UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE ENGENHARIA DE CONTROLE E AUTOMAÇÃO

IMPLEMENTAÇÃO DE INTERFACE DE COMUNICAÇÃO SEGUINDO NORMA IEC 61400-25 PARA AEROGERADORES

TRABALHO DE CONCLUSÃO DE CURSO

Juliano Grigulo

Santa Maria, RS, Brasil
2014

IMPLEMENTAÇÃO DE INTERFACE DE COMUNICAÇÃO SEGUINDO NORMA IEC 61400-25 PARA AEROGERADORES

por

Juliano Grigulo

Trabalho de conclusão de curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Maria (UFSM, RS)

Orientador: Prof. Dr. Frederico Menine Schaf

Santa Maria, RS, Brasil
2014

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Engenharia de Controle e Automação

A Comissão Examinadora, abaixo assinada, aprova o Trabalho de Conlusão de Curso

IMPLEMENTAÇÃO DE INTERFACE DE COMUNICAÇÃO SEGUINDO NORMA IEC 61400-25 PARA AEROGERADORES

elaborada por
Juliano Grigulo

como requisito parcial para obtenção do grau de
Engenheiro de Controle e Automação

COMISSÃO EXAMINADORA:

_____
Frederico Menine Schaf, Dr.
(Presidente/Orientador)
(UFSM)

_____
Claiton Moro Franchi, Dr.
(UFSM)

_____
Humberto Pinheiro, Ph.D.
(UFSM)

Santa Maria, 19 de Dezembro de 2014.

AGRADECIMENTOS

Agradeço a Deus, acima de tudo, por sempre ter me dado forças e motivação para que este trabalho pudesse ser concluído.

Ao meu orientador, Prof. Frederico Menine Schaf Dr. Eng., pelo conhecimento transmitido e orientação, que foram importatíssimos para este trabalho e para minha formação acadêmica.

À meus pais, pela educação, incentivo e suporte durante toda minha vida.

À minha namorada Ana Júlia Socal, dedico este Trabalho de Conclusão de Curso à ela. Que sempre esteve presente comigo, me apoiando e incentivando, nas horas mais difíceis. Me ajudando sempre que pudesse.

Aos amigos do Grupo de Eletrônica de Potência e Controle (GEPOC), pelo apoio e incentivo. Em especial, aos meus amigos Benhur Tessele, Fabrício Cazakevicius e Henrique Horst Figuera pelo companheirismo e ajuda em todos os momentos difíceis.

"É muito melhor lançar-se em busca de conquistas grandiosas, mesmo expondo-se ao fracasso, do que alinhar-se com os pobres de espírito, que nem gozam muito nem sofrem muito, porque vivem numa penumbra cinzenta, onde não conhecem nem vitória, nem derrota."

Theodore Roosevelt

RESUMO

Trabalho de Conclusão de Curso
Engenharia de Controle e Automação
Universidade Federal de Santa Maria

IMPLEMENTAÇÃO DE INTERFACE DE COMUNICAÇÃO SEGUINDO NORMA IEC 61400-25 PARA AEROGERADORES

AUTOR: JULIANO GRIGULO
ORIENTADOR: FREDERICO MENINE SCHAF
Data e Local da Defesa: Santa Maria, 19 de Dezembro de 2014.

Com o desenvolvimento econômico acelerado do país, a perspectiva é de um aumento significativo no consumo de energia para os próximos anos. Em função disto, torna-se importante a busca de novas tecnologias e alternativas viáveis para a diversificação da matriz energética brasileira. A geração de energia através do vento, chamada geração eólica, apresenta-se como a energia alternativa mais viável atualmente, em função de sua eficiência e custo (se comparado com a energia solar, por exemplo). A comunicação entre os componentes e dispositivos de uma planta eólica é essencial para uma operação segura e eficiente. Isto inclui aquisição de dados e supervisão do sistema. A norma IEC 61400-25 especifica os componentes da rede de comunicação para plantas eólicas, como protocolos, modelos de informação e etc. O protocolo DNP3, inicialmente utilizado por concessionárias de energia elétrica, empresas de óleo/gás, água/esgoto e indústria de segurança, é um grande candidato para implementação da norma IEC 61400-25, pois apresenta grandes vantagens para a aplicação em sistemas de geração de energia eólica.

Este trabalho apresenta o projeto, desenvolvimento e implemetação da rede de comunicação de um aerogerador, seguindo as diretrizes da norma IEC 61400-25, por meio dos protocolos DNP3 e TCP/IP. Resultados experimentais são apresentados ao final, comprovando a funcionalidade da interface de comunicação.

Palavras-chave: IEC 61400-25, *Smart Grid*, Energia eólica, sistemas embarcados, DNP3, RTOS, *Raspberry Pi.*

ABSTRACT

Degree Conclusion Project
Bachelor of Control and Automation Engineering
Federal University of Santa Maria

WIND POWER PLANT COMUNICATION INTERFACE THROUGH DNP3 PROTOCOL FOLLOWING IEC Std 61400-25

AUTHOR: JULIANO GRIGULO
ADVISER: FREDERICO MENINE SCHAF
Defense Place and Date: Santa Maria, December 19th, 2014.

With the fast economic development of the country, the energy consumption tends to increase significantly for the next years. Because of that, it becomes important to search for new alternatives technologies that are practicable for the diversification of the brasilian energy matrix. The generation of electricity from wind, called wind energy, presents itself as the most viable alternative energy nowadays, because of its cost-efficiency (if compared with solar energy). The communication between the components and devices in a wind power plant is an essential feature for a safe and efficient operation. This includes data acquisition and system supervision. The IEC 61400-25 standard specifies the communications for monitoring and control of wind power plants, like communication protocols, information models and others. The DNP3 protocol (initially used for electric utility, oil & gas, water/waste water and security industries) is a great candidate for the 61400-25 implementation, because it presents good advantages over other protocols for wind energy applications.

Key words: IEC 61400-25, Smart Grid, Wind Energy, Embedded Systems, DNP3, RTOS, *Raspberry Pi.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ANEXOS

SUMÁRIO

1. INTRODUÇÃO

Com o grande crescimento da demanda por energia elétrica, devido principalmente ao crescimento econômico do país[1], aerogeradores vêm se apresentando como uma alternativa viável para a diversificação da matriz energética brasileira, apresentando-se como uma solução para os problemas decorrentes da fragilidade do sistema de produção de energia hidroelétrico, maior produtor de energia elétrica brasileiro, frente às mudanças climáticas dos últimos tempos (períodos prolongados de secas)[2].

O Brasil possui um alto potencial para produção de energia proveniente dos ventos, principalmente nos litorais Nordeste e Sul e extremo sul do país[3]. Áreas quais vêm recebendo grande investimento nos últimos anos para produção de energia eólica. Frente à necessidade emergente de produção de energia e dos investimentos na área, nasce a justificativa para o investimento em pesquisa e desenvolvimento nacionais na área, pois grande parte da tecnologia atual é importada. Estes investimentos incentivam o crescimento econômico do país, e provocam economia com gastos oriundos de importações de tecnologias e produtos.

Este trabalho engloba a rede de comunicação do aerogerador, entre a nacele (localizada no topo da torre do aerogerador), o PCC (ponto de acoplamento comum, localizado na base) e uma interface SCADA (*Supervisory Control and Data Acquisition*) remota, utilizando um protocolo de comunicação que cumpra os requisitos da norma IEC 61400-25.

1.1 PROBLEMÁTICA

Sistemas de geração distribuída, como fazendas eólicas, compõem-se de um sistema complexo, multivariável, onde alguns dispositivos de controle e monitoramento devem comunicar-se para que haja uma sincronia nas atividades desempenhadas. Em aerogeradores, a comunicação comumente esteve presente entre os sistemas de controle, sistemas de monitoramento e sistemas supervisórios. Grande parte dos sistemas de monitoramento e controle usados na geração eólica são proprietários e, consequentemente, os protocolos e os dados fornecidos por estes sistemas são altamente dependentes de cada fabricante (SAN TELMO, 2007). Pensando nestes problemas, os principais fabricantes de equipamentos no

---

[1] Leitura Ministério de Minas e Energia, "Panorama Energético Brasileiro", 2008

[2] Consumo de energia elétrica no Brasil teve crescimento de 3,5% em 2013, segundo correio braziliense.

[3] "Atlas de Energia Elétrica - 2ª Edição", Agência Nacional de Energia Elétrica (ANEEL). 2008.

mercado se reuniram através da *International Electrotechnical Commission* (IEC) e criaram a IEC 61400-25, uma norma de comunicação projetada para geração eólica.

Na sessão IEC 61400-25-4, cinco protocolos existentes que conformam com a norma são definidos:

- MMS (*Manufacturing Message Specification*)
- OPC-XML-DA (*Object Linking and Embedding for Process Control (OPC) Foundation*)
- *Web* Services
- IEC 60870-5-104
- DNP3 (*Distributed Network Protocol*)

1.2 ESTRUTURA DO TRABALHO

O presente trabalho de conclusão de curso estrutura-se conforme descrição à seguir.

Capítulo 2: REVISÃO BIBLIOGRÁFICA. Histórico das redes industrias e protocolos de comunicação, modelo ISO/OSI, histórico do protocolo DNP3, protocolo TCP/IP, explicação sobre a IEC 61400-25**,** hipótese de solução para o problema enfrentado.

Capítulo 3: OBJETIVOS. Objetivos almejados ao fim do trabalho.

Capítulo 4: MATERIAIS E MÉTODOS. Apresenta topologia e estrutura da rede, dispositivos utilizados para implementação da mesma, especifica protocolo DNP3 e suas camadas, introdução à biblioteca aberta *openDNP3*.

Capítulo 5: RESULTADOS EXPERIMENTAIS. Apresentação dos resultados alcançados, configuração do Mestre e Escravo, desenvolvimento de aplicação DNP3, desenvolvimento da aplicação TCP/IP.

Capítulo 6: CRONOGRAMA. Lista de atividades realizadas e cumpridas durante a execução do trabalho.

Capítulo 7: CONCLUSÃO. Encerramento, conclusões e comentários sobre o trabalho desenvolvido.

Capítulo 8: REFERÊNCIAS BIBLIOGRÁFICAS

2. REVISÃO BIBLIOGRÁFICA

Neste capítulo serão expostos os fundamentos de redes industriais, bem como o padrão de interconexão de sistemas abertos (modelo OSI). Em seguida, o protocolo DNP3 e TCP/IP, o padrão IEEE (*Institute of Electrical and Electronics Engineers*) para aritmética de ponto flutuante e a norma IEC 61400-25 serão introduzidos teoricamente.

2.1 HISTÓRICO: REDES INDUSTRIAIS E PROTOCOLOS DE COMUNICAÇÃO

No início do século vinte os sistemas de controle de processos eram dominados pela tecnologia mecânica e por dispositivos analógicos como chaves contatoras, relés e *timers* (dispositivos contadores de tempo).

Controladores eletrônicos possibilitaram o controle de sistemas maiores nos anos 50 com o uso de conversores eletropneumáticos. Também nos anos 50 foram criadas as primeiras redes industriais para sistemas de controle, redes analógicas (4-20mA) e ligavam controladores aos seus periféricos e terminais.

No início dos anos 60 surgiu o controle digital direto (DDC) com o primeiro emprego do computador na indústria. Nos anos 60 computadores eram ainda muito caros o que proporcionou o advento dos primeiros CLPs (Controladores Lógicos Programáveis) que substituíram os controladores baseados em relés.

Na metade dos anos 70 com o uso de redes que empregavam dados digitais surgiu os sistemas de controle computacional distribuídos (DCCS - *Distributed Computer Control Systems*). A Figura 1 mostra o conceito de DCCS.

Nos anos 80 popularizou-se o uso de redes de áreas locais (LAN) e consequentemente algumas redes tradicionais que são empregadas na indústria ainda hoje. (SCHAF, 2014)

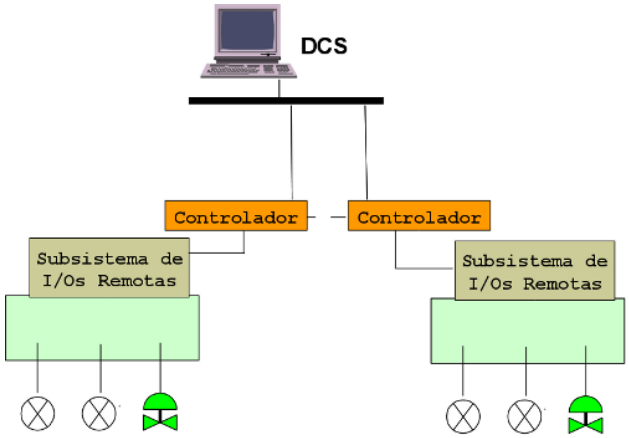


Figura 1- Sistema de Controle Distribuído. Fonte: SCHAF, 2014.

Decorrente da evolução na informática, chegada de sistemas operacionais multitarefas, e a complexidade dos processos industriais (necessitando de altas velocidades de comunicação, comunicação determinística, de várias topologias, segura e não suscetível à ruídos) nasceram os protocolos de comunicação, responsáveis por fazer a interface da aplicação do usuário com o meio físico, e vice-e-versa.

A indústria de produção e distribuição de energia elétrica acabou por seguir a tendência das redes industriais, sendo de grande aplicação à geração distribuída. Surgiram então os protocolos de comunicação para *smart grids* (do inglês: redes inteligentes), para o monitoramento, aquisição de dados e controle de dispositivos.

2.2 SISTEMAS ABERTOS E PADRÕES DE COMUNICAÇÃO

O modelo OSI, do inglês *Open Systems Interconnection* - interconexão de sistemas abertos, desenvolvido pela ISO (*International Standards Organization*- Organização Internacional de Padrões) é uma estrutura de comunicação que teve um tremendo impacto no projeto de sistemas de comunicação (TANENBAUM, 2003). O objetivo do modelo foi de prover uma estrutura para a coordenação no desenvolvimento de padrões de comunicação.

A conexão de um ou mais dispositivos é o primeiro passo para se estabelecer uma comunicação. Além de requisitos de *hardware*, os problemas de *software* devem ser solucionados, quando dispositivos de um mesmo fabricante são conectados os problemas de *software* são solucionados com facilidade, já que ambos foram projetados seguindo especificações semelhantes.

Sistemas abertos permitem que equipamentos de diversos fabricantes, que obviamente atendem ao padrão aberto, sejam intercambiáveis na rede. Os benefícios de tais sistemas incluem múltiplos fornecedores e uma alta disponibilidade de dispositivos, além de redução de custos e fácil integração com demais componentes.

Guiados de tal necessidade, em 1978 a ISO definiu o modelo de referência para comunicação entre sistemas abertos (ISO 7498), conhecido por ISO/OSI. O modelo OSI nada mais é que uma estrutura de comunicação de dados, que divide a comunicação de dados entre sete níveis hierárquicos. Cada nível possui suas próprias funções e serviços e faz a interface com seus vizinhos. A conformidade com o padrão OSI possibilita que um sistema possa se comunicar com qualquer outro sistema compatível.

O diagrama da Figura 2 mostra as sete camadas do modelo OSI.

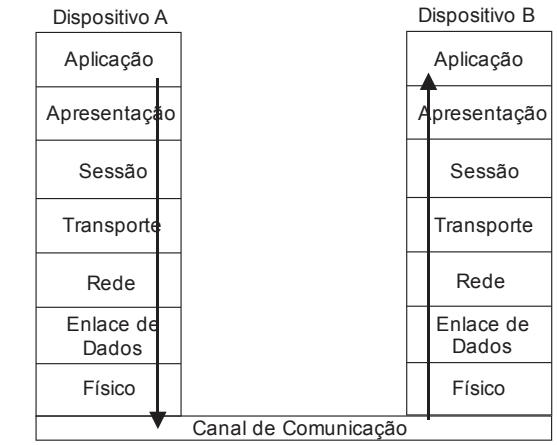


Figura 2 - Arquitetutra do modelo OSI.

Abaixo segue um breve resumo das camadas do OSI:

- **Aplicação**

  Prestação de serviços de rede para os programas de aplicação do usuário.

- **Apresentação**

  Encarrega-se, principalmente, da representação de dados (incluindo criptografia);

- **Sessão**

  Controle das comunicações entre os usuários.

- **Transporte**

  Gerenciamento das comunicações entre os dois sistemas finais.

- **Rede**

  Responsável pelo roteamento de mensagens.

- **Enlace de dados**

  Responsável pela montagem e envio de um quadro de dados de um sistema para outro.

- **Física**

  Define como e com que padrão os dados serão enviados e recebidos no meio físico do canal de comunicação.

2.3 DNP3 BREVE HISTÓRICO

Em 1988 a IEC começava a publicar um padrão chamado 'IEC 870 *Telecontrol equipment and systems*', que possuía na sua seção 5 a definição de protocolos de transmissão. O protocolo foi então definido em termos do modelo OSI possuindo um conjunto mínimo de

camadas; camada física, enlace de dados, e camada de aplicação. Durante este mesmo período, o protocolo DNP3, ou *Distributed Network Protocol* versão 3.0, estava sendo desenvolvido na América do Norte.

O DNP3 é um protocolo de comunicação desenvolvido pela divisão de *Distributed Automation Products* da *Harris Control,* durante os anos 90 e lançado pela DNP3 *Users Group* em Novembro de 1993. É o padrão de telecomunicações que define comunicações entre estações mestre, unidades remotas de telemetria (RTUs) e outros dispositivos eletrônicos inteligentes (IEDs). Foi desenvolvido para alcançar interoperabilidade entre sistemas de concessionárias de energia elétrica, óleo e gás, água/esgoto e indústrias de segurança (CLARKE, 2004).

DNP3 foi projetado especificamente para aplicações em sistemas SCADA. Englobando aquisição de informações e envio de comandos de controle entre computadores separados fisicamente, projetados para enviar pequenos pacotes de dados de uma maneira confiável e determinística.

2.4 PROTOCOLO TCP/IP

O TCP/IP é um conjunto de protocolos de comunicação entre computadores e dispositivos em rede, seu nome vem de dois protocolos, TCP (*Transmission Control Protocol* – Protocolo de Controle de Transmissão) e o IP (*Internet Protocol* – Protocolo de Internet/redes). É o padrão global de rede e camada de transporte utilizado na Internet, pela sua popularidade. A Internet foi parte de uma projeto militar comissionando pla *Advanced Research Projects Agency* (ARPA). O modelo de comunicação usado para construir o sistema é chamado de modelo ARPA.

Enquanto o modelo OSI foi desenvolvido na Europa pela *International Standards Organization* (ISO) e possui 7 camadas, o modelo ARPA foi desenvolvido nos EUA pela ARPA e possui 4 camadas. As camadas do modelo OSI são mapeadas dentro do modelo ARPA de acordo com a Figura 3.

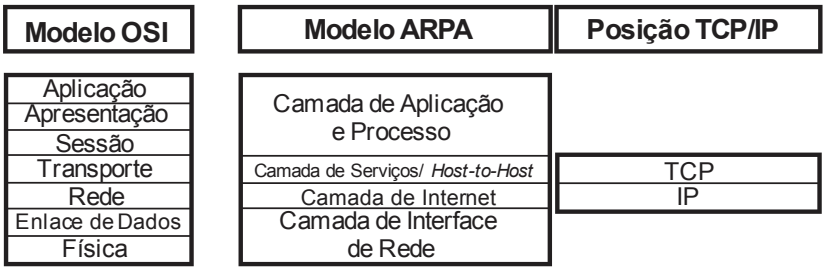


Figura 3 – Comparração entre modeo OSI e ARPA. Fonte: CLARKE, 2004.

O TCP/IP consiste de uma vasta gama de protocolos que ocupam as três camadas superiores do modelo ARPA. Além disso, o TCP/IP não possui a camada de interface de rede, mas depende da mesma para acessar o meio.

Como mostra a Figura 4 a seguir, o quadro de transmissão (*frame*) Internet originado em um computador específico conteria o cabeçalho e rodapé da rede local (normalmente *Ethernet*) aplicável ao mesmo. Ao percorrer a Internet, o cabeçalho e o rodapé podem ser substituídos dependendo do tipo de rede onde o pacote de dados se localiza (X.25, *frame relay* ou ATM). O datagrama IP manter-se-á intocável, a não ser que o memso tenha que ser fragmentado e reconstituído ao longo do percurso na rede.

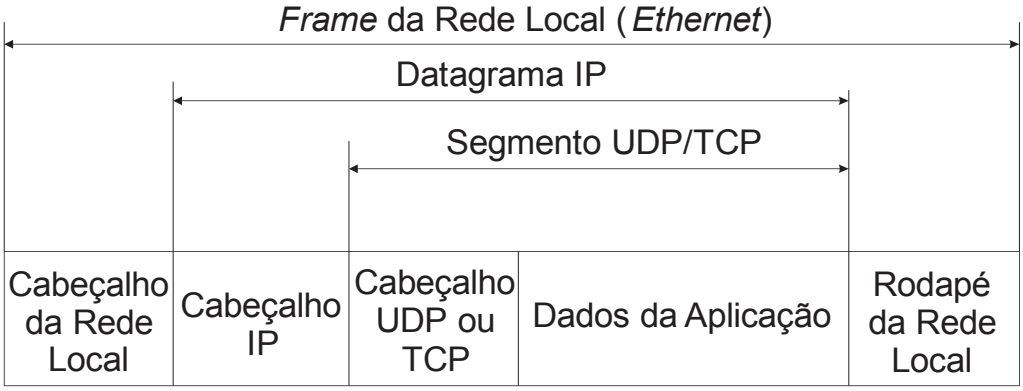


Figura 4 – *Frame* de Internet. Fonte: CLARKE, 2004.

2.4.1. A Camada de Internet

Essa camada é responsável pelo roteamento de pacotes de um dispositivo à outro. Cada pacote contém as informações de endereços necessárias para tal roteamento sobre a rede de Internet até o dispositivo de destino. O protocolo dominante neste nível é o protocolo de Internet (IP).

2.4.2. A Camada de Serviços

Essa camada é responsável pela integridade dos dados entre o dispositivo de envio e recebimento de dados, independente do caminho ou distância usado para transportar a mensagem. Possui dois protocolos associados:

- Protocolo de dados do usuário (UDP), um protocolo sem conexão (não confivável) usado para transmitir dados pouco sensíveis com o mínimo *overhead* (RFC 768).
- Protocolo de controle de transmissão (TCP), um protocolo orientado à conexão que oferece um método confiável para transmissão de um *stream* de dados entre aplicações (RFC 793).

2.4.3. A Camada de Aplicação e Processos

Essa camada fornece programas com interfaces à pilha TCP/IP ao usuário ou aplicação. Protocolos neste nível incluem: *file transfer protocol* (FTP), *trivial file transfer protocol* (TFTP), *simple mail transfer protocol* (SMTP), *telecommunications network* (TELNET), *post office protocol* (POP3)*, remote procedure calls* (RPC), *remote login* (RLOGIN), *hypertext transfer protocol* (HTTP), *network time protocol* (NTP) entre outros.

2.5 IEEE 754 – ARITMÉTICA DE PONTO FLUTUANTE

O IEEE 754 – padrão IEEE para aritmética de ponto flutuante, é uma norma técnica para cálculo de ponto flutuante estabelecida em 1985 pela IEEE. Existe uma imensidão de microcontroladores e componentes de *hardware* que utilizam a IEEE 754. O padrão veio solucionar muitos dos problemas relacionados às diversas implementações de cálculo em ponto flutuante, que as tornavam pouco confiáveis.

O padrão define:

- Formatos aritméticos: conjuntos de dados binários e decimais em ponto flutuante, consistindo de números finitos, infinitos e valores especiais para números nulos.
- Formatos intercambiáveis: codificação que deve ser usada para trocar dados em ponto-flutuante de maneria eficiente e compacta.
- Regras de arredodamento: propriedades que devem ser satisfeitas ao se arredondar números durante operações aritméticas e conversões.
- Operações: operações aritméticas e formatos.

- Manipulação de exceções: indicação de condições excepcionais (como divisão por zero, *overflow*).

2.6 IEC 61400-25

A norma IEC 61400-25 é uma adaptação da IEC 61850 (uma norma amplamente conhecida no setor elétrico e que foi projetada inicialmente para comunicação em subestações), com particularidades em tópicos de controle e monitoramento de plantas de geração eólica. A IEC 61400-25 é uma norma de comunicação e não somente um protocolo. Inclui especificações a respeito da velocidade do rotor, turbina e de outros componentes mais específicos do aerogerador (DE LUCA, 2013).

A norma é classificada em cinco diferentes tópicos:

- Descrição geral dos princípios e modelos (IEC 61400-25-1).
- Modelos de informação (IEC 61400-25-2).
- Modelos de trocas de informações (IEC 61400-25-3).
- Mapeamento do perfil de comunicação (IEC 61400-25-4).
- Teste de conformação (IEC 61400-25-5)
- Nós lógicos e classes de dados para monitoramento (IEC 61400-25-6).

A Figura 5 mostra uma visão geral da norma IEC 61400-25.

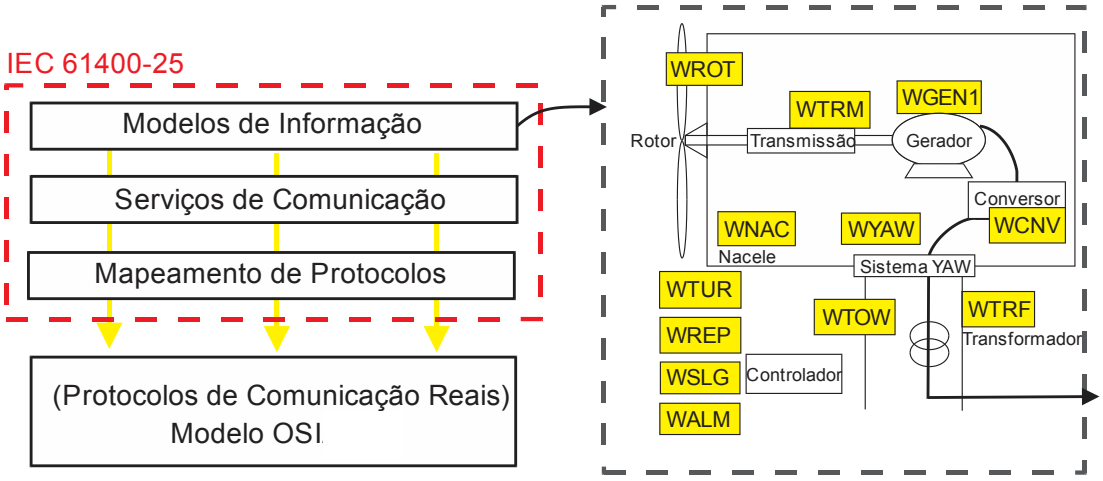


Figura 5- Visão geral da norma IEC 61400-25 e seu modelo de informação. Fonte: IEC 61400-25-2.

Na Figura 3 a estrutura da norma é dividida em três camadas. Cada uma destas camadas será melhor detalhada nas próximas subseções.

2.6.1. Modelos de Informação

Um dos principais avanços da norma para a interoperabilidade é a padronização do nome das informações trocadas no sistema. Cada componente do sistema (rotor, transmissão, conversor, entre outros) possui um objeto correspondente nas seções 2 e 6 da IEC 61400-25. Cada objeto possui um conjunto de dados que procura representar todas as informações que podem ser necessárias para o componente que ele está representando. Por exemplo, "Turbine_01/WROT1.MX.RotSpd.instMag.f" representa a velocidade do rotor em uma turbina chamada de "Turbine_01". De um modo mais geral, os modelos de informação definidos pela norma procuram trazer semântica às informações trocadas no sistema.

2.6.2. Serviços de Comunicação

Nesta camada, a norma define como as informações serão trocadas no sistema, para isso, os serviços de comunicação são definidos na parte 3 e da IEC 61400-25. Alguns serviços são listados a seguir.

- **Relatórios**
  Mensagens enviadas quando algum evento ocorre no sistema. Este tipo de serviço é o principal responsável pelo envio do SoE (*Sequence of Events*), necessário na maioria dos sistemas.
- **Log**
  Uma entrada é inserida em um *log* local, que pode ser acessado por qualquer sistema que queira conhecer eventos ocorridos num intervalo.
- **Comando**
  Estabelece quatro modos de comando, para vários tipos, como booleanos, inteiros, pontos flutuantes, etc.

Além destes serviços, existem outros de sincronização de relógio, *polling,* serviços de autodescrição dos modelos de informação entre vários outros.

2.6.3. Mapeamento para Protocolos de Comunicação

Camada que define como os modelos de informação e os serviços serão mapeados para protocolos de comunicação existentes. Através da IEC 61400-25-4 cinco protocolos são definidos.

- **MMS**
- **OPC-XML-DA**
- ***Web services***
- **IEC 60870-5-104**
- **DNP3**

Quando usado pela norma, um protocolo apenas encapsula as informações da própria norma.

3. OBJETIVOS

Este trabalho tem por objetivo a implementação do protocolo de comunicação DNP3 (IEEE Std 1815-2010) seguindo as exigências da norma 61400-25 de comunicação para aerogeradores e a comunicação do aerogerador como um todo (desde a aquisição de dados). Tal implementação será realizada em um processador contido no computador *Raspberry Pi Model-B*, um sistema SCADA e dois kits de *Texas Instruments Concerto.*

A transferência de dados entre o aerogerador e a aplicação SCADA remota se dará através da rede *Ethernet* utilizando o protocolo DNP3. A *Raspberry Pi* se comunicará com os DSPs *Concerto* por meio de protocolo TCP/IP. Um dos *kits Concerto* fará parte da *nacelle* do aerogerador, onde fará leitura de variáveis de: tensão e corrente no gerador; velocidade do vento; orientação da turbina segundo direção do vento; velocidade do rotor; potência gerada; entre outros. Além disso, ainda será responsável por atuar no sistema de YAW (orientação da turbina), do atuador de passo (ângulo de pás da turbina) e no freio de emergência. O segundo *kit Concerto* se encontrará no painel de controle, na base da torre, este responsável pelo controle do inversor, injeção de energia na rede elétrica e monitoramento de potência injetada.

Ambos os DSPs comunicar-se-ão com um computador *Raspberry Pi*, este último, consequentemente, se comunica e envia informações à uma interface SCADA remota, além de receber informações relativas ao status da rede e comandos do usuário por meio do painel de controle e de uma interface IHM localizados na base da torre do aerogerador.

Ainda é previsto a comunicação do sistema de geração eólica com uma unidade remota (laboratório) por meio da Internet ou através de rede GPRS/3G, enviando dados para um servidor, relativos à geração e *status* do sistema como um todo.

3.1 HIPÓTESE

Alguns protocolos de comunicação existentes obedecem à IEC 61400-25, um deles, *Distributed Network Protocol* (DNP3) é objeto de estudo deste trabalho, pois se apresenta como uma solução para o problema apresentado na seção 1.1. Sendo um protocolo não proprietário (aberto), confiável, interoperável, seguro, determinístico e que dispõem de bibliotecas *open-source* confiáveis.

Para implementação do protocolo DNP3 será utilizado o kit de desenvolvimento da *Raspberry Pi Foundation*, o computador de uma placa *Raspberry Pi Model B* (computador *Raspberry Pi* Modelo-B Rev1). A *Raspberry Pi Model-B* é um computador do tamanho de um

cartão de crédito que possui um circuito integrado em um *chip* denominado Broadcom BCM2835 (BROADCOM, 2012), que possui um processador ARM1176JZF-S de 700 MHz (ARM, 2009), processador de vídeo *dual-core* VideoCore IV GPU e 512 MB de memória RAM. Outra versão da mesma placa é a *Raspberry Pi Model-A* que possui configurações semelhantes à posterior, porém com 256 MB de memória RAM, um único conector USB e não dispõe do conector de *Ethernet* (RASPBERRY, 2014).

Além da *Raspberry Pi*, farão parte da rede de comunicação dois DSPs *Concerto* da *Texas Instruments* (Kits TMDXDOCK28M36) responsáveis pelo controle em tempo real dos conversores e aquisição de dados dos sensores. Cada kit *Concerto* possui um microcontrolador de dois núcleos, correspondendo a um processador digital de sinais (DSP TMS320F28335), popular em aplicações industriais e um núcleo ARM (Cortex-M3), microcontrolador de alto desempenho, utilizado para serviços de comunicação e aplicações de usuário. O primeiro núcleo, DSP TMS320F28335, será utilizado para tarefas de controle, acionamento e aquisição de dados, já o segundo núcleo, ARM CORTEX-M3, será responsável pela comunicação TCP/IP com o computador *Raspberry Pi.*

O sistema então será composto de dois Kits *Concerto* TMDXDOCK28M36, um deles localizados na *nacelle* e o outro na base da torre, comunicando via protocolo TCP/IP sobre rede *Ethernet*, um computador *Raspberry Pi*, comunicando TCP/IP com os DSPs e DNP3 com um sistema SCADA remoto.

Após implementação do protocolo DNP3, será incluído o modelo abstrato (modelos de informação) da norma IEC 61400-25, assim, a rede de comunicação estará atendendo aos padrões internacionais.

4. MATERIAIS E MÉTODOS

Esta seção irá descrever o ambiente de testes, ferramentas e métodos utilizados para implementar o protocolo DNP3 no sistema embarcado utilizado (*Raspberry Pi*).

4.1 ARQUITETURA GLOBAL DO SISTEMA

Considerando que o sistema de comunicação será aplicado a uma turbina eólica de eixo horizontal, um *kit* de desenvolvimento *Concerto* estará presente na nacele e outro estará presente em um gabinete na base da torre do aerogerador juntamente com um *kit Raspberry Pi*, ambos conectados através de um *switch*/roteador, que também se comunicam com um sistema SCADA remoto. Além disto, o sistema ainda inclui uma IHM presente na base da torre, para efeitos de monitoramento e controle local.

Um diagrama mais geral do sistema é apresentado na Figura 6.

O sistema então consiste dos seguintes elementos:

- Interface de controle e aquisição de dados com barramento de comunicação.
- Barramento de comunicação *Ethernet* e DNP3 por meio de roteador.
- Conversão de meio físico RJ-45 para fibra ótica, por propósitos de isolação, entre a base da torre e a nacele.
- Dois *kits Concerto*, responsáveis pelo processamento e controle do sistema.
- Uma interface IHM.
- Um *kit Raspberry Pi*, responsável pela transmissão dos dados ao sistema SCADA.
- Sistema SCADA remoto.

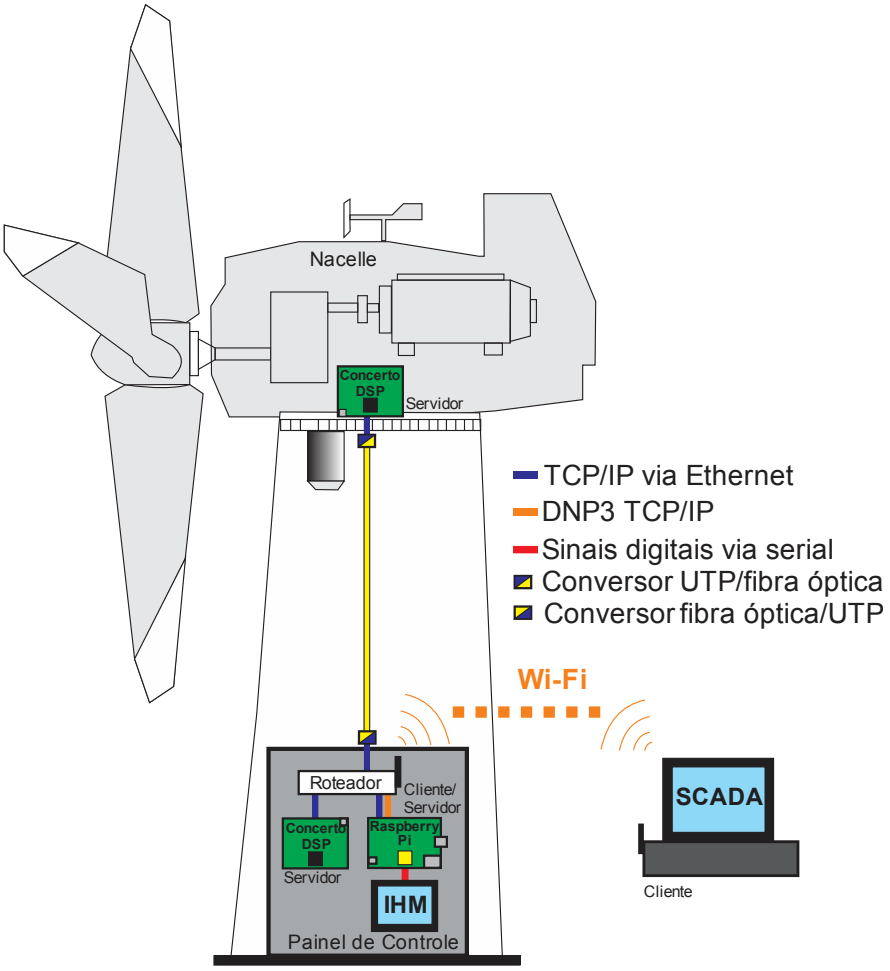
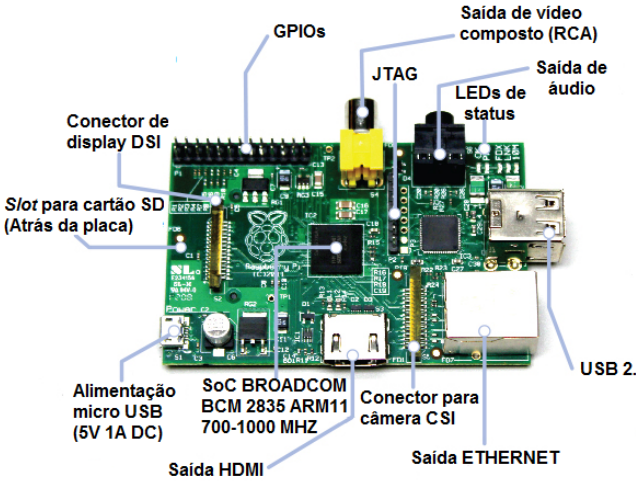


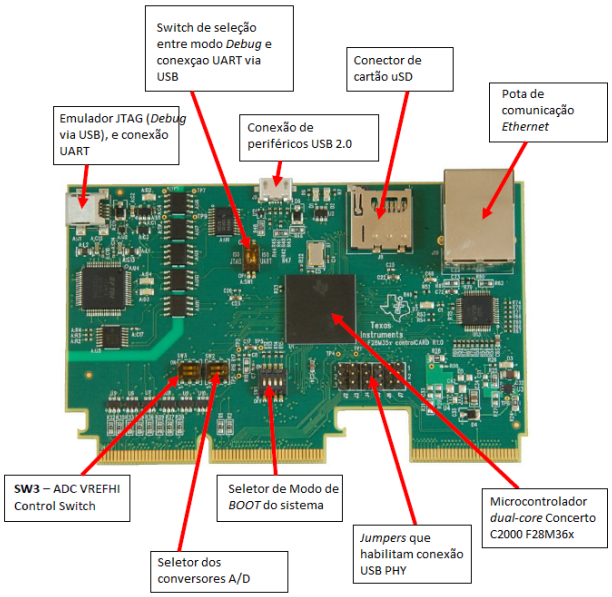
Figura 6- Diagrama geral do sistema de comunicação no aerogerador.

Como é possível visualizar por meio da Figura 6, O sistema configura uma comunicação de topologia do tipo estrela. Os microcontroladores da *nacelle* e do painel de controle, então, realizam tarefas de controle em tempo real e aquisição de dados, enviando-os para a *Raspberry Pi* por meio do protocolo TCP/IP sobre *Ethernet*, a *Raspberry Pi* por sua vez efetua requisições aos *kits Concerto* e se comunica com um sistema SCADA remoto por meio de protocolo DNP3 TCP/IP sobre *Ethernet*. Entre a *nacelle* e o roteador a conexão de meio físico deve ser isolada, portanto, optou-se pela adoção de conversores UTP/STP 100BaseT para fibra ótica (100BaseF), podendo assim utilizar o meio físico de fibra ótica, que possui a vantagem de ser isolado eletricamente. Neste caso, se houver um surto de tensão na *nacelle* (provocado por relâmpagos, por exemplo), a corrente não fluirá entre a nacele e o sistema de controle, evitando assim que os dispositivos eletrônicos venham a ser danificados.

4.2 AMBIENTE DE TESTES



(a) Computador *Raspberry Pi Model-B*



(b)*Kit* de desenvolvimento *TMDXDOCK28M36*

(c) Roteador TPLINK WR841N

(d) *software* Elipse SCADA

Figura 7- Dispositivos da rede de comunicação

Para implementação do protocolo de comunicação foi utilizado um kit *Raspberry Pi* (Figura 7.a), dois kits TMDXDOCK28M36 (*Concerto F28M36x Experimenter's Kit*) (Figura 7.b) um roteador TPLINK (Figura 7.c) e um computador com o software Elipse SCADA (Figura 7.d). O *hardware* do sistema embarcado utilizado possui seguintes configurações:

4.2.1. Computador *Raspberry Pi* Modelo B

A Tabela 1 abaixo descreve as especificações e componentes de *hardware* da *Raspberry Pi Model-B*.

Tabela 1- Especificações da *Raspberry Pi*

Circuito Integrado (System-on-Chip)	Broadcom BCM2835 (CPU, GPU, DSP, SDRAM e porta USB)
CPU	Núcleo ARM1176JZF-S de 700 - 1000 MHz (família ARM11, instruções ARMv6), 32-bit RISC. Possui unidade de processamento de ponto flutuante (FPU), unida de gerenciamento de memória (MMU) e suporte à sistemas operacionais como LINUX
GPU	Broadcom VideoCore IV @ 500 MHz OpenGL ES 2.0 (24 GFLOPS) MPEG-2 e VC-1 ,1080p à 30fps h.264/MPEG-4 AVC
Memória SDRAM	512 MB (compartilhada com a GPU)
Portas USB 2.0	2 portas USB
Entrada de vídeo	Conector MIPI CSI-2 (*Camera Serial Interface*) para módulo de câmera digital
Saídas de vídeo	HDMI (1.3 e 1.4), RCA composto (PAL e NTSC), painel LCD via conector serial
Saída de áudio	Conector 3.5 mm, HDMI e áudio I²S
Armazenamento	*Slot* para cartões SD / MMC / SDIO
Rede integrada	10/100 Mbit/s Ethernet
Periféricos de baixo nível	8 GPIO, UART, barramento I²C bus, barramento SPI, audio I²S, +3.3 V, +5 V, terra
Consumo	700 mA (3.5 W)
Fonte de potência	5 V via microUSB ou GPIO
Sistemas operacionais	Arch Linux ARM, Debian GNU/Linux, Gentoo, Fedora, FreeBSD, NetBSD, Plan 9, Inferno, Raspbian OS, RISC OS, Slackware Linux, Android

A Figura 8 abaixo mostra a pinagem dos conector de GPIOs (entradas e saídas de propósito geral) e periféricos de comunicação do computador *Raspberry Pi.*

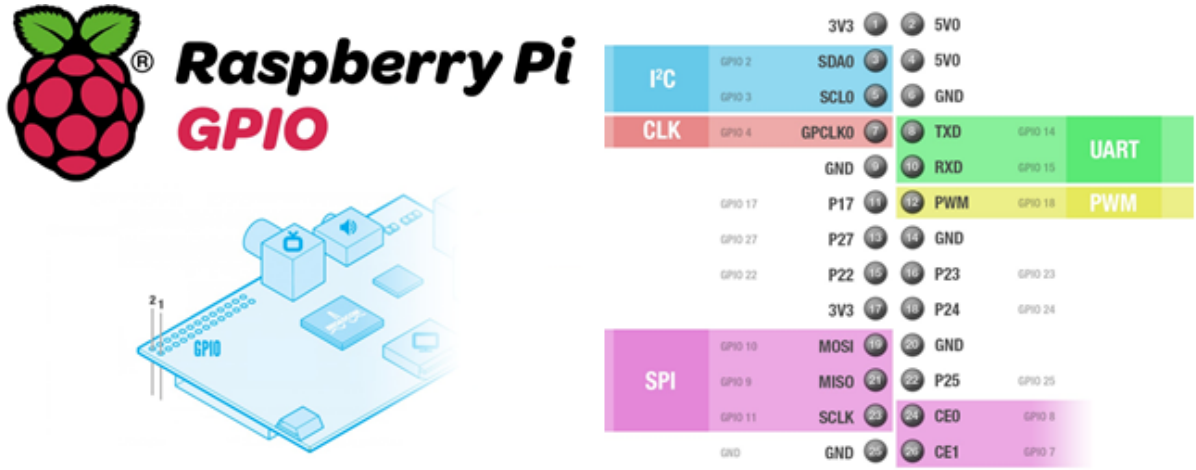


Figura 8- Dispositivos da rede de comunicação. Fonte: RASPBERRY FOUNDATION, 2014.

4.2.2. Concerto F28M36x Experimenter's Kit

- **Microcontrolador Concerto F28M36P63C2**
  Microcontrolador F28M36x *dual core* de alta performance.
- **Emulador JTAG isolado integrado**
  Emulador xds100v2 realiza interface entre *kit* de desenvolvimento e o compilador *Code Composer Studio* (CCS - *software* compilador C/C++ utilizado para programar o microcontrolador).
- **Periféricos e conexões**
  Periféricos e conectores *Ethernet,* cartão microSD, USB e UART/SCI (comunicação serial síncrona e assíncrona) com o microcontrolador F28M36x.
- **Estação *Dockstation***
  Base onde a *Controlcard* (placa que contém o microcontrolador e periféricos) é acoplada. Possui conectores de expansão das GPIOs (Portas analógicas e digitais) e conexão para alimentação de potência do circuito.

4.2.2.1. Microcontrolador Concerto

Concerto é um microcontrolador (MCU) *dual-core system-on-chip* com subsistemas de comunicação e controle em tempo real independentes (TEXAS INSTRUMENTS, 2012). O sistema de comunicação é baseado na CPU 32-bit ARM Cortex-M3 (padrão industrial) e

engloba uma variedade de periféricos de comunicação, incluindo *Ethernet,* USB OTG (USB *On-The-Go,* para conexão entre periféricos) com PHY (camada física da conexão USB), CAN (*controller area network*), UART (*universal asynchronous receiver/transmitter*), SSI (*Synchronous Serial Interface*), I2C (*Inter-Integrated Circuit*), e uma interface externa. O subsistema de controle em tempo real é baseado na CPU de ponto flutuante de 32-bit C28x TMSF28335 incluindo os periféricos de alta precisão para controle, como ePWMs (*Enhanced Pulse-Width Modulator*) com proteção de falhas e *encoders*.

A arquitetura ARM Cortex-M3 é vastamente utilizada na indústria para comunicações e tem um ambiente amplo de ferramentas e *software*, além de ser uma plataforma bem estabelecida para o desenvolvimento de interfaces homem-máquina (IHM) e interfaces gráficas (GUI). Analogamente, o núcleo C28x é a plataforma líder para aplicações de controle na indústria (SHEBI, 2013).

No sistema proposto neste trabalho, o núcleo ARM Cortex-M3 embarca o protocolo DNP3, programado em C e C++ através do *software* de compilação da *Texas Instruments Code Composer Studio 6*, sendo executado sobre um sistema operacional (SO) em tempo real (TI-RTOS). Já o núcleo C28x TMSF28335 executará funções de controle em tempo real, aquisição e tratamento de sinais, e acionamento.

4.2.3. Roteador TPLINK WR841N

Roteador *Wireless* de 300 Mbps, atende aos padrões IEEE 802.11n-g-b, possui duas antenas onidirecionais de 5 dB, frequência de operação de 2,4 à 2,4835 GHz.

4.2.4. Elipse SCADA

Software para desenvolvimento de sistemas supervisórios (SCADA) desenvolvido pela Elipse *Software*. Neste projeto é utilizado para embarcar o protocolo DNP3 no SCADA remoto.

4.3 DEFINICÃO DOS SISTEMAS OPERACIONAIS

4.3.1. Distribuição LINUX RASPBIAN

Raspbian (Figura 9) é uma variante do Debian (uma das mais populares distribuições de Linux) otimizada para o conjunto de instruções ARMVv6 do hardware do computador *Raspberry Pi*. Raspbian é uma palavra composição de *Raspberry Pi* e Debian sendo um

software livre (*open sorce*). Oferece mais de 35.000 pacotes de desenvolvimento de *software* pré-compilados, para serem facilmente instalados no computador *Raspberry Pi*. Tais componentes são especificamente configurados para desempenho otimizado a CPU ARM11 do *Raspberry Pi* .

A distribuição Linux Raspbian contém o ambiente de *desktop* LXDE, o gerenciador de janelas *OpenBox*, o navegador Midori, ferramentas para desenvolvimento de *software* e código fonte de exemplo para funções multimídia. Possui também uma linha de comando, console, onde aplicações podem ser depuradas facilmente.

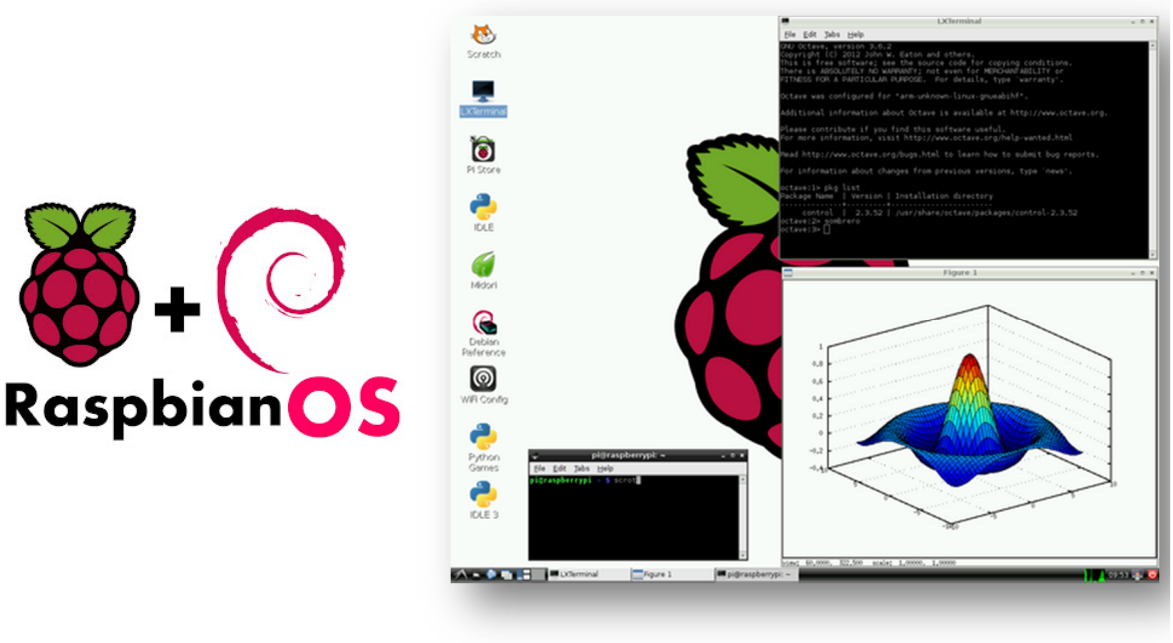


Figura 9- a) Logo Raspbian e b) ambiente de *Desktop* (GUI). Fonte: RASPBERRY FOUNDATION, 2014.

4.3.2. MySQL

O MySQL é um sistema de gerenciamento de banco de dados que utiliza a linguagem SQL (*Structured Query Language*) como interface. É o mais popular dos bancos de dados atuais com mais de 10 milhões de instalações por todo o Mundo.

O MySQL foi criado na Suécia por David Axmark, Allan Larsson e Michael Widenius e lançado por volta de 1995. É compatível com a maioria das lingagens de programação existentes, dentre elas, a mais comum é a PHP. Dentre suas principais características podemos citar:

- Portabilidade: suporta praticamente qualquer plataforma atual;

- Compatibilidade: possui *drivers* e módulos de interface para diversas linguagens de programação como Delphi, Java, C/C++, C#, Visual Basic, Python, Perl, PHP, ASP e Ruby;
- Alto desempenho e estabilidade;
- Pouco exigente quanto a recursos de *hardware*;
- Fácil manuseio;
- *Software* livre e de código aberto;
- Replicação facilmente configurável;
- Interfaces gráficas de fácil utilização.

O MySQL é utilizando neste trabalho para armazenar a variáveis que provém da turbina eólica, na *Raspberry Pi*. Podendo ser acessado a qualquer momento, e desempenha um papel importante no *link* entre os DSPs Concerto e a aplicação SCADA.

Figura 10- MySQL logo. Fonte: www.mysql.com

4.3.3. TI-RTOS

Junto com o kit de desenvolvimento Concerto a *Texas Instruments* disponibiliza um sistema operacional em tempo real fornecido pelo fabricante. TI-RTOS (do inglês *Texas Instruments Real Time Operational System*), combina um *kernel* multitarefa em tempo real com componentes adicionais de *software*, como camadas TCP/IP e USB, sistema de arquivos FAT, e *drivers* de dispositivos (TEXAS INSTRUMENTS, 2014).

Por meio deste sistema operacional é possível executar várias aplicações em um único microcontrolador, graças a sua escalabilidade, fornecendo ao programador a opção de dedicar *threads* (menor sequência de instruções programadas que podem ser gerenciadas

independentemente por um agendador de um SO) para cada tarefa em diferentes níveis de prioridade.

4.3.3.1. Componentes do TI-RTOS

O TI RTOS possui diversos componentes de *drivers* e periféricos de comunicação como ilustrado na Figura 11 abaixo.

Figura 11- Componentes do TI RTOS. Fonte: TEXAS INSTRUMENTS, 2014.

- **SYS/BIOS**

  SYS/BIOS é o *kernel* escalável em tempo real do TI RTOS. É projetado para ser utilizado em aplicações que requerem agendamento em tempo real e sincronia ou instrumentação em tempo real. Oferece *multi-threading* preemptivo, abstração de *hardware,* análise em tempo real, e ferramentas de configuração. O módulo FatFS (*File Allocation Table File System*) é parte do componente SYS/BIOS.

- **IPC**

  Contém pacotes projetados para permitir comunicação entre os processadores, em um sistema de múltiplos processadores, e periféricos de comunicação. Essa comunicação inclui: passagem de mensagens, *stream* e listas vinculadas.

- ***MWARE***

  Inclui drivers de nível baixo e exemplos aplicados ao ARM Cortex-M3.

- **NDK**

  Kit de desenvolvimento de rede. É uma plataforma para o desenvolvimento de aplicações de rede em processadores embarcados da *Texas Instruments*. O NDK provê uma plataforma de rápida prototipagem para o desenvolvimento de aplicações de redes e de processamento de pacotes de dados. Pode ser usado para adicionar conexão de rede em aplicações já existentes para comunicação, configuração e controle. O NDK é uma camada de rede que opera acima da SYS/BIOS do TI-RTOS. Este pacote é designado para ser um *add-on* transparente ao SYS/BIOS e às ferramentas do CCS.

- **UIA**

  Arquitetura unificada de instrumentação. Fornece conteúdo para aplicações, como criação e coleta de dados de instrumentação.

- ***XDCtools***

  Fornece ferramentas para configuração e construção do SYS/BIOS, IPC, NDK e UIA.

4.3.3.2. *Inter-Processor Communication* (IPC)

IPC é um componente que proporciona comunicação entre processadores em um ambiente com múltiplos processadores e periféricos de comunicação. Esta comunicação inclui pasagem de mensagens, *streams*, e listas ligadas.

4.3.3.2.1. Módulo *MessageQ*

O módulo *MessageQ* suporta estrutura de envio e recebimento de mensagens de tamanho variável. É independente do sistema operacional e para cada *MessageQ* criada, existe um único leitor e podem existir vários escritores.

O módulo *MessageQ* é a aplicação recomendada na maioria das aplicações com o DSP Concerto. Pode ser utilizado em ambas comunicações multi-processadores homogênea e

heterogênea, e com um único processador comunicando entre *threads.* A Figura 12 a seguir mostra um fluxograma do funcionamento da *MessageQ.*

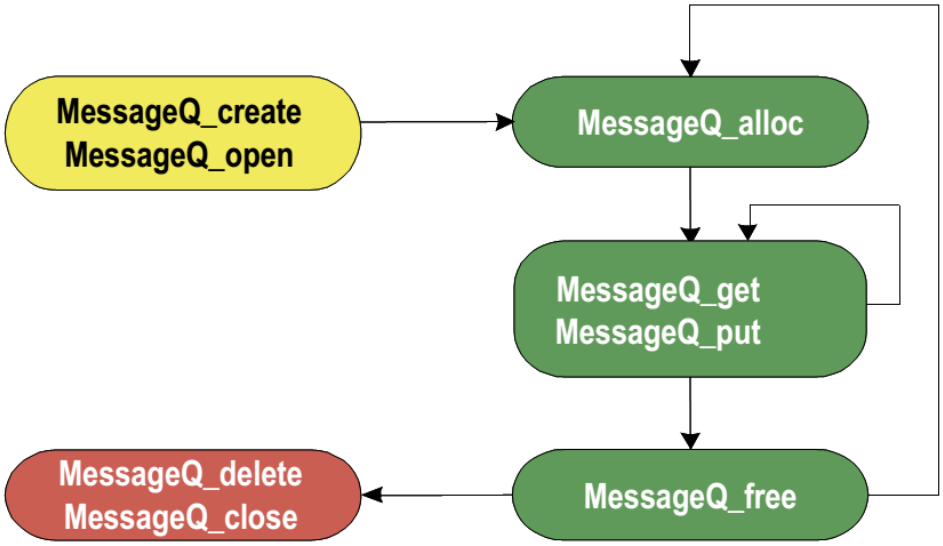


Figura 12- Módulo *MessageQ*-IPC, fluxograma. Fonte: TEXAS INSTRUMENTS, 2014.

4.3.3.3. DRIVERS

Drivers para os periféricos contidos na Tabela 2 são fornecidos com o sistema operacional TI-RTOS. Periféricos estes, que compõem o *kit* de desenvolvimento TMDXDOCK28M36 utilizado neste trabalho.

Os *Drivers* fornecidos pelo TI-RTOS podem ser configurados através de uma interface GUI (interface gráfica de usuário), Figura 13.

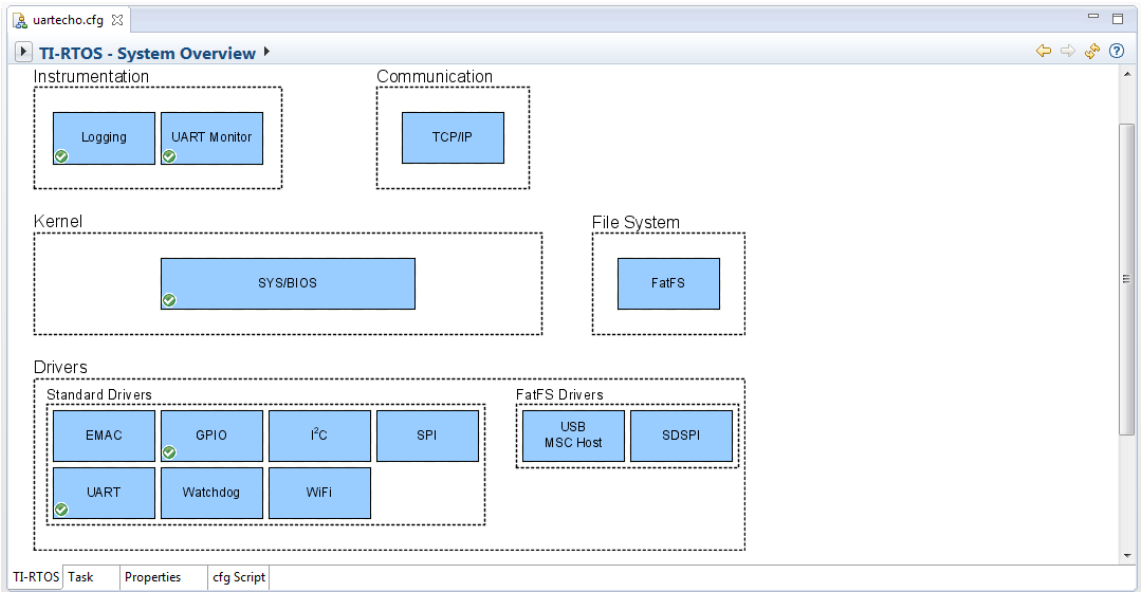


Figura 13– Interface de usuário do TI-RTOS no CCS 6.

Tabela 2- *Drivers* fornecidos pelo TI-RTOS

Driver	**Função**
EMAC	*Driver* de *Ethernet* usado pelo protocolo de rede.
WiFi	*Driver* para periférico de rede sem fio
SDSPI	*Driver* para cartão SD, no barramento de SSI, usado pelo sistema FatFS.
I^2C	API para ser utilizada diretamente pela aplicação para comunicação I^2C
SPI	API para ser utilizada diretamente pela aplicação para comunicação SPI
GPIO	API para ser utilizada diretamente pela aplicação para gerenciar interrupções de GPIO, pinos e portas.
UART	API para ser utilizada diretamente pela aplicação para comunicação UART
USBMSCHFatFS	*Driver* para conexão de pen drives (USB sobre FatFS)
Watchdog	API para ser utilizada diretamente pela aplicação para gerenciar temporização de *watchdog*

4.4 PROJETO DA REDE COMUNICAÇÃO

4.4.1. Topologia do Sistema

A comunicação entre a turbina eólica e um sistema SCADA remoto é realizada por meio do protocolo DNP3 sobre TCP/IP, já a comunicação interna do aerogerador (entre os dois *kits Concerto* e a *Raspberry Pi*) é realizada pelo protocolo TCP/IP. A topologia de cada rede de comunicação é melhor detalhada nos itens a seguir.

4.4.1.1. Comunicação com Protocolo DNP3

O protocolo de DNP3 suporta as seguintes topologias de rede:

- Mestre-escravo (*ponto à ponto*).
- *Multidrop* de uma estação mestre.
- Hierárquica, com concentradores de dados intermediários.
- Múltiplos mestres.

A topologia utilizada para comunicar o sistema SCADA remoto com a *Raspberry Pi* é a de configuração mestre-escravo (P2P), Figura 14, onde o sistema SCADA configura-se como mestre da *Raspberry Pi,* e esta configura-se como escravo do sistema SCADA.

4.4.1.2. Comunicação com Protocolo TCP/IP

A comunicação entre o computador *Raspberry Pi* e os *kits Concerto* (comunicação interna da turbina eólica) é de configuração Meste-Escravo com múltiplos escravos, neste caso a topologia de conexão física se configura como barramento. A Figura 14 ilustra a rede de comunicação TCP/IP.

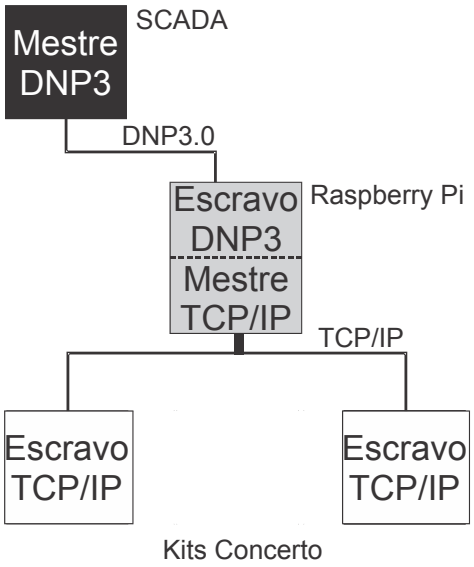


Figura 14- Diagrama da rede de comunicação

4.4.2. Arquitetura do Protocolo de Comunicação

O protocolo DNP3 segue o modelo de arquitetura EPA (*enhanced performance architecture*), que é uma divisão do modelo OSI de 7 camadas em um modelo de 3 camadas. As camadas utilizadas por esse modelo são as duas camadas de *hardware* e a camada mais alta de *software*, a camada de aplicação (CLARKE, 2004). A Figura 15 ilustra o modelo EPA.

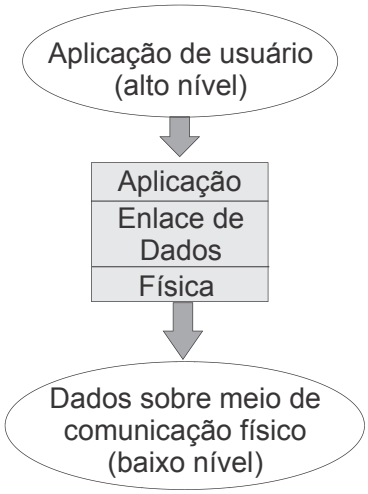


Figura 15- Modelo *Enhanced Performance Architecture* (EPA). Fonte: CLARKE, 2004.

O protocolo DNP3 utiliza as três camadas do modelo EPA, adicionando ainda, algumas funções de transporte. Tais funções são chamadas de camadas de pseudo-transporte, correspondendo às camadas de transporte e de rede com alugas limitações. Tal relação é mostrada na Figura 16, que faz um comparativo entre o modelo EPA, implementado no DNP3, com o modelo de referência OSI.

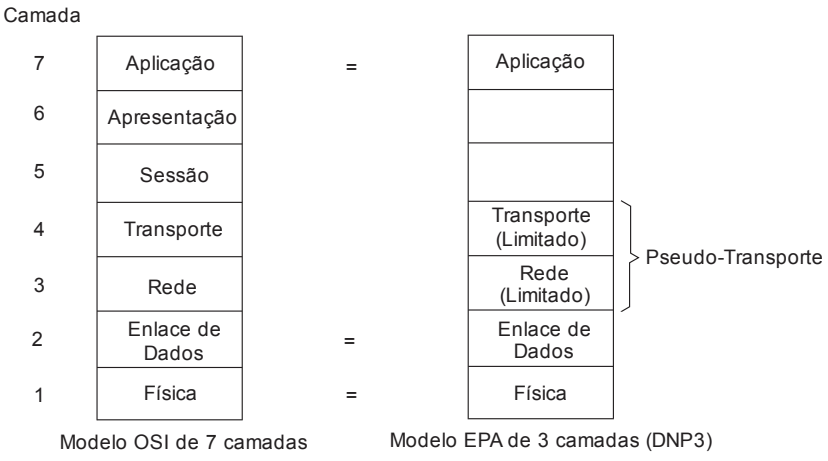


Figura 16- Comparação entre modelo EPA e OSI. Fonte: CLARKE, 2004.

4.4.2.1. Implementando DNP3 sobre uma Rede (TCP/IP)

A ideia de implementar DNP3 sobre um ambiente de rede envolve a encapsulação dos *frames* de dados da camada de enlace de dados DNP3 dentro dos *frames* da camada de transporte do protocolo Internet e permitir que este protocolo entregue *frames* da camada de enlace de dados do DNP3 para o endereço de destino, no lugar da camada física original do DNP3. A arquitetura de comunicação resultante é apresentada no diagrama da Figura 17.

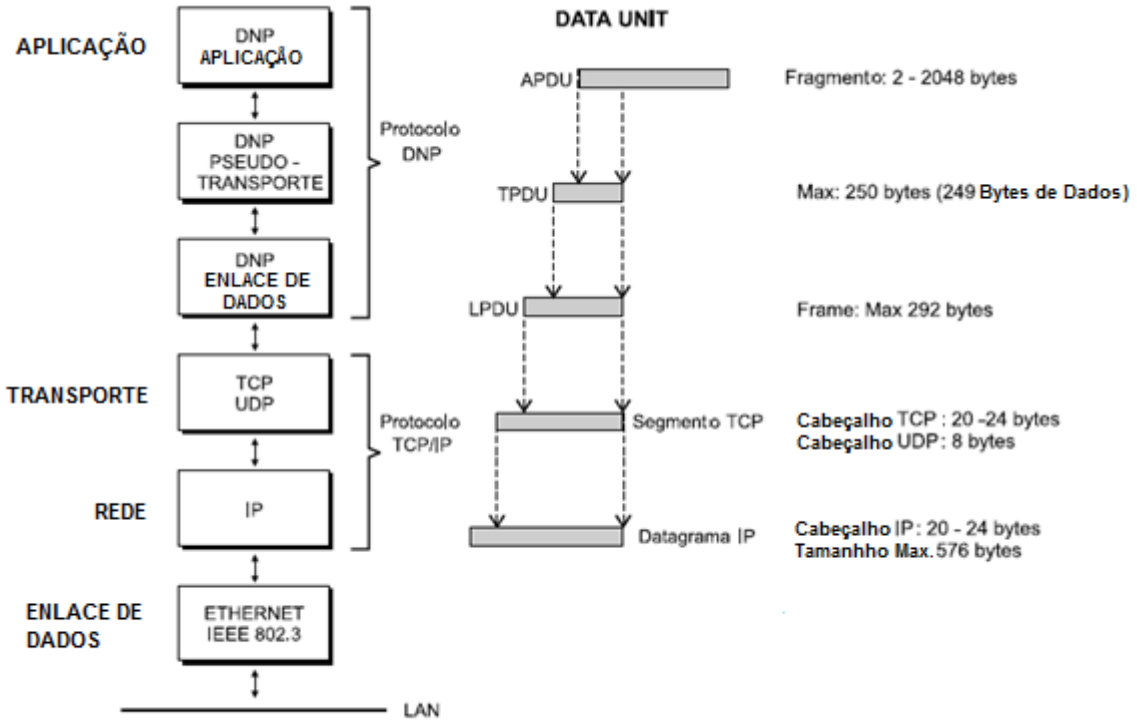


Figura 17- DNP3 sobre protocolo TCP/IP. Fonte: CLARKE, 2004.

4.4.3. Funções das Camadas do Protocolo

4.4.3.1. Camada Física

A camada física é o meio físico por onde o protocolo será transmitido, o bit é o elemento de dado neste nível. O protocolo DNP3 especifica alguns meios físicos como o EIA RS-232C para níveis de tensão e sinais de controle, e a CCITT V.24 para sinais DTE-DCE. Outros meios de comunicação, como o sobre *Ethernet*, estão em processo de definição pelo Comitê Técnico da DNP3 *User Group.*

Neste trabalho, o meio físico de comunicação utilizado será através da Ethernet, com cabeamento padrão IEEE 802.3 100BaseT. Assim, o protocolo resultante incorpora as funções do protocolo TCP/IP.

4.4.3.2. Camada de Enlace de Dados

A camada de enlace de dados provê uma transmissão confiável dos dados através do meio físico. Enquanto a camada física se preocupa com a passagem do sinal, ou o bit de dados, a camada de enlace de dados se preocupa com a transmissão dos grupos de dados, tais grupos

são conhecidos como quadros. No DNP3, funções fornecidas por tal camada incluem, controle de fluxo e detecção de erro.

4.4.3.3. Camada Pseudo-Transporte

Possibilita a transmissão de blocos maiores de dados. As funções de rede se preocupam com o roteamento e controle do fluxo dos pacotes de dados sobre a rede. Enquanto as funções de transporte fornecem transparência de rede para a entrega das mensagens de dispositivo a dispositivo, incluindo construção e desconstrução de mensagens e detecção de erro.

4.4.3.4. Camada de Aplicação

Nível onde os dados são gerados para envio ou para requisição de envio. Tal camada realiza interface com os níveis mais baixos para que a transmissão entre dispositivos seja alcançada. A camada de aplicação do DNP3 fornece serviços para os programas de aplicação do usuário, como em um sistema IHM, uma unidade terminal remota (RTU), ou outro sistema.

4.4.4. Construção da Mensagem DNP3

A Figura 18 mostra como a mensagem do protocolo DNP3 é transmitida. Cada camada do modelo obtém a informação passada de uma camada mais elevada, e adiciona informações a respeito dos serviços nesta camada. Tal informação é adicionada como um cabeçalho, isto é, na frente da mensagem original.

Assim, durante a construção da mensagem do protocolo, a mesma irá crescer em tamanho a cada camada que passa. Neste processo a mensagem também pode ser fragmentada em unidades menores de dados.

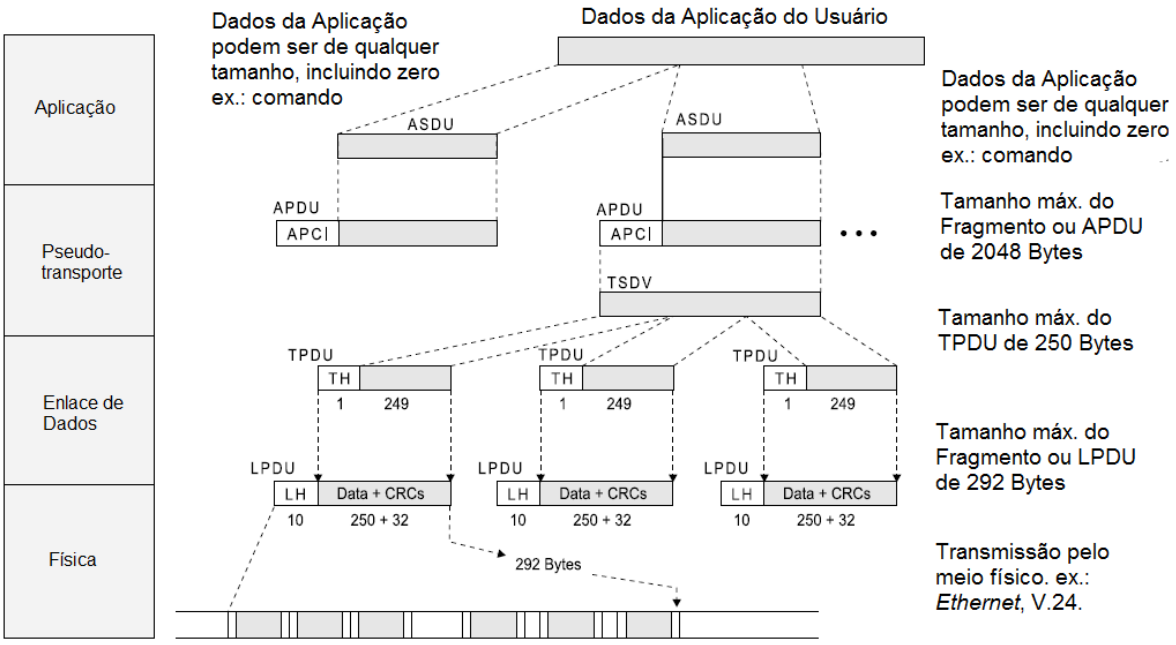


Figura 18- Construção da mensagem do DNP3. Fonte: CLARKE, 2004.

Durante a construção da mensagem o fluxo de dados vai da camada de aplicação até a camada física, então através do meio físico até chegar ao destinatário onde a mensagem original é reconstruída.

A construção da mensagem ilustrada na Figura 18 é brevemente descrita para cada camada nos itens a seguir (da camada mais alta, camada de aplicação, até a camada física). Esta sequência é adotada durante o envio de uma mensagem, a sequência na recepção é feita pelo caminho inverso.

4.4.4.1. Camada de Aplicação

Os dados do usuário correspondem aos dados provenientes da aplicação do usuário. Esta pode ser visualizada como uma camada acima da camada de aplicação, e pode ser, por exemplo, uma interface homem máquina (IHM) ou um programa em C++. Os protocolo não limita o tamanho total do pacote de dados, portanto, este pode ser de qualquer tamanho.

Primeiramente, a camada de aplicação constrói os dados em blocos de tamanhos gerenciáveis. Estes chamados de unidade de dados do serviço da aplicação (ASDUs). A camada de aplicação cria então a unidade de dados da aplicação do protocolo (APDU), combinando um cabeçalho com os dados do ASDU. O cabeçalho da camada de aplicação é chamado de informação de controle da aplicação do protocolo, ou APCI, sendo 2 ou 4 bytes em comprimento, dependendo se a mensagem é uma requisição ou uma resposta. No caso de um

comando ou requisição de usuário que não requisite dados adicionais, haverá somente o cabeçalho na mensagem.

Dependendo do tamanho total do dado a ser transmitido, ou seja, o tamanho do ASDU, um ou mais APDUs podem ser criados. Embora o número de fragmentos que representam um ASDU seja ilimitado, o tamanho de cada fragmento é limitado pelo tamanho máximo de 2048 bytes.

4.4.4.2. Camada de Pseudo-Transporte

O bloco APDU da camada de aplicação pode ser definido como a unidade de dados de serviços de transportes (TSDU) contido na camada de pseudo-transporte. É interpretado puramente como o dado a ser transportado pela camada de transporte. A camada de transporte quebra o TSDU em unidades menores chamado de unidade de dados de transporte do protocolo (TSDUs), compostos de um cabeçalho de um byte e seguido de, no máximo, 249 bytes de dados. O tamanho total do TPDU, 250 bytes, é determinado de maneira que cada TPDU caiba dentro de um *frame* ou LPDU na camada de enlace de dados.

4.4.4.3. Camada de Enlace de Dados

Esta camada pega os TPDUs da camada de pseudo-transporte e os adiciona (à cada TPDU) um cabeçalho de 10 bytes. Como a camada de enlace de dados é responsável por fornecer detecção de erro e funções de correção, a checagem de erro é introduzida nesta camada com um código cíclico de redundância (CRC) de 16 bits. Cada TPDU é convertido para um frame de até 292 bytes de comprimento.

4.4.4.4. Camada Física

A camada física converte cada *frame* em um *stream* de bits sobre o meio físico. Na documentação original do protocolo DNP3, a transmissão assíncrona *serial* (RS-232C) é especificada. Especificando pacote de dados de 8 bits: 1 bit de início, 1 bit de parada, sem paridade e níveis de tensão e sinais de controle conforme o padrão RS-232C.

4.4.5. Funções de Mensagens da Camada de Aplicação

Como a rede se comunica com um sistema SCADA, espera-se que as mensagens sejam capazes de carregar comandos e outros tipos de dados, que serão gerados em resposta aos comandos. Isto é a função de mais alto nível oferecida pelo sistema SCADA, controle e aquisição de dados. As mensagens, então, contêm ambos os códigos de função e objetos de dados. Códigos de função definem o significado e o propósito da mensagem. Objetos de dados definem a estrutura e interpretação dos dados.

O propósito disto é não somente fornecer comunicação entre dispositivos de diferentes fabricantes, mas também de tornar os dados entendíveis entre diferentes plataformas. Nesta seção será examinada tal padronização de mensagens. Começando pelos códigos de função e seguindo então aos objetos de dados definidos pelo protocolo DNP3.

4.4.5.1. Códigos de Função

O código de função é responsável por identificar o propósito da mensagem DNP3. Existem dois grupos de funções, para requisições e respostas. As principais requisições são ilustradas da Tabela 3 abaixo.

Tabela 3- Códigos de função DNP3

Código	Função	Descrição
1	**Ler**	Solicita os objetos especificados da estação remota e responde com os objetos pedidos disponíveis.
2	**Escrever**	Armazena os objetos especificados na estação remota; responde com o *status* da operação.
3	**Selecionar**	Seleciona ou arma pontos de saída, mas não seta ou produz qualquer ação (controles, *setpoints* ou saídas analógicas); responde com o *status* da operação. A função Operar deve ser usada para ativar estas saídas.
4	**Operar**	Seta ou produz ações nas saídas ou pontos previamente selecionados com a função Selecionar
5	**Operar Diretamente**	Seleciona e Opera as saídas especificadas; responde com o *status* dos pontos de controle.
6	**Operar Diretamente sem ACK**	Seleciona e Opera as saídas especificadas, mas não envia resposta.

4.4.5.2. Dados de Objetos

Os dados produzidos pelos sistemas inteligentes na camada de aplicação DNP3 são processados e armazenados como objetos de informação padronizados. Existem quatro tipos de categorias de objetos de dados:

- **Objetos Estáticos (*Static Objects*)**
  São os objetos que refletem o valor atual de uma variável de campo ou interna.
- **Objetos de Evento (*Event Objects*)**
  São os objetos que são gerados como o resultado de uma mudança de valor ou outro estimulante. São objetos históricos, ou seja, refletem o valor de um dado em algum instante no passado.
- **Objetos Estáticos Congelados (*Frozen Static Objects*)**
  Refletem o valor "congelado" atual de uma variável de campo ou interna. Dados são "congelados" como o resultado de um pedido de "congelamento" de dados.
- **Objetos de Evento Congelados (*Frozen Event Objects*)**
  São os objetos resultantes da mudança de um valor congelado.

Cada categoria listada acima é representada com um objeto diferente, conforme Tabela 4.

Tabela 4- Grupos de objetos DNP3

Objeto	**Descrição**
Entradas Digitais	Contém todos os objetos que representam entradas binárias (*status* ou atributos booleanos). Vão de 1 a 9.
Saídas Digitais	O grupo de saídas digitais contém todos os objetos que representam saídas binárias ou informação de controle de relés. Vão de 10 a 19.
Contadores	Este grupo contém todos os objetos contadores. 20 a 29.
Entradas Analógicas	Contém todas as entradas analógicas, e vão de 30 a 39.
Saídas Analógicas	Contém todas as saídas analógicas, e vão de 40 a 49.
Tempo	Contém todos os objetos que representam tempo em forma absoluta ou relativa, e vão de 50 a 59.
Classes	Este grupo contém todos os objetos que representam classes de dados ou prioridade de dados. De 60 a 69.
Arquivos	Arquivos ou sistema de arquivos. De 70 a 79.
Dispositivos	De 80 a 89.
Aplicações	Objetos que representam aplicações de *software* ou processos do sistema operacional. De 90 a 99.
Objetos numéricos alternativos	Representações numéricas customizadas. De 100 a 109.

4.4.5.3. Variação

São modificações ou subtipos que podem ocorrer nos objetos. Uma entrada digital pode ser representada apenas por um único bit (0 ou 1), por uma palavra de *status* (um byte) ou ainda conter ou não a informação de tempo (*timestamp*). Sendo assim, a combinação do objeto mais a variação descrevem completamente uma informação.

4.5 O *OpenDNP3*

O *OpenDNP3* é uma implementação aberta, portátil, escalável e testada rigorosamente do protocolo DNP3, escrita em C++ (AUTOMATAK, 2014). Sua biblioteca é otimizada para sistemas embarcados, estando em desenvolvimento desde 2006. Seu código fonte é totalmente aberto. Esta será a biblioteca utilizada neste trabalho, para implementar o protocolo DNP3 no computador *Raspberry Pi*, por meio do sistema operacional *Raspbian.*

5. RESULTADOS EXPERIMENTAIS

Para validar os testes do protocolo DNP3 utilizando a biblioteca openDNP3 foi utilizado o seguinte *setup*, conforme Figura 19. Consistindo de:

- Computador contendo aplicação SCADA;
- 2 Kits DSP Texas Instruments F28M36x Concerto para aquisição de dados e controle da turbina eólica.
- *Raspberry Pi,* operando como um *gateway* entre os DSPs Concerto e o SCADA, encapsulando os dados do aerogerador no protocolo DNP3, para atender a IEC 61400-25
- Roteador Wireless TP-LINK WR841N com duas antenas de alto alcance (270 metros).
- IHM local para *debug* e visualização do código do DSP e *Raspberry Pi.*

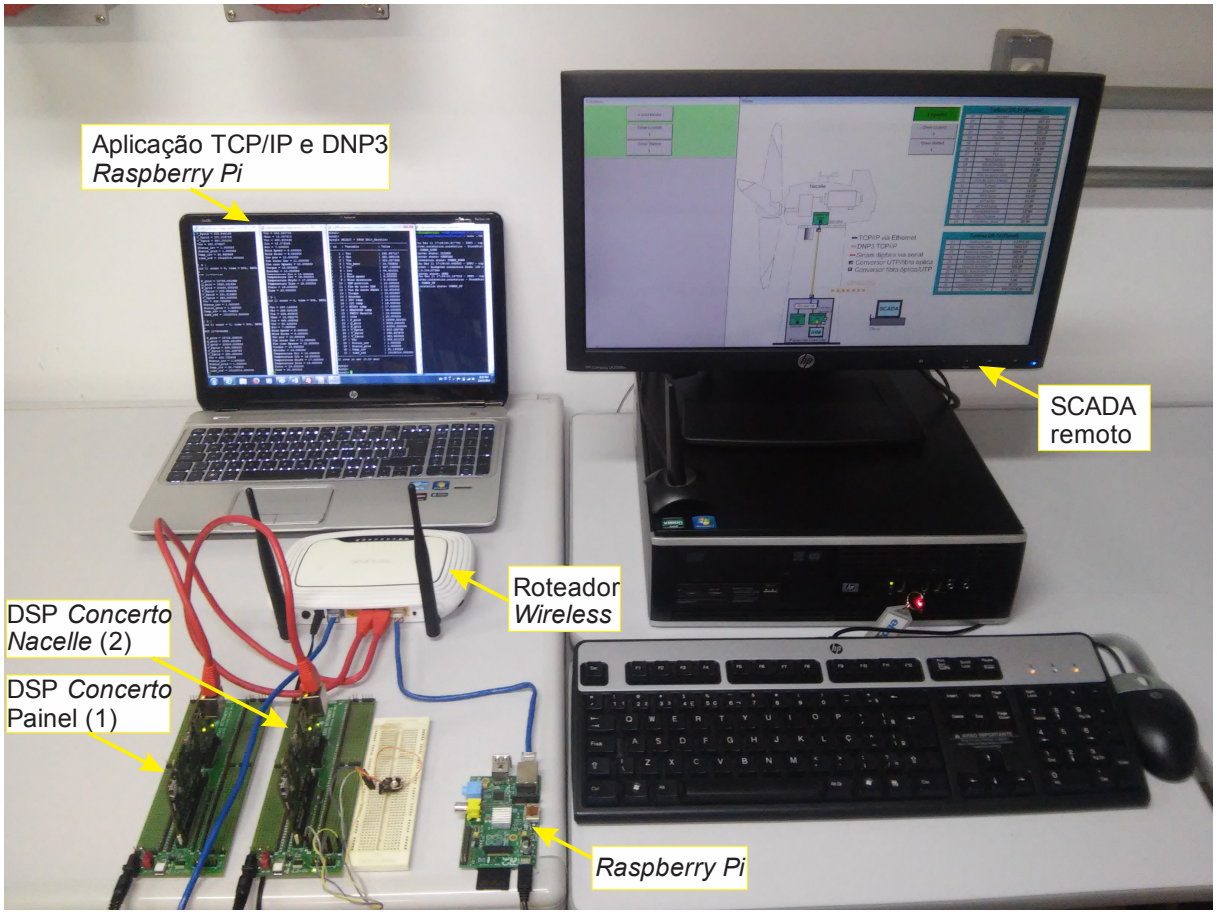


Figura 19- *Setup* experimental para teste da rede de comunicação

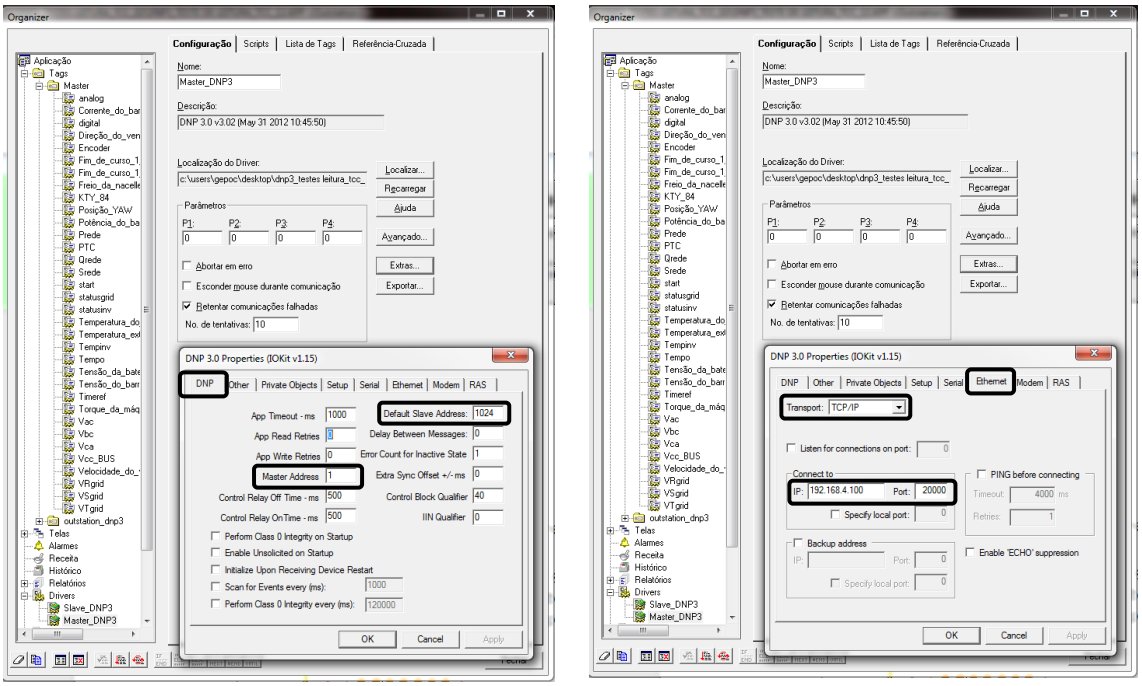
Primeiro, foi desenvolvida uma aplicação no *software* Elipse SCADA, de maneira a criar uma estação Mestre para efetuar a leitura e atualização de todas as variáveis localizadas

no aerogerador, neste caso a *Raspberry Pi* (*Outstation*). Para isso foi necessário parametrizar o *driver* DNP3 no Elipse SCADA, configurar as variáveis e criar o aplicativo supervisório, para que a estação Cliente pudesse se comunicar com o Servidor.

5.1 CONFIGURAÇÃO MESTRE SCADA

O protocolo DNP3, assim como muitos protocolos industriais, padroniza os tipos de mensagens e variáveis trocados na rede, por isso as variáveis trocadas devem ser devidamente parametrizadas, isto é, escolhendo-se corretamente o código de função e o objeto do dado, tanto no mestre como na *outstation*. Seguindo o manual do *driver* DNP3 fornecido pelo fabricante do *software* Elipse SCADA (ELIPSE, 2014):

1. O primeiro passo foi configurar o *driver* DNP3 mestre para o SCADA. A Figura 20.a) ilustra a tela de propriedades do DNP, o endereço da estação mestre (*Master Address*) é configurado para ser 1 enquanto o endereço padrão do escravo é 1024.
2. Feito isso, na mesma janela anterior, mas na aba *Ethernet* (Figura 20.b)), é configurada a conexão do mestre com a *outstation*, onde deve ser inserido o endereço de IP e a porta da mesma. Neste caso o escravo possui endereço IP 192.168.4.100 e porta 20000.

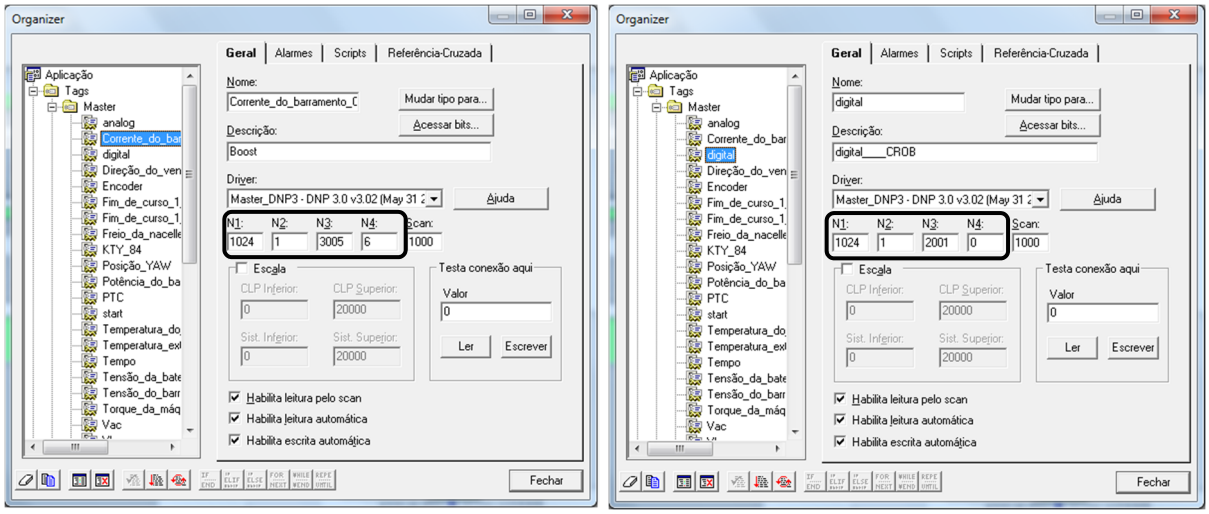


a) Propriedades DNP, endereços b) Configuração conexão *Ethernet*

Figura 20- Configuração propriedades DNP3

3. O segundo passo foi criar e parametrizar as variáveis no sistema SCADA. As variáveis configuram-se por *tags* no Elipse SCADA, que devem ser criadas dentro do grupo de

variáveis do mestre. Neste caso foram criadas 31 variáveis ou *tags* do Elipse SCADA, variáveis originárias da *nacelle* e painel de controle da turbina eólica, dentre elas 26 são do tipo ponto flutuante entrada analógica de 32 bits, Figura 21.a), e outras 5 variáveis do tipo contador de 32 bits, Figura 21.b). Conforme fabricante no *software* Elipse SCADA, para cada *tag*, é necessário definir os valores de N1 (Endereço do escravo), N2 (Código de função a realizar), N3 (Código do objeto e variação) e N4 (Endereço da variável, índice), que possuem e definem todas as informações pertinentes à variável declarada. A Tabela 5 mostra os valores de N1, N2, N3 e N4, parametrizadas para ambos tipos de variáveis utilizadas.



a) Parametrização variável *analog*

b) Parametrização variável *digital*

Figura 21- Configuração de variáveis (*tags*)

Tabela 5- Parametrização das variáveis

Variáveis *ponto flutuante* (26)		**Variáveis** *booleanas* (5)	
Item	**Descrição**	**Item**	**Descrição**
N1: 1024	Endereço da *outstation*	N1: 1024	Endereço da *outstation*
N2: 1	Código de função: Leitura	N2: 1	Código de função: Leitura
N3: 30|05	Código de objeto e variação: variável de 32 bits do tipo Entrada Analógica Ponto Flutuante	N3: 20|01	Código de objeto e variação: variável de 32 bits do tipo Contador
N4: 1-17	Endereço da variável na *outstation*	N4:18-20	Endereço da variável na *outstation*

As variáveis lidas pela aplicação SCADA podem ser conferidas em maior detalhe na Tabela 6 abaixo.

Tabela 6- Variáveis da Nacelle do aerogerador

ID	Variável	Descrição	Unidade	Tipo
01	Vac	Tensão eficaz de fase do gerador (CA)	Volts (V)	Ponto flutuante, entrada analógica 32-bits
02	Vbc	Tensão eficaz de fase do gerador (CA)	Volts (V)	Ponto flutuante, entrada analógica 32-bits
03	Vbc	Tensão eficaz de fase do gerador (CA)	Volts (V)	Ponto flutuante, entrada analógica 32-bits
04	Vcc bat	Tensão da bateria (CC)	Volts (V)	Ponto flutuante, entrada analógica 32-bits
05	Vcc	Tensão do barramento (CC)	Volts (V)	Ponto flutuante, entrada analógica 32-bits
06	Icc	Corrente do barramento (CC)	Amperes (A)	Ponto flutuante, entrada analógica 32-bits
07	Pcc	Potência gerada (CC)	Watts (W)	Ponto flutuante, entrada analógica 32-bits
08	Wind Speed	Velocidade do vento	Metros por segundo (m/s)	Ponto flutuante, entrada analógica 32-bits
09	Wind Direction	Direção do vento	Graus (°)	Ponto flutuante, entrada analógica 32-bits
10	YAW position	Ângulo YAW com relação ao Norte	Graus (°)	Ponto flutuante, entrada analógica 32-bits
11	Fim curso YAW	Estado do fim de curso do YAW	0 ou 1	Contador 32-bits
12	Fim curso Passo	Estado do fim de curso do atuador de passo	0 ou 1	Contador 32-bits
13	Torque	Torque no eixo do gerador	Quilograma-força metro ($kgfm$)	Ponto flutuante, entrada analógica 32-bits

14	Encoder	Velocidade no eixo do gerador	Rotações por minuto (rpm)	Ponto flutuante, entrada analógica 32-bits
15	PTC temp	Temperatura do Gerador (não linear)	Graus Celsius ($°C$)	Ponto flutuante, entrada analógica 32-bits
16	I2C temp	Temperatura externa da Nacelle (via barramento I2C)	Graus Celsius ($°C$)	Ponto flutuante, entrada analógica 32-bits
17	KTY-84 temp	Temperatura do Gerador (linear)	Graus Celsius ($°C$)	Ponto flutuante, entrada analógica 32-bits
18	Heatsink temp	Temperatura no dissipador dos semicondutores do conversor *boost*	Graus Celsius ($°C$)	Ponto flutuante, entrada analógica 32-bits
19	Freio Nacelle	Estado do freio da Nacelle	0 ou 1	Ponto flutuante, entrada analógica 32-bits
19	Timestamp	Estampa de tempo		Tempo e data

Tabela 7- Variáveis do Painel do Controle do aerogerador

ID	Variável	Descrição	Unidade	Tipo
21	P_grid	Potência ativa	Watts (W)	Ponto flutuante, entrada analógica 32-bits
22	Q_grid	Potência reativa	Volt-Ampère reativo (VAr)	Ponto flutuante, entrada analógica 32-bits
23	S_grid	Potência aparente	Volt-Ampère (VA)	Ponto flutuante, entrada analógica 32-bits
24	V_Rgrid	Tensão de linha R (CA)	Volts (V)	Ponto flutuante, entrada analógica 32-bits
25	V_Sgrid	Tensão de linha T (CA)	Volts (V)	Ponto flutuante, entrada analógica 32-bits
26	V_Tgrid	Tensão de linha T (CA)	Volts (V)	Ponto flutuante, entrada analógica 32-bits

27	Vcc	Tensão do barramento (CC)	Volts (V)	Ponto flutuante, entrada analógica 32-bits
28	Satatus_inv	Estado de operação do inversor	0 ou 1	Contador 32-bits
29	Status_grid	Estado de operação do inversor	0 ou 1	Contador 32-bits
30	Temp_inv	Temperatura do inversor	Graus Celsius ($°C$)	Ponto flutuante, entrada analógica 32-bits
31	time_ref	Estampa de tempo		Ponto flutuante, entrada analógica 32-bits

A aplicação do supervisório implementada no *software* Elipse SCADA é apresentada na Figura 22.

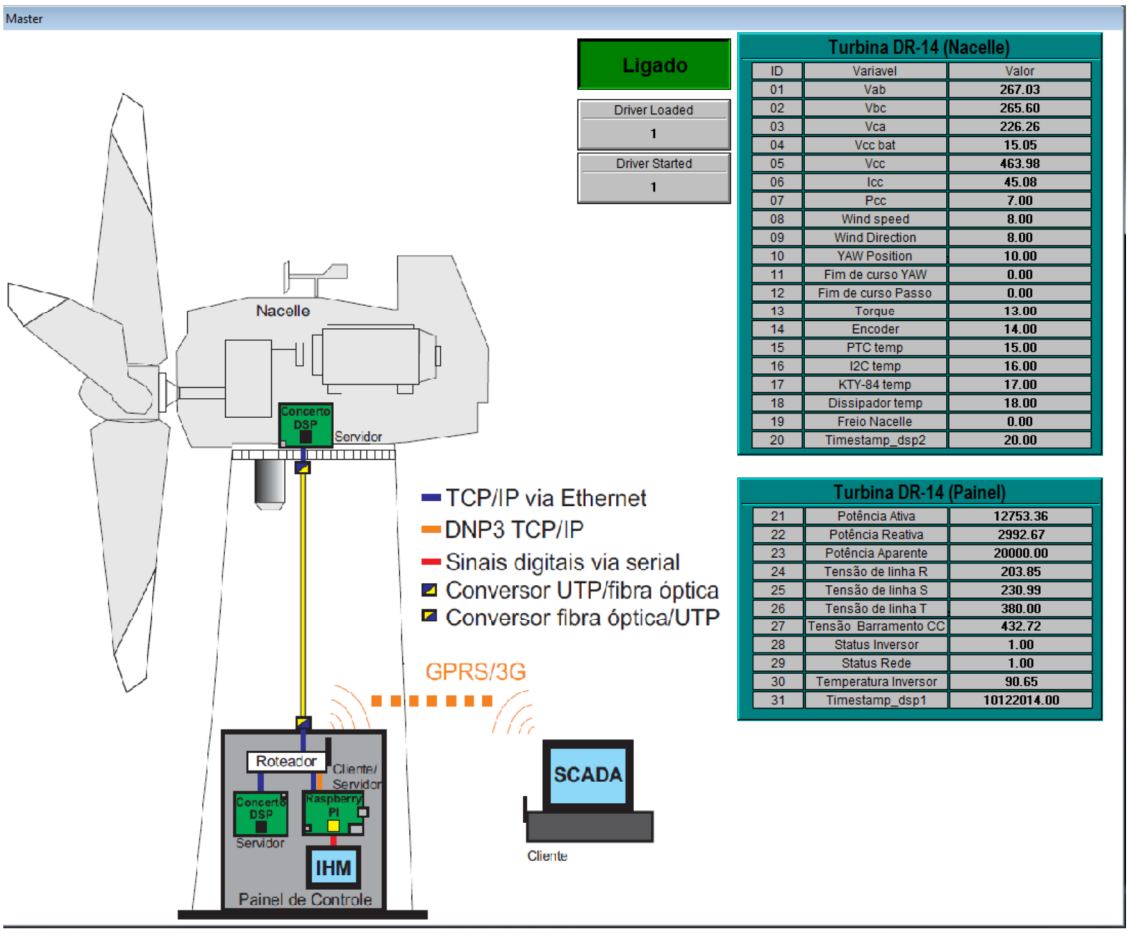


Figura 22- Aplicação SCADA

Com esta configuração o mestre SCADA fará a leitura de todas as variáveis no dispositivo escravo e mostrará o valor das variáveis na tela do supervisório. O próximo passo então, é configurar o dispositivo escravo (*outstation*) da rede DNP3, o computador *Raspberry Pi.*

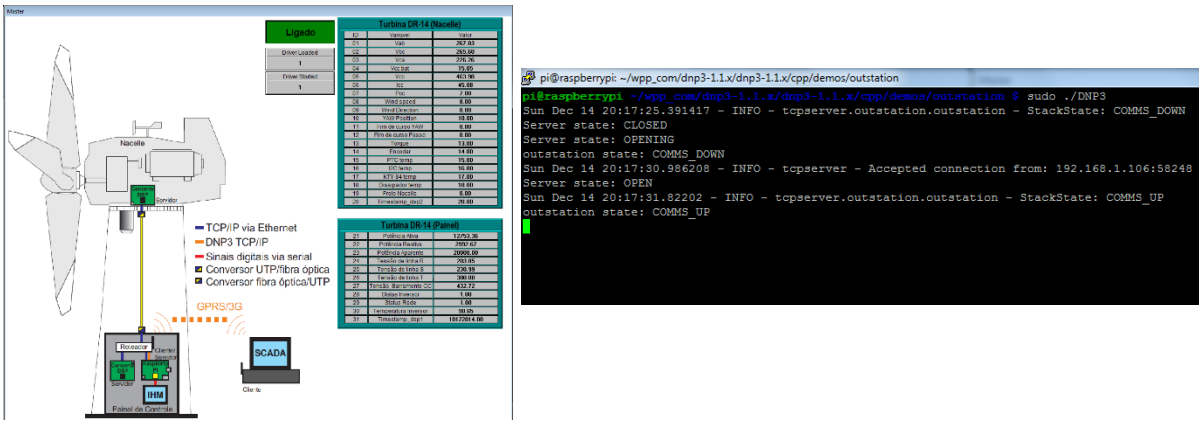
5.2 CONFIGURAÇÃO *RASPBERRY PI*

A *Raspberry Pi* opera como um *gateway* (ponte de ligação) entre o DSP Concerto F28M36x e a aplicação SCADA, encapsulando os dados de acordo com o padrão do protocolo DNP3. Ou seja, a *Raspberry Pi* realiza leitura das variáveis do DSP, gerencia a comunicação com o mesmo, encapsula tais variáveis no procolo DNP3 e as disponibiliza para a aplicação SCADA, pela rede *Ethernet*. É, por assim dizer, um concentrador local de informações.

5.2.1. Aplicação DNP3

Para configuração da *Raspberry Pi* como escravo DNP3, foi utilizado a biblioteca *open source,* openDNP3, desenvolvida pela empresa AUTOMATAK. A biblioteca, escrita em C++, possui todos os arquivos e funções para implementação do protocolo em sistemas embarcados LINUX.

1. O primeiro passo foi instalar o sistema operacional baseado em LINUX no computador *Raspberry Pi*, o sistema *Raspbian* (Debian) foi instalado, por se tratar de um SO confiável e extremamente utilizado para este *hardware* (arquiteturas ARM).
2. Em seguida foi necessário efetuar a compilação da biblioteca openDNP3 na *Raspberry Pi*, via linha de comando no ambiente LINUX.
3. Feito isso, uma aplicação para escravo (*outstation*) foi desenvolvida, com base na biblioteca fornecida, esta se encontra na íntegra no Anexo A. Tal aplicação consiste dos seguintes eventos: são configurados a conexão do escravo com o mestre; as variáveis da Tabela 6 são lidas de um banco de dados localizado na própria *Raspberry Pi* e são encapsuladas no protocolo DNP3; a cada intervalo de tempo definido pelo SCADA (*scan* em *ms*) o mestre efetuará a atualização e leitura das variáveis em questão. Em seguida, os valores atualizados são mostrados na aplicação SCADA, conforme Figura 23.



a) Tela do PC, SCADA executando aplicação mestre DNP3

b) Tela *Raspberry Pi* executando aplicação escravo DNP3

Figura 23- Conexão Mestre/Escravo e troca de dados

Um fluxograma simplicado do código da aplicação DNP3 na *Raspberry Pi*, Anexo A, contendo as principais funções da *outstation* descritas acima pode ser visto na Figura 24.

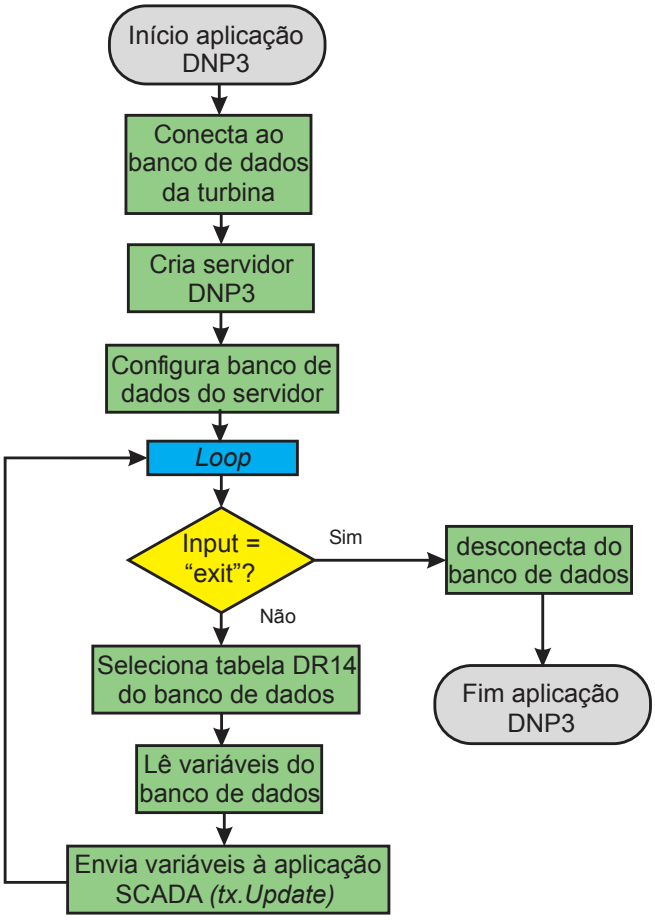


Figura 24- Fluxograma do código de aplicação DNP3 escravo na *Raspberry Pi*

Conforme é possível visualizar nas Figuras 23 e 24, o escravo espera pela conexão do mestre, assim que a conexão é realizada o mestre realiza a atualização e varredura das variáveis

gravadas, por meio da operação ler e comando *tx.Update* (Anexo A, vide Table 3), que são mostradas na tela do supervisório.

5.2.2. Aplicação TCP/IP

A comunicação entre *Raspberry Pi* e os DSPs *Concerto* da *Texas Instruments* é efetuada por meio do protocolo TCP/IP sobre rede *Ethernet.* O protocolo DNP3 não foi utilizado para efetuar a comunicação interna no aerogerador pelo fato de necessitar de um *hardware* mais poderoso, com pelo menos mais de 2 megabytes de memória RAM, e os *kits Concerto* possuirem somente 232 kilobytes, dessa forma, optou-se pelo uso do protocolo TCP/IP, que é um protocolo eficiente, robusto e seguro, sendo ainda nativo aos *kits DSP Concerto* (O TI-RTOS possui *drivers* para o TCP/IP embarcados).

Assim sendo, nessa configuração, a *Raspberry Pi* opera como cliente e os kits DSP *Concerto* como servidores da rede. Para isso, foram implementados em linguagem C dois códigos para gerenciar a comunicação com cada kit de DSP. Os códigos para ambos os DSPs são muito similares, diferenciando nas variáveis que são adquiridas e atualizadas no banco de dados. Um fluxograma representando o funcionamento da aplicação TCP/IP na *Raspberry Pi* pode ser conferido na Figura 25.

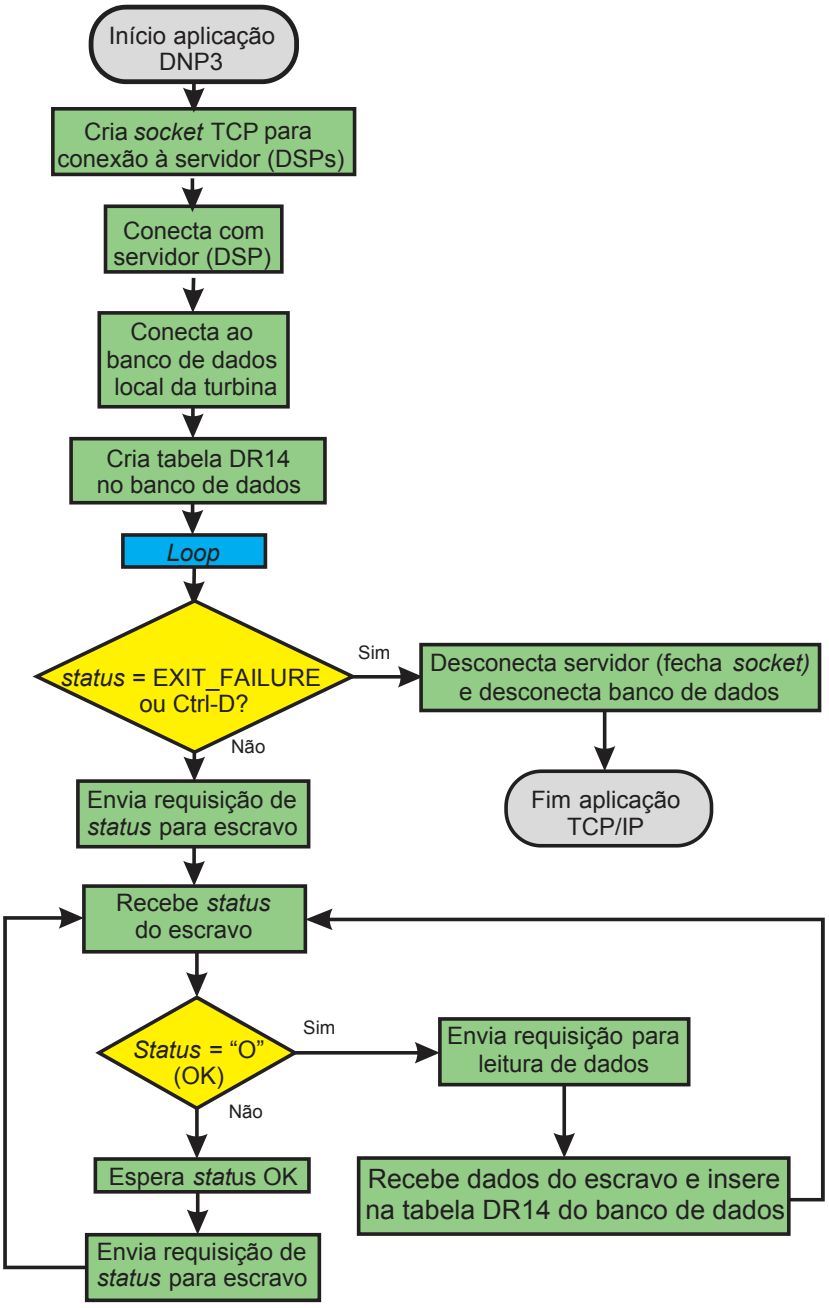


Figura 25- Fluxograma do código de aplicação TCP/IP cliente na *Raspberry Pi*

Como é possível visualizar no fluxograma da Figura 25, existe um tratamento inicial entre cliente e servidor, antes de efetuar a troca de dados, para que a estação mestre somente receba os dados (variáveis) do escravo quando o estado do mesmo estiver "*OK*". A Figura 26 exemplifica melhor esse tratamento inicial. Após receber o *status* do escravo (DSP *Concerto*), caso for "O" (*OK*) a comunicação é estabelecidade e a troca de dados ocorre enquanto o estado não mudar pra "N" (*NOK*). Caso o *status* seja "N", a estação mestre aguarda até que o *status* retorne para "O", para então posseguir com a troca de dados. Se ocorrer uma falha nas operações

de envio ou recebimento dos pacotes TCP a conexão é encerrada e a aplicação termina, retornando o estado do erro.

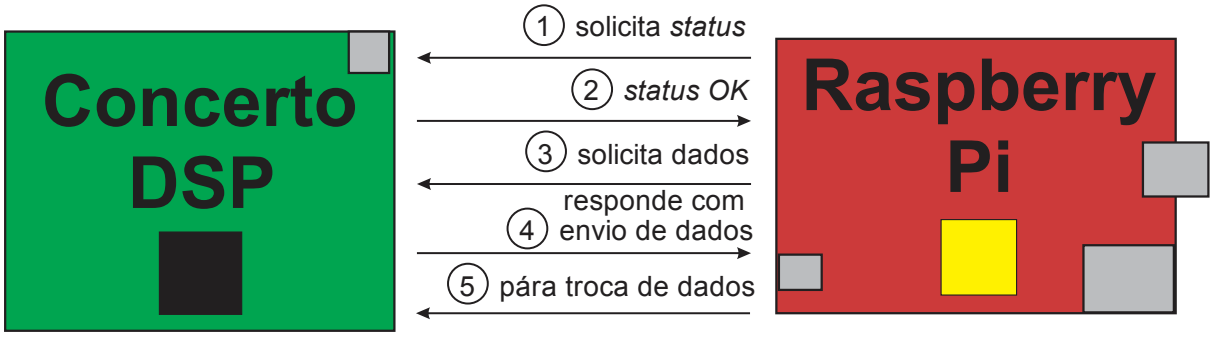


Figura 26- Tramento inical da comunicação via TCP/IP entre DSPs *Concerto* e *Raspberry Pi.*

A aplicação TCP/IP é executada por meio de um comando na linha de comando da *Raspberry Pi*. O comando consiste do endereço IP do escravo ao qual a estação mestre se conectará, a porta, o identificador do executável, o tamanho do pacote TCP e o tempo entre transmissões. O código da aplicação TCP/IP do DSP *Concerto* da *nacelle* pode ser conferido com mais detalhes no Anexo B.

5.2.3. Protocolo Juliano (camada de aplicação e processo TCP/IP)

Como o protocolo TCP/IP não engloba a camada de aplicação e processo do modelo ARPA (vide Figura 3), foi necessário implementá-la neste trabalho. Desta maneira foi criado o protocolo Juliano, que leva o nome de seu autor.

A camada mais alta do TCP/IP (camada de serviços) disponibiliza o *frame* do TCP/IP, com os dados da aplicação (dados crus, não tratados). Este *buffer* de dados deve ser tratado de maneira à fazer sentido para a estação que está recebendo-o ou enviando-o.

Neste trabalho, os *kits Concerto* enviam os dados para a *Raspberry Pi* por meio desse *buffer* de dados do TCP/IP (que contém 1024 bytes do tipo *char*). Como os dados, na sua grande maioria, são do tipo ponto flutuante de 32 bits, foi necessário convertê-los para sua representação em formato inteiro de 32 bits (conforme IEEE 754) e posteriormente escrevê-los no *buffer* de dados à ser enviado (pois o *buffer* do TCP/IP aceita somente bytes do tipo *char*) . Tal conversão é realizada por meio do tipo *Union* em C/C++.

A *Union* é uma estrutura que pode ser retratada como um pedaço de memória que é utilizada para armazenar variáveis de diferentes tipos de dados. Uma vez que um novo valor é atribuído a um campo, os dados existentes serão substituídos pelos novos dados. A área de

memória que armazena o valor não tem qualquer tipo intrínseco (não só bytes ou palavras de memória), mas o valor pode ser tratado como um dos vários tipos de dados abstratos (ponto flutuante, interio, *char* e etc.), tendo o tipo do último valor que foi escrito na memória. Portanto, em ambas estações (servidor – *kit Concerto* e cliente – *Raspberry Pi*), foram definidos os tipos *Union*, para envio e recebimento de dados, com representações em ponto flutuante de 32 bits e inteiro de 32 bits. Como é possível visualizar no fragmento abaixo extraído do código fonte referente a CPU Cortex-M3 do *kit Concerto,* presente no Anexo D.

```
typedef union {
  int i;
  float f;
 } u;
```

Após o recebimento dos dados pelo cliente *Raspberry Pi*, o *buffer* de dados deve ser tratado novamente, ou seja, a mensagem deve ser codificada. O dados são extraídos do *buffer*, convertidos de *struct* para inteiro de 32 bits e escritos na parcela inteira da *Union,* assim, os mesmos dados estarão representados em formato de ponto flutuante (IEEE 754) na parcela de ponto flutuante da estrutura *Union,* a conversão é direta. Assim, os dados estão disponíveis para escrita no banco de dados, e encapsulamento no protocolo DNP3 (para posterior envio ao SCADA).

5.3 CONFIGURAÇÃO DSPs *CONCERTO* F28M36x

Os *kits* de DSP *Concerto* são responsáveis pela aquisição de dados e controle em tempo real na turbina eólica. Lembrando que cada kit possui um processador de dois núcleos, microcontroladores TMS320F28335 e Cortex-M3, sendo o primeiro responsável por efetuar aquisição de dados e controle em tempo real e o segundo responsável pela comunicação com a *Raspberry Pi.* Para comunicação entre os núcleos do DSP *Concerto* foi utilizado *Message Queue* componente do TI-RTOS usado para comunicação entre processadores (IPC – *inter-processor commnication*).

5.3.1. Programação do Microcontrolador TMS320F28335

Como já foi dito anteriormente, o TMS320F28335, é responsável pela aquisição de dados e controle em tempo real. Para que isso seja possível os módulos de conversão analógico-digital (ADC) e de modulação do largura de puslo (PWM) devem ser configurados. Para a aplicação

deste trabalho a conversão analógico-digital é disparada pelo módulo PWM à uma frequência de 1,144 kHz que depende do modo de operação e do registrador de base de tempo (TEXAS INSTRUMENTS, 2009). Para cálculo da frequência de PWM de 1,144 kHz, no modo *Up and Down Count*, é usada a Equação 1 abaixo. A Figura 27 ilustra este modo do módulo PWM.

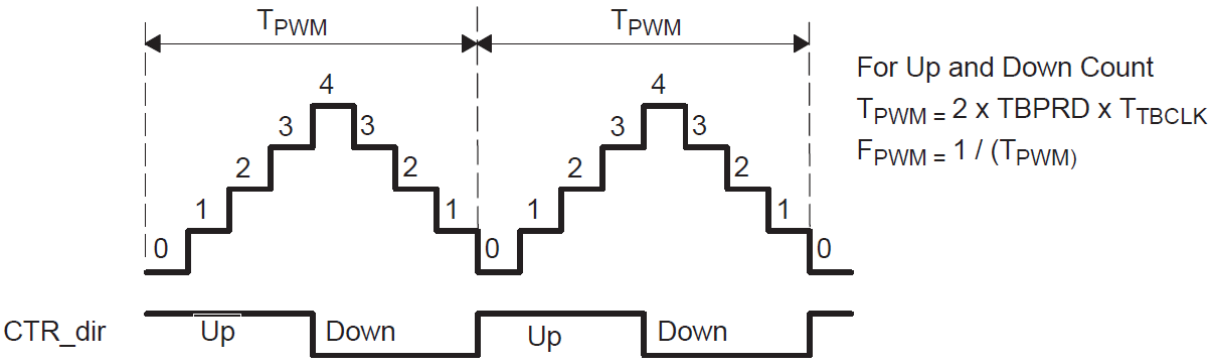


Figura 27- Modo *Up and Down Count* do móudlo PWM do TMS320F28335. (TEXAS INSTRUMENTS, 2009)

$$F_{PWM} = \frac{1}{2 \cdot TBPRD \cdot T_{TBCLK}} \tag{1}$$

onde,

F_{PWM} = frequência do PWM,

$TBPRD$ = registrador de período de base de tempo,

T_{TBCLK} = *clock* de base de tempo, inverso da frequência de operação da CPU.

desta forma, para se obter $F_{PWM} = 1{,}144\ kHz$

$$1{,}144\ kHz = \frac{1}{2 \cdot TBPRD \cdot (1/150\ MHz)}$$

$$TBPRD = 65535$$

Esta é a mínima frequência de PWM possível de ser sintetizada com a CPU operado na frequência de 150 MHz, pois o registrador TBPRD é do tipo interio sem sinal (16 bits), podendo possuir valor máximo de 65535.

Após calcular a frequência de operação do módulo PWM, é necessário parametrizar os registradores de modo que o módulo PWM1 faça o disparo da interrupção da conversão analógico-digital ADC1. Tal cofiguração, juntamente com a configuração do módulo *Message Queue* (componente do TI-RTOS), pode ser analisada com melhores detalhes por meio do Anexo C, onde o código implementado no microcontroaldor TMS320F28335 do kit DSP *Concerto* da *nacelle* está descrito.

5.3.2. Programação do Microcontrolador Cortex-M3

O microcontrolador Cortex-M3 do kit DSP *Concerto*, realiza comunicação com a *Raspberry Pi* (Mestre), enviando os dados relacionados à turbina eólica. Tal comunicação é realizada por meio do protocolo TCP/IP. Para isso foram criadas duas tarefas (*threads*) no TI-RTOS, uma com maior prioridade para comunicação via *Message Queue* com o TMS320F28335 e uma com menor prioridade para comunicação TCP/IP.

O código de implementação dos protocolos *Message Queue* e TCP/IP no Cortex-M3 do kit DSP *Concerto* da Nacelle estão localizados no Anexo D. Um fluxograma simplificado do código está disponível na Figura 28.

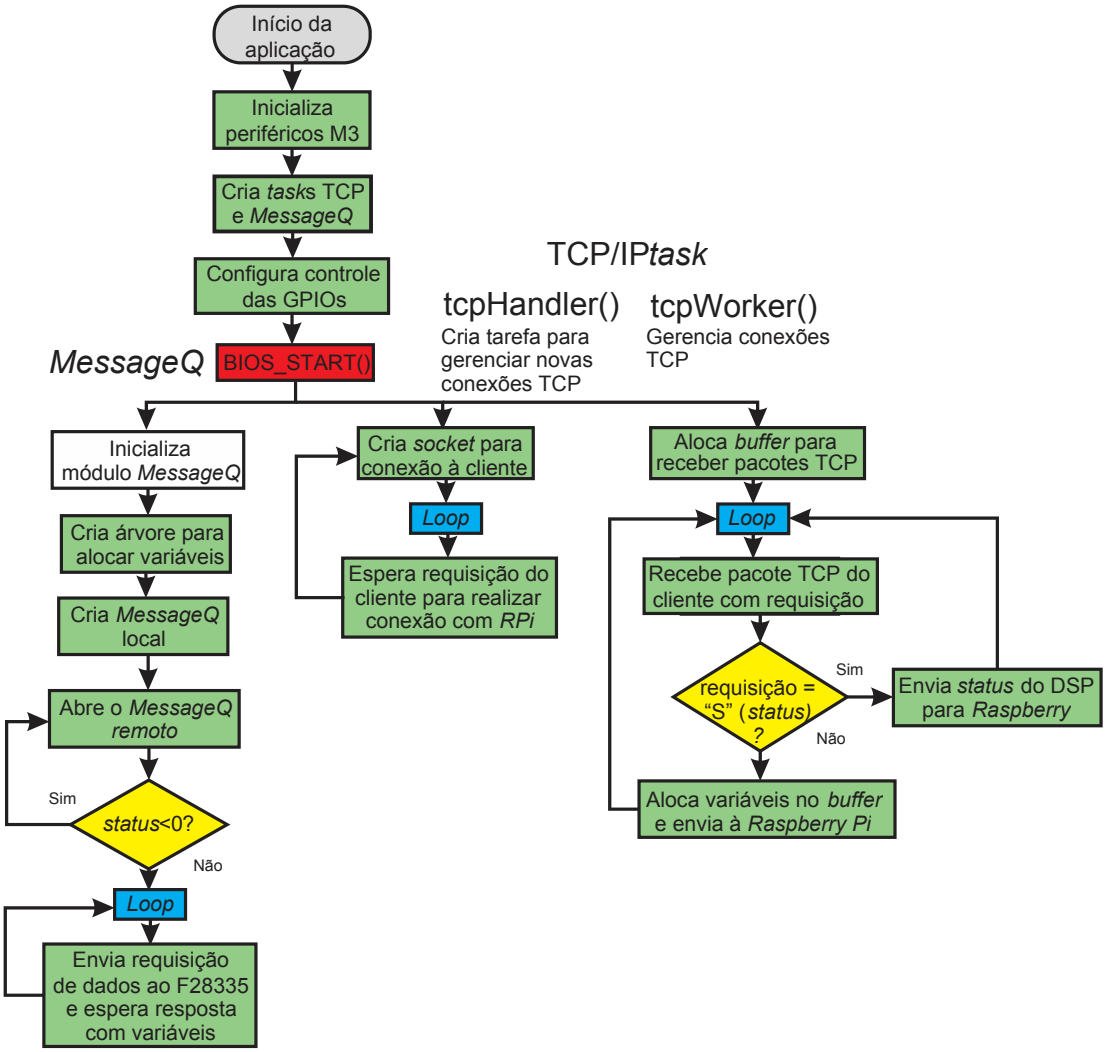


Figura 28- Fluxograma representando código do Cortex-M3, om aplicações *MessageQ* e TCP/IP.

Como o Cortex-M3, assim como o TMS320F28335, utiliza o sistema operacional em tempo real da *Texas Instruments* (TI-RTOS), o mesmo possui um escalonador de tarefas que gerencia a execução das mesmas em função de sua prioridade e *deadline*. Neste caso não foi

definido um *deadline* para nenhuma das tarefas, mas a prioridade é maior para a tarefa *Message Queue* e menor para a tarefa TCP/IP. Apesar de não ter sido especificado o *deadline*, cada tarefa possui um tempo mínimo de dormência ao final da mesma, vide códigos Anexos C e D, pela função Task_sleep(*x*) com *x* em *ms*, especificado pelo usuário.

5.4 TESTE COM *SOFTWARE* ANALISADOR DE REDES

Para monitorar a rede de comunicação implementada e confirmar que os protocolos DNP3 e TCP/IP realmente estão presentes nesta, foi utilizado o *software WIRESHARK* (WIRESHARK, 2014), instalado no computador com aplicação SCADA, para monitorar comunicação por meio de protocolo DNP3 entre *Raspberry Pi* e SCADA, e o *software TCPDUMP* (TCPDUMP, 2014), instalado na *Raspberry Pi*, para monitorar comunicação TCP/IP entre DSPs *Concerto* e *Raspberry Pi.* Ambos *softwares* possuem licença aberta.

5.4.1. Comunicação com Protocolo DNP3

O Programa foi configurado de modo a monitorar a rede do protótipo apresentado na Figura 20. Após iniciado o processo de análise uma lista de mensagens circulando na rede é mostrado pelo *software*. A Figura 29 mostra o tráfego de pacotes de dados DNP3 pela rede *Ethernet* configurada. Nesta configuração a *Raspberry Pi* (*outstation*) possui endereço de IP 192.168.1.108 e o SCADA remoto (mestre) possui endereço de IP 192.168.1.106.

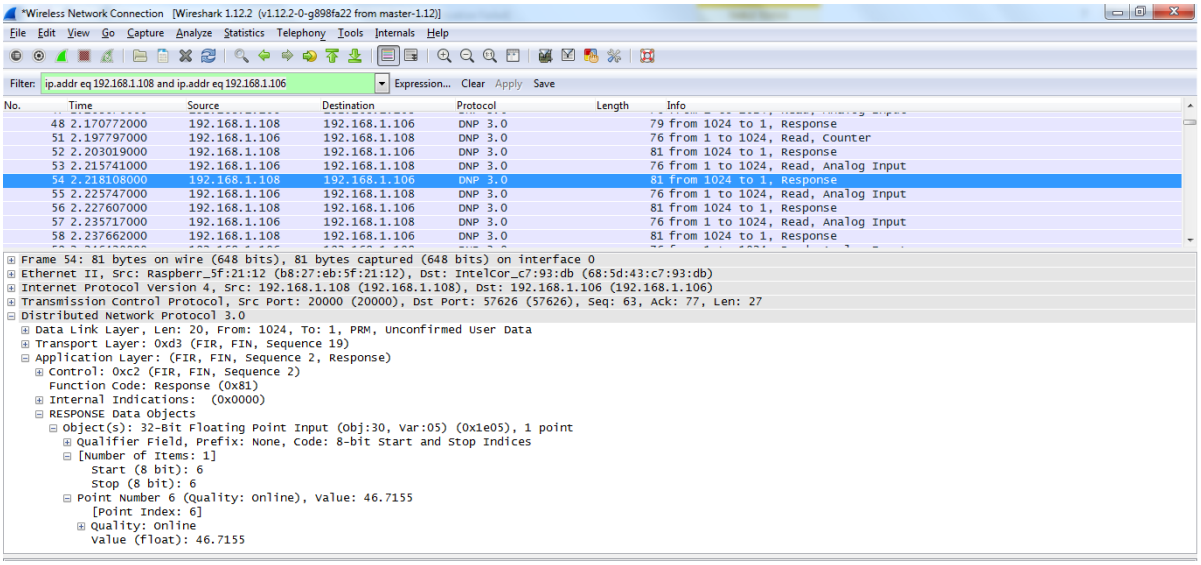


Figura 29- Tráfego de pacotes de dados DNP3 pela rede

Na parte inferior da figura 29 é possível visualizar mais a fundo como a mensagem é construída na rede. Neste caso, uma mensagem de resposta da estação escravo (*Raspberry Pi*)

é enviada ao mestre (PC com aplicação SCADA), a mensagem de resposta é uma variável do tipo Entrada Analógica (ponto flutuante) de 32 bits com endereço 6 (Variável Icc da Tabela 6) e, no caso selecionado, possui valor 46,7155 (codificado conforme IEEE 754). Também é possível visualizar que o tempo entre uma requisição de leitura do mestre e resposta do escravo é de aproximadamente 2 ms.

5.4.2. Comunicação com Protocolo TCP/IP

Para monitorar o tráfego de dados entre os kits DSP *Concerto* e a *Raspberry Pi* foi necessário executar o *software TCPDUMP*, por meio de linha de comando na *Raspberry Pi*. A Figura 30 mostra o tráfego de dados capturado pelo *TCPDUMP*. O DSP *Concerto* da Nacelle possui IP 192.168.1.110 e o DSP *Concerto* do Painel de Controle possui IP 192.168.1.109. O tempo entre uma requisição do mestre e resposta do escravo é de aproximadamente 2 ms.

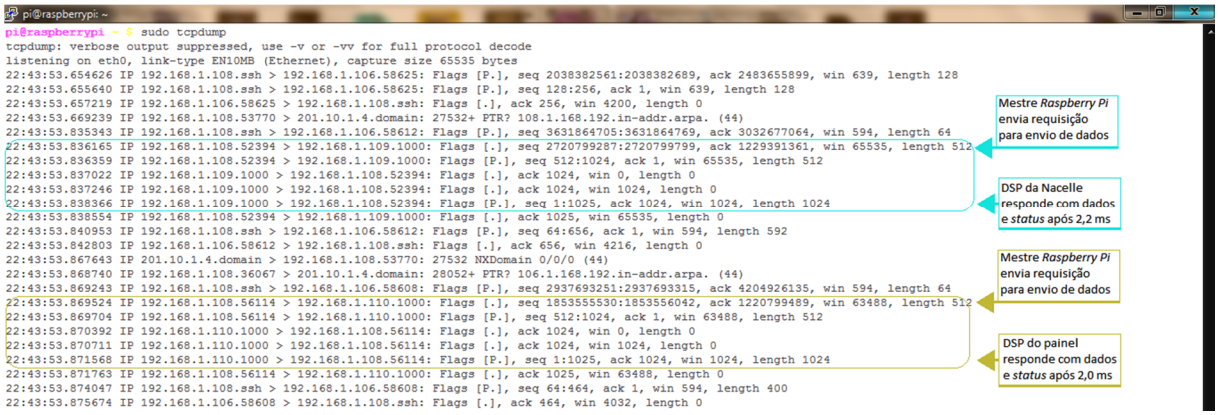


Figura 30- Tráfego de pacotes de dados TCP/IP entre DSPs e *Raspberry Pi.*

5.5 TRABALHOS FUTUROS

- Adequar o código de aplicação de maneira a atender a norma IEC 61400-25, isto é, incluir o modelo abstrato.
- Testar rede de comunicação na turbina eólica, desde a aquisição de dados até o recebimento da estação remota SCADA.
- *Raspberry Pi* deve ler *status* da turbina no PCC e na *nacelle* e comandar a operação da mesma, por meio dos *kits Concerto.*

8. CRONOGRAMA DE EXECUÇÃO

Item/Período	Agosto/14	Setembro14	Outubro/14	Novembro/14 Dezembro/14
Revisão bibliográfica	**X**	**X**		
Definição do protocolo de comunicação e plataforma de *hardware*	**X**	**X**		
Definição da topologia da rede e seus componentes		**X**	**X**	
Estudo da biblioteca openDNP3		**X**	**X**	**X**
Protocolo TCP/IP nos kits *Concerto*			**X**	**X**
Compilação da biblioteca na *Raspberry Pi*			**X**	**X**
Programação do código de aplicação para comunicação com sistema SCADA				**X**
Resultados experimentais			**X**	**X**
Escrita relatório final e artigo científico do trabalho			**X**	**X**

9. CONCLUSÃO

A energia eólica vem demonstrando ser uma energia alternativa extremamente factível e benéfica para os problemas atuais com a alta demanda de energia elétrica. A energia fornecida por meio de hidroelétricas vem cada vez mais demonstrando sua fragilidade em função das mudanças climáticas dos últimos anos.

Serviços como comunicação, instrumentação e medição são essenciais em sistemas de geração de energia distribuídos, como em fazendas eólicas, para operação segura e viável. Prova disto é a norma IEC 61400-25 que estabelece um padrão para o protocolo de comunicação em aerogeradores.

Motivado por essas razões, neste trabalho é apresentado o desenvolvimento e implementação de um protocolo aberto, interoperável e de alto nível para aplicação em um aerogerador, o protocolo DNP3, que possui vantagens notórias para a aplicação em sistemas de geração distribuída. Protocolo este implementado em um ambiente Linux (Raspbian/Debian) de um sistema embarcado da *Raspberry Pi Foundation*, o computador *Raspberry Pi Model-B,* um sistema embarcado do tamanho de um cartão de crédito, e um sistema SCADA da Elipse *Software,* o Elipse SCADA. Além disso, foi implementado o protocolo usando TCP/IP para comunicação interna à turbina eólica, por ser seguro, robusto e eficiente.

Por meio da aplicação, foi possível concluir que o protocolo DNP3 em questão é robusto e seguro a falhas e funciona com dispositivos de diferentes fabricantes (interoperabilidade). Resultados experimentais e apresentação do protótipo comprovam a funcionalidade do protocolo em estudo.

REFERÊNCIAS BIBLIOGRÁFICAS

ANEEL, **Atlas de energia elétrica do Brasil**, 3 ed. Brasília, 2008.

ARM INC., **ARM1176JZF-S™ Revision: r0p7 Technical Reference Manual**, Disponível em<http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf> Acesso em 26 de junho de 2014.

AUTOMATAK, **OpenDNP3 protocol**, 2014. Disponível em:< https://github.com/automatak/dnp3> Acesso em 29 de Abril de 2014.

BROADCOM INC,. **BCM2835 ARM Peripherals datasheet**, 2014. Disponível em <http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf> Acesso em 26 de junho de 2014.

CLARKE, G., REYNDERS, D., **Practical Modern SCADA Protocols: DNP3, 60870.5 and Related Systems**, Grã-Bretanha: Elsevier, 2004.

DE LUCA, V. H. M., et al. Uso da Norma IEC 61400-25 para Comunicação em Usinas Eólicas: Estudo e Análise Prática. Em: Brazil Windpower conference & ehibition, 2013.

ELIPSE SOFTWARE, Elipse SCADA *Software*, 2014.

IEC 61400-25-2, Wind turbines – Part 25-2: Communications for monitoring and control of wind power plants – Information models, December, 2006

IEEE 1815-2012 - IEEE Standard for Electric Power Systems Communications-Distributed Network Protocol (DNP3), IEEE, 2012.

KOHRSEN, T., **Deployment of IEC61400-25-4 - Mapping to Web services on an industrial PLC platform**, Master Thesis, Technical University of Denmark, Kongens Lyngby, 2008.

LOBÃO, E., **Panorama Energético Brasileiro**, Ministério de Minas e Energia, 2008.

MOON, J. I., et al. A hardware implementation of distributed network protocol. Em: Computer Standards & Interfaces 27, 2005, p. 221–232.

OMAR FARUK, A. B. M., **Testing & Exploring Vulnerabilities of the Applications Implementing DNP3 Protocol,** Master Thesis, Norwegian University of Science and Technology, Stockholm - Sweden 2008.

RASPBERRY FOUNDATION, ***Raspberry Pi Documentation***, Disponível em <http://www.raspberrypi.org/documentation/> Acesso em 26 de junho de 2014.

SAN TELMO, E., et al, The Use of 61400-25 Standard to Integrate Wind Power Plants Into the Control of Power Systems Stability, European Wind Energy Conference & Exhibition, Milan, 2007.

SCHAF, F de M., Apostila de Redes Industriais - Evolução Histórica das Redes Industriais, ministrada na disciplina DPEE1052 do curso de Engenharia de Controle e Automação da UFSM, 2012/1.

SHEBI, A. S., et al. Design of Wi-Fi Based Mobile Electrocardiogram Monitoring System on Concerto Platform. Em : Procedia Engineering 64, 2013, p. 65-73.

TANENBAUM, A. S., **Computer Networks,** 4 ed. Prentice Hall, 2003.

TCPDUMP/LIBCAP TEAM, **TCPDUMP** ***network analyzer v4.6.2,*** Disponível em: < http://www.tcpdump.org/> Acesso em 12 de Dezembro de 2014.

TEXAS INSTRUMENTS, F28M36x Concerto™ Microcontrollers datasheet, 2012. Disponível em:<http://www.ti.com/lit/ds/symlink/f28m36h53b2.pdf> Acesso em 29 de Abril de 2014.

TEXAS INSTRUMENTS, TI-RTOS 1.21 Getting Started Guide, 2014. Disponível em:< http://www.ti.com/lit/ug/spruhd3g/spruhd3g.pdf> Acesso em 29 de Abril de 2014.

TEXAS INSTRUMENTS, TMS320x2833x/2823x Enhanced Pulse Width Modulator (ePWM) Module, 2009. Disponível em :<http://www.ti.com/lit/ug/sprug04a/sprug04a.pdf> Acesso em 12 de Dezembro de 2014.

WIRESHARK TEAM, **WIRESHARK** ***network analyzer*** **r1.10.8**, Disponível em <http://www.wireshark.org/download.html> Acesso em 26 de junho de 2014.

GLOSSÁRIO

Analógico: Um fenômeno contínuo em tempo real aonde os valores de informação são representados por uma forma de onda contínua e variável.

Arquitetura da Rede: Um conjunto de princípios de projeto, incluindo a organização de funções e a descrição do formato dos dados usado de base para o projeto e implementação de uma rede.

ARM: *Advanced* RISC *Machine* é uma arquitetura de processador de 32 bits e é usada principalmente em sistemas embarcados.

Asynchronous: Comunicação onde os caracteres podem ser transmitida de maneira arbitrária e dessincronizada, e onde o intervalo de tempo entre a transmissão de caracteres pode ser variável. A comunicação é controlada por bits de início e parada.

RTOS: *Real Time Operational System* – Sistema Operacional em Tempo Real

BIOS: *Basic Input/Output System* – Sistema Básico de Entrada/Saída

Bit: Derivado de '*BInary digiT'*, uma condição um ou zero em um sistema binário.

Byte: Formado por oito bits de dados.

C/C++: Linguagem de programação de alto nível.

CCITT: *Consultative Committee International Telegraph and Telephone*. Organização internacional que impõe padrões de telecomunicações no mundo todo (ex.: V.24).

CLP: Controlador Lógico Programável

CPU: Unidade Central de Processamento

CRC : *Cyclic redundancy check*, ou verificação de redundância cíclica, é um método para checagem de erros, que se baseia em tratar sequências de bits da mensagem.

DCCS: *Distributed Computer Control Systems* – Sistemas Distribuídos de Controle por Computador.

DCE: equipamento responsável por realizar a comunicação dos dados.

Digital: Um sinal que possui estados definidos (normalmente dois)

DNP3: *Distributed Network Protocol version 3.0* – Protocolo de Rede Distribuído

DSP: *Digital Signal Processor* – Processador Digital de Sinais.

DTE: Equipamento onde os dados terminam e onde também podem ser iniciados.

FatFS: Módulo FAT genérico de sistemas de arquivos.

GPRS: *General Packet Radio Service* – Serviço de Rádio de Pacote Geral.

GPU: *Graphic Processing Unit* - Unidade de Processamento Gráfico.

GUI: *Graphical Human Interface* – Interface Gráfica do Utilizador

HDMI: *High-Definition Multimedia Interfac.* É uma interface condutiva totalmente digital de áudio e vídeo capaz de transmitir dados não comprimidos

I²C: *Inter-Integrated Circuit*. É um barramento serial multi-mestre desenvolvido pela Philips, usado para conectar periféricos de baixa velocidade.

I²S: *Integrated Interchip Sound*. É um padrão de barramento serial usado para conectar dispoitivos digitais de áudio.

IEC: *International Electrotechnical Commission* – Comissão Internacional de Eletrotécnica

IEC 60870-5-104: Protocolo usado em SCADA na automação de sistemas de potência. **IED**: dispositivos eletrônicos inteligentes.

IHM: Interface Homem Máquina.

IP: Protocolo de *Internet.*

LAN: *Local Area Network.* Sistema de comunicação de dados confinado à uma área limitada, tipicamente de 3 km, com altas taxas de transferência (4 Mbps - 155 Mbps).

LINUX: Termo utilizado para se referir a sistemas operativos ou sistemas operacionais que utilizem o núcleo Linux.

Log: registro de eventos em um sistema de computadores

Multidrop: Barramento de comunicação usado para conectar três ou mais pontos.

Nacele: Designação normalmente dada ao suporte do conjunto gerador em uma turbina eólica.

Topologia da Rede: Relaçãp física e lógica dos nós de uma rede.

NTSC: *National Television System Committee* – sistema de televisão analógico em uso nas Américas.

OPC-XML-DA:

Open-Source: Código aberto, *software* de licença livre.

PAL: *Phase Alternating Line,* uma forma de codificação da cor usada nos sistemas de transmissão televisiva.

Periféricos: Dispositivos de entrada e saída de dados conectados à um computador. Ex.: teclados, *mouse*, impressora, disco rígido entre outros.

Ponto-à-ponto: Conexão entre dois dispositivos finais.

Protocolo: Um conjunto formal de convenções que governa a formatação, procedimentos de controle e temporização das mensagens trocadas entre dois sistemas de comunicação.

PWM: Modulação por Largura de Pulso.

RAM: Memória de Acesso Randômico. Memória semicondutora volátil de leitura/escrita, cujos dados são perdidos quando a alimentação é desligada.

RCA composto: Vídeo composto é o formato de um sinal de TV analógica (somente imagens) antes de ser combinado com um sinal de som e modulado em uma portadora de rádio frequência.

RISC: *Reduced Instruction Set Computer* ou Conjunto Reduzido de Instruções de Computador é uma linha de arquitetura de processadores que favorece um conjunto simples e pequeno de instruções que levam aproximadamente a mesma quantidade de tempo para serem executadas.

Roteador: Dispositivo de conecta os segmentos da rede e que opera nas três camadas do modelo ISO/OSI (física, rede e enlace de dados).

RS-232C ou V.24: Padrão para troca serial de dados binários entre um DTE (terminal de dados) e um DCE (comunicador de dados).

RTU: Unidade remota de telemetria.

RJ-45: Conector usado para comunicação em redes.

SCADA: Sistema de supervisão e aquisição de dados.

Serial: Modo de transmissão mais comum, aonde a informação é enviado sequencialemnte de um único canal de dados.

SPI/SSI: Protocolo de dados síncrono e serial usado por microcontroladores para comunicação com um ou mais periféricos de maneira rápida e sobre pequenas distâncias.

Smart Grids: Redes elétricas inteligentes.

SoE: Sequência de Eventos de um determinado sistemas, registrado via *log.*

TCP: *Transmission Control Protocol* - Protocolo de Controle de Transmissão.

Timers: Dispositivos contadores de tempo.

Watchdog: Temporizador que dispara um *reset* ao sistema se o programa principal, devido a alguma condição de erro, deixar de reiniciar o *watchdog timer.*

***Web* Services:** Solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes.

YAW: orientação da turbina, ângulo de guinada.

UART: *universal asynchronous receiver/transmitter* – receptor/transmissor de dados assíncrono universal.

USB: *Universal Serial Bus*- Barramento Serial Universal.

ANEXO A

Código de aplicação servidor DNP3 na *Raspberry Pi*

```
////////////////////////////////////////////////////////////////////////////
// UNIVERSIDADE FEDERAL DE SANTA MARIA-UFSM
// GRUPO DE ELETRONICA DE POTENCIA E CONTROLE-GEPOC
// PROJETO EÓLICA 2014
////////////////////////////////////////////////////////////////////////////
// TCC-IMPLEMENTAÇÃO DE INTERFACE DE COMUNICAÇÃO SEGUINDO IEC 61400-25
// PARA AEROGERADORES
// Código DNP3 Raspberry Pi
// AUTOR: Juliano Grigulo E-MAIL: jgrigulo@gmail.com
////////////////////////////////////////////////////////////////////////////
// DESCRIÇÃO:
// Este código realiza:
// > Leitura dos dados da turbina eólica de um banco de dados
// > Encapsulamento dos dados nos padrões do protocolo DNP3
////////////////////////////////////////////////////////////////////////////

#include <opendnp3/LogToStdio.h>
#include <opendnp3/DNP3Manager.h>
#include <opendnp3/SlaveStackConfig.h>
#include <opendnp3/IChannel.h>
#include <opendnp3/IOutstation.h>
#include <opendnp3/SimpleCommandHandler.h>
#include <opendnp3/TimeTransaction.h>
#include <mysql/mysql.h>
#include <mysql/my_global.h>
#include <unistd.h>
#include <string>
#include <iostream>
#include <stdio.h>
using namespace std;
using namespace opendnp3;

// ERROR BLOCK
void finish_with_error(MYSQL *con)
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}

// MAIN CODE
int main(int argc, char* argv[])
{
//VARIAVEIS AUXILIARES PARA LER DO BANCO DE DADOS
 // NACELLE
 char  *vac;
 char  *vbc;
 char  *vca;
 char  *vcc_batt;
 char  *vcc;
 char  *icc;
 char  *pcc;
 char  *wind_speed;
```

```
char  *wind_direc;
char  *yaw_pos;
char  *fim_curso_yaw;
char  *fim_curso_passo;
char  *torque;
char  *encoder;
char  *ptc_temp;
char  *i2c_temp;
char  *kty84_temp;
char  *heatsink_temp;
char  *freio_nacelle;
char  *timestamp;
// PAINEL
char  *P_grid;
char  *Q_grid;
char  *S_grid;
char  *V_Rgrid;
char  *V_Sgrid;
char  *V_Tgrid;
char  *Vcc;
char  *Status_inv;
char  *Status_grid;
char  *Temp_inv;
char  *time_ref;

//VARs AUXs para CONVERSAO STRING Para DOUBLE
// NACELLE
std::string  vac_s ;
std::string  vbc_s ;
std::string  vca_s ;
std::string  vcc_batt_s ;
std::string  vcc_s ;
std::string  icc_s ;
std::string  pcc_s ;
std::string  wind_speed_s ;
std::string  wind_direc_s ;
std::string  yaw_pos_s ;
std::string  fim_curso_yaw_s ;
std::string  fim_curso_passo_s ;
std::string  torque_s ;
std::string  encoder_s ;
std::string  ptc_tempo_s ;
std::string  i2c_temp_s ;
std::string  kty84_temp_s ;
std::string  heatsink_temp_s ;
std::string  freio_nacelle_s ;
std::string  timestamp_s ;
// PAINEL
std::string  P_grid_s ;
std::string  Q_grid_s ;
std::string  S_grid_s ;
std::string  V_Rgrid_s ;
std::string  V_Sgrid_s ;
std::string  V_Tgrid_s ;
std::string  Vcc_s ;
std::string  Status_inv_s ;
std::string  Status_grid_s ;
std::string  Temp_inv_s ;
std::string  time_ref_s ;
```

```
//Variáveis  tipo double para envio ao mestre SCADA
 // NACELLE
 double   vac_d ;
 double   vbc_d ;
 double   vca_d ;
 double   vcc_batt_d ;
 double   vcc_d ;
 double   icc_d ;
 double   pcc_d ;
 double  wind_speed_d ;
 double  wind_direc_d ;
 double  yaw_pos_d ;
 double  fim_curso_yaw_d ;
 double  fim_curso_passo_d ;
 double  torque_d ;
 double  encoder_d ;
 double  ptc_tempo_d ;
 double  i2c_temp_d ;
 double  kty84_temp_d ;
 double  heatsink_temp_d ;
 double  freio_nacelle_d ;
 double  timestamp_d ;
 // NACELLE
 double   P_grid_d ;
 double   Q_grid_d ;
 double   S_grid_d ;
 double   V_Rgrid_d ;
 double   V_Sgrid_d ;
 double   V_Tgrid_d ;
 double   Vcc_d ;
 double   Status_inv_d ;
 double   Status_grid_d ;
 double   Temp_inv_d;
 double   time_ref_d ;

//MYSQL conexão ao banco de dados
  MYSQL *con = mysql_init(NULL);
  if (con == NULL)
  {
      fprintf(stderr, "mysql_init() failed\n");
      exit(1);
  }
  if (mysql_real_connect(con, "localhost", "user12", "34klq*",
          "testdb", 0, NULL, 0) == NULL)
  {
      finish_with_error(con);
  }
//INÍCIO PROTOCOLO DNP3
      // Specify a FilterLevel for the stack/physical layer to use.
      // Log statements with a lower priority will not be logged.
      const FilterLevel LOG_LEVEL = LEV_INFO;

      // This is the main point of interaction with the stack
      DNP3Manager mgr(1); // only 1 thread is needed for a single stack

      // You can optionally subcribe to log messages
      // This singleton logger just prints messages to the console
      mgr.AddLogSubscriber(LogToStdio::Inst());

      // Add a TCPServer to the manager with the name "tcpserver".
```

```
	// The server will wait 5000 ms in between failed bind calls.
	auto pServer = mgr.AddTCPServer("tcpserver", LOG_LEVEL, 5000,
"0.0.0.0", 20000);

	// You can optionally add a listener to the channel. You can do this
anytime and
	// you will receive a stream of all state changes
	pServer->AddStateListener([](ChannelState state) {
		std::cout << "Server state: " <<
ConvertChannelStateToString(state) << std::endl;
	});

	// The master config object for a slave. The default are
	// useable, but understanding the options are important.
	SlaveStackConfig stackConfig;

	// The DeviceTemplate struct specifies the structure of the
	// slave's database
	DeviceTemplate device(35, 35, 35, 35, 35);
	stackConfig.device = device;
	stackConfig.device.mStartOnline = true;
	stackConfig.slave.mDisableUnsol = true;

	// Create a new slave with a log level, command handler, and
	// config info this returns a thread-safe interface used for
	// updating the slave's database.
	auto pOutstation = pServer->AddOutstation("outstation", LOG_LEVEL,
SuccessCommandHandler::Inst(), stackConfig);

	// You can optionally add a listener to the stack to observer
communicate health. You
	// can do this anytime and you will receive a stream of all state
changes.
	pOutstation->AddStateListener([](StackState state) {
		std::cout << "outstation state: " <<
ConvertStackStateToString(state) << std::endl;
	});
	auto pDataObserver = pOutstation->GetDataObserver();
	std::string input;

	do {

		usleep(500000); //Espera 500 ms para próxima comunicação

		if(input == "exit") break;
		else {
			TimeTransaction tx(pDataObserver); //automatically calls
Start()/End() and sets time for each measurement

  //SELECIONA TABELA DR14 DO BANCO DE DADOS
  if (mysql_query(con, "SELECT * FROM DR14_Nacelle"))
  {
      finish_with_error(con);
  }

  MYSQL_RES *result = mysql_store_result(con);

  if (result == NULL)
  {
      finish_with_error(con);
```

```
}
int num_fields = mysql_num_fields(result);

MYSQL_ROW row;
//Adquire variáveis do banco de dados e salva nas variáveis locais
      //Nacelle
      row = mysql_fetch_row(result);
      vac = row[2];
      row = mysql_fetch_row(result);
      vbc = row[2];
      row = mysql_fetch_row(result);
      vca = row[2];
      row = mysql_fetch_row(result);
      vcc_batt = row[2];
      row = mysql_fetch_row(result);
      vcc = row[2];
      row = mysql_fetch_row(result);
      icc = row[2];
      row = mysql_fetch_row(result);
      pcc = row[2];
      row = mysql_fetch_row(result);
      wind_speed = row[2];
      row = mysql_fetch_row(result);
      wind_direc = row[2];
      row = mysql_fetch_row(result);
      yaw_pos = row[2];
      row = mysql_fetch_row(result);
      fim_curso_yaw = row[2];
      row = mysql_fetch_row(result);
      fim_curso_passo = row[2];
      row = mysql_fetch_row(result);
      torque = row[2];
      row = mysql_fetch_row(result);
      encoder = row[2];
      row = mysql_fetch_row(result);
      ptc_temp = row[2];
      row = mysql_fetch_row(result);
      i2c_temp = row[2];
      row = mysql_fetch_row(result);
      kty84_temp = row[2];
      row = mysql_fetch_row(result);
      heatsink_temp = row[2];
      row = mysql_fetch_row(result);
      freio_nacelle = row[2];
      row = mysql_fetch_row(result);
      timestamp = row[2];
      //painel
      row  = mysql_fetch_row(result);
      P_grid = row[2];
      row = mysql_fetch_row(result);
      Q_grid = row[2];
      row = mysql_fetch_row(result);
      S_grid = row[2];
      row = mysql_fetch_row(result);
      V_Rgrid = row[2];
      row = mysql_fetch_row(result);
      V_Sgrid= row[2];
      row = mysql_fetch_row(result);
      V_Tgrid = row[2];
      row = mysql_fetch_row(result);
```

```
        Vcc = row[2];
        row = mysql_fetch_row(result);
        Status_inv = row[2];
        row = mysql_fetch_row(result);
        Status_grid = row[2];
        row = mysql_fetch_row(result);
        Temp_inv = row[2];
        row = mysql_fetch_row(result);
        time_ref = row[2];
mysql_free_result(result);

//Passa variáveeis do tipo ponteiro para
//char para variáveis do tipo string
vac_s = vac;
vbc_s = vbc;
vca_s = vca;
vcc_batt_s = vcc_batt;
vcc_s = vcc;
icc_s = icc;
pcc_s = pcc;
wind_speed_s = wind_speed;
wind_direc_s = wind_direc;
yaw_pos_s = yaw_pos;
fim_curso_yaw_s = fim_curso_yaw;
fim_curso_passo_s = fim_curso_passo;
torque_s = torque;
encoder_s = encoder;
ptc_tempo_s = ptc_temp ;
i2c_temp_s = i2c_temp;
kty84_temp_s = kty84_temp;
heatsink_temp_s = heatsink_temp;
freio_nacelle_s = freio_nacelle;
timestamp_s  = timestamp;
P_grid_s = P_grid;
Q_grid_s = Q_grid;
S_grid_s = S_grid;
V_Rgrid_s = V_Rgrid;
V_Sgrid_s = V_Sgrid;
V_Tgrid_s = V_Tgrid;
Vcc_s = Vcc;
Status_inv_s = Status_inv;
Status_grid_s = Status_grid;
Temp_inv_s = Temp_inv;
time_ref_s = time_ref;

// Converte string para double, para envio ao SCADA
      vac_d = std::stod(vac_s);
      vbc_d = std::stod(vbc_s);
      vca_d = std::stod(vca_s);
      vcc_batt_d = std::stod(vcc_batt_s);
      vcc_d = std::stod(vcc_s);
      icc_d = std::stod(icc_s);
      pcc_d = std::stod(pcc_s);
      wind_speed_d = std::stod(wind_speed_s);
      wind_direc_d = std::stod(wind_direc_s);
      yaw_pos_d = std::stod(yaw_pos_s);
      fim_curso_yaw_d = std::stod(fim_curso_yaw_s);
      fim_curso_passo_d = std::stod(fim_curso_passo_s);
      torque_d = std::stod(torque_s);
      encoder_d = std::stod(encoder_s);
```

```
        ptc_tempo_d = std::stod(ptc_tempo_s);
        i2c_temp_d = std::stod(i2c_temp_s);
        kty84_temp_d = std::stod(kty84_temp_s);
        heatsink_temp_d = std::stod(heatsink_temp_s);
        freio_nacelle_d = std::stod(freio_nacelle_s);
        timestamp_d = std::stod(timestamp_s);
        P_grid_d = std::stod(P_grid_s);
        Q_grid_d = std::stod(Q_grid_s);
        S_grid_d = std::stod(S_grid_s);
        V_Rgrid_d = std::stod(V_Rgrid_s);
        V_Sgrid_d = std::stod(V_Sgrid_s);
        V_Tgrid_d = std::stod(V_Tgrid_s);
        Vcc_d = std::stod(Vcc_s);
        Status_inv_d = std::stod(Status_inv_s);
        Status_grid_d = std::stod(Status_grid_s);
        Temp_inv_d = std::stod(Temp_inv_s);
        time_ref_d = std::stod(time_ref_s);

        //Envia dados das variáveis ao SCADA remoto
          tx.Update(Counter(digital, CQ_ONLINE), 0);
          tx.Update(Analog(vac_d, AQ_ONLINE), 1);
          tx.Update(Analog(vbc_d, AQ_ONLINE), 2);
          tx.Update(Analog(vca_d, AQ_ONLINE), 3);
          tx.Update(Analog(vcc_batt_d, AQ_ONLINE), 4);
          tx.Update(Analog(vcc_d, AQ_ONLINE), 5);
          tx.Update(Analog(icc_d, AQ_ONLINE), 6);
          tx.Update(Analog(pcc_d, AQ_ONLINE), 7);
          tx.Update(Analog(wind_speed_d, AQ_ONLINE), 8);
          tx.Update(Analog(wind_direc_d, AQ_ONLINE), 9);
          tx.Update(Analog(yaw_pos_d, AQ_ONLINE), 10);
          tx.Update(Analog(fim_curso_yaw_d, AQ_ONLINE), 11);
          tx.Update(Analog(fim_curso_passo_d, AQ_ONLINE), 12);
          tx.Update(Analog(torque_d, AQ_ONLINE), 13);
          tx.Update(Analog(encoder_d, AQ_ONLINE), 14);
          tx.Update(Analog(ptc_tempo_d, AQ_ONLINE), 15);
          tx.Update(Analog(i2c_temp_d, AQ_ONLINE), 16);
          tx.Update(Analog(kty84_temp_d, AQ_ONLINE), 17);
          tx.Update(Analog(heatsink_temp_d, AQ_ONLINE), 18);
          tx.Update(Analog(freio_nacelle_d, AQ_ONLINE), 19);
          tx.Update(Analog(timestamp_d, AQ_ONLINE), 20);
          tx.Update(Analog(P_grid_d, AQ_ONLINE), 21);
          tx.Update(Analog(Q_grid_d, AQ_ONLINE), 22);
          tx.Update(Analog(S_grid_d, AQ_ONLINE), 23);
          tx.Update(Analog(V_Rgrid_d, AQ_ONLINE), 24);
          tx.Update(Analog(V_Sgrid_d, AQ_ONLINE), 25);
          tx.Update(Analog(V_Tgrid_d, AQ_ONLINE), 26);
          tx.Update(Analog(Vcc_d, AQ_ONLINE), 27);
          tx.Update(Analog(Status_inv_d, AQ_ONLINE), 28);
          tx.Update(Analog(Status_grid_d, AQ_ONLINE), 29);
          tx.Update(Analog(Temp_inv_d, AQ_ONLINE), 30);
          tx.Update(Analog(time_ref_d, AQ_ONLINE), 31);
          }
        }
        while(true);
        //Fecha conexão ao banco de dados
        mysql_close(con);
        return 0;
}
```

ANEXO B

Código de aplicação cliente TCP/IP na *Raspberry Pi*

```
////////////////////////////////////////////////////////////////////////
// UNIVERSIDADE FEDERAL DE SANTA MARIA-UFSM
// GRUPO DE ELETRONICA DE POTENCIA E CONTROLE-GEPOC
// PROJETO EÓLICA 2014
////////////////////////////////////////////////////////////////////////
// TCC-IMPLEMENTAÇÃO DE INTERFACE DE COMUNICAÇÃO SEGUINDO IEC 61400-25
// PARA AEROGERADORES
// Código comunicação TCP/IP Raspberry Pi
// AUTOR: Juliano Grigulo E-MAIL: jgrigulo@gmail.com
////////////////////////////////////////////////////////////////////////
// DESCRIÇÃO:
// Este código realiza:
// > Comunicação com MCU CORTEX-M3 do Concerto via TCP/IP
// > Leitura de variáveis do DSP e postagem no Banco de Dados DR14
////////////////////////////////////////////////////////////////////////
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <mysql/my_global.h>
#include <mysql/mysql.h>
#include "sockets.h"
#define MAXBUF    1024
// Artifício para conversão de dados de int para float, para posterior
//transmissão
typedef union {
int i;
float f;
} u;

u vac;
u vbc;
u vca;
u vcc_batt;
u vcc;
u icc;
u pcc;
u wind_speed;
u wind_direc;
u yaw_pos;
u fim_curso_yaw;
u fim_curso_passo;
u torque;
u encoder;
u ptc_temp;
u i2c_temp;
u kty84_temp;
u heatsink_temp;
u freio_nacelle;
u tempo

//  ======== error ========
void finish_with_error(MYSQL *con)
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}
// ======== main code========
int main(int argc, char *argv[])
{

    int i, c, k, first = 1;
    int sockfd = 0;
    int bytes_read, total_bytes_read, bytes_sent;
```

```
    struct addrinfo hints;
    struct addrinfo *results = NULL;
    int status = EXIT_SUCCESS;
    int count = 0;
    int id;
    int value, val;
    unsigned int sleepTime = 1000;
    unsigned int buffSize  = MAXBUF;
    char *buffer = NULL, str[11], str1[100] = "My String", *statusDSP;
    char  resa[25]  =  "carro1",  resb[25]  =  "carro2",  resc[25]  =  "carro3",
resd[25] = "carro4", rese[25] = "carro5";
    char statement[1024], *my_str = "MyString";
    float teste=5.5;

    time_t start;
    start = time(NULL);

    char * pquerystring = NULL;

    /* parameter check */
    if (argc < 4 || argc > 6) {
        printf("usage: %s <IPv4 or IPv6 addr> <port> <id> -l[length] -s[sleep
in uS]\n", argv[0]);
        status = EXIT_FAILURE;
        goto QUIT;
    }

    id = atoi(argv[3]);

    /* Parse options */
    i = argc - 1;
    while ((i > 3) && (argv[i][0] == '-')) {
        switch (argv[i][1]) {
            case 'l':
                buffSize = atoi(&argv[i][2]);
                break;
            case 's':
                sleepTime = atoi(&argv[i][2]);
                break;
            default:
                printf("Valid options are -l[length] and -s[sleep in uS]\n");
                status = EXIT_FAILURE;
                goto QUIT;
        }
        i--;
    }

    buffer = malloc(buffSize);
    if (buffer == NULL) {
        printf("malloc failed\n");
        status = EXIT_FAILURE;
    }

    memset(buffer, 0, buffSize);

    /* initialize sockets environment */
    socketsStartup();

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
```

```
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    /*
     * getaddrinfo() fills in the results struct for us appropriately
     * depending on whether the IP address is v4 or v6
     *
     * argv[1] = IPv4 or IPv6 address passed in from command line
     * argv[2] = port number passed in from command line
     */
    value = getaddrinfo(argv[1], argv[2], &hints, &results);

    if (value != 0) {
        fprintf(stderr, "getaddrinfo failed: %d\n", value);
        if (value == -2 || value == 11004) {
            fprintf(stderr, "unrecognized IP address\n");
        }
        status = EXIT_FAILURE;
        goto QUIT;
    }

    /* create socket. ai_family determined for us via getaddrinfo() call */
    if ((sockfd = socket(results->ai_family, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "socket failed: %d\n", errno);
        status = errno;
        goto QUIT;
    }

    /* connect. ai_addr set to AF_INET or AF_INET6 by getaddrinfo() call */
    if (connect(sockfd, results->ai_addr, results->ai_addrlen) < 0) {
        printf("connect failed: %d\n", errno);
        status = errno;
        goto QUIT;
    }

    printf("Starting test with a %d uSec delay between transmits\n",
sleepTime);

      //MYSQL DATABASE CONNECTION//////////////////////
      MYSQL *con = mysql_init(NULL);

      if (con == NULL)
      {
      fprintf(stderr, "%s\n", mysql_error(con));
      exit(1);
      }

      if (mysql_real_connect(con, "localhost", "user12", "34klq*",
              "testdb", 0, NULL, 0) == NULL)
      {
      finish_with_error(con);
      }
      if (mysql_query(con, "DROP TABLE IF EXISTS DR14_Nacelle")) {
        finish_with_error(con);
     }
      //Cria TABELA DR14 para adição de variáveis
      if (mysql_query(con, "CREATE TABLE DR14_Nacelle(id INT,Variable
TEXT,Value FLOAT)")) {
        finish_with_error(con);
      }
```

```
    /* Data exchange loop */
    i = 0;
    while (1) {
            //Tratamento inicial da comunicação
            if(first){
            first = NULL;
            buffer[0] = 'S';
            //ENVIA requisição de status para DSP
            bytes_sent = send(sockfd, buffer, buffSize, 0);
        if (bytes_sent != buffSize) {
            printf("[id %d] stopping test. send returned %d\n", id,
bytes_sent);
            status = EXIT_FAILURE;
            goto QUIT;
        }
            //RECEBE status do DSP
            bytes_read = recv(sockfd, &buffer[0], buffSize, 0);
            //Sleep
            #if defined(__GNUC__) && defined(linux)
        usleep(sleepTime);
            #else
        Sleep(sleepTime / 1000);
            #endif
            }
          // Se status for 'O' de OK , enviar requisição para troca de dados
          else if (buffer[0] == 'O'){

            buffer[0] = 'D';
        //ENVIA requisição 'D' para troca de Dados
        bytes_sent = send(sockfd, buffer, buffSize, 0);
        if (bytes_sent != buffSize) {
            printf("[id %d] stopping test. send returned %d\n", id,
bytes_sent);
            status = EXIT_FAILURE;
            goto QUIT;
        }
         //RECEBE dados e variáveis do DSP
        total_bytes_read = 0;
        while (total_bytes_read < buffSize) {

            bytes_read = recv(sockfd, &buffer[total_bytes_read], buffSize -
total_bytes_read, 0);

            if (bytes_read <= 0) {
                printf("[id %d] stopping test. recv returned %d\n", id,
bytes_read);
                status = EXIT_FAILURE;
                goto QUIT;
            }
            total_bytes_read += bytes_read;
        }
printf("[id %d] count = %d, time = %ld, DATA>  ", id, count, time(NULL) -
start);
//Armazena buffer TCP nas variáveis locais do tipo int da struct union
            //Vac
                for(k=0; k<10; k++){
                str[k]=buffer[k+1];
                }
                str[10]='\0';
```

```
    vac.i = atoi(str);
//Vbc
    for(k=0; k<10; k++){
    str[k]=buffer[k+11];
    }
    str[10]='\0';
    vbc.i = atoi(str);
//Vca
    for(k=0; k<10; k++){
    str[k]=buffer[k+21];
    }
    str[10]='\0';
    vca.i = atoi(str);
//Vcc_batt
    for(k=0; k<10; k++){
    str[k]=buffer[k+31];
    }
    str[10]='\0';
    vcc_batt.i = atoi(str);
//Vcc
    for(k=0; k<10; k++){
    str[k]=buffer[k+41];
    }
    str[10]='\0';
    vcc.i = atoi(str);
//Icc
    for(k=0; k<10; k++){
    str[k]=buffer[k+51];
    }
    str[10]='\0';
    icc.i = atoi(str);
//Pcc
    for(k=0; k<10; k++){
    str[k]=buffer[k+61];
    }
    str[10]='\0';
    pcc.i = atoi(str);
//Wind speed
    for(k=0; k<10; k++){
    str[k]=buffer[k+71];
    }
    str[10]='\0';
    wind_speed.i = atoi(str);
//Wind Direction
    for(k=0; k<10; k++){
    str[k]=buffer[k+81];
    }
    str[10]='\0';
    wind_direc.i = atoi(str);
//YAW position
    for(k=0; k<10; k++){
    str[k]=buffer[k+91];
    }
    str[10]='\0';
    yaw_pos.i = atoi(str);
//Fim de curso YAW
    for(k=0; k<10; k++){
    str[k]=buffer[k+101];
    }
    str[10]='\0';
```

```
                fim_curso_yaw.i = atoi(str);
            //Fim de curso Motor de passo
                for(k=0; k<10; k++){
                str[k]=buffer[k+111];
                }
                str[10]='\0';
                fim_curso_passo.i = atoi(str);
            //Torque
                for(k=0; k<10; k++){
                str[k]=buffer[k+121];
                }
                str[10]='\0';
                torque.i = atoi(str);
            //Encoder
                for(k=0; k<10; k++){
                str[k]=buffer[k+131];
                }
                str[10]='\0';
                encoder.i = atoi(str);
            //PTC temperature
                for(k=0; k<10; k++){
                str[k]=buffer[k+141];
                }
                str[10]='\0';
                ptc_temp.i = atoi(str);
            //I2C temperature
                for(k=0; k<10; k++){
                str[k]=buffer[k+151];
                }
                str[10]='\0';
                i2c_temp.i = atoi(str);
            //KTY84 temperature
                for(k=0; k<10; k++){
                str[k]=buffer[k+161];
                }
                str[10]='\0';
                kty84_temp.i = atoi(str);
            //Heatsink temperature
                for(k=0; k<10; k++){
                str[k]=buffer[k+171];
                }
                str[10]='\0';
                heatsink_temp.i = atoi(str);
            //Freio Nacelle
                for(k=0; k<10; k++){
                str[k]=buffer[k+181];
                }
                str[10]='\0';
                freio_nacelle.i = atoi(str);
            //Tempo (timestamp)
                for(k=0; k<10; k++){
                str[k]=buffer[k+191];
                }
                str[10]='\0';
                tempo.i = atoi(str);
//Insere dados no formato float no Banco de dados
            snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f'
WHERE Id = 1", vac.f);
            mysql_query(con, statement);
```

```
        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 2", vbc.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 3", vca.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 4", vcc_batt.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 5", vcc.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 6", icc.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 7", pcc.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 8", wind_speed.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 9", wind_direc.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 10", yaw_pos.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 11", fim_curso_yaw.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 12", fim_curso_passo.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 13", torque.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 14", encoder.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 15", ptc_temp.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 16", i2c_temp.f);
        mysql_query(con, statement);
```

```
        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 17", kty84_temp.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 18", heatsink_temp.f);
        mysql_query(con, statement);

        snprintf(statement, 512, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 19", freio_nacelle.f);
        mysql_query(con, statement);

        snprintf(statement, 1024, "UPDATE DR14_Nacelle SET Value = '%f' WHERE
Id = 20", tempo.f);
        mysql_query(con, statement);
        /* Sleep specified time */
            #if defined(__GNUC__) && defined(linux)
        usleep(sleepTime);
            #else
        Sleep(sleepTime / 1000);
            #endif
            }
            else{  //Se status nao for 'O' de Ok espera status ficar Ok
            printf("> waiting status OK \n");
            buffer[0] = 'S';
            //ENVIA requisição de status
            bytes_sent = send(sockfd, buffer, buffSize, 0);
        if (bytes_sent != buffSize) {
            printf("[id  %d]  stopping  test.  send  returned  %d\n",  id,
bytes_sent);
            status = EXIT_FAILURE;
            goto QUIT;
        }
            //Recebe status do DSP
            bytes_read = recv(sockfd, &buffer[0], buffSize, 0);
            //Sleep
            #if defined(__GNUC__) && defined(linux)
        usleep(sleepTime);
            #else
        Sleep(sleepTime / 1000);
            #endif
            }
    }
QUIT:
    /* clean up */
    if (sockfd) {
        closesocket(sockfd);
    }
    if (results) {
        freeaddrinfo(results);
    }
    socketsShutdown();
    if (buffer) {
        free(buffer);
    }
    //Fecha conexão com o banco de dados
    mysql_close(con);
    return (status);
}
```

ANEXO C

Código do DSP TMS320F28335 (ADC, PWM, GPIO e *MessageQ*)

```
//////////////////////////////////////////////////////////////////////////
// UNIVERSIDADE FEDERAL DE SANTA MARIA-UFSM
// GRUPO DE ELETRONICA DE POTENCIA E CONTROLE-GEPOC
// PROJETO EÓLICA 2014
//////////////////////////////////////////////////////////////////////////
// TCC-IMPLEMENTAÇÃO DE INTERFACE DE COMUNICAÇÃO SEGUINDO IEC 61400-25
// PARA AEROGERADORES
// Código comunicação TCP/IP Raspberry Pi
// AUTOR: Juliano Grigulo E-MAIL: jgrigulo@gmail.com
//////////////////////////////////////////////////////////////////////////
// DESCRIÇÃO:
// Este código realiza:
// > Comunicação com MCU CORTEX-M3 do Concerto via TCP/IP
// > Leitura de variáveis do DSP e postagem no Banco de Dados DR14
//////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////
// UNIVERSIDADE FEDERAL DE SANTA MARIA-UFSM
// GRUPO DE ELETRONICA DE POTENCIA E CONTROLE-GEPOC
// PROJETO EÓLICA 2014
//////////////////////////////////////////////////////////////////////////
// TCC - IMPLEMENTAÇÃO DE INTERFACE DE COMUNICAÇÃO SEGUINDO NORMA IEC 61400-25
// PARA AEROGERADORES
// Código referente ao MCU F28335 do DSP Concerto F28M36x
// AUTOR: Juliano Grigulo E-MAIL: jgrigulo@gmail.com
//////////////////////////////////////////////////////////////////////////
// DESCRIÇÃO:
// Este código realiza:
// > Parametrização dos módulos de conversão analógica/digital do F28335 e do
módulo PWM1 disparando a conversão AD por meio do módulo ePWM.
// > Aquisição de dados da turbina eólica e envio/comunicação para Cortex-M3
// > Pisca um LED de um GPIO dentro da rotina de PWM.
//////////////////////////////////////////////////////////////////////////
/* Arquivos de cabeçalho XDCtools */
#include <xdc/std.h>
#include <xdc/runtime/Log.h>
#include <xdc/runtime/IHeap.h>
#include <xdc/runtime/Memory.h>
#include <xdc/runtime/Error.h>
#include <xdc/runtime/System.h>
#include <xdc/cfg/global.h>
/* Arquivos de cabeçalho IPC */
#include <ti/ipc/MessageQ.h>
#include <ti/ipc/MultiProc.h>
/* Arquivos de cabeçalho BIOS  */
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/heaps/HeapBuf.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Semaphore.h>
#include <string.h>
#include "demo.h"
```

```
#include "DSP28x_Project.h"// Device Headerfile and Examples Include File
#define BUF_SIZE   100  // Sample buffer size
#define  DUTY_CYCLE_A   32767
#define  DUTY_CYCLE_B   32767
#define  PERIOD         65535 // Período do PWM
// Configure which ePWM timer interrupts are enabled at the PIE level:
// 1 = enabled,  0 = disabled
#define PWM1_INT_ENABLE  1
Void adc_fxn(UArg arg);
// Global variables used in this example
UInt16  adcResult;
UInt16  buffer[BUF_SIZE];
Uint32  EPwm1TimerIntCount, interrupcao = 0, count_led = 0;
// Variáveis transmitidas ao SCADA
float P_grid = 18000.0 , Q_grid = 2000.0, S_grid = 20000.0, V_Rgrid = 380.0,
V_Sgrid = 380.0, V_Tgrid = 380.0, Vcc = 800.0, Status_inv = 1.0, Status_grid =
1.0, Temp_inv = 80.0, time_ref = 10122014.093500;
// Rotina de comunicação com Cortex-M3 por meio do módulo MESSAGE_Q
/*  ======== tsk0_func ========
 *  Allocates a message and ping-pongs the message around the processors.
 *  A local message queue is created and a remote message queue is opened.
 *  Messages are sent to the remote message queue and retrieved from the
 *  local MessageQ. */
Void tsk0_func(UArg arg0, UArg arg1)
{
    MessageQ_Msg     msg;
    MessageQ_Handle  messageQ;
    MessageQ_QueueId remoteQueueId;
    int              status;
    //unsigned short   msgId = 0;
    Ptr              buf;
    HeapBuf_Handle   heapHandle;
    HeapBuf_Params   hbparams;
    SizeT            blockSize;
    unsigned int     numBlocks;
    Error_Block      eb;

    /* Compute the blockSize & numBlocks for the HeapBuf */
    numBlocks = 8;
    blockSize = sizeof(TempMsg);

    /* Alloc a buffer from the default heap */
    buf = Memory_alloc(0, numBlocks * blockSize, 0, NULL);

    /* Create the heap that is used for allocating MessageQ messages. */
    Error_init(&eb);
    HeapBuf_Params_init(&hbparams);
    hbparams.align          = 0;
    hbparams.numBlocks      = numBlocks;
    hbparams.blockSize      = blockSize;
    hbparams.bufSize        = numBlocks * blockSize;
    hbparams.buf            = buf;
    heapHandle = HeapBuf_create(&hbparams, &eb);
    if (heapHandle == NULL) {
        System_abort("HeapBuf_create failed\n" );
    }

    /* Register default system heap with MessageQ */
```

```
    MessageQ_registerHeap((IHeap_Handle)(heapHandle), HEAPID);

    /* Create the local message queue */
    messageQ = MessageQ_create(C28QUEUENAME, NULL);
    if (messageQ == NULL) {
        System_abort("MessageQ_create failed\n" );
    }

    /* Open the remote message queue. Spin until it is ready. */
    do {
        status = MessageQ_open(M3QUEUENAME, &remoteQueueId);
         /*Sleep for 1 clock tick to avoid inundating remote processor
         *  with interrupts if open failed         */
        if (status < 0) {
            Task_sleep(1);
        }
    } while (status < 0);
     /*Wait for a message from the M3 processor and
     *  send it back after converting to Fahrenheit.     */
    while (1) {
        /* Get a message */
    status = MessageQ_get(messageQ, &msg, MessageQ_FOREVER);
    if (status < 0) {
       System_abort("This should not happen since timeout is forever\n");
    }
       /* Envia mensagem para o processador Remoto (Cortex-M3)*/
       ((TempMsg *)msg)->P_grid =  P_grid;
       ((TempMsg *)msg)->Q_grid = Q_grid;
       ((TempMsg *)msg)->S_grid = S_grid;
       ((TempMsg *)msg)->V_Rgrid = V_Rgrid;
       ((TempMsg *)msg)->V_Sgrid = V_Sgrid;
       ((TempMsg *)msg)->V_Tgrid = V_Tgrid;
       ((TempMsg *)msg)->Vcc = Vcc;
       ((TempMsg *)msg)->Status_inv = Status_inv;
       ((TempMsg *)msg)->Status_grid = Status_grid;
       ((TempMsg *)msg)->Temp_inv = Temp_inv;
       ((TempMsg *)msg)->time_ref = time_ref;
        status = MessageQ_put(remoteQueueId, msg);
        if (status < 0) {
           System_abort("MessageQ_put had a failure/error\n");
        }
    }
}
// Função MAIN que executa inicialização do DSP, parametrização do ADC e PWM e
aquisição
// de dados
/* ======== main ========*/
Void main()
{
    // Initialize System Control for Control and Analog Subsytems
    // Enable Peripheral Clocks
    InitSysCtrl();

    // Step 2. Initialize GPIO:
    InitGpio();


    EALLOW;
```

```
    //LED's Control Card
    GpioG1CtrlRegs.GPADIR.bit.GPIO31  = 1;  //Set as output  //LEDs
    GpioG1CtrlRegs.GPAMUX2.bit.GPIO31 = 0;
    GpioG1CtrlRegs.GPADIR.bit.GPIO0  = 1;  //Set as output  PWM
    GpioG1CtrlRegs.GPAMUX1.bit.GPIO0 = 1 ;
    EDIS;
    GpioG1DataRegs.GPADAT.bit.GPIO31 = 1;

    // Copy time critical code and Flash setup code to RAM
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);

    // Call Flash Initialization to setup flash waitstates
    InitFlash();

    // Initialize all the Device Peripherals:
    InitAdc1();    // Initialize ADC1 module

    EALLOW;
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;      // Stop all the TB clocks
    EDIS;
    EALLOW;
    SysCtrlRegs.PCLKCR1.bit.EPWM1ENCLK = 1;  // ePWM1
    SysCtrlRegs.PCLKCR1.bit.EPWM2ENCLK = 1;  // ePWM2
    SysCtrlRegs.PCLKCR1.bit.EPWM3ENCLK = 1;  // ePWM3
    SysCtrlRegs.PCLKCR1.bit.EPWM4ENCLK = 1;  // ePWM4
    SysCtrlRegs.PCLKCR1.bit.EPWM5ENCLK = 1;  // ePWM5
    SysCtrlRegs.PCLKCR1.bit.EPWM6ENCLK = 1;  // ePWM6
    EDIS;
    EALLOW;
    //Assumes ePWM1 clock is already enabled in InitSysCtrl();
    //Set event triggers (SOCA) for ADC SOC1
    EPwm1Regs.ETSEL.bit.SOCAEN  = 1;          // Enable SOC on A group
    EPwm1Regs.ETSEL.bit.SOCASEL = ET_CTR_PRDZERO;    // Select SOC from CMPA on
upcount
    EPwm1Regs.ETPS.bit.SOCAPRD  = ET_1ST;        // Generate pulse on every 1st
event
    //Time-base registers
    EPwm1Regs.TBPRD = PERIOD;                     // Set timer period, PWM
frequency = 1/period
    EPwm1Regs.TBPHS.half.TBPHS = 0;               // Time-Base Phase Register
    EPwm1Regs.TBCTL.bit.PRDLD = TB_SHADOW;      // Set Immediate load
    EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;   // Count-up mode: used for
asymmetric PWM
    EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE;       // Disable phase loading
    EPwm1Regs.TBCTL.bit.SYNCOSEL = TB_CTR_ZERO;
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;
    EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV1;

    //Setup shadow register load on ZERO
    EPwm1Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;
    EPwm1Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;
    EPwm1Regs.CMPCTL.bit.LOADAMODE = CC_CTR_ZERO;   // load on CTR=Zero
    EPwm1Regs.CMPCTL.bit.LOADBMODE = CC_CTR_ZERO;   // load on CTR=Zero

    //Set actions
    EPwm1Regs.AQCTLA.bit.CAU = AQ_CLEAR;     // set actions for EPWM1A
    //EPwm1Regs.AQCTLA.bit.CAD = AQ_CLEAR;
    EPwm1Regs.AQCTLA.bit.CAD = AQ_SET;
```

```
    //Configure Dead Time
    EPwm1Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE;   // enable Dead-band module
    EPwm1Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;    // Active Hi complementary
    EPwm1Regs.DBFED = 50;         // FED = 50 TBCLKs
    EPwm1Regs.DBRED = 50;        // RED = 50 TBCLKs

    EPwm1Regs.CMPA.half.CMPA = DUTY_CYCLE_A;      // Set duty 50% initially

    EDIS;

    EALLOW;
    SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 1;         // Start all the timers synced
    EDIS;

    EALLOW;

    // Configure ADC

    Adc1Regs.ADCCTL2.bit.ADCNONOVERLAP = 1;     // Enable non-overlap mode i.e.
conversion and future sampling events dont overlap
    Adc1Regs.ADCCTL1.bit.INTPULSEPOS   = 1;     // ADCINT1 trips after AdcResults
latch
    Adc1Regs.INTSEL1N2.bit.INT1E       = 1;     // Enabled ADCINT1
    Adc1Regs.INTSEL1N2.bit.INT1CONT    = 0;     // Disable ADCINT1 Continuous mode
    Adc1Regs.INTSEL1N2.bit.INT1SEL     = 0;     // setup EOC0 to trigger ADCINT1
to fire
    Adc1Regs.ADCSAMPLEMODE.bit.SIMULEN0 = 1;    // Simultaneous sampling enable
for SOC0/SOC1
    Adc1Regs.ADCSAMPLEMODE.bit.SIMULEN2 = 1;    // Simultaneous sampling enable
for SOC2/SOC3
    Adc1Regs.ADCSAMPLEMODE.bit.SIMULEN4 = 1;    // Simultaneous sampling enable
for SOC4/SOC5
    Adc1Regs.ADCSAMPLEMODE.bit.SIMULEN6 = 1;    // Simultaneous sampling enable
for SOC6/SOC7
    Adc1Regs.ADCSAMPLEMODE.bit.SIMULEN8 = 1;    // Simultaneous sampling enable
for SOC8/SOC9
    Adc1Regs.ADCSAMPLEMODE.bit.SIMULEN10 = 1;   // Simultaneous sampling enable
for SOC10/SOC11
    //Setting up the trigger source
    AnalogSysctrlRegs.TRIG1SEL.all  = 5;      // Assigning EPWM1SOCA to ADC
TRIGGER 1 of the ADC module
    Adc1Regs.ADCSOC0CTL.bit.CHSEL = 0;      // set SOC0 channel select to
ADCINA0/ADCINB0 pair
    Adc1Regs.ADCSOC0CTL.bit.TRIGSEL = 5;      // Set SOC0 start trigger to ADC
Trigger 1(EPWM1 SOCA) of the adc
    Adc1Regs.ADCSOC0CTL.bit.ACQPS = 6;       // set SOC0 S/H Window to 7 ADC Clock
Cycles, (6 ACQPS plus 1)
    Adc1Regs.ADCSOC2CTL.bit.CHSEL = 2;      // set SOC2 channel select to
ADCINA2/ADCINB2 pair
    Adc1Regs.ADCSOC2CTL.bit.TRIGSEL = 5;      // set SOC2 start trigger to ADC
Trigger 1(EPWM1 SOCA) of the adc
    Adc1Regs.ADCSOC2CTL.bit.ACQPS = 6;       // set SOC2 S/H Window to 7 ADC Clock
Cycles, (6 ACQPS plus 1)
    Adc1Regs.ADCSOC4CTL.bit.CHSEL = 3;    // set SOC4 channel select to
ADCINA3/ADCINB3 pair
    Adc1Regs.ADCSOC4CTL.bit.TRIGSEL = 5;      // set SOC4 start trigger to ADC
Trigger 1(EPWM1 SOCA) of the adc
```

```
    Adc1Regs.ADCSOC4CTL.bit.ACQPS = 6;       // set SOC4 S/H Window to 7 ADC Clock
Cycles, (6 ACQPS plus 1)
    Adc1Regs.ADCSOC6CTL.bit.CHSEL = 4;    // set SOC6 channel select to
ADCINA4/ADCINB4 pair
    Adc1Regs.ADCSOC6CTL.bit.TRIGSEL = 5;      // set SOC6 start trigger to ADC
Trigger 1(EPWM1 SOCA) of the adc
    Adc1Regs.ADCSOC6CTL.bit.ACQPS = 6;       // set SOC6 S/H Window to 7 ADC Clock
Cycles, (6 ACQPS plus 1)
    Adc1Regs.ADCSOC9CTL.bit.CHSEL = 6;    // set SOC8 channel select to
ADCINA6/ADCINB6 pair
    Adc1Regs.ADCSOC9CTL.bit.TRIGSEL = 5;      // set SOC8 start trigger to ADC
Trigger 1(EPWM1 SOCA) of the adc
    Adc1Regs.ADCSOC9CTL.bit.ACQPS = 6;       // set SOC8 S/H Window to 7 ADC Clock
Cycles, (6 ACQPS plus 1)
    Adc1Regs.ADCSOC11CTL.bit.CHSEL = 7;    // set SOC10 channel select to
ADCINA6/ADCINB6 pair
    Adc1Regs.ADCSOC11CTL.bit.TRIGSEL = 5;      // set SOC10 start trigger to ADC
Trigger 1(EPWM1 SOCA) of the adc
    Adc1Regs.ADCSOC11CTL.bit.ACQPS = 6;       // set SOC10 S/H Window to 7 ADC
Clock Cycles, (6 ACQPS plus 1)
    EDIS;

    BIOS_start();    // does not return
}

// FUNÇÃO disparada pelo módulo PWM1 que realiza a leitura dos ADs e acionamento
do LED
Void adc_fxn(UArg arg)
{
    P_grid = ((float)(Adc1Result.ADCRESULT0)*25000/4095); // Leitura canal 1 do
ADC1
    Q_grid = ((float)(Adc1Result.ADCRESULT1)*5000/4095);
    V_Rgrid= ((float)(Adc1Result.ADCRESULT2)*450/4095);
    V_Sgrid = ((float)(Adc1Result.ADCRESULT3)*450/4095);
    Vcc = ((float)(Adc1Result.ADCRESULT4)*1000/4095);
    Temp_inv = ((float)(Adc1Result.ADCRESULT5)*200/4095);
             // LIGA LED
             //
         GpioG1DataRegs.GPADAT.bit.GPIO31 = 0;
         count_led++; // temporizador de acionamento do LED
         if (count_led < 10000) GpioG1DataRegs.GPADAT.bit.GPIO31 = 0;
         if (count_led > 10000 && count_led < 20000)
GpioG1DataRegs.GPADAT.bit.GPIO31 = 1;
        if (count_led > 20000 ) count_led = 0;
             // DESLIGA LED
             //
          GpioG1DataRegs.GPADAT.bit.GPIO31 = 1;

          interrupcao ++; // variável de controle
          Adc1Regs.ADCINTFLGCLR.bit.ADCINT1 = 1;  //Clear ADCINT1 flag reinitialize
                                                  // for next SOC

}

//TODO
```

ANEXO D

Código da CPU Cortex-M3 (TCP/IP, *MessageQ* e GPIO)

```
//////////////////////////////////////////////////////////////////////////////
// UNIVERSIDADE FEDERAL DE SANTA MARIA-UFSM
// GRUPO DE ELETRONICA DE POTENCIA E CONTROLE-GEPOC
// PROJETO EÓLICA 2014
//////////////////////////////////////////////////////////////////////////////
// TCC - IMPLEMENTAÇÃO DE INTERFACE DE COMUNICAÇÃO SEGUINDO NORMA IEC 61400-25
// PARA AEROGERADORES
// Código referente ao MCU Cortex-M3 do DSP Concerto F28M36x
// AUTOR: Juliano Grigulo E-MAIL: jgrigulo@gmail.com
//////////////////////////////////////////////////////////////////////////////
// DESCRIÇÃO:
// Este código realiza:
// > Comunicação com MCU F28335 via MESSAGEQ
// > Comunicação com Raspberry Pi via TCP/IP
//////////////////////////////////////////////////////////////////////////////
/* XDCtools Header files */
#include <xdc/std.h>
#include <xdc/cfg/global.h>
#include <xdc/runtime/Diags.h>
#include <xdc/runtime/Error.h>
#include <xdc/runtime/IHeap.h>
#include <xdc/runtime/Log.h>
#include <xdc/runtime/Memory.h>
#include <xdc/runtime/System.h>

/* IPC Header files */
#include <ti/ipc/MessageQ.h>
#include <ti/ipc/MultiProc.h>

/* BIOS Header files */
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/heaps/HeapBuf.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Clock.h>


/* TI-RTOS Header files */
#include <ti/drivers/SDSPI.h>
#include <ti/drivers/I2C.h>
#include <ti/drivers/GPIO.h>


 /* NDK Header files */
#include <ti/ndk/inc/netmain.h>
#include <ti/ndk/inc/_stack.h>

/* Example/Board Header file */
#include "TMDXDOCK28M36.h"

#include "demo.h"
#include "USBCDCD_LoggerIdle.h"
```

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <file.h>

//setup M3

#include "inc/hw_sysctl.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_nvic.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "board_drivers/set_pinout_f28m36x.h"
#include "driverlib/ipc.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/debug.h"
#include "driverlib/cpu.c"
#include "driverlib/gpio.h"

#define TCPPACKETSIZE 1024  // Tamanho pacote TCP IP, 1024 BYTES
#define TCPPORT 1000 // Porta TCP:1000
#define NUMTCPWORKERS 3

extern void SerialInit();
extern void SerialMain();


// UNION STRUCT para representação de variáveis float em inteiro para comunicação
via TCP/IP
typedef union {
  int i;                                        u Wind_direc;
  float f;                                      u YAW_pos;
 } u;                                           u Fim_curso_YAW;
                                                u Fim_curso_Passo;
 // Variáveis referentes à DR-14                u Torque;
 u Vac;                                         u Encoder;
 u Vbc;                                         u PTC_temp;
 u Vca;                                         u I2C_temp;
 u Vcc_batt;                                    u KTY84_temp;
 u Vcc;                                         u Heatsink_temp;
 u Icc;                                         u Freio_Nacelle;
 u Pcc;                                         u time;
 u Wind_speed;

int V1=10 ;
int I1=20 ;
int recordingEnabled = RECORDING_CLOSED, l =0;
char statusDSP = 'N', comms = 'D', FLAG_BUF0;

//Função que realiza comunicação com MCU F28335 via MESSAGEQ
Void temperature_func(UArg arg0, UArg arg1)
{
    MessageQ_Msg     msg;
    MessageQ_Handle  messageQ;
    MessageQ_QueueId remoteQueueId;
    int              status;
```

```
    Ptr              buf;
    HeapBuf_Handle   heapHandle;
    HeapBuf_Params   hbparams;
    SizeT            blockSize;
    unsigned int     numBlocks;
    //unsigned int   i = 0;
    FILE            *dst;
    char             logBuffer[40];
    //SDSPI_Handle   sdspiHandle;
    Error_Block      eb;

    /* Compute the blockSize & numBlocks for the HeapBuf */
    numBlocks = 16; // Número de Blocos da árvore binária (buffer do MessageQ)
deve ser múltiplo de 8 (2^n)
    blockSize = sizeof(TempMsg);

    /* Alloc a buffer from the default heap */
    buf = Memory_alloc(0, numBlocks * blockSize, 0, NULL);
    / *  Create the heap that is used for allocating MessageQ messages.
    Error_init(&eb);
    HeapBuf_Params_init(&hbparams);
    hbparams.align          = 0;
    hbparams.numBlocks      = numBlocks;
    hbparams.blockSize      = blockSize;
    hbparams.bufSize        = numBlocks * blockSize;
    hbparams.buf            = buf;
    heapHandle = HeapBuf_create(&hbparams, &eb);
    if (heapHandle == NULL) {
        System_abort("HeapBuf_create failed\n" );
    }
   /* Register default system heap with MessageQ */
    MessageQ_registerHeap((IHeap_Handle)(heapHandle), HEAPID);

    /* Create the local message queue */
    messageQ = MessageQ_create(M3QUEUENAME, NULL);
    if (messageQ == NULL) {
        System_abort("MessageQ_create failed\n" );
    }
    /* Open the remote message queue. Spin until it is ready. */
    do {
        status = MessageQ_open(C28QUEUENAME, &remoteQueueId);
        if (status < 0) {
            Task_sleep(1);
        }
    } while (status < 0);
    /* Allocate a message to be ping-ponged around the processors */
    msg = MessageQ_alloc(HEAPID, sizeof(TempMsg));
    if (msg == NULL) {
       System_abort("MessageQ_alloc failed\n" );
    }

    MessageQ_setMsgId(msg, TEMPERATURE_CONVERSION);

     /*  Send the message to the remote processor and wait for a message
     *  from the previous processor.*/
    System_printf("Start the main loop\n");
    while (true) {
```

```
        Log_print0(Diags_USER1, "sending msg");
        /* send the message to the remote processor */
        status = MessageQ_put(remoteQueueId, msg);
        if (status < 0) {
           System_abort("MessageQ_put had a failure/error\n");
        }
        Log_print0(Diags_USER1, "getting msg");

        /* Get a message */
        status = MessageQ_get(messageQ, &msg, MessageQ_FOREVER);
        if (status < 0) {
           System_abort("This should not happen since timeout is forever\n");
        }
        Log_print0(Diags_USER1, "got msg");

        /* Recebe variáveis do buffer MessageQ e transfere às variáveis globais
float da UNION */

        Vac.f = ((TempMsg *)msg)->Vac;
        Vbc.f = ((TempMsg *)msg)->Vbc;
        Vca.f = ((TempMsg *)msg)->Vca;
        Vcc_batt.f = ((TempMsg *)msg)->Vcc_batt;
        Vcc.f = ((TempMsg *)msg)->Vcc;
        Icc.f = ((TempMsg *)msg)->Icc;
        Pcc.f = ((TempMsg *)msg)->Pcc;
        Wind_speed.f = ((TempMsg *)msg)->Wind_speed;
        Wind_direc.f = ((TempMsg *)msg)->Wind_direc ;
        YAW_pos.f = ((TempMsg *)msg)->YAW_pos;
        Fim_curso_YAW.f = ((TempMsg *)msg)->Fim_curso_YAW;
        Fim_curso_Passo.f = ((TempMsg *)msg)->Fim_curso_Passo ;
        Torque.f = ((TempMsg *)msg)->Torque ;
        Encoder.f = ((TempMsg *)msg)->Encoder ;
        PTC_temp.f = ((TempMsg *)msg)->PTC_temp;
        I2C_temp.f = ((TempMsg *)msg)->I2C_temp ;
        KTY84_temp.f = ((TempMsg *)msg)->KTY84_temp;
        Heatsink_temp.f = ((TempMsg *)msg)->Heatsink_temp ;
        Freio_Nacelle.f= ((TempMsg *)msg)->Freio_Nacelle;
        time.f = ((TempMsg *)msg)->time;

        Log_print0(Diags_USER1, "sleep");
        Task_sleep(200); // SLEEP por 200 clock ticks
        Log_print0(Diags_USER1, "awake");
    }
}
//Rotina que gerencia conexão TCP
Void tcpWorker(UArg arg0, UArg arg1)
{
    SOCKET clientfd = (SOCKET)arg0;
    int nbytes;
    bool flag = true;
    char *buffer;

    Error_Block eb;


    fdOpenSession(TaskSelf());

    System_printf("tcpWorker: start clientfd = 0x%x\n", clientfd);
```

```
/* Make sure Error_Block is initialized */
Error_init(&eb);

/* Get a buffer to receive incoming packets. Use the default heap. */
buffer = Memory_alloc(NULL, TCPPACKETSIZE, 0, &eb);
if (buffer == NULL) {
    System_printf("tcpWorker: failed to alloc memory\n");
    Task_exit();
}
/* Loop while we receive data */
while (flag) {
    nbytes = recv(clientfd, (char *)buffer, TCPPACKETSIZE, 0);

    //System_printf(" %c \n", (char *)buffer[0]);
    if (nbytes > 0) {

            if(buffer[0] == 'S'){

               System_sprintf(buffer, "%c", statusDSP);

               send(clientfd, (char *)buffer, nbytes, 0 );
               }
               else /*if (buffer[0] == 'D') */{ // if(buffer[0] = 'D'){/*
               /* Variáveis enviadas para Raspberry Pi, formato inteiro da
               UNION STRUCT */

               FLAG_BUF0 = buffer[0] ;
               comms = 'U';

               System_sprintf(buffer, "
               %c%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d", statusDSP,
               Vac.i ,
               Vbc.i ,
               Vca.i ,
               Vcc_batt.i ,
               Vcc.i ,
               Icc.i ,
               Pcc.i ,
               Wind_speed.i ,
               Wind_direc.i ,
               YAW_pos.i ,
               Fim_curso_YAW.i ,
               Fim_curso_Passo.i ,
               Torque.i ,
               Encoder.i ,
               PTC_temp.i ,
               I2C_temp.i ,
               KTY84_temp.i ,
               Heatsink_temp.i ,
               Freio_Nacelle.i ,
               time.i );

               send(clientfd, (char *)buffer, nbytes, 0 );

         } //else
    }
    else {
```

```
            fdClose(clientfd);
            flag = false;
        }
    }
    System_printf("tcpWorker stop clientfd = 0x%x\n", clientfd);

    /* Free the buffer back to the heap */
    Memory_free(NULL, buffer, TCPPACKETSIZE);

    fdCloseSession(TaskSelf());
    / *  Since deleteTerminatedTasks is set in the cfg file,
       the Task will be deleted when the idle task runs.     */
    Task_exit();
}

//Rotina que incia e cria conexões TCP,  sockets
Void tcpHandler(UArg arg0, UArg arg1)
{
    SOCKET lSocket;
    struct sockaddr_in sLocalAddr;
    SOCKET clientfd;
    struct sockaddr_in client_addr;
    int addrlen=sizeof(client_addr);
    int optval;
    int optlen = sizeof(optval);
    int status;
    Task_Handle taskHandle;
    Task_Params taskParams;
    Error_Block eb;

    fdOpenSession(TaskSelf());

    lSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (lSocket < 0) {
        System_printf("tcpHandler: socket failed\n");
        Task_exit();
        return;
    }

    memset((char *)&sLocalAddr, 0, sizeof(sLocalAddr));
    sLocalAddr.sin_family = AF_INET;
    sLocalAddr.sin_len = sizeof(sLocalAddr);
    sLocalAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    sLocalAddr.sin_port = htons(arg0);

    status = bind(lSocket, (struct sockaddr *)&sLocalAddr, sizeof(sLocalAddr));
    if (status < 0) {
        System_printf("tcpHandler: bind failed\n");
        fdClose(lSocket);
        Task_exit();
        return;
    }

    if (listen(lSocket, NUMTCPWORKERS) != 0){
        System_printf("tcpHandler: listen failed\n");
        fdClose(lSocket);
        Task_exit();
        return;
```

```
    }

    if (setsockopt(lSocket, SOL_SOCKET, SO_KEEPALIVE, &optval, optlen) < 0) {
        System_printf("tcpHandler: setsockopt failed\n");
        fdClose(lSocket);
        Task_exit();
        return;
    }

    while (true) {
        /* Wait for incoming request */
        clientfd = accept(lSocket, (struct sockaddr*)&client_addr, &addrlen);
        System_printf("tcpHandler: Creating thread clientfd = %d\n", clientfd);

        /* Init the Error_Block */
        Error_init(&eb);

        /* Initialize the defaults and set the parameters. */
        Task_Params_init(&taskParams);
        taskParams.arg0 = (UArg)clientfd;
        taskParams.stackSize = 1024;
        taskHandle = Task_create((Task_FuncPtr)tcpWorker, &taskParams, &eb);
        if (taskHandle == NULL) {
            System_printf("tcpHandler: Failed to create new Task\n");
        }
    }
}

// Código main, incializa módulos GPIO, UART, ETHERNET-MAC e LIBERA PORTAS GPIO
/*
 *  ======== main ========
 */
int main(void)
{
    Task_Handle taskHandle;
    Task_Params taskParams;
    Error_Block eb;

    /* Set up the board specific items */
    TMDXDOCK28M36_initGeneral();
    TMDXDOCK28M36_initUART();
    TMDXDOCK28M36_initGPIO();
    TMDXDOCK28M36_initSDSPI();
    TMDXDOCK28M36_initUSB(TMDXDOCK28M36_USBDEVICE);
    TMDXDOCK28M36_initEMAC();

    System_printf("Demo with HTTP, I2C, and SD\nSystem provider is set to SysMin,
halt the target and use ROV to view output.\n");
    /* SysMin will only print to the console when you call flush or exit */
    System_flush();

    /*  Create the Task that farms out incoming TCP connections.
    *  arg0 will be the port that this task listens to.
    */
    Task_Params_init(&taskParams);
    Error_init(&eb);
    taskParams.stackSize = 1024;
    taskParams.priority = 1;
```

```
    taskParams.arg0 = TCPPORT;
    taskHandle = Task_create((Task_FuncPtr)tcpHandler, &taskParams, &eb);
    if (taskHandle == NULL) {
        System_printf("main: Failed to create tcpHandler Task\n");
    }

    // Cortex -M3 CPU tem domínio sobre GPIOs do Concerto, portanto devemos
designar para o DSP quais
    // serão usadas pelo Cortex-M3 e quais serão usadas pelo F28335.
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOM);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOP);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOQ);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOR);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOS);
    GPIOPadConfigSet(GPIO_PORTA_BASE, 0xFF, GPIO_PIN_TYPE_STD_WPU);
    GPIOPadConfigSet(GPIO_PORTB_BASE, 0xFF, GPIO_PIN_TYPE_STD_WPU);
    // Give C28 control of all GPIOs
    GPIOPinConfigureCoreSelect(GPIO_PORTA_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTB_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTC_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTD_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTE_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTF_BASE, 0xFF, GPIO_PIN_M_CORE_SELECT);
//Nucleo M3 controla GPIO's da porta F
    GPIOPinConfigureCoreSelect(GPIO_PORTG_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTH_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTJ_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTK_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTL_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTM_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTN_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTP_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTQ_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTR_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT);
    GPIOPinConfigureCoreSelect(GPIO_PORTS_BASE, 0xFF, GPIO_PIN_C_CORE_SELECT)

    Start BIOS. Will not return from this call. */
    statusDSP = 'O';
    BIOS_start();
    return (0);
}
```