre as quais o SO poderia ser executado. Por esse motivo optou-se pela implementação do SO sem o emprego de virtualização, maximizando assim a gama de plataformas às quais o SO poderá ser migrado futuramente.

Conforme citado anteriormente, existem diversos SOs compatíveis com a especificação ARINC 653, porém são *softwares* proprietários e apresentam alto custo financeiro por necessitarem de certificação para operação. Além disso, esses SOs são usualmente executados em plataformas de *hardware* cujo custo também é elevado, e portanto sua utilização para fins didáticos e experimentais é inviável na existência de restrições financeiras (ROMANSKI, 2011).

Dado esse panorama, justifica-se a realização deste trabalho pela inexistência de SOs compatíveis com a especificação ARINC 653 que possam ser utilizados para fins didáticos e experimentais e que apresentem baixo custo financeiro.

1.2 ORGANIZAÇÃO DO TRABALHO

Apresenta-se, a seguir, os assuntos abordados pelos próximos capítulos deste trabalho:

- No Capítulo 2 são apresentadas as principais definições da especificação ARINC 653;
- No Capítulo 3 são apresentadas as principais características da plataforma de *hardware* empregada neste trabalho;
- No Capítulo 4 são apresentadas as características técnicas mais relevantes da implementação do SO desenvolvido, sobre as quais são apresentados maiores detalhes no Apêndice C;

- No Capítulo 5 são apresentadas as ferramentas de configuração desenvolvidas para utilização em conjunto com o SO;
- No Capítulo 6 são apresentados os casos de teste elaborados para validação do SO desenvolvido, cujos detalhes são fornecidos no Apêndice D;
- No Capítulo 7 são apresentadas as principais conclusões obtidas neste trabalho;
- No Apêndice A é apresentado o código fonte de uma aplicação executada no SO desenvolvido;
- No Apêndice B é apresentado um exemplo de execução de uma aplicação sobre o SO desenvolvido.

2 *ARINC 653*

Fundada em 1929 nos Estados Unidos com o objetivo de servir às necessidades da indústria de transporte aéreo, a *Aeronautical Radio, Incorporated* (*ARINC*) é, atualmente, propriedade da *Rockwell Collins* e fornecedora de diversos serviços, como redes de comunicação, infraestrutura em terra e processamento de informações para aviação, transporte em trilhos e outros setores de infraestrutura crítica (ARINC, 2006a; ROCKWELL COLLINS, 2014).

A *Airlines Electronic Engineering Committee* (*AEEC*) é um comitê internacional de profissionais técnicos de aviação que lidera o desenvolvimento de padrões e soluções técnicas para equipamentos eletrônicos utilizados em aeronaves. Esse comitê estabelece padrões baseados em consenso que são publicados pela *ARINC*, cuja utilização por parte de empresas de aviação representa benefícios consideráveis por permitir a padronização e o intercâmbio de equipamentos aviônicos, além da redução de custos através da promoção da competitividade entre fornecedores (ARINC, 2006a; SAE ITC, 2014).

Dentre os padrões *ARINC*, é utilizado neste trabalho o ARINC 653, que especifica o ambiente de operação para o *software* de aplicação utilizado em equipamentos eletrônicos para aeronaves. Essa especificação tem como principal objetivo o estabelecimento de uma interface de propósito geral, denominada *APplication/EXecutive* (*APEX*), entre o SO e o *software* de aplicação executado em equipamentos aviônicos, definindo o comportamento desses serviços e os dados trocados de forma estática ou dinâmica (através de configuração ou de chamadas de serviço, respectivamente) no sistema. Os benefícios esperados da utilização dessa especificação são a portabilidade, a reusabilidade, a modularidade e a integração do *software* de aplicação. A especificação ARINC 653 é dividida em três partes (ARINC, 2006a):

Parte 1 Serviços requeridos – apresenta a *APEX* em seu formato mínimo, incluindo por exemplo serviços de gerenciamento de processos e de comunicação, e é utilizada extensivamente neste trabalho (ARINC, 2006a);

Parte 2 Serviços estendidos – apresenta serviços adicionais que podem ser agregados à *APEX*, incluindo por exemplo serviços de gerenciamento de arquivos, e não é abordada neste trabalho;

Parte 3 Especificação de testes de conformidade – apresenta o proce-

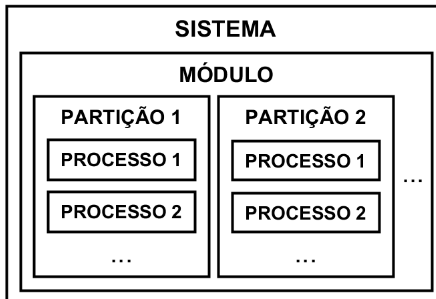
dimento de teste para demonstração e prova da adequação de um SO à ARINC 653 (ARINC, 2006b).

Apresenta-se, nas seções a seguir, a estrutura geral de sistemas que fazem uso de um SO compatível com a especificação ARINC 653, a forma como é realizada e mantida a configuração desses sistemas, assim como o conjunto mínimo de serviços que devem ser disponibilizados pelo SO para suporte às aplicações utilizadas nesse tipo de sistema.

2.1 ELEMENTOS DO SISTEMA

Os sistemas compatíveis com a especificação ARINC 653 são compostos de um módulo de execução, que será referenciado simplesmente como módulo, sobre o qual são escalonadas múltiplas partições que, do ponto de vista de isolamento, podem ser consideradas análogas aos processos de um SO tradicional. Nessas partições são executados processos, análogos às *threads* de SOs tradicionais desse mesmo ponto de vista, sendo esses utilizados para a implementação do *software* de aplicação (HAN; JIN, 2013). Um módulo está associado a um ou mais núcleos de processamento, mas uma partição, assim como todos seus processos, devem necessariamente ser executados sempre em um mesmo núcleo. Porém, deve ser possível a migração em tempo de projeto do *software* de aplicação entre diferentes módulos da aeronave, ou diferentes núcleos de processamento de um mesmo módulo, sem alterações na interface desse *software* com o SO (ARINC, 2006a). Ilustra-se, na Figura 1, a hierarquia de elementos de um sistema de acordo com a ARINC 653.

Figura 1 – Hierarquia de elementos de um sistema



A fim de assegurar a robustez para a execução concorrente e/ou

paralela de múltiplos *softwares* de aplicação em um único módulo, é empregada pela ARINC 653 uma abordagem de execução particionada dotada de isolamento temporal e espacial. Nessa abordagem, as partições são executadas em janelas de tempo alocadas dentro de um quadro cíclico denominado *major frame* (isolamento temporal) e as regiões de memória utilizadas por cada uma delas são protegidas contra acesso das demais (isolamento espacial). Além disso, as exceções ocorridas durante a execução de uma determinada partição são tratadas através de mecanismos que não violam o escalonamento, garantindo assim a não propagação de faltas entre partições (ARINC, 2006a; HAN; JIN, 2013).

Todo o *software* executado na plataforma de *hardware* sobre a qual é utilizado um SO compatível com a especificação ARINC 653 está enquadrado em uma das seguintes classes (ARINC, 2006a):

Partições de aplicação Partições utilizadas para execução de processos de aplicação, estando esses restritos à utilização da *Application Programming Interface* (*API*) compatível com a ARINC 653 fornecida pelo SO;

Partições de sistema Partições que utilizam recursos que vão além daqueles oferecidos pela *API* definida pela especificação, utilizadas, por exemplo, para o processamento de entrada e saída de dados, mas que estão sujeitas aos mesmos tipos de isolamento aplicados às partições de aplicação;

Núcleo do SO Fornece os serviços definidos na especificação e suporta o ambiente no qual o *software* de aplicação executa, incluindo, portanto, interfaces com a plataforma de *hardware*;

Funções específicas do sistema Compostas por interfaces com a plataforma de *hardware* como *drivers* de dispositivos ou *bootloaders*, por exemplo.

2.2 CONFIGURAÇÃO

A fim de garantir a máxima portabilidade e reusabilidade das aplicações utilizadas em sistemas aviônicos, é fundamental que as características da integração dessas ao ambiente no qual serão executadas sejam definidas e gerenciadas de forma que não sejam necessárias alterações em seu código quando reutilizadas. Para SOs compatíveis com a especificação ARINC 653, as principais características do módulo são definidas através de um arquivo de configuração no formato *Extensible Markup Language* (*XML*) cujo esquema básico é definido na própria

especificação. Entre as informações fornecidas através do arquivo de configuração destaca-se, por exemplo (ARINC, 2006a):

- A declaração das partições e processos do módulo;
- A definição da escala temporal das partições;
- O mapeamento de erros detectados pelo SO aos níveis (processo, partição ou módulo) nos quais devem ser tratados pelo mecanismo de monitoramento, em função do estado do sistema no qual ocorrem;
- A definição das ações a serem executadas para erros mapeados aos níveis de partição e de módulo, que são executadas automaticamente pelo SO quando de sua ocorrência;
- A configuração dos artefatos de comunicação utilizados pelas partições do módulo.

A definição e manutenção desses arquivos de configuração são tarefas do integrador do sistema, que pode ser uma das organizações envolvidas no projeto da aeronave ou uma combinação de algumas dessas organizações, e possui como principais responsabilidades (ARINC, 2006a):

Alocar partições a módulos De forma que todas as partições tenham suas necessidades temporais, de memória, de tratamento de erros e de comunicação, assim como seus requisitos de disponibilidade e de integridade, satisfeitos;

Gerenciar a comunicação entre partições Garantindo que todas as informações necessárias para a execução das partições do módulo sejam corretamente fornecidas, gerenciando, inclusive, as rotas de transmissão de mensagens entre partições e módulos e a compatibilidade dessas mensagens;

Gerenciar o tratamento de erros Definindo o comportamento dos mecanismos relacionados a essa tarefa para erros gerados a partir de todos os possíveis estados do sistema e levando em conta aspectos intimamente ligados à implementação da aplicação;

Realizar a verificação do sistema Com o objetivo de garantir que o sistema todo atende a seus requisitos funcionais, de disponibilidade e de integridade quando as aplicações trabalham de forma integrada, não incluindo, porém, a verificação das aplicações, que deve ser realizada pelo desenvolvedor de cada uma delas.

Os desenvolvedores das aplicações que serão executadas no módulo também têm participação fundamental no processo de

definição de suas configurações, uma vez que muitos dos requisitos que devem ser satisfeitos decorrem de características das aplicações. Os arquivos de configuração podem ainda precisar ser validados e/ou traduzidos através de ferramentas fornecidas juntamente com o SO no qual serão empregados para que possam ser efetivamente utilizados (ARINC, 2006a).

É também desejável que os arquivos de configuração de módulos ARINC 653 sejam independentes de implementação, de forma que um mesmo arquivo possa ser utilizado sem alterações, ou com o mínimo possível de alterações, em diferentes SOs e/ou sistemas. De qualquer maneira, em alguns casos a utilização de parâmetros específicos para determinado SO ou plataforma de *hardware* mostra-se necessária, motivo pelo qual o esquema definido pela ARINC 653 para esses arquivos possui pontos de expansão em todas as suas principais entidades (ARINC, 2006a).

A seguir são apresentados, de forma sucinta, os principais elementos do esquema básico dos arquivos de configuração de módulo definidos pela especificação ARINC 653, devendo os detalhes sobre esses elementos e sua utilização ser consultados no documento da especificação (ARINC, 2006a):

System_HM_Table Possui uma entrada **System_State_Entry** para cada estado do sistema, cada uma contendo uma entrada **Error_ID_Level** para cada erro que pode ser gerado no estado do sistema associado, sendo que cada uma dessas entradas mapeia um erro ocorrido em um determinado estado do sistema ao nível (processo, partição ou módulo) no qual esse deve ser tratado;

Module_HM_Table Possui uma entrada **System_State_Entry** para cada estado do sistema no qual existe algum erro mapeado para o nível de módulo, cada uma contendo uma entrada **Error_ID_Action** para cada erro mapeado a esse nível no estado associado, sendo que cada uma delas define a ação a ser executada pelo SO para o nível de módulo quando da ocorrência do erro em questão;

Partition Declara uma partição do módulo (podendo, portanto, ocorrer diversas vezes), e pode possuir entradas **Process**, **Sampling_Port** e **Queuing_Port**, que declaram os processos executados na partição e as portas de comunicação às quais a partição está conectada, respectivamente;

Partition_Memory Utilizada para declaração dos requisitos de memória de cada uma das partições do módulo, possui uma entrada **Memory_Requirements** para cada região de memória reservada à partição;

Module_Schedule Utilizada para definir a escala temporal das partições, indica o tamanho do *major frame* através do atributo **MajorFrameSeconds** e possui uma entrada **Partition_Schedule** para cada partição, cada uma contendo uma ou mais entradas **Window_Schedule** que são utilizadas para alocar janelas de tempo dentro do *major frame* para execução da partição em questão;

Partition_HM_Table Possui uma entrada **System_State_Entry** para cada estado do sistema no qual existe algum erro mapeado para o nível de partição, cada uma contendo uma entrada **Error_ID_Action** para cada erro mapeado a esse nível no estado associado, sendo que cada uma delas define a ação a ser executada pelo SO para o nível de partição quando da ocorrência do erro em questão;

Connection_Table Utilizada para declarar os canais de comunicação do módulo através de entradas **Channel**, sendo que cada uma dessas possui uma entrada **Source** e uma ou mais entradas **Destination**, que definem as partições de origem e de destino do canal e que podem ser partições do mesmo módulo (**Standard_Partition**) ou externas a ele (**Pseudo_Partition**).

2.3 SERVIÇOS

A especificação ARINC 653 define diversas categorias de serviços, que representam as funcionalidades mínimas que devem ser fornecidas pelo SO ao *software* de aplicação executado em uma partição (ARINC, 2006a). Apresenta-se a seguir essas categorias de serviços:

Gerenciamento de partições Fornece serviços relacionados à obtenção de informações e ao gerenciamento do estado da partição;

Gerenciamento de processos Define serviços para obtenção de informações e gerenciamento dos processos executados na partição;

Gerenciamento de tempo Oferece serviços para o gerenciamento do tempo alocado aos processos da partição;

Comunicação interpartição Disponibiliza serviços para comunicação entre duas ou mais partições do mesmo módulo ou de módulos diferentes (via rede);

Comunicação intrapartição Fornece serviços para comunicação e sincronização entre processos da partição;

Monitoramento (*Health Monitoring*) Define serviços para geração de históricos (*logs*) e para gerenciamento do mecanismo de tratamento de erros.

As seções a seguir detalham as características definidas pela especificação para cada uma dessas categorias, o comportamento do SO com relação aos elementos do sistema que são controlados por cada uma delas e, de forma sucinta, os serviços a elas relacionados que devem ser fornecidos pelo SO. Maiores informações devem ser obtidas na especificação ARINC 653 (ARINC, 2006a).

2.3.1 Gerenciamento de partições

Uma partição é uma unidade de *software* executada de forma particionada, ou seja, compartilhando recursos de *hardware* com outras partições. As partições de um módulo estão sujeitas a restrições temporais, não devendo portanto ser executada em momentos reservados às demais, e espaciais, tendo acesso somente às regiões de memória explicitamente reservadas para sua utilização (ARINC, 2006a).

A escala temporal das partições é criada em tempo de configuração através da definição de um quadro de tempo denominado *major frame*, que é repetido ciclicamente pelo SO, e da alocação de janelas de tempo para cada partição do módulo, que são intervalos de tempo no *major frame* durante os quais uma determinada partição será executada. Essa abordagem garante o determinismo no escalonamento de partições e, em consequência, o isolamento temporal delas. Partições são usualmente escalonadas de forma periódica, porém o mecanismo de configuração da escala permite a alocação de janelas de tempo que não correspondem ao período nominal a elas atribuído. Por isso é necessário que sejam indicadas na escala as janelas de tempo alocadas à partição que correspondem ao início de seu período, de forma que essa informação possa ser levada em conta no escalonamento de processos (apresentado na próxima seção). Assim como a escala temporal, são definidas em tempo de configuração as regiões de memória alocadas a cada uma das partições do módulo, devendo essas ser dimensionadas de acordo com as necessidades das aplicações nelas executadas. Essa abor-

dagem tem por objetivo garantir o isolamento espacial das partições, e para obtenção efetiva desse isolamento é necessário ainda que eventuais tentativas de acesso a regiões de memória às quais uma determinada partição não possui permissão sejam detectadas e tratadas de forma que o particionamento não seja violado (ARINC, 2006a).

Cada partição possui um modo de operação que varia ao longo de sua execução, que pode ser (ARINC, 2006a):

Ocioso (IDLE) A partição não é executada, de forma que as janelas de tempo alocadas a ela são consideradas ociosas;

Inicialização a frio (COLD_START) A partição é executada mas a inicialização de seus elementos está em progresso, e portanto o escalonador de processos não é executado;

Inicialização a quente (WARM_START) Semelhante a **COLD_START**, porém a transição para o modo de inicialização não foi a frio (a partir do sistema desligado), e portanto o contexto de *hardware* no qual a inicialização ocorre pode ser diferente – por exemplo, o *software* pode já estar residente em memória, não sendo necessário que seja carregado para que seja executado;

Normal (NORMAL) A inicialização da partição foi concluída e o escalonamento de processos está ativo.

No mínimo os seguintes serviços devem ser oferecidos por SOs compatíveis com a especificação ARINC 653 para o gerenciamento de partições (ARINC, 2006a):

GET_PARTITION_STATUS Obtém informações relacionadas ao estado da partição atual (a partir da qual o serviço é invocado);

SET_PARTITION_MODE Possibilita a alteração do modo de operação da partição atual, permitindo que essa entre em modo de operação normal após sua inicialização, que seja parada ou que seja reinicializada.

Durante a inicialização de uma partição, todos os recursos por ela utilizados a qualquer momento de sua execução, como memória e quaisquer mecanismos fornecidos pelo SO, devem ser alocados e/ou inicializados. Finalizada sua etapa de inicialização, a partição deve solicitar transição para o modo de operação **NORMAL** através do serviço **SET_PARTITION_MODE**, iniciando-se então o escalonamento de processos e não sendo mais permitida qualquer alocação de recursos, nem a liberação daqueles já alocados. Essa

abordagem tem por objetivo garantir máximo determinismo, já que uma vez em modo de operação **NORMAL** a partição possui reservados todos os recursos de que fará uso durante sua execução. Uma partição pode ainda ser reinicializada através do serviço **SET_PARTITION_MODE**, porém caso essa encontre-se em modo **COLD_START** então apenas a transição para esse mesmo modo poderá ser realizada, já que considera-se que o estado do sistema no modo de operação **WARM_START** é mais completo que no **COLD_START**. O modo de operação atual da partição e outras informações sobre seu estado, como por exemplo seu nível de bloqueio, podem ser consultadas através do serviço **GET_PARTITION_STATUS** (ARINC, 2006a).

2.3.2 Gerenciamento de processos

Um processo é uma unidade de *software* que é executada de forma concorrente com outros processos da partição na qual está contida, mas que não é visível – e portanto não pode ser controlada – a partir de outras partições. Os serviços que devem ser oferecidos pelo SO para realização do gerenciamento de processos da partição atual (a partir da qual os serviços são invocados) são (ARINC, 2006a):

GET_PROCESS_ID Obtém o identificador de um processo a partir de seu nome;

GET_PROCESS_STATUS Obtém informações sobre o estado de um processo a partir de seu identificador;

CREATE_PROCESS Cria um processo a partir de um conjunto de atributos, e fornece o identificador atribuído pelo SO ao processo criado;

SET_PRIORITY Altera a prioridade atual de um processo;

SUSPEND_SELF Utilizado por processos aperiódicos para solicitar a própria suspensão por um determinado período (ou indefinidamente);

SUSPEND Suspende a execução de um processo aperiódico;

RESUME Continua a execução de um processo aperiódico suspenso;

STOP_SELF Permite que um processo pare sua própria execução;

STOP Permite que um processo pare a execução de outro processo;

DELAYED_START Utilizado para que um processo possa iniciar a execução de outro processo com atraso;

START Equivalente à chamada de **DELAYED_START** com atraso zero;

LOCK_PREEMPTION Incrementa o contador de nível de bloqueio da partição atual, inibindo o escalonamento de processos;

UNLOCK_PREEMPTION Decrementa o contador de nível de bloqueio da partição, retomando o escalonamento de processos caso esse nível atinja zero;

GET_MY_ID Obtém o identificador do processo atual (a partir do qual o serviço é invocado).

Durante a etapa de inicialização de uma partição, o serviço **CREATE_PROCESS** pode ser utilizado para criação de múltiplos processos, cuja dinâmica combinada realiza a função – ou uma parte da função – projetada para uma determinada aplicação. Após criados, os processos devem ainda ser iniciados, durante a inicialização ou a execução normal da partição, através dos serviços **START** ou **DELAYED_START**. Cada processo é criado no máximo uma vez durante a execução de uma partição e não pode ser destruído; porém, pode ser parado por si mesmo ou por outros processos através dos serviços **STOP_SELF** e **STOP**, respectivamente, e reiniciado sempre que necessário (ARINC, 2006a).

Cada processo possui uma prioridade base definida em tempo de configuração, e uma prioridade atual que é utilizada no escalonamento de processos e recursos e que tem inicialmente o mesmo valor que a prioridade base, mas que pode ser alterada por qualquer processo da partição através do serviço **SET_PRIORITY**. O processo em execução é sempre aquele que encontra-se pronto e possui a maior prioridade atual, de forma que qualquer processo pode ser interrompido a qualquer momento por um processo de maior prioridade, podendo porém inibir essa interrupção em caso de necessidade através dos serviços **LOCK_PREEMPTION** e **UNLOCK_PREEMPTION**. Caso existam dois ou mais processos prontos de mesma prioridade atual e essa prioridade seja a mais alta dentre os processos da partição, o processo executado é o mais antigo, ou seja, aquele que tornou-se pronto para execução mais cedo em relação aos demais de mesma prioridade. Um processo pode ainda ser interrompido a qualquer momento pelo escalonador de partições, passando portanto a ser executada outra partição; porém, quando a partição à qual o processo pertence voltar a ser executada, o processo retomado deve necessariamente ser aquele anteriormente interrompido (ARINC, 2006a).

Os processos de uma partição podem ser periódicos, quando sua execução ocorre repetidamente a cada intervalo fixo de tempo (laços de controle, por exemplo), ou aperiódicos, quando sua execução ocorre em decorrência de eventos externos cujo momento de disparo não é previsível (o acionamento de teclas, por exemplo). Uma mesma partição pode conter processos de ambos os tipos executando de forma concorrente, sendo esses diferenciados através da utilização de um valor de período distinto para os aperiódicos (zero, por exemplo). Após a execução os processos periódicos devem utilizar o serviço **PERIODIC_WAIT** (apresentado na próxima seção) para aguardar por sua próxima liberação, que é automaticamente disparada pelo SO de acordo com o período atribuído ao processo. Quando escalonados, processos aperiódicos executam até que sejam interrompidos por um processo de mais alta prioridade ou suspensos através de uma chamada a **SUSPEND** por outro processo ou **SUSPEND_SELF** por si mesmos, tornando-se prontos novamente somente quando for realizada uma chamada a **RESUME** por outro processo ou quando o tempo de espera utilizado na chamada ao serviço **SUSPEND_SELF** expirar. Para processos periódicos, sua primeira liberação quando de sua inicialização ocorre em relação ao início do próximo período da partição à qual pertencem, conforme definido na escala de partições, e para processos aperiódicos a primeira liberação ocorre em relação ao horário no qual o processo foi iniciado (ARINC, 2006a).

O estado de um processo define sua situação em relação ao escalonamento de processos da partição à qual pertence, e pode ser um dos seguintes (ARINC, 2006a):

Dormente (DORMANT) O processo encontra-se inapto a ser executado (antes de ser iniciado e após ser parado);

Pronto (READY) O processo encontra-se pronto para ser executado, e será escalonado caso seja o de maior prioridade atual da partição;

Executando (RUNNING) O processo está atualmente sendo executado na partição – no máximo um processo encontra-se neste estado em cada partição a qualquer momento;

Esperando (WAITING) O processo encontra-se inapto a ser executado até que um determinado evento ocorra, sendo os possíveis motivos pelos quais esse estado pode ser alcançado os seguintes:

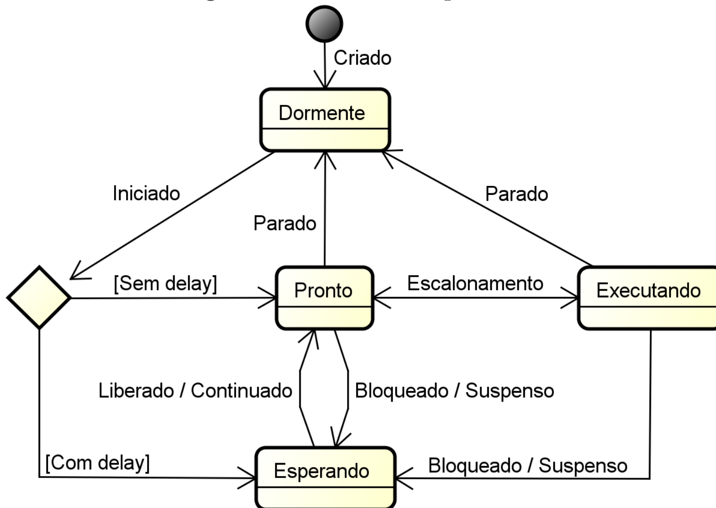
- O processo foi iniciado, porém a partição à qual pertence ainda não entrou em modo de operação **NORMAL**;

- O processo é aperiódico e foi suspenso por si mesmo ou por outro processo;
- O processo é periódico e aguarda sua próxima liberação;
- O processo encontra-se bloqueado por um recurso do SO (um semáforo ou um serviço de espera, por exemplo).

Quando um processo aperiódico bloqueado em uma chamada a um serviço de espera ou em outro recurso do SO (um semáforo, por exemplo) é suspenso por outro processo através da chamada de serviço **SUSPEND**, esse deve permanecer em estado de espera até que seja liberado por ambos os mecanismos, ou seja, só voltará ao estado **READY** quando for liberado tanto da suspensão quanto do bloqueio (ARINC, 2006a).

Apresenta-se, na Figura 2, os possíveis estados assumidos por processos de SOs compatíveis com a ARINC 653, assim como os principais eventos que disparam transições entre esses estados durante a execução do módulo.

Figura 2 – Estados de processos



2.3.3 Gerenciamento de tempo

Em STRs, especialmente naqueles que apresentam um certo nível de criticalidade, o tempo alocado à execução das diversas tarefas precisa ser gerenciado de forma que os recursos de processamento sejam utilizados de forma satisfatória, e que permita ainda o oferecimento de garantias quanto à escalonabilidade do sistema no pior caso. Os serviços que devem fundamentalmente ser oferecidos por SOs compatíveis com a especificação ARINC 653 para o gerenciamento de tempo são (ARINC, 2006a):

GET_TIME Solicita o valor atual do relógio do módulo;

TIMED_WAIT Solicita o bloqueio do processo atual por, no mínimo, um determinado período;

PERIODIC_WAIT Utilizado por processos periódicos para suspensão até o momento de sua próxima liberação;

REPLENISH Utilizado por processos para solicitar a postergação de seus *deadlines*.

O tempo do módulo é medido através de um relógio único, comum a todas suas partições e processadores, e que não é sincronizado com qualquer relógio externo. Valores relacionados a esse relógio são representados através do tipo de dados **SYSTEM_TIME_TYPE**, que consiste em um numérico inteiro de 64 *bits* com sinal, que indica durações de tempo em nanossegundos. O relógio do sistema, cujo valor é obtido através do serviço **GET_TIME**, indica o tempo decorrido entre o momento de sua inicialização e o momento atual, e todas as medidas de tempo são relativas a ele (ARINC, 2006a).

O serviço **TIMED_WAIT** é utilizado para inserção de atrasos de tempo relativos ao horário atual na execução de processos. O período de atraso solicitado através desse serviço representa o tempo mínimo durante o qual o processo permanecerá bloqueado, já que a escala de partições pode tornar o atraso efetivo mais longo. A utilização do serviço **TIMED_WAIT** com intervalo de tempo zero causa o reescalonamento de processos de mesma prioridade, ou seja, o processo atual é removido e reinserido na fila de processos prontos e, caso existam outros processos prontos de mesma prioridade, o processo atual deixará de ser o mais antigo entre eles e outro processo passará a ser executado (ARINC, 2006a).

Cada processo de uma partição possui uma capacidade de tempo associada, que pode ser finita ou infinita

(**INFINITE_TIME_VALUE**), e que é utilizada para limitar o tempo de processamento consumido pelo processo após sua liberação. Quando um processo é liberado seu *deadline* é atribuído para o horário atual do módulo somado à sua capacidade de tempo, e caso um *deadline* seja perdido o SO é responsável por disparar os mecanismos de tratamento de erros conforme a configuração do módulo. Esse evento pode ocorrer durante a execução da partição, caso no qual é tratado imediatamente, ou durante a execução de outra partição, devendo ser tratado assim que a partição à qual o processo pertence entrar novamente em execução. O *deadline* de um processo pode também ser postergado pelo uso do serviço **REPLENISH**, através do qual um processo solicita ao SO tempo adicional de execução, sendo o novo prazo calculado pela soma do horário atual ao tempo adicional solicitado pelo processo. Porém, não é permitido que processos periódicos posterguem seu *deadline* além do momento de sua próxima liberação (ARINC, 2006a).

2.3.4 Comunicação interpartição

Trata-se de uma classe de serviços que serve ao controle de mecanismos destinados à comunicação entre partições localizadas no mesmo módulo ou em módulos diferentes (via rede), podendo ainda suportar equipamentos externos ao módulo que não utilizam a especificação ARINC 653. Esse tipo de comunicação é realizada de forma que o *software* de aplicação não contenha informações sobre a localização de quaisquer das partições, assegurando assim a portabilidade das aplicações entre módulos sem alterações de código. Toda a comunicação é realizada através da troca de mensagens periódicas ou aperiódicas de tamanho fixo ou variável. Essas mensagens são transmitidas entre uma partição de origem e uma ou mais partições de destino de forma ordenada, íntegra e atômica, ou seja, as mensagens deixam a partição de origem e alcançam as partições de destino na mesma ordem, sem alterações (sem corrupção) e uma mensagem ou é transmitida integralmente ou é considerada perdida (ARINC, 2006a).

O mecanismo que conecta uma partição de origem a uma ou mais partições de destino é denominado **canal**, e define uma rota entre esses elementos, as características das mensagens transmitidas e o modo como essa transmissão é realizada. Canais são acessados por partições através da utilização de portas, que fornecem os recursos necessários para que essas possam enviar ou receber mensagens em um canal. Do

ponto de vista de uma partição, portas são utilizadas de forma que a rota utilizada por elas não afete a aplicação nela executada, ou seja, quaisquer mecanismos de fragmentação, sequenciamento, roteamento e garantia de integridade de mensagens que forem exigidos pela infraestrutura de rede empregada são transparentes para a aplicação (ARINC, 2006a).

Apresenta-se, nas seções a seguir, as principais características e os serviços associados aos mecanismos de comunicação interpartição exigidos pela especificação ARINC 653.

2.3.4.1 Portas de amostragem

Portas de amostragem (*sampling ports*) são utilizadas para a troca de mensagens periódicas nas quais são toleradas eventuais perdas (amostras de sensores, por exemplo). SOs compatíveis com a especificação ARINC 653 devem fornecer, no mínimo, os seguintes serviços para a utilização de portas de amostragem (ARINC, 2006a):

CREATE_SAMPLING_PORT Cria uma porta de amostragem vazia na partição atual;

WRITE_SAMPLING_MESSAGE Utilizado por processos da partição de origem de uma porta de amostragem para escrita de uma mensagem na porta;

READ_SAMPLING_MESSAGE Utilizado por processos das partições de destino de uma porta de amostragem para leitura da mensagem mais recentemente transmitida através da porta;

GET_SAMPLING_PORT_ID Obtém o identificador de uma porta de amostragem com base em seu nome;

GET_SAMPLING_PORT_STATUS Obtém informações sobre o estado de uma porta de amostragem com base em seu identificador.

Antes de ser utilizada, uma porta de amostragem precisa ser criada através do serviço **CREATE_SAMPLING_PORT**, tanto na partição de origem quanto nas partições de destino. Para isso é necessário o fornecimento de atributos da porta, como o tamanho máximo das mensagens trocadas através dela, a direção na qual a porta operará (de acordo com a partição na qual está sendo criada) e a taxa de atualização requerida para as amostras trocadas no canal. Uma mensagem escrita em uma porta de amostragem de origem, através do serviço

WRITE_SAMPLING_MESSAGE, é nela retida até que seja sobrescrita por outra mensagem, de forma que as partições de destino sempre leem, através do serviço **READ_SAMPLING_MESSAGE**, a mensagem mais recentemente escrita e transmitida no canal. Processos que solicitam leitura ou escrita em uma porta de amostragem nunca bloqueiam, já que a mensagem atual sempre pode ser lida ou sobrescrita, porém é fornecida uma propriedade de validade para as mensagens lidas, que indica se a taxa de atualização atribuída à porta na sua criação está ou não sendo satisfeita (ARINC, 2006a).

2.3.4.2 Portas de enfileiramento

Portas de enfileiramento (*queuing ports*) são utilizadas para a troca de mensagens sem tolerância a perdas, cuja transmissão é gerenciada através de filas (teclas pressionadas em um teclado, por exemplo). No mínimo os seguintes serviços devem ser fornecidos por SO compatíveis com a ARINC 653 para a utilização de portas de enfileiramento (ARINC, 2006a):

CREATE_QUEUING_PORT Cria uma porta de enfileiramento vazia na partição atual;

SEND_QUEUING_MESSAGE Utilizado por processos da partição de origem de uma porta de enfileiramento para envio de uma mensagem através da porta;

RECEIVE_QUEUING_MESSAGE Utilizado por processos das partições de destino de uma porta de enfileiramento para recepção de uma mensagem através da porta;

GET_QUEUING_PORT_ID Obtém o identificador de uma porta de enfileiramento com base em seu nome;

GET_QUEUING_PORT_STATUS Obtém informações sobre o estado de uma porta de enfileiramento com base em seu identificador.

Para portas de enfileiramento é exigido pela especificação apenas o suporte à transmissão de mensagens entre uma partição de origem e uma partição de destino. Assim como as portas de amostragem, antes que possam ser utilizadas as portas de enfileiramento precisam ser criadas através do serviço **CREATE_QUEUING_PORT**, tanto na partição de origem quanto nas partições de destino. Entre os atributos que devem ser fornecidos para sua criação estão o tamanho máximo das mensagens transmitidas, o número máximo de

mensagens enfileiradas, a direção na qual a porta operará na partição atual e a política de enfileiramento para os processos bloqueados. Nas portas de enfileiramento, as mensagens enviadas através do serviço **SEND_QUEUING_MESSAGE** não sobrescrevem as anteriores, ao invés disso elas são armazenadas em filas de política *First In First Out* (*FIFO*), tanto na partição de origem quanto nas de destino, para posterior transmissão no canal (no caso de filas de envio) ou entrega à aplicação através do serviço **RECEIVE_QUEUING_MESSAGE** (no caso de filas de recebimento) (ARINC, 2006a).

Caso a partição de origem de uma porta de enfileiramento envie mensagens mais rapidamente do que essas podem ser transmitidas, a fila de envio pode encher, causando o bloqueio do processo remetente pelo tempo máximo informado na chamada do serviço de envio ou até que uma entrada da fila torne-se livre. De forma semelhante, caso a partição de destino tente receber uma mensagem quando a fila de recebimento da porta está vazia o processo receptor será bloqueado, sendo liberado quando o tempo máximo de espera expirar ou quando uma mensagem tornar-se disponível na fila de recebimento. Os processos bloqueados nessas situações podem ser liberados em política *FIFO* ou por prioridade (sendo processos de mesma prioridade sempre atendidos em política *FIFO*), de acordo com a configuração informada na criação da porta de enfileiramento. É ainda possível que a partição de destino trate mensagens em ritmo menor do que elas são enviadas e transmitidas, de forma que a fila de recebimento pode encher, causando *overflow* (perda de mensagens) na porta de destino. Nesse caso, essa ocorrência é notificada à partição receptora através do serviço de recebimento, indicando que uma ou mais mensagens foram perdidas desde a última leitura da fila, devendo porém quaisquer mecanismos de retransmissão ser implementados pela aplicação caso sejam necessários (ARINC, 2006a).

2.3.5 Comunicação intrapartição

Fornece serviços relacionados a mecanismos de comunicação e sincronização para processos de uma mesma partição, que estão sujeitos a *overheads* muito menores que aqueles encontrados nos mecanismos de comunicação interpartição por não empregarem recursos de rede. Assim como nas portas de enfileiramento, processos bloqueados por esses mecanismos podem ser atendidos em política *FIFO* ou por prioridade, de acordo com a configuração informada em sua criação

(ARINC, 2006a).

Apresenta-se, nas seções a seguir, as principais características e os serviços associados aos mecanismos de comunicação intrapartição exigidos pela especificação ARINC 653.

2.3.5.1 *Buffers*

Buffers são elementos utilizados para a troca de mensagens de tamanho variável (porém limitado) entre processos, com armazenamento temporário em uma fila de política *FIFO*, de forma similar a portas de enfileiramento. Os seguintes serviços devem ser oferecidos para a utilização de *buffers* (ARINC, 2006a):

CREATE_BUFFER Cria um *buffer* vazio na partição atual;

SEND_BUFFER Envia uma mensagem em um *buffer*;

RECEIVE_BUFFER Recebe uma mensagem armazenada em um *buffer*;

GET_BUFFER_ID Obtém o identificador de um *buffer* com base em seu nome;

GET_BUFFER_STATUS Obtém informações sobre o estado de um *buffer* com base em seu identificador.

Antes de serem utilizados os *buffers* devem ser criados através do serviço **CREATE_BUFFER**, devendo para isso ser fornecidas informações como o tamanho máximo das mensagens transmitidas, o número máximo de mensagens armazenadas na fila e a política de enfileiramento de processos empregada. Caso um processo solicite o recebimento de uma mensagem através do serviço **RECEIVE_BUFFER** em um *buffer* vazio ou o envio através do serviço **SEND_BUFFER** em um *buffer* cheio, esse será bloqueado pelo tempo máximo definido na chamada do serviço ou até que a execução da operação seja realizada com sucesso, ou seja, quando uma mensagem estiver disponível para recebimento ou uma posição da fila estiver disponível para envio. Caso uma mensagem seja enviada em um *buffer* vazio no qual um processo encontra-se bloqueado para leitura, o serviço de envio copia a mensagem enviada não para a fila, mas para o endereço indicado no serviço de recebimento bloqueado, liberando o processo receptor. De forma semelhante, caso uma mensagem seja recebida em um *buffer* cheio no qual um processo encontra-se bloqueado para envio, após remover a mensagem mais antiga o serviço de recebimento insere na fila a mensagem

apontada pelo serviço de envio, liberando o processo emissor (ARINC, 2006a).

2.3.5.2 *Blackboards*

Blackboards permitem a troca de mensagens entre processos de forma similar a portas de amostragem, diferindo portanto dos *buffers* por não permitirem o enfileiramento de mensagens. No mínimo os seguintes serviços devem ser fornecidos para o suporte à utilização de *blackboards* por SOs compatíveis com a ARINC 653 (ARINC, 2006a):

CREATE_BLACKBOARD Cria uma *blackboard* vazia na partição atual;

DISPLAY_BLACKBOARD Exibe (escreve) uma mensagem em uma *blackboard*;

READ_BLACKBOARD Lê a mensagem atualmente exibida em uma *blackboard*;

CLEAR_BLACKBOARD Limpa a mensagem de uma *blackboard*;

GET_BLACKBOARD_ID Obtém o identificador de uma *blackboard* com base em seu nome;

GET_BLACKBOARD_STATUS Obtém informações sobre o estado de uma *blackboard* com base em seu identificador.

Blackboards devem ser criadas através do serviço **CREATE_BLACKBOARD**, devendo para isso ser fornecido o tamanho máximo das mensagens que serão trocadas através delas. Uma mensagem escrita em uma *blackboard* através do serviço **DISPLAY_BLACKBOARD** é retida até que seja limpa através do serviço **CLEAR_BLACKBOARD**, tornando-se vazia, ou até que seja sobrescrita por outra mensagem, de forma que a mensagem lida através do serviço **READ_BLACKBOARD** é sempre a mensagem mais recentemente escrita. Processos que escrevem em uma *blackboard* ou a limpam nunca são bloqueados, já que a mensagem atual pode sempre ser alterada, mas aqueles que realizam sua leitura podem ser bloqueados caso essa encontre-se vazia, até que uma mensagem seja escrita ou pelo tempo máximo definido na chamada do serviço de leitura. Quando processos que encontram-se bloqueados para leitura de uma *blackboard* vazia são liberados pela escrita de uma mensagem, esses acessam sempre a mensagem mais recentemente escrita. É possível que a *blackboard* seja limpa ou novamente escrita antes que

esses processos efetuem a leitura da mensagem cuja escrita causou sua liberação, devendo mesmo nessa condição a mensagem mais recentemente escrita ser por eles acessada (ARINC, 2006a).

2.3.5.3 Semáforos

Semáforos são mecanismos de sincronização de processos geralmente empregados para controle de acesso a recursos que estão disponíveis em número limitado ou que devem ser utilizados de forma mutuamente exclusiva. SOs compatíveis com a especificação ARINC 653 devem fornecer no mínimo os seguintes serviços para a manipulação de semáforos (ARINC, 2006a):

CREATE_SEMAPHORE Cria um semáforo na partição atual;

WAIT_SEMAPHORE Adquire (aguarda) um recurso de um semáforo;

SIGNAL_SEMAPHORE Libera um recurso de um semáforo;

GET_SEMAPHORE_ID Obtém o identificador de um semáforo com base em seu nome;

GET_SEMAPHORE_STATUS Obtém informações sobre o estado de um semáforo com base em seu identificador.

Semáforos precisam ser criados através do serviço **CREATE_SEMAPHORE** antes que possam ser utilizados, sendo para tanto necessário o fornecimento de atributos como a política de enfileiramento de processos (por prioridade ou *FIFO*), o valor máximo do semáforo – que define qual o número total de recursos que podem ser alocados –, e seu valor atual – que indica o número de recursos que encontram-se inicialmente disponíveis. O valor atual do semáforo é decrementado quando da aquisição de um recurso através do serviço **WAIT_SEMAPHORE** e incrementado quando de sua liberação por uma chamada a **SIGNAL_SEMAPHORE**. Quando um processo solicita a aquisição de um recurso, se o valor do semáforo é maior que zero esse continua sua execução, caso contrário o processo é bloqueado pelo tempo máximo definido na chamada do serviço ou até que um recurso seja liberado por outro processo (ARINC, 2006a).

2.3.5.4 Eventos

Eventos são mecanismos de sincronização que permitem que os processos de uma partição notifiquem a ocorrência de determinadas condições aos demais, que podem aguardar em bloqueio por essa ocorrência a fim de executar procedimentos de tratamento. No mínimo os seguintes serviços devem ser fornecidos por SOs compatíveis com a especificação ARINC 653 para a utilização de eventos (ARINC, 2006a):

CREATE_EVENT Cria um evento em estado *down* na partição atual;

SET_EVENT Atribui o estado de um evento para *up*, indicando que o evento está ocorrendo a partir do momento atual;

RESET_EVENT Atribui o estado de um evento para *down*, indicando que o evento não está ocorrendo a partir do momento atual;

WAIT_EVENT Aguarda em bloqueio pela ocorrência de um evento;

GET_EVENT_ID Obtém o identificador de um evento com base em seu nome;

GET_EVENT_STATUS Obtém informações sobre o estado de um evento com base em seu identificador.

Eventos devem ser criados através do serviço **CREATE_EVENT** durante a inicialização da partição na qual serão utilizados, e podem estar em estado *up*, indicando que o evento ocorre, ou *down*, indicando que o evento não ocorre no momento atual. Quando um processo solicita a espera por um evento que encontra-se em estado *up* através do serviço **WAIT_EVENT**, sua execução não é interrompida. Caso essa solicitação seja realizada para um evento que encontra-se em estado *down*, o processo é bloqueado durante o tempo máximo indicado na chamada de espera ou até que o evento tenha seu estado alterado para *up* através do serviço **SET_EVENT**, quando todos os processos que aguardam pelo evento são liberados simultaneamente. O estado do evento pode ainda ser restaurado para *down* através do serviço **RESET_EVENT**, causando o bloqueio de todos os processos que solicitarem, futuramente, o aguardo pelo evento (ARINC, 2006a).

2.3.6 Monitoramento

Em SOs compatíveis com a especificação ARINC 653, a função de monitoramento – *Health Monitoring* (*HM*) – é responsável pelo reporte e tratamento de erros através de mecanismos que não violam o isolamento de partições, ou seja, que permitem o controle da propagação de erros de forma a garantir sua contenção na partição cuja execução causou a falta. Erros podem ser gerados em decorrência de condições detectadas por mecanismos do processador (acessos inválidos à memória, por exemplo), pelo núcleo do SO (perdas de *deadline*, por exemplo) ou ainda podem ser lançados pela aplicação através de chamadas de serviço. Conforme citado anteriormente, os diversos tipos de erro detectados por esses mecanismos são mapeados através de uma tabela de configuração ao nível no qual serão tratados (de processo, de partição ou de módulo), em função dos estados do sistema a partir dos quais podem ser gerados. Existe ainda um nível máximo, o nível de sistema, ao qual erros não podem ser mapeados através de configuração, mas no qual são tratadas faltas geradas durante a execução de rotinas de tratamento a nível de módulo, por exemplo (ARINC, 2006a).

Cada partição pode possuir, além de seus processos de aplicação, um processo tratador de erros. Esse processo é disparado com prioridade máxima quando da ocorrência de erros mapeados ao nível de processo, que podem assim ser tratados pelo desenvolvedor da aplicação. Processos tratadores de erros não podem invocar serviços bloqueantes nem ser interrompidos, suspensos ou ter a prioridade modificada por outros processos, e podem ser parados apenas por si mesmos através do serviço **STOP_SELF**. Como possíveis ações que podem ser tomadas pelo processo tratador de erros destaca-se a parada ou a reinicialização de processos ou da partição, não podendo porém ser utilizado para correção de erros (através da alteração dos valores envolvidos em um cálculo, por exemplo) (ARINC, 2006a).

Nas tabelas de configuração associadas aos níveis de partição e de módulo podem ser definidos métodos opcionais, denominados *HM callbacks*, que são disparados pelo SO para tratamento dos erros mapeados a esses níveis. Nessas tabelas são também determinadas as ações de recuperação que devem ser executadas pelo SO após a conclusão da execução desses métodos, levando em conta o erro detectado e o estado do sistema no qual ele ocorreu. Entre as possíveis ações de recuperação estão a parada do elemento afetado, a reinicialização desse elemento ou ainda o erro pode ser ignorado, devendo ser tratado pelo *HM callback* do nível ao qual foi mapeado. Como possíveis ações executadas pe-

los *HM callbacks* pode-se citar o reporte do erro detectado e a parada ou reinicialização do(s) elemento(s) afetado(s), por exemplo (ARINC, 2006a).

É ainda possível o emprego de partições de sistema para o tratamento de erros, sendo que as aplicações localizadas em partições de aplicação podem reportá-los através dos mecanismos de comunicação interpartição apresentados. A principal vantagem dessa abordagem está no fato de que o tratamento dos erros ocorre de forma isolada em relação à partição na qual foram gerados, permitindo portanto a utilização de toda a infraestrutura oferecida pelo SO para desempenho dessa tarefa (ARINC, 2006a).

O conjunto mínimo de serviços que devem ser oferecidos por SOs compatíveis com a ARINC 653 para gerenciamento da função de monitoramento são (ARINC, 2006a):

REPORT_APPLICATION_MESSAGE Permite a transmissão de uma mensagem para geração de relatórios (*logs*);

CREATE_ERROR_HANDLER Utilizado para criação do processo tratador de erros da partição atual (a partir da qual o serviço é invocado);

GET_ERROR_STATUS Deve ser utilizado pelo processo tratador de erros para enumeração dos erros que causaram seu disparo, permitindo a execução de ações específicas para cada um deles;

RAISE_APPLICATION_ERROR Permite o lançamento de um erro de aplicação, disparando seu tratamento de acordo com a configuração do módulo.

Apresenta-se nas seções a seguir os tipos de erro compreendidos em cada um dos níveis de tratamento apresentados, assim como as características definidas pela especificação para os mecanismos de tratamento empregados em cada um desses níveis.

2.3.6.1 Nível de processo

Compreende erros que impactam um ou mais processos de uma partição, ou toda uma partição. Pode-se citar como exemplo os erros lançados através de chamadas ao serviço **RAISE_APPLICATION_ERROR**, erros numéricos (divisão por zero, por exemplo) e erros de acesso à memória (estouro de pilha, por exemplo), todos ocorridos durante a execução de processos. Na ocorrência de erros mapeados a este nível, o processo tratador de

erros da partição na qual o erro ocorreu é iniciado, caso exista, ou, em caso contrário, o erro é tratado no nível de partição. A execução de processos tratadores de erros é realizada de forma a não violar o particionamento, ou seja, ocorre exclusivamente durante as janelas de tempo alocadas à partição a partir da qual o erro foi gerado, e todas as demais partições continuam executando normalmente. Entre as ações que podem ser tomadas para tratamento de erros a nível de processo estão, por exemplo, a parada ou reinicialização do processo no qual o erro se originou, a inicialização de outros processos, ou ainda a parada ou reinicialização da partição afetada. A fim de manter a portabilidade do código utilizado para o tratamento de erros no nível de processo, esses são agrupados através dos seguintes códigos de erro (ARINC, 2006a):

DEADLINE_MISSED Perdas de *deadline* de processos;

APPLICATION_ERROR Erros lançados através do serviço **RAISE_APPLICATION_ERROR**;

NUMERIC_ERROR Erros originados em operações numéricas;

ILLEGAL_REQUEST Chamadas ilegais a serviços do SO;

STACK_OVERFLOW Tentativa de acesso além dos limites de regiões de memória utilizadas como pilha;

MEMORY_VIOLATION Tentativas de acesso a endereços de memória não permitidos;

HARDWARE_FAULT Erros originados na plataforma de *hardware*;

POWER_FAIL Erros decorrentes de falhas no suprimento de energia.

2.3.6.2 Nível de partição

São erros que têm impacto sobre apenas uma partição do módulo, por exemplo aqueles que ocorrem durante a inicialização de uma partição ou durante a execução de seu processo tratador de erros. Quando um erro deve ser tratado nesse nível, caso exista um *HM callback* para a partição esse é disparado pelo SO e, quando sua execução é concluída ou caso esse não exista, é executada a ação especificada na tabela de configuração correspondente. Essa ação pode ser ignorar o erro, devendo esse ser tratado no *HM callback* da partição, parar a partição afetada ou reiniciá-la. Assim como para os erros mapeados ao nível de processo, os mecanismos de tratamento de erros a nível de partição não violam o particionamento (ARINC, 2006a).

2.3.6.3 Nível de módulo

Compreende erros que impactam todas as partições do sistema, por exemplo aqueles ocorridos durante a inicialização do módulo, erros de configuração ou falhas no suprimento de energia. O processo de tratamento de erros a nível de módulo é análogo ao do nível de partição, porém no nível de módulo o particionamento é violado, ou seja, o *HM callback* do módulo é executado de forma exclusiva, ignorando-se a escala de partições; a ação tomada ao fim de sua execução pode ser ignorar o erro, devendo esse ser tratado no *HM callback* do módulo, parar o módulo ou reiniciá-lo (ARINC, 2006a).

2.3.6.4 Nível de sistema

São erros que impactam o sistema inteiro e para os quais a especificação não define os mecanismos de tratamento a serem oferecidos pelo SO, já que esses podem variar muito em função da plataforma de *hardware* utilizada, ficando portanto a cargo do desenvolvedor a elaboração dos recursos adequados a esse fim (ARINC, 2006a).

2.4 CONSIDERAÇÕES FINAIS

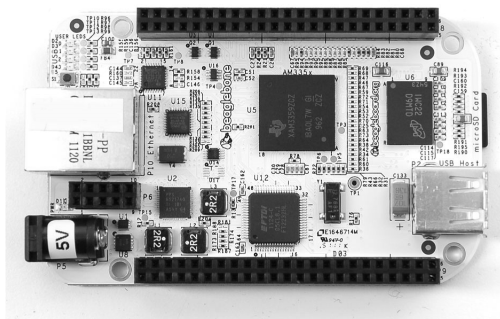
Neste capítulo foi apresentada uma introdução às principais características exigidas pela especificação ARINC 653 para SOs empregados em aeronaves, tendo como principal objetivo fornecer uma base acessível para a compreensão de sua estrutura e dos seus principais elementos. Maiores informações sobre as definições da ARINC 653, incluindo protótipos da interface *APEX* em diversas linguagens de programação, uma implementação de referência de todos os serviços oferecidos, o esquema dos arquivos de configuração de módulo e um exemplo desse tipo de arquivo devem ser consultadas no documento da especificação (ARINC, 2006a).

No próximo capítulo será apresentada a plataforma de *hardware* sobre a qual foi realizada a implementação do SO desenvolvido neste trabalho.

3 PLATAFORMA DE *HARDWARE*

O *hardware* sobre o qual é executado um SO compatível com a especificação ARINC 653 em aeronaves deve oferecer recursos de processamento e de temporização determinísticos, que permitam a implementação do isolamento temporal de partições e, ainda, tornem factível o oferecimento de garantias quanto à escalonabilidade do sistema no pior caso. Além disso, essas plataformas devem oferecer mecanismos de proteção de memória que viabilizem o isolamento espacial das partições do módulo, assim como o tratamento das faltas detectadas por esses mecanismos de acordo com os requisitos impostos pela especificação (ARINC, 2006a). Entretanto, dados os objetivos secundários deste trabalho relacionados à utilização de uma plataforma de *hardware* de baixo custo financeiro e aos fins exclusivamente didáticos e experimentais almejados, aliados ao fato de que as plataformas que atendem a esses requisitos com a robustez exigida para aplicações críticas – sendo portanto as mais adequadas à utilização em sistemas aviônicos – apresentam alto custo financeiro, algumas dessas restrições foram relaxadas na escolha da plataforma empregada neste trabalho.

Figura 3 – Plataforma *BeagleBone*



Neste capítulo será apresentada a plataforma de *hardware* mostrada na Figura 3, comercialmente conhecida como *BeagleBone*, sobre a qual o SO desenvolvido neste trabalho é executado. Essa plataforma consiste em um microcomputador de *hardware* aberto, ou seja, cuja documentação encontra-se disponível publicamente sem custo financeiro (BEAGLEBOARD.ORG, 2014), e que é equipado com, entre outros, os seguintes recursos (TI, 2011; COLEY, 2012):

- Processamento:
  - Microprocessador *Texas Instruments* (*TI*) AM335X (AM3358/AM3359);
  - Núcleo *Advanced RISC Machines* (*ARM*®) Cortex™-A8;
  - 2 *Programmable Real-time Units* (*PRUs*).
- Memória:
  - *Synchronous Dynamic RAM* (*SDRAM*) *DDR2* de 256MB;
  - Cartão *Secure Digital* (*SD*).
- Conectividade:
  - *Universal Serial Bus* (*USB*):
    * Alimentação;
    * *USB device*;
    * *USB host*;
    * Comunicação (porta *serial* virtual);
    * Depuração (*TI* XDS100v2 na placa).
  - Porta *Institute of Electrical and Electronics Engineers* (*IEEE*) 802.3 (*Ethernet*);
  - 92 pinos externos, fornecendo recursos como *General Purpose Input/Output* (*GPIO*), *Analog-to-Digital Converter* (*ADC*), *Pulse-Width Modulation* (*PWM*), *Inter-Integrated Circuit* (I^2C), *Serial Peripheral Interface* (*SPI*) e alimentação, através dos quais pode ser interconectada a placas de expansão.

Nas próximas seções deste capítulo serão apresentadas as principais características dessa plataforma de *hardware*, destacando suas potencialidades e limitações e avaliando a compatibilidade de suas características ao tipo de sistema tratado neste trabalho.

3.1 PROCESSADOR

O microprocessador *TI* AM335X, que será referenciado simplesmente como **processador**, possui um núcleo *ARM*® baseado na arquitetura *ARM*® versão 7 (*ARMv7*). Essa arquitetura possui três diferentes perfis: o **ARMv7-A** para processadores de aplicação, o **ARMv7-R** para processadores de tempo real e o **ARMv7-M** para microcontroladores. Ela ainda define duas diferentes arquiteturas de

gerenciamento de memória: a *Protected Memory System Architecture* versão 7 (*PMSAv7*) – empregada nos perfis **ARMv7-R** e, com algumas limitações, **ARMv7-M** – que oferece uma *Memory Protection Unit* (*MPU*) apenas para proteção de memória, e a *Virtual Memory System Architecture* versão 7 (*VMSAv7*) – utilizada pelo perfil **ARMv7-A** – que possui uma *Memory Management Unit* (*MMU*) com recursos para proteção e virtualização de memória. O processador *TI* AM335X possui um núcleo *ARM*® Cortex™-A8 baseado no perfil **ARMv7-A**, e portanto suporta apenas a arquitetura de gerenciamento de memória *VMSAv7* (ARM, 2012). Esse processador é equipado ainda com, entre outros, os seguintes recursos (ARM, 2014), (TI, 2011):

- *Pipeline* superescalar de 13 estágios *dual-issue* com execução em ordem;
- *Static Random Access Memory* (*SRAM*) de 64KB interna ao núcleo;
- *SRAM* de 64KB interna ao processador;
- Cache L1 de instruções e de dados de 32KB com linhas de 16 palavras e interface de 128 *bits*;
- Cache L2 de 256KB com linhas de 16 palavras e interface com o cache L1 de 128 *bits*;
- Coprocessador *VFP* baseado na arquitetura *ARM*® *Vector Floating-Point* versão 3 (*VFPv3*), compatível com o padrão *IEEE* 754 de representação e operação de números binários com ponto flutuante;
- *Memory Management Unit* (*MMU*) baseada na arquitetura *ARM*® *VMSAv7*, com suporte a páginas de 4KB, 64KB, 1MB e 16MB e *Translation Lookaside Buffers* (*TLBs*) separadas para dados e instruções de 32 entradas cada;
- Controlador de interrupções com até 128 diferentes *Interrupt Requests* (*IRQs*);
- 8 *timers* de modo *dual*.

É fundamental observar que processadores baseados nos perfis **ARMv7-R** ou **ARMv7-M** da arquitetura *ARMv7* são mais adequados para operação nos sistemas críticos aos quais a ARINC 653 destina-se, se comparados a processadores baseados no perfil **ARMv7-A** dessa mesma arquitetura. Isso porque a *MPU* da *PMSAv7* emprega registradores para definição dos mapeamentos de memória, enquanto a *MMU* da *VMSAv7* utiliza tabelas de tradução armazenadas em memória e *TLBs* para esse mesmo fim, de forma que

a *VMSAv7* apresenta menor determinismo temporal se comparada à *PMSAv7*. Além disso, destaca-se que o processador empregado neste trabalho não é voltado a aplicações críticas, e possui recursos temporalmente não determinísticos, como *branch prediction* e memória *cache*, que comprometem sua previsibilidade temporal. Essas características tornam esse processador pouco adequado para a execução de aplicações críticas de tempo real, já que inviabilizam o oferecimento de garantias quanto à escalonabilidade do sistema no pior caso, porém não desqualificam-no para fins didáticos e experimentais nesse mesmo contexto (ARM, 2012), (ARM, 2014), (WILHELM et al., 2008).

Nas próximas seções são apresentadas características e recursos do processador empregado, cujo conhecimento é fundamental para a compreensão do funcionamento do SO desenvolvido. Informações completas sobre as características do núcleo de processamento devem ser consultadas no manual de referência da arquitetura *ARMv7* (ARM, 2012) e no manual de referência técnica da implementação Cortex™-A8 (ARM, 2010), e informações sobre os demais recursos do processador podem ser obtidas no manual de referência técnica da linha de processadores *TI* AM335X (TI, 2011).

3.1.1 Modos de execução

A execução de *software* em diferentes modos e com níveis de privilégio distintos permite a restrição da utilização de algumas instruções (de manipulação de configurações do processador ou de entrada e saída de dados, por exemplo), assim como a aplicação de controles de acesso à memória de forma diferenciada para *software* de usuário, que são executadas em modos não privilegiados, e para o núcleo do SO, executado com privilégios (ARPACI-DUSSEAU, 2012). O núcleo *ARM*® Cortex™-A8 do processador utilizado neste trabalho suporta dois diferentes níveis de privilégio (ARM, 2012):

Sem privilégios (PL0) Não é permitido o acesso a alguns recursos da arquitetura, em particular à maioria das instruções que alteram configurações do processador e às instruções que causam a transição a outros modos de execução (apresentados a seguir), sendo a mudança de modo de execução possível apenas através da geração de exceções;

Privilegiado (PL1) Permite o acesso a todos os recursos da arquitetura, inclusive a instruções de alteração de configurações do pro-

cessador – como a desabilitação de interrupções – e de transição a outros modos de execução.

O modo de execução do núcleo de processamento *ARM*® Cortex™-A8 define, além do nível de privilégio no qual o *software* executa, o conjunto dos registradores que podem ser acessados. Alguns registradores, como os apontadores de pilha e de endereço de retorno, possuem réplicas em cada modo de execução suportado, de forma que cada um desses modos possui um valor distinto para eles; o registrador que armazena o estado de execução do processador, por sua vez, é salvo em uma réplica pertencente ao modo de execução de destino quando da transição entre modos, sendo restaurado por instruções específicas para retorno ao modo de origem. A transição entre modos de execução pode ocorrer tanto em decorrência de eventos de *hardware*, por exemplo interrupções externas, quanto por eventos de *software*, como as chamadas de supervisor (interrupções internas) ou através de instruções dedicadas a esse fim, desde que executadas em modo privilegiado (ARM, 2012). Apresenta-se a seguir os modos de execução suportados pelo processador empregado, associados ao nível de privilégio utilizado por cada um deles (ARM, 2010):

***User* (PL0)** Utilizado para execução do *software* de usuário;

System* (PL1)** Equivalente ao modo de execução ***user no que se refere aos registradores acessíveis, porém privilegiado;

***FIQ* (PL1)** Utilizado para tratamento de *Fast Interrupt Requests* (*FIQs*);

***IRQ* (PL1)** Utilizado para tratamento de *Interrupt Requests* (*IRQs*);

***Supervisor* (PL1)** Utilizado para tratamento de chamadas de supervisor, também conhecidas como interrupções internas ou interrupções de *software*;

***Abort* (PL1)** Modo para o qual o processador é levado quando da ocorrência de erros de acesso à memória ou de leitura/decodificação de instruções;

***Undefined* (PL1)** Modo para o qual o processador é levado quando da execução de instruções inválidas.

A existência de diferentes modos de execução com diferentes níveis de privilégio permite a proibição da alteração das configurações do processador pelo *software* de aplicação, de forma que apenas o núcleo do SO tenha controle sobre elas. Além disso, esse recurso permite a restrição da utilização de determinadas instruções, causando a devolução

do controle do processador ao SO (através da geração de exceções) caso uma instrução inválida seja executada pela aplicação. Esse tipo de mecanismo é fundamental para a implementação de recursos requeridos pela especificação ARINC 653 como o isolamento espacial e o controle da propagação de erros (ARINC, 2006a). Detalhes sobre a forma como esses diferentes modos de execução são empregados no SO desenvolvido neste trabalho serão apresentados no Capítulo 4.

3.1.2 *Memory Management Unit* (*MMU*)

*MMU*s são empregadas para tradução de endereços virtuais – *Virtual Addresses* (*VAs*) –, utilizados pelo *software* de usuário, em endereços físicos – *Physical Addresses* (*PAs*) –, que são determinados pelo SO, além de permitirem o controle de acesso à memória através da restrição das regiões que são acessíveis às diferentes tarefas executadas, gerando exceções em caso de acessos inválidos (ARPACI-DUSSEAU, 2012). Por tratar-se de um recurso fundamental para o desenvolvimento de SOs robustos, as principais características da *MMU* empregada neste trabalho serão abordadas em profundidade a seguir, limitando-se porém às funcionalidades que são efetivamente utilizadas pelo SO desenvolvido.

A *MMU* da arquitetura *VMSAv7*, implementada pelo núcleo *ARM*® Cortex™-A8, faz uso de tabelas de tradução armazenadas na memória principal que permitem tanto o mapeamento de endereços quanto o controle de acesso sobre toda a memória do processador. Essas tabelas de tradução são formadas por um conjunto de descritores, cada um relacionado a um determinado intervalo de endereços virtuais e associado, explícita ou implicitamente, a um intervalo de endereços físicos, a um conjunto de atributos e ainda às permissões de acesso nos diferentes níveis de privilégio suportados pelo processador (ARM, 2010). Se esses descritores fossem consultados na memória principal toda vez que uma instrução que realiza acesso à memória é executada, o tempo consumido por esse tipo de operação seria proibitivo. Por esse motivo as *MMU*s utilizam uma memória *cache*, denominada *TLB*, que armazena os descritores mais comumente utilizados e evita, assim, a consulta frequente às tabelas de tradução na memória principal (ARPACI-DUSSEAU, 2012).

Quando o processador gera um acesso a um endereço de memória, a *MMU* realiza uma busca pelo endereço virtual requisitado na *TLB* de dados ou de instruções, de acordo com a operação que

está sendo realizada. Se nela existir um descritor compatível com a requisição, esse é recuperado e determina se o acesso é permitido e, caso seja, o endereço virtual é traduzido para o endereço físico para efetivação do acesso; caso essa verificação falhe é gerada uma exceção de acesso à memória. Caso a busca na *TLB* falhe, as tabelas de tradução são acessadas na memória principal, geralmente por mecanismos implementados em *hardware*, que armazenam o descritor correspondente na *TLB*. Em seguida a instrução que causou a falha é executada novamente, sendo então o descritor correspondente recuperado da *TLB* (ARPACI-DUSSEAU, 2012).

Tabelas de tradução podem ser utilizadas na arquitetura *VMSAv7* em dois formatos distintos, sendo que a escolha do formato empregado deve ser realizada levando em conta a aplicação à qual a *MMU* servirá. Os formatos de tabela de tradução suportados por essa arquitetura são (ARM, 2012):

Descritores curtos Possuem descritores de 32 *bits* e mapeiam endereços virtuais de 32 *bits* a endereços físicos de até 40 *bits*, permitindo regiões de 4KB, 64KB, 1MB ou 16MB;

Descritores longos São formadas por descritores de 64 *bits* e mapeiam endereços de até 40 *bits*, tanto virtuais quanto físicos, suportando regiões de 4KB, 2MB e 1GB.

Assume-se que, para as finalidades às quais o SO desenvolvido neste trabalho destina-se, não é necessária a utilização de mecanismos de memória virtual e, portanto, o emprego de endereços virtuais longos (maiores que 32 *bits*). Por isso, as informações apresentadas a seguir supõem o emprego de tabelas de tradução de descritores curtos, as quais são menores, cuja utilização é mais simples e, ainda assim, são suficientes nesse contexto. Tabelas de tradução de descritores curtos podem ser de primeiro ou de segundo nível, e são compostas da seguinte forma (ARM, 2012):

De primeiro nível Possui até 4096 descritores associados a endereços virtuais alinhados a 1MB, e cada um deles pode ser, exclusivamente, um descritor:

- De falha – a região (1MB) não pode ser acessada;
- De propriedades da seção (1MB) associada;
- De propriedades de uma superseção (16MB), sendo que superseções devem estar alinhadas a 16MB e seus descritores devem ser repetidos 16 vezes consecutivas na tabela de tradução;

- De tabela de segundo nível, com um ponteiro para a tabela que define o mapeamento da região (1MB).

De segundo nível Possui 256 descritores associados a endereços base alinhados a 4KB dentro da seção de 1MB apontada pelo registro da tabela de primeiro nível, e cada um deles pode ser, exclusivamente, um descritor:

- De falha – a região (4KB) não pode ser acessada;
- De propriedades da página pequena (4KB) associada;
- De propriedades de uma página grande (64KB), sendo que páginas grandes devem estar alinhadas a 64KB e seus descritores devem ser repetidos 16 vezes consecutivas na tabela de tradução de segundo nível.

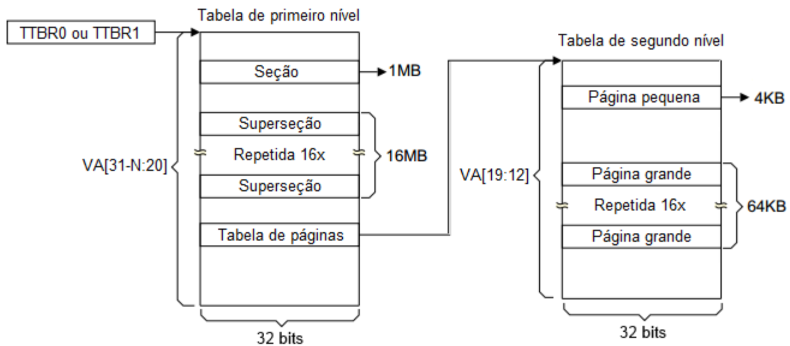
Na *MMU* da arquitetura *VMSAv7* estão disponíveis dois registradores apontadores de tabelas de tradução de primeiro nível: **TTBR0** e **TTBR1**. Usualmente, um deles é utilizado para apontar a tabela de tradução que define os acessos a regiões globais (válidas para qualquer tarefa do sistema), não sendo portanto alterada durante trocas de contexto, enquanto o outro é utilizado para apontar a tabela de tradução que define as regiões de memória acessíveis à tarefa atual, e que é portanto necessariamente alterada em trocas de contexto. A região de memória mapeada pela tabela de tradução apontada por **TTBR0** inicia no endereço zero (0x00000000) e tem tamanho variável definido através do campo **N** do registrador de controle **TTBCR**. Por sua vez, a tabela de tradução apontada pelo registrador **TTBR1** possui tamanho fixo de 4096 descritores (16KB), cobrindo portanto todo o espaço de endereços, porém é utilizada apenas na porção que não é mapeada por **TTBR0** segundo o campo **TTBCR.N**. A Tabela 1 apresenta como, de acordo com o valor atribuído a **TTBCR.N**, as regiões mapeadas por **TTBR0** e **TTBR1** são delimitadas quando do uso de tabelas de tradução de descritores curtos (ARM, 2012):

Tabela 1 – Mapeamento TTBR0-TTBR1

TTBCR.N	Região mapeada por TTBR0	Região mapeada por TTBR1
0b000	0x00000000-0xFFFFFFFF	Nenhuma
0b001	0x00000000-0x7FFFFFFF	0x80000000-0xFFFFFFFF
0b010	0x00000000-0x3FFFFFFF	0x40000000-0xFFFFFFFF
0b011	0x00000000-0x1FFFFFFF	0x20000000-0xFFFFFFFF
0b100	0x00000000-0x0FFFFFFF	0x10000000-0xFFFFFFFF
0b101	0x00000000-0x07FFFFFF	0x08000000-0xFFFFFFFF
0b110	0x00000000-0x03FFFFFF	0x04000000-0xFFFFFFFF
0b111	0x00000000-0x01FFFFFF	0x02000000-0xFFFFFFFF

Na Figura 4, adaptada de (ARM, 2012), representa-se a forma como são apontadas e interpretadas as tabelas de tradução de descritores curtos de primeiro e de segundo nível para o mapeamento de seções (1MB), superseções (16MB), páginas pequenas (4KB) e páginas grandes (64KB).

Figura 4 – Tabelas de tradução de descritores curtos



Conforme citado, na arquitetura *VMSAv7* as tabelas de tradução suportam a declaração de regiões de memória globais, cujo mapeamento é válido para qualquer contexto de execução, e de regiões de memória não globais, cujo mapeamento é válido apenas para o contexto de execução atual, o que torna necessária a utilização de um mecanismo de distinção de contextos de execução. Essa distinção é realizada através do registrador **CONTEXTIDR** da *MMU*, que deve ser carregado com o identificador do contexto de execução atual e, assim como cada entrada das *TLBs*, possui um campo denominado

Address Space Identifier (***ASID***). O campo ***ASID*** das *TLBs* armazena o ***ASID*** contido no registrador **CONTEXTIDR** no momento em que o descritor foi inserido na *TLB*, e uma entrada da *TLB* só é recuperada se, além de descrever os atributos da região de memória que está sendo acessada, possuir o campo ***ASID*** igual ao do registrador **CONTEXTIDR** no momento do acesso ou se o descritor armazenado for global (ARM, 2012). Dessa forma evita-se que dois descritores pertencentes a diferentes contextos de execução sejam utilizados erroneamente por referirem-se a uma mesma região de memória, e evita-se ainda que as *TLBs* precisem ser completamente invalidadas quando da troca de contexto de execução, o que faria com que as tabelas de tradução fossem sempre acessadas nessas ocasiões (ARPACI-DUSSEAU, 2012).

Os descritores de seção, superseção, página pequena e página grande das tabelas de tradução de descritores curtos da arquitetura *VMSAv7* suportam, entre outros, os seguintes atributos para cada região de memória (ARM, 2012):

Tipo de memória Define propriedades relacionadas ao comportamento do processador no acesso à região (quanto à memória *cache*, por exemplo);

Permissão de execução Indica se é permitida ou não a execução de instruções na região;

Permissão de acesso Indica o tipo de acesso permitido à região nos diferentes níveis de privilégio suportados pelo processador;

Globalidade Define se o descritor é ou não global, ou seja, se os atributos por ele definidos são ou não válidos para qualquer contexto de execução.

Quando utilizadas tabelas de tradução de descritores curtos da *VMSAv7*, a tradução de endereços virtuais (*VAs*) para endereços físicos (*PAs*), assim como a aplicação de restrições no acesso à memória, ocorre de acordo com o seguinte algoritmo (vale porém ressaltar que todo o processo é realizado por elementos de *hardware*) (ARM, 2012):

1. O endereço virtual é utilizado, inicialmente, para definição da tabela de tradução de primeiro nível a ser utilizada (**TTBR0** ou **TTBR1**) de acordo com o valor atribuído a **TTBCR.N**;
2. São utilizados os *bits* *VA*[31-N:20] (sendo **N**=0 caso seja utilizada **TTBR1**) para definição da região de 1MB na qual o endereço virtual está contido e, consequentemente, o descritor a ser acessado

na tabela de primeiro nível; caso o descritor da tabela de primeiro nível seja:

(a) De falta, é gerada uma falha de acesso à memória;
(b) De seção ou de superseção, o descritor determina o endereço físico do mapeamento, e suas propriedades indicam se o acesso é ou não permitido;
(c) De tabela de tradução, o descritor determina o endereço base da tabela de tradução de segundo nível, e então os *bits* *VA*[19:12] são utilizados para definição da região de 4KB dentro da região de 1MB de primeiro nível na qual o endereço virtual está contido e, consequentemente, o descritor a ser acessado na tabela de segundo nível; caso o descritor da tabela de segundo nível seja:
    i. De falta, é gerada uma falha de acesso à memória;
    ii. De página pequena ou grande, o descritor determina o endereço físico do mapeamento, e suas propriedades indicam se o acesso é ou não permitido.

No contexto deste trabalho, através da utilização da *MMU* é possível oferecer a garantia de que o *software* executado em uma determinada partição não acessará regiões de memória pertencentes às demais, garantindo assim o isolamento espacial exigido pela especificação (ARINC, 2006a). Detalhes sobre a forma como esse recurso foi utilizado no SO desenvolvido serão apresentados no Capítulo 4.

3.1.3 Coprocessador *VFP*

O núcleo *ARM*® Cortex™-A8 do processador empregado está equipado com um coprocessador que implementa a arquitetura *ARM*® *Vector Floating-Point* versão 3 (*VFPv3*), que é completamente compatível com o padrão *IEEE* 754-1985 de aritmética sobre valores binários de ponto flutuante. Esse coprocessador suporta operações de adição, subtração, multiplicação, divisão, multiplicação com acumulação e raiz quadrada sobre dados de precisão simples e dupla, além de instruções de conversão entre os formatos de dados de ponto fixo e de ponto flutuante (ARM, 2010).

A utilização desse mecanismo não é exigida pela ARINC 653, mas pode ser considerada desejável por acelerar a execução de operações numéricas que são muito comuns no processamento de informações de

sensores, por exemplo (ARINC, 2006a). Outras informações sobre esse coprocessador e sua utilização no SO desenvolvido serão apresentadas no Capítulo 4.

3.1.4 *Timers*

Os *timers* de propósito geral oferecidos pelo processador *TI* AM335X são utilizados para geração de eventos temporalmente precisos, podendo ser vistos simplificadamente como registradores de contagem crescente de 32 *bits* conectados a uma fonte de *clock*. Esses contadores podem ser lidos e escritos em tempo real (durante a contagem) e podem ser utilizados em três diferentes modos (TI, 2011):

Temporização A contagem pode ser iniciada ou parada a qualquer momento, podendo ainda ser geradas interrupções e/ou reiniciada a contagem (restaurando o valor inicial do contador) quando o valor máximo for excedido, ou seja, quando ocorrer *overflow* – utilizado para geração de interrupções periódicas, por exemplo;

Captura Copia o valor do contador para outro(s) registrador(es) quando da ocorrência de um evento (subida e/ou descida de borda) em uma determinada entrada digital do processador – utilizado para processamento de sinais digitais, por exemplo;

Comparação O valor do contador é continuamente comparado com o valor de um segundo registrador, podendo ser geradas interrupções e/ou pulsos em uma saída digital do processador quando forem iguais – utilizado para geração de sinais *PWM*, por exemplo.

A utilização de *timers* alinha-se à necessidade de fornecimento de recursos de temporização determinísticos pela plataforma de *hardware* para atendimento à especificação ARINC 653 (ARINC, 2006a). Outras informações sobre a forma como esses periféricos foram empregados no SO desenvolvido neste trabalho serão apresentadas no Capítulo 4.

3.2 MEMÓRIA

O processador *TI* AM335X possui dois bancos de memória *SRAM*, sendo um interno ao núcleo do processador e outro interno ao processador porém externo ao núcleo. O primeiro deles é mapeado na faixa de endereços de 0x402F0000 até 0x402FFFFF, tendo portanto

64KB, porém 1KB a partir de seu endereço inicial são reservados, podendo portanto ser utilizado apenas a partir do endereço 0x402F0400. O segundo deles está mapeado na faixa de endereços de 0x40300000 até 0x4030FFFF, tendo portanto 64KB, e pode ser utilizado integralmente (TI, 2011).

Esse processador fornece ainda uma interface que permite a utilização de memória *Random Access Memory* (*RAM*) externa dos tipos *DDR2*, *DDR3* ou *Mobile DDR* (*mDDR*) de até 1GB, que é mapeada na faixa de endereços de 0x80000000 até 0xBFFFFFFF. Na *BeagleBone* encontra-se conectada a essa interface uma memória *SDRAM* do tipo *DDR2* de 256MB, mapeada portanto na faixa de endereços de 0x80000000 até 0x8FFFFFFF (TI, 2011).

Informações relacionadas à forma como a memória da plataforma é utilizada pelo SO desenvolvido e pelas aplicações sobre ele executadas serão apresentadas no Capítulo 4.

3.3 PERIFÉRICOS

O processador *TI* AM335X oferece uma grande variedade de periféricos que servem às mais diversas finalidades, dentre os quais destaca-se (TI, 2011):

***General Purpose Input/Output* (*GPIO*)** Suporta entradas/saídas digitais na maioria dos pinos;

Acelerador gráfico Pode ser utilizado em aplicações que empregam gráficos 2D e/ou 3D;

Controlador *LCD* Utilizado para controle de visores *Liquid Crystal Display* (*LCD*);

Controlador *touchscreen* Subsistema de conversão analógico/digital com suporte a painéis resistivos sensíveis ao toque;

Subsistema *ethernet* Fornece comunicação segundo o padrão *IEEE* 802.3;

Subsistema *PWM* Possui módulos para geração, captura e tratamento de sinais;

***Universal Serial Bus* (*USB*)** Oferece dois módulos *USB On-The-Go* (*OTG*) para comunicação a até 480Mbps;

***Multimedia Card* (*MMC*)** Oferece três controladores para acesso a cartões de memória;

***Universal Asynchronous Receiver/Transmitter* (*UART*)** Oferece seis controladores para comunicação *serial*;

Controller Area Network (*CAN*) Possui dois controladores para comunicação a até 1Mbps.

Os registradores que permitem o controle desses periféricos são mapeados nos endereços de memória 0x44000000 até 0x7FFFFFFF, e portanto a restrição de sua utilização pode ser realizada através do controle de acesso a esse intervalo de endereços através da *MMU*. Ressalta-se que alguns desses registradores são acessíveis apenas em modo privilegiado, impedindo que alguns dos periféricos sejam utilizados a partir do *software* de usuário mesmo que o acesso aos registradores associados seja permitido pela configuração da *MMU* (TI, 2011). Vale ressaltar ainda que nem todos os periféricos oferecidos podem ser utilizados na plataforma *BeagleBone* em seu formato básico (sem placas de expansão), já que essa não possui todos os elementos de *hardware* externos ao processador que são necessários para isso. Detalhes sobre os elementos de *hardware* que estão disponíveis na plataforma em seu formato básico, assim como informações sobre algumas das placas de expansão que podem ser a ela conectadas, podem ser consultados no manual de referência do sistema da *BeagleBone* (COLEY, 2012).

Maiores informações sobre a utilização dos periféricos da plataforma quando empregado o SO desenvolvido neste trabalho serão fornecidas no Capítulo 4.

3.4 INTERFACES DE COMUNICAÇÃO

Dentre as principais interfaces de comunicação oferecidas pela plataforma *BeagleBone*, as quais destinam-se à troca de informações entre a plataforma e outros dispositivos, encontra-se a porta *USB device* utilizada para conexão da plataforma ao microcomputador. Essa porta serve às seguintes finalidades (COLEY, 2012):

Alimentação O suprimento de energia de 5V fornecido através da interface é utilizado para a alimentação da plataforma, descartando a necessidade de uma fonte externa de energia;

Depuração A plataforma possui uma interface de depuração *TI* XDS100v2, baseada no circuito integrado *Future Technology Devices International* (*FTDI*) FT2232 carregado com um *firmware* especial fornecido pela *TI*;

Porta *serial* virtual O circuito integrado *FTDI* FT2232 oferece duas portas *seriais* virtuais, uma destinada à interface de depuração e a outra conectada diretamente a uma *Universal*

Asynchronous Receiver/Transmitter (*UART*) do processador;

Porta *USB* Tanto o circuito integrado *FTDI* FT2232 quanto a porta *USB0* do processador são conectados a um *hub USB* localizado na placa da plataforma, e portanto essa porta também é acessível a partir do microcomputador.

Destaca-se ainda a porta *ethernet* da plataforma, que pode ser utilizada para acesso a redes de comunicação compatíveis com o padrão *IEEE* 802.3 e, portanto, permite sua utilização na construção de sistemas distribuídos. O subsistema *ethernet* do processador consiste em um *switch* de três portas, das quais duas podem ser conectadas a portas externas (na *BeagleBone* apenas uma delas encontra-se efetivamente conectada) e a terceira é acessada através de uma interface *Direct Memory Access* (*DMA*) a partir do processador. Esse *switch* pode ser configurado para comutar pacotes entre as portas de acordo com critérios diversos, como em função de endereços *MAC* específicos ou através da criação de *Virtual Local Area Networks* (*VLANs*), ou ainda pode realizar e atualizar essa configuração de forma automática através da análise do tráfego de pacotes (TI, 2011).

Estão ainda disponíveis na plataforma de *hardware* utilizada uma porta *USB host*, que é conectada à interface *USB1* do processador e pode ser empregada na comunicação com dispositivos *USB client* externos, e uma porta *MicroSD*, que é conectada a uma das interfaces *Multimedia Card* (*MMC*) do processador e permite a utilização de cartões de memória (COLEY, 2012).

A utilização das interfaces de comunicação oferecidas pela plataforma *BeagleBone* em conjunto com o SO desenvolvido neste trabalho permite a construção de sistemas complexos, levando a uma experiência realista no que compete à interação com elementos de *hardware* a partir de partições ARINC 653. Maiores informações sobre a utilização dessas interfaces em conjunto com o SO desenvolvido neste trabalho serão fornecidas no Capítulo 4.

3.5 FERRAMENTAS DE *SOFTWARE*

Conforme citado anteriormente, a *BeagleBone* dispõe de uma interface de depuração denominada *TI* XDS100v2. Essa interface permite, por exemplo, o controle do fluxo de execução do *software* a nível de instrução de máquina e a leitura da memória do processador, sendo portanto uma importante ferramenta para o desenvolvimento ou migração de SOs para essa plataforma. Essa interface é suportada apenas

pela *Integrated Development Environment* (*IDE*) *Code Composer Studio*, fornecida pela própria *Texas Instruments*, que pode trabalhar em conjunto com os compiladores *TI ARM Compiler* e *GNU C Compiler (GCC).* Para suporte à programação de processadores AM335X, é fornecido ainda pela *Texas Instruments* um pacote denominado *AM335X StarterWare.* Esse pacote contém, entre outros recursos, bibliotecas para utilização dos periféricos do processador, exemplos da utilização dessas bibliotecas e um conjunto de ferramentas auxiliares com finalidades diversas (COLEY, 2012; TI, 2013b, 2013c).

3.6 INICIALIZAÇÃO DA PLATAFORMA

O processador *TI* AM335X possui um *bootloader* estático, armazenado em uma memória *Read-Only Memory* (*ROM*) interna, que é responsável pelos primeiros estágios de sua inicialização. Em função de configurações aplicadas a determinados pinos do processador, esse *bootloader* tenta obter o programa a ser executado a partir de diferentes origens, armazena-o na memória *SRAM* e dispara sua execução. Na plataforma *BeagleBone*, o processador está configurado para que o programa seja buscado a partir das seguintes origens e na seguinte ordem (TI, 2011):

1. A partir do cartão *SD*, no arquivo de nome **MLO**;
2. A partir da interface *SPI*;
3. A partir da porta *USB client* da plataforma:
    (a) Interface *UART0* do processador, acessível via porta *serial* virtual, através do protocolo **XMODEM**;
    (b) Porta *USB0* do processador, sobre a qual é fornecida uma interface de rede *Remote Network Driver Interface Specification* (*RNDIS*) e executado um servidor *Trivial File Transfer Protocol* (*TFTP*).

Diferentemente da memória *SRAM*, que está disponível a partir do momento em que o suprimento de energia é fornecido ao processador, a memória *RAM* externa precisa ser configurada antes que possa ser utilizada. Por isso a inicialização da plataforma de *hardware* é geralmente realizada em duas etapas (TI, 2013c):

1. O *bootloader* estático do processador é utilizado para carga de um *bootloader* específico (desenvolvido de acordo com as características da plataforma de *hardware*) na memória *SRAM*;

2. Executando a partir da *SRAM*, o *bootloader* específico configura a interface de memória *RAM* externa, carrega o código da aplicação nessa memória e, então, dispara sua execução.

Um *bootloader* específico para a *BeagleBone* é fornecido no pacote *AM335X StarterWare*. Esse *bootloader* inicializa a memória *SDRAM* externa, carrega nela a imagem de memória da aplicação a partir do arquivo **APP** contido no cartão *SD* e finalmente dispara sua execução a partir do endereço indicado no cabeçalho desse arquivo (TI, 2013c). Essa abordagem garante que os endereços de memória ocupados pelas informações do sistema são os mesmos em tempo de depuração e em tempo de execução, o que acaba por beneficiar o aspecto didático do SO desenvolvido. Um exemplo desse benefício fica evidente no mecanismo de tratamento de erros, pelo qual são fornecidos endereços de memória (de dados ou de instruções) aos quais os erros estão relacionados, tornando fundamental o mapeamento desses endereços às informações às quais eles se referem.

O processo de transformação dos arquivos gerados na compilação de uma aplicação em imagens de memória que podem ser carregadas por esse *bootloader* específico está intimamente relacionado ao ambiente de desenvolvimento utilizado, já que geralmente depende de ferramentas fornecidas em conjunto com o compilador. As instruções para realização dessa transformação no ambiente empregado neste trabalho serão apresentadas no Apêndice C e exemplificadas no Apêndice B.

3.7 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentadas as principais características da *BeagleBone*, a plataforma de *hardware* empregada neste trabalho, assim como uma breve avaliação dos recursos oferecidos por ela frente aos requisitos impostos pela especificação ARINC 653 no que se refere ao equipamento sobre o qual SOs compatíveis devem ser idealmente executados (ARINC, 2006a).

Observa-se que alguns dos recursos oferecidos por essa plataforma, como por exemplo *branch prediction* e memória *cache*, não atendem a alguns dos requisitos impostos pelo tipo de sistema ao qual a especificação ARINC 653 se destina. Isso a torna pouco adequada para uso em aeronaves, mas não impede sua utilização com fins didáticos e experimentais no contexto de sistemas aviônicos. Observa-se ainda que a grande diversidade de dispositivos periféricos por ela fornecidos oferece muitas possibilidades para a construção de experimentos, e é

portanto fundamental destacar que este trabalho explora apenas uma pequena porção de suas potencialidades.

No próximo capítulo será apresentada a implementação do SO desenvolvido, detalhando seu código fonte e a forma como os recursos da plataforma de *hardware* foram empregados para atendimento à especificação ARINC 653.

4 IMPLEMENTAÇÃO

As ferramentas de *software* utilizadas neste trabalho para programação da plataforma alvo foram a *IDE TI Code Composer Studio* versão 6.0.1.00039, em conjunto com o compilador *TI ARM Compiler* versão 5.1.6. Utilizou-se ainda, para obtenção de exemplos e de código para interação em baixo nível com os periféricos do processador, o pacote de desenvolvimento *TI AM335X StarterWare* versão 02.00.01.01 (TI, 2013b, 2013a, 2013c).

Neste capítulo serão apresentados diversos aspectos da implementação realizada neste trabalho, destacando-se a organização do código fonte produzido, a forma como os recursos da plataforma de *hardware* escolhida foram utilizados, e as abordagens escolhidas para a implementação dos recursos fundamentalmente empregados em SOs, como filas de prioridades e algoritmos de escalonamento. Grande parte dos detalhes técnicos dessa implementação (tipos de dados, estruturas, serviços e métodos empregados), assim como a descrição dos processos de migração e execução do SO, podem ser encontrados no Apêndice C.

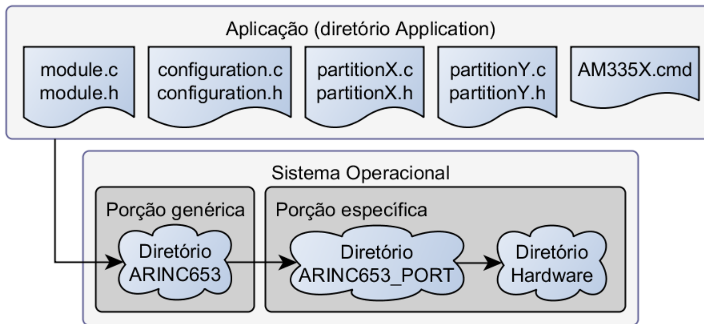
4.1 ORGANIZAÇÃO DO CÓDIGO FONTE

O código fonte do SO desenvolvido neste trabalho foi organizado de forma que sua porção genérica, que não varia em função da plataforma de *hardware* utilizada, permaneça isolada da porção específica, minimizando assim o esforço necessário para o suporte futuro a novas plataformas. Também é objetivo da organização de código empregada a expansibilidade futura dos elementos do SO, e por esse motivo foi realizada uma separação criteriosa dos conceitos de sistema e de módulo. Essa separação, apesar de não ser exigida pela especificação, permite que futuramente seja utilizada uma plataforma de *hardware* multiprocessada para a execução de múltiplos módulos sobre um único sistema, sendo necessárias para isso apenas alterações triviais. Ressalta-se ainda que, uma vez que esse SO possui apenas finalidades didáticas e experimentais, diversos dos algoritmos utilizados não encontram-se em sua forma otimizada, ou seja, o código fonte foi elaborado de forma a ser o mais compreensível, e não o mais veloz, quanto possível.

Apresenta-se, na Figura 5, o conjunto de diretórios e arquivos que compõe um projeto que faz uso do SO desenvolvido neste trabalho, representando também a divisão e as relações de utilização existentes

entre as diferentes porções de código contidas em cada diretório. Como pode-se observar, a aplicação (diretório **Application**) deve acessar exclusivamente a porção genérica do código do SO (contida no diretório **ARINC653**), que fornece a *API* definida pela ARINC 653, enquanto a porção específica do SO (localizada no diretório **ARINC653_PORT**) é responsável pelo acesso às bibliotecas de interação com o *hardware* (armazenadas no diretório **Hardware**).

Figura 5 – Componentes do SO



Apresenta-se, a seguir, detalhes sobre o conteúdo de cada um dos diretórios e arquivos apresentados, citando de forma sucinta a função de cada um desses elementos no SO desenvolvido. Vale ressaltar que, para a migração do SO a novas plataformas de *hardware*, podem ser necessários outros arquivos e/ou diretórios além destes, ou alguns deles podem não ser necessários.

O diretório **Application** contém os arquivos relacionados ao *software* de aplicação, que são apresentados a seguir:

module.c/module.h Contêm a implementação do módulo, que é composta de sua partição padrão e de seu *HM callback* (caso exista), além de estruturas que definem os atributos de cada uma de suas partições;

partitionX.c/partitionX.h Contêm a implementação da partição de nome **PartitionX**, que é composta de seu processo padrão, seus processos de aplicação, seu processo tratador de erros e seu *HM callback* (caso esses existam), e contêm ainda as estruturas de atributos de seus processos;

configuration.c/configuration.h Gerados com base no arquivo de

configuração do módulo, contêm as informações de configuração de todos os elementos do sistema;

AM335X.cmd Arquivo de comandos do *linker* (específico da plataforma de *hardware* utilizada), que é gerado com base no arquivo de configuração do módulo e determina o mapeamento de memória do sistema.

O diretório **ARINC653** contém a porção genérica do código do núcleo do SO, e é formado pelos seguintes arquivos:

arinc653.h Dá acesso a todas as definições necessárias ao *software* de aplicação, sendo o único arquivo do SO que deve ser incluído por esse;

arinc653_clock.h Define macros de manipulação do relógio do módulo;

arinc653_common.c/arinc653_common.h Contêm funções de propósito geral empregadas pelo núcleo;

arinc653_configuration.c/arinc653_configuration.h Contêm funções de acesso às estruturas de configuração do sistema;

arinc653_core.c/arinc653_core.h Definem estruturas e funções utilizadas no núcleo do SO para controle dos elementos do sistema;

arinc653_priorityqueue.c/arinc653_priorityqueue.h Contêm a implementação da fila de prioridades empregada pelo núcleo do SO;

arinc653_scheduler.c/arinc653_scheduler.h Contêm o algoritmo de escalonamento de partições e processos;

arinc653_heap.c/arinc653_heap.h Definem estruturas e funções para gerenciamento da alocação dinâmica de memória;

arinc653_systemManagement.c Contém os serviços de gerenciamento do sistema;

arinc653_moduleManagement.c Contém os serviços de gerenciamento do módulo;

arinc653_partitionManagement.c Contém os serviços de gerenciamento de partições;

arinc653_processManagement.c Contém os serviços de gerenciamento de processos;

arinc653_timeManagement.c Contém os serviços de gerenciamento de tempo;

arinc653_healthMonitoring.c Contém os serviços do mecanismo de monitoramento;

arinc653_interpartitionCommunication_samplingPort.c Contém os serviços de comunicação interpartição com portas de amostragem;

arinc653_interpartitionCommunication_queuingPort.c Contém os serviços de comunicação interpartição com portas de enfileiramento;

arinc653_intrapartitionCommunication_buffer.c Contém os serviços de comunicação intrapartição com *buffers*;

arinc653_intrapartitionCommunication_blackboard.c Contém os serviços de comunicação intrapartição com *blackboards*;

arinc653_intrapartitionCommunication_semaphore.c Contém os serviços de comunicação intrapartição com semáforos;

arinc653_intrapartitionCommunication_event.c Contém os serviços de comunicação intrapartição com eventos.

O diretório **ARINC653_PORT** contém a porção específica do código do núcleo do SO para a plataforma de *hardware* utilizada, que é formada pelos seguintes arquivos:

arinc653_port_types.h Contém definições de tipos de dados básicos;

arinc653_port.c/arinc653_port.h Contêm a implementação em linguagem C dos métodos requeridos pelo núcleo do SO para operação na plataforma de *hardware*;

arinc653_port_asm.asm Contém a implementação em linguagem *assembly* de métodos requeridos pelo núcleo do SO;

arinc653_port_reset_asm.asm Contém a implementação em linguagem *assembly* do procedimento de inicialização do processador;

arinc653_systempartition_io.c Contém a implementação de uma partição de sistema destinada ao tratamento da troca de dados para mecanismos de comunicação interpartição, incluindo a transmissão dessas informações através de uma ou mais interfaces de comunicação oferecidas pela plataforma de *hardware* utilizada.

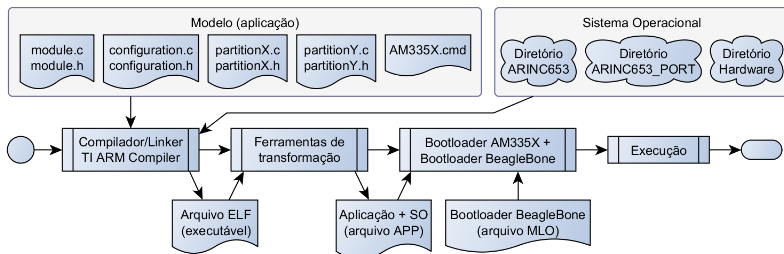
O diretório **Hardware**, por sua vez, contém diversas bibliotecas para interação com a plataforma de *hardware*, e portanto seu conteúdo

está intimamente relacionado a ela. A maioria das bibliotecas encontradas nesse diretório foram obtidas do pacote de desenvolvimento *AM335X StarterWare*, com exceção da biblioteca de manipulação das tabelas de tradução da *MMU*, que foi desenvolvida como parte deste trabalho por não possuir equivalente nesse pacote. As bibliotecas obtidas a partir do *AM335X StarterWare* sofreram o mínimo possível de alterações, a fim de evitar sua descaracterização, tendo sido apenas adicionados ou complementados procedimentos cujo formato original não atendia às necessidades do SO desenvolvido.

Conforme pode-se observar nessa estrutura, o código do núcleo do SO é compilado em conjunto com o *software* de aplicação executado sobre ele. Outras abordagens poderiam ser utilizadas, podendo por exemplo o SO carregar o *software* de aplicação em tempo de execução, porém dessa forma a depuração do código do núcleo em conjunto com o da aplicação só seria possível com a elaboração de elementos de instrumentação de código – o que não pertence ao escopo deste trabalho. Uma vantagem relevante dessa abordagem está no fato de que, com ela, o mapa de memória completo do sistema é definido antes de sua execução, e portanto os endereços e tamanhos das principais regiões de memória utilizadas por seus elementos são previamente conhecidos, o que representa ganhos didáticos consideráveis.

Apresenta-se na Figura 6 uma representação em alto nível do processo de compilação e execução de uma aplicação sobre o SO desenvolvido neste trabalho na plataforma de *hardware* empregada. Nesse processo, que é detalhado e exemplificado no Apêndice B, o código fonte do SO e das aplicações sobre ele executadas é compilado e transformado, através de um conjunto de ferramentas, para que possa ser finalmente carregado por um *bootloader* para execução na plataforma.

Figura 6 – Processo de compilação e execução



4.2 GERÊNCIA DE PROCESSADOR: FILA DE PRIORIDADES

Filas de prioridades são estruturas de dados que têm como função a seleção de elementos de forma ordenada, ascendente ou descendentemente, com base em um atributo desses elementos – que pode variar de acordo com o fim ao qual a estrutura serve – denominado **prioridade**. Por exemplo, para controle dos processos em espera por um determinado recurso pode ser utilizada uma fila de prioridades na qual seu atributo de ordenação (**prioridade**) indica, por exemplo, o tempo máximo durante o qual o processo utilizará o recurso. Nesse caso, a ordenação da fila de prioridades definirá se serão atendidos antes processos que utilizarão o recurso por pouco tempo ou antes aqueles que demandam sua utilização por um longo período (WEISS, 2011). Esta seção tem por objetivo apresentar a fila de prioridades implementada para o SO desenvolvido neste trabalho, destacando algumas das funções nas quais essa foi empregada.

Diversos dos recursos bloqueantes oferecidos por SOs compatíveis com a ARINC 653 devem suportar tanto o atendimento de processos em ordem de prioridade quanto em ordem de chegada, ou seja, em política *FIFO* (ARINC, 2006a). Por esse motivo, a fila de prioridades desenvolvida neste trabalho pode ter a ordenação por prioridades desabilitada, passando assim a apresentar comportamento idêntico ao de uma fila *FIFO*. Essa fila de prioridades consiste em uma lista duplamente encadeada de estruturas do tipo **priorityqueueENTRY** que contêm dois campos principais: a prioridade e o valor. A prioridade, conforme citado, define a ordenação das entradas na fila, e o valor é um campo genérico utilizado para identificar o elemento ao qual a entrada se refere. É definida ainda uma estrutura raiz do tipo **priorityqueueRECORD** que aponta a cabeceira da fila, indica se a fila utiliza ou não ordenação por prioridades e, caso utilize, indica ainda o tipo de ordenação por ela utilizado (ascendente ou descendente). Quando utilizada ordenação por prioridades, a cabeceira da fila é o elemento de maior ou menor prioridade nela contido segundo o tipo de ordenação empregado, e quando utilizada ordenação *FIFO* a cabeceira é o elemento mais antigo da fila (inserido há mais tempo). Por tratar-se de uma lista duplamente encadeada, as entradas possuem ponteiros para os elementos anterior e posterior, e portanto uma entrada nunca pode ser inserida em mais de uma fila de prioridades ao mesmo tempo.

O algoritmo de enfileiramento empregado itera sobre os elementos da fila até a localização da posição correta de inserção do elemento,

apresentando portanto complexidade da ordem $O(n)$. O algoritmo de remoção, tanto para a cabeceira quanto para qualquer outro elemento da fila, utiliza os ponteiros da lista duplamente encadeada, apresentando portanto complexidade $O(1)$. O acesso à cabeceira da fila sem removê-la, que é a operação mais frequentemente realizada pelo SO desenvolvido sobre as filas de prioridade, é realizada diretamente através do registro raiz e, portanto, também possui complexidade $O(1)$. A utilização de um algoritmo de enfileiramento iterativo de complexidade $O(n)$ justifica-se pela necessidade de atendimento à característica exigida pela especificação para diversos elementos do SO de que, se inseridas apenas entradas de mesma prioridade numa fila, o elemento da cabeceira deve ser aquele mais antigo, ou seja, nessa situação a fila de prioridades deve comportar-se como uma fila de política *FIFO* – essa exigência pode ser verificada tanto para o escalonamento de processos quanto para o tratamento de processos bloqueados em recursos (ARINC, 2006a). Na ausência dessa exigência seria possível o emprego de estruturas mais eficientes, como *heaps* binárias, que ofereceriam enfileiramento e remoção com complexidade $O(log_2 n)$; porém, destaca-se que os ganhos decorrentes da utilização de uma fila de prioridades mais eficiente seriam relevantes apenas para grandes sistemas (dezenas a centenas de processos).

No SO desenvolvido neste trabalho, filas de prioridades são utilizadas para desempenho de diversas tarefas, das quais destaca-se por exemplo:

Escalonamento de partições Emprega uma fila de prioridades com ordenação ascendente que tem como atributo de ordenação (prioridade) o horário de início da janela de tempo das partições prontas e como valor o identificador da partição, bastando para realização do escalonamento a escolha da partição da cabeceira da fila até que o horário de fim de sua janela de tempo atual seja alcançado;

Escalonamento de processos Emprega filas de prioridades com ordenação descendente cujo atributo de ordenação é a prioridade atual dos processos prontos e cujo valor é o identificador do processo, sendo executado sempre o processo da cabeceira da fila, ou seja, o de maior prioridade;

Bloqueio de processos Cada recurso bloqueante do SO possui uma fila de prioridades com ordenação descendente que tem como valor o identificador do processo e como atributo de ordenação a prioridade atual do processo, sendo atendido primeiro sempre o

processo que encontra-se na cabeceira da fila; caso o recurso opere com atendimento de processos em política *FIFO*, a ordenação por prioridade da fila é desabilitada.

São apresentados a seguir os tipos de dados utilizados pela fila de prioridades implementada, associados aos tipos genéricos aos quais são mapeados no SO desenvolvido (cuja definição pode ser consultada no Apêndice C). Observa-se que os tipos genéricos utilizados nesse mapeamento são os mais abrangentes dentre os suportados, para que a fila de prioridades possa servir a diversos fins sem a necessidade de emprego de diferentes tipos de dados.

priorityqueuePRIORITY (portUINT64) Armazena os atributos de ordenação de elementos (prioridades);

priorityqueueVALUE (portUINT64) Armazena os atributos de distinção de elementos (valores);

priorityqueueSIZE (portSIZE) Armazena o tamanho de uma fila, ou seja, o número de elementos nela contidos.

Apresenta-se a seguir uma breve descrição dos serviços fornecidos ao núcleo do SO desenvolvido neste trabalho para manipulação de filas de prioridades:

PRIORITYQUEUE_STARTUP Inicializa uma fila de prioridades, limpando-a e definindo se será ordenada por prioridade e, em caso positivo, seu tipo de ordenação;

PRIORITYQUEUE_INITIALIZEENTRY Inicializa uma entrada de fila de prioridades, considerando-a não enfileirada;

PRIORITYQUEUE_ENQUEUE Insere uma entrada em uma fila de prioridades, falhando caso essa já encontre-se enfileirada;

PRIORITYQUEUE_REMOVE Remove uma entrada da fila de prioridades na qual essa encontra-se enfileirada;

PRIORITYQUEUE_CLEAR Limpa uma fila de prioridades, removendo todos seus elementos e redefinindo seus parâmetros de ordenação;

PRIORITYQUEUE_GETCOUNT Retorna o número de entradas existentes em uma fila de prioridades;

PRIORITYQUEUE_ISEMPTY Retorna um valor lógico que indica se uma determinada fila de prioridades encontra-se vazia;

PRIORITYQUEUE_ISPRIORITIZED Retorna um valor lógico que indica se uma determinada fila de prioridades utiliza ordenação por prioridade;

PRIORITYQUEUE_ISASCENDING Retorna um valor lógico que indica se uma determinada fila de prioridades utiliza ordenação ascendente;

PRIORITYQUEUE_ISENQUEUED Retorna um valor lógico que indica se uma determinada entrada encontra-se enfileirada em alguma fila de prioridades.

4.3 GERÊNCIA DE PROCESSADOR: CONTEXTOS DE EXECUÇÃO

SOs geralmente suportam a execução de diversas tarefas de forma concorrente, ou seja, executam tarefas de forma alternada sobre um núcleo de processamento a fim de dar a impressão de que estão sendo executadas de forma paralela. Essa abordagem exige que a execução de uma tarefa possa ser interrompida a um determinado momento e retomada mais tarde, tornando necessário que as informações relacionadas ao estado do núcleo do processador (valores de registradores) sejam armazenadas quando de sua interrupção, para que possam ser restauradas no momento em que a execução da tarefa for retomada. A esse conjunto de informações dá-se o nome de **contexto de execução**, e o processo de interrupção de uma tarefa e retomada de outra é chamado de **troca de contexto de execução** (ARPACI-DUSSEAU, 2012). A troca de contexto de execução deve ser realizada de forma que, do ponto de vista de uma tarefa, após a restauração de seu contexto de execução o estado do processador seja exatamente o mesmo que era no momento em que essa foi interrompida (TANENBAUM; WOODHULL, 2005). Diferentes abordagens são possíveis para o armazenamento e restauração de contextos de execução, das quais duas serão apresentadas a seguir:

Armazenamento na pilha da tarefa Quando uma tarefa é interrompida os valores dos registradores são salvos na pilha de execução da tarefa, e para restauração esses são desempilhados e transferidos de volta aos mesmos registradores; essa abordagem oferece baixo custo computacional, já que o topo da pilha de execução da tarefa é apontado por um registrador do núcleo de processamento e muitas arquiteturas permitem o empilhamento/desempilhamento de múltiplos registradores através de uma única instrução, porém pode ser desvantajosa em caso de estouro da pilha, já que nesse caso se torna impossível o armazenamento do contexto de execução da tarefa

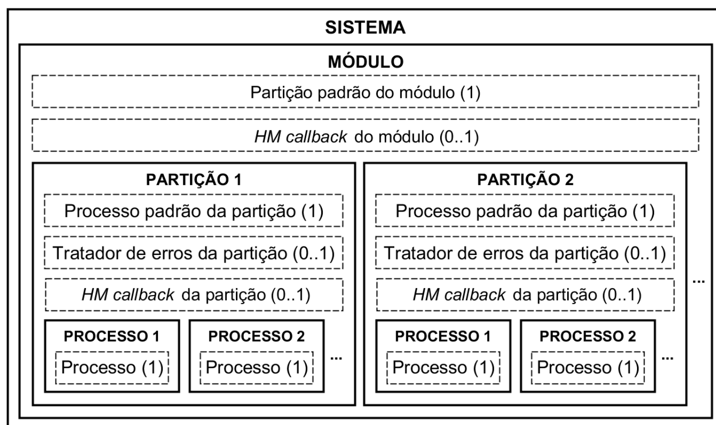
(REAL TIME ENGINEERS LTD., 2014);

Armazenamento em estrutura dedicada Quando uma tarefa é interrompida seu contexto de execução é armazenado em uma estrutura de dados dedicada a esse fim, que é alocada junto às demais informações da tarefa; nesse caso sua pilha de execução não é alterada, porém existe um *overhead* associado ao acesso à estrutura de dados na qual o armazenamento é realizado, já que seu endereço encontra-se na memória e não em um registrador, como é o caso da abordagem anteriormente apresentada (OAR CORPORATION, 2014).

No SO desenvolvido neste trabalho utilizou-se a abordagem de salvamento de contextos de execução em estruturas de dados dedicadas. Apesar de apresentar um custo de processamento ligeiramente maior em relação à outra abordagem apresentada, essa permite o salvamento do contexto de execução quando da ocorrência de erros de estouro de pilha. Isso acaba por tornar o tratamento desse tipo de erro (que é exigido pela especificação ARINC 653) mais simples e direto, por garantir a integridade das informações do contexto de execução afetado (ARINC, 2006a).

Em SOs tradicionais as tarefas que concorrem por execução no processador são as *threads*, sendo que cada processo possui no mínimo uma delas e pode realizar a criação de outras conforme a aplicação à qual serve (ARPACI-DUSSEAU, 2012). Por sua vez, SOs compatíveis com a especificação ARINC 653 possuem múltiplas tarefas ligadas aos diversos elementos do sistema, e o escalonamento de cada uma delas varia de acordo com a função à qual servem. Ilustra-se na Figura 7 a disposição dessas tarefas (em linhas tracejadas) em relação aos elementos do sistema aos quais estão relacionadas, e ainda sua cardinalidade (entre parênteses), que indica o número de tarefas do tipo em questão que podem ser encontradas no elemento ao qual pertence (ARINC, 2006a).

Figura 7 – Tarefas de sistemas ARINC 653



No SO desenvolvido neste trabalho, cada uma dessas tarefas representa um diferente contexto de execução. Para cada um deles é alocada uma região de memória para utilização como pilha e uma estrutura do tipo **CONTEXT_TYPE**, que armazena o estado do núcleo de processamento enquanto esse não estiver em execução, além de outras porções de memória utilizadas pelo núcleo do SO para outras finalidades. Entre as informações armazenadas pela estrutura **CONTEXT_TYPE** encontram-se o identificador do contexto de execução, apontadores para o início e o fim da região de memória utilizada por ele como pilha e um contador da profundidade de chamadas de serviços do SO atualmente efetuadas – que é utilizado para controle da habilitação de interrupções para que esses serviços sejam executados atomicamente. Outro importante atributo dessa estrutura é o endereço **PORT_CONTEXT**, que aponta uma estrutura de dados do tipo **PORT_CONTEXT_TYPE** cuja definição é específica da plataforma de *hardware* utilizada, e é empregada para armazenamento do conteúdo dos registradores do processador quando o contexto de execução é salvo, podendo armazenar também quaisquer outras informações que mostrarem-se necessárias. Apresenta-se a seguir os atributos da estrutura **PORT_CONTEXT_TYPE** empregados neste trabalho:

Identificador Identificador do contexto de execução a ser atribuído ao registrador **CONTEXTIDR** da *MMU*, que é utilizado para distinção dos contextos de execução do sistema frente ao mecanismo

de proteção de memória;

Endereço da tabela de tradução Endereço da tabela de tradução utilizada pela *MMU* durante o processamento do contexto de execução;

Registradores R0-R15 e CPSR São os registradores do núcleo do processador, incluindo **SP**, **LR** e **PC**, e o registrador de estado do núcleo **CPSR**;

Registradores D0-D15, FPSCR e FPEXC São os registradores do coprocessador *VFP* – observa-se que os registradores D16-D31 desse coprocessador são desabilitados pelo SO, já que não são necessários para operações numéricas típicas, e portanto não precisam ser salvos no contexto de execução.

Apresenta-se a seguir os contextos de execução utilizados pelo SO desenvolvido e suas funções, associando ao nome de cada um deles uma sigla que é empregada a seguir para referência resumida.

Partição padrão do módulo (MOD.DEF) Contexto de execução específico do SO implementado neste trabalho, ou seja, cuja existência não é exigida pela especificação ARINC 653, no qual o módulo é inicializado (suas partições são criadas e inicializadas); após a chamada ao serviço **SET_MODULE_MODE** para transição ao modo de operação **NORMAL**, passa a ser executado apenas durante as janelas de tempo do *major frame* que não foram alocadas a qualquer partição, passando então a ser denominado **partição ociosa do módulo**;

***HM callback* do módulo (MOD.HMC)** Contexto de execução do mecanismo de monitoramento que é disparado quando da ocorrência de erros mapeados em configuração para o nível de módulo;

***HM callback* de partição (PAR.HMC)** Contexto de execução do mecanismo de monitoramento que é disparado quando da ocorrência de erros mapeados em configuração para o nível de partição, ou na ocorrência de erros mapeados para o nível de processo quando da ausência de um processo tratador de erros na partição (ARINC, 2006a);

Tratador de erros de partição (PAR.EH) Contexto de execução do processo tratador de erros de uma partição, o qual é disparado quando da ocorrência de erros mapeados em configuração para o nível de processo;

Processo padrão de partição (PAR.DEF) Contexto de execução no qual uma partição é inicializada, ou seja, tem seus processos, artefatos de comunicação e demais elementos criados e inicializados; segundo a especificação, este contexto de execução pode ser ou não mantido após a transição da partição para o modo de operação **NORMAL** por uma chamada ao serviço **SET_PARTITION_MODE** (ARINC, 2006a); portanto, na implementação realizada neste trabalho, esse contexto de execução é mantido e passa a ser executado durante os intervalos de tempo alocados para a partição nos quais nenhum processo está pronto para execução, passando então a ser denominado **processo ocioso da partição**;

Processo (PRO) Contexto de execução de um processo de uma partição, utilizado para execução do *software* de aplicação.

4.4 GERÊNCIA DE PROCESSADOR: ESCALONAMENTO DE TAREFAS

Algoritmos de escalonamento de tarefas empregados em SOs são responsáveis por decidir qual (ou quais) dos contextos de execução disponíveis deve(m) receber recursos de processamento em um determinado momento, e a forma como eles são implementados influencia diretamente no comportamento temporal do sistema. Em SOs tradicionais, por exemplo, os algoritmos de escalonamento empregados têm por objetivo alocar a execução de processos de forma que nenhum deixe de ser executado durante longos períodos, a menos que encontrem-se bloqueados (ARPACI-DUSSEAU, 2012). Em SOTRs o escalonamento é geralmente realizado com base em prioridades, ou seja, o processo escolhido é sempre o de maior prioridade que encontra-se pronto para execução, porém são possíveis outras abordagens para obtenção de diferentes características temporais (ARPACI-DUSSEAU, 2012; WONG et al., 2008; CARPENTER et al., 2004). Por sua vez, os algoritmos empregados em SOs compatíveis com a especificação ARINC 653 devem realizar o escalonamento de tarefas em dois níveis: no primeiro nível ocorre o escalonamento de partições, de acordo com a escala temporal estática definida no arquivo de configuração do módulo, e no segundo nível é realizado o escalonamento dos processos de cada uma das partições com base em suas prioridades atuais (ARINC, 2006a).

O escalonamento de partições e de processos do SO desenvolvido neste trabalho é realizado através de uma interrupção periódica

disparada a cada 1ms (mas cujo período pode ser ajustado) que salva o contexto de execução atual, atualiza o relógio do sistema, executa o algoritmo de escalonamento e, finalmente, restaura o (possivelmente novo) contexto de execução. Para geração dessa interrupção periódica empregou-se o *timer* de propósito geral denominado **DMTimer2**, configurado para operar em modo de temporização com recarga automática do contador quando da ocorrência de estouro (TI, 2011). Devido à frequência com que o algoritmo de escalonamento é executado, fica evidente que a granularidade da escala de partições e dos atributos temporais de processos é da ordem de milissegundos. Essa frequência pode ser reduzida a fim de diminuir o *overhead* a ela associado, porém com essa redução a granularidade do escalonamento de partições e de processos é diretamente afetada. Por exemplo, com a alteração do período de execução do escalonador para dez milissegundos, todos os horários das janelas de tempo da escala de partições precisarão ser necessariamente convertidos em múltiplos desse valor, ou o escalonamento não será realizado corretamente.

Para implementação do algoritmo de escalonamento do SO desenvolvido neste trabalho, tomou-se como base aquele utilizado pelo **FreeRTOS** (REAL TIME ENGINEERS LTD., 2014) para escalonamento de processos por prioridade, sendo portanto necessário complementá-lo com o escalonamento de partições segundo uma escala estática e, ainda, o disparo de tarefas de alta prioridade para tratamento de erros segundo as exigências da ARINC 653. A fim de facilitar a compreensão de seu funcionamento, o algoritmo foi dividido nas operações básicas apresentadas simplificadamente a seguir, nas quais foi omitido o escalonamento de tarefas relacionadas ao mecanismo de monitoramento:

Escalonamento de saída de partição Caso haja uma partição em execução que, segundo a escala de partições, deve sair de execução no horário atual, coloca em execução a partição padrão do módulo;

Escalonamento de entrada de partição Caso não haja partição em execução e haja uma partição pronta que, segundo a escala de partições, deve entrar em execução no horário atual, coloca-a em execução;

Escalonamento de saída de processo Caso haja um processo em execução na partição que não é o processo pronto de maior prioridade, ou se não houver nenhum processo pronto, coloca em execução o processo padrão da partição atual;

Escalonamento de entrada de processo Caso não haja um pro-

cesso em execução na partição e existam processos prontos, coloca o processo pronto de maior prioridade atual em execução na partição atual;

Tratamento de eventos de processos Trata agendamentos de espera e a detecção de perdas de *deadline* dos processos da partição atual.

Apresenta-se no Trecho de Código 4.1 uma simplificação do algoritmo de escalonamento de tarefas empregado no SO desenvolvido, com o objetivo de clarificar a forma como essas operações básicas são implementadas e utilizadas para realização do escalonamento de partições e de processos. Essa simplificação omite diversos detalhes do algoritmo completo, porém inclui o escalonamento das tarefas do mecanismo de monitoramento exigido pela especificação ARINC 653, ou seja, dos processos tratadores de erros e *HM callbacks* das partições e do *HM callback* do módulo.

Trecho de Código 4.1 – Algoritmo de escalonamento

```
 1 Procedimento ESCALONAR_SAIDA_PARTICAO:
 2     Se o HM callback do módulo está em execução
 3         Se o HM callback do módulo deve ser parado
 4             Escalona a partição padrão do módulo
 5         Retorna
 6     Se não há uma partição em execução no módulo
 7         Retorna
 8     Se o HM callback do módulo deve ser iniciado
 9         Escalona a partição padrão do módulo
10         Retorna
11     Se não existe partição pronta no módulo
12         Escalona a partição padrão do módulo
13         Retorna
14     Se foi alcançado o fim da janela de tempo da partição atual
15         Remove a partição atual da fila de partições prontas
16         Prepara a próxima janela de tempo da partição atual
17         Insere a partição atual na fila de partições prontas
18         Escalona a partição padrão do módulo
19 Fim
20
21 Procedimento ESCALONAR_ENTRADA_PARTICAO:
22     Se há uma partição em execução no módulo
23         Retorna
24     Se o HM callback do módulo está em execução
25         Retorna
26     Se o HM callback do módulo deve ser iniciado
27         Escalona o HM callback do módulo
28         Retorna
29     Se há uma partição pronta cuja janela de tempo foi alcançada
30         Coloca a partição em execução
31 Fim
32
33 Procedimento TRATAR_EVENTOS_PROCESSOS:
34     Se não há uma partição em execução no módulo
35         Retorna
```

```
36     Para processos cujo tempo de espera foi atingido
37         Termina a espera do processo
38     Para processos cujo deadline foi atingido
39         Gera erro do tipo DEADLINE_MISSED
40 Fim
41
42 Procedimento ESCALONAR_SAIDA_PROCESSO:
43     Se não há uma partição em execução no módulo
44         Retorna
45     Se o HM callback da partição está em execução
46         Se o HM callback da partição deve ser parado
47             Escalona o processo padrão da partição
48         Retorna
49     Se o tratador de erros da partição está em execução
50         Se o tratador de erros da partição deve ser parado
51             Escalona o processo padrão da partição
52         Retorna
53     Se não há um processo em execução na partição
54         Retorna
55     Se o HM callback da partição deve ser iniciado
56         Escalona o processo padrão da partição
57         Retorna
58     Se o tratador de erros da partição deve ser iniciado
59         Escalona o processo padrão da partição
60         Retorna
61     Se o processo em execução não é o pronto de maior prioridade
62         Escalona o processo padrão da partição
63 Fim
64
65 Procedimento ESCALONAR_ENTRADA_PROCESSO:
66     Se não há uma partição em execução no módulo
67         Retorna
68     Se o HM callback da partição está em execução
69         Retorna
70     Se o HM callback da partição deve ser iniciado
71         Escalona o HM callback da partição
72         Retorna
73     Se o tratador de erros da partição está em execução
74         Retorna
75     Se o tratador de erros da partição deve ser iniciado
76         Escalona o tratador de erros da partição
77         Retorna
78     Se há um processo em execução na partição
79         Retorna
80     Se há processos prontos
81         Escalona o processo pronto de maior prioridade
82 Fim
83
84 Procedimento ESCALONADOR:
85     // Na partição atual
86     TRATAR_EVENTOS_PROCESSOS()
87     ESCALONAR_SAIDA_PROCESSO()
88     ESCALONAR_ENTRADA_PROCESSO()
89     ESCALONAR_SAIDA_PARTICAO()
90     // Se uma nova partição será executada
91     Se ESCALONAR_ENTRADA_PARTICAO()
92         // Na nova partição
93         TRATAR_EVENTOS_PROCESSOS()
94         ESCALONAR_SAIDA_PROCESSO()
95         ESCALONAR_ENTRADA_PROCESSO()
96 Fim
```

Apresenta-se a seguir os métodos definidos no núcleo do SO desenvolvido neste trabalho para utilização do algoritmo de escalonamento de tarefas:

SCHEDULER_SCHEDULEPARTITIONOUT Efetua o escalonamento de saída de partição;

SCHEDULER_SCHEDULEPARTITIONIN Efetua o escalonamento de entrada de partição;

SCHEDULER_STARTPARTITIONSCHEDULER Inicializa o escalonamento de partições, ou seja, preenche a fila de prioridades de partições prontas, inicializa o relógio do módulo, ativa a interrupção periódica de escalonamento e executa o escalonador pela primeira vez, ativando o primeiro contexto de execução (que poderá ser a partição padrão ou o processo padrão de uma das partições);

SCHEDULER_HANDLEPROCESSESEVENTS Efetua o tratamento de eventos de processos;

SCHEDULER_SCHEDULEPROCESSOUT Efetua o escalonamento de saída de processo;

SCHEDULER_SCHEDULEPROCESSIN Efetua o escalonamento de entrada de processo;

SCHEDULER_STARTPROCESSSCHEDULER Inicializa o escalonamento de processos na partição atual, ou seja, calcula os horários de liberação e *deadlines* dos processos que foram iniciados durante a inicialização da partição de acordo com as definições fornecidas pela especificação (ARINC, 2006a), e solicita a execução do algoritmo de escalonamento para eventual troca do contexto de execução de acordo com o novo cenário;

SCHEDULER Executa o algoritmo de escalonamento conforme apresentado no Trecho de Código 4.1;

TICK Atualiza o relógio do sistema de acordo com o intervalo da interrupção periódica, reinicializa a escala de partições caso o horário de fim do *major frame* tenha sido alcançado e, finalmente, executa o algoritmo de escalonamento.

O algoritmo de escalonamento de tarefas mantém uma variável global denominada **CURRENT_CONTEXT** que aponta a estrutura que armazena o contexto de execução que está sendo processado no momento ou que deverá ser retomado após a conclusão do algoritmo de escalonamento, ou seja, o contexto de execução é salvo nessa estrutura antes da execução do algoritmo e restaurado a partir

dela após sua conclusão. É utilizada ainda uma variável denominada **NEXT_CONTEXT** que, durante a execução do algoritmo de escalonamento, aponta a estrutura que armazena o contexto de execução que deve ser retomado após sua execução, ou apresenta valor nulo (**null**) caso o contexto de execução não deva ser alterado; quando o algoritmo de escalonamento é concluído, a variável **CURRENT_CONTEXT** assume o valor da variável **NEXT_CONTEXT**.

Essa abordagem de escalonamento tem como principal vantagem sua simplicidade de compreensão e implementação, além de exigir a utilização de uma única interrupção periódica de frequência ajustável, desde que observada a granularidade das escalas. Como desvantagem pode-se apontar o fato de que, com ela, o *jitter* causado pela desabilitação de interrupções afeta tanto o escalonamento de processos quanto o de partições, não violando porém o atendimento à especificação, já que essa admite a existência de *jitter*, desde que com mínimos efeitos no processo de escalonamento (ARINC, 2006a). Outras abordagens são possíveis para o escalonamento de partições e processos em SOs compatíveis com a ARINC 653, como por exemplo empregando recursos de virtualização de forma que cada partição seja executada em uma máquina virtual distinta, permitindo inclusive que sejam utilizados diferentes SOs nas partições (VANDERLEEST, 2010).

4.5 GERÊNCIA DE PROCESSADOR: ESTADOS DO SISTEMA

Cada um dos diferentes contextos de execução do SO desenvolvido neste trabalho representa também um ou mais dos possíveis estados do sistema, que são utilizados para definição do nível no qual os erros detectados devem ser tratados em função da tarefa que estava sendo executada no momento de sua ocorrência. Por exemplo, caso um erro seja gerado durante a inicialização de uma partição isso significa, necessariamente, que nenhum de seus processos está em execução, de forma que a partição toda encontra-se comprometida e não poderá ser executada. Em contrapartida, se um erro é causado durante a execução de um processo dessa mesma partição (ou seja, após sua inicialização ser concluída), os demais processos podem não ser afetados, continuando em execução após o tratamento do erro. Apresenta-se a seguir os estados do sistema utilizados pelo SO desenvolvido neste trabalho, associados aos identificadores atribuídos a cada um deles para utilização nos arquivos de configuração do módulo e na enumeração **SYSTEM_STATE_TYPE**, que é empregada no código do SO para

indicação desses estados.

***HM callback* do módulo (1)** Foi detectado um erro mapeado em configuração para o nível de módulo, e o *HM callback* do módulo foi disparado e está em execução;

Partição padrão do módulo (2) A partição padrão do módulo está sendo executada, ou seja, o módulo está em inicialização e o escalonamento de partições encontra-se desabilitado;

Partição ociosa do módulo (3) A inicialização do módulo foi concluída e a partição ociosa está sendo executada pois a janela de tempo atual não foi alocada a qualquer outra partição;

***HM callback* de partição (4)** Foi detectado um erro mapeado em configuração para o nível de partição, e o *HM callback* da partição foi disparado e está em execução;

Tratador de erros de partição (5) Foi detectado um erro mapeado em configuração para o nível de processo, e o processo tratador de erros da partição foi disparado e está em execução;

Processo padrão de partição (6) O processo padrão de uma partição está sendo executado, ou seja, uma partição está em inicialização e o escalonamento de partições encontra-se habilitado, porém o escalonamento de processos da partição atual encontra-se desabilitado;

Processo ocioso de partição(7) A inicialização de uma partição foi concluída e o processo ocioso dessa partição está sendo executado pois não há outro processo da partição pronto;

Processo (8) Um processo de uma partição está sendo executado;

Sistema operacional (9) Um serviço do SO foi invocado e está em execução.

4.6 GERÊNCIA DE MEMÓRIA: ALOCAÇÃO DINÂMICA

O tamanho de algumas informações manipuladas pelo SO desenvolvido neste trabalho pode variar de acordo com a configuração do sistema, ou até mesmo em função de parâmetros fornecidos através de serviços. A utilização de alocação estática de memória para armazenamento dessas informações obrigaria a reserva de regiões muito maiores que aquelas estritamente necessárias, ou exigiria um grande esforço de configuração para o dimensionamento adequado delas. Por esse motivo empregou-se neste trabalho regiões de memória de alocação dinâmica, nas quais segmentos de dados de tamanho variável podem ser reserva-

dos pelo núcleo do SO. São utilizadas regiões de alocação dinâmica de memória em dois diferentes escopos, que são apresentados a seguir:

De sistema É empregada apenas uma região deste tipo, que não pertence a qualquer partição e é utilizada para alocação de segmentos de dados destinados ao armazenamento de estruturas gerenciais relacionadas aos diversos elementos do sistema (contextos de execução, por exemplo);

De partição São empregadas diversas regiões deste tipo, cada uma pertencente exclusivamente a uma partição, que são utilizadas para armazenamento de informações gerais de seus elementos (como as mensagens manipuladas pelos mecanismos de comunicação interpartição e intrapartição, por exemplo).

Para manipulação das regiões de alocação dinâmica de memória é definida uma estrutura do tipo **heapRECORD**, que possui um apontador para o endereço inicial da região e campos que indicam seu tamanho total, seu tamanho já alocado e o alinhamento desejado para os endereços gerados (todos em *bytes*). Além disso os seguintes tipos de dados são definidos, mapeados aos tipos básicos associados (cuja definição pode ser consultada no Apêndice C):

heapPOINTER (portBYTE*) Utilizado para armazenar endereços de segmentos de memória alocados dinamicamente;

heapSIZE (portSIZE) Utilizado para indicar tamanhos de segmentos de memória alocados dinamicamente.

Apresenta-se, a seguir, uma breve descrição dos serviços disponibilizados ao núcleo do SO desenvolvido neste trabalho para a alocação dinâmica de memória:

HEAP_STARTUP Inicializa uma estrutura **heapRECORD**, definindo uma região de memória a ser utilizada para alocação dinâmica;

HEAP_ALLOCATE Realiza a alocação de um segmento de memória em uma região definida por uma estrutura **heapRECORD**; um ponteiro para a variável que armazenará o endereço do segmento alocado é fornecido a este serviço, e a alocação só é efetuada caso o valor dessa variável seja zero (**null**), sendo esse mantido inalterado em caso contrário, evitando assim sua realocação; os segmentos de memória alocados por este serviço têm todos seus *bytes* anulados (atribuídos com valor zero).

A utilização de regiões de alocação dinâmica sugere o não cumprimento às exigências da especificação ARINC 653 no que diz respeito à alocação de memória, já que segundo ela toda a memória utilizada pelo sistema deve ser reservada durante sua inicialização, e nunca deve ser liberada (ARINC, 2006a). Portanto, é fundamental ressaltar que, no SO desenvolvido neste trabalho, a memória das regiões de alocação dinâmica é reservada na primeira vez em que é solicitada alocação e não há realocação ou liberação em ocasiões posteriores, como seria o caso da reinicialização de partições. Ao invés disso, todos os ponteiros para memória alocada dinamicamente são mantidos inalterados nessas situações, conforme citado na descrição do método **HEAP_ALLOCATE**. É também fundamental ressaltar que, em decorrência dessa característica, não é possível a alocação de regiões maiores que aquelas inicialmente alocadas quando da reinicialização de quaisquer elementos do sistema.

4.7 GERÊNCIA DE MEMÓRIA: PROTEÇÃO DE MEMÓRIA

A principal função dos mecanismos de isolamento espacial (proteção de memória) em SOs compatíveis com a ARINC 653 é a detecção de tentativas de acesso a regiões de memória proibidas, geralmente causadas por erros de programação ou de configuração, permitindo que sejam disparados mecanismos para tratamento desses eventos de forma controlada (sem comprometimento do sistema) (ARINC, 2006a). Alguns tipos de erro de acesso à memória, como por exemplo estouros de pilha, podem ser detectados através de mecanismos implementados em *software*, conforme apresentado em (REAL TIME ENGINEERS LTD., 2014) e (OAR CORPORATION, 2014). Porém essas abordagens não oferecem garantias quanto à detecção desses erros, já que empregam verificações periódicas e, portanto, não são capazes de legitimar todas as instruções que acessam a memória. Soluções eficientes para proteção de memória empregam dispositivos de *hardware* dedicados a esse fim, como *MPU*s e *MMU*s, que permitem a especificação de permissões de acesso que são verificadas a cada instrução executada, garantindo assim a detecção de quaisquer acessos indevidos (ARM, 2012).

Conforme citado anteriormente, a utilização da *MMU* oferecida pelo processador empregado neste trabalho não é ideal para o tipo de sistema ao qual a ARINC 653 destina-se, sendo preferível nesse contexto o emprego de *MPU*s. Porém, uma vez que esse processador não

oferece uma *MPU*, empregou-se um subconjunto mínimo de recursos da *MMU* por ele oferecida para implementação da proteção de memória exigida pela especificação. Em outras palavras, a *MMU* do processador é utilizada pelo SO com tabelas de tradução construídas de forma que sejam gerados endereços físicos iguais aos endereços virtuais requisitados pelo *software*, tendo portanto como único efeito a aplicação de permissões de acesso.

Apresenta-se nas próximas seções a forma como a memória oferecida pela plataforma de *hardware* empregada é dividida em regiões para utilização pelo SO desenvolvido e as aplicações sobre ele executadas, assim como as permissões de acesso atribuídas a cada uma dessas regiões. É apresentada ainda a biblioteca desenvolvida para construção das tabelas de tradução e interação do núcleo do SO com a *MMU* do processador.

4.7.1 Regiões de memória

A separação da memória do sistema em regiões é fundamental para que seja possível o mapeamento de permissões de acesso de forma maximamente restritiva, permitindo o alcance de um isolamento espacial de partições compatível com as exigências da ARINC 653. A existência de algumas dessas regiões de memória decorre de características exigidas pela própria especificação, enquanto outras existem devido a decisões de projeto ou a características da plataforma de *hardware* empregada. Apresenta-se de forma sucinta na Tabela 2 as regiões de memória utilizadas pelo SO desenvolvido neste trabalho, atribuindo a cada uma delas uma sigla para referência resumida, e destaca-se ainda a cardinalidade dessas regiões, ou seja, quantas delas são alocadas no sistema, sendo Pa o conjunto de partições do módulo e $Pr(X)$ o conjunto de processos da partição X. Maiores detalhes sobre essas regiões de memória serão apresentados juntamente com o mapeamento de permissões a elas associado, que será abordado na próxima seção.

Tabela 2 – Regiões de memória

Sigla	Descrição	Cardinalidade
VECTBL	Tabela de vetores de exceção	1
PERPHR	Registradores de periféricos	1
SYS.STK	Pilha do sistema	1
SYS.COD	Código do sistema	1
SYS.DAT	Dados do sistema	1
SYS.HP	Alocação dinâmica do sistema	1
SYS.FLTT	Tabelas de tradução de primeiro nível	1
SYS.SLTT	Tabelas de tradução de segundo nível	1
MOD.COD	Código do módulo	1
MOD.DAT	Dados do módulo	1
MOD.HMC.STK	Pilha do *HM callback* do módulo	0 a 1
PAR.COD	Código de partição	$\|Pa\|$
PAR.DAT	Dados de partição	$\|Pa\|$
PAR.DAT.IMG	Imagem de dados de partição	$\|Pa\|$
PAR.HP	Alocação dinâmica de partição	$\|Pa\|$
PAR.DEF.STK	Pilha do processo padrão de partição	$\|Pa\|$
PAR.EH.STK	Pilha do tratador de erros de partição	0 a $\|Pa\|$
PAR.HMC.STK	Pilha do *HM callback* de partição	0 a $\|Pa\|$
PRO.STK	Pilha de processo	$\sum \|Pr(X \in Pa)\|$

A separação dessas regiões é efetivada de duas formas principais: através da geração de um arquivo de comandos para o *linker*, que indica a localização das regiões nas quais as informações processadas pelo compilador serão alojadas (código e dados, por exemplo), e através de estruturas de configuração, que indicam ao SO a localização das regiões utilizadas como pilha e para alocação dinâmica, por exemplo. O processo de definição dos endereços e tamanhos efetivos desses segmentos de memória é realizado pelas ferramentas de configuração fornecidas em conjunto com o SO desenvolvido, que serão apresentadas no Capítulo 5, e precisa levar em conta diversos critérios, dos quais destaca-se:

Configuração No arquivo de configuração do módulo é possível informar o endereço e/ou o tamanho desejado para as regiões de memória do sistema;

Alinhamento Algumas regiões de memória precisam que seus endereços e tamanhos estejam alinhados (sejam múltiplos de determinados valores) para o correto funcionamento do sistema;

Detecção de erros O funcionamento de alguns mecanismos de detecção de erros associados à proteção de memória depende que a alocação de regiões seja feita de forma adequada, como por exemplo a abordagem de diferenciação de erros de estouro de pilha e de

violação de memória utilizada no SO desenvolvido neste trabalho, que será apresentada na Seção 4.8.1;

Execução A forma como a imagem de memória do sistema é transportada ou construída para execução na plataforma de *hardware* pode estabelecer limitações na forma como as regiões de memória são dispostas; por exemplo, caso essa imagem seja transmitida através de um meio relativamente lento e grandes regiões vazias forem alocadas entre regiões não vazias, o tamanho da imagem pode tornar o processo de inicialização do sistema extremamente lento; a alocação de regiões de memória vazias (por exemplo pilhas, que são utilizadas apenas em tempo de execução) sempre ao final do mapa de memória é suficiente para o contorno desse tipo de limitação.

4.7.2 Mapeamento de permissões

Cada contexto de execução do SO está associado às regiões de memória apresentadas através de um mapeamento de permissões de acesso, que é empregado para a construção das tabelas de tradução utilizadas pela *MMU* sempre que o contexto de execução correspondente estiver sendo processado. Esse mapeamento define a que regiões de memória cada contexto de execução terá acesso, que operações poderão ser realizadas por eles sobre essas regiões (leitura, escrita e/ou execução), e ainda em que níveis de privilégio essas operações poderão ser executadas.

Uma vez que a especificação ARINC 653 exige o isolamento espacial a nível de partição, deduz-se que a construção de uma tabela de tradução por partição é suficiente para o mapeamento de todas as regiões de memória acessíveis pelos contextos de execução de cada partição. Porém, a detecção de determinados tipos de erro de acesso à memória, como a diferenciação entre violações de memória (acesso a endereços inválidos ou proibidos) e estouro de pilha (empilhamento ou desempilhamento de dados além dos limites da região de pilha), pode tornar-se complexa ou inviável se empregada apenas uma tabela de tradução por partição (ARINC, 2006a). Por esse motivo, assume-se neste trabalho que é construída uma tabela de tradução para cada contexto de execução do SO, o que gera *overheads* relacionados ao número relativamente grande de tabelas empregadas, porém beneficia a detecção de erros e facilita a compreensão dos mecanismos de proteção de memória, e está portanto de acordo com a orientação didática deste

trabalho.

Conforme citado anteriormente, a *MMU* do processador empregado possui dois registradores apontadores de tabelas de tradução, de forma que é possível a definição de uma tabela global, utilizada por todos os contextos de execução do sistema, e uma local, que varia de acordo com o contexto de execução atual. No SO desenvolvido neste trabalho, a tabela de tradução apontada pelo registrador **TTBR0** é global e mapeia o intervalo de endereços de memória 0x00000000 a 0x7FFFFFFF, fornecendo acesso irrestrito à região de registradores de periféricos do processador (**PERPHR**) e permitindo a leitura e execução somente em modo privilegiado para a região da tabela de vetores de exceção (**VECTBL**) (ARM, 2012). Por sua vez, a tabela de tradução apontada pelo registrador **TTBR1** é local e mapeia os demais endereços de memória, sendo construída de acordo com o contexto de execução pelo qual será utilizada.

As permissões utilizadas no mapeamento apresentado nesta seção são expressas no formato **PRV/NPR**[*], onde **PRV** e **NPR** são, respectivamente, as permissões para modo privilegiado e não privilegiado, e podem assumir os valores **NO** (sem permissão), **RO** (somente leitura) ou **RW** (leitura e escrita). O indicador opcional [*] denota que a execução de código é proibida na região em questão. Três diferentes permissões são utilizadas (padrão, partição e processo), e são aplicadas da seguinte forma:

1. Se o contexto de execução pertence ao mesmo processo ao qual a região de memória pertence e a permissão de processo não é vazia, é aplicada a permissão de **processo**;
2. Se a condição anterior não foi satisfeita, o contexto de execução pertence à mesma partição à qual a região de memória pertence e a permissão de partição não é vazia, é aplicada a permissão de **partição**;
3. Se as condições anteriores não foram satisfeitas e a permissão padrão não é vazia, é aplicada a permissão **padrão**;
4. Se nenhum caso anterior foi satisfeito, o acesso à região não é permitido.

Este mapeamento de permissões tem por objetivo atender às definições de isolamento espacial da especificação ARINC 653, que indica por exemplo que toda região de memória só pode ser escrita por contextos de execução pertencentes a no máximo uma das partições do módulo. A restrição de operações de leitura em nível semelhante a esse,

apesar de não ser exigida pela especificação, também foi tida como objetivo para atribuição de permissões (ARINC, 2006a). Apresenta-se nas seções a seguir detalhes sobre as regiões de memória utilizadas pelo SO, assim como o mapeamento de permissões atribuído a cada uma delas para cada contexto de execução do sistema e uma descrição justificada dessa atribuição.

4.7.2.1 Tabela de vetores de exceção (VECTBL)

Armazena a tabela de vetores de exceção que, na arquitetura *ARMv7*, consistem de instruções que desviam o fluxo de execução aos devidos procedimentos de tratamento quando da ocorrência de exceções, tais como *IRQs* e *Fast Interrupt Requests* (*FIQs*), entre outras. A tabela de vetores de exceção é copiada durante os primeiros estágios da inicialização do sistema para a memória *SRAM* interna ao núcleo do processador, e seus vetores apontam para tratadores implementados no núcleo do SO.

Exceções como a interrupção de escalonamento e erros de acesso à memória podem ocorrer a partir de qualquer contexto de execução, e quando da ocorrência de uma exceção o modo do processador é alterado para um modo privilegiado definido de acordo com a exceção disparada. Portanto, é suficiente para esta região a permissão de leitura e execução apenas em modo privilegiado para todos os contextos de execução. A permissão de execução é necessária, já que os vetores de exceção são instruções. Apresenta-se na Tabela 3 o mapeamento de permissões desta região.

Tabela 3 – Permissões para a região VECTBL

Contexto de execução	**Padrão**
MOD.DEF	RO/NO
MOD.HMC	RO/NO
PAR.DEF	RO/NO
PAR.EH	RO/NO
PAR.HMC	RO/NO
PRO	RO/NO

4.7.2.2 Registradores de periféricos (PERPHR)

Trata-se do intervalo de endereços de memória no qual são mapeados os registradores dos periféricos do processador, e portanto as permissões atribuídas a esta região definem quais contextos de execução serão capazes de utilizar esses periféricos de forma direta. Dada a orientação didática deste trabalho, optou-se por permitir o acesso a esses periféricos a partir de qualquer contexto de execução do sistema, e portanto esta região possui permissão de leitura e escrita em qualquer nível de privilégio a partir de todos os contextos de execução, sendo porém proibida a execução por tratar-se de endereços de registradores. Ressalta-se que essa abordagem **não** está de acordo com a especificação ARINC 653 no que se refere ao isolamento espacial de partições, já que com ela é possível que contextos de execução de partições diferentes efetuem operações de escrita nesta região (ARINC, 2006a). Porém, a fim de permitir que os periféricos do processador sejam facilmente explorados decidiu-se por deixar a cargo do usuário do SO o correto controle sobre os registradores de periféricos, ou seja, a garantia de que cada um deles é acessado por uma única partição. Apresenta-se na Tabela 4 o mapeamento de permissões desta região.

Tabela 4 – Permissões para a região PERPHR

Contexto de execução	**Padrão**
MOD.DEF	RW/RW*
MOD.HMC	RW/RW*
PAR.DEF	RW/RW*
PAR.EH	RW/RW*
PAR.HMC	RW/RW*
PRO	RW/RW*

4.7.2.3 Pilha do sistema (SYS.STK)

É a região de memória destinada à utilização como pilha durante a execução da partição padrão do módulo (contexto de execução a partir do qual o sistema é inicializado) e das rotinas de tratamento dos diversos tipos de exceção, sendo portanto necessário que seja subdividida de forma que possa servir a todos esses diferentes modos de execução. Apresenta-se na Tabela 5 a subdivisão aplicada sobre esta região, sendo o endereço de início de cada parte calculado a partir do endereço físico efetivo da região. Vale ressaltar que caso o tamanho

desta região, atualmente fixado em 16KB, seja alterado, é necessária a adequação do código de inicialização do processador para redimensionamento das sub-regiões.

Tabela 5 – Subdivisão da região SYS.STK

Modo de execução	**Início**	**Tamanho**
Undefined	0	3KB
Abort	3KB	3KB
IRQ	6KB	3KB
Supervisor	9KB	3KB
System/FIQ/User	12KB	4KB

Uma vez que exceções utilizam esta região como pilha e já que essas executam em modo privilegiado e podem ocorrer em qualquer contexto de execução, esta região possui permissão de leitura e escrita para todos os contextos de execução, porém apenas em modo privilegiado. Apenas para o contexto de execução da partição padrão do módulo, que conforme citado utiliza a pilha do sistema, é dada permissão de leitura e escrita em modo não privilegiado. Por tratar-se de uma região de pilha, a execução não é permitida. Apresenta-se na Tabela 6 o mapeamento de permissões desta região.

Tabela 6 – Permissões para a região SYS.STK

Contexto de execução	**Padrão**
MOD.DEF	RW/RW*
MOD.HMC	RW/NO*
PAR.DEF	RW/NO*
PAR.EH	RW/NO*
PAR.HMC	RW/NO*
PRO	RW/NO*

4.7.2.4 Código do sistema (SYS.COD)

Armazena o código do núcleo do SO, que deve ser acessível a partir de todos os contextos de execução já que é nele que encontra-se o código dos serviços fornecidos ao *software* de aplicação. Os serviços do SO são geralmente executados em modo privilegiado, porém na implementação realizada neste trabalho esses serviços são inicialmente executados em modo não privilegiado e, assim que iniciados, solicitam a transição para modo privilegiado através de uma chamada de super-

visor (interrupção de *software*), realizando a operação inversa quando do fim de sua execução. Em virtude disso, atribui-se a esta região permissões de leitura em qualquer nível de privilégio, já que o código da chamada de sistema que solicita a entrada em modo privilegiado encontra-se nela e executa em modo não privilegiado, e com permissão de execução por tratar-se de uma região de código. Apresenta-se na Tabela 7 o mapeamento de permissões desta região.

Tabela 7 – Permissões para a região SYS.COD

Contexto de execução	**Padrão**
MOD.DEF	RO/RO
MOD.HMC	RO/RO
PAR.DEF	RO/RO
PAR.EH	RO/RO
PAR.HMC	RO/RO
PRO	RO/RO

4.7.2.5 Dados do sistema (SYS.DAT)

É a região que armazena os dados do núcleo do SO, que são manipulados pelo código contido na região **SYS.COD**. Uma vez que o código do SO pode ser executado a partir de qualquer contexto de execução e considerando que todo código do núcleo do SO desenvolvido neste trabalho que manipula dados executa exclusivamente em modo privilegiado, atribui-se a esta região permissões de leitura e escrita apenas em modo privilegiado para todos os contextos de execução, porém com execução proibida por tratar-se de uma região de dados. Apresenta-se na Tabela 8 o mapeamento de permissões desta região.

Tabela 8 – Permissões para a região SYS.DAT

Contexto de execução	**Padrão**
MOD.DEF	RW/NO*
MOD.HMC	RW/NO*
PAR.DEF	RW/NO*
PAR.EH	RW/NO*
PAR.HMC	RW/NO*
PRO	RW/NO*

4.7.2.6 Alocação dinâmica do sistema (SYS.HP)

Utilizada para a alocação de segmentos de dados de tamanho variável utilizados para armazenamento de dados do núcleo do SO. Esta região equivale, do ponto de vista de permissões de acesso, à região de dados do SO (**SYS.DAT**), uma vez que destina-se ao armazenamento de dados de finalidade semelhante mas que são alocados dinamicamente (em tempo de execução). Apresenta-se na Tabela 9 o mapeamento de permissões desta região.

Tabela 9 – Permissões para a região SYS.HP

Contexto de execução	**Padrão**
MOD.DEF	RW/NO*
MOD.HMC	RW/NO*
PAR.DEF	RW/NO*
PAR.EH	RW/NO*
PAR.HMC	RW/NO*
PRO	RW/NO*

4.7.2.7 Tabelas de tradução de primeiro nível (SYS.FLTT)

É utilizada para alocação de tabelas de tradução de primeiro nível, que são empregadas para atribuição de permissões de acesso à memória através da utilização da *MMU* do processador. Uma vez que essas tabelas de tradução devem estar localizadas em endereços múltiplos de 16KB, o endereço inicial desta região deve possuir esse alinhamento, e já que essa é utilizada exclusivamente para alocação desse tipo de tabela – cujo tamanho também é 16KB –, a alocação de tabelas mantém o alinhamento necessário para novas alocações (ARM, 2012).

As tabelas de tradução são geralmente preparadas por serviços do SO invocados a partir da partição padrão do módulo (contexto de execução **MOD.DEF**) ou a partir do processo padrão da partição à qual pertencem (contexto de execução **PAR.DEF**), porém podem também ser preparadas a partir de processos (contextos de execução **PRO**) de partições de sistema em caso de necessidade. Sendo assim, é conferida a esses contextos de execução a permissão de leitura e escrita apenas em modo privilegiado, com execução proibida por tratar-se de tabelas de tradução. Apresenta-se na Tabela 10 o mapeamento de permissões desta região.

Tabela 10 – Permissões para a região SYS.FLTT

Contexto de execução	Padrão
MOD.DEF	RW/NO*
PAR.DEF	RW/NO*
PRO	RW/NO*

4.7.2.8 Tabelas de tradução de segundo nível (SYS.SLTT)

Região utilizada para alocação de tabelas de tradução de segundo nível. O mapeamento de permissões desta região equivale ao da região de tabelas de tradução de primeiro nível (**SYS.FLTT**), uma vez que ambas servem a finalidades semelhantes e são manipuladas a partir dos mesmos contextos de execução. O endereço inicial desta região deve estar alinhado a 1KB, já que as tabelas de tradução de segundo nível (cujo tamanho é 1KB) exigem esse alinhamento (ARM, 2012). Apresenta-se na Tabela 11 o mapeamento de permissões desta região.

Tabela 11 – Permissões para a região SYS.SLTT

Contexto de execução	Padrão
MOD.DEF	RW/NO*
PAR.DEF	RW/NO*
PRO	RW/NO*

4.7.2.9 Código do módulo (MOD.COD)

Armazena o código do módulo, que é composto pelo procedimento de inicialização do módulo (contexto de execução **MOD.DEF**) e pelo *HM callback* do módulo (contexto de execução **MOD.HMC**). Esses contextos de execução são, portanto, os únicos que têm permissão de acesso a esta região, e somente para leitura e execução em qualquer nível de privilégio. Apresenta-se na Tabela 12 o mapeamento de permissões desta região.

Tabela 12 – Permissões para a região MOD.COD

Contexto de execução	Padrão
MOD.DEF	RO/RO
MOD.HMC	RO/RO

4.7.2.10 Dados do módulo (MOD.DAT)

É a região que armazena os dados do módulo e as estruturas de configuração de todos os elementos do sistema, sendo portanto acessada pelo código do SO (contido na região **SYS.COD**) e pelo código do módulo (contido na região **MOD.COD**). Uma vez que o código do SO pode ser executado a partir de qualquer contexto de execução, todos eles têm por padrão, no mínimo, permissão para leitura e escrita em modo privilegiado. Por utilizarem esta região para armazenamento de dados e executarem em modo não privilegiado, os contextos de execução **MOD.DEF** e **MOD.HMC** possuem, ainda, permissão padrão de leitura e escrita em modo não privilegiado. Por tratar-se de uma região de dados, a execução é proibida. Apresenta-se na Tabela 13 o mapeamento de permissões desta região.

Tabela 13 – Permissões para a região MOD.DAT

Contexto de execução	**Padrão**
MOD.DEF	RW/RW*
MOD.HMC	RW/RW*
PAR.DEF	RW/NO*
PAR.EH	RW/NO*
PAR.HMC	RW/NO*
PRO	RW/NO*

4.7.2.11 Pilha do *HM callback* do módulo (MOD.HMC.STK)

Região utilizada como pilha pelo *HM callback* do módulo (contexto de execução **MOD.HMC**), que possui portanto permissão total sobre ela, porém com execução proibida por tratar-se de uma região de pilha. Apresenta-se na Tabela 14 o mapeamento de permissões desta região.

Tabela 14 – Permissões para a região MOD.HMC.STK

Contexto de execução	**Padrão**
MOD.HMC	RW/RW*

4.7.2.12 Código de partição (PAR.COD)

Armazena o código de uma determinada partição, que é composto por seu processo padrão (contexto de execução **PAR.DEF**), seus processos (contextos de execução **PRO**), seu processo tratador de erros (contexto de execução **PAR.EH**) e por seu *HM callback* (contexto de execução **PAR.HMC**). Esses são, portanto, os únicos contextos de execução que têm permissão de acesso a esta região, desde que pertencentes à partição à qual a região está associada, e somente para leitura e execução em qualquer nível de privilégio. Apresenta-se na Tabela 15 o mapeamento de permissões desta região.

Tabela 15 – Permissões para a região PAR.COD

Contexto de execução	**Partição**
PAR.DEF	RO/RO
PAR.EH	RO/RO
PAR.HMC	RO/RO
PRO	RO/RO

4.7.2.13 Dados de partição (PAR.DAT)

Armazena os dados de uma determinada partição e é portanto acessada pelo código dessa partição (contido na região **PAR.COD**), que compreende os contextos de execução **PAR.DEF**, **PAR.EH**, **PAR.HMC** e **PRO**. Esses contextos de execução têm, portanto, permissão de leitura e escrita em qualquer nível de privilégio, desde que pertencentes à partição à qual a região está associada. Por ser restaurada pelo SO a partir da região de imagem correspondente quando da reinicialização da partição, esta região possui ainda permissão padrão de leitura e escrita em modo privilegiado a partir de qualquer contexto de execução, porém sem permissão de execução por tratar-se de uma região de dados. Apresenta-se na Tabela 16 o mapeamento de permissões desta região.

Tabela 16 – Permissões para a região PAR.DAT

Contexto de execução	Padrão	Partição
MOD.DEF	RW/NO*	-
MOD.HMC	RW/NO*	-
PAR.DEF	RW/NO*	RW/RW*
PAR.EH	RW/NO*	RW/RW*
PAR.HMC	RW/NO*	RW/RW*
PRO	RW/NO*	RW/RW*

4.7.2.14 Imagem de dados de partição (PAR.DAT.IMG)

Região na qual é mantida a imagem (cópia) da região de dados da partição (**PAR.DAT**) no momento de sua criação, e que é utilizada durante a reinicialização da partição para restauração de seus dados originais. Uma vez que essas operações são executadas pelo núcleo do SO, é fornecida permissão de leitura e escrita para todos os contextos de execução, porém somente em modo privilegiado e sem permissão de execução por tratar-se de uma região de dados. Apresenta-se na Tabela 17 o mapeamento de permissões desta região.

Tabela 17 – Permissões para a região PAR.DAT.IMG

Contexto de execução	Padrão
MOD.DEF	RW/NO*
MOD.HMC	RW/NO*
PAR.DEF	RW/NO*
PAR.EH	RW/NO*
PAR.HMC	RW/NO*
PRO	RW/NO*

4.7.2.15 Alocação dinâmica de partição (PAR.HP)

Equivale à região de alocação dinâmica do sistema, porém é utilizada para armazenamento de informações pertencentes exclusivamente a uma determinada partição. Do ponto de vista de permissões de acesso equivale à região de dados de partição (**PAR.DAT**), uma vez que destina-se ao armazenamento de dados de finalidade semelhante mas que são alocados dinamicamente (em tempo de execução). Dessa forma, as permissões atribuídas são semelhantes às dessa região, porém sem quaisquer permissões em modo não privilegiado, já que não

destina-se à utilização pelo *software* de aplicação mas pelo núcleo do SO. Apresenta-se na Tabela 18 o mapeamento de permissões desta região.

Tabela 18 – Permissões para a região PAR.HP

Contexto de execução	**Partição**
PAR.DEF	RW/NO*
PAR.EH	RW/NO*
PAR.HMC	RW/NO*
PRO	RW/NO*

4.7.2.16 Pilha do processo padrão de partição (PAR.DEF.STK)

Utilizada como pilha pelo processo padrão de uma determinada partição (contexto de execução **PAR.DEF**), que possui portanto permissão total sobre ela, porém com execução proibida por tratar-se de uma região de pilha. Apresenta-se na Tabela 19 o mapeamento de permissões desta região.

Tabela 19 – Permissões para a região PAR.DEF.STK

Contexto de execução	**Partição**
PAR.DEF	RW/RW*

4.7.2.17 Pilha do tratador de erros de partição (PAR.EH.STK)

É utilizada como pilha pelo processo tratador de erros de uma determinada partição (contexto de execução **PAR.EH**), que possui portanto permissão total sobre ela, porém com execução proibida por tratar-se de uma região de pilha. Apresenta-se na Tabela 20 o mapeamento de permissões desta região.

Tabela 20 – Permissões para a região PAR.EH.STK

Contexto de execução	**Partição**
PAR.EH	RW/RW*

4.7.2.18 Pilha do *HM callback* de partição (PAR.HMC.STK)

Reservada para utilização como pilha pelo *HM callback* de uma determinada partição (contexto de execução **PAR.HMC**), que possui portanto permissão total sobre ela, porém com execução proibida por tratar-se de uma região de pilha. Apresenta-se na Tabela 21 o mapeamento de permissões desta região.

Tabela 21 – Permissões para a região PAR.HMC.STK

Contexto de execução	Partição
PAR.HMC	RW/RW*

4.7.2.19 Pilha de processo (PRO.STK)

É a região utilizada como pilha por um processo de uma determinada partição (contexto de execução **PRO**), que possui portanto permissão total sobre ela. A fim de viabilizar otimizações empregadas pelos mecanismos de comunicação intrapartição, especificamente *buffers*, nos quais a cópia de mensagens pode ocorrer diretamente entre variáveis locais (alocadas na pilha) de dois diferentes processos de uma mesma partição, o contexto de execução **PRO** dos demais processos da partição à qual o processo pertence possuem permissão para leitura e escrita nesta região, porém apenas em modo privilegiado e sem possibilidade de execução por tratar-se de uma região utilizada como pilha (ARINC, 2006a). Apresenta-se na Tabela 22 o mapeamento de permissões desta região.

Tabela 22 – Permissões para a região PRO.STK

Contexto de execução	Partição	Processo
PRO	RW/NO*	RW/RW*

4.7.3 Atribuição de permissões para depuração

A atribuição de *breakpoints* para depuração na plataforma de *hardware* utilizada neste trabalho é feita através de operações de escrita em endereços de memória nos quais localizam-se instruções (regiões de

código), e portanto a restrição dessa operação nessas regiões impede que os mecanismos de depuração funcionem corretamente. Para evitar prejuízos relacionados à utilização desse tipo de recurso, deve ser atribuída permissão total para todos os contextos de execução a todas as regiões de memória que contêm código. Essas permissões devem, preferencialmente, ser aplicadas apenas durante o processo de depuração do sistema, já que sua utilização causa perdas significativas no que se refere à detecção de erros de violação de memória.

4.7.4 Atribuição de permissões para partições de sistema

Partições de sistema possuem uma atribuição diferenciada de permissões, de forma que seja possível a partir delas uma máxima flexibilidade de operação sobre o sistema, mantendo-se porém seu isolamento temporal e espacial em relação às demais. As partições de sistema possuem as seguintes características especiais:

Execução em modo privilegiado Diferentemente de todos os contextos de execução das demais partições, os contextos de execução das partições de sistema executam sempre em modo privilegiado, possuindo portanto total controle sobre o processador, seus periféricos e suas configurações;

Acesso a regiões de dados Possuem permissão de leitura e escrita em todas as regiões de memória do sistema que contêm dados, inclusive naquelas utilizadas para alocação dinâmica, podendo portanto realizar a troca de informações entre partições.

4.7.5 Biblioteca da *MMU*

Com o objetivo de simplificar a interação com a *MMU* do processador utilizado, desenvolveu-se neste trabalho uma biblioteca que oferece métodos para a alocação de tabelas de tradução (tanto de primeiro quanto de segundo nível), preparação dos descritores nelas contidos e ainda para controle (habilitação e desabilitação, por exemplo) desse mecanismo. Apresenta-se a seguir os procedimentos oferecidos por essa biblioteca para a execução das tarefas mencionadas, categorizados pela função à qual servem.

- Manipulação de tabelas de tradução de primeiro nível:

MMU_FLTRANSLATIONTABLE_ALLOCATE Aloca uma tabela de tradução de primeiro nível e restaura todos seus descritores para descritores de falta – caso o ponteiro de tabela fornecido não tenha valor nulo, apenas restaura os descritores da tabela já alocada;

MMU_FLTRANSLATIONTABLE_MAPSECTION Mapeia um descritor de seção (1MB) a partir de um endereço alinhado a 1MB;

MMU_FLTRANSLATIONTABLE_MAPSECTIONS Mapeia descritores de seção (1MB) em uma faixa de endereços alinhados a 1MB;

MMU_FLTRANSLATIONTABLE_MAPSUPERSECTION Mapeia descritores de superseção (16MB) a partir de um endereço alinhado a 16MB;

MMU_FLTRANSLATIONTABLE_MAPSUPERSECTIONS Mapeia descritores de superseção (16MB) em uma faixa de endereços alinhados a 16MB;

MMU_FLTRANSLATIONTABLE_MAPSLTRANSLATIONTABLE Mapeia um descritor de tabela de tradução de segundo nível.

- Manipulação de tabelas de tradução de segundo nível:

MMU_SLTRANSLATIONTABLE_ALLOCATE Aloca uma tabela de tradução de segundo nível e restaura todos seus descritores para descritores de falta – caso o ponteiro de tabela fornecido não tenha valor nulo, apenas restaura os descritores da tabela já alocada;

MMU_SLTRANSLATIONTABLE_MAPSMALLPAGE Mapeia um descritor de página pequena (4KB) a partir de um endereço alinhado a 4KB;

MMU_SLTRANSLATIONTABLE_MAPSMALLPAGES Mapeia descritores de página pequena (4KB) em uma faixa de endereços alinhados a 4KB;

MMU_SLTRANSLATIONTABLE_MAPLARGEPAGE Mapeia descritores de página grande (64KB) a partir de um endereço alinhado a 64KB;

MMU_SLTRANSLATIONTABLE_MAPLARGEPAGES Mapeia descritores de página grande (64KB) em uma faixa de endereços alinhados a 64KB.

- Construção de tabelas de tradução:

MMU_MAPREGION Mapeia uma região de memória demarcada por endereços de início e de fim obrigatoriamente alinhados a 4KB, empregando os procedimentos de manipulação de tabelas de tradução para mapear seções, superseções, páginas grandes e páginas pequenas de forma a minimizar o número de descritores utilizados no mapeamento da região.

- Gerenciamento da *MMU*:

MMU_STARTUP Inicializa a biblioteca, garantindo que a *MMU* encontra-se desabilitada e definindo os endereços e tamanhos das regiões de memória utilizadas para a alocação das tabelas de tradução de primeiro e de segundo nível;

MMU_ENABLE Habilita a *MMU*, atribuindo o valor de **TTBCR.N**, dos registradores apontadores de tabelas de tradução (**TTBR0** e **TTBR1**) e ainda o valor inicial do registrador **CONTEXTIDR**.

4.8 TRATAMENTO DE ERROS

É fundamental que, quando da utilização de um SO, as exceções geradas pelos diversos recursos de proteção oferecidos pelo processador sejam tratadas de forma a garantir a não propagação de faltas entre tarefas, ou seja, de forma que erros causados por uma delas não afetem a execução das demais. Por isso, a ARINC 653 exige a implementação de um conjunto de mecanismos de monitoramento e tratamento de erros (*health monitoring*), que permitem a contenção de falhas de forma que não seja violado o escalonamento de partições (ARINC, 2006a).

No SO desenvolvido neste trabalho, a enumeração **ERROR_IDENTIFIER_TYPE** é utilizada para identificação dos erros detectados pelo mecanismo de monitoramento e tratamento de erros. Nessa enumeração, os valores entre 1 e 99 são utilizados para erros comuns (geralmente detectáveis por qualquer plataforma de *hardware*), enquanto valores maiores ou iguais a 100 são reservados para identificação de erros específicos da plataforma de *hardware* empregada. Os identificadores de erro definidos por essa enumeração são prefixados por **ERRORIDENTIFIER_**, resumido como **EI_**, e são apresentados a seguir associados aos valores atribuídos a cada um deles:

- Erros comuns:

EI_DEADLINEMISSED (1) Relacionado ao código de erro **DEADLINE_MISSED**, indica perdas de *deadline* de processos;

EI_APPLICATIONERROR (2) Relacionado ao código de erro **APPLICATION_ERROR**, indica erros lançados através do serviço **RAISE_APPLICATION_ERROR**;

EI_NUMERICERROR (3) Relacionado ao código de erro **NUMERIC_ERROR**, indica erros originados em operações numéricas;

EI_ILLEGALREQUEST (4) Relacionado ao código de erro **ILLEGAL_REQUEST**, indica chamadas ilegais a serviços do SO;

EI_STACKOVERFLOW (5) Relacionado ao código de erro **STACK_OVERFLOW**, indica tentativa de acesso além dos limites de regiões de memória utilizadas como pilha;

EI_MEMORYVIOLATION (6) Relacionado ao código de erro **MEMORY_VIOLATION**, indica tentativas de acesso a endereços de memória não permitidos;

EI_HARDWAREFAULT (7) Relacionado ao código de erro **HARDWARE_FAULT**, indica erros originados na plataforma de *hardware*;

EI_POWERFAIL (8) Relacionado ao código de erro **POWER_FAIL**, indica erros decorrentes de falhas no suprimento de energia.

- Erros específicos para processadores *TI* AM335X:

EI_TRANSLATIONFAULT (100) Relacionado ao código de erro **MEMORY_VIOLATION**, indica erros de tradução de endereços pela *MMU*;

EI_MEMORYALIGNMENTFAULT (101) Relacionado ao código de erro **MEMORY_VIOLATION**, indica erros de alinhamento de memória;

EI_MEMORYACCESSFAULT (102) Relacionado ao código de erro **MEMORY_VIOLATION**, indica erros de acesso à memória;

EI_CACHEFAULT (103) Relacionado ao código de erro **MEMORY_VIOLATION**, indica erros relacionados à memória *cache*;

EI_UNDEFINEDINSTRUCTION (104) Relacionado ao código de erro **MEMORY_VIOLATION**, indica tentativas de execução de instrução inválida.

Apresenta-se nas seções a seguir detalhes das abordagens empregadas para detecção dos principais tipos de erro previstos pela ARINC 653, além de considerações sobre características específicas de seu tratamento no SO desenvolvido.

4.8.1 Erros de memória

Um ponto fundamental da utilização de mecanismos de proteção de memória em SOs é o correto tratamento dos erros gerados quando da ocorrência de acessos indevidos, efetivando assim o isolamento espacial de suas tarefas. Os seguintes erros de acesso à memória são tratados pelo SO desenvolvido neste trabalho de acordo com as definições da especificação ARINC 653 (ARINC, 2006a):

Estouro de pilha Ocorre quando um contexto de execução empilha (adiciona) elementos além da capacidade da região de memória que utiliza como pilha (*stack overflow*), ou quando desempilha (remove) mais elementos do que foram previamente adicionados a essa região (*stack underflow*); isso pode ocorrer em contextos de execução que alcançam uma grande profundidade de chamadas de métodos ou que invocam procedimentos que alocam grandes volumes de dados em parâmetros ou em variáveis locais (que são geralmente armazenados na pilha), por exemplo;

Violação de memória Ocorre quando um endereço de memória não permitido é acessado, o que pode ocorrer quando um procedimento de uma tarefa acessa um ponteiro nulo (**null**) ou uma região de memória pertencente a outra tarefa, por exemplo.

Do ponto de vista da detecção desses erros, ambos geram um mesmo tipo de exceção no processador empregado neste trabalho (de acesso a dados), e portanto é necessária a utilização de uma abordagem que permita sua distinção com base nas informações disponibilizadas pela *MMU*. Entre essas informações destaca-se o endereço de memória cuja tentativa de acesso causou a exceção, a operação que estava sendo executada quando de sua ocorrência (leitura ou escrita) e o código do erro detectado (específico da arquitetura).

A estratégia de distinção empregada neste trabalho exige a garantia de que as regiões de memória que precedem e sucedem a região de

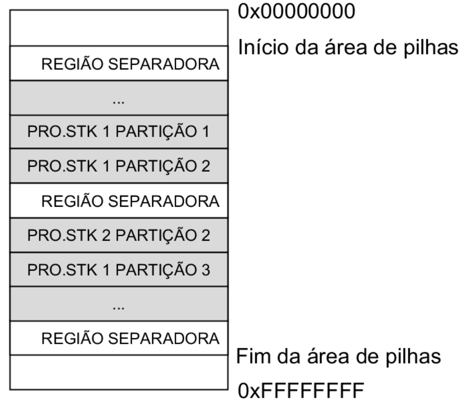
pilha utilizada por um determinado contexto de execução não possam ser acessadas por ele, gerando assim exceções em caso de tentativas de acesso a endereços próximos porém externos a essa região. Conforme pode ser observado no mapeamento de permissões apresentado anteriormente, as regiões utilizadas como pilha são geralmente acessadas exclusivamente pelo contexto de execução ao qual pertencem, de forma que essa garantia pode ser obtida parcialmente através da alocação dessas regiões de forma contígua. Porém, observa-se que essa característica não se sustenta nos seguintes casos:

- Se duas regiões de pilha de processo (**PRO.STK**) pertencentes à mesma partição forem alocadas de forma contígua – conforme apresentado na Tabela 22 –, essas regiões são acessíveis em modo privilegiado a partir dos demais processos da mesma partição;
- Se a primeira região de pilha for antecedida por uma região acessível pelo contexto de execução ao qual a pilha pertence;
- Se a última região de pilha for sucedida por uma região acessível pelo contexto de execução ao qual a pilha pertence.

Nesses casos é possível que ocorram estouros de pilha que não serão detectados, e que podem inclusive levar à corrupção dos dados contidos nas pilhas de outros contextos de execução do sistema. Para tratamento dessas condições é realizada a alocação de um conjunto de regiões separadoras que não são acessíveis por nenhum contexto de execução: uma antes da primeira região de pilha, uma após a última delas e uma entre todo par de regiões utilizadas como pilha de processo (**PRO.STK**) pertencentes à mesma partição e alocadas contiguamente. Essas regiões separadoras devem ser pequenas o suficiente para minimizar o desperdício de memória, porém devem também ser suficientemente grandes para que estouros gerados durante operações de pilha longas (alocação de vetores, por exemplo) possam ser detectadas. Neste trabalho foram utilizadas regiões separadoras de 4KB, por serem as menores possíveis frente à *MMU* empregada e, ainda assim, suficientes para a detecção de estouros de pilha em operações relativamente longas. Dessa forma garante-se que quaisquer duas regiões de pilha alocadas contiguamente não são acessíveis por um mesmo contexto de execução, eliminando assim a possibilidade de ocorrência de erros de estouro de pilha não detectados quando da execução de operações usuais sobre esse tipo de região de memória (empilhamento/desempilhamento de pequenas porções de dados). Uma vez que em modo não privilegiado toda região de pilha é acessível apenas por seu contexto de execução proprietário, garante-se ainda que nesse modo, isso é, durante

a execução do *software* de aplicação, todo erro de estouro de pilha será detectado. Ilustra-se na Figura 8 a alocação das regiões de pilha e de regiões separadoras utilizada pelo SO desenvolvido neste trabalho.

Figura 8 – Alocação das regiões de pilha



Uma vez aplicada essa alocação de regiões de memória e em posse do endereço de memória cuja tentativa de acesso gerou uma exceção de acesso a dados (fornecido pela *MMU*), basta que esse endereço seja comparado com os endereços de início e de fim da área de memória que contém todas as pilhas do sistema (indicados na Figura 8): se o endereço cujo acesso gerou a exceção encontra-se entre o endereço de início e o endereço de fim da área de pilhas, considera-se que o erro foi um estouro de pilha (**STACK_OVERFLOW**), e em caso contrário o erro é considerado uma violação de memória (**MEMORY_VIOLATION**) (ARINC, 2006a).

4.8.2 Perda de *deadline*

Conforme apresentado anteriormente, em SOs compatíveis com a especificação ARINC 653 devem ser gerados erros do tipo **DEADLINE_MISSED** quando um processo excede a capacidade de tempo a ele alocada (perde seu *deadline*). Essa capacidade é alocada a partir do momento em que o processo é liberado, e é realocada sempre que o processo aguarda o período (no caso de processos periódicos), é suspenso e continuado (no caso de processos

aperiódicos), é parado e reiniciado, ou solicita a postergação de seu *deadline* através do serviço **REPLENISH** (ARINC, 2006a).

A detecção de perdas de *deadline* no SO desenvolvido neste trabalho é realizada através de verificações periódicas, utilizando para isso uma fila de prioridades de ordenação ascendente cujo atributo de ordenação é o horário de *deadline* e cujo valor é o identificador do processo ao qual cada entrada se refere. O algoritmo de detecção, executado a cada milissegundo juntamente com o algoritmo de escalonamento, compara o horário atual do módulo com o *deadline* do processo localizado na cabeceira dessa fila de prioridades, gerando um erro do tipo **DEADLINE_MISSED** caso esse tenha sido excedido.

4.8.3 Erros numéricos

Coprocessadores aritméticos de ponto flutuante geralmente oferecem, além das funcionalidades relacionadas a operações numéricas, abordagens para tratamento de erros decorrente dessas operações – como a divisão por zero e o estouro (*overflow*), por exemplo. Perante a ARINC 653, a estratégia ideal para tratamento desse tipo de erro é através da geração de exceções, na qual o fluxo de execução é desviado para um procedimento de tratamento implementado pelo SO quando de sua ocorrência, o qual gera erros da classe **NUMERIC_ERROR** e dispara os mecanismos de tratamento adequados (ARM, 2012), (ARINC, 2006a).

Conforme citado anteriormente, é utilizado neste trabalho o coprocessador aritmético de ponto flutuante oferecido pelo núcleo *ARM*® Cortex™-A8, baseado na arquitetura *ARM® VFPv3*, que não suporta a geração de exceções para tratamento de erros numéricos. Ao invés disso, esse coprocessador oferece um conjunto de configurações que permitem determinar o comportamento da unidade quando da ocorrência de erros, efetuando o arredondamento ou atribuindo um determinado valor ao resultado do cálculo afetado, por exemplo (ARM, 2010). Essa característica inviabiliza a detecção e tratamento de erros numéricos pelo SO desenvolvido neste trabalho através dos mecanismos previstos pela especificação ARINC 653.

4.8.4 Tratamento de erros a nível de sistema

A especificação ARINC 653 deixa a cargo do desenvolvedor do SO a elaboração dos mecanismos empregados para tratamento de erros no nível de sistema (ARINC, 2006a). No SO desenvolvido neste trabalho, são considerados erros de sistema aqueles ocorridos durante a execução do *HM callback* do módulo que, segundo a configuração do mecanismo de tratamento de erros para esse nível, devem ser ignorados. Essa característica se justifica pela ausência de mecanismos de tratamento de erros em nível superior ao do *HM callback* do módulo, não podendo portanto a execução do sistema ser mantida caso um erro ocorra nesse contexto de execução. Para tratamento, foi adicionada ao esquema de configuração uma nova seção, que define as ações a serem tomadas pelo mecanismo de monitoramento quando erros precisam ser tratados no nível de sistema. As únicas ações possíveis para tratamento de erros nesse nível são o desligamento ou a reinicialização do módulo.

4.9 CONFIGURAÇÃO

Esta seção tem por objetivo apresentar a forma como os arquivos *XML* de configuração definidos pela especificação ARINC 653 são empregados em conjunto com o SO desenvolvido neste trabalho, abordando as extensões do esquema básico que foram utilizadas, algumas convenções adotadas a fim de contornar limitações desse esquema, e ainda a forma como as informações de configuração são traduzidas para serem acessadas em tempo de execução pelo SO.

4.9.1 Extensão do esquema de configuração

Conforme citado anteriormente, o esquema básico dos arquivos de configuração de módulo definidos pela especificação ARINC 653 permite a incorporação de informações adicionais, sendo para isso empregados esquemas secundários oferecidos pelo esquema básico em todas suas principais entidades. Esses esquemas secundários podem ser utilizados pelo desenvolvedor de SOs compatíveis com a especificação para a introdução de parâmetros específicos, e devem necessariamente utilizar os espaços de nomes (*namespaces*) reservados pelo esquema básico a essa finalidade (ARINC, 2006a).

Neste trabalho, o esquema de arquivos de configuração foi expan-

dido com o objetivo de comportar as especificidades da implementação realizada e, além disso, permitir a geração de código para modelos completos de módulos ARINC 653 com base nos arquivos de configuração correspondentes. A geração automatizada de código facilita a preparação de uma estrutura básica para o *software* de aplicação, alinhando-se assim aos objetivos didáticos almejados. Apresenta-se nas seções a seguir as extensões criadas, associadas aos *namespaces* aos quais pertencem, assim como os atributos definidos para cada uma delas.

4.9.1.1 SysHM_Ext:Extension

Entidade que estende as definições relacionadas à função de monitoramento do SO para o nível de sistema, e deve portanto estar contida na entidade **System_HM_Table**. Possui o atributo opcional **DisableHealthMonitoring**, que permite a desabilitação completa da função de monitoramento e tratamento de erros (para fins de teste) através da supressão da geração do código que serve a essa função. Contém ainda entradas **SysHM_Ext:Error_ID_Action**, cujo atributo **Action** define a ação a ser tomada quando da necessidade de tratamento de um determinado erro no nível de sistema. Esse erro é identificado pelo atributo **ErrorIdentifier** e descrito pelo atributo **Description** (opcional) dessas entradas, e as possíveis ações são o desligamento (**SHUTDOWN**) ou a reinicialização (**RESET**) do módulo.

4.9.1.2 Mod_HM_Ext:Extension

Extensão das definições da função de monitoramento do SO para o nível do módulo, que deve ser utilizada dentro da entidade **Module_HM_Table**. Possui o atributo opcional **ModuleCallbackStackSize**, que define o tamanho da pilha do *HM callback* do módulo **em palavras**, sendo utilizado o tamanho padrão caso omitido.

4.9.1.3 PortExt:Extension

Extensão da definição de portas (elementos de comunicação interpartição) que é requerida apenas para portas de enfileiramento, ou seja, deve estar contida apenas em entidades **Queuing_Port** perten-

centes a entidades **Partition**. Possui o atributo **QueuingDiscipline**, que define a política de enfileiramento (*FIFO* ou por prioridade) a ser utilizada para os processos bloqueados em uma determinada porta de enfileiramento, e é utilizado para a geração automatizada do código de inicialização desse tipo de elemento.

4.9.1.4 Proc_Ext:Extension

Entidade que estende a definição de processos, devendo portanto estar contida em entidades **Process** pertencentes a entidades **Partition**. Possui os seguintes atributos, utilizados principalmente para fins de geração de código:

EntryPoint Define o nome do método que implementa o processo, sendo utilizado o nome padrão caso suprimido;

BasePriority Define a prioridade base do processo (obrigatório);

PeriodSeconds Período do processo em segundos caso esse seja periódico, sendo considerado aperiódico caso o período seja suprimido;

TimeCapacitySeconds Capacidade de tempo do processo em segundos, que é considerada infinita caso suprimida;

Deadline Atributo obrigatório que define o tipo de *deadline* do processo (**SOFT** ou **HARD**);

StartDelay Atraso opcional para a primeira liberação do processo durante a inicialização da partição, considerado zero se omitido;

SuspensionTimeSeconds Tempo de suspensão para processo aperiódico, considerado infinito se omitido;

SuppressStart Permite a supressão da geração do código de disparo do processo (chamada aos serviços **START** ou **DELAYED_START**) durante a inicialização da partição;

SuppressMainLoop Para fins de teste, permite a supressão da geração do código do laço principal do processo;

SuppressSuspension Para fins de teste, permite a supressão da geração do código de suspensão do processo (chamada aos serviços **PERIODIC_WAIT** ou **SUSPEND_SELF**).

4.9.1.5 PartitionExt:Extension

Utilizada dentro de entidades **Partition** para estender as definições relacionadas a uma determinada partição. Oferece os seguintes atributos, que atuam principalmente sobre o processo de geração de código:

StackSize Define o tamanho da pilha do processo padrão da partição **em palavras**, sendo utilizado o tamanho padrão caso omitido;

ErrorHandler Nome do método que implementa o processo tratador de erros da partição (caso exista);

ErrorHandlerStackSize Define o tamanho da pilha do processo tratador de erros da partição **em palavras**, sendo utilizado o tamanho padrão caso omitido;

SuppressPartitionIdleProcessMainLoop Para fins de teste, permite a supressão da geração de código do laço principal do processo ocioso da partição;

SuppressErrorHandlerStopCall Para fins de teste, permite a supressão da geração de código da chamada de parada do processo tratador de erros da partição;

MaximumErrorStatusCount Define o número máximo de entradas de erro que poderão ser geradas para o tratador de erros da partição, para fins de alocação de recursos (utilizado valor padrão igual a duas vezes o número de processos se omitido);

MaximumBufferCount Define o número máximo de *buffers* utilizados pela partição, para fins de alocação de recursos (utilizado valor padrão zero se omitido);

MaximumBlackboardCount Define o número máximo de *blackboards* utilizadas pela partição, para fins de alocação de recursos (utilizado valor padrão zero se omitido);

MaximumSemaphoreCount Define o número máximo de semáforos utilizados pela partição, para fins de alocação de recursos (utilizado valor padrão zero se omitido);

MaximumEventCount Define o número máximo de eventos utilizados pela partição, para fins de alocação de recursos (utilizado valor padrão zero se omitido);

PredefinedSystemPartition Indica que o código fonte de uma determinada partição de sistema é predefinido, ou seja, faz parte do núcleo do SO e, portanto, não deve ser gerado.

4.9.1.6 Part_HM_Ext:Extension

Entidade para extensão da definição da função de monitoramento do SO para o nível de partição, utilizada em entradas **Partition_HM_Table**. Possui o atributo opcional **PartitionCallbackStackSize**, que define o tamanho da pilha do *HM callback* de partição **em palavras**, sendo utilizado o tamanho padrão caso omitido.

4.9.1.7 ModExt:Extension

Entidade que estende as definições do módulo, e que deve ser utilizada dentro da entidade raiz do arquivo de configuração (**ARINC_653_Module**). Possui o atributo obrigatório **DefaultStackSize**, que define o tamanho padrão a ser utilizado para as regiões de memória de pilha (em palavras), e o atributo opcional **SuppressModuleIdlePartitionMainLoop** que, para fins de teste, permite a supressão da geração de código do laço principal da partição ociosa do módulo.

4.9.1.8 ModExt:Extension_AM335X

Assim como a entidade **ModExt:Extension**, estende as definições do módulo e deve, portanto, ser utilizada dentro da entidade raiz do arquivo de configuração. Porém, esta possui configurações específicas para o SO quando executado sobre processadores *TI* AM335X, sendo obrigatória sua utilização nesse contexto. Os seguintes pares de atributos desta entidade definem endereços e tamanhos (em *bytes*) das regiões de memória alocadas durante o processo de geração de código:

RAMMemoryPhysicalAddress/SizeBytes Ambos obrigatórios, definem o endereço e o tamanho da memória *RAM* disponível para o módulo – definidos de acordo com a plataforma de *hardware* utilizada;

SystemBootRegionPhysicalAddress/SizeBytes Endereço (opcional) e tamanho (obrigatório) da região de memória do código de inicialização do sistema;

SystemStackRegionPhysicalAddress/SizeBytes Endereço (opcional) e tamanho (obrigatório) da região de memória de

pilha do sistema;

SystemCodeRegionPhysicalAddress/SizeBytes Endereço (opcional) e tamanho (obrigatório) da região de memória de código do sistema;

SystemDataRegionPhysicalAddress/SizeBytes Endereço (opcional) e tamanho (obrigatório) da região de memória de dados do sistema;

SystemHeapRegionPhysicalAddress/SizeBytes Endereço (opcional) e tamanho (obrigatório) da região de memória para alocação dinâmica do sistema;

SystemFLTranslationTableRegionPhysicalAddress/SizeBytes Endereço (opcional) e tamanho (obrigatório) da região de memória de tabelas de tradução de primeiro nível do sistema;

SystemSLTranslationTableRegionPhysicalAddress/SizeBytes Endereço (opcional) e tamanho (obrigatório) da região de memória de tabelas de tradução de segundo nível do sistema;

ModuleCodeRegionPhysicalAddress/SizeBytes Endereço (opcional) e tamanho (obrigatório) da região de memória de código do módulo;

ModuleDataRegionPhysicalAddress/SizeBytes Endereço (opcional) e tamanho (obrigatório) da região de memória de dados do módulo.

A entidade em questão possui ainda o atributo obrigatório **ModuleNetworkPhysicalAddress**, que define o endereço físico de rede do módulo e consiste em seus endereços *MAC* e *IP* concatenados e convertidos para hexadecimal. Por exemplo, para atribuição dos endereços *MAC* 00:00:01:01:02:02 e *IP* 1.2.3.4 a um determinado módulo, o valor do atributo em questão deve ser 0x000001010202201020304.

4.9.2 Convenções

O esquema básico dos arquivos de configuração de módulo definido pela ARINC 653 exige que toda entrada **Connection_Table** possua no mínimo uma entrada **Channel**, mesmo que o módulo não empregue qualquer canal de comunicação. O mesmo ocorre para as entidades **Module_HM_Table** e **Partition_HM_Table**, nas quais o esquema exige que exista no mínimo uma entrada **System_State_Entry** mesmo que nenhum erro seja mapeado aos níveis de módulo e de partição, res-

pectivamente. Essa característica acaba por gerar erros quando da validação de arquivos de configuração frente ao esquema básico, inviabilizando sua utilização mesmo que esses estejam semanticamente corretos. Por esse motivo, as ferramentas de configuração desenvolvidas neste trabalho permitem a utilização de entradas **Channel** com atributo **ChannelIdentifier** igual a -1 e **System_State_Entry** com atributo **SystemState** igual a -1 nas entidades citadas. Essas entradas são ignoradas pelas ferramentas de configuração, permitindo assim que os arquivos de configuração de módulo utilizados em conjunto com o SO desenvolvido neste trabalho contenham apenas os elementos que são estritamente necessários aos objetivos pretendidos.

4.9.3 Tradução

Para que possam ser acessadas em tempo de execução pelo SO desenvolvido neste trabalho, as informações fornecidas através de arquivos de configuração de módulo no formato *XML* precisam ser traduzidas para definições de estruturas na linguagem C. Essa tradução é realizada automaticamente pelas ferramentas de configuração fornecidas em conjunto com o SO, e são armazenadas nos arquivos **configuration.c** e **configuration.h** que são compilados junto ao sistema. Nesses arquivos são definidas as estruturas globais **SYSTEM_CONFIGURATION**, do tipo **SYSTEM_CONFIGURATION_TYPE**, e **MODULE_CONFIGURATION**, do tipo **MODULE_CONFIGURATION_TYPE**, que armazenam a configuração do sistema e do módulo (inclusive de suas partições e processos), respectivamente. Apresenta-se a seguir as estruturas de configuração utilizadas pelo SO, categorizadas de acordo com os elementos aos quais servem.

4.9.3.1 Sistema

São estruturas cujo nome possui o prefixo **SYSTEM_CONFIGURATION_** (resumido por **S_C_**), destinadas ao armazenamento das configurações globais do sistema, e são apresentadas a seguir:

S_C_TYPE Armazena as seguintes configurações do sistema:

- Endereços e tamanhos efetivos das regiões de memória de pilha, código, dados e alocação dinâmica do sistema;

- Uma entrada do tipo **S_C_HEALTHMONITORING_TYPE**, que aponta a árvore de estruturas de nome prefixado por **S_C_HEALTHMONITORING_** (resumido por **S_C_HM_**) apresentada a seguir;
- Um apontador genérico **PORT_SYSTEM_CONFIGURATION** para configurações do sistema específicas da plataforma de *hardware* – para a plataforma empregada neste trabalho esse campo aponta uma estrutura do tipo **PORT_SYSTEM_CONFIGURATION_TYPE**, que indica os endereços e tamanhos efetivos de regiões de memória do sistema específicas do processador utilizado (tabela de vetores de exceção, registradores de periféricos, código de inicialização e de tabelas de tradução).

S_C_HM_TYPE Armazena as informações de configuração do mecanismo de monitoramento do SO para o nível de sistema – contém um conjunto de entradas **S_C_HM_ERROR_TYPE**, sendo que cada uma mapeia um identificador de erro (do tipo **ERROR_IDENTIFIER_TYPE**) à ação (enumeração **S_C_HM_ERROR_ACTION_TYPE**) a ser executada pelo SO quando de seu tratamento no nível de sistema, que pode ser parar o módulo ou reiniciá-lo.

4.9.3.2 Módulo

Trata-se de estruturas de nome prefixado por **MODULE_CONFIGURATION_** (resumido por **M_C_**), que destinam-se ao armazenamento das configurações globais do módulo e são apresentadas a seguir:

M_C_TYPE Armazena as seguintes configurações do módulo:

- O número máximo de partições suportadas pelo módulo (para alocação de recursos);
- Os endereços e tamanhos efetivos das regiões de memória do módulo;
- A duração do *major frame* da escala de partições;
- Uma entrada do tipo **M_C_CONNECTIONTABLE_TYPE**, que aponta a árvore de estruturas de nome prefixado por **M_C_CONNECTIONTABLE_** (resumido por **M_C_CT_**) apresentada a seguir;

- Uma entrada do tipo **M_C_HEALTHMONITORING_TYPE**, que aponta a árvore de estruturas de nome prefixado por **M_C_HEALTHMONITORING_** (resumido por **M_C_HM_**) apresentada a seguir;
- Uma entrada do tipo **PARTITION_CONFIGURATION_TYPE** para cada partição do módulo;
- Um apontador genérico **PORT_MODULE_CONFIGURATION** para configurações do módulo específicas da plataforma de *hardware* – para a plataforma empregada neste trabalho esse campo aponta uma estrutura do tipo **PORT_MODULE_CONFIGURATION_TYPE**, que indica os endereços de início e de fim da área de memória na qual foram alocadas as regiões de pilha de todos os contextos de execução do sistema (utilizados para diferenciação entre erros de estouro de pilha e de violação de memória, conforme apresentado anteriormente), assim como o endereço físico de rede do módulo.

M_C_CT_TYPE Armazena a tabela de conexões do módulo, podendo conter um conjunto de entradas **M_C_CT_SAMPLINGPORT_TYPE** e/ou **M_C_CT_QUEUINGPORT_TYPE**;

M_C_CT_SAMPLINGPORT_TYPE Armazena informações sobre uma porta de amostragem, contendo uma entrada **M_C_CT_SAMPLINGPORTMAPPING_TYPE** para a partição de origem e um conjunto delas para as partições de destino, sendo que cada uma delas aponta ou uma estrutura que contém as informações de uma partição local (executada no próprio módulo), ou uma estrutura específica para a plataforma de *hardware* empregada que contém o endereço físico de uma partição externa (executada em outro módulo);

M_C_CT_QUEUINGPORT_TYPE Armazena informações sobre uma porta de enfileiramento, contendo uma entrada **M_C_CT_QUEUINGPORTMAPPING_TYPE** para a partição de origem e uma para a partição de destino, sendo que cada uma delas aponta ou uma estrutura que contém as informações de uma partição local (executada no próprio módulo), ou uma estrutura específica para a plataforma de *hardware* empregada que contém o endereço físico de uma partição externa (executada em outro módulo);

M_C_HM_TYPE Armazena as informações de configuração do mecanismo de monitoramento do SO para o nível de módulo, contendo o endereço e o tamanho da pilha do *HM callback* desse nível e um conjunto de entradas **M_C_HM_SYSTEMSTATE_TYPE**;

M_C_HM_SYSTEMSTATE_TYPE Mapeia um determinado estado do sistema a um conjunto de entradas **M_C_HM_ERROR_TYPE**, e cada uma dessas mapeia um identificador de erro ao nível no qual seu tratamento deve ser realizado (através da enumeração **ERROR_LEVEL_TYPE**); para erros mapeados ao nível de módulo, indica ainda a ação a ser executada pelo SO quando de sua ocorrência (através da enumeração **M_C_HM_ERROR_ACTION_TYPE**), e para aqueles mapeados ao nível de processo indica também o código associado ao erro (do tipo **ERROR_CODE_TYPE**).

4.9.3.3 Partição

São estruturas cujo nome é prefixado por **PARTITION_CONFIGURATION_** (resumido por **P_C_**), destinadas ao armazenamento das configurações de uma partição do módulo, e são apresentadas a seguir:

P_C_TYPE Armazena as seguintes configurações de uma partição:

- O identificador externo da partição (aquele definido no arquivo de configuração);
- O nome da partição;
- O número máximo de elementos (processos, portas, semáforos, entre outros) suportados pela partição (para alocação de recursos);
- Os endereços e tamanhos efetivos das regiões de memória da partição;
- Uma entrada do tipo **P_C_SCHEDULE_TYPE** apresentado a seguir;
- Uma entrada do tipo **P_C_HEALTHMONITORING_TYPE**, que aponta a árvore de estruturas de nome prefixado por **P_C_HEALTHMONITORING_** (resumido por **P_C_HM_**) apresentada a seguir;
- Uma entrada **PROCESS_CONFIGURATION_TYPE** para cada processo da partição;

- Um apontador genérico **PORT_PARTITION_CONFIGURATION** para configurações da partição específicas da plataforma de *hardware* – para a plataforma empregada neste trabalho esse campo não é utilizado.

P_C_SCHEDULE_TYPE Armazena a escala de uma partição – indica o período e a duração nominais da partição e possui um conjunto de entradas **P_C_SCHEDULE_WINDOW_TYPE**, sendo que cada uma delas armazena uma janela de tempo alocada dentro do *major frame* para a partição, definindo o tempo de início e a duração da janela e indicando se a janela é ou não um início de período da partição;

P_C_HM_TYPE Armazena as informações de configuração do mecanismo de monitoramento do SO para o nível de partição, contendo o endereço e o tamanho da pilha do *HM callback* desse nível e um conjunto de entradas **P_C_HM_SYSTEMSTATE_TYPE**;

P_C_HM_SYSTEMSTATE_TYPE Mapeia um determinado estado do sistema a um conjunto de entradas **P_C_HM_ERROR_TYPE**, e cada uma dessas mapeia um identificador de erro a uma ação a ser executada pelo SO quando de sua ocorrência (através da enumeração **P_C_HM_ERROR_ACTION_TYPE**).

4.9.3.4 Processo

Trata-se da estrutura denominada **PROCESS_CONFIGURATION_TYPE**, que destina-se ao armazenamento das seguintes configurações de um processo de uma partição:

- O nome do processo;
- O endereço e o tamanho efetivos da região de memória de pilha do processo;
- Um apontador genérico **PORT_PROCESS_CONFIGURATION** para configurações do processo específicas da plataforma de *hardware* – para a plataforma empregada neste trabalho esse campo não é utilizado.

4.10 PARTIÇÃO DE SISTEMA PARA ENTRADA/SAÍDA

Conforme citado anteriormente, partições de sistema podem ser utilizadas em sistemas compatíveis com a especificação ARINC 653 para, entre outras tarefas, o processamento de entrada e saída de dados (ARINC, 2006a). Algumas funções relacionadas a essa tarefa são fundamentais para o correto funcionamento de sistemas desse tipo, como por exemplo a transmissão das mensagens manipuladas por elementos de comunicação interpartição (portas de amostragem e de enfileiramento) entre partições de um mesmo módulo (via memória) ou de módulos diferentes (através de interfaces de rede). Por esse motivo, foi implementada neste trabalho uma partição de sistema que é responsável pelo repasse de mensagens nesse contexto, realizando tanto a transmissão entre partições pertencentes a um mesmo módulo (partições locais) quanto a módulos diferentes (partições externas). Para a transmissão de mensagens entre partições externas é utilizada a interface *ethernet* (*IEEE* 802.3) oferecida pela plataforma de *hardware*, que pode tanto ser conectada diretamente a outro módulo quanto a um *switch* compatível com esse padrão, possibilitando assim a interconexão de múltiplos módulos.

A partição de sistema à qual esta seção se refere é denominada **SYSTEMPARTITION_IO** e emprega processos aperiódicos escalonados de forma cooperativa (ARPACI-DUSSEAU, 2012), ou seja, após a realização de uma etapa de sua execução esses processos cedem o processador de forma voluntária aos demais, utilizando para isso o serviço **TIMED_WAIT** com **DELAY_TIME** igual a zero (ARINC, 2006a). Apresenta-se a seguir os processos dessa partição e as funções desempenhadas por cada um deles:

SAMPLINGPORTS Trata a troca de mensagens entre portas de amostragem, ou seja, lê mensagens das portas de origem locais e repassa-as para cada uma das portas de destino definidas na configuração do módulo; essa transmissão é realizada através de serviços internos do SO para partições de destino locais, e através da interface *ethernet* para partições de destino externas;

QUEUINGPORTS Trata a troca de mensagens entre portas de enfileiramento, ou seja, lê mensagens das portas de origem locais e repassa-as para a porta de destino definida na configuração do módulo; essa transmissão é realizada através de serviços internos do SO para partições de destino locais, e através da interface *ethernet* para partições de destino externas;

ETHERNET Inicializa a interface *ethernet* da plataforma e efetua o tratamento dos pacotes recebidos através dela, repassando as mensagens neles contidas às portas de amostragem e/ou de enfileiramento de destino definidas na configuração do módulo.

Com o objetivo de oferecer um nível mínimo de compatibilidade dos pacotes transmitidos através da interface *ethernet* com as exigências da parte 7 da especificação ARINC 664 (ARINC, 2005), que define uma rede determinística para utilização em aeronaves, a partição em questão utiliza os protocolos *User Datagram Protocol* (*UDP*) e *Internet Protocol* (*IP*) no encaminhamento de mensagens. Ressalta-se, porém, que não faz parte dos objetivos deste trabalho a compatibilização da transmissão de mensagens entre módulos com os requisitos dessa especificação, e esse é portanto um possível assunto para trabalhos futuros.

A elaboração da partição apresentada teve como objetivo, além de permitir a troca de mensagens entre módulos, a demonstração prática das limitações impostas pela utilização de SOs compatíveis com a especificação ARINC 653 no que compete à interação com elementos de *hardware*. As seguintes características precisaram ser levadas em conta na implementação da partição de entrada/saída:

- A inicialização da interface *ethernet* da plataforma demora alguns segundos para ser completada, e portanto é necessário que o processo **ETHERNET** ceda o processador aos demais durante a execução dessa tarefa, permitindo assim que as outras operações pelas quais a partição de entrada/saída é responsável não permaneçam inoperantes durante longos períodos;
- A utilização de interrupções para o tratamento dos pacotes recebidos através da interface *ethernet* causaria a violação do escalonamento de partições, já que qualquer uma delas poderia ser interrompida a qualquer momento e, em grande tráfego, a frequência de interrupções poderia ser extremamente alta; portanto, o processo responsável pelo tratamento desses pacotes deve utilizar *polling* (espera ocupada) para detecção de sua recepção, tratando-os exclusivamente durante a execução da partição de entrada/saída.

Conforme citado anteriormente, é utilizada uma pilha de protocolos *UDP*/*IP*/*ethernet* para a transmissão de mensagens através da interface de rede da plataforma, e portanto é necessária a atribuição de endereços *MAC* e *IP* e de portas *UDP* de forma que a comunicação possa ser efetivada. Esses atributos são definidos através do arquivo de configuração do módulo da seguinte forma:

- Na entidade **ModExt:Extension_AM335X** deve ser informado o atributo **ModuleNetworkPhysicalAddress**, o endereço físico de rede do módulo, que consiste em seus endereços *MAC* e *IP* concatenados e convertidos para hexadecimal; por exemplo, para atribuição dos endereços *MAC* 00:00:01:01:02:02 e *IP* 1.2.3.4 a um determinado módulo, o valor do atributo em questão deve ser 0x00000101020201020304;
- Nas entidades **Pseudo_Partition** contidas em entidades **Source** dos canais definidos na tabela de conexões do módulo, o atributo **PhysicalAddress** deve ser utilizado para definição da porta *UDP* (em hexadecimal) na qual serão recebidas as mensagens transmitidas aos destinos correspondentes; por exemplo, para atribuição da porta *UDP* 1234 como origem de um canal o valor do atributo em questão deve ser 0x04D2;
- Nas entidades **Pseudo_Partition** contidas em entidades **Destination** dos canais definidos na tabela de conexões do módulo, o atributo **PhysicalAddress** deve ser utilizado para definição dos endereços *MAC* e *IP* e da porta *UDP* que definem um endereço físico ao qual as mensagens transmitidas no canal devem ser encaminhadas, todos concatenados e convertidos para hexadecimal; por exemplo, para atribuição dos endereços *MAC* 00:00:01:01:02:02 e *IP* 1.2.3.4 e da porta *UDP* 1234 como destino de um canal, o valor do atributo em questão deve ser 0x00000101020201020304 04D2.

Destaca-se que a partição em questão não implementa mecanismos de ordenação ou de retransmissão, e portanto as mensagens são processadas na ordem em que são recebidas e aquelas contidas em pacotes perdidos são ignoradas, tanto para portas de amostragem quanto para portas de enfileiramento. Ressalta-se ainda que, por fazer uso de recursos específicos da plataforma de *hardware* empregada (no caso a interface *ethernet*), a partição de entrada/saída não é genérica, e portanto sua implementação deve ser considerada parte do processo de migração do SO. Guardadas as limitações apresentadas nesta seção, com a partição apresentada é possível a utilização plena dos recursos de comunicação interpartição exigidos pela especificação ARINC 653 no SO desenvolvido neste trabalho.

4.11 CARACTERÍSTICAS E LIMITAÇÕES

Apresenta-se a seguir algumas características e limitações encontradas no SO desenvolvido neste trabalho, justificando sua existência e, quando aplicável, indicando abordagens de contorno:

Bibliotecas A compilação do SO em conjunto com a aplicação executada sobre ele impõe limitações na utilização de bibliotecas, já que essas passam a fazer parte do sistema como um todo, e não de uma partição específica – isso faz com que as variáveis declaradas por essas bibliotecas sejam globais, e portanto a utilização da mesma biblioteca a partir de duas partições distintas torna-se inviável;

Flexibilidade de especificação/linguagem Segundo a especificação ARINC 653 (ARINC, 2006a), as diferentes aplicações executadas em um mesmo módulo podem não utilizar a mesma especificação e/ou a mesma linguagem – essa característica poderia ser alcançada, por exemplo, através do suporte à utilização de diferentes SOs em cada partição do módulo, conforme apresentado em (VANDERLEEST, 2010); porém, no SO desenvolvido neste trabalho, esse tipo de flexibilidade não é possível;

Políticas de *deadline* No SO desenvolvido, a atribuição das políticas de *deadline* dos tipos *soft* e *hard* aos processos de uma partição não tem efeito algum, já que não é citada pela especificação qualquer distinção em função desse parâmetro;

Validações Diversas das validações que a especificação ARINC 653 cita como tarefas do mecanismo de monitoramento (ARINC, 2006a) são executadas, no SO desenvolvido neste trabalho, ou pelas ferramentas de configuração fornecidas conjuntamente – por exemplo a validação da coerência na declaração dos canais de comunicação – ou pelo próprio processo de compilação – por exemplo a garantia de que todas as partições existem; por esse motivo, a utilização do mecanismo de monitoramento para detecção e tratamento dessas condições não é necessária, já que seu atendimento é garantido antes mesmo da compilação do sistema;

Configuração A especificação ARINC 653 cita que deve ser suficiente para a configuração de um SO compatível a utilização apenas dos parâmetros exigidos pelo esquema básico dos arquivos de configuração de módulo (ARINC, 2006a) – no SO desenvol-

vido neste trabalho, optou-se por exigir e/ou sugerir de forma estrita a declaração de diversos atributos, tanto do esquema básico quanto do esquema estendido; essa decisão se justifica pela orientação didática do trabalho, devido à qual considera-se que, para o usuário em treinamento, é mais útil a exigência estrita da corretude e da completude da configuração do que a flexibilidade de funcionamento quando informações incompletas são fornecidas;

Código de erro POWER_FAIL Em alguns locais da especificação ARINC 653 o código de erro que se refere a falhas no suprimento de energia da plataforma de *hardware* é chamado **POWER_FAIL**, enquanto em outros locais é chamado **POWER_FAILURE** – foi utilizada a forma unificada **POWER_FAIL** em todos os locais, uma vez que o principal objetivo da especificação ARINC 653 é a portabilidade de código e essa é a forma utilizada na definição da *APEX*;

Modos de operação COLD_START e WARM_START No SO desenvolvido neste trabalho, não há distinção alguma entre esses modos de operação;

Erros não tratados Os mecanismos definidos pela ARINC 653 permitem que erros não sejam tratados, ou seja, que o elemento afetado por uma falha não seja parado ou reiniciado – o comportamento do SO desenvolvido neste trabalho quando erros não são tratados varia de acordo com o tipo de erro detectado:

- Para erros de aplicação gerados pelo serviço **RAISE_APPLICATION_ERROR**, o contexto de execução no qual o erro ocorreu continua executando normalmente caso esse seja ignorado;
- Para quaisquer outros erros, a execução é retomada a partir da instrução que gerou o erro (a instrução é repetida), e possivelmente o erro será novamente gerado.

Comunicação interpartição em *HM callbacks* A especificação ARINC 653 define que devem existir mecanismos de comunicação que possam ser utilizados a partir de *HM callbacks* para o reporte de erros a partições de sistema responsáveis por tratá-los (ARINC, 2006a) – os mecanismos previstos pela especificação (portas de amostragem e portas de enfileiramento), conforme implementados neste trabalho, não são capazes de desempenhar essa tarefa;

Comunicação interpartição durante inicialização A especificação ARINC 653 define que portas de amostragem e

portas de enfileiramento devem ser utilizáveis em qualquer modo de operação de uma partição, o que inclui sua etapa de inicialização (ARINC, 2006a) – no SO desenvolvido neste trabalho, o processo padrão de uma partição não pode ser bloqueado por esse tipo de mecanismo e, portanto, a utilização de portas de enfileiramento (as mais adequadas a esse tipo de tarefa) não é possível;

Atraso do relógio do sistema Uma vez que o relógio do sistema é atualizado pela interrupção periódica de escalonamento e os serviços do SO são executados atomicamente (com interrupções desabilitadas), é possível que o relógio sofra atrasos caso um desses serviços execute durante um intervalo maior que o período da interrupção;

Interface *ethernet* Diversas são as limitações relacionadas à utilização de mecanismos de comunicação interpartição em conjunto com a interface de rede *ethernet* da plataforma, dentre as quais destaca-se seu não determinismo temporal (próprio do padrão *IEEE* 802.3), a potencialização do não determinismo temporal do processador imposto pela utilização de *DMA*, e ainda a impossibilidade de implementação de redundância, já que a plataforma possui apenas uma porta *ethernet*.

4.12 COMENTÁRIOS GERAIS

Apresenta-se, a seguir, as principais decisões de projeto tomadas durante o desenvolvimento deste trabalho:

- **Implementação em plataforma real** Conforme apresentado em (PASCOAL, 2008), podem ser empregados simuladores para o desenvolvimento e teste de aplicações aviônicas, e portanto o mesmo pode ser afirmado para fins didáticos e experimentais nesse contexto; neste trabalho optou-se pela utilização de uma plataforma de *hardware* real, com o objetivo de oferecer ao usuário uma experiência o mais próxima quanto possível do desenvolvimento em um cenário real, ou seja, de forma que a utilização do SO apresente desafios semelhantes àqueles encontrados nas plataformas empregadas em equipamentos aviônicos;
- **Implementação ao invés de adaptação** Conforme citado anteriormente, seria possível a adaptação ou mapeamento dos recursos de um SO de licença aberta às exigências da ARINC 653,

de forma semelhante ao apresentado em (RUFINO; FILIPE, 2007) e (HAN; JIN, 2012), evitando assim a implementação de um novo núcleo; porém, as opções disponíveis com licença aberta, como o **FreeRTOS** (REAL TIME ENGINEERS LTD., 2014), o **RTEMS** (OAR CORPORATION, 2014) e o **EPOS** (LISHA/UFSC, 2014), não oferecem suporte a diversos elementos exigidos pela ARINC 653, como o escalonamento em dois níveis e tarefas relacionadas à detecção e tratamento de erros (*health monitoring*); a adaptação de um desses SOTRs às definições da especificação exigiria grandes alterações infraestruturais, descaracterizando o *software* original e podendo comprometer sua estabilidade, motivos pelos quais decidiu-se pela implementação de um novo núcleo;

- **Não utilização de virtualização** Conforme apresentado em (VANDERLEEST, 2010) e (HAN; JIN, 2011), a implementação do escalonamento de partições ARINC 653 através da utilização de recursos de virtualização oferece ganhos consideráveis, como por exemplo o suporte à execução de um SO diferente em cada partição; porém, a implementação Cortex^{TM}-A8 da arquitetura *ARMv7*, utilizada pelo processador da plataforma de *hardware* empregada, não oferece esse tipo de recurso;
- **Fila de prioridades** Conforme citado anteriormente, a forma como a especificação ARINC 653 define que os processos devem ser enfileirados acaba por exigir, também, que as filas de prioridades empregadas funcionem como filas *FIFO* quando populadas apenas com entradas de mesma prioridade; por esse motivo, desenvolveu-se neste trabalho uma fila de prioridades cuja complexidade de enfileiramento é $O(n)$ e que obedece a essa característica, porém caso essa exigência não existisse seria preferível a utilização de estruturas mais eficientes, como as *heaps* binárias, que apresentam complexidade de enfileiramento $O(log_2 n)$ mas cujo princípio de funcionamento não garante o comportamento citado;
- **Alocação de memória** Optou-se por uma abordagem mista de alocação de memória, ou seja, parte realizada de forma estática e parte de forma dinâmica; isso porque a alocação de pequenas porções de dados (para estruturas gerenciais, por exemplo) é prejudicada por uma abordagem estática, por gerar um mapa de memória desnecessariamente extenso e complexo, enquanto a alocação dinâmica de porções maiores (para pilhas, por exemplo) não mostra-se vantajosa por dificultar a proteção dessas regiões através da *MMU* e/ou a correta detecção de erros a elas relacio-

nados, conforme apresentado na Seção 4.8;

- **Desabilitação de memória *cache* e *branch prediction*** A memória *cache* e o mecanismo de predição de fluxo (*branch prediction*) do processador utilizado neste trabalho são desabilitados pelo SO; essa decisão foi tomada com o objetivo de obter-se máxima constância no tempo de execução de instruções e, assim, uma maior previsibilidade temporal (WILHELM et al., 2008); apesar de impedir o usufruto do máximo desempenho do processador, essa característica é importante, por exemplo, para a execução dos casos de teste elaborados neste trabalho (apresentados a seguir), nos quais são utilizados laços cuja constância no tempo de execução é fundamental; uma alternativa à desabilitação da memória *cache* seria seu particionamento, conforme avaliado em (ALTMEYER et al., 2014), porém no processador empregado essa abordagem não pode ser aplicada;
- **Desabilitação dos registradores D16 a D31 do coprocessador *VFP*** Por tratar-se de registradores utilizados principalmente para operações *Single Instruction Multiple Data* (*SIMD*) (TI, 2013b), as quais não são empregadas para operações de ponto flutuante usuais, sua utilização foi desabilitada a fim de minimizar o tamanho das estruturas de armazenamento de contextos de execução;
- **Geração de código a partir de arquivos de configuração** A utilização dessa abordagem tem como principal vantagem reduzir a complexidade do código fonte do SO, por dispensar a utilização de mecanismos de transferência e leitura de arquivos de configuração, além de permitir a validação desses arquivos antes que esses sejam efetivamente utilizados;
- **Não utilização de chamadas de supervisor (interrupções de *software*) distintas para cada serviço do SO** Essa decisão torna o código fonte e sua depuração mais compreensíveis, já que diminui a complexidade percebida pelo usuário no que se refere à interface entre o *software* de aplicação e o núcleo do SO;
- **Salvamento de contextos de execução em estruturas dedicadas** Conforme citado anteriormente, alguns SOTRs efetuam o salvamento de contextos de execução na própria pilha de execução das tarefas, como é o caso do **FreeRTOS** (REAL TIME ENGINEERS LTD., 2014), enquanto outros, como o **RTEMS** (OAR CORPORATION, 2014), armazenam essas informações em estruturas dedicadas; a decisão pelo armazenamento de contextos de execução

em estruturas dedicadas se justifica principalmente por essa abordagem permitir o salvamento do contexto de execução mesmo quando da ocorrência de erros de estouro de pilha, o que não é possível na abordagem alternativa e acaba por comprometer os contextos de execução afetados por esse tipo de erro;

- **Utilização de tabelas de tradução de descritores curtos** Conforme apresentado no Capítulo 3, a *MMU* fornecida pelo processador empregado permite a utilização de dois formatos distintos de tabelas de tradução: de descritores curtos e de descritores longos; as tabelas de tradução de descritores longos são mais adequadas em situações nas quais a memória disponível para o sistema exige a utilização de endereços virtuais longos (de mais de 32 *bits*), o que não é o caso da plataforma de *hardware* empregada;
- **Elaboração de uma tabela de tradução para cada contexto de execução** Conforme citado anteriormente, é possível o atendimento aos requisitos de isolamento espacial impostos pela ARINC 653 através da utilização de uma única tabela de tradução por partição, já que desse ponto de vista é exigida apenas a proibição do acesso à memória reservada a outras partições; porém, essa abordagem não é suficiente para a detecção dos erros de memória previstos por essa especificação (violação de memória e estouro de pilha), já que a diferenciação desses dois tipos de erro de forma eficiente depende da proteção individual da região utilizada como pilha por cada contexto de execução, exigindo assim a elaboração de uma tabela de tradução para cada um dos contextos de execução; apesar de gerar um maior número de tabelas e de descritores, causando um maior número de faltas nas *TLBs* da *MMU* (ARPACI-DUSSEAU, 2012), na ausência de mecanismos alternativos para diferenciação de erros de memória utilizou-se neste trabalho uma tabela de tradução para cada contexto de execução;
- **Utilização de contextos de execução distintos para as tarefas do mecanismo de monitoramento** Algumas tarefas do mecanismo de monitoramento do SO (*health monitoring*), como os *HM callbacks*, podem tanto ser tratadas como contextos de execução distintos quanto executadas como invocações de métodos a partir de outros contextos de execução; neste trabalho optou-se por tratá-las como contextos de execução distintos, sendo assim seu escalonamento incorporado ao algoritmo de escalonamento de partições e processos;

- **Desabilitação de interrupções aninhadas** Apesar deste trabalho não abordar questões que tornam necessária a utilização de interrupções aninhadas, observou-se que o tratamento desse tipo de evento traria a necessidade de reorganização da estratégia de salvamento e restauração de contextos de execução empregada no SO desenvolvido, pois exigiria o controle sobre o número de interrupções que encontram-se em tratamento e a supressão da restauração de contexto antes que o tratamento da de menor prioridade fosse concluído;
- **Desenvolvimento de funções comuns** Bibliotecas tradicionalmente utilizadas na linguagem C oferecem funções equivalentes a algumas desenvolvidas neste trabalho, como **COMMON_COPYSTRING** e **COMMON_COMPARESTRINGS**, porém por tratar-se de um número reduzido de casos optou-se pela manutenção da independência do SO em relação a bibliotecas externas, tendo portanto essas sido desenvolvidas como parte do núcleo do SO.

Apresenta-se, a seguir, as principais facilidades encontradas durante o desenvolvimento do SO apresentado neste trabalho:

- **Especificação** A especificação ARINC 653 apresenta em detalhes o comportamento exigido para SOs compatíveis, além de trazer uma implementação de referência dos serviços que devem ser por eles oferecidos ao *software* de aplicação;
- **SOTRs de base** A utilização de SOTRs de código e licença abertos como fonte de informação proporcionou uma rápida evolução para elaboração do núcleo do SO, principalmente no que compete ao escalonamento de tarefas;
- ***TI AM335X StarterWare*** A utilização desse pacote permitiu o desenvolvimento a partir de exemplos especificamente criados para a plataforma de *hardware* escolhida, facilitando a preparação do ambiente (projeto na *IDE*) e a compreensão do funcionamento dessa plataforma;
- **Documentação da arquitetura** A documentação fornecida pela *ARM*®, tanto para a arquitetura *ARMv7* quanto para sua implementação Cortex™-A8, contém todas as informações que mostraram-se necessárias para o desenvolvimento do SO, tendo sido especialmente útil para a implementação dos procedimentos escritos em linguagem *assembly*.

Apresenta-se, a seguir, as principais dificuldades encontradas durante o desenvolvimento do SO apresentado neste trabalho:

- **Biblioteca da *MMU*** O desenvolvimento da biblioteca de interação com a *MMU* exigiu a compreensão extensiva do funcionamento dos mecanismos de proteção de memória e da estrutura das tabelas de tradução de descritores curtos suportadas pela *MMU* utilizada; as maiores dificuldades nessa tarefa se referem à ausência de mecanismos de depuração das tabelas de tradução, e foi portanto necessário o desenvolvimento de programas auxiliares para conferência das informações inseridas nessas tabelas;
- **Tratamento de erros de memória** Todos os erros relacionados ao acesso a memória são reportados da mesma forma pela *MMU* do processador empregado (como exceções de acesso a memória), e portanto sem indicações sobre o tipo de operação que o gerou; sendo assim, foi necessária a elaboração de uma abordagem de diferenciação de erros de estouro de pilha e de violação de memória, conforme apresentado na Seção 4.8, cujo desenvolvimento teve como maiores dificuldades a previsão de possíveis falhas nos diferentes modos de execução do processador e a elaboração de estratégias de tratamento dessas falhas;
- **Obtenção do endereço de retorno da função atual** Essa operação é necessária para a determinação do endereço de origem de erros lançados através do serviço **RAISE_APPLICATION_ERROR**; essa informação é geralmente armazenada na pilha quando da chamada de funções, porém pode assumir diferentes posições na pilha dependendo do nível e do tipo de otimização utilizado pelo compilador; a recuperação dessa informação com a utilização de uma única função também pode mostrar-se inviável, já que o endereço de retorno é alterado quando funções são invocadas; a solução utilizada neste trabalho foi o fornecimento de dois procedimentos: um de preparação, que armazena o endereço de retorno atual em um endereço de memória conhecido (geralmente escrito em linguagem *assembly*), e um de recuperação, que o lê desse endereço e fornece-o ao SO (e pode ser escrito em linguagem C), ambos implementados conforme as características da plataforma de *hardware* utilizada;
- **Configuração de *clocks*** A configuração dos domínios de *clock* do processador representou uma dificuldade devido à grande complexidade desse subsistema no processador empregado, agravado pela ausência de exemplos simples no pacote *AM335X*

StarterWare para alguns periféricos; além disso, essa configuração era originalmente realizada pelos exemplos do pacote *AM335X StarterWare* de forma diferente em tempo de depuração e durante a execução via *bootloader*, e foi portanto necessária a unificação desse procedimento;

- **Implementação do algoritmo de escalonamento** As maiores dificuldades enfrentadas nessa etapa foram relacionadas ao escalonamento em dois níveis em conjunto com as tarefas de tratamento de erros, já que para algumas dessas tarefas o particionamento não pode ser violado (*HM callbacks* de partição, por exemplo), enquanto para outras ele deve ser ignorado (como é o caso do *HM callback* do módulo);
- **Alocação e proteção de memória** Uma das maiores dificuldades enfrentadas no desenvolvimento do SO apresentado está relacionada à determinação das abordagens de alocação e de proteção da memória do sistema atendendo a características como: mínimo desperdício (evitar a alocação além da estrita necessidade), máxima proteção (utilização de mecanismos de proteção da forma mais restritiva possível) e máxima compreensibilidade do mapa de memória, sem prejuízos à depurabilidade do núcleo do SO, da aplicação e de ambos em conjunto.

Apresenta-se, a seguir, as principais lições aprendidas durante o desenvolvimento realizado neste trabalho:

- **Dados do núcleo** Observou-se que uma das etapas mais importantes do desenvolvimento do SO foi a definição das estruturas de informações empregadas em seu núcleo, já que a forma como essas informações são organizadas afeta diretamente a construção de todos os serviços oferecidos ao *software* de aplicação, pode impor limitações à sua evolução ou adaptação e, ainda, gera grande impacto caso precise ser alterada;
- **Abordagens utilizadas em SOs tradicionais** Foi possível compreender durante o desenvolvimento deste trabalho os motivos pelos quais os SOs tradicionais possuem determinadas características, por exemplo quanto à forma como as aplicações são compiladas para que possam ser neles executadas; foi possível perceber, inclusive, que muitas dessas características precisariam ser adotadas para contorno de algumas das principais limitações conhecidas do SO desenvolvido;

- **Contextos de execução** Em processadores simples (microcontroladores, por exemplo), as informações que precisam ser salvas e restauradas quando da troca de contexto de execução são bem conhecidas, limitando-se, geralmente, aos registradores de trabalho e de controle do núcleo do processador; porém em processadores mais complexos, como aquele empregado neste trabalho, esse conjunto de informações pode incluir outros elementos, como por exemplo registradores de trabalho e de controle pertencentes a coprocessadores;
- **Implementação em plataforma real** Foi possível perceber com clareza as vantagens da utilização de SOs ARINC 653 implementados em plataformas reais, ao invés de simulações, quando almejados objetivos didáticos; a principal vantagem percebida é que, quando utilizada simulação, todos os recursos do SO sobre o qual o simulador é executado já encontram-se disponíveis e configurados (como interfaces de rede, por exemplo), enquanto com a utilização de uma implementação em uma plataforma de *hardware* real esses recursos precisam ser preparados e gerenciados sob restrições impostas tanto pelo SO quanto pela plataforma; essa limitação impõe um desafio ao usuário que é necessariamente encontrado em plataformas aviônicas reais, mas que seria abstraído por simuladores que rodam sobre SOs tradicionais.

4.13 CONSIDERAÇÕES FINAIS

Foram apresentados de forma detalhada neste capítulo os elementos que compõem o código fonte do SO desenvolvido neste trabalho, os algoritmos nele empregados, os recursos da plataforma de *hardware* utilizados e as abordagens de utilização desses recursos a fim de atender aos requisitos da especificação ARINC 653. Buscou-se ainda ressaltar e justificar as principais decisões de projeto tomadas, indicando ainda as limitações impostas por características da plataforma e/ou pelas abordagens empregadas ao produto final obtido.

No próximo capítulo serão apresentadas as ferramentas de configuração implementadas neste trabalho para utilização em conjunto com o SO desenvolvido, cujo emprego é fundamental para a validação prévia dos arquivos de configuração e da geração automática de código a partir deles.

5 FERRAMENTAS DE CONFIGURAÇÃO

Os arquivos de configuração de módulo definidos pela ARINC 653 precisam ser pré-processados para que possam ser efetivamente utilizados pelo SO desenvolvido neste trabalho. Para isso desenvolveu-se, além do próprio SO, um conjunto de ferramentas que auxiliam o usuário no processo de validação desse tipo de arquivo, assim como na manutenção das aplicações pelas quais eles são utilizados. Essas ferramentas foram desenvolvidas para a plataforma *Java* versão 1.8.0_25 utilizando a *IDE Eclipse* versão *Kepler*, e suas funções na realização dessas tarefas serão apresentadas nas próximas seções.

5.1 GERENCIADOR DE MODELOS

Trata-se de uma ferramenta destinada à manutenção dos modelos de aplicação gerados a partir de arquivos de configuração de módulo para execução no SO pelo **Gerador de modelos** (apresentado a seguir), de forma que eventuais alterações nesses arquivos sejam propagadas para os modelos correspondentes com o menor esforço possível. Os modelos em questão contêm demarcadores para as regiões de código destinadas ao *software* de aplicação, nas quais deve ser inserido o código do usuário e que devem, portanto, ser mantidas inalteradas quando da regeração do modelo. Esta ferramenta suporta tanto arquivos de código fonte na linguagem C quanto arquivos de configuração no formato *XML*, e demarca regiões de código (*snippets*) com identificadores que devem, preferencialmente, indicar sua função no modelo. Seu funcionamento se baseia em duas operações principais:

Contração Percorre os arquivos de um modelo buscando *snippets* e armazenando-os em um arquivo externo ao modelo (denominado arquivo de **artefato**), removendo o código neles contido mas mantendo a demarcação de sua localização para posterior restauração;

Expansão Percorre os arquivos de um modelo buscando demarcações de localização de *snippets* e as preenche com o conteúdo armazenado em um arquivo de **artefato**, restaurando assim o conteúdo das regiões de código do usuário previamente armazenadas.

A chamada da ferramenta de gerenciamento de modelos é feita através da seguinte linha de comando:

TemplateManager <LANGUAGE> <OPERATION> <PATH> <TEMPLATE> <INCLUDE_PATTERN> <EXCLUDE_PATTERN> <WRITE>

Na qual são utilizados os seguintes argumentos:

LANGUAGE Define a linguagem do modelo – **C** ou **XML**;

OPERATION Define a operação a ser realizada – **COLLAPSE** para contração ou **EXPAND** para expansão;

PATH Caminho do diretório (ou arquivo) sobre o qual a operação deve ser executada;

TEMPLATE Arquivo no qual as regiões de código são armazenadas (escrito pela operação de contração e lido pela operação de expansão);

INCLUDE_PATTERN Expressão regular no padrão da classe **Pattern** da linguagem **Java** (ORACLE, 2014) para seleção dos arquivos do modelo a serem processados – por exemplo ".*\.c" (para arquivos de código fonte em **C**) ou ".*\.xml" (para arquivos no formato **XML**);

EXCLUDE_PATTERN Expressão regular no padrão da classe **Pattern** da linguagem **Java** (ORACLE, 2014) para ignorar arquivos do modelo – por exemplo "configuration\.c" para que o arquivo "configuration.c" não seja processado, mesmo se incluído pelo padrão **INCLUDE_PATTERN**;

WRITE Define se a operação deve efetivamente alterar os arquivos do modelo – **YES** para aplicar a operação aos arquivos do modelo, ou **NO** para mantê-los inalterados.

5.2 PRÉ-PROCESSADOR DE CONFIGURAÇÃO

É uma ferramenta integrada ao **Gerador de modelos** (apresentado a seguir) responsável pela validação sintática e semântica dos arquivos de configuração de módulo, além da execução de rotinas de pré-processamento dessas informações, como por exemplo a alocação de regiões de memória de acordo com as abordagens anteriormente apresentadas. A validação sintática dos arquivos de configuração é realizada frente ao esquema básico definido na especificação ARINC 653 (ARINC, 2006a) unido ao esquema estendido definido neste trabalho, e as validações semânticas incluem verificações de consistência e de coesão das informações contidas em todas suas entidades, das quais destaca-se:

Identificadores Verifica a consistência e unicidade dos identificadores associados aos elementos do sistema;

Nomes Verifica a consistência e unicidade dos nomes atribuídos aos elementos do sistema;

Mapeamento de erros Garante que todos os erros são mapeados a um dos níveis de tratamento e que possuem entrada correspondente na tabela do nível ao qual são mapeados (e apenas na tabela desse nível);

Consistência de declarações Exige que não sejam definidas características a elementos não existentes no módulo (atribuição de tamanhos de pilha a *HM callbacks* não declarados, por exemplo), além de verificar a consistência da tabela de conexões do módulo quanto à existência das partições e portas nela declaradas;

Recursos para testes Exibe avisos sobre a utilização de recursos específicos para testes (desabilitação do mecanismo de monitoramento, por exemplo);

Sugestões de alterações Apresenta sugestões de alterações no arquivo de configuração para facilitar sua leitura (inclusão de descrições de partições consistentes em todas as entidades relacionadas a partições, por exemplo);

Alocação de memória Verifica as configurações de alocação de memória, apontando valores inadequados à execução do sistema na plataforma alvo.

5.3 GERADOR DE MODELOS

É a ferramenta responsável pela geração, com base em um arquivo de configuração de módulo, de um modelo de aplicação que pode ser compilado e executado sobre o SO desenvolvido neste trabalho em uma determinada plataforma de *hardware*. O arquivo de configuração fornecido é inicialmente submetido ao **Pré-processador de configuração**, e a geração do modelo só é efetivada caso não sejam detectados erros por essa ferramenta. A chamada da ferramenta de geração de modelos é feita através da seguinte linha de comando:

TemplateGenerator <CONFIGURATION_FILE> <TARGET_PLATFORM> <OUTPUT_DIRECTORY>

Na qual são utilizados os seguintes argumentos:

CONFIGURATION_FILE Arquivo de configuração de módulo a ser utilizado;

TARGET_PLATFORM Define a plataforma alvo para a qual o modelo deve ser gerado – **DEFAULT** para geração de um modelo genérico (porém não executável), ou **AM335X** para geração de um modelo destinado a processadores *TI* AM335X;

OUTPUT_DIRECTORY Diretório no qual o modelo deve ser gerado.

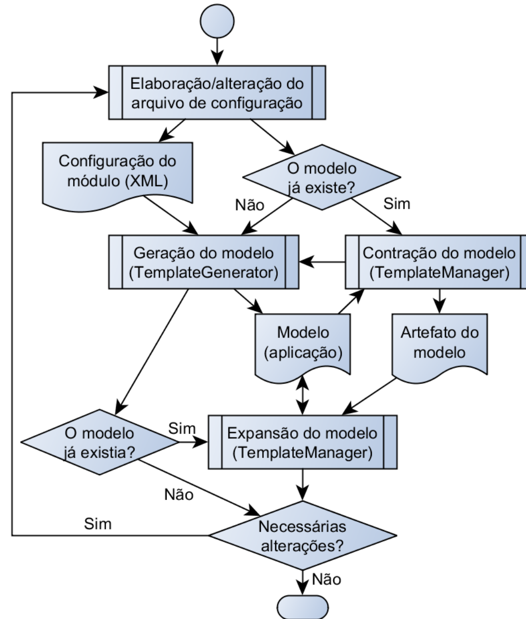
5.4 GERENCIAMENTO DE MODELOS

Conforme citado anteriormente, o modelo gerado pelo **Gerador de modelos** possui regiões demarcadas para a inserção de código do usuário, de forma que, em caso de necessidade de regeração do modelo em razão de alterações de configuração, não seja necessário o reposicionamento desse código ou o rastreamento das alterações efetivas do modelo. Geralmente o **Gerenciador de modelos** e o **Gerador de modelos** são utilizados conjuntamente, da seguinte forma:

1. A operação de **contração** do **Gerenciador de modelos** é utilizada para armazenamento das regiões de código do usuário de um modelo num arquivo de **artefato**;
2. O modelo é regerado (sobrescrito) através do **Gerador de modelos** – sugere-se a utilização de um sistema de controle de versões a fim de evitar a eventual perda de informações em decorrência da regeração do modelo;
3. A operação de **expansão** do **Gerenciador de modelos** é utilizada para restauração das regiões de código do usuário do modelo a partir do arquivo de **artefato** gerado anteriormente.

Ilustra-se na Figura 9 o processo de geração e atualização de modelos a partir de um arquivo de configuração de módulo, indicando os documentos necessários para execução de cada uma das operações desse processo e aqueles gerados por cada uma delas.

Figura 9 – Geração e atualização de modelos



5.5 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentadas as ferramentas elaboradas neste trabalho para auxílio ao usuário nos processos de validação de arquivos de configuração de módulo e de manutenção das aplicações pelas quais eles são utilizados. Sugere-se a utilização dessas ferramentas em conjunto com um sistema de controle de versões que permita a reversão de eventuais alterações indesejadas, já que o emprego de ferramentas de geração automatizada de código insere sérios riscos de perda de informações.

Um exemplo completo das operações necessárias à execução de aplicações sobre o SO desenvolvido neste trabalho é apresentado no Apêndice B.

No próximo capítulo serão apresentados os casos de teste desenvolvidos neste trabalho para evidenciar a aderência do SO desenvolvido às determinações da especificação ARINC 653.

6 CASOS DE TESTE

Parte fundamental do processo de desenvolvimento de um SO com base na ARINC 653, a demonstração de sua compatibilidade com as características exigidas por essa especificação deve ser realizada, no mínimo, através da execução de testes que forneçam evidências dessa aderência. Neste trabalho, um conjunto de casos de teste foi elaborado com o objetivo de demonstrar o funcionamento dos serviços e a aderência das características do SO desenvolvido às exigências da especificação ARINC 653. Esses casos de teste apresentam níveis de complexidade e abrangência variados, e podem portanto ser também utilizados no processo de migração do SO para novas plataformas de *hardware*, iniciando pela execução dos mais simples e alcançando aqueles mais completos e complexos, evidenciando assim o funcionamento do sistema de forma progressiva.

A parte 3 da especificação ARINC 653 (ARINC, 2006b) define um amplo conjunto de testes de conformidade, que têm por finalidade demonstrar e provar que o comportamento de um SO está de acordo com a parte 1 dessa mesma especificação; porém, os casos de teste elaborados neste trabalho não incluem aqueles definidos por esse documento. Ao invés disso, decidiu-se pela criação de um conjunto reduzido de testes, cuja execução fosse ponto de partida para a compreensão da especificação e do funcionamento do SO e demonstrasse a utilização de todos seus serviços, estando assim alinhado também aos objetivos didáticos do trabalho.

Os casos de teste desenvolvidos baseiam-se no acionamento de saídas digitais do processador (*GPIOs*), por tratar-se uma forma de sinalização simples e de resposta rápida. A utilização desse tipo de periférico permite tanto o emprego de um osciloscópio para análise da temporização de eventos quanto a percepção visual do comportamento do sistema através de pinos conectados a *Light-Emitting Diodes* (*LEDs*), o que acaba por tornar a execução dos procedimentos de teste mais intuitiva em comparação ao emprego de recursos como interfaces de depuração, por exemplo. Foram utilizadas as seguintes saídas digitais da plataforma de *hardware BeagleBone*:

LED1 Atua sobre a porta **GPIO1**, *bits* 21 (*LED* **USR0**) e 2 (pino externo **P8.5**);

LED2 Atua sobre a porta **GPIO1**, *bits* 24 (*LED* **USR3**) e 6 (pino externo **P8.3**).

Os seguintes métodos são utilizados nos casos de teste para preparação e manipulação dessas saídas digitais, e devem portanto ser implementados especificamente para a plataforma de *hardware* empregada:

TEST_STARTUP Inicializa as saídas digitais utilizadas nos testes;
TEST_LED1_ON Liga a saída digital **LED1**;
TEST_LED2_ON Liga a saída digital **LED2**;
TEST_LED1_OFF Desliga a saída digital **LED1**;
TEST_LED2_OFF Desliga a saída digital **LED2**.

Os casos de teste desenvolvidos utilizam laços que acionam as saídas digitais do processador durante diferentes períodos, com a finalidade de indicar a ocorrência de diferentes eventos. Para temporização desses laços são definidas as seguintes constantes, cujo valor deve ser calibrado através da utilização de um osciloscópio:

TEST_COUNTER_TINY Gera laços de $\approx$1ms;
TEST_COUNTER_TINYSMALL Gera laços de $\approx$5ms;
TEST_COUNTER_SMALL Gera laços de $\approx$10ms;
TEST_COUNTER_SMALLMEDIUM Gera laços de $\approx$25ms;
TEST_COUNTER_MEDIUM Gera laços de $\approx$50ms;
TEST_COUNTER_MEDIUMLARGE Gera laços de $\approx$100ms;
TEST_COUNTER_LARGE Gera laços de $\approx$250ms;
TEST_COUNTER_LARGEHUGE Gera laços de $\approx$750ms;
TEST_COUNTER_HUGE Gera laços de $\approx$1s.

Apresenta-se nas próximas seções uma visão geral dos casos de teste desenvolvidos neste trabalho, assim como informações referentes ao ambiente no qual eles foram executados e os resultados obtidos a partir de sua execução sobre o SO desenvolvido.

6.1 VISÃO GERAL

Os casos de teste desenvolvidos neste trabalho podem ser categorizados de acordo com a família de funções do SO à qual servem. Apresenta-se a seguir de forma breve o objetivo de cada uma dessas categorias, assim como a função de cada um dos casos de teste pertencentes a cada uma delas. Detalhes sobre cada um desses casos de teste podem ser consultados no Apêndice D.

6.1.1 Escalonamento e execução

Trata-se de testes que têm por objetivo demonstrar o funcionamento do escalonamento de partições (tanto de aplicação quanto de sistema) e de processos, e ainda permitir a calibração das constantes de temporização para os demais testes.

001_SLOWPARTITIONSCHEDULING Teste de escalonamento lento de partições (com janelas de tempo da ordem de segundos);

002_FASTPARTITIONSCHEDULING Teste de escalonamento rápido de partições (com janelas de tempo da ordem de milissegundos);

003_SLOWPROCESSSCHEDULING Teste de escalonamento lento de processos periódicos (com períodos da ordem de segundos);

004_FASTPROCESSSCHEDULING Teste de escalonamento rápido de processos periódicos (com períodos da ordem de milissegundos);

005_TESTCOUNTERCALIBRATION Utilizado para calibração das constantes de temporização utilizadas nos demais testes;

006_RETURNPOINT Teste de tratamento do retorno de todos os contextos de execução do SO;

007_SYSTEMPARTITION Teste de partição de sistema que acessa informações contidas nas estruturas do núcleo do SO.

6.1.2 Serviços ARINC 653 básicos

São testes que demonstram o funcionamento dos serviços básicos oferecidos pelo SO, englobando tanto aqueles exigidos pela especificação quanto aqueles desenvolvidos especificamente neste trabalho.

101_SETMODULEMODE_IDLE Teste de parada do módulo;

102_SETMODULEMODE_COLDSTART Teste de reinicialização do módulo;

103_GETPARTITIONID Teste de obtenção de identificador de partição;

104_GETPARTITIONSTATUS Teste de obtenção de estado de partição;

105_SETPARTITIONMODE_IDLE_PROCESS Teste de parada de partição a partir de um de seus processos;

106_SETPARTITIONMODE_IDLE_PARTITIONERRORHANDLER Teste de parada de partição a partir de seu processo tratador de erros;

107_SETPARTITIONMODE_IDLE_PARTITIONHMCALLBACK Teste de parada de partição a partir de seu *HM callback*;

108_SETPARTITIONMODE_COLDSTART_PROCESS Teste de reinicialização de partição a partir de um de seus processos;

109_SETPARTITIONMODE_COLDSTART_PARTITIONERRORHANDLER Teste de reinicialização de partição a partir de seu processo tratador de erros;

110_SETPARTITIONMODE_COLDSTART_PARTITIONHMCALLBACK Teste de reinicialização de partição a partir de seu *HM callback*;

111_GETPROCESSID Teste de obtenção de identificador de processo;

112_GETPROCESSSTATUS Teste de obtenção de estado de processo;

113_SETPRIORITY Teste de alteração de prioridade de processo;

114_SUSPEND_RESUME Teste de suspensão e continuação de processo;

115_SUSPENDSELF_RESUME Teste de auto-suspensão e continuação de processo;

116_SUSPENDSELF_RESUME_TIMEOUT Teste de auto-suspensão com limite de tempo e continuação de processo;

117_TIMEDWAIT Teste de bloqueio por tempo;

118_TIMEDWAIT_SUSPEND_RESUME Teste de suspensão e continuação de processo durante bloqueio por tempo;

119_TIMEDWAIT_COOPERATIVESCHEDULING Teste de escalonamento cooperativo de processos através do serviço **TIMED_WAIT**;

120_REPLENISH Teste de postergação de *deadline* de processos;

121_STOPSELF_START Teste de auto-parada e reinício de processo;

122_STOPSELF_DELAYEDSTART Teste de auto-parada e reinício de processo com atraso;

123_STOP_START Teste de parada e reinício de processo;

124_STOP_START_PERIODSTART Teste de parada e reinício de processo periódico para verificação de sua primeira liberação, que deve ser a partir do próximo início de período da partição;

125_STOP_DELAYEDSTART Teste de parada e reinício de processo com atraso;

126_DELAYEDSTART Teste de inicialização de processo com atraso;

127_LOCKPREEMPTION_UNLOCKPREEMPTION Teste de bloqueio de interrupção de processos;

128_LOCKPREEMPTION_STOPSELF_PROCESS Teste de auto-parada durante bloqueio de interrupção de processos;

129_LOCKPREEMPTION_STOPSELF_PARTITIONERRORHANDLER Teste de auto-parada do processo tratador de erros da partição durante bloqueio de interrupção de processos;

130_GETMYID Teste de obtenção de identificador do processo atual.

6.1.3 Serviços internos

Trata-se de testes que demonstram o funcionamento de alguns dos principais serviços internos utilizados no SO desenvolvido, assim como a interação desses com os serviços básicos que influenciam sua operação.

201_WAITRESOURCE_SIGNALRESOURCE Teste de aquisição e liberação de recurso;

202_WAITRESOURCE_SETPRIORITY_SIGNALRESOURCE Teste de aquisição e liberação de recurso com alteração de prioridade de processo;

203_WAITRESOURCE_SIGNALRESOURCE_TIMEOUT Teste de aquisição e liberação de recurso com limite de tempo;

204_WAITRESOURCE_SUSPEND_TIMEOUT_RESUME Teste de aquisição e liberação de recurso com limite de tempo e intercalação da suspensão e continuação de processo;

205_WAITRESOURCE_SUSPEND_SIGNALRESOURCE_RESUME Teste de aquisição e liberação de recurso com suspensão e continuação de processo;

206_WAITRESOURCE_SUSPEND_RESUME_SIGNALRESOURCE Teste de aquisição e liberação de recurso com intercalação da suspensão e continuação de processo.

6.1.4 Serviços ARINC 653 de comunicação

Demonstram o funcionamento dos elementos de comunicação intrapartição e interpartição, explorando ao máximo os serviços relacionados e as possibilidades de uso de cada um desses elementos.

301_EVENT Teste de eventos;

302_SEMAPHORE_FIFO Teste de semáforo com atendimento em política *FIFO*;

303_SEMAPHORE_PRIORITY Teste de semáforo com atendimento por prioridade;

304_SEMAPHORE_DEADLOCK Teste de *deadlock* de semáforos;

305_BLACKBOARD Teste de *blackboard*;

306_BUFFER Teste de *buffer*;

307_BUFFER_FIFO Teste de *buffer* com atendimento em política *FIFO*;

308_BUFFER_PRIORITY Teste de *buffer* com atendimento por prioridade;

309_BUFFER_TIMEOUT_RECEIVE Teste de *buffer* com limite de tempo de recebimento;

310_BUFFER_TIMEOUT_SEND Teste de *buffer* com limite de tempo de envio;

311_BUFFER_FULL_EMPTY Teste de *buffer* com preenchimento e esvaziamento totais;

312_SAMPLINGPORT_STANDARD Teste de portas de amostragem conectadas a partições do mesmo módulo;

313_SAMPLINGPORT_PSEUDO_MODULE1 Teste de portas de amostragem conectadas a partições de módulos diferentes (utilizado em conjunto com o teste **314_SAMPLINGPORT_PSEUDO_MODULE2**);

314_SAMPLINGPORT_PSEUDO_MODULE2 Teste de portas de amostragem conectadas a partições de módulos diferentes (utilizado em conjunto com o teste **313_SAMPLINGPORT_PSEUDO_MODULE1**);

315_QUEUINGPORT_STANDARD Teste de portas de enfileiramento conectadas a partições do mesmo módulo;

316_QUEUINGPORT_PSEUDO_MODULE1 Teste de portas de enfileiramento conectadas a partições de módulos diferentes (utilizado em conjunto

com o teste **317_QUEUINGPORT_PSEUDO_MODULE2**);

317_QUEUINGPORT_PSEUDO_MODULE2 Teste de portas de enfileiramento conectadas a partições de módulos diferentes (utilizado em conjunto com o teste **316_QUEUINGPORT_PSEUDO_MODULE1**).

6.1.5 Monitoramento

Têm por objetivo demonstrar o funcionamento dos mecanismos relacionados ao monitoramento (*health monitoring*) oferecidos pelo SO.

401_HEALTHMONITORING_CURRENTSYSTEMSTATE Teste de detecção dos estados do sistema pelo mecanismo de monitoramento;

402_HEALTHMONITORING_PROPAGATION_APPLICATIONERROR Teste de propagação de erros de aplicação;

403_HEALTHMONITORING_PROPAGATION_STACKOVERFLOW Teste de detecção e propagação de erros de estouro de pilha;

404_HEALTHMONITORING_PROPAGATION_MEMORYVIOLATION Teste de detecção e propagação de erros de violação de memória.

6.1.6 Especiais

São testes que fazem uso de recursos encontrados apenas em determinadas plataformas de *hardware* ou desenvolvidos de forma específica para uma determinada plataforma.

501_FLOATINGPOINT Demonstra o funcionamento do salvamento e da restauração de contexto quando da utilização de um coprocessador aritmético de ponto flutuante;

502_SAMPLINGPORT_SENSOR_MODULE1 Teste de portas de amostragem conectadas a partições de módulos diferentes que realiza a transmissão de amostras de um sensor obtidas através de uma entrada *ADC* (utilizado em conjunto com o teste **503_SAMPLINGPORT_SENSOR_MODULE2**);

503_SAMPLINGPORT_SENSOR_MODULE2 Teste de portas de amostragem conectadas a partições de módulos diferentes que gera um sinal *PWM* com base em amostras de um sensor (utilizado em conjunto com o teste **502_SAMPLINGPORT_SENSOR_MODULE1**).

6.2 EXECUÇÃO DOS TESTES

Esta seção tem por objetivo apresentar o *bootloader* adaptado para simplificação da execução dos casos de teste, assim como o ambiente no qual essa tarefa deve ser realizada e os resultados obtidos pela utilização desses testes durante o desenvolvimento do SO.

6.2.1 *Bootloader*

Para simplificação da execução dos casos de teste apresentados, foi realizado neste trabalho o desenvolvimento de uma versão especial do *bootloader* para a *BeagleBone* fornecido no pacote *AM335X StarterWare*. Esse novo *bootloader* tenta obter a aplicação a ser executada a partir das seguintes origens e na seguinte ordem:

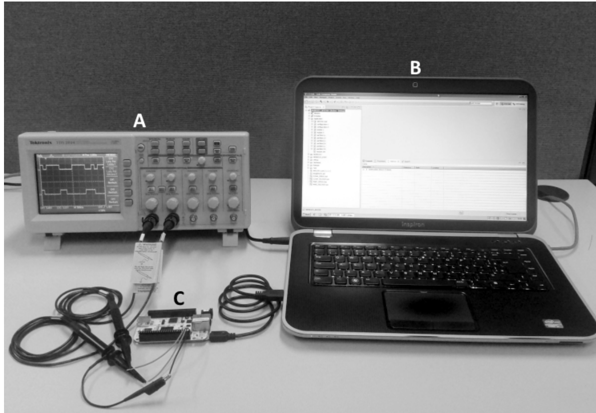
1. A partir do arquivo **APP** contido no diretório raiz do cartão *SD*;
2. Caso o arquivo **APP** não exista, aguarda o recebimento através da interface *UART0* (conectada a uma porta *serial* virtual acessível através da interface *USB client* da plataforma) do nome do arquivo a ser carregado, que também deve estar contido no cartão *SD*.

Esse *bootloader* simplifica a execução de diferentes testes na plataforma, já que todos podem ser armazenados em um único cartão *SD*, e permite ainda que um programa padrão seja carregado automaticamente quando a plataforma é inicializada, bastando para isso armazená-lo no cartão com o nome **APP**.

6.2.2 Ambiente

Para execução dos casos de teste desenvolvidos neste trabalho é necessária uma bancada composta de um microcomputador, um osciloscópio e uma *BeagleBone* (para alguns dos testes são necessárias duas delas). Esses elementos podem ser observados respectivamente nos destaques A, B e C da Figura 10, na qual é apresentada uma fotografia da bancada de testes utilizada durante o processo de desenvolvimento do SO.

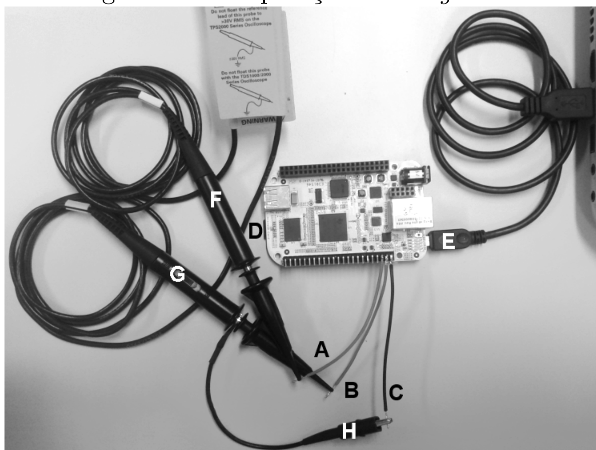
Figura 10 – Bancada de testes



Os elementos da bancada de testes devem ser preparados de acordo com as seguintes instruções:

- Plataforma *BeagleBone*:
  - Os pinos externos **P8.5**, **P8.3** e **P8.1** da placa devem ser expostos através de uma barra de pinos, para que seja possível a conexão do osciloscópio a eles (destaques A, B e C da Figura 11, respectivamente);
  - Antes de ser acionada, deve ser conectada a um cartão *SD* que contém as imagens de memória do *bootloader* (arquivo **MLO**) e de todos os casos de teste, previamente preparadas de acordo com o processo apresentado no Apêndice C e no Apêndice B (destaque D da Figura 11);
  - Deve ser conectada ao microcomputador através do cabo *USB*, tanto para alimentação quanto para acesso à porta *serial* virtual e, se necessário, para depuração (destaque E da Figura 11).

Figura 11 – Preparação da *BeagleBone*



- Osciloscópio:
  - Duas de suas ponteiras devem ser conectadas aos pinos externos **P8.5** e **P8.3** da *BeagleBone*, e a garra de aterramento de uma ou de ambas as ponteiras deve ser conectada ao pino **P8.1** da placa (destaques F, G e H da Figura 11, respectivamente);
  - As escalas utilizadas podem variar de acordo com o teste executado, porém em geral deve ser utilizada escala de amplitude de 5V (uma vez que as saídas digitais operam em 3.3V) e escalas de tempo da ordem de milissegundos (entre 10ms e 500ms, dependendo das características do teste em execução).
- Microcomputador:
  - Deve executar um terminal de acesso à porta *serial* virtual da *BeagleBone* que é conectada à *UART0* do processador e é acessível através da porta *USB* (destaque E da Figura 11).

6.2.3 Procedimento

Conforme citado anteriormente, os casos de teste desenvolvidos baseiam-se no acionamento de saídas digitais do processador conectadas a *LEDs* e a pinos externos. O procedimento de execução desses

testes consiste no disparo do programa correspondente a cada caso de teste seguido da observação visual dos *LEDs* da plataforma (para testes simples), e/ou da captura através de osciloscópio e análise do comportamento das saídas digitais correspondentes.

Tomemos como exemplo o teste **001**, denominado **SLOW-PARTITIONSCHEDULING**, que tem por objetivo demonstrar a corretude temporal do escalonamento de partições, e é executado segundo o cenário apresentado na Tabela 23. Observa-se nesse cenário a existência de três diferentes partições e a ausência de processos, sendo portanto o sistema composto apenas dos processos padrão das partições declaradas e da partição ociosa.

Tabela 23 – Cenário do teste 001

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	4s
Partição PARTITION1	
Período/duração	2s/1s
Escala	(0ms, 1s), (2s, 1s)
Partição PARTITION2	
Período/duração	2s/500ms
Escala	(1s, 500ms), (3s, 500ms)
Partição PARTITION3	
Período/duração	4s/250ms
Escala	(1.5s, 250ms)

Nesse teste, os seguintes padrões são repetidamente atribuídos pelas tarefas do módulo aos *LEDs* da plataforma de *hardware*, de forma a permitir a identificação da tarefa que encontra-se em execução a qualquer momento do teste:

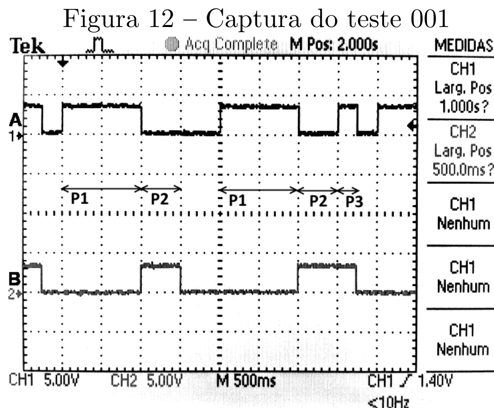
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

Espera-se, portanto, que durante a execução do caso de teste em questão seja possível a observação da seguinte sequência de padrões, repetidamente, nas saídas digitais da plataforma:

- Apenas **LED1** ligado durante 1s (PARTITION1);
- Apenas **LED2** ligado durante 500ms (PARTITION2);
- Ambos *LEDs* ligados durante 250ms (PARTITION3);
- Ambos *LEDs* desligados durante 250ms (partição ociosa);
- Apenas **LED1** ligado durante 1s (PARTITION1);
- Apenas **LED2** ligado durante 500ms (PARTITION2);
- Ambos *LEDs* desligados durante 500ms (partição ociosa).

Conectando-se à plataforma um osciloscópio e efetuando a captura da sequência de padrões gerados por esse caso de teste ao longo de alguns segundos, obtém-se uma saída semelhante à apresentada na Figura 12. Nessa captura, os canais de amostragem 1 e 2 encontram-se conectados às saídas digitais correspondentes aos *LEDs* **LED1** e **LED2**, respectivamente, e pode-se observar assim a corretude temporal dos eventos analisados a partir do tempo zero (demarcado pela seta no topo da imagem), observando-se porém que o tempo zero da amostragem apresentada na figura não corresponde ao tempo zero (início da execução) do teste.

Figura 12 – Captura do teste 001



Segundo essa análise, tem-se que o resultado do teste em questão é considerado aceito. A descrição dos objetivos, cenários e resultados esperados dos demais casos de teste desenvolvidos podem ser consultados no Apêndice D.

6.2.4 Resultados e cobertura

A execução dos casos de teste foi realizada em diversas ocasiões durante o desenvolvimento deste trabalho, tanto para a depuração do núcleo e dos serviços quando de sua implementação quanto para certificação da manutenção do funcionamento desses elementos após alterações significativas. De fato o conjunto de casos de teste apresentado foi construído a partir da necessidade de depuração do SO durante seu desenvolvimento, e foram portanto extensivamente verificados frente às exigências da especificação ARINC 653. Certificou-se, portanto, que os resultados esperados para cada caso de teste, conforme apresentado no Apêndice D, são satisfeitos pelo SO.

Destaca-se que os casos de teste desenvolvidos cobrem o funcionamento dos serviços do SO apenas em sua utilização normal, e não têm por objetivo a comprovação de sua robustez de forma semelhante aos testes definidos na parte 3 da especificação ARINC 653 (ARINC, 2006b). Esses casos de teste simulam cenários que causam a execução de todos os serviços do SO em todas as principais condições que exigem comportamentos alternativos desses serviços, cobrindo portanto as principais porções do código fonte desenvolvido.

6.3 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados os casos de teste elaborados neste trabalho, que têm a finalidade de demonstrar o funcionamento dos serviços oferecidos e auxiliar no processo de migração do SO a outras plataformas de *hardware*. Apesar de não englobar os testes definidos na parte 3 da ARINC 653 (ARINC, 2006b), o conjunto de casos de teste apresentado demonstra a utilização e o funcionamento de todos os serviços oferecidos pelo SO e permitem, com certas limitações, a verificação da aderência de seu comportamento à parte 1 da especificação. Conforme citado anteriormente, a descrição completa desses casos de teste pode ser encontrada no Apêndice D.

No próximo capítulo serão apresentadas as conclusões obtidas a partir do desenvolvimento realizado neste trabalho.

7 CONCLUSÃO

A parte 1 da especificação ARINC 653 estabelece um ambiente de operação padrão para o *software* de aplicação utilizado em equipamentos eletrônicos para aeronaves, definindo características funcionais e temporais para SOs que operam nesse contexto. Os benefícios esperados de sua utilização são a portabilidade, a reusabilidade, a modularidade e a integração do *software* de aplicação, além da redução de custos através da promoção da competitividade entre fornecedores. SOs compatíveis com essa especificação permitem que múltiplas aplicações com diferentes níveis de criticalidade sejam executadas sobre uma mesma plataforma de *hardware*, utilizando para isso partições isoladas temporalmente, através do estabelecimento de uma escala estática de execução, e espacialmente, por meio de mecanismos de proteção de memória. Sobre essas partições são executados processos com escalonamento por prioridade, que podem ser periódicos ou aperiódicos e que devem utilizar apenas os serviços oferecidos pelo SO. Os serviços que devem necessariamente ser oferecidos por SOs compatíveis com a ARINC 653 ao *software* de aplicação estão divididos nas categorias **Gerenciamento de partições**, **Gerenciamento de processos**, **Gerenciamento de tempo**, **Comunicação interpartição**, **Comunicação intrapartição** e **Monitoramento (*Health Monitoring*)** (ARINC, 2006a).

Neste trabalho foi desenvolvido um SO compatível com as determinações da parte 1 da especificação ARINC 653 para execução na plataforma de *hardware* comercialmente conhecida como *BeagleBone*, um microcomputador de *hardware* aberto equipado com um processador *TI* AM335X. Esse processador possui um núcleo *ARM*® Cortex™-A8, baseado na arquitetura *ARMv7* perfil **ARMv7-A** (para processadores de aplicação), que oferece uma *MMU* com suporte ao mapeamento de endereços e ao controle de acesso, a qual foi utilizada para implementação do isolamento espacial de partições. O processador oferece ainda *timers* capazes de gerar interrupções periódicas razoavelmente precisas, dos quais um foi utilizado para a manutenção do relógio do sistema e para o escalonamento de partições e de processos, garantindo assim o isolamento temporal de partições (COLEY, 2012; TI, 2011; ARM, 2012; ARINC, 2006a).

Para utilização da *MMU* oferecida pelo processador para efetivação do isolamento espacial de partições foi necessário um levantamento dos contextos de execução (tarefas) que são gerenciados pelo SO,

das regiões de memória que precisam ser utilizadas por cada um deles e das permissões de acesso que cada contexto de execução deve ter a cada região de memória alocada. Nesse processo foram levadas em conta tanto características comuns a todo SO, como a alocação de uma região de pilha para cada contexto de execução, como aspectos intimamente relacionados às definições da especificação, como a atribuição de permissões que permitam a transferência de informações entre pilhas de processos por determinados serviços do SO. Como resultado, obteve-se um mapeamento de permissões de acesso que atende às restrições impostas pela ARINC 653 no que se refere ao isolamento espacial de partições, como por exemplo a limitação da escrita a qualquer endereço de memória a no máximo uma partição (ARINC, 2006a).

O isolamento temporal exigido pela ARINC 653, que engloba tanto o escalonamento de partições quanto o de processos, foi implementado através de um único algoritmo que opera em dois diferentes níveis: o primeiro seleciona a partição a ser executada no momento atual de acordo com a escala estática definida em tempo de configuração, e o segundo define o processo da partição selecionada que será efetivamente executado de acordo com as prioridades dos processos da partição que encontram-se prontos. Esse algoritmo é disparado periodicamente por uma interrupção de *hardware*, e emprega filas de prioridades tanto para a seleção de partições quanto para o escalonamento de processos. As filas de prioridades utilizadas garantem que processos de mesma prioridade serão escalonados em política *FIFO* segundo o momento de sua liberação, estando portanto de acordo com as exigências impostas pela especificação com relação a essa ordem (ARINC, 2006a).

Por estar diretamente relacionado à criticalidade dos sistemas nos quais SOs compatíveis com a ARINC 653 operam, o subsistema de monitoramento (*health monitoring*) pode ser considerado um dos recursos mais importantes dentre aqueles exigidos por essa especificação. Esse subsistema destina-se ao tratamento de erros detectados através dos mecanismos de proteção oferecidos pelo processador (erros de acesso a memória detectados pela *MMU*, por exemplo), pelo núcleo do SO (perdas de *deadline*, por exemplo) ou pela própria aplicação. O SO desenvolvido neste trabalho suporta todas as tarefas de alta prioridade definidas pela ARINC 653 para esse subsistema, como os processos tratadores de erros (*error handlers*) e os *HM callbacks* de partição e de módulo, e realiza o escalonamento (disparo) dessas tarefas através do mesmo algoritmo que escalona partições e processos (ARINC, 2006a).

Para tratamento de erros de acesso à memória através do subsistema de monitoramento, a especificação ARINC 653 exige que esses

sejam classificados em erros de violação de memória (acesso a regiões de memória pertencentes a outras partições, por exemplo) ou erros de estouro de pilha (empilhamento ou desempilhamento de dados além dos limites da pilha de um contexto de execução), porém ambos são reportados pela *MMU* utilizada de forma idêntica. Para diferenciação desses tipos de erro, foi empregada uma abordagem que se baseia na alocação das regiões de pilha de forma contígua e exige a garantia de que um mesmo contexto de execução não tenha acesso a quaisquer duas regiões de pilha consecutivas. Uma vez obtida essa garantia, a diferenciação dos tipos de erro citados é realizada através de uma simples comparação de endereços.

Outro aspecto importante de SOs compatíveis com a ARINC 653 é sua configuração, que deve ser realizada de forma a permitir que uma mesma aplicação possa ser executada em aeronaves diferentes sem alterações de código. Isso é alcançado através do emprego de arquivos de configuração de módulo no formato *XML*, que contêm detalhes referentes às aplicações e ao acoplamento entre elas e a plataforma na qual são executadas. Neste trabalho foi elaborada uma extensão do esquema básico de configuração fornecido pela especificação, suportando assim as especificidades da plataforma de *hardware* utilizada, e foram desenvolvidas ainda ferramentas destinadas ao pré-processamento dos arquivos de configuração e à geração, com base neles, de modelos de aplicação para execução no SO desenvolvido. A utilização dessas ferramentas tem como principais vantagens a detecção de erros de configuração antes que esses afetem a aplicação, e ainda a atualização automatizada dos modelos de aplicação quando da alteração das configurações a eles associadas.

A fim de evidenciar o funcionamento do SO desenvolvido, a aderência de suas características às exigências da especificação ARINC 653 e demonstrar a utilização de todos os serviços por ele oferecidos ao *software* de aplicação, foi elaborado neste trabalho um conjunto de casos de teste. Esses casos de teste apresentam níveis de complexidade e abrangência variados, e reproduzem tanto cenários ordinários (execução normal de aplicações, por exemplo) quanto cenários nos quais é exigido pela especificação um comportamento específico por parte do SO (propagação de erros, por exemplo), podendo portanto ser utilizados ainda como recurso didático ou para auxílio à migração do SO a novas plataformas de *hardware*.

Apresenta-se na Tabela 24 algumas estatísticas relacionadas ao código fonte do SO desenvolvido neste trabalho. Essas estatísticas são separadas segundo as diferentes linguagens utilizadas (C e *assembly*)

e porções (genérica, específica e de uma aplicação simples). As estatísticas obtidas se referem ao número de linhas de código e ao número de funções declaradas em cada porção analisada.

Tabela 24 – Estatísticas de código

Porção	**Linguagem**	**Linhas de código**	**Funções**
Genérica	C	7317	111
Específica	C	2091	36
Específica	*Assembly*	322	30
Aplicação	C	583	6

O desenvolvimento deste trabalho forneceu subsídios para a publicação do artigo intitulado "Mapeamento do isolamento espacial da ARINC 653 para a arquitetura *ARMv7-A*" nos anais do IV Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC), realizado em Manaus/AM entre os dias 3 e 7 de novembro de 2014. Esse artigo descreve o funcionamento da *MMU* oferecida pela arquitetura *ARMv7* perfil A, e apresenta o mapeamento de permissões utilizado para garantia do isolamento espacial exigido pela ARINC 653 no SO desenvolvido neste trabalho.

Como possíveis trabalhos futuros destaca-se a migração do SO desenvolvido para uma plataforma de *hardware* determinista, permitindo assim o estudo do oferecimento de garantias de escalonabilidade em sistemas baseados na ARINC 653, o tratamento das principais limitações desse SO, alcançando assim máxima compatibilidade com essa especificação, e ainda o desenvolvimento de artefatos que permitam a utilização do SO em conjunto com equipamentos compatíveis com a especificação ARINC 664 (*AFDX*), oferecendo determinismo temporal na transmissão de pacotes de rede e compondo, assim, uma plataforma completa de experimentação de aplicações aviônicas compatível com os padrões modernos de programação e comunicação.

REFERÊNCIAS

ALTMEYER, S. et al. Evaluation of Cache Partitioning for Hard Real-Time Systems. In: **Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on**. [S.l.: s.n.], 2014. p. 15–26.

ARINC. **Aircraft Data Network Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network (ARINC Specification 664P7)**. 2551 Riva Road, Annapolis, Maryland 21401-7435, jun. 2005.

ARINC. **Avionics Application Software Standard Interface Part 1 - Required services (ARINC Specification 653P1-2)**. 2551 Riva Road, Annapolis, Maryland 21401-7435, mar. 2006.

ARINC. **Avionics Application Software Standard Interface Part 3 - Conformity Test Specification (ARINC Specification 653P3)**. 2551 Riva Road, Annapolis, Maryland 21401-7435, out. 2006.

ARM. **Cortex-A8 Technical Reference Manual**. [S.l.], maio 2010. revision r3p2, ARM DDI 0344K (ID060510).

ARM. **ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition**. [S.l.], jul. 2012. issue C.b, ARM DDI 0406C.b (ID072512).

ARM. **ARM Cortex-A Series Programmer's Guide**. [S.l.], jan. 2014. version 4.0, issue D, ARM DEN 0013D (ID012214).

ARPACI-DUSSEAU, R. H. **Operating Systems: Three Easy Pieces**. [s.n.], 2012. ISBN 9781105979125. Disponível em: <http://pages.cs.wisc.edu/ remzi/OSTEP/>.

BEAGLEBOARD.ORG. **BeagleBone**. jun. 2014. Disponível em: <http://beagleboard.org/Products/BeagleBone>.

CARPENTER, J. et al. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. In: **Handbook on Scheduling Algorithms, Methods, and Models**. [S.l.]: Chapman Hall/CRC, Boca, 2004.

COLEY, G. **BeagleBone Rev A6 System Reference Manual**. [S.l.], maio 2012. revision 0.0, reference BBONE_SRM.

FARINES, J.-M.; FRAGA, J. da S.; OLIVEIRA, R. S. de. **Sistemas de Tempo Real**. [S.l.: s.n.], 2000. IME-USP, São Paulo (SP), 24-28 de Julho.

HAN, S.; JIN, H.-W. Full virtualization based ARINC 653 partitioning. In: **Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th**. [S.l.: s.n.], 2011. p. 7E1–1–7E1–11. ISSN 2155-7195.

HAN, S.; JIN, H.-W. Kernel-level ARINC 653 Partitioning for Linux. In: **Proceedings of the 27th Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2012. (SAC '12), p. 1632–1637. ISBN 978-1-4503-0857-1. Disponível em: <http://doi.acm.org/10.1145/2245276.2232037>.

HAN, S.; JIN, H.-W. Resource partitioning for Integrated Modular Avionics: comparative study of implementation alternatives. **Software: Practice and Experience**, 2013. ISSN 1097-024X. Disponível em: <http://dx.doi.org/10.1002/spe.2210>.

LISHA/UFSC. **EPOS User Guide**. set. 2014. Software/Hardware Integration Lab, Federal University of Santa Catarina. Disponível em: <http://epos.lisha.ufsc.br/>.

OAR CORPORATION. **RTEMS On-Line Library**. maio 2014. Disponível em: <http://www.rtems.org/onlinedocs/doc-current/share/rtems/html/>.

ORACLE. **Java SE 7 Documentation**. ago. 2014. Disponível em: <http://docs.oracle.com/javase/7/docs/>.

PASCOAL, E. M. C. da S. **AMOBA - ARINC 653 Simulator for Modular Space Based Applications**. Dissertação (Mestrado) — Faculdade de Ciências, Universidade de Lisboa, Lisboa, 2008. `http://hdl.handle.net/10455/3241`.

PRISAZNUK, P. ARINC 653 role in Integrated Modular Avionics (IMA). In: **Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th**. [S.l.: s.n.], 2008. p. 1.E.5–1–1.E.5–10.

REAL TIME ENGINEERS LTD. **FreeRTOS Documentation**. maio 2014. Disponível em: <http://www.freertos.org/RTOS.html>.

ROCKWELL COLLINS. **The History of Rockwell Collins' ARINC**. jun. 2014. Disponível em: <http://www.arinc.com/about/history.html>.

ROMANSKI, G. The Challenges of Software Certification. **CROSSTALK: The Journal of Defense Software Engineering**, 2011.

RUFINO, J.; FILIPE, S. **AIR Project Final Report**. [S.l.], 2007.

SAE ITC. **AEEC - Engineering Standards for Aircraft Systems**. jun. 2014. Disponível em: <http://www.aviation-ia.com/aeec/>.

TANENBAUM, A. S.; WOODHULL, A. S. **Operating Systems Design and Implementation (3rd Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005. ISBN 0131429388.

TI. **AM335x ARM Cortex-A8 Microprocessors (MPUs) Technical Reference Manual**. [S.l.], out. 2011. revised August 2013, literature number SPRUH73I.

TI. **ARM Assembly Language Tools v5.1 User's Guide**. [S.l.], jun. 2013. literature number SPNU118L.

TI. **ARM Optimizing C/C++ Compiler v5.1 User's Guide**. [S.l.], jun. 2013. literature number SPNU151I.

TI. **Quick Start Guide StarterWare 02.00.XX.XX (supports AM335x)**. [S.l.], jul. 2013.

VANDERLEEST, S. H. ARINC 653 hypervisor. In: **Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th**. [S.l.: s.n.], 2010. p. 5.E.2–1–5.E.2–20. ISSN 2155-7195.

WEISS, M. A. **Data Structures and Algorithm Analysis in Java**. 3rd. ed. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132576279, 9780132576277.

WILHELM, R. et al. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. **ACM Trans. Embed. Comput. Syst.**, New York, NY, USA, v. 7, n. 3, p. 36:1–36:53, maio 2008. ISSN 1539-9087.

WONG, C. et al. Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. In: **Information Technology, 2008. ITSim 2008. International Symposium on**. [S.l.: s.n.], 2008. v. 4, p. 1–8.

APÊNDICE A – Exemplo de código de aplicação

Este apêndice apresenta um exemplo do código fonte de uma aplicação executada sobre o SO desenvolvido neste trabalho, sendo essa composta de uma única partição que executa um único processo periódico. Destaca-se que são exemplificados apenas os arquivos de usuário dessa aplicação, ou seja, apenas os arquivos nos quais o usuário deve inserir código. As seções a seguir apresentam, respectivamente, o código fonte do módulo e da partição dessa aplicação.

A.1 MÓDULO

O código fonte do módulo engloba a inicialização do módulo e a criação das partições. O Trecho de Código A.1 apresenta o cabeçalho do módulo, no qual são declarados seus símbolos globais e incluídos os cabeçalhos de cada uma das partições, e o Trecho de Código A.2 apresenta o código do módulo, que contém os atributos das partições, o código de inicialização (preparação da plataforma e criação das partições), e ainda o código da partição padrão/ociosa do módulo.

Trecho de Código A.1 – Cabeçalho de módulo (module.h)

```
1  #ifndef MODULE_H_
2  #define MODULE_H_
3
4  // ARINC653 includes
5  #include "arinc653.h"
6
7  // Partitions includes
8  #include "partitionX.h"
9
10 // Module default partition
11 extern void MODULE_DEFAULTPARTITION(void);
12
13 #endif
```

Trecho de Código A.2 – Código de módulo (module.c)

```
1  // Module include
2  #include "module.h"
3
4  // Configuration include
5  #include "configuration.h"
6
7  // PARTITIONX partition attributes
8  static PARTITION_ATTRIBUTE_TYPE PARTITIONX_PARTITION_ATTRIBUTE={
9  /*NAME*/(PARTITION_NAME_TYPE) "PARTITIONX",
10 /*ENTRY_POINT*/(SYSTEM_ADDRESS_TYPE) &PARTITIONX_DEFAULTPROCESS,
11 /*STACK_SIZE*/(STACK_SIZE_TYPE) 256,
12 /*SYSTEMPARTITION*/(BOOLEAN_TYPE) false };
13
14 // PARTITIONX partition identifier
```

```
15 static PARTITION_ID_TYPE PARTITIONX_PARTITION_ID;
16
17 // Module default partition
18 void MODULE_DEFAULTPARTITION(void) {
19     RETURN_CODE_TYPE RETURN_CODE;
20
21     // Starts system up
22     STARTUP_SYSTEM(&RETURN_CODE);
23
24     // Starts module up
25     STARTUP_MODULE(&RETURN_CODE);
26
27     // Creates PARTITIONX partition
28     CREATE_PARTITION(&PARTITIONX_PARTITION_ATTRIBUTE,
           &PARTITIONX_PARTITION_ID, &RETURN_CODE);
29
30     // Sets module mode
31     SET_MODULE_MODE(NORMAL, &RETURN_CODE);
32
33     // Idle partition code
34 }
```

A.2 PARTIÇÃO

O código fonte da partição engloba o código de seus processos e a inicialização da partição (criação e inicialização dos processos). O Trecho de Código A.3 apresenta o cabeçalho da partição, no qual são declarados seus símbolos globais, e o Trecho de Código A.4 apresenta o código da partição, que contém os atributos e o código de seus processos e o código do processo padrão/ocioso da partição.

Trecho de Código A.3 – Cabeçalho de partição (partitionX.h)

```
1 #ifndef PARTITIONX_H_
2 #define PARTITIONX_H_
3
4 // ARINC653 includes
5 #include "arinc653.h"
6
7 // PARTITIONX partition default process
8 extern void PARTITIONX_DEFAULTPROCESS(void);
9
10 #endif
```

Trecho de Código A.4 – Código de partição (partitionX.c)

```
1 // Partition include
2 #include "partitionX.h"
3
4 // PROCESSY process identifier
5 static PROCESS_ID_TYPE PROCESSY_PROCESS_ID;
```

```
6
7 // ------------------- PROCESSY PROCESS START -------------------
8
9 // PROCESSY process
10 static void PROCESSY(void) {
11     RETURN_CODE_TYPE RETURN_CODE;
12
13     // Main loop
14     while (true) {
15
16         // PROCESSY process code
17
18         // Waits for next period
19         PERIODIC_WAIT(&RETURN_CODE);
20     }
21 }
22
23 // PROCESSY process attributes
24 static PROCESS_ATTRIBUTE_TYPE PROCESSY_PROCESS_ATTRIBUTE = {
25 /*NAME*/(PROCESS_NAME_TYPE) "PROCESSY",
26 /*ENTRY_POINT*/(SYSTEM_ADDRESS_TYPE) &PROCESSY,
27 /*STACK_SIZE*/(STACK_SIZE_TYPE) 256,
28 /*BASE_PRIORITY*/(PRIORITY_TYPE) 10,
29 /*PERIOD*/(SYSTEM_TIME_TYPE) 500000000,
30 /*TIME_CAPACITY*/(SYSTEM_TIME_TYPE) 250000000,
31 /*DEADLINE*/(DEADLINE_TYPE) HARD };
32
33 // -------------------- PROCESSY PROCESS END --------------------
34
35 // PARTITIONX partition default process
36 void PARTITIONX_DEFAULTPROCESS(void) {
37     RETURN_CODE_TYPE RETURN_CODE;
38
39     // Creates PROCESSY process
40     CREATE_PROCESS(&PROCESSY_PROCESS_ATTRIBUTE,
           &PROCESSY_PROCESS_ID, &RETURN_CODE);
41
42     // Starts PROCESSY process
43     START(PROCESSY_PROCESS_ID, &RETURN_CODE);
44
45     // Sets partition mode
46     SET_PARTITION_MODE(NORMAL, &RETURN_CODE);
47
48     // Idle process code
49 }
```

APÊNDICE B – Exemplo de execução

Este apêndice tem por objetivo fornecer um exemplo completo das operações necessárias à execução de aplicações sobre o SO desenvolvido neste trabalho, assim como à manutenção dos modelos de aplicação utilizados nesse processo. Para isso, explica-se a utilização tanto das ferramentas oferecidas pelo fabricante do processador quanto daquelas desenvolvidas neste trabalho, apresentadas respectivamente no Apêndice C e no Capítulo 5. Apresenta-se a seguir os recursos que são necessários nos processos de **preparação do ambiente**, **compilação e execução** e de **gerenciamento de modelos** que serão exemplificados nas próximas seções:

- A *IDE TI Code Composer Studio* versão 6 ou posterior com compilador *TI ARM Compiler* versão 5.1 ou posterior é necessária para a compilação do SO e da aplicação;
- A plataforma *Java* versão 8 ou posterior é necessária para execução das ferramentas de configuração;
- O arquivo **MLO** contendo a imagem de memória do *bootloader* da *BeagleBone*, fornecido no pacote *AM335X StarterWare*, é necessário para a execução do SO e da aplicação na plataforma;
- A ferramenta de configuração **TemplateGenerator** é necessária para a geração/regeração dos modelos de aplicação;
- A ferramenta de configuração **TemplateManager** é necessária para o gerenciamento de modelos de aplicação;
- Um modelo de arquivo *XML* de configuração de módulo, que pode ser obtido a partir de um dos casos de teste desenvolvidos neste trabalho;
- O código fonte do SO desenvolvido neste trabalho.

B.1 PREPARAÇÃO DO AMBIENTE

O processo de preparação do ambiente compreende os estágios iniciais da utilização do SO desenvolvido, ou seja, a preparação da *IDE* e a geração de um modelo de aplicação com base em um arquivo de configuração. Esse processo é executado através das seguintes operações:

1. Criar um projeto vazio na *IDE TI Code Composer Studio* configurado para utilizar o compilador *TI ARM Compiler* e destinado a processadores Cortex^{TM}-A8, ou importá-lo a partir do código fonte do SO (se possível);

2. Se necessário, copiar para dentro do projeto o código fonte do SO desenvolvido neste trabalho, que é composto dos diretórios **ARINC653**, **ARINC653_PORT** e **Hardware**;
3. Se necessário, configurar os seguintes aspectos do projeto:
    - Adicionar referências de inclusão a todos os diretórios **include** do projeto;
    - Habilitar o suporte ao coprocessador de ponto flutuante *VFPv3*;
    - Atribuir o tamanho da memória de alocação dinâmica (*heap*) da linguagem C para zero;
    - Atribuir o tamanho da pilha do sistema para 16384 *bytes* (16KB);
    - Desabilitar a inicialização do ambiente de execução (*runtime environment*) da linguagem C;
    - Incluir os símbolos predefinidos **am335x** e **beaglebone**, que são exigidos por algumas das bibliotecas fornecidas no pacote *AM335X StarterWare* utilizadas pelo SO.
4. Criar um diretório chamado **Application** no projeto;
5. Executar a ferramenta **TemplateGenerator** sobre o arquivo *XML* de configuração de módulo, gerando o modelo inicial da aplicação dentro do diretório **Application** do projeto, através do seguinte comando:

   TemplateGenerator module.xml AM335X Application

6. Executar a ferramenta **TemplateManager** sobre o diretório **Application**, expandindo os demarcadores de regiões de código criados pelo processo de geração do modelo para que possam ser preenchidos, através do seguinte comando:

   TemplateManager C EXPAND Application TEMPLATE.ART “.*\.c” “” YES

B.2 COMPILAÇÃO E EXECUÇÃO

O processo de compilação e execução compreende a geração de uma imagem de memória a partir do arquivo *ELF* executável gerado pela compilação do projeto na *IDE*, assim como a execução dessa ima-

gem na plataforma de *hardware*. Esse processo é executado através das seguintes operações:

1. Utilizar a *IDE* para efetuar a compilação do projeto, gerando o arquivo *ELF* executável denominado **PROJETO.out**;
2. Executar a ferramenta **armofd** sobre o arquivo **PROJETO.out**, gerando o arquivo **PROJETO.xml** que contém informações sobre os elementos do arquivo *ELF*, através do seguinte comando:

   armofd -x –obj_display=none,header,sections,segments PROJETO.out > PROJETO.xml

3. Executar a ferramenta **mkhex4bin** sobre o arquivo **PROJETO.xml**, gerando o arquivo **PROJETO.add** que contém o endereço efetivo e o tamanho total da imagem de memória do programa, através do seguinte comando:

   mkhex4bin PROJETO.xml > PROJETO.add

4. Executar a ferramenta **armhex** sobre os arquivos **PROJETO.out** e **PROJETO.add**, gerando o arquivo **PROJETO.bin** que contém a imagem de memória do programa, através do seguinte comando:

   armhex -q -b -image -o PROJETO.bin PROJETO.add PROJETO.out

5. Executar a ferramenta **tiimage** sobre o arquivo **PROJETO.bin**, gerando o arquivo **APP** para carregamento e execução do programa através do *bootloader* da *BeagleBone*, através do seguinte comando:

   tiimage 0x80000000 NONE PROJETO.bin APP

6. Copiar o arquivo **APP**, gerado através da ferramenta **tiimage**, para o diretório raiz de um cartão *SD*;
7. Copiar o arquivo **MLO** do *bootloader* da *BeagleBone*, fornecido no pacote *AM335X StarterWare*, para o diretório raiz do cartão *SD*;

8. Inserir o cartão *SD* no conector da *BeagleBone* e acioná-la, iniciando a execução do programa.

B.3 GERENCIAMENTO DE MODELOS

O processo de gerenciamento de modelos compreende a regeração de modelos de aplicação, mantendo porém inalterado o código inserido pelo usuário nas regiões de código demarcadas. Esse processo deve ser executado sempre que forem necessárias alterações no arquivo de configuração do módulo, e é realizado através das seguintes operações:

1. Executar a ferramenta **TemplateManager** sobre o diretório **Application**, armazenando as regiões de código do modelo de aplicação no arquivo de artefato **TEMPLATE.ART**, através do seguinte comando:

   TemplateManager C COLLAPSE Application TEMPLATE.ART ".*\.c" "" YES

2. Executar a ferramenta **TemplateGenerator** sobre o arquivo de configuração, regerando o modelo da aplicação, através do seguinte comando:

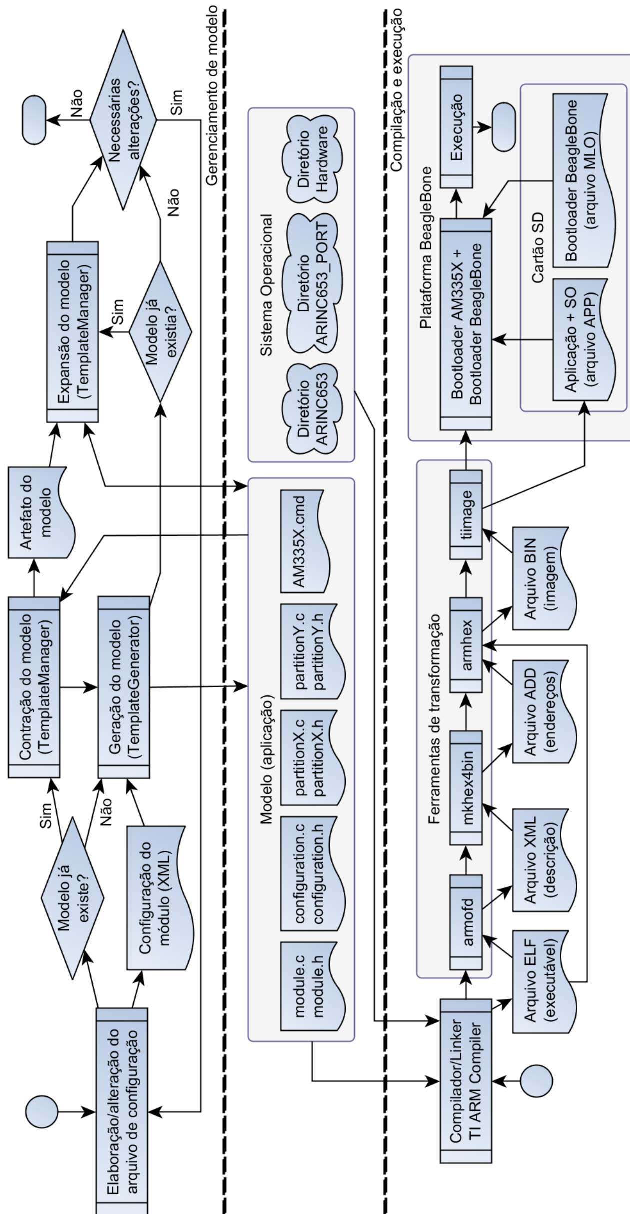
   TemplateGenerator module.xml AM335X Application

3. Executar a ferramenta **TemplateManager** sobre o diretório **Application**, restaurando as regiões de código anteriormente armazenadas no arquivo **TEMPLATE.ART**, através do seguinte comando:

   TemplateManager C EXPAND Application TEMPLATE.ART ".*\.c" "" YES

Ilustra-se na Figura 13 a integração entre os processos de gerenciamento de modelos de aplicação e de compilação e execução desses modelos no SO desenvolvido neste trabalho, representando a utilização de todas as ferramentas neles envolvidas.

Figura 13 – Gerenciamento, compilação e execução de modelos de aplicação



APÊNDICE C – Implementação

A apresentação de alguns aspectos do SO desenvolvido neste trabalho considerados secundários ou triviais para sua compreensão é realizada neste apêndice, com o objetivo de tornar o Capítulo 4 mais sucinto. Esses aspectos são, portanto, abordados nas próximas seções.

C.1 TIPOS DE DADOS E CONSTANTES

Um conjunto de tipos de dados genéricos é definido a fim de garantir a portabilidade do SO a diferentes arquiteturas, além de tornar explícito o fim ao qual as variáveis empregadas servem e os valores que podem ser por elas armazenados. Esses tipos genéricos são mapeados a tipos específicos suportados pelo compilador utilizado, sendo que esses podem eventualmente ser mais abrangentes (suportar uma faixa de valores maior), mas nunca menos abrangentes, que os tipos genéricos associados. Esses tipos genéricos são apresentados a seguir, juntamente com o tipo específico ao qual são mapeados no SO desenvolvido neste trabalho.

portBOOLEAN (unsigned char) Armazena um valor lógico (verdadeiro/falso);

portCHARACTER (unsigned char) Armazena um caractere de um segmento de texto;

portBYTE (unsigned char) Armazena um *byte* de um segmento de dados;

portINT8 e portUINT8 (signed char e unsigned char) Armazenam valores inteiros de 8 *bits* com e sem sinal, nos intervalos $[-2^7, 2^7 - 1]$ e $[0, 2^8 - 1]$, respectivamente;

portINT16 e portUINT16 (signed short e unsigned short) Armazenam valores inteiros de 16 *bits* com e sem sinal, nos intervalos $[-2^{15}, 2^{15} - 1]$ e $[0, 2^{16} - 1]$, respectivamente;

portINT32 e portUINT32 (signed int e unsigned int) Armazenam valores inteiros de 32 *bits* com e sem sinal, nos intervalos $[-2^{31}, 2^{31} - 1]$ e $[0, 2^{32} - 1]$, respectivamente;

portINT64 e portUINT64 (signed long long int e unsigned long long int) Armazenam valores inteiros de 64 *bits* com e sem sinal, nos intervalos $[-2^{63}, 2^{63} - 1]$ e $[0, 2^{64} - 1]$, respectivamente;

portINTBASE e portUINTBASE (signed int e unsigned int) Armazenam valores inteiros de tamanho igual ao de uma palavra de memória na arquitetura empregada (no caso 32 *bits*) com e

sem sinal, respectivamente;

portUINTPOINTER (unsigned int) Armazena um valor inteiro sem sinal de tamanho igual ao de um endereço de memória (ponteiro) da arquitetura utilizada (no caso 32 *bits*), utilizado para cálculo de endereços;

portADDRESS (void*) Armazena um endereço de memória;

portSTACKROW (unsigned int) Armazena um valor inteiro de tamanho igual àquele empregado por operações de pilha na arquitetura utilizada (no caso 32 *bits*);

portPARAMETER (unsigned int) Armazena um valor inteiro de tamanho igual àquele empregado para passagem de parâmetros para métodos na arquitetura utilizada (no caso 32 *bits*);

portSIZE (unsigned int) Armazena um valor inteiro sem sinal que denota o tamanho de um vetor ou de um segmento de texto ou de dados;

portINDEX (unsigned int) Armazena um valor inteiro sem sinal que denota um índice de um vetor ou de um segmento de texto ou de dados;

portIDENTIFIER (unsigned int) Armazena um valor inteiro sem sinal que é utilizado para identificação de elementos do SO, e portanto cujo tamanho limita o número de elementos suportados, sendo que variáveis deste tipo podem assumir o valor distinto definido pela constante **INVALID_IDENTIFIER**, que indica que o identificador é inválido, foi omitido, ou que o elemento correspondente não existe.

A especificação ARINC 653 define tipos de dados que são empregados tanto na interface com o *software* de aplicação quanto na implementação dos serviços fornecidos através dela. Apresenta-se a seguir o mapeamento desses tipos de dados aos tipos genéricos definidos pelo SO:

SYSTEM_TIME_TYPE Mapeado a **portINT64**;

LOCK_LEVEL_TYPE Mapeado a **portUINTBASE**;

STACK_SIZE_TYPE Mapeado a **portSIZE**;

SYSTEM_ADDRESS_TYPE Mapeado a **portADDRESS**;

PRIORITY_TYPE Mapeado a **portUINTBASE**;

PARTITION_ID_TYPE Mapeado a **portIDENTIFIER**;

PARTITION_NAME_TYPE Mapeado a **portCHARACTER**[];

PROCESS_ID_TYPE Mapeado a **portIDENTIFIER**;

PROCESS_NAME_TYPE Mapeado a **portCHARACTER[]**;
SAMPLING_PORT_ID_TYPE Mapeado a **portIDENTIFIER**;
SAMPLING_PORT_NAME_TYPE Mapeado a **portCHARACTER[]**;
QUEUING_PORT_ID_TYPE Mapeado a **portIDENTIFIER**;
QUEUING_PORT_NAME_TYPE Mapeado a **portCHARACTER[]**;
BUFFER_ID_TYPE Mapeado a **portIDENTIFIER**;
BUFFER_NAME_TYPE Mapeado a **portCHARACTER[]**;
BLACKBOARD_ID_TYPE Mapeado a **portIDENTIFIER**;
BLACKBOARD_NAME_TYPE Mapeado a **portCHARACTER[]**;
SEMAPHORE_ID_TYPE Mapeado a **portIDENTIFIER**;
SEMAPHORE_NAME_TYPE Mapeado a **portCHARACTER[]**;
SEMAPHORE_VALUE_TYPE Mapeado a **portUINTBASE**;
EVENT_ID_TYPE Mapeado a **portIDENTIFIER**;
EVENT_NAME_TYPE Mapeado a **portCHARACTER[]**;
MESSAGE_RANGE_TYPE Mapeado a **portSIZE**;
WAITING_RANGE_TYPE Mapeado a **portSIZE**;
MESSAGE_ADDR_TYPE Mapeado a **portCHARACTER[]**;
MESSAGE_SIZE_TYPE Mapeado a **portSIZE**;
ERROR_MESSAGE_TYPE Mapeado a **portCHARACTER[]** com tamanho **MAX_ERROR_MESSAGE_SIZE**;
ERROR_MESSAGE_SIZE_TYPE Mapeado a **portSIZE**.

A fim de manter-se um padrão de nomenclatura de tipos de dados semelhante a esse utilizado pela especificação ARINC 653, os seguintes tipos são definidos para utilização na porção genérica do código fonte do SO:

BOOLEAN_TYPE O mesmo que **portBOOLEAN**;
CHARACTER_TYPE O mesmo que **portCHARACTER**;
UINTBASE_TYPE O mesmo que **portUINTBASE**;
UINTPOINTER_TYPE O mesmo que **portUINTPOINTER**;
STACKROW_TYPE O mesmo que **portSTACKROW**;
PARAMETER_TYPE O mesmo que **portPARAMETER**;
SIZE_TYPE O mesmo que **portSIZE**;

INDEX_TYPE O mesmo que **portINDEX**;
IDENTIFIER_TYPE O mesmo que **portIDENTIFIER**.

As seguintes constantes são definidas pela especificação ARINC 653, e tiveram seus valores mantidos conforme a declaração apresentada para a linguagem C na própria especificação (ARINC, 2006a):

MAX_NAME_LENGTH (30) Comprimento máximo dos nomes de elementos;
MAX_LOCK_LEVEL (16) Valor máximo que o nível de bloqueio de uma partição pode assumir;
MIN_PRIORITY_VALUE (1) Valor mínimo das prioridades de processos;
MAX_PRIORITY_VALUE (63) Valor máximo das prioridades de processos;
MAX_ERROR_MESSAGE_SIZE (64) Comprimento máximo das mensagens de erro;
INFINITE_TIME_VALUE (-1) Valor distinto que indica um intervalo de tempo infinito, que pode também ser interpretado como inválido.

Conforme apresentado no Capítulo 4 e nas demais seções deste apêndice, além desses tipos de dados simples e constantes são também utilizadas no SO desenvolvido diversas enumerações. Destaca-se que todas essas enumerações possuem valores definidos de forma que zero (**null**) não represente uma opção válida, e assim esse valor possa ser utilizado para denotar a omissão da informação correspondente. Ou seja, todos os valores associados às enumerações declaradas no código fonte do SO desenvolvido neste trabalho devem ser maiores ou iguais a um, pois o valor zero é considerado ausência de informação nesse contexto.

C.2 SERVIÇOS INTERNOS

Alguns serviços são definidos para uso exclusivo no núcleo do SO e/ou na implementação dos serviços exigidos pela ARINC 653, e não estão portanto disponíveis ao *software* de aplicação. Apresenta-se a seguir esses serviços, categorizados de acordo com a função à qual estão relacionados:

- Comuns:

COMMON_COPYSTRING Efetua a cópia de um segmento de texto finalizado em caractere nulo de tamanho máximo conhecido de um endereço de memória para outro;

COMMON_COMPARESTRINGS Efetua a comparação de dois segmentos de texto finalizados em caractere nulo de tamanho máximo conhecido, retornando um valor lógico que indica sua identidade.

- Configuração:

CONFIGURATION_GETPARTITIONCONFIGURATION Busca na estrutura de configuração do módulo a configuração de uma partição a partir de seu nome;

CONFIGURATION_GETPROCESSCONFIGURATION Busca em uma estrutura de configuração de partição a configuração de um processo dessa partição a partir de seu nome.

- Relógio:

CLOCK_STARTUP Inicializa as variáveis do relógio do módulo com horário zero, e calcula o horário de início do próximo *major frame*;

CLOCK_TICKWINDOW É invocado pela interrupção periódica de escalonamento para atualização do relógio do módulo, somando ao horário atual do módulo o período dessa interrupção;

CLOCK_TICKMAJORFRAME Calcula o horário de início do próximo *major frame*;

CLOCK_GETSYSTEMTIME Consulta o horário atual do módulo.

- Gerenciamento de partições:

GET_PARTITION_INTERNAL_IDENTIFIER Partições possuem um identificador interno, atribuído pelo SO no momento de sua criação, e um identificador externo, que é aquele utilizado no arquivo de configuração do módulo para referência à partição; este serviço é utilizado para obtenção do identificador interno de uma partição a partir de seu identificador externo.

- Gerenciamento de processos:

WAIT_RESOURCE É responsável por efetuar o bloqueio de um processo por tempo limitado ou ilimitado em um determinado recurso do SO (semáforo ou evento, por exemplo), ou seja, este serviço remove o processo a partir do qual foi invocado da fila de processos prontos, insere-o na fila do recurso e invoca o escalonador; se for o caso, insere o processo na fila de bloqueio por tempo limitado;

SIGNAL_RESOURCE Libera um processo bloqueado através de uma chamada ao serviço **WAIT_RESOURCE**, ou seja, remove o processo da cabeceira da fila do recurso e, caso esse não encontre-se suspenso, insere-o na fila de processos prontos e invoca o escalonador; se for o caso, remove o processo da fila de bloqueio por tempo limitado.

- Monitoramento (*Health Monitoring*):

RAISE_ERROR Dispara os mecanismos de tratamento para um determinado erro detectado pelo SO de acordo com a configuração do módulo, ou seja, consulta as estruturas de configuração do mecanismo de monitoramento para determinar em que nível (processo, partição ou módulo) o erro deve ser tratado em função do estado atual do sistema, delegando esse tratamento aos serviços **RAISE_PROCESS_ERROR**, **RAISE_PARTITION_ERROR** ou **RAISE_MODULE_ERROR**, respectivamente;

RAISE_PROCESS_ERROR Dispara o mecanismo de tratamento de erros para o nível de processo, ou seja, caso um processo tratador de erros exista para a partição sua execução é iniciada, e em caso contrário o tratamento do erro é redirecionado para o nível de partição (serviço **RAISE_PARTITION_ERROR**), conforme definido pela especificação (ARINC, 2006a);

RAISE_PARTITION_ERROR Dispara o mecanismo de tratamento de erros para o nível de partição, ou seja, caso um *HM callback* exista para a partição sua execução é iniciada, e em caso contrário ou caso trate-se de um erro ocorrido durante a execução do *HM callback* da partição, a ação definida para o erro no nível de partição é executada imediatamente através do serviço **RUN_PARTITION_ERROR_ACTION**;

RAISE_MODULE_ERROR Dispara o mecanismo de tratamento de erros para o nível de módulo, ou seja, caso um *HM*

callback exista para o módulo sua execução é iniciada, e em caso contrário a ação definida para o erro no nível de módulo é executada imediatamente através do serviço **RUN_MODULE_ERROR_ACTION**; caso trate-se de um erro ocorrido durante a execução do *HM callback* do módulo, a ação definida para o erro no nível de módulo é imediatamente executada e, caso essa defina que o erro deve ser ignorado, a ação definida para o erro no nível de sistema é disparada através do serviço **RUN_SYSTEM_ERROR_ACTION**;

RUN_PARTITION_ERROR_ACTION Executa a ação definida para um determinado erro no nível de partição, ou seja, consulta a ação a ser executada nas estruturas de configuração e, em função dela, ignora o erro, para a partição ou a reinicializa;

RUN_MODULE_ERROR_ACTION Executa a ação definida para um determinado erro no nível de módulo, ou seja, consulta a ação a ser executada nas estruturas de configuração e, em função dela, ignora o erro, para o módulo ou o reinicializa;

RUN_SYSTEM_ERROR_ACTION Executa a ação definida para um determinado erro no nível de sistema, ou seja, consulta a ação a ser executada nas estruturas de configuração e, em função dela, para ou reinicializa o módulo;

STOP_MODULE_HEALTHMONITORINGCALLBACK Termina a execução do *HM callback* do módulo, retomando o escalonamento de partições;

STOP_PARTITION_HEALTHMONITORINGCALLBACK Termina a execução do *HM callback* de partição, retomando o escalonamento dos processos da partição.

- Comunicação interpartição - Portas de amostragem:

GET_PARTITION_SAMPLING_PORT_ID Obtém o identificador de uma porta de amostragem de uma determinada partição com base no nome da porta;

WRITE_PARTITION_SAMPLING_MESSAGE Escreve uma mensagem em uma partição de destino de uma porta de amostragem;

READ_PARTITION_SAMPLING_MESSAGE Lê uma mensagem da partição de origem de uma porta de amostragem para que seja efetuada sua transmissão.

- Comunicação interpartição - Portas de enfileiramento:

GET_PARTITION_QUEUING_PORT_ID Obtém o identificador de uma porta de enfileiramento de uma determinada partição com base no nome da porta;

SEND_PARTITION_QUEUING_MESSAGE Adiciona uma mensagem à fila da partição de destino de uma porta de enfileiramento, liberando um processo que eventualmente esteja bloqueado à espera de uma mensagem nessa porta;

RECEIVE_PARTITION_QUEUING_MESSAGE Lê uma mensagem da fila da partição de origem de uma porta de enfileiramento para que seja efetuada sua transmissão.

C.3 SERVIÇOS ESPECÍFICOS

Trata-se de serviços que são disponibilizados ao *software* de aplicação pelo SO desenvolvido e que não fazem parte da ARINC 653, porém cuja utilização é fundamental para a inicialização e a manutenção dos elementos do sistema. Esses serviços são apresentados a seguir, categorizados de acordo com a função à qual estão ligados:

- Gerenciamento do sistema:

STARTUP_SYSTEM Efetua a inicialização do sistema, ou seja, realiza a configuração básica da plataforma de *hardware* e prepara a região de alocação dinâmica de memória do sistema, e deve portanto ser a primeira função chamada quando a plataforma é acionada.

- Gerenciamento do módulo:

STARTUP_MODULE Efetua a inicialização do módulo, ou seja, prepara o contexto de execução inicial e aloca memória para armazenamento das informações das partições;

SET_MODULE_MODE Efetua a transição de modos do módulo de forma análoga ao serviço **SET_PARTITION_MODE** definido pela especificação para o gerenciamento de partições, podendo portanto ser utilizado para inicializar o escalonamento de partições do módulo (modo **NORMAL**), assim como para solicitar sua parada ou reinicialização (modos **IDLE** e **COLD_START/WARM_START**, respectivamente).

- Gerenciamento de partições:

GET_PARTITION_ID Obtém o identificador de uma partição a partir de seu nome, de forma análoga ao serviço **GET_PROCESS_ID** definido pela especificação para o gerenciamento de processos;

CREATE_PARTITION Utilizado pela partição padrão do módulo durante sua etapa de inicialização para efetuar a criação de partições, podendo portanto ser invocado apenas antes da chamada ao serviço **SET_MODULE_MODE** para transição ao modo de execução **NORMAL**; as partições do módulo são criadas com base em um conjunto de atributos contidos em uma estrutura do tipo **PARTITION_ATTRIBUTE_TYPE**, análoga à estrutura **PROCESS_ATTRIBUTE_TYPE** definida pela especificação para o gerenciamento de processos, que contém informações como o nome da partição, algumas informações relacionadas ao seu processo padrão, e ainda um campo que indica se a partição é ou não uma partição de sistema.

C.4 SERVIÇOS ARINC 653

Apresenta-se a seguir alguns detalhes relevantes da implementação desenvolvida neste trabalho dos serviços definidos pela ARINC 653, sendo porém omitidas todas as informações que decorrem diretamente da especificação e que podem portanto ser nela consultadas (ARINC, 2006a).

- Gerenciamento de partições:

CREATE_PARTITION Antes de efetivar a criação da partição, este serviço executa as seguintes operações:

1. Valida os atributos fornecidos (o nome da partição, por exemplo);
2. Valida as configurações fornecidas (a coerência da escala da partição, por exemplo);
3. Inicializa os contextos de execução da partição (do processo padrão, do processo tratador de erros e do *HM callback* da partição, caso existam);
4. Inicializa o mecanismo de alocação dinâmica de memória da partição;

5. Aloca e inicializa as estruturas que armazenam informações relacionadas a todos os elementos da partição (processos e mecanismos de comunicação, por exemplo);
6. Armazena uma cópia da região de memória de dados da partição, que é utilizada para restauração dessa região em caso de reinicialização da partição através do serviço **SET_PARTITION_MODE**.

SET_PARTITION_MODE Este serviço apresenta diferentes comportamentos em função do modo ao qual é solicitada transição:

NORMAL Inicializa o escalonamento de processos da partição;

COLD_START/WARM_START A fim de realizar a reinicialização da partição, executa as seguintes operações:

1. Reinicializa as estruturas que armazenam informações sobre os elementos da partição, porém mantendo inalteradas informações relacionadas à memória alocada para esses elementos, evitando assim sua realocação;
2. Reinicializa o contexto de execução do processo padrão da partição;
3. Restaura a região de memória de dados da partição com base na cópia armazenada pelo serviço **CREATE_PARTITION** no momento de sua criação;
4. Redireciona o fluxo de execução para o processo padrão da partição, efetivando assim sua reinicialização.

IDLE Reinicializa as estruturas que armazenam as informações da partição, removendo-a assim da fila de partições prontas, e então invoca o algoritmo de escalonamento – a partição não é adicionada novamente à fila por encontrar-se em modo de execução **IDLE**.

- Gerenciamento de processos:

CREATE_PROCESS Processos aperiódicos devem ser diferenciados de processos periódicos através da utilização de um valor distinto de período (ARINC, 2006a), que no SO implementado neste trabalho é definido pela constante **APERIODIC_PERIOD_VALUE** cujo valor é zero;

SET_PRIORITY De acordo com a especificação, este serviço pode ou não reordenar as filas de prioridade associadas a recursos bloqueantes do SO (semáforos e eventos, por exemplo) caso a prioridade de um processo bloqueado seja alterada por outro processo

(ARINC, 2006a), e portanto é fundamental ressaltar que no SO desenvolvido neste trabalho essas filas são reordenadas nessa situação;

START A especificação define que este serviço equivale ao **DELAYED_START** invocado com **DELAY_TIME** zero (ARINC, 2006a), e portanto sua implementação neste trabalho consiste simplesmente de uma chamada a esse outro serviço.

- Comunicação interpartição:

CREATE_SAMPLING_PORT e CREATE_QUEUING_PORT Estes serviços alocam dinamicamente porções de memória pertencentes à partição para o armazenamento do nome dos elementos de comunicação interpartição e das mensagens por eles manipuladas; o nome desses elementos (recebido por parâmetro pelos serviços) é copiado para uma região de memória de dados própria da partição pois, em alguns casos, pode estar localizado em uma região de memória não pertencente à partição – até mesmo em uma região de código – e, portanto, o acesso direto pode ser considerado uma violação do isolamento espacial de partições; ressalta-se ainda que estes serviços não verificam se os elementos correspondentes foram declarados de forma consistente no arquivo de configuração do módulo, uma vez que considera-se que esse arquivo foi devidamente validado antes de ser utilizado;

GET_SAMPLING_PORT_ID e GET_QUEUING_PORT_ID Estes serviços fazem uso dos serviços internos **GET_PARTITION_SAMPLING_PORT_ID** e **GET_PARTITION_QUEUING_PORT_ID**, respectivamente;

SEND_QUEUING_MESSAGE e RECEIVE_QUEUING_MESSAGE Estes serviços utilizam os serviços internos **WAIT_RESOURCE** e **SIGNAL_RESOURCE** para gerenciamento do bloqueio de processos em portas de enfileiramento.

- Comunicação intrapartição:

CREATE_BUFFER, CREATE_BLACKBOARD, CREATE_SEMAPHORE e CREATE_EVENT Estes serviços alocam dinamicamente porções de memória pertencentes à partição para o armazenamento do nome dos elementos de comunicação intrapartição e das mensagens por eles manipuladas, pelos mesmos motivos citados para os serviços

CREATE_SAMPLING_PORT e **CREATE_QUEUING_PORT**;

SEND_BUFFER, RECEIVE_BUFFER, DISPLAY_BLACKBOARD, READ_BLACKBOARD, WAIT_SEMAPHORE, SIGNAL_SEMAPHORE, SET_EVENT e WAIT_EVENT Estes serviços utilizam os serviços internos **WAIT_RESOURCE** e **SIGNAL_RESOURCE** para gerenciamento do bloqueio de processos pelos elementos de comunicação intrapartição.

- Monitoramento (*Health Monitoring*):

REPORT_APPLICATION_MESSAGE O tamanho máximo das mensagens reportadas através deste serviço são definidas pela constante **MAX_APPLICATION_MESSAGE_SIZE**;

RAISE_APPLICATION_ERROR Este serviço obtém o endereço a partir do qual foi invocado pela aplicação através dos métodos **PORT_PREPARECALLADDRESS** e **PORT_GETCALLADDRESS** (apresentados a seguir), e em seguida utiliza o serviço interno **RAISE_ERROR** para disparar o mecanismo de tratamento para erros de aplicação.

C.5 NÚCLEO

Esta seção tem por objetivo apresentar os elementos de código utilizados no núcleo do SO desenvolvido neste trabalho, e que assumem portanto papel fundamental no desempenho de suas principais funções. Por ser necessária a compreensão extensiva desses elementos para familiarização com o funcionamento dos serviços oferecidos pelo SO, cada um deles será abordado em profundidade nas próximas seções.

C.5.1 Estruturas de informações gerenciais

Trata-se de estruturas que armazenam informações gerenciais do núcleo do SO, ou seja, nas quais são armazenadas as principais informações sobre o estado dos elementos do sistema e que são, portanto, utilizadas pela maioria dos serviços oferecidos ao *software* de aplicação.

C.5.1.1 SYSTEM_INFORMATION_TYPE

Armazena informações gerenciais globais do sistema, e possui os seguintes atributos:

SYSTEM_CONFIGURATION Aponta a estrutura de configuração do sistema;

REC_HEAP Registro da região de alocação dinâmica de memória do sistema;

NEXT_CONTEXT_IDENTIFIER Armazena o próximo identificador de contexto de execução a ser alocado, e é utilizado somente durante a inicialização do sistema.

C.5.1.2 MODULE_INFORMATION_TYPE

Armazena informações gerenciais globais do módulo, e possui os seguintes atributos:

MODULE_CONFIGURATION Aponta a estrutura de configuração do módulo;

OPERATING_MODE Indica o modo de operação atual do módulo (análogo ao modo de operação de partição);

CONTEXT Estrutura que armazena o contexto de execução da partição padrão do módulo;

PARTITION_COUNT Indica quantas partições foram criadas no módulo através do serviço **CREATE_PARTITION**;

PARTITION_INFORMATION Vetor de estruturas de informações das partições do módulo (do tipo **PARTITION_INFORMATION_TYPE**), cujo índice é o identificador dessas partições;

CURRENT_PARTITION_INFORMATION Aponta a estrutura da partição que encontra-se atualmente em execução no módulo, e assume valor nulo (**null**) caso não haja uma partição em execução, ou seja, caso a partição padrão do módulo esteja em execução;

HEALTHMONITORINGCALLBACK_INFORMATION Aponta a estrutura de informações do *HM callback* do módulo;

MAJOR_FRAME_START Indica o horário no qual o *major frame* atual do módulo foi iniciado;

MAJOR_FRAME_TIME Indica o horário atual dentro do *major frame* atual do módulo, sendo que o horário do módulo (re-

tornado pelo serviço **CLOCK_GETSYSTEMTIME**) pode ser calculado por **MAJOR_FRAME_START + MAJOR_FRAME_TIME**;

NEXT_MAJOR_FRAME_START Indica o horário de início do próximo *major frame* do módulo, que é utilizado no escalonamento de processos;

REC_PARTITIONREADY Fila de prioridades de partições prontas do módulo – organiza as partições do módulo que encontram-se prontas para execução em ordem ascendente de horário de início da próxima janela de tempo na qual serão executadas (segundo a escala de partições configurada), e é utilizada no escalonamento de partições.

C.5.1.3 PARTITION_INFORMATION_TYPE

Armazena informações gerenciais de uma partição do módulo, e possui os seguintes atributos:

INITIALIZED Campo que indica se a partição já foi inicializada, e é utilizado para que os recursos alocados à partição não sejam realocados quando de sua reinicialização;

PARTITION_ATTRIBUTE Aponta a estrutura de atributos da partição;

PARTITION_CONFIGURATION Aponta a estrutura de configuração da partição;

IDENTIFIER Identificador interno da partição;

OPERATING_MODE Indica o modo de operação atual da partição;

START_CONDITION Indica a condição de inicialização da partição, diferenciando inicializações normais de reinicializações em decorrência de falhas, por exemplo;

LOCK_LEVEL Indica o nível de bloqueio da partição, de forma que o processo em execução não pode ser interrompido por outro processo da partição caso este campo tenha valor maior que zero;

HEALTHMONITORING_PARTITION_RESTART Durante a reinicialização da partição, indica se esse processo foi ou não disparado pelo mecanismo de monitoramento do SO (em decorrência de uma falha) – é utilizado na definição do atributo **START_CONDITION**;

REC_HEAP Registro da região de alocação dinâmica de memória da partição;

CONTEXT Estrutura que armazena o contexto de execução do processo padrão da partição;

PROCESS_COUNT Indica quantos processos foram criados na partição através do serviço **CREATE_PROCESS**;

PROCESS_INFORMATION Vetor de estruturas de informações dos processos da partição (do tipo **PROCESS_INFORMATION_TYPE**), cujo índice é o identificador desses processos;

CURRENT_PROCESS_INFORMATION Aponta a estrutura do processo que encontra-se atualmente em execução na partição, assumindo valor nulo (**null**) caso não haja um processo em execução, ou seja, caso o processo padrão da partição esteja em execução;

ERRORHANDLER_INFORMATION Aponta a estrutura de informações do processo tratador de erros da partição;

HEALTHMONITORINGCALLBACK_INFORMATION Aponta a estrutura de informações do *HM callback* da partição;

SCHEDULE_WINDOW_INDEX Índice da janela de tempo atual da partição na escala de partições;

ENT_PARTITIONREADY Entrada utilizada na fila de prioridades de partições prontas;

REC_PROCESSREADY Fila de prioridades de processos prontos da partição – organiza os processos da partição que encontram-se prontos para execução em ordem descendente de prioridade, e é utilizada no escalonamento de processos da partição;

REC_PROCESSDEADLINE Fila de prioridades de *deadlines* dos processos da partição – organiza os processos liberados da partição em ordem ascendente de *deadline*, e é utilizada na detecção de perdas de *deadline*;

REC_PROCESSWAITING_TIMEOUT Fila de prioridades de espera por *timeout* dos processos da partição – organiza os processos da partição em ordem ascendente de horário de *timeout*, e é utilizada no bloqueio de processos por tempo limitado;

ERROR_STATUS_COUNT Indica o número de entradas de erro atualmente armazenadas no vetor **ERROR_STATUS**;

NEXT_ERROR_STATUS_INDEX Índice da próxima entrada de erro do vetor **ERROR_STATUS** a ser lida;

ERROR_STATUS Vetor de entradas de erro a serem tratadas pelo processo tratador de erros da partição, utilizado de forma circular;

SAMPLINGPORT_COUNT Indica quantas portas de amostragem foram criadas na partição através do serviço **CREATE_SAMPLING_PORT**;

SAMPLINGPORT_INFORMATION Vetor de estruturas de informações das portas de amostragem da partição, cujo índice é o identificador dessas portas;

QUEUINGPORT_COUNT Indica quantas portas de enfileiramento foram criadas na partição através do serviço **CREATE_QUEUING_PORT**;

QUEUINGPORT_INFORMATION Vetor de estruturas de informações das portas de enfileiramento da partição, cujo índice é o identificador dessas portas;

BUFFER_COUNT Indica quantos *buffers* foram criados na partição através do serviço **CREATE_BUFFER**;

BUFFER_INFORMATION Vetor de estruturas de informações dos *buffers* da partição, cujo índice é o identificador desses *buffers*;

BLACKBOARD_COUNT Indica quantas *blackboards* foram criadas na partição através do serviço **CREATE_BLACKBOARD**;

BLACKBOARD_INFORMATION Vetor de estruturas de informações das *blackboards* da partição, cujo índice é o identificador dessas *blackboards*;

SEMAPHORE_COUNT Indica quantos semáforos foram criados na partição através do serviço **CREATE_SEMAPHORE**;

SEMAPHORE_INFORMATION Vetor de estruturas de informações dos semáforos da partição, cujo índice é o identificador desses semáforos;

EVENT_COUNT Indica quantos eventos foram criados na partição através do serviço **CREATE_EVENT**;

EVENT_INFORMATION Vetor de estruturas de informações dos eventos da partição, cujo índice é o identificador desses eventos.

C.5.1.4 PROCESS_INFORMATION_TYPE

Armazena informações gerenciais de um processo de uma partição, e possui os seguintes atributos:

PROCESS_ATTRIBUTE Aponta a estrutura de atributos do processo;

PROCESS_CONFIGURATION Aponta a estrutura de configuração do processo;

IDENTIFIER Identificador do processo;

CURRENT_PRIORITY Prioridade atual do processo;

PROCESS_STATE Estado atual do processo;

DELAY_TIME Tempo de atraso antes da primeira liberação do processo, definido através do serviço **DELAYED_START**;

RELEASE_TIME Horário da próxima (ou da última) liberação do processo;

DEADLINE_TIME Horário do próximo (ou do último) *deadline* do processo;

WAIT_TIME Horário no qual o processo entrou em espera pela última vez;

WAKE_TIME Horário no qual o processo saiu de espera pela última vez;

CONTEXT Estrutura que armazena o contexto de execução do processo;

ENT_PROCESSREADY Entrada utilizada na fila de prioridades de processos prontos da partição;

ENT_PROCESSDEADLINE Entrada utilizada na fila de prioridades de *deadlines* dos processos da partição;

ENT_PROCESSWAITING_TIMEOUT Entrada utilizada na fila de prioridades de espera por *timeout* dos processos da partição;

ENT_PROCESSWAITING_RESOURCE Entrada utilizada na fila de prioridades de espera por recursos (semáforos, por exemplo) da partição;

SUSPENDED Indica se o processo encontra-se suspenso;

SUSPENDED_TIMEOUT Indica se o processo encontra-se suspenso por tempo limitado (com *timeout*);

BUFFER_SEND_ADDRESS/LENGTH Armazenam o endereço e o tamanho da mensagem fornecida por uma chamada pendente em bloqueio ao serviço **SEND_BUFFER**, de forma que a cópia da mensagem possa ser realizada durante uma chamada ao serviço **RECEIVE_BUFFER** por outro processo;

BUFFER_RECEIVE_ADDRESS/LENGTH Armazenam o endereço e o tamanho da mensagem fornecida por uma chamada

pendente em bloqueio ao serviço **RECEIVE_BUFFER**, de forma que a cópia da mensagem possa ser realizada diretamente (sem enfileiramento) quando de uma chamada ao serviço **SEND_BUFFER** por outro processo.

C.5.1.5 HEALTHMONITORINGCALLBACK_INFORMATION_TYPE

Armazena informações gerenciais dos *HM callbacks* do módulo e das partições, e possui os seguintes atributos:

EXISTS Indica se o *HM callback* existe;

ENTRY_POINT Indica o ponto de entrada do método que implementa o *HM callback*;

STACK_SIZE Indica o tamanho da pilha alocada para o *HM callback*;

CONTEXT Estrutura que armazena o contexto de execução do *HM callback*;

SYSTEM_PARTITION_CONTEXT Indica se o *HM callback* foi disparado por um contexto de execução pertencente a uma partição de sistema, devendo portanto ser executado em modo privilegiado;

START Indica se o escalonador deve iniciar a execução do *HM callback*;

STOP Indica se o escalonador deve parar a execução do *HM callback*;

RUNNING Indica se o *HM callback* está em execução;

SYSTEM_STATE Armazena o estado do sistema no qual o erro que causou o disparo do *HM callback* ocorreu;

ERROR_IDENTIFIER Armazena o identificador do erro que causou o disparo do *HM callback*.

C.5.1.6 ERRORHANDLER_INFORMATION_TYPE

Armazena informações gerenciais dos processos tratadores de erros das partições, e possui os seguintes atributos:

EXISTS Indica se o processo tratador de erros existe;

ENTRY_POINT Indica o ponto de entrada do método que implementa o processo tratador de erros;

STACK_SIZE Indica o tamanho da pilha alocada para o processo tratador de erros;

CONTEXT Estrutura que armazena o contexto de execução do processo tratador de erros;

SYSTEM_PARTITION_CONTEXT Indica se o processo tratador de erros foi disparado por um contexto de execução pertencente a uma partição de sistema, devendo portanto ser executado em modo privilegiado;

START Indica se o escalonador deve iniciar a execução do processo tratador de erros;

STOP Indica se o escalonador deve parar a execução do processo tratador de erros;

RUNNING Indica se o processo tratador de erros está em execução;

PREEMPTED_PROCESS_IDENTIFIER Armazena o identificador do processo que foi interrompido para disparo do processo tratador de erros, e é utilizado pelos serviços **STOP** e **SUSPEND** – maiores detalhes sobre o emprego dessa informação devem ser consultados na especificação (ARINC, 2006a).

C.5.1.7 SAMPLINGPORT_INFORMATION_TYPE

Armazena informações gerenciais de uma porta de amostragem de uma partição, e possui os seguintes atributos:

IDENTIFIER Identificador da porta de amostragem;

NAME Nome da porta de amostragem;

MAX_MESSAGE_SIZE Tamanho máximo das mensagens trocadas através da porta de amostragem;

PORT_DIRECTION Indica se, na partição, a porta atua como origem ou como destino de mensagens;

REFRESH_PERIOD Indica o período de atualização da porta de amostragem;

UPDATED Indica se a mensagem da porta de amostragem foi atualizada desde a última vez que foi transmitida às partições de destino, devendo portanto ser transmitida novamente;

LAST_READ Indica o valor do relógio do módulo no momento em que a mensagem da porta de amostragem foi lida pela última vez – utilizado para controle das mensagens já transmitidas através do canal;

LAST_WRITE Indica o valor do relógio do módulo no momento em que a mensagem da porta de amostragem foi escrita pela última

vez – utilizado para controle da validade das mensagens transmitidas no canal;

LAST_VALIDITY Armazena o último indicador de validade retornado por uma chamada ao serviço **READ_SAMPLING_MESSAGE**;

EMPTY_INDICATOR Indica se a mensagem da porta de amostragem encontra-se vazia ou ocupada;

MESSAGE Armazena a mensagem atual da porta de amostragem;

LENGTH Armazena o tamanho da mensagem atual da porta de amostragem.

C.5.1.8 QUEUINGPORT_INFORMATION_TYPE

Armazena informações gerenciais de uma porta de enfileiramento de uma partição, e possui os seguintes atributos:

IDENTIFIER Identificador da porta de enfileiramento;

NAME Nome da porta de enfileiramento;

MAX_MESSAGE_SIZE Tamanho máximo das mensagens trocadas através da porta de enfileiramento;

MAX_NB_MESSAGE Número máximo de mensagens armazenadas na fila da porta de enfileiramento;

PORT_DIRECTION Indica se, na partição, a porta atua como origem ou como destino de mensagens;

QUEUING_DISCIPLINE Determina se os processos bloqueados pela porta de enfileiramento serão atendidos em política *FIFO* ou por prioridade;

MESSAGE_COUNT Número de mensagens atualmente armazenadas na fila da porta de enfileiramento;

NEXT_MESSAGE_INDEX Índice da próxima mensagem da fila a ser lida;

OVERFLOW Indica se ocorreu *overflow* na porta de enfileiramento desde a última vez em que ela foi lida;

MESSAGES Aponta o endereço inicial do segmento de dados de tamanho **MAX_MESSAGE_SIZE · MAX_NB_MESSAGE** que armazena a fila de mensagens da porta de enfileiramento;

MESSAGES_LENGTH Vetor do tipo **MESSAGE_SIZE_TYPE** de tamanho **MAX_NB_MESSAGE** que armazena o tamanho das mensagens atualmente armazenadas na fila da porta de enfi-

leiramento;

REC_QUEUINGPORT Fila de prioridades da porta de enfileiramento, utilizada para bloqueio de processos de acordo com a política indicada por **QUEUING_DISCIPLINE**.

C.5.1.9 BUFFER_INFORMATION_TYPE

Armazena informações gerenciais de um *buffer* de uma partição, e possui os seguintes atributos:

IDENTIFIER Identificador do *buffer*;

NAME Nome do *buffer*;

MAX_MESSAGE_SIZE Tamanho máximo das mensagens trocadas através do *buffer*;

MAX_NB_MESSAGE Número máximo de mensagens armazenadas na fila do *buffer*;

QUEUING_DISCIPLINE Determina se os processos bloqueados pelo *buffer* serão atendidos em política *FIFO* ou por prioridade;

MESSAGE_COUNT Número de mensagens atualmente armazenadas na fila do *buffer*;

NEXT_MESSAGE_INDEX Índice da próxima mensagem da fila a ser lida;

MESSAGES Aponta o endereço inicial do segmento de dados de tamanho **MAX_MESSAGE_SIZE · MAX_NB_MESSAGE** que armazena a fila de mensagens do *buffer*;

MESSAGES_LENGTH Vetor do tipo **MESSAGE_SIZE_TYPE** de tamanho **MAX_NB_MESSAGE** que armazena o tamanho das mensagens atualmente armazenadas na fila do *buffer*;

REC_BUFFER Fila de prioridades do *buffer*, utilizada para bloqueio de processos de acordo com a política indicada por **QUEUING_DISCIPLINE**.

C.5.1.10 BLACKBOARD_INFORMATION_TYPE

Armazena informações gerenciais de uma *blackboard* de uma partição, e possui os seguintes atributos:

IDENTIFIER Identificador da *blackboard*;

NAME Nome da *blackboard*;

EMPTY_INDICATOR Indica se a mensagem atual da *blackboard* encontra-se vazia ou ocupada;

MAX_MESSAGE_SIZE Tamanho máximo das mensagens trocadas através da *blackboard*;

MESSAGE Armazena a mensagem atual da *blackboard*;

LENGTH Armazena o tamanho da mensagem atual da *blackboard*;

REC_BLACKBOARD Fila de prioridades da *blackboard*, utilizada para bloqueio de processos.

C.5.1.11 SEMAPHORE_INFORMATION_TYPE

Armazena informações gerenciais de um semáforo de uma partição, e possui os seguintes atributos:

IDENTIFIER Identificador do semáforo;

NAME Nome do semáforo;

CURRENT_VALUE Valor atual do semáforo;

MAXIMUM_VALUE Valor máximo do semáforo;

QUEUING_DISCIPLINE Determina se os processos bloqueados pelo semáforo serão atendidos em política *FIFO* ou por prioridade;

REC_SEMAPHORE Fila de prioridades do semáforo, utilizada para bloqueio de processos de acordo com a política indicada por **QUEUING_DISCIPLINE**.

C.5.1.12 EVENT_INFORMATION_TYPE

Armazena informações gerenciais de um evento de uma partição, e possui os seguintes atributos:

IDENTIFIER Identificador do evento;

NAME Nome do evento;

EVENT_STATE Estado atual do evento (*up* ou *down*);

REC_EVENT Fila de prioridades do evento, utilizada para bloqueio de processos.

C.5.2 Símbolos

Apresenta-se a seguir os símbolos acessados e/ou manipulados por todos os métodos do núcleo do SO direta ou indiretamente, inclusive pelos serviços por ele oferecidos ao *software* de aplicação.

SYSTEM_INFORMATION Estrutura do tipo **SYSTEM_INFORMATION_TYPE** que armazena informações gerenciais do sistema;

MODULE_INFORMATION Estrutura do tipo **MODULE_INFORMATION_TYPE** que armazena informações gerenciais do módulo, das partições, dos processos e dos demais elementos do sistema;

CURRENT_CONTEXT Aponta a estrutura do tipo **CONTEXT_TYPE** na qual o contexto de execução atual é armazenado;

NEXT_CONTEXT Aponta a estrutura do tipo **CONTEXT_TYPE** do contexto de execução que deve ser retomado após a conclusão da execução do escalonador, e é portanto utilizada apenas durante a execução do escalonador;

_CURRENT_PARTITION_INFORMATION Aponta a estrutura de informações da partição que está atualmente em execução no SO, assumindo valor nulo (**null**) caso não haja uma partição em execução;

_CURRENT_PROCESS_INFORMATION Aponta a estrutura de informações do processo que está atualmente em execução no SO, assumindo valor nulo (**null**) caso não haja um processo em execução na partição atual – pode ser utilizado apenas quando há uma partição em execução, ou seja, quando **_CURRENT_PARTITION_INFORMATION** não tem valor nulo (**null**).

C.5.3 Métodos

Apresenta-se a seguir os métodos utilizados pelo núcleo do SO para tarefas como o gerenciamento da execução dos serviços oferecidos ao *software* de aplicação em modo privilegiado, manipulação das estruturas de informações gerenciais, e ainda para tratamento do retorno dos métodos executados pelos diversos elementos do sistema.

STARTUP_CORE Utilizado pelo serviço específico **STARTUP_SYSTEM** para inicialização dos símbolos globais do núcleo do SO;

ENTER_CORE Método invocado ao início da execução dos serviços do SO que desabilita as interrupções, entra em modo de execução privilegiado e incrementa a profundidade de chamadas de serviços do SO, garantindo assim que esses serviços sejam executados de forma atômica e privilegiada mesmo se invocados de forma aninhada – este método não deve acessar quaisquer dados antes de entrar em modo privilegiado;

EXIT_CORE Método invocado ao fim da execução de serviços do SO que decrementa a profundidade de chamadas de serviços do SO e, caso essa alcance valor zero, habilita as interrupções e retorna ao modo de execução não privilegiado (a menos que tenha sido invocado por um contexto de execução pertencente a uma partição de sistema);

INITIALIZE_SYSTEM_INFORMATION Inicializa a estrutura de informações gerenciais do sistema (**SYSTEM_INFORMATION**);

INITIALIZE_MODULE_INFORMATION Inicializa a estrutura de informações gerenciais do módulo e invoca o método **INITIALIZE_PARTITION_INFORMATION** para inicialização equivalente das partições;

INITIALIZE_PARTITION_INFORMATION Inicializa a estrutura de informações gerenciais de uma determinada partição e invoca o método **INITIALIZE_PROCESS_INFORMATION** para inicialização equivalente dos processos – este método trata também a reinicialização de partições, operação na qual algumas das informações gerenciais da partição são mantidas;

INITIALIZE_PROCESS_INFORMATION Inicializa a estrutura de informações gerenciais de um processo de uma determinada partição;

RETURNPOINT_MODULE_HEALTHMONITORINGCALLBACK Ponto de retorno do *HM callback* do módulo, que é alcançado quando esse método retorna – executa a ação de recuperação definida em configuração para o nível de módulo através do serviço interno **RUN_MODULE_ERROR_ACTION**, e então termina a execução do *HM callback* através do serviço interno **STOP_MODULE_HEALTHMONITORINGCALLBACK**;

RETURNPOINT_PARTITION_HEALTHMONITORING-CALLBACK Ponto de retorno dos *HM callbacks* das partições, que é alcançado quando um desses métodos retorna – executa a ação de recuperação definida em configuração para o nível de partição através do serviço interno **RUN_PARTITION_ERROR_ACTION**, e então termina a execução do *HM callback* através do serviço interno **STOP_PARTITION_HEALTHMONITORINGCALLBACK**;

RETURNPOINT_PARTITIONERRORHANDLER Ponto de retorno dos processos tratadores de erros das partições, que é alcançado caso um desses métodos retorne – termina a execução do processo tratador de erros através do serviço **STOP_SELF**;

RETURNPOINT_PARTITIONDEFAULTPROCESS Ponto de retorno dos processos padrões das partições, que é alcançado caso um desses métodos retorne – executa um laço infinito, já que processos padrões de partições não podem ser terminados;

RETURNPOINT_PROCESS Ponto de retorno de processos, que é alcançado caso um desses métodos retorne – termina a execução do processo através do serviço **STOP_SELF**.

C.6 MIGRAÇÃO

Descreve-se nesta seção os elementos de código que devem ser elaborados para a migração do SO desenvolvido neste trabalho para uma determinada plataforma de *hardware*. Juntamente com a apresentação desses elementos, cita-se ainda a forma como esses foram desenvolvidos neste trabalho para processadores *TI* AM335X.

C.6.1 Símbolos

Os símbolos apresentados a seguir devem ser definidos de acordo com as características e configurações da plataforma de *hardware* sobre a qual o SO será executado:

PORT_TICKWINDOW Indica o período da interrupção de escalonamento em nanossegundos, e é utilizado para atualização do relógio do módulo – para processadores *TI* AM335X, assume valor 1000000ns (1ms);

PORT_HEAP_ALIGNMENT Indica o alinhamento de memória que deve ser fundamentalmente respeitado para as regiões de alocação dinâmica de memória – para processadores *TI* AM335X, assume valor 4 por tratar-se de uma arquitetura de 32 *bits*.

C.6.2 Métodos

A implementação dos métodos apresentados a seguir é parte fundamental do processo de migração do SO desenvolvido neste trabalho. Esses métodos podem ser implementados tanto como procedimentos quanto como *macros*, e tanto em linguagem C quanto em *assembly*, conforme os requisitos específicos impostos pela plataforma de *hardware* utilizada.

PORT_SHUTDOWNMODULE Utilizado pelo serviço específico **SET_MODULE_MODE** para interromper a execução do módulo, e portanto de todas suas partições – para processadores *TI* AM335X, desabilita as interrupções e entra em um laço infinito;

PORT_RESTARTMODULE Utilizado pelo serviço específico **SET_MODULE_MODE** para reinicialização do módulo – para processadores *TI* AM335X, dispara a reinicialização do processador;

PORT_PREPARECALLADDRESS Prepara a leitura do endereço a partir do qual o método atual foi chamado, que representa o endereço dos erros gerados através do serviço **RAISE_APPLICATION_ERROR** – para processadores *TI* AM335X, empilha o registrador **LR**;

PORT_GETCALLADDRESS Recupera o endereço a partir do qual o método atual foi chamado (preparado através de uma chamada ao método **PORT_PREPARECALLADDRESS**) – para processadores *TI* AM335X, desempilha o endereço previamente empilhado pelo método **PORT_PREPARECALLADDRESS** no registrador de retorno da função;

PORT_INITIALIZETICK Inicializa a interrupção periódica de escalonamento, garantindo que a primeira interrupção ocorrerá após o primeiro período e não imediatamente após a execução do método – para processadores *TI* AM335X, invoca métodos do pacote *AM335X StarterWare* para configuração e inicialização dessa interrupção;

PORT_INITIALIZECONTEXT Inicializa uma estrutura de armazenamento de contexto de execução, e é invocado apenas uma vez para cada uma dessas estruturas;

PORT_PREPARECONTEXT Prepara uma estrutura de armazenamento de contexto de execução para sua efetiva utilização, e pode ser invocado diversas vezes (para processos que são parados e reiniciados, por exemplo);

PORT_ENTERCORE É invocado pelo método **ENTER_CORE** para efetivação da desabilitação de interrupções e da entrada em modo de execução privilegiado – pode ser invocado múltiplas vezes para chamadas aninhadas de serviços, devendo surtir esse efeito apenas na primeira chamada;

PORT_EXITCORE É invocado pelo método **EXIT_CORE** para efetivação da reabilitação de interrupções e retorno ao modo de execução não privilegiado, mas deve manter a execução em modo privilegiado caso o contexto de execução pertença a uma partição de sistema – é invocado apenas uma vez, mesmo para chamadas aninhadas de serviços;

PORT_YIELD Solicita a execução do algoritmo de escalonamento, possivelmente causando a interrupção do contexto de execução atual;

PORT_SWITCH Altera o contexto de execução atual imediatamente com base na estrutura global **CURRENT_CONTEXT**, sem salvar o contexto atual e sem executar o algoritmo de escalonamento – utilizado para parada de um contexto de execução a partir de serviços do SO;

PORT_STARTSCHEDULER Responsável pela primeira execução do algoritmo de escalonamento, podendo realizar configurações finais se forem necessárias, e caso não sejam necessárias pode simplesmente invocar **PORT_YIELD**;

PORT_REPORTSYSTEMERROR Utilizado para o reporte (geralmente através de algum tipo de saída padrão) de erros no nível de sistema – é executado com interrupções desabilitadas, e portanto causa *jitter*;

PORT_REPORTMODULEERROR Utilizado para o reporte de erros no nível de módulo – é executado com interrupções desabilitadas, e portanto causa *jitter*;

PORT_REPORTPARTITIONERROR Utilizado para o reporte de erros no nível de partição – é executado com interrupções desabilitadas, e portanto causa *jitter*;

PORT_REPORTPROCESSERROR Utilizado para o reporte de erros no nível de processo – é executado com interrupções desabilitadas, e portanto causa *jitter*;

PORT_REPORTAPPLICATIONMESSAGE Utilizado para o reporte de erros no nível de aplicação – é executado com interrupções desabilitadas, e portanto causa *jitter*.

Métodos denominados *hooks* são invocados pelo SO a partir dos principais serviços de gerenciamento dos elementos do sistema, e podem portanto ser utilizados para configuração de mecanismos da plataforma de *hardware* relacionados a eles. Apresenta-se a seguir esses métodos, que para processadores *TI* AM335X são utilizados para elaboração das tabelas de tradução da *MMU*:

PORT_HOOK_BEFORE_STARTUP_SYSTEM Executado pelo SO antes da inicialização do sistema (serviço **STARTUP_SYSTEM**) – para processadores *TI* AM335X, efetua a preparação da tabela de vetores de exceção (definida pela constante global **PORT_VECTORTABLE**), inicializa a saída padrão de depuração (porta *serial*) e prepara a biblioteca de interação com a *MMU*;

PORT_HOOK_AFTER_STARTUP_SYSTEM Executado pelo SO após a inicialização do sistema – para processadores *TI* AM335X, prepara a tabela de tradução global, que define as permissões para as regiões de memória da tabela de vetores de exceção e dos registradores de periféricos;

PORT_HOOK_BEFORE_STARTUP_MODULE Executado pelo SO antes da inicialização do módulo (serviço **STARTUP_MODULE**) – para processadores *TI* AM335X, não executa qualquer ação;

PORT_HOOK_AFTER_STARTUP_MODULE Executado pelo SO após a inicialização do módulo – para processadores *TI* AM335X, prepara as tabelas de tradução da partição padrão e do *HM callback* do módulo, e finalmente habilita a *MMU* e inicia a execução em modo não privilegiado;

PORT_HOOK_BEFORE_CREATE_PARTITION Executado pelo SO antes da criação de uma partição (serviço **CREATE_PARTITION**) – para processadores *TI* AM335X, não executa qualquer ação;

PORT_HOOK_AFTER_CREATE_PARTITION Executado pelo SO após a criação de uma partição – para processadores *TI*

AM335X, prepara as tabelas de tradução do processo padrão e do *HM callback* da partição criada;

PORT_HOOK_BEFORE_CREATE_PROCESS Executado pelo SO antes da criação de um processo (serviço **CREATE_PROCESS**) – para processadores *TI* AM335X, não executa qualquer ação;

PORT_HOOK_AFTER_CREATE_PROCESS Executado pelo SO após a criação de um processo – para processadores *TI* AM335X, prepara a tabela de tradução do processo criado;

PORT_HOOK_BEFORE_CREATE_ERROR_HANDLER Executado pelo SO antes da criação de um processo tratador de erros (serviço **CREATE_ERROR_HANDLER**) – para processadores *TI* AM335X, não executa qualquer ação;

PORT_HOOK_AFTER_CREATE_ERROR_HANDLER Executado pelo SO após a criação de um processo tratador de erros – para processadores *TI* AM335X, prepara a tabela de tradução do processo tratador de erros criado.

C.7 MIGRAÇÃO PARA *TI* AM335X

Descreve-se nesta seção os elementos do SO desenvolvido neste trabalho que foram definidos exclusivamente para sua execução na plataforma de *hardware* empregada, que conforme citado anteriormente é equipada com processador *TI* AM335X.

C.7.1 Constantes

Apresenta-se a seguir as constantes definidas para controle de aspectos específicos da execução do SO em processadores *TI* AM335X:

Geração de *logs* Devido ao fato de que a geração de *logs* na saída padrão (porta *serial*) causa *jitter* no escalonamento de partições e processos, são definidas constantes cujo nome é prefixado por **PORT_GENERATECONSOLEOUTPUT_** que indicam se esses *logs* devem ou não ser gerados quando da ocorrência de erros e da geração de reportes;

Tamanhos de segmentos de dados São definidas constantes que indicam os tamanhos mínimos e máximos das mensagens e das filas empregadas pelos elementos de comunicação interpartição e in-

trapartição, impondo assim limites globais para o tamanho dos segmentos de dados alocados por esses elementos;

Tabela de vetores de exceção As constantes **PORT_VECTORTABLE_ADDRESS** e **PORT_VECTORTABLE_SIZE** indicam respectivamente o endereço de memória a partir do qual deve ser alocada a tabela de vetores de exceção e o tamanho dessa tabela **em palavras**.

C.7.2 Métodos

Apresenta-se a seguir os métodos definidos especificamente para execução do SO desenvolvido neste trabalho em processadores *TI* AM335X:

PORT_RESETHANDLER Procedimento de inicialização do processador – subdivide a região de memória utilizada como pilha do sistema para os múltiplos modos de execução, limpa a região de dados não inicializados do sistema, limpa os registradores de exceções de acesso a dados e instruções, habilita o mecanismo de *branch prediction*, habilita o coprocessador *VFP* e, finalmente, desvia a execução para a partição padrão do módulo;

PORT_DISABLEINTERRUPTS Desabilita interrupções;

PORT_ENABLEINTERRUPTS Habilita interrupções;

PORT_TICK Trata a interrupção periódica de escalonamento;

PORT_IRQHANDLER Trata interrupções do tipo *IRQ*, no caso apenas a interrupção periódica de escalonamento (invocando **PORT_TICK**);

PORT_FIQHANDLER Trata interrupções do tipo *FIQ*, não utilizadas neste trabalho;

PORT_EXITPRIVILEGEDMODE Utilizado para sair do modo de execução privilegiado após a inicialização do módulo;

PORT_ABORTERRORHANDLER Tratador de alto nível (em linguagem C) para exceções geradas por erros de acesso a dados e instruções – reporta o erro, diferencia violações de memória e estouros de pilha (para erros de acesso a dados) e dispara o tratamento do erro através de uma chamada ao serviço interno **RAISE_ERROR**;

PORT_ABORTHANDLER Tratador de baixo nível (em linguagem *assembly*) para exceções geradas por erros de acesso a dados e

instruções – salva o contexto de execução atual, coleta as informações necessárias para o tratamento do erro, invoca o método **PORT_ABORTERRORHANDLER**, executa o algoritmo de escalonamento e restaura o contexto de execução;

PORT_UNDEFINEDINSTRUCTIONERRORHANDLER Tratador de alto nível (em linguagem C) para exceções geradas por erros de decodificação de instruções – reporta o erro e dispara seu tratamento através de uma chamada a **RAISE_ERROR**;

PORT_UNDEFINEDINSTRUCTIONHANDLER Tratador de baixo nível (em linguagem *assembly*) para exceções geradas por erros de decodificação de instruções – salva o contexto de execução atual, coleta as informações necessárias para o tratamento do erro, invoca o método **PORT_UNDEFINEDINSTRUCTIONERRORHANDLER**, executa o algoritmo de escalonamento e restaura o contexto de execução;

PORT_SVCHANDLER Tratador de chamadas de supervisor (interrupções de *software*), que são utilizadas para a execução das operações **PORT_ENTERCORE**, **PORT_EXITCORE**, **PORT_YIELD**, **PORT_SWITCH**, **PORT_STARTSCHEDULER**, **PORT_DISABLEINTERRUPTS**, **PORT_ENABLEINTERRUPTS** e **PORT_EXITPRIVILEGEDMODE**.

C.7.3 Partição de sistema para entrada/saída

Conforme citado no Capítulo 4, deve ser considerada parte do processo de migração do SO desenvolvido neste trabalho para uma nova plataforma de *hardware* a implementação da partição de sistema que realiza a transmissão de informações para elementos de comunicação interpartição.

C.8 MODELOS DE APLICAÇÃO

Descreve-se a seguir de forma aprofundada os modelos de aplicação gerados pelas ferramentas de configuração apresentadas no Capítulo 5. Destaca-se ainda as porções desses modelos que variam em função da plataforma de *hardware* na qual eles serão executados, apresentando a forma como são elaboradas para processadores *TI*

AM335X. Descreve-se nas próximas seções a função dos arquivos gerados nesse processo:

C.8.1 configuration.c/configuration.h

Contêm a definição das estruturas globais de configuração do sistema e do módulo (**SYSTEM_CONFIGURATION** do tipo **SYSTEM_CONFIGURATION_TYPE** e **MODULE_CONFIGURATION** do tipo **MODULE_CONFIGURATION_TYPE**, respectivamente), que armazenam todas as informações de configuração necessárias para a execução do sistema. O conteúdo destes arquivos varia muito em função da plataforma de *hardware* à qual o modelo destina-se, já que armazenam diversas informações específicas dos elementos do sistema (como os endereços e tamanhos das regiões de memória a eles alocadas, por exemplo).

C.8.2 module.c/module.h

Contêm a definição de estruturas **PARTITIONX_PARTITION_ATTRIBUTE** do tipo **PARTITION_ATTRIBUTE_TYPE**, que armazenam os atributos das partições, e variáveis **PARTITIONX_PARTITION_ID** do tipo **PARTITION_ID_TYPE**, utilizadas para armazenamento dos identificadores atribuídos pelo SO para cada uma das partições do módulo. Contêm ainda o método **MODULE_DEFAULTPARTITION**, que implementa a partição padrão e a partição ociosa do módulo, e podem conter também o método **MODULE_HMCALLBACK**, que implementa o *HM callback* do módulo.

C.8.3 partitionX.c/partitionX.h

Contêm a implementação da partição de nome **PartitionX**. Possuem estruturas chamadas **PROCESSY_PROCESS_ATTRIBUTE** do tipo **PROCESS_ATTRIBUTE_TYPE**, que armazenam os atributos dos processos, e variáveis chamadas **PROCESSY_PROCESS_ID** do tipo **PROCESS_ID_TYPE**, utilizadas para armazenamento dos identificadores atribuídos pelo SO para cada um dos processos da partição. Contêm ainda o método **PARTITIONX_DEFAULTPROCESS**, que implementa o

processo padrão e o processo ocioso da partição, um conjunto de métodos denominados **PROCESSY**, que implementam cada um dos processos da partição, e podem conter os métodos **PARTITIONX_ERRORHANDLER** e **PARTITIONX_HM-CALLBACK**, que implementam respectivamente o processo tratador de erros e o *HM callback* da partição **PartitionX**.

C.8.4 AM335X.cmd

Contém a definição do mapeamento de memória do sistema no formato de um arquivo de comandos para o *linker* empregado neste trabalho, declarando os endereços e tamanhos efetivos de todas as regiões de memória alocadas aos elementos do sistema e indicando a essa ferramenta em quais dessas regiões o código e os dados desses elementos devem ser inseridos. O conteúdo deste arquivo varia (ou ele pode até mesmo não ser necessário) em função da plataforma de *hardware* à qual o modelo se destina, e portanto todas as informações nele contidas podem ser consideradas específicas para processadores *TI* AM335X. Apresenta-se a seguir as regiões de memória descritas neste arquivo cuja localização é fixa:

SRAM_VECTORTABLE Localização da tabela de vetores de exceção, alocada na *SRAM* interna do núcleo do processador a partir do endereço **PORT_VECTORTABLE_ADDRESS** = 0x40300000, e com tamanho reservado de 4KB para viabilizar o controle de acesso pela *MMU*, porém dos quais são ocupadas apenas **PORT_VECTORTABLE_SIZE** = 16 palavras = 64 *bytes*;

PERIPHERALS Região na qual estão mapeados os registradores de periféricos do processador, a partir do endereço 0x44000000 e com tamanho 0x3C000000 *bytes*;

DDR0_SYSTEM_BOOT Região onde localiza-se o código de inicialização do processador (método **PORT_RESETHANDLER**), mapeada a partir do endereço 0x80000000 e com tamanho definido em configuração, a partir da qual a execução do *software* é iniciada.

Apresenta-se a seguir as demais regiões de memória declaradas neste arquivo, cuja função foi apresentada no Capítulo 4 e cuja alocação de endereços e tamanhos efetivos é realizada pelas ferramentas de configuração apresentadas no Capítulo 5:

DDR0_SYSTEM_STACK Pilha do sistema;

DDR0_SYSTEM_CODE Código do sistema;

DDR0_SYSTEM_DATA Dados do sistema;

DDR0_SYSTEM_HEAP Alocação dinâmica do sistema;

DDR0_SYSTEM_FLTRANSLATIONTABLE Tabelas de tradução de primeiro nível;

DDR0_SYSTEM_SLTRANSLATIONTABLE Tabelas de tradução de segundo nível;

DDR0_MODULE_CODE Código do módulo;

DDR0_MODULE_DATA Dados do módulo – armazena também as estruturas de configuração definidas pelos arquivos **configuration.c** e **configuration.h**;

DDR0_MODULE_HMCALLBACK_STACK Pilha do *HM callback* do módulo;

DDR0_PARTITIONX_HEAP Alocação dinâmica da partição **PartitionX**;

DDR0_PARTITIONX_PARTITION_CODE Código da partição **PartitionX**;

DDR0_PARTITIONX_PARTITION_DATA Dados da partição **PartitionX**;

DDR0_PARTITIONX_PARTITION_DATA_IMAGE Imagem de dados da partição **PartitionX**;

DDR0_PARTITIONX_PARTITION_DEFAULTPROCESS_STACK Pilha do processo padrão da partição **PartitionX**;

DDR0_PARTITIONX_PARTITION_ERRORHANDLER_STACK Pilha do processo tratador de erros da partição **PartitionX**;

DDR0_PARTITIONX_PARTITION_HMCALLBACK_STACK Pilha do *HM callback* da partição **PartitionX**;

DDR0_PARTITIONX_PARTITION_PROCESSY_PROCESS_STACK Pilha do processo **ProcessY** da partição **PartitionX**;

Regiões separadoras de regiões de pilha Conforme descrito em seção anterior deste capítulo.

C.9 EXECUÇÃO

Para execução de uma aplicação no SO desenvolvido neste trabalho sem a conexão da plataforma de *hardware* a um microcomputador, ou sem a utilização da interface de depuração para isso, é necessário que essa seja compilada e convertida para um formato a partir do qual possa ser carregada pelos mecanismos de inicialização oferecidos pela plata-

forma, conforme descrito no Capítulo 3. O compilador utilizado neste trabalho gera como saída um arquivo executável no formato *Executable and Linkable Format* (*ELF*) que precisa ser transformado, através de ferramentas auxiliares, em um arquivo binário que contém a imagem de memória a ser transferida para a plataforma. Apresenta-se a seguir as ferramentas empregadas nesse processo, conforme apresentado também em (TI, 2013c), sendo **ARINC653_AM335X** o nome do projeto na *IDE Code Composer Studio*:

armofd Fornecida juntamente com o compilador, é utilizada para geração de um arquivo no formato *XML* que contém informações sobre os elementos do arquivo *ELF*:

armofd -x –obj_display=none,header,sections,segments ARINC653_AM335X.out > ARINC653_AM335X.xml

mkhex4bin Fornecida juntamente com a *IDE*, é empregada na geração de um arquivo que contém o endereço efetivo e o tamanho total da imagem de memória a ser gerada, com base no arquivo *XML* gerado pela ferramenta **armofd**:

mkhex4bin ARINC653_AM335X.xml > ARINC653_AM335X.add

armhex Fornecida juntamente com o compilador, gera a imagem de memória a ser carregada na plataforma, com base no arquivo *ELF* e no arquivo gerado pela ferramenta **mkhex4bin**:

armhex -q -b -image -o ARINC653_AM335X.bin ARINC653_AM335X.add ARINC653_AM335X.out

tiimage Fornecida no pacote *AM335X StarterWare*, prepara a imagem de memória para ser carregada pelo *bootloader* da plataforma – adiciona um cabeçalho que contém o tamanho da imagem e o endereço a partir do qual a execução deve ser iniciada (que é fornecido na linha de comando):

tiimage 0x80000000 NONE ARINC653_AM335X.bin APP

O arquivo **APP** gerado pela ferramenta **tiimage** deve ser armazenado no diretório raiz de um cartão *SD*, juntamente com o arquivo **MLO** do *bootloader* específico da plataforma *BeagleBone* que também é fornecido no pacote *AM335X StarterWare*. Inserindo o cartão *SD* e acionando a plataforma, o *bootloader* efetua a cópia da imagem contida no arquivo **APP** para a memória da plataforma, e finalmente dispara sua execução a partir do endereço indicado no cabeçalho desse arquivo.

Caso esses passos sejam seguidos e a execução da aplicação não seja iniciada, deve-se verificar a saída padrão (porta *serial*) da plataforma *BeagleBone* para maiores informações. Pode ser necessária a formatação do cartão *SD*, nesse caso torna-se necessária a consulta à documentação do pacote *AM335X StarterWare* (TI, 2013c) para instruções detalhadas.

APÊNDICE D – Casos de teste

Este apêndice apresenta detalhadamente os casos de teste desenvolvidos neste trabalho para demonstração do funcionamento dos serviços e da aderência das características do SO às exigências da especificação ARINC 653. Para cada um desses casos de teste apresenta-se seu objetivo, o cenário no qual é executado, a descrição de seu funcionamento e o comportamento esperado quando de sua execução.

D.1 TESTE 001

Nome: SLOWPARTITIONSCHEDULING

Objetivo: Demonstrar a corretude temporal do escalonamento de partições em velocidade visualmente observável através de *LEDs*.

D.1.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	4s
Partição PARTITION1	
Período/duração	2s/1s
Escala	(0ms, 1s), (2s, 1s)
Partição PARTITION2	
Período/duração	2s/500ms
Escala	(1s, 500ms), (3s, 500ms)
Partição PARTITION3	
Período/duração	4s/250ms
Escala	(1.5s, 250ms)

D.1.2 Descrição

Os seguintes padrões são repetidamente atribuídos pelas tarefas do módulo aos *LEDs*:

- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.1.3 Comportamento esperado

Os *LEDs* devem apresentar, repetidamente, o seguinte comportamento:

- Apenas **LED1** ligado durante 1s (PARTITION1);
- Apenas **LED2** ligado durante 500ms (PARTITION2);
- Ambos *LEDs* ligados durante 250ms (PARTITION3);
- Ambos *LEDs* desligados durante 250ms (partição ociosa);
- Apenas **LED1** ligado durante 1s (PARTITION1);
- Apenas **LED2** ligado durante 500ms (PARTITION2);
- Ambos *LEDs* desligados durante 500ms (partição ociosa).

D.2 TESTE 002

Nome: FASTPARTITIONSCHEDULING
Objetivo: Demonstrar a corretude temporal do escalonamento de partições em velocidade adequada ao escalonamento de processos.

D.2.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.2.2 Descrição

Os seguintes padrões são repetidamente atribuídos pelas tarefas do módulo aos *LEDs*:

- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.2.3 Comportamento esperado

Os *LEDs* devem apresentar, repetidamente, o seguinte comportamento:

- Apenas **LED1** ligado durante 25ms (PARTITION1);
- Apenas **LED2** ligado durante 10ms (PARTITION2);

- Ambos *LEDs* ligados durante 10ms (PARTITION3);
- Ambos *LEDs* desligados durante 5ms (partição ociosa);
- Apenas **LED1** ligado durante 25ms (PARTITION1);
- Apenas **LED2** ligado durante 10ms (PARTITION2);
- Ambos *LEDs* desligados durante 15ms (partição ociosa).

D.3 TESTE 003

Nome: SLOWPROCESSSCHEDULING
Objetivo: Demonstrar a corretude temporal do escalonamento de processos de uma partição em velocidade visualmente observável através de *LEDs*.

D.3.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	4s
Partição PARTITION1	
Período/duração	2s/1s
Escala	(0ms, 1s), (2s, 1s)
Processo PROCESS1	Período/capacidade = 4s/4s Prioridade = 30
Processo PROCESS2	Período/capacidade = 8s/4s Prioridade = 20
Partição PARTITION2	
Período/duração	2s/500ms
Escala	(1s, 500ms), (3s, 500ms)
Partição PARTITION3	
Período/duração	4s/250ms
Escala	(1.5s, 250ms)

D.3.2 Descrição

Os processos ociosos das partições e a partição ociosa repetidamente desligam ambos os *LEDs*, enquanto os processos da partição PARTITION1, a cada liberação, atribuem os seguintes padrões aos *LEDs* por um curto período:

- PROCESS1: **LED1** ligado, **LED2** desligado;
- PROCESS2: **LED1** desligado, **LED2** ligado.

D.3.3 Comportamento esperado

A partir do início do segundo período da partição PARTITION1 (que inicia 2s após a inicialização do sistema), os *LEDs* devem apresentar repetidamente o seguinte comportamento:

- Apenas **LED1** ligado por um curto período (PROCESS1);

- Apenas **LED2** ligado por um curto período (PROCESS2);
- Ambos *LEDs* desligados até 4s a partir do início da sequência;
- Apenas **LED1** ligado por um curto período (PROCESS1);
- Ambos *LEDs* desligados até 8s a partir do início da sequência.

D.4 TESTE 004

Nome: FASTPROCESSSCHEDULING
Objetivo: Demonstrar a corretude temporal do escalonamento de processos em velocidade adequada à execução de aplicações.

D.4.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.4.2 Descrição

Os processos ociosos das partições e a partição ociosa repetidamente desligam ambos os *LEDs*, enquanto os processos da partição PARTITION1, a cada liberação, atribuem os seguintes padrões aos *LEDs* por um curto período:

- PROCESS1: **LED1** ligado, **LED2** desligado;
- PROCESS2: **LED1** desligado, **LED2** ligado.

D.4.3 Comportamento esperado

A partir do início do segundo período da partição PARTITION1 (que inicia 50ms após a inicialização do sistema), os *LEDs* devem apresentar repetidamente o seguinte comportamento:

- Apenas **LED1** ligado por um curto período (PROCESS1);

- Apenas **LED2** ligado por um curto período (PROCESS2);
- Ambos *LEDs* desligados até 250ms a partir do início da sequência;
- Apenas **LED2** ligado por um curto período (PROCESS2);
- Ambos *LEDs* desligados até 500ms a partir do início da sequência.

D.5 TESTE 005

Nome: TESTCOUNTERCALIBRATION
Objetivo: Simplificar o processo de calibração das constantes de temporização utilizadas nos casos de teste.

D.5.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	4s
Partição PARTITION1	
Período/duração	2s/1s
Escala	(0ms, 1s), (2s, 1s)
Processo PROCESS1	Período/capacidade = 4s/4s Prioridade = 30
Partição PARTITION2	
Período/duração	2s/500ms
Escala	(1s, 500ms), (3s, 500ms)
Partição PARTITION3	
Período/duração	4s/250ms
Escala	(1.5s, 250ms)

D.5.2 Descrição

Os processos ociosos das partições e a partição ociosa repetidamente desligam ambos os *LEDs*. A cada liberação o processo PROCESS1 atribui um determinado padrão aos *LEDs* durante um diferente período, de acordo com as constantes de temporização utilizadas nos casos de teste.

D.5.3 Comportamento esperado

A cada liberação do processo PROCESS1 os *LEDs* devem apresentar um dos seguintes padrões:

- Apenas **LED1** ligado por ≈1ms;
- Apenas **LED1** ligado por ≈5ms;
- Apenas **LED1** ligado por ≈10ms;
- Apenas **LED1** ligado por ≈25ms;
- Apenas **LED1** ligado por ≈50ms;

- Apenas **LED1** ligado por $\approx$100ms;
- Apenas **LED1** ligado por $\approx$250ms;
- Apenas **LED1** ligado por $\approx$750ms;
- Apenas **LED1** ligado por $\approx$1s.

D.6 TESTE 006

Nome: RETURNPOINT
Objetivo: Garantir que os pontos de retorno, que são funções alcançadas quando os métodos que implementam as tarefas do sistema terminam ou retornam, são corretamente executados.

D.6.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros são ignorados

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.6.2 Descrição

Os processos PROCESS1 e PROCESS2 piscam o **LED1**, disparam um erro de aplicação, desligam ambos os *LEDs* e então retornam. O processo tratador de erros da partição PARTITION1 pisca o **LED2**, lê todas as entradas de erro, dispara um erro de aplicação, desliga ambos os *LEDs* e então retorna. Os processos ociosos de partições, após a inicialização, disparam um erro de aplicação, desligam ambos os *LEDs*

e então retornam. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo tratador de erros da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo tratador de erros da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Processo tratador de erros da partição PARTITION3: **LED1** desligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.6.3 Comportamento esperado

O comportamento deste teste deve ser verificado através de depuração a nível de instruções de máquina. Todas as tarefas, ao retornar, devem alcançar o ponto de retorno correspondente e, assim, executar o tratamento adequado. Visualmente, os *LEDs* devem piscar rapidamente antes que todas as tarefas sejam concluídas, e então os *LEDs* devem ficar permanentemente desligados.

D.7 TESTE 007

Nome: SYSTEMPARTITION
Objetivo: Demonstrar a utilização e o correto funcionamento de partições de sistema que acessam dados do núcleo do SO.

D.7.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo
Módulo	
Major frame	100ms
HM callback	Habilitado
Tratamento de erros	Erros são ignorados
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.7.2 Descrição

A cada liberação o processo PROCESS1 escreve na saída padrão informações sobre as partições e os processos do módulo, e a cada liberação o processo PROCESS2 escreve nessa mesma saída o horário atual do módulo. Os padrões de *LEDs* repetidamente atribuídos pelas demais tarefas são os seguintes:

- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;

- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.7.3 Comportamento esperado

Os *LEDs* devem apresentar repetidamente o padrão de escalonamento do módulo. Na saída padrão deve ser exibido o horário atual do módulo na frequência de liberação do processo PROCESS2, e informações sobre as partições e processos a cada liberação do processo PROCESS1.

D.8 TESTE 101

Nome: SETMODULEMODE_IDLE
Objetivo: Demonstrar o correto funcionamento do serviço **SET_MODULE_MODE** quando solicitada transição para o modo de operação **IDLE**.

D.8.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.8.2 Descrição

Após um determinado número de liberações, o processo PROCESS2 da partição PARTITION1 atribui o modo de operação do módulo para **IDLE**, parando sua execução. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;

- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.8.3 Comportamento esperado

Os *LEDs* devem apresentar repetidamente o padrão de escalonamento até que o módulo seja parado, sendo então exibido permanentemente o padrão do processo PROCESS2 ou ambos os *LEDs* desligados, dependendo da forma como a parada do módulo é implementada.

D.9 TESTE 102

Nome: SETMODULEMODE_COLDSTART
Objetivo: Demonstrar o correto funcionamento do serviço **SET_MODULE_MODE** quando solicitada transição para o modo de operação **COLD_START**.

D.9.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.9.2 Descrição

Após um determinado número de liberações, o processo PROCESS2 da partição PARTITION1 atribui o modo de operação do módulo para **COLD_START**, reiniciando sua execução. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;

- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.9.3 Comportamento esperado

Os *LEDs* devem apresentar repetidamente o padrão de escalonamento, com pequenos intervalos durante os quais o módulo é reiniciado.

D.10 TESTE 103

Nome: GETPARTITIONID
Objetivo: Demonstrar o correto funcionamento do serviço **GET_PARTITION_ID**.

D.10.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.10.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*, invoca **GET_PARTITION_ID** para todas as partições do módulo, verifica o resultado e, caso uma falha seja detectada, entra em um laço infinito. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;

- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.10.3 Comportamento esperado

Os *LEDs* devem apresentar repetidamente o padrão de escalonamento (nenhum laço infinito deve ser alcançado).

D.11 TESTE 104

Nome: GETPARTITIONSTATUS
Objetivo: Demonstrar o correto funcionamento do serviço **GET_PARTITION_STATUS**.

D.11.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.11.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*, invoca **GET_PARTITION_STATUS**, verifica o resultado e, caso uma falha seja detectada, entra em um laço infinito. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;

- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.11.3 Comportamento esperado

Os *LEDs* devem apresentar repetidamente o padrão de escalonamento (nenhum laço infinito deve ser alcançado).

D.12 TESTE 105

Nome: SETPARTITIONMODE_IDLE_PROCESS
Objetivo: Demonstrar o correto funcionamento do serviço **SET_PARTITION_MODE** quando solicitada transição para o modo de operação **IDLE** a partir de um processo.

D.12.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.12.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*. Após algumas liberações o processo PROCESS2 atribui o modo de operação da partição PARTITION1 para **IDLE**, parando qualquer processamento nela. Mais tarde, os processos ociosos das partições PARTITION2 e PARTITION3 também atribuem o modo de suas partições para **IDLE**, nessa ordem e em momentos diferentes. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;

- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.12.3 Comportamento esperado

Os *LEDs* devem inicialmente apresentar repetidamente o padrão de escalonamento. Após algum tempo as partições PARTITION1, PARTITION2 e PARTITION3 são paradas, nessa ordem e em diferentes momentos. O padrão apresentado nos *LEDs* deve refletir esses eventos, e finalmente apenas a partição ociosa deve permanecer em execução e, portanto, ambos os *LEDs* devem ficar permanentemente desligados.

D.13 TESTE 106

Nome:
SETPARTITIONMODE_IDLE_PARTITIONERRORHANDLER

Objetivo: Demonstrar o correto funcionamento do serviço **SET_PARTITION_MODE** quando solicitada transição para o modo de operação **IDLE** a partir de um processo tratador de erros.

D.13.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros são ignorados

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.13.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*. Após algumas liberações o processo PROCESS1 entra em um laço infinito, perdendo assim seu *deadline* e fazendo com que o processo tratador de erros da partição PARTITION1 seja iniciado. O processo tratador de erros atribui o modo de operação da partição

para **IDLE**, parando qualquer processamento nela. Mais tarde, os processos ociosos das partições PARTITION2 e PARTITION3 também atribuem o modo de suas partições para **IDLE**, nessa ordem e em momentos diferentes. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.13.3 Comportamento esperado

Os *LEDs* devem, inicialmente, apresentar repetidamente o padrão de escalonamento. Após algum tempo o padrão do processo PROCESS1 deve ser exibido por um período relativamente longo (causando uma perda de *deadline*), e então as partições PARTITION1, PARTITION2 e PARTITION3 são paradas, nessa ordem e em diferentes momentos. O padrão apresentado nos *LEDs* deve refletir esses eventos, e finalmente apenas a partição ociosa deve permanecer em execução e, portanto, ambos os *LEDs* devem ficar permanentemente desligados.

D.14 TESTE 107

Nome: SETPARTITIONMODE_IDLE_PARTITIONHMCALLBACK

Objetivo: Demonstrar o correto funcionamento do serviço **SET_PARTITION_MODE** quando solicitada transição para o modo de operação **IDLE** a partir de um *HM callback* de partição.

D.14.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros são ignorados

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.14.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*. Após algumas liberações o processo PROCESS1 entra em um laço infinito, perdendo assim seu *deadline* e fazendo com que o processo tratador de erros da partição PARTITION1 seja iniciado. O processo tratador de erros dispara um erro de aplicação, fazendo com

que o *HM callback* da partição seja iniciado. O *HM callback* atribui o modo de operação da partição para **IDLE**, parando qualquer processamento nela. Mais tarde, os processos ociosos das partições PARTITION2 e PARTITION3 também atribuem o modo de suas partições para **IDLE**, nessa ordem e em momentos diferentes. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.14.3 Comportamento esperado

Os *LEDs* devem, inicialmente, apresentar repetidamente o padrão de escalonamento. Após algum tempo o padrão do processo PROCESS1 deve ser exibido por um período relativamente longo (causando uma perda de *deadline*), e então as partições PARTITION1, PARTITION2 e PARTITION3 são paradas, nessa ordem e em diferentes momentos. O padrão apresentado nos *LEDs* deve refletir esses eventos, e finalmente apenas a partição ociosa deve permanecer em execução e, portanto, ambos os *LEDs* devem ficar permanentemente desligados.

D.15 TESTE 108

Nome: SETPARTITIONMODE_COLDSTART_PROCESS
Objetivo: Demonstrar o correto funcionamento do serviço **SET_PARTITION_MODE** quando solicitada transição para o modo de operação **COLD_START** a partir de um processo.

D.15.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.15.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*. Após algumas liberações o processo PROCESS2 atribui o modo de operação da partição PARTITION1 para **COLD_START**, reiniciando-a. Mais tarde, os processos ociosos das partições PARTITION2 e PARTITION3 também atribuem o modo de suas partições para **COLD_START**, nessa ordem e em momentos diferentes. Os processos padrão de todas as partições verificam se as variáveis globais da partição têm seus valores iniciais corretos, alterando seus valores em caso positivo e entrando em um laço infinito em caso negativo, e verificando assim se os dados das partições são corretamente restaurados pelo SO quando de sua reinicialização. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;

- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.15.3 Comportamento esperado

Os *LEDs* devem inicialmente apresentar repetidamente o padrão de escalonamento. Após algum tempo as partições PARTITION1, PARTITION2 e PARTITION3 devem ser reiniciadas, nessa ordem e em diferentes momentos, e continuar exibindo o mesmo padrão novamente. Pequenas interrupções devem ser percebidas no padrão geral, durante a reinicialização das partições, e nenhuma das partições deve deixar de ser executada.

D.16 TESTE 109

Nome: SETPARTITIONMODE_COLDSTART_PARTITIONERRORHANDLER

Objetivo: Demonstrar o correto funcionamento do serviço **SET_PARTITION_MODE** quando solicitada transição para o modo de operação **COLD_START** a partir de um processo tratador de erros.

D.16.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros são ignorados

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.16.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*. Após algumas liberações o processo PROCESS1 entra em um laço infinito, perdendo assim seu *deadline* e fazendo com que o processo tratador de erros da partição PARTITION1 seja iniciado. O

processo tratador de erros atribui o modo de operação da partição para **COLD_START**, reiniciando-a. Mais tarde, os processos ociosos das partições PARTITION2 e PARTITION3 também atribuem o modo de suas partições para **COLD_START**, nessa ordem e em momentos diferentes. Os processos padrão de todas as partições verificam se as variáveis globais da partição têm seus valores iniciais corretos, alterando seus valores em caso positivo e entrando em um laço infinito em caso negativo, e verificando assim se os dados das partições são corretamente restaurados pelo SO quando de sua reinicialização. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.16.3 Comportamento esperado

Os *LEDs* devem inicialmente apresentar repetidamente o padrão de escalonamento. Após algum tempo as partições PARTITION1, PARTITION2 e PARTITION3 devem ser reiniciadas, nessa ordem e em diferentes momentos, e continuar exibindo o mesmo padrão novamente. Pequenas interrupções devem ser percebidas no padrão geral, durante a reinicialização das partições, e nenhuma das partições deve deixar de ser executada. Antes da reinicialização da partição PARTITION1, deve ser possível perceber que o padrão do processo PROCESS1 é exibido por um período relativamente longo, causando a perda de *deadline* que leva à reinicialização da partição.

D.17 TESTE 110

Nome: SETPARTITION-MODE_COLDSTART_PARTITIONHMCALLBACK

Objetivo: Demonstrar o correto funcionamento do serviço **SET_PARTITION_MODE** quando solicitada transição para o modo de operação **COLD_START** a partir de um *HM callback* de partição.

D.17.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros são ignorados

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.17.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*. Após algumas liberações o processo PROCESS1 entra em um laço infinito, perdendo assim seu *deadline* e fazendo com que o processo tratador de erros da partição PARTITION1 seja iniciado. O

processo tratador de erros dispara um erro de aplicação, fazendo com que o *HM callback* da partição seja iniciado. O *HM callback* atribui o modo de operação da partição para **COLD_START**, reiniciando-a. Mais tarde, os processos ociosos das partições PARTITION2 e PARTITION3 também atribuem o modo de suas partições para **COLD_START**, nessa ordem e em momentos diferentes. Os processos padrão de todas as partições verificam se as variáveis globais da partição têm seus valores iniciais corretos, alterando seus valores em caso positivo e entrando em um laço infinito em caso negativo, e verificando assim se os dados das partições são corretamente restaurados pelo SO quando de sua reinicialização. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.17.3 Comportamento esperado

Os *LEDs* devem inicialmente apresentar repetidamente o padrão de escalonamento. Após algum tempo as partições PARTITION1, PARTITION2 e PARTITION3 devem ser reiniciadas, nessa ordem e em diferentes momentos, e continuar exibindo o mesmo padrão novamente. Pequenas interrupções devem ser percebidas no padrão geral, durante a reinicialização das partições, e nenhuma das partições deve deixar de ser executada. Antes da reinicialização da partição PARTITION1, deve ser possível perceber que o padrão do processo PROCESS1 é exibido por um período relativamente longo, causando a perda de *deadline* que leva à reinicialização da partição.

D.18 TESTE 111

Nome: GETPROCESSID
Objetivo: Demonstrar o correto funcionamento do serviço **GET_PROCESS_ID**.

D.18.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.18.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*, invoca **GET_PROCESS_ID** para os processos PROCESS1 e PROCESS2, verifica o resultado e, caso uma falha seja detectada, entra em um laço infinito. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;

- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.18.3 Comportamento esperado

Os *LEDs* devem apresentar repetidamente o padrão de escalonamento (nenhum laço infinito deve ser alcançado).

D.19 TESTE 112

Nome: GETPROCESSSTATUS
Objetivo: Demonstrar o correto funcionamento do serviço **GET_PROCESS_STATUS**.

D.19.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.19.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*, invoca **GET_PROCESS_STATUS** para dois processos, verifica o resultado e, caso uma falha seja detectada, entra em um laço infinito. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;

- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.19.3 Comportamento esperado

Os *LEDs* devem apresentar repetidamente o padrão de escalonamento (nenhum laço infinito deve ser alcançado).

D.20 TESTE 113

Nome: SETPRIORITY
Objetivo: Demonstrar o correto funcionamento do serviço **SET_PRIORITY**.

D.20.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.20.2 Descrição

A cada liberação os processos PROCESS1 e PROCESS2 atribuem um diferente padrão aos *LEDs* por um curto período. Após algumas liberações, o processo PROCESS1 atribui a prioridade do processo PROCESS2 para um valor maior que sua prioridade, e após mais algumas liberações restaura a prioridade original desse processo. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.20.3 Comportamento esperado

Durante a execução dos processos com as prioridades originais, o **LED1** deve piscar precisamente a cada 500ms e o **LED2** deve piscar uma vez logo após o **LED1** e outra 250ms após o **LED1** ser ligado. Após a alteração da prioridade do processo PROCESS2 o **LED2** deve piscar precisamente a cada 250ms, enquanto o **LED1** deve piscar uma vez sim e uma vez não logo após o **LED2**. Em outras palavras, a ordem na qual os *LEDs* piscam quando os processos PROCESS1 e PROCESS2 estão em fase deve ser invertida quando a prioridade do processo PROCESS2 é alterada.

D.21 TESTE 114

Nome: SUSPEND_RESUME
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **SUSPEND** e **RESUME**.

D.21.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = -/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.21.2 Descrição

A cada liberação o processo PROCESS1 liga o **LED1** durante um curto período, e alternadamente suspende ou continua o processo PROCESS2, que repetidamente liga o **LED2**. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;

- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.21.3 Comportamento esperado

O **LED2** deve permanecer ligado durante todo o tempo alocado à partição PARTITION1 até que o processo PROCESS1 seja liberado, suspendendo o processo PROCESS2 e permanecendo, portanto, o **LED2** desligado até que PROCESS1 seja liberado novamente, continuando PROCESS2 e reiniciando a sequência. Essa sequência deve ser repetida indefinidamente.

NOTA: Quando o módulo é ativado o processo PROCESS2 deve iniciar sua execução antes do processo PROCESS1, já que processos aperiódicos são liberados imediatamente que iniciados e processos periódicos são liberados apenas no início do próximo período da partição.

D.22 TESTE 115

Nome: SUSPENDSELF_RESUME
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **SUSPEND_SELF** (com suspensão por tempo indeterminado) e **RESUME**.

D.22.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.22.2 Descrição

O processo PROCESS1 inicia e imediatamente solicita sua própria suspensão por tempo indeterminado. A cada liberação o processo PROCESS2 pisca o **LED2**, e após algumas liberações continua o processo PROCESS1. O processo PROCESS1 então interrompe o processo PROCESS2, pisca o **LED1** e suspende novamente por tempo indeterminado, reiniciando a sequência. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.22.3 Comportamento esperado

O **LED2** deve piscar precisamente a cada 250ms e, a cada $\boldsymbol{n}$ liberações do processo PROCESS2, o **LED1** deve piscar uma vez e permanecer desligado o restante do tempo. Esse comportamento deve ser repetido indefinidamente.

D.23 TESTE 116

Nome: SUSPENDSELF_RESUME_TIMEOUT
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **SUSPEND_SELF** (com suspensão por tempo limitado) e **RESUME**.

D.23.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.23.2 Descrição

O processo PROCESS1 inicia e imediatamente solicita sua própria suspensão por no máximo 500ms, enquanto a cada liberação o processo PROCESS2 pisca o **LED2**. O processo ocioso da partição PARTITION1 repetidamente aguarda um longo período e então libera o processo PROCESS1, que pisca o **LED1** e suspende novamente por no máximo 500ms, reiniciando a sequência. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.23.3 Comportamento esperado

O **LED2** deve piscar aproximadamente a cada 250ms, podendo variar porque o processo PROCESS1 pode interromper o processo PROCESS2, e a cada 500ms ou menos o **LED1** deve piscar. Quando o **LED1** pisca precisamente 500ms após a última vez em que foi ligado o processo PROCESS1 foi continuado pela expiração do tempo de suspensão, e quando pisca em um tempo menor que esse o processo PROCESS1 foi continuado pelo processo padrão da partição PARTITION1. Esse comportamento deve ser repetido indefinidamente.

D.24 TESTE 117

Nome: TIMEDWAIT
Objetivo: Demonstrar o correto funcionamento do serviço **TIMED_WAIT**.

D.24.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.24.2 Descrição

A cada liberação o processo PROCESS1 aguarda por no mínimo 150ms e pisca o **LED1**. A cada liberação o processo PROCESS2 pisca o **LED2**, aguarda por no mínimo 50ms e pisca o **LED2** novamente. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;

•Partição ociosa: **LED1** desligado, **LED2** desligado.

D.24.3 Comportamento esperado

O **LED1** deve piscar a cada 500ms, e o **LED2** deve piscar duas vezes a cada 250ms com um intervalo de 50ms. Já que o **LED1** é ligado 150ms mais tarde em relação ao momento de liberação do processo PROCESS1, durante esse atraso o processo PROCESS2 poderá ser executado. Portanto, o **LED2** deve piscar antes que o **LED1** em relação ao momento de liberação de ambos os processos quando esses encontram-se em fase, ou seja, na ordem oposta àquela esperada de acordo com a atribuição de prioridades.

D.25 TESTE 118

Nome: TIMEDWAIT_SUSPEND_RESUME
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **SUSPEND** e **RESUME** quando invocados para um processo que encontra-se bloqueado em uma chamada ao serviço **TIMED_WAIT**. Em outras palavras, demonstrar que chamadas aos serviços **SUSPEND** e **RESUME** durante uma espera por tempo não causam a liberação prematura do processo.

D.25.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.25.2 Descrição

O processo PROCESS1 repetidamente pisca o **LED1**, aguarda 2s, pisca o **LED1** novamente e solicita sua própria suspensão por 500ms. A cada liberação o processo PROCESS2 pisca o **LED2** e então suspende e continua (imediatamente) o processo PROCESS1. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.25.3 Comportamento esperado

O **LED2** deve piscar aproximadamente a cada 250ms, podendo variar porque o processo PROCESS1 pode interromper o processo PROCESS2. O **LED1** deve piscar uma vez e, no mínimo 2s mais tarde, começar a piscar duas vezes a cada aproximadamente 2s: uma vez antes de suspender e uma depois que o processo PROCESS2 liberar o processo PROCESS1.

D.26 TESTE 119

Nome: TIMEDWAIT_COOPERATIVESCHEDULING
Objetivo: Demonstrar o funcionamento do serviço **TIMED_WAIT** para a realização de escalonamento cooperativo, que baseia-se na cessão voluntária de recursos de processamento (ARPACI-DUSSEAU, 2012) e que é possível quando dois ou mais processos possuem a mesma prioridade.

D.26.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/- Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 30
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.26.2 Descrição

O processo PROCESS1 liga o **LED1** por um longo período e então cede o processamento ao processo PROCESS2, que também liga o **LED2** por um longo período e então cede o processamento ao processo PROCESS1. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.26.3 Comportamento esperado

A seguinte sequência de padrões de *LEDs* deve ser exibida repetidamente:

- **LED1** piscando de acordo com o padrão de escalonamento por algum tempo;
- **LED2** piscando de acordo com o padrão de escalonamento por algum tempo.

D.27 TESTE 120

Nome: REPLENISH
Objetivo: Demonstrar o funcionamento do serviço **REPLENISH**.

D.27.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros são ignorados
Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 1s/500ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/250ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados

D.27.2 Descrição

A cada liberação o processo PROCESS1 liga o **LED1**, aguarda por 400ms, desliga o **LED1**, solicita postergação de seu *deadline* por 400ms, liga o **LED1**, aguarda por mais 300ms, desliga o **LED1** novamente e então aguarda por sua próxima liberação. A cada liberação, o processo PROCESS2 liga o **LED2**, aguarda por 150ms, desliga o **LED2**, solicita postergação de seu *deadline* por 300ms, liga o **LED2**, aguarda por mais 200ms, desliga o **LED2** novamente e então aguarda

por sua próxima liberação.

D.27.3 Comportamento esperado

O **LED1** deve permanecer ligado por aproximadamente 700ms a cada 1000ms, e o **LED2** por aproximadamente 350ms a cada 500ms. Ou seja, ambos os processos executam por um período maior que suas capacidades de tempo nominais, porém nenhum *deadline* deve ser perdido pois ambos os processos solicitam postergação. Se um *deadline* for perdido, o processo tratador de erros da partição PARTITION1 será alcançado e ambos os *LEDs* permanecerão desligados indefinidamente.

D.28 TESTE 121

Nome: STOPSELF_START
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **STOP_SELF** e **START**.

D.28.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.28.2 Descrição

A cada liberação o processo PROCESS2 pisca o **LED2**, e após $\boldsymbol{n}$ liberações invoca o serviço **STOP_SELF**, terminando sua própria execução. A cada liberação o processo PROCESS1 pisca o **LED1**, e após $\boldsymbol{m}$ liberações invoca o serviço **START** para o processo PROCESS2, retomando sua execução. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.28.3 Comportamento esperado

O **LED1** deve piscar a cada 500ms, e o **LED2** deve piscar $\boldsymbol{n}$ vezes a cada 250ms e então permanecer desligado. Após $\boldsymbol{m}$ liberações do processo PROCESS1, o **LED2** deve voltar a piscar, reiniciando a sequência.

D.29 TESTE 122

Nome: STOPSELF_DELAYEDSTART
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **STOP_SELF** e **DELAYED_START**.

D.29.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.29.2 Descrição

A cada liberação o processo PROCESS2 pisca o **LED2**, e após n liberações invoca o serviço **STOP_SELF**, terminando sua própria execução. A cada liberação o processo PROCESS1 pisca o **LED1**, e após m liberações invoca o serviço **DELAYED_START** para o processo PROCESS2, reiniciando sua execução com um atraso de 100ms. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.29.3 Comportamento esperado

O **LED1** deve piscar a cada 500ms, e o **LED2** deve piscar $\boldsymbol{n}$ vezes a cada 250ms e então permanecer desligado. Após $\boldsymbol{m}$ liberações do processo PROCESS1, o **LED2** deve voltar a piscar, reiniciando a sequência. Visualmente o atraso de início do processo PROCESS2 é dificilmente percebido, portanto deve ser utilizado um osciloscópio a fim de verificar que a cada vez que o processo PROCESS2 é reiniciado ele é liberado com um atraso de 100ms em relação ao desligamento do **LED1**.

D.30 TESTE 123

Nome: STOP_START
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **STOP** e **START**.

D.30.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.30.2 Descrição

A cada liberação o processo PROCESS2 pisca o **LED2**. A cada liberação o processo PROCESS1 pisca o **LED1** e, a cada n liberações, invoca alternadamente os serviços **STOP** ou **START** para o processo PROCESS2. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.30.3 Comportamento esperado

O **LED1** deve piscar a cada 500ms, e o **LED2** deve piscar algumas vezes durante as primeiras $\boldsymbol{n}$ liberações do processo PROCESS1 e permanecer desligado durante as próximas $\boldsymbol{n}$, voltando então a piscar e reiniciando a sequência, que deve ser repetida indefinidamente.

D.31 TESTE 124

Nome: STOP_START_PERIODSTART
Objetivo: Demonstrar a corretude do momento da primeira liberação de processos periódicos, levando em conta as janelas de tempo de início de período da partição.

D.31.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros são ignorados

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/36ms
Escala	(0ms, 25ms), (35ms, 10ms)*, (45ms, 1ms)*, (50ms, 25ms), (85ms, 10ms)*, (95ms, 1ms)*
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/10ms Prioridade = 30

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	50ms/4ms
Escala	(46ms, 4ms), (96ms, 4ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.31.2 Descrição

A cada liberação o processo PROCESS1 liga o **LED1** e entra em um laço infinito, perdendo seu *deadline* e fazendo com que o processo tratador de erros da partição PARTITION1 seja iniciado. O processo tratador de erros liga o **LED2** e reinicia o processo PROCESS1. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo tratador de erros da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.31.3 Comportamento esperado

Um osciloscópio deve ser utilizado para verificar que o momentos nos quais o **LED1** é ligado estão corretamente alinhados em relação à escala da partição PARTITION1, levando em conta as janelas de tempo de início de período. Visualmente nenhuma observação é conclusiva.

D.32 TESTE 125

Nome: STOP_DELAYEDSTART
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **STOP** e **DELAYED_START**.

D.32.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.32.2 Descrição

A cada liberação o processo PROCESS2 pisca o **LED2**. A cada liberação o processo PROCESS1 pisca o **LED1** e, a cada ***n*** liberações, invoca alternadamente os serviços **STOP** ou **DELAYED_START** para o processo PROCESS2 com atraso de 100ms. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.32.3 Comportamento esperado

O **LED1** deve piscar a cada 500ms, e o **LED2** deve piscar algumas vezes durante as primeiras $\boldsymbol{n}$ liberações do processo PROCESS1 e permanecer desligado durante as próximas $\boldsymbol{n}$, voltando então a piscar e reiniciando a sequência, que deve ser repetida indefinidamente. Visualmente o atraso de início do processo PROCESS2 é dificilmente percebido, portanto deve ser utilizado um osciloscópio a fim de verificá-lo.

D.33 TESTE 126

Nome: DELAYEDSTART
Objetivo: Demonstrar o correto funcionamento do serviço **DELAYED_START**.

D.33.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	1s
Partição PARTITION1	
Período/duração	500ms/250ms
Escala	(0ms, 250ms), (500ms, 250ms)
Processo PROCESS1	Período/capacidade = 5s/1s Prioridade = 30
Processo PROCESS2	Período/capacidade = 10s/1s Prioridade = 20
Partição PARTITION2	
Período/duração	500ms/125ms
Escala	(250ms, 125ms), (750ms, 125ms)
Partição PARTITION3	
Período/duração	1s/75ms
Escala	(375ms, 75ms)

D.33.2 Descrição

O processo PROCESS1 é iniciado com atraso de 2s, e o processo PROCESS2 com atraso de 5s, evitando que esses sejam liberados em fase. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.33.3 Comportamento esperado

A seguinte sequência de padrões deve ser exibida repetidamente, iniciando a partir do início do segundo período da partição PARTITION1 (que inicia 500ms após a inicialização do módulo):

- Ambos *LEDs* desligados por 2s (atraso de início de PROCESS1);
- Apenas **LED1** ligado por um curto período (PROCESS1);
- Ambos *LEDs* desligados até 5s a partir do início da sequência (atraso de início de PROCESS2);
- Apenas **LED2** ligado por um curto período (PROCESS2);
- Ambos *LEDs* desligados até 7s a partir do início da sequência;
- Apenas **LED1** ligado por um curto período (PROCESS1);
- Ambos *LEDs* desligados até 10s a partir do início da sequência.

D.34 TESTE 127

Nome: LOCKPREEMPTION_UNLOCKPREEMPTION
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **LOCK_PREEMPTION** e **UNLOCK_PREEMPTION**.

D.34.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 250ms/500ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/500ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.34.2 Descrição

O processo PROCESS2, que tem a menor prioridade, executa um laço de duração maior que o período do processo PROCESS1, mas não suficiente para causar uma perda de *deadline*. Durante a execução desse laço, o processo PROCESS2 bloqueia a interrupção por outros processos, e portanto o processo PROCESS1 – mesmo tendo maior prioridade – não deve interrompê-lo. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.34.3 Comportamento esperado

A seguinte sequência de padrões deve ser exibida repetidamente:

- **LED1** piscando (de acordo com a escala de partições) por um curto período;
- Ambos *LEDs* desligados por algum tempo;
- **LED2** piscando (de acordo com a escala de partições) por um intervalo mais longo que o período do processo PROCESS1.

D.35 TESTE 128

Nome: LOCKPREEMPTION_STOPSELF_PROCESS
Objetivo: Demonstrar o correto funcionamento do serviço **STOP_SELF** se utilizado por um processo enquanto sua interrupção encontra-se bloqueada pelo serviço **LOCK_PREEMPTION**.

D.35.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.35.2 Descrição

A cada liberação o processo PROCESS2 pisca o **LED2** com a interrupção por outros processos bloqueada, e após n liberações termina sua própria execução (invoca **STOP_SELF**) enquanto essa interrupção ainda encontra-se bloqueada. A cada liberação o processo PROCESS1 pisca o **LED1**, e após m liberações inicia novamente o processo PROCESS2, reiniciando a sequência. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.35.3 Comportamento esperado

O **LED1** deve piscar a cada 500ms, e o **LED2** deve piscar $\boldsymbol{n}$ vezes a cada 250ms e então permanecer desligado. Após $\boldsymbol{m}$ liberações do processo PROCESS1, o **LED2** deve voltar a piscar, reiniciando a sequência.

D.36 TESTE 129

Nome:
LOCKPREEMPTION_STOPSELF_PARTITIONERRORHANDLER

Objetivo: Demonstrar o correto funcionamento do serviço **STOP_SELF** se utilizado por um processo tratador de erros após uma chamada ao serviço **LOCK_PREEMPTION**, situação na qual o serviço **STOP_SELF** deve continuar a execução do processo interrompido pelo processo tratador de erros (ARINC, 2006a).

D.36.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros são ignorados

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Desabilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.36.2 Descrição

A cada liberação o processo PROCESS1 pisca o **LED1**. A cada liberação o processo PROCESS2 pisca o **LED2** com a interrupção por outros processos bloqueada, e após n liberações executa um laço por

tempo suficiente para causar uma perda de *deadline* – e portanto o disparo do processo tratador de erros – enquanto essa interrupção ainda encontra-se bloqueada. O processo tratador de erros lê as entradas de erro e invoca **STOP_SELF**, devendo então ser retomada a execução do processo PROCESS2. O processo tratador de erros é executado novamente quando o processo PROCESS1 perde seu *deadline*, devendo ser retomada mais uma vez a execução do processo PROCESS2. O processo PROCESS2 então termina a execução do longo (mas não infinito) laço, desbloqueando a interrupção de processos e retornando ao comportamento inicial. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.36.3 Comportamento esperado

Os *LEDs* devem inicialmente piscar de acordo com a escala de processos. Após n liberações do processo PROCESS2 o **LED2** deve permanecer ligado durante todo o tempo alocado à partição PARTITION1, mesmo quando os *deadlines* dos processos PROCESS1 e PROCESS2 forem perdidos, e após o desligamento do **LED2** a sequência deve ser reiniciada.

D.37 TESTE 130

Nome: GETMYID
Objetivo: Demonstrar o correto funcionamento do serviço **GET_MY_ID**.

D.37.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.37.2 Descrição

Cada tarefa repetidamente atribui um diferente padrão aos *LEDs*, invoca **GET_MY_ID**, verifica o resultado e, caso uma falha seja detectada, entra em um laço infinito. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION3: **LED1** ligado, **LED2** ligado;

- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.37.3 Comportamento esperado

Os *LEDs* devem apresentar repetidamente o padrão de escalonamento (nenhum laço infinito deve ser alcançado).

D.38 TESTE 201

Nome: WAITRESOURCE_SIGNALRESOURCE
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços internos **WAIT_RESOURCE** (com espera por tempo indeterminado) e **SIGNAL_RESOURCE**.

D.38.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/- Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.38.2 Descrição

O processo PROCESS1 repetidamente aguarda por um recurso por tempo indeterminado, e então pisca o **LED1**. O processo PROCESS2 repetidamente pisca o **LED2** a cada 250ms e, a cada n iterações, libera o recurso. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.38.3 Comportamento esperado

O **LED2** deve piscar aproximadamente a cada 250ms e, após $\boldsymbol{n}$ iterações do processo PROCESS2, o **LED1** deve piscar uma vez, reiniciando a sequência.

D.39 TESTE 202

Nome:
WAITRESOURCE_SETPRIORITY_SIGNALRESOURCE

Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços internos **WAIT_RESOURCE** (com espera por tempo indeterminado) e **SIGNAL_RESOURCE** quando a prioridade de um processo bloqueado é alterada antes de sua liberação, caso no qual a ordem de atendimento dos processos bloqueados deve mudar de acordo com essa alteração.

D.39.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/- Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.39.2 Descrição

O processo PROCESS1 repetidamente aguarda por um recurso por tempo indeterminado, e então pisca o **LED1**. O processo PROCESS2 repetidamente aguarda pelo mesmo recurso por tempo indeterminado, e então pisca o **LED2**. O processo ocioso da partição PARTITION1 aguarda durante um longo período e então libera o recurso e, alternadamente, atribui a prioridade do processo PROCESS2 para um valor maior ou menor que a prioridade do processo PROCESS1. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;

- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.39.3 Comportamento esperado

Os *LEDs* devem piscar de forma alternada, indicando que a alteração da prioridade do processo PROCESS2 afeta sua ordem de atendimento em relação ao processo PROCESS1.

D.40 TESTE 203

Nome: WAITRESOURCE_SIGNALRESOURCE_TIMEOUT
Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços internos **WAIT_RESOURCE** (com espera por tempo limitado) e **SIGNAL_RESOURCE**.

D.40.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/- Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.40.2 Descrição

O processo PROCESS1 repetidamente aguarda por um recurso por no máximo 500ms ou até que seja liberado, e então pisca o **LED1**. O processo PROCESS2 repetidamente pisca o **LED2** a cada 250ms e, após n iterações, libera o mesmo recurso. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;
- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;

- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.40.3 Comportamento esperado

O **LED2** deve piscar aproximadamente a cada 250ms, e o **LED1** a cada 500ms ou menos. Quando o **LED1** pisca exatamente 500ms após ser desligado, o processo PROCESS1 foi liberado pela expiração do tempo limite de espera pelo recurso, e quando pisca em um intervalo menor que esse o recurso foi liberado pelo processo PROCESS2.

D.41 TESTE 204

Nome: WAITRESOURCE_SUSPEND_TIMEOUT_RESUME

Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **WAIT_RESOURCE** (com espera por tempo limitado), **SUSPEND** e **RESUME** quando o tempo limite de espera por um recurso expira enquanto o processo encontra-se suspenso, situação na qual esse deve permanecer suspenso até que seja continuado por outro processo.

D.41.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/- Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/- Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.41.2 Descrição

O processo PROCESS1 repetidamente aguarda por um recurso por no máximo 2s e, quando adquire o recurso ou quando o tempo limite é excedido, pisca o **LED1**. A cada liberação o processo PROCESS2 pisca o **LED2**, após $\boldsymbol{n}$ liberações suspende o processo PROCESS1 e após $\boldsymbol{m \approx 3n}$ liberações continua esse mesmo processo. Os valores de $\boldsymbol{n}$, de $\boldsymbol{m}$, do período do processo PROCESS2 e do tempo limite de espera pelo recurso são definidos de forma que o tempo limite de espera expire enquanto o processo PROCESS1 encontra-se suspenso. Os padrões de *LEDs* utilizados pelas tarefas são os seguintes:

- Processo PROCESS1 da partição PARTITION1: **LED1** ligado, **LED2** desligado;

- Processo PROCESS2 da partição PARTITION1: **LED1** desligado, **LED2** ligado;
- Processo ocioso da partição PARTITION1: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION2: **LED1** desligado, **LED2** desligado;
- Processo ocioso da partição PARTITION3: **LED1** desligado, **LED2** desligado;
- Partição ociosa: **LED1** desligado, **LED2** desligado.

D.41.3 Comportamento esperado

O **LED2** deve piscar aproximadamente a cada 250ms, e o **LED1** deve piscar uma vez a cada $\boldsymbol{m}$ liberações do processo PROCESS2, permanecendo desligado quando o tempo máximo de espera pelo recurso expirar já que nesse momento o processo PROCESS1 encontra-se suspenso.

D.42 TESTE 205

Nome: WAITRESOURCE_SUSPEND_SIGNALRESOURCE_RESUME

Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **WAIT_RESOURCE**, **SIGNAL_RESOURCE**, **SUSPEND** e **RESUME** quando um processo é liberado da espera pelo recurso enquanto encontra-se suspenso, situação na qual esse deve permanecer suspenso até que seja continuado por outro processo.

D.42.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.42.2 Descrição

O processo PROCESS2 repetidamente aguarda por um recurso por tempo indeterminado (até que seja liberado), e então pisca o **LED2**. A cada liberação o processo PROCESS1 pisca o **LED1** e, a cada n liberações, alternadamente suspende o processo PROCESS2, libera o recurso ou continua o processo PROCESS2 (nessa ordem).

D.42.3 Comportamento esperado

O **LED1** deve piscar a cada 500ms, e o **LED2** deve piscar a cada $3n$ liberações do processo PROCESS1. Em outras palavras, o **LED2**

só deve piscar depois que o processo PROCESS1 continuar a execução do processo PROCESS2, pois quando o recurso é liberado esse processo ainda encontra-se suspenso.

D.43 TESTE 206

Nome:
WAITRESOURCE_SUSPEND_RESUME_SIGNALRESOURCE

Objetivo: Demonstrar o correto funcionamento da utilização conjunta dos serviços **WAIT_RESOURCE**, **SIGNAL_RESOURCE**, **SUSPEND** e **RESUME** quando um processo é suspenso e continuado enquanto encontra-se bloqueado à espera de um recurso, situação na qual esse deve continuar sua execução apenas quando o recurso for liberado.

D.43.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.43.2 Descrição

O processo PROCESS2 repetidamente aguarda por um recurso por tempo indeterminado (até que seja liberado), e então pisca o **LED2**. A cada liberação o processo PROCESS1 pisca o **LED1** e, a cada $\boldsymbol{n}$ liberações, alternadamente suspende o processo PROCESS2, continua o processo PROCESS2 ou libera o recurso (nessa ordem).

D.43.3 Comportamento esperado

O **LED1** deve piscar a cada 500ms, e o **LED2** deve piscar a cada $\boldsymbol{3n}$ liberações do processo PROCESS1. Em outras palavras, o **LED2**

só deve piscar depois que o processo PROCESS1 liberar o recurso, pois quando o processo PROCESS2 é continuado o recurso ainda não foi liberado.

D.44 TESTE 301

Nome: EVENT
Objetivo: Demonstrar a utilização e o correto funcionamento de eventos (mecanismo de comunicação intrapartição).

D.44.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/- Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.44.2 Descrição

Os processos PROCESS1 e PROCESS2 repetidamente esperam pelo evento, piscam o **LED1** e o **LED2** (respectivamente) e aguardam 500ms e 250ms (respectivamente). O processo ocioso da partição PARTITION1 repetidamente obtém o identificador do evento por seu nome, aguarda durante um período longo, alternadamente habilita ou desabilita o evento (atribui seu estado para *up* ou *down*, respectivamente) e, finalmente, obtém e verifica o estado do evento.

D.44.3 Comportamento esperado

Inicialmente o evento encontra-se em estado *down*, portanto ambos os *LEDs* devem permanecer desligados. Após algum tempo o evento é ativado (assume estado *up*) e, portanto, o **LED1** deve piscar aproximadamente a cada 500ms e o **LED2** aproximadamente a cada 250ms. Mais tarde o evento é restaurado, voltando ao estado *down*, e

portanto ambos os processos devem terminar o processamento atual e bloquear, reiniciando a sequência.

D.45 TESTE 302

Nome: SEMAPHORE_FIFO

Objetivo: Demonstrar a utilização e o correto funcionamento de semáforos (mecanismo de comunicação intrapartição) quando empregada política *FIFO* de enfileiramento de processos.

D.45.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/- Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.45.2 Descrição

Os processos PROCESS1 e PROCESS2 repetidamente esperam no semáforo e então piscam o **LED1** e o **LED2** (respectivamente). O processo ocioso da partição PARTITION1 repetidamente obtém o identificador do semáforo por seu nome, aguarda durante um período curto, libera o semáforo e, finalmente, obtém e verifica o estado do semáforo.

D.45.3 Comportamento esperado

O semáforo tem valor inicial 2, portanto espera-se que inicialmente o **LED1** pisque duas vezes seguidas (visualmente percebido como se piscasse apenas uma vez), já que o processo PROCESS1 possui a maior prioridade e, portanto, o processo PROCESS2 não é iniciado até que PROCESS1 seja bloqueado. Após algum tempo o semáforo é libe-

rado e, assim, o **LED1** deve piscar mais uma vez (já que o processo PROCESS1 consumiu os dois recursos iniciais do semáforo e solicitou aguardo por mais um, quando foi bloqueado). Mais tarde o semáforo é liberado novamente e, uma vez que sua política de enfileiramento é *FIFO*, desta vez o **LED2** deve piscar. A partir deste ponto, os *LEDs* devem piscar alternadamente a cada vez que o semáforo é liberado.

D.46 TESTE 303

Nome: SEMAPHORE_PRIORITY
Objetivo: Demonstrar a utilização e o correto funcionamento de semáforos (mecanismo de comunicação intrapartição) quando empregado enfileiramento de processos por prioridade.

D.46.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/- Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.46.2 Descrição

Os processos PROCESS1 e PROCESS2 repetidamente esperam no semáforo e então piscam o **LED1** e o **LED2** (respectivamente). O processo ocioso da partição PARTITION1 repetidamente obtém o identificador do semáforo por seu nome, aguarda durante um período curto, libera o semáforo e, finalmente, obtém e verifica o estado do semáforo.

D.46.3 Comportamento esperado

O semáforo tem valor inicial 2, portanto espera-se que inicialmente o **LED1** pisque duas vezes seguidas (visualmente percebido como se piscasse apenas uma vez), já que o processo PROCESS1 possui a maior prioridade e, portanto, o processo PROCESS2 não é iniciado até que PROCESS1 seja bloqueado. Após algum tempo o semáforo é libe-

rado e, assim, o **LED1** deve piscar mais uma vez (já que o processo PROCESS1 consumiu os dois recursos iniciais do semáforo e solicitou aguardo por mais um, quando foi bloqueado). Mais tarde o semáforo é liberado novamente e, uma vez que sua política de enfileiramento é por prioridade, o **LED1** deve piscar novamente. A partir deste ponto, o **LED1** deve piscar a cada vez que o semáforo é liberado, e o **LED2** nunca deve ser ligado.

D.47 TESTE 304

Nome: SEMAPHORE_DEADLOCK
Objetivo: Demonstrar a ocorrência de *deadlock* através da utilização de semáforos (mecanismo de comunicação intrapartição).

D.47.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 30
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.47.2 Descrição

A cada liberação o processo PROCESS1 pisca o **LED1**, espera pelo semáforo SEMAPHORE1, cede o processamento (através do serviço **TIMED_WAIT**), espera pelo semáforo SEMAPHORE2, pisca o **LED1** novamente e libera ambos os semáforos. A cada liberação o processo PROCESS2 pisca o **LED2**, espera pelo semáforo SEMAPHORE2, cede o processamento (através do serviço **TIMED_WAIT**), espera pelo semáforo SEMAPHORE1, pisca o **LED2** novamente e libera ambos os semáforos.

D.47.3 Comportamento esperado

Este exemplo causa um *deadlock*, pois o processo PROCESS1 adquire o semáforo SEMAPHORE1 e o processo PROCESS2 adquire o semáforo SEMAPHORE2, e portanto o processo PROCESS1 aguardará por tempo indeterminado por SEMAPHORE2 e o processo PROCESS2

aguardará por tempo indeterminado por SEMAPHORE1. Portanto, inicialmente os *LEDs* devem piscar uma vez e, então, ambos devem desligar permanentemente.

D.48 TESTE 305

Nome: BLACKBOARD
Objetivo: Demonstrar a utilização e o correto funcionamento de *blackboards* (mecanismo de comunicação intrapartição).

D.48.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.48.2 Descrição

Os processos PROCESS1 e PROCESS2 repetidamente leem a *blackboard*, verificam a mensagem lida e então piscam o **LED1** e o **LED2** (respectivamente). O processo ocioso da partição PARTITION1 repetidamente obtém o identificador da *blackboard* por seu nome, aguarda durante um período de duração média, alternadamente mostra uma mensagem na *blackboard* ou limpa-a, e finalmente obtém e verifica o estado da *blackboard*.

D.48.3 Comportamento esperado

A *blackboard* inicia vazia, portanto inicialmente nenhum dos *LEDs* deve piscar. Após algum tempo a *blackboard* é escrita e, portanto, o **LED1** e o **LED2** devem piscar de acordo com a escala de processos. Mais tarde, a *blackboard* é limpa e, portanto, ambos os *LEDs* devem permanecer desligados, reiniciando a sequência.

D.49 TESTE 306

Nome: BUFFER
Objetivo: Demonstrar a utilização e o correto funcionamento de *buffers* (mecanismo de comunicação intrapartição).

D.49.1 Cenário

Sistema	
Monitoramento	Desabilitado

Módulo	
Major frame	100ms

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 20
Processo PROCESS3	Período/capacidade = 1s/100ms Prioridade = 10

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.49.2 Descrição

O processo PROCESS1 repetidamente obtém o identificador do *buffer* por seu nome, envia uma mensagem nele e, finalmente, obtém e verifica seu estado. Os processos PROCESS2 e PROCESS3 repetidamente recebem uma mensagem do *buffer* e então piscam o **LED1** e o **LED2** (respectivamente).

D.49.3 Comportamento esperado

O seguinte comportamento deve ser observado ao longo dos primeiros 2s de execução:

- No tempo 0ms o processo PROCESS1 envia uma mensagem, que será consumida pelo processo PROCESS2 logo depois que o processo PROCESS1 aguardar por seu próximo período (fazendo

com que o **LED1** pisque), e quando o processo PROCESS2 esperar por seu próximo período o processo PROCESS3 tenta receber uma mensagem mas o *buffer* está vazio, portanto o processo PROCESS3 será bloqueado;

- No tempo 250ms o processo PROCESS1 envia outra mensagem, liberando o processo PROCESS3 (fazendo com que o **LED2** pisque);
- No tempo 500ms o processo PROCESS1 envia uma nova mensagem, que será consumida pelo processo PROCESS2 logo depois que o processo PROCESS1 aguardar por seu próximo período (fazendo com que o **LED1** pisque novamente);
- No tempo 750ms o processo PROCESS1 envia mais uma mensagem, porém nenhum processo aguarda por ela e portanto nenhum *LED* é acionado;
- No tempo 1000ms o processo PROCESS1 envia mais uma mensagem, e portanto os processos PROCESS2 e PROCESS3 consomem mensagens (piscando o **LED1** e, em seguida, o **LED2**);
- No tempo 1250ms o processo PROCESS1 envia mais uma mensagem;
- No tempo 1500ms o processo PROCESS1 envia mais uma mensagem e o processo PROCESS2 consome uma das mensagens enfileiradas (piscando o **LED1**);
- No tempo 1750ms o processo PROCESS1 envia mais uma mensagem;
- No tempo 2000ms mais uma mensagem é enviada e os processos PROCESS2 e PROCESS3 consomem duas mensagens, piscando o **LED1** e, depois, o **LED2**.

Uma vez que o processo PROCESS1 envia uma nova mensagem a cada 250ms e os processos PROCESS2 e PROCESS3, que as consomem, têm períodos mais longos que esse, o *buffer* irá em algum momento ser completamente preenchido. Portanto, o comportamento a ser observado a longo prazo nos *LEDs* é o padrão imposto pelo escalonamento do conjunto de processos da partição PARTITION1.

D.50 TESTE 307

Nome: BUFFER_FIFO
Objetivo: Demonstrar a utilização e o correto funcionamento de *buffers* (mecanismo de comunicação intrapartição) quando empregada política *FIFO* de enfileiramento de processos.

D.50.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20
Processo PROCESS3	Período/capacidade = -/- Prioridade = 10
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.50.2 Descrição

O processo PROCESS1 repetidamente obtém o identificador do *buffer* por seu nome, envia uma mensagem nele e, finalmente, obtém e verifica seu estado. Os processos PROCESS2 e PROCESS3 repetidamente recebem uma mensagem do *buffer* e então piscam o **LED1** e o **LED2** (respectivamente).

D.50.3 Comportamento esperado

A cada 500ms o **LED1** e o **LED2** devem piscar alternadamente, iniciando com o **LED1** pois o processo PROCESS2 tem prioridade mais alta que o processo PROCESS3 e a política de enfileiramento do *buffer* é *FIFO*. Esse comportamento deve ser repetido indefinidamente.

D.51 TESTE 308

Nome: BUFFER_PRIORITY
Objetivo: Demonstrar a utilização e o correto funcionamento de *buffers* (mecanismo de comunicação intrapartição) quando empregado enfileiramento de processos por prioridade.

D.51.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20
Processo PROCESS3	Período/capacidade = -/- Prioridade = 10
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.51.2 Descrição

O processo PROCESS1 repetidamente obtém o identificador do *buffer* por seu nome, envia uma mensagem nele e, finalmente, obtém e verifica seu estado. Os processos PROCESS2 e PROCESS3 repetidamente recebem uma mensagem do *buffer* e então piscam o **LED1** e o **LED2** (respectivamente).

D.51.3 Comportamento esperado

A cada 500ms o **LED1** deve piscar, e o **LED2** nunca deve ser acionado, pois o processo PROCESS2 tem prioridade mais alta que o PROCESS3 e a política de enfileiramento do *buffer* é por prioridade. Esse comportamento deve ser repetido indefinidamente.

D.52 TESTE 309

Nome: BUFFER_TIMEOUT_RECEIVE
Objetivo: Demonstrar a utilização e o correto funcionamento de *buffers* (mecanismo de comunicação intrapartição) com espera pelo recebimento de mensagens por tempo limitado.

D.52.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = -/- Prioridade = 20
Processo PROCESS3	Período/capacidade = -/- Prioridade = 10
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.52.2 Descrição

O processo PROCESS1 repetidamente obtém o identificador do *buffer* por seu nome, envia uma mensagem nele e, finalmente, obtém e verifica seu estado. Os processos PROCESS2 e PROCESS3 repetidamente recebem uma mensagem do *buffer* com limite de tempo de 250ms e 100ms (respectivamente) e piscam o **LED1** e o **LED2** (respectivamente).

D.52.3 Comportamento esperado

O **LED1** deve piscar no mínimo a cada 500ms, podendo piscar antes quando o processo PROCESS2 recebe uma mensagem. O **LED2** deve piscar no mínimo a cada 250ms, podendo piscar antes quando

o processo PROCESS3 recebe uma mensagem. Esse comportamento deve ser repetido indefinidamente.

D.53 TESTE 310

Nome: BUFFER_TIMEOUT_SEND
Objetivo: Demonstrar a utilização e o correto funcionamento de *buffers* (mecanismo de comunicação intrapartição) com espera pelo envio de mensagens por tempo limitado.

D.53.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = -/- Prioridade = 30
Processo PROCESS2	Período/capacidade = 1s/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.53.2 Descrição

O processo PROCESS1 repetidamente obtém o identificador do *buffer* por seu nome, envia uma mensagem nele com limite de tempo de 250ms, pisca o **LED1** e, finalmente, obtém e verifica o estado do *buffer*. O processo PROCESS2 repetidamente recebe uma mensagem do *buffer* e então pisca o **LED2**.

D.53.3 Comportamento esperado

O **LED2** deve piscar aproximadamente a cada 1000ms, enquanto o **LED1** deve piscar no mínimo a cada 250ms, podendo piscar antes caso o processo PROCESS2 receba uma mensagem. Esse comportamento deve ser repetido indefinidamente.

D.54 TESTE 311

Nome: BUFFER_FULL_EMPTY
Objetivo: Demonstrar o correto funcionamento de *buffers* (mecanismo de comunicação intrapartição) quando a fila de mensagens encontra-se cheia ou vazia.

D.54.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 100ms/- Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/- Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.54.2 Descrição

A cada liberação o processo PROCESS2 recebe uma mensagem do *buffer* e pisca o **LED2**, podendo bloquear caso esse encontre-se vazio. O processo PROCESS1 mantém um contador de liberações, e possui um diferente comportamento de acordo com o valor desse contador:

- Nas 20 primeiras liberações, envia uma mensagem no *buffer*, podendo bloquear caso esse encontre-se cheio, e pisca o **LED1**;
- Entre as liberações 20 e 99, não executa qualquer ação;
- Na liberação de número 100, reinicia a sequência atribuindo zero ao contador de liberações.

D.54.3 Comportamento esperado

Inicialmente o **LED1** deve piscar a cada 100ms e o **LED2** a cada 500ms até que o *buffer* esteja cheio. Então ambos os *LEDs* devem piscar a cada 500ms até que o processo PROCESS1 pare de enviar mensagens no *buffer*, quando o **LED1** parará de piscar. Depois disso, o **LED2** deve piscar a cada 500ms até que o *buffer* esvazie, quando ambos os *LEDs* devem permanecer desligados até que a sequência seja reiniciada. Esse comportamento deve ser repetido indefinidamente.

D.55 TESTE 312

Nome: SAMPLINGPORT_STANDARD

Objetivo: Demonstrar o correto funcionamento de portas de amostragem (mecanismo de comunicação interpartição) conectadas a partições do mesmo módulo.

D.55.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30

Partição SYSTEMPARTITION_IO	
Período/duração	100ms/10ms
Escala	(85ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo SAMPLINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo QUEUINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo ETHERNET	Período/capacidade = -/- Prioridade = 10

D.55.2 Descrição

O processo PROCESS1 da partição PARTITION1 escreve em uma porta de amostragem a cada 500ms, enquanto os processos PROCESS1 das partições PARTITION2 e PARTITION3 leem de portas de amostragem a cada 250ms e, se o valor é válido, piscam o *LED1* e, se inválido, piscam o *LED2*.

D.55.3 Comportamento esperado

Os *LEDs* devem piscar duas vezes alternadamente a cada 250ms (uma vez para a partição PARTITION2 e outra para a partição PARTITION3). Isso porque a porta é escrita a cada 500ms mas seu período de atualização é de 250ms e, por isso, o valor lido apenas é considerado válido a cada 500ms.

D.56 TESTE 313

Nome: SAMPLINGPORT_PSEUDO_MODULE1

Objetivo: Demonstrar o correto funcionamento de portas de amostragem (mecanismo de comunicação interpartição) conectadas a partições de dois módulos diferentes (utilizado em conjunto com o teste **SAMPLINGPORT_PSEUDO_MODULE2**).

D.56.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição SYSTEMPARTITION_IO	
Período/duração	100ms/10ms
Escala	(85ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo SAMPLINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo QUEUINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo ETHERNET	Período/capacidade = -/- Prioridade = 10

D.56.2 Descrição

Os processos denominados PROCESS1 das partições PARTITION1, PARTITION2 e PARTITION3 escrevem em portas de amostragem a cada 250ms, enquanto os processos denominados PROCESS2 dessas mesmas partições leem de portas de amostragem a cada 500ms e, se um valor válido é lido, piscam o *LED1* e, em caso contrário, piscam o *LED2*. As tabelas de conexão dos módulos envolvidos neste teste definem que o processo PROCESS1 de todas as partições escreve na porta que é lida pelo processo PROCESS2 da partição de mesmo nome no módulo oposto. Este teste deve ser utilizado em conjunto com o teste **SAMPLINGPORT_PSEUDO_MODULE2**.

D.56.3 Comportamento esperado

Enquanto os módulos estiverem **desconectados**, o **LED2** deve piscar três vezes a cada 500ms (uma vez para cada partição), pois as portas de destino não estarão sendo atualizadas e, portanto, seus valores serão considerados inválidos.

Enquanto os módulos estiverem **conectados**, o **LED1** deve piscar três vezes a cada 500ms (uma vez para cada partição), pois as portas de destino estarão sendo atualizadas a cada 250ms e, portanto, seus valores serão considerados válidos.

D.57 TESTE 314

Nome: SAMPLINGPORT_PSEUDO_MODULE2

Objetivo: Demonstrar o correto funcionamento de portas de amostragem (mecanismo de comunicação interpartição) conectadas a partições de dois módulos diferentes (utilizado em conjunto com o teste **SAMPLINGPORT_PSEUDO_MODULE1**).

D.57.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição SYSTEMPARTITION_IO	
Período/duração	100ms/10ms
Escala	(85ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo SAMPLINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo QUEUINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo ETHERNET	Período/capacidade = -/- Prioridade = 10

D.57.2 Descrição

Os processos denominados PROCESS1 das partições PARTITION1, PARTITION2 e PARTITION3 escrevem em portas de amostragem a cada 250ms, enquanto os processos denominados PROCESS2 dessas mesmas partições leem de portas de amostragem a cada 500ms e, se um valor válido é lido, piscam o *LED1* e, em caso contrário, piscam o *LED2*. As tabelas de conexão dos módulos envolvidos neste teste definem que o processo PROCESS1 de todas as partições escreve na porta que é lida pelo processo PROCESS2 da partição de mesmo nome no módulo oposto. Este teste deve ser utilizado em conjunto com o teste **SAMPLINGPORT_PSEUDO_MODULE1**.

D.57.3 Comportamento esperado

Enquanto os módulos estiverem **desconectados**, o **LED2** deve piscar três vezes a cada 500ms (uma vez para cada partição), pois as portas de destino não estarão sendo atualizadas e, portanto, seus valores serão considerados inválidos.

Enquanto os módulos estiverem **conectados**, o **LED1** deve piscar três vezes a cada 500ms (uma vez para cada partição), pois as portas de destino estarão sendo atualizadas a cada 250ms e, portanto, seus valores serão considerados válidos.

D.58 TESTE 315

Nome: QUEUINGPORT_STANDARD

Objetivo: Demonstrar o correto funcionamento de portas de enfileiramento (mecanismo de comunicação interpartição) conectadas a partições do mesmo módulo.

D.58.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 250ms/100ms Prioridade = 30

Partição SYSTEMPARTITION_IO	
Período/duração	100ms/10ms
Escala	(85ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo SAMPLINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo QUEUINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo ETHERNET	Período/capacidade = -/- Prioridade = 10

D.58.2 Descrição

O processo PROCESS1 da partição PARTITION1 envia em duas diferentes portas de enfileiramento a cada 500ms (uma conectada à partição PARTITION2 e outra à partição PARTITION3). Os processos chamados PROCESS1 das partições PARTITION2 e PARTITION3 solicitam recebimento da porta de enfileiramento atribuída às suas partições a cada 250ms com limite de tempo de 100ms e, se uma a mensagem é recebida corretamente, piscam o *LED1* e, se o tempo de espera expira ou a mensagem está incorreta, piscam o *LED2*.

D.58.3 Comportamento esperado

Os *LEDs* devem piscar alternadamente a cada 250ms, já que os envios ocorrem a cada 500ms e os recebimentos a cada 250ms e, portanto, os recebimentos expiram a cada 500ms.

D.59 TESTE 316

Nome: QUEUINGPORT_PSEUDO_MODULE1

Objetivo: Demonstrar o correto funcionamento de portas de enfileiramento (mecanismo de comunicação interpartição) conectadas a partições de dois módulos diferentes (utilizado em conjunto com o teste **QUEUINGPORT_PSEUDO_MODULE2**).

D.59.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição SYSTEMPARTITION_IO	
Período/duração	100ms/10ms
Escala	(85ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo SAMPLINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo QUEUINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo ETHERNET	Período/capacidade = -/- Prioridade = 10

D.59.2 Descrição

Os processos denominados PROCESS1 das partições PARTITION1, PARTITION2 e PARTITION3 enviam em portas de amostragem a cada 500ms, enquanto os processos denominados PROCESS2 dessas mesmas partições solicitam recebimento em portas de amostragem também a cada 500ms com limite de tempo de 100ms e, se um valor válido é recebido, piscam o *LED1* e, em caso contrário, piscam o *LED2*. As tabelas de conexão dos módulos envolvidos neste teste definem que o processo PROCESS1 de todas as partições envia na porta em que o processo PROCESS2 da partição de mesmo nome no módulo oposto solicita recebimento. Este teste deve ser utilizado em conjunto com o teste **QUEUINGPORT_PSEUDO_MODULE2**.

D.59.3 Comportamento esperado

Enquanto os módulos estiverem **desconectados**, o **LED2** deve piscar três vezes a cada 500ms (uma vez para cada partição), pois as portas de destino não estarão sendo alimentadas e, portanto, as solicitações de recebimento expirarão.

Enquanto os módulos estiverem **conectados**, o **LED1** deve piscar três vezes a cada 500ms (uma vez para cada partição), pois as portas de destino estarão sendo alimentadas e, portanto, as mensagens serão recebidas com sucesso.

D.60 TESTE 317

Nome: QUEUINGPORT_PSEUDO_MODULE2
Objetivo: Demonstrar o correto funcionamento de portas de enfileiramento (mecanismo de comunicação interpartição) conectadas a partições de dois módulos diferentes (utilizado em conjunto com o teste **QUEUINGPORT_PSEUDO_MODULE1**).

D.60.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/100ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 500ms/100ms Prioridade = 30

Partição SYSTEMPARTITION_IO	
Período/duração	100ms/10ms
Escala	(85ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo SAMPLINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo QUEUINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo ETHERNET	Período/capacidade = -/- Prioridade = 10

D.60.2 Descrição

Os processos denominados PROCESS1 das partições PARTITION1, PARTITION2 e PARTITION3 enviam em portas de amostragem a cada 500ms, enquanto os processos denominados PROCESS2 dessas mesmas partições solicitam recebimento em portas de amostragem também a cada 500ms com limite de tempo de 100ms e, se um valor válido é recebido, piscam o *LED1* e, em caso contrário, piscam o *LED2*. As tabelas de conexão dos módulos envolvidos neste teste definem que o processo PROCESS1 de todas as partições envia na porta em que o processo PROCESS2 da partição de mesmo nome no módulo oposto solicita recebimento. Este teste deve ser utilizado em conjunto com o teste **QUEUINGPORT_PSEUDO_MODULE1**.

D.60.3 Comportamento esperado

Enquanto os módulos estiverem **desconectados**, o **LED2** deve piscar três vezes a cada 500ms (uma vez para cada partição), pois as portas de destino não estarão sendo alimentadas e, portanto, as solicitações de recebimento expirarão.

Enquanto os módulos estiverem **conectados**, o **LED1** deve piscar três vezes a cada 500ms (uma vez para cada partição), pois as portas de destino estarão sendo alimentadas e, portanto, as mensagens serão recebidas com sucesso.

D.61 TESTE 401

Nome: HEALTHMONITORING_CURRENTSYSTEMSTATE
Objetivo: Demonstrar a corretude dos estados do sistema detectados pelo núcleo do SO, que são empregados pelo mecanismo de monitoramento (*health monitoring*) do SO.

D.61.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo
Módulo	
Major frame	100ms
HM callback	Habilitado
Tratamento de erros	Erros são ignorados
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.61.2 Descrição

A partição ociosa, os processos ociosos e os processos de aplicação de todas as partições repetidamente verificam o estado atual do sistema. Após $\boldsymbol{n}$ liberações, o processo PROCESS1 lança um erro de aplicação que faz com que o processo tratador de erros da partição PARTITION1 seja iniciado. O processo tratador de erros da partição PARTITION1 verifica o estado atual do sistema e lança um erro de aplicação, causando

a inicialização do *HM callback* da partição. O *HM callback* da partição PARTITION1 verifica o estado atual do sistema e lança um erro de aplicação, fazendo com que o *HM callback* do módulo seja iniciado. O *HM callback* do módulo verifica o estado atual do sistema e lança um erro de aplicação, fazendo com que a ação definida para tratamento desse tipo de erro a nível de sistema seja imediatamente executada, no caso a reinicialização do módulo, fazendo com que a sequência seja repetida indefinidamente. Caso um estado inválido seja detectado por qualquer uma das tarefas, a tarefa passa a executar um laço infinito ligando ambos os *LEDs*.

D.61.3 Comportamento esperado

Nenhum *LED* deve ser acionado durante a execução do teste.

D.62 TESTE 402

Nome:
HEALTHMONITORING_PROPAGATION_APPLICATIONERROR
Objetivo: Demonstrar o correto funcionamento do mecanismo de monitoramento (*health monitoring*) do SO, utilizando para isso erros de aplicação lançados a partir de todas as tarefas do módulo.

D.62.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.62.2 Descrição

A partição ociosa e os processos ociosos das partições repetidamente desligam ambos os *LEDs*. O processo PROCESS1 pisca o **LED1** e, após ***n*** liberações, lança um erro de aplicação que faz com que o processo tratador de erros da partição PARTITION1 seja iniciado. O processo tratador de erros da partição PARTITION1 liga o

LED1 por um longo período e então lança um erro de aplicação, causando a inicialização do *HM callback* da partição. O *HM callback* da partição PARTITION1 liga o **LED2** por um longo período e lança um erro de aplicação, fazendo com que o *HM callback* do módulo seja iniciado. O *HM callback* do módulo liga ambos os *LEDs* durante um longo período e lança um erro de aplicação, fazendo com que a ação definida para tratamento desse tipo de erro a nível de sistema seja imediatamente executada, no caso a reinicialização do módulo, fazendo com que a sequência seja repetida indefinidamente.

D.62.3 Comportamento esperado

O seguinte padrão deve ser apresentado repetidamente:

- O **LED1** deve piscar algumas vezes, até que o processo PROCESS1 lance um erro;
- Apenas o **LED1** deve permanecer ligado por algum tempo, **RESPEITANDO O PARTICIONAMENTO** (processo tratador de erros da partição PARTITION1);
- Apenas o **LED2** deve permanecer ligado por algum tempo, **RESPEITANDO O PARTICIONAMENTO** (*HM callback* da partição PARTITION1);
- Ambos os *LEDs* devem permanecer ligados por algum tempo, **SEM RESPEITAR O PARTICIONAMENTO** (*HM callback* do módulo).

D.63 TESTE 403

Nome:
HEALTHMONITORING_PROPAGATION_STACKOVERFLOW

Objetivo: Demonstrar o correto funcionamento do mecanismo de monitoramento (*health monitoring*) do SO, utilizando para isso erros de estouro de pilha gerados a partir de todas as tarefas do módulo através da chamada de uma recursão infinita. Em conjunto com o **Teste 404**, demonstra ainda o funcionamento da diferenciação entre erros de estouro de pilha e de violação de memória.

D.63.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.63.2 Descrição

A partição ociosa e os processos ociosos das partições repetidamente desligam ambos os *LEDs*. O processo PROCESS1 pisca o **LED1**

e, após ***n*** liberações, gera um erro de estouro de pilha que faz com que o processo tratador de erros da partição PARTITION1 seja iniciado. O processo tratador de erros da partição PARTITION1 liga o **LED1** por um longo período e então gera um erro de estouro de pilha, causando a inicialização do *HM callback* da partição. O *HM callback* da partição PARTITION1 liga o **LED2** por um longo período e gera um erro de estouro de pilha, fazendo com que o *HM callback* do módulo seja iniciado. O *HM callback* do módulo liga ambos os *LEDs* durante um longo período e gera um erro de estouro de pilha, fazendo com que a ação definida para tratamento desse tipo de erro a nível de sistema seja imediatamente executada, no caso a reinicialização do módulo, fazendo com que a sequência seja repetida indefinidamente.

D.63.3 Comportamento esperado

O seguinte padrão deve ser apresentado repetidamente:

- O **LED1** deve piscar algumas vezes, até que o processo PROCESS1 lance um erro;
- Apenas o **LED1** deve permanecer ligado por algum tempo, **RESPEITANDO O PARTICIONAMENTO** (processo tratador de erros da partição PARTITION1);
- Apenas o **LED2** deve permanecer ligado por algum tempo, **RESPEITANDO O PARTICIONAMENTO** (*HM callback* da partição PARTITION1);
- Ambos os *LEDs* devem permanecer ligados por algum tempo, **SEM RESPEITAR O PARTICIONAMENTO** (*HM callback* do módulo).

D.64 TESTE 404

Nome:
HEALTHMONITORING_PROPAGATION_MEMORYVIOLATION

Objetivo: Demonstrar o correto funcionamento do mecanismo de monitoramento (*health monitoring*) do SO, utilizando para isso erros de violação de memória gerados a partir de todas as tarefas do módulo através do acesso a um ponteiro nulo. Em conjunto com o **Teste 403**, demonstra ainda o funcionamento da diferenciação entre erros de estouro de pilha e de violação de memória.

D.64.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados

D.64.2 Descrição

A partição ociosa e os processos ociosos das partições repetidamente desligam ambos os *LEDs*. O processo PROCESS1 pisca o

LED1 e, após ***n*** liberações, gera um erro de violação de memória que faz com que o processo tratador de erros da partição PARTITION1 seja iniciado. O processo tratador de erros da partição PARTITION1 liga o **LED1** por um longo período e então gera um erro de violação de memória, causando a inicialização do *HM callback* da partição. O *HM callback* da partição PARTITION1 liga o **LED2** por um longo período e gera um erro de violação de memória, fazendo com que o *HM callback* do módulo seja iniciado. O *HM callback* do módulo liga ambos os *LEDs* durante um longo período e gera um erro de violação de memória, fazendo com que a ação definida para tratamento desse tipo de erro a nível de sistema seja imediatamente executada, no caso a reinicialização do módulo, fazendo com que a sequência seja repetida indefinidamente.

D.64.3 Comportamento esperado

O seguinte padrão deve ser apresentado repetidamente:

- O **LED1** deve piscar algumas vezes, até que o processo PROCESS1 lance um erro;
- Apenas o **LED1** deve permanecer ligado por algum tempo, **RESPEITANDO O PARTICIONAMENTO** (processo tratador de erros da partição PARTITION1);
- Apenas o **LED2** deve permanecer ligado por algum tempo, **RESPEITANDO O PARTICIONAMENTO** (*HM callback* da partição PARTITION1);
- Ambos os *LEDs* devem permanecer ligados por algum tempo, **SEM RESPEITAR O PARTICIONAMENTO** (*HM callback* do módulo).

D.65 TESTE 501

Nome: FLOATINGPOINT
Objetivo: Demonstrar a corretude do procedimento de salvamento e restauração de contexto do coprocessador de ponto flutuante (caso um esteja sendo utilizado).

D.65.1 Cenário

Sistema	
Monitoramento	Desabilitado
Módulo	
Major frame	100ms
Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
Processo PROCESS1	Período/capacidade = 500ms/250ms Prioridade = 30
Processo PROCESS2	Período/capacidade = 250ms/100ms Prioridade = 20
Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)

D.65.2 Descrição

Todas as tarefas do módulo executam operações sobre variáveis numéricas de ponto flutuante (soma, subtração, multiplicação e divisão) e, se algum erro for detectado no resultado dessas operações, essas entram em um laço infinito ligando ambos os *LEDs*.

D.65.3 Comportamento esperado

Nenhum *LED* deve ser acionado durante a execução do teste.

D.66 TESTE 502

Nome: SAMPLINGPORT_SENSOR_MODULE1
Objetivo: Demonstrar a utilização prática de portas de amostragem (mecanismo de comunicação interpartição) conectadas a partições de dois módulos diferentes para transmissão de amostras de um sensor (utilizado em conjunto com o teste **SAMPLINGPORT_SENSOR_MODULE2**).

D.66.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 100ms/50ms Prioridade = 10

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados

Partição SYSTEMPARTITION_IO	
Período/duração	100ms/10ms
Escala	(85ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo SAMPLINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo QUEUINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo ETHERNET	Período/capacidade = -/- Prioridade = 10

D.66.2 Descrição

O processo PROCESS1 da partição PARTITION1 inicializa a entrada *ADC* do processador chamada **AIN5** (pino **P9.36** da plataforma *BeagleBone*) e então, a cada 100ms, lê uma amostra gerada por esse dispositivo, escreve-a na porta de amostragem e pisca o **LED1**. Este teste deve ser utilizado em conjunto com o teste **SAMPLING-PORT_SENSOR_MODULE2**.

D.66.3 Comportamento esperado

O **LED1** deve piscar a cada 100ms.

D.67 TESTE 503

Nome: SAMPLINGPORT_SENSOR_MODULE2

Objetivo: Demonstrar a utilização prática de portas de amostragem (mecanismo de comunicação interpartição) conectadas a partições de dois módulos diferentes para transmissão de amostras de um sensor (utilizado em conjunto com o teste **SAMPLINGPORT_SENSOR_MODULE1**).

D.67.1 Cenário

Sistema	
Monitoramento	Habilitado
Tratamento de erros	Erros reiniciam o módulo

Módulo	
Major frame	100ms
HM callback	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION1	
Período/duração	50ms/25ms
Escala	(0ms, 25ms), (50ms, 25ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados
Processo PROCESS1	Período/capacidade = 100ms/50ms Prioridade = 10

Partição PARTITION2	
Período/duração	50ms/10ms
Escala	(25ms, 10ms), (75ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados

Partição PARTITION3	
Período/duração	100ms/10ms
Escala	(35ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Desabilitado
Tratamento de erros	Erros são ignorados

Partição SYSTEMPARTITION_IO	
Período/duração	100ms/10ms
Escala	(85ms, 10ms)
HM callback	Habilitado
Processo tratador de erros	Habilitado
Tratamento de erros	Erros são ignorados
Processo SAMPLINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo QUEUINGPORTS	Período/capacidade = -/- Prioridade = 10
Processo ETHERNET	Período/capacidade = -/- Prioridade = 10

D.67.2 Descrição

O processo PROCESS1 da partição PARTITION1 inicializa a saída *PWM* do processador chamada **EHRPWM1A** (pino **P9.14** da plataforma *BeagleBone*) e então, a cada 100ms, lê a amostra atual da porta de amostragem e, se válida, atribui o *duty cycle* da saída *PWM* de acordo com a amostra e pisca o **LED1**, senão atribui *duty cycle* zero e pisca o **LED2**. Este teste deve ser utilizado em conjunto com o teste **SAMPLINGPORT_SENSOR_MODULE1**.

D.67.3 Comportamento esperado

Enquanto os módulos estiverem **desconectados** o **LED2** deve piscar a cada 100ms, pois a porta de amostragem não estará sendo atualizada e, portanto, seu valor será considerado inválido.

Enquanto os módulos estiverem **conectados** o **LED1** deve piscar a cada 100ms e o *duty cycle* da saída *PWM* deve variar de acordo com as amostras coletadas pelo módulo emissor, pois a porta de amostragem estará sendo atualizada e, portanto, seu valor será considerado válido.