UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Habilitando processamento de tempo real em sistemas embarcados ROS2

Lucas da Silva Medeiros

Itajubá, Dezembro de 2017

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Lucas da Silva Medeiros

Habilitando processamento de tempo real em sistemas embarcados ROS2

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação

Área de Concentração: Matemática da Computação

Orientador: Prof. Dr. Guilherme Sousa Bastos
Coorientador: Prof. Dr. Rodrigo Maximiano Antunes de Almeida

Dezembro de 2017
Itajubá - MG

Dedico esse trabalho a memória de meu pai,
o homem que me ensinou os valores da honestidade,
do trabalho duro e de ajudar sem olhar quem.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, irmãos e demais familiares pelo apoio incondicional.

Um agradecimento especial a minha mãe, pelo apoio desde sempre, e por ser um exemplo de pessoa batalhadora que cresceu na vida.

Ao meu orientador, Guilherme Sousa Bastos, por toda ajuda, ensinamentos, sermões, dicas que se mostraram valiosas ao termino desse projeto e principalmente paciência nos meus momentos mais complicados durante o desenvolvimento desse trabalho.

Ao meu coorientador, Rodrigo Maximiano Antunes de Almeida, por toda ajuda, ensinamentos, e as boas horas de conversa sobre esse trabalho e temas que tenho interesse.

À banca examinadora, por aceitar estar presente e avaliar o meu trabalho, fornecendo correções e dicas que serão utilizadas em outros trabalhos na minha vida.

Aos amigos e colegas do LRO, em especial ao Adriano, Alyson, Audeliano e Pedro, que estiveram comigo desde o início do mestrado e sempre ajudaram com o meu aprendizado no ROS. Ao Ricardo, por ajudar na correção e submissão deste trabalho para o livro ROSBook, 2018.

À Capes pelo apoio financeiro durante 2 anos.

As valiosas correções indicadas pela banca avaliadora que, em adição ao meu orientador e co-orientador, era composta também pelos professores Dr. Walter Fetter Lages (UFRGS) e Dr. Carlos Henrique Valério de Moraes (Unifei).

E a Unifei, essa universidade pelo qual tenho eterna paixão. Apresentou-me amigos pra vida, amigos de república e aos projetos Cheetah Racing de Formula SAE e EcoVeículo, nos quais tive maior prazer de participar durante minha graduação, sem mencionar o suporte com equipamentos durante o mestrado.

"Comece fazendo o que é necessário, depois o que é possível, e de repente você estará fazendo o impossível."

São Francisco de Assis

RESUMO

Processamento em tempo real é um importante componente no controle digital de processos, especialmente na área de robótica, onde muitas aplicações requerem processamento de dados à medida que os dados chegam e também para controle de atuadores. Assim, esse trabalho busca integrar o FreeRTPS, uma implementação portável e minimalista de um publicador/assinante de tempo real (RTPS), que provê uma opção para aplicações ROS2 em sistemas embarcados onde o tamanho da memória é um fator crítico, com o FreeRTOS, um sistema operacional embarcado de tempo real para sistemas pequenos. Como resultado, tem-se um sistema que possui ferramentas para se efetuar sensoriamento e processamento em tempo real enquanto pode efetuar a troca de mensagens com outras aplicações ROS2 na rede Ethernet.

Palavras-chave: ROS, Sistemas embarcados, Tempo real, ROS2, Robot operating system.

ABSTRACT

Real-time processing is an important component in digital process control, especially in the field of robotics, where many applications require data processing as data arrives and also for actuators control. Thus, this work seeks to integrate FreeRTPS, a portable and minimalist implementation of a real-time publisher/subscriber (RTPS), which provides an option for ROS2 applications in embedded systems where memory size is a critical factor, with FreeRTOS, a real-time embedded operating system for small systems. As a result, offer a system that offer tools to perform real-time sensing and processing while it can exchange messages with other ROS2 applications on the Ethernet network.

Key-words: ROS, Embedded systems, Real time, ROS2, Robot operating system.

SUMÁRIO

Sumário 6

Lista de ilustrações 8

Lista de tabelas 10

1 INTRODUÇÃO 13
1.1 Objetivos 14
1.2 Motivação 15
1.3 Contribuições 16
1.4 Estrutura da dissertação 16

2 REFERENCIAL TEÓRICO 17
2.1 Sistemas embarcados 17
2.1.1 Sistemas operacionais 18
2.1.2 Sistemas operacionais embarcados 21
2.1.3 Tempo real 24
2.1.4 Controle digital de sistemas 25
2.2 Redes de computadores 28
2.2.1 Data Distributed System 30
2.2.2 RTPS 31
2.2.2.1 FreeRTPS 34
2.3 ROS 34
2.3.1 ROS2 37
2.4 Trabalhos relacionados 40

3 DESENVOLVIMENTO 45
3.1 Modelo proposto 45
3.1.1 Arquitetura original do FreeRTPS 45
3.1.2 Arquitetura proposta 48
3.2 Implementação 56
3.2.1 Implementação de mensagens multicast 61

4 EXPERIMENTOS E RESULTADOS 64
4.1 Hardware utilizado 64
4.2 Teste de tempo real 66
4.3 Teste de desempenho em rede 72

4.4 Consumo de recursos 77
4.5 Teste de Portabilidade 81

5 CONCLUSÃO E TRABALHOS FUTUROS 84
5.1 Conclusão 84
5.2 Trabalhos futuros 86

REFERÊNCIAS 87

APÊNDICE A – GUIA, UTILIZANDO RTPS+RTOS NA STM . . 92

APÊNDICE B – CÓDIGO, TESTE TEMPO REAL 94

APÊNDICE C – GUIA, PORTANDO O RTPS+RTOS PARA A TEXAS STELLARIS LM3S6965 EVAL. BOARD . 101

LISTA DE ILUSTRAÇÕES

Figura 1 – Interfaceamento realizado pelo sistema operacional 19
Figura 2 – Multitasking 20
Figura 3 – Escalonamento de processos por fatia de tempo 21
Figura 4 – Sistemas embarcados de tempo real 22
Figura 5 – Sinal PWM e seu correspondente analógico 26
Figura 6 – Módulo do RTPS 32
Figura 7 – Sistema de arquivos do ROS 36
Figura 8 – A interface middleware 39
Figura 9 – Formato do pacote 41
Figura 10 – Estrutura do rosserial 41
Figura 11 – Estrutura do rosbridge 42
Figura 12 – Conceito do ROS STM32 43
Figura 13 – Estrutura do ROS STM32 44
Figura 14 – Estutura do ROS 2.0 NuttX 44
Figura 15 – Diretório do FreeRTPS 46
Figura 16 – Modelo simples da arquitetura proposta 49
Figura 17 – Caminho das mensagens recebidas 51
Figura 18 – Caminho das mensagens enviadas 52
Figura 19 – Modelo FreeRTPS 52
Figura 20 – Modelo proposto, FreeRTPS+FreeRTOS 53
Figura 21 – STM32F4DISCOVERY 64
Figura 22 – STM32F4 *Discovery Base Board* 65
Figura 23 – *Stellaris LM3S6965 Evaluation Board* 66
Figura 24 – Circuito utilizado para efetuar o teste do controlador PID 67
Figura 25 – Controle de tensão de um circuito RC em malha aberta 71
Figura 26 – Controle de tensão de um circuito RC utilizando controlador PID . . . 72
Figura 27 – Modelo de blocos de um controlador PID equivalente ao sistema testado 73
Figura 28 – Resultado obtido na simulação utilizando os parâmetros do teste real . 73
Figura 29 – Comparação do sinal simulado ao sinal obtido no teste real 74
Figura 30 – Interface Wireshark 75
Figura 31 – Consumo de recurso com um processo PID à 100 Hz e 1 publicador do tipo *string* a 100 Hz 78
Figura 32 – Consumo de recurso com um processo PID à 1000 Hz e 1 publicador do tipo *string* a 1000 Hz 79
Figura 33 – Consumo de recurso com um processo PID à 1000 Hz 80

Figura 34 – Consumo de recurso com um processo PID à 1000 Hz e 5 publicadores do tipo *string* a 1000 Hz 81
Figura 35 – Tela inicial da IDE IAR Embedded Workbench 102
Figura 36 – Criando novo projeto 102
Figura 37 – Seleção do microcontrolador utilizado 103
Figura 38 – C Compiler Preprocessor - Include directories 104
Figura 39 – Assembler Preprocessor - Include directories 104
Figura 40 – Parametros do Debugger 105
Figura 41 – Arquivos fonte do FreeRTOS 105
Figura 42 – Arquivos fonte do FreeRTOS+UDP 105

LISTA DE TABELAS

Tabela 1 – Tamanho do código (exemplo com GCC para ARM Cortex-M) 23
Tabela 2 – Implementações RMW suportadas 40
Tabela 3 – Quantidade máxima de mensagens enviadas pela STMDISCOVERY para o primeiro teste de desempenho 75
Tabela 4 – Quantidade máxima de mensagens enviadas pela *Stellaris Eval Board* para o primeiro teste de desempenho 76
Tabela 5 – Quantidade máxima de mensagens enviadas pela STMDISCOVERY para o terceiro teste de desempenho, utilizando apenas o FreeRTPS . . 76
Tabela 6 – Quantidade máxima de mensagens enviadas pela STMDISCOVERY para o quarto teste de desempenho, executando um processo com um publicador 77
Tabela 7 – Quantidade máxima de mensagens enviadas pela STMDISCOVERY para o quinto teste de desempenho, executando um processo com um publicador e com conexão direta ao computador 77
Tabela 8 – Consumo médio de processamento do sistema com processos de PID, publicador e assinante, todos a uma taca de 100 Hz 78
Tabela 9 – Consumo médio de processamento do sistema com processos de PID, publicador e assinante, todos a uma taca de 1000 Hz 79
Tabela 10 – Consumo médio de processamento do sistema com apenas um processo PID à uma taca de 1000 Hz 79
Tabela 11 – Consumo médio de processamento do sistema com um processo PID à 1000 Hz e 5 publicadores a uma taxa de também 1000 Hz 80

GLOSSÁRIO

API	*Application programming interface*
ARM	*Advanced RISC Machine*
CAN	*Controller Area Network*
DDS	*Data Distribution System*
EDF	*Earliest deadline first scheduling*
EDP	*Endpoint Discover Protocol*
GPL	*GNU General Public License*
GPL2	*GNU General Public License 2*
I2C	*Inter-Integrated Circuit*
IDL	*Interface Description Layer*
IEC	*International Electrotechnical Commission*
IoT	*Internet of Things*
IP	*Internet Protocol*
IPV4	*Internet Protocol Version 4*
JSON	*JavaScript Object Notation*
LAN	*Local Area Network*
LCD	*Liquid crystal display*
LGPL	*GNU Lesser General Public License*
NRTT	*Non-real-time task*
OMG	*Object Management Group*
QoS	*Quality of Service*
PDP	*Participant Discover Protocol*
PID	*Proportional Integral Derivative*
POSIX	*Portable Operating System Interface*
PWM	*Pulse Width Modulation*
RAM	*Random Access Memory*
RCL	*ROS Client Library*
RMW	*ROS Middleware*
ROM	*Read Only Memory*
ROS	*Robot Operating System*
ROS2	*Robot Operating System 2*
ROSCon	*ROS Conference*
RPC	*Remote procedure call*
RTI	*Real-Time Innovations*

RTOS	*Real Time Operating System*
RTPS	*Real Time Publish Subscribe*
RTT	*Real-time tasks*
SAIL	*Stanford Artificial Intelligency Laboratory*
SO	Sistema operacional
SPI	*Serial Peripheral Interface*
TCP	*Transmission Control Protocol*
TUV	*Technischer Überwachungsverein*
UART	*Universal asynchronous receiver transmitter*
UDP	*User Datagram Protocol*
USB	*Universal Serial Bus*
XML	*eXtensible Markup Language*

1 INTRODUÇÃO

Em certos tipos de processos robóticos deseja-se que o sistema seja responsivo. Em aplicações de missão crítica, um atraso de menos de um milésimo de segundo no sistema pode causar uma falha catastrófica (KAY, 2017). Segundo ainda (KAY, 2017), softwares de tempo real são aqueles que garantem uma computação correta no tempo determinado, ou seja, é necessário que o sistema possua determinismo para que o dado seja entregue no tempo correto.

Como exemplo de aplicação utilizado tempo real, temos sistemas de aquisição de dados. Apesar da amostragem do dado respeitar os requisitos de tempo real, o envio dos dados de um ponto para outro depende de diversos outros fatores, como: velocidade da camada física de comunicação, número de dispositivos conectados, qualidade do serviço, e a quantidade de dados trafegando de um ponto a outro.

Outras aplicações para sistemas de tempo real, e estão entre as principais, é controle digital de sistemas e filtros digitais. O controle de sistemas é um método utilizado em diversos sistemas como, por exemplo, controle de posição, velocidade, vazão e outros. Um sistema de controle atua em um processo de forma a fornecer a saída que o operador deseja obter. O sistema a ser controlado é, em geral, chamado de processo ou planta e pode ser modelado e representado matematicamente por uma função chamada função de transferência (PHILIPS, 1995).

Para se obter tempo real em sistemas embarcados, utilizam-se sistemas operacionais de tempo real embarcado (RTOS, *Real Time Operating System*). Esses fornecem um kernel de tempo real, que oferecem recursos para se garantir o determinismo do sistema, pouco consumo de memória e baixa latência de troca de contexto. É possível encontrar aplicativos em que o RTOS deve executar, não apenas um conjunto de tarefas em tempo real, RTTs (*Real-time tasks*), mas também um conjunto de tarefas não tempo real, NRTT (*Non-real-time task*) (PÁEZ et al., 2015). Esses tipos de sistemas que lidam com um conjunto heterogêneo de tarefas, que envolvem RTTs e NRTTs, são denominados Sistemas Críticos Mistos (VESTAL, 2007). O FreeRTOS é um SO (sistema operacional) de tempo real embarcado, ele oferece um escalonador de processos que garante a preferência para os que possuem maior prioridade, os mais críticos, e através desse recurso tem-se um mecanismo necessário, mas não suficiente para se programar sistemas de tempo real.

Com o interesse de integrar os sistemas embarcados em aplicações ROS (*Robot Operating System*), diversas metodologias para troca de mensagens foram desenvolvidas. O **rosserial** e **rosbridge** são dois exemplos desses sistemas, eles efetuam a comunicação para outros nós através de um nó ponte, esse responsável por converter a mensagem

de um protocolo especifico para o modelo de comunicação do ROS (ROSSERIAL, 2017; ROSBRIDGE_SUITE, 2017), e de forma recíproca do protocolo ROS para um específico. Já o **uROSnode** e o **rosc** implementam o ROSTCP, podendo assim comunicar diretamente com o Master do sistema e outros nós(UROSNODE. . . , 2017; ROSC, 2017). Com a alteração do middleware de comunicação na segunda versão do ROS para DDS (*Data Distribution System*), deu-se inicio ao desenvolvimento do FreeRTPS, uma implementação RTPS (*Real Time Publish Subscribe*) compacta para dispositivos embarcados que possuem memória RAM (*Random Access Memory*) e ROM (*Read Only Memory*) limitadas (FREERTPS, 2017). O intuito desse sistema é integrar sistemas embarcados ao mundo ROS, permitindo utilizar dispositivos de propósito específico com processamento menor que os oferecidos por computadores de mesa, porém, com menor consumo de energia e custo reduzido em comparação a dispositivos de propósito geral.

1.1 Objetivos

O principal objetivo desse trabalho é obter uma arquitetura capaz de se comunicar com outros processos ROS2 (*Robot Operating System 2*) em uma rede Ethernet, enquanto consegue oferecer os recursos para se garantir processamento interno em tempo real. A capacidade de processamento do sistema para tempo real será dada em função das características intrínsecas ao hardware que estiver embarcando o sistema.

Uma parte da estrutura final será composta pelo sistema em desenvolvimento, FreeRTPS, que implementa o necessário para envio e recebimento de mensagens da camada de comunicação DDS com o ROS2. Outra parte do sistema será composta pelo FreeRTOS, um sistema embarcado de tempo real para dispositivos embarcados, que fornece o necessário para executar aplicações com restrições de tempo bem definidas, o kernel de tempo real. Por isso nomeamos o sistema de FreeRTOS+FreeRTPS.

Com a integração desses sistemas, espera-se obter uma implementação que seja capaz de:

a) respeitar restrições de tempo;

b) efetuar troca de mensagens com outros nós distribuídos ROS2;

c) possuir uma forma intuitiva de ser utilizado;

d) ser facilmente portado para diversas arquiteturas de sistemas embarcados;

e) adicionar atributos ao FreeRTPS, como recursos ao protocolo UDP(*User Datagram Protocol*)/IP(*Internet Protocol*) e escalonamento de processos.

1.2 Motivação

Sistemas embarcados e tempo real estão dentro dos interesses da comunidade de robótica e também da comunidade ROS, como se pode observar no trabalho apresentado na ROSCon (*ROS Conference*) 2013, *"Bridging ROS to Embedded Systems: A Survey"* por Morgan Quigley (QUIGLEY, 2017a), e os trabalhos apresentados na ROSCon de 2015, *"ROS 2 on 'small' embedded systems"* por Morgan Quigley (QUIGLEY, 2017b) e "Real-time Performance in ROS 2" por Jackie Kay e Adolfo Tsouroukdissian (KAY; TSOUROUKDISSIAN, 2017).

Dentro da área da robótica, o controle de sistemas é um dos recursos mais utilizados. Porém, para se obter um controle estável, no mundo digital, é necessário que o sistema possua determinismo na computação do valor de saída do processo. Uma das formas de garantir as restrições de tempo impostas por esses controlares é através do uso de sistemas operacionais de tempo real. Os sistemas operacionais em tempo real oferecem políticas de agendamento que produzem certo comportamento da aplicação (PÁEZ et al., 2015), o que não garante, mas permite implementar processamento de tempo real.

Atualmente, a utilização de sistemas embarcados nos mais variados dispositivos tem atraído a atenção dos mais diversos públicos. A atual facilidade de implementação, disponibilidade de bibliotecas, aumento dos recursos e periféricos oferecidos, baixo custo entre outros itens que tem simplificado a utilização desses sistemas, fazem com que os mais diversos tipos de pessoas venham a desenvolver soluções dos mais variados tipos e fazendo esse mercado crescer cada vez mais.

Com o enfoque de integrar o ROS a sistemas embarcados, foram desenvolvidas várias formas para se utilizar os dois sistemas em conjunto. Uma delas é o FreeRTPS, esse sistema oferece uma comunicação direta com outros nós ROS na rede, estando eles em computadores ou em outros dispositivos embarcados. Apesar de o FreeRPTS ser uma implementação minimalista, dependendo da complexidade das tarefas atribuídas ao microcontrolador ou da forma como essas tarefas foram elaboradas, por exemplo, muitas interrupções de hardware com código mal implementado, podem fazer com que o sistema não atinja as restrições de tempo real.

Difundir sistemas embarcados junto ao recurso de tempo real ao ROS2 é uma das maneiras de aumentar a gama de possibilidades no desenvolvimento de aplicações no mundo da robótica, e também oferecer a aplicações não-robóticas em sistemas embarcados os recursos do ROS. Buscando atingir os requisitos de tempo real e complementar alguns recursos nessa versão minimalista de um RTPS, Iniciou-se o projeto de integração do FreeRTPS ao sistema operacional embarcado FreeRTOS. Como resultado dessa integração, espera-se obter um sistema que possa contribuir em pesquisas e desenvolvimento de aplicações da comunidade que utiliza ROS.

1.3 Contribuições

Com o sistema final será possível oferecer, aos desenvolvedores de aplicações em sistemas embarcados, uma ferramenta capaz de se comunicar com processos ROS2 e disponibilizar mecanismos para manter seus processos críticos respeitando requisitos de tempo. Isso fornecerá a esses desenvolvedores uma gama de possibilidades, tendo em vista que o ROS é uma ferramenta utilizada por uma enorme quantidade de pessoas e empresas ao redor do mundo. Será ainda possível integrar o ROS em uma área que vem se desenvolvendo nos últimos tempos, a Internet das Coisas (IoT, *Internet of Things*), que possui as mais variadas aplicações implementadas em sistemas embarcados e distribuídos. Com o recurso de tempo de real, será ainda possível distribuir processos de controle de atuadores em pequenos nós de processamento, de baixo consumo de energia e com menor custo, oferecendo uma alternativa aos sistemas maiores que fazem uso de um computador central, de elevado processamento, para cuidar de todos os processos.

1.4 Estrutura da dissertação

Este documento se divide em 5 capítulos. O capitulo 2 possui o referencial teórico do trabalho e também alguns sistemas desenvolvidos que almejam o mesmo resultado que o trabalho presente. O capitulo 3 apresenta o desenvolvimento do trabalho, essa seção oferece as informações obtidas do sistema FreeRTPS para que fosse possível a implementação da arquitetura proposta, bem como todas as modificações que foram elaboradas para se obter tal arquitetura. O capitulo 4 oferece uma análise dos resultados através de testes de conceito. O capítulo 5 traz as conclusões obtidas do trabalho e também as limitações do sistema. Por fim, o apêndice A apresenta códigos e formas de utilizar o sistema.

2 REFERENCIAL TEÓRICO

Para se obter os conceitos bases para o desenvolvimento desse trabalho, buscou-se efetuar uma pesquisa dos temas que fornecem informações técnicas e teóricas dos componentes utilizados.

Dentre os assuntos temos o conceito dos sistemas embarcados, necessidade e recursos de sistemas operacionais, bem como os sistemas operacionais embarcados, protocolos de comunicação utilizados e o framework ROS2.

2.1 Sistemas embarcados

"Os sistemas embarcados são sistemas microprocessados projetados para um propósito ou aplicação especifica, possuindo, em geral, poucos recursos de memória e processamento limitado. Na maioria dos casos, são sistemas projetados para aplicações que não necessitem de intervenção humana." (ALMEIDA, 2013, p.4).

Tem-se como características de dispositivos embarcados a incorporação de periféricos dentro de um único chip. Por exemplo, memória RAM e ROM, onde sua programação está embarcada. Um sistema embarcado pode ou não ser utilizado em *standalone*, isso é, executar uma tarefa sem a dependência de outro dispositivo ou sistema (LI; YAO, 2003).

Um dos motivos que fazem os dispositivos embarcados serem conhecidos como computadores de propósito especifico, é pelo fato de que na maioria dos casos eles são programados para realizar apenas uma função dedicada, isto é, o usuário final não pode alterar a sua lógica de operação. O usuário pode apenas configurar a maneira como o sistema se comporta, porém não pode adicionar ou remover funções do sistema (MARWEDEL, 2010; STUDNIA et al., 2013; KERMANI et al., 2013).

O incentivo da grande demanda do uso de dispositivos embarcados nos dias atuais se deve a necessidade da portabilidade e redução do tamanho de sistemas e de consumo de energia (SANTOS, 2017). Um assunto comum de ouvir atualmente é a sustentabilidade, sistemas embarcados oferecem o desenvolvimento de aplicações menores, que envolvem a utilização de menos material. Já o baixo consumo de energia envolve não só economia, mas como também maior autonomia, itens importantes em aplicações no ramo da robótica.

Logo, observa-se então que a seleção de um dispositivo embarcado está em função dos requisitos de sua tarefa. Chips com pouca memória ou processamento são mais baratos e indicados para aplicações que não necessitam de velocidade, como por exemplo, um relógio digital de parede.

O microcontrolador que será utilizada nesse no desenvolvimento desse trabalho é o STM32F407VGT6, da empresa ST. Ele possui um núcleo ARM M4 de 32 bits, 1 Mbyte de memória flash, 192 Kbyte de RAM, com um periférico de Ethernet e muitos outros. Para evitar a necessidade do desenvolvimento do circuito, será utilizada uma placa de desenvolvimento produzida pela própria ST (STM32F4DISCOVERY, 2017). Essa placa já fornece todos os componentes necessários para operação e programação do microcontrolador, além do conector Ethernet para efetuar a conexão em uma rede de computadores.

2.1.1 Sistemas operacionais

Sistemas operacionais são softwares que fornecem uma camada de abstração entre o usuário para o hardware em um sistema (WULF et al., 1974). Um SO provê a interface e funcionalidades para que os usuários possam utilizar o sistema de forma eficiente e transparente para com o hardware. Permite também mover uma aplicação de um sistema para outro de forma que implementação sofra pouca ou nenhuma alteração, tornando o código não dependente do hardware (HEATH, 2002). "Esta é uma característica muito desejada em sistemas embarcados, onde existe uma pluralidade nos tipos de periféricos dificultando a reutilização de código" (ALMEIDA, 2013, p.4)

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

a) manusear a memória disponível e coordenar o acesso dos processos a ela;

b) gerenciar e coordenar a execução dos processos através de algum critério;

c) intermediar a comunicação entre os periféricos de hardware e os processos.

Esses itens se relacionam com os três recursos fundamentais de um computador: processador, memória e periféricos de entrada e saída, 1.

O núcleo, ou Kernel, é software que contém os componentes centrais de um sistema operacional, tais como: escalonador de processos, gerenciamento de memória, gerenciamento de entrada e saídas gerenciamento do sistema de arquivos (DEITEL; DEITEL; CHOFFNES, 2005). Dessa forma, o kernel efetua a interface entre os códigos de acesso ao hardware, drivers, e as ferramentas para fazer uso do hardware de forma transparente (ALMEIDA, 2013). O sistema geralmente não fornece total acesso ao hardware por questões de segurança.

Em sistemas operacionais, as tarefas a serem executadas pelo processador são organizadas em programas, que são seqüências de código organizadas para executar uma tarefa especifica. A partir do momento que um programa é colocado para execução ele passa a ser definido como processo (STALLINGS, 2009).

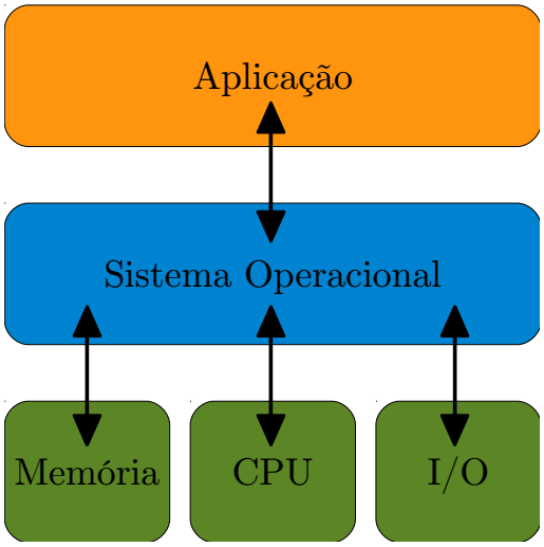

Figura 1 – Interfaceamento realizado pelo sistema operacional

Fonte: (ALMEIDA, 2013)

Para que o kernel possa gerenciar as tarefas, é necessário que o processo possua informações básicas para que o kernel possa executar e gerenciar as trocas de contexto, bem como os dados utilizados por esse programa. As informações mínimas necessárias são:

a) o código a ser executado;

b) variáveis internas do processo;

c) ponteiros do estado anterior para memória de dados e código.

Apesar de nos dias atuais os processadores possuírem mais de um núcleo de processamento, em muitos casos a quantidade de processos executando ainda é maior que a quantidade de núcleos. Perceber então, com essa restrição física, que só é possível executar um processo a cada momento em processadores com um núcleo, e X processos simultaneamente em processadores com X núcleos. Como existem processos que podem ficar em execução durante todo o período que o sistema estiver operando (ex. kernel do SO) e outros apenas enquanto o usuário desejar (ex. aplicações do usuário), os sistemas operacionais utilizam o mecanismo conhecido como multitasking para aparentar que os processos estejam executando concorrentemente (MULTITASKING, 2017). Esse método efetua uma rápida comutação dos processos, executando uma porção do processo por vez, assim como pode ser observado na figura 2.

A comutação dos processos no núcleo é chamada troca de contexto, ela é responsável por armazenar todos os dados do processo que está sendo atualmente executado e também inserir os dados do próximo processo no núcleo.

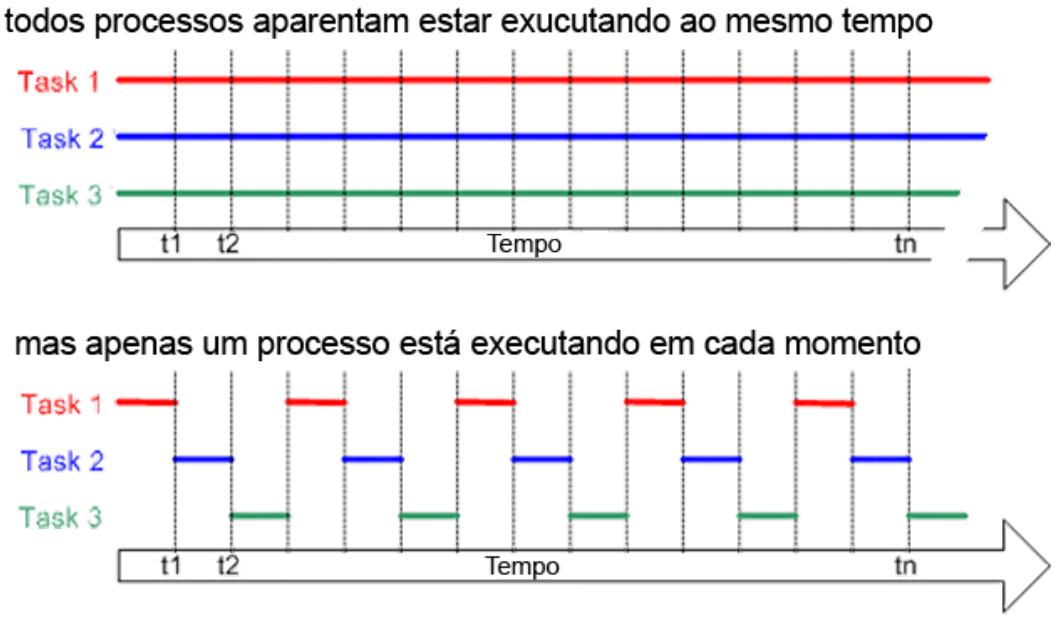

Figura 2 – Multitasking

Fonte: (MULTITASKING, 2017)

As aplicações que são executadas no sistema operacional são implementadas como processos. Desta forma, as aplicações são gerenciadas pelo kernel através de seu componente escalonador de processos, que define a sequência de execução dos processos em função do algoritmo de escalonamento definido. Existem vários tipo de escalonadores e cada um com sua política de seleção de processos, possuindo assim um melhor desempenho para um determinado tipo de ambiente. A figura 3 apresenta um exemplo de processos escalonados por fatia de tempo durante a execução do sistema.

Como descrito por (DEITEL; DEITEL; CHOFFNES, 2005), apresenta-se a seguir alguns exemplos de escalonadores e suas políticas:

a) first in first out ou First come first served: Um dos algoritmos mais simples. Os processos são executados conforme o momento que chegaram à fila de processos. Ele é não preemptivo, ou seja, uma vez no processador o processo executa ate o fim.

b) round-Robin: Os processos são executes na mesma ordem que o first in first out, porem com a diferença de ser preemptivo. Isto é, possuem um intervalo de tempo (quantum) de execução. A cada intervalo de tempo o processo é pausado e volta para o fim da fila e só é executado novamente na próxima parcela de tempo do processador.

c) shortest Process First: O processo que será inserido para execução no processador é aquele que possui menor tempo execução estimado. É um algoritmo não preemptivo.

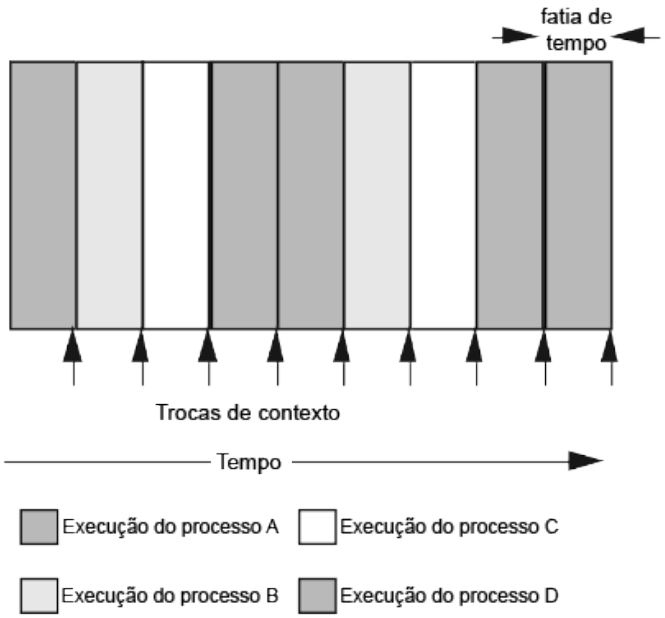

Figura 3 – Escalonamento de processos por fatia de tempo

Fonte: (DEITEL; DEITEL; CHOFFNES, 2005)

d) highest Response Ratio next: Esse método corrige algumas deficiências do shortest process first, como garantir que um processo com tempo de execução elevado seja executado quando estiver sempre na presença de processos mais curtos. A prioridade de um processo longo aumenta em função do tempo que ele esta esperando na fila.

e) shortest Remainig Time: É a versão preemptiva do shortest process first. Ele executa, por um período de tempo, o processo com menor tempo de execução restante estimado.

Os algoritmos de escalonamento de processos para tempo real procuram atender as necessidades de restrição de tempo de um processo. Através de políticas de prioridade o escalonador define qual processo deve executar, fazendo com que o processo seja executado no tempo correto.

2.1.2 Sistemas operacionais embarcados

Nem todas aplicações necessitam de um sistema operacional, elas são executadas diretamente sobre o hardware (HEATH, 2002). Os dispositivos embarcados se enquadram nesse caso, a maioria dos projetos embarcados não faz uso desse recurso por motivos como: aumento da complexidade do sistema ou aumento no consumo de memória. Contudo, a utilização de sistemas operacionais em aplicações mais complexas pode simplificar drasticamente a fase de projeto, através dos recursos oferecidos pelo mesmo.

Apesar dos sistemas operacionais de computadores possuírem elevado tamanho, existem sistemas operacionais que foram desenvolvidos especialmente para dispositivos embarcados. Esses sistemas possuem geralmente baixos consumo de processamento e baixo consumo de memória, uma vez que esses recursos que são escassos em muitos microcontroladores.

A relação entre sistemas embarcados e sistemas de tempo real pode ser representada da seguinte forma: nem todo sistema embarcado possui comportamento de tempo real e nem todo sistema de tempo real pode ser embarcado. Porém, existe uma zona de intersecção entre os dois sistemas, nessa região estão os sistemas conhecidos como sistemas embarcados de tempo real, figura 4, (LI; YAO, 2003).

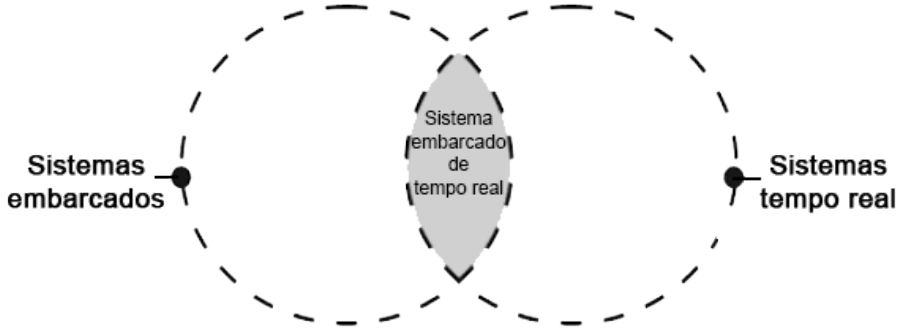

Figura 4 – Sistemas embarcados de tempo real

Fonte: (LI; YAO, 2003)

Os sistemas operacionais mais utilizados nessa categoria possuem ainda a capacidade de tempo real. Alguns exemplos de sistemas operacionais embarcados:

a) Nuttx:

b) ChibiOS:

c) Riot:

d) Freertos.

Os sistemas operacionais de tempo real (RTOS) possuem a capacidade de atender a eventos com um atraso bem pequeno. Essas características são referenciadas como determinismo e responsividade (GALLMEISTER, 1995). Essas qualidades são pontos críticos principalmente para sistemas hard real time (SANTOS, 2017).

Para execução deste trabalho, em função da vasta utilização e suporte a diferentes arquiteturas, escolheu-se o FreeRTOS como sistema operacional. Com ele será possível adicionar o tempo real almejado por este trabalho em sistemas embarcados mais simples, desde que possuam quantidade de memória e processamento necessário para embarcar o sistema.

O FreeRTOS trabalha sobre uma licença GPL (*GNU General Public License*) modificada. Isso permite que o seu Kernel possa ser utilizado integralmente, porém, quaisquer modificações efetuadas nos arquivos originais não podem ser de código fechado e devem fazer referencia ao site de distribuição (SANTOS, 2017). A cláusula modificada da licença permite que o se o sistema for utilizado junto a código proprietário, o código linkado ao sistema não necessita de ser aberto (LICENSE. . . , 2017). Além disso, o FreeRTOS oferece recursos como o SafeRTOS, um certificado TUV (*Technischer Überwachungsverein*) RTOS para sistemas críticos (versão paga), um kernel com tamanho entre 6Kbytes e 12Kbytes e é gratuito para ser utilizado em aplicações comercias (FREERTOS. . . , 2017).

Outro ponto extremamente relevante para escolha do FreeRTOS é a disponibilidade de uma pilha UDP/IP totalmente integrada com o sistema através de processos, o FreeRTOS+UDP, também projetada e oferecida pelos desenvolvedores do FreeRTOS.

O FreeRTOS+UDP é um pequena e eficiente versão da pilha UDP/IP (IPV4, *Internet Protocol Version 4*), que foi desenvolvido para ser utilizado juntamente com o FreeRTOS, em sistemas embarcados. Ele é totalmente compatível com os processos do sistema operacional de tempo-real embarcado e é baseado no conceito de sockets (FREERTOS+UDP, 2017). O tamanho do sistema pode variar em função dos recursos habilitados, como pode ser visto na tabela 1.

Característica	Com otimização -O1	Com otimização -Os
Pilha UDP/IP básica	3.6K	2.9K
... com processamento de requisição de ping	3.7K	3.3K
... com saída e processamento de respostas de pings	4K	3.2K
... com fragmentação de pacotes de saída	4K	3.2K
... com auto configuração por DHCP	5.5K	4.2K
... com DNShabilitado	4.4K	3.5K
... com todas funções habilitadas	6.7K	5.1K

Tabela 1 – Tamanho do código (exemplo com GCC para ARM Cortex-M)

Fonte: freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_UDP/FreeRTOS_Plus_UDP.shtml

O FreeRTOS+UDP utiliza um modelo de licença dual, que permite que o software seja utilizado sob o padrão de código aberto GPL ou licença comercial. Ele é gratuito é pode ser utilizado, modificado, avaliado e distribuído sem custo, desde que, o usuário respeite a os termos GPL2 (*GNU General Public License 2*) e não remova o aviso de direito autoral.

Empresas ou indivíduos que por motivos comerciais ou outros não podem cumprir Com os termos da licença GPL2, devem obter uma licença comercial Incorporando FreeRTOS+UDP em software proprietário para distribuição.

Quando o sistema operacional embarcado FreeRTOS não está executando nenhum processo, o sistema entra em estado ocioso e executa o processo interno "idle", que possui a menor prioridade, ou seja, o sistema só entra em estado ocioso caso todos os processos do sistema estejam parados. Um processo entra no estado 'paused' quando funções de delay ou que aguardam um dado sem tempo limite são invocadas.

A função 'vTaskDelayUntil' é um exemplo de delay utilizado no sistema, utiliza-se esse recurso para garantir processamento periódico de um processo. Como o processo não irá efetuar nenhuma ação durante o período de espera, o sistema coloca o processo atual no estado 'paused'. Enquanto o processo está no estado de 'paused' ela não entra na lista do escalonador de processos, já após o tempo de espera acabar o processo volta para o estado ativo.

No momento em que nenhum processo está sendo executado pelo escalonador de processos, o processo interno 'Idle' passa a ser executada. O FreeRTOS possui a função vApplicationIdleHook para que o usuário possa executar algum código enquanto em modo ocioso ou até mesmo verificar por quanto tempo o sistema permanece nesse estado.

2.1.3 Tempo real

Sistemas de tempo real possuem a característica de que um resultado correto depende tambem de atender requisitos temporais, diferenciando-se de sistemas não tempo real (sistemas convencionais). "Em um sistema convencional a resposta é considerada correta se tiver os valores corretos. Já em um sistema de tempo real a resposta só é considerada correta se, além dos valores corretos, ocorrer dentro do intervalo de tempo correto"(LAGES, 2014, p.33).

Diversas vezes, sistemas de tempo real são referenciados como sistemas de alto processamento, porém, tal sistema não é necessariamente de tempo real a não ser que entregue um resultado com as restrições de tempo impostas pelo processo. Essa forma de interpretação vem sido utilizada principalmente por pessoas que não possuem conhecimento na área e acabam popularizando um significado errôneo para o termo (KAY, 2017).

Como exemplos das necessidades de processamento em tempo real têm-se sistemas de controle digital e filtro digital, que necessitam de restrições de tempo bem definidas para oferecer uma saída estável. Um projeto de um sistema de tempo real deve ser bem elaborado para evitar custos desnecessários. Pode-se citar o caso de um sistema de controle de temperatura, que geralmente possui uma resposta lenta em sistemas com massa térmica elevada. Nesse caso não é necessário um hardware caro com elevado processamento, já que a saída do processo pode ser mais lenta que o controle de um pêndulo invertido, por exemplo.

Com o crescente uso do termo tempo real para designar sistemas que não são, a rigor, sistemas de tempo real, criou-se uma classificação para diferenciá-los (LAGES, 2014; KAY, 2017; SANTOS, 2017), tem-se então:

a) hard real time: sistemas de tempo real no sentido estrito, ou seja, a correção da resposta está associada a especificação de temporização que precisa ser cumprida durante toda a operação. Uma única violação da restrição temporal significa falha do sistema;

b) firm real time: sistemas nos quais a violação das especificações temporais, embora tolerada, implica uma degradação do desempenho do sistema;

c) soft real time: sistemas nos quais podem ocorrer violações esporádicas das especificações temporais, no entanto, na maioria das vezes as especificações são cumpridas. De certa forma é como se a especificação temporal fosse encarada como uma media a ser mantida, desde que a media seja mantida não há degradação no desempenho do sistema.

Um controlador PID (*Proportional Integral Derivative*), como será descrito na seção 2.1.4, é um exemplo de sistema que necessita ser hard real time. Se as restrições de tempo impostas não forem atendidas, pode-se levar o sistema à instabilidade, o que pode ser catastrófico em algumas aplicações.

Já como exemplo de um sistema soft real time, tem-se a reprodução de um vídeo ou áudio. Nesse caso, a violação esporádica do tempo de resposta é muitas vezes imperceptível para os sentidos humanos.

Pode-se ter também sistema que executam mais de um processo, e cada processo pode ter seu requisito como um tipo de real time. Um dos casos onde isso ocorre, são em sistemas de aquisição de dados com display. Nesse sistema é necessário que um processo atue na aquisição e armazenamento dos dados de forma uniforme e com tempos bem definidos, principalmente quando o dado em questão está entrelaçado com um filtro digital. Já no processo de apresentação dos dados para o usuário, pode ser utilizado um soft real time, uma vez que, o usuário não é capaz de analisar dados sendo apresentados em milésimos de segundo e pequenos atrasos não serão demasiadamente inconvenientes.

2.1.4 Controle digital de sistemas

Um sistema de controle pode ser definido como um conjunto de componentes que altera o comportamento de um sistema para que ele produza uma resposta desejada. Define-se como planta ou processo o sistema que deseja-se controlar, esse sistema pode ser modelado e representado matematicamente por uma função, a função de transferência (PHILIPS, 1995). A função de transferência é definida como a razão entre a transformada de Laplace da saída pela transformada de Laplace da entrada do sistema. Através de

comandos, como mecânicos ou elétricos, ajustam-se as saídas dos atuadores obtendo-se assim o valor desejado.

Sistemas de controle que não utilizam o estado atual da saída para ajustar o atuador são conhecidos como controle de malha aberta. A partir do momento que se utiliza realimentação para ajustar a saída do atuador, o sistema recebe o nome de controle de malha fechada (OGATA, 2011). Observa-se então que o controle de malha fechada pode oferecer um comportamento mais efetivo do que o de malha aberta, principalmente em sistemas susceptíveis a distúrbio, pois sabendo o estado atual do sistema o controlador pode ajustar o atuador de forma a compensar tais variações.

Uma das ferramentas utilizadas para atuar em sistemas de controle digitais é o PWM (*Pulse Width Modulation*). O PWM produz basicamente uma forma de onda digital. Com apenas alguns componentes eletrônicos, ela pode ser usada como um conversor digital analógico barato. A conversão da forma de onda do PWM para sinais analógicos envolve o uso de filtros passa-baixo. Em um PWM típico, a frequência base do sinal é fixada e apenas a largura do pulso é variada. A largura do pulso é conhecida como *duty_cycle*, variando o *duty_cycle* de 0% a 100% do período do sinal PWM, é possível obter um sinal analógico correspondente de 0V até a tensão de alimentação do sinal, (USING. . . , 2017). A figura 5 apresenta um sinal PWM e seu equivalente analógico após aplicado a um filtro passa-baixa.

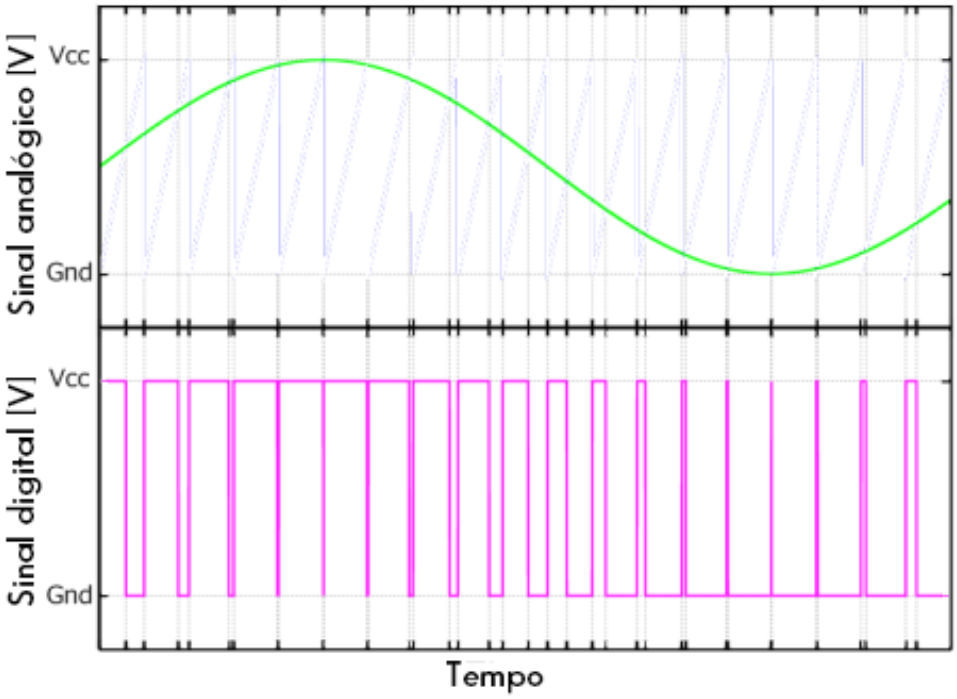

Figura 5 – Sinal PWM e seu correspondente analógico

Fonte: (PULSE-WIDTH. . . , 2017)

Controlador PID digital

O controle PID utiliza a diferença entre o valor atual desejado e o valor mensurado, da saída do sistema, para ajustar o atuador referente a essa saída e assim alcançar o valor almejado. Esse tipo de controle é bastante utilizado por possibilitar baixos erros de offset. Ele combina os modos de controle proporcional (P), integral (I) e derivativo (D) (LEAL, 2012).

A parcela proporcional ajuda a minimizar o erro da variável controlada, mas durante regime transitório pode gerar valores de pico acima do desejado (*overshoot*). A componente derivativa ajuda a reduzir o *overshoot*, mas pode acrescentar um *offset* de erro quando o sistema chega ao regime permanente. Já a componente integral adiciona um pólo na origem, eliminando o erro em regime permanente (NISE, 2009).

A função de transferência do controlador PID é representada por:

$$G_c(s) = K_p + K_d s + \frac{K_i}{s} \tag{2.1}$$

Onde K_p, K_d e K_i representam as constantes das parcelas proporcional, derivativa e integral, respectivamente. Esses valores são ajustados, em função do sistema controlado, para se obter baixos valores de overshoot, erro e tempo de resposta.

Porém, a equação anterior não pode ser utilizada para o controle digital de processos, uma vez que no mundo digital os sinais são analisados de forma discreta. Para se empregar a equação anterior em sistemas discretos utiliza-se a transformada Z, essa transformada desempenha em sistemas discretos (amostrados) o mesmo que a transformada de Laplace em sistemas contínuos.

Existem alguns métodos matemáticos para conversão de uma função em 's', Laplace, para 'Z', tais como: Pólos e zeros casados, aproximação Zero-Order-Hold, Impulse Invariant. Um dos métodos mais simples e que fornece um modelo discreto aproximado do modelo contínuo é a transformação bilinear, ou Transformação de Tustin. A transformação bilinear é um método matemático que permite levar equações que representam sistemas dinâmicos analógicos para o mundo digital, de modo que o sistema passa a ser analisado em intervalos constantes de tempo T (tempo de amostragem)(FADALI; VISIOLI, 2009). Nesse ponto que se faz necessário o uso de um sistema de tempo real, desde que é necessário respeitar o período T para executar o processamento correto.

Obtêm-se a conversão pelo método de Tustin através da substituição de 's' pelo seu equivalente em 'Z'. A equação a seguir apresenta a aproximação da função equivalente para esse método.

$$s \equiv \frac{2}{T}\left[\frac{z-1}{z+1}\right] \tag{2.2}$$

Aplicando a equação equivalente anterior, 2.2, na função de transferência de um controlador PID em 's' (tempo contínuo), 2.1, e em seguida aplicando a transformada Z inversa para o domínio do tempo discreto (LEAL, 2012), obtêm-se a seguinte equação.

$$\begin{aligned} U(n) = U(n-2) + k_p * [e(n) - e(n-2)] + \\ k_d \frac{2}{T}[e(n) - 2 * e(n-1) + e(n-2)] + \\ k_i \frac{T}{2}[e(n) + 2 * e(n-1) + e(n-2)] \end{aligned} \tag{2.3}$$

Nesse trabalho, utiliza-se a equação obtida acima para efetuar a validação dos requisitos de tempo real do sistema proposto.

2.2 Redes de computadores

Em um nível mais baixo, toda comunicação envolve a transmissão de dados em forma de energia através de um meio. Em sistemas eletrônicos, algumas das formas de transmissão utilizadas são dadas através de: cabos elétricos, utilizando a tensão, ou através do ar, utilizando ondas de radio. Nos meios eletrônicos, o hardware é o responsável por codificar e decodificar esses sinais elétricos em informações que são manipuladas em software (COMER, 2004).

Redes de computadores têm crescido rapidamente. Algumas décadas atrás poucas pessoas tinham acesso a uma rede. Agora, a comunicação entre computadores tem se tornado uma parte essencial de nossa estrutura. O continuo crescimento global da internet é um dos fenômenos mais interessantes em redes (COMER, 2004). Hoje em dia utiliza-se esse sistema em uma variedade de aplicações, como: comunicação interna em empresas, escolas, plantas industriais entre outros. Uma das principais utilidades no passado e até mesmo nos dias atuais é o foco no compartilhamento de recursos, algumas das antigas redes de computadores foram construídas com o intuito de estender sistemas já instalados. Podemos citar como um exemplo mais simples, a necessidade de compartilhamento de dispositivos, como impressoras.

"Tecnologias LAN (*Local Area Network*) tem se tornado a mais popular forma de redes de computadores. LANs conectam mais computadores do que qualquer outro tipo de rede" (COMER, 2004, p.104). Cada LAN consiste de um meio compartilhado nos quais vários dispositivos são conectados. Os dispositivos usam turnos para enviar pacotes através desse meio, buscando evitar colisões de mensagens enviadas por dois dispositivos ao mesmo tempo.

Uma LAN é uma rede de comunicação digital que fornece comunicação a uma variedade de dispositivos dentro de uma área geográfica delimitada. Décadas atrás, uma série de protocolos LAN fundamentalmente diferentes, como Ethernet e Token Ring, foram bem sucedidos no mercado de venda. A tecnologia madura e o baixo custo dos componentes devido à produção em massa tornaram a Ethernet um candidato atraente para aplicações na indústria como alternativa/complemento aos sistemas tradicionais (ZHANG; WANG, 2010).

Com o aumento da quantidade de dados trafegando em vários sistemas, muitos trabalhos vêm sendo elaborados com o intuito de aprimorar o desempenho da troca de mensagem em redes Ethernet. Esses trabalhos têm geralmente como objetivo atender requisitos de tempo real, cada vez mais rígidos, nesse tipo de comunicação.

De acordo com (SCHIMMEL; ZOITL, 2010), atualmente, mais e mais unidades de sensores estão cooperando junto com atuadores a fim de realizar tarefas de controle e tarefas de mensuração em sistemas de automação. No futuro, essa tendência acelerará em domínios que vão desde a automação industrial, cooperação de robôs até sistemas de logísticas.

No trabalho realizado por (SCHIMMEL; ZOITL, 2010), o autor descreve maneiras para melhorar a performance em uma rede, padrão IEC (*International Electrotechnical Commission*) 61499, através de um switch de Ethernet em um sistema fechado (toda troca de dados é interna, não há comunicação com nós externos ao sistema). Em outros trabalhos, como em (HOANG et al., 2001) e (YIMING; EISAKA, 2005), o método EDF (*Earliest deadline first scheduling*), que trata a mensagem com menor deadline, é utilizado. Em outros trabalhos é utilizado traffic shapping, como em (LOESER; HAERTIG, 2004) e (LOESER; WOLTER, 2004). Já no trabalho proposto por (KIM; LEE; PARK, 2013), foi elaborado um escalonador preemptivo no switch e a alteração do campo data no pacote Ethernet, assim tratando os pacotes da fila com maior prioridade na frente.

Como apresentado, dependendo das restrições de tempo, é possível obter tempo real em redes Ethernet. Porém, muitos fatores podem afetar o desempenho desse sistema como, por exemplo, quantidade de dispositivos anexados, tamanho e frequência das mensagens que trafegam por esse meio.

O sistema desenvolvido nesse trabalho efetua a comunicação com outras aplicações distribuídas através da rede Ethernet e utilizando o protocolo UDP/IP para troca de mensagens.

2.2.1 Data Distributed System

DDS provê um transporte do tipo publicador subscritor do similar ao transporte do ROS. Ele utiliza uma IDL (*Interface Description Layer*) definida pela OMG (*Object Management Group*) para definição e serialização de mensagens. O modelo de descoberta do DDS é um sistema distribuído de descoberta, isso permite que o DDS possa comunicar com outros Nós sem a necessidade de um Master, assim como acontecia no ROS1, o que torna o sistema mais flexível e tolerante a falha. Ele teve seu inicio como um grupo de companhias que tinham frameworks similares de middleware e tornaram um padrão quando clientes comuns desejaram a interoperabilidade entre os fornecedores (WOODALL, 2017).

O middleware é um software que liga duas aplicações, geralmente incompatíveis. Dessa forma é possível habilitar a comunicação entre arquiteturas diferentes (DEITEL; DEITEL; CHOFFNES, 2005).

O DDS possui uma extensa lista de sistemas que o utilizam, tais como:

a) sistemas financeiros;

b) sistemas espaciais;

c) sistemas de vôos;

d) sistemas de central de comboio.

O DDS é implementado por padrão no protocolo UDP e não depende de um meio de transporte ou hardware confiável para comunicação. Dessa forma, o DDS acaba precisando desenvolver métodos para aumentar a confiabilidade, mas em troca possui portabilidade e controle sobre do seu comportamento. Através de parâmetros de confiabilidade, QoS (*Quality of Service*), oferece flexibilidade no controle do comportamento da comunicação. Por exemplo, quando se está preocupado com a latência de um sistema de tempo real soft, pode-se definir a qualidade de modo a aumentar a confiabilidade da comunicação.

Os fornecedores populares de DDS incluem:

a) RTI (*Real-Time Innovations*);

b) PrismTech;

c) Software Twin Oaks;

d) eProsima.

Observando as vantagens de se utilizar o DDS, e ainda atribuindo o feedback que a equipe do ROS obteve de usuários ROS que já utilizaram o DDS, a equipe do ROS optou por remover a camada de comunicação que era utilizada no ROS1 por esse modelo de transporte de mensagens (WOODALL, 2017).

2.2.2 RTPS

RTPS é um protocolo publicador/assinante com comunicação confiável sobre meios de transporte de dados não confiáveis, como UDP. O RTPS é padronizado pela OMG como um protocolo de interoperabilidade para implementações DDS, um padrão amplamente utilizado nos setor aeroespacial e de defesa para aplicações em tempo real (RTPS..., 2017).

O RTPS vem sendo utilizado como middleware em diversas aplicações. Por exemplo, no trabalho de (ALMADANI, 2005), o autor realiza um experimento para avaliar se o RTPS atende aos requisitos de um sistema de visão industrial distribuído e em tempo real. No trabalho um programa em Ava é elaborado utilizando como protocolo de comunicação o RTPS. No teste 4414 imagens, de 10 KB, são enviadas para outro ponto a uma taxa de 35 HZ e são replicadas de volta para a fonte. De acordo com o autor, o desempenho obtido atende os requisitos de tempo Firm e Soft nesse tipo de sistema, além de simplificar projeto, construção e reduzir custos de manutenção.

Já no trabalho realizado por (AL-MADANI; AL-SAEEDI; AL-ROUBAIEY, 2013), os autores utilizam o RTPS para efetuar o streaming de vídeo através de uma rede wireless. No projeto, foram utilizados os recursos de QoS e publicação e subscrição, oferecidos pelo middleware, onde se obteve uma suave degradação na qualidade do vídeo e mantendo o streaming robusto.

Dentre as vantagens do protocolo pode-se citar:

a) desempenho e qualidade de serviço, habilitando comunicações confiáveis e com best-effort para aplicações de tempo real utilizando redes IP padrão;

b) extensibilidade, permitindo a compatibilidade com versões anteriores e interoperabilidade com outros protocolos DDS;

c) conectividade plug and play, permitindo descoberta automática de outros participantes na rede e sendo possível entrar e sair da rede a qualquer momento;

d) modularidade, permitindo equilibrar os requisitos de confiabilidade e pontualidade na entrega de dados;

e) escalabilidade, permitindo o sistema crescer para uma larga rede de publicadores e subscritores;

f) fortemente tipada, prevenindo que erros de programação possam comprometer os nós remotos na rede.

O protocolo é baseado em quatro diferentes módulos que controlam a troca de informação entre diferentes aplicações DDS (DOCUMENTS. . . , 2017).

a) o módulo de estrutura define os pontos de extremidade de comunicação os mapeia para duas contrapartes DDS;

b) o módulo de mensagem define quais mensagens os pontos de extremidade podem trocar e como eles são criados;

c) o módulo de comportamento define o conjunto de operações legais e como elas afetam cada um dos pontos de extremidade;

d) o módulo de descoberta define o conjunto de nós internos dos pontos de extremidade.

Modulo de estrutura

Como o RTPS é um protocolo de fio projetado para implementar aplicações DDS, cada conceito ou entidade DDS, naturalmente, mapeia uma entidade RTPS. Todas as entidades RTPS estão associadas a um domínio, que representa um plano de comunicação separado contendo um conjunto de participantes. Cada participante pode conter vários pontos de extremidade locais de dois tipos: Writers e Readers. Esses dois pontos de extremidade trocam informações na rede RTPS enviando mensagens. Os Writers enviam as informações disponíveis aos Readers, que por sua vez, podem solicitar ou confirmar dados. Essa estrutura é mostrada na figura 6

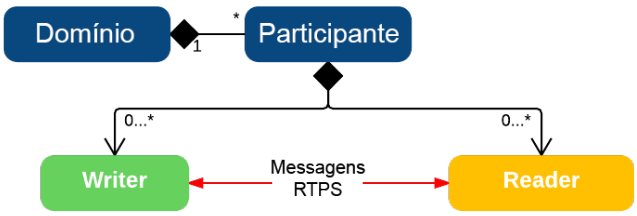

Figura 6 – Módulo do RTPS

Fonte: (RTPS. . . , 2017)

A interface entre os pontos de extremidade RTPS (Writers e Readers) e suas entidades DDS correspondentes é o HistoryCache. As informações trocadas entre os pontos de extremidade geralmente são armazenadas em um CacheChange. Por exemplo, cada operação de escrita apresenta um CacheChange no Histórico do Writer. O Writer RTPS envia uma mensagem para todos os leitores combinados. Após o recebimento, o RTPS Reader adiciona o CacheChange ao seu HistoryCache correspondente e notifica a entidade DDS de que um novo dado está disponível.

Modulo de mensagem

O módulo de mensagens define o conteúdo das trocas de informações entre os Writers e Readers. As mensagens são compostas por um cabeçalho seguido de uma série de submensagens. O Cabeçalho identifica a mensagem como parte do protocolo RTPS, bem como a versão do protocolo que está sendo usado e o fornecedor da implementação utilizada para enviar a mensagem. Ele também identifica o participante da rede que está enviando a mensagem.

Cada submensagem é composta por um cabeçalho e uma série de elementos da submensagem. Essa estrutura foi escolhida para permitir que o vocabulário das submensagens e a composição de cada submensagem sejam estendidos e mantendo a compatibilidade com versões anteriores. O cabeçalho da submensagem contém um ID que identifica o tipo, o comprimento e as flags da submessagem. Existem doze tipos diferentes de submensagens, as três mensagens mais importantes são:

a) DATA: Essa submensagem é enviada de um Writer para um Reader com dados;

b) HEARTBEAT: Essa submensagem é enviada de um Writer para um Reader comunicando os CacheChanges que o Writer tem disponível no momento;

c) ACKNACK: Essa submensagem é enviada de um Reader para um Writer, permitindo que Reader notifique o Writer sobre quais mudanças recebeu e quais estão faltando. Pode ser utilizado para efetuar confirmações.

Modulo de comportamento

Este módulo descreve as trocas de mensagens válidas que podem ocorrer entre um Writer e um Reader. Ele também define as mudanças do estado do Writer e Reader dependendo de cada mensagem. Um conjunto completo de regras pode ser encontrado no documento de especificação do RTPS. Essas regras são definidas para garantir a interoperabilidade entre diferentes implementações.

Modulo de descoberta

Esse módulo descreve o protocolo que permite aos participantes obter informações sobre a existência e os atributos de todos os outros participantes e pontos de extremidade no domínio. Essa troca de informações é chamada metatraffic. Uma vez que os pontos de extremidade remotos foram descobertos, os pontos de extremidade locais podem ser configurados de acordo para estabelecer a comunicação. O protocolo de descoberta é dividido em duas camadas: Protocolo de descoberta de participantes (PDP, *Participant Discover Protocol*) e protocolo de descoberta de ponto final (EDP, *Endpoint Discover Protocol*). O PDP especifica como os participantes são descobertos. Após a descoberta, os participantes trocam informações sobre seus pontos de extremidade usando o EDP.

Fornecedores diferentes podem implementar vários protocolos de descoberta, porém, para assegurar a interoperabilidade, um PDP e um EDP devem ser implementados por todos os fornecedores. Estes protocolos de descoberta são denominados "Simples"(Simple PDP e Simple EDP) e são muito adequados para a maioria das aplicações. A característica mais importante deste mecanismo de descoberta é que ele permite conectividade plug and play de forma simples, sem a necessidade de qualquer configuração pelo usuário.

2.2.2.1 FreeRTPS

O FreeRTPS tem como objetivo ser uma implementação gratuita, portável e minimalista do RTPS. Os desenvolvedores buscam prover uma opção para aplicações ROS2 em sistemas embarcados, onde os tamanhos das memórias ROM e RAM são fatores críticos.

Atualmente, o sistema só oferece suporte para comunicação em redes Ethernet no protocolo UDP, porem, os desenvolvedores almejam estender o sistema para outras interfaces de comunicação, tais como: USB, serial, entre outros que são comumente integrados na maioria dos microcontroladores (FREERTPS, 2017).

O sistema foi desenvolvido com o objetivo de oferecer portabilidade, não possuindo assim nenhuma dependência de execução. Para compilar e executar o FreeRTPS em um micro-controlador é necessário um compilador cross-compiler. No momento, o sistema original oferece suporte apenas para micro-controladores STM32 da empresa ST. Todos os requisitos e passos para fazer uso do sistema estão no repositório github do ROS2 (PREREQUISITES, 2017).

2.3 ROS

O desenvolvimento de software para a área da robótica possui diversos desafios. A reutilização de códigos ao escrever softwares para robôs é uma tarefa difícil devido à grande variedade de hardwares. Essa complexidade cresce na medida em que a diversidade de robôs utilizados no sistema também cresce. Um sistema para robótica inclui diversas áreas do desenvolvimento de software, indo do nível do desenvolvimento de drivers para a comunicação com o hardware até o desenvolvimento de aplicações de alto nível de abstração (JULIO, 2015)

O ROS é um framework flexível para desenvolvimento de software para robôs. É um sistema composto por uma coleção de ferramentas, bibliotecas e convenções que visa simplificar a tarefa de criar sistemas robóticos complexos e robustos em uma grande variedade de plataformas robóticas (ABOUT..., 2017).

O framework foi originalmente desenvolvido em 2007 pelo SAIL (*Stanford Artificial Intelligency Laboratory*) com o suporte do Stanford AI Robot Project e provê as facilidades de um sistema operacional padrão, como: a abstração de hardware, controle de dispositivos de baixo nível, implementação de funcionalidades largamente utilizadas, trocas de mensagens entre processos e gerenciamento de pacotes (MARTINEZ; FERNÁNDEZ, 2013).

Como descrito anteriormente o ROS oferece diversas vantagens no desenvolvimento de aplicações robóticas tanto para empresas, como para projeto pessoais e pesquisas em laboratórios ou no meio acadêmico. Tendo como uma de suas atribuições a reutilização de código, a capacidade de programação em uma vasta gama de robôs e possuindo licença livre, o sistema encoraja os desenvolvedores e pesquisadores a compartilhar suas aplicações e bibliotecas. Assim, esse disponibilidade de recursos faz com que o desenvolvimento na área de robótica cresça de forma mais rápida, uma vez que grupos de usuários podem fazer uso de ferramentas que são especialidade de outro grupo, e de forma recíproca esse grupo pode fazer uso de bibliotecas desenvolvidas por outros grupos. Por exemplo, podemos citar o caso de um pesquisador, que efetua trabalhos na área de mapeamento e, que necessita de recursos de visão (ROS..., 2017c).

Apesar do ROS2 estar em desenvolvimento, o ROS1 continua sendo a versão estável para desenvolvimento. A seguir serão apresentadas algumas das características da primeira versão do sistema.

O ROS foi projetado para ser tão pequeno quanto o possível, de forma a não "bagunce" o código do usuário, de tal modo que os códigos escritos em ROS possam ser utilizados com outros frameworks. Alguns dos frameworks que são integrados com ROS são: OpenRAVE, Orocos e Player. Outro objetivo foi a independência de linguagem, o ROS possui suporte para C++, Python, Lisp e possui bibliotecas experimentais em Java, Lua, C# e Ruby (ROS..., 2017c).

A versão inicial opera atualmente apenas em sistemas operacionais baseados em Unix. Para transmissão e recepção de dados utiliza como protocolo de transporte os protocolos TCPROS e UDPROS (em desenvolvimento).

Segundo (QUIGLEY et al., 2009), o ROS foi desenvolvido seguindo algumas metas específicas, como: arquitetura peer-to-peer, baseado em ferramentas, multilingual, pequeno, gratuito e open-source. De acordo ainda com (QUIGLEY et al., 2009), um sistema construído em ROS consiste de um conjunto de processos, que podem ser executados em diferentes hosts, conectados em uma topologia peer-to-peer. Essa topologia permite resolver diversos problemas encontrados em frameworks baseados em um servidor central.

Grafo de computação

Como descrito na página oficial do ROS, o grafo de computação é a rede peer-to-peer de processos ROS que estão processando dados juntos. Os conceitos básicos do grafo são: nós, Master, Parameter Server, mensagens, serviços, tópicos e bags, os quais fornecem dados ao grafo de diferentes maneiras, figura 7.

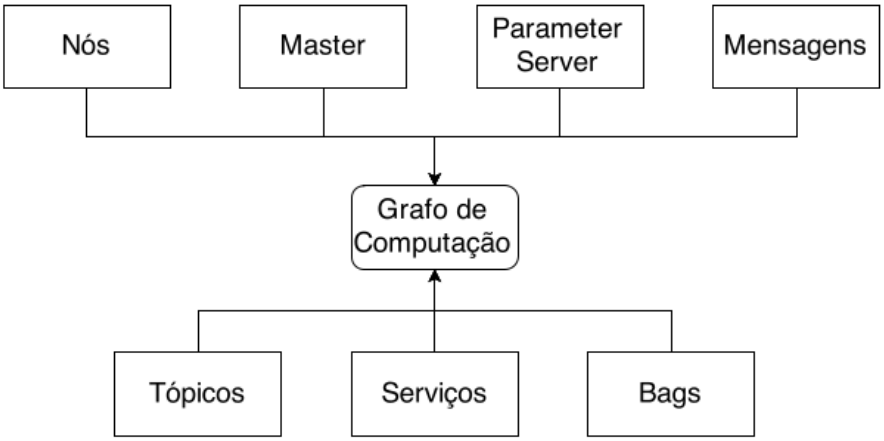

Figura 7 – Sistema de arquivos do ROS

Fonte: (JULIO, 2015)

Nós: São processos que efetuam computação. O ROS é projetado para ser modular em uma escala fina de forma que, por exemplo, um sistema de controle de robôs geralmente possui vários nós. Nesse exemplo poderíamos ter um nó para controlar o laser, um nó para controlar o motor das rodas, outro nó para efetuar o planejamento de rota e assim por diante. Um nó é escrito utilizando uma biblioteca cliente do ROS, tal como roscpp, no caso da linguagem C++.

Master: O master provê registro de nomes e uma tabela de consulta para o resto do grafo. Sem o auxilio do master, os nós não seriam capazes de encontrar outros nós, trocar mensagens ou invocar serviços.

Parameter Server: O Parameter Server é um sub-módulo do Master que permite que dados sejam armazenados na forma de chave-valor em um local central disponível para outros nós da rede.

Mensagens: Uma mensagem é simplesmente uma estrutura de dados contendo campos com tipos, é utilizada na comunicação dos nós. Tipos primitivos padrões como: inteiros, ponto flutuante, boleanos e outros são suportados, assim como arrays desses tipos. O ROS também define alguns tipos de estrutura, esses tipos são baseados em necessidades reais e de forma que ajude o programador simplificando a programação como, por exemplo, a estrutura de mensagem “Point” que possui os campos x, y e z.

Tópicos: As mensagens são enviadas através de um sistema baseado no padrão publicador assinante. Para enviar uma mensagem um nó deve a publicar em um tópico. O tópico é um nome que é utilizado para identificar o conteúdo da mensagem. Um nó que esteja interessado em certo tipo de dado deve assinar o tópico apropriado. Podem haver múltiplos publicadores e assinantes concorrentes para um único tópico, da mesma forma que um único tópico pode publicar e/ou assinar múltiplos tópicos. Em geral, publicadores e assinantes não estão cientes da existência uns dos outros. A idéia é desacoplar a produção de informação de seus consumidores.

Serviços: O modelo publicador/assinante é um paradigma de comunicação muito flexível, mas sua comunicação de caminho único e de muitos para muitos não é apropriada para ações de requisição/resposta, nas quais são requeridas em sistemas distribuídos. Uma requisição/resposta é feita através de serviços, que são definidos por um par de mensagens estruturadas: uma para a requisição e outra para a resposta. Um nó provê um serviço sob um nome e um cliente usa esse serviço enviando a mensagem de requisição e aguardando a resposta.

Bags: Bag são formatos para gravar e reproduzir dados de mensagens ROS. Bag é um importante mecanismo para armazenar dados, tais como dados de sensores, que podem ser árduos para coletar, mas são necessários para desenvolvimento e teste de algoritmos.

2.3.1 ROS2

Uma grande quantidade de código ROS que é utilizado hoje é compatível desde a versão 0.4 (Mango Tango), lançada em fevereiro de 2009. Apesar de ser uma grande vantagem, em relação à estabilidade e reutilização de código, a arquitetura em que o sistema foi elaborado já não se mostra a melhor para o estado atual e futuro do sistema (WHY..., 2017).

A conclusão dos desenvolvedores foi que, simplesmente alterar o sistema atual poderia ser muito arriscado, uma vez que o sistema já é utilizado por muitas pessoas. Assim, uma das principais razões para iniciar o projeto da nova versão do sistema, o ROS2, foi a chance de poder aprimorar e redefinir as APIs (*Application programming interface*) do sistema. Para alcançar o nível desejado, está sendo utilizando ao máximo as experiências da comunidade com a primeira versão do sistema (WHY..., 2017)

Apesar da redefinição das APIs do sistema para a nova versão, os conceitos chaves do framework serão mantidos, como: processamento distribuído, publicação/subscrição anônima de mensagens, chamada remota de rotinas (RPC, *Remote procedure call*) com resposta, neutralidade de linguagem de programação, entre outros. Ainda assim, não se deve esperar que a nova versão seja compatível com o código da primeira, mas, mesmo assim haverá mecanismos para garantir a interoperabilidade entre as duas versões, utili-

zando pacotes como, por exemplo, o ROSBridge.

Dentre as principais mudanças entre a primeira e a segunda versão (CHANGES..., 2017), temos:

a) o ROS2 está sendo testado e suportado no Ubuntu Xenial, OS X El capitain e Windows 10;

b) o ROS 2 faz muito uso do C++11 e algumas partes estão em C++14;

c) ROS2 possui como requisito a versão 3.5 do Python;

d) o ROS2 não necessidade de uma master para executar, a descoberta de outros nós é distribuída;

e) o código gerado usa namespaces separados para evitar colisões de nomes;

f) no ROS2 os tipos de dados de duração e tempo são definidos como mensagens, oferecendo consistência entre linguagens, e não mais como estrutura de dados;

g) não possui atualmente o conceito de ações;

h) no ROS 2 o roslaunch será escrito em python ao invés de XML (*eXtensible Markup Language*);

i) será possível executar processos de tempo real, quando utilizando sistemas operacionais RTOS;

No núcleo do ROS 1 está um sistema middleware de publicação e subscrição anônimo construído quase inteiramente do zero. A partir de 2007, foi elaborado um próprio sistema de descoberta, definição de mensagens, serialização e transporte. Do inicio do desenvolvimento do ROS 1 para os dias atuais, surgiram várias novas tecnologias relevantes para ROS (WHY..., 2017), tais como:

a) Zeroconf;

b) Protocol Buffers;

c) ZeroMQ (and the other MQs);

d) Redis;

e) WebSockets;

f) DDS.

Com auxilio dessas tecnologias, agora já é possível construir um sistema de middleware tipo ROS usando bibliotecas de código aberto. Através dessa abordagem, é possível então se beneficiar de várias maneiras, tais como:

a) menos código para manter, especialmente códigos não específicos em robótica;

b) aproveitar recursos das bibliotecas; e

c) beneficiar de melhorias contínuas feitas para essas bibliotecas.

A RCL (*ROS Client Library*) define uma API que expõe os conceitos de publicação e subscrição aos usuários. No ROS1 a implementação desses conceitos foi construída em protocolos customizados como o TCPROS e o UDPROS. No ROS2 a decisão tomada foi de construir a RCL no topo de soluções middleware existentes (DDS) (ROS. . . , 2017a).

Existem diversas implementações DDS e cada uma dessas implementações possuem seus pros e contras. Dessa forma, o ROS2 possui como objetivo suportar múltiplas implementações DDS, e assim, permitindo o usuário definir a melhor implementação para sua aplicação. Para isso, uma interface de abstração de middleware foi inserida entre a RCL e as implementações DDS, essa interface de abstração é apresentada na figura 8.

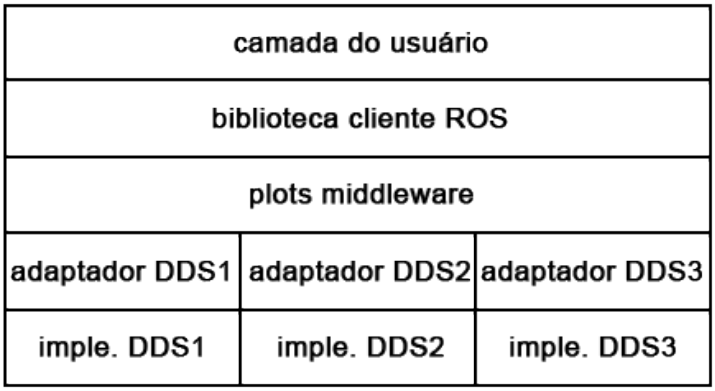

Figura 8 – A interface middleware

Fonte: (ROS. . . , 2017a)

A RCL então esconde do usuário qualquer implementação do DDS, isso para esconder a complexidade relativa ao DDS e suas APIs (PROPOSAL. . . , 2017).

Dentre a implementações RMW (*ROS Middleware*) suportadas até o momento, temos as apresentadas na tabela 2.

Outra proposta para integração ao ROS2 é o suporte a tempo real. Para isso, propõe-se uma abordagem iterativa para identificar as limitações que levam o sistema a um comportamento não determinístico e imprevisível na API. Assim, será possível identificar soluções para melhorar o desempenho do sistema.

Nome	Licença	Implementação RMW	Situação
eProsima Fast RTPS	Apache 2	rmw_fastrtps_cpp	Suporte total. RMW padrão. Empacotado com versões binárias.
RTI Connext	comercial, pesquisa	rmw_connext_cpp	Suporte total. Compilação do código fonte necessária.
RTI Connext (implementação dinâmica)	comercial, pesquisa	rmw_connext_dynamic_cpp	Suporte pausado. Suporte total até o alpha 8.*
PrismTech Opensplice	LGPL (*GNU Lesser General Public License*) (apenas v6.4), comercial	rmw_opensplice_cpp	Suporte pausado. Suporte total até alpha 8.*
OSRF FreeRTPS	Apache 2	–	Suporte parcial. Desenvolvimento pausado.

Tabela 2 – Implementações RMW suportadas

Fonte: https://github.com/ros2/ros2/wiki/DDS-and-ROS-middleware-implementations

Nota: *Atributos adicionados a verões mais recentes não foram implementados nessas RWMs.

2.4 Trabalhos relacionados

Para elaboração desse trabalho será utilizado o FreeRTPS. O FreeRTPS é uma implementação gratuita, portável e minimalista do RTPS. Esse sistema busca ser uma implementação "compacta" que possa ser embarcado em uma variedade de microcontroladores, usando apenas a memória interna, e prover aplicações ROS2 em sistemas embarcados de baixo custo (FREERTPS, 2017).

Com o inicio do desenvolvimento da segunda versão do framework ROS, e com a mudança da camada de comunicação do sistema, o projeto do FreeRTPS oferece a chance do ROS se inserir no mundo dos sistemas embarcados de uma forma natural, direta e sem pontes. Apesar da capacidade de processamento do sistema na placa de desenvolvimento STMDISCOVERY, único microcontrolador suportado até o momento (FREERTPS, 2017), o sistema pode não conseguir fornecer requisitos de tempo dependendo da complexidade do software embarcado.

O rosserial é uma das alternativas existentes para comunicação entre sistemas embarcados e a primeira versão do ROS. O rosserial é uma protocolo para "empacotar" mensagens serializadas do padrão ROS e multiplexa-las para vários para dispositivos periféricos, tal como porta serial ou socket (ROSSERIAL, 2017). O formato do pacote pode ser observado na figura 9. Essa ferramenta fornece suporte para plataforma Arduino, sistemas Linux embarcados, aplicações no sistema operacional Microsoft Windows, pla-

taformas mbed e para alguns microcontroladores da família Tiva da Texas Instruments. Para que o dispositivo embarcado possa enviar e receber mensagens é necessário existir um nó rosserial no computador hospede. Esse nó de interface fica responsável por efetuar a ponte entre o dispositivo utilizando a biblioteca rosserial com o resto da rede ROS, figura 10. Essa interface possui implementações em python (rosserial_python), C++ (rosserial_server) e Java (rosserial_java).

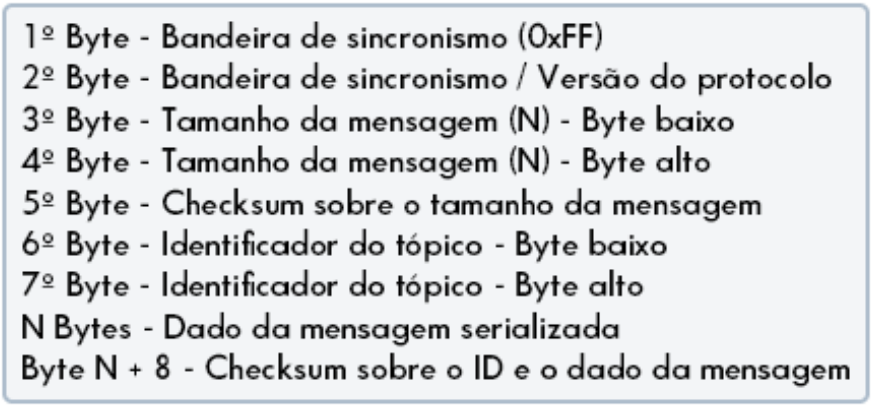

Figura 9 – Formato do pacote

Fonte: (ROSSERIAL, 2017)

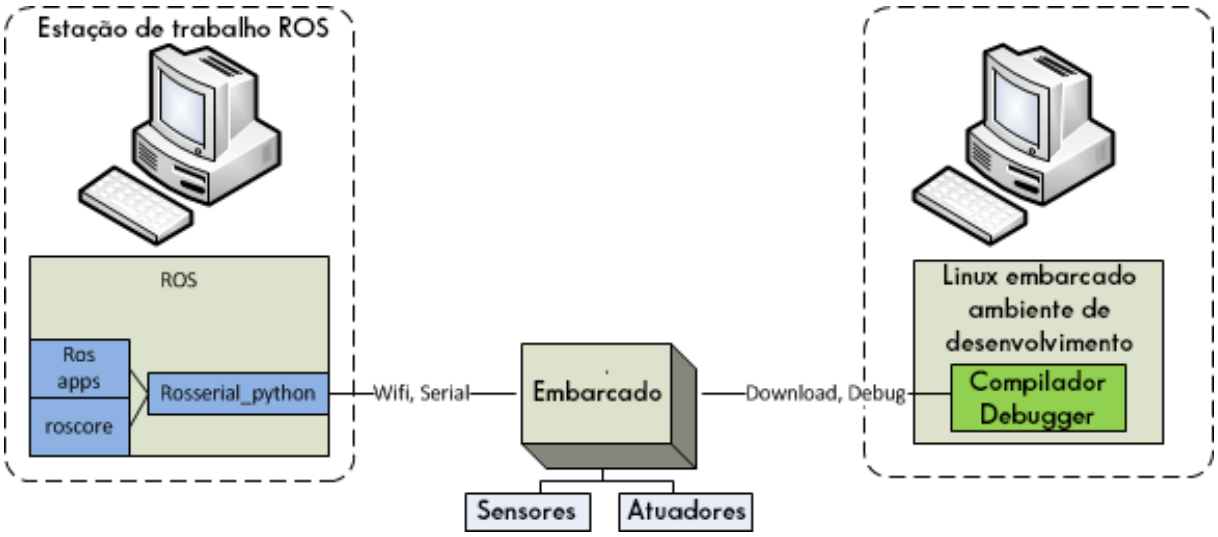

Figura 10 – Estrutura do rosserial

Fonte: (ROSSERIAL_EMBEDDEDLINUX, 2017)

Outra ferramenta desenvolvida que pode efetuar a interface entre programas ROS embarcados é o rosbridge. O rosbridge provê uma API em JSON (*JavaScript Object Notation*), um padrão de texto baseado em um subconjunto da linguagem JavaScript para trocas de dados (INTRODUÇÃO. . . , 2017), para funcionalidades ROS em sistemas não

ROS (ROSBRIDGE_SUITE, 2017). Esse sistema possui a biblioteca rosbridge_library, responsável por receber mensagens JSON e convertê-las para o ROS e vice-versa, a API rosapi, que fornece funções em JSON para sistemas não ROS, e por fim o rosbridge_server, que provê uma conexão Websocket para que sistemas enviem suas requisições em JSON, figura 11.

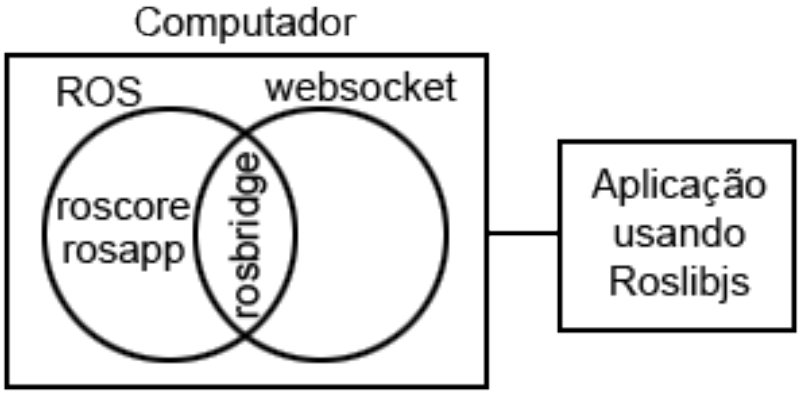

Figura 11 – Estrutura do rosbridge

Uma das formas de se utilizar o rosbridge para comunicação é através do roslibjs. O roslibjs é uma biblioteca JavaScript que fornece funções compatíveis com o ROS e também efetua a conexão da aplicação com o rosbridge_server. Ela fornece funcionalidades ROS, tais como: publicação, subscrição, chamada de serviços, actionlib e outras (ROSLIBJS, 2017).

Através dessa ferramenta, é possível então efetuar a comunicação com aplicações ROS em dispositivos embarcados, que possuam conexão Ethernet e suporte a browser, como smartphones ou sistemas operacionais embarcados. Em <http://wiki.ros.org/roslibjs/Tutorials>, é descrito como utilizar um telefone celular para publicar mensagens de vídeo, através de sua câmera, e de aceleração, através de seu acelerômetro.

O uROSnode é um cliente ROS compacto escrito em C que pode ser executado em microcontroladores modernos, como ARM Cortex M. Ele oferece as principais características necessárias para que um nó ROS possa ser executado. <http://wiki.ros.org/uROSnode>. O uROSnode foi programado com um sistema operacional RTOS em mente, então ele conta com algumas primitivas de um sistema operacional, como processos e mutex. Atualmente ele é portado para ChibioOS, um sistema operacional para dispositivos embarcado, e POSIX (*Portable Operating System Interface*).

O rosc é uma implementação de cliente ROS em C e sem nenhuma dependência, que visa suportar pequenos sistemas embarcados, bem como qualquer sistema operacional. Possui como objetivo se tornar uma implementação eficiente e altamente portável do middleware do ROS, tornando-o um boa escolha para uso em aplicações industriais ou

desenvolvimento de produtos (ROSC, 2017). Para bare metal, a memória é alocada estaticamente em tempo de compilação. A quantidade de memória para alocação de mensagens deve ser definida pelo usuário. Em versões com sistema operacional, não é necessário definir a quantidade de memória, uma vez que ela é alocada em tempo de execução

No repositório stm32, do usuário nosch-ros-pkg no github, foi desenvolvido um sistema que integra uma placa de desenvolvimento embarcada STM32F4Discovery, o sistema operacional embarcado FreeRTOS, um middleware ROS embarcado e um client library ROS embarcado. O client library embarcado possui a capacidade de criar nós, publicadores, subscritores e definir mensagens ROS (PROGRAM..., 2017). As camadas desse sistema podem ser observadas na figura 12, já o conceito do software pode ser observado na figura 13. A última submissão no repositório foi em 4 de setembro de 2016.

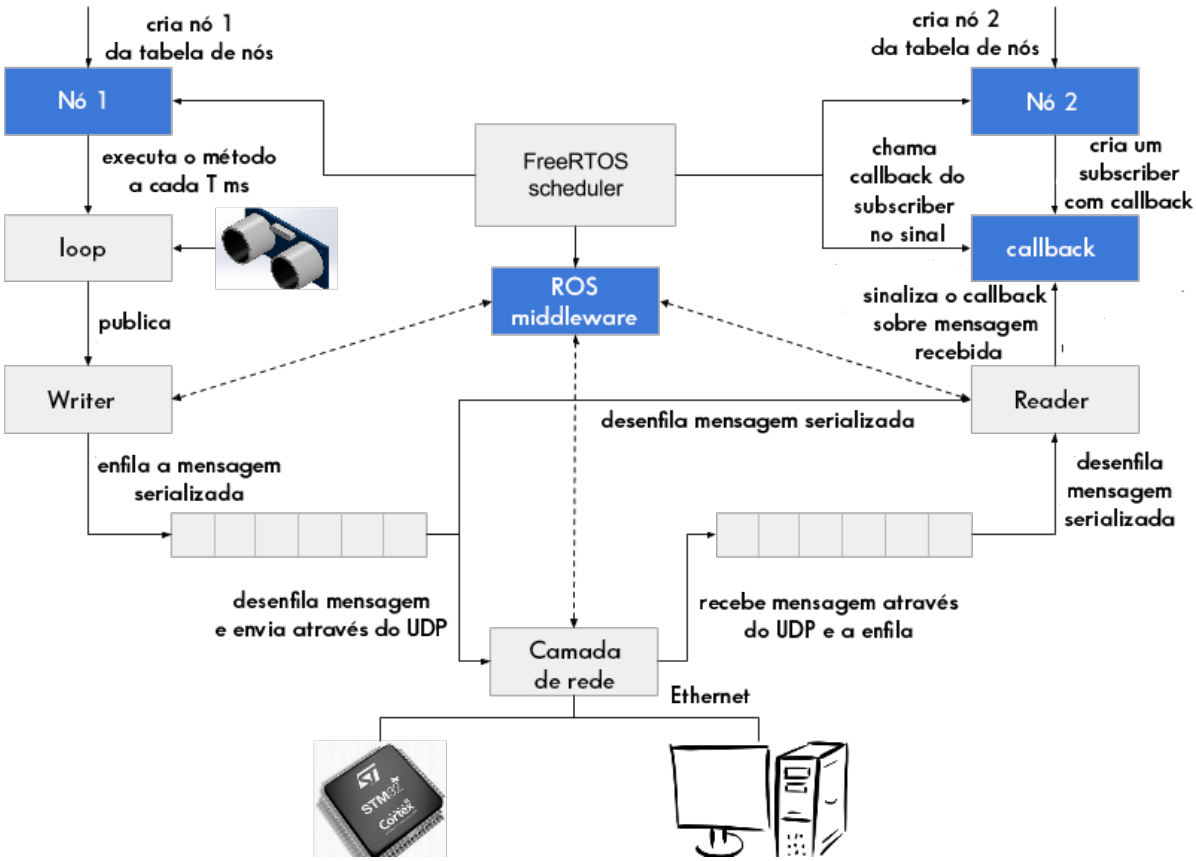

Figura 12 – Conceito do ROS STM32

Fonte: (PROGRAM..., 2017)

Na página do mesmo usuário ainda existe o repositório ROS 2.0 NuttX, esse repositório possui o protótipo de um sistema embarcado utilizando ROS2, a implementação DDS Tinq e o sistema operacional NuttX em um microcontrolador da família STM32F4 da ST (ROS..., 2017b). A estrutura implementada por esse arquitetura está presente na

Figura 13 – Estrutura do ROS STM32

Fonte: (PROGRAM. . . , 2017)

figura 14. De acordo com a descrição encontrada no repositório, esse sistema possui uma capacidade de publicar 50 mensagens por segundo.

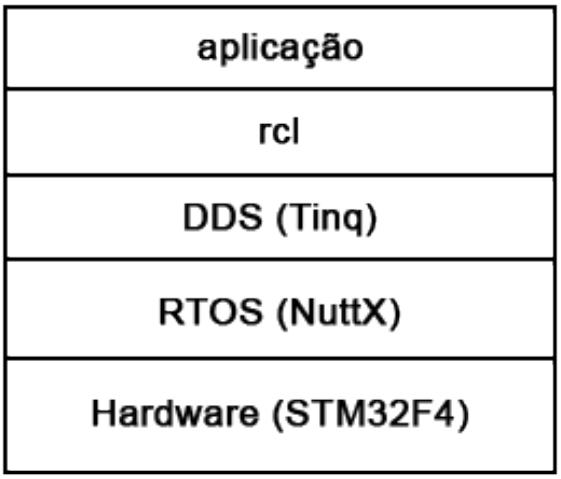

Figura 14 – Estutura do ROS 2.0 NuttX

Fonte: (ROS. . . , 2017b)

3 DESENVOLVIMENTO

Nesse capítulo será apresentado a estrutura original do FreeRTPS, bem como o modelo proposto, e as modificações necessárias nos códigos do FreeRTPS e do FreeRTOS para se atingir esse modelo.

3.1 Modelo proposto

Como descrito no capitulo anterior, existe atualmente uma grande busca em oferecer processamento de tempo real ao ROS. As formas atuais para se atingir esse objetivo são através de pontes entre o ROS e sistemas que possam oferecer essa característica. Observou-se também que já existem ferramentas para se executar o sistema ROS2 em sistemas embarcados e ao mesmo tempo oferecer recursos para se obter processamento em tempo real.

Porém, não foi encontrado nenhum tipo de pesquisa com o foco em garantia de tempo real utilizando o sistema FreeRTPS, que tem como ponto central oferecer uma camada de comunicação direta e leve para sistemas embarcados com o ROS. As implementações encontradas, que possuem tempo real e comunicação com o ROS, ou ROS2, possuem um pouco mais de complexidade e maior tamanho do que o sistema aqui proposto. Já os sistemas que utilizam o recurso de ponte acabam possuindo uma latência intrínseca em função do processamento intermediário na transição das mensagens.

Buscou-se desenvolver então uma integração entre o FreeRTPS e o FreeRTOS e assim juntar as principais qualidades dos dois sistemas. Espera-se obter com o sistema final uma ferramenta com bom desempenho, ferramentas para oferecer processamento em tempo real e facilidade de utilização.

3.1.1 Arquitetura original do FreeRTPS

Iniciou-se o desenvolvimento do trabalho efetuando uma avaliação profunda no código do FreeRTPS. Através dessa avaliação, obtiveram-se então os requisitos de hardware necessários, fluxo de execução do código do sistema, divisão dos processos em camadas de operação e organização dos arquivos.

Nota-se que a árvore do diretório do FreeRTPS foi desenvolvida com a reutilização de código em mente. Tem-se então que sistemas que possuam códigos de driver em comum como, por exemplo, famílias de microcontroladores que compartilham de uma mesma biblioteca, possuam uma pasta para os arquivos compartilhados e outra para código especifico. Esse detalhe pode ser visualizado na figura 15:

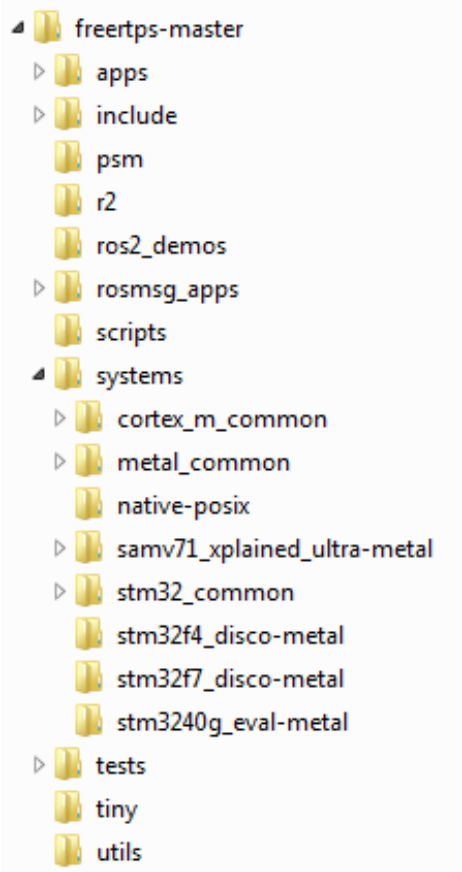

Figura 15 – Diretório do FreeRTPS

Verificou-se também que o sistema necessita basicamente de dois periféricos obrigatórios do microcontrolador, um dos periféricos é o timer, utilizado para fornecer o tempo de operação do sistema e também para o sincronismo das mensagens, e o outro é o periférico de Ethernet, utilizado para efetuar a comunicação entre dispositivos através do envio e recebimento de mensagens pela rede Ethernet.

Para se obter a fluxo de execução do código do FreeRTPS, analisaram-se então as rotinas de inicialização do sistema e dos recursos e também o fluxo das mensagens que são enviadas e recebidas pelo dispositivo. As funções de inicialização basicamente: definem o ID do sistema, habilitam o periférico de Ethernet junto a sua interrupção de hardware, iniciam os recursos dos publicadores e assinantes e também os readers e writers do modulo de descoberta.

Em seguida, analisou-se o fluxo das funções que são executadas periodicamente durante a execução do sistema FreeRTPS:

a) Função discover: após a chamada da função discover, uma mensagem de descoberta é enviada a rede. Essa mensagem contém o ID do sistema e o endereço de IP do mesmo.

b) Função listen: como todas as mensagens do sistema são armazenadas em uma fila circular, a chamada da função listen executa o processamento de cada mensagem presente nesse buffer. Em um primeiro momento e verificado se o pacote é uma mensagem UDP/IP, e em seguida se é uma mensagem RTPS válida. No caso de ser uma mensagem assinada pelo dispositivo, o callback da função responsável por processá-la é chamada. Já no caso de ser uma mensagem de descoberta de outro dispositivo, o ID do mesmo é armazenado e uma mensagem é enviada de volta para ele, requisitando quais tópicos são assinados e publicados por ele.

Por fim, avaliou-se a rotina que pode ser executada várias vezes depois do início do programa:

a) Publish: efetua-se a elaboração de uma nova mensagem RTPS do tipo DATA e verifica-se se o dispositivo já possui conhecimento de assinante do tópico que será publicado. Efetua-se então o envio da mensagem para todos os dispositivos que assinam esse tópico, isso é feito através de um loop que verifica todos assinantes conhecidos pelo dispositivo que está enviando a mensagem.

Através da análise mais aprofundada do sistema, definiu-se quais pontos do sistema FreeRTPS serão editados ou removidos para se obter a arquitetura desejada.

A pilha UDP/IP utilizada pelo FreeRTPS foi desenvolvida juntamente com o sistema. Apesar da mesma possuir licença aberta, a implementação possui algumas limitações como: operação apenas com IP estático (não possui DHCP), filtragem simples das portas e alocação estática para armazenamento das mensagens recebidas pelo periférico Ethernet.

Durante a execução da aplicação, chama-se a função 'listen' que recebe como parâmetro um intervalo de tempo fixo, a critério do usuário, para efetuar a decodificação das mensagens RTPS recebidas e armazenadas no buffer. Nos exemplos do FreeRTPS, a função 'listen' é utilizada como um delay, ou seja, o resto do código principal é executado em intervalos de tempo que são passados como parâmetro da função.

Ainda durante a execução do programa principal, executa-se a função 'frudp_disco_tick' para enviar mensagens de descoberta na rede. Essa função é responsável por empacotar uma mensagem, de acordo com o protocolo RTPS, com as características da aplicação. A partir dessa mensagem, outras aplicações que estão na mesma rede são notificados sobre a existência do dispositivo. O código apresentado no listing 3.1 apresenta um exemplo de aplicação utilizando o sistema FreeRTPS.

```
1  /*Funcao principal*/
2  int main( int argc, char **argv ){
3      /*Inicia o sistema FreeRTPS*/
4      freertps_system_init();
5      /*Publicador para o topico chatter*/
6      frudp_pub_t *pub = freertps_create_pub( "chatter",
           ↪ std_msgs__string__type.rtps_typename );
7      /*Inicia o sistema de descoberta*/
8      frudp_disco_start();
9
10     /*Variaveis utilizadas pelo publicador*/
11     struct std_msgs__string msg;
12     char data_buf[ 10 ] = { 0 };
13     msg.data = data_buf;
14     uint8_t cdr[ 10 ] = { 0 };
15
16     /*Loop principal*/
17     while( freertps_system_ok() ){
18         /*Processamento das mensagens recebidas pelo periferico Ethernet*/
19         frudp_listen( 500000 );
20         /*Envio de mensagens de descoberta na rede*/
21         frudp_disco_tick();
22         /*Preparando e publicando uma mensagem string*/
23         snprintf(msg.data, sizeof( data_buf ), "Hello" );
24         int cdr_len = serialize_std_msgs__string( &msg, cdr, sizeof( cdr ))
               ↪ ;
25         freertps_publish( pub, cdr, cdr_len );
26     }
27     /*Finalizando sistema*/
28     frudp_fini();
29     return 0;
30 }
```

Listing 3.1 – Exemplo de aplicação desenvolvida utilizando o sistema FreeRTPS original

3.1.2 Arquitetura proposta

A arquitetura desejada neste trabalho consiste da integração de um sistema operacional embarcado, o FreeRTOS, e do protocolo de comunicação FreeRTPS. Como o FreeRTPS é um recurso que será utilizado pela aplicação do usuário para efetuar a publicação e assinatura de mensagens, no sistema proposto ele será executado como um processo e assim gerenciado pelo sistema operacional.

Pretende-se obter nesse trabalho, um sistema que ofereça: facilidade de uso, portabilidade e recursos para processamento em tempo real. Para isso, propõe=se a seguinte arquitetura, figura16. Ela possui dois níveis na vertical: um nível é composto pela região infra-estrutura e o outro pela região aplicação. Na horizontal tem-se 3 níveis: um representando os processos, outro representando as filas do sistema e por último a região representando as bibliotecas.

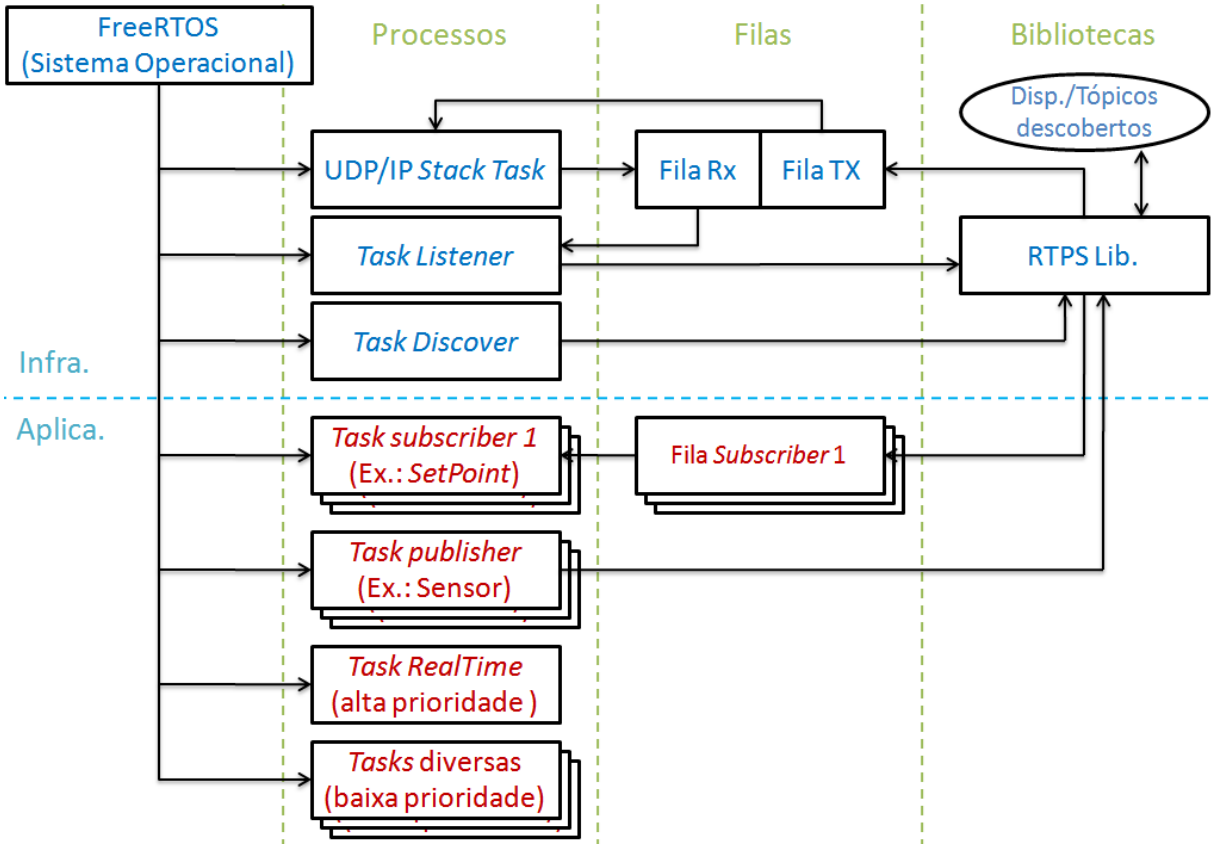

Figura 16 – Modelo simples da arquitetura proposta

Nessa arquitetura, tem-se o sistema operacional FreeRTOS gerenciando todos os processos, incluindo: processos do usuário e processos do sistema FreeRTOS+FreeRTPS. Com um sistema operacional pode-se ter melhor aproveitamento dos recursos do microcontrolador, uma vez que, todos os momentos que um processo estiver em estado parado, o processamento da CPU pode ser utilizado por processos que estejam executando ou pode-se colocar o microcontrolador em um estado de baixo consumo se não houver processos executando.

Na região de interseção entre infra-estrutura e processos temos:

a) O processo da pilha UDP/IP executando o código que gerencia as mensagens que estão chegando e saindo pela porta Ethernet. Esse processo coleta as mensagens armazenadas na fila TX e as lança na rede e armazena todas as

mensagens recebidas na fila RX, para processamento futuro.

b) o processo listener fica responsável pela leitura e interpretação das mensagens UDP recebidas, determinando o tipo da mensagem e tomando as ações necessárias. Coleta todas as mensagens recebidas no soquete da porta RTPS e as decodifica para identificar outros nós ou receber dados deles. Quando recebe uma mensagem de descoberta, a biblioteca RTPS armazena o novo nó. Quando recebe uma mensagem de dado, a biblioteca RTPS armazena os dados na fila do tópico do respectivo assinante.

c) o processo discover é responsável pelo envio de mensagens de descoberta na rede, notificando assim a sua existência para outros dispositivos, bem como os tópicos que são subscritos e publicados pelo dispositivo. Utiliza rotinas da biblioteca RTPS para enviar mensagens RTPS de descoberta na rede, a rotina discover do RTPS codifica uma mensagem RTPS e a coloca na fila TX.

Na região de interseção entre aplicação e processos temos:

a) Subscriber task 1...N representando todos os processos que tratam os dados de seus respectivos assinantes. Cada processo assinante tem sua própria fila onde ele verifica a recepção de novos dados.

b) Publisher task 1...N representa todas os processos que chamam rotinas de publicação, da biblioteca RTPS. Quando uma rotina de publicação é chamada, o RTPS verifica se conhece o nó que inscreve esse tópico, cria uma massagem formatada no protocolo RTPS e a envia para a fila TX.

c) Real-time tasks representa processos que executam processamento em tempo real, essas tarefas devem ser bem implementadas para garantir os requisitos de tempo real.

d) General task representa processos que executam processamento geral, como: impressão em tela, status de botões e outros.

O caminho de uma mensagem recebida é apresentado na figura 17. Primeiro, o hardware recebe uma mensagem e filtra-a para verificar se é uma mensagem válida, caso positivo a coloca na pilha UDP/IP. Depois disso, a processo UDP/IP decodifica o pacote Ethernet e armazena o dado na sua fila UDP/IP RX. Quando o processo Listener está ativo e em execução, ele obtêm todas as mensagens da fila UDP/IP RX endereçadas à porta do soquete RTPS. Em seguida, o listener chama uma rotina RTPS para decodificar a mensagem RTPS para armazenar um novo dispositivo conhecido, se for uma mensagem de descoberta, ou colocar os dados em uma das filas dos assinantes, se a mensagem for do tipo dado. Finalmente, quando o dado é adicionado a fila de seu assinante, o assinante vai ao estado ativo e, quando começar a executar, lê os dados armazenados na fila para

executar o código programado para manipular os dados. Depois disso o processo do assinante volta ao estado inativo até que um novo dado chegue.

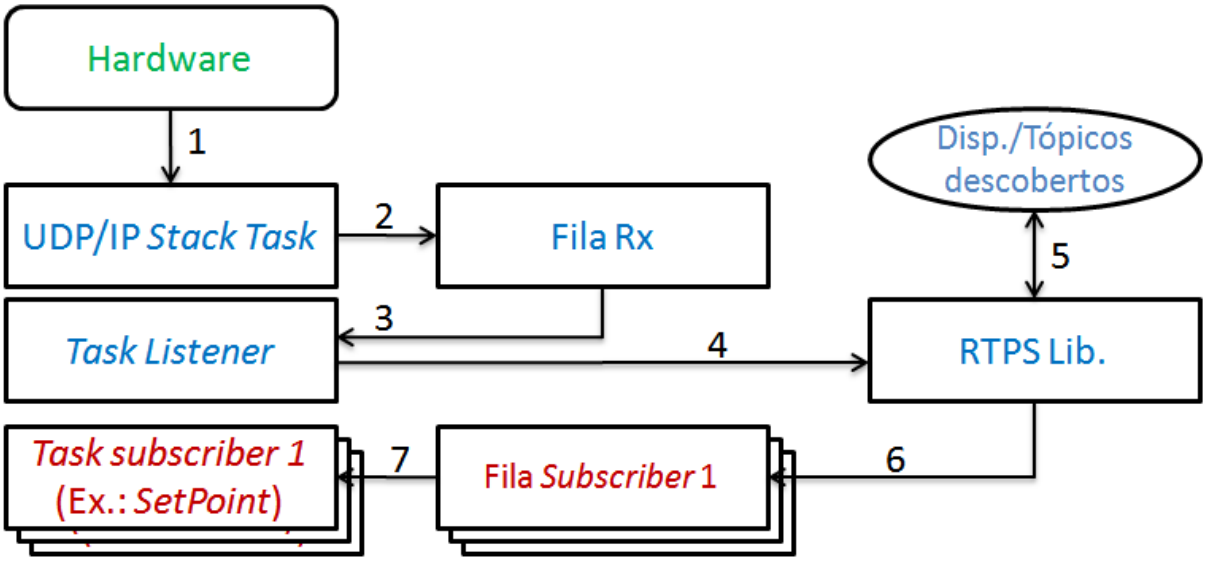

Figura 17 – Caminho das mensagens recebidas

O caminho de um dado de saída é apresentado na figura 18. Em caso de descoberta, a tarefa de descoberta efetua a chamada de uma rotina RTPS para codificar uma mensagem de descoberta RTPS e a coloca na fila TX. No caso do publicador, o sistema chama uma rotina RTPS, para verifica se o dispositivo conhece algum nó assinando o tópico publicado, codifica uma mensagem RTPS do tipo dado e a coloca na fila TX. Quando o processo de pilha UDP/IP é executado e a fila TX possui mensagens para despachar, a pilha UDP/IP codifica a mensagem em um pacote Ethernet e a envia para o hardware. Finalmente, o hardware lança a mensagem na rede.

A modelo original do FreeRTPS pode ser observado na figura 19.

O modelo do sistema depois das modificações propostas pode ser visto na figura 20. Observa-se que nesse contexto a aplicação pode ter acesso aos recursos do FreeRTPS, do FreeRTOS e do hardware, enquanto o FreeRTPS possui apenas acesso aos recursos do sistema operacional e da pilha UDP. Dessa forma a aplicação torna-se independente do FreeRTOS e do FreeRTPS.

Para facilitar a compreensão do código e também o desenvolvimento de aplicações utilizando o sistema, definiu-se uma interface que define o modelo de programação que deve ser utilizado. Dessa forma as aplicações desenvolvidas não precisam ter seus códigos alterados em alto nível, apenas em nível de hardware.

A função principal, main, foi definida de forma a inserir as configurações de clock e periféricos do hardware utilizado. A função setup foi implementada para definir os tópicos que serão utilizados para publicações ou assinaturas e também os processos que serão

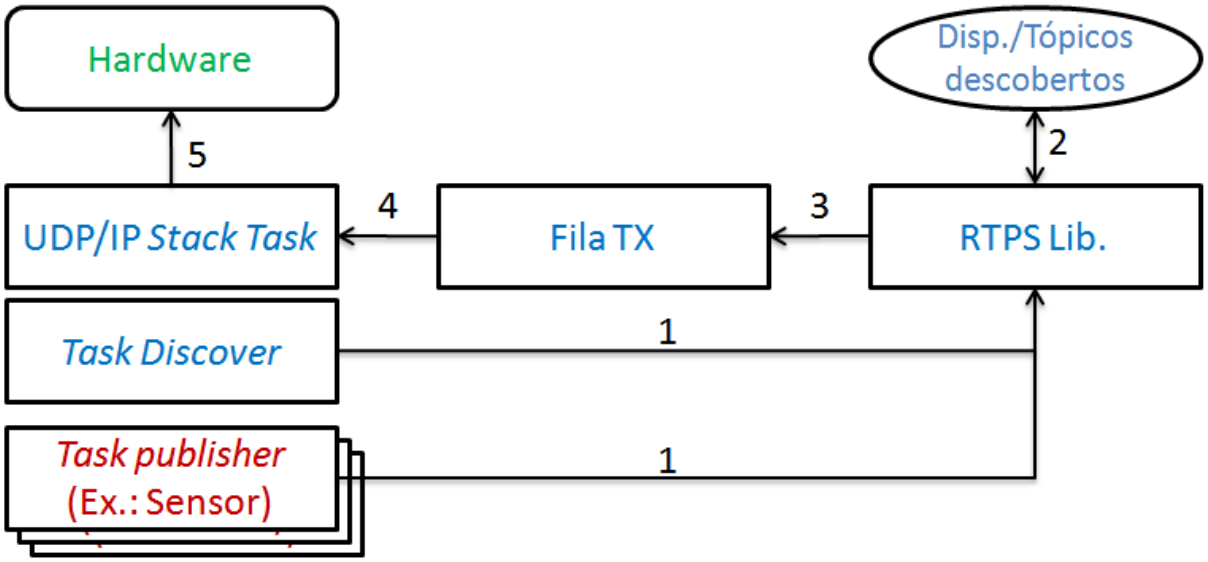

Figura 18 – Caminho das mensagens enviadas

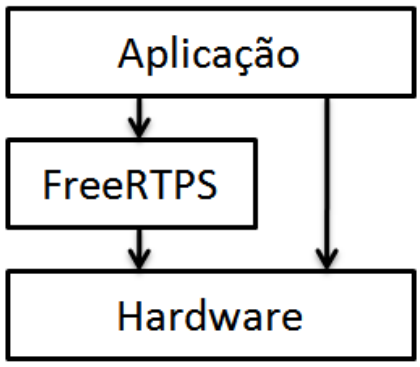

Figura 19 – Modelo FreeRTPS

executados. A estrutura final do sistema é apresentada no listing 3.2.

```
1  static void genericTask( void *parameters );
2  frudp_pub_t *publisher;
3
4  //Device Ethernet MAC address
5  uint8_t ucMACAddress[ 6 ] = { 0x2C, 0x4D, 0x59, 0x01, 0x23, 0x53 };
6  //Desired IP parameter if DHCP do not work
7  static const uint8_t ucIPAddress[ 4 ]         = { 192, 168, 2, 153 };
8  static const uint8_t ucNetMask[ 4 ]           = { 255, 255, 255, 0 };
9  static const uint8_t ucGatewayAddress[ 4 ]    = { 192, 168, 2, 0 };
10 static const uint8_t ucDNSServerAddress[ 4 ] = { 208, 67, 222, 222 };
11
12 //Main function
13 int main( void ){
```

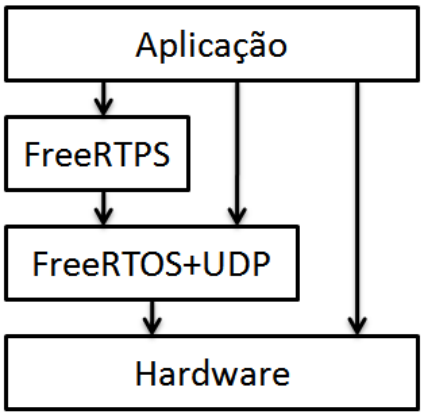

Figura 20 – Modelo proposto, FreeRTPS+FreeRTOS

```
14    //Do necessary clock configuration before FreeRTOS_IPInit
15    hardwareConfiguration();
16    //Start FreeRTOS+UDP stack
17    FreeRTOS_IPInit( ucIPAddress, ucNetMask, ucGatewayAddress,
          ↪ ucDNSServerAddress, ucMACAddress );
18    //Start operating system scheduler
19    vTaskStartScheduler();
20    //Infinite loop, the program must not reach this loop if ok
21    for(;;){}
22    return 0;
23  }
24
25  //Function to setup pubs, subs and others tasks
26  //Pub handles must be declared outside this function
27  //For pubs, use: pub = freertps_create_pub( topic name, type name );
28  //For subs, use: freertps_create_sub( topic name, type name, handle task,
        ↪ receive queue size );
29  void setup( void ){
30    //Add here peripheral init function, subscribers, publishers and general
          ↪ tasks
31
32    //Pubs here. Example pub = freertps_create_pub( topicName, typeName );
33    pub = freertps_create_pub( "talking", std_msgs__uint32__type.
          ↪ rtps_typename );
34
35    //Subs here. Example freertps_create_sub( topicName, typeName,
          ↪ handlerTask, dataQueueSize );
36    freertps_create_sub( "listening", std_msgs__uint32__type.rtps_typename,
          ↪ listeningTask, 10 );
37
38    //General tasks here. ctrlTask with greater priority. Example xTaskCreate
          ↪ ( genericTask, "genericTask", configMINIMAL_STACK_SIZE, NULL,
```

```
        ↪ tskIDLE_PRIORITY + 1, NULL );
39  }
40
41  static void genericTask( void *parameters ){
42    while( true ){
43      //Código genérico e/ou publicações
44    }
45  }
46
47  //Processo de um assinante
48  static void listeningTask( void *parameters ){
49    //Fila para armazenar as mensagens recebidas no tópico
50    QueueHandle_t sQueue = ( QueueHandle_t )parameters;
51    //Variáveis para manipular o dado recebido
52    uint8_t  msg[ 128 ] = { 0 };
53    while( true ){
54      //portMAX_DELAY indica que xQueueReceive irá colocar o processo em
          ↪ inativo até haver algum dado na fila de dados do tópico
55      if( xQueueReceive( sQueue, msg, portMAX_DELAY ) == pdPASS ){
56        //Manipulação do dado recebido
57      }
58    }
59  }
```

Listing 3.2 – Estrutura final do sistema FreeRTOS+FreeRTPS

Neste modelo, têm-se duas funções RTPS principais ao nível do usuário. Uma delas é a freertps_publish. Para usar esta função, o usuário deve criar um pub com o tipo de dado e o nome tópico a ser publicado com a função freertps_create_pub. Depois disso, deve-se chamar a função que efetivamente faz o envio da mensagem, a função freertps_publish, com o dado e o pub relacionado ao tópico desejado, conforme mostrado no seguinte código, listing 3.3.

```
1  //Create a publisher that publish at the topic "topic" and has the String
       ↪ type.
2  frudp_pub_t *pub;
3  pub = freertps_create_pub( "topic", std_msgs__string__type.rtps_typename );
4
5  //freertps_publish publish a message "message" (String type), with size of
       ↪ message_length (integer type) at the publisher "pub" (frudp_pub_t
       ↪ type)
6  freertps_publish( pub, message, message_length );
```

Listing 3.3 – Exemplo de como publicar um dado

Para usar o assinante, o usuário deve definir uma rotina static void que funcionará como um dos processos do sistema, para lidar com os dados recebidos. Nesse modelo, para registrar um assinante, o usuário deve usar a rotina "freertps_create_sub"e especificar o nome do tópico assinado, o tipo de dado do assinante, a rotina static void que foi criada para lidar com os dados e o tamanho da fila utilizada para armazenar os dados recebidos.

A rotina usada para lidar com os dados do tópico assinado deve ter o seguinte formato, listing 3.4:

```
1  static void subTaskFormat( void *parameters ){
2    //Queue responsible for handle receiver information on "topic"
3    QueueHandle_t sQueue = ( QueueHandle_t )parameters;
4    //Struct to handle the data
5    Message xMessage;
6
7    while( true ){
8      //portMAX_DELAY tells that xQueueReceive will be blocked until data
           ↪ arrive in sQueue
9      if( xQueueReceive( sQueue, &xMessage, portMAX_DELAY ) == pdPASS ){
10       //Handle the arrived message
11     }
12   }
13 }
```

Listing 3.4 – Formato do processo para manipular os dados de um assinante

O seguinte código apresenta como usar a função "freertps_create_sub"para registrar uma nova assinatura, listing 3.5.

```
1  freertps_create_sub( "topic", std_msgs__string__type.rtps_typename,
       ↪ subTaskFormat, queue_size );
```

Listing 3.5 – Função para assinar um novo tópico

Uma fila de dados do tópico é criada no momento que se assina um novo tópico através da função freertps_create_sub. O processo de manipulação do tópico assinado se coloca no estado inativo quando nenhum dado está presente na sua fila, ele só muda para o estado ativo quando existe alguma informação na fila de dados recebidos do tópico assinado. Assim, o processo de manipulação dos dados do tópico não consome processamento do hardware enquanto nenhum dado é recebido. A tarefa responsável por colocar os dados nas filas dos assinantes é o processo de *listener*, quando uma mensagem do tipo dado é enviada por outro nó e a aplicação possui a assinatura desse tópico, a mensagem chega ao dispositivo e é decodificada pelo processo *listener*, que coloca os dados descodificados na respectiva fila do tópico assinado.

A vantagem apresentada em utilizar esse método de gerenciamento dos dados de um tópico assinado, se deve ao fato de que o manipulador do assinante não possui nenhuma ligação com o código do processo listener. Dessa forma, caso um dado de um tópico assinado seja recebido, a função listener não efetua a chamada do manipulador em si, apenas insere o dado na fila do respectivo tópico. Com o dado na fila do tópico, o processo manipulador muda para o estado ativo e efetua o processamento do dado assim que esse processo entra em estado de execução.

Na FreeRTPS a manipulação do dados de um tópico assinado é chamada pela função listen através de um *callback*. Dessa forma, se a função se manipulação do dado for muito longa ou mal possuir uma lógica mal implementada, ele pausará a função listen enquanto efetua esse processamento.

3.2 Implementação

Uma vez que o FreeRTPS possui execução de código linear, com exceção das interrupções dos *timers* e periférico de Ethernet, o primeiro passo para se integrar o sistema operacional de tempo real é quebrar o código em processos menores e independentes, assim eles podem ser gerenciados pelo sistema operacional de uma forma mais dinâmica.

Para a rotina de escuta e interpretação de mensagens criou-se a processo RTOSListenTask, esse processo faz a chamada da função frudp_listen para verificar o recebimento de novas mensagens nas portas do FreeRTPS com um periodo de 1 ms. Já para a rotina de descoberta criou-se a tarefa RTOSDiscotickTask, a rotina de descoberta consistem em chamadas da função frudp_disco_tick do FreeRTPS, que efetua o envio de mensagens com informações do participante (dispositivo) e dos seus endpoints (tópicos), com um período de 1 segundo (mas que pode ser alterado a desejo do usuário). O listing 3.6 apresenta os processos listen e discover do novo sistema.

```
1  static void RTOSListenTask( void * parameters ){
2    while( true ){
3      vTaskDelay( 1 / portTICK_PERIOD_MS );
4      frudp_listen();
5  }}
6  static void RTOSDiscotickTask( void * parameters ){
7    while( true ){
8      vTaskDelay( 1000 / portTICK_PERIOD_MS );
9      frudp_disco_tick();
10 }}
```

Listing 3.6 – Processos para decodificar mensagens recebidas e enviar mensagens de descoberta

O sistema operacional embarcado de tempo real fornece as ferramentas necessárias para implementação de um sistema de tempo real, mesmo assim fica a encargo do programador implementar o sistema de forma que o mesmo possa garantir as restrições de tempo. Como o foco desse trabalho é oferecer tempo Levando isso em consideração atribuiu-se aos processos que efetuam as funcionalidades básicas do FreeRTPS uma prioridade baixa, o que pode adicionar algum delay extra a essas tarefas quando existirem processos de maior prioridade executando em paralelo. Porém, como as mensagens de discover não necessitam de determinismo, apenas um intervalo regular para avisar sobre a sua existência, delays esporádicos não iram interferir no funcionamento do sistema.

Por fim, para integrar a pilha UDP/IP de forma eficiente com o sistema FreeRTPS + FreeRTOS, removeu-se toda a implementação UDP/IP do FreeRTPS até o hardware e a substituiu pelo FreeRTOS+UDP. Essa substituição oferecerá um melhor desempenho ao sistema proposto, já que essa implementação da pilha UDP é um complemento ao sistema operacional FreeRTOS.

O primeiro passo consiste em alterar o código da função responsável por adicionar as portas de comunicação do FreeRTPS, a rotina frudp_add_ucast_rx, no arquivo udp.c. Para isso adiciona-se a função FreeRTOS_setsockopt, do FreeRTOS+UDP, que habilita as portas desejadas para envio e recebimento de mensagens na pilha UDP/IP. Dessa forma o sistema se torna apto a enviar e receber mensagens através do protocolo RTPS.

A interface entre o FreeRTPS e a pilha UDP/IP acontece nas funções frudp_listen e frudp_tx, ambas no arquivo udp.c. A função frudp_listen efetua a leitura das mensagens RTPS recebidas pelo protocolo UDP e em seguida faz a chamada da função frudp_rx (arquivo udp.c), para que essa mensagem seja desempacotada e processada de acordo com as informações do cabeçalho do protocolo RTPS. A função frudp_listen original possui a forma apresentada no listing 3.7.

```
1  bool frudp_listen(const uint32_t max_usec){
2    // just busy-wait here for the requested amount of time
3    volatile uint32_t t_start = systime_usecs();
4    while (1){
5      enet_process_rx_ring();
6      volatile uint32_t t = systime_usecs();
7      if (t - t_start >= max_usec)
8        break;
9    }
10   return true;
11 }
```

Listing 3.7 – Rotina para decodificar mensagens recebidas do FreeRTPS original

Essa rotina recebe como parâmetro o intervalo de tempo, em microsegundos, fornecido pelo usuário para processar as mensagens que chegaram e foram armazenas no buffer circular da pilha UDP implementada. O processamento se faz através da função enet_process_rx_ring (enet.c). Durante esse tempo o programa principal não é executado. A função enet_process_rx_ring efetua todo o processamento do pacote Ethernet, desde a camada física ate a camada de aplicação, filtrando as mensagens por IP, MAC e portas.

A modificação da função frudp_listen consiste em pegar as mensagens, recebidas pela pilha UDP/IP do FreeRTOS+UDP, e em seguida enviá-las para a função frudp_rx, que possui o conjunto de regras para processar mensagens do protocolo RTPS. Para isso, inseriu-se um loop para analisar se as portas registradas para comunicação possuem novas mensagens e em seguida enviá-las para processamento. A nova rotina frudp_listen possui a estrutura apresentada no listing 3.8.

```
1  void frudp_listen( void ){
2    struct freertos_sockaddr xSourceAddress;
3    int32_t iReturned = 0;
4    for( int i = 0; i < g_enet_allowed_udp_ports_wpos; i++ ) {
5      do{
6        iReturned = FreeRTOS_recvfrom( xSocket[ i ], ucBuffer, 1600, 0, &
             ↪ xSourceAddress, ( uint32_t *)( sizeof( xSourceAddress ) ) );
7        if( iReturned > 0 ){
8          if( frudp_rx( ucBuffer, iReturned ) ){}
9        }
10     }while( iReturned > 0 );
11   }
12 }
```

Listing 3.8 – Rotina FreeRTPS modificada para decodificar mensagens recebidas

Já a rotina frudp_tx original recebia como parâmetro uma mensagem RTPS formatada, em função do tipo de informação que seria enviado através do protocolo, junto com o endereço de IP e porta de destino. Em seguida a mensagem era passada para função enet_send_udp_ucast (enet.c), que era responsável por inseri-la dentro de um pacote UDP/IP e ao fim em um pacote Ethernet. Depois de empacotado em um frame Ethernet, a mensagem era submetida para envio através da rede. A função frudp_tx possuía a forma apresentada no listing 3.9.

```
1  bool frudp_tx(const uint32_t dst_addr,const uint16_t dst_port,const uint8_t
     ↪ *tx_data,const uint16_t tx_len){
2    if ((dst_addr & 0xe0000000) == 0xe0000000){
3      // ipv4 multicast
```

```
4       enet_send_udp_mcast(dst_addr, dst_port, tx_data, tx_len);
5     }else{
6       uint8_t dst_mac[6] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff }; // todo:
            ↪ not this
7       enet_send_udp_ucast(dst_mac,dst_addr, dst_port,FRUDP_IP4_ADDR, dst_port
            ↪ ,tx_data, tx_len);
8     }
9     return true;
10  }
```

Listing 3.9 – Rotina para enviar mensagens no FreeRTPS original

Para inserir a pilha UDP/IP do FreeRTOS na rotina frudp_tx, necessitava-se indicar a porta UDP que estava enviando a mensagem. A versão original definia a porta de fonte da mensagem como a mesma de destino. Para melhorar a compatibilidade dessa função na continuação do desenvolvimento do FreeRTPS, optou por adicionar o campo 'srcPort' aos seus parâmetros. A rotina modificada ficou com a estrutura apresentada no listing 3.10:

```
1   bool frudp_tx( const uint32_t dstAddr, const uint16_t dstPort, const
        ↪ uint16_t srcPort, const uint8_t *data, const uint16_t length ){
2     struct freertos_sockaddr xDestinationAddress;
3     int32_t iReturned;
4     xDestinationAddress.sin_addr = FreeRTOS_htonl( dstAddr );
5     xDestinationAddress.sin_port = FreeRTOS_htons( dstPort );
6     for( int i = 0; i < g_enet_allowed_udp_ports_wpos; i++ ){
7       if( g_enet_allowed_udp_ports[ i ] == srcPort ){
8         iReturned = FreeRTOS_sendto( xSocket[ i ], data, length, 0, &
              ↪ xDestinationAddress, sizeof( xDestinationAddress ) );
9         if( iReturned == length ){
10          return true;
11        }
12      }
13    }
14    return false;
15  }
```

Listing 3.10 – Rotina para enviar mensagens modificada

Como pode-se observar, inseriu-se um loop para verificar qual a porta de saída será utilizada para enviar a mensagem. Uma vez identificada, seleciona-se o socket responsável por essa porta e chama-se então a função FreeRTOS_sendto da pilha UDP/IP do FreeRTOS, ela é responsável por empacotar a mensagem no protocolo UDP e despachar a mensagem na rede.

A última modificação de código consiste da edição da rotina que adiciona as portas utilizadas pelo protocolo FreeRTPS, no arquivo enet.c. A função original utilizava a lógica apresentada no listing 3.11.

```
1  bool enet_allow_udp_port(const uint16_t port){
2    // make sure we aren't already listening to this port
3    printf("enet_allow_udp_port(%d)\r\n", port);
4    for (int i = 0; i < ENET_MAX_ALLOWED_UDP_PORTS; i++)
5      if (g_enet_allowed_udp_ports[i] == port)
6        return true; // it's already allowed
7    if (g_enet_allowed_udp_ports_wpos >= ENET_MAX_ALLOWED_UDP_PORTS)
8      return false; // sorry, no room, have a nice day
9    g_enet_allowed_udp_ports[g_enet_allowed_udp_ports_wpos++] = port;
10   return true;
11 }
```

Listing 3.11 – Função do FreeRTPS original para habilitar portas UDP

Na versão modificada a habilitação das portas ficou por conta da rotina frudp_add_ucast_rx, arquivo udp.c. Na rotina, a função FreeRTOS_bind vincula um socket a uma porta local e assim habilita a recepção e envio de mensagens através da combinação do IP local junto a porta selecionada na rede. No listing 3.12 é apresentado a rotina modificada.

```
1  bool frudp_add_ucast_rx(const uint16_t port){
2    TickType_t xReceiveTimeout_ms = 0;
3    for( int i = 0; i < g_enet_allowed_udp_ports_wpos; i++ ){
4      if( g_enet_allowed_udp_ports[ i ] == port )
5        return true;
6    }
7    if( g_enet_allowed_udp_ports_wpos >= ENET_MAX_ALLOWED_UDP_PORTS )
8      return false;
9    g_enet_allowed_udp_ports[ g_enet_allowed_udp_ports_wpos ] = port;
10   xSocket[ g_enet_allowed_udp_ports_wpos ] = FreeRTOS_socket(
       ↪ FREERTOS_AF_INET, FREERTOS_SOCK_DGRAM, FREERTOS_IPPROTO_UDP );
11   FreeRTOS_setsockopt( xSocket[ g_enet_allowed_udp_ports_wpos ], 0,
       ↪ FREERTOS_SO_RCVTIMEO, &xReceiveTimeout_ms, 0 );
12
13   if( xSocket[ g_enet_allowed_udp_ports_wpos ] != FREERTOS_INVALID_SOCKET )
       ↪ {
14     xBindAddress[ g_enet_allowed_udp_ports_wpos ].sin_port = FreeRTOS_htons
         ↪ ( port );
15     if( FreeRTOS_bind( xSocket[ g_enet_allowed_udp_ports_wpos ], &
         ↪ xBindAddress[ g_enet_allowed_udp_ports_wpos ], sizeof( &
         ↪ xBindAddress[ g_enet_allowed_udp_ports_wpos ] ) ) != 0 )
16       return false;
17   }
```

```
18    else
19      return false;
20    g_enet_allowed_udp_ports_wpos++;
21    return true;
22  }
```

Listing 3.12 – Habilitação das portas UDP no sistema modificado

Uma vez que o sistema passou a utilizar a pilha UDP do sistema FreeRTOS, removeu-se o arquivo enet.c, que era responsável por essa executar as tarefas inerentes a pilha UDP/IP. Removeu-se também o arquivo arm_trap.c, que continha a rotina para interrupções não registradas, deixando agora as interrupções não registradas por conta do arquivo especifico do microcontrolador. Excluiu-se também o arquivo bswap.c, deixando agora as funções de ordem do byte por conta das funções do FreeRTOS. Os arquivos delay.c e USB.c foram removidos, já que não são utilizados pelo sistema. E por fim, as rotinas do arquivo system.c foram transferidas para o arquivo udp.c.

A implementação do arquivo udp.c era dividida em dois níveis, em um nível estavam as funções de processamento gerais do protocolo FreeRTPS e em outro nível estavam as funções de recebimento e envio das mensagens RTPS. O segundo nível era dividido em dois arquivos udp.c, um responsável pela interface para sistemas POSIX e outro para sistemas bare metal. Com o intuito de deixar o sistema compatível apenas com microcontroladores, removeu-se a implementação POSIX e transferiu-se as rotinas do arquivo udp.c da pasta metal_common para o arquivo geral udp.c.

As modificações foram efetuadas com o intuito de deixar o sistema mais simples e robusto, na expectativa de deixar a portabilidade para outros sistemas e IDEs de microcontroladores mais simples.

3.2.1 Implementação de mensagens multicast

O FreeRTPS efetua o envio de mensagens de descoberta na rede utilizando endereço de IP multicast, porém observou-se que na versão 9.0.0 do FreeRTOS+UDP não possui uma implementação para efetuar tal ação. Analisando o código UDP do sistema FreeRTOS, verificou-se que é possível atribuir tal recurso adicionando algumas linhas de código no arquivo FreeRTOS_UDP_IP.c. As modificações foram efetuas nas funções: prvGetARPCacheEntry (linha 792), eConsiderFrameForProcessing (linha 1305) e prvProcessIPPacket (linha 1478).

A rotina prvGetARPCacheEntry (FreeRTOS_UDP_IP.c - linha 792) sofreu a seguinte modificação: adicionou-se um condicional complementar ao “if(*pulIPAddress == ipBROADCAST_IP_ADDRESS)”, listing 3.13.

```
1 else if( *pulIPAddress == ipMULTICAST_IP_ADDRESS ){
2     memcpy( ( void * ) pxMACAddress, &xBroadcastMACAddress, sizeof(
      ↪ xMACAddress_t ) );
3     eReturn = eARPCacheHit;
4 }
```

Listing 3.13 – Modificação do código do FreeRTOS para habilitar multicast

Essa inserção de código se faz necessária para que o dispositivo possa enviar mensagens multicast na rede. Verificando que o IP pertence a um domínio multicast, a função retorna o valor eARPCacheHit e sinaliza que pacote pode ser enviado. Antes da modificação o IP multicast não era identificado como um IP válido, uma vez que só era avaliado se o destino era um IP unicast ou broadcast, e assim retornava o valor eCantSendPacket.

Adicionou-se também um condicional complementar a rotina eConsiderFrameForProcessing (FreeRTOS_UDP_IP.c - linha 1305), listing 3.14.

```
1 else if( pxEthernetHeader->xDestinationAddress.ucBytes[ 2 ] == 0x5E ){
2     eReturn = eProcessBuffer;
3 }
```

Listing 3.14 – Modificação do código do FreeRTOS para habilitar multicast

Essa rotina é responsável por verificar se um pacote recebido deve ser enviado para a pilha UDP do FreeRTOS, assim que a mensagem é recebida pelo hardware. Para isso, a rotina analisa se o endereço MAC de destino é o MAC do dispositivo ou um MAC broadcast, todos seis octetos com valor 0xFF (0xFFFFFFFFFFFF). Sabendo que todo endereço MAC multicast possui o terceiro octeto com o valor 0x5E, adicionou-se um condicional à verificação dos MACs aceitos fazendo com que o processamento do pacote em questão seja efetuado.

A rotina prvProcessIPPacket (FreeRTOS_UDP_IP.c - linha 1478) também recebeu mais uma comparação em seu condicional, verifica-se se o pacote recebido é destinado a esse dispositivo. O condicional ficou na forma apresentada no listing 3.15.

```
1 if( ( pxIPHeader->ulDestinationIPAddress == *ipLOCAL_IP_ADDRESS_POINTER )
    ↪ || ( pxIPHeader->ulDestinationIPAddress == ipBROADCAST_IP_ADDRESS )
    ↪ || ( *ipLOCAL_IP_ADDRESS_POINTER == 0 ) || ( pxIPHeader->
    ↪ ulDestinationIPAddress == ipMULTICAST_IP_ADDRESS ) )
```

Listing 3.15 – Modificação do código do FreeRTOS para habilitar multicast

Essa rotina efetua o processamento inicial do pacote IP, analisando o tipo da mensagem e a enviando para o processamento correto. Os protocolos que são aceitos nesse nível, no FreeRTOS+UDP, são: ICMP e UDP. Adicionando a condição "pxIPHeader->ulDestinationIPAddress == ipMULTICAST_IP_ADDRESS", permite que mensagens com IP de destino multicast sejam processadas pelo sistema.

Como as alterações acima utilizam a variável ipMULTICAST_IP_ADDRESS para determinar se o endereço de IP pertence a classe multicast, adicionou-se então a definição dessa variável ao arquivo FreeRTOS_IP_Private.h, do sistema FreeRTOS+UDP, listing 3.16.

```
1 #define ipMULTICAST_IP_ADDRESS 0x0100FFEFUL
```

Listing 3.16 – Modificação do código do FreeRTOS para habilitar multicast

Como o FreeRTOS+UDP utiliza o arquivo FreeRTOS_IP_Private.h para a definição do IP de broadcast, optou-se por deixar a definição do IP de multcast nesse mesmo arquivo, mantendo assim o padrão utilizado pelo FreeRTOS.

Deve-se observar que efetuando essas modificações, todos os pacotes multicast que transitam na rede e destinados a uma porta que esteja aberta no dispositivo serão aceitos e processados pela pilha UDP do sistema, mas mensagens não reconhecidas na decodificação do formato RTPS serão descartadas logo no início.

4 EXPERIMENTOS E RESULTADOS

A fim de avaliar a funcionalidade e desempenho do sistema, alguns testes foram elaborados para validação dos requisitos de tempo real e desempenho do sistema. Para validação do tempo real, elaborou-se um teste de controle de tensão de um capacitor utilizando um controlador PID. Já para avaliar o desempenho do sistema, verificou-se a quantidade de mensagens que o sistema é capaz de enviar, enquanto executando o um controle PID.

4.1 Hardware utilizado

O sistema original do FreeRTPS foi desenvolvido sobre a placa de desenvolvimento STM32F4DISCOVERY, figura 21. Uma placa que já possui todo circuito necessário para operação do microcontrolador STM32F407VGT6, um circuito incluso para programar o microcontrolador, alguns recursos (botões, leds, acelerômetro, saída de áudio e conectores) (STM32F4DISCOVERY, 2017).

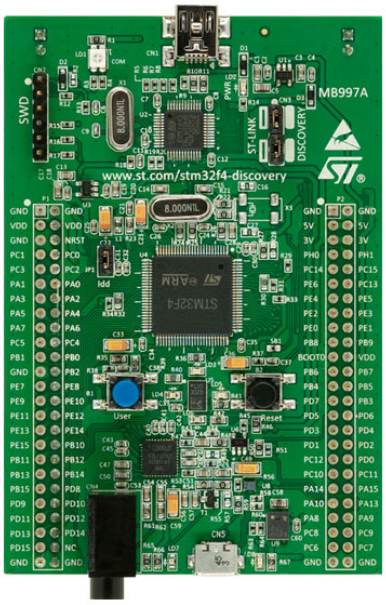

Figura 21 – STM32F4DISCOVERY

Fonte: (STM32F4DISCOVERY, 2017)

A memória flash desse chip é de 1 Mbyte com 192 Kbytes de memória RAM. Sua frequência máxima de *clock* pode alcançar valores de 168 Mhz (STM32F407VG..., 2017). Esse chip oferece ainda uma gama de periféricos: conversor analógico digital, conversor digital analógico, diversos interfaces de comunicação (I2C (*Inter-Integrated Circuit*),

SPI (*Serial Peripheral Interface*), UART (*Universal asynchronous receiver transmitter*), Ethernet, CAN (*Controller Area Network*), USB (*Universal Serial Bus*)), timers além de portas de entrada e saída digitais.

Como o FreeRTPS comunica-se através de mensagens UDP, será necessário fazer uso do periférico Ethernet do microntrolador. Para acesso a recursos extras na placa de desenvolvimento, a ST disponibiliza uma outra placa já pronta e projetada para ser integrada a STM32F4DISCOVERY, figura 22. A placa extra que será utilizada nesse trabalho é a *base board*, ela possui: um *slot* para cartão de memória microSD, conector RJ45 10/100 Ethernet, conectores para os protocolos de comunicação e conectores para as placas de expansão câmera e LCD (*liquid crystal display*), do kit DISCOVERY (STMICROELECTRONICS. . . , 2017)

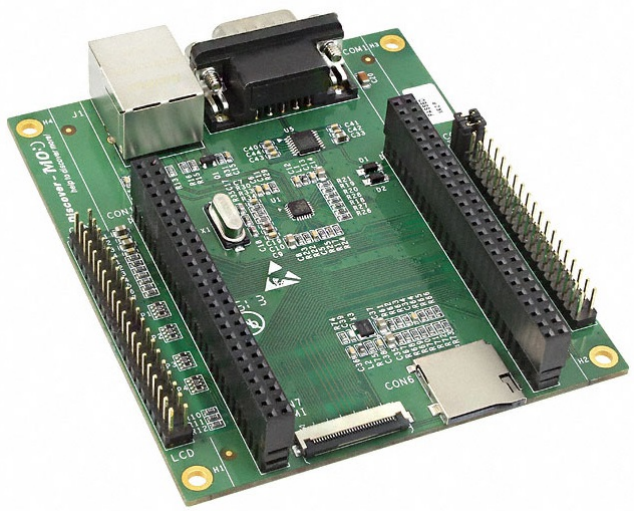

Figura 22 – STM32F4 *Discovery Base Board*

Fonte: (STMICROELECTRONICS. . . , 2017)

Para verificar a portabilidade do sistema proposto, efetuou-se também a implementação do sistema em uma placa de desenvolvimento da *Texas Instruments*, a *Stellaris LM3S6965 Evaluation Board*, figura 23. A placa utilizada possui um microcontrolador ARM (*Advanced RISC Machine*) Cortex M3 e alguns dispositivos periféricos, tais como: um *display* LED, botões, LEDs, cartão de memória, *buzzer* e um conector Ethernet RJ45 (STELLARIS®. . . , 2017).

O microcontrolador presente nessa placa pode operar a frequência máxima de 50Mhz, possui 256kB de flash e 64kB de RAM. Possui ainda um periférico de Ethernet 10/100, SPI, I2C, UART, ADC, *timers* e 42 entradas e saídas de uso geral. Pode ser programado utilizando a biblioteca *StellaresWare* fornecida pela *Texas Instruments*, que fornece rotinas para configuração e utilização de todos os periféricos em um nível mais alto (STELLARIS. . . , 2017).

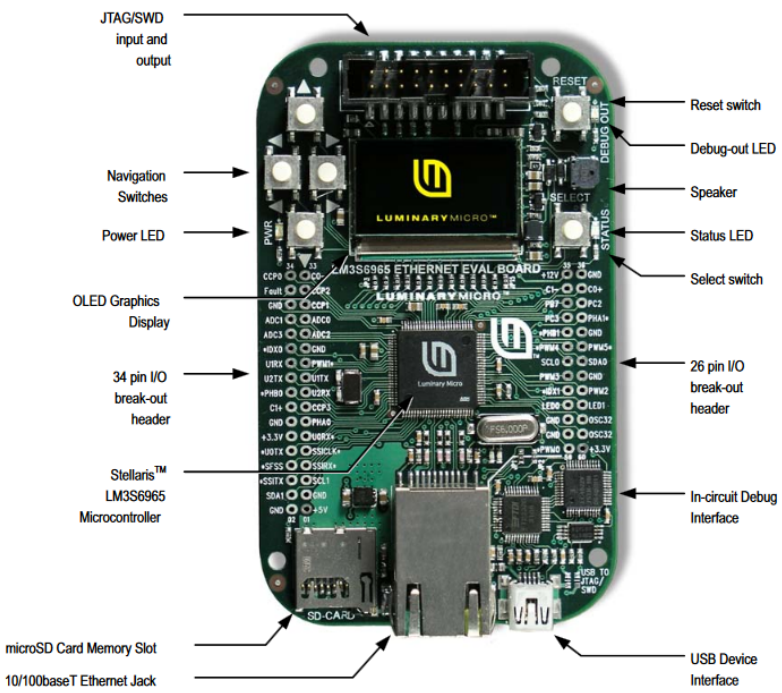

Figura 23 – *Stellaris LM3S6965 Evaluation Board*

Fonte: (STELLARIS®..., 2017)

4.2 Teste de tempo real

Esse é o principal teste desse trabalho, uma vez que o foco é fornecer recursos para possibilitar tempo real em aplicações embarcadas em ROS2. Para efetuar a validação do sistema uma malha de controle fechada, utilizando um controlador PID, foi implementada com uma frequência de 1000Hz. A validação se da através da comparação do saída do sistema físico com uma simulação de um controlador PID contínuo no domínio S e também discretizado, utilizando o método de aproximação bilinear (Tustin), com os mesmo parâmetros utilizados no teste físico.

O controle escolhido para validação foi um controle de tensão de um circuito RC, escolheu-se esse sistema por possuir poucos componentes para montagem e rápida prototipagem. Os valores escolhidos foram um resistor de 10 KΩ e um capacitor de 1uF, a conexão dos componentes possui a seguinte estrutura, figura 24. Com os valores utilizados, espera=se obter um sistema com constante de tempo RC de 10ms.

A malha de controle é efetuada por um processo que possui a maior prioridade do sistema, e assim espera-se sempre obter a preferência do escalonador de processos. O processo de controle é um *loop* infinito, com frequência de 1000Hz, que efetua a leitura do valor atual da tensão de saída do sistema e em seguida calcula o valor de saída do PWM (*Pulse Width Modulation*), para que o valor de tensão desejado seja alcançado. A frequência utilizada no PWM nesse teste foi de 20000Hz.

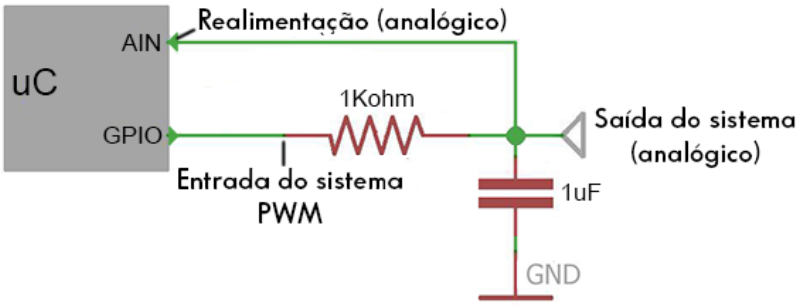

Figura 24 – Circuito utilizado para efetuar o teste do controlador PID

Criaram-se mais dois processos junto ao processo que computa o calculo do controlador PID, uma com a tarefa de publicar mensagens com o valor atual de saída do sistema controlado, a uma frequência de 100Hz, e outro para o assinante do tópico '*desiredVolt*', que recebe os valores desejados para a saída do sistema controlado. Os valores desejados de saída do sistema são publicados por outra placa conectada a mesma rede, esse segundo dispositivo publica os valores no tópico '*desiredVolt*'. Como o sistema possui um limite de tensão de saída de 0 V a 3 V, e os valores são publicados na rede utilizando o tipo 'uInt32', o valor de tensão foi convertido para notação digital de 12 bits com referência de 3 V. Escolheu-se essa conversão pelo fato do próprio hardware operar com um conversor digital de 12 bits na aquisição da tensão de saída do sistema.

A implementação completa do código está presente no apêndice B. A seguir a parte do código que implementam as tarefas do controlador PID e do assinante do valor desejado, listing 4.1.

```
1  //Processo do controle PID
2  static void ctrlTask( void *parameters ){
3    //Variável que armazena o tempo do último bloqueio do processo
4    TickType_t xLastWakeTime = xTaskGetTickCount();
5    //Variáveis do PID
6    double y0, y1, y2, e0, e1, e2;
7    const double kp = KP;
8    const double kd = KD * 2.0 / PIDsampleTime;
9    const double ki = KI * PIDsampleTime / 2.0;
10   //Inicialização das variáveis do PID
11   e0 = e1 = e2 = y0 = y1 = y2 = 0.0;
12   //PID loop
13   while( true ){
14 #if usePID == 1
15     //Aquisitando o valor atual do ADC
16     currentADCValue = getADCValue();
17     //Efetuando os cálculos do PID
18     y2 = y1;
```

```
19      y1 = y0;
20      e2 = e1;
21      e1 = e0;
22      e0 = setPoint - ( currentADCValue / ( ( 1 << ADC_BITS ) - 1 ) );
23      //Aproximação da equação do PID utilizando o modelo bilinear
24      y0 = y2 + kp * ( e0 - e2 ) + ki * ( e0 + 2.0 * e1 + e2 ) + kd * ( e0 -
            ↪ 2.0 * e1 + e2 );
25  #else
26      //teste em malha aberta, o valor desejado na saída é aplicado direto a
            ↪ entrada
27      y0 = setPoint;
28  #endif
29      //Saturando o valor máximo em termos de volts
30      if( y0 > PWM_MAX_VOLTAGE )
31        y0 = PWM_MAX_VOLTAGE;
32      //Saturando o valor mínimo em termos de volts
33      if( y0 < PWM_MIN_VOLTAGE )
34        y0 = PWM_MIN_VOLTAGE;
35      //Aplicando o valos de saída calculado ao PWM
36      //Convertendo a tensão em valor digital de 16bits
37      TIM4->CCR4 = ( uint16_t )( y0 * ( ( 1 << PWM_BITS ) - 1 ) /
            ↪ PWM_MAX_VOLTAGE );
38      //Aplicando o período do processo do PID
39      vTaskDelayUntil( &xLastWakeTime, ( PIDsampleTime * 1000.0 ) /
            ↪ portTICK_PERIOD_MS );
40  }}
41
42  //Processo para receber as messagens do tópico desiredVoltage
43  static void desiredVoltSubTask( void *parameters ){
44    //Fila para armazenar as mensagens recebidas no tópico
45    QueueHandle_t sQueue = ( QueueHandle_t )parameters;
46    //Variáveis para manipular o dado recebido
47    uint8_t  msg[ 128 ] = { 0 };
48    while( true ){
49      //portMAX_DELAY indica que xQueueReceive irá colocar o processo em
            ↪ inativo até haver algum dado na fila de dados do tópico
50      if( xQueueReceive( sQueue, msg, portMAX_DELAY ) == pdPASS ){
51        //Pisca o LED para indicar que um dado foi recebido
52        led_toggle();
53        //Recebe o dado em um valor digital de 12 bits
54        setPoint = *( ( uint32_t* )msg );
55        //Converte o valor digital de 12 bits para volts com referência de 3V
56        setPoint = PWM_MAX_VOLTAGE * setPoint / ( ( 1 << ADC_BITS ) - 1 );
57  }}}
```

Listing 4.1 – Implementação do processo do PID e do assinante do valor desejado de controle

O segundo dispositivo conectado a rede, que efetua a publicação dos valores desejados de tensão do sistema, possui uma tarefa periódica que alterna o tensão desejada entre os valores 1.2 V e 1.8 V, com um período de 100 ms. Como descrito anteriormente, os valores analógicos de tensão são convertidos em valores digitais com um referência de 3 V e representação de 12 bits. Logo, os valores que são realmente publicados são de: 1638, para o valor de 1.2 V, e 2457, para o valor de 1.8 V. O código do processo que efetua a publicação do valor de tensão desejado no segundo dispositivo foi implementado da seguinte maneira, listing 4.2.

```
1  //Uint32 processo publicador das tensões desejadas
2  static void pubTask( void *parameters )
3  {
4    //Variáveis para o publicador
5    uint8_t cdr[ 20 ] = { 0 };
6    int cdr_len;
7    struct std_msgs__uint32 digital12bitsVoltage;
8    double desiredVolt;
9    while( true )
10   {
11     //Valor de tensão desejado
12     desiredVolt = 1.2;
13     //Convertendo a tensão de 0V-3V para o valor digital de 12 bits
14     digital12bitsVoltage.data = ( uint32_t )( desiredVolt * ( ( 1 <<
         ↪ ADC_12_BITS ) - 1 ) / ADC_REF_VOLTAGE );
15     //Pisca o led a cada dado publicado
16     led_toggle();
17     //Serialização do dado para envio
18     cdr_len = serialize_std_msgs__uint32( &digital12bitsVoltage, cdr,
         ↪ sizeof( cdr ) );
19     //Publicando o dado
20     freertps_publish( pub, cdr, cdr_len );
21     //Aguarda um intervalo de 100 ms até a próxima publicação
22     vTaskDelayUntil( &xLastWakeTime, 100 / portTICK_PERIOD_MS );
23
24     //Valor de tensão desejado
25     desiredVolt = 1.8;
26     //Convertendo a tensão de 0V-3V para o valor digital de 12 bits
27     digital12bitsVoltage.data = ( uint32_t )( desiredVolt * ( ( 1 <<
         ↪ ADC_12_BITS ) - 1 ) / ADC_REF_VOLTAGE );
28     //Pisca o led a cada dado publicado
29     led_toggle();
30     //Serialização do dado para envio
31     cdr_len = serialize_std_msgs__uint32( &digital12bitsVoltage, cdr,
         ↪ sizeof( cdr ) );
32     //Publicando o dado
```

```
33      freertps_publish( pub, cdr, cdr_len );
34      //Aguarda um intervalo de 100 ms até a próxima publicação
35      vTaskDelayUntil( &xLastWakeTime, 100 / portTICK_PERIOD_MS );
36    }
37  }
```

Listing 4.2 – Código do publicador das tensões desejadas de saída

O código completo pode ser encontrado no apêndice B.

O primeiro teste consiste na obtenção dos sinais utilizando um controle de malha aberta, ou seja, o sistema simplesmente atribui ao PWM o valor referente a tensão media desejada. Como os pinos digitais da STMDISCOVERY possuem nível lógico baixo em 0V e nível lógico alto em 3V, tem-se que a tensão média de saída para 0% de *duty cycle* do PWM é 0V e para 100% de *duty cycle* do PWM a tensão média é 3V. Assim, para os valores de 1.2 V e 1.8 V foram aplicados 40% e 60% respectivamente. Como essa placa de desenvolvimento possui um PWM de 16bits, os valores digitais aplicados foram de 26214 (40%) e 39321 (60%).

Com o auxilio de um osciloscópio para verificar o nível de tensão do capacitor durante a operação do sistema, inicia-se a execução dos códigos das placas com o controle em malha aberta e obtêm-se a forma de onda apresentada na figura 25.

Percebe-se a curva típica de carregamento de um capacitor, delineada por uma exponencial com constante de tempo dependente do circuito RC. No caso do experimento, para os valores de resistência de 10 KΩ e capacitância de 10uF, obtêm-se a constante de tempo de 10ms. Percebe-se que o valor acomoda entre 30ms à 40ms após a mudança do novo valor de saída do PWM, coerente com a teoria de controle.

Como o foco do projeto não era obter um controlador PID bem ajustado, mas sim validar o tempo real, atribuiu-se os ganhos proporcional, integrativo e derivativo com valores para se obter uma boa resposta de saída, para comparação com o sistema em malha aberta e a simulação de um controlador equivalente não discretizado. Gravando a placa de controle com o código PID, obtêm-se a forma de onda da figura 26:

Observa-se que enquanto o controle em malha aberta possui uma curva de subida e descida mais suave limitada pela constante de tempo do circuito, o controle PID oferece uma forma de onda com resposta mais rápida. Isso se deve ao fato do controle PID aplicar um *duty cycle* de 100% ao PWM no inicio do degrau aplicado e ir ajustando essa porcentagem para manter o valor de tensão desejado de 1.8V, diferente do controle em malha aberta que fornece constantemente 60% de *duty cycle* do PWM, porcentagem referente a tensão média de 1.8V para a placa utilizada. O mesmo acontece para o nível de tensão mais baixo, o controlador PID aplica 0% de *duty cycle* do PWM no inicio, forçando a saída a ter um abaixamento de tensão mais rápido, e depois vai ajustando o

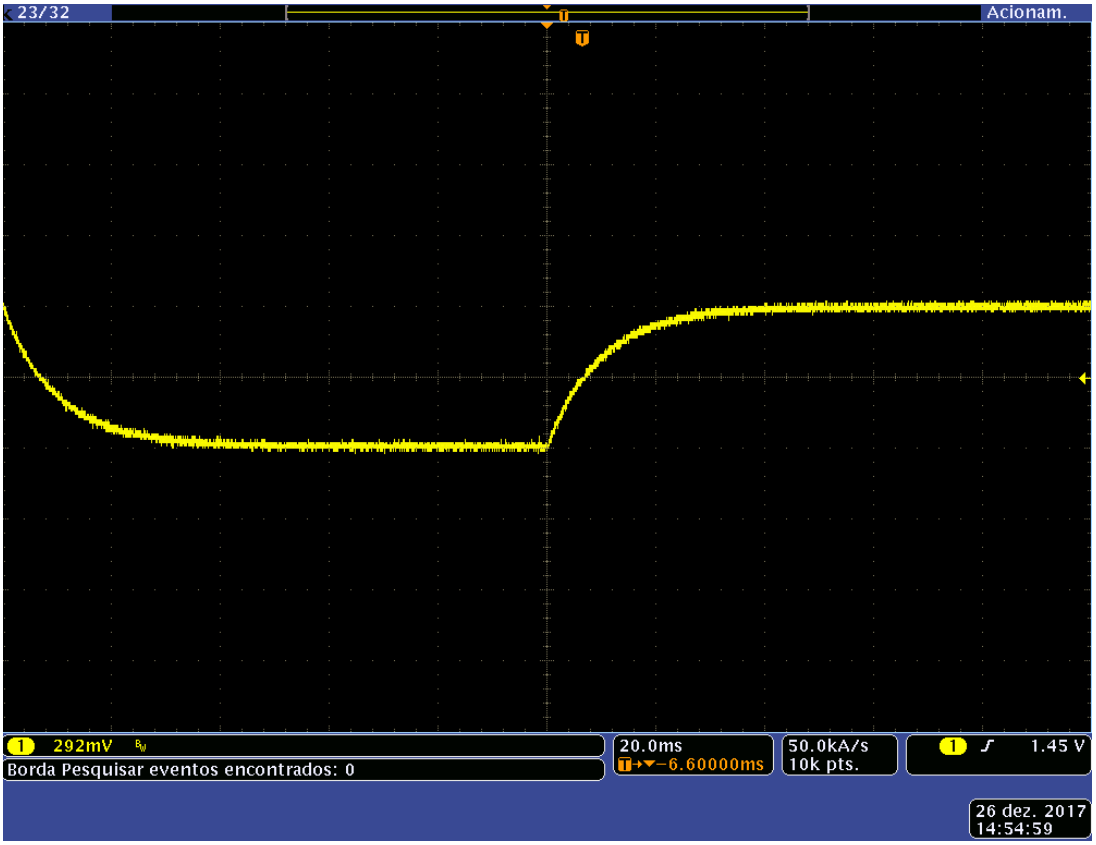

Figura 25 – Controle de tensão de um circuito RC em malha aberta

valor do *duty cycle* para se manter a saída em 1.2V, enquanto o controle em malha aberta mantém 40% de *duty cycle* do PWM, equivalente a tensão media de 1.2V, durante todo o tempo que a tensão de saída desejada for 1.2V.

Por fim, efetuou-se a simulação de um controlador PID atuando em um sistema que representa matematicamente o circuito RC utilizado. O programa utilizado para efetuar a simulação foi o SciLab, onde aplicaram-se as limitações do circuito físico: valor de saturação inferior e superior, e também os parâmetros utilizados no controlador, como: frequência de amostragem, resolução do PWM, do ADC e ganhos do controlador. O modelo elaborado para simulação pode ser observado na figura 27.

Executando a simulação, observa-se que para os mesmos valores de tensão desejados e parâmetros utilizados no teste físico, a método utilizado no modelo discreto apresenta uma curva bem próxima do dele, figura 28.

Sobrepondo agora a curva obtida no teste físico, junto a curva obtida na simulação, figura 29,observa-se que o sistema com controle digital forneceu uma resposta estável e próxima do seu modelo simulado, seguindo as tendencias do sinal simulado, o que mostra que os requisitos de tempo do controlador estão sendo atendidos.

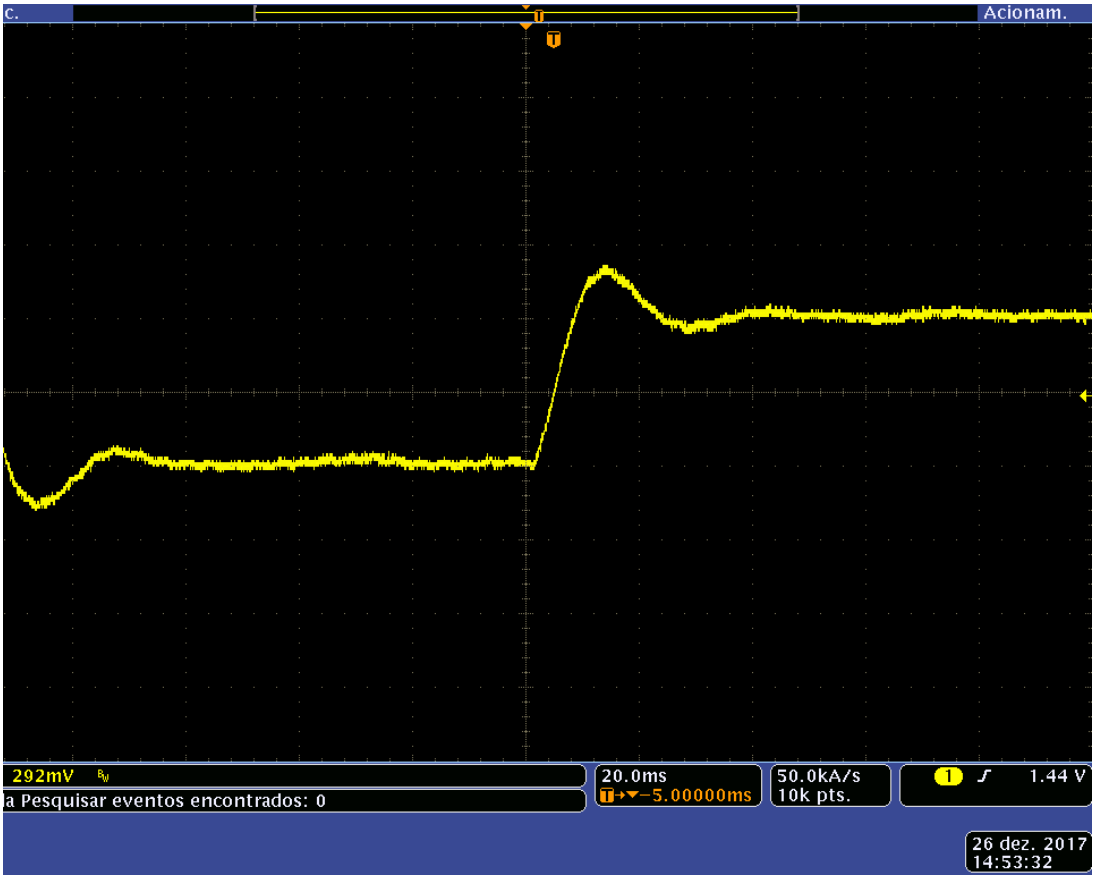

Figura 26 – Controle de tensão de um circuito RC utilizando controlador PID

4.3 Teste de desempenho em rede

Para verificar o teste de desempenho do sistema, elaborou-se uma estrutura de rede com um computador Dell Vostro 14-5480, uma placa de desenvolvimento STMDISCOVERY e uma placa de desenvolvimento *Stellaris LM3S6965 Evaluation Board*. Todos conectados por cabos Ethernet a um switch TP-LINK modelo TL-WR741ND, através de suas portas LAN 10/100Mbps.

Esse teste possui como foco observar a taxa de publicação em um tópico que as placas são capazes de alcançar, enquanto executando o sistema FreeRTOS+FreeRTPS com o processo de um controlador PID ajustando a tensão de um capacitor.

Para isso, definiu-se a período do processo responsável pelo controlador PID com um valor de 1ms, o que resulta em uma frequência de 1000Hz. Executando em paralelo ao processo do controlador PID, implementou-se um processo para ficar publicando mensagens de texto de 5 bytes e sem *delay* entre sucessivos envios.

Para a placa STMDISCOVERY foi definido que a publicação seria no tópico 'stm-topic', enquanto que para a *Stellaris Eval Board* seria no tópico 'titopic'. Executou-se então duas aplicações no computador e cada uma assinando um dos tópicos.

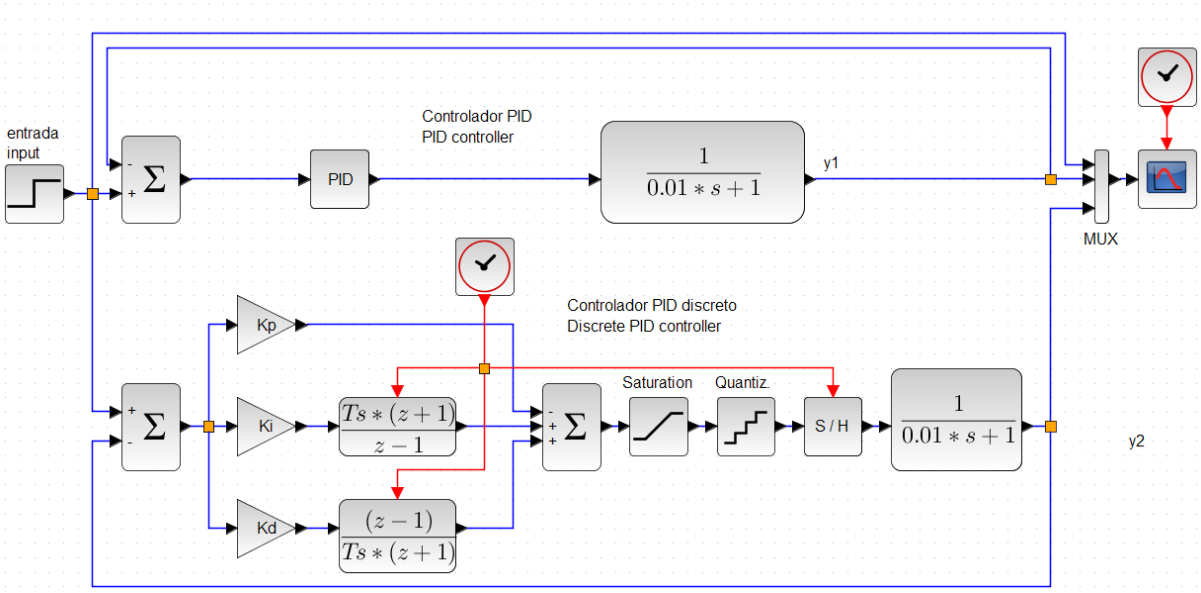

Figura 27 – Modelo de blocos de um controlador PID equivalente ao sistema testado

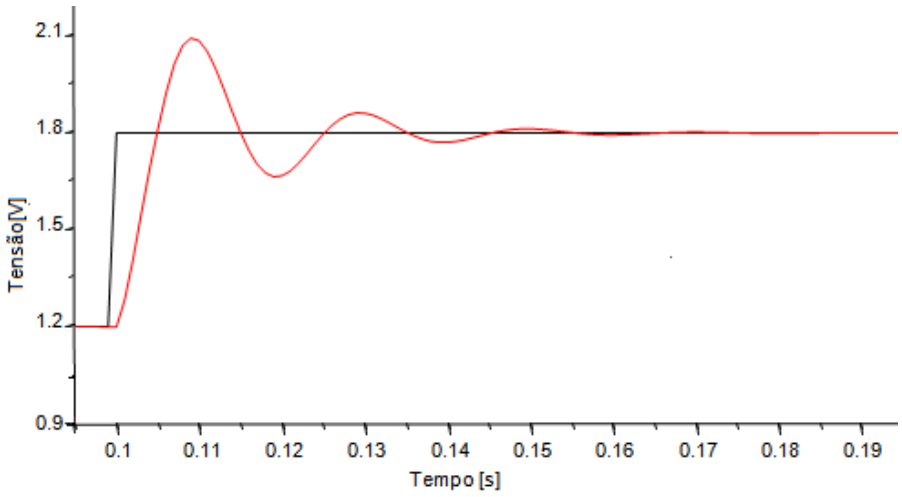

Figura 28 – Resultado obtido na simulação utilizando os parâmetros do teste real

Como o programa *Wireshark* oferece a visualização dos pacotes que chegam na porta Ethernet, com uma riqueza de informações, optou-se por utilizá-lo para averiguar a quantidade de mensagens que foram enviadas pelas placas de desenvolvimento. Fez-se uso do seguinte filtro para selecionar apenas as mensagens de interesse para contagem, listing 4.3:

```
1 rtps and ip.src==xxx.xxx.xxx.xxx and frame.len==112 and
2 udp.port==7411 and frame.time_relative>=8 and frame.time_relative<=9
```

Listing 4.3 – Filtrando apenas as mensagens de interesse aquisitadas pelo Wireshark

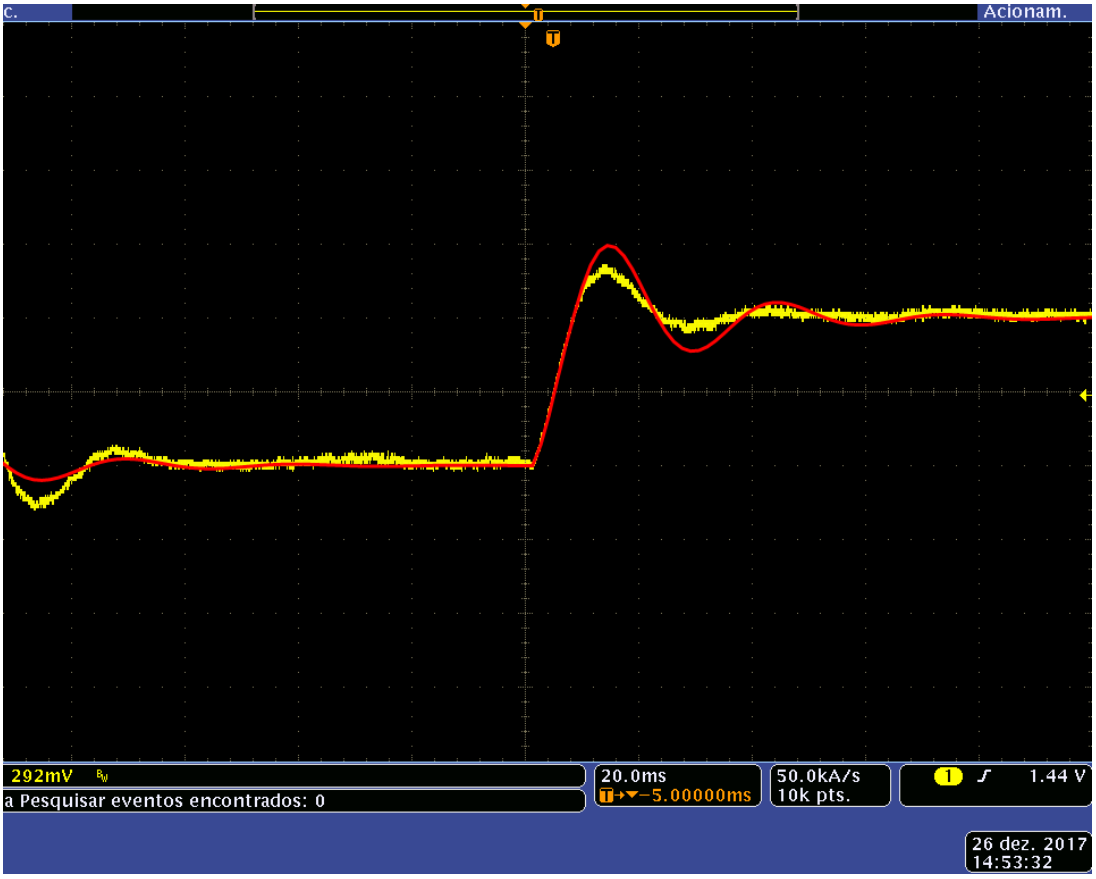

Figura 29 – Comparação do sinal simulado ao sinal obtido no teste real

Esse seleciona os pacotes que foram enviados pelo endereço de IP desejado (no caso do teste utilizou-se os endereços de IP atribuídos a cada uma das placas), o tamanho do frame Ethernet que se deseja filtrar (o frame de uma publicação do tipo *string* e com 5 caracteres do FreeRTPS possui 112 bytes), a porta fonte a ser filtrada (o FreeRTPS utiliza a porta 7411 para publicação) e por fim o intervalo de tempo utilizando o termo frame.time_relative, que nesse teste será maior que X segundos e menor que X + 1 segundos. O programa possui a seguinte interface, figura 30, onde a etiqueta *Displayed*, canto inferior direito, apresenta a quantidade de pacotes restantes após utilizar o filtro.

Inicialmente, filtram-se apenas as mensagens que foram recebidas pelo STMDISCOVERY e em seguida verifica-se a quantidade de mensagens que foram recebidas no intervalo de 1 segundo, tabela 3.

Efetuando o mesmo processo para a *Stellaris Eval Board*, obteve-se o seguinte resultado, tabela 4.

Como pode-se observar, a placa STMDISCOVERY obteve uma capacidade média de 25504 mensagens por segundo enquanto a *Stellaris Eval Board* alcançou a marca de 2739 mensagens por segundo. Essa defasagem de valores deve-se a diferença dos *clocks* internos dos microcontroladores. Enquanto utilizou-se a STMDISCOVERY à 168 Mhz,

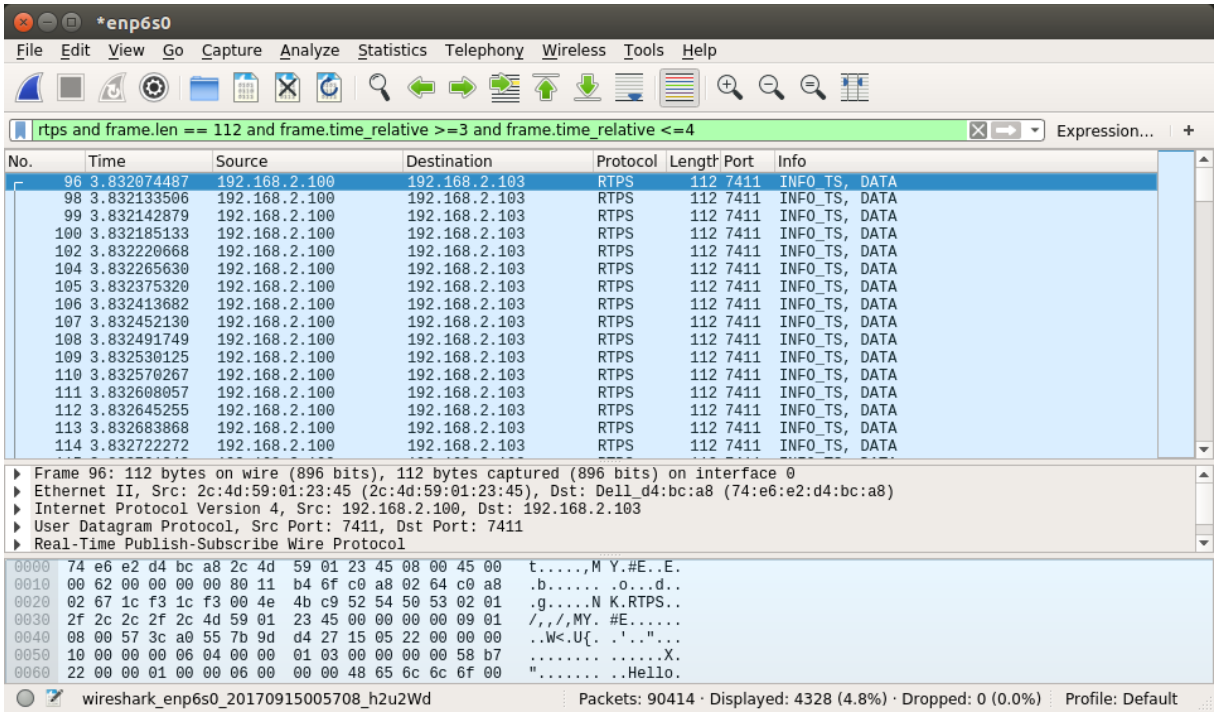

Figura 30 – Interface Wireshark

Intervalo de tempo	Quantidade
Entre 6 e 7 segundos	25489 mensagens enviadas
Entre 7 e 8 segundos	25493 mensagens enviadas
Entre 8 e 9 segundos	25516 mensagens enviadas
Entre 9 e 10 segundos	25519 mensagens enviadas
Entre 10 e 11 segundos	25502 mensagens enviadas
Média	25504 mensagens enviadas

Tabela 3 – Quantidade máxima de mensagens enviadas pela STMDISCOVERY para o primeiro teste de desempenho

frequência máxima desse chip, configurou-se a *Stellaris Eval Board* para trabalhar com um *clock* na metade de sua frequência máxima, 25 Mhz.

A dimensão dos valores obtidos do teste aplicado as duas placas é de aproximadamente 9.3 vezes. Observando que a relação do valor de *clock* de uma placa para outra chega a 6.7 e a microcontrolador da STM possui DMA no periférico de Ethernet, a quantidade de mensagens comparando um teste ao outro está condizente.

Para fins de comparação, executou-se o teste de desempenho também com o sistema FreeRTPS original. Essa comparação será executada apenas rodando o sistema com um publicador sem intervalos entre *loops*. Como o sistema original possui execução de código linear, a rotina 'frudp_listen' interrompe o processamento do código da aplicação do usuário para efetuar a decodificação das mensagens RTPS recebidas, pelo

Intervalo de tempo	Quantidade
Entre 6 e 7 segundos	2729 mensagens enviadas
Entre 7 e 8 segundos	2742 mensagens enviadas
Entre 8 e 9 segundos	2729 mensagens enviadas
Entre 9 e 10 segundos	2745 mensagens enviadas
Entre 10 e 11 segundos	2746 mensagens enviadas
Média	2739 mensagens enviadas

Tabela 4 – Quantidade máxima de mensagens enviadas pela *Stellaris Eval Board* para o primeiro teste de desempenho

tempo definido pelo usuário. Para aproveitar o máximo do processamento executando o código da aplicação, que nesse teste consiste de apenas sucessivos envios de mensagens por um publicador, atribuiu-se ao valor da função 'frudp_listen' o menor valor de tempo para processamento de mensagens recebidas possível, 1us. Dessa forma, a quantidade de mensagens publicadas será a máxima possível e fornecerá assim um bom parâmetro para a comparação com o sistema proposto. Utilizando a mesma estrutura de testes, com computador e roteador, os resultados obtidos são expostos na tabela 5. Observa-se que a taxa de mensagens enviadas por segundo fica bem reduzida, em comparação ao mesmo na STMDiscovery utilizando o sistema aqui proposto.

Intervalo de tempo	Quantidade
Entre 6 e 7 segundos	1739 mensagens enviadas
Entre 7 e 8 segundos	1820 mensagens enviadas
Entre 8 e 9 segundos	1791 mensagens enviadas
Entre 9 e 10 segundos	2003 mensagens enviadas
Entre 10 e 11 segundos	1950 mensagens enviadas
Média	1860 mensagens enviadas

Tabela 5 – Quantidade máxima de mensagens enviadas pela STMDISCOVERY para o terceiro teste de desempenho, utilizando apenas o FreeRTPS

Agora executando o sistema proposto nesse trabalho com apenas um processo efetuando a publicação sucessiva de mensagens sem intervalos, obteve-se a tabela 6.

Verifica-se, em comparação com o primeiro teste de desempenho, que apenas removendo o processo que estava executando o controle PID a 100Hz, esse teste obteve um leve aumento na quantidade de mensagens publicadas, indo de uma média de 25504 para 25859.

Por fim, observou-se que quando conectada diretamente a porta Ethernet do computador, por um cabo RJ45, o sistema FreeRTPS original ofereceu uma velocidade muito superior ao comparado quando utilizando a mesma conectada em um roteador. Para possuir uma melhor noção desse valor, efetuou-se o mesmo método dos testes anterio-

Intervalo de tempo	Quantidade
Entre 6 e 7 segundos	25789 mensagens enviadas
Entre 7 e 8 segundos	25893 mensagens enviadas
Entre 8 e 9 segundos	25716 mensagens enviadas
Entre 9 e 10 segundos	25919 mensagens enviadas
Entre 10 e 11 segundos	25902 mensagens enviadas
Média	25859 mensagens enviadas

Tabela 6 – Quantidade máxima de mensagens enviadas pela STMDISCOVERY para o quarto teste de desempenho, executando um processo com um publicador

res. Os valores obtidos são exibidos na tabela 7. Verifica-se que realmente, conectando a placa de desenvolvimento ao computador, houve um aumento expressivo na quantidade de mensagens enviadas.

Intervalo de tempo	Quantidade
Entre 6 e 7 segundos	65100 mensagens enviadas
Entre 7 e 8 segundos	65409 mensagens enviadas
Entre 8 e 9 segundos	65344 mensagens enviadas
Entre 9 e 10 segundos	65036 mensagens enviadas
Entre 10 e 11 segundos	65279 mensagens enviadas
Média	65233 mensagens enviadas

Tabela 7 – Quantidade máxima de mensagens enviadas pela STMDISCOVERY para o quinto teste de desempenho, executando um processo com um publicador e com conexão direta ao computador

4.4 Consumo de recursos

No teste de consumo de recursos, recorreu-se a função vApplicationIdleHook para analisar o consumo de recursos do sistema. Para isso, utiliza-se uma das portas de saída do microcontrolador em modo digital, ou seja, o pino só pode possuir os estados de nível lógico alto ou baixo. Inseri-se então como primeira operação de todos os processos do usuário o comando *'pin_low'* e dentro do processo de *idle*, vApplicationIdleHook, o comando *'pin_high'*. Com essa inserção de código, obtêm-se que o pino digital escolhido estará em nível lógico baixo sempre que estiver executando algum processo, e em nível lógico alto caso não esteja executando nenhum.

Com o auxílio de um osciloscópio analisa-se o pino digital por um certo período de tempo. Efetuando a somatório do tempo total que o pino permaneceu em nível lógico baixo, dentro do período de tempo analisado, obtêm-se a porcentagem de tempo que o microcontrolador permaneceu efetivamente processando código. Tem-se então que a

porcentagem de processamento utilizado é praticamente a mesma da porcentagem de tempo em nível lógico baixo.

Para esse teste, elaborou-se novamente uma estrutura de rede com um computador Dell Vostro 14-5480, uma placa de desenvolvimento STMDISCOVERY e uma placa de desenvolvimento *Stellaris LM3S6965 Evaluation Board*, conectados por cabos Ethernet a um switch TP-LINK modelo TL-WR741ND.

Utilizou-se a placa da ST à uma frequência de *clock* de 168 Mhz assim como no teste anterior. Elaborou-se o primeiro teste de consumo de recurso com o seguinte modelo: um processo executando um controle PID a uma frequência de 100Hz e outro processo publicando mensagens do tipo *String* a uma taxa de 100Hz, figura 31.

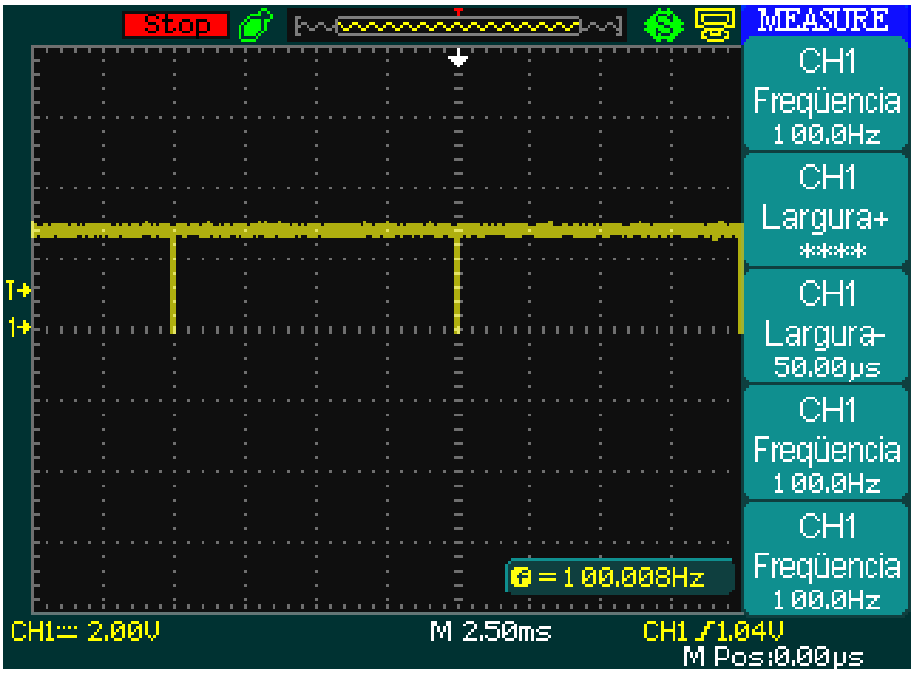

Figura 31 – Consumo de recurso com um processo PID à 100 Hz e 1 publicador do tipo *string* a 100 Hz

Analisando os tempos obtidos do primeiro teste, obtêm-se os seguintes valores apresentados na Tabela 8.

Tempo em nível lógico baixo [s]	Tempo em nível lógico alto [s]	Porcentagem em nível baixo [%]
50 us	9950 us	0.5 %

Tabela 8 – Consumo médio de processamento do sistema com processos de PID, publicador e assinante, todos a uma taca de 100 Hz

Para o segundo teste de consumo de recurso adicionou-se um processo publicando mensagens do tipo *String* a uma taxa de 1000 Hz em paralelo a um processo executando um controle PID a uma frequência de 1000 Hz, figura 32.

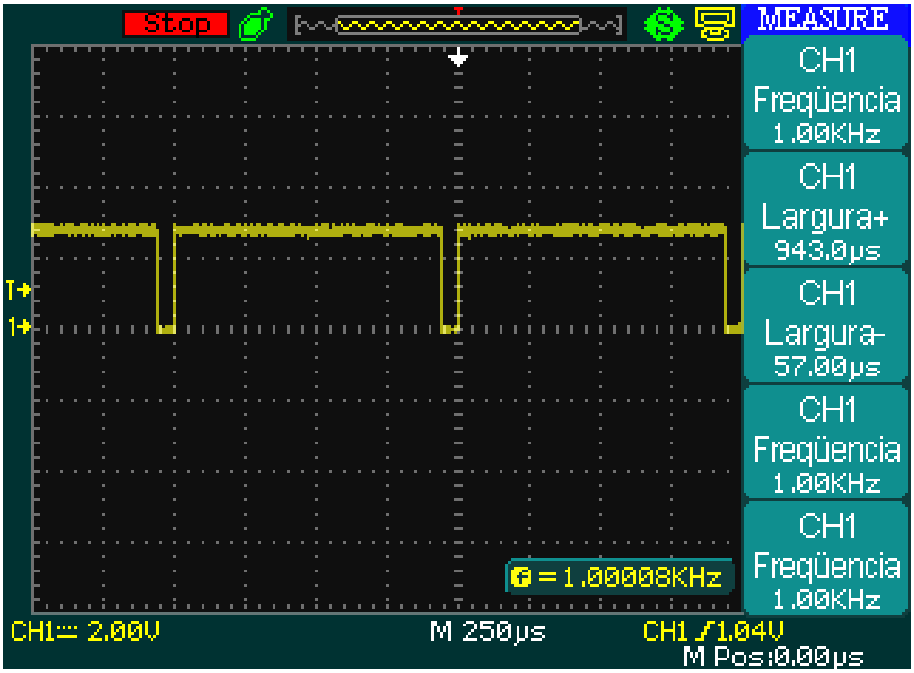

Figura 32 – Consumo de recurso com um processo PID à 1000 Hz e 1 publicador do tipo *string* a 1000 Hz

Do segundo teste, obtêm-se os seguintes valores apresentados na tabela 9.

Tempo em nível lógico baixo [s]	Tempo em nível lógico alto [s]	Porcentagem em nível baixo [%]
57 us	943 us	5.7 %

Tabela 9 – Consumo médio de processamento do sistema com processos de PID, publicador e assinante, todos a uma taca de 1000 Hz

Para o terceiro teste de consumo de recursos adotou-se apenas um processo executando um controle PID a uma frequência de 1000 Hz, figura 33.

Analisando os tempos obtidos do terceiro teste, obtêm-se os seguintes valores apresentados na tabela 10.

Tempo em nível lógico baixo [s]	Tempo em nível lógico alto [s]	Porcentagem em nível baixo [%]
22 us	973 us	2.2 %

Tabela 10 – Consumo médio de processamento do sistema com apenas um processo PID à uma taca de 1000 Hz

O último teste consiste de um processo com 5 publicadores, todos do tipo *String*, a uma taxa de 1000 Hz e outro processo executando um controle PID a uma frequência de 1000 Hz, figura 34.

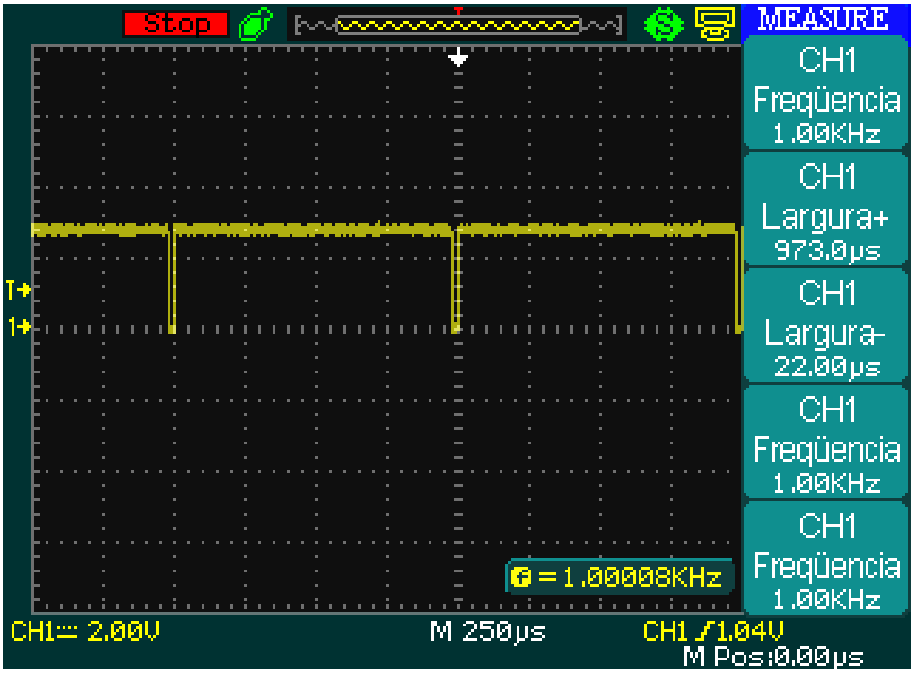

Figura 33 – Consumo de recurso com um processo PID à 1000 Hz

Obtêm-se, com o último teste, os seguintes valores apresentados na tabela 11.

Tempo em nível lógico baixo [s]	Tempo em nível lógico alto [s]	Porcentagem em nível baixo [%]
207 us	793 us	20.7 %

Tabela 11 – Consumo médio de processamento do sistema com um processo PID à 1000 Hz e 5 publicadores a uma taxa de também 1000 Hz

Com os dois primeiros testes de consumo de recurso, verifica-se que nas condições atribuídas ao sistema são necessários 50us de processamento para executar um ciclo do processo do controlador PID e do processo do publicador, e isso independente da frequência do processo PID, o que está coerente, já que os processos possuem um número fixo de instruções.

O intuito do terceiro teste do consumo de recurso foi obter o processamento utilizado para se manter apenas um processo de um controlador PID à 1000 Hz, com o processo do sistema FreeRTPS executando ao fundo. Com esse teste obteve-se que o processo PID, para as características do hardware utilizado, utiliza cerca de 22us para completar um ciclo de processamento. Ou seja, nas circunstâncias do teste, como o processo PID é executado a uma frequência de 1000 Hz, dentro de um segundo o hardware está efetuando processamento durante apenas 22ms.

Com o quarto teste de consumo de recursos, verifica-se que o tempo de processamento aumenta proporcionalmente com a quantidade de publicadores. Para cada publicador adicionado, o tempo de processamento do sistema aumenta em cerca de 35 micro

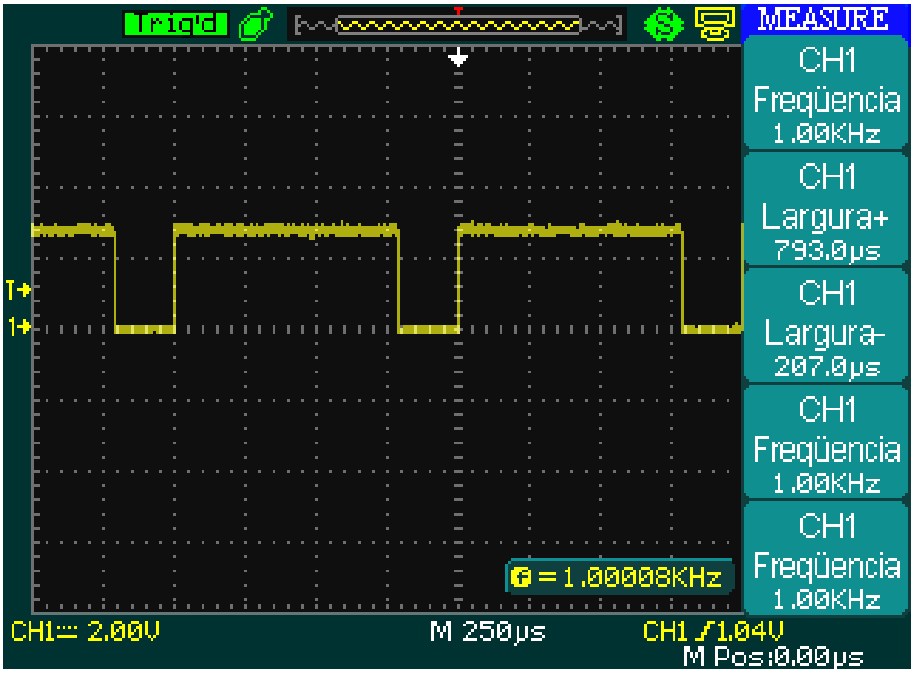

Figura 34 – Consumo de recurso com um processo PID à 1000 Hz e 5 publicadores do tipo *string* a 1000 Hz

segundos. Obtêm-se esse dado da comparação dos resultados obtidos no segundo e terceiro teste. No teste do sistema com um processo PID e um publicador obteve-se 57 micro segundos enquanto que no teste com apenas um processo PID obteve-se 22 micro segundos. Subtraindo os dois valores, adquiri-se como resultado o tempo de processamento utilizado para efetuar uma publicação, 35 micro segundos.

Utilizando o valor de tempo encontrado e considerando um programa que possua um processo PID e 5 publicadores, chega-se ao seguinte resultado, equação 4.1:

$$\begin{aligned} 5(publicadores)x35(micro\ segundos)+ \\ 22(micro\ segundos\ do\ PID) = 197(micro\ segundos) \end{aligned} \tag{4.1}$$

Comparando com o valor prático encontrado no quarto teste, verifica-se um erro de 5%, de 207 para o valor medido versus 197 do calculado.

4.5 Teste de Portabilidade

Para verificar o desempenho e a portabilidade do sistema proposto em outro microcontrolador, efetuou-se a programação do sistema em uma placa de desenvolvimento que utiliza um processador com arquitetura ARM M3, da empresa *Texas Instruments*. O teste de portabilidade nessa placa permitiu verificar se o sistema pode ser implementado sem complicações em um microcontrolador que utiliza outra arquitetura, fornecido

por um fabricante diferente e utiliza uma IDE para efetuar a compilação e gravação do programa (diferente do sistema original do FreeRTPS que utiliza o GCC e terminal do Ubuntu para essa tarefa).

Para o teste de portabilidade, utilizou-se a placa de desenvolvimento *Stellaris LM3S6965 Evaluation Board*, que conta com um microcontrolador LM3S6965 da *Texas Instruments*, e a IDE IAR *Embedded Workbench* para programação. O sistema operacional utilizado foi o *Microsoft Windows* 7 64bit. Ou seja, um dispositivo totalmente diferente da STMDiscovery.

Diferente do teste efetuado no Ubuntu 16.04, não foi necessário definir os arquivos de *setup* do microcontrolador, isto é, arquivos de *linker*, *loader* e *stack*. Os *setups* desses arquivos já são fornecidos pela própria IDE quando um novo projeto é criado, então necessitou-se apenas dos arquivos fonte e de cabeçalho. Escolheu-se a IDE IAR *Embedded Workbench* pelo fato da mesma oferecer suporte a controladores de diversas empresas e suportar mais de 11000 dispositivos (IAR..., 2017). O FreeRTOS fornece os arquivos necessários para que um programa desenvolvido no IAR possa executar seu Kernel.

Um guia detalhando os passos utilizados para se portar o sistema, para a nova placa de desenvolvimento *Stelletis Eval Board*, pode ser encontrado no apêndice C.

Como o FreeRTOS foi desenvolvido para ser facilmente portado, oferecendo disponibilidade para mais de 70 diferentes arquiteturas de microcontroladores (FREERTOS..., 2017), apenas 3 arquivos necessitaram ser editados. Os 3 arquivos são:

a) *NetworkInterface*: pertence a implementação FreeRTOS+UDP e é responsável por efetuar a interface entre o periférico de Ethernet e a pilha UDP. Como cada hardware de Ethernet possui sua característica, a tarefa de efetuar a implementação do *driver* de Ethernet para receber/enviar dados para a pilha UDP é do desenvolvedor;

b) FreeRTOSConfig: arquivo de configuração do FreeRTOS. Possui as definições para habilitar recursos do FreeRTOS bem como: frequência do *clock* do sistema, frequência do escalonador, tamanho da pilha do processo;

c) FreeRTOSIPConfig: arquivo de configuração do FreeRTOS+UDP. Assim como no FreeRTOSConfig, possui definições para habilitar recursos extras e também definir parâmetros dos sistema, como: tamanho da pilha, numero de *buffers*.

Em função da arquitetura do sistema ser diferente, e também pelo fato de utilizar uma IDE para programação, alguns arquivos foram importados para o projeto de acordo com essas características. O FreeRTOS oferece os arquivos necessários para cada sistema e arquitetura dentro da pasta *portable*. Como esse teste utilizou a IDE IAR *Embedded Workbench* e um microcontrolador com arquitetura ARM m3, os arquivos fonte e de cabeçalho contidos no diretório <FreeRTPS+FreeRTOS/FreeRTOS/portable/IAR/AR

M_CM3> foram incorporados aos arquivos do projeto. Já para o FreeRTOS+UDP, fez-se necessário importar os arquivos contidos no diretório <FreeRTPS+FreeRTOS/FreeRTOS_Plus_UDP/portable/Compiler/GCC>.

Ao final do teste integrou-se a placa a uma rede, que dispunha de um computador executando publicadores e assinantes, e observou-se que sistema foi implementado com sucesso, pois os tópicos correspondentes aos dois dispositivos completaram as trocas de mensagem com êxito.

5 CONCLUSÃO E TRABALHOS FUTUROS

Neste capítulo são apresentadas as conclusões do presente trabalho e as recomendações para a continuidade dos trabalhos nesta área de estudo.

5.1 Conclusão

Analisando os resultados obtidos nos testes de desempenho e consumo de recursos, conclui-se que o consumo de processamento inerente ao FreeRTOS não é significante no sistema como um todo, nas condições estabelecidas nesse trabalho, enquanto que as ferramentas oferecidas pelo sistema operacional agregam muitos recursos ao FreeRTPS, as mais importante delas os recursos para implementação de tempo real. Deve-se atentar que o sistema não garante necessariamente tempo real, mas sim ferramentas para se implementar um, a capacidade da aplicação, utilizando esse sistema, oferecer tempo real fica sob responsabilidade do desenvolvedor.

O teste de portabilidade mostra que é possível portar o sistema entre diferentes dispositivos sem muitas modificações em código. Atingindo assim a meta de oferecer um sistema portável. Tem-se também, que a arquitetura desenvolvida para programação do sistema mostrou-se de fácil compreensão e implementação de novas aplicações.

Já os dados obtidos do teste de desempenho mostram que o sistema final possui uma alta capacidade de publicação, mesmo enquanto mantém um processo em tempo real a 1000 Hz. A placa STMDiscovery, rodando a 168 Mhz, ofereceu uma média de 25000 mensagens publicadas por segundo, enquanto a Stellaris Eval Board forneceu cerca de 2700 publicações por segundo, com seu *clock* principal a 25 Mhz. Efetuando a comparação dos resultados obtidos junto à frequência de operação do *clock*, observa-se que é possível obter uma estimativa dos requisitos mínimos do dispositivo que será utilizado em função das características da aplicação, pois, como foi observado, o desempenho do sistema está diretamente ligado à capacidade de processamento do microcontrolador. Isso demonstra que a escolha do micro controlador deve ser levada em consideração, levando em conta a quantidade, velocidade e complexidade dos processos inseridos no sistema.

Comparando os resultados dos testes de desempenho, observa-se também, que quando conectada diretamente ao computador, a placa de desenvolvimento executando o código do FreeRTPS original, oferece uma dimensão na taxa de publicação gigantesca. Porém, quando o mesmo sistema é atribuído a um roteador, intermediando a transmissão, o sistema possui uma redução de 30 vezes. Levantou-se a hipótese de que essa variação seja inerente a forma como os pacotes, da pilha UDP/IP, do sistema FreeRTPS original

foram implementados, fazendo com que o roteador tenha que fazer mais operações para distribuir as mensagens do mesmo.

Durante o teste de portabilidade, a IDE utilizada forneceu o tamanho do arquivo binário compilado,o sistema FreeRTOS+FreeRTPS, assinando um tópico, e efetuando o processo de um controlador PID em tempo-real a 100 Hz, atingiu o tamanho de 32Kbyts. Esse valor nos apresenta a possibilidade de portar o sistema para uma variedade de micro-controladores com menores recursos de processamento e memória. Já o arquivo binário gerado pelo GCC em ambiente linux para a STMDiscovery, chegou ao tamanho de 74KBytes. Comparando o tamanho final e os resultados do teste de desempenho, verifica-se que o sistema proposto oferece uma excelente alternativa aos projetos de sistemas embarcados para ROS.

Apesar das limitações oferecidas pela licença do FreeRTOS+UDP, observa-se que os recursos adicionados por essa pilha UDP são consideráveis. Pelo fato dela ser integrada ao sistema de processos do FreeRTOS, todo o fluxo de processamento de pacotes ethernet acaba sendo otimizado para se trabalhar nesse sistema. Além disso, o essa pilha UDP fornece vários recursos interessantes, tais como: DHCP, ping, DNS e fragmentação de mensagem.

Da maneira como foi implementado o processo de manipulação dos dados do assinante, a rotina de manipulação não interrompe a execução do processo *listener*. O FreeRTPS original utiliza um *callback* para executar a rotina do manipulador dos assinantes e, se este for muito longo ou mal implementado, o processo *listener* aguardará o fim do manipulador para continuar o processamento das mensagens recebidas pela rede. No sistema proposto aqui, o processo *listener* apenas coloca os dados na fila de assinantes e continua com a decodificação dos dados. Quando o gerenciador de processos coloca o manipulador do assinante para executar, ele obtêm os dados da fila e os processa, tornado-se independente do *listener*.

Por fim, apesar do foco do trabalho não ser alto desempenho, mas sim validação da arquitetura proposta, verifica-se que a arquitetura ofereceu resultados satisfatórios e funcionais. Mesmo o sistema operacional adicionando consumo de processamento ao sistema, a taxa de publicação de mensagens permaneceu em um nível bem satisfatório. Comparando com o trabalho relacionado (ROS..., 2017b), que oferece uma publicação máxima de 50 mensagens por segundo, observa-se que a resposta do sistema proposto nesse trabalho foi relativamente alta, apesar de oferecer menos recursos do que a implementação que foi desenvolvida em NuttX.

5.2 Trabalhos futuros

Observou-se durante o desenvolvimento, que é possível integrar o sistema FreeRTPS a um nível mais profundo com o sistema operacional FreeRTOS. Um dos possíveis trabalhos futuros para esse sistema, consiste em utilizar os recursos oferecidos pelo sistema operacional na implementação do FreeRTPS. Um estudo efetuando essa maior incorporação pode mostrar se é possível melhorar o desempenho do sistema FreeRTOS+FreeRTPS. Dentre os recursos estão alocação dinâmica de memória e uma melhor fragmentação de funções em processos.

Deve-se observar que as prioridades de cada processo do sistema foram implementadas para teste de conceito, o único processo de tempo real foi implementado com maior prioridade enquanto os outros processos do usuário e do sistema foram implementados com menor prioridade. Um estudo mais elaborado para se encontrar quais os níveis de prioridade, devem ser utilizados para oferecer uma melhor utilização da aplicação, deve ser feito.

REFERÊNCIAS

ABOUT ROS. 2017. Disponível em: <http://www.ros.org/about-ros/>. Citado na página 34.

AL-MADANI, B.; AL-SAEEDI, M.; AL-ROUBAIEY, A. A. Scalable wireless video streaming over real-time publish subscribe protocol (rtps). In: IEEE. *Distributed Simulation and Real Time Applications (DS-RT), 2013 IEEE/ACM 17th International Symposium on*. [S.l.], 2013. p. 221–230. Citado na página 31.

ALMADANI, B. Rtps middleware for real-time distributed industrial vision systems. In: IEEE. *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*. [S.l.], 2005. p. 361–364. Citado na página 31.

ALMEIDA, R. M. A. de. *Troca de contexto segura em sistemas operacionais utilizando técnicas de detecção e correção de erros*. Tese (Doutorado) — Universidade Federal de Itajubá, 12 2013. Citado 3 vezes nas páginas 17, 18 e 19.

CHANGES between ROS 1 and ROS 2. 2017. Disponível em: <http://design.ros2.org/articles/changes.html>. Citado na página 38.

COMER, D. E. *Computer Networks and Internetts*. [S.l.]: Pearson Education, 2004. ISBN 0-13-143351-2. Citado na página 28.

DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R. *Sistemas Operacionais*. [S.l.]: Pearson Education, 2005. ISBN 85-7605-011-0. Citado 4 vezes nas páginas 18, 20, 21 e 30.

DOCUMENTS Associated With The Real-Time Publish-Subscribe Wire Protocol DDS Interoperability™ Wire Protocol Specification, V2.2. [S.l.], 2017. Disponível em: <http://www.omg.org/spec/DDSI-RTPS/2.2/>. Citado na página 32.

FADALI, M.; VISIOLI, A. *Digital Control Engineering: Analysis and Design*. [S.l.]: Elsevier Science, 2009. (Analysis and Design Series). ISBN 9780080922867. Citado na página 27.

FREERTOS FAQ - What is This All About? 2017. Disponível em: <http://www.freertos.org/FAQWhat.html#WhyUseRTOS>. Citado 2 vezes nas páginas 23 e 82.

FREERTOS+UDP. 2017. Disponível em: <http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_UDP/FreeRTOS_Plus_UDP.shtml>. Citado na página 23.

FREERTPS. 2017. Disponível em: <https://github.com/ros2/freertps/wiki>. Citado 3 vezes nas páginas 14, 34 e 40.

GALLMEISTER, B. *POSIX.4 Programmers Guide: Programming for the Real World*. [S.l.]: O'Reilly Media, Incorporated, 1995. (O'Reilly Series). ISBN 9781565920743. Citado na página 22.

HEATH, S. *Embedded Systems Design*. [S.l.]: Elsevier Science, 2002. ISBN 9780080477565. Citado 2 vezes nas páginas 18 e 21.

HOANG, H. et al. Switched real-time ethernet with earliest deadline first scheduling protocols and traffic handling. In: IEEE. *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*. [S.l.], 2001. p. 6–pp. Citado na página 29.

IAR Embedded Workbench - IAR Systems. 2017. Disponível em: <https://www.iar.com/iar-embedded-workbench/>. Citado na página 82.

INTRODUÇÃO ao JSON. 2017. Disponível em: <http://www.json.org/json-pt.html>. Citado na página 41.

JULIO, R. E. *DBML: Uma Biblioteca de Gerenciamento Dinâmico de Banda para Sistemas Multirrobôs Baseado em ROS*. Dissertação (Mestrado) — Universidade Federal de Itajubá, 7 2015. Citado 2 vezes nas páginas 34 e 36.

KAY, J. *Introduction to Real-time Systems*. 2017. Disponível em: <http://design.ros2.org/articles/realtime_background.html>. Citado 3 vezes nas páginas 13, 24 e 25.

KAY, J.; TSOUROUKDISSIAN, A. *Real-time Performance in ROS 2*. 2017. Disponível em: <https://roscon.ros.org/2015/presentations/RealtimeROS2.pdf>. Citado na página 15.

KERMANI, M. M. et al. Emerging frontiers in embedded security. In: IEEE. *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*. [S.l.], 2013. p. 203–208. Citado na página 17.

KIM, J.; LEE, B. Y.; PARK, J. Preemptive switched ethernet for real-time process control system. In: IEEE. *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*. [S.l.], 2013. p. 171–176. Citado na página 29.

LAGES, W. F. *Sistemas de Tempo Real*. [S.l.]: Editora UFRGS, 2014. ISBN 978-85-386-0234-7. Citado 2 vezes nas páginas 24 e 25.

LEAL, E. de S. *Projeto de uma controladora Single Loop com arquitetura ARM*. Dissertação (Mestrado) — Universidade Federal de Itajubá, 6 2012. Citado 2 vezes nas páginas 27 e 28.

LI, Q.; YAO, C. *Real-Time Concepts for Embedded Systems*. [S.l.]: Taylor & Francis, 2003. (CMP books). ISBN 9781578201242. Citado 2 vezes nas páginas 17 e 22.

LICENSE Details. 2017. Disponível em: <http://www.freertos.org/a00114.html>. Citado na página 23.

LOESER, J.; HAERTIG, H. Low-latency hard real-time communication over switched ethernet. In: IEEE. *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*. [S.l.], 2004. p. 13–22. Citado na página 29.

LOESER, J.; WOLTER, J. Scheduling support for hard real-time ethernet networking. In: *Workshop on Architectures for Cooperative Embedded Real-Time Systems (WACERTS)*. [S.l.: s.n.], 2004. Citado na página 29.

MARTINEZ, A.; FERNÁNDEZ, E. *Learning ROS for robotics programming*. [S.l.]: Packt Publishing Ltd, 2013. Citado na página 35.

MARWEDEL, P. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. [S.l.]: Springer Netherlands, 2010. (Embedded Systems). ISBN 9789400702578. Citado na página 17.

MULTITASKING. 2017. Disponível em: <http://www.freertos.org/implementation/a00004.html>. Citado 2 vezes nas páginas 19 e 20.

NISE, N. *Engenharia De Sistemas De Controle*. [S.l.]: LTC, 2009. ISBN 9788521621355. Citado na página 27.

OGATA, K. *Engenharia de controle moderno*. [S.l.]: Pearson Prentice Hall, 2011. ISBN 9788576058106. Citado na página 26.

PÁEZ, F. E. et al. Freertos user mode scheduler for mixed critical systems. *2015 Sixth Argentine Symposium and Conference on Embedded Systems (CASE)*, 2015. Citado 2 vezes nas páginas 13 e 15.

PHILIPS, T. N. C. *Digital Control Systems - Analysis and Design*. [S.l.]: Pearson Prentice Hall, 1995. Citado 2 vezes nas páginas 13 e 25.

PREREQUISITES. 2017. Disponível em: <https://github.com/ros2/freertps/wiki/Prerequisites>. Citado na página 34.

PROGRAM Real-Time ROS Nodes on STM32. 2017. Disponível em: <https://github.com/bosch-ros-pkg/stm32>. Citado 2 vezes nas páginas 43 e 44.

PROPOSAL for Implementation of Real-time Systems in ROS 2. 2017. Disponível em: <http://design.ros2.org/articles/realtime_proposal.html>. Citado na página 39.

PULSE-WIDTH modulation. 2017. Disponível em: <https://en.wikipedia.org/wiki/Pulse-width_modulation>. Citado na página 26.

QUIGLEY, M. *Bridging ROS to Embedded Systems: A Survey*. 2017. Disponível em: <https://roscon.ros.org/2013/wp-content/uploads/2013/06/ros_and_embedded_systems.pdf>. Citado na página 15.

QUIGLEY, M. *ROS 2 on 'small' embedded systems*. 2017. Disponível em: <https://roscon.ros.org/2015/presentations/ros2_on_small_embedded_systems.pdf>. Citado na página 15.

QUIGLEY, M. et al. Ros: an open-source robot operating system. In: KOBE. *ICRA workshop on open source software*. [S.l.], 2009. v. 3, n. 3.2, p. 5. Citado na página 35.

ROS 2 middleware interface. 2017. Disponível em: <http://design.ros2.org/articles/ros_middleware_interface.html>. Citado na página 39.

ROS 2.0 NuttX prototype. 2017. Disponível em: <https://github.com/bosch-ros-pkg/ros2_embedded_nuttx>. Citado 3 vezes nas páginas 43, 44 e 85.

ROS Introduction. 2017. Disponível em: <http://wiki.ros.org/ROS/Introduction>. Citado na página 35.

ROSBRIDGE_SUITE. 2017. Disponível em: <http://wiki.ros.org/rosbridge_suite>. Citado 2 vezes nas páginas 14 e 42.

ROSC. 2017. Disponível em: <http://wiki.ros.org/rosc>. Citado 2 vezes nas páginas 14 e 43.

ROSLIBJS. 2017. Disponível em: <http://wiki.ros.org/roslibjs>. Citado na página 42.

ROSSERIAL. 2017. Disponível em: <http://wiki.ros.org/rosserial>. Citado 3 vezes nas páginas 14, 40 e 41.

ROSSERIAL_EMBEDDEDLINUX. 2017. Disponível em: <http://wiki.ros.org/rosserial_embeddedlinux>. Citado na página 41.

RTPS Introduction. 2017. Disponível em: <http://www.eprosima.com/index.php/resources-all/rtps>. Citado 2 vezes nas páginas 31 e 32.

SANTOS, C. A. M. dos. *Sistema Dinâmico de Economia de Energia em RTOS*. Dissertação (Mestrado) — Universidade Federal de Itajubá, 2 2017. Citado 4 vezes nas páginas 17, 22, 23 e 25.

SCHIMMEL, A.; ZOITL, A. Real-time communication for iec 61499 in switched ethernet networks. In: IEEE. *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on*. [S.l.], 2010. p. 406–411. Citado na página 29.

SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. *Operating System Concepts, 9th Edition*. [S.l.]: Wiley & Sons, 2009. Citado na página 18.

STALLINGS, W. *Operating Systems: Internals and Design Principles*. [S.l.]: Pearson/Prentice Hall, 2009. (GOAL Series). ISBN 9780136006329. Citado na página 18.

STELLARIS LM3S Microcontroller. 2017. Disponível em: <http://www.ti.com/product/LM3S6965>. Citado na página 65.

STELLARIS® LM3S6965 Evaluation Board, User's Manual. 2017. Disponível em: <http://www.ti.com/lit/ug/spmu029a/spmu029a.pdf>. Citado 2 vezes nas páginas 65 e 66.

STM32F407VG, High-performance foundation line, ARM Cortex-M4 core with DSP and FPU, 1 Mbyte Flash, 168 MHz CPU, ART Accelerator, Ethernet, FSMC. 2017. Disponível em: <http://www.st.com/en/microcontrollers/stm32f407vg.html>. Citado na página 64.

STM32F4DISCOVERY. 2017. Disponível em: <http://www.st.com/en/evaluation-tools/stm32f4discovery.html>. Citado 2 vezes nas páginas 18 e 64.

STMICROELECTRONICS STM32F4DIS-BB. 2017. Disponível em: <https://www.digikey.com/product-detail/en/stmicroelectronics/STM32F4DIS-BB/497-13545-ND/3878236>. Citado na página 65.

STUDNIA, I. et al. Survey on security threats and protection mechanisms in embedded automotive networks. In: IEEE. *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*. [S.l.], 2013. p. 1–12. Citado na página 17.

UROSNODE: a middleware targeted to embedded systems. 2017. Disponível em: <https://github.com/openrobots-dev/uROSnode>. Citado na página 14.

USING PWM to Generate Analog Output. 2017. Disponível em: <http://ww1.microchip.com/downloads/en/AppNotes/00538c.pdf>. Citado na página 26.

VESTAL, S. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: IEEE. *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*. [S.l.], 2007. p. 239–243. Citado na página 13.

WHY ROS2. 2017. Disponível em: <http://design.ros2.org/articles/why_ros2.html>. Citado 2 vezes nas páginas 37 e 38.

WOODALL, W. *ROS on DDS*. 2017. Disponível em: <http://design.ros2.org/articles/ros_on_dds.html>. Citado na página 30.

WULF, W. et al. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, ACM, v. 17, n. 6, p. 337–345, 1974. Citado na página 18.

YIMING, A.; EISAKA, T. A switched ethernet protocol for hard real-time embedded system applications. *Journal of Interconnection Networks*, World Scientific, v. 6, n. 03, p. 345–360, 2005. Citado na página 29.

ZHANG, L.; WANG, Z. Real-time performance evaluation in hybrid industrial ethernet networks. In: IEEE. *Intelligent Control and Automation (WCICA), 2010 8th World Congress on*. [S.l.], 2010. p. 1842–1845. Citado na página 29.

APÊNDICE A – GUIA, UTILIZANDO RTPS+RTOS NA STM

Como o sistema proposto nesse trabalho foi desenvolvido com base no projeto FreeRTPS, necessita-se instalar algumas ferramentas que são utilizadas para compilar e gravar o código na placa STMDISCOVERY. Os passos são os mesmos descritos na página <https://github.com/ros2/freertps/wiki/Prerequisites> (ultimo acesso 08/2017), exceto a necessidade de edição de um dos arquivos do openOCD, que não efetua a gravação da STMDISCOVERY se não ajustado.

Porem, como é possível que a pagina sofra modificações com o tempo, a instalação dos itens necessários será descrita a seguir e também estará disponível na pagina <https://github.com/Expertinos/FreeRTPS-FreeRTOS>. O sistema operacional utilizado nesse passo a passo é o Ubuntu 16.04.

Inicialmente, é necessário instalar os pacotes build-essential, cmake e git, através do seguinte comando no terminal.

```
1 sudo apt-get install build-essential cmake git
```

Agora, necessita-se instalar o cross-compiling toolchain para ARM Cortex-M. Ele é responsável por compilar o código fonte da aplicação e gerar o binário para a arquitetura ARM. Os passos a seguir efetuam a instalação do pacote GNU ARM Embedded Toolchain.

```
1 sudo apt-get install software-properties-common
2 sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
3 sudo apt-get update
4 sudo apt-get install gcc-arm-embedded
```

A próxima etapa consiste na instalação do OpenOCD, uma ferramenta que efetua a gravação do programa na placa de desenvolvimento. Nos próximos comandos cria-se uma pasta no diretório raiz (aonde o código fonte da aplicação será baixado), configura-se os parâmetros da ferramenta e por fim se efetua a instalação.

```
1 sudo apt-get install libtool autoconf automake pkg-config libusb-1.0-0-dev
    ↪ libhidapi-dev
2 cd ~
3 git clone http://repo.or.cz/openocd.git
4 cd openocd
```

```
./bootstrap
./configure --enable-stlink --enable-ftdi --enable-cmsis-dap --prefix=/usr/
    ↪ local
make -j4
sudo make install
```

O próximo passo é efetuar o download do sistema FreeRTPS + FreeRTOS. Para isso necessita-se acessar o site <https://github.com/Expertinos/FreeRTPS-FreeRTOS> e efetuar o download da arquivo FreeRTPS+FreeRTOS.rar. Finalizado o download, o conteúdo deve ser extraído em uma pasta a desejo do usuário.

Como o sistema ainda só possui suporte para mensagens string do ROS2, necessita-se apenas navegar ate a pasta do sistema através do terminal e digitar o comando 'make'. Ao executar o comando 'make' no terminal as aplicações 'talker' e 'listener', contidas no diretório 'freertps_freertos/apps', são compiladas e seus arquivos binários são gerados.

Para finalizar, efetua-se a gravação das aplicações através dos comandos 'makepr ogram-listener-stm32f4_disco-metal' e 'makeprogram-talker-stm32f4_disco-metal'. Para que os comandos anteriores sejam executados corretamente, necessita-se que os arquivos binários tenham sido criados e que a linha de comando do terminal esteja no diretório do sistema (freertps_freertos).

Em tempo de escrita desse documento, o ROS2 para desktop possui 2 exemplos que podem se comunicar com as aplicações descritas acima. Os exemplos do ROS2 são o 'listener___rmw_opensplice_cpp' e o 'talker___rmw_opensplice_cpp', o exemplo listener subescreve o tópico chatter, enquanto o exemplo talker publica strings no tópico chatter. Para executar esses exemplos, necessita-se instalar o ROS2 na maquina através do tutorial descrito na pagina <https://github.com/ros2/ros2/wiki/Installation>.

Após instalado o ROS2, os exemplos podem ser executados através dos comandos

```
cd ~/ros2_ws
source install/setup.bash
listener__rmw_opensplice_cpp
```

Para executar o programa listener no terminal, ou atraves dos comandos:

```
cd ~/ros2_ws
source install/setup.bash
talker__rmw_opensplice_cpp
```

Para executar o programa talker. Aonde 'ros2_ws' é a pasta aonde o ROS2 foi baixado e compilado.

APÊNDICE B – CÓDIGO, TESTE TEMPO REAL

No listing B.1 é apresentado o código utilizado em uma placa de desenvolvimento STMDISCOVERY, ele é responsável por executar a rotina de um controlador PID. O sistema implementado efetua o controle de tensão de um capacitor através de PWM e um controlador PID, recebe os setpoints de controle através do tópico 'setpoint' (dado tipo uint32) e publica a valor atual da tensão do capacitor no tópico 'adcvalue' (dado tipo uint32).

```
1  #include <stdio.h>
2  #include "freertps/freertps.h"
3  #include "std_msgs/uint32.h"
4  #include "freertps/RTOSInterface.h"
5
6  //If use PID is set to 1 the program use closed loop control, if not the
       ↪ program use open loop control
7  #define usePID 1
8  //PID variables
9  #define KP  10.0
10 #define KI  500.0
11 #define KD  0.0
12 #define PIDsampleTime 0.001
13 //Max and Min voltage of the PWM wave
14 #define PWM_MAX_VOLTAGE 3.0
15 #define PWM_MIN_VOLTAGE 0.0
16 //Number of bits from PWM
17 #define PWM_BITS 16
18 //Number of bits from ADC
19 #define ADC_BITS 12
20 //PID control task
21 static void ctrlTask( void *parameters );
22 //Voltage publisher task
23 static void pubCurVoltTask( void *parameters );
24 //Topic setPoint, uint32, subscriber task
25 static void desiredVoltSubTask( void *parameters );
26 //PWM init function
27 void initPWM();
28 //ADC init function
29 void initADC();
30 //Current ADC value function
```

```
31 uint16_t getADCValue();
32 //Voltage publisher variable
33 frudp_pub_t *pub;
34 //PID voltage setpoint variable. Start with 2048 what means 1.5 volts
35 double setPoint = 0.0;
36 //Current analogic to digital value
37 double currentADCValue = 0.0;
38 //MAC address, must be different for each device
39 uint8_t ucMACAddress[ 6 ] = { 0x2C, 0x4D, 0x59, 0x01, 0x23, 0x51 };
40 //Desired IP parameter if DHCP do not work
41 static const uint8_t ucIPAddress[4]={192,168,2,151};
42 static const uint8_t ucNetMask[4]={255,255,255,0};
43 static const uint8_t ucGatewayAddress[4]={ 192,168,2,0 };
44 static const uint8_t ucDNSServerAddress[4]={208,67,222,222 };
45
46 //Main function
47 int main( void ){
48   //Clock configuration before FreeRTOS_IPInit
49   //Start FreeRTOS+UDP stack
50   FreeRTOS_IPInit( ucIPAddress, ucNetMask, ucGatewayAddress,
       ↪ ucDNSServerAddress, ucMACAddress );
51   //Start operating system scheduler
52   vTaskStartScheduler();
53   //Infinite loop, the program must not reach this loop
54   for(;;){}
55   return 0;
56 }
57
58 //Function to setup pubs, subs and others tasks
59 void setup( void ){
60   //Init ADC and PWM peripheral
61   initADC();
62   initPWM();
63   //Pubs here. Example pub = freertps_create_pub( topicName, typeName );
64   pub = freertps_create_pub( "currentVolt", std_msgs___uint32___type.
       ↪ rtps_typename );
65   //Subs here. Example freertps_create_sub( topicName, typeName,
       ↪ handlerTask, dataQueueSize );
66   freertps_create_sub( "desiredVolt", std_msgs___uint32___type.rtps_typename,
       ↪  desiredVoltSubTask, 10 );
67   //General tasks here. ctrlTask with greater priority
68   xTaskCreate( pubCurVoltTask, "pubCurVoltTask", configMINIMAL_STACK_SIZE,
       ↪ NULL, tskIDLE_PRIORITY + 1, NULL );
69   xTaskCreate( ctrlTask, "ctrlTask", configMINIMAL_STACK_SIZE, NULL,
       ↪ tskIDLE_PRIORITY + 2, NULL );
70 }
71
```

```
72 //PID control task
73 static void ctrlTask( void *parameters ){
74   //Variable that holds the time at which the task was last unblocked
75   TickType_t xLastWakeTime = xTaskGetTickCount();
76   //PID variables
77   double y0, y1, y2, e0, e1, e2;
78   const double kp = KP;
79   const double kd = KD * 2.0 / PIDsampleTime;
80   const double ki = KI * PIDsampleTime / 2.0;
81   //Init the variables
82   e0 = e1 = e2 = y0 = y1 = y2 = 0.0;
83   //PID loop
84   while( true ){
85 #if usePID == 1
86     //Getting the current ADC value
87     currentADCValue = getADCValue();
88     //Doing the calculations
89     y2 = y1;
90     y1 = y0;
91     e2 = e1;
92     e1 = e0;
93     e0 = setPoint - ( currentADCValue / ( ( 1 << ADC_BITS ) - 1 ) );
94     //PID digital equation using bilinear (tustin) aproximation
95     y0 = y2 + kp * ( e0 - e2 ) + ki * ( e0 + 2.0 * e1 + e2 ) + kd * ( e0 -
         ↪ 2.0 * e1 + e2 );
96 #else
97     //open loop test, output just receives the desired value
98     y0 = setPoint;
99 #endif
100    //Saturating at maximum value in terms of volts output
101    if( y0 > PWM_MAX_VOLTAGE )
102      y0 = PWM_MAX_VOLTAGE;
103    //Saturating at minimum value in terms of volts output
104    if( y0 < PWM_MIN_VOLTAGE )
105      y0 = PWM_MIN_VOLTAGE;
106    //Setting the new PWM value to the controled system
107    //converting voltage to digial in PWM bits
108    TIM4->CCR4 = ( uint16_t )( y0 * ( ( 1 << PWM_BITS ) - 1 ) /
         ↪ PWM_MAX_VOLTAGE );
109    //Delay sampleTime milliseconds to get the correct PID time control
110    vTaskDelayUntil( &xLastWakeTime, ( PIDsampleTime * 1000.0 ) /
         ↪ portTICK_PERIOD_MS );
111 }}
112
113 //Function to receive messages from desiredVoltage topic
114 static void desiredVoltSubTask( void *parameters ){
115   //Queue responsible for handle receiver information on "setPoint" topic
```

```
116   QueueHandle_t sQueue = ( QueueHandle_t )parameters;
117   //Variables to handle the data
118   uint8_t  msg[ 128 ] = { 0 };
119   while( true ){
120     //portMAX_DELAY tells that xQueueReceive will be blocked until data
          ↪ arrive in sQueue
121     if( xQueueReceive( sQueue, msg, portMAX_DELAY ) == pdPASS ){
122       //Led toggle to see that data has been arrived
123       led_toggle();
124       //Getting the received setpointd as 12 bit digital value
125       setPoint = *( ( uint32_t* )msg );
126       //Convert digital value to voltage
127       setPoint = PWM_MAX_VOLTAGE * setPoint / ( ( 1 << ADC_BITS ) - 1 );
128 }}}
129
130 //Voltage publisher task
131 static void pubCurVoltTask( void *parameters ){
132   //FreeRTPS publisher variables
133   uint8_t cdr[ 20 ] = { 0 };
134   int cdr_len;
135   struct std_msgs__uint32 voltage;
136   while( true ){
137     //Getting the current voltage in digital value
138     voltage.data = currentADCValue;
139     //serialize the data
140     cdr_len = serialize_std_msgs__uint32( &voltage, cdr, sizeof( cdr ) );
141     //Publish the data
142     freertps_publish( pub, cdr, cdr_len );
143     //Task period
144     vTaskDelay( 10 / portTICK_PERIOD_MS );
145 }}
146
147 //Init ADC1 - Channel 11 - Pin C1
148 void initADC(){
149   //Enable PORT C clock
150   RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN;
151   //Enable clocl for AD1
152   RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
153   //Enale port C as analog
154   GPIOC->MODER |= GPIO_MODER_MODER1_1 + GPIO_MODER_MODER1_0;
155   //Set ADC1 with channel 11
156   ADC1->SQR3 |= 0x0000000B;
157   //Enable AD1
158   ADC1->CR2 |= ADC_CR2_ADON;
159 }
160
161 //Init PWM - TIMER 4 - Channel 4 - PIN D15
```

```
162 void initPWM(){
163   //Enable PORT D clock
164   RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;
165   //Enale port D as alternate function (PWM)
166   GPIOD->MODER |= GPIO_MODER_MODER15_1;
167   //Alternate function for Pin D15 (TIMER 4 PWM Channel 4)
168   GPIOD->AFR[ 1 ] |= 0x20000000;
169   //Enable TIMER 4, used to PWM
170   RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;
171   //Timer 4 Auto-reload value
172   TIM4->ARR = 0x00000FFF;
173   //Set Timer 4 as PWM
174   TIM4->CCMR2 |= TIM_CCMR2_OC4M_2 | TIM_CCMR2_OC4M_1;
175   //TIMER 4 compare enable
176   TIM4->CCER |= TIM_CCER_CC4E;
177   //Enable TIMER 4
178   TIM4->CR1 |= TIM_CR1_CEN;
179 }
180
181 //Routine to get current ADC1 value
182 uint16_t getADCValue(){
183   //Start AD1 conversion
184   ADC1->CR2 |= ADC_CR2_SWSTART;
185   //Wait for the end of AD1 conversion
186   while( !( ADC1->SR & ADC_SR_EOC ) );
187   //Get the conversion result
188   return ADC1->DR;
189 }
```

Listing B.1 – Código utilizado no teste de validação do tempo real, executa um controlador PID à 1000Hz

No listing B.2 apresenta-se o código utilizado em outra placa STMDISCOVERY, possui uma rotina para publicar o valores desejados da tensão, que é controlado por outro nó. O código possui apenas um processo responsável por publicar os valores desejados no tópico 'setpoint' (dado tipo uint32).

```
1 #include <stdio.h>
2 #include "freertps/freertps.h"
3 #include "std_msgs/uint32.h"
4 #include "freertps/RTOSInterface.h"
5 //Number of bits from ADC
6 #define ADC_12_BITS 12
7 //ADC reference voltage
8 #define ADC_REF_VOLTAGE 3.0
9 //Voltage setpoint publisher task
```

```
10 static void pubTask( void *parameters );
11 //Voltage setpoint publisher variable
12 frudp_pub_t *pub;
13 //Device ethernet MAC address, must be different for each device
14 uint8_t ucMACAddress[6]={0x2C,0x4D,0x59,0x01,0x23,0x52};
15 //Desired IP parameter if DHCP do not work
16 static const uint8_t ucIPAddress[4]={192,168,2,152};
17 static const uint8_t ucNetMask[4]={255,255,255,0};
18 static const uint8_t ucGatewayAddress[4]={192,168,2,0};
19 static const uint8_t ucDNSServerAddress[4]={208,67,222,222};
20 //Main function
21 int main( void ){
22   //Do necessary clock configuration before FreeRTOS_IPInit
23   //Start FreeRTOS+UDP stack
24   FreeRTOS_IPInit( ucIPAddress, ucNetMask, ucGatewayAddress,
       ↪ ucDNSServerAddress, ucMACAddress );
25   //Start operating system scheduler
26   vTaskStartScheduler();
27   //Infinite loop, the program must not reach this loop
28   for(;;){}
29   return 0;
30 }
31 //Function to setup pubs, subs and others tasks
32 void setup( void ){
33   //Pubs here. Example pub = freertps_create_pub( topicName, typeName );
34   //Publisher for thetopic desiredVolt
35   pub = freertps_create_pub( "desiredVolt", std_msgs__uint32__type.
       ↪ rtps_typename );
36   //General tasks here. ctrlTask with greater priority
37   xTaskCreate( pubTask, "pubTask", configMINIMAL_STACK_SIZE, NULL,
       ↪ tskIDLE_PRIORITY + 1, NULL );
38 }
39
40 //Uint32 publisher task
41 static void pubTask( void *parameters ){
42   //FreeRTPS publisher variables
43   uint8_t cdr[ 20 ] = { 0 };
44   int cdr_len;
45   struct std_msgs__uint32 digital12bitsVoltage;
46   double desiredVolt;
47   while( true ){
48     //Desired voltage to reach
49     desiredVolt = 1.2;
50     //Converting 0V-3V volts to digital value in 12 bits
51     digital12bitsVoltage.data = ( uint32_t )( desiredVolt * ( ( 1 <<
         ↪ ADC_12_BITS ) - 1 ) / ADC_REF_VOLTAGE );
52     //Blink led each time that data is published
```

```
53      led_toggle();
54      //serialize the data
55      cdr_len = serialize_std_msgs__uint32( &digital12bitsVoltage, cdr,
            ↪ sizeof( cdr ) );
56      //Publish the data
57      freertps_publish( pub, cdr, cdr_len );
58      //Delay for 100 ms
59      vTaskDelayUntil( &xLastWakeTime, 100 / portTICK_PERIOD_MS );
60
61      //Desired voltage to reach
62      desiredVolt = 1.8;
63      //Converting 0V-3V volts to digital value in 12 bits
64      digital12bitsVoltage.data = ( uint32_t )( desiredVolt * ( ( 1 <<
            ↪ ADC_12_BITS ) - 1 ) / ADC_REF_VOLTAGE );
65      //Blink led each time that data is published
66      led_toggle();
67      //serialize the data
68      cdr_len = serialize_std_msgs__uint32( &digital12bitsVoltage, cdr,
            ↪ sizeof( cdr ) );
69      //Publish the data
70      freertps_publish( pub, cdr, cdr_len );
71      //Delay for 100 ms
72      vTaskDelayUntil( &xLastWakeTime, 100 / portTICK_PERIOD_MS );
73  }}
```

Listing B.2 – Código utilizado no teste de validação do tempo real, executa um publicador com tensões de saída desejado

APÊNDICE C – GUIA, PORTANDO O RTPS+RTOS PARA A TEXAS STELLARIS LM3S6965 EVAL. BOARD

Inicialmente, necessita-se efetuar o download e instalação da IDE IAR Embedded Workbench for ARM, da biblioteca StellarisWare (drivers de periféricos da família Stellaris), do FreeRTOS e do sistema FreeRTPS modificado para o FreeRTOS. A seguir os links de onde esses arquivos podem ser encontrados:

a) IAR Embedded Workbench: <https://www.iar.com/iar-embedded-workbench>(utilizou-seaversao8.11).

b) StellarisWare: <http://www.ti.com/tool/SW-LM3S>.

c) FreeRTOS: <https://sourceforge.net/projects/freertos/files/FreeRTOS/V9.0.0>(utilizou-seaversao9.0.0)

d) FreeRTPS+FreeRTOS: <https://github.com/Expertinos/FreeRTPS-FreeRTOS>

Os arquivos do StellarisWare são direcionados por padrão para a raiz do sistema Windows, mas podem ser instalados em outro local desejado. Os arquivos do FreeRTOS e do FreeRTPS podem ser extraídos em qualquer pasta com permissão para acesso. Após a instalação, executou-se a aplicação IAR Embedded Workbench e obteve-se a tela exibida na figura 35.

Nesse ponto, cria-se o projeto que irá possuir os códigos da aplicação, do sistema FreeRTOS e do sistema FreeRTPS + FreeRTOS. Seleciona-se então o item 'Project' na barra de menus e em seguida 'Create New Project'. Na janela seguinte seleciona-se o template 'main' em linguagem C, figura 36, e define-se então o nome e diretório do projeto.

Adiciona-se então o código de startup do microcontrolador, esse código fornece a tabela de interrupção e os manipuladores padrões das interrupções do dispositivo. A biblioteca StellarisWare fornece os arquivos de startup para as IDEs: IAR, CCS, CodeRed e para o compilador GCC. Navega-se então ate o diretório do microcontrolador utilizado, <StellarisWare/boards/ek-lm3s6965/blinky>, e copia-se o arquivo startup_ewarm.c para o diretório do projeto criado anteriormente. Mesmo copiando o arquivo para o diretório, é necessário adicionar o arquivo copiado ao projeto para que esse entre na lista de arquivos a serem processados. Para isso, clica-se com o botão direito sobre o projeto e seleciona-se o item 'Add Files', na aba 'Add'.

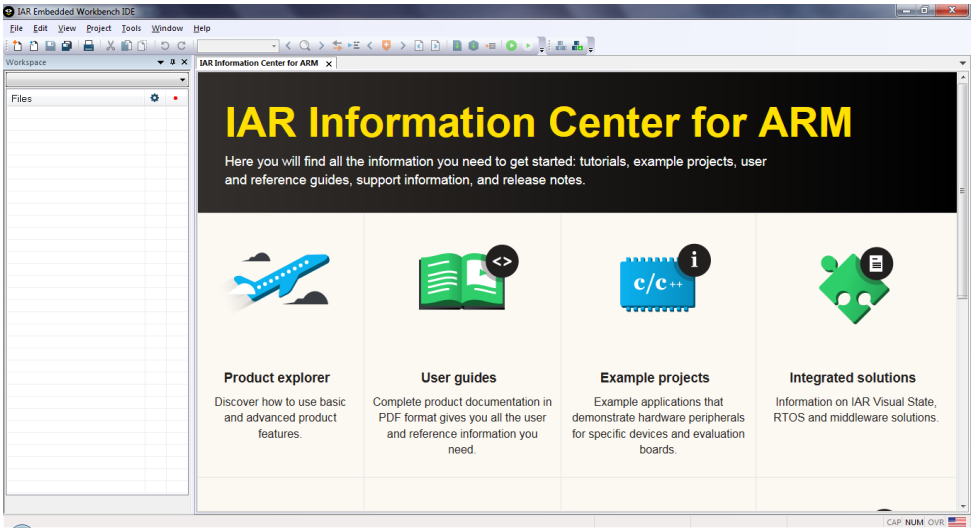

Figura 35 – Tela inicial da IDE IAR Embedded Workbench

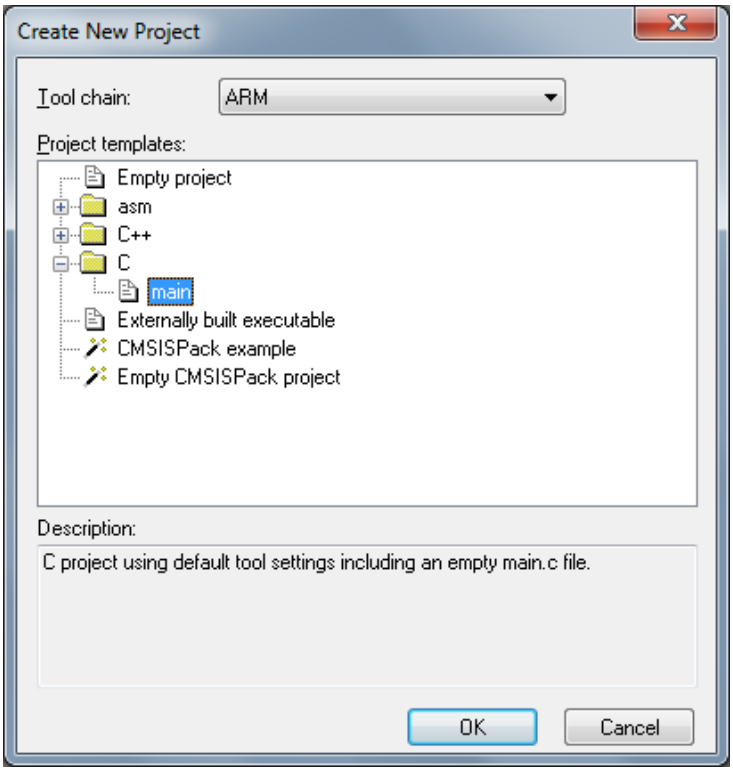

Figura 36 – Criando novo projeto

Seleciona-se agora o dispositivo alvo do projeto, para que a IDE gerencie os arquivos necessários para compilar e gravar o programa em um dispositivo especifico. Para isso, clica-se com o botão direito do mouse sobre o projeto e seleciona-se o item 'Options'. Na janela a seguir, seleciona-se a categoria 'General Options' e em seguida na aba 'Target' (à direita da janela) seleciona-se o dispositivo 'TexasInstruments LM3S6965' na caixa 'Processor Variant', figura 37.

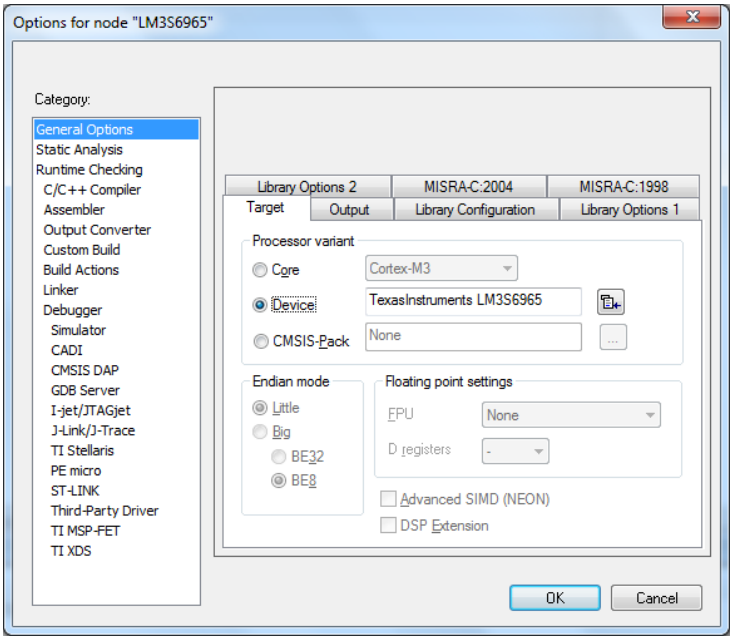

Figura 37 – Seleção do microcontrolador utilizado

Nesse ponto se insere os arquivos de dependência da biblioteca StellarisWare ao projeto. Inicialmente adiciona-se o arquivo driverlib.a ao projeto, o processo é o mesmo utilizado para inserir o arquivo de startup. Clica-se com o botão direito do mouse sobre o item 'Add Files', na aba 'Add', e em seguida seleciona-se o arquivo 'driverlib.a' no diretório <StellarisWare/driverlib/ewarm/Exe>.

Adiciona-se agora os arquivos fonte e de cabeçalhos da biblioteca StellarisWare, do FreeRTOS e do FreeRTPS modificado. Nessa etapa, clica-se como botão do mouse sobre o item 'Options' e seleciona-se a categoria 'C/C++ Compiler'. Em seguida seleciona-se a aba 'Preprocessor', que se encontra a direita da janela, e adicionam-se os diretórios dos cabeçalhos. Para os cabeçalhos do StellaresWare adiciona-se o diretório <C:/StellarisWare>.

Para o FreeRTOS são necessários dois diretórios, o diretório <FreeRTPS+FreeRTOS/FreeRTOS/include>, que contem os cabeçalhos dos arquivos fontes gerais, e o diretório <FreeRTPS+FreeRTOS/FreeRTOS/portable/IAR/ARM_CM3>, que contem os arquivos cabeçalho referentes a IDE IAR e a arquitetura ARM M3. Para o FreeRTOS+UDP são adicionados mais dois diretórios, o <FreeRTPS+FreeRTOS/FreeRTOS_lus_UDP/include> e o <FreeRTPS+FreeRTOS/FreeRTOS_Plus_UDP/portable/Compiler/GCC>. Para o FreeRTPS, adiciona-se o diretório raiz do sistema FreeRTPS baixado. E por fim o diretório do projeto através da palavra reservada '$PROJ_DIR$', figura 38.

É necessário também adicionar o diretório do projeto ao 'Preprocessor' da categoria

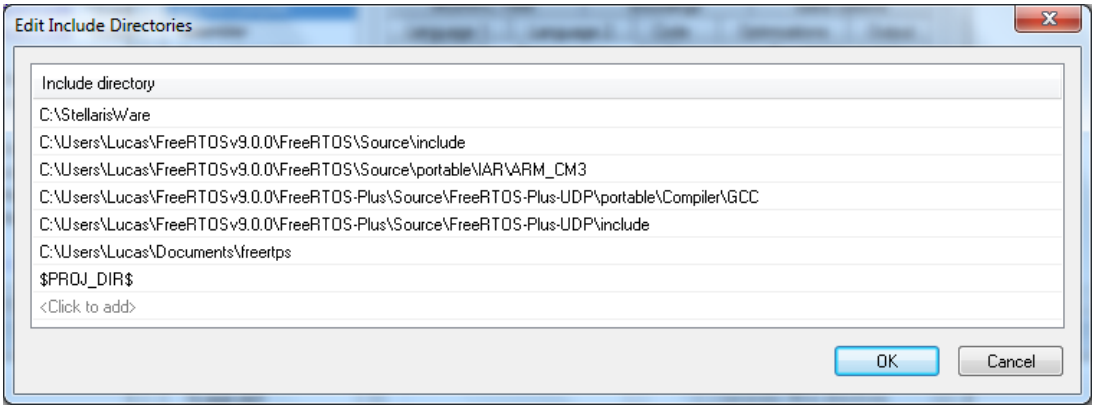

Figura 38 – C Compiler Preprocessor - Include directories

'Assembler', figura 39.

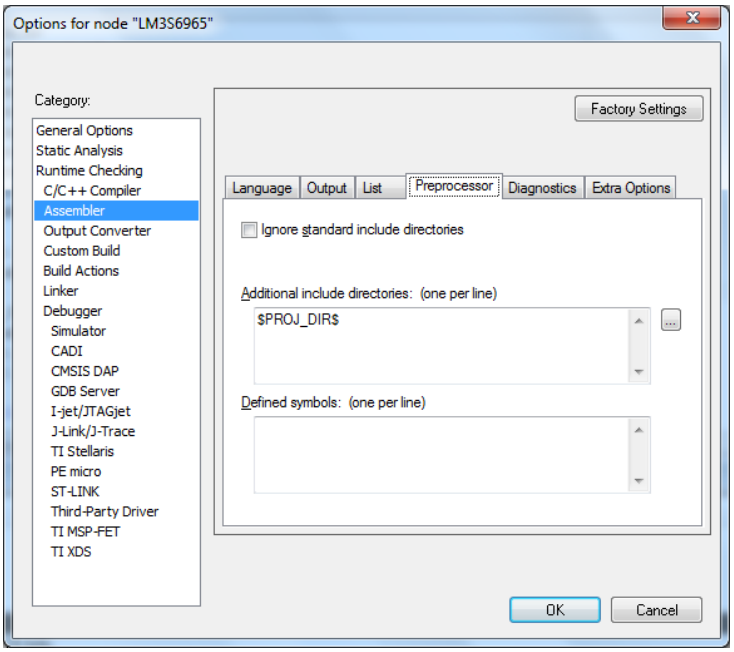

Figura 39 – Assembler Preprocessor - Include directories

Para finalizar as configurações de opções, editam-se alguns parâmetros da categoria 'Debugger'. A direita da janela na aba 'Setup', seleciona-se o 'Driver TI Stellaris', e na aba 'Download' marca-se as opções 'Verify download' e 'Use flash loader', figura 40.

Importam-se enfim os arquivos fonte do FreeRTOS e do FreeRTPS para o projeto. Novamente, clica-se com o botão direito do mouse no projeto e utiliza-se a opção adicionar arquivos. Para o FreeRTOS, adicionam-se os seguintes arquivos localizados no diretório <FreeRTPS+FreeRTOS/FreeRTOS>, figura 41.

Os arquivos 'port.c' e 'portasm.s', do diretório <FreeRTPS+FreeRTOS/FreeRTOS/portable/IAR/ARM_CM3>, e o arquivo heap4.c, do <FreeRTPS+FreeRTOS/Free

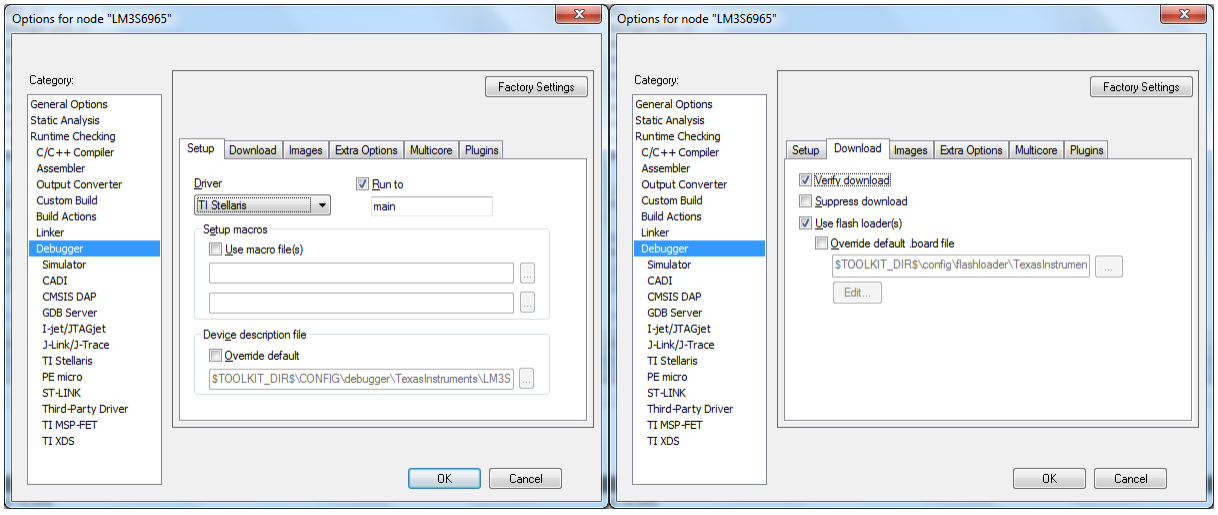

Figura 40 – Parametros do Debugger

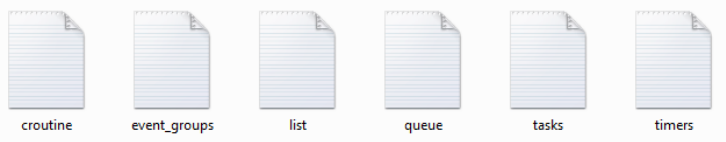

Figura 41 – Arquivos fonte do FreeRTOS

RTOS/portable/MemMang>, também devem ser inseridos.

Para o FreeRTOS+UDP, adicionam-se os seguintes arquivos localizados no diretório <FreeRTPS+FreeRTOS/FreeRTOS_Plus_UDP>, figura 42.

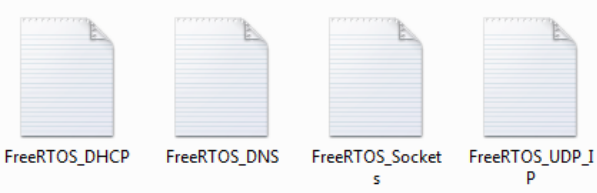

Figura 42 – Arquivos fonte do FreeRTOS+UDP

Adiciona-se também o arquivo 'BufferAllocation_2', localizado no diretório <FreeRTPS+FreeRTOS/FreeRTOS_Plus_UDP/portable/BufferManagement>, responsável por efetuar o gerenciamento da alocação de memória do FreeRTOS.

A seguir, adiciona-se o arquivo 'LM3s6965NetworkInterface.c', disponível para donwload em <https://github.com/Expertinos/FreeRTPS-FreeRTOS>. Ele é responsável por fazer a interface da pilha UDP com o hardware e é de responsabilidade do desenvolvedor. Na pasta FreeRTPS+FreeRTOS/FreeRTOS_Plus_UDP> existem alguns exemplos para outros dispositivos.

Por fim, adiciona-se o arquivo principal da aplicação, o 'main.c'. Esse arquivo também encontra-se disponível para donwload no GitHub citado no início dessa seção.

Clicando no botão Download e Debug, grava-se o dispositivo através da porta USB e a partir desse ponto a aplicação inicia sua operação normal. Quando o dispositivo é conectado a uma rede Ethernet ele procura por um endereço de IP válido, caso exista um servido DHCP na rede. No caso de não existir um servidor DHCP, o IP estático definido no arquivo 'main.c' é atribuído ao mesmo.