UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

GABRIEL SALES LINS RODRIGUES

Desenvolvimento de Interface de Usuário e API Embarcada para Avaliação de Desempenho de Transceiver RF com Tecnologia LoRa™

Campina Grande, Paraíba

Junho de 2016

GABRIEL SALES LINS RODRIGUES

Desenvolvimento de Interface de Usuário e API Embarcada para Avaliação de Desempenho de Transceiver RF com Tecnologia LoRa™

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Elétrica.

Área de Concentração: Sistemas Embarcados

Orientador: Prof. Alexandre Cunha Oliveira, D.Sc.

Campina Grande, Paraíba

Junho de 2016

GABRIEL SALES LINS RODRIGUES

Desenvolvimento de Interface de Usuário e API Embarcada para Avaliação de Desempenho de Transceiver RF com Tecnologia LoRa™

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Elétrica.

Aprovado em ____ /____ /____

Professor Avaliador
Universidade Federal de Campina Grande
Avaliador

Prof. Alexandre Cunha Oliveira, D.Sc.
Universidade Federal de Campina Grande
Orientador

Campina Grande, Paraíba
Junho de 2016

Dedico este trabalho aos meus pais.

Agradecimentos

A Deus, sobre todas as coisas, pois a Ele tudo devo e por tudo sou eternamente grato.

Aos meus pais, Marcos e Eulália, que sempre fizeram de um tudo para que nada me faltasse. Por todos os ensinamentos e todo apoio para que sempre seguisse meus sonhos. Não são vocês que devem se orgulhar de mim, mas eu de vocês.

A todos da minha família que sempre estiveram ao meu lado. Meu irmão Iago, minha avó Marlene, e meu tio Deja, que sempre serviu de exemplo para mim. Também aos meus avós, Severino, Maria e Antonio, que não puderam, em vida, ver onde cheguei.

A Fabrícia, que vive comigo cada uma das minhas angústias, que não desiste de mim mesmo nas minhas ausências e que é para mim alguém muito melhor do que eu sou para ela.

Aos meus grandes amigos de toda uma vida, a Nata.

À minha amiga Ana Paula, que sempre me ouve, me apoia e me socorre.

Aos meus amigos de graduação, Felipe Henrique, Luís Trovão, Lucas Henriques, Geraldo Landim, Felipe Pontes, Diogo Menezes, Renata Garcia e Luciana Joviniano. Sem vocês, não teria sido igual.

Àqueles que me auxiliaram neste trabalho de uma forma ou de outra. Os Professores Antonio Marcus Nogueira Lima e Marcos Ricardo Alcântara Morais, e o colega de laboratório Yago Monteiro.

Ao Professor Alexandre Cunha Oliveira, pela orientação deste Trabalho de Conclusão de Curso, por toda paciência e todos os ensinamentos que me passou durante toda a graduação, seja dentro ou fora de sala de aula.

Live as if you were to die tomorrow.
Learn as if you were to live forever.
(Mahatma Gandhi)

Resumo

A *Internet* foi uma das mais importantes invenções deste século. O que a princípio se restringia à tarefa de interligar computadores de alguns poucos, hoje é uma rede global que interliga bilhões de pessoas e pode ser considerada parte intrínseca da vida em sociedade. Além do acesso ter sido democratizado, o próprio conceito de *internet* se estendeu. A rede que interligava apenas computadores, hoje liga também telefones celulares, *tablets*, relógios. Em meio a essa extensão do conceito de *internet*, surgiu a Internet das Coisas (*Internet of Things* - IoT) que se propõe a conectar "coisas" entre si, sejam elas relógios ou uma máquina de vendas. Dada a dimensão que essas redes podem tomar, é imprescindível que se dedique bastante atenção à escolha da rede que irá de fato interconectar estes dispositivos. Para os casos sem fio, as redes de radiofrequência (RF) se mostram alternativas muito viáveis. No contexto deste trabalho, serão desenvolvidas ferramentas, tanto de interface de usuário quanto embarcadas, para utilização e avaliação de uma rede sem fio utilizando a tecnologia LoRa™, da Semtech Corporation, para aplicações de IoT.

Palavras-chave: *Internet of Things*. Redes sem Fio. Sistemas Embarcados. LoRa™.

Abstract

The Internet was one the most relevant inventions of the century. What, at first, was restricted to the task of connecting the computers of a few, nowadays is a global network that interconnects billions of people and can be considered an intrinsic part of life in society. Beyond the fact that it became more accessible, the very concept of internet has been extended. The network which connected only computers, now connects cellphones, tablets, watches. In between the conceptual extension of what is internet, emerged the Internet of Things (IoT) that proposed to interconnect things, whatever they are watches or vending machines. Given the dimension these networks may take, it is quite important to give some thought about the network that will indeed connect these devices. Regarding wireless cases, the radiofrequency (RF) networks seem really viable options. In the context of this work, some tools are going to be developed, such as user interfaces and embedded systems, to make use and evaluate a wireless network using the LoRa™ technology, from Semtech Corporation, in IoT applications.

Keywords: Internet of Things. Wireless Networks. Embedded Systems. LoRa™.

Lista de ilustrações

Figura 1 – Diagrama simplificado de blocos do *transceiver* SX1272 (SEMTECH CORPORATION, 2015). . . . 21
Figura 2 – *Buffer* de dados do modo LoRa^{TM} (SEMTECH CORPORATION, 2015). 22
Figura 3 – Estrutura de pacoe LoRa^{TM} (SEMTECH CORPORATION, 2015). . . 23
Figura 4 – Exemplo de DSSS (FOROUZAN, 2013). . . . 25
Figura 5 – MSP-EXP430G2 (TEXAS INSTRUMENTS, 2016b). . . . 27
Figura 6 – Diagrama da organização funcional da API. . . . 28
Figura 7 – Diagrama de tempo de acesso à interface SPI do *transceiver* SX1272 (SEMTECH CORPORATION, 2015). . . . 29
Figura 8 – Interface gráfica de usuário, assim que é inicializada. . . . 32
Figura 9 – Interface gráfica de usuário, após configuração e estabelecimento da comunicação serial. . . . 32
Figura 10 – Módulos de teste . . . 36
Figura 11 – Localização do nó transmissor (GOOGLE MAPS, 2016). . . . 37
Figura 12 – Distância de transmissor e receptor no primeiro ponto de testes (GOOGLE MAPS, 2016). . . . 38
Figura 13 – Distância de transmissor e receptor no segundo ponto de testes (GOOGLE MAPS, 2016). . . . 39
Figura 14 – Distância de transmissor e receptor no terceiro ponto de testes (GOOGLE MAPS, 2016). . . . 39
Figura 15 – Distância de transmissor e receptor onde cessou a comunicação do Si4468 durante o quarto teste (GOOGLE MAPS, 2016). . . . 40
Figura 16 – Distância de transmissor e receptor (SX1272) no quarto ponto de testes (GOOGLE MAPS, 2016). . . . 41
Figura 17 – Distância de transmissor e receptor onde cessou a comunicação do Si4468 durante o quinto teste (GOOGLE MAPS, 2016). . . . 41
Figura 18 – Distância de transmissor e receptor (SX1272) no quinto ponto de testes (GOOGLE MAPS, 2016). . . . 42
Figura 19 – Distância de transmissor e receptor no primeiro teste de máxima configuração (GOOGLE MAPS, 2016). . . . 43
Figura 20 – Distância de transmissor e receptor no segundo teste de máxima configuração (GOOGLE MAPS, 2016). . . . 43

Lista de tabelas

Tabela 1 – Estimativas de sensibilidade e taxa de transmissão 26

Lista de abreviaturas e siglas

ANATEL Agência Nacional de Telecomunicações

API Interface de Programação de Aplicações (*Application Programming Interface*)

ARPA *Advanced Research Projects Agency*

ARPANET *Advanced Research Projects Agency Network*

BW Largura de Banda (*Bandwidth*)

CR Taxa de Codificação (*Coding Rate*)

CRC Verificação de Redundância Cíclica *Cyclic Redundancy Check*

CSS *Chirp Spread Spectrum*

DSSS Espalhamento de Espectro por Sequência Direta (*Direct Sequence Spread Spectrum*)

EXE Arquivo Executável (*Executable File*)

FEC Correção de Erros a Posteriori *Forward Error Correction*

FIFO Primeiro a entrar, primeiro a sair *First In, First Out*

FSK Chaveamento de Deslocamento de Frequência (*Frequency-Shift Keying*)

GFSK Chaveamento Gaussiano de Deslocamento de Frequência (*Gaussian Frequency-Shift Keying*)

GMSK Chaveamento Gaussiano de Deslocamento Mínimo (*Gaussian Minimum-Shift Keying*)

GUI Interface Gráfica de Usuário (*Graphical User Interface*)

IDE Ambiente de Desenvolvimento Integrado (*Integrated Development Environment*)

IEEE *Institute of Electrical and Electronics Engineers*

ISM Industrial, Científica e Médica (*Industrial, Scientific, and Medical*)

IoT Internet das Coisas (*Internet of Things*)

LAN Rede Local (*Local Area Network*)

LoRaWAN™ Rede de Longa Distância LoRa™ (LoRa™ *Wide Area Network*)

LPWAN Rede de Longa Distância de Baixa Potência (*Low Power Wide Area Network*)

MISO *Master Input, Slave Output*

MOSI *Master Output, Slave Input*

MSK Chaveamento de Deslocamento Mínimo (*Minimum-Shift Keying*)

NSS *Slave Select.* A letra N representa a polaridade (nível baixo).

OOK Chaveamento Ligado-Desligado (*On-Off Keying*)

PaqTcPB Fundação Parque Tecnológico da Paraíba

P&G Procter & Gamble

RF Radiofrequência (*Radiofrequency*)

RFID Identificação por Radiofrequência (*Radiofrequency Identification*)

SCK *System Clock*

SF Fator de Espalhamento (*Spreading Factor*)

SPI *Serial Peripheral Interface*

SRI *Stanford Research Institute*

TCC Trabalho de Conclusão de Curso

TCXO Cristal Oscilador com Compensação de Temperatura (*Temperature Compensated Crytal Oscillator*)

TXT Arquivo de Texto (*Text File*)

UART Receptor/Transmissor Universal Assíncrono (*Universal Asynchronous Receiver/Transmitter*)

UCLA *University of California at Los Angeles*

UCSB *University of California at Santa Barbara*

USCI Interface de Comunicação Serial Universal (*Universal Serial Communication Interface*)

WAN	Rede de Longa Distância (*Wide Area Network*)
WNR	*Write, Not Read*

Sumário

1 INTRODUÇÃO . . . 15
1.1 Objetivos . . . 16
1.2 Organização do Texto . . . 16

2 *INTERNET OF THINGS* . . . 17
2.1 *Smart Grids* . . . 18
2.2 LPWAN . . . 18
2.2.1 LoRaWAN™ . . . 19

3 TECNOLOGIA LORA™ . . . 20
3.1 *Transceiver* SX1272 . . . 20
3.1.1 Modos de Operação . . . 22
3.2 Estrutura de Pacote . . . 23
3.2.1 Preâmbulo . . . 23
3.2.2 Cabeçalho . . . 24
3.2.3 *Payload* . . . 24
3.3 *Coding Rate* (CR) . . . 24
3.4 *Spreading Factor* (SF) . . . 25
3.5 Largura de Banda (BW) . . . 25
3.6 Frequência Central de Transmissão . . . 26
3.7 Sensibilidade e Taxa de Transmissão . . . 26

4 API EMBARCADA . . . 27
4.1 Comunicação UART do ponto de vista do MSP430 . . . 28
4.2 Comunicação SPI do ponto de vista do MSP430 . . . 28
4.3 Interface SPI com *transceiver* SX1272 . . . 29
4.4 Configuração e Operação do SX1272 . . . 30
4.5 Módulo de interface GUI-API . . . 30

5 INTERFACE GRÁFICA DE USUÁRIO . . . 31
5.1 Protocolo de Instruções . . . 31
5.2 Utilização da Interface . . . 31

6 PROCEDIMENTOS E RESULTADOS . . . 35
6.1 Metodologia Utilizada . . . 36
6.2 Resultados . . . 37
6.2.1 Testes Comparativos . . . 37

6.2.2 Teste de Alcance Máximo do SX1272 42

7 CONCLUSÃO 44

REFERÊNCIAS 45

APÊNDICES 47

APÊNDICE A – CÓDIGO DA API EMBARCADA 48

1 Introdução

Os primeiros passos do que viria a ser a *Internet* foram dados por volta da década de 1960. A ARPA, órgão do Departamento de Defesa dos Estados Unidos da América, tinha interesse em conectar os *mainframes*, que eram completamente independentes, dos pesquisadores que ela financiava, para que eles pudessem compartilhar suas descobertas e evitar esforços duplicados. Em 1969, nasceu a ARPANET que conectava quatro instituições de pesquisa: a UCLA (*University of California at Los Angeles*), a UCSB (*University of California at Santa Barbara*), o SRI (*Stanford Research Institute*), e a University of Utah (FOROUZAN, 2013). Hoje, a *Internet* conecta bilhões de dispositivos e, por consequência, as bilhões de pessoas por trás destes.

Há tempos a *Internet* deixou de conectar apenas computadores. Passou a fazer sentido conectar entre si muitos outros dispositivos que podem interagir, trazendo benefícios aos usuários. São telefones celulares, relógios, sensores do mais diversos tipos, automóveis, construções, e tantos outros. Essa extensão do conceito de *internet*, que não é exatamente nova, vem se tornando cada dia mais presente no dia a dia das pessoas.

As redes que conectam estes objetos físicos podem ter diversas configurações diferentes. Com fio, sem fio, LAN (*Local Area Network* - Rede Local), WAN (*Wide Area Network* - Rede de Longa Distância), Wi-Fi, *Bluetooth*, são algumas dentre uma infinidade de possibilidades. A escolha do tipo de rede é feita de acordo com a aplicação, e muitas aplicações de IoT se propõem a conectar dispositivos afastados uns dos outros, espalhados por ambientes não controlados (como ao longo de uma cidade, por exemplo). Nestes casos, há um favorecimento às redes sem fio e de baixo consumo de energia, já que seus dispositivos podem precisar funcionar a base de baterias. Estas são as chamadas LPWAN (*Low Power Wide Area Network* - Rede de Longa Distância de Baixa Potência).

Dentre as LPWAN, uma chama atenção atualmente: a LoRaWAN™ (LoRa™ *Wide Area Network* - Rede de Longa Distância LoRa™). As LoRaWAN™ são LPWAN que utilizam a tecnologia LoRa™ em seus *transceivers*. Com LoRa™, a Semtech Corporation promete comunicação sem fio, a uma baixa taxa de transmissão, para longas distâncias e com baixo consumo de corrente (SEMTECH CORPORATION, 2016b). Além disso, uma das vantagens citadas é que não é necessária a utilização de um oscilador sofisticado para que se consiga atingir as baixas taxas de transmissão que impactam diretamente no seu alcance, diminuindo o custo do projeto (SEMTECH CORPORATION, 2015).

1.1 Objetivos

Neste contexto, busca-se desenvolver ferramentas que possibilitem a utilização da tecnologia LoRa^{TM} e a avaliação do seu desempenho. Estas ferramentas são: uma API (*Application Programming Interface* - Interface de Programação de Aplicações) embarcada para microcontroladores da família MSP430, da Texas Instruments, para possibilitar a configuração e operação do *transceiver* SX1272, da Semtech Corporation, dotado da tecnologia LoRa^{TM}; uma GUI (*Graphical User Interface* - Interface Gráfica de Usuário) para computador que, por meio de comunicação serial, interage com o microcontrolador para configurar, operar e monitorar o *transceiver* de maneira simples e contínua. Estas ferramentas serão utilizadas também para operação do *transceiver* SX1272 durante testes comparativos com o *transceiver* Si4468, da Silicon Labs, que é um dos concorrentes da Semtech por espaço no mercado de IoT.

1.2 Organização do Texto

Este trabalho encontra-se dividido em 6 capítulos. No Capítulo 2 é realizada uma discussão geral acerca do tema *Internet of Things*, conceituando, citando um pouco do histórico e exemplificando aplicações. No Capítulo 3 é apresentada a tecnologia LoRa^{TM}, suas vantagens, características e parâmetros de configuração. Na sequência, no Capítulo 4 é descrita a API, suas funcionalidades, e a tarefa de cada um de seus módulos. O Capítulo 5 detalha o funcionamento da interface gráfica de usuário desenvolvida, como ela funciona, em que plataforma ela se baseia e como ela é operada. Além disso, são descritos os testes realizados e seus resultados. Por último, o Capítulo 7 traz as conclusões seguido das referências bibliográficas e os apêndices.

2 *Internet of Things*

A *Internet of Things*, como o próprio nome já sugere, define uma rede de "coisas". Estas "coisas" são objetos físicos dos mais diversos tipos, dotados de algum tipo de inteligência que os dá capacidade de serem "sentidos" e controlados (HARVARD BUSINESS REVIEW, 2014). As "coisas" podem ser automóveis, câmeras, máquinas de venda, medidores de energia, conectados em rede com o objetivo de aumentar a eficiência, comodidade, segurança e até a sustentabilidade de um sistema. Já a inteligência que lhes é dada se traduz na forma de *softwares*, sistemas embarcados, sensores e dispositivos que permitem a sua conexão em rede, como *transceivers*.

O conceito da IoT, apesar dos ares futuristas, não é novo. O próprio nome foi cunhado em 1999 pelo britânico Kevin Ashton (WOOD, 2015), durante uma apresentação sobre RFID (*Radiofrequency Identification* - Identificação por Radiofrequência) quando trabalhava para a P&G (ASHTON, 2009). Por mais de 20 anos, companhias têm utilizado sensores em rede para fornecer informação a respeito dos seus equipamentos, seu funcionamento, as condições do ambiente, sua localização. O que mudou nesse meio tempo para possibilitar a sofisticação e a expansão da área de aplicação dessas redes foram fatores como os avanços na computação em nuvem, nas redes sem fio, nos dispositivos móveis e na indústria de sistemas embarcados (HARVARD BUSINESS REVIEW, 2014).

A imensa maioria das aplicações em IoT, assim como é notório das aplicações em rede atualmente, dá preferência às redes sem fio. Este tipo de rede traz consigo grande flexibilidade principalmente no que diz respeito ao alcance e, consequentemente, à dimensão que uma rede pode ter. As desvantagens ficam por conta da maior complexidade, da necessidade de maior segurança, tanto para os equipamentos, quando para os dados, e o consumo de energia, já que muitas vezes os dispositivos serão alimentados por bateria.

Outro obstáculo à expansão inicial da IoT é a falta de padronização do serviço (HARVARD BUSINESS REVIEW, 2014). Muito vem se fazendo a esse respeito e é fundamental para o crescimento destas redes. A partir da padronização de um determinado tipo de rede, pessoas das mais diferentes partes do mundo podem desenvolver seus projetos de maneira independente, desde que de acordo com o padrão, e implementá-los de modo complementar, fazendo com que estejam em conformidade e funcionem bem em conjunto. Isso sem que nenhum das partes sequer considere ou conheça a outra durante seu projeto.

2.1 *Smart Grids*

Smart grids são um ótimo exemplo de tudo que a IoT representa e estão entre suas mais promissoras aplicações. Os benefícios e as possibilidades são uma infinidade. Um *smart grid* é uma rede elétrica evoluída, dotada de "inteligência", que administra a eletricidade de maneira mais eficiente, confiável e sustentável (ABB, 2015).

Os *smart grids* oferecem vantagens especialmente no monitoramento contínuo e a distância de características importantes da rede. Soluções na detecção de faltas de alta impedância, localização de faltas em geral e reestabelecimento do serviço de maneira rápida, são alguns exemplos disso. Para os clientes, existe a possibilidade da gerar energia na sua residência, integrá-la à rede e vendê-la à concessionária para abater o valor na sua conta (conhecida como Geração Distribuída); e, ainda, ter maior controle sobre os detalhes do seu consumo, em tempo real, fazendo uso dos Medidores Inteligentes (*Smart Meters*). Os *smart meters* já são alvo de pesquisa e desenvolvimento ao redor do mundo e alguns modelos, inclusive, já estão disponíveis no mercado.

O IEEE (*Institute of Electrical and Electronics Engineers*), maior associação de profissionais do mundo, também já possui uma iniciativa chamada IEEE *Smart Grid* na intenção de unificar, padronizar e promover os *smart grids* ao redor do mundo, além de levar o conhecimento sobre o tema a partir das suas publicações. A IEEE *Smart Grid* busca ser a voz da autoridade e credibilidade mundial no ramo de *smart grids* (IEEE SMART GRID, 2016).

2.2 LPWAN

As LPWAN são redes sem fio de grande alcance especializadas em conectar dispositivos de baixo consumo de energia. Em geral, os dispositivos dessas redes apresentam baixas taxas de transmissão de dados. Diferente de tecnologias como *Bluetooth*, *ZigBee* e Wi-Fi, mais adequadas para as aplicações de IoT de menor dimensão, as LPWAN são mais apropriadas para aplicações de escala industrial, civil e comercial (IOT AGENDA, 2016).

Exemplos de tecnologias LPWAN são:

- *Greenwaves*, da Greenwave Systems, que funciona a taxas de transmissão da ordem de Mbps
- *Haystack*, da Haystack Technologies, baseado no protocolo da DASH7 Alliance
- *Weightless*, da Weightless SIG
- LoRaWAN™, baseado na tecnologia LoRa™ da Semtech Corporation

2.2.1 LoRaWAN™

LoRaWAN™ é um tipo de LPWAN que utiliza a tecnologia LoRa™, da Semtech Corporation, para fornecer uma comunicação segura, bi-direcional, de longa distância, baixo custo e feita especificamente para dispositivos sem fio alimentados por bateria (LORA ALLIANCE, 2016c). Pela própria descrição da tecnologia, ela parece ter sido desenvolvida especificamente para aplicações de IoT: e foi. A Semtech deixa isso claro inclusive quando promove o LoRa™ (SEMTECH CORPORATION, 2016c). As LoRaWAN™ chamam atenção pelas organizações ao seu redor, como a LoRa™ Alliance, e projetos de grande porte já implementados, como o da The Things Network.

A LoRa™ Alliance é uma associação aberta e sem fins lucrativos, iniciada por líderes da indústria, tendo como objetivo a padronização e popularização das LPWAN, especificamente as que utilizam o protocolo LoRa™ (LoRaWAN™) (LORA ALLIANCE, 2016b). A LoRa™ Alliance oferece programas de certificação para projetos com intuito de garantir a conformidade do mesmo com os seus padrões (LORA ALLIANCE, 2016a).

The Things Network é, segundo eles próprio, um grupo de pessoas, colaboradores, com a intenção de criar uma rede de dados global, aberta da *Internet of Things* (THE THINGS NETWORK, 2016b). Eles implantaram sobre toda a cidade de Amsterdam, Holanda, uma rede da IoT, num intervalo de seis semanas. Afirmam também que o alcance da comunicação pode chegar a uma raio de 15 km — em ambientes sem, ou com muito poucos, obstáculos. Tudo foi feito por meio de colaboração (THE THINGS NETWORK, 2016a).

3 Tecnologia LoRa^{TM}

A tecnologia LoRa^{TM} (*Long Range* - Longo Alcance) é uma técnica de modulação, patenteada pela Semtech Corporation, que oferece grande alcance de comunicação e baixo consumo de corrente, fazendo uso de técnicas de espalhamento de espectro (*spread spectrum*) e uma variação de CSS (*Chirp Spread Spectrum*) com FEC (*Forward Error Correction* - Correção de Erros a Posteriori). A modulação LoRa^{TM} proporciona um aumento significativo da sensibilidade do receptor e, como outras técnicas de espalhamento de espectro, toda a largura de banda é utilizada para transmitir um sinal, fazendo com que ela possua maior imunidade ao ruído do canal e às possíveis variações de frequência causadas pela utilização de cristais osciladores de baixo custo. A modulação LoRa^{TM} compõe a camada física de uma rede e pode servir de base para várias arquiteturas —como *Mesh*, ou *Star* (SEMTECH CORPORATION, 2016a). A Semtech oferece uma família de dispositivos dispondo da tecnologia LoRa^{TM}. São eles os *transceivers* SX127x (2; 3; 6-9), e o concentrador SX1301. Neste trabalho, foi utilizado um dos modelos mais simples: o SX1272.

3.1 *Transceiver* SX1272

O *transceiver* SX1272, da Semtech Corporation, possui um modem LoRa^{TM}, além de dar suporte também às modulações FSK (*Frequency-Shift Keying* - Chaveamento de Deslocamento de Frequência), GFSK (*Gaussian Frequency-Shift Keying* - Chaveamento Gaussiano de Deslocamento de Frequência), MSK (*Minimum-Shift Keying* - Chaveamento de Deslocamento Mínimo), GMSK (*Gaussian Minimum-Shift Keying* - Chaveamento Gaussiano de Deslocamento Mínimo) e OOK (*On-Off Keying* - Chaveamento Ligado-Desligado). Ele conta com um amplificador de potência de $+20$ dBm e uma alta sensibilidade de até -137 dBm, a depender da sua configuração, enquanto o consumo de corrente é de, no máximo, cerca de 10 mA no lado receptor. A taxa de transmissão pode chegar a 300 kbps no seu máximo e 183 bps no seu mínimo (SEMTECH CORPORATION, 2015).

Os melhores resultados do *transceiver*, tendo como métrica o alcance e o consumo de corrente, ocorrem nas taxas de transmissão mais baixas. Uma das suas principais vantagens está justamente nesse ponto. Nos circuitos convencionais, é comum que quão mais baixa é a taxa de transmissão, mais preciso, e oneroso, é o cristal oscilador necessário para que funcione corretamente. Na modulação LoRa^{TM}, não. O SX1272 alcança essas taxas mais baixas com um cristal mais barato, de ± 10 ppm, considerando valores de largura de banda de $62,5$ kHz ou mais[1,2] (SEMTECH CORPORATION, 2016a), desta

[1] O menor valor de largura de banda configurável com o *transceiver* SX1272 é de 125 kHz
[2] Para valores de largura de banda inferiores, recomenda-se o uso de um TCXO

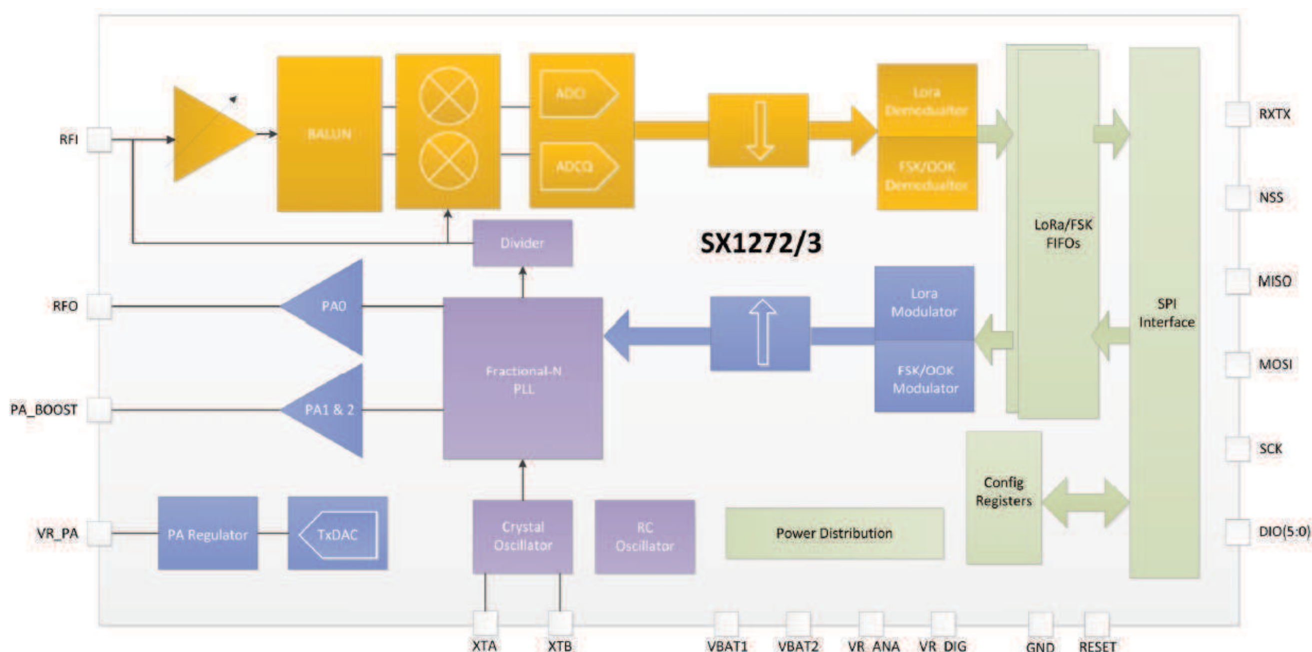


Figura 1 – Diagrama simplificado de blocos do *transceiver* SX1272 (SEMTECH CORPORATION, 2015).

maneira reduzindo o preço da montagem.

É também importante destacar a banda de frequência na qual o *transceiver* trabalha: a banda ISM (*Industrial, Scientific, and Medical* - Industrial, Científica e Médica). Esta faixa de espectro é reservada internacionalmente para aplicações de RF nas áreas, como o nome sugere, industrial, científica e médica. Portanto, ela é livre e não precisa ser negociada junto a qualquer órgão regulador como, no caso do Brasil, a ANATEL (Agência Nacional de Telecomunicações).

Toda a interface com o SX1272 é feita via comunicação SPI (*Serial Peripheral Interface*), como se vê no diagrama simplificado da Figura 1. Os registradores são palavras de 8 *bits* e são os mesmos, independente da modulação utilizada. O mapeamento desses registradores, no entanto, varia caso a modulação utilizada seja LoRa^{TM}ou alguma das outras disponíveis.

Para o modo LoRa^{TM}, existe um *buffer* de 256 *bytes* utilizado para transmissão e recepção. Este *buffer* é comumente dividido em dois (*buffer* de transmissão e recepção) de 128 *bytes*, mas isso fica a critério do usuário e é configurável. É possível alterar o endereço de base do *buffer* de transmissão e recepção e, por consequência, suas dimensões. A Figura 2 deixa isso mais claro.

Os principais parâmetros de configuração do *transceiver*, operando com modulação LoRa^{TM}, são: modo de cabeçalho; potência de transmissão; frequência central de transmissão; *spreading factor* (SF); *coding rate* (CR); largura de banda (BW); otimização de

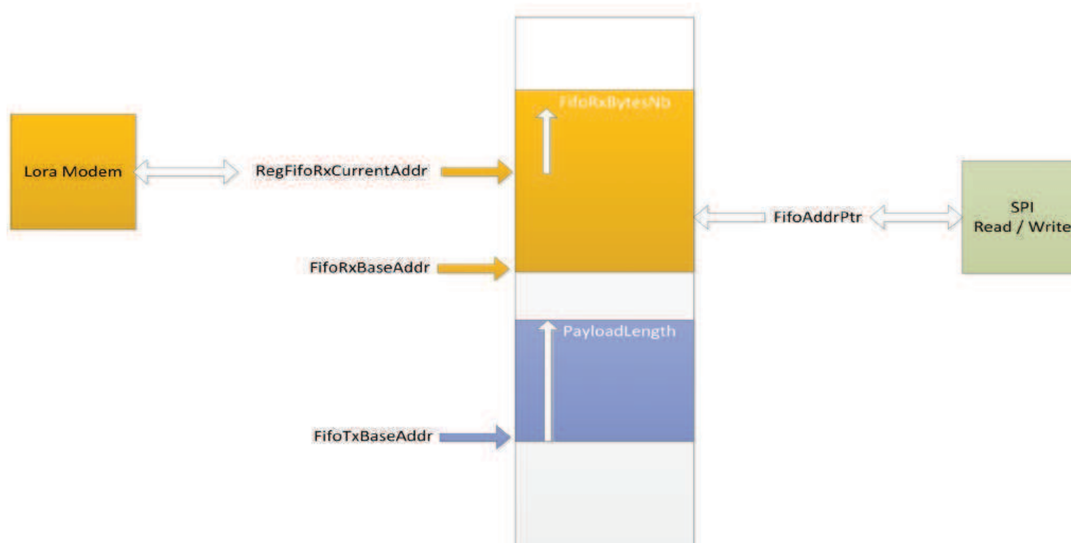


Figura 2 – *Buffer* de dados do modo LoRa™ (SEMTECH CORPORATION, 2015).

para baixa taxa de dados; e inclusão de CRC (*Cyclic Redundancy Check* - Verificação de Redundância Cíclica) no pacote a ser transmitido. Mais especificamente, planejando as configurações de SF, BW e CR, é possível otimizar a modulação LoRa™ para uma dada aplicação, balanceando fatores como ocupação de espectro, taxa de transmissão e imunidade a interferências. O impacto dessas e outras configurações (em termos de tempo do pacote "no ar", sensibilidade do receptor, taxa de transmissão, entre outros) pode ser calculado baseado nas informações encontradas no *datasheet* ou obtido por meio da LoRa™ *Calculator*, fornecida pela Semtech.

Nas seções a seguir, alguns aspectos da tecnologia LoRa™ serão melhor explicados. Todas essas seções serão baseadas na forma como funciona a tecnologia especificamente no *transceiver* SX1272, que foi o utilizado neste trabalho.

3.1.1 Modos de Operação

O SX1272 dispõe de oito modos de operação para a modulação LoRa™. São eles: SLEEP; STANDBY; FSTX; FSRX; TX; RXCONTINUOUS; RXSINGLE; CAD. Cinco deles foram considerados mais importantes e serão rapidamente descritos a seguir.

Mode SLEEP Modo de baixa potência onde apenas a interface SPI e os registradores de configuração estão acessíveis. Apenas nesse modo são permitidas trocas no modo de transmissão entre FSK/OOK e LoRa™.

Mode STANDBY Modo onde os blocos oscilador e LoRa™ estão ligados, mas nenhuma atividade de transmissão está ocorrendo. Apenas neste modo e no modo SLEEP são permitidas modificações nos registradores.

Modo TX Ao ativar este modo, o SX1272 transmite a quantidade de *bytes* determinados pelo registrador que configura o tamanho de pacote. O pacote se inicia no endereço do *buffer* configurado como base do transmissor. Ao finalizar a transmissão, o *transceiver* volta ao modo STANDBY.

Modo RXCONTINUOUS Neste modo, o SX1272 está contínua e ciclicamente buscando pacotes para receber.

Modo RXSINGLE Uma vez colocado neste modo, o SX1272 aguarda até receber um único pacote ou até que se alcance o *timeout* de recepção configurado. Acontecendo qualquer um dos dois, o *transceiver* volta ao modo STANDBY.

3.2 Estrutura de Pacote

São dois os tipos de pacote na modulação LoRa^{TM}, a depender do modo de cabeçalho configurado. O modo *default* é o cabeçalho implícito, onde não é incluído nenhum cabeçalho com informações adicionais sobre o pacote. A outra opção é o cabeçalho explícito, que faz com que seja adicionado um cabeçalho ao pacote.

O pacote LoRa^{TM}se divide em três elementos: preâmbulo; cabeçalho opcional; e os dados transmitidos (com inclusão opcional de CRC neste último elemento) (SEMTECH CORPORATION, 2015). Isso pode ser visto na Figura 3.

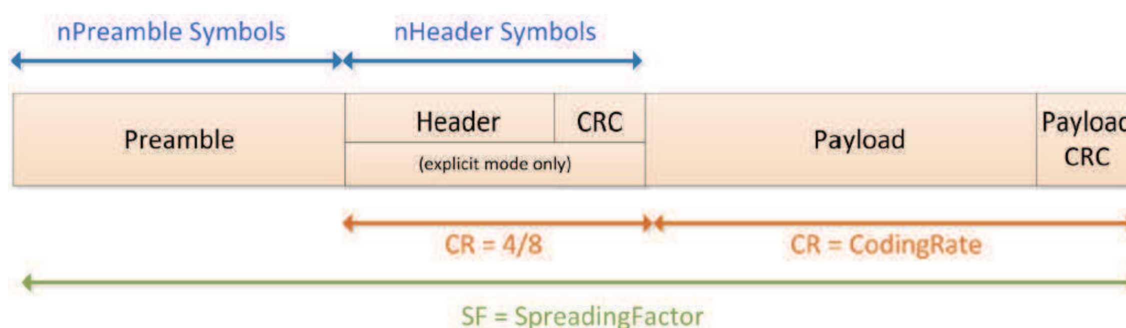


Figura 3 – Estrutura de pacoe LoRa^{TM} (SEMTECH CORPORATION, 2015).

3.2.1 Preâmbulo

O preâmbulo é parte obrigatória e fundamental no processo de transmissão de pacote. O nó receptor executa periodicamente um processo de detecção de preâmbulo. Em outras palavras, o receptor detecta um que existe um pacote para receber pelo seu preâmbulo. O tamanho deste prêambulo é configurável e, no caso do SX1272, pode ir de de $(6+4)$ até $(65535+4)$ símbolos (SEMTECH CORPORATION, 2015). A única observação

é que ele deve ser idêntico nos nós transmissor e receptor, já que, do contrário, o pacote seria descartado pelo receptor.

3.2.2 Cabeçalho

O cabeçalho é opcional, como citado previamente, e é inserido no pacote apenas caso seja especificada a opção de cabeçalho explícito. Ele inclui as seguintes informações sobre o pacote: número de *bytes* do *payload*; *coding rate* (detalhes sobre *coding rate* na seção 3.3); e a disponibilidade, ou não, de CRC.

O cabeçalho é transmitido sempre com *coding rate* 4/8 e com seu próprio CRC, para que o receptor seja capaz de descartar cabeçalhos inválidos (SEMTECH CORPORATION, 2015).

Um detalhe importante é que, nos casos em que seja configurado $SF = 6$, apenas o modo de cabeçalho implícito é permitido.

3.2.3 *Payload*

O *payload* pode chegar a, no máximo, os 256 *bytes* da FIFO (*First In, First Out* - Primeiro a entrar, primeiro a sair) (SEMTECH CORPORATION, 2016a). Além disso, o *payload* é codificado em um número maior de *bytes* que a quantidade de dados não-redundantes. Essa quantidade final depende do parâmetro CR. Por fim, um CRC opcional pode ser incluído ao pacote para ser utilizado como parâmetro de integridade de pacote pelo nó receptor.

3.3 *Coding Rate* (CR)

Para utilização em aplicações de correção de erro, os pacotes transmitidos num sistema de comunicação podem ser codificados de modo a conter mais símbolos do que o dado que se deseja transmitir. O excesso de dados é redundante, útil à correção de erros, e a taxa na qual ocorre essa codificação é dada pelo *coding rate*. Em outras palavras, se um *coding rate* tem valor 4/8, a cada 4 símbolos que se deseja transmitir, serão transmitidos, na realidade, 8, sendo $(8 - 4)$ a quantidade de símbolos redundantes (HUFFMAN; PLESS, 2003).

No caso do SX1272, estão disponíveis quatro opções de CR: 4/5; 4/6; 4/7; e 4/8. Não é possível suprimir a codificação.

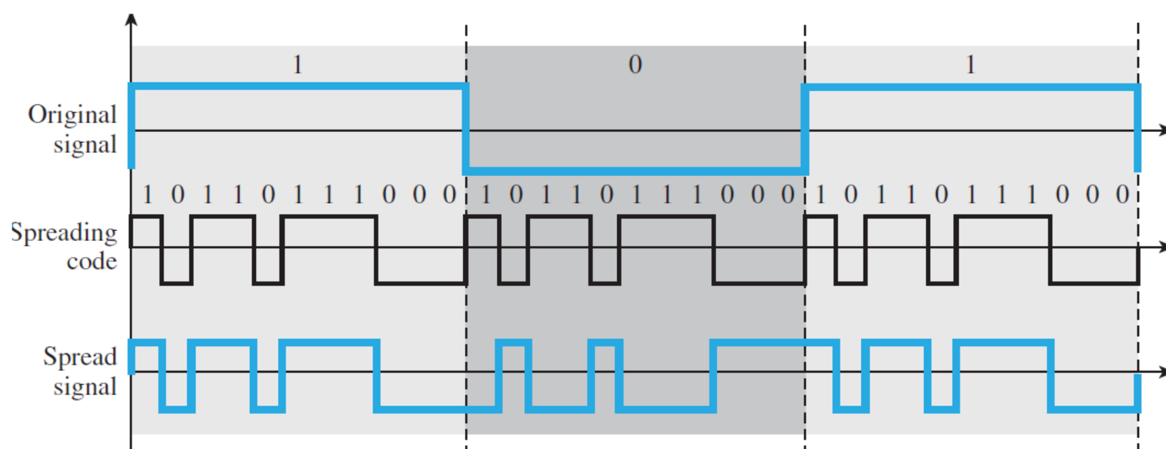


Figura 4 – Exemplo de DSSS (FOROUZAN, 2013).

3.4 *Spreading Factor* (SF)

A modulação LoRa^{TM} funciona com uma estratégia de transmissão de pacotes utilizando espalhamento de espectro (*spread spectrum*). Essa estratégia é similar a um DSSS (*Direct Sequence Spread Spectrum* - Espalhamento de Espectro por Sequência Direta), onde cada *bit* é codificado em um número n de *bits*. Esses n *bits* são chamados *chips* (FOROUZAN, 2013). A Figura 4 exemplifica o funcionamento de um esquema DSSS.

No caso do LoRa^{TM}, cada símbolo também é representado por múltiplos *chips*, e a razão entre o número de *chips* transmitidos e a quantidade nominal de símbolos é chamada *spreading factor*. No SX1272, existem sete opções de SF (SF6 até SF12). Ao contrário do que pode parecer, os valores de 6 a 12 não são de fato os valores de SF. Estes são os valores que devem ser escritos no devido registrador para configurar os reais números de SF. Estes reais valores são 2^x, sendo x o valor escrito no registrador. Portanto, um modem LoRa^{TM} configurado com SF12 apresenta *spreading factor* igual a 2^{12} (4096).

Quão maior o valor de SF, maior será a banda ocupada, menor a taxa de transmissão w maior a sensibilidade do receptor (SEMTECH CORPORATION, 2015). Além disso, é importante destacar que diferentes SF são ortogonais. Ou seja, o mesmo SF deve ser configurado de ambos os lados da transmissão, ou a comunicação não terá sucesso.

3.5 Largura de Banda (BW)

Existem três possíveis larguras de banda configuráveis no modo LoRa^{TM} para o *transceiver SX1272*. São elas: 125 kHz; 250 kHz; e 500 kHz. Diferente do usual em modulações FSK, essa largura de banda se refere à banda total do canal (SEMTECH CORPORATION, 2015).

O aumento na banda de passagem permite maiores taxas de transferência de dados.

No entanto, esse aumento ocorre às custas de uma depreciação em termo de sensibilidade no nó receptor.

3.6 Frequência Central de Transmissão

O *transceiver* SX1272 pode funcionar na faixa de frequências entre 860 e 1020 MHz. A configuração é bastante simples e segue a relação matemática abaixo:

$$f_{RF} = \frac{F(XOSC) \cdot Frf}{2^{19}} \tag{3.1}$$

Sendo f_{RF} a frequência desejada, em Hz; $F(XOSC)$ a frequência do cristal oscilador; e Frf o valor que deve ser escrito nos registradores de configuração da frequência central (são três registradores).

3.7 Sensibilidade e Taxa de Transmissão

A tabela a seguir foi construída com auxílio da LoRa^{TM} *Calculator*. O objetivo é propor variações dos parâmetros SF, CR e BW, e observar como é previsto que se comportem os parâmetros de sensibilidade e taxa de transmissão.

SF	**CR**	**BW [kHz]**	**Sensibilidade do receptor [dBm]**	**Taxa de Transmissão [bps]**
6	4/5	125	-118	9375
6	4/5	500	-112	37500
6	4/8	125	-118	5860
6	4/8	500	-112	23438
12	4/5	125	-137	293
12	4/5	500	-131	1172
12	4/8	125	-137	183
12	4/8	500	-131	7332

Tabela 1 – Estimativas de sensibilidade e taxa de transmissão

4 API Embarcada

A API embarcada foi desenvolvida para facilitar o controle e operação do SX1272. Ela foi implementada para dois microcontroladores, ambos da família MSP430, da Texas Instruments. Por serem microcontroladores da mesma família, as mudanças existentes entre as duas versões são mínimas. Os dois microcontroladores foram o MSP430G2553, utilizado por meio do *launchpad* MSP-EXP430G2 (Figura 5); e o MSP430F6736, uma versão mais robusta e com mais recursos que o anterior.

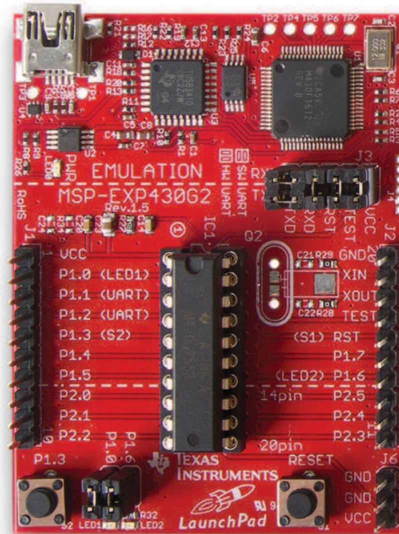


Figura 5 – MSP-EXP430G2 (TEXAS INSTRUMENTS, 2016b).

Foi utilizado o IAR Embedded Workbench, versão 5.6, da IAR Systems, como ambiente de desenvolvimento para ambos os microcontroladores. O código desenvolvido foi feito em linguagem C++ e a versão compatível com MSP430F6736 está disponível no Apêndice A.

A API pode ser dividida nos seguintes módulos:

- Módulos auxiliares (padronização de variáveis; lista de registradores do SX1272; protocolo de comunicação serial; constantes)
- Comunicação UART (*Universal Asynchronous Receiver/Transmitter* - Receptor/-Transmissor Universal Assíncrono) do ponto de vista do MSP430
- Comunicação SPI do ponto de vista do MSP430
- Interface SPI do ponto de vista do SX1272

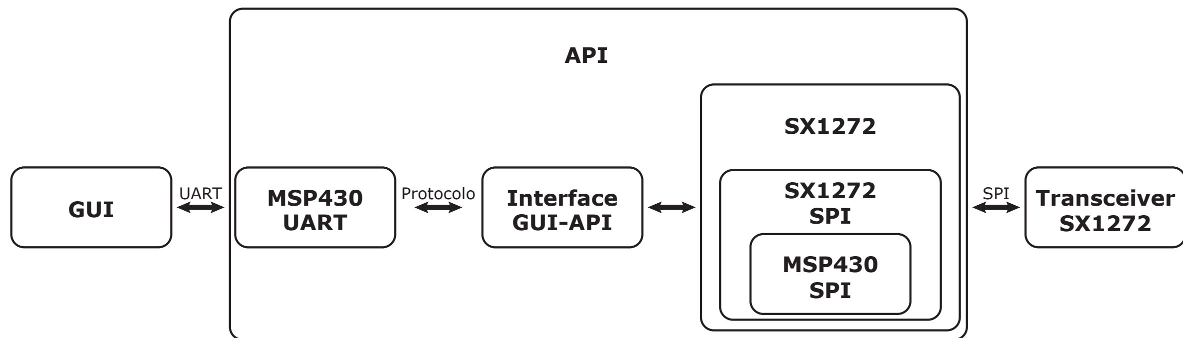


Figura 6 – Diagrama da organização funcional da API.

- Configuração e operação do SX1272
- Módulo que faz interface entre comandos da GUI e execução de baixo nível

A Figura 6 dá uma boa ideia da organização funcional desses módulos. Nas seções a seguir, apresenta-se uma descrição mais detalhada sobre eles.

4.1 Comunicação UART do ponto de vista do MSP430

A comunicação UART é utilizada entre o microcontrolador MSP430 e a interface gráfica de usuário desenvolvida para computador (a GUI será descrita mais adiante no Capítulo 5). Essa comunicação tem por finalidade manter um fluxo de informação da situação do *transceiver*, bem como configurá-lo e controlá-lo, seja no nó transmissor ou receptor.

Para implementar esse módulo, foram utilizadas rotinas de interrupção no MSP430, tanto para transmissão quanto recepção.

A classe que compõe este módulo é chamada *TMsp430UART*. Mais detalhes no Apêndice A.

4.2 Comunicação SPI do ponto de vista do MSP430

Este módulo serve de base para a comunicação com o *transceiver*. Aqui residem as funções de mais baixo nível que serão utilizadas para interfaceamento com o SX1272. Em termos de implementação, é basicamente análogo ao módulo de comunicação UART. Isso se deve ao fato de ambos serem periféricos do mesmo tipo no microcontrolador. Tanto SPI quanto UART são periféricos USCI (*Universal Serial Communication Interface* - Interface de Comunicação Serial Universal) e, no caso do MSP430G2553, partilham inclusive das mesmas portas e do mesmo vetor de interrupção.

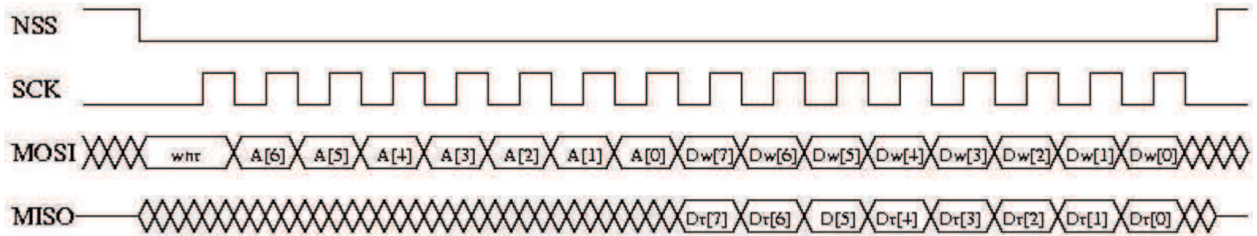


Figura 7 – Diagrama de tempo de acesso à interface SPI do *transceiver* SX1272 (SEMTECH CORPORATION, 2015).

A implementação deste módulo se dá pela classe *TMsp430SPI*, disponível no Apêndice A.

4.3 Interface SPI com *transceiver* SX1272

A interface SPI do SX1272 é utilizada para qualquer tipo de escrita e leitura dos seus registradores. Isso inclui o acesso aos registradores de configuração e à FIFO. Independentemente do propósito do acesso, a interface funciona da mesma forma para cada operação individual de escrita/leitura (Figura 7).

Para se iniciar um acesso, o pino NSS (*Slave Select*, nível baixo ativo) deve ser levado ao nível lógico 0. Neste momento, o canal SCK (*System Clock*), que estava inativo, é ativado. A partir daí, são utilizadas as funções de comunicação SPI de mais baixo nível criadas no módulo de comunicação SPI do ponto de vista do MSP430. Se escreve no pino MOSI (Master Output, Slave Input) um palavra de 8-*bits* com a seguinte estrutura: o bit mais significativo (escrito primeiro) é chamado WNR (*Write, Not Read*) e define se o acesso é para escrita ou leitura; os 7 *bits* seguintes são, mais significativo ao menos significativo, o endereço de acesso (seja para escrita ou leitura).

Cada vez que se escreve no pino MOSI, algo novo é escrito pelo *transceiver* no pino MISO (*Master Input, Slave Output*), e somente nesses instantes. Portanto, independentemente da operação ser de leitura ou escrita, é necessário que se escreva duas palavras no pino MOSI.

Caso a operação seja de escrita, uma vez enviado o endereço no qual se deseja escrever, escreve-se no pino MOSI o dado desejado. Caso a operação seja de leitura, uma vez enviado o endereço do registrador que se deseja ler, alguma palavra de 8 *bits*, não importando qual, deve ser escrita no pino MOSI para que a leitura do dado desejado esteja disponível no pino MISO.

Este módulo está implementado na classe *TSx1272SPI*, no Apêndice A.

4.4 Configuração e Operação do SX1272

Este módulo faz uso do módulo de interface SPI para implementar funções úteis na utilização do SX1272. Exemplos dessas funções são: escrita/leitura da FIFO; configuração de parâmetros e modo de operação; rotina de transmissão/recepção (no caso de recepção, foi implementada apenas a rotina para o modo RXSINGLE).

Em outras palavras, este módulo é responsável pela configuração de mais alto nível em contato direto com o *transceiver*, fazendo uso, de modo encapsulado, dos módulos anteriores de mais baixo nível. A implementação dele é feita pela classe *TSx1272*, também disponível no Apêndice A.

4.5 Módulo de interface GUI-API

Este módulo, implementado pela classe *MidInterfaceUILowLevel* (Apêndice A), funciona como o administrador do sistema como um todo. Ele se mantém em comunicação constante com a GUI via UART e, como o nome sugere, faz a interface entre a interface gráfica operando em um computador e a API em si. De certo modo, pode-se dizer que ele recebe as requisições vindas da interface gráfica, as "traduz" e aplica ao *transceiver*. A palavra "tradução" se aplica bem, inclusive, pois foi desenvolvido um protocolo simples a ser utilizado na comunicação via porta UART. O protocolo está disponível, como foi implementado, no Apêndice A.

É neste módulo que se encontram funções como a rotina completa do modo de transmissão/recepção (permitindo reconfiguração nos momentos de *standby*), configuração inicial (emulando uma configuração feita pela GUI), e outras funcionalidades citadas mais adiante no Capítulo 5, quando será tratado sobre a GUI desenvolvida.

5 Interface Gráfica de Usuário

O *software* foi desenvolvido no Visual Studio 2013, IDE (*Integrated Development Environment* - Ambiente de Desenvolvimento Integrado) da Microsoft, na linguagem C#. Para que seja possível utilizar a interface, é necessário que o usuário tenha instalada a plataforma de *softwares* .NET Framework (4.5 ou superior). Não é necessária nenhuma instalação da interface em si, apenas a execução do seu arquivo .EXE.

A interface foi criada para facilitar a configuração e os testes do *transceiver* SX1272, fazendo uso do MPS430 como controlador. A conexão é feita via porta serial com a UART do MSP430 que, por sua vez, interpreta o protocolo criado e executa as devidas tarefas (configuração local, configuração remota, transmissões, recepções, checagem de status do receptor a distância).

5.1 Protocolo de Instruções

O protocolo de instruções utilizado via porta serial é bastante simples e não será inteiramente detalhado. Porém, para exemplificar, as instruções são, normalmente, compostas de uma letra inicial, que identifica a tarefa, e números, que determinam como aquela tarefa deve ser executada. Por exemplo, para configurar a largura de banda do SX1272 para 125 kHz, seria enviada a seguinte mensagem via UART: "b1"; "b" que indica configuração de largura de banda, e 1 que indica 125 kHz (125 kHz sendo representado pelo índice 1; 250 kHz sendo representado pelo índice 2; e 500 kHz sendo representado pelo índice 3). Todo o protocolo segue de maneira análoga ao exemplo.

5.2 Utilização da Interface

A Figura 8 apresenta a GUI assim que ela é iniciada. A primeira coisa a se fazer, antes de se iniciar a comunicação, é configurar e inicializar a porta serial de acordo com a maneira que ela foi configurada no MSP430. Para tal, utiliza-se o menu da direita, onde é possível selecionar nome da porta, *baud rate*, paridade, quantidade de bits de dados e bits de parada. Uma vez feito isso, clica-se no botão "Inicializar" para estabelecer a comunicação serial; o sinal deve mudar de vermelho para verde, como na Figura 9.

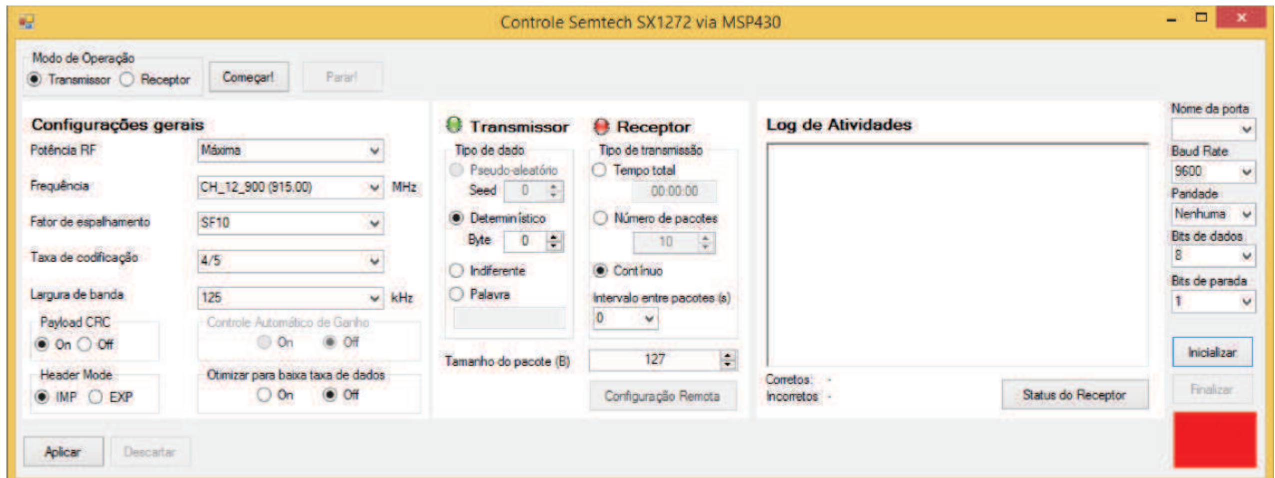


Figura 8 – Interface gráfica de usuário, assim que é inicializada.

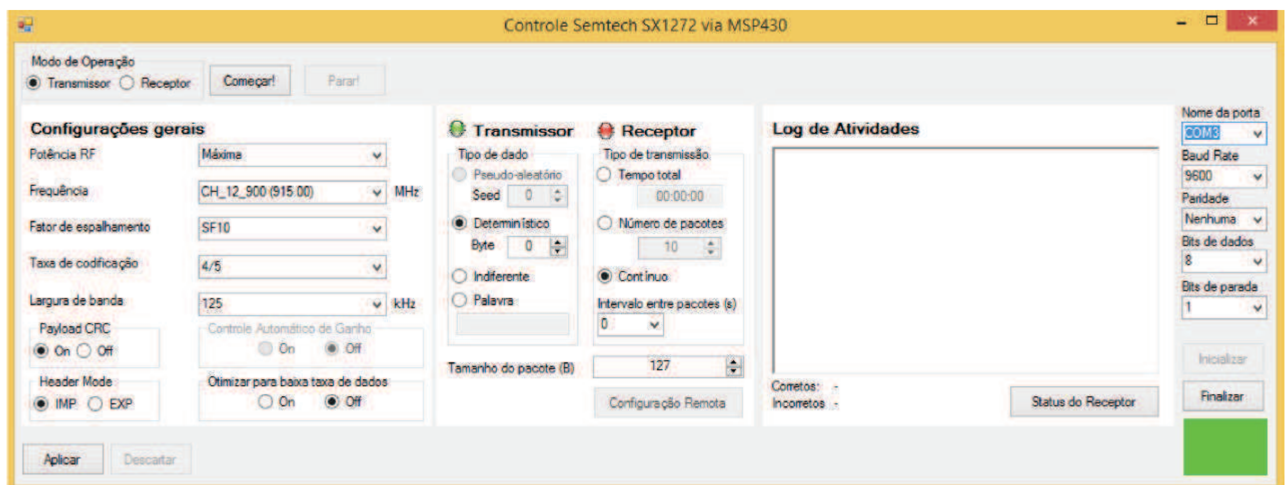


Figura 9 – Interface gráfica de usuário, após configuração e estabelecimento da comunicação serial.

Em seguida, a depender do modo de operação (ou mesmo de alguma característica como *header mode*, algumas configurações são mandatórias, outras não são necessárias. Isso sempre se refletirá na interface. Por exemplo, no modo transmissor, não é necessária a configuração do controle automático de ganho, pois ele só interfere no modo receptor. Por esse motivo, essa configuração nem está disponível quando selecionado o modo transmissor.

Ao mudar o modo de operação e aplicar a configuração feita (clicando no botão "Aplicar", isso será indicado na interface pela mudança do sinal ao lado dos nomes "Transmissor" e "Receptor". O modo atual será sinalizado pela cor verde e o outro pela cor vermelha.

As configurações de "Tipo de dado" e "Tipo de transmissão" não são enviadas via porta serial, pois elas são utilizadas na própria interface para gerenciar a transmissão. É válido lembrar que todos os dados a serem enviados são gerados pela interface.

Os tipos de transmissão são três: por "Tempo total", onde a transmissão ocorre durante um tempo determinado e se encerra sozinha (por "sozinha", entende-se que não há interferência do usuário); por "Número de pacotes", em que é transmitido exatamente o número de pacotes selecionado; e o modo "Contínuo", onde o usuário não só inicia a transmissão, como também a finaliza quando bem entender. Além disso, existe a opção de configurar o intervalo mínimo entre pacotes, gerenciado pela interface gráfica, que garante que o próximo pacote não seja enviado num intervalo menor que o determinado.

Os tipos de dado são quatro: "Pseudo-aleatório", em que é fornecida a "Seed" de geração aleatória para que os dados não sejam todos idênticos, mas que seja possível saber o que esperar no nó receptor; "Determinístico", onde, com o tamanho de pacote definido, este modo faz com que todos os *bytes* do pacote sejam iguais e definidos pela configuração do "Byte", que vai de 0 a 255; o tipo "Indiferente" que gera dados totalmente aleatórios, para quando não há importância no dado enviado, apenas na informação de se o pacote chegou correto ou incorreto (essa informação é fornecida pela API, pela verificação do CRC); o modo "Palavra" que, apesar de se chamar "palavra", dá a opção de se enviar, num pacote, qualquer tipo de texto, até 127 caracteres. Recomenda-se que não seja enviado neste modo nada que se inicie com a letra "i" (*case sensitive*), pois pacotes iniciados com essa letra indicam, ao receptor, uma instrução remota.

Cada configuração só é enviada ao MSP430 no momento em que é clicado o botão "Aplicar". Cada mudança na interface que não foi ainda aplicada pode ser retornada ao estado anterior ao clicar no botão "Descartar". Este procedimento de configuração é válido quando feito localmente, via porta serial. Entretanto, o procedimento para configuração remota é muito similar, a nível de usuário. Os parâmetros são escolhidos como se a configuração fosse local mas, ao invés de clicar no botão "Aplicar", clica-se no botão "Configuração Remota". Desta maneira, o *transceiver* transmissor enviará um pacote com as instruções de reconfiguração ao receptor e, após esse envio, reconfigura a si próprio para entrar em conformidade com o outro nó.

Após aplicada a configuração, para iniciar a transmissão real, clica-se no botão "Começar". Para parar, clica-se no botão "Parar".

Para checar o *status* do receptor, é necessário que não esteja sendo transmitido nada no momento. Garantido isso, apenas clica-se no botão "Status do Receptor". O nó transmissor enviará um requerimento do *status* ao receptor e aguardará por um determinado tempo (tempo definido na API como *timeout* de recepção) sua resposta. Se a resposta for recebida, será mostrado no "Log de Atividades" a quantidade de pacotes recebida correta e incorreta.

Além dessa função de mostrar o *status* do receptor, o "Log de Atividades" registra todas as atividades de transmissão em detalhes, como hora em que foi enviado um pacote, se ele foi ou não enviado com sucesso e ainda o que estava contido no pacote. Ele funciona

de forma similar no nó receptor, onde registra as mesmas informações a respeito dos pacotes recebidos. Ao fim de cada transmissão/recepção, este *log* é transformado em um relatório simples que é salvo num arquivo .TXT na pasta onde está o arquivo .EXE. Além do *log*, estes relatórios contêm a configuração utilizada em cada transmissão/recepção.

6 Procedimentos e Resultados

Os testes realizados buscaram comparar o *transceiver* SX1272 com um dos seus concorrentes no mercado de *transceivers* de alta performance e baixo consumo, o Si4468, da Silicon Labs.

No quesito consumo, não existe grande margem para qualquer dos lados. Ambos apresentam baixos valores de corrente para recepção. No caso do Si4468, essa corrente se situa entre 10 e 13 mA (SILICON LABORATORIES, 2014), enquanto no SX1272, não passa dos 10 mA (SEMTECH CORPORATION, 2015). Para fins de comparação, outro *transceiver* considerado de baixo consumo e alta performance, o CC1121, da Texas Instruments, pode atingir correntes de 22 mA durante o processo de recepção (TEXAS INSTRUMENTS, 2016a).

A suposta vantagem que o SX1272 leva é por motivos óbvios: melhor desempenho por um "menor" preço. A palavra "menor" foi posta entre aspas porque o preço em si do SX1272 é maior[1]. Na pesquisa de preços feita, a unidade do *chip* SX1272 é vendido por US$ $3,97800$[2], enquando a unidade do *chip* Si4468 custa US$ $2,52555$[3], dando a vantagem inicial ao *transceiver* da Silicon Labs. A diferença surge quando se fala da montagem como um todo. Para alcançar taxas muito baixas de transmissão, onde o recepetor opera com a maior sensibilidade, o SX1272 utiliza um cristal oscilador de 32 MHz e precisão de 10 ppm, que custa entre US$ $0,42050$ e US$ $0,79000$[4], enquanto o Si4468 precisa de um cristal oscilador mais sofisticado, mais preciso e mais caro. Um cristal TCXO (*Temperature Compensated Crytal Oscillator* - Cristal Oscilador com Compensação de Temperatura) de 32 MHz, recomendado para uso junto ao Si4468 para operação a baixas taxas de transmissão (SILICON LABORATORIES, 2014), custa, no mínimo, US$ $2,14600$[5], podendo custar até mais de US$ $3,00$.

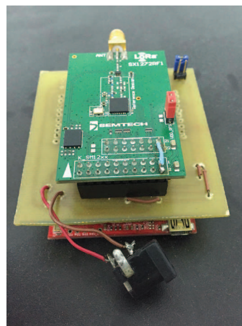
Colocando o SX1272 no pior caso (considerando o oscilador de US$ $0,79000$) e o Si4468 no melhor caso (considerando o oscilador de US$ $2,14600$), o Si4468 levaria uma leve vantagem de cerca de US$ $0,10$, porém, mesmo nessas condições, conseguindo alcançar uma taxa de transmissão de 100 bps (inferior aos 183 bps do SX1272), a sensibilidade do receptor do Si4468 chega a -132 dBm (SILICON LABORATORIES, 2014), que ainda é inferior aos -137 dBm do SX1272.

[1] Preços consultados na loja virtual Digi-Key Electronics
[2] Mínimo de 3000 unidades
[3] Mínimo de 2500 unidades
[4] Mínimo de 3000 unidades
[5] Mínimo de 1000 unidades

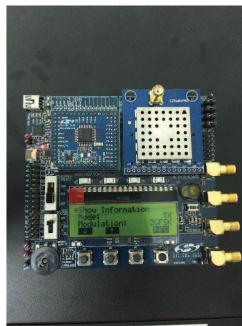
6.1 Metodologia Utilizada

Para realizar os testes, foram utilizados dois *kits*, um de cada fabricante. No caso do SX1272, um *kit* com o módulo *transceiver* já montado e uma antena monopolo (Figura 10a). Foi feita ainda uma montagem com um *shield* para acoplar este módulo e o *launchpad* do MSP430 (Figura 10b). Já quanto ao Si4468, foi utilizado um módulo já programado e configurável fornecido pela Silicon Labs (Figura 10c).

(a) Módulo *transceiver* SX1272 (DIGI-KEY ELECTRONICS, 2016).



(b) Módulo *transceiver* SX1272 acoplado ao *launchpad*.



(c) Módulo *transceiver* Si4468.

Figura 10 – Módulos de teste

Como não se dispunhas de um TCXO no módulo do Si4468, os testes foram feitos de maneira "igualitária", a princípio. Ambos os *transceivers* foram configurados para operar na mesma taxa de transferência (1.2 kbps, a mínima configurável no módulo do Si4468) e utilizando a mesma frequência central (915 MHz). Enquanto o SX1272 utilizou a modulação LoRa$^{\text{TM}}$, o Si4468 utilizou uma modulação 2GFSK. O tamanho de pacote

foi configurado para 63 *bytes* além da inserção do CRC (pois esse era o limite configurável para o módulo do Si4468).

Uma vez findado esse teste, o SX1272 foi colocado na configuração onde, supostamente, a sensibilidade no seu receptor é máxima (SF: 12; CR: 4/5; BW: 125 kHz) e foram testados os limites do SX1272 num ambiente urbano (é esperado que os resultados sejam significativamente melhores num ambiente sem obstáculos).

O nó transmissor de ambos os *transceivers* foi colocado no topo do prédio do eSMART/DEE/CEEI/UFCG (Laboratório de Sistemas Inteligentes), aproximadamente no ponto indicado na Figura 11, transmitindo continuamente na taxa citada previamente.

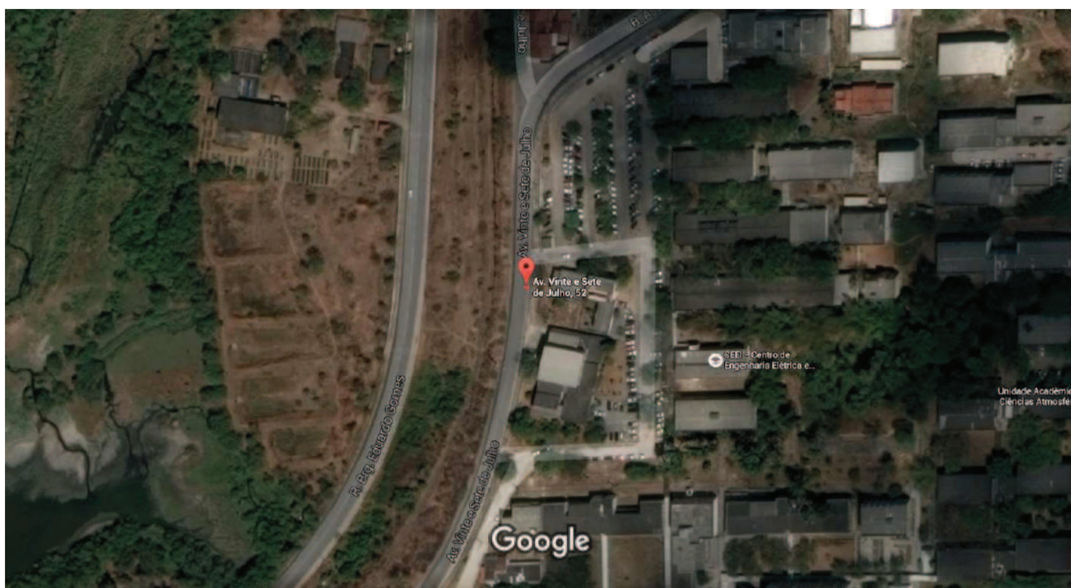


Figura 11 – Localização do nó transmissor (GOOGLE MAPS, 2016).

6.2 Resultados

6.2.1 Testes Comparativos

Os testes comparativos foram realizados no período da tarde, por volta das 16 horas, de um dia bastante ensolarado. Com os nós transmissores instalados no ponto previamente citado, rumou-se, inicialmente, para o ponto mostrado na Figura 12, situado a cerca de 920 m de distância.

Tanto para este teste quanto para os outros que virão a seguir, verificava-se se era possível estabelecer comunicação e, caso sim, eram recebidos um total de 100 pacotes, verificando quantos estavam corretos e incorretos. Quaisquer detalhes e comentários específicos a serem feitos sobre um ou outro teste, serão trazidos a tona quando for tratado de cada teste.

Figura 12 – Distância de transmissor e receptor no primeiro ponto de testes (GOOGLE MAPS, 2016).

Neste teste inicial, ambos os *transceivers* tiveram um percentual de sucesso na recepção de 100%, sem nenhum grande intervalo entre pacotes. Este foi escolhido como ponto inicial pois a distância já beirava 1 km e havia uma visada praticamente direta entre os nós, favorecendo a comunicação.

O segundo ponto de testes foi em frente à Fundação Parque Tecnológico da Paraíba (PaqTcPB), a cerca de $1,68$ km (Figura 13). Este local era mais distante que o primeiro e já não existia visada direta, pois a vista era encoberta pelo relevo, pela própria construção do PaqTcPB, e por algumas outras poucas construções mais distantes. Mesmo assim, o percentual de sucesso da transmissão de ambos foi, novamente, de 100%.

O próximo teste foi feito por trás do Residencial Dona Lindu, que ficava a uma distância de $1,95$ km do nó transmissor (Figura 14). Neste ponto, apesar de se manter um pouco afastado de regiões mais urbanizadas, onde há mais obstáculos à comunicação, haviam dois grandes residenciais encobrindo e a distância já beirava 2 km. Neste teste ocorreram os primeiros erros de ambos, mas a taxa de sucesso ainda seguiu alta. O SX1272 teve taxa de sucesso de 94%, enquanto o Si4468 atingiu um percentual de 90%. Uma observação a ser feita aos dois casos, é que houveram alguns intervalos grandes entre alguns pacotes, provavelmente devido à perda de alguns pacotes transmitidos, mas não recebidos pelo respectivo nó. Esse problema ocorreu com frequência bem maior no Si4468, enquanto apenas algumas poucas vezes no SX1272.

No quarto teste, pela primeira vez, ocorreu de não se conseguir estabelecer comunicação com um dos *transceivers*. Aproximadamente no ponto da Figura 15, a $1,50$ km,

Figura 13 – Distância de transmissor e receptor no segundo ponto de testes (GOOGLE MAPS, 2016).

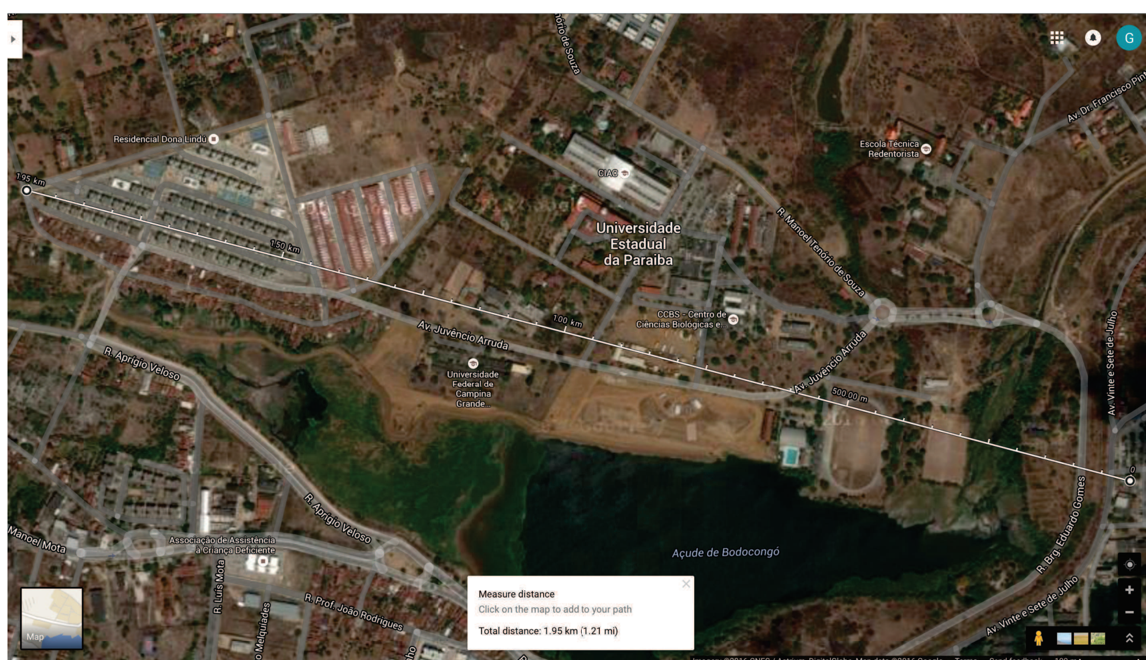


Figura 14 – Distância de transmissor e receptor no terceiro ponto de testes (GOOGLE MAPS, 2016).

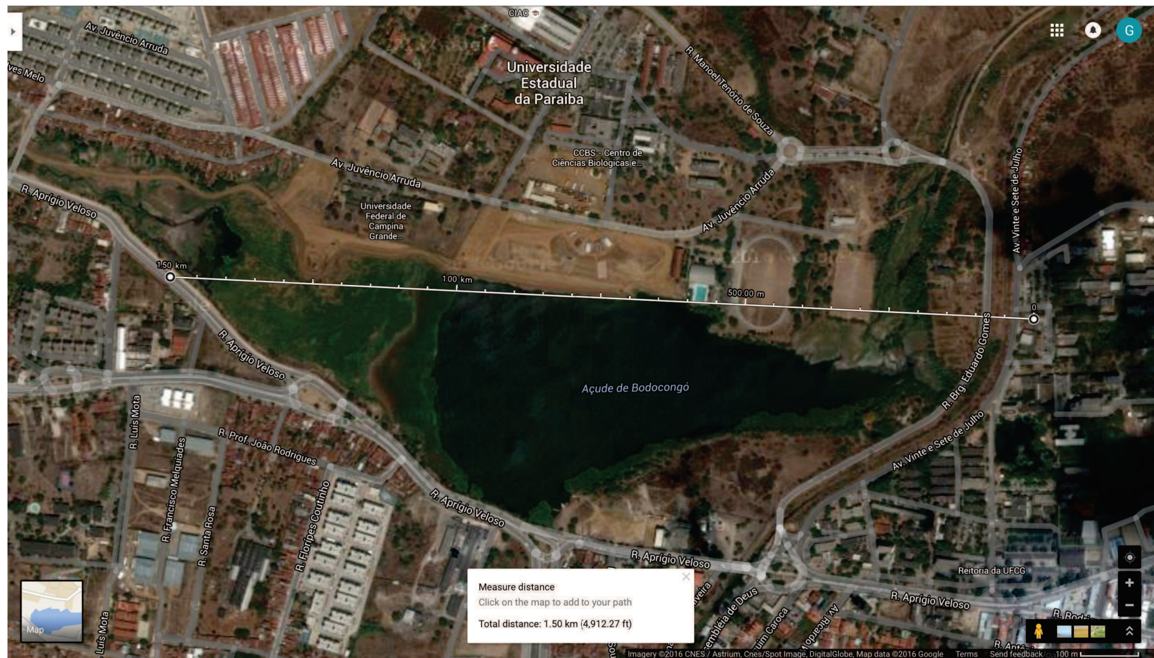


Figura 15 – Distância de transmissor e receptor onde cessou a comunicação do Si4468 durante o quarto teste (GOOGLE MAPS, 2016).

seguindo pela pista à esquerda do Açude de Bodocongó, a comunicação com o Si4468 foi perdida e não conseguiu se reestabelecer. Durante o trajeto até esse ponto, a comunicação foi se depreciando até que cessou. Isso não era esperado pois, em teoria, ambos *transceivers* haviam conseguido se comunicar a distâncias maiores e com mais obstáculos. O local onde o quarto teste se passou — apenas com o SX1272, dadas as condições — está indicado pela Figura 16, a cerca de $1,80$ km. O SX1272 se comportou bem e alcançou percentual de sucesso de 92%, com alguns poucos intervalos maiores que a média entre pacotes, onde se infere que ocorreu perda dos pacotes transferidos, visto que, como era mantida uma taxa de transmissão constante, os intervalos de recepção deveriam ser constantes.

O quinto, e último, teste comparativo se deu no Canal de Bodocongó. Este, supostamente, era o teste que mais exigiria dos *transceivers*, pois existem muitos obstáculos no trajeto. Mais uma vez, o Si4468 perdeu comunicação num ponto anterior ao SX1272 (Figura 17), por volta de $1,33$ km. O SX1272 perdeu comunicação pela primeira vez neste teste, ao se colocar por trás de um prédio, com uma distância de, aproximadamente, $1,60$ km (Figura 18). Ambos os sinais foram sendo depreciados, apresentando muitos grandes intervalos entre pacotes, muitos pacotes errados, até que cessaram, cada um num respectivo ponto, a comunicação.

É importante deixar claro que, apesar do nó transmissor ter sido instalado no topo de um prédio, o nó receptor sempre esteve no nível do chão, colocando vários obstáculos no trajeto. Num ambiente urbano, em geral, quão mais altos estiverem localizados esses nós, melhor. Isso acontece devido à existência de prédios e construções verticais que se comportam como obstáculos à propagação do sinal. Logo, quão mais altos os nós, tendem a

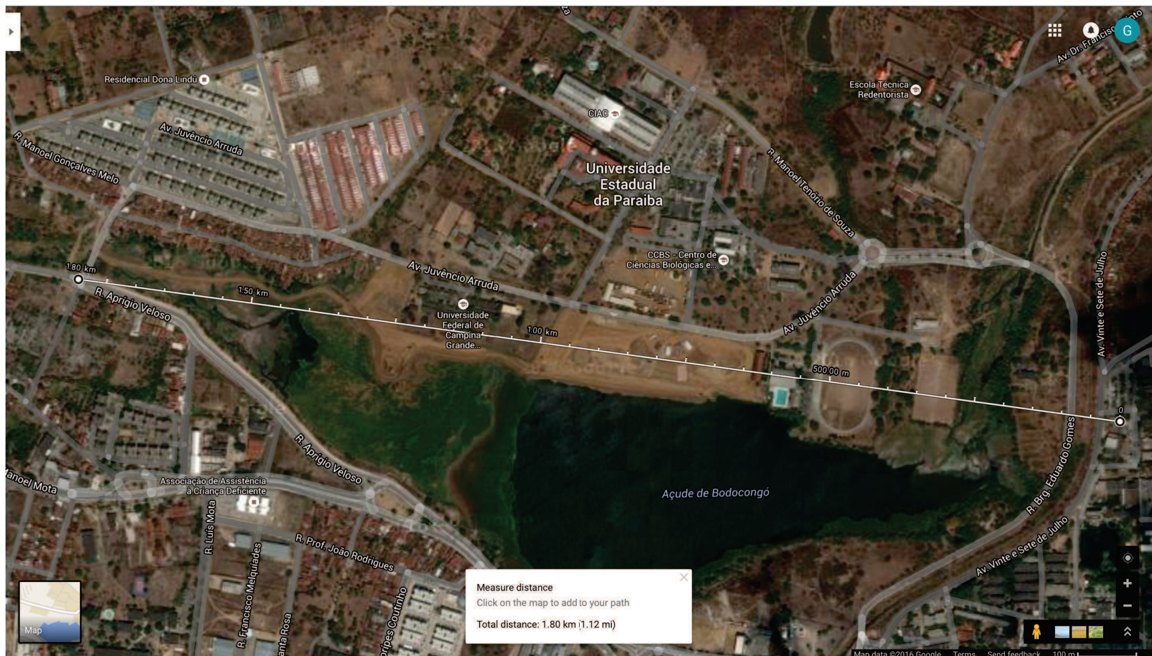


Figura 16 – Distância de transmissor e receptor (SX1272) no quarto ponto de testes (GOOGLE MAPS, 2016).

Figura 17 – Distância de transmissor e receptor onde cessou a comunicação do Si4468 durante o quinto teste (GOOGLE MAPS, 2016).

Figura 18 – Distância de transmissor e receptor (SX1272) no quinto ponto de testes (GOOGLE MAPS, 2016).

existir menos obstáculos entre os nós e isso pode fazer uma diferença considerável. Como o teste foi apenas comparativo e ambos *transceivers* foram submetidos às mesmas condições, esse fator não compromete a comparação feita, mas deixa claro que ela se aplica àquele ambiente, especificamente.

6.2.2 Teste de Alcance Máximo do SX1272

Como o SX1272, a partir de certo ponto, não conseguiu estabelecer comunicação no quinto e último teste comparativo, considerou-se que seria um bom local para um dos testes de alcance quando fazendo uso de sua máxima configuração.

O alcance obtido desta vez foi consideravelmente maior que aquele obtido nos testes comparativos, chegando a $2,53$ km (Figura 19) — quase 1 km a mais que o anterior. Seguindo a mesma rotina de teste, o percentual de sucesso de transmissão neste caso foi de 96%.

Apenas mais um teste foi realizado, e num ponto estratégico. Rumou-se à Avenida Dinamérica, uma avenida de aclive acentuado e realizou-se um teste no topo da ladeira. Esta foi a maior distância de comunicação estabelecida durante os testes para este trabalho, atingindo $2,81$ km (Figura 20). Após este ponto, na descida da ladeira, de modo que o nó receptor acabava por ficar encoberto pelo relevo, a comunicação começava a ficar precária, com muita perda de pacotes e muitos pacotes recebidos incorretamente. Isso sugere que, especialmente a longas distâncias, a presença de obstáculos pode ser crítica. Enquanto a comunicação se manteve, obteve-se um percentual de sucesso de 93% na recepção de

pacotes.

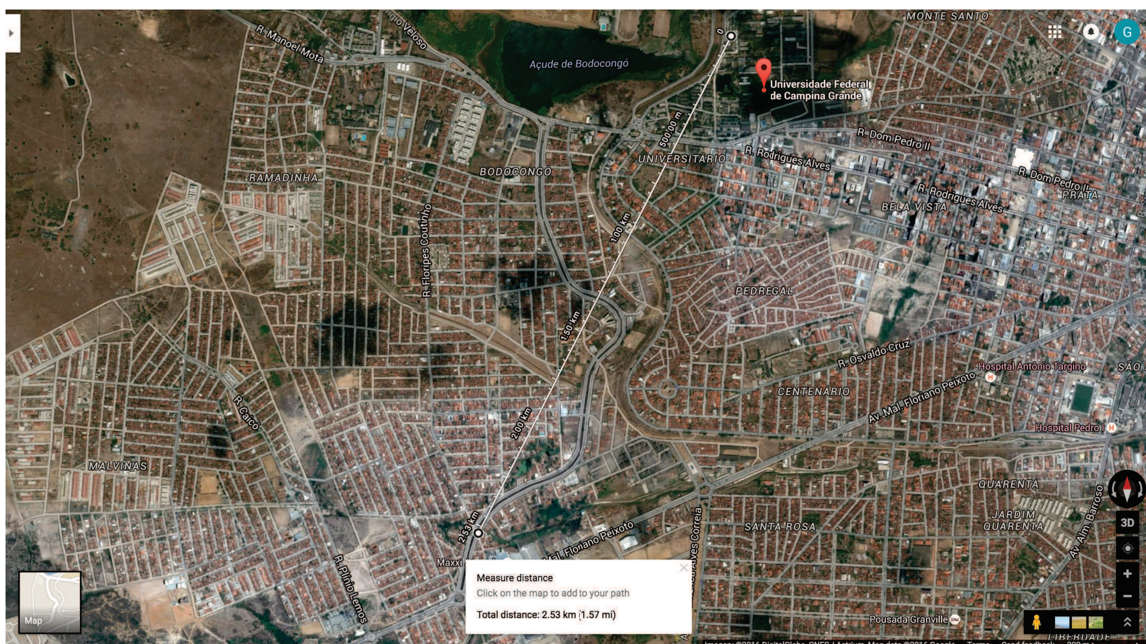


Figura 19 – Distância de transmissor e receptor no primeiro teste de máxima configuração (GOOGLE MAPS, 2016).

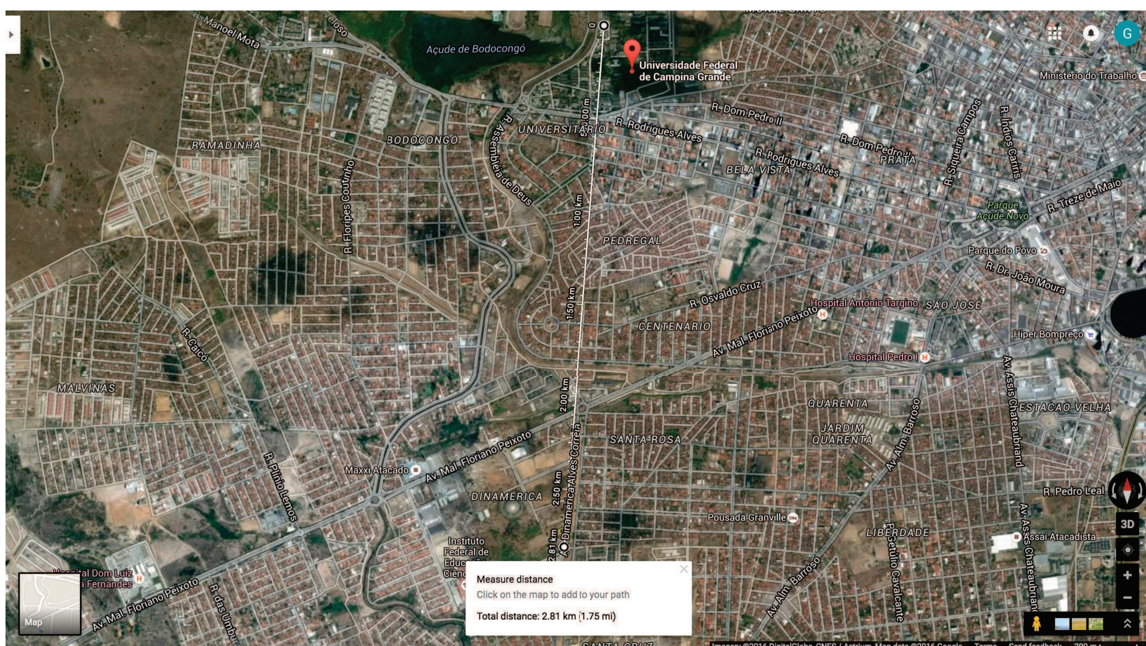


Figura 20 – Distância de transmissor e receptor no segundo teste de máxima configuração (GOOGLE MAPS, 2016).

7 Conclusão

A tecnologia LoRa™ vem conquistando mais adeptos a cada dia, e não é surpresa, dada a experiência durante este trabalho. Os resultados dos testes foram bastante satisfatórios, considerando o ambiente urbano onde eles se deram. O SX1272 se mostrou sempre um pouco superior ao Si4468 na comparação — utilizando como métrica o percentual de sucesso de recepção num mesmo ponto — para estas condições de testes, isso sem ao menos estar configurado para entregar o máximo de seu potencial. É importante ressaltar que o nível de ruído nas bandas de frequência utilizadas não era muito alto e que seriam necessários testes realizado num ambiente diferente para comparar o desempenho de ambos os *transceivers* sob esse ponto de vista. Outra condição na qual não foram realizados testes é a de chuva. Entre as sugestões para trabalhos futuros está, portanto, a avaliação do desempenho do SX1272 sob essas condições de alto nível de ruído e diferentes condições climáticas.

Os testes de máximo desempenho, por sua vez, deram clara vantagem ao SX1272, superando por mais de 1 km o alcance do Si4468. Desta forma, é possível concluir que, para aplicações num ambiente semelhante ao de teste, o *transceiver* da Semtech é superior.

Como todo o percurso de testes foi feito utilizando carro, e o nó receptor estava constantemente ligado, percebeu-se que o desempenho do SX1272 é muito variável quando em movimento. Muitas vezes o *link* se perdia quando em movimento, mas se reestabelecia com alta qualidade quando se parava no local de teste. Outra observação importante a se fazer é em relação ao percentual de sucesso de recepção, de forma que fica a impressão de que, uma vez estabelecida a conexão, pouquíssimos pacotes são entregues errados, e vice versa. Em outras palavras, ou se tem uma conexão muito boa, com percentual de sucesso altíssimo (sempre acima de 90%), ou uma conexão muito ruim, onde a maioria, ou totalidade, dos pacotes se perdem ou estão incorretos. Aparentemente, não há um meio termo.

Quanto às ferramentas desenvolvidas, ambas foram de grande valia para a operação do *transceiver* e funcionaram exatamente como esperado, podendo servir como ferramental de base para futuros estudos.

Referências

ABB. *What is a smart grid?* 2015. <http://new.abb.com/smartgrids/what-is-a-smart-grid>. Acesso 06 de Junho de 2016. Citado na página 18.

ASHTON, K. *That 'Internet of Things' Thing*. 2009. <http://www.rfidjournal.com/articles/view?4986>. Acesso 06 de Junho de 2016. Citado na página 17.

DIGI-KEY ELECTRONICS. *Semtech-Corporation SX1272RF1CAS*. 2016. <http://www.digikey.com/product-detail/en/semtech-corporation/SX1272RF1CAS/SX1272RF1CAS-ND/4490408>. Acesso 09 de Junho de 2016. Citado na página 36.

FOROUZAN, B. A. *Data Communications and Networking*. 5. ed. New York: McGraw-Hill, 2013. (McGraw-Hill Forouzan Networking). Citado 3 vezes nas páginas 8, 15 e 25.

GOOGLE MAPS. *eSMART/DEE/CEEI/UFCG*. 2016. <https://goo.gl/OwQaMb>. Acesso 09 de Junho de 2016. Citado 8 vezes nas páginas 8, 37, 38, 39, 40, 41, 42 e 43.

HARVARD BUSINESS REVIEW. *Internet of Things: Science Fiction or Business Fact*. [S.l.], 2014. Citado na página 17.

HUFFMAN, W. C.; PLESS, V. *Fundamentals of Error-Correcting Codes*. Cambridge: Cambridge University Press, 2003. Citado na página 24.

IEEE SMART GRID. *About IEEE Smart Grid*. 2016. <http://smartgrid.ieee.org/about-ieee-smart-grid>. Acesso 06 de Junho de 2016. Citado na página 18.

IOT AGENDA. *LPWAN (low-power wide area network)*. 2016. <http://internetofthingsagenda.techtarget.com/definition/LPWAN-low-power-wide-area-network>. Acesso 06 de Junho de 2016. Citado na página 18.

LORA ALLIANCE. *5-Minute Introduction to LoRa^{TM} Alliance*. 2016. <https://www.lora-alliance.org/What-Is-LoRa/LoRaWAN-Videos>. Acesso 07 de Junho de 2016. Citado na página 19.

LORA ALLIANCE. *About the LoRa^{TM} Alliance*. 2016. <https://www.lora-alliance.org/The-Alliance/About-the-Alliance>. Acesso 06 de Junho de 2016. Citado na página 19.

LORA ALLIANCE. *LoRa^{TM} Technology*. 2016. <https://www.lora-alliance.org/What-Is-LoRa/Technology>. Acesso 06 de Junho de 2016. Citado na página 19.

SEMTECH CORPORATION. *SX1272/73 - 860 MHz to 1020 MHz Low Power Long Range Transceiver Datasheet*. Camarillo, 2015. Citado 10 vezes nas páginas 8, 15, 20, 21, 22, 23, 24, 25, 29 e 35.

SEMTECH CORPORATION. *LoRa^{TM} FAQ*. 2016. <http://www.semtech.com/wireless-rf/lora/LoRa-FAQs.pdf>. Acesso 10 de Junho de 2016. Citado 2 vezes nas páginas 20 e 24.

SEMTECH CORPORATION. *LoRa^{TM} Product Family*. 2016. <http://www.semtech.com/wireless-rf/lora.html>. Acesso 05 de Junho de 2016. Citado na página 15.

SEMTECH CORPORATION. *Semtech Internet of Things (IoT)*. 2016. <http://www.semtech.com/wireless-rf/iot.html>. Acesso 06 de Junho de 2016. Citado na página 19.

SILICON LABORATORIES. *Si4468/7 - High-Performance, Low-Current Transceiver Datasheet*. Austin, 2014. Citado na página 35.

TEXAS INSTRUMENTS. *CC1121 High-Performance Low-Power RF Transceiver Datasheet*. Dallas, 2016. Citado na página 35.

TEXAS INSTRUMENTS. *MSP-EXP430G2*. 2016. <http://www.ti.com/ww/en/launchpad/launchpads-msp430-msp-exp430g2.html>. Acesso 09 de Junho de 2016. Citado 2 vezes nas páginas 8 e 27.

THE THINGS NETWORK. *The Things Network: What we do*. 2016. <https://thethingsnetwork.org/>. Acesso 07 de Junho de 2016. Citado na página 19.

THE THINGS NETWORK. *The Things Network Wiki: Home*. 2016. <https://staging.thethingsnetwork.org/wiki/Home>. Acesso 07 de Junho de 2016. Citado na página 19.

WOOD, A. *The internet of things is revolutionising our lives, but standards are a must*. 2015. <http://www.theguardian.com/media-network/2015/mar/31/the-internet-of-things-is-revolutionising-our-lives-but-standards-are-a-must>. Acesso 06 de Junho de 2016. Citado na página 17.

Apêndices

APÊNDICE A – Código da API Embarcada

Código A.1 – Padronização de tipos de variáveis

```
#include <stdint.h>
#include <stdbool.h>

typedef int8_t int8;
typedef uint8_t uint8;
typedef int16_t int16;
typedef uint16_t uint16;
typedef int32_t int32;
typedef uint32_t uint32;
typedef int64_t int63;
typedef uint64_t uint64;
typedef float float32;
typedef double float64;
typedef bool boolean;
```

Código A.2 – Lista de registradores do *transceiver* SX1272

```
enum LoraReg
{
    RegFifo = 0x00,
    RegOpMode = 0x01,
    RegFrfMsb = 0x06,
    RegFrfMib = 0x07,
    RegFrfLsb = 0x08,
    RegPaConfig = 0x09,
    RegPaRamp = 0x0A,
    RegOcp = 0x0B,
    RegLna = 0x0C,
    RegFifoAddrPtr = 0x0D,
    RegFifoTxBaseAddr = 0x0E,
    RegFifoRxBaseAddr = 0x0F,
    RegFifoRxCurrentAddr = 0x10,
    RegIrqFlagsMask = 0x11,
    RegIrqFlags = 0x12,
    RegRxNbBytes = 0x13,
    RegRxHeaderCntValueMsb = 0x14,
    RegRxHeaderCntValueLsb = 0x15,
    RegRxPacketCntValueMsb = 0x16,
    RegRxPacketCntValueLsb = 0x17,
    RegModemStat = 0x18,
    RegPktSnrValue = 0x19,
    RegPktRssiValue = 0x1A,
    RegRssiValue = 0x1B,
    RegHopChannel = 0x1C,
    RegModemConfig1 = 0x1D,
    RegModemConfig2 = 0x1E,
    RegSymbTimeoutLsb = 0x1F,
    RegPreambleMsb = 0x20,
    RegPreambleLsb = 0x21,
    RegPayloadLength = 0x22,
    RegMaxPayloadLength = 0x23,
```

```
    RegHopPeriod = 0x24,
    RegFifoRxByteAddr = 0x25,
    RegFeiMsb = 0x28,
    RegFeiMid = 0x29,
    RegFeiLsb = 0x2A,
    RegRssiWideband=0x2C,
    RegDetectOptimize =0x31,
    RegInvertIQ = 0x33,
    RegDetectionThreshold = 0x37
};
```

Código A.3 – Lista de valores constantes do *transceiver* SX1272

```
enum MODE
{
  FSKOOK = 0x00,
  LORA = 0x80
};

enum OPMODE
{
  SLEEP = 0x00,
  STDBY = 0x01,
  FSTX = 0x02,
  TX = 0x03,
  FSRX = 0x04,
  RXCONTINUOUS = 0x05,
  RXSINGLE = 0x06,
  CAD = 0x07
};

enum SPREADINGFACTOR
{
  SF6 = 0x06,
  SF7 = 0x07,
  SF8 = 0x08,
  SF9 = 0x09,
  SF10 = 0x0A,
  SF11 = 0x0B,
  SF12 = 0x0C
};

enum CODINGRATE
{
  CR45 = 0x01,
  CR46 = 0x02,
  CR47 = 0x03,
  CR48 = 0x04
};

enum BANDWIDTH
{
  BW125 = 0x00,
  BW250 = 0x01,
  BW500 = 0x02
};

enum HEADER
{
```

```
  EXPLICIT = 0x00,
  IMPLICIT = 0x01
};

//! Frequency channels
//!
//!      channel 10, central freq = 865.20MHz
//!      channel 11, central freq = 865.50MHz
//!      channel 12, central freq = 865.80MHz
//!      channel 13, central freq = 866.10MHz
//!      channel 14, central freq = 866.40MHz
//!      channel 15, central freq = 866.70MHz
//!      channel 16, central freq = 867.00MHz
//!      channel 00, central freq = 903.08MHz
//!      channel 01, central freq = 905.24MHz
//!      channel 02, central freq = 907.40MHz
//!      channel 03, central freq = 909.56MHz
//!      channel 04, central freq = 911.72MHz
//!      channel 05, central freq = 913.88MHz
//!      channel 06, central freq = 916.04MHz
//!      channel 07, central freq = 918.20MHz
//!      channel 08, central freq = 920.36MHz
//!      channel 09, central freq = 922.52MHz
//!      channel 10, central freq = 924.68MHz
//!      channel 11, central freq = 926.84MHz
//!      default channel 915MHz, the module is configured with it
enum FREQ_CHANNELS
{
  CH_10_868 = 0xD84CCC, // channel 10, central freq = 865.20MHz
  CH_11_868 = 0xD86000, // channel 11, central freq = 865.50MHz
  CH_12_868 = 0xD87333, // channel 12, central freq = 865.80MHz
  CH_13_868 = 0xD88666, // channel 13, central freq = 866.10MHz
  CH_14_868 = 0xD89999, // channel 14, central freq = 866.40MHz
  CH_15_868 = 0xD8ACCC, // channel 15, central freq = 866.70MHz
  CH_16_868 = 0xD8C000, // channel 16, central freq = 867.00MHz
  CH_17_868 = 0xD90000, // channel 16, central freq = 868.00MHz
  CH_00_900 = 0xE1C51E, // channel 00, central freq = 903.08MHz
  CH_01_900 = 0xE24F5C, // channel 01, central freq = 905.24MHz
  CH_02_900 = 0xE2D999, // channel 02, central freq = 907.40MHz
  CH_03_900 = 0xE363D7, // channel 03, central freq = 909.56MHz
  CH_04_900 = 0xE3EE14, // channel 04, central freq = 911.72MHz
  CH_05_900 = 0xE47851, // channel 05, central freq = 913.88MHz
  CH_06_900 = 0xE5028F, // channel 06, central freq = 916.04MHz
  CH_07_900 = 0xE58CCC, // channel 07, central freq = 918.20MHz
  CH_08_900 = 0xE6170A, // channel 08, central freq = 920.36MHz
  CH_09_900 = 0xE6A147, // channel 09, central freq = 922.52MHz
  CH_10_900 = 0xE72B85, // channel 10, central freq = 924.68MHz
  CH_11_900 = 0xE7B5C2, // channel 11, central freq = 926.84MHz
  CH_12_900 = 0xE4C000  // default channel 915MHz, the module is configured with it
};

enum CRC_MODE
{
  CRC_OFF = 0x00,
  CRC_ON = 0x01
};

enum LDR_OPTMIZE
{
  LDR_OFF = 0x00,
```

```
  LDR_ON = 0x01
};

enum LOW_PLL
{
  LOW_PLL_TX_OFF = (0x00 << 4),
  LOW_PLL_TX_ON = (0x01 << 4)
};

enum PA_RAMP
{
  RFT_3400 = 0x00,
  RFT_2000 = 0x01,
  RFT_1000 = 0x02,
  RFT_500 = 0x03,
  RFT_250 = 0x04,
  RFT_125 = 0x05,
  RFT_100 = 0x06,
  RFT_62 = 0x07,
  RFT_50 = 0x08,
  RFT_40 = 0x09,
  RFT_31 = 0x0A,
  RFT_25 = 0x0B,
  RFT_20 = 0x0C,
  RFT_15 = 0x0D,
  RFT_12 = 0x0E,
  RFT_10 = 0x0F
};

enum OCP
{
  OCP_OFF = (0x00 << 5),
  OCP_ON = (0x01 << 5),
  OCP_TRIM_100 = 0x0B, // Default
  OCP_TRIM_120 = 0x1B  // Max
};

enum LNA_GAIN
{
  G1 = (0x01 << 5),
  G2 = (0x02 << 5),
  G3 = (0x03 << 5),
  G4 = (0x04 << 5),
  G5 = (0x05 << 5),
  G6 = (0x06 << 5)
};

enum LNA_BOOST
{
  BOOST_OFF = 0x00,
  BOOST_ON = 0x03
};

enum AGC
{
  AGC_OFF = 0x00,
  AGC_ON = 0x01
};

enum RX_TIMEOUT
```

```
{
  MAX_RX_TIMEOUT = 0x03FF
};

enum RF_POWER_MODE
{
  MAX_POWER  = 0x8F,
  HIGH_POWER = 0x87,
  LOW_POWER  = 0x80
};

enum AuxiliaryValues
{
  TX_DONE_MASK = 0x08,
  PAYLOAD_CRC_ERROR_MASK = 0x20,
  RX_DONE_MASK = 0x40,
  RX_TIMEOUT_MASK = 0x80,
  FIFO_STARTING_ADDRESS = 0x80,
  REMOTE_CONFIG_SIZE_INSTR = 3,
  REMOTE_CONFIG_QNT_INSTR = 7,
  UART_STRING_LENGTH = 8,
  REMOTE_STATUS_SIZE_INSTR = 2,
  REMOTE_STATUS_STRING_LENGTH = 11,
  MAX_PAYLOAD_LENGTH = 127
};
```

Código A.4 – Protocolo de interface GUI/MSP430

```
enum INTERFACE_OPMODE
{
  INTERFACE_TX = 0,
  INTERFACE_RX = 1
};

enum INTERFACE_SPREADINGFACTOR
{
  INTERFACE_SF6 = 1,
  INTERFACE_SF7 = 2,
  INTERFACE_SF8 = 3,
  INTERFACE_SF9 = 4,
  INTERFACE_SF10 = 5,
  INTERFACE_SF11 = 6,
  INTERFACE_SF12 = 7
};

enum INTERFACE_CODINGRATE
{
  INTERFACE_CR45 = 1,
  INTERFACE_CR46 = 2,
  INTERFACE_CR47 = 3,
  INTERFACE_CR48 = 4
};

enum INTERFACE_BANDWIDTH
{
  INTERFACE_BW125 = 1,
  INTERFACE_BW250 = 2,
  INTERFACE_BW500 = 3
};
```

```
enum INTERFACE_HEADER
{
  INTERFACE_EXPLICIT = 0,
  INTERFACE_IMPLICIT = 1
};

enum INTERFACE_FREQ_CHANNELS
{
  INTERFACE_CH_10_868 = 1, // channel 10, central freq = 865.20MHz
  INTERFACE_CH_11_868 = 2, // channel 11, central freq = 865.50MHz
  INTERFACE_CH_12_868 = 3, // channel 12, central freq = 865.80MHz
  INTERFACE_CH_13_868 = 4, // channel 13, central freq = 866.10MHz
  INTERFACE_CH_14_868 = 5, // channel 14, central freq = 866.40MHz
  INTERFACE_CH_15_868 = 6, // channel 15, central freq = 866.70MHz
  INTERFACE_CH_16_868 = 7, // channel 16, central freq = 867.00MHz
  INTERFACE_CH_17_868 = 8, // channel 16, central freq = 868.00MHz
  INTERFACE_CH_00_900 = 9, // channel 00, central freq = 903.08MHz
  INTERFACE_CH_01_900 = 10, // channel 01, central freq = 905.24MHz
  INTERFACE_CH_02_900 = 11, // channel 02, central freq = 907.40MHz
  INTERFACE_CH_03_900 = 12, // channel 03, central freq = 909.56MHz
  INTERFACE_CH_04_900 = 13, // channel 04, central freq = 911.72MHz
  INTERFACE_CH_05_900 = 14, // channel 05, central freq = 913.88MHz
  INTERFACE_CH_06_900 = 15, // channel 06, central freq = 916.04MHz
  INTERFACE_CH_07_900 = 16, // channel 07, central freq = 918.20MHz
  INTERFACE_CH_08_900 = 17, // channel 08, central freq = 920.36MHz
  INTERFACE_CH_09_900 = 18, // channel 09, central freq = 922.52MHz
  INTERFACE_CH_10_900 = 19, // channel 10, central freq = 924.68MHz
  INTERFACE_CH_11_900 = 20, // channel 11, central freq = 926.84MHz
  INTERFACE_CH_12_900 = 21  // default channel 915MHz, the module is configured with it
};

enum INTERFACE_CRC_MODE
{
  INTERFACE_CRC_OFF = 0,
  INTERFACE_CRC_ON = 1
};

enum INTERFACE_LDR_OPTMIZE
{
  INTERFACE_LDR_OFF = 0,
  INTERFACE_LDR_ON = 1
};

enum INTERFACE_AGC
{
  INTERFACE_AGC_OFF = 0,
  INTERFACE_AGC_ON = 1
};

enum INTERFACE_RF_POWER_MODE
{
  INTERFACE_MAX_POWER  = 0,
  INTERFACE_HIGH_POWER = 1,
  INTERFACE_LOW_POWER  = 2
};
```

Código A.5 – Cabeçalho da classe TMsp430UART

```
#include "msp430f6736.h"
#include "Padronizacao.h"

#ifndef TMSP430UART_H
#define TMSP430UART_H

class TMsp430UART
{
private:
  volatile static boolean _bTxFlag;
  volatile static boolean _bRxFlag;

  volatile static uint8 _ucTxChar;
  volatile static uint8 _ucRxChar;

public:
  TMsp430UART(void);

  void InitUART(void);

  static boolean GetTxFlag();
  static void SetTxFlag(boolean bTxFlag);

  static boolean GetRxFlag();
  static void SetRxFlag(boolean bRxFlag);

  static uint8 GetTxChar();
  static void SetTxChar(uint8 ucTxChar);

  static uint8 GetRxChar();
  static void SetRxChar(uint8 ucRxChar);

  uint8 GetChar(void);
  void PutChar(uint8 ucChar);

  void GetString(uint8* pString, uint16 uiStringLength);
  void PutString(uint8* pString);

  static void EnableRxInterrupt(void);
  static void DisableRxInterrupt(void);

  static void EnableTxInterrupt(void);
  static void DisableTxInterrupt(void);
};

#endif
```

Código A.6 – Implementação da classe TMsp430UART

```
#include "HdsMsp430UART.h"

volatile boolean TMsp430UART::_bTxFlag = true;
volatile boolean TMsp430UART::_bRxFlag = false;
volatile uint8 TMsp430UART::_ucTxChar = '\0';
volatile uint8 TMsp430UART::_ucRxChar = '\0';

TMsp430UART::TMsp430UART(void)
{
  InitUART();
```

```
}

void TMsp430UART::InitUART(void)
{
   //! Configura pinos de i/o
  P1SEL |= (BIT4 + BIT5);

  //! Ajusta fonte de clock
  UCA1CTL1 = UCSSEL_2 + UCSWRST + UCRXEIE /*+ UCBRKIE*/;

  //! Sem paridade, 8 bits de dados, 1 stop bit, UART.
  UCA1CTL0 = 0;

  // Use low-frequency baud-generator. Set to 9600bps.
  UCA1BR0 = (uint16) 81;
  UCA1BR1 = 00;

  UCA1MCTLW = (/*UCBRF_8 +*/ UCOS16); //!< Activate over-sampling baud-rate mode.

  UCA1CTL1 &= ~UCSWRST;
  UCA1IE |= UCTXIE | UCRXIE; //enable rx interrupt

  SetTxFlag(true);
  SetRxFlag(false);

  return;
}

boolean TMsp430UART::GetTxFlag()
{
  return _bTxFlag;
}

void TMsp430UART::SetTxFlag(boolean bTxFlag)
{
  _bTxFlag = bTxFlag;
  return;
}

boolean TMsp430UART::GetRxFlag()
{
  return _bRxFlag;
}

void TMsp430UART::SetRxFlag(boolean bRxFlag)
{
  _bRxFlag = bRxFlag;
  return;
}

uint8 TMsp430UART::GetTxChar()
{
  return _ucTxChar;
}

void TMsp430UART::SetTxChar(uint8 ucTxChar)
{
  _ucTxChar = ucTxChar;
  return;
}
```

```
uint8 TMsp430UART::GetRxChar()
{
  return _ucRxChar;
}

void TMsp430UART::SetRxChar(uint8 ucRxChar)
{
  _ucRxChar = ucRxChar;
  return;
}

uint8 TMsp430UART::GetChar(void)
{
  while (!GetRxFlag());
  SetRxFlag(false);
  return GetRxChar();
}

void TMsp430UART::PutChar(uint8 ucChar)
{
  SetTxChar(ucChar);
  EnableTxInterrupt();
  while (GetTxFlag());
  SetTxFlag(true);

  return;
}

void TMsp430UART::GetString(uint8* pString, uint16 uiStringLength)
{
  uint16 i = 0x00;

  while (i < uiStringLength)
  {
    pString[i] = GetChar();

    if (pString[i] == '\r')
    {
      for ( ; i < uiStringLength; i++)
        pString[i] = '\0';
    }

    i++;
  }

  return;
}

void TMsp430UART::PutString(uint8* pString)
{
  uint8 uiCounter = 0;
  boolean bEndFlag = false;

  while (!bEndFlag)
  {
    PutChar(pString[uiCounter]);

    if (pString[uiCounter] == '\n')
    {
```

```
      bEndFlag = true;
    }
    else
    {
      uiCounter++;
    }
  }

  return;
}

void TMsp430UART::EnableRxInterrupt(void)
{
  UCA1IE |= UCRXIE;
}

void TMsp430UART::DisableRxInterrupt(void)
{
  UCA1IE &= ~UCRXIE;
}

void TMsp430UART::EnableTxInterrupt(void)
{
  UCA1IE |= UCTXIE;
}

void TMsp430UART::DisableTxInterrupt(void)
{
  UCA1IE &= ~UCTXIE;
}

#pragma vector = USCI_A1_VECTOR
__interrupt void USCI_A1_ISR(void)
{
  switch (__even_in_range(UCA1IV, 8))
  {
    // No interrupt
    case USCI_NONE:
      break;

    // RXIFG
    case USCI_UART_UCRXIFG:
      TMsp430UART::SetRxChar(UCA1RXBUF);
      TMsp430UART::SetRxFlag(true);
      break;

    // TXIFG
    case USCI_UART_UCTXIFG:
      UCA1TXBUF = TMsp430UART::GetTxChar();
      TMsp430UART::SetTxFlag(false);
      TMsp430UART::DisableTxInterrupt();
      break;

    // TTIFG
    case USCI_UART_UCSTTIFG:
      break;

    // TXCPTIFG
    case USCI_UART_UCTXCPTIFG:
      break;
```

```
    default:
      break;
  }
}
```

Código A.7 – Cabeçalho da classe TMsp430SPI

```
#include "msp430f6736.h"
#include "Padronizacao.h"

#ifndef TMSP430SPI_H
#define TMSP430SPI_H

class TMsp430SPI
{
private:
  volatile static boolean _bTxFlag;
  volatile static boolean _bRxFlag;

  volatile static uint8 _ucTxChar;
  volatile static uint8 _ucRxChar;

public:
  TMsp430SPI(void);

  void InitSPI(void);

  static boolean GetTxFlag();
  static void SetTxFlag(boolean bTxFlag);

  static boolean GetRxFlag();
  static void SetRxFlag(boolean bRxFlag);

  static uint8 GetTxChar();
  static void SetTxChar(uint8 ucTxChar);

  static uint8 GetRxChar();
  static void SetRxChar(uint8 ucRxChar);

  uint8 GetChar(void);
  void PutChar(uint8 ucChar);

  static void EnableRxInterrupt(void);
  static void DisableRxInterrupt(void);

  static void EnableTxInterrupt(void);
  static void DisableTxInterrupt(void);
};

#endif
```

Código A.8 – Implementação da classe TMsp430SPI

```
#include "HdsMsp430SPI.h"

volatile boolean TMsp430SPI::_bTxFlag = true;
volatile boolean TMsp430SPI::_bRxFlag = false;
volatile uint8 TMsp430SPI::_ucTxChar = '\0';
```

```
volatile uint8 TMsp430SPI::_ucRxChar = '\0';

TMsp430SPI::TMsp430SPI(void)
{
  InitSPI();
}

void TMsp430SPI::InitSPI(void)
{
  P2OUT = 0x00;
  P2DIR |= BIT6 | BIT7; // Set port 2.7 as as output (FEM_CTX) & 2.6 as RESET of Sx1272
  P2DIR |=  BIT4 ;      // NSS
  P3OUT &= ~BIT0;

  // enable port mapping
  PMAPPWD = 0x02D52;
  PMAPCTL = 0x02;

  UCA2CTLW0 |= UCSWRST;
  P2SEL |=  BIT2 |  BIT3 | BIT5; // 2.2 SOMI, 2.3 SIMO, 2.5 SCK
  UCSCTL4|= SELS__VLOCLK;
  UCA2CTLW0 = UCMSB | UCMST | UCSYNC | UCCKPH |UCSSEL__SMCLK  ; // 3-pin, 8-bit SPI master
  UCA2CTLW0 &= ~UCSWRST;
  UCA2BRW_L = 0x08;
  UCA2BRW_H = 0x00;

  P2OUT &= ~BIT4;                 // Now with SPI signals initialized,
  P2OUT |= BIT4;                  // reset slave

  UCA2IE |= UCRXIE; //enable rx interrupt

  SetTxFlag(true);
  SetRxFlag(false);
}

boolean TMsp430SPI::GetTxFlag()
{
  return _bTxFlag;
}

void TMsp430SPI::SetTxFlag(boolean bTxFlag)
{
  _bTxFlag = bTxFlag;
  return;
}

boolean TMsp430SPI::GetRxFlag()
{
  return _bRxFlag;
}

void TMsp430SPI::SetRxFlag(boolean bRxFlag)
{
  _bRxFlag = bRxFlag;
  return;
}

uint8 TMsp430SPI::GetTxChar()
{
  return _ucTxChar;
```

```
}

void TMsp430SPI::SetTxChar(uint8 ucTxChar)
{
  _ucTxChar = ucTxChar;
  return;
}

uint8 TMsp430SPI::GetRxChar()
{
  return _ucRxChar;
}

void TMsp430SPI::SetRxChar(uint8 ucRxChar)
{
  _ucRxChar = ucRxChar;
  return;
}

uint8 TMsp430SPI::GetChar(void)
{
  while (!GetRxFlag());
  SetRxFlag(false);
  return GetRxChar();
}

void TMsp430SPI::PutChar(uint8 ucChar)
{
  SetTxChar(ucChar);
  EnableTxInterrupt();
  while (GetTxFlag());
  SetTxFlag(true);

  return;
}

void TMsp430SPI::EnableRxInterrupt(void)
{
  UCA2IE |= UCRXIE;
}

void TMsp430SPI::DisableRxInterrupt(void)
{
  UCA2IE &= ~UCRXIE;
}

void TMsp430SPI::EnableTxInterrupt(void)
{
  UCA2IE |= UCTXIE;
}

void TMsp430SPI::DisableTxInterrupt(void)
{
  UCA2IE &= ~UCTXIE;
}

#pragma vector = USCI_A2_VECTOR
__interrupt void USCI_A2_ISR(void)
{
  switch (__even_in_range(UCA2IV, 8))
```

```
  {
    // RXIFG
    case USCI_SPI_UCRXIFG:
      TMsp430SPI::SetRxChar(UCA2RXBUF);
      TMsp430SPI::SetRxFlag(true);
      break;

    // TXIFG
    case USCI_SPI_UCTXIFG:
      UCA2TXBUF = TMsp430SPI::GetTxChar();
      TMsp430SPI::SetTxFlag(false);
      TMsp430SPI::DisableTxInterrupt();
      break;

    default:
      break;
  }
}
```

Código A.9 – Cabeçalho da classe TSx1272SPI

```
#include "HdsMsp430SPI.h"

#ifndef TSX1272SPI_H
#define TSX1272SPI_H

class TSx1272SPI
{
private:
  TMsp430SPI _SPI;
  volatile uint8 _ucDataRead;

public:
  TSx1272SPI(void);

  void EnableNSS(void);
  void DisableNSS(void);

  uint8 EnableWNR(uint8 ucAddress);
  uint8 DisableWNR(uint8 ucAddress);

  uint8 ReadByteFromRegister(uint8 ucAddress);
  void WriteByteToRegister(uint8 ucAddress, uint8 ucData);
};

#endif
```

Código A.10 – Implementação da classe TSx1272SPI

```
#include "HdsSx1272SPI.h"

TSx1272SPI::TSx1272SPI(void) {}

void TSx1272SPI::EnableNSS(void)
{
  P2OUT |= BIT4;
}

void TSx1272SPI::DisableNSS(void)
```

```
{
  P2OUT &= ~BIT4;
}

uint8 TSx1272SPI::EnableWNR(uint8 ucAddress)
{
  ucAddress |= BIT7;

  return ucAddress;
}

uint8 TSx1272SPI::DisableWNR(uint8 ucAddress)
{
  ucAddress &= ~BIT7;

  return ucAddress;
}

uint8 TSx1272SPI::ReadByteFromRegister(uint8 ucAddress)
{
  ucAddress = DisableWNR(ucAddress);
  DisableNSS();

  _SPI.PutChar(ucAddress);
  _SPI.GetChar();

  _SPI.PutChar(0x00);
  _ucDataRead = _SPI.GetChar();

  EnableNSS();

  return _ucDataRead;
}

void TSx1272SPI::WriteByteToRegister(uint8 ucAddress, uint8 ucData)
{
  ucAddress = EnableWNR(ucAddress);
  DisableNSS();

  _SPI.PutChar(ucAddress);
  _SPI.GetChar();

  _SPI.PutChar(ucData);
  _SPI.GetChar();

  EnableNSS();
}
```

Código A.11 – Cabeçalho da classe TSx1272

```
#include "HdsSx1272SPI.h"
#include "HdsLoraReg.h"
#include "HdsSx1272ConstValues.h"

#ifndef TSX1272LORA_H
#define TSX1272LORA_H

class TSx1272
{
```

```
private:
  TSx1272SPI _SPI;

  uint8 _ucMode;
  uint8 _ucRegIrqFlagsCopy;

  uint8 _ucFifoRxBaseAddr;
  uint8 _ucFifoTxBaseAddr;

  uint8 _ucPaRamp;
  uint8 _ucOcp;
  uint8 _ucLna;

  void _SetPaRamp(void);
  boolean _CheckPaRamp(void);

  void _SetOcp(void);
  boolean _CheckOcp(void);

  void _SetLna(void);
  boolean _CheckLna(void);

public:

  TSx1272(void);
  TSx1272(uint8 ucMode);

  void InitLoRaMode(void);
  void InitFSKOOKMode(void);

  void SetOpMode(uint8 ucOpMode);
  uint8 GetOpMode(void);

  void SetSpreadingFactor(uint8 ucSpreadingFactor);
  uint8 GetSpreadingFactor(void);

  void SetCodingRate(uint8 ucCodingRate);
  uint8 GetCodingRate(void);

  void SetHeaderMode(uint8 ucHeaderMode);
  uint8 GetHeaderMode(void);

  uint8 GetCrcOnPayload(void);

  void SetBandwidth(uint8 ucBandwidth);
  uint8 GetBandwidth(void);

  void SetFreqChannel(uint32 ulFreqChannel);
  uint32 GetFreqChannel(void);

  void SetRxPayloadCrcOn(uint8 ucCrcMode);
  uint8 GetRxPayloadCrcOn(void);

  void SetLowDataRateOptimize(uint8 ucLDRMode);
  uint8 GetLowDataRateOptimize(void);

  void SetPower(uint8 ucPowerMode);
  boolean CheckPower(uint8 ucPowerMode);

  void SetAutoGainControl(uint8 ucAutoGainControl);
```

```
  uint8 GetAutoGainControl(void);

  void SetPayloadLength(uint8 ucPayloadLength);
  uint8 GetPayloadLength(void);

  void SetRxTimeout(uint16 uiSymbTimeout);
  uint16 GetRxTimeout(void);

  uint8 GetRxNbBytes(void);

  void ClearIrqFlags(void);
  uint8 GetIrqFlags(void);

  void WriteToFifo(uint8* ucData, uint8 ucLength);
  void ReadFromFifo(uint8* ucData, uint8 ucLength);

  void ResetFifo(void);

  boolean Transmit(uint8 *ucData, uint8 ucLength);
  int8 ReceiveSingle(uint8 ucHeaderMode);

  void ActivateFEMCTX(void);
  void DeactivateFEMCTX(void);
};

#endif
```

Código A.12 – Implementação da classe TSx1272

```
#include "HdsSx1272.h"
#include "HdsMsp430UART.h"

TSx1272::TSx1272(void)
{
  _ucMode = LORA;

  _ucPaRamp = (LOW_PLL_TX_ON | RFT_40);
  _ucOcp = (OCP_ON | OCP_TRIM_120);
  _ucLna = (G1 | BOOST_ON);

  InitLoRaMode();
}

TSx1272::TSx1272(uint8 ucMode)
{
  _ucMode = ucMode;

  _ucFifoRxBaseAddr = 0x00;
  _ucFifoTxBaseAddr = 0x80;

  _ucPaRamp = (LOW_PLL_TX_ON | RFT_40);
  _ucOcp = (OCP_ON | OCP_TRIM_120);
  _ucLna = (G1 | BOOST_ON);

  if (ucMode == LORA)
    InitLoRaMode();
  else
    InitFSKOOKMode();
}
```

```
void TSx1272::InitLoRaMode()
{
  _SPI.WriteByteToRegister(RegOpMode, SLEEP);
  _SPI.WriteByteToRegister(RegOpMode, (LORA | SLEEP));

  _SPI.WriteByteToRegister(RegFifoRxBaseAddr, _ucFifoRxBaseAddr);
  _SPI.WriteByteToRegister(RegFifoTxBaseAddr, _ucFifoTxBaseAddr);
}

void TSx1272::InitFSKOOKMode()
{
  _SPI.WriteByteToRegister(RegOpMode, SLEEP);
  _SPI.WriteByteToRegister(RegOpMode, (FSKOOK | SLEEP));
}

void TSx1272::SetOpMode(uint8 ucOpMode)
{
  _SPI.WriteByteToRegister(RegOpMode, (_ucMode | ucOpMode));
}

uint8 TSx1272::GetOpMode(void)
{
  return _SPI.ReadByteFromRegister(RegOpMode);
}

void TSx1272::SetSpreadingFactor(uint8 ucSpreadingFactor)
{
  uint8 temp = 0x00;

  SetOpMode(STDBY);

  // Setting Spreading Factor
  temp |= _SPI.ReadByteFromRegister(RegModemConfig2);
  temp &= (~(BIT7 | BIT6 | BIT5 | BIT4)); // Mask
  temp |= (ucSpreadingFactor << 4);

  _SPI.WriteByteToRegister(RegModemConfig2, temp);

  if (ucSpreadingFactor == SF6)
  {
    // Setting Detection Optimize
    temp |= _SPI.ReadByteFromRegister(RegDetectOptimize);
    temp &= (~(BIT2 | BIT1 | BIT0)); // Mask
    temp |= 0x05;

    _SPI.WriteByteToRegister(RegDetectOptimize, temp);

    // Setting Detection Threshold
    _SPI.WriteByteToRegister(RegDetectionThreshold, 0x0C);
  }
  else
  {
    // Setting Detection Optimize
    temp |= _SPI.ReadByteFromRegister(RegModemConfig2);
    temp &= (~(BIT2 | BIT1 | BIT0)); // Mask
    temp |= 0x03;

    _SPI.WriteByteToRegister(RegDetectOptimize, temp);
```

```
    // Setting Detection Threshold
    _SPI.WriteByteToRegister(RegDetectionThreshold, 0x0A);
  }
}

uint8 TSx1272::GetSpreadingFactor(void)
{
  uint8 temp = 0x00;

  temp |= _SPI.ReadByteFromRegister(RegModemConfig2);

  return (temp >> 4);
}

void TSx1272::SetCodingRate(uint8 ucCodingRate)
{
  uint8 temp = 0x00;

  SetOpMode(STDBY);

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= (~(BIT5 | BIT4 | BIT3)); // Mask
  temp |= (ucCodingRate << 3);

  _SPI.WriteByteToRegister(RegModemConfig1, temp);
}

uint8 TSx1272::GetCodingRate(void)
{
  uint8 temp = 0x00;

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= (BIT5 | BIT4 | BIT3); // Mask

  return (temp >> 3);
}

void TSx1272::SetHeaderMode(uint8 ucHeaderMode)
{
  uint8 temp = 0x00;

  SetOpMode(STDBY);

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= (~BIT2); // Mask
  temp |= (ucHeaderMode << 2);

  _SPI.WriteByteToRegister(RegModemConfig1, temp);
}

uint8 TSx1272::GetHeaderMode(void)
{
  uint8 temp = 0x00;

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= BIT2; // Mask

  return (temp >> 2);
}
```

```
uint8 TSx1272::GetCrcOnPayload(void)
{
  uint8 temp = 0x00;

  temp |= _SPI.ReadByteFromRegister(RegHopChannel);
  temp &= BIT6; // Mask

  return (temp >> 6);
}

void TSx1272::SetBandwidth(uint8 ucBandwidth)
{
  uint8 temp = 0x00;

  SetOpMode(STDBY);

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= (~(BIT7 | BIT6));
  temp |= (ucBandwidth << 6);

  _SPI.WriteByteToRegister(RegModemConfig1, temp);
}

uint8 TSx1272::GetBandwidth(void)
{
  uint8 temp = 0x00;

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= (BIT7 | BIT6); // Mask

  return (temp >> 6);
}

void TSx1272::SetFreqChannel(uint32 ulFreqChannel)
{
  uint8 temp = 0x00;

  SetOpMode(STDBY);

  // LSB
  temp = ulFreqChannel;
  _SPI.WriteByteToRegister(RegFrfLsb, temp);

  // MIB
  temp = (ulFreqChannel >> 8);
  _SPI.WriteByteToRegister(RegFrfMib, temp);

  // MSB
  temp = (ulFreqChannel >> 16);
  _SPI.WriteByteToRegister(RegFrfMsb, temp);
}

uint32 TSx1272::GetFreqChannel(void)
{
  uint32 reg = 0x00000000;
  uint32 temp = 0x00000000;

  reg = _SPI.ReadByteFromRegister(RegFrfMsb);
  temp |= (reg << 16);
```

```
  reg = _SPI.ReadByteFromRegister(RegFrfMib);
  temp |= (reg << 8);

  reg = _SPI.ReadByteFromRegister(RegFrfLsb);
  temp |= reg;

  return temp;
}

void TSx1272::SetRxPayloadCrcOn(uint8 ucCrcMode)
{
  uint8 temp = 0x00;

  SetOpMode(STDBY);

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= (~BIT1); // Mask
  temp |= (ucCrcMode << 1);

  _SPI.WriteByteToRegister(RegModemConfig1, temp);
}

uint8 TSx1272::GetRxPayloadCrcOn(void)
{
  uint8 temp = 0x00;

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= BIT1; // Mask

  return (temp >> 1);
}

void TSx1272::SetLowDataRateOptimize(uint8 ucLDRMode)
{
  uint8 temp = 0x00;

  SetOpMode(STDBY);

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= (~BIT0); // Mask
  temp |= (ucLDRMode);

  _SPI.WriteByteToRegister(RegModemConfig1, temp);
}

uint8 TSx1272::GetLowDataRateOptimize(void)
{
  uint8 temp = 0x00;

  temp |= _SPI.ReadByteFromRegister(RegModemConfig1);
  temp &= BIT0; // Mask

  return temp;
}

void TSx1272::SetPower(uint8 ucPowerMode)
{
  SetOpMode(STDBY);

  _SetPaRamp();
```

```
  _SetOcp();
  _SetLna();

  _SPI.WriteByteToRegister(RegPaConfig, ucPowerMode);
}

boolean TSx1272::CheckPower(uint8 ucPowerMode)
{
  if (_CheckPaRamp() && _CheckOcp() && _CheckLna() &&
        (_SPI.ReadByteFromRegister(RegPaConfig) == ucPowerMode))
    return true;
  else
    return false;
}

void TSx1272::_SetPaRamp(void)
{
  _SPI.WriteByteToRegister(RegPaRamp, _ucPaRamp);
}

boolean TSx1272::_CheckPaRamp(void)
{
  if (_SPI.ReadByteFromRegister(RegPaRamp) == _ucPaRamp)
    return true;
  else
    return false;
}

void TSx1272::_SetOcp(void)
{
  _SPI.WriteByteToRegister(RegOcp, _ucOcp);
}

boolean TSx1272::_CheckOcp(void)
{
  if (_SPI.ReadByteFromRegister(RegOcp) == _ucOcp)
    return true;
  else
    return false;
}

void TSx1272::_SetLna(void)
{
  _SPI.WriteByteToRegister(RegLna, _ucLna);
}

boolean TSx1272::_CheckLna(void)
{
  if (_SPI.ReadByteFromRegister(RegLna) == _ucLna)
    return true;
  else
    return false;
}

void TSx1272::SetAutoGainControl(uint8 ucAutoGainControl)
{
  uint8 temp = 0x00;

  temp |= _SPI.ReadByteFromRegister(RegModemConfig2);
  temp &= (~BIT2); // Mask
```

```
  temp |= (ucAutoGainControl << 2);

  _SPI.WriteByteToRegister(RegModemConfig2, temp);
}

uint8 TSx1272::GetAutoGainControl(void)
{
  uint8 temp = 0x00;

  temp |= _SPI.ReadByteFromRegister(RegModemConfig2);
  temp &= BIT2; // Mask

  return (temp >> 2);
}

void TSx1272::SetPayloadLength(uint8 ucPayloadLength)
{
  SetOpMode(STDBY);

  _SPI.WriteByteToRegister(RegPayloadLength, ucPayloadLength);
}

uint8 TSx1272::GetPayloadLength(void)
{
  return _SPI.ReadByteFromRegister(RegPayloadLength);
}

void TSx1272::SetRxTimeout(uint16 uiSymbTimeout)
{
  uint8 temp = 0x00;

  _SPI.WriteByteToRegister(RegSymbTimeoutLsb, uiSymbTimeout);

  temp |= _SPI.ReadByteFromRegister(RegModemConfig2);
  temp &= (~(BIT1 | BIT0));
  temp |= (uiSymbTimeout >> 8);

  _SPI.WriteByteToRegister(RegModemConfig2, temp);
}

uint16 TSx1272::GetRxTimeout(void)
{
  uint8 reg = 0x00;
  uint16 temp = 0x0000;

  temp |= _SPI.ReadByteFromRegister(RegSymbTimeoutLsb);

  reg |= _SPI.ReadByteFromRegister(RegModemConfig2);
  reg &= (BIT1 | BIT0);

  temp |= (reg << 8);

  return temp;
}

uint8 TSx1272::GetRxNbBytes(void)
{
  return _SPI.ReadByteFromRegister(RegRxNbBytes);
}
```

```
void TSx1272::ClearIrqFlags(void)
{
  _SPI.WriteByteToRegister(RegIrqFlags, GetIrqFlags());
}

uint8 TSx1272::GetIrqFlags(void)
{
  return _SPI.ReadByteFromRegister(RegIrqFlags);
}

void TSx1272::WriteToFifo(uint8* ucData, uint8 ucLength)
{
  uint8 temp = 0x00;

  _SPI.WriteByteToRegister(RegFifoAddrPtr, _ucFifoTxBaseAddr);

  for (temp = 0; temp < ucLength; temp++)
  {
    _SPI.WriteByteToRegister(RegFifo, ucData[temp]);
  }
}

void TSx1272::ReadFromFifo(uint8* ucData, uint8 ucLength)
{
  uint8 temp = 0x00;

  _SPI.WriteByteToRegister(RegFifoAddrPtr, _ucFifoRxBaseAddr);

  for (temp = 0; temp < ucLength; temp++)
  {
    ucData[temp] = _SPI.ReadByteFromRegister(RegFifo);
  }
}

void TSx1272::ResetFifo(void)
{
  _SPI.WriteByteToRegister(RegFifoAddrPtr, 0x00);

  for (uint16 i = 0; i < 0x0100; i++)
  {
    _SPI.WriteByteToRegister(RegFifo, 0x00);
  }
}

boolean TSx1272::Transmit(uint8 *ucData, uint8 ucLength)
{
  uint16 uiCounter = 0x0000;

  ActivateFEMCTX();

  SetOpMode(STDBY);
  WriteToFifo(ucData, ucLength);
  SetOpMode(TX);

  do
  {
    uiCounter++;

    _ucRegIrqFlagsCopy = GetIrqFlags();
  } while (((_ucRegIrqFlagsCopy & TX_DONE_MASK) != TX_DONE_MASK) && uiCounter < 1000);
```

```
  ClearIrqFlags();

  DeactivateFEMCTX();

  if ((_ucRegIrqFlagsCopy & TX_DONE_MASK) == TX_DONE_MASK)
    return true;
  else
    return false;
}

int8 TSx1272::ReceiveSingle(uint8 ucHeaderMode)
{
  DeactivateFEMCTX();
  SetOpMode(RXSINGLE);

  do
  {
    _ucRegIrqFlagsCopy = GetIrqFlags();
  } while (((_ucRegIrqFlagsCopy & RX_DONE_MASK) != RX_DONE_MASK) &&
        ((_ucRegIrqFlagsCopy & RX_TIMEOUT_MASK) != RX_TIMEOUT_MASK) &&
        !TMsp430UART::GetRxFlag());

  ClearIrqFlags();

  if ((_ucRegIrqFlagsCopy & RX_DONE_MASK) == RX_DONE_MASK)
  {
    if(ucHeaderMode == IMPLICIT)
    {
      if ((_ucRegIrqFlagsCopy & PAYLOAD_CRC_ERROR_MASK) == PAYLOAD_CRC_ERROR_MASK)
      {
        return (-1);
      }
      else
      {
        return 1;
      }
    }
    else
    {
      if (GetCrcOnPayload() == CRC_ON)
      {
        if ((_ucRegIrqFlagsCopy & PAYLOAD_CRC_ERROR_MASK) == PAYLOAD_CRC_ERROR_MASK)
        {
          return (-1);
        }
        else
        {
          return 1;
        }
      }
      else
      {
        return (-1);
      }
    }
  }
  else
  {
    return 0;
```

```
  }
}

void TSx1272::ActivateFEMCTX(void)
{
  P2OUT |= BIT7;
}

void TSx1272::DeactivateFEMCTX(void)
{
  P2OUT &= ~BIT7;
}
```

Código A.13 – Cabeçalho da classe TInterfaceUILowLevel

```
#include "HdsMsp430UART.h"
#include "HdsSx1272.h"

#ifndef TINTERFACEUILOWLEVEL_H
#define TINTERFACEUILOWLEVEL_H

class TInterfaceUILowLevel
{
private:
  TMsp430UART _Msp430UART;
  TSx1272 _Sx1272;

  uint16 _uiOpMode;
  uint16 _uiHeaderMode;
  uint16 _uiFrequency;
  uint16 _uiPowerRF;
  uint16 _uiAutoGainControl;
  uint16 _uiSpreadingFactor;
  uint16 _uiCodingRate;
  uint16 _uiBandwidth;
  uint16 _uiLDROptimize;
  uint16 _uiPayloadLength;
  uint16 _uiPayloadCRC;

  uint8 _ucPacket[MAX_PAYLOAD_LENGTH];
  uint16 _uiBytesToCompletePacket;

  uint8 _ucRegIrqFlagsCopy;

  uint32 _ulCorrectPacketsCount;
  uint32 _ulWrongPacketsCount;

  uint8 _ucDigit;
  uint32 _ulCount;

  uint8 _ucReturnCharacter;

public:
  TInterfaceUILowLevel(void);

  uint8 ucStringFromUART[UART_STRING_LENGTH];
  uint8 ucStringToUART[UART_STRING_LENGTH];

  TSx1272* GetSx1272(void);
```

```
  uint16 GetOpMode(void);

  void SetUARTHeaderMode(uint16 uiHeaderMode);
  boolean CheckHeaderMode(void);

  void SetUARTFrequency(uint16 uiFrequency);
  boolean CheckFrequency(void);

  void SetUARTPowerRF(uint16 uiPowerRF);
  boolean CheckPowerRF(void);

  void SetUARTAutoGainControl(uint16 uiAutoGainControl);
  boolean CheckAutoGainControl(void);

  void SetUARTSpreadingFactor(uint16 uiSpreadingFactor);
  boolean CheckSpreadingFactor(void);

  void SetUARTCodingRate(uint16 uiCodingRate);
  boolean CheckCodingRate(void);

  void SetUARTBandwidth(uint16 uiBandwidth);
  boolean CheckBandwidth(void);

  void SetUARTLDROptimize(uint16 uiLDROptimize);
  boolean CheckLDROptimize(void);

  void SetUARTPayloadLength(uint16 uiPayloadLength);
  boolean CheckPayloadLength(void);

  void SetUARTPayloadCRC(uint16 uiPayloadCRC);
  boolean CheckPayloadCRC(void);

  void InitialSetup(void);

  void TxModeRoutine(void);
  void RxModeRoutine(void);

  void RemoteRxStatusCheck(void);
  void RemoteInstruction(uint8* ucInstruction);

  void SendResultToUART(uint32 uiCorrectPacketsCount, uint32 uiWrongPacketsCount);
};
#endif
```

Código A.14 – Implementação da classe TInterfaceUILowLevel

```
#include "MidInterfaceUILowLevel.h"
#include "MidInterfaceProtocolValues.h"

TInterfaceUILowLevel::TInterfaceUILowLevel(void)
{
  InitialSetup();

  _uiBytesToCompletePacket = 0x0080;

  _ucRegIrqFlagsCopy = 0x00;

  _ulCorrectPacketsCount = 0x00000000;
```

```
  _ulWrongPacketsCount = 0x00000000;

  _ucDigit = 0x00;
  _ulCount = 0x00000000;

  _ucReturnCharacter = 0x00;

  _Sx1272.InitLoRaMode();

  _Sx1272.ClearIrqFlags();
}

TSx1272* TInterfaceUILowLevel::GetSx1272(void)
{
  return &_Sx1272;
}

uint16 TInterfaceUILowLevel::GetOpMode(void)
{
  return _uiOpMode;
}

void TInterfaceUILowLevel::SetUARTHeaderMode(uint16 uiHeaderMode)
{
  switch (uiHeaderMode)
  {
  case INTERFACE_EXPLICIT: _Sx1272.SetHeaderMode(EXPLICIT); break;
  case INTERFACE_IMPLICIT: _Sx1272.SetHeaderMode(IMPLICIT); break;
  default: _Sx1272.SetHeaderMode(IMPLICIT); break;
  }

  return;
}

boolean TInterfaceUILowLevel::CheckHeaderMode(void)
{
  switch (_uiHeaderMode)
  {
  case INTERFACE_EXPLICIT: if (_Sx1272.GetHeaderMode() == EXPLICIT) {return true;} break;
  case INTERFACE_IMPLICIT: if (_Sx1272.GetHeaderMode() == IMPLICIT) {return true;} break;
  default: if (_Sx1272.GetHeaderMode() == IMPLICIT) {return true;} break;
  }

  return false;
}

void TInterfaceUILowLevel::SetUARTFrequency(uint16 uiFrequency)
{
  switch (uiFrequency)
  {
  case INTERFACE_CH_10_868: _Sx1272.SetFreqChannel(CH_10_868); break;
  case INTERFACE_CH_11_868: _Sx1272.SetFreqChannel(CH_11_868); break;
  case INTERFACE_CH_12_868: _Sx1272.SetFreqChannel(CH_12_868); break;
  case INTERFACE_CH_13_868: _Sx1272.SetFreqChannel(CH_13_868); break;
  case INTERFACE_CH_14_868: _Sx1272.SetFreqChannel(CH_14_868); break;
  case INTERFACE_CH_15_868: _Sx1272.SetFreqChannel(CH_15_868); break;
  case INTERFACE_CH_16_868: _Sx1272.SetFreqChannel(CH_16_868); break;
  case INTERFACE_CH_17_868: _Sx1272.SetFreqChannel(CH_17_868); break;
  case INTERFACE_CH_00_900: _Sx1272.SetFreqChannel(CH_00_900); break;
  case INTERFACE_CH_01_900: _Sx1272.SetFreqChannel(CH_01_900); break;
```

```
  case INTERFACE_CH_02_900: _Sx1272.SetFreqChannel(CH_02_900); break;
  case INTERFACE_CH_03_900: _Sx1272.SetFreqChannel(CH_03_900); break;
  case INTERFACE_CH_04_900: _Sx1272.SetFreqChannel(CH_04_900); break;
  case INTERFACE_CH_05_900: _Sx1272.SetFreqChannel(CH_05_900); break;
  case INTERFACE_CH_06_900: _Sx1272.SetFreqChannel(CH_06_900); break;
  case INTERFACE_CH_07_900: _Sx1272.SetFreqChannel(CH_07_900); break;
  case INTERFACE_CH_08_900: _Sx1272.SetFreqChannel(CH_08_900); break;
  case INTERFACE_CH_09_900: _Sx1272.SetFreqChannel(CH_09_900); break;
  case INTERFACE_CH_10_900: _Sx1272.SetFreqChannel(CH_10_900); break;
  case INTERFACE_CH_11_900: _Sx1272.SetFreqChannel(CH_11_900); break;
  case INTERFACE_CH_12_900: _Sx1272.SetFreqChannel(CH_12_900); break;
  default: _Sx1272.SetFreqChannel(CH_12_900); break;
  }

  return;
}

boolean TInterfaceUILowLevel::CheckFrequency(void)
{
  switch (_uiFrequency)
  {
  case INTERFACE_CH_10_868: if (_Sx1272.GetFreqChannel() == CH_10_868) {return true;}
        break;
  case INTERFACE_CH_11_868: if (_Sx1272.GetFreqChannel() == CH_11_868) {return true;}
        break;
  case INTERFACE_CH_12_868: if (_Sx1272.GetFreqChannel() == CH_12_868) {return true;}
        break;
  case INTERFACE_CH_13_868: if (_Sx1272.GetFreqChannel() == CH_13_868) {return true;}
        break;
  case INTERFACE_CH_14_868: if (_Sx1272.GetFreqChannel() == CH_14_868) {return true;}
        break;
  case INTERFACE_CH_15_868: if (_Sx1272.GetFreqChannel() == CH_15_868) {return true;}
        break;
  case INTERFACE_CH_16_868: if (_Sx1272.GetFreqChannel() == CH_16_868) {return true;}
        break;
  case INTERFACE_CH_17_868: if (_Sx1272.GetFreqChannel() == CH_17_868) {return true;}
        break;
  case INTERFACE_CH_00_900: if (_Sx1272.GetFreqChannel() == CH_00_900) {return true;}
        break;
  case INTERFACE_CH_01_900: if (_Sx1272.GetFreqChannel() == CH_01_900) {return true;}
        break;
  case INTERFACE_CH_02_900: if (_Sx1272.GetFreqChannel() == CH_02_900) {return true;}
        break;
  case INTERFACE_CH_03_900: if (_Sx1272.GetFreqChannel() == CH_03_900) {return true;}
        break;
  case INTERFACE_CH_04_900: if (_Sx1272.GetFreqChannel() == CH_04_900) {return true;}
        break;
  case INTERFACE_CH_05_900: if (_Sx1272.GetFreqChannel() == CH_05_900) {return true;}
        break;
  case INTERFACE_CH_06_900: if (_Sx1272.GetFreqChannel() == CH_06_900) {return true;}
        break;
  case INTERFACE_CH_07_900: if (_Sx1272.GetFreqChannel() == CH_07_900) {return true;}
        break;
  case INTERFACE_CH_08_900: if (_Sx1272.GetFreqChannel() == CH_08_900) {return true;}
        break;
  case INTERFACE_CH_09_900: if (_Sx1272.GetFreqChannel() == CH_09_900) {return true;}
        break;
  case INTERFACE_CH_10_900: if (_Sx1272.GetFreqChannel() == CH_10_900) {return true;}
        break;
  case INTERFACE_CH_11_900: if (_Sx1272.GetFreqChannel() == CH_11_900) {return true;}
```

```
        break;
  case INTERFACE_CH_12_900: if (_Sx1272.GetFreqChannel() == CH_12_900) {return true;}
        break;
  default: if (_Sx1272.GetFreqChannel() == CH_12_900) {return true;}
        break;
  }

  return false;
}

void TInterfaceUILowLevel::SetUARTPowerRF(uint16 uiPowerRF)
{
  switch (uiPowerRF)
  {
  case INTERFACE_MAX_POWER: _Sx1272.SetPower(MAX_POWER); break;
  case INTERFACE_HIGH_POWER: _Sx1272.SetPower(HIGH_POWER); break;
  case INTERFACE_LOW_POWER: _Sx1272.SetPower(LOW_POWER); break;
  default: _Sx1272.SetPower(MAX_POWER); break;
  }

  return;
}

boolean TInterfaceUILowLevel::CheckPowerRF(void)
{
  switch (_uiPowerRF)
  {
  case INTERFACE_MAX_POWER: if (_Sx1272.CheckPower(MAX_POWER)) {return true;} break;
  case INTERFACE_HIGH_POWER: if (_Sx1272.CheckPower(HIGH_POWER)) {return true;} break;
  case INTERFACE_LOW_POWER: if (_Sx1272.CheckPower(LOW_POWER)) {return true;} break;
  default: if (_Sx1272.CheckPower(MAX_POWER)) {return true;} break;
  }

  return false;
}

void TInterfaceUILowLevel::SetUARTAutoGainControl(uint16 uiAutoGainControl)
{
  switch (uiAutoGainControl)
  {
  case INTERFACE_AGC_OFF: _Sx1272.SetAutoGainControl(AGC_OFF); break;
  case INTERFACE_AGC_ON: _Sx1272.SetAutoGainControl(AGC_ON); break;
  default: _Sx1272.SetAutoGainControl(AGC_OFF); break;
  }

  return;
}

boolean TInterfaceUILowLevel::CheckAutoGainControl(void)
{
  switch (_uiAutoGainControl)
  {
  case INTERFACE_AGC_OFF: if (_Sx1272.GetAutoGainControl() == AGC_OFF) {return true;}
        break;
  case INTERFACE_AGC_ON: if (_Sx1272.GetAutoGainControl() == AGC_ON) {return true;}
        break;
  default: if (_Sx1272.GetAutoGainControl() == AGC_ON) {return true;}
        break;
  }
```

```
  return false;
}

void TInterfaceUILowLevel::SetUARTSpreadingFactor(uint16 uiSpreadingFactor)
{
  switch (uiSpreadingFactor)
  {
  case INTERFACE_SF6: _Sx1272.SetSpreadingFactor(SF6); break;
  case INTERFACE_SF7: _Sx1272.SetSpreadingFactor(SF7); break;
  case INTERFACE_SF8: _Sx1272.SetSpreadingFactor(SF8); break;
  case INTERFACE_SF9: _Sx1272.SetSpreadingFactor(SF9); break;
  case INTERFACE_SF10: _Sx1272.SetSpreadingFactor(SF10); break;
  case INTERFACE_SF11: _Sx1272.SetSpreadingFactor(SF11); break;
  case INTERFACE_SF12: _Sx1272.SetSpreadingFactor(SF12); break;
  default: _Sx1272.SetSpreadingFactor(SF7); break;
  }

  return;
}

boolean TInterfaceUILowLevel::CheckSpreadingFactor(void)
{
  switch (_uiSpreadingFactor)
  {
  case INTERFACE_SF6: if (_Sx1272.GetSpreadingFactor() == SF6) {return true;} break;
  case INTERFACE_SF7: if (_Sx1272.GetSpreadingFactor() == SF7) {return true;} break;
  case INTERFACE_SF8: if (_Sx1272.GetSpreadingFactor() == SF8) {return true;} break;
  case INTERFACE_SF9: if (_Sx1272.GetSpreadingFactor() == SF9) {return true;} break;
  case INTERFACE_SF10: if (_Sx1272.GetSpreadingFactor() == SF10) {return true;} break;
  case INTERFACE_SF11: if (_Sx1272.GetSpreadingFactor() == SF11) {return true;} break;
  case INTERFACE_SF12: if (_Sx1272.GetSpreadingFactor() == SF12) {return true;} break;
  default: if (_Sx1272.GetSpreadingFactor() == SF7) {return true;} break;
  }

  return false;
}

void TInterfaceUILowLevel::SetUARTCodingRate(uint16 uiCodingRate)
{
  switch (uiCodingRate)
  {
  case INTERFACE_CR45: _Sx1272.SetCodingRate(CR45); break;
  case INTERFACE_CR46: _Sx1272.SetCodingRate(CR46); break;
  case INTERFACE_CR47: _Sx1272.SetCodingRate(CR47); break;
  case INTERFACE_CR48: _Sx1272.SetCodingRate(CR48); break;
  default: _Sx1272.SetCodingRate(CR45); break;
  }

  return;
}

boolean TInterfaceUILowLevel::CheckCodingRate(void)
{
  switch (_uiCodingRate)
  {
  case INTERFACE_CR45: if (_Sx1272.GetCodingRate() == CR45) {return true;} break;
  case INTERFACE_CR46: if (_Sx1272.GetCodingRate() == CR46) {return true;} break;
  case INTERFACE_CR47: if (_Sx1272.GetCodingRate() == CR47) {return true;} break;
  case INTERFACE_CR48: if (_Sx1272.GetCodingRate() == CR48) {return true;} break;
  default: if (_Sx1272.GetCodingRate() == CR45) {return true;} break;
```

```
  }

  return false;
}

void TInterfaceUILowLevel::SetUARTBandwidth(uint16 uiBandwidth)
{
  switch (uiBandwidth)
  {
  case INTERFACE_BW125: _Sx1272.SetBandwidth(BW125); break;
  case INTERFACE_BW250: _Sx1272.SetBandwidth(BW250); break;
  case INTERFACE_BW500: _Sx1272.SetBandwidth(BW500); break;
  default: _Sx1272.SetBandwidth(BW500); break;
  }

  return;
}

boolean TInterfaceUILowLevel::CheckBandwidth(void)
{
  switch (_uiBandwidth)
  {
  case INTERFACE_BW125: if (_Sx1272.GetBandwidth() == BW125) {return true;} break;
  case INTERFACE_BW250: if (_Sx1272.GetBandwidth() == BW250) {return true;} break;
  case INTERFACE_BW500: if (_Sx1272.GetBandwidth() == BW500) {return true;} break;
  default: if (_Sx1272.GetBandwidth() == BW500) {return true;} break;
  }

  return false;
}

void TInterfaceUILowLevel::SetUARTLDROptimize(uint16 uiLDROptimize)
{
  switch (uiLDROptimize)
  {
  case INTERFACE_LDR_OFF: _Sx1272.SetLowDataRateOptimize(LDR_OFF); break;
  case INTERFACE_LDR_ON: _Sx1272.SetLowDataRateOptimize(LDR_ON); break;
  default: _Sx1272.SetLowDataRateOptimize(LDR_OFF); break;
  }

  return;
}

boolean TInterfaceUILowLevel::CheckLDROptimize(void)
{
  switch (_uiLDROptimize)
  {
  case INTERFACE_LDR_OFF: if (_Sx1272.GetLowDataRateOptimize() == LDR_OFF) {return true;}
        break;
  case INTERFACE_LDR_ON: if (_Sx1272.GetLowDataRateOptimize() == LDR_ON) {return true;}
        break;
  default: if (_Sx1272.GetLowDataRateOptimize() == LDR_OFF) {return true;}
        break;
  }

  return false;
}

void TInterfaceUILowLevel::SetUARTPayloadLength(uint16 uiPayloadLength)
{
```

```
  _Sx1272.SetPayloadLength((uint8) uiPayloadLength);

  return;
}

boolean TInterfaceUILowLevel::CheckPayloadLength(void)
{
  if (_Sx1272.GetPayloadLength() == _uiPayloadLength)
    return true;
  else
    return false;
}

void TInterfaceUILowLevel::SetUARTPayloadCRC(uint16 uiPayloadCRC)
{
  switch (uiPayloadCRC)
  {
  case INTERFACE_CRC_OFF: _Sx1272.SetRxPayloadCrcOn(CRC_OFF); break;
  case INTERFACE_CRC_ON: _Sx1272.SetRxPayloadCrcOn(CRC_ON); break;
  default: _Sx1272.SetRxPayloadCrcOn(CRC_ON); break;
  }

  return;
}

boolean TInterfaceUILowLevel::CheckPayloadCRC(void)
{
  switch (_uiPayloadCRC)
  {
  case INTERFACE_CRC_OFF: if (_Sx1272.GetRxPayloadCrcOn() == CRC_OFF) {return true;}
        break;
  case INTERFACE_CRC_ON: if (_Sx1272.GetRxPayloadCrcOn() == CRC_ON) {return true;}
        break;
  default: if (_Sx1272.GetRxPayloadCrcOn() == CRC_ON) {return true;}
        break;
  }

  return false;
}

void TInterfaceUILowLevel::InitialSetup(void)
{
  _uiOpMode = INTERFACE_TX;
  _uiHeaderMode = INTERFACE_EXPLICIT;
  _uiFrequency = INTERFACE_CH_12_900;
  _uiPowerRF = INTERFACE_MAX_POWER;
  _uiAutoGainControl = INTERFACE_AGC_OFF;
  _uiSpreadingFactor = INTERFACE_SF12;
  _uiCodingRate = INTERFACE_CR45;
  _uiBandwidth = INTERFACE_BW500;
  _uiLDROptimize = INTERFACE_LDR_ON;
  _uiPayloadLength = 63;
  _uiPayloadCRC = INTERFACE_CRC_ON;

  do
  {
    SetUARTHeaderMode(_uiHeaderMode);

  } while (!CheckHeaderMode());
```

```
  do
  {
    SetUARTPowerRF(_uiPowerRF);
  } while (!CheckPowerRF());

  do
  {
    SetUARTAutoGainControl(_uiAutoGainControl);
  } while (!CheckAutoGainControl());

  do
  {
    SetUARTFrequency(_uiFrequency);
  } while (!CheckFrequency());

  do
  {
    SetUARTSpreadingFactor(_uiSpreadingFactor);
  } while (!CheckSpreadingFactor());

  do
  {
    SetUARTCodingRate(_uiCodingRate);
  } while (!CheckCodingRate());

  do
  {
    SetUARTBandwidth(_uiBandwidth);
  } while (!CheckBandwidth());

  do
  {
    SetUARTLDROptimize(_uiLDROptimize);
  } while (!CheckLDROptimize());

  do
  {
    SetUARTPayloadLength(_uiPayloadLength);
  } while (!CheckPayloadLength());

  do
  {
    SetUARTPayloadCRC(_uiPayloadCRC);
  } while (!CheckPayloadCRC());

  do
  {
    _Sx1272.SetRxTimeout(MAX_RX_TIMEOUT);
  } while (_Sx1272.GetRxTimeout() != MAX_RX_TIMEOUT);

  return;
}

void TInterfaceUILowLevel::TxModeRoutine(void)
{
  boolean bTxSuccess;

  while (_uiOpMode == INTERFACE_TX)
  {
    _Msp430UART.GetString(ucStringFromUART, UART_STRING_LENGTH);
```

```
if (ucStringFromUART[0] == 't')
{
  _ucPacket[_uiPayloadLength - _uiBytesToCompletePacket] =
        (ucStringFromUART[1] - '0')*100 + (ucStringFromUART[2] - '0')*10 +
        (ucStringFromUART[3] - '0');

  _uiBytesToCompletePacket--;

  if (_uiBytesToCompletePacket == 0)
  {
    _Msp430UART.DisableRxInterrupt();

    _uiBytesToCompletePacket = _uiPayloadLength;

    bTxSuccess = _Sx1272.Transmit(_ucPacket, _uiPayloadLength);

    if (bTxSuccess)
      _Msp430UART.PutString((uint8 *)"c\n");
    else
      _Msp430UART.PutString((uint8 *)"w\n");

    _Msp430UART.EnableRxInterrupt();

    if (_ucPacket[0] == 'i' && _ucPacket[1] == 'u' && _uiPayloadLength ==
          REMOTE_STATUS_SIZE_INSTR)
    {
      RemoteRxStatusCheck();
    }
  }
}
else if (ucStringFromUART[0] == 'm')
{
  _uiOpMode = ucStringFromUART[1] - '0';

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'h')
{
  _uiHeaderMode = ucStringFromUART[1] - '0';

  do
  {
    SetUARTHeaderMode(_uiHeaderMode);
  } while (!CheckHeaderMode());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'w')
{
  _uiPowerRF = (ucStringFromUART[1] - '0');

  do
  {
    SetUARTPowerRF(_uiPowerRF);
  } while (!CheckPowerRF());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'a')
```

```
{
  _uiAutoGainControl = ucStringFromUART[1] - '0';

  do
  {
    SetUARTAutoGainControl(_uiAutoGainControl);
  } while (!CheckAutoGainControl());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'f')
{
  _uiFrequency = (ucStringFromUART[1] - '0')*10 + (ucStringFromUART[2] - '0');

  do
  {
    SetUARTFrequency(_uiFrequency);
  } while (!CheckFrequency());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 's')
{
  _uiSpreadingFactor = ucStringFromUART[1] - '0';

  do
  {
    SetUARTSpreadingFactor(_uiSpreadingFactor);
  } while (!CheckSpreadingFactor());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'c')
{
  _uiCodingRate = ucStringFromUART[1] - '0';

  do
  {
    SetUARTCodingRate(_uiCodingRate);
  } while (!CheckCodingRate());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'b')
{
  _uiBandwidth = ucStringFromUART[1] - '0';

  do
  {
    SetUARTBandwidth(_uiBandwidth);
  } while (!CheckBandwidth());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'o')
{
  _uiLDROptimize = ucStringFromUART[1] - '0';

  do
  {
```

```
        SetUARTLDROptimize(_uiLDROptimize);
      } while (!CheckLDROptimize());

      _Msp430UART.PutString((uint8 *)"ready\n");
    }
    else if (ucStringFromUART[0] == 'l')
    {
      _uiPayloadLength = (ucStringFromUART[1] - '0')*1000 +
        (ucStringFromUART[2] - '0')*100 + (ucStringFromUART[3] - '0')*10 +
        (ucStringFromUART[4] - '0');
      _uiBytesToCompletePacket = _uiPayloadLength;

      do
      {
        SetUARTPayloadLength(_uiPayloadLength);
      } while (!CheckPayloadLength());

      _Msp430UART.PutString((uint8 *)"ready\n");
    }
    else if (ucStringFromUART[0] == 'p')
    {
      _uiPayloadCRC = ucStringFromUART[1] - '0';

      do
      {
        SetUARTPayloadCRC(_uiPayloadCRC);
      } while (!CheckPayloadCRC());

      _Msp430UART.PutString((uint8 *)"ready\n");
    }
  }

  return;
}

 void TInterfaceUILowLevel::RxModeRoutine(void)
{
  int8 iRxResult = 0x00;
  uint8 temp = 0x00;

  _ulCorrectPacketsCount = 0;
  _ulWrongPacketsCount = 0;

  while (_uiOpMode == INTERFACE_RX)
  {
    _Msp430UART.GetString(ucStringFromUART, UART_STRING_LENGTH);

    if (ucStringFromUART[0] == 'r')
    {
      while (!_Msp430UART.GetRxFlag())
      {
        iRxResult = _Sx1272.ReceiveSingle(_uiHeaderMode);

        if (iRxResult != 0)
        {
          temp = _Sx1272.GetRxNbBytes();

          _Sx1272.ReadFromFifo(_ucPacket, temp);

          if (iRxResult == (-1))
```

```
            {
              _ulWrongPacketsCount++;

              _ucPacket[temp] = 'w';
              _ucPacket[temp + 1] = '\n';
              _Msp430UART.PutString(_ucPacket);
            }
            else
            {
              _ulCorrectPacketsCount++;

              if (_ucPacket[0] == 'i') // Instruction from Tx
              {
                _ulCorrectPacketsCount--;
                RemoteInstruction(_ucPacket);
              }

              _ucPacket[temp] = 'c';
              _ucPacket[temp + 1] = '\n';
              _Msp430UART.PutString(_ucPacket);
            }
          }
        }
      }
      else if (ucStringFromUART[0] == 'e')
      {
        _Msp430UART.PutString((uint8 *)"RxFinished\n");

        SendResultToUART(_ulCorrectPacketsCount, _ulWrongPacketsCount);

        _ulCorrectPacketsCount = 0;
        _ulWrongPacketsCount = 0;
      }
      else if (ucStringFromUART[0] == 'm')
      {
        _uiOpMode = ucStringFromUART[1] - '0';

        _Msp430UART.PutString((uint8 *)"ready\n");
      }
      else if (ucStringFromUART[0] == 'h')
      {
        _uiHeaderMode = ucStringFromUART[1] - '0';

        do
        {
          SetUARTHeaderMode(_uiHeaderMode);
        } while (!CheckHeaderMode());

        _Msp430UART.PutString((uint8 *)"ready\n");
      }
      else if (ucStringFromUART[0] == 'w')
      {
        _uiPowerRF = (ucStringFromUART[1] - '0');

        do
        {
          SetUARTPowerRF(_uiPowerRF);
        } while (!CheckPowerRF());

        _Msp430UART.PutString((uint8 *)"ready\n");
```

```
}
else if (ucStringFromUART[0] == 'a')
{
  _uiAutoGainControl = ucStringFromUART[1] - '0';

  do
  {
    SetUARTAutoGainControl(_uiAutoGainControl);
  } while (!CheckAutoGainControl());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'f')
{
  _uiFrequency = (ucStringFromUART[1] - '0')*10 + (ucStringFromUART[2] - '0');

  do
  {
    SetUARTFrequency(_uiFrequency);
  } while (!CheckFrequency());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 's')
{
  _uiSpreadingFactor = ucStringFromUART[1] - '0';

  do
  {
    SetUARTSpreadingFactor(_uiSpreadingFactor);
  } while (!CheckSpreadingFactor());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'c')
{
  _uiCodingRate = ucStringFromUART[1] - '0';

  do
  {
    SetUARTCodingRate(_uiCodingRate);
  } while (!CheckCodingRate());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'b')
{
  _uiBandwidth = ucStringFromUART[1] - '0';

  do
  {
    SetUARTBandwidth(_uiBandwidth);
  } while (!CheckBandwidth());

  _Msp430UART.PutString((uint8 *)"ready\n");
}
else if (ucStringFromUART[0] == 'o')
{
  _uiLDROptimize = ucStringFromUART[1] - '0';
```

```
      do
      {
        SetUARTLDROptimize(_uiLDROptimize);
      } while (!CheckLDROptimize());

      _Msp430UART.PutString((uint8 *)"ready\n");
    }
    else if (ucStringFromUART[0] == 'l')
    {
      _uiPayloadLength = (ucStringFromUART[1] - '0')*1000 +
        (ucStringFromUART[2] - '0')*100 + (ucStringFromUART[3] - '0')*10 +
        (ucStringFromUART[4] - '0');
      _uiBytesToCompletePacket = _uiPayloadLength;

      do
      {
        SetUARTPayloadLength(_uiPayloadLength);
      } while (!CheckPayloadLength());

      _Msp430UART.PutString((uint8 *)"ready\n");
    }
    else if (ucStringFromUART[0] == 'p')
    {
      _uiPayloadCRC = ucStringFromUART[1] - '0';

      do
      {
        SetUARTPayloadCRC(_uiPayloadCRC);
      } while (!CheckPayloadCRC());

      _Msp430UART.PutString((uint8 *)"ready\n");
    }
  }

  return;
}

void TInterfaceUILowLevel::RemoteInstruction(uint8 *ucInstruction)
{
  if (ucInstruction[1] == 'w' || ucInstruction[1] == 'a' || ucInstruction[1] == 'f' ||
      ucInstruction[1] == 's' || ucInstruction[1] == 'c' || ucInstruction[1] == 'b' ||
      ucInstruction[1] == 'o')
  {
    for (uint8 i = 1; i < (REMOTE_CONFIG_QNT_INSTR*REMOTE_CONFIG_SIZE_INSTR + 1); i = i +
        REMOTE_CONFIG_SIZE_INSTR)
    {
      if (ucInstruction[i] == 'w')
      {
        _uiPowerRF = (ucInstruction[i + (REMOTE_CONFIG_SIZE_INSTR - 1)] - '0');

        do
        {
          SetUARTPowerRF(_uiPowerRF);
        } while (!CheckPowerRF());
      }
      else if (ucInstruction[i] == 'a')
      {
        _uiAutoGainControl = (ucInstruction[i + (REMOTE_CONFIG_SIZE_INSTR - 1)] - '0');

        do
```

```
        {
          SetUARTAutoGainControl(_uiAutoGainControl);
        } while (!CheckAutoGainControl());
      }
      else if (ucInstruction[i] == 'f')
      {
        _uiFrequency = (ucInstruction[i + (REMOTE_CONFIG_SIZE_INSTR - 2)] - '0')*10 +
        (ucInstruction[i + (REMOTE_CONFIG_SIZE_INSTR - 1)] - '0');

        do
        {
          SetUARTFrequency(_uiFrequency);
        } while (!CheckFrequency());
      }
      else if (ucInstruction[i] == 's')
      {
        _uiSpreadingFactor = (ucInstruction[i + (REMOTE_CONFIG_SIZE_INSTR - 1)] - '0');

        do
        {
          SetUARTSpreadingFactor(_uiSpreadingFactor);
        } while (!CheckSpreadingFactor());
      }
      else if (ucInstruction[i] == 'c')
      {
        _uiCodingRate = (ucInstruction[i + (REMOTE_CONFIG_SIZE_INSTR - 1)] - '0');

        do
        {
          SetUARTCodingRate(_uiCodingRate);
        } while (!CheckCodingRate());
      }
      else if (ucInstruction[i] == 'b')
      {
        _uiBandwidth = (ucInstruction[i + (REMOTE_CONFIG_SIZE_INSTR - 1)] - '0');

        do
        {
          SetUARTBandwidth(_uiBandwidth);
        } while (!CheckBandwidth());
      }
      else if (ucInstruction[i] == 'o')
      {
        _uiLDROptimize = (ucInstruction[i + (REMOTE_CONFIG_SIZE_INSTR - 1)] - '0');

        do
        {
          SetUARTLDROptimize(_uiLDROptimize);
        } while (!CheckLDROptimize());
      }
    }
  }
  else if (ucInstruction[1] == 'r') // Return packet of size N
  {
    _ucReturnCharacter = ucInstruction[2];
    _uiPayloadLength = ucInstruction[3];

    do
    {
      SetUARTPayloadLength(_uiPayloadLength);
```

```
    } while (!CheckPayloadLength());

    for (uint8 i = 0; i < _uiPayloadLength; i++)
    {
      _ucPacket[i] = _ucReturnCharacter;
    }

    _Msp430UART.DisableRxInterrupt();

    _Sx1272.Transmit(_ucPacket, _uiPayloadLength);

    _Msp430UART.EnableRxInterrupt();
  }
  else if (ucInstruction[1] == 'u') // Return status
  {
    _uiPayloadLength = REMOTE_STATUS_STRING_LENGTH;

    do
    {
      SetUARTPayloadLength(_uiPayloadLength);
    } while (!CheckPayloadLength());

    _ucPacket[0] = 'c';
    _ulCount = _ulCorrectPacketsCount;

    for (uint8 i = 10; i > 0; i--)
    {
      _ucDigit = _ulCount % 10;
      _ulCount = _ulCount / 10;

      _ucPacket[i] = _ucDigit + '0';
    }

    _Msp430UART.DisableRxInterrupt();

    _Sx1272.Transmit(_ucPacket, _uiPayloadLength);

    _Msp430UART.EnableRxInterrupt();

    _ucPacket[0] = 'w';
    _ulCount = _ulWrongPacketsCount;

    for (uint8 i = _uiPayloadLength - 1; i > 0; i--)
    {
      _ucDigit = _ulCount % 10;
      _ulCount = _ulCount / 10;

      _ucPacket[i] = _ucDigit + '0';
    }

    _Msp430UART.DisableRxInterrupt();

    _Sx1272.Transmit(_ucPacket, _uiPayloadLength);

    _Msp430UART.EnableRxInterrupt();
  }
}

void TInterfaceUILowLevel::SendResultToUART(uint32 ulCorrectPacketsCount,
        uint32 ulWrongPacketsCount)
```

```
{
  uint8 i;
  uint8 j;

  for (i = 0; i < 2; i++)
  {
    if (i == 0)
    {
      _ulCount = ulCorrectPacketsCount;
      ucStringToUART[0] = 'c';
    }
    else
    {
      _ulCount = ulWrongPacketsCount;
      ucStringToUART[0] = 'w';
    }

    for (j = UART_STRING_LENGTH - 2; j > 0; j--)
    {
      _ucDigit = _ulCount % 10;
      _ulCount = _ulCount / 10;

      ucStringToUART[j] = _ucDigit + '0';
    }

    ucStringToUART[UART_STRING_LENGTH - 1] = '\n';

    _Msp430UART.PutString(ucStringToUART);
  }

  return;
}

void TInterfaceUILowLevel::RemoteRxStatusCheck(void)
{
  int8 iRxResult = 0x00;
  uint8 temp = 0x00;

  // First reception (rights or wrongs)
  iRxResult = _Sx1272.ReceiveSingle(_uiHeaderMode);

  if (iRxResult == 1) // Sucessfull
  {
    temp = _Sx1272.GetRxNbBytes();

    _Sx1272.ReadFromFifo(_ucPacket, temp);
    _ucPacket[temp] = '\n';

    _Msp430UART.PutString(_ucPacket);
  }
  else
  {
    _Msp430UART.PutString((uint8 *)"failed \n");
  }

  // Second reception (rights or wrongs)
  iRxResult = _Sx1272.ReceiveSingle(_uiHeaderMode);

  if (iRxResult == 1) // Sucessfull
  {
```

```
    temp = _Sx1272.GetRxNbBytes();

    _Sx1272.ReadFromFifo(_ucPacket, temp);
    _ucPacket[temp] = '\n';

    _Msp430UART.PutString(_ucPacket);
  }
  else
  {
    _Msp430UART.PutString((uint8 *)"failed \n");
  }
}
```

Código A.15 – Função principal

```
#include "MidInterfaceUILowLevel.h"
#include "msp430.h"

void ClockSetup(void);

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;        // Stop WDT
  PMMCTL0 = PMMPW | PMMCOREV_3;  // PMM Core Voltage 3 (1.85V)

  P2DIR |= BIT7;

  ClockSetup();

  __enable_interrupt();

  TInterfaceUILowLevel InterfaceUILowLevel;

  while (1)
  {
    if (InterfaceUILowLevel.GetOpMode() == 0)
      InterfaceUILowLevel.TxModeRoutine();
    else
      InterfaceUILowLevel.RxModeRoutine();
  }
}


void ClockSetup(void)
{
  UCSCTL6 = (XCAP_3 +          //!< Internal load cap set to 12 pF
             XT1DRIVE_3 +      //!< Maximum current consumption
             XT2OFF);          //!< Turn off the XT2]

  UCSCTL3 = 0;

  do
  {
    UCSCTL7 &= ~((uint16) XT1LFOFFG);          //!< Clear XT1 fault flags

  } while (UCSCTL7 & XT1LFOFFG);

  //! The microprocessor will run at 32kHz, all the clocks will be set to 32768
  UCSCTL4 = (SELA__XT1CLK +   //!< Set ACLK, SMCLK and MCLK to 32.768 Hz
```

```
        SELS__XT1CLK + SELM__XT1CLK);

    __bis_SR_register(SCG0);

    UCSCTL0 = 0x0000;           //!< Set lowest possible DCOx, MODx ?????
    UCSCTL1 = DCORSEL_6;           //!< Set RSELx for DCO (See datasheet of device)
    UCSCTL2 = FLLD_1 + 381;        //!< N = 381
                                    //!< FLLD = / 2;
                                    //!< D*(N + 1) * FLLRef = Fdco
                                    //!< 2*(381 + 1) * 32768 = 25.034.752 MHz

    __bic_SR_register(SCG0);

    __delay_cycles(3200);

    //! Configures the alternate, secondary and master clock sources
    UCSCTL4 = (SELA__XT1CLK +       //!< alteranate clock is internal REF clock (32.768kHz)
               SELS__DCOCLK +       //!< secondary clock is DCO (25MHZ or 1Mhz)
               SELM__DCOCLK);       //!< master clock is DC0 (25MHz or 1Mhz)

    //! Loop until DCO fault flag is cleared


    //! Divides the secondary clock by 2
    UCSCTL5 = DIVS__2;    //!<  SMCLK = 12,5 Mhz or 500 khz.
}
```