Centro Federal de Educação Tecnológica de Santa Catarina

CEFET-SC

Curso de Pós-Graduação em Desenvolvimento de Produtos Eletrônicos

DATA LOGGER PARA GPS

por

Ernesto Rolim Theodorico da Silva

Florianópolis, julho de 2009

Centro Federal de Educação Tecnológica de Santa Catarina

CEFET-SC

Curso de Pós-Graduação em Desenvolvimento de Produtos Eletrônicos

DATA LOGGER PARA GPS

Monografia apresentado ao curso de Pós-Graduação em Desenvolvimento de Produtos Eletrônico Digitais como requisito parcial a obtenção do titulo de especialista em desenvolvimento de produtos eletrônicos.

Orientador:
Prof. Dr. Golberi de Salvador Ferreira

Florianópolis, julho de 2009

Ernesto Rolim Theodorico da Silva

Monografia sob o título Data Logger para GPS, defendido por Ernesto Rolim Theodorico da Silva em 10 de julho de 2009 e aprovado pela banca examinadora constituída conforme abaixo:

_____

Prof. Dr. Golberi de Salvador Ferreira

_____

Prof. Dr. Muriel Bittencourt de Liz

_____

Prof. Msc. Everton Luis Ferret dos Santos

Florianópolis, julho de 2009

AGRADECIMENTOS

Primeiramente a Deus, por ter me dado saúde para que eu pudesse chegar a esse momento. A Fabiana, minha esposa, que sempre esteve ao meu lado, me ajudando e apoiando, acreditando que no fim tudo daria certo, mesmo quando nada funcionava.

RESUMO

O presente trabalho traz um sistema de coleta automática e armazenamento cronológico de dados (*Data Logger*), para um dispositivo GPS (*Global Positioning System*). O módulo GPS, fabricado pela UniTraQ, enviará as sentenças através do protocolo NMEA-0183 V3.01 a uma placa contendo um FPGA da Xilinx®, este se encarregará de armazenar as sentenças em uma memória não volátil.

Ao ser conectado a um PC (*personal computer*), através da porta serial, o Data Logger irá descarregar as informações armazenadas. Estas informações serão tratadas por um software no PC que irá gerar um mapa do percurso através da utilização das APIs do Google Maps.

Palavras-chave: FPGA, GPS, Data Logger.

ABSTRACT

This paper presents an automatic system to collect and store chronological (Data Logger) for a GPS (Global Positioning System) device. The GPS, device build by UniTraQ, will send queries using the NMEA-0183 V3.01 to a Xilinx FPGA board, which will be responsible to store the queries on a non-volatile memory.

Once connected to a PC through the serial port, the Data Logger will download the stored data. After that, a software running on the PC will process the information drawing a map, which will be presented in Google Maps though its APIs.

Keywords: FPGA, GPS, Data Logger.

SUMÁRIO

LISTA DAS FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
ASCII	American Standard Code for Information Interchange
CPLD	Complex Programmable Logic Device
DGPS	Differencial GPS
EEPROM	Erasable Programmable Read Only Memory
FPGA	Field Programmable Gate Array
GPS	Global Position System
IEEE	Institute of Electrical and Electronic Engineers
JTAG	Joint Test Action Group
NMEA	National Marine Electronics Association
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
RAM	Random Access Memory
ROM	Read Only Memory
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TDI	Test Data In
TDO	Test Data Out
UART	Universal Asynchronous Receiver Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1 INTRODUÇÃO

O desejo do homem em saber a sua localização já vem de tempos muito remotos, desde a antiguidade o homem tem aprimorado suas técnicas de localização, passando pela observação de corpos celestes, sinais de rádios etc. Atualmente, o homem conseguiu realizar este sonho com uma precisão espantosa de poucos centímetros, o responsável por esse feito é o GPS (*Global Positioning System* – Sistema de Posicionamento Global), desenvolvido pelo governo dos Estados Unidos e que se apresenta em duas versões, a de uso civil e a restrita as forças armadas dos Estados Unidos.

Saber a sua exata localização é uma informação muito importante, entretanto, quando se deseja um histórico cronológico destas posições, faz-se necessário o uso de algum dispositivo que a armazene essas informações durante um determinado período de tempo. Um dispositivo capaz de realizar esta tarefa chama-se Data Logger, um equipamento que possui uma memória não volátil onde são armazenadas as informações de localização de forma cronológica, ou seja, sua posição e a hora referente a algum sistema padrão.

Desenvolver um Data Logger para GPS exige por parte do projetista, a tomada de decisão de como será realizado o gerenciamento de todo o sistema. O projetista terá que escolher se usará, por exemplo, um microcontrolador ou alguma lógica programável; além de escolher a tecnologia, ainda terá que decidir entre uma infinidade de componentes disponíveis no mercado para a tecnologia escolhida, para tanto, será necessário um profundo conhecimento do assunto em questão. Nesse trabalho será usado um *kit* de desenvolvimento para FPGA. Por se tratar de um *kit*, entre suas diversas características, somente o FPGA, a memória *Flash* SPI e as portas seriais serão usadas.

Dessa forma espera-se como resultado final, um dispositivo capaz de armazenar as informações de posição na memória *Flash* SPI, para posterior análise em um computador. Com essas informações será possível o traçado em um mapa da trajetória feita pelo Sistema Data Logger bem como a geração de uma tabela com dados de posição, velocidade, tempo etc.

1.1 Justificativa

Muitos módulos receptores de sinal GPS disponíveis no mercado dispõem de uma saída de dados no padrão RS-232 e outra no padrão TTL, porém, por serem módulos de baixo custo, não possuem nem um tipo de dispositivo de armazenamento das informações e de sua visualização pelo usuário.

Permitir que estes dados possam ser armazenados e posteriormente visualizados, só é alcançado com o desenvolvimento de algum dispositivo de hardware/software que faça a interface entre o módulo e o usuário.

1.2 Objetivos gerais

Este trabalho tem por objetivo a elaboração de um sistema capaz de armazenar dados enviados por um módulo GPS através de uma conexão padrão RS-232, na forma de sentenças NMEA 0183.

Para alcançar este objetivo, será usado um kit de desenvolvimento fabricado pela empresa Digilent Inc.. O componente principal desse *kit* é o FPGA da família Spartan-3E XC3S500E. Nele será implementado todo o sistema digital que fará a comunicação RS-232 com o módulo GPS, todo o controle de gravação na memória não volátil, bem como a recuperação destes dados para envio a um PC para posterior análise.

1.3 Objetivos específicos

O primeiro objetivo específico desse trabalho é a construção em VHDL dos blocos que realizarão a tarefa de receber as informações fornecidas pelo módulo GPS, gerar todos os sinais necessários para gravação na memória *Flash* SPI, bem como gerar os sinais para recuperar as informações gravadas e transmiti-las a um PC. O segundo objetivo específico é o desenvolvimento de uma aplicação baseada em PC para visualização dos dados armazenados pelo Sistema Data Logger, e geração de um mapa com o percurso realizado pelo Sistema Data Logger.

1.4 Áreas de aplicação

Nos dias de hoje, o GPS está cada vez mais presente em nossas vidas. Muitos telefones celulares mais sofisticados, conhecidos como *smartphones*, já vêm com um módulo GPS embutido com mapas de cidades e POI (*Points Of Interest* – pontos de interesse), entre outras tecnologias. Navegadores para automóveis estão cada vez mais populares e com preços cada vez mais acessíveis. Esses são alguns exemplos de uso não comercial. No campo profissional, as aplicações são bastante vastas, deste a utilizada em navegação aérea e marítima, passando pelo rastreamento de veículos e cargas, entre outros.

As informações disponibilizadas pelos módulos GPS, de um modo geral, são atualizadas a cada segundo. A importância de se armazenar estas informações para uma posterior analise é fundamental em muitos casos. Este dispositivo de armazenamento de informações é conhecido por "Data Logger", que é o objetivo desse trabalho.

1.4.1 Restrições e especificações

Como todo sistema eletrônico, o Sistema Data Logger para GPS possui suas restrições. As duas principais são com relação à autonomia de funcionamento e a capacidade de armazenamento. As duas estão ligadas exclusivamente aos componentes utilizados. No caso da capacidade de armazenamento, espera-se que este projeto tenha uma capacidade de armazenamento de informações por um tempo aproximado de 10 horas. Isso se deve ao fato da memória que vem instalada no *kit* ser de 16 Mbit. Por se tratar de um sistema móvel, com relação à autonomia de funcionamento, o fator restritivo é a bateria que alimenta o sistema. Uma bateria com maior capacidade aumentará sua autonomia, porém como consequência negativa, seu peso aumentara e sua portabilidade será prejudicada.

Como todo produto eletrônico, espera-se que tenha um custo que seja viável a sua industrialização. Para um sistema mínimo utilizando a tecnologia de lógica programável como unidade de controle, os componentes para se construir um Sistema Data Logger com funções básicas de armazenamento e transferência de dados para um PC é bastante pequena. Basicamente limita-se a um FPGA ou CPLD, uma memória não volátil, um circuito integrado RS-232, alguns componentes discretos, como capacitores de desacoplamento, resistores, conectores etc., além da placa de circuito impresso e uma caixa plástica de proteção. Um preço estimado utilizando um CPLD como unidade de controle e considerando somente os componentes mais importantes, o valor do Sistema Data Logger fica em torno de $ 82,00

(dólares americanos). Não foi considerado o valor do transporte nem impostos. A placa de circuito impresso não foi levada em conta, pois depende da capacidade do projetista em desenhá-la para se ter o tamanho final, bem como o número de camadas utilizadas.

2 O SISTEMA GPS

O Sistema de Posicionamento Global (GPS) é um sistema de navegação baseado em satélite que foi desenvolvido pelo Departamento de Defesa dos Estados Unidos da América (DoD) no início dos anos 70. Inicialmente o sistema foi desenvolvido para preencher as necessidades militares. Mais tarde foi disponibilizado ao uso civil e hoje é um sistema dual, atendendo tanto o uso civil quanto ao militar [6].

O sistema fornece informação contínua de tempo e posição em qualquer local do mundo e sob quaisquer condições climáticas.

O sistema GPS pode ser usado por qualquer pessoa sem a necessidade de pagamento de qualquer tipo de tarifa ou taxa. Sendo assim, por questões de segurança, o receptor GPS é um sistema passivo, ou seja, pode somente receber os sinais dos satélites.

2.1 Visão geral

O sistema GPS consiste de uma constelação de 24 satélites conhecida como capacidade operacional inicial (IOC) [5]. Para alcançar a continua cobertura mundial, os satélites GPS estão dispostos de modo que quatro satélites são colocados em cada um dos seis planos orbitais (figura 1).

Figura 1- Constelação de satélites GPS

Fonte: Garmin

Com esta constelação geométrica, de 4 a 10 satélites podem ser vistos simultaneamente em qualquer lugar do globo terrestre.

O sistema GPS consiste de três segmentos: segmento espacial, segmento de controle e segmento de usuário (figura 2) [5].

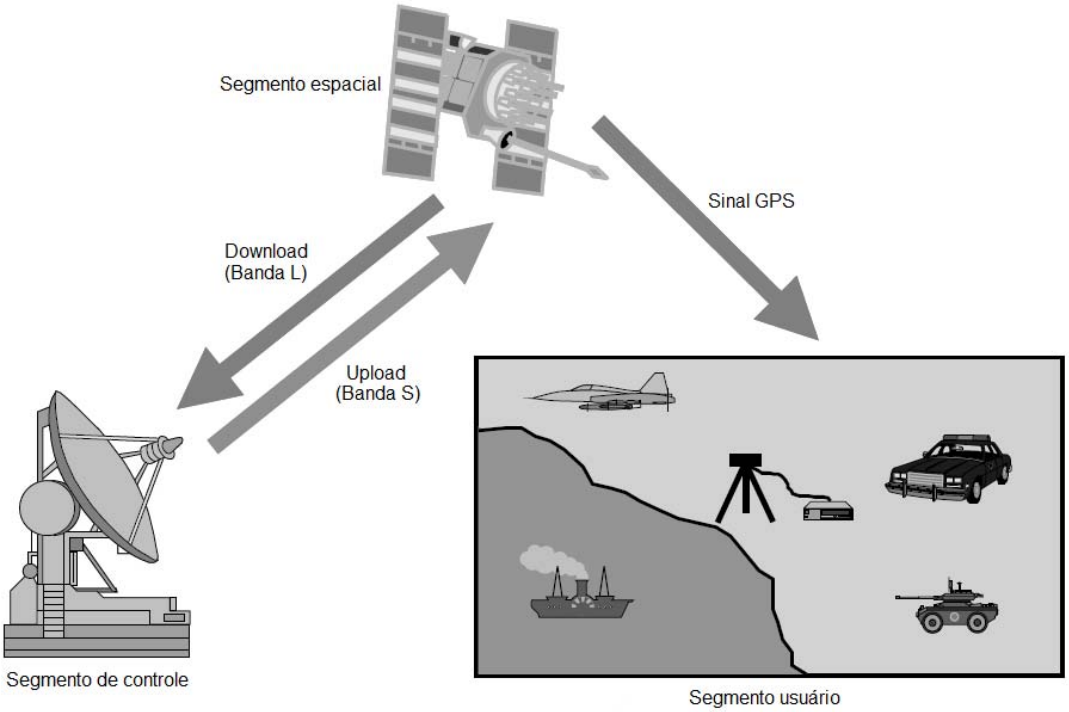


Figura 2 – Segmentos GPS

Fonte: Introduction to GPS – The Global Positioning System (2002, p. 3)

O segmento espacial é formado pela constelação dos 24 satélites mencionado anteriormente. Cada satélite envia um sinal composto por diversos componentes: duas ondas senoidais, conhecidas como frequências portadoras, dois códigos digitais e uma mensagem de navegação. Os códigos e a mensagem de navegação são adicionados a portadora. A portadora e os códigos são usados principalmente para determinar a distância entre o receptor GPS e os satélites. As mensagens de navegação contem entre outras informações as coordenadas (localização) dos satélites como função do tempo.

O sistema de controle consiste de uma rede mundial de estações de rastreamento. A tarefa primária do segmento de controle é rastrear os satélites em sequência para determinar e predizer a localização dos mesmos, verificar sua integridade, o comportamento dos relógios

atômicos, dados atmosféricos, entre outras considerações. Estas informações são empacotadas e transmitidas ao satélite através da banda S.

O segmento usuário inclui todas as aplicações civis e militares.

2.2 Funcionamento

A ideia por trás do GPS é muito simples. Se a distância entre um ponto na terra (receptor GPS) e três satélites for conhecida, juntamente com a localização dos satélites, a localização do receptor pode ser determinada pela técnica conhecida por trilateração [10].

Na trilateração, a posição de um ponto desconhecido é determinada pela medida dos lados de um triângulo entre um ponto desconhecido e dois ou mais pontos conhecidos, ou seja, os satélites. Contrariamente a triangulação que usa a informação dos ângulos para determinar a localização.

A melhor forma de se entender como funciona a trilateração, é através de um exemplo. Inicialmente será explicado a trilateração em duas dimensões, conhecida como trilateração 2D.

Imagine que uma pessoa esteja perdida e que a única informação que ela tem é que está a 100 km de Ituporanga – SC. Com essa informação nas mãos, ela só sabe que pode estar em qualquer ponto sobre uma circunferência de 100 km de raio, tendo como centro dessa circunferência a cidade de Ituporanga.

Mas apenas com essa informação ela não está apta, a saber, a sua exata localização, porem se lhe for informado que ela está a 79 km de Barra Velha – SC e essa pessoa cruzar essa informação com a anterior poderá verificar que a circunferência em torno dessas duas cidades se cruza em dois pontos. Mesmo restringindo a duas possíveis posições aonde ela possa estar, sua localização ainda não está completamente definida.

Uma terceira informação é necessária para que a completa localização dessa pessoa seja possível. Se a mesma souber a distância de uma terceira cidade, sua localização estará completamente definida, como pode ser observado na figura 3.

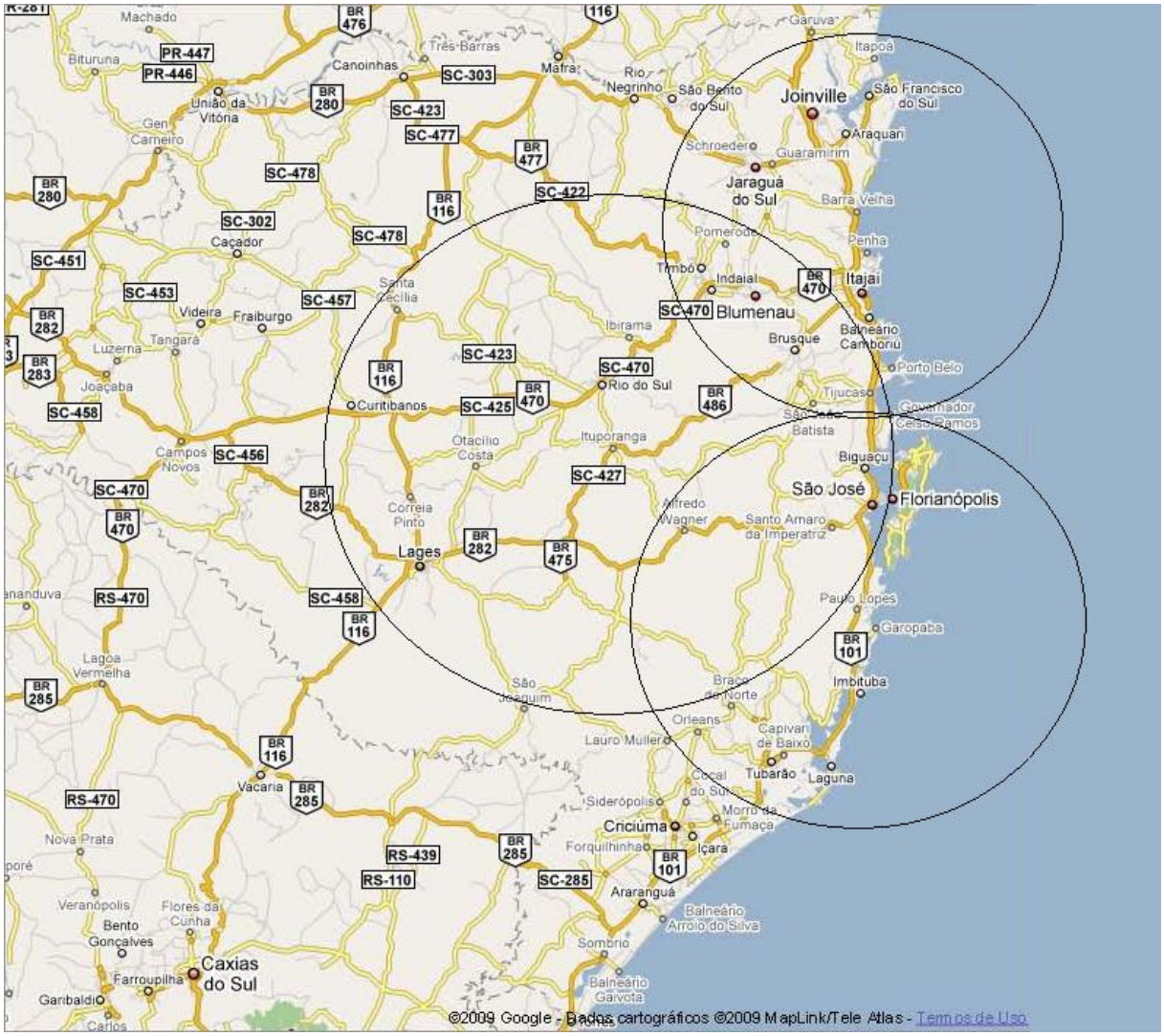


Figura 3 – Trilateração 2D

Fonte: Própria

A informação de que ela está a 73 km de Paulo Lopes – SC gera a terceira circunferência, e agora somente um único ponto é a intersecção das três circunferências, no caso a cidade de Governador Celso Ramos – SC, determinando a sua localização.

Esse conceito funciona também para espaços tridimensionais, nesse caso usam-se esferas ao invés de circunferências.

A visualização da trilateração tridimensional (3D) é bem mais complexa do que a bidimensional e seu funcionamento é detalhado a seguir.

Se soubermos que a distância entre o receptor GPS e o satélite A é de 10 km, o receptor poderá estar em qualquer lugar em uma esfera imaginária de raio 10 km (figura 4).

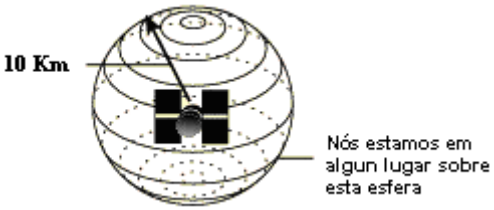


Figura 4 – Trilateração 3D

Fonte: António Pestana (2002, p. 3)

Se soubermos que o receptor está a 17 km do satélite B podermos sobrepor as duas esferas obtendo um circulo perfeito (figura 5).

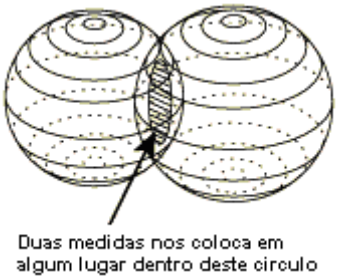


Figura 5 – Trilateração 3D com 2 satélites

Fonte: António Pestana (2002, p. 3)

Com a informação de distância de um terceiro satélite, por exemplo, 22 km, teremos uma terceira esfera que se cruzara com as outras duas em dois pontos. A própria terra pode agir com uma quarta esfera, tendo um dos pontos de cruzamento a própria superfície terrestre e o outro ponto o espaço. Ignorando o ponto que se encontra no espaço, temos a localização exata desse ponto, como pode ser observado na figura 6.

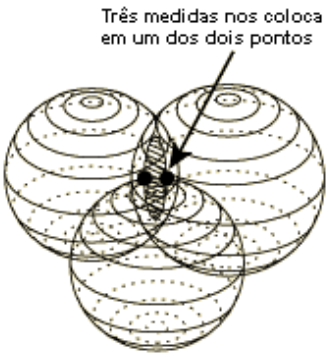


Figura 6 – Trilateração 3D com 3 satélites

Fonte: António Pestana (2002, p. 3)

A medição de distância de um satélite até o receptor é bastante simples. Cada satélite emite um sinal binário que lhe é único. O sinal é altamente complexo e de características aleatórias, como pode ser visto na figura 7.

Figura 7 – Sinal GPS

Fonte: António Pestana (2002, p. 4)

Na realidade trata-se de sequências perfeitamente pré-determinadas, definidas por funções do tipo y=f(t), é por esse motivo que esses sinais são conhecidos como pseudo-aleartórios. As funções y=f(t) são divulgadas publicamente pelo DoD (departamento de defesa dos Estados Unidos da América).

Internamente o receptor gera funções idênticas ao transmissor, comparando-a com a que está recebendo num determinado instante (figura 8).

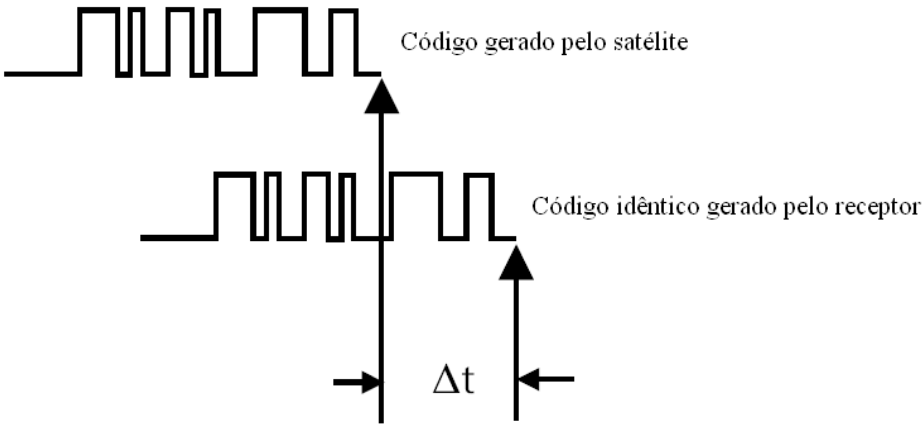


Figura 8 – Medida dos códigos Pseudo-aleartório

Fonte: Introduction to GPS – The Global Positioning System (2002, p. 20)

Caso o relógio do receptor GPS for tão preciso quanto o do satélite, a distância entre o receptor e o satélite pode ser determinada pela equação a seguir:

$$Dist = c \times \Delta t$$

Onde:

Dist = Distância entre o satélite e o receptor;

c = Velocidade da luz;

Δt = Diferença de tempo entre o código gerado pelo satélite e o gerado pelo receptor.

Porem o relógio interno do receptor não é tão preciso quanto o do satélite. Na realidade, o Δt vem acrescido de uma parcela que reflete o erro cometido pelo relógio do receptor (*clock bias – CB*), assim a equação da distância torna-se:

$$Dist = c \times (\Delta t + CB)$$

O *CB* depende apenas do relógio do receptor, afetando de forma igual à medição a qualquer satélite.

A figura 9 mostra a medida real entre dois satélites e o receptor.

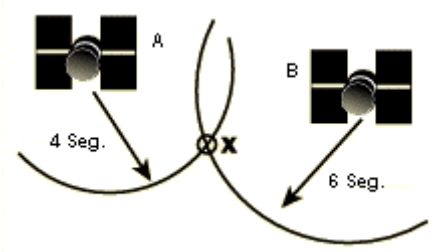


Figura 9 – Distância real entre dois satélites

Fonte: António Pestana (2002, p. 6)

Na figura 10, é possível verificar o efeito de CB na medição das distâncias.

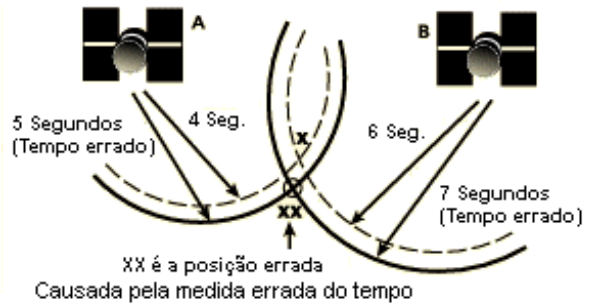


Figura 10 – Efeito *Clock Bias*

Fonte: António Pestana (2002, p. 6)

Como é possível verificar, com relação ao satélite A, a diferença de tempo entre a transmissão e a recepção foi de 5 segundos, quando na verdade o tempo correto seria de 4 segundos, o mesmo ocorre com o satélite B, o tempo correto é de 6 segundos, ao invés de 7 segundos. Por causa dos valores errados, a intersecção destes pontos é o ponto XX, quando na realidade deveria ser o ponto X.

Para resolver o problema da imprecisão, adiciona-se um terceiro satélite ao conjunto, como mostrado na figura 11.

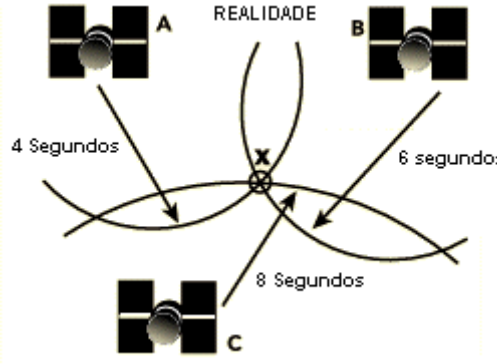


Figura 11 – Medição com três satélites

Fonte: António Pestana (2002, p. 6)

Através de um processo interativo, o receptor introduz diversos valores de CB na equação da distância, até encontrar um no qual as três circunferências se cruzam num mesmo ponto, como pode ser observado na figura 12.

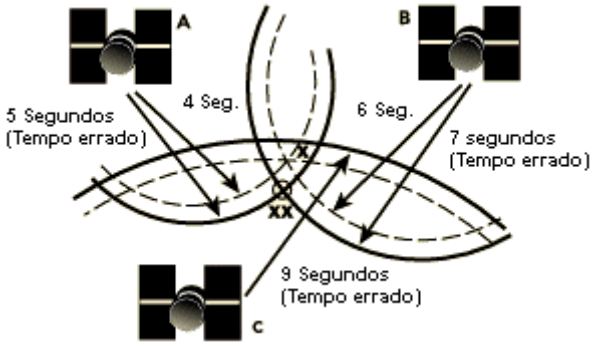


Figura 12 – Medição corrigida

Fonte: António Pestana (2002, p. 6)

Todos esses sinais acima descritos são enviados por duas frequências portadoras L1 e L2 respectivamente com 1.575,42 MHz e 1.227,60 MHz (figura 13). Cada satélite transmite nessas duas frequência, mas com códigos diferentes, selecionados dessa maneira para terem baixa correlação entre os códigos enviados pelos outros satélites [4].

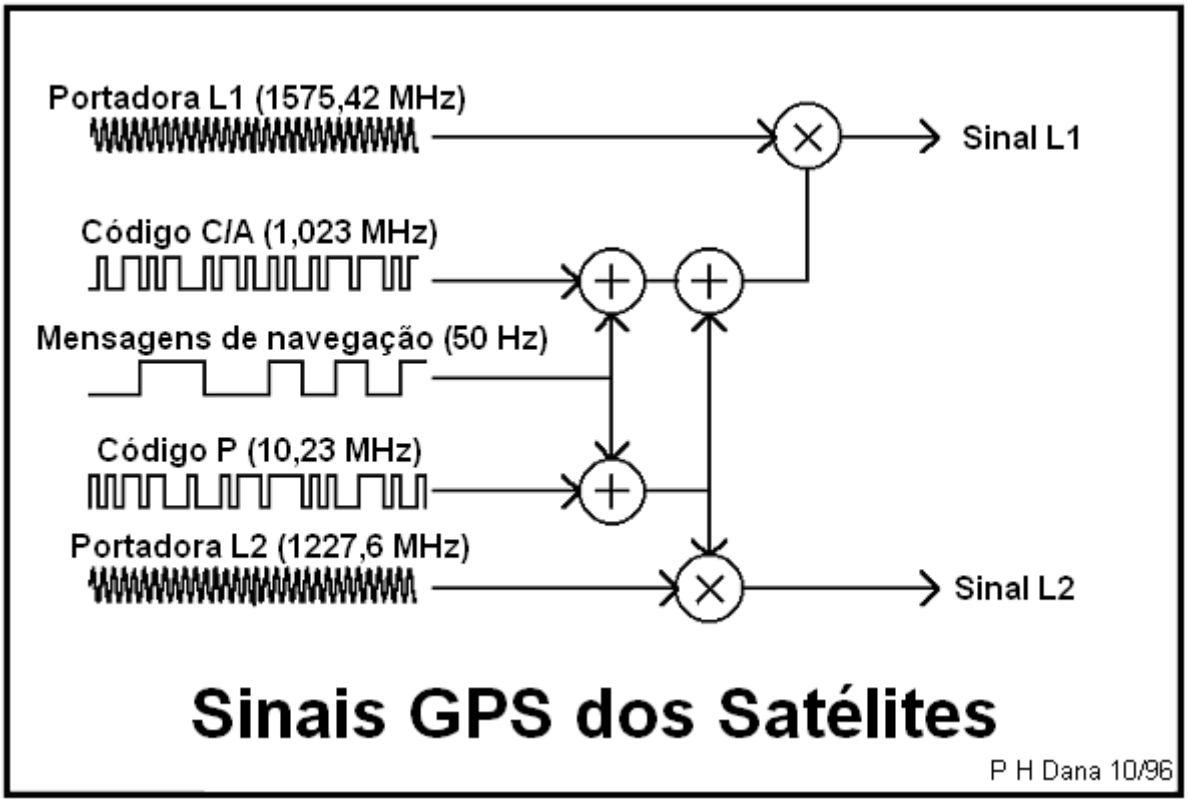


Figura 13- Sinais dos satélites

Fonte: Peter H. Dana

Para que haja precisão mínima no sistema, cada receptor GPS informará a mesma hora, minuto e segundo no mesmo instante em qualquer local do planeta. Isto é possível porque os satélites possuem internamente relógios atômicos muito precisos, na ordem de nano segundos.

Preocupados com o uso inadequado do sistema, o departamento de defesa americano implantaram duas opções de precisão: para usuários autorizados (forças armadas americanas) e usuários não autorizados (demais usuários). Para os usuários não autorizados, o código utilizado é o C/A, modulado somente dentro da portadora L1. Para os usuários autorizados, o código é o P, modulado em ambas as portadoras L1 e L2. A precisão para uso civil é da ordem de 5 metros, já para uso militar pode chegar à ordem de decímetros.

2.3 O receptor GPS

O módulo receptor utilizado será o modelo GT-320R da UniTraQ que recebe os sinais dos satélites e os transforma em níveis TTL ou RS-232. A saída RS232 pode ser ligada diretamente ao *Kit* de desenvolvimento da Digilent sem nenhuma adaptação.

Por se tratar de um módulo de uso civil, a recepção dos sinais dos satélites é realizada pela frequência L1 (*C/A code*). O módulo utiliza o Datum WGS-84.

A comunicação com o *kit* FPGA da Digilent é realizada através do protocolo padrão NMEA-0183 V3.01 com uma taxa de transferência de 4.800 bps.

O receptor tem um consumo menor que 45 mA e sua alimentação pode estar na faixa de 3,8 V ~ 8,0 V.

Suas dimensões são de 8,6 mm x 34 mm x 34 mm (AxLxP) e seu peso é de somente 14 gramas (figura 14).

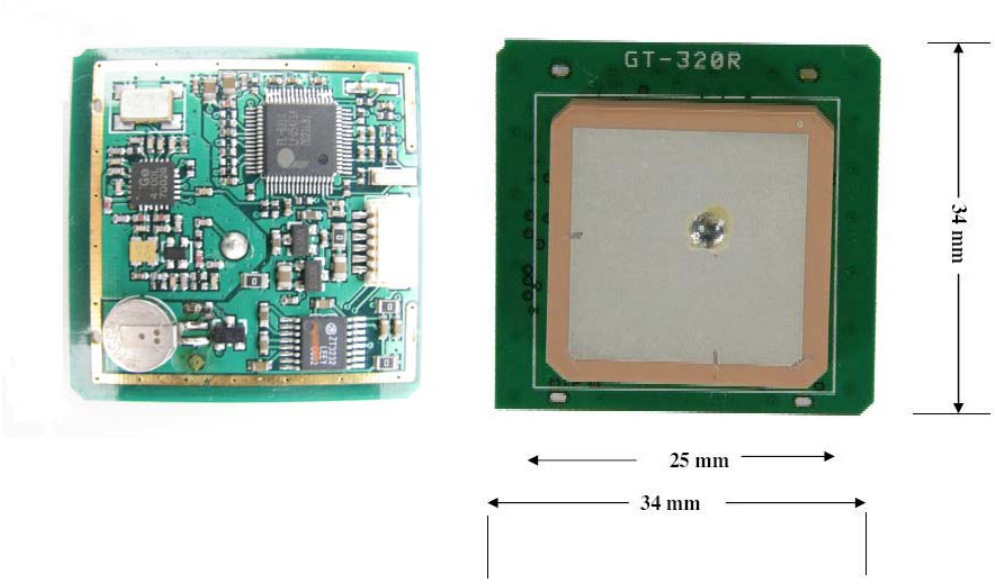


Figura 14 – Módulo GPS

Fonte: Própria

2.4 Padrão NMEA 0183

O NMEA 0183 é um padrão da indústria para o interfaceamento de dispositivos marítimos. Sua primeira versão foi lançada em março de 1983. O NMEA 0183 define as características elétricas dos sinais, protocolo de transmissão de dados, especificação de tempo e formato das sentenças.

O protocolo NMEA tornou-se padrão para interfacear dispositivos de navegação, por exemplo, receptores GPS e DGPS (*Differencial* GPS), e está baseado na interface RS232.

As informações são transmitidas dos satélites para os receptores GPS na forma de sentenças com um comprimento máximo de 80 caracteres. A sentença GPGSV possui 120 caracteres, para que possa ser enviada, a mesma deve ser dividida em sentenças menores que não ultrapassam 80 caracteres.

Todo o início de sentença NMEA começa com um "$" e termina com [CR][LF], como exemplo temos a seguinte sentença.

$GPRMC,154232,A,2758.612,N,08210.515,W,085.4,084.4,230394,003.1,W*43[CR][LF]

Os cincos caracteres seguintes ao "$" são conhecidos como campo de endereços; o restante da linha são campos de dados delimitados por vírgulas. Os dois primeiros caracteres do campo de endereços são conhecidos como *Talker-ID*. O *Talker-ID* identifica o equipamento transmissor, para os dispositivos GPS o *Talker-ID* é o "GP". Os três caracteres restantes descrevem o tipo de sentença, no caso RMC (*Recommended Minimum Navigation Information*).

O módulo receptor fornece, após adquirir os dados do satélite, as seguintes mensagens no formato MNEA: GPGGA, GPGLL, GPGSA, GPGSV, GPRMC, GPVTG, GPZDA.

Destas mensagens, a de maior interesse é a GPRMC, pois ela fornece as informações de tempo, data, posição, curso e velocidade que serão utilizadas para gerar a trajetória percorrida pelo Data Logger. Seu formato segue abaixo:

$GPRMC,HHMMSS.sss,A,DDMM.mmmm,d,DDDMM.mmmm,d,z.z.y.y,ddmmyy,d.d,v*h

Onde:

$GPRMC	-	Cabeçalho
HHMMSS.sss	-	Posição Fixa UTC
A	-	Status (A=Valido, V=Inválido)
DDMM.mmmm	-	Latitude DD: Graus (00..90) MM.mmmm: Minutos (00.0000 .. 59.9999)
d	-	Direção Norte – Sul
DDMM.mmmm	-	Longitude DD: Graus (00..180) MM.mmmm: Minutos (00.0000 .. 59.9999)
d	-	Direção Oeste – Leste
z.z	-	Velocidade em nós

y.y	-	Curso sobre o chão
ddmmyy	-	Tempo Fixo UTC
d.d	-	Variação Magnética (0.0 .. 180.0)
v	-	Sensibilidade da Variação
*h	-	Checksum

Na aplicação pretendida, que é armazenar os dados enviados pelo módulo GPS para traçar posteriormente sua trajetória, serão importantes somente os campos latitude, longitude, direção, data, hora, status e velocidade. Os demais campos, como por exemplo, variação magnética não serão usados nesse trabalho.

2.5 Sistemas alternativos de posicionamento global

O sistema GPS do governo americano não é o único sistema de posicionamento existente, o GLONASS (*Global Navigation Satellite System*) é um sistema de posicionamento geográfico semelhante ao GPS. Este sistema conta com uma constelação de 24 satélites divididos em três orbitas, e pertence à Federação Russa. Os planos orbitais dos satélites GLONASS são de 64.8º, diferentemente dos GPS que é de 55º. Esta inclinação maior beneficia os usuários localizados nas latitudes altas e/ou baixas, uma vez que os satélites viajam mais ao norte ou ao sul. Sua orbita também é menor, estando a 19.100 km e o GPS a 20.200 km.

Outro sistema de posicionamento é o Galileo. Este sistema atualmente está sendo construído pela União Europeia e pela Agencia Espacial Europeia. Contrariamente ao projeto russo e americano que tem origem militar, a proposta do sistema Galileo é de ser um sistema civil. O sistema foi planejado para ter uma maior precisão que os sistemas GPS e GLONASS.

O sistema possuirá 30 satélites e será inter-operável com os sistemas GPS e GLONASS, permitindo uma maior cobertura. Está prevista sua entrada em funcionamento em 2010.

A China está iniciando seu programa de instalação de satélites de posicionamento global, conhecido como *Compass*. O sistema completo será composto de 30 satélites e deverá entrar em operação em 2015 para concorrer com o sistema americano, russo e europeu. O primeiro satélite foi lançado no início de abril de 2009 e chama-se Beidou II.

Muitos dos módulos receptores de sinais GPS são fabricados em regime de OEM (*Original Equipment Manufacturer*), que são dispositivos que não são comercializados aos

consumidores finais, ou seja, são vendidos a outras empresas que os integram a um produto final.

Como foi mencionado anteriormente, a comunicação com os módulos GPS são baseados na RS232. Existem diversos dispositivos que podem interfacear com os módulos GPS através da comunicação serial, entre eles, os microcontroladores, microprocessadores, computadores pessoais etc., ou desenvolvendo a própria comunicação serial em algum dispositivo programável, como por exemplo, um FPGA (*Field Programmable Gate Array*) entre outros dispositivos programáveis. O FPGA, por ser um dispositivo bastante flexível do ponto de vista de implementação de circuitos digitais, foi escolhido para executar a tarefa de interfacear com o módulo GPS.

3 DISPOSITIVOS LÓGICOS PROGRAMÁVEIS

Os dispositivos lógicos programáveis foram inventados no final dos anos 70, e deste então têm-se tornado muito popular entre os engenheiros de desenvolvimento. Essa popularidade está se refletindo como um dos setores da indústria de semicondutores que mais tem crescido nos últimos anos. O motivo dos dispositivos lógicos serem tão amplamente utilizados é porque oferece a máxima flexibilidade de projeto, menor prazo de colocação do produto no mercado, uma melhor concepção de integração e possibilidade de ser reprogramado inúmeras vezes, inclusive em campo.

Com o mercado exigindo cada vez mais produtos com características de desempenho complexas, os fabricantes estão sendo obrigados a aumentar o nível de integração dos circuitos em seus produtos. Diminuído a quantidades de CIs e fazendo com que os mesmos executem tarefas mais complexas. O uso de dispositivos lógicos programáveis tem auxiliado muito os projetistas nesse desafio. Particularmente, o FPGA (*Field Programmable Gate Array*) aumenta em muito a eficiência do projeto, devido a sua capacidade de atender apenas as necessidades do produto. O uso de circuitos digitais convencionais, como portas lógicas, microprocessadores e microcontroladores com certas características que não são aproveitáveis nos produtos, muitas vezes podem comprometer o desempenho e o custo do mesmo. Com o uso de FPGAs, obtém-se uma adequação da função do CI ao projeto ótimo do sistema.

Outra vantagem muito importante que os FPGAs proporcionam é o uso eficiente da placa de circuito impresso (PCI). Isso se deve ao fato que esses componentes conseguem integrar muitos CIs convencionais, reduzindo o tamanho da PCI [11].

Com a complexidade dos sistemas digitais aumentando, cada vez mais componentes serão necessários na placa de circuito impresso, o que aumenta a taxa de falhas do sistema como um todo. Um produto construído com FPGAs consegue reduzir drasticamente este número, obtendo-se uma melhor confiabilidade.

3.1 VHDL

VHDL é uma linguagem de descrição de *hardware*. Ela descreve o comportamento de circuitos de sistemas eletrônicos digitais.

VHDL representa *VHSIC Hardware Description Language*. VHSIC é abreviatura de *Very High Speed Integrated Circuit*, uma iniciativa do departamento de defesa dos Estados Unidos na década de 80.

VHDL é destinado à síntese, simulação e documentação de circuitos. Apesar de que em VHDL seja possível simular todas as construções da linguagem, nem todas as construções são possíveis sintetizar.

A principal motivação para usar VHDL é por ser uma linguagem padrão independe de fornecedor ou tecnologia, portanto portável e reutilizável.

As duas principais aplicações de VHDL estão no campo de dispositivos lógicos programáveis (FPGA e CPLD) e no domínio ASIC (*Application Specific Integrated Circuit* – Circuito Integrado de Aplicação Específica).

3.2 Kit de desenvolvimento FPGA

Esta placa é fabricada pela empresa Digilent Inc. (figura 15), está baseada no componente XC3S500E da família Spartan 3E. Devido à grande capacidade desse dispositivo, é possível a implementação de circuitos digitais, dos mais simples, aos mais complexos, como os *soft cores*, que são microprocessadores/microcontroladores ou DSPs embarcados dentro de um FPGA. A própria Xilinx disponibiliza gratuitamente o *soft core* Picoblaze. O Picoblaze é um microcontrolador de 8 bits de arquitetura RISC (*Reduced Instruction Set Computer* – Computador com um Conjunto Reduzido de Instruções), o Picoblaze é otimizado para eficiência e ocupa em torno de 200 células lógicas, cerca de 2,5% de um FPGA XC3S500E. Como não foi projetado para alta performance, ele é compacto e flexível, podendo ser usado em processamento de controle, dados simples e operações com I/O (*Input/Output* – Entrada/Saída). Outro *soft core* fornecido pela Xilinx é o Microblaze. Trata-se de um microcontrolador RISC de 32 bits, com ele é possível até rodar sistemas operacionais como o Linux Kernel. Este *soft core* não é distribuído livremente, e deve ser adquirido junto a Xilinx.

Por se tratar de um kit de desenvolvimento, o fabricante incluiu uma grande variedade de dispositivos na placa, como saída VGA, memória DDR, *ethernet* com o PHY (*Physical Layer*) e conector com trafo para ethernet 10/100 Mbps. No caso se projetar algum dispositivo que tenha acesso a *ethernet*, somente o MAC (*Media Access Control* – Controle de Acesso ao Meio) deverá ser implementado em alguma linguagem de descrição de *hardware*.

A escolha desse kit levou em consideração o fato de que a fabricante do kit, a Digilent Inc., ser homologada pela Xilinx, tornando a tarefa do desenvolvimento mais fácil, pois no site de ambos os fabricantes existir uma grande variedade de literatura disponível.

Entre as principais características dessa placa, pode ser citado.

Spartan-3E XC3S500E

Até 232 pinos de I/O

500K Gates

10.000 células lógicas

4 Mbit de memória Flash para configuração

64 Mbyte de memória DDR SDRAM

16 Mbyte de memória paralela Flash

Para armazenamento de configurações do FPGA

Armazenamento do código do microcontrolador Microblaze

16 Mbit de SPI serial Flash

LCD 2 linha x 16 caracteres

2 portas RS 232

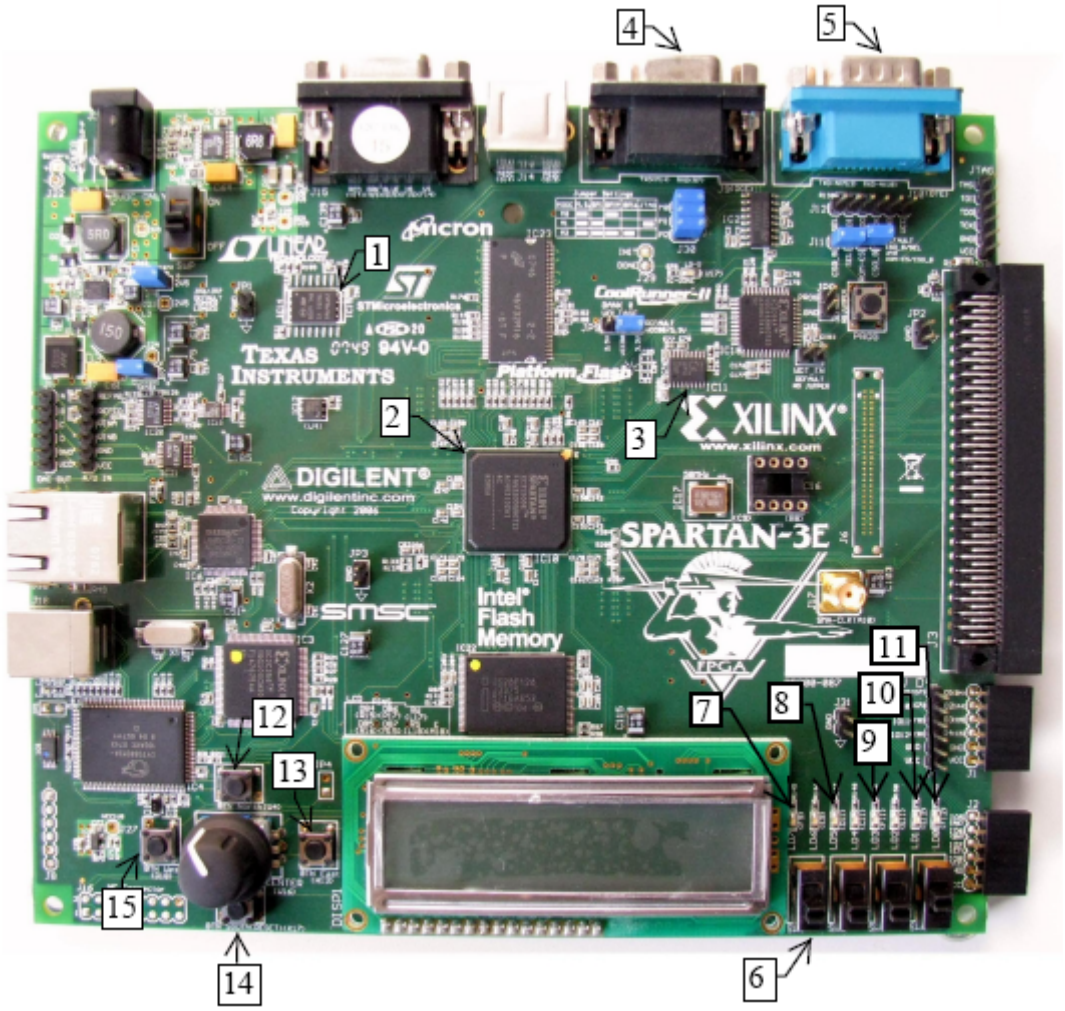
Conversores ADC / DAC

Ethernet PHY 10/100

Porta VGA

Entre outros detalhes.

Dentre todas essas características, uma das que será usada será as duas portas RS 232 (itens 4 e 5 da figura 15). Como já é de conhecimento, os módulos GPS transferem seus dados pela porta serial. Assim sendo, uma das portas estará constantemente conectada ao módulo, recebendo as informações de posicionamento, a outra porta será usada para transferir os dados do Data Logger para o PC.



1	Memória Flash SPI	6	Apagar memória	11	LED Erro
2	FPGA XC3S500E	7	LED Gravando	12	Botão de DOWNLOAD
3	Platform Flash PROM	8	LED Lendo	13	Botão de STOP
4	Serial TX	9	LED Gravando/Apagando	14	Botão de RESET
5	Serial RX	10	LED Reset	15	Botão de Start

Figura 15 – Kit FPGA Spartan 3E

Fonte: Própria

4 COMUNICAÇÃO SERIAL E O PADRÃO RS232

RS-232 é um padrão de telecomunicações para comunicação binária entre dois dispositivos utilizando a porta serial. Os dispositivos são geralmente referenciados como DCE (*Data Circuit-terminating Equipment* – Equipamento de Comunicação de Dados) e DTE (*Data Terminal Equipment* – Equipamento Terminal de Dados), por exemplo, um computador e um modem, respectivamente.

Atualmente o padrão RS-232 chama-se EIA232. A EIA232 apenas especifica as características elétricas dos circuitos e a numeração dos pinos. Outras características como o conector em forma de "D", o uso de código ASCII, formato dos dados e comunicação assíncrona não são obrigatoriamente partes do EIA232. Mas quando nos referimos ao "padrão EIA232" todas estas características aparecem juntas, de modo a tornarem-se efetivamente obrigatórias.

Neste protocolo os caracteres são enviados um a um como um conjunto de bits. A codificação mais comumente usada é a assíncrona, que usa um bit de início de transmissão (*start bit*), seguido de sete ou oito bits de dados, opcionalmente um bit de paridade e 1, 1,5 ou 2 bit de parada (*stop bit*) (figura 16).

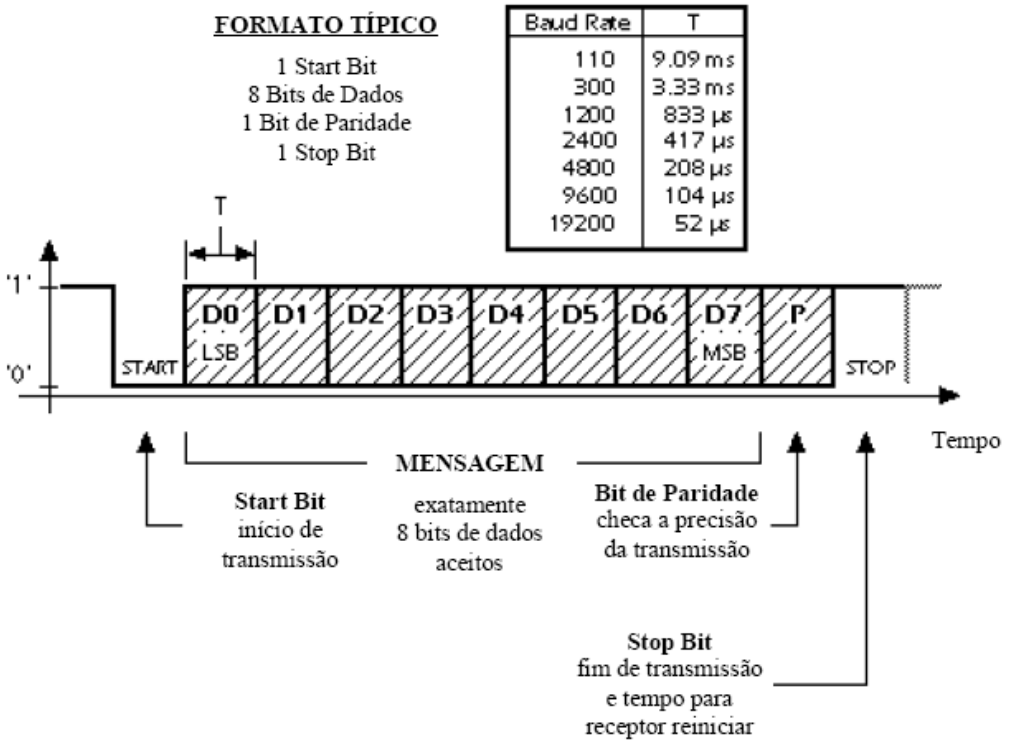


Figura 16 – Bitstream

Fonte: Edmur Canzian

Outro componente importante do sistema Data Logger é a memória aonde os dados serão armazenados temporariamente, até serem transferidos ao PC. O kit da Digilent Inc. possui dois tipos de memória não volátil que poderiam ser utilizadas para essa função, uma flash paralela de 128 Mbit ou uma serial flash SPI de 16 Mbit. Foi optado por utilizar a memória serial por seu protocolo de comunicação SPI ser bastante simples e altamente eficiente, atendendo ao propósito do trabalho.

5 BARRAMENTO SPI

O SPI (*Serial Peripheral Interface*) é um barramento que permite a comunicação entre os microcontoladores e chips periféricos ou a comunicação entre dois ou mais microcontroladores. O SPI é algumas vezes chamado de interface a quatro fios, podendo ser utilizado para interfacear dispositivos como, por exemplo, LCDs, sensores, memórias, ADC, DAC, entre outros.

O barramento SPI usa um protocolo síncrono para comunicação. Onde a recepção e a transmissão são iniciadas pelo master e sincronizadas pelo sinal de clock. A interfece SPI permite conectar vários dispositivos SPI que são selecionados pelo sinal *Chip Select* (CS).

O barramento SPI consiste dos seguintes sinais.

MOSI – Master Out Slave In

MISO – Master In Salve Out

SCLK – Serial Clock

CS – Chip Select

O modo SPI utilizado para comunicação com a memória M25P16 (STMicroelectronics) será o de polaridade (CPOL) e fase (CPHA) igual a 1 (figura 17).

A velocidade de comunicação entre a memória SPI e o seu controlador que estará implementado dentro do FPGA, será de 12,5 MHz. Esta velocidade de comunicação é mais do que suficiente para o projeto, pois a velocidade com que os dados são transferidos do módulo GPS para o Data Logger é padronizado pela MNEA em 4.800 bps. Nesse caso não há a necessidade de se criar nenhum tipo de memória *cache* internamente no FPGA para evitar que os dados enviados pelo GPS sejam perdidos.

A utilização dessa memória exige alguns cuidados, como por exemplo, antes de iniciar uma nova gravação em um endereço que já havia algum dado gravado previamente, deve-se primeiro ser apagada, isso porque, quando a memória é apagada, seus bits internos são setados para ‘1’. Quando gravamos algum dado, somente os bits com valor lógico ‘0’ a serem gravados serão alterados, os bits com valor ‘1’ não são sobrescritos, assim, se não for efetuado o apagamento antes de uma nova gravação, os bits que eram ‘1’ e passam para ‘0’ são alterados, porém os bits que eram ‘0’ e deveriam passar para ‘1’, não são alterados, tornando aos dados incoerentes.

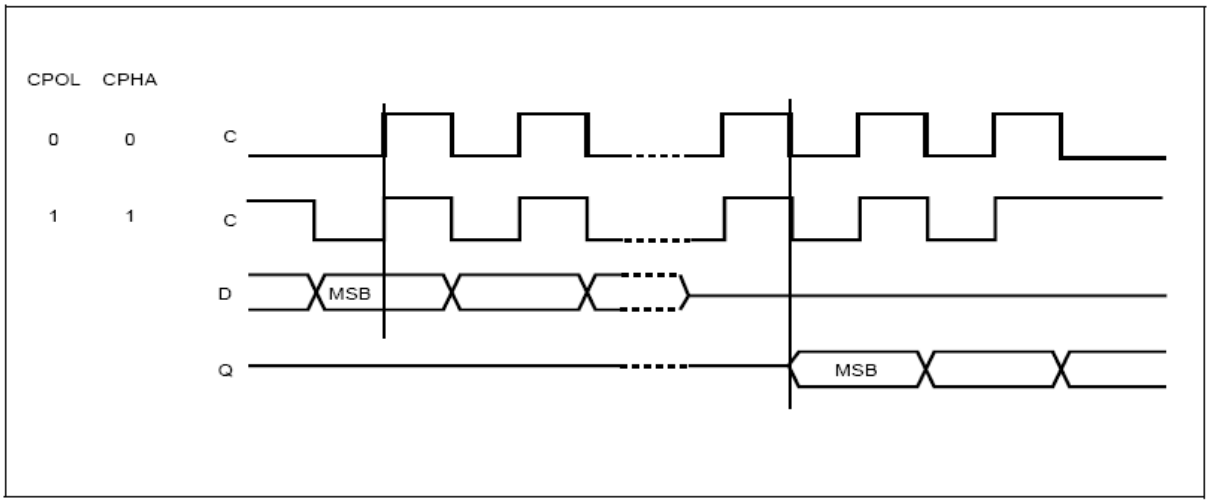


Figura 17 – Representação dos modos SPI

Fonte: M25P16 *Datasheet* (2004, p. 7)

6 DESENVOLVIMENTO

O principal desenvolvimento desse trabalho é a elaboração do desenho de hardware com a ferramenta ISE Webpack® fornecido pela Xilinx, utilizando a linguagem VHDL, de um Data Logger para armazenar as sentenças envidas por um módulo GPS em uma memória não volátil. Posteriormente os dados armazenados na memória do kit da Digilent serão transferidos ao PC através da porta serial, os mesmos serão apresentados na forma de um mapa utilizando a API do Google Maps onde poderá ser visto a trajetória feita pelo GPS, desde o momento do início da gravação até o momento em que a gravação foi encerrada.

6.1 ISE Webpack

A ferramenta ISE Webpack é uma GUI (*Graphical User Interface*) – Interface Gráfica do Usuário – desenvolvido pela Xilinx® que facilita a criação de projetos, sua edição, simulação, síntese e gravação no dispositivo. Sua aparência poder ser vista na figura 18.

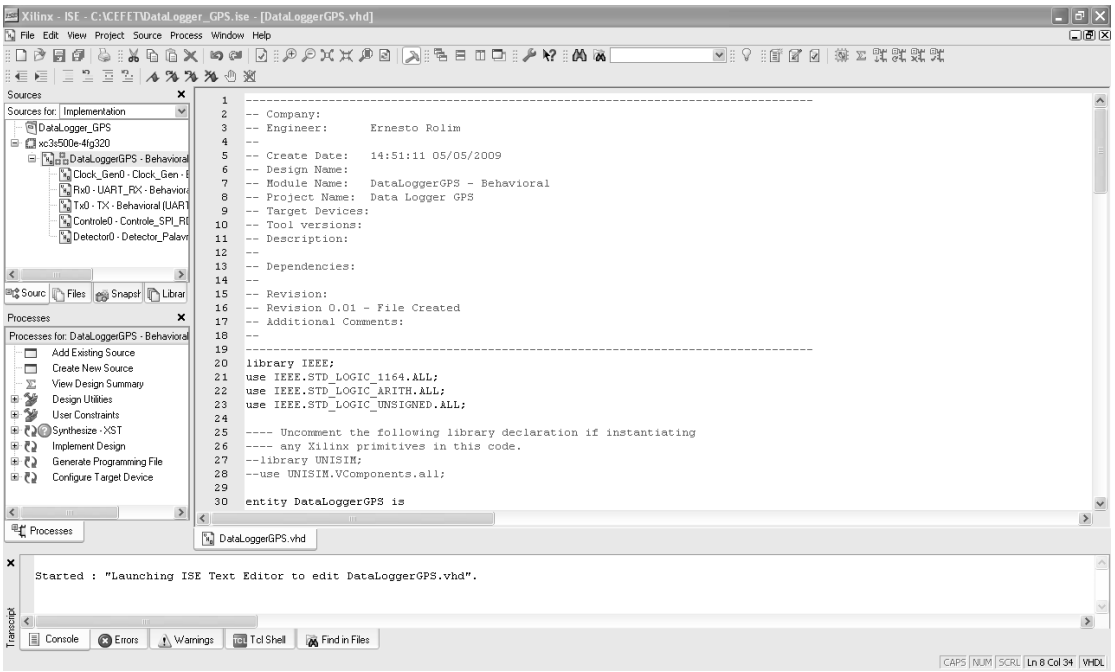


Figura 18 – ISE Webpack

Fonte: Própria

A parte central é denominada de *wokrspace*, é nesse local que é feita a digitação dos códigos VHDL que descrevem o projeto, este espaço também se destina a criação de esquemáticos quando o projeto é descrito graficamente.

No canto esquerdo superior, na caixa *Source*, fica a árvore do projeto com seus respectivos arquivos. Abaixo, na caixa *Process*, ficam os processos que estão disponíveis no momento para cada arquivo. Estes processos geralmente são os de síntese, checagem de sintaxe, arquivos de restrições do usuário, geração de arquivo de programação, entre outros.

Na parte inferior, fica a caixa de diálogo da ferramenta, aonde são mostradas informações relevantes sobre o processo de síntese, simulação etc.

6.2 Sequência de desenvolvimento de sistemas em FPGA

A sequência completa dos passos de um projeto em HDL pode ser observada na figura 19.

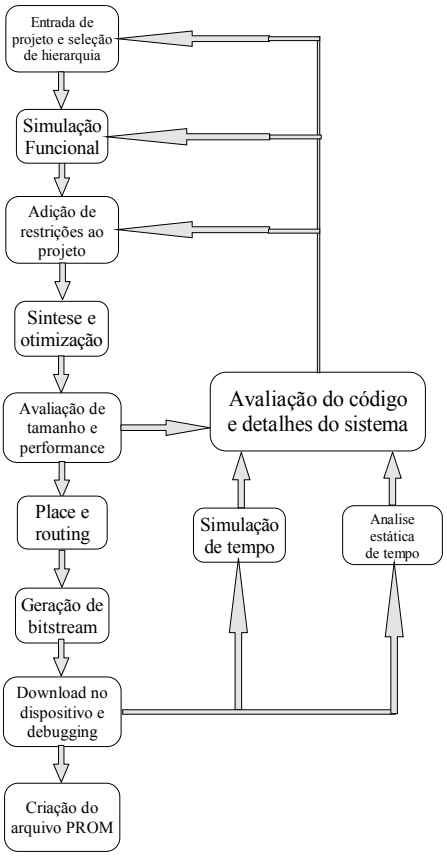


Figura 19 – Fluxograma de desenvolvimento em HDL

Fonte: Synthesis and Simulation Design Guide 10.1 (p. 22)

Inicialmente cria-se o projeto hierárquico na ferramenta ISE Webpack, modelando o projeto usando alguma linguagem HDL, como por exemplo, VHDL ou Verilog. Efetua-se a checagem verificando se não há algum tipo de erro de sintaxe para que possa ser simulado.

No processo de simulação funcional, é possível verificar o comportamento do projeto para ver se atende às especificações iniciais. Isto é alcançado usando sinais de entrada e verificando os sinais de saída.

Após o projeto ter sido simulado e atendido os requisitos, o passo seguinte é a introdução das restrições do projeto. Estas restrições geralmente estão relacionadas ao tamanho final do circuito, restrições de tempo e velocidade.

O passo seguinte é o processo de síntese e otimização, a síntese é o processo no qual é gerada as interconexões entre as *Look Up Table* (LUT) internas do FPGA, formando assim os circuitos.

As LUT são pequenos conjuntos de circuitos lógicos básicos que são usados para implementar qualquer função lógica de N entradas e uma saída. N geralmente encontra-se entre 2 e 6, sendo o mais popular as LUTs de 4 entradas.

Um ponto muito importante quando se trabalha com lógica programável, é saber o tamanho físico final do circuito sintetizado e o seu desempenho. A importância dessas informações é tão grande que pode ser determinante na escolha do FPGA. Por exemplo, se o circuito sintetizado ficar muito grande do ponto de vista físico, ou seja, necessite de uma quantidade muito grande de lógica para ser implementado, pode ser que o FPGA inicialmente escolhido não atenda a essa especificação, tendo que ser escolhido outro FPGA, que pode impactar no preço final do produto, podendo torná-lo inviável.

Somente com os dados do tamanho final da síntese nas mãos, o projetista não tem ainda condições de escolher o FPGA que atenda aos requisitos do projeto.

Muitas das especificações de projetos em lógica programável possuem alguma restrição no que diz respeito ao desempenho com relação à velocidade. Em circuitos de alta velocidade, geralmente há a necessidade de se fazer um estudo do roteamento dos sinais dentro do FPGA evitando um *delay* muito grande entre os sinais, é exatamente nessa fase do desenvolvimento que estas restrições são analisadas e implementadas, bem como atribuição das nets aos pinos do FPGA. Em sistemas críticos com relação ao tempo, a não observação desses detalhes pode trazer resultados inconsistentes e difíceis de serem depurados.

O bitstream é o arquivo compilado que pode ser baixado para o dispositivo diretamente utilizando alguma ferramenta JTAG. JTAG é um padrão do IEEE 1149.1 para o acesso as portas de teste de muitos circuitos integrados, podendo o projetista depurar, testar e

programar os circuitos eletrônicos. Após o download no dispositivo, é possível a realização do processo de *debugging* para verificar o funcionamento prático do circuito.

Tendo o circuito funcionado conforme esperado, é o momento de se gerar o arquivo PROM. No arquivo PROM está contida a informação das ligações dos circuitos internos do FPGA para desempenhar a função pretendida. Este arquivo será gravado em uma memória não volátil (figura 15 – componente 3) e será carregado no FPGA cada vez que a alimentação for ligada, configurando-o para trabalhar conforme especificado.

6.3 Visão geral do sistema Data Logger para GPS

O sistema completo é composto por um módulo GPS, um kit de desenvolvimento contendo um FPGA e um *software* de captura de dados no PC.

As sentenças provenientes do módulo GPS são lidas e armazenadas pelo kit em uma memória Flash SPI M25P16. O *software* de captura instalado em um PC recupera essas informações e as apresenta na forma de um mapa, com a trajetória efetuada pelo GPS, e na forma de uma tabela, com informações de posição, hora, direção etc.

O desenho de *hardware* que representa o Data Logger na figura 20 foi dividido em blocos funcionais que foram implementados dentro do FPGA.

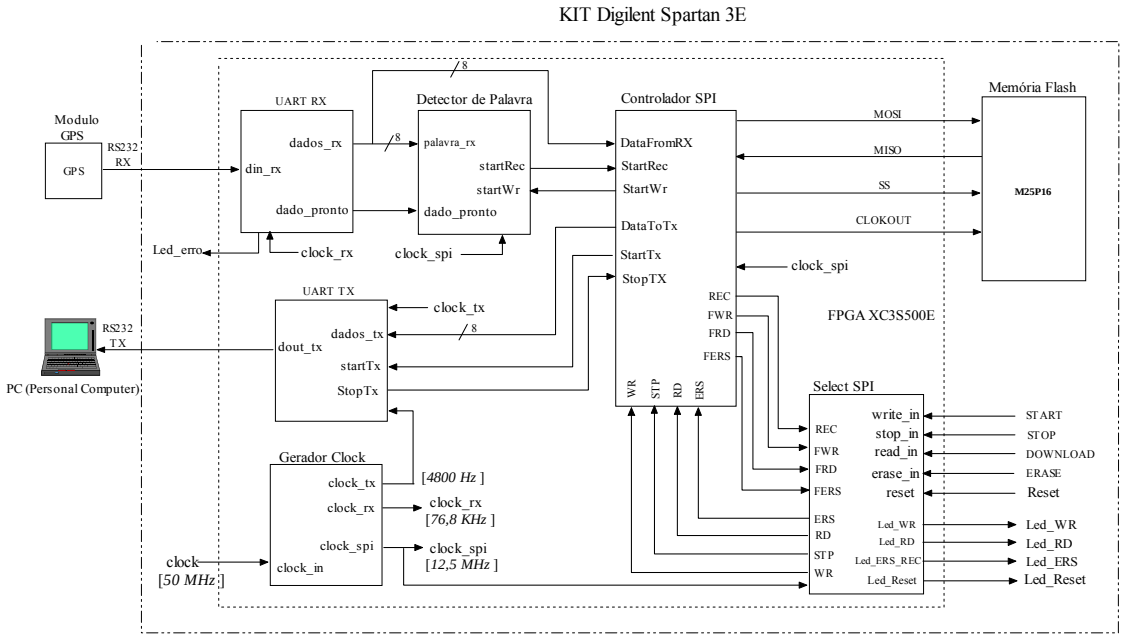


Figura 20 – Diagrama Data Logger

Fonte: Própria

O tracejado externo representa o Kit de desenvolvimento fabricado pela Digilent Inc.. Nele estão incluídos, entre outros CIs, o FPGA XC3S500E e a memória Flash M25P16. O tracejado interno representa o FPGA, e os blocos internos são os circuitos que foram implementados para interfacear a comunicação com o módulo GPS, o PC e a memória M25P16. Cada bloco é interconectado aos outros através de sinais. O sistema possui 4 botões de pressão com as funções de iniciar a gravação, parar a gravação, ler o conteúdo da memória e resetar o sistema, e uma chave do tipo H-H para iniciar o processo de apagamento, representados respectivamente pelos sinais START, STOP, DOWNLOAD, Reset e ERASE na figura 20.

No diagrama acima, por questões de simplicidade, não está representado o CI MAX232, utilizado para realizar a interface física entre a UART (*Universal Asynchronous receiver/transmitter* – Receptor/transmissor assíncrono universal) do Kit e a do PC.

6.4. Descrição geral do funcionamento do diagrama

As operações realizadas pelo Data Logger muitas vezes necessitam de que mais de um bloco atue para a sua correta execução. Abaixo será descrita cada função os respectivos blocos envolvidos.

6.4.1 Detecção de Sentença

O sinal proveniente do módulo GPS é capturado pelo bloco UART RX (figura 21). Este bloco recebe serialmente cada palavra enviada pelo módulo GPS, verifica se o byte recebido é válido, se for, o disponibiliza em sua saída (sinal Palavra_rx) e sinaliza para o bloco Detector de Palavra através do sinal dado_pronto que existe um byte válido a ser detectado.

O processo de detecção da sentença inicia quando o bloco Controlador SPI está em modo de gravação e avisa através do sinal StartWR para o bloco Detector de Palavra iniciar o procedimento de detecção. Neste momento o bloco Detector de Palavra passa a avaliar os sinais enviados pelo bloco UART RX (palavra_rx e dado_pronto).

O bloco Detector de Palavra passa a verificar a sentença desejada e sinaliza ao bloco Controlador SPI os bytes que podem ser gravados. Para isto o bloco Detector de Palavra aguarda a detecção do cabeçalho $GPRMC da sentença e em seguida a cada byte da sentença lido, que deve ser gravado, avisa ao Controlador SPI através do sinal StartRec que o byte

presente no sinal palavra_rx deve ser gravado. Quando o bloco Detector de Palavra receber o byte de fim de sentença “LF” ele para de enviar o sinal para o bloco Controlador SPI gravar e volta a aguardar uma nova sentença, repetindo o ciclo. A figura 21 ilustra os blocos principais para a detecção de sentença.

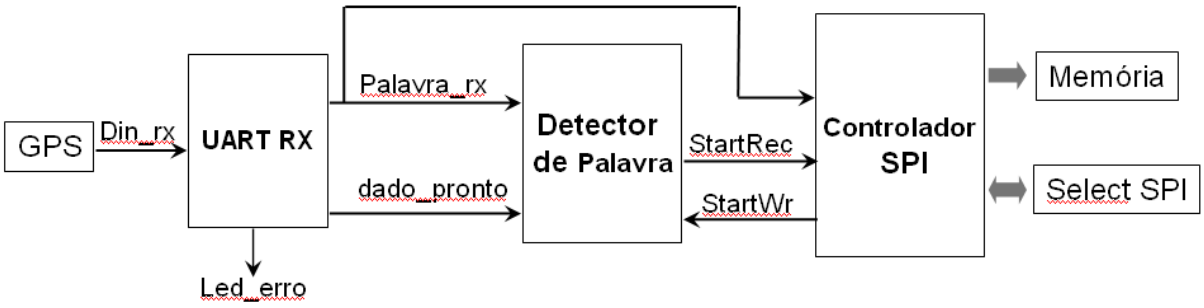


Figura 21 – Principais blocos da detecção de sentença

Fonte: Própria

6.4.2 Ativação das operações da memória

Sempre que o sistema Data Logger é inicializado, seja quando for alimentado ou resetado, o bloco Controlador SPI verifica o estado da memória, “vazia” ou “com gravação”. O estado da memória é representado no circuito pelo sinal REC, apresentado na figura 22. Este sinal é utilizado pelo bloco Select SPI para avaliar quais as funções estão disponíveis no momento.

Se a memória estiver vazia, o bloco Select SPI analisa somente se o botão Start (item 15 da figura 15) foi acionado e avisa o bloco Controlador SPI quando este evento ocorrer. Se a memória estiver com dados gravados, o bloco Select SPI analisa somente o acionamento dos botões Download (item 12 da figura 15) e Erase (item 6 da figura 15). Se o botão RESET (item 14 da figura 15) for acionado é reiniciado todo o sistema.

As funções dos sinais WR, STP, FWR, RD, FRD, ERS e FERS representados na figura 22 são utilizados nas operações de gravação, leitura e apagamento da memória.

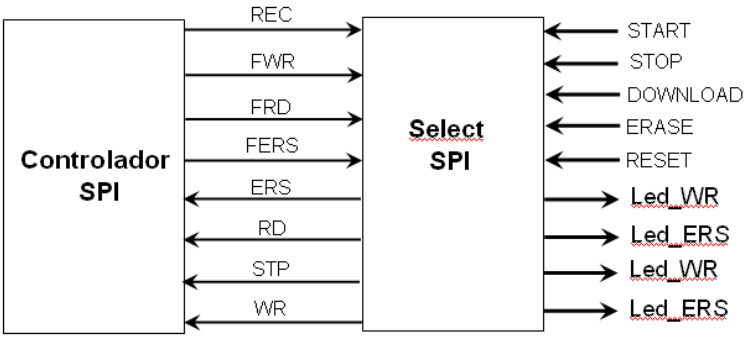


Figura 22 – Sinais de gravação, leitura e apagamento da memória

Fonte: Própria

6.4.3 Operação de gravação de dados na memória

Para estar no processo de gravação o sinal REC deve ter o nível “0” e o bloco Select SPI estará analisando o acionamento do botão Start (item 15 da figura 15).

Ao ser acionado o botão Start o bloco Select SPI acende o led Led_WR (item 7 da figura 15), como indicação visual que tem uma gravação em andamento, e informa através do sinal WR ao bloco Controlador SPI para o mesmo iniciar o processo de gravação. A partir deste momento o bloco Controlador SPI ativa o bloco Detector de Palavra e passa a aguardar o sinal starRec para gravar o byte na memória, conforme explicado no item 6.4.1 na detecção da sentença. A comunicação entre a memória e o bloco Controlador SPI é realizada pelos sinais SS (Chip Select), ClockOut (clock de 12,5 MHz para a memória), MOSI (Master Out Slave In).

Durante o processo de gravação o bloco Select SPI passa a analisar se o botão Stop (item 13 da figura 15) foi acionado. No momento que este botão for acionado o bloco Select SPI avisa ao bloco Controlador SPI, através do sinal STP (stop), para encerrar o processo de gravação.

Para não correr o risco de gravar uma sentença corrompida, somente quando o bloco Controlador SPI termina de gravar uma sentença completa ele verifica o estado do sinal STP (stop). Se for para encerrar a gravação o bloco Controlador SPI termina o processo de gravação, atualiza o estado da memória e desabilita a detecção da sentença. Se não, continua na operação de gravação.

Quando for finalizada a gravação, o bloco Controlador SPI sinaliza através do sinal FWR o bloco Select SPI, o mesmo apaga o led Led_WR e acende o led Led_ERS (item 9 da figura 15) que indica que a memória está com gravação (figura 23).

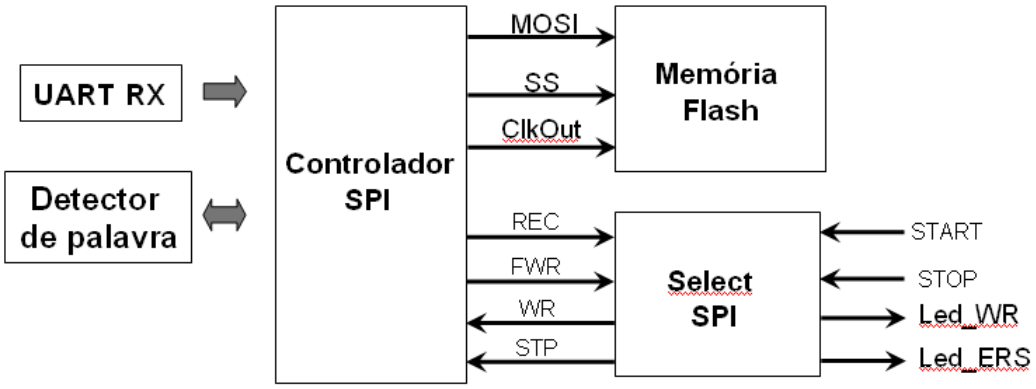


Figura 23 – Blocos da operação de gravação

Fonte: Própria

6.4.4 Operação de apagamento de dados da memória

Para estar na operação de apagamento, o estado da memória deve ser “com gravação” e o bloco Select SPI detectar o acionamento do botão Erase (item 6 da figura 15).

Quando o bloco Select SPI perceber o acionamento do botão Erase, o led_ERS passa a piscar indicando visualmente o processo de apagamento, e o bloco Select SPI sinaliza através do sinal ERS para o bloco Controlador SPI iniciar o processo de apagamento da memória.

Quando o bloco Controlador SPI terminar o processo de apagamento da memória, atualiza o estado da memória (sinal REC), e sinaliza através do sinal FERS para o bloco Select SPI que a operação de apagamento da memória foi concluída. Recebendo este sinal o bloco Seletc SPI apaga o led Led_ERS, indicando que a memória está vazia e disponível para nova gravação (figura 24).

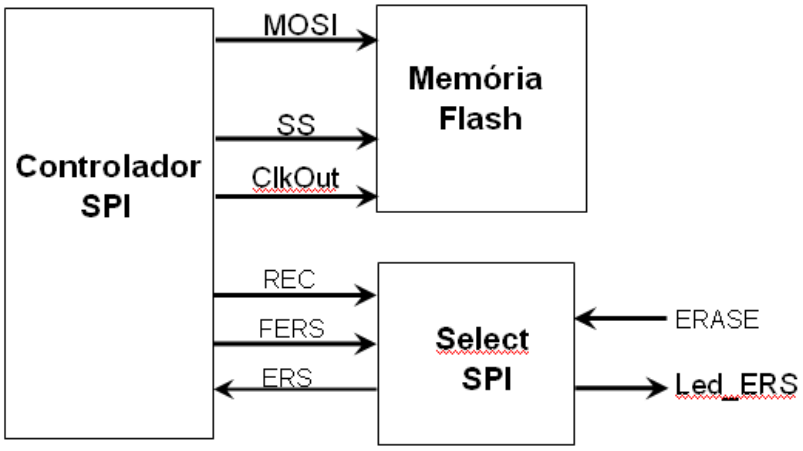


Figura Figura 24 – Blocos da operação de apagamento

Fonte: Própria

6.4.5 Processo de leitura dos dados da memória

Para estar na operação de leitura, o estado da memória deve ser “com gravação” e o bloco Select SPI detectar o acionamento do botão Download (item 12 da figura 15).

Quando o bloco Select SPI verificar o acionamento do botão Download, acende o led Led_RD, indicador visual de leitura, e avisa através do sinal RD para o bloco Controlador SPI iniciar a operação de leitura.

O bloco Controlador SPI ao receber o sinal para iniciar a leitura realiza duas operações: lê o byte da memória e envia o byte para UART TX transmitir para o PC.

Na operação de leitura o bloco Controlador SPI solicita o byte para a memória enviando o endereço do mesmo através do sinal MOSI. Quando a memória responde com o byte no sinal MISO, o controlador disponibiliza-o para o UART TX na saída DataToTx e sinaliza através do sinal StartTx para a UART TX iniciar sua execução. Neste momento o bloco Controlador SPI passa a aguarda o final da operação da UART TX através do sinal StopTx.

A UART TX ao receber o byte o envia serialmente para o PC no padrão RS232, ou seja, envia o start bit em seguida os 8 bytes de dados e por fim o stop bit. Quando terminar de enviar o stop bit, avisa ao bloco Controlador SPI que está liberado para receber novo byte através do sinal StopTx.

Esta operação de leitura da memória e envio para UART TX se repete até o bloco Controlador SPI ler da memória o caractere “~” que representa fim de arquivo. Neste momento o bloco Controlador sinaliza através de sinal FRD o bloco Select SPI o fim de leitura, apagando o Led_RD (figura 25).

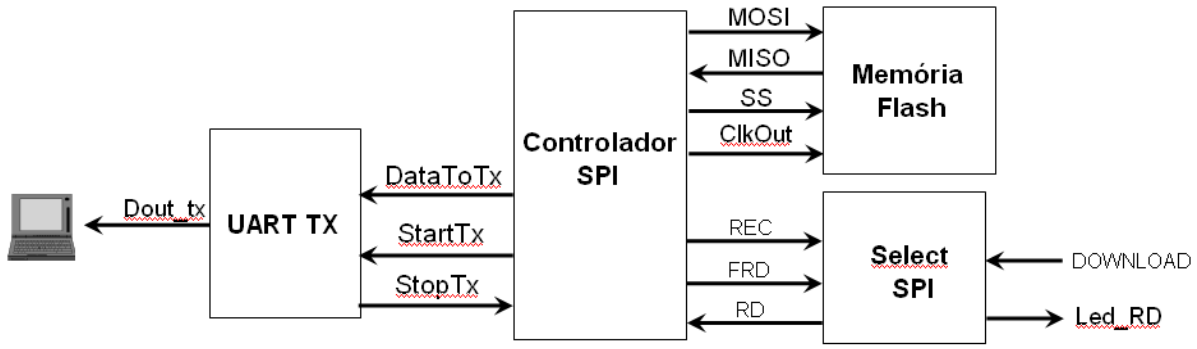


Figura 25 – Blocos da operação de gravação

Fonte: Própria

6.4.6 Tratamento da sentença

A sentença passa por algumas modificações desde a sua detecção pelo bloco Detector de Sentença até a sua efetiva gravação no arquivo ".txt" no PC, pelo Software de Captura.

O módulo GPS utilizado no desenvolvimento disponibiliza quatro sentenças NMEA em suas saídas.

Das sentenças NMEA disponibilizada pelo módulo, a sentença GPRMC foi escolhida por fornecer todas as informações necessárias como, por exemplo: hora, velocidades, posição etc.

Abaixo temos um exemplo de uma sentença enviada pelo módulo GPS:

$GPRMC,180423.65,A,2959.8913,S,05118.4862,W,000.0,356.1,260409,015.7,W,A*2D

O Controlador SPI grava na memória a sentença acima desconsiderando os caracteres de detecção da sentença desejada "$GPRMC," gravando então a sequência a seguir:

180423.65,A,2959.8913,S,05118.4862,W,000.0,356.1,260409,015.7,W,A*2D

Quando o bloco Controlador SPI ler a sequência a cima gravada na memória e enviar para o PC, o Software de Captura formatará os campos hora, latitude, longitude, velocidade e data. Estes campos estão descritos no item 2.4. Além da formatação dos campos o Software de Captura troca o separador de campo de "," para ";", em virtude de que na formatação do campo velocidade utilizamos a vírgula para separação das casas decimais. Mais informações sobre a formatação são explicadas no item 6.6 Software de Captura.

A sentença formatada que efetivamente é gravada no arquivo texto de extensão ".txt" é mostrada abaixo:

18:04:23;A;29.99818;S; 51.30595;W;0,00;26/04/09;

6.5 Descrição interna dos blocos funcionais

A seguir será descrito detalhadamente o funcionamento interno de cada bloco implementado no FPGA, ilustrado na figura 20.

6.5.1 Gerador de clock

Este bloco é responsável pela geração de todos os sinais de clock usados no sistema.

O kit de desenvolvimento sai de fábrica com um oscilador de 50 MHz que é ligado diretamente ao pino de I/O chamado de *Global Buffer*. Os pinos denominados *Global Buffer* são pinos especiais nos FPGA, aonde sinais externos de clocks são inseridos e distribuídos internamente no FPGA.

A geração dos sinais de clock para os outros blocos é bastante simples. O clock de 50 MHz passa por um divisor e na saída desse divisor temos o sinal de clock na frequência desejada.

Para o sinal de clock_spi, o clock de 50 MHz é dividido por 4, ou seja, para cada 4 ciclos de clock de 50 MHz, temos um ciclo do clock_spi, gerando um sinal de 12,5 MHz.

Para o sinal de clock_rx, o processo é semelhante ao clock_spi, exceto que o clock de 50 MHz é dividido por 651, gerando um clock de 76,8 KHz.

O sinal de clock para o bloco de transmissão é gerado dividindo-se o clock de 50 MHz por 5208, gerando um sinal de 4800 Hz para o sinal clock_tx.

6.5.2 UART RX

O bloco UART RX (figura 26) recebe os dados enviados serialmente pelo GPS, os valida e disponibiliza para o bloco Detector de Palavra e para o bloco Controlador SPI. O GPS envia para UART RX o dado no seguinte padrão: um sinal de start bit, o byte de dados e um stop bit.

No estado inicial a UART RX fica aguardando o sinal de start bit, din_rx = ‘0’. Detectado o start bit os próximos 8 bits, o byte de dado, são lidos do bit menos significativo D0 ao bit mais significativo D7, em seguida o sinal de stop bit é lido.

A validação feita pela UART RX é verificar se o stop bit chegou corretamente, se sim, o byte será considerado valido, o sinal dado_pronto passa para ‘1’ e passa a aguardar por um novo start bit. Se o stop bit não chegou corretamente, o byte será considerado invalido, o sinal dado_pronto permanece nível ‘0’ e o led de erro acende. O bloco UART RX retorna para o início do bloco para aguardar novo start bit. No anexo B temos a estrutura do código VHDL desse bloco.

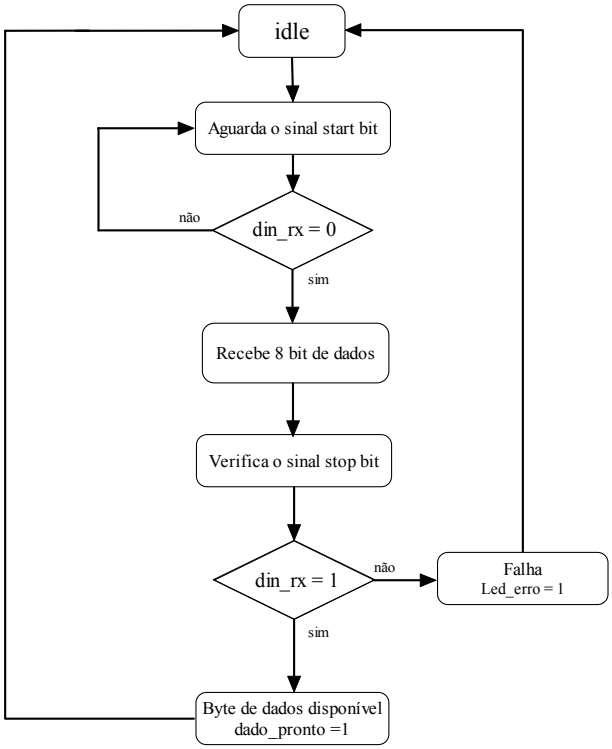


Figura 26 – Fluxograma do bloco UART RX

Fonte: Própria

6.5.3 Detector de palavra

Como mencionado anteriormente, o módulo GPS envia continuamente sentenças NMEA 0183. Estas sentenças iniciam invariavelmente com o caractere ASCII “$”, seguidos por mais cinco caracteres que identifica a sentença em particular. O bloco Detector de Palavra (figura 27) identifica a sentença pelo cabeçalho $GPRMC, essa sentença contem as informações de hora, posição, direção entre outras informações que serão usadas posteriormente para traçar a rota efetuada pelo sistema Data Logger. Detectado o cabeçalho da sentença, os caracteres seguintes serão gravados pelo bloco Controlador SPI até o fim da sentença, quando for detectado o caractere ASCII “LF”.

Inicialmente o bloco fica aguardando que o sinal startWR torne-se ‘1’. Isto indica que o bloco Controlador SPI está em modo de gravação aguardando os bytes que deveram ser gravados.

A partir desse momento o bloco Detector de Palavra passa a aguardar a sentença desejada. Toda vez que o sinal dado_pronto passa para ‘1’ o caractere ASCII enviado pelo bloco UART RX é avaliado. No primeiro momento o bloco Detector de Palavra valida o cabeçalho da sentença “$GPRMC”. Detectado o cabeçalho, a cada novo caráter recebido o

sinal startRec passa para ‘1’, avisando ao bloco Controlador SPI que ele deve gravar o caractere. Quando o bloco Detector de Palavra receber o caractere de fim de sentença “LF” ele retorna para o início do bloco. Se o startWR continuar ‘1’ recomeça a detecção de uma nova sentença, se o startWR for ‘0’ indica que o bloco Controlador SPI não está mais em modo de gravação.

Durante o processo de verificação do cabeçalho, caso seja detectado alguma divergência com relação à palavra esperada, o sistema retorna ao estado inicial, aguardando por um novo início de sentença. No anexo C temos a estrutura do código VHDL desse bloco.

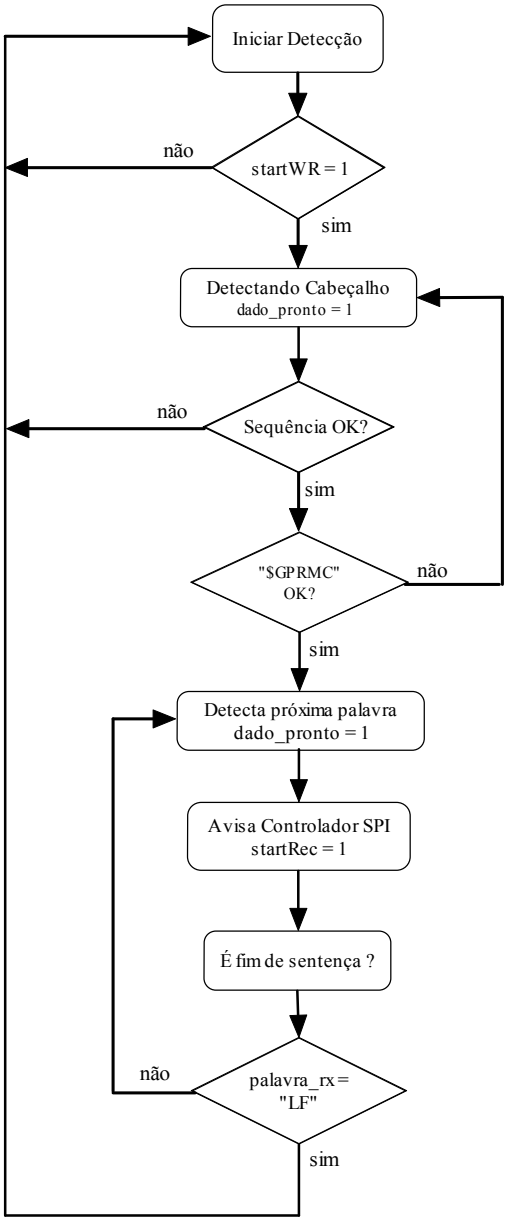


Figura 27 – Fluxograma do bloco Detector de Palavra

Fonte: Própria

6.5.4 Select SPI

Este bloco é responsável por ler o acionamento dos botões de Start, Stop, Download, Erase e Reset, e sinalizar as seguintes operações:

- Gravando na memória: o Led_WR fica acesso durante a gravação;
- Lendo dados da memória: o Led_RD fica aceso durante a leitura;
- Memória com Gravação: com o Led_ERS fica aceso até ser apagada a memória;
- Apagando dados da memória: com o Led_ERS piscando durante o processo;
- Erro de recepção: o Led_erro permanece aceso durante o erro;
- Reset: com o Led_Reset aceso durante a reinicializarão do FPGA;

O botão de Reset é sinalizado pelo Select SPI, mas atua em todos os blocos do sistema Data Logger reiniciando-os.

O bloco Select SPI foi projetado para detectar o acionamento dos botões Start, Download e Erase e valida-los conforme o estado da memória. O bloco Controlador SPI, que será apresentado no próximo item é responsável por verificar o estado atual da memória passando esta informação para o bloco Select SPI através do sinal REC, se REC = ‘1’ o estado da memória é gravada e se REC = ‘0’ o estado da memória é vazia.

O bloco Select SPI ao analisar REC = ‘1’ só considera o acionamento dos botões Download e Erase, se for acionado o botão Start o bloco Select SPI o desconsidera. Já se o sinal REC = ‘0’, o bloco Select SPI só considera o acionamento do botão Start, desconsiderando os outros dois. O botão Stop só é lido pelo bloco Select SPI após a validação do botão Start, ou seja, se estiver gravando.

Ao detectar um acionamento de botão válido, o bloco Select SPI repassa está informação para o bloco Controlador SPI que é o responsável por executar as tarefas. A seguir será descrito o funcionamento do acionamento de cada botão:

Botão Start: ao ser acionado este botão, sinal write_in = ‘1’ e o sinal REC = ‘0’, o bloco Select SPI sinalizará ao bloco Controlador SPI através do sinal WR = ‘1’ que o mesmo pode iniciar o processo de gravação. O bloco Select SPI acende o led_WR indicando gravação em andamento e passa a aguardar que o botão Stop seja pressionado.

Botão Stop: quando este botão for acionado, sinal stop_in = ‘1’, o bloco Select SPI avisa para o bloco Controlador SPI através do sinal STP = ‘1’ que o controlador deve encerrar o processo de gravação. O bloco Select SPI passa a aguardar a resposta do controlador de fim

de gravação verificando quando o sinal FWR passa para ‘1’. Neste momento o led_WR de gravando é apagado, o sinal WR é setado para ‘0’ e o led_ERS de memória gravada é aceso.

Botão Download: ao ser acionado este botão, o sinal read_in = ‘1’ e o sinal REC for ‘1’, o bloco Select SPI avisa ao bloco Controlador SPI para iniciar o processo de leitura da memória, passando o sinal RD para ‘1’ e acendendo o led_RD de leitura em andamento. A partir desse momento, o bloco Select SPI fica aguardando o fim da leitura pelo bloco Controlador SPI através do sinal FRD = ‘1’. Ao receber a resposta o bloco Select SPI apaga o led de leitura e passa o sinal RD para ‘0’.

Botão Erase: ao ser acionado este botão, o sinal erase_in = ‘1’ e o sinal REC = ‘1’, o bloco Select SPI informa ao bloco Controlador SPI para iniciar o processo de apagar a memória, setando o sinal ERS para ‘1’ e o led_ERS ficará piscando. O bloco Select SPI passa a aguardar o fim do apagamento pelo bloco Controlador SPI através do sinal FERS = ‘1’. Ao receber este sinal, o bloco Select SPI apaga o led_ERS e passa o sinal ERS para ‘0’.

Ao término de cada processo de gravação, de leitura ou de apagamento, o bloco Select SPI retorna ao início verificando o estado atual da memória, sinal REC, e passa a aguardar o acionamento dos botões permitidos. O fluxograma pode ser observado na figura 28. No anexo D encontra-se a estrutura do código VHDL desse bloco.

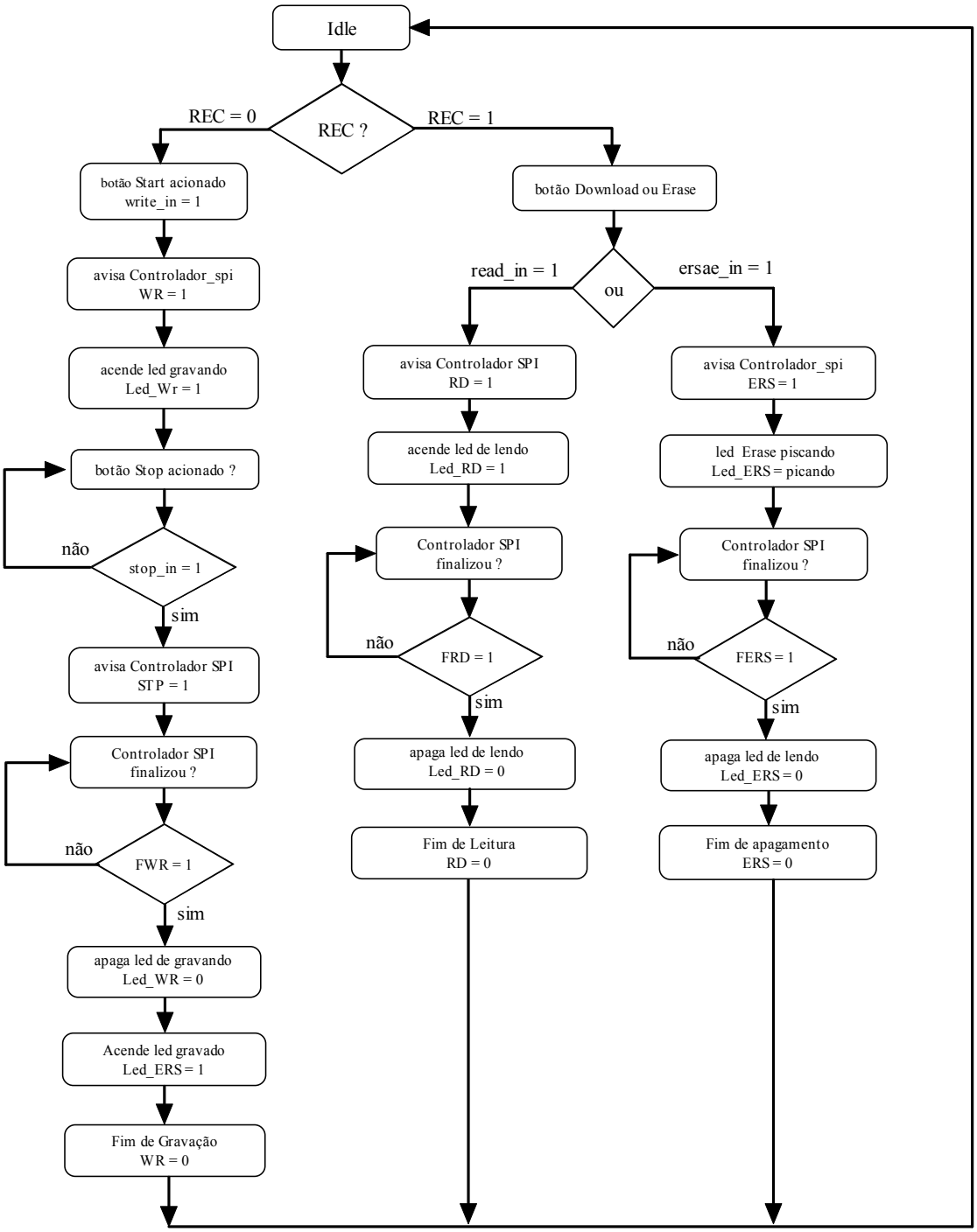


Figura 28 – Fluxograma do bloco Select SPI

Fonte: Própria

6.5.5 Controlador de SPI

O bloco Controlador SPI executa as tarefas de escrita, leitura e apagamento da memória. Na figura 29 temos a ilustração do fluxograma de uma forma macro do bloco Controlador SPI, veremos mais detalhes de cada tarefa nos itens posteriores.

Ao ser iniciado o bloco Controlador SPI, a primeira verificação é com relação ao estado da memória, representado pelo sinal REC. A partir do estado deste sinal é permitido a realização de determinadas tarefas como já mencionado no bloco Select SPI.

Conforme ilustrado na figura 29, através do valor do sinal REC poderemos fazer determinadas funções; com REC = ‘1’ ler e apagar a memória, e com REC = ‘0’ gravar os dados na memória. Os sinais WR (escrita), RD (leitura), ERS (apagar) representam os botões lidos no bloco Select SPI e os sinais FWR (fim de escrita), FRD (fim de leitura), FERS (fim de apagar) são sinais de resposta do bloco Controlador SPI para o bloco Select SPI quando a tarefa for finaliza. Após o término de cada tarefa o sinal REC pode ser atualizado, no término do processo de escrita da memória esse sinal é setado para ‘1’, no fim do processo de apagar a memória o sinal é setado para ‘0’, somente quando o processo for de leitura o estado do sinal REC se mantém ‘1’. No anexo E temos a estrutura do código VHDL desse bloco.

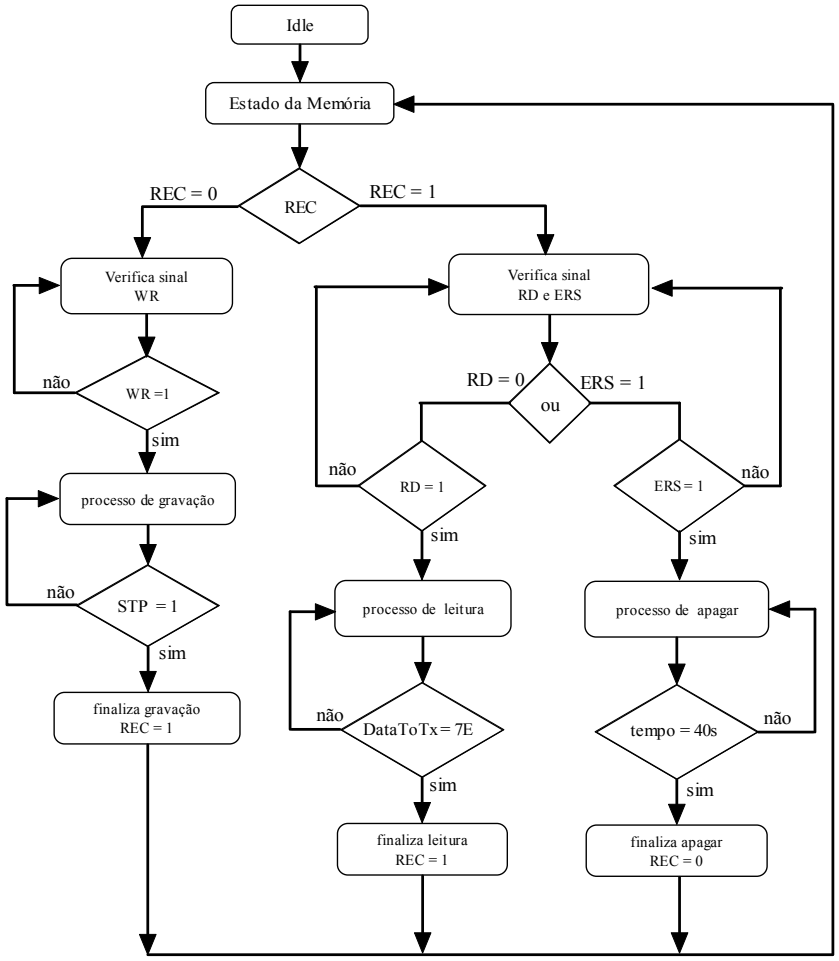


Figura 29 – Fluxograma do bloco Controlador SPI

Fonte: Própria

6.5.5.1 Processo de gravação

Quando a memória está limpa, REC = ‘0’, o bloco Controlador SPI fica aguardando a única operação possível, que é gravar novos dados na memória.

O processo de gravação se inicia ao ser detectado o sinal WR = ‘1’ (figura 30). A partir desse momento, o bloco Controlador SPI avisa através do sinal startWR = ‘’1 para o bloco Detector de Palavra iniciar a detecção da sentença desejada.

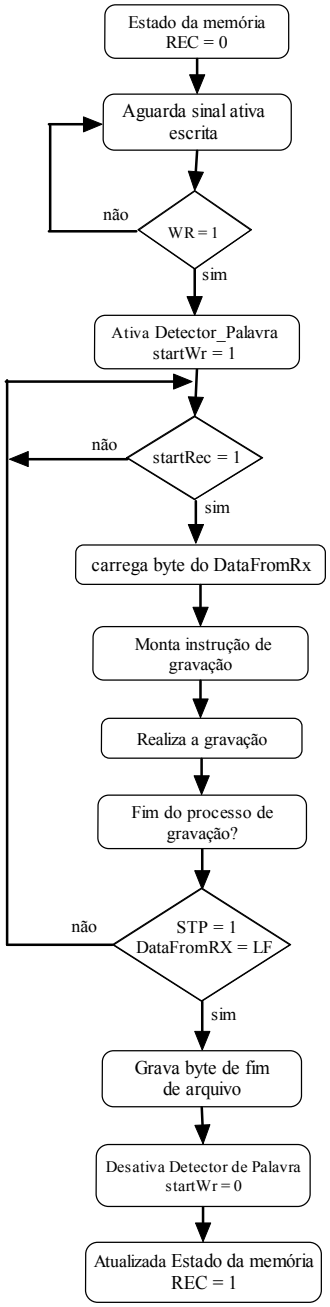


Figura 30 – Fluxograma do processo de gravação

Fonte: Própria

Quando o bloco Controlador SPI detecta que o sinal startRec está em nível alto, ele lê os dados da porta DataFromRx, caso contrario, ele fica aguardando até que este sinal fique em nível alto, a seguir é montada a instrução de gravação e enviado juntamente com o byte a ser gravado para a memória, novamente o ciclo se repete, o endereço é incrementado e novo byte é gravado.

O ciclo continua a se repetir até que seja pressionado o botão Stop. Quando isso ocorre, o bloco Controlador SPI aguarda o final da sentença determinado pelo caractere “LF”, para não gravar sentença corrompida. Em seguida o processo de gravação se encerra com a gravação de um caractere de fim de arquivo, no caso o caractere ‘~’ que equivale a 7E em hexadecimal.

O bloco Controlador SPI envia o sinal para desativar a Detecção de Palavra, sinal startWR = ‘0’, atualiza o sinal do estado de memória para REC = ‘1’, memória gravada e retorna para seu estado inicial.

Para que possa ser efetivada a gravação, a memória M25P16 exige que uma instrução de *Write Enable* tenha sido previamente executada e em seguida a instrução *Page Program*.

A sequência completa da instrução de gravação fica:

SeqInstruct_wr: WREN

Onde:

WREN : instrução de write enable;

Seguida da instrução:

SeqInstruct_wr: PP + address_wr + DataFromRx.

Onde:

PP: instrução de programação;

Address_wr: endereço na memória onde será gravado o byte;

DataFromRx: byte a ser gravado.

6.5.5.2 Processo de leitura

Este é o processo que faz com que as informações armazenadas previamente na memória possam ser descarregadas através da saída serial para um computador (figura 31).

Quando o sinal REC = ‘1’ e o sinal RD = ‘1’ o bloco Controlador SPI inicia a leitura da memória.

A instrução completa para a leitura de um byte na memória é:

SeqInstruct_rd: READ + address_rd.

Onde:

READ: Instrução de leitura;

Address_rd: endereço do byte a ser lido;

Após o envio da instrução de leitura e do endereço para a memória, ela responde com o dado (um byte). O controlador envia o dado ao bloco UART TX e passa o sinal starTx para

‘1’, avisando ao bloco que o dado está disponível e pode ser iniciado o processo de transmissão ao PC.

O bloco Controlador SPI fica aguardando que o sinal stopTx passe para ‘1’, indicando que a transmissão serial do byte foi concluída e um novo ciclo de leitura de dado na memória pode ser iniciado. A importância do sinal stopTx se deve ao fato de que a velocidade de leitura dos dados da memória ser de 12,5 MHz e a taxa de transmissão da UART TX ser de apenas 4800 bps. Ao receber a resposta de envio do UART TX o controlador verifica se o byte enviado é igual a 7E, se for, o controlador finaliza o processo de leitura, se não, incrementa o endereço e recomeça o ciclo.

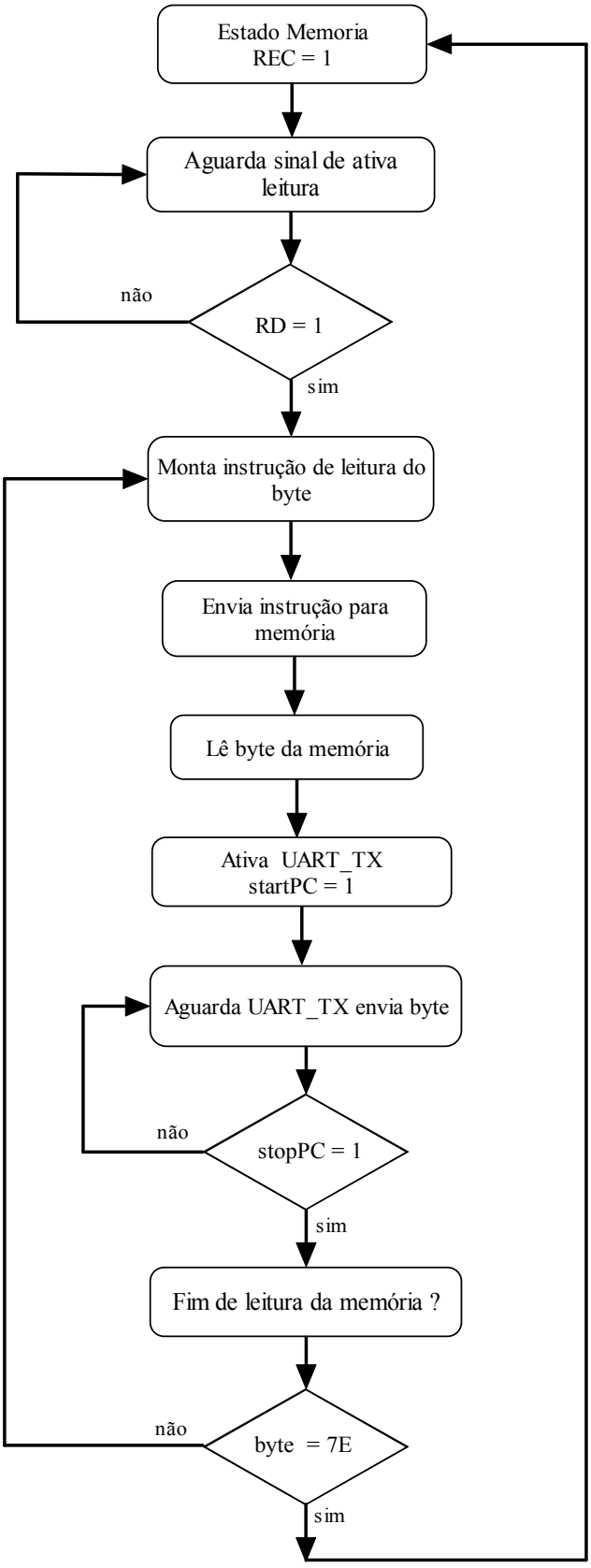


Figura 31 – Fluxograma do processo de leitura

Fonte: Própria

6.5.5.3 Processo de apagamento

O processo de apagar memória inicia quando o estado da memória esta REC = ‘1’ e o sinal ERS = ‘1’ (figura 32). A partir da detecção desses dois sinais o controlador inicia o processo de apagamento. Primeiramente monta a instrução, e em seguida a envia para memória. O bloco controlador SPI passa a aguardar o tempo de apagamento da memória que é de 40s, terminado este tempo, o bloco Controlador SPI envia o sinal FERS = ‘1’ para o bloco Select SPI, informando o final do processo de apagamento da memória e atualiza o sinal de estado da memória para REC = ‘0’.

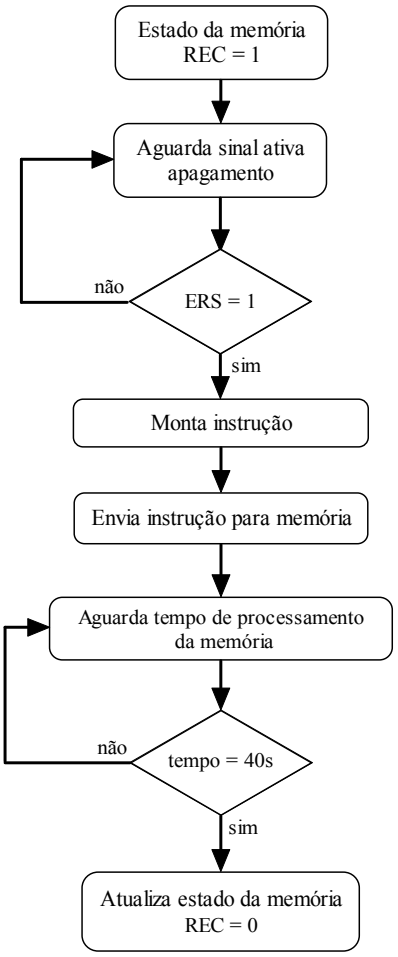


Figura 32 – Fluxograma do processo de apagamento

Fonte: Própria

6.5.6 UART TX

Este bloco implementa o transmissor da UART TX (figura 33). No estado inicial do bloco da UART TX, o sinal startTx fica sendo monitorado até que fique em nível alto. Quando esse evento ocorre, o byte presente na porta dados_tx é enviado serialmente pela saída dout_tx.

Inicialmente o transmissor gera o start bit, dout_tx = '0', para avisar o receptor do PC que uma transmissão será iniciada, em seguida é iniciada a transmissão do byte de dados a partir da transmissão do bit menos significativo D0 até o mais significativo D7 na saída dout_tx, e finaliza enviando o sinal de stop bit, dout_tx = '1', mantendo a saída em nível '1' até a próxima transmissão. O UART TX avisa ao bloco Controlador SPI através do sinal stopTx = '1' que finalizou a transmissão do byte e retorna ao início do bloco aguardando novo byte a ser enviado. No anexo F temos a estrutura do código VHDL desse bloco.

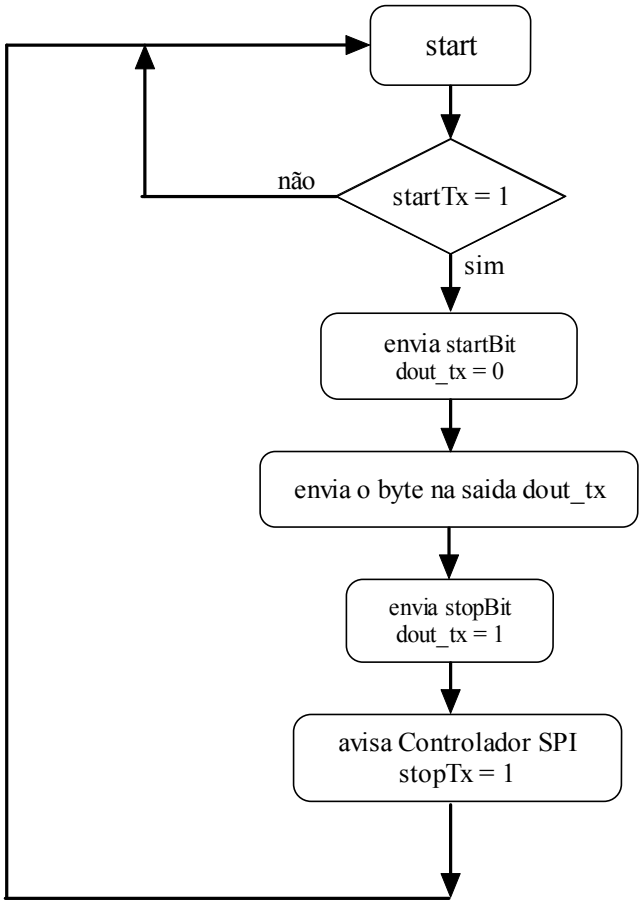


Figura 33 – Fluxograma do bloco UART TX

Fonte: Própria

6.6 Software de captura

Os dados armazenados na memória do Data Logger são transferidos ao PC pela porta serial. Para viabilizar esta transferência, foi desenvolvida uma aplicação que trata os dados adquiridos pela UART TX, gerando uma tabela com informações de hora, data, posição, velocidade e direção. Além de gerar um mapa com o percurso percorrido.

Como ferramenta de desenvolvimento desse software, foi optada pela linguagem Delphi, por possuir um ambiente de desenvolvimento amigável e com uma vasta biblioteca de componentes visuais, como por exemplo, o componente para visualizar o mapa, para criar a tabela, os botões entre outros. Além de uma forma fácil de utilizar a API do Google.

Para a comunicação serial, como o Delphi não tem nenhum componente nativo foi utilizado o pacote Async32 desenvolvido pela empresa TMS Software. Desse pacote foi utilizado o componente "VaComm" para manipulação da porta serial, onde podem ser configuradas as características da porta serial como paridade, número de stop bit entre outros.

A aplicação possui duas funções principais: comunicação com GPS e geração mapa/tabela, conforme ilustra a figura 34.

No anexo G temos a estrutura do código VHDL desse bloco.

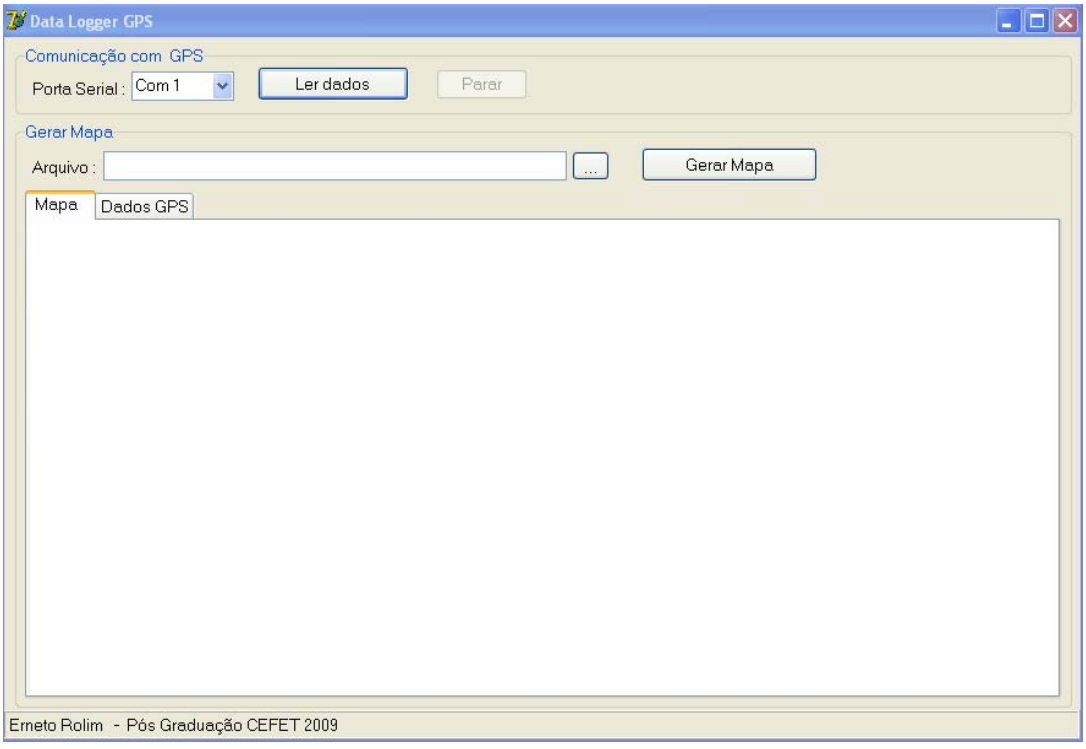


Figura 34 – Software de captura

Fonte: Própria

6.6.1 Comunicação com o GPS

A partir do momento em que o botão *Ler dados* (figura 34) for pressionado, o componente serial inicia a recepção dos dados enviados pelo Sistema Data Logger. Ao ser encerrado o processo de recebimento, os dados colhidos serão filtrados e formatados para então serem armazenados em um arquivo texto com extensão ".txt".

Na filtragem foram selecionados apenas os seguintes campos: hora, validade da posição, latitude, direção da latitude, longitude, direção da longitude, data e velocidade.

Na formatação os campos filtrados serão padronizados para a geração do mapa e visualização da tabela (figura 36). Para a geração do mapa foi necessário formatar o campo latitude e longitude enviados pelo módulo GPS para o formato compreendido pelo Google Maps. Para a tabela foi formato o campo hora (hhmmss.ss => hh:mm:ss) e data (ddmmyy => dd:mm:yy).

6.6.2 Geração do mapa

O mapa mostrará a trajetória percorrida conforme os dados recebidos do GPS, utilizando a latitude e a longitude (figura 35).

Para gerar o mapa cria-se através da ferramenta Delphi um código HTML (*HyperText Markup Language*), uma linguagem de marcação para criar página WEB, e a API do Google Maps, um conjunto de componentes para criar e manipular mapas em uma página da web. O principal componente da API do Google Maps utilizado foi o *GPolyline* que cria uma sobreposição linear no mapa, ou seja, a linha do percurso percorrido.

Visualiza-se o mapa com o percurso percorrido através do componente WebBrowser disponível no Delphi, que carrega uma página através do código HTML.

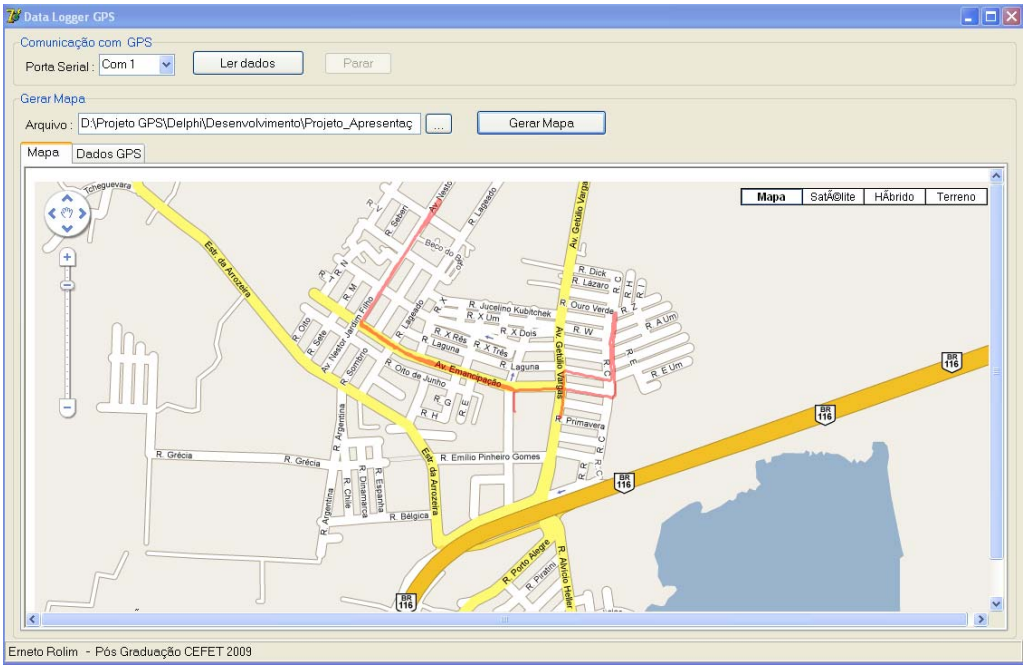
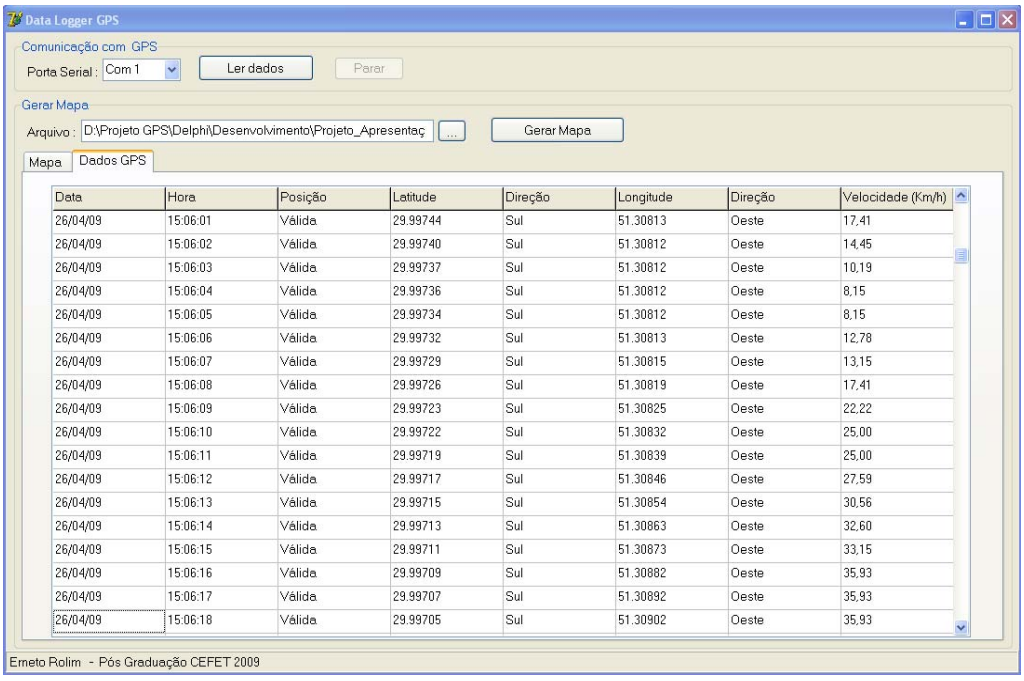


Figura 35 – Mapa do percurso

Fonte: Própria

A tabela mostrará todos os campos gravados em arquivo (figura 36).



Data	Hora	Posição	Latitude	Direção	Longitude	Direção	Velocidade (Km/h)
26/04/09	15:06:01	Válida	29.99744	Sul	51.30813	Oeste	17,41
26/04/09	15:06:02	Válida	29.99740	Sul	51.30812	Oeste	14,45
26/04/09	15:06:03	Válida	29.99737	Sul	51.30812	Oeste	10,19
26/04/09	15:06:04	Válida	29.99736	Sul	51.30812	Oeste	8,15
26/04/09	15:06:05	Válida	29.99734	Sul	51.30812	Oeste	8,15
26/04/09	15:06:06	Válida	29.99732	Sul	51.30813	Oeste	12,78
26/04/09	15:06:07	Válida	29.99729	Sul	51.30815	Oeste	13,15
26/04/09	15:06:08	Válida	29.99726	Sul	51.30819	Oeste	17,41
26/04/09	15:06:09	Válida	29.99723	Sul	51.30825	Oeste	22,22
26/04/09	15:06:10	Válida	29.99722	Sul	51.30832	Oeste	25,00
26/04/09	15:06:11	Válida	29.99719	Sul	51.30839	Oeste	25,00
26/04/09	15:06:12	Válida	29.99717	Sul	51.30846	Oeste	27,59
26/04/09	15:06:13	Válida	29.99715	Sul	51.30854	Oeste	30,56
26/04/09	15:06:14	Válida	29.99713	Sul	51.30863	Oeste	32,60
26/04/09	15:06:15	Válida	29.99711	Sul	51.30873	Oeste	33,15
26/04/09	15:06:16	Válida	29.99709	Sul	51.30882	Oeste	35,93
26/04/09	15:06:17	Válida	29.99707	Sul	51.30892	Oeste	35,93
26/04/09	15:06:18	Válida	29.99705	Sul	51.30902	Oeste	35,93

Figura 36 – Tabela com os dados do Data Logger

Fonte: Própria

7 RESULTADOS

No início do desenvolvimento do código em FPGA, foi tirado proveito do simulador da ferramenta ISE Webpack. Foram feitas simulações para todos os blocos quem compõem o Data Logger, como também para sistema completo. Somente após as simulações refletirem o sistema desejado é que o código foi efetivamente testado na prática.

Num segundo momento, foi realizado testes em bancada com o módulo GPS conectado ao Data Logger. Pelo fato do módulo estar em um ambiente fechado de uma construção, seus dados de posicionamento não eram corretos, porem serviu para verificar com o auxílio de um analisador lógico o funcionamento individual de cada bloco funcional (figura 37). Nesse caso, pode-se observar no sinal MOSI da memória M25P16 a instrução ‘03h’, que é a instrução de leitura, e no sinal MISO a memória respondendo com os dados armazenados nela.

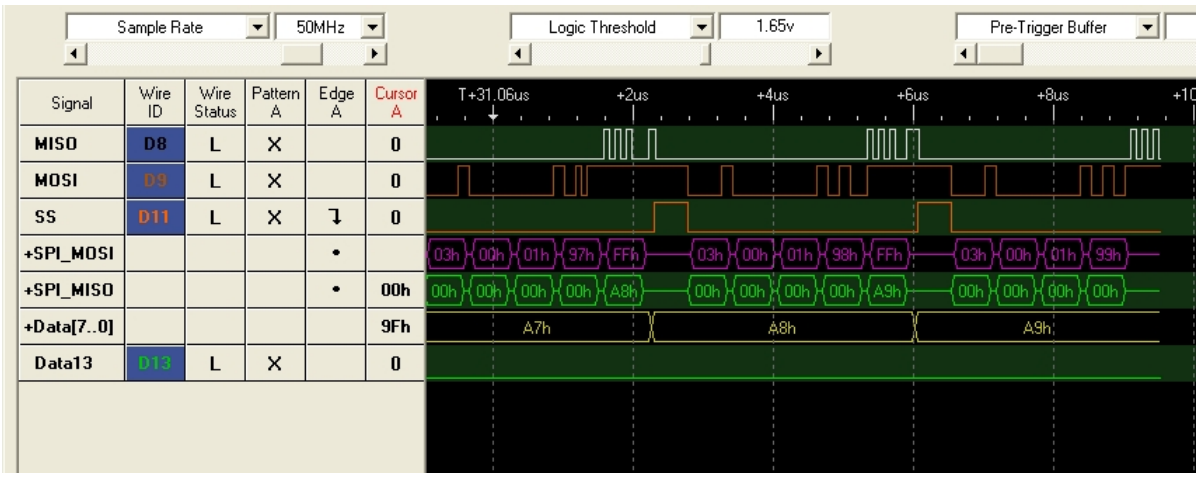


Figura 37 – Sinais capturados com um Analisador Lógico

Fonte: Própria

O teste final foi em campo. Nesse teste foi possível verificar que o sistema funcionou conforme o esperado. O percurso traçado pelo Sistema Data Logger condiz com o percurso efetuado, bem como as outras informações também retratavam a realidade do teste. O percurso efetuado pode ser observado na figura 35, o tracejado vermelho é a trajetória percorrida pelo módulo durante o teste. Na figura 36, pode-se observar a tabela gerada com os dados obtidos pelo módulo GPS, verifica-se perfeitamente que os dados são adquiridos a cada segundo, como pode ser observado na coluna “Hora”.

O desempenho geral do Data Logger foi excelente, o módulo GPS utilizado tem ótima qualidade, o que garante uma grande precisão das informações de posicionamento. A grande capacidade de memória instalada no kit, 16 Mbit, possibilita que o Data Logger armazene informações por até 10,5 horas.

8 CONCLUSÃO

Com o mercado cada vez mais ávido por novos produtos, construir plataformas que possam ser reconfiguráveis é sem dúvida uma vantagem competitiva. O projeto deste Sistema Data Logger possui essa característica. Vários recursos que a placa possui não foram implementados, como por exemplo, o display LCD, que poderia ser usado para mostrar as informações *on-line* enquanto as mesmas são adquiridas, o uso da saída VGA, ethernet entre outras.

Outro ponto interessante é que o VHDL por ser padronizado pelo IEEE (*Institute of Electrical and Electronics Engineers* – Instituto de Engenheiros Eletricistas e Eletrônicos), a migração de um fabricante para outro torna-se bastante tranquila, obviamente se não for usada nenhuma característica particular do dispositivo.

O uso de módulo OEM que fornecem as informações no formato NMEA 0183 facilita muito o trabalho, além de ser um padrão, as sentenças são enviadas no formato ASCII, facilitando a sua interpretação e tratamento.

O uso da ferramenta da Webpeck da Xilinx é bastante fácil, pois existem diversos tutoriais explicando seu funcionamento, inclusive vários em língua portuguesa. O único ponto negativo nessa ferramenta e a impossibilidade de se visualizar as variáveis utilizadas no código, obrigando o projetista a ter que usar de artifícios para obter essa informação, como por exemplo, atribuir o valor da variável a um sinal e com esse sinal o valor da variável é apresentado de forma indireta.

Como o kit que foi utilizado é fabricado pela Digilent e vendido pela própria Xilinx, não houve qualquer dificuldade de integração entre ferramenta de desenvolvimento e hardware.

Pelo fato do sistema de controle estar totalmente embarcado no FPGA, caso se desejasse construir um sistema comercial, o tamanho da placa de circuito impresso não passaria do tamanho da área de uma carta de baralho, sem dúvida um sistema bastante compacto.

ANEXOS

ANEXO A – DataLoggerGPS (Top de Hierarquia)

```
----------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    20:03:19 05/25/2009
-- Design Name:
-- Module Name:    DataLoggerGPS - Behavioral
-- Project Name:   Data Logger GPS
--
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity DataLoggerGPS is
  Port (BtReset    : in  std_logic; -- sinal de reset
   clock : in    std_logic; -- clock 50 MHz
   din_rx: in STD_LOGIC;  -- entrada de dados serial
   dout_tx: out  STD_LOGIC := '1'; -- saida de dados serial
   erro : out STD_LOGIC; -- sinalização de erro de leitura
   BtStop: in std_logic;  -- para o processo de escrever dados na
memória
   BtWrite : in std_logic;  -- inicia o processo para escrever dados na
memória
   BtRead : in std_logic;   -- inicia  o processo para ler dados da
memória
   BtErase : in std_logic;   -- inicia  o processo para apagar dados da
memória
   Led_WR : out std_logic;
   Led_RD : out std_logic;
   Led_ERS : out std_logic; -- sinaliza processo de apagamento
   Led_Reset: out std_logic;
   clkOut: out std_logic; -- clock para memoria
   SS    : out std_logic; -- chip select para a memória SPI
   MOSI   : out  std_logic;
   MISO   : in std_logic;
   DAC_CS : out STD_LOGIC := '1'; -- Desabilita DAC
   AMP_CS : out STD_LOGIC := '1'; -- desabilita pré amplificador
   AD_CONV : out STD_LOGIC := '0'; -- Desabilita ADC
   SF_CE0 : out STD_LOGIC := '1'; -- Desabilita strada flash
   FPGA_INIT_B : out STD_LOGIC := '1'  -- manter em 1   );
end DataLoggerGPS;

architecture Behavioral of DataLoggerGPS is
  component UART_RX
    Port ( clock_rx     : in  STD_LOGIC;
           din_rx       : in  STD_LOGIC;
           reset        : in  STD_LOGIC;
           dados_rx     : out STD_LOGIC_VECTOR (7 downto 0);
           erro         : out STD_LOGIC;
           dado_pronto  : out STD_LOGIC
         );
  end component;
```

```
component UART_TX
     Port ( dados_tx  : in  STD_LOGIC_VECTOR (7 downto 0);
            startTx    : in  STD_LOGIC;
            clock_tx  : in  STD_LOGIC;
            reset      : in  STD_LOGIC;
            dout_tx    : out STD_LOGIC;
            stopTx     : out STD_LOGIC
          );
end component;

component Detector_Palavra
  Port ( clock_spi   : in  STD_LOGIC;
         reset              : in  STD_LOGIC;
         palavra_rx  : in STD_LOGIC_VECTOR (7 downto 0);
         startRec     : out STD_LOGIC;
         dado_pronto : in  STD_LOGIC;
         startWR     : in  STD_LOGIC
       );
end component;

component Select_SPI
  Port ( reset        : in  std_logic;
         clock_spi  : in  std_logic;
         stop_in    : in std_logic;
         write_in   : in std_logic;
         read_in    : in std_logic;
         erase_in   : in std_logic;
         WR                 : out std_logic;
         RD         : out std_logic;
         STP        : out std_logic;
         REC        : in std_logic;
         FWR        : in std_logic;
         FRD        : in std_logic;
         ERS        : out std_logic;
         FERS       : in std_logic;
         Led_WR     : out std_logic;
         Led_RD     : out std_logic;
         Led_ERS    : out std_logic;
         Led_Reset  : out std_logic
    );
end component;

component Controlador_SPI
  Port( reset         : in  std_logic;
        clock_spi   : in  std_logic;
        clkOut        : out std_logic;
        SS            : out std_logic;
        MOSI          : out  std_logic;
        MISO          : in std_logic;
        DataFromRx  : in std_logic_vector(7 downto 0);
        DataToTx      : out std_logic_vector(7 downto 0);
        WR                 : in std_logic;
        RD          : in std_logic;
        STP         : in std_logic;
        REC         : out std_logic;
        FWR         : out std_logic;
         FRD         : out std_logic;
        ERS         : in std_logic ;
```

```
          FERS        : out std_logic;
          startRec      : in std_logic;
          startWr     : out std_logic;
          startTx       : out std_logic;
          stopTx      : in std_logic
      );
  end component;

 component Clock_Gen
   Port ( reset     : in STD_LOGIC;
            clock_in  : in  STD_LOGIC;
            clock_rx  : out  STD_LOGIC;
            clock_tx  : out  STD_LOGIC;
            clock_spi : out  STD_LOGIC
          );
  end component;



-------- sinais dos botões --------
 signal reset : std_logic;
 signal write_in : std_logic;
 signal stop_in : std_logic;
 signal read_in : std_logic;
 signal erase_in : std_logic;

-------- sinais internos --------
 signal sDados_rx    : STD_LOGIC_VECTOR (7 downto 0);
 signal sDado_pronto :  STD_LOGIC;
 signal sDados_tx      : STD_LOGIC_VECTOR (7 downto 0);
 signal sStartTX    :  STD_LOGIC;
 signal sStopTx      : std_logic;
 signal sStartRec    : std_logic;
 signal sStartWR     : std_logic;
 signal sClock_spi   : std_logic;
 signal sClock_tx      : std_logic;
 signal sClock_rx      : std_logic;
 signal sWR  : std_logic;
 signal sRD   : std_logic;
 signal sSTP  : std_logic;
 signal sREC  : std_logic;
 signal sFWR  : std_logic;
 signal sFRD  : std_logic;
 signal sERS  : std_logic;
 signal sFERS : std_logic;

begin
  reset <= BtReset;
  write_in <= BtWrite;
  stop_in  <= BtStop;
  read_in  <= BtRead;
  erase_in <= BtErase;

 DAC_CS  <= '1'; -- Desabilita DAC
 AMP_CS  <= '1'; -- desabilita pré amplificador
 SF_CE0  <= '1'; -- Desabilita strada flash
 AD_CONV <= '0'; -- Desabilita ADC
 FPGA_INIT_B <= '1'; -- manter em 1
```

```
Uart_RX0 : UART_RX
                      Port map( sClock_rx,
                                din_rx,
                                reset,
                                sDados_rx,
                                erro,
                                sDado_pronto
                                );

Uart_TX0 :  UART_TX
            Port map ( sDados_tx,
                       sStartTx,
                       sClock_tx,
                       reset,
                       dout_tx,
                       sStopTx
                      );

Detector_Palavra0 : Detector_Palavra
           Port map ( sClock_spi,
                      reset,
                      sDados_rx,
                      sStartRec,
                      sDado_pronto,
                      sStartWr
                   );

Clock_Gen0 : Clock_Gen
             Port map ( reset,
                        clock,
                        sClock_rx,
                        sClock_tx,
                        sClock_spi
                      );

Controle0  : Controlador_SPI
             Port map ( reset,
                        sClock_spi,
                        clkOut,
                        SS,
                        MOSI,
                        MISO,
                        sDados_rx,
                        sDados_tx,
                        sWR,
                        sRD,
                        sSTP,
                        sREC,
                        sFWR,
                        sFRD,
                        sERS,
                        sFERS,
                        sStartRec,
                        sStartWr,
                        sStartTx,
                        sStopTx
                   );
```

```
 Select0 : Select_SPI
           Port map ( reset ,
                  sClock_spi,
                  stop_in ,
                  write_in,
                  read_in,
                  erase_in,
                  sWR,
                  sRD,
                  sSTP,
                  sREC,
                  sFWR,
                  sFRD,
                  sERS,
                  sFERS,
                  Led_WR,
                  Led_RD,
                  Led_ERS,
                  Led_Reset
           );

end Behavioral;
```

ANEXO B – UART - RX

```
----------------------------------------------------------------------
-- Company:  CEFET - Pós Graduação
-- Engineer: Ernesto Rolim
--
-- Create Date:    18:46:55 10/20/2008
-- Module Name:    UART_RX - Behavioral
-- Project Name:   Data logger GPS
--
-- Description: Uart RX recebe sinal serial do gps, analisa e sinaliza
-- para o detector.
-- Para garantir um inicio de senteça válido aguarda-se uma sequencia
-- de 8 bits uns para então aguardar srtart bit.
-- Ao detetar o start le os 8 bits de dados e verifica o stop bit,
-- se o stop bit for 1, avisa detector para ler dado e retorna para
-- aguardar próximo start bit
-- Se stop bit for zero é sinalizado erro(led) e retorna para o estado
-- inicial.

----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity UART_RX is
    Port ( clock_rx    : in  STD_LOGIC; -- clock para ckt de amostragem
da recepção
           din_rx      : in  STD_LOGIC; -- entrada de dados serial
           reset             : in  STD_LOGIC; -- sinal de reset
           dados_rx    : out STD_LOGIC_VECTOR (7 downto 0) := x"FF"; --
saida de dados paralelos
```

```
                  erro          : out STD_LOGIC:= '0'; -- sinal de
erro de leitura
                  dado_pronto : out STD_LOGIC := '0'

                  );
end UART_RX;

architecture Behavioral of UART_RX is

-----------------------------------------------
-- cria os estados de recepção
-----------------------------------------------
type RX_state is(
                                  idle,
                                  contagem_bit,
                                  Verifica_nivel,
                                  wait_start_bit,
                                  start_bit,
                                  recebe_dado,
                                  stop_bit,
                                  byte_valido,
                                  retorna_start,
                                  falha
                                  );

signal proximo_estado : RX_state := idle;

begin

-----------------------------------------------
-- Maquina de estados
-----------------------------------------------
process (clock_rx, reset)
   variable contClock  : natural;
      variable contBit    : natural;
      variable pos_bit    : integer range 0 to 8;   -- posição do bit
dentro do byte

begin

      if (reset = '1') then
        <-- reinicializa as variaveis -->

      elsif clock_rx'event and clock_rx = '1' then
        case proximo_estado is
          when idle =>
            <-- incia ao detectar nivel 1 -->
                  . . .
                erro <= apaga;
               proximo_estado <= contagem_bit;

            when contagem_bit =>
                 <-- conta 16 vezes -->
                   . . .
                   proximo_estado <= verifica_nivel;

             when verifica_Nivel =>
             <-- verifica uma sequencia de 8 bits -->
```

```
                . . .
                proximo_estado <= wait_start_bit;

             when wait_start_bit =>
              <-- aguara o sinal de start -->
                . . .
                din_rx = 0;
                proximo_estado <= start_bit;

             when start_bit =>
               <-- Conta 8 clocks para ficar no meio do sinal start
                . . .
               proximo_estado <= recebe_dado;

            when recebe_dado =>
               <--  Recebe os 8 bits, conta 15 clocks e le sinal
                    . . .
                   proximo_estado <= stop_bit;


            when stop_bit =>
                 <-- se din_rx = '1' (stop bit)
                       . . .
                       proximo_estado <= byte_valido;
                 <-- se din_rx = '0' (falha)
                       proximo_estado <= falha;

            when byte_valido =>
              . . .
              dado_pronto <= '1';
              proximo_estado <= retorna_start;

            when retorna_start =>
              . . .
              dado_pronto <= '0';
              proximo_estado <= wait_start_bit;

            when falha =>
              <--Estado de falha, acende led por 15 clocks -->
              erro <= acende;
              . . .
              <-- retorna ao estado inicial -->
              proximo_estado <= idle;

      end case;
  end if;
 end process;
end Behavioral;
```

ANEXO C – Detector de Palavra

```
-- Company:       CEFET - Pós Graduação
-- Engineer:      Ernesto Rolim
--
-- Create Date:   18:57:31 11/26/2008
-- Module Name:   Detector_Palavra - Behavioral
-- Project Name:  Datalogger GPS
--
-- Description:  o Detector_Palavra tem o objetivo de detectar o
-- cabeçalho de sentença desejada e qdo detectado avisar ao
-- controlador para gravar as palavras da sentença até a detecção
-- de fim de sentença.
--
----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Detector_Palavra is
     Port (clock_spi   : in  STD_LOGIC;
           reset       : in  STD_LOGIC;
           palavra_rx  : in STD_LOGIC_VECTOR (7 downto 0);
           startRec    : out STD_LOGIC;
           dado_pronto : in  STD_LOGIC;
           startWR     : in  STD_LOGIC

           );

end Detector_Palavra;

architecture Behavioral of Detector_Palavra is

----------------------------------------------------------------------
-- cria os estados de detecção
----------------------------------------------------------------------
 type state_type_detector is (
                      word_Cifrao,
                       word_G,
                       word_P,
                       word_R,
                       word_M,
                       word_C,
                       word_Virg,
                       reading
                 );
 signal state_detector : state_type_detector := word_Cifrao;


begin
----------------------------------------------------------------------
Processo principal, responsavel por detectar o cabeçalho
-- desejado e quando detectado setar sinal stat_spi = 1, indicando
-- para o processo ControlStartSpi que as palavras podem ser
-- gravadas pelo controlador spi
----------------------------------------------------------------------
process (dado_pronto, reset)
  begin
```

```
if (reset = '1') then
  <-- reinicializa as varivaies -->
     state_detector <= word_cifrao;

 elsif dado_pronto'event and dado_pronto = '1' then
     case state_detector is
        when word_cifrao =>
             . . .
           <-- se byte $     -->
                . . .
                state_detector <= word_G;

         when word_G =>
               . . .
              <-- byte G -->
                . . .
              state_detector <= word_P;

         when word_P =>
            . . .
            <-- byte P -->
                . . .
               state_detector <= word_R;

         when word_R =>
            . . .
              <-- byte R  -->
                 . . .
                state_detector <= word_M;

         when word_M =>
            . . .
            <-- byte M   -->
              . . .
               state_detector <=    word_C;

         when word_C =>
            . . .
            <-- byte C -->
              . . .
             state_detector <= word_Virg;

         when word_Virg => -- desconsidera primeira virgula
            . . .
            <-- se virgula -->
                . . .
                 state_detector <= word_cifrao;

         when reading =>
            . . .
            <-- lendo os próximos bytes da sentença -->
                 . . .
                    state_detector <= reading;
            <-- se byte = LF, reinicia a detecção -->
                 . . .
                state_detector <= word_cifrao;

     end case;
```

```
       end if;
  end process;

end Behavioral;
```

ANEXO D – Select SPI

```
----------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    15:21:35 05/19/2009
-- Design Name:
-- Module Name:    Select_SPI - Behavioral
-- Project Name:   Data Logger GPS
-- Description: Este processo é responsavel por ler os botões de
-- escrita, leitura e erase:
--  -> sinalizar para o controlador_spi de qual função executar;
--  -> detectar o fim da execução de uma das funções pelo
--    contralador_spi;
--  -> controlar os leds de sinalização de escrita, leitura, erase e
--     reset.
-- Após um botão de escrita, leitura ou erase for acionado os
-- outros ficarão inativos até o final da execução, com exceção do
-- botão de reset que reinicializa o sistema.
--
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Select_SPI is
  Port (reset   : in  std_logic;-- sinal de reset
      clock_spi : in  std_logic;-- clock do gerador
      stop_in   : in std_logic;-- para o processo de escrever dados na
memória
      write_in  : in std_logic;-- inicia o processo para escrever dados
na memória
      read_in   : in std_logic; -- inicia  o processo para ler dados da
memória
      erase_in  : in std_logic;
      WR    : out std_logic := '0'; -- envia habilita escrita
      RD    : out std_logic := '0'; -- envia habilita leitura
      STP   : out std_logic := '0'; -- envia finaliza escrita
      REC   : in std_logic;  -- recebe estado memória
      FWR   : in std_logic;  -- recebe fim de escrita
      FRD   : in std_logic;  -- recebe fim de leitura
      ERS   : out std_logic := '0'; -- envia apagar memória
      FERS   : in std_logic;  -- recebe fim de apagamento mem
      Led_WR  : out std_logic := '0'; --Led de gravando em execução
      Led_RD  : out std_logic := '0'; -- Led de leitura em execução
      Led_ERS  : out std_logic := '0'; -- Led de gravado/apagando
      Led_Reset  : out std_logic := '0'  -- Led indicador de reset
 );
end Select_SPI;
```

```
architecture Behavioral of Select_SPI is

-----------------------------------------------------------------------
-- cria os estados para os botões de escrita, leitura e apagar
-----------------------------------------------------------------------
 type state_type_acao is (
                           idle,
                           writing,
                           wait_finish,
                           reading,
                            erasing
                          );
 signal state_acao : state_type_acao := idle;

begin

  process(clock_spi, reset) is
  begin
    if reset = '1' then
     <-- reinicializa as variaveis -->
 else
    if clock_spi'event and clock_spi='1' then
      case state_acao is
          when idle  => -- verifica se algum botão foi pressionado
             <-- memoria com gravação REC=1 -->
               . . .

                <-- se botão apagar acionado -->
                      . . .
                      ERS <= '1';
                      state_acao <= erasing;
                      Led_ERS <= piscando;

                      <-- se botão ler acionado -->
                            . . .
                            RD <= '1';
                            led_RD <=aceso;
                            state_acao <= reading;
                            . . .

                  <-- memoria vazia REC = 0 -->
                      . . .
                      <-- se botaõ escrita -->
                           . . .
                           WR <= '1';
                           led_WR <= aceso;
                           state_acao <= writing;

                      . . .

          when writing => -- aguarda o botão de stop ser acionado
             <-- se botão Stop foi acionado -->
                 . . .
                 STP <= '1';
                 state_acao <= wait_finish;

          when wait_finish => -- aguarda finalização de escrita
                 . . .
```

```
                <-- se FWR =  1, finalizou a escrita
                      . . .
                      led_WR <= apagado;
                      led_ERS <= aceso;
                      state_acao <= idle;

          when reading => -- aguarda sinal fim de leitura
                    . . .
                <-- se FRD = 1 , terminou de leitura
                      . . .
                      RD <= '0';
                      led_RD <= apagado;
                      state_acao <= idle;

           when erasing => -- aguarda fim do apagar
                     . . .
                 <-- se FERS = 1 , terminou de apagar
                       . . .
                       ERS <= '0';
                       led_ERS <= apagado;
                       state_acao <= idle;

        end case;
      end if;
    end if;
  end process;

end Behavioral;
```

ANEXO E – Controlador de SPI

```
----------------------------------------------------------------------
-- Company:        CEFET - Pós Graduação
-- Engineer:       Ernesto Rolim
--
-- Create Date:    17:36:13 02/10/2008
-- Module Name:    Controle_SPI_RD_WR - Behavioral
-- Project Name:   Data logger GPS
--
-- Description: O controlador tem como fuções escreve os dados
-- lidos do GPS na memoria, ler os dados gravados anteriormente na
-- memória e envia para PC e apagar os dados armazenados na
-- memória. Para executar estas funções temos os seguintes processos:
--   Principal : controla as 3 funções: de escrita, leitura e
--               apagar memória.
--   Leitura   : é ativado pelo processo principal para ler os
--               dados do pino Miso da memória.
--   AtivaSS   : é ativado pelo processo principal para libera o
--               chip select da memória
--   sendDados : é ativado pelo processo principal para enviar os
--               dados para memória no pino Mosi
----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity Controlador_SPI is
    Port( reset  : in  std_logic;-- sinal de reset
      clock_spi  : in  std_logic;-- clock do gerador
      clkOut: out std_logic;-- clock para memoria
      SS    : out std_logic;-- chip select para a memória SPI
      MOSI  : out std_logic;-- sinal de entrada de dados da memória
      MISO  : in std_logic;-- sinal de saida de dados da memória
      DataFromRx :in std_logic_vector(7 downto 0);-- dados uart rx
      DataToTx:out std_logic_vector(7 downto 0);-- dados p/ uart tx
      WR  : in std_logic;   -- recebe habilita escrita
      RD  : in std_logic;   -- recebe habilita leitura
      STP : in std_logic;   -- recebe finaliza escrita
      REC : out std_logic := 'Z';-- estado da memoria(vazia/cheia)
      FWR : out std_logic := '0'; -- fim de escrita
      FRD : out std_logic := '0'; -- fim de leitura
      ERS : in std_logic ;-- habilita apagar memória
      FERS : out std_logic := '0'; -- fim do apagar
       startRec : in std_logic; -- gravação do dado na memoria
       startWr : out std_logic := '0';-- habilta detectar a sentença
      startTx : out std_logic := '0';-- avisa Uart TX para ler da
                                          saida DataToTx
      stopTx  : in std_logic  -- sinal do Uart TX para sinalisar que
                                  o dado lido está sendo enviado
);

end Controlador_SPI;

architecture Behavioral of Controlador_SPI is

-----------------------------------------------------------------------
cria os estados de escrita, leitura e limpeza da memória
-----------------------------------------------------------------------
 type state_type_write is (
      idle, -- estados para verificar estado da memoria
      waitActionWR, -- memória vazia
                                                  waitActionRD_ERS, --
memória com gravação


      prepWREN,--estados para enviar WREN, usado pela WR e erase
      sendWREN,
      nextState,

      startWrite, -- estados para WR do process Principal
      prepPP,
      sendPP,
      WaitAnyClk,
      verifStop,
      prepStop,
      stopWrite,
      stopFinish,
      waitSelClearWR,

       startRead, -- estados para RD do process Principal
      sendReqRead,
      readingMemory,
      finishRead,
      stateMemory,
```

```
      waitStartPC,
      DownloadFinish,
      waitFinishStopPc,
     waitSelClearRD,

      prepErase, -- estados para erase
      sendErase,
      waitErase,
      waitSelClearERS
   );
 signal state_write : state_type_write := idle;


-----------------------------------------------------------------
-- estados para leitura do MISO da memória
-----------------------------------------------------------------
 type read_type_data is (
                   waiting,
                   readData,
                  readFinish
                 );
 signal read_data : read_type_data := waiting;


-----------------------------------------------------------------
-- estados para proximo estado após um WREN
-----------------------------------------------------------------
 type next_type_state is (
                  nextPP,
                  nextStop,
                  nextErase
                                  );
 signal next_state : next_type_state;

-------------------------------------------------------------------------
--
-- sinais para escrita e leitura
-------------------------------------------------------------------------
--
  <-- declaração dos sinais-->

begin
  ClkOut <= clock_spi ; -- clock da memória = clock_spi

  Principal : process (clock_spi,reset) is
      . . .
    <-- variaveis para escrita na memoria -->
      . . .
    <-- variaveis para leitura da memoria -->
       . . .

  begin
    if reset ='1' then
         . . .
           <-- seta valores iniciais das variaves e sinais -->

    else
       if clock_spi'event and clock_spi='1' then
```

```
        case state_write is
          when idle => -- verifica se memoria tem gravação
                . . .
               <-- seta variavies/sinais com valores iniciais -->
                . . .
                state_write <= startRead;

           when waitActionWR => -- memória vazia (REC = 0)
               <-- se WR = '1' inicia escrita dos dados na memória
                      FWR <= '0';
                      startWr <= '1';
                      . . .
                      state_write <= startWrite;

           when waitActionRD_ERS => -- memória grava (REC = 1)
              <-- se RD = '1' inicia leitura dos dados da memoria.
                      FRD <= '0';
                      . . .
                      state_write <= startRead;

              <-- ERS = '1' incia apagamento da memoria.
                      FERS <= '0';
                      . . .
                      state_write <= prepWREN;

           when prepWREN =>
              < -- monta solicitação de escrita
                 . . .
                 SeqInstruct_wr := WREN;
                 state_write <= sendWREN;

           when sendWREN =>
              < -- envia solicitação de escrita
                   . . .
                  state_write <= nextState;

         when nextState =>                    . . .
              <-- verifica proximo estado PP, Stop ou Erase -->
                      state_write <= prepPP;
                                  or
                      state_write <= prepStop;
                                  or
                      state_write <= prepErase;

------------------- incia a função de escrita na memoria ----------

        when startWrite =>
            <--   se startRec = '1', inicia gravação
                   . . .
                  state_write <= prepWREN;
                  next_state <= nextPP;

        when prepPP =>
          <-- monta instrução para envia um byte
              . . .
              Instrução= PP + <endereço memória> + DataFromRx
              state_write <= sendPP;
```

```
        when sendPP =>
           <-- envia instrução
                . . .
                state_write <= WaitAnyClk;


        when WaitAnyClk =>
           <-- aguarda 1ms(tempo suficiente para envio de um byte)
                 . . .
                 state_write <= verifStop;

        when verifStop =>
           <-- verifica situação de parada (botão de stop acionado
           e fim de sentença)
                 . . .
                 if (STP = '1')  and (DataFromRx = x"0A") then
                             . . .
                             state_write <= prepWREN;
                             next_state <= nextStop;
                 else
                 <-- continua processo de escrita na memória -->
                          .  .  .
                         state_write <= startWrite;


         when prepStop =>
           < -- Monta instrução de fim de escrita(7E = ~)
                 . . .
                 Instrução= PP + <endereco > + <fim de arquivo>
                 state_write <= StopWrite;

        when stopWrite =>
           <-- envia instrução
                . . .
                state_write <= StopFinish;

       when StopFinish =>
          < -- avisa SelectSPI do final do processo de escrita
                . . .
                state_write <= waitSelClearWR;
                FWR <= '1'; -- Sinal de fim de escrita
                REC <= '1'; -- sinal estado de memoria gravada

      when waitSelClearWR =>
         < -- aguarda baixar sinal WR para ir para o estado
              waitActionRD_ERS →
                   . . .
                  <-- se WR = '0', fim de escrita
                       FWR <= '0';
                       state_write <= waitActionRD_ERS;


------------------- Incia função de leitura da memória ------------

      when startRead =>
         <-- Monta Solicitação de leitura.
              ...
              Instrução= REQ_READ + <endereco memoria>;
```

```
            state_write <= sendReqRead;

      when sendReqRead => -- envia solicitação
            . . .
            state_write <= readingMemory;

      when readingMemory =>
            . . .
            send_data <= waiting;
            state_write <= finishRead;

-------- processo de leitura executando => process Leitura --------

       when finishRead =>
          <-- aguarda o fim da leitura do miso e atualiza sinal
                 para uart TX -- >
                     . . .
                     DataToTx <= (byte lido da MISO)
                     state_write <= stateMemory;

       when stateMemory =>
         < -- verifica o estado da memoria para reiniciar o ciclo
                       . . .
                 <-- se memoria vazia ==>
                    . . .
                    state_write <= waitActionWR;

                 <-- se memória gravada   -->
                     . . .
                     state_write <= waitActionRD_ERS;

      when waitStartPC =>
            . . .
            <-- aguarda tempo equivale a 3 clock_tx -->
            state_write <= DownloadFinish;


    when DownloadFinish =>
       < -- uart tx enviando dado para PC enquanto stopTx = 0
            . . .
            <-- aguarda stopPC voltar para 1 -->
            state_write <= waitFinishStopPc;


    when waitFinishStopPc =>
        <-- verifica se todos os dados já foram lidos
              . . .
            <-- se byte lido era de fim de gravação "~" -->
                FRD <= '1';  -- seta sinal de fim de leitura
                state_write <= waitSelClearRD;
                          ...
            <-- se não for ultimo byte -->
               state_write <= startRead;

    when waitSelClearRD =>
       <-- aguarda baixar sinal RD para retornar ao estado
           waitActionRD_ERS -- >
                . . .
```

```
                <-- aguarda RD = 0 -->
              state_write <= waitActionRD_ERS;


--------- Inicia a função de apagar os dados da memória -----------

   when prepErase =>
      . . .
     Instrucao = BULK_ERASE;
      state_write <= sendErase;

  when sendErase =>
     . . .
      <-- envia instrução -->
           state_write <= waitErase;

  when waitErase => -- aguarda 40s, tempo max para apagar mem
     . . .
     <-- aguarda 40s, tempo max para apagar mem -->
        FERS <= '1';  -- avisa bloco Select SPI fim de erase
        REC <= '0';   -- Atualiza estado da memória
        state_write <= waitSelClearERS;

  when waitSelClearERS =>
     . . .
     <-- aguarda ERS = 0 -->
         . . .
         FERS <= '0';
        state_write <= waitActionWR;

   end case;
  end if;
 end if;

end process Principal;

-------------------------------------------------------------------
-- Processo de leitura da memoria
-- o processo é ativado pelo processo Principal
-- o sianl MISO dever ser lido na descida do clock
-------------------------------------------------------------------
  Leitura : process (clock_spi, reset) is
    variable index_rd : natural;
  begin
    if reset = '1' then
        ...
         -- seta varivaes e sinais com valores iniciais
    else
      if clock_spi'event and clock_spi='0' then
        case read_data is
          when waiting =>
              . . .
              <-- verifica incio de leitura -->
                 . . .
                read_data <= readData;

         when readData =>
              . . .
```

```
              <-- lendo 8 bits de dados da miso da memoria -->
                  <--(recebe os 8 bits ) <= MISO -->
                  read_data <= readFinish;

          when readFinish =>
              . . .
              <-- aguadar processo principal ler o byte-->
                   . . .
                   read_data <= waiting;

         end case;
        end if;
       end if;
  end process Leitura;

-------------------------------------------------------------
-- Processo de ativa/desativa Chip Select da memória (SS)
-- SS é ativado na descida do clock para na proxima subida
-- o sinal estar liberado para a memória
-------------------------------------------------------------
  AtivaSS : process(clock_spi, reset)
  begin
     .  .  .
       if clock_spi'event and clock_spi='0' then
             . . .
         <-- Habilitar ChipSelect memória -->
              SS <= '0';
           . . .
          <-- Desabilitar ChipSelect memória -->
              SS <= '1';
         . . .

       end if;

  end process AtivaSS;

-----------------------------------------------------
-- Processo para enviar dados para memoria
-- atualiza o sinal no MOSI na descida do clock para
-- na próxima subida a memória ler o sinal
-----------------------------------------------------
  sendDados : process(clock_spi, reset)
  begin
    . . .
    if clock_spi'event and clock_spi='0' then
      . . .
         <-- envia os 8 bits para memória-->
         MOSI <= (envia os 8 bit serialmente)
      . . .
         <-- aguarda proximo envio - send_data = waiting then -->
         MOSI <= '1';
         . . .
        end if;
  end process sendDados;

 end Behavioral;
```

ANEXO F – UART - TX

```
----------------------------------------------------------------------
-- Company:       CEFET - Pós Graduação
-- Engineer:      Ernesto Rolim
--
-- Create Date:    19:54:29 03/17/2009
-- Module Name:    UART_TX - Behavioral
-- Project Name:   Data Logger GPS
--
----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity UART_TX is
    Port (
     dados_tx:in  STD_LOGIC_VECTOR (7 downto 0);
     startTx: in  STD_LOGIC; -- sinal que avisa que o dado esta
                             --disponível para transmissão
     clock_tx : in  STD_LOGIC;  -- gerador de taxa para transmissão
     reset   : in  STD_LOGIC;        -- sinal de reset
     dout_tx: out STD_LOGIC := '1'; -- saída de dados seriais
     stopTx: out STD_LOGIC := '0'  -- sinal que avisa que os dados
                                    --seriais já foram enviados
   );
end UART_TX;

architecture Behavioral of UART_TX is
---------------------------------
-- Cria os estados da transmissão
---------------------------------
type TX_state is (
                      start,
                      data,
                      stop,
                      data_sent
                 );
signal next_state : TX_state := start;

begin

---------------------------------
-- Maquina de estados
---------------------------------
  process (clock_tx, reset)

  begin
    if (reset = '1') then
           <-- reinicializa as variaveis -->

       elsif clock_tx'event and clock_tx = '1' then
           case next_state is
             when start =>
                 <-- se sinal startTx = '1' inicia transmissão
                    dout_tx <= '0'; -- sinal de startbit
                    next_state <= data;
```

```
            when data =>
               <-- envia sinais de D0 a D7
                  . . .
                  dout_tx <= dados_tx(pos_bit);
                  next_state <= stop;

            when stop =>
                <-- envia sinal de stop bit -->
                  . . .
                  dout_tx <= '1'; -- sinal stop
                  next_state <= data_sent;

            when data_sent =>
               <-- Dados enviados -->
                  . . .
                 stopTx <= '1';
                 next_state <= start;

        end case;
      end if;
  end process;
end Behavioral;
```

ANEXO G – Software de captura

```
unit fmGPSmain;
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, OleCtrls, SHDocVw, ExtCtrls, VaComm,
  VaClasses, XPMan, ComCtrls, Grids;
```

```
type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    btLerDados: TButton;
    VaComm1: TVaComm;
    Label1: TLabel;
    cbPorta: TComboBox;
    GroupBox2: TGroupBox;
    Panel1: TPanel;
    edArquivo: TEdit;
    Label2: TLabel;
    btGeraMapa: TButton;
    OpenDialog1: TOpenDialog;
    btAbrir: TButton;
    XPManifest1: TXPManifest;
    TimerFimRx: TTimer;
    PageControl1: TPageControl;
    tsMapa: TTabSheet;
    WebBrowser1: TWebBrowser;
    tbTabela: TTabSheet;
    sgDadosGPS: TStringGrid;
    btParar: TButton;
    pnRodape: TPanel;
    procedure btLerDadosClick(Sender: TObject);
    procedure btGeraMapaClick(Sender: TObject);
    procedure btAbrirClick(Sender: TObject);
    procedure VaComm1RxChar(Sender: TObject; Count: Integer);
    procedure btGeraArquviClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure TimerFimRxTimer(Sender: TObject);
    procedure VaComm1Close(Sender: TObject);
    procedure btPararClick(Sender: TObject);
  private
    leDados : TStringList;
    largura : string;
    altura  : string;
    nFimRx  : integer;
  public
    procedure MontarMapa(Arquivo: String);
    procedure GravarArquivo(grDados : TStringList);
    function FomatarCampo(nPosicao: integer; sCampo:string): string;
  end;
var
  Form1: TForm1;

implementation
uses DateUtils;
{$R *.dfm}

procedure TForm1.btLerDadosClick(Sender: TObject);
begin
  leDados := TStringList.Create;
  nFimRx := 0;
  if cbPorta.ItemIndex + 1 <> VaComm1.PortNum then
    VaComm1.PortNum := cbPorta.ItemIndex + 1;
  VaComm1.Open;
  btLerDados.Enabled := False;
```

```
  btParar.Enabled := True;
end;

procedure TForm1.btGeraMapaClick(Sender: TObject);
begin
  if trim(edArquivo.Text) <> '' then begin
    MontarMapa(edArquivo.Text);
    WebBrowser1.Navigate('file://'+
ExtractFilePath(Application.ExeName) +   'Mapa.html');
  end else
    MessageDlg('Selecione o arquivo .txt',mtWarning,[mbOK],0);
end;

procedure TForm1.MontarMapa(Arquivo: String);
 var
    slXSL: TStringList;
    key : String;
    i : integer;
    aCampos : array [1..7] of string ;
    dadosArquivo : TStringList;

  procedure MontaCampo(sCampo : string; nLin : integer);
  var
    nContVirgula,j : integer;
    sMontaCampo : string;
    nCol : integer;
    contCampo : integer;
  begin
     nContVirgula := 0;
     sMontaCampo := '';
      j := 1 ;

     //**separa os campos, colocando cada um e uma posição do array
aCampos
      while nContVirgula < 7 do begin
        if  sCampo[j] = ';' then begin
          inc(nContVirgula);
          aCampos [nContVirgula] := sMontaCampo;
          sMontaCampo := '';
        end else
          sMontaCampo:= sMontaCampo + sCampo[j];
        inc(j);
      end;

       //Monta tabela dados GPS
      for nCol := 0 to 6 do begin
        contCampo := nCol+1;
        case contCampo of
          1, 3, 5: sgDadosGPS.Cells[nCol,nLin+1] := aCampos[contCampo];
          2: begin
              if trim(aCampos[contCampo]) = 'A' then
                sgDadosGPS.Cells[nCol,nLin+1] := 'Válida'
              else
                sgDadosGPS.Cells[nCol,nLin+1] := 'Inválida';
             end;
          4: begin
               if trim(aCampos[contCampo]) = 'S' then
                 sgDadosGPS.Cells[nCol,nLin+1] := 'Sul'
```

```
              else
                sgDadosGPS.Cells[nCol,nLin+1] := 'Norte';
            end;
          6: begin
               if trim(aCampos[contCampo]) = 'W' then
                 sgDadosGPS.Cells[nCol,nLin+1] := 'Oeste'
               else
                 sgDadosGPS.Cells[nCol,nLin+1] := 'leste';
            end;
          7 : begin
                if trim(aCampos[2]) = 'A' then
                  sgDadosGPS.Cells[nCol,nLin+1] := aCampos[contCampo]
                else
                  sgDadosGPS.Cells[nCol,nLin+1] := '--';
              end;
        end;
      end;

      //** coloca sinal para ser utilizado na geração do mapa
      if aCampos[4] = 'S' then // se for S-sul o campo latide é
negativa
         aCampos[3] := '-' + aCampos[3];

      if aCampos[6] = 'W' then // se for W-Oeste o campo longitute é
negativa
         aCampos[5] := '-' + aCampos[5];
  end;

begin
  try
    dadosArquivo := TStringList.Create;
    dadosArquivo.LoadFromFile(Arquivo);

    //** define o num de linha do grid conforme o tamanho de linhas do
arquivo
    sgDadosGPS.RowCount := dadosArquivo.Count - 1;

    //** verifica se existe o arquivo mapa.html no diretorio, caso
exista deleta o mesmo}
    if FileExists(ExtractFilePath(Application.ExeName) + 'Mapa.html')
then
       DeleteFile(ExtractFilePath(Application.ExeName) + 'Mapa.html');

    //**  key é a chave que você vai gerar no site do google maps
   //**  http://code.google.com/apis/maps/signup.html
    key := 'ABQIAAAAzr2EBOXUKnm_jVnk0OJI7xSosDVG8KKPE1-
m51RBrvYughuyMxQ-
               i1QfUnH94QxWIa6N4U6MouMmBA';

    //* cria a StringList, e atribui a mesma as informações para gerar
o arquivo Mapa.Html
    slXSL := TStringList.Create;
    slXSL.Add('<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"
                          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">');
    slXSL.Add('<html xmlns="http://www.w3.org/1999/xhtml">');
    slXSL.Add('<head>');
```

```
    slXSL.Add('<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />');
    slXSL.Add('<title>Data Logger GPS - Percurso</title>');
    slXSL.Add('<script
src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=' +
                          QuotedStr(Key) +
'"type="text/javascript"></script>');
    slXSL.Add('<script
src="http://maps.google.com/maps?file=api&amp;v=2"
                          type="text/javascript"></script>');
    slXSL.Add('<script type="text/javascript">');

    slXSL.Add(' function initialize() {  ');
    slXSL.Add('      if (GBrowserIsCompatible()) {  ');
    slXSL.Add('        var map = new
GMap2(document.getElementById("map_canvas"),
                              { size: new GSize( '+ largura + ',' +
altura + ')  }); ');
    slXSL.Add('        map.setCenter(new GLatLng(-29.99819, -51.30810),
15);  ');  //Original
    slXSL.Add('        map.setUIToDefault(); ');

    //** Início percurso
    slXSL.Add('var polyline = new GPolyline([ ');
    for i := 0 to dadosArquivo.Count-1 do begin
      MontaCampo(dadosArquivo[i],i);

      slXSL.Add('             new GLatLng(' + aCampos[3] + ', ' +
aCampos[5] + ' ), ');
      if i =  dadosArquivo.Count-1 then
         slXSL.Add('          new GLatLng(' + aCampos[3] + ', ' +
aCampos[5] + ' ) ')
      else
        slXSL.Add('      new GLatLng(' + aCampos[3] + ', ' + aCampos[5]
+ ' ), ');
    end; //** fim for
        slXSL.Add('    ], "#ff0000", 3); ');
        slXSL.Add('    map.addOverlay(polyline); ');
    //** Fim percurso

    slXSL.Add('      }  ');
    slXSL.Add('    }  ');
    slXSL.Add('    </script>  ');
    slXSL.Add('  </head>  ');
    slXSL.Add('  <body onload="initialize()" onunload="GUnload()">  ');
    slXSL.Add('    <div id="map_canvas" style="width: 500px; height:
300px"></div>  ');
    slXSL.Add('  </body>  ');
    slXSL.Add('</html>  ');

    //** salva o arquivo Mapa.html no diretorio da Aplicação. **//
     slXSL.SaveToFile(ExtractFilePath(Application.ExeName) +
'Mapa.html');

  finally
    FreeAndNil(slXSL);
    FreeAndNil(dadosArquivo);
  end;
```

```
end;

procedure TForm1.btAbrirClick(Sender: TObject);
begin
  OpenDialog1.InitialDir := ExtractFilePath(Application.ExeName) +
'Arquivos';
  if OpenDialog1.execute then
  begin
    edArquivo.Text := opendialog1.filename;
  end;
end;

procedure TForm1.VaComm1RxChar(Sender: TObject; Count: Integer);
var
  dadosRX : string;
begin

  //** le dados recebidos pela serial
  dadosRX := VaComm1.ReadText;

  //** cada sentença será colocada em uma linha, pois serial recebe CR
e LF
  leDados.Text := leDados.Text + dadosRX;

  //** Habilita Timmer para verificar fim de recebimento
  TimerFimRx.Enabled := true;
end;

//*********************************************************************
*
//**gera arquivo com os dados recebidos da serial
//** campos:  Hora ,posiçãoVálida , Latitude , N/S, Longitude , E/W,
velocidade
//*********************************************************************
*
procedure TForm1.GravarArquivo(grDados: TStringList);
var
  gravarDadosArquivo : TStringList;
  dadosLidosSerial : TStringList;
  sRegistro : string;
  sMontaRegistro : String;
  sNovoRegistro : string;
  sData : string;
  nContRegistros : integer;
  nContCampo : integer;
  nContVirgula : integer;
begin
  dadosLidosSerial := TStringList.Create;
  gravarDadosArquivo := TStringList.Create;

  try
    dadosLidosSerial := grDados;
    nContRegistros:= 0;

    while nContRegistros <= dadosLidosSerial.Count-1 do begin
      sRegistro := dadosLidosSerial[nContRegistros];
      nContCampo:= 0;
      nContVirgula := 0;
```

```
      sNovoRegistro := '';
      sMontaRegistro := '';
      while (nContVirgula < 7) do begin
        if  sRegistro[nContCampo] = ',' then begin
          inc(nContVirgula);
          sNovoRegistro := sNovoRegistro + FomatarCampo(nContVirgula,
sMontaRegistro) + ';';
          sMontaRegistro := '';
        end else
          sMontaRegistro := sMontaRegistro + sRegistro[ncontCampo];
       inc(nContCampo);
      end; //while
      gravarDadosArquivo.Add(sNovoRegistro);
      inc(nContRegistros);
    end; //end while

    //** nome do arquivo data logger com  data e hora
    sData := IntToStr(DayOf(now)) + '-' + IntToStr(MonthOf(now)) + '-'
+
                  IntToStr(YearOf(now)) + '_' +  IntToStr(HourOf(now))+
'h-' +
                  IntToStr(MinuteOf(now)) + 'm-' +
IntToStr(SecondOf(now)) + 's';

    gravarDadosArquivo.SaveToFile(ExtractFilePath(Application.ExeName)+

'Arquivos\DataLogger_' + sData + '.txt');
finally
    dadosLidosSerial.Free;
    gravarDadosArquivo.Free;
  end;
end;
procedure TForm1.FormResize(Sender: TObject);
begin
   { Form1 está maximizado }
   if IsZoomed(Form1.Handle) then begin
     largura := '1211';
        altura := '570';
   end else begin
     largura := '1060';
      altura := '506';
   end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
   largura := '506';
   altura := '1060';

   sgDadosGPS.Cells[0,0]:= 'Hora';
   sgDadosGPS.Cells[1,0]:= 'Posição';
   sgDadosGPS.Cells[2,0]:= 'Latitude';
   sgDadosGPS.Cells[3,0]:= 'Direção';
   sgDadosGPS.Cells[4,0]:= 'Longitude';
   sgDadosGPS.Cells[5,0]:= 'Direção';
   sgDadosGPS.Cells[6,0]:= 'Velocidade (Km/h)';

  btLerDados.Enabled := True;
```

```
  btParar.Enabled    := False;

  pnRodape.Caption := ' Erneto Rolim   -  Pós Graduação CEFET 2009'
end;

procedure TForm1.TimerFimRxTimer(Sender: TObject);
begin
   if nFimRx < leDados.Count then
     nFimRx := leDados.Count

  //** Considera que parou rx quando não aumentar mais o número de
linhas LeDados
   else begin
     TimerFimRx.Enabled := false;
     VaComm1.Close; //neste evento é gravado os dados em arquivo
   end;
end;

//** Este evento é acionado pelo TimerFimRx qdo detectar fim de
recepção ou
//** pela ação do botão parar
procedure TForm1.VaComm1Close(Sender: TObject);
begin
   nFimRx := 0;
   GravarArquivo(leDados);
   btLerDados.Enabled := true;
end;


//*********************************************************************
//** formata os dados recebidos do GPS
//*********************************************************************
function TForm1.FomatarCampo(nPosicao :integer; sCampo: string):
string;
var
  sFormata : string;
  nCampo : integer;
  sCampotemp : string;
  lat_dd : double;
  lat_mm : double;
  latitude : string;
  long_dd : double;
  long_mm : double;
  longitude : string;
  velocidade : double;
  hora : integer;

begin
  case nPosicao of
    1: begin  //** hhmmss.ss => hh:mm:ss
         sCampotemp := trim(sCampo);
         hora := StrToInt(copy(sCampotemp,1,2)) - 3; // de UTC para
hora Brasil
         sFormata := IntToStr(hora) +':'+ copy(sCampotemp,3,2) +':'+
copy(sCampotemp,5,2);
       end;
    2: begin  //** A- Válido ou V- Inválido
         sFormata := trim(sCampo);
```

```
       end;
    3: begin  //** Lat GPS dddmm.mmmm  => ddd.mmmmm [formato para usar
no google map]
         nCampo := Length(sCampo) - 7; //**  7= mm.mmmm e ncampo é d
que pode variar 1 a 3
         lat_dd := StrToFloat(copy(sCampo,1,nCampo));
         sCampotemp := copy( sCampo,nCampo+1,2) + ',' + copy(
sCampo,nCampo+4,4);
         lat_mm := strToFloat(sCampotemp);
         latitude := FloatToStrf(lat_dd + ( lat_mm/60),ffFixed,8,5);
         nCampo := Length(latitude) - 6;    //** 6 é length (.xxxxx) e
nCampo que varia de 1 a 180
         sFormata := copy(latitude,1,nCampo) + '.' +
copy(latitude,nCampo+2,5);

       end;
    4: begin //** N ou S
          sFormata := Trim(sCampo);
       end;
    5: begin  //** Longitude dddmm.mmmm => ddd.mmmmm [formato para usar
no google map]

         //** 7 é  length(mm.mmmm) e ncampo é ddd que pode variar 1 a 3
(0 a 180 graus)
         nCampo := Length(sCampo) - 7;
         long_dd := StrToFloat(copy(sCampo,1,nCampo));

         //**  filtra mm.mmmm e coloca ',' no lugar de '.' para fazer o
cálculo
         sCampotemp := copy( sCampo,nCampo+1,2) + ',' + copy(
sCampo,nCampo+4,4);
         long_mm := strToFloat( sCampotemp);
         longitude := FloatToStrf(long_dd + ( long_mm/60),ffFixed,8,5);

         //**  nCampo varia de 1 a 180 e 6 é length(.xxxxx)
         nCampo := Length(longitude) - 6;

         //** troca ',' por '.', para poder ser interpretado pelo
google map
         sFormata :=  copy(longitude,1,nCampo) + '.' +
copy(longitude,nCampo+2,5);
       end;
    6: begin  //**  E-Leste / W-Oeste
         sFormata := Trim(sCampo);
       end;
    7: begin  // **  transformar graus para km/h
          velocidade := strToFloat( copy(sCampo,1,3) + ',' +
copy(sCampo,5,1));
          sFormata := FloatToStrf((velocidade * 1.852),ffFixed,5,2);
       end;
   end;
   result := sFormata;
end;

//** aborta o processo de recebimento de dados
procedure TForm1.btPararClick(Sender: TObject);
begin
  TimerFimRx.Enabled := false;
```

```
  btParar.Enabled := false;
  btLerDados.Enabled := true;
  VaComm1.Close;
end;

end.
```

BIBLIOGRAFIA

[1] ASHENDEN Peter J. **The Designer's Guide to VHDL.** San Francisco: Morgan Kaufmann Publishers, 2002. ISBN 1-55860-674-2.

[2] CHU Pong P. **FGPA Prototyping by Examples** Xilinx Spartan 3 Version. New Jersy: John Wiley & Sons, Inc, 2008. ISBN 978-0-470-18531-5.

[2] PEDRONI Volnei A. **Circuit Design with VHDL.** Massachusetts: MIT Press, 2004. ISBN 0-262-16224-5.

[3] BROWN, S; VRANESIC, Z. **Fundamentals of Digital Logic with VHDL Design.** 2.ed. New York: Mc Graw Hill, 2005. ISBN 0-07-246085-7.

[4] McNAMARA, Joel **GPS for Dummies.** Indianápolis: Wiley Publishing, Inc., 2004. ISBN 0-7645-6933-3.

[5] EL-RABBANY, Ahmed **Introduction to GPS** The Global Positioning System. Norwood: Artech House, Inc., 2002. ISBN 1-58053-183-0.

[6] FRENCH, Gregory T. **Understanding the GPS.** Bethesda: GeoResearch, Inc., 1996. ISBN 0-9655723-O-7.

[7] FERNANDES, José Ricardo H. **Usando Google Maps na Aplicação Delph 7**. Disponível em: http://www.devmedia.com.br/articles/viewcomp_forprint.asp?comp=5540. Acesso em 25 de fevereiro de 2009.

[8] GANZ, Marcos Venicios **Traçando mapas ponto a ponto utilização API do Google Maps.** Disponível em: http://www.linhadecodigo.com.br/artigoimpressao.aspx?id=1966. Acesso em 25 de fevereiro de 2009.

[9] **API do Google Maps.** Disponível em: http://code.google.com/intl/pt-br/apis/maps/documentation/index.html. Acesso em 25 de fevereiro de 2009.

[10] What is Trilateration? Disponível em: http://www.roseindia.net/technology/gps/what-is-trilateration.shtml. Acesso em 25 de fevereiro de 2009.

[11] HORTA, Edson Lemos **Projeto de Sistemas Digitais Utilizando FPGAs.** Disponível em : http://www.pucsp.br/~ehorta/Apost-PUC-CD2007_Parte1_v100.pdf. Acesso em 25 de fevereiro de 2009.