UNIVERSIDADE DO SUL DE SANTA CATARINA

MAX BACK

SISTEMA EMBARCADO COM RTOS:
UMA ABORDAGEM PRÁTICA E VOLTADA A PORTABILIDADE

Palhoça

2018

MAX BACK

SISTEMA EMBARCADO COM RTOS:
UMA ABORDAGEM PRÁTICA E VOLTADA A PORTABILIDADE

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Elétrica com ênfase em Telemática da Universidade do Sul de Santa Catarina como requisito parcial à obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Djan de Almeida do Rosário, Esp. Eng.

Palhoça

2018

MAX BACK

SISTEMA EMBARCADO COM RTOS:
UMA ABORDAGEM PRÁTICA E VOLTADA A PORTABILIDADE

Este Trabalho de Conclusão de Curso foi julgado adequado à obtenção do título de Engenheiro Eletricista e aprovado em sua forma final pelo Curso de Graduação em Engenharia Elétrica com ênfase em Telemática da Universidade do Sul de Santa Catarina.

Palhoça, 26 de junho de 2018.

Professor e orientador Djan de Almeida do Rosário, Esp.
Universidade do Sul de Santa Catarina

Prof. Sheila Santisi Travessa, Dr.
Universidade do Sul de Santa Catarina

Fabiano Leonida, Ms.

Prof. André Tonon, Esp.
Universidade do Sul de Santa Catarina

Dedico este trabalho a Deus, por ser o verdadeiro princípio e fim de todas as coisas (das que valem a pena).

AGRADECIMENTOS

Agradeço a minha esposa Gizele e meus filhos, Eric e Thomas, meus pais e a toda minha família que, com muito carinho e apoio, não mediram esforços para que eu chegasse até esta etapa de minha vida.

Agradeço aos professores do curso por terem transmitido seu conhecimento a mim e meus colegas com tanto empenho e dedicação.

Não existe triunfo sem perda, não há vitória sem sofrimento, não há liberdade sem sacrifício. (O Senhor dos Anéis – J.R.R. Tolkien, 1954)

RESUMO

A crescente demanda pelo desenvolvimento de solução conectadas entre sistemas já existentes, assim como a ampliação do uso destes sistemas embarcados mostram a importância de soluções robustas, de tempo real, com ciclos de desenvolvimento cada vez mais curtos e necessidade de reaproveitamento de código. Este trabalho apresenta a seleção do sistema operacional de tempo real FreeRTOS e o experimento de sua utilização como base para o desenvolvimento de um controlador *wearable* (vestível) em duas plataformas de hardware diferentes, implementando a solução inicialmente em uma delas e depois portando para a outra plataforma, desenvolvendo principalmente a programação específica do novo *hardware* e procurando manter a parte da aplicação inalterada e independente de plataforma. Como resultado foi possível realizar uma avaliação dos ganhos com portabilidade possibilitados pelo uso do sistema operacional e da quantidade de código aproveitado. Verificou-se que a ganho foi muito positivo, com cerca de setenta porcento do código da plataforma portada sendo reaproveitado. A partir deste experimento podemos concluir que o uso do FreeRTOS, além de permitir desempenho em tempo real, constitui uma ferramenta importante para o desenvolvimento de sistemas portáveis.

Palavras-chave: Sistemas Embarcados. RTOS. Portabilidade.

ABSTRACT

The increasing demand for solution development connected between existing systems as well as the increased use of these embedded systems show the importance of robust, real-time solutions with ever shorter development cycles and need for reuse of code. This work presents the selection of the real-time operating system FreeRTOS and the experiment of its use as the basis for the development of a wearable controller on two different hardware platforms, implementing the solution initially in one of them and then porting to the other platform, mainly developing the specific programming of the new hardware and trying to keep the application part unaltered and platform independent. As a result, it was possible to carry out an evaluation of the gains with portability made possible using the operating system and the amount of harnessed code. It was verified that the gain was very positive, with about seventy percent of the code of the ported platform being reused. From this experiment we can conclude that the use of FreeRTOS, besides allowing real-time performance, is an important tool for the development of portable systems.

Keywords: Embedded Systems. RTOS. Portability.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo de sistema embarcado em camadas 18
Figura 2 – Spectrum Ranking – Filtro para linguagem de sistemas embarcados 19
Figura 3 – Tipos de sistemas multiprogramáveis/multitarefas 20
Figura 4 – Tipos de escalonamento 24
Figura 5 – Time slice e preempção 25
Figura 6 – Kits comerciais para de desenvolvimento de produtos 29
Figura 7 – Principais características do kit STM32F411E-Discovery 31
Figura 8 – Modelos de placas arduino disponíveis em fornecedor local 33
Figura 9 – Módulo WiFi ESP8266 a Espressif - Informações em uma loja virtual 36
Figura 10 – Modelo cascata (ou clássico) 38
Figura 11 – Modelo iterativo e incremental 39
Figura 12 – Elementos chave do IBM Rational Unified Process 39
Figura 13 – Quadro Kanban 42
Figura 14 – Critérios para a escolha do RTOS 44
Figura 15 – Critérios para a escolha das plataformas de hardware 45
Figura 16 – Slide do webinar mostrando a comparação entre licenças (da mais permissiva até a fortemente protetora) 46
Figura 17 – Instalação do FreeRTOS para Arduino com suporte as placas Uno, Leonardo e Mega 53
Figura 18 – Monitor serial mostrando resultado da execução de aplicativo de teste do FreeRTOS no Arduino 54
Figura 19 – Ambiente de desenvolvimento Atollic TrueSTUDIO - Destaque para os painéis de informação sobre FreeRTOS durante o debug 59
Figura 20 – Diagrama de blocos do KIT STM32F411e-Discovery 61
Figura 21 – A placa Arduino Mega 2560 63
Figura 22 – Kanban digital utilizado 70
Figura 23 – Uso de diretiva de compilação condicional 75
Figura 24 – Diagrama de blocos do hardware da primeira plataforma 76
Figura 25 – KIT STM32F4e-Discovery - Principais componentes do KIT utilizados no projeto da primeira plataforma 77
Figura 26 – Esquema elétrico da placa de expansão 78
Figura 27 – Esquema mecânico de conexão entre as placas da primeira plataforma 79

Figura 28 – Fotografia do protótipo com STM32F411e-Discovery (primeira plataforma) 80
Figura 29 – Diagrama de blocos do hardware da segunda plataforma 81
Figura 30 – Esquema elétrico para a placa de expansão do Arduino (segunda plataforma) 82
Figura 31 – Esquema mecânico de conexão entre as placas da segunda plataforma 83
Figura 32 – Conjunto de fotografias do protótipo com Arduino Mega 2560 (segunda plataforma) 83
Figura 33 – Diagramas de Classes 85
Figura 34 – Interface da plataforma 86
Figura 35 – Implementação da plataforma STM32F411e-Discovery 86
Figura 36 – Implementação da plataforma Arduino Mega 2560 86
Figura 37 – Diagrama de pacotes (visão geral da aplicação) 87

LISTA DE TABELAS

Tabela 1 – Lista resumida de RTOS com código aberto ou disponível 27
Tabela 2 – Lista de IDEs disponíveis para os microcontroladores da família STM32 32
Tabela 3 – Especificações e preços dos modelos disponíveis em fornecedor local 33
Tabela 4 – Critérios para pilhas TCP/IP em sistemas embarcados 34
Tabela 5 – comparativo lado a lado entre valores e princípios de lean e agile 41
Tabela 6 – Principais Tópicos do Webinar. 47
Tabela 7 – comparativo das principais capacidades dos das duas plataformas 63
Tabela 8 – Levantamento de requisitos para o produto 67
Tabela 9 – Comparativo das plataformas 88
Tabela 10 – Porcentagem de código para implementação da plataforma STM32F4. 88
Tabela 11 – Economia na portabilidade para a plataforma Arduino. 89

LISTA DE ABREVIASÕES E SIGLAS

ABS: *Antilock braking system*, sistema de frenagem à prova de travamento das rodas;

API: *Application Programming Interface* ou Interface de Programação de Aplicativos;

C: Linguagem de programação denominada “C”;

CPU: *Central Processing Unit* ou Unidade central de processamento;

Flash: Memória *flash* é um tipo de Dispositivo de armazenamento não volátil que pode ser eletricamente apagado e reprogramado;

FreeRTOS: *real-time operating system* ou Sistema Operacional de Tempo Real, distribuído sob a licença MIT .

IBM: *International Business Machines*, que significa Máquinas de Negócio Internacionais, e é uma empresa americana que trabalha com produtos voltados para a área de informática, como computadores, *hardwares* e *softwares*.

IDE: *Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado

IOT: *Internet of Things* ou Internet das Coisas;

KIT: jogo de elementos que atendem juntos a um mesmo propósito ou utilidade;

IoT: *Internet of Things* ou Internet das Coisas;

IPC: *Inter-Process Communication* ou Comunicação entre Processos;

ISR: *Interrupt Service Routine* ou tela Rotina de Serviço de Interrupção;

LINUX: é uma família de sistemas operacionais de *software* livre e de código aberto;

MCU: *MicroController Unit* ou Microcontrolador;

MIT: *Massachusetts Institute of Technology*;

NASA: *National Aeronautics and Space Administrat*ion ou Administração Nacional do Espaço e da Aeronáutica;

PIC: família de microcontroladores;

PPP: *Point-to-Point Protocol* ou Protocolos ponto-a-ponto;

RAM: *Random Access Memory* que significa Memória Volátil;

RF: Requisito Funcional;

RNF: Requisito Não-Funcional;

RUP: *Rational Unified Process* ou Processo Unificado da Racional;

SCRUM: é um processo de desenvolvimento iterativo e incremental para gerenciamento de projetos e desenvolvimento ágil de *software*;

SRAM: (RAM estática) é a memória de acesso aleatório (RAM);

RTOS: *Real Time Operation System* ou Sistema Operacional de Tempo Real;

RX: *Receive* ou Receber;

TCP/IP: É o principal protocolo de envio e recebimento de dados MS internet. TCP significa *Transmission Control Protocol* (Protocolo de Controle de Transmissão) e o IP, *Internet Protocol* (Protocolo de Internet);

TX: *Transmit* ou Transmitir;

UNISUL: Universidade do Sul de Santa Catarina

USB: *Universal Serial Bus* ou Porta Universal

UTI: Unidade de Terapia Intensiva;

VPN: *Virtual Private Network* ou Redes Privadas virtuais;

WI-FI: Wi-Fi é uma abreviação de "*Wireless Fidelity*", que significa fidelidade sem fio, em português. Wi-fi, ou wireless é uma tecnologia de comunicação que não faz uso de cabos, e geralmente é transmitida através de frequências de rádio, infravermelhos etc.

SUMÁRIO

1 INTRODUÇÃO 12
1.1 JUSTIFICATIVA 13
1.2 OBJETIVOS 14
1.2.1 Objetivo Geral 14
1.2.2 Objetivos específicos 14
1.3 METODOLOGIA E ESTRUTURA DO TRABALHO 15
2 REVISÃO DE LITERATURA 17
2.1 SISTEMAS 17
2.1.1 Sistemas Embarcados 17
2.1.2 Sistemas Operacionais 19
2.1.3 Sistemas Operacionais de Tempo Real 21
2.1.4 Recursos oferecidos por um RTOS 23
2.1.5 Sistemas operacionais de tempo real com código fonte aberto ou disponível 26
2.2 PLATAFORMAS PARA DESENVOLVIMENTO 29
2.2.1 KIT de desenvolvimento de fabricante ST 30
2.2.2 Plataformas open source Arduino 32
2.2.3 Módulos de comunicação sem Fio TCP/IP (ethernet) ESP8266 34
2.3 METODOLOGIA DE DESENVOLVIMENTO DE SISTEMAS 37
2.3.1 Principais metodologias 37
2.3.2 Desenvolvimento com a metodologia Lean-Agile 40
3 DESENVOLVIMENTO 43
3.1 DELIMITAÇÃO E ESCOLHA DE PLATAFORMA 43
3.1.1 Escolha do Sistema Operacional de Tempo Real FreeRTOS 45
3.1.2 Escolha da plataforma de hardware STM32F411e-Discovery 56
3.1.3 Escolha da plataforma de hardware Arduino Mega2560 62
3.1.4 Escolha de módulos de comunicação e outros periféricos 65
3.2 PROBLEMA ILUSTRATIVO E REQUISTOS 66
3.2.1 Problema ilustrativo 66
3.2.2 Requisitos 66
3.3 DESENVOLVIMENTO DO PROJETO (TRABALHO PRÁTICO) 68
3.3.1 Descrição geral 68
3.3.2 Etapas de desenvolvimento 70

3.3.3 Projeto de hardware resultante 75
3.3.4 Projeto de software embarcado resultante 84
3.3.5 Analise da portabilidade entre as plataformas 87
4. RESULTADOS E DISCUSSÕES 90
CONCLUSÃO 93
REFERÊNCIAS 94
ANEXOS 98
ANEXO A – TABELA DETALHADA DE RTOS PARA SISTEMAS EMBARCADOS DE CÓDIGO ABERTO OU DISPONIBILIZADO 12
ANEXO B – CÓDIGO FONTE DE EXEMPLO DO ARDUINO 12

1 INTRODUÇÃO

Um crescimento constante da utilização dos sistemas embarcados, deu lugar para uma grande popularidade de sistemas e dispositivos que os utilizam, mesmo sem o conhecimento de seus usuários. A presença destes sistemas em nosso cotidiano aumenta em uma escala que parece logarítmica, como pode ser facilmente observado pela grande quantidade de *software* em dispositivos eletrônicos como TVs, roteadores (e outros dispositivos de rede), automações residenciais e até dispositivos vestíveis (*wearables*).

Os sistemas embarcados não estão mais confinados a indústria de áreas específicas como indústria aeroespacial, onde recordamos o reconhecimento do sistema de controle desenvolvido para o projeto Apollo da NASA como o primeiro sistema embarcado. Outros exemplos clássicos de uso de sistemas embarcados são os encontrados dentro de equipamentos médicos, micro-ondas, controle em várias partes de automóveis como o controle de tração e freios ABS.

Uma pesquisa realizada pelo site embarcados.com.br em 2017, sobre o mercado brasileiro de desenvolvimento de sistemas embarcados, com o objetivo de “identificar e traçar um perfil das principais tendências do mercado brasileiro de desenvolvimento de sistemas embarcados”, que vem sendo realizada a alguns anos pelo site embarcados.com.br indica uma tendência no aumento da capacidade dos microcontroladores/microprocessadores, já que aponta tendência de declínio no uso de versões de 8 e 16 bits e adoção de versões com 32 e 64 bits.

Destaca-se nesta área a atuação dos engenheiros, que, de acordo com a referida pesquisa correspondem a quase 70% dos profissionais que atuam na área, principalmente em aplicações que aliam tanto áreas inovadoras como IoT (Internet das coisas ou *Internet of Things*) quanto outras mais tradicionais como controles industriais e comunicação (ou redes).

A utilização de um sistema operacional em tais sistemas é cada vez mais viável, tendo-se em vista a crescente adoção de *hardware* mais rápidos e com maiores recursos computacionais, grande oferta de alternativas tanto de micro controladores quanto de sistemas operacionais, com grande suporte a vários modelos e fabricantes. Grandes atores têm investido nesta área como Microsoft, IBM e Amazon. Esta última recentemente anunciou o Amazon FreeRTOS, sua versão voltada para facilitar a IoT, com segurança e suporte nativo a funcionalidade de comunicação e armazenamento na “nuvem”.

Embora existam vários tipos de sistemas operacionais, os sistemas de tempo-real costumam ser escolhidos dado seu foco em requisitos que garanta o processamento de

estímulos de sensores e canais de comunicação (entre outros) em um tempo máximo determinado.

Este trabalho irá discutir o uso de um sistema operacional de tempo real, assim como se buscará analisar sua aplicabilidade em dois sistemas embarcados distintos, com características distintas, a fim de propiciar um maior entendimento das facilidades de cada caso e elementos próprios do desenvolvimento de sistemas embarcadas com sistemas operacionais de tempo real.

1.1 JUSTIFICATIVA

A crescente demanda pelo desenvolvimento de solução conectadas entre sistemas já existentes, que antes funcionavam isolados uns dos outros, assim como proeminente desempenho econômico do lançamento de soluções tecnológicas voltadas para a comodidade e entretenimento, encabeçada pelos smartphones, que podem ser considerados hoje os primeiros dispositivos *wearables* (vestíveis), já que além de ser algo que se possa vestir ou ter junto ao corpo constantemente, permite que o seu usuário tenha acesso a conexão com outros equipamentos e com a internet. Esta é a maior justificativa para o investimento no conhecimento e aplicação de quaisquer técnicas que permitam o desenvolvimento de produtos de forma mais eficiente e segura, diminuindo o assim chamado *Time to Market*, ou seja, o tempo necessário desde o início de um projeto de produto até seu lançamento o comercial.

Também a necessidade de reaproveitamento de código ente modelos e até dispositivos de famílias diferenciadas, além de diversos fatores técnicos que deverão ser explicados neste trabalho, é justificativa para a busca pelo domínio de sistemas operacionais que sirvam de apoio para o desenvolvimento de sistemas embarcados de auto desempenho, porém com porções de código compartilhado e reaproveitado, tão grandes quanto se possa obter, de forma segura e com garantia de qualidade, economizando recursos de equipe de desenvolvimento e outros gastos com consultorias e treinamentos.

Requisitos de confiabilidade e previsibilidade são fundamentais em sistemas de áreas como a médica e a aeroespacial, mas também são desejáveis e até indispensáveis quando se conecta uma gama cada vez maior de dispositivos, ligados em rede e comunicando-se constantemente para enviar informação por redes dedicadas e até pela internet.

É o engenheiro eletricista, o engenheiro eletrônico ou da computação, além de outros profissionais correlatos os naturais desenvolvedores destes produtos, já que não se pode desprezar aspectos ligados ao *hardware* como compatibilidade eletromagnética,

confiabilidade do sistema em condições extremas de temperatura e até a análise de viabilidade econômica e busca da melhor seleção entre diversas opções de microcontroladores/processadores e outros componentes dedicados, além de dispositivos de lógica programável. No entanto este profissional deve transcender o *hardware* tão somente e dominar, ao menos no nível de escolha entre diversas opções, as arquiteturas de *software* e sistemas operacionais de tempo real disponíveis.

Este projeto se justifica, pois, foca em uma das áreas próprias de nossa atuação, além de contribuir para o melhor entendimento das alternativas de uso e características chaves para permitir a tomada de decisões de projeto e entender as vantagens e desvantagens da adoção de sistemas operacionais no ligar da programação diretamente escrita para o *hardware*.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Testar a aplicação de sistemas operacionais de tempo real no projeto de sistemas embarcados, visando melhor compreender os requisitos exigidos por tais sistemas, bem como sua destinação apropriada para produtos.

1.2.2 Objetivos específicos

Investigar os requisitos chave para a escolha de um RTOS para um projeto.

Avaliar o ganho de tempo de reaproveitamento de código obtido por um mesmo sistema em mais de um modelo de equipamento.

Levantar os requisitos de sistemas embarcados comumente exigidos e novos projetos.

Averiguar a conformidade entre as funcionalidades e recursos disponibilizados pelos sistemas e os requisitos levantados.

1.3 METODOLOGIA E ESTRUTURA DO TRABALHO

Este trabalho será desenvolvido apoiado em uma pesquisa bibliográfica para apresentar os conceitos básicos de sistemas embarcados e sistemas operacionais de tempo real aliado a caráter de acompanhamento in loco da aplicação dos conceitos em projeto ilustrativo proposto. Serão feitas observações que permitam a aferição de informações e conclusão de utilidade para outros trabalhos no mesmo ramo de conhecimento ou correlatos.

Inicialmente será feita uma consulta a bibliografia que dá base teórica a estes sistemas assim como materiais de divulgação e artigos da comunidade de projetistas e desenvolvedores de tais sistemas.

Será feita uma primeira delimitação das opções para RTOS e as plataformas de *hardware*, buscando a viabilização do estudo exploratório e da análise empírica. Se isto não fosse realizado, poderíamos esbarrar em dificuldades relacionadas ao fornecimento de equipamentos e componentes eletrônicos assim como de licenças de *software* e do próprio sistema operacional, em alguns casos. Esta delimitação é importante tendo em vista a grande variedade tanto de sistemas operacionais, mesmo limitando-se aos de “tempo-real”, assim como de plataforma de *hardwares* e kits de desenvolvimentos (fornecidos pelos fabricantes como um ponto de partida para projetos de produtos utilizando seu microcontrolador ou microprocessador) e produtos destinados ao ecossistema *maker* (em que se observa acentuado caráter de *hobby* ou voltado ao desenvolvimento rápido de produtos em *startups*).

Uma vez escolhido o sistema operacional e as plataformas de *hardware*, será proposto um problema ilustrativo, que possibilite a adoção de um sistema operacional e tempo real como parte da própria metodologia de desenvolvimento de produto eletrônico, com sistema embarcado. As funcionalidades que serão desenvolvidas com o objetivo de atender a solução do problema específicos buscarão fazer o uso mais adequado possível dos recursos oferecidos tanto pelo *hardware* quanto pelo RTOS, levando em conta os requisitos do projeto que serão estabelecidos.

A elaboração destes requisitos de projeto são parte constituinte da própria descrição do problema ilustrativo, mas também visam direcionar e controlar o desenvolvimento do produto para que os objetivos deste trabalho possam ser atendidos. Estes requisitos não serão detalhados, mas sim gerais, atendo-se ao que tem valor e deixando os detalhes para serem decididos durante o desenvolvimento.

Embora não tenha sido colocado com um objetivo, procuraremos utilizar conceitos de metodologias ágeis do mundo do *software*, procurando adaptá-las tanto para o *software* embarcado quanto para o *hardware*.

Destaca-se para isso o próprio objetivo principal de testar a aplicação dos sistemas operacionais de tempo real no projeto de sistemas embarcados, mas também busca-se garantir a possibilidade de reaproveitamento de código entre as implementações para duas plataformas (ou modelos de equipamento) e o próprio ganho de tempo com este reaproveitamento.

Durante o período de trabalho prático, no qual se fará tanto o projeto de *hardware* dos protótipos e das versões de *software* embarcado para cada um deles, (incluindo a programação e testes de funcionamento). Pretende-se ao final ter desenvolvido, ao menos até a etapa de um protótipo funcional, um novo produto com duas plataformas de *hardware* e *software* escolhidas, porém com certa porção de código-fonte compartilhado entre as duas implementações.

Será registrada memória da evolução do projeto, com ênfase na sequência e nos passos desenvolvidos para escrever o código fonte da solução. Desta forma espera-se acentuar o caráter empírico através da definição e adoção de técnicas de desenvolvimento de *software*, seguida da avaliação daquilo que foi observado por experiência própria, alicerçada no relato da própria realidade do desenvolvimento, buscando registrar as vantagens e desvantagens de determinadas estratégias adotadas assim como (e principalmente) a prévia escolha do uso do RTOS.

Por fim serão avaliados os resultados obtidos. Esta avaliação será feita com foco no cumprimento dos requisitos definidos para o problema ilustrativo e nos objetivos do trabalho, a fim de verificar tanto a efetividade da implementação quanto a utilização a adequação do uso de um RTOS para a solução do problema escolhido.

2 REVISÃO DE LITERATURA

Apresenta-se na sequência a revisão literária, mostrando aqueles conhecimentos básicos que dão sustentação e contextualização ao presente trabalho de conclusão de curso.

2.1 SISTEMAS

Um conceito amplamente utilizado em nosso campo, mas também no dia a dia é o conceito de sistema, palavra cuja origem etimológica vem “o latim *systema*. Isto é: *sy* (junto) + *sta* (permanecer). *Sýstema* é aquilo que permanece junto. ” (Dicionário Etimológico, 2008).

Segundo Weckowicz (2018) a teoria geral dos sistemas, conhecida por T.G.S. foi elabora pelo biólogo Ludwig Bertalanfy e tem o objetivo de apresentar teorias e conceitos para aplicação empírica com utilidade em várias áreas de conhecimento, já que o comportamento sistêmico seria observado em diversos campos e é destacada a importância do conhecimento das iterações do todo e não somente de partes. Seguindo nesta escola, podemos apresentar o conceito de sistema: “O Sistema é um conjunto de partes interagentes e interdependentes que, conjuntamente, formam um todo unitário com determinado objetivo e efetuam determinada função” (OLIVEIRA, 2002, p. 35)

Como podemos perceber a definição de sistema é ampla e abrange a descrição da interligação entre as partes em vistas de um objetivo bem definido. Dado este entendimento, precisamos descrever dois tipos específicos de sistemas, nos domínios da área da engenharia de *software* e da engenharia eletrônica, respectivamente: Sistemas Embarcados e sistemas Operacionais (especificamente os de tempo real).

2.1.1 Sistemas Embarcados

Costa (2008) define sistemas embarcados (ou sistema embutidos) como um sistema onde o computador é completamente dedicado ao sistema que ele controla, ou seja, a uma finalidade específica. Diferencia ainda este em relação a outros sistemas como computadores e supercomputadores, já que, ao contrário destes, realiza tarefas pré-definidas, de forma dedicada e geralmente com requisitos específicos.

Com foco no circuito eletrônico, ou seja, o *hardware* do sistema embarcado, Mehl (2011 p3) destaca que:

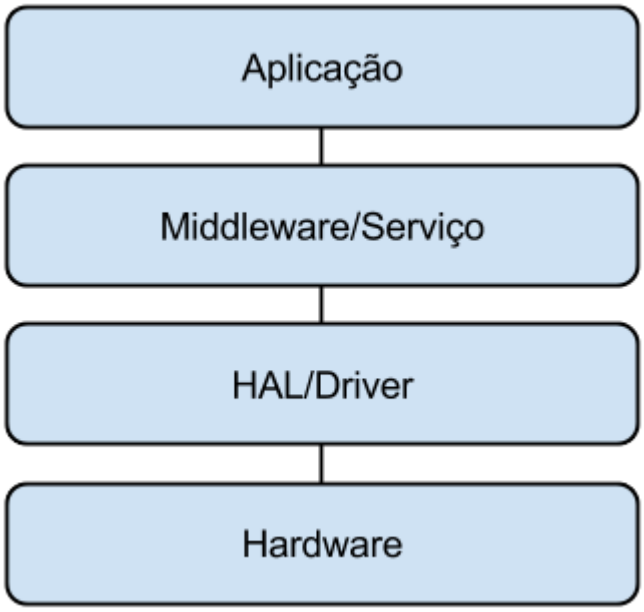
a) o circuito eletrônico é otimizado para realizar apenas aquela tarefa;

b) redução do tamanho;
c) redução dos recursos computacionais;
d) redução do custo final;
e) aumento da confiabilidade.

Como consequência do *hardware* específico e limitado “o *software* para um sistema embarcado é decisivo. Um programa de *software* embarcado deve executar dentro de parâmetros de resposta muito específicos e é limitado na quantidade de espaço que pode executar.” (Null e Lobur, 2009).

Os conhecimentos de arquitetura de *software* também devem ser usados nos projetos de tais sistemas, como nos ensina Rossi (2015). Ele propõe, entre os padrões arquiteturais disponíveis, o sistema em camadas, que organiza hierarquicamente os módulos de *software* em camadas, de forma que cada camada seja cliente da camada abaixo dela e fornecedor de serviços para a camada acima, como pode ser visto na Figura 1.

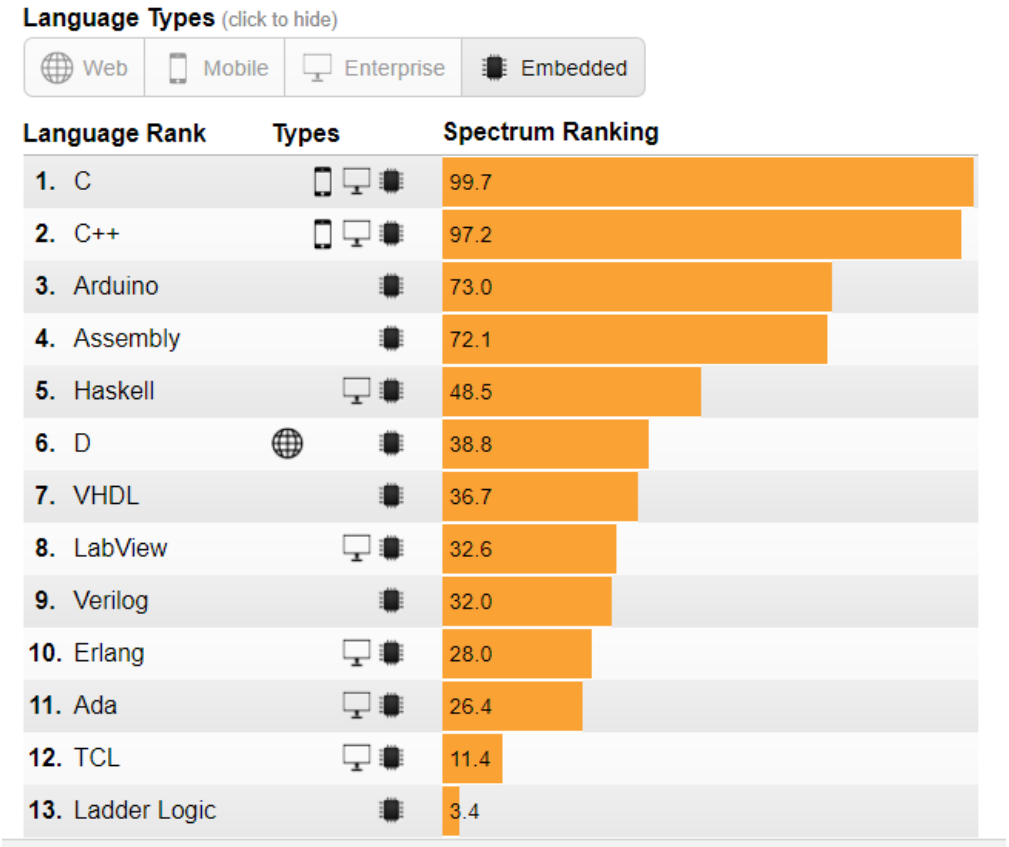
Figura 1 – Exemplo de sistema embarcado em camadas



Fonte: Rossi (2015).

Na categoria do desenvolvimento do *software* do sistema embarcado, também chamado de programação embarcada ou *firmware*, as linguagens mais populares em 2017 são, segundo o *ranking* da IEEE Spectrum (IEEE Spetrum, 2018) são C e C++. Na Figura 2 podemos ver o *ranking* completo.

Figura 2 – Spectrum Ranking – Filtro para linguagem de sistemas embarcados



Fonte: IEEE Spectrum (2018).

2.1.2 Sistemas Operacionais

Estamos habituados em nosso dia a dia a utilizar, como usuários de microcomputadores sistema operacionais multitarefa. Um sistema operacional pode ser entendido sob o aspecto de uma máquina virtual, como nos explicam Machado e Maia (2011, p 4):

> O computador pode ser compreendido como uma *máquina de camadas* ou *máquina de níveis*, em que inicialmente existem dois níveis: *hardware* [...] e Sistema Operacional [...]. desta forma, a aplicação do usuário interage diretamente com o sistema operacional, ou seja, como se o *hardware* não existisse. Essa visão modular e abstrata é chamada de máquina virtual.

O sistema operacional também pode ser visto sob a ótica do gerenciamento de recursos. Segundo Tanenbaum e Woodhull (2000, p. 19) nesta visão o sistema operacional deve monitorar e controlar o acesso de diversos programas (ou usuários) a recursos como o processador, regiões da memória, arquivos armazenados e interfaces de comunicação,

impedindo o caos resultante de acessos concorrente a um recurso, como uma impressora (que poderiam imprimir trechos embaralhados enviados por mais um programa simultaneamente). Para isso o sistema deve gerenciar a alocação dos recursos de forma em um ambiente com vários programas e usuários conectados.

Em todas estas situações vários processos (ou programas) concorrentes tentam acessar um mesmo recurso, mas também pode acontecer, que um processo precise de dados que são gerados por um outro processo. É tarefa do sistema operacional garantir que os dados não tentem ser acessados antes da hora, ou seja, antes que o processo responsável os tenham gerados. Este é o caso típico de tratamento de entrada e saída, em que vários processos diferentes têm participação na transferência de dados (SHAY, 1996 p. 19).

Desta forma temos, entre os sistemas operacionais multitarefa, uma classificação quanto ao gerenciamento de suas aplicações, que é mostrada na Figura 3.

Figura 3 – Tipos de sistemas multiprogramáveis/multitarefas

Fonte: Adaptado de Machado e Maia (2011, p. 8).

Os sistemas *batch* ou de programas em lote foram muito utilizados entre as décadas de 1950 e 1970[1], onde todos os comandos do programa eram armazenados em cartões

[1] Embora seu uso seja raro atualmente, os sistemas operacionais dão suporte a estes tipos de programas, armazenados em arquivos de lote, e de fato são usados de forma útil para automatizar comandos para o sistema operacional que precisariam ser executados um a um em um prompt de comando.

perfurados e o sistema operacional lia-os um a um e executava-os apresentando o resultado ao fim da operação. (SHAY, 1996 p. 12).

Os sistemas de tempo compartilhado são aqueles em que diversos programas são executados em fatias de tempo (*time-slice*) reservadas a cada um dos programas, que executa como se fosse o único programa, porém o processador dedica-se a ele por um intervalo definido de tempo, para em seguida passar a outro programa, deixando-o no aguardo de uma nova fatia de tempo para si. Estes sistemas são os mais utilizados, inclusive para aplicações comerciais, pois tem tempos de respostas razoáveis aos usuários e custos mais baixos (Maia, 2011, p. 9).

Param Kevin (2017) nos sistemas operacionais de tempo real ou RTOS (*Real time Operation System*) o termo tempo-real (ou Real Time) indica que a resposta do sistema deve ser rápida. Este tipo de sistema operacional será melhor conceituado a seguir.

2.1.3 Sistemas Operacionais de Tempo Real

Estes tipos de sistemas costumam ser desenvolvidos para microcontroladores ou seja, para sistemas embarcados e são bem pequenos se comparados aos que rodam em computadores, já que estes não possuem muito espaço para armazenar os programas.

Sobre o imperativo da resposta rápida nestes sistemas, Tanenbaum e Woodhull destacam que se referem a entrada de estímulos provenientes de um ou mais dispositivos externos, que demandam uma resposta apropriada do sistema dentro de uma quantidade de tempo fixo. A resposta deve ser dada rapidamente, já que receber a resposta certa muito tarde é com frequência tão ruim como não receber resposta alguma. São exemplos a conversão dos bits de um fluxo de dados para conversão em música, os sistemas de monitoramento de pacientes em uma UTI, o piloto automático de uma aeronave e o sistema de controle de segurança de um reator nuclear. (TANENBAUM E WOODHULL, 2000 p.74).

O foco nos RTOS rodando em microcontroladores é especialmente útil quando pensamos em muitos sistemas embarcados que não usam computadores e sim circuitos eletrônicos baseados nesta tecnologia. Segundo Kevin (2017), como consequência da falta de espaço para armazenar *software* nos microcontroladores, estes sistemas diminuem seu escopo, oferecendo um escopo mínimo, porém avançado, suportando pelo menos suporte a encadeamento, agendamento e monitoramento de diversas tarefas em sistemas pequenos.

Costumam ser acoplados a aplicação com o objetivo de decidirem qual a próxima tarefa que deve executar, de acordo com suas prioridades, permitir a troca de mensagens entre tarefas[2].

Neste contexto podemos perceber que este tipo de sistema operacional prioriza o aspecto de gerenciamento de recursos em detrimento do aspecto de máquina virtual, já que a primeira atribuição é prioridade e previne a ocorrência de situações caóticas, como o envio de trabalhos de impressão simultaneamente recebidos de três fontes (como três programas diferentes) como exemplificado por Tanenbaum e Woodhull (2000, p. 19), que acrescenta:

> Quando um computador (ou uma rede) tem múltiplos usuários, a necessidade de gerenciar e de proteger a memória, os dispositivos de E/S e outros recursos é ainda maior, uma vez que os usuários talvez interfiram um no outro. Além disso, os usuários frequentemente necessitam não só compartilhar *hardware*, mas também as informações[...]. Em resumo, essa visão do sistema operacional sustenta que sua tarefa primária é monitorar quem está utilizando qual recurso, atender requisições de recursos. Medir a utilização dos recursos e medir as requisições conflitantes de diferentes programas em usuários.

Em se tratando dos sistemas embarcados é fácil perceber que o que se disse sobre diversos usuários e programas, aplica-se não só para ações diretas do usuário (como pressionar um botão ou tocar a tela, como de fato pode ocorrer em um sistema embarcado) mas também a chegada de dados por um canal de comunicação e leituras de dados de diversos sensores, assim como a concorrência por recursos como memória e interfaces de entrada e saída dentro de uma mesma aplicação, por diferentes partes do programa do sistema embarcado.

Estes sistemas possuem algumas características inerentes, como destaca o V SEGeT (2018):

a) são utilizados diversos processos concorrendo entre si;
b) é necessário escalonar os processos e estes processos devem ter prioridades determinadas para cada um deles;
c) existe a necessidade de comunicação entre os processos;
d) é necessário que exista uma sincronização entre os processos (sincronismo);
e) a necessidade de sincronismo também existe em relação aos eventos externos;

[2] Os termos programa, processos e tarefa (e thread) são utilizados dependendo do autor ou contexto, podem ser entendidos de forma equivalente para os propósitos de nosso estudo.

f) suas implementações levam em conta a possibilidade de uso com grande variedade de arquitetura, como microcontroladores de 8-bits, 32-bits e computadores;
g) existe forte interação com o ambiente externo.

Diante destas características são as funções de gerenciamento de recursos oferecidos por um RTOS que permitem gerenciar e impedir estes conflitos de recursos, assim como possibilitar que isto seja feito respeitando requisitos de tempo. A seguir veremos os principais recursos oferecidos por um típico sistema operacional de tempo real e sua importância no gerenciamento de recursos.

2.1.4 Recursos oferecidos por um RTOS

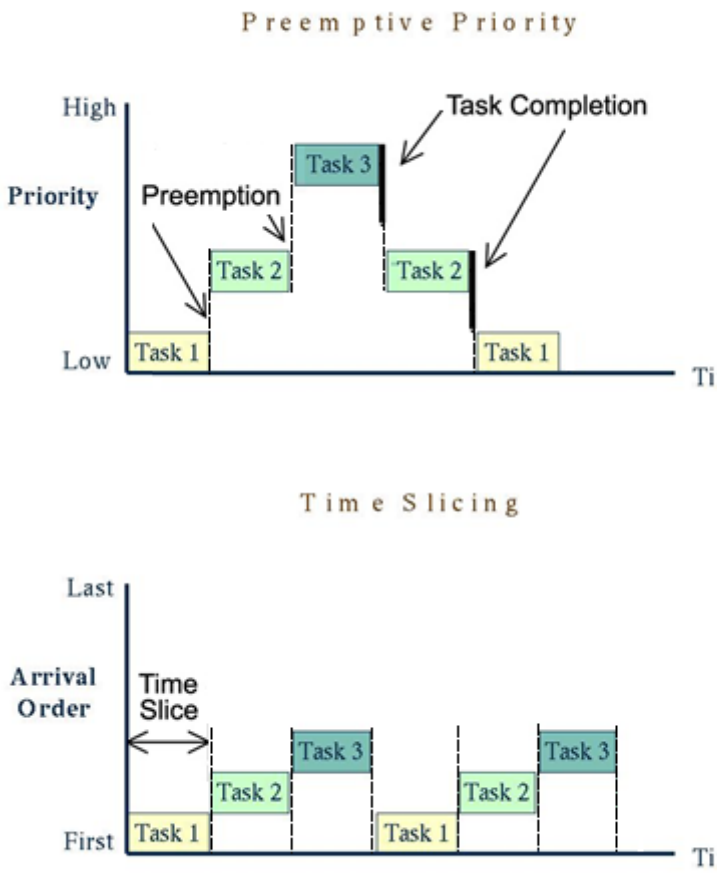
2.1.4.1 Escalonamento

Os processos ou tarefas, que executam sobre a CPU, podem compartilhar este recurso segundo duas arquiteturas típicas (WIKIPEDIA, 2018):

a) baseado em eventos ou escalonamento prioritário: O escalonador só muda a tarefa que está em execução caso uma outra tarefa precise ser executada e ela possua uma prioridade maior que a tarefa atual. Esta troca de execução devido á maior prioridade denomina-se preempção;
b) baseado em partilha de tempo: O escalonador muda a tarefa em execução em períodos predefinidos, ou seja, em fatias de tempo baseadas no em tiques do relógio do processador.

Na Figura 4 podemos ver um exemplo com três tarefas sendo executada primeiramente pelo método de escalonamento prioritário (*preemptive priority*) e depois segundo a partilha de tempo (*time slicing):*

Figura 4 – Tipos de escalonamento



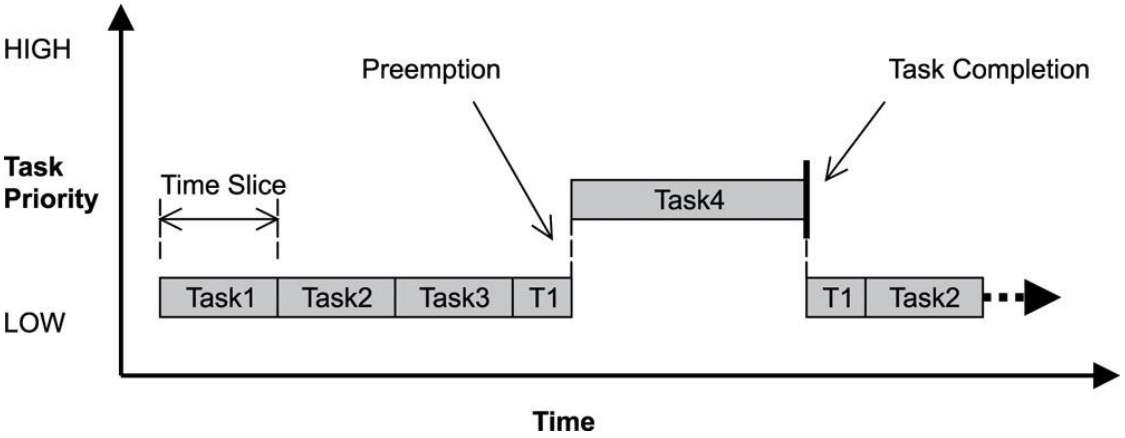
As tarefas de maior prioridade podem interromper (preempção) tarefas de menor prioridade. Neste exemplo percebemos que a tarefa "Task 3" passou a executar, fazendo com quem "Task 2", que até então executava, fosse interrompida.

Várias tarefas são executadas sequencialmente até a conclusão. A cada tarefa executada é atribuída a uma quantidade de tempo fixo, chamada *time slice*, durante o qual executa.

Fonte: Adaptado de *Universadade of Saskatchewan* (2018)

As tarefas também podem se alternar na execução em fatias de tempo (*time slice*) enquanto todas as tarefas em condições de serem executadas possuírem a mesma prioridade ou ocorrer uma preempção apenas se necessário, quando uma tarefa de maior prioridade passar a estar apta a ser executada, como pode ser ver na Figura 5.

Figura 5 – *Time slice* e preempção



Fonte: http://www.embeddedlinux.org.cn (2018)

2.1.4.2 Exclusão mútua

É usual, em programação concorrente utilizando diversas tarefas, que dados precisem ser compartilhados, porém de forma exclusiva. Ou seja, quando a área de dados compartilhada está sendo acessada por um processo, todos os outros que precisam acessar esta área terá que esperar o primeiro processo terminar para só então outro processo poder utilizá-la. V SEGeT (2018).

2.1.4.3 Semáforo

É um mecanismo de sinalização entre as *tasks* (e interrupções) que não carrega nenhuma informação além da própria sinalização. Assim a sinalização está implícita no estado do próprio objeto de semáforo, de forma que mais de uma sinalização com significado diferente precisa de mais de um semáforo. Sua utilização mais simples, a do semáforo binário, pode ser entendida geralmente como uma chamada a uma função do RTOS para "pegar" ("*take*") o objeto (semáforo) que, se não estiver sinalizado ainda, retorna imediatamente e sinaliza o semáforo. Caso já esteja sinalizado, vai bloquear a tarefa, até que a outra tarefa que "pegou" o semáforo o "devolva" ("*gives*"), ou seja, encerra a sinalização. A implementação da exclusão mútua se dá justamente por um semáforo binário, porém com um contexto de seção crítica em que a operação de "pegar" ocorre imediatamente antes de iniciar o uso do recurso que somente uma *task* pode usar por vez (excluindo as demais) e após concluído o

código que acessa o recurso a operação deve devolver". Finalmente temos os semáforos de contagem, que possuem um contador interno maior que um (como pode ser entendido um semáforo binário) usados para controlar quantos itens de um recurso limitado ainda podem ser "pegos" com sucesso. Quando o recurso for "pego" o contador diminui e se uma tarefa tentar "pegar" o semáforo quando o contador chegou a zero a operação falha e o retorno de erro deve ser tratado com espera indefinida ou com um timeout e posterior tratamento do resultado com falha na operação. (PERCEPIO, 2018).

2.1.4.4 Filas

As filas são mecanismos que permitem comunicação entre tarefas, entre interrupções e tarefas, de tarefas para interrupções e podem armazenar uma quantidade finita e fixa de itens com um tamanho determinado. Geralmente são usadas como *buffer* do tipo FIFO (*First In First Out*) de forma que a informação é adicionada no final da fila pela tarefa ou interrupção que deseja enviar a informação e retirada no seu início, embora também possa ser possível escrever no início da fila e sobrescrever o último (ou único) item de uma fila (BARRY 2016, Pag. 103).

2.1.5 Sistemas operacionais de tempo real com código fonte aberto ou disponível

A Wikipédia lista uma grande quantidade de RTOS em seu artigo especializado no tempo, sendo que destes, os que são classificados como *open source* ou *souce available*, por ser um critério mais atrativo para uso acadêmico e em um projeto, já que a possibilidade do código fonte permite entender melhor o funcionamento. Também foram selecionados os RTOS destinados para uso em sistemas embarcados e que ainda estão ativos, podemos obter a Tabela 1[3]:

[3] A tabela no site possui 191 itens, que foram copiados e colados em uma planilha eletrônica, onde foi feito o seguinte filtro: **"*Source Model*"** igual a "*open*", "*open source*", "*source available*", "*source code & Design Assurance Pack available*", "*source code available*" ou "*source code provided*", **"*Target uses*"** contém "*embedded*" e **"*Status*"** igual a "*active*", resultando nos 108 itens apresentados na tabela. Para tornar mais compacta ainda a apresentação da lista ocultamos as colunas filtradas e a coluna com o endereço da web site do fornecedor. A coluna plataforma foi reduzida para caber sempre em uma linha, adicionando a expressão "e mais n plataformas.", indicando em "n" quantas plataformas deixaram de ser listadas, em relação a lista original.

Tabela 1 – Lista resumida de RTOS com código aberto ou disponível

Nome	***Platformas***
FX-RTOS	ARMv6-M (Cortex-M0/M1), ARMv7-M (Cortex-M3), e mais 6 plataformas.
Nucleus RTOS	ARM (Cortex-M3-M4-R4-R4F-A8-A9, ARM7-9-11) e mais 5 plataformas.
OS21	ST40/SH4,ST200,ARM
Portos	(ARM soon)
RODOS	ARMv7 (M3, A8, A9), AVR32, PowerPC 405, sparc64
LynxOS	Motorola 68010, x86/IA-32, ARM, Freescale PowerPC e mais 2 plataformas.
OpenRTOS	Same as FreeRTOS
SafeRTOS	Same as FreeRTOS
RTXC Quadros	ARM - Atmel/Freescale/NXP/ST/TI, Blackfi e mais 6 plataformas.
RTX Keil Real-Time Operating System	ARM
T-Kernel	ARM, MIPS, SH, more
Neutrino	ARM, MIPS, PPC, SH, x86, XScale
eCosPro	ARM7-9, Cortex-A5-A9-M3-M4-M7, 680x0-ColdFire e mais 12 plataformas.
uKOS	Cortex-M3-M4-M7-H7, 6833x, PIC, CSEM icyflex-1, STM32
Atomthreads	AVR, STM8, ARM, MIPS
BeRTOS	ARM, Cortex-M3, ARM ARM7TDMI, Atmel AVR, e mais 3 plataformas.
BOOS Core	ARMv5TEJ (TI AM18x), DSP (TI TMS320C64x)
BRTOS	Freescale Kinetis (ARM Cortex-M4) e mais 11 plataformas.
CapROS	IA-32, ARM9
ChibiOS/RT	x86, ARM7, ARM9, Cortex-M0-M3-M4 e mais 7 plataformas.
CoActionOS (now Stratify OS)	ARM Cortex-M3, LPC17xx
cocoOS	AVR, MSP430, STM32
Contiki	MSP430, AVR, ARM
CooCox CoOS	STM32, NXP LPC1000, TI LM3S8962 e mais 2 plataformas.
distortos	ARMv6-M architecture e ARMv7-M architecture
dnx RTOS	ARM Cortex-M3, STM32
DuinOS	AVR (Arduino)
eChronos	ARM Cortex-M3 - M7
eCos	ARM-XScale-Cortex-M, CalmRISC, 680x0-ColdFire, fr30 e mais 10 plataformas.
Embox	ARM, LEON, MicroBlaze, MIPS, PowerPC, x86
Embkernel	ARM Cortex M
ERIKA Enterprise	ARM7, ARM Cortex MX, Hitachi H8, Altera Nios2 e mais 11 plataformas.
Femto OS	AVR
FreeOSEK	Cortex-M4, MIPS, MSP430, SPARC LEON3
FreeRTOS	ARM, AVR, AVR32, ColdFire, HCS12, IA-32 e mais 10 plataformas.
Frosted	ARM Cortex-M
Fuchsia	?
FunkOS	AVR, MSP430, Cortex-M3
Helium	HCS08, AVR
Hybridthreads	Xilinx Virtex-II Pro ML310, Xilinx Virtex-II Pro XUP
IntrOS	ARM Cortex-M, STM8, AVR8
ISIX	Cortex-M3
iRTOS	AVR, ARM7
MaRTE OS	IA-32
mbed-rtos	Cortex-M, Cortex-R
MenuetOS	IA-32

Milos	Cortex-M3
miosix	stm32, efm32 e LPC2000
mipOS	Cortex-M3, ST7, STM8, x86
Microsoft Invisible Computing (MMLite)	ARM7-9, i386, h8, mips, Trimedia, eCog1
Nano-RK	AVR, MSP430
nOS	AVR, MSP430, Cortex-M0-M3-M4, M16C, RX600, PIC24, Win32, POSIX, STM8
Nut/OS	AVR, AVR32, ARM7, ARM9, Cortex-M3
NuttX	Linux user mode, ARM7-9 e mais 11 plataformas.
OpenEPOS	ARM, x86, AVR, MIPS, PowerPC
OSA	PIC10-PIC24, AVR 8-bit, STM8
PICOS18	PIC18
picoOS	6502, 80x86, ARM7, AVR, PowerPC,Cortex-M,MSP430,PIC32
Piko/RT	ARM Cortex-M3/M4
Pharos	ARM926 with MMU, Cortex-R5 and Cortex-M4 with MPU
POK	x86, PowerPC, SPARC
RIOT	ARM, MSP430, AVR, x86
RTEMS	ARM, Blackfin, ColdFire, TI C3x/C4x, H8/300, x86 e mais 10 plataformas.
RT-Thread	ARM, ARM Cortex-M0-M3-R4-M4-M7, IA-32, AVR32 e mais 7 plataformas.
scmRTOS	ARM, Cortex-M3, Blackfin, MSP430, AVR, STM8
SDPOS	ARM, Cortex-M3, Blackfin, PIC18, PIC24, i386 win32/linux synthetic targets
silRTOS	ARM, Cortex-M3 Cortex-A9 ARM-11MP, Blackfin, MIPS, AVR
SOOS Project	H8/300
StateOS	ARM Cortex-M, STM8
Stratify OS	ARM Cortex-M3, M4
TizenRT	ARM
TI-RTOS Kernel (SYS/BIOS)	Mostly Texas Instruments: MSP430-432, C2000-5000-6000, TI's ARM families (Cortex M3-4F-R4-A8-A15), SimpleLink Wireless CC2xxx-CC3xxx
Tock OS	ARM Cortex
Trampoline Operating System (OSEK and AUTOSAR)	AVR, H8-300H, POSIX, NEC V850e, ARM7, Infineon C166, HCS12 or PowerPC
TNKernel	ARM, PIC24-32-dsPIC, HCS08, STM32 (Cortex-M3)
TNeo	Cortex-M0-M0+-M1-M3-M4-M4F, PIC24-32-dsPIC
XMK	?
Y@SOS	Cortex-M3, STM32
MontaVista Linux	?
uOS	AVR, ARM, MIPS32, MSP430, Intel i386, Linux386
Zephyr	x86, ARM, ARC, NIOS2, XTensa, RISC V 32
Apache Mynewt	ARM Cortex-M, MIPS32, Microchip PIC32, RISC-V
VxWorks	ARM, IA-32, Intel 64, MIPS, PowerPC, SH-4, StrongARM, xScale
UNOS	680x0

Fonte: Adaptado de: https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems (2018)

A lista com todas as colunas originalmente disponível, assim como, todas as plataformas de cada RTOS pode ser encontrada no Anexo A.

2.2 PLATAFORMAS PARA DESENVOLVIMENTO

Embora os sistemas embarcados conceitualmente tenham apenas o *hardware* necessário, de forma que seja otimizado para a tarefa a que se destinam, como lembrado por Mehl (2011, p3) pode-se decidir, durante a fase de desenvolvimento do produto, por outras opções de *hardware*, que não sejam projetar o equipamento desde o início, para só então dar início a uma segunda fase, em que se inicia a programação do sistema embarcado.

> Na fase de concepção de projeto, o uso de *kits* comerciais para desenvolvimento de produtos é um aliado muito importante no que tange a experimentação de uma nova tecnologia e a validação e prova de conceito de determinados componentes ou soluções. Isso porque permite que essas experiências sejam feitas de modo mais rápido do que seria desenvolver o projeto do zero nessa fase. (SUEIRO, SOUZA, et al. 2015).

A Figura 6 ilustra a grande variedade de KITs de desenvolvimento comerciais disponíveis no mercado.

Figura 6 – Kits comerciais para de desenvolvimento de produtos

Fonte: Embarcados.com (2018)

Segundo os autores, tem-se optado na indústria em utilizar os *kits* como parte constituindo do próprio produto, e não somente como ferramenta útil para experimentar a tecnologia e o conceito, especialmente quando as condições não exigem requisitos extremos como temperatura e vibração. Estes kits costumam ser acompanhados de bibliotecas de auxílio validadas pelas equipes de engenharia e pela comunidade de desenvolvedores, o que ajuda e economizar tempo também no desenvolvimento do *software* embarcado. Mesmo que por fim não se utilize os *kits* no produto final, as principais vantagens de sua utilização são apontadas:

a) o tempo de desenvolvimento do projeto é reduzido;
b) aplicação final com melhor qualidade;
c) eliminação da complexidade da comunicação (entre sistemas compatíveis);
d) a compatibilidade com outros dispositivos que seguem protocolos padrões usados pelos kits;
e) possibilidade de portabilidade com aplicativos de arquitetura similar a oferecida pelo kit de desenvolvimento.

2.2.1 *KIT* de desenvolvimento de fabricante ST

Um exemplo de kits oferecido pelo fabricante são os *kits Discovery* para a linha STM32F411 da STMicroeletronics, que segundo o fabricante tem o objetivo de ajudar a desenvolver os aplicativos rapidamente, oferecendo tudo o que é necessário para se começar logo tanto iniciantes como usuários experientes (STMicroeletronics, 2018). A Figura 7 sintetiza as principais características deste KIT.

Figura 7 – Principais características do kit STM32F411E-Discovery

Key Features

- STM32F411VET6 microcontroller featuring 512 KB of Flash memory, 128 KB of RAM in an LQFP100 package
- On-board ST-LINK/V2 with selection mode switch to use the kit as a standalone ST-LINK/V2 (with SWD connector for programming and debugging)
- Board power supply: through USB bus or from an external 5 V supply voltage
- External application power supply: 3 V and 5 V
- L3GD20: ST MEMS motion sensor 3-axis digital output gyroscope
- LSM303DLHC: ST MEMS system-in-package featuring a 3D digital linear acceleration sensor and a 3D digital magnetic sensor
- MP45DT02: ST MEMS audio sensor, omnidirectional digital microphone
- CS43L22, audio DAC with integrated class D speaker driver
- Eight LEDs:
  - LD1 (red/green) for USB communication
  - LD2 (red) for 3.3 V power on
  - Four user LEDs: LD3 (orange), LD4 (green), LD5 (red) and LD6 (blue)
  - Two USB OTG LEDs: LD7 (green) VBus and LD8 (red) over-current
- Two push-buttons (user and reset)
- USB OTG with micro-AB connector
- Extension header for LQFP100 I/Os for a quick connection to the prototyping board and an easy probing
- Comprehensive free software including a variety of examples, part of STM32CubeF4 package or STSW-STM32136 for legacy Standard Libraries usage

Fonte: Adaptado de: http://www.st.com/en/evaluation-tools/32f411ediscovery.html (2018)

Os diversos sensores e periféricos permitem o desenvolvimento de um bom conjunto de funcionalidades, sendo que outras podem ser adicionadas ligando-se circuitos externos nos conectores de expansão. O site do fabricante oferece este *kit* pelo preço de US$15,00 e são disponibilizados todos os dados técnicos necessários, incluindo esquemático, *layout* e lista de componentes da placa.

Existem diversas opções de IDEs (*Integrated Development Environments*) disponíveis, e são listados no site do fabricante dezessete opções, dentre as quais quatro estão ativos e possuem licença livre (*Free*), conforme pode ser visto na Tabela 2:

Tabela 2 – Lista de IDEs disponíveis para os microcontroladores da família STM32

Nome **Descrição**	**Compilador**	**Fornecedor**
SW4STM32 Ambiente de trabalho paraSTM32: IDE Livre para Windows, Linux e OS X	GCC C/C++	AC6
iSYS-winIDEAOpen A plataforma de desenvolvimento de *software* gratuito e ilimitado do iSYSTEM para todos os dispositivos baseados em STM32 Cortex-M	GNU GCC	Isystem
CoIDE CooCox CoIDE, um ambiente de desenvolvimento de *software* gratuito e altamente integrado para MCUs ARM Cortex	GCC C/C++	CooCox
TrueSTUDIO Uma poderosa ferramenta de desenvolvimento integrada baseada em C / C ++ para seus projetos STM32	GCC	Atollic
MDK-ARM-STM32 Ambiente de desenvolvimento de *software* MDK-ARM para MCUs baseados em Cortex-M	Arm C/C++	Keil

Fonte: Adaptado de: www.st.com (Acesso em 12/06/2018)

2.2.2 Plataformas *open source* Arduino

Além dos *kits* e plataformas de fabricantes, pode-se optar por plataformas abertas muito utilizadas no universo *maker*, como o Arduino, que é definido em seu site da seguinte forma:

> "Arduino é uma plataforma eletrônica *open-source* baseada em *hardware* e *software* fáceis de usar. As placas Arduino podem ler entradas – capturadas de um sensor, um dedo em um botão ou mensagens do *Twitter* e transformá-los em uma saída – ligando um motor, acendendo um LED, publicando algo online. Você pode dizer à sua placa o que fazer e enviando um conjunto de instruções para o microcontrolador na placa. Para fazer isso você usa a linguagem de programação Arduino [...]"[4]. (Arduino 2018, tradução nossa).

Uma pesquisa em um fornecedor de equipamentos ilustra a variedade dos produtos Arduino disponíveis no Brasil, com custos entre R$ 22,90 (h) e R$ 249,90 (e), como pode ser visto na Figura 8 e na Tabela 3, a seguir:

[4] A linguagem Arduino é na prática a soma de um compilador C/C++ com bibliotecas padronizadas que abstraem os detalhes dos vários modelos de microcontroladores e demais detalhes dos circuitos eletrônicos das placas Arduino.

Figura 8 – Modelos de placas arduino disponíveis em fornecedor local

Fonte: Adaptado de: www.filipeflop.com (Acesso em 12/06/2018)

Tabela 3 – Especificações e preços dos modelos disponíveis em fornecedor local

Item	Modelo	Microprocessador	*Clock*	*Flash*	SRAM	Valor (R$)
a	Placa Uno R3	ATmega328	16MHz	32KB	2KB	49,90
b	Placa Mega 2560 R3	ATmega2560	16MHz	256KB	8KB	63,90
c	Placa Due	AT91SAM3X8E	84MHz	512KB	96KB	129,90
d	Placa Leonardo R3	ATmega32u4	16MHz	32KB	2,5KB	59,90
e	Placa Mega ADK R3	ATmega2560	16MHz	256KB	8KB	249,90
f	Placa Esplora	ATmega32u4	16MHz	32KB	2,5KB	159,90
g	Arduino Fio Sparkfun	ATmega328p	8MHz	32KB	2KB	159,90
h	Placa Pro Mini	ATmega328p	16MHz	32KB	1KB	22,90
i	Placa Nano V3	ATmega328	16MHz	32KB	2KB	39,90

Fonte: Adaptado de: www.filipeflop.com (Acesso em 12/06/2018)

2.2.3 Módulos de comunicação sem Fio TCP/IP (*ethernet*) ESP8266

Segundo Herbert (2000), muitas escolhas devem ser feitas para selecionar e integrar TCP/IP em um projeto, para que este tenha capacidade de conexão com redes locais ou internet. Os sistemas embarcados herdaram a parte de programação de plataformas maiores, remontando ao *software* original escrito na Universidade da Califórnia em Berkeley e foram sendo adaptados visando maior eficiência, pelos muitos fornecedores de solução para sistemas embarcados, sendo que os cuidados que devem ser tomados são apresentados seguindo a Tabela 4.

Tabela 4 – Critérios para pilhas TCP/IP em sistemas embarcados

Critério	**Explicação**
Gerenciamento de *buffer*	Deve-se utilizar pré-alocados em memória ao invés de alocação dinâmica de memória, em tempo de execução (via malloc).
Temporizadores	O RTOS deve gerenciar todas as temporizações, de forma que a implementação não utilize interrupções de *timer* para controles dos protocolos (gerenciamento de conexões, *timeouts* e esperar por novas tentativas). O uso das interrupções de *timer* independente do RTOS rouba tempo de processamento e causará problemas (de simultaneidade) a execução do RTOS.
Latência	O RTOS por sua vez não deve sobrecarregar as interrupções adicionando latência. As interfaces de manuseio de interrupções devem ser rápidas e determinísticas. A transmissão e recepção dos quadros não deve se tornar mais demoradas em seus tratamentos por interrupções devido ao RTOS. A grande quantidade de chaveamento de contexto e processamento de CPU necessários para lidar com um pacote aumenta a importância do uso de um sistema operacional com o mínimo de tempo de alternância de tarefas.
Concorrência	O gerenciamento de concorrência por recursos deve ser feito por semáforos para proteger os *buffers* de acesso simultâneo devido à concorrência, de forma mais performática possível.
Cópia de dados minimizada	A implementação do TCP / IP deve minimizar a quantidade de cópias de dados. Os dados dentro de cada quadro podem ser mantidos no *buffer*, para que ele não precise ser copiado e copiado novamente pela CPU em cada estágio do protocolo.
Multiplexação de camada de *link*	A implementação do protocolo requer uma estrutura com mecanismos para enfileiramento e gerenciamento de *buffer*. Além disso, os protocolos modernos exigem interfaces de driver de dispositivo mais flexíveis e multiplexação mais flexível. Isso é particularmente verdadeiro quando os protocolos ponto-a-ponto seriais, como o PPP, são estendidos para suportar encapsulamento IP e redes privadas virtuais (VPN).
Largura de banda da CPU	Cada aplicativo de sistema embarcado possui requisitos diferentes para sua pilha TCP / IP. Por exemplo, uma pilha TCP / IP na maioria dos dispositivos de Internet provavelmente não seria considerada em tempo real. Além disso, se a rede for usada para funções de controle e gerenciamento, os requisitos de largura de banda rígida serão bastante baixos. Por outro lado, se o aplicativo for *streaming* de vídeo ou voz, as taxas de pacote mais rápidas qualificariam o aplicativo como um aplicativo em tempo real.

Fonte: Adaptado de: www.embedded.com (Acesso em 12/06/2018)

Como possíveis solução, com suas vantagens e desvantagens o autor sugere as seguintes alternativas:

a) **implementação total de *hardware*:** muitas empresas estão desenvolvendo módulos[5] compactos e completamente independentes que implementam uma interface serial e permitem a sistemas já existentes comunicar via rede como se estivessem conectados via interface física serial, de forma transparente, sem necessidade de adaptar o *software* embarcado.
b) **Implementação de pilha TCP/IP sem RTOS:** envolve adaptar uma pilha no *software* sem as facilidades de um sistema operacional. Os recursos básicos de escalonamento, temporização e alocação de memória teriam que ser providenciados via implementação escrita diretamente para o *hardware*, misturado com o código do sistema embarcado.
c) **integração de terceiros:** com esse método, você compraria um RTOS de um fornecedor e uma pilha TCP / IP de outro.
d) **tudo em um:** com essa opção, você compraria um produto incluído de um fornecedor que possui uma solução completa que inclui um RTOS, ferramentas de desenvolvimento e uma pilha TCP / IP.

Sobre a complexidade de *software* inerente a uma solução com Wi-Fi (o que engloba tudo o que foi exposto sobre a pilha TCP/IP), Mayers, acrescenta:

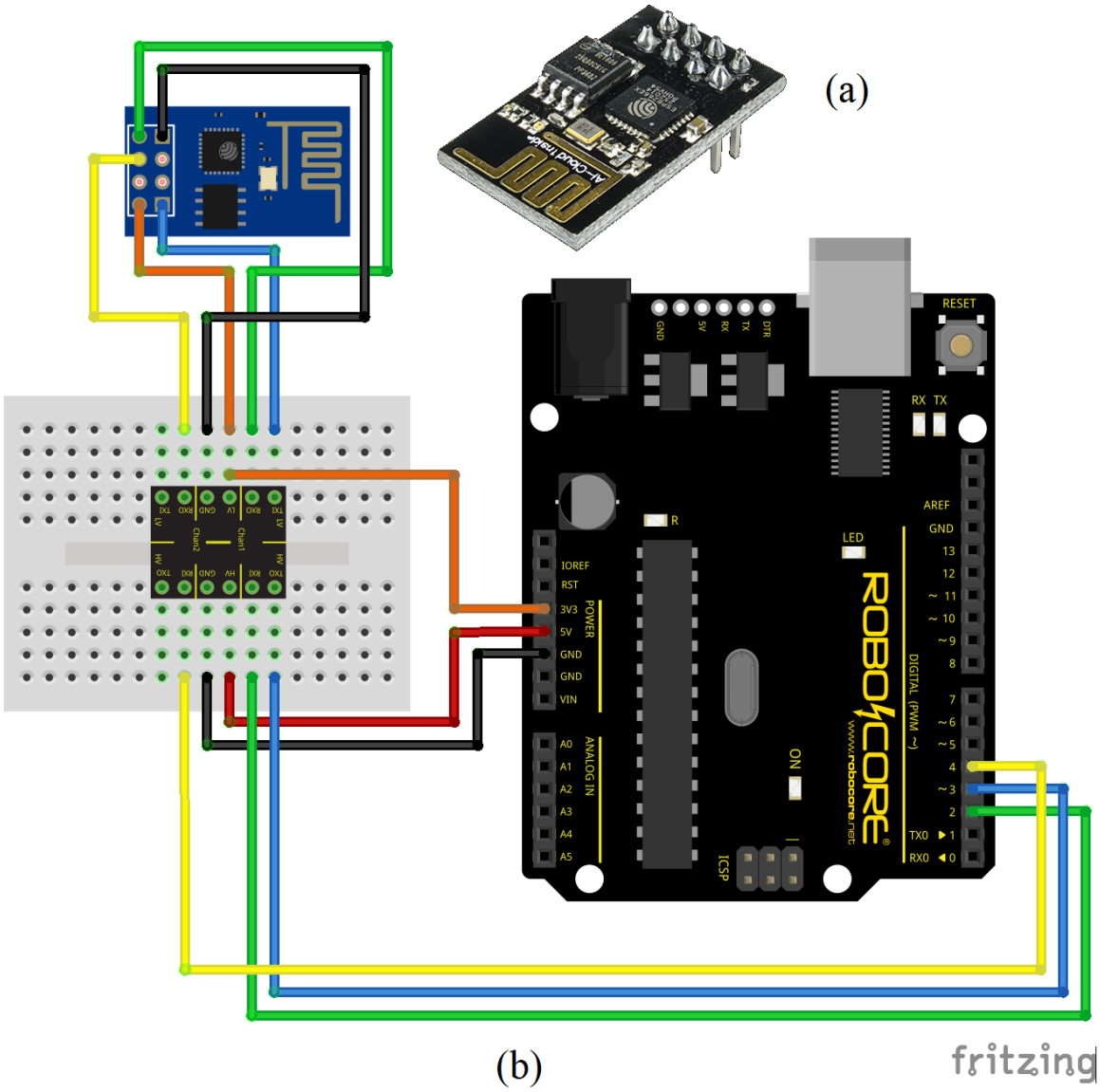
> As soluções tradicionais de Wi-Fi vêm com uma arquitetura de *software* extremamente complexa, que é impensável para um desenvolvedor típico de microcontroladores (MCU). [...] além desse desafio principal de complexidade de *software*, uma solução Wi-Fi para MCUs também precisa ser simples e confiável. Poucos desenvolvedores de sistemas baseados em MCU terão capacidade interna para o projeto de RF e certificações regulatórias. Portanto, ter módulos de RF pré-certificados é fundamental. Além disso, os produtos baseados em MCU geralmente têm um tempo de comercialização mais rápido, portanto ferramentas de desenvolvimento fáceis e uma ampla variedade de aplicativos de *software* de exemplo são muito importantes para garantir que a adição de Wi-Fi não se torne um obstáculo. (MAYERS 2013, tradução nossa)

Um bom exemplo desta destes módulos é o módulo WiFi ESP8266, que segundo o portal RoboCore (2018) afirma em sua loja, é o SoC que pode ser usado para dar acesso a rede Wi-Fi para qualquer MCU, além de permitir a programação, se desejado (o que se aproximaria da segunda alternativa indicada por Herbert). A utilização do módulo por outro microcontrolador se dá pelo envio de comandos AT já que o módulo vem com o firmware pré-programado para este protocolo. Existem muitas informações na internet sobre este módulo e uma gigante comunidade que utiliza este módulo em seus projetos (ver Figura 9).

---

[5] No texto original em inglês foi usado o termo *cards*, no contexto de cartões PCM, uma interface para conexão de *hardware* para *Notebooks*.

Figura 9 – Módulo WiFi ESP8266 a *Espressif* - Informações em uma loja virtual



Um código proposto para verificar se a comunicação está funcionando pode ser o que segue, após gravá-lo na placa e fazer a ligação corretamente, abra o monitor serial com taxa de 9600, coloque o monitor serial no modo *BOTH NL & CR*:

```
#include "SoftwareSerial.h"

SoftwareSerial ESP(2, 3); // RX, TX

const int RST = 4;

void setup(){
  Serial.begin(9600);
  ESP.begin(9600);

  //Pulso em Reset para aceitar comunicacao
  pinMode(RST, OUTPUT);
  digitalWrite(RST, LOW);
  delay(300);
  digitalWrite(RST, HIGH);
}

void loop(){
  if (ESP.available()){
    Serial.write(ESP.read());
  }
  if (Serial.available()){
    ESP.write(Serial.read());
  }
}
```

(c)

(a) Imagem do módulo ESP8266;
(b) As lojas costumam já ensinar a ligar o módulo no Arduino com um protoboard e algum mecanismo de adaptação dos níveis de tensão, já que o módulo opera com 3.3V e a maioria das placas Arduino com 5V;
(c) Também é comum código de exemplo e artigos para auxiliar especialmente quem não tem experiência, a dar os primeiros passos com o módulo.

Fonte: Adaptado de RoboCore. Disponível em https://www.robocore.net/loja/produtos/modulo-wifi-esp8266.html

2.3 METODOLOGIA DE DESENVOLVIMENTO DE SISTEMAS

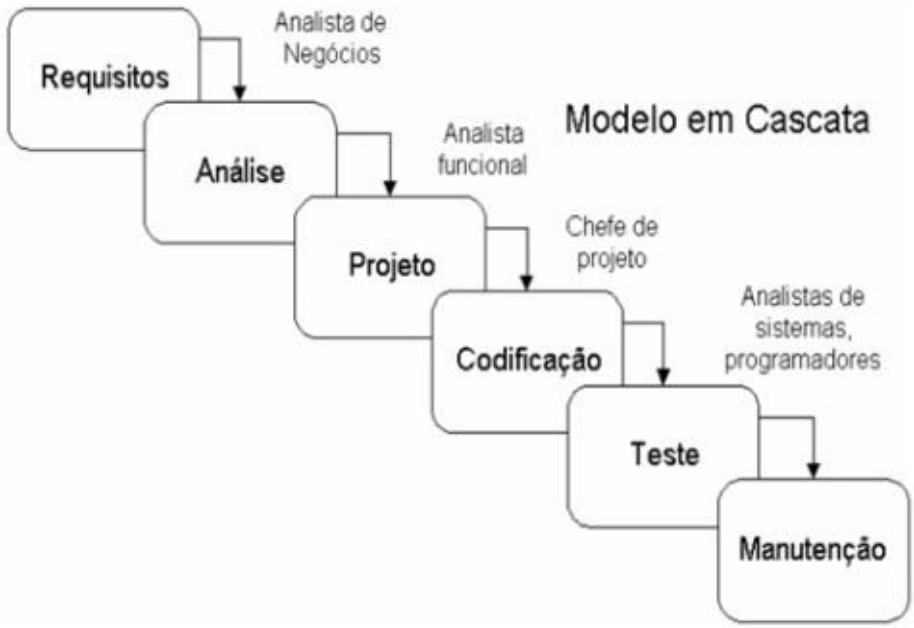
O aumento do uso de computadores comerciais e do desenvolvimento de sistemas para automatizar e auxiliar no gerenciamento e execução dos trabalhos nas empresas fez, segundo Uchôa (2018), com que surgissem metodologias ou modelos de desenvolvimento de *software* destes sistemas, permitindo o desenvolvimento de *software* de grande complexidade. As principais metodologias que surgiram, em constante evolução, foram o modelo em cascata, o modelo iterativo ou incremental e as metodologias ágeis.

2.3.1 Principais metodologias

2.3.1.1 Modelo cascata (ou clássico)

Caracteriza-se por um progresso sequencial entre as fases, sendo a seguinte precedida pelo término da anterior. Pode haver retroalimentação de uma fase para a anterior, mas em uma visão macro, as fases seguem de forma fundamentalmente sequencial. (Uchôa, 2018) conforme pode ser visto na Figura 10. Segundo Farinha (2014) este modelo foi adotado pelo departamento de segurança dos Estados Unidos da América assim como por inúmeras empresas;

Figura 10 – Modelo cascata (ou clássico)

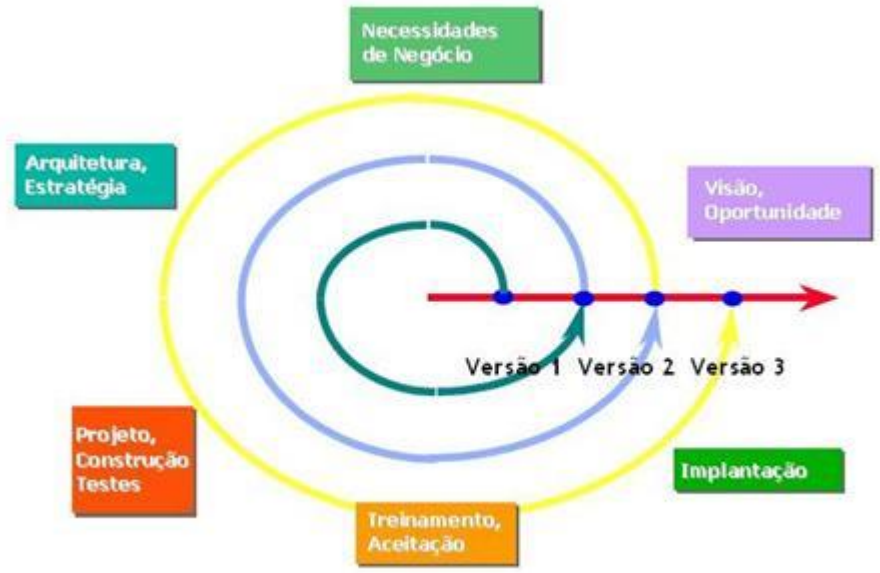


Fonte: Adaptado de: https://sistemasdeinfoige.wordpress.com/2014/10/23/modelo-waterfall/

2.3.1.2 Modelo iterativo ou incremental

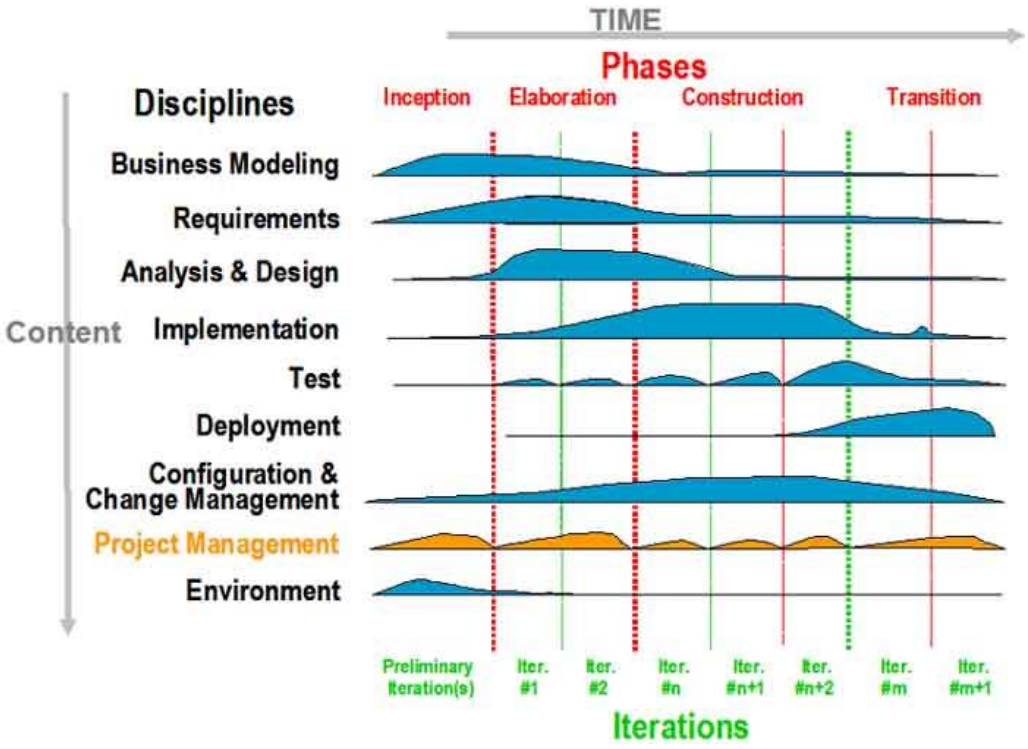
Continua Uchôa, relatando que este método foi proposto para responder aos problemas do modelo anterior, dividindo o processo de desenvolvimento de um produto de *software* em ciclos. Cada ciclo é composto pelas fases de análise, projeto, implementação e testes, ou outras fases, conforme a melhor adequação do método, como pode ser visto na Figura 11. O Modelo incremental é adotado pela proposta patenteada RUP (*Rational Unified Process*) que é definido como "um processo iterativo de desenvolvimento de *software* baseado em gerenciamento de risco e em casos de uso, centrado em arquitetura" (COTTRELL, 2004, tradução nossa). Na Figura 12, é possível visualizar como atividades de uma fase continuam nas fases seguintes, porém com menor ênfase de forma que todas as atividades necessárias ocorrem durante todo o projeto de desenvolvimento do sistema.

Figura 11 – Modelo iterativo e incremental



Fonte: http://www.linhadecodigo.com.br/artigo/2108/evolucao-da-metodologia-do-desenvolvimento-de-sistemas.aspx (Acesso em 16/06/2018)

Figura 12 – Elementos chave do IBM *Rational Unified Process*



Fonte: https://www.ibm.com/developerworks/rational/library/4763.html (Acesso em 16/06/2018)

2.3.1.3 Métodos ágeis

Em 2001, levando em conta o descontentamento de muitas pessoas em relação aos métodos pesados e burocráticos de desenvolvimento, diversos especialistas na área trabalharam em conjunto para lançar o manifesto ágil (no original em inglês: "*Agile Manifesto*"), Este manifesto em ênfase na entrega de *software* operável, parceria com o solicitante do *software*, além de frisar a necessidade de uma adequada interação entre os diversos envolvidos no projeto e a flexibilidade frente a mudanças (GROFFE, 2012).

Trata-se de um conjunto de metodologias e desenvolvimento de *software*, providenciando opções de estruturas conceituais para reger projetos de engenharia de *software*, com a existência de diversos *frameworks* de processos para desenvolvimento de *software*. A maioria destes métodos utiliza de curtos períodos de desenvolvimento (iterações) com o objetivo de minimizar o risco no desenvolvimento, entregando continuamente mini incrementos de funcionalidade, entregando uma nova versão do *software* após terminada cada ciclo destes. Entres estes métodos, adotou-se originalmente, como uma reação aos métodos "pesados", altamente regulamentados e usando o método de cascata (WIKIPEDIA, 2018):

a) *Scrum*;
b) *Crystal Clear*;
c) Programação extrema (XP);
d) *Adaptative Software Development;*
e) Feature Driven Development;
f) *Dinamic System Development Method*.

2.3.2 Desenvolvimento com a metodologia *Lean-Agile*

Passados dois anos do manifesto ágil, foi identificado por Mary e Tom Poppendick um conjunto de alguns princípios atrelados a abordagem "enxuta" (*lean*), inicialmente propagados no contexto de manufatura como uma base para as metodologias ágeis de *software*, pois visam a melhoria de seu desempenho operacional (KISTE E MIYAKE, 2013 apud POPPENDIECK; POPPENDIECK, 2003). Ainda segundo Kiste e Miyake, alguns autores veem as práticas *lean* como um reforço para os métodos ágeis, enquanto outros defendem que o desenvolvimento *lean* representa a evolução do desenvolvimento ágil, no contexto do desenvolvimento do *software* (2013).

Segundo Oliveri (2017), o *lean* ainda carece de uma definição clara e de um consenso profissional. O termo foi cunhado originalmente para designar um modelo de

produção baseado no sistema Toyota de Produção, mas costuma ser considerado um *subframework* debaixo do guarda-chuvas dos métodos ágeis de desenvolvimento de *software*. Atualmente há muita confusão sobre o que é *lean* e o que é *agile* (metodologia ágil), se eles são uma coisa só, e como podem ser usados, mas os dois foram desenvolvidos e adotados como resposta aos mesmos problemas.

Aponta as diferenças entre o *lean*, da forma como é apontada por duas autoridades no assunto (Poppendiek e Charette) e o que está no manifesto ágil, como pode ser visto na tabela 5:

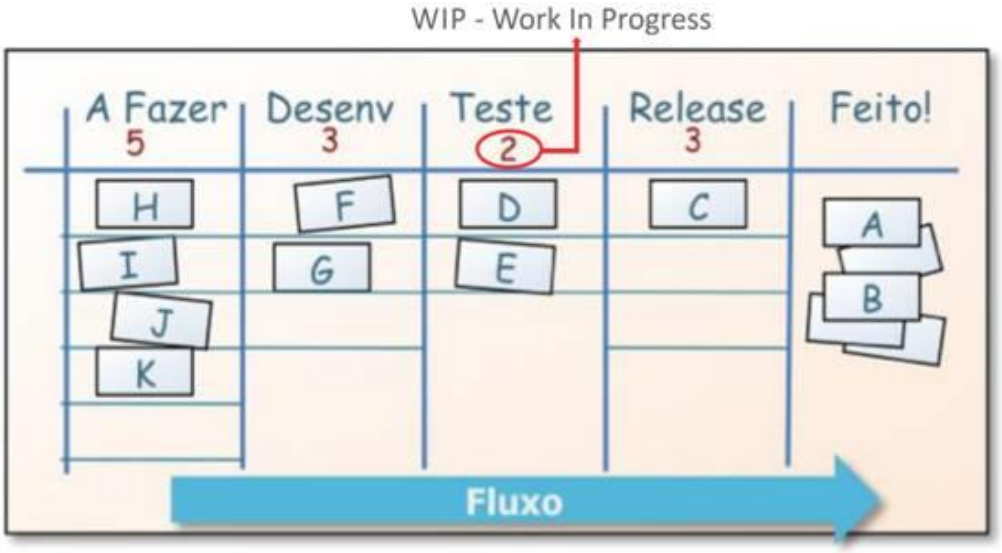
Tabela 5 – comparativo lado a lado entre valores e princípios de *lean* e *agile*

Lean* Vs. *Agile		
***Software Lean* (Segundo Dr. Charette)**	**O manifesto Ágil**	***Lean* (Segundo Poppendieck)**
5 objetivos 1. 1/3 do esforço humano 2. 1/3 das horas de desenvolvimento 3. 1/3 do tempo 4. 1/3 do investimento 5. 1/3 do esforço para se adaptar	**4 objetivos** 1. Indivíduos e interações 2. *Software* funcionando 3. Colaboração com o cliente 4. Respondendo a mudança **Princípios** 1. Satisfação do cliente 2. Bem-vindos requisitos em mudança 3. Ciclos frequentes de entrega 4. Colaboração de *Stakeholders* (pessoas envolvidas e interessadas no projeto) 5. Cultura de apoio à confiança e motivação 6. Comunicação cara a cara 7. O *software* funcionando é a métrica 8. Desenvolvimento sustentável 9. Excelência técnica 10. Simplicidade 11. Equipes auto gerenciáveis 12. Reflexão da equipe (melhoria contínua)	**Princípios** 1. Eliminar desperdício 2. Amplifique o aprendizado 3. Entrega o mais rápido possível 4. Decida o mais tarde possível 5. Capacite a equipe 6. Construa integridade em (o produto) 7. Veja o todo (processo)

Fonte: Adatado de codementor.io (Disponível em https://www.codementor.io/blog/software-development-methodologies-lean-vs-agile-principles-2fdodzuyzy, Acesso em 16 jun. 2018)

É possível observar que existem muitas semelhanças entre os doze princípios do *lean* (segundo Dr. Charrete) e dos doze princípios do manifesto ágil. Os sete princípios propostos por Poppendiecks já apresentam tanta semelhança, mas ainda assim se sobrepõe ao manifesto ágil e ao *Software lean* do Dr. Charrete. Um princípio comum aos dois é o uso quadros Kanban, porém entre outras diferenças[6], no caso do método ágil limita-se o tempo de duração de um ciclo (*sprint*), em que se espera que os itens sejam concluídos, enquanto no *lean*, limita-se o número de tarefas em andamento (a qualquer momento), isso permite que no *lean* se limite o WIP (*work in proccess*), medindo com mais precisão o tempo de entrega e identificando desperdícios na produção (OLIVERI, 2017). A Figura 13 exemplifica um quadro Kanban com indicação de WIP.

Figura 13 – Quadro Kanban



Fonte: Ritter (2014)

[6] Embora o autor coloque a existência de tempo fixo para a *sprint* e o uso WIP como características presentes no método ágil e no *lean*, na pratica os conceitos de *lean* e *agile* se entrelaçam e pode-se ter os dois mecanismos combinados no mesmo quadro Kanban.

3 DESENVOLVIMENTO

Como a metodologia proposta tem caráter prático e experimental, começando pela própria definição de um escopo inicial e acompanhamento de seu desenrolar, sua apresentação terá um conteúdo documental e característico de um projeto de desenvolvimento. Serão apresentados diagramas e tabelas indicando os requisitos e a arquitetura, assim como registro fotográfico e diagramas elétricos do *hardware* adotados, adaptado e expandido, para a criação das plataformas de desenvolvimento.

3.1 DELIMITAÇÃO E ESCOLHA DE PLATAFORMA

Dentro dos objetivos do presente trabalho, tornou-se necessário primeiramente escolher a plataforma de *software* e *hardware*. Já que se pretendia então usar um RTOS e mais de uma opção de *hardware*, a fim de se averiguar os benefícios do uso do RTOS para a portabilidade entre mais de um modelo de equipamento.

Dado o caráter prático e empírico deste projeto, os princípios que embasaram o estabelecimento dos critérios foram fruto de reflexão diante das possibilidades apresentadas, uma vez realizada a pesquisa sobre os RTOS e opções de *hardware* acessíveis para uso.

Através de uma análise das variedades de opções apresentadas detectou-se um conjunto de cinco classes de características dos sistemas operacionais de tempo real, que são listados a seguir:

- quanto ao custo;
- quanto a disponibilização ou não do código fonte;
- quanto a quantidade de microcontroladores suportados;
- quanto a documentação de conhecimento disseminado em comunidades de desenvolvedores;
- quanto a simplicidade de configuração e economia de recursos dos microcontroladores.

Uma vez estabelecidas estas classes, foram estabelecidos critérios com base nos princípios da maior eficiência econômica, visando minimizar os gastos com o projeto, probabilidade de sucesso, baseado em maior número de opções de *hardware* e fontes de pesquisa abundantes para obter exemplos e assistência ao desenvolver o código usando o RTOS e facilidade de uso.

Estes critérios para a escolha do sistema operacional de tempo real estão listados na Figura 14.

Figura 14 – Critérios para a escolha do RTOS

1. O sistema operacional deveria ser gratuito (livre de licença);
2. O RTOS deveria ter código aberto ou disponibilizado para facilitar seu entendimento;
3. O sistema operacional deveria ter grande quantidade de versões para microcontroladores diversos, a fim de maximizar as chances de viabilização da implementação em mais de uma plataforma;
4. Deveria ter documentação disponível e sua utilização ser difundida, facilitando a obtenção de material auxiliar de estudo;
5. O gerenciamento, configurações, distribuição deve ser simples, de memória de programado do SO e da aplicação somadas deve ser baixo, permitindo o uso de microcontroladores de pouca memória (baixo *footprint*).

Fonte: do autor.

Além disto, necessitou-se fazer esta escolha de forma combinada com a escolha das plataformas de *hardware*, visando a viabilização do estudo. Foram adotados os mesmos princípios para estes critérios, adicionando-se a eles princípios relativos ao contato entre o universo *maker* e o desenvolvimento tradicional de produtos e facilidade e produtividade no desenvolvimento (expressos nos requisitos IV e V). Estes critérios estão listados na Figura 15.

Figura 15 – Critérios para a escolha das plataformas de *hardware*

I. As duas plataformas deveriam ser de baixo custo ou facilitada disponibilidade de obtenção de kit de desenvolvimento;
II. Não precisariam ambas as plataformas terem grande poder de processamento, sendo desejável alguma variabilidade;
III. As duas plataformas deveriam ter implementações do sistema operacional com funções e demais itens de API compatíveis, para permitir real uso do RTOS como ferramenta também para portabilidade.
IV. Deveria ter ambiente IDE para a plataforma, facilitando o uso, sem precisar de estudo prévio sobre detalhes muito específicos de montagem, de distribuição ou compilações complexas (exemplo: uso de *makefile*);
V. Deveria ter um projeto de exemplo para cada plataforma, a fim de servir como um ponto de partida para a implementação da aplicação que usa o RTOS.

Fonte: do autor.

3.1.1 Escolha do Sistema Operacional de Tempo Real FreeRTOS

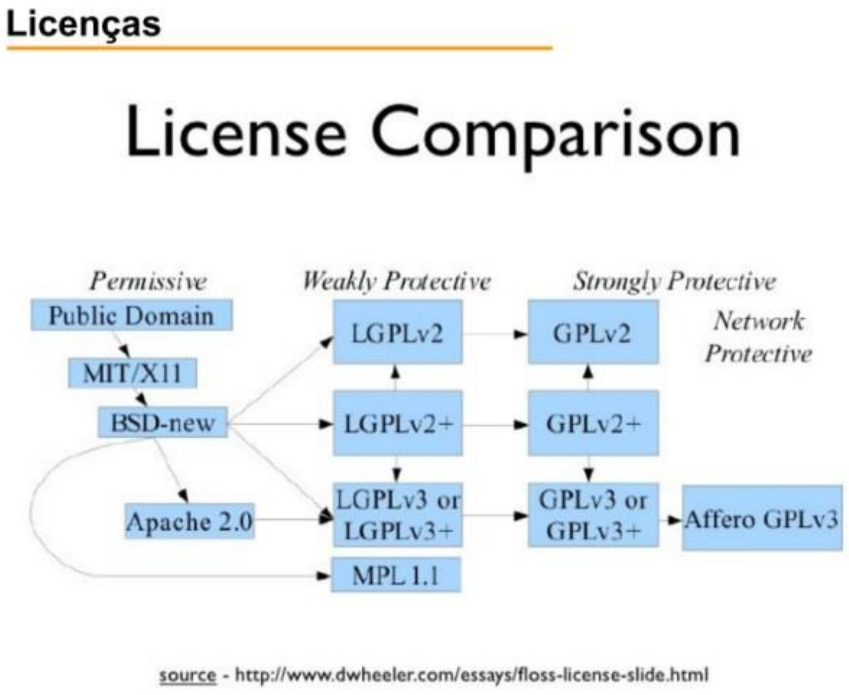
Dentre todos os procedimentos, de fato, a fase de seleção do sistema operacional a ser utilizado no presente trabalho foi bem adiantado.

3.1.1.1 Linux embarcado

Apoiados em materiais como os slides do *webinar* Desenvolvendo com Linux Embarcado (Transmitido ao vivo por embarcados.com em 20 fev. 2017), primeiramente foi abordada a opção de utilizar Linux embarcado, com os *patch* para torná-los um RTOS, porém a medida que foram sendo estabelecidos os critérios (ou requisitos) para a escolha neste trabalho, ele foi abandonado como opção, principalmente por não atender aos requisitos do sistema operacional 3, sobre a quantidade de opções para microcontroladores, já que é destinado a microprocessadores, conforme Bueno e Prado(2017), costuma ter versões grandes que até podem ser mais compactas, customizando-se as distribuições Linux, retirando muito serviços desnecessários. No entanto esta é uma complexidade que fugia de nossos objetivos de análise mais controlada de todo o conjunto de *software*.

Também cabe pontuar que embora o Linux em si atenda aos requisitos 1 e 2 (licença gratuita e código aberto) é necessário ter cuidado, pois os diversos componentes de *software* presentes em suas distribuições, possuem tipos variados de licença de uso, cada uma com suas próprias regras. A Figura 16 nos dá uma ideia da variedade de tipos de licenças que podem ser encontradas:

Figura 16 – *Slide* do *webinar* mostrando a comparação entre licenças (da mais permissiva até a fortemente protetora)



Fonte: https://drive.google.com/file/d/0Byvkmnfcr5auM1l6MHdmLXBjcWJaaF84cnd6aDd0Vk16RE9J/view. Acesso em 16/06/2018

3.1.1.2 FreeRTOS

Posteriormente foi examinada uma segunda opção: O sistema operacional de tempo real FreeRTOS, a partir da audiência de um seminário *web* promovido pela Microgenios e pelo site embarcados.com.br com o professor Fernando Simplício. Neste *webinar*, cuja memória é apresentada na tabela 6, a partir da captura de tela dos slides

(realizada para registro das informações passadas[7]), ele enfatizou aspectos positivos do uso destes sistemas e encorajou fortemente seu uso e popularização no Brasil, abordando principalmente os seguintes pontos, recriados da forma mais fiel possível dada a sua riqueza e importância.

Tabela 6 – Principais Tópicos do *Webinar*.

Título	Resumo
Projeto com MCU	Os projetos com MCU costumam se caracterizar pela conexão de diversos sensores, atuadores, módulos de comunicação e dispositivos de saída, como LCDs.
Programa em C	Apresentou a forma como os programas em C para MCU costumam ser feitas. Primeiramente abordou o caso em que o *loop* principal chama uma função para executar a tarefa de tratar cada dispositivo (ou os dados salvos a seu respeito no tratamento de sua interrupção), precisando ele ser tratado ou não, ou seja, o método de *polling* dos dispositivos. Este tipo de procedimento demora muito a atender um sensor em particular, já que tem que atender todos os outros antes de retornar à função deste sensor.
Máquina de estado	Uma alternativa para minimizar isto (sem usar um escalonador) é quebrar as funções de tarefas que sejam longas em várias funções menores que dividem a tarefa em passos, de acordo com uma máquina de estados. Isto diminui um pouco o tempo de tratamento já que a cada iteração do *loop* principal apenas um estado (ou passo de seu processamento) da tarefa é executado, passando depois a próxima *task*.
Compartilhamento de recursos	É abordado o fato de que os diversos arquivos .h e .c criados para modularizar as partes do *software* embarcado, tendo em vistas os recursos do *hardware*. Estes recursos costumam ser compartilhados e utilizados por inclusão direta da aplicação principal (*main.h* e *main.c*).
Drivers dos	O palestrante começa a mostrar como os recursos podem ser

[7] Como os *slides* não foram tornados públicos no site, não é permitido reproduzir total ou parcialmente suas capturas de tela.

recursos	compartilhados por diversas partes do sistema, em um RTOS, com a divisão do trabalho em tarefas do sistema operacional (*tasks*[8]). Os *drivers* (chamados de *drivers* harmônicos) são escritos levando em conta a competição por seus recursos e usam mecanismos do sistema operacional, como semáforos, para harmonizar (no exemplo dado) duas chamadas a uma função de driver que envia um texto pela serial. Se o cenário sugerido é o seguinte: a) Algum evento ocorre e a *Task* A passa a executar, de menor prioridade, chama a função do driver da USART para escrever o texto "*Hello Word*". Este inicia a transmissão e antes de terminar o envio a *task* B passa a executar (devido a outro evento, aliado ao fato da *Task* B ter maior prioridade: preempção). b) A *Task* B por sua vez, também faz uma chamada a função do driver para escrever o texto "*Go Rattlers!*", porém como o recurso está bloqueado pelo driver, o sistema operacional faz com que a *task* seja bloqueada sem que seja iniciado o envio da segunda cadeia de caracteres. Uma vez que *Task* B foi bloqueada, *Task* A volta a executar; c) *Task* A reassume a execução e espera a escrita ser concluída, antes de seu término. Assim que a função do driver libera o recurso, liberando o semáforo (por exemplo) o RTOS faz com que a tarefa B assuma a execução, já que espera pelo recurso e tem maior prioridade. d) *Task* B retoma a execução e a função de escrita finalmente executa a operação. Este slide exemplifica como o RTOS preocupa-se com certos detalhes, enquanto cada função executa como se fosse a única a escrever coisas na serial.
Em *multitask* devemos	• O tempo para troca de contexto. • O consumo de memória para armazenamento do contexto.

[8] Não fazer confusão entre as funções que executam uma *task*, no programa em C sem RTOS com as funções de *task*, que tem como sinônimos termos como processo e thread e rodam "em paralelo".

considerar (...)	• As *Tasks* podem ser tolerantes a *loops* infinitos. • O compartilhamento de recursos (registradores internos, memória, unidades lógicas aritméticas da CPU e periféricos tais como LCD, Sensores e atuadores) entre as outras *tasks* concorrentes.
Exemplo de programa *Multitask*	Apresenta um pequeno exemplo de como funcionaria um programa *multitask* em um sistema não preemptivo.
Multitasking policy	Exemplificou como o escalonador (*scheduler*) determina as regras de chaveamento (como já comentamos neste trabalho).
multitasking preemptive	Mostra um exemplo em que as ocorrências de interrupções fazem com que ocorra preempção entre duas *tasks*, mostrando a ação do escalonador e o chaveamento de contexto salvando e restaurando os registradores de controle de execução do microcontrolador.
Seções Críticas	Comentou sobre a questão das seções críticas (já comentamos sobre isto neste trabalho)
Reentrância ("*re-enter*")	• Denota os problemas que podem ocorrer quando o mesmo código é executado simultaneamente por várias tarefas ou quando os dados globais são acessados simultaneamente por várias tarefas. • Em um ambiente *multitasking*, as funções preferencialmente devem ser reentrantes.
Seções críticas	O palestrante então mostra um exemplo de uma seção crítica, exemplificando o código em PIC para fazer uma operação OR de *byte* entre o valor da porta PORTB e o valor 01 em hexadecimal. Primeiro é mostrada em linguagem C, ocupando uma linha e seu equivalente em *Assembly*, que ocupa quatro linhas e descreve as instruções realmente executadas. Isto é para exemplificar um acesso não atômico, ou seja, que precisa de vários comandos de máquina para executar. Como esta porta é um recurso compartilhado, e seu acesso é não atômico, temos uma função não reentrante e uma seção crítica, ou seja, uma seção que se o escalonador chavear o contexto, causará inconsistência da operação na

	outra *task*, ou ao voltar para a atual. Em outro slide ele mostra uma função que é reentrante, com as seguintes características: • Ela recebe o dado para trabalhar via parâmetros (que ou é passado pela pilha, ou por um registrador especial do microcontrolador). Como cada *task* tem sua própria pilha e conjunto de registradores alvos e restaurados, o acesso ao parâmetro passado não constitui um acesso a algo compartilhado. • Ela utiliza também uma variável local (no escopo da função), alocada para cada execução da função, também na área de pilha ou registradores da *task* que a chamou; Conclusão: Mesmo que duas *tasks* a chamem simultaneamente cada uma delas terá uma cópia do parâmetro e uma cópia da variável local, sem haver confusão entre os dados das duas chamadas.
Seções críticas	O exemplo anterior é sucedido pelo exemplo de uma função que não é reentrante, pelo fato de acessar uma variável global ao módulo, que é estática e não é alocada na pilha, ou seja, uma única área de memória será acessada por todas as chamadas simultâneas da função em *tasks* separadas e ocorrerá corrupção da informação.
Exemplo: Reentrância ("*re-enter*")	O palestrante exemplifica o caso de erro de concorrência a uma variável global através do código C e A*ssembly* e de um diagrama mostrando o escalonamento entre duas tarefas que acessam a mesma variável. Por fim conclui: • Nunca deve haver acesso a dados globais simultâneo se pelo menos uma das tarefas puder modificar estes dados. • Por isso, deve-se evitar o uso dos dados globais compartilhados. Se isso não for possível deve-se utilizar um semáforo para proteger estes dados do acesso simultâneo.
Benefícios de uma prog. *Multitasking*	• Um processador convencional só pode executar uma única tarefa por vez. Porém um sistema operacional *multitasking* pode fazer parecer como se cada tarefa estivesse sendo

	executada simultaneamente.
Schedure; Filas e *Schedure*; Filas e *Schedure* (c/ Prioridades); *Schedure* (c/ Prioridade); *Schedure* (tipos básicos);	Apresentou uma explicação do escalonador com uso de filas e filas por prioridade assim como os tipos básicos de escalonador (este segundo tópico achamos por bem adicionar na revisão bibliográfica deste trabalho).
RTOS	Ele conceituou o RTOS evidenciando seu caráter fundamental: A questão do tempo, ou seja, as restrições de tempo para atender eventos que ele tem que respeitar, como já discutimos. Adicionou o dado interessante de que um *software* de tempo real não precisa ser multitarefa.
RTOS Comercial	• Suportado por muitas plataformas de pequeno e médio desempenho. • Os RTOS comerciais são amplamente testados por seus desenvolvedores e comunidade.
FreeRTOS	• Sistema Operacional de Tempo Real. • Gratuito e de código aberto. • Desenvolvido pela Real Time Engineers Ltda[9]. • Possui serviços de gerenciamento de *tasks* e memória. • Desenvolvido para ser aplicado em MCUs de pequena capacidade de memória (ARM7 ~4.3kbytes). • Desenvolvido em linguagem C (podendo ser compilado em GCC, Borland C++, etc.). • Suporta limitado número de *tasks* e prioridades (dependendo do *hardware* utilizado). • Infelizmente não acompanha drivers/interfaces de comunicação de rede, *drivers* ou arquivos (*FileSystem*) – (para algumas

[9] Uma visita ao site do RTOS mostra que ele foi recentemente adquirido pela Amazon, porém anunciando que ele continua sendo disponibilizado com as mesmas condições de licença MIT.

	famílias de MCUs é *free*). • Implementa serviços de Filas (*queues*), semáforos binários, contadores, *mutexes* e *software timers*. • Suporta atualmente mais de 38 diferentes arquiteturas. Após elencar estas características explanou como funciona a estrutura em pasta padronizada da distribuição do FreeRTOS e o esquema de um *port* (referente a portabilidade para uma plataforma específica). Por fim orienta a começar no FreeRTOS a partir do exemplo que vem com a *port* da plataforma desejada.
FreeRTOS	Continuou apresentando o padrão de nomenclatura de variáveis, parâmetros e funções e explicou as diversas opções de alocação de memória dinâmica (provida pelo RTOS em substituição a função malloc() do C). Este sistema é usado tanto para criação de *tasks*, filas e afins, como para alocação o dinâmica que seja necessária no código do sistema que está sendo desenvolvido. Este tópico é aprofundado demais para ser exaurido e pode ser consultado na documentação do fornecedor, que apresentaremos a seguir.
Tasks	Por fim apresentou e explicou um exemplo de código de criação de uma *task* em C e explicou seus diversos estados de execução.

Fonte: do autor

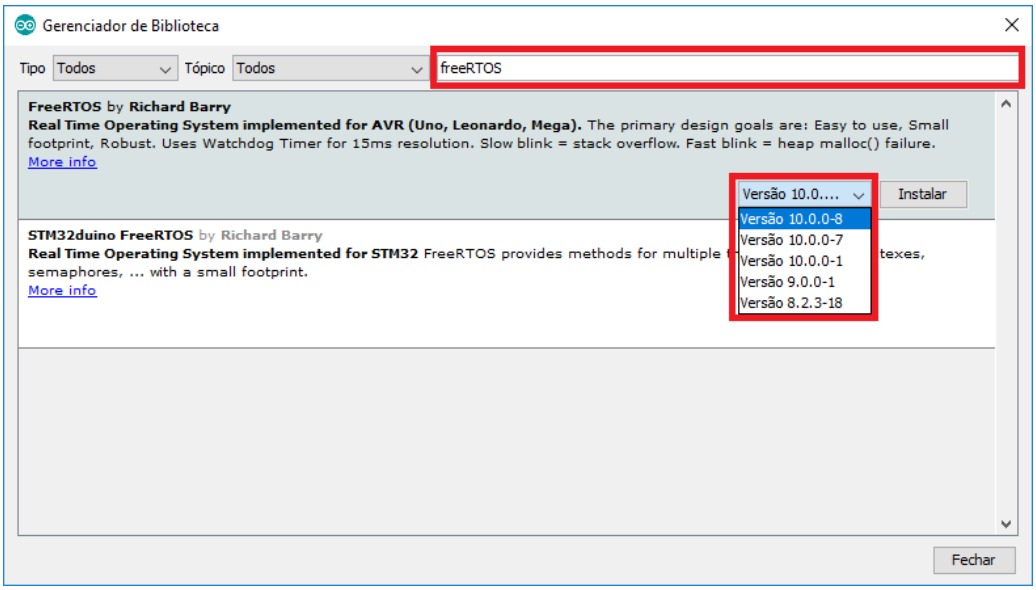
Esta apresentação foi de grande utilidade e de grande estímulo para aprofundar no conhecimento do FreeRTOS, o tornou-se possível através da consulta a documentação no site do fornecedor (www.freertos.org), onde obteve-se acesso a uma boa documentação com o *download* do livro *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide* (*Pre-release 161204 Edition. BARRY, Richard 2017*).

Foi possível então estudar os conceitos e as funções do FreeRTOS para cada um de seus objetos, como a criação e implementação de *tasks*, semáforos, filas, etc.

Sem compromisso de escolha de plataforma, mas dispensando a possibilidade de usar o próprio Windows (não como RTOS, mas como simulação), instalamos a versão 10.0.0-8 do *port* para Arduino em sua IDE, como mostrado na Figura 17, e desenvolvemos vários

exemplos próprios para entender e dominar, pelo menos em parte, o RTOS, antes de decidir adotá-lo para este trabalho.

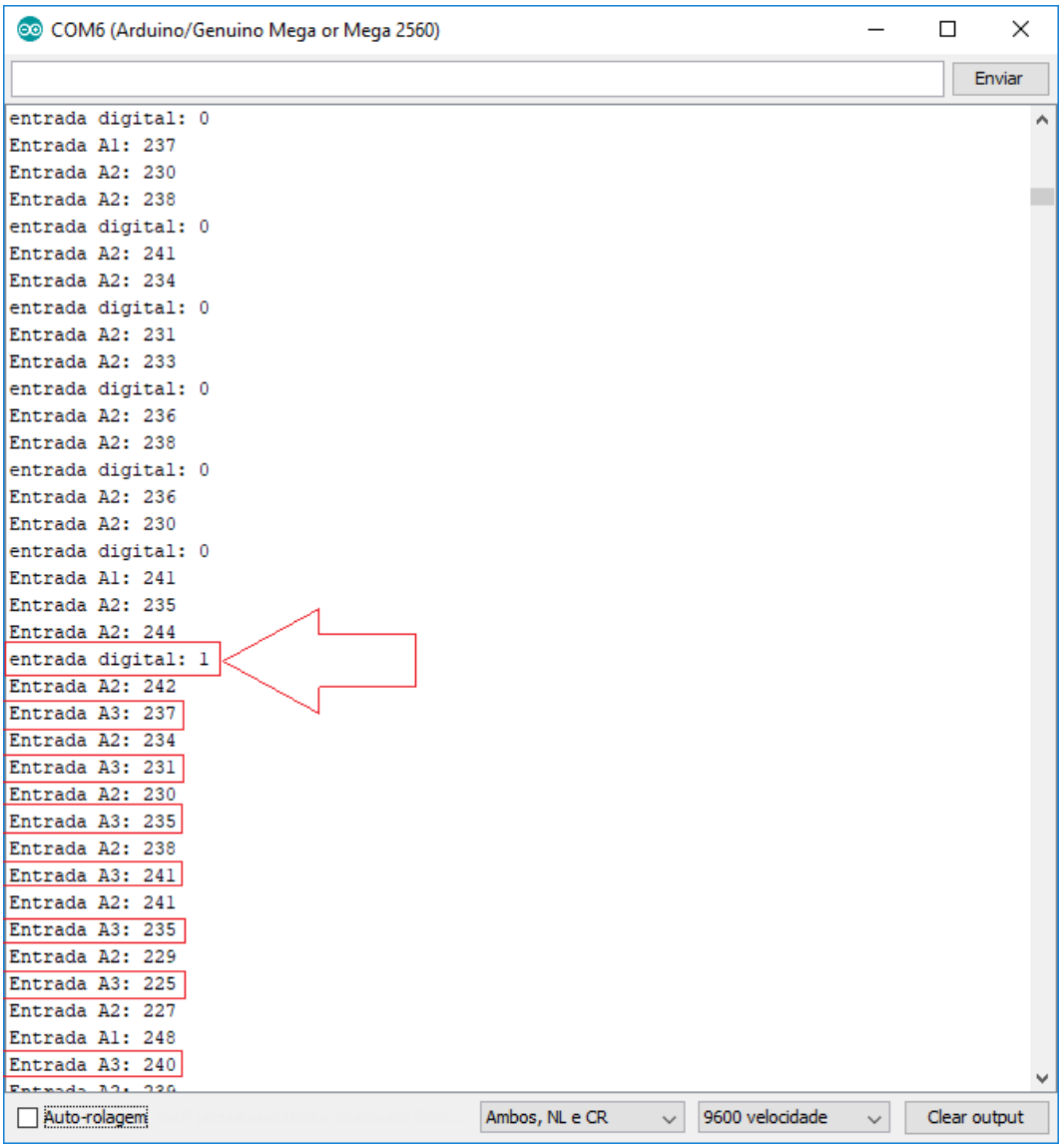
Figura 17 – Instalação do FreeRTOS para Arduino com suporte as placas Uno, Leonardo e Mega



Fonte: do Autor

Um exemplo foi uma modificação do programa de exemplo que é fornecido juntamente com o FreeRTOS para Arduino (analogRead_DigitalReal) para que lesse várias entradas, de acordo com um parâmetro passado com valor diferente para uma mesma *task* (tornando-a na prática diferente, pois opera o mesmo código, sobre uma entrada diferente das demais). Além disto, foi feito o teste de suspender e reativar um *task*, vinculado a leitura de uma entrada digital, que quando ligada por uma das *task*, faz com que esta chame a função do FreeRTOS que reinicia a *task* desejada. Segue tela com o monitor de serial da ID do Arduino, mostrando em destaque os eventos importantes. Ver destaques na Figura 18, que mostram o instante em que é detectado o acionamento da chave e posterior ativação da *task* que a entrada analógica A3.

Figura 18 – Monitor serial mostrando resultado da execução de aplicativo de teste do FreeRTOS no Arduino



Fonte: Autor

O código completo pode ser visto no Anexo B.

Retomando os critérios estabelecidos para a escolha do RTOS (Figura 14), avaliaremos a seguir a adequação do FreeRTOS a cada um deles:

a) **o sistema operacional deveria ser gratuito (livre de licença):** Este requisito é plenamente atendido com a vantagem que usa a licença MIT, que, como vimos na Figura 16, é uma licença permissiva:

> "O FreeRTOS é verdadeiramente gratuito e suportado, mesmo quando usando em aplicações comerciais. A licença MIT de código aberto do FreeRTOS não exige que você exponha suas informações e direitos de propriedade intelectual. Você pode levar um produto ao mercado usando o FreeRTOS sem falar conosco, muito menos pagar qualquer taxa, e milhares de pessoas fazem exatamente isso." (FREERTOS, 2018);

b) **o RTOS deveria ter código aberto ou disponibilizado para facilitar seu entendimento:** Segundo o fornecedor (e de fato verificamos isto), o FreeRTOS foi projetado para ser fácil de usar, por isso está contido em três arquivo de fontes que são comuns a todas os *ports* de plataformas e apenas um arquivo fonte específico para o modelo de microcontrolador em particular. Sua API foi projetada também para ser simples e intuitiva (2018);
c) **o sistema operacional deveria ter grande quantidade de versões para microcontroladores diversos, a fim de maximizar as chances de viabilização da implementação em mais de uma plataforma:** Segundo a seção que lista os *ports* oficiais do FreeRTOS, ele suporta microcontroladores dos de diversos fabricantes, com diversas arquiteturas: Microseimi, Altera, Atmel, Cortus, Cypress, Freescale, Infineon, Spancion, Liminary Micro / Texas Instrument, Microship, microsemi, Renesas, NXP, Silicon Labs, ST, TI, Intel/x86 (FREERTOS, 2018) além de muitos *ports* da comunidade (FreeRTOS *Interactive*) com por exemplo um *port* para Atmel nos modelos usados pelas placas do Arduino, em que ele pode ser incluído como uma simples biblioteca no IDE do Arduino, desenvolvido por Philip Stevens (STEVENS, 2016);
d) **deveria ter documentação disponível e sua utilização ser difundida, facilitando a obtenção de material auxiliar de estudo:** A documentação que baixamos do site é bem satisfatória, rica de exemplos de cada mecanismo da API do FreeRTOS e na média levantada em 2017, foram realizados *downloads* do sistema a cada dois minutos (segundo o site oficial);
e) **o gerenciamento e configurações e distribuição deve ser simples e de memória de programado do SO e da aplicação somadas deve ser**

> **baixo, permitindo o uso de microcontroladores de pouca memória (baixo *footprint*):** O FreeRTOS "Tem mínimos usos de ROM, RAM e sobrecarga de processamento. Tipicamente, uma imagem binária do kernel RTOS ocupa entre 6K a 12K *bytes*." (FREERTOS, 2018, tradução nossa).

Diante de tudo o que foi apresentado, o FreeRTOS mostrou ser uma boa opção de Sistema Operacional de Tempo Real para os objetivos deste trabalho de conclusão de curso.

3.1.2 Escolha da plataforma de *hardware* STM32F411e-Discovery

Após definido um Sistema Operacional de Tempo Real, a suporte ao RTOS definido passou a ser i critério principal para as escolhas das duas plataformas de *hardware*. Se alguma impossibilidade, devido ao não atendimento de um dos outros critérios definidos (ver Figura 15 – Critérios para a escolha das plataformas de hardware), sem a possibilidade de mitigar esta não conformidade, seria necessário rever a escolha do FreeRTOS e escolher algum outro sistema operacional.

A disponibilidade de *kits* de desenvolvimento do fabricante STMicroeletronics na Unisul com suporte a FreeRTOS que encorajou a ideia de realizar primeiro um *benchmark* da conformidade de duas plataformas quanto ao suporte ao referido sistema (a segunda seria o próprio Arduino, naquela altura já testado no RTOS, mesmo que superficialmente).

Depois houve a definição dos propósitos do trabalho, que, em relação as plataformas de *hardware* levaram a formulação da hipótese de que este *kit* de desenvolvimento disponível, o STM32F411e-Discovery poderia ter um bom suporte ao FreeRTOS e que o Arduino poderia ser uma opção interessante por estar em uma outra categoria de produto, a saber: a do desenvolvimento *maker*, tipicamente sem fins comerciais.

Então passamos a criar aplicações básicas de teste do uso do FreeRTOS, também a partir do exemplo que é baixado via IDE Attolic TrueStudio, escolhido por ser gratuito e trazer suporte nativo e simples e pré-instalado ao FreeRTOS (e outros RTOS), possuía um ambiente integrado e que configurava automaticamente os projetos, inclusive com exemplo para o FreeRTOS que pode ser baixado através da IDE.

Este ambiente para a STM32F411e-Discovery é mais elaborado, se comparado com a IDE do Arduino, com possibilidade de *debug* (limitada a alguns *breakpoints*) e execução passo a passo, além da visualização de valores de variáveis.

Porém, antes de partir para a prática de desenvolvimento no ambiente, foi necessário primeiro o estudo atento da plataforma, que nos era desconhecida. A fonte inicial de estudos foi o livro *Discovering the STM32 Microcontroler* (BROWN, 2015), que tinha o inconveniente de ser escrito sobre outro modelo de microcontrolador (porém da mesma família) e o conveniente de ter uma boa explicação da arquitetura básica do Cortex-M3 e vários exemplos de uso de periféricos usando a biblioteca *ST Standard Peripheral Library*. Esta biblioteca permite um acesso de mais alto nível aos registros de configuração do microcontrolador, agilizando o desenvolvimento de aplicações.

Por fim, e principalmente por este motivo, o livro foi escolhido por incluir um capítulo (16) sobre RTOS e FreeRTOS.

Após este período de estudos do ambiente e plataforma, foi implementada uma aplicação baseada em FreeRTOS, que consiste em três *tasks*:

a) uma *task* de Rx em nível de caractere: Recebia de uma fila do FreeRTOS os caracteres que eram recebidos do *hardware*, a partir de sua função de tratamento de interrupção serial (presente no arquivo de driver escrito para isso) e agrupá-los em *strings* até a quebra de linha ou tamanho máximo do texto. Este texto era inserido em outra fila, que então era lida pela *task* de eco serial;
b) uma *task* de Tx em nível de caractere, responsável por receber um texto na fila de entrada (enviado pela *task* de eco serial) e dividi-los em caracteres, enviando-os em uma fila para o driver serial. Após enfileirar todos os caracteres a *task* enviava o primeiro diretamente via uma chamada ao *,10* e os demais caracteres iam sendo retirados da fila e enviados a cada interrupção de *buffer* de Tx vazio (o que significa que acabou de enviar um byte);
c) por fim a *task* de eco serial, a nível de aplicação, recebia os textos de sua fila de entrada e os ecoava, envolvido nos textos “eco: {” e “}”, com caráteres de quebras de linha antes e depois. Foram testados dois métodos de interação da *task* com a fila de saída (envio): 1) A fila de

envio era desconhecida e a *task* conhecia apenas uma função de *callback*[10] que acessava a fila; 2) A fila de recepção era informada via parâmetro para a *task* em sua inicialização, e assim ele podia adicionar o texto diretamente na fila.
Esta *task* de eco, fica bloqueada o tempo todo, exceto quando uma linha de texto é recebida (montada pela *task* de Rx e colocada na fila lida pela *task* de eco).

Posteriormente, este exemplo foi ampliado para utilizar quatro *tasks*, incluindo uma *task* de saudação, que ficava alimentando, de tempos em tempos (através de uma fila) a *task* serial com itens de texto que tinham o objetivo de "saudar" o usuário no PC. Esta temporização foi realizada com o uso de uma função de *delay* do sistema operacional, que suspende a *task* durante o tempo de espera e a torna apta a executar após decorrido este tempo.

A programação deste exemplo e sua execução com sucesso permitiu a validação da plataforma com o RTOS. No entanto algumas dificuldades foram observadas, tanto no ambiente quanto em relação à biblioteca do fabricante:

a) a aplicação mostrava-se instável quando rodada através da IDE em modo de depuração (com o uso do FreeRTOS);
b) foi descoberto um bug na biblioteca ST Standard Peripheral Library, justamente na função que "setava" prioridade das interrupções que impedia que as interrupções do microcontrolador fossem configuradas conforme o necessário para o FreeRTOS[11].

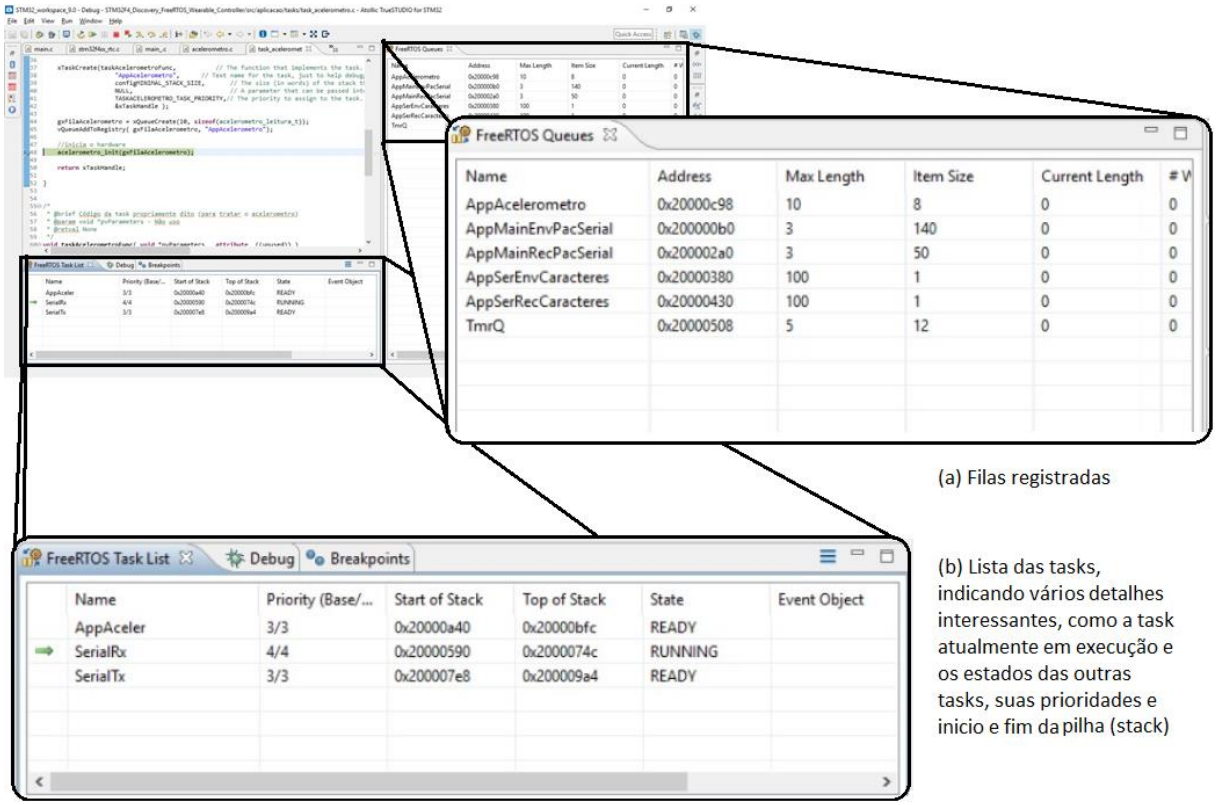
Apesar destes inconvenientes, escolheu-se esta plataforma devido as suas boas características, em conformidade com os requisitos e a funcionalidade de suporte ao FreeRTOS já mencionada, como a visualização de suas filas (se registradas) e *tasks*, conforme pode ser visto na Figura 19.

---

[10] Uma função de *callback* é um recurso de programação onde um ponteiro para uma função é definido (passado por parâmetro para outra função, por exemplo) para que seja possível executar um trecho de código em resposta a algum evento, sem, no entanto, conhecer detalhes sobre sua localização exata. Neste caso específico, a função de *callback* é chamada pela *task* sempre que ele desejar ecoar um texto.

[11] Conforme discutido no artigo *Relevance when using the RTOS* (https://www.freertos.org/RTOS-Cortex-M3-M4.html) nenhuma interrupção deve ter prioridade acima da do FreeRTOS e a prioridades não deve usar o mecanismo de sub-prioridade. Porém a função da biblioteca estava definindo o registrador de prioridade de forma errada, o que feria justamente estes princípios. Foi necessário então usar comandos de baixo nível, que configurassem diretamente o registrador NVIC->IP para definir a prioridade de forma correta.

Figura 19 – Ambiente de desenvolvimento Atollic TrueSTUDIO - Destaque para os painéis de informação sobre FreeRTOS durante o *debug*



Fonte: Do autor

Se revermos os critérios elencados na Figura 15 um a um, poderemos avaliar o quanto a escolha da plataforma foi adequada:

a) **as duas plataformas deveriam ser de baixo custo ou facilitada disponibilidade de obtenção de *kit* de desenvolvimento:** Levando em conta que o kit foi disponibilizado a instituição como amostra, para estimular seu uso acadêmico, ele atende aos requisitos. Em nossa pesquisa também verificamos que o preço no site do fabricante é US$15,00, o que o torna interessante para aplicações comerciais;

b) **não precisariam ambas as plataformas terem grande poder de processamento, sendo desejável alguma variabilidade:** Neste momento, diante deste critério, o nosso procedimento de escolha coloca este *kit* de desenvolvimento, com seu processador STM32F411VET6 como a opção que viria a ser a de maior capacidade das duas, com as seguintes especificações:

- Core: Arm® 32-bit Cortex® -M4 CPU with FPU, Adaptive real-time accelerator (ART Accelerator™) allowing 0-wait state execution from Flash memory, frequency up to 100 MHz, memory protection unit, 125 DMIPS/1.25 DMIPS/MHz (Dhrystone 2.1), and DSP instructions
- Memories
  - Up to 512 Kbytes of Flash memory
  - 128 Kbytes of SRAM
- Clock, reset and supply management
  - 1.7 V to 3.6 V application supply and I/Os
  - POR, PDR, PVD and BOR
  - 4-to-26 MHz crystal oscillator
  - Internal 16 MHz factory-trimmed RC
  - 32 kHz oscillator for RTC with calibration
  - Internal 32 kHz RC with calibration (STMMICROELECTRONICS, 2018);

Na tabela 7 é apresentado um comparativos com a segunda plataforma, que permite constatar esta diferença entre as plataformas.

c) as duas plataformas deveriam ter implementações do sistema operacional com funções e demais itens de API compatíveis, para permitir real uso do RTOS como ferramenta também para portabilidade: Como descrito verificamos o correto funcionamento de várias funções básicas do FreeRTOS na nossa aplicação de testes, porém não foi realizada uma verificação completa, fiando-se na garantia do fornecedor de que as funções são padronizadas entre as plataformas;

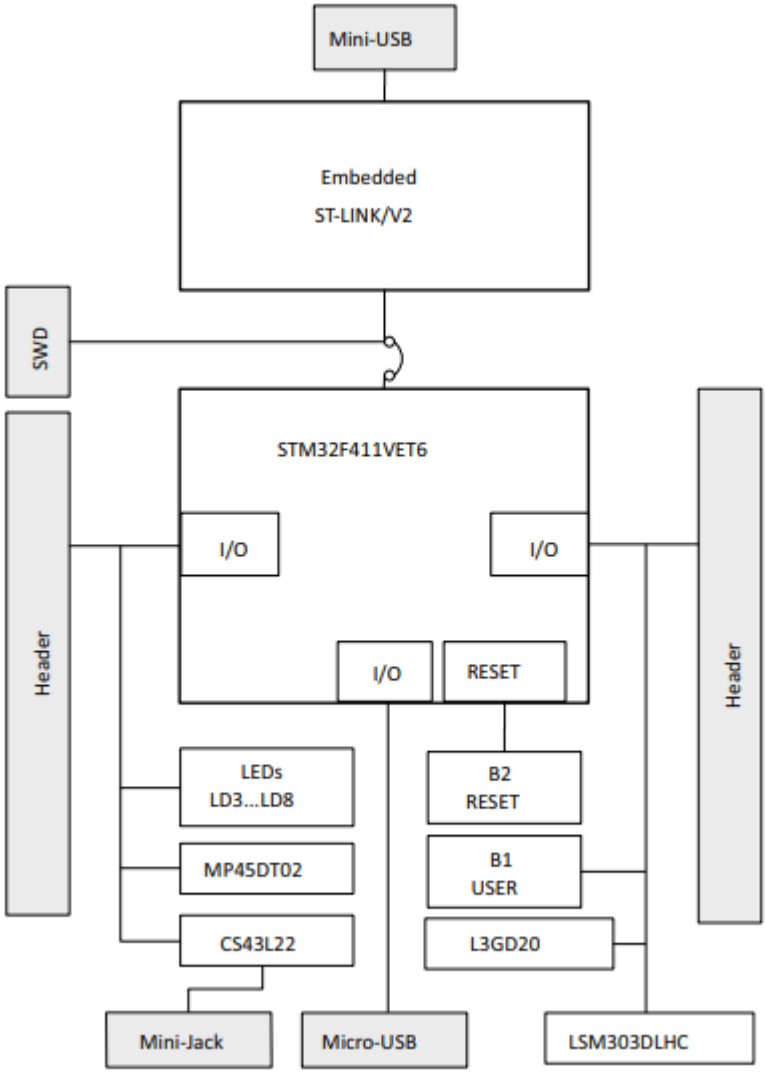
d) deveria ter ambiente IDE para a plataforma, facilitando o uso, sem precisar de estudo prévio sobre detalhes muito específicos de montagem de distribuição ou compilações complexas (exemplo: uso de *makefile*): O critério foi atendido com a IDE Atollic TrueSTUDIO apresentada ao longo do item 2.2.1 e descrito a pouco, quando apresentávamos o teste que realizamos com a IDE e o *kit* com FreeRTOS;

e) deveria ter um projeto de exemplo para cada plataforma, a fim de servir como um ponto de partida para a implementação da aplicação que usa o RTOS: Verificado que o projeto existe, funciona e é útil, por apresentar toda a estrutura de arquivos e função de *callback* necessárias, além de exemplos básicos do uso da API do FreeRTOS. Este projeto de exemplo pode ser baixado na referida IDE no item de menu *File* \ New \ *Download new example Project from TrueSTORE*. Então basta selecionar o projeto STM32F4_Discovery_FreeRTOS_Simple_Demo (no caminho da

arvore *FreeRTOS* \ *STM32F4_Discovery* ou *STMicroelectronics* \ *STM32F4_Discovery*).

Na Figura 20 é apresentado o diagrama de blocos do *hardware* do *kit* de desenvolvimento, obtido de seu manual de usuário. Os dois blocos intitulados *Header*, são conectores de expansão com o objetivo de permitir acesso a maioria dos pinos do microcontrolador:

Figura 20 – Diagrama de blocos do KIT STM32F411e-Discovery



Fonte: STMicroeletronics (2018 pag. 9)

3.1.3 Escolha da plataforma de *hardware* Arduino Mega2560

Durante a definição do Sistema Operacional de Tempo Real foi antecipado o teste do RTOS adotado com a plataforma Arduino. Desta forma sua pronta disponibilidade passou a ser critério determinando para sua escolha, identificada como a opção de menor poder de processamento.

Outro fator que encorajou a escolha do Arduino foi o fato de que ele é uma plataforma muito utilizada no universo *maker* e ter capacidades mais comparáveis com sistemas embarcado de *hardware* limitado dos microcontroladores. Pode ser utilizado para se obter recursos adicionais, como o giroscópio através do uso do *Shields*, que são placas de expansão que podem ser compradas ou manufaturadas. O modelo escolhido possui várias opções de interfaces com os circuitos de expansão, embora seja muito mais limitado quanto a quantidade de memória RAM disponível, se comparado com a plataforma STM32F4.

Além das *Shields*, que tem um formato direcionado para Arduino, existe uma quantidade muito grande de módulos com sensores como giroscópios, acelerômetros, sensores de umidade, ultrassom e assim por diante. Também podem-se utilizar módulos de comunicação, como Wi-Fi.

Evidentemente que estes módulos não são de uso exclusivo do Arduino, mas podemos nos beneficiar dos programas de exemplo para Arduino que encontramos facilmente, muitas vezes a própria loja que vende o módulo ou nos muitos sites dedicados a Arduino e outros placas usadas universo *maker*.

O modelo de Arduino escolhido, dentre os mostrados na Figura 8 – Modelos de placas arduino disponíveis em fornecedor local, foi o Arduino Mega 2560, pois, entre as opções listadas no suporte do FreeRTOS (Uno, Leonardo e Mega) este é que apresenta maiores capacidades, embora ainda pequenas se comparada com a outra plataforma, como veremos a seguir.

Na tabela 7 é apresentado um comparativo das características do microcontrolador STM32F411VET6, do *kit* STM32F411e-discovery e do microcontrolador ATmega2560 da placa Arduino Mega2560:

Tabela 7 – comparativo das principais capacidades dos das duas plataformas

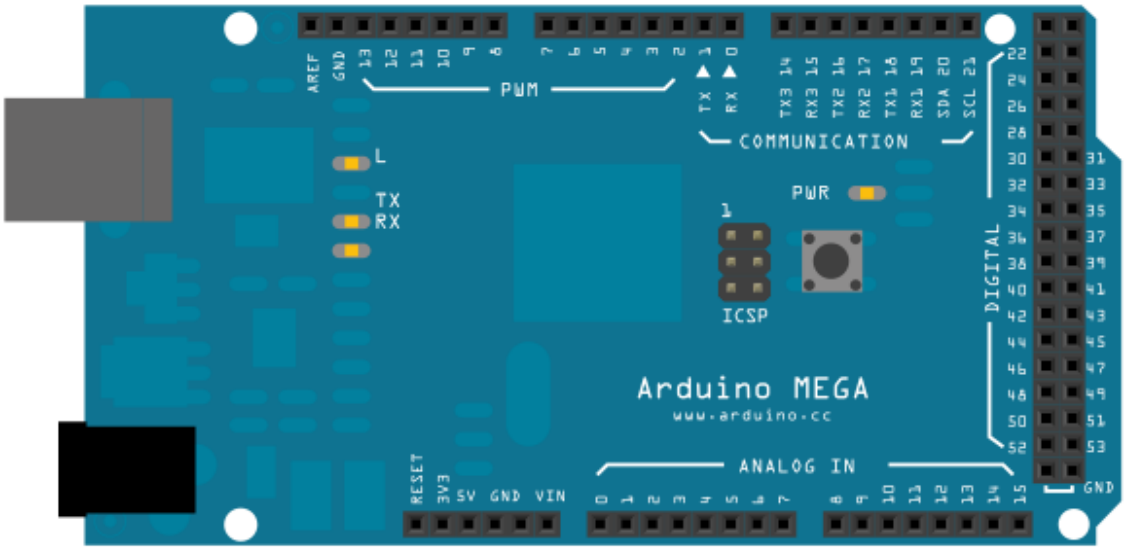
	STM32F411VET6 (Kit STM32F411e-Discovery)	ATmega2560 (placa Arduino Mega 2560)
Arquitetura	Arm® Cortex®-M4 32-bits RISC	Atmel® AVR® 8-Bit Microcontroller
Memória flash (Kbytes)	Até 512	256*
SRAM (Kbytes)	128	8
Clock	Até100 MHz	16 MHz

* Destes 8kb são usados pelo *bootloader* do Arduino.
Fonte: Adaptado de http://www.st.com/en/microcontrollers/stm32f411ve.html, https://www.filipeflop.com/produto/placa-mega-2560-r3-cabo-usb-para-arduino/ e http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf (Acesso em 17 jun. 2018)

De todas as características que o microcontrolador do Arduino tem de capacidade inferior, a que causará mais impacto é a pouca memória SRAM. Será certamente o fator limitante quanto ao uso de *tasks*, filas e demais objetos do FreeRTOS, já que cada um reserva para si determinada quantidade de memória para ter filas e áreas de variáveis no caso das *tasks* e para armazenar os itens de tamanho fixo nas filas (que também tem quantidade máxima de elementos fixa).

A Figura 21 apresenta as interface de conexões externas da placa do /Arduino Mega 2560.

Figura 21 – A placa Arduino Mega 2560



Fonte: Arduino.cc (2018)

3.1.3.1 Diferença de versões do FreeRTOS entre as plataformas

Um detalhe que terá que ser levado em conta é o de que a versão do FreeRTOS pré-instalada na IDE usada para a plataforma STM32F4 é a 9, enquanto a versão 10 do FreeRTOS já estava disponível para download no IDE do Arduino (mesmo que lá seja uma versão do *port* da comunidade e não oficial).

Este problema pode ser facilmente resolvido assumindo que apenas as funcionalidades comuns as duas versões devam ser usadas, ou seja, ignorando as funcionalidades implementadas na versão 10, que são:

> O FreeRTOS 10 contém dois novos recursos significativos: *buffers* de fluxo (*Stream Buffers*) e *buffers* de mensagem (*Message buffers*).
>
> *Stream Buffers* são uma primitiva de comunicação entre processos (IPC) otimizada para uso em cenários onde há apenas um leitor e apenas um gravador, como enviar um fluxo de dados de uma rotina de serviço de interrupção (ISR) para uma tarefa RTOS ou de um núcleo do processador para outro.
>
> *Message buffers* são criados com base nos *Stream Buffers*. Enquanto os *Stream Buffers* enviam um fluxo contínuo de bytes, os *Buffers* de Mensagem enviam mensagens discretas que podem ter tamanho variável. (FREERTOS, 2018)

Embora estes recursos sejam valiosos, inclusive na implementação da comunicação serial em nível de caractere e em nível de texto que implementamos já na fase de seleção da plataforma, não poderemos utilizá-lo, para que possamos aproveitar o benefício da portabilidade entre as plataformas com FreeRTOS versão 9 (STM32F4) e FreeRTOS versão 10 (Arduino).

3.1.4 Escolha de módulos de comunicação e outros periféricos

Seria esperado que após escolhido o RTOS e as plataformas bases, passássemos a definir todo o resto do projeto de *hardware*, escolhendo cada periférico, para cada plataforma, concluindo esta fase inicial de projeto, para só então passar para a fase de desenvolvimento do *software*.

Não entendemos que esta escolha deva ser feita aqui, e que o projeto de *hardware* deva ser fechado, para só então passarmos para a próxima fase, como na metodologia em cascata. Defendemos que após delimitado o problema ilustrativo, haveria um posterior amadurecimento do entendimento de como atender aos seus requisitos e assim poder-se-ia decidir sobre que módulos escolher e como implementar sua integração. Assim agiríamos de forma que a especificação pudesse emergir com o avanço do desenvolvimento, adiando ao máximo as decisões de *hardware* e *software*, para diminuir os riscos de retrabalho e excesso de burocracia e especificações distantes no tempo do momento em que seriam implementadas. Estes são os tipos de coisas defendidos metodologia *lean-agile* (como foi apresentado na Tabela 5 – comparativo lado a lado entre valores e princípios de *lean* e *agile*) e que pretendemos aplicar com esta decisão.

Em uma mão temos um *kit* que tem vários periféricos embutidos, como giroscópios, microfones e sintetizadores de áudio em outra uma placa que não tem muitos recursos embutidos, limitando-se basicamente a disponibilização de interfaces para expansão no padrão do Arduino (*Shields*).

Um outro aspecto que deve ser levado em consideração é como implementar ou adaptar *drivers* existentes para os módulos e outros periféricos, já que se deve implementar esta comunicação em harmonia com o FreeRTOS, ou seja, sem chamadas a funções reentrantes ou com proteções providas pelo RTOS para impedir problemas de compartilhamento de dados em operações não atômicas.

Estas escolhas e comentários serão apresentadas à medida que estas especificações forem sendo feitas, no próprio processo de desenvolvimento.

3.2 PROBLEMA ILUSTRATIVO E REQUISTOS

Com a definição do FreeRTOS e as plataformas baseadas no kit da STMicrolectronic e Arduino, para sua implementação devemos o problema ilustrativo e seus requisitos.

3.2.1 Problema ilustrativo

Tendo em vista os sensores existentes no kit da STM e as tendências de mercado de sistemas embarcados, foi elaborado o seguinte problema: **Desenvolver um dispositivo controlador vestível, que detecte ações do usuário, como gestos, e os envie para um computador.**

A ideia é desenvolver um *hardware* que não precisará ser ergonômico, mas que poderia sê-lo, caso o projeto avançasse além da construção de um protótipo de *hardware* com o *kit* STM32F411e-Discovery e a placa Arduino Mega 2560.

Teríamos então uma versão do produto comercial, de código fechado e que poderia ser comercializada e outra aberta para uso do ecossistema *maker*, que poderia até contribuir com melhorias e ampliações na aplicação, utilizando o controlador para os mais diversos fins.

O uso comercial da versão para a plataforma do fabricante STMicroelectronics é totalmente factível, uma vez que todo o *software* usado tem licenças que permitem este uso.

Já a plataforma baseada em Arduino possibilita a entrega de uma versão para a comunidade, já que utiliza *hardware* e *software* disponibilizado com licenças voltadas para *open source* e *open hardware* que inviabilizam a comercialização sem seguir os mesmos princípios.

3.2.2 Requisitos

A definição do problema ilustrativo se completa com a definição de requisitos funcionais e não funcionais. Estes requisitos formam um enunciado das capacidades do equipamento, sem, no entanto, ditar em detalhes a forma ou tecnologia a ser utilizada.

Com base nestes requisitos, se pode guiar de forma mais precisa no processo de desenvolvimento e gerenciamento do trabalho, já que permitem a verificação objetiva de sua

satisfação, através do confronto das funcionalidades e outras características obtidas com os requisitos definidos.

Os requisitos são divididos em duas categorias:

- requisito funcional (RF): São aqueles relacionados a funções claramente observadas e requeridas. São identificados como RF1, RF2, ..., RFn;
- requisito não-funcional (RNF): São aqueles relacionados a características necessárias porém que não vinculadas de forma direta a funções. São identificados como RNF1, RNF2, ..., RNFn;

A tabela 8 apresenta os requisitos iniciais, que foram estabelecidos para o problema ilustrativo:

Tabela 8 – Levantamento de requisitos para o produto

RNF1	**Requisito:** Uma implementação que puder ser feita usando um recurso do FreeRTOS deve ser preferida, exceto se se mostrar pouco performática ou inadequada. **Objetivo:** Usar a API do FreeRTOS como pilar da programação portável para outras plataformas.
RNF2	**Requisito:** Nenhuma implementação específica de plataforma deve ser inserida em um arquivo com implementação no nível de aplicação ou serviços, mas sim como um driver específico de plataforma, que possuirá sua própria versão na outra plataforma. Da mesma forma, uma implementação que seja e sua unidade de coesão independente de plataforma, deverá ser implementada no nível de aplicação ou serviço. **Objetivo:** Facilitar a escrita do *port* do sistema para a segunda plataforma, a partir do implementado na primeira e aplicar facilmente correção de erros encontrados na aplicação e serviço de uma plataforma na outra.
RF1	**Requisito:** O sistema deverá ler a intervalos regulares um sensor de movimento que detectará os gestos do braço do usuário que o estiver vestindo. **Objetivo:** Criar um controlador vestível (*wearable*) de uso geral.
RF2	**Requisito:** Os dados lidos deverão ser enviados a um computador que então poderá interpretá-los e controlar algo, de acordo com as leituras recebidas. **Objetivo:** Disponibilizar o envio da informação para ser aproveitada em um

	trabalho futuro ou algum outro desenvolvedor usar este produto de forma integrada.
RF3	**Requisito:** O sistema deverá ler o pressionamento de um botão que poderá ser acionado a qualquer momento pelo usuário que o estiver vestindo. **Objetivo:** Disponibilizar uma forma simplificada de interação do usuário com o controlador vestível.
RF4	**Requisito:** A informação de que o pressionamento ocorreu deverá ser enviada a um computador que então poderá interpretá-los e controlar algo. **Objetivo:** Disponibilizar o envio da informação de uma interface simplificada, para ser aproveitada em um trabalho futuro ou algum outro desenvolvedor usar este produto de forma integrada.
RNF3	**Requisito:** todas as leituras realizadas a intervalos regulares e eventos de pressionamento deverão ser tratados brevemente, sem demoras excessivas devido a outras tarefas do sistema. **Objetivo:** Em um sistema de tempo real, deve-se atender os estímulos sem perde-los e respeitando a limites de tempo.
RNF4	**Requisito:** Qualquer tratada que não possa ser enviada para o computador devido a um problema de link deve ser perdida, após um número padrão de tentativas de comunicação. **Objetivo:** Em um sistema de tempo real, deve-se atender os estímulos sem perde-los. Atendê-los de forma muito atrasada é pior que não os atender.

Fonte: do autor

3.3 DESENVOLVIMENTO DO PROJETO (TRABALHO PRÁTICO)

3.3.1 Descrição geral

Tendo em vista os sensores existentes no *kit* da STM e as tendências de mercado este trabalho caracterizou-se por ações concomitantes de diversas naturezas, além da revisão bibliográfica e elaboração dos primeiros capítulos deste trabalho, as seguintes atividades se alternavam e combinavam:

a) ligações do *hardware* de uma plataforma via *jumpers* e *protoboard*;

b) implementação de aplicações de testes para uma determinada funcionalidade;
c) integração de uma funcionalidade ao projeto principal (sobre o FreeRTOS);
d) desenvolvimento de placas de expansão com placas universais, fios, conectores e módulos.

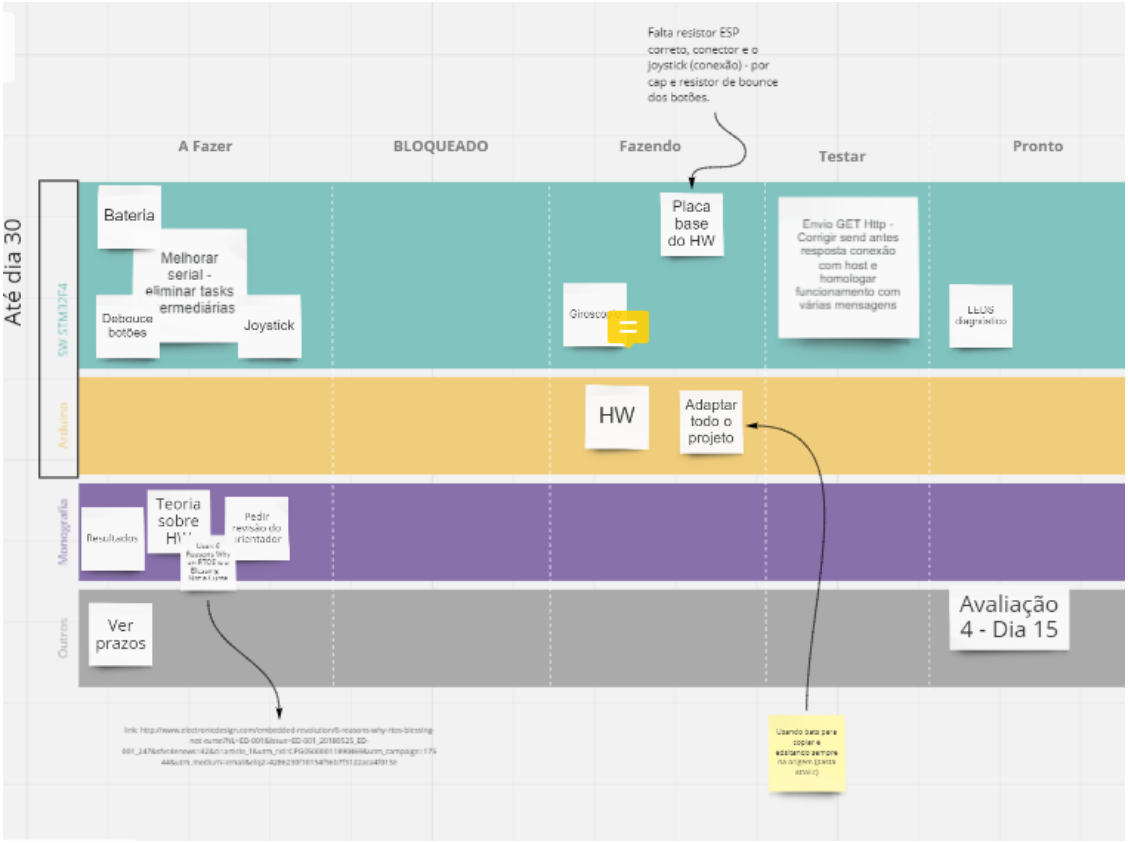
A estratégia adotada foi a de entregar valor aos poucos, mas com entregas constantes, possibilitando uma aproximação progressiva ao cumprimento dos requisitos. Não havendo tempo para implementar todos os requisitos, ao menos poderíamos apresentar os que foram completados e realizar o estudo sobre eles.

Isto se traduzirá na prática de realizar uma implementação de uma funcionalidade até um funcionamento satisfatório para só então passar o foco para a próxima funcionalidade[12]. Por implementação entenda-se tanto montar o *hardware* para a funcionalidade (soldando os fios que ligavam eletricamente o módulo na placa de expansão e conectado o próprio módulo), quanto o *software* embarcado que o suporta (*driver*) que o gerencia (*serviço*) e sua integração ao todo (aplicação).

Foi utilizado um quadro de Kanban digital, durante parte do desenvolvimento, como é apresentado na captura de tela realizada (Figura 22).

[12] Embora este seja o fluxo de execução ideal, funcionalidades foram iniciadas sem que as predecessoras estivessem concluídas. Entre os motivos para tal estão condições de bloqueio e dificuldades diversas. Também houve um replanejamento que ocorreu quando foi avaliado que seria conveniente definir a interface entre os módulos que pretendíamos implementar, escrevendo parte do código de cada um deles juntamente com as declarações das funções.

Figura 22 – Kanban digital utilizado



Fonte: do autor

3.3.2 Etapas de desenvolvimento

3.3.2.1 Organização do projeto

Tomamos como base para o início do desenvolvimento o projeto de teste para a plataforma STM32F411e-Discovery que foi montado para teste, visando confirmar a escolha da plataforma. Então consolidados a estrutura do processo com uma divisão em duas pastas principais:

- “aplicacao”: Destinada a conter todos os fontes independentes de plataforma, incluindo as camadas de aplicação e serviços.
- “plataforma”: Destinada a incluir as subpastas das plataformas. Inicialmente foi criada a subpasta “STM32F4” para onde foram movidos todos os arquivos da aplicação de exemplo baixada no TrueSTUDIO, incluindo a pasta completa com os arquivos fontes do FreeRTOS para a plataforma e os arquivos de driver serial escritos.

3.3.2.2 Acelerômetro

Buscando atender ao requisito RF1, baixamos um projeto de exemplo da placa que demonstrava como acessar o acelerômetro e adaptamos as chamadas ao driver fornecido dentro do projeto. Como o sensor está na própria placa, nenhuma mudança de *hardware* foi necessária.

No entanto a adequação das chamadas para leitura do sensor não funcionou. Para facilitar o diagnóstico do problema passamos a testar esta funcionalidade no projeto original de exemplo, ou seja, sem o FreeRTOS, assim poderíamos detectar uma possível incompatibilidade do driver do *hardware* com o RTOS (como função não reentrantes).

Mesmo neste outro projeto não foi possível ler o sensor, que retornava sempre uma leitura com o valor 0 em todos os endereços de registro para os eitos X, Y e Z do acelerômetro. A falha devia-se ao fato do driver não ser para esta placa, mas sim para uma variação dela e o sensor que estava respondendo era o giroscópio (L3GD20[13]), que passamos a utilizar como sensor de gestos do usuário.

3.3.2.3 Giroscópio

Para que fosse possível a comunicação a contento com o giroscópio, foi adicionado o driver do mesmo a estrutura da plataforma e os primeiros testes com uma aplicação separada mostraram que o sensor respondia, enviando valores em variação de ângulo medida em radianos por segundos. Após este teste passamos ao nosso projeto e criamos o nosso arquivo de driver (que encapsulava o outro padronizando a interface da plataforma com a aplicação) assim como o serviço para ele. Quanto ao suporte ao acelerômetro, foi abandonado pois o giroscópio já atenderia o RF1.

Sob o ponto de vista do uso do RTOS, criamos uma *task* no driver que faria chamadas ao driver "interno" do fabricante a intervalos regulares e colocaria os valores lidos, desde que representassem algum movimento, em uma fila (do FeeRTOS) definida para o driver pelo serviço do giroscópio. Este serviço então implementou uma *task* que lia esta fila com os dados salvos pelo driver e os transformava em uma representação de texto, que então

---

[13] Modificando o *software* de exemplo foi lido o valor do registro "*Who Am I*", que é 0x0F por padrão na linha de sensores da ST, e a resposta foi a do giroscópio e não a do Acelerômetros.

era colocado em uma outra fila, a ser tratada pelo serviço que fosse responsável pela comunicação com o computador (atendendo o RF2).

3.3.2.4 Comunicação com o computador

Um serviço foi implementado com uma *task* que lia uma fila de onde receberia os textos tanto do giroscópio quanto do serviço do botão e os enviaria ao computador. O requisito funcional RF2 estava sendo atendido desta forma. O *hardware* neste momento consistia da placa da plataforma e de um circuito externo responsável pela conversão da comunicação serial em USB que posteriormente era lida como serial novamente no PC.

3.3.2.5 Botão de usuário

Em paralelo a solução do problema com o acelerômetro e migração para o uso do giroscópio, a implementação do RF3, que descreve a necessidade de um botão para o usuário, foi realizada. A placa da plataforma também tem um botão de usuário e baseado em seus códigos de exemplo implementamos a interrupção externa que detectaria a borda de descida, lia a quantidade de *ticks* do sistema (usando uma função da API do FreeRTOS) e media quanto tempo (em *ticks*) havia decorrido desde a leitura anterior e colocava esta estrutura de dados em uma fila, que será lida pelo serviço de botão.

Neste ponto do desenvolvimento, como uma funcionalidade de bônus[14], a ser implementada se possível, foi adquirido um *joystick.* Ele consiste de uma placa com dois potenciômetros acoplados a uma alavanca e um botão ligado o eixo Z de sua alavanca.

Foram realizadas as ligações do circuito do botão do *joystick* na placa, de forma que acionasse uma segunda interrupção externa, que foi atendida no mesmo arquivo de driver. De fato, as duas interrupções inserem itens em uma mesma fila, porém sem confusão entre as fontes da informação, pois faz parte do item na fila um identificador do botão. A informação do botão da placa tem o identificador 0, enquanto o do botão do *joystick* tem o identificador 1.

A exemplo do que faz o serviço do giroscópio, o serviço do botão então passou a ler a fila alimentada pelo driver e montar textos descritivos dos eventos nos dois botões. Este

[14] A ideia deste tipo de funcionalidade é atender os requisitos além do mínimo ou adicionar funcionalidades a serem implementada se sobrar tempo, após a implementação das planejadas.

serviço também é responsável por aplicar uma regra de eliminação de eventos de alto nível, de forma que se a quantidade de *ticks* entre leituras for inferior a determinado valor, a leitura será ignorada.

3.3.2.6 Comunicação HTTP via Wi-Fi

Embora o RF2 estivesse sendo atendido pelo *software*, através do serviço de serial, a adaptação de *hardware* ainda precisaria ser feita (em substituição ao circuito externo envolvendo *protoboard* e uma adaptação sobre uma placa do Arduino Uno). Além disso, uma comunicação via cabo em um dispositivo *wearable*, seria atender os requisitos de forma demasiado limitada.

Foi então adicionado ao *hardware* um módulo externo para comunicação Wi-Fi que já tivesse toda a pilha de *software* necessária, desde o TCP/IP até o HTTP e todas as funcionalidades de conexão Wi-Fi, *login* na rede, etc. O Módulo ESP8266-01, do fabricante Espressif é controlado via comandos AT próprios e comunicação serial.

Para isto foi criado um outro serviço[15], que, em substituição ao serviço serial passou a ler a fila com os dados dos sensores (giroscópio, botões e *joystick*) e gerenciar a comunicação de rede e envio dos dados via requisições a um servidor web (passado a mensagem como parâmetro). O serviço serial, por sua vez, passou a ler da fila alimentada pelo serviço do cliente http e sua fila de dados recebidos (antes sem utilidade) passou a transferir as respostas do módulo Wi-Fi aos comandos AT enviados.

Para testar o recebimento dos dados um servidor apache com PHP foi utilizado em um computador ligado a mesma rede e um *script* responsável por salvar os dados recebidos em um pequeno banco de dados e depois exibi-lo foi implementado.

Em termos de *hardware*, o módulo ESP8266-01 foi conectado provisoriamente a placa e testado em sua funcionalidade básica.

3.3.2.7 Joystick

Todo o driver de leitura do *joystick* foi implementado, através da leitura de entradas A/D que converteriam as leituras de tensão elétrica proporcionais a posição dos

---

[15] Embora este pudesse ser claramente caracterizado como um driver, decidimos colocá-lo como um serviço para que ficasse na área independente de plataforma ,atendendo ao RNF2.

potenciômetros ligados ao eixo X e Y. O serviço também monta o texto representando as leituras e coloca na fila de envio.

3.3.2.8 LEDs

Para facilitar diagnósticos e permiti uma rápida leitura do status de conexão à rede e existência de leituras a serem enviadas (entre outros sinais) foi adicionado um driver para os quatro LEDS de uso geral presentes na placa e passaram a ser controlados pelo serviço do cliente HTTP.

3.3.2.9 Placa de expansão da plataforma STM32F411e-Descovery

Em paralelo as implementações dos serviços e drivers, foi elaborada e confeccionada uma placa de expansão que realizasse todas as conexões entre a placa STM32F411e-Discovery, o módulo Wi-Fi e o *joystick*.

3.3.2.10 Port para a plataforma Arduino Mega 2560

Dentro da pasta “plataforma” da estrutura dos códigos fontes foi criada uma pasta para a segunda plataforma com o nome “ArduinoMega2560” e foram criados os arquivos de cabeçalho e fontes de todos os driver já implementados para a primeira plataforma. Além disto foi criado um arquivo do Arduino, chamado “ArduinoMega2560.ino” e iniciamos o esforço de viabilizar a portabilidade da aplicação.

Devido a limitação de estrutura de arquivos da IDE do Arduino criamos em separado uma outra pasta que trabalhava com uma cópia de todos os arquivos das camadas de aplicação e serviços, assim como os de drivers em uma única pasta, de forma que a IDE conseguisse com isso detectar que todos teriam se ser considerados na compilação.

Este foi um dos motivos do inconveniente, porém necessária, inclusão de diretivas de compilação condicional no início dos arquivos, de forma que lesse os arquivos de *header* ora com uma estrutura de pastas, ora sem subpastas, a exemplo do trecho mostrado na Figura 23.

Figura 23 – Uso de diretiva de compilação condicional

```
#ifdef ARDUINO
#include "leds.h"
#else
#include <plataforma/includes/leds.h>
#endif
```

Fonte: do autor

Após o sucesso em compilar o projeto para Arduino, foi possível implementar os *driveres* específicos da plataforma, assim como seu *hardware* em uma placa de expansão, no estilo dos *Shields* característicos deste tipo de plataforma.

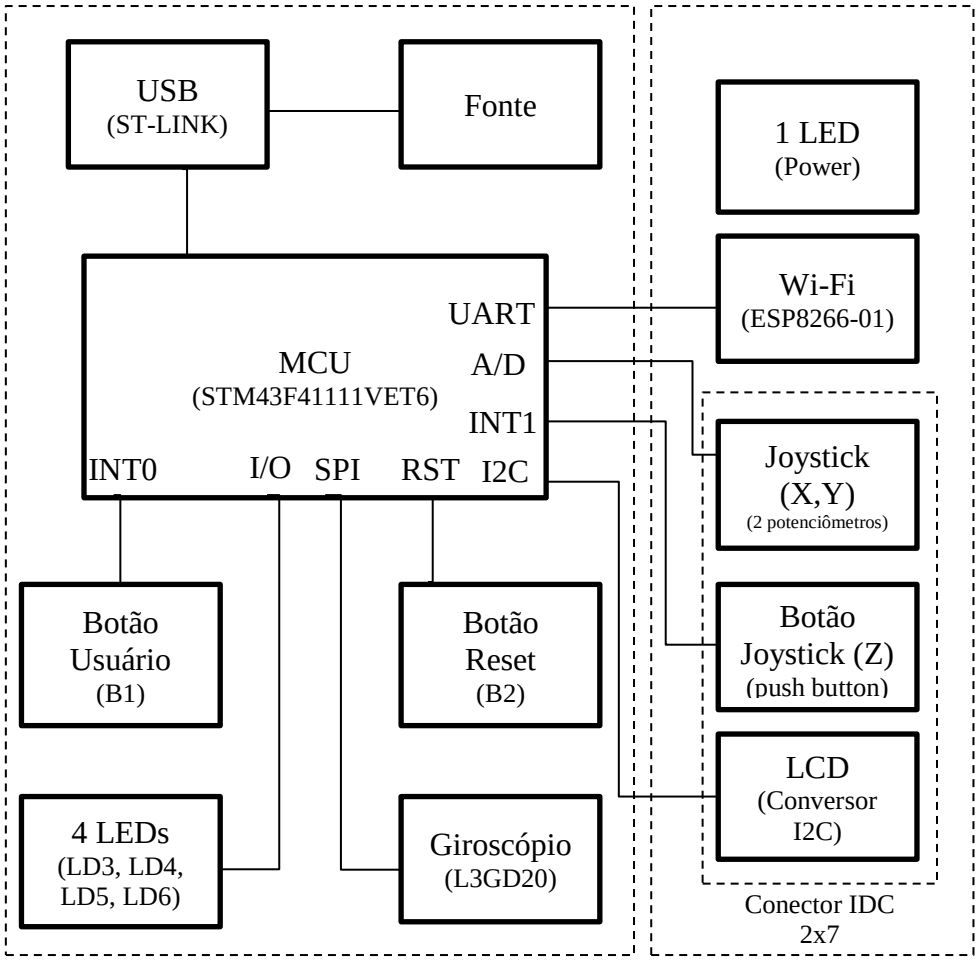
3.3.2.11 Placa de expansão da plataforma Arduino Mega 2560

Esta placa de expansão foi responsável por ligar o módulo ESP8266-01, o *Joystick*, quatro LEDS e o botão de usuário de identificador 0 ao Arduino, compondo uma plataforma de funcionalidades equivalentes. A maior mudança foi o uso do módulo GY-80, que inclui, entre vários sensores, o giroscópio L3G4200D da STMicrolectronics.

3.3.3 Projeto de *hardware* resultante

Não se utilizou todos os recursos da placa STM32F45-Discovery selecionada e além destes alguns componentes foram adicionados na placa de expansão. No diagrama de blocos da Figura 24 podemos ver as partes utilizadas tanto na placa do KIT quanto da placa de expansão.

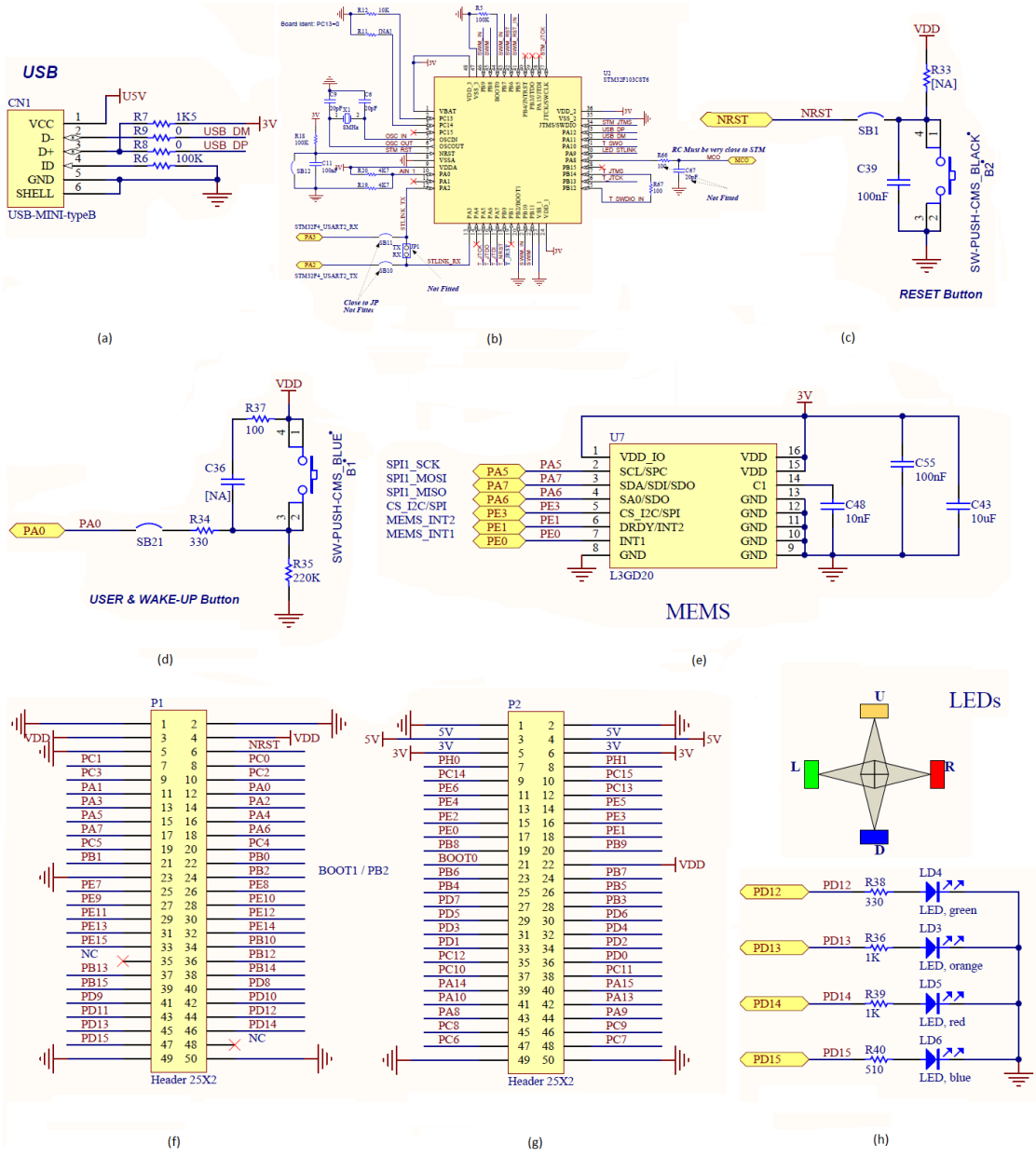
Figura 24 – Diagrama de blocos do hardware da primeira plataforma



Fonte: do autor

A Figura 25 mostra as principais partes do circuito eletrônico do KIT que são realmente utilizados, correspondentes aos blocos indicados na Figura 24(a) e será apresentado em detalhes o projeto do *hardware* resultante.

Figura 25 – KIT STM32F4e-Discovery - Principais componentes do KIT utilizados no projeto da primeira plataforma



(a) Conector USB; (b) Microcontrolador STM32; (c) Botão de reset da placa; (d) Botão do usuário; (e) Sensor Giroscópio MEMS; (f) Conector de expansão P1; (g) Conector de expansão P2; (h) LEDs disponíveis para o programa

Fonte: Adaptado de: ww.st.com (acesso em 09/05/2018)

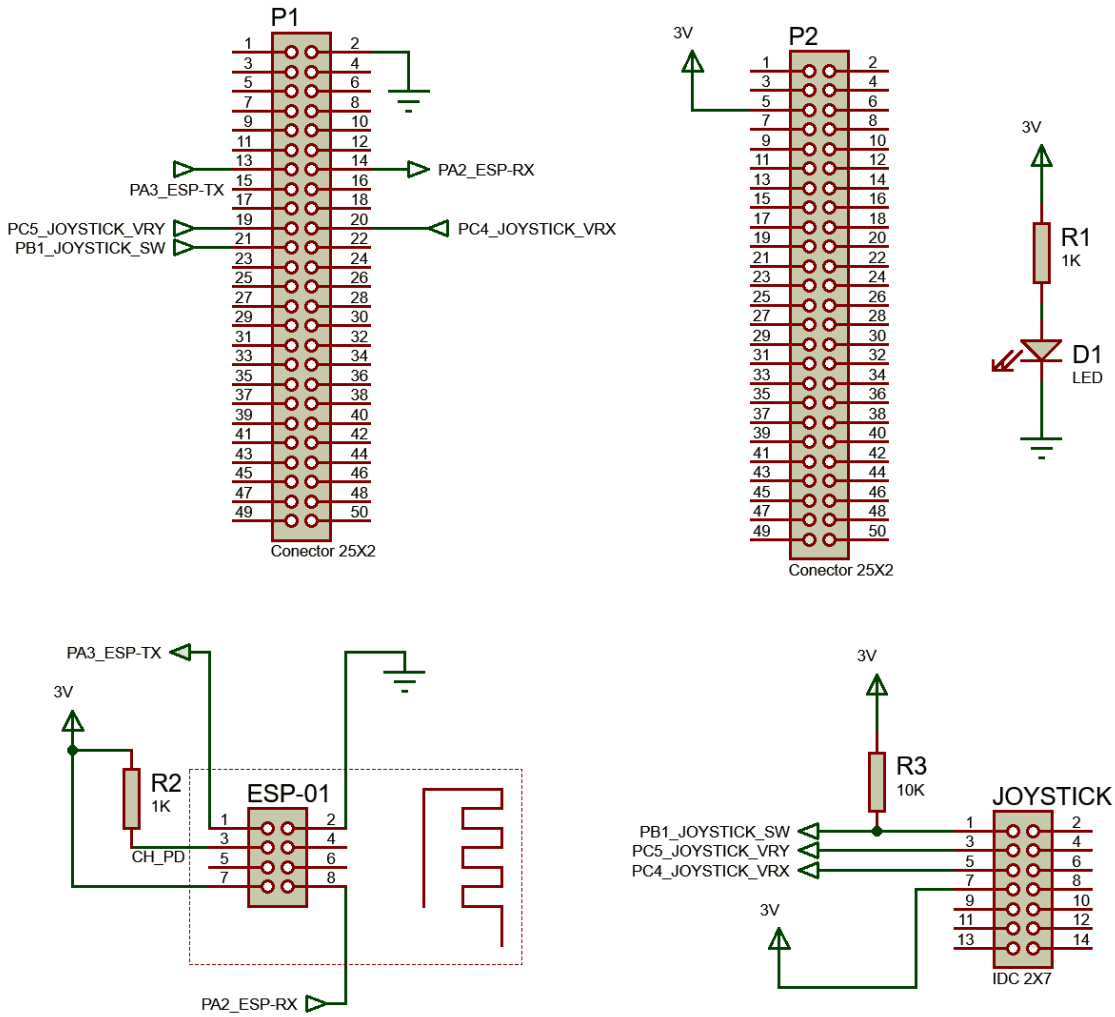
A partir das conexões disponibilizadas pelos conectores P1 e P2 foi possível desenvolver uma placa de expansão para permitir a adição de outros componentes eletrônicos:

a) módulo *ethernet* eSP-01, que se comunica com o sistema através de comandos da serial 1 (pinos PA3 e PA2 de P1);

b) conector para o *joystick*, composto por dois potenciômetros e um botão, conectados através da porta P1 a PC5, PC5 e PB1 (correspondentes ao eixo X, eixo Y e botão, respectivamente).

A Figura 26 mostra o esquema elétrico da placa de expansão, onde é possível ver que também foi adicionado um LED indicador da alimentação (D1) assim como um resistor de *pull-up* para o sinal do botão (R3) e outros resistor (R2), responsável por manter o sinal de controle CH_PD do módulo ESP-01 em nível lógico 1, o que o coloca em regime de operação normal. A alimentação desta placa é obtida nos pinos 2 de P1 e 5 de P2.

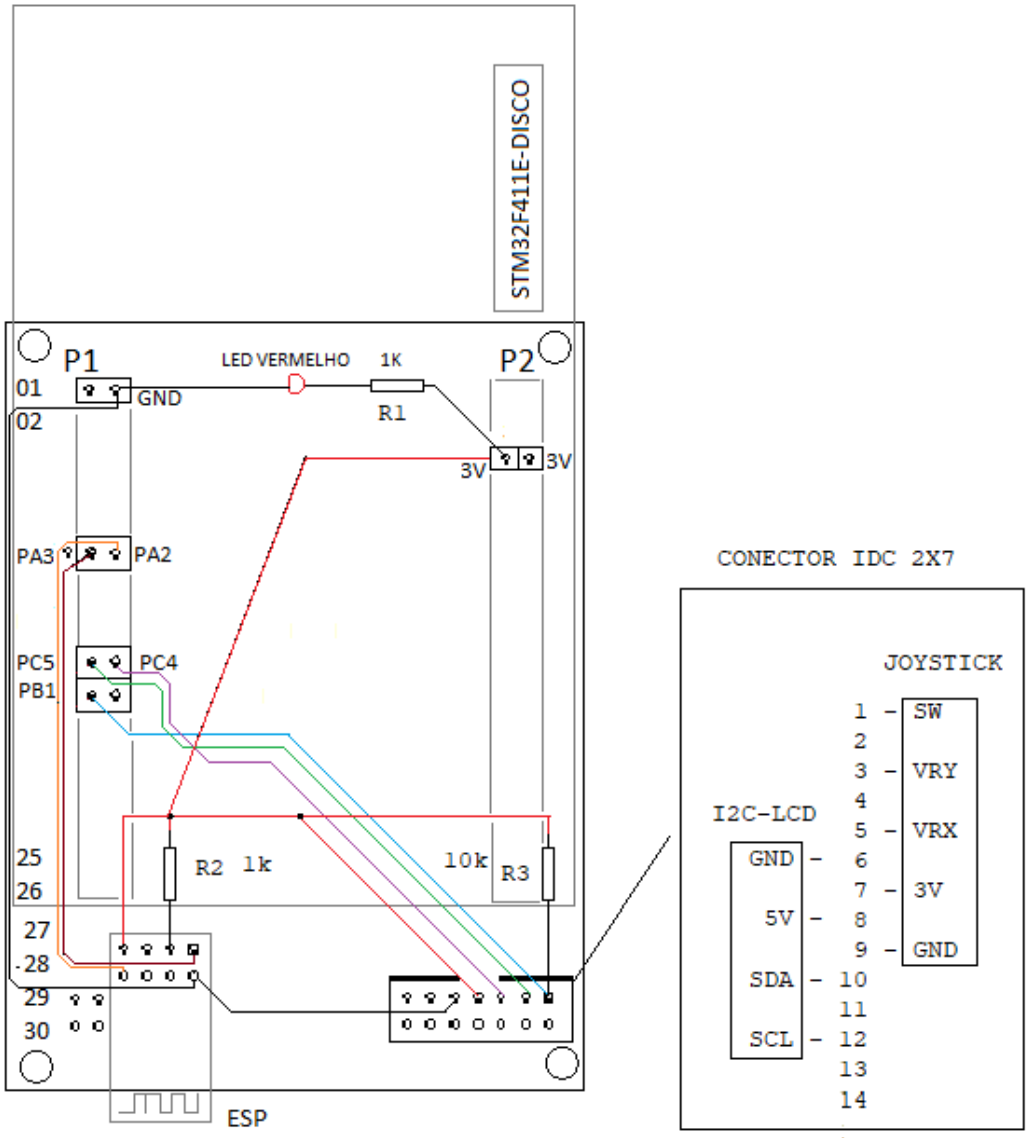
Figura 26 – Esquema elétrico da placa de expansão



Fonte: do autor

A seguir será apresentado o esquema mecânico, mostrando os contornos e ligações da placa de expansão sobreposta ao contorno da placa do KIT STM32F411e-Discovery(Figura 27), assim como uma foto do conjunto (Figura 28).

Figura 27 – Esquema mecânico de conexão entre as placas da primeira plataforma



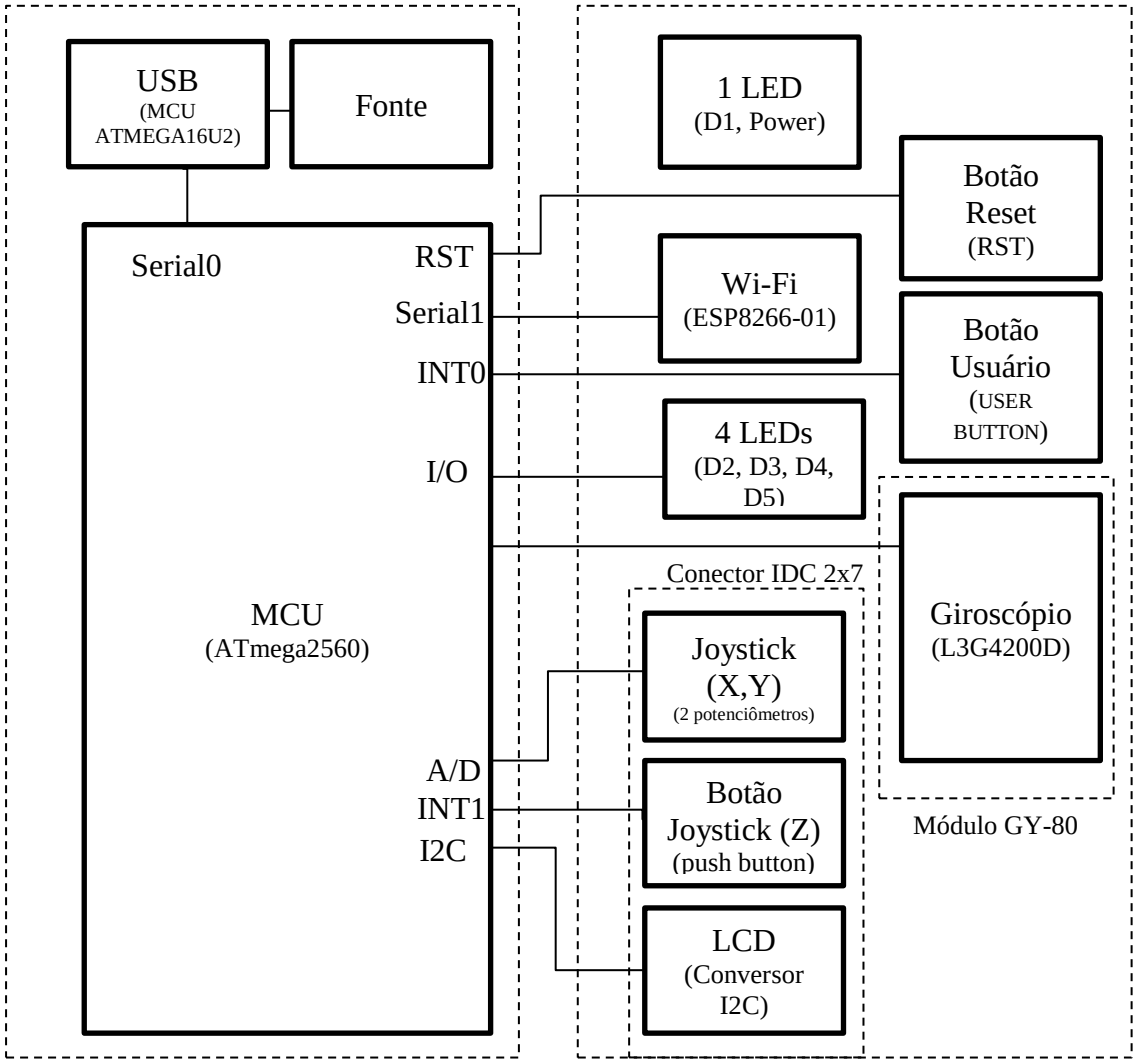
Fonte: do autor

Figura 28 – Fotografia do protótipo com STM32F411e-Discovery (primeira plataforma)

Fonte: do autor

Já o uso do Arduino, para segunda plataforma, exigiu o projeto de um circuito adicional muito maior em sua placa de expansão. Isto se deve a necessidade de acréscimos de alguns circuitos, além do módulo *ethernet* ESP-01 e do conector para o *joystick* e LCD adicionados ao kit STM32F411e-Discovery. Neste caso a plataforma, para atender a aplicação, precisa de um sensor giroscópio e de LEDs indicadores e um botão de usuário, que na primeira plataforma é próprio da placa. A Figura 29 apresenta o diagrama de blocos mostrando as funcionalidades contidas na placa do Arduino e na placa de expansão.

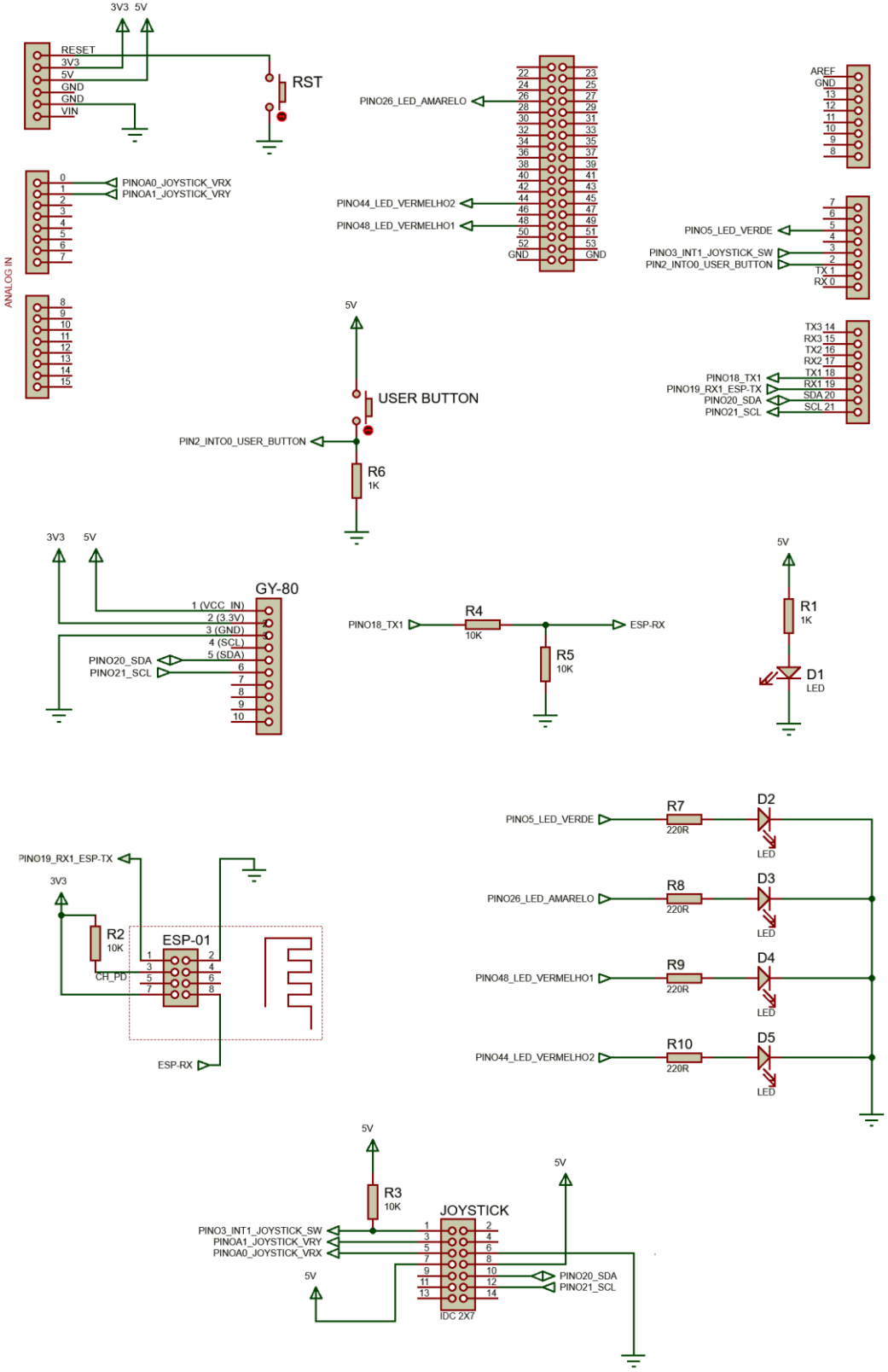
Figura 29 – Diagrama de blocos do hardware da segunda plataforma



Fonte: do autor

O circuito elétrico da placa de expansão, correspondente aos blocos da Figura 29(b) é apresentado no diagrama esquemático da Figura 30.

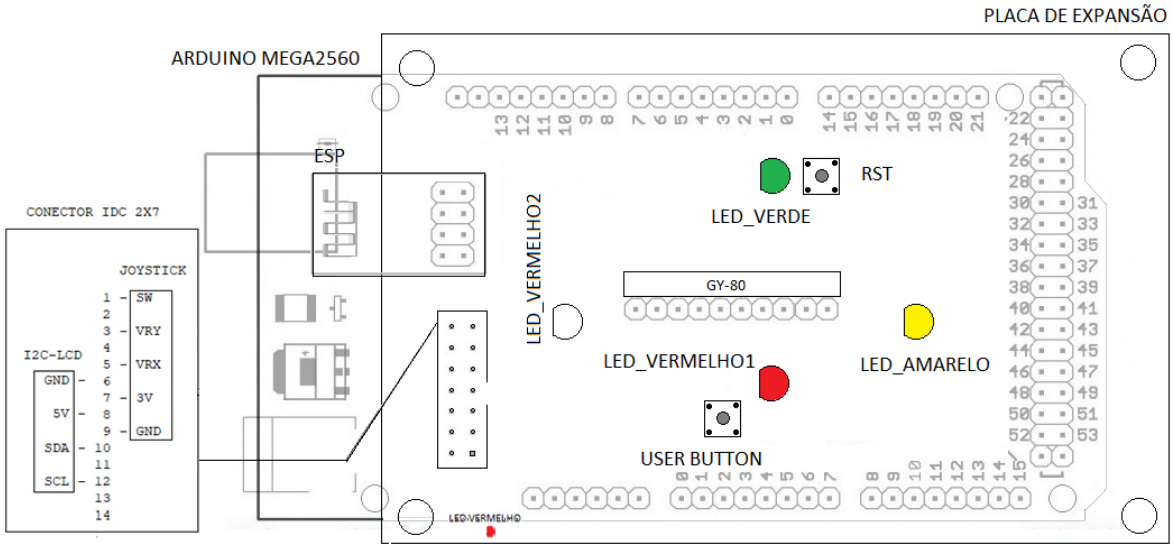
Figura 30 – Esquema elétrico para a placa de expansão do Arduino (segunda plataforma)



Fonte: do autor

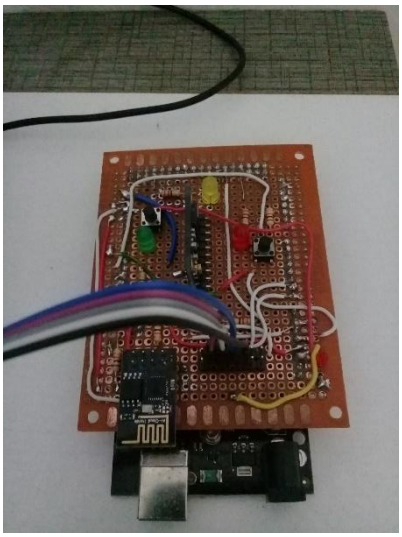
Também foi adicionada ao conector do *joystick* uma interface I2C para comunicação com um LCD adicional. Este LCD cumpre o papel de dar *feedback* ao usuário, quando este liga ou reinicia o sistema, ou ainda se usar o botão do usuário para alterar o filtro de dados do giroscópio, a fim de saber qual a nova configuração selecionada a cada pressionamento do botão. A Figura 31 apresenta o esquema mecânico, omitindo as muitas ligações realizadas (mostradas no esquemático da figura anterior) e a Figura 32 fotos do conjunto.

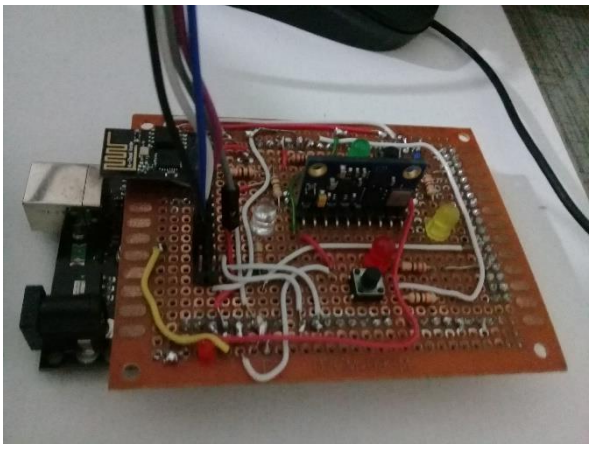
Figura 31 – Esquema mecânico de conexão entre as placas da segunda plataforma

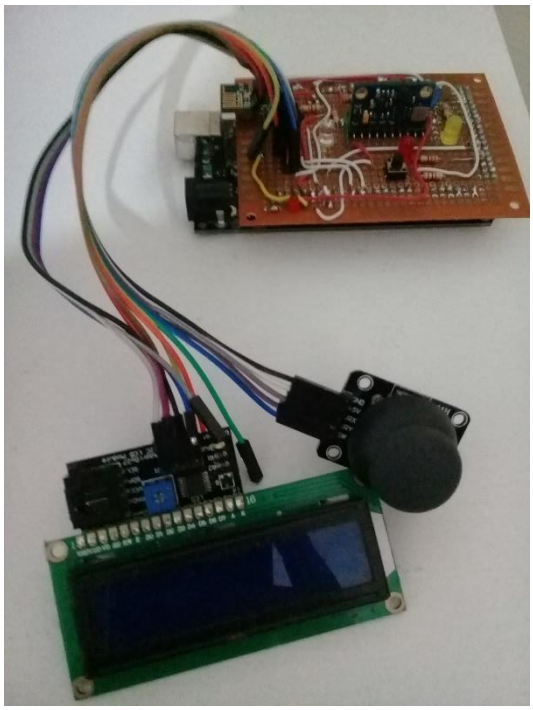


Fonte: do autor

Figura 32 – Conjunto de fotografias do protótipo com Arduino Mega 2560 (segunda plataforma)





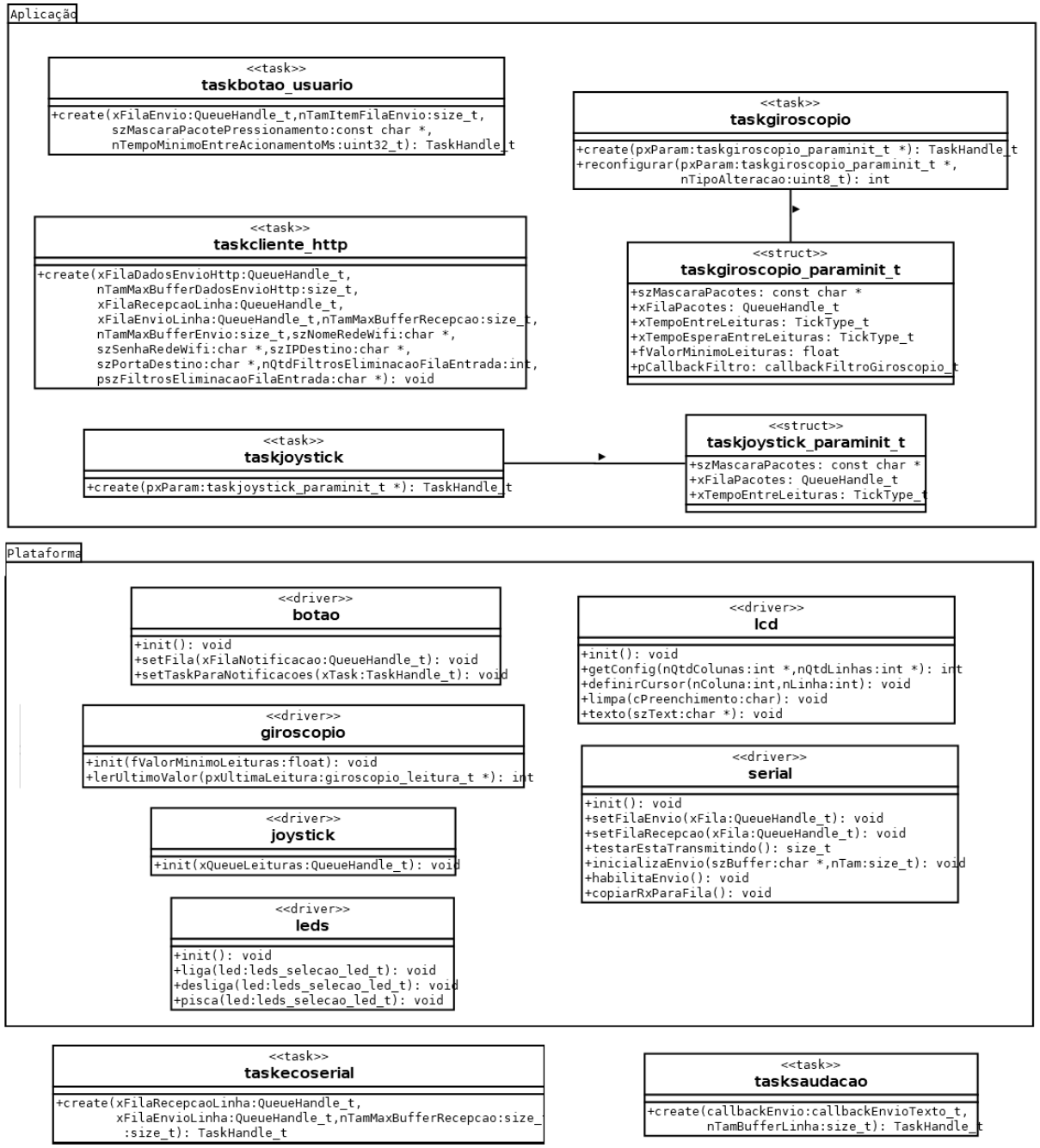


Fonte: do autor

3.3.4 Projeto de *software* embarcado resultante

Apesar da implementação ter sido realizada com programação estruturada e C, seguiu-se uma orientação de modelagem UML com uma visão tipicamente caracterizada por orientação a objetos, de forma que os módulos podem ser entendidos como classes e suas funções como métodos. Na Figura 33 pode ser visto o diagrama de classes, com a separação entre os pacotes de aplicação (contendo também os serviços) e plataforma. Os objetos *taskecoserial* e *tasksaudacao* são apenas para testes e são compilados se necessário.

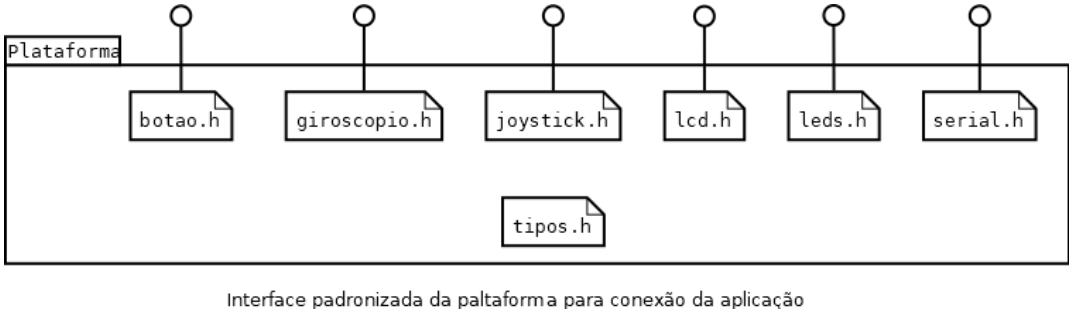
Figura 33 – Diagramas de Classes



Fonte: do autor

A interface entre a plataforma e as camadas acima se dá por um conjunto de arquivos de cabeçalho que são definidos fora das pastas de implementação das plataformas em particular, mas devem ser implementados conforme suas definições. Para os serviços e a aplicação, é este o ponto de conexão, a interface de comunicação, é ilustrado no diagrama da Figura 34.

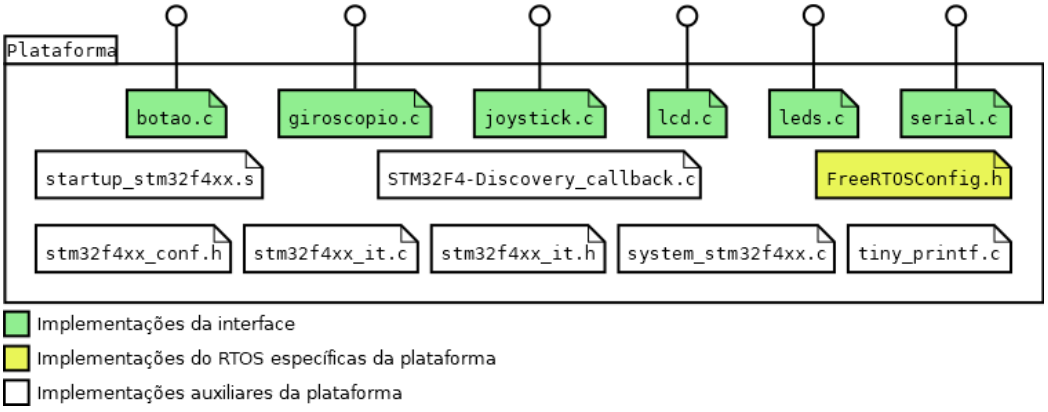
Figura 34 – Interface da plataforma



Fonte: do autor

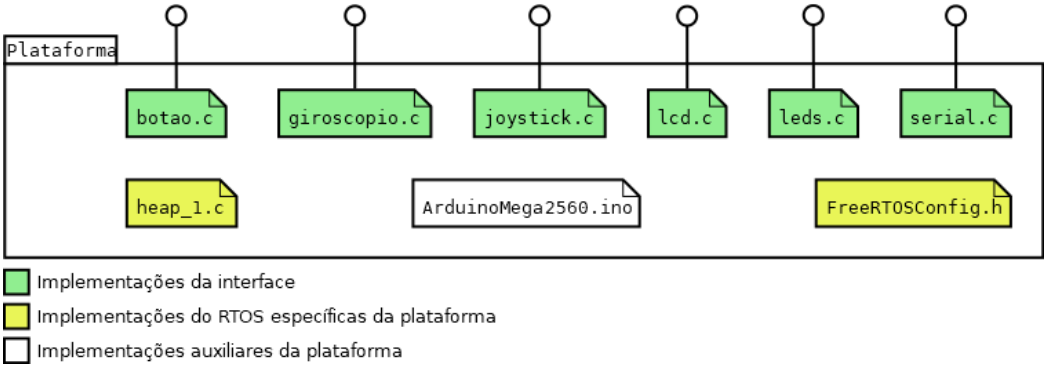
Já as implementações destas interfaces foram desenvolvidas de acordo com cada plataforma e suas características específicas, resultou nos seguintes grupos de arquivos (para cada uma das plataformas), como pode ser visto na Figura 35 e Figura 36.

Figura 35 – Implementação da plataforma STM32F411e-Discovery



Fonte: do autor

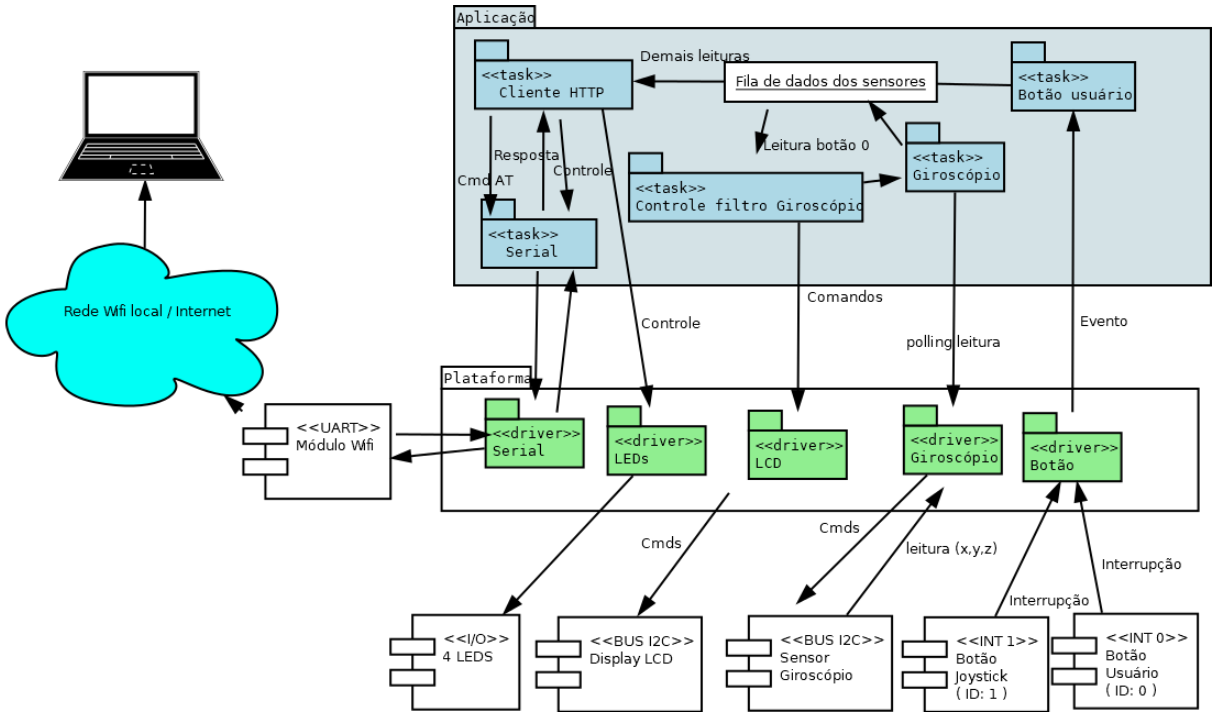
Figura 36 – Implementação da plataforma Arduino Mega 2560



Fonte: do autor

Por fim é possível ter uma visão geral de como todos os módulos se relacionam, mostrada na Figura 37, sendo que muitas vezes este relacionamento é totalmente desacoplado, ou seja, um serviço roda sobre uma *task* que não sabe realmente de onde vem os dados que ela lê de uma fila e também não sabe quem irá ler aqueles que ela mesma adiciona ao fim de outra. Esta decisão é tomada pela função de inicialização da aplicação, que cria as filas de ligação e as passas por parâmetros para serem usadas.

Figura 37 – Diagrama de pacotes (visão geral da aplicação)



Fonte: do autor

3.3.5 Analise da portabilidade entre as plataformas

A realização da portabilidade da aplicação desenvolvida primeiramente para a plataforma STM32F411e-Discovery para a plataforma Arduino Mega 2560, permitindo que a partir daquele momento, quaisquer modificações nos arquivos no nível da aplicação e serviços refletisse nas duas plataformas.

Então tornou-se possível realizar uma medição comparativa, em termos de quantidade de arquivos, tamanho total dos arquivos assim como do número de linhas, entre a

parte comum (aplicação e serviço) e as partes específicas de plataforma, que é apresentada na tabela 9.

Tabela 9 – Comparativo das plataformas

	Quantidade de arquivos	Tamanho total (em Kb)	Número total de linhas
Aplicação	18	105	3553
STM32F4	12	95,3	2956
ArduinoMega2560	9	44,5	1772

Fonte: do autor

Da plataforma, foram ignorados todos os arquivos fontes que não eram relevantes, ou seja, arquivos de drivers usados como bibliotecas assim como o próprio fonte do FreeRTOS. O que se quer aqui é comparar os arquivos efetivamente usados para implementar a solução nesta ou naquela plataforma.

A partir destes dados calculou-se a porcentagem de código para a implementação da plataforma STM32F4, conforme mostrado na tabela 10.

Tabela 10 – Porcentagem de código para implementação da plataforma STM32F4.

	Quantidade de arquivos (%)	Tamanho total (%)	Número total de linhas (%)
Aplicação	60%	60%	55%
STM32F4	40%	40%	45%

Fonte: do autor

A tabela 10 mostra que a aplicação representa 55% dos fontes relevantes levando em conta o número de linhas e 60% pelos dois outros critérios (quantidade de arquivos e tamanho total dos arquivos).

A seguir pode-se verificar o quanto economizou-se na portabilidade para a plataforma Arduino, conforme dados da tabela 11, pois a porcentagem correspondente as linhas de código da aplicação foram reaproveitadas. O valor entrado foi 67% (70%, se levarmos em conta tamanho dos arquivos).

Tabela 11 – Economia na portabilidade para a plataforma Arduino.

	Quantidade de arquivos (%)	Tamanho total (%)	Número total de linhas (%)
Aplicação	67%	70%	67%
ArduinoMega2560	33%	30%	33%

Fonte: do autor

Na verdade, alguma adaptação necessitou ser realizada através de compilação condicional de acordo com a plataforma, devido a uma incompatibilidade do arquivo .h do FreeRTOS, da diferença de estruturas de pastas, macros de debug e configurações globais que determinam tamanhos e propriedades (entre outras coisas) para cada plataforma.

4. RESULTADOS E DISCUSSÕES

Nem todos os serviços iniciados foram concluídos, pois faltou terminar os módulos de *joystick* e LCD, que foram totalmente programados no caso do *joystick* e apenas para o Arduino (no caso do LCD), embora nenhum dos dois tenha sido validado e testado.

Evocando princípios de metodologias ágeis, o trabalho em um *frame* fixo de tempo, como uma *Sprint*, deve resultar em entrega de valor, ou seja, de *software* funcionando. Neste contexto estas duas funcionalidades (ambas adicionadas como bônus), ficariam para ser entregues após concluído o próximo período, e o cliente e demais interessados poderiam aproveitar logo o que fosse entregue.

Pode-se avaliar o ganho de tempo e reaproveitamento de código especialmente pela comparação, em valores absolutos da quantidade de linhas de código independente de plataforma, e assim compartilhado entre as duas implementações, mas também em termos percentuais em relação ao todo do trabalho de codificação empreendido, ponderando-se a adoção de bibliotecas prontas para a placa STM32F4-discovery e para o Arduino. O ganho de tempo foi observado no fato de que a implementação da mesma solução em uma nova plataforma dispensou a escrita de 67% do código na segunda plataforma.

Toda esta quantidade de código pode ser livre de erros de adaptação. Estes erros são os que seriam incluídos por codificação provenientes da alteração inadvertida do algoritmo ligado a aplicação em si por estar próximo ou misturado ao código de chamadas ao *hardware* tanto para acessar periféricos da nova plataforma quanto para os controles de sincronização que também seriam dependentes de plataforma[16].

A funcionalidade de leitura periódica do sensor do giroscópio ilustra esta situação. Caso sua implementação não utilizasse uma metodologia visando reaproveitamento de código, teríamos que tomar um código desenvolvido para a primeira plataforma intercalando linhas de código que continuariam as mesmas, por serem algoritmos independentes de plataforma com outras específicas para o novo *hardware* quem fariam, por exemplo, a configuração de interrupção de *timer*, o tratamento da interrupção e a comunicação com novo modelo de sensor, presente na nova plataforma. Por estarem mescladas, poderiam ocorrer erros em linhas responsáveis na lógica que trata os valores lidos ou que controla o momento

[16] Este código dependente de plataforma, desenvolvido no projeto ficou nos arquivos de driver, que tem versões para cada plataforma. Também o FreeRTOS tem uma versão para cada plataforma, adaptando-se aos detalhes de *hardware* deste e daquele microcontrolador, sem que as partes independentes de plataforma que utiliza o RTOS precisem se preocupar com detalhes como temporização e chaveamento de contexto entre as *tasks*.

de executar as leituras. Estes erros também podem ser mais sutis, como diferenças de comportamento da funcionalidade entre as plataformas.

Seria muito difícil reconhecer porções significativas do código como iguais entre as duas implementações, que dificultaria muito uma mudança do algoritmo que se aplicasse facilmente as duas plataformas. Chegou-se à conclusão que, embora fosse possível obter tal vantagem sem usar um sistema operacional, a utilização do mesmo permitiu emoldurar a aplicação nos padrões impostos pelo próprio uso do RTOS, como *software timers*, *tasks* e objetos de sincronização (filas, semáforos e eventos de *tasks*) utilizando-se destes para escrever código mais modularizado e recursos de alto nível para comunicar os dados e sincronizar as execuções de tarefas bem definidas.

Sobre esta modularização, pode-se observar que se dá inicialmente pelo uso das *tasks* do RTOS que modularizam o código sob o aspecto da execução, já que várias tarefas com códigos distintos fazendo cada uma parte do trabalho. Mas esta modularização é um esforço independente dele, já que é uma abordagem clássica de desenvolvimento do *software* profissional (embarcado ou não). A modularização propriamente auxiliada pelo uso do sistema operacional de tempo real, permitiu uma migração mais facilitada entre plataformas de *hardware* que poderia ter sido feita inclusive por outra pessoa (ou equipe), após definidas as funções que a aplicação necessitaria.

Os mecanismos de temporização disponibilizados pelo FreeRTOS, sejam as funções de *delay* ou as temporizações por *software*, economizam tempo de portabilidade liberam o projetista de preocupação com detalhes referentes ao *clock* ou em como impedir que o processamento ficasse preso em uma temporização de *loop* enquanto outras tarefas poderiam estar sendo desempenhadas.

Sobre a separação em camadas, visando a portabilidade e implementar a abstração do *hardware* para a aplicação, como apresentado na revisão bibliográfica, foi indispensável, uma vez que o sistema operacional não oferecia o recurso de máquina virtual em si e sendo preciso que código específico de uma plataforma não se misturasse com partes independentes da outra plataforma.

Também obtivemos um resultado benéfico para plataformas de poucos recursos, principalmente o Arduino, ao definirmos uma arquitetura enxuta, definida especialmente para o problema: A minimização da quantidade de código. Foi reduzido ao máximo o tamanho do *software*, dado que se desenvolveu somente código específico para o *hardware* realmente utilizado para aquela solução, o que torna a solução mais compacta se comparada a um sistema operacional com diversos recursos, especialmente se a camada de aplicação com

suporte a vários tipos de *hardware* vier pré-compilada com a distribuição do *hardware* e não for simples ou possível criar uma compilação ou distribuição que retire tudo o que não for indispensável.

Certamente a abordagem adotada para o desenvolvimento deste projeto, centralizada em um *kernel* RTOS, modularização, desenvolvimento em camadas e minimização da quantidade de código é mais dispendiosa em termos de recursos e em tempo de projeto, mas permite dar um passo em direção as vantagens de um sistema operacional e dos recursos de garantia de tempo de resposta de um RTOS e com alto grau de portabilidade, sem precisar de um equipamento com muitos recursos, compensando o maior tempo de desenvolvimento e *time to market* pelo menor custo por unidade do equipamento. Esta desvantagem tente a diminuir, tendo em vista que é possível ir adicionando *drivers* para novos sensores e outros *hardwares* a medida que outros modelos de equipamentos vão sendo projetado, reaproveitando cada vez mais do que os desenvolvimentos anteriores, economizando tempo de projeto já a partir do segundo projeto.

CONCLUSÃO

Quanto ao problema ilustrativo, o protótipo de controlador vestível, acabou não incluindo muitos periféricos e uma solução complexa, ficando limitado ao sensor giroscópio e a dois botões, sendo que o segundo botão se trata do próprio *joystick* que não se incorporou a solução de fato como se pretendia, mas poderia ser incluído em uma segunda versão. A adição das capturas dos níveis de tensão (leituras analógicas) deste *joystick* assim como a inclusão mais eficiente de um *display* com informações (e de outros periféricos), poderia evidenciar mais ainda as vantagens em reaproveitamento de código e em outros aspectos possibilitados pela adoção do RTOS, dada a maior riqueza de funcionalidade que se poderia ter à disposição e em uso na aplicação. Uma aplicação utilizando mais recursos possibilitaria explorar mais o gerenciamento eficiente dos recursos do microcontrolador (especialmente processamento da CPU) e da sincronização da comunicação com periféricos e entre *tasks,* ou seja, processamento paralelo das funções da aplicação (por compartilhamento de tempo e preempção). Outra funcionalidade que seria interessante está associada ao gerenciamento de energia, implementando-se um sistema de economia de energia da bateria a ser aplicado na ausência de movimento no equipamento e de ação do usuário sobre seus controles.

Mas para os objetivos deste trabalho pode-se ver claramente que o problema proposto atendeu ao seu objetivo principal: Possibilitar o desenvolvimento de um projeto de sistema embarcado, aplicando para isso um sistema operacional de tempo real e permitir a escrita do mesmo sistema em outra plataforma, com significativa quantidade de código fonte reaproveitado.

Os aspectos anteriormente citados resultam em sugestões para trabalhos futuros, que incluam um estudo do aproveitamento de processamento introduzido por tais sistemas operacionais, assim como outros que tratem da questão da economia de energia nos sistemas embarcados, especialmente os alimentados por baterias.

REFERÊNCIAS

Arduino.cc. **What is Arduino?**. Disponível em < https://www.arduino.cc/en/Guide/Introduction>. Acesso em 12. Jun. 2018

Arduino.cc. **MultiSerialMega**. Disponível em <https://www.arduino.cc/en/Tutorial/MultiSerialMega>. Acesso em 09. Jun. 2018

BARRY, Richard. **Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide.** Pre-release 161204 Edition. Disponível em <http://www.openrtos.net/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf> Acesso em 28 nov. 2017

BUENO, Cleiton; PRADO, Sérgio. **WEBNAR DESENVOLVENDO COM LINUX EMBARCADO.** Disponível em < https://experience.embarcados.com.br/webinars/desenvolvendo-com-linux-embarcado/>. Acesso em 16 jun. 2018

7Graus. **Dicionário etimológico**: etimologia e origem das palavras. Disponível em: < <https://www.dicionarioetimologico.com.br/> Acesso em: 28 abr. 2018.

COSTA, Yuri Gonzaga G. da. Sistemas Embarcados. 2008. Disponível em: <http://www.petccufpb.com.br/wp-content/uploads/2008/03/Sistemas-Embarcados.pdf> Acesso em: 28 ab. 2018.

COTTRELL, William. **Standards, compliance, and Rational Unified Process: Integrating RUP and PMBOK.** Disponível em < <https://www.ibm.com/developerworks/rational/library/4763.html>. Acesso em 16 jun. 2018.

FARINHA, Diogo. **Modelo Waterfall.** Disponível em < <https://sistemasdeinfoige.wordpress.com/2014/10/23/modelo-waterfall/> Acesso em 16 jun. 2018

FELIPEFLOP Componentes eletrônicos. **Loja virtual : Arduino > Placas.** Disponível em <https://www.filipeflop.com/categoria/arduino/placas-arduino/> Acesso em 12 jun. 2018.

FREERTOS. **FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions.** Disponível em <https://www.freertos.org/> Acesso em 17 jun. 2018

FREERTOS. **Microcontrollers and compiler tool chains supported by FreeRTOS.** Disponível em <https://www.freertos.org/> Acesso em 17 jun. 2018

FREERTOS. **FreeRTOS version 10.** Disponível em <https://www.freertos.org/FreeRTOS-V10.html> Acesso em 17 jun. 2018.

HERBERT, Thomas. **Embeddeding TCP/IP.** 01 jan. 2000. Disponível em < <https://www.embedded.com/design/other/4227210/Embedding-TCP-IP> Acesso em 12 jun. 2018.

IEEE Spectrum. **Interactive: The Top Programming Languages 2017 - IEEE Spectrum**. Disponível em: https://spectrum.ieee.org/ns/IEEE_TPL_2017/index/2017/0/0/0/1/1/50/1/50/1/50/1/30/1/30/1/30/1/20/1/20/1/5/1/5/1/20/1/100/ . Acesso em: 28 abr. 2018.

KEVIN, Mathew. **What is Real Time Operationg System (RTOS) – How it Works?** Disponível em < http://www.circuitstoday.com/what-is-real-time-operating-system-rtos>. Acesso em 29 abr. 2018.

KISTE, Ricardo Patricio e MIYAKE, Dario Ikuo. **Abordagem lean no desenvolvimento de software ágil**: **Um estudo de caso em uma editora.** Disponível em < http://www.inovarse.org/filebrowser/download/15650> Acesos em 16 jun. 2018.

MACHADO, Francis Berenger e MAIA, Luiz Paulo. **Fundamentos de Sistemas Operacionais.** Rio de Janeiro: LTC, 2011.

MAYERS, Dana. **Wi-Fi for embedded Internet of Things applications.** Disponível em **https://www.ecnmag.com/article/2013/07/wi-fi-embedded-internet-things-applications**. Acesso em 13 jun. 2018.

MEHL, Ewaldo Luiz de Matos. **Sistema Eletrônicos Embarcados.** Disponível em < http://www.eletrica.ufpr.br/mehl/te200/aulas/embarcados.pdf>. Acesso em: 28 abr. 2018.

NULL, Linda e LOBUR, Julia. **Princípios Básicos de Arquitetura e Organização de Computadores.** 2. Ed. Bookman Editora, 2009

OLIVEIRA, Djalma de Pinho Rebouças de**. Sistemas de informação gerenciais**: estratégias, táticas, operacionais. São Paulo: Atlas, 2002.

OLIVERI, Ben. **Software Development Methodologies: Len vs Agile Principes.** Disponível em < https://www.codementor.io/blog/software-development-methodologies-lean-vs-agile-principles-2fdodzuyzy> Acesos em 16 jun. 2018

PERCEPO. **RTOS 101: Semaphores and Queues.** Disponível em <https://percepio.com/2016/08/11/rtos-101-semaphores-and-queues/> Acesso em 10 jun. 2018

ROBOCORE. **Módulo WiFi - ESP8266 - Internet das Coisas – RoboCore.** Disponível em < https://www.robocore.net/loja/produtos/modulo-wifi-esp8266.html> Acesso em 13 jun. 2018

ROSSI, Henrique. **Arquitetura de Software em Sistemas Embarcados.** Disponível em < https://www.embarcados.com.br/arquitetura-de-software-em-sistemas-embarcados/>. Acesso em: 28 abr. 2018.

RITTER, Roger. **Scrum e Plannig Poker: Análise de estimativa de software.** Disponível em < https://www.devmedia.com.br/scrum-e-planning-poker-analise-de-estimativa-de-software/31019> Acesso em 16 lun. 2018

SHAY, Willian A. **Sistemas Operacionais**. São Paulo: MAKRON Books, 1996.

SUEIRO, D.; SOUZA, F.; FERNANDES, G.; ROSSI, H.; ALMEIDA, R.; PEREIRA, R. LIMA, T. **Editorial: Uso de kits comerciais para desenvolvimento de produtos.** 2015. Disponível em < https://www.embarcados.com.br/editorial-uso-de-kits-comerciais-para-desenvolvimento-de-produtos/> Acesso em 10 jun. 2018.

STMicroletronics. **Discovery kit with STM32F411VE MCU**. Disponível em < http://www.st.com/en/evaluation-tools/32f411ediscovery.html>. Acesso em 10 jun. 2018

STMicroletronics. **UM1842 - User manual: Discovery kit with STM32F411VE MCU**. Disponível em <http://www.st.com/content/ccc/resource/technical/document/user_manual/e9/d2/00/5e/15/46/44/0e/DM00148985.pdf/files/DM00148985.pdf/jcr:content/translations/en.DM00148985.pdf>. Acesso em 10 jun. 2018

STMicroletronics. **STM32F411VE - High-performance access line, ARM Cortex-M4 core with DSP and FPU, 512 Kbytes Flash, 100 MHz CPU, ART Accelerator - STMicroelectronics**. Disponível em < http://www.st.com/en/microcontrollers/stm32f411ve.html>. Acesso em 17 jun. 2018

STEVENS, Phillip. **Arduino (AVR) FreeRTOS – FreeRTOS Interactive.** Disponível em < https://interactive.freertos.org/hc/en-us/community/posts/210025806-Arduino-AVR-FreeRTOS> Acesso em 17 jun. 2018.

STMicroletronics. **STM32 IDEs**. Disponível em < http://www.st.com/en/development-tools/stm32-ides.html?querycriteria=productId=LN1200>. Acesso em 12 jun. 2018

TANENBAUM, Andrew S. e WOODHULL, Albert S. **sistemas Operacionais**: Projeto e Implementação. 2. Ed. Porto Alegre: BOOKMAN, 2000.

Uchôa, Juliana Prado. **Evolução da metodologia do desenvolvimento de sistemas.** Disponível em <http://www.linhadecodigo.com.br/artigo/2108/evolucao-da-metodologia-do-desenvolvimento-de-sistemas.aspx> Acesso em 16 jun. 2018.

University of Saskatchewan. **Simple I/O and multitasking.** Disponível em < http://physics.usask.ca/~angie/ep414/labmanual/multitask.htm> Acesso em 10 jun. 2018.

V SEGeT – Simpósio de Excelência em Gestão e Tecnologia. **Seleção de Ssitemas Operacionais de Tempo Real para sistemas Embarcados.** Disponível em < https://www.aedb.br/seget/arquivos/artigos08/361_SELECAO_DE_SISTEMAS_OPERACIONAIS-SEGeT.pdf> Acesso em 10. Jun. 2018

Weckowicz, Thaddus E. **Ludwig von Bertalanffy (1901-1972):A Pioneer of General Systems Theory.** Disponível em: <http://www.richardjung.cz/bert1.pdf> Acesso em: 5 luj. 2018.

Wikipedia. **Teoria geral de sistemas.** Disponível em: <https://pt.wikipedia.org/wiki/Teoria_geral_de_sistemas> Acesso em: 28 abr. 2018.

Wikipedia. **Sistema operacional de tempo-real.** Disponível em: <https://pt.wikipedia.org/wiki/Sistema_operacional_de_tempo-real> Acesso em: 10 jun. 2018.

Wikipedia. **Desenvolvimento ágil de software.** Disponível em: < https://pt.wikipedia.org/wiki/Desenvolvimento_%C3%A1gil_de_software> Acesso em: 16 jun. 2018.

ANEXOS

ANEXO A – Tabela detalhada de RTOS para sistemas embarcados de código aberto ou disponibilizado

Name	License	Source model	Target uses	Status	Platforms	Official website
FX-RTOS	Proprietary	source code provided	embedded	active	ARMv6-M (Cortex-M0/M1), ARMv7-M (Cortex-M3), ARMv7-A (including Cortex A9 MPCore), x86, AVR32, PIC32, MIPS M4K, TI MSP430	rtos.eremex.com/products, fxrtos.ru
Nucleus RTOS	Proprietary	source code provided	embedded	active	ARM (Cortex-M3-M4-R4-R4F-A8-A9, ARM7-9-11), PowerPC, MIPS32-16e, microMIPS, ColdFire, SuperH	www.mentor.com/embedded-*software*/nucleus
OS21	Proprietary	source code provided	embedded (STB)	active	ST40/SH4,ST200,ARM	?
Portos	Proprietary	source code provided	embedded, small footprint	active	(ARM soon), ARMv7 (M3, A8, A9), AVR32, PowerPC 405, sparc64	www.portos.org
RODOS	Proprietary	source code provided	embedded	active	ARMv7 (M3, A8, A9), AVR32, PowerPC 405, sparc64	https://www.montenegros.de/sergio/rodos/rodos-de.html
LynxOS	Proprietary	source code available	embedded	active	Motorola 68010, x86/IA-32, ARM, Freescale PowerPC, PowerPC 970, LEON	www.lynuxworks.com/rtos
OpenRTOS	Proprietary	source code available	embedded	active	Same as FreeRTOS	www.highintegritysystems.com/rtos/openrtos
SafeRTOS	Proprietary	source code & Design Assurance Pack available	embedded, safety critical	active	Same as FreeRTOS	Same as FreeRTOS
RTXC Quadros	Proprietary	source available	embedded	active	ARM - Atmel/Freescale/NXP/ST/TI, Blackfin, 680x0-ColdFire, PowerPC, StarCore, TI-Luminary Stellaris, TI	www.quadros.com/products/operating-systems

					OMAP, XScale	
RTX Keil Real-Time Operating System	Proprietary, royalty free	source available	embedded	active	ARM	www.keil.com/arm/rl-arm/kernel.asp
T-Kernel	T-License	source available	embedded RTOS	active	ARM, MIPS, SH, more	www.t-engine.org
eCosPro	Modified GNU GPLor eCosPro license	open source with non-free portions	embedded	active	ARM7-9, Cortex-A5-A9-M3-M4-M7, 680x0-ColdFire, H8-H8S, IA-32, MIPS32, MIPS64, microMIPS, NIOS II, OpenRISC, PowerPC, SPARC, SH4/4A, TILE-Gx, XScale	www.ecoscentric.com/ecos/ecospro.shtml
uKOS	GNU GPL	open source	embedded	active	Cortex-M3-M4-M7-H7, 6833x, PIC, CSEM icyflex-1, STM32	www.ukos.ch
Atomthreads	BSD	open source	embedded	active	AVR, STM8, ARM, MIPS	atomthreads.com
BeRTOS	modified GNU GPL	open source	embedded	active	ARM, Cortex-M3, ARM ARM7TDMI, Atmel AVR, PowerPC (emu), x86 (emu), x86-64 (emu)	www.bertos.org
BOOS Core	GNU GPL	open source	embedded	active	ARMv5TEJ (TI AM18x), DSP (TI TMS320C64x)	www.baigudin.*software*/boos/
BRTOS	MIT	open source	embedded	active	Freescale Kinetis (ARM Cortex-M4), Freescale Coldfire V1, Freescale HCS08, ST STM32F4xx (ARM Cortex-M4F), NXP LPC11xx (ARM Cortex-M0), NXP LPC176x (ARM Cortex-M3), Renesas RX600 (RX62N), Texas Instruments MSP430, Texas Instruments Stellaris LM3S8968 (ARM Cortex-M3), Texas Instruments Stellaris LM4F120H5QR (ARM Cortex-M4F), Atmel ATMEGA328/128 and Microchip PIC18	github.com/brtos/brtos
CapROS	GNU GPL	open source	embedded	active	IA-32, ARM9	www.capros.org
ChibiOS/RT	Mixed, GNU GPLor proprietary	open source	embedded, small footprint	active	x86, ARM7, ARM9, Cortex-M0-M3-M4, PowerPC e200z, STM8, STM32, AVR, MSP430, ColdFire, H8S	www.chibios.org/dokuwiki/doku.php, sourceforge.net/projects/

CoActionOS (now Stratify OS)	Modified GNU GPLor proprietary	open source	embedded	discontinued	ARM Cortex-M3, LPC17xx	www.coactionos.com
cocoOS	BSD	open source	embedded	active	AVR, MSP430, STM32	www.cocoos.net
Contiki	BSD	open source	embedded, WSN	active	MSP430, AVR, ARM	www.contiki-os.org
CooCox CoOS	BSD	open source	general purpose	active	STM32, NXP LPC1000, TI LM3S8962, Nuvoton NU1xx,Holtek HT32	www.coocox.org/CoOS.htm
distortos	Mozilla	open source	embedded	active	ARMv6-M architecture (ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1): STM32F0, STM32L0; ARMv7-M architecture (ARM Cortex-M3, ARM Cortex-M4(F), ARM Cortex-M7(F)): STM32F1, STM32F4, STM32F7	distortos.org
dnx RTOS	GNU GPL, modified GPL, BSD	open source	embedded (Unix-like)	active	ARM Cortex-M3, STM32	www.dnx-rtos.org
DuinOS	Modified GNU GPL	open source	general purpose	active	AVR (Arduino)	code.google.com/p/duinos
eChronos	AGPLv3	open source	embedded	active	ARM Cortex-M3 - M7	https://ts.data61.csiro.au/projects/
eCos	Modified GNU GPL	open source	embedded	active	ARM-XScale-Cortex-M, CalmRISC, 680x0-ColdFire, fr30, FR-V, H8, IA-32, MIPS, MN10300, OpenRISC, PowerPC, SPARC, SuperH, V8xx	ecos.sourceware.org
Embox	BSD	open source	embedded	active	ARM, LEON, MicroBlaze, MIPS, PowerPC, x86	github.com/embox/embox
Embkernel	GNU LGPL	open source	embedded	active	ARM Cortex M	sourceforge.net/p/embkernel
ERIKA Enterprise	Modified GNU GPL + linking exception	open source	embedded	active	ARM7, ARM Cortex MX, Hitachi H8, Altera Nios2, Microchip dsPIC (including dsPIC30, dsPIC33, and PIC24), Microchip PIC32, ST Microelectronics ST10, Infineon C167, Infineon Tricore, Freescale PPC e200 (MPC 56xx) (including PPC e200 z0, z6, z7), Freescale S12XS, EnSilica eSi-RISC,	erika.tuxfamily.org

					AVR, Lattice Mico32, MSP430, RenesasRX200	
Femto OS	GNU GPLv3	open source	embedded	active	AVR	www.femtoos.org
FreeOSEK	GNU GPLv3	open source	embedded	active	Cortex-M4, MIPS, MSP430, SPARC LEON3	github.com/ciaa/Firmware
FreeRTOS	MIT	open source	embedded	active	ARM, AVR, AVR32, ColdFire, HCS12, IA-32, Cortex-M3-M4-M7, MicroBlaze, MSP430, PIC, PIC32, Renesas H8/S, RX100-200-600-700, 8052, STM32, EFM32	www.freertos.org
Frosted	GNU GPL	open source	embedded, POSIXcompliant, unix-like	active	ARM Cortex-M	github.com/insane-adding-machines/frosted
Fuchsia	Varies	open source	embedded	active	?	fuchsia.googlesource.com
FunkOS	modified Sleepycat License	open source	embedded	active	AVR, MSP430, Cortex-M3	funkos.sourceforge.net
Helium	GNU LGPL	open source	Embedded	Active	HCS08, AVR	helium.sourceforge.net
Hybridthreads	?	open source	FPGA	active	Xilinx Virtex-II Pro ML310, Xilinx Virtex-II Pro XUP	hthreads.csce.uark.edu/wiki/About_Hthreads
IntrOS	MIT	open source	embedded, small footprint	active	ARM Cortex-M, STM8, AVR8	github.com/stateos/IntrOS
ISIX	GNU LGPL	open source	embedded	active	Cortex-M3	bryndza.boff.pl/index.php?dz=rozne&id=isixrtos
iRTOS	GNU LGPL	open source	embedded	active	AVR, ARM7	irtos.sourceforge.net
MaRTE OS	GNU GPL	open source	embedded	active	IA-32	marte.unican.es
mbed-rtos	MIT	open source	embedded	active	Cortex-M, Cortex-R	mbed.org
MenuetOS	GNU GPL	open source	?	active	IA-32	www.menuetos.net

Milos	GNU GPL	open source	embedded	active	Cortex-M3	www.milos.it
miosix	GNU GPL	open source	embedded	active	stm32, efm32 e LPC2000	www.miosix.org
mipOS	Proprietary	open source	embedded	active	Cortex-M3, ST7, STM8, x86	sites.google.com/site/eantcal/home/mipos
Microsoft Invisible Computing (MMLite)	Proprietary	open source	embedded	active	ARM7-9, i386, h8, mips, Trimedia, eCog1	research.microsoft.com/invisible
Nano-RK	Mixed	open source	embedded, WSN	active	AVR, MSP430	www.nanork.org
nOS	Mozilla	open source	embedded	active	AVR, MSP430, Cortex-M0-M3-M4, M16C, RX600, PIC24, Win32, POSIX, STM8	github.com/jimtremblay/nOS
Nut/OS	BSD	open source	embedded, industrial	active	AVR, AVR32, ARM7, ARM9, Cortex-M3	www.ethernut.de/en/firmware/
NuttX	BSD	open source	embedded, small footprint	active	Linux user mode, ARM7-9, Cortex-A5-A8-A9-M0-M3-M4-M7, 8052, Espressif ESP32, Lattice LM32, Renesas MC16C/26/SH-1, RISC-V , Zilog Z16F, Zilog eZ80 Acclaim!, Zilog Z8Encore!, Z80, MIPS PIC32MX, PIC32MZ	www.nuttx.org
OpenEPOS	Proprietary	open source	embedded	active	ARM, x86, AVR, MIPS, PowerPC	epos.lisha.ufsc.br
OSA	BSD	open source	embedded	active	PIC10-PIC24, AVR 8-bit, STM8	wiki.pic24.ru/doku.php/en/osa/ref/intro
PICOS18	GNU GPL	open source	embedded	?	PIC18	?
picoOS	Modified BSD	open source	embedded	maintenance	6502, 80x86, ARM7, AVR, PowerPC,Cortex-M,MSP430,PIC32	picoos.sourceforge.net,github.com/AriZuu/picoos
Piko/RT	Modified BSD	open source	embedded	active	ARM Cortex-M3/M4	github.com/pikort/pikoRT
Pharos	Apache 2.0	open source	embedded, industrial, safety critical	active	ARM926 with MMU, Cortex-R5 and Cortex-M4 with MPU	sourceforge.net/projects/rtospharos/

POK	BSD	open source	embedded	active	x86, PowerPC, SPARC	pok.safety-critical.net
RIOT	GNU LGPL	open source	embedded	active	ARM, MSP430, AVR, x86	www.riot-os.org
RTEMS	modified GNU GPL	open source	embedded	active	ARM, Blackfin, ColdFire, TI C3x/C4x, H8/300, x86, 68k, Milkymist SoC, MIPS, Nios II, PowerPC, SuperH, SPARC, ERC32, LEON, Mongoose-V	rtems.com
RT-Thread	GNU GPLv2	open source	embedded	active	ARM, ARM Cortex-M0-M3-R4-M4-M7, IA-32, AVR32, Blackfin, nios, PPC, M16C, MIPS (loongson-1b-1c, PIC32, xburst), MicroBlaze, V850, unicore32,	code.google.com/p/rt-thread, www.rt-thread.org
scmRTOS	Free	open source	embedded	active	ARM, Cortex-M3, Blackfin, MSP430, AVR, STM8	scmrtos.sourceforge.net
SDPOS	GNU LGPL	open source	embedded	active	ARM, Cortex-M3, Blackfin, PIC18, PIC24, i386 win32/linux synthetic targets	www.sdpos.org
silRTOS	Free	open source	embedded	active	ARM, Cortex-M3 Cortex-A9 ARM-11MP, Blackfin, MIPS, AVR	spanidea.com/products.php
StateOS	MIT	open source	embedded, small footprint	active	ARM Cortex-M, STM8	github.com/stateos/StateOS
Stratify OS	Modified GNU GPLor proprietary	open source	embedded	active	ARM Cortex-M3, M4	stratifylabs.co/Stratify-
TizenRT	Apache 2.0	open source	embedded	active	ARM	github.com/Samsung/TizenRT
TI-RTOS Kernel (SYS/BIOS)	BSD	open source	embedded	active	Mostly Texas Instruments: MSP430-432, C2000-5000-6000, TI's ARM families (Cortex M3-4F-R4-A8-A15), SimpleLink Wireless CC2xxx-CC3xxx	www.ti.com/tool/sysbios
Tock OS	Apache 2.0/MIT	open source	embedded	active	ARM Cortex	https://www.tockos.org
Trampoline Operating System (OSEK and AUTOSAR)	GNU GPLv2	open source	embedded	active	AVR, H8-300H, POSIX, NEC V850e, ARM7, Infineon C166, HCS12 or PowerPC	trampoline.rts-software.org

TNKernel	BSD	open source	embedded	active	ARM, PIC24-32-dsPIC, HCS08, STM32 (Cortex-M3)	www.tnkernel.com
TNeo	BSD	open source	embedded	active	Cortex-M0-M0+-M1-M3-M4-M4F, PIC24-32-dsPIC	github.com/dimonomid/tneo
XMK	BSD	open source	embedded	inactive-beta	?	www.integerfox.com/xmk
Y@SOS	GNU LGPL	open source	embedded	active	Cortex-M3, STM32	www.yasos.org
MontaVista Linux	GNU GPL	open source	embedded	active	?	www.mvista.com
uOS	GNU GPL	open source	embedded	active	AVR, ARM, MIPS32, MSP430, Intel i386, Linux386	code.google.com/p/uos-embedded/wiki/ab
Zephyr	Apache 2.0	open source	embedded	active	x86, ARM, ARC, NIOS2, XTensa, RISC V 32	www.
Apache Mynewt	Apache 2.0	open	embedded	active	ARM Cortex-M, MIPS32, Microchip PIC32, RISC-V	mynewt.apache.org
UNOS	?	?	?	historic	680x0	?

ANEXO B – Código Fonte de exemplo do Arduino

Segue listagem do código obtido a partir da mudança do sketch de exemplo do arduino, chamado "AnalogRead_digitalRead" para ler várias entradas e reiniciar a execução de uma task suspensa, quando uma entrada digital foi acionada:

```
    #include <Arduino_FreeRTOS.h>

    #include <semphr.h> // add the FreeRTOS functions for Semaphores (or
Flags).



    //tipo de parametros

    typedef struct {

    const char *pcNomeTask; //nome da task A SER CRIADA

    const char *pcTexto; //texto a ser enviado pela serial antes da
leitura, identificando sua fonte

    int SensorID; //ID da entrada analógica usada

    int TempoDelayTicks; //tempo entre envio de leituras, em ticks

    } AnalogReadParametro_t;



    // Declare a mutex Semaphore Handle which we will use to manage the
Serial Port.

    // It will be used to ensure only one Task is accessing this resource
at any time.

    SemaphoreHandle_t xSerialSemaphore;



    // define two Tasks for DigitalRead & AnalogRead

    void TaskDigitalRead( void *pvParameters );

    //aceita como parametro AnalogReadParametro_t

    void TaskAnalogReadParam( void *pvParameters );



    //define parametros para três tasks de leitura analógica que aceitam
paramtros de estrutura
```

```
    AnalogReadParametro_t xParams[3], *pxParam = xParams;




    // the setup function runs once when you press reset or power the board

    void setup() {



    // initialize serial communication at 9600 bits per second:

    Serial.begin(9600);

    while (!Serial) {

    ; // wait for serial port to connect. Needed for native USB, on
LEONARDO, MICRO, YUN, and other 32u4 based boards.

    }



    // Semaphores are useful to stop a Task proceeding, where it should be
paused to wait,

    // because it is sharing a resource, such as the Serial port.

    // Semaphores should only be used whilst the scheduler is running, but
we can set it up here.

    if ( xSerialSemaphore == NULL ) // Check to confirm that the Serial
Semaphore has not already been created.

    {

    xSerialSemaphore = xSemaphoreCreateMutex(); // Create a mutex semaphore
we will use to manage the Serial Port

    if ( ( xSerialSemaphore ) != NULL )

    xSemaphoreGive( ( xSerialSemaphore ) ); // Make the Serial Port
available for use, by "Giving" the Semaphore.

    }




    //define parametros para tres tasks de leitura analógica que acietam
parametros de estrutura
```

```
    pxParam->pcNomeTask = "TaskEntradaA1";

    pxParam->pcTexto = "Entrada A1";

    pxParam->SensorID = A1;

    pxParam->TempoDelayTicks = pdMS_TO_TICKS( 250UL );

    pxParam++;


    pxParam->pcNomeTask = "TaskEntradaA2";

    pxParam->pcTexto = "Entrada A2";

    pxParam->SensorID = A2;

    pxParam->TempoDelayTicks = 1; //pdMS_TO_TICKS( 250UL );

    pxParam++;


    pxParam->pcNomeTask = "TaskEntradaA3";

    pxParam->pcTexto = "Entrada A3";

    pxParam->SensorID = A3;

    pxParam->TempoDelayTicks = 1; //pdMS_TO_TICKS( 250UL );

    pxParam++;




    //xTaskCreate(TaskAnalogReadParam, pxParam->pcNomeTask, 128, (void
*)&xParams[0], 1 /* Priority */, NULL );

    //xTaskCreate(TaskAnalogReadParam, pxParam->pcNomeTask, 128, (void
*)&xParams[1], 1 /* Priority */, NULL );

    //xTaskCreate(TaskAnalogReadParam, pxParam->pcNomeTask, 128, (void
*)&xParams[2], 1 /* Priority */, NULL );

    //armazena a última task criada

    TaskHandle_t xUltimaTask;

    //cria as tasks percorre array de ponteiros até o nulo (para facilitar)
```

```
    for(pxParam = &xParams[0]; pxParam < &xParams[3]; pxParam++)

    {

    Serial.println("\n---------------------------------");

    Serial.print("Criando ");

    Serial.println(pxParam->pcNomeTask);

    xTaskCreate(TaskAnalogReadParam, pxParam->pcNomeTask, 128, (void
*)pxParam, (configMAX_PRIORITIES - 2) /* Priority */, &xUltimaTask );

    }



    vTaskSuspend(xUltimaTask);



    // Now set up two Tasks to run independently.

    xTaskCreate(

    TaskDigitalRead

    , (const portCHAR *)"DigitalRead" // A name just for humans

    , 128 // This stack size can be checked & adjusted by reading the Stack
Highwater

    , (void *)xUltimaTask //Passa o valor do hadle da task com cast para
(void *)

    , (configMAX_PRIORITIES - 3) // Priority, with 3 (configMAX_PRIORITIES
- 1) being the highest, and 0 being the lowest.

    , NULL );



    // Now the Task scheduler, which takes over control of scheduling
individual Tasks, is automatically started.

    }



    void loop()

    {

    // Empty. Things are done in Tasks.
```

```
}

void SerialDebugComSemaforo(const char *pszTexto)
{
if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) == pdTRUE )
{
Serial.print(pszTexto);

xSemaphoreGive( xSerialSemaphore ); // Now free or "Give" the Serial
Port for others.
}
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/

void TaskDigitalRead( void *pvParameters) // This is a Task.
{
/*
DigitalReadSerial
Reads a digital input on pin 2, prints the result to the serial monitor

This example code is in the public domain.
*/

// digital pin 2 has a pushbutton attached to it. Give it a name:
uint8_t pushButton = 2;
```

```
    TaskHandle_t xTaskParaReiniciar = (TaskHandle_t) pvParameters;


    // make the pushbutton's pin an input:
    pinMode(pushButton, INPUT);


    for (;;) // A Task shall never return or exit.
    {
    // read the input pin:
    int buttonState = digitalRead(pushButton);


    // See if we can obtain or "Take" the Serial Semaphore.
    // If the semaphore is not available, wait 5 ticks of the Scheduler to
see if it becomes free.
    if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) == pdTRUE )
    {
    //Envia o texto recebido por parâmetro
    Serial.print("entrada digital: ");


    // We were able to obtain or "Take" the semaphore and can now access
the shared resource.
    // We want to have the Serial Port for us alone, as it takes some time
to print,
    // so we don't want it getting stolen during the middle of a
conversion.
    // print out the state of the button:
    Serial.println(buttonState);


    xSemaphoreGive( xSerialSemaphore ); // Now free or "Give" the Serial
Port for others.
```

```
    //Decide se reinicia a task
    if( (xTaskParaReiniciar != NULL) && (buttonState == 1) )
    {
    vTaskResume(xTaskParaReiniciar);
    }
    }

    vTaskDelay(1); // one tick delay (15ms) in between reads for stability
    }
    }


    void TaskAnalogReadParam( void *pvParameters ) // This is a Task.
    {
    //pegando o parametro como o texto da string
    AnalogReadParametro_t *pxParams = (AnalogReadParametro_t *)
pvParameters;

    for (;;)
    {
    // read the input on analog pin:
    int sensorValue = analogRead(pxParams->SensorID);

    // See if we can obtain or "Take" the Serial Semaphore.
    // If the semaphore is not available, wait 5 ticks of the Scheduler to
see if it becomes free.
    if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) == pdTRUE )
    {
```

```
    //Envia o texto recebido por parâmetro

    Serial.print(pxParams->pcTexto);

    Serial.print(": ");

    // We were able to obtain or "Take" the semaphore and can now access
the shared resource.

    // We want to have the Serial Port for us alone, as it takes some time
to print,

    // so we don't want it getting stolen during the middle of a
conversion.

    // print out the value you read:

    Serial.println(sensorValue);



    xSemaphoreGive( xSerialSemaphore ); // Now free or "Give" the Serial
Port for others.



    }



    vTaskDelay(pxParams->TempoDelayTicks); // one tick delay (15ms) in
between reads for stability

    }

    }
```